CG2111A Engineering Principle and Practice II

Semester 2 2023/2024

"Alex to the Rescue"

Final Report

Team: B03-2B

Name	Student #	Sub-Team	Role
EDWARD RAFAEL LUKITO	A0288098H	Firmware	Set up ultrasonic sensor code and main Arduino code.
LI ZEKUAN	A0287942M	Hardware	Assemble the robot and movement code.
LIM JUN FUN	A0271690B	Firmware	Set up colour sensor code and main Arduino code.
LIU WENYI	A0273715B	Software	Set up LiDAR and SLAM programs. Code master control program on RPi and Laptop.

"The real Alex was the friends we made along the way" - Linus Sebastian (maybe not)

Table of Contents

Section 1 Introduction

At the start of 2024, Japan experienced an earthquake which resulted in over a hundred deaths and left many missing or trapped under the rubble of destroyed buildings.[1] Occurrences like this happen all across the world, causing thousands to lose their lives, many of whom could have been saved if they were rescued earlier. Enter search and rescue robots, bots which go through rough terrain to find survivors to increase their chances of being saved. These robots can enter areas humans are unable to by being controlled remotely and map out the area using methods like SLAM as well as identify survivors, aiding rescuers in their rescue operations.[2]

In our project: Alex to the Rescue, we built a robot named Alex, a search and rescue robot to achieve those functions. Alex made use of LiDAR mounted on top of it to map surroundings and could be controlled remotely with teleoperation over a network. Alex is also equipped with other various devices to complete other tasks, such as colour detection to identify simulated earthquake victims. For this project run, our Alex was tasked to map out an unknown area in under 6 minutes and to identify two targets, one green and one red. These had to be done while avoiding collisions with walls and avoiding mistakenly identifying a white dummy target.

Section 2 Review of State of the Art

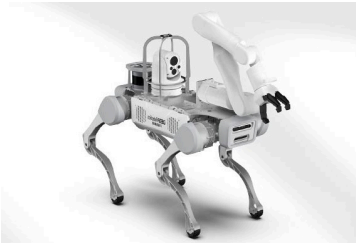
2.1 Teledyne FLIR's SkyRanger R70 Drone

a. **Description:** Designed to be operated remotely for search and rescue missions in challenging terrains and concealed locations, its notable functionalities include motion tracking, thermal imagery, object detection, and classification. It is equipped with various optical sensors such as the HDZoom 30 and the EO/IR Mk-II, enabling accurate motion tracking at long distances, and high-fidelity daylight/thermal imagery respectively. With multiple embedded NVIDIA TX2 processors, the drone is equipped with the capabilities of real-time artificial intelligence processing, enabling precise object detection and classification. The addition of four computer vision cameras also enables autonomous operations and flights in GPS-denied environments.[3]

b. Summary of its strengths and weaknesses

- **Strengths:** Due to its advanced hardware, the drone is able to execute its main system functionalities precisely and accurately. It is also attached with an Osprey carry and delivery payload, which enables operators to deliver nearly any object up to 2 kg such as radios, medkits, and more. The R70's carbon and magnesium airframe also makes it very resilient, enabling it to sustain winds at 65 kph and operate up to 15000' MSL.
- **Weaknesses:** Its typical endurance is over 40 minutes with its standard endurance propulsion system, or about 50 minutes with its high endurance propulsion system, which is not very long and can be a weakness for longer tasks. Furthermore, its radio range is also up to 8 km only, making it limited to only being operated within this radius.[4]

2.2 DEEP Robotics' Jueying X20



a. **Description:** By integrating a long-distance communication system, a bi-spectrum PTZ camera, gas sensing equipment, an omnidirectional camera, and a pickup, the robot includes long-distance control and image transmission, heat source tracking, real-time detection of harmful gases, and rescue calls, among other functions. Moreover, Jueying X20 is available with an optional rotating laser scanner, which can obtain high-precision point cloud data from indoor & outdoor disaster and accident sites, providing vital information for post-disaster data analysis.[5]

b. Summary of its strengths & weakness

- **Strength:** With The advanced sensors equipped, its strong durability and terrain adaptability allow it to navigate complex terrains and extreme weather conditions. Its payload capacity of 40 kg also makes it useful for tasks such as delivering supplies.
- **Weakness:** The battery life (about 4 hours) can be a limitation for longer tasks. What's more, while the hardware is reliable in extreme situations, signal loss may become a potential pain point.[6]

Section 3 System Architecture

Our Alex consists of 9 devices in total: A Raspberry Pi acting as the main controller, connected to a LiDAR, Arduino and LCD, and powered by a power bank. The Arduino acted as a controller for the motors through a motor driver, encoders, colour sensor, ultrasonic sensor and walkie-talkie. The driver was powered by a 9V battery pack. On the operator end, we had two laptops, one for the TLS Client to send commands to the Pi and one for RViz to visualise the LiDAR and SLAM data. We also had the other end of the walkie-talkie.

.

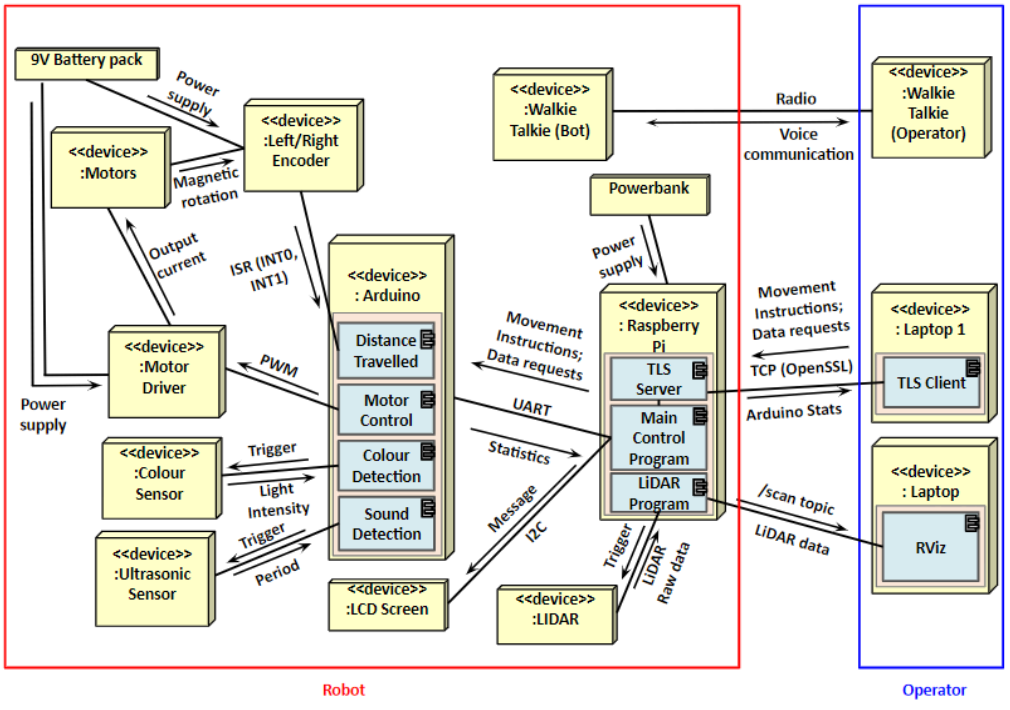


Figure 1: The simplified UML deployment diagram for our project

Section 4 Hardware Design

4.1 Placement of components

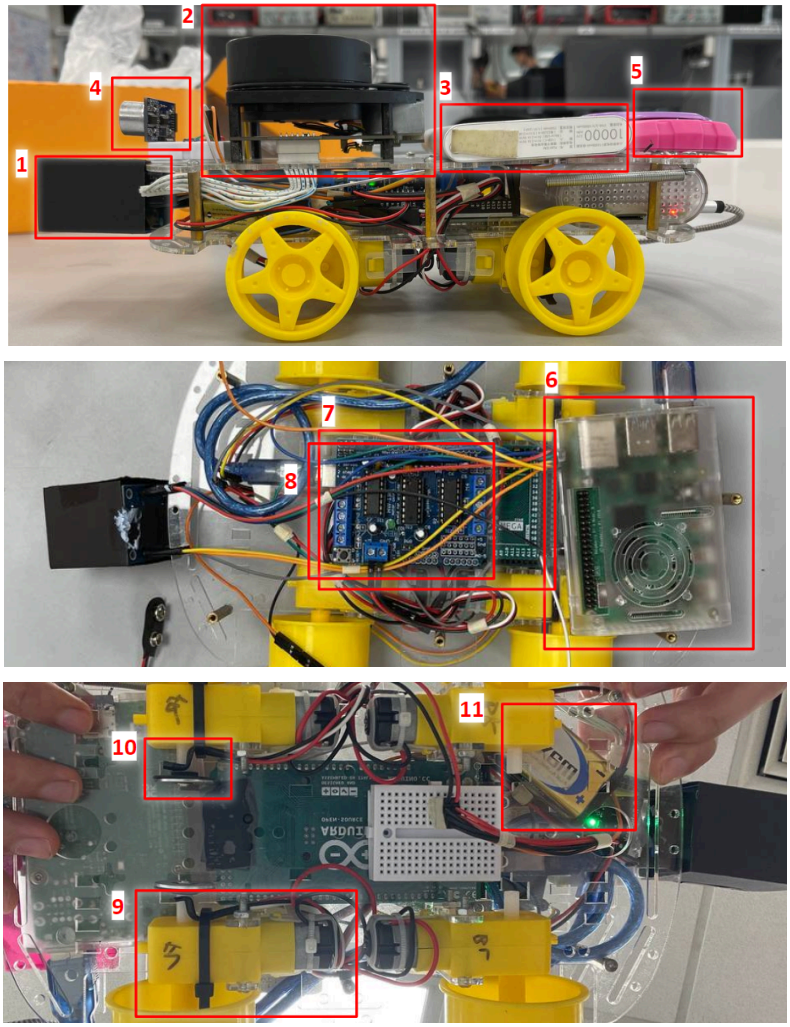


Figure 2: Images of Alex with components labelled (Side view, top view, bottom view).

No.	Name	Description
1	Colour sensor	Colour sensor with shielding to prevent external light interference, connected to the Arduino.
2	LiDAR	LiDAR connected to the RPi, placed at the top to prevent other components from blocking its laser.
3	Power bank	Power bank used to power the RPi.
4	Ultrasonic sensor	An additional component added to detect the distance between Alex and objects in front. Used to prevent collisions.
5	Walkie	An additional component added to facilitate communication between

	Talkie	the operator and survivors.
6	Raspberry Pi	Main controller for Alex. It was placed at the back in a position that made it easy for us to access all the ports.
7	Arduino Mega	Controller for external devices. Placed in the centre of the board so that wires could be connected to devices all around the bot without issue.
8	Motor Driver	Control the direction and speed of the motors, mounted on the Mega.
9	Motors	Mounted at the bottom of Alex.
10	Hall-effect sensors	Zip-tied to the two rear motors, track the number of rotations by sending interrupts to the Mega.
11	9V battery	9V used instead of the provided 6V to power the motor driver.

4.2 Additional hardware features:

4.2.1 Ultrasonic sensor

The ultrasonic sensor is mounted on a breadboard at the front of Alex to determine the distance between obstacles and Alex upon request by the operator. The trigger pin emits ultrasonic sound waves, which will be received by the echo pin. The duration can then be found and the distance will be calculated in the code. As the ultrasonic was mounted behind the colour sensor's shield which was about 4 cm long, we made sure to take that into consideration whenever reading the data to avoid crashes.

4.2.2 Colour Sensor

The colour sensor (TCS3200) is placed at the front of Alex to assist in identifying the victims (Red / Green) or false targets (White). The sensor converts the readings from the photodiodes into square waves using a current-to-frequency converter. From there, we read in the respective frequency from the Red, Green, and Blue photodiodes respectively to determine the colour of the object. This is done when the user sends a command to the Arduino Mega to identify colour. Since the readings are affected by ambient lighting, we have covered the sensor with a black paper shield to minimise the amount of ambient light being read by the sensor, to ensure accuracy in identifying the colours needed. The software implementation and the colour sensor algorithm will be discussed more in section 6.2.

4.2.3 Heatsink on RPi

We noticed that after long periods of continuous operation, the RPi would become significantly hotter, affecting its performance. To prevent this and keep it operating at optimal temperature ranges, we added heat sinks to its chips to help with heat dissipation.

4.2.4 Walkie talkie

We connected a walkie talkie to the Arduino Mega by stripping the walkie talkie, cutting the power wires and attaching them to jumper wires. The jumper wires were then connected to pins on the Mega, allowing us to turn the walkie talkie on and off via the Mega.

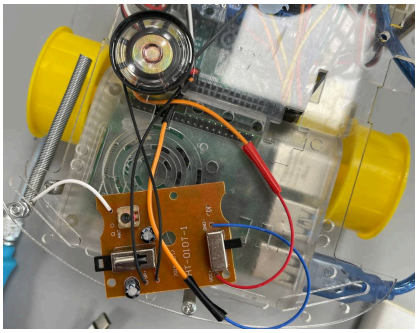


Figure 3: The internals of the walkie talkie. The orange wires are connected to the Mega.

4.2.5 LCD

We had intended to connect the LCD to the RPi via the SDA and SCL pins for I2C. However, after testing we realised that the LCD was not functioning correctly and removed it for the final run (elaboration on this is under *Conclusion - Mistakes made*).

4.3 Other hardware considerations:

4.3.1 Cable Management

By using tape and bluetack, we made sure that the wires are kept within the dimensions of the chassis, and also keeping the wires attached to the electronic components such as the colour sensors and the breadboard (where we have opted to put at the bottom of the chassis).

4.3.2 Manoeuvrability

To ensure that our Alex could be manoeuvred easily, we took into account the weight distribution, power to the motors and the traction of the wheels when building Alex. We placed the heaviest component (the power bank) in the centre of the bot to make its centre of gravity as near the middle as possible, aiding Alex's pivot turning. The 6V battery pack was replaced with a 9V battery as we felt that 6V was not sufficient to move around easily, and the 9V package is lighter than the 6V package. Finally, we removed the tires on the wheels as we realised that the plastic had better traction on the ground where Alex would be tested on.

Section 5 Firmware Design

5.1 Main component firmware:

After the initial handshake process between the RPi and the Arduino, the Arduino continuously polls for packets from the RPi and responds to them according to the algorithm below.

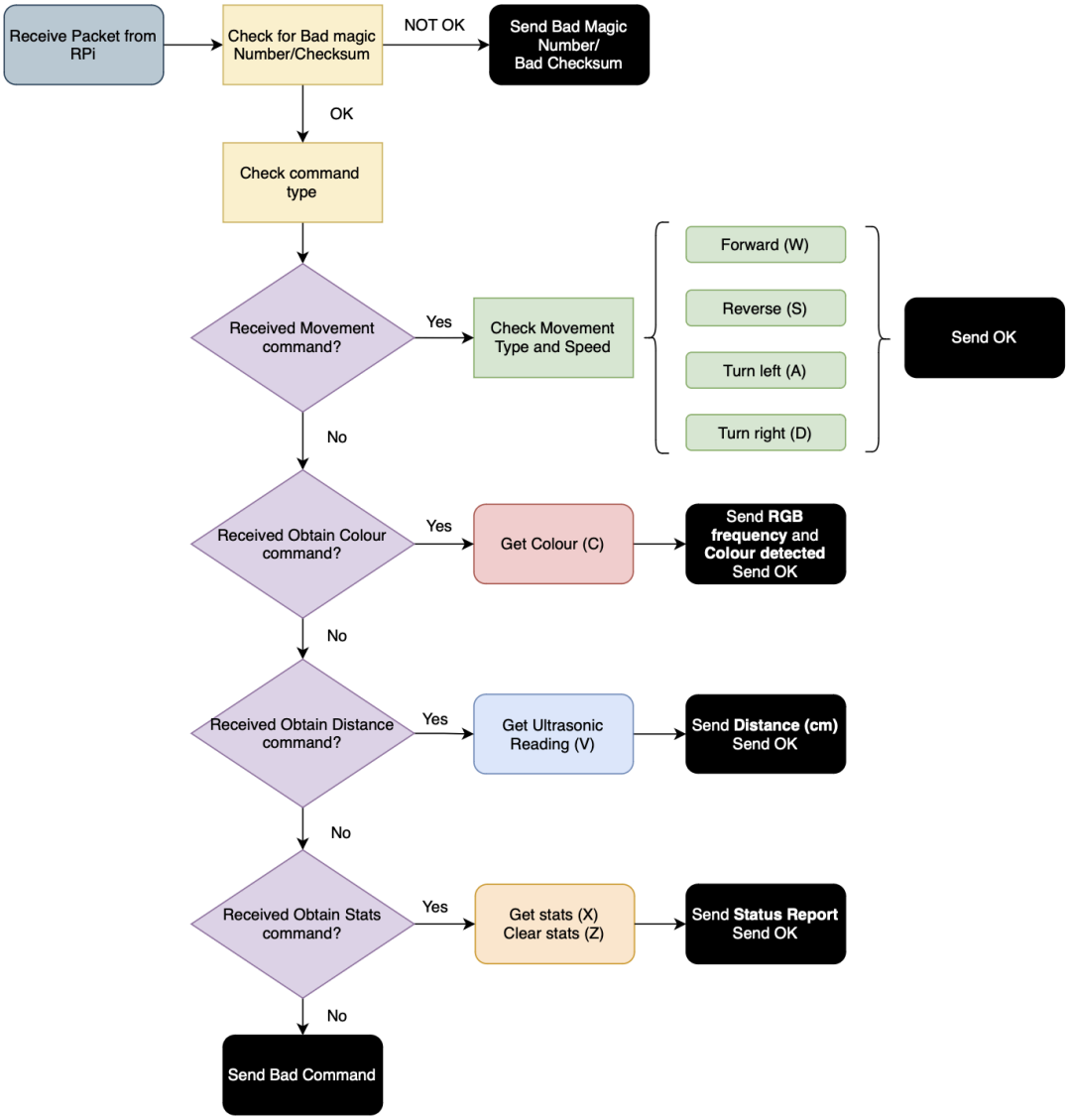
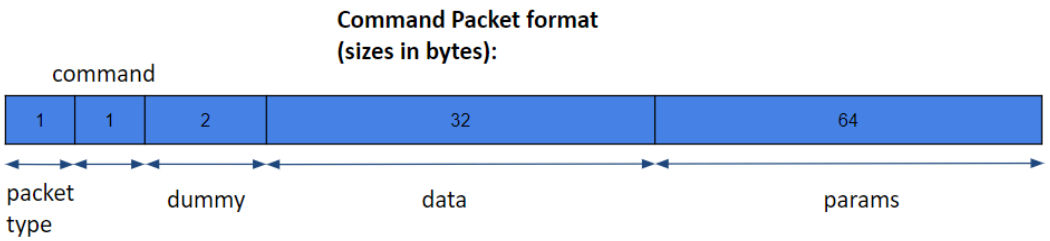


Figure 4: High level algorithm for firmware on Arduino Mega

5.1.1 UART

UART was used for the serial communications between the RPi and Arduino to send commands and responses. It was set up with 9600 baud and an 8N1 frame format. The bare-metal code used for this can be found in *Appendix 1*. The command packets sent over to the Arduino followed this format:



Command types	Response types

```
typedef enum
{
  //movement
  COMMAND_FORWARD = 0,
  COMMAND_REVERSE = 1,
  COMMAND_TURN_LEFT = 2,
  COMMAND_TURN_RIGHT = 3,
  COMMAND_STOP = 4,

  //send and clear stats
  COMMAND_GET_STATS = 5,
  COMMAND_CLEAR_STATS = 6,

  //send colour and u/s data
  COMMAND_COLOUR = 7,
  COMMAND_ULTRASONIC = 8
} TCommandType;
```

```
typedef enum
{
  //alls good
  RESP_OK = 0,

  //odometry stats
  RESP_STATUS=1,

  //all is not good
  RESP_BAD_PACKET = 2,
  RESP_BAD_CHECKSUM = 3,
  RESP_BAD_COMMAND = 4,
  RESP_BAD_RESPONSE = 5,

  //colour and u/s data
  RESP_COLOUR = 6,
  RESP_ULTRASONIC = 7
} TResponseType;
```

Table 1: The different command and response types used during the serial communication.

5.2 Additional features firmware:

5.2.1 Ultrasonic code

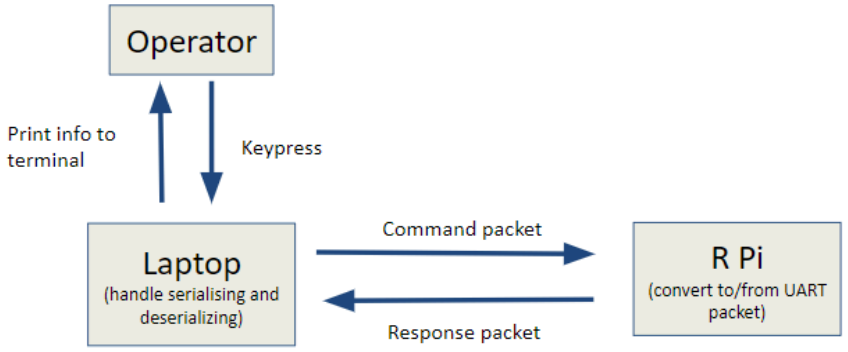
The distance between the objects in front and Alex is calculated using the duration taken for the ultrasonic waves to travel. This was obtained with pulseIn(), which returns the duration in µs, and dividing by 2 to get the one-way duration. The speed of sound (340 m/s) is converted to cm/s by multiplying by 100, and divided by 10^6 to convert it to cm/µs. The final distance will be computed using the following equation (full code with setup in *Appendix 2*):

$$Distance\ (in\ cm)\ = \frac{duration\ in\ \mu s \times 340}{2 \times 10000}$$

Section 6 Software Design

6.1 Network and TLS

The network used for the operation of our Alex was the hotspot of an iPhone belonging to one of the group members. The two laptops and RPi were all connected to this network, allowing them to transfer information across to one another. To increase network security, TLS was used by generating certs, public and private keys for the server and client using OpenSSL, as per the TLS studio.



6.1.1 Raspberry Pi server and Laptop client

We made use of the RPi to host a server that could receive commands sent over the network. The code for the server was created using the one provided in the Studios as a basis, which was written with the libssl C library that provided the functions needed for TLS. When the MCP was started, the program would start a listener thread which would wait for connections from clients. Once a client is connected to the server, a new worker thread would be created. The worker thread managed the communication between the client, the server and the Arduino by deserializing the packets sent over network and serialising the commands and data into packets to be sent to the Arduino. The translation of commands as characters received from the laptop into commands for the Arduino can be found in *Appendix 5*.

On the laptop, we create a client program which would connect to the RPi server and act as an interface between the operator and the main control program. The client would continuously poll for user input and subsequently serialise the data into a network packet that is sent to the server.

6.1.2 WASD controls

To make the operation of Alex more intuitive, we replaced the original controls where operators would have to enter a direction followed by the distance and speed with WASD and surrounding keys. We also removed the need to press the enter key by using a new function get_keypress() instead of scanf(). The new function edited the terminal state to read in characters pressed without the need for a newline character by disabling canonical mode and echoing using the termios library and restored it afterward. This new version of the controls is similar to the movement controls used by many video games, making it easier for the operator to learn and master. The functions used to implement this are in *Appendix 6.1*.

Command	Key	Description
Forward	w	Sends a move forward command to the RPi, which will then send it to the Arduino. Preset distance and speed are dependent on the ‘gear’.
Backward	s	Sends a move backward command to the RPi which will then send it to the Arduino. Preset distance and speed are dependent on the ‘gear’.
Turn left	a	Sends a turn left command to the RPi which will then send it to the Arduino. Preset angle and speed are dependent on the ‘gear’.
Turn right	d	Sends a turn right command to the RPi which will then send it to the Arduino. Preset angle and speed are dependent on the ‘gear’.
Gear 1	1	This gear is used when precise movement is needed. Sets movement presets to: Distance/Speed: 2cm/70% Angle/Speed: 20 deg/100%
Gear 2	2	This is the default gear, used to balance precision and speed. Sets movement presets to:

		Distance/Speed: 5cm/70% Angle/Speed: 20 deg/100%
Gear 3	3	This gear is used when speed is needed. Sets movement presets to: Distance/Speed: 20cm/100% Angle/Speed: 30 deg/100%
Get data	z	Sends an odometry data request to the RPi, which will then send it to the Arduino. It will subsequently print the received data on the Laptop.
Clear data	x	Sends a reset odometry data command to the RPi, which will then send it to the Arduino.
Colour sensor data	c	Sends a colour sensor data request to the RPi, which will then send it to the Arduino. It will subsequently print the RGB frequency values and the derived colour on the Laptop.
Ultrasonic data	v	Sends an ultrasonic data request to the RPi, which will then send it to the Arduino. It will subsequently print the measured distance on the Laptop.
Mode	m	Changes the mode of operation from WASD controls back to the original "Enter command, distance, speed" format provided.
Quit	q	Disconnects the client and exits the Laptop client program.

Table 2: List of commands and their respective keys

As this method sent commands much faster than the original method, this caused the "BAD MAGIC NUMBER" error from the Arduino occasionally. To prevent this, we implemented an additional layer of check in the client by creating a new variable readyToReceive. This variable would be set to 1 when a RESPONSE_OK was received by the client, letting it know the Arduino was ready to get a new command and set to 0 whenever data was sent to prevent excessive sending. The code used to manage this is in *Appendix 6.2*.

6.2 Colour detection algorithm

To correctly identify the colour of the victims, the colour sensor will first obtain the raw R,G,B frequency values from the colour sensor, which will be stored in the first 3 elements of the data array in the colour data packet. The calibration of the sensor is conducted by obtaining raw frequency values from the colour sensor detecting red, green and white paper at 2, 6 and 10 cm away from the edge of the shield (*appendix 4*), to determine the working range and the optimal distance between the target and Alex to measure colour from. Then it will be compared in the following algorithm using least square approximation, which is elaborated in *Figure 5*. Afterwards, we will be sending over the packet determining which colour is the closest to the actual colour detected, and within the colour packet, we will be sending the colour obtained, with '0' representing RED, '1' representing GREEN and '2' representing WHITE, in the fourth element of the data array. From the RPi, it will decipher

the colour according to the number obtained, and prints out the respective colour. The code used to manage this is in *Appendix 3*.

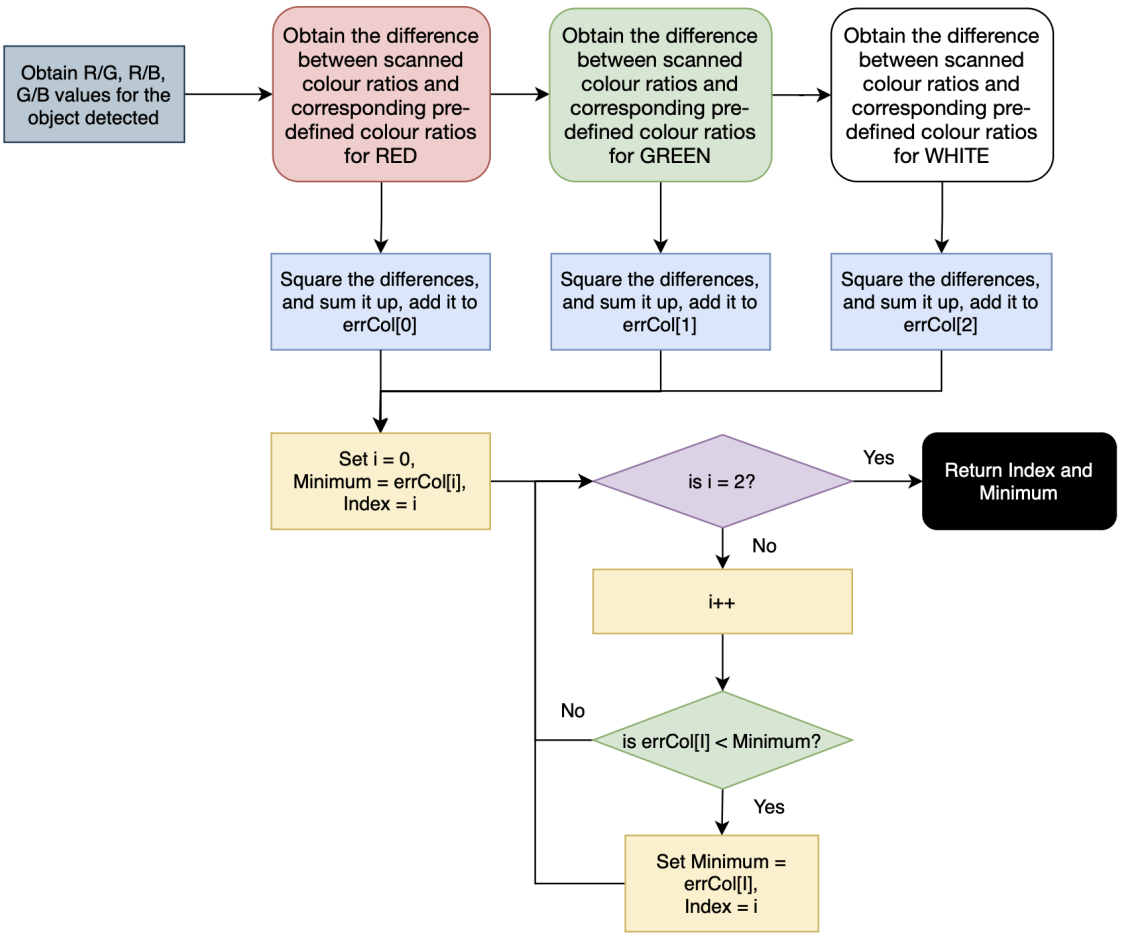


Figure 5 : Colour Detection Logic diagram

6.3 ROS

To visualise data from the LiDAR and to map out the area, we made use of the following ROS programs: RPLidar, Hector SLAM and RViz. The RPi was used as the ROS Master and to launch the RPLidar node since it was connected to the LiDAR. A laptop was used to run Hector SLAM and RViz rather than using RViz through VNC as this reduced the workload for the RPi and made it less likely to lag. A screenshot of our RViz mapping is in *Appendix 6*.

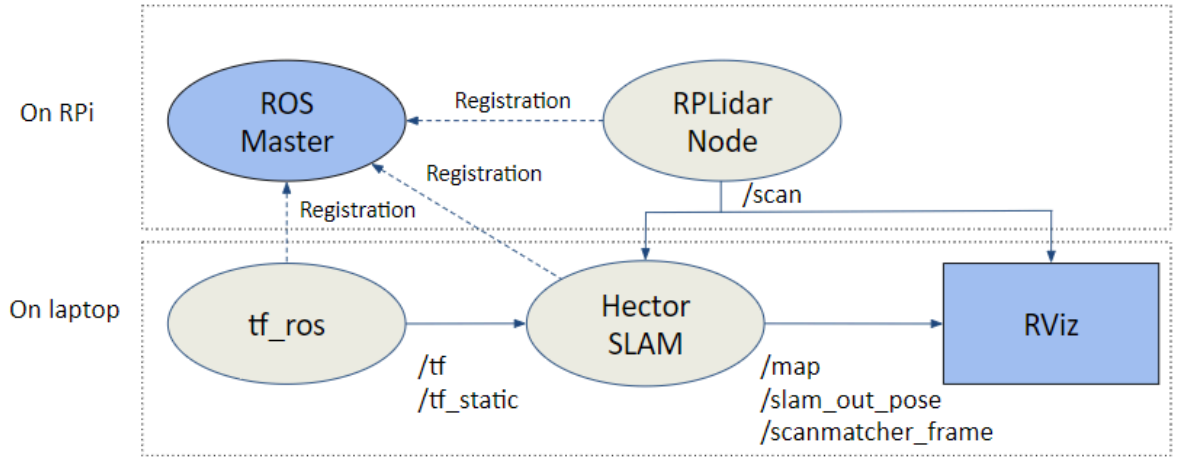


Figure 6: The main nodes and topics of our ROS environment. Some topics such as /base_link have been left out for brevity.

Section 7 Conclusion

7.1 Lessons Learnt

The first important lesson we learnt was to consider the proper termination of our programs, such as the deallocation of memory or setting things back to default. In our case, this meant resetting our terminal state as we had edited it using termios for our laptop client. Originally, we had not considered sudden termination of the program before the terminal state had been reset, as such canonical mode and echoing was left disabled, preventing us from using the terminal normally. This caused us to have to close that terminal session and open a new one whenever the client did not close cleanly. To remedy this, we set up a signal handler to catch signals such as Ctrl+C or the termination signal using the signal.h library to ensure that the terminal would always be reset before the program was terminated. This experience showed us how certain programs could affect our device after termination and that it is good practice to ensure our programs close cleanly regardless of whether it was quit normally, force stopped or crashed.

We also learnt how to improve our productivity as a group by using GitHub. By having all our files on a GitHub repository (less the keys and certs), we were able to work easily on different parts of the code with our own devices before pushing it to origin and merging to main. This also allowed us to work on the RPi's code without having to ssh in or use VNC all the time as we could simply pull from origin on the RPi. Forcing ourselves to use git helped us to get used to the git commands and the process of pushing and pulling etc, a skill which is important to know for future projects.

7.2 Mistakes Made

One mistake we made was that we did not practise navigating with Alex enough before the trial run. In the time leading up to the trial run, we focused mainly on improving Alex rather than practising using it and learning how to interpret and navigate using the SLAM produced map. This made our movement and decision making during the trial run very slow and resulted in us running out of time before parking. For example, when we could not find the "red dummy", we kept searching for it despite time running out and did not prioritise completing the run instead. Learning from this mistake, we invested more time into conducting practice runs and to come up with a priority list (such as complete mapping before worrying about detecting targets) before the final run and were able to complete the run in time.

Another mistake we made was testing the LCD we purchased too late. As we wanted to focus on the main features of our Alex first, we decided to leave setting up the LCD towards the end. However, when we tried to set it up just before the trial run, we realised that it could not properly display text even after double checking the code and the connections. In the end, we decided to remove it for the final run as it was taking up space and adding unnecessary weight. This mistake highlighted the importance of testing components early so that if there is any issue there would be enough time to fix it or source for a replacement.

Section 8 References

1. Kim, Kyung Hoon, and Takenaka Kiyoshi. “Japan Earthquake Death Toll Exceeds 100, with Hundreds Still Missing | Reuters.” Reuters, January 6, 2024. https://www.reuters.com/world/asia-pacific/japan-earthquake-death-toll-tops-100-with-hundreds-still-missing-media-2024-01-06/
2. “10 Impressive Examples of Rescue Robots You Should Be Aware of - Emergency Drone Responder.” The Emergency Drone Responder, September 15, 2023. https://emergencydroneresponder.com/10-impressive-examples-of-rescue-robots-you-should-be-aware-of/
3. “FLIR SkyrangerTM R70 Uas - Cratos Limited Australia.” Cratos, August 18, 2021. https://www.cratos.com.au/uas/flir-skyranger-r70/
4. “Skyranger R70 Datasheet.” Cratos. Accessed April 24, 2024. https://www.cratos.com.au/wp-content/uploads/SkyRanger-R70-Datasheet-A4.pdf
5. “Jueying X20, the Chinese Waterproof Robot Dog from Deep Robotics.” ROBOfluence, October 3, 2021. https://robofluence.com/deep-robotics-unveils-jueying-x20/
6. Robotics, DEEP. “Jueying X20 Robot Dog Hazard Detection Rescue Solution.” Wevolver, April 11, 2023. https://www.wevolver.com/article/jueying-x20-robot-dog-hazard-detection-rescue-solution

Appendix 1 - Bare-metal UART Code

Code used for setting up UART and for reading and writing to the serial port.

```
//Set up the serial connection
void setupSerial()
{
  PRR0 &= ~(1 << PRUSART0);
  UBRR0H = 0;
  UBRR0L = (unsigned char)103; // 103 for 9600 baud
  UCSR0C = (1 << UCSZ01) | (1 << UCSZ00); // Asynchronous USART Mode
  UCSR0A = 0; // Clear the bits of UCSR0A while setting up
}

//Start serial communications
void startSerial()
{
  UCSR0B = (1 << RXEN0) | (1 << TXEN0); // set the receiver and
transmitter bits
}

//Read the serial port
int readSerial(char *buffer)
{
  int count = 0;
  // Read data from the buffer
  while ((UCSR0A & (1 << RXC0))) {
    // Read the received byte and store it in the buffer
    buffer[count++] = UDR0;
  }

  return count;
}

// Write to the serial port.

void writeSerial(const char *buffer, int len)
{
  for (int i = 0; i < len; i++) {
    while (!(UCSR0A & (1 << UDRE0)));
    UDR0 = buffer[i]; // write each byte of the buffer
  }
}
```

Appendix 2 - Ultrasonic Code

Code used for setting up the ultrasonic sensor and for sending the data.

```
#include "constants.h"
#include "packet.h"
#include <math.h>
#define TIMEOUT 4000
#define SPEED_OF_SOUND 340

void setupUltrasonic() // Code for the ultrasonic sensor
{
  DDRA |= (1 << 4); // set trigger pin to output
  PORTA &= ~(1 << 4); // write LOW to trigger pin
  DDRA &= ~(1 << 5); // set echo pin to input
}

uint32_t readUltrasonic() { // detect distance of ultrasonic sensor from
any objects in front of it
  PORTA |= (1 << 4); // emit pulse from ultrasonic sensor
  delayMicroseconds(100); // delay 100 microseconds
  PORTA &= ~(1 << 4); // stop emitting sound from ultrasonic sensor
  double duration = pulseIn(ECHO, HIGH, TIMEOUT); // measure time taken
to detect echo from initial ultrasonic pulse
  double dist = (duration * SPEED_OF_SOUND) / 20000; // calculate
distance of object from ultrasonic sensor, 20000 from 2 * 10000
  return (uint32_t) round(dist); // return distance of object from
ultrasonic sensor in cm
}

void sendDist(uint32_t distance) {
  TPacket distancePacket;
  distancePacket.packetType = PACKET_TYPE_RESPONSE;
  distancePacket.command = RESP_ULTRASONIC;
  distancePacket.params[0] = distance;
  sendResponse(&distancePacket);
}
```

Appendix 3 - Colour Sensor Code

```
#include "constants.h"
#include "packet.h"

void setupColour() {
  DDRA |= 0b00001111;
  DDRK &= ~0b1;
  // Setting frequency-scaling to 20%
  PORTA |= 0b1;
  PORTA &= ~(1 << 1);
}

void evaluateColour(int r, int g, int b, TPacket *colour) {
  float rgbRatioArr[3][3] = {{0.6425624, 0.7596533, 1.1819191},
{1.1747332, 1.0639985, 0.9094113}, {0.9444405, 1.0347558, 1.095375}}; //
contains ratios of r:g, r:b and g:b of the colours of red, green, and
blue respectively.

  float rgNew = (float)r / (float)g; // calculates the r:g ratio of the
colour detected
  float rbNew = (float)r / (float)b; // calculates the r:b ratio of the
colour detected
  float gbNew = (float)g / (float)b; // calculates the g:b ratio of the
colour detected
  float rgbCol[3] = {rgNew, rbNew, gbNew}; // stores the ratios in an
array

  float errCol[3] = {0, 0, 0}; // an array to store the error terms,
i.e. the sum of the absolute differences of the ratios of the target
detected and Red, Green and White paper when compared to the ratios of
red, green, and blue respectively.

// loop through to calculate and store the error terms in the array
  for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
      errCol[i] += (rgbCol[j] - rgbRatioArr[i][j]) * (rgbCol[j] -
rgbRatioArr[i][j]);
    }
  }

  float minimum = errCol[0]; // set minimum error term to first term
  int min_index = 0; // set the minimum index to first colour

// loop through to find the colour with the minimum error term
  for (int k = 0; k < 3; k++) {
    if (errCol[k] < minimum) {
      minimum = errCol[k];
```

```
      min_index = k;
    }
  }
  colour->params[3] = min_index; // store the colour with the lowest
error margins
}

void readColour() {
  TPacket colour;
  colour.packetType = PACKET_TYPE_RESPONSE;
  colour.command = RESP_COLOUR;
  uint32_t frequency = 0;

  // Setting red filtered photodiodes to be read
  PORTA &= ~((1 << 2)|(1 << 3));

  // Reading the output frequency
  frequency = pulseIn(sensorOut, LOW);
  colour.params[0] = frequency;

  delay(100);

  // Setting Green filtered photodiodes to be read
  PORTA |= ((1 << 2)|(1 << 3));

  // Reading the output frequency
  frequency = pulseIn(sensorOut, LOW);
  colour.params[1] = frequency;

  delay(100);

  // Setting Blue filtered photodiodes to be read
  PORTA &= ~(1 << 2);
  PORTB |= (1 << 3);

  // Reading the output frequency
  frequency = pulseIn(sensorOut, LOW);
  colour.params[2] = frequency;

  delay(100);

  evaluateColour(colour.params[0], colour.params[1], colour.params[2],
&colour);

  sendResponse(&colour);
}
```

Appendix 4 - Colour Sensor Calibration

	Red			Green			White		
Dist	**R**	**G**	**B**	**R**	**G**	**B**	**R**	**G**	**B**
2	251	480	403	468	379	408	253	249	223
6	400	577	500	558	512	494	382	394	353
10	487	610	532	605	577	534	454	470	424
Ratio@ dist	**R/G**	**R/B**	**G/B**	**R/G**	**R/B**	**G/B**	**R/G**	**R/B**	**G/B**
2	0.52291667	0.62282878	1.191067	1.2348285	1.14705882	0.92892157	1.01606426	1.13452915	1.11659193
6	0.6932409	0.8	1.154	1.08984375	1.12955466	1.03643725	0.96954315	1.08215297	1.11614731
10	0.79836066	0.91541353	1.14661654	1.0485268	1.1329588	1.08052434	0.96595745	1.07075472	1.10849057
Average	0.67150607	0.77941411	1.16389451	1.1243997	1.13652409	1.01529439	0.98385495	1.09581228	1.11374327

Appendix 5 - Server Code

Appendix 5.1 Network packet types

```
typedef enum
{
  NET_ERROR_PACKET=0,
  NET_STATUS_PACKET=1,
  NET_MESSAGE_PACKET=2,
  NET_COMMAND_PACKET=3,
  NET_COLOUR_PACKET=4,
  NET_ULTRASONIC_PACKET=5
} TNetConstants;
```

Appendix 5.2 Server command handling

```
void handleCommand(void *conn, const char *buffer)
{
	// The first byte contains the command
	char cmd = buffer[1];
	uint32_t cmdParam[2];

	// Copy over the parameters.
	memcpy(cmdParam, &buffer[2], sizeof(cmdParam));

	TPacket commandPacket;

	commandPacket.packetType = PACKET_TYPE_COMMAND;
	commandPacket.params[0] = cmdParam[0];
	commandPacket.params[1] = cmdParam[1];

	printf("COMMAND RECEIVED: %c %d %d\n", cmd, cmdParam[0], cmdParam[1]);

	switch(cmd)
	{
		case 'f':
		case 'F':
			commandPacket.command = COMMAND_FORWARD;
			uartSendPacket(&commandPacket);
			break;

		case 'b':
		case 'B':
			commandPacket.command = COMMAND_REVERSE;
			uartSendPacket(&commandPacket);
			break;

		case 'l':
```

```
            case 'L':
                  commandPacket.command = COMMAND_TURN_LEFT;
                  uartSendPacket(&commandPacket);
                  break;

            case 'r':
            case 'R':
                  commandPacket.command = COMMAND_TURN_RIGHT;
                  uartSendPacket(&commandPacket);
                  break;

            case 's':
            case 'S':
                  commandPacket.command = COMMAND_STOP;
                  uartSendPacket(&commandPacket);
                  break;

            case 'c':
            case 'C':
                  commandPacket.command = COMMAND_CLEAR_STATS;
                  commandPacket.params[0] = 0;
                  uartSendPacket(&commandPacket);
                  break;

            case 'g':
            case 'G':
                  commandPacket.command = COMMAND_GET_STATS;
                  uartSendPacket(&commandPacket);
                  break;

            case 'u':
            case 'U':
                  commandPacket.command = COMMAND_COLOUR;
                  uartSendPacket(&commandPacket);
                  break;

            case 'i':
            case 'I':
                  commandPacket.command = COMMAND_ULTRASONIC;
                  uartSendPacket(&commandPacket);
                  break;

            default:
                  printf("Bad command\n");

      }
}
```

Appendix 6 - Client Code

Appendix 6.1 Functions used for WASD controls

```
#include <signal.h>
#include <termios.h>

void signal_handler(int signum) {
    printf("Received signal %d. Restoring terminal settings...\n",
signum);
    tcsetattr(STDIN_FILENO, TCSANOW, &original_term_state);
    exit(signum);
}

// Read a single character from terminal without a newline
// character. Done by modifying terminal attributes.
char getKeypress() {
  char key = 0;
    struct termios term_state = {0};
    // Save original terminal settings
    tcgetattr(STDIN_FILENO, &original_term_state);
    // Set up signal handler to restore terminal settings on signals
    signal(SIGINT, signal_handler);
    signal(SIGTERM, signal_handler);
    // Copy original terminal settings to modify
    term_state = original_term_state;
    term_state.c_lflag &= ~ICANON; // Temporarily disable canonical mode
    term_state.c_lflag &= ~ECHO; // Temporarily disable echo
    term_state.c_cc[VMIN] = 1;
    term_state.c_cc[VTIME] = 0;
    // Set modified terminal settings
    tcsetattr(STDIN_FILENO, TCSANOW, &term_state);
    // Read a single character
    if (read(STDIN_FILENO, &key, 1) < 0) {
        perror("read()");
        // Restore terminal settings in case of error
        tcsetattr(STDIN_FILENO, TCSANOW, &original_term_state);
        exit(EXIT_FAILURE);
    }
    // Restore original terminal settings
    tcsetattr(STDIN_FILENO, TCSANOW, &original_term_state);

    return key;
}

// Control the bot with new and improved WASD movement!
void commandBetter(void *conn, int *quit)
{
```

```
char ch = getKeypress();
char buffer[10];
int32_t params[2];
buffer[0] = NET_COMMAND_PACKET;

switch (ch)
{
  case 'w': // go forward
    buffer[1] = 'f';
    params[0] = distance;
    params[1] = speed;
    memcpy(&buffer[2], params, sizeof(params));
    sendData(conn, buffer, sizeof(buffer));
    break;
  case 's': // go backwards
    buffer[1] = 'b';
    params[0] = distance;
    params[1] = speed;
    memcpy(&buffer[2], params, sizeof(params));
    sendData(conn, buffer, sizeof(buffer));
    break;
  case 'a': // go left
    buffer[1] = 'l';
    params[0] = angle;
    params[1] = 100;
    memcpy(&buffer[2], params, sizeof(params));
    sendData(conn, buffer, sizeof(buffer));
    break;
  case 'd': // go right
    buffer[1] = 'r';
    params[0] = angle;
    params[1] = 100;
    memcpy(&buffer[2], params, sizeof(params));
    sendData(conn, buffer, sizeof(buffer));
    break;
  case 'z': // get stats
    buffer[1] = 'g';
    params[0] = 0;
    params[1] = 0;
    memcpy(&buffer[2], params, sizeof(params));
    sendData(conn, buffer, sizeof(buffer));
    break;
  case 'x': // clear stats
    buffer[1] = 'c';
    params[0] = 0;
    params[1] = 0;
    memcpy(&buffer[2], params, sizeof(params));
    sendData(conn, buffer, sizeof(buffer));
    break;
```

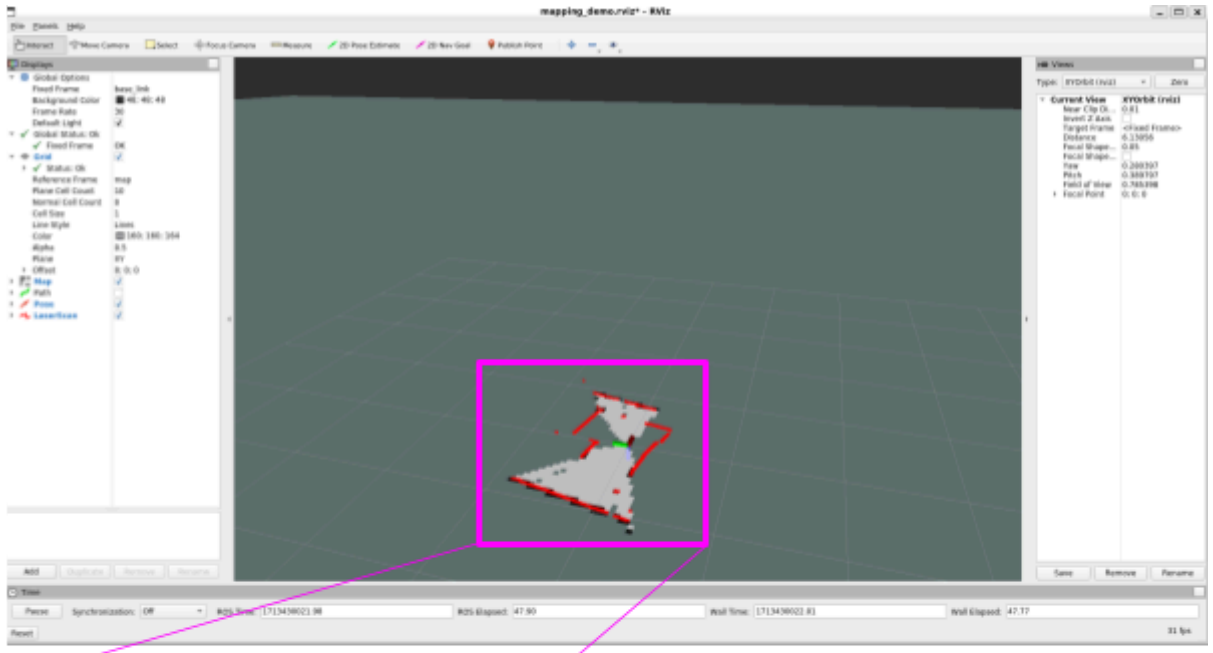
```
    case 'c': // get coloursensor values
      buffer[1] = 'u';
      params[0] = 0;
      params[1] = 0;
      memcpy(&buffer[2], params, sizeof(params));
      sendData(conn, buffer, sizeof(buffer));
      break;
    case 'v': // get ultrasonic values
      buffer[1] = 'i';
      params[0] = 0;
      params[1] = 0;
      memcpy(&buffer[2], params, sizeof(params));
      sendData(conn, buffer, sizeof(buffer));
      break;
    case 'm': // change mode
      printf("Changing mode to original controls...\n");
      mode = 0;
      break;
    case '1': // gear 1
      printf("Swapping to gear 1...\n");
      distance = 2;
      speed = 70;
      angle = 20;
      break;
    case '2': // gear 2
      printf("Swapping to gear 2...\n");
      distance = 5;
      speed = 70;
      angle = 20;
      break;
    case '3': // gear 3
      printf("Swapping to gear 3!!!\n");
      distance = 20;
      speed = 100;
      angle = 20;
      break;
    case 'q': // quit
      *quit = 1;
      break;
    default:
      printf("mannnnn its like 10 keys how'd u get it wrong...\n");
  }
}
```

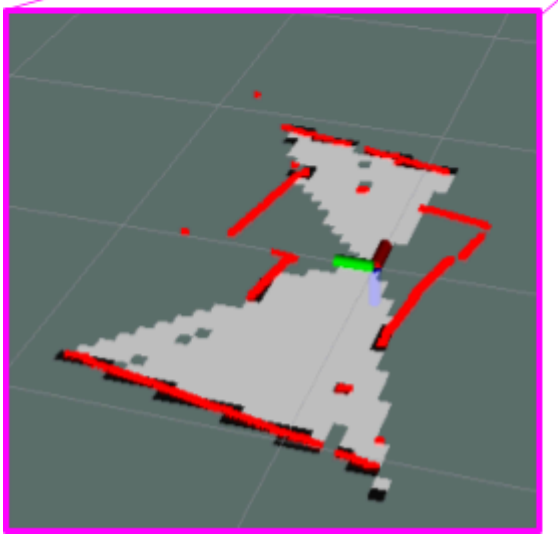
Appendix 6.2 Measure to prevent BAD MAGIC NUMBER:

```
int readyToReceive = 1; //1 if Arduino is ready to receive command
```

```
//inside the switch statement handling responses
  case RESP_OK:
    readyToReceive = 1;
    printf("Command / Status OK\n");
    break;

//inside the function to send data
  if (readyToReceive)
  {
    if (networkActive)
    {
      printf("\nSENDING %d BYTES DATA\n\n", len);
      c = sslWrite(conn, buffer, len);

      if (c < 0)
      {
        perror("MAN! Error writing to server: ");
      }

      networkActive = (c > 0);
      readyToReceive = 0; // set to 0 while waiting for Arduino to say
OK
    }
  } else {
    printf("HOL UP! Arduino not ready\n");
  }
```

Appendix 7 - SLAM visualisation





On top of the SLAM visualisation, we added LaserScan (the red dots) and Pose. For Pose, we edited the length of the red line at the front and the green line at the side to end at the edges of our bot to help avoid collisions.

"The Alex knows where it is, because it knows where it isn't"