Individual Lab report

Lunar ROADSTER

Team I

Author: **Ayapilla Sri Bhaswanth**
Andrew ID: bayapill
E-mail: *bayapill@andrew.cmu.edu*

Supervisor: **Dr. William "Red" Whittaker**
Department: Field Robotics Center
E-mail: *red@cmu.edu*

February 7, 2025

Carnegie Mellon University
Robotics Institute

Contents

1 Individual Progress

1.1 Sensors and Motors Lab

My responsibility for this lab was to develop the circuit and write code for the MB1000 LV-MaxSonar-EZ0 Ultrasonic Range Finder, implementing a mean filter and transfer function, and also to implement a pushbutton to control the servo motor.

1.1.1 Ultrasonic Range Finder

The Ultrasonic sensor has 7 pins, but I used only the following 3 pins:

- **Pin 3-AN:** Outputs analog voltage with a scaling factor of (Vcc/512) per inch. A supply of $5V$ yields $\sim 9.8mV$/in. and $3.3V$ yields $\sim 6.4mV$/in. The output is buffered and corresponds to the most recent range data.
- **Pin 6-+5V:** Vcc – Operates on $2.5V - 5.5V$. Recommended current capability of $3mA$ for $5V$, and $2mA$ for $3V$.
- **Pin 7-GND:** Return for the DC power supply

The sensor gives a range between 6-inches ($\sim 15.24cm$) to 20-inches ($\sim 645.16cm$), with 1-inch resolution ($\sim 2.54cm$).

The above information is taken from the datasheet.

I utilized the analog pin of the ultrasonic sensor and connected it to an analog pin on the Arduino board. Doing a simple `analogRead` gives me the analog value, which can be converted into a physically interpretable distance value using the following transfer function:

$$\text{distance} = \text{analogVal} * 0.498 * 2.54$$

The above transfer function returns the distance in centimeters and was obtained by conducting experiments myself.

To get more stable readings, I decided to use a mean filter. It's a simple and fast way to smooth out the data since it doesn't require sorting like some other filtering methods..

```
int Dialog::filterMean(std::vector<int>& arr) {
    if(arr.empty()) {
        return 0;
    }

    int sum = 0;

    for (size_t i = 1; i < arr.size(); i++) {
        sum += arr[i];
    }

    return sum / arr.size();
}
```

1.1.2 Pushbutton

I implemented a pushbutton for servo motor control. It essentially acts as an interrupt, overwriting all other functions to increment the servo angle by 30 degrees every time it is pressed. Once the servo reaches its maximum limit of 180 degrees, pressing the pushbutton again will move the servo to its 0 degree position.

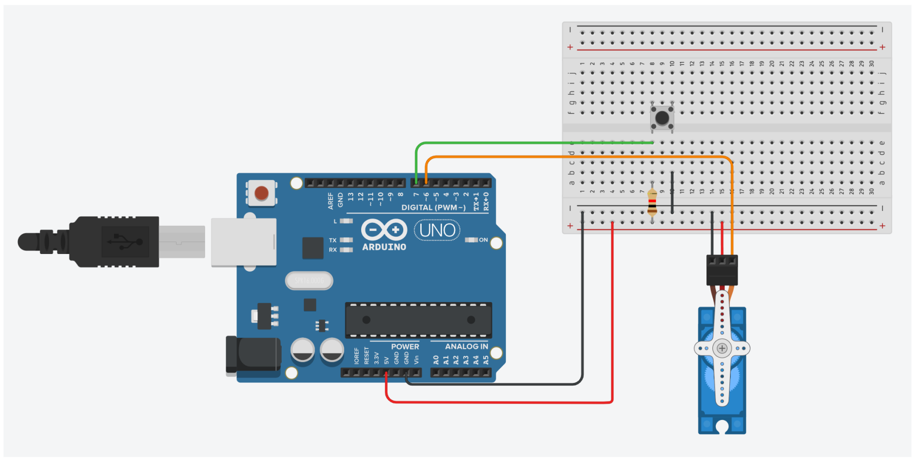


Figure 1: Pushbutton for Servo Control

```
void setup(){
   // Push Button
  attachInterrupt(digitalPinToInterrupt(PushbuttonPin), pushISR, FALLING);
}

void pushISR() {
  unsigned long currentTime = millis();
  if ((currentTime - lastDebounceTime) > debounceDelay) {
    servoAngle += incrementServo;
    if (servoAngle > 180) {
      servoAngle = 0;
    }
    lastDebounceTime = currentTime;
    servo.write(servoAngle);
  }
}
```

1.1.3 Full Circuit

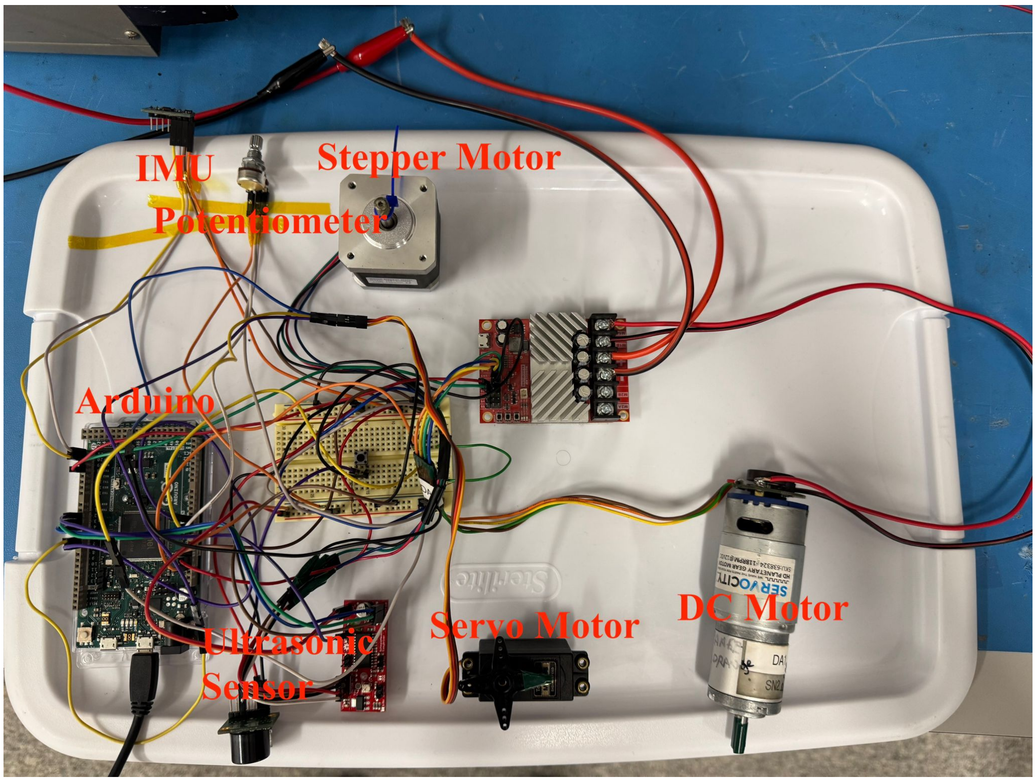


Figure 2: Sensors & Motor Control Lab Physical Setup

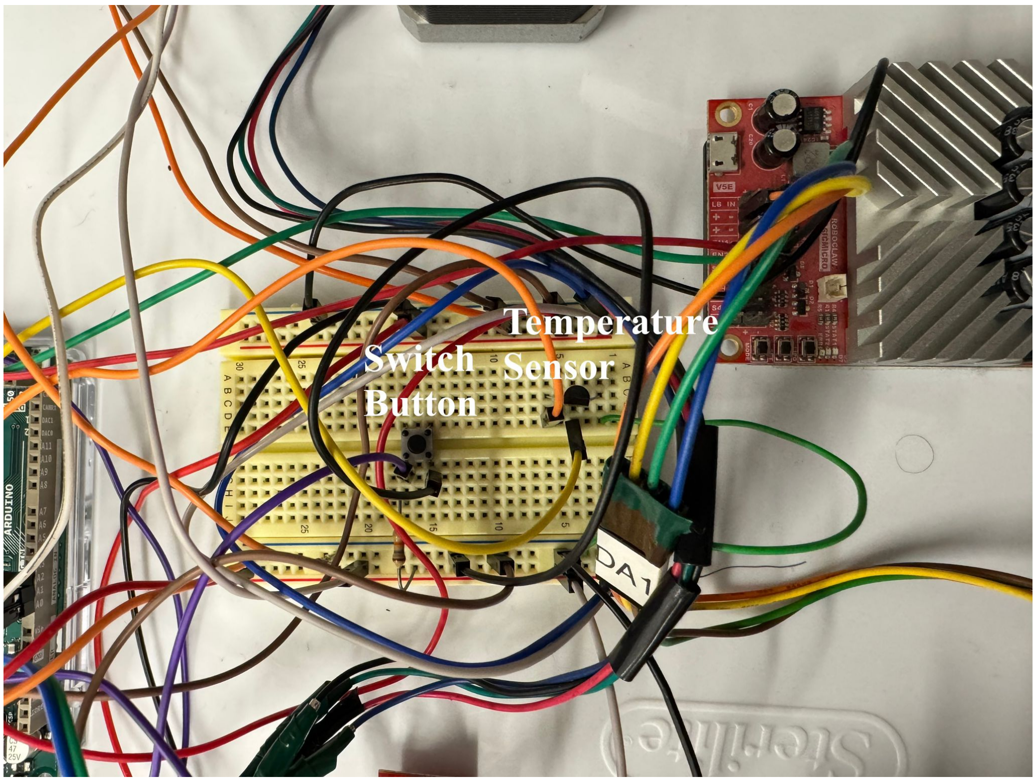
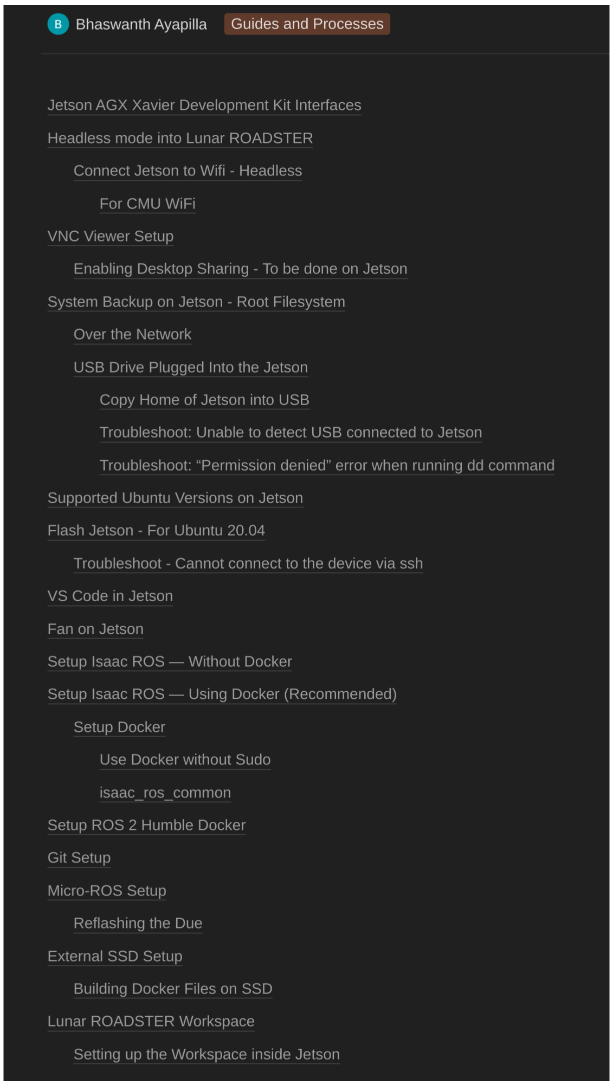


Figure 3: Close-up of Breadboard Setup

1.2 MRSD Capstone Project: Lunar ROADSTER

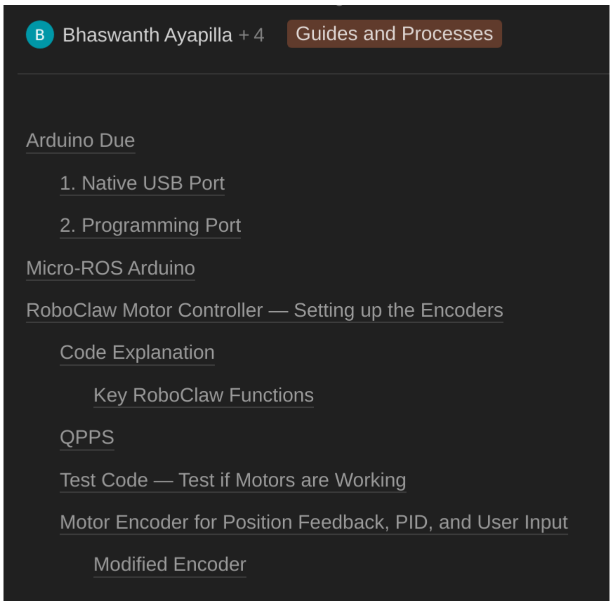
Software: Our team began working on the project over the winter break, and my initial focus was to set up the software stack. I started with the Jetson AGX Xavier, which will serve as the core of our system and runs the entire software stack. Since we are building on the work of the previous MRSD team, Crater Grader, the Jetson we got from them was initially flashed with Ubuntu 18.04. However, being an outdated version, we decided to upgrade the entire software, including transitioning from ROS2 Galactic to ROS2 Humble using Docker. To facilitate this, I re-flashed the Jetson with Ubuntu 20.04, configured VNC for remote access, and fully set up Docker for easier deployment. I also helped set up of the operations terminal (our main workstation) and conducted teaching sessions for my team on running and using the Jetson, and using Docker as well.



- Jetson AGX Xavier Development Kit Interfaces
- Headless mode into Lunar ROADSTER
  - Connect Jetson to Wifi - Headless
    - For CMU WiFi
- VNC Viewer Setup
  - Enabling Desktop Sharing - To be done on Jetson
- System Backup on Jetson - Root Filesystem
  - Over the Network
  - USB Drive Plugged Into the Jetson
    - Copy Home of Jetson into USB
    - Troubleshoot: Unable to detect USB connected to Jetson
    - Troubleshoot: “Permission denied” error when running dd command
- Supported Ubuntu Versions on Jetson
- Flash Jetson - For Ubuntu 20.04
  - Troubleshoot - Cannot connect to the device via ssh
- VS Code in Jetson
- Fan on Jetson
- Setup Isaac ROS — Without Docker
- Setup Isaac ROS — Using Docker (Recommended)
  - Setup Docker
    - Use Docker without Sudo
    - isaac_ros_common
- Setup ROS 2 Humble Docker
- Git Setup
- Micro-ROS Setup
  - Reflashing the Due
- External SSD Setup
  - Building Docker Files on SSD
- Lunar ROADSTER Workspace
  - Setting up the Workspace inside Jetson

Figure 4: Documentation of Jetson AGX Xavier Setup

Sensors: My next task was setting up the motor encoders. The rover is equipped with 4 motors — two drive motors and two steer motors, both in the front and rear. All motors are controlled by the Roboclaw motor controllers, and I interfaced them with an Arduino Due. To enable communication between the motors and the Jetson AGX Xavier, I configured Micro-ROS on the Arduino. This will allow the system to send drive commands to the motors and also receive encoder data and other feedback by publishing and subscribing to relevant ROS topics. Following this, I set up the ZED 2i depth camera with the help of my teammate William, which we will use for active mapping and validating the grading done by our rover.



Bhaswanth Ayapilla + 4 Guides and Processes

Arduino Due

1. Native USB Port

2. Programming Port

Micro-ROS Arduino

RoboClaw Motor Controller — Setting up the Encoders

Code Explanation

Key RoboClaw Functions

QPPS

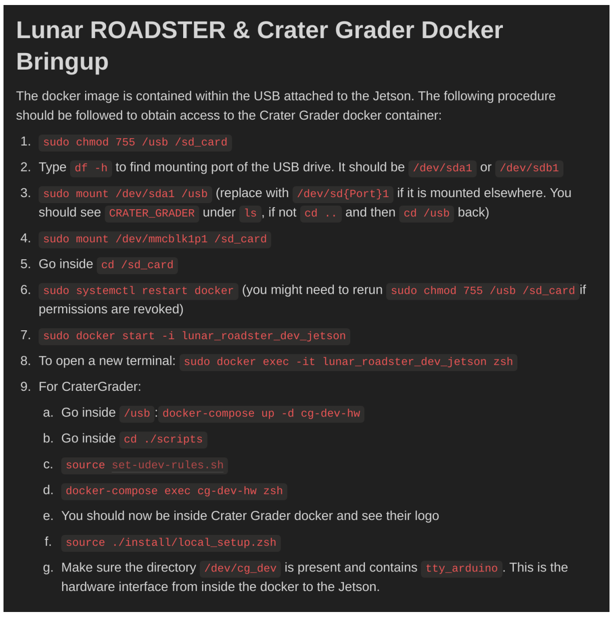
Test Code — Test if Motors are Working

Motor Encoder for Position Feedback, PID, and User Input

Modified Encoder

Figure 5: Documentation of Arduino Due, Micro-ROS, Encoder Setup

Teleoperation: Since we had Crater Grader's entire software, a big initial step was to get the rover running and teleoperate it with a joystick. Me and William worked on this during the winter break and successfully got it running. Because of this, the team is now able to carry out all the necessary subsequent tests on the Moon Yard.



Lunar ROADSTER & Crater Grader Docker Bringup

The docker image is contained within the USB attached to the Jetson. The following procedure should be followed to obtain access to the Crater Grader docker container:

1. `sudo chmod 755 /usb /sd_card`
2. Type `df -h` to find mounting port of the USB drive. It should be `/dev/sda1` or `/dev/sdb1`
3. `sudo mount /dev/sda1 /usb` (replace with `/dev/sd{Port}1` if it is mounted elsewhere. You should see `CRATER_GRADER` under `ls`, if not `cd ..` and then `cd /usb` back)
4. `sudo mount /dev/mmcblk1p1 /sd_card`
5. Go inside `cd /sd_card`
6. `sudo systemctl restart docker` (you might need to rerun `sudo chmod 755 /usb /sd_card` if permissions are revoked)
7. `sudo docker start -i lunar_roadster_dev_jetson`
8. To open a new terminal: `sudo docker exec -it lunar_roadster_dev_jetson zsh`
9. For CraterGrader:
   a. Go inside `/usb`: `docker-compose up -d cg-dev-hw`
   b. Go inside `cd ./scripts`
   c. `source set-udev-rules.sh`
   d. `docker-compose exec cg-dev-hw zsh`
   e. You should now be inside Crater Grader docker and see their logo
   f. `source ./install/local_setup.zsh`
   g. Make sure the directory `/dev/cg_dev` is present and contains `tty_arduino`. This is the hardware interface from inside the docker to the Jetson.

Figure 6: Documentation of Rover Bringup and Teleoperation

Equipment: In the first week of January, our team underwent a training for the FARO 3D Scanner with Wennie Tabib from FRC. This training was essential as we will use the scanner's output to generate maps for the rover's navigation and to evaluate the performance of the grading operation. After the training, we tested out the scanner and successfully mapped the Moon Yard at the Planetary Robotics Lab. Over the winter break, the team also received training from Warren "Chuck" Whittaker to use the TS16 Leica-Geosystems Robotic Total Station. We will use this total station to track a prism mounted on the rover and ultimately localize it.

2 Challenges

2.1 Sensors and Motors Lab

The biggest challenge I faced in the sensors and motors lab was implementing the filter. Initially, I used a median/mode filter, which required a sorting algorithm like bubble sort. However, running the filter alongside sorting took too much time, causing delays that blocked other sensor and motor functions. Instead, I switched to a mean filter, which provided faster processing without the need for sorting.

2.2 MRSD Capstone Project: Lunar ROADSTER

Setting up Docker on the Jetson AGX Xavier was a significant challenge for me. I was fairly new to Docker and it was a pretty big learning curve for me as I had to write the dockerfiles and scripts myself. Another major challenge was teleoperating the rover using the existing software stack. I spent a considerable amount of time trying to understand the general architecture and code of Crater Grader. Additionally, I faced issues in running the ZED camera's SDK on the Jetson, despite building the relevant docker image. Ultimately, I just decided to use the camera without the SDK, and use OpenCV and other relevant libraries to get our job done.

3 Teamwork

A breakdown of the contributions of each team member are tabulated below:

3.1 Sensors and Motors Lab

- **Bhaswanth Ayapilla:** Interfacing Ultrasonic Range Finder, implementing mean filter and transfer function, implementing pushbutton for servo control, README file for the complete code.
- **Ankit Aggarwal:** Implemented the stepper motor speed and direction controller using a potentiometer.
- **Deepam Ameria:** Interfacing IMU sensor and servo motor controller.
- **Simson D'Souza:** Implemented DC motor control with encoder feedback and interfaced an IR sensor for distance measurement.
- **Boxiang (William) Fu:** GUI development and Arduino template code.

3.2 MRSD Capstone Project: Lunar ROADSTER

- **Bhaswanth Ayapilla:** NVIDIA Jetson setup, setup encoder drivers, setting up teleoperation (in collaboration with William), ZED Camera setup (in collaboration with William), setting up operations terminal, FARO scanner setup and Moon Pit scanning (in collaboration with team).
- **Ankit Aggarwal:** Wheel design and printing, rover hardware setup/maintenance, circuit diagram design (in collaboration with Simson), VectorNav IMU interfacing, preliminary testing (in collaboration with team), project manager.
- **Deepam Ameria:** Prototype dozer development (in collaboration with Simson), preliminary tests for teleoperation and grading with prototype dozer (in collaberation with Simson), dozer blade and mechanism design, FARO scanner setup and Moon Pit scanning (in collaboration with team).
- **Simson D'Souza:** Prototype dozer development (in collaberation with Deepam), preliminary tests for teleoperation and grading with prototype dozer (in collaberation with Deepam), circuit diagram design (in collaboration with Ankit), FARO scanner setup and Moon Pit scanning (in collaboration with team), processing of point cloud data to generate an occupancy grid map for navigation.
- **Boxiang (William) Fu:** Moon Pit Crater Distribution, LAN Setup, setting up teleop (in collaboration with Bhaswanth), ZED camera setup (in collab with Bhaswanth)

4 Plans

4.1 Sensors and Motors Lab

Working on the sensors and motors lab provided valuable insight into sensor integration and potential challenges we might encounter. The knowledge gained will certainly benefit our capstone project. However, since this assignment is not directly related to our project, we have no further plans for it. As part of the lab, we used the RoboClaw motor driver originally installed on our rover to control the DC motor, which we will reinstall after the demonstration. Similarly, the Arduino Due will be returned to the rover setup.

4.2 MRSD Capstone Project: Lunar ROADSTER

Our team is working diligently to meet out first internal milestone on 12^{th} February. By this deadline, we aim to complete all hardware-related tasks, including manufacturing the excavator blade, iterating on wheel designs, and performing quality assurance for the mechanical subsystem. Before the deadline, we also aim to complete the localization task and simulating the rover's navigation. Once these are completed, we will begin working on the excavator planner and the FSM planner, transitioning navigation from simulation to real-world deployment, and implementing validation using the depth camera. My current focus is on the localization of the rover. Me and William will set up the total station in the Moon Yard and use the data obtained from it and also an onboard IMU to localize the rover. Following this, I will be working on navigation and validation along with Simson. Once all team members complete their individual tasks, we will begin rigorous testing of the complete rover operation on the Moon Yard.

5 Code

Below is the Arduino code for the Sensors and Motors Lab:

```
#include <Adafruit_Sensor.h>
#include <Adafruit_MPU6050.h>
#include <Wire.h>
#include <Servo.h>
#include <SharpIR.h>
#include <RoboClaw.h>
#include <AccelStepper.h>

#define SERIAL_PORT Serial

#define ADDRESS 0x80

// Motor & Encoder parameters
#define ENCODER_TICKS_PER_REV 3416
#define DEGREE_TO_TICKS (ENCODER_TICKS_PER_REV / 360.0)

// Position PID Tuning
#define Kp_pos 2.0
#define Ki_pos 0.5
#define Kd_pos 1.0

// Velocity PID Tuning
#define Kp_vel 1.0
#define Ki_vel 0.5
#define Kd_vel 0.25

// Maximum speed in encoder counts per second
#define MAX_QPPS 6718

// Pins
#define EN_StepperDriver 2
#define Stp_StepperDriver 3
#define Dir_StepperDriver 4
#define servoPin 6
#define PushButtonPin 7
#define PotentiometerPin A0
#define temperaturePin A1
#define ultrasonicPin A2
#define PushbuttonPin 7

AccelStepper stepper(AccelStepper::DRIVER, Stp_StepperDriver, Dir_StepperDriver);


// GLOBAL VARIABLES
// Note: GUI will execute commands sent by Arduino once every 2 intervals as it disca
unsigned long previousMillis = 0;
const unsigned long interval = 100;
```

```
Adafruit_MPU6050 mpu;
Servo servo;
RoboClaw roboclaw(&Serial1, 10000);

int servoAngle = 0;
int dc_motor_speed = 0;
int dc_motor_angle = 0;
double cm = 0.0;

int PotControlFlag = 0;
volatile int PotVal = 0;
volatile int globalStepperValue = 0;
volatile int globalStepperAngle = 0;

// Debounce variables
volatile unsigned long lastDebounceTime = 0;
const unsigned long debounceDelay = 100;
const int incrementServo = 30;

bool isMoving = false;
uint32_t targetPosition = 0;

void setup() {
  SERIAL_PORT.begin(9600);
  Serial1.begin(38400);
  while (!SERIAL_PORT) {
    // Wait for the serial port to be ready
  }

  // IMU
  if (!mpu.begin()) {
    Serial.println("Failed to find MPU6050 chip");
    while (1)
      ;
  }

  // Servo Motor
  servo.attach(servoPin);
  servo.write(0);

  // Push Button
  attachInterrupt(digitalPinToInterrupt(PushbuttonPin), pushISR, FALLING);

  // DC Motor
  unsigned long startTime = millis();

  // Stepper
  pinMode(EN_StepperDriver, OUTPUT);
  digitalWrite(EN_StepperDriver, LOW);  // enable stepper(s)
  stepper.setMaxSpeed(2000);
  stepper.setAcceleration(1000);
  stepper.setSpeed(0);

  // // Initialize RoboClaw
  // if (!roboclaw.ReadError(ADDRESS)) {
  //   Serial.println("RoboClaw connected successfully.");
  // } else {
  //   Serial.println("Error detected in RoboClaw!");
  // }

  roboclaw.SetM1VelocityPID(ADDRESS, Kp_vel, Ki_vel, Kd_vel, MAX_QPPS);
  roboclaw.SetM1PositionPID(ADDRESS, Kp_pos, Ki_pos, Kd_pos, Kp_vel, Ki_vel, Kd_vel, 

  roboclaw.SpeedM1(ADDRESS, 0);
  // Serial.println("Motor stopped at startup.");

  SERIAL_PORT.print("Arduino Due Serial is ready!;");
}


void loop() {
  if (isMoving) {
    uint32_t currentPos = roboclaw.ReadEncM1(ADDRESS);
    if (abs((int32_t)(currentPos - targetPosition)) <= 15) {  // Position tolerance
      roboclaw.SpeedM1(ADDRESS, 0);                           // Stop motor
      // Serial.println("Target position reached.");
      isMoving = false;  // Reset movement flag
    }
  }

  // Check if data is available to read from the serial port
  if (SERIAL_PORT.available() > 0) {
    // Read the incoming string
    String receivedString = SERIAL_PORT.readStringUntil(';');

    // // Echo the string back to the serial port
    // SERIAL_PORT.print("Arduino received command: ");
    // SERIAL_PORT.print(receivedString);
    // SERIAL_PORT.print(";");
    // SERIAL_PORT.flush();

    if (receivedString.length() > 1) {
      char commandType = receivedString.charAt(0);

      String valueString = receivedString.substring(1);
      if (isNumeric(valueString)) {
        int commandValue = valueString.toInt();
        handleCommand(commandType, commandValue);
      }
    }
  }

  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= interval) {
    previousMillis = currentMillis;
    timerCallback();
  }

  stepperCallback();
}

void timerCallback() {
  // SERIAL_PORT.print("Timer callback executed at: ");
  // SERIAL_PORT.print(previousMillis);
  // SERIAL_PORT.print(";");

  // A "$" is used to indicate the serial port return is a command
  String serialReturn;
  serialReturn.concat("$");

  int servoMotorState = servoMotorStateCallback();
  serialReturn.concat(servoMotorState);
  serialReturn.concat(",");

  int stepperMotorState = stepperMotorStateCallback();
  serialReturn.concat(stepperMotorState);
  serialReturn.concat(",");

  int velDCMotorState = velDCMotorStateCallback();
  serialReturn.concat(velDCMotorState);
  serialReturn.concat(",");

  int angleDCMotorState = angleDCMotorStateCallback();
  serialReturn.concat(angleDCMotorState);
  serialReturn.concat(",");

  int potentiometerSensorState = potentiometerSensorCallback();
  serialReturn.concat(potentiometerSensorState);
  serialReturn.concat(",");

  double imuSensorState = imuSensorCallback();
  serialReturn.concat(imuSensorState);
  serialReturn.concat(",");

  double temperatureSensorState = temperatureSensorCallback();
  serialReturn.concat(temperatureSensorState);
  serialReturn.concat(",");

  int ultrasonicSensorState = ultrasonicSensorCallback();
  serialReturn.concat(ultrasonicSensorState);
  serialReturn.concat(",");

  double electricalInput = analogRead(ultrasonicPin);
  double transferFunctionState = transferFunctionCallback(electricalInput);
  double electricalVoltage = analogRead(ultrasonicPin) * (5.0 / 1023.0);

  serialReturn.concat(electricalVoltage);
  serialReturn.concat(":");
  serialReturn.concat(transferFunctionState);

  SERIAL_PORT.print(serialReturn);
  SERIAL_PORT.print(";");
}


void handleCommand(char commandType, int value) {
  switch (commandType) {
    case 'R':
      servoMotorController(value);
      break;
    case 'S':
      stepperMotorController(value);
      break;
    case 'V':
      velDCMotorController(value);
      break;
    case 'A':
      angleDCMotorController(value);
      break;
    case 'B':
      buttonStepperMotorController(value);
      break;
    default:
      break;
  }
}


bool isNumeric(String str) {
  if (str.length() == 0) return false;

  int startIndex = 0;

  if (str[0] == '-') {
    if (str.length() == 1) return false;
    startIndex = 1;
  }

  for (unsigned int i = startIndex; i < str.length(); i++) {
    if (!isDigit(str[i])) {
      return false;
    }
  }

  return true;
}


// CONTROLLER FUNCTIONS

void servoMotorController(int control) {
  /*
  INPUT: Integer in min/max range of 0 to 180 corresponding to desired angle
  OUTPUT: Void
  */

  SERIAL_PORT.print("Servo motor controller received command: ");
  SERIAL_PORT.print(control);
  SERIAL_PORT.print(";");

  // TODO: IMPLEMENT FUNCTION BELOW
  servoAngle = control;
  servo.write(servoAngle);
}

void stepperMotorController(int control) {
  /*
  INPUT: Integer in min/max range of -180 to 180 corresponding to desired angle
  OUTPUT: Void
  */

  SERIAL_PORT.print("Stepper motor controller received command: ");
  SERIAL_PORT.print(control);
  SERIAL_PORT.print(";");

  // TODO: IMPLEMENT FUNCTION BELOW
  if (PotControlFlag == 0) {
    globalStepperValue = map(control, -180, 180, -1600, 1600);
    globalStepperAngle = control;
  }
}

void velDCMotorController(int control) {
  /*
  INPUT: Integer in min/max range of -118 to 118 corresponding to desired RPM
  OUTPUT: Void
  */

  SERIAL_PORT.print("Velocity DC motor controller received command: ");
  SERIAL_PORT.print(control);
  SERIAL_PORT.print(";");

  // TODO: IMPLEMENT FUNCTION BELOW
  // Set motor velocity (positive for forward, negative for reverse)
  int dc_motor_speed = (control / 118.0) * 100.0;
  int encoderSpeed = (dc_motor_speed * MAX_QPPS) / 100;  // Scale input speed (user en
  roboclaw.SpeedM1(ADDRESS, encoderSpeed);
}

void angleDCMotorController(int control) {
  /*
  INPUT: Integer in min/max range of -360 to 360 corresponding to desired angle
  OUTPUT: Void
  */

  // SERIAL_PORT.print("Angle DC motor controller received command: ");
  // SERIAL_PORT.print(control);
  // SERIAL_PORT.print(";");

  // TODO: IMPLEMENT FUNCTION BELOW
  uint32_t currentPos = roboclaw.ReadEncM1(ADDRESS);
  targetPosition = currentPos + (control * DEGREE_TO_TICKS);

  // Serial.print("Moving motor to position: ");
  // Serial.println(targetPosition);

  roboclaw.SpeedAccelDeccelPositionM1(ADDRESS, 10000, MAX_QPPS, 10000, targetPosition

  isMoving = true;  // Set flag for movement tracking
}

void buttonStepperMotorController(int control) {
  /*
  INPUT: Boolean with 0 indicating GUI control and 1 indicating potentiometer control
  OUTPUT: Void
  */

  SERIAL_PORT.print("Button controller received command: ");
  SERIAL_PORT.print(control);
  SERIAL_PORT.print(";");

  // TODO: IMPLEMENT FUNCTION BELOW
  PotControlFlag = control;
}


// CALLBACK FUNCTIONS
int servoMotorStateCallback() {
  /*
  INPUT: Void
  OUTPUT: Integer in min/max range of 0 to 180 corresponding to servo motor angle
  */

  // TODO: IMPLEMENT FUNCTION BELOW
  return servoAngle;
}

int stepperMotorStateCallback() {
  /*
  INPUT: Void
  OUTPUT: Integer in min/max range of -180 to 180 corresponding to stepper motor angl
  */

  // TODO: IMPLEMENT FUNCTION BELOW

  return globalStepperAngle;
}

int velDCMotorStateCallback() {
  /*
  INPUT: Void
  OUTPUT: Integer in min/max range of -118 to 118 corresponding to DC motor RPM
  */

  // TODO: IMPLEMENT FUNCTION BELOW

  int speed = roboclaw.ReadSpeedM1(ADDRESS);
  speed = speed * 118 / 6718;

  return speed;
}

int angleDCMotorStateCallback() {
  /*
  INPUT: Void
  OUTPUT: Integer in min/max range of -360 to 360 corresponding to DC motor angle
  */

  // TODO: IMPLEMENT FUNCTION BELOW

  int enc = roboclaw.ReadEncM1(ADDRESS);
  int angle = (enc * 360) / 3416;
  angle = angle % 360;

  return angle;
}

int potentiometerSensorCallback() {
  /*
  INPUT: Void
  OUTPUT: Integer corresponding to potentiometer reading
  */

  // TODO: IMPLEMENT FUNCTION BELOW
  PotVal = analogRead(PotentiometerPin);

  if (PotControlFlag == 1) {
    globalStepperValue = map(PotVal, 0, 1022, -1600, 1600);
    globalStepperAngle = map(globalStepperValue, -1600, 1600, -180, 180);
  }

  return PotVal;
}

double imuSensorCallback() {
  /*
  INPUT: Void
  OUTPUT: Double corresponding to sensed IMU pitch reading
  */

  // TODO: IMPLEMENT FUNCTION BELOW
  sensors_event_t a, g, temp;
  mpu.getEvent(&a, &g, &temp);

  // Pitch using accel data
  double pitchAccel = atan2(a.acceleration.x, a.acceleration.z) * 180 / PI;

  return pitchAccel;
}

double temperatureSensorCallback() {
  /*
  INPUT: Void
  OUTPUT: Double corresponding to temperature reading (degree celsius)
  */

  // TODO: IMPLEMENT FUNCTION BELOW
  int reading = analogRead(temperaturePin);
  double voltage = reading * 3.3;
  voltage /= 1024.0;

  double temperatureC = (voltage - 0.5) * 100;

  return temperatureC;
}

int ultrasonicSensorCallback() {
  /*
  INPUT: Void
  OUTPUT: Integer corresponding to ultrasonic reading (cm)
  */

  // TODO: IMPLEMENT FUNCTION BELOW
  cm = analogRead(ultrasonicPin);
  cm = transferFunctionCallback(cm);

  return int(cm);
}

// OTHER FUNCTIONS
void pushISR() {
  /*
  INPUT: Void
  OUTPUT: Void
  */

  // TODO: IMPLEMENT FUNCTION BELOW
  unsigned long currentTime = millis();
  if ((currentTime - lastDebounceTime) > debounceDelay) {
    servoAngle += incrementServo;
    if (servoAngle > 180) {
      servoAngle = 0;
    }
    lastDebounceTime = currentTime;
    servo.write(servoAngle);
  }
}

double transferFunctionCallback(double electricalInput) {
  /*
  INPUT: Double corresponding to electrical input voltage
  OUTPUT: Double corresponding to ultrasonic reading (cm)
  */

  // TODO: IMPLEMENT FUNCTION BELOW
  double val = electricalInput * 0.498 * 2.54;

  return val;
}

void stepperCallback() {
  /*
  INPUT: Void
  OUTPUT: Void
  */
  // TODO: IMPLEMENT FUNCTION BELOW
  stepper.moveTo(globalStepperValue);
  stepper.run();
}
```

6 Sensors and Motor Control Lab Quiz

6.1 Question 1

What is the sensor's range?

The measurement range along each axis is $\pm 3g$.

What is the sensor's dynamic range?

Maximum dynamic range is $7.2g$ and the minimum dynamic range is $6g$.

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

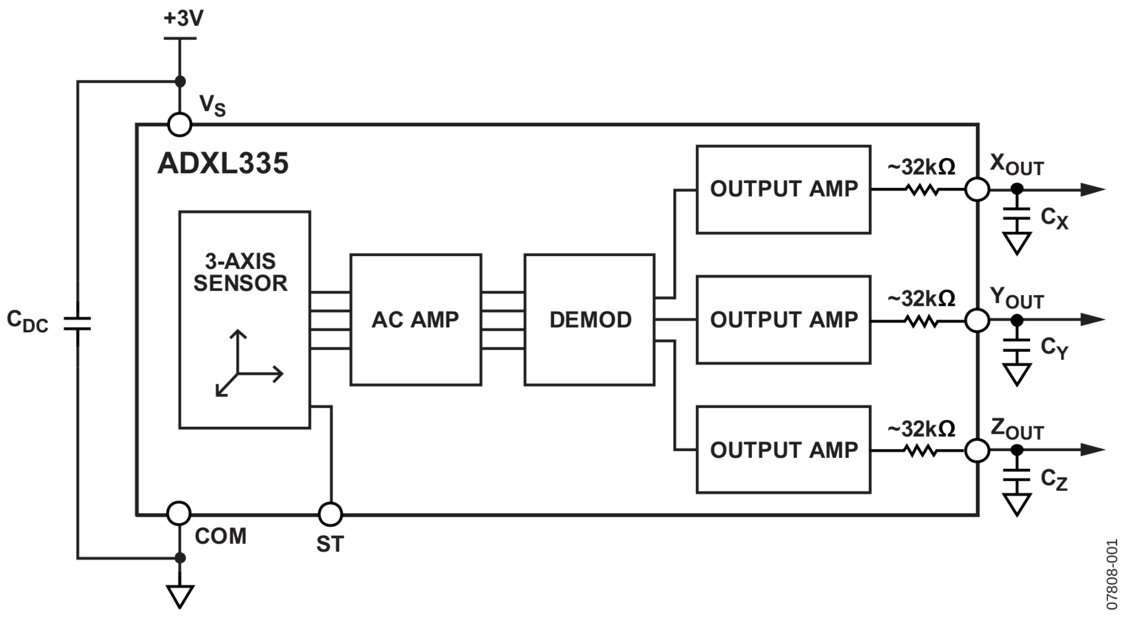


Figure 1.

Figure 7: Functional Block Diagram

The capacitor C_{DC} on the left side of the functional block diagram (Figure 1) serves as a decoupling capacitor. Its purpose is to filter out high-frequency noise from the power supply and stabilize the voltage supplied to the sensor. It does this by blocking sudden spikes or drops in voltage, making sure the sensor runs smoothly.

Write an equation for the sensor's transfer function.

General form of transfer function:

$$V_{out} = V_{zero-g} + S \cdot a$$

where

- V_{out} =Output voltage corresponding to the measured acceleration
- V_{zero-g} = Output voltage when no acceleration is applied (Typical value of $1.5V$

- S =Sensitivity of the sensor ($300mV/g$ at $3V$ supply)
- a =Applied acceleration in g

Using the above values, the transfer function is

$$V_{out} = 1.5V + 0.3V/g \cdot a$$

What is the largest expected nonlinearity error in g?

The nonlinearity error is typically $\pm 0.3\%$ of the full scale. Hence for a range of $\pm 3g$, we get

$$\text{Nonlinearity error} = 0.003 \times 3g = \pm 0.009g$$

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

Bandwidth along X and Y axes is typically 1600 Hz.

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

Typical noise of ADXL335 is determined by, as given on p.11 of the datasheet

$$rms\text{ Noise} = \text{Noise Density} \times (\sqrt{BW \times 1.6})$$

Noise density along X and Y axes is typically $= 150\mu g\ \sqrt{Hz}$ rms.

$$rms\text{ Noise} = 150\mu g\ \sqrt{Hz} \times \sqrt{1.6 \times 25} = 948.683\mu g$$

$$rms\text{ Noise} = 0.949mg$$

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.

Steps:

- Mount the accelerometer securely
- Collect raw output from the X and Y axes at a high sampling rate, ensuring no movement
- Calculate the mean (zero-g bias) and subtract it to isolate the noise component
- Apply a low-pass filter
- Calculate RMS noise by subtracting the mean from each measurement sample, taking the sum of their squares, and then taking the square root.
- Perform multiple trials and average results for accuracy

6.2 Question 2

Filtering:

Problems with a moving-average filter:

1. The averaging window causes a lag in the output, so sudden changes in the input appear late in the filtered signal
2. Sharp edges or fast changes in the signal can get lost because the filter blurs rapid transitions

Problems with a median filter:

1. Finding the median requires sorting, which can be more computationally expensive
2. While median filtering removes large spike, it can also remove small, real features if they fall outside the median too often

Operational Amplifiers:

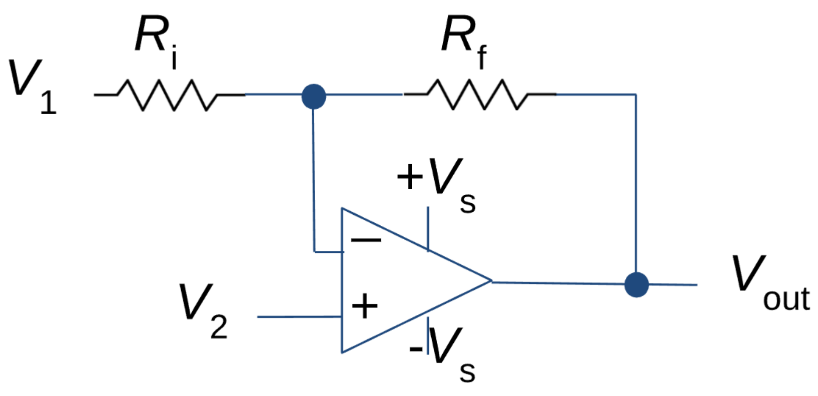


Figure 8: Opamp gain and offset circuit

From the figure above, we get the equation

$$V_{out} = (1 + G)V_2 - GV_1$$

where $G = \frac{R_f}{R_i}$.

1) Uncalibrated sensor has a range of $-1.5V$ to $1V$

Case 1: V_1 is the input voltage

On substituting this in the above equation, we get.

$$0 = (1 + G)V_2 + 1.5G$$
$$5 = (1 + G)V_2 - G$$

On solving this, we get $G = -2$, which is physically not possible.

Case 2: V_2 is the input voltage

$$0 = -1.5 - 1.5G - GV_1$$
$$5 = 1 + G - GV_1$$

On solving this, we get $G = 1$. Substituting this back, we get $V_1 = -3V$.

Hence for the uncalibrated sensor with range $-1.5V$ to $1V$, V_1 is the reference voltage, equal to $-3V$, and V_2 is the input voltage. The ratio $\frac{R_f}{R_i}$ is equal to 1.

2) Uncalibrated sensor has a range of $-2.5V$ to $2.5V$

Case 1: V_1 is the input voltage

On substituting this in the above equation, we get.

$$0 = (1 + G)V_2 + 2.5G$$
$$5 = (1 + G)V_2 - 2.5G$$

On solving this, we get $G = -1$, which is physically not possible.

Case 2: V_2 is the input voltage

$$0 = -2.5 - 2.5G - GV_1$$
$$5 = 2.5 + 2.5G - GV_1$$

On solving this, we get $G = 0$, which again, is not possible. Hence, calibration cannot be done with this circuit, because one operational amplifier alone is unable to produce the desired gain to amplify the input signal and also voltage shift it.

6.3 Question 3

To control a DC motor to its desired position using a PID controller, it has to be formulated in the following way.

For every discrete time k, we define the error as the difference between the desired position and the current encoder position.

$$e[k] = \text{desiredposition}[k] - \text{measuredposition}[k]$$

The proportional term is just the proportional gain K_p multiplied by the position error.

$$P[k] = K_p e[k]$$

The integral term accumulates the error over time. So for a sampling time period T_s and gain K_i, the integral term is given as

$$I[k] = I[k-1] + K_i T_s e[k]$$

The derivative term predicts where the error is heading to and shows how fast the error is changing. For a sampling time period T_s and gain K_d, the derivative term is given as

$$D[k] = K_d \frac{e[k] - e[k-1]}{T_s}$$

On combining all the terms, we get the control output that drives the motor as

$$u[k] = K_p e[k] + (I[k-1] + K_i T_s e[k]) + (K_d \frac{e[k] - e[k-1]}{T_s})$$

In case my system is too sluggish, I would play with the K_p values to speed up its response. Increasing the proportional K_p term will decrease the rise time, essentially giving a fast response but this can also cause more overshoot or oscillations.

Following this, in case my system has a significant steady-state error, I would increase the integral K_i term.

If the system still has an overshoot or oscillations, I would increase the derivative K_d term, as it causes damping.

In general, from what we learned from the Manipulation, Estimation, and Control course, we follow the steps below:

1. Start with $K_p = 0, K_i = 0, K_d = 0$
2. Turn up K_p until t_r reaches desired value
3. Slowly turn up K_i until $y_{ss} \rightarrow y_d$ in the desired time
4. Turn up K_d until t_s, M_p reach desired values
5. Refine the gains iteratively until desired transient performance is achieved