EEE-201: Arduino 2

In this class, you will learn how to use an Arduino in a more advanced way: you will be building a servo-controlled laser pointer. The three hours will switch between instruction, hands-on and some free time. Example programs will be provided. You can cut and paste from the examples to start writing your own programs.

The class fee includes the servos and the laser diode. Please bring a laptop with the Arduino software and drivers pre-installed as well as your Arduino compatible board. The software may be downloaded online.

If you bring your own laptop, you **MUST** pre-download and install the Arduino software and drivers.

You must make arrangements for a computer before signing up for the class - either be able to bring a laptop, or call the shop for assistance.

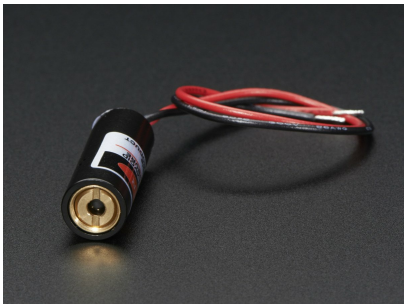
A GitHub repo has been created which houses all of the course material including a course document PDF as well as a ZIP with all the course material including code.

Course Supplies

You will be working with **and given** all the supplies listed below; **please**

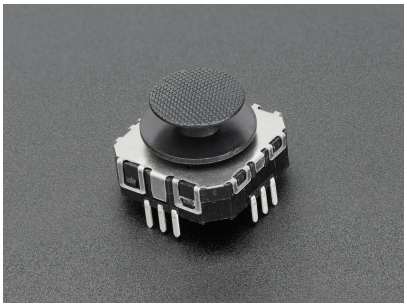
note you must supply your own Arduino

1 x Laser diode

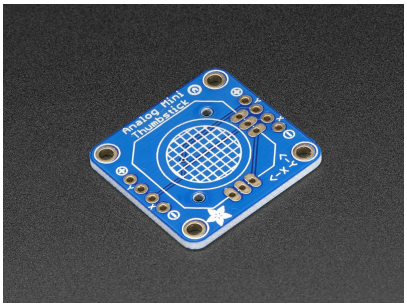


1 x Pan/tilt module

1 x 2-axis joystick



1 x Joystick breakout board



Chapter 1

In the Arduino 1 class, we covered the topic of pulse width modulation (PWM); we will revisit that concept in this chapter and see how it is used to control servos. We'll be covering the following concepts:

- pulse width, voltage and period
- duty cycle
- servo control

Chapter 2

We're now going to look into the basics of controlling a servo with the Arduino as well as how to use a library to make this task a lot easier. We'll be covering the following concepts:

- Servo library
- `for()` loop
- `while()` loop

Chapter 3

In this final chapter we will be using a simple 2-axis joystick to control our servos; we'll be covering the following concepts:

- potentiometers as joysticks
- smoothing arrays

EEE-201 Chapter 1

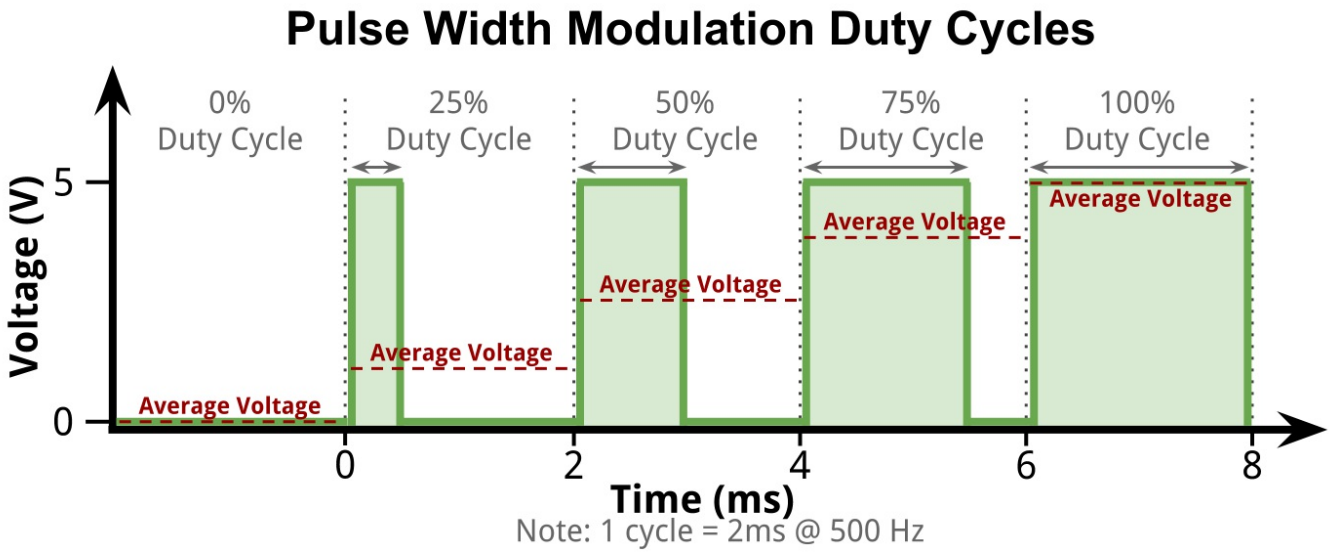
In the Arduino 1 class, we covered the topic of pulse width modulation (PWM); we will revisit that concept in this chapter and see how it is used to control servos. We'll be covering the following concepts:

- pulse width, voltage and period
- duty cycle
- servo control

Table of contents

Part 1 - pulse width modulation

We previously learned that PWM is a method of using a digital output pin (which can only be `LOW` or `HIGH`) and turning it into a quasi-analog output pin which can have almost any value between 0V to 5V; the figure below shows how this is achieved:



A PWM is a signal (square wave) made up of a repetitive pattern pulses or cycles. As the name implies, a square wave as a square pattern with a voltage being either `LOW` (0V) or `HIGH` (5V in this case). The start/end of the pulse period in designated by the voltage changing from 0V to 5V. So, in the example above, we can see that the pulse period is set to 2ms (or 0.002s) - that is, every 2ms the signal repeats itself.

These pulses are often designated in units of frequency which is just the inverse of the pulse period in seconds - in this case, 1/0.002s = 500 Hz.

The amount of time the signal is at `HIGH` (or at 5V in this case) defines the duty cycle or pulse width; for example, if the pulse is `HIGH` for 1ms, the duty cycle equates to 50% (1ms/2ms). At a 50% duty the cycle, the average voltage during the entire cycle will be 50% of the pulse voltage (5V * 50% = 2.5V). In this way, by adjusting the pulse width, we can adjust the voltage on the output pin. We had used this technique in EEE-105 to control the brightness of a LED, but in this class we will use the same technique to control a servo!

Part 2 - controlling a servo

We can use a signal generator to show how to use PWM to control a servo. The majority of hobby servos requires a 50Hz signal (0.02s or 20ms) with a pulse width in the range of 0.5-2.5ms; if you do that math, that equates to a duty cycle range of 2.5-12.5%.

We will use the Rigol DG1022A signal generator to generate a square wave using the `Pulse` functionality. Note that we can't use the `Square` function on the signal generator because the smallest duty cycle it allows is 20%, which is out of the range of the servo - so instead, we use the `Pulse` function.

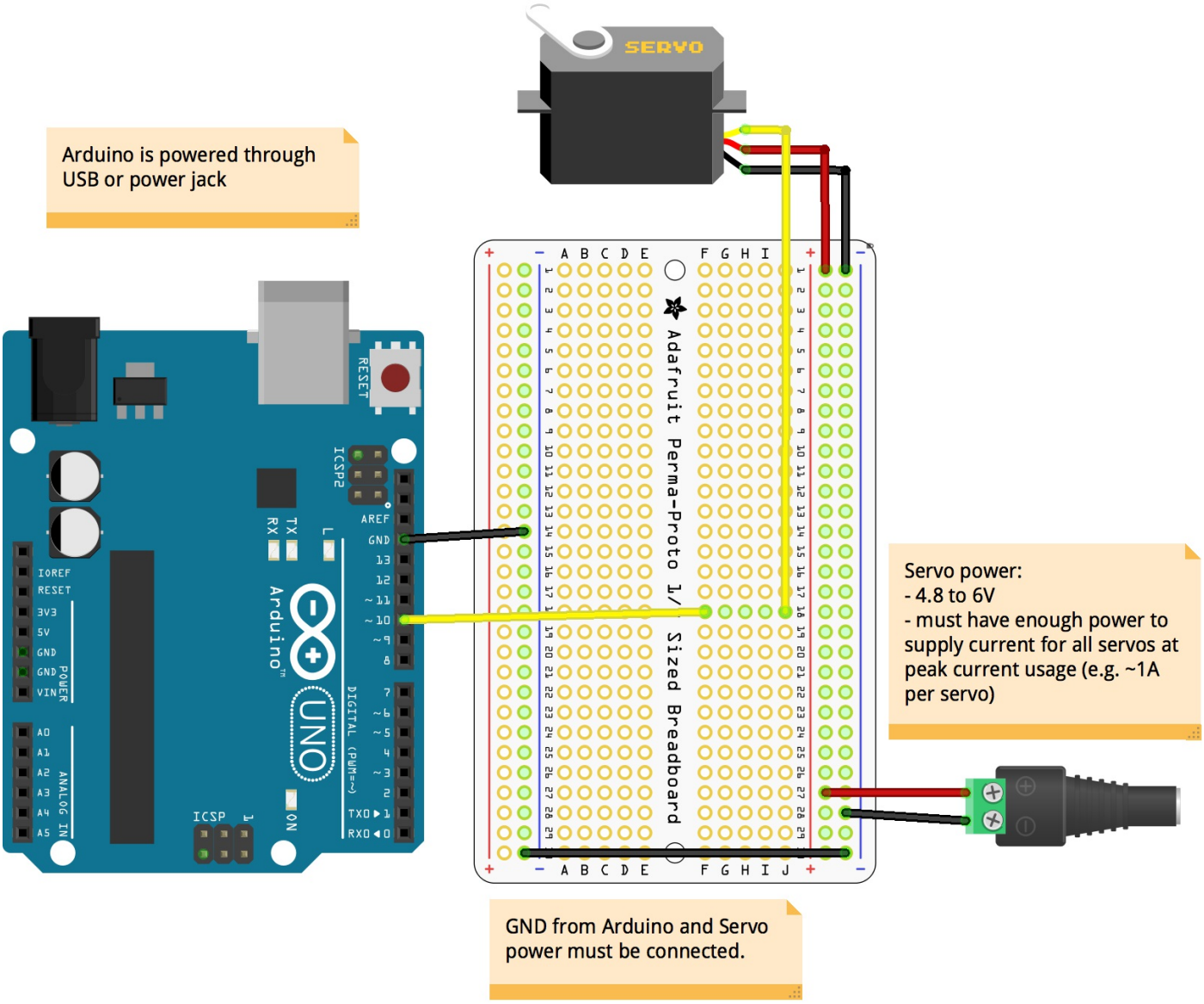
By hooking the servo up to the signal, we can see that by adjusting the duty cycle, the position of the servo also changes. It's important to note here that the pulse width can only be changed at a frequency of 50Hz; if it is changed more rapidly, you can get erratic behavior in your servo.

Part 3 - important note about servo power and wire color

In this class, we will be powering the servos directly from the Arduino. What that means is that the red (+V) wire of the servo is hooked up to the +5V pin of the Arduino and thus the power is being drawn through the voltage regulator of the Arduino. This voltage regulator has a current limit of 0.5A when powered through USB and 1A when powered through the external power jack.

Depending on the type of servo and how it's being used, servos can pull a large amount of current (~1A or so depending on the servo). The way we are using the servos in this class, the current draw is not very large (perhaps 40mA). So, when running two servos, we may experience 40mA * 2 = 80mA which is well within the 0.5A voltage regulator limit.

However if you were going to use a servo in a high load case (such as driving a robot wheel), you risk drawing too much power through the voltage regulator of the Arduino and then burning it up. The way of getting around this is, instead of attaching the red +V wire to the Arduino, attach it to an external voltage supply like this:



For reference, servos come with various wire colors; the table below details the function of each colored wire.

Futaba	JR (*this class*)	HiTec	Function
Red	Red	Red	+V (typically 4-6V)
Black	Brown	Black	GND
White	Orange	Yellow	DATA

EEE-201 Chapter 2

Now that we know what sort of signal a servo requires in order to be

controlled, let's learn how to do that with an Arduino. We'll also be looking into how to make this process a lot easier through the use of the Servo library; and we'll be diving into more programming by learning several new control structures.

Table of contents

Part 1 - square wave on the Arduino

We saw previously that a square if simply a wave that spends part of its time in the `HIGH` (or `ON`) position, and part of its time in the `LOW` (or `OFF`) position; we also saw that this square wave can be used to drive a servo. So, how can we do this with the Arduino? Download the sketch SquareWave.ino and find out!

The sketch is quite basic, with all the magic happening in the `loop()`:

```
void loop() {

  // turn pulse on for pulsewidth duration
  digitalWrite(pin, HIGH);
  delayMicroseconds(pulsewidth);

  // turn pulse off
  digitalWrite(pin, LOW);
  delayMicroseconds(period - pulsewidth);

}
```

All we're doing here is turning the servo pin `HIGH` for a duration equal to the variable `pulsewidth`, after which we turn the pin `LOW` for a duration of the desired `period` minus the `pulsewidth` (this way the total `HIGH`/`LOW` time is equal to `period`).

If we look on the oscilloscope, we can see the clear shape of the square wave, and we can see that the wave is `ON` for the duration of `pulsewidth`. However, the keen observer should have noticed that the period isn't 25ms. The reason for this is a bit technical but part of it has to do with the fact that behind the scenes, various other pieces of code are being run; also, the function `digitalWrite()` is relatively slow. All of this works together to throw the timing that we are hoping for (in this case a square wave of 50 Hz) off. See here for a discussion. One way of getting around this is to use libraries!

Part 2 - Arduino libraries

One way of getting around the messy timing of signals and all the lines needed just to drive the servo are to use Arduino libraries

> The Arduino environment can be extended through the use of libraries, just like most programming platforms. Libraries provide extra functionality for use in sketches, e.g. working with hardware or manipulating data.

You can think of libraries as snippets of code that make doing things like controlling servos or working with other hardware much easier. Let's take a look at the Servo library by loading the Servo.ino sketch.

Let's break the code down.

```
#include <Servo.h> // load the servo library

Servo panServo;  // create servo object to control a servo
```

We begin by including the Servo library into our sketch so that the computer knows it has to upload that Servo library code to the Arduino. Then, we create a new `Servo` object and name it `panServo` - in programming speak we *instantiate* the object. You can think of this as your physical servo; if you had two servo, then you would *instantiate* two `Servo` objects.

We then use the method `attach()` to assign the control pin to our `panServo` through the code `servo.attach(10);` - in this case we are using digital pin 10.

Then, within the loop, we tell the servo to go various positions through the line of code: `servo.write(0);`

One thing you should notice when using libraries is that each library will allow you to use special methods (or functions) as defined within that library. In our case, we have access to methods such as `attach()` and `write()`. If you consult the documentation you can see there are many more functions available to the `Servo` object.

Part 3 - control structures

Now that we can easily control the position of servos, let's learn about a few more control structures to make our coding even more powerful. Load the Sweep.ino sketch and plug in your servo.

The section of code that's most interesting is the the `for` loop and looks

like:

```
for (pos = 0; pos <= 180; pos += 1) { // goes from 0 degrees to 18
  // in steps of 1 degree
  panServo.write(pos);              // tell servo to go to positio
  delay(15);                        // waits 15ms for the servo to
}
```

> A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

The for loop has three parts to it:

```
for ( init; condition; increment ) {
   statement(s);
}
```

1. *init* declares and initializes our loop control variable; in this case `pos` is set to 0.
2. *condition* is evaluated at each iteration of the loop. If `true` the body of the loop is executed; if `false`, the loop is exited.
3. *increment* changes the value of the loop control variable (`pos` in our case) every time the loop iterates. Thus, in our example `pos` is increased by a value of 1.

Through the use of for loops, we can change the position of the servo smoothly and create a repetitive back and forth motion.

Another way of achieving the same motion is through a `while` loop; the `loop()` below achieves the same servo motion as before:

```
void loop() {
    while (pos < 180) {
        pos += 1;
        myservo.write(pos);              // tell servo to go to po
        delay(15);                       // waits 15ms for the ser
      }
      while (pos > 0) {
        pos -= 1;
        myservo.write(pos);              // tell servo to go to po
        delay(15);                       // waits 15ms for the ser
      }
}
```

Like the `for` loop, a `while` loop checks whether a *condition* is true; as long as it is, the loop continues. You can read a great discussion about what the difference is between a for or while loop.

A discussion about loops would not be complete without discussing loop control structures: `break` and `continue`:

`break`:

> Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.

`continue`:

> Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

EEE-201 Chapter 3

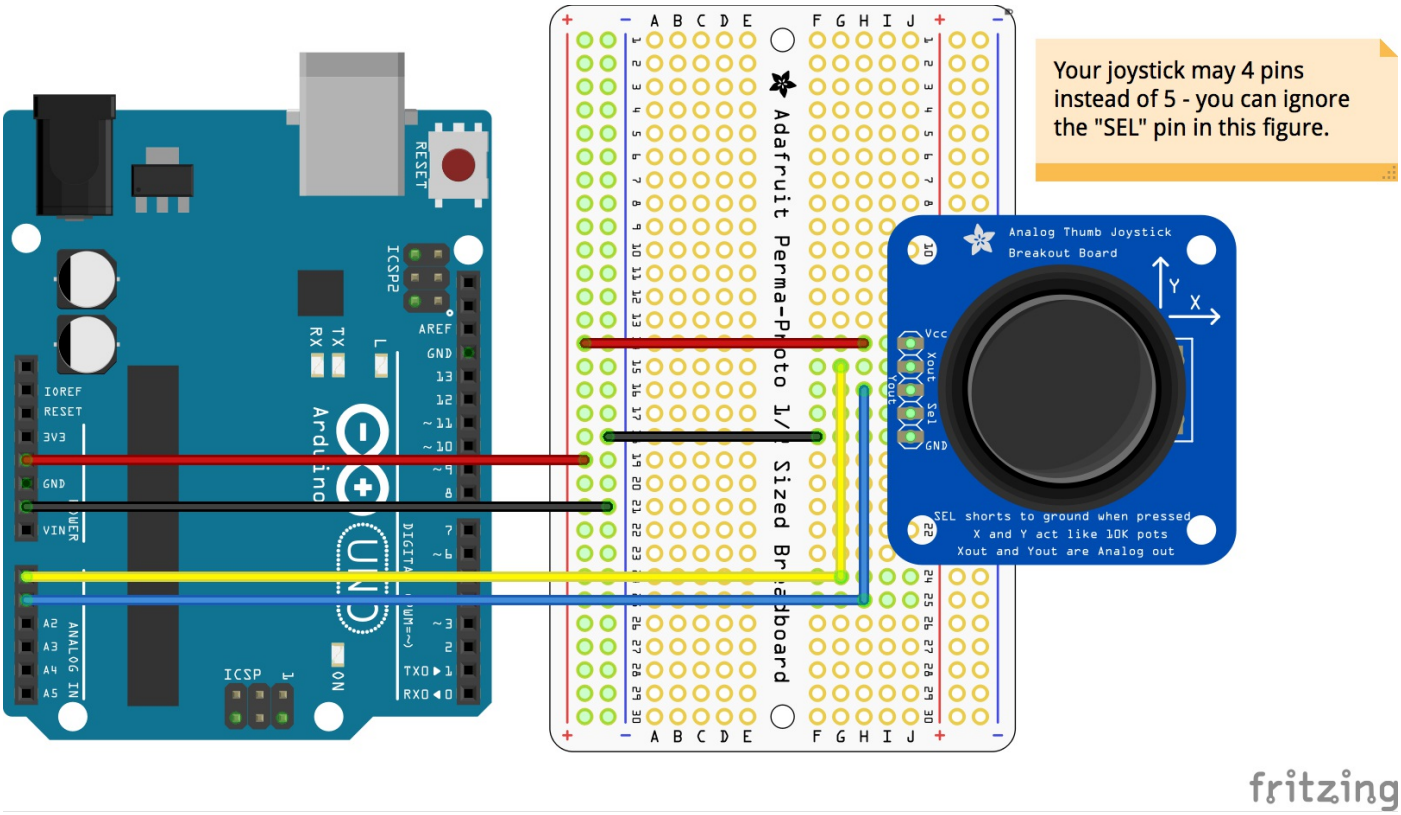
Let's complete our project by learning how to control our servos

manually through the use of a joystick; we'll look at how to use arrays to make our motion more smooth. Finally, we'll be putting on laser diode on the servos and controlling it through various means.

Table of contents

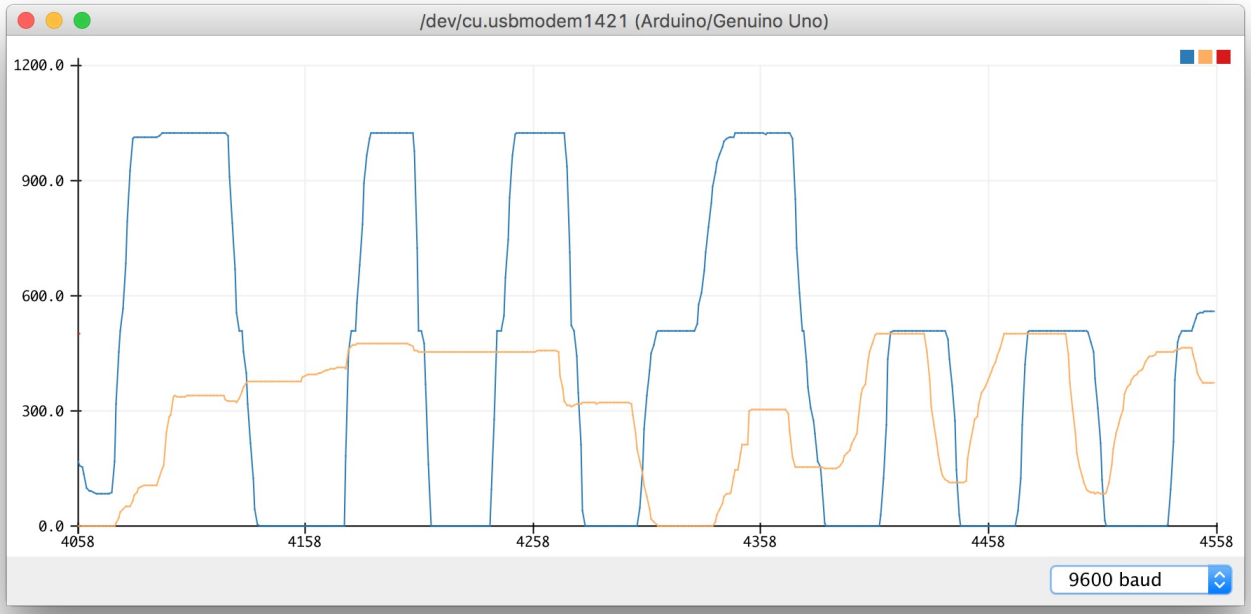
Part 1 - 2-axis joystick

We begin by first getting the joystick working; hook it up as shown in the diagram below:



Load the sketch named joystick.ino onto your Arduino and then open up the serial monitor. You should see the values on it change as you push the joystick around.

You can even open up the serial plotter (Tools > Serial Plotter) to visualize the joystick position



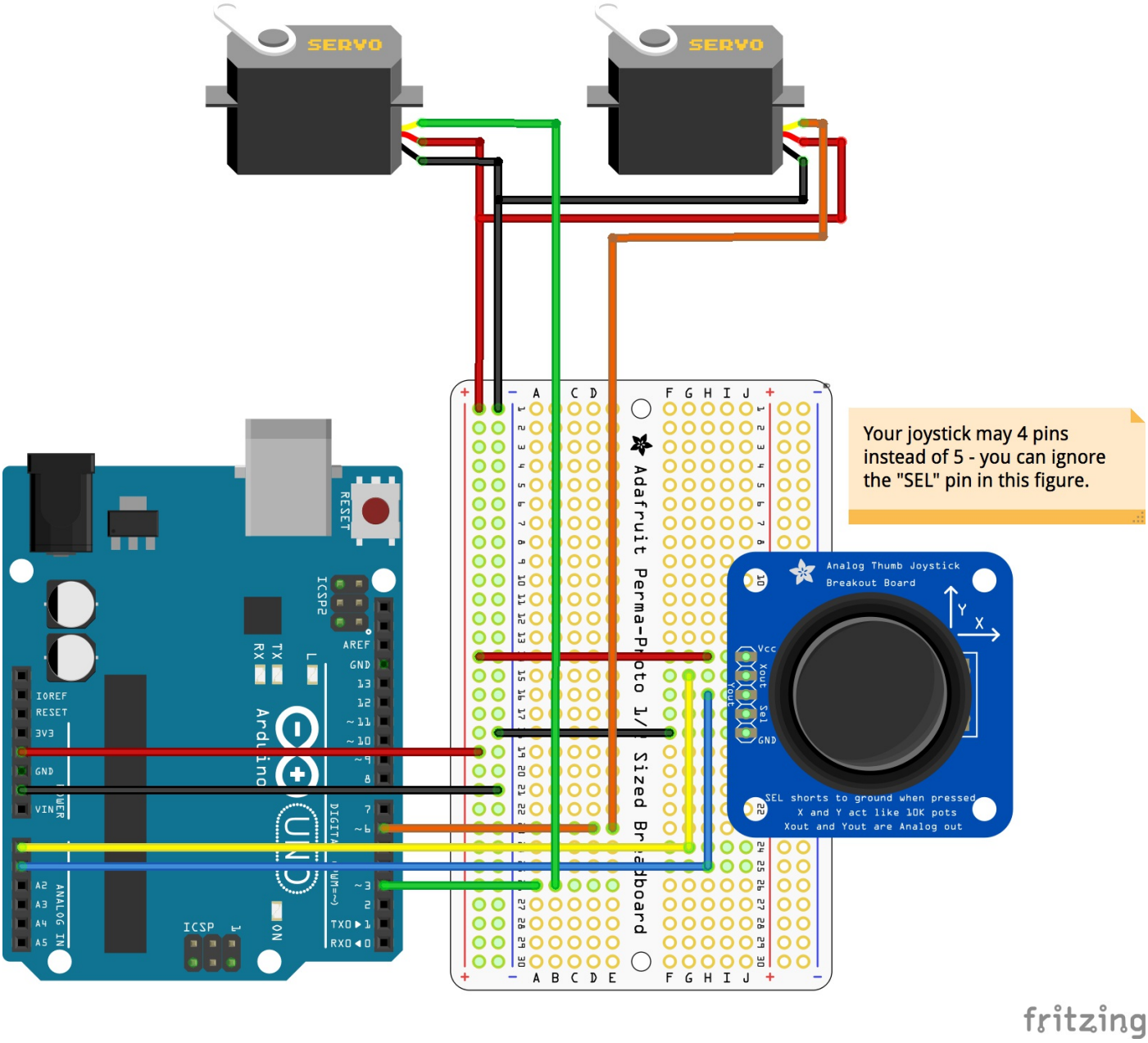
Part 2 - smoothing the joystick

When looking at how your joystick values change, you may have noticed that the values are very sensitive to small changes to the joystick, and that those changes are "sharp" - we can smooth out the joystick values through the use of an array. You can think of an array as a list of values. Each value has a defined location within the array, and can be accessed by this position - a simple analogy is a train with multiple train cars, each train car holds a single number and that number can be acquired by its position within the train.

Upload the sketch smooth.ino and inspect the serial plotter again - you should see that the joystick value (this sketch only logs one of the directions, not both) has been smoothed out.

Part 3 - putting it all together

Now that we know how to use our joystick, let's use it to control our pan/tilt servo unit; we will hook the servos and the joystick together as shown in the figure below:



Upload the sketch joystick_servo.ino and start controlling your servos!