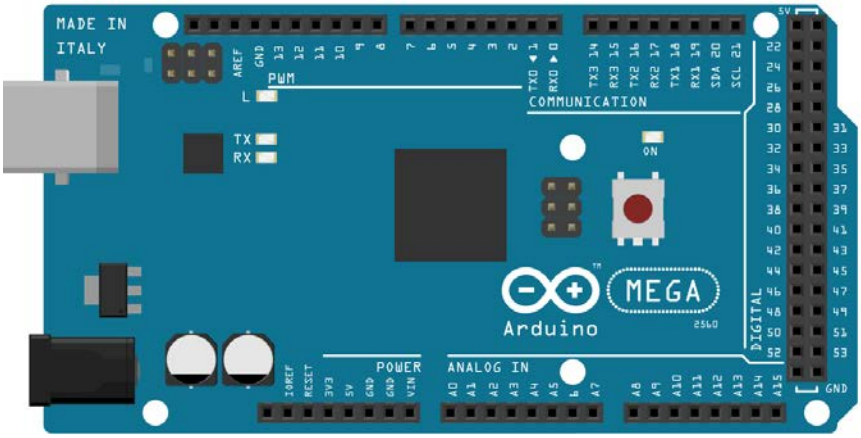
Department of Electrical and Computer Engineering

ECE 100 – Introduction to Electrical Engineering

Making, Breaking, and Hacking Stuff

An introduction to electrical and computer engineering. Topics include circuit theory, assembly, and testing, embedded systems programming and debugging, transducer mechanisms and interfacing transducers, signals and systems theory, digital signal/image processing, and modular design techniques.

Introduction to Arduino



Electrical and Computer Engineering 100

Introduction to the Arduino Platform

A Quick Overview of the Arduino MEGA

What You Will Need

Materials:

- 1 Arduino Mega (or Uno)
- 1 USB Cable A-B
- 1 Breadboard
- 10 Jumper Wires
- 3 Resistors (330 Ohms)
- 1 LED
- 1 Pushbutton / Switch
- 1 Photoresistor

Machinery:

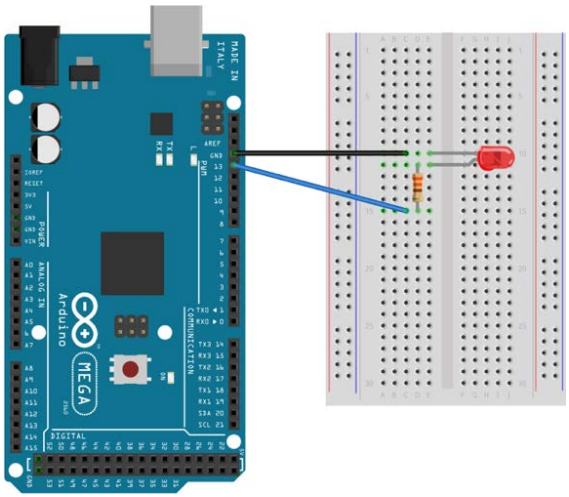
- Computer / Laptop

Software:

- Arduino Software (IDE)

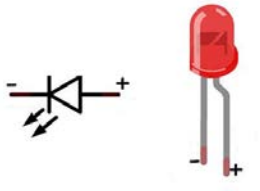
Challenge #1: Blinking LED Light

Objective: You are going to create a simple circuit using an LED light. The goal is to have the LED light turn on for 2 seconds, and then turn off for 1 second. Please see Appendix for more information about how to use a breadboard.



Components:
1. 1 Arduino MEGA
2. 1 Breadboard
3. 2 wires (choose one black/brown for GROUND, and the other is any color)
4. 1 resistor (330 ohms)
5. LED light

Wiring: Using the circuit diagram provided, wire the Arduino.



Note: LEDs are light emitting diodes. Diodes allow current to flow only in one direction. The negative side is the cathode and the positive side is the anode. With that, the shorter leg goes to GND (ground), and longer leg goes to a pin. Ground is the reference point in an electrical circuit which voltages are measured (usually zero).

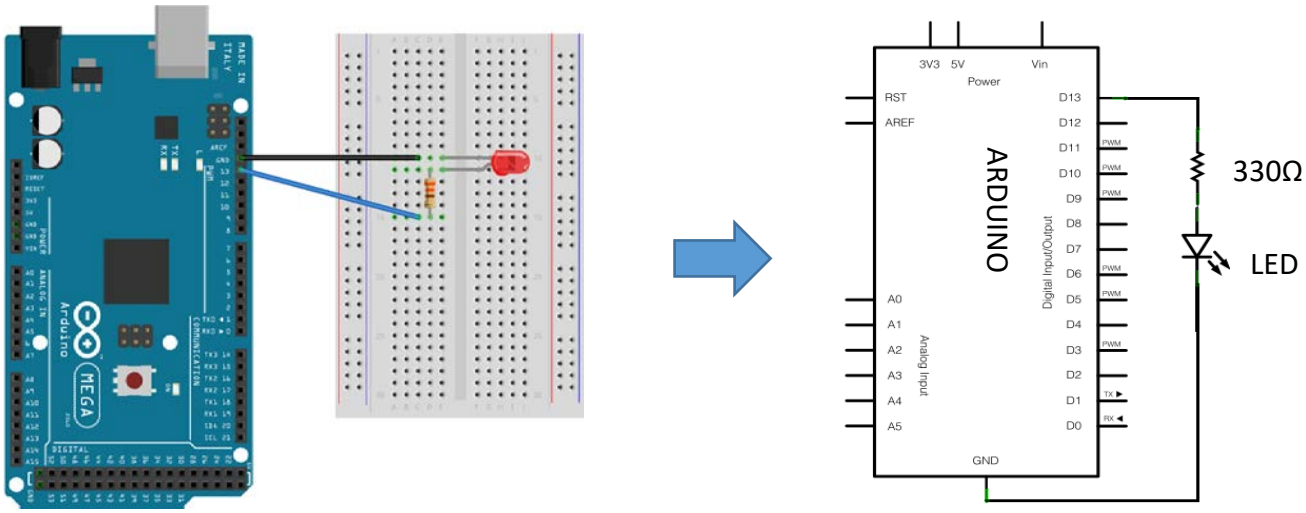
Coding/Programming: Open the Arduino IDE software and write the following code:

```
// assign a value of 13 to the integer variable led
int led = 13;

// setup() function runs once when the Arduino turns on or is reset
void setup() {
  // initialize the digital pin as output.
  // assign variable led (pin 13) as this output
  pinMode(led, OUTPUT);
}

// loop() function runs the code inside repeatedly while Arduino is still on
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(2000);               // wait for two seconds
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);               // wait for a second
}
```

Note: Everything behind these slashes // are just comments, not actual code. The // indicates that everything after it will be seen as a comment and ignored by the compiler and ARDUINO.
Note: WAIT! **Don't take down the circuit!** You'll need it for the next challenge. ☺

Terminology:
LED (Light Emitting Diode): A diode is a device that only allows for the flow of electricity to pass in one direction. A LED is a diode that generates a specific wavelength of light when a voltage is applied across its leads.
digitalWrite(): Is a built in function that outputs to a specific pin to turn something ON (HIGH) OR OFF (LOW). In Arduino Coding, HIGH translates to ON which is 1 and LOW translates to OFF which is 0 (to the devices).
pinMode(): Is a built in function that initializes a pin as an input or output (default is output). Meaning, it tells the ARDUINO, "Hey! This pin will be your target for input or output!"
delay(): Is a built in function that does what the name says: delays. It stops everything in the ARDUINO and holds for however many milliseconds specified. In the code above, the LED is turned on with digitalWrite, then, delay of two seconds occurs, then digitalWrite turns the LED off, and the delay of one second occurs before looping back up. Normally, delay in a non-LED based circuit will be problematic for sensors because once a delay occurs, this stops everything in the ARDUINO, including the reading of data.

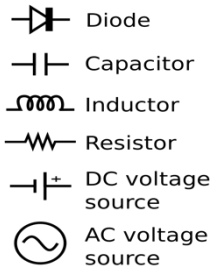
Transitioning to Schematics



The circuit diagram in Challenge #1 is a nice visual representation of what the circuit looks like. Unfortunately, it is a bit unpractical when designing larger scaled electronics. A more efficient yet harder to understand diagram is the one shown on the right.

The first observable difference is the lack of a breadboard; it saves a lot of space to directly indicate where everything is connected!

We have also replaced the nice pictorial representations of the circuit elements with their corresponding symbols. The zig-zag line is the resistor, and the triangle is the LED.



These elements are connected to the same Arduino pins as shown before. The result is a much neater circuit diagram that provides the same information in a lot less space. **Please get accustomed to this type of diagram; it will become the standard for displaying circuits.** A few symbols found in common circuit diagrams are listed to the right.

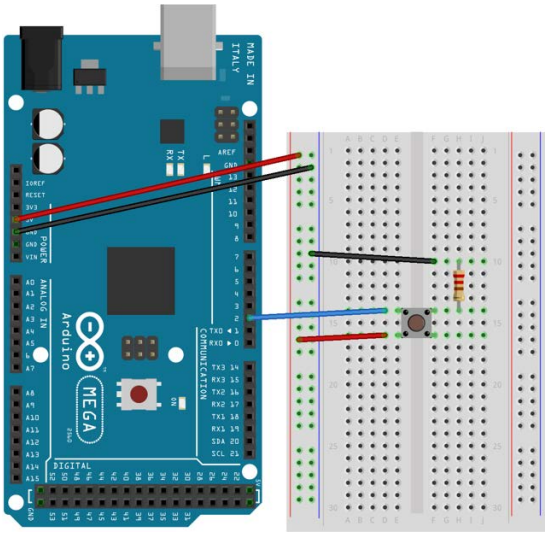
Note: The diagram above (on the right) uses an Arduino UNO, but in practice we will be using an Arduino Mega. The reason we chose to display the UNO is that the diagram takes much less space, and achieves the same purpose (the Arduino UNO has less pins).

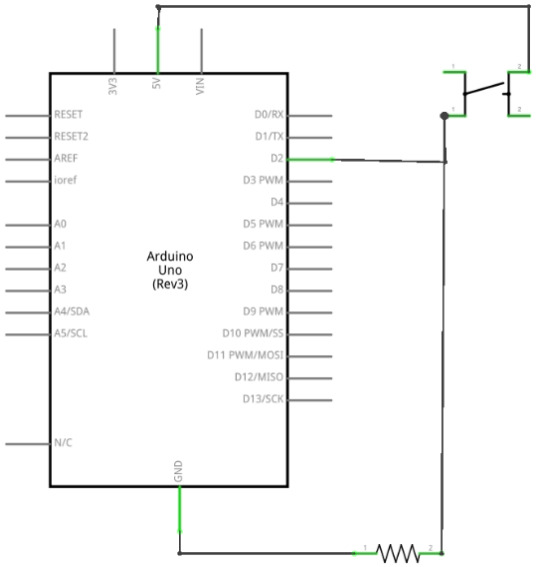
Challenge #2: Integrating LED and Button

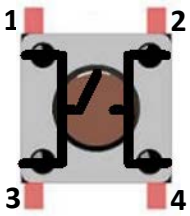
Objective: Building from the last circuit you've created, you are going to take what you learned from challenge #1 to construct a more complicated circuit involving both a button and LED (integrating a button circuit into the existing LED circuit). You will keep the previous circuit and continue building it. The goal is to turn the LED on and off at the press of a button. Press the button, and the LED turns on (button/LED should STAY on until pressed again). Press it again, and it turns off (STAYS off). The following circuit diagrams show the button circuit ONLY, make sure to properly combine the LED circuit (from the last challenge) and the button circuit:

Components (for the button circuit):

1. 1 Arduino MEGA
2. 1 breadboard
3. 5 jumper wires (choose 2 black or brown for GROUND, 2 red or orange for 5V and the other any color)
4. 1 pushbutton
5. 1 resistor (330 ohms)







Note: Pushbuttons are switches and remain in the open position while not being pushed. This is where, according to the diagram to the right, **1** & **3** are connected and **2** & **4** are connected, but **1** & **2** are not connected to **3** & **4**. While the pushbutton is being pushed, the switch is in the closed position. This means that **1**, **2**, **3**, & **4** are all connected. The figure to the left is shown in the open position.

Wiring: The beauty of the Arduino is the fact that it will allow you to integrate both circuits without needing to disconnect the LED circuit. Using the circuit from the previous challenge, and the circuit in the figure, wire the ARDUINO (essentially add the circuit on the right to the circuit from the previous example). You will not be connecting the button and LED in series! **They are two separate circuits communicating with each other via the Arduino!**

Note: Remember to keep the LED circuit intact! You'll need it for a later challenge. ☺ You can take down the button circuit.

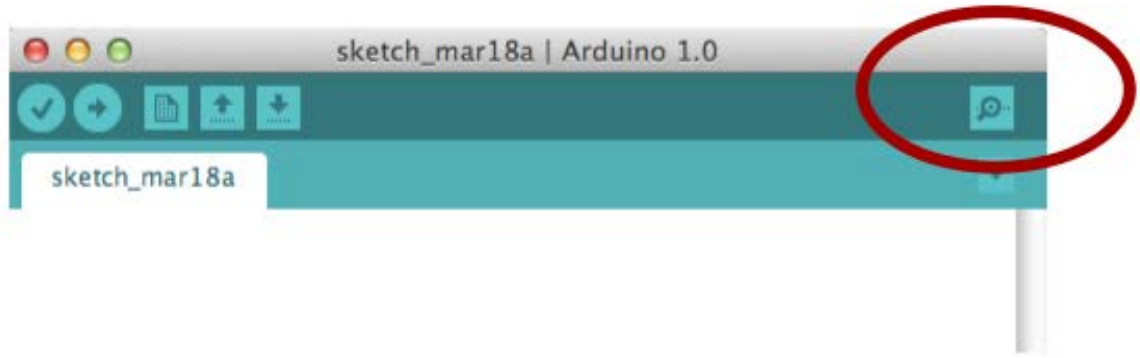
Coding/Programming: The following code is nearly complete. You must fill in the missing lines for the ARDUINO to complete the task. This does take some good thinking. Make sure you read the comments! They will help you fill in the code.

```
int buttonPin = _______;     // the number of the pushbutton pin
int ledPin = _______;      // the number of the LED pin

// variables will change:
int buttonState;         // variable for reading the pushbutton status
int ledState = LOW; // variable to store the LED state: HIGH = on, LOW = off

void setup() {

  pinMode(____________, INPUT);

  pinMode(____________, OUTPUT);

 }

void loop(){
  // read the state of the pushbutton value:
  buttonState = digitalRead( buttonPin );

  // check if the pushbutton is pressed.
  if (buttonState == ___________ && ledState == LOW) {
    // turn LED on:
    digitalWrite(___________ , ___________ );
    ledState = ___________;
  }
  else if (buttonState == ___________ && ledState == HIGH) {
    // turn LED off:
    digitalWrite(___________, ___________);
    ledState =  ___________;
  }
  delay(100);   // delay in order to give some time to read the button press
                 //you can change this value and observe the difference
}
```

Terminology:
digitalRead(): Is a built in function that reads in what the state of the pin is: HIGH (on) or LOW (off). It will return an integer value of 1 for HIGH (on) and 0 for LOW (off).

Note: WAIT! DON'T TAKE DOWN THE LED CIRCUIT (JUST THE BUTTON CIRCUIT IS REMOVED)!

Challenge #3: Printing to the Serial Monitor



The serial monitor is a pop-up window that acts as a separate terminal that communicates by receiving and sending Serial Data. Its job is to allow you to both send messages from your computer to an Arduino board (over USB) and also to receive messages from the Arduino. This challenge will make use of the serial monitor and allow you to print text. As indicated in the figure, to access the text printed, click on the icon in the top right corner of the Arduino IDE (software interface). The serial monitor will open as a new window if the Arduino is connected to the computer and there is text to display. Make use of the serial monitor to view data or debug your program!

Objective: Using the following code, print the line "HELLO, WORLD!" into the serial port once, print an empty line, and lastly, print the names of three of your neighbors sitting next to you. (Introduce yourself if you don't know them ☺) **No wiring of electrical components is necessary in this challenge**. You are simply going to connect the Arduino to the computer.

Begin by placing the following line in the void setup() function:

```
Serial.begin( 9600 );
```

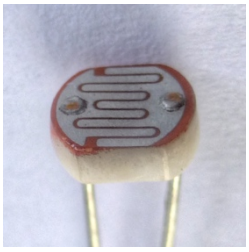
This opens the serial line for data transmission via printing. The 9600 is the baud rate, which is the number of bits per second being transferred to the serial line for printing (data transmission rate).

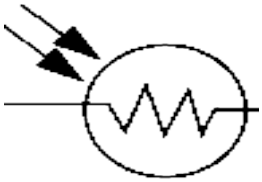
```
Serial.print("_____insert text_____");
//Prints the text between the quotation to the serial monitor

Serial.print("\n");
.....
.....
.....
```

Note: The \n means "to the next line". This is equivalent to hitting the enter/return key on Microsoft Word. How many of these do you need to print an empty line?

To see the text in the serial monitor, simply click on the button on the top right hand corner of the Arduino IDE window.

Challenge #4: Photoresistor and LED

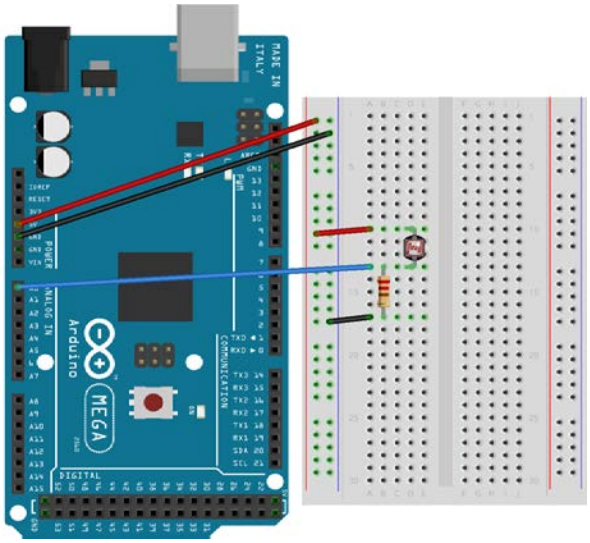


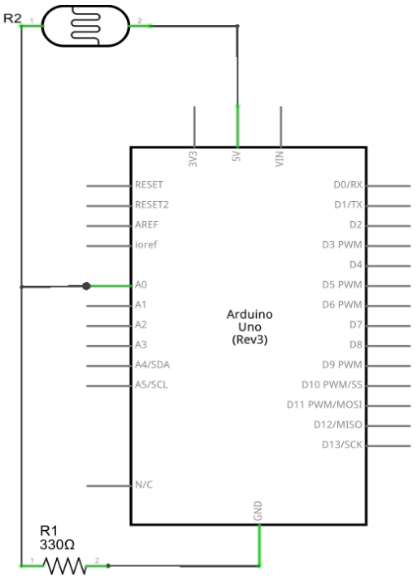


Photoresistor symbol

What is a photoresistor? Also known as a light-dependent resistor (LDR) and photocell, a photoresistor is a resistor with varying resistance that depends on light intensity (the resistance decreases as light intensity increases, while decreasing light intensity will increase resistance). Thus, a photoresistor's resistance is inversely proportional to the intensity of light directed on it.

Objective: You are going to create a photo-resistor and LED light based circuit. The goal is to use the photo-resistor to detect when the LED is turned on. As with the previous challenge, you will need to build this circuit on top of the circuit created in Challenge #1 (the blinking LED). The circuit diagram *and* the code will only cover the photo-resistor part of the challenge; **it is your task to combine them**!





Components (for the photo-resistor circuit):

1. 3 wires (choose 1 black or brown for GROUND, 1 red for 5V and the other one any color)
2. 1 resistor (330 ohms)
3. 1 Photoresistor

Wiring: Using the circuit in the two figures above, wire the ARDUINO as shown in the figure to the right. Similar to the previous exercise, simply add this circuit next to what you have already designed to integrate the photoresistor and LED circuits. NOTE: In the circuit diagram above, notice that the photoresistor is not wired to the digital pin this time! It uses the analog side! You are plugging into an **ANALOG PIN**!

Terminology:

Analog vs Digital: Digital pins are either high or low. 0 or 1. Black or white. Light or dark. Nothing in between. They can be used as inputs **AND** outputs. These are mostly for tactile or digital sensors, like buttons and switches. Analog inputs are 0 and 1, with everything in between. Instead of just black and white, you can have dark grey, light grey, etc. These **ARE ONLY** inputs. When you assign a pin number associated with an analog pin you can use the full name A0, for instance, or simply 0. In this case high refers to 5V and low refers to 0V.

Coding/Programming: Fill in the highlighted blanks based on how you wire the ARDUINO. Insert code from challenge #1 to turn the LED on.

```
// constants won't change. They're used here to set pin numbers:
const int sensorPin = _____;     // the number of the sensor pin

// variables will change:
int lightLevel;         // variable for storing the photodiode data


void setup() {
  Serial.begin(9600);
  pinMode( ___________, INPUT);
}

void loop(){
  lightLevel = analogRead(sensorPin);
  Serial.print("The photodiode is reading:");
  Serial.println(lightLevel);
}
```

Notice the serial prints! Click on the serial monitor button in the top right corner of the Arduino IDE to see your output!

Congratulations!
You have completed Lab 0!

APPENDIX

1. Please visit Sparkfun to learn more about how breadboards work!
   https://learn.sparkfun.com/tutorials/how-to-use-a-breadboard

2. The Arduino website is a very good resource for grasping circuits and Arduino in general. Use the following link and click on the "Learning" tab, where you will find how to get started, tutorials, and other useful references!

   https://www.arduino.cc/en/Main/Documentation#

3. SparkFun is a vendor that sells a variety of platforms, one of them Arduino. Luckily, they have great resources on the site to help you learn Arduino.

   https://learn.sparkfun.com/tutorials/what-is-an-arduino

4. Adafruit is a DIY (Do-It-Yourself) vendor that sells and utilizes a variety of platforms. Luckily, they have a section dedicated to Arduino! Simply click on the "Learn" tab and follow the Arduino links. It will take you to a large amount of projects for you to build on your own!

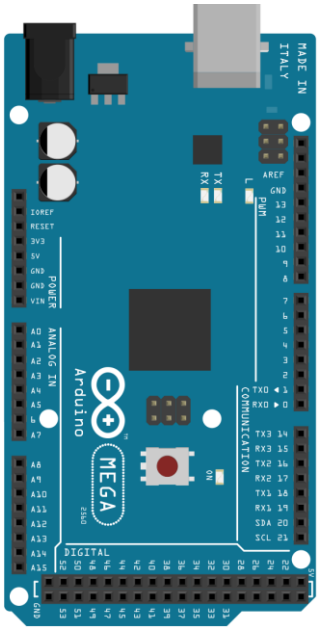
   https://www.adafruit.com

5. Instructables is a good project-based resource with a variety of platforms just like Adafruit. Check the website out for more interesting projects!

   http://www.instructables.com

Acknowledgements: Circuits presented in this document were developed courtesy of ARDUINO, Adafruit, and Sparkfun documentation.

Communication



Electrical and Computer Engineering 100

Wireless Communication using Arduino

What You Will Need

Materials:

- 1 Arduino Mega (or Uno)
- 1 USB Cable A-B
- 1 Bread board
- 10 Jumper Wires
- 1 Resistor (330 Ω)
- 1 Resistor (180 Ω)
- 2 Resistors (100 Ω)
- 1 IR LED*
- 1 IR Receiver (TSOP382)*
- 1 IR Remote*
- 1 Pushbutton
- 1 RGB LED

* parts included in IR Kit

Machinery:

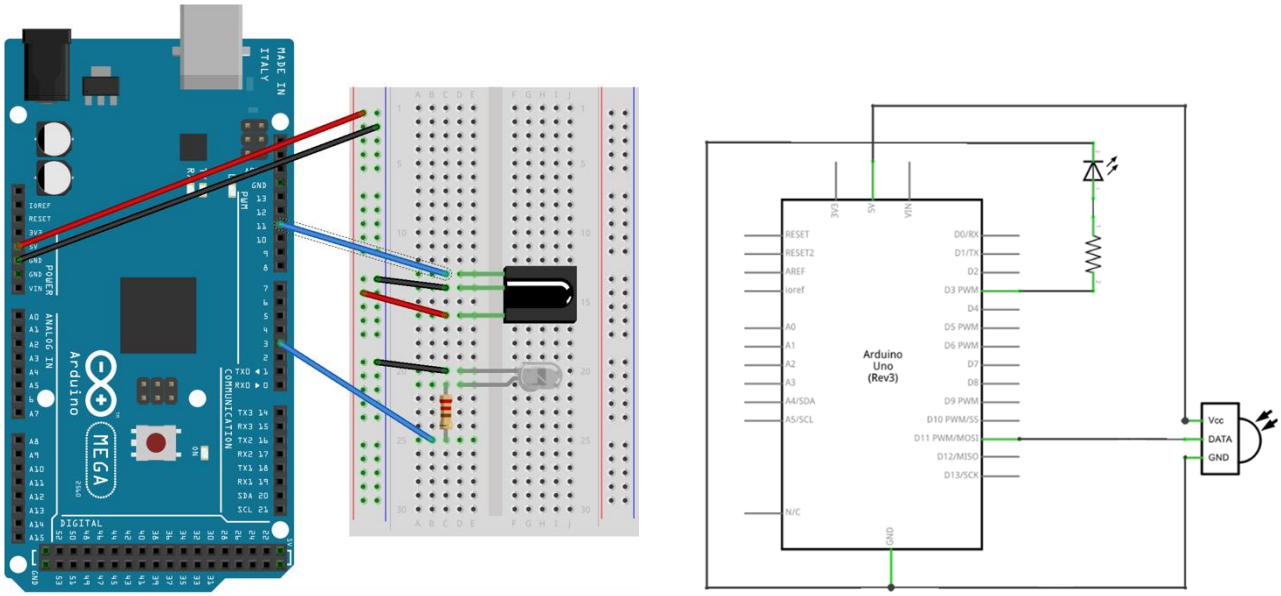
- Computer / Laptop

Software:

- Arduino Software (IDE)
- Timer Library (https://code.google.com/p/arduino-timerone/downloads/list)
- Ken Shirriff's Library (https://github.com/z3t0/Arduino-IRremote)

Challenge #1: IR Sensor and IR LED

Objective: As you might have experienced, the photo-resistor had difficulties recognizing when the LED was on (see Lab 0). This is due to noise caused by many other light sources. A way to alleviate this and improve the circuit is to incorporate Infrared (IR) light into the equation. IR is invisible to humans, but it is detectable with the right sensors. The goal of this challenge is to emulate the last circuit by replacing visible light with IR. Since the IR sensor only does a digital read, the values of the output will only be either 1 or 0.



Components:

1. 1 Arduino Mega
2. 1 Bread-board
3. 7 wires (2 red for voltage/pin, 3 black (ground), 2 colored wire for I/O)
4. 1 resistors (330 ohms)
5. 1 IR LED
6. 1 TSOP382 IR receiver.

You will also be working with new concepts: libraries and timer interrupts.

Sometimes we want to tackle a problem that has already been solved before. If we all had to build everything from scratch, it would not be the most efficient way to create our projects, so we resort to building from other people's code. These solutions are compiled into packages called "libraries". A library has a variety of files in it, all organized in a specific structure. They are designed to work with Arduino and to flow nicely with the IDE. Adding a library is very simple. All you have to do is go to Sketch -> Include Library -> Add .ZIP Library and then select the .ZIP file from your computer. Libraries are often found all over the internet, anyone can create them! Arduino libraries are written in C++ (an object oriented programming language), which looks slightly different from the Arduino C that you've been learning.

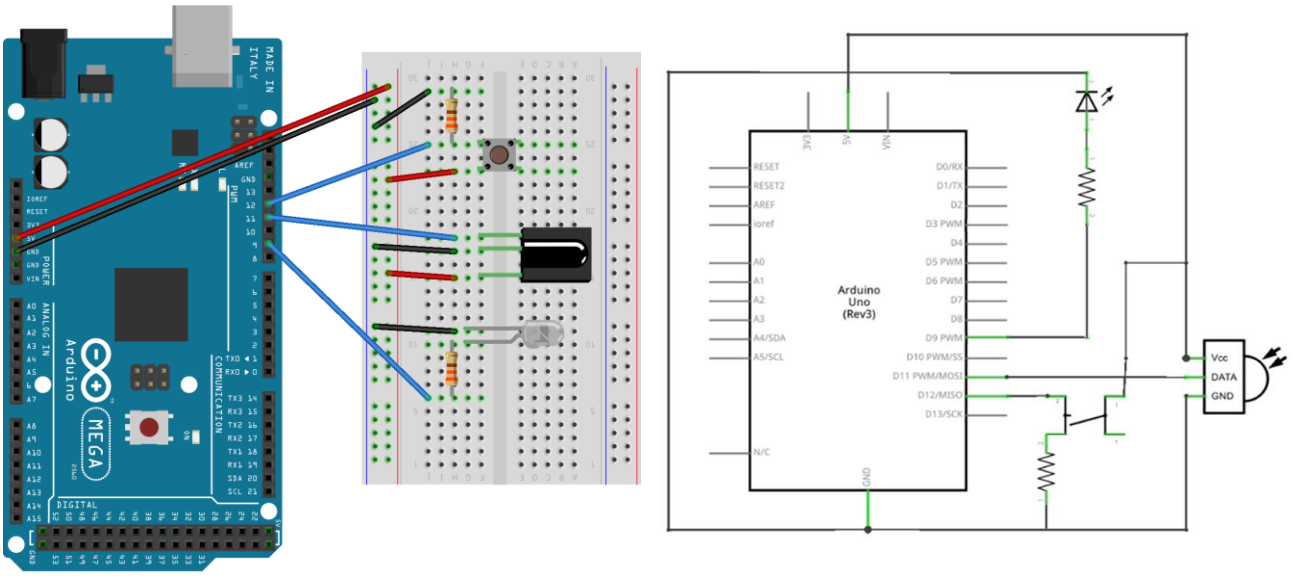
In the sketch (Arduino IDE interface), the Arduino repeats the loop() code over and over again, and the speed in which this code runs depends on how intense the work is. So the time it takes to run each iteration of loop() will vary. Sometimes we want to run a specific piece of code in a precise timed manner. We are able to accomplish this using timer interrupts. Timer interrupts interrupt whatever is running inside loop() in order to run the time sensitive code of your choosing. We are able to set the specific frequency in which we require our sensitive code to run, with great results.

In this example, we need to turn the IR LED on and off at a frequency of 38kHz, or every 26 microseconds. This is necessary because out IR receiver has this characteristic, it will only read IR in that specific frequency. In order to accomplish this, we will use a Timer library, found at https://code.google.com/p/arduino-timerone/downloads/list

Coding/Programming: The code is very similar to the last example. Can you identify what changed?

```
#include "TimerOne.h"
// constants won't change. They're used here to
// set pin numbers:
const int ledPin =  _____;      // the number of the IR LED pin
const int sensorPin = _____;    // the number of the IR sensor pin

// variables will change:
int lightLevel;         // variable for storing the sensor data

void setup() {
  Serial.begin(9600);
  pinMode( ___________, INPUT);
  pinMode( ___________, OUTPUT);

  Timer1.initialize(26); // set a timer of length 26 microseconds (or 38kHz)
  Timer1.attachInterrupt(callback); //attach the routine that will happen at the
interrupt
}

void loop(){
  lightLevel = digitalRead(sensorPin);
  Serial.print ("The IR Sensor is reading:");
  Serial.println (lightLevel);
  delay(1000);    // delay in order to give some time to read the button press
                  //you can change this value and observe the difference
}

void callback() // time sensitive function to toggle the IR LED
{
  digitalWrite(ledPin, digitalRead(ledPin) ^ 1); //Toggle the IR LED
}
```

Challenge #2: IR Remote



Objective: Now we get to the cool stuff. IR is the technology used in TV remotes, they transmit the exact same type of light from the last exercise! So we could potentially use our circuit to function just like a TV remote. And that is what we will do.

Components:

1. 1 Arduino Mega
2. 1 Bread-board
3. 10 wires (3 red for voltage/pin, 4 black (ground), 3 colored wire for I/O)
4. 2 resistors (330 ohms)
5. 1 IR LED
6. 1 TSOP382 IR receiver
7. 1 Push Button
8.

Resources:

We will be using Ken Shirriff's IR library found on github: https://github.com/z3t0/Arduino-IRremote
This library contains all the encoding and decoding necessary in order to properly send and receive the information. Since it is so complex, and someone has already done it, it is always better to use a library.

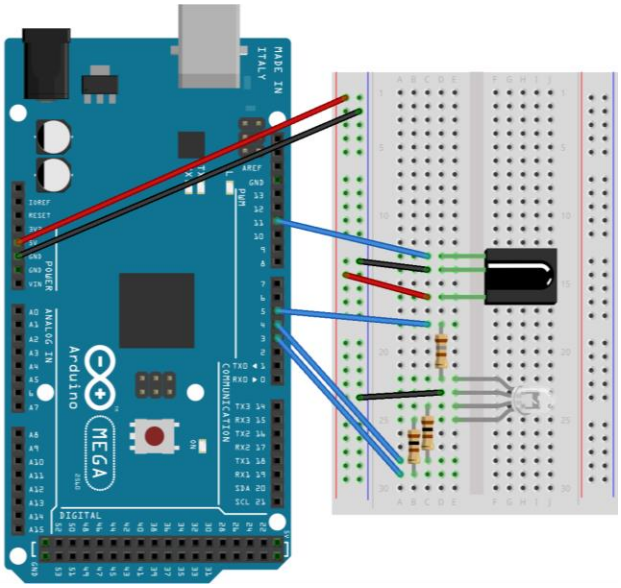
Note: *Different versions of the library have different installation instructions, so please look at the instructions on the github page in order to make sure you can get the library working. Also, with the IR-Remote Library Pin 9 is used for connecting Arduino Mega 2560 to the IR LED but Pin 3 is used if you were using an Arduino Uno.*

If there are difficulties with updated versions, download version 1 of the software found on the github site under releases. If you are having difficulties including the Arduino Library, try using the built in library manager within the IDE found through Sketch -> Include Library -> Manage Libraries… or Add .ZIP Library….

Select the file "IRrecord" from File -> Examples -> IRremote -> IRrecord
Try to go through the code and understand the basic concepts of it all. The comments do a good job at explaining what the code does, which is act as a 'relay'. This means that the circuit will read the signal sent from the remote, and then send the same signal whenever you press the button. You can have a lot of fun with this. Program it to send the TV on/off button, and sneakily turn your roommate's TV off, even when they have the remote with them.

Challenge #3: IR Controlled RGB LED

Objective: Time for some RGB fun! In this lab, we will use the IR remote to control the color of an RGB LED. We are also going to use a different programming concept: ***functions***. Functions are blocks of code used to perform a specific task.



Components:

9. 1 Arduino Mega
10. 1 Bread-board
11. 9 wires (2 red for voltage/pin, 3 black (ground), 4 colored wire for I/O)
12. 3 resistors (1 180ohms for Red pin, 2 100ohms for Blue and Green pins)
13. 1 RGB LED
14. 1 TSOP382 IR receiver

Note: For the nine wires, choose 2 red for voltage/pin, 3 black (ground), 4 any colored wire for I/O (input/output). The longest pin on the RGB LED is the ground pin, the other three depend on the manufacturer so check the datasheet!

You have already used many different functions at this point. When you use analogRead(), you are calling a function from deep inside the Arduino libraries. In the Timer example, callback() is a function. Functions are useful because they allow you to split up your code into smaller segments, instead of putting it all in loop(). This helps with simplicity of the code. When you want to repeat a specific task many times, it is much easier to call a function instead of rewriting the same code over and over again. We will use functions to simplify the process of changing the LED color.

Data Sheets:

The RGB LED is a common component. Check out the following data sheet to learn more about the specific component and find out which pins are for Red, Green, Blue, and Ground.

https://www.sparkfun.com/datasheets/Components/YSL-R596CR3G4B5C-C10.pdf

Let's look at one specific function, setColor:

```
void setColor(int red, int green, int blue){
  analogWrite(redPin, red);
  analogWrite(greenPin, green);
  analogWrite(bluePin, blue);
}
```

This function's sole purpose is to write to the three RGB pins (redPin, greenPin, and bluePin) and assign the three parameters (int red, int green, int blue) as their values. setColor's return type is void, meaning that it doe's not send any information back to the calling function.

Another important aspect is the use of pointers. Pointers, like int and char, are a datatype. Their main purpose is to "point" to other variables. In order for them to "point" at a variable, they must store the variable's address, which is used to figure out what the pointer is referencing.

Let's suppose you have three ints, int red, int blue, and int green. Red is set to 5, blue to 10 and green to 15. Like such:

```
int red = 5;
int green = 10;
int blue = 15;
```

We also create a pointer called active. But how do we name it? pointer active is not correct (at least under Arduino C) so the correct way to create a pointer, and specify that it will point to an int is as follows:

```
int *active;
```

The '*' is what sets it apart from other ints. It turns it into a pointer to an int, instead of just an int. It is currently not pointing to anything, so let's give it an address.

```
active = &red;
```

The '&' tells the pointer to grab the address of red, instead of the value of red (which is in this case 5). Now that our pointer points at something, how do we extract the value from that something? We *dereference* it. Here is how you do it:

```
green = *active
```

The '*' makes it so you access the value of whatever active is pointing to, in this case 5. This sets the value of green to 5.

Wiring: Using the circuit at the beginning of the section, wire the ARDUINO as shown.

Coding/Programming:

```
#include <IRremote.h>
const int RECV_PIN = 11;
const int redPin = 5;
const int greenPin = 4;
const int bluePin = 3;
int i = 0;
int red = 0;
int green = 0;
int blue = 0;
int *active;
int OnOff = 0;
int ColorActive = 0;
IRrecv irrecv(RECV_PIN);
decode_results results;
void setup() {
  Serial.begin(9600);
  irrecv.enableIRIn(); // Start the receiver
  // Set the three LED Pins as outputs
  pinMode(redPin, OUTPUT);
  pinMode(greenPin, OUTPUT);
  pinMode(bluePin, OUTPUT);
}
void loop() {
  if (irrecv.decode(&results)) {
    //Serial.println(results.value, HEX);
    irrecv.resume(); // Receive the next value
    if (results.value == 0x10EFD827 && OnOff == 1){
        //Power off
      Serial.println("Turning off");
      setColor(0,0,0);
      OnOff = 0;
      ColorActive = 0;
    }
    else if (results.value == 0x10EFD827 && OnOff == 0){
      Serial.println("Turning on");
      setColor(red,green,blue);
      OnOff = 1;
    }
    if (OnOff == 1){
      if (results.value == 0x10EFA05F && ColorActive == 1){
        //Up arrow, increment the active color
        colorUp(*active);
      }
      if (results.value == 0x10EF00FF && ColorActive == 1){
        //Down arrow, decrement the active color
        colorDown(*active);
      }
      if (results.value == 0x10EF807F && ColorActive == 1){
        //Right arrow, maximize the active color
        *active = 230;
        colorUp(*active);
      }
      if (results.value == 0x10EF10EF && ColorActive == 1){
        //Left arrow, minimize the active color
        *active = 25;
        colorDown(*active);
      }
      if (results.value == 0x10EF20DF){ // Circle button
        circle();
      }
      if (results.value == 0x10EFF807){
```

```
        //A, Set red as the active color, and turn on Red as an indicator
        Serial.println("Red is active");
        active = &red;
        ColorActive = 1;
      }
      if (results.value == 0x10EF7887){
        //B, Set green as the active color, and turn on Green as an indicator
        Serial.println("Green is active");
        active = &green;
        ColorActive = 1;
      }
      if (results.value == 0x10EF58A7){
        //C, Set blue as the active color, and turn on Blue as an indicator
        Serial.println("Blue is active");
        active = &blue;
        ColorActive = 1;
      }
    }
  }
  delay(100);
}
// Incrememnt the active color by 25
void colorUp(int &color){
  color = (color + 25) % 256;
  setColor(red,green,blue);
  Serial.println(color);
}
// Decrememnt the active color by 25
void colorDown(int &color){
  color = (color - 25) % 256;
  setColor(red,green,blue);
  Serial.println(color);
}
// Set the colors to the parameter values
void setColor(int red, int green, int blue){
  analogWrite(redPin, red);
  analogWrite(greenPin, green);
  analogWrite(bluePin, blue);
}
void circle(){
  red = 255;
  green = 127;
  blue = 0;
  Serial.println("Start Circle");
  while(true) {
    red = (red + 25 +256) % 256;
    green = (green + 25 +256) % 256;
    blue = (blue + 25 +256) % 256;
    setColor(red,green,blue);
    delay(150);
    if (irrecv.decode(&results)) {
      Serial.println("Stop Circle");
      irrecv.resume();
      return;
    }
  }
}
```

Resources

The Arduino website is a very good resource for grasping circuits and Arduino in general. Use the following link and click on the “Learning” tab, where you will find how to get started, tutorials, and other useful references!

https://www.arduino.cc/en/Main/Documentation#

SparkFun is a vendor that sells a variety of platforms, one of them Arduino. Luckily, they have great resources on the site to help you learn Arduino.

https://learn.sparkfun.com/tutorials/what-is-an-arduino

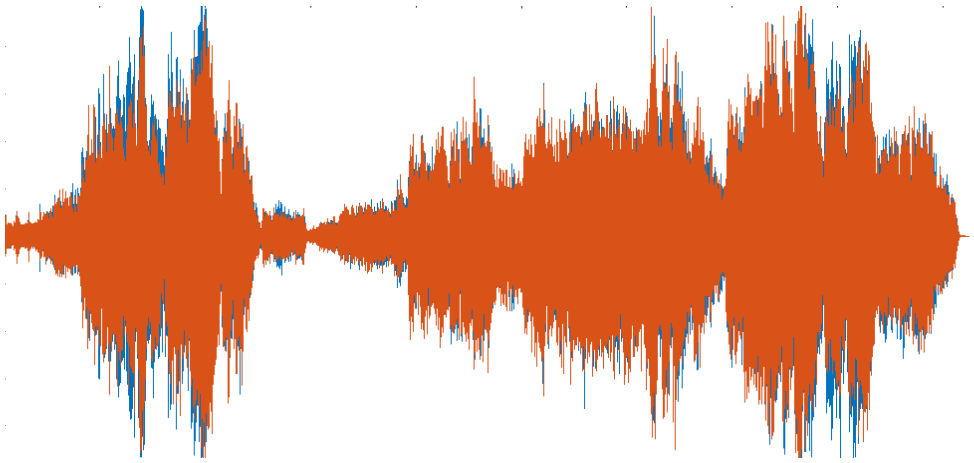
Adafruit is a DIY (Do-It-Yourself) vendor that sells and utilizes a variety of platforms. Luckily, they have a section dedicated to Arduino! Simply click on the “Learn” tab and follow the Arduino links. It will take you to a large amount of projects for you to build on your own!

https://www.adafruit.com

Instructables is a good project-based resource with a variety of platforms just like Adafruit. Check the website out for more interesting projects!

http://www.instructables.com

Digital Signal Processing using MATLAB



Electrical and Computer Engineering 100

Exploring Digital Signal Processing

Introduction

Welcome to Lab 2! This assignment focuses on the field of Electrical Engineering known as Digital Signal Processing (DSP). DSP is a sub-discipline of the more general field of Signal Processing, a broad and highly important field which has applications in fields ranging from image and audio processing to financial services. In the infancy of signal processing, signals were mainly processed using analog amplifiers; this was largely due to the fact that it just was not feasible to perform the desired operations and manipulations on the current digital systems. Nowadays, huge advances in computing power and technology have made DSP a powerful alternative to analog signal processing. In addition, many modern systems can be classified as "mixed signal" systems as that they contain both digital and analog components. It is to your advantage as an engineer to know how to work in both of these domains.

What You Will Need

Materials:

- Provided Image and Sound files.

Equipment:

- Desktop computer / Laptop

Software Tools:

- MATLAB suite with valid license

Objective

The DSP application we will be exploring in this lab follows three general steps:

1. First, a measurement from the world is captured as an analog signal. This analog signal is *digitized*, or *sampled*, at a specific frequency (represented in Hertz (Hz)), and the values of the samples are converted to a number of binary bits (in a process called quantization). These bits form the *digital representation* of the original analog signal. This entire process is known as Analog to Digital Conversion (ADC).
2. These bits form *digital signals* which can be manipulated and processed by a digital computer. The useful information these signals contain can be extracted at this stage.
3. Depending on the application, the processed signals can be converted back to an analog signal in a process known as Digital to Analog Conversion (DAC). This is often the case in digital audio processing, as a speaker system is driven by an analog voltage/current signal.

In this lab you will have the opportunity to perform the first two of these steps (the third step can be done in many ways. One could build a simple DAC circuit and speaker driver circuit for example. You will find many such projects online.) In doing this, you will gain exposure to many of the key concepts and tools used in DSP.

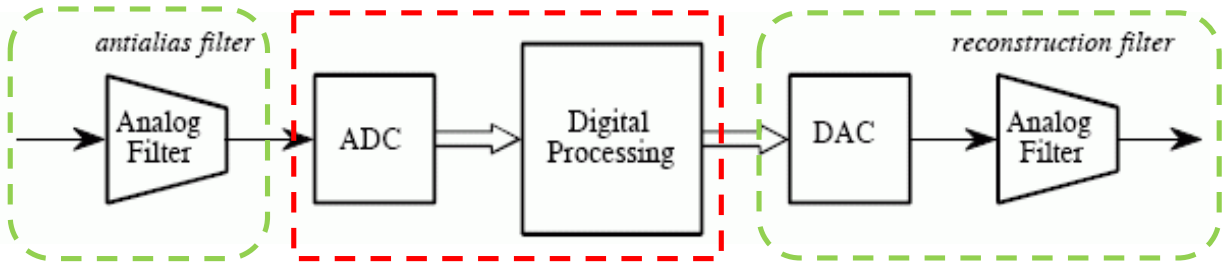


Figure 1: (Source: http://www.dspguide.com/graphics/F_3_7.gif)

Figure 1 shows the general DSP procedure mentioned previously. We will be focusing on the processes inside of the red, dashed rectangle. This will be done using MATLAB, a powerful computing platform which is widely used by professional engineers in many different industries. The goal of this section is to explore the concepts of sampling, filtering, and how signals change over time.

Part 1: Audio Signal Processing using MATLAB

When you first open up MATLAB, you will notice three distinct windows (Figure 2: numbers added for clarity.)

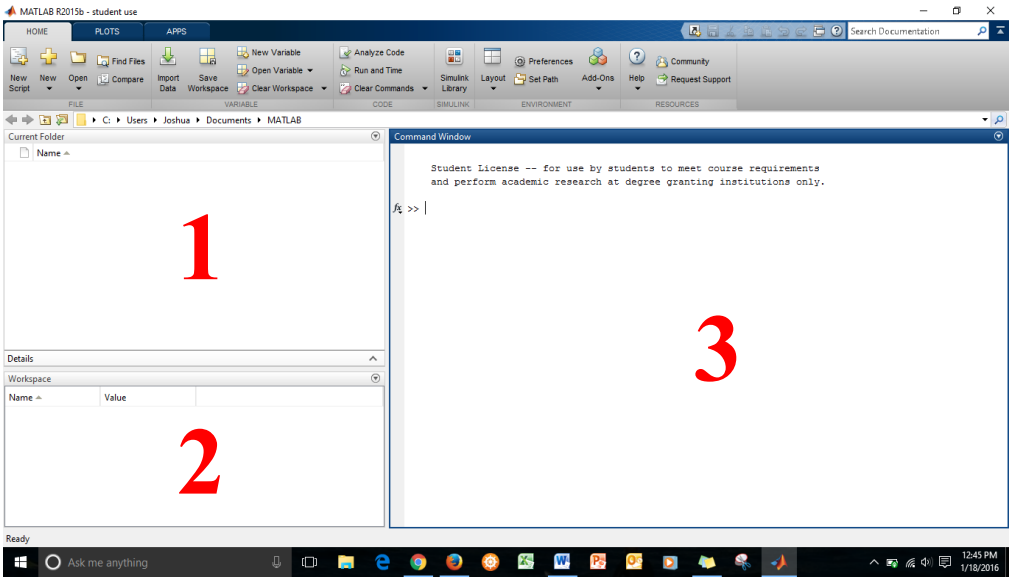


Figure 2: MATLAB home screen.

Window "1" shows the current file path which MATLAB is using to open and save files. Changing the path can be accomplished by either specifying the path in the search bar or clicking the "browse for folder" icon located at the top left of the window.

Window "2" is known as the "Workspace". MATLAB stores all variables and objects which you create in the workspace, it is a convenient location to access your variables and to check for errors in your programs. Window "3" is called the "Command Window", and here is where you can execute your MATLAB Commands. Executing a command is as simple as typing in the command and pressing enter. For example, type in the MATLAB command *"ver"* (omit quotation marks) and press enter. This command will list the version numbers for all the features of your MATLAB suite. To clear your old commands from the Command Window, type in "*clc*". For help with any command or MATLAB function, type "*help*" followed by the name of the function or command in question. For example, if I wanted to know more about the MATLAB absolute value function, I would type the following into the command window: "*help abs"*.

The command window will only execute one command at a time, so it is difficult to execute long programs requiring multiple lines of code. So instead, we create what is called a "script", a collection of commands which can all be executed at once, just like a typical computer program. To create a script, click on the "New Script" button at the top left corner of the MATLAB home screen. You may find that performing the following exercises using MATLAB scripts is much easier than entering each line of code individually in the command window.

Section 0: MATLAB Fundamentals

MATLAB is a relatively simple programming language to learn given the fact that it is used to solve a wide variety of engineering problems. In many of the exercises of this lab, sample code is provided for your reference. Additionally, we will be using a large number of predefined functions in our assignments, and all of these functions are well documented (this is one major reason why MATLAB is used here over many open source alternatives). Some of the problems in this lab are left open ended for a reason, as one of the goals of this lab is for you to practice consulting the help documentation. This will enable you to figure out how to use many of the functions included in MATLAB and will help improve your general competency with the tool. For those of you who have never programmed before or if you need a refresher on the MATLAB toolset, the following resources may be useful before you begin the lab.

1. For a thorough yet quick introduction to MATLAB, which will help you get started writing MATLAB scripts (a must for competing this Lab), please visit and complete the "Getting started with MATLAB" tutorial ("Matlab Academy Onramp") located here:

   http://www.mathworks.com/support/learn-with-matlab-tutorials.html

   You will need to have an active MathWorks account in order to sign onto the Academy. Instructions on how to create one are on the MathWorks website.

2. Also useful is a collection of MATLAB code examples found here:

   http://www.mathworks.com/products/matlab/examples.html?requestedDomain=www.mathworks.com

Section 1: Creating a Signal in MATLAB

This exercise will show you how signals can be created using MATLAB. In the following example, you will be shown how to create and plot a signal represented by a sine wave.

First, create a new script in MATLAB and save it as {last_name}_(first_name}_Exercise_{#} (We would also recommend creating new scripts for each exercise in this lab with a similar name structure). To begin, we will be creating a basic sine wave. In further courses you will see that a variety of real world periodic signals can be represented using the sine wave. Normally, a sine wave would be represented digitally as a vector of repeating data points. However, instead of typing in all of these data points by hand, we can use the MATLAB *sin()* function to generate the data points we need. To do this all we need to do is specify the frequency of the sine wave, the sampling frequency, and also the length of time over which we wish to plot.

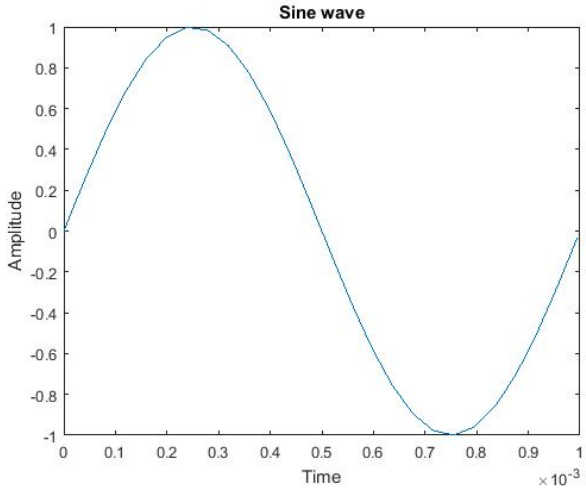
For example:

```
clear all;
close all;

f = 1000;           % This is the frequency of the sine wave. Not to…
                    % be confused with the sampling frequency
w = 2*pi*f;         % omega ('w') is the angular frequency
a = 1;              % maximum amplitude.

fs = 4*w;           %The sampling frequency
dt = 1/fs;
t = 0:dt:0.001;     %length of time for plotting.

y = a*sin(w*t);
figure;
plot(t,y);
```

When ran, the code should generate a plot of a sine wave with a frequency of 1kHz and a maximum amplitude of 1. Before you run the code, it is good practice to label your axes on your plots and to also give them a title. In order to do this, we will use the MATLAB commands title, xlabel, and ylabel (Use the help command to find out how to use these features). To do this, enter the following code segment into your previous script directly after you call the plot command.

```
xlabel('Time');                 %Specify label for x axis.
ylabel('Amplitude');            %Specify label for y axis.
title('Sine wave');             %Specify title for figure.
```

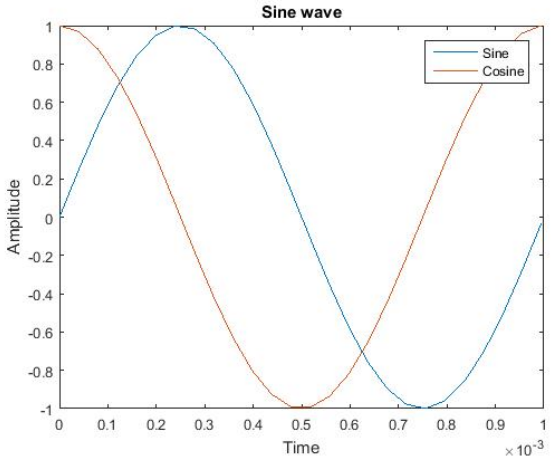
If done properly, you will see a labeled figure which looks like this:



Often times we wish to plot multiple functions on the same graph as this is a quick and easy way to make comparisons (by default, MATLAB will create a separate plot each time you call the plot command). Let's say we wish to plot the cosine wave on the same plot as our sine wave. We want the cosine wave to have the same characteristics of the sine wave (same time base, frequency, sampling frequency, and amplitude). To do this, we will add the following code to our script:

```
z = a*cos(w*t);                 %same w, t, and a as in sine wave.
hold on;                        %tell matlab to use the same plot
plot(t,z);
legend('Sine', 'Cosine');       %create a map legend.
```

We first use the "*cos()"* function, passing the same arguments to it as the "*sin()"* function to generate the cosine wave data. In order to plot this data on the same plot as sine wave, we use the line *"hold on"*, which tells MATLAB to use the same figure to plot the new data. The *"legend()"* command creates a legend with the specified labels. Running the entire script will show a figure which should look like this:



Using the above example, perform the following exercises.

1. Plot a sine wave with a frequency of 2000 Hz and amplitude of 1.5. Set the sampling frequency (fs) to be 5 times of the angular frequency, "w".
2. Plot different sine waves at the same amplitude and frequency as step 1, but vary the frequency at which they are sampled. Plot sine waves sampled at 0.5*w, 1*w, 2*w, and finally 5*w. You can create these on separate plots, over plot them on the same graph (as explained before), or you can make use of the subplot() command to plot them separately but on the same figure. Whatever method you use, be sure to title and label each of these plots and specify the frequency at which they were sampled.
3. Comment on how the shape of the sine wave changes as you change the sampling frequency. Do certain plots look better than others?

Section 2: Importing sound files into MATLAB

This exercise will show you how to import sound files into MATLAB. You will notice that MATLAB will import your sound files as a large vector (which is also called an array) of data points. Don't worry if the data seems large, MATLAB is a tool which can perform operations on large sets of numbers with ease.

1. Download the provided sound file and save it in your folder (*audio.wav*).
2. Place the sound file in the current MATLAB path, to do this you can drag the file directly into the MATLAB "Current Folder" window or you can click the "Browse for folder" button and select the parent directory or folder containing your sound file.
3. Next load the file into the MATLAB workspace. Use the MATLAB command: *audioread()* to do this. Create a variable to hold the data samples and also a variable to store the sampling frequency (use the help command to see how to do this, or browse online documentation). More likely than not, your audio signal will be some two dimensional vector (right and left channels). You can either discard one channel or work with both.
4. Plot your audio signal using the MATLAB plot command. Label the x and y axes appropriately (time and signal amplitude). See MATLAB help on plots for more details. Include this plot in your report. An example of such a plot is given in Figure 3. Note: Your plots will look different than this particular plot. Listen to the audio file and try to see if you can track the correspondence to your plot (where is your sound louder, softer?).

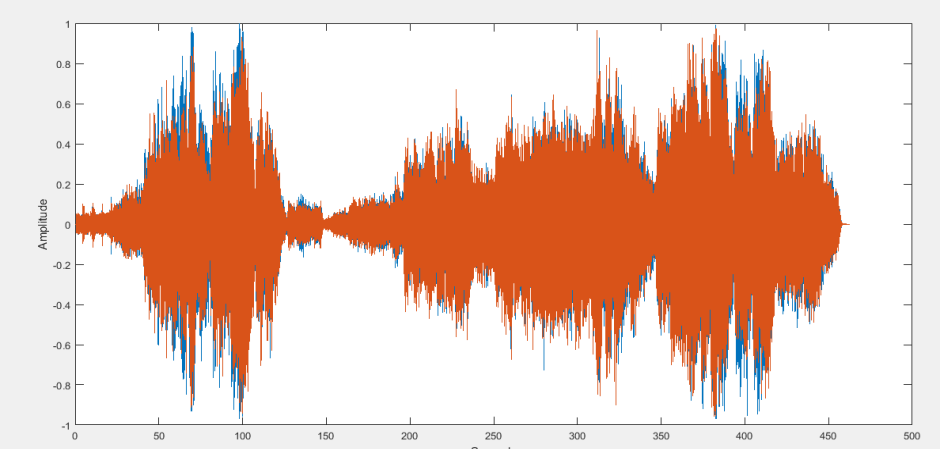


Figure 3: MATLAB plot of a sound file.

Section 3: Sampling an Analog Signal

As stated earlier, a digital signal is created by sampling an analog signal. Since we are interested in recovering the original analog signal, we must be careful when choosing the rate at which we sample our analog signal. If we sample our signal too slowly, the quality of the new signal will degrade in relation to the original. Sampling too fast will result in large amounts of unnecessary data to store and process. This exercise explores the relationship between the quality of a reconstructed signal and the rate at which it was sampled. In particular, we will identify the optimum rate at which to sample a signal by trying different rates and comparing the results to a benchmark.

1. Load the provided audio sample (*audio.wav*) into MATLAB using the same procedure followed in the previous exercise. *Make sure to create both a variable to hold the data samples and a variable to store the sampling frequency.* Be careful not to overwrite these variables in future steps. Since using built in MATLAB functions guarantees that our sampled signal will be almost perfect, we will use this as a standard with which to judge our work. Bonus: Guess the name of the song!
2. Now we will effectively change the sampling frequency of this audio file through a process known as resampling. This is often used in audio processing applications, such as when wanting to transfer the sound recorded on magnetic digital audio tape (48 kHz) to a Compact Disc (44.1kHz).

    - The following example on the MathWorks documentation illustrates how to accomplish this task:
      http://www.mathworks.com/help/signal/ug/changing-audio-sampling-rate.html

    - The following example code will resample the audio signal 'audio.wav' at a new sampling frequency specified by "F_n".

```
[x,Fs] = audioread('audio.wav');
Fn = 1000;              %The rate to resample at.
[P,Q] = rat(Fn/Fs);     %Fn is the new desired sampling rate.
xnew = resample(x,P,Q);
sound(x_samp,Fn);       %Play the new resampled sound.
```

3. Resample the original audio signal at 1 kHz, 1.5 kHz, 10kHz and finally at the standard sampling rate for audio, 44 kHz. Create plots for each of the resampled signals and include each of these plots in your lab write-up. Additionally, use the MATLAB "sound()" function to listen to each of your resampled audio signals. Comment on the differences; do some sound better, worse, or about the same as the original?

   For those of you who are musicians, the terms bass, midrange, and brilliance may be more intuitive in describing sound pitches than just stating frequency ranges. A good page which explains how the audible band (20-~20kHz) is broken up into these ranges is found at this link: http://www.teachmeaudio.com/mixing/techniques/audio-spectrum

Section 4: Time and Frequency Representation

You have seen from the previous exercise what the waveform of a speech signal looks like, you saw how the signal peaks and changes with respect to time, and you may have drawn some conclusions about the relationship between the sound intensity and the peaks on the signal plot. However, even more information can be extracted from this signal if you can consider the signal from a different perspective. We will now consider the signal from the perspective of the *Frequency Domain.*

Most interesting audio signals, such as music, contain different sound components. There may be the low sounds from the bass guitar, higher, sharper sounds from vocals or other instruments, and then midrange sounds. We often ascribe to these different sounds a quality known as “pitch”. In general we consider higher frequency sounds to have higher pitch, and lower frequency sounds to have lower pitch. What if there was a way that we could look at a signal in terms of its “pitches”. To do this, we need to establish a relationship between the time signal and the pitches/frequencies it contains.

The relationship between the Time and Frequency Domain is usually described in terms of a mathematical operation known as a *Transform*, which maps a function in one domain to the other. There are several such transforms that perform this mapping, and there rules which dictate when each of these can be used. For our purposes, the *Fourier Transform* and the *Fourier Series Representation* are the best choices. We will not go into the mathematics of this transform operation; those interested can find more information in the lecture or available online. Because we will not cover these mathematics, we will rely on software tools to do this for us. There are methods which exist that enable digital computers to perform these mathematics very accurately and efficiently, so these operations are almost always performed using a computer tool. What is important at the present time is an understanding of how these operations can aid us in our DSP applications.

In essence, the Fourier transform and series are important because they make the analysis of complicated signals much easier. The idea, developed by French mathematician Joseph Fourier, is to take a complicated signal and break it up into simpler pieces. These smaller pieces are simple waveforms which are represented by sine and cosine functions. These sinusoidal functions are periodic, which means that they repeat themselves. They have a property ascribed to them known as frequency, a quantity which specifies the number of times the sinusoid repeats per unit time. Take a look at the following figure.

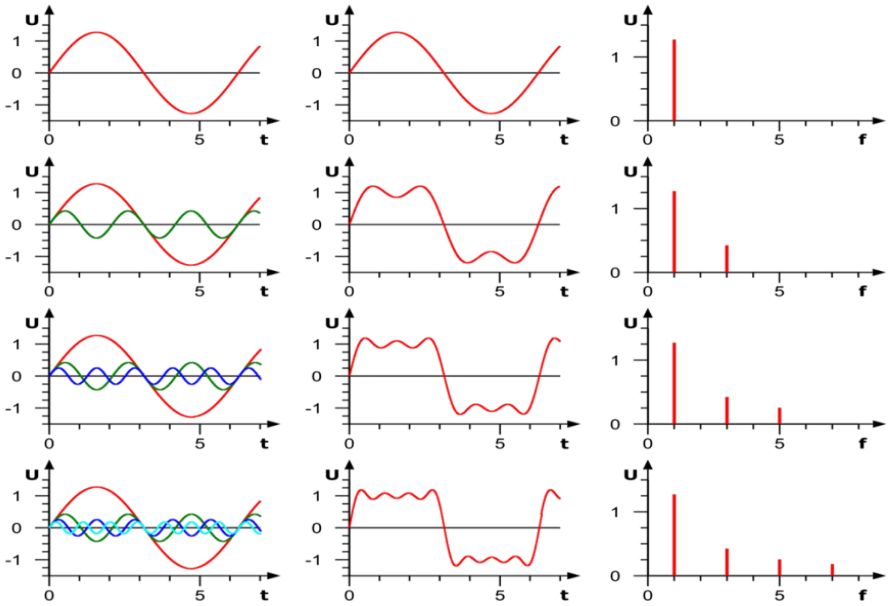


Figure 4: Time domain signals and their spectral components. Source: http://opticalengineering.spiedigitallibrary.org/article.aspx?articleid=1891707

This figure has three columns with three different types of graphs in each column. The first column shows individual sinusoids of different frequencies and amplitudes plotted together, by themselves they are not very interesting. However the second column shows the resulting signals when all the individual components of the first column are added together. As you add more and more sinusoidal signals together, you begin to see how the graph in column three changes. Joseph Fourier (French mathematician who developed the Fourier series) showed that you can create any signal by just adding together a series of sinusoids, and in essence that is the main idea behind the Fourier Series.

Getting to column three from the information in columns one and two is a little more complicated. It involves taking mathematical transform, the Fourier Transform of the signal in column two. The plot in column three is the result of such an operation plotted on an x axis representing frequency. These types of plots are known as *Power Spectrum* plots. The red lines on the plots in column three correspond to different sinusoids of a particular frequency and amplitude. You can interpret the taller lines on the plot as those frequency components of the time signal which have higher power, that is the sine waves in column one with the highest amplitudes. Basically, each of the red lines on the graph represents one component from the graph in column one.

The important concept for this lab is that you know how to construct and interpret the Power Spectrum (PS) plots, and you will be using MATLAB to do this.

1. From Exercise 1.2, you should have variables containing your signals which you have resampled at different rates, and the corresponding frequency at which you performed the resampling. You will now take the Fourier Transform of these variables and plot the results on a PS plot. Use the *fft()* function in MATLAB to compute the discrete time Fourier transforms of your resampled signals. Store your transformed signals into new variables.

2. Make PS plots of each of your transformed signals. You may find the following tutorial useful for doing this:

   http://www.mathworks.com/help/matlab/ref/fft.html

   Label your axes and specify in the title the frequency at which the signal was sampled. Plot the PS for your original signal and compare it to your resampled signal PS plots. You will want to plot what is known as the *single sided spectrum*, the tutorial mentioned above will help you do this. As an example, the following code computes and plots the single sided spectrum of a single sine wave at frequency of 120Hz.

```
Fs = 1000;            % Sampling frequency
T = 1/Fs;             % Sampling period
L = 1000;             % Length of signal
t = (0:L-1)*T;        % Time vector
X = sin(2*pi*120*t);

%Plot the signal in the time domain.

figure(1);
plot(1000*t(1:50),X(1:50))
title('Signal in Time Domain')
xlabel('t (milliseconds)')
ylabel('X(t)')

%Compute the Fourier Transform of X.

Y = fft(X);

%Compute and plot the single sided PSD of X.
P2 = abs(Y/L);
P1 = P2(1:L/2+1);
P1(2:end-1) = 2*P1(2:end-1);
f = Fs*(0:(L/2))/L;
figure(2);
plot(f,P1)
title('Single-Sided Amplitude Spectrum of X(t)')
xlabel('f (Hz)')
ylabel('|P1(f)|')
```

Look at the tutorial link for a detailed explanation of what each of these commands does.

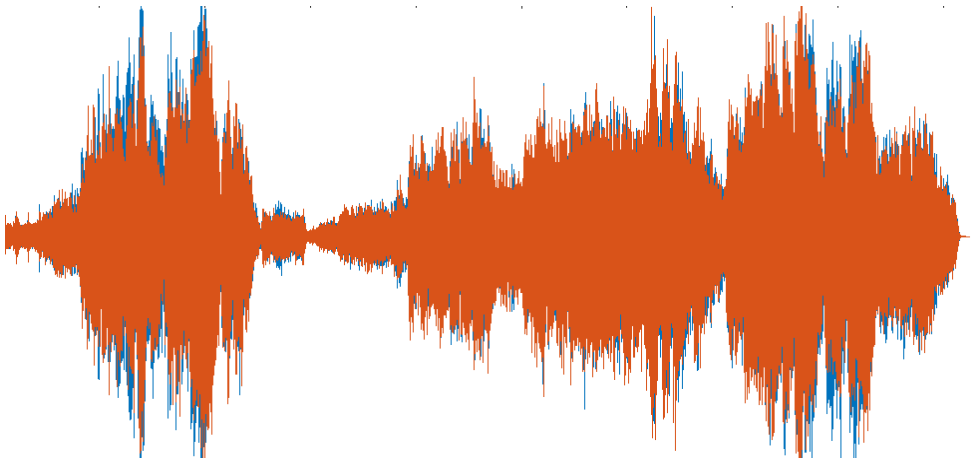
3. Visually inspect each of your PS plots and determine the maximum frequency component which can be represented on the frequency axis. Create a table of two columns and put these values in one column, and then in another column write the sampling frequency of the signal. What is the approximate relationship between these two sets of numbers?

The Sampling Theorem:

You may notice that as you increase your sampling frequency, your resampled signals begin to sound more and more like the original signal. This is no mere coincidence. It is actually the result of what is known as the *Sampling Theorem.* The Sampling Theorem effectively sets the lower limit on what frequency you should sample a signal at. *For the best results, you must sample a signal at a rate which is at least twice the highest frequency component of that signal.* This is why Audio signals are sampled at rates of at least 44kHz. This is because the highest frequency component of the audible frequency band (frequencies the human ear can detect) is 22kHz. Therefore, we must sample our signal at a frequency which is at least twice this frequency, hence 44kHz.

Now from the previous two sections, which of your resampled signals sounded most like the original? What was the frequency at which you resampled that particular signal?

Digital Signal Processing using MATLAB



Electrical and Computer Engineering 100

Exploring Digital Signal Processing

What You Will Need

Materials:

- Provided Image and Sound files.

Equipment:

- Desktop computer / Laptop

Software Tools:

- MATLAB suite with valid license

Part 2: Image Processing

The second portion of this lab will be on Image Processing. Image processing may sound like a difficult topic, however many of you may have already done some basic image processing without even realizing it. For those of you who are familiar with applications such as Instagram may have seen how dramatically an image's appearance can be altered using such applications. With the simple press of a button, you are able to greatly enhance the aesthetics of your photos.

This is one of the goals what image processing attempts to accomplish. While Instagram filters are used to add artistic effects to an image, tools such as Photoshop can perform manipulations to images to remove defects and improve the quality of the image. All of this can fall under the general category of *Image Enhancement*, and while there are other fields of image processing such as Feature Recognition and Computer Graphics, we will focus on improving the quality of images by enhancement.

We will explore how images are constructed from *pixels*, and we will see how we can rearrange the *pixel values* using an *image histogram*. By the end of this lab, you will understand exactly what each of these italicized terms mean.

Section 1: Pixels and Matrices

As mentioned in lecture, the raw voltage output from a CCD can be digitized and transformed into a digital signal by sampling it. The numeric values of that signal are interpreted as the intensity of light striking various pixels of the image. In order to keep track of all of these values, the computer usually stores your image signal in a structure called a matrix, which is just a collection of numbers arranged in columns and rows. For an image, each numeric value represents a *pixel* and the position, which row and column within the matrix, of that numeric value corresponds to the position of the pixel in the overall image.

The following examples are meant to get you accustomed to and familiar with working with images and matrices in MATLAB. Please run each of the following code snippets on your computers.

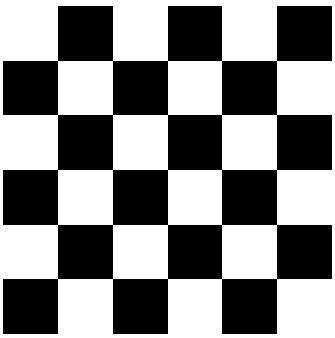
To better understand the way pixels work, we can use MATLAB to develop an 8-bit greyscale image. To do this we can define a matrix filled with numeric values within the range of 0 and 255. 0 corresponds to black and 255 corresponds to white and the pixel definition varies linearly between white and black between 0 and 255, hence the image being greyscale.

```
% 2 dimensional (3x3) matrix A

A =   [0 130 255;
       0 130 255;
       0 130 255;]
```

With matrix A defined in our MATLAB workspace we can now use the *imshow()* function to display / plot the matrix as an image. (Note: (1) the function *uint8()* is used to ensure the variable type is recognizable by the function which requires an unsigned 8 bit integer to display this image correctly (2) *truesize()* and *set()* functions and their inputs will resize pixels to appear larger and enlarge the figure window respectively. See following example.)

```
imshow(uint8(A))
truesize(gcf,[100 100]) % This function is only so pixels appear larger
set (gcf, 'Units', 'normalized', 'Position', [0.1,0.1,0.7,0.7]);
```

Exercise 1a: After understanding the development of a 2-dimensional matrix and functions used to display the defined image, develop a 6x6 checkerboard image using MATLAB as shown here:



More definition can be added to your image by defining a matrix with more rows and columns. Below is a 12x11 matrix. The code illustrates one way to develop matrix A and, then, an equivalent but easier way. MATLAB makes developing and manipulating matrices quick and easy. For example:

```
A = [1     1     1     1     1     1     1     1     1     1     1;
     1     1     1     1     1     1     1     1     1     1     1;
     1     1     1     1     1     1     1     1     1     1     1;
     1     1     1     1     1     1     1     1     1     1     1;
     0    25    50    75   100   125   150   175   200   225   250;
     0    25    50    75   100   125   150   175   200   225   250;
     0    25    50    75   100   125   150   175   200   225   250;
     0    25    50    75   100   125   150   175   200   225   250;
   255   255   255   255   255   255   255   255   255   255   255;
   255   255   255   255   255   255   255   255   255   255   255;
   255   255   255   255   255   255   255   255   255   255   255;
   255   255   255   255   255   255   255   255   255   255   255;  ]
```

is equivalent to:

```
A =   [1*ones(4,11);
       0:25:250;
       0:25:250;
       0:25:250;
       0:25:250;
       255*ones(4,11);]
```

The colon operator ":" and ones() function are used here instead of typing out individual elements of the matrix, this can be easily scaled for your image to include thousands of pixels if desired. And to display the image defined with matrix A we can input the following code:

```
A = uint8(A);
figure(1)
imshow(A)
truesize(gcf,[100 100])
set (gcf, 'Units', 'normalized', 'Position', [0.1,0.1,0.4,0.7]);
title('Image from Matrix A')
```

A simple "transpose" operation may be performed on the matrix by using the "'" transpose operator. By running the following lines of code you can visually understand how a transpose alters an image in a new figure.

```
figure(2)
imshow(A') % Transpose A
truesize(gcf,[100 100])
set (gcf, 'Units', 'normalized', 'Position', [0.5,0.1,0.4,0.7]);
title('Image from Matrix A'' (transposed)')
```

So far we have been working with greyscale images, where pixel intensity values from 0 to 255 represent black, white, and various shades of grey. How about color images, how are they different from greyscale images? First off, color will add depth to your image as well as to the matrix producing it. By adding color, there will be 3 dimensions to your matrix (MxNxD). The additional 3rd dimension (denoted by D), added to the 2-dimnesional case seen prior as greyscale, accounts for the RGB layers (Red, Green, Blue) of the color image. Each pixel now has 3 components, so, we will have 3 matrices to describe the R and G and B pieces of the colored image. Here is an example where you will be able to recognize many similarities to the previous greyscale code:

```
% Color: RGB 3-Dimensional Matrix
  % Overlay 3 Matricies
  clear all; clc; close all;

  A_R = [ 255 0 0;
          255 0 0;
          255 0 0;]
  A_G = [0 255 0;
         0 255 0;
         0 255 0;]
  A_B = [0 0 255;
         0 0 255;
         0 0 255;]

     A = zeros(3,3,3);
     A(:,:,1) = A_R;
     A(:,:,2) = A_G;
     A(:,:,3) = A_B;

  imshow(uint8(A))
  truesize(gcf,[100 100])
  set (gcf, 'Units', 'normalized', 'Position', [0.1,0.1,0.7,0.7]);
```

Here is an example of a larger image where colors overlay and are seen in an additive sense:

```
% https://en.wikipedia.org/wiki/Additive_color
% Overlay 3 Matrices
clear all; clc; close all;

   A_R = [ 255*ones(6,4) zeros(6,1) zeros(6,1)];
   A_G = [ zeros(6,1) 255*ones(6,4) zeros(6,1)];
   A_B = [  zeros(6,1) zeros(6,1) 255*ones(6,4)];

      A = zeros(6,6,3);
      A(:,:,1) = A_R;
      A(:,:,2) = A_G;
      A(:,:,3) = A_B;

   imshow(uint8(A))
   truesize(gcf,[100 100])
   set (gcf, 'Units', 'normalized', 'Position', [0.1,0.1,0.7,0.7]);
```

Exercise 1b: After understanding the development of a 3-dimensional matrix to produce an RGB image and functions used to display the defined image, create a color pattern of your choice with at least a 6 x 6 x 3 matrix.

Section 2: Loading Images into MATLAB

The previous section was to get you familiar with using MATLAB to create and manipulate your own images. You should now understand that images are stored as matrices in MATLAB. Throughout this lab, we will be making use of the prebuilt functions in MATLAB's Image Processing Toolbox. If you decide to take an upper division course in Image Processing/Computer Vision later in your studies, you will learn about the algorithmic implementation of many of these functions. For now, we will just use just use the available functions. Provided is a link to function descriptions for reference:

http://www.mathworks.com/help/images/functionlist.html

Now let's take an existing image and see how we can manipulate it. Before we can do this, we must first load the image into MATLAB. In other words, the image must be placed in the MATLAB "workspace", a special area where MATLAB stores data for use in functions or scripts. There are a number of ways you can do this, simply dragging the image into your workspace will work, however we are going to do this programmatically. Loading an image into MATLAB can be performed using a single function.
Namely, use the following function to load your image ('test.jpg'):

```
IMG = imread('test.jpg');
```

Note that in order to use this method, your image must be in your Current Folder in order for MATLAB to recognize and load it into your Workspace. Here is an alternative method to load an image from anywhere on your computer.

```
IMG = imread('path');
```

Here you must specify the path to the image in the parenthesis of the *imread()* function. Either method should produce the same result, a variable called IMG which stores your image.

Exercise 2a: Do a google images search and download a color JPG format image. Use the above methods to place your image into the MATLAB Workspace. Execute the *imread()* command and store your image as a variable called "IMG".

Notice that after you execute this command, a variable called IMG will be created. Notice that this variable is actually a matrix of numbers, as explained in the first section of this lab. Take a second to recall the significance of this. MATLAB and pretty much all digital image processing software tools store images as just huge matrices of numbers. From mathematics, we can perform mathematical operations on matrices with ease. Many of the effects we will explore when we get to the section on filtering involve simple mathematical operations on matrices. Keep this in mind as you progress through the lab.

Now there are a few questions you should ask yourself at this point. From our previous discussion the answers should hopefully be clear by now. One, why is the image sized the way it is, why does it have three dimensions. Two, what do the values of the numbers mean? Before proceeding to the next exercise, be sure to spend some time thinking about these questions.

Section 3: Exploring the Image Structure

Notice that your dot JPG image is stored as a MxNx3 dimensional matrix (M denotes the number of rows, N the number of columns, and 3 is the number of *layers*). You know that you can represent any color as a combination of Red, Green, and Blue. These layers specify the exact content of each of these RGB building blocks per pixel. To get a better understanding of this, you should break the image up into components and display each layer as a separate image.

If I have an image matrix which is composed of three RGB layers, to get the R layer, I would have to tell MATLAB to slice off the other layers. To do this, if I have an image called "IMG", I would call the following command:

```
R = IMG(:,:,1);
```

Remember that each layer of the image is MxN, so the colons tell MATLAB to preserve all the pixel elements in the rows and columns. The number 1 in the third position tells MATLAB to access the 1st layer, the layer which corresponds to R (red).

Exercise 3a: Separate your image into RGB layers. Execute the previously mentioned command for each layer (R, G, and B, or the 1st, 2nd, and 3rd layers) of the image. Save your results in variables R,G, and B for the next exercise.

In order to display these layers individually, try calling imshow() on your variables for R, G, and B individually. When you call imshow() on the red layer, one would expect to see an image comprised entirely of red pixels? Unfortunately, due to the way MATLAB interprets an RGB image, a bit more work is required.

Look at the dimensions on each of these images, you will notice that they are only MxN and they are missing the third dimension. This is because you have in effect taken the single 3-layered image and have split it up into three single layered images. MATLAB will display Color RGB images if the images have three layers, otherwise it will display the image as greyscale. In order to get around this issue, for each image layer (R,G, and B), we will recreate the other two layers. We will use all zeroes for the pixel values of these reconstructed layers, this will in effect only show the RGB layer of interest when we call imshow() (an RGB image with a nonzero Red layer and zero Blue and Green layers). Here is the code which will accomplish all of the above.

```
a = zeros(size(IMG, 1), size(IMG, 2));

just_red = cat(3, R, a, a);
just_green = cat(3, a, G, a);
just_blue = cat(3, a, a, B);
```

Exercise 3b: Look at the original image; can you guess which of the RGB layers seems to be of most importance or is most prevalent? (Does the image have a redder tone to it? You could say that the Red layer is most prevalent). Display each of the RGB layers using imshow().

After you have identified which layer you think is of most importance, let's see what happens to the image if we alter that particular layer.

Exercise 3c: Scale this layer of your image by some factor which is less than 1. For example, using the following line of code, I can scale down the red layer of my image by a scale factor of 0.2, and then add it back to my original image:

```
new_R = (0.2 .* R);
new_image = cat(3, new_R, G, B);
```

Exercise 3d: Display this new image and compare it to the original image, would you say your image looks better or worse? Comment.

Exercise 3e: Teeth Whitening.

Whitening toothpaste often achieves its goal by taking advantage of a phenomenon known as an optical shift. Basically, a blue pigment contained in the toothpaste is applied to stained teeth during regular brushing. A thin layer of this pigment is deposited which alters the optical qualities (how the light reflected off of the surface of the tooth is perceived by the brain) of the stained tooth and makes the tooth appear visibly whiter. In this exercise, we will simulate the effects of whitening toothpaste on some yellowed teeth using MATLAB and the concepts you have learned so far. Complete the following steps using the provided image, 'stained_teeth.jpg'.

Step 1) Import the image "stained_teeth.jpg" into your workspace.

Step 2) Separate the blue layer from this image. Amplify this layer by scaling it up by some number *greater* than 1.

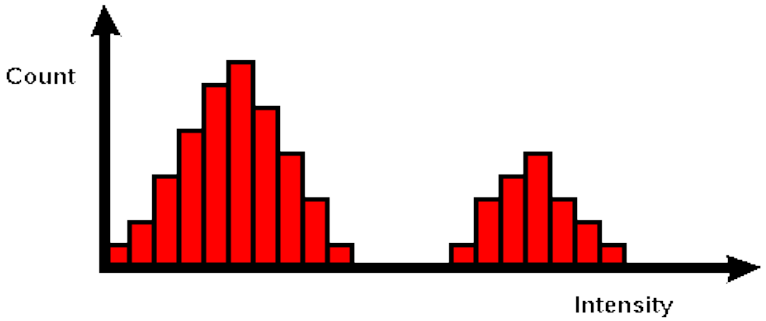
Step 3) Take your new blue layer, and combine it with the original red and green layers of the "stained_teeth" image. (Hint: You can make use of the *cat()* function to do this.)

Step 4) Show the new image next to the old image using *subplot().* Do the teeth appear whiter? If there is no noticeable effect, try increasing your scale factor.

Step 5) Try different scaling factors for the blue layer, display two new images with differently scaled blue layers.

Section 4: Image Histograms

The histogram is an indispensable tool to any engineer working in image processing. By just looking at a histogram, we can get an idea of how the overall image will look, and what adjustments should be made to improve its quality. A histogram is simply a graph which plots pixel intensity values (X axis) and their frequency of occurrence (Y axis). Below is an example of an image intensity histogram:



Source: http://homepages.inf.ed.ac.uk/rbf/HIPR2/histgram.htm

Looking at this particular histogram, how do we interpret it? Well the easiest way is to keep in mind that the horizontal X axis represents pixel intensity values, and that vertical Y axis represents how often these intensity values occur in the image. Knowing that low pixel intensity values represent darker shades, and that high pixel intensity values represent brighter shades, we can infer that the image this histogram represents would have a large distribution of dark pixels and a few very bright pixels, but there would be few midrange gray pixels.

Here is a real example of an image and its associated histogram:

Image with very low contrast, Source: https://visart1.wordpress.com/

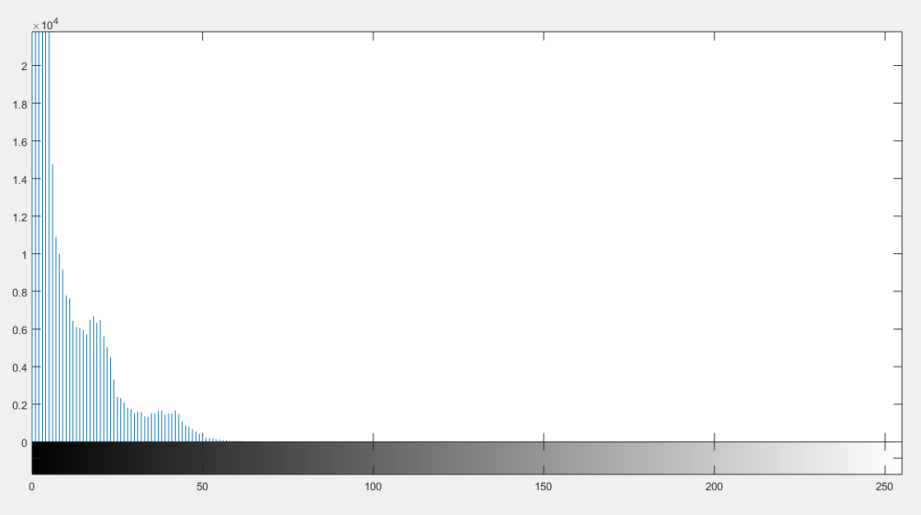
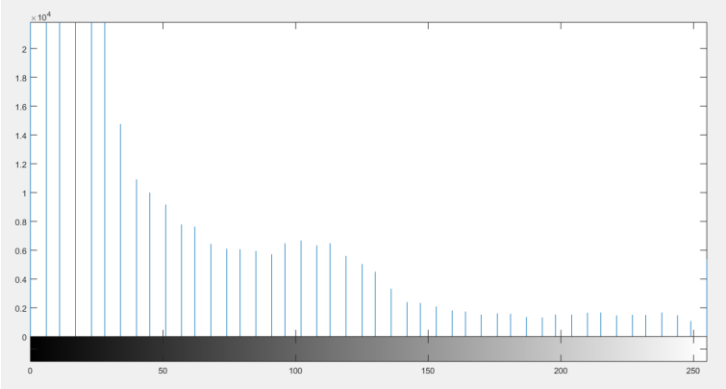


Image histogram showing most pixel intensity values are concentrated at the darker tones/lower intensity values. This explains the darkness of the image.

The histogram shows that most, if not all of the pixels have low intensity values. Because of this, the image has very low contrast and would not be very useful. Thus methods for improving the contrast of these images are of interest to engineers and photographers. Two simple methods which we will explore here are called *Histogram Equalization* and *Contrast Stretching*.

Contrast stretching attempts to spread the pixel values along the entire intensity range (1-255). Applying contrast stretching to the previous image, we obtain a much better spread over the intensity range as seen in the following figures:



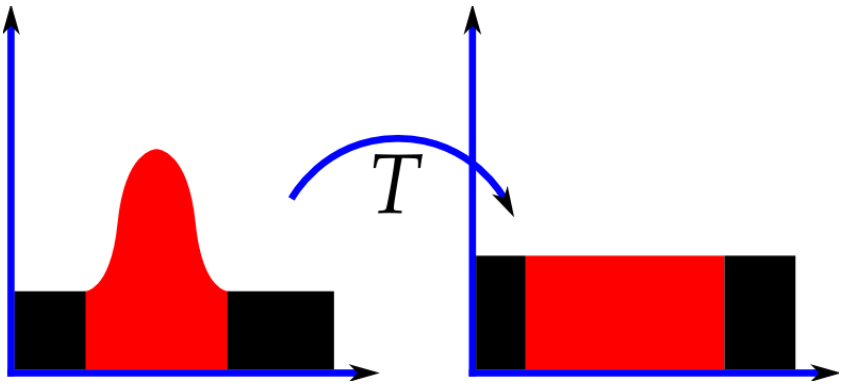
Lightened "Boot" and its associated Histogram.

Contrast Spreading can be achieved by using the *imadjust()* and *stretchlim()* functions (refer to online documentation) and a single line of code.

```
C = imadjust(IMG,stretchlim(IMG),[]);
```

The *imadjust()* function takes 3 arguments. The first is an input image "IMG", and it redistributes the pixel intensity values according to the 2nd and 3rd arguments provided (use the *stretchlim()* function in the 2nd argument, and keep the 3rd argument as default by using empty square brackets).

Another method for improving image contrast is histogram equalization. This method attempts to flatten the distribution of pixel intensity values along the entire intensity range. Ideally, histogram equalization applied to an image would transform the images histogram in the manner of the following image:



Source: https://en.wikipedia.org/wiki/Histogram_equalization

The histogram on the left is flattened to the histogram on the right. This is how histogram equalization attempts to improve the overall contrast of the image.

Plotting an image histogram in MATLAB is a single function call, the following line of code will plot the histogram for an image IMG:

```
imhist(IMG);
```

Exercise 4a: Use histogram equalization on the provided image (dark.jpg) to improve its contrast. To do this, make use of the MATLAB *histeq()* function. Plot the original image and new image histograms.

Exercise 4b: Perform Contrast Stretching on the same image (see previous example). Plot the new image histogram.

Exercise 4c: Compare the original image, the image after Contrast Stretching, and the image after Histogram Equalization. Which looks better?

Section 5: Image Filtering/Manipulation using Kernel Matrices

By now, you know that images are just large matrices of numbers. You also know how to do basic manipulations to those matrices to achieve some desired effect. You have also covered a fundamental tool for image analysis, the intensity histogram. We are now in a position to start exploring more advanced effects such as image filtering. Understand that the word "filtering" is used in the common sense, in that a filter is just something which removes undesirable components and lets the good components pass. In this section, we will start off with image filtering to remove undesirable effects, formally known as "noise".

Noise is just defined as any undesirable visual component of the image. Images can become corrupted by noise in a wide variety of ways; typically it is a result of the sensors we use to capture the image. Since noisy images are not very useful, Engineers and Photographers are interested in developing techniques to remove such undesirable noise from images. For example, take a look at the following images:

The image on the left has been corrupted by "salt and pepper" noise (unwanted black and white pixels, visually similar to common table salt and pepper). The image on the right is the same image with the noise removed. The most common way to remove this particular form of noise is by using what is called a *median filter*. This filter belongs to a broader class of filters known as Kernel Filters, which we shall explore here.

Kernel Filters are basically small matrices which when applied to a larger matrix (such as the matrix representing your image), produce some useful effect. In formal terms, you may hear engineers speak of the *convolution* of a kernel filter with a matrix. Convolution is just a mathematical operation which explains what happens when the image is filtered; it is an intermediate topic which will be treated in detail in your further courses.

Exercise 5a: Navigate to this link which contains a very useful visualization of what a Kernel Filter is and how it affects an image: http://setosa.io/ev/image-kernels/

Spend some time applying different kernel filters to the provided example image on the site. Pick your favorite Kernel Filter and save a copy of the resulting image after you apply that filter. Be sure to label the image with the name of the filter you used.

We will now use functions in MATLAB to achieve these same effects.

Exercise 5b: Noise removal using kernel filters.

Step 1) Import provided image Penguin.jpg into your workspace.

Step 2) Plot the image and its histogram. The image contains two forms of noise, White Gaussian Noise and Salt and Pepper Noise.

Step 3) Remove the Salt and Pepper noise using a median filter (*medfilt2* function). Plot the new image and its histogram. Compare it to the original histogram.

Step 4) In order to remove the Gaussian White Noise, we will use a mean averaging filter. In order to do this, you will need to use the *fpsecial*() function to create your Kernel Matrix, and the *imfilter()* function to apply it to your image. See the following link for examples on how to do this:

http://www.mathworks.com/help/images/ref/fspecial.html

Step 5) Plot the image with all noise removed and its histogram. Again, compare this histogram to the original.

Optional

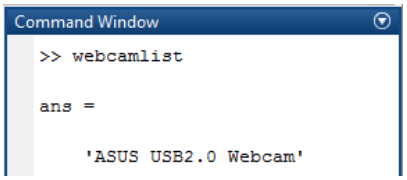
In theory, one could create their own kernel matrix and apply it to an image of their choice. Typically one uses a prebuilt Kernel to perform basic manipulations, but for those interested, here is the skeleton of the code to build and apply a custom Kernel "h" to any image "x".

```
%Image manipulation using Kernels
%Writing your own kernels

close all

h = [-2 -1 0; -1 1 1; 0 1 2];       %My Kernel Matrix
x = rgb2gray(imread('Barbara.jpg')); %My image
y1 = (filter2(h,x));                %Applying my Kernel Matrix

figure; subplot(1,2,1)
imshow(x, [])
title('Original Image')
subplot(1,2,2)
imshow(y1, [])
title('Altered Image')
set(gcf,'Position', get(0,'Screensize'));
```

Section 6: Photo booth Challenge

1. Download the MATLAB USB Webcams Support Package if not installed already.
   a. Home → Add-Ons → Get Hardware Support Packages → Install from Internet → Search for "USB Webcams" → Next → Accept terms/conditions and install

2. In the command window type, "webcamlist". It will reply back with the name(s) of the webcams installed or attached.



3. Set the webcam to be an object.

```
>> cam = webcam(1)
```

4. Test to see that your webcam is on and working by using the *preview()* function

```
>> preview(cam)
```

Up top on the preview window you can see the camera name. On the bottom part of the window you can see the camera name, resolution, frame rate, and the timestamp (in seconds).

You can close the preview with the function *closePreview().*

5. Take a picture!
To take an image with your webcam use the function *snapshot().* Use the function *imshow()* to display your image.

```
>> img = snapshot(cam);
>> imshow(img)
```

6. To save your image use the function *imwrite().*

```
>> imwrite(img, 'myfirstimage.jpg');
```

7. Close the webcam using the function *delete().*

8. Photobooth Challenge!
As in the prior challenges to Lab 3, you can now take that image and do some signal processing! Take the image from your webcam and create your very own photobooth reel! Use your knowledge from lab 3 to change each photo in the reel showing different types of effects. For instance Photo 1 can be black and white, Photo 2 can be the original image but flipped, Photo 3 create your own!

Check out this link for some interesting signal processing ideas:
http://blogs.mathworks.com/steve/category/special-effects/

Here is example code you may use to better understand how each of the functions above work. Since each step is separated by "%%" you should "run section" one section at a time instead of running the entire code at once.

```
%% Photobooth Challenge for ECE 5
% The purpose of this program is to take a photo from a webcam and do some
% signal processing with the image. We will be manipulating the pixels by
% various means such as changing the color or flipping the image.

% Resources - Much of the code can be found on the previous part of Lab 3
% or http://blogs.mathworks.com/steve/category/special-effects/

%% Clears figure windows, variables, commands, etc.
clear all; close all; clc;

%% Shows what webcams are connected to the computer
webcamlist

%% Creates webcam as an object and tests to see that webcam is working.
cam = webcam(1); % input maybe 2 or 3 if using attached webcam
preview(cam);

%% Take an image with webcam and saves as JPG
image = snapshot(cam);
imwrite(image, 'imagetesting.jpg');
imshow('imagetesting.jpg');     % displays the image

%% Closing windows
closePreview(cam);      % closes the preview window
delete(cam);            % closes the webcam
```

Cyan

```
% Creates different filter colors by changing the R, G, or B values
 R = img(:,:,1);
 G = img(:,:,2);
 B = img(:,:,3);

 z = zeros(size(img, 1), size(img, 2));

 cyan = cat(3, z, G, B);
```

Original

Color

```
% Contrast Enhancement of the image
 SE = strel('Disk', 18);
 imgEnhanced = imadjust(img,[0.20 0.00 0.09; 0.83 1.00 0.52],...
     [0.00 0.00 1.00; 1.00 1.00 0.00], [1.10 2.70 1.00]);
```

Black and White

```
% Creating a black and white image
 x = rgb2gray(imread(fileName));
 black_and_white = uint8(zeros(480, 640, 3));
 black_and_white(:,:,1) = x;
 black_and_white(:,:,2) = x;
 black_and_white(:,:,3) = x;
```

Flipped Image

```
% Flipping an image
a = fliplr(img(:,:,1));
b = fliplr(img(:,:,2));
c = fliplr(img(:,:,3));

flippedImage = cat(3,a,b,c);
```

Decorrelation

```
% Decorrelation streching - useful for visulaizing image by reducing
% inter-plane autocorelation levels in an image.
 colorImg = decorrstretch(img);
 colorImg = imadjust(colorImg,[0.10; 0.79],[0.00; 1.00], 1.10);
```

Other Filter

```
just_red = cat(3, R, z, z);
yellow = cat(3, R, G, z);
just_green = cat(3, z, G,a);
just_blue = cat(3, z, z, B);
purple = cat(3, z, a, z);
```

Other Saturation

```
imgEnhanced1 = imadjust(img,[0.00 0.00 0.00; 1.00 0.38 0.40],...
    [1.00 0.00 0.70; 0.20 1.00 0.40], [4.90 4.00 1.70]);
imgEnhanced2 = imadjust(img,[0.13 0.00 0.30; 0.75 1.00 1.00],...
    [0.00 1.00 0.50; 1.00 0.00 0.27], [5.90 0.80 4.10]);
imgEnhanced3 = imadjust(img,[0.20 0.00 0.00; 0.70 1.00 1.00],...
    [1.00 0.90 0.00; 0.00 0.90 1.00], [1.30 1.00 1.00]);
```

```
%% Altering the photos and plotting as photoreel
fileName = 'myfirstimage.jpg';      % Replace with the name of your image
pic = imread(fileName);
img = imresize(pic, [480 640]);         % Resizes the image

% Creates different filter colors by changing the R, G, or B values
 R = img(:,:,1);
 G = img(:,:,2);
 B = img(:,:,3);

 z = zeros(size(img, 1), size(img, 2));

 cyan = cat(3, z, G, B);

% Creating a black and white image
 x = rgb2gray(img);
 black_and_white = uint8(zeros(480, 640, 3));
% uint8() fixes the matrix size so it can go in photoreel
 black_and_white(:,:,1) = x;
% stores the black and white image into different layers
 black_and_white(:,:,2) = x;
 black_and_white(:,:,3) = x;

% Contrast Enhancement of the image
 SE = strel('Disk', 18);
 imgEnhanced = imadjust(img,[0.20 0.00 0.09; 0.83 1.00 0.52],...
     [0.00 0.00 1.00; 1.00 1.00 0.00], [1.10 2.70 1.00]);

% Decorrelation streching - useful for visulaizing image by reducing
% inter-plane autocorelation levels in an image.
 colorImg = decorrstretch(img);
 colorImg = imadjust(colorImg,[0.10; 0.79],[0.00; 1.00], 1.10);

 % Flipping an image
 a = fliplr(img(:,:,1));
 b = fliplr(img(:,:,2));
 c = fliplr(img(:,:,3));

 flippedImage = cat(3,a,b,c);

%% Creating the final photobooth strip.
 % list image name in desired order
 CompositeImage = [img; cyan; imgEnhanced; black_and_white;...
                   colorImg; flippedImage];
 imshow(CompositeImage);
```

Some of the useful image processing functions used above include rgb2gray, imadjust, decorrstretch,and image adjust. Use the reference and help menu to explore these functions further. The new images are combined at the end in a single column for the photobooth reel. This works because each matrix is the same size in the orientation they are concatenated, in this case, having the same number of columns since they are combined one on top of the other.

References

In addition to the web links provided in the lab, here are some classic resources on Image Processing which give both a practical and theoretical treatment of the subject.

Pratt, William K. Digital Image Processing. New York: John Wiley and Sons, 2001

Gonzales, Rafael K. Woods, Richard E. Digital Image Processing 3E. New Jersey: Prentice Hall 2007

Manipulating Sound

Electrical and Computer Engineering 100

Manipulating Sound

Overview

The objective of this lab is to introduce you to analog circuit design using OpAmps. We will study this in the context of manipulating signals by building and testing the following circuits: an amplifier, a low-pass filter, a high-pass filter, and a band-pass filter. Each of these circuits will be tested across a wide range of frequencies to observe their behavior. In addition to these circuits, we will also learn about representing signals in both the time domain and frequency domain as well as how to read schematic and implement the circuit on a bread board.

Time vs. Frequency Domain Signals

Before we learn about filters, we first need to understand the difference between the time domain and the frequency domain. Time domain refers to the analysis of a signal with respect to time, whereas frequency domain refers to the analysis of a signal with respect to frequency. The transformation from the time domain to the frequency domain is done through the Fourier Transform. (You will learn this in ECE 45 and ECE 101.) The reason why we sometimes prefer to use the frequency domain is because it can be easier to analyze. For example, sound is usually described in the frequency domain, since processing and shaping of the audio signals are easier described and designed in the frequency domain.

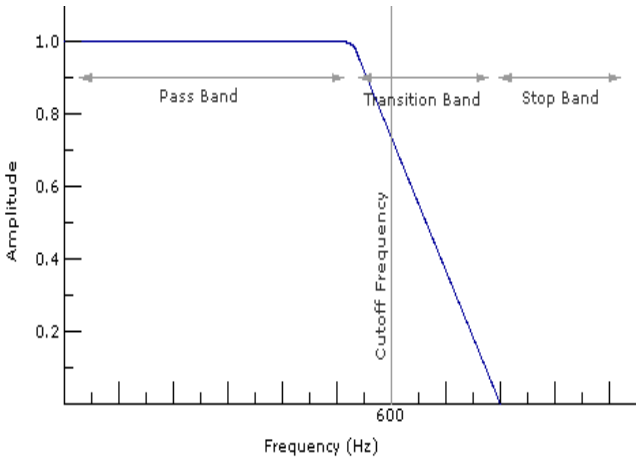
Abbreviations

Decibel: The intensity of sound is often reported in units of *decibel (dB)*. Decibel is given in a logarithmic scale, such that:

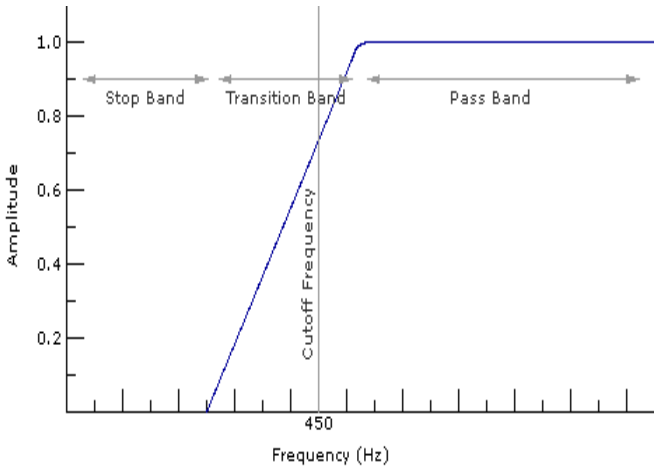
$$dB = 20\ log\left(\frac{A}{A_0}\right)$$

Cutoff (Corner) Frequency: The frequency at which energy flowing through a system is beginning to be reduced. Typically, we use -3dB as the cutoff point because it is the point where voltage of the system is

decreased from the maximum to half power.



Low pass filter with a cutoff frequency of 600Hz



High pass filter with a cutoff frequency of 450Hz

Hertz: The SI unit of frequency of vibrations, such as sound. Hertz is measurement of the number of cycles per second.

What You Will Need

Materials:

- Resistors (1× 10Ω, 1× 680Ω, 2× 220Ω, 3× 1kΩ)
- Potentiometer (1× 10kΩ)
- Capacitors (5× 0.1µF)
- Electrolytic Capacitor (1× 10µF, 1× 100uF, 1× 1000µF)
- Diode (1× 1N4001 or similar)
- Op-Amps (2× LM741, 1×LM386)
- 9V Battery (1×)
- Battery strap/connector (1×)
- Microphone (1×)
- Speaker (1× 8Ω)
- Audio jack (1×)
- Switches (1× SPST, 1×SPDT)
- Breadboard (1×)
- Protoboard (1×)
- Wires (lots!)

Machinery:

- Computer / Laptop
- National Instruments VirtualBench (or a Power Supply + Function Generator + Oscilloscope)
- Soldering Iron

Software:

- National Instruments VirtualBench Software (immediately loads after connecting Virtual Bench to the computer)

Challenge #1: Using a Microphone

1. Build the microphone circuit shown above in Figure 1. The breadboard implementation of this circuit is provided in Figure 2.
2. Set up the virtual bench using the following settings:
    - Power supply: +2V
    - Turn on math channel:
        - Math Operation --> FFT
        - Center: 800Hz with 200Hz per division
3. Adjust the math trigger so that the spectrum can be seen clearly on the screen.
4. Using your phone or computer, play the two files (note.wav and note2.wav) from the course website. What is the fundamental frequency of each signal?

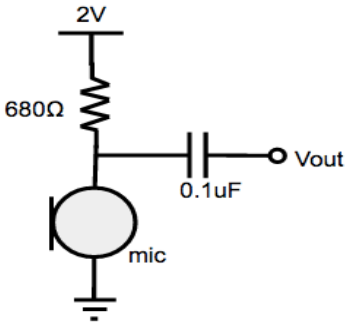


Figure 1. Schematic of microphone circuit.

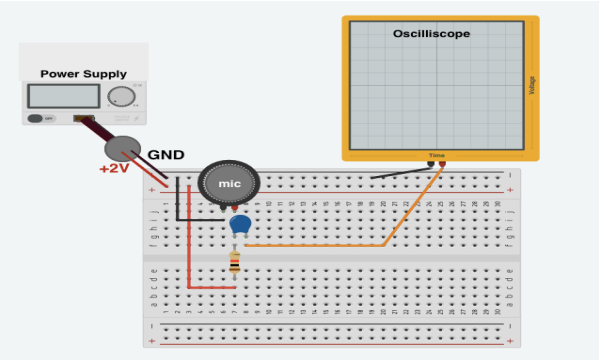


Figure 2. Breadboard implementaiton

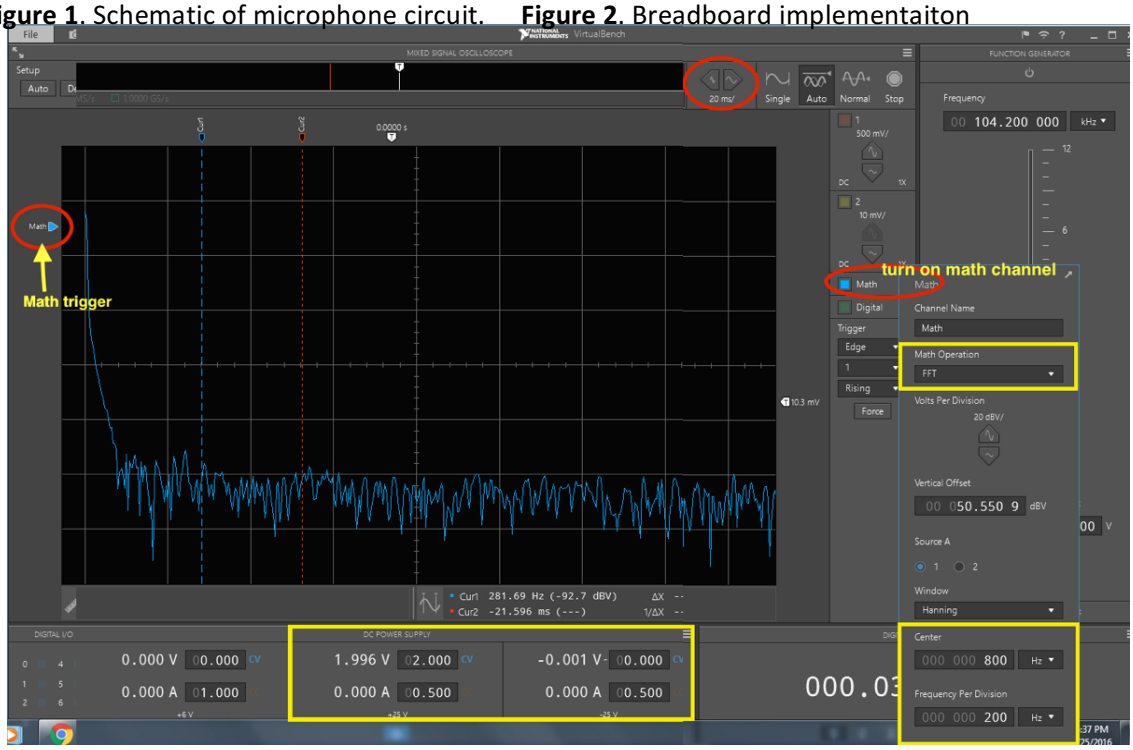


Figure 3. VirtualBench interface. (Can also use Power Supply + Function Generator + Oscilloscope)

Challenge #2: Creating an Amplifier

1. Build the amplifier circuit shown above (Figure 4). The breadboard implementation of this circuit is provided in Figure 5 (**Note:** Study this and make sure that you understand it since you will need to do all of the others on your own!)
2. Supply an input voltage v_{in} using the function generator feature of the VirtualBench. Use the following settings:
   Amplitude = 500mV_{pp}, Offset = 0V, Waveform = Sine, Frequency = 100Hz. Refer to Figure 6 below for reference.

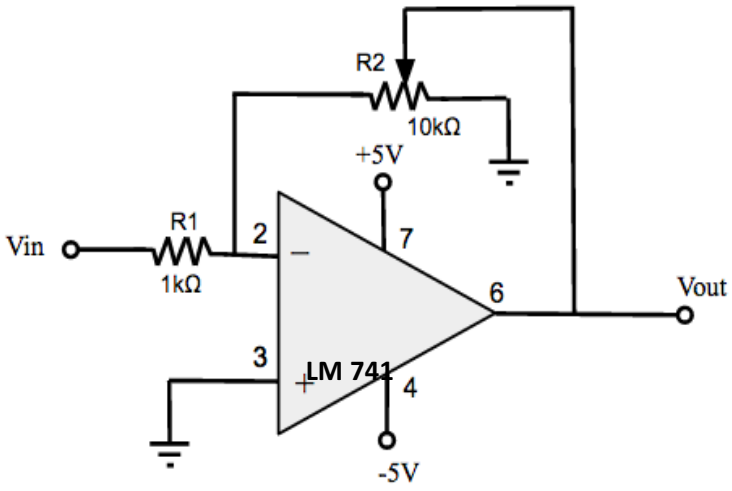


Figure 4. Amplifier schematic.

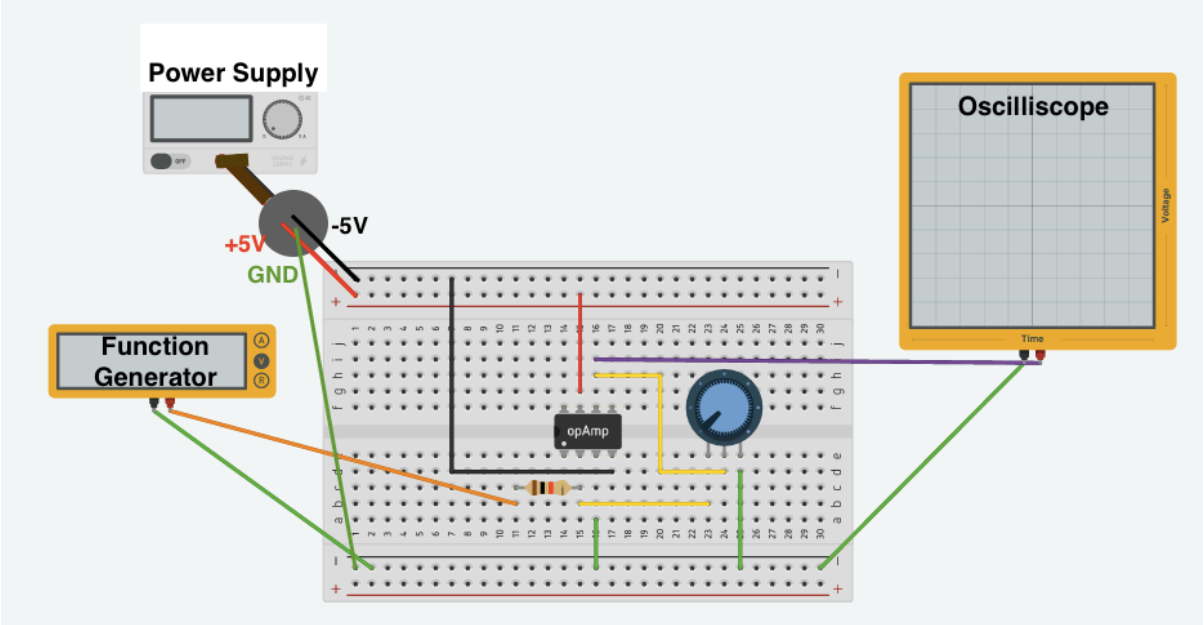


Figure 5. Breadboard implementation.

3. Observe the input and output voltages using the oscilloscope. Connect channel 1 to the input and channel 2 to the output. Save the input and output signals from the oscilloscope. (**Note:** You should see a sine wave at the same frequency at the output. If you do not, check that you have built the circuit properly.)
4. Adjust the knob on the potentiometer (R_2), and observe the change in the gain. Derive a relationship for the gain ($A_v = v_o/v_i$) as a function of the two resistor values $R_1$and R_2. (**Note:** In order to measure $R_1$and R_2, you must either remove R_2 from the circuit or turn off the power supplies and function generator.)
5. Leave the potentiometer at a fixed position. Change the frequency of the input signal. Record and plot the gain of the amplifier as a function of the frequency for the following frequencies: 50, 100, 500, 1k, 5k, 10k Hz. Does this amplifier have a cutoff frequency?

6. Change the function generator to output a square wave (with the same amplitude and frequency). What does the output look like?

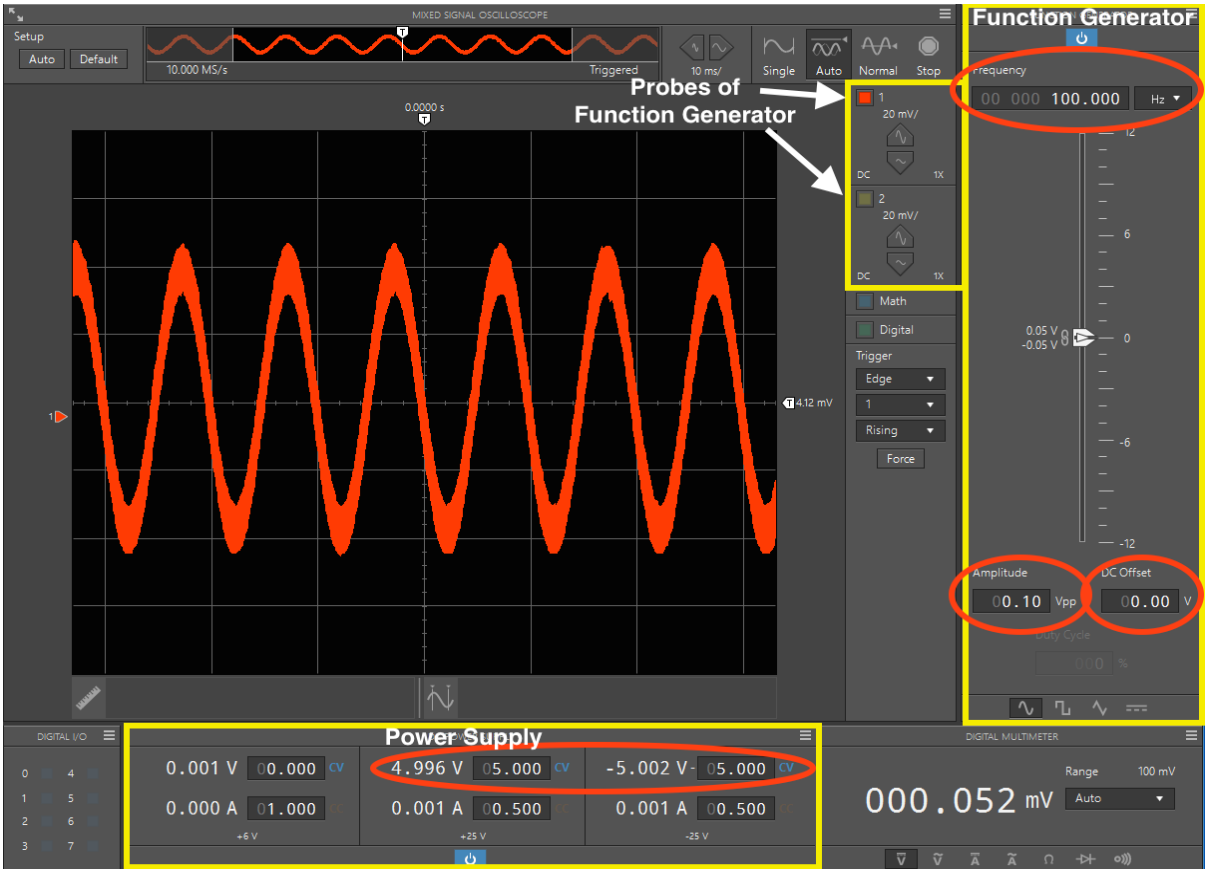


Figure 6. VirtualBench settings.

Challenge #3: Low-Pass Filter (LPF)

1. Build the active low-pass filter shown in Figure 7. (**Hint:** Look back at Figure 4 and Figure 5 to make sure that you understand how to translate a schematic to the breadboard!)
2. Supply an input voltage v_{in} using the function generator. Use the following settings:

   Amplitude = 500mV_{pp}, Offset = 0V, Waveform = Sine, Frequency = 100Hz.

3. Observe the input and output voltages using the oscilloscope. Connect channel 1 to the input and channel 2 to the output. (**Note:** You should see a sine wave at the same frequency at the output. If you do not, debug your circuit.)
4. Change the frequency of the input signal and observe how the output signal changes. Observe the amplitude of the signals and save the input and output signals from the oscilloscope at the following frequencies: 50, 100, 500, 1k, 5k, 10k Hz. Plot the gain of the filter (in dB) versus frequency (log). Estimate the cutoff frequency of this filter. Why is this called a "low-pass" filter?
5. Change the function generator to output a square wave (with the same amplitude and frequency). What does the output look like? What happens if you add an offset voltage of 100 mV? What happens if you change the frequency to 10 kHz?

NOTE: Do **NOT** take circuit apart. The circuit will be used in challenge #5.

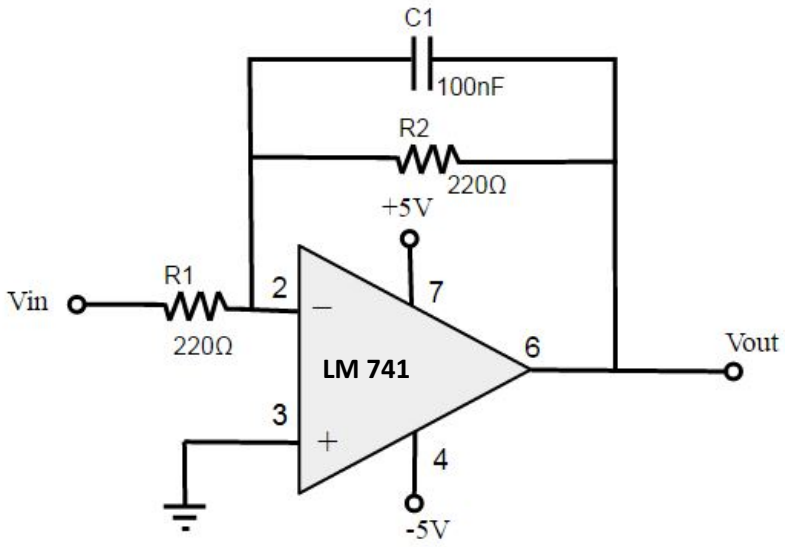


Figure 7. Schematic of a low-pass filter.

Challenge #4: High-Pass Filter (HPF)

1. Build the active high-pass filter shown in Figure 8.
2. Supply an input voltage v_{in} using the function generator. Use the following settings:

   Amplitude = 500mV_{pp}, Offset = 0V, Waveform = Sine, Frequency = 100Hz.

3. Observe the input and output voltages using the oscilloscope. Connect channel 1 to the input and channel 2 to the output. (**Note:** You should see a sine wave at the same frequency at the output.)
4. Change the frequency of the input signal and observe how the output signal changes. Observe the amplitude of the signals and save the input and output signals from the oscilloscope at the following frequencies: 50, 100, 500, 1k, 5k, 10k Hz. Plot the gain of the filter (in dB) versus frequency (log). Estimate the cutoff frequency of this filter. Why is this called a "high-pass" filter?
5. Change the function generator to output a square wave (with the same amplitude and frequency). What does the output look like? What happens if you add an offset voltage of 100 mV? What happens if you change the frequency to 10 kHz?

NOTE: Do **NOT** take circuit apart. The circuit will be used in challenge #5.

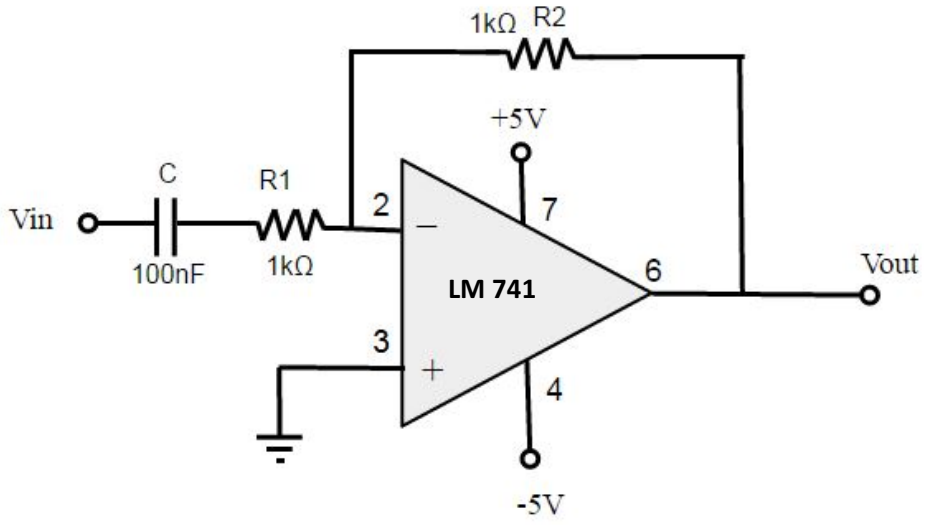


Figure 8. Schematic of a high-pass filter.

Challenge #5: Band-Pass Filter (BPF)

A band-pass filter can be made by cascading a low-pass filter with a high-pass filter.

1. Using both the HPF and LPF you previously built, build a band-pass filter as shown in Figure 9.
2. Supply an input voltage v_{in} using the function generator. Use the following settings:

   Amplitude = 500mV_{pp}, Offset = 0V, Waveform = Sine, Frequency = 1kHz.

3. Observe the input and output voltages using the oscilloscope. Connect channel 1 to the input and channel 2 to the output. (**Note:** You should see a sine wave at the same frequency at the output.)
4. Change the frequency of the input signal and observe how the output signal changes. Observe the amplitude of the signals and save the input and output signals from the oscilloscope at the following frequencies: 50, 100, 500, 1k, 5k, 10k Hz. Plot the gain of the filter (in dB) versus frequency (log). What are the cutoff frequencies and what is the bandwidth?
5. Change the function generator to output a square wave (with the same amplitude and frequency). What does the output look like? What happens if you add an offset voltage of 100 mV? What happens if you change the frequency to 100 Hz? What about 10 kHz?

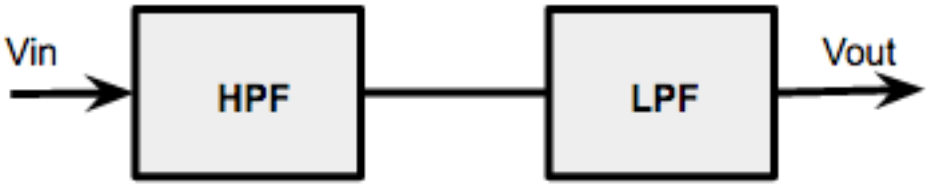


Figure 9. Block diagram for band-pass filter.

Challenge #6: Audio Amplifier

Now that we understand a bit more on how signals flow through analog circuits and various ways to manipulate them, we want to build our own portable audio amplifier and speaker system that we can plug into any music playing device. We will setup and solder a commonly used circuit design to make this happen using capacitors, resistors, and a new Op-Amp.

There are many different types of Op-Amps available for various applications. LM 741 is a general purpose Op-Amps , and can be used in most applications, even though it may not be the best option all the time. In this part of the lab, we will be using LM386, a low voltage amplifier made specific for audio amplification. For more detailed information on the LM386. Please refer to Appendix 1 and the LM386 datasheet.

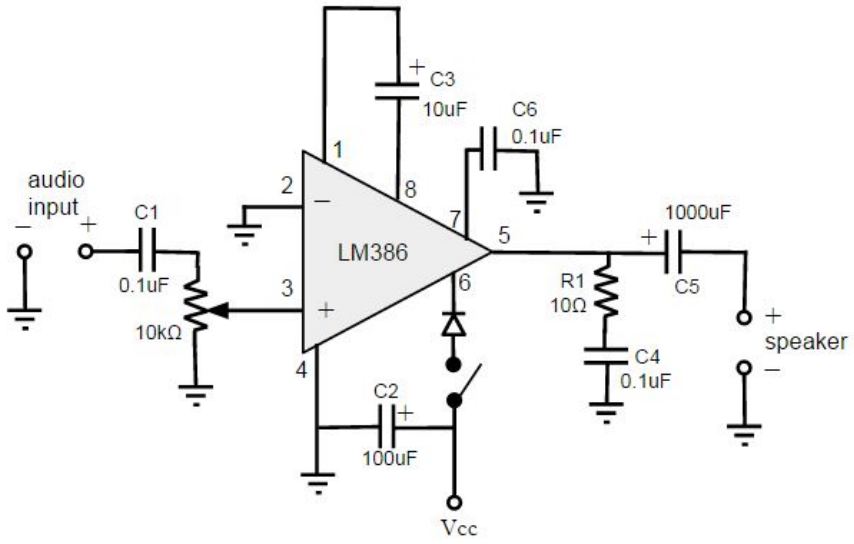


Figure 11. Schematic of speaker amplifier using LM386.

1. Using a breadboard, build the circuit according to the given schematic in Figure 11. In this circuit, C_3 sets the gain of the amplifier to 200. C_6 acts as a bypass capacitor to prevent degradation of gain, and possible instabilities. C_2 connected between power and ground helps maintain the supply voltage. The potentiometer controls amplitude of the input signal, adjusting the volume of the sound. C_1 and C_5 remove DC offset from the input and output signals respectively. R_1 and C_4 forms a Zobel network when connected in parallel with the speaker, removing high frequency noise.
2. Connect the audio jack by soldering the **sleeve** of the jack to the GND wire and the **ring** to the audio input wire. (**Note:** We do not need to use the tip since we only have one audio output, the speaker.) Refer to Figure 12 for details.
3. Add a diode between V_{CC} and pin 6 for safety reasons. If you by accident switched the polarity of the battery, the diode will not allow the current to flow.
4. Connect the 9V battery to V_{CC} (+) and GND (-). Use the function generator to test your circuit first by using it to generate a 1kHz sine wave with an amplitude of 200mV. (If you do not hear anything, stop and debug your circuit.)
5. Using the function generator, sweep the frequency to plot the frequency response of the circuit. What is the lower cutoff frequency? What is the upper cutoff frequency? What is the gain of this amplifier?

6. Turn on the switch and plug the audio input to your computer or MP3 player. Check to make sure it is working properly.
7. Now that you know your speaker is working, solder the circuit as instructed in class. Enjoy your music!

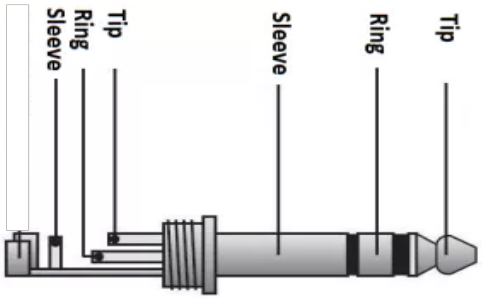


Figure 12. Audio jack pin-out.

Optional Challenge: Add-on microphone input

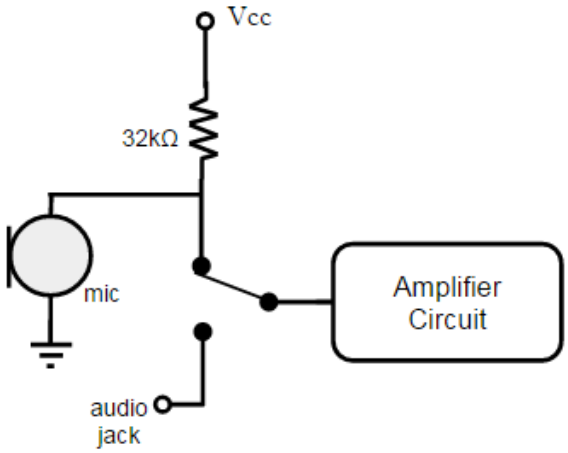


Figure 13. Allows switching between microphone and audio jack.

Now that you have a working audio amplifier, you can add an electret microphone to the circuit and have the amplifier circuit take the signals from the microphone as the input.

1. Connect the electret microphone to the input of the amplifier from challenge #6. Remember to power the microphone and add a resistor to ensure that no more than 0.5mA of current is flowing through the electret microphone.
2. Double check your connections, then test the circuit by speaking into the microphone.
3. Once the circuit is working, you can add a SPDT (Single Pole Double Throw) switch that allows you to switch between the microphone input and the audio jack input.

Other Challenges: Audio Related Projects

There are a wide variety of audio related projects available online. Here are some possible online projects that you can try out:

1. Tone Control Circuit
   http://www.instructables.com/id/Active-tone-control-circuit/

2. Guitar Distortion Petal
   http://www.instructables.com/id/Make-an-easy-guitar-distortion-pedal-STEP-BY-STEP/

3. Overdrive Pedal
   http://www.instructables.com/id/Overdrive-Pedal/?ALLSTEPS

4. Design a 3-D Printed Speaker Case

Appendix I - Resources

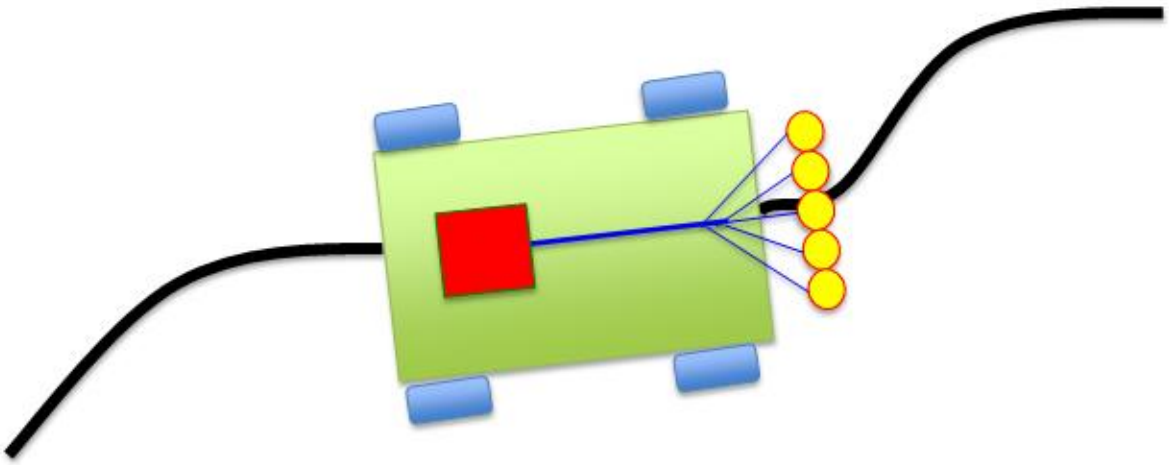
This is a really good resource for understanding how sound signals are processed! Go here to learn more about filters:
http://beausievers.com/synth/synthbasics/

This is the datasheet for Low Voltage Audio Power Amplifier LM741:
http://www.ti.com/lit/ds/symlink/lm741.pdf

This is the datasheet for Low Voltage Audio Power Amplifier LM386:
http://www.ti.com/lit/ds/symlink/lm386.pdf

This article is a good resource for understanding how to select an appropriate op amp for different applications:
http://www.ti.com/lit/an/slyt213/slyt213.pdf

Robotics



Electrical and Computer Engineering 100

Embedded Systems and Control:
Line Following Robot

Objective

The objective of this final project is to create a line following robot using much of the knowledge gained throughout the quarter and adding to it, a better understanding of sensors, actuators, programming, systems, and controls. Upon completing this objective, you will have the opportunity to compete against your classmates and push your creativity further in controller and system design.

Outline

1) **Potentiometers**
2) **Photoresistors**
3) **The Cart**
4) **Motor Driver and Arduino Mega**
5) **Hardware Challenges**
6) **System Integration**
7) **Calibration of Photoresistors**
8) **PID Control**
9) **Competition**
10) **Appendix**

What You Will Need

Materials:

- 1 Arduino Mega
- 1 USB Cable A-B
- 1 Cart Chassis
- 1 Caster
- 1 Adafruit Motor Shield
- 3 Resistor (330 Ohms)
- 3 LEDs
- 2 DC Motors
- 2 Wheels
- 2 Breadboards
- 2 Battery Packs
- 4 Potentiometers
- 7 Resistors (10k Ohms)
- 7 Photoresistor
- 9 AA Batteries
- Colored Wire
- Solder
- Electrical tape

Machinery:

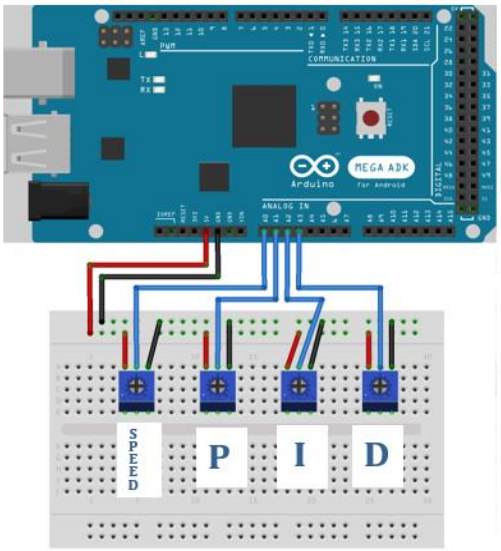
- Computer / Laptop
- Solder Station
- Screw Driver

Software:

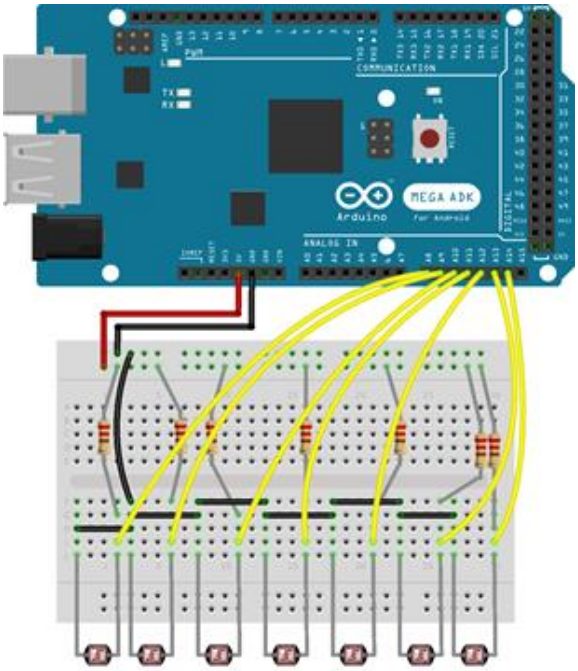
- Arduino Software (IDE)

Challenge #1: Potentiometers

For the first challenge, set up potentiometers on a bread board and follow the wiring diagram shown on the right. This challenge is not setting up your PID controller, it is only setting up a way to interface/change your Speed, P, I, and D coefficient values (variables) without the need to re-upload new code to your Arduino each time. Use the code in Appendix 10.1. Test this to ensure it prints the potentiometer output to the serial monitor. Remember that a potentiometer acts as a variable resistor. Do counter clockwise or clockwise twists of the potentiometers increase the value printed to the serial monitor? What are different ways to switch it from counter-clockwise to clockwise or vice versa?



Challenge #2: Photoresistors



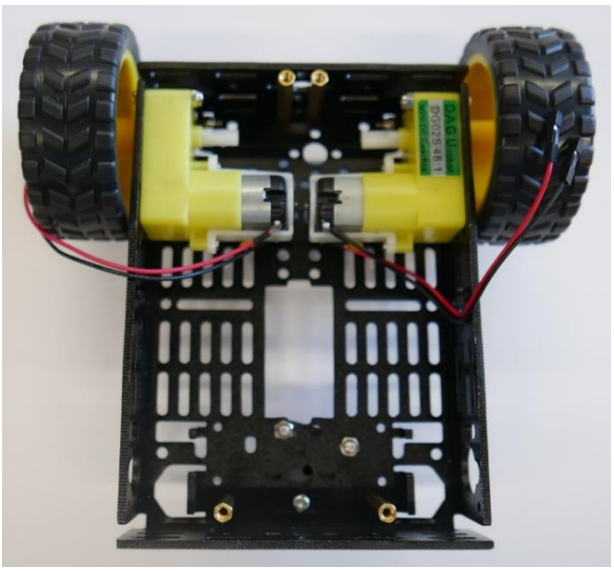
Develop multiple voltage dividers with, instead of variable resistors being made of knobs, they are now sensors that change resistance dependent on light. The voltage dividers use 10kΩ resistors.

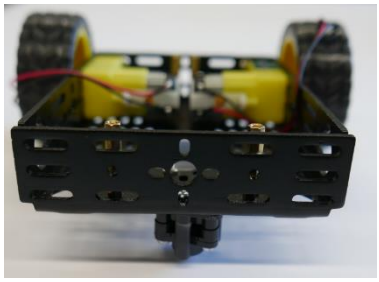
Once you have the board set up, use the code in Appendix 10.2 where you can read the values from each of your photo resistors and print them to the serial monitor. Test this to be sure each photo resistor is working properly.

Try placing it close to different colored surfaces and then try it out on a line. What is the best distance for your sensors to be away from the surface to properly differentiate when the sensors are hovering over black or white? How does surrounding light or shadows affect the readings?

Challenge #3: The Cart

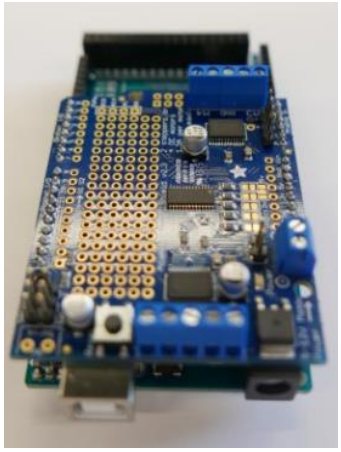
Now we would like to put together the cart. Use the instruction manual found with your cart kit. For our design we will only need 2 motors and 2 wheels. The front two wheels and motors will be replaced with a caster. The caster fits very close to the center front in our design.



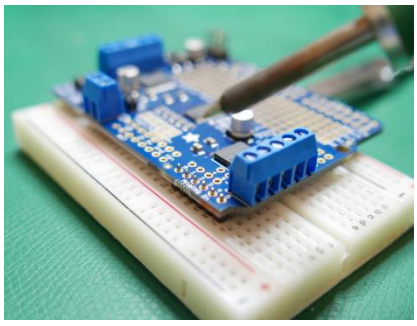


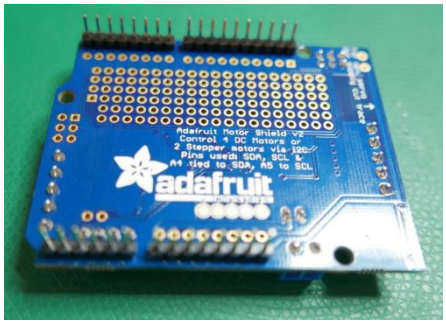
Challenge #4: Motor Driver and Arduino Mega

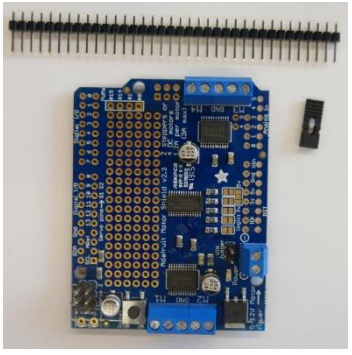
For the fourth challenge, first solder the header pins onto the motor shield. It helps to solder your first one or two pins of each strip while it is attached to an Arduino or a breadboard for alignment purposes. Check out the motor shield manual for some helpful hints. Note that the header pins on the outer edge of motor shield seen in the figures should have the longer ends capable of connecting to the Arduino. The header pins just on the inside of those should have the longer ends on the other side for easy connectivity to other devices using female jumper wires.



Use the motor shield manual to download and install the Adafruit motor shield library for Arduino. Without the motors connected, but with the motor shield attached to the Arduino, ensure the code from Appendix 10.3 compiles and uploads without error.

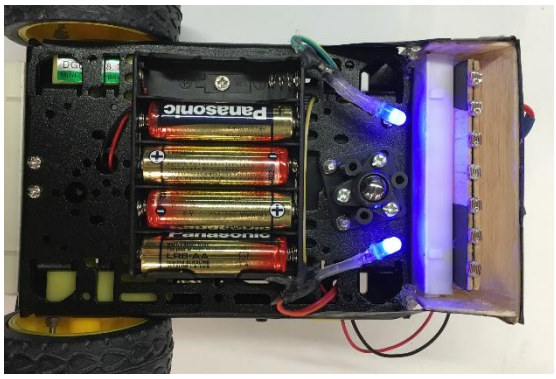






Challenge #5: Hardware (at least 2)

Light Shield and Light Source Additions:



One problem that arises is the robot's sensitivity to ambient light changes and shadows. Add a light shield and light source on your robot to help mitigate this problem. You may do something similar to the robots shown below. You can be creative with the 3D printer follow the 3D printing tutorial, or simply cover with paper or balsa wood. Add LEDs that are always on by taking advantage of the 5V rails on the breadboard of your photoresistors and don't forget to add a resistor in series with each of your LEDs.

3D Print/laser cut Chassis:

The bulky metal chassis may not be optimal for this line following exercise and competition. Use your newly learned CAD and 3D printing and laser cutting skills to design your own chassis. Minor modifications to the current metal design are valuable as well.

H-bridge replacement of Adafruit motor shield:

For a line following robot, it is not necessary to have a full motorshield. Design your own H-bridge / half – bridge circuit to drive the motors.

PID indicator:

Currently, only knobs are used to set the value on your PID controller. Create your own realtime indicator of what your PID values are. Be creative using LEDs, servos or other to illustrate these values.

3D Print Wheels / Caster:

Design and 3D print new wheels that can attain improved performance possibly through larger distance per revolution, traction, and style.

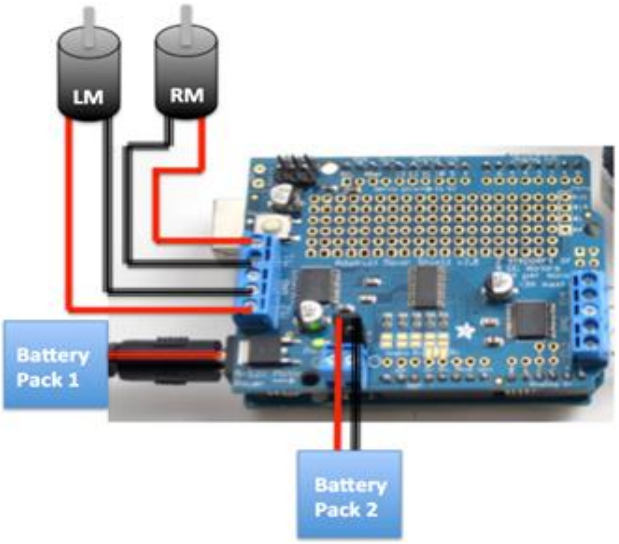
Collision avoidance:

It is unfortunately very common that these robots will run into a wall, another robot, or fall of a table. Using sensors are your Arduino to detect one of these and avoid a disaster.

Propose your own:

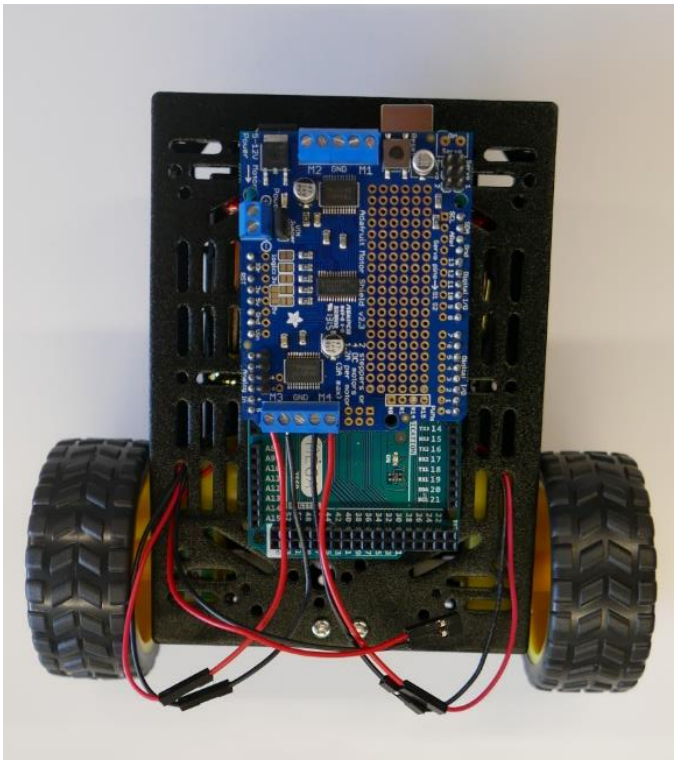
Propose your own hardware addition using design tools, sensors, and/or actuators. Must be approved by a TA or Professor.

Challenge #6: System Integration Part 1



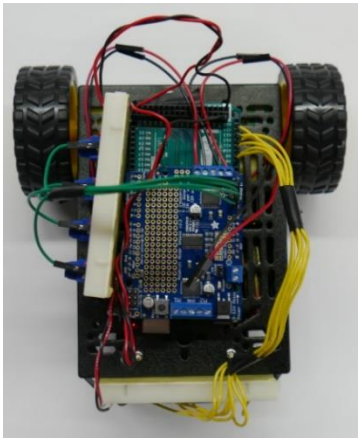
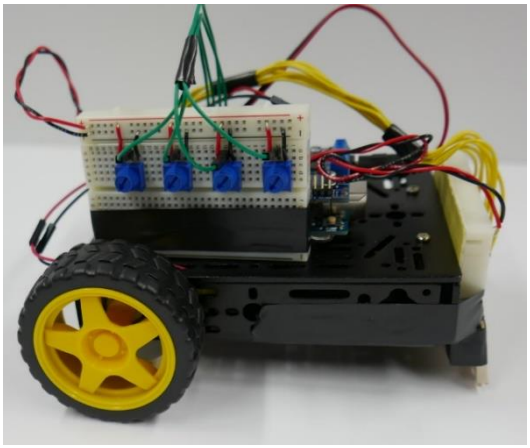
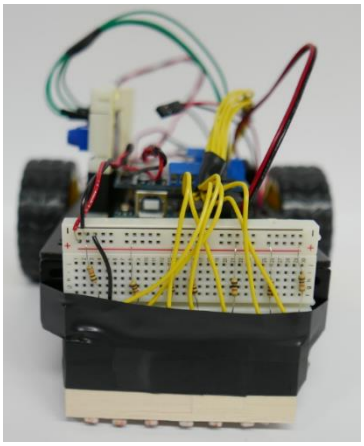
Looking at the diagram on the left, connect your motors to M1 and M2 on your motor shield. Ensure both power sources are available to connect, one for the Arduino Mega, and one for the motor shield. Do not connect the power sources and have your computer plugged into Arduino through USB at the same time.

Code it! You can use the code in Appendix 10.3 or build your own. You can use the Appendix 10.3 test code to move forward for 3 seconds and backward for 3 seconds. After reading through the code, can you try another maneuver like going in a circle or making a figure 8. Be careful/maybe do not try on top of a table or near coffee, because it can be pretty fast.



Note: With code 10.3, if you notice that one wheel moves forward and one wheel moves backward, or both wheels move backwards when it should be moving forward, you could simply change the polarity of wires coming from your motorshield from M1 and M2 by inserting the black wire where the red wire is and vice versa. You can also us use M3 and M4 slots instead of M1 and M2 with a small code alteration declaring M3 and M4 as the motors to "attach" instead of M1 and M2.

Challenge #6: System Integration Part 2

First, there are many ways to put your robot together from here.

Remember to consider the height of those photoresistors from the ground for good sensitivity measuring white vs. black surfaces. It may also be good to use a thin piece of balsa wood or thick paper to keep photoresistors aligned and kept from being easily bent.

Tape helps with nearly everything including putting your bread boards in place holding the Arduino Mega to a specific spot on your cart, keeping the wires harnessed/organized together, etc. Tape does not help so much with coding.

Run each of the programs from Challenges #1-#3 (10.1-10.3) individually. Does each potentiometer work? Does each photoresistor work and sense the difference between a black and white surface? Does the cart still drive forward and backward?

Challenge #7: Calibration

Calibrating your sensors is very important since sensor readings can easily alter due to changes in sensor position/alignment, changes in surrounding lighting, or other various reasons. This challenge will walk you through the process of calibrating your device using code found in Appendix 10.4. The main goal with your calibration is to take the possible readings from each sensor shown at the top of the two figures below, and then map those readings to a more uniform measure making it more clear where a black line may be located.

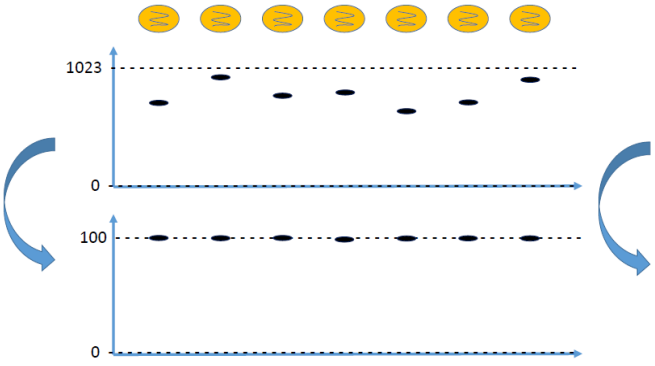


Figure 1: Mapped reading for a black line

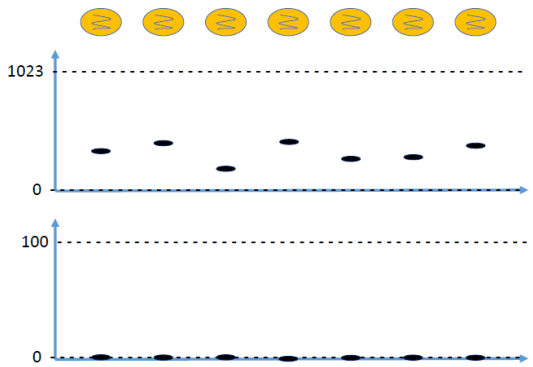


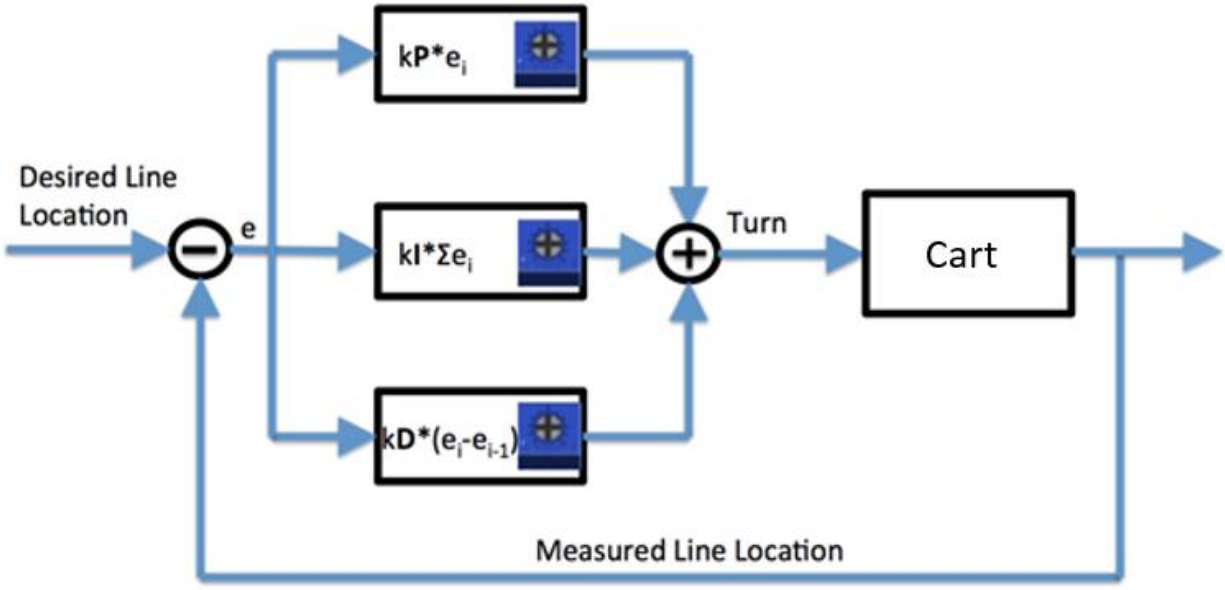
Figure 2: Mapped reading for a white line

Before implementing the new code, you may want to add an LED to make the high/low output on pin 13 more visible or choose another more accessible digital pin to control the LED, like digital pin 31 since pin 13 is hiding below the Adafruit Motor Shield. This LED is used to help you know what step of the calibration you should be performing.

After that, read through the code and comments found in Appendix 10.4. Test this code by uploading to the Arduino and following these steps:

1. While the connected LED is blinking slowly, place the photoresistors attached to the breadboard and cart over a white surface.
2. After a few seconds the LED will blink quickly and at this point it is important not to touch the cart or cast any shadows over it because it is finding the nominal photoresistor reading of each pin for what correlates with pure white.
3. The LED will then blink slowly again giving you time to move the photoresistors over a pure black surface or line. Ensure that each photoresistor is fully over the black surface.
4. Again, after a few seconds the LED will blink quickly and again it is important not to touch the cart or cast any shadows over it because it is finding the nominal photoresistor reading of each pin for what correlates with pure black.
5. Now the sensors should be calibrated and the photoresistors/Arduino should be able to distinguish clearly between what is black and white and the readings should be more uniform moving between 0 and 100. You should see these readings through the serial monitor.
6. Error, or distance from the centerline of the cart, is then calculated through a weighted average.

Challenge #8: PID Control

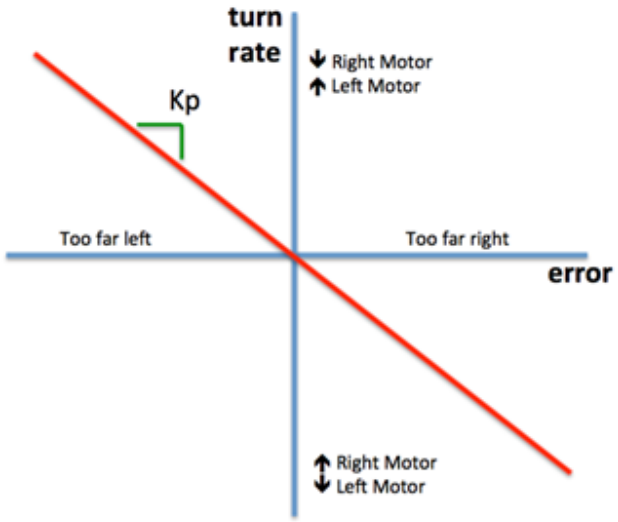


Now you will be able to apply what you learned in this week's lecture on PID control. In the past four sections you have been putting together each of the main components seen in the block diagram above, and now you will need to organize your code in a manner to connect everything together. The first step is to realize what is your error, e, and how to quantify it. This is what your sensors are for and your error may be found from comparing your measured value to your desired value.

error = (measured value - desired value)

You now need to operate on this error and you may break this portion down into 3 main operations: proportional, integral, and derivative control.

Proportional control, as explained in class, takes your current error value and scales it by your proportional gain, kP. This scaling should transform your error into a desired turn rate to correct for that error. The tank's turn rate is controlled through your left motor and right motor. If your error indicates that you are slightly to the right of the line you will need to increase the speed of your right motor and/or decrease speed to your left motor. You can imagine that as you are further and further to the right your turn rate will increase in magnitude to respond to this error more intensely. The next figure explains this proportional control graphically.



Integral control takes into account a sum of your past errors. This can be implemented into your code with:

sumerror = (sumerror + error)

This can be considered as memory of your error because as your error remains on one side of center, it will grow into a more and more impactful role affecting your turn rate. This can eliminate small errors and steady state error since as your error is small, maybe your proportional error won't affect the system enough to push it into your desired state, but as the error persists the gain, kI will be multiplied by a greater and greater "sumerror" to counter that small yet persistent inaccuracy.

Derivative control can predict an increase or decrease in your error and prepare your turn rate in advance. You can imagine this being very helpful on a sudden and sharp turn. By taking your current error and subtracting your previous error, you will be able to see how much your error has changed during this one time step.

deriverror = (error – previous error)

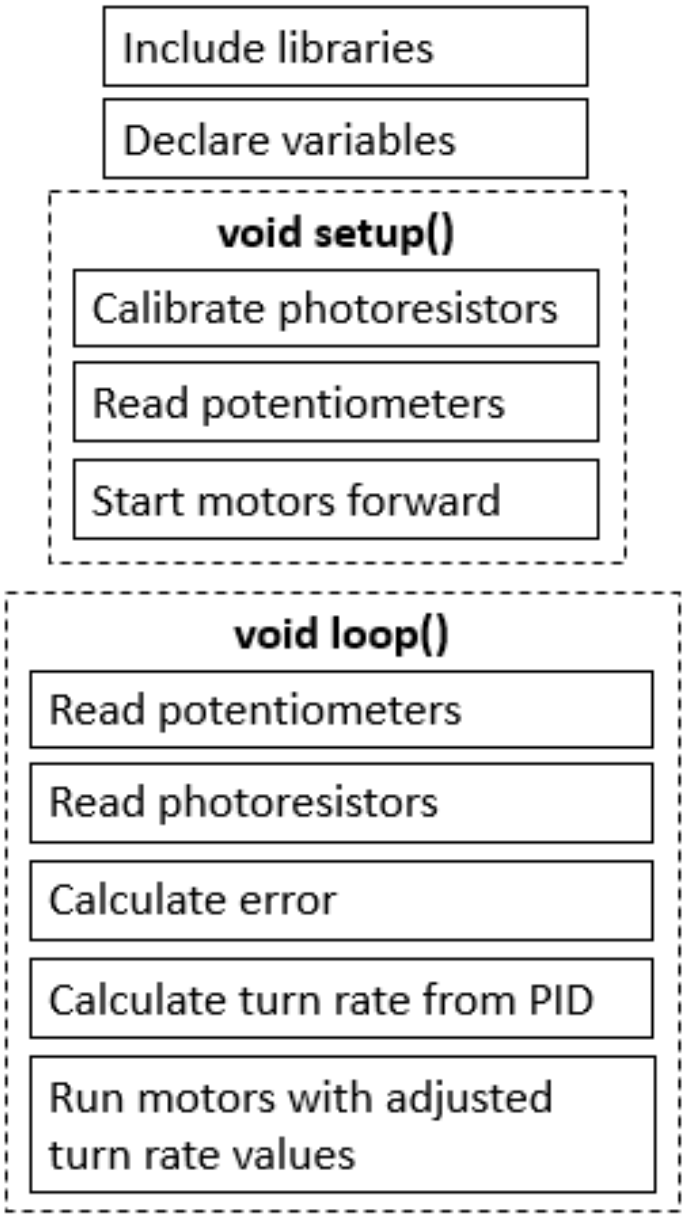
Odds are high that, unless corrected for, the rate at which your error changes will remain the same from time step to time step. By recognizing this worsening (or improvement) of your error, you will be able to increase (or decrease) the amount of additional right or left motor speed necessary to counter such a change.

Now we can add all of these terms together to calculate our turn rate to control our tank.

*Turnrate = (kP*error + kI*sumerrror + kD*deriverror)*

This turn rate should be transferred to an increase or decrease in motor speed for your left and/or right motor.

Code Flow:


Include libraries
Declare variables
void setup()
Calibrate photoresistors
Read potentiometers
Start motors forward
void loop()
Read potentiometers
Read photoresistors
Calculate error
Calculate turn rate from PID
Run motors with adjusted turn rate values

Challenge #9: Competition

1) A series of line courses

Each course will have different criteria for score including (1) distance of line followed, (2) time to complete certain portions of the course, (3) error off center at certain points, etc. With the ability to control your speed and PID knobs, your robot will compete against your classmate's robots.

2) Innovation/Creativity challenge

Using any of the sensors or actuators you have learned about throughout the quarter or any others that are mentioned above, you will have the opportunity to piece together additional components to improve your robot's control, add to usefulness, or increase the level of interactivity/fun.

Challenge #10: Appendix

Note: Arduino (.ino) files are also provided for 10.1-10.5 so it is not necessary to copy and paste

10.1 - Potentiometer Code Outline

```
/* Potentiometer Code Outline */
 /* Declare Variables for Potentiometer */
   const int S_pin = A0; // proportional - analog pin 0
   const int P_pin = A1; // proportional - analog pin 1
   const int I_pin = A2; // integral - analog pin 2
   const int D_pin = A3; // derivative - analog pin 3
   int Sp = 0; // speed gain coeficient
   int kP = 0; // proportional gain coeficient
   int kI = 0; // integral gain coeficient
   int kD = 0; // derivative gain coeficient

void setup() {                /* Setup - runs once (when power is supplied or after reset) */

  Serial.begin(9600);        // For serial communication set up
}

void loop() {                 /* Loop - loops forever (until unpowered or reset) */

  ReadPotentiometers();     // Call on user-defined function to read Potentiometer values

  Print();                  // Call on user-defined function to print values from potentiometers

}

//
**********************************************************************************************
//
// function to read and map values from potentiometers
void ReadPotentiometers()
{
      Sp = map(analogRead(S_pin),0,1023,0,100);
      kP = map(analogRead(P_pin),0,1023,0,100);
      kI = map(analogRead(I_pin),0,1023,0,100);
      kD = map(analogRead(D_pin),0,1023,0,100);
}

//
**********************************************************************************************
//
// function to print values of interest
void Print()
{


 Serial.print(Sp); Serial.print(" ");
 Serial.print(kP); Serial.print(" ");
 Serial.print(kI); Serial.print(" ");
 Serial.println(kD);

 delay(200); //just here to slow down the output for easier reading if desired

}
```

10.2 Photoresistor Code Outline

```
/* Photoresistor Code Outline */
  /* Variables for Light Sensors*/
   int LDR_Pin0 = A8 ; // analog pin  8
   int LDR_Pin1 = A9 ; // analog pin  9
   int LDR_Pin2 = A10; // analog pin 10
   int LDR_Pin3 = A11; // analog pin 11
   int LDR_Pin4 = A12; // analog pin 12
   int LDR_Pin5 = A13; // analog pin 13
   int LDR_Pin6 = A14; // analog pin 14

   // Initialize Photo Resistor Variables
   int LDR0 = 0, LDR1 = 0, LDR2 = 0, LDR3 = 0, LDR4 = 0, LDR5 = 0, LDR6 = 0;

void setup() {

   Serial.begin(9600);        // For serial communication set up

}

void loop() {

   ReadPhotoResistors(); // Read photoresistors and map to 0-100 based on calibration

   Print();              // Print values to serial monitor
}

//
**********************************************************************************************
//
// function to read photo resistors
void ReadPhotoResistors()
{

 LDR0 = analogRead(LDR_Pin0);
   delay(2);
 LDR1 = analogRead(LDR_Pin1);
   delay(2);
 LDR2 = analogRead(LDR_Pin2);
   delay(2);
 LDR3 = analogRead(LDR_Pin3);
   delay(2);
 LDR4 = analogRead(LDR_Pin4);
   delay(2);
 LDR5 = analogRead(LDR_Pin5);
   delay(2);
 LDR6 = analogRead(LDR_Pin6);
   delay(2);
}

//
**********************************************************************************************
//
// function to print values of interest
void Print()
{

 Serial.print(LDR0); Serial.print(" ");
 Serial.print(LDR1); Serial.print(" ");
 Serial.print(LDR2); Serial.print(" ");
 Serial.print(LDR3); Serial.print(" ");
 Serial.print(LDR4); Serial.print(" ");
 Serial.print(LDR5); Serial.print(" ");
 Serial.println(LDR6);

 delay(200); //just here to slow down the output for easier reading if desired

}
```

10.3 Motor Driver Code

```
/* Motor Driver Code Outline */
// Libraries for Motor
#include <Wire.h>
#include <Adafruit_MotorShield.h>  // Must add libary - see MotorShield Manual
//https://cdn-learn.adafruit.com/downloads/pdf/adafruit-motor-shield-v2-for-arduino.pdf

// Initialize Motors
Adafruit_MotorShield AFMS = Adafruit_MotorShield();
Adafruit_DCMotor *Motor1 = AFMS.getMotor(1); // Motors can be switched here (1) <--> (2)
Adafruit_DCMotor *Motor2 = AFMS.getMotor(2);

//Set Initial Speed of Motors
int M1Sp = 60; // initial speed may vary and later can be increased with Sp potentiometer
int M2Sp = 60;


//Set LED Pin
int led_Pin = 13;  // can change to another digital pin and connect extra LED to me more easily
seen


// setup - runs once
void setup(){
 Serial.begin(9600);
 AFMS.begin(); //for Motor

 pinMode(led_Pin, OUTPUT); // set pin mode to output voltage
      // Gives you a moment before tank actually moves
        for (int waitii = 0; waitii < 20; waitii++) {
            digitalWrite(led_Pin, HIGH);   // turn the LED on (HIGH is the voltage level)
             delay(100);              // wait for 100 milliseconds
             digitalWrite(led_Pin, LOW);      // turn the LED off by making the voltage LOW
             delay(100);              // wait for 100 milliseconds
           }
 }


// loop - loops forever
void loop(){

    // Start Motors in forward direction
       Motor1->setSpeed(M1Sp);
       Motor1->run(FORWARD);
       Motor2->setSpeed(M2Sp);
       Motor2->run(FORWARD);
     delay(3000);             // let run forward for 3 seconds

    // Start Motors in backward direction
       Motor1->setSpeed(M1Sp);
       Motor1->run(BACKWARD);
       Motor2->setSpeed(M2Sp);
       Motor2->run(BACKWARD);
     delay(3000);             // let run backward for 3 seconds

    // Stop Motors
       Motor1->setSpeed(M1Sp);
       Motor1->run(RELEASE);
       Motor2->setSpeed(M2Sp);
       Motor2->run(RELEASE);
     delay(3000);             // stop for 3 seconds


 }
```

10.4 Calibration Code

```
/* Calibration Code ECE 5 Lab 4 */
// Variables and Libaries for Motor

  // Variables for Light Sensors
    int LDR_Pin[7] = {A8,A9,A10,A11,A12,A13,A14};   // Arrays are used top simplify the code
    int LDR[7];                                     // these are variables that have multiple elements to each variable name
                                                    // LDR_Pin hold 7 values and A8 is the 0th element and A11 is the 4th element
  // Calibration Variables
    int led_Pin = 13;      /// This pin is for a led built into the Arduino that indicates what part of the calibration you are on
                             /// You can use any digital pin like digital pin 31 with an LED connected for better visibility
    float Mn[7];
    float Mx[7];
    float LDRf[7] = {0.,0.,0.,0.,0.,0.,0.};

    int   MxRead;
    int   MxIndex;
    float AveRead;
    int   CriteriaForMax;
    float WeightedAve;
    int   ii;
    int   im0,im1,im2;
    float error;

// ********************************************************************************************** //
// setup - runs once
void setup()
{

      Serial.begin(9600);        // For serial communication set up
      pinMode(led_Pin, OUTPUT);  // Note that all analog pins used are INPUTs by default so don't need pinMode

      Calibrate();               // Calibrate black and white sensing

}  // end setup()

// ********************************************************************************************** //
// loop - runs/loops forever
void loop()
{

      ReadPhotoResistors(); // Read photoresistors and map to 0-100 based on calibration

      CalcError();

      Print();          // Print values to serial monitor  //currently commented out but could be good for debugging =)

}  // end loop()

// ********************************************************************************************** //
// function to calibrate
void Calibrate()
 {
  // wait to make sure in position
     for (int calii = 0; calii < 4; calii++)
      {
        digitalWrite(led_Pin, HIGH);   // turn the LED on
        delay(100);                    // wait for 0.1 seconds
        digitalWrite(led_Pin, LOW);    // turn the LED off
        delay(900);                    // wait for 0.9 seconds
      }

   // Calibration
    // White Calibration
    int numMeas = 40;
    for (int calii = 0; calii < numMeas; calii++)
       {
              digitalWrite(led_Pin, HIGH);   // turn the LED on
              delay(100);                    // wait for 0.1 seconds
              digitalWrite(led_Pin, LOW);    // turn the LED off
              delay(100);                    // wait for 0.1 seconds

            for ( int ci = 0; ci < 7; ci++ )
               {
                  LDRf[ci] = LDRf[ci] + (float) analogRead(LDR_Pin[ci]);
                    delay(2);
               }
       }

       for ( int cm = 0; cm < 7; cm++ )
          {
              Mn[cm] = round(LDRf[cm]/(float)numMeas);  // take average
              LDRf[cm]=0.;
          }

    // Time to move from White to Black Surface
    for (int calii = 0; calii < 10; calii++)
      {
        digitalWrite(led_Pin, HIGH);
        delay(100);
        digitalWrite(led_Pin, LOW);
        delay(900);
      }

      // Black Calibration
```

```
      for (int calii = 0; calii < numMeas; calii++)
         {
              digitalWrite(led_Pin, HIGH);
              delay(100);
              digitalWrite(led_Pin, LOW);
              delay(100);

              for ( int ci = 0; ci < 7; ci++ )
                 {
                    LDRf[ci] = LDRf[ci] + (float) analogRead(LDR_Pin[ci]);
                      delay(2);
                 }
         }
       for ( int cm = 0; cm < 7; cm++ )
          {
              Mx[cm] = round(LDRf[cm]/(float)numMeas);  // take average
              LDRf[cm]=0.;
          }

  }  // end Calibrate()

  // ********************************************************************************************** //
// function to read photo resistors, map from 0 to 100, and find darkest photo resitor (MxIndex)
void ReadPhotoResistors()
 {
     for (int Li = 0; Li < 7; Li++)
        {
            LDR[Li] = map(analogRead(LDR_Pin[Li]), Mn[Li], Mx[Li], 0, 100);
              delay(2);
        }
 }   // end ReadPhotoResistors()

 // ********************************************************************************************** //
// Calculate error from photoresistor readings
void CalcError()
 {
        MxRead = -99;
        AveRead = 0.0;
        for (int ii=0;ii<7;ii++)
            {
              if (MxRead < LDR[ii])
                {
                  MxRead = LDR[ii];
                  MxIndex = -1*(ii-3);
                  im1 = (float)ii;
                }
              AveRead = AveRead+(float)LDR[ii]/7.;
            }
      CriteriaForMax = 2;  // max should be at least twice as big as the other values
      if (MxRead > CriteriaForMax*AveRead)
         {
            if (im1!=0 && im1!=6)
              {
               im0 = im1-1;
               im2 = im1+1;
               WeightedAve = ((float)(LDR[im0]*im0 + LDR[im1]*im1 + LDR[im2]*im2))/((float)(LDR[im0]+LDR[im1]+LDR[im2]));
               error = -1*(WeightedAve - 3);
              }
            else if (im1 == 0)
               {
                im2 = im1+1;
                WeightedAve = ((float)(LDR[im1]*im1 + LDR[im2]*im2))/((float)(LDR[im1]+LDR[im2]));
                error = -1*(WeightedAve - 3);
               }
             else if (im1 == 6)
              {
               im0 = im1-1;
               WeightedAve = ((float)(LDR[im0]*im0 + LDR[im1]*im1))/((float)(LDR[im0]+LDR[im1]));
               error = -1*(WeightedAve - 3);
              }
         }
 }  // end CalcError()

 // ********************************************************************************************** //
// function to print values of interest
void Print()
{

        Serial.print(LDR[0]); Serial.print(" ");  // Each photo resistor value is shown
        Serial.print(LDR[1]); Serial.print(" ");
        Serial.print(LDR[2]); Serial.print(" ");
        Serial.print(LDR[3]); Serial.print(" ");
        Serial.print(LDR[4]); Serial.print(" ");
        Serial.print(LDR[5]); Serial.print(" ");
        Serial.print(LDR[6]); Serial.print("    ");

        Serial.print(MxRead); Serial.print(" ");    // the maximum value from the photo resistors is shown again
        Serial.print(MxIndex);Serial.print("    "); // this is the index of that maximum (0 through 6) (aka which element in LDR)
        Serial.println(error);  // this will show the calculated error (-3 through 3)

        delay(100); //just here to slow down the output for easier reading if wanted

}  // end Print()
```

10.5 Line Follower Code

```
/* ********************************************************************************************** */
                      // * ECE 5 Lab 4 Code: Line Following Robot with PID * //
/* ********************************************************************************************** */

//   This is code for your PID controlled line following robot.
//
//
//
//
// Code Table of Contents
    //  1)  Declare Variables     - declares many variables as global variables so each variable can be accessed from every function
    //  2)  Setup (Main)          - runs once at beginning when you press button on arduino or motor drive or when you open serial monitor
    //  3)  Loop  (Main)          - loops forever calling on a series of functions
    //  4)  Calibration           - makes white = 0 and black = 100 (a few seconds to prep, a few seconds on white, a few seconds to move to black, a few seconds of black)
    //  5)  Read Potentiometers   - reads each potentiometer
    //  6)  Run Motors            - runs motors
    //  7)  Read Photoresistors   - reads each photoresistor
    //  8)  Calculate Error       - calculate error from photoresistor readings
    //  9)  PID Turn              - takes the error and implements PID control
    //  10) Print                 - used for debugging but should comment out when not debugging because it slows down program

// ********************************************************************************************** //
// Declare Variables


// Variables and Libaries for Motor
    #include <Wire.h>
    #include <Adafruit_MotorShield.h>

    Adafruit_MotorShield AFMS = Adafruit_MotorShield();
    Adafruit_DCMotor *Motor1 = AFMS.getMotor(1);  // you may switch 1 with 2 if needed (if you see motors responding to error in the opposite way they should be.
    Adafruit_DCMotor *Motor2 = AFMS.getMotor(2);

    int M1Sp = 20, M2Sp = 20;   // this is the nominal speed for the motors when not using potentiometer
    int M1SpeedtoMotor, M2SpeedtoMotor;

  // Variables for Potentiometer
    const int S_pin = A0; //proportional control
    const int P_pin = A1; //proportional control
    const int I_pin = A2; //integral control
    const int D_pin = A3; //derivative control
    int       SpRead = 0; int Sp; //Speed Increase
    int       kPRead = 0; //proportional gain
    int       kIRead = 0; //integral gain
    int       kDRead = 0; //derivative gain

  // Variables for Light Sensors
    int LDR_Pin[7] = {A8,A9,A10,A11,A12,A13,A14};   // Many arrays are used in this code to simplify it
    int LDR[7];                                     // these are variables that have multiple elements to each variable name
                                                    // LDR_Pin hold 7 values and A8 is the 0th element and A11 is the 4th element

  // Calibration Variables
    int   led_Pin = 31;      /// This is a led set up to indicate what part of the calibration you are on.
    float Mn[7];             ///     You could use pin 13 instead which is a built in LED to Arduino
    float Mx[7];
    float LDRf[7] = {0.,0.,0.,0.,0.,0.,0.};


    int   MxRead;
    int   MxIndex;
    float AveRead;
    int   CriteriaForMax;
    float WeightedAve;
    int   ii;
    int   im0,im1,im2;

  // For Motor/Control
    int   Turn, M1P = 0, M2P = 0;
    float error, lasterror = 0, sumerror = 0;
    float kP,kI,kD;



// ********************************************************************************************** //
// setup - runs once
void setup()
{

      Serial.begin(9600);        // For serial communication set up
      AFMS.begin();              // For motor setup
      pinMode(led_Pin, OUTPUT);  // Note that all analog pins used are INPUTs by default so don't need pinMode

      Calibrate();               // Calibrate black and white sensing

      ReadPotentiometers();      // Read potentiometer values (Sp, P, I, & D)

      delay(2000);

      RunMotors();               // Starts motors forward and strait depending on Sp (Speed from potentiometer) and M1Sp/M2Sp (Nominal values)

}  // end setup()

// ********************************************************************************************** //
// loop - runs/loops forever
void loop()
{

      ReadPotentiometers(); // Only if you want to see Potentiometers working in set up as you run the line following

      ReadPhotoResistors(); // Read photoresistors and map to 0-100 based on calibration

      CalcError();

      PID_Turn();         // PID Control and Output to motors to turn

      RunMotors();        // Uses info from

   //   Print();          // Print values to serial monitor  //currently commented out but could be good for debugging =)

}  // end loop()

// ********************************************************************************************** //
// function to calibrate
void Calibrate()
 {
  // wait to make sure in position
     for (int calii = 0; calii < 4; calii++)
      {
        digitalWrite(led_Pin, HIGH);   // turn the LED on
        delay(100);                    // wait for 0.1 seconds
        digitalWrite(led_Pin, LOW);    // turn the LED off
        delay(900);                    // wait for 0.9 seconds
      }

   // Calibration
    // White Calibration
    int numMeas = 40;
    for (int calii = 0; calii < numMeas; calii++)
```

```
      {
            digitalWrite(led_Pin, HIGH);   // turn the LED on
            delay(100);                    // wait for 0.1 seconds
            digitalWrite(led_Pin, LOW);    // turn the LED off
            delay(100);                    // wait for 0.1 seconds

          for ( int ci = 0; ci < 7; ci++ )
             {
                LDRf[ci] = LDRf[ci] + (float) analogRead(LDR_Pin[ci]);
                  delay(2);
             }
      }

      for ( int cm = 0; cm < 7; cm++ )
         {
            Mn[cm] = round(LDRf[cm]/(float)numMeas);  // take average
            LDRf[cm]=0.;
         }

    // Time to move from White to Black Surface
    for (int calii = 0; calii < 10; calii++)
      {
       digitalWrite(led_Pin, HIGH);
       delay(100);
       digitalWrite(led_Pin, LOW);
       delay(900);
      }

      // Black Calibration
      for (int calii = 0; calii < numMeas; calii++)
         {
            digitalWrite(led_Pin, HIGH);
            delay(100);
            digitalWrite(led_Pin, LOW);
            delay(100);

            for ( int ci = 0; ci < 7; ci++ )
               {
                  LDRf[ci] = LDRf[ci] + (float) analogRead(LDR_Pin[ci]);
                    delay(2);
               }
       }
       for ( int cm = 0; cm < 7; cm++ )
          {
            Mx[cm] = round(LDRf[cm]/(float)numMeas);  // take average
            LDRf[cm]=0.;
          }

  }  // end Calibrate()

// ********************************************************************************************** //
// function to read and map values from potentiometers
void ReadPotentiometers()
{
       SpRead = map(analogRead(S_pin),0,1023,0,50); Sp=SpRead;
       kPRead = map(analogRead(P_pin),0,1023,0,10);
       kIRead = map(analogRead(I_pin),0,1023,0,5);
       kDRead = map(analogRead(D_pin),0,1023,0,10);
}    // end ReadPotentiometers()

// ********************************************************************************************** //
// function to start motors using nominal speed + speed addition from potentiometer
void RunMotors()
{
       M1SpeedtoMotor = min(M1Sp+Sp+M1P,255); // limits speed to 255
       M2SpeedtoMotor = min(M2Sp+Sp+M2P,255); // remember M1Sp & M2Sp is defined at beginning of code (default 60)

       Motor1->setSpeed(abs(M1SpeedtoMotor));
       Motor2->setSpeed(abs(M2SpeedtoMotor));

       if (M1SpeedtoMotor > 0)
         {
            Motor1->run(FORWARD);
         }
       else
         {
            Motor1->run(BACKWARD);
         }

       if (M2SpeedtoMotor > 0)
         {
            Motor2->run(FORWARD);
         }
       else
         {
            Motor2->run(BACKWARD);
         }

}  // end RunMotors()

// ********************************************************************************************** //
// function to read photo resistors, map from 0 to 100, and find darkest photo resitor (MxIndex)
void ReadPhotoResistors()
 {
     for (int Li = 0; Li < 7; Li++)
        {
            LDR[Li] = map(analogRead(LDR_Pin[Li]), Mn[Li], Mx[Li], 0, 100);
              delay(2);
        }
 }   // end ReadPhotoResistors()

// ********************************************************************************************** //
// Calculate error from photoresistor readings
void CalcError()
 {
        MxRead = -99;
        AveRead = 0.0;
        for (int ii=0;ii<7;ii++)
            {
              if (MxRead < LDR[ii])
                {
                  MxRead = LDR[ii];
                  MxIndex = -1*(ii-3);
                  im1 = (float)ii;
                }
              AveRead = AveRead+(float)LDR[ii]/7.;
            }
      CriteriaForMax = 2;  // max should be at least twice as big as the other values
      if (MxRead > CriteriaForMax*AveRead)
         {
            if (im1!=0 && im1!=6)
              {
               im0 = im1-1;
               im2 = im1+1;
               WeightedAve = ((float)(LDR[im0]*im0 + LDR[im1]*im1 + LDR[im2]*im2))/((float)(LDR[im0]+LDR[im1]+LDR[im2]));
               error = -1*(WeightedAve - 3);
              }
            else if (im1 == 0)
               {
                im2 = im1+1;
                WeightedAve = ((float)(LDR[im1]*im1 + LDR[im2]*im2))/((float)(LDR[im1]+LDR[im2]));
                error = -1*(WeightedAve - 3);
```

```
              }
             else if (im1 == 6)
               {
                im0 = im1-1;
                WeightedAve = ((float)(LDR[im0]*im0 + LDR[im1]*im1))/((float)(LDR[im0]+LDR[im1]));
                error = -1*(WeightedAve - 3);
               }
          }
 }  // end CalcError()


// ********************************************************************************************** //
// function to make a turn ( a basic P controller)
void PID_Turn()
{
       // *Read values are between 0 and 100, scale to become PID Constants
       kP = (float)kPRead/1.;      // each of these scaling factors can change depending on how influential you want them to be
       kI = (float)kIRead/1000.;    // the potentiometers will also scale them
       kD = (float)kDRead/100.;
  // error holds values from -3 to 3

  Turn = error*kP + sumerror*kI + (error - lasterror)*kD; //PID!!!!!

       sumerror = sumerror + error;
       if (sumerror > 5) {sumerror = 5;} // prevents integrator wind-up
       else if (sumerror < -5) {sumerror = -5;}

       lasterror = error;


       if      (Turn < 0) { M1P = Turn; M2P = -Turn;}   //One motor becomes slower and the other faster
       else if (Turn > 0) {M1P = Turn ; M2P = -Turn;}
       else               {M1P = 0    ; M2P = 0;}
}  // end PID_Turn()


// ********************************************************************************************** //
// function to print values of interest
void Print()
{
        Serial.print(SpRead); Serial.print(" ");  // Initial Speed addition from potentiometer
        Serial.print(kP); Serial.print(" ");      // PID values from potentiometers after scaling
        Serial.print(kI); Serial.print(" ");
        Serial.print(kD); Serial.print("    ");

        Serial.print(LDR[0]); Serial.print(" ");  // Each photo resistor value is shown
        Serial.print(LDR[1]); Serial.print(" ");
        Serial.print(LDR[2]); Serial.print(" ");
        Serial.print(LDR[3]); Serial.print(" ");
        Serial.print(LDR[4]); Serial.print(" ");
        Serial.print(LDR[5]); Serial.print(" ");
        Serial.print(LDR[6]); Serial.print("    ");

        Serial.print(MxRead); Serial.print(" ");    // the maximum value from the photo resistors is shown again
        Serial.print(MxIndex);Serial.print("    "); // this is the index of that maximum (0 through 6) (aka which element in LDR)
        Serial.print(error); Serial.print("    ");  // this will show the calculated error (-3 through 3)

        Serial.print(M1SpeedtoMotor); Serial.print(" "); // This prints the arduino output to each motor so you can see what the values (0-255)
        Serial.println(M2SpeedtoMotor);                  //  that are sent to the motors would be without actually needing to power/run the motors

       // delay(100); //just here to slow down the output for easier reading if wanted
                     // ensure delay is commented when actually running your robot or this will slow down sampling too much
}  // end Print()
```