Lab #5: Full Closed Loop Control of a DC Motor Position

ECE-121 Introduction to Microcontrollers

University of California Santa Cruz — Fall 2019

Additional Required Parts

You will again be using the USB connection between your Uno32 kit and the *ECE121 Console* application on the lab PCs. You will be using the AMS rotary encoder in your lab kit, and your previously assembled DC motor test stand. You will also be using the H-bridge provided in your lab kit, and will need some associated wires to connect your H-bridge to the micro (as you did in Lab 4).

You will be adding a pair of phototransitors to the head and running them into the ADC pins of you Uno32. The phototransistors are provided in your lab kit, but you will need some resistors (and possibly capacitors) and some hookup wire to get them working the way you need.

You will need to solder wires onto the phototransistors and hot glue them during the assembly of the motor head. And ensure that the wires are long enough that you can do a full 360 degree rotation without pulling them free from your Uno32.

Introduction

One of the primary tasks that Microcontrollers are used for is to control some device or process (hence, the micro***controller*** moniker). In the previous lab, you controlled the rate of the DC motor using a PID loop (with $K_d = 0$). In this lab, you are going to be tracking position of the drive head (above the gearhead). You will also be operating directly on a sensor (two phototransistors) that give you a directional signal.

This lab will combine many of the libraries that you have previously developed, and will (hopefully) not require all that much new code. We will, again, walk you through the steps to get to the final application.

Note here that this is **NOT** a controls class, and you are not at all expected to have any prior knowledge about how to implement the control loops. That will all be provided to you, and explained in detail in this (or the previous) lab manual.[1]

This lab will divide itself into several parts that will be solved sequentially. To borrow Blue Origin's motto: *Gradatim Ferociter*[2], you will be developing and testing each required subpart before integrating them into a coherent whole.

[1]For those of you who have taken some of the controls classes, understand that there are many different and more sophisticated ways of controlling the position of the motorhead; however, you must first master the simpler methods (e.g.: PID) before going onto the more sophisticated ones (successive loop closure, digital design, etc.).

[2]Meaning "ferociously incremental" in Latin

The lab will divide into sections that include:

- Reading of position of the motorhead (accounting for rollovers)
- Storing and retrieving PID gains in NVM
- Closed Loop Drive (PID) of the motorhead using accumulated angle
- Analog Sensor input
- Bandpass Digital Filtering of analog sensors
- Closed loop tracking of IR source

As can be seen from the short list above, this lab will require several different subsystems on the micro and they will need to be integrated so that they all work seamlessly together. You have already written all of the libraries that you need, so hopefully very little modification is required. Again, approach this incrementally and make a plan of how you will get each subsection working and tested. Once there, you can work on putting them together.

Note that the order that we have listed the sections is not necessarily the required order, but one that made sense to us at the time. Some of you may decide to not do the sections sequentially, which is completely fine as long as you finish the requirements.

> **Info:** The strategy of incremental development is one which we will continually emphasize and reinforce through this and many other classes in your engineering education. Much work has been done analyzing project failures, and the most common problem is that systems integration took too long or did not work because the component parts had not been tested together. While at first glance, incremental development appears to be slow, it is–in fact–the fastest way to accomplish the project. This is especially true as complexity grows and more people are involved. Build (code) a little, test a little, repeat. Break things up into smaller, manageable pieces, and get those working and tested. This is our last chance to emphasize it, so consider yourselves *ad nauseum*'ed.

1 Sensing of Motor Head position

All control applications begin with sensing the thing to be controlled. Without a measurement of what you desire, it is difficult to form an error that should be driven to zero. In Lab 2, you set up the AMS encoder in PWM mode and used the input capture to generate angle (which was then sent as a command to the RC servo). In Lab 4, you used the AMS encoder in SPI mode and used subsequent measurements to determine motor rate (using two points and thresholding for speed).

In this lab, we will be tracking the position of the motor head (above the gearbox) in absolute terms. However, the sensor (the encoder) is attached to the motor end below the gearbox, thus the gear ratio is a large factor. Because the motor used in this class has a very large reduction planetary gear (from the datasheet 84:1), the encoder will spin many revolutions (84) per revolution of the motor output shaft.

This means that you will need to handle the rollover of the encoder to keep track of the angle of the output shaft (top of the motor). There are two fundamental ways to keep track of the angle with the multiple rollovers: (1) Integrate the rate, and (2) use the position directly and use the rate to handle the rollover direction.

Simply integrating the rate has the advantage of simplicity. You simply have an accumulator (initially set to 0), and at every time step, you add the (signed) rate to the accumulator. This will allow you to handle a count up and down to a full int range. The disadvantage is that the rate calculation is inherently noisy and you are adding noise at each measurement.[3] It also seems just a bit silly to take position data, difference it to get rate, and then integrate the rate to get back to position.

The direct position has less noise on it than the integrated rate, but the rollovers are more tricky to handle; and with an 84:1 gearhead, you are going to need to handle them all the time. The essential characteristic of the rollover is that the subtraction of previous from current switches sign; that is the *velocity* shows the wrong sign. This should be very familiar as you handled it previously using a velocity thresholding in Lab 4, Alg. 1.

There are many ways to deal with the rollover problem, and each impose problems on the position/velocity calculation. The simplest method (again) is to consider a threshold that the velocity cannot exceed; if you are over that threshold then you have the wrong direction indicated. Using the calculation how many counts you could possibly get at the highest speed available on your motor with your power supply, again we `#define MAX_RATE`. Again, define the encoder rollover $(1 \ll 14)$ as `ENCODER_ROLLOVER` and assume you are already performing this calculation at the desired tick rate.

The algorithm is such that when your velocity has exceeded `MAX_RATE` then you have rolled over and need to subtract `ENCODER_ROLLOVER` from the accumulated angle. Likewise, when your velocity is less than $-$`MAX_RATE` you have rolled over and need to add `ENCODER_ROLLOVER` to the accumulated angle.

The pseudo-code for this is shown in Alg. 1; the `ENCODER_ROLLOVER`$= (1 \ll 14)$ from the AMS encoder which reports absolute angle to 14 bits. The output of the algorithm is the total angle ($\Theta_{\texttt{total}}$) which is the angle of the *output shaft* accounting for the rollovers. Internal to the algorithm, a static int ($n\Theta_{accumulated}$) keeps track of the number of rollovers you have experienced.

Algorithm 1: `Accumulated Angle`

```
Input: $\Theta_{current}$, current position read from encoder
Result: $\Theta_{total}$, encoder counts including rollovers
Require: static $\Theta_{previous} \leftarrow 0$, $n\Theta_{accumulated} \leftarrow 0$;

$\omega \leftarrow \Theta_{current} - \Theta_{previous}$;                    // number of counts
$\Theta_{previous} \leftarrow \Theta_{current}$;                             // update previous
if $\omega >$ MAX_RATE then
    $\omega \leftarrow \omega -$ ENCODER_ROLLOVER;                            // negative rollover
    $n\Theta_{accumulated} \leftarrow n\Theta_{accumulated} -$ ENCODER_ROLLOVER;
end
if $\omega < -$MAX_RATE then
    $\omega \leftarrow \omega +$ ENCODER_ROLLOVER;                            // positive rollover
    $n\Theta_{accumulated} \leftarrow n\Theta_{accumulated} +$ ENCODER_ROLLOVER;
end

$\Theta_{total} = \Theta_{current} + n\Theta_{accumulated}$;
return ($\Theta_{total}$) ;
```

Using the *ECE121 Console*, you should code up your algorithm for keeping track of the accumulated angle in a main.c file. Don't flood the bus; calculate the accumulated angle (and speed) at 1Khz, but only report back to the console at 200Hz. Under the "Lab 5" tab, "Lab5 Preflight" is a panel with the top section reporting back Absolute Position. Your code should send an `ID_ENCODER_ABS` message to the console, which will be displayed.

[3]When you take a difference of two noisy numbers, the noise adds. If you can make the assumption that the noise is white (Gaussian) and zero mean, then integration is the ultimate low-pass filter and this should be fine in the long term, but will still be noisy in the short term. Before doing this, you would need to gather enough data to make a statistical analysis of the noise.

Warning: You will need to insert a delay at startup of approximately 5-10msec before the readings on the encoder are stable. Until it has had time to self-calibrate, the position will be quite far off. Insert this delay *before* the `while(1)` part of your code where you return the measurement back at 200Hz.

Use your variable power supply hooked up to the motor directly (bypassing the H-bridge). Turn the motor slowly in one direction, and verify that the count is indeed increasing or decreasing as appropriate; make sure that it is accounting for rollovers appropriately. Reverse the polarity on the motor and make sure you are still getting the correct behavior.

2 Closed Loop Control of DC Motor Position

Now that you have a measure of total angle (or if you will a scaled angle of the output shaft), we are going to use your same PID loop from Lab 4 but this time to control position.

There are a few things to consider about the control algorithm in this application which are different from the rate control PI loop you did in Lab 4, the first of which is that you are going to store and retrieve your PID loop gains from the non-volatile memory (using the code you developed in Lab 3).

Note that you will want to store the gains associated with each control mode (command and sensor) and retrieve them whenever you switch modes. This will become clear in later sections.

2.1 Storing Gains in NVM

In Lab 4, the gains were set from the *ECE121 Console*, and used in the internal calculation of the PID loop (in the `FeedbackControl_Update()` function). In this lab, those gains should be stored in the non-volatile memory every time they are updated from the console. Likewise, they should be retrieved from the NVM on startup. One of the key things we are going to have you demonstrate to the TAs is the ability to restart the processor and still have the control loop run.

You application code should respond to an `ID_FEEDBACK_SET_GAINS` message by correctly interpreting the PID gains (as you did in Lab 4), store those gains at some convenient location in the NVM, and respond with an `ID_FEEDBACK_SET_GAINS_RESP` message.

Warning: You are going to have to decide at which address in the NVM to place your gains. You will need room for six `int`s for the gains (three each for two modes). These locations are going to be set using `#define`'s in your application code. Remember that you will also need room to store the FIR filter weights for the two ADC channels that we will be using in Sec. 4.1.

On startup, your application code should read the NVM to set the gains using their stored values. The *ECE121 Console* has a button on the Lab5 Application tab to both *set* the gains, and also to *get* the gains. When the "Get Gains" button is pushed, the console will send an `ID_FEEDBACK_REQ_GAINS` message. You code needs to respond with an `ID_FEEDBACK_CUR_GAINS` message, with the payload correctly formatted and in the right endian-ness.

Setting the gains using the console and the getting them will demonstrate that you have the message responses correctly. Resetting the micro in between will demonstrate that you are correctly writing and reading to NVM.

2.2 Closed Loop PID Control of Motor Position

With the absolute angle being tracked (Sec. 1), and the gains being set and retrieved from the NVM (Sec. 2.1), you are ready to implement closed loop control of the position of the motorhead using PID. For this part of the control, we will be using the encoder as the sensor (using your total angle) and injecting a desired target, `ID_COMMANDED_POSITION`, from the *ECE121 Console*.

Info: There are other methods of controlling the position of the output shaft given the encoder than the PID algorithm. They are beyond the scope of this class, but a description of the most common alternative, bang-bang control, is outlined in Appendix A for the interested student.

There are two distinct modes for the PID control loop to operate in: `COMMAND_MODE` (0) and `SENSOR_MODE` (1). These need to be `#define`'d in your code. The command mode indicates that you are using the absolute angle from the encoder and commanding the motor head to a specified position. Sensor mode will be driven off of the two IR phototransistors (see Sec. 4).

Note that because of the characteristics of the encoder and the IR phototransistors, your gains are going to be *very* different. As such, you should have two sets of gains, each corresponding to the different modes. When the mode is switched, your code should retrieve the appropriate gains and also send an `ID_FEEDBACK_CUR_GAINS` message to the console.

The mode should be set to `COMMAND_MODE` by default at startup, and is changed from the console using the `ID_LAB5_SET_MODE` message. The console can also request the current mode using a `ID_LAB5_REQ_MODE`, which requires a repones from the microcontroller of `ID_LAB5_CUR_MODE`. At this point in the lab, your code should **only** operate in `COMMAND_MODE`.

When in `COMMAND_MODE`, the sensor reading for the PID loop is the total angle (Θ_{total}), and the commanded position is from the console message `ID_COMMANDED_POSITION`. Note that this is an int, allowing for multiple turns of the output shaft before reaching the target position.

2.3 Lab 5 Application: Position PID using Encoder

For this lab, you will be running the motor at 12V (not 24V as in Lab 4). This slows down the response a bit, which makes it easier to control. You will also use the 12V power supply to power your minibeacon in Sec. 4.

Warning: For safety, on startup the initial commanded position should be set to the current total angle, which is read directly from the encoder (there have been no opportunities for a rollover). This will set the error to 0, and effectively shut down the motor drive until another position is commanded.

This part of the Lab 5 application (there will be additional parts later) requires the following specifications (and you will note that many of them are quite similar to the Lab 4 application):

1. On startup:
   (a) Send an ID_DEBUG message that includes compilation date and time
   (b) Load PID gains for `COMMAND_MODE` from NVM
   (c) Set PID mode to `COMMAND_MODE`
   (d) Send an `ID_LAB5_CUR_MODE` message with current mode
   (e) Send an `ID_FEEDBACK_CUR_GAINS` message with current gains
   (f) Set commanded angle to current absolute angle
2. Respond correctly to protocol `ID_FEEDBACK_SET_GAINS` with `ID_FEEDBACK_SET_GAINS_RESP`
3. Correctly decode and set the P, I, and D gains in `ID_FEEDBACK_SET_GAINS`, reset the controller
4. Store the new gains in NVM for appropriate PID mode
5. Respond to a `ID_FEEDBACK_REQ_GAINS` message by sending an `ID_FEEDBACK_CUR_GAINS` message back to the console
6. Respond correctly to protocol `ID_LAB5_SET_MODE` by changing the mode
7. If mode changed, load appropriate gains from NVM, and send an `ID_FEEDBACK_CUR_GAINS` message with new gains
8. Respond to `ID_LAB5_REQ_MODE` by sending back the current mode in the `ID_LAB5_CUR_MODE` message
9. Respond correctly to the `ID_COMMANDED_POSITION` message by setting the commanded position and resetting the controller
10. At 100Hz update rate:
    (a) Run the PID loop
    (b) Scale the output of the PID loop appropriately
    (c) Set the motor speed (maximum of $pm1000$)
    (d) Send an `ID_LAB5_REPORT` message to the *ECE121 Console*

There are a few things to be cautious about when using the command mode. The first is that while this is still a benign system, it is less benign then was controlling the speed in Lab 4. Tuning the gains will be more challenging, and you will almost certainly need a "D" term to reduce the oscillations. You will also need an "I" term (though very small) to get rid of the steady state error.

The oscillations are a large function of the motor "stiction" where a certain amount of current is needed to get the motor to start moving. This causes the motor to overshoot and then the integrator term builds up until it can move the motor in the other direction, overshooting again.

Note that for this controller, the error is likely very large (commanded - sensed position is potentially huge). Thus you are going to have to ensure that your saturation limits on the controller work well, as you are going to go to full PWM until you get close. That is, the motor is going to hit its *max slew rate* while the error gets smaller; once the error is small enough, the control will actually modulate the drive to drive the error to zero. The anti-windup and the control saturation are both going to be very important to get this to work.

```
newmotorSpeed = ((int64_t)newmotorSpeed * 1000) >> FEEDBACK_MAXOUTPUT_POWER;
```

Note that the scaling of the motor speed (between ± 1000) will easily overflow the int because of saturation. In order to properly scale the output you will need the C code above (`int64_t` is from `stdint.h`). It uses the #defined FEEDBACK_MAXOUTPUT_POWER so that if this is changed later in your code, it will still give you the correct result.[4]

[4]Most control systems are implemented using floating point numbers and computations to keep from having to scale things; however they assume that the underlying language will just take care of it. There is a huge penalty for using floats in terms of speed of processing; floating point divide is approximately 80 cycles where a shift is 1 cycle. That said, the funky gain scaling is hidden from the designer if the entire thing is implemented in floats and the output is scaled between ± 1.0.

The *ECE121 Console* “Lab 5” tab, “Lab 5 Application” has buttons to set and request gains, to set and request mode, and to set a commanded position. There is also a special button to set the commanded position to 0. Below, there is a panel which shows the response of the `ID_LAB5_REPORT` message.

Start with just a proportional gain, K_p, and set it somewhere in the range of 5000 $\rightarrow$ 10000. You will find that the system overshoots quite a bit (the motor comes in to the target with too much speed and goes past the target point before coming back). Because of the stiction and deadband in the drive (motor won’t spin below a certain duty cycle), just the “P” part of the loop will leave you with some steady state error; once it gets close enough, the error isn’t large enough to overcome the stiction and deadband.

In order to address the overshoot, leave your K_p where it is and add in a healthy amount of K_d. This will increase the damping and reduce the overshoot. Finally, add in a very small amount of K_i and see if you can get the steady state error to reduce. Too much integral gain and the system will oscillate around the target.

As always, when changing gains, make sure that you change the target position (send it back to zero or from zero to your previous position) to ensure that the controller has to stabilize the system.

The *ECE121 Console* can log the `ID_LAB5_REPORT` messages to a file. Use this to create plots of the system response. Try commands that are close by, and commands that are many turns away.

2.4 Tuning the PID gains with Zeiger-Nichols

As in Lab 4, we will need to figure out what gains work to control the system. The process of figuring out what the gains should be is referred to as *controller tuning*. The PID loop is relatively easy to tune, and can easily be made to control any 2^{nd} order system without knowing anything about that system.[5]

There are quite a few methods for tuning PID gains out there, and some work better than others.[6] The one we will be using in this lab is the Zeigler-Nichols method from the 1940’s. The gains that come out of the Z-N method are not the best, but at least adequate and give a good place to start with further tuning.

To run Z-N, turn the integral and derivative terms off ($K_i = 0$ and $K_d = 0$), and increase the proportional gain, K_p until the system starts to oscillate. Note that you will get better results if you turn the motor off, set the new gain, and turn the motor back on. Once the system is oscillating, record your gain and the period of oscillation. Again, the logging capability will be essential here.

The gain at which the system starts to oscillate is designated K_c, for *critical* gain. The period of the oscillation is designated as P_c and is measured in consistent time units with the controller. This time base needs to match the integration time base to be consistent.

For a PD controller (which works quite well for the position control), set the gains as:

$$K_p = 0.80 K_c$$
$$K_d = \frac{K_c P_c}{10}$$

[5]Again, we are glossing over a huge amount of complexity and knowledge; take the controls courses if you are interested in understanding that complexity.

[6]A particularly good article on this is PID without a PhD.

The Z-N PID control gains are:

$$
\begin{aligned}
K_p &= 0.60 K_c \\
K_i &= 1.20 \frac{K_c}{P_c} \\
K_d &= 3.0 \frac{K_c P_c}{40}
\end{aligned}
$$

For more detail, see the Wikipedia article on Zeigler-Nichols. Note that the Z-N method is quite aggressive, but should work quite well as a starting point. If you find your system has too much overshoot in the speed, try reducing the proportional gain and integral gain by half and see if the response improves.

Experiment with the motor and set the commanded position to different values (within the range of the motor) to see how well you do. Make some plots using the logging capabilities of the console and include these in the final report.

Once you have all of this working, you should be able to command the motor head to both small angle changes and large angle changes in both directions. Demonstrate this to the TAs and show them that it works. They are going to reset your system, and it should keep working once it comes back up. Using the utilities tab of the *ECE121 Console*, log the data and plot it for some typical commands; see how it changes when you change the gains. Include these plots in your lab report.

Warning: Be sure that when you are driving the motor in `COMMAND_MODE` that the output shaft head is **NOT** attached. Any wires attached to the shaft head will get pulled out when you do multiple rotations. You will want some indication of the shaft position so that you can see that it is working.

In the lab report, ensure that a student who has already taken the class and is fairly well versed in embedded software would be able to recreate your solution. Include details and algorithms, pseudo-code, flow-charts, and any "bumps and road-hazards" that a future student might watch out for. Please be clear and concise, use complete sentences, and maintain a dispassionate passive voice in your writing.[7]

CONGRATULATIONS: you have completed a closed loop PID control loop for a real system, and you have largely coded it from scratch. The system is non-trivial and more challenging to control than the Lab 4 implementation. This is more than most of the graduate students in controls have ever done.[8]

3 Safety Limits on Head Position

Now that you have successfully demonstrated closed loop control on the motor output shaft using the encoder, you have almost all of the required software modules and functionality to implement analog sensor control (see Sec. 5).

The sensor head will be using two IR phototransistors in to the ADC module such that the *differential* signal between the two of them will be used to drive the PID loop. The implication of having a sensor head is that

[7]English teachers hate the passive voice, but it is used as the primary narration in scientific writing as you want the reader to know what was done, but not be concerned with who did it. In this sense, proper scientific writing is *egoless*. If passive voice is too burdensome for a particular sentence, use the "we performed" or other such construct instead.

[8]They have better algorithms than PID and have much more math and analysis (including predictions of performance and stability). That said, PID is the most common closed loop control there is, and you now have that under your belt.

the sensor head must necessarily be attached to the microcontroller through wires; this means that multiple turns off the motor head will yank the wires and possibly damage the sensor head.

Thus, you are going to be implementing a *software safety stop* to shut down your control if the sensor head goes out of bounds; this will be triggered by the absolute angle you developed for the closed loop motor control in command mode.[9] The basic idea is to keep track of the head position, and shut down the drive when you exceed the limit.

The first step to implementing the software safety limits is to define these limits (using `#define RANGE_LIMIT`) to be no more than $\pm 90^\circ$ from center. Note here that you can use your solution in Sec. 2.3 to determine exactly what commanded position corresponds to $\pm 90^\circ$ of the motor head.

The second step is to revisit your `DCMotorDrive.h/c` module and implement the `DCMotorDrive_SetBrake()` function. From the L298 datasheet, the brake mode is turned on by setting `IN1` and `IN2` to the same value **AND** leaving `ENA` high. This will turn on both bottom (or top) FETs on the H-bridge and cause the motor to be dynamically braked; that is the back EMF will flow through the coil and oppose the motion of the motor causing it to slow down and stop more quickly.

The steps to tripping the safety limit are outlined in Alg. 2.

```
Algorithm 2: Software safety Limit
Input: $\Theta_{total}$, newMotorSpeed; absolute angle and motor speed
Result: DCMotorDrive_Brake() deployed
if (($\Theta_{total}$ >RANGE_LIMIT) && (newMotorSpeed $> 0$)) then
    MotorDrive ← DCMotorDrive_Brake();
else if (($\Theta_{total}$ $<$ −RANGE_LIMIT) && (newMotorSpeed $< 0$)) then
    MotorDrive ← DCMotorDrive_Brake();
else
    MotorDrive ← DCMotorDrive_SetMotorSpeed(newMotorSpeed);
Set MotorDrive PWM accordingly
```

The basic premise is very straightforward; if the motor is out of limits (Θ_{total} >`RANGE_LIMIT`), and being driven farther out of limits (`newMotorSpeed` > 0), then the motor drive should be put into brake mode. The limits are symmetrical, and should have the negative side implemented as well. Alg. 2 allows you to drive back to center from the limit, but not drive further outside of the safety bounds. Note that due to inertia of the motor, even with the safety limits in place you will go farther than your limits before the head comes to rest.

Info: Note here that this is not the only way to implement the software safety limits; this is just the most simple method. Another viable method is to check if you are outside of your range limits, and then immediately switch to `COMMAND_MODE` (if not already in it) and set the $\Theta_{\texttt{command}}$ to ±`RANGE_LIMIT` as appropriate. This will use the machinery of the PID control to drive the motor head to the limit and hold there (thus no `DCMotorDrive_Brake()` required).

In order to get a visual indication that your drive head has exceeded the limits, turn on half the LEDs if you are off in the positive direction, and the other half if you are off in the negative direction. This will help you verify that your software safety limit is actually working (plus it makes debugging much easier).

[9]Software safety systems have a rather decidedly mixed record in actually improving safety; more often than not they introduce an unexpected complexity that proves disastrous. Mechanical stops are better, because their affect is more easily understood.

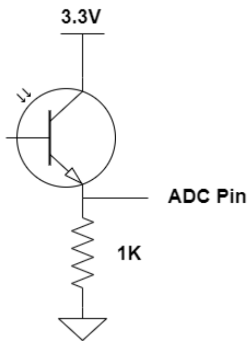

Figure 1: Phototransistor schematic for analog sensing

As with all such software safety systems (as opposed to mechanical stops), the safety requires that there be no mistakes in the underlying algorithms. That is, if your `DCMotorDrive_SetMotorSpeed(newMotorSpeed)` is incorrectly coded, you might not stop at the limit.

With your safety limit algorithm in place, use the Lab 5 Application to ensure that you cannot command the motor head beyond the limits you have established. Set commanded angles to farther than your limit, and the motor head should stop until you drive to a position inside your limits. Test this very thoroughly, as you will rely on it to save your sensor head.

4 Analog Sensor Head: IR Phototransistors

The final part of the lab is to run the PID loop directly from an analog sensor in order to track a target. You have implemented the safety limits on the control in Sec. 3 precisely to protect the sensor head from over-rotation.

The sensor we are using are simple IR phototransistors, along with an emitter at a specific frequency. The reason to use a modulated emitter is for signal discrimination: there should be far less noise out at 1.5KHz than around DC. Also, since we are manipulating the signal in software, we will be doing things like rectification and smoothing in software rather than build complicated analog filters.[10]

The first thing to do is to hook up the actual phototransistors. They fit through the holes in the bottom plate of the sensor head, and you must solder wires to the leads *after* they are inserted in the holes. The basic schematic is shown in Fig. 1, with two identical phototransistors set up in a sourcing configuration. The supply power should be 3.3V, and the resistor should be $1K\Omega$; the input should go directly into the ADC pins. Set one phototransistor to go into AD1 and the other to go into AD2.

Hook up the 3.3V and the resistor to ground, and use one of the "mini beacons" available in the lab to test your phototransistors and make sure that you have them hooked up correctly. Hook up an oscilloscope and make sure that you are getting what looks like a rounded square wave at 1.5KHz. Note how the amplitude of the square wave will change greatly with distance. If you assemble the sensor head mechanical parts, you can move it back and forth and see the signal change on the two phototransistors.

[10]Analog filters are an important thing to understand and build, and give their own set of advantages and disadvantages. That said, they are beyond the scope of this class.

4.1 Digital Manipulation of the Sensor Stream

From looking at the output of the pair of phototransistors on the oscilloscope, you should see that the amplitude of one of them compared to the other changes quite dramatically as you go off axis with the mini beacon. That differential signal is what you are going to track.

ADC Specifications: In order to work with the stream, you will have to set up your ADC to sample the signal fast enough that you can actually capture the square wave. Nyquist says as long as we sample faster than 3KHz we can see the 1.5KHz wave, but giving ourselves a little overhead, set the ADC such that you are sampling each pin at 6KHz.[11]

Bandpass Filtering: The first thing you are going to need to do with the signal is to bandpass filter it to get rid of any frequency content outside the range of interest. This will also kill off any DC bias in the signal, which will be very important when taking the difference. Under the Lab3 tab in the *ECE121 Console* is a button to "Load Band Pass," which operates exactly as your other filters did in that lab. When the button is pressed, an `ID_ADC_FILTER_VALUES` message is generated from the console and sent to your microcontroller.

Your microcontroller should load the filter for *both* ADC channel 1 and 2. It should also copy the values to a suitable location in the NVM (and of course load them on startup). Your signals from the ADC are run through the FIR bandpass filter.

Rectification: The post-bandpass signal is going to be something akin to a square wave (rounded edges) centered around 0, with the amplitude being a measure of how much 1.5KHz IR is falling on the sensor. If you average the signal, you will get 0. In order to track the amplitude, you must first rectify the signal. This means that if the filtered reading is negative, multiply it by -1 (turning it positive again).

Moving Average: The band-passed, rectified signal is still quite noisy, and needs to be smoothed in order to make it useful enough to feed into the PID control loop. A moving average is simply the average of n samples; this means you need to keep a 2-D array that is the AD channel for each row, and the samples in the column (the length of the column should be #defined as `MOVING_AVERAGE_LENGTH` and should be a power of 2).

As before, the 2-D array will be treated as a circular buffer (except that it is full and you are always overwriting the oldest data). This is easy to accomplish using the increment (++) and modulo (`%`) operators. Create a helper function inside your main that implements the moving average (which sums the numbers in the column of your 2-D array and then divides the total by `MOVING_AVERAGE_LENGTH`).[12]

Differencing: The final step of the software manipulation is to take the difference of the two (post-bandpass, rectified, moving averaged) signals to create a sensor reading that is signed and gives you a signal proportional to how far off axis you are.

> **Info:** The phototransistors have an attenuation from going off axis that will cause the signal to fall off as the angle between the IR signal and the phototransistor itself gets large. In addition there is an MDF wall in the sensor head between the two IR phototransistors. The total of your signal is going to be in the range of ± 300; and it won't keep getting bigger as you get farther off to one side. It should be quite small when directly in front of the sensors.

[11] Yes, we realize this means going back and changing your ADC code, and in the future we will probably have you do a conditional compilation using #defines to change it.

[12] If the `MOVING_AVERAGE_LENGTH` is a power of 2, then the divide can be replaced by a shift (and note that the compiler is probably smart enough to recognize the number and implement it for you). If you are really clever, your helper function will also handle overwriting the oldest value in the array with the new one at the same time it is calculating the moving average.

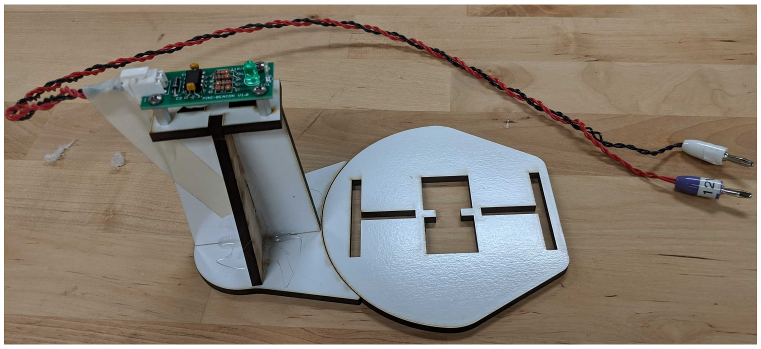

Figure 2: Minibeacon stand for testing sensors and control

4.2 Testing the Analog Sensor

Testing the IR sensor head can (and should) be done with the motor unpowered. You are going to power your minibeacon at 12V and using one of the provided stands and base additions (shown in Fig. 2), you will have a mechanical stop to ensure that the minibeacon is at a constant radius from the sensor head. Keeping a constant radius keeps the power level on your sensor constant.

The test stand is designed such that your motor test stand fits into the base to provide uniform distance to the sensor head. The wires are marked 12V and ground, and should be connected to your power supply at 12V.

Warning: Powering the minibeacon at higher than 12V will result in damaging the mini-beacon. Given that you have only one power supply available to power your motor and mini-beacon, you will be running the motor at 12V for this lab.

The *ECE121 Console* provides a panel for testing the sensor head and ensuring that you are getting good values. The message `ID_LAB5_ADC` contains three `shorts`, which will be displayed in the Lab 5, Lab 5 Preflight tab as Channel 1, Channel 2, and Channel 1 - Channel 2, respectively. The console will calculate internally all of the averages, min, max, peak to peak, and display them.

Note that the preflight tab is a powerful debugging tool. You can send *any* three shorts and they will be displayed (and some rudimentary statistics calculated). Thus you could start with the raw ADC signal on one and the filtered ADC signal on the second, and then the moving average on the third. Once you are convinced that your filter and moving average are working, you can change that to the moving average of both channels and the difference on the third short (this matches the labels).

While monitoring the output of your sensor head on the Lab5 PreFlight tab, move the minibeacon back and forth and watch the output. Are you getting numbers that make sense. Is the signal trackable? What happens when you go really far off axis? Show this to the TAs for checkoff. Write it up in your lab report using any appropriate diagrams to make it clear how (and why) you did things the way you did.

5 Closed Loop Control of DC Motor Position: Tracking IR Sensors

At long last, all of the pieces needed to complete this lab are in place, and you are ready to implement the closed loop *regulator* that will drive the motor head to face the IR source. This is called a regulator (as opposed to a tracker) because you are always attempting to drive the sensor source to 0.

Since your sensors (as implemented in Sec. 4) are filtered, rectified, and smoothed (as implemented in Sec. 4.1) and differenced, you have a signal that is positive on one side, negative on the other, and roughly zero when facing the IR source.

This similar to calculating the error (reference - actual) that you did in the command mode except that now the reference is always zero and the actual is the filtered sensor reading. The very big difference is the scale of the errors you will feed to the PID algorithm. With the encoder, a 20° error in the motor head corresponds to an error of on the order of $80K$ counts. With the IR phototransistors, that same 20° error will give you a sensor error of approximately 300.

Thus, the gains that you used on the command mode *will not work* when you are in sensor mode; they will need to be *much* larger. Set your proportional gain, K_p, to the range of $400,000 \rightarrow 500,000$ and see how it works. Use your beacon mount and base extension (Fig. 2) to keep the minibeacon well aimed and at a constant distance from the sensor head.

The minibeacon needs to be pretty well centered on the sensor head for it to track, but once it locks on, you should be able to move the minibeacon back and forth and have the sensor head track the minibeacon.

Info: When you are setting gains in sensor mode, these gains should be stored in your NVM in a *different* location from the gains in command mode. When switching modes, you should load the appropriate gains (command or sensor) from NVM.

Again, use the Zeiger-Nichols method (Sec. 2.4) for tuning your gains. Try both the PID and the PD control, see which works better. You may find that integral gain, K_i, does more harm than good when tracking the sensor. We don't really expect you to have exquisitely tuned control loops; rather we hope that this lab gives you an opportunity to play with the gains a bit and build up some intuition.

6 Lab 5 Full Application

With the sensor mode working (and hopefully the command mode already working), it should take relatively little effort to complete the full Lab 5 application. This is what you will demonstrate to the TAs, and will also complete the lab requirement of this class.

This full Lab 5 application requires the following specifications (and you will note that many of them are quite similar to the command mode):

1. On startup:
    (a) Send an ID_DEBUG message that includes compilation date and time
    (b) Set ADC to trigger at 6KHz per channel, ripple through AD1 $\rightarrow$ AD4
    (c) Load bandpass FIR filter from NVM
    (d) Load PID gains for `COMMAND_MODE` from NVM

    (e) Set PID mode to `COMMAND_MODE`
    (f) Send an `ID_LAB5_CUR_MODE` message with current mode
    (g) Send an `ID_FEEDBACK_CUR_GAINS` message with current gains
    (h) Set commanded angle to current absolute angle

2. Respond correctly to protocol `ID_FEEDBACK_SET_GAINS` with `ID_FEEDBACK_SET_GAINS_RESP`
3. Correctly decode and set the P, I, and D gains in `ID_FEEDBACK_SET_GAINS`, reset the controller
4. Store the new gains in NVM for appropriate PID mode
5. Respond to a `ID_FEEDBACK_REQ_GAINS` message by sending an `ID_FEEDBACK_CUR_GAINS` message back to the console
6. Respond to a `ID_ADC_FILTER_VALUES` by storing bandpass filter in NVM and loading it into memory to process ADC values
7. Respond correctly to protocol `ID_LAB5_SET_MODE` by changing the mode
8. If mode changed, load appropriate gains from NVM, and send an `ID_FEEDBACK_CUR_GAINS` message with new gains
9. Respond to `ID_LAB5_REQ_MODE` by sending back the current mode in the `ID_LAB5_CUR_MODE` message
10. Respond correctly to the `ID_COMMANDED_POSITION` message by setting the commanded position and resetting the controller
11. If controller takes sensor head out of allowable range: light the appropriate half of the LEDs, set drive to `BRAKE`, and only allow PWM that brings the sensor head back in range.
12. At 100Hz update rate:
    (a) FIR filter, rectify, and moving average the ADC IR readings
    (b) Difference the filtered ADC IR readings to form composite sensor reading
    (c) Run the PID loop
    (d) Scale the output of the PID loop appropriately
    (e) Set the motor speed (maximum of ± 1000)
    (f) Send an `ID_LAB5_REPORT` message to the *ECE121 Console*

You should be able to set your gains appropriately for each mode, drive the motor either based on the encoder (`COMMAND_MODE`) or the IR sensors (`SENSOR_MODE`). You should respond correctly to all of the *ECE121 Console* messages, and report back the control loop data. Because your gains are all stored in NVM (as well as your FIR filter coefficients), you should be able to reset your micro, and change the targets (either commanded position or move the minibeacon). Your system should still track just fine, since all of the constants you needed were retrieved from NVM at startup.

Log the data using the utilities tab of the console; plot out the response to both commanded positions in encoder mode, and to moving the minibeacon in sensor mode. The `ID_LAB5_REPORT` message has enough information to make these plots interesting (even though some of the information is redundant in certain modes). Include your plots in your lab report.

Demonstrate this working to the TAs for checkoff. In the lab report, ensure that a student who has already taken the class and is fairly well versed in embedded software would be able to recreate your solution. Include details and algorithms, pseudo-code, flow-charts, and any "bumps and road-hazards" that a future student might watch out for. Please be clear and concise, use complete sentences, and maintain a dispassionate passive voice in your writing.[13]

[13] English teachers hate the passive voice, but it is used as the primary narration in scientific writing as you want the reader to know what was done, but not be concerned with who did it. In this sense, proper scientific writing is *egoless*. If passive voice is too burdensome for a particular sentence, use the "we performed" or other such construct instead.

Epilogue

You should note that this lab brings together almost all of the pieces you have developed over the course of the quarter. From Lab 1 you are using the protocol to send commands to your micro as well as reporting and logging of the data. From Lab 2 you are using the free running timer and the encoder library (with SPI angle reads from Lab 4). From Lab 3 you are using the NVM reads and writes to ensure that your system works well at startup, and from Lab 4 you are using the PID control loop and the motor drive through the H-bridge.

If you have reached this point, and successfully completed these five labs, then you should be able to tackle almost any embedded project on any device. While each micro has its own SFRs and differs subtly in how to use them, you have learned how to read the datasheet (FRM) and decode the information. Bringing up a micro from a different vendor will require some time and study, but should not be fundamentally any different that what you already know how to do.

And while we have not covered some of the peripherals on the PIC32 processor (e.g.: DMA), these are again quite tractable given what you already know how to do; pick up the FRM, start tinkering, and get it working.

Since this is the first time this class has been taught, there have been some pitfalls, trials, and tribulations; that said, we (the instructional staff) believe that you have learned a great deal, covered enormous ground, and are very well positioned to complete future embedded projects. Even if you remain in the “big” computer world (as opposed to microcontrollers), the techniques and methods you have learned will serve you well.

Please do not hesitate to reach out to us and give us feedback on where to smooth the edges. We will read every comment and consider every suggestion with humility and professionalism (though there might be reasons why we decide not to implement your specific suggestion). We know what we know about embedded systems, but you know what it is like to experience the class from the other side—please share that experience with us.

A Bang-Bang Control of DC Motor Position

Bang-bang control (where the control is full forward, full reverse, or shut down) is used in many applications. It is almost always the simplest control to implement, and in can actually have very high performance. There are a ton of variations on it that make bang-bang quite sophisticated, but these are beyond the scope of this class.

People often confuse bang-bang control with "open loop" since there is rather limited feedback. However, the feedback is rather important. The basic idea is to drive the motor (in the correct direction) until you are *close enough*, and then shut the drive down. Typically, there is a *deadband* that defines close enough. Thus the control boils down to:

$$\begin{aligned}\varepsilon > \texttt{DEADBAND} &:\mapsto \texttt{PWM} \leftarrow -100.0\% \\ \varepsilon < -\texttt{DEADBAND} &:\mapsto \texttt{PWM} \leftarrow +100.0\% \\ \|\varepsilon\| \leq \texttt{DEADBAND} &:\mapsto \texttt{PWM} \leftarrow 0.0\%\end{aligned}$$

If your `DEADBAND` is too small, then the system will chatter (oscillate back ad forth), and if it is to large, then you will have errors. `DEADBAND` also defines the smallest step you can take (any step within `DEADBAND` won't cause a response).

Most of the embellishments to bang-bang control have to do with reducing this chatter and the error. If you consider the `PWM` drive signal vs time, it looks like a square wave (with the up time corresponding to the distance you need to travel). One improvement to bang-bang is to break up the trajectory into three segments: (i) Acceleration to Maximum Velocity, (ii) Cruise at Maximum Velocity, and (iii) Deceleration to a stop.

With a small bit of imagination, this turns into a *trapezoidal* trajectory (when looking at the velocity vs time). The idea is to change the cruise time (flat part of the trajectory) so that you hit your target exactly. When the step size gets small, you never get to the flat part, you accelerate and then decelerate to hit the target.

One of the important areas of non-linear control (of which bang-bang certainly qualifies) is to determine the *switching line* when to switch from full forward to full reverse such that you arrive at exactly the right place with zero velocity.

You can get very fancy with this kind of control, and it is often the basis of the feedforward part of the control loop (with a PID loop wrapped around the error from the trapezoidal trajectory). In other words, for any size step, you have a trajectory of velocity vs time that will get you to the target with no error (assuming your predictions about acceleration and velocity are 100% accurate). At any point in time, the difference between that nominal trajectory and where you actually are become the error that you drive to zero.

All of this is, of course, way beyond the scope of this class. But it is provided as interesting background material, and hopefully inspiration to learn more.

Acknowledgements

- Figure 1 courtesy of Gabriel Hugh Elkaim (using Draw.io)
- Figure 2 courtesy of Maxwell Dunne