Sensor and Motor Lab

Individual Lab Report

Name: Zhihao Zhu

Team: PhoeniX (H)

Date: 02/14/2019

Team Member: Akshit Gandhi

Shubham Garg

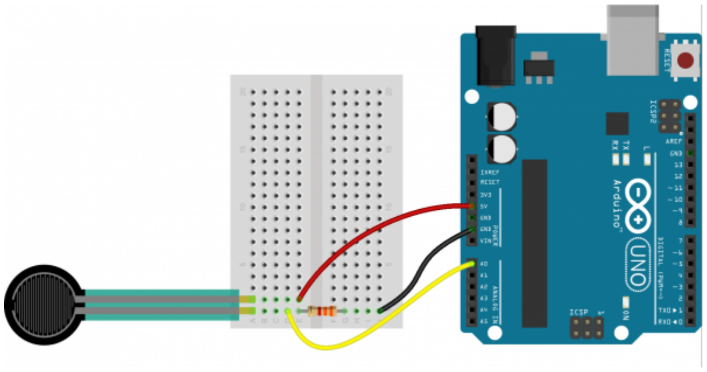
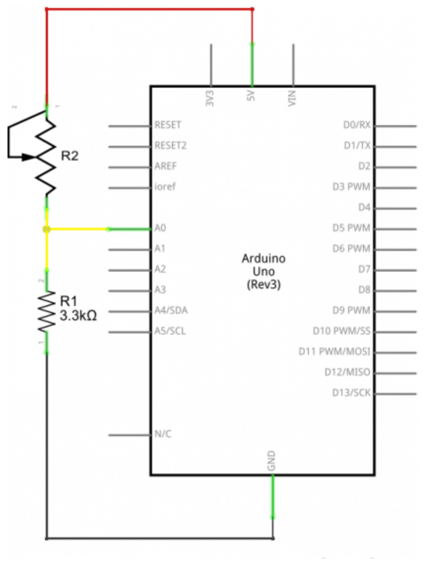
Parv Parkhiya

2. Sensors and Motor Control Lab—Individual Progress

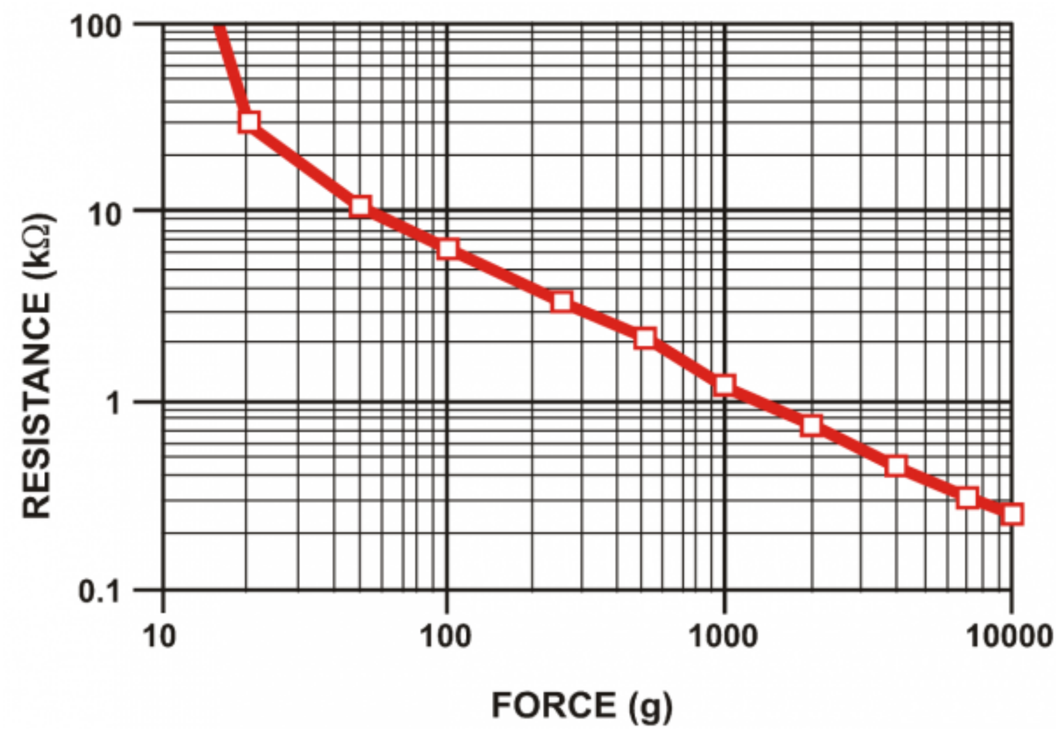
My role in the “sensor and motor control lab” task is to collect the force applied on the Force Sensitive Resistor (FSR), and use that to control how many degrees the stepper motor need to rotate. Larger the force applied on the FSR, more degrees will the stepper motor rotate. So there are two separate steps to establish the system: 1. read the force, 2. control the step motor.

1. Read the force

First, because the signal input into the Arduino is voltage, not the resistance. So we need to first build a circuit to obtain a voltage value which can represent how large the resistance is. In our case, the circuit model we build is:

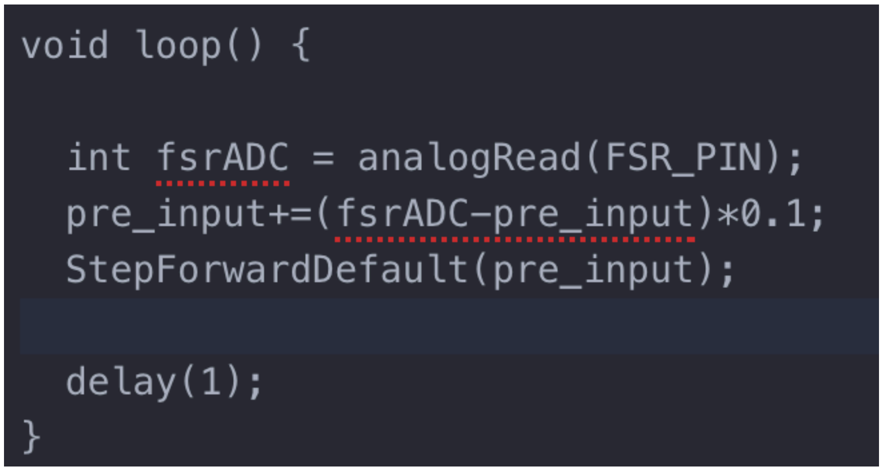



where 5v power supply from Arduino applies on the circuit. To avoid short circuit, we add an additional 3.3kΩ resistor *R1* in the circuit. Then, we measure the voltage over the *R1* with respect to the ground, and the measured value is used as input into Arduino’s input port *A0.* To get the relationship between force applied on FSR and the input of *A0*, we need to first study the dynamic behavior of the FSR:



The relationship is generally linear from 50g and up, but note what the relationship does below 50g, and even more-so below 20g. These sensor's have a turn-on threshold – a force that must be present before the resistance drops to a value below 10kΩ, where the relationship becomes more linear. But in our experiment, we find simplifying the force-voltage relationship does not harm the overall control performance. We directly read the input value from *A0* (as it ranges from about 0 to 400).

and in the code, I implement as this:



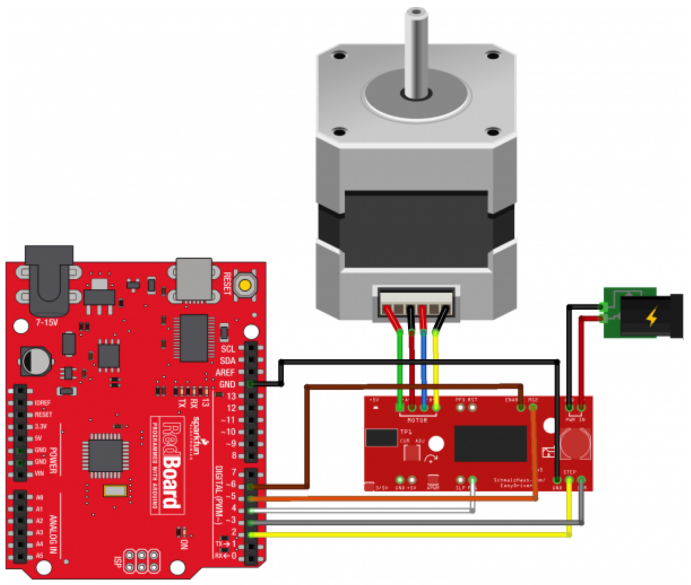
```
void loop() {

  int fsrADC = analogRead(FSR_PIN);
  pre_input+=(fsrADC-pre_input)*0.1;
  StepForwardDefault(pre_input);

  delay(1);
}
```

where fsrADC is the input value read from port A0. For us to achieve more stable input flow, we update only one tenth the difference between previous input and current input. Then, the newly updated value pre_input is fed into the function StepForwardDefault( ) to control the stepper motor rotation.

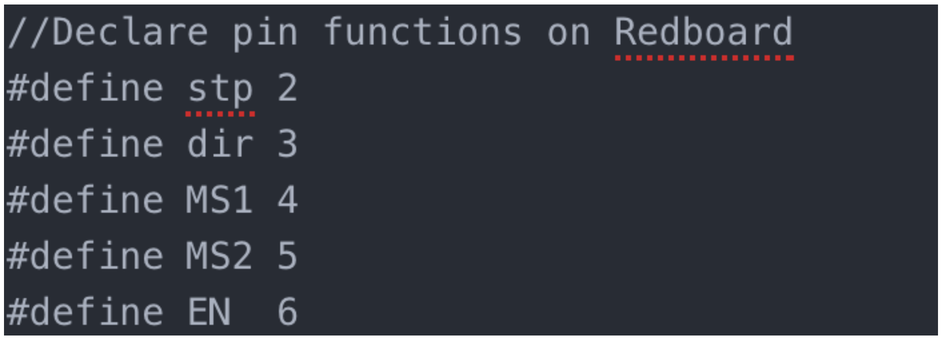
2. Control the step motor

After obtaining the input control signal (the voltage input into analog port *A0*), we can use that signal to control how many degrees the stepper motor should rotate.

First, the stepper motor should be connected to the red board, which is then connected to the Arduino, in order for the Arduino to control the stepper motor. The connection is as follows (from documentation):



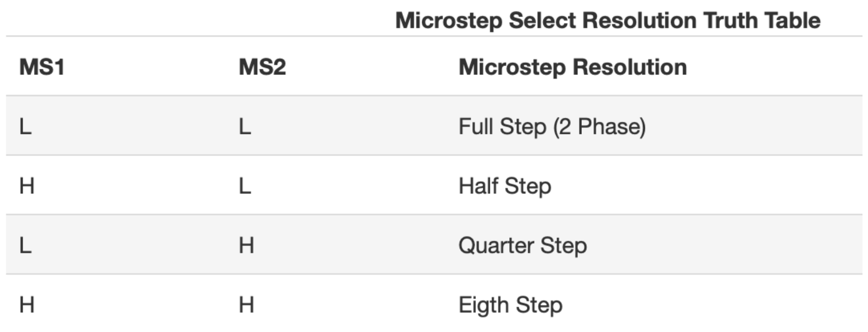
We assign the pin function on Redboard as:



```
//Declare pin functions on Redboard
#define stp 2
#define dir 3
#define MS1 4
#define MS2 5
#define EN  6
```

Explanation for the function of each pin:

stp: if set to *HIGH*, run one step
dir: if set to *HIGH*, run in “reverse” direction
En: if set to *LOW*, allow the control of the motor.
MS1 and *MS2* are together used to control the degrees each step will go:

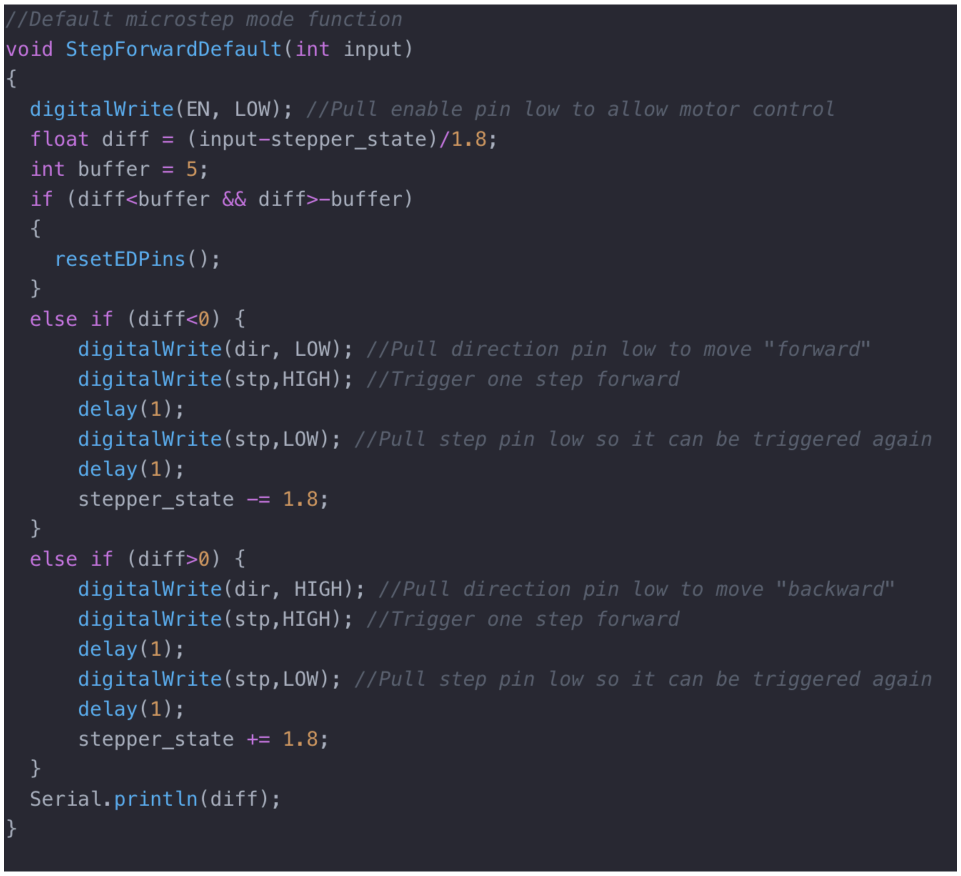


Microstep Select Resolution Truth Table

MS1	**MS2**	**Microstep Resolution**
L	L	Full Step (2 Phase)
H	L	Half Step
L	H	Quarter Step
H	H	Eigth Step

For the motor provided, 1.8 *deg* is the default full step size. which means for each full step, the motor will rotate 1.8 *deg.*

Finally the motor control function can be written as:



```
//Default microstep mode function
void StepForwardDefault(int input)
{
  digitalWrite(EN, LOW); //Pull enable pin low to allow motor control
  float diff = (input-stepper_state)/1.8;
  int buffer = 5;
  if (diff<buffer && diff>-buffer)
  {
    resetEDPins();
  }
  else if (diff<0) {
      digitalWrite(dir, LOW); //Pull direction pin low to move "forward"
      digitalWrite(stp,HIGH); //Trigger one step forward
      delay(1);
      digitalWrite(stp,LOW); //Pull step pin low so it can be triggered again
      delay(1);
      stepper_state -= 1.8;
  }
  else if (diff>0) {
      digitalWrite(dir, HIGH); //Pull direction pin low to move "backward"
      digitalWrite(stp,HIGH); //Trigger one step forward
      delay(1);
      digitalWrite(stp,LOW); //Pull step pin low so it can be triggered again
      delay(1);
      stepper_state += 1.8;
  }
  Serial.println(diff);
}
```

First, we set a variable “buffer”, which is used to avoid small input disturbance. It means when the difference between new input and previous input is large enough, we deem it as a viable input. Otherwise, we reset the stepper motor to the default resting state. Then, if the difference between the current position and desired position is positive, we will make stepper motor rotate forward one step (which is 1.8 deg). Else, we will make the stepper motor rotate backward one step. This will repeat until the final difference between the current position and desired position is zero.

3. MRSD project progress

After submitting the CoDR, we worked on building our drone’s hardware/ software system from scratch. We assembled drone’s components, as well as the sensors (e.g. infrared camera, stereo camera..). Then, we also configured the software system (the installation of Ubuntu on the Nvidia TX2 onboard computer, and also the necessary packages including CUDA, OpenCV, etc) We have successfully make the drone to fly and hold stable in certain attitude. Our next step is to make the drone fly smoothly in the horizontal direction. In addition, we will make the drone and Husky operate autonomously (follow certain path to reach the target position) without human control.

1. Reading a data sheet. Refer to the ADXL335 accelerometer data sheet to answer the below questions.

1.1 What is the sensor's range?

The sensor has a minimum full-scale range of ±3 g, and a typical range of ±3.6 g

1.2 What is the sensor's dynamic range?

The dynamic range of the sensor is 7.2 g.

1.3 What is the purpose of the capacitor CDC on the LHS of the functional block diagram on p. 1? How does it achieve this?

The CDC is used to decouple the accelerometer from noise on the power supply. The capacitor acts like a high-frequency-pass filter, and filters the high-frequency noise. Because when the signal's frequency is high, the capacitor's resistance becomes small, and alternating current will pass through CDC.

1.4 Write an equation for the sensor's transfer function.

$$V_out = 1.5V + 0.3V * \frac{acceleration}{g}$$

1.5 What is the largest expected nonlinearity error in g?

Error_nl = 6 * 3% = 0.18

1.6 How much noise do you expect in the X- and Y-axis sensor signals when the sensor is excited at 25 Hz?

Noise = 150 * 5 ug = 750 ug

1.7 How about at 0 Hz? If you can't get this from the data sheet, how would you determine it experimentally?

Apply a signal with 0 Hz, then measure on the output to determine the noise.

2. Signal Conditioning

2.1 Filtering

2.1.1 What problem(s) might you have in applying a moving average?

It may remove important signals with certain frequency. Also, it has time lag.

2.1.2 What problem(s) might you have in applying a median filter?

Applying the median filter has a high computational cost.

2.2 Opamps

According to the circuit, we can write the equation as following:

$$(\frac{R_f}{R_i} + 1)V_2 - \frac{R_f}{R_i}V_1 = V_{out}$$

case 1: Your uncalibrated sensor has a range of -1.5 to 1.0V (-1.5V should give a 0V output and 1.0V should give a 5V output).

we use V_i as reference voltage, so we have:

$$-1.5(\frac{R_f}{R_i} + 1) = 0 + \frac{R_f}{R_i}V_1$$

$$(\frac{R_f}{R_i} + 1) = 5 + \frac{R_f}{R_i}V_1$$

By solving the above equations, we have:

$$V_1 = -3$$

$$\frac{R_f}{R_i} = 1$$

case 2: Your uncalibrated sensor has a range of -2.5 to 2.5V (-1.5V should give a 0V output and 1.0V should give a 5V output).

The calibration can't be done. Because no matter we use V_1 or V_2 as reference voltage, we can't solve the above equation with appropriate choice of $\frac{R_f}{R_i}$ (the $\frac{R_f}{R_i}$ we got is either 0 or negative).

4. Control

4.1 If you want to control a DC motor to go to a desired position, describe how to form a digital input for each of the PID (Proportional, Integral, Derivative) terms.

(1) Proportional: difference between current position and desired position

(2) Integral: Sum of accumulated error from previous steps.

(3) Derivative: rate of the error changes.

4.2 If the system you want to control is sluggish, which PID term(s) will you use and why?

I will just proportional term. Because the proportional control regulates error more quickly. Using integral term will even prolong the system to converge. Derivative has little influence on a sluggish system as the derivative of that system is small.

4.3 After applying the control in the previous question, if the system still has significant steady-state error, which PID term(s) will you use and why?

I will use the integral term, because just using the proportional term can't eliminate the steady-state error. By applying the integral control, the accumulated error can be diminished

4.4 After applying the control in the previous question, if the system still has overshoot, which PID term(s) will you apply and why?

I will apply derivative control. Because by adding derivative control, system will decrease the control input when it saw the error rate change is fast.