Exploration and Communication Using Mobile Robots

Final Report for CSM6960 Major Project

Author: Stefan Klaus (stk4@aber.ac.uk)
Supervisor: Dr. Myra Wilson (mxw@aber.ac.uk)

24th September 2015
Version: 1.0 (Release)

This report was submitted as partial fulfilment of a MSc degree in Intelligent Systems (G496)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Declaration of originality

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signature ....

Date ....

Statement

This work is the result of my own investigations, except where otherwise stated. Where ***correction services** have been used, the extent and nature of the correction is marked in a footnote(s). Other sources are acknowledged(e.g. by footnotes and explicit references).
A bibliography is appended.

Signature ....

Date ....

[*this refers to the extent to which the text has been corrected by others]

Consent to share this work

I hereby give consent for my work, if accepted, to be available for photocopying and for inter-library loan, and for the the title and summary to be made available to outside organisations.

Signature .... (candidate)

Date ....

Acknowledgements

I am grateful to...

I’d like to thank...

Abstract

Using a swarm of mobile robots to explore and environment while maintaining communications between all robots is an interesting challenge. In this paper a solution for this challenge utilising a Artificial Neural Network and a swarm of E-Puck robots is proposed. The task of the swarm is to map a simple environment while maintaining communication range to each other. The created solution and its results are shown, critically analysed, and suggestions for future improvements are made.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Listings

Chapter 1

Introduction & Background

1.1 Introduction

Autonomous robots must be able to create and maintain a model of their environment. Doing so allows autonomous robots to be used in a wide variety of tasks, ranging from cartography of buildings; the exploration of hazardous environments, such as encountered in many search-and-rescue operations after natural disasters; to the autonomous exploration of other planets like Mars, where even the most minimal human input needs to be carefully planned and executed using a small predefined time window.

Scientists and engineers face a couple of problems when trying to tackle these tasks; how to create an autonomous robot that can react to a wide variety of different environments, and how to ensure that they are able communicate with other robots and handle the exploration itself. One of the key approaches to these tasks is the research into evolutionary robotics. Evolutionary robotics is a technique for the automated training of autonomous robots.

This approach views robots as autonomous, artificial organisms which develop their own skills in interaction with the environment, without any human interaction. The training or evolution of a robot is inspired by the Darwinian principle of selective reproduction of the fittest individuals inside a population. Evolutionary robotics uses natural sciences like biology and ethology and makes use of techniques like neural networks, genetic algorithms, dynamic systems and biomorphic engineering [13].

Research has gone into different approaches. Some approaches, like Silva *et al.* [14], concentrate on navigation and mapping in an unknown environment, using a single robot with a variety of sensors and an artificial neural networks for classification of detected objects. Others, such as Costanzo *et al.* [3], have done research into exploration and communication between a swarm of mobile robots.

The inspiration for this project was to demonstrate how an evolutionary robotics approach could be implemented, as well as to explore the challenges presented by implementing different aspects into a single work, effectively creating a system for autonomous exploration of multiple mobile robots using limited communication possibilities, with the task of mapping comparative simple environments such as buildings.

The evolutionary algorithm to guide the exploration has been implemented alongside communication and mapping algorithms. Some results have been gathered and the system has been tested in different environments. However, due to a bug in the system that the author was not able to remove (covered in a latter part of this document), this work has not been entirely finished. With the implemented communication algorithms the robots are able to move in formation, limited by the communication range. However, in the current implementation they aim to drive at maximum extent of the communication range which can cause problems, as any sudden change in movement by a robot(e.g. when an obstacle is detected) can lead to a loss of communication as they move outside the maximum range. The current implementation is also not optimal as robot build pairs and stick together, ignoring other robots. The current implementation of the mapping algorithm is able to map any obstacle encountered by a robot. The limitations of the implementation is that the algorithm maps anything triggering the IR sensors, even other robots. At this moment each cell of the map (a standard occupancy grid map) is assumed to be the same size as the robot. This leads can lead to inaccurate readings. While the project in its current state has not full filled all goals that have been set at the beginning of it, a well functioning baseline has been established from which further work in future implementations can be done.

The following chapters will explore the background of and analysis the key parts of the project, explain their design, their implementation, as well as explain the testing procedures and experiments which were performed. The achieved work is summarised and analysed in a conclusion at the end of this document.

1.2 Aim of this Project

The goal of this project is to create a program that can control a swarm of E-Puck robots using evolutionary methods to explore and map an environment. The E-Pucks must always remain in communication, as they share a global map. The control algorithm is an Artificial Neural Network, an evolutionary algorithm. The neural network is trained so that it will learn to develop a solution on its own. The benefit of neural networks is that they are extremely versatile. A trained neural network will perform very well, even in an different environment. Neural networks also allow for quick reactions from robots with very little processing time once they are fully evolved.

The communication algorithm must ensure that all robots are within a maximum communication range to each other. In a real world application communication ranges are limited, and robots who move outside this communication range can not be located any more. The goal for the communication algorithm of this project is to ensure that all robots are always staying within their communication range, and follow the restrictions given by the communication method: line of sight is needed.

The mapping algorithm must be able to map any obstacle encountered in the environment as accurate as possible. To do this localisation algorithms need to be implemented that are able to pinpoint the robots location and rotation in the environment.

1.3 Information about the Project

1.3.1 Choice of Development Environment and Programming Language

Since the control algorithm for the robots is an artificial neural network, it needs to be trained before it can be tested. Therefore, a simulator is needed. The simulator chosen for this project is the evolutionary simulator which has been created by Elio Tuci(elt7@aber.ac.uk) and Muhanad Hayder Mohammed(mhm4@aber.ac.uk) at Aberystwyth University. The other candidate I considered was Cyberbotics Webots[1], which was dismissed as the evolutionary simulator posses some probabilities(e.g. implemented Genetic Algorithm) that Webots is lacking. The simulator by Elio Tuci was chosen because I worked with it throughout the second semester of this Masters course and therefore know it well. Other reasons include that the simulator was build for the creation of evolutionary algorithms, and as such already possesses an implemented genetic algorithm. At the beginning of the project it was deemed ambitious to create a working genetic algorithm as well as an Artificial Neural Network and train it to perform the tasks explained in this chapter. The programming language chosen for this project was C++ as the simulator is written in it.

1.3.2 Choice of Robot

The E-Puck[2] robot was chosen for this project as it is commercially available and versatile. The *standard* robot comes with 8 IR proximity sensors placed at different intervals around the robot. It is powered by a lithium-ion battery that is easily chargeable. 2 stepper motors allow it too move [11].

Since the project is done in a simulator no direct considerations need to be taken into account for battery life. However, battery life could be simulated by calculating battery usage. But this was not done because the primary objective of this project took precedence.

The E-Puck is also useful because there are s a wide range of extension boards available for it, and its bus interface makes it possible and easy to design and add extension boards. For this project the official Range and Bearing Board is used. More info about that can be found in section 1.4.1 [6].

1.3.3 Choice of Environment Design

An environment was designed to train the Artificial Neural Network. The environment represents a large room through which the robot swarm can move. There are multiple obstacles placed throughout the room to train and test the swarm's communication and mapping abilities. A representation of the created environment can be found in Appendix B.1 on page 42.

[1] `http://goo.gl/BrPK98`
[2] `http://www.e-puck.org/index.php`

1.4 Analysis

1.4.1 Communications

The Communication capabilities of the E-Puck where analysed. The Standard E-Puck comes with bluetooth communication and posses now WiFi capabilities.
Bluetooth communication for this project has been deemed infeasible as Bluetooth communications can take somewhere around 19.5 ± 4 seconds. A multi robot exploration and mapping project such as this requires almost constant communication, in which case bluetooth connection times of $\tilde{1}$9 seconds are too long.

There are a few proposed ways to implement WiFi communications on the e-puck robot. One of the methods was proposed by Christopher M. Cianci *et al.* [2] is the creation and implementation of a WiFi extension board for the e-puck, enabling communication between ZigBee and other IEEE 802.15.4 compliant transceivers. There designed communication board is based on the MSP430 Microcontroller[3] and the Chipcon CC2420[4] radio. Allowing the e-puck a communication range between 15cm and 5 meters. However such an board is not commercially available and would need to be custom designed and build, which is outside the spectrum of this project.

For the purposes of this project the use of the official e-puck range and bearing board has been deemed appropriate, as it is would be commercially available. The range and bearing board is an extension board for the E-Puck which allows for localisation and local communication between E-Pucks using infra-red transmission. The board is powered by its own processor and consists of 12 sets of IR emission/reception modules. The board was first designed and build by Guiłrrez *et al.* [6]. The performance of the range and bearing board is very good, the only limitation being that direct line of sight between robots is needed. As documented in the paper by Guiłrrez *et al.* the board has an effective range of up to 6 meter. With less than 1cm in distance and less than 2 degrees in bearing error for distance bellow 1 meter. For the sake of scaling and to simulate limited communication possibilities the maximum range for the range and bearing sensors are reduced to 60 cm.

1.4.2 Evolutionary Algorithm

The control algorithm for this project is an artificial neural network assisted by an genetic algorithm.
The purpose of the neural network is to handle the control of the robots and asses it's own performance using a predefined fitness function. The genetic algorithm is used to evolve the neural network using the fitness assessment done by the Neural Network.
This section will explore and explain the background of this approach.

1.4.2.1 Artificial Neural Network

The control algorithm used for this project is an artificial neural network(ANN). Artificial Neural Networks are inspired by the brain. A biological brain functions by passing electrical signals

[3] `http://www.ti.com/product/msp430f169`
[4] http://www.ti.com/product/cc2420

through nodes, so called neurons. Neurons of the brain can be compared to simple Input/Output connectors which transmit pulse coded analogue information. The relation between a the inputs and outputs can be displayed a simple sigmoid function [8]. To a neuron not all inputs are the same however, different inputs(i.e. outputs from other neurons) have stronger influence on it than others, in neuroscience this is defined as synaptic weights. This influence can be trained, in the course of a life a neuron *learns* to trust some inputs more than others, the same training is done in an artificial neural network. The synaptic weights are represented by the *weight* value each link between 2 nodes holds. This weight value is calculated and evolved using a genetic algorithm.

Artificial Neural Networks have been used in similar projects. One such project by Silva *et al.* [14] used a hierarchical Neural Network to classify data gathered by ultrasound sensors and image data from a camera. The classification was used to determine if a object detected by the ultrasound sensors and the camera is an obstacle or not. Costanzo *et al.* [3] propose an approach for the self-deployment of a swarm of mobile robots. In their approach a Neural Network is is used to control the robots while a Genetic Algorithm is used to train the neural network. Their work is very similar to the program created in this project. The aim of the paper by Costanzo *et al.* was to create a neural network capable of deploying mobile robots to cover as much as possible of an environment. Every robots has a given communication range which is seen as the area *covered* by that robot. In their work the robots, controlled by a Neural Network, would deploy and explore the environment and stop once they find a suitable position. The difference between the work proposed by Costanzo *et al.* and this project is that the goal of this project is to map an environment. While this still means that the robots need to have covered the entirety of the environment of the map at least once to map all obstacles the aim is to work with a swarm limited in size so that the robots always need to move. In future work this could be extend to research into how many robots are needed to cover a environment of a given size fully. On the other hand the work by Costanzo aims at effectively deploying a network a mobile sensors which can act as communication and wireless sensor network. In their experiments Costanzo *et al.* used up to 64 robots in order to cover their experiment environments. The size of the robot swarm used in this work is more limited, increasing the responsibilities every robot has while at the same time forcing the swarm to always be moving in order to cover an entire environment.

1.4.2.2 Genetic Algorithm

Genetic Algorithms(GA) are a evolutionary algorithm inspired by the genetics of living organism. An GA works by having a population of genes, which make out an chromosome. The genes in regard to this GA represent the weights used in the neural network. A chromosome is constructed as followed, the first 1 gene holds a number representing the genotype length, in other words the number of genes in this chromosome and thereby the number of weights in the neural network. This number is calculated when the neural network is created at the start of the program and passed along to the GA. The following genes represent each a weight for a specific link between 2 nodes in the neural network. The value is a number between 0 and 1. The last gene in the chromosome holds the fitness assigned to this chromosome, which is calculated using the fitness function.

An genetic algorithms modifies the genes using operators which mimic their biological counterpart.

The genetic modifier described as *crossover* switches genes in chromosome to vary the genetic pool between generations. This crossover can be done with either single genes, or groups of genes depending on the implementation, and of course any restrictions based on the nature of the data.

The other operator is mutation, which is the possibility that a random gene switches its value to another random value. Which genes are chosen for crossover is based on the selection method implemented in the algorithm. The third major part of an GA is the selection method, the method that is used to choose which chromosomes of the gene pool should carried over into the next generation. The selection method implemented in this simulator is *Roulette Wheel Selection*. This selection method is part of the *Proportionate Reproduction* scheme this reproduction scheme chooses individuals to be carried over into the next generation based on their objective fitness function [5].

The basic part of the selection process is that the fittest individuals have the highest change to be carried over. This replicated nature in a way that a fitter individual tends to have a higher change of survival and will go forward to the mating pool of the next generation. However weaker individuals(chromosomes) are not without a probabilistic change to get selected. It is called roulette wheel selection because its graphical representation is similar to a roulette wheel, as figure 1.1 shows [18].

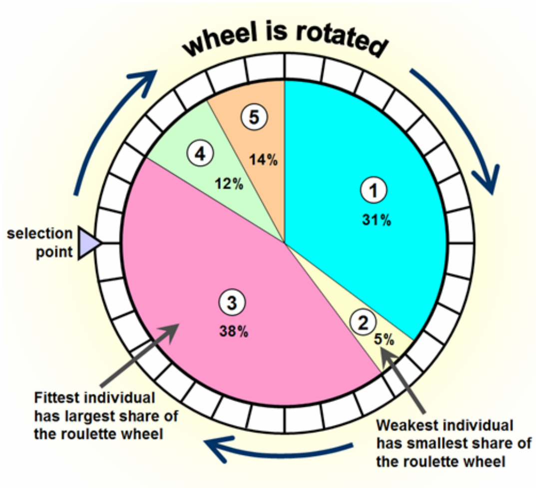

Figure 1.1: Roulette Wheel Selection[5]

As can be seen in figure 1.1 individuals with a higher fitness occupy a large of the overall available area.

1.4.3 Fitness Function

A fitness function is used to guide the evolution of the neural network and the GA. It is used to rate the performance of a neural network based on a predefined formulae or criterion.

[5]Image credit: Newcastle University Engineering and Design Center, accessed 7th of September 2015 `http://goo.gl/uwMSVB`

The baseline for the fitness function implemented in this project was proposed by Floreano *et al.* [4].
The proposed fitness function is a behavioural fitness function. This means it measures the quality of different features while a robot is performing it's allocated task. It does not measure directly how well the robot does its allocated task, but rather measures various aspects of the robots behaviour and rates it [12]. This fitness function is a good baseline as it prevents the robot from stopping, spinning on the spot, and crashing into obstacles, but still being close enough to an obstacle to get a positive return(obstacle found) from at least 1 sensor reading.
This fitness function was expanded upon as new features where implemented to lead to more complex behaviour of the robot.
The final fitness function is shown and explained in Chapter 2.1.2 on page 13.

1.4.4 Localisation

In real life scenarios GPS information is not always available, especially when mapping the insight of buildings. For the sake of this project the localisation and rotation readings are taken from the simulator.

While not realistic, a time shortage prevented further development of a localisation algorithm. Thought went into to question of localisation and there are a couple of approaches which could be taken in future work to incorporate them. One of the approaches is to use odometry, in which the robots position and location is calculated using the knowledge of how many steps the stepper motors did between readings and the wheel diameter of the robots wheels. Knowing how many *steps* equal a full rotation, in case of the E-Pucks motors this is 20 steps[6], and the diameter of the robots wheels, around 41mm[7], allows to calculate what distance the robot has driven in a straight line.

The rotation of a robot can be calculated using the same data combined with the knowledge of the robots wheelbase. However odometry is not a perfect localisation method. Uncertainty about for example the robots wheel diameter, or a wrongly calibrated stepper motor can throw of the location and rotation calculation completely. In real world applications, or simulators which simulate real world properties such as friction between the wheels and the floor, can cause additional problems. This *error* in the calculation and movement get bigger overtime unless the localisation and rotation values stored in the robots memory are reset at certain intervals. In order to be able to reset it however exact knowledge of the location in the world is needed, not something possible in all environments. Therefore odometry can be at best be seen as an estimate of the robots location and rotation.

Another possible approach is to locate a robot using another robots sensor, such as the IR sensors on the range and bearing board. However without prior knowledge of where the *searching* robot is in the world it is impossible to calculate the location of the *searched after* robot, only it distance and bearing from the *searching* robot. This knowledge might be enough for some applications, however there are also limits to the localisation possibilities using this approach. IR sensors beams widen over distance, meaning the error of an bearings reading increases over

[6]e-puck.org webside, accessed 8th of September, 2015, `http://goo.gl/YpQ2nf`
[7]e-puck.org webside, accessed 8th of September, 2015, `http://goo.gl/YpQ2nf`

distance. Therefore there is a a limit to over how large distances a robots location can be calculated using this.

Thoughts went also in to combining both the odometry calculations with the range and bearing information of the robot, however a shortage of time let to that no method was implemented in the system.

1.4.5 Mapping

To map the environment the E-Pucks IR sensors are used. Using the knowledge of the robots position as well as the sensor read out it is possible to calculate the position of an obstacle in regard to the robot.

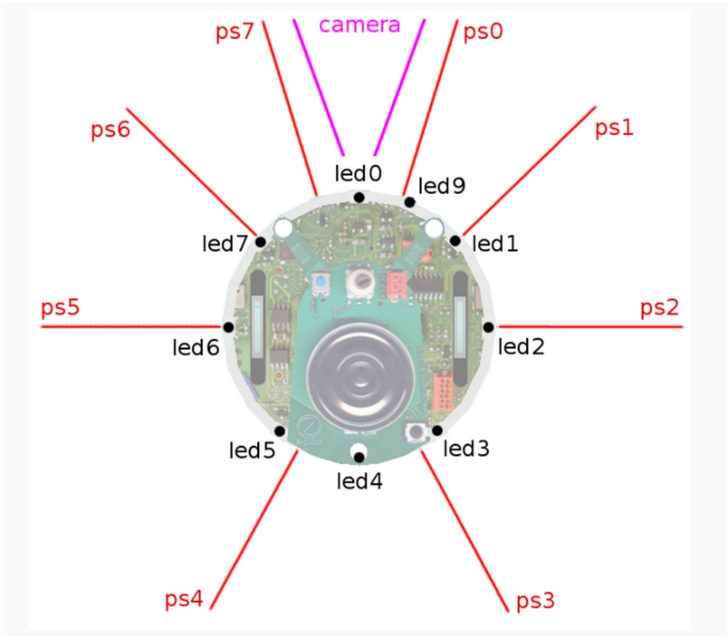

Figure 1.2: E-Puck Sensor Placement[8]

Figure 1.2 shows the placement of the robots IR sensors, labelled *ps0* through *ps7*. With the knowledge of where a particular sensor is placed on the robot combined with the knowledge of the robots position, rotation and the return of the IR reading, it is possible to calculate the placement of a obstacle in the environment.

The robots all share a single global map, which is updated with the position of all robots, as well as the position of any encountered obstacles, at each iteration. The map it self a standard occupancy grid map: a 2 dimensional grid where each cell can have 1 of 3 possible values: 0 = unexplored, 1 = obstacle, 2 = robot position.

[8]Image credit: Webots User Guide, accessed: 8th of September, 2015, `http://goo.gl/pTmp45`

1.5 Methodology

The life cycle model used for this project is "Feature Driven Development" (FDD) as it seems the be more appropriate for this project than other models, such as Extreme Programming or Test Driven Development. The reason FDD is more appropriate is that this is a single person project, as well as the requirements allowed for easy distinguish in which order features need to be implemented. For example: the controller(neural network) needs to be implemented to be able to train it. Once the it is implemented and the robots are able to move based on its control, the communication between the robots can be implemented. Only once this is done and tested the mapping algorithm can be started to be developed, as the features build on each other.

The milestones of the project rather small and incremental "upgrades" on each other. For example the fitness function went from "move in certain direction" to "move in a certain direction and avoid obstacles" to much later "move throughout the environment, don't spin at the same spot, avoid crashing into obstacles but be close enough to map them and stay in communication range with other robots".

Chapter 2

Design

2.1 Overall Architecture

2.1.1 Artificial Neural Network

The robot swarm is controlled by an artificial neural network. The neural network is consistent of 8 inputs, 3 hidden nodes, and 4 outputs. There are bias nodes connected to the hidden and output layer, both of which are always set to 1. A representation of the network is shown in figure 2.2. The ANN is a multilayer feed forward network, meaning all layers nodes are connected to all nodes in the following layer and that data is only passed forward in the network, never back as would be the case using a different type of neural network, like a backpropagation algorithm.

The inputs are taken from the E-Pucks 8 IR proximity sensors. The hidden layer consists of 3 hidden nodes, which give more computational depth to the network. From the 4 outputs of the neural the speed of the 2 stepper motors of the e-puck are calculated. This is done by calculating the difference between them. The weights of the neural network are generated by a Genetic algorithm, which is part of the used simulator. The genes created by the GA are a value between 0 and 1, however the neural network algorithm scales them to be be a value between -5 and 5. The reason for scaling is explained in section 2.1.1.2 of this chapter.

The bias is needed to be able to shift the entire sigmoid function along the x axis.

2.1.1.1 Input Layer

The inputs to the ANN are taken from the robots IR sensors which return a value between 0 and 4096, however the inputs have been scaled to a range of 0 to 1.

As can be in the in figure 2.3 the returned IR sensors value rises drastically after the robot comes closer to an obstacle than 3 centimetres.

[1] Image credit Stackoverflow, accessed 8th of September, 2015 `http://goo.gl/Vktx0X`
[2] Image credit: Webots User Guide `http://goo.gl/kyCINM`

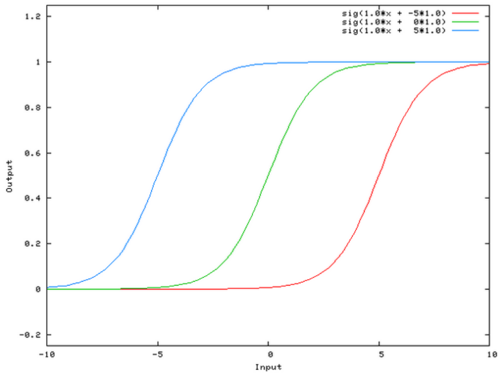

Figure 2.1: Representation of a Shifting Sigmoid Function[1]

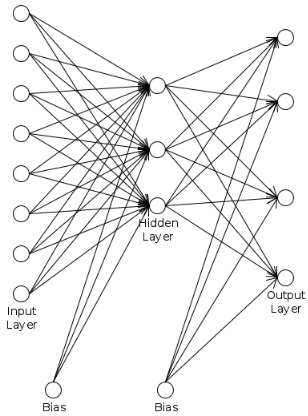

Figure 2.2: Representation of the Neural Network

2.1.1.2 Hidden Layer & Sigmoid Function

In the hidden layer the sum of all inputs to a node is multiplied by the weights to that node and than fed to sigmoid function.

A sigmoid function refers to a mathematical function that has an "S"(sigmoid) shape. A sigmoid, or activation function, is an abstract representation of a neuron firing(activating) in the brain. There are a number of different approaches to activation functions, the simplest is a simple binary step function, with only 2 stages: *on* or *off*. The activation function in this neural network is a sigmoid function which is given by:

$$S(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

Where e represents *Euler's number* which is 2.71828[...][3] and x represents the input to the function, in this case the sum of all inputs multiplied by the weights. The output of the sigmoid

[3] Eulers Number wikipedia, urlhttps://goo.gl/jwHciq

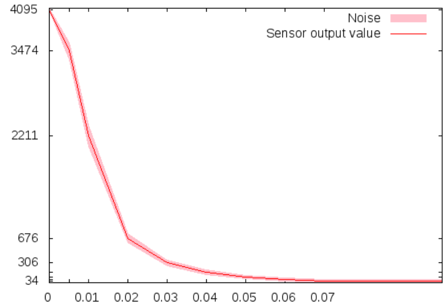

Figure 2.3: IR Sensor Response Against Distance[2]

function is a number between 0 and 1.

Sufficient scaling of inputs is important as it increases the *range* of the activation functions calculation. Figure 2.4 shows a graphical representation of this sigmoid function. The red lines represent the range on which an activation can happen if the inputs are between 0 and 1. Figure 2.5 however shows the area of activation if the inputs are between -5 and 5. The graph shows that scaled inputs have a higher activation range which leads to better performing networks.

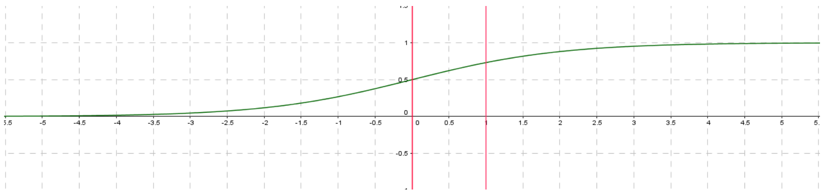

Figure 2.4: Activation Range for Unscaled Inputs[4]

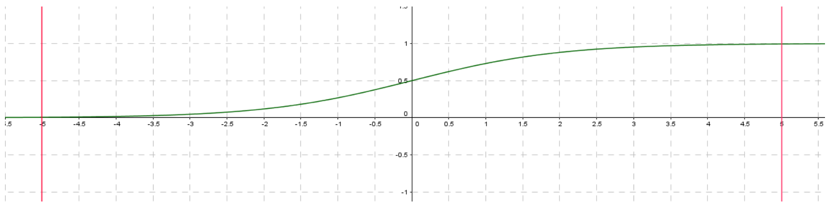

Figure 2.5: Activation Range for Scaled Inputs[5]

2.1.1.3 Output Layer

In the outputlayer the outputs from the hiddenlayer are multiplied with their respective weights, and than passed to a sigmoid function again. The outputs of this layer are than send back to the experiment class. The velocity which will be set for each wheel is calculated by the difference

[4]Created by the author of this document using geogebra `https://www.geogebra.org/`
[5]Created by the author of this document using geogebra `https://www.geogebra.org/`

between the 2 of the nodes. The *first* 2 nodes are used to calculate the velocity for the left wheel, the *last* 2 nodes for the right wheel.

2.1.2 Fitness Function

The fitness function is used to guide the evolution towards to the best solution. The fitness function used in this project is as follows:

$$f = mean(V_l, V_r) \times (1 - \sqrt{|V_l - V_r|}) \times (1 - S_{ir}) \times coms \times pos_y \tag{2}$$

Where V_l and V_r represent the velocity of the left and right wheel, S_ir represents the highest IR sensor reading of the that iteration, $coms$ represent the robots distance to a another robot in the swarm. pos_y represents the y position of the robot. This is a modified version of a fitness function proposed by Floreano *et al.* [4]. Their fitness function, as is shown in equation 3, covers the basic movement rules for a robot.

$$f = mean(V_l, V_r) \times (1 - \sqrt{|V_l - V_r|}) \times (1 - S_{ir}) \tag{3}$$

In this equation the values are normalized to fit:

$$0 \leq V \leq 1$$
$$0 \leq \Delta V \leq 1$$
$$0 \leq S_{ir} \leq 1$$

Where V is the velocity of a wheel, here 0 would mean full speed backwards and 1 full speed forwards. Stand still is represented by a value of 0.5.
ΔV represents the absolute difference between the left and right wheel velocities. The IR reading S_{ir} is normalized to a number between 0 and 1, where 1 would mean the robot is very close to an object, 0 would mean the sensor has no return at all, i.e. no object is in range. However the simulator introduces noise into system, similar to how there would be noise in a real world application. Therefore this value will never be exactly 0. The standard values of the velocity are all in a range from -1 to 1 and have been normalized to 0 to 1 in order to ensure that the fitness function works properly [4]. The rationality behind the different *segments* of the fitness function is a follows: $mean(V_l, V_r)$ is the average between the velocities, this ensures that the robot does not stand still as stopping would prevent the fitness value of increasing. This is also means that the fitness is increasing when the robot is turning. $1 - \sqrt{|V_l - V_r|}$ motivates the robots to move in a straight line rather than driving in large circles. $1 - S_{ir}$ does reward the robot for not crashing into walls as the normalized function does return 1 when the robot is close to colliding with an obstacle, this would mean this sub-calculation would return 0. $coms$ represents the the distance between it and another robot. The higher the value the better, this prevents the robots from driving close to each other, and gets them to drive at the full extent of their communication range. If the robots pass a certain distance this value is set to 0. pos_y is the individual robots *y* position. This is used to reward the robots for moving *downwards* in the environment as this increases the *y* value. This was implemented in the beginning of the development phase to drive the evolution forward and was planned to switched out with a fitness function segment that rewards the robots for moving towards unexplored areas, by checking if the map cell the robot is located in has already been

marked previously(as ”explored”). Due the non-evolution bug, covered in more detail in chapter 3, further development of this came to a stand still, and at the moment of submission this work uses the robots *y* position in the fitness calculation.

2.1.3 Genetic Algorithm

The genetic algorithm was not implemented for the sake of this project, it is build into the simulator. Therefore only the parameters used for the GA and their effect on the evolution process will be covered here. The GA parameters used for this project are the same as has have been used in previous course work for this MSc course. They have been found to be sufficient in guiding the evolution. The GA parameters are:

Number of Chromosomes	100
Number of Elite	20
Probability of Mutation	5%
Probability of Crossover	30%

Table 2.1: Genetic Algorithm Parameters

The number of chromosomes defines the size of the gene pool. A number of 100 chromosomes are generated each generation. Of those an elite of 20 chromosomes with the best assigned fitness function is chosen using the roulette wheel selection method to be taken to the next generation. A gene has a 30% change to be selected to be part of a crossover operation. All genes have also a 5% change to be mutated. The selection of how much chance a cell has to be selected for crossover or mutation has a big influence on the evolutionary properties of a genetic algorithm. While a too high crossover rate at the beginning of a evolution can prevent good evolution results. It is very important that ,good, genetic material is maintained during the early generations in order to guide the evolution forwards, having a too high crossover rate increases the change that crossover happens to good genes, eventually making them worse. At the other hand the crossover rate can not be to low either as that would lead to premature convergence of the evolution, leading to local minima. Some authors propose *dynamic* crossover rates, low rates at the start of an evolution and increasing the rate at a later part of the evolution process, this has however not been explored in this work as the genetic algorithm was already implemented in the algorithm and has not been modified by the author of this paper. The mutation rate is important throughout the evolutionary process, however is conventionally kept low as a too high mutation rate can stop / damage a evolution by having a too high chance to change good genes into bad.

2.2 Program set up

The user specifies the environment and other simulator parameters, such as how many robots to use, how many generations to evolve, iteration length per generation, where and how often to save genomes, in the *init_run.txt* file located in the *MISC* folder. This file directly set’s up the environment, and once compiled the program can be executed by going to the *build* folder and typing:

$$./EVOSIM - e - nname_of_the_run - srandom_seed$$

Where $random_seed$ is a simple string of numbers between 0 and 9.
This method is used to execute the program in *evolution mode* which will evolve it using the parameters defined by the user in the *init_run.txt.* file.

After the evolution is finished, the result can be shown by executing:

$$./EVOSIM - v$$

which will execute the program in *viewing mode*, the genome files saved by the simulator during the *evolution mode* can be found in the *EXP/GENOME/* folder. Generally the last genome is the furthest developed genome, chromosome 0 of the genome file is the chromosome with the highest fitness. For more information about how to execute the program and which packages to install to run the code on your system, please refer to the installation guide, which has been delivered electronically with this submission.

A UML representation of the program can be seen in appendix B section 2.6 from page 54 and onward. Note that due the size of the classes it was not able to represent all of them in one large figure, and still be able to read them. Therefore the UML diagram has been split up into multiple figures. Figure B.3 represents a very simplified class diagram without functions or member variables, it only shows the relationship between the different classes. The classes shown in this representation are not all classes used in the simulator, they are however all classes that have been actively used or been modified by the author in the course of this project. As the other classes of the simulator have not been modified or accessed they are treated as an blackbox and excluded from this report.
Figures B.4, B.5, B.6 and B.7 represent the UML diagrams for the experiment, parameter, simple_agent, and mycontroller classes.

The parameter class reads the parameter file in which the user specifies program parameters such as how many generations should been run, how many evaluations and iterations per generation, as well as defining the environment. This class also handles the genetic algorithm and the neural network, and the agents. The SIMPLE_Agent class represents the variables and functions available to the robots. The parameter class creates as many agents as specified by the user.

MyController class holds the neural network and inherits variables and methods from the Controller superclass. The user can change the number of hidden nodes in the MyController header file. The EXP_Class is the class where a user set's up the experiment, and from where the experiment is handled. In this class the fitness function is defined. Occupancy_Map class holds the variables and function needed to create and manipulate the map. EXP_Class holds a single occupancy_map object, the map.

2.3 Communication

The communication between robots aspect of this project has been simplified due the limited time available. What that means is that rather than implementing and simulating full network protocols the communication has been simplified to simple distance. A maximum communications distance is defined based on the background information which can be found in chapter 1 section 1.4.1 on page 4. Should a robot move outside this maximum communications range the it is assumed that said robot is out of range. This corresponds to a real-world implementation in that different communication systems(wifi, bluetooth, IR pulse) have a limited maximum range, and in case of IR methods a inaccuracy which increase over distance needs also to be considered. While the *baseline* project did only aim to simulate communications inside a maximum distance, thought and work went into the implementation of a more accurate communication model, which is explained in chapter 4 section 4.2.1 from page 29 and onward. Based on the background information available for the used communication approach, the range and bearing sensor, a maximum communication range of 0.6 or 60cm has been defined.

2.4 Mapping Algorithm

2.4.1 Mapping Procedure

The mapping algorithm works by locating the robot, using the simulators inbuilt functions.
The robots share a global map, the map itself is a 2D occupancy grid. Where each cell can have one of 3 possible values:

$$\begin{aligned} 0 &= \text{Unexplored} \\ 1 &= \text{Occupied} \\ 2 &= \text{Robot's Position} \end{aligned}$$

The entire map is initialised to be 0 in the beginning. For this project each cell is defined as a space a robot can move through. If a robot is located only partially between 2 cells, its location will be rounded or down, depending on its coordinates, so that at it is always assumed that the complete robot is inside a cell.

To map the right cells it is important about to know the heading of the robot. This is done by taken the robots rotation and assigning one of pre-defined headings to it based on it. The possible headings are inspired by compass direction and are: North, East, South, West, North-East, South-East, South-West, North-West. Once the robots position and heading are known surrounding cells can be marked based on the IR sensors readings. Since a cell has a the same dimensions as a robot(which has a diameter of 70 mm, and the IR sensor has a range of about 4cm(where an obstacle can still be located with any form of certainty, see figure 2.3 on page 12 for more information, the robot is only able to map any cells adjacent to the cell occupied by the robot. Knowledge of the robot's sensor placement allows for easy determinate which cell to mark as occupied based on the IR sensor activation value.

PS6	PS7 & PS0	PS1
PS5	↑ Robot Heading	PS2
	PS4 & PS3	

Table 2.2: Correlation between map cells and Robot sensors

Table 2.2 shows a representation of how robot sensors correlate to to the adjacent cells of where the robot is located. Here *PS* stands for *proximity sensor* followed by the sensor number. See figure 1.2 on page 8 for a graphical representation of the E-Puck robot and its sensors. Knowing the robots location, heading, and sensor activation a cell is easily modified using by taken the robots X and Y coordinates and increasing or decreasing both of them to correspond with the cell which needs to be marked. This is done using the above representation as guideline. Every iteration each robot marks its position on the map with '2' and each found occupied cell as '1'.

2.4.2 Map Visualisation

In order do visualize the map, the X and Y coordinates of all cells marked with '1' are saved to a text file. This file can be loaded into a different program and be visualized using the C++ SDL2 library. A map that has been created by the mapping algorithm and been displayed by visualisation program can be seen in figure 4.3 on page 32.

Chapter 3

Implementation

This chapter is divided into multiple smaller sections. The first sections will describe how the neural network was implemented, and cover the evolution of the fitness function and the test done using the neural network. Following the implementation of the mapping, communication and visualisation algorithms will be covered in detail. Throughout the chapter updated fitness functions will be shown and the changes made to both the fitness function and the algorithm reflected upon and explained why they were done. This structure is meant to show the reader how the program *evolved* and why changes were made in the order they where implemented.

3.1 Implementation of the GA Assisted Artificial Neural Network

The Artificial Neural Network(ANN) inherits the simulators controller framework, and uses the inbuilt genetic algorithm to evolve the weights used in the Neural Network. The network implementation in itself was swift and easy as different Neural Networks had been implemented during different assignments in one of this years modules. The following subsections will explain what the different functions in the neural network do.

3.1.1 Constructor and Genotype Length Computation

The code for this can be found in Appendix B 2.2.1 on page 43. The constructor is used to initialise the Neural Network. Upon initialisation it calls a function to calculate the genotype length, the amount of genes needed for all weights in the network. This number is passed to the genetic algorithm so that the required number of genes can be created. The number of required genes is calculated by multiplying the number of nodes of the different layer with each other and than summing the results of that calculation. It has been decided to implement a neural network with 3 hidden nodes to add computational complexity compared to a simple 2 layer perceptron network.

3.1.2 Initialisation

The code for this function can be seen in Appendix B 2.2.2 on page 43. This function initialised the arrays needed for the neural network. It also takes a vector of *genes* as a parameter, this vector holds the genes as they have been evolved by the genetic algorithm. The genes are scaled according to be in a range of -5 to 5, refer to section: 2.1.1.2 on page 11 for more information as to why gene scaling is important. The scaled genes are assigned to a 2D array representing the weights of the neural network.

3.1.3 Step function

The step function is called every iteration. The network calculates the outputs based on the inputs, the evolved weights and the calculations done in the hidden and output layer of the network. The code for this function can be seen in Appendix B 2.2.3 on page 44

After the neural network was implemented work on the fitness function was started in order to test the performance of the neural network.

3.2 First Fitness Function and Neural Network Performance Test

The first fitness function was a behavioural fitness function as proposed by Floreano *et al.* [4]. This builds the basis for all following changes to the fitness function.

$$f = mean(V_l, V_r) \times (1 - \sqrt{|V_l - V_r|}) \times (1 - S_{ir}) \times pos_y \tag{1}$$

Equation 1 shows the base fitness function that was implemented at this point. A thorough analysis and explanation of the base fitness function can be found in chapter 2 section 2.1.2 on page 13. First tests had shown that the standard fitness function is not able to move the robot, as the robot never started moving. To counter this problem the pos_y was added to the function. Pos_y represents the robots position on the y axis, so the robot is rewarded for increasing its y position, by moving downwards in the environment. The reason that the standard fitness function alone was not able to move the robot was that the robot spawned to far away from an obstacle so $1 - S_{ir}$ was not able to return any number. In their papers Floreano *et al.* [4] as well as Nelson *et al.* [12], which implemented Floreano's fitness function, the environment was designed so that the robot always had a positive sensor return. As this was not the case in this test environment the fitness function needed to be modified. With the modified fitness function the robot was able to traverse most of the environment, only the small cap in the between the 2 boxes at the end of the environment proved to be a problem.

Code for this specific fitness function will not be shown, as its components are still included in later fitness functions. It has therefore been deemed unnecessary to duplicate the code multiple times. The code will be shown in and explained in a later part of this chapter when the final fitness function is discussed. The test environment in which the neural network and this fitness function was tested can be seen in figure B.2 on page 46.

It was not deemed necessary that the robot would need to finish the test environment, the test was meant to show if the neural network is able to evolve and guide the robot as well setting up a base level fitness function which could move a robot while avoiding obstacles.

After this the environment which can be seen in figure B.1 on page 42 was implemented, and work on the next milestone began: multi-robot movement.

3.3 Multi Robot Movement

After the simple movement algorithm was implemented work on coordinated multi-robot movement was started. Minor changes needed to be to the experiment class, while it was designed to work with multiple agents(robots) some functions only called the first element of the vector of agents by default. These were easily changed to iterate over the vector.

An algorithm which is able to measure and return the distance and bearing information between 2 robots was needed in order to be able to move the robots in any form of formation. After talking with one of my lecturers(and creator of the simulator), Muhanad, about it, he said that he has code for that already written and has allowed the usage of said code in this project. The code can be seen in Appendix A section 1.1 on page 40. The code needs the location of a robot to measure to. It only needs this one location as it locates the robot the function is called from on its own using the simulators inbuilt functions. The second parameter passed to the function is a pointer to an vector, which the algorithm will save its results too. The function only returns when the distance between the robots is less than a user defined distance. After integrating this function into the program this distance value was set the 0.4, which equals 40cm for testing purposes. The max distance was later increased to 0.6 or 60cm, based on the information that has been covered in chapter 2.

In order to test the the algorithm another computation was added to the fitness function, the new computation added the distance between 2 robots into the calculation.

$$f = mean(V_l, V_r) \times (1 - \sqrt{|V_l - V_r|}) \times (1 - S_{ir}) \times pos_y \times dist \tag{2}$$

Where $dist$ is the distance between 2 robots. Using this calculation the robot will try to reach the maximum extend of the maximum distance value defined by the user. As larger distance between directly increases the fitness value.
The code that was added for this calculation can be seen in listing 3.3.

Listing 3.1: Fitness function calculation to consider the distance between 2 robots

```
double comp_4 = 0.0;
     if (param->num_agents != 1) {
        if (r == param->num_agents) {
           param->agent[r]->get_randb_reading(param->agent[r -
               1]->get_pos(), randB_reading);
           comp_4 = randB_reading[0];
        }
        else {
```

```
        param->agent[r]->get_randb_reading(param->agent[r +
            1]->get_pos(), randB_reading);
        comp_4 = randB_reading[0];
    }
}
```

Note that this is the code is a snipped of the fitness function as it was used at this point in the development and there are some problems with this approach. There were a couple of problems with this approach at the time. Note that the code is designed to only use either the robot next or previous in the array of the robots, which lead to that robots would always form groups of 2 and stick together, while ignoring all other robots. A problem with the design in itself was that the robot always tried to drive close to the maximum extend of the user defined range in order to get the maximum fitness value. This is problematic as even the slightest turn of one of the robots might lead to them losing *sight* of each other. Once this is happened the robots generally where not able to find back together. Once the robots are out of *sight* this part of their fitness calculation would be set to 0.

3.4 Mapping

After the basic where able to move in formation and the fitness function was able to navigate them through the environment working on the mapping algorithm was started.

3.4.1 Map initialisation

The initialisation function is shown in appendix B section 2.4.1 on page 46. The method initialised a 2d array. In order to keep the array in memory it is an array of pointers which holds pointers to arrays rather than a simple 2d integer array. All cells in the map are set to 0 after it has been initialised, and the pointer to an pointer to an array is returned to the experiment class, from where this method is called. The map height and width are user defined in the class header file.

3.4.2 Calculate the Robot Position on the Map

The code for this function can be found in Appendix b section 2.4.3 on page 48, This function is used to calculate the robots position on the map based. The reason the coordinates can not be taken directly from the simulator is that different coordinate systems are used. In the simulator the the coordinate '0' of the *x* coordinate is in the middle of the map, everything to the right of is positive anything the the left of the center is negative *x*. The *y* coordinate starts at the top of the environment and increases as its goes downwards. The coordinate system of the map is that of an standard 2 dimensional array, with both '0' for both *x* and *y* at the top left corner, in an graphical representation. The fact that the simulator is very likely to return negative *x* values based on its coordinate system has been countered by *simulating* the same coordinate system in the map by defining a map width and height value, which is user defined and also defines the dimensions of the map array. The value of the map width is divided by 2, given the exact center of the map, which would correspond to '0.0' in the simulator. Any positive *x* values are added to this value to place it on the *right* side of the center, any negative values are subtracted so that they will be displayed on the left side of the center line.

As the robot might be located in between 2 cells of the map this function also handles the rounding of coordinates as needed. To do so the c++ math.h library is used to split the coordinate values of type *double* into integral and fractal parts, which allows for the rounding to happen. Note that all the coordinates are timed by 1000. The reason for that is to scale them from the comparative small coordinates used for the development environment, where *x* can be a value between -0.6 and 0.6 and *y* a value between 0 and 2.0 to a value which could be used in the gui program. The gui handles the coordinates by pixels, so large numbers are needed in order to display them in any form that can be visualized.

3.4.3 Calculate the Robots Heading

The heading of a robot is calculated based on its movement direction. This is needed in order to map the correct cells based on the robots directions. The code for this function can be found in Appendix B section 2.4.2 on page 47.

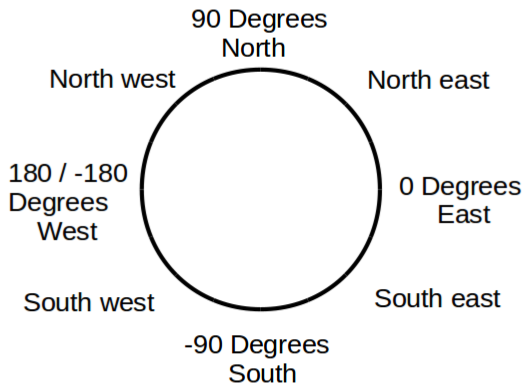

Figure 3.1: Graphical representation of the headings and the corresponding degrees

The heading is calculated based on the robots rotation, which is returned from the simulator. Rather than returning degrees from 0 to 360, the simulator returns degrees from 0 to 180 and -180 respectively. The decision fell to handle the robots directions based on compass directions as it is very easy to interpret in which direction the robot is moving on a map. There are some restrictions to when a heading is assigned. For the 4 main directions: north, east, south, west a selection threshold of +/- 10 degrees is chosen. For the *side* directions north east, south east, south west and north west a threshold of +/- 5 degrees is chosen. The reason for the threshold is it is unlikely for the robot to for example drive exactly 90 degrees. The threshold of +/- 10 degrees was chosen as this gave the best results while mapping. Other thresholds such as +/- 5, 15, 20 were tried, but either gave a too large error or almost no mapping at all, in the case of 5, since the threshold was too small so a heading was never selected.

The threshold for the side directions was chosen to be less than for the main direction since moving at an angle increased the error rate already. Here as well different thresholds where chosen, again increasing in steps of 5. Results showed that the mapping error increased drastically when thresholds larger than 5 degrees where selected.

3.4.4 Deciding Which Cell of the Map to Mark

To algorithm to mark a cell in the map is divided into multiple smaller functions. Note that to make describing the direction a sensor is mapping naval terms are borrowed.

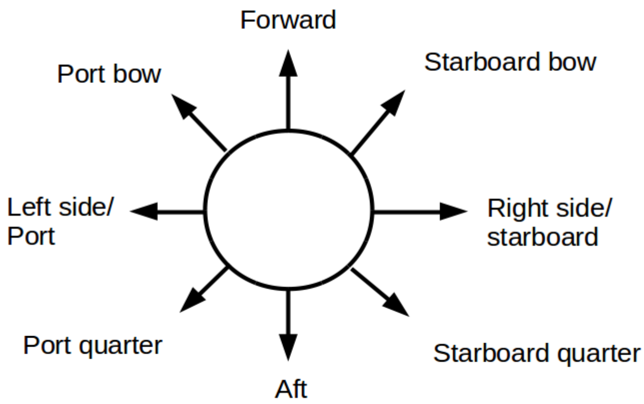

Figure 3.2: Representation of the terminology used to describe the robot

Figure 3.2 shows a representation of the naval terms borrowed and how they apply to the robot.

3.4.4.1 Return the Correct Sensor Number

The IR reading function the robots does return an array, however the cells of that array do not correspond to the IR sensor with the same number. When the mapping function is called this function is used to determine which sensor was activated. The code for this function can be found in Appendix B section 2.4.4 on page 49. In each function the sensors which are used for the mapping are mentioned, refer to figure 1.2 on page 8 to see where the sensors are placed on the robot.

3.4.4.2 Control function of the mapping algorithm

As the mapping algorithm is split into multiple smaller functions the program requires a control function which decides which of those functions to call. The control algorithm receives a number of parameters, such as an vector holding the sensor readings, the heading of the robot, the robots x and y coordinates, and a pointer to the map. It uses the previously explained function to return the correct sensor number and than calls one of the following functions accordingly to the information it was given. The code for this function can be found in Appendix B section 2.4.5 on page 49.

3.4.4.3 Set Cells in Front of the Robot

The code for this function can be found in appendix B section 2.4.6 on page 50. This function is used to set the cells directly in front of the robot, using sensor 0 and 7 on the robot. This function takes the heading, the sensor number, the robots x and y coordinates as well as a pointer to the map as the parameters, the same goes for all other functions in this section.

3.4.4.4 Set Cells off the Bow of the Robot

The code for this function can be found in appendix B section 2.4.7 on page 51.
This function is used to map cell to the front left and front right of the robot. This sensors used for this function are sensor 1 & 6 on the robot.

3.4.4.5 Set Cells to the Sides of the Robot

This function used to set the cells to side of the robot, the corresponding robot sensors are 2 and 5. The code for this function can be seen in appendix B section 2.4.8 on page 51.

3.4.4.6 Set Cells in the Back of the Robot

The code for this function can be found in appendix B section 2.4.9 on page 52. This code is used to map cells directly behind the robot, the corresponding robot sensors are 4 and 3. Since a single cell is the size of the robot sensor 4 & 3 are only able to map a cell right behind the robot. The reason for that being that while sensor 1 & 6 are placed at a 45 degree angle and therefore are able to map cells to the front left and front right sensor 4 & 3 and placed at a much smaller angle.

3.4.4.7 Marking a Cell in the Map

This function is is used to mark a cell in the map. The other functions in this section calculate which cell to mark and than call this function in order to mark it on the map.

Listing 3.2: Mark a cell on the map

```
void Occupancy_Map::mark_cell(int x_coord, int y_coord, int mark,
   int** matrix) {

    matrix[x_coord][y_coord] = mark;
}
```

3.5 Displaying the Map

A separate program was developed in order to display the map data gathered by the robots. The main program can save the coordinates of the marked cells to a file at the end of a evolutionary run, if the user chooses so. This file can be loaded into the visualisation program, all that is needed is that the user is aware of the map dimensions. The program will than generate another, empty, 2d map, and mark the occupied cells on is based on the coordinates saved in the file. The map will be visualised afterwards. This program uses the C++ SDL2 graphics library. The code for this program can be seen in appendix B section 2.5 on page 53.

The code reads a file of coordinates, where the coordinates represent a cell in the map that has been marked as '1'. The file is read into 2 different vectors, one for *x* and one for *y* coordinates. These coordinates are read from the vectors to create new objects of a self defined *newSDL_Rect* struct, the structs are drawn on the screen using standard SDL operations. Note that the coordinates are scaled down by a factor of 5, this is done in order to represent maps which large dimensions(the one used for this is $2500x2500$) in an window that is $640x420$.

3.6 Non-Evolution Bug

After the mapping programs were implemented work went into refining the fitness function. The problem with previous versions of the fitness function were that the robots would always choose the next in line, or if a robot is the last in the line the one before it, in the array that holds the robots. Therefore they would always form the same groups and stick together within those groups and not work together with other robot groups. During the process of modifying the fitness function to loop through all robots and check which of them are in communication range a bug occurred that prevented the network from evolving in following experiments. The only behaviour that was able to be evolved from this was that the robots would start off, drive between 5 and 10 cm and than just make a full stop. Debug methods where tried and even resetting the state of the project to previous points using version control did not fix the the problems encountered due to the bug.

The program was tried to be debugged using standard debugging procedure: setting breaking points, checking that all objects have been correctly initialized, that there are no NULL pointers and that information is correctly assigned to variables and parsed between functions. As those techniques did not show that anything was wrong with the evolution other methods were tried to test if there existed a problem within the simulator. The actuator speed of the robot was manually set to test the kinematics of the robot model, the robot was able to move with the changes, which hints that there is an problem with the evolution as the actuator speed is taken right from the output of the neural network. The outputs of the neural network were checked, they showed that the robot should be able, though slowly, to move. Robots were placed close to obstacles and rotated in order to check IR readings for any discrepancies, this turned up nothing all read outs were as expected. Every declaration and initialisation of the neural network, experiment class, simple_agent(robot) class and map was checked for NULL pointers, faulty pointers or other discrepancies. None of which turned up any information. Checking the installed versions of the packages required to run the simulator did not show any obvious problems as all were installed, to be sure that nothing was overlooked all packages and files needed for the simulator were removed from the system and reinstalled. Testing the program on 4 different machines showed the same problems.

At this point it has been deaned that resetting the state of the project to a previous, working state using version control software(github) should be tried. However even this did not turn up any anything, since even after hard-resetting the state of the files to previous commits did not fix this problem. At which point it was assumed that it might have something to do with cache of the IDE used, that the version of the code cached and executed might differ from the actual code on the hard drive. Checking this proved this to be true, the files hold in the IDE's[1] cache differed from the files on the file system. But even after cleaning the cache did problem still prevailed. It was tested if cloning the github repository to another folder where the code was observed and manipulated directly using text editors and compiling it using the command line would fix the bug, however the bug prevailed.

Even discussing the problem with both creators of the simulator, Elio and Muhannad and going through the code with them did not turn up anything, no obvious piece of code the author simply overlooked was found. Elio Tuci proposed the use of a different fitness function in order to test if this approach would fix the problem.

3.6.0.8 Different Fitness Function

The new fitness function is taken from a paper by Trianni *et al.* [16].

$$f = \frac{|V_l| + |V_r|}{2V_{max}} \times (1 - \sqrt{\frac{|V_l - V_r|}{2V_{max}}}) \times (1 - \frac{S_{ir}}{IR_{max}}) \tag{3}$$

Where V_l and V_R represent the speed of the left and right wheel and V_{max} represents the maximum possible speed for a wheel. S_{ir} represents the maximum IR sensor activation at that time step, and IR_{max} represent the maximum possible IR sensor value. All values need to be normalised, for this approach and have been normalised to :

$$0 \leq V \leq 1$$
$$0 \leq \Delta V \leq 1$$
$$0 \leq S_{ir} \leq 1$$

This approach is very similar to the original fitness function implemented in this system, which is shown in equation 3 on page 13. The difference being that the original fitness does not divide it's single segments by the maximum possible value for either the actuators or the IR sensors. The changed fitness function itself was not able to get any response from the robots, which based on previous experience from working with a similar fitness function, is based on the reason that the robots starting position is far enough away from obstacles to not get any IR reading. The environment the Trianni *et al.* used was a corridor which led to constant IR sensor returns. To fix this problem another segment, rewarding the robot to move towards the lower part, south, of the map.

$$f = \frac{|V_l| + |V_r|}{2V_{max}} \times (1 - \sqrt{\frac{|V_l - V_r|}{2V_{max}}}) \times (1 - \frac{S_{ir}}{IR_{max}}) * pos_y \tag{4}$$

[1] CLion from Jetbrain `https://www.jetbrains.com/`

Where pos_z represents the robots current y position. With this change it is possible to evolve a single robot to move towards the south of the environment while avoiding obstacles, however the same no-evolution bug occurred when trying to evolve a controller using multiple robots. It is as of this moment this document is written unclear what caused this bug and how it can be fixed.

Chapter 4

Testing & Experiments

This chapter holds the overall approach to testing. This includes the test and experiments which have been performed during the development phase, as well as a limited number of experiments. The number of experiments is unfortunately limited based on the still existing bug which prevents the program to evolve further.

4.1 Overall Approach to Testing

As the program requires a simulator to run in and controllers are needed to be evolved the only possible way to test changes made to the program, i.e. a new fitness function, refinement of some parameters, implementation of a new function, etc. can only be done by running the program in evolution mode over multiple generations in order to see the effect of the changes made on the robot behaviour and performance. That is one of the problems encountered when working with evolutionary algorithms, the evolution itself can be seen as a black box and only testing a change multiple times can give a correct view of the behaviour changes created by chancing part of the code. To run major changes multiple times in order to get an understanding of the change in behaviour is needed as evolution in itself is has a very random element. It might be that the user gives a good, or bad, seed for evolution which causes the behaviour to be better than other instances or be much worse. Examples for this which were encountered during this development were that running the same, unchanged, code multiple times with a different seed every time were as different as: Controller not evolving at all(the robot did not move), the robot only spinning around, however the average of runs evolved normally. One of the more rare evolutions caused the robot to drive only backwards.

The nature of needing an evolution over multiple generations limits testing possibilities in a way that the only way to test and document their function beyond integration testing is by evolving a controller and document it's behaviour and performance by using the simulators viewing mode after it has been evolved. This makes automated unit testing impossible. New pieces of code that where added to the program have been tested before they have been used in complete evolutions. This was done by checking, using the used IDE's debug methods if they have been integrated correctly. This was done by setting breakpoints at key points of the the program and checking that all objects have been correctly initialised, that pointers point to the correct objects and information is passed correctly between functions. In another case, when the implementation of the range and bearing sensor was tested, the function to update the actuators of one of the robots was disabled and a short evolutionary run of 25 generations was performed. This showed that the robot with working actuators evolved to move to the maximum extend of the defined communication range before stopping. This showed that defining the maximum communication range worked as intended and that the fitness function, which is designed to set the fitness value to 0 if they move outside their communication range, also worked.

During development(apart from the test environment for the first fitness function) only a single environment was used for evolution and testing. A representation of this environment can be seen in appendix B figure B.1 on page 42. While all changes to the program have been tested multiple times only the milestones will be documented in this paper.

4.2 Experiments

4.2.1 Experiment A: Multi Robot Movement

This experiment was performed after the range and bearing algorithm was implemented. It simulates the communication of the robot where the needs to be within a certain distance to each other in order to stay in contact.

Figure 4.5 shows the result of that experiment. Here the communication range has been set to a maximum distance of 0.6. The fitness function at the time of this experiment defined that the higher distance value between 2 robots the better, however after a distance of 0.6 the fitness would be set to 0. This creates some problems: the robots aim to stay at the full extend of their communication range as they receive the highest possible fitness, therefore even a small change of movement from on of the robots(e.g. avoiding an obstacle) might be enough to bring them outside their communication threshold. Another problem with the fitness function at this point is that robots only check the distance to the next robot in the array of robots, this leads to the forming of groups which can be observed in figure 4.5.

As this is not optimal a plan was made to implement a function would would reduce the fitness gradually after a certain point.

Figure 4.2 shows a representation of the communication range of the robots. After a certain distance the signal strength would start getting weaker, this is true for all available communication methods. Wifi and Bluetooth communication ranges vary though at a certain point signal strength begin to drop, the same goes for communication using the range and bearing board, at a certain point the error rate starts increasing. See chapter 1 section 1.4.1 on page 4 for more information on the different considered communication methods.

The plan was to reward the robot for staying close to the point where signal strength starts reducing, and gradually receive a lower fitness value the further they are away from that point. Should a robot go beyond the maximum communication range the fitness would be set to 0. The function was to be designed in such a way that the robot would always stay behind the point

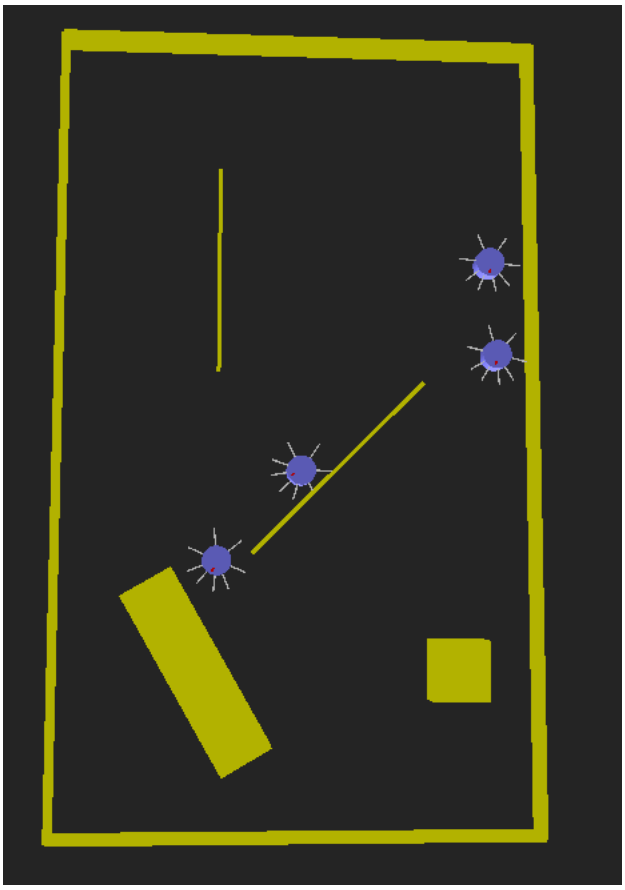

Figure 4.1: Experiment A: Multi-Robot movement

where the communication range starts decreasing, this is to ensure that the robot spread sufficiently throughout the environment.

The aim at that time of the development process was to first get a baseline of all feature implemented into the system. Therefore the function to gradually decrease the fitness value after a certain point was not implemented at this point and work on the mapping algorithm began.

4.2.2 Experiment B: Mapping

After the base communication algorithm was implemented and the robots were able to drive in *formation* the work on the mapping algorithm began, refer to chapter 3 for more information.

This experiment was performed with the same communication parameters than experiment A.

Figure 4.3 shows map generated by the mapping algorithm. Note that the while the map is not very accurate, or complete, it manages to create a ruff outline of the environment, which can be seen in appendix B section B.1 on page 42.

Examining this map visualisation gives insight of the limitations of the program at this point

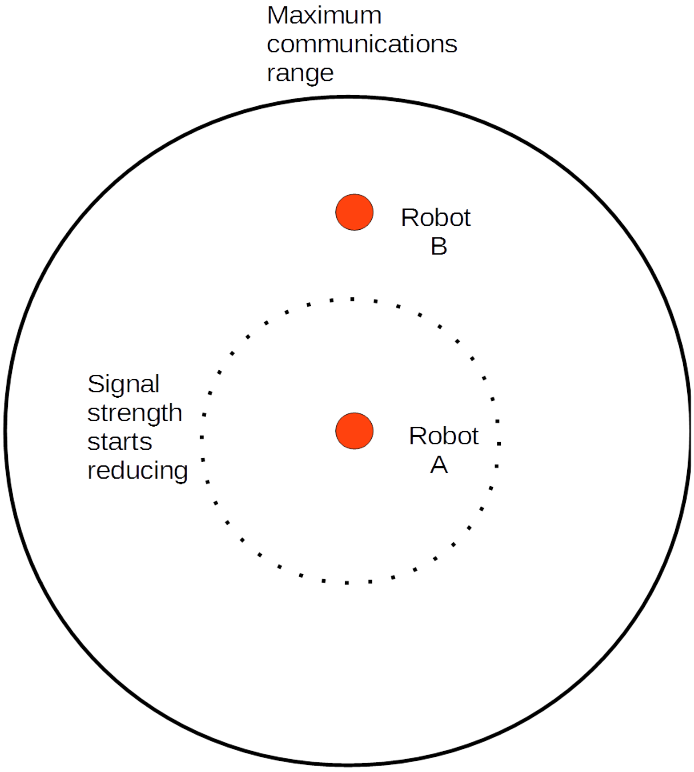

Figure 4.2: Communications range

in the project. That the map is not fully completed because the robots do not cover the entire map. This is due a few limitations. At this moment in the development process the robots moved to slow, which led to them running out of time. A user defines the number of iterations, time steps, the network should evolve with, once the maximum number of iterations is done the next evolution / generation is started. The robot speed can be influenced by reducing the number of iterations which leads the robot to drive faster in order to reach the highest fitness fastest. At the same time if the number of iterations is set to low the network does not evolve fully. At this time in the project the network was evolved with 1000 iterations and 5 evaluations per iteration. The number of iterations should have been decreased in order to force the robots to move faster

However the robot speed is not the only short coming. The robots do not cover the entire map because of how they are deployed.

Figure 4.4 shows environment 1, the red circle shows a problematic area on the map which the robot have problems with traversing. The angled between the obstacles encircled is to high, so the robots will turn away from the obstacle and will drive upwards again. On the other side of the environment the robots are following the left outer border of the environment. This is because of the fitness function, which rewards the robots to drive close enough to an obstacle to get a definitive sensor return. This leads the robots on this side of the environment to follow the wall downwards and pass through the gap between the small block and the outer wall in the lower left corner of the environment, and not exploring towards the middle of the environment.

There are problems which are rooted in the behaviour of the IR sensors and the mapping algorithm itself as well. The simulator introduces noise to the IR sensors which influences the reading

Figure 4.3: Map generated by the mapping algorithm

accuracy and might lead to cells being wrongly marked as occupied. The IR sensors do not differentiate between any obstacles or other robots, which will lead to a robot marking a cell as occupied another robot triggers its sensors. This is a flaw in the mapping algorithm which was realised but was categorized as not program breaking. It was planed to fix this bug bug after the baseline was implemented fully.

This experiment was performed multiple times in order to refine the parameters used in the mapping algorithm. In the first experiment the thresholds for the heading calculation were set to +/- 10 degrees for both *main* and *secondary* headings, see chapter 3 section 3.4.3 on page 22 for more information. The experiment was than re-run with different thresholds to see the performance and the results in the map representation. During the experiments it was shown that setting the thresholds for the *main* headings(north, east, south, west) to +/- 10 degrees is the most accurate as well as setting it to +/- 5 degrees for the *secondary* headings(north-east, south-east, south-west, north-west). While the robot will map obstacles less frequent with a threshold set to +/- 5, the resulting map will be more accurate.

The main reason for the inaccurate map creation in the way the mapping algorithm marks cells. As described in chapter 3 the algorithm marks cells in a specific area around the robot, the problem with that is that something will be marked when the sensor give a high enough reading. The way this could have been improved is by implementing an probability approach which will only mark a cells as occupied if it has a certain probability of being occupied. This involves marking a cell as *scanned* and when another robot, or the same robot, scan the same cell again the probability that this cell really is occupied increases.

4.2.3 Different Environments

In order to test the performance of the evolved neural network, different test environments were implemented and the robots where set to perform the mapping task in them. Neural networks are

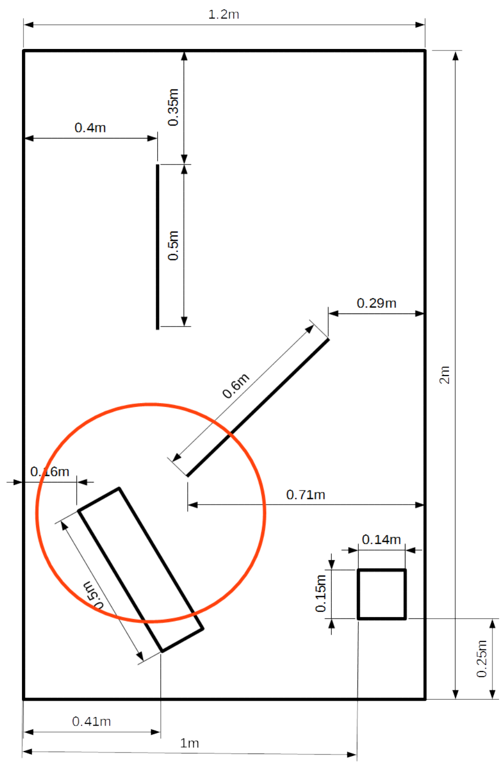

Figure 4.4: Environment 1 with a problematic area marked

very able to perform in different environments to those they have been trained in. The reason for that is that even if the environment changes, the network is trained to react to inputs from it's sensors in real time and a very specific way and will continue doing so. To test the performance of the mapping and communication algorithm in different environments 3 test environments have been created. They will be introduced and their results analysed in this section.

4.2.3.1 Experiment Environment A

Environment A has been created to test how the trained neural network reacts to strongly angled obstacles. To increase the difficulty other obstacles where added to the environment. Figure 4.5 shows a representation of the environment, while figure 4.6 shows the map that has been generated for this environment.

While 2 of the robots managed to bypass the large obstacle and move into the rest of the environment, 2 robots got *stuck* close to their start position at the top of the map. Figure 4.6 shows the generated map, as it can be seen the barely anything has been mapped. That hints at the behaviour of the robot to turn to sharply away from angled walls as barely anything of the major wall obstacle in the room has been mapped. At this moment this is assumed to be caused by the fact that the development environment, figure B.1 page 42 only has walls that stand either at large angle to the robots starting position in the top of the map or at a 90 degree angle to them. One possible way to improve on that is to add more walls with different angles to the development environment.

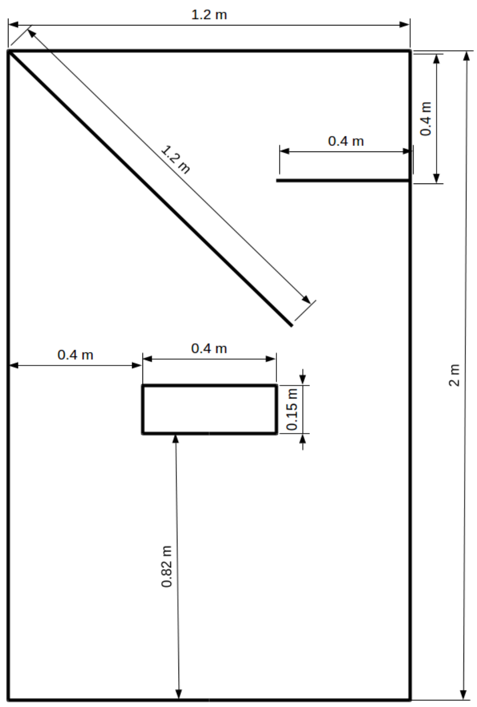

Figure 4.5: Environment A: Angled obstacles

4.2.3.2 Experiment Environment B

Experiment environment B was designed to represent a cross intersection between 2 corridors. As usual the robots would start at the top center of the map.

After the problems encountered with angled walls in experiment A the aim of this environment was to observe the mapping behaviour in an environment with straight walls, such as is often encountered within buildings.

Figure 4.8 shows the map generated for this environment. As can be seen large parts of the corridor they start it have been mapped, however some problems were encountered when it came to the intersection, and some robots drove close enough by each other to map themselves. This reinforces the limitations of the mapping algorithm observed in experiment B, that at this point all cells that are observed to be a obstacle are marked, even if it as another robot that caused the sensor return. As discussed this can be prevented by implementing a probability based mapping algorithm.

Figure 4.6: Map for environment A

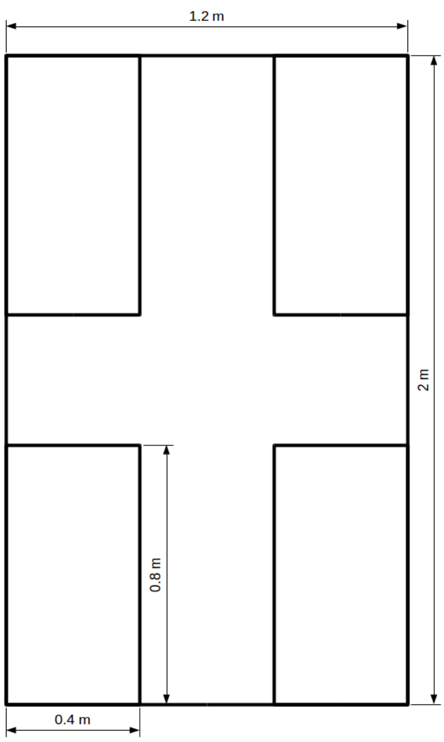

Figure 4.7: Environment B: Corridor situation

Figure 4.8: Map for environment B

Chapter 5

Conclusion & Future Work

5.1 Future Work

The developed program is a good baseline on which future work can be done to extend the capabilities it. Possible future work include improving on the communication and mapping algorithm and fitness function to fix the limitations created by the current implementation of those functions. What those limitations are and how they can be improved has been extensively covered in chapter 4 of this report.

Further work the author would like to perform is the implementation of different Neural Networks and also the testing of different Neural Networks of the same type. It would be very interesting to test and document the performance of different Neural Networks in the same environment and with the same tasks.

Other improvements that could be done is to make the implementation more realistic. This can be done by calculating battery usage and connecting the this to the movement and communication algorithms. Both movement and communication require power and implementing this would reduce the amount of movement a robot could perform. Based on this different roles could be developed for the swarm, while some robots do the movement and mapping, others could act as stationary communication nodes and thereby reduce their battery usage. Out of this more advanced deployment methods can be created assigning different *"classes"* to the robots such as *explorer* and *communication node*.

Also can the localisation algorithm be modified to make it more realistic. At this current state of the program the localisation is handled by build in functions in the simulator, which is not doable in a real world application. There are different methods which can be implemented and tested in order to find the accurate localisation algorithms. As it can be assume that GPS information for the environment is not available other methods need to be implemented. One such method is odometry. In odometry the robots location and rotation can be calculated using information about the dimensions of the robot, the wheels, the wheelbase, the stepper motors. One problem with odometry is that there always is a slight error in the calculation. This error is due to influences like friction between the wheels and the floor, faulty dimensions of the robot used in the calculation, etc. This error is increasing over time unless it can be *reset*. In previous work the author has worked with odometry using a single e-puck robot. In that project the odometry error grew to big and influenced the performance of the robot. However, using a multi robot system the odometry knowledge of one robot combined with the range and bearing information gathered by the *range and bearing* board could possibly be used to adjust the odometry error of another robot.

The communication algorithm implemented in the system is a good baseline on which improvements can be made to make the communication behaviour more realistic. In a real world application it is required that the robots maintain a communication link back to a *base station* where the map data collected by the robots is presented to a human operator. As the robots posses limited communication range robots would need to stop and form *"lines"* along which information can be transmitted from the robots furthest into the environment.

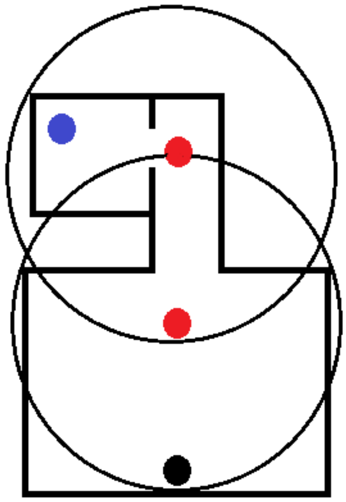

Figure 5.1: Example of a communication link needed for an environment[1]

Figure 5.1 shows how such an communication link could look like for a more advanced environment. The black circle represents the stationary *base station* to which all data must be send, the red circle represent robots which moved to their location as part of the exploration but than became *stationary* to act as communication nodes for the rest of the swarm. The blue circle represents a robot still in *exploration* mode mapping the environment. The robots communication range is represented by the large black circles. This method could be combined with the improvements to the communication algorithm suggest in chapter 4.

5.2 Conclusion

While there have been problems during the development of this program which led to a standstill in the implementation and testing of further improvements, a clear *baseline* has been reached.

[1]Image taken from the Authors own BSc Major Project "Odometry based map building", submitted May 2014

Multi robot movement and communication algorithms have been implemented, with some third party code; a fitness function, based on research, has been developed. The communication algorithm allows the robots to move in formation and explore the environment while staying within the communication range. The fitness function does evolve robot controllers which spread the robots out to maximise exploration area, and guide the evolution of their movement and obstacle avoidance behaviour. A mapping algorithm has been implemented which is able to map cells fairly accurate using the capabilities of the IR proximity sensors.

There are problems with the developed program, as has been discussed in detail in chapter 3 & 4. There is the problem with the communication segment of the fitness function. This segment causes the robots to form groups of 2s and ignore other robots. The accuracy of the mapping could be increased by increasing the resolution of the map or implement a mapping algorithm based on probability rather than simply on measurements. However a good basis has been developed during this project, giving a good baseline on which future work can be developed.

The largest problem encountered in the development was the non-evolution bug, which was a major setback and stalled all further development and progress in the last 2 weeks of the project. However, all previous evolutions have been saved and experiments have been performed. The experiments, performed in different environments, show the possibilities of the program. Analysing the results of these experiments also gives some idea of what needs to be done to train more advanced Neural Networks. Other than further developing the communication and mapping algorithm works needs to be done on the design of the training environment. The results have demonstrated that the robots have problems with angular walls, presumably because they are not exposed to them enough in the training environment. Increasing the difficulty of the training environment could train more advanced robot controllers given that more work would be done on the fitness function and the limitations of the communication and mapping algorithms are remedied.

I think the right tools and methods have been chosen for this project. The use of the Evolutionary Simulator was wise as it is easy to work with and gives a very good platform from which evolutionary algorithms can be developed. Using the C++ programming language allows to access and use all standard C++ libraries and the language is easy to work with.

Overall the work achieved in the course of the project and the results gained are satisfactory. A solid baseline has been achieved; communication and mapping implemented. Where the progress not halted by the non-evolution bug more progress could have done, but this can be done in a future implementation of this work.

Appendix A

Third-Party Code and Libraries

1.1 Range and Bearing reading

This code was written by Muhanad Hayder Mohammed(mhm1@aber.ac.uk).
This code is used to calculate the distance and bearing between 2 robots and return the results as an array. The first member of the array is the distance between 2 robots, and the second member is the bearing.
The code only returns when a result when the distance between the robots is below a user defined maximum distance. This distance can be changed by modifying the *work_range* variable(in this code snipped set to 0.4 = 40 cm).

Listing A.1: Code of the Range and Bearing function

```
double SIMPLE_Agents::get_randb_reading( vector <double>
   _to_robot_pos, vector <double> &_reading){
   double work_range = 0.4;
   randb_from = btVector3(0.0,0.0,0.0);
   randb_to = btVector3(0.0,0.0,0.0);
   this->pos = this->get_pos();
   // get the distance between your robot and to destination robot
      "_to_robot_pos"
   double range =
      sqrt(((_to_robot_pos[0]-pos[0])*(_to_robot_pos[0]-pos[0]) +
      (_to_robot_pos[2]-pos[2])*(_to_robot_pos[2]-pos[2])));
   if(range < work_range){
      _reading[0] = range;

      // get the robot orienation
      btMatrix3x3 m =
         btMatrix3x3(body->getWorldTransform().getRotation());
      double rfPAngle = btAsin(-m[1][2]);
      if(rfPAngle < SIMD_HALF_PI){
         if(rfPAngle > -SIMD_HALF_PI) this->rotation =
            btAtan2(m[0][2],m[2][2]);
         else this->rotation = -btAtan2(-m[0][1],m[0][0]);
      }
      else this->rotation = btAtan2(-m[0][1],m[0][0]);
```

```
      // check the collision accross the distance between your robot
         and destination robot
      randb_from =
         btVector3(_to_robot_pos[0],_to_robot_pos[1]+0.025,_to_robot_pos[2]);
      randb_to = btVector3(pos[0], pos[1]+0.025, pos[2]);
      btCollisionWorld::ClosestRayResultCallback res(randb_from,
         randb_to);
      this->world->rayTest(randb_from, randb_to, res);
      if(res.hasHit()){
         World_Entity* object = (World_Entity*)
            res.m_collisionObject->getUserPointer();
         if(object->get_type_id() == ROBOT && object->get_index() ==
            this->index){

            double bearing,nest_angle,robot_angle;
            robot_angle =rotation;
            if(robot_angle<0.0)
               robot_angle = TWO_PI + robot_angle;
            nest_angle = -atan2(_to_robot_pos[2]-pos[2],
               _to_robot_pos[0]-pos[0]);
            if(nest_angle <0.0)
               nest_angle = TWO_PI + nest_angle;
            bearing = nest_angle - robot_angle;
            if(bearing < 0.0)
               bearing = TWO_PI + bearing;
            //if you want to add noise bearing
            bearing += (gsl_rng_uniform_pos(
               GSL_randon_generator::r_rand )*0.30 - 0.15);
            _reading[1] = bearing;

         }
         randb_to =res.m_hitPointWorld;
      }
   }
}
```

Appendix B

Code samples

2.1 Environment Design

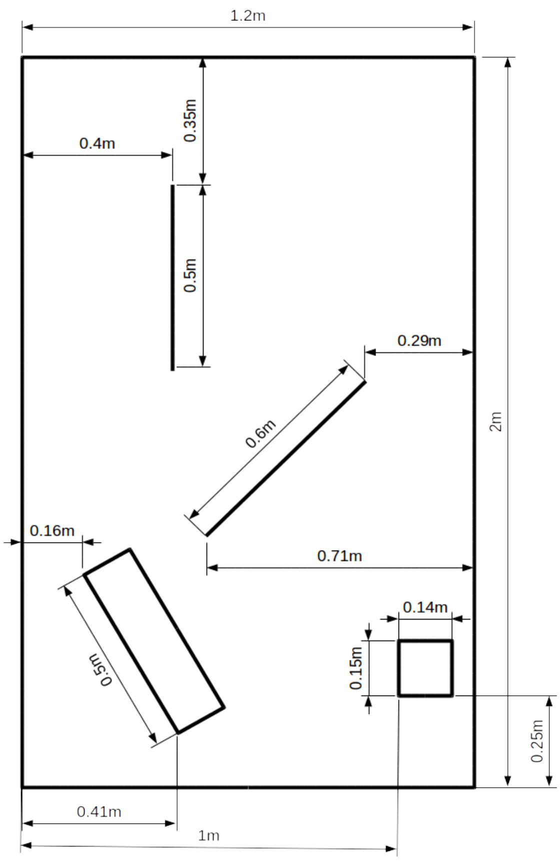

Figure B.1: Environment Design

2.2 GA assisted Artificial Neural Network

2.2.1 Constructor and genotype length computation

Here the constructor of the neural network is shown as well as the function that calculates the genotype length.
It also shows how the arrays that hold the nodes for the layers are declared. More information can be found in chapter 3 section 3.1.1 on page 18.

Listing B.1: GA assisted Artificial Neural Network constructor and genotype length computation

```
#include "myController.h"

double MyController::inputlayer[];
double MyController::hiddenlayer[];
double MyController::outputlayer[];

/*Contructor*/
MyController::MyController() {
   compute_genotype_length();
}

void MyController::compute_genotype_length ( void ){
   /*Input and hiddenlayer each have a BIAS node therefor + 1 */
   genotype_length = (((num_input+1) * hiddenlayer_size) +
       ((hiddenlayer_size + 1) * num_output));
}
```

2.2.2 Initialisation

This function initialises the arrays needed for the neural network, scales the genes and assigns the scaled values to the *weight* arrays. A more detailed explanation of this code can be found in chapter 3 section 3.1.2 on page 18.

Listing B.2: GA assisted Artificial Neural Network initialisation

```
void MyController::init ( const vector <chromosome_type> &genes ){
//   vector <double> new_gene;//vector to hold the scaled gene values
   double new_gene[genes.size()];
   /*initalize the layers*/
   for(int i = 0;i < num_input+1;i++){
      inputlayer[i] = 0.0;
   }

   for(int i = 0;i < num_output;i++){
      hiddenlayer[i] = 0.0;
   }

   for(int i = 0;i < hiddenlayer_size+1;i++){
      outputlayer[i] = 0.0;
   }
```

```
    /*Set new_genes to 0.0*/
    for(int i = 0;i < genes.size();i++){
        new_gene[i] = 0.0;
    }

    /*initialise the input-to-hiddenlayer weights*/
    for(int i = 0;i < num_input+1;i++){
        for(int j = 0;j < hiddenlayer_size;j++){
            weights1[i][j] = 0.0;
        }
    }
    /*initialise the hidden-to-outputlayer weights*/
    for(int m = 0;m < hiddenlayer_size+1;m++){
        for(int n = 0;n < num_output;n++){
            weights2[m][n] = 0.0;
        }
    }

    /*scale the genes from -5 to 5*/
    for(int i = 0;i < genes.size();i++){
        new_gene[i] = ((high_bound - low_bound) * get_value(genes, i)) +
             low_bound;
    }

    /*set the weights*/
    int counter = 0;
    for(int i = 0;i < num_input+1;i++){
        for(int j = 0;j < hiddenlayer_size;j++){
            weights1[i][j] = new_gene[counter];
            counter++;
        }
    }
    for(int m = 0;m < hiddenlayer_size+1;m++){
        for(int n = 0;n < num_output;n++){
            weights2[m][n] = new_gene[counter];
            counter++;
        }
    }
}
```

2.2.3 Step function

In this function the neural network calculates the outputs based on its inputs, the evolved weights, as well as the calculations done in the hidden and output layer.
More details on this function can be found in chapter 3 section 3.1.3 on page 19.

Listing B.3: GA assisted Artificial Neural Network step function

```
void MyController::step ( const vector <double> &input, vector
    <double> &output){
    /*set values for the inputlayer*/
    for(int i = 0;i < input.size();i++){
```

```
        inputlayer[i] = get_value(input, i);
    }

    /*set Bias for the hiddenlayer*/
    inputlayer[num_input] = 1.0;

    /*reset the outputlayer*/
    for(int i = 0;i < num_output;i++){
        outputlayer[i] = 0.0;
    }

    /*add the weights from input to hiddenlayer*/
    for(int i = 0;i < hiddenlayer_size;i++){
        for(int j = 0;j < num_input+1;j++){
            hiddenlayer[i] +=(inputlayer[j] * (weights1[j][i]));
        }
    }

    /*calculate the sigmoid for the hiddenlayer*/
    for(int i = 0;i < hiddenlayer_size;i++){
        hiddenlayer[i] = f_sigmoid(hiddenlayer[i]);
    }

    /*add the bias for the outputlayer*/
    hiddenlayer[hiddenlayer_size] = 1.0;

    /*add the weights from the hidden-to-outputlayer*/
    for(int i = 0;i < num_output;i++){
        for(int j = 0;j < hiddenlayer_size+1;j++){
            outputlayer[i] +=(hiddenlayer[j] * (weights2[j][i]));
        }
    }

    /*calculate the sigmoid for the outputlayer*/
    for(int i = 0;i < num_output;i++){
        output[i] = f_sigmoid(outputlayer[i]);
    }
}
```

2.3 Test Environment for first fitness function

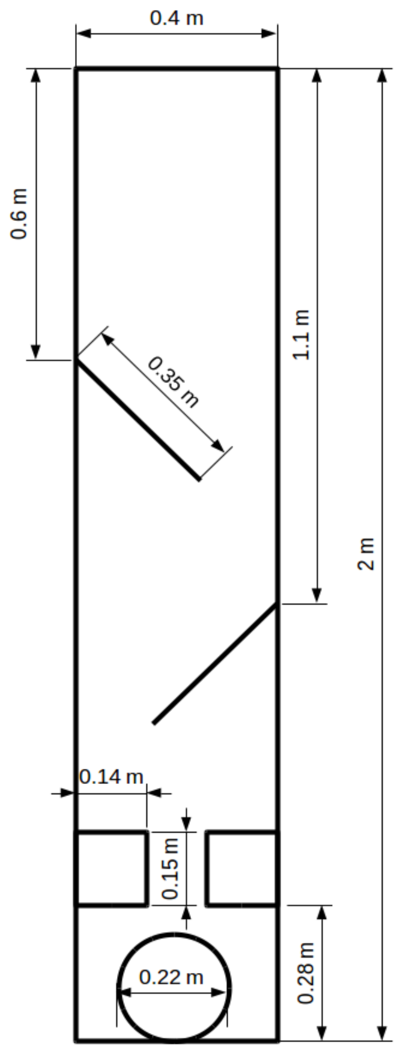

Figure B.2: Representation of the test environment for the first fitness function

2.4 Map

2.4.1 Initialisation

In this function the map is initialised.
All cells in the map are set to 0 and returns a pointer to a pointer to the array. Further explanation the code can be found in chapter 3 section 3.4.1 on page 21.

Listing B.4: Map initialisation

```
int** Occupancy_Map::init() {
   map = new int *[map_height]; //initialise array of pointers
   for(int i = 0;i < map_height;i++){ //add array pointers to the array
      map[i] = new int[map_width];
   }

   for(int i = 0;i < map_height;i++){
      for(int j = 0;j < map_width;j++){
         map[i][j] = 0;
      }
   }

   return map;
}
```

2.4.2 Calculate Heading

Listing B.5: Calculate the robots heading

```
/* 0: north, 1:east, 2:south, 3:west, 4:northeast, 5:southeast,
    6:southwest, 7:northwest
 * +/- 10 spread for main headings(north, south, east, west)
 * +/- 5 spread for secondary headings(northeast, southeast, etc)*/
int Occupancy_Map::calc_heading(double rotation) {
   int heading;
   if(rotation <= 10 && rotation >= -10){
      heading = 1;
   }
   else if(rotation <= 100 && rotation >= 80){
      heading = 0;
   }
   else if(rotation >= -170 && rotation >= 170){
      heading = 3;
   }
   else if(rotation <= -80 && rotation >= -100){
      heading = 2;
   }
   else if(rotation <= 50 && rotation >= 40){
      heading = 4;
   }
   else if(rotation <= -40 && rotation >= -50){
      heading = 5;
   }
   else if(rotation <= - 130 && rotation >= -140){
      heading = 6;
   }
   else if(rotation <= 140 && rotation >= 130){
      heading = 7;
   }
   else{
      heading = -1;
   }
```

```
    return heading;
}
```

2.4.3 Calculation of robot position on the map

This function calculates robots coordinates on the map.
Further explanation of this function can be found in chapter 3 section 3.4.2 on page 21.

Listing B.6: Calculate the robot position on the map

```
int* Occupancy_Map::calc_robot_pos(double x_coord, double y_coord){
    int matrix_x = 0;
    int matrix_y = 0;
    int *array = new int[2];
    double integral, fractal ;

    if(x_coord > 0.0 ){
        x_coord = x_coord * 1000;
        fractal = modf(x_coord, &integral);
        if(fractal < 0.5){
            x_coord = integral;
        }
        else if(fractal > 0.5){
            x_coord = integral + 1;
        }
        matrix_x = (map_width/2) + x_coord;
        matrix_y = y_coord * 1000;

    }else if(x_coord < 0.0){
        x_coord -= x_coord*2;
        x_coord = x_coord * 1000;

        fractal = modf(x_coord, &integral);
        if(fractal < 0.5){
            x_coord = integral;
        }
        else if(fractal > 0.5){
            x_coord = integral + 1;
        }


        matrix_x = (map_width/2) - x_coord;
        matrix_y = y_coord * 1000;
    }

    array[0] = matrix_x;
    array[1] = matrix_y;

    return array;
}
```

2.4.4 Return the correct Sensor number

This function is used to return the correct sensor number. The reason this function was needed was because in the array returned by the IR reading function the cells of the array do not correspond to the same sensor number.
More information on this method can be found in chapter 3 section 3.4.4.1 on page 23

Listing B.7: Calculate sensor number

```
int Occupancy_Map::calc_sensor(int array_num) {
   int sensor_num;

   if(array_num == 0){
      sensor_num = 0;
   }
   else if(array_num == 1){
      sensor_num == 7;
   }
   else if(array_num == 2){
      sensor_num == 1;
   }
   else if(array_num == 3){
      sensor_num == 6;
   }
   else if(array_num == 4){
      sensor_num == 2;
   }
   else if(array_num == 5){
      sensor_num == 5;
   }
   else if(array_num == 6){
      sensor_num == 3;
   }
   else if(array_num == 7){
      sensor_num == 4;
   }

   return sensor_num;
}
```

2.4.5 Decide which cells to mark

This function is used to analyse which mapping function to call based on the activated sensor number.
Further analysis of this function can be found in chapter 3 section 3.4.4.2 on page 23.

Listing B.8: calculate which cells to mark

```
void Occupancy_Map::calc_matrix_values(vector <double> &ir_reading,
    int heading, int robot_x, int robot_y, int **matrix){
   double sensor_value;
   int sensor_num;
```

```
    for(int i = 0;i < ir_reading.size();i++){
        sensor_value = ir_reading[i];
        if(sensor_value != -1){
            sensor_num = calc_sensor(i);
            if(heading == 0 || heading == 1 || heading == 2 || heading ==
                3){
                if(sensor_num == 0 || sensor_num == 7){
                    set_front_cells(heading, sensor_num, robot_x, robot_y,
                        matrix);
                }
                else if(sensor_num == 6 || sensor_num == 1){
                    set_front_side_cells(heading, sensor_num, robot_x,
                        robot_y, matrix);
                }
                else if(sensor_num == 5 || sensor_num == 2){
                    set_side_cells(heading, sensor_num, robot_x, robot_y,
                        matrix);
                }
                else if(sensor_num == 3 || sensor_num == 4){
                    set_aft_cells(heading, sensor_num, robot_x, robot_y,
                        matrix);
                }
            }else if(heading == 4 || heading == 5 || heading == 6 ||
                heading == 7){
                set_angeld_cells(heading, sensor_num, robot_x, robot_y,
                    matrix);
            }
        }
    }
}
```

2.4.6 Map cells in the direct front of the robot

The robot

Listing B.9: Code to set cells in front of the robot

```
/*Sensors set on a 15 degree angle to the front of the robot. Sensor 7
    and 0 on the robot*/
void Occupancy_Map::set_front_cells(int heading, int sensor, int
    robot_x, int robot_y, int **matrix) {
    if(heading == 0){
        mark_cell(robot_x, robot_y-1, 1, matrix);
    }
    else if(heading == 1){
        mark_cell(robot_x+1, robot_y, 1, matrix);
    }
    else if(heading == 2){
        mark_cell(robot_x, robot_y+1, 1, matrix);
    }
    else if(heading == 3){
        mark_cell(robot_x-1, robot_y, 1, matrix);
```

```
    }
}
```

2.4.7 Map cells off the bow of the robot

Listing B.10: Code to set cells off the bow of the robot

```
/*Sensors placed in a 45 degree angle on the front of the robot.
    sensor 1 and 6 on the robot*/
void Occupancy_Map::set_front_side_cells(int heading, int sensor, int
    robot_x, int robot_y, int **matrix) {
   if(sensor == 1){
      if(heading == 0){
         mark_cell(robot_x+1, robot_y-1, 1, matrix);
      }
      else if(heading == 1){
         mark_cell(robot_x+1, robot_y+1, 1, matrix);
      }
      else if(heading == 2){
         mark_cell(robot_x-1, robot_y+1, 1, matrix);
      }
      else if(heading == 3){
         mark_cell(robot_x-1, robot_y-1, 1, matrix);
      }
   }
   else if(sensor == 6){
      if(heading == 0){
         mark_cell(robot_x-1, robot_y-1, 1, matrix);
      }
      else if(heading == 1){
         mark_cell(robot_x+1, robot_y-1, 1, matrix);
      }
      else if(heading == 2){
         mark_cell(robot_x+1, robot_y+1, 1, matrix);
      }
      else if(heading == 3){
         mark_cell(robot_x-1, robot_y+1, 1, matrix);
      }
   }
}
```

2.4.8 Map cells to the side of the robot

Listing B.11: Code to set cells of the sides of the robot

```
/*Side sensors placed at a 90 degree angle. 2 and 5 on the epuck */
void Occupancy_Map:: set_side_cells(int heading, int sensor, int
    robot_x, int robot_y, int **matrix) {
     if(sensor == 2){
       if(heading == 0){
```

```
            mark_cell(robot_x+1, robot_y, 1, matrix);
         }
         else if(heading == 1){
            mark_cell(robot_x, robot_y+1, 1, matrix);
         }
         else if(heading == 2){
            mark_cell(robot_x-1, robot_y, 1, matrix);
         }
         else if(heading == 3){
            mark_cell(robot_x, robot_y-1, 1, matrix);
         }
      }
      else if(sensor == 5){
         if(heading == 0){
            mark_cell(robot_x-1, robot_y, 1, matrix);
         }
         else if(heading == 1){
            mark_cell(robot_x, robot_y-1, 1, matrix);
         }
         else if(heading == 2){
            mark_cell(robot_x+1, robot_y, 1, matrix);
         }
         else if(heading == 3){
            mark_cell(robot_x, robot_y+1, 1, matrix);
         }
      }
}
```

2.4.9 Set Aft cells of the robot

Listing B.12: Code to set cells to the aft of the robot

```
/*Sensors placed at a 25 degree angle to the back of the robot.
    Sensors 3 and 4 on the epuck*/
void Occupancy_Map::set_aft_cells(int heading, int sensor, int
    robot_x, int robot_y, int **matrix) {
   if(heading == 0){
      mark_cell(robot_x, robot_y+1, 1, matrix);
   }
   else if(heading == 1){
      mark_cell(robot_x-1, robot_y, 1, matrix);
   }
   else if(heading == 2){
      mark_cell(robot_x, robot_y-1, 1, matrix);
   }
   else if(heading == 3){
      mark_cell(robot_x+1, robot_y, 1, matrix);
   }
}
```

2.5 SDL program for map visualisation

This program is used to draw the map data on the screen.
For more information see chapter 3 section 3.5 on page 24.

Listing B.13: SDL program for map visualisation

```
#include <SDL2/SDL.h>
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <vector>

#define map_height 2500
#define map_width 2500

SDL_Rect newSDL_Rect(int xs, int ys, int widths, int heights) {
   SDL_Rect rectangular;
   rectangular.x = xs;
   rectangular.y = ys;
   rectangular.w = widths;
   rectangular.h = heights;
   return rectangular;
}

int main(int argc, char* args[]) {
   SDL_Window* window = NULL;
   SDL_Surface* surface = NULL;

   int y = 0;
   int x = 0;

   std::vector<int> y_value;
   std::vector<int> x_value;

   std::ifstream in;
   in.open("map.txt");
   while(in >> y >> x){

      y_value.push_back(y);
      x_value.push_back(x);
   }


   if (SDL_Init(SDL_INIT_VIDEO) < 0) //Init the video driver
   {
      printf("SDL_Error: %s\n", SDL_GetError());
   }
   else
   {
      window = SDL_CreateWindow("SDL 2", SDL_WINDOWPOS_UNDEFINED,
          SDL_WINDOWPOS_UNDEFINED, 640,420, SDL_WINDOW_SHOWN);
          //Creates the window
      if (window == NULL)
```

```
        {
           printf("SDL_Error: %s\n", SDL_GetError());
        }
        else
        {
           SDL_Renderer* renderer = NULL;
           renderer = SDL_CreateRenderer(window, 0,
               SDL_RENDERER_ACCELERATED); //renderer used to color rects

           SDL_SetRenderDrawColor(renderer, 51, 102, 153, 255);
           SDL_RenderClear(renderer);

           SDL_Rect rects[y_value.size()];
           printf("Y: %lu\n X: %lu\n",y_value.size(), x_value.size());
           for (int i = 0; i < y_value.size(); i++){
              int x = 0;
              int y = 0;

              rects[i] = newSDL_Rect(y_value[i]/5, x_value[i]/5, 1, 1);
              SDL_SetRenderDrawColor(renderer, 255, 102, 0, 255);

              SDL_RenderFillRect(renderer, &rects[i]);
           }
           SDL_RenderPresent(renderer);
           SDL_UpdateWindowSurface(window);
           SDL_Delay(15000);
        }
    }
    SDL_DestroyWindow(window);
    SDL_Quit();
    return 0;
}
```

2.6 UML

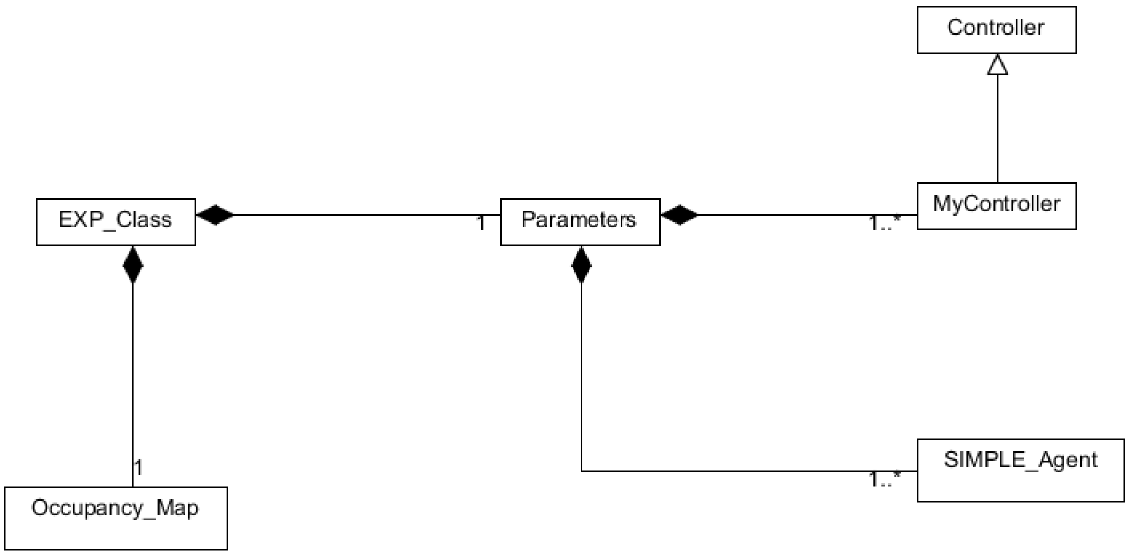

Figure B.3: A simplified diagram that shows the relationships between the classes

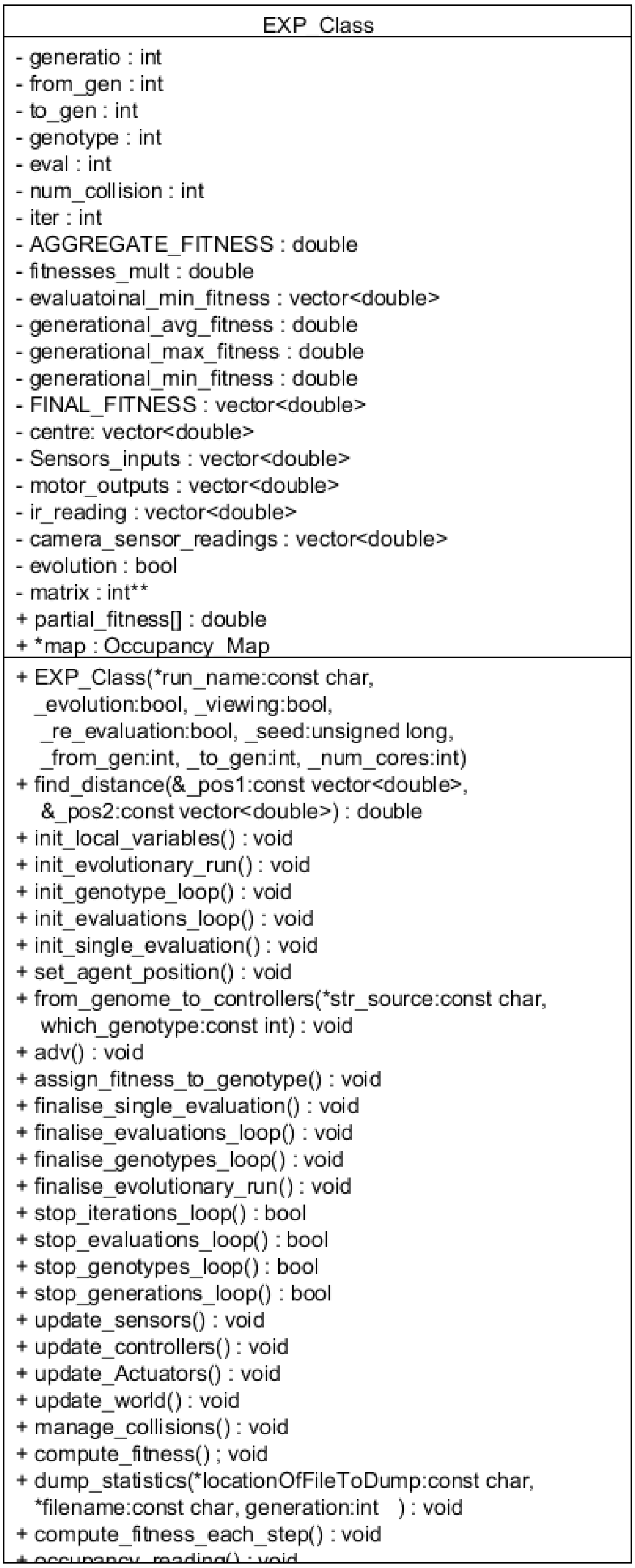

Figure B.4: UML diagram for the experiment class

Figure B.5: UML diagram for the parameters class

Figure B.6: UML diagram for the SIMPLE_Agent class

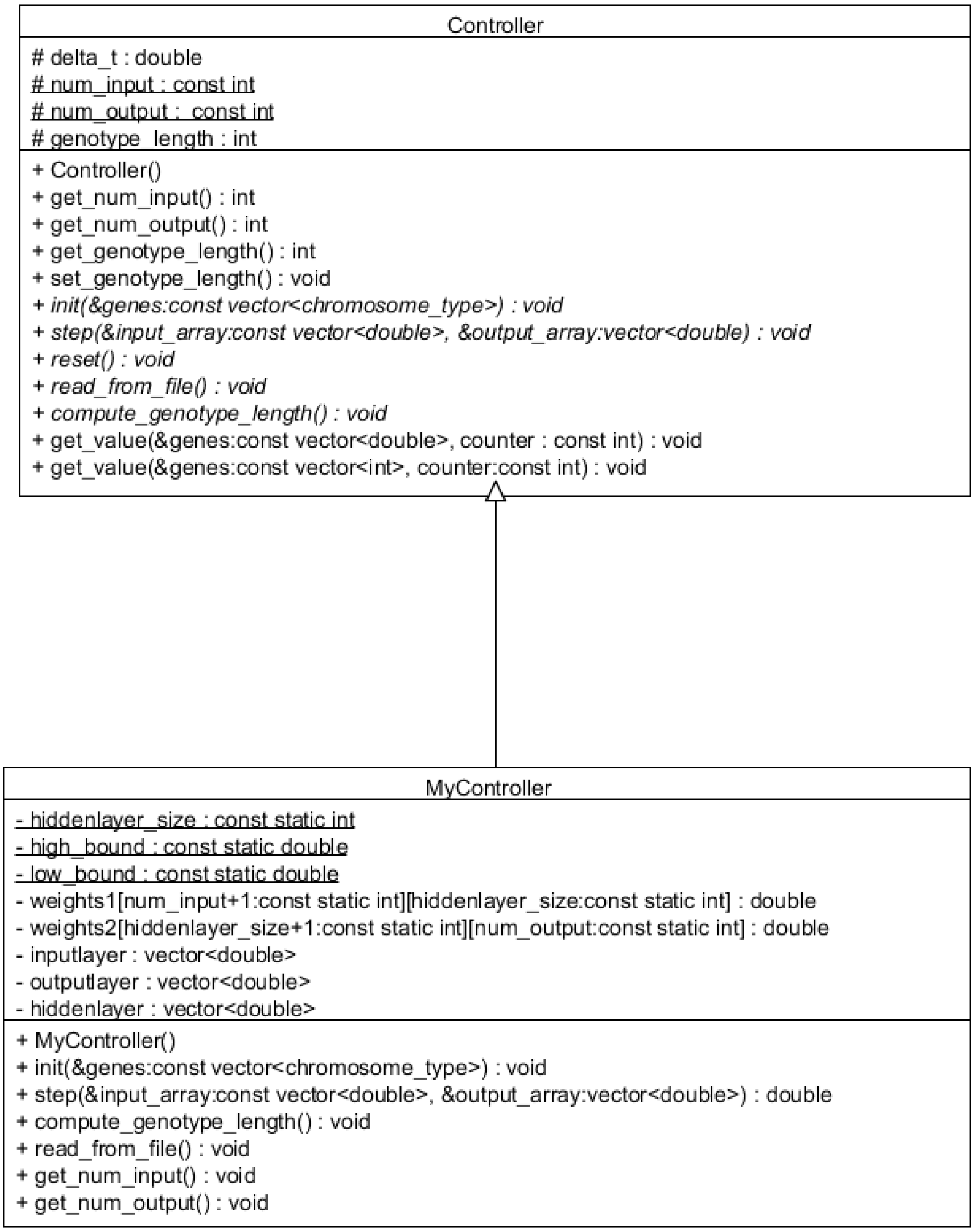

Figure B.7: UML diagram of the MyController class and its superclass

Annotated Bibliography

[1] A. Birk and S. Carpin, “Merging Occupancy Grid Maps From Multiple Robots,” *Proceedings of the IEEE*, vol. 94, no. 7, pp. 1384–1397, July 2006. [Online]. Available: http://dx.doi.org/10.1109/jproc.2006.876965

[2] C. Cianci, X. Raemy, J. Pugh, and A. Martinoli, “Communication in a Swarm of Miniature Robots: The e-Puck as an Educational Tool for Swarm Robotics,” in *Swarm Robotics*, ser. Lecture Notes in Computer Science, E. Şahin, W. Spears, and A. Winfield, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, vol. 4433, ch. 7, pp. 103–115. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-71541-2_7

[3] C. Costanzo, V. Loscr, E. Natalizio, and T. Razafindralambo, “Nodes self-deployment for coverage maximization in mobile robot networks using an evolving neural network,” *Computer Communications*, vol. 35, no. 9, pp. 1047 – 1055, 2012, special Issue: Wireless Sensor and Robot Networks: Algorithms and Experiments. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S014036641100274X

[4] D. Floreano and F. Mondada, “Evolution of homing navigation in a real mobile robot,” *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 26, no. 3, pp. 396–407, Jun 1996.

[5] D. E. Goldberg and K. Deb, “A comparative analysis of selection schemes used in genetic algorithms,” *Foundations of genetic algorithms*, vol. 1, pp. 69–93, 1991.

[6] A. Gutiérrez, A. Campo, M. Dorigo, D. Amor, L. Magdalena, and F. Monasterio-Huelin, “An Open Localization and Local Communication Embodied Sensor,” *Sensors*, vol. 8, no. 11, pp. 7545–7563, Nov. 2008. [Online]. Available: http://dx.doi.org/10.3390/s8117545

[7] A. Gutiérrez, A. Campo, M. Dorigo, D. Amor, L. Magdalena, and F. Monasterio-Huelin, “An open localization and local communication embodied sensor,” *Sensors*, vol. 8, no. 11, pp. 7545–7563, 2008.

[8] J. Hopfield, “Artificial neural networks,” *Circuits and Devices Magazine, IEEE*, vol. 4, no. 5, pp. 3–10, Sept 1988.

[9] R. Kala, A. Shukla, and R. Tiwari, “Robotic path planning using evolutionary momentum-based exploration,” *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 23, no. 4, pp. 469–495, July 2011. [Online]. Available: http://dx.doi.org/10.1080/0952813x.2010.490963

[10] Q. Meng and M. H. Lee, “Active Exploration in Building Hierarchical Neural Networks for Robotics,” in *Instrumentation and Measurement Technology Conference, 2006. IMTC*

2006. Proceedings of the IEEE. IEEE, Apr. 2006, pp. 2095–2100. [Online]. Available: http://dx.doi.org/10.1109/imtc.2006.328464

[11] F. Mondada, M. Bonani, X. Raemy, J. Pugh, C. Cianci, A. Klaptocz, S. Magnenat, J.-C. Zufferey, D. Floreano, and A. Martinoli, "The e-puck, a robot designed for education in engineering," in *Proceedings of the 9th conference on autonomous robot systems and competitions*, vol. 1, no. LIS-CONF-2009-004. IPCB: Instituto Politécnico de Castelo Branco, 2009, pp. 59–65.

[12] A. L. Nelson, G. J. Barlow, and L. Doitsidis, "Fitness functions in evolutionary robotics: A survey and analysis," *Robotics and Autonomous Systems*, vol. 57, no. 4, pp. 345–370, Apr. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.robot.2008.09.009

[13] S. Nolfi and D. Floreano, *Evolutionary robotics: The biology, intelligence, and technology of self-organizing machines*. MIT press, 2000.

[14] L. Silva, M. Tronco, H. Vian, G. Pellinson, and A. Porto, "Environment mapping for mobile robots navigation using hierarchical neural network and omnivision," in *Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*, June 2008, pp. 3292–3297.

[15] S. Thrun, "Learning Occupancy Grid Maps with Forward Sensor Models," *Autonomous Robots*, vol. 15, no. 2, pp. 111–127, Sept. 2003. [Online]. Available: http://dx.doi.org/10.1023/a:1025584807625

[16] V. Trianni and M. Lpez-Ibez, "Advantages of task-specific multi-objective optimisation in evolutionary robotics," *PLoS ONE*, vol. 10, no. 8, p. e0136406, 08 2015. [Online]. Available: http://dx.doi.org/10.1371%2Fjournal.pone.0136406

[17] K. M. Wurm, A. Hornung, M. Bennewitz, C. Stachniss, and W. Burgard, "OctoMap: A probabilistic, flexible, and compact 3D map representation for robotic systems," in *Proc. of the ICRA 2010 workshop on best practice in 3D perception and modeling for mobile manipulation*, vol. 2, 2010.

[18] J. Zhong, X. Hu, M. Gu, and J. Zhang, "Comparison of performance between different selection strategies on simple genetic algorithms," in *Computational Intelligence for Modelling, Control and Automation, 2005 and International Conference on Intelligent Agents, Web Technologies and Internet Commerce, International Conference on*, vol. 2, Nov 2005, pp. 1115–1121.