

CG2111A Engineering Principle and Practice
Semester 2 2022/2023

"Alex to the Rescue"
Final Report
Team: B02-5B

Name	Student #	Sub-Team	Role
Ayagari Mukund	A0255413L	Software	Movement calibration
Guok Qi-En Avril	A0264770Y	Software	LiDAR implementation
Dao Trong Khanh	A0255826W	Software	Colour sensor
Jackie Neo Seh Jie	A0254470H	Hardware	Circuit Design

Content Page

Section 1 Introduction **3**
Section 2 Review of State of the Art **3**
Section 3 System Architecture **5**
Section 4 Hardware Design **5**
Hardware Components: 5
Additional Components: 5
Section 5 Firmware Design **7**
High level algorithm on the Arduino Uno 7
Communication protocol 7
Section 6 Software Design **8**
High level algorithm 8
Colour Detection 10
Section 7 Lessons Learnt - Conclusion **11**
References **13**
Appendix **14**

Section 1 Introduction

Around 45,000 people are killed per year on a global scale due to natural disasters which are responsible for 0.1% of global fatalities during the last decade. The first 72 hours following a tragedy are essential, as the activities and responses of the search and rescue team during this time can determine whether the survivors live or die.

As such, we developed **'Alex'**, a robot vehicle with search and rescue functionalities that can be controlled remotely. The function of Alex is to navigate an environment in the aftermath of a disaster to identify survivors by conducting search and rescue functionalities. Alex is expected to map the given simulated environment and relay it back to the operator throughout the operation.

To be able to carry out these functions, there are certain desired system functionalities that Alex needs to have, namely tele-operation, environment mapping, and movement control. Tele-operation is used to control Alex's movement from a laptop, acting as a remote control. To enable it, the system is composed of 2 main components, a Master Control Program (MCP) that runs on Raspberry Pi, and an Arduino board which controls the movement of Alex. MCP receives sensory data from the Light Detection and Ranging (LiDar) sensor on Alex and sends data back to the operator in the form of an environment map. The Simultaneous Localization and Mapping (SLAM) algorithm is used to map the environment accurately.

Green and red objects representing victims will be placed randomly in the simulated environment of $3m^2$ with 2-4 areas segregated by walls of at least 18 cm in height. Dummy objects with unknown colours will similarly be present to test if Alex is able to differentiate victims from dummies. Using a colour sensor, Alex will need to identify either green, red, or dummy, and relate this information back to the operator.

Section 2 Review of State of the Art

Search and rescue operations can be helped by robotic platforms that can be operated remotely to locate and rescue victims in dangerous environments where it may not be possible or safe for people to go.



Unmanned aerial vehicle (UAV) Predator: This platform for aerial search and rescue can be operated remotely by a person. It has cameras, sensors, and other specialist equipment that can find and follow people or objects from the air. It has an engine, a fuselage, wings, a tail, and a control station that the driver uses to operate the vehicle remotely.

Strengths: It can operate in a variety of environments and quickly cover large areas. As a result, it's ideal for search and rescue missions that require aerial surveillance. Its cameras and sensors generate high-quality images and data that can be used to locate victims or

identify potential hazards. It can also be outfitted with tools like infrared cameras and sensors, which allow it to detect heat signatures and track individuals even in low light or at night.

Weaknesses: It has a limited range and battery life, making it difficult to travel long distances or operate continuously. It also relies on clear weather conditions and may be hampered by high winds or other environmental factors.

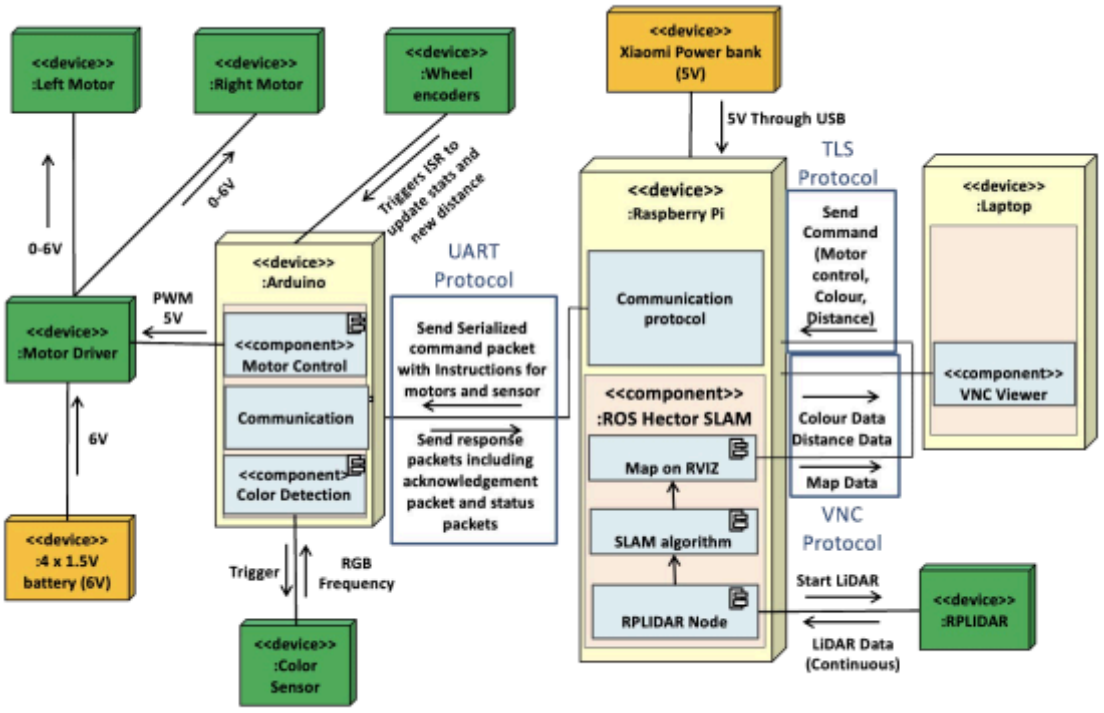


TALON: It is a land-based search and rescue robot that can traverse rough terrain and find victims in dangerous environments. It has a variety of sensors, such as cameras, microphones, and gas detectors, along with tools such as a claw arm and a cutting tool. The TALON has tracks for mobility and a control station from which the operator can control the robot remotely.

Strengths: The TALON can operate in a variety of environments, including disaster zones, chemical spills, and explosive environments, without endangering human operators. Its tank-like tracks allow it to traverse rough terrain and climb over obstacles. Furthermore, its sensors are capable of detecting and analyzing data such as sound, temperature, and gas levels.

Weaknesses: Because the TALON is large and has limited mobility, it may be difficult to access certain areas. It is also controlled by its operator, which may cause delays or errors in the search and rescue process. Furthermore, its tools are limited and may not be adequate for all rescue scenarios.

Section 3 System Architecture



Section 4 Hardware Design

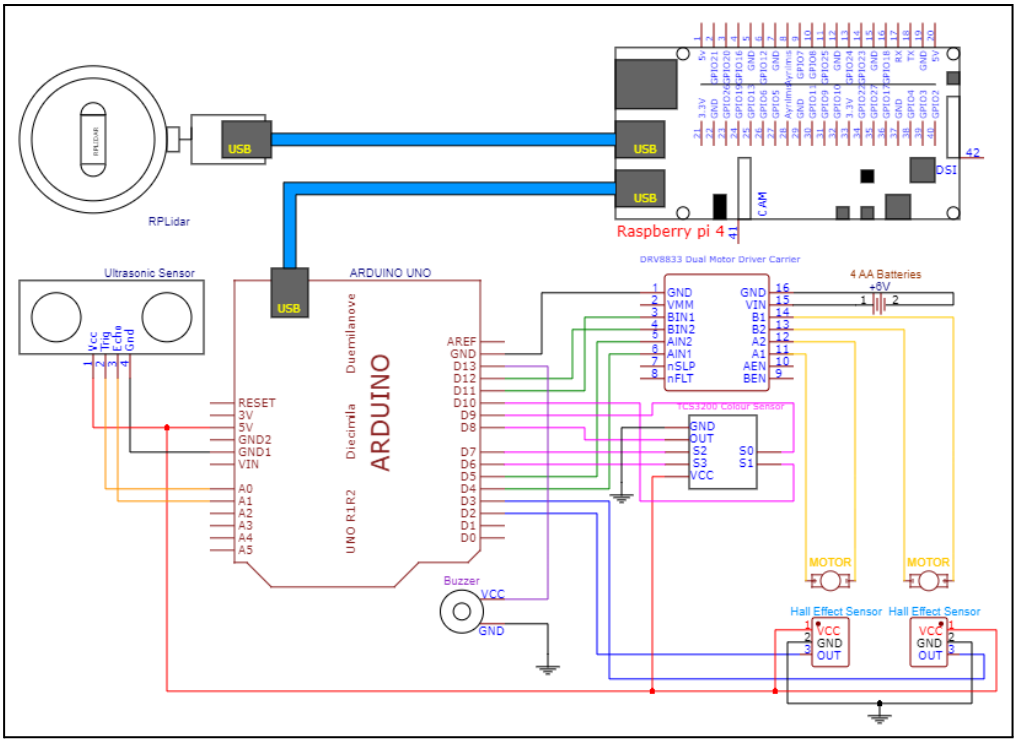
Hardware Components:

1. Raspberry Pi: Powered by a 5V power bank. Responsible for sending commands, and receiving and processing information from Arduino and LiDar. Communicates through UART communication.
2. Motors: Powered by 4 x 1.5v AA batteries. Responsible for the movement of Alex.
3. Arduino: Powered by the Raspberry Pi. Responsible for the control of the motors and receiving and processing data from the colour sensor.
4. LiDar: Powered by the Raspberry Pi. Sends data to Raspberry Pi through UART.
5. Colour Sensor: Connected to the Arduino. Detects colour and sends frequency back to Arduino to process the colour.

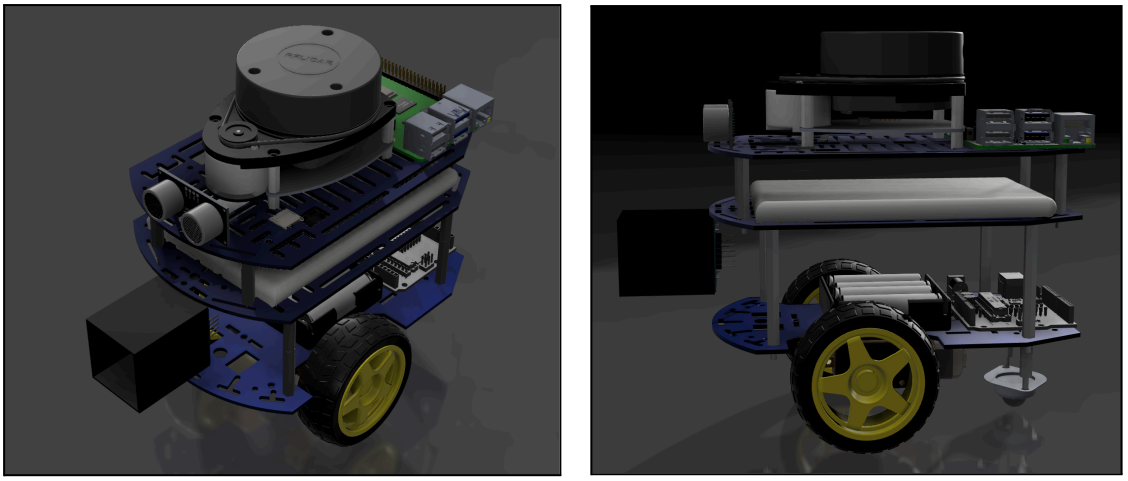
Additional Components:

Buzzer - Powered by Raspberry Pi. Buzzes a tone according to what was detected by the colour sensor. For example, buzzes with a short delay of 100 milliseconds when it detects an enemy (red) resulting in an urgent and threatening sounding tone, and a longer delay of 1000 milliseconds when it is a victim (green), which sounds like a SOS tone. It does not buzz when it is a dummy. In the real world this would alert on the ground rescue troops of the location of the impostor and victims without having to wait on the teleoperator for updates. Moreover, the impostor tone was modelled after information we learnt during National Service about Urban Operation Close Quarter Battle (UOCQB). The urgent tone stuns the impostor allowing own forces to have a faster reaction time in Urban Combat.

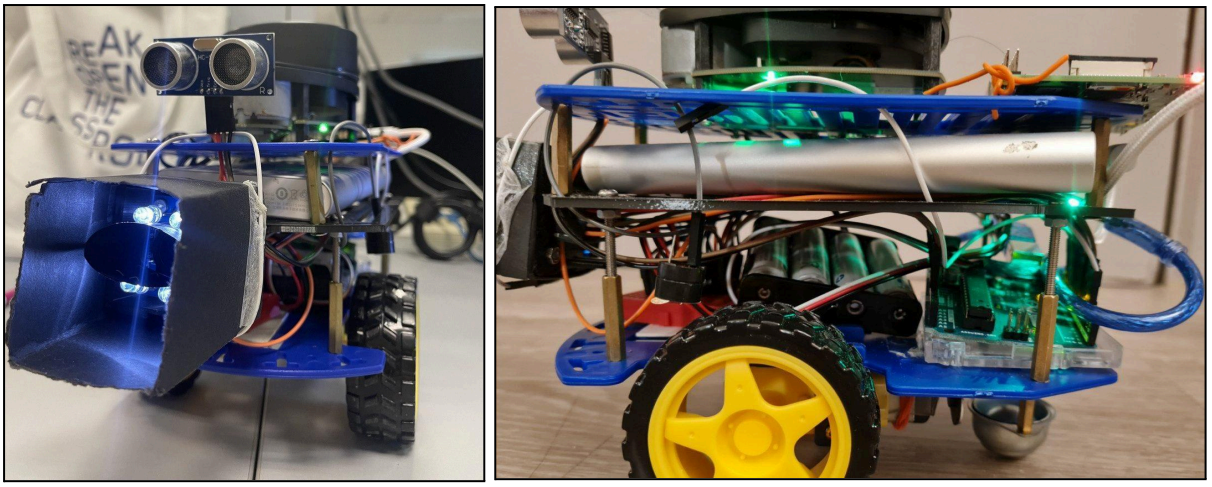
Ultrasonic Sensor - Powered by Raspberry Pi. Measures the distance between the front of Alex and the nearest object in front of it.



Component Schematic used for Alex



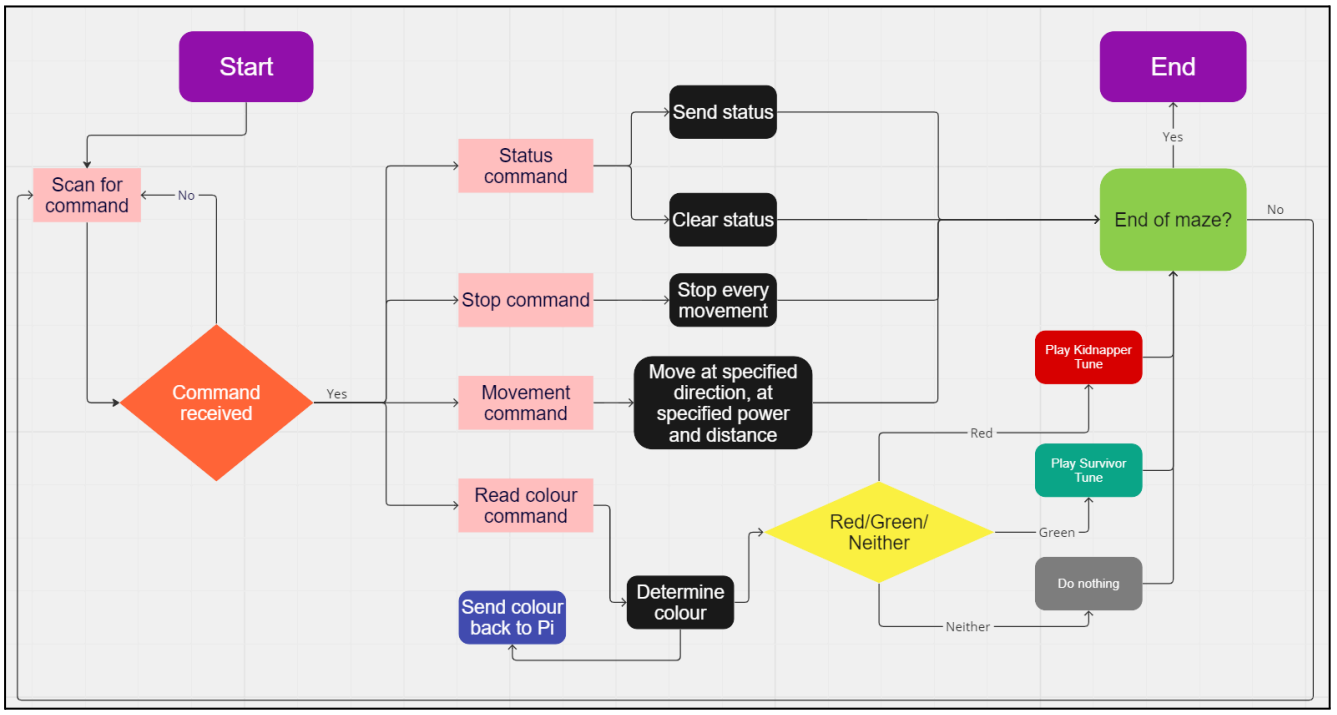
CAD Design of Alex using Fusion 360 (Top Front Right View and Side View)



Photographs of Alex (Same View)

Section 5 Firmware Design

High level algorithm on the Arduino Uno



Communication protocol

Messages and answers are transmitted in packets of 100 bytes each:

1. packetType (char - 1 byte): describes the type of the packet
2. command (char - 1 byte): guides Alex how to handle the information in the packet
3. dummy (array of 2 char - 2 bytes): pads to the nearest divisible by 4, as the RPi reads 4 bytes at a time
4. data (array of 32 char - 32 bytes): Information to be sent in char type
5. params (array of 16 32-bits integers - 64 bytes): Information to be sent in integer type

When sending data from the RPi to the Arduino, the operator input will be written on the "command" char and "params" integers which indicate the movement or activity Alex should execute. After receiving a packet from the RPi, Arduino will compare checksum and magic number to ensure that the packet is received properly. If either checksum or magic number is incorrect then Arduino will send a bad checksum/ magic number packet back to RPi.

1. There are 4 movement commands: forward (f), reverse (b), turn left (l), or turn right (r), which tell the Arduino to activate certain motor pins to move Alex in the desired direction.
    a. Forward (f) and Reverse (b): There are 2 values to be considered, which indicate the distance (in cm) and power, which will be saved into the params array.
    b. Left (l) and Right (r): There are 2 values to be considered, which indicate the angle (in degrees) and power, which will be saved into the params array.
    c. All 4 of the commands once executed will set the green LED pins to 1 for them to light up and clear the red LED pins to 0 for them to turn off.

    After the desired ticks are reached, motors will stop by calling stop() function. The stop command once executed will set the red LED pins to 1 for them to light up, and clear the green LED pins to 0 for them to turn off.

2. The stop command (s) instructs the Arduino to stop all motors.
3. The "get" command (g) will obtain Alex's status from the Arduino (numbers of ticks for each wheel, distance,…)
4. The "clear stats" (c) command will set the number of ticks and distance moved to 0.

The Arduino will send a packet OK to the RPi to acknowledge that the packet was received correctly. By default, Arduino will send a BadCommand error packet to the RPi if the received command is unrecognisable.

A noteworthy software-related stuff: Bare-metal Programming

We decided to go on with the bare-metal programming. The part in Alex.ino that we modified to Bare-metal is in the **Appendix**

Control: Bare-metal programming allows us to optimise our code for the specific needs of Alex and can help achieve better performance compared to relying on higher-level abstractions or libraries.

Efficiency: By directly manipulating the hardware registers of the microcontroller, we can reduce the overhead and latency associated with using libraries or operating systems and can make Alex more responsive.

Section 6 Software Design

High level algorithm

The main function of Alex is to map out an environment and relay that information back to the operator. The algorithm to do so can be broken down into the following steps:

Step 1: Initialization

At this stage, the Raspberry Pi is connected to a laptop via a USB cable and the laptop will initiate a VNC connection to the Raspberry Pi. The Arduino is then connected to Raspberry Pi, the LiDar, and the motors. After connecting all the hardware components, the software components are initialized by running the MCP, which acts as an interface between the Arduino and the operator, on the Raspberry Pi. It translates command codes from the operator into actual motor movements, moving Alex in the necessary directions.

Function **startSerial()** is invoked to initialise the program to establish serial communication between the Arduino Uno and Raspberry Pi. It uses the UART protocol with 8 data bits, 0 parity bits, 1 stop bit, with a baud rate of 9600 bps.

Step 2: Receiving & carrying out user commands (Main Algorithm)

The operator will send movement commands like move forward, reverse, and turn left/right to Alex through the laptop. These commands are serialised and sent to the Arduino.

The commands are sent in the form of keyboard inputs using Command Packets which will then be translated into actual specific movements of Alex as displayed below:

Command	Keyboard Input (not case sensitive)
COMMAND_FORWARD	F
COMMAND_REVERSE	B
COMMAND_TURN_RIGHT	R
COMMAND_TURN_LEFT	L
COMMAND_STOP	S
COMMAND_CLEAR_STATS	C
COMMAND_GET_STATS	G
COMMAND SENSE COLOUR	I

Upon receiving the commands, the Arduino deserializes and processes it. To ensure the data is transmitted and received correctly, each packet is subjected to 2 checks: checksum, magic number check.

If the packet is invalid, the Arduino will return PACK_BAD. And once the packet is checked to be valid, the checksum is calculated. If the packet is invalid, Arduino will return PACKET_CHECKSUM_BAD, whereas if the packet is valid, the Arduino will return PACKET_OK. Once it is confirmed that the command is valid and received successfully, the Arduino will return the result. Otherwise, a bad packet/checksum will be sent to the Raspberry Pi. It will then wait for the user to input the next command.

To perform the search and rescue operation, and map out the environment, the Hector SLAM algorithm works with the ROS visualisation on Raspberry Pi and LiDar sensors.

a. ***Scan Match***
   By comparing the current and previous LiDar scans, Hector SLAM estimates Alex's current position and orientation. The algorithm then finds the 'best match' between Alex's real-time position and the scans and provides the best estimate of the robot's position.

b. ***Mapping***
   As Alex moves through the simulated environment, SLAM will map the environment using LiDar scans. It combines Alex's estimated position with LiDar scans to generate a 2D map on the operator's laptop using Robot Operating System (ROS) visualisation tools such as Rviz.

   Rviz is a 3D visualisation tool that allows the operator to view Alex's movement in the environment and the map that is generated by SLAM. To start SLAM along with Rviz, the command "roslaunch rplidar_ros view_slam.launch" is typed into the terminal. This command launches the RPLidar ROS package with 'view_slam.launch'.

Colour Detection

There are multiple ways to detect colours by observing the LDR values. However, to increase the accuracy of our colour detection, we decided to detect raw RGB values derived from photodiodes readings and incorporate if-else statements to determine the colours.

We set the frequency scaling of the colour sensor to 20% by setting digital PIN S0 to HIGH and S1 to LOW. The serial communication is initialised at a baud rate of 9600. The colour sensor is set up by defining digital pins S0, S1, and S2 as output pins, and sensorOut as input pin.

The **SendColour()** function is our main function that is used to read and output the colours detected by the colour sensor. To do so, the photodiodes need to be set to be read, and this is done using the values in the table below.

Photodiode	S2	S3
Red	LOW	LOW
Green	HIGH	HIGH
Blue	LOW	HIGH

The **pulseIn()** function is then used to measure the output frequency of the sensor at the respective wavelengths. This function measures the duration of the pulse on a pin and returns the length of that pulse in microseconds. The output frequency is then calculated by dividing the length of pulse by the period of the pulse.

The output frequency for red, green and blue is then mapped to a value between 0 and 255 in the **map()** function. This function takes in the input value, minimum, and maximum values of the input range, and the output value, minimum, and maximum values of the output range and

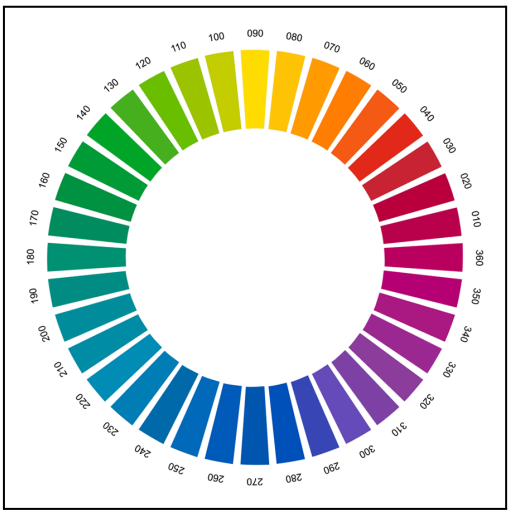
returns a new remapped value. This new value is proportional to the input value but scaled to a new range. The new remapped values of red, green and blue are stored under redColour, greenColour and blueColour respectively.

Next, the **calchue()** function is called to calculate the hue of the detected colour. This function takes in the values of the red, green, and blue frequencies detected by the colour sensor. It then divides each value by 255 to normalise them between 0 and 1. The function then calculates the maximum and minimum values in the array and uses them to calculate the hue value based on the following formulae:

Max	Hue
R	(G - B) / (max - min)
G	2.0 + (B - R) / (max - min)
B	4.0 + (R - G) / (max - min)

The **senseColour()** function is then called to detect the colour using an RGB colour sensor. It reads and stores the output frequency from the photodiodes for each colour channel (red, green, and blue) and remaps its frequency values to a range of 0 to 255 using the **map()** function (as explained above).

The **calchue()** function is then called with the remapped red, green, and blue intensity values to calculate the hue of the detected colour by multiplying the value by 60. This converts the values to degrees on the colour wheel. An example is given below:



Based on the calculated hue value, the code determines which colour was detected. If the hue value is less than 23, it is considered as red, and a message "RED detected" is sent using the **sendMessage()** function. If the hue value is between 132 and 160, it is green, and a message "GREEN detected" is sent. If the hue value falls outside of these ranges, a message "Trolling" is sent.

Section 7 Lessons Learnt - Conclusion

The 2 most important lessons tie in with the greatest mistakes we made. They are:

1. **Mistake: Not switching to WASD commands:** Despite numerous practice runs, we exceeded the time limit in the actual run and were unable to park. Looking back, this was a result of using the default commands for movement which is **Direction** (e.g. F for forward) followed by **Distance** and **Speed** (Angle and speed for turning). During our practice runs, this did not seem to be too much of an issue as we remained under the time limit. In hindsight, one realisation was that we were still a lot slower than other groups who used WASD for their controls (e.g. we took 7 minutes to complete a practice maze that took them 6).

   For the actual run, we had a few issues such as getting stuck at the hump and also exploring the area with the parking space first (instead of at the end). The second issue meant that we had to traverse the entire maze almost twice. Due to this, the time is taken in typing the Direction followed by Distance and Speed each time added up, resulting in us not having enough time.

   Solution: Add WASD as commands. In this case, W and S will be forward and backward respectively and A and D will be left and right. Each key will be considered a fixed distance (or direction) and speed. For example, every time W is clicked, Alex will move 3cm with a speed of 51%. In essence, this is a shortcut to the default command-sending format and will save time, allowing us to finish the maze and park successfully. It would also make manoeuvring easier as it is more mechanical with repeated clicks like in a video game with no computation of distances and angles involved. The reason this works is also because the slam-out pose in our HECTOR SLAM is very accurate and updates our orientation very quickly.

2. **Mistake: Wire connections to Raspberry Pi:** We had 7 collisions through our final run. After the run, we were told that the majority of them were because of the Arduino and LiDAR wires connected to the RPi brushing against the walls, especially when turning. Our LiDAR failed to detect that we were in contact with the wall, and since the wires were on the side, our additional collision measure: the ultrasonic also failed to detect the issue. After the run we inspected the wires and compared it to photos we had taken earlier. Initially, we had wound the wires around the chassis to ensure they did not protrude out. However, the tension had reduced over time causing them to slowly protrude out more and more, something we failed to notice.

   Solution: Turn the RPi: We realised that the solution to this problem was rather easy. By turning the RPi 90 degrees, we would be able to connect the wires from the back

of the chassis instead of the sides. This would ensure that the wires did not brush against the walls when turning, thus reducing our collisions significantly.

Overall, our run did not go as well as expected. Some of the problems were out of our control, like checking the parking area part of the map first. However, we learnt that we need to always think of the worst-case scenario. This project showed us that mishaps and obstacles are common in the engineering world and small shortcomings (such as the wires protruding slightly more) can result in big problems (7 collisions). However, we had a lot of fun building this robot, and the mishaps during the run helped us learn a lot about troubleshooting and the importance of foresight!

References

Ritchie, H., Rosado, P., & Roser, M. (2022, December 7). *Natural disasters*. Our World in Data. Retrieved April 21, 2023, from https://ourworldindata.org/natural-disasters

Vpt, Karampela, E., & Vara, V. (2022, September 2). *Predator RQ-1 / MQ-1 / MQ-9 reaper UAV, United States of America*. Airforce Technology. Retrieved April 21, 2023, from https://www.airforce-technology.com/projects/predator-uav/

Murphy, R. R., Tadokoro, S., Nardi, D., Jacoff, A., Fiorini, P., Choset, H., & Erkmen, A. M. (2008). Search and rescue robotics. Springer Handbook of Robotics, 1151–1173. https://doi.org/10.1007/978-3-540-30301-5_51

What is SLAM (simultaneous localization and mapping) – matlab & simulink. What Is SLAM (Simultaneous Localization and Mapping) – MATLAB & Simulink - MATLAB & Simulink. (n.d.). Retrieved April 21, 2023, from https://www.mathworks.com/discovery/slam.html#:~:text=SLAM%20(simultaneous%20localization%20and%20mapping)%20is%20a%20method%20used%20for,to%20map%20out%20unknown%20environments.

Wiki. ros.org. (n.d.). Retrieved April 21, 2023, from http://wiki.ros.org/rplidar

John, user82991, Rafael, & Stan. (1965, July 1). Color wheel with constant subjective hue gradient (better than CIELAB). Graphic Design Stack Exchange. Retrieved April 21, 2023, from https://graphicdesign.stackexchange.com/questions/115546/color-wheel-with-constant-subjective-hue-gradient-better-than-cielab

Appendix

Bare-metal modification(from left to right):

```
void setupSerial()
{
  Serial.begin(9600);
}
```

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

```
void startSerial()
{

}
```

```
void startSerial()
{
  UCSR0B = 0b00011000;
}
```

```
int readSerial(char *buffer)
{
  int count=0;

  while(Serial.available())
    buffer[count++] = Serial.read();

  return count;
}
```

```
int readSerial(char *buffer)
{
  int count = 0;
  while (UCSR0A & 0b10000000) {
    unsigned char data = UDR0;
    buffer[count += 1] = data;
  }
  return count;
}
```

```
void writeSerial(const char *buffer,
int len)
{
  Serial.write(buffer, len);
}
```

```
void writeSerial(const char *buffer,
int len)
{
  int count = 0;
  while (count < len) {
    while ((UCSR0A & 0b00100000) == 0);
    UDR0 = buffer[count];
    count += 1;
  }
}
```

```
void setupMotors()
{
   pinMode (LF,OUTPUT);
   pinMode (RF,OUTPUT);
   pinMode (RR,OUTPUT);
   pinMode (LR,OUTPUT);
}
```

```
void setupMotors()
{
  TCNT0 = 0; // Timer 0
  TCNT1 = 0; // Timer 1
  TCNT2 = 0; // Timer 2

  DDRD |= (LF | LR);
  DDRB |= (RF | RR);

  TCCR0A |= 0b1;
  OCR0A = 0;
  OCR0B = 0;
```

```
                                        TCCR1A |= 0b1;
                                        OCR1A = 0;
                                        OCR1B = 0;

                                        TCCR2A |= 0b1;
                                        OCR2A = 0;
                                        OCR2B = 0;
                                      }
--------------------------------------  --------------------------------------
void startMotors()                    void startMotors()
{                                     {
                                        TCCR0B |= 0b11; // start Timer 0
                                        TCCR1B |= 0b11; // start Timer 1
                                        TCCR2B |= 0b100; // start Timer 2
}                                     }
--------------------------------------  --------------------------------------
void forward(float dist, float speed) void forward(float dist, float speed)
{                                     {
...                                   ...
  analogWrite(LF, val);                 OCR2A = val;
  analogWrite(RF, val);                 TCCR2A = 0b10000001; //RF
  analogWrite(LR,0);                    OCR0A = val;
  analogWrite(RR, 0);                   TCCR0A = 0b10000001; //LF
}                                     }
--------------------------------------  --------------------------------------
void reverse(float dist, float speed) void reverse(float dist, float speed)
{                                     {
...                                   ...
  analogWrite(LR, val);                 OCR1B = val;
  analogWrite(RR, val);                 TCCR1A = 0b00100001; // RR
  analogWrite(LF, 0);                   OCR0B = val;
  analogWrite(RF, 0);                   TCCR0A = 0b00100001; // LR
}                                     }
--------------------------------------  --------------------------------------
void left(float ang, float speed)     void left(float ang, float speed)
{                                     {
...                                   ...
  analogWrite(LR, val);                 OCR2A = val;
  analogWrite(RF, val);                 TCCR2A = 0b10000001; // RF
  analogWrite(LF, 0);                   OCR0B = val;
  analogWrite(RR, 0);                   TCCR0A = 0b00100001; // LR
}                                     }
--------------------------------------  --------------------------------------
void right(float ang, float speed)    void right(float ang, float speed)
{                                     {
...                                   ...
  analogWrite(RR, val);                 OCR1B = val - 10;
  analogWrite(LF, val);                 TCCR1A = 0b00100001; // RR
  analogWrite(LR, 0);                   OCR0A = val;
  analogWrite(RF, 0);                   TCCR0A = 0b10000001; // LF
}                                     }
```

```
---------------------------------------
void stop()
{
  dir =  STOP;
  analogWrite(LF, 0);
  analogWrite(LR, 0);
  analogWrite(RF, 0);
  analogWrite(RR, 0);
}
```

```
---------------------------------------
void stop()
{
  dir = STOP;
  TCCR0A = 0b00000001;
  TCCR1A = 0b00000001;
  TCCR2A = 0b00000001;
}
```