Sensors and Motors Lab

Individual Lab Report #1

Akshit Gandhi

Team H (PhoeniX)

February 13, 2019

Team Mates:

Shubham Garg
Parv Parkhiya
Zhihao Zhu

Individual Progress

Sensors and Motor Control Lab

My role for the lab was to interface the DC Motor and the analog temperature sensor. My deliverables were three major functions to control the position of the motor, control the velocity of the motor and to read the raw temperature sensor value and convert it to degrees. More specifically I wrote 2 PID controllers to control the position given a setpoint in degrees and control the velocity given the velocity setpoint in degrees/sec for either direction rotations.

To control the motor the preliminary task was to understand the working of the encoder on the motor and getting the encoder pulses as inputs and count the number of rotations performed by the motor in degrees. The fundamental thing was to develop a ISR to increment or decrement the counter which counts the ticks of the encoder. Since we have two encoder pin A and pin B, I have a flag for each pin which denotes if we are expecting a rising edge on pin A or pin B and if let's say the ISR for pin A is called and it's flag is set then it means that the motor is rotating in the same direction and I can increment the encoder counter and decrement if that was the opposite case in ISR for pin B. The given logic is represented in the code as follows with 2 ISRs and some global variables.

```
static int pinA = 2;
static int pinB = 3;
volatile byte aFlag = 0;
volatile byte bFlag = 0;
volatile int encoderPos = 0;
volatile int oldEncPos = 0;

void PinA(){
  cli();
  reading = PIND & 0xC;
  if(reading == B00001100 && aFlag) {
    encoderPos --;
    bFlag = 0;
    aFlag = 0;
  }
  else if (reading == B00000100) bFlag = 1;
  sei();
}

void PinB(){
  cli();
  reading = PIND & 0xC;
  if (reading == B00001100 && bFlag) {
    encoderPos ++;
    bFlag = 0;
    aFlag = 0;
  }
  else if (reading == B00001000) aFlag = 1;
  sei();
}
```

The functions cli() and sei() stop and restart the listening on the interrupt pin respectively, preventing us from reading inconsistency. aFlag and bFlag are the variables which help us to understand the sense of rotation direction.

The very first thing which I did was to calculate the number of encoder ticks required to traverse full 360 degrees rotation of the motor. For my motor it was 109 ticks equal to 360 degrees. This value can now be used to convert arbitrary ticks into degrees. This was through trial and error to find the encoder ticks and also verified using the datasheet.

I have followed a Object Oriented Programming approach for the PID Controller where I have a class PID_controller which has some attributes like Kp, Ki, Kd, errorSum and lastError along with the encoder value which depicts a full 360 degree motion. The class has a constructer that initializes the gains, error terms and also the encoder value for 360-degree motion. The class has 2 functions which are the controllers for rotation and speed and a function to control the rotation direction. Both the controller functions take a setpoint and apply PID control strategy to compute a PWM output signal for the motor control. To control the motor rotation direction like forward and reverse, I have a function inside the controller that changes the direction based on the output from the PID controller. The above features can be seen in the code snippet below:

```
class PID_controller {
  public:
  double encoder_360;
  double Kp;
  double Ki;
  double Kd;
  double lastError;
  double errorSum;

  PID_controller(double encoder_360, double Kp, double Ki, double Kd) {
    this->encoder_360 = encoder_360;
    this->Kp = Kp;
    this->Ki = Ki;
    this->Kd = Kd;
    this->lastError = 0.0;
    this->errorSum = 0.0;
  }

  void refresh() {
    this->lastError = 0.0;
    this->errorSum = 0.0;
  }
```

The individual controllers are shown below:

```
int controlDegrees(double setpoint) {
   double output = 0.0;
   double error = setpoint - c_angle;
   errorSum += error*delta_t;

   double error_D = (error - this->lastError) / delta_t;
   this->lastError = error;
   output = Kp * error + Ki * errorSum + Kd * error_D;
   //Performing Saturation on Integral
   if(errorSum>150) {
     errorSum = 150;
   } else if(errorSum<-150) {
     errorSum = -150;
   }

   //Performing Saturation on PWM Output
   if(output>30) {
     output = 30;
   }else if (output<-30) {
     output = -30;
   }

   //Setting Motor Direction
   if(output > 0) {
     this->controlMotor(1, output);
   } else if (output < 0) {
     this->controlMotor(-1, output);
   }
   return output;
 }

 int controlVelocity(double setpoint) {
   int dir = setpoint>0?1:-1;
   setpoint = abs(setpoint);
   double output = 0.0;
   double error = setpoint - c_vel;
   errorSum += error*delta_t;

   double error_D = (error - this->lastError) / delta_t;
   this->lastError = error;

   output = Kp * error + Ki * errorSum + Kd * error_D;

   //Performing Saturation on Integral
   if(errorSum>750) {
     errorSum = 750;
   } else if(errorSum<-750) {
     errorSum = -750;
   }

   //Performing Saturation on PWM Output
   if(output>255) {
     output = 255;
   }else if (output<-255) {
     output = -255;
   }
   //Setting Motor Direction
   if(output > 0) {
     this->controlMotor(dir, output);
   } else if (output < 0) {
     this->controlMotor(dir, output);
   }
   return output;
 }
```

The output of these controllers are given to the motor controller function which sets the direction and gives the control command to the motor using the PWM on the motor enable pin

```
void controlMotor(int dir, int output) {
  if(dir==1) {
    //Serial.print("Control output: ");Serial.println(output);
    reverse();
    analogWrite(enablePin, output);
  }else {
    output = abs(output);
    //Serial.print("Control output: ");Serial.println(output);
    forward();
    analogWrite(enablePin, output);
  }
 }
```

The helper functions forward and reverse just change the direction by setting the 2 motor driver H-Bridge pins to Low and High.

```
void forward() {
  //Serial.println("Forward");
  digitalWrite(I1Pin, HIGH);
  digitalWrite(I2Pin, LOW);
}

void reverse() {
  //Serial.println("Reverse");
  digitalWrite(I1Pin, LOW);
  digitalWrite(I2Pin, HIGH);
}
```

PID Controller working

The PID controller is the simplest form of controller where the generated output of the controller can be expressed mathematically as:

Output = Kp * error + Ki * integral(error) + Kd * derivative(error)

So, to control the rotation, the error will be difference between the setpoint and the current angle. For controlling the velocity, the error will be difference between setpoint and current angular velocity.

Integral of the error is nothing but a cumulative sum of the error over time and this is stored in the class attribute. The derivative of the error is the difference in the error and previous error divided by delta time (difference in the time of taking the samples). The previous error attribute in the class ensures that it stores the error values so that they can be used in the next time step.

The generated output can be much larger and to cap the values (saturate those values) I have kept a limit of 30 on either side as moving the motor with a PWM of more than 30 caused it to overshoot a lot and took a lot of time to converge, also it helped by putting a limit on the integral error as the motor has a dead band on the PWM and thus the integral error does help in overcoming that band and accumulates error if the motor does not spin, but it also needs to be kept under a saturation limit.

If the output is negative it means that we have to rotate the motor in reverse direction and if the output is positive then we need to rotate the motor in the forward direction.

I have some helper functions that compute the angle, velocity and the acceleration as shown below and pretty self explanatory.

To convert encoder ticks to angle, I have the function getAngle that spits out the current motor shaft angle in degrees.

```
void getAngle() {
  c_angle = abs((encoderPos*360.0)/control->encoder_360);
  //c_angle = int(c_angle)%360;
  cTime = millis()/1000.0;
}
```

Here c_angle specifies the current angle, encoderPos specifies the current encoder counter value and cTime tells us the current time from start in seconds.

```
void getVelocity() {
  c_vel = (c_angle - p_angle) / delta_t;
}

void getAcceleration() {
  c_acc = (c_vel - p_vel) / delta_t;
}
```

Thus, to control the motor you just need to call the appropriate control function via the PID_controller object and pass the desired setpoint.

Temperature Sensor Reading:

The given temperature sensor gives out analog readings and it’s easy to read the analog values using the Arduino’s inbuilt function, but we need to actually convert the value into voltage and convert voltage to an appropriate temperature. The code below shows that after getting the raw value, I map it to a voltage between 0 and 5V, use that voltage value and convert it into degree Celsius. Basically, converting from 10 mv per degree with 500 mV offset.

```
void readTempSensor() {
  tempSensorValue = analogRead(tempSensorPin);

  float voltage = tempSensorValue * 5.0;
  voltage /= 1024.0;

  tempSensorValue = (voltage - 0.5) * 100 ;
}
```

Results:

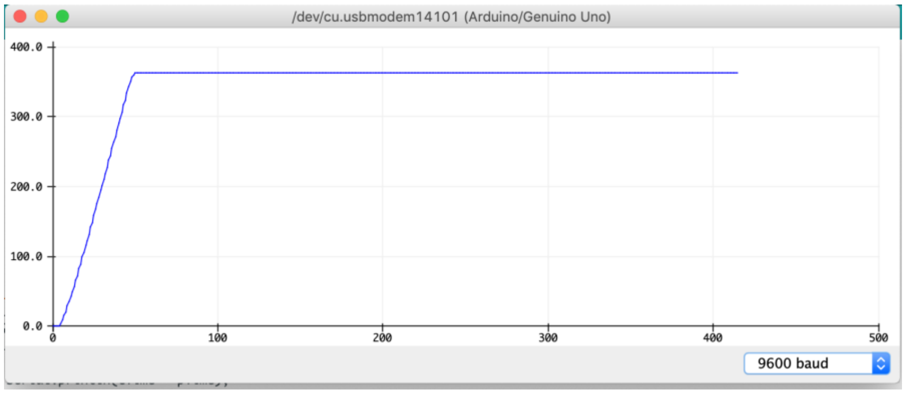

Figure 1: Motor going to 360 degrees

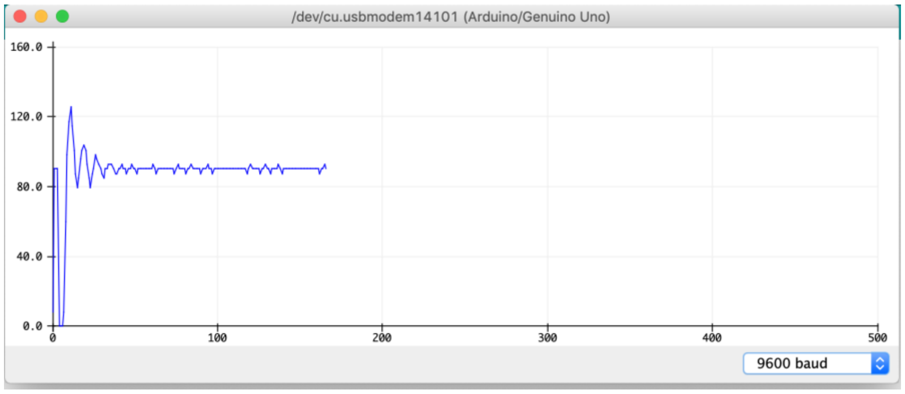

Figure 2: Motor velocity set to 90 degrees/sec

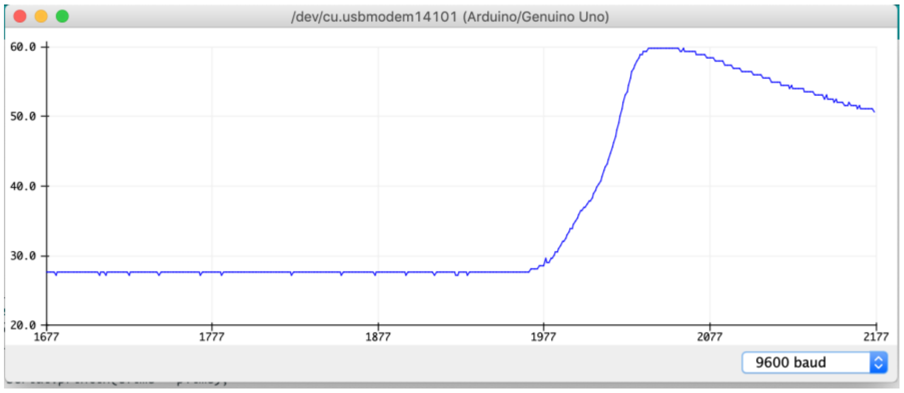

Figure 3: Temperature Sensor reading with heat gun

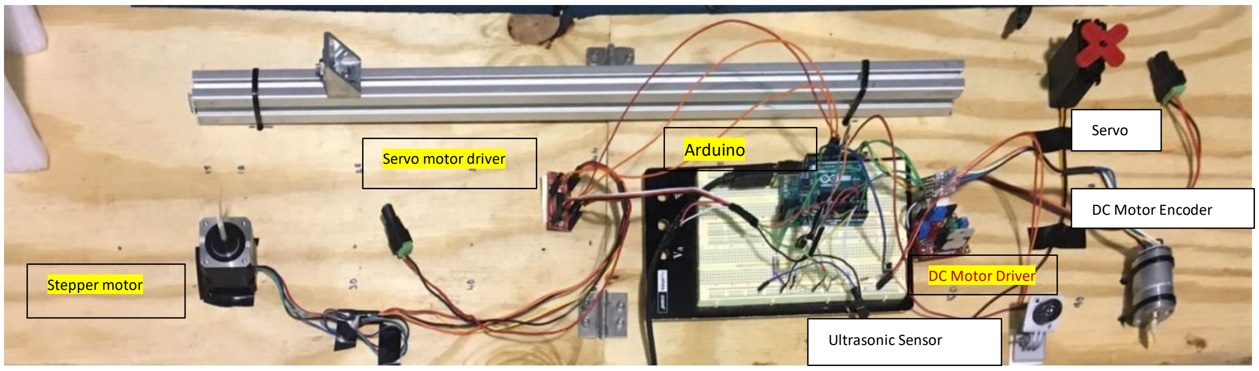

Figure 4: Sensors and Motor Control Setup

PhoeniX

After the conceptual design review, I had been working during the winter break and through the month of January majorly on getting up the UAV into a stable configuration such that it can be controlled manually as well as autonomously using the onboard Jetson and autonomous control of the UAV.

The key tasks performed by me was assembling the UAV (team task), setting up the Jetson by installing the Operating system, ROS, CUDA, ZED Stereo Camera SDK, implementation of a backup system for the OS Snapshot. Setting up the Zotac computer used for the husky and installation of OpenCV, CUDA and ZED SDK Drivers. I worked with Parv on the website.

After the successful manual altitude hold flight test, the next target was to do the same task autonomously, so I wrote the code for auto take off and land, extensive ground testing with the code and then the final execution inside the cage. The final test was to arm the drone, take-off to a height of 1.5m, hover for a time of 10 seconds and then land and disarm the drone.

I am also the team purchaser for PhoeniX so I performed tasks on managing the orders, inventory for the team and making sure that the components fall in the right place so that we can get them whenever we want those.

Challenges:

During the course of manual flights, we had a lot of challenges with IMU, Mag, etc. calibration as the weather outside was not suitable, since we had built a custom drone the gains had to be tuned and which resulted into some crashes with the drone sticking up the ceiling of the cage, landing gear parts required replacements.

We faced a lot of issues with CUDA installations and we had to re-flash the boards several times so I came up with a shell script which automated the process and installs CUDA into most machines successfully. The issue was with CUDA removing some dependencies on the graphics display driver and it is not a very well solved problem on the internet. Since 3 of the team members were working on the same computer and installations caused package dependencies to clash and we faced an incident of our OS on Jetson getting corrupt and thus we had to go through the whole setup process again so I developed a backup mechanism of the OS and thus from then we have been using the tool to store the backup after some significant work being done/stored on the computer.

I spent a lot of time with Shubham in debugging the issues with the stability of the drone, did log analysis and flight tests with me as the pilot. Since the initial approach was to use the tilted hexacopter approach but the platform was inherently unstable (due to the size of the drone and the gains not being tuned properly by AirLab) and by speaking to our sponsors we moved onto the normal hexacopter configuration and performed flight tests with that. I did our first successful altitude hold flight test with ensuring that the drone does not lose the altitude and maintains the Z co-ordinate to near perfection despite X-Y drift.

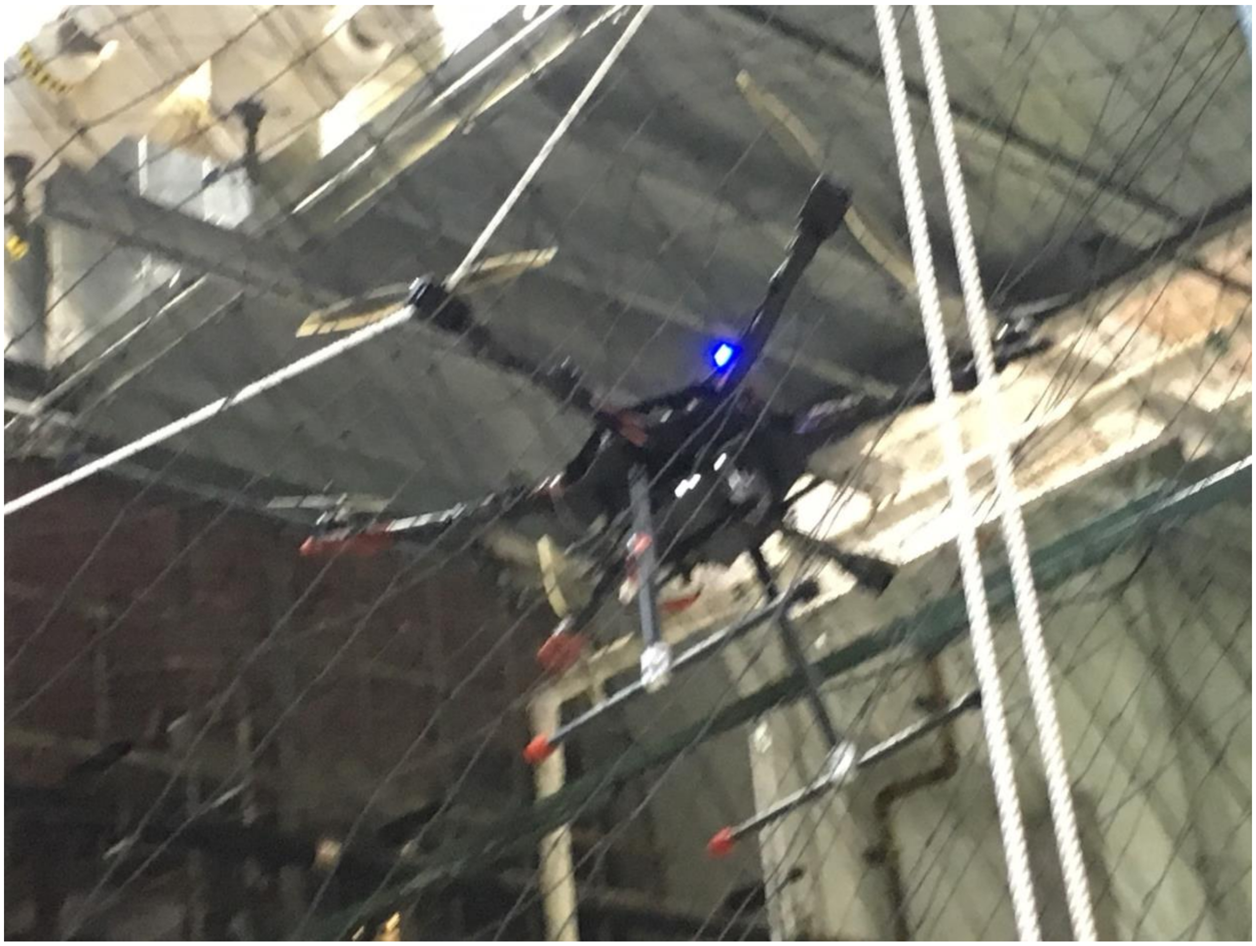
Figure 5: PhoeniX UAV stuck to the cage net

Challenges:

Sensors and Motor Control Lab:

The major challenge for me was to implement the PID controller as I had to deal with a substantially large dead band in PWM, soldering of the PCB for the motor driver in which I made some errors which required significant time in debugging. Mainly the error which I made was in shorting 2 pads which I got to know when all the LEDs on the board began to glow.

Understanding of the encoder was tough and to get the interrupts to fire up and record the actual values rather than the false triggers required me to implement an approach to read the low-level registers which demanded a good amount of time for understanding of the underlying concept and I referred Arduino website to get more information on the ISR working along with reading the low level registers.

The PID gains were very tough to tune as if I overshoot the position the motor will go crazy and The integral term also did not have any sort of saturation built in so it, so if the output of the controller was in the dead-band the motor would accumulate the integral error and it would just shoot the motor output. The other challenge was to control the PWM of the motor output, having a higher PWM control output caused the motor to overshoot and move very fast and thus I capped the maximum value to solve this issue.

The control for velocity was trickier as the velocity needed to be controlled in both the directions and it put my encoder ISRs to a test and with tuning the encoder code with some flags I could sense the rotation direction more precisely namely to get information if it were rotating clockwise or counterclockwise. The control output was a bit jittery as when the motor reached the desired velocity the control would drop and this would result in a drop in the motor velocity, thus to maintain the speed the control signal in the next timestep would be higher and thus I saw a graph like hills and sudden valleys, I tried to get some filtered output by playing with the gains but it required a lot of fine tuning to get it to the right spot as we could see it in the results from the graph above but it is still not very flat as the output for controlDegrees.

Team Work

Sensors and Motor Control Lab

Shubham: Implemented the Ultrasonic sensor and servo motor
Parv: Implemented the GUI and the code integration
Zhihao: Implemented the force sensor and stepper motor

Plans for PhoeniX

We have set an internal deadline to complete the task of writing ROS nodes to control the UAV and UGV using ROS bond and action server, which is a stricter implementation of ROS nodes as it requires development of safety protocol and nodes which help us with preventing fatalities and damage to the robots, thus saving us significant time in maintenance.

Completion of this task requires us to extensively test this hypothesis and once it is complete, we will start with the vision pipeline for visual servoing.

Specifically, my task is to work on the UAV to get it off ground autonomously and follow a marker to maintain it’s position in space.

Task 4 (Sensors and Motor Control Lab) Quiz

1. Reading a datasheet. Refer to the ADXL335 accelerometer datasheet (https://www.sparkfun.com/datasheets/Components/SMD/adxl335.pdf) to answer the below questions.
   - What is the sensor's range? +- 3.6g Typical and minimum +- 3g
   - What is the sensor's dynamic range? 6 g minimum and typical 7.2 g
   - 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?
     For most applications, a single 0.1 µF capacitor, CDC, placed close to the ADXL335 supply pins adequately decouples the accelerometer from noise on the power supply. It basically acts a filter to remove the sudden spikes in the power supply.
   - Write an equation for the sensor's transfer function.

     Vout = 1.5V + (300 mV/g)*a
   - What is the largest expected nonlinearity error in g?

     Nonlinearity (%) = Max deviation in input / Max full scale input
     Thus Max deviation = max error in g = (0.3 * 3.6)/100 = +-0.0108g
   - How much noise do you expect in the X- and Y-axis sensor signals when the sensor is excited at 25 Hz?

     Noise Density XOUT, YOUT = 150 µ*g*/√Hz rms
     Bandwidth XOUT, YOUT = 1600 Hz
     So here we are exciting the sensor at 25 Hz, meaning the Noise Density = 750 * sqrt(1.6) = 948.68 µ*g rms*
   - How about at 0 Hz? If you can't get this from the datasheet, how would you determine it experimentally?
     To do it experimentally we can keep it on a stationary object, since we know it's not moving we can get the readings from the sensor and see the error as the actual values are supposed to be 0 in all directions so we can now measure the error.

2. Signal conditioning
   - Filtering
     - What problem(s) might you have in applying a moving average?
       While implementing the moving average filter, the mean would get carried away with the outliers in sensor read values and would just shift the moving average in the direction to the outlier which won't provide good smoothened effect
     - What problem(s) might you have in applying a median filter?
       Calculating median is a computationally expensive task and might just affect the computation performance of the filter.
   - Opamps

- 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: 1) which of V1 and V2 will be the input voltage and which the reference voltage; 2) the value of the reference voltage; and 3) the value of Rf/Ri in each case. If the calibration can't be done with this circuit, explain why.
  - 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).
    Case 1: Let's take V1 as input and V2 as reference:
    V1 = -1.5 and Vout = 0
    The equation used is:
    Vout = (1 + Rf/Ri)*V2 – Rf/Ri*V1
    0 = (1 + Rf/Ri)*V2 +1.5Rf/Ri -------- Equation 1
    When V1 = 1.0V and Vout = 5.0V
    5 = (1+Rf/Ri)*V2 – Rf/Ri*1. -------- Equation 2
    Hence solving the equations gives Rf/Ri = -2 and that's not possible.

    Case 2: V2 as input and V1 as reference
    Equation 1 = 0 = (1+rf/ri)*(-1.5) – (rf/ri)*V1
    Equation 2 = 5 = (1+rf/ri)*(1) – (rf/ri)*V1
    The equations give that V1 = -3V and Rf/Ri = 1. Thus, V2 = input, V1 = Reference = -3V and Rf/Ri = 1
  - 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).
    Case 1: V1 = input and V2 = Reference
    Equation 1: 0 = (1+Rf/Ri)*V2 + 2.5*(Rf/Ri)
    Equation 2: 5 = (1+Rf/Ri)*V2 -2.5*(Rf/Ri)
    Rf/Ri = -1, which is not possible

    Case 2:
    Equation 1: 0 = (1+Rf/Ri)*-2.5 - V1*(Rf/Ri)
    Equation 2: 5 = (1+Rf/Ri)*2.5 – V1*(Rf/Ri)
    Rf/Ri = -1 ; implies that this is not possible as ratio of 2 resistors can't be negative

    Thus there is no solution for this case

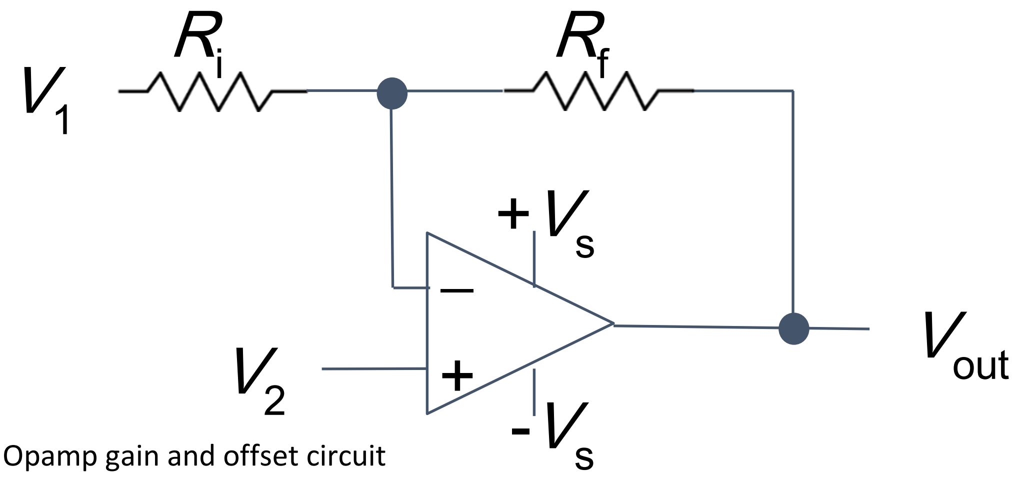

Fig. 1 Opamp gain and offset circuit

3. Control
   - 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.

     The first step is to calculate the error at the current time step which will be the difference between the setpoint and the currently sensed value. This is fed as the input to the P term.
     Let's call the input to the I term as errorSum. The input of the I term is computed by summing up the previous value of errorSum and the current error*delta_t; where delta_t is the difference in time between the previous timestep and the current time step.
     Let's call the input to the D term as errorDot; which can be computed by (error – previouserror)/delta_t. Just to make things clear we need to store the error at each time step so that it can be used in the subsequent time steps.
   - If the system you want to control is sluggish, which PID term(s) will you use and why?

     I would use the P term because since the system is slow to respond despite the error between the setpoint and the currently sensed values is high which means that the P term is low and increasing it will eliminate some of the sluggishness.
   - 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 I term for the steady state error. The steady state error is removed by the I term as it accumulates the error over time and then applies the Ki which will try to fix the steady state error.
   - After applying the control in the previous question, if the system still has overshoot, which PID term(s) will you apply and why?

To remove the overshoot and possible oscillations we can use the D parameter along with the P term. The D term acts as a damper as we see in the mass-spring damper system and will try to remove the overshoot and oscillations and bring the system to the desired state.