Development of a Simple Swarm Behaviour Simulator and Development of a Reinforcement Learning AI System for Swarm Members

Simon Alger

October 23, 2012

Abstract

The behaviour of groups consisting of individual agents capable of making decisions is affected by a complex balance between actions that directly benefit the individual, and those that benefit the group at large. The effect of how an individual interacts with (and interprets interactions from) its peers can be simulated using a robotic "swarm" of individuals. The purpose of this project is to develop a computer simulation to model swarm behaviour. The project is focused on both modelling of swarms, and swarm learning.

A computer program capable of performing simulation of swarm behaviour in mobile robotic agents was developed. These agents were able to move around in a simulated environment and interact with the environment and their peers. Following the creation of the simulator, the simulated agents were given simple pre-set rules to follow, in order to replicate swarm behaviour observed in nature. It was seen that these simple rules, when implemented in a global context, led to complex emergent behaviours. The swarm behaviours observed included clustering and chain formation.

After swarm behaviours had been successfully modelled, a reinforcement learning algorithm was developed to create simple AI behaviour in individual swarm members. The task assigned to agents was that of learning to explore a given environment as quickly as possible, all the while avoiding collisions with obstacles and other agents. The agents were not given any guidance on how to achieve this task, but were free to learn through trial and error. Although initially colliding with obstacles in the process of learning, the agents were eventually able to teach themselves how to predict and avoid collisions.

The final part of the project was the creation of a distributed resource allocation scenario, where agents equipped to extinguish fires were required to explore an environment and extinguish any fires they encountered. The simulations demonstrated the need for inter-agent cooperation in cases where single robots were not capable of extinguishing large fires. Two different cooperative approaches were analysed and compared, with the possibility of using AI to elicit self-organised cooperation being explored.

Plagiarism Declaration

I, Simon Roy Alger, declare that

1. I know that plagiarism is wrong. Plagiarism is to use another's work and to pretend that it is one's own.
2. I have used the modified IEEE convention for citation and referencing. Each significant contribution to, and quotation in, this project from work(s) of other people has been attributed, cited and referenced.
3. This project is my own work.
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as his or her own.

Signed: ....

Date: ....

Acknowledgements

I would like to thank the following people for their assistance with this project.

My supervisor, **Ernesto Ismail**, without whose frequent advice, enthusiasm and constructive criticism this project would not exist in its current form.

Janine Alger, my mother, for her encouragement, patience and unwavering support.

Table of Contents

List of Tables

List of Figures

Nomenclature

α	Q-learning rate
β	rate of decrease of propensity to explore state-action space (T)
η	neural network learning rate
γ	Q-learning discount factor
κ	computational cost of physics engine
$\bar{\mathbf{x}}$	centroid of visible agents
$\mathbf{a}$	action (RL)
$\mathbf{C_a}$	agent position in global coordinates
$\mathbf{C_s}$	sensor position in global coordinates
$\mathbf{c_s}$	sensor position in agent local coordinates
$\mathbf{R}$	total return, i.e. accumulated reward (RL)
$\mathbf{s}$	environment state (RL)
$\mathbf{x_{NN,f}}$	nearest neighbour in the forward direction
ws	fractional wheel slip
ω	rotational velocity of agent
Θ	angle in global coordinates
θ	angle in agent's local coordinates
$\vec{m}$	apparent change in centroid position
$\vec{v}$	apparent centroid velocity
c	computational cost of agent control algorithm

H	temperature of fire-fighting agent
M	map complexity
m_L	left motor speed
m_R	right motor speed
n	number of agents
P	machine performance
Q	agent approximation of Q^*, i.e. utility
Q^*	true utility function
r	radius of agent body
r	reward received from environment (RL)
s	number of sensors per agent
T	fire temperature
T	propensity to explore state-action space
T_c	minimum T-value (propensity to explore action space)
v_L	non-slipping left wheel velocity
v_O	velocity of agent centre of mass
v_R	non-slipping right wheel velocity
X	agent generalized coordinates

Chapter 1

Executive Summary

Introduction

The purpose of this report is to document a project that was undertaken to build a computer simulation of swarm behaviour, as well as a reinforcement learning algorithm for swarm members.

The basic premise of swarm theory is that swarm members act both as individual agents and as a collective. In the most general sense, an agent can be considered as any entity capable of making decisions based on observed phenomena. The most common types of agency encountered everyday are in the natural world: other people, animals and all living matter down to the simple gene. However, computer programs can also be considered agents, because they have the ability to make decisions based on observation. One of the goals of this project is to explore the way in which agents interact on a large scale. When a large number of such agents find themselves in an environment together, complex inter-dependencies and network effects can emerge. This project seeks to model and study those interactions.

The agents modelled in this project are small, autonomous, mobile robots, as shown in Figure 1.1 on the following page. The robots are equipped with two wheels and a number of infrared proximity sensors. A key characteristic of these robots is that they are not capable of communicating between themselves. This creates interesting dynamics which are explored in this report.

The specific goals of the project were to:

1. Design, program and test a simulator capable of modelling individual robotic agent behaviour, as well as large swarms of robotic agents (up to 50 agents).

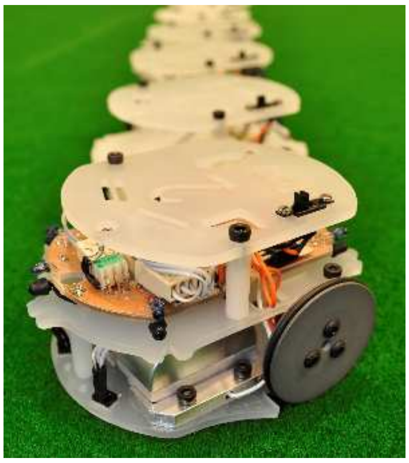


Figure 1.1: Autonomous mobile robot.

2. Simulate swarm behaviour by giving agents simple rules to follow.
3. Program a reinforcement learning algorithm that is able to interact with the simulator and produce AI behaviour in each simulated agent.

Robotic Agent Simulator

In order to model the swarm behaviour in mobile robots, it was necessary to program a computer simulator. Research was undertaken into existing best practices in this field, using which information a simulator was created. A complete physical model of the simulated environment was created, and algorithms designed to handle collisions, robot locomotion and operation of simulated sensors.

To assist and expedite future work with the simulator, a number of additional features were added, beyond the initial specification. The most important of these features was to separate the simulation and visualisation operations, so that large computationally-intensive simulations (which would run too slowly to observe in real-time) could be executed in the background, with visualisation taking place at the user's convenience and at a speed not affected by the complexity of the simulation. Figure 1.2 on the next page shows the basic simulated robot developed and used throughout the project, and a typical simulated environment with a number of dispersed robotic agents. Further, a key design characteristic of the simulator is its extensibility: that is, new robot configurations or sensor types can easily be created, allowing simulation of a wide range of hardware.

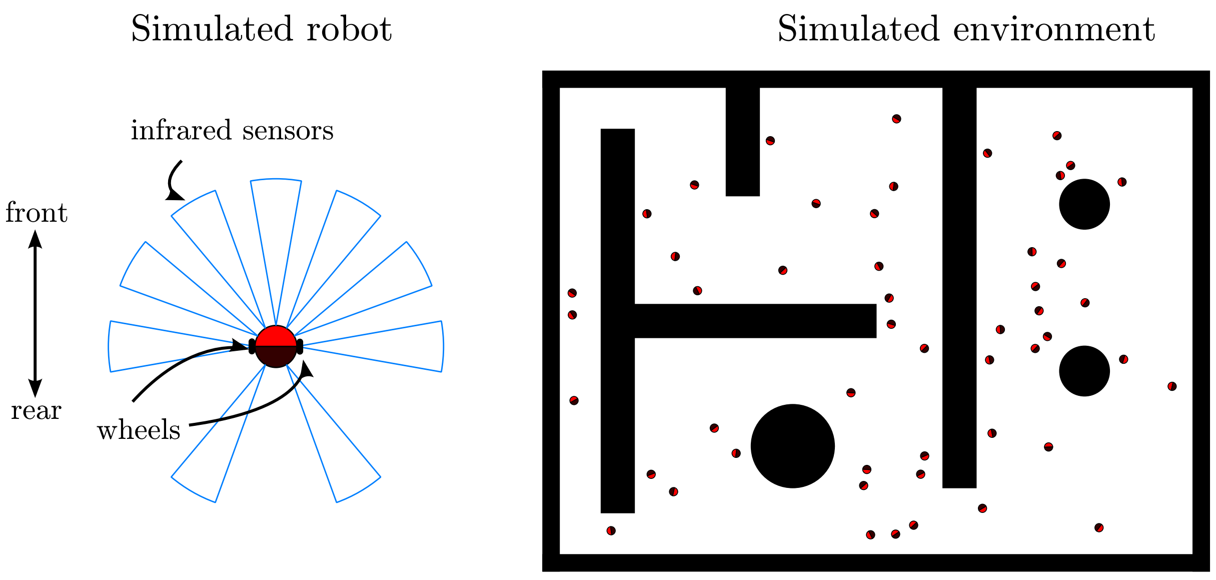


Figure 1.2: Representation of a simulated robot and a typical simulated environment.

Simple Swarm Behaviour

Once the basic simulator had been created, attention was turned to replicating swarm behaviour. In natural swarms, individual agents follow basic rules such as "steer towards the group centroid" to cause agglomeration, or "match the velocity of your observable peers" to form long chains. The simulator was used to give each simulated robotic agent simple instructions such as these, with the result being complex emergent behaviours.

Three such rule-based approaches were implemented: exploration with collision avoidance, agglomeration and chain formation. Results of these simulations are shown in Figure 1.3 on the following page.

Some of the emergent behaviours observed were completely unexpected. For example, in the chain formation simulations, chains were observed to decompose, merge with other chains and reorganise after encountering obstacles. The occurrence of these unexpected behaviours highlight the utility of a swarm simulator: it is often difficult or even impossible to predict the effect of even the simplest rules when they are applied in a global context. Examples of this phenomenon may be found in many domains from economics to the formation of mobs. In many respects, the behaviours observed closely follow those found in nature, for example the way in which a school of fish can grow, shrink and reorganise.

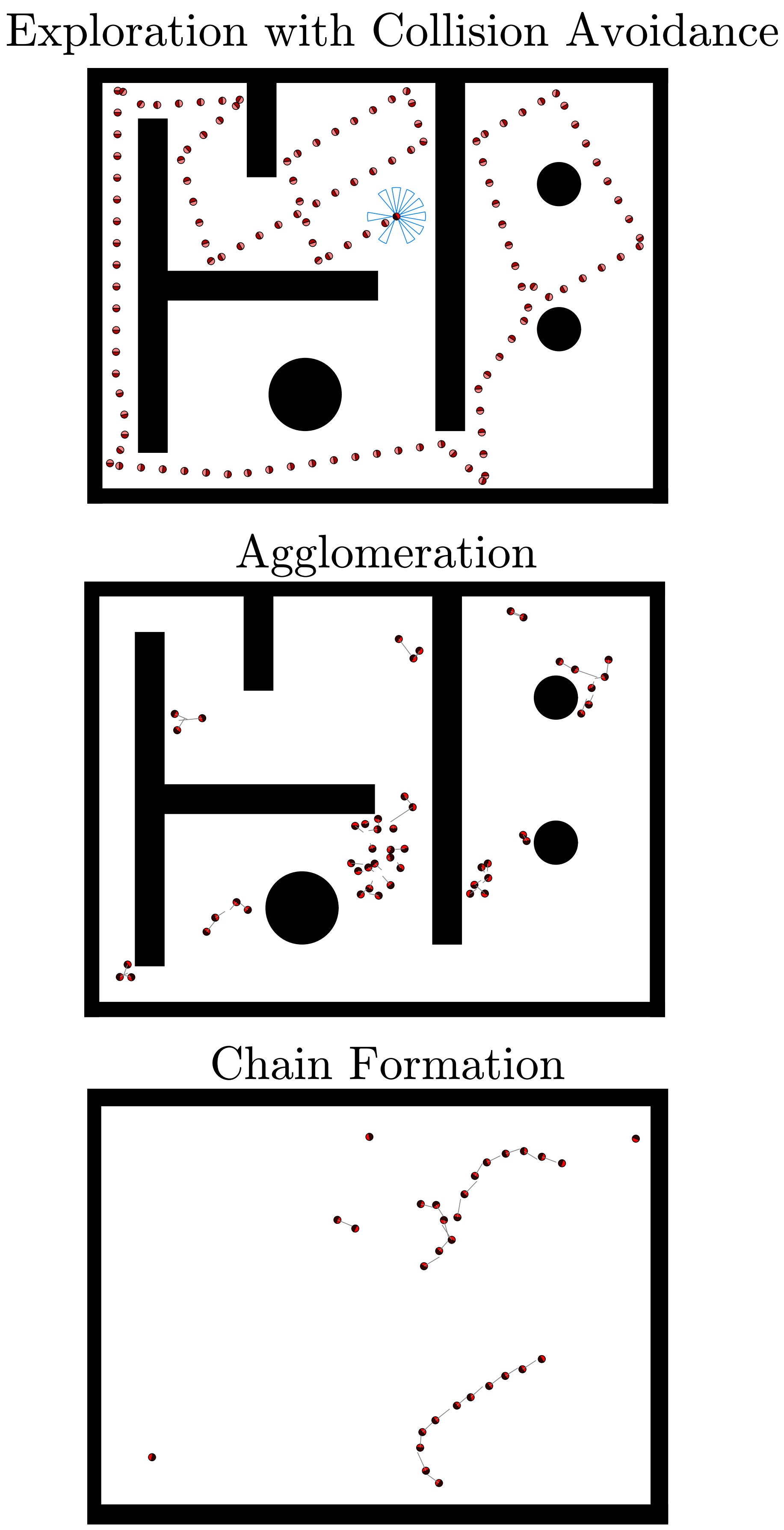


Figure 1.3: Simulation of simple swarm behaviours.

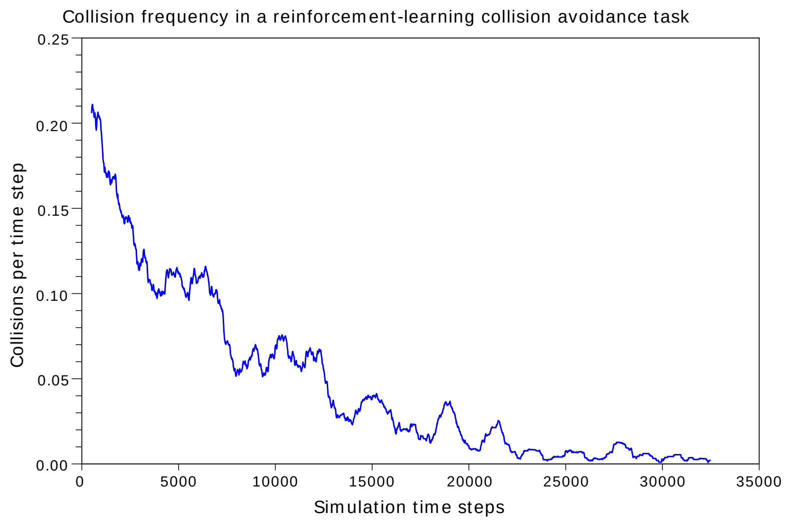


Figure 1.4: Frequency of collisions between agents and obstacles as agents learn to avoid collisions.

AI Behaviour

Following the simulation of swarm behaviour, a reinforcement learning algorithm was written to produce simple AI behaviour in each swarm member. The task assigned to agents was that of learning to explore a given environment as quickly as possible, all the while avoiding collisions with obstacles and other agents. The distinguishing characteristic of the AI approach is that agents were not given any guidance on how to achieve this task, but were free to learn through trial and error. Agents were rewarded if they explored the map in a straight line at high speed, and punished (assigned negative reward) if they approached obstacles too closely. Although initially colliding with obstacles frequently while in the process of learning, the agents were eventually able to teach themselves how to predict and avoid collisions. A plot of the frequency of collisions versus simulation time is shown in Figure 1.4.

In many respects, the collision avoidance behaviour that agents were able to teach themselves was very similar to the rule-based approach that had previously been implemented. Once agents had spent sufficient time learning, they were able to avoid collisions almost permanently, as reflected in the graph. However, the method by which they avoided head-on collisions was observed to be quite different from the rule-based approach that had previously been implemented. When collision avoidance was programmed into behaviour-based agents, the rule given to them was to turn away from any obstacles detected by their lateral sensors, and choose a random direction of turn when an obstacle was directly ahead. In contrast, when an AI agent was heading straight for an obstacle, it was sometimes observed to first move directly backwards and then turn a small amount and continue moving in a different direction.

The emergence of this unanticipated method of achieving a task, which a human programmer can easily prescribe using a rule-based approach, is an indication of the importance of machine learning in situations involving large numbers of individual agents. In particular, a strong case can be made for the use of artificial intelligence techniques in simulated agents to model real-life swarm behaviour, such as the actions of investors during a stock crash or the flocking of birds to reduce aerodynamic drag.

Fire-fighting Task

The final part of the project was the creation of a distributed resource allocation scenario, where agents equipped to extinguish fires were required to explore an environment and extinguish any fires they encountered. The simulations demonstrated the need for inter-agent cooperation in cases where single robots were not capable of extinguishing large fires.

The basic model was an environment (such as a factory or warehouse) with a number of distinct locations to which a fire had spread. These could represent combustible materials such as chemical stores or any other flammable material. Robotic agents deployed in such an environment are required to explore the map as quickly as possible, while extinguishing any fires they encounter. However, when extinguishing a fire, a robot can suffer component damage from excessive heat. Therefore, a robot is required to leave a fire if its temperature should become too high. It then needs to spend a period of time 'cooling down' before returning to fire-fighting duty. Because some fires will be too large for a single robot to extinguish alone, the only way of achieving the group goal of eliminating all fires is by working cooperatively. However, each agent is also concerned with preservation of its own hardware, resulting in intricate group dynamics. A number of still frames depicting cooperative behaviour are shown in Figure 1.5 on the following page.

Two different cooperative approaches were analysed and compared. The first such approach was for the simulated robots to leave a fire and look for another, possibly smaller fire if they were at first unsuccessful at extinguishing it. The second approach was for robots to try extinguish the same fire until it had been completely extinguished. While the simulation results that were obtained indicated the second approach would be favourable under certain circumstances, the significance of the fire-fighting scenario is not in this specific result, but rather in the fact that the scenario creates an environment favourable to the formation of swarms. As this task was beyond the requirements of the project, there

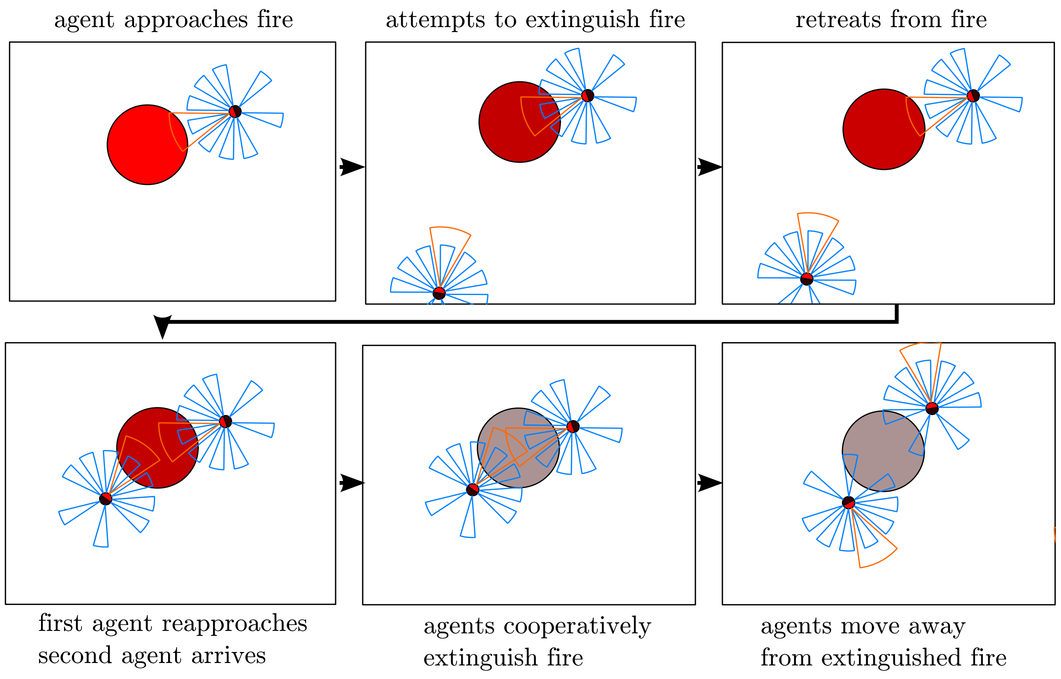


Figure 1.5: A series of frames depicting cooperative fire-fighting behaviour.

was not sufficient time to implement AI behaviour in the fire-fighting agents. However, it is believed that programming such AI behaviour into the agents could unveil unexpected techniques for solving the problem, perhaps even with direct practical applications.

Conclusions and Recommendations

The phenomenon of swarm behaviour is somewhat remarkable: a large number of individual agents, each acting in the interest of their own self-preservation, self-organising into swarms that are able to cooperatively realise individual and group goals.

It has been shown how simple rules, when implemented by a number of autonomous agents on a large scale, can lead to the emergence of behaviour that is, at worst, unexpected, and at best, provides novel insights into better ways of achieving group-based outcomes. The use of artificial intelligence in creating agents capable of learning based on feedback from their environment has been explored, with the result that they are often able to determine better methods of task execution than human designers attempting to analytically decompose such tasks into individual behaviours.

A number of recommendations are made to improve the simulator and to further investigate the topics presented in this report. The most important of these are to optimise the simulator's contact model to increase simulation speed, investigate different learning approaches (such as evolutionary algorithms) and to implement a state-action space encoding capable of differentiating between agents and obstacles.

It is hoped that this report will leave the reader with a greater appreciation of the prevalence of swarm behaviour across multiple domains, as well as an understanding of the important role played by computer simulation in modelling these complex interactions.

Chapter 2

Introduction

The purpose of this report is to document a project that was undertaken to build a computer simulation of swarm behaviour, as well as a reinforcement learning algorithm for swarm members.

This chapter provides an introduction to the concepts of agency, learning and self-organisation, with an emphasis on the biological origins of these ideas. The application of these constructs to two classes of robotic models will be considered, followed by an adaptation of these ideas into a concrete problem formulation for the project. The chapter concludes with a plan of development for the remainder of the report.

2.1 Agency and Learning

In the most general sense, an agent can be considered as any entity capable of making decisions based on observed phenomena [1]. This definition of agency is usually extended to include the ability to adapt how decisions are made, based on past experiences. The most common types of agency encountered everyday are in the natural world: other people, animals and all living matter down to the simple gene.

However, the ability to make decisions is not limited exclusively to that which is living: computer programs can also be considered agents, because they have the ability to make decisions based on observation. That is not to say that all computer programs should be considered agents, however. Programs that follow a static, deterministic approach from start to finish, always producing the same output for a given output show no signs of agency. Therefore, the notions of agency and learning are strongly linked [2].

The goal of this project is to explore the way in which agents interact on a large scale. Just like a human baby learns and adapts to its environment as it grows older, so agents attempt to adapt to their environment for their own advantage. When a large number of such agents find themselves in an environment together, complex inter-dependencies and network effects can emerge. This project seeks to model and study those interactions.

2.2 Self-organisation

Perhaps the most interesting feature observed in large groups of biological agents is the phenomenon of *self-organization*, or the ability of autonomous agents to coordinate their efforts to effect some outcome only achievable by this coordination [1, 3]. This process can be witnessed almost anywhere in nature: the formation of birds into flocks to avoid predators and reduce aerodynamic drag; the cooperative hunting patterns of wolves or the cooperative transport mechanisms used by ants and termites [4].

The significance of these and other behaviours is that they are not centrally coordinated, but rather emerge from the interaction of local rules applied in a global context [1, 4]. These emergent behaviours are even more remarkable because the rules that produce them are often very simple, yet the behaviour that emerges can be extremely complex. In many cases, so simple are the rules that individual agents are not even aware that their actions are leading to a phenomenon of group self-organisation. For example, in ant colonies, worker ants form a protective boundary around the body of the queen by dropping items at various distances from her. The remarkable part about this behaviour is that the ants do not appear to be intentionally building a protective barrier around the queen. Instead, the queen exudes a particular pheromone which, when encountered above a certain concentration by worker ants, instinctively causes them to drop whatever they are carrying. This effect has been reproduced using the pheromone in the absence of an actual queen [4].

2.3 Mobile Robotic Agents

Following from the observation that self-organising groups in nature are composed of a large number of individually-acting biological agents, there has been a trend in robotics in the last two decades to create and study mobile robotic agents [1, 5, 6]. It is hoped that these agents will be able to interact and produce group dynamics similar to those observed in nature. A key facet of this approach is the use of artificial intelligence and machine learning, in particular, to allow these agents to discover unexpected ways of solving problems by working cooperatively.

Each individual mobile robot, therefore, can be viewed as an agent. In this case, the term ‘agent’ may be used interchangeably with ‘robot’. Indeed, this is the terminology used throughout most of this report. However, it need not be the case that the agent and physical robot boundaries coincide. A complex robot may make use of a number of internal agents operating cooperatively with each other, although to an external observer there may appear to be only one ostensible agent. This is the basis of humanoid robotics.

2.4 Humanoid Robotics

The notion that a single system may be composed of several internal agents working together forms the basis for many approaches in humanoid robotics [7]. Broadly, humanoid robots are those which physically resemble parts of the human body either in appearance, function, or both. Examples include robotic arms modelled on the human hand, and robots that perform complex anthropomorphic behaviours such as playing soccer.

The application of agency and learning to such robots is in the decomposition of the robot into several agents, which learn to work together to produce an effective outcome for the collective (which is the entire robot). For example, humanoid robots that are designed with articulated leg-like appendages are able to simulate legged locomotion by treating each joint as an individual agent, and having the agents interact cooperatively to ensure that robot moves forward without losing balance. Thus, on a conceptual level, these agents may be regarded as exhibiting swarm behaviour. Indeed, many of the techniques and algorithms used to simulate such behaviour are either identical or very similar to those used for swarming in mobile robots.

2.5 Problem Formulation

The initial objectives of this project were to create a swarm behaviour simulator and reinforcement learning system. These goals are sufficiently open-ended for their implementation to take on a variety of forms. As has been mentioned, there are a number of contexts in which the interaction of individual agents gives rise to swarm behaviour.

In the initial stages of the project, research was undertaken into the various forms the project could assume. Ideas such as simulation of legged locomotion and robotic arms were considered as broad interpretations of swarm behaviour in autonomous agents. It was ultimately decided, however, to proceed with the swarm behaviour simulator in the context of modelling mobile robotic agents.

2.5.1 Project Objectives

The overall objective of this project is to simulate swarm behaviour. In order to do this, a program is first required that can model the interactions between just a single agent and a simulated physical environment. Once such a basic platform has been created, it should be extended to allow for multiple-agent interactions. At that stage it is possible to begin giving the simulated agents pre-set behaviours on a situation-by-situation basis. These simple rules should allow the agents to exhibit swarm behaviour on the group scale. Once these rule-based behaviours have been implemented, an AI approach should be taken by creating a reinforcement learning algorithm to create learnt behaviours in each individual agent.

Concisely, the objectives of this project are stated as follows.

1. Design, program and test a simulator capable of modelling individual robotic agent behaviour, as well as large swarms of robotic agents (up to 50 agents).
2. Simulate swarm behaviour by giving agents simple rules to follow.
3. Program a reinforcement learning algorithm that is able to interact with the simulator and produce AI behaviour in each simulated agent.

2.6 Plan of Development

Following this introductory chapter is a short survey of the basic concepts of swarm behaviour to familiarise the reader with the topic, with attention given to natural and robotic swarms.

In Chapter 4, the physical simulator that was created for the project will be introduced and explained in-depth. The chapter begins by presenting a survey of existing simulators and best practices in the field, in order to inform and justify the design decisions that were made. Following this, the physical model that was used in the simulator will be explained. Attention will be then given to the details of the specific implementation, complete with comprehensive performance testing.

Chapter 5 will present the theory of behaviour-based control, with descriptions of the various behaviours that were implemented. The results of simulations using these behaviours are presented and analysed.

Chapters 6 and 7 will present the theory of neural networks and reinforcement learning, laying the groundwork for understanding the AI simulations that are to follow. These chapters are largely based on a review of relevant literature.

Chapter 8 will present the reinforcement learning algorithm that was developed during this project. The specifics of its implementation will be explained and related to their theoretical foundations in the foregoing chapters. The results of the corresponding simulations will then be presented and discussed.

The application of swarm behaviour to a unique distributed resource allocation problem will be described in Chapter 9. The results of simulations will again be presented and discussed.

Finally, a critical assessment of the project will be given in Chapter 10, reviewing the work that was performed and making recommendations for future work. The conclusion to the report follows in Chapter 11. Attached to the report are a number of appendices which provide further details not deemed appropriate for the main report.

The approach of presenting literature on all relevant topics in a single chapter was not taken. Instead, relevant literature has been grouped with the appropriate practical work and is presented alongside such work. In general, an attempt has been made to avoid forward references in the text. However, this is not always possible, and in such circumstances a brief description of the ulterior concept is given together with the reference.

Chapter 3

Swarm Behaviour

This chapter aims to give a brief, non-technical overview of various swarm behaviour concepts to serve as an introduction to the topic. It begins with background on swarms found in nature and proceeds to describe some of the behavioural patterns observed in biological swarms. Finally, the concept of robotic swarms is introduced, and three robotic swarm platforms are briefly surveyed.

3.1 Natural Swarms

Swarm behaviour is prevalent throughout nature. The phenomenon of individual biological agents forming groups can be seen in, inter alia, flocks of birds, schools of fish, herds of antelope, pods of dolphins and nest-building ants [4]. Perhaps the most remarkable aspect of such collective animal behaviour is that group cooperation can emerge purely as a result of the interaction of local rules applied in the group context. A number of possible explanations have been presented as to the function of these emergent behaviours, which will be briefly reviewed.

Protection from predators.

It has been proposed that antelope move in herds and fish swim in schools in order to reduce their chances of being attacked by predators [4]. A number of mechanisms may be responsible for this effect. The presence of a large group of rapidly moving individuals may cause a sensory overload for a potential predator. Large groups are able to identify predators faster and more consistently than an individual acting alone, and spread this information rapidly, thus protecting the entire group. Finally, the group itself may be stronger than the individual members acting alone, and thus pose a threat to potential predators [4].

Improved access to resources.

Studies on certain types of fish have found that groups of fish were able to find a patch of food faster than the same individuals acting separately. This can be generalised to resources in a broad sense: by acting in the group's interest and sharing information, it is possible for all individuals in the group to find resources faster than if acting alone [4].

Increased locomotion efficiency.

For animals moving in a fluid environment, there is a distinct advantage to moving in a group: reduction in drag force. Ducklings have been shown to save energy when swimming in a line [8], and reduction in aerodynamic drag could be partly responsible for the flocking behaviour of birds and shoaling of fish.

Social interaction.

Biological agents may also benefit from the social aspects of being near to other animals of the same species, including possible reproductive advantages [4].

3.2 Observed Behaviours

A number of behaviours and formations are observed in natural swarms. This section provides an inexhaustive account of some of the most prevalent phenomena.

3.2.1 Agglomeration

Agglomeration can be defined as the assembly of individual agents into a high-density group. While agglomeration generally refers to a single, large group, it can also include a greater number of smaller clusters. Figure 3.1 on the next page shows ants progressively assembling into a number of distinct groups.

3.2.2 Chain formation

Chain formation is a leader-following behaviour where agents move one behind the other. Leaders may be chosen explicitly, as in the case of ducklings following their mother, shown in Figure 3.2 on the following page, or the leader may emerge as a result of group interactions.

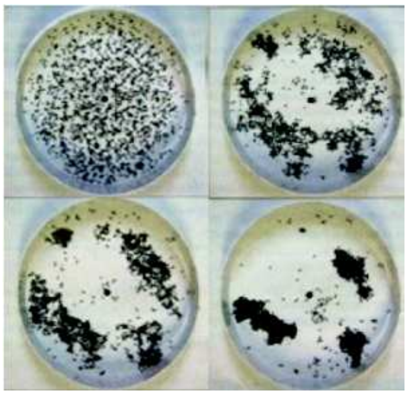


Figure 3.1: Progressive agglomeration of ants into distinct clusters. Reproduced from [9].

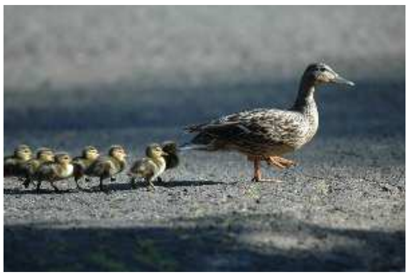


Figure 3.2: A number of ducklings form a chain by following their mother. Reproduced from [10].

3.2.3 Perimeter formation

Perimeter formation may occur for a number of reasons, including protection of weaker group members, guarding of resources or improved circumferential awareness. Figure 3.3 shows a number of soldier termites guarding the entrance to their nest while workers transport food inside.

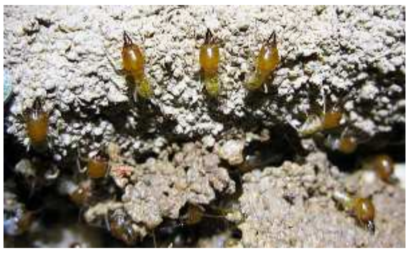


Figure 3.3: A number of soldier termites form a perimeter around a tunnel leading to the nest while workers transport food inside. Reproduced from [11].

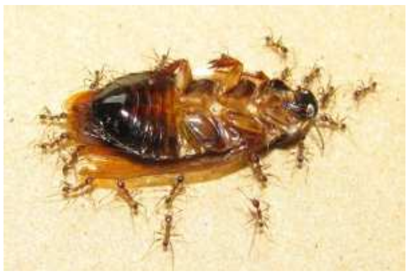


Figure 3.4: A group of ants cooperatively transporting a cockroach. Reproduced from [12].

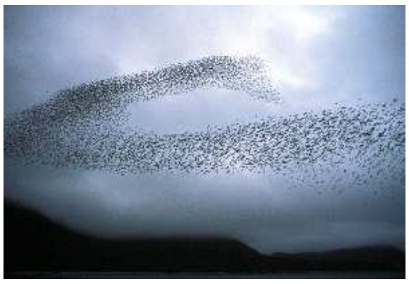


Figure 3.5: A flock of auklets. Reproduced from [13].

3.2.4 Cooperative Transport

If an ant encounters a food source that is too large for the individual to move alone, that ant will recruit nestmates to assist with transportation of the prey back to the nest [3]. A remarkable emergent behaviour occurs as the group attempts to align itself and synchronise the members' individual efforts to move the prey, in the absence of real-time communication [3]. Figure 3.4 shows a group of ants attempting to move a cockroach. The notion of cooperative transport could also be generalised to other cooperative goal-achieving behaviours, such as building or repairing ant nests and termite mounds.

3.2.5 Flocking

Flocking is a behaviour most commonly associated with birds, but can be generalised to other collective animal locomotion, such as shoaling of fish, herding of antelope or swarming of bees. It has been shown by computer models that flocking is the result of each individual in a group following simple rules, with the result being complex emergent behaviour. Flocking can be differentiated from agglomeration in that flocks generally continue to move, whereas agglomerated clusters do not. Figure 3.5 shows a flock of auklets in flight.

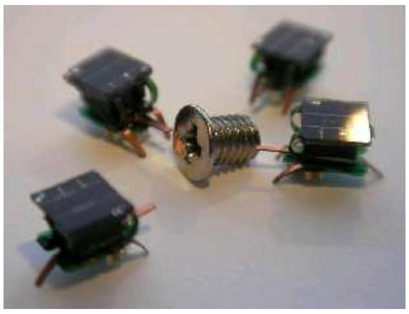


Figure 3.6: A number of I-Swarm microrobots gather around a screw. Reproduced from [14].

3.3 Robotic Swarm Platforms

Swarm robotics is a relatively new field of research, aimed at reproducing in robots the swarm behaviours and swarm intelligence seen in nature [5]. This is achieved by using a (typically large) number of autonomous robotic agents in a distributed fashion. These robots are free to interact with each other, much like biological agents. They are also often given some kind of artificial intelligence in order to readily adapt to their environments [5]. Potential applications for swarm robotics include tasks that require miniaturisation, such as sensing tasks in the human body. Other uses include distributed mining and agricultural foraging.

This section will present a short selection of existing swarm robotics platforms; that is, the physical robots on which swarm robotic software can be based. It is by no means comprehensive, but serves to provide the unacquainted reader with an overview of the existing technology.

3.3.1 I-Swarm

The I-Swarm project is an initiative by a number of European universities to produce micro robots capable of functioning in a *VLSAS* (very large-scale artificial swarm) of up to 1000 nodes [14]. The size of these robots, shown in Figure 3.6, is approximately $3 \times 3 \times 2$ mm^3.

The robots are powered by micro solar cells, which provide energy for miniaturised wireless communication, locomotion and actuators. The robots have integrated circuits (ICs) to provide complex control control, such as on-board intelligence [15].

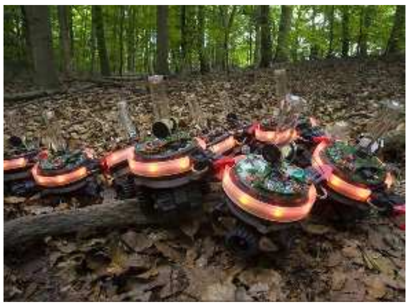


Figure 3.7: A group of s-bots traverse forest terrain. Reproduced from [17].

3.3.2 Swarm-bots Project

The swarm-bots project is a community effort to build swarm robots, mainly for the purpose of shape formation and navigation of difficult terrain [16]. The robots, known as *s-bots*, are 116mm in diameter and 100mm high, and are shown in Figure 3.7. The locomotion system makes use of tracks and wheels with differential drive, allowing stationary turns and traversing of challenging terrain.

The main body of the s-bot is equipped with a number of sensors for navigation, such as infrared proximity sensors, light sensors and accelerometers, as well as a rigid gripper. The robots also have communication devices in the form of coloured LEDs and sound emitters [16]. In contrast to other swarm robots, s-bots are able to physically link themselves together (using their grippers), allowing novel behaviour such as crossing over crevices.

3.3.3 UCT Swarm Robots

The University of Cape Town has developed a number of small, inexpensive swarm robots, shown in Figure 3.8 on the following page. Each robot has six infrared proximity sensors, as shown in Figure 3.9. They also feature light sensors and infrared line sensors, the latter allowing them to recognise lines on the ground. This can be used to force the robots to move in a pre-defined, delineated area. The robots are equipped with two wheels on either side which provide differential locomotion. The motors are powered by a lithium-ion battery which has a life of approximately 150 minutes [18].

Their simple construction and low cost make them an attractive option in comparison to other swarm robots. In addition, the fact that they are available for use at UCT led to the decision to use them as the basis for the simulated robots in this project.

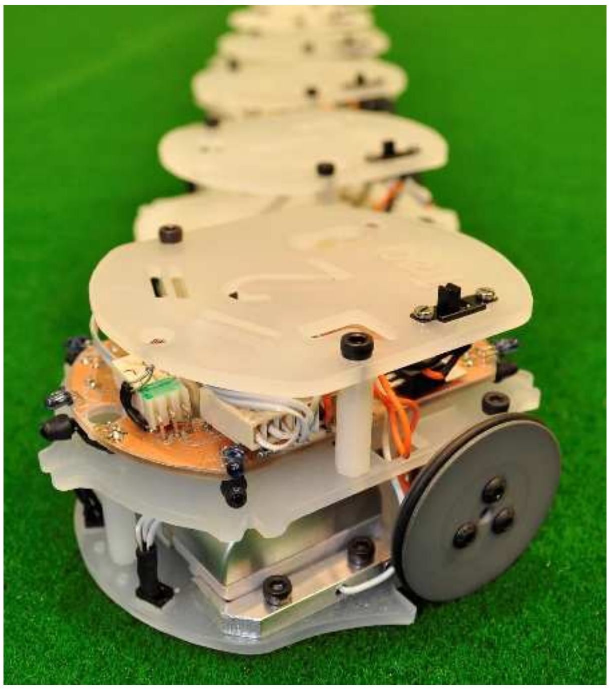


Figure 3.8: Swarm robots developed by the UCT Robotics and Agents Research Lab. Reproduced from [18].

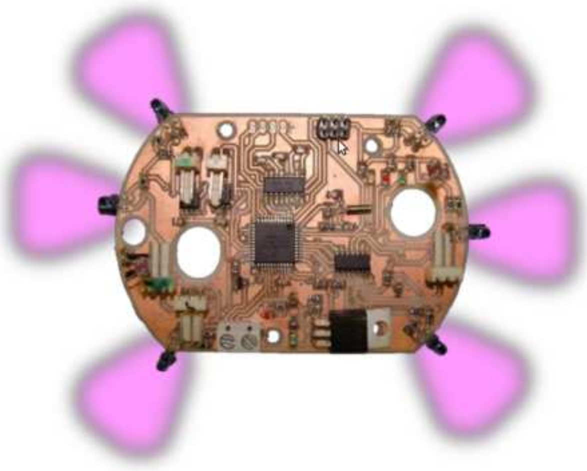


Figure 3.9: Circuit board of a UCT swarm robot, showing the location and field of vision of its infrared sensors. Modified from [18].

Chapter 4

Physical Simulator

In many technical disciplines, computer simulation is an invaluable tool that allows analysts to simulate real-world systems by the creation of an abstract computer model. This chapter will focus on the simulator that was developed during this project for analysis of multiple-agent robotic interactions. The benefits of performing computer simulations are first discussed, followed by a survey of the existing simulators suitable for this task. The abstract physical model used by the simulator is described and its fidelity evaluated. The focus then shifts to the programmatic details of the actual simulator that was developed, as well as an explanation of its overall operation.

Calculations and programmatic details relating to the physical model that were thought too technical for the main report may be found in Appendix A on page 126.

4.1 Rationale for Simulation

Computer simulation is a powerful tool for predicting and studying system behaviour which is too complex to model using a purely analytical approach. While this statement is true in general, robotic simulations in particular confer a number of unique advantages over real-life robot testing. In order to ensure that the simulator to be designed would allow the user to realize the benefits of simulation, while attempting to avoid some of the common pitfalls, an evaluation of the merits of robotic simulation was undertaken. These advantages are presented first below, followed by some known caveats.

Robustness of simulated robots.

Damage can result if real-life robots are programmed with faulty control strategies that result in high-speed collisions with obstacles and other robots, or sustained strong motor currents when trying to push into a wall [1]. Simulation allows such behaviours to be anticipated before transferring these control strategies to actual robots. This ability is particularly advantageous when modelling robot learning strategies, such as the one presented in Chapter 8, where robots regularly collide with obstacles while learning how to avoid them.

Infinite energy supply.

While simulated robots can certainly be modelled so that they have a finite energy supply (if such a degree of realism is desired), it is often the case that the control strategies are of greater interest than the longevity of the robot's batteries. Simulations can therefore detect emergent behaviours which only manifest after a period of time greater than the robot's battery life. Such behaviours would not be witnessed in real-life testing.

Ease of analysis.

A posteriori analysis of physical robot behaviours can be problematic if the data required for analysis was not captured by the robot's controller during testing. Data typically not captured could include global coordinate positions, orientations and data not detectable by an individual robot's sensors (e.g. if a collision occurs in a sensor 'blind spot'). The advantage of simulation here is clear: since all physical data is stored in the simulator, such data can easily be accessed and post-processed, providing insight into the robot's behaviour during the simulation. This can be particularly useful for very long simulations where it is infeasible to directly observe robot-world interactions; in such cases various data handling techniques (such as graphing selected variables) can be utilized.

Speed of simulation.

In many cases, simulation trials can be performed faster than real-time [19], thus allowing the data sets referred to in the previous paragraph to be generated much faster than if testing were performed on actual robots. However, it is not always the case that simulation proceeds faster than real-world testing [1], as the computational load of the simulation is a function of the complexity of the simulation's environment, and the robots' interactions therewith.

Since the computation of the physics engine and robotic controllers is borne as a monolithic burden by the simulator, the speed of the simulation will depend strongly on the fidelity of the physics engine, as well as on the number of simulated agents and their computational complexity. For example, if robots are equipped with vision sensors (as opposed to much simpler IR sensors), computational load per agent increases dramatically. However, even if the speed of simulation on a single processor core is lower than real-time, simulation still provides the unique advantage of being able to parallelize the computation across multiple processor threads (in certain cases), multiple cores and multiple workstations. Simulator performance is further discussed in Section 4.8 on page 45.

Low cost.

The financial costs of simulating robotic behaviour computationally are trivially small in comparison to building, maintaining and testing a corresponding collection of physical robots [1]; this is particularly true if such robots are to be used for advanced applications, such as marine robots [20] or military robots [21]. It is noted, however, that particularly intensive simulations may require the purchase of computing time on high-powered clusters.

In spite of the above advantages, there also a number of problems associated with attempting to simulate real-world robot behaviour.

Noisy data.

Robot sensors rarely deliver fully accurate signals, but rather an approximation of what they are measuring [22]. Similarly, actuators and motors often do not perform as expected for a variety of unpredictable reasons [23]. Simulated sensors and motors do not, by default, have such inaccuracies. While superficially this may appear to be an advantage as robot behaviour can be evaluated in simulation without the need to consider data noise, it can in fact be a severe problem as such noise affects robot behaviour profoundly [1]. Indeed, when simulating agent learning algorithms, the drop in performance when moving from the simulated to the real environments is markedly less for simulations that include noise [24, 25].

An obvious method of overcoming this problem is to add appropriately-bounded random noise to the signals returned by sensors, and also to the effects of motors. Random noise within a certain range can be added at each timestep or the same random noise added for some finite number of timesteps [24]. However, appropriate selection of the amount of noise added is important if the simulation is to be of practical value. For example, in evolutionary learning simulations, it has been found that where too much noise was present in the simulation, the performance of the evolved controllers in a real-world test was substantially lower [25].

Inaccurate representations.

The particular assumptions that are made in the abstracted physical model can have a profound effect on the behaviour of simulated robotic agents [1]. modelling certain physical phenomena, such as sonar or the low-velocity motion of wheeled vehicles, can be difficult for physics engines [22]. Additionally, a danger of simulation is that programmers and simulated robots alike will attempt to solve problems that are merely artifacts of the physical model used, and would not appear in the real world [23]. A corollary is that control algorithms that work well in a simulated environment will perform worse, or not at all, in a real-world environment [23].

Determination of physical constants.

Closely related to the previous point of *how* physical phenomena are modelled, is the issue of "*how much*" they should be modelled or more precisely, how to determine the constants that are used in the physical model. For example, in a robotic simulation that features gravity, how the value of the gravitational constant should be chosen is non-trivial. In general, the time taken resolving physical relationships into a set of simulation constants is a drawback of simulation [22]. Whether these constants accurately reflect reality is a separate question.

It is often possible to preserve dimensionless numbers if, for example, length and time in the simulation are linearly mapped to length and time in the real world. Then it would be possible to suitably determine the gravitational constant by dimensional analysis. For more complicated physical phenomena (such as the fire-fighting task described in Chapter 9), however, it is not always possible to use dimensional analysis and some degree of modelling (with appropriate assumptions) may be necessary.

The characteristics of simulations listed above influenced the design choices that were made, in a number of ways. To allow easy analysis of data and improve the speed of simulation, the visualisation and core simulation functions were segregated, as is further described in Section 4.6. The ability to add custom code to input files that will store simulation data and output it into other formats (such as SciLab matrix files) for subsequent analysis, was added. One of the reasons for choosing Python as the design language was the ease with which multicore processing can be attained by use of the *Parallel Python* module (other reasons are discussed in Section 4.5.2). Finally, the ability to add random noise to sensor inputs, as well as random fluctuations to motor behaviours, was implemented and is described further in Section 4.3.2.2.

4.2 Survey of Existing Simulators

In order to design a robotic simulator capable of simulating swarm behaviour, a survey of existing robotic simulators was performed. This was done chiefly to determine existing best practices in robotic simulators, as well as to identify traits that could make such simulators unsuitable for modelling swarm behaviour, in particular.

This section serves two purposes: to provide the reader with an account of the existing software for robotic agent simulation, as well as to present those simulators whose

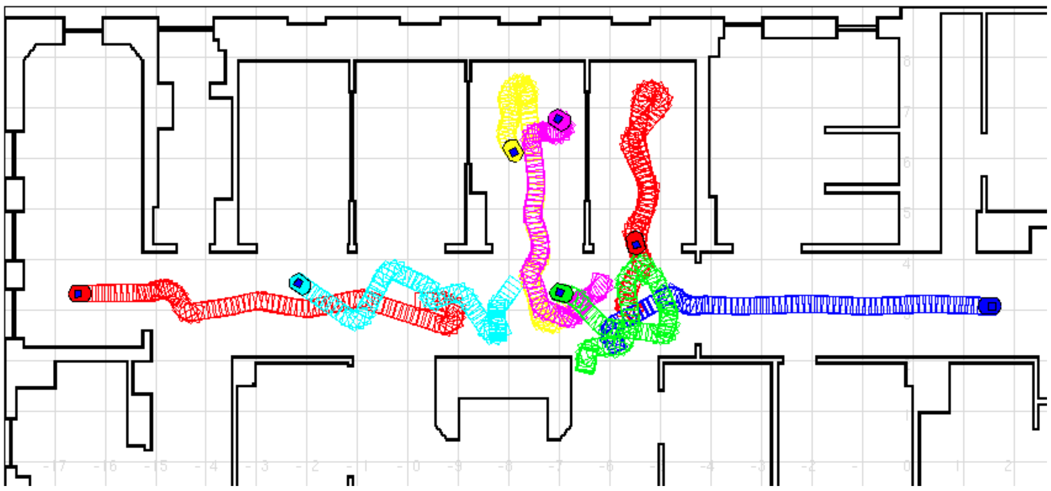


Figure 4.1: A typical Player/Stage simulation, showing a number of variously-coloured robotic agents traversing a 2D map. Reproduced from [26].

properties most influenced the design process of the simulator created in this project.

4.2.1 Player Project

The Player Project is a collection of computer programs used for simulation of robotic behaviour [19]. It consists of three components: the Player server, the Stage 2D simulation environment and the Gazebo 3D simulation environment, available in a number of programming languages. The general mode of operation is to use the Player server to provide client software with an interface to robot controls and sensors, while using either Stage or Gazebo as a simulated robotic environment into which artificial robots can be deployed, as shown in Figure 4.2.1. Player can also be used in non-simulated environments: it provides application programming interfaces (APIs) which client software can use to communicate with real-world robots while actually deployed. This is possible if the robot supports the Player protocol (as do most of those mentioned in Section 3.3). It thus has the significant advantage of code transferability; that is, code written for a simulated environment can be transferred to a real-world environment with little or no modifications [19].

The Player server operates by abstracting generalized robot behaviour into a set of predefined interfaces such as *power*, *position2d*, *ir* and others. Thus, the Player server provides client software with common and standardized methods to interact with robots' sensors and other devices (usually actuators and motors) through an abstract interface. The Player server also interacts either with a physical robot or the Stage/Gazebo environments which perform the actual simulation. These environments have their own physics engines that perform, inter alia, rigid body dynamics and collision detection. Although

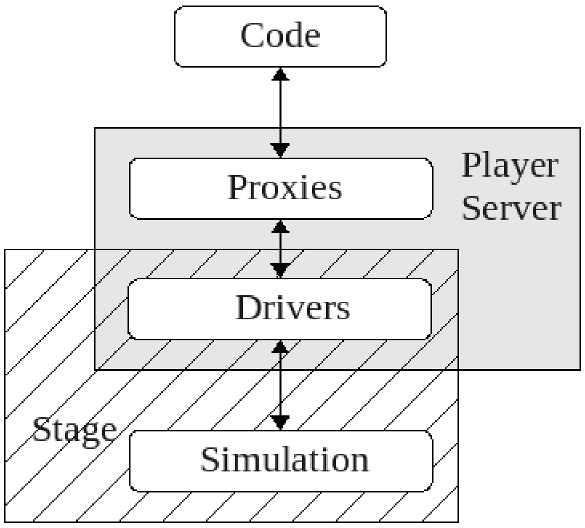


Figure 4.2: Functional relationships in the Player/Stage environment. Adapted from [27].

simulations can be performed with Stage alone [19], the combination of Player and Stage is most commonly used and the combination is often known simply as “Player/Stage”. The functional relationships between the client software, Player server and Stage environment are shown diagrammatically in Figure 4.2 (‘Code’ in the figure refers to client software).

Typically, the client software referred to above would be the robot’s control software which sends commands to actuators and motors based on sensor input. Thus, the client software has no way of knowing if the agent with which it is interacting is actually physical or simulated. In terms of computational efficiency, the physics engine of Stage has, as the authors describe it, “*good enough* fidelity”, with the stated goal being to use computationally cheap models in order to simulate a large range of devices [19, page 3].

The connection between the client software and the Player server, and between the Player server and Stage environment, is made over the TCP/IP protocol, even if the client software, Player server Stage environment all reside on the same computer. A potential consequence of this is feedback lag: commands issued by, and data sent to, the client software may experience much higher latency than if all these components were in a single process. Aside from this potential problem, the Player/Stage platform is a highly modular and feature-rich robotic simulation environment, well-suited to modelling of swarm behaviour.

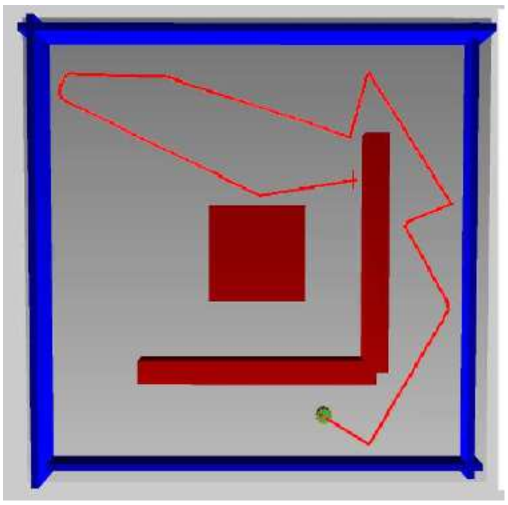


Figure 4.3: Trace of robot movement from a Simbad simulation. Reproduced from [28].

4.2.2 Simbad

Simbad is a three-dimensional robotic simulator written in the Java programming language [28]. It has been designed for research into autonomous agent behaviour, and is specifically oriented towards artificial intelligence and machine learning [28]. In many respects it is very similar in appearance and operation to Player/Stage, with the key difference being that it comes with a number of AI features included, such as neural networks, reinforcement learning algorithms and evolutionary algorithms. A trace of a robot that has been instructed to 'wander' in a Simbad simulation is shown in Figure 4.3.

4.2.3 Boids

Boids is generally regarded as the first simulator of autonomous mobile agents. In particular, it was designed to simulate the flocking behaviours of birds (in three dimensions) [29]. While not a robotic agent simulator, it is included here because it demonstrates the emergence of complex behaviour from agents following simple rules. Specifically, each boid is given the follow rules to follow [29]:

1. Steer to avoid being too close to local flockmates.
2. Steer to match the average direction of local flockmates' movement.
3. Steer toward the centroid of local flockmates.

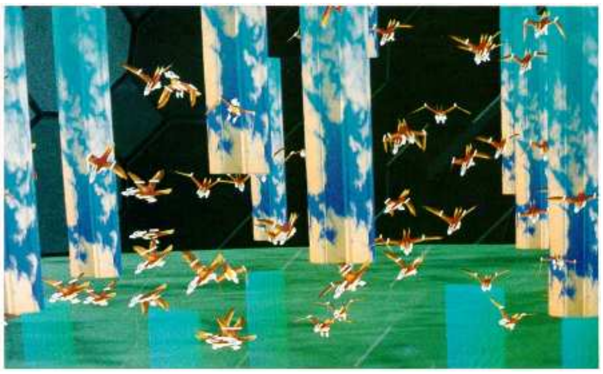


Figure 4.4: Screen capture from Boids showing a number of autonomous agents exhibiting flocking behaviour. Reproduced from [29].

These rules are followed by each agent, and the emergent behaviour is flocking, as shown in Figure 4.4. These simple rules influenced the behaviour-based simulations that were implemented in this project and will be presented in Chapter 5.

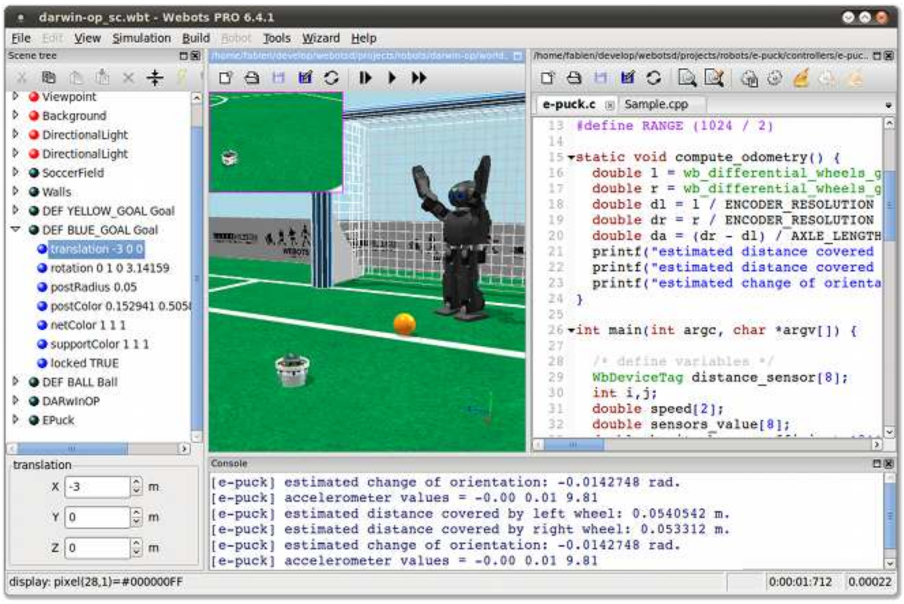
4.2.4 Webots

Webots is a development environment used for the programming and simulation of mobile robots [30]. As a development environment and simulator combined, it provides tools that assist with the development and execution of simulation code, shown in Figure 4.5. The simulator aspect provides a high fidelity three-dimensional environment within which multiple agents can interact. While such a high fidelity environment is certainly useful for advanced applications (such as robots that play soccer, as shown in Figure 4.5), it was decided that such an accurate physical representation was unnecessary for the goals of this project.

4.2.5 Critical Evaluation of Existing Simulators

Much information was gained from studying existing simulators. The model used by Player/Stage, where simplified agents move around a map, was a considerable influence on the initial design concepts. However, while the ability of Player to provide a common interface that can be used for either real robots or the Stage simulation software is useful for practical robot design, it was felt to be unnecessarily complex for this project. This was because the explicit goals of the project were to simulate robotic swarm behaviour, and not to directly transfer such behaviour into real robots.

Simbad was also influential on the early design and, together with Player/Stage, resulted in the adoption of a two-dimensional model consisting of flat map entities. Boids served as



```
#define RANGE (1024 / 2)

static void compute_odometry() {
  double l = wb_differential_wheels_g
  double r = wb_differential_wheels_g
  double dl = l / ENCODER_RESOLUTION
  double dr = r / ENCODER_RESOLUTION
  double da = (dr - dl) / AXLE_LENGTH
  printf("estimated distance covered
  printf("estimated distance covered
  printf("estimated change of orienta
}

int main(int argc, char *argv[]) {

  /* define variables */
  WbDeviceTag distance_sensor[8];
  int i,j;
  double speed[2];
  double sensors_value[8];
```

```
[e-puck] estimated change of orientation: -0.0142748 rad.
[e-puck] accelerometer values = -0.00 0.01 9.81
[e-puck] estimated distance covered by left wheel: 0.0540542 m.
[e-puck] estimated distance covered by right wheel: 0.053312 m.
[e-puck] estimated change of orientation: -0.0142748 rad.
[e-puck] accelerometer values = -0.00 0.01 9.81
```

Figure 4.5: The Webots development environment for the programming and simulation of mobile robots, using a high fidelity 3D physics engine. Reproduced from [31].

a useful introduction to the topic of emergent behaviour, and the rules used in the Boids simulation influenced the choice of rules for the simulations in Chapter 5. Finally, Webots did not influence the design of the simulator directly, but did assist in differentiating the requirements of different simulators and deciding what was unnecessary for the project. For example, the integrated development environment provided by Webots, as well as the high-fidelity physics model, were useful to observe for the sole reason that they illustrated the complexity associated therewith.

Overall, the research on existing simulators helped inform the decision to use a two-dimensional map environment, with a relatively low-fidelity physics model. Based on the results of simulations performed using other simulators, it was seen that notable behaviour can still emerge, even without a rich physics engine. Indeed, as noted by the creators of Player/Stage, “low fidelity simulation can actually be an advantage when designing robot controllers that must run on real robots, as it encourages the use of robust control techniques” [19, page 3].

4.3 Physical Model and Physics Engine

This section details the decisions, assumptions and simplifications made in the development of a physical world model for the simulator, as well as the corresponding physics engine. The method of modelling the environment will first be presented, followed by the model of individual robotic agents and their components. Finally, the broad design of the physics engine will be explained. Detailed calculations for the physics engine are given in Appendix A.

4.3.1 Simulation Environment

As mentioned already, the decision was taken to create the simulated environment in 2D. This was done to reduce time spent on developing complex physical models, such models being unnecessarily complicated for the simple swarm behaviours that this project seeks to capture.

The core of the simulation environment is a 2D *map*. A map defines the limits of the simulated *world*, and includes all elements of the world that are not robotic agents. These elements are generally *obstacles*, which can be thought of as solids objects impenetrable by agents, but can include any environmental artifacts that a designer should wish to model. For example, for the fire-fighting task detailed in Chapter 9, it was necessary to create a new type of map object, namely a ‘fire’. Map entities can be given arbitrary properties (such as ‘temperature’ for fires), and can also be excluded from the contact model, meaning that agents cannot collide with these entities. The latter may be useful for entities such as ground markings.

Before proceeding, a brief note on the world-related nomenclature used in this report may be helpful. The term ‘object’ is used to refer to any element in the simulation, including agents, obstacles and any other simulation entities, such as fires, ground markings, etc. The term ‘world’, by contrast with ‘map’, is used in this report to refer to all objects in the simulation, be they obstacles, agents or other special types of objects, such as fires.

In the broadest sense, the simulator operates by discretising the flow of time into a number of ‘simulation time steps’. At each time step, a *world loop* is performed, which consists of three main events:

1. Every object in the world is *refreshed*; this can have a number of consequences, depending on the object.
2. The physical contact model is enforced, to ensure that the object refreshing that took place in the previous step did not violate the simulator’s physical model (and correct this, if it occurred).
3. Once compliance with the physical model has been ensured, the state of the world (i.e. all objects’ coordinates and properties) are made available to the rest of the simulation.

What also generally follows from the last step in the world loop is an updating of the graphical view of the simulated environment, but this is not strictly necessary. The state of the world could be written to disk, or passed to some other software for analysis.

The first step of the world loop, a refresh of all objects in the world, is performed so that all objects may update their properties at the new time step. For agents, this equates to changing map positions and orientations to match their velocities. When an agent is refreshed, it, in turn, sends a refresh instruction to its sensors, which provide the agent with updated sensory data. While objects such as obstacles are generally static, they are still provided with the ability to refresh should they be made dynamic in the future (for example, if moving obstacles were to be implemented). Other world objects are also required to have an ability to refresh (specifically, a `refresh()` method), which can perform any function desired of the object, even if this function is simply to do nothing.

The second event in the world loop ensures compliance with the physical model. For example, if, in the previous step, an agent has moved into an obstacle and actually crossed the boundary of the obstacle, this collision will be detected and appropriate steps taken to ensure that the agent no longer occupies the same space as the obstacle. The physical compliance step is purposefully performed *after* the refresh step, to ensure that the final state of the world is physically-compliant when it is released to the rest of the simulation at the end of the world loop.

A selection of maps, provided with the simulator package, are shown in Appendix D.2.

4.3.2 Agent Representation

While a number of robotic hardware configurations are available for swarm robotics, the model developed and used for all the simulations in this project was purposefully based on the existing swarm robots at UCT (shown in Figure 3.8 on page 20).

Essentially, these robots have two wheels mounted on either side, and a number of infrared (IR) sensors placed at intervals around the periphery of the agent, as shown in Figure 3.9 on page 20. The robot uses differential wheel locomotion: it turns and moves forward and backwards by controlling the speed of each wheel individually. It also has a line sensor (to detect lines on the floor) and a bump sensor, which detects a collision with another object.

The model of this swarm robot that was created for the simulation is shown in Figure 4.6 on the following page (a). The robot is modelled as circular in shape, for the sake of simplicity. It has two wheels on either side, and uses them in the same manner as the real robot (these wheels are not shown in simulations). Since the orientation of the real

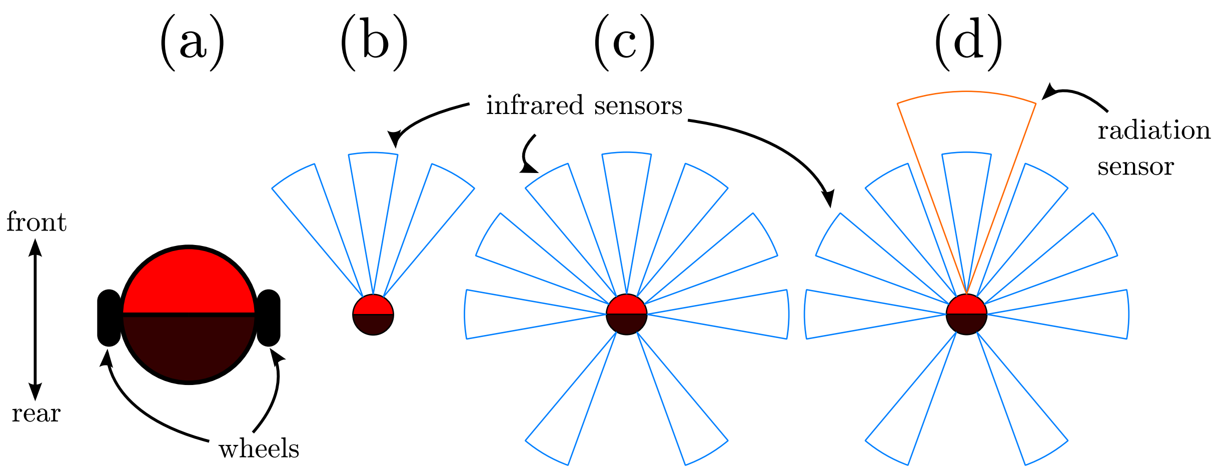


Figure 4.6: Pictorial representations of various simulated agent designs.

robot can change as it moves, the simulated robot is drawn with two differently-coloured halves, so that its orientation is perceptible.

To provide modularity and make the simulated robot easily configurable, a robot superclass (named `AgentCir`) was created. It is capable of providing simulated robots of any size, with a circular shape and two laterally-fixed wheels that function differentially. In order to obtain an actual simulated robot, a sensor configuration is required. This configuration is specified in a subclass, such as `Agent3` for the robot shown in (b) in the figure, or `Agent9` for the robot shown in (c). Sensors need not be limited to infrared, the fire-fighting agent shown in (d) is equipped with a radiation sensor, able to sense fire temperatures (this agent is in the `AgentFire` class).

The extensible nature in which the simulator has been written permits the user to easily create new sensory configurations using the `AgentCir` class. With a bit more work, it is possible to create completely novel robot designs that can integrate with the simulator. Such designs may include rectangular robots that feature wheel sets aligned at 90° to each other, or robots that use a single spherical rubber ball to instantly move in any direction (similar to the ball in a computer mouse). In these cases, users are required to do two things:

1. If the overall shape of the robot is not circular, the contact model needs to be updated to check for collisions between the new shape, and the existing shapes. For example, if a rectangular robot shape is created, the contact model needs to check for collisions between rectangles and circles (as it already does), but also for the new possibility of contact between rectangles and rectangles.

2. Each robot class is required to have a `refresh()` method, which is responsible for updating the robot's position and orientation (locomotion), and querying the

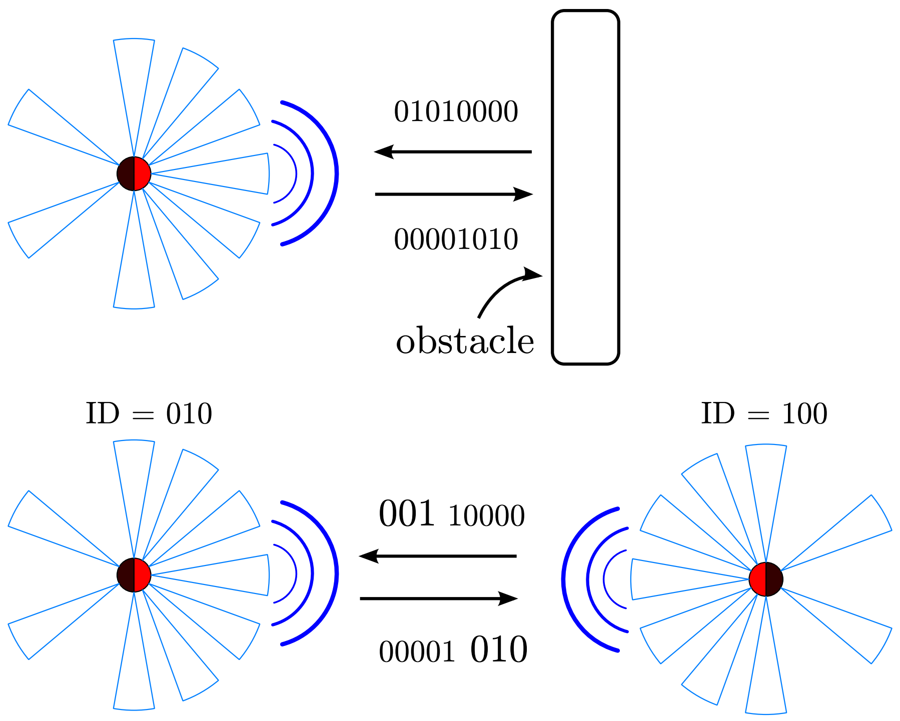


Figure 4.7: Schematic showing how robots distinguish sensed obstacles from agents. Drawn using information from [32].

sensors. Therefore, if a novel locomotion method (such as the spherical rubber ball) is to be used, the robot class needs to provide a method for this locomotion to be simulated by updating the robot's positions and orientations accordingly.

New robot designs can still use the existing sensor objects, such as IR sensors or radiation sensors. Additionally, it is possible for the user to write their own sensor classes that have totally different functionality, such as sonar. Existing sensors can also be modified to simulate the properties of other sensors. For example, it would be possible to simulate a laser sensor by simply using the existing IR sensor with a very narrow angular range, and perhaps an increased depth of vision. Bump sensors could be simulated by using IR sensors that have a very shallow depth.

4.3.2.1 Sensor Representation

The primary sensors used on the simulated agents in this project were IR proximity sensors. Each sensor has a depth of field, and angular range, as shown in Figure 4.6. When a sensor's `sense()` method is called by its parent agent, the sensor returns the value to the nearest object within its field of vision.

Sensors were also given the ability to distinguish agents from other obstacles. In the real world, this can be achieved by observing the difference between the emitted and received IR signals, as shown in Figure 4.7 on the previous page [32]. Agents are able to distinguish obstacles from other agents by examining the incoming infrared signal and comparing the signal to the one emitted. The cited work employs IDs unique to each agent, but this is not necessary. In order for an agent to distinguish another agent from an obstacle, it is required that the second agent's emitting diode be in the field of vision of the first agent's sensor. However, in the simulation, the simplification was made that a sensor can intrinsically differentiate between agents and obstacles without the need to detect emitter diodes directly.

Because simulated sensors only return to the controller a single scalar value indicating the distance to the nearest detectable object, there is an inherent error in using such information to reconstruct the position of the object. IR sensors have a non-zero angular range, therefore detected objects could lie anywhere within this angular field of vision. The agent has no means of directly determining where in this angular range a particular object lies. The result is that the best calculation of object position the agent can make is by assuming the object lies within the middle of the sensor angular range. This introduces potential error into the reconstructed position of the object.

Another important simplification that was made in representation of sensors is that their refresh rates were assumed to be the same as the 'refresh rate' of the discretised simulation (or more correctly, sensor period was taken to be the same as the size of the simulation time step). This could potentially overstate (in the simulation) the ability of an agent to react swiftly to changing sensory input. If desired, the sensor class could certainly be modified to only provide data at pre-determined intervals (e.g. every n time steps).

The mathematics of sensor operation is covered in Appendix A.3.

4.3.2.2 Random Noise

In Section 4.1 it was described how the lack of random noise in simulations can make such simulations at best, inaccurate, and at worst, invalid. The presence of noise in sensory data, as well as unpredictable motor characteristics, have a significant effect on real robot behaviour. Therefore, if a simulation is to remain relevant and present a faithful representation of physical reality, the simulation should include random noise.

The ability to simulate random noise has been designed into the simulator in two ways: sensory noise and unpredictable motor behaviour. The former is a property of each individual sensor object, with the latter being an agent property.

Error on sensor readings is implemented by means of two variables, which are attributes of each sensor: the probability of sensor error occurring (`sep`) and the fractional magnitude of the corresponding error (`sem`). Briefly, at each time step, sensors calculate their values as described in the previous section, but with probability `sep` they will add or subtract a fraction `sem` of the true reading to the reading returned to the agent.

Error on wheel behaviour is implemented by means of wheel slip. Similar to the case with sensors, this is defined by the probability of wheel slip occurring (`wsp`) and the fractional magnitude of the corresponding error (`wsm`). Note that each wheel is treated separately, so it is possible that one wheel may slip, and not the other, leading to unintended turning.

4.3.2.3 Other Issues

The simplified agent model necessarily lacks some properties of real robots. For example, real robots use batteries for powering motors, controllers and sensors, and therefore the length of a particular robot excursion is limited by the life of the on-board batteries. Additionally, robot components (such as motors and controllers) can occasionally suffer from malfunctions or complete breakdown, such as wear on motor brushes. While these and other real-world phenomena can certainly be included in the simulator, they were deemed outside the scope of the project and therefore not implemented.

4.3.3 Contact Model

At each simulation time step, after all objects have been refreshed, a contact function is invoked, which detects contact between objects and, in the case of a collision, performs the appropriate actions. The general algorithm is to consider every possible pair of objects in the world at each time step and mathematically check if they have made contact. This algorithm can be considered naive, in that it considers contact between objects that are potentially far apart. The effect of this on the performance of the simulator will be analysed in Section 4.8 on page 45, with the conclusion that it is relatively inefficient. Despite this inefficiency, the simulator performance was generally found to be acceptable for the simulations performed during this project, mainly because of the ability to separate core simulation and visualisation (detailed in Section 4.7 on page 42).

When a moving object collides with another object (moving or stationary), the collision function is invoked to handle the collision and ensure that no areas overlap. In initial versions of the simulator, a collision model employing conservation of momentum and coefficients of restitution was used. This was found to be inaccurate because it did not account for the friction between the ground and an agent's wheels (which prevents lateral motion of the agent). It was decided that to incorporate such friction into a high-fidelity physics model would be unnecessarily complex, so a simplified model was adopted instead.

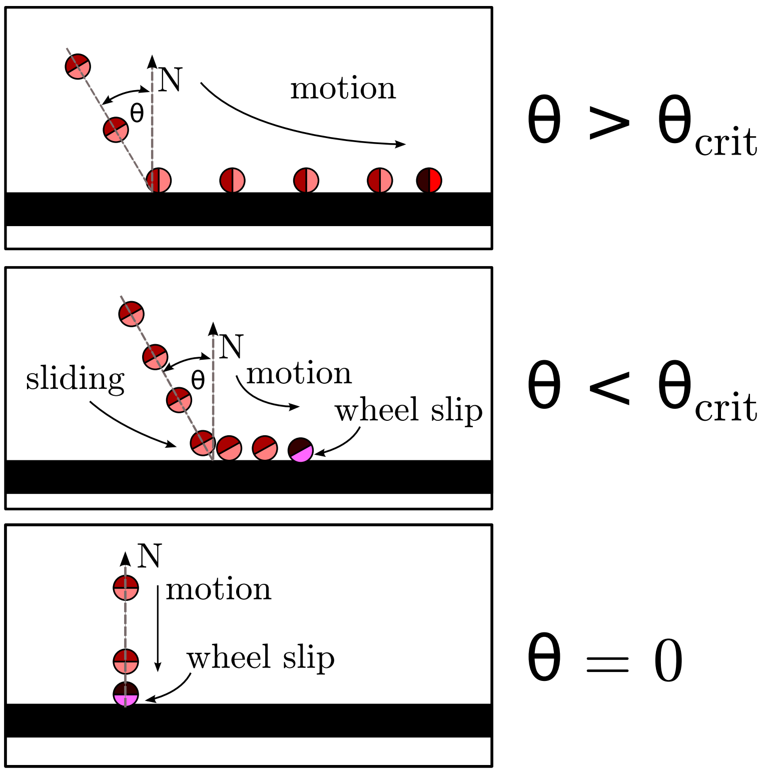


Figure 4.8: Several cases of a circular agent colliding with a rectangular object.

Essentially the model considers two types of collisions: those between rectangular and circular objects, and collisions between two circular objects. General cases of such collisions are shown in Figures 4.8 and 4.9 on the next page, respectively.

Essentially, when a circular agent collides with a rectangular object, one of three things can happen, depending on the angle of contact. If the angle of contact is shallow enough, the force of the object on the agent causes the agent to rotate so that its motion continues parallel to the object. If the angle of contact is more direct, the agent will not be rotated by the object, but will instead attempt to move into the object. If this is done at any angle other than 90°, the agent will slide along the object, as shown in the middle frame. In the case of a collision at 90°, the agent will come to a halt, but if it continues to rotate its wheels, the wheels will slip.

For circular agents colliding with circular objects, the agent is never rotated by the object. Instead, the agent will slide along the object surface, maintaining its initial orientation, as shown in Figure 4.9. If the agent were to collide with the circular surface at precisely 90° (not shown), the agent would halt in a similar fashion to the last frame of 4.8. Note that the foregoing collision rules apply equally to agent-on-agent collisions, with the only difference being that both agents experience equal sliding and displacement.

The effects of wheel slip were also included in the physical model. Thus, if an agent

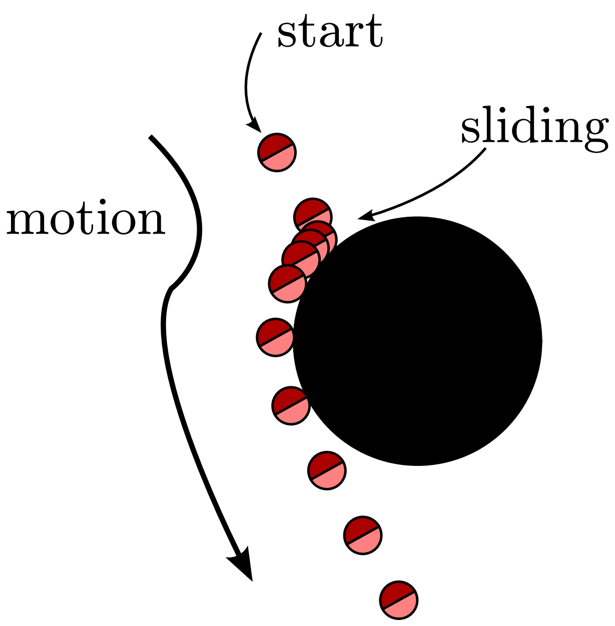


Figure 4.9: Case of a circular agent colliding with a circular object, resulting in sliding.

attempts to move into another object, the collision model will not stop the agent's wheels from turning, but rather cause the wheels to slip. The effect of this is that the agent does not move according to the rate at which its motors and wheels are rotating, but rather according to the amount of contact they make with the ground. An important consequence of this fact is that an agent has no direct way of knowing if it is actually moving relative to the ground or if its wheels are slipping, as is consistent with real robots.

The mathematics of the contact and collision models are given in Appendix A.2.

4.4 Inter-agent Communication

A number of different robot communication methods have been proposed and implemented. Balch and Arkin [33] classify these methods into three functional groups, in increasing order of information entropy.

No explicit communication.

Even if the absence of explicit communication between agents, it is still possible for agents to communicate with each other, if they are able to distinguish between obstacles and other agents [33]. For example, if an agent observes that a large number of other agents have congregated in a certain location, this is a form of communication and the agent might infer from that information that it would be advantageous for it to join the group. It has been shown that implicit communication is sufficient to lead to cooperation in a variety of different robot tasks [6, 33].

State communication.

State communication can be defined as the communication of an agent's internal state, which could implicitly include information about environmental conditions [33]. For example, if a robot is currently engaged in an activity (such as extinguishing a fire), this could be regarded as its internal state and it could communicate this state to other agents. Agents receiving that communication would be able to infer that the first agent must be near a particular resource (in this case, fire), but do not explicitly know where the resource is.

Goal communication.

An agent can also communicate goal-oriented information to other agents. Suppose that in the fire example previously mentioned an agent were to communicate the location of a fire, as opposed to the fact that it is extinguishing a fire. This information is much richer and, depending on the circumstances, could be more useful to other agents than state communication.

The simulations in this project were based on *decentralised* actions with *local* information, in the absence of communication and internal world representations. A brief semantic discussion of these terms is appropriate in order to contrast this approach with other possible methods of swarming. That agents' actions are decentralised implies that each agent is responsible for taking its own actions, which will be a function of its particular control strategy and the extent of its knowledge about the environment (which could be local or global knowledge). This decentralised approach is in contrast to the centralised method of coordination where agents receive instructions from a central authority. In that case, the central authority receives information about the environment, decides which actions each agent should take in order for the desired group outcome to be achieved, and communicates to each agent its specific instructions (e.g. "turn left, then travel straight for 20 units and stop").

The information used in the decision-making process, whether centralised or not, could be obtained in a number of ways. Three such ways will be discussed here, and can be described as *local*, *shared local* and *global*. If information is purely local, then each agent's sensory input is available only to that agent, necessarily implying decentralised coordination. Each agent's local sensory information could also be shared, either with other agents (in decentralised coordination) or with a central coordinator. Finally, information could also be obtained globally, which would require a means of accumulating all information available. This is generally only possible by means of some central entity, which would likely also function as coordinator.

The decision was ultimately taken to implement agents without the ability for explicit communication (thus implying decentralised decision-making using only local information). This was done for several reasons.

1. Being a swarm behaviour simulator, one of the goals of the project is to artificially reproduce swarm patterns seen in nature. Many of the structures arising from such behaviours do not require communication to occur (flocks of birds, school of fish, herds of antelope, etc). However, in spite of this, many swarm behaviours do require the use of some (limited) communication, such as cooperative transport in ants, or the migration of bees.
2. Control systems that rely on communication are necessarily less robust than those which do not require it.
3. The UCT swarm robots, on which the agents in this simulator were modelled, are not equipped with any means of explicit communication.

In spite of the above, it would not be a difficult task to modify an agent class (either the `AgentCir` superclass or any of its subclasses) to make provision for communication emitters and receivers.

4.5 Implementation Details

4.5.1 Software Design Methodology

In order to ensure that the simulator would be well-designed and incorporate existing best practices in the field of scientific software, research was undertaken to gain an understanding of the applicable principles of software design. These are briefly summarised below.

Modularity.

Software should be highly modular; that is, be composed of a number of interacting modules which can easily be added, changed or removed, without disrupting the operation of the overall package [34]. Each module should perform the minimum required of it and make relatively few assumptions about the context in which it might be used. Modularization can be present at various levels of the software, from algorithms to data structures [35].

Object-oriented programming.

Object-oriented programming (OOP) is a programming paradigm that emphasises the use of programming *objects*: collections of data and functions which perform related tasks, grouped into *classes*. Instances of classes can be introduced to the program, which instances inherit the attributes (data and functions) of the classes whence they were created [36]. This principle of inheritance promotes modularity of code, as superclasses can be designed to perform a minimum of tasks, with more specific functions implemented by subclasses which inherit the functionality of their parent classes.

Parametrisation.

Values that are liable to change over the life of the software should be programmed explicitly as variables, or *parametrised*, rather than as constants [34]. For example, in the collision avoidance algorithm introduced in Chapter 5, the minimum distance between an agent and obstacle before the agent turns to avoid a collision is coded as a variable, rather than as a fixed value. This again promotes software modularity as another control algorithm wishing to use the collision avoidance algorithm, but with a different minimum distance, can simply call the algorithm with a different parameter.

Reuse of code.

In general, code to accomplish a specific task should only be written in one place, and not repeated [36]. If the same functionality is required elsewhere, the code should be objectified (as a function, class, or whatever else is appropriate).

Reuse of existing resources.

Similar to the idea of not "reinventing the wheel", an attempt should be made to reuse existing resources as far as possible. While it is often preferable to write software from first principles. as the assumptions made in the process are known explicitly to the programmer, it is frequently the case that exogenous resources developed by specialists in the appropriate field are faster and more correct than those developed by non-specialists [35].

Efficiency.

While it may appear to be a truism that software should be efficient, this is particularly important for scientific software (such as a simulator) where the volume of computation is comparatively larger than in other disciplines and small losses in efficiency can translate to large losses in time.

4.5.2 Choice of Programming Language

The Python programming language was chosen for writing the simulator software. Briefly described below are a number of reasons for this choice.

Object-oriented.

Python was designed as an object-oriented language, and its class structure supports advanced OOP applications such as polymorphism (see next paragraph) and multiple inheritance, which allows classes to inherit attributes from more than one parent class [36]. However, the decision if and when to use the OOP paradigm is left to the programmer: Python also supports procedural programming, allowing software to be developed in a flexible manner to fit the design requirements [36].

Readability.

Python's syntax has been designed to be highly readable often using English words such as 'is', 'not' and 'in' in place of less-readable constructs [36]. This approach follows the notion that code is written (usually) once, but read multiple times. Therefore, it is more important that code is easily read than easily written.

Abstraction.

Python is a relatively high-level (abstracted) language, meaning the software designer is free to focus more on functional aspects of the code, and less on aspects related to implementation. To this end, Python provides a large number of high-level features, including automatic memory management ("garbage collection"), dynamic typing (variables can change type) and myriad built-in object types and tools, allowing common operations to be accomplished easily [36]. Such abstraction in a programming language is usually accompanied by poor performance, and Python is not an exception. However, it is possible to mitigate these performance losses by taking advantage of Python's extensibility.

Extensibility.

Python is able to execute code that is written in other, lower-level languages, such as C, thus realising the significant speed of such languages. Additionally, many common tasks in Python are actually executed by pre-compiled C inside the Python interpreter, thus running at "C speed". For intensive computational work (such as the simulations in this project), the *NumPy* and *SciPy* extensions allow computationally-expensive mathematical operations (such as multiplying matrices) to be executed at the speed of a C program, since these extensions are written in C [36]. This extensibility allows programmers to realise the performance benefits of lower-level languages while enjoying the power and convenience inherent to higher-level languages.

4.6 Simulator Architecture

The simulator is composed of a number of Python scripts, three of which are executable by the user. A relational diagram of all scripts, including brief descriptions of their functions, is shown in Figure 4.10 on the following page. The three executable scripts are shown with a thick border.

The basic procedure for operation of the simulator is as follows. A user will create a `.job` file in a text editor, normally by modifying the sample `.job` files provided with the simulator package. The job file specifies the map, type and number of agents, as well as the control functions used by agents. It can also include other task-specific scripts, e.g. for Q-learning or fire-fighting tasks which will be presented in later chapters.

Once an appropriate job file has been created, the user runs the main simulation program, `sim.py`. This program takes a job file as input, and either displays the simulation on-screen in real-time, saves it to disk, or both. If the simulation is saved to disk, it is written to a `.sim` file which can subsequently be played back using executable `viewer.py`. If the user wishes to define custom scripts, this can be done using executable `mapImporter.py`, which will convert a vector graphics image into the native `.map` format used by the simulator.

4.7 Offline Visualisation

It became apparent at an early stage of the project that if computationally-intensive simulations were to be carried out, the time taken to execute a world loop might become large enough that the refresh rate of the simulation would be unacceptably low if viewed in real time. This would result in jerky visualisations and make analysis of agent behaviour exceedingly difficult and time-consuming.

The decision was therefore taken to separate the visualisation and core simulation components of the simulator, similar in operation to other simulation software such as finite element modellers. Thus, simulation is performed without the requirement that the user be present during simulation, and with the output of the simulation being saved to disk for subsequent viewing. A major advantage of writing to disk is that simulations involving randomness (whether in agent decision-making or in physical phenomena such as random noise) can be reproduced identically every time, which is important for study and analysis of simulations.

All the parameters defining a particular simulation are contained in a *job* file, with file extension `.job`. When the user wishes to run a simulation, he need only locate the appro-

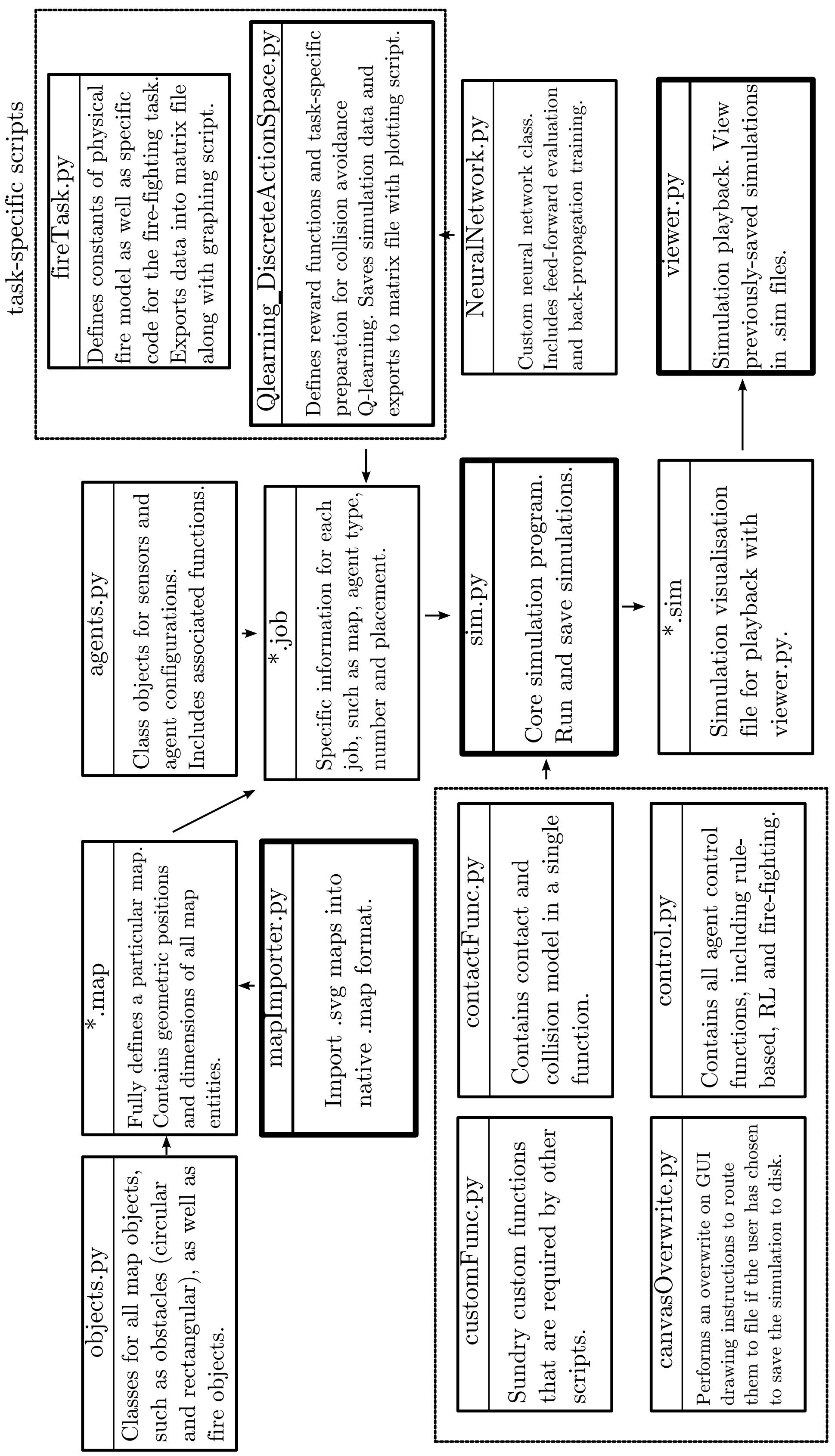
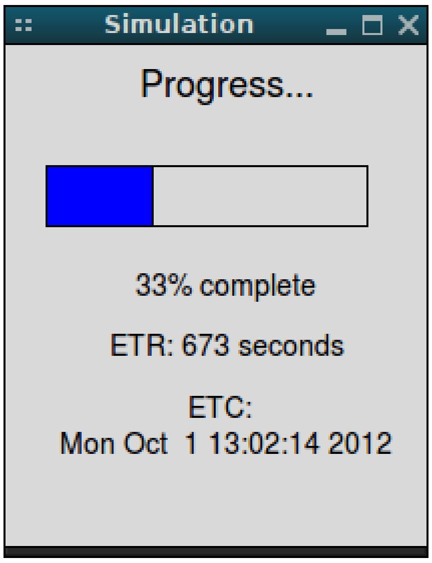

task-specific scripts
fireTask.py
Defines constants of physical fire model as well as specific code for the fire-fighting task. Exports data into matrix file along with graphing script.
Qlearning_DiscreteActionSpace.py
Defines reward functions and task-specific preparation for collision avoidance Q-learning. Saves simulation data and exports to matrix file with plotting script.
objects.py
Classes for all map objects, such as obstacles (circular and rectangular), as well as fire objects.
*.map
Fully defines a particular map. Contains geometric positions and dimensions of all map entities.
agents.py
Class objects for sensors and agent configurations. Includes associated functions.
mapImporter.py
Import .svg maps into native .map format.
*.job
Specific information for each job, such as map, agent type, number and placement.
NeuralNetwork.py
Custom neural network class. Includes feed-forward evaluation and back-propagation training.
customFunc.py
Sundry custom functions that are required by other scripts.
contactFunc.py
Contains contact and collision model in a single function.
sim.py
Core simulation program. Run and save simulations.
canvasOverwrite.py
Performs an overwrite on GUI drawing instructions to route them to file if the user has chosen to save the simulation to disk.
control.py
Contains all agent control functions, including rule-based, RL and fire-fighting.
*.sim
Simulation visualisation file for playback with viewer.py.
viewer.py
Simulation playback. View previously-saved simulations in .sim files.

Figure 4.10: Overall simulator architecture.



33% complete

ETR: 673 seconds

ETC:
Mon Oct 1 13:02:14 2012

Figure 4.11: Dialog box showing progress of a simulation.

priate job file and execute it using the simulator (contained in `sim.py`). The simulator will either show the simulation in real-time, save it to disk, or both. If the simulation is saved to disk, the drawing commands that are normally sent from the simulation program to the graphical user interface (GUI) at the end of each world loop are instead written to disk (or simultaneously executed, if so desired) in a *simulation* file with file extension `.sim`. To visualise a simulation that has previously been saved, the user runs the viewer program (contained in `viewer.py`) and opens the appropriate simulation output file.

If a user runs a simulation and opts to save it to disk and not view it in real-time, a progress indicator is displayed on-screen to give an estimate of when core simulation is expected to finish, as shown in Figure 4.11. It includes a graphical progress bar, as well as the expected time remaining (ETR) and the expected time of completion (ETC).

The drawing commands written to file are in their raw form, exactly as they would be executed by *Tkinter* (Python's GUI). Accordingly, they are relatively verbose and have a low information entropy, which would result in extremely large output files if written to disk in this form. To provide a sense of scale, a particular simulation, which took just under 60 seconds to visualise, has 350 000 drawing instructions which would produce a file 36 MB in size if the commands were written unaltered. This file size is excessive for what is a relatively short simulation. To avoid such large files, Gzip compression is used to reduce the file size. With compression, the same simulation uses under 6 MB, about one-sixth of the original size.

The rationale for writing the drawing commands in raw form, as opposed to developing a special encoding, is portability. The drawing commands used to draw on the *Tkinter* GUI are almost identical to those found in other languages, which have their own implementations of *Tk* (the general GUI of which *Tkinter* is the Python implementation). Therefore, simulation files written with raw drawing instructions can be executed, with only minor modification, in any programming language that has a *Tk*-based GUI.

It was found during testing of the viewer program that certain simulations with a low number of drawing instructions per world loop (usually corresponding to there being few agents) would play back exceptionally fast, making analysis of the simulation difficult. Therefore, an extra function was added to the viewer program allowing it to self-regulate the playback speed. This is an option that can be turned on and off by changing variable `speedReg` from `True` to `False`. When enabled, if the viewer detects that playback is occurring too fast, it will automatically decrease the frame rate to 30 frames per second, which was found empirically to be an acceptable playback speed. Speed regulation is dependent purely on playback speed and will not be affected by the speed of the user's platform.

It should be noted that the viewer is only able to down-regulate the frame rate, not increase it. This is because playback normally occurs at the fastest speed possible. If the user should find that the playback speed is unacceptably slow in the absence of speed regulation setting the `drawSensors` option to `False` in the job file and re-executing the simulation, speeds up playback dramatically. In addition to automatic speed regulation, the viewer is able to slow down playback to a user-determined speed, which can be set in variable `delay` in `viewer.py`.

4.8 Performance of Simulator

This section will provide a qualitative and quantitative analysis of the computational performance of the simulator. Different aspects of the simulator's performance will be evaluated, namely the efficiency of the contact model, the scalability of the program, as well as the effects of different maps and agent configurations on performance.

While there are a number of conceivable metrics that can be used to evaluate the performance of a simulator, the analysis in this section will be based on the time required to complete one *world loop*. A world loop refers to the set of instructions that are executed each time the discretised world model updates. It generally consists of two broad types of computation: physics-based and agent-based. The physics-based computations refer to all calculations and steps performed by the physics engine, and generally include collision avoidance and rigid body kinematics. Agent-based computations refer to calculations and steps performed by each agent, and would generally consist of sensor calculations, as well as calculations within the control algorithm.

While the foregoing classification is generally useful, it should not be understood in a purely literal sense as some ostensibly agent-based calculations, such as a sensor detecting objects in the environment, are related more to the physics of the simulator than to the agent itself. That is to say, the computational cost of each sensor considering every other object in the world to decide which lie within its field of vision would not be borne by the controller of a robot during real-world operation, but rather by the environment itself. If we define τ as the time taken for a single world loop, then it is possible to write

$$\tau = f^*(\kappa, n, s, c, M, X, P)$$

where

$$\begin{aligned}
\kappa &= \text{cost of physics engine} \\
n &= \text{number of agents in simulation} \\
s &= \text{number of sensors per agent} \\
c &= \text{cost of agent control algorithm} \\
M &= \text{complexity of map} \\
X &= \text{list of agents' generalized coordinates} \\
P &= \text{machine performance}
\end{aligned}$$

and f^* is some function of its arguments. A few comments on these variables are in order. Firstly, f^* is unique to a particular agent size and sensor range (i.e. these two factors have not been considered as variables). κ refers to the computational *cost* of the physics engine, which is not identical to the *fidelity* of the engine. It is entirely possible that a poorly-implemented low-fidelity engine could have higher cost than a well-implemented high-fidelity engine. The notation used above also implies that the number of sensors per agent (s) is the same for every agent, but the simulator imposes no such restriction. This assumption of equality has been made to simplify the subsequent performance evaluations, where some variables are systematically altered (while keeping others constant) to determine their effect on performance. While the speed of the simulation likely depends only on the only the total number of sensors in the simulation, it would be somewhat misleading to imply that the number of agents and the total number of sensors were independent of each other.

What is meant by c is the computational cost of one loop of each agent's control algorithm, e.g. a simple instruction-based collision avoidance algorithm would have a much lower c value than a controller that uses neural networks and Q-learning with interpolation. The map complexity, M, refers both to the number of objects in the map, and also how they are arranged. Thus two maps with the same number of objects may have vastly different

M values if one map has all its objects closely packed in one corner, whereas the other has a more even distribution. The latter of these two maps would be considered as having a higher M value as its more distributed nature would mean each sensor has to perform a greater number of sensor checks, since these checks are only performed on objects that lie within a certain range of the sensor. Closely related to the complexity of the map is X, which describes each agent's position in some set of generalized coordinates (e.g. Cartesian position coordinates and angular orientation). Since the position of each agent determines which objects its sensors check, the agent position will have an effect on τ. Finally, P is some measure of the performance of the platform on which the simulator is running. This need not only be processor speed, but would also refer to available memory and take into account other complex factors such as operating system swap management.

While some of these variables may not be easy to quantify (or even quantifiable in principle), the values of these individual variables are not of particular interest. What is of interest is the effect they have on τ. For example, if we wish to assess how the contact model scales with respect to the number of agents, we can set $s = 0$ and make the control function a null function, so that it does not consume computational time (thereby setting $c = 0$). Then, performing the tests on different map configurations (thereby varying M), on the same machine (constant P) for the same physics engine (constant κ), we obtain the computational time complexity of the contact model,

$$\tau_c = f_1(n, M) \qquad . \tag{4.1}$$

The results of simulations for three different map configurations are shown in Figure 4.12 on the following page.

Instead of measuring the actual time elapsed for each world loop, which is strongly dependent on available system resources, CPU cycle time was used. The measure of time complexity shown in the figure has been indexed relative to the lowest data point, for ease of comparison. The maps used for the simulations are described as 'dense', 'sparse' and 'open', corresponding to the number of obstacles they contain, and the arrangement thereof (these maps are shown, respectively, in Figures D.1, D.2 and D.3). The trend lines fitted to the data points are quadratic polynomials with coefficients to give the best least-squares-of-error fit, therefore indicating that the complexity of the contact model is (no worse than) $O(n^2)$. This result is easily predicted. Since the contact model checks each pair of world objects for contact, the number of checks is equal to $\frac{n_w(n_w-1)}{2}$ where n_w is the total number of objects in the world, equal to the sum of the number of agents and the number of obstacles. Therefore, the number of contact checks is $O(n^2)$ and since the time taken per check does not scale with n, the overall complexity of the algorithm is $O(n^2)$. It is interesting to note that, even in the case of the dense map, the increase in

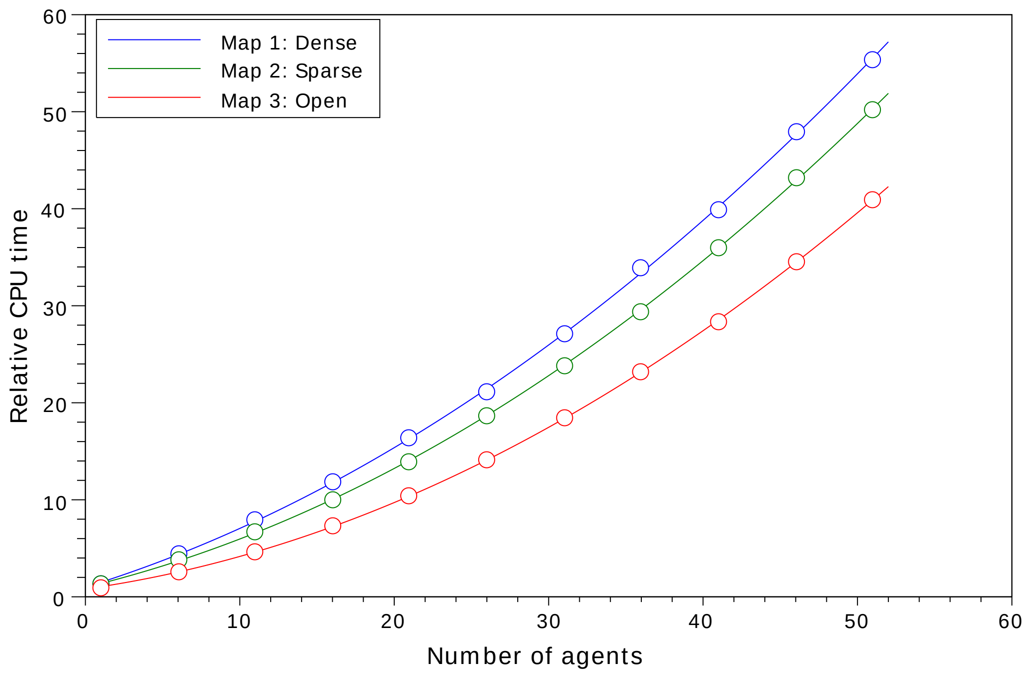


Figure 4.12: Time complexity of the contact algorithm as a function of agent number, for three different maps. Trend lines are least-squares quadratic polynomials.

the cost of the contact algorithm is less than proportional to agent number up to $n = 42$, which is generally sufficient for most swarm simulations.

So far we have only considered the cost of the contact algorithm, which is only part of the full world loop, the other part being agent-based calculations such as updating of positions and sensing operations. However, sensing operations are strongly dependent on the agent's position on the map, due to 'pruning' checks that avoid computationally-intensive sensing calculations if sensors are not near particular obstacles. Despite this, it is possible to eliminate (or at least, minimise) the effects of X by allowing agents to move around the map sufficiently long that they have visited most X values. It is then possible to approximate τ by the mean value during the simulation, $\overline{\tau}$. If agents are given a fixed number of sensors (s), for a fixed control strategy (c), it is possible to write the total world loop complexity as

$$\overline{\tau} \approx \tau = f_2(n, M) \qquad . \tag{4.2}$$

Simulations were performed for the same three map configurations mentioned earlier, with a sensor configuration of three forward-facing sensors, using the collision avoidance rule presented in Section 5.2.2 on page 56, and for 3000 time steps. The computational

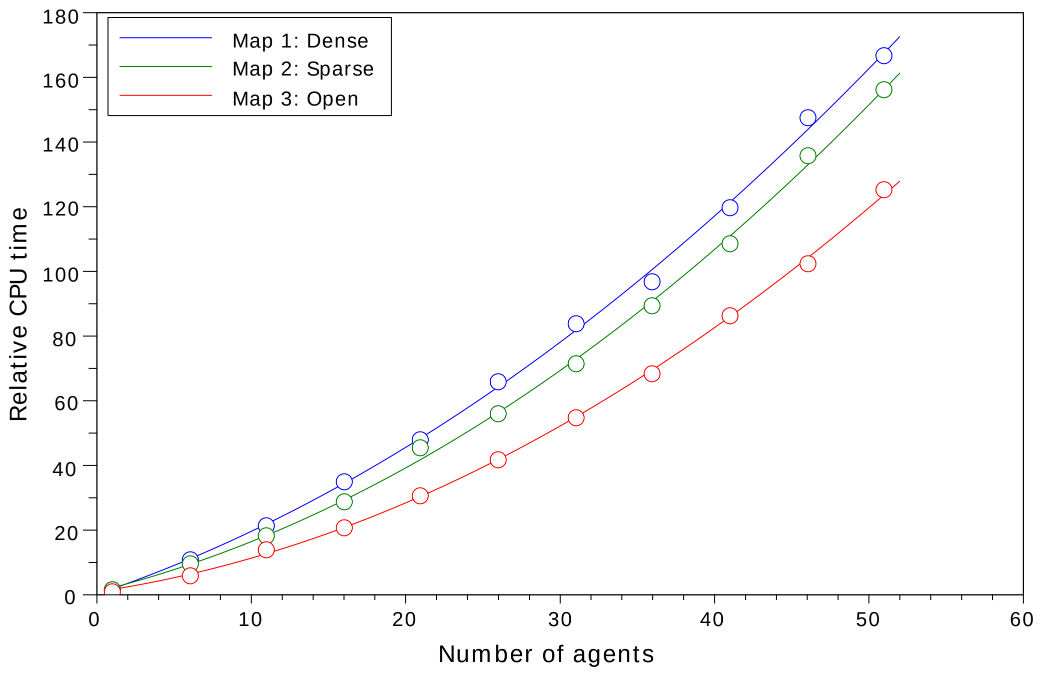


Figure 4.13: Time complexity of the total world loop as a function of agent number, for three different maps. Trend lines are least-squares quadratic polynomials.

cost of the entire world loop for these simulations is shown in Figure 4.13.

Again, the trend lines used are quadratic polynomials, which give an acceptable fit. It is expected that the total world loop cost would be $O(n^2)$, since the contact model has already been shown to be $O(n^2)$ and we can expect that the sensor-based calculations scale with n^2, since each agent's sensors check all other objects in the world. A notable result on the graph is that the complexity for the total world loop (y-axis) is significantly larger than for the contact function alone. In order to gain a better understanding of the fraction of the world loop computation spent on the contact function, the cost of the contact function and total world loop for 'Map 1' are shown together on Figure 4.14 on the next page. Note that the raw CPU time values for the contact function and total world loop were obtained and then normalised relative to each other, in order to make the values comparable.

It is clear from this graph that the complexity of the contact function as a proportion of the world loop's complexity decreases with increasing agent number. This can be explained by the fact that, as each additional agent is added, there is an additional object in the world for every other agent to sense, but there are also s additional sensors, which add to the cost of the world loop.

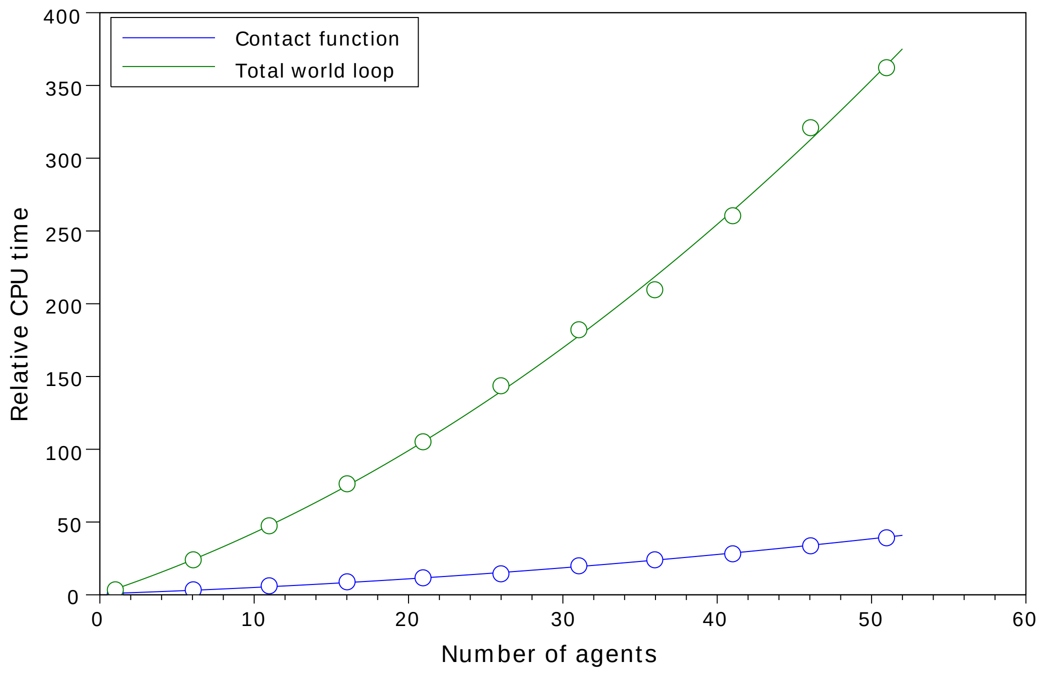


Figure 4.14: Time complexity of contact function and total world loop as a function of agent number. Trend lines are least-squares quadratic polynomials.

Tests similar to the ones above can be executed to understand the effect on the simulator's performance of varying the number of sensors per agent. If simulations of varying s value are performed using identical maps and agent control strategies (constant M, c), and running each simulation for a large number of world loops, then

$$\overline{\tau} \approx \tau = f_3(n, s)$$

A number of simulations were carried out for varying values of (n, s). This performance test is particularly intensive, and in order to reduce computational time only $s \in \{3, 6, 9\}$ was considered. In order to eliminate the influence of X, each (n, s) pair was simulated for 3000 world loops, as before. 'Map 1' was used. The results of the test are presented graphically in Figure 4.15 on the following page.

As might be expected, increasing the number of sensors per agent increases the computational cost, and the complexity remains $O(n^2)$ irrespective of the number of agents. It seems reasonable to expect that, for a given number of agents on the map (constant n), the world loop complexity would increase near-linearly with s. This is because, as shown in Figure 4.14, the complexity of the contact function (which is independent of s), contributes a relatively small fraction of the total world loop cost. In order to verify this

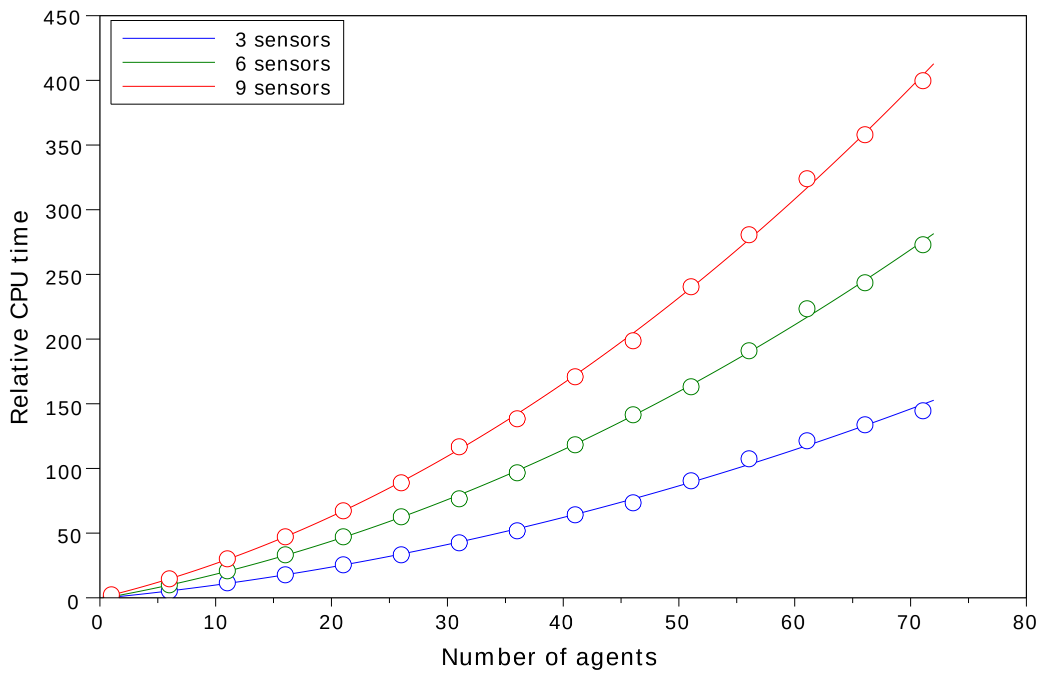


Figure 4.15: Time complexity of the world loop as a function of agent number and sensor configuration. Trend lines are least-squares quadratic polynomials.

hypothesis, Figure 4.16 on the following page shows the same data as Figure 4.15, but indexed by the time complexity for the 3-sensor configuration (the line for the 3-sensor configuration is included to clarify how the data is obtained).

It is apparent that past a certain threshold number of agents (approximately $n = 5$), increasing the number of sensors per agent while keeping the number of agents constant has an almost linear effect on the complexity of the world loop. (That the relationship is not completely linear can be explained by the inclusion of the contact function complexity, which does not change at all with the number of sensors). Additionally, this linear relationship between number of sensors and total cost is the same regardless of the number of agents. It should be noted that these results pertain only to the particular control algorithm and map used, but it is not unreasonable to expect that a similar relationship would hold for other cases. However, if each agent's control algorithm were to become computationally expensive (as is the case with learning strategies presented later in this report), it should be expected that the world loop complexity will begin to show a more linear relationship with n. This is because the time complexity of the control algorithm (which clearly is proportional to n) would be so large as to make the cost of the contact and sensor algorithms unnoticeably small.

The consequence and utility of the above analysis is as follows: all else held constant, the

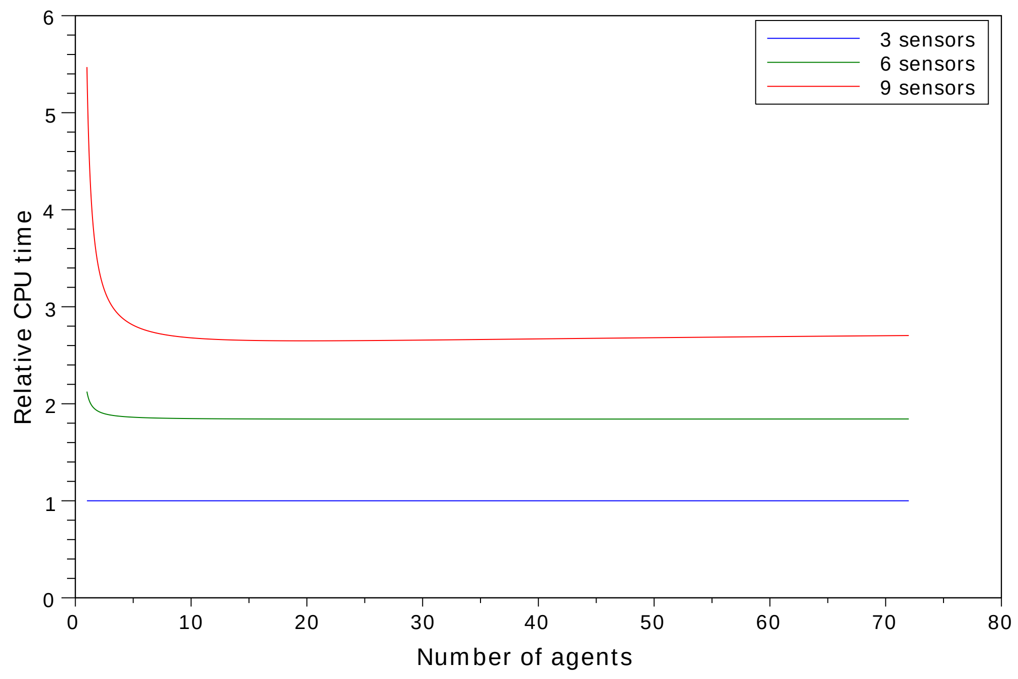


Figure 4.16: Time complexity of the world loop as a function of agent number and sensor configuration, relative to the time taken for the 3-sensor configuration.

complexity of the simulator increases linearly with the number of sensors per agent, and quadratically with the number of agents. It was also demonstrated that the geometric properties of the map can have a significant on the time complexity of the simulator's world loop. While the effect of varying the number of sensors per agent may not be of practical concern, the scalability of the simulator with respect to the number of agents certainly is, given that the purpose of the simulator is to model large swarms. The complexity of $O(n^2)$ is largely due to the expense of the naive algorithm that has been implemented for contact detection and sensor operation, which considers contact between two objects even if they are far apart. Highly optimised approaches exist that are significantly cheaper from a computational perspective, such as spatial partitioning, which involves fragmenting the map space up into separate regions and only considering contact between objects within a particular region [37].

The contact detection algorithm that was implemented, while not optimal, was considered acceptable for use in this project, primarily because of the simulator's ability to separate core simulation and visualisation. Had this feature not been implemented, it would likely have been necessary to pursue a computationally cheaper contact detection approach, at the expense of time that was spent on the work that follows.

The job files for the performance tests shown in this section may be found in the package directory.

4.9 Map Creation

The map format that was created for use by the simulator allows creation of rectangular and circular map elements by specifying their size and position. A typical map specification is shown in Listing 4.1 (this particular specification corresponds to the map shown in Figure D.1 on page 149).

Listing 4.1: Map data format used by the simulator.

```
map = [Obstacle("rec", [[0,0],[20,600]]),
       Obstacle("rec", [[0,0],[800,20]]),
       Obstacle("rec", [[780,0],[800,600]]),
       Obstacle("rec", [[0,580],[800,600]]),
       Obstacle("rec", [[70,70],[110,530]]),
       Obstacle("rec", [[110,280],[400,320]]),
       Obstacle("rec", [[220,0],[260,150]]),
       Obstacle("rec", [[480,0],[520,500]]),
       Obstacle("cir", [[650,160],30]),
       Obstacle("cir", [[650,360],30]),
       Obstacle("cir", [[300, 450], 50])]
```

Rectangular elements are specified by the coordinates of their upper-left and lower-right vertices; circular elements by their centre and radius. While this format is simple by design, it is still laborious to construct a map in this way. To expedite the process of map creation, an additional program was written for the simulator package that allows importation of a Scalable Vector Graphics (SVG) image into the native map format. This program is in the file `mapImporter.py` in the main package directory.

The SVG image format is compatible with the eXtensible Markup Language (XML) [38]. This means that SVG file contents are encoded in plain text, rather than binary. The map importer program works by performing a regular expression string search-and-replace on the SVG file, converting its contents to the native map format. In the current version, it recognizes rectangular and circular graphics primitives, and converts them into the corresponding map obstacles. It is also able to recognize 'fire' objects (introduced in Chapter 9).

Chapter 5

Behaviour-based Control

5.1 Introduction

The first type of control algorithms that were implemented using the simulator were those where agent behaviour was pre-programmed. That is to say, agents were not given the ability to learn from their environment, but were rather programmed to react to sensory inputs in a systematic way. This approach is generally known as *behaviour-based control* [1], referring to the fact that overall agent behaviour emerges from the internal interaction between several pre-defined basic behaviours (such as avoiding collisions, finding a specified target, exploring the map, and so forth) [39]. The actions produced by each of these basic behaviours are a function of the environmental condition at any given time.

In the broadest possible terms, each basic behaviour generates a potential action, which is the action that would be taken by the agent if that basic behaviour were its only behaviour [1]. Some coordination mechanism is then responsible for coordinating these basic behaviours into a single overall behaviour for the agent. One possible way of categorising such coordination mechanisms would be based on whether basic behaviours compete for the overall agent behaviour, or whether they work in cooperation with each other. These approaches are known as *competitive* and *cooperative* methods, respectively.

Cooperative methods allow each individual basic behaviour to contribute towards the overall agent behaviour, with different strengths. Thus in an abstract sense, the overall behaviour may be considered a weighted average of the basic behaviours. Arkin [40] proposes such an approach, based on vector summation of the basic behaviours. By contrast, in competitive methods, basic behaviours compete with each other for the overall agent behaviour at any given time. A competitive method that has proven to be effective for a number of robot control tasks is the *subsumption architecture* originally proposed in 1986 [39]. In a subsumption architecture, behaviours are ordered in a layered fashion with more important behaviours taking preference over those less important.

In terms of behaviour-based control methods, the subsumption architecture was the primary approach taken in this project. Thus, the remainder of this chapter will focus on the subsumption architecture. First, its implementation will be more fully described, followed by a detailed description of various subsumption-based control strategies related to mobile navigation and swarm behaviour. Finally, the results of simulations using these control strategies will be presented and discussed.

5.2 Subsumption Architecture

5.2.1 Description

The design of mobile robot controllers can be a challenging task because of the dynamical, non-linear interactions between each agent and its environment [1]. The problem lies in the fact that an agent's sensory input at any time is determined by the environment it is in, as well as the previous actions taken by the agent. Classical approaches to solving this problem have been based on the agent first perceiving its environment, planning which actions to take and finally executing those actions [1]. However, these approaches have had limited success for various reasons [1].

The subsumption architecture attempts to overcome the problem of mobile robot control by decomposing overall agent behaviour into a series of independent behavioural layers [39]. Each layer corresponds to a particular basic behaviour, which provides a direct mapping from sensor inputs to motor outputs. Behaviours can also be designed to be active or inactive under certain conditions. For example, a collision avoidance behaviour may only be active when the agent is near an obstacle or wall (or more precisely, when the agent's sensor inputs indicate an obstacle is nearby). Layers are ordered by importance, as shown in Figure 5.1 on the following page.

Coordination of behaviours works as follows: starting with the most important behaviour and moving to the least important, the first behaviour that is in an active state is chosen to be the overall behaviour and subsumes all others. For example, in the architecture shown in Figure 5.1, the controller would first check if the collision avoidance behaviour is active, as it is the most important. If active, that behaviour subsumes the others and becomes the overall behaviour of the agent. If inactive, the next most-important behaviour is checked, and if active, executed. This process continues until an active behaviour has been found. If no behaviours are active given the current sensory input, no action is taken.

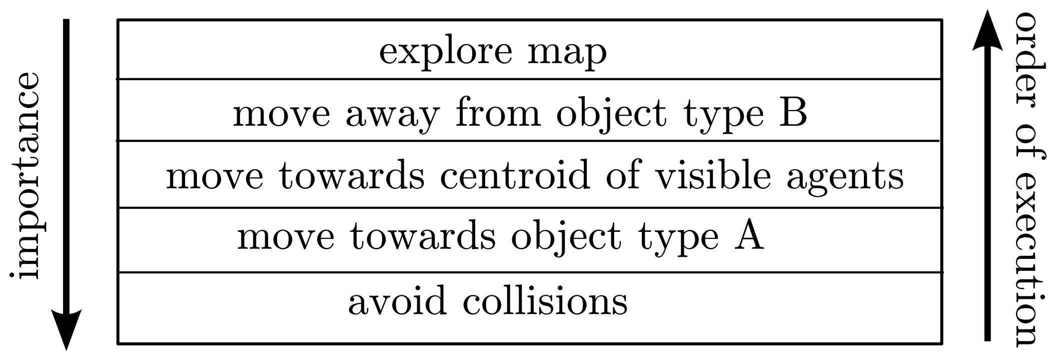

importance
explore map
move away from object type B
move towards centroid of visible agents
move towards object type A
avoid collisions
order of execution

Figure 5.1: A possible subsumption architecture showing importance of behaviours and flow of execution.

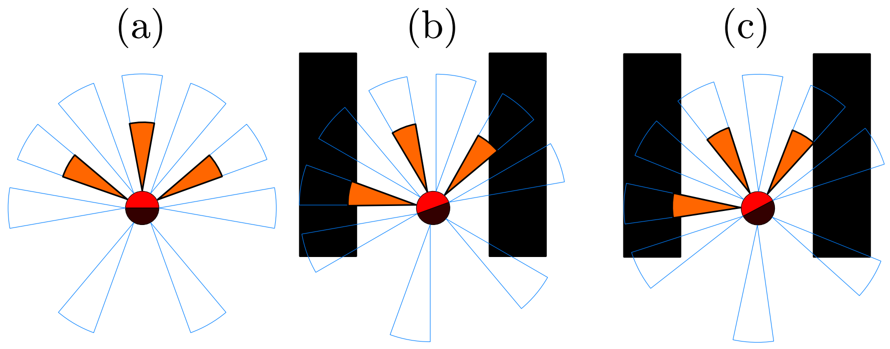


Figure 5.2: Agent diagrams showing collision avoidance cases. (a) A typical agent sensor configuration, showing (in orange) the areas in which an obstacle will activate the CA algorithm. (b) A sensor activation conflict. (c) Deadlock position resulting from the conflict shown in (b).

5.2.2 Collision Avoidance

The problem of avoiding collisions between agents and obstacles can be solved in a number of ways, most of which are dependent on the agent's sensory configuration. In this section an explanation will be given of the particular collision avoidance (CA) behaviour which was found to be sufficient for use throughout this project. However, it is noted that different CA algorithms may give different results.

The CA algorithm used is a basic one, designed for the agent sensor configuration shown in Figure 5.2, but can be adapted to work on any agent with three forward-facing proximity sensors. Essentially, if the reading on a particular sensor is below a minimum specified value, the agent attempts to avoid the obstacle. A sensor will be called 'activated' if it returns a reading lower than the minimum specified value, meaning that it is detecting an object. The areas within which an obstacle must lie for this to occur are shown as orange segments in the figure. For the side sensors, avoidance is achieved by turning away from the obstacle until the obstacle is no longer within the minimum distance. This can be performed using either a stationary (agent turns ‘on the spot’) or translational (agent turns and translates) turn, but with different effects (discussed below).

For the front sensor, a number of methods of moving away from an obstacle were trialled. The initial method used was to force the agent to move directly backwards for a specified distance, and then turn either left or right, based either on a random decision or a pre-defined tendency. This requires putting the agent into a state of *non-sensory execution*, during which time its actions are not being made based on the state of its sensors, but rather based on a previous decision, which can have undesirable effects such as collisions. Instead, it was decided that, similar to the case with the side sensors, an obstacle within range on the front sensor should cause the agent to turn. Initially the direction of turn was chosen randomly at each discrete event during which the front sensor was found to be active. However, the consequence of this was that left and right turns were executed with approximately equal frequency, resulting in no net turn. As a solution to this problem, it was decided that each agent should have a pre-defined ‘turning tendency’, being either left or right. Under this regime, whenever the front sensor is activated the agent simply turns in the pre-defined direction, until the front sensor is no longer activated.

It is apparent that a conflict may arise if two sensors are activated at the same time, for example while navigating a narrow corridor as shown in Figure 5.2 (b). The agent’s left and right sensors are both detecting obstacles within the minimum distance and are thus both attempting to turn the agent. Which action is taken will depend on the specific way the CA algorithm has been implemented. In the current implementation, the right sensor is checked after the left sensor, meaning that the action of turning left will be the one that persists. However, once the agent has turned, a further difficulty arises, shown in Figure 5.2 (c). In this new state, only the agent’s left sensor detects an obstacle, so the agent turns right, bringing it back into state (b). If the agent is making stationary turns the net effect will be that agent oscillates indefinitely around state (c). Note that a similar, but symmetrical, scenario would occur if the CA algorithm checked the left sensor last.

There are a number of ways this problem may be solved. The CA algorithm may be written so as to disregard the side sensors if they are both activated, in which case the agent would ‘zig-zag’ through the corridor, turning each time its front sensor is activated. The chosen solution was to use translational turns instead of stationary turns, meaning that the agent would still exhibit the oscillation described above, but at the same time would move forward and eventually out of the corridor.

The behaviour of an agent following the CA algorithm can vary significantly depending on the threshold distance at which avoidance is performed, as well as which sensors are checked for obstacles. This behaviour can, depending on the circumstances, be undesirable. For example, if the diametrically-opposed left and right sensors are checked for obstacles, the agent will turn a greater angle to avoid such obstacles. This may lead to poor map traversing behaviour or other undesirable effects.

5.2.3 Swarm Behaviours

Once the basic behaviour of collision avoidance had been implemented, the subsumption architecture was extended to include higher-order behaviours that would give agents the ability to form swarm-based geometries. Two such geometries were implemented: static agglomeration and the formation of chains ("leader-following").

These simulations were implemented using decentralised coordination, with unshared local information, in the absence of communication and internal representations about the environment. As mentioned in Section 4.3.2.1 on page 33, agents' IR sensors are able to distinguish between obstacles and other agents. This ability is critical to the success of swarming behaviour. Without the ability to distinguish directly between agents and obstacles, such a distinction may have to be inferred from whether an object is perceived as stationary or moving (under the assumption that obstacles are stationary), which is a significantly harder task.

For swarming control strategies, two quantities describing the relative positions of other agents are consistently useful: the position of the centroid of visible agents and the position of the nearest neighbour (NN). When calculating these quantities there is the additional possibility of including data only from particular sensors (e.g. forward-facing sensors), to obtain the desired behaviour. Assume an agent has n sensors returning distance readings of $d_1, d_2, \ldots, d_n$ with positions in the agent's local coordinate system of $\mathbf{c_1}, \mathbf{c_2}, \ldots, \mathbf{c_n}$ and orientations in the agent's local coordinates of $\theta_1, \theta_2, \ldots, \theta_n$. If we denote the unit vector in the θ direction as $\hat{\theta}$, then it is possible to define the centroid of all visible agents, $\bar{\mathbf{x}}$, using a vector average, viz.

$$\bar{\mathbf{x}} = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{c_i} + d_i\hat{\theta}_\mathbf{i}) \qquad . \tag{5.1}$$

Note that $\bar{\mathbf{x}}$ is in the agent's local coordinate system. Similarly, we might define the position of the nearest neighbour in the forward direction, $\mathbf{x_{NN,f}}$, as

$$\mathbf{x_{NN,f}} = \mathbf{c_t} + d_t\hat{\theta}_\mathbf{t} \qquad t = \operatorname*{arg\,min}_{y\in\{p\,|-90^\circ<\theta_p<90^\circ\}} d_y \tag{5.2}$$

Further detail on the agent's local coordinate system is given in Appendix A.4 on page 138.

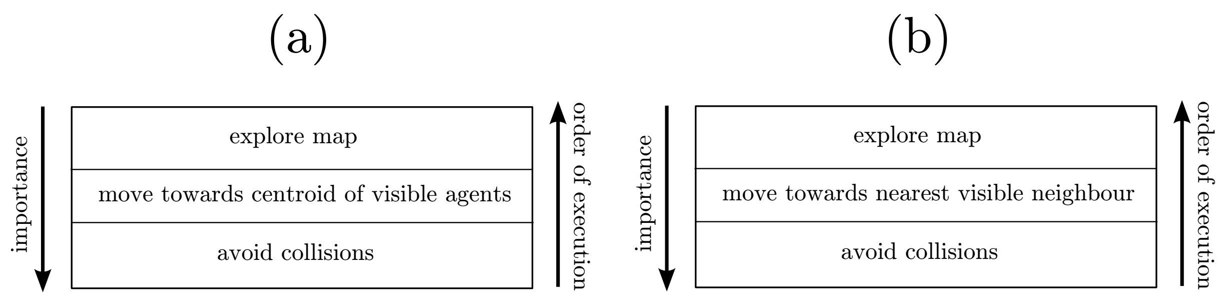


Figure 5.3: Schematic showing subsumption-based control strategies used for swarm behaviour tasks. (a) Agglomeration strategy. (b) Chain formation strategy.

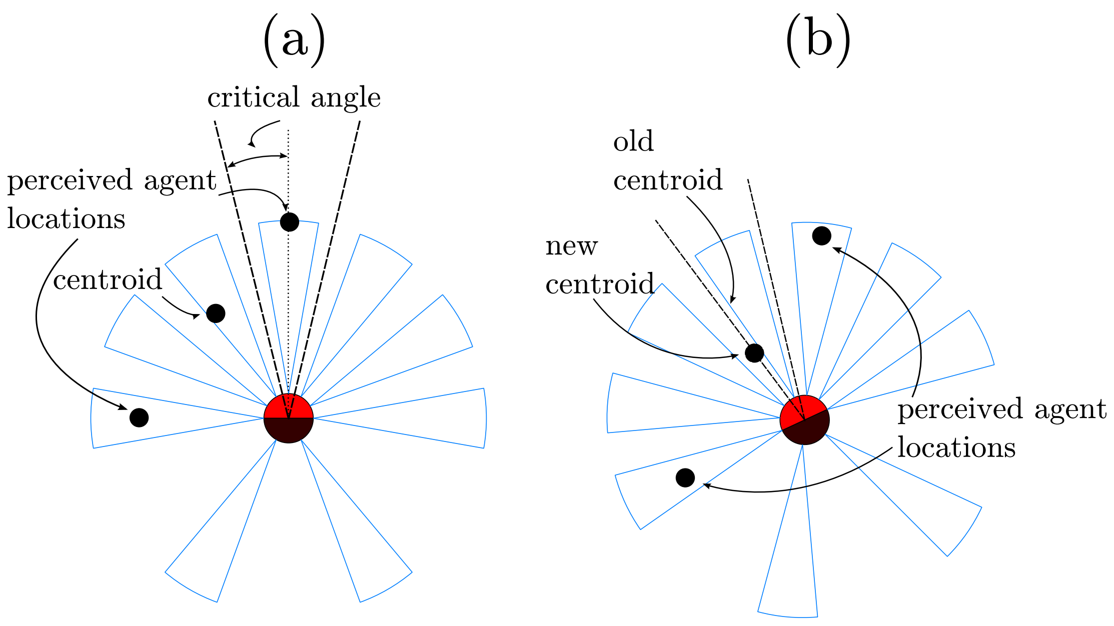


Figure 5.4: Method of moving towards centroid of visible agents in an agglomeration strategy.

5.2.3.1 Agglomeration

In order to implement agglomeration behaviour, each agent was programmed with a three-layer subsumption-based control algorithm, as shown in Figure 5.3 (a).

Essentially, agents would explore the map while avoiding collisions, but if their sensors detect any other agents, then immediately move toward the centroid thereof. The centroid is calculated by means of equation 5.1. This strategy has the merit that no explicit planning is required of where to agglomerate, nor is it necessary for agents to know whether they are in the centre a swarm or on the perimeter; their actions are the same regardless. In order for agents to move toward the centroid, they are programmed to turn toward the centroid if it lies more than a critical angle away from their forward orientation, as illustrated in Figure 5.4.

If the centroid lies within that critical angle then the agglomeration layer of the subsumption architecture is no longer active, and higher-order behaviours are evaluated. Alternatively, the agent could be instructed to only continue with higher-order behaviours (such as exploration) if the distance to the centroid exceeds a critical value (this will be discussed in Section 5.3.2).

5.2.3.2 Chain formation

Chain formation can be a particularly useful swarm behaviour, as it allows completion of tasks that may not be possible by agents acting individually. The greatest challenge in designing a control algorithm to produce emergent chain formation is how to decide which agent should become the leader of a particular group. The leader is free to move in any direction and thus choose the trajectory the group, whereas those agents behind the leader are required to follow. Clearly, if centralised coordination were present, the problem of chain formation becomes almost trivial.

Even if centralised coordination is not possible, if agents are able to communicate with each other it would be possible to establish leadership on the basis of this communication. For example, if two agents approach one another and it is not clear which should lead and which should follow, they could each generate a random number, communicate their number to the other agent, and the agent having chosen the lower number then begins to follow the other agent.

In the absence of central coordination and inter-agent communication, a control structure is therefore required that allows agents to sometimes follow and sometimes explore freely. Such a control algorithm is easily designed by means of a subsumption architecture, as illustrated in Figure 5.3 (b). Again, with collision avoidance as the base behaviour, agents explore the map until they detect one or more other agents, at which point they move towards the closest of these agents, but only if said agent is positioned in front of the first agent. This last condition is crucial as it ensures that all agents within a chain will only follow those that are ‘in front’ of them (where ‘in front’ means ‘ahead of the direction of the travel’). If this condition is not enforced, agents will not form chains but will instead agglomerate, although in a more dispersed fashion than with centroid-based agglomeration because each agent only approaches its nearest neighbour.

The middle layer of the subsumption architecture becomes active whenever another agent is detected. When this layer is active, the agent calculates the position of its nearest neighbour by means of equation 5.2. Once the position of the nearest neighbour has been computed, the agent turns toward this position in a fashion identical to that for centroid-based agglomeration, shown in Figure 5.4 on the preceding page.

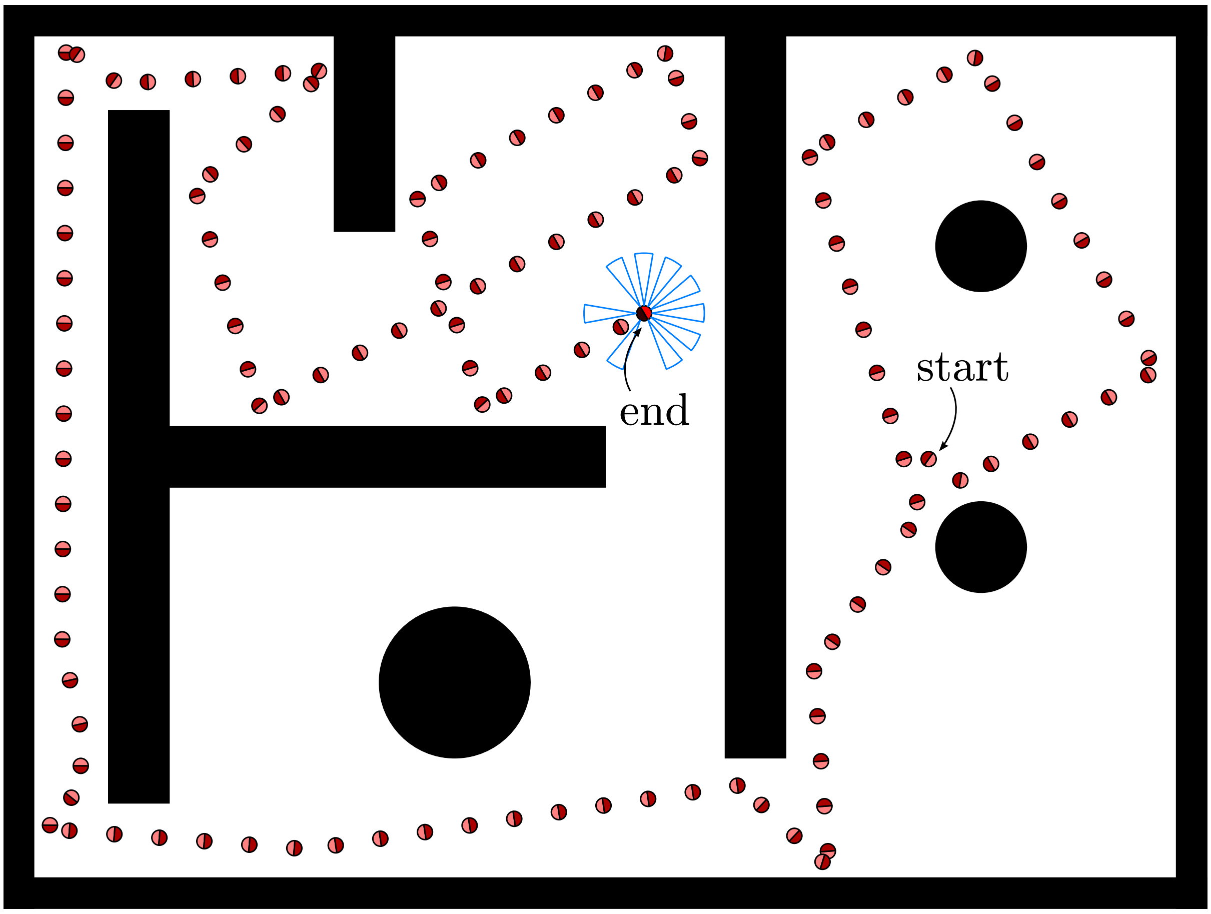


Figure 5.5: Simulation trace of a single agent employing a subsumption-based map exploration algorithm with collision avoidance.

5.3 Simulation Results and Discussion

While the previous section presented and discussed the theory of the various behaviour-based approaches that were implemented, this section will focus on the problems encountered and results obtained during actual simulations. Each of the three behaviours previously described will be considered, namely collision avoidance and the swarm behaviours of agglomeration and chain formation.

5.3.1 Collision Avoidance

The results of a simulation using the CA algorithm described in Section 5.2.2 are shown in Figure 5.5. A two-layer subsumption architecture was used, with a base behaviour of collision avoidance (most important) and the higher-order behaviour being map exploration in a straight line.

The agent avoids all obstacles and eventually reaches all areas of the map, despite the

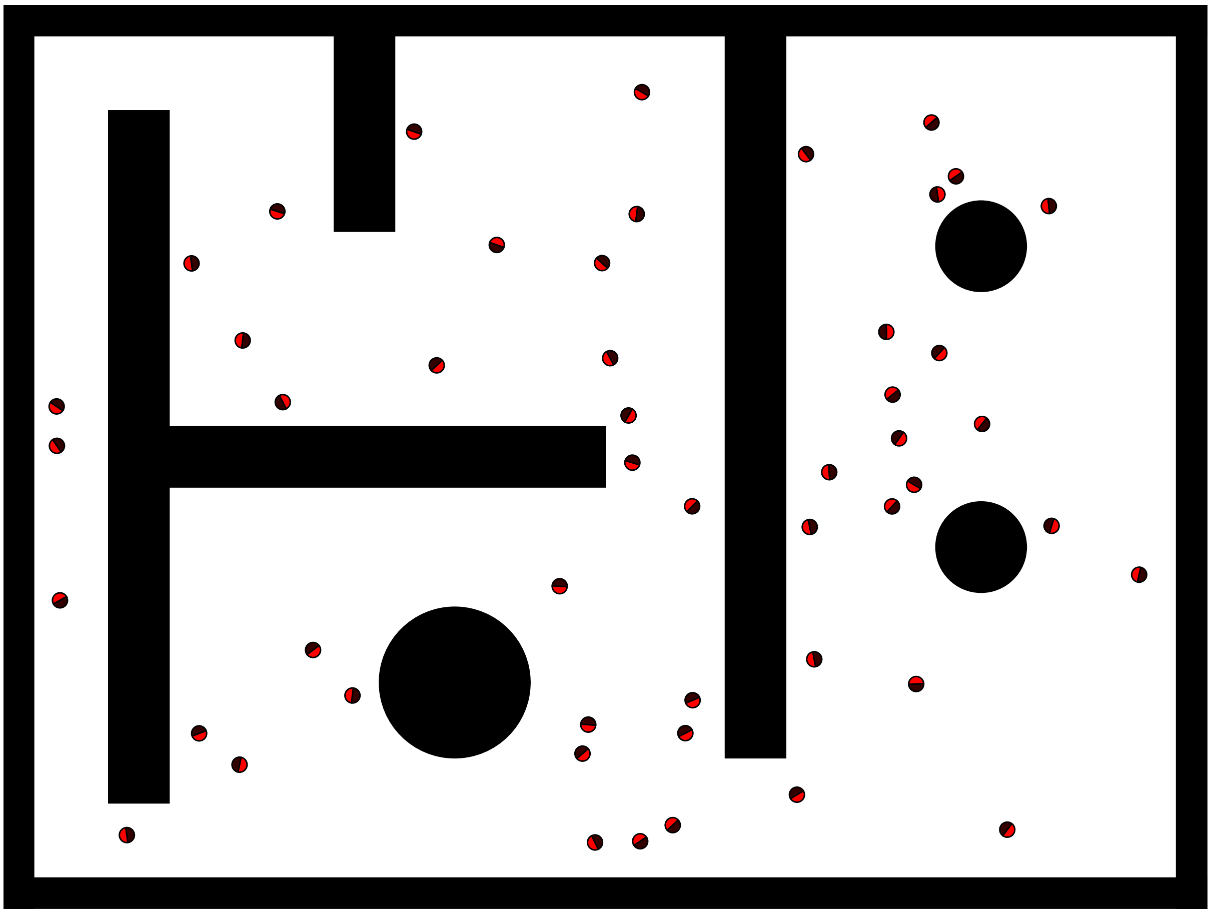


Figure 5.6: Randomised initial agent positions for an agglomeration simulation with 50 agents.

presence of narrow corridors. If desired, the agent's ability to explore the map could be improved, for example, by instructing it to make random turns at specific intervals, rather than always moving in a straight line.

5.3.2 Agglomeration

A simulation of the agglomeration algorithm was performed using a critical angle of 10 degrees on either side, meaning that if the computed centroid of visible agents did not lie within 10 degrees of a particular agent's forward direction, the agent would turn toward the centroid. Agents were initially randomly placed on a map, and allowed to agglomerate. To assist with diagnosing agent behaviour, grey lines are drawn from each agent to the centroid of all the agents it can sense.

The randomised initial agent positions are shown in Figure 5.6, and the state of the simulation after 1000 and 3000 time steps are shown in Figure 5.7 and 5.8, respectively.

Before analysing the behaviour that was observed, it is worth discussing some of the

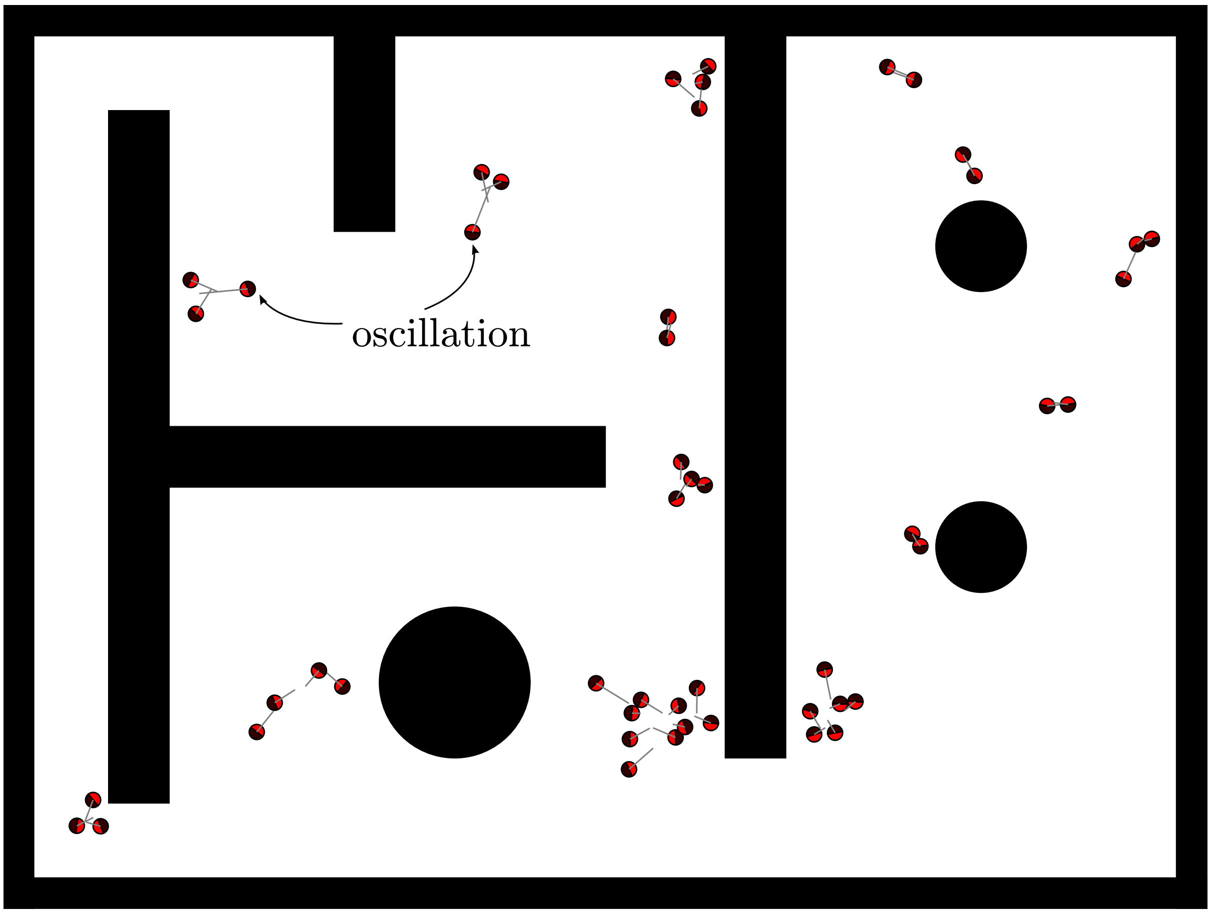


Figure 5.7: Results of an agglomeration simulation with 50 agents after 1000 time steps. The grey lines emanating from agents indicate each agent's perception of the centroid of other agents.

problems detected during the simulation.

After running and observing simulations using the agglomeration control strategy a number of times, a number of shortcomings became evident. In the original subsumption-based agglomeration algorithm described in Section 5.2.3.1, the only circumstance under which the agglomeration layer would become inactive (i.e. pass control on to higher-order behaviours) is if the computed centroid lay within the specified limits. However, it is possible that, under certain circumstances, it may never actually occur that the agent reaches a state where the centroid lies within the specified angle.

This anomaly is due to the information provided by the sensors being incorrect, for two reasons: the agent's incomplete sensor field and the discretised nature of the sensors. The former is self-explanatory: it will almost always be the case that there are 'blind spots' in an agent's sensory field, meaning that a slight change in the agent's orientation can result in discontinuous changes in the agents that are sensed. For example, if the agent shown in Figure 5.4 (a) were to rotate slightly to the left, the agent at the top

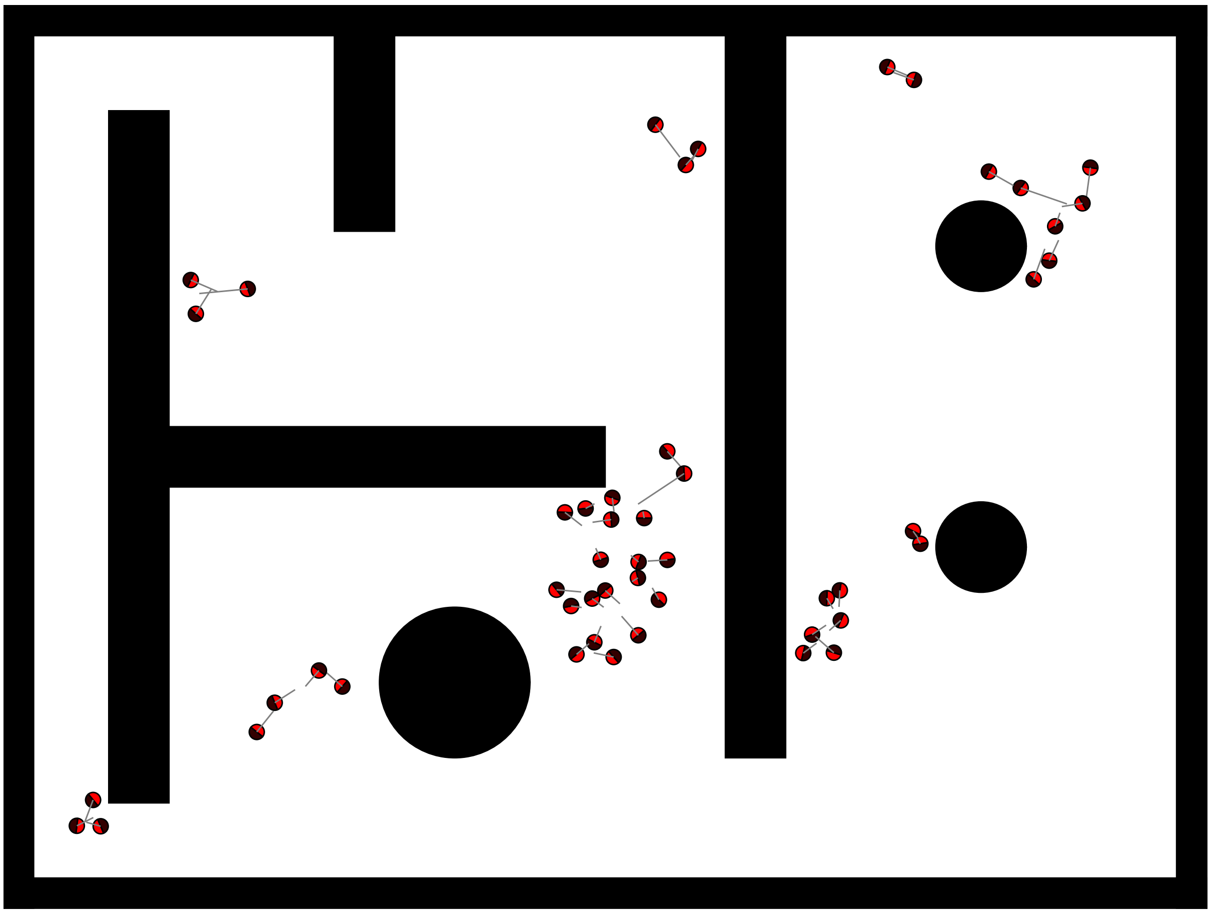


Figure 5.8: Results of an agglomeration simulation with 50 agents after 3000 time steps. The grey lines emanating from agents indicate each agent's perception of the centroid of other agents.

of the figure would no longer be sensed, and the first agent's perception of the centroid would move considerably. The second factor responsible for an agent not being able to align itself with the group centroid is that the sensors are discrete. What is meant by this is that each sensor only returns the distance to the nearest object it detects, with which information the agent's controller reconstructs the position of the corresponding object that was detected, using knowledge of the sensor's position and orientation in local coordinates. It follows that any object detected by a particular sensor at a specified distance will be perceived by the agent as being at the same position, regardless of where the object was positioned within the sensor's angular range.

The result of these inaccuracies is that, as an agent turns in an attempt to align with the centroid, its perception of the centroid changes, as shown in Figure 5.4 (b). If the new centroid position is on the other side of the agent's forward-facing axis than the old centroid position, the agent will oscillate between these two positions, all the while never releasing control from the agglomeration layer. The way this problem manifests in agent behaviour is that an agent will position itself at a comparatively large distance from the rest of the group, and oscillate between two states. This can be seen in Figure 5.7 on page 63, marked "oscillation".

Another minor problem was observed. When an agent had correctly aligned itself with its perception of the centroid position, the agglomeration layer would become inactive and flow of execution would transfer to the exploration behaviour. The result is that the agent would move towards the centroid until the collision avoidance behaviour became active, resulting in oscillatory behaviour. As a remedy for this, the agglomeration algorithm was modified so that if the centroid lay straight ahead of the agent within a specified distance, the agglomeration layer would become active and instruct the agent to remain stationary. This was generally found to be effective.

It was initially expected that the agglomeration subsumption architecture would result in all agents agglomerating into one large group. However, as shown in the simulation results, many groups of varying size emerge. In hindsight it is trivial to explain such behaviour: as soon as an agent is aligned with the centroid of its visible neighbours, and come to be stationary, it will not move unless the position of the centroid moves. The centroid position will only move if the other visible agents in the group move. Thus it is possible that a form of local convergence emerges, in which agents form groups of 2 or 3.

In order for global agglomeration to be achieved, the algorithm would need to be modified to allow groups to move once they have formed. Such group movement does occur in the present algorithm, but its occurrence is not guaranteed by design. The behaviour of groups moving together can occasionally emerge from the interaction between the subsumption layers activating at unexpected times, as well as the fact that sensors are relatively sparse on the posterior of the agent, meaning that it is possible for individual agents to inadvertently 'lead' a group because the group is not visible to the agent. However, if any of the agents in the group move into the leader's sensory vision, the leader will turn around and move towards the group.

While it is possible to observe and analyse a multitude of other problems and behaviours in this simulation alone, the core objective of agent agglomeration has been achieved. The very fact that these problems and unexpected behaviours were observed in simulation highlights the usefulness of a simulator: *a priori* analysis of control strategies may leave the analyst certain that a particular emergent behaviour will result, but it is only by simulation that unexpected behaviours are exposed. It is interesting to note that, while simple rules can lead to complex emergent behaviour, it is equally true that simple rules are not necessarily simple to analyze; this is, in essence, the reason for creating a simulator.

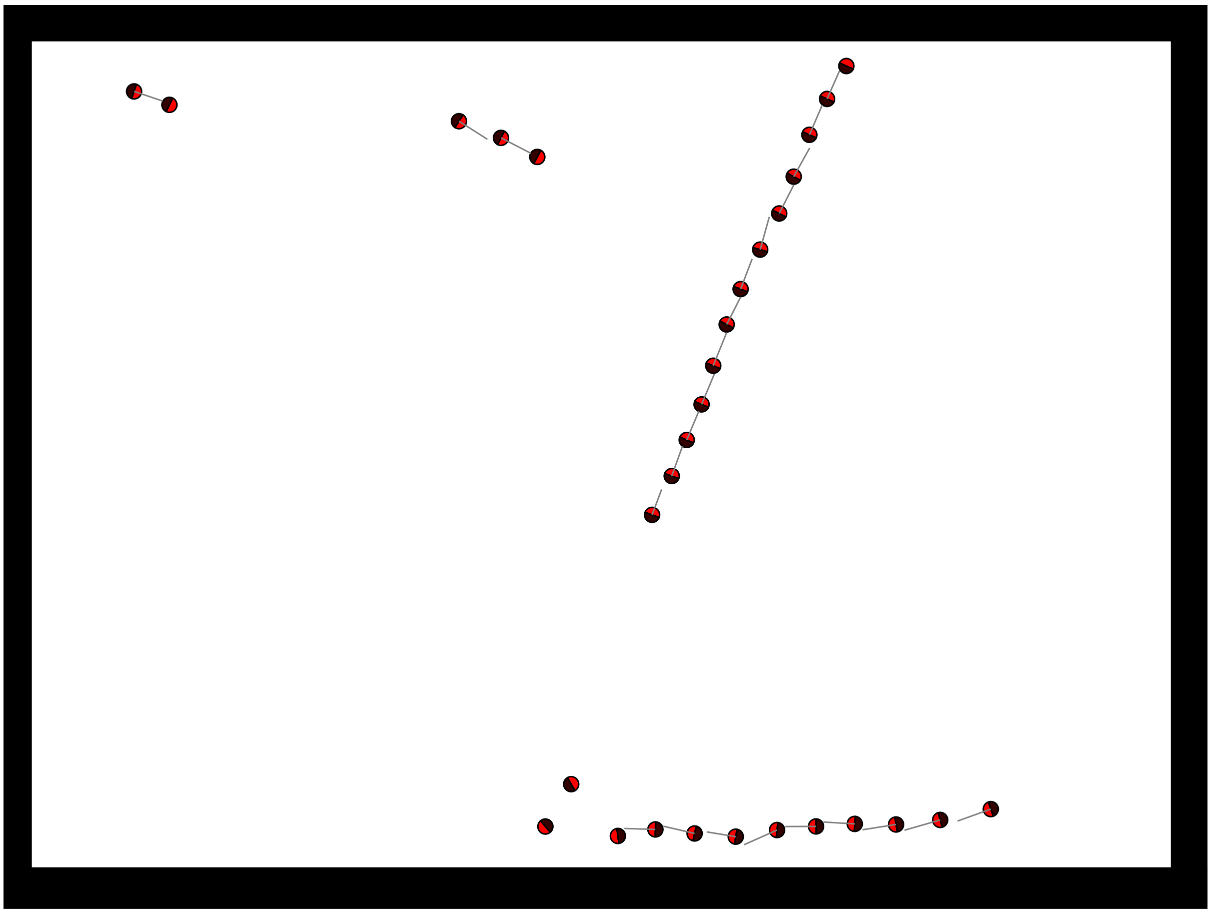


Figure 5.9: Screen capture from a chain formation simulation showing most agents having formed into groups.

5.3.3 Chain Formation

The results of a simulation (in file `chainFollowing.sim`) using the chain formation rules given above are shown in Figures 5.9 and 5.10. The simulations were performed on an enclosed map with no obstacles. This was done to allow longer chains to form, since the presence of obstacles results in the decomposition of chains.

These simulations show that chain formation rules lead to a greater degree of cohesion than for agglomeration rules. This is thought to occur for two reasons: chains being more dispersed than compact groups and the fact that once agents have assembled into a chain, they continue to move. That chains are more dispersed increases the chance that individual agents will find a particular chain, since a chain formation occupies a greater span of map space than the corresponding compact group. Additionally, once agents have assembled into chains, they continue to explore the map in their groups, leading to the merging of separate chains. This prevents the local convergence seen in the previous section where agents would stop moving once they had formed small groups.

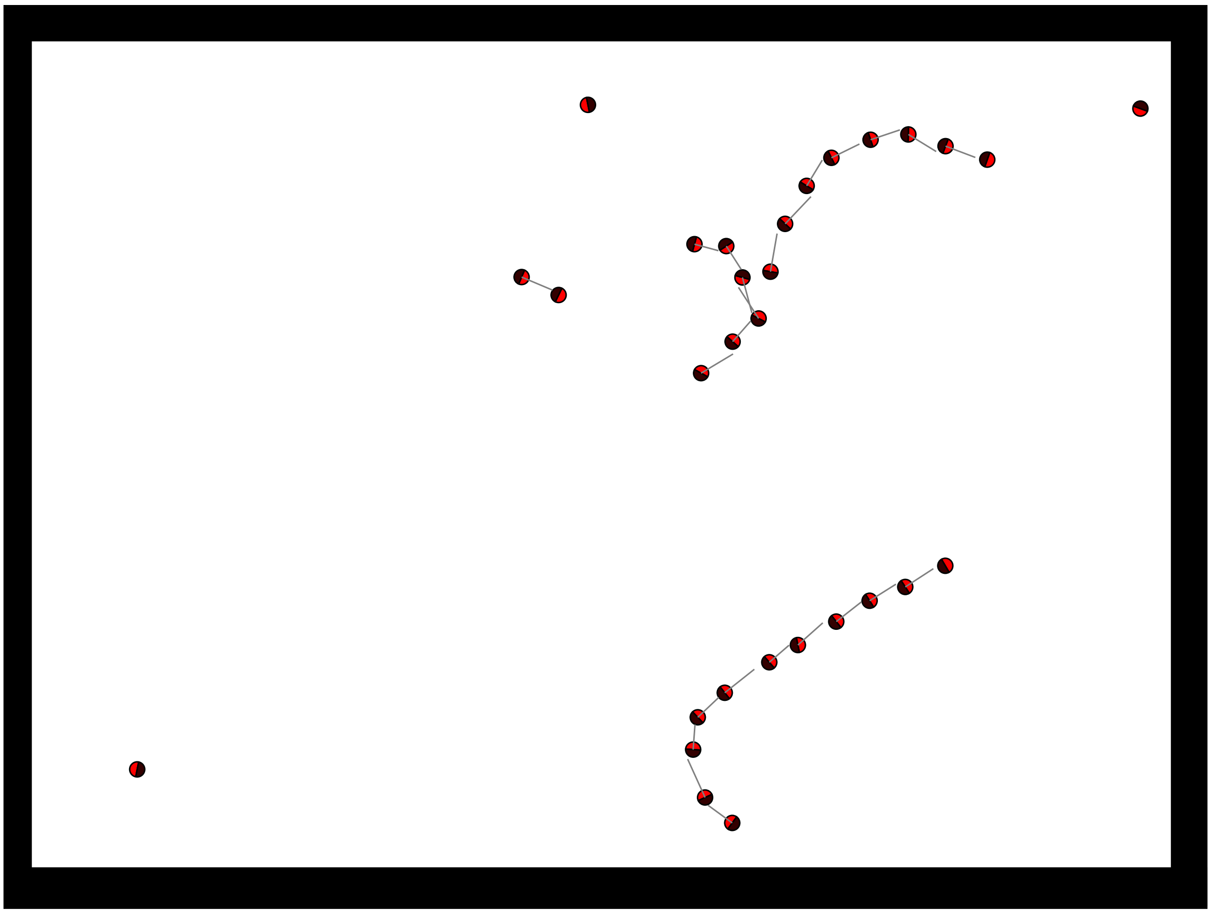


Figure 5.10: Screen capture from a chain formation simulation showing how chains can bend (bottom) and decompose (top).

Since all agents were programmed to move at the same speed, they would tend to follow other agents at a distance equal to the maximum range of their sensors. This resulted in chains decomposing more frequently, because only a slight discontinuity in the motion of an anterior agent was required for it to move outside of the field of vision of the posterior agent. The result is that the posterior agent would no longer follow the anterior agent, but rather continue to explore the map. This was particularly prevalent near obstacles (such as walls) where highly irregular behaviour is observed while chains undergo a process of reorganisation.

To solve this problem, the chain formation rule was modified so that once an agent senses another agent, it accelerates towards that agent until they are a specified distance apart. This means that the average spacing between agents in a chain is decreased, with the result being that greater irregularities in the behaviour of anterior agents are required for the chain to decompose; therefore, longer chains form.

From these simulations it was observed that agents following chain formation rules exhibit a number of emergent behaviours. For example, when the agent at the front of a chain approaches an obstacle, such as a wall, its collision avoidance behaviour becomes active and causes the agent to turn away from the wall. If the agent is able to turn away from the obstacle fast enough so that the collision avoidance behaviour of the posterior agent does not become active, the chain will simply change direction, intact. This was observed to occur when agents approach obstacles obliquely, as they are able to turn away from the obstacle in a relatively short time. However, if the front agent takes longer to turn away from the obstacle, the collision avoidance behaviour of the posterior agent becomes active. This results in the second agent turning away from the first. This was observed to either lead to the formation of two new chains, or to the former leader rejoining the original chain, but in a posterior position.

Other interesting emergent behaviour occurs when two chains meet, as shown near the top of Figure 5.10 on the preceding page. In the case shown, a chain of three agents is approaching a longer chain of eleven agents. Because the smaller chain is aligned opposite to the direction of the larger chain, the behaviour that emerges in this case is the formation of two new chains. However, if the smaller group were to be aligned with the larger group, it has been observed that a single larger chain will result.

The chain formation rules outlined in Section 5.2.3.2 were observed to be generally effective. However, one problem with these simple rules that was mentioned earlier is that, in the absence of communication, there is no explicit method for assignment of the roles of leader and follower. This is not a problem if agents approach each other from behind or obliquely, as agents only attempt to follow other agents detected in their forward sensors. However, if two agents approach one another 'head on' (i.e. at an angle of 180°), the result is that they will keep moving toward each other until their collision avoidance behaviours become active, at which point they turn away from each other and disperse. A number of solutions to this problem were contemplated but not implemented, such as each of the two agents performing a stationary turn at a randomly-selected speed. The agent which turns at the slower speed will have the other agent in its sensory field for longer, and thus become the follower once the faster-turning agent has moved away. This rule would thus be an attempt at using the environment to effect a form of implicit communication between agents. Further investigation of this phenomenon is recommended.

5.3.4 Other Swarm Behaviours

While only the swarm behaviours of agglomeration and chain formation have been implemented and shown here, it is certainly possible to recreate, using the simulator, other behaviours discussed in Section 3.2 on page 15. For example, simulation of a distributed resource allocation task which can only be accomplished by cooperative behaviour is presented in Chapter 9.

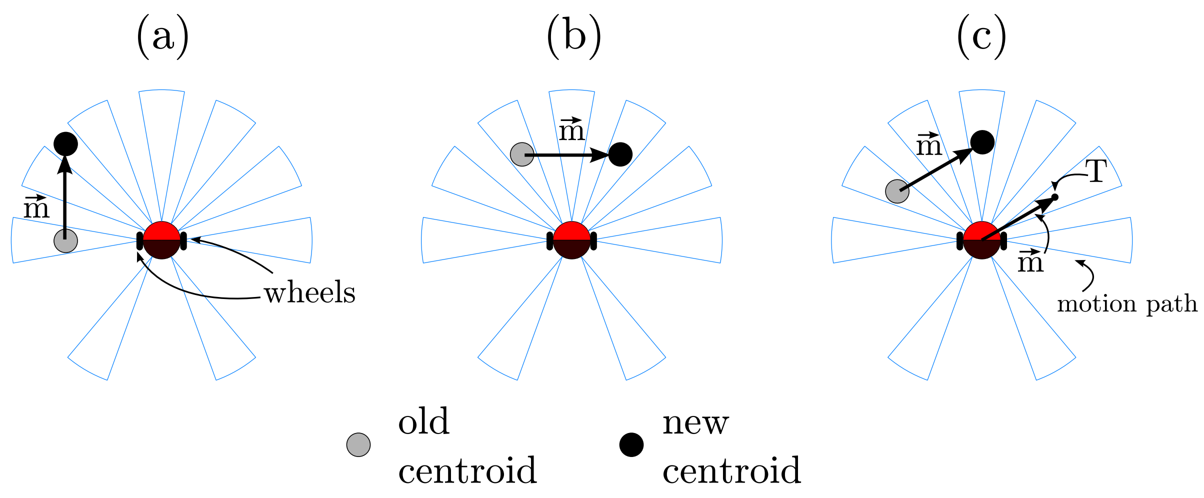


Figure 5.11: Schematic showing different velocity-matching cases for an agent with differential wheel locomotion.

The common swarm behaviour of *flocking* can be thought of as a modification of agglomeration where the the agglomerated group continues to move, rather than remain at rest. A simple rule that would effect this behaviour is for each agent to match the velocity of the centroid of its local neighbours, as shown by the Boids simulation presented in Section 4.2.3 on page 27. While the simulator is not inherently unable to perform such a simulation, agent configurations using differential wheel locomotion are not easily able to perform velocity matching. This is illustrated in Figure 5.11.

Suppose the agent in Figure 5.11 (a) in a particular time step perceives the local centroid to be at the position marked "old centroid" (grey dot) and at the next time step this position changes to that marked by "new centroid" (black dot). From the change in position of the centroid, the agent is trivially able to compute the apparent centroid velocity,

$$\vec{v} = \vec{m}\Delta t \qquad . \tag{5.3}$$

In order to match this velocity the agent need only move directly forward. The Δt appearing in equation 5.3 refers to the refresh rate of the agent's sensors which, as mentioned in Section 4.3.2.1, is assumed to be equal to the refresh rate of the discretised simulation. It is further assumed that the agent has knowledge of this sensor refresh rate and also knows the relationship between motor speed and wheel tangential velocity, thus enabling it to control its motors in such a way that it can match the velocity of an object detected by its sensors. These are not unreasonable assumptions as these quantities are fixed, physical properties of the robot and could thus be pre-programmed into the controller.

In contrast, consider the agent in Figure 5.11 (b) and the corresponding centroid positions. Because the agent is only able to move in the direct forward or reverse directions, it is unable to match the apparent centroid velocity.

Finally, suppose that the agent shown in Figure 5.11 (c) is attempting to match the centroid velocity shown. If we construct point T by adding the vector $\vec{m}$ to the agent's position, then the agent is able to match the centroid velocity if it can move to point T within a single time step. While unable to control its own instantaneous velocity to match the centroid velocity (because of the constraint of only being able to move forward or backward), the agent is able to set its motor speeds in a ratio that will result in its arc of travel passing through point T, thus making its average velocity over the time step equal to the apparent centroid velocity (details of how this is possible are given in Figure A.1 on page 127). While the same method could be applied by the agent shown in (b), it would result in the agent being aligned away from the direction of centroid travel.

This algorithm was not implemented in the project, but it is recommended that it be further investigated. Additional consideration should be given to issues of deadlock (when agents in a group do not move because the group centroid is not moving) and the effectiveness of the algorithm for cases similar to that shown in Figure 5.11 (b).

Chapter 6

Neural Networks

Neural networks are a class of mathematical models that attempt to map sets of inputs to outputs, using known data to approximate unknown data. Such models are more correctly known as *artificial* neural networks, to contrast them with *biological* neural networks, by which they were originally inspired. This chapter will introduce the basic theory of neural networks, followed by a description of the techniques used to train them. The mathematical basis of their operation will then be explored, together with a basic analysis of their numerical stability and accuracy.

6.1 Introduction

In general, an artificial neural network (ANN) is a collection of nodes connected by weighted links, as shown in Figure 6.1 on the next page. The way that ANNs are structured and perform manipulations on information are said to be analogous to the biological neural networks in animal brains [41], although the number of synapses in human brains is typically much larger (on the order of 10^{14} [42]). ANNs can be made to learn on a similar operational principle to animal brains: they are shown sets of training data in the form of input and output pairs, and are later required to reproduce the same output when given identical input data. However, this much can be achieved by simple lookup tables. The property of ANNs that makes them 'intelligent' is the fact that when they are given input data that is similar, but not identical, to inputs they have previously learned, they are able to approximate the correct output data. The accuracy of these inferences will be discussed in Section 6.4.

Following this biological similarity, the nodes in ANNs are sometimes called 'neurons' and the links between the nodes 'synapses', from their counterparts in BNNs. These terms are often used interchangeably. Essentially, neurons are used to transmit signals

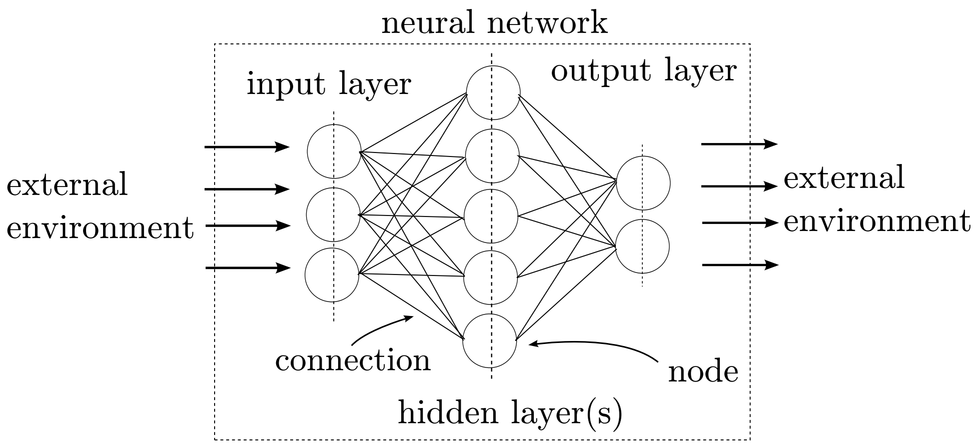


Figure 6.1: A simple feed-forward neural network with 3 input neurons, 5 hidden neurons and 2 output neurons.

from an input layer to an output layer, with the ANN's only interaction with the external environment occurring at the input and output layers, respectively. ANNs can be designed in several architectures, the simplest of which is the feed-forward architecture shown in Figure 6.1, where the input signals are transmitted forward from the input layer, through any hidden layers, to the output layer.

In 1943, McCulloch and Pitts [43] proposed a simple model for the operation of each neuron, which still sees extensive use today. In the model, each neuron has a number of inputs and a single output, with the output of the neuron y_i being a function Φ of the sum of all incoming links x_j weighted by the strength of the synaptic connection w_{ij}. This is depicted in Figure 6.2 on the following page and can be written mathematically as

$$y_i = \Phi(\sum_j^n w_{ij}x_j)$$

McCulloch and Pitts [43] originally proposed using the step function shown in Figure 6.3 (a), for which there is only non-zero output when the weighted sum of incoming signals exceeds a threshold (usually zero). Accordingly, the output function Φ is sometimes known as the activation function because it dictates when the neuron becomes active. Because of the binary nature of the step function, it is only able to transmit one bit of data, regardless of the amount of data it was provided (in the form of input signals). For this reason, it is sometimes necessary to make use of alternative activation functions. Other common activation functions [1] are linear ($\Phi(x) = kx$), logistic ($\Phi(x) = \frac{1}{1+e^{-kx}}$), shown in Figure 6.3 (b) and (c), respectively. The hyperbolic tangent is also commonly used in place of the logistic function, and will be further discussed in Section 6.2. The logistic function and the hyperbolic tangent are often collectively referred to as sigmoidal

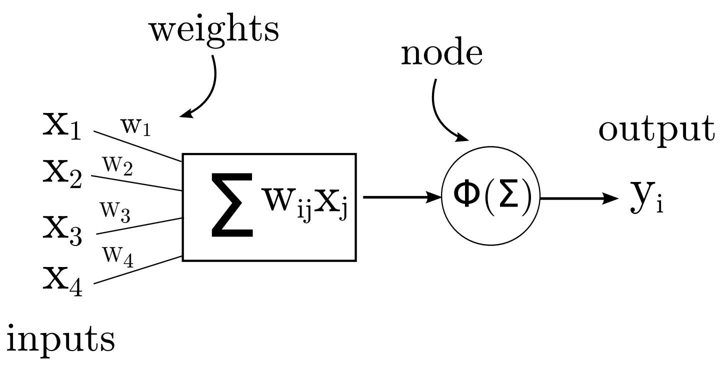


Figure 6.2: Operation of an individual neuron/node in a neural network. The neuron has several inputs x_j and one output y_i.

functions, owing to their S-shaped appearance when graphed.

The advantage of these activation functions over the step function is that they can encode infinitely more data than the step function, due to their graded output. Another important property of activation functions is the numerical range of output that is produced. For example, both the logistic and hyperbolic tan functions have horizontal asymptotes, whereas the linear function has none. This is neither an outright advantage nor disadvantage; the data being processed by the ANN will determine which functions are

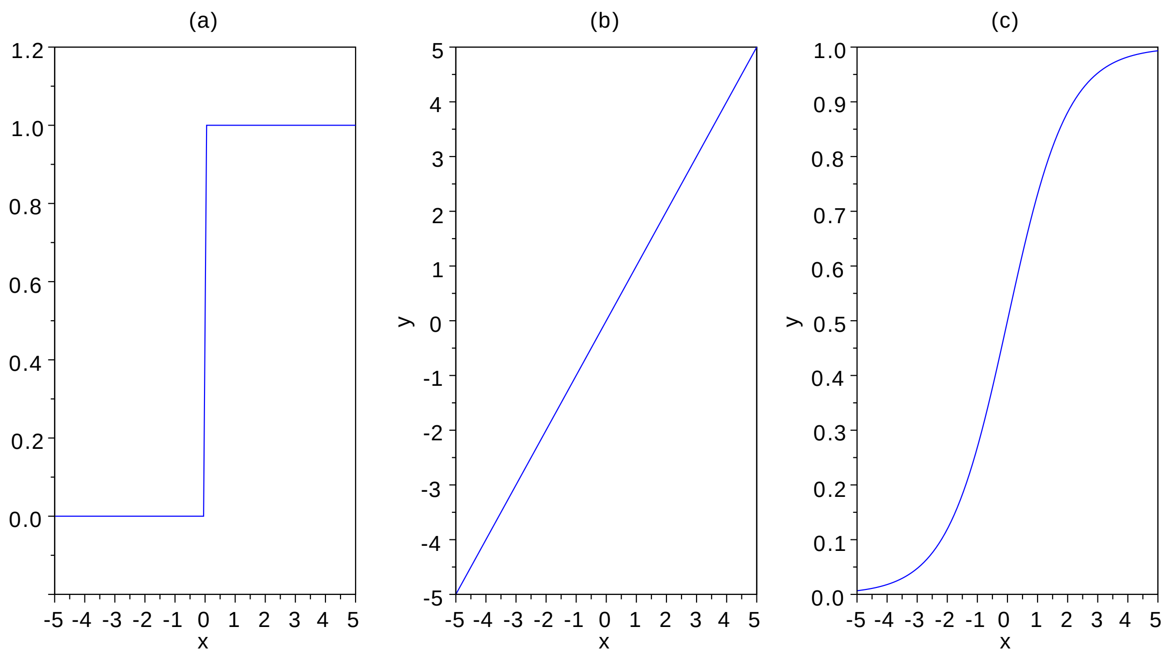


Figure 6.3: Commonly used neuron activation functions. These are the (a) step, (b) linear and (c) logistic (k = 1) functions.

appropriate. It will be shown later in Section 7.3 that linear functions can be appropriate when bounds on the output data are not known, and that sigmoidal functions are useful to enforce bounds on data in order to ensure numerical stability.

Step activation functions have the inherent property that a neuron only becomes active when a particular threshold value of the net input, $\sum_{j}^{n} w_{ij}x_j$, is reached. In order to obtain a similar property in neurons with continuous activation functions, use is made of an additional incoming signal, known as a *bias* signal. This has the effect of offsetting the net input by an amount independent of the other incoming signals. The bias signal is usually implemented by means of a *bias unit*: a fictitious unit feeding into the neuron with a constant value of 1 or -1, and an associated weight.

It is apparent that, for a particular set of input data, the output of an ANN is wholly dependent on the value of the synaptic weights. Changing the network weights will therefore alter the output signals for a particular set of input signals. This fact is utilised to train the ANN; that is, the weights of the network are systematically altered so that when known input data is fed into the network, the correct known output data is produced, to within some level of accuracy. This ability of the network weights to change is known as *synaptic plasticity*, and is the mechanism used by animal brains to record information in memory [41]. The application of synaptic plasticity to the problem of training ANNs will be discussed in the next section.

From a mathematical point of view, the operation of a neural network is identical to that of an interpolator, or function approximator. Starting with an ANN with random initial weights, the network is systematically altered until known input produces desired output, to within some degree of accuracy. As alluded to earlier, the useful property of an ANN is the ability to produce output that is close to the correct output when input data that is similar to previously-learned input data is provided. Mathematically, the network is therefore performing a regression, since there is no guarantee that the output data for previously-learned input data will be identical to the correct output that was previously learned. This will be further discussed and illustrated in Section 6.3.

The ANN architecture depicted in Figure 6.1 is known as a *feed-forward* architecture. Neurons can be conceptualized as existing in distinct layers, with signals passing in a linear fashion from one layer to the next. The network shown in the figure is known variously as a three- or two-layer network [1], depending on whether the input signals are regarded as a layer. It is apparent that if the input signals were to be relayed directly to the output neurons, the output of the network would be a linear combination of the input signals (or more correctly, the input to the output neurons' activation functions would be a linear combination of the inputs). This places severe constraints on the type of data that may be learned by the network, so often use is made of one or more *hidden* *layers* that lie between the input and output layers. This has the effect of allowing the ANN to learn data when the outputs are not a linear combination of the inputs.

As mentioned earlier, ANNs can be designed in a number of different architectures and need not have the simple structure depicted in Figure 6.1. While such feed-forward networks were found to be sufficient for this project, it is worth noting the existence of more complex network designs such as *recurrent architectures*, where neurons do not reside in conceptually distinct layers, but rather outputs from neurons are fed simultaneously to neurons in the same and forward layers [41]. This can lead to complex network behaviour as a function of time [1].

6.2 Training

The process of training a neural network consists of presenting various correct input-output pairs to the network. While an algorithmic description of the method is beyond the scope of this section and can be readily found in literature, a qualitative overview is given.

The most common method of training non-linear neural networks (that is, networks with non-linear activation functions) is known as *back-propagation of error* [42]. The overall principle is as follows: a specific set of inputs is fed forward through the network, producing the network's current approximation of the corresponding output for that input data. The error between the approximated output and the correct output is then determined, and this error back-propagated through the network. This is done by examining the gradient of each activation function, and determining the required change in the nodal weights to give the correct output. This is done backwards, from output to input, for each layer. For a network with a single hidden layer, the operation is performed only on the output and hidden layers. The mathematical basis of this is an optimisation technique known as *gradient descent*, which essentially seeks to minimise the error between correct and approximated outputs.

Because gradient descent requires the activation functions to be differentiable, back-propagation eliminates the use of step activation functions. While the logistic function given above meets this criterion, it has been shown by [42] that the hyperbolic tangent function provides better numerical behaviour. It is similar in shape to the logistic function, except its asymptotes are at y-values of -1 and +1, not 0 and 1, as with the logistic function. It provides better numerical behaviour because its output is centred around 0, as opposed to the logistic function, which is centred around 0.5.

As mentioned, the basic method for training a network is to calculate the required change

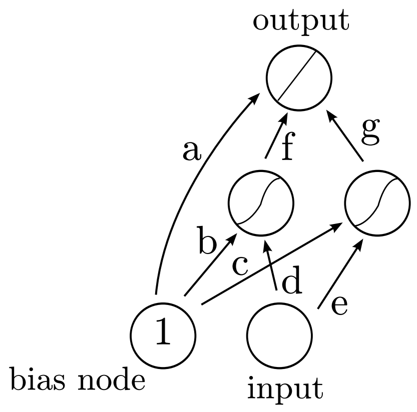


Figure 6.4: Schematic of a three-layer neural network used for performing regression. Redrawn from [42].

in the nodal weights to give the correct output for a given output. However, the weights are not simply replaced with the newly-calculated values [42]. Instead, a parameter known as a *learning rate* is implemented, denoted by η, where $0 < \eta < 1$. The learning rate is essentially a weighting between old information and new information, and allows the network to learn at a controlled rate to prevent divergent behaviour.

6.3 Neural Networks as Regression

Thus far it has been shown how neural networks can 'remember' data on which they have previously been trained. This is achieved by changing the weights of the network every time a new input-output pair is learnt. The goal of this section is to demonstrate that the apparent 'memory' of a neural network is due to the persistence of these weights and that the ability to learn new data is no more than mathematical regression. This is done to provide support for the ideas on speed, stability and accuracy presented in the next section.

Suppose we have a neural network consisting of one input node, a single hidden layer of 2 nodes with hyperbolic tangent activation functions and a single output node with linear activation, as shown schematically in Figure 6.4. Nodal weights are labelled $a, b, \ldots, f, g$.

If we denote the output of the network y and the input x, then it is possible to write

$$y = f\tanh(dx + b) + g\tanh(ex + c) + a \qquad . \tag{6.1}$$

The output of this network will always have this form. The difference between two such networks trained on different sets of data will simply be the values of the weights $a, b, \ldots, f, g$. Therefore, when a neural network is trained with a particular set of data

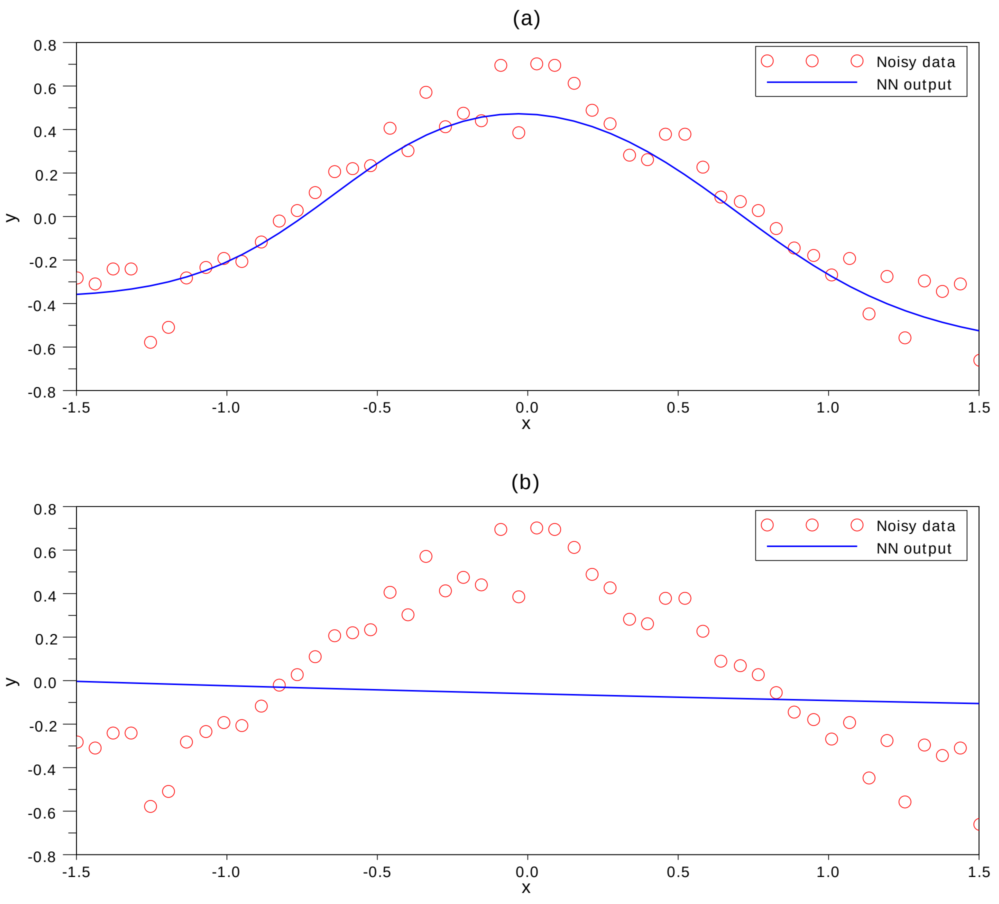


Figure 6.5: Output of a neural network with one hidden layer, trained with noisy data. Networks (a) and (b) used identical learning rates and data points.

(e.g. by backpropagation), what is actually occurring is a systematic altering of these weights so as to reduce the mean squared error between learnt data and the approximation thereof. To demonstrate this graphically consider a set of noisy data shown by the red circles in Figure 6.5.

Using the neural network class that was developed for use in this project, a neural network with randomised initial weights was trained with this data by presenting the data points in random order, 200 times in total per data point. Two different cases of the network's outputs for all x values in the domain of the data points is shown in Figure 6.5 (a) and (b).

In graph (a), a good fit to the data is seen. The network has clearly performed regression on the data. However, graph (b) shows the network having performed a poor fit, exhibiting what is known as *bias*. Generally, behaviour similar to graph (b) is seen only a small percentage of the time (less than 10% during simple testing). In informal terms, it results from the particular order in which the data points are presented. In some cases the fitted curve can initially be a straight line (because the initial data points are close to being in a straight line), with the result that the curve can remain a straight line, even in the presence of new data [42]. Methods for overcoming this problem will be mentioned in the next section.

6.4 Speed, Stability and Accuracy

As was shown in the previous section, neural networks essentially perform a generalised regression on input-output pairs. They are therefore prone to the speed, stability and accuracy issues found in typical regression algorithms. This section will briefly highlight some of these issues, with reference to the practical techniques which were used in this project to prevent such problems occurring.

An interesting property of neural networks not found in typical function approximators is that the accuracy of learnt information is strongly dependent on the number of times the network was trained using that information, as well as the order in which the data was presented. One consequence of this is that ANNs can 'forget' information that was previously learned [41]. This can be both advantageous and problematic, particularly in the case of agents which are constantly learning from their environment [44]. Forgetting can be useful if the agent is to adapt to a changing environment, in which case old, irrelevant data will be discarded and the corresponding input signals mapped to new outputs. However, if the agent is attempting to approximate a function that does change, such as with the collision avoidance behaviour detailed in Chapter 8, forgetting can be particularly undesirable.

A number of methods may be used to overcome the problem. *Interleaved learning* [45] involves breaking the input data into several overlapping sections, and learning them in succession. However, this method is more applicable to offline learning [45], and is therefore not suitable for autonomous robots. A method suitable for online learning is *consolidation learning* [44] which makes use of a small ANN for online learning, which network is subsequently used to train a larger, offline ANN at a later stage. It is thought that this mimics the learning and interference patterns found in in the human brain [44]. Catastrophic forgetting was not found to be particularly problematic in this project.

The method by which the initial weights in the network are determined can have a significant effect on network behaviour. If the initial weights are ill-suited to the particular data on which the network will be trained, convergent behaviour can result [42]. The most common technique is to use random numbers in the range $(-r, r)$, for some small number r. For data following Gaussian distributions, it has been shown that the value of

r least likely to lead to network divergence is $r_i = \frac{1}{\sqrt{A_i}}$, where A_i is the number of weights feeding in to each node [42].

The other major parameter of a neural network is its learning rate. However, unlike the initial weights, there is no well-established method for determining learning rates *a priori*. A general heuristic that may be used is to start with a learning rate of $\eta = 1$, and decrease the rate until convergence of the network occurs [42]. However, since learning rates close to 1 can lead to divergence in certain circumstances, a global learning rate of $\eta = 0.5$ was used throughout this project, without any network divergence being experienced.

Chapter 7

Reinforcement Learning

Reinforcement learning is a form of unsupervised machine learning that utilises an internal critic to reward or punish an agent's actions, which are made on the basis of the environmental state at any given time. This chapter will present the theory of reinforcement learning necessary for the collision avoidance problem presented in the next chapter.

It should be mentioned that rigorous, formal methods for the analysis of reinforcement learning problems have been developed. Despite this, an attempt will be made here to introduce only the minimum theory and notation required for adequate analysis of the collision avoidance problem which follows. General information will also be given about the applicability of reinforcement learning techniques to other tasks relevant to autonomous robotic agents. Most of the information contained in this chapter is a summarised survey of the relevant concepts presented in a 1998 book by Sutton and Barto [2], widely recognised as the definitive work on reinforcement learning [1,46].

7.1 Basic Theory

In essence, reinforcement learning (RL) is a technique that allows agents to formulate a mapping from situations to actions. This mapping is made in such a way so as to maximise a numerical, scalar reward value [2]. In the context of an autonomous robotic agent, RL helps agents decide which motor actions are best, given a particular sensory input. The primary advantage of RL over other machine learning approaches is that it is not necessary for a supervisor or critic to know correct input/output pairs. Rather, the agent implicitly deduces the correctness of a chosen output for a given input based on the magnitude of the reward the environment assigned to that decision. Additionally, the

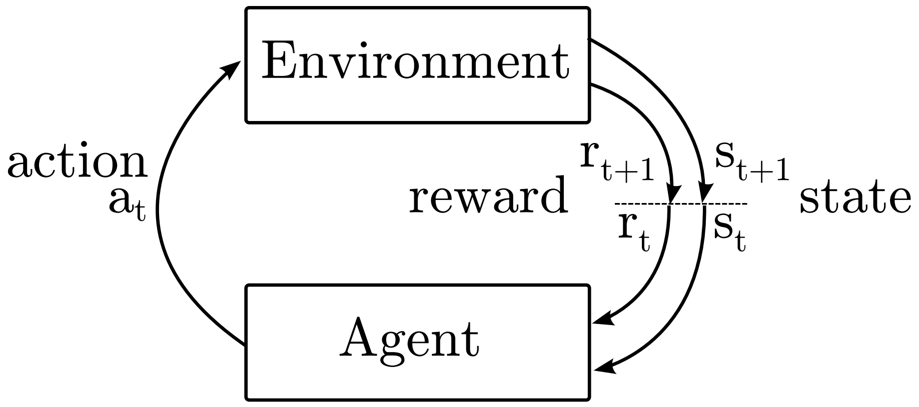


Figure 7.1: Information flows between agent and environment. Modified from [2].

internal critic does not need to correct sub-optimal actions; the agent will learn which actions are optimal for a given state by means of *exploration* [2].

The general formulation of an RL problem is as follows, adapted from [2]. Consider an agent interacting with its environment. At time t, the agent receives the state of the environment $s_t \in S$, where S is the set of all possible states. Based on the environmental state, the agent selects some action $a_t \in A(s_t)$, where $A(s_t)$ is the set of actions which are possible in state s_t. In the following time step, the environment provides the agent with two pieces of information: the new environmental state s_{t+1}, as well as a scalar reward value r_{t+1}. The reward r_{t+1} is temporal, and thus the reward associated with taking action a_t in state s_t need not always equal r_{t+1}.

There are thus three distinct signal flows in the agent-environment interaction: the agent providing the environment with its chosen action, and the environment responding with a new state and a reward for taking the chosen action in the previous state [2]. This flow of information is shown graphically in Figure 7.1.

The agent's method of choosing which action a to take at a particular time is called the agent's *policy*, and can be denoted by π_t, indicating that the policy is liable to change. The distinguishing factor in different RL approaches is how the policy is updated. One such approach that will be discussed in Section 7.3 is Q-learning, where the notion of *utility* is used to choose actions that appear to maximise future rewards.

In the context of a mobile robotic agent, care needs to be taken in defining the boundaries of the agent and environment. As mentioned in Section 2.3 on page 10, the terms 'agent' and 'robot' are used almost synonymously throughout this report. However, in the context of agent-environment interactions in reinforcement learning, a careful distinction needs to be drawn between 'agent' and 'robot'. While 'robot' refers to all tangible apparatus that are part of the machine, including the physical controller, 'agent' is an abstract term and refers to the part of the controller that selects actions based on sensory data. In the context of RL, then, the rest of the physical robot, such as sensors, motors, wheels and chassis are considered external to the agent.

A crucial consequence of this distinction is that, while the *value* of the reward at each time step may be calculated by the robot's controller, the reward itself is still provided by the environment, and is thus external to the agent. This distinction does not require that the agent (in the sense defined in the previous paragraph) not know how the reward is calculated. It is entirely possible that an agent knows precisely how reward is based on states and actions, but still faces the problem of how to choose actions so as to maximise long-term reward [2]. Consider the analogy of a person playing chess: the person (agent) has full knowledge of the rules of the game that lead to a win (reward structure), but choosing the sequence of actions that will lead to a win is far from trivial.

Thus, at the risk of repetition, it is critical that the reward computation be considered as occurring in the environment, and relayed as a signal to the agent, as shown in Figure 7.1. A consequence of this conceptual framework is that the reward function encodes information relating to *what* the goal of the agent is, and not *how* to achieve it. For example, in the collision avoidance simulation presented in Chapter 8, the reward function punishes agents for being within a certain distance to an obstacle. Crucially, the reward function provides no instruction as to how an agent should avoid being near obstacles; this is left up to the agent to learn [2].

While it may seem trivial or unnecessary to use RL to learn a basic behaviour such as collision avoidance, which can quite easily be implemented in a deterministic control function, the benefits of RL become readily apparent when the required task is too complex for a behaviour-based approach. For example, an agent-based computer program was written that learnt to play backgammon using RL [47] (or more correctly, it learnt how to *win*). The program was given no more information than the rules of the game, and a reward function that rewarded only a win and otherwise provided no further guidance. By playing against itself thousands of times, the program was learnt how to win the game so well that it could compete on the level of the world's best players [47]. While a conventional, analytical approach to the game is certainly possible, it is undoubtedly much harder to resolve a winning strategy into a series of "if this, do that" rules. RL in scenarios such as these, where reward is only provided after a long series of time steps, suffers from what is known as the *temporal credit assignment problem*, and will be discussed further later.

Having presented the basic theory of RL, the remainder of this chapter will focus on specific details of RL implementation. The nature of reward functions will be discussed, followed by an explanation of Q-learning and corresponding methods of encoding the state-action space. Finally, the dilemma of exploration and exploitation will be briefly discussed.

7.2 Reward Functions

In informal terms, reward functions are a way of communicating to the agent *which* goals are desirable but not *how* to achieve them. More formally, the goal of any agent in an RL task is to maximise the reward received by the environment. Therefore, by selection of a reward function that will return highest reward for results considered desirable by the designer, it is possible to 'coax' the agent into make its goal of maximising reward coincident with the designer's goals.

With reference to Figure 7.1, the reward value r_{t+1} returned by the environment is a function of the state s_t and the action chosen by the agent in that state, a_t. As previously stated, the goal of an agent is to maximise reward. But since new reward values are received at each time step, it is not well-defined which reward should be maximised. This requires categorisation of tasks into two broad classes: *episodic* and *continuing* [2]. Episodic tasks are those which naturally repeat and can thus be decomposed into discrete *episodes*, whereas continuing tasks have no clear terminal state.

Real-world tasks need not exclusively be episodic or continuing in nature, but their formulation as RL problems should be exclusive [2]. For example, consider the problem of a mobile robot learning to avoid collisions. This could be framed as an episodic task which ends when the robot collides with an object, or as a continuous task where the robot continually explores the map, occasionally colliding with obstacles. However, some tasks are necessarily episodic because they have a terminal outcome, such as a game of chess.

For episodic tasks, there is a finite set of rewards $r_1, r_2, \ldots, r_n$ where t_n is the terminal time step of the system (such as checkmate in a game of chess, or colliding with an obstacle in the case of a mobile robot) [2]. It is possible to define the a function $R(r_1, r_2, \ldots, r_n)$ to be the total return, that should be maximised. Perhaps the most obvious choice for R is

$$R_t = r_{t+1} + r_{t+2} + \cdots + r_n \qquad . \tag{7.1}$$

By contrast, continuous tasks have no terminal state s_n and we therefore require a way to quantify total return in the absence of a terminal state. One such approach [2] is to make use of a *discount factor*, γ, such that

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \qquad . \tag{7.2}$$

The use of a discount factor places more weight on earlier rewards, with lower discount factors corresponding to greater weight on earlier rewards. Clearly, if $\gamma = 0$, $R_t = r_{t+1}$ and the agent seeks to maximise immediate reward. For convergence of R_t, it is apparent that $0 \leq \gamma < 1$.

Note that in the case of episodic tasks, learning can only take place at the end of the episode, since it is only then that the total reward R is known. In contrast, continuing tasks allow learning at each time step. For tasks with a delayed reward (such as a chess game where positive reward is only given for a win) the problem emerges of how to assign credit for the total return to each individual action in a (possibly long) series of actions, known as the *temporal credit assignment* problem [1, 2]. Different RL techniques solve this problem in different ways. One such technique is Q-learning, which will now be described.

7.3 Q-learning

7.3.1 Description

Q-learning is an RL technique that uses an *action-value function* to estimate the utility to the agent of being in a particular state [48]. Informally, it is based on the notion that some states may be more or less beneficial to be in than others, so the best actions to select are those that put the agent in the most beneficial state.

More formally, denote $V^{\pi}(s)$ as the *value* of state s when the agent is following policy π. The inclusion of the agent's policy is necessary because the total future return that can be expected from a particular state is dependent on the sequence of actions the agent takes after being in that state, which actions are determined by the agent's policy. Next, define $Q^{\pi}(s, a)$ as the *utility* (or usefulness) of taking a particular action a in state s, and following policy π thereafter. This utility is largely based on the *value* of the succeeding state, $V^{\pi}(s')$, but will also be affected by the immediate reward received for selecting action a in state s. To paraphrase the above notation in informal terms, the *value* of a particular state is the return that can be expected from being in that state and following a particular policy thereafter. Closely related to the value, the *utility* of performing a particular action in a particular state is determined by the immediate reward received for performing that action in that state, as well as the prospect of future returns from being in the new state (which future returns are equal to the *value* of the successive state) [2].

The Q-learning approach is to define an action-value function $Q(s, a)$ that is independent of the chosen policy π. As agents learn, they continually update their internal mapping of the utility function Q. A major strength of Q-learning is that it is policy-independent:

agents can follow any chosen policy, while simultaneously gaining knowledge about the mapping from state-action space to reward space. At any time, the agent can change its policy and the learnt Q values remain valid. Specifically, as more learning occurs and the agent is able to build a more accurate internal representation of the Q function, $Q \to Q^*$, where Q^* is the utility function for the *optimal* policy, i.e. the policy that will result in greatest total returns.

As just mentioned, the agent continually refines its internal representation of the function $Q(s, a)$ as it gains reward information at each time step. Mathematically, the update function for Q-learning [2] is

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \quad .$$

The new variable α is known as the Q-learning rate and expresses how quickly the Q function is updated to include new knowledge and disregard old knowledge. This becomes clear when rearranging the update equation,

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}}(1 - \alpha) + \alpha \underbrace{\left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) \right]}_{\text{learned value}} \quad . \tag{7.3}$$

Equation 7.3 makes explicit the fact that α is a weighting between the old and new Q values.

The most computational work that is done by an agent during a Q-learning time step is approximating the function Q^*, and there are a number of methods to do this. If the state space and action space are both discrete, a simple look-up table may be used [48]. However, if either the state or action spaces are continuous, more optimal strategies exist. These will be explored in the next section.

7.3.2 State-action Space Encoding

As already mentioned, it is the job of the agent to approximate the Q function and continually update this approximation so that it can converge to the true Q function for an optimal policy, Q^*. Neural networks serve well as function approximators. It would thus be conceivable to use a single neural network to approximate Q. The input could be a combined vector representing s and a, and the output of the network could be a single node giving the corresponding Q value.

However, neural networks have difficulties approximating data where similar input values give very different output values [42], as would be the case if the state and action values were combined into a single input vector. There are a number of alternative conceivable methods of representing the state and action values, depending on whether both are continuous, or at least one of them is discrete. The case of both being discrete can be handled by a look-up table, as mentioned, or using one of the methods that will now be presented for discrete data.

In some cases, both the state and action spaces may be continuous, but it is desirable to discretise them to reduce complexity. Consider for example the action space of the mobile robots presented in this project. Their action space can be represented as a vector in $\Re^2$, with the elements corresponding to the left and right motor speeds. However, the action space could be discretised into a finite number of such vectors, corresponding to actions such as 'directly forward', 'directly backward', 'stationary turn left', 'stationary turn right', and so forth. This was the approach taken in the collision avoidance problem in this project and will be detailed in the next chapter.

In order to represent the utility function when at least one of the inputs is discretised, a series of neural networks can be used, one for each discrete input [49]. For example, if the action space is discretised, each action such as 'directly forward' could have a neural network associated with it. The input to each neural network would be the sensor values (which can, but need not, be discretised) and the output for each network the Q value corresponding to the state-action pair.

If both the state and action spaces are continuous, methods exist for constructing appropriate function approximators. One such method uses a neural network coupled with an interpolator [50]. A vector representing the state value is the input to the neural network, the output of which is a series of ordered pairs representing specific action values and their corresponding Q values in the given state. A wire fitter interpolator is used so that the Q value for any action value can be determined. Thus, if regarded as a single unit, the neural network and interpolator operate as a function approximator which accept state and action values and return Q values. The operation of this function approximator is shown in Figure 7.2 on the next page.

When an updated estimate of Q has been calculated by means of equation 7.3, this Q value is fed 'backwards' through the interpolator to produce a new set of ordered pairs, as described above. The neural network is then trained to associate the new set of ordered pairs with the original input data (the state vector), and the function approximator is thus updated.

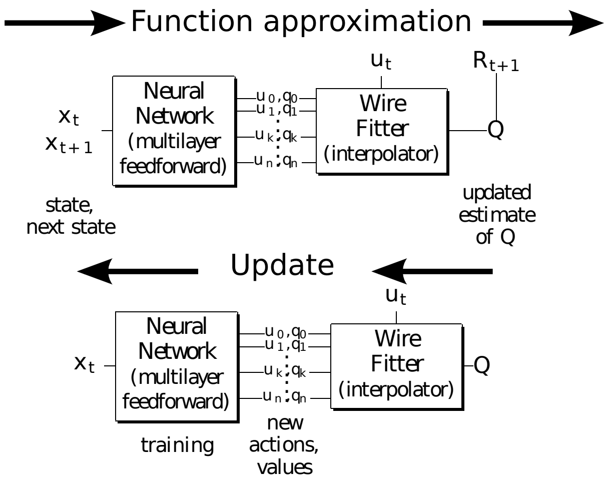


Figure 7.2: Q-learning with continuous state-action spaces. x, u denote states and actions, respectively. Reproduced from [50].

7.4 Exploration and Exploitation

In general, agents engaged in RL tasks (episodic or continual) attempt to maximise the total return they receive, R_t. This is achieved by, at each time step, selecting the action a_t that will result in highest return, based on past observation. For agents that have developed an adequate mapping between state, action and reward, the previous statement is unconditionally true, because agents have reliable knowledge of which actions will result in highest reward in a given state. However, when knowledge of the state-action reward mapping is incomplete, agents cannot be certain whether best reward will result from an action whose reward value they have previously determined, or from an action whose reward value is, as yet, unknown.

This dilemma is often referred to as the *exploration versus exploitation* problem, due to the fact that agents need to decide whether to *explore* unknown state-action pairs (i.e. determine the reward associated therewith), or to *exploit* previously-gained knowledge about which action is best given the current state. There are a number of approaches to resolving this dilemma. A popular method of resolving this problem, and the one implemented in this project, is to use a Boltzmann probability distribution [1,2,46,49]. With roots in thermodynamics, this probability distribution uses a single variable T (originally to refer to temperature) to describe an agent's propensity to explore or exploit.

Suppose that, in a particular state s_t, an agent can choose from a number of actions $a_i \in A(s_t), i = 1, 2, \ldots, m$. For convenience, denote $Q(s_t, a_i)$ as Q_i. Then each action a_i has an associated Q value Q_i, indicating the expected reward from taking that action in the current state. Define a variable $T > 0$ to be the tendency of the agent to explore the state-action space, with larger values corresponding to more exploration, and lower

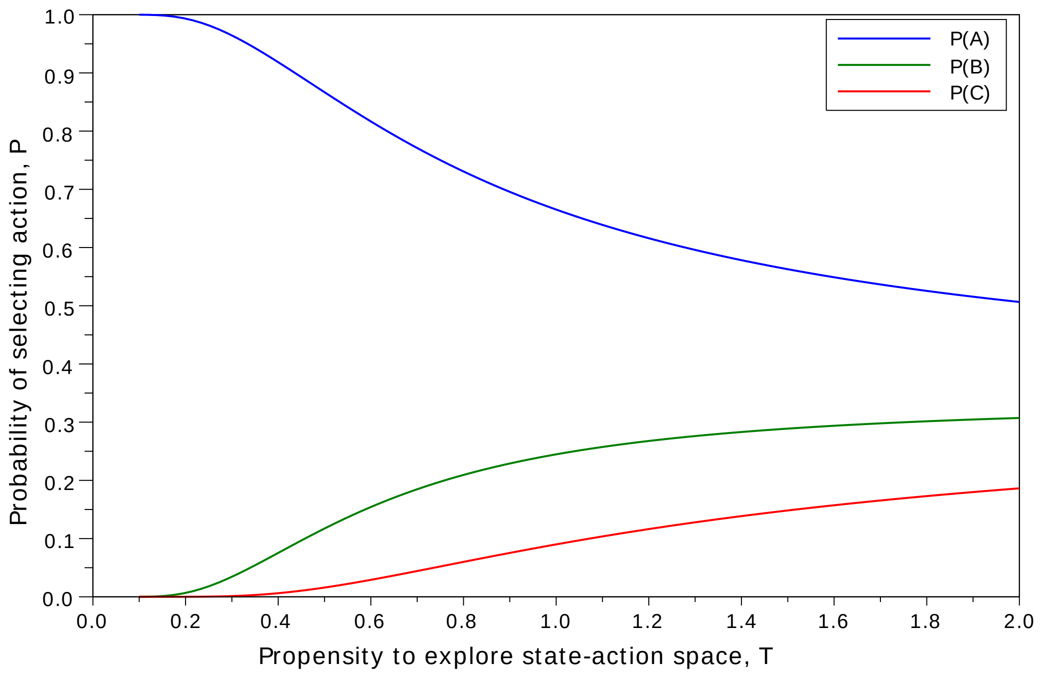


Figure 7.3: Boltzmann probability distribution, showing the probability of selecting three actions A, B and C where $Q_A = 3$, $Q_B = 2$ and $Q_C = 1$.

values to exploitation. For a given value of T, the agent can select which action to take based on a probabilistic approach, where

$$P(a_i) = \frac{e^{Q_i/T}}{\sum_{k=1}^{m} e^{Q_k/T}} \tag{7.4}$$

To implement this rule, the agent's policy is set such that it will randomly choose an action a_i with probability $P(a_i)$. To gain a sense of how these probabilities vary with T, a graphical aid is helpful. Suppose in a particular state three actions are available: A, B and C, with corresponding Q values 3, 2 and 1, respectively. The graphs of $P(\{A, B, C\})$ as functions of T are shown graphically in Figure 7.3.

As can be seen in Figure 7.3, the probabilities of each action being taken approach equality as $T \to \infty$. Conversely, as $T \to 0$, the probability of the action with highest Q value being chosen approaches 1. In practice, an agent would begin with a high T value, which is gradually lowered as the agent learns more about the mapping from state-action space to reward. The result is that near the beginning of its learning process the agent will often pursue actions that cause it to receive low reward. However, as T is lowered, the agent will pursue these actions less frequently, instead selecting actions that return a higher reward.

Chapter 8

Reinforcement Learning for Collision Avoidance

In Chapter 5, agents were given simple rules to follow and it was seen that complex behaviours emerged from the interaction of these simple rules in a group context. While many biological agents have instinctive knowledge of the rules they follow (that is, they are born with such rules 'programmed' into their genes), a great number of biological agents also learn from their environment the rules that, if followed, can lead to behaviour beneficial to the individual and to the group as a whole. This ability to learn what is advantageous shows a closer conceptual fit to the definition of agency given in Section 2.1. Seeking to reproduce this behaviour in simulated robotic agents, the task of agents exploring a map, while avoiding obstacles, was attempted. While this behaviour had previously been implemented in Chapter 5 through a behaviour-based (i.e. rule-based) approach, the goal with the present simulation is to reproduce such collision avoidance behaviour, but in the absence of specific instruction. This is achieved by using the theory on artificial intelligence, and specifically reinforcement learning, that was presented in the previous two chapters.

This chapter begins by explaining the conceptual foundations of the task, presents the reward function used and ends with a discussion and analysis of the results obtained from simulation.

8.1 Introduction

The key difference between the behaviour-based approach and the machine learning approach to robotics is that, in the behaviour-based methodology, desired agent outcomes are decomposed (by the designer) into a number of basic behaviours. If the agent follows the behaviour, the overall outcome (in the individual and possibly group contexts) is achieved [1]. The machine learning paradigm takes the view that agents are better able to determine the optimal way to execute a task by being given feedback on *how well* they are accomplishing the goals of the task, rather than being instructed specifically *how to execute* the task. This is the approach presented in this chapter.

The goal of the simulation was to put agents in an unknown environment, with no knowledge of how they should map sensor inputs to motor outputs, and have them learn to explore the map as quickly as possible, all the while avoiding collisions with map objects. Agents are not made aware of the significance of sensor readings (i.e. they have no knowledge that their sensory input refers to distances to objects). They are given no basic behaviours to follow and are free to perform any set of actions. A reward function is used to reward behaviour that the designer considers 'good', and provide negative reward to behaviour considered 'bad' in the context of the task.

As was mentioned in Section 7.2, the reward function does not provide any guidance on *how* to achieve the task, but instead only communicates to the agent how well it is performing. Therefore, the problem given to the agent could be posed as follows: "explore the map as quickly as possible, and avoid getting too close to other objects". Recall that the agent has no knowledge of how its motor outputs affect subsequent sensory inputs, and thus has no direct knowledge of how to achieve goals such as 'explore quickly' and 'avoid collisions'. What the agent *is* able to do is to perform different actions, record how well those actions are rewarded, and base future decisions on previous reward. This is the goal of Q-learning and reinforcement learning in general.

Section 7.2 differentiated two kinds of RL tasks: episodic and continual. The task of exploring a map while avoiding collisions could be implemented in either of these two ways. In the episodic approach, an agent would explore the map until it collides with another object, at which point the episode concludes and the agent is moved to another position on the map to begin a subsequent episode. In the continual approach, an agent would continuously explore the map, occasionally colliding with obstacles, but after a collision would continue exploring without the simulation being restarted. While the task of collision avoidance has been implemented in the episodic sense with success [49], it was decided to use a continual approach, as this represents the more feasible physical reality. That is, if the task were to be moved from the simulated domain to physical robots, it would likely be desirable that these robots could learn throughout their 'lifetime', as opposed to being monitored and their learning restarted each time a collision occurs. These approaches represent on-line and off-line learning, respectively.

Section 7.4 discussed the dilemma of exploration and exploitation, or how agents should decide between selecting actions known to provide good return (accumulated future reward), as opposed to trying other actions whose return is unknown. Agents' propensity to explore the state-action space is described by variable T when using a Boltzmann probability distribution to decide the probability of selecting exploratory versus exploitative actions. For this task, the approach was taken of starting each agent with an initial T value, and having agents decrease their T values at an exponential rate as the simulation progressed. Figure 7.3 on page 88 shows that the probability of selecting the most optimal action becomes asymptotically close to 1 (certainty) for some small, positive value of T which will be denoted T_c. It is therefore not necessary to reduce T past T_c, as the agent's behaviour will be near-identical. T can therefore be expressed as a function of simulation time, in the form

$$T_t = e^{-\beta t} + T_c \tag{8.1}$$

for some positive β. However, since agents are required to update their T value at each time step, and they have no notion of a global time variable t, it should be possible for them to update T based on its value alone. Thus, differentiating,

$$\frac{dT}{dt} = -\beta e^{-\beta t} \qquad . \tag{8.2}$$

The corresponding update rule for a single time step is of the form

$$T_{t+1} = T_t + \frac{dT}{dt}|_t \times 1 \qquad .$$

Substituting (8.2),

$$T_{t+1} = T_t - \beta e^{-\beta t} \qquad .$$

And finally, using the definition of T from (8.1),

$$T_{t+1} = T_t - \beta(T_t - T_c) \qquad . \tag{8.3}$$

Therefore, at each time step, agents need only apply the update rule in equation 8.3, and their T value will follow the same exponential trajectory regardless of its initial value. It is clear from the form of equation 8.1 that smaller values of β will provide slower convergence to T_c. This fact was used in simulations and will be discussed in the results section to follow.

Section 7.3.2 discussed various methods of encoding the state-action space, depending on whether the states and actions are considered discrete or continuous. In this task, both states and actions are continuous, but the decision was taken to discretise the action space, to simplify and expedite agent learning. The particular discretisation used is described in the next section.

The method for approximating Q was to use a separate neural network for each discrete action. The input to each network is the state s, and the output is the corresponding Q-value for the state-action pair. The network architecture used had a single hidden layer with hyperbolic tangent activation functions, with the output layer having linear activation functions, similar in architecture to Figure 6.1 on page 72. It was necessary to use linear functions on the output nodes, because such functions are capable of producing output of any magnitude, as opposed to the hyperbolic tangent, which is only capable of producing output between -1 and 1. The reason output of any magnitude was required is that Q does not necessarily lie within a bounded range. This will be discussed further in the next section.

The number of nodes in the input layer was equal to the number of sensors. The number of hidden nodes was in all cases double the number of input nodes, which was found to provide the networks with sufficient convergent behaviour. A single output node was used, corresponding to the Q value of the state-action pair. Updating of the neural network was performed by the back-propagation of error method detailed in Chapter 6.

8.2 Reward Function

The goal of the reward function in RL is to communicate to the agent how beneficial its behaviour is in the context of a given task. The two goals of this task are for (1) agents to explore the map as quickly as possible and (2) avoid collisions while doing so. Since the reward function $r(s, a)$ takes as arguments both the environmental state (s) and the corresponding action chosen by the agent (a), it is possible to reward agents either for the state they are in, or for the actions they have selected, or both. However, the two goals of this task are generally distinct, and it is therefore possible to decompose the reward function into two parts, each of which is designed to elicit the two behaviours mentioned above. Accordingly, the reward function that will be presented can be considered, under certain circumstances, "state-based", and in other cases, "action-based". Note that it is entirely possible for the reward function to be based both on states and actions simultaneously.

When the `refresh()` method of an agent is called by the simulator, the agent queries its sensors' `sense()` methods, which return the scalar value of the distance to the nearest object (or -1 if no object is sensed), and a corresponding number that encodes the type of object (0 for obstacles, 1 for agents and -1 for no object sensed). The agent assembles all these sensor inputs into an $n_s \times 2$ matrix (where n_s is the number of sensors), and passes this matrix to the controller, which interprets it. For this task it is not necessary

Table 8.1: Discretised action values

(-1, -1)	full speed backward
(-1, 1)	stationary left turn
(1, -1)	stationary right turn
(1, 1)	full speed forward
(-1, 0)	backward right turn
(0, -1)	backward left turn
(1, 0)	forward right turn
(0, 1)	forward left turn

to differentiate between obstacles and agents (as they should both be avoided), so the state data passed to the reward function is a single vector of length n_s consisting of the sensor distance values.

In order to provide a limit on the speed that agents are able to rotate their wheels, the controller was written so that the learning process only ever uses speed values between -1 and +1, which correspond to full-speed rotation in the reverse and forward directions. To obtain the actual motor speed, the controller multiplies the above value by a fixed constant speed. This provides a physical maximum speed that agents are able to move, which in the real world can prevent damage to hardware during the early stages of learning. Action data is therefore encoded as a single vector of length 2, consisting of the left and right motor speeds as a fraction of the maximum prescribed speed.

As has been mentioned, a discretised action space was used. The discrete action values used are shown in Table 8.1. Initially, only the first four action values were used, but during testing the final four were added, for reasons discussed in the next section.

To encourage collision avoidance, variable i was defined as the minimum sensor distance value, and lower i values were given more negative reward than higher i values, below a certain threshold, d_{crit}. To encourage map exploration at maximum forward speed in a straight line, reward was assigned based on action only if i was above the threshold value. For assigning action-based reward, two approaches were taken. The initial method was to make the reward equal to the algebraic sum of the action vector. This would give reward of -2 for action (-1, -1), reward of +2 for action (1, 1) (the most desirable action), and intermediate values for other actions. However, problems were encountered with this approach and are described in the next section, so an alternative reward structure was created which gave maximum positive reward for action (1, 1), and maximum negative reward for all other actions. Note that this last reward structure will only work in a discretised action space if (1, 1) is one of the possible actions.

Recall from Section 7.3 that for Q-learning, agents are concerned with selecting actions that maximise utility, Q. Thus the neural networks created for each discrete action value provide a mapping from state to Q values. It was also mentioned in Section 6.4 that neural networks with hyperbolic tangent activation functions perform best when inputs and outputs are generally small and centred around 0. It is therefore desirable to ensure Q values do not become too large. From equation 7.3 on page 85, the new (learnt) Q-value at each time step is given by

$$Q_{learned} = \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a)\right] \qquad .$$

Therefore, the maximum possible Q value is given by

$$Q_{max} = r_{max} + \gamma Q_{max} \qquad .$$

Solving for Q_{max},

$$Q_{max} = \frac{r_{max}}{1-\gamma} \qquad . \tag{8.4}$$

Clearly, the maximum value of Q is a function of the maximum possible reward value and discount factor, γ. Therefore, in order to keep Q small, r should also be bounded. It was decided to normalise r to lie within the range $-1 \leq r \leq 1$. Then, for a discount factor $\gamma = 0.7$, $Q_{max} \approx 3.3$, and for $\gamma = 0.9$, $Q_{max} \approx 10$, which is sufficiently small for the neural networks to converge. Discount factors above 0.9 would rarely be used.

Figure 8.1 on the following page shows the reward functions used for the simulation, after applying the above normalisation. In the "state-based" region, the chosen action has no effect and reward is calculated solely based on distance to the nearest object. This alone is sufficient to encourage collision avoidance. To reward exploring the map as quickly as possible when not near obstacles, the "action-based" regions take into account only the chosen action and not the sensory state. In case (a), the reward distribution is well-spaced, whereas in case (b), maximum positive reward is given for action (1, 1), and maximum negative rewarded given for any other actions.

The curve of the reward function in the "state-based" region is the natural logarithm with the appropriate transformations. Specifically, reward function (a) is defined as

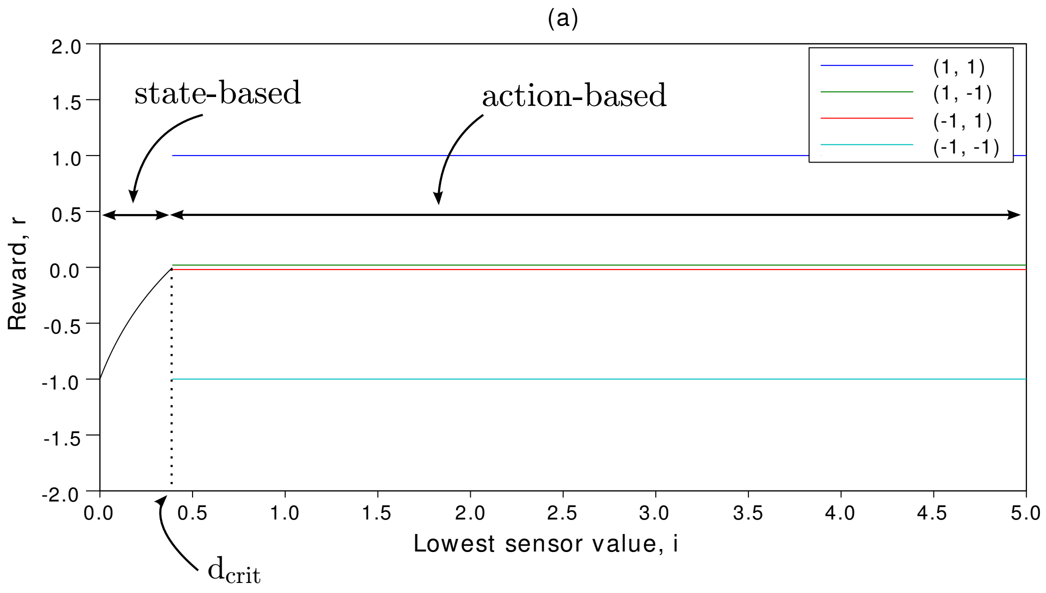
$$r(\mathbf{s}, \mathbf{a}) = \begin{cases} \ln(q \min \mathbf{s} + 1) - 1 & \text{if } \min \mathbf{s} < d_{crit} \\ \dfrac{\text{sum}(\mathbf{a})}{2} & \text{otherwise} \end{cases} \tag{8.5}$$

for some constant q where $d_{crit} = \frac{e-1}{q}$. q was typically chosen to give $d_{crit} = 0.4$. Note that d_{crit} is the distance at which obstacles will generally be avoided, but only for agents with $\gamma = 0$. If $\gamma > 0$, the actual distance at which obstacles are avoided will be larger.

This will be shown in a later section.

Similarly, reward function (b) has equation

$$r(\mathbf{s}, \mathbf{a}) = \begin{cases} \ln(q \min \mathbf{s} + 1) - 1 & \text{if } \min \mathbf{s} < d_{crit} \\ 1 & \text{if } \min \mathbf{s} \geq d_{crit} \text{ and } \mathbf{a} = (1,1) \\ -1 & \text{otherwise} \end{cases} \tag{8.6}$$



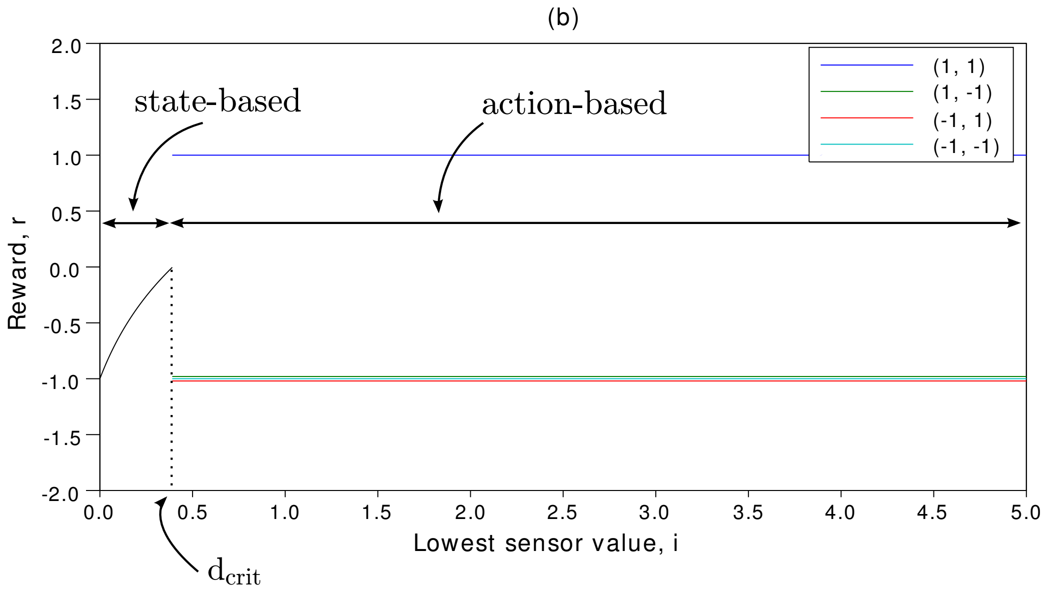


Figure 8.1: Reward functions used in Q-learning collision avoidance simulation, as functions of i. (a) Original function. (b) Modified function.

8.3 Simulation Results and Discussion

Collision avoidance Q-learning simulations were run with the reward functions mentioned above. These were performed both in the single agent and multiple agent domains (the corresponding simulation files are `QlearningSingleAgent.sim` and `Qlearning-MultiAgent.sim`), using the `Agent9` configuration shown in Figure 4.6 (c) on page 32. In both of these cases, agents were observed to move around the map, initially colliding with obstacles fairly frequently. As the simulation progressed and agents gained a more accurate mapping of state-action space to reward space, and as their T values were decreased, the frequency of these collisions were observed to drop sharply. All simulations were done using the map in file `original.map`, shown in Figure D.1 on page 149.

The initial T value used for all agents was 1. The minimum T value used was $T_c = 0.2$. Note that this value is only ever approached asymptotically. Based on empirical testing, it was found that at this T value the frequency of collisions had become nearly 0. Two different rates for decreasing T were used: $\beta = 0.005$ and $\beta = 0.0001$.

The other parameters affecting simulation are the Q-learning rate α, the neural network learning rate η and the discount factor γ. For now, it will just be stated that values of $\alpha = 0.4, \eta = 0.1, \gamma = 0.7$ were used in the simulations presented below. A fuller discussion of the effect of these parameters will follow.

In order to present the frequency of collisions in a format that is easy to analyse, the simulator was modified to record every occurrence of a collision between an agent and another object (either another agent or obstacle). A graph showing the frequency of collisions for a single agent over 33 000 time steps is given in Figure 8.2 on the following page, along with the corresponding T values as a function of time ($\beta = 0.0001$). Since the collision history for each time step is binary (1 for a collision, 0 for none), a plot of the raw data would make analysing trends difficult. As such, the "mean collisions" shown for a particular time step is the mean of the collisions for the 10 closest time steps (5 on each side).

It is clear from Figure 8.2 that the frequency of collisions, while showing some relationship with T, also has a great deal of random noise present, which can be attributed to local conditions such as the starting position of the agent and also its position on the map at any given time. In order to exclude such effects, a simulation was performed using 15 agents on the same map, all simultaneously learning collision avoidance. Following the simulation, post-processing was performed to average the collision history of all agents, and also smooth the data using a central moving average of total length 1000 to minimise the effects of random noise. The simulation output can be found in file `QlearningMultiAgent.sim`, and a graph of collision frequency and T as a function of simulation time is shown in

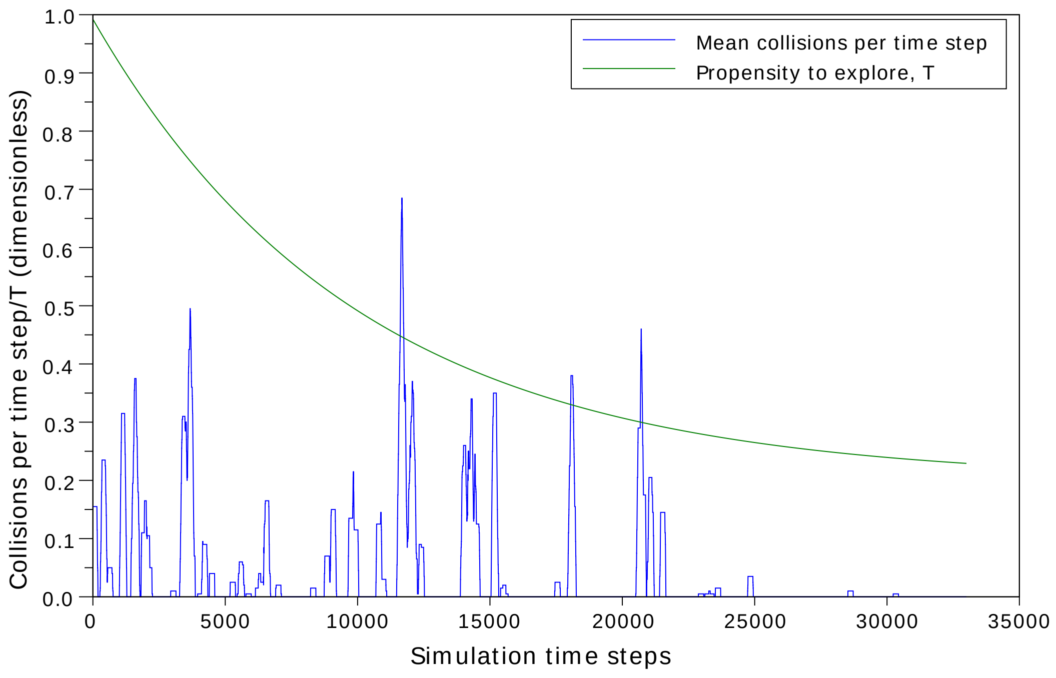


Figure 8.2: Mean number of collisions per time step, and propensity to explore map (T) for a simulation of a single agent using Q-learning to learn collision avoidance, $\beta = 0.0001$.

Figure 8.3 on the next page (again with $\beta = 0.0001$).

Evident from Figure 8.3 is a clear, positive correlation between T and the frequency of collisions. When T is near the minimum value of T_c, the collision frequency goes down to almost 0. This simulation was performed over 33 000 time steps, which is a relatively larger number compared to the other simulations in this project. Such a large number was used because, for $\beta = 0.0001$, it takes approximately that amount of time for T reach 95% of T_c. In order to determine the effect of T on the learnt behaviour, a similar simulation was performed, again using 15 agents on the same map, this time with $\beta = 0.005$. The results are shown in Figure 8.4 on page 99.

Figure 8.4 clearly demonstrates the effect of decreasing T too quickly: when agents have not had a sufficient chance to fully explore the state-action space, there will be certain sensory states for which they have not previously found the optimal action. When agents encounter such states, they will select the action they *believe* to be best, based on their incomplete approximation of Q. It is the actions taken in these states that correspond to the high frequency of collisions observed initially. Despite this, the agents represented in Figure 8.4 still eventually learn to avoid almost all collisions. Thus, the value of T does not affect an agent's rate of learning directly, but rather changes how likely the agent is to choose actions for state-action pairs that have previously been unexplored.

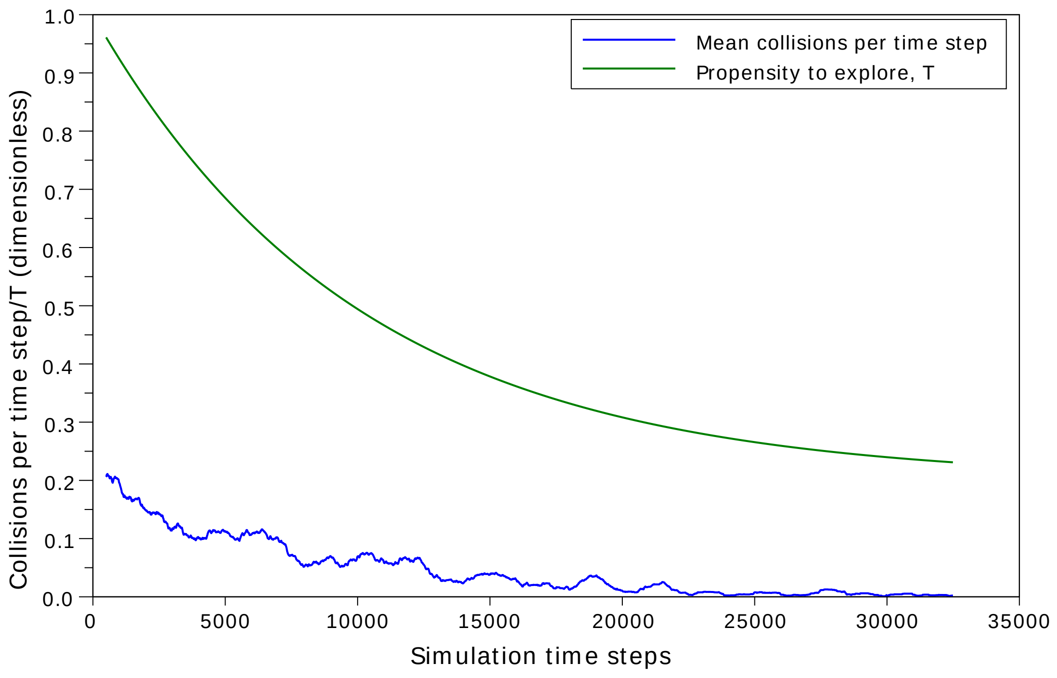


Figure 8.3: Mean number of collisions per agent per time step, and propensity to explore map (T) for a simulation of 15 agents using Q-learning to learn collision avoidance. Data is smoothed with a central moving average of total length 1001, $\beta = 0.0001$.

The effect of the rate at which T is decreased can be better understood by considering the total number of collisions as a function of simulation time, which is shown in Figure 8.5 on page 100. As can be seen, when T is decreased rapidly, the rate of collisions is higher initially, since agents have an incomplete approximation of Q, but are nevertheless using this approximation as if it were accurate. This results in a high rate of collision initially. However, agents which decrease T at a slower rate do not experience the same high rate of collision initially, but do experience a greater number of collisions overall. In both cases, the total number of collisions appear to plateau, which corresponds to agents having made an accurate internal mapping of state-action space to Q-values. This demonstrates the assertion made in the previous paragraph, namely that T does not affect an agent's ability to learn, but rather its tendency to explore new state-action pairs, which will indirectly affect its rate of learning optimal behaviour.

An important result of the foregoing analysis is that T is not a variable which can be optimised: it is, inherently, a *choice*. The designer (or agent, as the case may be) can choose between enduring a greater total number of collisions overall but fewer in the initial stages, or enduring more collisions at the start of learning but fewer overall. The rate of decreasing T is thus an indirect measure of the importance of short-term versus long-

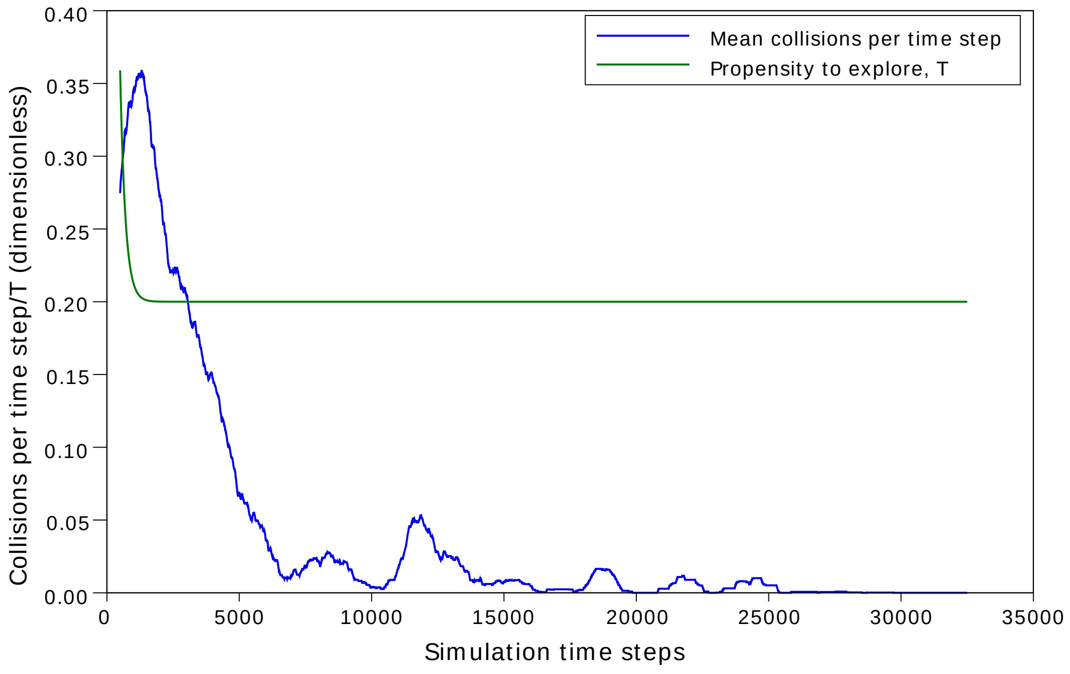


Figure 8.4: Mean number of collisions per agent per time step, and propensity to explore map (T) for a simulation of 15 agents using Q-learning to learn collision avoidance. Data is smoothed with a central moving average of total length 1001, $\beta = 0.005$.

term success. The very need to make the choice between these two mutually exclusive objectives encapsulates the exploration-exploitation dilemma.

Having discussed the effects of T on the learnt behaviour, attention will now be paid to some problems and anomalies that were observed during the simulations. Initially, the discrete action values used were the first four in Table 8.1, i.e. they did not include translational turns. This set of action values was thought to be sufficient for agents to explore the map and avoid obstacles. The reward function used was that shown in Figure 8.1 (a).

When using this reward function, agents would not explore the map in a straight line at full speed (as was desired), but would rather oscillate in all directions, with the result over a long period of time being a net forward movement. This was found to occur because the reward associated with actions in the "action-based" region of the reward function was fairly equally distributed. In other words, not enough emphasis was placed on moving forward in a straight line, namely action (1, 1), resulting in similar Q values for all actions. The consequence is that agents would perform the actions with a far greater frequency than was desired, leading to the behaviour just described.

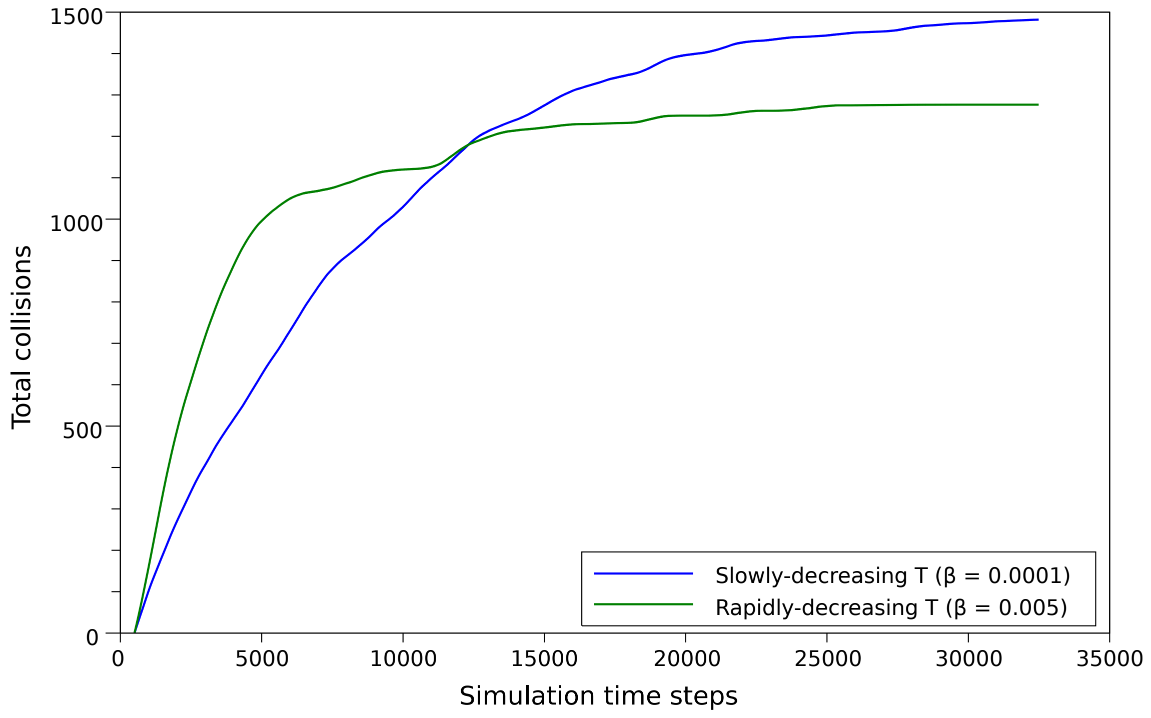


Figure 8.5: Total collisions per agent (average over 15 agents) as a function of simulation time for two different rates of decreasing T, $\beta = 0.005$ and $\beta = 0.0001$. Data is smoothed with a central moving average of total length 1001.

To solve this, the reward function was changed to that shown in Figure 8.1 (b), where a far greater emphasis is placed on action (1, 1). This was found to be highly effective at causing agents to move directly forward when not near obstacles.

As was mentioned, the first four action values shown in Table 8.1 were those used initially, and were thought to be sufficient for the task. However, an oscillatory behaviour was observed, as shown in Figure 8.6 (a) on page 101. If an agent became positioned such that it had an obstacle on its left or right side, it would simply oscillate back and forth, as shown in the figure.

After some testing it was discovered that, in the position shown, selecting any of the available actions would cause the agent to receive the same reward, a consequence of the agent's sensory configuration. It is clear that if the agent were to move forward or backward in this position, the value of i (its minimum sensor reading) would not change. Additionally, if the agent were to perform a stationary turn to the left or right, i would again not change. Therefore, in this state all actions appeared to have equal reward. Because the agent was not able to move out of this state into a different state, the Q value for all actions in the state shown would become equal. This means that the agent

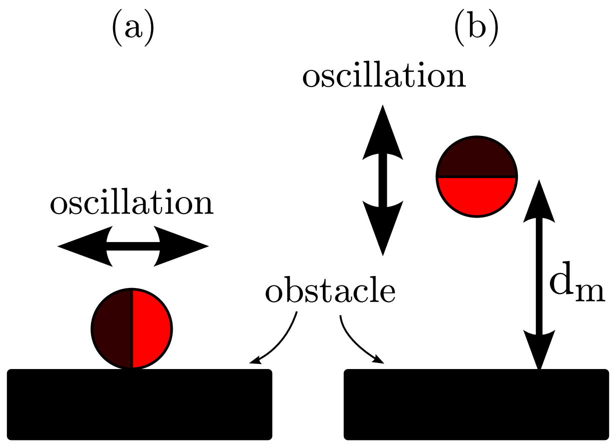


Figure 8.6: Oscillatory agent behaviours exhibited while learning collision avoidance.

would pursue all available actions in the current state, resulting in an oscillatory motion.

To solve this problem, the set of discrete action values was modified to include translational turns (i.e. forward turns and backward turns), shown in the last four rows of Table 8.1. Allowing agents to move forward while simultaneously turning would allow them to increase i marginally, thus resulting in a larger (less negative) reward. This solution proved to be effective, and after implementation the same class of oscillatory behaviour was not observed.

However, another class of oscillatory motion was observed, shown in Figure 8.6 (b). In certain cases, if an agent were to approach an obstacle at a roughly 90° angle, it would occasionally oscillate back and forth, at a mean distance from the obstacle equal denoted by d_m. It was found that this generally occurred with agents having a relatively low T value. The behaviour occurs because of a discontinuity in Q^* when $i = d_m$. Recall that agents attempt to approximate the function Q^*, which is unknown to them and represents the true utility of any state-action pair. The discontinuity in Q^* arises precisely because of the discontinuity in the reward function at $i = d_{crit}$. It is easier to analyse the reward function than the utility function Q^*, so the following analysis explains how the discontinuity in r leads to the oscillatory motion seen. However, since agents are only concerned with maximising Q values, it is actually the discontinuity in Q^* (or more precisely, agents' approximations thereof) that results in the behaviour observed. Therefore, the following analysis is exactly true in the case of $\gamma = 0$, when $Q^* \equiv r$, but true in principle for all cases. The relationship between $Q*$ and r will be further explored in the next section.

Consider the reward function graph shown in Figure 8.1 (b). If an agent's i value is equal to d_{crit} (on the left side of the "action-based" region), the action which will generate greatest reward is moving forward, i.e. action (1, 1). Taking this action will place the agent in the "state-based" region, at some i value $0 < i < d_{crit}$. However, the agent will receive negative reward for being in this state and will have previously determined that the best action to select in this state is an action that will increase i, i.e. any action with a backward component. Selecting such an action will bring the agent back to its initial i value of $i \approx d_{crit}$, allowing the process to repeat. The reason this behaviour is observed with agents having low T values is that the agent is more inclined to select the action it has previously determined to give the greatest utility for the current state. Agents with higher T values may undergo this oscillatory motion for a short period, but the agent is bound to eventually select an action it considers sub-optimal (such as a turn), which will move it out of this deadlock position.

This problem occurs as a direct result of the reward function used. Because in the "action-based" region there is a high reward associated with action (1, 1) and a very low reward associated with any other action, the agent is essentially compelled to select (1, 1). This becomes a problem when i reaches d_{crit} because the agent reaches a point of deadlock, alternately moving forward and backward. This problem can be solved in a number of ways. The simplest solution is to change the reward function so that the reward associated with action (1, 1) in the "action-based" region is not so comparatively high, thus allowing agents (even with low T) values to attempt other actions (such as turning). This could be achieved by reverting to the reward structure in Figure 8.1 (a), which was found to be an effective solution, although this would cause the previously-mentioned problem associated with that function. Another solution could be to smooth the reward function in the region of d_{crit} to eliminate the discontinuity there. Thus all actions could have a reward of 0 at $i = d_{crit}$, with the reward gradually increasing to its final value for each particular action.

In general, the oscillatory behaviours that have been observed are largely a consequence of the reward function used, as well as the discretised action space. Future simulations should certainly explore the use of continuous spaces.

A particularly interesting behaviour was observed in some agents that had learnt for a long period of time: when approaching an object, their method of avoiding it was to move backwards for a certain distance, then turn away and move in a different direction. Contrast this behaviour with the rule-based collision avoidance algorithm given in Chapter 5, where agents were instructed to perform either a stationary or forward turn when encountering an obstacle. If a forward turn is performed, agents would actually move *closer* to the obstacle before turning away from it. The method learnt using RL is arguably a more efficient way of moving away from an obstacle as quickly as possible, and this fact highlights the benefits of using RL to develop efficient agent behaviours, rather than having a human designer trying to decompose complex behaviours into a series of rules.

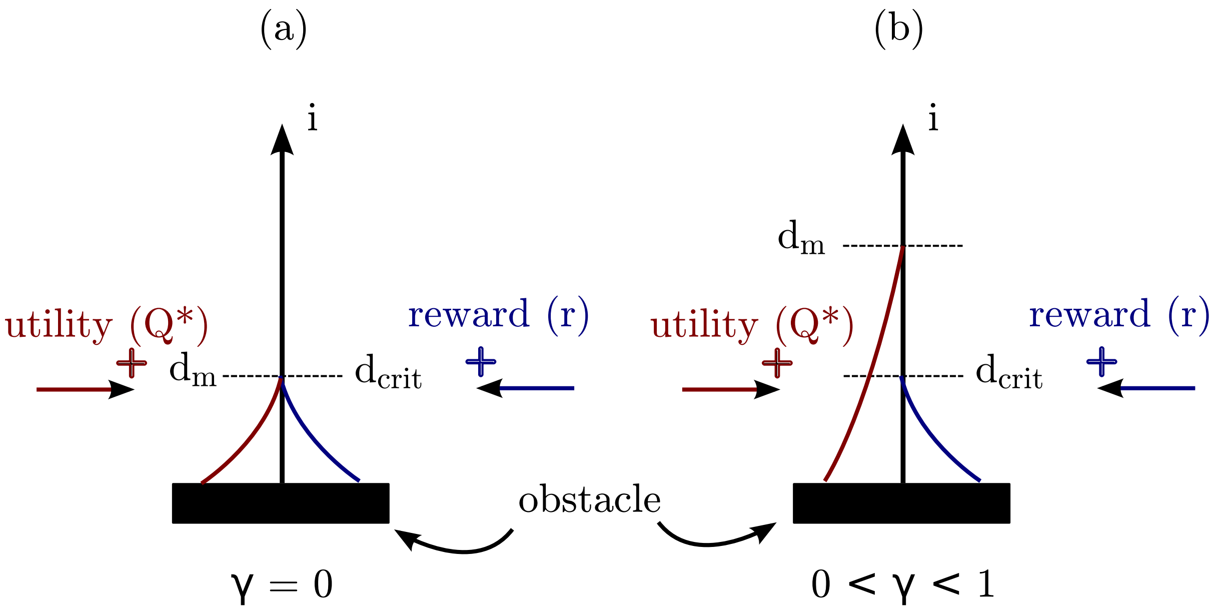


Figure 8.7: Relationship between utility and reward for the cases where (a) $\gamma = 0$ and (b) $0 < \gamma < 1$.

8.4 Effect of Learning Parameters

Thus far, the only learning parameter whose effect on the learnt behaviour that has been discussed is T. It was seen that T expresses a choice between exploration and exploitation and therefore cannot be optimised in an absolute sense. However, the effects of the other learning parameters, γ, α, η, have not yet been discussed. Recall that γ and α are the discount factor and Q-learning rate, respectively, and η is the learning rate of the neural networks used to map s to Q for each action a.

It has already been mentioned that γ weights the utility of future rewards relative to immediate rewards. Therefore, $\gamma = 0$ corresponds to $Q^* \equiv r$, leading to a highly "opportunistic" agent which only considers immediate reward and not long-term returns [2]. In the context of this collision avoidance problem, γ will control the distance from obstacles at which agents begin to avoid these obstacles, for agents whose approximation of Q has become sufficiently close to Q^*. This can be seen in the above simulations, where $d_{crit} \approx 0.4$, but the distance at which agents avoid obstacles is much larger (for a sense of scale, the agents had a diameter of 10 units). It was observed that changing the value of γ directly controlled the distance at which agents avoided obstacles. Denote this distance by d_m. The relationship between Q and r, for a simplified one-dimensional case, is shown in Figure 8.7.

As can be seen, γ controls to what extent agents anticipate the negative reward that is to follow if they approach obstacles. The precise value of d_m will be a function of d_{crit}

and γ, with $d_m \to \infty$ as $\gamma \to 1$.

The effects of the other two parameters are not as pronounced, as they are simply learning rates. The neural network learning rate, η, will control how rapidly the network will update its approximation of the mapping $s \to Q$, as well as the convergence properties of the network. The Q-learning rate, α, will determine how much an agent discards old knowledge and accepts new knowledge about Q^*.

It was mentioned earlier that variable T is a choice and cannot be optimised in a global sense. The same can be said for γ. Despite this, T and γ could certainly be optimised with respect to some goal, such as 'minimise the total number of collisions in the first 1000 time steps'. This could either be implemented in an offline approach, where simulations are repeated for different values of T, γ, and some optimisation approach used to find the best values. Alternatively, agents could be pre-programmed with the goal that their learning needs to achieve, and continually adjust the values of these parameters to approach this goal. It should be noted, however, that it is not entirely possible to escape the use of pre-defined constants, since even the process just described would require the specification of the rate at which new information is added and old information discarded.

8.5 Final Remarks

The reason for presenting the specific anomalies encountered during these simulations was not that they are believed to be fundamental or particularly significant. To the contrary, they are largely a consequence of the specific reward function that was employed. They have been included here to impart to the reader an appreciation of the difficulties that can be encountered with reinforcement learning, and artificial intelligence in general. While the overarching reason for using AI approaches in mobile robotics is to avoid the problems associated with a designer prescribing and proscribing the specific actions that are thought to lead to a particular outcome, it is equally true that the design of suitable reward functions is certainly not trivial. Indeed, it has been noted that the task of writing reward functions that elicit desired behaviour does not make the designer's work any easier, but rather changes the nature of this work [51].

Nevertheless, it is the author's opinion that AI techniques constitute a highly useful tool-set in the design of robotic agent controllers. If the goals of the task can be adequately specified (by creation of a suitable reward function), then it is almost certain that intelligent robotic agents will find an optimal or near-optimal approach to solving the problem.

Chapter 9

Swarm Behaviour In Distributed Resource Allocation

Thus far in this report it has been shown how the simulator can be used to model emergent swarm behaviour when agents are following simple rules, as well as how artificial intelligence can be used to make robotic agents learn novel and efficient ways ta complete a given task, which are often contrary (and superior) to behaviour-based approaches. The former case – using simple rules to elicit complex behaviour – demonstrates the mechanics of *how* swarms are able to form. However, the reasons *why* swarm behaviour emerges have not yet been addressed in the context of simulation. The latter is the goal of this chapter, which will present a highly simplified model of a real-world task which requires agent cooperation.

The chapter begins by motivating the need for a simple task to elicit cooperative behaviour in agents. Such a task is proposed together with the corresponding simulation model. The results of a simple investigation into two different behaviour-based approaches to the task are presented, followed by recommendations for future work.

9.1 Introduction

In Section 3.1 on page 14, a number of reasons were given to explain why swarm behaviour emerges in biological agents. Briefly, these reasons were to avoid predators, improve access to resources, increase locomotion efficiency and improve social interactions. While the last of these may not have an obvious parallel for robotic agents, the first three do. For example, aerial military drones carrying explosive payloads designed to destroy buildings may benefit from moving in swarms. The swarm may provide better protection against surface-to-air missiles (predators), improved explosive damage capability when operating in concord (access to 'resources') and also reduced aerodynamic drag from flying in formation (locomotion efficiency).

Of these three functions, the one that is simplest to implement in the simulator, and perhaps most realistic in the context of the swarm robots described in Section 3.3, is 'improved access to resources'. In the context of biological agents, "resource" usually refers to a foraging source or other source of nutrition. For robotic agents, the term could be interpreted as referring to any physical entity an agent must visit in order to complete a given task.

In this chapter, the task under consideration is that of robotic agents extinguishing fires. The application of this could be in a scenario, such as in a factory or warehouse, in which specialised robots are deployed and instructed to explore the area, extinguishing any fires if they are encountered. Due to the precise nature of the task, which will be given in the next section, some fires will be extinguishable by a single agent acting alone, but others will require multiple agents working together to be extinguished. This task therefore requires that robots act cooperatively in order to achieve a common goal.

9.2 Problem Definition

The goal of this section is to define the details of the task outlined above, so that cooperative behaviour between agents will result in a mutually beneficial outcome. As such, the fire-fighting model that will be developed is not intended to be completely realistic nor directly applicable to a real fire-fighting scenario. It is rather intended as an abstract resource allocation problem, designed to demonstrate the functional benefits of swarm behaviour and self-organisation.

Consequently, it is necessary to create a simulated model of an actual fire-fighting environment. The basic scenario is that agents are deployed on a map which has a number of *fire* objects positioned in various locations. These objects are a new kind of map entity introduced for this task. The model of the robotic agent has been adapted to include a radiation sensor, which allows agents to sense the temperatures of various fires. This sensor is positioned fore on the agent, as shown in Figure 4.6 (d) on page 32.

In order to develop a suitable model of a fire-fighting scenario, a number of qualitative requirements of the scenario were outlined, with the goal being to subsequently translate these descriptions into a mathematical model. These qualitative requirements are presented below.

1. The extinguishing capacity required to extinguish a fire should be dependent on the fire's temperature and the area of the specific fire.

2. The extinguishing capacity required to extinguish a fire should increase with the passing of time, making it advantageous to extinguish fires earlier, rather than later.
3. The extinguishing capacity of agents at a particular fire should be additive; that is, the rate at which a fire is extinguished should be proportional to the number of agents extinguishing the fire.
4. Fires should reach a terminal temperature, meaning that there are no runaway fires.
5. Agents should be able to extinguish fires at a constant rate.
6. An agent can only extinguish one fire at a given time.
7. The temperature of agents which are extinguishing a particular fire should increase as time goes by, and also increase at a rate commensurate with the temperature of the fire, as well as the area of the fire.
8. For the sake of simplicity, only agents which are close enough to a fire to extinguish it will absorb heat from that fire.
9. When an agent is extinguishing a fire and its temperature exceeds a particular threshold value, it should leave the fire to avoid physical damage.
10. Agents which have a temperature above a certain baseline should release heat to the environment, thus cooling down and allowing them to return to extinguish fires. This can be assumed to occur at a constant rate.

Based on the above requirements, mathematical modelling was performed to produce a set of equations governing the behaviour of the system. Detailed calculations for this model are given in Appendix B on page 141. Needless to say, this is a highly simplified model of a complex physical phenomenon. It should, however, be reiterated that the goal is not to perform an accurate simulation of the physics and thermodynamics inherent in fire-fighting, but rather to analyse the individual and group behaviours that are present in a task where agents need to work cooperatively but also ensure self-preservation.

The "resource" referred to in the title of this chapter is the extinguishing capacity of each agent. Since each agent can only extinguish one fire at a time, it is up to the agent to decide where it will direct its fire-extinguishing resource. A number of interesting consequences of this fact will be seen to emerge.

9.3 Simulation Setup

A number of agents are randomly placed on a map containing several fires. The map used for this simulation is shown in Figure D.4 on page 150. To simplify the simulation, the only internal objects in this map are fires, not obstacles, although it is entirely possible to use a combination of the two. Fires are always circular in shape, and drawn so that their apparent area is proportional to their 'area' property (which determines various characteristics of the fire, as mentioned above). The temperature of the fire is indicated by its colour: black for very low, red for maximum temperature and grey for an extinguished fire.

The assumption is made that agents' IR proximity sensors are able to detect the presence of fires, but not their temperatures. The radiation sensor mounted on the front of the agent is able to detect the temperature of a fire.

Agents are given a three-layer subsumption-based control algorithm, with the base layer being collision avoidance. The next behaviour is to approach a fire if one is detected by the IR sensors, provided that the agent's temperature is below a certain threshold, and begin extinguishing it until the agent's temperature reaches an upper threshold. The third subsumption layer is map exploration in a straight line.

Two different approaches were trialled when agents leave fires: turning away from the fire and going in search of another fire, or moving backwards until far enough away from the fire not to absorb its heat, and remaining stationary. In the latter case, when an agent's temperature has decreased sufficiently for it to resume fire-fighting, it will begin moving forward and approach the same fire at which it was previously. These two approaches will be referred to as 'strategy A' and 'strategy B', respectively.

Like the rest of the simulations detailed in this report, agents are not able to explicitly communicate with one another, nor do they have any knowledge of their environment beyond what their sensors detect.

9.4 Simulation Results and Discussion

Simulations were run for the two control strategies described above. The behaviour exhibited by agents, while a direct result of the particular model that was used, is nonetheless interesting. As expected, agents survey the map for fires, and when their sensors detect a fire, move towards it and begin extinguishing it. The constants of the fire model were chosen such that a single agent would not be able to extinguish a fire alone, but only with the assistance of other agents. The number of agents required would depend on

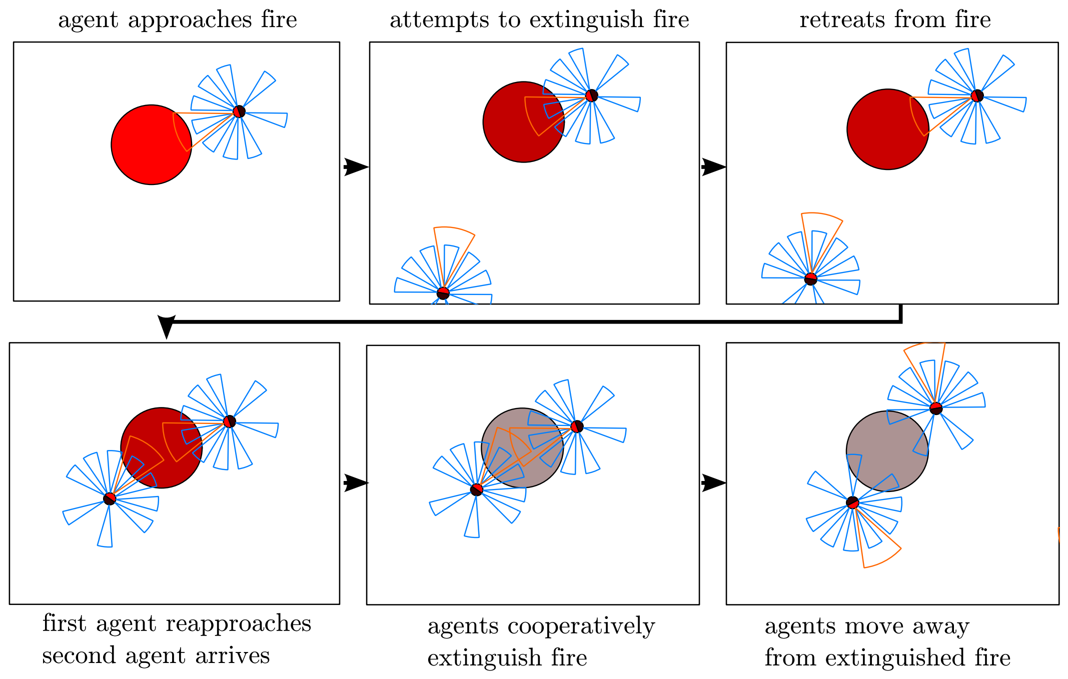


Figure 9.1: A series of frames depicting cooperative fire-fighting behaviour.

the temperature and area of the fire, the temperature of each agent, the control strategy being used, as well as the relative timing of the agents' arrivals at a particular fire.

Figure 9.1 depicts a typical scenario observed using strategy B. An agent approaches a fire and attempts to extinguish the fire. During the agent's time at the fire, its own temperature increases to a point where its damage-avoidance mechanism activates, and the agent is compelled to leave the fire, as shown. After having expelled enough heat to resume the task of fire-fighting, the agent moves forward and begins extinguishing the same fire. Coincidentally, a second agent arrives at the fire at approximately the same time. With both agents extinguishing the fire, its temperature drops more rapidly. Since the heat transferred to agents is proportional to fire temperature, this means both agents are able to stay at the fire longer than if only one agent were present. Consequently, the agents extinguish the fire and continue exploring the map.

It is interesting to note that neither of the agents extinguishing the fire are aware of the other's presence. Each agent is simply attempting to perform its assigned task, all the while ensuring that it does not suffer excessive damage to itself. Thus it could be said that the agents are implicitly cooperating, because neither agent has recruited the help of the other.

A similar, but not identical, behaviour can be seen in ants attempting to cooperatively transport prey. Typically, an ant will discover a prey and attempt to move it by lifting or dragging. If unable to move the prey immediately, the ant will spend a period of time attempting different angles and lifting methods. If other ants should arrive during this period, they will all attempt to move the prey. If the combined effort is sufficient, the ants will transport the prey to the nest, possibly unaware of the assistance of the other ants [3] (this is only true if the prey is *dragged*, rather than lifted). Where ant behaviour differs from the fire-fighting agents is that, if a single ant is unsuccessful after a long period of time, it will travel back to the nest to recruit more ants, while leaving a trail of pheromones to assist itself and other ants in finding the prey.

Having observed the above implicit cooperation between fire-fighting agents using strategy B, it is natural to ask whether strategy A might not be more effective. When using strategy A, an agent will leave a particular fire if it is not successful in extinguishing it. This approach could have the advantage that agents do not waste time attempting to extinguish fires beyond their individual capabilities, but instead move around the map and possibly find smaller fires they are able to extinguish alone. An obvious disadvantage, however, is that for larger fires it is far less likely that enough agents will happen upon the fire at the same time to cooperatively extinguish it. It is not clear *a priori* which effect will dominate, making a clear case for simulation.

Parallels can be drawn between robotic agents following strategy A and biological agents. For example, a predator on the hunt for prey may give chase to a potential prey, but if this potential prey proves to be elusive, the predator may instead opt to abandon that particular prey and hunt for another.

To perform a simulation to establish which strategy may be better, it is necessary to monitor the state of fires as the simulation progresses. However, a more robust analytical technique is required than simply observing the simulation as it progresses. To implement such a technique, the simulator was modified to record the history of each fire at each time step (or any user-defined interval of time steps). Both the number of agents at each fire is recorded, as well as the temperature of the fire. These results are then exported in matrix form, ready for subsequent analysis.

The graphs that are presented show the number of agents extinguishing each of the 7 fires on the map, as the simulation progresses. If a fire has been extinguished, it is shown as having -1 agents present, to distinguish it from fires which are active, but have no agents extinguishing them. The results of the simulations for agents using strategies A and B are shown in Figures 9.2 and 9.3, respectively.

As can be seen in the graphs for strategy A, all of the fires were extinguished, except for the one represented by the second graph, which corresponds to the large central fire (see map in Figure D.4 on page 150). It is also clear from this second graph that the fire never had more than one agent attempting to extinguish it at one particular time. This is due to each agent leaving the fire as soon as it had reached its threshold H value. Therefore,

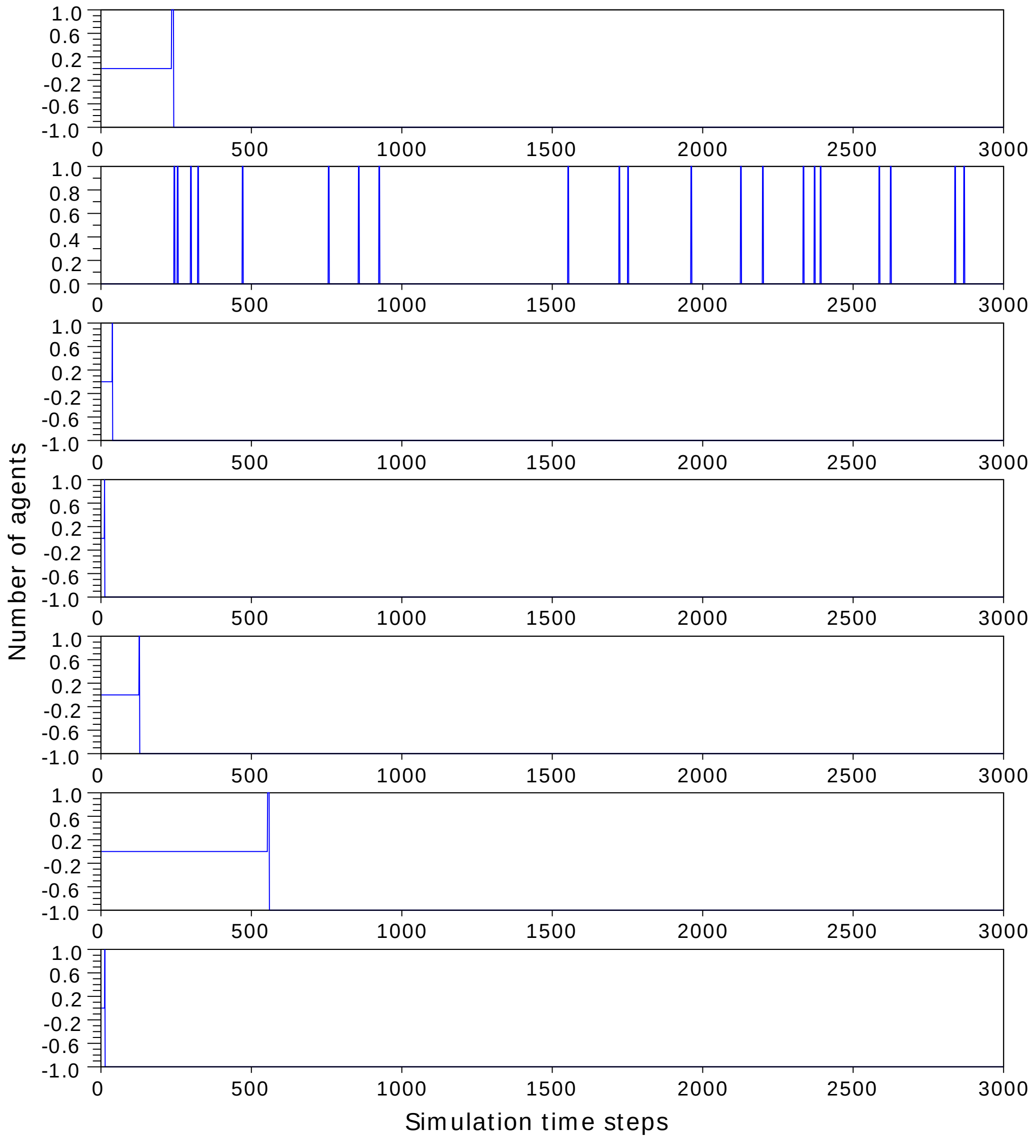


Figure 9.2: Fire history graphs for all fires in a fire-fighting simulation, showing the number of agents at each fire as a function of simulation time, for agents using strategy A.

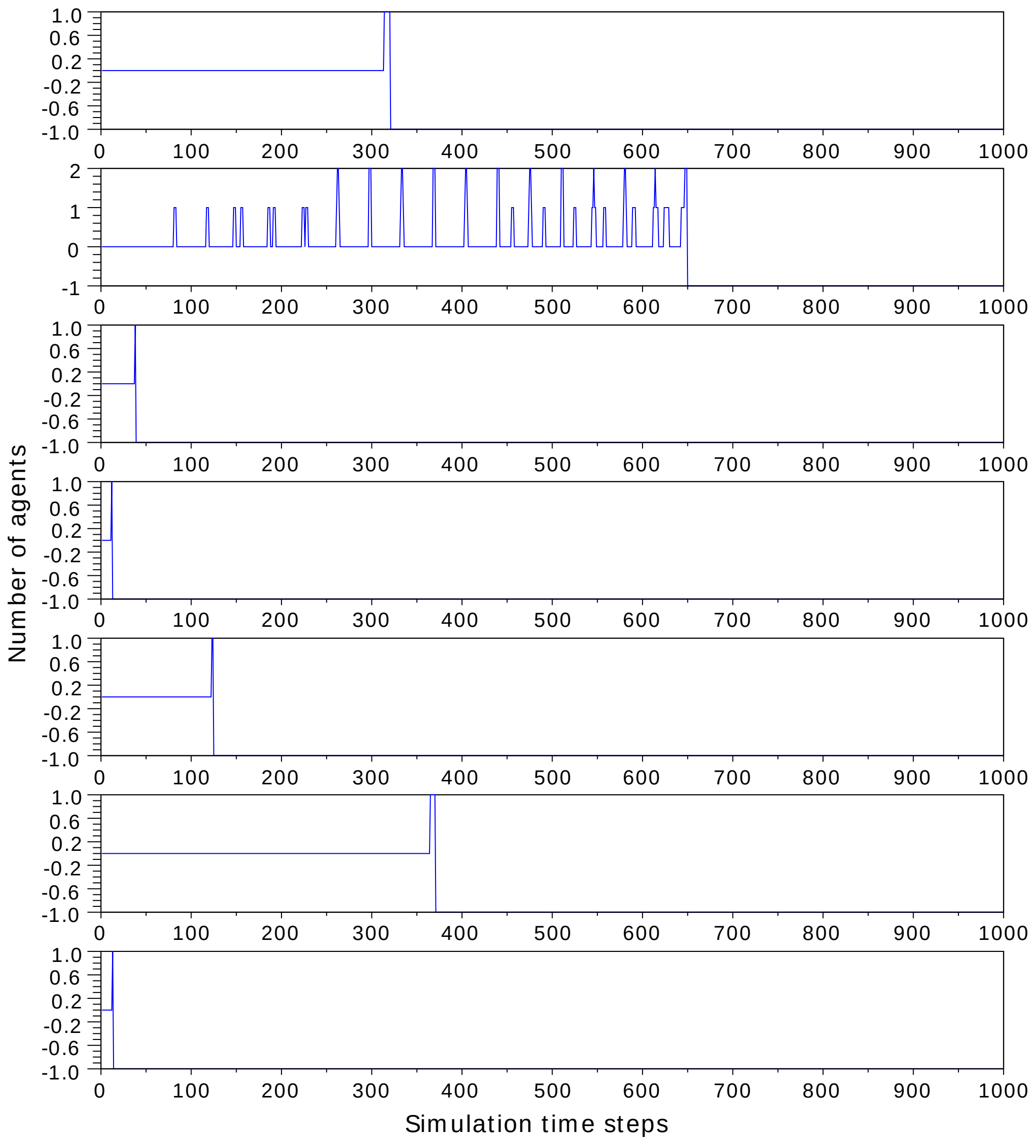


Figure 9.3: Fire history graphs for all fires in a fire-fighting simulation, showing the number of agents at each fire as a function of simulation time, for agents using strategy B.

in the case of strategy A, whether a fire will be extinguished by multiple agents depends on those agents arriving at the fire at very similar times, which is highly unlikely on a large map.

Contrast this behaviour with what is seen in Figure 9.3, with strategy B. The physical constants in the model were kept identical between the two simulations, meaning that agents stay at a fire for exactly the same length of time whether using strategy A or B. However, the benefit of strategy B is marked: by agents not leaving the fire when they are unable to extinguish it alone, they allow for the possibility that other agents will come to the same fire. When this happens, they are able to implicitly cooperate with each other, thus achieving their common goal, but without even knowledge of the presence of other agents. This closely resembles the cooperative transport behaviour in ants that was previously discussed.

The simulations referred to in this section can be found in files `fireFighting1.sim` and `fireFighting2.sim`.

9.5 Final Remarks

It is not intended that the simulations presented in this chapter should convince the reader of the superiority of one control strategy over the other. In any case, the results obtained are likely only applicable to this artificial model of distributed resource allocation. What is significant about the work presented here is that it broaches, in the context of a simulation, the *reason* that swarm behaviour might occur among autonomous agents. All of the robotic simulators presented in Section 4.2, as well as most swarm simulator software packages, focus on reproducing, on a superficial level, the patterns of swarm behaviour observed in nature. Perhaps a more fundamental and powerful approach to simulating swarms is that which has been presented in this chapter, namely to elicit swarm behaviour by creating *conditions* which might cause intelligent autonomous agents to decide to form close geometrical relationships with their peers, for the purpose of collectively achieving individual goals.

In this preliminary investigation, map exploration was implemented by means of moving in a straight line, in the same way as the simulations of Chapter 5. It was also observed that agents spent a large amount of time exploring the map without finding fires, even if there were active fires. It would therefore greatly improve the collective's fire-fighting ability if individual agents could find fires faster. However, this would seem to be unachievable in the absence of communication.

It was mentioned in Section 4.4 that agents without explicit communication can use their physical location in the environment as a form of implicit communication. Thus, it might be the case that agents which tend to follow other agents end up finding fires faster than those simply exploring in a straight line. Alternatively, some other heuristic not yet considered may prove to be even more effective. This type of problem is extremely well-suited to an artificial intelligence approach because the goal can be clearly defined, even if the steps to achieve it are unknown. Thus it is envisaged that agents could be given large positive rewards for happening upon fires, and small negative reward for every time step for which they are not at a fire (to encourage *fast* fire-finding behaviour). In this way, a reinforcement learning-based control structure could be used to make agents learn the actions best suited to finding fires.

It should be noted that the current implementation of RL in the simulator does not differentiate between obstacles and agents, as this was not necessary for the collision avoidance task previously presented. Therefore, if it is desirable that some kind of neighbour-based behaviour should emerge, the state space encoding would need to be altered to accommodate for sensors' values as well as object types.

Regrettably, there was insufficient time during this project to implement such an encoding and further explore the ideas mentioned above. However, the problem of distributed fire-fighting presented here, while perhaps somewhat artificial, is well-suited to eliciting swarm behaviour. The programmatic groundwork has been laid for such simulation to take place, and it is recommended that future work be undertaken in this domain.

Chapter 10

Critical Assessment

The purpose of this chapter is to review the work that has been undertaken in each section of the project, evaluate it against the objectives of the project, and make recommendations for related future work that might be undertaken.

10.1 Simulator

The simulator that was created is capable of modelling interactions between individual agents and their environment, as well as the behaviour of large groups of such agents (i.e. swarms). Therefore, the project objective of building such a simulator, described by item 1 on page 12 has been met. Furthermore, a number of additional features have been programmed into the simulator, beyond the stated requirements. These features were incorporated to make the software easier to use, more scalable and highly configurable. They include

- the separation of core simulation from visualisation;
- automatic regulation of visualisation playback speed;
- the creation and importation of custom maps;
- built-in analysis of certain common kinds of simulation, and subsequent exportation of data to other formats;
- the modular nature of the software in general, which permits configuration of new agent representations, sensory configurations and world objects;
- a simple graphical user interface; and

- the ability to enqueue serial simulations by the use of job files.

Despite the above features, there are nevertheless a number of areas in which the simulator might be improved or extended. These could include

- having a command line interface (CLI) so that simulation jobs can be executed from shell scripts or scheduled in advance;
- a more diverse array of built-in analysis tools;
- the ability to take screen shots natively inside the simulator, either on demand or at specific intervals;
- conversion of output `.sim` files into other formats, such as AVI or animated SVG;
- modification of the physics engine to permit cooperative transport tasks such as dragging, lifting and pushing;
- the ability to abort a simulation that is in progress, while retaining any data saved thus far;
- implementation of inter-agent communication models;
- built-in ability to parallelise computation;
- adaptation of the physics engine to allow arbitrary polygonal shapes; and
- a more efficient contact model that makes use of spatial partitioning.

10.2 Behaviour-based Control

Chapter 5 details the behaviour-based approach that was used to give agents simple rules to follow, which result in complex emergent group behaviours. Therefore, the project objective mentioned in item 2 on page 12, which was to "simulate swarm behaviour by giving agents simple rules to follow", has been met.

Suggestions for improvements and future work in this domain include

- solving the problem of leader determination when two agents approach head-on in a chain formation simulation;
- developing an algorithm that allows agents to perform velocity matching;

- investigating the effects of different agent locomotion and sensory configurations on the observed behaviour; and
- attempting to simulate other swarm behaviours, such as flocking, cooperative transport or perimeter formation.

10.3 Reinforcement Learning

The final project objective, given in item 3 on page 12, was to "program a reinforcement learning algorithm that is able to interact with the simulator and produce AI behaviour in each simulated agent". Such an algorithm was created for the behaviour of collision avoidance and is detailed in Chapter 7. Therefore, this project objective has been met.

Since artificial intelligence is a broad field, there are a number of possible points of departure for future work and investigation, which could include

- implementing different learning approaches, such as evolutionary algorithms or Hebbian learning;
- using continous state-action spaces as opposed to the discretised model that was employed;
- modifying the state-action space encoding to include the type of each object detected by sensors, which would be required for neighbour-based learning;
- investigating alternative reward functions for the collision avoidance task; and
- having agents which are aware of their own learning, and thus able to self-adjust the parameters thereof.

10.4 Distributed Resource Allocation

This task was pursued as a purely exploratory endeavour and has no corresponding 'project objective'. Despite this, a substantial amount of work was done in creating a model that can be used to elicit swarm behaviour in intelligent agents from first principles, as opposed to using a rule-based approach to merely emulate such behaviour. Therefore, AI techniques would be extremely well-suited to this problem. Topics for further investigation could include

- implementing communication between agents to observe the effect on group efficacy;

- a reinforcement learning algorithm to reward agents for finding fires, thus possibly causing swarm behaviour; and
- a reinforcement learning algorithm to reward agents for *extinguishing* fires, which could reveal problem-solving heuristics that are anticipated, such as agents first congregating before extinguishing a fire.

Chapter 11

Conclusions

The phenomenon of swarm behaviour is remarkable: a large number of individual agents, each acting in the interest of their own self-preservation, self-organising into swarms that are able to cooperatively realise individual and group goals.

This report has documented, in some detail, the design, development and applications of a computer program to simulate swarm behaviour.

The notion of swarm behaviour was introduced in Chapter 3 with a survey of its interpretations in nature and robotics. Some of the natural patterns of swarms that can be observed were mentioned.

Attention was then paid to the development of the computer simulator. A survey of existing simulators and best practices in the field was presented in order to inform and justify the design decisions that were made. Following this, the physical model that was used in the simulator was explained in depth, with reference to detailed calculations contained in an appendix. Descriptions of the details of the specific implementation were given, as well as comprehensive performance testing and analysis.

Having explained the development of the simulator, the notion of behaviour-based control was then introduced, with descriptions of the various behaviours that were implemented. The results of simulations using these behaviours were presented and analysed, describing the complex emergent behaviour that was seen to result from the application of simple rules on a large scale.

A survey of relevant literature on artificial intelligence was then presented. Neural networks and reinforcement learning algorithms were discussed in depth, in order to lay the groundwork for the AI simulations that followed.

The reinforcement learning algorithm that was developed during the project was then presented and described in detail. The specifics of its implementation were explained and related back to their theoretical foundations in the foregoing chapters. The results of the corresponding simulations were then presented, with an analysis of the numerous anomalies and unexpected behaviours that were observed. In particular, it was shown how the use of AI techniques does not necessarily make the job of the designer any easier, but rather changes the nature of the work that is required. In spite of this, reasons were given for why AI approaches to agent models can often be superior to the behaviour-based equivalents.

In the final simulation-based chapter, the distributed resource allocation problem of multiple fire-fighting agents was proposed and described, with the goal being to create a scenario where swarm behaviour might emerge out of *requirement*, as opposed to prescribing rules that mechanistically result in its occurrence. Some simple simulations were conducted to compare two rule-based cooperative approaches, with the caveat that the simulation results were likely specific to the nature of the problem, and did not speak to a more fundamental cause. The use of an AI approach to elicit self-organised cooperation for the completion of the task was contemplated, despite there having not been time to implement it programmatically.

Finally, the work done in the project was reviewed against the project objectives and critically assessed. A number of recommendations were made, the most important of which were to optimise the simulator's contact model using spatial partitioning, investigate different learning approaches (such as evolutionary algorithms) and to implement a state-action space encoding capable of differentiating between agents and obstacles.

It is hoped that the reader is left with an appreciation of the prevalence of swarm behaviour across multiple domains, as well as an understanding of the important role played by computer simulation in modelling these complex interactions.

References

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

[2] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA., 1998.

[3] C. R. Kube and E. Bonabeau. Cooperative transport by ants and robots. *Robotics and Autonomous Systems*, 30:85–101, 2000.

[4] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems.* Oxford University Press, 1999.

[5] E. Csahin. Swarm robotics: From sources of inspiration to domains of application. *Swarm Robotics*, pages 10–20, 2005.

[6] R. C. Arkin. Cooperation without communication: Multi-agent schema based robot navigation. *Journal of Robotic Systems*, 9(3):351–364, 1992.

[7] M.L. Swinson and D.J. Bruemmer. Expanding frontiers of humanoid robotics. Technical report, DTIC Document, 2000.

[8] F.E. Fish. Kinematics of ducklings swimming in formation: consequences of position. *Journal of Experimental Zoology*, 273(1):1–11, 1995.

[9] U. Boryczka. Finding groups in data: Cluster analysis with ants. *Applied Soft Computing*, 9(1):61–70, 2009.

[10] A. Orange. Letters 158. `http://www.orange-papers.org/orange-letters158.html`. Accessed: 15/10/2012.

[11] Author unspecified. Photos and info on ants and termites in Malaysia. `http://termitesandants.blogspot.com/2009/10/macrotermes-gilvus.html`. Accessed: 15/10/2012.

[12] M Clune. How scent can net bigger reward for ants in hunt for food. `http://www.sussex.ac.uk/research/news/?id=14359`. Accessed: 15/10/2012.

[13] D. Dibenski. Auklet flock. `https://en.wikipedia.org/wiki/File:Auklet_flock_Shumagins_1986.jpg`. Accessed: 15/10/2012.

[14] J. Seyfried, M. Szymanski, N. Bender, R. Estana, M. Thiel, and H. Woern. The i-swarm project: Intelligent small world autonomous robots for micro-manipulation. *Swarm Robotics*, pages 70–83, 2005.

[15] H. Woern. I-SWARM. `http://www.i-swarm.org`. Accessed: 14/10/2012.

[16] M. Dorigo, E. Tuci, R. Groß, V. Trianni, T. Labella, S. Nouyan, C. Ampatzis, J.L. Deneubourg, G. Baldassarre, S. Nolfi, et al. The swarm-bots project. *Swarm Robotics*, pages 31–44, 2005.

[17] Author unspecified. Darwin's robots. `http://hplusmagazine.com/2009/10/05/darwins-robots/`, October 2009. Accessed: 16/10/2012.

[18] M. Rieger. Simple robotic platform for swarm behaviour modelling, October 2010. UCT Undergraduate Final Year Project.

[19] B. Gerkey, R. Vaughan, and A. Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the International Conference on Advanced Robotics.*, pages 317–323, 2003.

[20] PGCN Senarathne, WS Wijesoma, KW Lee, and B Kalyan. MarineSIM: robot simulation for marine environments. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5603839`, 2010. Accessed: 11/10/2012.

[21] A. Dickie. Modeling robot swarms using agent-based simulation. Master's thesis, Naval Postgraduate School, Monterey, CA., 2002.

[22] Microsoft. Microsoft robotics developer studio: Simulation overview. `http://msdn.microsoft.com/en-us/library/bb483076.aspx`, 2012. Accessed: 11/09/2012.

[23] R.A. Brooks. Artificial life and real robots. In F.J. Varela and P. Bourgine, editors, *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, Cambridge, MA, 1992. MIT Press/Bradford Books.

[24] O. Miglino and S Nolfi. Evolving mobile robots in simulated and real environments. *Artificial Life*, 2:101–116, 1995.

[25] N. Jakobi, P. Husbands, and I. Harvey. Noise and the reality gap: The use of simulation in evolutionary robotics. In F. Moran, A. Moreno, J. Merelo, and P. Chancon, editors, *Advances in Artificial Life: Proceedings of the Third European Conference on Artificial Life*, Berlin, 1995. Springer Verlag.

[26] Author unspecified. Stage: 2D multiple robot simulator. `http://playerstage.sourceforge.net/stage/stage.html`. Accessed: 11/10/2012.

[27] Daisy Tang. Player/stage howto. `http://www.csupomona.edu/~ftang/courses/player%20stage/player-stage-intro.htm`, September 2011. Accessed: 15/09/2012.

[28] L. Hugues and N. Bredeche. Simbad: an autonomous robot simulation package for education and research. *From Animals to Animats 9*, pages 831–842, 2006.

[29] C.W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *ACM SIGGRAPH Computer Graphics*, volume 21, pages 25–34. ACM, 1987.

[30] O. Michel. Webots: Professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1:39–42, 2004.

[31] Cyberbotics. Webots: Commercial mobile robot simulation software. `http://www.cyberbotics.com`. Accessed: 11/10/2012.

[32] F. Arvin, K. Samsudin, and A.R. Ramli. Development of a miniature robot for swarm robotic application. *International Journal of Computer and Electrical Engineering*, 1(4):436–442, 2009.

[33] T Balch and R. C. Arkin. Communication in reactive multiagent robotic systems. *Autonomous Robots*, 1:1–25, 1994.

[34] D. O'Leary. Guidelines on writing scientific computing software. `http://www.cs.umd.edu/~oleary/c661/softwarestandards.pdf`, January 2010. Accessed: 5/10/2012.

[35] R.C. Gentleman, V.J. Carey, D.M. Bates, and B. Bolstad. Bioconductor: open software development for computational biology and bioinformatics. *Genome Biology*, 5(10):Article R80, 2004.

[36] M. Lutz. *Learning Python.* O'Reilly Media, 2009.

[37] M. Lin and S. Gottschalk. Collision detection between geometric models: a survey. In *Proceedings of the Conference on Modeling in Computer Graphics*, 1998.

[38] Scalable Vector Graphics (SVG) 1.1 (second edition). W3C recommendation. `http://www.w3.org/TR/SVG/`. Accessed: 27/09/2012.

[39] R.A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2:14–23, 1986.

[40] R. C. Arkin. Motor schema-based mobile robot navigation. *International Journal of Robotics Research*, 8:92–112, 1989.

[41] K. Gurney. *An Introduction to Neural Networks*. Routledge, London, 1997.

[42] G. Orr. CS-449 neural networks lecture notes. `http://www.willamette.edu/~gorr/classes/cs449/intro.html`, 1999. Accessed: 28/09/2012.

[43] W. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[44] Heni Ben Amor, Shuhei Ikemoto, Takashi Minato, Bernhard Jung, and Hiroshi Ishiguro. A neural framework for robot motor learning based on memory consolidation. Technical report, TU Bergakademie Freiberg, 1997.

[45] Abdallah El Ali, Loes Bazen, Iris Groen, Elisa Hermanides, Wouter Kool, David Neville, and Kendall Rattner. Forget-me-net: Overcoming catastrophic forgetting in back-propagation neural networks. In *Proceedings of the CSCA Summerschool*, 2008.

[46] L. Wang, C.K. Tan, and C.M Chew. *Evolutionary Robotics: From Algorithms To Implementations*. World Scientific, 2006.

[47] G Tesauro. Programming backgammon using self-teaching neural nets. *Artificial Intelligence*, 134:181–199, 2002.

[48] C.J.C.H. Watkins and P. Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[49] B Huang, G Cao, and M Guo. Reinforcement learning neural network to the problem of autonomous mobile robot obstacle avoidance. In *Proceedings of the Fourth International Conference on Machine Learning and Cybernetics*, pages 85–89, 2005.

[50] Chris Gaskett, David Wettergreen, and Alexander Zelinsky. Q-learning in continuous state and action spaces. In *Australian Joint Conference On Artificial Intelligence*, 1999.

[51] L.P. Kaelbling, M.L. Littman, and A.W. Moore. Reinforcement learning: A survey. *arXiv preprint cs/9605103*, 1996.

Appendices

Appendix A

Physics Modelling and Calculations

The kinematics and kinetics employed in the simulator are relatively simple. The high-fidelity physics engine that was initially implemented made use of conservation of momentum and coefficients of restitution to resolve the kinetics of collisions. However, in the lower-fidelity engine that was eventually implemented, the collision model was replaced by one which, although based less on the classical principles of physics, still provides behaviour which is sufficiently realistic.

This appendix first gives a technical treatment of the rigid body kinematics of differential wheel locomotion, before describing the contact model that was employed. The method of sensor operation is explained, followed by a brief note on the coordinate systems used in the simulator.

A.1 Rigid Body Kinematics

The physics engine uses a simplified kinematic model. Essentially, all agents are considered rigid bodies, with their velocities determined by the linear velocity of their wheels at the outer radius. The following assumptions are made:

1. Both wheels are of the same diameter and have the same coefficient of friction with the ground.
2. The agent is able to rapidly and independently control motor speeds to effect translation and rotation.
3. The wheels may slip for one of two reasons. Either the agent is attempting to push into another object, or random slip is being applied to simulate unpredictable motor behaviour (see Section 4.3.2.2 on page 34).

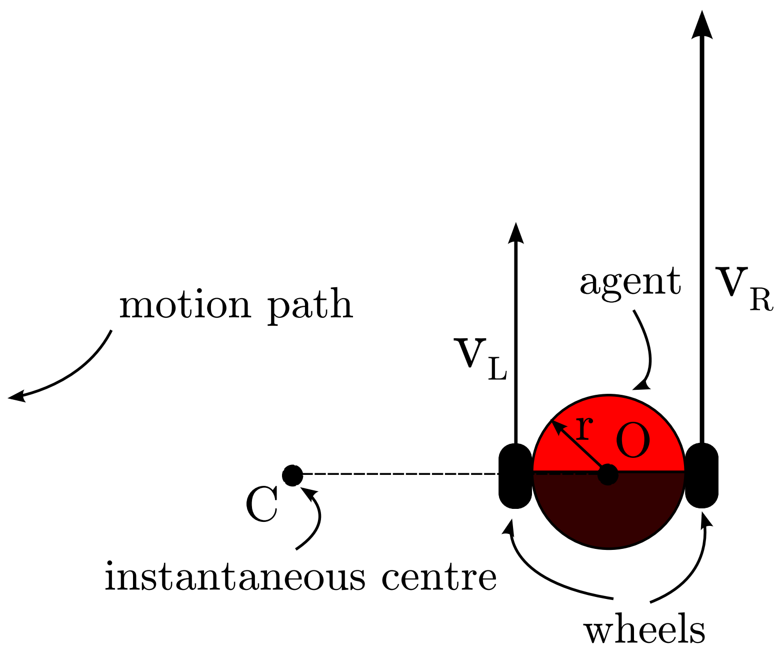


Figure A.1: Agent velocity vector diagram overlaid on agent drawing.

4. Both wheels are aligned to be parallel and coaxial with each other.

Denote the left and right wheel velocities v_L, v_R, respectively. These variables refer to the *no-slip speed* of the wheels; the actual motor speeds (m_L, m_R) may be higher. For example, if the left wheel is slipping, m_L would be some non-zero value known to the agent, whereas $v_L = 0$ and is unknown to the agent. Therefore, the agent does not know directly whether it is moving or not; it can only know whether its motors are active.

Consider the entire agent to be a single rigid body with velocities relative to the ground of v_L, v_R as shown in Figure A.1. It is then possible to determine the agent's *instantaneous centre of zero velocity* (or just *instantaneous centre* for brevity), denoted by **C**. Let **O** be the midpoint of the line joining the two wheels' centres, as shown in the figure.

Since the agent is a rigid body, it is possible to write

$$v_R = (||\vec{\mathbf{CO}}|| + r)\omega \qquad\qquad v_L = (||\vec{\mathbf{CO}}|| - r)\omega \tag{A.1}$$

where ω is the rotational velocity of the agent, positive in the counter-clockwise sense. Solving for ω,

$$\omega = \frac{v_R - v_L}{2r} \tag{A.2}$$

Similarly, the velocity of point **O**, v_O is given by

$$v_O = ||\vec{\mathbf{CO}}||\omega \tag{A.3}$$

Substituting (A.1) into (A.3)

$$v_O = \frac{v_R + v_L}{2} \tag{A.4}$$

Equations A.2 and A.4 describe the rotational and translational velocities of the agent. Because the assumption has been made that the wheels are coaxial and parallel, point **O** is coincident with the centre of the circle which describes the agent. At each time step, the position of this centre is updated according to equation A.4, and the orientation of the agent is updated according to eqaution A.2. This operation is performed in the `refresh()` method of the `AgentCir` superclass in file `agents.py`, and is therefore inherited by all agent classes based on the `AgentCir` superclass.

A few comments on these results are in order. Clearly, for the agent to move in a straight line, $v_L = v_R$. This fact is used by all the control strategies implemented in this project, to either force or encourage (in the case of reinforcement learning) the agent to move in a straight line. Positive values of v_L, v_R result in forward motion, negative values result in backward motion. If $v_L \neq v_R$, the agent will turn, with the rate of turn (ω) being proportional to the disparity in v_L and v_R. In the extreme case where $v_L = -v_R$, the agent will execute a stationary turn (i.e. no translational motion). This may be useful in a number of cases. For example, in collision avoidance algorithms where a turn needs to be made without the possibility of colliding while making said turn. Conversely, 'translational turns' are turns when the agent's centre (point O in Figure A.1) is not stationary, which can be useful in avoiding deadlock (see discussion in Section 5.2.2 on page 56).

Define the fractional wheel slip on each wheel to be

$$ws_L = \frac{m_L - v_L}{m_L} \quad \text{and} \quad ws_R = \frac{m_R - v_R}{m_R} \tag{A.5}$$

where 0 indicates no slip and 1 indicates full slip. The value of ws is not known to the agent; it is determined by the physics engine based on whether the agent is attempting to travel into another object, or based on random slip. The values of v_L, v_R are calculated once ws_L, ws_R are known.

A.2 Contact Model

As mentioned in Section 4.3.3 on page 35, the simulator uses a simplified collision model and a relatively expensive contact detection algorithm. For the purposes of this discussion, ‘world’ refers to the collection of all objects (obstacles, agents, fires) in the simulated environment at any given time.

As described in Section 4.6, at each time step the world loop first refreshes all objects in the world, which process involves, inter alia, calculating the new positions and orientations of all moving objects. Following this refresh, the contact detection algorithm is invoked pair-wise on all objects in the world. If an overlap of two objects is detected, the collision algorithm is applied to those two objects. The contact detection and collision algorithms described below are both contained in file `contactFunc.py`.

A.2.1 Contact Detection

The contact detection algorithm takes, as its input, two world objects. These two objects will be checked to see if they have any overlapping areas. At present, the contact model used in the simulator is capable of checking rectangle-circle and circle-circle pairs, the former case corresponding to obstacle-agent contact and the latter to both obstacle-agent and agent-agent contact. It is certainly possible to extend the algorithm to perform contact detection on other geometries.

The general case of two proximate circular objects is shown in Figure A.2 on the following page. It is apparent that contact has occurred if the following inequality holds:

$$||\vec{\mathbf{C_A C_B}}|| < (r_A + r_B) \tag{A.6}$$

The case of contact between a rectangular and circular object is slightly more complicated. The general case is shown in Figure A.2.1 on the next page. Conceptually, a frame with rounded corners can be envisaged around the rectangle (shaded region in the figure). This frame has width equal to the radius of the circular object. If the circular object’s centre lies anywhere within this frame, contact has occurred, as is the case with circle B. In order to check whether the circular object’s centre lies within this frame, a further conceptual construct is introduced: a bounding box, as shown in the figure.

Initially, the circle’s centre is compared against the coordinates of the bounding box. If it lies within the bounding box, a further calculation is performed to classify in which area of the bounded box it lies. If the centre lies in the light-shaded region, contact has occurred. If the centre lies within the bounding box but not in the light-shaded regions (as is the case with circle A), the distance is measured between the circle’s centre and

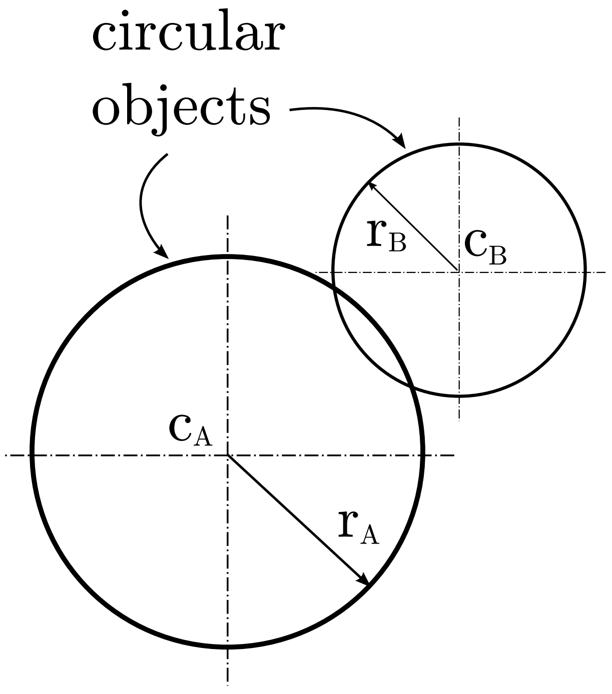


Figure A.2: Schematic showing case of two circular objects in contact.

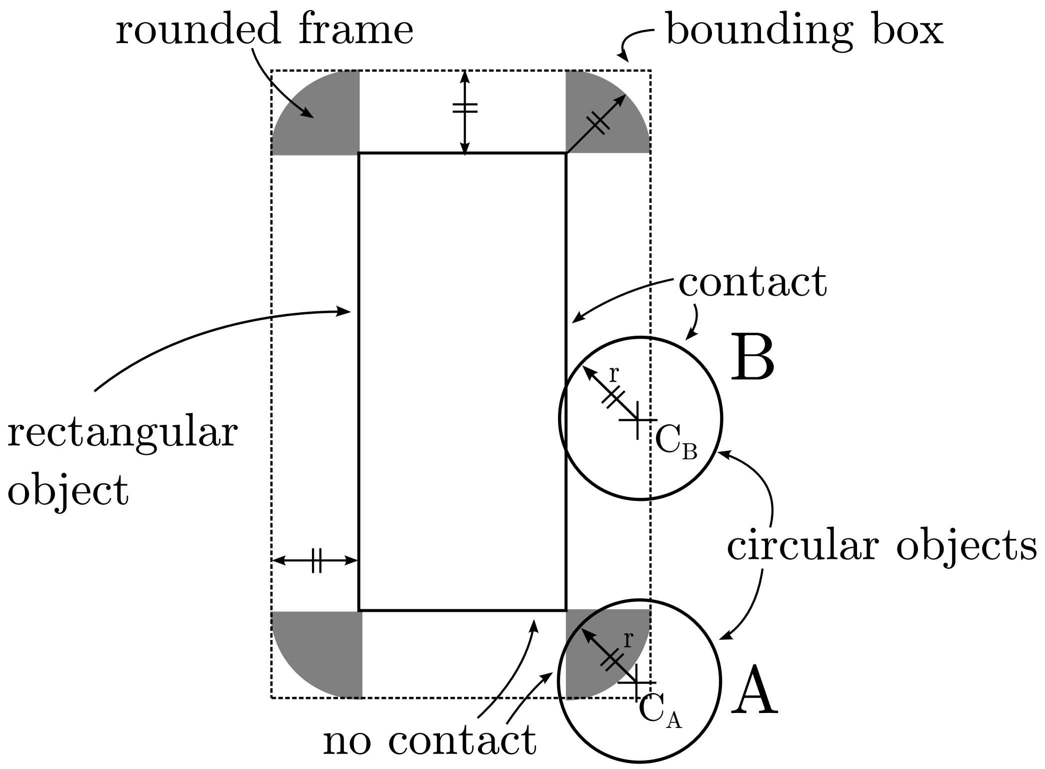


Figure A.3: Schematic showing case of circular objects making contact and near-contact with a rectangular object.

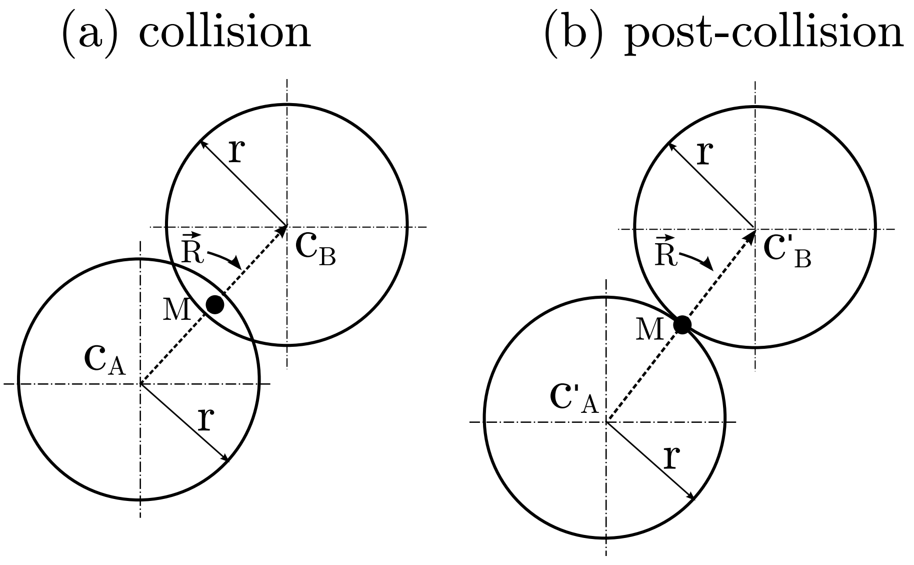


Figure A.4: Method of separation of two colliding circular objects.

the corresponding vertex of the bounding box. If this distance is less than the circle's radius, the centre lies within the dark-shaded region and contact has occurred.

A.2.2 Collision Model

The collision model works on a simple principle: if the angle between an agent and a rectangular object with which it is colliding is greater than a critical angle θ_{crit} (measured from the object's normal), then the agent is turned so as to be parallel with the object at the point of collision. If the angle of collision is less than a critical angle, the agent slides against the obstacle. This was illustrated in Figure 4.8. With circular objects, sliding always occurs, as shown in 4.9 on page 37.

Since a discrete contact model is used, contact is only detected after the event. This implies that, to prevent overlapping matter, two objects which have collided need to be separated first, after which other collision rules can be applied (such as that described in the previous paragraph). In the initial physics engine that was written for the simulator, collisions were resolved by applying the principle of conservation of momentum together with coefficients of restitution for agent-obstacle and agent-agent interactions.

In the present simplified model, the post-collision positions of the two objects are determined as follows. For agent-obstacle collisions, obstacles' positions do not change (or from a physical perspective, obstacles are regarded as having inertial mass infinitely higher than that of agents). For agent-agent collisions, agents are regarded as having equal inertia and therefore their post-collision positions differ from their positions at the moment of impact by an equal amount.

Post-collision positions are calculated differently for circular-circular interactions and for rectangular-circular interactions. In the former case, suppose two circular objects have collided and overlap as shown in Figure A.4 on the previous page. For the moment, suppose this is an agent-agent collision, so the circles are the same size. Construct a direction vector $\vec{\mathbf{R}}$ from the centre of circle A to the centre of circle B, as shown. Also construct point $\mathbf{M}$, the midpoint of $\vec{\mathbf{R}}$. Note that $\mathbf{C_A}, \mathbf{C_B}, \mathbf{M}$ are position vectors. Then the new positions of the circle centres, $\mathbf{C'_A}, \mathbf{C'_B}$ are given by

$$\mathbf{C'_A} = \mathbf{M} - \frac{\vec{\mathbf{R}}}{||\vec{\mathbf{R}}||} \times r \qquad\qquad \mathbf{C'_B} = \mathbf{M} + \frac{\vec{\mathbf{R}}}{||\vec{\mathbf{R}}||} \times r \tag{A.7}$$

Equations A.7 are applicable for the case of an agent-agent collision, under the assumption that two agents colliding will displace equally post-collision. However, this assumption may not always be true, for at least two reasons. Firstly, the assumption has been made that the agents have identical inertias; however, inertia is not relevant here as the agents are in contact with the ground via their wheels. This means that in order for an agent to be displaced, its wheels have to slip. The maximum restraining frictional force between the ground and agent wheels is proportional to the normal force, i.e. mass of the agent. Secondly, it is more likely for the agent to displace in the longitudinal than transverse direction (although this may depend on the type of motor control used). However, the model does not take this anisotropy into account, but rather assumes that agents' wheels are equally likely to slip in any direction. If this model were to be revised to eliminate these assumptions, the way point $\mathbf{M}$ is determined would change. Instead of $\mathbf{M}$ being the midpoint of $\vec{\mathbf{R}}$, it would lie at some other position on $\vec{\mathbf{R}}$ in the area of overlap between the two circles.

Equations A.7 are easily adapted for obstacle-agent collisions. Assume circle A is the obstacle and circle B the agent. By simply moving point $\mathbf{M}$ to the intersection of $\vec{\mathbf{R}}$ and circle A's circumference, and applying only the second of equations A.7, circle A will remain stationary and circle B will move so it is no longer overlapping with A.

In the case of a circular object colliding with a rectangular object, a similar process of separation is followed. If, after the collision time step, the centre of the circle lies in any of the dark-shaded regions on Figure A.2.1 on page 130, it is treated as a circular-circular interaction and the separation process just described is enforced. If the circular object's centre falls in a light-shaded region (such as circle B in the figure), the centre is moved so that it is just touches the nearest rectangular edge.

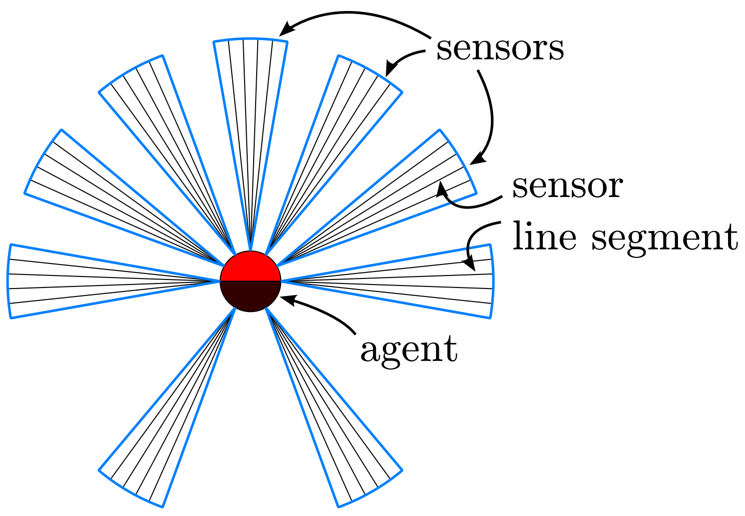


Figure A.5: A typical agent sensor configuration, showing the decomposition of the sensor field of vision into a number of finite line segments.

A.3 Sensor Operation

In general, IR sensors have been modelled as having a finite field of vision covering an area equal to a segment of a circle. The sensor returns a scalar value equal to the distance to the closest object, or part thereof, that lies within the field of vision. The sensing operation has been modelled by splitting the continuous sensor range into a number of discrete, finite line segments, as shown in Figure A.5. The density of these line segments is variable, and can be changed in the `IRSensor` class in the `agents.py` file.

When a sensor's `sense()` function is called, each of these finite line segments (hereafter referred to as 'sensor line segments') is checked against every object in the world, except its 'parent' agent. This process involves first checking if a proximity condition is met, to save computational expense on objects that are not near the sensor, followed by a calculation of the shortest distance to the object, if it intersects with the line segment. These calculations are detailed for circular and rectangular objects in Sections A.3.2 and A.3.3, respectively.

A.3.1 Sensor Positioning

Each instance of the `IRSensor` class of infrared sensor objects has four intrinsic attributes: centre position ($\mathbf{C_s}$), orientation (Θ_s), depth of vision (r) and angular range. The first two of these are in global coordinates (as they are actual properties of the sensor itself) and are stored as such in the `IRSensor` object. However, the `Agent` object corresponding to a particular sensor stores more information about the sensor, including the sensor's

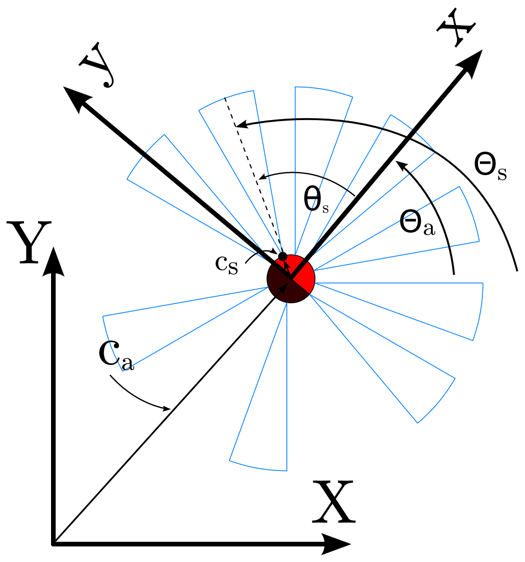


Figure A.6: Translations and rotations from agent's local coordinates to global coordinates.

position in local coordinates ($\mathbf{c_s}$) and the sensor's orientation in the local coordinate system ($\theta_\mathbf{s}$).

Each sensor's position in the agent's local frame of reference does not change. However, its global position and orientation will certainly change as the agent's position and orientation changes. Accordingly, it is required that at each time step the physics engine transform the sensor's position and orientation from the agent's local coordinates to the world's global coordinates. For the purpose of this analysis let all uppercase symbols refer to positions and orientations in the global coordinate system ($\mathbf{C}, \Theta$) and all lowercase symbols to positions and orientations in the local coordinate system ($\mathbf{c}, \theta$), as shown in Figure A.6.

Clearly, two operations are required. The origin of the local coordinate system first needs to be translated to the global origin, and the local coordinates must be rotated to be aligned with the global coordinates. If the agent has global orientation Θ_a, then the corresponding rotation matrix $\mathbf{R}$ to transform all points in the agent's local coordinates to points in a coordinate system with origin at agent centre, but aligned with global coordinates (shown as grey in figure), is

$$\mathbf{R} = \begin{bmatrix} \cos\Theta_a & -\sin\Theta_a \\ \sin\Theta_a & \cos\Theta_a \end{bmatrix} \tag{A.8}$$

Note that the above transformation is a counter-clockwise rotation by Θ_a, because orientations are positive in the counter-clockwise sense in the figure (see Section A.4 for a fuller

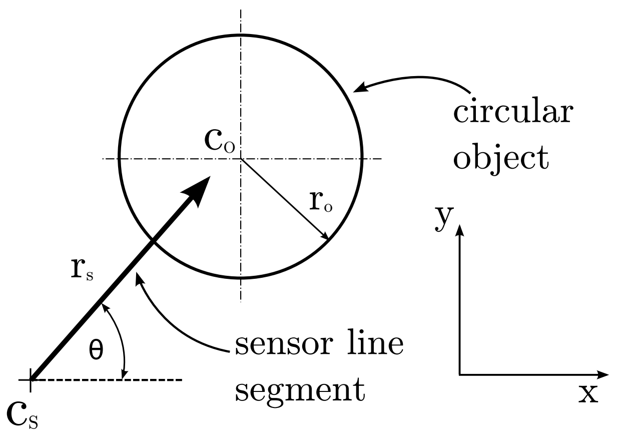


Figure A.7: Schematic of the general case of a sensor line segment intersecting a circular object.

discussion of coordinate systems used). Then, translating all points in this coordinate system to global coordinates yields

$$\mathbf{C_s} = \mathbf{C_a} + \mathbf{R}\mathbf{c_s}$$

Substituting $\mathbf{R}$ from equation A.8,

$$\mathbf{C_s} = \mathbf{C_a} + \begin{bmatrix} \cos\Theta_a & -\sin\Theta_a \\ \sin\Theta_a & \cos\Theta_a \end{bmatrix} \mathbf{c_s} \tag{A.9}$$

Global sensor orientations are obtained trivially,

$$\Theta_s = \Theta_a + \theta_s \tag{A.10}$$

A.3.2 Sensing Circular Objects

The problem of sensing a circular object can be reduced to the problem of detecting the intersection (if any) of a line segment and a circle. The general case of a sensor line segment intersecting a circular object is shown in Figure A.7. We define an x,y-coordinate system as shown, and we further define $\mathbf{C_s}, \mathbf{C_o}$ to be the sensor and object centres, respectively, and r_s, r_o to be the sensor and object radii, respectively. The sensor line segment makes an angle of θ with the positive x-axis. Define $\mathbf{m} = (m_x, m_y) = (\cos\theta, \sin\theta)$ to be a unit vector in the direction of the sensor line segment. All points $\mathbf{X} = (x, y)$ on the circle satisfy the equation

$$\begin{aligned} ||\mathbf{X} - \mathbf{C_o}|| &= r_o \\ \therefore (x - C_{o,x})^2 + (y - C_{o,y})^2 &= r_o^2 \end{aligned} \tag{A.11}$$

Similarly, all points on the sensor line segment satisfy

$$\begin{aligned} \mathbf{X} = \mathbf{C_s} + \lambda \mathbf{m} \qquad & 0 \leq \lambda \leq r_s \\ \therefore x = C_{s,x} + \lambda m_x \quad \text{and} \quad & y = C_{s,y} + \lambda m_y \end{aligned} \tag{A.12}$$

Substituting (A.12) into (A.11), and defining $\mathbf{d} = \mathbf{C_s} - \mathbf{C_o} = (d_x, d_y)$

$$\begin{aligned} (d_x + \lambda m_x)^2 + (d_y + \lambda m_y)^2 &= r_o^2 \\ d_x^2 + 2\lambda d_x m_x + \lambda^2 m_x^2 + d_y^2 + 2\lambda d_y m_y + \lambda^2 m_y^2 - r_o^2 &= 0 \end{aligned}$$

Grouping by λ,

$$\lambda^2 \underbrace{(m_x^2 + m_y^2)}_{\text{A}} + \lambda \underbrace{(2d_x m_x + 2d_y m_y)}_{\text{B}} + \underbrace{d_x^2 + d_y^2 - r_o^2}_{\text{C}} = 0 \tag{A.13}$$

where A, B, C are constants. Constant A must be equal to 1, since $m_x^2 + m_y^2 = ||\mathbf{m}|| \equiv 1$. Equation A.13 is a quadratic function of λ and can be solved by Vieta's formula, viz.

$$\lambda = \frac{-B \pm \sqrt{B^2 - 4C}}{2} \qquad .$$

An assumption made about the sensor is that it will only detect the object nearest to it. Therefore, the desired value of λ will be the smallest root of equation A.13 such that $0 \leq \lambda \leq r_s$, i.e. within sensor range. This will ensure that if there are two points of intersection, the nearest is returned. If there are no acceptable solutions for λ, it means that there are no circular objects within range of that sensor line segment (although other line segments belonging to the same sensor may detect a circular object). The distance from the sensor to the object is then simply the numerical value of λ, since $||\mathbf{m}|| \equiv 1$.

In order to reduce the computational cost for objects that are not near the sensor, a broad proximity condition needs to be met before the above mathematical operations are performed. For circular objects, this condition is

$$||\mathbf{d}|| = ||\mathbf{C_s} - \mathbf{C_o}|| \leq (r_s + r_o) \qquad . \tag{A.14}$$

This condition dictates that the sensor's containing circle (i.e. the circle containing the segment representing the sensor's field of vision) must intersect with the boundary of the circular object.

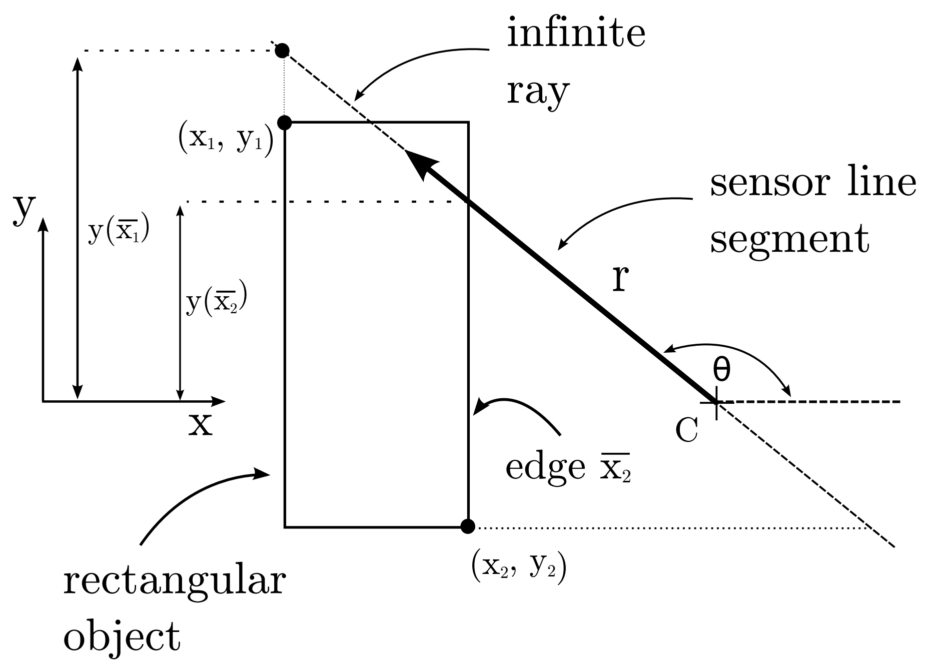


Figure A.8: Schematic of the general case of a sensor line segment intersecting a rectangular object.

A.3.3 Sensing Rectangular Objects

Sensing rectangular objects is slightly more complex than sensing circular objects, due to the former being composed of four separate lines, as opposed to only one. Similar to the method for sensing circular objects, however, the problem is reduced to finding the intersection of a line segment and a rectangle. Let the rectangle's coordinates be x_1, y_1, x_2, y_2 as shown in Figure A.8. For each of x_1, y_1, x_2, y_2 define $\bar{x_1}, \bar{y_1}, \bar{x_2}, \bar{y_2}$ to be the corresponding edge of the rectangle having either constant x or y value, as x_2 is shown in the figure. The sensor's centre has coordinates $\mathbf{C} = (C_x, C_y)$, the sensor has a range of r and its forward direction makes an angle θ with the positive x-axis.

Define the matrix $\mathbf{d} = \begin{bmatrix} d_{\bar{x_1}} & d_{\bar{y_1}} \\ d_{\bar{x_2}} & d_{\bar{y_2}} \end{bmatrix}$ where $d_{\bar{x_i}}, d_{\bar{y_i}}$ are the distances from the sensor centre to $\bar{x_i}$ and $\bar{y_i}$, respectively, along the infinite ray corresponding to the sensor line segment. From plane trigonometry,

$$\mathbf{d} = \begin{bmatrix} \frac{x_1 - C_x}{\cos\theta} & \frac{y_1 - C_y}{\sin\theta} \\ \frac{x_2 - C_x}{\cos\theta} & \frac{y_2 - C_y}{\sin\theta} \end{bmatrix} \tag{A.15}$$

It is now possible to calculate the points of intersection of the infinite ray with each of the edges $\bar{x_i}, \bar{y_i}$. Define

$$\mathbf{P} = \begin{bmatrix} y(\bar{x_1}) & x(\bar{y_1}) \\ y(\bar{x_2}) & x(\bar{y_2}) \end{bmatrix}$$

where $y(\bar{x_i}), x(\bar{y_i})$ are the y and x values at which the sensor infinite ray intersects the infinite rays described by $\bar{x_i}, \bar{y_i}$, respectively. Again, from simple trigonometry it follows that

$$\mathbf{P} = \begin{bmatrix} C_y - d_{\bar{x_1}} \sin\theta & C_x - d_{\bar{y_1}} \cos\theta \\ C_y - d_{\bar{x_2}} \sin\theta & C_x - d_{\bar{y_2}} \cos\theta \end{bmatrix} \tag{A.16}$$

Note that, in practice, each element of $\mathbf{P}$ should only be calculated if the corresponding distance in $\mathbf{d}$ is positive and less than the sensor range, r, in order to save computational expense. That is to say, calculate $P_{i,j}$ iff $0 \leq d_{i,j} \leq r$. Once the appropriate elements of $\mathbf{P}$ have been computed, a check is performed to see whether the corresponding points lie within the rectangle. Formally, the sensor line segment intersects edge $\bar{x_i}$ iff $y_2 \leq y(\bar{x_i}) \leq y_1$ and similarly for edges $\bar{y_i}$. Once the edges which intersect the line segment have been determined, the lowest corresponding distance value from $\mathbf{d}$ may be regarded as the minimum distance from the sensor to the rectangle, provided the value falls in the range $0 \leq \min(\mathbf{d}) \leq r$.

Similar to the case with circular objects, a proximity condition must be met prior to performing the above computations, in an attempt to reduce computational expense. The proximity condition is

$$x_1 - r \leq c_x \leq x_2 + r \qquad\qquad y_2 - r \leq c_y \leq y_1 + r \tag{A.17}$$

This condition can be visualized as a frame of thickness r placed around the object, as shown in Figure A.9 on the following page. If the sensor centre lies within the frame or the object (shaded region in the figure), then the proximity condition is met.

A.4 A Note on Coordinate Systems

The standard two-dimensional Cartesian axes form a right-handed system. That is to say, the x-axis lies clockwise of the y-axis. However, in Tkinter (Python's GUI), the axes have a left-handed orientation, as shown in Figure A.10 on page 140. It was decided that, since right-handed systems would likely be more familiar to users of the simulator, the agents' local coordinate systems should be right-handed. The rationale was that defining new sensor configurations on agents would need to take place within the agents' local

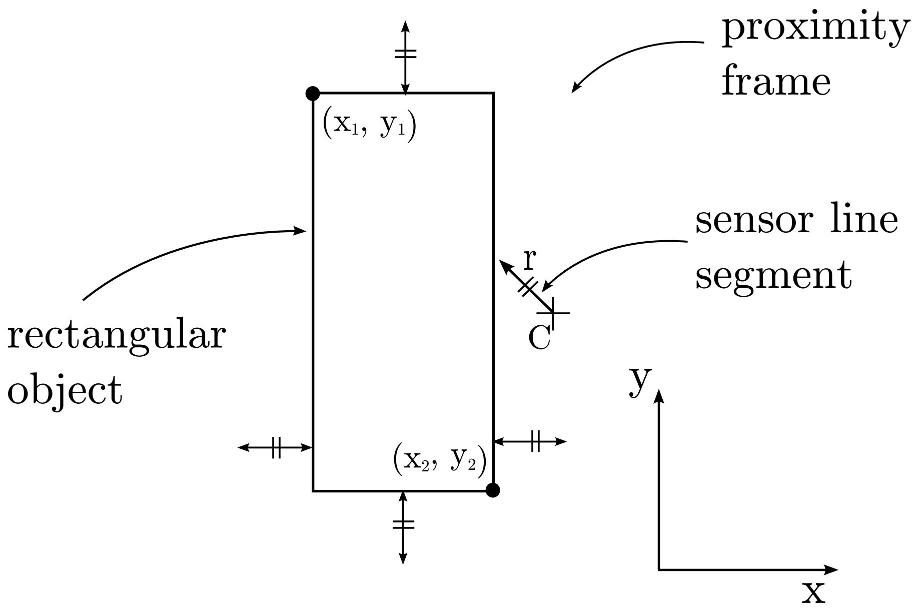


Figure A.9: Proximity condition for a sensor line segment intersecting a rectangular object.

coordinate systems, and so these coordinate systems should be as intuitive for the user as possible. For the sake of consistency, and also to separate the *theory* of the simulator from its particular *implementation*, the foregoing physics calculations were presented for a right-handed global coordinate system.

Consequently, the global coordinate system remains left-handed, but agents' local coordinates are right-handed. This means that if creating custom maps by hand, as opposed to using the map importer (see Section 4.9 on page 53), the ordinate axis increases in the *downward* direction; the abscissa is unaffected. However, if defining new sensors on an agent, the usual coordinate conventions apply (and are illustrated in Figure A.10). Note that these conventions only apply for the `AgentCir` superclass that has been provided with the simulator; if creating custom agent classes the default left-handed orientation will apply.

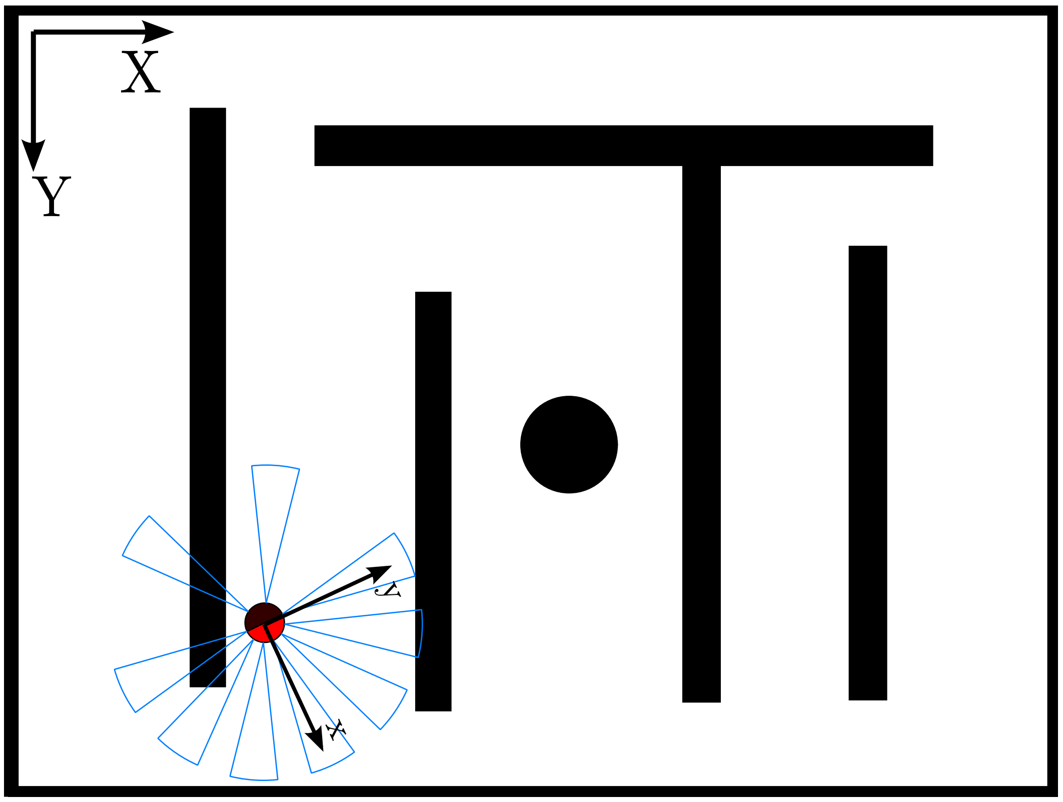


Figure A.10: Global and local coordinate systems in the simulator.

Appendix B

Modelling of Fire-fighting Environment

Since the details of the fire-fighting task assume a number of simplifications about the nature of the task, it is not possible to use dimensional analysis to infer appropriate 'ball park' figures for constants in the simulation, such as specific heat capacities, rates of heat transfer, and so forth. Therefore, in order to ensure that the simulation progresses in a fashion that allows meaningful analysis of the swarm-based behaviours it is designed to elicit (such as implicit cooperation, emergent swarming, individual and group self-preservation, etc), it is necessary to develop a mathematical model of the system so that various parameters can be adjusted to ensure this outcome is met. It should be stressed that this model is *not* intended to be physically accurate, nor is it intended to simulate actual fire-fighting behaviours. Being a swarm behaviour simulator, the fundamental purpose of this fire-fighting task is to demonstrate of how swarm behaviours emerge in a distributed resource allocation problem, such as the assignment of fire-fighting agents to fires.

This appendix will present the mathematical modelling that was performed to define equations representing the physical system and determine suitable values for constants in those equations. Note that the T used in this appendix refers to fire temperature and is unrelated to the T used to indicate an agent's propensity to explore the state-action space, as it was used in Chapter 8.

We begin by modelling the rate at which the fire's temperature grows when not being extinguished. In the qualitative requirements for the model, given in Chapter 9, it is required that the rate of change of a fire's temperature increases as time progresses (thus making the fire harder to extinguish). Additionally, it is required that a fire reach a

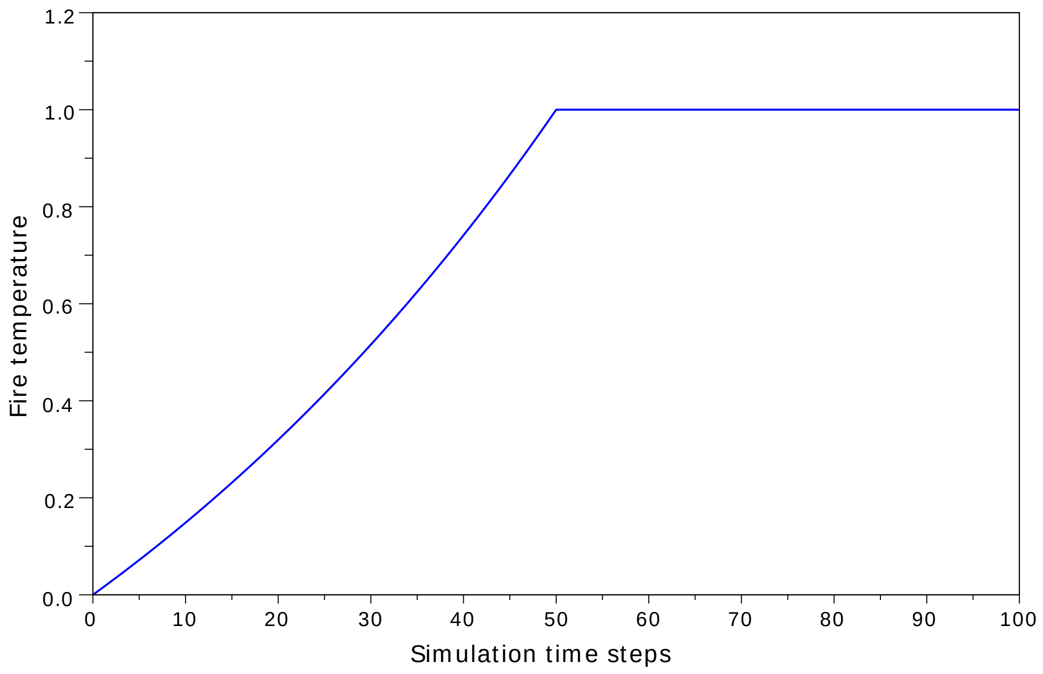


Figure B.1: Fire temperature as a function of simulation time steps, when not being extinguished.

terminal temperature. One such function that meets these requirements is

$$T = \min(e^{k_1 t} - 1, 1) \qquad . \tag{B.1}$$

A graph of this function, for an arbitrary value of k_1 is shown in Figure B.1. Differentiating equation B.1 yields

$$\frac{dT}{dt} = k_1 e^{k_1 t} = k_1(T+1) \qquad .$$

We are now able to add in the effect on a fire's temperature of agents extinguishing that fire. The qualitative requirements dictate that fires of larger area should be 'harder' to extinguish, so we define

$$\frac{dT}{dt} = k_1(T+1) - \frac{k_2}{A}n \tag{B.2}$$

where k_2 is a second constant, and n is the number of agents extinguishing that particular fire. Equation B.2 may now be turned into an update rule which is performed at each time step of the simulation,

$$T_{t+1} = \min(T_t + k_1(T_t + 1) - \frac{k_2}{A}n, 1) \qquad . \tag{B.3}$$

We now consider the effect on an agent of being near a fire. Denote an agent's temperature by H. When near a fire, agents absorb heat at a rate proportional to the temperature and size of the fire. Additionally, it is assumed that agents expel heat at a constant rate, until their temperature reaches 0. This can be modelled mathematically as

$$\frac{dH}{dt} = k_3TA - k_4 \qquad . \tag{B.4}$$

The corresponding update rule, ensuring that the minimum value of 0 is enforced, is given by

$$H_{t+1} = \max(H_t + k_3TA - k_4, 0) \qquad . \tag{B.5}$$

Now that the system has been modelled to give the desired qualitative characteristics, what remains is to determine the constants k_{1-4} that will ensure the simulation is feasible. This entails considering a number of possible problems such as fires growing inexorably large in a short space of time, agents' abilities to extinguish fires being too small to extinguish fires, or so large that fires are trivially extinguished by a single agent, and so forth. Since the mathematical model used here is somewhat abstracted from reality, it is not possible to use dimensional analysis to convert real-world constants into the model constants k_{1-4}. Therefore, the above equations need to be reworked to make the constants calculable from certain desirable system properties, such as the time taken for a fire to reach maximum temperature, time taken by a certain group of agents to extinguish a particular fire, and so forth. Once suitable values for these constants have been determined, they can be substituted into the model, and the simulation should provide the desired characteristics.

Constant k_1 was chosen by considering a fire that is not being extinguished, and solving equation B.1 for the time taken for a fire to reach the maximum temperature of 1, starting from approximately 0, thus

$$k_1t = \ln(T + 1)$$

$$k_1 = \frac{\ln 2}{t} \tag{B.6}$$

Note that a fire cannot start from a temperature of exactly 0, since its rate of growth is proportional to its temperature. Constant k_2 is the rate which each agent extinguishes a fire, and therefore will determine the time taken by a certain number of agents to extinguish a particular fire, starting at a particular temperature T_0. Equation B.2 is a linear first-order differential equation, and has a solution of the form

$$T = \alpha e^{k_1 t} - \frac{k_1 - \frac{k2}{A}n}{k_1} \tag{B.7}$$

for a suitable constant α. If we assume that at time $t = 0$ the temperature of the fire is $T = T_0$, then substituting into (B.7), we obtain

$$\alpha = T_0 + 1 - \frac{k_2 n}{k_1 A} \qquad . \tag{B.8}$$

We are interested in the time taken to extinguish the fire, so we denote this time as t_1, where $T(t_1) = T_1 = 0$. Substituting (B.8) into (B.7) and rearranging,

$$(1 - \frac{k_2 n}{k_1 A})e^{k_1 t_1} + T_0 e^{k_1 t_1} = (1 - \frac{k_2 n}{k_1 A})$$

$$\therefore (1 - \frac{k_2 n}{k_1 A})(e^{k_1 t_1} - 1) = -T_0 e^{k_1 t_1}$$

$$\therefore \frac{k_2 n}{k_1 A}(e^{k_1 t_1} - 1) = T_0 e^{k_1 t_1} + e^{k_1 t_1} - 1 \qquad .$$

Finally, solving for k_2,

$$k_2 = \frac{\frac{k_1 A}{n} T_0 e^{k_1 t_1} + e^{k_1 t_1} - 1}{e^{k_1 t_1} - 1} \tag{B.9}$$

Constant k_2 can therefore be determined, by means of equation B.9, from the time (t_1) it should take n agents to extinguish a fire of area A and initial temperature T_0.

Constants k_3, k_4 control the rates at which agents absorb heat (when near a fire) and expel heat (at a constant rate, regardless of proximity to a fire). From the form of equation B.4 it is clear that there must be a critical value of TA at which the rate of agent heat absorption equals the rate of heat expulsion. We base our analysis on the largest fire on the map, with area A_{max}, and denote the critical temperature T_c. At this temperature, $\frac{dH}{dt} = 0$. Substituting this into equation B.4, we obtain

$$k_4 = T_c A_{max} k_3 \qquad . \tag{B.10}$$

We will require a second equation in terms of k_3, k_4 in order to obtain numerical values for these constants. It is possible to place a constraint on these constants by limiting the maximum magnitude of $\frac{dH}{dt}$ to a constant value, such as the number 1. From equation

B.4,

$$\max \frac{dH}{dt} = T_{max} A_{max} k_3 - k_4$$

$$\therefore 1 = A_{max} k_3 - k_4 \qquad .$$

Substituting (B.10),

$$1 = A_{max}(1 - T_c) k_3$$

$$\therefore k_3 = \frac{1}{A_{max}(1 - T_c)} \qquad . \tag{B.11}$$

Now that expressions have been found for physical constants k_{1-4}, it remains to determine expressions for the H values at which agents choose to leave and rejoin fires, denoted by H_{leave} and H_{join}, respectively. The maximum value of $\frac{dH}{dt}$ has been set to 1, which corresponds to the largest fire on the map, at its maximum temperature of 1. If we consider an agent attempting to extinguish this fire starting from $H = 0$, and define t_m to be the maximum time the agent will stay at that fire before leaving, then

$$H_{leave} = \int_0^{t_m} H'(t)\,\mathrm{d}t \qquad .$$

For small values of t, $H'(t) \approx H'(0) = 1$. Thus,

$$H_{leave} \approx \int_0^{t_m} 1\,\mathrm{d}t$$

$$\therefore H_{leave} \approx t_m \qquad . \tag{B.12}$$

Note that equation B.12 has assumed that the agent's H value upon reaching the fire was 0: if it were any higher (any value up to H_{join} is possible), the agent would remain at the fire for less time and equation B.12 therefore provides an upper bound.

When an agent's H value reaches H_{leave}, the agent leaves the fire and will only return to a fire once its H value has decreased to H_{join}. This means constant H_{join} may be determined by the amount of time an agent should spend away from a fire, denoted by t_a. From equation B.4, an agent's H value decreases at a rate of k_4 when not near a fire, to a minimum value of 0 (by definition). Therefore,

$$H_{join} = \max(H_{leave} - k_4 t_a, 0) \qquad . \tag{B.13}$$

Appendix C

User Instructions for Simulator

This appendix presents a minimal set of instructions for use of the simulator developed in this project, named RAS2D (**R**obotic **A**gent **S**imulator in **2 D**imensions).

RAS2D was developed on i386 Linux using Python 3. It is almost certain to work with Python 3 on any Unix® -like system, and will most likely work without modification on other platforms (e.g. Windows, iOS) and architectures (e.g. AMD64). It will not work "out of the box" on Python 2, due to a number of Tkinter (Python's implementation of the Tk GUI) functions being renamed between Python 2 and 3, including the name of the module itself.

As was mentioned in the report, the package has three executable files. Simulation is accomplished by means of input files with extension `.job`, which contain all specifications for a particular simulation. The three basic modes of operation are given below.

Core simulation.
The basic procedure for operation of the simulator is as follows. The user creates a `.job` file in a text editor, normally by modifying the sample `.job` files provided with the simulator package. The job file specifies the map, type and number of agents, as well as the control functions used by agents. It can also include other task-specific scripts, e.g. for Q-learning or fire-fighting tasks.

Once an appropriate job file has been created, the user runs the main simulation program, `sim.py`. This program takes a job file as input, and either displays the simulation on-screen in real-time, saves it to disk, or both. If the simulation is saved to disk, it is written to a `.sim` file.

Visualisation. To visualise a simulation that has previously been saved, run the `viewer.py` script. When prompted, select the appropriate `.sim` file and visualisation will begin. To adjust playback speed, speed regulation can be turned on and off by setting variable `speedReg` (within `viewer.py`) either to `True` or `False`, respectively.

Map creation. The executable `mapImporter.py` may be used to convert SVG images into the native `.map` format used by the simulator. At present, only circular and rectangular elements are supported. To import a map, run the script and when prompted, select the appropriate input `.svg` file and output `.map` file.

The scripts comprising the RAS2D package are thoroughly commented, allowing easy modification of their operation. Nevertheless, a set of instructions for performing custom modifications on each script can be found in the directory `instructions`, located in the main package directory. The instructions for a file with name `filename.py` may be found in `instructions/filename.py.instr`.

Appendix D

Examples Included on Disc

D.1 Job and Simulation Files

Simulation `.job` and `.sim` files are provided on disc for the simulations referred to in the report. The files included are:
`collisionAvoidance`
`agglomeration`
`chainFormation`
`QlearningSingleAgent`
`QlearningMultiAgent`
`fireFighting1`
`fireFighting2`.

Note that running a particular job file might not yield exactly the same behaviour as the corresponding visualisation file that has been provided. This is because simulations involving randomness can have different behaviour when run on different machines, due to slight differences in the pseudorandom number generators.

D.2 Maps

A number of sample maps are included with the RAS2D package. These maps are contained in files `original.map`, `maze.map`, `openspace.map` and `openspace_FIRE.map`, and are shown here for the reader's convenience. Also provided on disc are the SVG graphics files that were used to create the maps using the map importer (see Section 4.9 on page 53). These files have the same base names, but with extension `.svg`.

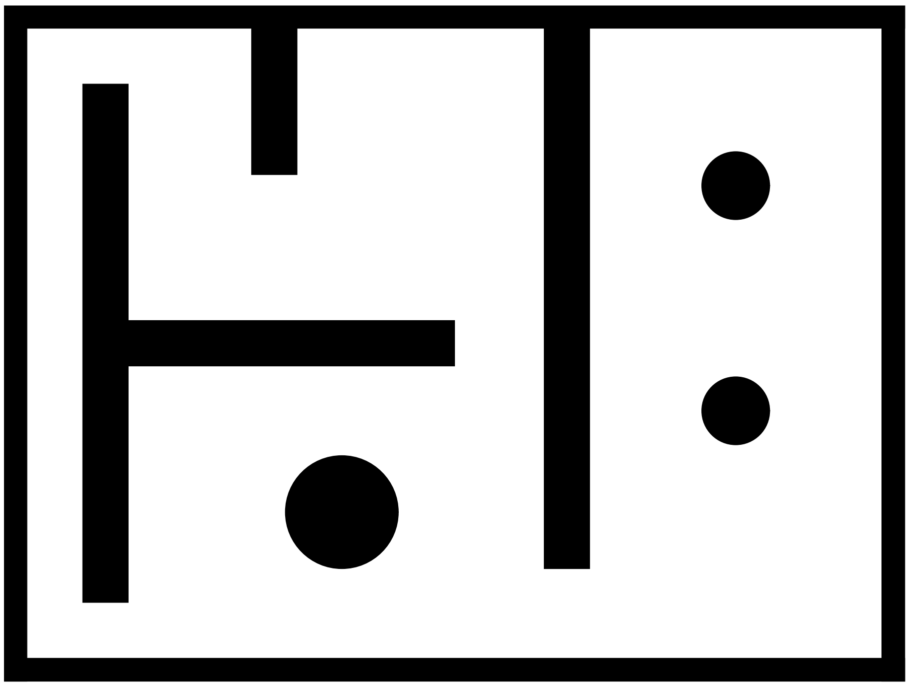


Figure D.1: A map with circular and rectangular obstacles, showing a moderate degree of complexity. Filename: `original.map`.

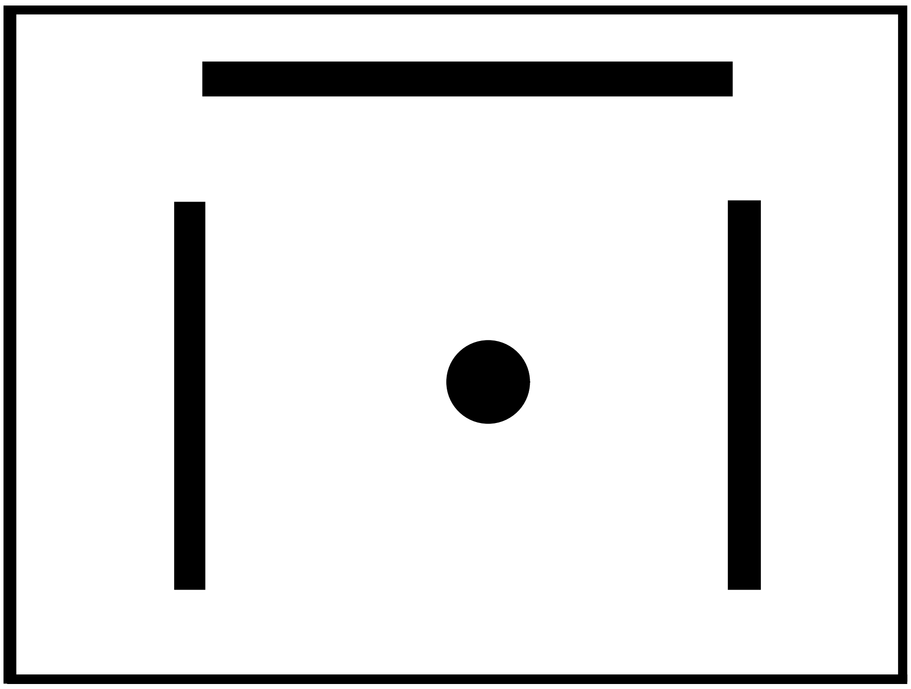


Figure D.2: A map with circular and rectangular obstacles, showing a low degree of complexity. Filename: `sparse.map`.

Figure D.3: A map with four walls and no internal obstacles. Filename: `openspace.map`.

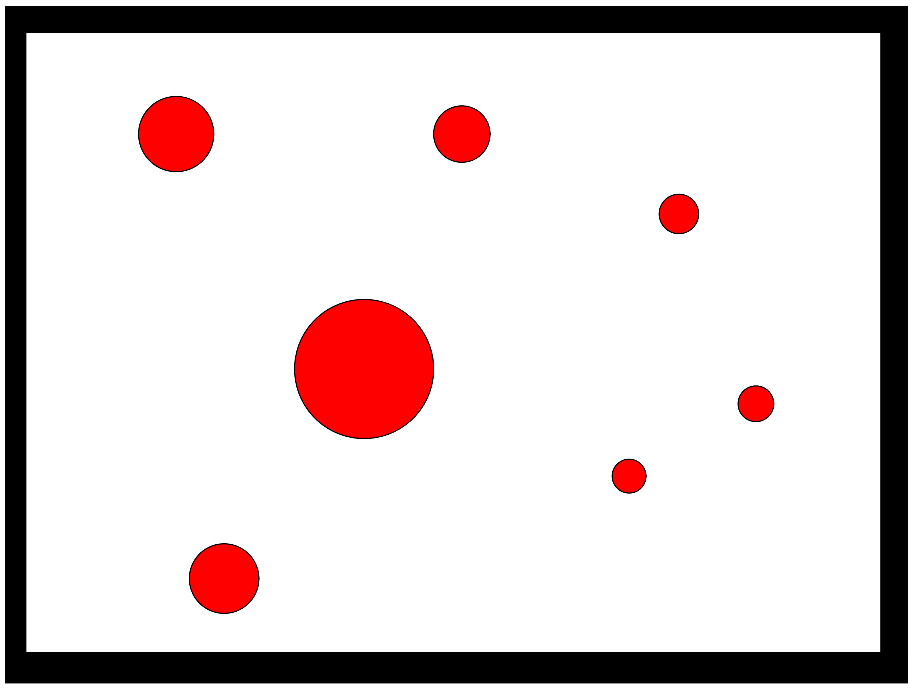


Figure D.4: A map with four walls and a number of internal fires. Filename: `openspace_FIRE.map`.

Appendix E

Learning Outcomes

This appendix aims to give a brief review of the knowledge that was gained by the student during the course of this project.

Broadly speaking, much of the content and theory in the project had not been encountered by the student before undertaking the project. Topics in artificial intelligence are not covered by the mechanical engineering curriculum and the student was thus unexposed to these areas of knowledge. The student therefore had to first acquaint himself with the appropriate concepts, and then perform in-depth research to reach a point where programmatic implementation of AI techniques was possible.

Despite this, much of the physics, dynamics and mathematics required for the physical simulator had been taught to the student, making these sections of the project comparatively easier. While the student had not previously been exposed to formal algorithm analysis and had to undertake research into topics such as algorithmic complexity, previous courses in numerical analysis proved helpful in troubleshooting irregular behaviour.

In terms of the programmatic work, the student had only been formally exposed to Matlab as a programming language. However, Matlab was not deemed to be a suitable choice of language for the simulator, requiring the student to learn a completely new programming language, namely Python. Although this was done on a self-taught basis, no significant problems were experienced and the student was able to reach a level of proficiency sufficient to create the entire simulator package in Python.

Appendix F

Exit Level Outcome Forms