ME230 Lecture Notes

PWM, Optical Encoders, and Rollover

1 Introduction to Control Systems

The purpose of a control system is to achieve the desired output from a physical system by controlling the input to a system. The components of a control system include sensors which measure system output, actuators which generate the input to the system, and the controller which is responsible for deciding what input will be sent to the system based upon the measured output.

In this handout we will discuss methods for delivering the control signal from the controller to the actuator. We will also detail the measurement of system output using sensors.

2 Pulse-Width Modulation

In many applications, DC motors are controlled with variable-voltage amplifiers. Often called "linear amplifiers", these amps provide an output which is equal to a gain times the input voltage over their operating range. They also provide the necessary current to drive the motor. If you had to move one of these units, you may have noticed how large and heavy it was. Linear power amplifiers are in general large, heavy, and expensive. Figure 1 provides an idea of what a linear amplifier consists of. A closeup of the capacitors is given in Figure 2.

A smaller, lighter, and cheaper (as well as more energy-efficient) alternative is the switching amplifier. Switching amps cannot continuously vary their output; they can only turn it on or off. They can also reverse the direction of current flow. An example of a switching amplifier is given in Figure 3.

If an amplifier can provide only 0% or 100% of maximum power, how do we make the motor run at half speed? The answer lies in switching the amplifier on and off rapidly so that the average current to the motor is half what it would be if the amplifier were on all the time. There are a number of schemes by which to control the switching, but the one we will use here is pulse-width modulation, or PWM. Three pulse-width modulated motor control signals are shown in Figure 4. Note that a PWM signal is defined by it's period (the amount of time between two rising edges of the signal) and the duty cycle (the percentage of the period that the signal is in the "on" state).

To show that pulse-width modulation can work, consider the following simulation. We'll use the standard first order differential equation for a motor. Once again, the dynamics are described by

$$\frac{d\omega}{dt} = A\omega + B \cdot V$$

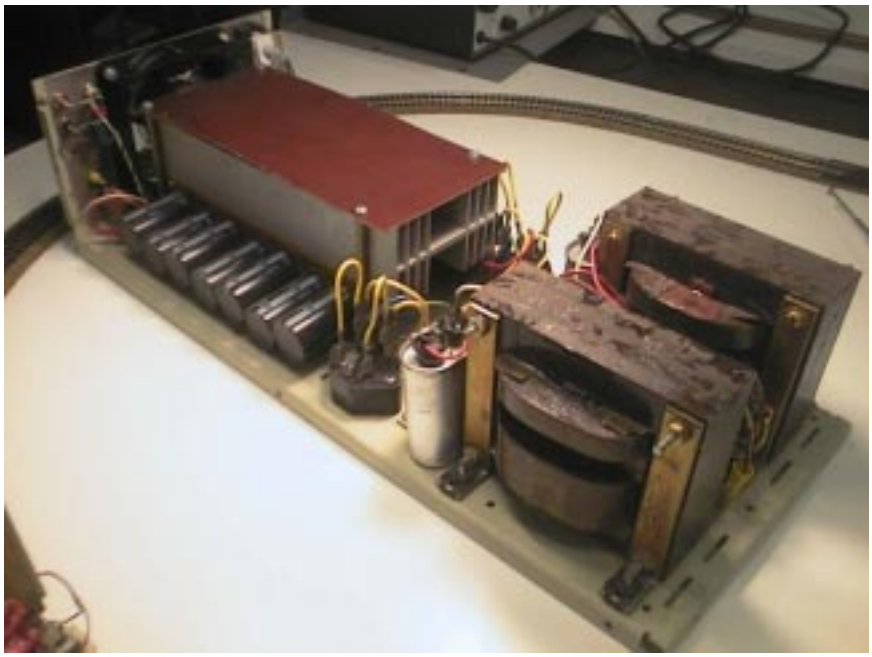

Figure 1: A linear amplifier capable of amplifying one input. Note the transformers, and heat sink/wind tunnel.

Figure 2: D-Cell batteries? Not quite. These are 6800 μF capacitors. The linear amplifier above has 14 of these.

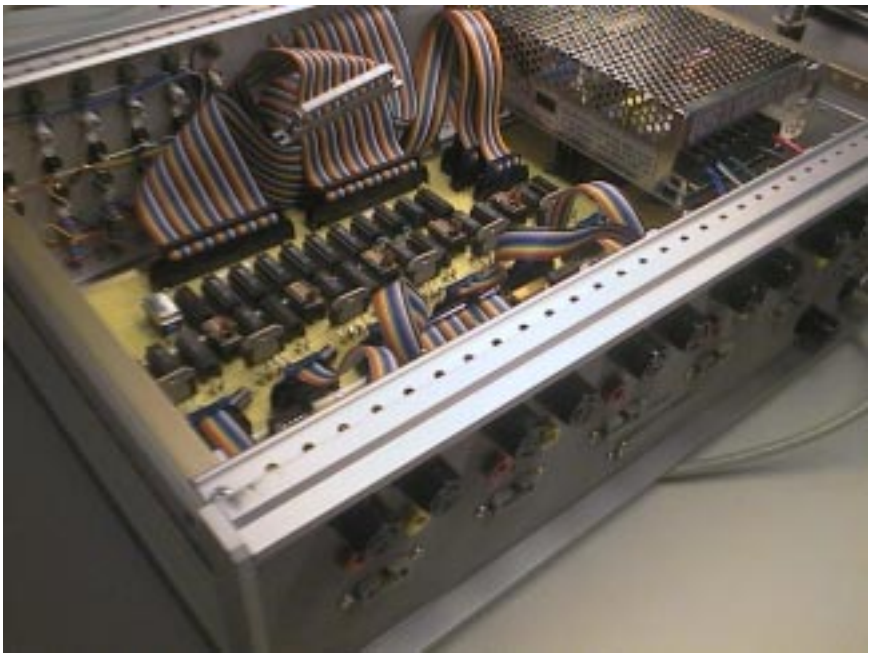

Figure 3: A switching amplifier capable of driving six different motors.

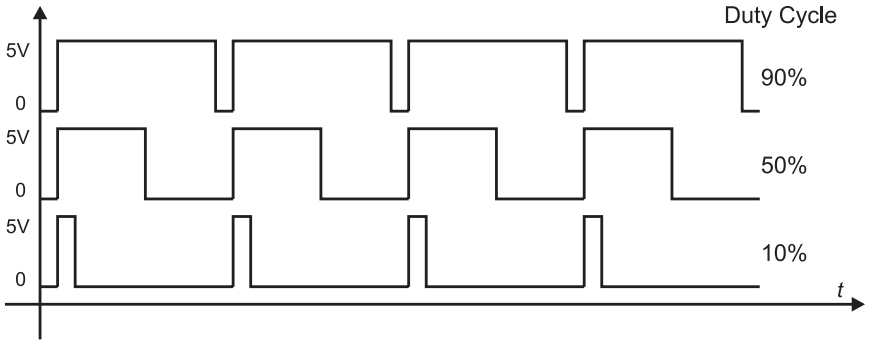

Figure 4: Signal waveforms of a voltage PWM. Note that even though the duty cycle changes, the period remains constant.

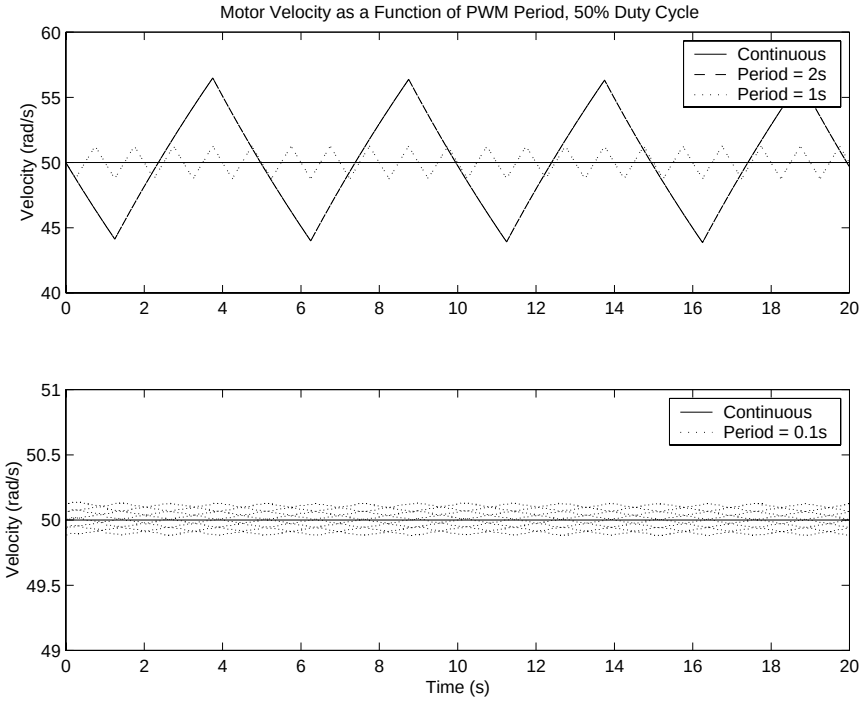

Figure 5: Motor velocities for different PWM voltage inputs. Note that different scales are used between the two graphs.

where ω is the angular velocity of the motor shaft, t represents time, A and B are experimental constants, and V is the input voltage. As far as numerical values are concerned, in this simulation, we will use $A = -0.1$ and $B = 1$. We will attempt to run the motor at a constant $50\,rad/s$. By applying a continuous voltage of $5\,V$ we can maintain this velocity exactly. We can also approximate this behavior using a PWM voltage input which is $10\,V$ when on, $0\,V$ when off, and is on "half the time". More precisely, the duty cycle for the PWM signal is 50%. Figure 5 shows the resulting velocity traces for different PWM periods.

2.1 Optical Encoders

The heart of an optical encoder is a pair of light emitters and detectors. These devices transmit and receive infra-red light of about the same wavelength as that used in television remote controls and laptop computer IR links. In an optical encoder, the light sources transmit continuously. The receivers detect light whenever the path between transmitter and receiver is not broken by the presence of an object which blocks the light.

A clear plastic mask painted with dark bands is placed between the transmitter and receiver as shown in Figure 6.[1] As the shaft rotates, the dark bands on the mask move across the path from each transmitter to detector. When a detector senses light, it sends out a voltage of zero volts, or logic 0; when no light is sensed because the path is blocked, the detector's output is 5 volts, or logic 1. The result is a sequence of logic pulses, 1 0 1 0 1 0 ..., as the shaft rotates. Because the dark

[1]Some encoders use a metal disk with holes or slots cut in it—for an example look at the Rhino robots in 2170.

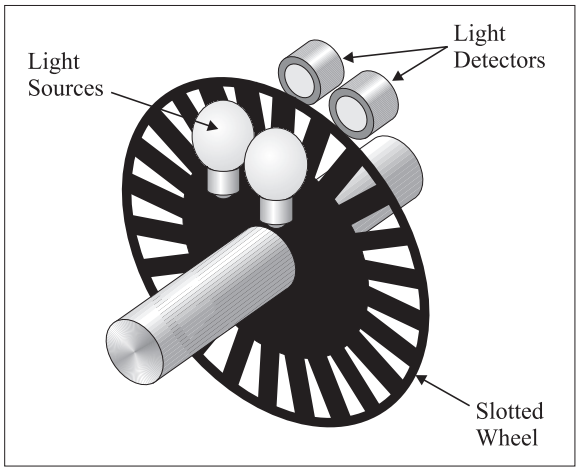

Figure 6: Graphical depiction of an optical encoder.

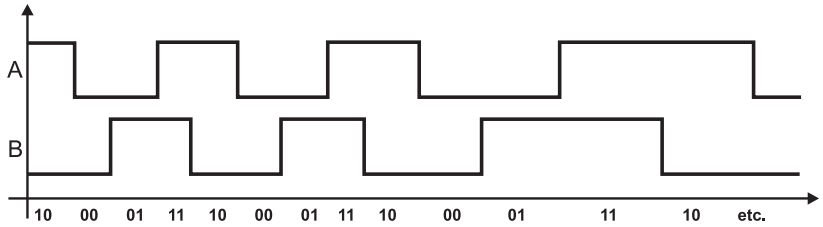

Figure 7: Possible quadrature signal. The waveform of channel A is the output of one receiver, while channel B is connected to the second receiver.

bands are spaced at equal angles along the shaft, each logic pulse corresponds to a given angle. For example, if you have a 512-band-per-revolution encoder, each pulse indicates $360/512 = 0.70$ degrees of angle.

If you count the logic pulses, you know how far you've gone – but which way? If we've counted 256 pulses, the shaft of the 512-line encoder could have turned through 180 degrees clockwise, 180 degrees counterclockwise, or 90 degrees each way for a net motion of zero. In order to allow direction to be determined, optical encoders are constructed with two light sources and detectors arranged in such a way that the sequences of pulses which come from the detectors are approximately 90 degrees out of phase. A typical sequence of these pulses is shown in Figure 7. This is a two-bit quadrature signal, where the output of one photoreceiver is the first bit, and the output of the other receiver is the zeroth bit. If the shaft moves clockwise, a sequence such as {00, 01, 11, 10, 00, ...} may occur; while if the shaft moves counterclockwise, the sequence will run in the opposite direction: {00, 10, 11, 01, 00, ...}.

A typical DC motor may turn at several thousand RPM. With a typical optical encoder, this can cause the generation of a rather rapid sequence of pulses. Consider a motor running at 5000 RPM (or 83.3 revolutions per second) turning a 512-line encoder. This will produce (do the math) over 42,000 pulses per second from each of the two channels. To make matters worse, the quadrature signal changes state four times for every complete pulse (from 0 to 1 and back to 0 again) on each

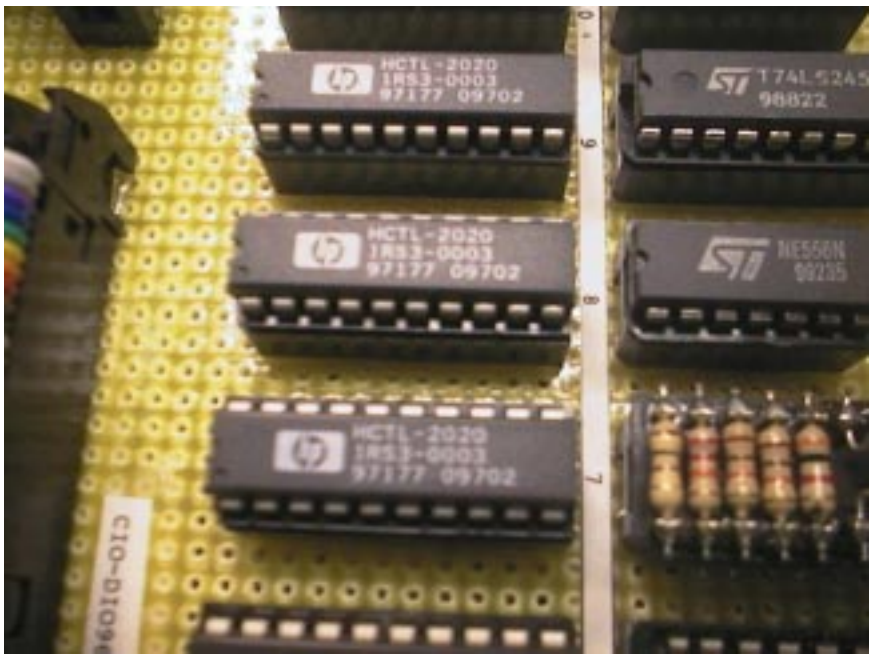

Figure 8: A closeup of Hewlett-Packard HCTL-2020 counter/decoder chips. Each 2020 chip is capable of measuring the position of a motor fitted with an optical encoder.

channel. This gives us about 171,000 transitions per second. Even using our techniques of software design does not allow a personal computer to keep track of such rapid transitions. The answer is to use a hardware device to keep track of the transitions and present a total number to the computer. The device used is an integrated quadrature counter/decoder chip such as Hewlett-Packard's HCTL-2020. This circuit can measure quadrature counts at speeds of over a million per second and present a total count of transitions recorded to the computer.[2] It records one count for each transition in the quadrature signal, so when attached to a 512-pulse encoder an HCTL-2020 chip will record 2048 pulses for each revolution of the encoder shaft. A picture of a HCTL-2020 chip is given in Figure 8.

2.2 The Rollover Problem

So now that our computer can read a total number of counts from the encoder, it might seem that our problems are over, because we can read the output of an HCTL-2020 chip anytime we like and know the position of the motor shaft. Not quite! The reason is that the counter chip only has 16 bits. Once it has counted up past 32,767 or down past -32,768 the counter will overflow. It's just the same rollover problem as we experience when using a 16-bit integers (`short int`). As an example, try to place the number 33,000 into a `short int`! You'll find that it has become -32536! This phenomenon is called an overflow. In the case of a counter, attempting to increase (or decrease) the counter value past the maximum (or minimum) value is termed as rollover.

[2]If you wish to know more about how quadrature counting chips and other high-speed mechatronics hardware work, take a course like ME235, in which mechatronics hardware design is taught.

An intuitive example of rollover is a clock. As the minutes increase {56, 57, 58, 59, ...} one might expect that the next number is 60. We know better however, and the minute counter rolls over and become 0. For the HCTL-2020 counter, at 5000 RPM for a 512-line encoder, these rollovers occur about 2.6 times per second. This rate is slow enough that we can detect these overflows in software. In this way we can use the decoder chip output to maintain a total count (kept in a `long int` of course[3]) of how far the shaft has rotated.

There are many solutions of the rollover problem. For example, simple `if/else` logic can be implemented as in the following pseudocode:

```
long int NewCount, OldCount, TotalCount

...

NewCount = ReadEncoderCounter ();

// This only guards against one direction of rollover.
if ((NewCount - OldCount) > 32767)
TotalCount += -65536 + (NewCount - OldCount)

...
```

A much more elegant solution involves using the fundamental data types of C++. To look at that requires digression into the binary number system.

2.2.1 Binary Addition

Given two integers in base-10 representation, addition is trivial. But how do we do this if the numbers are in base-2? The addition rules for binary numbers are similar to base-10 additions, and are as follows:

1. Two binary integers are stacked on top of each other with the least significant bits lining up, and addition occurring between respective bits. If the numbers are not of the same length, padding by 0 is used.
2. Addition proceeds from right to left.
3. The sum of two bits is trivial except in the in the case of adding two 1's, in which case the resultant digit is 0 and a 1 is carried over to the next operation. If you carry a 1 and the next column already has the addition of two 1's, the resultant digit is 1 and a 1 is carried over to the next operation.

Some illustrative examples are given in Table 2.2.1.

[3]Recall that this is a 32 bit integer, capable of holding well over 4 billion counts.

Base-10	Base-2	Base-10	Base-2
2	10	7	111
+3	+11	+7	+111
=5	=101	=14	=1110

Table 1: Examples of binary addition.

Signed Base-10	Base-2	Unsigned Base-10
-1	1111	15
-2	1110	14
-3	1101	13
-4	1100	12
-5	1011	11
-6	1010	10
-7	1001	9
-8	1000	8
7	0111	7
6	0110	6
5	0101	5
4	0100	4
3	0011	3
2	0010	2
1	0001	1
0	0000	0

Table 2: Positive and negative representation of signed integers.

2.2.2 Two's Complement Notation for Representing Binary Numbers

Previous discussions have used only nonnegative integers in explaining the binary number system. Clearly however, we must be able to represent negative integers using binary. To accomplish this different methods are used, but the most common in computer applications is called two's complement notation.

First of all, the one's complement (represented in C++ as the tilde) of a binary number is found by taking all the 1's in the number and changing them to 0's and all the 0's and changing them to 1's. The two's complement of a number is found by adding one to the one's complement of a number. For example, the two's complement of 1011_2 is 0101_2, and the two's complement of 10011000_2 is 01101000_2 (remember to add one to the one's complement of the number).

The claim is that the negative of an integer is simply the two's complement of the number. To verify, first realize that 0_{10} and -0_{10} should be the same number. In four bits, $0_{10} = 0000_2$ and by claim, -0_{10} is the two's complement of 0000_2 which equals $1111_2 + 0001_2 = 0000_2$ (where the carried one is lost because we're only using four bits). Further apply the rules of addition to a positive number and its two's complement and note that the sum is zero![4] Therefore, constrained to four bits, we have the results given in Table 2.2.2.

[4]Binary subtraction is often performed as the addition of a number and the two's complement of the subtracted number.

One observation is that in a signed number, the most significant bit is interpreted as the sign of the number. Note that the computer cares little whether or not 1111_2 is a 15_{10} or a -1_{10}. Clarification of this issue is left to the programmer and will be discussed in the following sections.

2.2.3 Binary Manipulation Applied to Rollover

The rollover problem can be solved by interpreting a current position reading as an `unsigned int`[5] and the difference between a current and previous reading as a `signed int`. The total count (a `long int`) is updated with the difference. In pseudocode, this might look something like:

```
unsigned short int NewCount, OldCount;
signed short int DeltaCount;
long int TotalCount;

...

NewCount = ReadEncoderCounter ();

DeltaCount = NewCount - OldCount;

TotalCount += DeltaCount;

...
```

At first glance, it doesn't seem as though this would work at all. But imagine the case of a four bit counter. Imagine that the previous position reading (returned as an `unsigned int`) was 3 and the new position reading is 14. It's fairly obvious to us that rather than going {3, 4, ..., 13, 14} the more likely case is {3, 2, 1, 0, 15, 14}. Following the algorithm above, we take the unsigned values and perform binary subtraction.

$$(14 - 3)_{10} = (1110 - 0011)_2 = (1110 + 1101)_2 = 1011_2$$

Here's where it's important that we interpret the difference as a signed binary number. Note that if we interpret as unsigned, the difference is 11 — the wrong answer. But 1011_2 is -5 when looked at as a signed integer!

2.3 Velocity Measurement

Either the total count of pulses or the raw data from the HCTL-2020 decoder can be used to compute an estimated velocity of the motor. If the total count is used, rollover is not a problem; however, if the raw decoder output is used, rollover must be accounted for. Either way, velocity can be estimated by dividing the number of encoder counts which occurred during a period of time by the length of that time period. Because the velocity is calculated from integer data, the measurement can become rather imprecise when the motion is slow or the time interval is short. A really good program will detect this problem and use some type of filtering to improve the precision in these cases.

[5] This is not necessarily the case, however, the ReadEncoderCounter() function used in the lab returns the count as an unsigned integer.

Design and implementation of filters can be a very advanced topic. However, if we look at the problem in terms of what we already know, creating a filter to smooth the velocity signal calculated from an optical encoder isn't that hard. Consider the PWM signal from Section 1. Note that as the period of the input (PWM voltage) signal decreases, the velocity signal is much less affected by the on/off transitions of the input signal. In terms of frequency, as the period decreases, the frequency of the signal increases. And as the frequency increases, it has less and less of an effect on the output (velocity). In addition, it should be intuitively obvious that if the motor had a greater inertia, high frequency signals would affect it even less. For example think of a large road vehicle and a PWM signal applied to the throttle. Since the vehicle has so much mass, quick fluctuations have little to no effect on the velocity of the vehicle.

In this way, we can interpret the DC motor as a filter. More accurately, the first order differential equation which describes the relationship between voltage and velocity is a filter. So to smooth the velocity signal calculated by numerically differentiating the optical encoder position, we can run it through a first order differential equation!

For example, let v_u be the unfiltered velocity, and v_f be the filtered velocity. We can write an equation that we will use to relate the two:

$$\tau \dot{v}_f + v_f = v_u$$

where τ will be the time constant for the filter.[6] Now, since we will be implementing this filter in the digital domain (on a computer) we need to make this into a discrete signal. As a simple approximation, we can use Euler's method. Let k be the time of the current velocity measurement, $k-1$ be the time of the previous time measurement, and Δt be the difference in time of the two measurements. Applying Euler's algorithm, we have

$$\tau \frac{v_{f,k} - v_{f,k-1}}{\Delta t} + v_{f,k} = v_{u,k}$$

Solving for $v_{f,k}$ gives the filtered velocity at the current time step in terms of the filtered velocity at the previous time step, and the unfiltered velocity at the current step — both items which we know!

$$v_{f,k} = \frac{\tau}{\tau + \Delta t} v_{f,k-1} + \frac{\Delta t}{\tau + \Delta t} v_{u,k}$$

[6] This is a tuneable parameter which determines what frequencies we choose to filter out.