CG2111A Engineering Principles and Practice

Semester 2 2024/2025

"Alex to the Rescue"
Final Report
Team: B02-3A

Name	Student #	Main Role
Cui Jiahao	A0311801X	Software Lead
John Kenneth Layba Agpaoa	A0308049A	Hardware Assembly
Foo Kang	A0307913E	Software Development
Chong Kai Jie	A0306749U	Hardware Assembly

Section 1 Introduction

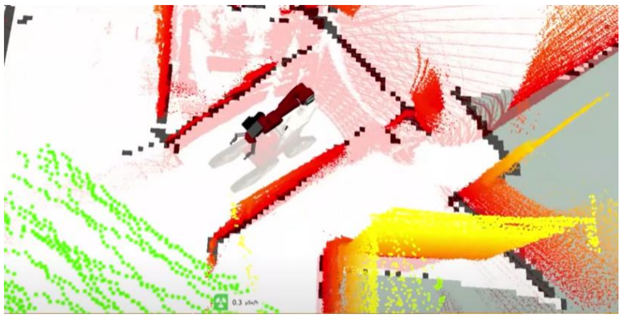
The “Search and Rescue” problem to be addressed by Alex takes place in a simulated lunar environment representing the aftermath of an explosion at Moonbase CEG. The environment contains obstacles and astronauts who require medical aid. Due to limited battery life and operational duration, Alex is tasked to efficiently navigate the terrain to locate and rescue the astronauts within an 8-minute time frame.

The astronauts are categorised into Green – require minimal medical attention and only require a medical package, or Red – in critical condition and require immediate evacuation for further treatment. Alex must explore the terrain, identify the astronauts correctly, and perform the necessary rescue action accordingly.

To accomplish this, Alex is equipped with LIDAR, allowing it to navigate and map an unknown environment. Alex will be teleoperated from our laptop, and navigation will mainly be done using the generated map. It is also equipped with a colour sensor for identifying the astronaut’s colour and a robotic arm to deliver medical packages to Green astronauts and to drag Red astronauts to safety.

Note: For the sake of succinctness, this report will focus on high-level concepts, algorithms and notable phases of development. Unless otherwise stated, trivial or repetitive ideas and concepts will be left out.

Section 2 Review of State of the Art



2.1 DRZ Telemax – Team Hector

DRZ Telemax is a tele-operated ground robot developed for autonomous exploration in a disaster environment. It integrates sensors such as LiDAR, cameras, and radiation detectors to create detailed 3D maps and radiation intensity distribution [1].

Strengths:

- Advanced autonomous mapping using SLAM through the use of data from LiDAR, inertial measurement system and 360 camera
- Maneuverable through uneven terrains such as stairs
- Equipped with a robotic arm for manipulation in hazardous environments

Weaknesses

- Tele-operation can be disrupted due to signal loss [2]

- It requires a second robot to carry a radio repeater for full communication coverage in disconnected zones

2.2 PackBot – Teledyne

PackBot is a compact, tele-operated robot used in search and rescue and EOD. It features multiple cameras and sensors, and a manipulator arm. It can be operated via a handheld controller or a laptop [3].

Strengths:

- Able to manoeuvre through tough terrains and navigate narrow passages
- Equipped with four colour cameras with zoom and illumination, providing strong situational awareness
- Lightweight and highly portable, allowing for quick deployment in the field

Weakness:

- Limited autonomy, which requires an operator to manage most decisions

Section 3 System Architecture

System Architecture Summary

The system is designed to support sensing and teleoperation by integrating various hardware and software components. The architecture consists of motor control, sensor integration, data processing, and a secure communication framework. The system is simplified in the figure below.

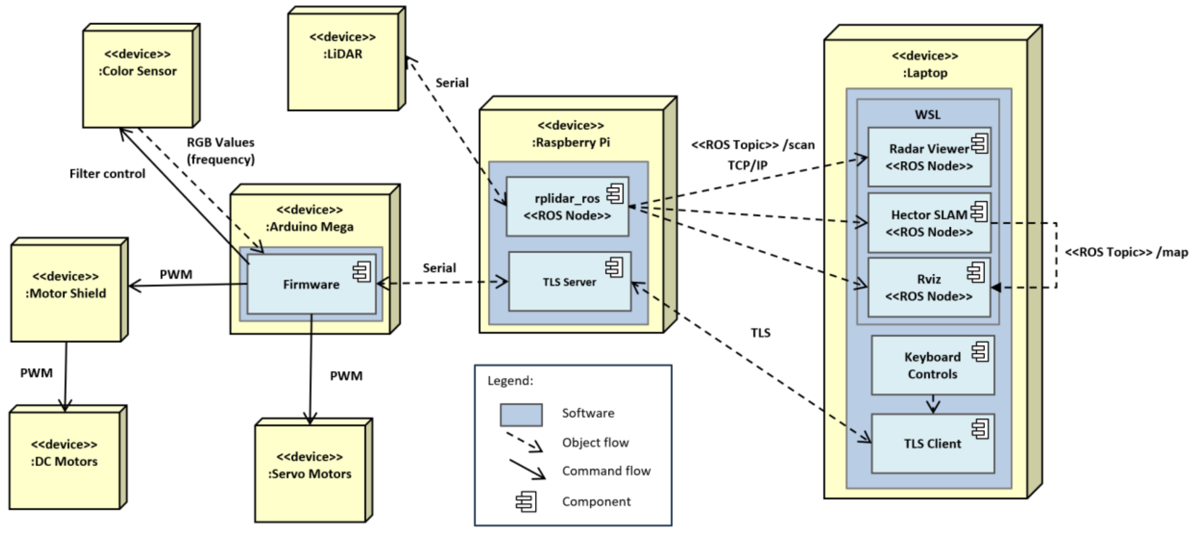


Figure 1: UML Deployment Diagram of high-level system architecture

Section 4 Hardware Design

4.1 Images of Alex

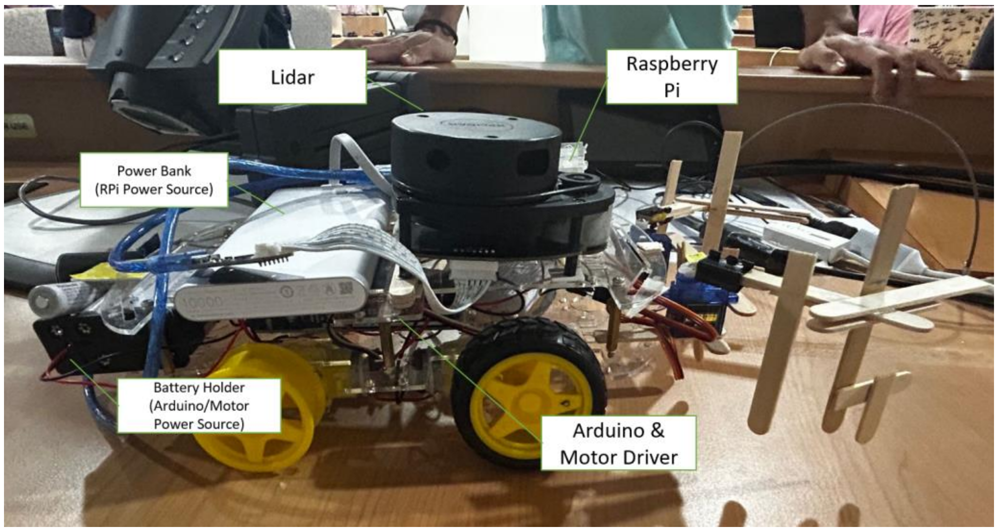

Lidar
Raspberry Pi
Power Bank (RPi Power Source)
Battery Holder (Arduino/Motor Power Source)
Arduino & Motor Driver

Figure 2: Side view of Alex

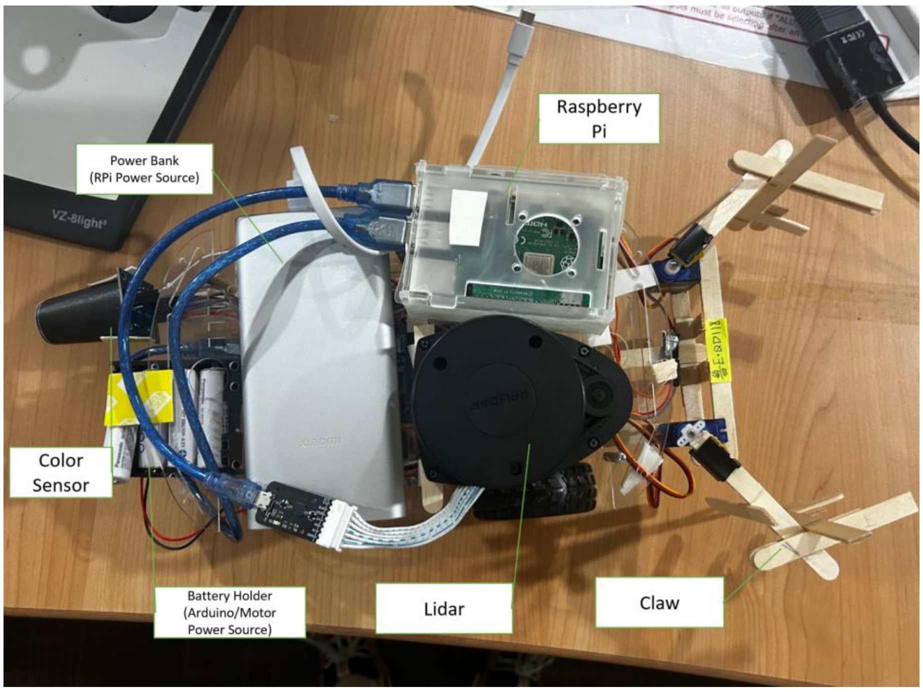

Raspberry Pi
Power Bank (RPi Power Source)
Color Sensor
Battery Holder (Arduino/Motor Power Source)
Lidar
Claw

Figure 3: Top View of Alex

4.2 Hardware Description

<u>4.2.1 General Layout</u>

Alex does not use any non-standard hardware. A mechanical claw is fixed at the front of the chassis. From the very start, we eschewed the idea of mounting the colour sensor on the front to provide more open space around it, thereby allowing more freedom in manoeuvring and, in turn, allowing colour data to be taken at positions where reliability can be maximised. An early design featured the colour sensor mounted on the side, but this was later changed in favour of placing the sensor on the rear to facilitate easier controlling and to give more room for movement in narrower passageways.

During testing, it was found that Alex struggled to turn at low motor powers. When turning, Alex would pivot around the front motors instead of its geometric center, and the rear tires would become a pure hindering force to turning. Therefore, the rear tires were removed to achieve a balance between grip and turning smoothness.

4.2.2 Claw

The earliest design for the claw was drafted under the assumption that the astronauts would have wheels, and thus would incur minimal friction. We adopted a "broom-and-dustpan" model. A forked holder is mounted on the chassis's left side, and a "broom" mounted on a servo would "sweep" the astronaut into the holder.

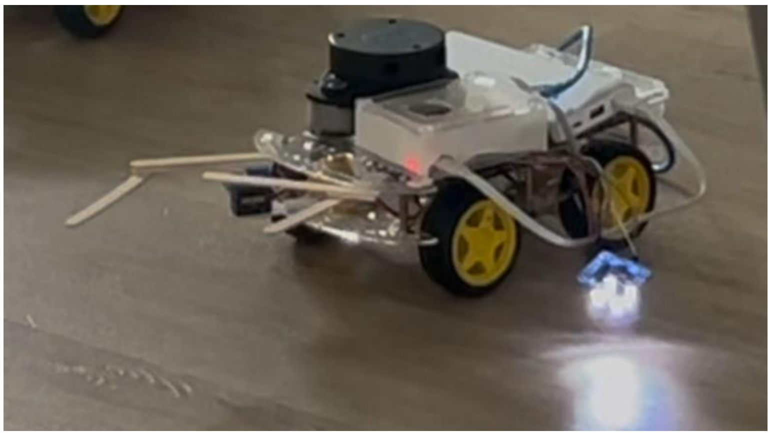


Figure 4: First Claw Design (With Colour Sensor Unmounted)

When it was announced that the astronauts would not have wheels, it became evident that a single servo would not be strong enough to drag the astronauts. Specifically, when turning at angles where the "broom" is nearly parallel to the direction of movement, the required clamping force increases dramatically due to simple trigonometry and exceeds the maximum output of a single servo. This is illustrated in the following highly abstracted figure:

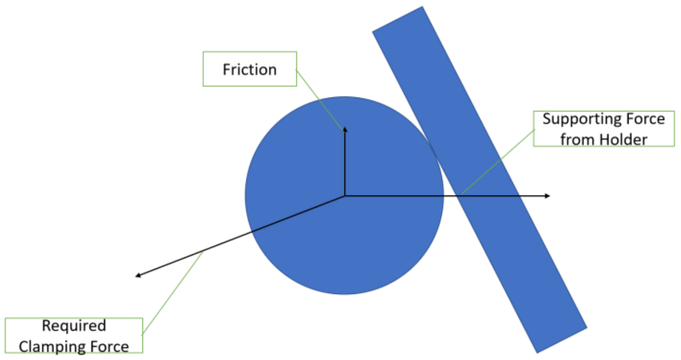


Figure 5: Demonstration of Component Forces

Therefore, in the final design as specified below, a two-servo claw design was implemented to firmly grip the astronaut. No less than 7 adhesives of various brands, ranging from duct tape to polyvinyl acetate, were tried, and eventually, cyanoacrylate was found to provide the most reliable adhesion. To provide better adhesion at the servo coupler-ice cream stick contact surface, which is a plastic-to-wood connection and the effect of cyanoacrylate here is limited, metal binder clips were used to support the contact surface.

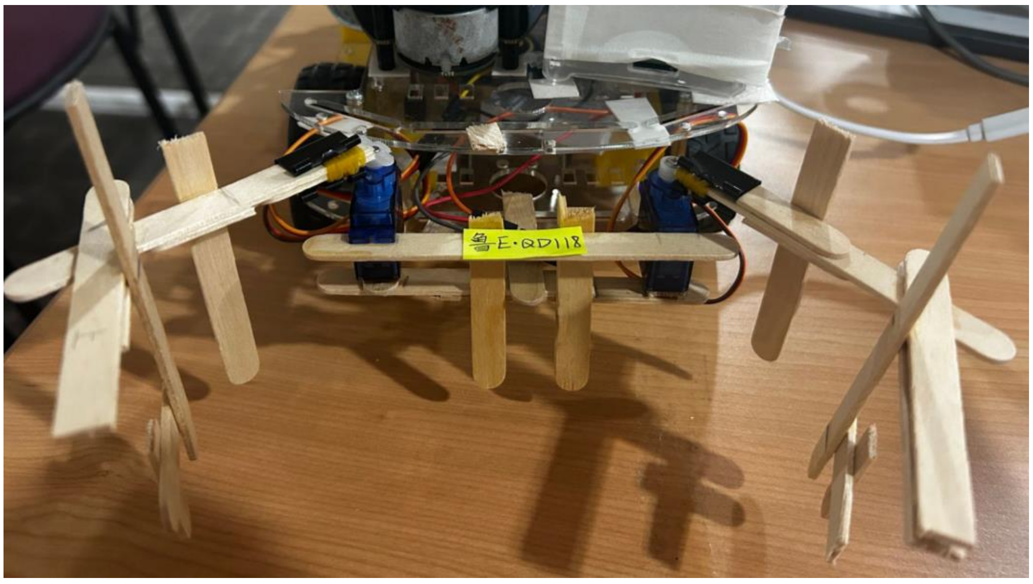


Figure 6: Close View of Claw

Section 5 Firmware Design

5.1 Communication Protocol between RPi and Arduino

The communication between RPi and Arduino is done through UART serial communication with a baud rate of 9600 bps and a frame of 8N1 (Refer to **Code Segment 1** for code implementation), using a structured packet format (TPacket) defined in *packet.h*. To initiate a UART communication session, the RPi first creates a TPacket with packetType PACKET_TYPE_HELLO and transmits it to the Arduino.

```
typedef struct
{
	char packetType;
	char command;
	char dummy[2]; // Padding to make up 4 bytes
	char data[MAX_STR_LEN]; // String data
	uint32_t params[16];
} TPacket;
```

Code Segment 1: Definition of TPacket structure

On the Arduino, a polling approach is used to continuously check for incoming data from the RPi. When sufficient bytes have been received to form a complete TPacket, the packet is checked for validity.

- If the received packet is valid, the handlePacket() function is called to process it.
- If the packet is invalid, an error TPacket is sent back to RPi for handling.

In the handlePacket function, only PACKET_TYPE_COMMAND will be handled, while the rest will be discarded, as we are only expecting commands to be communicated by the RPi. In the handleCommand function, the TPacket is parsed, and the respective command is carried out. This is done through a switch-case structure, the specific implementation of which can be found in the Appendix.

5.2 Movement Control

Movement Control on the Arduino is divided into two distinct types:

1. Indefinite Movement
   This mode is used for continuous motion in a specific direction and speed until a stop command is received. It allows for manual control where Alex keeps moving until instructed to stop.
2. Fine Movement
   This mode moves the robot by a small distance or angle. It is mainly used for fine adjustment and precise positioning. Alex automatically stops after completing the nudge action.

5.2.1 Indefinite Movement

Indefinite movement is handled by the move(speed, direction) function, which takes a speed percentage value and direction. Speed is internally scaled to match the motor driver's range of 0-255. Direction is determined using a switch-case structure. With each command, the motors are set to run in the corresponding directions using the Adafruit Motor Shield library. For convenience and simplicity, helper functions like forward, backward, right and left are used throughout the code. The specific implementation can be found in the Appendix.

5.2.2 Fine Movement

Fine movement is carried out based on time rather than distance. It consists of a directional movement function with distance 0 (as we are not using distance for motion control), and a short delay allows brief movement before executing the stop() command. This allows for a precise movement of Alex. The delay timing had also been carefully tested to our preferred distance. Refer to Appendix for code implementation for the fine movement function.

5.3 Servo Control

The servo mechanism on Alex is controlled directly through timer 5 on the Arduino. There are two separate commands for opening and closing the arm, which are controlled through the alteration of the duty cycle of the PWM signal. Refer to Appendix for code implementation for servo control.

5.4 Colour sensor

The colour sensor on Alex uses a combination of photodiodes for the detection of colour. The sensor works by illuminating the object with each colour at a time and measuring the intensity of the reflected red, green and blue light using photodiodes. This intensity is represented by the width of a square wave generated by the sensor. The specific LED colour is selected by manipulating control pins S2 and S3 through direct port control.

To reduce noise and improve accuracy, we took five readings per colour and computed the average through the avgFreq() function. This helps to smooth out fluctuations. Although readings are taken for each RGB colour, only red and green values are sent to the RPi, as the project only focused on detecting either red or green astronauts. All required data are packaged into a TPacket and transmitted to the RPi through the sendColour() function. Refer to Appendix for code implementation for the colour sensor.

Section 6 Software Design

6.1 Control Software

Below is a flowchart which illustrates the high-level teleoperation algorithm implemented for the system:

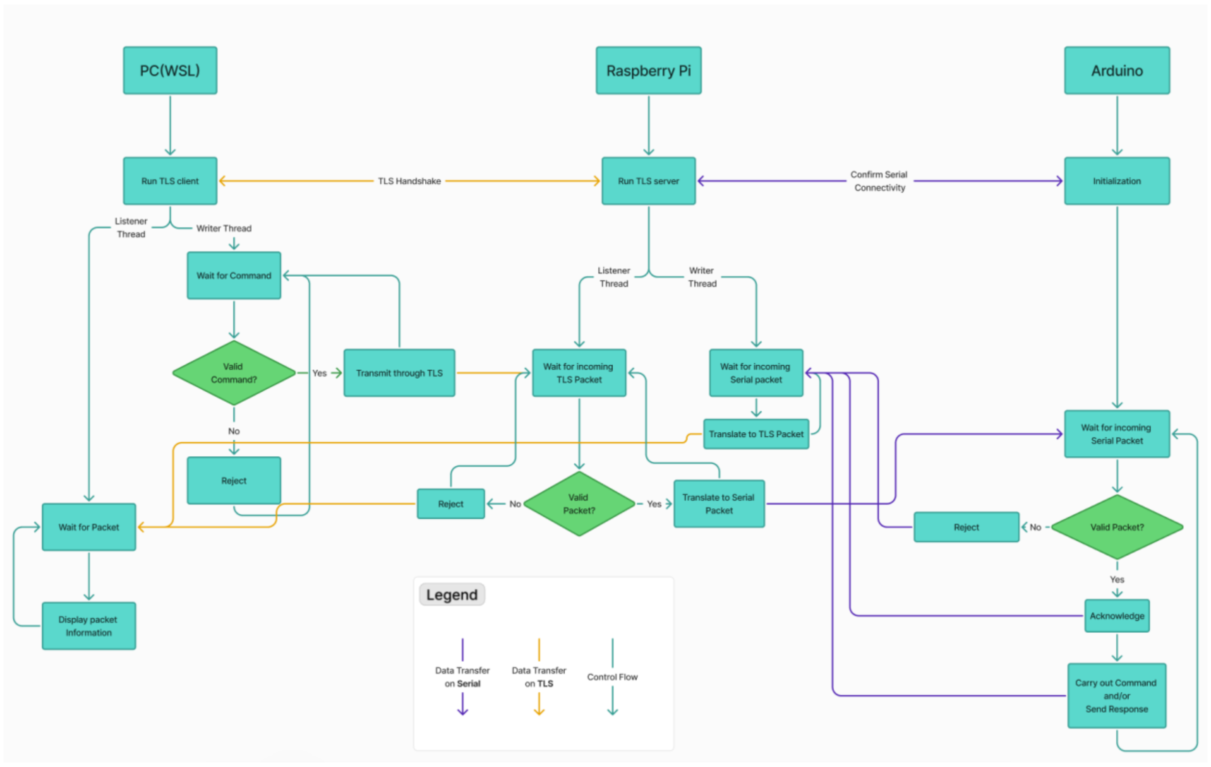


Figure 7: Flow Chart of Control Software

Instead of using f, b, l and r followed by the distance or angle, we used a WASD-based setup. Using the *termios.h* library, the terminal local mode flags ICANON and ECHO were set to false, thereby setting the terminal mode to non-canonical (which reads input characters without waiting for the newline character), and does not display the typed character onto the terminal. Below is the function to configure the terminal accordingly:

```
void setNonBlockingInput() {
    struct termios ttystate;
    tcgetattr(STDIN_FILENO, &ttystate);
    ttystate.c_lflag &= ~(ICANON | ECHO);
    ttystate.c_cc[VMIN] = 1;
    ttystate.c_cc[VTIME] = 0;
    tcsetattr(STDIN_FILENO, TCSANOW, &ttystate);
}
```
Code Segment 2: Terminal Configuration Function

This allows for much more straightforward and fluid control. Taking forward motion for example, the pilot would simply press W once, which commands Alex to start motion; and press W again to signal Alex to stop. It eliminated the need for wheel encoders, which were found to be unreliable and imprecise. In addition, a 9-speed gear system was implemented for speed control. The user would press "1" for 10% motor power, "6" for 60% motor power, etc. This allows us to configure the speed of Alex with greater precision.

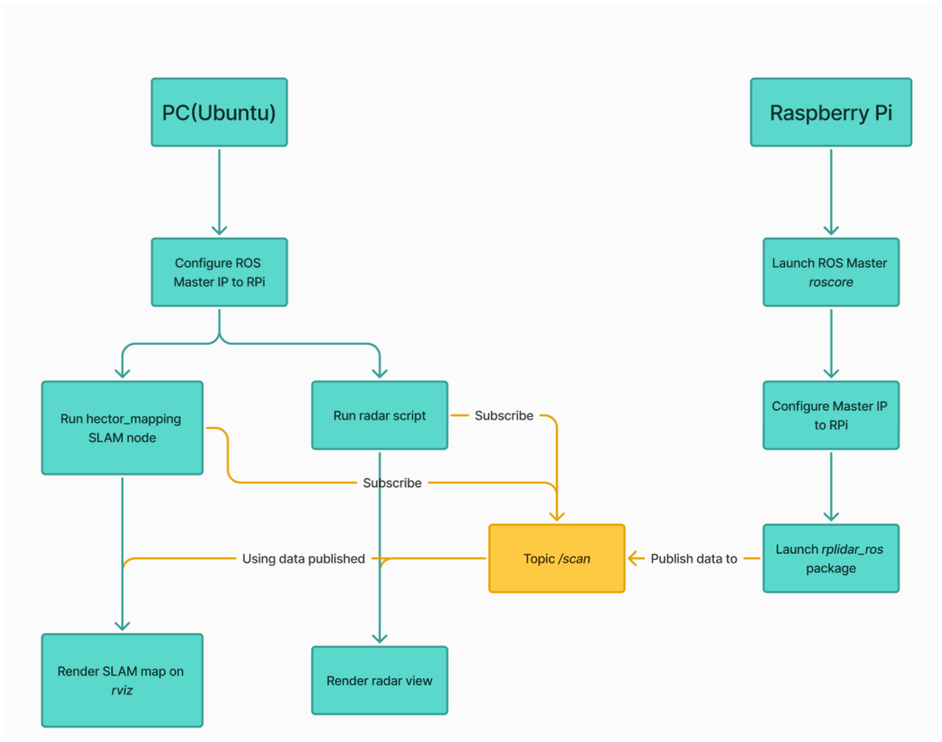
In practice, the pilot may want to command Alex to perform precision movements; i.e, turn at very small angles; in which case the user would press the same key twice in rapid succession. This, however, creates a problem: when two command packets are sent in rapid succession, they can interfere with and corrupt each other, resulting in the Arduino receiving packets with bad magic numbers. Therefore, the terminal is programmed to block user inputs until the acknowledgement of packet validity is relayed from the Arduino. The specific implementation can be found in the Appendix.

Tests have revealed that, even with this failsafe, the precision achieved could still be too poor to perform certain tasks, such as taking colour readings. Therefore, we have decided to implement precision manoeuvring, as detailed in Section 5. Furthermore, during testing, there have been instances where the quit command *q* is typed by mistake due to its close proximity to WASD. Hence, the quit command is changed to "|", which is nearly impossible to be pressed unintentionally.

In addition to movement commands, there is also a command to obtain data from colour sensors, as well as commands to open and close the robotic arm. Their implementations are trivial, and a full list of keybindings can be found in the Appendix.

6.2 SLAM, LiDAR and ROS

Below is a flowchart which illustrates the high-level SLAM and LIDAR visualisation algorithm implemented for the system:



PC(Ubuntu)
Raspberry Pi
Configure ROS Master IP to RPi
Launch ROS Master *roscore*
Run hector_mapping SLAM node
Run radar script
Subscribe
Configure Master IP to RPi
Subscribe
Using data published
Topic */scan*
Publish data to
Launch *rplidar_ros* package
Render SLAM map on *rviz*
Render radar view

Figure 8: SLAM / LiDAR software flowchart

The initial version of the SLAM algorithm implemented the visualisation of raw scan data (radar charts) and a Breezy SLAM map on the RPi itself, without deploying ROS. This brought a multitude of problems. For starters, the RPi is extremely limited in terms of performance, making the algorithm sluggish. In addition, the only method to display the visualisation on the control terminal is through VNC, which is extremely inefficient for this type of data transfer and further slows down the algorithm. Furthermore, to enable better control at point-blank range, we implemented multiple radar chart displays at different scales, again increasing the computational burden on the RPi. Testing has verified that this setup creates noticeable delays, requiring extra care from the pilot and making control difficult.

For all the reasons stated above, we instead chose to implement the radar chart and SLAM on the control terminal, which is supported by an NVIDIA GeForce RTX 4060 GPU, a 13th-Gen Intel Core i9 CPU, and 16 Gigabytes of RAM and can render all visualisations with negligible delays. To better take advantage of a blooming online open-source community [4], we fully deployed ROS on both the PC and RPi, and switched to hector mapping. The hector mapping node and radar chart program subscribe to the */scan* topic, to which the RPi broadcasts scan data from the LiDAR. The node then uses the acquired data to render visualisations.

To facilitate piloting Alex through cramped environments and performing tasks that require high precision (e.g, taking colour readings), we rendered two radar charts of different scales and displayed several assistive markers. Below is a screenshot of the radar chart interface:

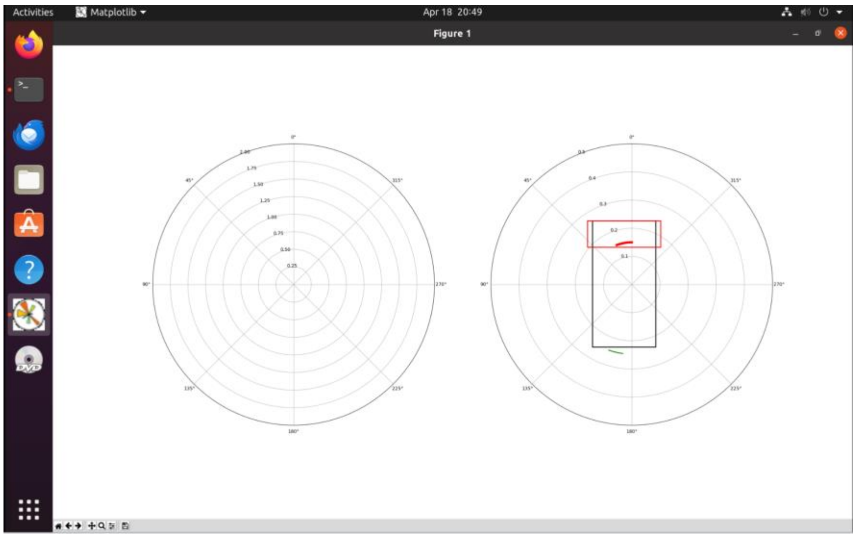


Figure 9: Screenshot of Control Interface

The left represents a zoomed-out view of Alex's surroundings within 2 meters, and the right is a zoomed-in view displaying scan data within 50 centimetres. The black rectangle marks the perimeter of Alex, and the red rectangle marks the perimeter in which the claw operates. The red arc indicates the optimal location at which the astronaut's image should appear on radar for pickup, while the green arc indicates the position where colour readings would be most precise. This allows for both general navigation and high-precision manoeuvring. Coupled with the SLAM map on *RViz*, the pilot can effectively identify astronauts, take colour measurements, and grab it with the claw, thereby eliminating the need to use a camera.

Finally, to reduce preparation time, we created two .sh command files on the RPi that automatically launch relevant applications. The code for both .sh files can be found in the Appendix.

Section 7 Lessons Learnt - Conclusion

7.1 Key Learnings

7.1.1 Timely Backups

During the development phase, all relevant folders are backed up on multiple devices prior to each version update. This includes backups for files that were not changed. We chose to update entire folders to preserve directory information, thereby enabling easy rollbacks. This has saved the project multiple times, most notably during the installation of Debian 10 while deploying ROS. When we inserted the SD card into a laptop, it slid past the SD card slot and fell into the motherboard. Lacking the specific screwdriver to dismantle the laptop, the SD card is effectively lost. Luckily, all relevant folders have been backed up on a separate hard drive, and we were able to recover the files after using a new SD card. This is a very vivid lesson that taught us the importance of timely backups across multiple devices.

7.1.2 Using Reliable Prototypes

Due to constraints with lab time, during most of the developmental phase, we did not have access to real astronauts for testing. As such, we used empty cola cans to test the strength of the claw. However, as it turns out, the weight, coefficient of friction, and center of mass of the cola can are all significantly different from the astronauts themselves, which meant that emergency design patches were needed to guarantee reliability. For future projects, we will use testing prototypes that better reflect real scenarios.

7.2 Mistakes

7.2.1 Maintaining Upgrade Compatibility

The first version of the claw was mounted using cyanoacrylate without extensive testing on real astronauts. This created a problem when it was decided to switch to a two-servo design. The original claw had to be ripped off, and it was only through good luck that it was done without inducing major structural damage to the chassis. In future projects, we will make permanent structural changes only after thoroughly validating their reliability.

7.2.2 Piloting Proficiency

During the project, we over-concentrated on implementing the technical aspects of Alex and neglected practising its controls. This led to a number of human errors, especially during the trial demo. Below is an incomplete list of pilot errors made during the trial demo (Note: This list is compiled by the pilot himself as a self-reflection):

1. The pilot accidentally turned on airplane mode on his PC. While he realised that the WiFi was switched off, he did not realise that the Bluetooth was deactivated as well, leading to him erroneously thinking that the mouse had failed and resorting to the trackpad for the rest of the setup process, inducing delays.
2. The pilot made a number of typos when typing Linux commands for setup, such as typing “http” as “httjp”, further inducing delays
3. The pilot forgot to switch to low gear when turning, inducing turn speeds that exceed the designed capacity of the SLAM algorithm and causing the map to drift. A long period of time was required to reset the map.

Even during the final demo, the pilot still occasionally pressed keystrokes by mistake, commanding Alex to perform unintended motions. Fortunately, these mistakes were corrected before causing serious consequences such as hard collisions with obstacles. We recognise the importance of practising controlling our engineering solution in future projects.

References

1. *Robots - Team Hector*. (n.d.). Team Hector. https://www.teamhector.de/robots

2. Wakasa, Y., Hakamada, K., Morohashi, H., Kanno, T., Tadano, K., Kawashima, K., Ebihara, Y., Oki, E., Hirano, S., & Mori, M. (2024). Ensuring communication redundancy and establishing a telementoring system for robotic telesurgery using multiple communication lines. *Journal of robotic surgery*, *18*(1), 9. https://doi.org/10.1007/s11701-023-01792-8

3. *Packbot 510*. (n.d.). Teledyne FLIR. https://www.flir.com/products/packbot/

4. *hector_slam* (n.d.). ROS Wiki. https://wiki.ros.org/hector_slam

Appendix

```
void setupSerial()
{
  UCSR0C = 0b00000110;
  // Set baud rate to 9600
  UBRR0H = 0;
  UBRR0L = 103;
  UCSR0A = 0;
}

void startSerial()
{
    UCSR0B = (1 << RXEN0) | (1 << TXEN0);
}
```

Code Segment: Serial Communication setup

```
void handleCommand(TPacket *command)
{
  switch(command->command)
  {
    // For movement commands, param[0] = distance, param[1] = speed.
    case COMMAND_FORWARD:
        sendOK();
        forward((double) command->params[0], (float) command->params[1]);
      break;
      case COMMAND_REVERSE:
        sendOK();
        backward((double) command->params[0], (float) command->params[1]);
      break;
      case COMMAND_TURN_LEFT:
        sendOK();
        right((double) command->params[0], (float) command->params[1]);
      break;
      case COMMAND_TURN_RIGHT:
        sendOK();
        left((double) command->params[0], (float) command->params[1]);
      break;
      case COMMAND_STOP:
        sendOK();
        stop();
      break;
      case COMMAND_GET_STATS:
        sendOK();
        sendStatus();
        break;
      case COMMAND_CLEAR_STATS:
        sendOK();
        clearOneCounter(command->params[0]);
        break;
      case COMMAND_OPEN_ARM:
        sendOK();
        openArm();
        break;
      case COMMAND_CLOSE_ARM:
        sendOK();
        closeArm();
        break;
      case COMMAND_COLOR:
        findColor();
        sendOK();
        break;
      case COMMAND_NLEFT:
        sendOK();
        nLeft();
        break;
      case COMMAND_NRIGHT:
        sendOK();
        nRight();
        break;
      case COMMAND_NFORWARD:
        sendOK();
        nForward();
        break;
      case COMMAND_NBACKWARD:
        sendOK();
        nBackward();
        break;
    default:
      sendBadCommand();
  }
}
```

Code Segment: handleCommand() function

```
void move(float speed, int direction)
{
  int speed_scaled = (speed/100.0) * 255;
  motorFL.setSpeed(speed_scaled);
  motorFR.setSpeed(speed_scaled);
  motorBL.setSpeed(speed_scaled);
  motorBR.setSpeed(speed_scaled);

  switch(direction)
    {
      case BACK:
        motorFL.run(FORWARD);
        motorFR.run(FORWARD);
        motorBL.run(FORWARD);
        motorBR.run(FORWARD);
      break;
      case GO:
        motorFL.run(BACKWARD);
        motorFR.run(BACKWARD);
        motorBL.run(BACKWARD);
        motorBR.run(BACKWARD);
      break;
      case CW:
        motorFL.run(FORWARD);
        motorFR.run(BACKWARD);
        motorBL.run(FORWARD);
        motorBR.run(BACKWARD);
      break;
      case CCW:
        motorFL.run(BACKWARD);
        motorFR.run(FORWARD);
        motorBL.run(BACKWARD);
        motorBR.run(FORWARD);
      break;
      case STOP:
      default:
        motorFL.run(STOP);
        motorFR.run(STOP);
        motorBL.run(STOP);
        motorBR.run(STOP);
    }
}
```

Code Segment: move() function

```
void nRight(){
  ccw(0,45);
  delay(100);
  stop();
}
void nLeft(){
  cw(0,45);
  delay(100);
  stop();
}
void nForward(){
  forward(0,45);
  delay(80);
  stop();
}
void nBackward(){
  backward(0,45);
  delay(100);
  stop();
}
```

Code Segment: Fine movement function

```
void setupServo(){
    // 1. Set PL3 as output
    DDRL |= (1 << 3) | (1<<4);
    // 2. Set Phase Correct PWM mode
    TCCR5A = 0b10000010;
    TCCR5B = 0b00010010;
    // 3. Set ICR5
    ICR5 = 20000;
    // 4. Set OCR5A for duty cycle
    OCR5A = 0;
    OCR5B = 0;
    // 5. Enable Non-Inverting Mode on OC5A and B
    TCCR5A |= (1 << COM1A1);
    TCCR5A |= (1 << COM1B1);
    TCNT5 = 0;
}
void openArm(){
  OCR5A = 950;
  OCR5B = 1000;//1000, The larger the bigger open
}
void closeArm(){
  OCR5A = 1600;//1500
  OCR5B = 400;//500
}
```

Code Segment: Servo Control

```
void sendColour() {
  TPacket colourPacket;
  colourPacket.packetType = PACKET_TYPE_RESPONSE;
  colourPacket.command = RESP_COLOR;
  colourPacket.params[0] = redFreq;
  colourPacket.params[1] = greenFreq;
  sendResponse(&colourPacket);
}
void colourSetup() {
  DDRA |= ((1 << S0) | (1 << S1) | (1 << S2) | (1 << S3)); //Set S0, S1, S2,
S3 to OUTPUT
  DDRA &= ~(1 << sensorOut_mapped); //Set sensorOut to INPUT
  //Setting frequency scaling to 20%
  PORTA |= (1 << S0);
  PORTA &= ~(1 << S1);
}
int avgFreq() {
  int reading;
  int total = 0;
  //Obtain 5 readings
  for (int i = 0; i < 5; i++) {
    reading = pulseIn(sensorOut, LOW);
    total += reading;
    delay(colorSensorDelay);
  }
  //Return the average of 5 readings
  return total / 5;
}
void findColor() {
  // Setting RED filtered photodiodes to be read
  PORTA &= ~((1 << S2) | (1 << S3));
  delay(colorSensorDelay);
  // Reading the output frequency for RED
  redFreq = avgFreq();
  delay(colorSensorDelay);
  // Setting GREEN filtered photodiodes to be read
  PORTA |= ((1 << S2) | (1 << S3));
  delay(colorSensorDelay);
  // Reading the output frequency for GREEN
  greenFreq = avgFreq();
  delay(colorSensorDelay);
  // Setting BLUE filtered photodiodes to be read
  PORTA &= ~(1 << S2);
  PORTA |= (1 << S3);
  delay(colorSensorDelay);
  // Reading the output frequency for BLUE
  blueFreq = avgFreq();
  delay(colorSensorDelay);
  sendColour();
}
```

Code Segment: Color Sensing

Key	Command
W	Start/Stop Forward Motion
A	Start/Stop Reverse Motion
S	Start/Stop Left Turn
D	Start/Stop Right Turn
E	Fine Movement: Forward
X	Fine Movement: Backward
,	Fine Movement: Turn Left
.	Fine Movement: Turn Right
V	Get Color Reading
1-9	Set Power Level to Corresponding Number
J	Open Claw
K	Close Claw
|	Quit Program

Table 1: Key Bindings

```
void *writerThread(void *conn) {
    int quit = 0;
    static char current_cmd = 0;  // Last command sent

    setNonBlockingInput();  // Switch terminal to raw (non-canonical) mode

    while (!quit) {
        char ch;

        // Check if we're still waiting on a previous response
        pthread_mutex_lock(&responseMutex);
        int ignore = waitingForResponse;
        pthread_mutex_unlock(&responseMutex);

        // If not waiting, try to read one character (no ENTER needed)
        if (!ignore && read(STDIN_FILENO, &ch, 1) > 0) {
            char buffer[10];
            int32_t params[2];

            buffer[0] = NET_COMMAND_PACKET;

            switch (ch) {
```

```
        case 'w':
        case 'W':
            // Toggle forward / stop
            if (current_cmd == 'f') {
                buffer[1] = 's';
                params[0] = 0;
                params[1] = 0;
                current_cmd = 0;
            } else {
                buffer[1] = 'f';
                params[0] = 0;
                params[1] = powerLevel;
                current_cmd = 'f';
            }
            break;

        case 'a':
        case 'A':
            // Toggle turn left / stop
            if (current_cmd == 'l') {
                buffer[1] = 's';
                params[0] = 0;
                params[1] = 0;
                current_cmd = 0;
            } else {
                buffer[1] = 'l';
                params[0] = 0;
                params[1] = turnLevel;
                current_cmd = 'l';
            }
            break;

        case 's':
        case 'S':
            // Omitted for Simplicity


        case 'd':
        case 'D':
            // Omitted for Simplicity


        case '|':
            // Exit the loop
            quit = 1;
            break;

        case 'J': case 'j':
        case 'K': case 'k':
        case 'G': case 'g':
                case 'V': case 'v':
                case '<': case ',':
                case '>': case '.':
                case 'E': case 'e':
                case 'X': case 'x':
                    // Open/close arm, etc. - single-key commands
                    buffer[1] = ch;
                    params[0] = 0;
                    params[1] = 0;
                    current_cmd = ch;
                    break;

                case '1': case '2': case '3': case '4':
                case '5': case '6': case '7': case '8': case '9':
                    // Set power and turn levels (10-90)
                    powerLevel = (ch - '0') * 10;
                    turnLevel = powerLevel;
                    printf("Power level set to %d\n", powerLevel);
                    printf("Turn level set to %d\n", turnLevel);
                    // No sendData() for just level change
                    continue;

                default:
                    // Unrecognized key - ignore
                    continue;
            }

            // Copy params and send packet
            memcpy(&buffer[2], params, sizeof(params));
            sendData(conn, buffer, sizeof(buffer));

            // Now block further input until response arrives
            pthread_mutex_lock(&responseMutex);
            waitingForResponse = 1;
            pthread_mutex_unlock(&responseMutex);
        }

        usleep(50000);
    }

    resetTerminal();  // Restore canonical mode and echo
    printf("Exiting keyboard thread\n");
    stopClient();
    EXIT_THREAD(conn);
    return NULL;
}
```

Code Segment: Writer Thread in client program

```bash
#!/bin/bash
source ~/cg2111a/devel/setup.bash
roscore
```
Code Segment: runcore.sh

```bash
#!/bin/bash
read -p "Enter <RPi_IP>: " Addr
source ~/cg2111a/devel/setup.bash
export ROS_MASTER_URI=http://$Addr:11311
roslaunch rplidar_ros rplidar.launch
```
Code Segment: runlidar.sh