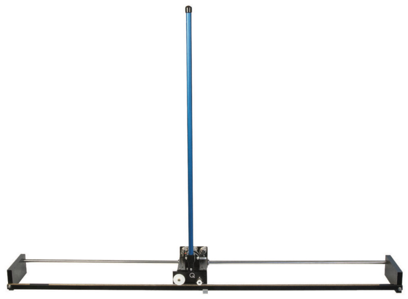
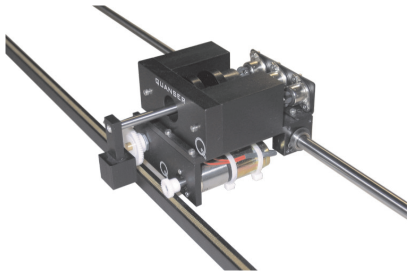
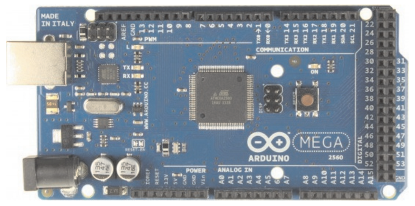
Lab 1 – ECE557F Linear Control Theory
Familiarization With Lab Equipment and Proportional Control

Main concepts of this lab

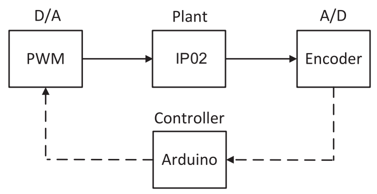
- Structure of Arduino code for real-time control implementation
- Introduction to the experimental setup
- DC Motor actuation from the Arduino
- Reading encoder signals in Arduino
- Rudimentary proportional control of the cart-pendulum

1 Introduction

The laboratories of this course use the Quanser IP02 system and the Arduino Mega 2560 microcontroller.

The IP02 system, showed on the left-hand side of the figure above, is a cart-pendulum system. The cart, shown in the centre, is actuated by a DC motor. An Arduino Mega microcontroller, shown on the right-hand side, is mounted on the back of the cart and controls the motor. There are two sensors in this system: an optical encoder measuring the position of the cart along the track and an optical encoder measuring the pendulum angle.



As shown in the block diagram, the Arduino receives the position of the cart and angle of the pendulum rod as digital signals from the encoders, and it outputs the control input to be applied to the cart. This output is fed to a Cytron 10A Motor Shield mounted on top of the Arduino board. This shield converts the Arduino output into a PWM signal that drives the motor.

In this introductory lab you will familiarize yourself with the Arduino framework. Since Arduino programming is a topic of independent interest, and one on which we cannot dwell upon too much in this course, we have included appendices in this document that you can read for your own interst and/or for future projects you might undertake, e.g., in a capstone course.

2 Organization of this lab

This lab has the following components:

- To prepare for the lab, you need to read this document *in advance of the lab*.
- In the lab, you will perform these activities:
  - You will learn how to apply a voltage to the DC motor (Section 4).
  - You will learn how to read the encoder signal and convert it into the cart displacement (Section 5).
  - You will finally put everything together by implementing a *rudimentary* proportional controller to control the position of the cart (Section 6).

Throughout the lab, there are requests to modify code (Arduino or Matlab). These requests are written in light blue colour.

3 Grading and submission guidelines

This lab *will not be graded* but attendance and active participation are required. There will be a penalty of 2% of the course grade for not attending the lab.

4 Applying a voltage to the DC motor

In this section, you will apply a sinusoidal input voltage u of 1.5V from the Arduino Mega to the motor of the IP02 plant. Follow the procedure described below:

1. Open the Arduino IDE software from the Windows start menu and load the file `lab1_part1.ino`. The Arduino code is known as a `sketch` and is written in `C` code. The sketch contains a program which will be uploaded and run on the Arduino board to output a desired voltage to the IP02 plant. In the steps that follow, you will be guided through each section of the code labeled Section 1 to Section 6. By the end of the procedure you should be able to understand how each section of the program works.

2. Open the `Tools` menu and select `Board`. Then select `Arduino/Genuino Mega or Mega 2560`.

3. Consider Sections 1 and 2 of the Arduino code shown below.

```
 1 #include "Timer.h"
 2 #include "math.h"
 3
 4 /** -------------------------------------------------------------
 5 * SECTION 1
 6 *
 7 * @students
 8 * TODO: configure the amplitude and frequency of your sine wave
 9 * applied as motor voltage; these global variables can be
10 * read and updated throughout the entire duration of the
11 * program
12 *
13 * ------------------------------------------------------------ **/
14 double amp = ; // amplitude of sine wave in Volts
15 double freq = ; // frequncy of sine wave in rad/s
16
17 double u = 0; //Initial voltage value applied to motor
18
19 /** -------------------------------------------------------------
20 * SECTION 2
21 * Do NOT modify
22 *
23 * Sets up timer, encoder, and motor pin constants
24 * ------------------------------------------------------------ **/
25 Timer t;
26 double T_sample = 0.001; // Controller sample time in seconds
27 double T_tot = 0; // Total elapsed time
28
29 // motor control
30 int PWM_PIN = 11; // pin to send the magnitude of applied motor voltage
31 int DIR_PIN = 8; // pin to send the polarity of applied motor voltage
32 int duty_cycle; // duty cycle of pwm to send to motor
```

   In these two sections, all necessary libraries, variables and pins are defined for the experiment. The amplitude and frequency of the sine wave control input u are defined on lines 14 and 15. Set the amplitude to 1.5V and frequency to 1 rad/s. A timer called `t` of the `Timer` class has been added on line 25 and takes care of all scheduled tasks in the program. The variable `T_tot` keeps track of the total experiment time and is initialized to zero. `T_sample` is the sampling interval at which the voltage u applied to the cart motor is updated.

4. In Section 3, shown below, the method `void setup()` is implemented. For any Arduino sketch, this function is run once, just after the code is uploaded to the Arduino board, and doesn't return any value.

```
34 /** -------------------------------------------------------------
35 * SECTION 3
36 * Do NOT modify
37 *
38 * Setup function which sets the mode for Arduino pins
39 * ------------------------------------------------------------ **/
40 void setup() {
41   Serial.begin(115200); // open serial connection for writing
42   configurePins();
43   t.every(T_sample*1000, applyVoltage); // Perform applyVoltage every T_sample
44                  // t.every takes in ms
45 }
```

```
100 void configurePins(){
101   pinMode(PWM_PIN, OUTPUT);
102   pinMode(DIR_PIN, OUTPUT);
103
104   TCCR1A = _BV(COM1A1) | _BV(WGM21) | _BV(WGM20);
105   TCCR1B = _BV(CS10);
106   OCR1A = 0;
107 }
```

The `setup()` function does the following:

- In a call to `void configurePins()`, the function `pinMode` is used to set pins 11 and 8 (named `PWM_PIN` and `DIR_PIN` respectively) as PWM outputs from the Arduino board. The voltage output from these pins is a PWM signal which is amplified by the motor driver and delivered to the IP02 plant at the desired voltage level. In particular, pin 11 outputs a PWM signal with the desired duty cycle. However, the signal varies between low=0V and high=5V and therefore always outputs a positive value. By setting pin 8 to either low or high, the sign of the PWM signal output between pins 8 and 11 is made either positive or negative as desired in order to drive the cart in either the forward or reverse direction. The register `OCR1A` on line 106 is used to set the PWM duty cycle on pin 11 in fast PWM mode. In the code, it is initialized to zero. It takes values from 0 to 1024 where 0 corresponds to a 0% duty cycle and 1024 corresponds to a 100% duty cycle.
- On line 43, `t.every(T_sample*1000, applyVoltage)` performs a method call on the `Timer` instance `t`. This invokes the scheduled function `applyVoltage()` every `T_sample` seconds. In the function `applyVoltage()` the output voltage from the Arduino board to the IP02 plant is updated.

5. Section 4, shown below, is the main processing loop, an infinite loop running during any Arduino sketch.

```
47 /** ------------------------------------------------------------
48 * SECTION 4
49 * Do NOT modify
50 *
51 * main loop() which executes over and over again on the Arduino
52 * ---------------------------------------------------------- **/
53 void loop(){
54   t.update(); // update timer, which triggers scheduled functions
55   // according the t.every(...) set in the setup() function
56 }
```

The method call `t.update()` tells the `Timer` instance `t` to trigger any scheduled tasks. In this case, `applyVoltage()` is the only scheduled task.

6. In Section 5, shown on the following page, the output voltage u of the Arduino board is updated. This is done in the scheduled task `applyVoltage()` which is invoked every `T_sample` seconds by the `Timer` instance `t`.

The PWM signal is updated in the function `void mapVoltageToMotorShield()`, which is called at the end of `applyVoltage()`. On line 84, the required voltage u is mapped linearly to the range (-1024, 1024) whose magnitude corresponds to the PWM duty cycle in the register `OCR1A` and sign in pin `DIR_PIN` (pin 8). A value of 1024 corresponds to a 100% duty cycle which will result in an output of 11.75V from the motor driver. The magnitude of the voltage is saturated after line 86 to not exceed motor voltage limits of 5.875V. On lines 91 to 97, the desired duty cycle is written to the register `OCR1A`. The sign of the PWM signal is set through `DIR_PIN` (pin 8) using the `digitalWrite()` function.

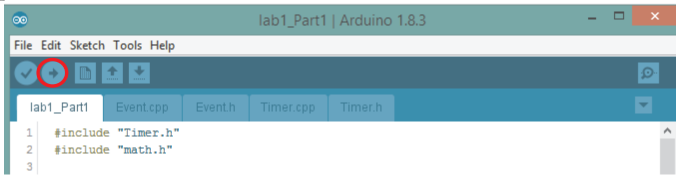
```
58 /** ------------------------------------------------------------
59 * SECTION 5
60 *
61 * @students
62 * TODO: compute the control input (voltage applied to the motor)
63 * which is applied to the motor in the call to
64 * mapVoltageToMotorShield()
65 *
66 * ---------------------------------------------------------- **/
```

```
void applyVoltage(){
  T_tot = T_tot + T_sample; // Total experiment time update

  u = ;

  mapVoltageToMotorShield();
}

/** ----------------------------------------------------------
* SECTION 6
* Do NOT modify
*
* Utility functions for interfacing with the motor shield and
* setting up pins
* ---------------------------------------------------------- **/
void mapVoltageToMotorShield(){
  //Mapping between required voltage and 11.75V motor shield
  duty_cycle = round(u/11.75 * 1024);
  if (duty_cycle > 512 ){ //Saturation to not exceed motor voltage limits of 5.875 V
    duty_cycle = 512; //motor moves left
  } else if (duty_cycle < -512 ){
    duty_cycle = -512; //motor moves right
  }

  if (duty_cycle > 0){ // Update the direction of motor based on the sign of duty_cycle
    digitalWrite(DIR_PIN, HIGH);
    OCR1A = duty_cycle;
  } else if (duty_cycle <= 0){
    digitalWrite(DIR_PIN, LOW);
    OCR1A = -duty_cycle;
  }
}
```

In the `applyVoltage()` function, update the value of the control input u to the desired sinusoidal output in terms of the `amp` and `freq` that you set earlier, as well as the current experiment time `T_tot`.

7. We are almost ready to run the experiment. For safety, we need to be able to stop the experiment. To this end, create a new empty Arduino file (Ctrl+n) and save it as `stop_experiment.ino`. When you wish to stop the experiment, upload `stop_experiment.ino`. The cart will then stop. Pulling out the Arduino USB cable from the computer side is an alternative stopping method. There is also a physical reset button on the motor shield.

8. Before the demonstration, the cart should be positioned near the middle so that the pendulum rod is at least 40 centimeters from the right end of the toothed section (do not account for the untoothed section right beside the track side wall). In the Arduino Software, make sure the correct `Board` and `Port` are selected under the Tools tab. When the cart is in position, upload the Arduino Code to the Arduino Mega Board with the upload button (or press Ctrl+u)



```
#include "Timer.h"
#include "math.h"
```

9. The cart should follow a sinusoidal motion after a short delay. Repeat the experiment by varying the amplitude and frequency of the sinusoid within 20% of their nominal values set above. Understand the impact of these variations on the amplitude and frequency of the cart motion.

5 Reading the encoder signals

In this section you will monitor and measure the actual position of the IP02 cart and pendulum rod angle as sensed by the cart and pendulum rotary encoders. Section C in the Appendix explains the operation of the encoders. Follow the procedure described below:

1. Open the Arduino IDE software and load the file `lab1_part2.ino`.

2. Open the `Tools` menu and select `Board`. Then select `Arduino/Genuino Mega or Mega 2560`.

3. Consider Section 2 of the Arduino code illustrated below.

```
12 /** -----------------------------------------------------------
13 * SECTION 2
14 * Do NOT modify
15 *
16 * Sets up timer, encoder, and motor pin constants
17 * ---------------------------------------------------------- **/
18 Timer t;
19 double T_sample = 0.001; // Controller sample time in seconds
20 double T_plot = 0.01; // writing to serial every T_plot seconds
21 double T_tot = 0; // Total elapsed time in seconds
22 char serial_delim = ','; // delimeter for serial interface communications
23
24 enum PinAssignments { //Cart and Pendulum Encoder pins
25   //Cart Encoder signals A and B
26   encoderPinA = 2,// brown wire, blk wire gnd
27   encoderPinB = 3,//green wire, red wire 5v
28
29   //Pendulum Encoder signals A and B
30   encoderPinA_P = 18, //lab Din5 blk wire
31   encoderPinB_P = 19, //lab Din5 orange wire
32 };
33
34 volatile int encoderPos = 0; //count cart ticks
35 volatile int encoderPos_P = 0; //count pendulum ticks
36
37 //Constants to map encoder clicks to position and angle respectively
38 const float pi = 3.14;
39 double K_encoder = 2.2749*0.00001;
40 double K_encoder_theta = 2*pi*0.000244140625;
41
42 //Boolean variables for determining encoder position changes
43 bool A_set = false;
44 bool B_set = false;
45 bool A_set_P = false;
46 bool B_set_P = false;
```

In lines 24 to 32, the cart encoder and pendulum encoder pins are defined. Recall that each encoder has both an A and a B channel. When these channels transition from low-to-high or from high-to-low, the encoder experiences a *click* which translates to about a 0.09 degree turn. The number of encoder clicks are stored in the variables `encoderPos` for the cart encoder and in `encoderPos_P` for the pendulum encoder used to measure the cart displacement and pendulum angle. Both of these variables are initialized to zero on lines 34 and 35. Therefore, displacement of the cart and angle of the pendulum are measured relative to their initial values.

4. In Section 3 of the Arduino code, shown below, the method `void setup()` is implemented.

```
48 /** -----------------------------------------------------------
49 * SECTION 3
50 * Do NOT modify
51 *
52 * Setup function which sets the mode for Arduino pins
53 * ---------------------------------------------------------- **/
54 void setup() {
55   Serial.begin(115200); // open serial connection for writing
56   configurePins(); // sets all pins and modes
57
58   t.every(T_sample*1000, takeReading);
59   t.every(T_plot*1000, writeToSerial);
60 }
```

```
101 void configurePins(){
102   //Set up Cart Encoder pins - Arduino input
103   pinMode(encoderPinA, INPUT);
104   pinMode(encoderPinB, INPUT);
105   digitalWrite(encoderPinA, HIGH); // turn on pullup resistor
106   digitalWrite(encoderPinB, HIGH); // turn on pullup resistor
107   attachInterrupt(0, doEncoderA, CHANGE);
108   attachInterrupt(1, doEncoderB, CHANGE);
109
110   //Set up Pendulum Encoder pins - Arduino input
111   pinMode(encoderPinA_P, INPUT);
112   pinMode(encoderPinB_P, INPUT);
113   digitalWrite(encoderPinA_P, HIGH); // turn on pullup resistor
114   digitalWrite(encoderPinB_P, HIGH); // turn on pullup resistor
115|  attachInterrupt(5, doEncoderA_P, CHANGE); //int.5 on pin18
116|  attachInterrupt(4, doEncoderB_P, CHANGE); //int.4 on pin 19
117| }
```

At line 43 and 51 respectively, the A and B channels for the cart and pendulum encoder are set up as input pins on the Arduino board. Interrupts are set up to be triggered whenever the encoder A or B channels perform a low-to-high/high-to-low transition (triggering event). In the case of the cart encoder, the interrupt handlers are called `doEncoderA` and `doEncoderB` for its A and B channel respectively. A scheduled function `writeToSerial()` is set to run every `T_plot` seconds. This function writes to a serial port channel to send the encoder measurements to Matlab, where the data will be plotted. The serial communication rate with Matlab is set to 115200 bps.

5. In Section 6 of the Arduino code, the `writeToSerial()` function is implemented, as shown below.

```
85| /** ---------------------------------------------------------------
86| * SECTION 6
87| * Do NOT modify
88| *
89| * Utility functions writing to the serial port and setting up
90| * pins
91| * ------------------------------------------------------------ **/
92| void writeToSerial(){
93|   Serial.print(z_measured); // cart position
94|   Serial.print(serial_delim);
95|   Serial.print(theta_measured); // pendulum angle
96|
97|   Serial.println(); // prints a new line to the serial port, indicating
98|   // the end of the data transmitted for the current time step
99| }
```

`writeToSerial()` writes the cart position (`z_measured`) and pendulum angle (`theta_measured`) to a serial port that we will read from in MATLAB. Since the variables `encoderPos` and `encoderPos_P` store the position and angle in terms of clicks, they must be transformed into meters and radians respectively. This is achieved through the scaling factors `K_encoder` and `K_encoder_P`.

6. In Section 5 of the Arduino code, the function `void takeReading()` is implemented. `takeReading()` is invoked by the `Timer` instance `t` every `T_sample` seconds. Modify the `takeReading` function to update the global variables `z_measured` and `theta_measured` (these variables are declared in Section 1 of the Arduino code).

```
73| /** ---------------------------------------------------------------
74| * SECTION 5
75| *
76| * @students
77| * TODO: update the global variables z_measured and theta_measured
78| * based on the values measured by the encoders
79| *
80| * ------------------------------------------------------------ **/
81| void takeReading(){
82|   // update z_measured and theta_measured here
83| }
```

7. In Section 7 of the Arduino code, shown below, the interrupt handler functions are implemented.

```
119| /** ---------------------------------------------------------------
120| * SECTION 7
121| * Do NOT modify
122| *
123| * Interrupts for reading cart position/pendulum angle
124| * encoders and updating encoderPos / encoderPos_P
125| * ------------------------------------------------------------ **/
126| void doEncoderA(){ // Interrupt on A changing state
127|   // Test transition
128|   A_set = digitalRead(encoderPinA) == HIGH;
129|   // and adjust counter + if A leads B
130|   encoderPos += (A_set != B_set) ? +1 : -1;
131| }
132|
133| void doEncoderB(){ // Interrupt on B changing state
134|   // Test transition
135|   B_set = digitalRead(encoderPinB) == HIGH;
```

```
  // and adjust counter + if B follows A
  encoderPos += (A_set == B_set) ? +1 : -1;
}

void doEncoderA_P(){ // Interrupt on A changing state
  // Test transition
  A_set_P = digitalRead(encoderPinA_P) == HIGH;
  // and adjust counter + if A leads B
  encoderPos_P += (A_set_P != B_set_P) ? +1 : -1;
}

void doEncoderB_P(){ // Interrupt on B changing state
  // Test transition
  B_set_P = digitalRead(encoderPinB_P) == HIGH;
  // and adjust counter + if B follows A
  encoderPos_P += (A_set_P == B_set_P) ? +1 : -1;
}
```

When the code finds a low-to-high/ high-to-low transition on the A channel, it checks to see if the B channel transitioned then increments/decrements the click variable `encoderPos` or `encoderPos_P` to account for the direction that the encoder must be turning (and vice-versa for the B channel). The details of this algorithm are found in Section C.

8. When the cart is in position, upload the Arduino Code to the Arduino Mega Board with the upload button (or press Ctrl+u)

9. Open the Matlab file `lab1_plot.m`, located in the parent folder of the Arduino sketch folder. Change the value stored in the variable `port` on line 10, which is set to "COM3", to the serial port that is being used by the Arduino. Set `stop_time` to the total time you'd like the experiment to be plotted for. Run the m-file, which will start reading values from the serial port given by `port`.

10. Move the cart back and forth with your hand.

11. MATLAB will plot the cart position versus time automatically.

12. If the Arduino cannot connect, clear the serial object in MATLAB with the following command:

    `clear serial;`

13. In the plot file `lab1_plot.m`, set `POSITION_EXPERIMENT` to `false` in order to plot the pendulum angle vs. time live. Repeat the experiment in Part 2, but this time instead of moving the cart with your hand, twist the pendulum rod.

6 Putting things together: proportional control of the cart

Now that you understand how to send control input signals to the actuator (the DC motor) and read the measurement outputs (encoder signals) from the plant, you'll try your hand at some rudimentary control. In the next labs you will use control theory in a more thoughtful and systematic manner.

The basic controller we will test is a proportional controller $u = -Ky$, where y is the cart position in metres and $K > 0$ is the controller gain to be tuned by you. This controller attempts to regulate the position of the cart to zero, i.e., in the middle of the track, without caring about the behaviour of the pendulum. To what extent will this controller work, and why would it work are not topics of concern for the scope of this lab. You will just test this controller and see what you get.

Follow the procedure described below:

1. Open the Arduino file `lab1_Pcontrol.ino`.

2. Open the `Tools` menu and select `Board`. Then select `Arduino/Genuino Mega or Mega 2560`.

3. In Section 1 of the Arduino code, define the proportional control gain K and set it initially equal to 5.
4. In Section 5 of the Arduino code, include the updated measurements for `z_measured` and `theta_measured` you implemented in `takeReading` from `lab1_part2.ino`. Also, update the control input to the motor using the feedback control law $u = -Ky$.
5. In Section 6 of the Arduino code, Modify the `writeToSerial` function to also write the control input (the motor voltage u) to the serial port.
6. When the cart is in position with the pendulum hanging down and completely motionless, upload the Arduino Code to the Arduino Mega Board with the upload button (or press Ctrl+u).
7. Modify the Matlab file `lab1_plot.m` to also plot the control input u versus time.
8. Set `POSITION_EXPERIMENT` to `true` and run the m-file.
9. Gradually increase K and repeat the procedure above. Observe the behaviour of the closed-loop system. Show the TA the best results you get.

Appendix A Arduino Mega

Arduino is a microcontroller that can provide extremely portable computing to circuits, enabling them to read both analog and digital inputs, perform preliminary calculations, and produce a variety of outputs. It's a physical computing platform, and a development environment for applying small-scale logic and software in circuits.

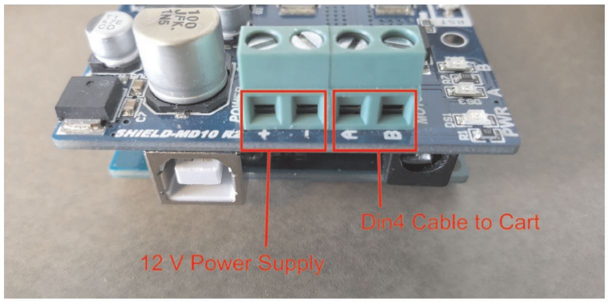
The versatility of these microcontrollers can be used to develop interactive circuits, that can receive both analog and digital inputs from a wide variety of switches and/or sensors, as well as control a variety of LEDs, motors, and other physical outputs. Arduino boards can also be used either as the primary processor in implementing logic in circuits, or as nodes that can communicate with software in other devices.

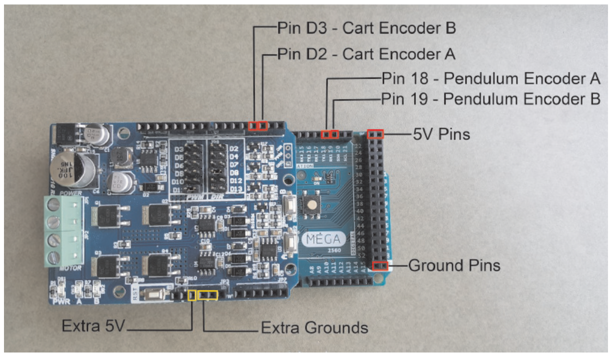
Arduino's open-source IDE (Integrated Development Environment) can be used to develop and publish software for its microcontrollers, and is available for download on Arduino's website.

The Arduino Mega is a microcontroller board based on the ATmega2560 processor. It has 54 digital input/output pins (of which 14 can be used as PWM outputs), 16 analog inputs, a 16 MHz ceramic resonator, a USB connection, a power jack, and a reset button. It contains everything needed to support the microcontroller; all that is required to power the microcontroller is a connection to a computer with a USB cable or to an AC-to-DC adapter or battery.

One of the features that make the Arduino a suitable tool to control a DC motor is its ability to generate Pulse Width Modulation (PWM) signals. PWM signals, as detailed in Section B, can be interpreted as a simple DC voltage under specific conditions. Motors, however, typically require voltages that exceed what can be provided by the controlling logic circuits such as those on the Arduino board. For this reason, a motor driver is used to amplify the voltage to the desired value.

In this experiment, the Cytron 10A Motor Shield is used and is mounted on top of the Arduino Mega. The ports of the motor driver are shown below. The 12 V port powers the motor driver while the Din4 cable port outputs the desired PWM voltage to drive the cart motor on the IP02 system. The encoder ports on the Arduino board are illustrated below on the right-hand side. Each encoder has two channels, denoted A and B, each with their own input port on the Arduino Mega.

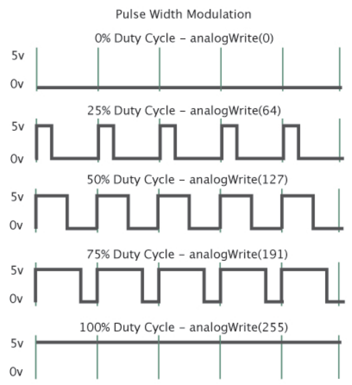




Appendix B Pulse Width Modulation (PWM)

Pulse Width Modulation, or PWM, is a technique for getting analog results with digital means. Digital control is used to create a square wave, a signal switched between on and off. This on-off pattern can simulate DC signals with a voltage between the full on voltage (5 Volts) and off voltage (0 Volts) by changing the portion of the time the signal spends on versus the time that the signal spends off. The duration of "on time" is called the "pulse width". To get varying analog values, you change, or modulate, that pulse width.

One thing to keep in mind is that, without a high enough PWM frequency, this effect deteriorates. Also, the Arduino's default PWM frequency ($\approx 500Hz$) is much lower than the desired frequency ($\approx$ a few KHz). As a result, the frequency of the Arduino's timers had to be explicitly increased using the timers' registers, and the PWM duty cycle had to be altered using more complex methods. This is discussed in more detail in the following sections.



In the image above, the vertical spikes in the top graph represent a regular time period. The period is given as Period=1/(PWM Frequency). In other words, with Arduino's PWM frequency at about 500Hz, the spikes would measure 2 milliseconds each. A call to analogWrite() is on a scale of 0 - 255, such that analogWrite(255) induces a 100% duty cycle (always on), while analogWrite(127) induces a 50% duty cycle (pulse width = 1ms) for example.

B.1 The ATmega 2560 Timers and using the PWM Registers

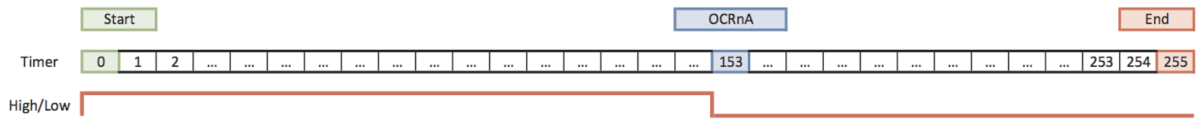
The ATmega2560 has five timers known as Timer 0, Timer 1, ... Timer 4. Each has a corresponding compare register. When the timer reaches the compare register value, the corresponding output is toggled.

Each of the timers has a register that controls the frequency, and the frequency is set by dividing the system clock frequency of 16MHz by a prescale factor, determined by the register's value. The prescale factor may include values such as 1, 8, 64, 256, or 1024. The timers have several different modes. The main two PWM modes are "Fast PWM" and "Phase- correct PWM". In this lab, only "Fast PWM" will be discussed.

Timer Registers are used to control each timer. The Timer/Counter Control Registers TCCRnA and TCCRnB hold the main control bits for timer n. In the section below, we present a code-snippet that sets control bits for timer 1 using registers TCCR1A and TCCR1B.

B.2 Fast PWM mode

In the simplest PWM mode, the timer repeatedly counts from 0 to 255 for an 8-bit timer (or from 0 to 1024 for a 10-bit timer). The output turns on when the timer is at 0, and turns off when the timer matches the Output Compare Register (OCRnA). The higher the value in the output compare register, the higher the duty cycle.



The following code fragment sets up fast PWM on pins 8 and 11 (Timer 1). To give an example of the use of register settings:

```
1. int PWM_A = 11;
```

```
2. int DIR_A = 8;

3. pinMode(PWM_A, OUTPUT);
4. pinMode(DIR_A, OUTPUT);

5. TCCR1A = _BV(COM1A1) | _BV(WGM21) | _BV(WGM20);
6. TCCR1B = _BV(CS10);
7. OCR1A = 800;
```

Without dwelling on the meaning of each variable and line of code, the snippet above performs the following operations,

- On lines 1 to 4, pins 8 and 11 are set up as outputs on Arduino for PWM.
- On line 5 the control bits for timer 1 are set in register TCCR1A. It selects fast PWM with a 10-bit timer (counts from 0 to 1024 for each pulse width cycle) and provides non-inverted PWM for pin 11.
- On line 6 the control bits for timer 1 are set in register TCCR1B. It sets the prescaler to 1 (Timer Clock = System Clock).
- The output compare register OCR1A is arbitrarily set to 800 on line 7 to control the PWM duty cycle on Arduino Mega pin 11. The computation of the duty cycle is described below.

On the Arduino Mega, these values yield:

- Output A PWM frequency: 16 MHz / 1 / 1024 = 15625 Hz
- Output A PWM duty cycle: (800+1) / 1024 = 78.2%

The output frequency is the 16 MHz system clock frequency, divided by the prescaler value (1), divided by the 1024 cycles it takes for the timer to wrap around. As for the duty cycle, Arduino has the convention that the duty cycle is calculated based upon the value of the output compare register OCR1A+1.

B.3 The Relationship Between PWM and DC

Given that the DC motor acts like a filter, the observed DC output at the motor's side is simply the average of the PWM signal.

So, if the peak-to-peak voltage of the PWM signal = 5V, and the Duty Cycle is 40%, the corresponding DC output observed by the motor is 40% of the 5V , which equals 2V. Note that the PWM frequency should be high enough for the motor to behave in the desired manner. In order to get a certain DC output, lets say 3V, we start by determining the duty cycle of the PWM by dividing the desired output over the peak-to-peak voltage of the PWM signal, 3V / 5V = 0.6 = 60%. Then the value in the output compare register OCR1A = (0.6 * 1024) - 1 $\approx$ 613.

Appendix C Rotary encoders

A rotary encoder is a device for measuring angular movement and is used to measure the rotation of motors or rotating shafts. The simplest form of encoders utilizes a light source and two photodetectors (light sensors) within close proximity of one another behind a slotted disk; those photodetectors form the channel waveforms A and B.

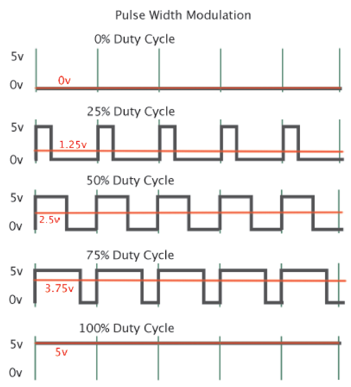
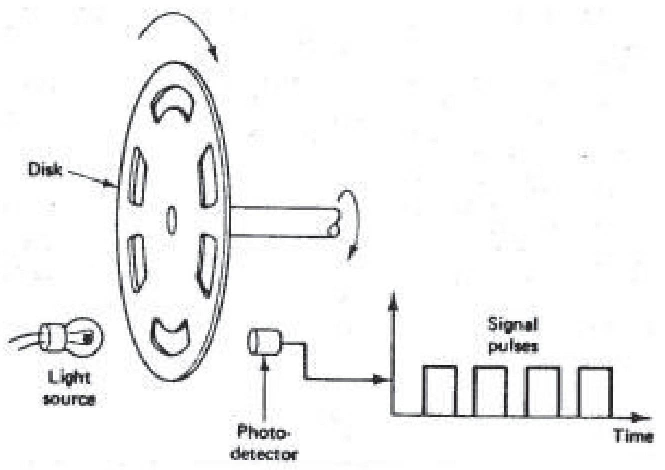
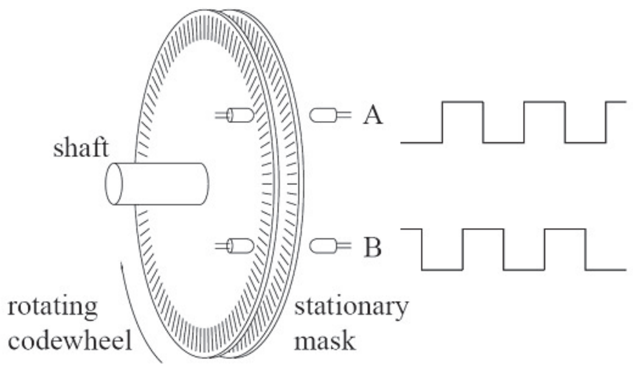


Figure 1: PWM at different duty cycles and the corresponding DC output



(a) Components of an optical encoder.



(b) A & B waveforms.

Figure 2: Illustration of optical encoders

The slotted disk is attached to the rotating shaft, and as it rotates the slots allow the light to reach the photodetectors turning their signals on and off. Reading those waveforms simultaneously can provide a very precise measurement of the angular movement. We discuss how to read the encoder waveforms in the interrupts section.

Below is an image showing the waveforms of the A & B channels of an encoder.

When the code finds a low-to-high/ high-to-low transition on the A channel, it checks to see if the B channel transitioned then increments/decrements the variable to account for the direction that the encoder must be turning in order to generate the waveform found (and visversa for the B channel).

Forward direction (CW):

- (A=Low & B=High) then (A=High & B=High),
- (A=High & B=Low) then (A=Low & B=Low)

Reverse direction (CCW):

- (A=High & B=Low) then (A=High & B=High),
- (A=Low & B=High) then (A=Low & B=Low)

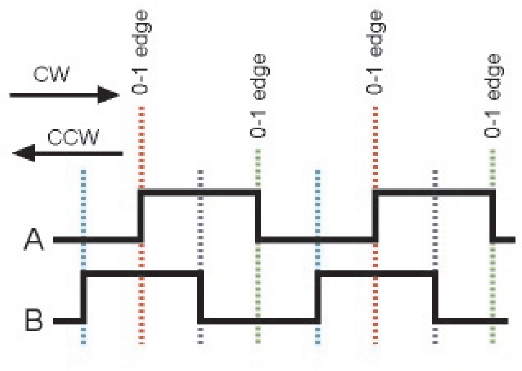


Figure 3: Waveforms of the A & B channels of an encoder

C.1 Using Interrupts to Read Rotary Encoders

An interrupt, as the name hints, is a signal emitted from hardware or software events to the processor, allowing the processor to pause its current activities momentarily to perform some (usually high-priority) processing based on the triggering event. The processor responds by suspending its current activities, saving its state, and executing a method or function called an interrupt handler to deal with the event. This interruption is temporary, and after the interrupt handler finishes, the processor resumes execution of the previous thread.

One could attach interrupts to the A & B channels. When the Arduino sees a change on the A channel, it immediately skips to the “doEncoder" function, which parses out both the low-to-high and the high-to-low edges, and the same happens for the B channel. The “doEncoder" function increments/decrements a click variable `encoderPos` to account for the direction that the encoder must be turning.

Using interrupts to read a rotary encoder is very suitable, since the interrupt service routine (a function) is very fast.