Engineering Design and Innovation

Robot Rescue Project Report

Team - Phat Boi

Members

- Dominic He (z5024963)
- Akshat Agarwal (z5117523)
- Tristan Bagnulo (z5113473)
- Raycole Dai (z5116725)
- Thomas George (z5112681)
- Nikodemus Limanuel (z5113992)

Lecturer: Claude Samut

Submitted Sunday, 5 June 2016

Abstract

This report will outline our team’s design process and thought structure into the construction of the robot. The main aim in robot rescue was to navigate through a particular maze and retrieve a ‘survivor’ in the shape of a can. In order to fulfil the goal we needed to create reliable clamps that could handle the weight of the can and not lose it on they way back out. Also, more importantly, the software designed to run the robot needs to be tested continuously to avoid hiccups on the day of testing. Our robot put out a solid performance, managing to travel through the maze. The return journey however was a little disappointing as we failed to pick up the can at times and crashed to the walls a bit. Some improvements that could have been made for the run was to focus on fixing the pathing for our robot. Extensive testing is needed to make sure that our robot runs smoothly and reliably, although there were issues as the robot would crash and freeze multiple times.

Table of Contents

1. Introduction

Our challenge was Robot Rescue where the goal is to traverse through the maze, pick up the can located within the maze, and travel back the way it came. To implement this rescue, the design brief was to build the robot using the Lego Mindstorm kit and attach various sensors and features that will allow it to navigate the maze, locate the can and pick it up. The smoother and more efficient the robot is at completing the maze, the better the score. The rules of this challenge are as follows:

- The components of the robot must be built using the EV3 kit.
- It must avoid heavy contact with the walls, meaning it should not push the walls around.
- Once the can has been picked up, the robot must travel the same path it took to get in.
- There is a time limit of five minutes for each evaluation test.

There are two different test cases the robot must follow. The first is the simple maze which is just a normal zig-zagging maze the robot follows and returns from. The second maze is more difficult with both an intersection and loop. The intersection and loop are both out of five marks, while the simple maze is worth 10 marks.

Constructing the build for the robot proved to be of little difficulty as that was not the main focus. The clamps however held certain challenges as it was difficult to attach the motor and gears to the clamps and the robot. Furthermore, the design of the clamps were continuously updating and changing as a result of many tests. Each design held some improvements from previous iterations such as reliability, ease of use, efficiency, stability, and overall effectiveness.

Our final claw settled on a semi-interlocking design to ensure that the can doesn't slip out when it is picked up. Another difficulty handed to us was the fact that our robot struggled to continue on a straight path, instigating numerous discussions as to how to fix this problem, we managed to use our ultrasonic sensor to gauge the distance from the walls and from that programmed the robot to stay certain distance from it, thus creating a somewhat linear path.

This report has been mainly split up in two parts. Conceptual design focuses on the team's thought process, discussing ideas for the hardware and software section. While brainstorming concepts can be useful to plan out, it can be useless if the implementation and testing fail to reach our desired goals. This was seen in the clamps as it took many prototypes to finally settle on the final design. This was also seen in the software side, where the robot seemed to not follow the instructions we had given it, mainly the pathing problem. Also there were numerous times when our robot just froze and crashed, severely delaying the efficiency of our testing, as rebooting the brick took some time to complete.

2. Conceptual Design

2.1 Hardware

Our given task was quite clear; to navigate a maze, retrieve a red object and return to the starting point with the object. Despite the task's clarity, conceptualising a design for the robot was a very difficult process, demanding significant contributions from each member to imagine a mechanical system able to complete sets of complicated and intricate process in order to achieve given aims. Admittedly, most of the final designs for our team's rescue robots were reached through a somewhat disorganised process of repeated testing and altering of spontaneous ideas rather than an ordered and superior conceptualisation process involving compilation and selection. Consequently some aspects of the final design bear only a slight similarity to the original concepts below. Despite the improper nature of our conceptualisation process I do believe it fit our situation well, seeing as the given task was far more complex than originally perceived. This evoked continuous changes in our understanding of it, thus creating constant conceptual and physical alterations, which would have broken down the structure of the more proper approach.

2.1.1 Ultrasonic Sensor (Navigation)

The biggest problems we faced were the basic movement of the robot within the maze and its overall ability to solve the maze. In short, our robot had to recognise the presence of paths and act accordingly. We decided to use an ultrasonic sensor, mounted on the side of the robot to understand whether or not there was a path adjacent to its position. The sensor was to be mounted on the left side of the robot, constantly giving live information of the proximity of wall to the sensor. The reason behind this is that the robot would be able to understand if there either a wall or a path directly to its left and turn left if it could (figure 2.1) (there's more detail on this in the software section of the report). Contributing to this concept was advice from our lecturer to make our robot "left-turning" so that it would turn left whenever possible which provided the the robot a sound ability to navigate through basic mazes.

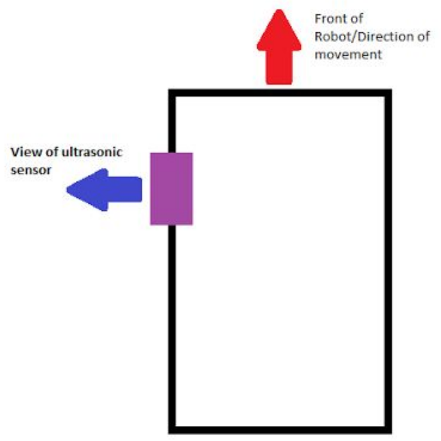

Front of Robot/Direction of movement
View of ultrasonic sensor

Figure 2.1 (left-mounted ultrasonic sensor concept)

2.1.2 Touch sensor (Navigation)

The idea of a front-mounted touch sensor was simultaneously conceptualised along-side the left-mounted ultrasonic sensor as a part of the basic navigational abilities of the robot. Our group's idea for a system of basic movement throughout the maze was clear; dependent on both the ultrasonic and touch sensors, which were enabled a simple, yet effective means by which the robot could act effectively given any immediate situation within the maze. Our idea was to mount a touch sensor to the front of the robot such that if it were activated (when the robot drove into a wall head-on) the robot would turn towards the right. This design would work hand in hand with feedback from the ultrasonic; the robot would turn left whenever it could but in the case that there were no left options the robot would turn right (figure 2.2). Before deciding on this integrated system we drew mazes and mapped the potential movement of the robot in numerous situations to ensure that at least the basic idea would work (figure 2.3).

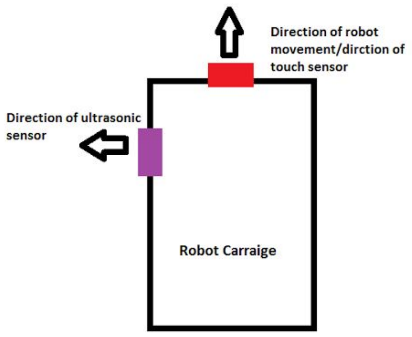

Direction of robot movement/dirction of touch sensor
Direction of ultrasonic sensor
Robot Carraige

Figure 2.2 (final concept of the touch/ultrasonic-mechanism of movement)

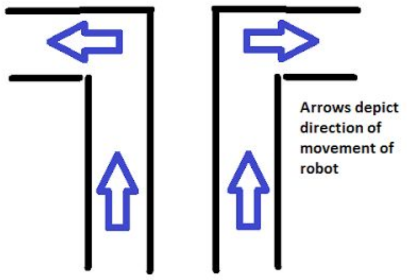


Figure 2.3 (the hypothetical movement of robot in two instances)

2.1.3 Front-mounted colour sensor (Rescuing)

The next problem for us to tackle was creating a system by which the robot could detect the presence of a red-coloured object. To us the most obvious idea was to mount the colour sensor, included in the lego kit on the front of the robot. The reasoning behind placing it on the front of the robot is that we imagined it to be most likely that the red object would be in front of the moving robot, and thus that the robot would meet it head on. There was no argument about the detection system between group members, given that this particular objective was so simple to achieve and required only common sense.

2.1.4 Front-mounted ball-bearing/wheel placement (Basic Movement)

Another important consideration was to decide on how the robot would touch the ground. There was some disagreement whether the third point of contact that the robot had with the ground should be either a flat surface or the ball-bearing provided within the kit. One argument was that the ball-bearing would be more suitable for allowing movement by reducing friction and another was that it would be an unstable/flimsy structure on which the weight of the robot had to rest. An agreement was reached when it was apparent that the robot would be travelling on a carpeted surface rather than a smooth plastic surface (as in the sumo competition in which we didn't use the ball-bearing) and thus that eliminating friction with the ground was essential.

The ball-bearing wheel was placed at the front of the robot such that it was equidistant from the two rear wheels, creating an isosceles triangle shape from a skyview (figure 2.4). The idea behind triangular points of contact was that the robot wouldn't be unbalanced and tilt to any particular side (which could have distorted its navigation within the maze). Furthermore, we decided to place most of the weight on the robot's carriage towards the back. This meant that less weight hung over the unstable ball-bearing and more above the stronger wheels so that whilst turning the shifting of weight would not capsize the robot.

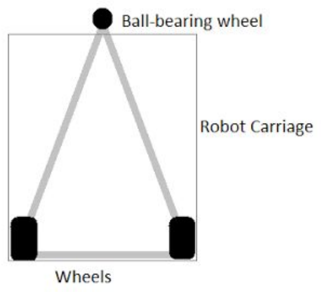


Figure 2.4 (depicts the 3 points of ground contact in a triangular formation with the carriage parametres on top. Ball-bearing is equally distant from both wheels)

2.1.5 Clamps (Transporting the Object)

The most difficult concept for our group to complete was the mechanism by which the robot could grab and move with the rescue object. It was difficult for the group to come up with any more than a few basic features for a mechanism and what materials we would solve this problem with. We decided to use the third motor and a simple system of gears to pick up and hold on to the robot. The gears and motor were to be constructed such that two "arms" of a clamp could be moved at equal speeds to close on a given object. To understand our design well enough before moving onto implementation we tested numerous formations of gears and the rotating end of the motor. Quite quickly we made a design that the team agreed would work (figure 2.5).

A big issue we encountered was where on the robot to place the clamps. Initially we thought to put them on the front but quickly we realised that the clamps would interfere with the touch and colour sensors on the robot. With some hesitation we decided to put the clamps on the back of the robot. This created another task for us; to make to robot turn 180 degrees when it detected an object but there was no other way around it (there is more information about this in the software section of the report).

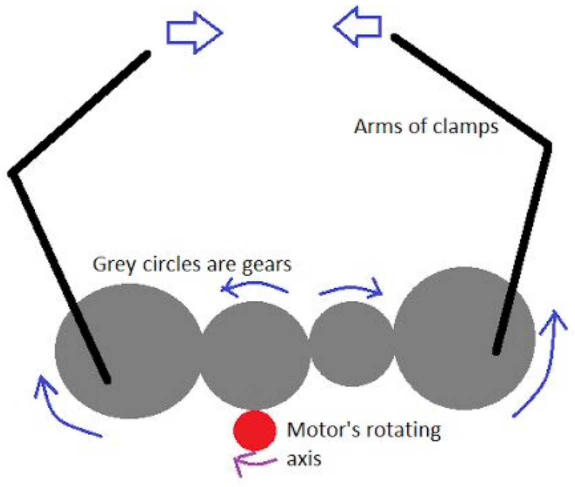


Figure 2.5 (depicts the final concept for the gearing mechanism)

2.2 Software

The software solution for the rescue robot makes use of its hardware features to achieve the goal of locating a red drink can inside a maze and getting it back to the robot's starting position. The team devised a number of software functions the robot should perform:

- Using the left wall follower algorithm to search the maze for the red can;
- Ensuring that the robot moves in a straight line such that it avoids bumping into side walls;
- Grappling the red can; and
- Returning to the starting position of the robot.

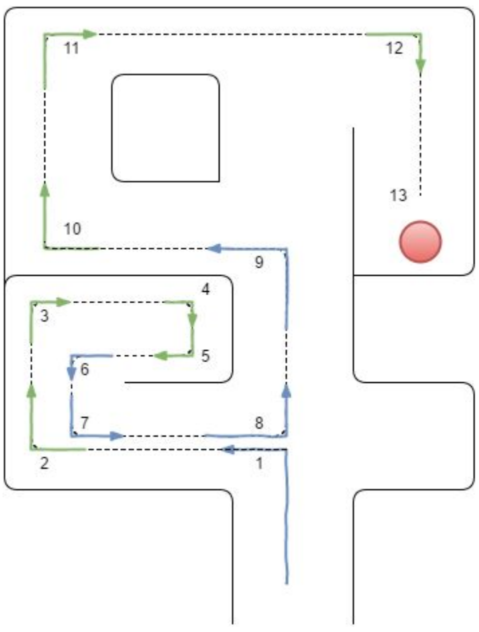
2.2.1 Solving the maze

The wall follower algorithm was selected as the most practical method of searching through the maze to locate the red can, since it only requires sensors for searching paths on one side and the front. Also known as the left hand rule, it involves following the wall on the left, until the red can is found, by running straight forward in a maze path, turning left if an intersection is found to the left of the robot, and turning right if the path ahead is blocked.

The ultrasonic sensor, which is mounted facing the left of the robot, is used for finding paths on the left, by identifying when the distance between the robot and the left wall is above a certain threshold, approximately the length of an A3 sized paper. The touch sensor is used to detect whether or not there is a wall ahead of the robot, so that the robot turns right if it bumps against the wall.

The wall follower algorithm, with flowchart shown in **Appendix A1**, has the robot running straight forward, while a loop checks the conditions for rotating left or right, or stopping when the red can has been found:

- When the ultrasonic sensor detects a distance beyond a threshold indicating a new path on the left, the robot stops, rotates 90 degrees anticlockwise, and continues moving forward.
- When the touch sensor is bumped, the robot reverses a small distance away from the wall, then rotates 90 degrees clockwise, and continues moving forward.
- If the colour value detected by the colour sensor is red, then the loop stops, and the robot proceeds to grab the can.



These steps in the loop ensure the robot is running along the left wall until it has found the can, as shown in an example maze in Figure 2.6. The blue arrows indicate where the robot finds a new path on the left with the ultrasonic sensor and turns left, while the green arrows show where the robot bumps into the wall and turns right.

Figure 2.6 Example maze and robot’s wall-following path through it

2.2.2 Moving straight

While running forward in a maze, the robot applies the wheel motors with equal power, which does not necessarily cause it to move in a sustained straight line. Thus the function of ensuring the robot is moving straight, uses the ultrasonic sensor and gyroscope to ensure that any deviation from the expected straight line is corrected.

2.2.3 Gyroscope correction

Due to the inaccuracy of the gyroscopic sensor which we experienced first hand in the Robo-rescue, we knew that we would need a way to make our gyro value more accurate. There are a multitude of ways to achieve this, and we decided to use as many as possible, to help improve the robot's understanding of its position and relative motion inside the maze. We make sure that at the beginning of the robots straight path through the maze, and every time after it has turned either left or right, we reset the gyro. We do this so the gyro value is now set at 0 degrees parallel to the walls of the straight path.

While running forward, if we detect that our gyro value is skewing by more than a certain threshold, we can alter the power to the wheels individually to correct its deviance from a straight line.

Since our ultrasonic sensor is to be placed on the left side of our robot, we can use the sensor to both determine how centred we are in the path, and also by measuring the change in distance over time, we can calculate the rate at which we are diverging or converging from the left wall. We can use this information along with the changes in the gyroscope values to calculate with reasonable certainty our position inside the path, and how much rectification is required by the motors to fix our path.

2.2.4 Returning from maze

To return from the maze, we had to somehow store where we had been in the maze. The main idea that we came up with was to record how long the motors ran for and the distance that the motors had run whenever it turned, and store it in a stack. Using a stack would provide the added advantage of its first-in, last-out capability, which was perfect for returning, as we needed to travel the path that was last recorded each time. This functionality gave it advantage over the use of a list or set.

Whenever we recorded a path, we needed to store 3 items, the distance, time and direction it turned. Therefore, in order to store the different paths taken, we needed to store 3 different values. Tuples are the easiest way as they count as 1 element in the stack but can also store the 3 values required. Another option was to use a 2-D array, but that would be too hard to implement.

3. Implementation and Testing

3.1 Hardware

The implementation and testing of the hardware designs was quite simple for some but very drawn out for others. As previously stated many of the original designs created during the conceptualisation phase have undergone significant changes and could only be translated into reality with a significant quantity of improvisation and multiple returns to a brief conceptualising stage amidst implementation. Some designs including the ultrasonic sensor, colour sensor and the wheel allocation (except for a bit of piece-mashing) were implemented in a very straight-forward fashion with little to no deviation from the original concepts and no revisitation of the planning stage. However, designs such as the touch sensor and the clamps especially required us to come up with new concepts to bring their basic idea into reality. It is due to the repeated cycles of conceptualising and testing that made the implementation of the basic designs of the robot particularly lengthy and quite tedious.

3.1.1 Ultrasonic Sensor

The idea to put the ultrasonic sensor on the left-hand side of the robot was swiftly implemented without any difficulty. To keep it level we attached it to connection ports on the side of the brick (figure 3.1 and figure 3.2)

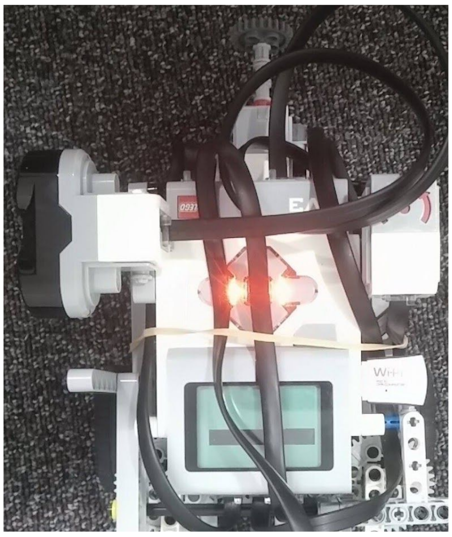


Figure 3.1

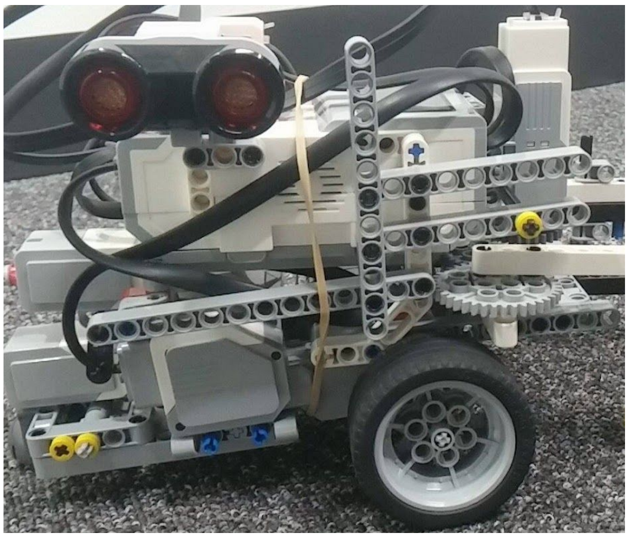


Figure 3.2

3.1.2 Front-mounted ball-bearing/Wheel allocation

It was quite difficult to attach the ball-bearing wheel to the front of the robot and it took attempts with numerous arrangements of parts included in the kit to ensure that it didn't wobble. The biggest problem we noticed when doing this is that most pieces werent able to attach the 3rd wheel to the front without the entire structure sagging at the joints between pieces that connected it to the underside of the robot's carriage. It was therefore necessary to attach it with sturdy pieces and as little pieces as possible (to eliminate the number of joints at which bending could occur). We managed to attach the wheel with a rectangular piece from the lego kit that had holes running through it adjacent to one another however the final result wasn't perfect as we couldn't place it directly in middle of the robot (figure 3.3). This made the ball-earing unequally distant from each of the wheels which made the robot skew slightly to the left. This was an enormous problem and it created new tasks for us but we decided to leave the robot with this build and try to correct its misdirection (which was also caused by other factors) in our code.

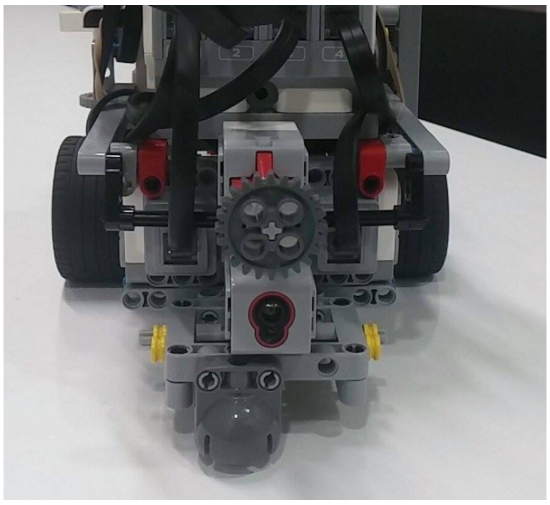


Figure 3.3 (depicts the positions of the three contact points and rectangular piece joining the ball-bearing to the carriage. Also shows position of colour and touch sensor)

3.1.3 Front-mounted colour sensor

This was fairly easy to implement, all it required was sort of rack through which bars could run through. Initially we had it side by side with the touch sensor but numerous changes were made to the front section of the robot whilst adding the wheel so it was decidedly easier to have the touch sensor and colour sensors lined up one on top of the other (figure 3.3 above). Furthermore, having the sensors lined up vertically gave us the added benefit of having the colour sensor centred, thus increasing the chance it had with facing and therefore, detecting the red object given that the robot was coded to remain centred within the mazes, paths.

3.1.4 Front-mounted touch sensor

As stated above the original plan was to have the touch sensor horizontally adjacent to the colour sensor but changes to the front wheel made it possible to have the touch sensor above the colour sensor (figure 3.3 above). This proved to be beneficial as the touch sensor could be placed directly in the centre of the front of the robot and in line with the robot's centre of mass which would prevent the its direction from being tilted to a specific side by the force acting back on the robot when the touch sensor met with the wall.

Another problem with the touch sensor was that it needed to be the front-most part of the robot so that it could be activated before any other part met the wall. First we tried moving the entire sensor forward but given its awkward design doing this seemed too difficult and required numerous pieces that could have interfered with the colour sensor. It seemed easier to insert a rod into the port in the front of the touch sensor. There were some issues with this idea found during testing; the rod would meet the wall and bend to the side during testing, and thus the sensor was not activated. To fix this we placed a small cog on the front of the rod with small pieces behind it to hold it to keep it from sliding backwards. During all subsequent testing, when the rod happened to meet the wall at an unfavourable angle its direction would be corrected by the sides of the circle and the sensor would be activated (figure 3.4).

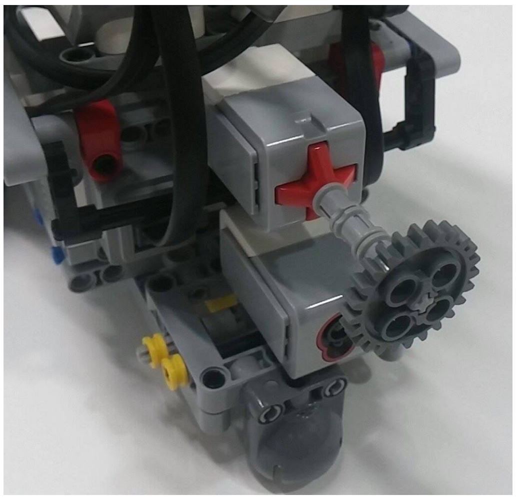


Figure 3.4 (touch sensor centred with rod, wheel and braces attached to the front)

3.1.5 Clamps

The opening and closing mechanism of the clamps involving a series of gears and the motor was quite easy to implement, we simply replicated the design that we'd come up with during the conceptualisation process and it worked perfectly during testing. Attaching the actual mechanism to the back of the robot was a little more challenging as we had to work in a very tight space and the cogs had to be isolated from surrounding pieces so that their movement wasn't hindered (figure 3.5). Doing this required the collaboration of both myself and Niko and a significant amount of time to test numerous ideas.

The most difficult and time-consuming aspect of the clamps to implement was the arms. Our initial design included a set of arms that, when closed, met each other perfectly to form a solid-wall casing around the can. This design proved to be problematics as the arms would prevent each other from moving around the can when the met, thus the can moved around whilst within the grasp of the arms when the robot carried it and it would fall out shortly after being picked up. Eventually we came up the idea of having arms that interlocked and formed a tighter grip on the can. There was a failed set of interlocking arms that ended up being too heavy and unstable due to the large number of joined pieces but a functional/sturdy design was swiftly discovered (figure 3.6).

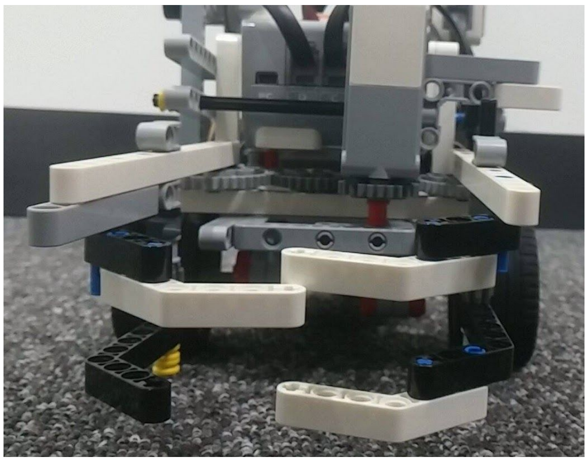


Figure 3.5 (gives a view of the gearing mechanism that opens/closes the arms of the clamp)

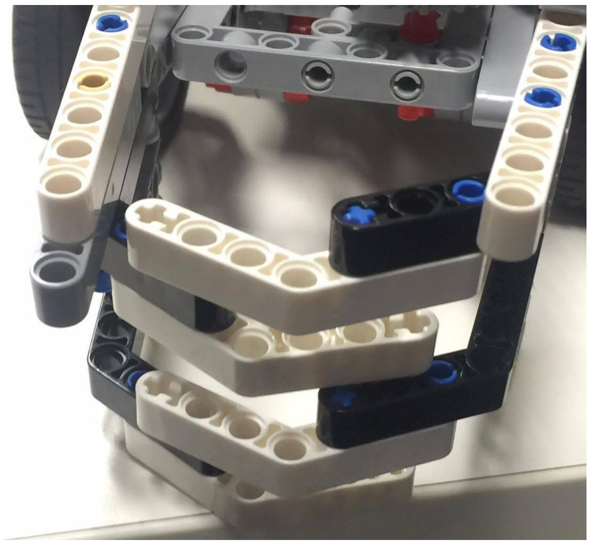


Figure 3.6 (the interlocking clamps whilst closed forming a tight space in which can cannot move during motion)

3.2 Software

The team’s implementation of the software for the robot is split into 2 main parts, Search and Rescue. The Python program uses two separate threads to separate the logic between moving in a straight line and following the left wall to find the software can.

3.2.1 Motors and sensors

To control the motors and access the sensors, we created small functions for actions that would be reused continually throughout the code, shown in Table 3.1. Apart from making our code less cluttered, it helped other members in the group who were less versed in the Python to understand what was going on in our program. For example, run_motors() is a lot more intuitive to understand than left_motor.run_direct(), right_motor.run_direct(). Also, since our robot’s motors were mounted backwards, applying the duty cycle of the motor to a positive value moves the robot backwards. It was more intuitive to define the run_motors function to reverse the motor power parameters. We decided to use “stop_command = ‘brake’” as opposed to setting motor power to 0 as it would stop a lot quicker. This would help the accuracy of our turning and mapping of our path.

Table 3.1 Python functions for motor control and sensors

MOTORS	**SENSORS**

def run_motors(left, right):
    left_motor.run_direct(duty_cycle_sp=left * -1)
    right_motor.run_direct(duty_cycle_sp=right * -1)
	def reset_gyro():
	    gyro.mode = gyro.MODE_GYRO_RATE
	    gyro.mode = gyro.MODE_GYRO_ANG

def move_clamp(direction):
    clamp_motor.run_direct(duty_cycle_sp=direction*60)
	def obstacle_ahead():
	    return push.value() == 1

def reverse():
    run_motors(-35, -35)
    time.sleep(0.5)
    stop()
    time.sleep(0.4)
	def is_red():
	    try:
	        global color_r, color_g, color_b
	        color_r = color.value(0)
	        color_g = color.value(1)
	        color_b = color.value(2)
	        if color_r < 5:
	            return False
	        if color_g <=2 or color_b <= 2:
	            return True
	        if color_r > (color_g+color_b)/2:
	            return True
	        return False
	    except:
	        traceback.print_exc()
	        return False

def stop():
    left_motor.stop(stop_command='brake')
    right_motor.stop(stop_command='brake')	

def stop_clamps():
    clamp_motor.stop()	

def release_clamps():
    move_clamp(1)
    time.sleep(1.5)	

def average_wheel_dist():
    return (l_motor_pos + r_motor_pos) / -2	

3.2.2 Avoiding bumping into the side walls

```
while not self.interrupt:
   l_motor_pos = left_motor.position
   r_motor_pos = right_motor.position
   us_value = us.value()
   if (not self.new_path) and (not us_value > US_THRESHOLD):
      print "new wall on left, us=", us_value
      time.sleep(0.1)
      self.new_path = True
      previous_wall_dist = us_value
   time.sleep(0.1)
   if self.new_path and us_value < US_THRESHOLD: # There's a wall on the left
      if time.time() - self.wall_time > 0.5:
         if us_value < 100: # If too close moving towards left wall
            current_wall_dist = us_value
            if current_wall_dist - previous_wall_dist < 0: # and converging with left wall
               print  "\nToo close to wall; veering right"
               print "offset =", self.offset
               self.wall_time = time.time()
               self.offset -= 5
               previous_wall_dist = current_wall_dist
         elif us_value > 180: # If too far from left wall and moving away
            current_wall_dist = us_value
            if current_wall_dist - previous_wall_dist > 0: # and diverging from left wall
               print "\nToo far from wall; veering left"
               print "offset =", self.offset
               self.offset += 3
               self.wall_time = time.time()
               previous_wall_dist = current_wall_dist
```

The above code runs in the thread concerned with running the robot in a straight line. Every 0.5 seconds the robot checks the current distance detected by the ultrasonic and compares it to the previous distance a half-second earlier. If the robot is too close to the wall and the distance is decreasing, it changes the angle at which the robot should be moving in a straight line, by decreasing self.offset, and vice versa with the robot being too far from the wall. The values for which the robot is too close or too far, were determined through centering the robot on top of an A3 sized piece of paper and measuring the distances from the ultrasonic and the edges. This section of the code only runs if there is a wall on the left for which the robot can judge whether or not it is too close or too far from.

3.2.3 Running Straight

```
if gyro_value + self.offset >= 3:
    self.running_straight = False
    run_motors(45, 60)
    print "robot on right: adjusting left"
elif gyro_value + self.offset <= -3:
    self.running_straight = False
    run_motors(60, 45)
    print "robot on left: adjusting right"
elif abs(gyro_value + self.offset) < 3 and self.running_straight:
    run_motors(50, 50)
    self.running_straight = True
    print "robot nominal"
```

The above code ensures that the robot runs in a straight line forward that is parallel to the walls. Using the gyroscope to detect its bearing, if the measured bearing is divergent from the expected angle by 3 degrees, it increases the power of one of the wheel motors until the bearing has been corrected.

3.2.4 Searching the maze

```
fred.start()
while fred.isAlive():
    time.sleep(0.1)
    # check if red
    if is_red():
        # Record the wheel distance, time taken and direction
        path_stack.append((average_wheel_dist(), time.time() - fred.wall_time, "found"))
        print "-------------- FOUND --------------------"
        print "rgb", color_r, color_g, color_b
        fred.stop()
        fred.join()
        Sound.beep()
        time.sleep(0.5)
        print path_stack
        uturn()
        return
    if fred.new_path and us_value > US_THRESHOLD:
        print "new path on left: us=", us_value
        init_rotation_val = average_wheel_dist()
        threshold_check = 0
        false_alarm = False
        while average_wheel_dist() - init_rotation_val < 360:  # Wait robot to be fully visible in path
            time.sleep(0.1)
            if threshold_check == 0.5 and us_value < US_THRESHOLD:
                false_alarm = True
        if not false_alarm and us_value > US_THRESHOLD: # Still path on left
            fred.stop()
            fred.join()
            path_stack.append((average_wheel_dist() - 360, time.time() - fred.wall_time, TURN_LEFT))
            fred = RunMotors(TURN_LEFT, rotate_offset)
            fred.start()
            while fred.isAlive():
                time.sleep(0.1)
        else:
            print "***** false alarm, keep going", us_value
    else:
        if obstacle_ahead() or color_b >= 60:
            print "obstacle ahead"
            fred.stop()
            while fred.isAlive():
                fred.new_path = False
            dist_travelled = average_wheel_dist()
            print color_r, color_g, color_b
            reverse()
            print color_r, color_g, color_b
            if us_value > US_THRESHOLD:
                path_stack.append((dist_travelled, time.time() - fred.wall_time, TURN_LEFT))
                fred = RunMotors(TURN_LEFT, rotate_offset)
            else:
                path_stack.append((dist_travelled, time.time() - fred.wall_time, TURN_RIGHT))
                fred = RunMotors(TURN_RIGHT, rotate_offset)  # Rotate clockwise
            fred.start()
            while fred.isAlive():
                time.sleep(0.1) # wait for turning to complete
```

The code for the wall following algorithm is run on a separate thread, and calls the other threads for moving in a straight line, and rotating left and right.

3.2.5 Turning left and right

```
if self.task == TURN_RIGHT:
    print "-- turning right\n"
    run_motors(40, -40)
elif self.task == TURN_LEFT:
    print "-- turning left\n"
    run_motors(-40, 40)
while abs(gyro.value()) < 85 and not self.interrupt:
    time.sleep(0.06)
stop()
```

Since the only difference between rotating left and right were the direction the motors were spinning, the robot could check the degree to which it had rotated in both scenarios, by checking the absolute value of the rotation. Although the robot is meant to rotate in right angles, the team adjusted, after experimentation, the bearing to 85 degrees. Otherwise, the checking condition would only stop if robot has rotated *beyond* 90 degrees.

3.2.6 Returning from the maze

Once the robot finds the robot, we make the robot reverse so that the clamps would not knock over the can while it spun around 180°. Following this, the clamps will open and then travel towards the can and close the clamps once it was within the reach of the clamps. The robot would then proceed to follow the path recorded during the search to get back to its starting position.

To travel back the way we came from, we used the stack generated from the Search function. Using a stack allows us to use first in-last out, which is perfect for returning the way we came from.

```
while len(path_stack) > 0:
  target_distance, target_time, direction = path_stack.pop()
```

The only difference was that we would turn the opposite way than the direction of the tuple as the returning direction would be mirrored. During testing, we discovered that the number recorded would not always be 100% accurate. For example, the first instruction since our robot's position after rescue was slightly different to the robot's position before rescue. But more importantly, sometimes our robot would overshoot the path and keep hitting the wall. To prevent this from happening, we implemented a function from Search which was, if the touch sensor was activated, it would turn left/right so that it would turn rather than continually ramming into the wall.

```
if push.value() == 1:
  return_thread.stop()
  return_thread.join()
  if average_wheel_dist() < 150 and direction == TURN_LEFT:
    right_buffer()
    eturn_thread = RunMotors(FORWARD, rotate_offset)
    time_elapsed = 0
    return_thread.start()
    continue
  else:
    break
```

Once the robot has finished returning to where it started, it opens it's clamps to release the can to signal that it has finished.

4. Results and Analysis

Our robot performed quite well during the rescue competition scoring a total of 15/20. No other groups were able to achieve full marks in the intersection section and we don't know how the final mark would have been allocated so we were completely satisfied with the 4 out of 5 we received. Our biggest mishap occurred during the simple maze phase, in which it was thought full marks could have been easily achieved. Furthermore, the loop was an area of significant faults.

Stage	Simple Maze	Loop	Intersection
Score	8/10	3/5	4/5

4.1 Simple Maze

In the first the robot managed to achieve an 8 out of 10 for reaching the can and returning to the starting point. However, it was unable to transport the can to the starting point.

The fault that prevented the robot from finding the can was caused this was the robot's lack of a system to enable it to search for the can in the case that it wasn't directly in front of it. It was simply unfortunate that the can wasn't directly in front of the colour sensor during the competition (figure 4.1). Despite this letdown, our mark was still relatively good due to our robot's ability to navigate through the simple maze with ease. The directional correction system that read and responded to information from the ultrasonic sensor (regarding its proximity to the left wall) was very effective in keeping the robot from veering course and running into nearby walls. Furthermore, our robot was awarded points for its ability to return to the starting point. We were able to achieve this due to the robot's path-memory system which it used to reverse its steps once reaching the can. It was through the implementation of the directional- correction and path mapping systems that our robot was able to perform quite well but it was disadvantaged by the fact that it could not find the can.

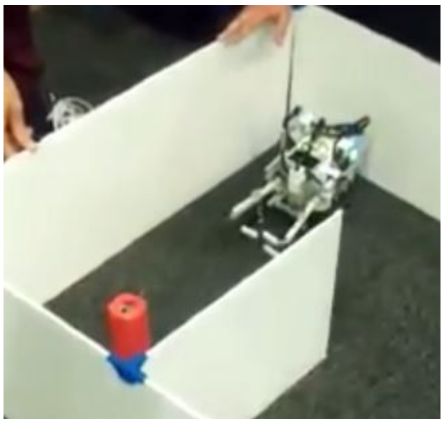


Figure 4.1 (the robot faces away from the can after failing to notice it)

4.2 Loop/Intersection

The next test the maze consisted of an intersection and a loop. Overall our robot managed to score a 4 out of 5 in the intersection and 3 out of 5 in the loop. Running through the maze, our robot passed the intersection with ease and minor disturbances to the walls. However when it got to the loop, our robot diverged slightly from the path and due to the butterfly effect, it eventually drove diagonally into a wall, messing up the calibration thus not being able to collect the can. On the second try, our robot managed to reach the can and bring it back to its starting position, however by the time our robot returned to its original position the timer had run out

- Our robots relatively compact design meant that maneuverability through the maze wasn’t a challenge unlike some other robots we saw.
- The clamps worked as they intended, grabbing the can without losing it on the way back as well as not obstructing the navigation of the robot.
- Our strategy for moving in straight line cost us to miss the can, as it swerved slightly away from the can.

5. Improvements

5.1 Being more practical:

When our group was coming up with ideas we only verbally discussed them and went with the one that we best thought would the job more appropriately but we overlooked some key factors and hence we weren’t able to achieve the results we wanted. We think that a better way of choosing the functions of our robot was to implement the top 3 features that our team came up with and hence decide which would work most effectively in getting the job done. In this way we would have practically tested out all our designs and hence give us the satisfaction that we are heading in the right direction.

5.2 Testing

Testing of the robot was a necessary part of the process and that is one of the weakness of the group. We didn't finish the robot until the last moment hence we weren't able to thoroughly test and make it less vulnerable to flaws. An effective way of testing could be through trying out different combinations of hardware and hence it will point out apparent weaknesses on our robot.

5.3 Stability of the clamp

When we attached the clamp onto the robot we realised that the robot has become unstable and needed to do something about. We tried evening out the weight and distributing the weight evenly throughout the whole robot. To implement the clamp more successfully and making it as compact as possible we attached the claw to motor using a series of gears. When we were running tests in the day of the task we felt as if there was friction between the robot and the ground and it was being heightened by level of instability in our robot. We can improve this problem through making the robot's weight more evenly distributed by shifting the mass towards the back as the clamp would be carrying most of the weight.

5.4 The Code

There were many problems associated with developing the code that the robot would use to navigate through the maze, and can be broken down into the following problems. The light sensor and the ultrasonic sensor would have to be calibrated and when they did they would go all whack. This was an unpleasant task as it was very annoying but a solution to this problem is to be patient and allow for the sensor to turn back on. Restarting the program helped our sensors.

Getting the sensors and the motors took forever and our group wasn't sure exactly on what data they would return and how they would be controlled. This could be better managed next time as we could experiment with different values and see which one works best for us.

A great threat that our robot faced was it not being able to recognise intersections as it could not detect where to turn as the wall were either too far away to too on an angle . For example, as a turn was being performed, sometimes the robot would turn the way it is supposed to but other times it will detect something else and turn some random way. This was caused by the gyroscope and the ultrasonic sensor as they would not give the correct readings. This made it really hard for the robot to go through the maze without bumping into a wall. Members from team tried to work out solutions for the problem but we couldn't really do anything but just go with it. But our team thought about it and how it could be fixed and we soon came to a realization that we could implement into our code a way of keeping the robot in line, whenever it turns too much or not travelling straight it will align the robot with the maze.

5.5 Universalism

Even though this isn’t a major issue in this project, but the robot we have created and programmed could not work for some certain maze, so in order for us to make sure it works for all mazes we should test it as much as possible as mentioned earlier.

5.6 Too much friction

There was too much friction and it was slowing our robot down and it was also allowing our robot to navigate slight of its paths. The ball at the front of the robot was not a good implementation as it caused the robot to sway into different directions too much. Also there was also friction caused by the wires that were hanging loose from the robot which interfere with the surrounding walls and caused inaccuracies when our robot was turning. This could be improved in the future by making the wires compact and making sure they don’t ram everything that the robot goes past. The ball that was causing the heaving distilment of movement could also be improved by attaching support beans which wouldn’t allow the robot to sway off path.

5.7 Another sensor or motor

Our group could have implemented another sensor which would make the robot to go straight so much easier and hence allow it to travel through a straight line. This is a consideration for next time.

5.8 Rotating ultrasonic sensor

We had experimented with a rotating ultrasonic sensor during the Robo-Sumo competition and found that while it was effective in that competition, the motor controlling it was imprecise, meaning that the ultrasonic sensor, when returning to the front would usually be off by a couple of degrees to the left or the right. Because of our awareness of this, we decided to scrap the rotating ultrasonic sensor as we thought it would make it difficult for the robot to have a clear understanding of its position within the path of the maze. However, by doing this, we sacrificed our ability to have 360 degrees of vision, which would come in extremely useful as we could easily identify all the possible paths that would be available to the robot at any time. By deciding against a rotating ultrasonic sensor, we were essentially blind to the whole right side of the maze, which meant that if we were placed into the maze in a certain position, we would be stuck in a loop, and would have to rely on our loop detection code to break us out of the loop, which would waste valuable seconds (as the code only detects a loop if the robot has done 2 loops) with a time limit of only 5 minutes.

Our decision against the rotating ultrasonic hurt our robot more in the rescue phase of our robot. Because our ultrasonic sensor is on the left side of our robot, when we return from the maze, all left turns will turn into right turns, which are essentially invisible by our robot (since there is no ultrasonic on the right). Because of this, we had to work hard on our software to create a competent fix, by storing our distance travelled and time taken for each path in our search phase, and creating the right_buffer function, which would allow the robot to try turning into a new path multiple times if it tried to turn right before the path on the right had opened up.

5.9 Not using the touch sensor

To let our robot have an understanding of what lay ahead of it, we placed a touch sensor on the front of our robot, which would activate when we pushed up against a wall. While this was very useful, and gave our robot more spatial awareness, it was an inelegant solution as the walls would keep getting pushed forward, requiring someone to continually hold all the walls down. Moreover, it meant that we would have to reverse a little after hitting the wall before turning either left or right, as we would be too close to the wall. This extra movement in an opposing direction to our normal motion brings about a lot of errors, especially in the recording of our directions in our path stack. Because of this small reverse, the time value we store for each path will be slightly different. While these changes are small, over the period of a long maze, they can easily add up and result in our robot not reaching the exit. Another large issue was that our touch sensor could only really register a touch if it was pushed in a 60 degree arc from its centre. That meant that if your robot hit a wall at a large enough misalignment, our touch sensor would not activate and our robot would be stuck indefinitely.

Another strategy we could have used instead could have been the colour sensor. Instead of just using it to detect red, we could have programmed the sensor to detect a white wall. This would make it act just like a touch sensor, except it would activate at a further distance away from the wall, it wouldn’t disturb the maze, and it would work on all angles.

6. Conclusion

Our performance in the first round was very well done as we scored a 8 which really boosted our self belief, as our robot was able to navigate through the maze but unable to pick up the can. It went through the maze and followed all the commands that had been coded when it saw the bottle via its colour sensors it comes to a full stop and makes a beep.

As designers, we mainly looked at the structure of the robot, and also the functions behind the robot such as where all the weight of the robot was equally balanced. We were able to think like engineers and approached this problem in which an engineer would do.

However our robot’s performance during the RoboRescue task was only satisfactory, due to last minute technical problems, our program failed to initiate the return function which was meant to guide the robot back to its starting position. The error was not solved due to time constraints. Also our robots constraints came into play where it saw different walls and it started going anywhere it wanted. This caused the robot to goo of track and hence was not able to go back where it came from.

Through this design process we have learned that the most important link between the design process and teamwork importance of working together. The design process played a significant role in our final results.

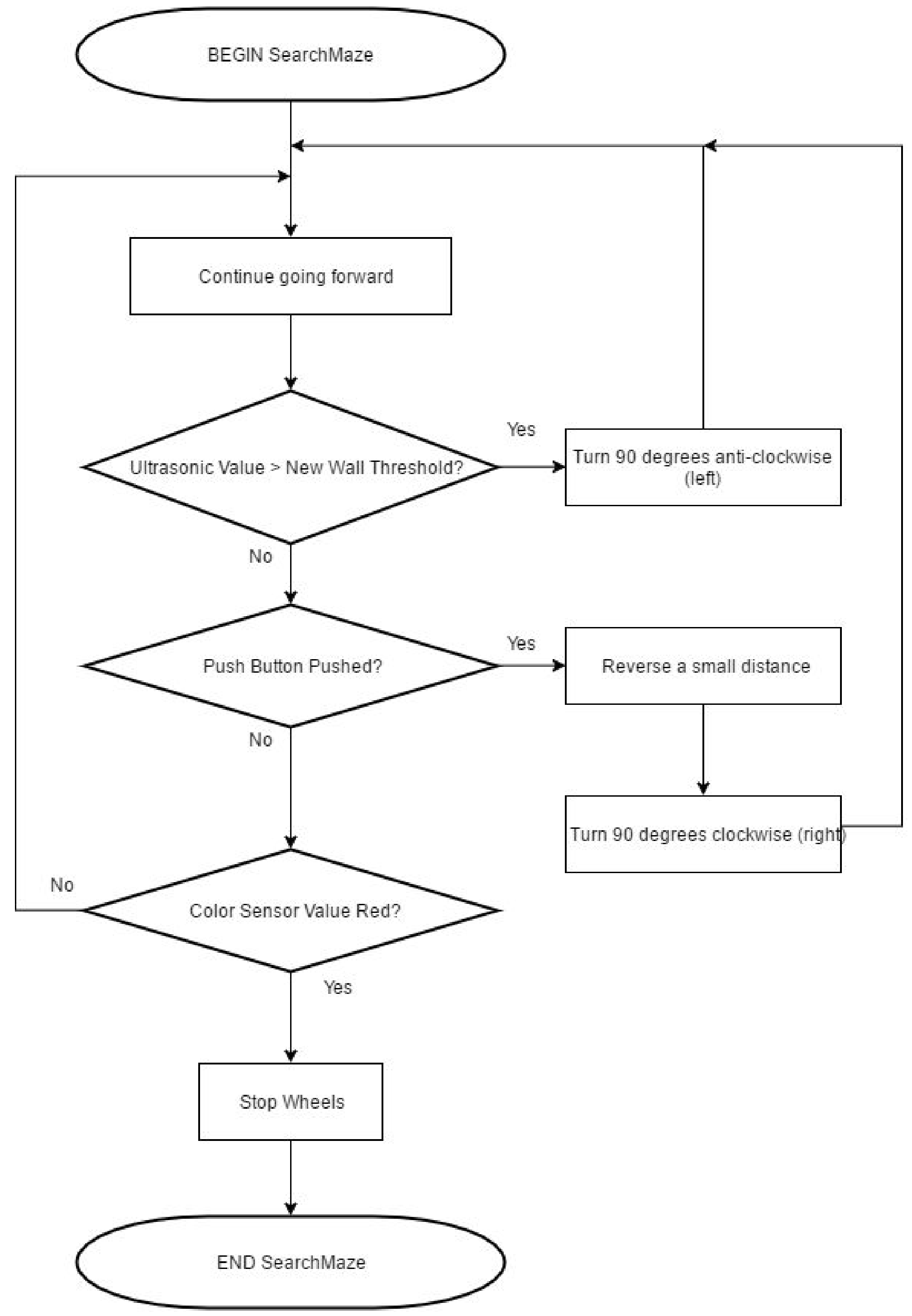
7. Acknowledgements

We would like to thank our lecturer Claude Samut for his informative lectures and facilitating to our needs.
We would like to thank Hayden Smith for his helpful insight and guidance in our many tutorials with him.
We would like to also thank all the people who helped make the whole project happen, and those who helped run both the Robo-sumo and Robo-rescue events.

8. Appendices

Appendix A - Conceptual Design (Software)

Appendix A1. Maze Searching


BEGIN SearchMaze
Continue going forward
Ultrasonic Value > New Wall Threshold?
Yes
Turn 90 degrees anti-clockwise (left)
No
Push Button Pushed?
Yes
Reverse a small distance
Turn 90 degrees clockwise (right)
No
No
Color Sensor Value Red?
Yes
Stop Wheels
END SearchMaze