IMAT3404 - Mobile Robotics

Implementation of a Robot Controller

Michael Bull, P12190492

22nd October 2014

Module Leader: Dr. Benjamin Passow, benpassow@dmu.ac.uk

Project Supervisor: Pamela Hardaker, pamela.hardaker@dmu.ac.uk

Contents

1 Introduction **1**

2 Literature Review **1**
2.1 Feedback Control 1
2.2 High Level Control Methodology 2
2.3 High Level Control Architecture 3

3 Behaviour Design **4**
3.1 Obstacle Avoidance 4
3.2 Wandering 4
3.3 Edge Following 4

4 Architecture Design **5**
4.1 Finite state machine 5

5 Test Results **6**
5.1 Test Maps 6
5.2 Test Data 9
5.2.1 Root-mean-square error 9
5.2.2 Map 1 10
5.2.3 Map 2 11
5.2.4 Map 3 12
5.3 Observations 13

6 Conclusion **13**

Appendix A Source File **15**

Appendix B Header File **21**

References **23**

List of Figures **23**

Acronyms **23**

1 Introduction

For this project there has been a software requirement of an autonomous mobile robot controller that is designed and operates according to a set specification. This report will discuss and evaluate design and architectural methodologies, indicate design and implementation choices, and provide test results with a conclusion based on the controller's performance.

The controller itself will be written in the C++ programming language, utilising the ARIA robotic controller library. The ARIA library allows us to dynamically control the velocity, angle, and motion of the robot, as well as retrieve sensor data such as sonar readings. The controller will then be compiled and ran using the PeopleBot robot through the MobileSim simulator. This simulation will allow the observation and testing of the controller throughout development, providing a test driven development workflow that accurately represents the effectiveness of the programmed controller.

2 Literature Review

For this project to succeed we must first review available literature to ensure that our design decisions will be informed and correct, based on information provided by academics in the field.

2.1 Feedback Control

As the specification outlines a requirement of edge based following, a feedback control methodology must be applied for the robot to dynamically adjust its velocity and angle based on the feedback it has received from its sensors. This feedback is gained with the use of sonar readings and provides information on how close an obstacle may be to the robot. The three readings used include sonars placed on the front, left, and right of the robot. With the readings taken, they must then be applied to a 'transfer' function to determine any change in velocity and heading. Transfer functions can be very simple, for example using simple binary rules to determine whether our readings are 'above' or 'below' the setpoint. However, simple binary control rules do not perform well when the readings are fluctuating around the setpoint. With the setpoint being defined in the specification as 0.5m, the robot will want to be following the edge at this distance consistently, meaning that it may fluctuate and go slightly over or slightly under the setpoint whilst following the edge – for this reason binary control rules will not be suitable for the project.

A more appropriate feedback control methodology is the proportional-integral-derivative (PID) control algorithm. This algorithm is 'the most popular feedback controller used within the process industries' and 'has been successfully used for over 50 years' (Willis 1999). Its wide adoption in industry is related to 'its simple control structure and ease of design' (Li 1998). PID applies itself moreso to this project than binary control rules as it is only marginally more complex to implement and has been used widely in industry, thus

providing firm reassurance from qualified experts that it performs efficiently and correctly.

2.2 High Level Control Methodology

With an appropriate feedback control methodology selected, we must now choose how to control the robot from a more ‘abstract’ higher level. This is done by selecting a control methodology (deciding how the robot will control itself over time) and then applying this methodology with the use of a high level architecture, providing the robot with decisions it can make for more rational behaviour.

There two most widely used control methodologies are ‘reactive’ and ‘deliberative’. Reactive control provides a non-linear mapping of multiple inputs to outputs. This means that the robot may take a multitude of inputs, process all of them and result in a more well informed output. The processing of these inputs is typically fast, however the tasks that the robot may process typically lack depth, and information about each task is not retained over time.

Deliberative control works with three core steps:

1. **Sense**
   The robot will use its sensors and gain all relevant information about itself and the world it is in.

2. **Plan**
   The robot will plan its move for the next cycle based on the information that it sensed in the previous step.

3. **Act**
   The robot will then act out the plan that it computed in the previous step.

This approach is generally used for ‘goal driven’ tasks, such as a robot attempting to reach a specific part of the map, however it is much simpler to implement than reactive control. Deliberative control can also be easier to visualise from a developer’s stand point, as it is easy to identify which task the robot is currently computing.

The downfalls of this approach mainly lie in reaction to dynamic events; as the methodology is reliant on planning, any dynamic events that were not expected during the planning phase may confuse the robot. Fortunately the specification does not outline any dynamic events in the maps, and therefore the assumption can be made that the maps the robot must traverse will be static, thus making this a non-issue.

As it is easier to implement and does not have to react to dynamic events, deliberative control has been chosen as the more appopriate methodology.

2.3 High Level Control Architecture

With the methodology selected, we must now provide the robot with the tools to decide how to use the methodology in the most appropriate manner. This is done with a high level architecture. The two most common architectures are motor schemas and finite state machines.

Motor schemas provide a biologically inspired architecture that incorporates heavy use of parallelism. This is achieved by the robot creating a form of 'muscle memory' as it continues to interact with its world, slowly learning until each action is stored in its memory and ready for it to be utilised in future. This architecture is very scalable and appropriate for large controllers as the parallelism provides the robot with use of all processing units available to it. However, with the added complexity of multi-threading and general complexity of process sequencing, it is clear that this architecture is aimed towards projects that are much bigger than this, therefore it has been deemed as not appropriate.

The chosen architecture will be a finite state machine (FSM). A FSM provides a linear one-to-one mapping of each behaviour and a potential 'state' for the controller to be in. These states are ordered synchronously and each one represents one of the potential behaviours the robot can exhibit. The transition between states occurs when a set of criteria are met, typically checked for in each processing cycle. This simple and powerful implementation provides a clean implementation that outlines the various behaviours of the controller, without the added complexity of parallelism. Sacrificing the more complex motor schema architecture does mean that the controller is therefore not as scalable, however given the size of this project it has been deemed a non-issue.

3 Behaviour Design

3.1 Obstacle Avoidance

The first required behaviour is ‘obstacle avoidance’. This involves the robot being aware of its surroundings and not coming closer than 0.1 meters to any obstacle on the map. This can be implemented with the use of sonar readings provided by the ARIA library.

The robot can check for obstacles in front of it using a sonar reading between $-20°$ and $20°$. The reading will then be checked to determine whether an object is within 1.1m in front of it, and if so will begin to decellerate until reaching the 0.1m threshold. At this point it will then begin to rotate $30°$ per cycle until the reading shows that there is no longer an object in front of the robot.

3.2 Wandering

When the robot is not currently avoiding an obstacle, the behaviour it should elicit is that of a ‘wandering’ mobile unit. The robot should wander the map at a speed of 0.5m/s for a random distance between 0.5m and 1.5m. Once it has wondered for the randomly chosen distance, it will then begin to turn for three seconds until reaching a randomly chosen angle between $20°$ and $160°$. This will ensure that the robot is simply not going from one side of the map to the other endlessly, as it will have a random element to its planned path. Prior to the turning phase, the robot will decellerate in a similar fashion to the obstacle avoidance behaviours, slowing down to a near halt before performing the turn. This will ensure that the turn is executed before the robot continues to wander and encounter any unexpected obstacles.

3.3 Edge Following

When the robot detects an edge on either its left or right, it will attempt to follow the edge to the best of its ability. This is done with the use of the PID algorithm outlined earlier in the literature review. Once detecting a wall that is 1m or closer to either the robots left or right side, it will transition its state and begin to follow that wall until either the wall ends, or an obstacle appears in front of the robot.

4 Architecture Design

4.1 Finite state machine

The FSM discussed earlier in the paper was implemented with five different states:

- WANDER
  The wander state is the default state of the robot. During this state the robot moves at full speed (0.5m/s) in a straight line. Whilst wandering, the controller continually checks for a set criteria to be fulfilled in order for it to transition into a different state. These include checking if there is an obstacle in the near distance of the robot (in order to begin to decelerate and transition to the OBSTACLE_INFRONT state), checking if there is a wall on the left or the right of the robot in order to transition to the edge following behaviour (either the FOLLOW_WALL_LEFT or FOLLOW_WALL_RIGHT state), and finally checking if the robot has wondered for a random distance between 0.5m and 1.5m, and if so transitioning to the TURN_RANDOMLY state.

- OBSTACLE_INFRONT
  During this state the robot will attempt to avoid an obstacle that it has sensed in the near distance. This is done by decelerating and then turning either left or right, depending on which direction has an opening. When there is no longer an obstacle sensed in the near distance, the state will transition back to WANDERing.

- TURN_RANDOMLY
  This state is entered when the robot has wandered in a straight line for a random distance between 0.5m and 1.5m. This ensures that the robot does not continue on a straight path forever, and instead has an element of randomness to it. Once it has travelled the specified random distance, it will turn for a random angle between 20° and 160°, two seconds after which it will return to the WANDER state.

  The random distance is checked each cycle whilst in the WANDER state by using the Pythagorean theorem and calculating the distance travelled between the coordinates when we started wandering and our current coordinates this cycle. As soon as the distance is calculated as more than the randomly selected value (between 0.5m and 1.5m), the state is transitioned.

- FOLLOW_WALL_LEFT & FOLLOW_WALL_RIGHT
  These two states both have identical but opposite behaviours, as they apply either the left or right sonar reading to the PID algorithm discussed earlier in the report. The algorithm will then use these readings to calculate the error it currently has to the setpoint (the setpoint being the consistent distance the robot should follow the wall at) and apply it to the PID algorithm. If at any time the robot should wander too far from the wall (indicating the wall has finished), the robot will transition back to the WANDER state. Similarly, even whilst following a wall, the controller will check for obstacles in front of the robot and act within the same cycle if detecting anything, thus reacting immediately and slowing the robot down before colliding with an obstacle.

5 Test Results

To test the controller's implementation, three maps were constructed with the use of the Mapper3Basic program. These are shown in the figures below, with an example of the robots traversal path in each (shown in red). The figures display how the robot successfully traversed a single path through the map, and will be the basis for our test data.

5.1 Test Maps

MobileSim

File View Clock Help

p3dx-sh: Port opened. Listening on TCP port 8101... [948]
p3dx-sh: Ready for a client to connect on TCP port 8101.
p3dx-sh: Client connected from GH582-015 (127.0.0.1)
p3dx-sh: Laser on
p3dx-sh: Got CLOSE command, closing connection.
p3dx-sh: Client disconnected.
p3dx-sh: Port opened. Listening on TCP port 8101... [1080]
p3dx-sh: Ready for a client to connect on TCP port 8101.

Time: 0:0:01:20.700 (avg sim/real time: 0.92) 0 active devices MobileSim 0.7.2 (Stage v2.0.0a-amr11)

Figure 1: The first test map

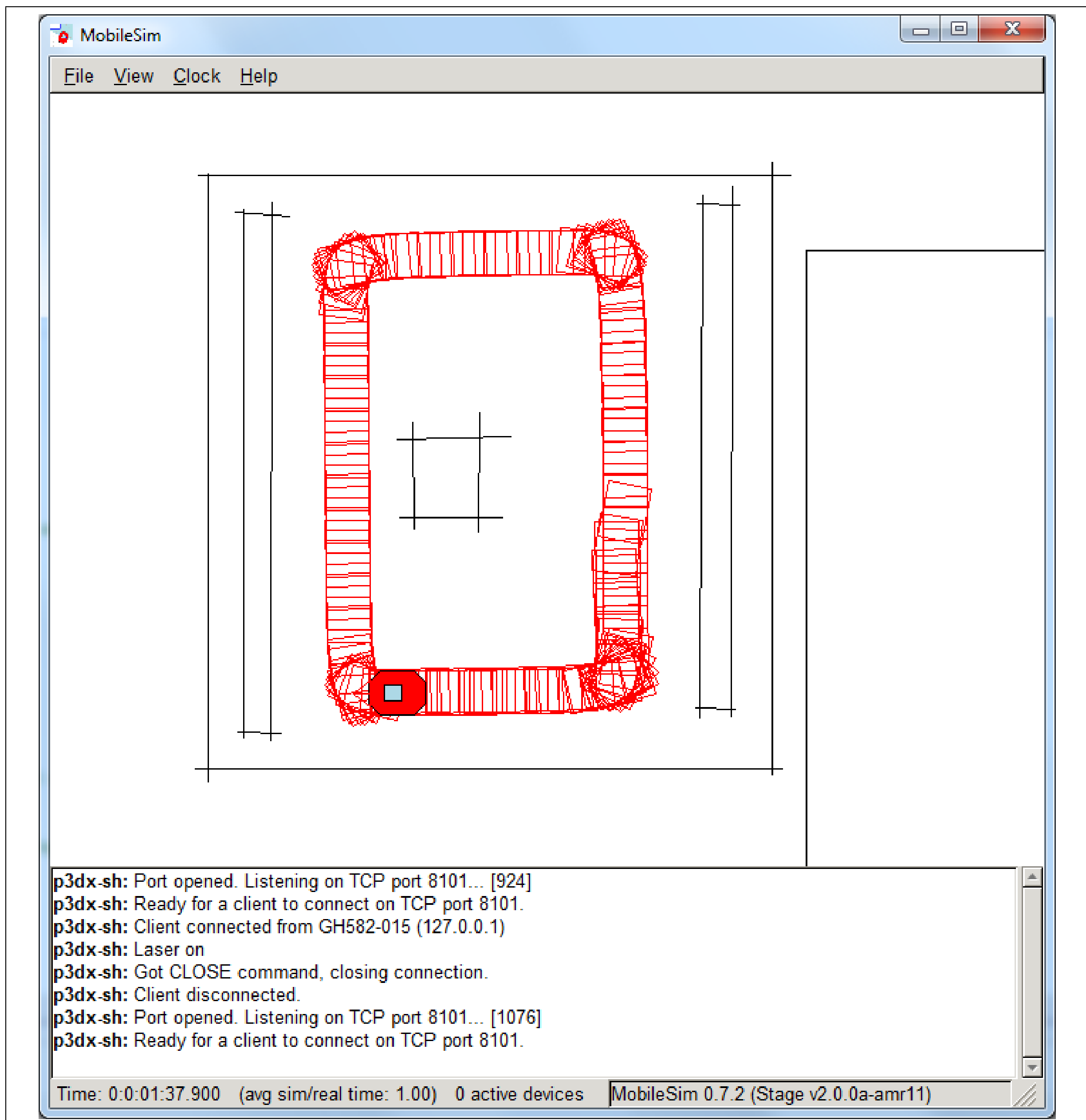

Figure 2: The second test map

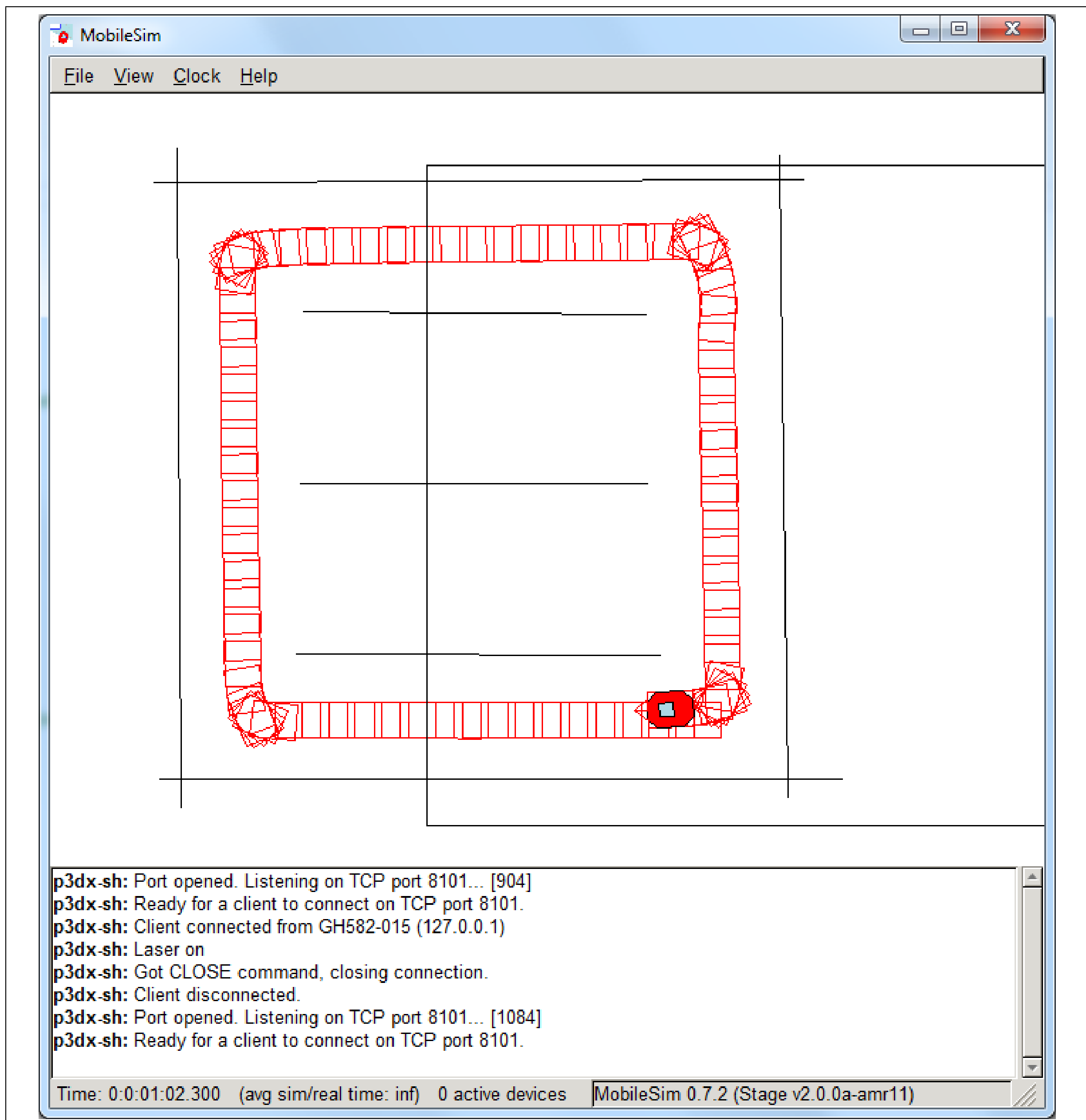

Figure 3: The third test map

5.2 Test Data

The effectiveness of the implementation was put to the test with the use of the constructed test maps by measuring and recording its raw test data. This was done by taking readings of the robot during each test map over a span of ten minutes, recording how close the robot was to the setpoint (0.5m) on a number of different edges that it followed. These statistics are shown below.

5.2.1 Root-mean-square error

The root-mean-square error (RMSE) is a commonly used measuring method that provides a measurement of the difference between a predicted and actual value. The calculation then provides the sample standard deviation of the differences between predicted and actual values for a set of statistics.

Applying this to our project, we can observe the actual distance that the robot was following a wall in comparison with the defined setpoint of 0.5m. In each test map the robot follows a large number of walls, and for each wall an average distance was taken ($\hat{x}_i$). An average distance was recorded for ten walls in each map. These distances can then be used in the RMSE formula to provide an indication on how accurately the robot followed the setpoint. The formula for the RMSE is shown below.

$$\sqrt{\frac{\sum_{i=1}^{n}(\hat{x}_i - setpoint)^2}{n}}$$

5.2.2 Map 1

Test №	1	2	3	4	5	6	7	8	9	10
Distance (mm)	512	490	496	488	480	521	495	508	515	502

Root-mean-square error calculation

$$\sqrt{\frac{\begin{array}{c}(512-500)^2+(490-500)^2+(496-500)^2+(488-500)^2+(480-500)^2+\\(521-500)^2+(495-500)^2+(508-500)^2+(515-500)^2+(502-500)^2\end{array}}{10}}$$

$$=\sqrt{\frac{\begin{array}{c}(12)^2+(-10)^2+(-4)^2+(-12)^2+(-20)^2+\\(21)^2+(-5)^2+(8)^2+(15)^2+(2)^2\end{array}}{10}}$$

$$=\sqrt{\frac{\begin{array}{c}144+100+16+144+400+\\441+25+64+225+4\end{array}}{10}}$$

$$=\sqrt{\frac{1563}{10}}$$

$$=\sqrt{156.3}$$

$$=12.5$$

5.2.3 Map 2

Test №	1	2	3	4	5	6	7	8	9	10
Distance (mm)	502	505	515	498	503	491	507	489	501	494

Root-mean-square error calculation

$$\sqrt{\frac{\begin{array}{c}(502-500)^2+(505-500)^2+(515-500)^2+(498-500)^2+(503-500)^2+\\(491-500)^2+(507-500)^2+(489-500)^2+(501-500)^2+(494-500)^2\end{array}}{10}}$$

$$=\sqrt{\frac{\begin{array}{r}(2)^2+(5)^2+(15)^2+(-2)^2+(3)^2+\\(-9)^2+(7)^2+(-11)^2+(1)^2+(-6)^2\end{array}}{10}}$$

$$=\sqrt{\frac{\begin{array}{r}2+25+225+4+9+\\81+49+121+1+36\end{array}}{10}}$$

$$=\sqrt{\frac{553}{10}}$$

$$=\sqrt{55.3}$$

$$=7.44$$

5.2.4 Map 3

Test №	1	2	3	4	5	6	7	8	9	10
Distance (mm)	502	490	495	505	509	518	487	504	507	491

Root-mean-square error calculation

$$\sqrt{\frac{\begin{array}{c}(502-500)^2+(490-500)^2+(495-500)^2+(505-500)^2+(509-500)^2+\\(518-500)^2+(487-500)^2+(504-500)^2+(507-500)^2+(491-500)^2\end{array}}{10}}$$

$$=\sqrt{\frac{\begin{array}{c}(2)^2+(-10)^2+(-5)^2+(5)^2+(9)^2+\\(18)^2+(-13)^2+(4)^2+(7)^2+(-9)^2\end{array}}{10}}$$

$$=\sqrt{\frac{\begin{array}{c}4+100+25+25+81+\\324+169+16+49+81\end{array}}{10}}$$

$$=\sqrt{\frac{874}{10}}$$

$$=\sqrt{87.4}$$

$$=9.35$$

5.3 Observations

Looking at the calculated RMSE values, we can see a correlation that applies well to the demonstrated traversal maps in Figures 1, 2, and 3.

The first map provided the highest RMSE of 12.5. We can observe that the reason for this being higher than the others is related to the square in the top left corner of the map that the robot has to avoid, thus adding a small part of the map that the robot struggles to follow correctly. Comparing this to the second map, in which we see the lowest RMSE of 7.44, we can see how the robot cleanly traverses around the map in a rectangular shape, having no issues navigating around the obstacle in the centre of the map. The final map provides a RMSE that is almost perfectly in between the previous two maps, at 9.35. Analysing the traversal map in Figure 3 we can see that the slightly angled lines in the centre and on the edges of the map cause variations in the robots traversal, occurring mostly in the top right corner.

6 Conclusion

Overall the implementation of the robotic controller performs well and adheres to the given specification in an appropriate manner. The FSM provides a clear and simple architecture that generates a good trade off between investment in development time and overall system performance. As discussed in the literature review, given more development time or if collaborating with other developers, a motor schema could have been more beneficial to the overall scalability and performance of the system, however the FSM performed acceptably.

Further improvements other than the use of a motor schema would include incorporating a level of reactive control, allowing the controller to respond to dynamic events. This would again add another level of complexity to the system but provide the controller with the ability to react to dynamic changes in the map, a factor that the deliberative control method reacts to poorly. However, as there was no requirement for dynamic response within this specification, deliberative control still provided the required functionality for the robot to operate successfully.

The implementation of the PID algorithm proved to be very effective within the project. Further tweaking to the individual P, I, & D values may provide better results in the long run, however the testing phase showed that, on the provided maps, the current values apply themselves well and allow the robot to successfully follow an edge with minimal traversal issues.

A Source File

```
#include <iostream>
#include <stdlib.h>
#include <math.h>
#include <Aria.h>

#include "wanderAndAvoid.h"

// PID variables
double dState; // Last position input
double iState; // Integrator state
double iMax, iMin;

// Maximum and minimum allowable integrator state
double iGain, // integral gain
               pGain, // proportional gain
               dGain; // derivative gain

// Calculates a random integer between min and max inclusive
int randomBetween(int min, int max) {
       return rand() % (max - min + 1) + min;
}

// Pythagoras theorem for distance calcuation for random turning
double distanceBetween(double x1, double y1, double x2, double y2) {
       return sqrt((x2 - x1) * (x2 - x1) + (y2-y1) * (y2-y1));
}

// Updates the PID values
double updatePID(double error, double position)
{
       double pTerm,
       dTerm, iTerm;
       pTerm = pGain * error;
       // calculate the proportional term

       // calculate the integral state with appropriate limiting
       iState += error;
       if (iState > iMax) {
              iState = iMax;
       } else if (iState < iMin) {
              iState = iMin;
       }

       iTerm = iGain * iState; // calculate the integral term
       dTerm = dGain * (position - dState);
       dState = position;

       return pTerm + iTerm - dTerm;
```

```
49 }
50
51 // Constructor
52 FSM::FSM() : ArAction("FSM")
53 {
54         currentState = WANDER;
55
56         // Zero the starting wander coordinates
57         startWanderX = 0;
58         startWanderY = 0;
59         wanderDistance = randomBetween(500, 1500);
60
61         speed = FULL_SPEED; // 0.5m/s
62         deltaHeading = 0; // 0 degrees
63
64         // Set PID values
65         pGain = 0.02;
66         iGain = 0;
67         dGain = 2;
68 }
69
70 boolean obstacleLeft = false;
71
72 // Body of action
73 ArActionDesired * FSM::fire(ArActionDesired d)
74 {
75         // Read sonar readings
76         leftSonar = myRobot->getClosestSonarRange(-20, 100);
77         rightSonar = myRobot->getClosestSonarRange(-100, 20);
78         frontSonar = myRobot->getClosestSonarRange(-20, 20);
79
80         // The current distance we have travelled this cycle from when we
               started wandering
81         double currentDistance = distanceBetween(startWanderX, startWanderY,
               myRobot->getX(), myRobot->getY());
82
83         switch (currentState) {
84         case WANDER:
85                 printf("Wandering \n");
86                 speed = FULL_SPEED;
87
88                 // Set us back to moving straight
89                 deltaHeading = 0;
90
91                 if (frontSonar < (OBSTACLE_DISTANCE + 1000)) { // +1000 ensures
                       we have enough time to slow down before hitting the obstacle
92                         currentState = OBSTACLE_INFRONT;
93                         obstacleLeft = (leftSonar < rightSonar);
94                 } else if (leftSonar < FIND_WALL_DISTANCE) {
95                         currentState = FOLLOW_WALL_LEFT;
96                 } else if (rightSonar < FIND_WALL_DISTANCE) {
```

```
97                      currentState = FOLLOW_WALL_RIGHT;
98              }
99
100             if (currentState == WANDER) { // If no change was made to the
                    state then we can check if we've wandered too far
101                     if (currentDistance > wanderDistance) { // Random distance
                             betwen 0.5m and 1.5m
102                             currentState = TURN_RANDOMLY;
103                             startedRandom.setToNow();
104                     }
105             }
106             break;
107
108     case OBSTACLE_INFRONT:
109             printf("Obstacle infront \n");
110             speed = 0;
111
112             deltaHeading = obstacleLeft ? -30 : 30; // Turn right if there's
                    an obstacle to our left, otherwise turn right
113
114             if (frontSonar >= OBSTACLE_DISTANCE + 1000) { // Object no longer
                     in the near distance, return to wandering
115                     currentState = WANDER;
116                     startWanderX = myRobot->getX();
117                     startWanderY = myRobot->getY();
118             }
119             break;
120
121     case TURN_RANDOMLY:
122             printf("Turning randomly \n");
123             speed = 0;
124
125             // Move at a random angle between 20-160
126             deltaHeading = randomBetween(20, 160);
127
128             if (randomBetween(0, 1) == 0) {
129                     deltaHeading = -deltaHeading;
130             }
131
132             if (startedRandom.mSecSince() > 2000) {
133                     currentState = WANDER;
134                     startWanderX = myRobot->getX();
135                     startWanderY = myRobot->getY();
136                     wanderDistance = randomBetween(500, 1500);
137             }
138             break;
139
140     case FOLLOW_WALL_LEFT:
141             printf("Following wall left \n");
142             speed = FULL_SPEED;
143
```

```
144             // Find error
145             if (leftSonar < (2 * FOLLOW_WALL_DISTANCE)) { // Left < 1m
146                 error = (FOLLOW_WALL_DISTANCE - leftSonar);
147                 deltaHeading = -updatePID(error, leftSonar);
148             } else {
149                 // No error, return to wandering
150                 currentState = WANDER;
151                 startWanderX = myRobot->getX();
152                 startWanderY = myRobot->getY();
153                 wanderDistance = randomBetween(500, 1500);
154             }
155 
156             // Ensure we are still checking front sonar even when following
                    the wall, incase anything is in front of us at the end of the
                    wall
157             if (frontSonar < (OBSTACLE_DISTANCE + 1000)) {
158                 currentState = OBSTACLE_INFRONT;
159                 speed = 0;
160                 deltaHeading = 0;
161                 obstacleLeft = (leftSonar < rightSonar);
162             }
163 
164             break;
165 
166         case FOLLOW_WALL_RIGHT:
167             printf("Following wall right \n");
168             speed = FULL_SPEED;
169 
170             // Find error
171             if (rightSonar < (2 * FOLLOW_WALL_DISTANCE)) { // Right < 1m
172                 error = -(rightSonar - FOLLOW_WALL_DISTANCE);
173                 deltaHeading = updatePID(error, rightSonar);
174             } else {
175                 // No error, return to wandering
176                 currentState = WANDER;
177                 startWanderX = myRobot->getX();
178                 startWanderY = myRobot->getY();
179                 wanderDistance = randomBetween(500, 1500);
180             }
181 
182             // Ensure we are still checking front sonar even when following
                    the wall, incase anything is in front of us at the end of the
                    wall
183             if (frontSonar < (OBSTACLE_DISTANCE + 1000)) {
184                 currentState = OBSTACLE_INFRONT;
185                 speed = 0;
186                 deltaHeading = 0;
187                 obstacleLeft = (leftSonar < rightSonar);
188             }
189             break;
190         }
```

```
191
192         desiredState.reset(); // reset the desired state (must be done)
193         desiredState.setVel(speed); // set the speed of the robot in the desired
                state
194         desiredState.setDeltaHeading(deltaHeading); // set the new angle
195
196         return &desiredState; // give the desired state to the robot for
              actioning
197 }
```

B Header File

```
// Signatures

class FSM: public ArAction // Class action inherits from ArAction
{
public:
	FSM(); // Constructor
	virtual ~FSM() {} // Destructor
	virtual ArActionDesired * fire(ArActionDesired d); // Body of the action
	ArActionDesired desiredState; // Holds state of the robot that we wish to action

	int speed; // Speed of the robot in mm/s
	double deltaHeading; // Change in heading

	enum state {
		WANDER,
		FOLLOW_WALL_LEFT,
		FOLLOW_WALL_RIGHT,
		OBSTACLE_INFRONT,
		TURN_RANDOMLY
	};

	// Reading
	double leftSonar;
	double rightSonar;
	double frontSonar;

	// FSM state
	state currentState;

	// Starting wander coordinates
	double startWanderX, startWanderY;
	double wanderDistance;

	// Control variables
	double error; // Current error
	double output; // Final output signal

	ArTime startedRandom;
	int randomWaitDistance;

	// speeds
	#define FULL_SPEED 500 // 0.5m/s

	// distance thresholds
	#define OBSTACLE_DISTANCE 100 // will not come into 0.1m of obstacles
	#define FIND_WALL_DISTANCE 1000 // follow any wall edges detected within 1.0m
	#define FOLLOW_WALL_DISTANCE 500 // at a constant distance of 0.5m
};
```

References

Willis, MJ (1999). "Proportional-Integral-Derivative Control". In: *Dept. of Chemical and Process Engineering University of Newcastle.*

Li, Wei (1998). "Design of a hybrid fuzzy logic proportional plus conventional integral-derivative controller". In: *Fuzzy Systems, IEEE Transactions on* 6.4, pp. 449–463.

List of Figures

1 The first test map 6

2 The second test map 7

3 The third test map 8

Acronyms

FSM finite state machine. i, 3, 5, 13

PID proportional-integral-derivative. 1, 4, 5, 13

RMSE root-mean-square error. i, 9–13