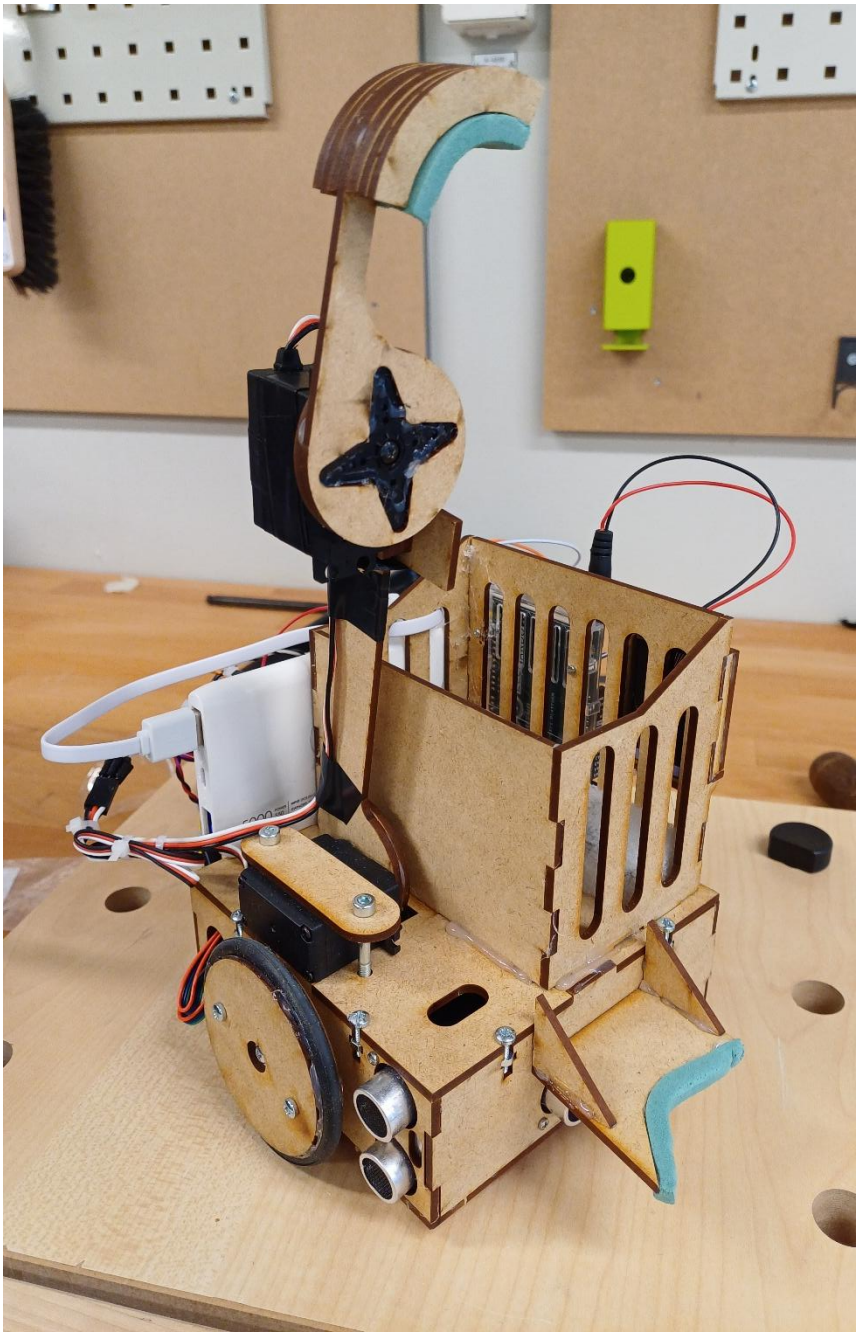
Search and Rescue Robot

Service Robotics TNSN01

Group 5:

Jonathan Giegold, Albin Harrstedt, Georg Lundqvist, Sean Small



Preface

In the beginning we all started getting acquainted with Arduino IDE, the electrical wiring for the components and which libraries could be used when programming the different components. All group members discussed and agreed on how the search and rescue robot should complete its task and what would be needed to achieve that. Once we all were acquainted with the task and what needed to be done the work was split up between everyone.

Building phase:

Sean: modelling, laser cutting and building of the body, installing sensors and motors

Albin: modelling and building gripper, installing servos, programming gripper software

Georg: programming pathing algorithms, electrical wiring and connecting all components to the microcontroller

Jonathan: programming pathing algorithms and the logic between the motors and sensors

Testing phase:

Everyone helped during the testing, finetuning and improving the overall performance of the robot.

Table of Contents

Preface **1**

Introduction **3**

Description of the Robot System/Prototype **4**

Body *4*

Gripper Arm *4*

Wiring *6*

Power supply *7*

Sensors *8*

Line sensor 8

Ultrasound sensors 8

Microcontroller *8*

Navigation *9*

Results at the Robot Demo **10**

Discussion **11**

Gripper design *11*

Pathing algorithms *11*

Reliability *12*

Conclusion **13**

References **14**

Appendix A – Code used for exam **15**

Appendix B – Digital filter code **30**

Introduction

This project involved developing a search and rescue robot capable of finding and carrying an injured person out of a building. This scenario was simulated where a small-scale autonomous robot, with appropriate sensors, had to find three ‘people’ in a maze, and then carry them out of the maze.

As a starting point, each team was given a 2-wheeled robot platform, a microcontroller unit (MCU), a pair of DC-motors, a motor driver, servo motors, ultrasound distance modules, line follower sensors and common electronic components. The robot platform was designed to be easily expandable through laser cut MDF accessories and the system used for carrying the people out of the building was to be designed and built from scratch.

The maze had corridors, at least, 300mm wide and 150mm high with a black line down its centre. The maze contained two caveats: at two squares there were no black lines and there were also two “islands” in the maze. Three injured people were placed inside the maze where one position was predetermined, and the other two positions were randomized immediately before each groups demo. The people consisted of small wooden cylinders and each cylinder was placed on a black line in the centre of a square. The maximum total time to complete the task was 6 minutes and the robot was required to complete this task completely autonomously.

Description of the Robot System/Prototype

Body

The robot body, seen in Figure 1, consists of the main platform and the basket. The main platform was based off the initial robot platform received for the project where the size of the panels was kept. The front and side panels were redesigned in SolidWorks to allow for the ultrasound sensors to be mounted from the inside of the robot. The back panel was redesigned to allow for better cable management and the top panel was redesigned to allow for the basket, servo motor and gripper arm to be installed. The main body was designed so that the bottom and side panels would be permanently glued together while the top panel would only be attached with screws. This allowed for easy access to the inside of the robot whilst keeping the body's rigidity.

For the collection system the group agreed on having a basket on top of the robot and collecting all three cylinders in one go before exiting the maze. This way the robot would only need to traverse the maze once instead of collecting one cylinder at a time and having to navigate the same parts of the maze multiple times. All these new parts were then made by laser cutting 3 mm MDF. A piece of foam was also added at the bottom of the basket to get the cylinders to land in a consistent manner every time.

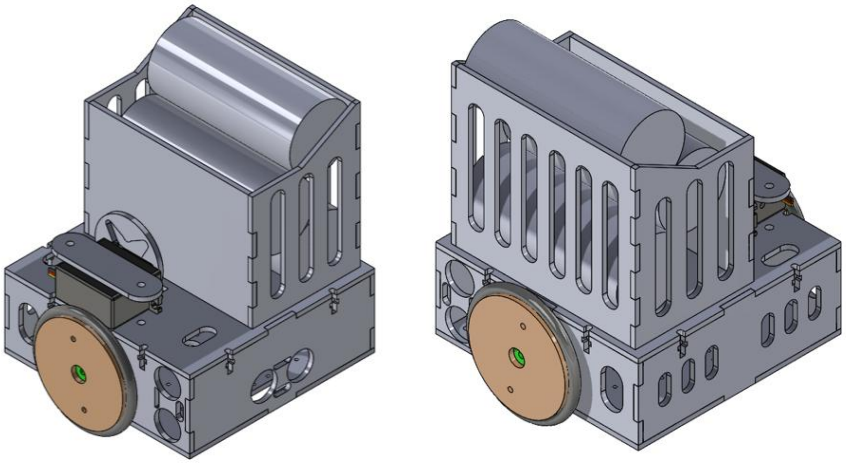


Figure 1: CAD of new robot platform

Gripper Arm

The design process for the gripper arm began by deciding on how it should pick up the cylinder. Having taken inspiration from a robot from a previous year, the group decided to try an approach where the cylinder is first tipped over to the side and then lifted above the body, which hopefully would result in the cylinder sliding out of the gripper and into the basket, as illustrated in Figure 2.

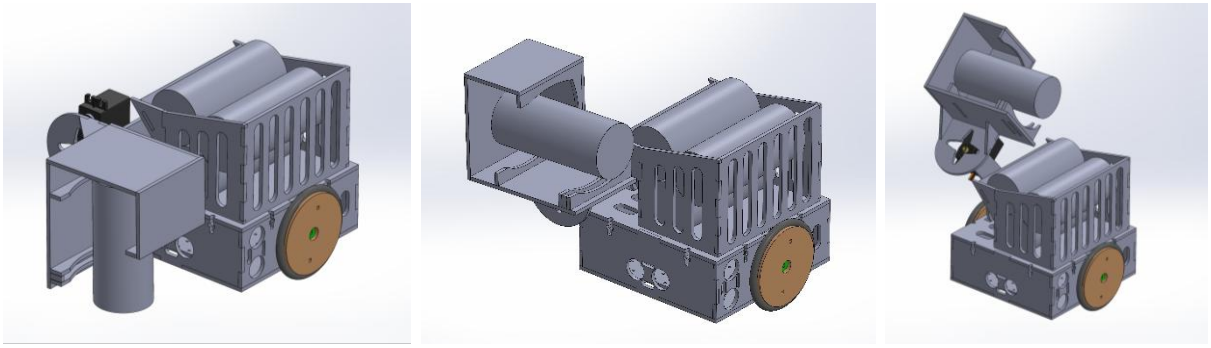


Figure 2: CAD illustration of the prototype pick up process

Prototyping resulted in the three iterations seen in Figure 3. This concept was ultimately abandoned due to reasons which will be touched upon in the discussion.

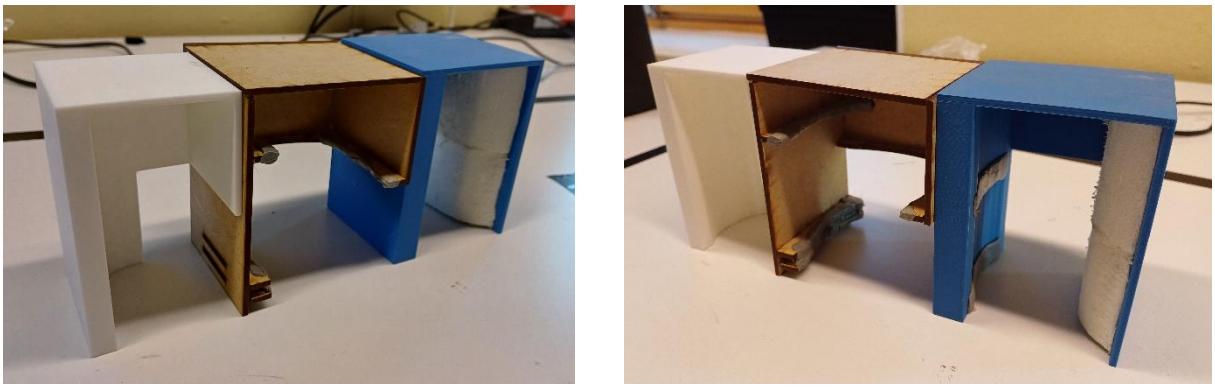


Figure 3: Three iterations of the gripper prototype

The final design utilises a more conventional approach, in which the gripper squeezes the cylinder against the arm itself; this being shown in Figure 4.

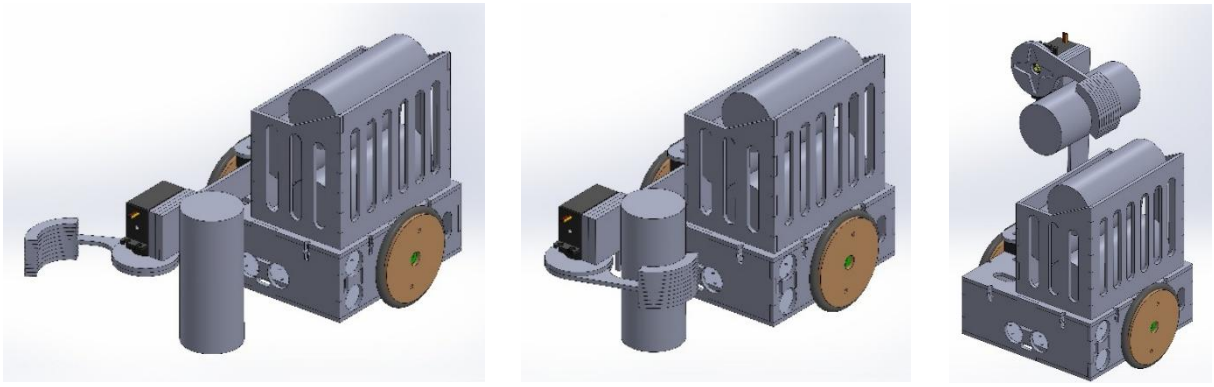
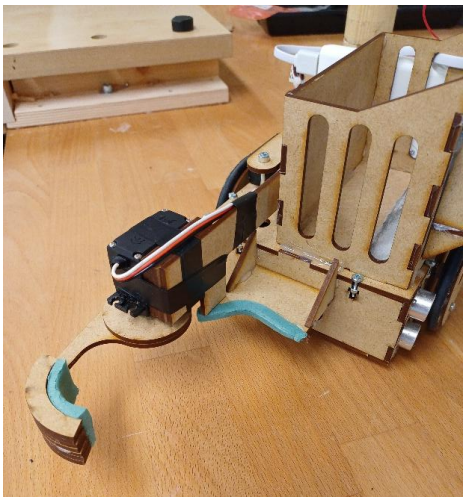


Figure 4: CAD illustration of the final pick-up process

Figure 5 shows the physical gripper arm in raised as well as lowered position. This design has been reliable and maintained structural rigidity, despite the use of electrical tape. Foam was added to the gripper to increase friction between it and the cylinder, in addition to allowing for some give when squeezing the cylinder. Lastly, a cylinder guide (the piece with foam on the front) was added after testing since the arm sometimes would hit the cylinder on the way down.



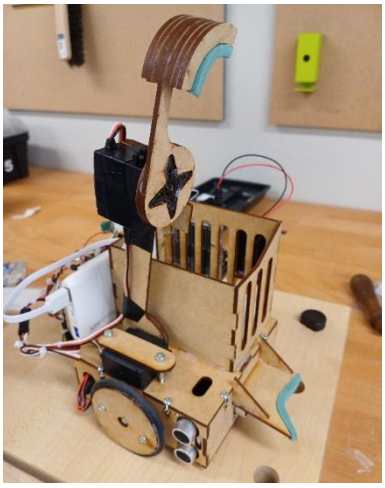


Figure 5: Final gripper arm design

As seen above, the gripper arm utilizes two servos, each being represented in the code by a Servo object imported from the Arduino Servo library (Margolis, 2024). The gripper file (seen in Appendix A) is split into two functions, the first being gripperSetup(). This function is run during the setup and simply attaches the servos to their respective digital pins on the Arduino, as well as setting them to their standby angle, which can be seen to the right in Figure 5. The second function gripAndRelease() does the following: lowers the arm (to the left position seen in Figure 5), closes the gripper to grab the cylinder, raises the arm to 15 degrees beyond the standby angle, opens the gripper to release the cylinder, and finally returns to the standby angle. This function is called whenever the forward-facing ultrasound sensor detects anything within 5 cm, which must be a cylinder (given that the robot has not gone of path).

Wiring

The purpose of the wiring is to connect the MCU, servos, sensors, and power supply correctly and in an orderly fashion. There are a lot of wires present in the robot, hence thought must be put to keep them organised. This is done through two methods, colour coding, as seen in Table 1, and grouping using zip ties and designated holes in the body of the robot, as seen in **Fel! Hittar inte referenskälla.**.

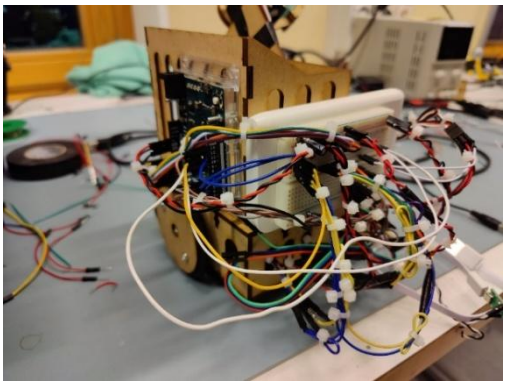


Figure 6: Picture of the wiring on the outside of the robot

*Table 1: Colour scheme for the wiring**

Red	Black	Yellow	Blue	White	Beige	Gray
Positive power	Common ground	Trigger pins for ultrasound sensor	Echo pins for ultrasound sensors	Gripper servo control	Motor1A and Motor2A	Motor1B and Motor2B

**The line sensor deviated from this standard due to needing too many cables*

Electromagnetic compatibility (EMC): EMC is unlikely to pose a significant issue. However, due to the variety of signals transmitted through the cable, potential problems could arise through unwanted noise in sensor cables. To mitigate the possibility of such issues, the power and ground cables were twisted together (Wikipedia, 2024).

Breadboard: A breadboard was used as a hub where all the components can speak to each other. The breadboard was taped on the power bank due to their similar size and packaging proximity. The main advantages of using the breadboard, instead of just wiring everything together which would also work, is that it allows for efficient power distribution using the plus and minus rails on the sides. One side of the breadboard is a 5 V supply from the Arduino board which powers all sensors, and the other side is also 5V but from the power bank, this is due to the heavy current load from the motors and gripper sensors, which the power bank can supply. The other reason to use the breadboard is flexibility, thanks to the breadboard we can rapidly add components, swap a broken component, change pins and more. An example of this is the button, which was added after testing began, which would have been a challenge without the breadboard. A problem encountered with the breadboard is due to it being mounted sideways is that the cables can come loose, which is mitigated by the zip ties, but checks between tests are needed to ensure that the connections are robust.

Pinout: To keep track of the pins on the Arduino Miro was used as a common image editor where everyone can allocate pins for their servos, sensors and motors. See Figure 7 for pinout in Miro.

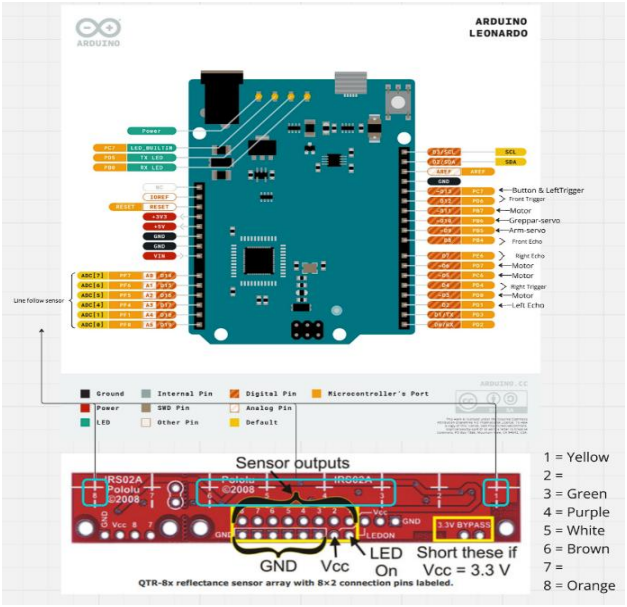


Figure 7: The image editor Miro being used to keep track of allocated pins

Power supply

As shortly mentioned in the breadboard section, we use two different batteries for the robot, one 9-volt battery and a 5-volt power bank. The 9-volt battery only power the Arduino, which in turn power the sensors of the robot. While the power bank power the motors and gripper. The differentiation is made due to the high current draw of the motors and servos which only the power bank can supply. The power bank can supply 4 amps achieving a power of 20 watts.

Both power sources needed to be mounted, and this was originally handled by electrical tape for the 9-volt battery. However, after having to swap the 9-volt battery multiple times, it was decided to design a small battery holder to make the swapping process easier. This was manufactured using MDF and glued to the side of the basket. As for the power bank, being one of the heavier components, it was glued to the back of the basket to keep the robot from tipping forward. It was also laid on its side to keep an even weight distribution across the body.

Sensors

The sensors used on the robot are one infra-red reflectance sensor array used to detect the line on the floor and three ultrasound distance sensors. The purpose of the sensor array is to detect the line relative to the robot and the purpose of the ultrasound sensors is to detect the walls and whether there is a person in front of the robot or not. The encoders of the motors were considered for odometry when building the robot. However, testing showed that the previously mentioned sensors were sufficient and in an effort of keeping the design simple, the encoder signals were omitted.

Line sensor

The line sensor was mounted to the underside and at the very front of the robot using screws. It was also centred ensuring that it could be efficient when used for line following and detecting turns and intersections. There was a hole for the pins of the sensor to allow for the cables to be drawn through the inside of the robot. A notable detail for the mounting of the line sensor is the distance above the ground, according to the datasheet it should be below 6 mm above the ground (Pololu Corporation, 2014).

The Arduino uses the “QTRsensor” library (Pololu, 2019) to read the data from the line sensor. Due to only having six analog inputs on the Arduino lead to limited use of the line sensor. That is, using the four sensors in the centre for line following control and the two most outer sensors to detect intersections and corners. The output of the sensor is six values between 0 and 1000 and a value that estimates the line position relative to the sensors.

Before using the sensor, it needs to be calibrated for the environment, showing it both the lightest and darkest surface that it will encounter. This is done by allowing the robot to spin in place for 5 seconds actively calibrating. To ease testing, this spin is triggered by pressing the button on the breadboard.

Ultrasound sensors

Using the holes cut out in the front and side panels the ultrasound sensors could easily be installed from the inside of the robot using screws. This allowed for the robot to be more compact and the cables to be drawn inside the robot.

The Arduino uses the “NewPing” library (Eckel, 2023) to control the ultrasound sensors. This library makes it simple to use the sensors with the “ping” function. This function sends a short pulse to the “trig” pin, which triggers the sensor to seen eight ultrasonic pulses which bounce of objects in front of the sensor. These are then received, and the time is used to calculate the distance to the object in front which is sent to the “echo” pin.

Microcontroller

The microcontroller used for the robot was an Arduino Leonardo (Arduino, 2024). The microcontroller was mounted on the outside of the basket to keep it close to the breadboard and minimize the distance that the cables had to be drawn. It was mounted using screws to allow for the Arduino to be removed easily in case it would be moved or needed a replacement.

Several variables were initialized as well as the array of line sensor values. The center position was set to 2500 and line threshold set to 650 based on previous testing. Then enums were defined for Orientation and Linetype.

Lastly configuration for the motors and ultrasound sensors were made and the array of movements through the maze were defined. In the setup the button state is read within a while loop so that the program does not follow through with the setup until the button is pressed. After configuration of the sensors and configuration of the gripper the setup was done.

The main loop is divided into three parts: the detection and rescue of cylinders, the reading of the sensors and categorization of linetype, and lastly a switch statement with different actions.

Navigation

To navigate the maze the robot used a pair of DC-motors, wheels and a motor driver. The DC-motors where mounted to the side panels inside the robot and the wheels were screwed on from the outside. The DC-motors were then connected to the motor driver to allow for easier control over the motor speeds and spin direction. Since the robot only had two wheels a ball caster was also installed at the back of the robot to ensure it would not drag along the ground.

The navigation in the main loop begins with reading the position of the line sensors with qtr.readLineBlack(sensorValues) and then calculating the linetype based on which sensors are active. After the three ultrasound sensors have been read, the switch statement of the different linetypes is prompted, and a straight linetype calls on the function followLine(position) which sets the motor speed using a PD controller. Through tuning a proportional coefficient of 0.1 was found to be suitable for stable line following. However, some oscillations of the robot were still observed and to solve these the derivative part was added, which disincentivises the error of the robot position from growing. Here a suitable value of the coefficient for the derivative part was found to be 0.25. Normally an integral part is used for full PID-control, but the robot showed no signs of any stationary error (which is reasonable due to the natural integration of velocity to position) and integral-action often induces oscillations due to integral wind-up.

For the left and right linetype-turn there is first a check for a wall to the opposite side of the direction of the turn, using the ultrasound sensor. If this condition is true, the loop breaks, restarting the entire loop. This is done to give the robot another chance to get better readings from the sensors. If the condition was not fulfilled however, comes next another check for a large distance from the front ultrasound sensor. The sensor value is read three times and then averaged for extra validity. If the condition still holds true, then it means the ultrasound sensors overrule the line sensors decision of linetype, calling handleIntersection(), which iterates through the movements array and calls on the corresponding driver function. If neither of these conditions hold it is assumed that the linetype is correct and a left- or right-turn driver function is called. There is also the regular intersection linetype which when detected only calls on handleIntersection().

Lastly if no line is detected then all ultrasound sensors are read once more and if the condition holds that the sensors detect walls in all three directions, the robot turns around. If the number of turns (index in movements array) is as expected, a hardcoded left turn is made. Which is also done two more times, when the robot enters the no line intersection.

Results at the Robot Demo

The major milestones of the robot project testing are documented in Table 2 below. The time before the first labyrinth test was spent developing code, building the robot and unit testing single parts of the system one by one. The final hardware of the robot was finished on the 29th of November and on the same day successful line following was achieved. After which it took 6 days to create a routine to rescue the first person. In 5 days, the program was extended to finish the whole maze successfully rescuing all persons for the first time. After that the code was refined to increase reliability, cleaned up and tested achieving a 3 out of 5 success rates in 5 successive tests. With this the robot was ready for the exam. The results at the exam were mixed. The robot drove in the same manner as the previous tests; however, it failed considerably in differentiating corners from intersections. This caused the robot to get lost, needing considerable aid to finish the mission and rescue all three people on the third attempt.

Table 2: Test and exam results of the robot

Test Number	First test	One rescued	All rescued	All rescued	Demo
Date	29-11-2024	5-12-2024	10-12-2024	13-12-2024	16-12-2024
Result	Line following	Rescued one person	Drove whole maze rescuing all three people	3/5 attempts rescuing all three people	1/4 attempts rescuing all three people
Best Time	NA	26 s	2 m 51 s	2 m 42 s	2 m 56 s
Interventions	None	None	None	None	Major
Video Proof	No	Yes	Yes	Yes (1 of 5)	Yes (Exam)
Best Points	NA	36 p	206 p	208 p	106*

**Assuming roughly 10 interventions*

Video of first successful attempt 5-12-2024

Video of best achieved time 13-12-2024

Discussion

Gripper design

As alluded to previously, there was a lot of time spent on prototyping an alternate gripper design. The hope was to come up with a design that would quickly tip the cylinder into a scoop, instead of utilizing a more traditional and potentially slower grabbing motion. After 3D printing a proof of concept (the white gripper seen in Figure 3), and hand testing it by simply tilting it to the side, it was decided to iterate on the concept since it appeared to be able to tip and catch the cylinder without it sliding out during the process. These tests assumed a rotational point right by the base of the gripper, which meant that the cylinder was subjected to a tipping motion. However, this rotational point would not be possible in practice due to the body of the robot being in the way. The following iterations of the gripper were therefore tested with the rotational point (the servo attachment point) higher up and further away from the gripper, as seen in Figure 2. Instead of a tipping motion, this resulted in more of a lifting motion, which led to both the other iterations (seen in Figure 3) losing the cylinder due to a lack of centripetal force. A few modifications were made to address the issue, including adding high friction material to the contact surfaces, as well as adding styrofoam to allow for less wiggle room, but there was still no success, and therefore it was decided to abandon this concept. A substantial redesign of the body could have allowed for a lower rotational point, like the one used successfully for the hand tests, but this was realised at a point in the design process at which making changes to the body would require too much time and risk interference with other components.

Instead, the team opted for the simpler solution of redesigning the gripper arm by taking inspiration from the example robot from the previous year, which resulted in the final gripper arm seen in Figure 5. What differentiates these two designs is that the one from the previous year grips the cylinder from behind, whilst this design grips the cylinder from the front. The latter design allows the cylinder to be dropped into the basket, as opposed to being lowered, which is the case for the robot from last year. This leads to saved time, at the cost of a longer arm which needs to be kept in the raised position to not hit the walls while driving.

After finding an appropriate drop angle for the arm and adding the cylinder guide at the front of the body, the gripper arm has yet to fail in picking up a detected cylinder. The whole pick-up process takes approximately four seconds, which in combination with the gripper’s reliability, leads the group to conclude it as a successful design. Perhaps the prototype design could have been faster if implemented correctly, although the extra amount of work and time required would most likely not have been worth it.

Pathing algorithms

In the beginning the proposed algorithms were breadth-first search for searching and A* for backtracking. This however turned out to be troublesome and not the smartest idea since it would take too much time driving to do either a breadth- or depth-first search with all the backtracking involved in those algorithms (Shatha, et al., 2021). Then came the idea of making a custom algorithm for this maze with the specific task of keeping track of which intersection had been visited and from which direction it had been visited. It started with defining the intersections as nodes and giving each of them directional properties which could

either be true, false, dead-end or no-edge. This followed by an adjacency list of nodes, set up as dictionaries, where at each turn, the value was set to true for the direction, meaning the current direction of the robot also had to be updated at all turns. This algorithm turned out to be unsuitable for this problem because of the difficulties with the constantly incorrect detections of intersections. There was also no thought of implementing any backtracking, meaning if all paths in an intersection had been explored, then there was no way out. This led to the realization that a hard coded route was a possibility, leading to the movements array of driver options hardcoded in the array, with and index incremented each time the robot entered an intersection.

Reliability

Reliability has been the biggest hurdle of the project. The robot was able to rescue all people quite early in the testing phase, however in the start it needed 10 attempts, without changing anything, to complete the mission. The largest issue was the robot confusing “normal” 90-degree corners with intersection, causing the robot to lose track of its position in the maze and not being able to solve the problem. To solve this problem, the ultra-sonic sensors were used to a greater extent to check the surrounding walls and use that information to determine whether it is a corner or an intersection. This had mixed results, it allowed the robot to successfully measure and categorize all corners and intersections in the maze. That comes at one assumption, that the sensor data from the ultra-sonic sensors is accurate. Using the serial plotter in the Arduino IDE, a real-time graph is constructed that revealed a lot. It showed that the sensor generally provided accurate data in steady state conditions where the walls are perpendicular to the robot, as seen Figure 8. However, problems were revealed too. Most notably, that the distance measurement is very noise when measuring walls at an angle, as seen Figure 8. To try and solve this, digital filters were developed. The filters are causal hybrid filters, combining a median filter for outlier removals and a moving average filter for reducing noise, as seen in Figure 9. These filters drastically improved the signal quality for the ultra-sonic sensors. Unfortunately, due to the late notice of the problem these filters did not have time to go through enough testing to be used on the exam. The filters have some drawbacks too, most notably that they are computationally heavy and react slower to changes in the sensor data. This means that tuning of these filters is necessary, which we did not have time to do before the exam. The code for these filters can be seen in Appendix B – Digital filter code.

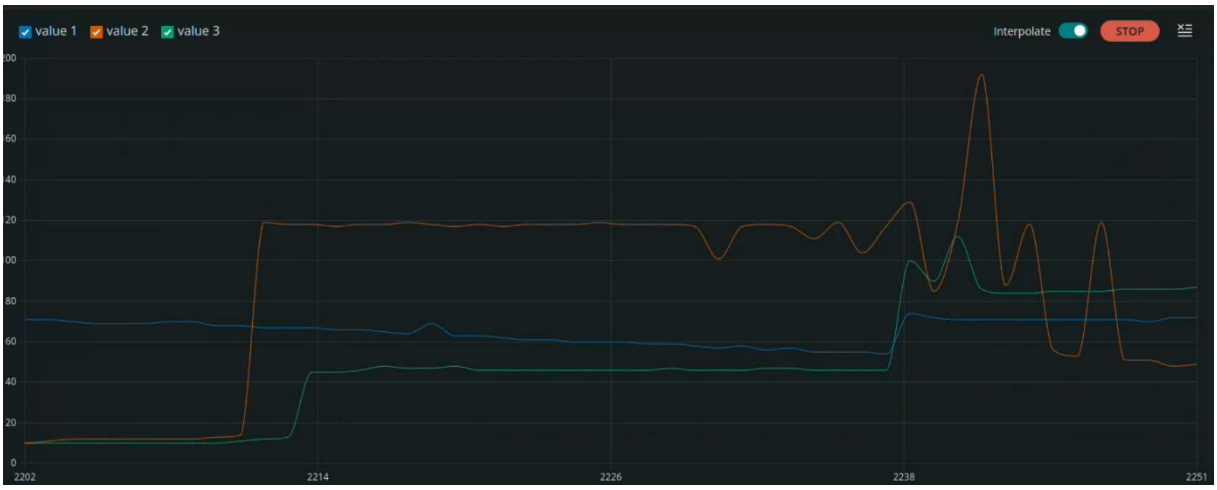


Figure 8: Graph showing the measurements of the front, right and left ultra sonic sensors. Analysis of these plots shows that the ultra-sonic sensors can give steady measurements of the distances, which later becomes noisy.

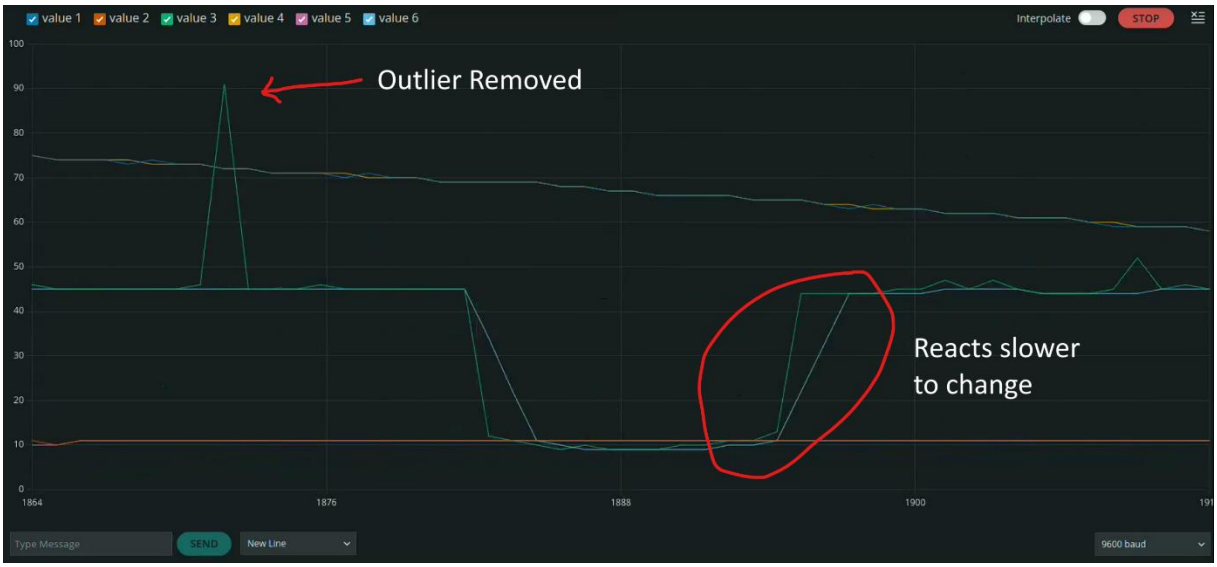


Figure 99: Graph showing the digital filters compared to their original measurements for the front, right and left sensors. The performance of these filters is highlighted by them keeping close to the real values, being able to filter out a bad reading but also that they react slower to changes too.

Conclusion

To conclude this project, there were very few hardware issues with the finished robot, and the gripper, wheels and body worked well. If there had been more time, a general pathing algorithm, such as breadth- or depth first search, would have been interesting to try and implement. Due to the unreliability of the ultrasound sensors, they could have been removed completely and the robot's route entirely hardcoded, meaning it would only have relied on line following. Alternatively, reliability of the ultrasound sensors could have been increased with use of filters or checks that the measurements are good. Lastly, implementing error handling or a re-localization method would have allowed for the robot to be more autonomous and still get back on track if it got lost.

References

Arduino, 2024. *Leonardo.* [Online]
Available at: https://docs.arduino.cc/hardware/leonardo/

Eckel, T., 2023. *NewPing.* [Online]
Available at: https://docs.arduino.cc/libraries/newping/#Compatibility
[Accessed 29 November 2024].

Margolis, M., 2024. *Servo.* [Online]
Available at: https://docs.arduino.cc/libraries/servo/
[Accessed 13 December 2024].

Pololu Corporation, 2014. *QTR-8A and QTR-8RC Reflectance Sensor Array User's Guide.* [Online]
Available at: https://www.pololu.com/docs/pdf/0j12/qtr-8x.pdf
[Accessed 25 November 2024].

Pololu, 2019. *Arduino library for the Pololu QTR Reflectance Sensors.* [Online]
Available at: https://docs.arduino.cc/libraries/qtrsensors/
[Accessed 29 November 2024].

Shatha, A. et al., 2021. *ResearchGate.* [Online]
Available at:
https://www.researchgate.net/publication/355170025_Autonomous_Maze_Solving_Robotics_Algorithms_and_Systems

Wikipedia, 2024. *Twisted pair.* [Online]
Available at: https://en.wikipedia.org/wiki/Twisted_pair

Appendix A – Code used for exam

```
#include <NewPing.h>
#include "CytronMotorDriver.h"
#include <QTRSensors.h>
#include <Servo.h>

// Distance PINS
#define FRONT_TRIGGER_PIN 13
#define RIGHT_TRIGGER_PIN 12
#define LEFT_TRIGGER_PIN 4
#define FRONT_ECHO_PIN 7
#define RIGHT_ECHO_PIN 8
#define LEFT_ECHO_PIN 2
#define MAX_DISTANCE 200

#define MOTOR1A 3
#define MOTOR1B 5
#define MOTOR2A 11
#define MOTOR2B 6

#define buttonPin 13

int distanceFront = 0;
int distanceRight = 0;
int distanceLeft = 0;

// int distanceFrontAvg

int error = 0;
```

```
int rescuedCylinders = 0;
int turnNR = 0; //6 for three way blind
uint16_t position;

const uint8_t SensorCount = 6;
uint16_t sensorValues[SensorCount];

// Line-following thresholds
const uint16_t LINE_THRESHOLD = 650; // Adjust based on calibration
const uint16_t CENTER_POSITION = 2500;

// Orientation
enum Orientation {FORWARD = 0, LEFT = 1, RIGHT = 2, BACKWARD = 3};
enum LineType { STRAIGHT, LEFT_TURN, RIGHT_TURN, INTERSECTION, NONE};

QTRSensors qtr;

// Motor configuration
CytronMD motorLeft(PWM_PWM, MOTOR1A, MOTOR1B);
CytronMD motorRight(PWM_PWM, MOTOR2A, MOTOR2B);

// Ultrasonic sensors
NewPing sonarFront(FRONT_TRIGGER_PIN, FRONT_ECHO_PIN, MAX_DISTANCE);
NewPing sonarRight(RIGHT_TRIGGER_PIN, RIGHT_ECHO_PIN, MAX_DISTANCE);
NewPing sonarLeft(LEFT_TRIGGER_PIN, LEFT_ECHO_PIN, MAX_DISTANCE);

// Arrays for turn indices 1 = left 2 = right 3 = straigth forward
Orientation movements[] = {LEFT, LEFT, LEFT, LEFT, FORWARD, FORWARD, LEFT,
LEFT, LEFT, FORWARD, LEFT, LEFT, LEFT, RIGHT, LEFT, LEFT, LEFT, LEFT,
FORWARD, FORWARD, RIGHT, FORWARD, RIGHT, BACKWARD, LEFT, RIGHT};
```

```
// Setup
void setup() {
  pinMode(buttonPin, INPUT_PULLUP);
  int buttonState = digitalRead(buttonPin);
  while(buttonState != LOW){
    // Serial.println("WAIT FOR CALIBRATION");
    buttonState = digitalRead(buttonPin);
  }

  // configure the sensors
  qtr.setTypeAnalog();
  qtr.setSensorPins((const uint8_t[]){A0, A1, A2, A3, A4, A5}, SensorCount);

  pinMode(LED_BUILTIN, OUTPUT);
  digitalWrite(LED_BUILTIN, HIGH); // turn on Arduino's LED to indicate we are in
calibration mode

  // configure the gripper
  gripperSetup();
  spin();
  for (uint16_t i = 0; i < 400; i++)
  {
    qtr.calibrate();
  }
  stopMotors();
  digitalWrite(LED_BUILTIN, LOW); // turn off Arduino's LED to indicate we are through
with calibration

  Serial.begin(9600);
  // Serial.println("Robot Initialized");
  pinMode(buttonPin, INPUT_PULLUP);
```

```
  int loopButtonState = digitalRead(buttonPin);
  while(loopButtonState != LOW){
    // Serial.println(loopButtonState);
    loopButtonState = digitalRead(buttonPin);
  }
}

// Main loop
void loop() {
  // Step 1: Cylinder detection and rescue
  if (detectCylinder()) { // && lineType != NONE
    rescueCylinder();
    delay(100);
  }

  // Step 2: Read QTR sensors
  position = qtr.readLineBlack(sensorValues); // 0 for sensor 0, 1000 for sensor 1, 2000 for sensor 2 etc.
  LineType lineType = detectLineType(sensorValues, LINE_THRESHOLD);

  distanceFront = sonarFront.ping_cm();
  distanceRight = sonarRight.ping_cm();
  distanceLeft = sonarLeft.ping_cm();

 // At the end of loop(), after reading sensor values:
Serial.print(distanceFront);
Serial.print("\t");
Serial.print(distanceRight);
Serial.print("\t");
Serial.println(distanceLeft);
```

```
if (turnNR >= 26 && lineType == NONE) {
  stopMotors();
  while (true) {
  }
}

// Step 3: Handle Line Type
switch (lineType) {
  case STRAIGHT:
    followLine(position); // Continue line-following
    break;

  case LEFT_TURN:
    // Serial.println("LEFT_TURN");
    if (distanceRight > 20) {
      break;
    }

    if(distanceFront > 40){
      int averageDistanceFront = 0;
      for (int i = 0; i < 3; i++) {
        distanceFront = sonarFront.ping_cm();
        averageDistanceFront += distanceFront;
      }
      if (averageDistanceFront/3 > 40) {
        // Serial.println("INTERSECTION");
        handleIntersection();
        break;
      }
    }
```

```
turnLeft(); // Perform left turn
break;

case RIGHT_TURN:
  // Serial.println("RIGHT_TURN");
  if (distanceLeft > 20) {
    break;
  }

  if(distanceFront > 40){
    int averageDistanceFront = 0;
    for (int i = 0; i < 3; i++) {
      distanceFront = sonarFront.ping_cm();
      averageDistanceFront += distanceFront;
    }
    if (averageDistanceFront/3 > 40) {
      // Serial.println("INTERSECTION");
      handleIntersection();
      break;
    }
  }
  turnRight(); // Perform right turn
  break;

case INTERSECTION:
  // Serial.println("INTERSECTION");
  handleIntersection();
  break;

case NONE:
```

```
      handleNoLine();
      break;
    }
}


LineType detectLineType(uint16_t *sensorValues, uint16_t threshold) {
  int activeSensors = 0;       // Count of sensors detecting the line
  int firstActive = -1;        // Index of the first sensor detecting the line
  int lastActive = -1;         // Index of the last sensor detecting the line

  // Analyze sensor values to find active sensors
  for (uint8_t i = 0; i < 6; i++) { // Assuming 6 sensors
    if (sensorValues[i] > threshold) {
      activeSensors++;
      if (firstActive == -1) {
        firstActive = i; // Mark the first active sensor
      }
      lastActive = i; // Continuously update the last active sensor
    }
  }

  // Determine line type
  if (activeSensors == 0) {
    return NONE; // No line detected
  } else if (activeSensors <= 3) { // Straight line case (at most 3 sensors)
    return STRAIGHT;
  } else if (activeSensors >= 4) { // At least 4 sensors detecting the line
    if (firstActive <= 2 && lastActive <= 3) {
      return RIGHT_TURN; // Line on the left (left turn)
```

```
    } else if (firstActive >= 2 && lastActive >= 3) {
      return LEFT_TURN; // Line on the right (right turn)
    } else {
      return INTERSECTION; // Line spans across center and edges
    }
  }

  return NONE; // Default case (shouldn't happen)
}

// Line-following logic
void followLine(uint16_t position) {
  int prev_error = error;
  error = position - CENTER_POSITION; // Calculate error relative to the center
  int derivative_error = error - prev_error;
  int turnSpeed = error / 10 + derivative_error/4; // Adjust motor speed based on error

  // Adjust motors to stay on the line
  motorLeft.setSpeed(150 - turnSpeed);
  motorRight.setSpeed(150 + turnSpeed);
}

void handleIntersection() {
  // Serial.println(turnNR);
  switch(movements[turnNR]){
    case FORWARD:
      moveForwardIntersection();
      break;
    case LEFT:
      turnLeft();
```

```
      break;
    case RIGHT:
      turnRight();
      break;
    case BACKWARD:
      turnAround();
      break;
  }
  turnNR++;
}

// Rescue logic
bool detectCylinder() {
  int objectDistance = sonarFront.ping_cm();
  return (objectDistance > 0 && objectDistance < 5);
}

void rescueCylinder() {
  stopMotors();
  delay(100);
  rescuedCylinders++;
  // Serial.print("Cylinder rescued! Total rescued: ");
  // Serial.println(rescuedCylinders);
  gripAndRelease();
}

void handleNoLine(){//int distanceFront, int distanceRight, int distanceLeft){
  stopMotors();
  int distanceFront = sonarFront.ping_cm();
  int distanceRight = sonarRight.ping_cm();
```

```
  int distanceLeft = sonarLeft.ping_cm();


  if(distanceFront < 30 && distanceRight < 30 && distanceLeft < 30){
    turnAround();
    return;
  }


  if(turnNR < 5){
    moveForwardBlind();
    turnRightBlind();
    moveForwardBlind();
    position = qtr.readLineBlack(sensorValues); // 0 for sensor 0, 1000 for sensor 1, 2000 for
sensor 2 etc.
    followLine(position);
    return;
  }
  switch(movements[turnNR]){
    case LEFT:
      if (turnNR < 7) {
        moveForwardBlindLong();
        turnLeftBlind1();
      } else if (turnNR > 8) {
        moveForwardBlind();
        turnLeftBlind2();
      }
      moveForwardBlindShort();
      position = qtr.readLineBlack(sensorValues); // 0 for sensor 0, 1000 for sensor 1, 2000 for
sensor 2 etc
      for (int i = 0; i < 3; i++) {
        followLine(position);
      }
```

```
      break;
  }
  if(turnNR > 5){
    turnNR++;
  }
}

void delayOrLine(uint16_t time){
  long timer_0 = millis();
  LineType line = NONE;

  while (millis() < (timer_0 + time) && line != STRAIGHT ){
  //position = qtr.readLineBlafck(sensorValues); // 0 for sensor 0, 1000 for sensor 1, 2000 for sensor 2 etc.
  line = detectLineType(sensorValues, LINE_THRESHOLD);
  delay(50);
  }
}

void moveForwardBlindShort() {
  // Serial.println("moveForwardBlindShort");
  motorLeft.setSpeed(100);
  motorRight.setSpeed(100);
  delayOrLine(1000);
}

// Movement functions
void moveForward() {
  // Serial.println("moveForward");
  motorLeft.setSpeed(150);
  motorRight.setSpeed(150);
```

```
}

void moveForwardIntersection() {
  // Serial.println("moveForwardIntersection");
  motorLeft.setSpeed(150);
  motorRight.setSpeed(150);
  delay(100);
}

void spin(){
  // Serial.println("spin");
  motorLeft.setSpeed(-100);
  motorRight.setSpeed(100);
}
void turnLeft() {
  // Serial.println("turnLeft");
  motorLeft.setSpeed(-100);
  motorRight.setSpeed(100);
  delay(500);
}
void turnRight() {
  // Serial.println("turnRight");
  motorLeft.setSpeed(100);
  motorRight.setSpeed(-100);
  delay(500);
}
void turnAround() {
  // Serial.println("Turn Around");
  motorLeft.setSpeed(100);
  motorRight.setSpeed(-100);
```

```
  delayOrLine(1700);
}

void stopMotors() {
  // Serial.println("stopMotor");
  motorLeft.setSpeed(0);
  motorRight.setSpeed(0);
}

void moveForwardBlind() {
  // Serial.println("moveForwardBlind");
  motorLeft.setSpeed(100);
  motorRight.setSpeed(100);
  delayOrLine(1400);
}

void moveForwardBlindLong() {
  // Serial.println("moveForwardBlindLong");
  motorLeft.setSpeed(100);
  motorRight.setSpeed(100);
  delay(1600);
}

void turnLeftBlind1() {
  // Serial.println("turnLeftBlind");
  motorLeft.setSpeed(-100);
  motorRight.setSpeed(100);
  delay(800);
}
```

```
void turnLeftBlind2() {
  // Serial.println("turnLeftBlind");
  motorLeft.setSpeed(-100);
  motorRight.setSpeed(100);
  delay(750);
}

void turnRightBlind() {
  // Serial.println("turnRightBlind");
  motorLeft.setSpeed(100);
  motorRight.setSpeed(-100);
  delay(800);
}

//Servo objects for arm and gripper
Servo arm;
Servo gripper;

int pos = 0;  //angle tracker
uint8_t standby_angle = 90;
uint8_t gripp_angle = 170;
uint8_t release_angle = 75;

void gripperSetup(){
  arm.attach(9);
  gripper.attach(10);
  arm.write(standby_angle);  //start angle arm, 170 degrees is straight forward and 0 degrees is straight backwards
  gripper.write(90);   //start angle gripper, 180 degrees is fully open and 0 is a little more than closed
}
```

```
//call the following function when a cylinder has been detected
void gripAndRelease() {
  for(pos = arm.read(); pos <= gripp_angle ; pos += 1) {
     arm.write(pos);
     delay(10);
    }
    delay(100);
    gripper.write(5);
    delay(500);
    for(pos = arm.read(); pos >= release_angle ; pos -= 1) {
     arm.write(pos);
     delay(10);
    }
    delay(500);
    gripper.write(90);
    delay(500);
    for(pos = arm.read(); pos <= standby_angle ; pos += 1) {
     arm.write(pos);
     delay(10);
    }
}
```

Appendix B – Digital filter code

```
void measureFront(){

  i_front++;
  if (i_front == NUM_MEASUREMENTS){
    i_front = 0;
  }
  distanceFrontAvg[i_front] = sonarFront.ping_cm();
}

void measureLeft(){
  i_left++;
  if (i_left == NUM_MEASUREMENTS){
    i_left = 0;
  }
  distanceLeftAvg[i_left] = sonarLeft.ping_cm();
}

void measureRight(){


  i_right++;
  if (i_right == NUM_MEASUREMENTS){
    i_right = 0;
  }
  distanceRightAvg[i_right] = sonarRight.ping_cm();
}


int calcAvgFront(){
```

```
int sortedDistances[NUM_MEASUREMENTS];

// Copy the array to avoid modifying the original
for (int k = 0; k < NUM_MEASUREMENTS; k++) {
  sortedDistances[k] = distanceFrontAvg[k];
}

// Sort the array
for (int k = 0; k < NUM_MEASUREMENTS - 1; k++) {
  for (int l = k + 1; l < NUM_MEASUREMENTS; l++) {
    if (sortedDistances[k] > sortedDistances[l]) {
      int temp = sortedDistances[k];
      sortedDistances[k] = sortedDistances[l];
      sortedDistances[l] = temp;
    }
  }
}

// Select the median 3 values (middle three values in the sorted array)
int medianValues[3];
medianValues[0] = sortedDistances[1];
medianValues[1] = sortedDistances[2];
medianValues[2] = sortedDistances[3];

// Calculate the average of the median 3 values
int sum = medianValues[0] + medianValues[1] + medianValues[2];
int average = sum / 3;

return average;
```

```
}

int calcAvgLeft(){

 int sortedDistances[NUM_MEASUREMENTS];

 // Copy the array to avoid modifying the original
 for (int k = 0; k < NUM_MEASUREMENTS; k++) {
  sortedDistances[k] = distanceLeftAvg[k];
 }

 // Sort the array
 for (int k = 0; k < NUM_MEASUREMENTS - 1; k++) {
  for (int l = k + 1; l < NUM_MEASUREMENTS; l++) {
   if (sortedDistances[k] > sortedDistances[l]) {
    int temp = sortedDistances[k];
    sortedDistances[k] = sortedDistances[l];
    sortedDistances[l] = temp;
   }
  }
 }

 // Select the median 3 values (middle three values in the sorted array)
 int medianValues[3];
 medianValues[0] = sortedDistances[1];
 medianValues[1] = sortedDistances[2];
 medianValues[2] = sortedDistances[3];

 // Calculate the average of the median 3 values
 int sum = medianValues[0] + medianValues[1] + medianValues[2];
```

```
  int average = sum / 3;

  return average;
}

int calcAvgRight(){

  int sortedDistances[NUM_MEASUREMENTS];

  // Copy the array to avoid modifying the original
  for (int k = 0; k < NUM_MEASUREMENTS; k++) {
    sortedDistances[k] = distanceRightAvg[k];
  }

  // Sort the array
  for (int k = 0; k < NUM_MEASUREMENTS - 1; k++) {
    for (int l = k + 1; l < NUM_MEASUREMENTS; l++) {
      if (sortedDistances[k] > sortedDistances[l]) {
        int temp = sortedDistances[k];
        sortedDistances[k] = sortedDistances[l];
        sortedDistances[l] = temp;
      }
    }
  }

  // Select the median 3 values (middle three values in the sorted array)
  int medianValues[3];
  medianValues[0] = sortedDistances[1];
  medianValues[1] = sortedDistances[2];
  medianValues[2] = sortedDistances[3];
```

```
  // Calculate the average of the median 3 values
  int sum = medianValues[0] + medianValues[1] + medianValues[2];
  int average = sum / 3;

  return average;
}

//Servo objects for arm and gripper
Servo arm;
Servo gripper;

int pos = 0;  //angle tracker
uint8_t standby_angle = 90;
uint8_t gripp_angle = 170;
uint8_t release_angle = 75;

void gripperSetup(){
  arm.attach(9);
  gripper.attach(10);
  arm.write(standby_angle);  //start angle arm, 170 degrees is straight forward and 0 degrees is straight backwards
  gripper.write(90);   //start angle gripper, 180 degrees is fully open and 0 is a little more than closed
}

//call the following function when a cylinder has been detected
void gripAndRelease() {
  for(pos = arm.read(); pos <= gripp_angle ; pos += 1) {
      arm.write(pos);
      delay(10);
```

```
    }
    delay(100);
    gripper.write(5);
    delay(500);
    for(pos = arm.read(); pos >= release_angle ; pos -= 1) {
      arm.write(pos);
      delay(10);
    }
    delay(500);
    gripper.write(90);
    delay(500);
    for(pos = arm.read(); pos <= standby_angle ; pos += 1) {
      arm.write(pos);
      delay(10);
    }
}
```