Team F

Individual Lab Report 1

Sensors and Motors Lab

Yuchi Wang

Teammates

Danendra Singh

Pulkit Goyal

Rahul Ramkrishnan

Pratibha Tripathi

October 14, 2017

1 Individual Progress

For my contribution the sensors and motor lab, I designed the circuitry for the force sensitive resistor (FSR) sensor, implemented the PID controller for the DC servo motor with Rahul, and pulled in everybody's part for the full system integration into a single Arduino program.

1.1 Force Sensitive Resistor

The FSR that was used in our lab was a small FSR 400 sensor by Interlink Electronics that can be bought from creatron, shown in figure 1. The FSR on the circuit board is shown in figure 2. The FSR works by altering its resistive value whenever a force is applied to the center pad. The data sheet for the FSR states that the resistance value can range from 20 kΩ for 20 g of force to 0.2 kΩ for 10000 g of force (Fig 3). This can be converted to a voltage by using a voltage divider which can then be read by an analog pin on an Arduino chip.

Figure 1: FSR 400 Sensor [1]

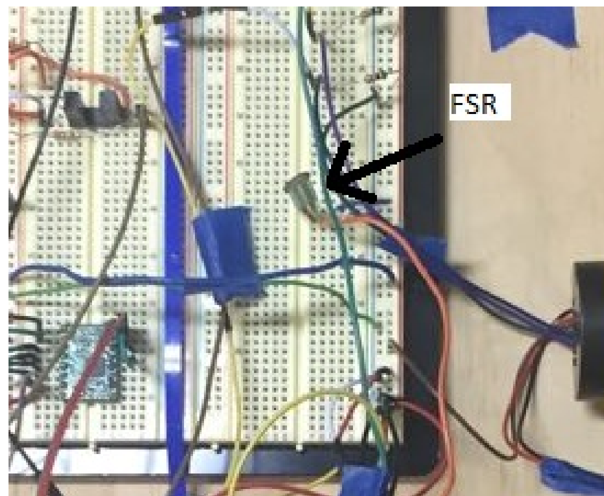

Figure 2: FSR on circuit board

[1] Source: https://www.sparkfun.com/products/9673

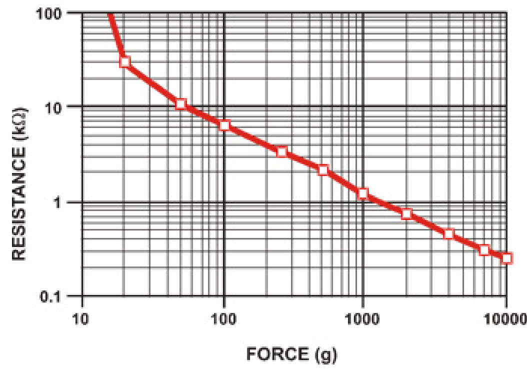

Figure 3: FSR 400 response [2]

The voltage divider was not too difficult to implement. The circuit consisted a ground, a vcc (5v), and a 10 kΩ resistor (Fig 4). Of course, the response shown in figure 3 is non-linear and the voltage given by the voltage divider is inversely proportional. As a result, the relation between the amount force applied to the FSR and the voltage output from the voltage divider has a non-linear relationship. To gain an idea of the approximate transfer function of the entire system, I recorded the force that I applied and the value received on the Arduino pin (Table 1).

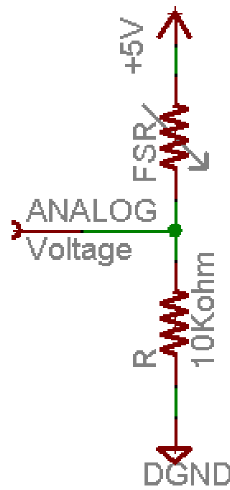

Figure 4: FSR voltage divider

Force (g)	Analog IN
0	1
20	256
50	512
100	642
250	764
400	857
1000	912

Table 1: Force vs Arduino reading

[2]Source: FSR 400 Data Sheet

It is noted that in Table 1, the experiment was conducted with weights, large objects, and a scale. When I tried applying the maximum amount of force with my hand, I could not achieve a corresponding voltage reading of more than 900. Thus, in the actual software implementation, the reading is scaled from 0-900 to 0-255. Although the relationship shown in Table 1 is non linear, I found that my sense of force intensity was also non-linear. In fact, I found it easier to perceive the difference between 100 and 200g of force as opposed to 800 and 900g. As a result, the mapping in software is a linear mapping to preserve the non-linearity of the sensor. The code for reading the FSR sensor and actuating the DC motor is shown below.

```
void loop_FSR(){

 if (modeState == 1){
 // Forward motion
    digitalWrite(dir1, LOW);
    digitalWrite(dir2, HIGH);
 }
 else{
 // Backward motion
    digitalWrite(dir1, HIGH);
    digitalWrite(dir2, LOW);
 }

  sensorValue = analogRead(A0);
  if (auto_read == 1)
    analogWrite(pwm, sensorValue / 900 * 255);
  else
    analogWrite(pwm, sensorValue_read / 900 * 255);

}
```

1.2 PID Control

The work completed in designing the PID controller for the DC motor's position and speed was shared between Rahul and me (note: I did not work on reading and pulling information from the encoder). The PID controller was developed from scratch and did not use an existing controller libraries. Because no digital system is able to model a continuous PID controller, the controller is based off the discrete equations for PID controls. The derivative is approximated by the difference in the current and the previous time step error and the integral is approximated by the sum of the error. The code implementation is shown below:

```
//count is a volatile variable incremented in an interrupt attached to the DC encoder

void PIDcalculation_POS()
{
  //Convert count to angle
  angle = 0.9 * count;

  //proportional, integral, and derivative terms
  error = setpoint - angle;
  changeError = error - last_error;
  totalError += error;

  //calculating and constraining total term
  pidTerm = (Kp * error) + (Ki * totalError) + (Kd * changeError);
  pidTerm = constrain(pidTerm, -255, 255);
  pidTerm_scaled = abs(pidTerm);
  last_error = error;
```

```
}


void PIDcalculation_SPD()
{
  //record the count 50 ms apart
  prev_angle = 0.9 * count;
  delay(50);
  angle = (1 * count);

  curr_spd = (angle - prev_angle)/ 50;
  error = abs(tar_spd) - abs(curr_spd);

  changeError = error - last_error;
  totalError += error;
  pidTerm = (Kps * error) + (Kis * totalError) + (Kds * changeError);
  pidTerm = constrain(pidTerm, -255, 255);
  pidTerm_scaled = abs(pidTerm);
  last_error = error;

}
```

Tuning for the PID controllers was done by first setting all parameters to 0 and then tuning each parameter sequentially. The K_p term was selected such that the motor reached steady state in a reasonable time, K_i was selected such that the steady state error was negligible, and K_d was selected to reduce oscillations and overshoot. For position, the final parameters chosen are $K_p = 5$, $K_i = 0$, $K_d = 2$ and we achieved a maximum overshoot of 1°. For velocity, the final parameters chosen are $K_p = 2$, $K_i = 0.02$, $K_d = 0.1$ and we achieved a maximum overshoot of 5% of our target velocity.

1.3 Arduino software integration

The software integration task involved everyone but I was the main member in charge of integrating the individual Arduino code into one program. In an effort to reduce integration bugs and for the sake of simplicity, the modularity of each member's code was preserved. Each member's code was copied "as is" into the program under a different function name. The main loop in the program consisted of an if-statement that called the corresponding function according to the mode of the system. The limitation of this approach is that functionality of each member's program remains independently but that is acceptable as we have delegated our tasks so that our codebase can be independent. The main loop code is provided in the following section.

1.4 Arduino GUI communication

The communication between the Arduino and the GUI was completed by Pulkit and me. Because the serial communication between the Arduino and the GUI is a stream, the build-in receiver and the sender functions do not maintain message boundaries. This means that the functionality of determining message beginning and ends has to be implemented by us. This requirement motivated our implementation of a CSV message format.

For the message from the Arduino to the GUI, the message starts with a "S" followed by 4 numbers, each separated by a comma, and ends with a "E". For the message from the GUI to the Arduino, the message starts "M" followed by 5 numbers, each separate by a comma, and ends with "R". On the Arduino side, to parse the message, sscanf is used. This works nicely as the messages all have a set format and the risk of losing messages if multiple commands are received at once is not present as the poll rate is much higher than the send rate from the GUI.

When receiving commands from the GUI, the first number indicates whether the motors will be GUI or user controlled, the second number indicates the sensor that will be used, the third number indicates the motor to be controlled, the fourth number specifies the speed or degree and the fifth number controls the direction of rotation. Nested if statements are used to process the command. Not all combinations are valid. The code for the main loop function is shown below.

```
void loop(){
  char str[255];

  //Debouncing code
  if (flag == true){
    if (millis() - currtime > 200){
      switch (modeState) {
        case 1:
          modeState = 2;
          break;
        case 2:
          modeState = 1;
          break;
        default:
          break;
      }
      flag = false;
    }
  }

  \\Parse incoming command
  if (Serial.available()){
    String stringIn = Serial.readString();
    stringIn.toCharArray(str, 255);

    if (int num = sscanf(str, "M%d,%d,%d,%d,%dR", &auto_read, &sensor_read, &motor_read,
         &val_read_1, &val_read_2) == 5){
       if (auto_read == 1) {
          if (sensor_read == 1){
            mode = "STEPPER";
          }
          else if(sensor_read == 2){
            mode = "SERVO";
          }
          else if(sensor_read == 3){
            mode = "FSR";
          }
          else if (sensor_read == 4){
            mode = "IR";
          }
       }
         else if (auto_read == 2){
          if (motor_read == 1){
            mode = "SERVO";
            val_read = val_read_1;
          }
          else if (motor_read == 2){
            mode = "PIDPOS";
            setpoint = val_read_1;
          }
          else if (motor_read == 3){
            mode = "PIDSPD";
            tar_spd = val_read_2 == 1? -val_read_1: val_read_1;
```

```
        }
        else if (motor_read == 4){
          mode = "STEPPERANG";
          setpoint = val_read_1;
          tar_dir = (val_read_2)-1;
          a = 0;
        }
      }
    }
  }

  \\Go into corresponding sub-function
  if (mode == "PIDPOS"){
    loop_PID_POS();
  }
  else if (mode == "PIDSPD"){
    loop_PID_SPD();
  }
    else if (mode == "FSR"){
    loop_FSR();
  }
    else if (mode == "SERVO"){
    loop_servo();
  }
    else if (mode == "IR"){
    loop_ir();
  }
    else if (mode == "STEPPERANG"){
    loop_stepper();
  }
    else if (mode == "STEPPER"){
    loop_step();
  }

  \\Update GUI with sensor values
  String strstr = String(" S" + String(vals) + "," + String(val) + "," + String(sensorValue) +
      "," + String(distance) + "E ");
  Serial.print(strstr);
  Serial.flush();
}
```

2 Challenges

The main challenges in the motor and sensors lab arose from the integration step; the force sensitive resistor was very simple and only required 3 lines of code. When integrating code from all the members, although each code is meant to be run independently, certain variables and function names have to be renamed to avoid conflicts from duplicates. Keeping track of what each function and variable refers to with the new names is a challenge. To address this problem, I tried to fully test and verify each subsystem before incorporating another person's code to the main program. This worked relatively well except that sometimes there was backward compatibility issues when variable values became overwritten.

Another challenge was sending and receiving the code from the GUI. The main problem here was the design of a message system that fulfilled the requirements needed by both systems. Pulkit and I had different requirements for our systems so deciding upon a format that met our needs while not necessitating too much work for each other took some time. In the end, we decided to not send all the data at once and instead use generic fields that could represent different values depending on the value of

other fields. Debugging the communication system also represented a major challenge and headache for us. The problem was that when running both the GUI and Arduino on one system, we were unable to see the serial feed from the Arduino side. This is a problem because most of our debugging skills relied upon printing to serial. Eventually, we worked around this problem by echoing the Arduino-side error messages to the GUI which printed it to the GUI serial.

3 Teamwork

- Yuchi: Responsible for FSR sensor circuitry and code, PID controller, Arduino-side software integration
- Pulkit: GUI
- Pratibha: Circuitry and code implementation for servo motor with IR sensor
- Danendra: Circuitry and code implementation for stepper motor and slot sensor
- Rahul: Circuitry and code implementation for DC motor, PID controller

Our entire team worked predominantly in the MRSD lab. This allowed us to have access to a wide range of electronic and diagnostic tools that definitely helped when we tried to get the sensors and motors running, especially for Danendra, Rahul and Pratibha. As well, it allowed us to communicate with each other which was essential during the integration stage.

We also fully divided our tasks from the very beginning and decided to only integrate at the very end as to maximize individual efficiency. Clearly, certain parts of the system had reliance on other parts so for those sections, we tried to decide upon a certain design before splitting off (such as the communication protocol that I discussed with Pulkit). Overall, it worked well as most of the difficulties arose during the integration stage. As previously said, we tried to address these problems by integrating each subsystem one by one.

4 Future plans

Over the past few weeks, we have spent lots of time talking with George Cantor, John Dolan and upper year MRSD students who have worked with drones for their advice on how to appropriately scope the project as we have a lot of flexibility in terms of choosing the application and the details of our project. We have talked with Katia and now have a better idea of how to proceed in the following weeks. Currently, we have obtained a Clearpath Husky robot from George Cantor and a Parrot Bebop 2 drone from Katia Sycara. We have flown the Bebop 2 drone to get a sense of its auto-stability capabilities. At the moment, we are trying to get the Husky to move in simple motions (ie forward/backward). The Husky is not responding to our commands so we believe that we need to wipe the on-board computer with a factory image. Our plans for the immediate future are twofold: first, we plan to install a new image, install the necessary drivers and ROS packages, and be able to teleoperate the Husky Robot. When we complete that, we will start trying different sensors on the Husky to see their performance - if needed, we will purchase a Velodyne puck. Second, we will get the Bebop 2 SDK running on a computer and attempt to lift off, land, and perform simple maneuvers with the drone. When that is possible, we will look for a way to control the fish-eye lens on the drone and capture images with a laptop.

5 Quiz

1. (a) 6g min, 7.2g typical

   (b) $20\log(\frac{0.707\cdot 3.6}{150\cdot 10^{-6}}) = 194dB$

(c) The capacitor is used for decoupling purposes. When placed in parallel with the sensor, it acts as a low pass filter and thus removes all high frequency noise.

(d) $y = 1.5 + 0.3x$, where y is the output voltage and x is the input in g.

(e) $\pm 0.3\%$ of full scale

(f) 750 μg

(g) We find the 0Hz noise by measuring the output voltage at 0Hz, finding the deviation from the nominal value and converting it into a RMS value.

2. (a) The response time may be slow if the window size is large, particularly if the frequency that needs to be filtered out is high. In addition, the moving average filter does not distinguish between different frequencies, making it difficult to be used with frequency encoded signals.

   (b) One problem is the high computational cost of the filter. The best sorting algorithms operate in O(nlogn) and thus large window sizes may not be suitable for real-time applications

   (c) For uncalibrated range -1.5 to 1V: $V_2 = V_{in}$, $V_1 = V_{ref} = -3$, $R_i = R_f = 1$. Solved by using a system of equations and Wolfram Alpha.

   For uncalibrated range -2.5 to 2.5: Not possible. Putting in a system of equations with 2 equations and 3 variables returned no solution from Wolfram Alpha. Both configurations for V_{ref} had no solution.

3. (a) The digital input to the proportional term can be the difference between the current location and the desired location. For the integral term, a digital system can simply sum up the previous values. The derivative term is harder to implement as a pure derivative cannot be implemented on a real digital system. Instead, we can use the difference between the past two error values as an approximation for the derivative.

   (b) I would use the P term as a large gain is required to counteract a slow system.

   (c) If there is significant steady-state error, then a P and I term is required. The I term is especially helpful as it increases in magnitude as long as there is sustained error. Thus, with the I term, the system should eventually reach 0 error.

   (d) If there is overshoot, I would apply the derivative term because it counteracts the rapid changing of the error function. The derivative component is designed to bring the error rate to 0 and thus it enforces damping and prevents overshooting and oscillations.