Individual Lab Report 01

Autonomous Reaming for Total Hip Replacement

HIPSTER | ARTHuR

Anthony Kyu

Team C:

Kaushik Balasundar | Parker Hill | Anthony Kyu
Sundaram Seivur | Gunjan Sethi

Sponsor:
Redacted by Sponsor's Request

February 10th 2022

Carnegie Mellon University
The Robotics Institute

Contents

1 Individual Progress

1.1 Sensors and Motors Lab

My responsibilities for the Sensors and Motors Lab included measuring and determining a transfer function for the force resistive sensor (FSR), designing the circuit for the FSR sensor, connecting and controlling the servo from the Arduino, designing and coding the state machine to swap between various sensors and motors, and coordinating with Gunjan to communicate with the GUI. I also helped with system integration along with the rest of my team.

1.1.1 Force Sensistive Resistor (FSR)

For setting up the Force Sensitive Resistor, the first step I took was to determine the transfer function of the sensor. To do this, resistance across the FSR was measured using a Digital Multimeter while various weights were applied to the FSR. The weights used were measured with a scale. These measurements were taken several times (Figure 1).

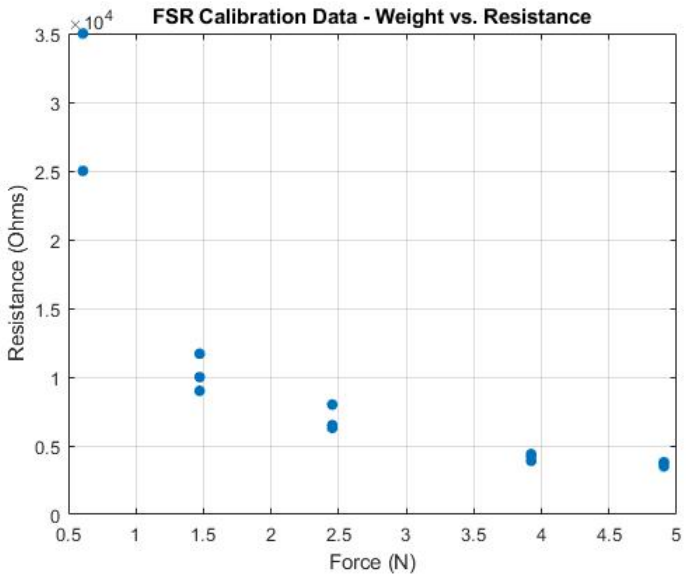

Figure 1: Resistance Data Collected for Force Sensitive Resistor

Using this data, a voltage divider circuit was designed, and the fixed resistor was optimized to maximize the difference in output voltage when a weight of 50 grams was used versus when a weight of 500 grams was used. This relationship is defined by the equation

$$R_{fixed} = \sqrt{R_{50} * R_{500}}$$

where R_{fixed} is the resistance of the fixed resistor in the voltage divider and R_{50} and R_{500} are the measured resistance of the FSR at 50 and 500 grams.

The output of the voltage divider was then put into an op-amp to mitigate signal loss, and then filtered using a simple RC circuit with a cutoff of about 50 Hz to reduce typical 60 Hz noise. Decoupling capacitors were also introduced into the circuit. Figure 2 shows the circuit diagram used.

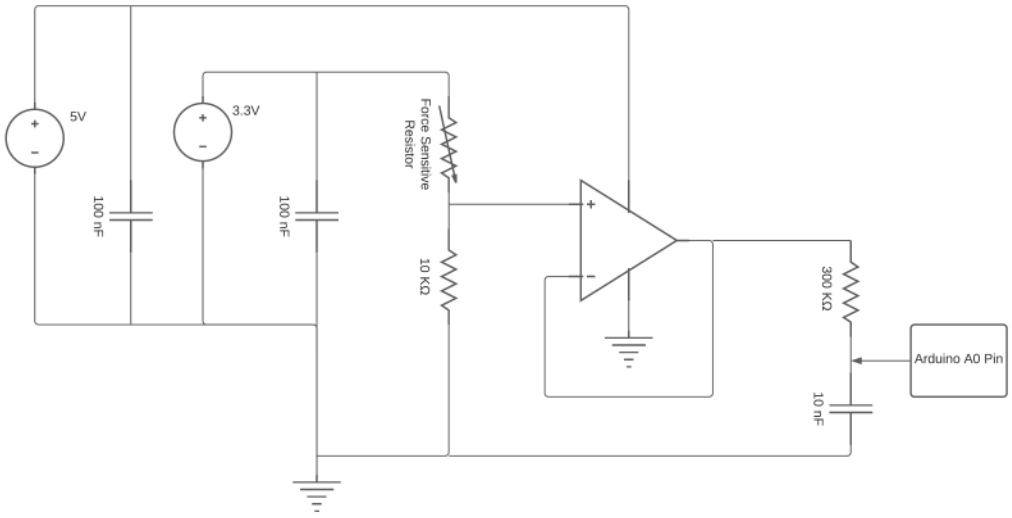

Figure 2: Force Sensitive Resistor Circuit Diagram

The resistance data was converted to output voltage and then fitted with several polynomial functions. The function that fitted the data the best was a 2nd-Order Polynomial, and this result (Figure 3) agreed with the plots in the FSR datasheet.

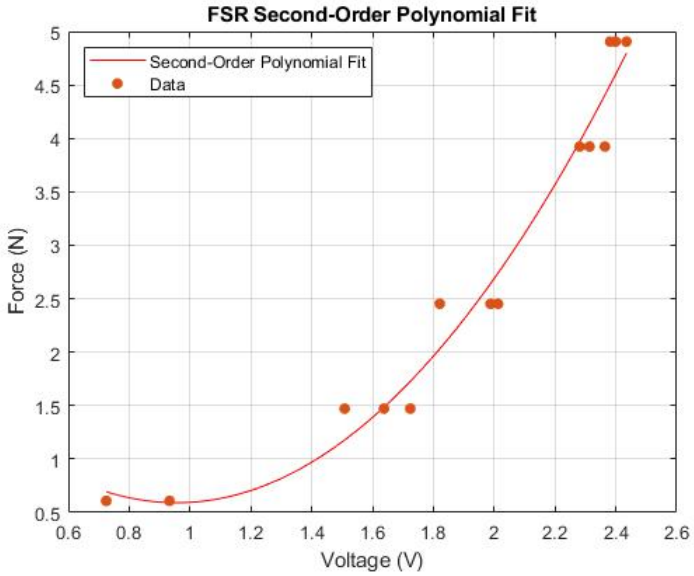

Figure 3: Transfer Function Plot of Force Sensitive Resistor

The resulting transfer function was determined.

$$Force = 1.923 * Voltage^2 - 3.677 * Voltage + 2.348$$

where Force is in units of Newtons and Voltage is in units of Volts. Because this transfer function becomes less accurate at small voltages, a linear interpolation was used between the apex of the

quadratic and zero.

To implement this in Arduino, the output voltage of the circuit was connected to the A0 pin and the 3.3V and 5V lines were connected from the Arduino to the circuit. AnalogRead() was then used to read the sensor, and the transfer function was used to calculate the force on the sensor.

1.1.2 Servo Motor Control

The servo was controlled using the Arduino Servo Library and controlled through PWM sent through the Digital Pin 10 of the Arduino (Figure 4). The +5V input was from the power supply since the current draw of the servo interfered with other sensor readings. A basic circuit diagram is shown in Figure 4 of the servo connecting to the Arduino. Inputs from the sensors (0-1023) were mapped to an angle (0 to 180 degrees) for the servo to turn to.

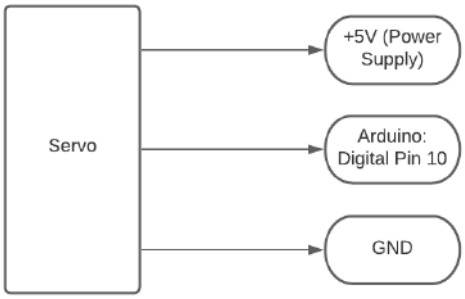

Figure 4: Servo Circuit Diagram

1.1.3 Arduino-to-GUI Communication

The Arduino and GUI communicated through Serial. The GUI would send commands to the Arduino to actuate motors, sending three numbers. These numbers determined 1) if the Gui control was enabled or disabled, 2) what motor to control if enabled, and 3) what input to send the motor.

The Arduino would send the GUI sensor and motor data, telling the GUI what the sensor and motor values are in real time. These commands were sent by sending a string through serial for which the GUI would have to parse.

1.1.4 State Machine

The state machine was set up to allow for changing what sensor is controlling the motors, and to change what motor the selected sensor is controlling. These state machine states were changed through the use of push buttons or GUI commands. Interrupts were used to trigger the push buttons and a switch statement was used to swap states.

1.2 MRSD Project

For our team's capstone project, I worked on some logistical tasks such as improving the layout of the MRSD website, allowing for much a clearer and presentable format if the team would like

to share the website or if our company sponsor wants to explore the website. Since I am one of the team's designated hardware engineers, I also designed a camera mount adapter in SOLIDWORKS, made a basic drawing (Figure 5) and fabricated the camera mount adapter in the RI Machine Shop (Figure 6). This task was one of the first tasks that our team needed done since later tasks would hinge on our perception elements.

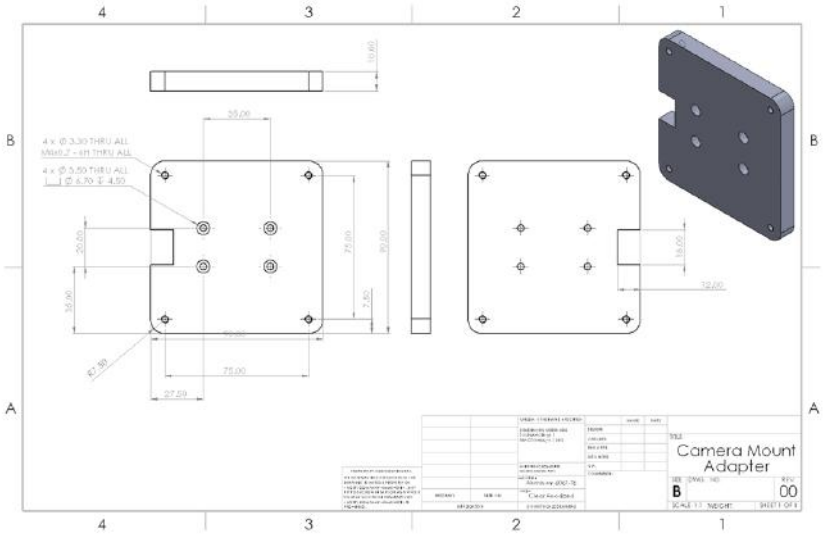

Figure 5: Drawing of Camera Mount Adapter

Figure 6: Camera Mounted onto a VESA Mount Using Fabricated Camera Mount Adapter

As the Controls and Actuation Lead of our team, I have also been exploring and researching different controls algorithms that are being used for orthopedic robots, specifically looking for hybrid force/position control. Various methods were considered including Damping Control through time-warping the trajectory equation based on external force readings or using Model Predictive

Control to not only achieve optimal control, but also guarantee force, velocity, and positional constraints required in our application. It was decided that we would pursue Model Predictive Control for our project based on its merits and potential benefits over other methods. I have also looked at potential ROS packages or repositories that we could use in our development.

2 Challenges

2.1 Sensors and Motors Lab

There were several challenges that our team faced during the development of our Sensors and Motors Lab system. First, our team had a Covid-19 Infection, where Anthony (me) had to quarantine because they were exposed to a person who had tested positive. This made development and integration harder and less coordinated. Good communication within the team helped mitigate this challenge. Implementation was also a huge challenge in this lab. First, when integrating the servo motor and the ambient light sensor, the servo motor current draw would affect the current passing through the light sensor, generating noise. This was resolved by changing the power source of the Servo from the Arduino to the power supply. This was also a challenge because the power supply itself is not very reliable and the voltage tends to vary a lot, sometimes making the servo move without PWM input. Another issue was with the PID control integration. Because of the increased loop execution time, the control signal frequency was decreased, and the PID control had to be re-tuned. Re-tuning and decreasing print statements to Serial mitigated this challenge. The unreliable power supply also changed how the PID constants would affect the system, making tuning more difficult.

2.2 MRSD Project

A major challenge for our MRSD Project revolved around the lack of manipulator specifications and lack of communication with our manipulator arm's manufacturer. This made development of our simulation slower. I helped to coordinate communication with the manufacturer to get the proper information we needed such as the URDF of the arm we are using. However, with the lack of communication, our team also ran into another major hurdle. When we had a meeting with the manufacturer to set up our arm and learn more about its capabilities, we were shocked to find out that the arm had no API support because it is still in development, and would not be available until June 2022 at the earliest. This undoubtedly set our team's schedule back at least one week if not more in terms of hardware. Luckily, we can still proceed to use simulation to test our perception, planning, and controls algorithms. For Controls and Actuation, there have been major challenges while doing research. This challenge mainly stems from my lack of knowledge on advanced controls algorithms and the implementation of these algorithms, but by taking an Optimal Control class and looking at external resources, this challenge was mitigated.

3 Team Work

3.1 Sensors and Motors Lab

The tasks for the sensors and motors lab were assigned to each team member and generally distributed evenly. A summary of each member's contributions can be found in Table 1.

Table 1: Team Member Contributions and Collaborations for the Sensors and Motors Lab

Team Member	Motor	Sensor	Motor Lab Contribution
Anthony Kyu	Servo Motor	Force Sensitive Resistor	• Implemented Servo Motor Control Code • Designed Force Sensitive Resistor Circuit by collecting transfer function measurements, and implementing analog lowpass filtering • Developed State Machine Code • Collaborated with Gunjan for communicating with the GUI from Arduino • Collaborated with the entire team for system Integration and Hardware Integration
Parker Hill	Stepper Motor	Ultrasonic Range Finder	• Developed code for controlling the Stepper Motor • Developed code to measure distance from the Ultrasonic Range Finder • Collaborated with the team to do Hardware/System integration
Sundaram Seivur	DC Motor (Velocity Control)	Ambient Light Sensor	• Collaborated with Kaushik to implement PID Velocity Control for the DC motor • Developed DC Motor Encoder software to obtain encoder counts for later use (in both Position and Velocity measurements) • Developed code and circuit to get measurements for the ambient light sensor • Collaborated with the team to do hardware/system integration, taking the lead on hardware integration
Kaushik Balasundar	DC Motor (Position Control)	Potentiometer	• Collaborated with Sundaram to implement DC motor PID position control • Developed code to obtain measurements from the potentiometer • Collaborated with the team to finish system integration
Gunjan Sethi	GUI		• Developed the GUI using Qt within C++, enabling control of the motors from the GUI and constant sensor measurements • Collaborated with Anthony for GUI+Sensors/Motors Integration • Collaborated with the team for GUI and general debugging

The culmination of the Team's contributions resulted in a successful and functioning system. The full hardware setup can be seen in Figure 7, and a the final GUI can be seen in Figure 8.

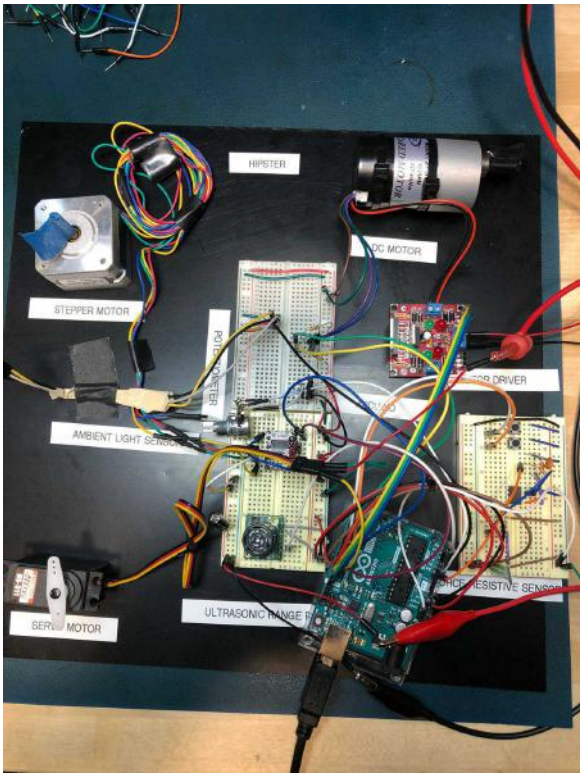

Figure 7: Full Hardware Integration Setup

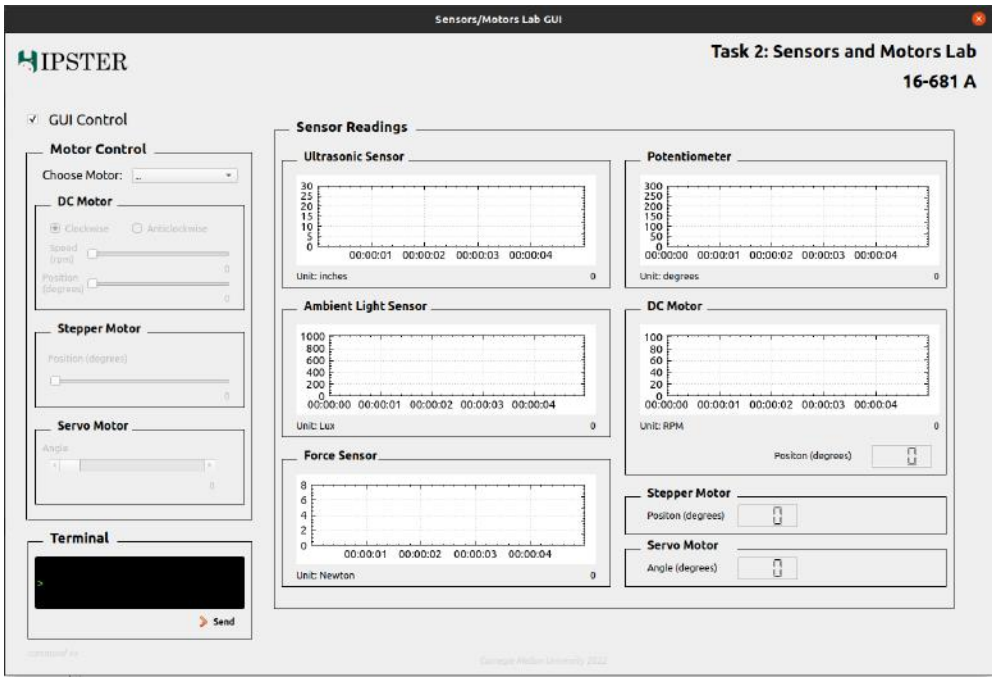

Figure 8: Final GUI to Interface with Sensors and Motors

3.2 MRSD Project

A summary of each team member's contributions towards the MRSD project can be found in the Table below (Table 2).

Table 2: Team Member Contributions and Collaborations Towards the MRSD Project

Team Member	Contributions and Work Descriptions for MRSD Project
Anthony Kyu	• Further developed the MRSD website, improving aesthetics and removing information that our sponsor has asked to remain confidential. This included collaborating with Sundaram to get some ideas about how to present the information on the website. • Designed the camera mount adapter and worked with Sundaram and Parker to fabricate the camera mount adapter in the RI Machine Shop • Trained in the RI Machine Shop • Do extensive research on controls methods and algorithms used for surgical and orthopedic robots, and did a final selection based on the merits of each method found
Parker Hill	• Collaborated with Anthony and Sundaram on fabricating the Camera Mount Adapter • Researching, collecting, and designing marker mounts for the robot arm • Collecting necessary information to design an end effector adapter for the arm that our team is using • Picking up the robot arm from our sponsor • Learning and familiarizing with ROS
Sundaram Seivur	• Collaborated with Anthony and Parker to fabricate the Camera Mount Adapter • Ensuring the team maintains to the planned schedule • Integrating some initial ROS controllers into simulation to increase the acceleration of implementation when Anthony has finished research • Integrating MoveIt with the given arm in simulation to allow for future simulation testing
Kaushik Balasundar	• Setup a ROS Docker environment to ensure that the team can have consistent development • Setup the code repository on Github for our team to have a development pipeline • Collaborated with Sundaram to refine the given URDF for our arm • Collaborated with Gunjan for Open3D ROS integration • Collaborated with Sundaram for MoveIt Package configuration
Gunjan Sethi	• Setup the Atracsys SDK setup and looked through sample scripts to better understand how to implement into our project • Gained deeper understanding of CMake • Linked Atracsys SDK with ROS • Developed a ROS package to discover camera through ROS Node • Manage Jira and general documentation

4 Plans

4.1 Sensor and Motors Lab

We plan on using the lessons gained from the Sensors and Motors Lab moving forward when considering our schedule. Integration was a big hurdle for our team, and we will likely provide more time for integration for our capstone project.

4.2 MRSD Project

To address some of the challenges discussed previously, our team reached out to our company sponsor and came to an agreement to get a new arm. This will be done in one of two ways. Ask professors within CMU RI to see if there is a spare arm that suites our needs, or to ask Kinova to send a Kinova Gen 3 Arm. To make this more likely to succeed our company sponsor would ask Kinova for us. Using a well supported arm should help our project accelerate forward as well, since we would not have to worry about filling in gaps (either in the URDF, drivers, etc.) that we had to do with our previous robot arm. Arms such as UR5s, Kinova Gen 3s, or even Franka Arms would be sufficient as a replacement. Because of these challenges, we will also be likely adjusting our schedule accordingly to ensure that we can maintain a reasonable and organized timeline.

In the following weeks, I will be familiarizing myself more with ROS, and then proceed to start designing and developing our Model Predictive Controller within ROS and simulate ideally before the Preliminary Design Review, although with the recent challenges and delays, these plans could be pushed back. I will also be designing the state machine use ROS Smach for the user to swap between gravity compensation mode and autonomous mode. This task, however, is based on what arm we will be using, which is yet to be determined.

5 Appendix A: Arduino Code

Because I worked on the State Machine and Arduino-to-GUI communication, the entire integrated code is below. Please note that the sensor and motor I worked on are only the Force Sensitive Resistor (FSR) and the Servo Motor. Other code relating to other motors and sensors were developed by my teammates.

```
/** Libraries */
#include <BasicStepperDriver.h> // From Library "StepperDriver" by Laurentiu Badea
#include <PinChangeInterrupt.h> // From Library "PinChangeInterrupt" by NicoHood
#include <PinChangeInterruptBoards.h>
#include <PinChangeInterruptPins.h>
#include <PinChangeInterruptSettings.h>
#include <Servo.h>
#include <string.h>
#include <util/atomic.h> // For the ATOMIC_BLOCK macro


/** Macros */
#define LIGHTSENSORPIN A2 //Ambient light sensor reading
#define light_window 50

/** Pin Assignments */
const int sensorPushButton = 4;
const int motorPushButton = 5;
const int servoPin = 10;
const int FSR_Pin = A0;
const int USRF_Pin = A5;
const int Stepper_Dir_Pin = 7;
const int Stepper_Step_Pin = 8;
//Motor encoderPins
const int ENCA = 2;
const int ENCB = 3;
//Motor signal pins
const int PWM = 9;
const int IN1 = 11;
const int IN2 = 12;
//Potentiometer pins
const int pot = A1;

/** Global Variables */
// State Machine Variables
enum MotorState {SERVO, DC_MOTOR_POS, DC_MOTOR_VEL, STEPPER};
enum SensorState {FSR, ULTRASONIC, POT, LIGHT, GUI};
static MotorState motorState;
```

```
static SensorState sensorState;
static SensorState prevSensor;
const unsigned long debouncingPeriod = 250; // ms
unsigned long motorButtonTimer;
unsigned long sensorButtonTimer;
int motorInput;

// GUI Variables
int guiInput;
const char delim[2] = ",";

// FSR Variables
const int fsrSampleFrequency = 1000; // sampling frequency in hertz
const int fsrWindowLength = 25; // length of window (samples)
double window[fsrWindowLength]; // window of moving-average filter
const double Vin = 3.3; // voltage input to voltage divider circuit
const double saturationVoltage = 2.7; // volts
const double maxAnalogVoltage = 5; // volts
double force;
unsigned long fsrTimer;
double prevFSRValue = 0;

// Ambient Light Sensor Variables
int a = 0;
int a_analog = 0;
float sum = 0;
float sum_analog = 0;
float readings[light_window];
float readings_analog[light_window];
float avg = 0;
float avg_analog = 0;
float lux;
int lux_analog;
long lightTimer = 0;

int m = 0;
float sum_motor = 0;
float readings_motor[light_window];
float avg_motor = 0;

// Ultrasonic Range Finder Variables
float distance;
float new_distance;
float duration;
const float usrf_scale = 147;
```

```
const int usrfWindowSize = 10;
double usrf_window[usrfWindowSize];
float usrf_sma = 0;

// Potentiometer Variables
int raw = 0;
int target_map = 0;
float angle = 0;
float potAngle = 0;
float potRaw = 0;

// Servo Variables
Servo myservo;
const int servoUpdateFrequency = 10; // update every 100 ms
unsigned long servoTimer;

// Stepper Motor Variables
const int Stepper_Steps = 200;
const int Stepper_RPM = 240;
const int Stepper_Microsteps = 1;
BasicStepperDriver stepper(Stepper_Steps, Stepper_Dir_Pin, Stepper_Step_Pin);
int previousStepperInput = 0;
int stepperInputDiff;
int stepperRotation;
const int stepperDegrees = 360;
const float stepperUpdate = 100;
float stepperTimer;

// DC Motor Variables - Velocity Control
// DC Motor Velocity Control Variables
int posPrev = 0;
float previousError = 0;
char incomingByte;
volatile int encoderValue = 0;
float edot = 0;
float proportional = 0.5; //2; //k_p = 0.5
float integral = 0.55; //3 //k_i = 3
float derivative = 0.0001;//0.12; //k_d = 1
float controlSignal = 0; //u - Also called as process variable (PV)
float errorIntegral = 0;
float deltaT;
//int pos;
long previousTime;
long currentTime;
```

```
// DC Motor Variables - Position Control
long prevT = 0;
float eprev = 0;
float eintegral = 0;
float deltaTpos;
float kp = 25;
float kd = 1.2;
float ki = 0.5;
float currT = 0;
float target = 0;

// Setup**************************************************************
void setup()
{
  // Attach pins
  pinMode(FSR_Pin, INPUT);
  pinMode(sensorPushButton, INPUT_PULLUP);
  pinMode(motorPushButton, INPUT_PULLUP);
  attachPCINT(digitalPinToPCINT(sensorPushButton), changeSensor, FALLING);
  attachPCINT(digitalPinToPCINT(motorPushButton), changeMotor, FALLING);
  pinMode(USRF_Pin, INPUT);
  pinMode(LIGHTSENSORPIN,  INPUT);


  // State machine setup
  sensorButtonTimer = 0;
  motorButtonTimer = 0;
  motorState = SERVO;
  sensorState = POT;


  // GUI setup
  prevSensor = FSR;
  Serial.begin(115200);
  guiInput = 0;

  // Sensor setup
  fsrTimer = micros();
  force = 0;

  // Motor setup
  motorInput = 0;
  setupServo(myservo, servoPin);
  stepperTimer = micros();
  stepper.begin(Stepper_RPM, Stepper_Microsteps);
```

```
  pinMode(ENCA, INPUT);
  pinMode(ENCB, INPUT);
  pinMode(PWM, OUTPUT);
  pinMode(IN1, OUTPUT);
  pinMode(IN2, OUTPUT);
  attachInterrupt(digitalPinToInterrupt(ENCA), encoder, RISING);
  currentTime = micros();
  previousTime = micros();
}


// loop****************************************************************
void loop()
{
  // Read from GUI
  // Reads 3 comma-delimited numbers.
  // First number determines whether GUI is enabled,
  // second is which motor is enabled (same order as enum declaration above),
  // third number is input for the chosen motor
  if (Serial.available()) {
    String incomingMessage = Serial.readString();
    char buffer[incomingMessage.length() + 1];
    incomingMessage.toCharArray(buffer, incomingMessage.length() + 1);
    char *token;
    token = strtok(buffer, delim);
    int inputs[3] = { -1, -1, -1};
    for (int i = 0; i < 3; i++) {
      if (token != NULL) {
        inputs[i] = atoi(token);
        token = strtok(NULL, delim);
      } else {
        //Serial.println("Invalid Message");
        i = 3;
      }
    }
    if (inputs[0] != -1 && inputs[1] != -1 && inputs[2] != -1) {
      for (int i = 0; i < 3; i++) {
        switch (i) {
          case 0:
            if (inputs[i] == 0) {
              //              Serial.println("Gui Disabled");
              disableGUIControl();
            } else {
              //              Serial.println("Gui Enabled");
              enableGUIControl();
```

```
          }
          break;
        case 1:
          //            Serial.println(inputs[i]);
          guiChangeMotor((MotorState) inputs[i]);
          break;
        case 2:
          if (sensorState == GUI) {
            //              Serial.println(inputs[i]);
            //              controlSignal = 0;
            //              errorIntegral = 0;
            guiInput = inputs[i];
          }
          break;
      }
    }
  }
}

// Read FSR Sensors
if ((micros() - fsrTimer) > ((unsigned long)1000000) / fsrSampleFrequency) {
  force = getForce();
  fsrTimer = micros();
}
// Read USRF Sensor
distance = getDistance();

//Read ambient light sensor - Lux output for GUI
if ((micros() - lightTimer) > 100000) {
  lux = ambientLightRead();
  lux_analog = ambientLightAnalog();
  lightTimer = micros();
}

//Read Potentiometer angle
potRaw = getPot();
angle = getAngle();


// Determine which sensor is the input to the motor
switch (sensorState)
{
  case FSR:
    motorInput = f2input(force);
    break;
```

```
    case ULTRASONIC:
      motorInput = d2input(distance);
      break;
    case POT:
      motorInput = potRaw;
      break;
    case LIGHT:
      motorInput = lux_analog;
      break;
    case GUI:
      motorInput = guiInput;
      break;
  }
  // Drive chosen motor
  switch (motorState)
  {
    case SERVO:
      //      Serial.println("SERVO");
      if ((micros() - servoTimer) > ((unsigned long)1000000) / servoUpdateFrequency) {
        updateServo(myservo, motorInput);
        servoTimer = micros();
      }
      break;
    case STEPPER:
      //      Serial.println("STEPPER");
      if ((millis() - stepperTimer) > stepperUpdate) {
        updateStepper(motorInput, previousStepperInput);
        stepperTimer = millis();
      }
      break;
    case DC_MOTOR_POS:
      //      Serial.println("DC_MOTOR_POS");

      currT = micros();
      deltaTpos = ((float) (currT - prevT) / 1000000);
      if (sensorState != GUI) {
        int target_map = map(motorInput, 0, 1023, 0, 300);
        //          motorInput = 0.25 * target_map;
        //          Serial.println(motorInput);
      }
      if (deltaTpos > 0.045) {


        //    Serial.print("Target: ");
        //    Serial.println(motorInput);
```

```
// time difference
long currT = micros();
//  float deltaTpos = ((float) (currT - prevT))/( 1.0e6 );
prevT = currT;

// Read the position in an atomic block to avoid a potential
// misread if the interrupt coincides with this code running
// see: https://www.arduino.cc/reference/en/language/variables
// /variable-scope-qualifiers/volatile/
int pos = 0;
ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
  pos = encoderValue;
}
//  Serial.print("Current Position: ");
//  Serial.println(pos);
// error
target = motorInput / 4;
int e = pos - target;

// derivative
float dedt = (e - eprev) / (deltaTpos);

// integral
eintegral = eintegral + e * deltaTpos;

// control signal
float u = kp * e + kd * dedt + ki * eintegral;
//  Serial.print("u: ");
//  Serial.println(u);

// motor power
float pwr = fabs(u);
if ( pwr > 255 ) {
  pwr = 255;
}

// motor direction
int dir = 1;
if (u < 0) {
  dir = -1;
}

// signal the motor
setMotor(dir, pwr, PWM, IN2, IN1);
```

```
    // store previous error
    eprev = e;

    //  Serial.print(target);
    //  Serial.print(" ");
    //  Serial.print(pos);
    //  Serial.println();
    //  Serial.print(u);
    //  Serial.print(" ");

  }
  break;


case DC_MOTOR_VEL:
  if (sensorState != GUI)
  {
    motorInput = map(motorInput, 0, 1023, 0, 100);
    //        Serial.println(motorInput);
  }

  int pos_vel = encoderValue;

  long currentTime = micros();
  deltaT = ((float) (currentTime - previousTime) / 1000000);
  //      Serial.print(deltaT);

  if (deltaT > 0.01)
  {
    float velocity = (pos_vel - posPrev) / deltaT;
    posPrev = pos_vel;
    previousTime = currentTime;

    float v1 = velocity * 60 / 90;
    //        Serial.println(v1);
    float error = motorInput - v1;
    float u = calculatePIDVel(error);

    int dir = 1;
    if (u < 0)
    {
      dir = -1;
    }
```

```
      int pwr = (int) fabs(u);
      if (pwr > 255)
      {
        pwr = 255;
      }
      else if (pwr < 0)
      {
        pwr = 0;
      }
      setMotor(dir, pwr, PWM, IN1, IN2);

      sum_motor = sum_motor - readings_motor[m];
      readings_motor[m] = v1;
      sum_motor =  sum_motor + v1;
      m = (m + 1) % light_window;
      avg_motor = sum_motor / light_window;
      //        Serial.println(avg_motor);
    }
    break;
}

// Sends outgoing messages to GUI to inform gui of sensor readings
// and motor positions/speeds
Serial.print("u,"); Serial.println(distance); // u is for ultrasonic sensor
Serial.print("f,"); Serial.println(force); // f is for force sensor
Serial.print("a,"); Serial.println(lux); // a is for ambient light sensor
Serial.print("p,"); Serial.println(angle); // p is for potentiometer reading
// m is for motor, and the numbers after that indicate which motor is
// switched on (0-3), servo-pos, dc-motor-pos, dc-motor-speed,
// stepper-pos in that order
// only sends motor reading for the motor currently selected,
// otherwise the reading is 0
// Example: m,0,512,0,0,0, which means that the servo motor is selected
// and has an input of 512. Other motors are set to 0.
switch (motorState)
{
  case SERVO:
    Serial.print("m,"); Serial.print(0); Serial.print(",");
    Serial.print(motorInput); Serial.print(","); Serial.print(0);
    Serial.print(","); Serial.print(0); Serial.print(",");
    Serial.println(0);
    break;
  case DC_MOTOR_POS:
    Serial.print("m,"); Serial.print(1); Serial.print(","); Serial.print(0);
    Serial.print(","); Serial.print(motorInput); Serial.print(",");
```

```
      Serial.print(0); Serial.print(","); Serial.println(0);
      break;
    case DC_MOTOR_VEL:
      Serial.print("m,"); Serial.print(2); Serial.print(","); Serial.print(0);
      Serial.print(","); Serial.print(0);
      Serial.print(","); Serial.print(avg_motor);
      Serial.print(","); Serial.println(0);
      break;
    case STEPPER:
      Serial.print("m,"); Serial.print(3); Serial.print(","); Serial.print(0);
      Serial.print(","); Serial.print(0); Serial.print(","); Serial.print(0);
      Serial.print(","); Serial.println(motorInput);
      break;
  }
}

// State Machine Functions************************************************
void changeSensor()
{
  if ((millis() - sensorButtonTimer) > debouncingPeriod) {
    switch (sensorState)
    {
      case FSR:
        sensorState = ULTRASONIC;
        //        Serial.println("ULTRASONIC State");
        break;
      case ULTRASONIC:
        sensorState = POT;
        //        Serial.println("POT State");
        break;
      case POT:
        sensorState = LIGHT;
        //        Serial.println("LIGHT State");
        break;
      case LIGHT:
        sensorState = FSR;
        //        Serial.println("FSR State");
        break;
      case GUI:
        break;
    }
    sensorButtonTimer = millis();
  }
}
```

```
void enableGUIControl()
{
  if (sensorState != GUI) {
    motorInput = 0;
    prevSensor = sensorState;
    sensorState = GUI;
  }
}

void disableGUIControl()
{
  sensorState = prevSensor;
}

void changeMotor()
{
  if ((millis() - motorButtonTimer) > debouncingPeriod) {
    switch (motorState)
    {
      case SERVO:
        motorInput = 0;
        motorState = DC_MOTOR_VEL;
        //        Serial.println("DC_MOTOR_POS State");
        break;
      case DC_MOTOR_POS:
        motorInput = 0;
        motorState = DC_MOTOR_VEL;
        errorIntegral = 0;
        //        Serial.println("DC_MOTOR_VEL State");
        break;
      case DC_MOTOR_VEL:
        motorInput = 0;
        motorState = STEPPER;
        //        Serial.println("STEPPER State");
        break;
      case STEPPER:
        motorInput = 0;
        motorState = SERVO;
        //        Serial.println("SERVO State");
        break;
    }
    motorButtonTimer = millis();
  }
}
```

```
void guiChangeMotor(MotorState state)
{
  if (state >= 0 && state <= 3) {
    motorState = state;
    if (state == STEPPER || state == SERVO) {
      setMotor(0, 0, PWM, IN1, IN2);
    }
  }


}

// Sensor Functions*****************************************************
// FSR Functions-------------------------------------------------------
/**
   Updates the Force Sensitive Resistor reading and gives the estimated
   force applied on the FSR
   @return the Force (N) that is applied to the FSR
*/
double getForce()
{
  return v2F(updateFilter(min(saturationVoltage, (analogRead(A0) / 1023.0)
  * maxAnalogVoltage)));
}

/**
   Maps a given force to a 10-bit integer to be used as motor input
   @param f the force read by the FSR
   @return the motor input (10-bit integer)
*/
int f2input (double f)
{
  return (f / v2F(saturationVoltage)) * 1023;
}

/**
   Updates the moving-average filter/window by recalculating the filters
   output with the given current input,
   and then updates window with the given value.
   @param v the most recent sampled voltage
   @return the most recent filter output
*/
double updateFilter(double v)
{
  prevFSRValue = ((prevFSRValue * ((double)fsrWindowLength)) - window[0] + v)
```

```
  / ((double)fsrWindowLength);
  for (int i = 0; i < fsrWindowLength - 1; i++) {
    window[i] = window[i + 1];
  }
  window[fsrWindowLength - 1] = v;
  return prevFSRValue;
}

/**
   Converts voltage (volts) to force (newtons).
   Uses the 2nd-order calibration curve that was calculated in
   part 1 of the project.
   @param v the voltage (volts)
   @return the force (newtons)
*/
double v2F (double v)
{
  // 0.956 volts is when the calibration curve is actually true
  if (v < 0.956 && v > -0.001) {
    return v * 0.59 / 0.956; // linear increase
  }
  return 1.923 * (v * v) + -3.677 * v + 2.348;
}

// USRF Functions-----------------------------------------------------
float getDistance() {
  new_distance = 5 * analogRead(USRF_Pin) / 9.8;
  if (distance > 25) {
    distance = 25;
  }
  //Implementing a 5 point SMA to eliminate noise and ease transitions
  for (int i = 0; i < usrfWindowSize; i++) {
    if (i == usrfWindowSize - 1) {
      usrf_window[i] = new_distance;
    }
    else {
      usrf_window[i] = usrf_window[i + 1];
    }
  }
  usrf_sma = usrf_sma + (1 / (double)usrfWindowSize) * (usrf_window[4] - usrf_sma);
  return usrf_sma;
}
int d2input(double d) { //Conversion factor from distance to 0->1024
  double input = map(d, 5.6, 25, 1024, 0);
  if (input > 1023) {
```

```
    input = 1023;
  }
  else if (input < 0) {
    input = 0;
  }
  return input;
}

//Ambient Light sensor function
float ambientLightRead() {
  sum = sum - readings[a];
  float reading = analogRead(LIGHTSENSORPIN);
  readings[a] = reading;
  sum =  sum + reading;
  a = (a + 1) % light_window;
  avg = sum / light_window;

  //      Serial.print("Reading: "); Serial.print(reading);

  //      Serial.print("Avg. Reading: "); Serial.print(avg);

  float volt = (avg * 5.0) / 1023.0;
  //      Serial.print(" Volt:"); Serial.print(volt);

  float curr = pow(10, 6) * (volt / 100000);
  //      Serial.print(" Curr:"); Serial.print(curr);

  float lux = pow(10, 0.1 * curr);
  //      Serial.print(" Lux: "); Serial.println(lux);
  return lux;
}

//Ambient sensor analog output
float ambientLightAnalog() {
  sum_analog = sum_analog - readings_analog[a_analog];
  float reading_analog = analogRead(LIGHTSENSORPIN);
  readings_analog[a_analog] = reading_analog;
  sum_analog =  sum_analog + reading_analog;
  a_analog = (a_analog + 1) % light_window;
  avg_analog = sum_analog / light_window;

  //    Serial.print("Avg. Reading: "); Serial.print(avg_analog);
  return avg_analog;
}
```

```
//Potentiometer Functions-------------------------------------------

float getAngle() {

  int raw = analogRead(pot);
  int potAngle = map(raw, 0, 1023, 0, 300);
  int target_map = map(raw, 0, 1023, 0, 600);
  float target = 0.25 * target_map;
  return potAngle;
}

float getmotorAngle() {
  int raw = analogRead(pot);
  int target_map = map(raw, 0, 1023, 0, 600);
  float target = 0.25 * target_map;
  return target;
}

float getPot() {
  int raw = analogRead(pot);
  return raw;
}

// Motor Functions*****************************************************
// Servo Functions-----------------------------------------------------
/**
   Runs the functions needed to setup the servo
   @param s the servo object
   @param controlPin the pin controlling the PWM sent to the servo
*/
void setupServo(Servo s, int controlPin)
{
  s.attach(controlPin);
  s.write(0);
  servoTimer = micros();
}

/**
   Updates the servo position based on the input
   @param s the servo object
   @param input the 10-bit number representing the input to the servo
   (maps input to between 0-180)
*/
void updateServo(Servo s, int input)
{
```

```
  s.write((int)map(input, 0, 1023, 0, 180));
}

// Stepper Functions-------------------------------------------------------
void updateStepper(int input, int previous_input) {
  stepperInputDiff = input - previous_input;
  stepperRotation = stepperInputDiff * ((double)stepperDegrees / 1023);
  stepper.rotate(stepperRotation);
  previousStepperInput = input;
}


// DC Motor Velocity Control

// sVelocity Control main PID Loop (calculatePID(errorValue) gets
// called insider here)

void velocityControl() {

  int pos = encoderValue;
  float velocity = (pos - posPrev) / deltaT;
  posPrev = pos;
  previousTime = currentTime;

  float v1 = velocity * 60 / 90;
  //  Serial.print("ActualVelocity: ");
  //  Serial.println(v1);
  //  Serial.print("MotorInput: ");
  //  Serial.println(motorInput);
  float error = motorInput - v1;
  float u = calculatePIDVel(error);
  //    Serial.print("u: ");
  //  Serial.println(u);
  int dir = 1;
  if (u < 0) {
    dir = -1;
  }
  int pwr = (int) fabs(u);
  if (pwr > 255) {
    pwr = 255;
  }
  else if (pwr < 0) {
    pwr = 0;
  }
  //  Serial.println("Inside velocity");
```

```
  setMotor(dir, pwr, PWM, IN1, IN2);

}

//Velocity control PID Function

float calculatePIDVel(float errorValue)
{
  edot = (errorValue - previousError) / deltaT; //edot = de/dt - derivative term
  errorIntegral = errorIntegral + (errorValue * deltaT);
  Serial.println(errorIntegral);
  if (errorIntegral > 480) {
    errorIntegral = 480;
  } else if (errorIntegral < -480) {
    errorIntegral = -480;
  }
  controlSignal = (proportional * errorValue) + (derivative * edot) +
  (integral * errorIntegral); //final sum, proportional term also calculated here
  if (controlSignal > 260) {
    controlSignal = 260;
  } else if (controlSignal < -260) {
    controlSignal = -260;
  }
  //save the error for the next iteration to get the difference (for edot)
  previousError = errorValue;
  Serial.println(controlSignal);
  return controlSignal;
}

//Velocity and Position Control Encoder

void encoder() {
  if (digitalRead(ENCB) == HIGH) // if ENCODER_B is high increase the count
    encoderValue++; // increment the count

  else // else decrease the count
    encoderValue--;  // decrement the count

}

//General Motor Function

void setMotor(int dir, int pwmVal, int pwm, int in1, int in2)
{
  //  Serial.print(pwmVal);
```

```
  analogWrite(pwm, pwmVal); // Motor speed
  if (dir == 1) {
    // Turn one wayb
    digitalWrite(in1, HIGH);
    digitalWrite(in2, LOW);
  }
  else if (dir == -1) {
    // Turn the other way
    digitalWrite(in1, LOW);
    digitalWrite(in2, HIGH);
  }
  else {
    // Or dont turn
    digitalWrite(in1, LOW);
    digitalWrite(in2, LOW);
  }
}
```

6 Appendix B: Sensors and Motors Lab Quiz

6.1 Question 1: Reading A Datasheet - ADXL335

Refer to the ADXL335 accelerometer datasheet (https://www.sparkfun.com/datasheets/Components/SMD/adxl335.pdf) to answer the below questions.

1. What is the sensor's range?

   The sensor's typical range is ±3.6 g. The minimum range is ±3 g.

2. What is the sensor's dynamic range?

   The sensor's typical dynamic range 7.2 g, while the minimum is 6 g.

3. What is the purpose of the capacitor C_{DC} on the LHS of the functional block diagram on p. 1? How does it achieve this?

   The purpose of the C_{DC} capacitor is to make it a decoupling capacitor. This capacitor ensures that any high frequency noise in the power line is filtered out. In other words, this capacitor will decrease the magnitude of any high frequency spikes to the voltage. The capacitor achieves this by its nature to resist fast changes in the voltage.

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

   Voltage = (0.3 * acceleration/ 9.81) + 1.5
   Where Voltage is in units of Volts.
   Where acceleration is in units of m/s^2
   And 9.81 is g.

5. What is the largest expected nonlinearity error in g?

   The largest expected nonlinearity error is ±0.0108g.

6. What is the sensor's bandwidth for the X- and Y-axes?

   The sensor's bandwidth is 1600 Hz.

7. How much noise do you expect in the X- and Y-axis sensor signals when your measurement bandwidth is 25 Hz?

   The amount of noise expected in the X- and Y-axis at a measurement bandwidth of 25 Hz is around 948.68 µg.

8. If you didn't have the datasheet, how would you determine the RMS noise experimentally? State any assumptions and list the steps you would take.

I would connect the sensor to an oscilloscope and power the accelerometer with a DC (0 Hz) voltage using a DC power supply. Keeping the accelerometer still (physical signal of 0 Hz), I would measure the output voltage of the sensor with the oscilloscope and measure the RMS noise from those measurements (either by using oscilloscope functions or by recording data and using MATLAB).

6.2 Question 2: Signal conditioning

Filtering:

1. Name at least two problems you might have in using a moving average filter.
    - A moving average filter will have delay in the signal because it is averaging multiple past values.
    - A moving average filter will filter out high frequency signals, so if our physical signal is high frequency, we may be filtering those important signals out.
2. Name at least two problems you might have in using a median filter.
    - A median filter may ignore the ground truth values if the outliers are arranged such that the outliers are the median values. For instance, Gaussian noise which is evenly distributed noise may cause the noise to become dominant in the filter due to its
    - Median filters are computationally more expensive because you must arrange all the values to determine which value is the median.

Op-Amps:

In the following questions, you want to calibrate a linear sensor using the circuit in Fig. 1 so that its output range is 0 to 5V. Identify in each case: 1) which of V1 and V2 will be the input voltage and which the reference voltage; 2) the values of the ratio Rf/Ri and the reference voltage. If the calibration can't be done with this circuit, explain why.
Using the circuit diagram below (Figure 9), the following equations were derived:

V1 – Ri*I – Rf*I = Vout
V1 – Ri*I = V2
I = (V1 – V2)/Ri
Vout = V2 + (Rf/Ri)*(V2 – V1)

- 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).

  1) V2 will be the input and V1 will be the reference voltage.

  2) The value of the ratio of Rf/Ri is 1, and the reference voltage (V1) should be -3V to satisfy the mapping of the input to output range.

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

  Based on the derived equations, and the given input and output mappings, it is not possible to solve for a Rf/Ri ratio and reference voltage. By setting V1 to the reference voltage, the Rf/Ri ratio becomes negative (not physically possible), and by setting V2 to reference voltage, the Rf/Ri is again negative and V2 becomes unsolvable (divide by zero).

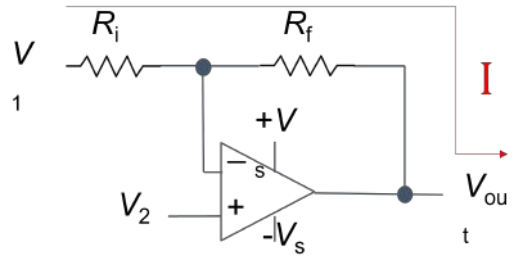

Figure 9: Op-Amp Gain and Offset Circuit

6.3 Question 3: Control

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.

   - For the Proportional control term, you would take the motor encoder output to get the current position. Subtract the current position by the desired position (error) and multiply by a Kp term to get the proportional term.
   - For the Integral term, you would sum up the error over time for each time step (current position minus desired position) and multiply by the control timestep. This term would then be multiplied by a constant Ki to get the Integral term.
   - For the derivative term, you would subtract the current position from the previous position and divide the result by the time step. Then multiply this by a constant Kd to get the derivative term. This term tends to increase the magnitude of noise too.

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

   I would increase the proportional term which would decrease the rise time and increase how much the system reacts to error, therefore increasing system responsiveness.

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 would use the Integral term because this term will accumulate that steady-state error over time and feed it back into the control loop. By doing this, the control signal will push the position to the desired position over time, decreasing the steady-state error to zero over time.

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 would use the Derivative term because this term reduces the control signal gains as the current position approaches the desired position. This would limit the overshoot of the system.