IMAT3404 – Mobile Robotics
Implementing a Robot Controller for the PeopleBot

Christian Manning – p0928544x
De Montfort University

February 2012

Contents

1 Introduction **2**

2 Architectural Design **2**

3 Behavioural Design **3**
3.1 Obstacle Avoidance 3
3.2 Edge Following 4
3.2.1 PID 4
3.2.2 Tuning 5
3.3 Wandering 6
3.4 Mapping 7

4 Experimental Design **8**
4.1 Obstacle Avoidance 8
4.2 Edge Following 9
4.3 Wandering 9
4.4 Mapping 9
4.5 Architecture 10

5 Results **10**
5.1 Obstacle Avoidance 10
5.2 Edge Following 10
5.3 Wandering 11
5.4 Mapping 11
5.5 Architecture 11

6 Conclusion **12**

1 Introduction

The aim of this project is to design and implement a controller for the PeopleBot robot, using C++ with ARIA. The PeopleBot is a robot based around a Pioneer 3-DX design intended for human-interaction projects. It uses sonar and lasers as environmental sensors. This project entails the use of the sonar sensors and will utilise ARIA software library for the control of the robot, MobileSim for experimentation and testing, and Mapper for designing maps. This document details design decisions for the architecture, behaviour and experiments, the reason such decisions were made, and the the outcome of the project.

2 Architectural Design

The architecture of a robot controller concerns the coordination of a multitude of behaviours and the summation of these different behaviours. This is an important aspect of controller design for behaviour-based robots due to the inevitable conflicts of several behaviours and an architecture can work to resolve those conflicts.

For this project, the architecture will be based around the subsumption model. A subsumption architecture consists of many behaviours (or *tasks*) which are broken down into simple units (e.g. move forward, avoid obstacle, etc.). These simple units can be modelled in a linear hierarchy, their relative position outlining the priority of each unit. The priority of a unit denotes whether it can suppress lower priority units, or can be suppressed by those with a higher priority.

The subsumption architecture is not concerned with what exactly the tasks aim to accomplish, only the importance of each of them. This system of suppression allows important tasks to be completed without being impeded by lesser tasks, while still letting lesser tasks execute when more important ones are idle.

In each of the units, there is a finite state machine which determines what exactly the task should be doing to accomplish its goal, or even whether it's doing anything at all. What this means is that there is some condition that must be met for a task to be attempted, and if more than one task meets their condition then the one with the highest priority is the one which runs.

This has been implemented using switch statement (see figure 1) in each behaviour. Each behaviour maintains a *state* variable which represents a certain state that a behaviour can be, as an integer. An example, that will be explained in more detail later in this document, is an obstacle avoidance behaviour which will have three states: *idle*, *turn left*, and *turn right*. In the *idle* state, there will be code which determines if an obstacle is close and whether it is detected to the right or to the left of the robot. If there is an obstacle, the state is updated to *turn left* or *turn right* if the obstacle is to the right or left of the robot, respectively.

```
switch(state) {
case IDLE:
    //detect obstacles
case TURN_LEFT:
    //turn left
case TURN_RIGHT:
    //turn right
}
```

Figure 1: A C++ switch statement

These state machines are used to define a *desired state* to be passed to ARIA. ARIA determines whether this *desired state* is unchanged, and if so allows a lower priority behaviour to take the fore.

3 Behavioural Design

There are several behaviours required to be implemented for the robot controller in this project. They are designed to conform to the above specified architecture, the details of which are below. These behaviours are implemented using feedback control methods which determines its output by taking in relevant sensor readings and adjusting accordingly which means that the environment affects the robots behaviours.

All the listed behaviours are implemented as classes inheriting the *ArAction* class from ARIA. A single instance of each of these classes is instantiated in ARIA's main thread.

3.1 Obstacle Avoidance

The obstacle avoidance behaviour ensures that the robot does not come within $0.1m$ ($100mm$) of an obstacle. This is accomplished by using a binary feedback control system. Binary control creates a system of two states: *on* or *off*. In the case of this behaviour, the robot is either turning or not.

The implementation of this has a $250mm$ proximity, or, the distance to an obstacle at which the robot should start turning. This value for the proximity along with a heading of $\pm 135°$ allows obstacles to be totally avoided and the robot will not approach within $100mm$ of an obstacle. The $\pm 135°$ heading was chosen because it ensures that the robot gets clear of an obstacle. The robot will not actually turn $\pm 135°$ as this cannot be done in the time it takes for the desired behaviour to be overridden, but when called many times successively the robot will turn very sharply, potentially until the heading of $\pm 135°$ is realised.

Figure 2 shows a sharp turn caused by heading towards an obstacle at a $\pm 90°$ angle. In this situation the robot will turn left, an arbitrary decision, likely resulting in turning the full $\pm 135°$ heading. A $\pm 180°$ heading was not chosen so as to let the robot's movements be less monotonous. When turning, the speed of the robot is also reduced to $25mm/s$ to facilitate a tighter turning circle.

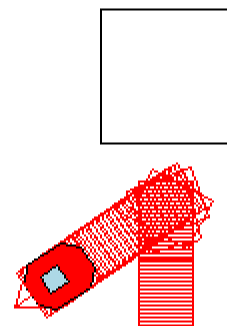


Figure 2: Sharp turn

This behaviour is implemented in the *Avoid* class, whose instance is given the highest priority of those implemented with a value of 70 in ARIA. The reason for this is that the robot *cannot* come within $100mm$ of an obstacle and tasks such as *wander* are likely to come across obstacles, so it is paramount that this task is able to suppress all others concerned with movement.

3.2 Edge Following

The next behaviour is one which aims to follow any edges that are detected within $1.0m$ ($1000mm$) at a distance of $0.5m$ ($500mm$). This requires a much more sophisticated method of control than a simple binary feedback control as used above, namely a proportional-integral-derivative (PID) controller, its constituent formulae shown in figure 3, where e is error and t is time.

3.2.1 PID

A PID controller defines a set point, in this case the desired distance to an edge, and three gain (K) values: proportional (K_p), integral (K_i) and derivative (K_d). These gains are used to calculate the overall output from the error; the error being how far the robot is away from the set point, the output being a heading.

$$P = K_p e_t$$

$$I = K_i \int_0^t e_\tau d\tau$$

$$D = K_d \frac{d}{dt} e_t$$

$$O = P + I + D$$

Figure 3: Formulae for Proportional, Integral, Derivative and Overall output values

The proportional value is used to diminish any error in the present by multiplying K_p by the error at a specific time to produce an output value of some *proportion* to the error, accounting for the *present* error.

The integral value is the sum of the previous n errors multiplied by K_i. K_i is typically a very small value. The integral accounts for *past* errors, bringing the output closer to the set point at a faster rate.

The derivative value attempts to eliminate *future* errors by taking the current rate of change and multiplying by K_d. This purpose of this is to balance out the proportional and integral values by reducing the rate of change and therefore reducing the overall oscillation, keeping the robot from veering too far from the set point.

$$I = K_i \sum_{i=(t-n)}^{n} e_i$$

$$D = K_d(e_t - e_{t-1})$$

Figure 4: Formulae as implemented.

Implementing these formulae directly could be problematic and also quite expensive computationally. Instead, some approximations are used, see figure 4. These offer a good compromise between accuracy and overhead. For the integral output, a ring buffer with 50 elements is used to store the last 50 errors, the sum of which is then multiplied by K_i. The previous error (e_{t-1}) can be accessed via the *last()* method of the custom ring buffer implementation, allowing the derivative output to be calculated with no extra state. The proportional output can be calculated as originally defined as it does not use any operations other than multiplication.

3.2.2 Tuning

Unfortunately there is no general case in which to determine the gain values as they are context sensitive. To find ideal values for K_p, K_i and K_d, the controller needs to be tuned. Oscillation in this domain relates to how far to output moves the robot past the set point. The resulting gains should limit oscillation while still acting appropriately to changes.

Tuning begins by setting K_p to a low value, 0.01 was used for this project. The first variable to tune is K_d which is initially set to $100K_p$, in this case 1. K_d is then raised until there is reasonably large oscillation about the set point, and then it is lowered by a factor of 4. In this instance the oscillations were large with $K_d = 5$, resulting in a K_d of 1.3 after some rounding to 1 d.p. This value for K_d gives a smooth output and is taken as the final value.

K_p is to be tuned next. With $K_p = 0.01$ the system is not showing oscillation, which means that it should be raised by a factor of 10 until it does so. It begins oscillating with the first increase to 10, and should therefore be reduced by a factor of 4, ending with $K_p = 0.03$, again rounded to 1 d.p.

To tune K_i, it initially needs to be set to a very low value, so $K_i = 0.00001$. Similarly to K_p it is raised by a factor of 10 until the system oscillates, which was immediately when it was raised to 0.0001. At this point the value should be lowered by a factor of 2 to give $K_i = 0.00005$. This gain value works very well and didn't need any fine tuning.

The end result of this tuning can be seen in figure 5. This shows that it can work not only on the outside of a shape, but also the inside. It can also be seen that it take the corners somewhat wide. This is because the sensors used to detect an edge range from $-110°$ to $110°$, rather than $-90°$ to $90°$, due to the sensor readings not taking into account the sensors at $\pm 90°$, so the robot goes slightly past the corner before turning due to the sensor being further back. This led to the robot being past its set point at all times and so would not follow the specified limit of $500mm$, though it did take the corners much tighter.

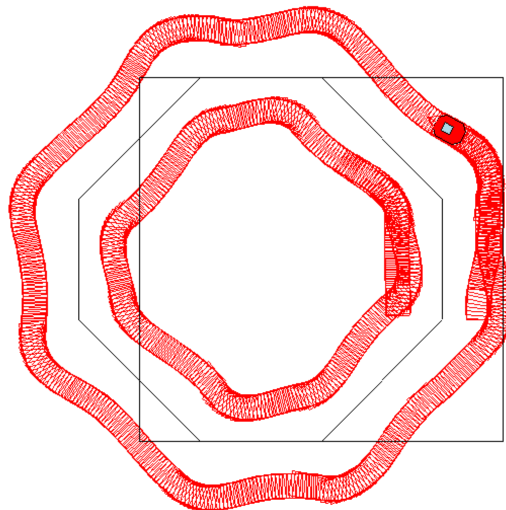


Figure 5: Edge-following an octagon.

The implementation of this behaviour in the *Follow* class is carried out in a similar fashion to *avoid obstacles*, so that it can be used in the subsumption architecture. This consists of a state machine with two states: *idle* and *follow*, the latter contains the PID implementation. This behaviour is given a priority of 60 in ARIA, allowing it to be suppressed so that obstacles can be avoided should the robot get too close.

3.3 Wandering

With the currently defined behaviours the robot would be immobile in an empty space, due to the requirement of proximity of the current behaviours. Therefore, a behaviour allowing the robot to wander will be defined. It should move at a speed of $50mm/s$ for a randomly chosen distance in the range of $0.5m$ and $1.5m$ and then turn in a random direction by a random angle between $20°$ and $160°$.

Computers are not very good at generating random numbers without complex algorithms or additional hardware, neither of which being suitable for this project. A compromise is to use a pseudo-random number generator such as the *rand()* function from the C standard library (*stdlib.h*). To get a number in a specific range it is necessary to use the % or *modulo* C/C++ operator. This binary operator operator returns the remainder of the division of two integers. The result (x) of an expression $a\%b$ is guaranteed to satisfy the condition $0 \leq x \leq b$. Using a combination of the *rand()* function, the % operator and simple addition, the *angle* to be turned and *distance* to be travelled is calculated as shown in figure 6.

```
//random num between 500 & 1500
distance = (rand() % 1000) + 500;
//random num between 20 & 160
angle = (rand() % 140) + 20;
```

Figure 6: Generating pseudo-random numbers in a range in C++.

The direction of the turn is derived from the parity of the generated angle, i.e. whether it is odd or even. If the angle is odd the robot will turn right and left if it is even. This approach was chosen to reduce the number of calls to the *rand()* function, as it was deemed unnecessary in this situation due to the angle itself being randomly generated.

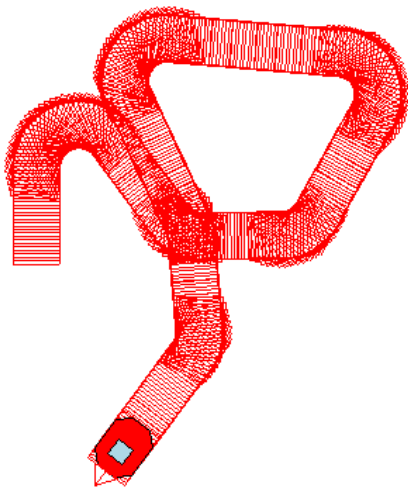


Figure 7: Wandering around.

The distance the robot has travelled at its current iteration begins when a turn is completed. A turn is said to be complete when the angle the robot is facing is the summation of the angle it was facing when it began turning and the desired heading. This is implemented in the *bool turnComplete()* method of the *Wander* class.

The instance of this behaviour has been given the priority of 50. The reason for this is that it is necessary for the *follow* and *avoid* behaviours to inhibit *wandering* or else it will interfere with their operations. For example, the robot could be following a wall and then *wander* decides that it is time to turn in to the wall being followed causing it to crash.

3.4 Mapping

Mapping is the process of creating a map of the robot's surroundings relative to the position of itself. The robot first needs to know its own position relative to where it started, so that it knows how far it has moved and in which direction. The process used to calculate this is called *dead reckoning*.

Dead reckoning uses the last known position along with some movement to calculate the current position. The formula $p_t = p_{t-1} + movement$ can be used to generalise this process, where p is position and t is time. To measure movements the PeopleBot uses optical encoders to determine the speed of each wheel. These encoders record a number of *ticks*, the difference between ticks being a known amount, and the time taken for these ticks. From these readings it can be calculated how far the robot has moved, and also how much it has turned, using some formulae.

Figure 8 shows the formulae for calculating the differences in x, y and θ using readings from the robot's encoders and some known values. T is the number of ticks reported by each encoder (T_1 and T_2 for each wheel) and T_R is the number of ticks needed for a full rotation. r_w is the radius of each wheel and L is the distance between the two wheels. The new position is calculated by adding these differences to the previous position, i.e. $(x', y', \theta') = (x + \Delta x,\ y + \Delta y,\ \theta + \Delta\theta)$.

$$\Delta\theta = 2\pi(T_1 - T_2)\frac{r_w}{T_R L}$$

$$\Delta x = \cos(\theta)(T_1 + T_2)\frac{\pi r_w}{T_R}$$

$$\Delta y = \sin(\theta)(T_1 + T_2)\frac{\pi r_w}{T_R}$$

Figure 8: Dead reckoning formulae

Now that the robot knows where it is, it can begin mapping the environment around it using the sonar sensors. The sonar sensors report two pieces of information to Aria when an obstacle is detected: the distance and the angle to the obstacle (D and θ_s, respectively). From these two things, the coordinates of the point of detection for the robot's current coordinate system (x_s, y_s) can be calculated. This point now needs to be transformed to the global coordinate system, first by rotating by the robot's heading (θ_r), then translate by the robot's position (x_r, y_r) to give the the final map point (x, y). Figure 9 shows the formulae for calculating the map point, R is the robot's radius.

$$x_s = \cos(\theta_s)(D + R)$$

$$y_s = \sin(\theta_s)(D + R)$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x_s \\ y_s \end{bmatrix} \begin{bmatrix} \cos(\theta_r) & -\sin(\theta_r) \\ \sin(\theta_r) & \cos(\theta_r) \end{bmatrix}$$

$$(x, y) = (x' + x_r,\ y' + y_r)$$

Figure 9: Mapping formulae

This process is completed every other *tick* of the robot's clock. The sonar sensors used to detect edges are grouped into ranges of 30° and iterated through, adding to five readings from distinct sensors. Once there are five points collected, they are transmitted over a network connection via TCP/IP to a client application. The client application has been written using the Qt framework to allow real-time visualisation of the map being created. The application allows zooming (on the (invisible) axes) with the mouse wheel and movement of the axes with

the arrow keys. Upon executing the application it is necessary to connect to the server (the robot controller) by pressing the ”C” key, and can be disconnected from the server by pressing the ”D” key. Pressing the ”S” key will save a *png* image file of the current visualisation area to the working directory named ”map.png”. It was chosen to buffer five points then send them all at once to save on small network transfers, though this makes the visualisation slightly less real-time.

In the *Map* class where this is all implemented, the use of *ArSocket::accept* caused the whole application to stall until a client connected, making the robot stand still and eventually MobileSim disconnected. This problem was solved by having *ArSocket::accept* run in a separate thread.

This behaviour is given a priority of 40 in ARIA though this is irrelevant in this scenario as it doesn't need to ever change the *desiredState*; it is a purely passive behaviour and will be run regardless.

4 Experimental Design

The above specified behaviours are to be tested using several purpose built maps which will test the effectiveness of each behaviour individually, as well as combined to test the subsumption architecture.

4.1 Obstacle Avoidance

Testing the obstacle avoidance behaviour involves a simple map with a small square obstacle in a rectangular enclosure, see figure 10. Three tests will be completed which utilise this map:

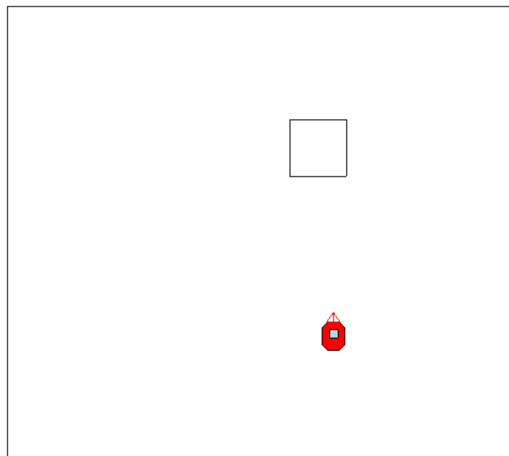
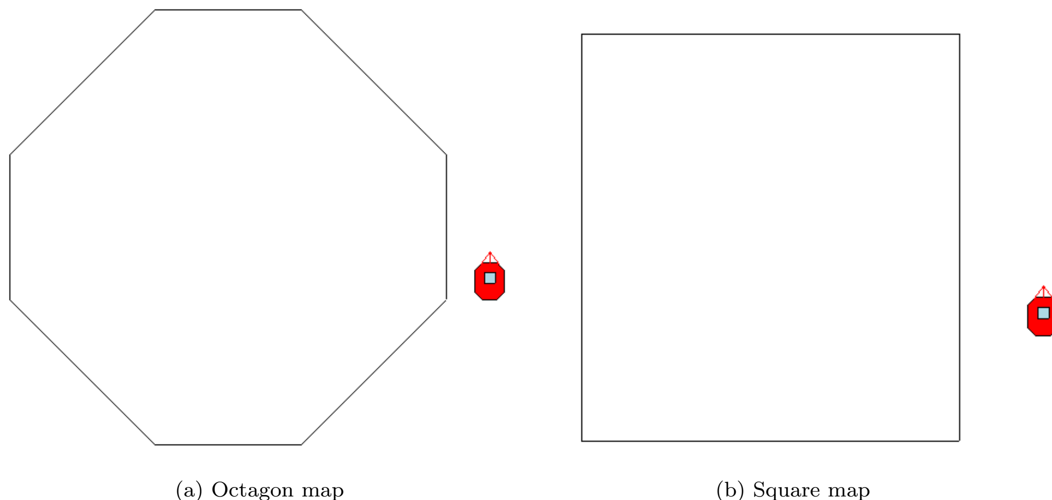


Figure 10: Avoidance test map.

1. Move towards an obstacle head-on. This will test whether the behaviour works in this situation as it was given no special treatment in the implementation.
2. Move towards an obstacle at a shallow angle. This is a typical scenario for avoiding obstacles
3. Move towards a sharp corner. This situation may arise when the *follow* behaviour gets too close to an edge and connot turn away in time.

4.2 Edge Following

To test edge following two maps have been created: an octagon (figure 11a) and a square (figure 11b). Having the robot follow the outside of the octagon will demonstrate how smoothly it can follow around shallow corners and how much oscillation it will show. This map is also useful for showing how well it will follow the internal corner of the inside of the shape. The square map



(a) Octagon map (b) Square map

Figure 11: Edge following test maps.

when followed from the outside should show that some corners are too sharp to follow and it will carry on straight forward or veer away. Allowing the robot to follow the inside of the square will show how well the PID controller holds up against some tight corners.

4.3 Wandering

The nature of the *wandering* behaviour limits the ability to test due to its random nature, though figure 7 demonstrates its abilities above.

4.4 Mapping

The *mapping* behaviour is tested by connecting the client to the robot controller which will then automatically begin transmitting coordinates to the client. The octagon map (figure 11a) will

be the one to be mapped.

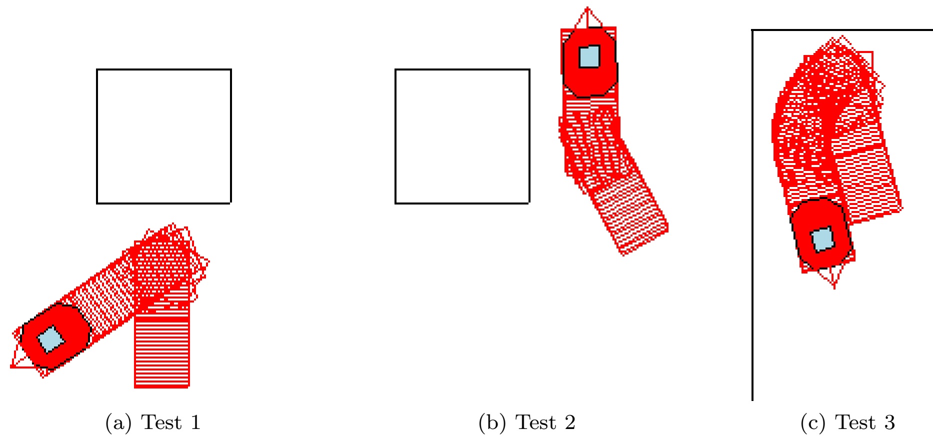
4.5 Architecture

For testing the overall architecture of the behaviours implemented for this project, the map from the *avoidance* tests (figure 10) will be used with a different starting point and the robot will be let loose to demonstrate all the behaviours, while also mapping at the same time.

5 Results

5.1 Obstacle Avoidance

Figure 12 shows the results of the tests performed on the obstacle avoidance behaviour. Figure 12a shows a head-on obstacle being avoided by sharply turning left to the full heading of 135°. This is the expected result of this test and shows the performance of the behaviour in this circumstance is satisfactory. Figure 12b shows the avoidance of an obstacle at a shallow angle and also performs as expected. Figure 12c demonstrates the effectiveness of this behaviour when the robot is sent in to a corner. It shows how it will sharply turn away from the first edge, then gradually turn away from the second edge. This is a practical solution for a tricky situation, despite not even being treated as a special case.



(a) Test 1 (b) Test 2 (c) Test 3

Figure 12: Obstacle Avoidance test runs.

5.2 Edge Following

The result of the edge following of an octagon can be seen above in figure 5, both inside and out. The performance of this test is discussed above. The outside test on the square map (figure 13) shows some unexpected behaviour when the first corner is met. This corner should be too steep for the PID controller to handle as it should reach more than $1000mm$ away before it has turned

enough to see it. The second corner is handled as expected: it begins gently turning around the corner then veers off due to moving beyond $1000mm$ of the edge. Following the inside of the square functions as intended, though it does get quite close the the edges during the corners. Due to this it can be seen some sharp turns being made at the apex. This is the *avoid* behaviour kicking in and is the first demonstration of the subsumption architecture at work.

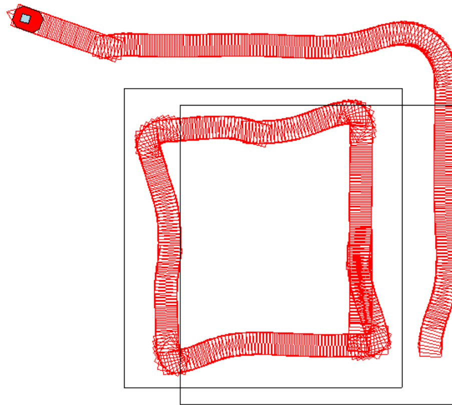


Figure 13: Edge following a square.

5.3 Wandering

As explained in the previous section figure 7 demonstrates the results of testing the *wandering* behaviour and shows the expected outcome.

5.4 Mapping

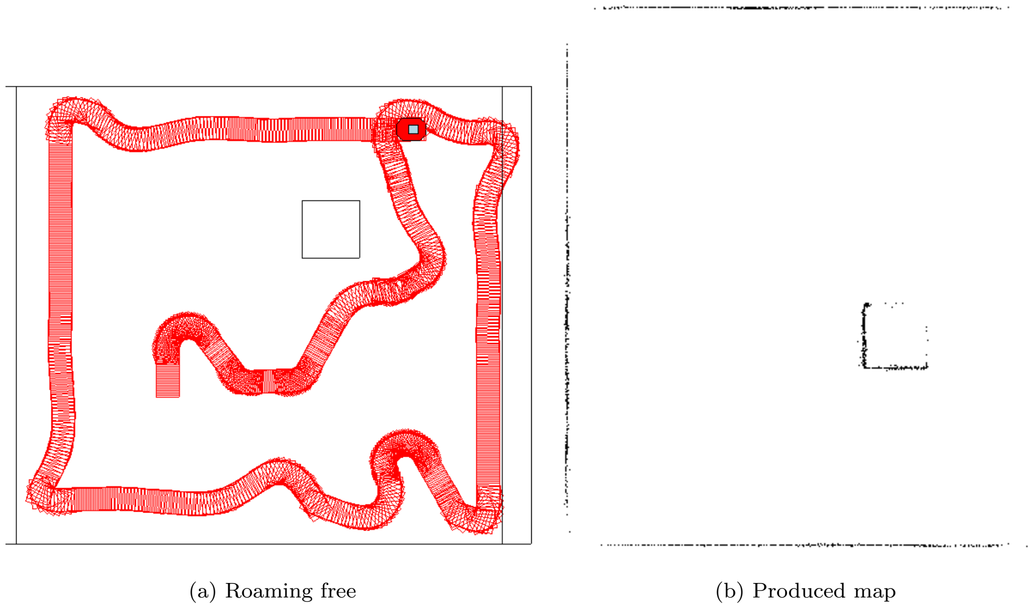
Figure 14 shows the output produced by the mapping client. It shows the points transmitted by the robot's controller transformed to a different coordinate system so that the map produced is understandable to humans. It is also rotated by 90° which is due to the robot having a starting heading of 90°, though it always begins on an internally set 0°. The resulting image not only shows the outline of the map correctly, but the points are close enough together that edges can easily be deduced.

Figure 14: Mapping an octagon.

5.5 Architecture

The result of the subsumption test can be seen in figure 15.
It shows all of the behaviours working: It wanders around until it finds the obstacle; It tries to follow the edge of the obstacle, then resumes wandering; It comes across the wall and begins

following; In the bottom-right corner it gets a little too close to the wall and the avoids it by turning away sharply; At this point it has gotten far enough from the wall that it resumes wandering; It then continues following the wall around two more corners. The map produced by the client is shown in figure 15b (rotated 90°). This is a fairly complete map, though some areas such as the top and left side of the obstacle and the corners are incomplete. The corners are hard to map as the sensors are unlikely to pick them up. The obstacle wasn't fully detected as the robot doesn't get very close to two of the sides.



(a) Roaming free

(b) Produced map

Figure 15: Subsumption test.

6 Conclusion

This project has proven to be a success from the results of the testing. It has provided simple functionality, including obstacle avoidance and wandering, and the advanced functionality of edge following and mapping with real-time visualisation. An extra feature that could have been added is an implementation of the RANdom SAmple Consensus (RANSAC) algorithm. This algorithm would enable the mapping to determine where the edges are, and so can replace the dots with lines. The addition of a RANSAC implementation would have made this project complete. Despite the lack of this feature, this robotics controller fulfils its purpose of implementing multiple behaviours using appropriate control methods and an appropriate architecture.