MAE106 Laboratory Exercises Lab # 2 - Introduction to data acquisition systems and filters

University of California, Irvine
Department of Mechanical and Aerospace Engineering

Goals

- To learn how to read an analog signal into a microcontroller and process it, then send out a signal.
- To learn about analog and digital filters and how they can be applied to robotics and your final project.
- To observe the dynamics of a first-order system.
- To learn about pulse-width modulation.

Parts & equipment

Qty	Part/Equipment
1	Breadboard
2	10KOhm Potentiometers
Various	Wire
1	Function generator (Trainer kit)
1	Oscilloscope
1	Multimeter
1	Seedduino board
1	Capacitor

Introduction

When dealing with electromechanical systems we often need to read signals from sensors that can convert physical phenomena into electrical signals (i.e. potentiometers, acceloremeters, thermometers, etc.), process them, and take actions based on these readings. A key part of this process involves being able to read these analog signals and transform them into digital ones so that a computer can manipulate them. To do this, we will use an inexpensive data acquisition system (DAQ): the Seeeduino microcontroller.

In a common configuration for DAQ systems (Figure 1) - and one we will be using throughout this class - we first want to read from one or multiple sensors. We will do so by using the DAQ system (i.e. seeeduino) to convert the analog (i.e. continuous) signal into a digital (i.e. 0's and 1's) signal; this is known as analog-to-digital (A/D) conversion.

Once the signal is read into the microcontroller we can now process it and finally create new signals to control outputs such as lights or motors; this is known as digital-to-analog (D/A) conversion.

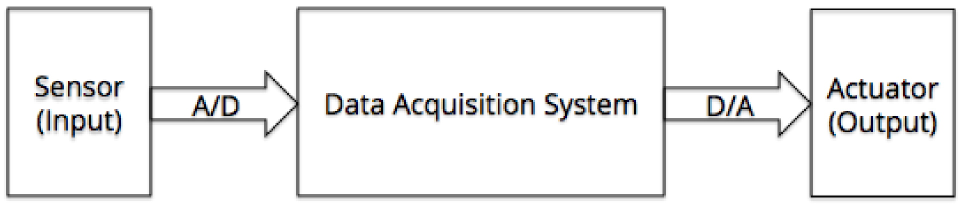


Figure 1. Common configuration for a DAQ system

Part I: Analog filters and Pulse-width modulation

In many cases when dealing with real-world systems the input and output signals that we use will require some type of pre- and/or post-processing. A key tool in this processing is filtering.

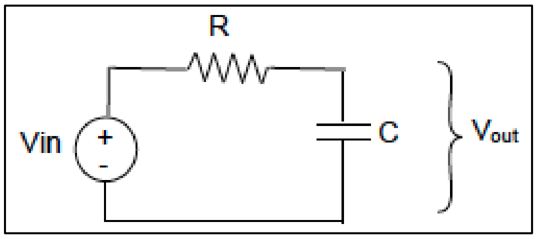
Signal processing filters are an important part of engineering design. They are often found in stereos (cross-overs, graphic equalizers, etc.), control systems (for cars and planes, to clean up sensor readings from strain gauges, tachometers, potentiometers, etc.), and many other applications. In control systems, filters can help remove unwanted high-frequency noise that may adversely affect the controller. Filters are also important conceptually because we can view any system as a filter. For example, you can view the steering system of a new car in terms of how it responds to low, medium, and high frequency inputs. Understanding how systems respond to different input frequencies requires understanding how filters work. So, in summary, as a mechanical engineer designer, it is likely that you will want to use filters in your designs, or at least think about the devices you build as filters.

In this lab, we will study the RC circuit, which can be used as a low-pass or high-pass filter. Low-pass filters attenuate high frequency signals (i.e. reduce them in amplitude), but leave low frequency signals relatively unchanged. Low pass filters are often useful for filtering out high-frequency noise (due to electromagnetic interference from the lights or radio signals, for example.) Also, many objects and systems in the world act like low-pass filters. For example, anything with mass acts like a low pass filter in the way any force applied to it gets turned into a change in position of the object (i.e. if you push very slowly on a block of ice, it moves a lot back and forth, but if you push at high frequency, such as vibrating it, it does not move as far back and forth). In contrast, high pass filters attenuate low frequency signals, so they are used to filter low frequency noise from signals. They are used in many applications as well, such as filtering low frequency noise from electrodes measuring muscle or brain activity.

First-Order Linear Systems: This lab also provides a chance for you to measure how a first-order linear system behaves in the time and frequency domains. The RC circuit is a first-order system (i.e. it is described by a first-order differential equation). The mathematics behind a first-order system are extremely common in engineering, so measuring an actual first-order system will give you experience and intuition about how any first-order system behaves, whether it be electrical, mechanical, chemical, or biological.

Low-pass filter

The circuit shown in Figure 2 is a low-pass filter. Thus, it tends to get rid of any high frequencies on the input Vin, just allowing low frequencies through to the output Vout. To get some physical intuition why, consider the response of the circuit a square wave voltage input. A square wave is a signal that changes very rapidly for short periods of time (when the voltage jumps up, or when it jumps down). Based on Fourier analysis, in which we view a signal as a sum of sinusoids at different frequencies, we know that these rapid changes are enabled by very high frequency sinusoids contained in the square wave.



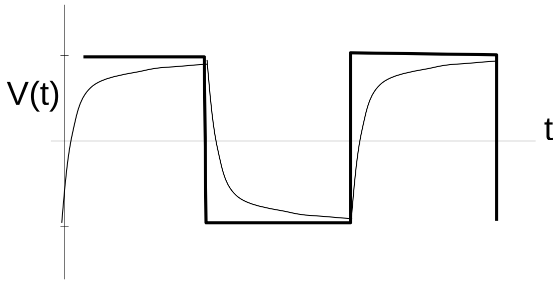


Figure 2. Low-pass filter

Now, the RC filter circuit filters out these high frequency sinusoids, and therefore the rapid changes. In terms of the motion of charges in the circuit, the capacitor (C) acts as a charge bucket, which is alternately charged (filled) and discharged by the square wave input Vin. Essentially, the capacitor is charged by Vin until the voltage across the capacitor matches that of Vin. If Vin is then switched off to a lower voltage value, C begins to discharge through R so that Vout heads toward that low value again. It takes time, however, for the capacitor to charge and discharge through the resistor. That is, the capacitor has dynamics that slow down Vout and prevent it from exactly following Vin.

The differential equation that relates Vout to a step change in Vin can be derived from KCL and KVL for the RC circuit and is given by:

$$V_{out} = V_{in}\left(1 - e^{-t/RC}\right)$$

where τ=RC is the time constant of the system. Thus, because of the first order dynamics, the net effect of the low-pass filter is to get rid of the rapid transitions in voltage in the square wave.

Time Constant

The time constant is an important descriptor of the circuit, and of all first order, linear systems in general. Note that when time t elapses long enough to equal τ, then we get:

$$V_{out} = V_{in}(1 - e^{-1}) = 0.63{*}V_{in}$$

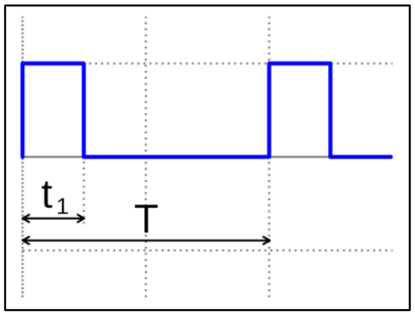
That is, after time as elapsed for a duration equal to one time constant τ, the output has gone 63% of the way toward its final value, Vin. Here is a more formal definition of time constant:

"In physics and engineering, the **time constant**, usually denoted by the Greek letter τ (tau), is the parameter characterizing the response to a step input of a first-order, linear time-invariant (LTI) system. The time constant is the main characteristic unit of a first-order LTI (linear time-invariant) system. Physically, the constant represents the time it takes the system's step response to reach $1 - 1/e \approx 63.2\,\%$ of its final (asymptotic) value *for systems that **in**crease in value* (say from a step increase), or it can represent the time for systems to decrease in value to $1/e \approx 36.8\,\%$ (say from a step **de**crease)." (Wikipedia). In other words, after one time constant, the system has gone 63% of the way to where it output is going.

The time constant also relates to the filtering properties of the circuit. For the low-pass filter, the frequency at which the filter starts to filter out higher frequencies is known as the "cut-off frequency". The cut-off frequency for the first-order, low-pass filter is $\omega_c = \frac{1}{\tau}$ rad/sec, or $f_c = \frac{1}{2\pi\tau}$ Hz.

Pulse-width modulation (PWM)

Another use of low pass filters is in pulse-width modulation. Some digital boards cannot output a fixed constant voltage other than at a few pre-selected levels. In the case of the Seeeduino board these levels are only 0 V or 5 V. However, some applications will require the use of a broader range of voltages. Imagine for instance that you want to make an LED fade or (as we will use later in coming labs) to spin a motor at a desired velocity. This is when Pulse-Width Modulation (PWM) is useful.



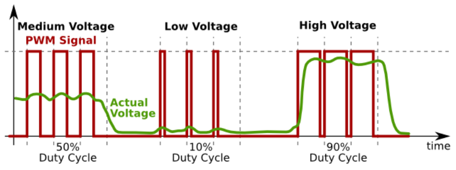


Figure 3. t1 - pulse duration, T - period duration. t1/T = duty cycle

PWM consists in sending a set of pulses at high frequency. We divide each period in two intervals, the ON part (pulse) and the OFF part. The term "duty cycle" describes the percentage of ON time compared to a full period. Therefore, a signal with a 0% duty cycle is always OFF while one with a 100% duty cycle is always ON. If we low pass filter this PWM signal with a filter with an appropriate time constant, the resulting signal is approximately a constant voltage that is proportional to the duty cycle. We can then use this resulting signal similar to an analog output. Note that because many physical objects (like motors) already act like low pass filters, you might not even need to incorporate a separate low pass filter – the object itself will serve that purpose.

For this portion of the lab we will use the microcontroller to read an input signal (the angle of the potentiometer), process it, and generate a PWM output signal proportional to this angle. We will also filter the PWM signal through an RC-low-pass filter and study both signals with the oscilloscope. Begin by replicating the circuit in Figure 4 and then use the code provided to read, process, and output the required signals. Note that this circuit incorporates the same RC circuit as shown in Figure 2, but the resistor is the internal resistor of the pot, which we will be able to change using the pot.

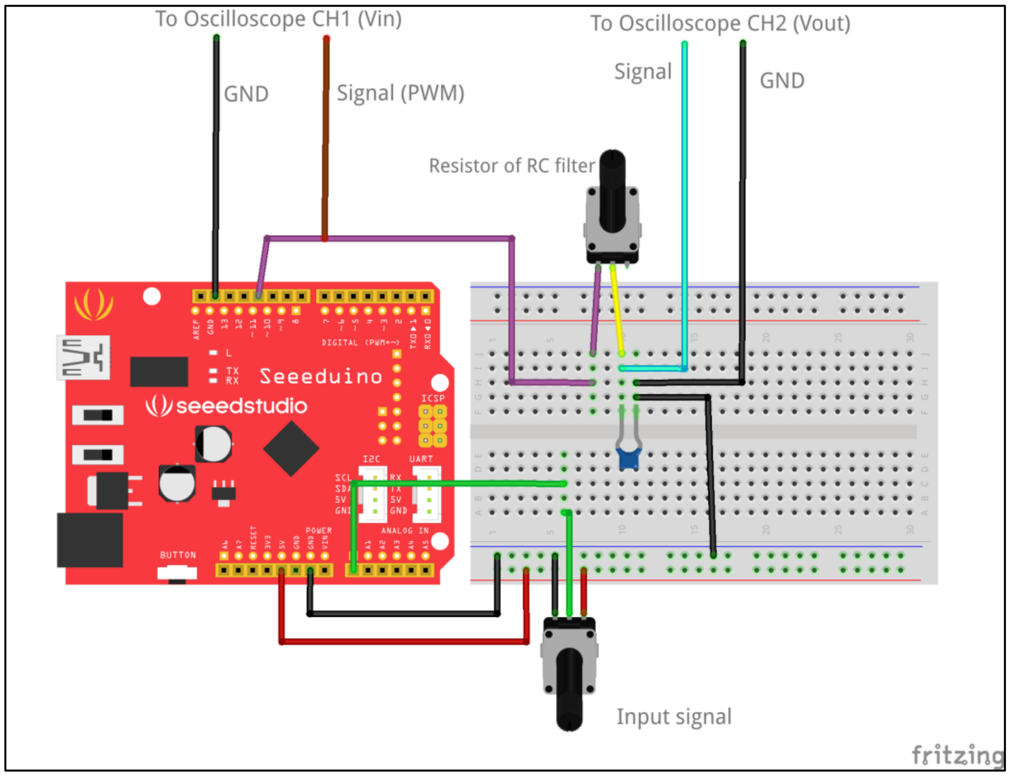


Figure 4. Circuit for Part I.

Code - Part I

```
// variables for creating a fixed time loop
unsigned long startTime = 0;
unsigned long waitTime = 0;
unsigned long elapsedTime = 0;
long interval = 1000;

//readValues
float potentiometer = 0;
int pwmPin = 11;

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
}

void loop() {
  startTime = micros();
  // put your code in here
  potentiometer = analogRead(A0);

  // analogRead: reads outputs between 0 and 1023,
  // analogWrite: writes values between 0 and 255.
  analogWrite(pwmPin, potentiometer * 255/1023);
  Serial.println(potentiometer);

  // do not put your code below here
  elapsedTime = micros() - startTime;
  waitTime = interval - elapsedTime;
  if (waitTime > 0){
    delayMicroseconds(waitTime);
 }

}
```

Sampling Rate

As stated in the Introduction, the Seeeduino is sampling the input signal from one of the pots using an A/D converter. The "sampling rate" is the frequency at which the Seeduino is taking sample. For example, if you programmed the Seeduino to have a sample rate of 100 Hz, it would take 100 samples per second. Clearly, it is important to sample at a high enough rate to be able to accurately sense the signal you are sampling. There is a mathematical theorem, the Nyquist Sampling Theorem, that says you need to sample at least at 2 times the maximum frequency in the signal you are sampling, to be able to accurately reproduce the signal (this is called the "Nyquist Frequency"). For example, if you want to sample and reproduce a 10 Hz sine wave, you need to sample it at 20 Hz, at a minimum. In reality, you will probably want to sample at least 10-100 times the Nyquist frequency. For this piece of code, the sampling rate is set to be 1000 Hz. Look in the code above and see if you can determine how the sampling rate is set.

Practical Exam # 1

Setup the circuit as shown in Figure 4 and show your TA that you can change the width of the pulses from the PWM output signal (CH1 in the oscilloscope). In addition, show your TA that you can filter this signal (CH2 in the oscilloscope) using the RC circuit; also show that you can change the time constant by turning the pot and changing the resistor of the RC filter. Explain where the code sets the sampling rate.

Part II: Digital filters

Similarly to analog filters, digital filters have the goal of enhancing or reducing some aspects of a signal. However, unlike analog filters that need a continuous signal, digital filters only use samples of the signal taken at discrete time intervals. These filters are implemented in computer code after the signal is sampled by the Seeeduino board. They are useful because you can quickly change their parameters in computer code, rather than having to swap out resistors and capacitors. But they require a computer.

In general, digital filters will use some past samples of the already filtered signal ($y_k, y_{k-1}, y_{k-2}, \ldots$) and the present and past values of the unfiltered signal ($u_k, u_{k-1}, u_{k-2}, \ldots$) to compute the present value of the filtered signal (y_k). For instance, the filter we will be using in the lab obeys the equation:

$$y_k = (1 - \gamma)u_k + \gamma * y_{k-1}$$

Note, there are 100's of digital filters that are possible. This is just a simple and useful digital low pass filter. In this part of the lab you will implement a digital filter and use the potentiometer as an input device to control the gain of the filter. You should see how rotating the potentiometer changes the behavior of your filter. To do this, build the circuit as shown in Figure 5 and implement the code provided below for Part II.

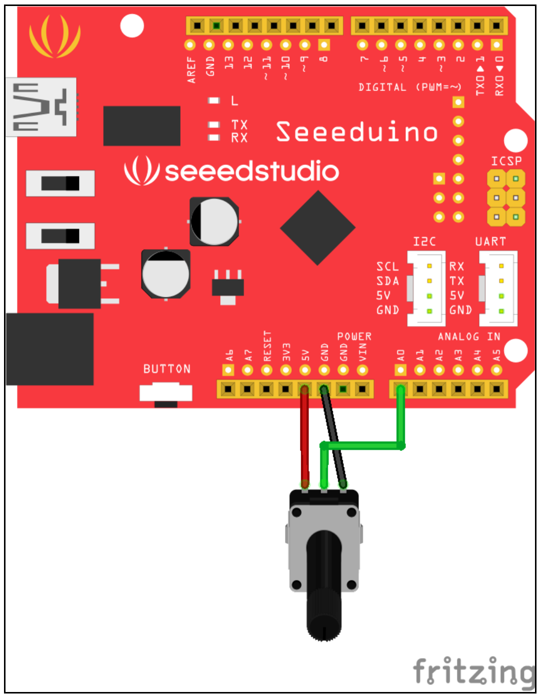


Figure 5. Circuit for Part II

Code - Part II

```
// variables for causing the loop() function to iterate at a constant
//rate
long loopDuration = 10000; // the length of time we want each iteration
                  //of the loop() function to take (in microseconds)
unsigned long startTime = 0; // the time the current iteration of the
                  //loop() function began
unsigned long elapsedTime = 0; // the amount of time elapsed while
      //current iteration of the loop() function was being performed
unsigned long waitTime = 0; // amount of time to delay at the end of
      //the current iteration of the loop() function

// define a square wave
float squareWaveState = 0.0; // the current state of the square wave
            //signal, it will toggle between 0 and 1
int numberOfLoops = 10; // number of times to run the loop() function
            //between toggling the square wave's state
int loopCount = 0; // counts the number of times the loop() function
            //has run since last toggling the square wave's state
int loopNumber = 0; // counts the number of times the loop() function
            //has run since the Arduino began

// variables pertaining to filtering the square wave
float potentiometer = 0; // A 0 to 1 signal reflecting the
                        //potentiometer knob position
float filteredSignal = 0; // the variable that represents the output of
                        //our digital filter
float filteredSignal_previous = 0; // the previous state of
                        //filteredSignal
```

```
void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);
}

void loop() {
  startTime = micros();
  // put your code in here
  potentiometer = analogRead(A0)/1023.0; // analogRead converts a
      //voltage of 0-5V into a number between 0-1023
  if (loopCount >= numberOfLoops){
    squareWaveState = 1.0 - squareWaveState; // this will toggle square
                                    //wave state - think about it.
    loopCount = 0; // after we toggle the square wave, it has been zero
                  //loop iterations since we last toggled it
    digitalWrite(13, squareWaveState); // the built in LED will display
                                    //the square wave state
  }

  loopNumber++; // increases loopNumber by 1
  loopCount++;

  filteredSignal = averagingFilter(squareWaveState, potentiometer);
      // apply the digital filter to the square wave signal

  Serial.print(loopNumber);
  Serial.print("\t");
  Serial.print(squareWaveState);
  Serial.print("\t");
  Serial.print(potentiometer);
  Serial.print("\t");
  Serial.println(filteredSignal);

  // do not put add any code below this line

  // these lines ensure that each iteration of the loop() function runs
the same amout of time
  elapsedTime = micros() - startTime; // note that the output of
micros() has changed since we called it earlier in the loop
  waitTime = loopDuration - elapsedTime;
  if (waitTime > 0){
    delayMicroseconds(waitTime);
  }
}

//------------------ simple low pass filter ------------------//
float averagingFilter(float measuredSignal, float filterStrength){
  float filterOutput = (1-filterStrength)*measuredSignal +
filterStrength*filteredSignal_previous;
  filteredSignal_previous = filterOutput;
  return filterOutput;
}
```

Practical Exam #2

Use the code provided for Part II to create a square wave using the microcontroller. Use the filter provided along with the potentiometer to digitally filter the square wave. Record both the input and output signals using the serial monitor and plot them in the computer; show these two signals to your TA and be prepared to explain what is happening. Show your TA the line of code that implements the equation for the digital filter.

Terms to study on your own

We covered a lot of concepts in this lab. Here is a list of terms that you should review and make sure you understand. You may need to consult not only the lab, but the lecture notes, the optional text book, and the web to understand these terms.

DAQ
AD convertor
DA convertor
Fourier analysis
Sampling rate
Nyquist sampling theorem
Pulse-width modulation
Filter
Low-pass filter
First-order, linear system
Time constant
Cut-off frequency
Digital low pass filter

Write-Up

1. Plot the unfiltered and filtered signals from Part II of the lab. Make sure to label all axes and lines in the plot.

2. On the same plot, for one cycle only, use the appropriate mathematical equation to predict what the filtered signal should look like, and plot it on the same plot. You will have to find a way to estimate the time constant, to use it in your equation.

3. Briefly explain how you could use a low-pass filter in your project. Do you think you will use an analog or digital filter for this purpose?