2020

CS26010 Robotics Practical Assignment

MAZE RUNNER: HYBRID CONTROL

OLLIE THOMAS – OLT13 - 180001541

1 Introduction

For this assignment, I put together the algorithms I had written this term, merging them into one hybrid controller to allow a robot to explore a maze and record information about it. I believe that my controller fulfils most of the brief set. It is able to navigate through the entire maze (only occasionally bumping into walls), to notify an onlooker that it has found a nest area, to store where the nest areas are in the maze, to store where the walls are in the maze (although this is not as accurate as it should be), and when the maze is complete the robot displays the stored maze information on the LCD through a series of lines for walls and an 'x' for nest areas.

2 Controller Design

2.1 A Brief Overview

My hybrid controller works by breaking down the desired behaviour into smaller components within separate functions. The functions I chose are detecting when the robot crosses a line, driving forwards straight for a distance, turning, detecting obstacles, and detecting whether the robot is in a nest area. These functions are then called in a sequence and some are linked through returned values.

2.2 A List of Behaviours and Control Strategies

2.2.1 Driving Straight for a Distance

To drive straight, I used a proportional controller that calculates an error between the motors using the encoder values and updates the drive accordingly (Snippet 1). By only updating the 'slave' motor power, it gives the robot a greater chance of achieving the symetry between the motors that is required to drive straight.

```
FA_SetMotors(masterPower, slavePower);
totalTicks += FA_ReadEncoder(0);
error = FA_ReadEncoder(0) - FA_ReadEncoder(1);
slavePower += error/k;
FA_ResetEncoders();
```

Snippet 1: Detecting error between the encoder values

Driving for a set distance is achieved simililarly. A target number of ticks is calculated by multiplying the required distance by the distance travelled per encoder tick. Once the total ticks from the encoders excedes the target, the robot has travelled the distance required and can move onto another action (Snippet 2). Programming this way enables other actions to be performed whilst driving, such as detecting obstacles or lines. Furthermore, using a timeout timer (Snippet 3) ensures the robot does not get stuck in a loop driving straight and reverts to a default if the target is not reached within an expected period of time.

```
if (totalTicks > target) {
    state = 3;
}
break;
```

Snippet 2: Testing whether the robot has travelled the target distance

```
if (FA_ClockMS() > timer) {
    state = 1;
}
```

Snippet 3: The timeout state

2.2.2 Turning Precisely

I also used a proportional controller to turn a precise angle. To find the proportional constants used in Snippet 4, I made the robot turn in circles five times, read the encoder tick values and calculated how many ticks passed per degree for each motor (similar methods are used for the driving constants). This particular controller waits for both motors to reach their respective targets before moving onto the next state.

```
int targetL = angle * tickL;
int targetR = angle * tickR;
if (FA_ReadEncoder(0) > targetL && FA_ReadEncoder(1) > targetR) state = 3;
if (FA_ClockMS() > timeout) state = 3;
```

Snippet 4: Setting the encoder targets, testing whether the robot has turned the correct amount or timing out

2.2.3 Detecting Lines

Detecting when the robot has crossed a line is important for knowing where the robot is in the maze. I used the line sensor underneath the robot to detect changes in the reflectivity of the floor. When the value from the sensor returned falls short of a determined threshold the robot has passed over a darker area. The function tells the mapping function that the robot has changed location.

2.2.4 Detecting Obstacles (while driving)

As I am using a series of state machines (section 2.3), the robot can perform several actions almost simultaneously. Hence, while the robot is driving, it can detect whether it is about to drive into a wall and perform evasive action if necessary. This function makes use of the infrared sensors all around the robot and pre-determined thresholds (described further in section 4).

2.2.5 Detecting Nest Areas

Comparable to sections 2.2.3 and 2.2.4, detecting the change in the environment when in a nest area is down to the values received from the ambient light sensor on top of the robot and a threshold programmed into the robot. When the value of the sensor drops below the threshold, the robot signifies that it has found a nest by flashing an LED.

2.3 Combining the Strategies

These control strategies are combined using a state machine in each function; enabling the robot to perform more than one action or sensory reading seemingly at the same time.

In order to use state machines, I have broken each behaviour into several constituent parts that represent different 'states' within that action or reading. For example, the detecting nest function (Snippet 5) is split into three states that continuously loop: checking if the robot is in the nest, flashing the LED, checking if the robot has left the nest area.

```
switch(state) {
    case 1:
        if (lightReading < 1000) {
            state = 2;
            timer = FA_ClockMS() + LEDWAIT;
            result = 1;
            map.grid[map.robotPos[0]][map.robotPos[1]].nest = 1;
        }
        break;
    case 2:
        FA_LEDOn(0);
        while (FA_ClockMS() < timer) {
            return result;
        }
        FA_LEDOff(0);
        state = 3;
        break;
    case 3:
        if (lightReading > 1500) state = 1;
        break;

}
```

Snippet 5: The nestDetect state machine

State machines enable the different functions to work in tandem. If a state is incomplete, another function is called, where it remembers its previous state and checks whether that part is complete. If it is complete, then that function's state is progressed, and the program moves on. Otherwise the function breaks to check the next function's state. Hence, if the functions are called one after another, they seem to run simultaneously, as the individual processes do not take up any unnecessary time.

3 Internal Maze Model

3.1 The Structure

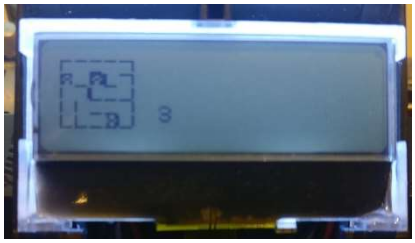
The internal structure for storing data about the maze consists of two C structures. The first, Square, is used to hold information about individual cells within the maze. It has attributes for whether there are walls to the left, right, top or bottom, whether it has been previously visited, and whether there is a nest area at the square. The second structure, Grid, is used to hold information about the whole maze. It has attributes that hold the current position of the robot, the orientation of the robot (in relation to the maze), and it has a 2D array of Square structures that represents the maze. A global variable of type Grid is declared so that all the functions in the program have access to the same map for the maze.

```
typedef struct {
    //walls: 1 is wall 0 is no wall
    int left;
    int right;
    int top;
    int bottom;
    //if robot has seen it 1/0
    int visited;
    int nest;
} Square;

typedef struct {
    Square grid[5][5];
    int robotPos[2];
    int orientation; // 1:top, 2:right, 3:left, 4:down
} Grid;

Grid map;
```

Snippet 6: The structures that map the maze

3.2 The Connection to the Behaviours

The map is updated by the 'mapping' function, which updates the position of the robot depending on its orientation and places the walls of the maze. This function is called along with the robot's other behaviours if a line has been detected by the robot; meaning the map is constantly being updated as the robot moves around the maze from square to square. If the square was previously unvisited, the function calls the 'place walls' function, which gets readings from the infrared sensors to work out where the walls are.



Picture 1: This shows the display of the robot once it has explored the maze

Another way the behaviours are connected to the mapping of the maze, is that when the robot turns, the orientation of the map needs to change. This is accomplished through the 'change orientation' function being called when the robot turns left, right or 180˚. The function must consider which orientation the robot is already in to find the correct new orientation.

4 Sensors [10]

4.1 Obstacle Sensors Research

As part of this project I had to deal with different kinds of sensors that all play an essential part to make the robot behave how it is supposed to. However, as sensors must deal with the real world, their readings can sometimes be unreliable. For this reason, I had to research what the correct values and thresholds should be.

I focused my research on the infrared sensors and how their readings changed over the distance to the object and the colour of the object, as these are particularly important to make sure the robot does not crash into any walls. The research consisted of taking 5 readings from the sensor at a set of distances and plotting the averages. The resulting graphs enabled me to understand more about how the sensor behaves, giving me also a suitable range of values that could be used as thresholds for my robot. I did this for a black and a white object and the results are shown in the figures below.

Distance (cm)	Reading					
	1	2	3	4	5	Average
1	3829	3845	3879	3864	3866	3856.6
2	2741	2746	2729	2709	2734	2731.8
3	1502	1497	1495	1531	1557	1516.4
4	911	917	902	926	942	919.6
5	477	495	471	456	461	472
10	75	78	77	88	93	82.2
15	20	27	24	20	23	22.8
20	11	14	13	20	19	15.4
25	9	13	11	9	5	9.4
30	3	6	3	9	10	6.2

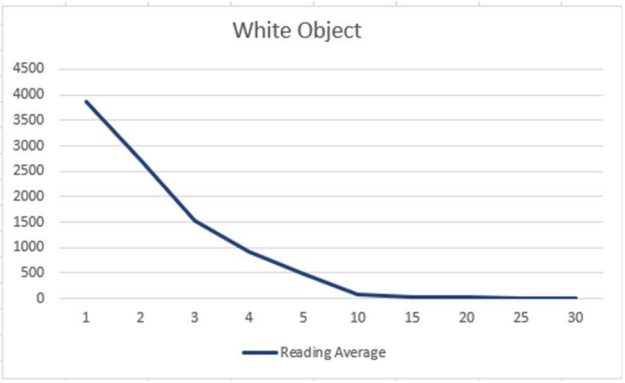


Figure 1: Data from experiments on a white object

Distance (cm)	Reading					
	1	2	3	4	5	Average
1	446	450	459	427	410	438.4
2	94	83	82	103	110	94.4
3	14	11	19	29	21	18.8
4	6	15	3	10	10	8.8
5	2	6	2	3	3	3.2
10	0	3	8	3	1	3
15	0	3	4	3	6	3.2
20	0	0	0	0	0	0
25	0	0	0	0	0	0
30	0	0	0	0	0	0

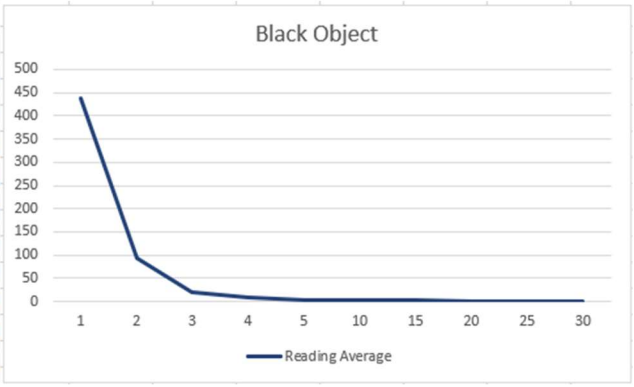


Figure 2: Data from experiments with a black object

The results clearly show a trend that as you get closer to both objects, the reading from the sensor sharply increases. However, the results also show that the sensors can detect white objects from much further away and the range of values that could be expected when detecting a white object is much greater. This is due to the reflectivity of white being greater than that of black.

4.2 How I used this Information

Snippet 7 shows the code that stops the robot from bumping head-on into walls. The numbers 700 and 500 both correspond to a distance between 4 and 5 centimetres away from the wall (information used from Figure 1). I then tested these values in the maze, and they ensure that the robot does not hit the walls head-on.

```
if (FA_ReadIR(IR_FRONT) > 700) {
    state = 3;
} else if (FA_ReadIR(IR_FRONT_RIGHT) > 500) {
    FA_SetMotors(masterPower, slavePower + k);
} else if (FA_ReadIR(IR_FRONT_LEFT) > 500) {
    FA_SetMotors(masterPower + k, slavePower);
}
```

Snippet 7: Shows the use of the researched correlation between IR readings and the distance from an object

Another example is in Snippet 8; showing how I used the research to ensure the robot explored the whole maze. Snippet 8 is run when the robot enters a square and is deciding where to go. First it looks to see if the left path is clear (i.e. there is no object within 10cm to the left which corresponds to an IR reading of less than 100), if it is clear then it goes left. If there is an object to the left, it tries the front, then the right, before finally turning around if there is no other option.

```
if (FA_ReadIR(IR_LEFT) < distance) state = 4;
else if (FA_ReadIR(IR_FRONT) < distance) state = 1;
else if (FA_ReadIR(IR_RIGHT) < distance) state = 5;
```

Snippet 8: This shows the use of the IR readings to determine which direction to go

I also did similar research for the other sensors I used, such as the light sensor on top and the line sensor underneath. I used this research to find the thresholds required for the robot to behave in the way set out by the brief.

5 Difficulties and Problems encountered

I ran into three main issues when completing this assignment. The first being the robot not turning exactly 90˚ or 180˚ every time when going round the maze, hence it would bump into the walls. For some reason my robot still occasionally turns more or less than it should. This has been confusing because I could not see any reason for this behaviour to be caused by the code. If I had been able to test this in more depth, I think that I would have been able to resolve the issue.

This leads into the second difficulty I faced, which was that my code to avoid bumping into the walls was hard to fine tune. To begin with, I overestimated the value for both the infrared sensor and the change in the motor speed in order to avoid the wall. This led to the robot swerving from side to side, hitting both walls as it went. So, I then decreased how much the wheels sped up, but this caused the robot to just continue dragging along the wall. Finally, I managed to get a balance so that when the robot moves slightly towards a wall it should automatically straighten up.

The third difficulty was getting the robot to map the maze correctly. Sometimes the robot puts walls in the wrong place or does not register ones that are there! I presume that this is an issue with the values I used for the sensors. Again, with a little more time I would have been able to address this issue. Furthermore, the robot would occasionally not count the number of squares it had visited correctly, so it would carry on exploring when it should stop.

6 Conclusion

To extend my controller, I would connect the robot to a serial Bluetooth receiver on a mobile. I would use this to send information about the maze; making it more accessible while the robot is exploring its environment. This would also make the information gathered more readable, as the current LCD format is cramped. I would also make the robot beep once upon finding a nest and twice when it has finished exploring, to ensure that these events are more easily recognised.

After looking at the brief and writing this report, I think that I have been able to complete most of the project to a high standard. Given more time to tweak some algorithms and sensor values mentioned in section 5, I believe that I would have had a fully functioning ‘maze runner’.