Author:
Jones, Simon W

Title:
Onboard Evolution of Human-Understandable Behaviour Trees for Robot Swarms

Onboard Evolution of Human-Understandable Behaviour Trees for Robot Swarms

Simon William Jones

A dissertation submitted to the University of Bristol in accordance with the requirements for award of the degree of Doctor of Philosophy in the Faculty of Engineering.

March 16, 2020

Word count : 58497

Abstract

Swarm robotics, inspired by swarms in nature, has great potential. The resilience, scalability, robustness and redundancy of having many robots collectively perform tasks such as mapping, disaster recovery, pollution control, and cleaning make for a compelling vision. To achieve this, we need to design swarm robot systems to have a desired collective behaviour, but the design of controllers for the individual robots of a swarm such that this behaviour emerges from the interaction of the individual robots is difficult. Current solutions often use off-line automatic discovery by artificial evolution of robot controllers, which are then transferred into the swarm. This is problematic for two important reasons. Firstly, since there is the need for additional supporting infrastructure, both to evolve the new controllers and to communicate them to the swarm, the swarm is not self-sufficient. Secondly, the evolved controllers are often opaque and hard to understand, an important consideration for safety and explainability reasons.

In this work we tackle both of these issues. We build a swarm of robots with very high computing performance using recently available mobile computation devices. This high performance allows us to move the evolutionary process, dependent on processing power for simulation, into the swarm. Because the computational power of the swarm grows with the size of the swarm, it is both autonomous and scalable. We use behaviour trees as the individual robot controller architecture. They are modular, hierarchical and human readable. By developing automatic tools to simplify large evolved trees, we can understand, explain, and even improve the evolved controllers.

By moving the evolutionary process into the swarm, and by using understandable controllers, we make the swarm autonomous, scalable, and understandable, necessary steps towards their real-world deployment.

Author's Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: .... DATE:..........................

Dedication

For Lucy

"I never am really satisfied that I understand anything; because, understand it well as I may, my comprehension can only be an infinitesimal fraction of all I want to understand about the many connections and relations which occur to me."

Ada Lovelace

Acknowledgments

I would like to think my supervisors Dr. Matthew Studley, Dr. Sabine Hauert, and Prof. Alan Winfield for their support and assistance throughout my journey from being an engineer towards being a scientist.

I would also like to thank my wonderful wife Lucy, without whom this could never have happened. Her inspiration, example, belief, boundless positivity and love have changed my life for the better.

Finally, I'd like to thank my dad, who inspired me to become an engineer and started me on the path to this point.

Contents

Acronyms

ADF	Automatically Defined Function
AI	Artificial Intelligence
API	Application Programming Interface
BLAS	Basic Linear Algebra Subprograms
BNF	Backus-Naur Form
BT	Behaviour tree
CD	Compact Disc
CHDS	Controlled Hybrid Dynamical Systems
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
DLP	Digital Light Processor
DMA	Direct Memory Access
DNN	Deep Neural Network
EA	Evolutionary Algorithm
ES	Evolution Strategies
FLOPS	Floating Point Operations per Second
FOV	field of view
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GA	Genetic Algorithm
GP	Genetic Programming
GPGPU	General Purpose computing on Graphics Processing Units
GPIO	General Purpose IO
GPU	Graphics Processing Unit
HMP	Heterogeneous MultiProcessing
HPC	High Performance Computing
I^2C	Inter-Integrated Circuit
IMU	Inertial Measurement Unit
IR	Infra-red
ISA	Instruction Set Architecture

JTAG	Joint Test Action Group
LEB	Linux Extension Board
LED	Light Emitting Diode
MIPS	Million Instructions Per Second
NEAT	NeuroEvolution of Augmenting Topologies
NN	Neural Network
NPC	Non-Player Character
PCB	Printed Circuit Board
PFSM	Probabilistic Finite State Machine
RAID	Redundant Array of Inexpensive Disks
RAM	Random Access Memory
RASP	Random Access Stored Program
ROS	Robot Operating System
RTOS	Real-time Operating System
SBC	Single Board Computer
SGEMM	Single precision floating point General Matrix Multiply
SoC	System-on-Chip
SOM	Self Organising Map
SPI	Serial Peripheral Interface
USB	Universal Serial Bus
UVC	USB Video Class
VGA	Video Graphics Array

List of Figures

List of Tables

List of Equations

Chapter 1

Introduction and motivation

1.1 Overview

Swarm robotics [Şahin, 2005] takes inspiration from collective phenomena in nature, where global behaviours emerge from the local interactions of the agents of the swarm with each other and with the environment. Such natural phenomena as flocking birds, schooling fish, ant trail formation, bee foraging and decision-making all exhibit properties that are desirable from a robotics perspective.

Swarms have redundancy and resilience, no single agent is necessary for the swarm to function. They are distributed and decentralised, there is no central controlling agent, the swarm self-organises, and there is no single point of failure. They are scalable, since they depend on local interactions rather than global, adding more agents is not exponentially costly. Swarm robotics aims to construct artificial swarms that exhibit these desirable properties in order to both solve problems and to better understand how natural swarms work.

Classic problems that are seen as amenable to solution using swarm robotics are mapping, search-and-rescue, emergency communications for disaster recovery, pollution monitoring and control. In all these cases, being able to deploy many relatively cheap robots that self-organise, and construct a communications network, or monitor an area, or locate pollutants may be better than using a few more capable but much more expensive robots.

1.2 Motivation

The central problem of swarm robotics is the design of individual robot controllers that, when instantiated in a swarm of real robots, results in the swarm as a whole behaving in the way we want. In other words, the emergence of a desired collective behaviour from the individual interactions of the robots with each other and the environment. This process is hard, because it is not obvious how individual local interactions in complex systems will combine, a fundamental question in many fields.

One of the common ways in which swarm robotics approaches the problem of designing controllers is to use artificial evolution. Potential controller solutions are instantiated in simulated swarms of robots and the swarm is evaluated against some criteria, often an objective or fitness function. Controllers that perform well are preferentially kept for combination or alteration to produce new controllers. This evolutionary process results in better controllers being discovered over time. Like other bio-inspired approaches, artificial evolution is attractive because of the potential for novel solutions outside the thoughts of the designer. The cost of this approach is the need to run many simulations, a computationally expensive process. In most cases, this means that the evolutionary algorithms are run off-line, on powerful computers, before the controllers are transferred into robots. This off-line process cannot be made adaptive and relies on outside resources. It would be possible to use external resources within an adaptive process by having communications links between the external resources and the swarm, but this is unreliable and difficult to scale, since the required bandwidth increases with swarm size.

We have referred several times to scalability. In the context of swarm robotics we mean by this that there should be no inherent limit to the number of robots within the swarm. This implies that the interactions between the robots, and between the robots and the environment, should be local. A swarm reliant on a central resource, for example for computation, would not be very scalable since it would require non-local communication, a bandwidth limited resource. Distributed systems relying only on local communication and computation add aggregate communication bandwidth and performance by adding more robots and occupying more physical space. Scaling may not be perfect, imposing some actual limit on swarm size, but this limitation is far less severe than those imposed by centralised architectures.

The type of controller used in this evolutionary process is important. There are many possible controller architectures, neural networks being very widely used for the reason that they provide solutions to non-linear problems and fit well within the evolutionary algorithm paradigm. One major drawback, however, is the opacity of the resulting controllers. It is not generally possible to say what a neural network controller will do beyond very simple examples without actually running it. Behaviour trees (BTs), on the other hand, are human readable, and can be analysed, explained and understood. There are two important reasons why we might want to be able to understand and analyse a controller, firstly for safety. In a real-world situation, we need to be able to perform a safety analysis and say what a robot might do in any given situation. Secondly, for understanding and insight. A swarm running an evolved controller may perform well, being able to analyse that controller can teach us new things, and possibly assist in closing the gap in our understanding of the connection between the individual and emergent collective behaviours. For these reasons, we argue that behaviour trees are a good controller architecture for use in robot swarms.

We tackle these two issues in this work. Firstly by giving the swarm sufficient processing power to be able to run evolutionary algorithms. The exponential progress in computing means that low-cost single-board computers are now available to build such a swarm. We enhance the widely used e-puck robots to create a swarm with a collective processing power of two Teraflops. We write a fast simulator capable of running on the robots and use this to build a distributed evolutionary algorithm running on the swarm in a scalable way; as the swarm size increases, so does the processing power and the evolutionary algorithm performance. Secondly, we use behaviour trees as our controller architecture. Originating in the computer games industry to control non-player characters, they are modular, hierarchical and extendible, and human readable. The hierarchical nature means they can be decomposed into subtrees for understanding. They are a natural fit for the evolutionary techniques of Genetic Programming (GP), and amenable to simplification by automatic tools. We demonstrate the practicality of evolving behaviour trees for swarm robotics, then go on to design a behaviour tree architecture for our Teraflop swarm. We show the swarm is capable of evolving fit solutions entirely in-swarm within a period of only 15 minutes, and that we can analyse, understand, and engineer improvements to these evolved behaviour tree controllers.

1.3 Hypotheses

We make two hypotheses in this work:

Hypothesis 1 It is possible to automatically generate understandable Behaviour Trees to control a swarm of robots in the performance of a collective task.

Hypothesis 2 It is possible to perform automatic generation of robot swarm controllers entirely within the swarm.

These hypotheses motivate various research questions. To demonstrate Hypothesis 1, we need to answer: How do we design a behaviour tree to control a robot? How do we represent a behaviour tree for the purpose of automatic generation? What methods of automatic generation should we use? How can we analyse and understand the resultant trees? These questions are considered in Chapters 3, 4, 6, and 8. To demonstrate Hypothesis 2, we need to answer: What computational performance is required for automatic generation of controllers? What computational platforms can we use? How do we build such a robot swarm? How do we best use the available computational capabilities? How can we make performance scalable with swarm size? How do we manage the *reality gap*? These questions are considered in Chapters 5, 6, 7, and 8.

1.4 Structure

This thesis is organised in the following way. Chapter 2 discusses the background and related work in the various areas that we cover, situating this work.

Chapter 3 looks at behaviour trees, their origins, theory and semantics, ways they can be manipulated, and their relationship with other controller architectures. We then look at ways of applying evolutionary algorithms to behaviour trees, demonstrating the practicality of this approach in Chapter 4, using a swarm of Kilobot robots.

In Chapter 5 we cover the design of the Xpuck, the robot that forms the Teraflop swarm. We briefly survey progress in cheap computational power that has resulted in Graphics Processing Units (GPUs) supporting General Purpose computing on Graphics Processing Units (GPGPU) becoming available with low enough power consumption to make possible their application in swarm robotics. We detail the physical and electronic design, and operating point tuning. The design of a fast physics simulator is described and we demonstrate the performance of the swarm with an in-swarm evolutionary algorithm and a image processing task.

Having shown that evolving behaviour trees as swarm controllers is viable, and explored the design of the Xpuck, in Chapter 6 we take a more formal approach to the design of a behaviour tree architecture suitable for use in our Teraflop swarm, balancing expressiveness and conciseness. This is informed by awareness of how too much expressiveness in the controller can lead to *reality gap* problems.

Chapter 7 covers the issue of ensuring that controllers evolved in simulation can successfully be transferred to real robots, the *reality gap* issue. We define a benchmark collective movement task that is used as the target for our experiments. The behaviour of the real robots is measured and used to calibrate and align the simulator. Where simulator fidelity is low, such as collisions, we mitigate the problem at the behavioural level, and we apply masking noise to the simulated robots. These mitigating measures are tested by evolving a controller in simulation to perform the benchmark task, then measuring the performance of the swarm of real robot. This shows that there is no significant difference in performance between the simulated and the real swarm.

In Chapter 8 we move the evolutionary algorithm into the swarm. We introduce a modified evolutionary algorithm that takes into account our noisy fitness function and makes better use of the simulation budget. We extend this to become a distributed island model evolutionary algorithm, running across the whole swarm in a scalable manner. The particular behaviour tree architecture that we use for the experiments is detailed, as is the experimental protocol. The experimental runs show the swarm is able to evolve fit controllers within 15 minutes. We select data to analyse by using a Self Organising Map (SOM) to cluster controllers of similar behavioural characteristics, then using automatic simplification methods and further human manipulations to describe in detail the functioning of several different evolved controller trees. The effect of heterogeneity on the performance of the swarms in simulation and real-life is investigated. Finally, using the understanding gained, we engineer a performance improvement in one behaviour tree.

In Chapter 9 we summarise the work of the previous chapters, looking at the achievements. We outline possible future work, particularly in the direction of making the swarm adaptive to changes in the environment. Finally, we talk about some issues, lessons learned, and recommendations for experimental swarm robotics.

1.5 Contributions to swarm robotics

Although this thesis is written in the first person plural, I am the primary originator of the work contained herein. I designed, built, tuned and programmed the Xpuck robots. I developed the use of behaviour trees as a controller architecture for evolutionary swarm robotics. I designed the system infrastructure and all the programming necessary to run experiments. I developed the fast GPU simulator, behaviour tree interpreter and distributed evolutionary algorithms. I designed and performed all of the experiments, analysed all of the data, and wrote this thesis.

This work makes the following contributions to the field of swarm robotics:

- A clear description of the semantics of behaviour trees, resolving ambiguities and describing a complete algorithm for evaluation
- A set of rules that allow the automatic manipulation and simplification of behaviour trees for further analysis
- Demonstrated the possibility of automatically evolving behaviour trees for a swarm robotics task
- Designed and built a robot swarm with a collective processing power of two teraflops
- Developed a fast simulator capable of running entirely hosted within the swarm
- Developed a distributed evolutionary algorithm, using the fast simulator, and running entirely hosted within the swarm, that shows performance scalability with the size of the swarm

- Demonstrated via a combination of approaches a methodology for minimising the *reality gap* when transferring automatically generated controllers to real robots
- Developed an improved evolutionary algorithm that makes better use of the simulation budget for our noisy fitness function
- Developed an island model parallelisation of the evolutionary algorithm, showing good scaling of performance
- Demonstrated in-swarm evolution of effective swarm robot controllers within 15 minutes
- Developed and used tools to analyse, explain, and improve automatically generated controllers

Chapter 4 contains work that was published as *Evolving behaviour trees for swarm robotics* in Proceedings of DARS 2016 - International Symposium on Distributed Autonomous Robotic Systems [Jones *et al.* , 2016].

Chapter 5 contains work that was published as *A Two Teraflop Swarm* in Frontiers of Robotics and AI [Jones *et al.* , 2018].

The design of the simulator described in Chapter 5 was informed by work published as *Mobile GPGPU Acceleration of Embodied Robot Simulation* in Proceedings of Artificial Life and Intelligent Agents: First International Symposium (ALIA 2014) [Jones *et al.* , 2015].

Chapter 8 contains work that was published as *Onboard Evolution of Understandable Swarm Behaviors* in Advanced Intelligent Systems [Jones *et al.* , 2019].

We have made the hardware design files for the Xpuck robot, and the OpenCL accelerated simulation software available under a permissive open source license, see Appendix A.1.

Chapter 2

Background and related work

In this chapter, we give a broad overview of the state of the art. More in-depth coverage of relevant aspects of the state of the art can be found in later chapters.

2.1 Swarm robotics

Swarm robotics [Şahin, 2005] takes inspiration from collective phenomena in nature, where global behaviours emerge from the local interactions of the agents of the swarm with each other, and with the environment. Swarms have many desirable properties that make them interesting from a robotics perspective. They are decentralised, resilient, and robust, relying on no central controller or authority, and no individual agent is necessary for the fulfilment of a task. Because interactions are local, swarms are scalable without exponential increases in complexity. They have the potential to be cheaper than conventional approaches, since each agent can be mass-produced.

In general, from Şahin [2005] again, to be regarded as a swarm robotics system, there should be a relatively large number of real, physically embodied robots (perhaps 10 to 20, at least more than 5), each robot should be autonomous and capable of sensing and interacting with the world, and the robots should generally be homogeneous, both in their capabilities and controllers. The individual capability of the robots will not be high compared to the task, it is the collective capability of the swarm that performs the task. Perhaps most importantly in judging what makes a system a *swarm* system is that the robots are only capable of local sensing and communication. Any reliance on global capabilities are likely to severely restrict the scalability of a swarm.

None of these qualifications are absolute, for example Dorigo *et al.* [2013] breaks with homogeneneity, explicitly exploring heterogeneous swarms comprising three different robot types to fill different roles, capable of moving on flat surfaces, climbing, for flying and attaching themselves to the ceiling. There is however a strong assumption in the literature, eg Brambilla *et al.* [2013]; Trianni *et al.* [2014], that limited individual robot capability means limited computational capability. This seems to stem from the success of Brooks' [1991] reactive architectures, views of social insects as extremely simple individual agents [Deneubourg & Goss, 1989], and the fact that complex behaviours can emerge from the interaction of very simple rules [Reynolds, 1987]. But even the simplest of insects have many thousands of neurons; the parasitic wasp *Megaphragma mymaripenne* has 7400, an ant has 2.5×10^5 , and a honey bee has a million [Menzel & Giurfa, 2001; Polilov, 2012]. We argue that limiting the computational abilities of each individual agent of the swarm is an unnecessary restriction, even the Xpuck robots in this work with their on-board GPU are only computationally equivalent to a few thousand biologically plausible neurons, Minkovich *et al.* [2014], orders of magnitude less processing power than possessed by honey bees.

Emergence A fundamental concept of swarm robotics is that of *emergence*, individual agents interact with each other and with the environment according to sets of rules they each follow, and from these local interactions a global swarm behaviour emerges. The global behaviour is different from, and is not obvious from, the local rules of behaviour [Bedau, 1997]. A related concept is *self-organisation*, whereby a system maintains some form of structure, spatially, or temporally, without external control. Emergence and self-organisation can exist individually in a system, but for interesting behaviours of the type we wish to engineer in robot swarms, both characteristics will tend to exist together [De Wolf & Holvoet, 2004]. That is, we want self-organised collective behaviours to emerge from the local interactions of the individual agents following their rules.

Manual rule design A central problem in swarm robotics is the design of the local rules for the individual agents such that the desired collective behaviour emerges. This is a hard problem for which there exist no analytical solutions. There do exist, however, many techniques for engineering these rules. Bioinspiration was an early and important method; observations of natural swarms leading to inference of possible rules. Reynolds [1987] was interested in designing better, more natural looking, computer animations of large collections of agents, for example birds. Methods at the time were essentially individually scripted movements for each agent. He observed that flocks of birds seemed to follow three rules: 1) Avoid collisions, 2) Align with neighbours, and 3) Move towards centre of neighbours. When combined with a simple perceptual model of limited range and a basic physical model in computer simulation, these rules gave rise to many convincing flocking and schooling behaviours, depending on the parameters of the rules. The Reynolds model inspired much theoretical work, e.g Olfati-Saber [2006] and application to real robots, e.g Hauert *et al.* [2011].

Hand design of the rules is a common method often, as with Reynolds, combined with bioinspiration. Mataric [1993] divides observed swarm behaviours into a number of primitives, including collision avoidance, following, dispersion, and aggregation, and by an iterative process designs rules for twenty individual robots that cause the swarm to show the desired emergent behaviour. This use of primitive swarm behaviours, which can then be combined into more complex ones (the paper describes flocking in terms of the above four) is widely used [Brambilla *et al.* , 2013].

Because there currently exist no analytical methods for rule design, these manual methods rely on the designers intuition and experience, combined with multiple iterative trials, often both in simulation and on real robots.

Automatic rule design Automatic rule design treats the problem as one of optimisation, representing the control rules in some way that can be easily manipulated, and expressing the performance at some task of a swarm of robots executing the rules as an objective or fitness function. The automatic process modifies the rules and measures their performance, usually in simulation, using the objective function. The measured performance guides the process in the exploration of the design space represented by all possible rules. The process generates new sets of rules that perform better. This process continues until some desired level of performance at the task has been reached [Francesca & Birattari, 2016].

The main automatic approach used is various forms of evolutionary algorithm, usually termed *evolutionary robotics* [Nolfi *et al.* , 2000; Vargas *et al.* , 2014]. Commonly in evolutionary robotics, the evolutionary algorithm is used to modify the weights of a neural network which accepts sensor inputs and produces actuator outputs. Sometimes the structure is also modified [Stanley & Miikkulainen, 2002]. Classically, evolutionary robotics was applied to single robots [Bongard, 2013], the application to swarms is known as *evolutionary swarm robotics* [Trianni, 2008]. Other approaches than neural networks and evolutionary algorithms are used, for example, Francesca *et al.* [2014a] use a Probabilistic Finite State Machine (PFSM) as the controller, with parameters optimised using the F-Race algorithm [Birattari *et al.* , 2002]. In all cases, the controller and its parameters encapsulate the rules governing the interaction between agents of the swarm, and the optimisation algorithm explores the rule space to find rules that cause the desired emergent collective swarm behaviour.

Evolutionary algorithms Evolutionary algorithms are a very interesting example of bio-inspiration, due to the possibilities of generating novel solutions beyond those conceived of by a designer. Since evolutionary swarm robotics is well established as a viable technique, we chose this path for the generation of the required rules. It is worth looking briefly at the literature regarding evolutionary algorithms, which is extensive, with regard to applicability to swarm robotics. Evolutionary algorithms, inspired by the process of natural selection, can be broadly divided into three families [Bäck, 1996]. These are Evolution Strategies (ES), Genetic Algorithms (GAs), and Genetic Programming (GP). All share some similarities; they maintain a population of potential solutions which are tested against a fitness or objective function. The measured fitness of the population is used to *select* individuals, which are then *combined* and *mutated* to form a new population, and this process is repeated until some termination criteria is reached, commonly a particular level of fitness or a certain number of iterations. Whitley [2001] gives a good overview of the separate origins and the similarities and differences.

Evolution Strategies [Schwefel, 1993; Beyer & Schwefel, 2002] has its origins in the 1960s in Germany, where it was devised to optimise real parameters of aerodynamic bodies to reduce drag. Each individual solution consists of a set of real valued parameters to be optimised, and usually a corresponding set of mutation strengths known as the *strategy.* Selection is deterministic based on fitness rank, combination randomly chooses parents from the selected set and either uses some form of averaging between parameters of the parents, or randomly chooses individual parameters from one or other parent. The mutation step applies random changes of Gaussian distribution to the parameters, with the characteristics of the distribution controlled by the associated strategy variable, which is itself updated first, using a special strategy mutation operator. The strategy mutator results in adaptive mutation rates, with smaller mutations as solutions near maxima.

Genetic Algorithms, introduced by Holland *et al.* [1975] have a population that consists of individual *genomes*, usually a fixed length bit string, that encode a solution in some way, for example by allocating a fixed number of bits to each parameter. Selection is stochastic, usually based on the fitness rank. A common method is the *tournament* selector, which randomly picks n individuals from the population and returns the fittest of those n. The value of n is the tournament size, varying this controls the degree of selection pressure, higher numbers leading to a greater tendency to chose higher ranking individuals. A typical tournament size is $n = 3$. Combination usually takes two parent genome bit strings, randomly picks a crossover point, and creates two children by swapping the bit strings at the crossover point. Mutation randomly alters every bit in the resultant bit strings with some probability. A common feature is the use of elitism, where the fittest n_{elite} individuals are always transferred to the new population unaltered [Bäck *et al.* , 2000; Davis, 1991; Whitley, 1994].

Genetic Programming, by Koza [1992, 1994], maintains a population of programs. These are not written in a general purpose language but a domain-specific set of functions, variables, and constants. The programs are usually represented as a tree, with functions as inner nodes and the variables and constants, *terminals*, the leaves of the tree. Each program, when run, is a potential solution that is measured for fitness. Selection is stochastic and similar to GA. Also, like GAs, elitism is often used [Poli *et al.* , 2008]. Combination works on tree structures, for example, randomly selecting a node in each of two parents, then creating two new individuals by swapping the subtrees underneath the nodes. Mutation has many variants, altering constants, replacing nodes, or replacing whole subtrees. An additional complexity of GP is the requirement that the tree is *type-consistent*, that only a function node that returns the correct type is placed as a child of another node requiring that type. This affects both combination and mutation operators. Another characteristic particular to GP is the phenomenon of *bloat*, the uncontrolled growth in size of the individuals of the population, possible because of the tree representation. This is analysed by Langdon [2000] and Luke & Panait [2006] examine methods to control bloat.

In reality, many practical uses of evolutionary algorithms make use of features from the three families outlined above as appropriate to the problem domain [Whitley, 2001].

We wish to run evolutionary algorithms within the swarm of robots, which implies the use of parallel or distributed forms. One simple method is the fine-grained approach of executing the expensive fitness evaluation step on each individual in the population in parallel. The behaviour of such an approach is identical to a serial implementation, but relies on a central controller dispatching evaluation sub-tasks, receiving fitness results, then performing the selection, combination, and mutation steps centrally. The algorithm relies on tight coupling and high levels of communication between processes, more suitable to multiprocessor systems. Another approach to parallelism is the *Island Model*, originally coined in Wright [1943] to refer to a theoretical evolutionary biology model of subdivided populations with some amount of migration between them. As first applied to GAs in Whitley & Starkweather [1990], multiple sub-populations of individuals are maintained with a certain level of migration of the fittest individuals between the *islands*. The loose coupling between sub-populations requires only limited amounts of communication and is well suited to a swarm robotics system with each robot hosting a sub-population. Further work showed that can be more effective than a single panmictic population due to the promotion of niching and maintenance of diversity, and examine effects of topology and migration rates [Gorges-Schleuter, 1990; Whitley *et al.* , 1997; Cantú-Paz, 1998; Whitley *et al.* , 1999; Konfrst, 2004].

2.2 Controllers for evolutionary swarm robotics

When using evolutionary methods to discover the rules governing the interactions of the robots in a swarm with each other and the environment that give rise to a desired collective behaviour, it is necessary to encapsulate these rules in a form suitable for the evolutionary algorithm and for controlling the robots. We refer to this as the controller architecture.

Various different controller architectures are used. These include neural networks, probabilistic finite state machines, behaviour trees, and hybrid combinations, [Baldassarre *et al.* , 2003; Francesca *et al.* , 2015; Jones *et al.* , 2016; Duarte *et al.* , 2016]. See Francesca & Birattari [2016] for a recent review.

For swarm robotics systems, there are two very commonly used controller architectures, Neural Networks (NNs) and Finite State Machines (FSMs), illustrated in Figure 2.1. We will briefly describe these architectures, considering the design issues that occur when using them, before moving on to behaviour trees as a possible alternative.

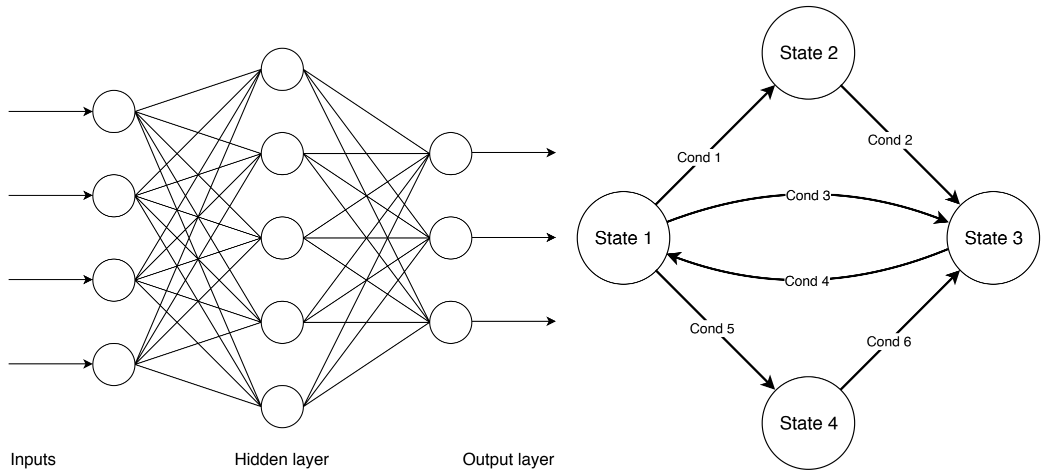


Figure 2.1: On the left, a simple feed-forward neural network. The four inputs are connected to all of the five neurons in the hidden layer, which are in turn connected to the three neurons of the output layer. The hidden and output layer neurons have weights for each input and often a sigmoid transfer function. Inputs would typically be connected to sensors and outputs to actuators. On the right, a simple finite state machine with four states. The FSM can be in only one state at any one time. Transitions between states occur when certain conditions are met, denoted 'Cond *i*'. Typically, the conditions will be associated with sensor inputs and actuator outputs will depend on the current state.

2.2.1 Neural Networks

Neural Networks consist of a directed graph of nodes, called *neurons*, which have at least one input and one output. Each neuron has a transfer function relating the input values to the output value. The inputs may come from external inputs, in the case of a robot this would be sensors, or from the outputs of other neurons. The outputs may connect to external outputs, actuators in the case of a robot, or other neurons. Neural networks can be divided into *feed-forward*, where the network is a directed acyclic graph, and *recurrent*, where cycles exists in the network. The function of the network is completely described by the transfer functions of the neurons and the topology of the network [Haykin, 1994].

The classical target of evolutionary robotics is a fixed topology completely connected feed-forward network with a two layers of neurons, one connected to the inputs, called the *hidden* layer, and one taking the outputs from the neurons of the hidden layer and producing the final outputs. Each neuron has a sigmoid transfer function, *tanh* or the logistic function, Figure 2.2, operating on the weighted sum of the inputs. For inputs I_i, hidden layer neurons H_j, and outputs O_k with numbers of each given by n_{inputs}, n_{hidden}, $n_{outputs}$, the transfer function is given by:

$$H_j = tanh(\sum_{i=0}^{n_{inputs}} w_{ji} I_i)$$
$$O_k = tanh(\sum_{j=0}^{n_{hidden}} w_{kj} H_j) \tag{2.1}$$

The behaviour is defined by the weights w of which there are:

$$n_{weights} = n_{inputs} n_{hidden} + n_{hidden} n_{outputs} \tag{2.2}$$

Typically, one of the inputs to the hidden layer is connected to a fixed value of 1 to provide a bias input. Sometimes an additional bias input is provided to the output layer, although the same effect can be achieved with a hidden layer neuron having all its weights at zero except for the weight of the bias input.

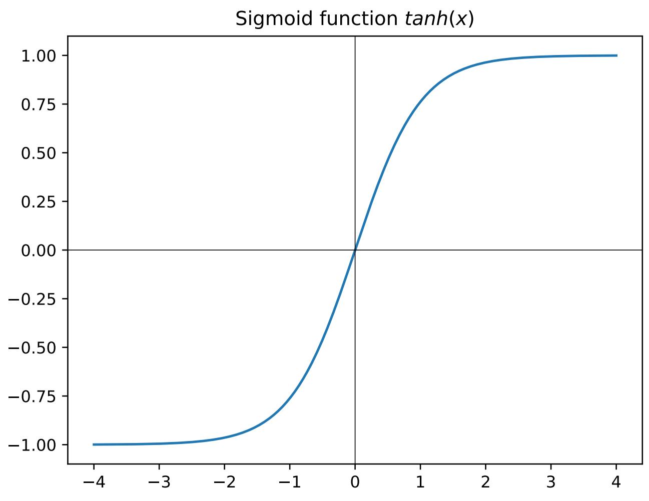


Figure 2.2: Sigmoid function *tanh*, commonly used as the neuron transfer function

The task for an evolutionary algorithm is to discover weights such that the instantiated NN produces the desired behaviour, as defined by the objective function the evolutionary algorithm seeks to maximise.

Even with this simple example, there are a number of design issues which need to be answered. How are the real sensor inputs conditioned into neuron inputs? How are the outputs converted into real actuator controls? How many neurons should there be in the hidden layer? How should the weights be represented within the evolutionary algorithm? Floreano *et al.* [2008] survey some of these questions.

Possible answers to these questions are: inputs can in theory be mapped to any real number, since each input is weighted, but since the *tanh* function is essentially saturated outside input values around $-4..4$, the Evolutionary Algorithm (EA) will need to use the weights to bring the inputs input this range for them to have meaningful effect. This then affects the consideration of ways of representing the weights within the EA. With an ES approach [Beyer & Schwefel, 2002], where genes within the genome are real numbers, perturbed by Gaussian noise in a mutation step, they might just be represented as standard floating point numbers, although allowing the full range of a floating point number results in a vast search space that is uninteresting since it results in a saturated neuron. If we use a classic GA where the genome is a series of bits, we may want to interpret chunks of the bits as a fixed-point format, for example 8 bit chunks interpreted as signed 3.5, capable of representing numbers from -4.0 to +3.96875 in steps of 0.03125. If we use this approach, the more limited dynamic range of the weights necessitates conditioning the inputs to be within that range. Mapping of outputs to actuators is usually done quite directly, for example with a two-wheeled robot, there might be two output neurons whose minimum and maximum output values of -1 and 1 are mapped directly to the maximum reverse and forward wheel velocities respectively. The number of neurons to use within the hidden layer is not obvious, Wang [1994] suggest $2n/3$ where n is the number of input neurons, Fletcher & Goss [1993] suggest between $2\sqrt{n} + m$ and $2n + 1$ when m is the number of output neurons.

The classical NN target described above has a fixed topology, and one important issue with that is the designers decision on the number of neurons imposes a maximum complexity on the controller. Variable topology approaches such as NeuroEvolution of Augmenting Topologies (NEAT) [Stanley & Miikkulainen, 2002] describe an evolutionary algorithm that allows new neurons and edges to be added to the graph. This means the EA has greater freedom to discover solutions outside those possible with a fixed topology.

NNs are probably the commonest target control architecture within evolutionary robotics. It is well understood how to use them to achieve good results, although their black-box nature makes it hard to explain why they produce particular actions for given stimulii [Nelson *et al.* , 2009].

2.2.2 Finite State Machines

An FSM is a system which may be in one of a number of discrete *states*. It can be represented by a directed graph where the nodes represent states and the edges represent *transitions* between states. Each state defines what the outputs of the state machine should be. Each transition defines the input conditions, known as *guard* conditions, that are necessary for the transition between states to take place. A common variant in evolutionary swarm robotics is the PFSM which adds probabilities to the transition guard conditions. If the condition is met, then the transition will take place with a certain probability. Similarly with behaviour trees, FSMs and PFSMs are human readable and transparent, and can be used to construct a system of ordinary differential equations representing the swarm [Liu *et al.* , 2007; Winfield *et al.* , 2008]. One of the main limitations of conventional FSMs is the issue of state and transition explosion with increasing complexity of controller. Each possible combination of outputs is represented by a separate state. Transitions which are common events need to be replicated across many states. There are software formalisms for describing hierarchical state machines that address some of these issues.

As with NNs, we have to make design decisions when using an FSM within an evolutionary robotics context. What are the inputs and outputs? What are the valid combinations of outputs for which we need states? How do we represent these in an EA? A common approach is to define a limited number of states that represent behaviours at a higher level than are typically used for a NN controller. So, rather than direct motor control, we might have states specifying *go-toward-light*, *random-walk*, etc. Inputs are often treated with thresholds or comparisons and combined to form boolean conditions, again having some higher level meaning than is typical with a NN. The EA representation might then consider a fully connected graph of all states, with each possible transition represented by one of the defined conditions (with an associated probability for a PFSM). The numbers of states used is generally low because of the previously mentioned state explosion; the number of transitions in a fully connected graph increasing as $N(N-1)$.

FSMs are sometimes used as a higher level structure for combining lower level behaviours implemented in some other architecture, e.g. Duarte *et al.* [2014] evolve NNs for the lower level behaviours, then evolve the state transitions to achieve useful swarm level behaviours.

2.3 Behaviour trees as understandable controllers

A behaviour tree is a hierarchical tree structured graph of nodes, with leaf nodes interacting with the world and inner nodes combining leaf nodes and subtrees in various conditional and sequential ways. They can accept sensor input and produce actuator output, acting as an agent controller. Behaviour trees have a number of desirable properties; they are modular - any subtree of a BT is itself a complete BT, and any complete BT can be used as a subtree, so they can be used to encapsulate and reuse useful behaviours. They are extensible, there is no combinatorial explosion from increasing the number of nodes, as there is for state machines. They are human readable, at least in principle, and so aid analysis and reverse engineering of evolved BTs. The tree structure can be evolved using the techniques of Genetic Programming. Ogren [2012] shows that FSMs can be represented by a BT, provided there are both sequence and selection type operators. With the addition of a source of randomness, PFSMs can also be represented. Compared to an FSM or PFSM, the state transitions are implicit in the tree structure, and modular[1] structure is explicit; all subtrees are legal behaviour trees.

One way to think about BTs as compared to FSMs is to make an analogy to programming languages [Colledanchise & Ogren, 2014]. In an FSM, changes of state, the transitions, are encoded in the states themselves; once a transition has happened, knowledge of the state transitioned from is lost. This is analogous to the use of GOTOs in early programming languages. By contrast, evaluation of a BT traverses and *returns* from sub-trees, in a way more analogous to function calls than GOTOs. As with more modern programming languages, this promotes modularity and eases the construction and understanding of more complex systems.

The goal of explainability is important for automatically generated robot controllers, and machine learning more generally. Being able to verify that a controller will never produce dangerous outputs, or learning from effective controllers are desirable properties [Samek *et al.* , 2017]. As noted above, there are many possible controller architectures that can be used as the target of automatic design methods. The most widely used, neural networks, are the least explainable, with to date no adequate tools to predict behaviour apart from direct testing [Nelson *et al.* , 2009]. Recent success in Deep Neural Networks (DNNs) has made explainability and trust more important, Došilović *et al.* [2018] survey work in this area. Forms of FSMs and behaviour trees are human readable and the hierarchical structure of behaviour trees makes them amenable to automatic analysis. In Chapter 3 we set out a formal approach to simplifying automatically generated trees to aid analysis, and in Chapter 8 we demonstrate these methods to explain and understand several examples. The hierarchical structure of BTs means that a divide-and-rule approach is possible, unlike with FSMs; simpler sub-behaviours closer to the leaves of the tree can be analysed and understood, then larger behaviours composed of the sub-behaviours themselves explained and understood. However, as Došilović *et al.* [2018] note, there is a performance-transparency trade-off going from NNs to tree or rule-based models; we may have greater trust in explainable and understandable controllers but never reach the same levels of performance that are possible with NNs, although the shape of the trade-off in the domain of swarm robotics is unclear.

Behaviour trees in approximately the form outlined above appeared at around 2002

[1]Perhaps mirroring a fundamental property of nature [Clune *et al.* , 2013].

as a software engineering tool, a method of constructing believable Artificial Intelligence (AI) characters in games, and a way of controlling robots. There are earlier references to 'behaviour trees' in the literature, but these refer to ways of organising the understanding of the behaviours of a system, rather than the current meaning, e.g. Clancy & Kuipers [1993]. They were introduced as a way of specifying software requirements in a formal and hierarchically composable way to ease the path to actual design [Dromey, 2003; Gonzalez-Perez *et al.* , 2005]. Mateas & Stern [2002] describe a behaviour control language for game agents that has similarities to current behaviour trees, although lacking some features. Various authors describe tree-like control architectures for robotics, with advantages of modularity and encapsulation, but with rather different semantics [Fujita *et al.* , 2003; Hoshino *et al.* , 2004; Wang *et al.* , 2005].

The search by the games industry for methods of describing the decision processes and actions of Non-Player Characters (NPCs) resulted in recognisably the first descriptions of what we now think of as behaviour trees. Isla [2005] discusses the problems faced by games designers in handling increasing complexity, and the use of behaviour trees to solve these issues. Champandard [2007] in a talk first introduces the standard graphical notation and starts to formalise them, and Dyckhoff & Bungie LLC [2008] describe the use of behaviour trees for NPC control. It is not clear when *decorators*, single child inner nodes that modify the result of a subtree, were introduced but Lim *et al.* [2010] describes them explicitly. Because this early work was in commercial industry, there is little in the literature until a few years later but even by this point there had been no formal treatment of the semantics [Cutumisu & Szafron, 2009; Weber *et al.* , 2010; Perez *et al.* , 2011; Dill & Lockheed Martin, 2011; Shoulson *et al.* , 2011].

The control of autonomous agents within games has clear similarities to the control of robots, in both cases, the agent must process sensory input and produce actuator controlling outputs that effect change within the environment. This led to greater consideration of BTs within the robotics community with Bagnell *et al.* [2012] demonstrating early use as a robot arm controller, and other practical applications are shown [Abiyev *et al.* , 2013; Scheper *et al.* , 2016]. Ogren [2012] formalised some of the concepts and showed that all FSMs, and with available randomness, all PFSMs can be synthesised from BTs. They only use memoryless versions of compositional nodes. Klöckner [2013a] make a modification to introduce transient behaviours. Colledanchise & Ogren [2014] showed that robustness and safety analysis is aided by the inherent modularity of BTs. In Marzinotto *et al.* [2014] a more consistent formalism is developed, motivating extensions of the available composition node types to include memory forms and demonstrating equivalence between BTs and Controlled Hybrid Dynamical Systems (CHDS). Other work covers performance analysis [Colledanchise & Ogren, 2014], verification [Klöckner, 2013b], application of

GP to behaviour trees [Colledanchise *et al.* , 2015; Jones *et al.* , 2016].

One very important motivation for using behaviour trees as a controller architecture comes from the arguments of Francesca *et al.* [2014a]. They argue that the *reality gap* is a particular problem when the representational power of a controller is too high. That is, it is an example of the *bias-variance* tradeoff in machine learning. A neural network controller, for example, might have very low bias. It has no or few[2] assumptions about the solution. The consequence of this is that it has high variance and is prone during the learning algorithm to overfitting to the training set, and thereby not generalising well. In the specifics of automatically discovering controllers in simulation for swarm robotics, this means that the controller is overfitted to the simulation environment, with its inevitable infidelities to the real world, and thus transfers badly to the real world. Francesca *et al.* argue that by reducing the representational power of the controller, defining some fixed constituent behaviours that are building blocks for the controller, we can move towards the *bias* end of the tradeoff, avoiding overfitting to the simulator environment and thus transferring more successfully to the real world. They successfully demonstrate this approach with a swarm of e-pucks. Behaviour trees are a natural fit for this approach, since we can easily control the expressivity, encapsulating the constituent behaviours we desire within subtrees.

We wish to apply evolutionary methods to behaviour trees in order to discover controllers such that, when instantiated within a swarm of robots, a desired collective behaviour emerges as a result of the individual robot actions and the interactions between them and the environment. In order to do this, we need a complete description of the semantics of behaviour trees, such that we can implement interpreters that perform correctly. This is discussed in detail in Chapter 3, where we formally describe the semantics we use, resolving ambiguities in the existing work, specifying a complete algorithm for tree evaluation, and discuss analysis methods. We also cover the application of GP techniques to BTs.

Here we give a less formal description of the execution of BTs using an example. Figure 2.3 shows a simple behaviour tree for performing collision avoidance. The tree is read in a depth-first left-to-right manner starting at the root of the tree at the top, and this one can be described in english as 'if there is an obstacle in front then turn left otherwise move forward'.

A BT controller is evaluated in the following way; at regular intervals, generally corresponding to the control timestep of the robot although this is not necessary, a *tick* event is sent into the node at the root of the tree. The node performs its function and responds one of only three ways - *success*, *failure*, or *running.* All nodes

[2]Conditioning the inputs and outputs of the NN constitute implicit assumptions about the solution space.

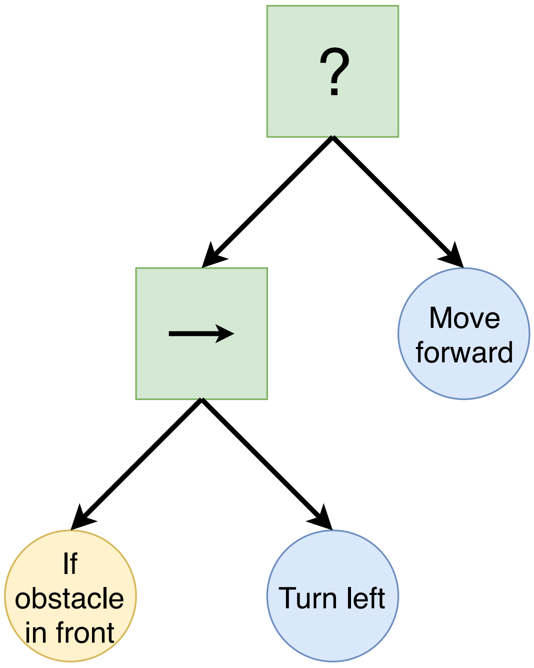


Figure 2.3: A simple behaviour tree for robot collision avoidance

have this interface. The inner, or compositional nodes of a BT are of several types and relatively generic to all BTs. We may choose to use a subset of these, but a minimum set would be the use of *sequence* and *selection* nodes. A sequence node, on receiving a tick, will send a tick event to each of its children from left to right in turn until any return *running* or *failure*, or they have all returned *success*, itself returning that, respectively. A selection node, on receiving a tick, will send a tick event to each of its children in turn until any return *running* or *success*, or they have all returned *failure* again returning that, respectively.

Leaf nodes, on receiving a tick, interact with the environment within which they are embedded. Within this work, we formalise the environment as a set of variables termed the *blackboard*[3] which constitutes the interface between the BT and the real world. It consists of a set of variables that reflect the value of sensors, control actuators, or act as memory. Leaf nodes can be conceptually divided into *query* and *action* nodes. A *query* node purely returns a result based on the blackboard with no side effect. An *action* node may result in alterations to the blackboard, that is, perform an action on the environment.

Returning to the example in Figure 2.3, the node at the top is a *selection* node, and will activate with a *tick* the left hand child tree first, and only if this returns *failure* will it activate the right hand child node, an action node that causes the robot to mode forward. The left hand subtree is a *sequence* node. This node will activate its left hand child first, and only if it returns *success* will it activate the right hand

[3]This is terminology from the games industry

node. causing the robot to turn left.

2.4 Robot design

We want a robot design that has sufficient computing power on-board that a swarm of them can host evolutionary algorithms. This must also be cheap to build. This work is focussed on robots working on a 2D surface, mainly because of the prior knowledge of the author and the availability suitable infrastructure. There is no reason that the principles cannot be also applied to any suitably performant 3D swarm platform.

Moore's Law [Mack, 2011] is the observation made in 1965, six years after the invention of the planar transistor, that the number of minimum cost components on a silicon chip doubles every year. Amazingly, 50 years later, this exponential growth is still broadly true, though now generally stated as the number of transistors per chip doubling every 18 months. This scaling comes from the continual reduction of the size of the transistors, the economical maximum size of a chip remaining roughly constant. Closely related is Dennard's Law [Dennard *et al.* , 1974; Bohr, 2007] which states that as transistors shrink in size, they get faster and consume less power in proportion. Together, this means that computational performance per Watt tracks the exponential increase in transistor counts.

In the late 90s and early 2000s, the increasing transistor budget was used to increase the performance of serial Central Processing Units (CPUs) with larger caches, deeper pipelines allowing faster clock speeds, superscalar designs that preserved the illusion of a serial stream of instructions, while extracting parallelism behind the scenes. But the limits to this approach lead to a plateauing of clock speed and then multicore CPUs to consume the available transistors. Alongside this, the games industry was a major driver in improving the performance of 3D graphics rendering hardware, initially with fixed function pipelines, but in 2001 the first programmable floating point graphics pipelines were introduced, and the raw processing power started to outstrip that available on the CPU. Because the problem domain was limited to graphics rendering, an inherently parallel operation, there were no legacy constraints on architecture and the programming models were explicitly parallel. The availability of large amounts of processing power on commodity graphics cards, albeit difficult to use, made them attractive to the scientific community. By 2004, a typical GPU such as the Nvidia 6800 Ultra had a peak performance of 40 GFLOPS and a power consumption of 110 W, 0.35 GFLOPS/W [Manocha, 2005]. More convenient APIs to use this processing power like Nvidia's CUDA [Nvidia, 2007] and Khronos' OpenCL standard [Khronos OpenCL Working Group *et al.* , 2010] became available, enabling general purpose computation on graphics processing units (GPGPU). See Keckler *et al.* [2011] for more discussion of these trends.

As with the desktop space, so we see the same trends with mobile devices. Because they are hand-held and passively cooled, the power envelope of a mobile device is of the order of a few watts, otherwise the casing becomes uncomfortably hot to hold. By the scaling laws above, after ten years we should expect to see roughly the same processing performance at one hundredth of the power consumption, and indeed we do. The Samsung Exynos 5422 mobile phone SoC has a peak performance of around 120 GFLOPS at a power consumption of 5 W, or 24 GFLOPS/W. Grasso *et al.* [2014] looks at at High Performance Computing (HPC) applications of mobile SoCs and shows much higher performance with lower power consumption than desktop equivalents.

This increase in the available computational power in mobile devices has not been reflected in available robotics platforms. A number of different platforms have been used for swarm robotics research. The e-puck by Mondada *et al.* [2009] is widely used for experiments with numbers in the tens. Rubenstein *et al.* [2012] introduced the Kilobot, which enables swarm experiments involving over 1000 low-cost robots. Both platforms work on a 2D surface. Other platforms include Swarmbots [Dorigo *et al.* , 2004], R-one [McLurkin *et al.* , 2013], and Pheeno [Wilson *et al.* , 2016]. Swarm platforms working in 3D are also described, Hauert *et al.* [2009] demonstrate Reynolds flocking [Reynolds, 1987] with small fixed-wing drones, see also Kushleyev *et al.* [2013]; Vásárhelyi *et al.* [2014]. Most described platforms are homogeneous, but heterogeneous examples exist such as the Swarmanoid [Dorigo *et al.* , 2013]. It is only with the very recent platforms of the Pi-puck and Pheeno (unavailable at the time of design of Xpuck) that the processing power exceeds 1.2 GFLOPS.

We designed our robot, the Xpuck, explicitly with the e-puck in mind, because, like many labs, we already have a reasonably large number of them. The e-puck is very successful, with in excess of 3500 shipped units, perhaps due to its simple reliable design and extendability. Expansion connectors allow additional boards that add capabilities. Three such are relevant here because they extend the processing power of the e-puck. The Linux Extension Board [Liu & Winfield, 2011] adds a 200 MHz Atmel ARM processor running embedded Linux, with wifi communication. The e-puck extension for Gumstix Overo COM is a board from GCTronic that interfaces a small Linux single board computer, the Gumstic Overo Earthstorm[4], to the e-puck. A recent addition is the Pi-puck [Millard *et al.* , 2017] which provides a means of using the popular Raspberry Pi single board computers to control an e-puck. The extension board connects the Pi to the various interfaces of the e-puck and provides full access to all sensors and actuators except the camera. The design described here has more computational power than any of the above.

[4]`https://store.gumstix.com/coms/overo-coms/overo-earthstorm-com.html`

2.5 Reality gap

When using evolution or other methods of automatic design within a simulated environment, the problem of the transferability of the controller from simulation to real robots arises, the so-called *reality gap*, or *sim-to-real* problem. There are various approaches to alleviating this. This issue arises due to the differences between the simulated environment and the real world. It is generally the case that there is a direct trade-off between higher simulation fidelity and slower simulation speed [Vaughan, 2008]. Given that we have finite computational resources available, how do we best spend them? Evolutionary algorithms and other automatic methods rely on many simulations, so we wish these to be as fast as possible, but low fidelity simulations result in solutions fitted to something different than the real world, how do we ensure they transfer well to control real robots?

Firstly, there are methods that are aimed at making the automatically discovered controllers resistant to the differences between simulation and reality, such as noise injection within a minimal simulation [Jakobi *et al.* , 1995; Jakobi, 1998], making transferability a goal within the evolutionary algorithm [Koos *et al.* , 2013; Mouret & Chatzilygeroudis, 2017], and varying the simulator parameters [Peng *et al.* , 2018; Tobin *et al.* , 2017] Reducing the representational power of the controller [Francesca *et al.* , 2014a, 2015; Birattari *et al.* , 2016] seems to be a powerful technique and is a strong motivator in our use of BTs, which allow for easy encapsulation and tuning of the granularity of individual behaviours exposed to the EA. Birattari *et al.* propose a manifesto of good practice for automatic off-line controller design [Birattari *et al.* , 2019].

Other approaches use reality sampling to alter the simulated environment to better match true fitnesses, [Zagal *et al.* , 2004; O'Dowd *et al.* , 2014]. This requires either off-board processing with communication links to the robot, or sufficient processing power on the robot to run simulations. Related is the concept of surrogate fitness functions [Jin, 2011] with cheap but inaccurate fitness measures made in simulation and expensive but accurate measures made in reality. Some works use simulation for the initial automatic controller generation, followed by a short period on training on a real robot [Rusu *et al.* , 2016].

Embodied evolution directly tests candidate controllers in reality. This sidesteps the reality gap entirely by not having simulation at all. When applied to swarms [Watson *et al.* , 2002] the evolutionary algorithm is distributed over the robots [Usui & Arita, 2003; Bredeche *et al.* , 2012; Doncieux *et al.* , 2015]. The time taken to evolve solutions can be days or weeks, and badly adapted controllers pose a risk to the robots. Hybrid approaches, using some simulation and some evolution or other learning in reality exist, but much embodied evolution work with collective robotics never actually uses real robots because of the cost and time implications [Bredeche *et al.* , 2018].

Related is the idea of using internal simulation models as a way of detecting differences between those models and reality. This can be means of detecting malfunction and adapting [Bongard *et al.* , 2006], or asking *what-if* questions, so as to evaluate the consequences of possible actions in simulation [Marques & Holland, 2009]. This is applied to the fields of both robot safety and machine ethics [Winfield *et al.* , 2014; Winfield, 2015; Vanderelst & Winfield, 2018; Blum *et al.* , 2018]. Any robot relying on simulation for its ethical or safe behaviour must either embody that simulation or use extremely reliable communications links, which are difficult to achieve. Swarms are usually assumed to be robust to failure, but Bjerknes & Winfield [2013] show that this is not always the case. By using internal models and observing other agents within the swarm, agents not behaving as predicted can be identified [Millard *et al.* , 2013, 2014]. The system we describe is well-suited to these types of study.

The approach we take in this work to tackling the reality gap combines several of the techniques detailed above. Firstly, we take inspiration from Francesca *et al.* [2014b] and use the modularity of BTs to design useful sub-behaviours of a granularity we hope will decrease the representational power of the controller architecture, increasing resistance to reality gap effects, while maintaining sufficient expressivity for the EA to find effective solutions. Secondly, we pay careful attention to tuning the simulator to minimise measured differences with reality and we inject noise to mask the some of the remaining differences between simulator and reality [Jakobi *et al.* , 1995]. Finally, we modify the allowed behaviours of the controllers to avoid particularly problematic areas of simulation fidelity, such as collisions.

Having described the background of the work, in the next chapter we move on to look in detail at behaviour trees, making explicit the semantics we use, and detailing all of the inner node types used, describing ways the BTs can be transformed into different representations, and can be automatically manipulated for the purpose of analysis.

Chapter 3

Behaviour trees

When using evolutionary methods to discover controllers for producing a desired collective behaviour when instantiated within a swarm, the question arises of what controller architecture to use? There are a number of aspects by which this question may be examined. Do there exist techniques for representing a description of the controller in a suitable way to undergo evolution within an evolutionary algorithm? How easy is it to examine an evolved controller in order to reverse engineer it and perhaps learn new swarm control techniques? Is the architecture capable of representing controllers of varying complexity in a flexible way? Can we abstract and compose sub-behaviours in a modular way? In this chapter, we argue that behaviour trees are a controller architecture with a number of desirable properties, and then discuss in depth the theory and design of BTs for use in evolutionary swarm robotics.

3.1 Behaviour tree theory

A behaviour tree is a hierarchical tree structured graph of nodes, with leaf nodes interacting with the world and inner nodes combining leaf nodes and subtrees in various conditional and sequential ways. Behaviour trees are widely used in the games industry to control the decision and action processes of non-player characters, where they have mostly supplanted previous use of FSMs. The control of autonomous agents within games has clear similarities to the control of robots, in both cases, the agent must process sensory input and produce actuator controlling outputs that effect change within the environment. This has led to greater consideration of BTs within the robotics community and the formal treatment of them in various works, detailed in Chapter 2.

Behaviour trees have a number of desirable properties; they are modular - any subtree of a BT is itself a complete BT, and any complete BT can be used as a subtree, so they can be used to encapsulate and reuse useful behaviours. They are extensible, there is no combinatorial explosion from increasing the number of nodes, as there is for state machines. They are human readable, at least in principle, and so aid analysis and reverse engineering of evolved BTs. The tree structure can be evolved using the techniques of Genetic Programming.

Behaviour trees are evaluated at regular intervals, generally corresponding to the controller update rate of the robot although this is not necessary. At each update, a *tick* event is sent into the node at the root of the tree, which responds with either *success*, *failure*, or *running*, depending on the evaluation of that node. All nodes have this interface. The process of evaluating the node may involve further ticks being sent to child nodes according to the node type. Eventually ticks will reach leaf nodes, that interact with the *blackboard*, which is a table of variables. The blackboard is the architectural representation of the environment, that is, the sensors and actuators that the robot possesses. When referring to a BT tick, it is important to note that a single tick at the root node can produce a cascade of ticks lower down the tree. All these ticks occur within the same controller update cycle, they are ordered but take zero physical time in theory. This is similar to the *delta cycle* concept in event driven simulators. In practice, the evaluation of a behaviour tree takes a certain amount of processing, and there must be sufficient processing power for this to occur within the robot controller update period. Sometimes within this work we refer to events happening in the same tick, by this, we mean within the same tick at the root node, or same controller update cycle.

The design process for a BT architecture consists of deciding; the representations of the sensors that will be available in the blackboard variables, the way that the blackboard variables will connect to the actuators, and the design of the leaf or *action* nodes that will connect the BT and the blackboard. We also need to decide the representation of the BT within the evolutionary algorithm. Here there are two main possibilities. Firstly to use the standard approaches of Genetic Programming, whereby a computer program is represented as a tree of nodes of function calls. This approach translates very easily to behaviour trees. In fact, since each node has an identical interface, there are no complications with ensuring that only nodes of compatible type are used. The second approach is generative, to use a binary string, for example, as the genome in the evolutionary algorithm, then use that string as the input into a grammar to generate the tree.

Behaviour trees are often represented graphically, with the root node at the top and leaf nodes at the bottom, a left-to-right priority for sequence and selection nodes. There are some symbols for node types that are relatively standard across the literature, which we will use. In addition, we will sometimes use exactly equivalent indented textual representation obtained from a depth-first left-to-right traversal of a tree. Finally, for larger trees it is sometimes clearer with a graphical representation to show the root node on the left and the leaf nodes to the right, with top-to-bottom ordering.

We now look in detail at the various aspects of behaviour trees, starting with the nodes used to construct them, then semantics of behaviour tree evaluation and the necessary algorithms to accomplish this, and various ways in which they can be manipulated and simplified. Apart from the *sequence* and *selection* nodes, which are as specified in Marzinotto *et al.* [2014] but stated here for completeness, all other nodes have either incompletely described semantics or differ in some aspect from previous work, or are completely original to this work.

3.1.1 Composition nodes

Here we cover the inner, or composition, node types. These types are relatively universal across different works. The notation used, of question marks for *selection*, arrows for *sequence*, and diamonds for `decorator` nodes seems to have first been used in the literature by Ogren [2012], with the dotted form representing nodes with memory introduced by Marzinotto *et al.* [2014]. In text form, Marzinotto *et al.* [2014] refer to the memory forms as *sequence** and *selection**. We use the `m` suffix, thus `seqm` and `selm`.

Sequence `seq` A *sequence* node has n children $C_1, ...C_n, n > 1$. When it receives a *tick* event, it sends ticks to each of its children in succession, starting from C_1 and checking each return result for *success* before sending the next tick. If a child returns *running* or *failure*, the node returns *running* or *failure* respectively and ceases sending ticks to further children, otherwise it keeps sending ticks until it reaches child C_n. If all children return *success*, the node returns *success*.

The standard symbol for a sequence node is a box with a rightward pointing arrow within it, to denote the left to right evaluation of the children.

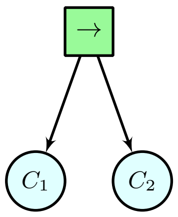


Figure 3.1: The sequence node

Selection `sel` A *selection* node has n children $C_1, ...C_n, n > 1$. When it receives a *tick* event, it sends ticks to each of its children in succession, starting from C_1 and checking each return result for *failure* before sending the next tick. If a child returns *running* or *success*, the node returns *running* or *success* respectively and ceases sending ticks to further children, otherwise it keeps sending ticks until it reaches child C_n. If all children return *failure*, the node returns *failure*.

A selection node is represented graphically with a box and a question mark.

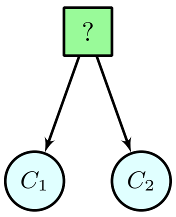


Figure 3.2: The selection node

Sequence with memory `seqm` A *sequence* node with memory behaves similarly to a node without memory, except that the node has a single item of state pointing to the starting child node, which starts pointing to the left-most child. When the node receives a *tick* it sends a tick to this child and all following children in succession. If a child C_i returns *running*, the node remembers this child in the item of state and returns to it at the next tick. Whenever the node returns a terminal condition, that is *success*, due to all children returning *success*, or *failure*, due to any children returning *failure*, the state is reset to point to child C_1. If a child node returns *running*, the node returns *running* likewise.

A sequence with memory node is represented with a box and rightward arrow, as with sequence, but with the addition of a dot above the arrow to denote memory.

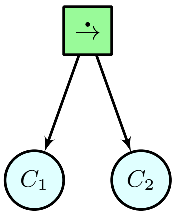


Figure 3.3: The sequence with memory node

Selection with memory `selm` A *selection with memory* node behaves similarly to a *selection* node without memory, except that the node has a single item of state pointing to the starting child node, which starts pointing to the left-most child. When the node receives a *tick* it sends a tick to this child and all following children in succession. If a child C_i returns *running*, the node remembers this child in the item of state and returns to it at the next tick. Whenever the node returns a terminal condition, that is *success*, due to any children returning *success*, or *failure*, due to all children returning *failure*, the state is reset to point to child C_1. If a child node returns *running*, the node returns *running* likewise.

A selection node with memory is represented graphically with a box containing a question mark with a dot above it.

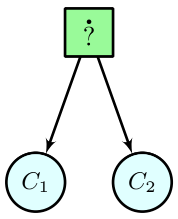


Figure 3.4: The selection with memory node

Parallel We do not use the parallel node in the rest of this work but describe it here for completeness. The parallel node sends the *tick* event to all of its children at the same time. If any children return *running*, the node returns *running*, otherwise it returns *success* if the number of child nodes returning *success* is above some threshold value, and *failure* if it is below the threshold. The threshold value is specified as a parameter of the node.

Decorators `successd, failured, invert, repeati, repeatr` A decorator node is an inner node with a single child that modifies the child tree in some way. In all cases, if the child node returns *running*, the decorator returns *running*. In this work, we specify `invert`, which changes *success* to *failure* and vice versa, `successd` which always returns *success*, and `failured` which always returns *failure*. In addition, we specify two repeat decorators, `repeati` and `repeatr`, which return *running* until it has seen n occurrences of *success* at which point it returns *success*, or *failure* on any occurrence of *failure*. The number n is either specified as a parameter of the node, or chosen randomly from a range specified as a parameter of the node.

A decorator is shown with a diamond shape with the symbols ✓ for `successd`, $\times$ for `failured`, ! for `invert`, n for `repeati`, and $\leq n$ for `repeatr`.

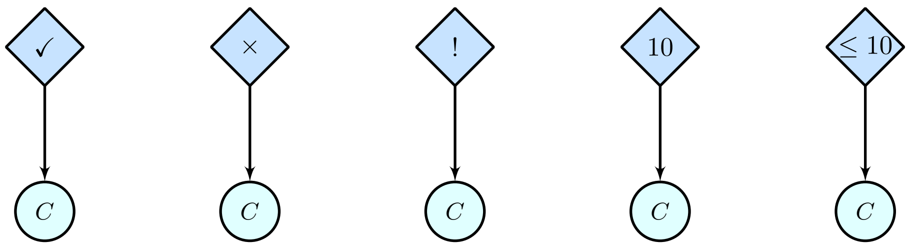


Figure 3.5: Decorator nodes. From left to right: `successd`, `failured`, `invert`, `repeati`, `repeatr`.

The basic node types are summarised in Table 3.1.

Table 3.1: Composition nodes used in this work. If a child returns *running*, the composition node will also return *running*, otherwise the return will depend on the rules of the individual node. Nodes with memory remember the child tree visited in the last *tick* if it returned *running* and resume with that child. Nodes without memory always evaluate children starting from the left, regardless of previous behaviour.

Name	Parameters	Description
`seq`	$\{C_1, C_2, ..., C_n\}$	Tick child trees in order until *failure*, no memory
`sel`	$\{C_1, C_2, ..., C_n\}$	Tick child trees in order until *success*, no memory
`seqm`	$\{C_1, C_2, ..., C_n\}$	Tick child trees in order until *failure*, memory
`selm`	$\{C_1, C_2, ..., C_n\}$	Tick child trees in order until *success*, memory
`success`	C	Always return *success*, regardless of result of tree
`failure`	C	Always return *failure*, regardless of result of tree
`invert`	C	Invert the result of the child tree
`repeati`	C, i	Repeat child tree C i times
`repeatr`	C, i	Repeat child tree C $rand(i)$ times

3.1.2 Leaf nodes

Unlike the composition nodes, the leaf nodes of a behaviour tree are domain-specific. The leaf nodes provide the interface between the BT and the environment it acts within, formalised as the *blackboard*. As with all nodes, they receive a *tick* event from their parent, and respond in one and only one of three ways; *success*, *failure*, and *running*. Conceptually, they may be divided into *query* nodes, that evaluate the environment but make no change to it, and *action* nodes, that may alter the environment in some way. Strictly, the set of all query nodes Q is a subset of the set of all action nodes A such that the blackboard B representing the environment is always unchanged after evaluation $\{\forall Q \in A : B_{t+1} = B_t\}$.

3.1.3 Blackboard

The blackboard represents the environment, or, more formally, part of the state of the system. The entire state of the system is given by $S = B \cup N_s$ where B is the blackboard and N_s is the state of all nodes with memory `seqm` and `selm`.

3.1.4 Behaviour tree semantics

In order to use behaviour trees as a controller architecture, we must completely describe their semantics, such that we can write a correct execution engine for evaluating them. In this section we resolve a particular ambiguity produced when combining nodes with and without memory.

The semantics of behaviour trees has in general been quite poorly defined until recently. The descriptions of the composition nodes given above show how an individual node will behave, but do not completely describe the semantics of tree evaluation, and so with these descriptions we cannot fully reason about behaviour trees, nor turn the node descriptions into tree evaluation code.

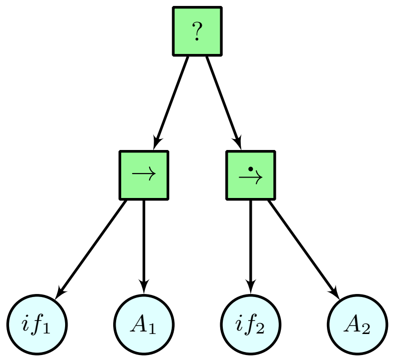


Figure 3.6: Combining sequence and sequence with memory. Each subtree is a guarded sequence where the `if` controls whether the second node of the subtree will be ticked. The first subtree has no memory, so will always check the guard condition. The second has memory, so if the node A_2 was *running* previously, it will not check the condition.

Consider the following example, shown in Figure 3.6. A selection node with two subtrees consisting of a guarded[1] sequence and a guarded sequence with memory. Because the root `sel` and the first sequence `seq` are memoryless, the if_1 query will be evaluated every tick, and if it is true, child action node A_1 will receive ticks, otherwise the sequence with memory `seqm` subtree will receive ticks. Consider the case that A_1 and A_2 are long-running tasks, requiring more than n ticks, and that the condition has always evaluated as false for some number of ticks $< n$, such that A_2 is in the *running* state, and therefore the `seqm` node is returning *running* with its internal index pointing to A_2. What should happen if the if_1 query evaluates for true for one tick, then false again for many ticks $< n$? At that true evaluation, a tick will be sent to A_1, which will then move into a *running* state, and no tick will be sent to A_2. What should happen to the state of A_2? It was *running*, should it remain *running* but in some sort of suspended state, or should it return to some sort of baseline state? Likewise, what should happen to the `seqm`?

These are not difficult questions to answer, but the answers imply certain things. One motivation for the use of memoryless composition nodes is the idea of *reactivity*, that the tree-as-a-controller should be able to describe situations where higher priority tasks can interrupt or take precedence over lower priority tasks. For example, consider a very simple gardening robot, which will start cutting the grass if it is not raining and will continue until finished, even if it starts raining, but must pay attention to its battery level. Cutting the grass takes 10 minutes, and this is not long enough for the grass to get too wet if it starts raining during that time. Charging the battery takes an hour but is a priority. We can map this to the behaviour tree in Figure 3.6 with the conditions and actions shown in Table 3.2.

[1]A guarded sequence is a sequence that starts with a conditional. Unless the conditional, the 'guard', evaluates to *success*, the remaining children of the sequence will not be evaluated.

Table 3.2: Mapping of nodes to behaviours

Node	Behaviour
if_1	'Is battery level low?'
A_1	'Charge battery'
if_2	'Is it not raining?'
A_2	'Cut grass'

The question as posed then becomes; if we were cutting the grass and have to stop to recharge the battery, should we resume cutting the grass, even if it is now raining? Clearly, it might now have been raining for an hour and the grass is far too wet to cut so the answer should be no. The more general argument is that if a *running* subtree is not ticked when it was previously receiving ticks, it is because conditions have changed, so on subsequent future ticks arriving, the whole subtree should be reevaluated. A *running* node should become inactive on not receiving ticks, and the indices of compositional nodes with memory should be reset.

Behaviour tree evaluation We thus evaluate a behaviour tree in the following way:

1. All nodes with memory N_i have a state $s_i \in \{idle, active, running\}$ and all nodes start in state $s_i = idle$
2. Upon each *tick* event at the root of the tree, a two-phase process takes place:
   (a) The *reset* phase, where all nodes that are in state *active* are moved to state *idle*, and *running* nodes are moved to state *active*.
   (b) The *update* phase, where the tree is traversed in a depth-first left-to-right manner, according to the rules of each node type.
   (c) During the *update* phase, we know that any node types with memory that are *idle* were not running at the last tick, so their indices and counters should be started from the reset state.

Simplification when using only memory nodes If we support only the memory forms of the sequence and selector nodes, the evaluation of the tree becomes simpler. There is no need for a two-phase evaluation process, since a *running* node can never be orphaned by the re-evaluation of a higher priority subtree. Evaluation then is simply the recursive descent of the tree according to the rules of each node.

3.1.5 Complete algorithm for behaviour tree evaluation

We can now describe the complete algorithm for evaluating a behaviour tree. We consider only the inner compositional nodes here. The leaf nodes will have domain-specific functionality.

A behaviour tree is a set of nodes T constructed from the set of all possible nodes N. One and only one node within the set is called *root*, and is the only node which is not a child of another node, Eqn 3.1.

A node $n \in N$ is a tuple (t, s, p, c) of kind k, state s, parameters p, and children c, Eqn 3.2. A node may have $q \in \mathbb{N}$ parameters and $r \in \mathbb{N}$ children, though different node kinds have specific numbers of parameters and children, the `seq` node can have an arbitrary number of children but zero parameters, the decorator node `repeat` has exactly one parameter and one child. The evaluation of a node returns only $success \equiv S$, $failure \equiv F$,$running \equiv R$, Eqn 3.7. The notation n_k, n_s, n_p, n_c refers to the kind, state, parameters, and children respectively of the node n.

$$T = \{n_i \in N : \exists! i = root, (i \neq root) \in \mathbb{N}\} \tag{3.1}$$

$$n = (k, s, p, c) \tag{3.2}$$

$$k \in \{seq, seqm, sel, selm, repeat, success, failure, invert, \text{LEAF}\} \tag{3.3}$$

$$s \in \{idle, active, running\} \tag{3.4}$$

$$p = [p_1, p_2, .., p_q] \tag{3.5}$$

$$c = [c_1, c_2, .., c_r] \tag{3.6}$$

$$eval(n) \in \{S, F, R\} \tag{3.7}$$

The memoryless nodes `seq, sel, successd, failured, invert` have no state, but are regarded for the purpose of the algorithm as having the fixed state $n_s \equiv idle$. The `seqm, selm, repeat` nodes have additional local state tracking the last *running* child, or the number of times a child has returned *success.*

To evaluate a tree T for one *tick*, the function Tick(T) from Algorithm 1 is called. This performs the Reset operation on the tree, then recursively descends from the root, performing Update on each node. The Update function is overloaded and specific to each node kind. The Algorithms 2 to 10 specify the Update function for each node type.

Algorithm 1 Evaluate

```
Precondition: Behaviour tree $T$
1: function TICK($T$)
2:     RESET($T$)
3:     UPDATE($T_{root}$)
4: function RESET($T$)
5:     for all $n \in T$ do
6:         if $n_s = running$ then
7:             $n_s \leftarrow active$
8:         else
9:             $n_s \leftarrow idle$
10: function UPDATE($n$)
11:     return KINDUPDATE($n$)        ▷ Call the UPDATE function for this node type
```

Algorithm 2 Sequence, `seq`

```
1: function SEQUPDATE($n$)
2:     for $i \leftarrow 1$ to $|n_c|$ do
3:         $result \leftarrow$ UPDATE($c_i$)        ▷ Get the state of the subtree
4:         if $result = running$ then
5:             return $running$
6:         if $result = failure$ then
7:             return $failure$
8:     return $success$
```

Algorithm 3 Sequence with memory, `seqm`

```
Precondition: Persistent state $index$
1: function SEQMUPDATE(n)
2:     if $n_s = idle$ then
3:         $index \leftarrow 1$
4:     for $i \leftarrow index$ to $|n_c|$ do
5:         $result \leftarrow$ UPDATE($c_i$)
6:         if $result = running$ then
7:             $index \leftarrow i$
8:             return $running$
9:         if $result = failure$ then
10:            $index \leftarrow 1$
11:            return $failure$
12:    $index \leftarrow 1$
13:    return $success$
```

Algorithm 4 Selection, `sel`

```
function SelUpdate($n$)
    for $i \leftarrow 1$ to $|n_c|$ do
        $result \leftarrow$ Update($c_i$)                ▷ Get the state of the subtree
        if $result = running$ then
            return $running$
        if $result = success$ then
            return $success$
    return $failure$
```

Algorithm 5 Selection with memory, `selm`

Precondition: Persistent state *index*

```
function SelmUpdate(n)
    for $i \leftarrow index$ to $|n_c|$ do
        $result \leftarrow$ Update($c_i$)
        if $result = running$ then
            $index \leftarrow i$
            return $running$
        if $result = success$ then
            $index \leftarrow 1$
            return $success$
    $index \leftarrow 1$
    return $failure$
```

Algorithm 6 Success decorator, `successd`

```
function SuccessdUpdate(n)
    $result \leftarrow$ Update($c$)
    if $result = running$ then
        return $running$
    return $success$
```

Algorithm 7 Failure decorator, `failured`

```
function FailuredUpdate(n)
    $result \leftarrow$ Update($c$)
    if $result = running$ then
        return $running$
    return $failure$
```

Algorithm 8 Invert decorator, `invertd`

```
1: function InvertUpdate(n)
2:     $result \leftarrow$ Update($c$)
3:     if $result = running$ then
4:         return $running$
5:     if $result = success$ then
6:         return $failure$
7:     return $success$
```

Algorithm 9 Repeati decorator, `repeati`

Precondition: Persistent state *count*, repeats i

```
1: function RepeatiUpdate(n)
2:     $result \leftarrow$ Update($c$)
3:     if $result = failure$ then
4:         $count \leftarrow 0$
5:         return $failure$
6:     if $result = success$ then
7:         $count \leftarrow count + 1$
8:         if $count = i$ then
9:             $count \leftarrow 0$
10:            return $success$
11:    return $running$
```

Algorithm 10 Repeatr decorator, `repeatr`

Precondition: Child C, repeat count $r = rand(1, i)$

```
1: $result \leftarrow tick(C_i)$
2: if $result = failure$ then
3:     $count \leftarrow 0$
4:     return $failure$
5: if $result = success$ then
6:     $count \leftarrow count + 1$
7:     if $count = r$ then
8:         $count \leftarrow 0$
9:         return $success$
10: return $running$
```

A node about implementation Because of the tree structure of a BT, the code to implement it is easy to write with recursive function calls. In this case, the result of a node after a tick event is explicitly passed as the return value of the

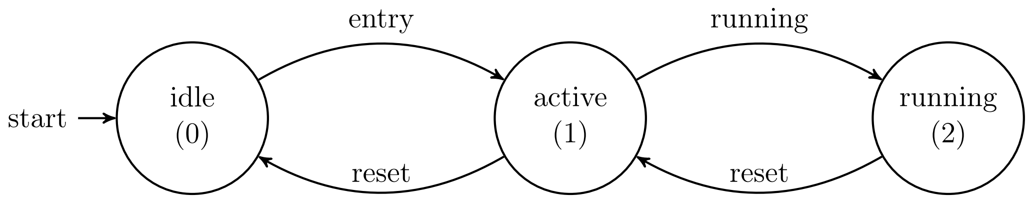


Figure 3.7: State diagram for nodes with memory

evaluation function. However, this is not necessarily the best implementation choice. Some languages explicitly do not support function recursion (e.g. OpenCL). In other cases, there may be limits on the available stack size, limiting the depth of recursion. Passing results this way is far less efficient than might be apparent, since there is at a minimum the overhead of the function return address to be kept on the stack.

Using an explicit stack, and writing without recursion deals with these problems but makes the code more opaque. The pseudocode in this section is written using recursion, we will note the non-recursion method in a later section detailing the implementation of a BT evaluator written for OpenCL.

3.1.6 Memory nodes as syntactic sugar

We wish to perform automatic analysis of behaviour trees, by specifying certain equivalent relations so that we can manipulate and simplify then. One complication with this is the analysis of nodes with memory. By providing a step-by-step method of transforming all trees to memoryless form, we extend the generality of our automatic analysis.

Here, we note that all behaviour tree nodes with memory are essentially syntactic sugar. We can construct a tree using only memoryless forms that behaves identically to a tree that uses `seqm` with the following procedure:

1. For every `seqm` node with memory n_k, we create two explicit items of state $s_k \in \{idle, active, running\}$ and $i_k \in \mathbb{N}$ in the blackboard

2. The entire tree becomes the second child of a `sel` node, with the first child being a tree which performs the *reset* transition of the state diagram in Figure 3.7 for all s_k. This is shown in Figure 3.8 for a single item of state s. All other states s_k within the tree will have a similar subtree as a preceding child of the root node from the last subtree.

3. Each `seqm` is replaced with the tree shown in Figure 3.9. Note, this is for a two child `seqm`, with the children being C_1, C_2.

The state i maintains the index of the last running child. The state s maintains the node state necessary to implement the state machine in Figure 3.7. For the

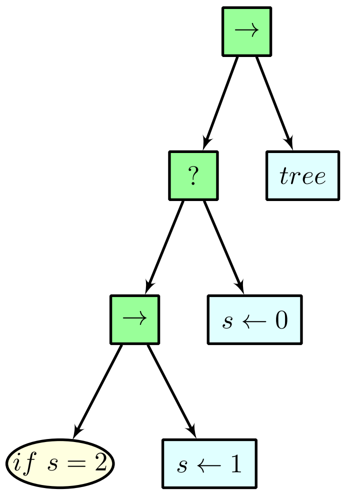


Figure 3.8: Enclosing tree for replacing `seqm` with memoryless forms. The left-hand subtree performs the reset transition on one nodes state. Each node in the original tree will have a similar reset subtree.

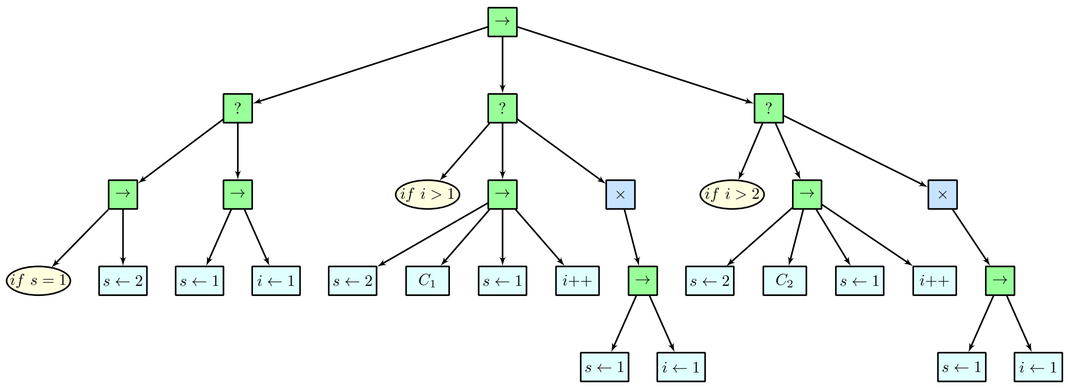


Figure 3.9: Replacement memoryless subtree for `seqm`. When combined with the reset enclosing structure above, a `seqm` with two children C_1 and C_2 can be written using only memoryless composition nodes as shown.

purpose of brevity within the tree diagrams, the states $\{idle, active, running\}$ are abbreviated to $\{0, 1, 2\}$. Figure 3.8 performs the *reest* transition of the FSM; if the state is *running*, it is moved to *active*, otherwise it is moved to *idle*. Once the state has been reset, the original tree is executed, as the second child of the root `seq`. Within the original tree, all instances of `seqm` are replaced with the equivalent tree shown in Figure 3.9. The first branch of the root ensures that the index into the children is reset to the first child if the previous state was not *running*. The second and third subtrees are wrappers around the children C_1, C_2 that serve to extract the implicit information about the *success, failure, running* return state of the original children and use it to appropriately alter the state variables i and s.

Consider that C_1 has returned *success* and C_2 has returned *running*. In this case i will contain 2 and s will contain *running* (2). After the reset subtree has been run, the first subtree of the replacement for `seqm` ensures that if the previous state of the `seqm` was *running* (now *active*), the index i will not be reset. The second and third subtrees have a guard condition to ensure that the tick is directed to the child that was previously *running*, i.e C_2.

By similarly using blackboard entries to hold explicitly the state represented implicitly, we can also rewrite `selm` and `repeat` nodes purely in terms of the memoryless forms.

3.1.7 Manipulation

Given that we can represent all nodes with memory by using the basic memoryless nodes, we now show how the memoryless forms can be manipulated. These rules of manipulation will allow us to perform automatic simplification of behaviour trees for analysis and understanding. Table 3.3 shows the basic node types, expressed symbolically as functions and variables. The leaf nodes are divided into Q query nodes that do not alter the state of the blackboard, and A action nodes that may alter the blackboard state.

Table 3.3: Basic node types symbolically expressed as functions. C, C_1, C_2 are child subtrees, functions $f(x), x \in \{seq, sel, S, F, I\}$ evaluate their children and return their result. Leaf nodes are Q, A, S, F.

Type		
`seq`	$seq(C_1, C_2)$	Sequence
`sel`	$sel(C_1, C_2)$	Selection
`successd`	$S(C)$	Subtree always success
`failured`	$F(C)$	Subtree always failure
`invert`	$I(C)$	Invert subtree result
`query`	Q	Query blackboard state
`action`	A	May alter blackboard state
`successl`	S	Leaf node always success
`failurel`	F	Leaf node always failure

Using this notation we can write any tree as a set of recursive functions. In Table 3.4 we specify a series of equivalences that allow manipulations of an arbitrary behaviour tree.

From Table 3.4 items 1 and 2, sequence and selector nodes with an arbitrary number of children can be constructed by nesting the two-child forms. Subtrees to the right of always failing (sequence, item 3) or always succeeding (selector, item 4) subtrees can be removed. A sequence with an always succeeding first subtree with no side effects is equivalent to the second subtree alone (item 5), and respectively with a selector with an always failing first subtree with no side effects (item 6). An always failing subtree with no side effects is equivalent to just an F leaf node (item 7) and an always succeeding subtree with no side effects is equivalent to just an S node (item 8).

Table 3.4: Tree manipulations. Let C_i be a subtree. Q_i is a subtree with no effect on the state i.e conditional (query) nodes only. $F(C_i)$ is a subtree guaranteed to return *failure*. $S(C_i)$ is a subtree guaranteed to return success. F and S respectively are leaf nodes that just return *failure* or *success*

Rule			
1	$seq(C_1, seq(C_2, C_3))$	$\equiv$	$seq(seq(C_1, C_2), C_3)$
2	$sel(C_1, sel(C_2, C_3))$	$\equiv$	$sel(sel(C_1, C_2), C_3)$
3	$seq(F(C_1), C_2)$	$\equiv$	$F(C_1)$
4	$sel(S(C_1), C_2)$	$\equiv$	$S(C_1)$
5	$seq(S(Q_1), C_2)$	$\equiv$	C_2
6	$sel(F(Q_1), C_2)$	$\equiv$	C_2
7	$F(Q_i)$	$\equiv$	F
8	$S(Q_i)$	$\equiv$	S

By repeatedly applying these manipulations, we can simplify a behaviour tree. This is of particular interest when using evolutionary methods, as a common issue is that of *bloat*, where a large tree is the result of evolution, but much of it may have no effect. Using these manipulations, we can reduce the tree to a more human-readable form that can be analysed, or we can optimise the resource usage.

3.1.8 Equivalence of BT and FSM

Behaviour trees and finite state machines are equivalent. We can show this by providing a step-by-step process to convert between them.

Express an FSM as a BT

Assume a state machine with current state $s_i \in S$, a set of events $e_j \in E$, and transition rules $t_{ij} : s_i \mapsto s_k$ if e_j. We can map this to a behaviour tree with the following procedure, referring to Figure 3.10:

1. Create as the root node a selector with m children, where m is the number of states.
2. For each child corresponding to a state s_i, create a sequence guard, with a check for the current state followed by a selector with $|t_i|$ children, corresponding to the number of transitions that exist out of the state s_i, conditioned by events e.
3. For each child of the selector, corresponding to a particular transition t_{ij} conditioned by event e_j, construct a sequence with a guard followed by the transition.

The figure shows one transition from one state, but this can obviously be extended arbitrarily to any number of state with any number of transitions. One thing that is made explicit is that there is a priority ordering to the transitions; if there are two transitions out of a state, and the two enabling events both occur simultaneously,

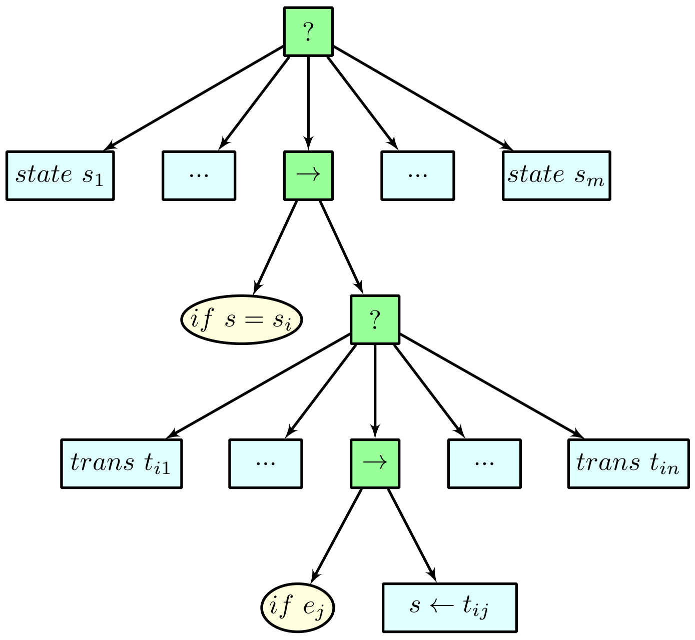


Figure 3.10: Mapping a finite state machine to a behaviour tree. Each state is a subtree of the root selector node, and consists of a sequence with a guard checking the state, and a further subtree which checks for all valid events that trigger a state transition in this state.

only one can occur and that is the leftmost one within the selector. In a state machine diagram, it is either implicitly assumed that there can be no simultaneous enabling events, or the priority ordering is explicitly stated.

The method can be extended to a probabilistic finite state machine (PFSM) by adding a second condition child to the transition sequence, see Figure 3.11. The second guard only allows the transition to take place with probability p.

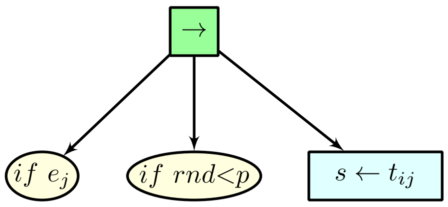


Figure 3.11: Transition sequence extended to support PFSM

Express a BT as an FSM

In order to translate from a BT to an FSM, we use the method described by Colledanchise [2017]. Consider a single node BT. It accepts *tick* events and returns *S,F,R*. We can represent that as a state machine, with a state to represent the single node, with one incoming transition, the *tick*, and three outgoing transitions, *S,F,R*, and a second state which accepts the nodes outgoing transitions *S,F,R* and has a single outgoing transition *tick*.

We can express a sequence in the following way. We replace the single BT node in Fig-

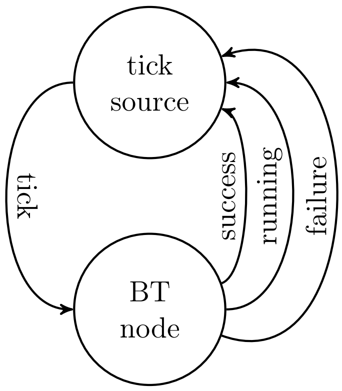


Figure 3.12: State diagram for a single BT node

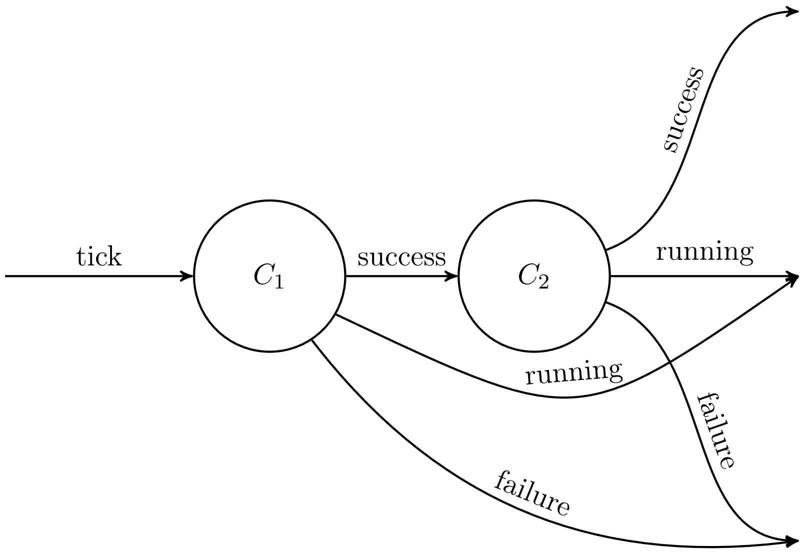


Figure 3.13: State machine fragment for a two child sequence node

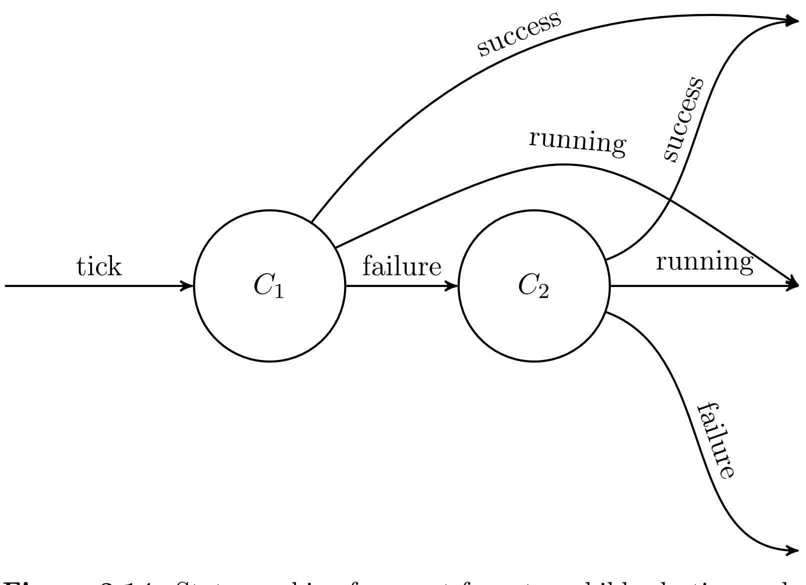


Figure 3.14: State machine fragment for a two child selection node

ure 3.12 with the state machine fragment shown in Figure 3.13. The *tick* events arrive as tick transitions into the first child node C_1, which results in $success, failure, running$. The result causes the respective transition out of C_1 and if it is *success* it transitions to C_2. The three exit transitions are to the enclosing state machine. Likewise, if we need a selection node, we replace the single node with the state machine fragment in Figure 3.14.

We recursively convert the entire behaviour tree in this manner, substituting in state machine fragments in these patterns until there are no more inner nodes to convert. At that point, we have a finite state machine representing the behaviour tree.

We appreciate the argument made by Colledanchise [2017] that there is an analogous comparison between behaviour trees and finite state machines as between *function calls* and the *goto* statement. The essence of the argument is that the method of switching control flow in a program represented by *goto* contains no information about the context from which it came. This mean that it is harder to write modular reusable code, since there is the need to embed specific information about the enclosing context[2]. Conversely, with a function call, the information about calling context is preserved, making a return to that context easy. By analogy with *goto* based programming, as state machines grow in complexity, the number of transitions to keep track of quickly becomes unmanageable. By making the transitions of state in a behaviour tree part of the tree structure, we keep the context that allows modularisation and reuse as complexity grows.

3.1.9 Turing completeness of a BT

It is trivially possible to define a Turing complete behaviour tree. Since the action nodes of a BT are domain specific, we can simply define one as, for example, 'Fetch and execute next instruction from blackboard'. Although true, this obviously is not a useful description. Let us examine in more detail what would be necessary to fully define the blackboard and action nodes necessary to implement a Turing complete system.

Consider a Universal Turing Machine of the Random Access Stored Program (RASP) type similar to that defined by Cook & Reckhow [1973][3]. This consists of an accumulator A, an instruction pointer PC, and an infinite set of registers $\{..., X_{-1}, X_0, X_1, ...\}$, each capable of holding an arbitrary integer. It is capable of executing a limited set of instructions, shown in Table 3.5.

Each instruction occupies two registers $X_{2k}, X_{2k+1}, k \in \mathbb{Z}$. The first register of the pair contains the instruction i and the second contains the parameter j. The program

[2]E.g. the calling context writes a specific *goto* address into the code at the end of the called function, requiring self-modifying code

[3]We remove the input output instructions, and allow negative register numbers

Table 3.5: RASP instruction set

Instruction	Mnemonic	Opcode	Operations	
load immediate	ldi j	1	$A \leftarrow j$	$PC \leftarrow PC + 2$
add	add j	2	$A \leftarrow A + X_j$	$PC \leftarrow PC + 2$
subtract	sub j	3	$A \leftarrow A - X_j$	$PC \leftarrow PC + 2$
store	st j	4	$X_j \leftarrow A$	$PC \leftarrow PC + 2$
branch if positive	bp j	5	$PC \leftarrow \begin{cases} j & \text{if } A > 0 \\ PC + 2 & \text{otherwise} \end{cases}$	
halt	halt	other		

is stored starting at register X_0 and the initial value of A and PC is zero. Each cycle of execution consists of the operations: 1) Examine the contents of $i \equiv X_{PC}$. 2) Perform the operations in the operations column for the opcode i, where $j \equiv X_{PC+1}$. Execution continues until *halt*. Any value $i \notin \{1, 2, 3, 4, 5\}$ causes the execution to halt.

To map this to a behaviour tree, we define the blackboard as containing $\{A, PC, ..., X_{-1}, X_0, X_1, ...\}$. A behaviour tree that implements the operations in Table 3.5 is shown in Figure 3.15.

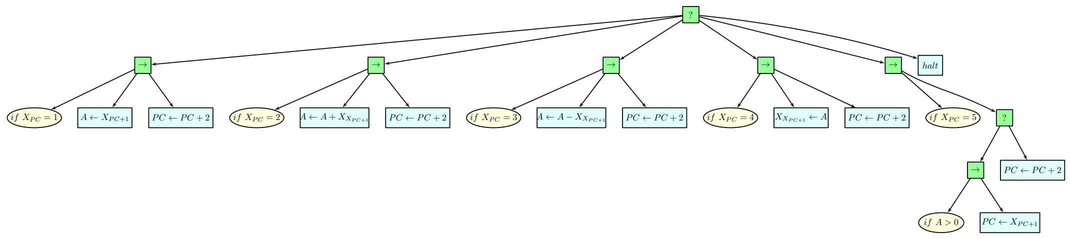


Figure 3.15: Behaviour tree to implement the RASP Turing Machine

Since we can implement a Universal Turing Machine with a suitable behaviour tree and blackboard, it is thus Turing complete and is theoretically capable of any computation.

3.1.10 Subsumption robot controller architecture

A behaviour tree naturally represents the subsumption architecture [Brooks, 1986] with the selection operator. The children represent the increasingly higher level layers of behaviour. In the original paper, the layers are; collision avoidance, aimless wander, explore, map, and then further higher level behaviours. Figure 3.16 shows a behaviour tree for the first two levels; if the robot is about to collide with an object, it stops, otherwise then aimless wander behaviour takes over, which wanders on a fixed heading until approaching an object, in which case a new movement heading is set.

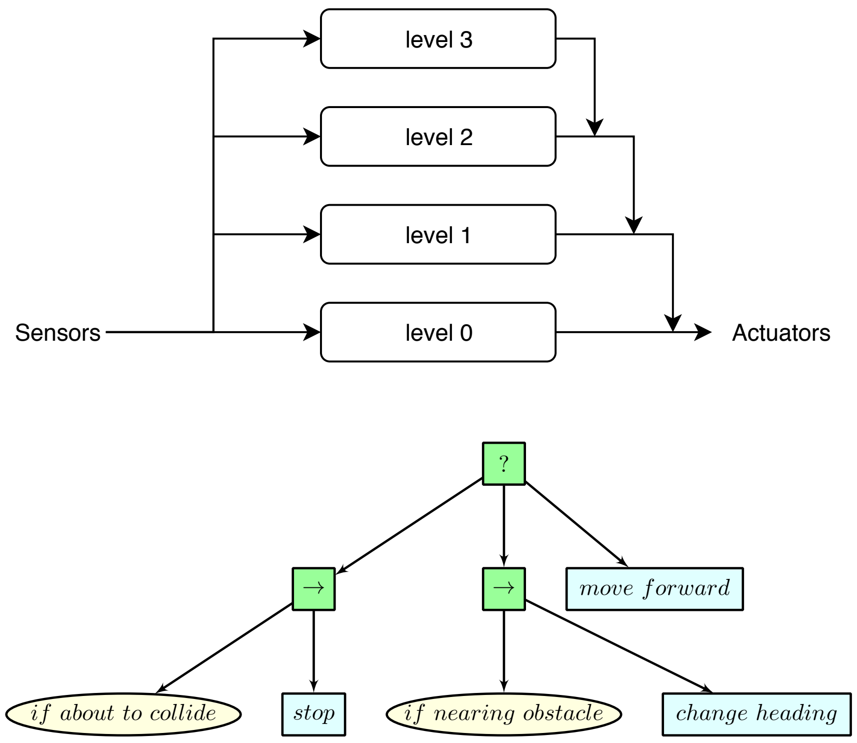


Figure 3.16: Subsumption architecture behaviour tree. Original subsumption architecture from Brooks [1986] on top, and first two layers of a subsumption controller behaviour tree to explore the environment on the bottom

3.1.11 Classes of behaviour tree

We can consider three classes of BT, depending on the composition nodes that are used. Each class is capable of representing a different range of controller architectures.

1. **Blocking:** `seqm, selm` The memory type nodes remember if a child node returns running and subsequent *tick*s are sent there, rather than to child nodes to their left. This is the simplest type of BT and is not reactive, in that if a condition in a higher priority subtree changes, the running subtree will not be stopped and will continue until completion. Only one leaf node can be running in a given *tick*.

2. **Reactive:** `seqm, selm, seq, sel` The addition of the memoryless versions of the select and sequence nodes allows for already running subtrees to be preempted by changing conditions. This allows the implementation of the standard robotics subsumption architecture. Implementation is more complex, because a running node my be orphaned and need to be moved to a non-running state gracefully, such that when ticked again it behaves appropriately. As with the blocking BT, only one leaf node can be running in a given *tick*.

3. **Parallel:** `seqm, selm, seq, sel, par` The `par` node ticks all of its children

in parallel, and waits until a certain number return *success* or *failure*, returning *running* otherwise. Unlike the previous two types, there can be many subtrees in the *running* state. Subtrees in running state may therefore conflict in access to resources, this is outside the semantics of the BT and has to be controlled with other mechanisms.

In this work, we use Type 1 (blocking) behaviour trees for initial investigations and demonstration that it is possible to use the techniques of Genetic Programming to evolve effective BT controllers in Section 4. For the later work with the Xpucks, we use Type 2 (reactive) behaviour trees to allow the use of subsumption architecture controllers.

3.2 Applying evolutionary methods to behaviour trees

We wish to apply evolutionary methods to behaviour trees in order to discover controllers for swarms of robots such that a desired collective behaviour results. Our reason for using evolutionary methods is the large amount of previous work in the literature on evolutionary swarm robotics, as noted in Chapter 2, that can provide some guidance. There is little previous work on evolving behaviour trees, however. In this section, we examine two possible methods.

All evolutionary algorithms follow the same general set of steps: A population of candidates exists, which is then *evaluated* according to some criteria, commonly called the *objective function* or *fitness function*. From this population, individuals are *selected* on the basis of their evaluation and undergo *alteration*, typically by combination and mutation, to form a new population. By repeating this process, the characteristics of the population move towards the goal set by the criteria. There are many different techniques used for each of these steps, giving rise to a wide variety of differently-named subfields, e.g. Evolution Strategies, Genetic Algorithm, Genetic Programming, but the fundamentals are the same.

The concept of *genotype* and *phenotype* is useful here. The *genotype* is the representation we use for an individual within the population, and its *phenotype* is the expression of that representation. DNA and bodies and behaviours, applied to us. This division gets slightly blurrier when applied within evolutionary algorithms. If we are evolving, for example, the coefficients of a polynomial to fit some data, the genotype and the phenotype are one and the same. If we are evolving a set of instructions to a 3D printer, the phenotype is the resultant printed part. In our work, we seek to engineer swarm behaviour., so the behaviour tree is the genotype and the emergent behaviour of the swarm the phenotype.

What should we use as the genotype when evolving behaviour trees? There are two approaches that have been used previously; using a tree representation Lim *et al.* [2010], and using a generative grammar, Perez *et al.* [2011]. In the first approach, the representation is the behaviour tree itself. The genetic operators that are used for the *alteration* step manipulate the tree(s) directly. The second approach uses a completely different representation, usually a flat fixed-length bit string which is manipulated with genetic operators familiar from Genetic Algorithms. The behaviour tree of an individual is generated using a grammar. We examine this approach next.

3.2.1 Grammatical generation

We describe allowed behaviour trees using a Backus-Naur Form (BNF) grammar, in the same way that legal programs in a programming language are often described. Consider the following, where each line is a *production rule*, terminals are in quotes, and alternatives are separated by |:

Listing 3.1: Simple behaviour tree BNF grammar

```
<tree>          ::= <node>
<node>          ::= <leaf_node> | <inner_node>
<inner_node>    ::= <seq> | <sel>
<seq>           ::= "seq2" <node> <node> | "seq3" <node> <node> <node>
<sel>           ::= "sel2" <node> <node> | "sel3" <node> <node> <node>
<leaf_node>     ::= "if" <condition> | "left" | "right" | "forward"
<condition>     ::= "a" | "b"
```

This completely describes a behaviour tree grammar. A tree consists of a node, which may be an inner or a leaf node. The inner nodes can be `seq` and `sel` with either two or three children, which are themselves nodes, and leaf nodes can be either one of two conditions (`a` or `b`), or one of three actions (`left`, `right`, `forward`). To convert a bitstring to a tree using this grammar, we can use the following procedure: Maintain a state s_i, corresponding to the current location within the grammar, and initialise this to $s_0 = \texttt{tree}$. Perform a depth-first recursive traversal of the grammar, that is, set the next state s_{i+1} to each of the elements of the right hand side of the rule specified by the current state s_i in turn. Where there is a choice |, consume sufficient bits of the bit string to make that choice. If the element is a terminal, enclosed in quotes, output it. If the end of the bit string is reached, wrap around to the beginning again. Continue until the entire tree has been generated.

We illustrate with an example: take the bit string `1001000001001000`, and consume from the left. The following shows the consumed bits, the state, and the output:

In the process of generating the tree, we wrap around once and consume the first bit a second time (marked with a *) when choosing the last `<condition>`.

There are advantages and problems with the grammatical generation approach. Using fixed length bit strings is a very familiar approach, techniques of mutation and

Listing 3.2: Grammatical generation from bitstream `1001000001001000`

```
1   <inner_node>
0   <seq>
0   "seq2" <node> <node>
1   <inner_node>
0   <seq>
0   "seq2" <node> <node>
0   <leaf_node>
00  "if" <condition>
1   "b"
0   <leaf_node>
01  "left"
0   <leaf_node>
00  "if" <condition>
1*  "b"

seq2
    seq2
        if b
        left
    if b
```

crossover are well described in the literature. It is rather elegant. There are several problems though, it is quite easy to construct a bit string that never terminates in the generation process. Consider if we had used the string `1111`, or any string consisting of all ones with the above grammar. This would produce a never ending series of `sel3`s, descending ever deeper. Obviously, we can modify the generative process to handle this, eg, below a certain depth we always choose a leaf node rather than an inner node, but this compromises the elegance of the approach. Wrapping round and reusing the bit string when the tree generation has yet to terminate also feels rather unsatisfactory, although this is somewhat of a value judgement.

Controlling the structure of the tree involves controlling the choice mechanism, that is, how many bits we consume, and how those bits are used to make a selection. Our simple example above, which only has power-of-two options in each choice is a straight-forward mapping, but we can effectively have a set of probabilities for each choice, and consume sufficient bits to give a suitably fine-grained random number.

It is not obvious how many bits the string should contain. We can only represent a maximum of 2^N distinct trees with N bits, but how does increasing the number of bits affect the behaviour of the evolutionary algorithm? We can find little in the literature addressing this. Trivially, if the average number of bits needed to make a choice in the grammar is n, and we have storage capacity for a tree of m nodes, there is no point in having a bit string longer than $n \cdot m$, since bits beyond that length will not get used. We have simulator memory for a maximum of 2048 nodes (see Chapter 5) which implies an upper bound of useful bitstream length of several thousand bits. What is more interesting is a string length that allows many possible trees yet is much shorter than the upper limit, consider $N = 32$. There are a maximum of $2^{32} = 4.3 \times 10^9$ possible trees, yet any tree larger than a few nodes will be reusing bits due to the wrap round behaviour. The actual number of trees is likely to be much lower, consider the simple grammar above, all bit strings starting with zero will consist of a single leaf node, of which there are only five possible. There is obviously scope for much investigation, with little guidance from the literature.

3.2.2 Genetic Programming

Here we use the term Genetic Programming (GP) [Koza, 1992] for the techniques of evolving computer programs represented as tree structures, to solve problems. Rather than the indirect representation of grammatical generation, we directly represent the behaviour tree within the evolutionary algorithm. We use genetic operators that can manipulate the tree representation.

Traditional GP specifies a set of functions and terminals, where all functions with one or more parameters are inner nodes of a tree, and terminals and functions with no parameters are leaf nodes. The tree structure is much like what a compiler will produce when parsing a program. Unlike general purpose programming languages though, in traditional GP the functions and terminals are usually chosen to fit the problem domain. It is also important that a tree produced by the evolutionary algorithm is *type-consistent*; the return type of functions, and terminal types, are compatible with the types of the parameter inputs they provide. This can be achieved by defining the problem such that all types are compatible, only allowing genetic operators to create legal trees, automatic type conversion, and other techniques.

The standard genetic operators of crossover and mutation have their equivalents with the tree structure of a GP. There are many possibilities, but a simple crossover operator picks at random one node on each of the two trees and swaps the subtrees below those nodes to create two new individuals.

Behaviour trees are interesting from the perspective of GP, since they have a completely uniform and consistent interface; all trees are valid sub-trees, any sub-tree is a complete tree in its own right, and the return value is always one and only one of the three values *success*, *failure*, and *running*. Thus there are no complications regarding type-consistancy. We may have nodes which have parameters, but these would not be regarded as terminals in the same way as with traditional GP.

So, a GP-style evolutionary algorithm with behaviour trees would proceed something like this: Create an initial random population. There are various methods, but Koza's *ramped-half-and-half* is recommended by Poli *et al.* [2008], and we use this method. This creates trees up to a maximum depth which are either `full` trees, with every leaf at the same depth, or `grow` trees, where nodes are chosen randomly until the maximum depth is reached. The initial population is evaluated and a new population is generated by selecting individuals on the basis of the evaluation, combining with tree crossover, and applying various forms of mutation.

Figure 3.17 shows an example of how the tree crossover operation works; a node from each parent P_1 and P_2 is selected at random, shown ringed in red. Two new individuals, C_1 and C_2 are created by copying the parents and swapping the subtrees at and below the selected nodes. The majority of nodes in any tree are leaf nodes, so a uniform random selection of nodes will lead to little change in the larger structure of the tree. For this reason, it is recommended by Poli *et al.* to bias the selection of nodes such that the probability of an inner node being selected is much higher, they suggest $P_{inner} = 0.9$.

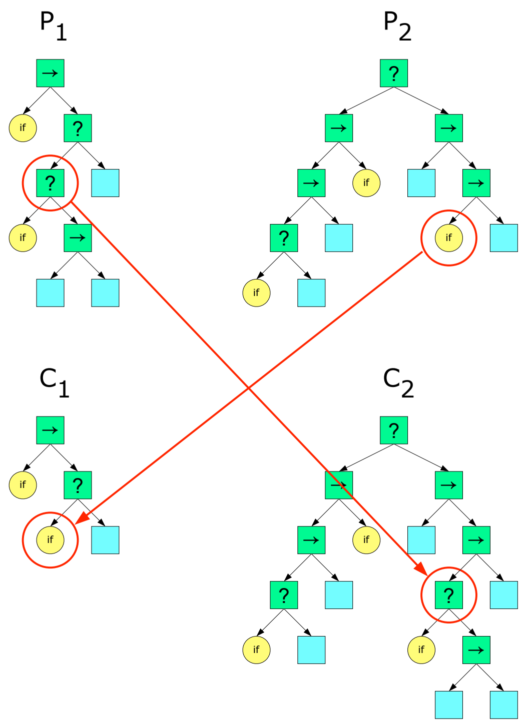


Figure 3.17: Tree crossover genetic operator. A random node is chosen on P_1 and P_2 and two new individuals are created by copying the parents and swapping the subtrees starting at the chosen nodes.

Figure 3.18 shows examples of two mutation operators being applied to an individual I. The first, row A, is an example of *point* mutation, where a node is selected at random within the individual (often subject to the $P_{inner} = 0.9$ rule above) and that node is changed to a different but compatible node, that is, a node that has the same *arity*, or number of children. This is generally an easier operation with behaviour trees than other forms of GP, since there is no need to take return type into consideration. The second row, B, shows an example of subtree mutation, where a the subtree starting from a randomly selected node within the individual I is replaced with a new, randomly generated subtree. In both cases, the mutation

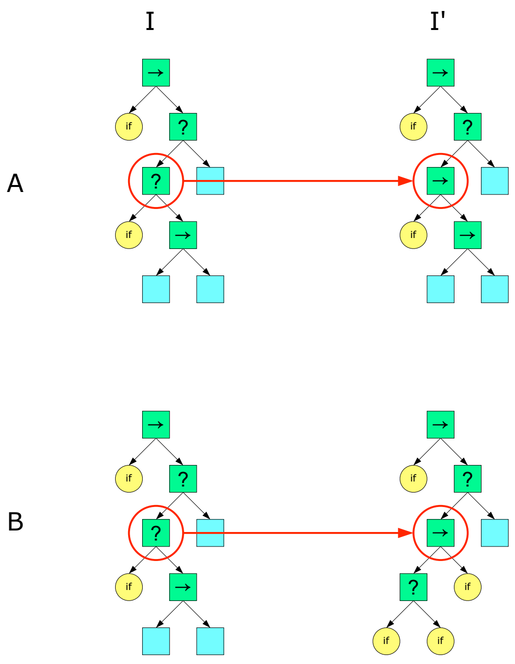


Figure 3.18: Tree mutation operators. A is *point* mutation, where a randomly selected node shown ringed is replaced with another node of the same *arity*, that is the same number of children. B shows *subtree* mutation, where the selected node and its subtree are replaced with a new randomly generated subtree.

takes place on a copy of the individual I, to give a new individual I'.

Another form of mutation we use is *parameter* mutation. As a consequence of all nodes of a behaviour tree being inner *composition*, of leaf *action* or *query* nodes, there do not exist separate terminal nodes for constants and variables as there does in conventional GP. But the leaf nodes specifying some query or action often have associated parameters, for example a particular blackboard register, or a constant value. Parameter mutation, then, selects a node at random and if it has parameters, mutates them according to their allowed values. It is generally leaf nodes that have parameters, though `repeat` decorators and the `probm` probabilistic selector in the work below do also.

3.2.3 Conclusion on BT representation

We decided to use the direct tree representation of behaviour trees within the evolutionary algorithm because of the perceived greater uncertainties with using grammatical generation and the lack of guidance in the literature. It is not clear which would be better, so this is fundamentally quite an arbitrary choice, but partially driven by the thought that it would be easier to find guidance in the literature when using a quite traditional form of GP.

3.3 Conclusion

In this chapter, we have briefly looked at other control architectures for swarm robotics before proposing the use of behaviour trees as a control architecture with some compelling advantages, namely their modularity and ability to encapsulate complete sub-behaviours, and their human-readability and amenability to automatic analysis. In addition to these advantages, their modularity makes it easier to tune their representational power compared to other controller architectures when used with automatic discovery methods, potentially aiding the reduction of reality gap effects.

We discuss in detail the semantics of behaviour trees, from the inner composition nodes that are common across most works, to the action nodes and blackboard which provide the domain-specific interface between the environment and the behaviour tree controller. By formally setting out the semantics, we can describe the complete algorithms needed to implement a practical behaviour tree interpreter for use in robots. We also show methods of manipulating the trees such that they can be automatically simplified, and demonstrate the equivalence of BTs and finite state machines by providing an step-by-step means of converting between them.

Given a behaviour tree architecture, we then discuss how they may be automatically discovered using evolutionary techniques, the possible representations that may be used, and look in more detail at grammatical generation and Genetic Programming as applicable techniques, deciding upon GP as the route we chose to follow.

Chapter 4

Evolving behaviour trees for swarm robotics

In the previous chapter we looked at the theory of behaviour trees and how they might be automatically generated by evolutionary methods. In this chapter we test this with a real swarm of Kilobot robots. We demonstrate that it is possible to evolve a behaviour tree controller in simulation for a swarm of Kilobots to a high level of fitness. We then transfer this controller to real robots and show that the swarm maintains a good level of fitness. Finally, we analyse and explain the evolved controller.

Some of this work has been published as *Evolving behaviour trees for swarm robotics* in Proceedings of DARS 2016 - International Symposium on Distributed Autonomous Robotic Systems [Jones *et al.* , 2016].

We design a behaviour tree controller architecture suitable for instantiation in a swarm of kilobot robots. We then automatically evolve behaviour trees in simulation to enable the swarm to perform a collective foraging task. The fittest behaviour tree is then evaluated in a swarm of real robots and analysed.

This chapter is organised as follows; Section 4.1 gives a brief overview of the kilobot platform, Section 4.2 describes the experimental procedure, Section 4.4 details the results and Section 4.5 discusses results and possible further work.

4.1 Kilobots

Kilobots are small cheap robots introduced by Rubenstein *et al.* [2012]. They are capable of motion using two vibrating motors, communication with each other over a limited range using Infra-red (IR), distance sensing using the communication signal strength, environmental sensing with an upwards facing photo detector, and signalling with a multicolour Light Emitting Diode (LED). They are cheap enough to make it practical to build very large swarms and capable enough to run interesting experiments. Collective control of the kilobots in order to program and to start or stop them is achieved using a high intensity IR system using the same protocol as the inter-kilobot communication system.

Each kilobot has 32kbytes of program memory and 2kbytes of RAM. This allows the creation of reasonably complex programs. Programming the kilobots is accomplished using C targeting the *kilolib* API, which provides abstractions for accessing the communication system and the sensors and actuators.

4.2 Materials and methods

Foraging as a collective task is often used as a benchmark for swarm systems [Winfield, 2009b]. It involves robotic agents leaving a nest region, searching for food, and returning food to the nest. Cooperative strategies are often more effective than individuals acting alone [Cao *et al.* , 1995].

We designed a simple foraging experiment for a swarm of kilobots in an arena upon which we can project patterns of light to define the environment (Fig. 4.1). At the centre of the arena is a circular *nest* region. Surrounding this is a gap, then beyond that is the *food* region. A kilobot which moves into the food region is regarded as having picked up an item of food, a kilobot which is carrying an item of food that enters the nest region is regarded as depositing the food in the nest. Multiple kilobots are placed in the central region in a grid and all execute the same controller (homogenous swarm) for a fixed amount of time. The fitness of the swarm is related to the total amount of food returned to the nest within the test time. The maximum possible number of food items depends on the starting spatial distribution of the kilobots. For a theoretical maximum, assume that the kilobots start on the edge of the *nest* region and for the duration of the test move directly back and forth between *nest* and *food* regions by the shortest distance. Let $food_{max}$ be the maximum food items, t_{test} be the test time, v_{avg} be the average linear velocity of the kilobots, n be the number of kilobots, fn_{dist} be the shortest (radial) distance between the food and nest regions:

$$food_{max} = \frac{n \cdot v_{avg} \cdot t_{test}}{2 \cdot fn_{dist}} \tag{4.1}$$

We normalise the actual collected food items within the time of the test to give a fitness value. Let $food_{collected}$ be the total collected food items and k be a derating factor. The fitness f of the controller is given by:

$$f = k \cdot \frac{food_{collected}}{food_{max}} \tag{4.2}$$

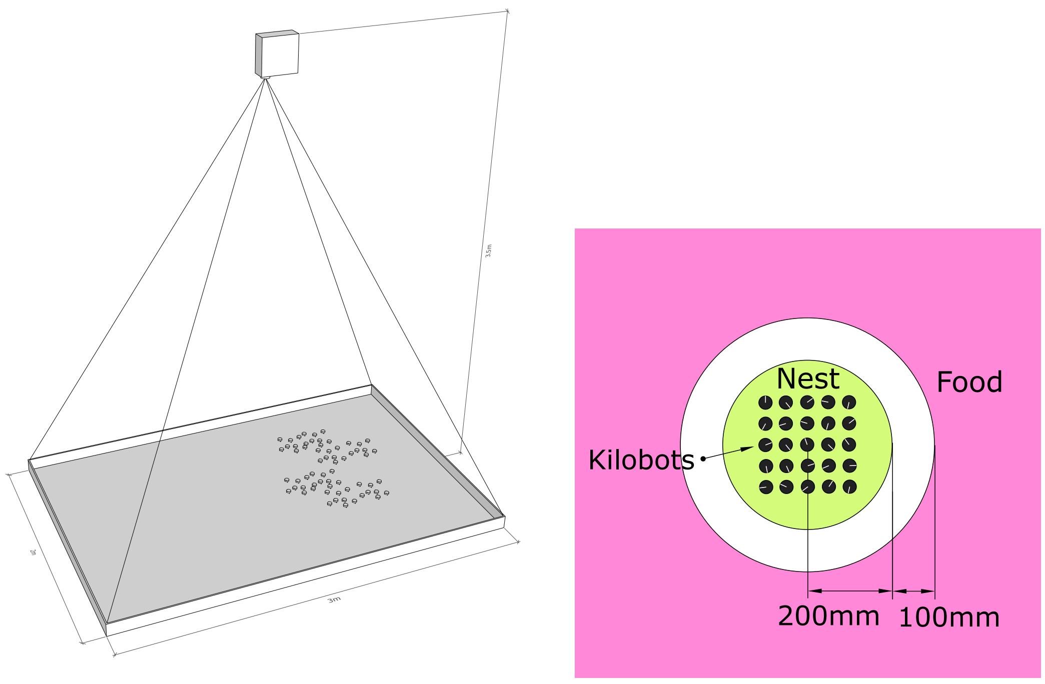


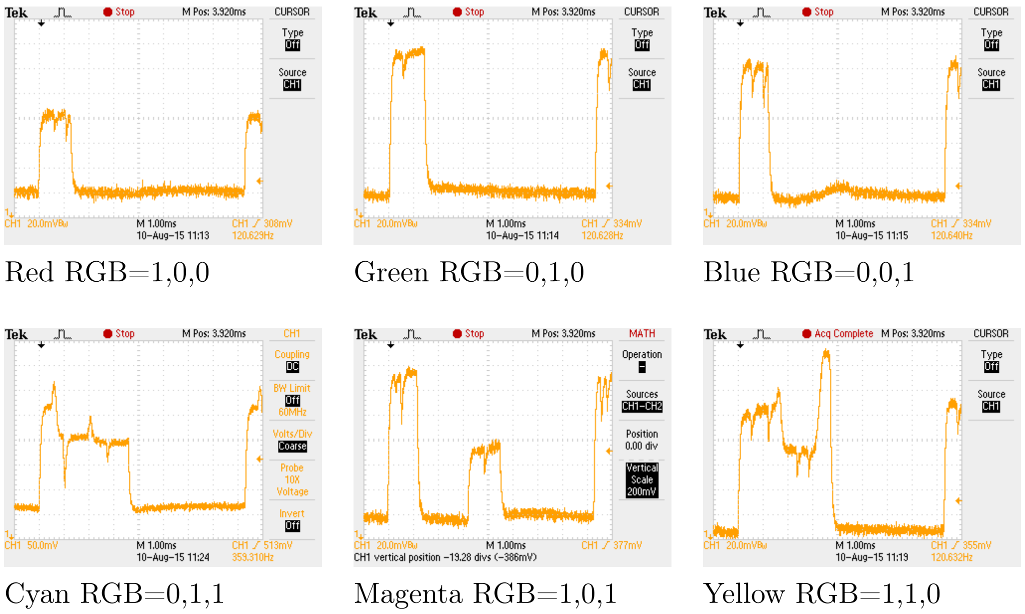
Figure 4.1: Left: Kilobot arena. The arena is a 3m x 2m surface upon which a projector defines the environment with patterns of light. Right: Starting configuration for kilobot foraging experiment. 25 kilobots are placed in a 5x5 grid in the centre of the nest region, with random orientations. Surrounding the nest is a 100mm gap, then outside that is the food region.

The derating factor k is used to exert selection pressure towards smaller behaviour trees to ensure they will fit within the limited RAM resources of the kilobots. It is related to the RAM resource usage fraction r_{usage} (4.4) in the following way: $k = 1.0$ when $r_{usage} < 0.75$ decreasing linearly to 0 when $r_{usage} = 1.0$.

For our experiments, we want to be able to sense whether we are within a particular region (nest or food) of the arena. Regions are delineated within the arena by using different coloured light from a video projector and detected with the upwards-facing phototransistor of the kilobots. In order to create a robust region sensing capability with a monochrome sensor, we exploited some particular characteristics of low cost Digital Light Processor (DLP) projectors [Hutchison, 2005]. The projector we have is a Benq-MS619ST.

The optical path of these type of projectors consists of a white light source, an optical modulator array, and a spinning colour wheel with multiple segments. Different full intensity primary and secondary colours produce different, quite distinct brightness modulation patterns in the light, which our eyes integrate but which we can detect easily with a series of samples from the photodetector. We measured the modulation patterns of each of the full brightness primary and secondary colours, this is illustrated in Figure 4.2. In our case, the projector had a wheel spinning at 120 Hz.

Within each 8.3 ms period, primary colours were represented with a single pulse of about 1.2 ms, cyan and yellow with a pulse of 3.5 ms, and magenta with two pulses of 1.2 ms separated by a gap of 2 ms, giving, including black, four distinguishable patterns. We take 16 brightness samples from the phototransistor at 520 us intervals, covering one complete cycle, and classify the pattern.


Red RGB=1,0,0 Green RGB=0,1,0 Blue RGB=0,0,1

Cyan RGB=0,1,1 Magenta RGB=1,0,1 Yellow RGB=1,1,0

Figure 4.2: Photodetector waveforms captured with Kilobot positioned under DLP projector showing primary and secondary colours. Timebase is 1 ms/div

The IR communication system between the kilobots has a range of about 100 mm. Twice a second, the kilobot system software sends any available outgoing message, retrying if the sending attempt collided with another sender. A kilobot receiving a valid message calls a user specified function to handle it. The message has a payload of nine bytes, and associated with the message is signal strength information to enable the distance from the sender to be calculated.

4.3 Controller

In order to control a robot with a behaviour tree, we need to define the interface between the behaviour tree action nodes and the robot, and the action nodes that act on the interface. This interface is known as the *blackboard*. Here there is a trade-off between the capabilities that we choose to hard code and those that we hope will evolve in the BT. We do not design the behaviour of the swarm but we do make assumptions about what kind of sensory capabilities might be useful for the evolutionary algorithm. This is often implicit in swarm robotics. The kilobot has no in-built directional sensors, like the range-and-bearing sensors that are common in swarm robotics experiments, so we synthesise collective sensing such that it is possible for a robot to tell if it is moving towards or away from the food or nest. We also give the capability of sensing the environment and the local density of kilobots, and of sending and receiving signals to other kilobots.

This relatively rich set of hardwired capabilities is outlined in Table 4.1. There are ten blackboard entries, **motors** maps to the motion control commands of the kilolib API, The **send_signal** and **receive_signal** entries allow for communication

Table 4.1: Behaviour tree blackboard, defining interface between the behaviour tree and the robot.

Index	Name	Access	Description
0	$motors$	W	0=off, 1=left turn, 2=right turn, 3=forward
1	$scratchpad$	RW	Arbitrary state storage
2	$send\_signal$	RW	>0.5 = Send a signal flag
3	$received\_signal$	R	1=A signal flag has been received
4	$detected\_food$	R	1=Light sensor showing food region
5	$carrying\_food$	R	1=Carrying food
6	$density$	R	Density of kilobots in local region
7	$\Delta density$	R	Change in density per update cycle
8	$\Delta dist_{food}$	R	Change in distance to food per update cycle
9	$\Delta dist_{nest}$	R	Change in distance to nest per update cycle

between kilobots initiated within the BT; **send_signal** is writeable from the BT. When the value is greater than 0.5, it is considered true, and a signal flag will be set in the stream of outgoing message packets. The **receive_signal** entry will be set to 1 if any message packets were received over the previous update cycle that had their signal flag set, otherwise it will remain zero. The **scratchpad** can be read and written, and has no defined meaning, it makes available some form of memory for the evolution of the BT to exploit. **Detected_food** is read-only, and is 1 if the environment sensing shows that the kilobot is in the food region, and zero otherwise, and **carrying_food** denotes whether the kilobot is considered to be carrying a food item. This entry is set to 1 if the kilobot enters the food region, and cleared to zero if the kilobot enters the nest region.

The remaining four entries are all metrics derived from the incoming stream of messages and their associated distance measurements. The kilobot *kilolib* API returns distances in mm but these blackboard entries are expressed in terms of metres. *Density* and $\Delta density$ are measures of the local population density and how it is changing. Each kilobot has a unique ID, which is embedded in its outgoing message packets. By tracking the number of unique IDs and the distances associated with messages from them, we can estimate the local density. Let $UID_{received}$ be the set of unique IDs received in the last update cycle, $dist_i$ be the distance in mm associated with the unique ID, the raw local density in $kilobots \cdot m^{-2}$ in an update cycle d_{raw}

is given by:

$$d_{raw} = \sum_{i \in UID_{received}} \frac{1}{\pi (dist_i/1000)^2} \tag{4.3}$$

This value is filtered with a moving average over $w = 5$ update cycles[1] to give $density(t)$ at update cycle t and $\Delta density(t) = density(t) - density(t-1)$.

The two distance metrics $\Delta dist_{food}$ and $\Delta dist_{nest}$ are calculated by tracking the minimum communication hops [Hauert *et al.* , 2008] needed to reach the respective region, illustrated in Fig. 4.3. For both food and nest, within the message packet are two fields, a hop count and an accumulated distance. The hop count is the minimum number of message hops to reach either the food or the nest region. The accumulated distance is the total length of those hops. Kilobots receiving messages select the lowest hop count, increment it and forward it and the new accumulated distance in the outgoing message stream. If no messages are received, we default to a distance of 0 m if in a food or nest region, or 0.5 m if not in a region. At every

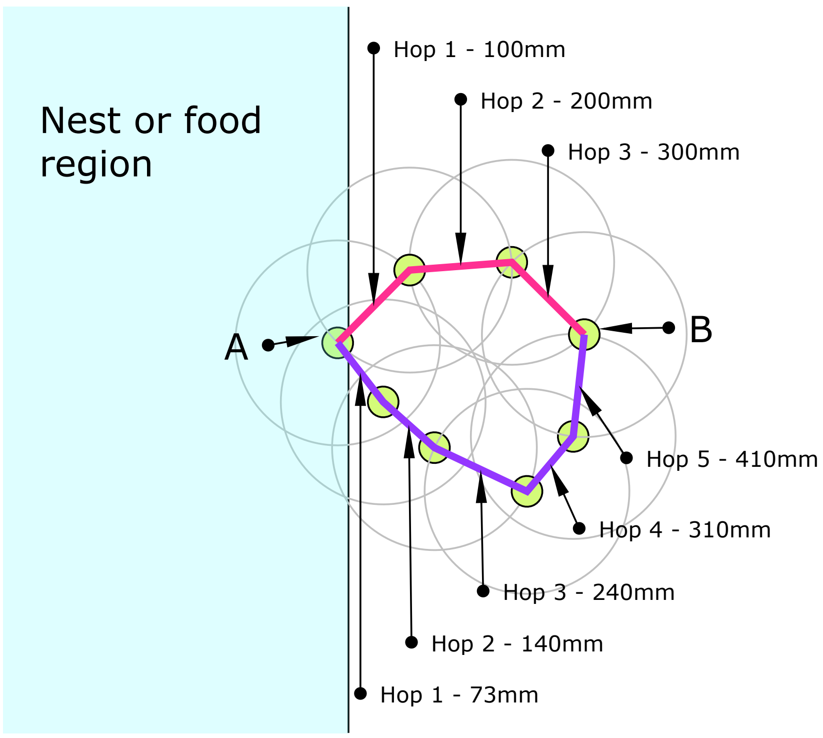


Figure 4.3: Calculation of distance metrics. Kilobot ‘A’ is in a food or nest region, kilobot ‘B’ is connected to ‘A’ via two routes. Grey circles denote maximum communications radius. ‘B’ selects the message from the top route because the hop count is lowest, giving an accumulated distance along hops to the region of 300mm.

update cycle, we calculate two raw distance measures $dist_{food\_raw}$ and $dist_{nest\_raw}$. These are then filtered with a moving average in the same way as the *density* value.

The design of the behaviour tree architecture for this experiment deliberately uses

[1]Chosen in simulation as a reasonable compromise between responsiveness and stability

only the memory forms of the select and sequence nodes, making this a Type 1 (blocking) behaviour tree. This was done for several reasons. Firstly, it makes implementation easier; as noted in Section 3.1.11, there is no need to handle orphaned *running* nodes, so the two-phase update process, together with additional node state, does not apply. Evaluation can be completed with a simple recursive descent of the tree. Secondly, none of the action nodes are long-running, writing to the motors takes a single *tick*, the others are instant. though is possible to create a long-running subtree, with the use of the `repeat` node. We anticipated that, due to the relatively slow movement of the kilobots, the non-reactive nature of the Type 1 behaviour tree would not be an issue. Thirdly, since this was an experiment to demonstrate the feasability of evolving behaviour trees for a swarm robotics problem, it seemed sensible to start with the simplest implementation.

The behaviour tree nodes we implement are outlined in Table 4.2. Nodes are divided into two types; composition and action. Composition nodes are always inner nodes of the tree and combine or modify the results of subtrees in various ways. Action nodes are always leaf nodes and interface with the blackboard. Every update cycle, occurring at 2 Hz, the root node of the tree is sent the *tick* event. Each node handles the *tick* according to its function and returns *success*, *failure*, or *running*. The propagation of *tick* events down the tree and the return of the result to the root happen every cycle.

We use composition nodes `seqm`, `selm`, and `probm`, which can have either 2, 3, or 4 children. The nodes `seqm` and `selm` are as described earlier, the node `probm` is introduced here as a way of introducing randomness by weighted probabilistic selection of a child node.

On receiving a *tick* they process their child nodes in the following way: `seqm` will send *tick* to each child in turn until one returns *failure* or all children have been *tick*ed, returning *failure* or *success* respectively, `selm` will send *tick* to each child in turn until one returns *success* or all children have been *tick*ed, returning *success* or *failure* respectively, `probm` will probabilistically select one child node to send *tick* to and return what the child returns. They all have memory, that is, if a child node returns *running* the parent node will also return *running*, and the next *tick* event will start from that child node rather than the beginning of the list of child nodes. The `repeat, successd, failured` nodes have a single child. **repeat** sends up to a constant number of *tick*s to its child for as long as the child returns *success*, **successd** and **failured** send *tick* to their child and then always return *success* or *failure* respectively.

The action nodes are leaf nodes and interface with the blackboard, described in Table 6.3. `ml`, `mr`, and `mf` activate the kilobot motors to turn left, right, or move forward, returning *running* for one cycle, then *success*. The various `if` nodes compare blackboard entries with each other or with a constant, and the `set` node writes a constant to a blackboard entry.

Table 4.2: Behaviour tree nodes. $Ch \equiv children$, $S \equiv succeeded$, $F \equiv failed$, $R \equiv running$, $N \equiv num\ children$, $I \equiv repeat\ iterations$, $r \equiv randomly\ selected\ child$, $t \equiv ticks$, $v, w \equiv blackboard\ entry$, $k \equiv contant$. Notation from Marzinotto *et al.* [2014]. Size is the size of the node in bytes within the controller representation running on the kilobots.

Node	Size	*success* if	*failure* if	*running* if	Description
Composition					
`seqm2,3,4`	7,9,11	$N\ Ch\ S$	$1\ Ch\ F$	$1\ Ch\ R$	Sequence, *tick* until *failure*
`selm2,3,4`	7,9,11	$1\ Ch\ S$	$N\ Ch\ F$	$1\ Ch\ R$	Selection, *tick* until *success*
`probm2,3,4`	11,17,23	$Ch_r\ S$	$Ch_r\ F$	$Ch_r\ R$	Probabilistic choice
`repeat`	6	$I\ Ch\ S$	$1\ Ch\ F$	$Ch\ R$	Repeat subtree I times
`successd`	4	$Ch\ \bar{R}$	*never*	$Ch\ R$	Always succeed subtree
`failured`	4	*never*	$Ch\ \bar{R}$	$Ch\ R$	Always fail subtree
Action					
`mf`	2	$t = 1$	*never*	$t = 0$	Move forward for 1 *tick*
`ml`	2	$t = 1$	*never*	$t = 0$	Turn left for 1 *tick*
`mr`	2	$t = 1$	*never*	$t = 0$	Turn right for 1 *tick*
`ifltvar`	4	$v_1 < v_2$	$v_1 \geq v_2$	*never*	If $v_1 < v_2$
`ifgevar`	4	$v_1 \geq v_2$	$v_1 < v_2$	*never*	If $v_1 \geq v_2$
`ifltcon`	7	$v < k$	$v \geq k$	*never*	If $v < k$
`ifgecon`	7	$v \geq k$	$v < k$	*never*	If $v \geq k$
`set`	7	*always*	*never*	*never*	Set $w \leftarrow k$
`successl`	2	*always*	*never*	*never*	Always succeed
`failurel`	2	*never*	*always*	*never*	Always fail

The controller runs an update cycle at 2Hz. Message handling takes place asynchronously, and a message is always sent at each sending opportunity. Environmental sensing takes place at 8Hz, synchronously with the update cycle, with a median filter over 7 samples to remove noise. Each cycle, the following steps take place: 1) New blackboard values are calculated based on the messages received and the environment. 2) The behaviour tree is *tick*ed, possibly reading and writing the blackboard. 3) The movement motors are activated, and the message signal flag set according to the blackboard values.

Implementation of the behaviour tree for execution on the kilobot requires careful use of resources; the processor has only 2kbytes RAM, which must hold all variables, the heap, and the stack. The tree structure is directly represented in memory, with each node being a structure with type, state, and additional type-dependent data such as pointers to children. Execution of the behaviour tree involves a recursive descent following node child pointers and as such, each deeper level uses entries on the stack.

The compiled kilobot code uses about 500 bytes for all non-heap variables. We allocate 1024 bytes to the tree storage, leaving another 500 bytes for the stack and some margin for Interrupt Service Routine stack usage. Each level of tree depth uses 16 bytes of stack. Let tr_{size} be tree storage bytes and tr_{stack} be tree stack usage. The resource usage is given by:

$$r_{usage} = max(\frac{tr_{size}}{1024}, \frac{tr_{stack}}{500}) \tag{4.4}$$

This gives a maximum tree depth of about 30 and a maximum number of about 140 nodes at the average node size.

4.3.1 Evolutionary algorithm and simulator

Behaviour trees are amenable to evolution using Genetic Programming [Koza, 1992] techniques. Using the DEAP library [Fortin *et al.* , 2012] a primitive set of strongly typed nodes were defined to represent behaviour tree nodes and their associated allowable constants. There are several types of constants: `if` and `set` use $k \in [-1.0, 1.0]$, this range was chosen to cover the physically possible blackboard values. Number of `repeat` iterations is $I \in [1..9]$, `if` blackboard index $v_i \in [1..9]$ to cover the readable blackboard entries. `set` blackboard index $w \in [1..2]$ for the writable entries except the motors, which have dedicated nodes to control. For `probm`, probabilities $p \in [0.0, 1.0]$.

Evolution proceeds as follows: The population of n_{pop} is evaluated for fitness by running 10 simulations for each individual, each simulation with a different starting configuration. The starting position is always a 5x5 grid with 50mm spacing in the centre of the nest region, but the orientation is randomly chosen from interval $(-\pi, \pi)$ radians. The simulation runs for 300 simulated seconds and fitness is as Eqn 4.2.

An elite of n_{elite} is transferred unchanged to the next generation. The remainder are chosen by tournament selection with size t_{size}. A tree crossover operator is applied with probability p_{xover} to all pairs of non-elite, then three different mutation operators are applied to the non-elite individuals. Firstly, with probability p_{mutu}, a node in the tree is selected at random and the subtree at that point is replaced with a randomly generated one. Next, with probability p_{muts}, a subtree is chosen randomly and replaced with one of its terminals. Next, with probability p_{mutn} a node is picked at random and replaced with another node with the same argument types. Lastly, with probability p_{mute}, a constant is picked randomly and its value changed. Parameters are shown in Table 4.3. The initial values were chosen firstly with reference to the literature, particularly Poli *et al.* [2008], for guidance, and designer knowledge from previous evolutionary algorithm experiments. Population size was smaller than typical examples in the literature, mostly driven by the need to reduce runtimes. We then conducted a series of trials to tune the values, aiming for rapid increases in fitness. Once a reasonable set of parameters were arrived at, they were left unchanged, although it is likely that further beneficial tuning would be possible.

Table 4.3: Parameters for a single evolutionary run

Parameter	Value	Description
n_{gen}	200	Generations
t_{test}	300	Test length in seconds
n_{pop}	25	Population
n_{elite}	3	Elite
t_{size}	3	Tournament size
p_{xover}	0.8	Crossover probability
p_{mutu}	0.05	Probability of subtree replacement
p_{muts}	0.1	Probability of subtree shrink
p_{mutn}	0.5	Probability of node replacement
p_{mute}	0.5	Probability of constant replacement

We wrote a simple 2D simulator based on the games physics engine Box2D [Catto, 2009]. The physics engine is capable of simulating interactions between simple convex geometric shapes. We model the kilobots as disks sliding on a flat surface with motion modelled using two-wheel kinematics, with forward velocity of $8 \times 10^{-3} ms^{-1}$ and turn velocity of $0.55 rad\, s^{-1}$, based on measurements of 25 kilobots, see Table 4.4. Physical collisions between kilobots, and movement into and out of communication range were handled by Box2D, with an update loop running at 10Hz. Simulator deficiencies that could cause *reality gap* effects were masked using the addition of noise [Jakobi *et al.* , 1995]. Gaussian noise was added to linear ($\sigma = 1 \times 10^{-3} ms^{-1}$) and angular ($\sigma = 0.2 rad\, s^{-1}$) components of motion at every simulator timestep, and each kilobot had a unique fixed linear ($\sigma = 1.3 \times 10^{-3} ms^{-1}$) and angular ($\sigma = 0.06 rad\, s^{-1}$) velocity bias added, to reproduce measured noise performance and variability of real kilobots. Message reception probability was fixed at 0.95. Simulation performance r_{acc}, measured using the methodology described in Jones *et al.* [2015] on an iMac 3.2GHz machine was approximately 8×10^4.

Table 4.4: Kilobot performance in rotational and forward motion. 25 kilobots were calibrated and measured.

Parameter	$\bar{x}$	σ	Units
Forward motion	8.0	1.3	mms^{-1}
Rotate right	0.56	0.073	$rads^{-1}$
Rotate left	0.52	0.062	$rads^{-1}$

Twenty five independent evolutionary runs were conducted, each one using the parameters in Table 4.3. Each individual fitness evaluation was the mean over ten simulations with different starting configurations. A total of 1.1 million simulations were run[2].

The fittest individual across the 25 separate populations was evaluated again for fitness, this time over 200 simulations with different starting configurations. This individual controller was then instantiated uniformly across a swarm of real kilobots, giving a homogenous swarm. The real kilobots were run 20 times with different starting configurations and their fitness measured.

4.4 Results and discussion

The results (Fig. 4.4) show that we have successfully evolved a behaviour tree for use as a swarm robot controller to perform a foraging task. When instantiated in a swarm of real robots, it performs similarly to the simulation, validating the applicability of using this simulator for evolving kilobot swarm controllers. The performance is slightly lower in real life (0.058) compared to the simulated (0.075) performance, this is expected due to *reality gap* [Jakobi *et al.* , 1995] effects. It is worth noting this is still a good outcome, the robots are able to effectively forage.

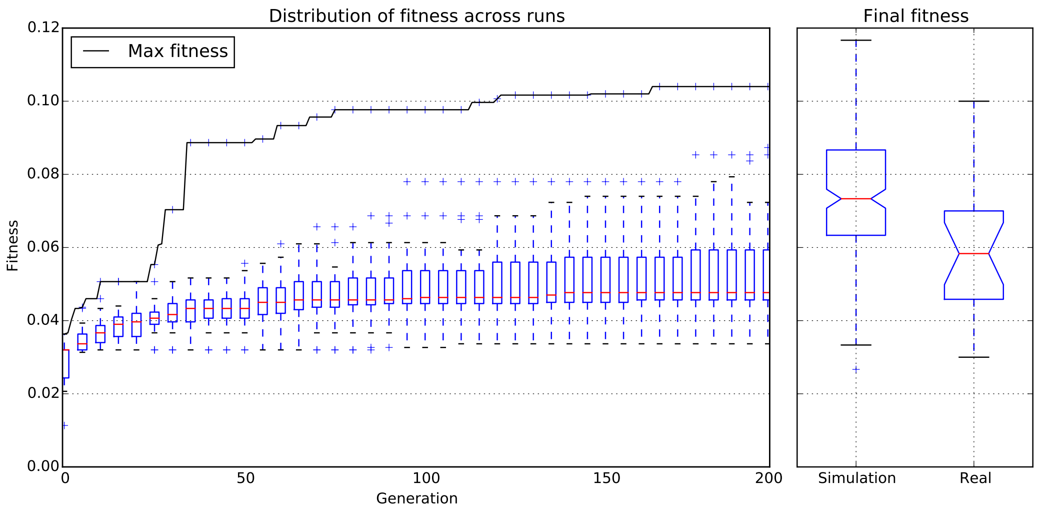


Figure 4.4: Result of evolutionary runs. The left hand graph shows the maximum individual fitness across all 25 independent evolutionary runs, with a box plot every 5 generations to show the distribution. The right hand shows the distribution of fitnesses of the fittest individual, measured over 200 simulation and 20 real runs.

Fitness rises fast to about 0.03 after the first generation. This is due to the fact that an extremely simple controller that does nothing except move forward will still

[2]Due to the elitism policy, three individuals per generation are unchanged and need no fitness evaluation

collect some food; because of the variability of the kilobots, some will move in large arcs that leave the nest, enter the food region and return to the nest. This type of controller is easily discovered by the evolutionary algorithm, confirmed by examining the fittest controller after one generation in the fittest lineage.

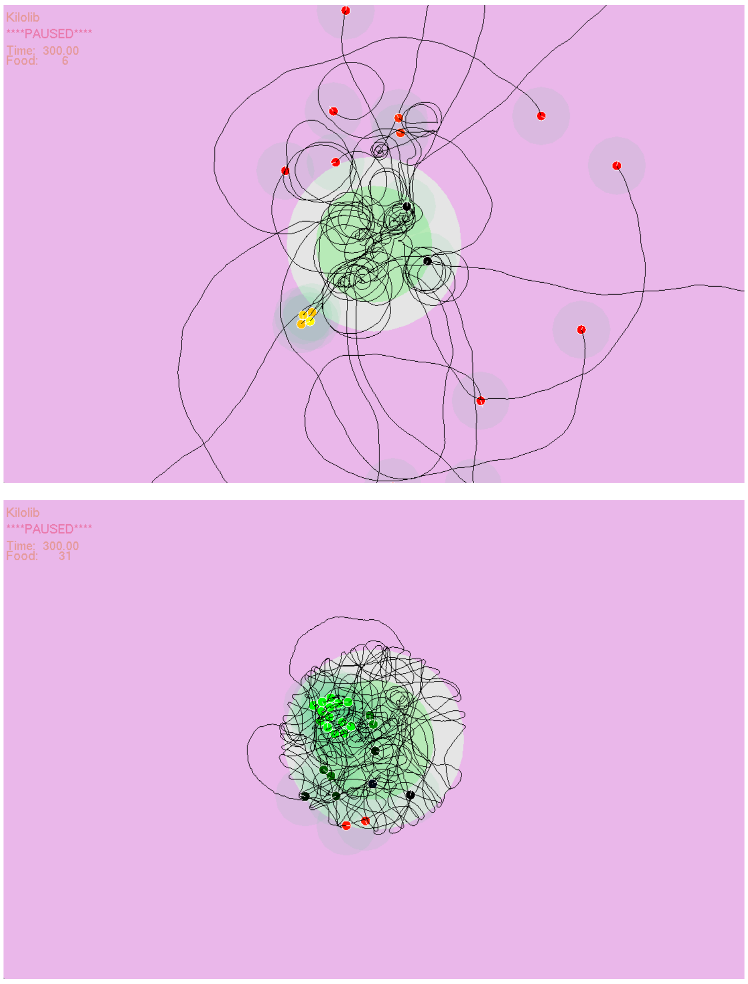


Figure 4.5: Kilobot trails from simulation of the fittest controller in the first generation (top) and the 200th generation (bottom) of the fittest lineage. Small discs are kilobots, coloured red if they are carrying food, and green based on their current density. The group of four kilobots in the lower left of the top image are yellow because they are both carrying food and in close proximity. Black trails show previous motion.

Some example kilobot paths in simulation are shown in Fig. 4.5. The wide looping trails through the food region in the first generation controller are characteristic of a controller that just runs the motors in the forward direction continually, while the final generation controller exhibits much more intentional movement, with the swarm remaining mostly in the nest and the food/nest gap.

It is noteworthy that the fittest of the 25 lineages is much fitter than the median, and the innovation seems to have been discovered around generation 30. This suggests that the evolutionary algorithm is possibly not exploring the fitness landscape very effectively, otherwise we would expect evolution to discover similar behavioural innovations within other lineages.

Table 4.5: Individuals from top five lineages and their usage of the blackboard and behaviour tree constructs. All individuals use at least the forward and one other of the motor action nodes. Usage is after redundant or unreachable nodes have been removed.

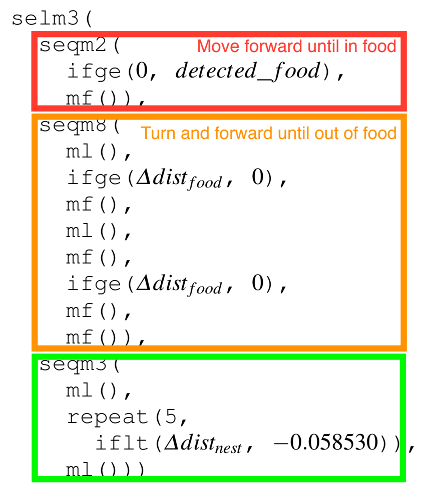
		Blackboard entry									BT Nodes					
Rank	Fitness	1	2	3	4	5	6	7	8	9	`seq`	`sel`	`prob`	`repeat`	`if`	`set`
1	0.104				x				x	x	x	x		x	x	
2	0.0873		x	x	x						x	x	x		x	x
3	0.0853				x	x		x		x	x		x		x	x
4	0.0723	x	x		x	x	x	x	x	x	x	x	x	x	x	x
5	0.0710				x		x		x		x	x	x		x	

We can examine the fittest BT, shown in Fig. 4.6, to gain insights into its workings. First of all, it is interesting to note that not all of the hardwired capabilities are used, only $detected\_food$, $\Delta dist_{food}$, and $\Delta dist_{nest}$. Both *scratchpad* and $send\_signal$ are read but never written, so are equivalent to zero. This is not the case with all the evolved behaviour trees, see Table 4.5 for details of the blackboard usage of the top five fittest trees from different lineages. Between these individuals, every behaviour tree construct and blackboard entry is used. There is no obvious correlation between the features used and the fitness of the individual, perhaps indicating that there are multiple ways to solve this foraging problem.

The overall structure is a three-clause `selm`, the child trees will be *tick*ed in turn until one returns *success*. Consider a single kilobot, with no neighbours in communication with it. The first clause causes the kilobot to move forward as long as it is not in the food region. If it enters the food, the second clause comes into play, performing a series of left turns and forward movements until it moves out of the food region. Behaviour will then revert to the first clause and it will move forward again, likely hitting the nest region. We can see that this will produce reasonable individual foraging behaviour, and this pattern is visible in the lower trail plot in Fig 4.5. The foraging behaviour will be enhanced in the presence of neighbours, since in this case the second clause will promote movement away from food generally, rather than just on the food region boundary. Finally, if the kilobot is executing the second clause, manages to leave the food then re-enters it, or moves towards it in the presence of neighbours, the third clause is triggered, which produces some additional left turning. The `repeat` sub-clause will fail on the first iteration since it is not physically possible for the kilobot to move 59 mm in one update cycle of half a second.

This evolved behaviour tree is sufficiently small that it can be analysed by hand relatively easily. It may be that greater foraging performance could be obtained by removing the selective pressure to small trees, and a larger tree would be harder to analyse. But, in contrast to evolved neural networks, which are a black box for which

```
selm3(
  seqm2(
    ifgevar(send_signal, detected_food),
    mf()),
  seqm3(
    seqm3(
      ml(),
      ifgevar(Δdist_food, scratchpad),
      mf()),
    ifgevar(Δdist_food, scratchpad),
    seqm2(
      seqm3(
        seqm3(
          ml(),
          ifgevar(Δdist_nest, Δdist_nest),
          mf(),
        ifgevar(Δdist_food, send_signal),
        mf()),
      mf()))),
  seqm3(
    ml(),
    repeat(5,
      ifltcon(Δdist_nest, −0.058530)),
    ml()))
```


Move forward until in food
Turn and forward until out of food

```
selm3(
 seqm2(
   ifge(0, detected_food),
   mf()),
 seqm8(
   ml(),
   ifge(Δdist_food, 0),
   mf(),
   ml(),
   mf(),
   ifge(Δdist_food, 0),
   mf(),
   mf()),
 seqm3(
   ml(),
   repeat(5,
     iflt(Δdist_nest, −0.058530)),
   ml()))
```

Figure 4.6: Fittest behaviour tree. Left shows the code as evolved. Right shows the code with redundant lines removed by hand, the `seqm` nodes condensed, and conditionals simplified. Boxes highlight the three functional clauses.

there are no adequate tools to predict behaviour apart from direct testing [Nelson *et al.* , 2009], it is possible at least in principle to analyse any behaviour tree, in the same way it is possible to analyse any computer program. The behaviour of each sub-tree can be analysed in isolation, descending until the size of the sub-tree is tractable, and automatic tools can simplify and prune branches which will never be entered, or will always do nothing.

Understanding the behaviour of an evolved BT does not mean that it becomes possible to predict the emergent swarm behaviour that the interaction between the kilobots will produce. However, the more easily we can understand the controller, the more likely we are to gain insights into the problem of predicting these higher-level behaviours.

4.5 Conclusions

In order to demonstrate that it is indeed possible to use evolutionary methods to discover successful behaviour tree controllers for a swarm robotics problem, we describe a proof-of-concept experiment with a swarm of kilobots in which we design a behaviour tree architecture and robot simulator, use off-line evolution of behaviour trees to find a fit controller for a swarm robotics foraging task, then show that the controller works on a real swarm of robots, transferring effectively. Evolved controllers for swarm robotics are generally hard to understand. We show that using behaviour trees, it is possible to analyse by hand the fittest evolved controller for insight into the discovered foraging algorithm.

This proof-of-concept experiment demonstrates that behaviour trees are a viable controller architecture for evolved swarm robotics controllers. We now build on this work to take a more considered approach to the design of a behaviour tree architecture for the Xpuck robots, a much more capable robot swarm, both computationally and in terms of the available sensors, with the goal of moving towards an in-swarm evolutionary process for the generation of BT controllers.

Chapter 5

Xpuck design

This chapter contains work that was published as *A Two Teraflop Swarm* in Frontiers of Robotics and AI [Jones *et al.* , 2018].

In this chapter, we describe the design of a new swarm robotics platform that makes use of recently available and cheap high-performance computing capability to augment the widely used e-puck robot, which many labs will already have available. We have designed it to have higher computational capability than any other swarm platforms, see Table 5.1, and to have a battery life at least as good as other solutions, while minimising costs to allow the building of large swarms. By providing swarms with high computational power, we can move towards fully autonomous swarms, not tied to external infrastructure, that can be deployed in the wild.

The total cost of building the complete swarm of 16 Xpuck robots was less than £2300, so each robot was less than £150 on top of the cost of the e-puck base, and the design is straightforward for a university technician to reproduce. In order to facilitate further research we have make the design open source, please see Appendix A.1 for further details.

As validation of our design, we provide power and performance characterisation. Design challenges, trade-offs and solutions are discussed, including battery and power supply, data exchange with the e-puck, providing full access to the e-puck camera, and changes to the Linux kernel to support hard realtime operation. The wider system-as-a-whole is described, integrating a Vicon tracking system, support for virtual sensing in addition to the already available e-puck senses, and experiment management and data logging. We demonstrate the computational capability of the platform in two ways. Firstly we evaluate a fiducial tracking image processing application using the e-puck camera that would not be computationally possible on the standard e-puck. Secondly, and to lay the groundwork for future experiments, we implement a fast parallel physics-based robot simulator running on the GPU of

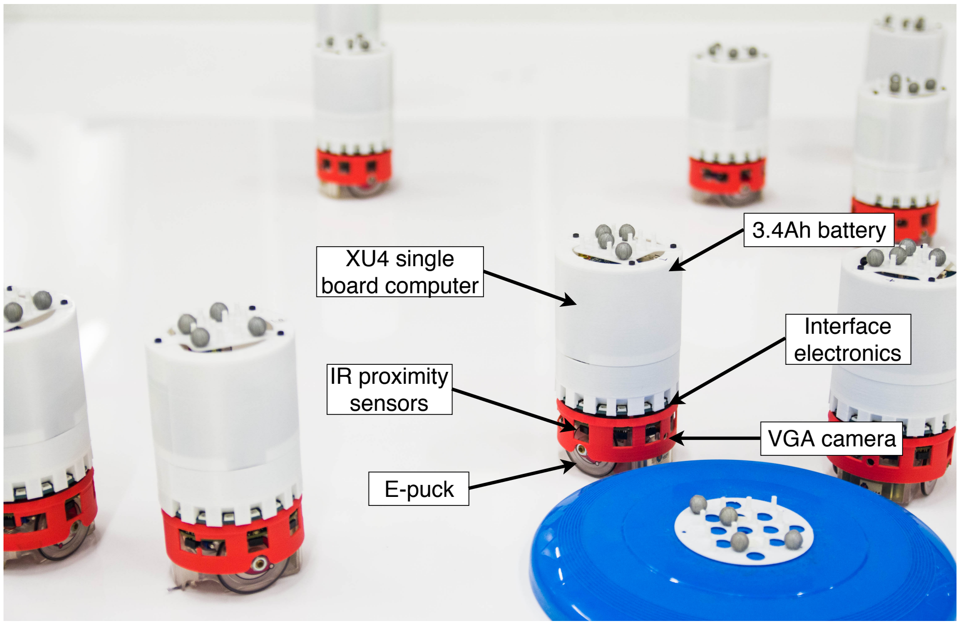


Figure 5.1: Several Xpucks in the arena, together with a blue frisbee used for foraging experiments. Each Xpuck has an e-puck base, with interface electronics, an XU4 single-board computer, and additional battery enclosed in the white cylinder.

the Xpuck, and use this within a distributed island-model evolutionary system to discover swarm controllers. Figure 5.1 shows several Xpucks in the experimental arena, pushing a blue frisbee.

5.1 Xpuck electronics design

In this section we set out our system requirements. We outline potential computing modules. We characterise the power/performance tradeoffs of our chosen compute module and then discuss the design and implementation of the Xpuck hardware and associated system infrastructure to enable running experiments. We then detail the design and implementation of a fast physics-based robot simulator specifically tailored to the Xpuck to enable on-board evolutionary algorithms. We also describe two demonstrations of the Xpuck computational capabilities, a fiducial tracking application that could not be run on a standard e-puck, and an island model evolutionary algorithm running on multiple Xpucks.

In order to run experiments building on the literature, we decided that, in addition to much higher processing power, the Xpuck must meet or exceed the capabilities provided by the existing e-puck robots with additional processing boards. The e-puck is a two-wheel stepper motor driven robot. Its sensors comprise a ring of IR proximity sensors around its periphery, a three-axis accelerometer, three microphones, and a VGA video camera.

As with the Linux Extension Board (LEB), introduced by Liu & Winfield [2011], we require a battery life of at least 1.5 hours and full access to the e-puck's IR proximity and accelerometer sensors, and control of the stepper motors and LEDs. In addition we require that the VGA camera can stream full frame at >10 FPS. The Xpuck must run a full standard Linux, able to support ROS [Quigley *et al.* , 2009]. It must have WiFi connectivity. GPGPU capabilities must be made available through a standard API such as OpenCL or CUDA [Khronos OpenCL Working Group *et al.* , 2010; Nvidia, 2007]. We also want multicolour LED signalling capability for future visual communication experiments [Floreano *et al.* , 2007; Mitri *et al.* , 2009]. Since many labs already have multiple e-puck robots, we wished to minimise the additional cost of the Xpuck to facilitate the construction of relatively large swarms of robots. With this in mind, we chose a target budget per Xpuck of £150.

We decided at an early stage not to add hardware for short range communication such as the Range-and-Bearing board [Gutiérrez *et al.* , 2009b]. Instead, the system-as-a-whole would comprise multiple Xpucks, each running ROS under Linux, with the already existing Vicon motion tracking system at the laboratory, together with software infrastructure to create a virtual senses including range and bearing, allowing detailed logging and control over the communication range and noise levels.

In addition, we require that the processing power be generally accessible, if there are multiple CPU cores, they should all be useable simultaneously. If there is a GPU with a notionally large processing capacity, this must be available though some general purpose API such as OpenCL or CUDA.

5.1.1 Survey of available platforms

Given the requirements, Table 5.1 sets out some of the current swarm platforms and potential modules that could be used to enhance the e-puck.

There are a number of interesting devices, but unfortunately there are very few that are commercially available at a budget suitable to satisfy the cost requirement of £150. Within these cost constraints, of the two Samsung Exynos 5 Octa based devices, the Hardkernel XU4 and the Samsung Artik 1020, only the XU4 was more widely available at the time of design. The Artik module became generally available in early 2017 and would be interesting for future work because of its small form-factor. There are other small form-factor low-cost modules such as the Raspberry Pi Zero, as used in the Pi-puck [Millard *et al.* , 2017], but none that provide standard API access to GPGPU capability. For these reasons, we chose to base the Xpuck on the Hardkernel Odroid XU4 single board computer.

5.1.2 High performance computing

The Hardkernel Odroid XU4 is a small Single Board Computer (SBC) based around the Samsung Exynos 5422 System-on-Chip (SoC). It has 2 GBytes of RAM, mass

Table 5.1: Current and potential swarm platforms

Product	SoC or microcontroller	GFLOPS (fp32)	RAM (bytes)	Price (£)
		Robot platforms		
Kilobot	Atmel atmega328p	0.0008[a]	2K	15
e-puck	dsPIC	0.0015[a]	8K	650
r-one	TI Stellaris LM3S8962	0.005	64K	165
Linux Extension Board	Atmel AT91SAM9260	0.02[a]	64M	80[i]
Swarmbots	Intel Xscale	0.04[a]	64M	not known
GCTronic Gumstick	TI AM3703	1.2	512M	600[i]
Khepera IV	TI OMAP3730	1.2	512M	2000
Pi-puck	Broadcom BCM2835	1.4[b]	512M	110[i]
Pheeno	Broadcom BCM2836	7.2[c]	1G	205
Xpuck	Samsung Exynos 5 Octa (5422)	36+122[d]	2G	135
		Single Board Computers		
Hardkernel XU4	Samsung Exynos 5 Octa (5422)	36+122[d]	2G	70
Samsung Artik 1020	Exynos 5 Octa[e]	36+122[d]	2G	98
Wandboard IMX6Q	NXP i.MX6 Quad	25[f]	2G	120
Intrinsyc Open-Q820SOM	Qualcomm Snapdragon 820	250[g]	3G	250
Nvidia Jetson TX1	Nvidia Tegra 210	512[h]	2G	290

[a] Integer only, assumes 10 integer instructions per floating point operation
[b] VMLA x 0.7GHz. VideoCore IV GPU has no OpenCL support
[c] VMLA x 4 x 0.9GHz. VideoCore IV GPU has no OpenCL support
[d] CPUs A7 1.4GHz, A15 0.8GHz + ARM Mali-T628MP6 GPU, 4 vector multiplies, 4 vector adds, 1 scalar multiply, 1 scalar add, 1 dot product per cycle, 6 cores, each with 2 arithmetic pipelines at 600MHz. OpenCL 1.2 full profile
[e] Assumption. The product literature doesn't state the SoC but Samsung only used the Mali-T628MP6 in the Exynos 5 Octa family
[f] Vivante GC2000 GPU only, 4 vector multiplies, 4 vector adds, 4 cores at 794MHz, OpenCL 1.1 embedded profile
[g] Very little open information, https://en.wikipedia.org/wiki/Adreno states 498.5 at 624MHz but assumed to be fp16 rather than fp32. OpenCL 2.0
[h] According to AnandTech, Ho & Smith [2015]
[i] In addition to e-puck cost

storage on microSD card, ethernet and Universal Serial Bus (USB) interfaces, and connectors exposing many General Purpose IO (GPIO) pins with multiple functions.

The SoC contains eight ARM CPU cores in a big.LITTLE[1] formation, that is, two clusters, one of four small low power A7 cores, and one of four high performance A15 cores. The system concept envisages the small A7 cores being used for regular but undemanding housekeeping tasks, and the higher performing A15 cores being used

[1]https://developer.arm.com/technologies/big-little

Table 5.2: Hardkernel Odroid XU4 specifications

Spec	Details
SoC	Samsung Exynos 5 Octa (5422)
CPU organisation	big.LITTLE 4+4
CPU big	4x ARM Cortex A15 2GHz 4x 32K L1I, 4x 32K L1D, shared 2M L2 25.6 GFLOPS[a]
CPU little	4x ARM Cortex A7 1.4GHz 4x 32K L1I, 4x 32K L1D, shared 512K L2 11.2 GFLOPS[b]
GPU	ARM Mali T628MP6 600MHz 122 GFLOPS[c]
Memory	2Gbytes LPDDR3 933MHz PoP
Memory bandwidth	14.9 GBytes/s
Idle power	2 W
Maximum power	21W

[a] 4-wide SP NEONv2 FMA x 4 x 800 MHz
[b] VMLA x 4 x 1.4 GHz
[c] 4 vector multiply, 4 vector add, 1 scalar multiply, 1 scalar add, 1 dot product per cycle x 2 pipelines x 6 cores x 600 MHz

when the computational requirements exceed that of the A7 cores, at the expense of greater power consumption. It also contains an ARM Mali T628-MP6 GPU, which supports OpenCL 1.2 Main Profile, allowing the relatively easy use of the GPU for GPGPU computation. Some important specifications are detailed in Table 5.2.

The Linux kernel supplied by Hardkernel supports full Heterogeneous MultiProcessing (HMP) scheduling across all eight cores, with the frequencies of the two clusters being varied according to the current process mix and load, the specified minimum and maximum frequencies for each cluster, and the kernel *governor* policy[2]. It was evident from manually changing the CPU frequencies during initial investigation that there was little subjective performance boost from using the highest frequencies, but a large increase in power consumption.

5.1.3 Operating point tuning

Computational efficiency is an important metric, directly affecting the battery life. Initial tests showed that setting the maximum frequencies to the highest allowed by the hardware (A15 - 2 GHz, A7 - 1.4 GHz) and running a computationally heavy load caused the power consumption to exceed 15 W. In order to characterise the system and find an efficient operating point, we chose to perform benchmarking with a large single precision matrix multiplication using the standard BLAS API function SGEMM. This computes $C = \alpha AB + \beta C$, which performs $2N^2(N+1)$ operations for an $N \times N$ matrix. Good performance requires both high real floating point performance and good memory bandwidth. The OpenBLAS libraries [Xianyi *et al.* , 2012] provide optimised routines capable of running on multiprocessor systems and can

[2]Essentially how fast clock frequency will be varied to meet changing CPU load.

utilise all available processor cores. ARM provide useful application notes on implementing an efficient single precision GEMM on the GPU [Gronqvist & Lokhmotov, 2014].

Power consumption was measured for the XU4 board as a whole, using an INA231 power monitoring chip [Texas Instruments, 2013] with a 20 mΩ shunt resistor in series with the 5 V supply . A cooling fan attached to the SoC was run continuously from a separate power supply to prevent the fan control thermal regulation from affecting the power readings. Clock frequency for the A7 and A15 clusters of the Exynos 5422 were varied in 200 MHz steps from 200 MHz to 1.4 GHz for the A7, and from 200 MHz to 2 GHz for the A15 clusters respectively. At each step, a 1024 by 1024 SGEMM was performed continuously and timed for at least 5 seconds while the power values were measured to give Floating Point Operations per Second (FLOPS) and FLOPS/W. All points in the array were successfully measured except for the highest frequency in both clusters; 1.4 GHz for A7 and 2 GHz for A15, which caused the SoC temperature to exceed 95 °C during the 5 second window, even with the cooling fan running, resulting in the automatic clock throttling of the system to prevent physical damage.

The results confirm that increasing CPU clock frequencies, particularly of the A15 cluster, produced little performance gain but much higher power consumption. Figure 5.2 shows that the most efficient operating point of 1.95 GFLOPS/W and 9.1 GFLOPS occurs at the maximum A7 cluster frequency of 1.4 GHz, and the relatively low A15 cluster frequency of 800 MHz. Increasing the A15 frequency to the maximum achievable of 1.8 GHz results in a 6% increase in performance to 9.7 GFLOPS but at the cost of 40% drop in efficiency to 1.21 GFLOPS/W. Because of this dramatic drop in efficiency, we fix the maximum A15 frequency to 800 MHz.

As with the CPU measurement, GPU power consumption was measured for the system as a whole, in the same way. The clock frequency of the GPU was set to each of the allowed frequencies of 177, 266, 350, 420, 480, 543 and 600 MHz and an OpenCL kernel implementing a cache efficient SGEMM was repeatedly run on both the OpenCL devices. Figure 5.2 shows that efficiency only declines slightly from the peak at around 480 MHz to 2.24 GFLOPS/W and 17.7 GFLOPS at the maximum 600 MHz. For this reason, we left the maximum allowed frequency of the GPU unchanged.

Note that the GFLOPS figures in these tests are much lower than the theoretical peak values in Table 5.2 because the SGEMM task is mostly memory bound.

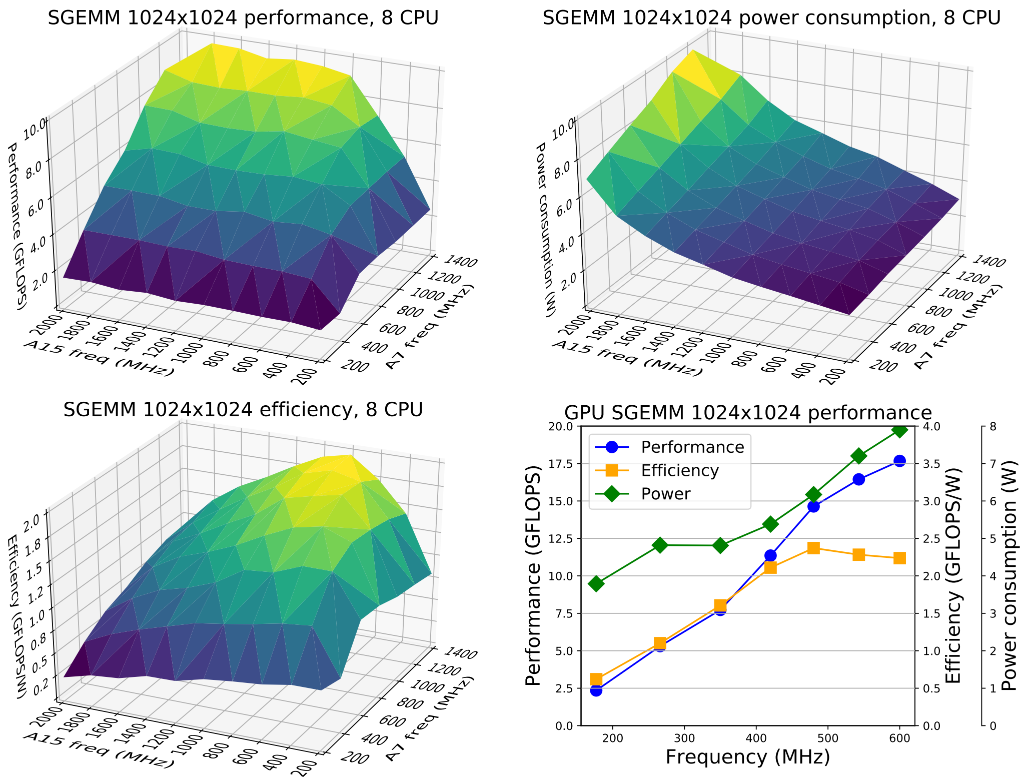


Figure 5.2: Performance, power consumption, and efficiency of the CPUs and GPU while continuously running a 1024 x 1024 single precision matrix multiplication. Highest efficiency for the CPU clusters is with the maximum A7 frequency of 1.4 GHz but a relatively low A15 frequency of 800 MHz. The GPU efficiency stays relatively flat above 480 MHz.

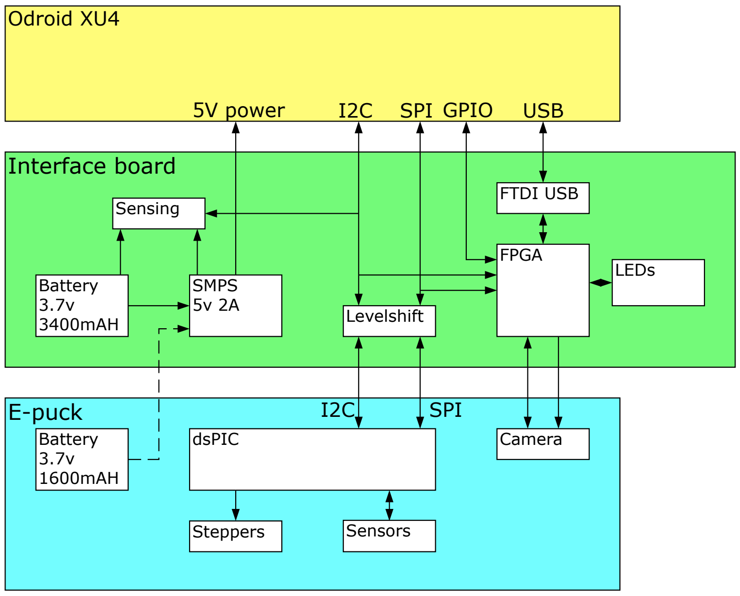


Figure 5.3: Block diagram showing the functionality of the interface board. The yellow box at the top is the XU4 single board computer, communicating over I^2C, SPI, and USB interfaces with the interface board in green. This performs voltage level shifting, provides a USB interface to the e-puck camera, and supplies 5 v power to the XU4. The e-puck in blue acts as a slave device to the XU4, running low level proximity sensing and stepper motor control.

5.1.4 Interface board

An interface board was created to provide power to the XU4 single board computer, interface between the XU4 and the e-puck, and provide new multicolour LED signalling. The overall structure is shown in Figure 5.3.

There are three interfaces to the e-puck, all exposed through the expansion connectors; a slow Inter-Integrated Circuit (I^2C) bus that is used for controlling the VGA camera, a fast Serial Peripheral Interface (SPI) bus that is used for exchanging data packets between the XU4 and the e-puck, over which sense and control information flow, and a parallel digital interface to the VGA camera. In each case, the interfaces have 3.3v logic levels.

The XU4 board has a 30 pin expansion connector that exposes a reasonable number of the GPIO pins of the Exynos 5422 SoC, some of which can be configured to be I^2C and SPI interfaces. The XU4 interface logic levels are 1.8 V. A camera interface was not available, and initial investigation showed that it would not be possible to use pure GPIO pins as a parallel data input from the camera due to the high required data rate. We decided to use a USB interface to acquire camera data.

We use visual signalling as a means of communication within swarms. For this purpose we included a ring of fifteen programmable RGB LEDs, known as Neopixels, around the edge of the interface board. Neopixels are relatively recently available digital multicolour RGB LEDs which are controlled with a serial bitstream. They can be daisy chained in very large numbers and each primary colour is controllable to 256 levels.

Power supply

The XU4 requires a 5 V power supply. In order to design the power supply, the following constraints are assumed:

- The XU4 and supporting electronics will be powered from their own battery, separate from the e-puck battery
- The average power consumption will be 5 W
- The peak power consumption will be 10 W

It is immediately clear that the e-puck battery, a single cell Li-Ion type with a capacity of $C = 1600mAh$, would not be able to power the XU4 as well. At a nominal cell voltage of $v_{nom} = 3.7\,\mathrm{V}$, converter efficiency of $r_{eff} = 85\%$ and a nominal power consumption of $P = 5\,\mathrm{W}$, battery life in hours would be at best be given by:

$$\begin{aligned} t &= \frac{v_{nom} \cdot C \cdot r_{eff}}{P} \\ &= \frac{3.7\,\mathrm{V} \cdot 1.6\,\mathrm{Ah} \cdot 0.85}{5\,\mathrm{W}} \\ &= 1 \text{ hour} \end{aligned} \tag{5.1}$$

This is not counting the requirements of the e-puck itself. These estimates are based on battery characteristics in ideal conditions and real world values will be lower. Hence the need for a second battery. In order to get a 1.5 hour endurance, assuming a conservative margin $r_{margin} = 50\%$ to account for real world behaviour, the required battery capacity is given by:

$$\begin{aligned} C &= \frac{t \cdot (1 + r_{margin}) \cdot P}{v_{nom} \cdot r_{eff}} \\ &= \frac{1.5\,\mathrm{V} \cdot 1.5 \cdot 5\,\mathrm{W}}{3.7\,\mathrm{V} \cdot 0.85} \\ &= 3.6 \text{ Ah} \end{aligned} \tag{5.2}$$

Mobile devices are generally designed to work within a power envelope of around 5 W or the case becomes too hot to hold comfortably, see for example Gurrum *et al.* [2012]. We assume that with attention to power usage, it will be possible to keep the average power at this level.

The third constraint was motivated by a survey of the readily available switch-mode power supply solutions for stepping up from 3.7 V single cell lithium to the required 5 V. Devices tended to fall into two types - boost converters that were capable of high currents (>2 A) but with low efficiencies and large-sized inductors due to low operating frequencies, or devices designed for mobile devices which include battery protection and have small sized inductors due to their high efficiency and operating frequency. Of the latter class the highest output current was 2 A, with future higher current devices planned but not yet available. Measurements of the XU4 showed an idle current of 400 mA but very high current spikes, exceeding 3 A during booting. In order to meet the third constraint and enable the use of a high efficiency converter, the kernel was modified to boot using a low clock frequency, reducing boot current to below 1.5 A.

The power supply regulator chosen was the Texas Instruments TPS61232. It is designed for single-cell Li-Ion batteries, has a very high efficiency of over 90%, a high switching frequency of 2MHz resulting in a physically small inductor, and has battery protection with undervoltage lockout.

One aspect of the power supply design that is not immediately obvious is that the battery current is quite high, reaching 4 A as the cut-off discharge limit of 2.5 V is reached. This seriously constrains switching the input power. In fact, physically small switches capable of handling this amount of current are not readily available. For this reason, and to integrate with the e-puck, two Diodes Incorporated AP2401 high side switches were used in parallel to give electronic switching, allowing the use of the e-puck power signal to turn on the XU4 supply. The high current also necessitates careful attention to the resistance budget and undervoltage lockout settings.

In order to monitor battery state and energy, we use two Texas Instruments INA231 power monitoring chips, sensing across 20 mΩ resistors on the battery and XU4 side of the switching regulator. These devices perform automatic current and voltage sensing, averaging and power calculation, and are accessible over an I^2C bus. The Hardkernel modified Linux kernel also targets the older Odroid XU3 board, which included the same power monitor chips, so the driver infrastructure is already present to access them.

We used branded Panasonic NCR18650B batteries, rated at 3400 mAh. Further investigations showed that the resistance between the battery and the switching regulator was around 200 mΩ of which 70 mΩ was the sense resistor and the solid state switch, such that at the undervoltage cutout voltage we had set of 2.8 V, the cell voltage was still over 3.2 V, causing the power supply to cut out when more than half the battery capacity remained unused. We reduced the resistance to 100 mΩ by replacing a PCB jumper with a soldered connection, and soldering and shortening the flying battery lead, rather then connecting to a socket. The undervoltage lockout was also lowered to 2.3 V.

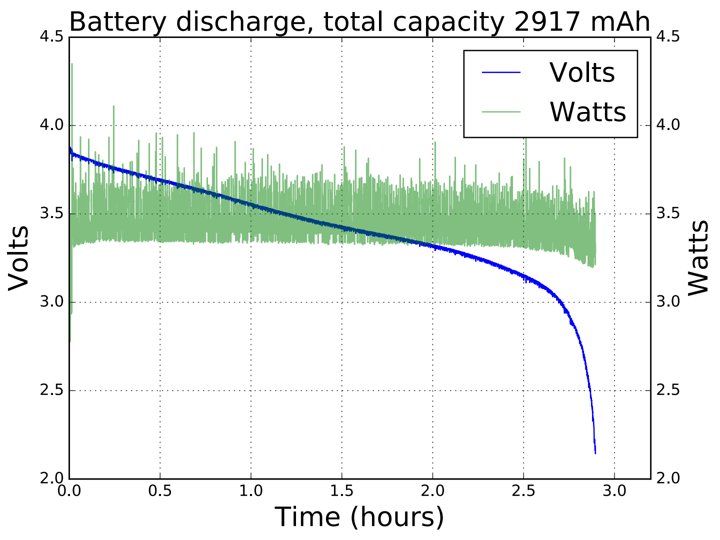


Figure 5.4: Battery voltage and power consumption. Battery life of close to three hours while running a ROS graph with nodes retrieving camera data at 640x480 pixels 15 Hz, performing simple blob detection, exchanging control packets at 200 Hz with the e-puck dsPIC, and running a basic behaviour tree interpreter. All the Neopixel LEDs were lit at 50% brightness and varying colour, and telemetry was streamed over WiFi at an average bandwidth of 10 KBytes/s. The fall-off in power consumption at the 2.5 hour point is due to the battery voltage falling below the threshold voltage of the blue LEDs within the Neopixels.

This resulted in a battery life of close to 3 hours while running a ROS graph with nodes retrieving camera data at 640x480 pixels 15 Hz, performing simple blob detection, exchanging control packets at 200 Hz with the e-puck dsPIC and conditioning the returned sensor data, and running a simple swarm robot controller. All the LEDs were lit at 50% brightness and varying colour, and telemetry was streamed over WiFi at an average bandwidth of 10 KBytes/s. Figure 5.4 shows the discharge curve. Power is relatively constant throughout at about 3.3 W except at the end, where it drops slightly. This is due to the Neopixel LEDs being supplied directly from the battery. As the voltage drops below about 3.1 V, the blue LEDs stop working, reducing the power consumption.

Camera interface

The e-puck VGA camera is a Pixelplus PO3030K or PO6030K, depending on the e-puck serial number. Both types have the same electrical interface, although the register interface is slightly different. It is a 640x480, 30fps CMOS sensor, controlled by I^{2}C, and supplies video on an eight bit parallel bus with some additional lines for H and V sync. By default, the camera provides 640x480 data within an 800x500 window in CrYCbY format. Each pixel is 16 bits and takes two clocks. The maximum clock frequency of 27 MHz gives 30 fps, with a peak bandwidth of 27 MBytes/s, sustained 18.4 MBytes/s. At our minimum desired framerate of 10 Hz, the clock would be

9 MHz.

We considered a number of possible solutions to the problem of getting the VGA camera data into the XU4, initially focussing on implementing a USB Video Class device, which would then be simply available under the standard Linux webcam driver but available devices were relatively expensive (e.g. XMOS XS1-U8A-64 £18, Cypress Semiconductor CYUSB3014 £35, UVC app notes available for both). In the end, we settled on a more flexible approach, using the widely available and cheap FTDI FT2232 USB interface chip, together with a low power and small FPGA from Lattice.

We wanted a low cost solution; the FT2232H is around £5, and provides a USB2.0 High Speed interface to various other protocols such as synchronous parallel, high speed serial, JTAG, etc. It is not programmable though, and cannot enumerate as a standard UVC device. The FT2232H provides a bulk transfer mode endpoint. This is not ideal for video, since it provides no latency guarantees, unlike isosynchronous mode, but since we control the whole system, we can ensure that there will be no other devices on the USB bus that could use transfer slots.

Although the FT2232H provides a synchronous parallel interface, it is not directly compatible with the camera. The FT2232H has a small amount of buffering, and uses handshaking to provide backpressure to the incoming data stream if it cannot accept new data, whereas the camera has no storage and simply streams data at the clock rate during the active 640 pixels of each line. In order to provide buffering and handle interfacing, we chose to use the Lattice Semiconductor iCE40HX1K FPGA. This low cost device, less than £4 in a TQ144 package, has 96 programmable IO pins in four banks each of which that can run with 1.8 V, 2.5 V, or 3.3 V IO standards. It has 64 Kbits of RAM, sufficient to buffer 6.4 lines of video, or 1.3 ms at our minimum desired framerate. We assume that the Linux USB driver at the XU4 end can handle all incoming USB data provided there is an available buffer for the data, meaning that the combined maximum latency of the user application and kernel driver must not exceed 1.3ms to avoid underruns. Given reported sustained datarates of 25 MBytes/s for the FT2232H, this seems plausible, although should this not prove possible, we had the fallback position of being able to lower the camera clock frequency to a sustainable level.

The decision to use an FPGA with the large number of IOs capable of different voltage standards gave greater design freedom. There is no need for any other glue logic, and it is possible to design defensively, with a number of alternative solutions to each interface problem. It also makes possible the later addition of other peripherals. For this reason, sixteen uncommitted FPGA pins were brought out to an auxiliary connector. Lattice Semiconductor provide an evaluation kit, the iCEstick, broadly similar to the proposed subsystem, allowing early development before the completion of the final PCBs.

The final system proved capable of reliably streaming camera data at 15 fps, or 9.2 MBytes/s, with a camera clock of 12 MHz.

I^{2}C and SPI communications, Neopixel LEDs

All the e-puck sense and control data, except for the camera, flow over the SPI interface. It is used to control the e-puck motors and LEDs, the Neopixel LEDs on the interface board, and to read from the accelerometers and IR proximity sensors on the e-puck. The I^{2}C bus is only used to set the parameters of the VGA camera.

The SPI bus is a synchronous full-duplex serial communication *defacto* standard for communicating between chips. It has a fixed single master, which initiates and controls all communication, and potentially multiple slaves, each sharing clock and data in and out lines, and each with their own enable line. It is generally capable of much faster data rates than IIC, with typical clock rates of multiple MHz. Communication takes place in packets, with controllers usually providing sizes in multiples of eight.

As with the LEB, the XU4 board acts as the SPI master, providing the clock and enable signals, and the dsPIC of the e-puck the slave. SPI communication is formed of 16-bit packets. Both the master and slave have a 16 bit shift register and communication is full duplex. The master loads data into its register and signals the start of communications, followed by 16 clocks, each shifting one bit of the shift register out from the master and into the slave. Simultaneously, the slave data is shifted into the master. Between each 16 bit packet, communication pauses for long enough for the master and slave to process the received packet and prepare the next outgoing packet. This is handled in hardware with fully pipelined DMA at the XU4 end requiring no delay between packets, but the dsPIC has no DMA and uses an interrupt routine to perform this. We used an SPI clock frequency of 5 MHz, chosen as the maximum that maintained good signal integrity as measured with a scope at each end of the signal path[3]. Interpacket delay was set to 32 SPI clock cycles, corresponding to 6.4 us. This allows time for about 94 dsPIC instruction cycles, sufficient to handle the longest possible interrupt latency of 88 cycles with some margin[4].

The SPI signals were routed to the FPGA and the board design allows for them to be routed through it. This enables two things; firstly, the FPGA can watch the data from the XU4 and use fields within that to control its own peripherals, currently the Neopixel LEDs, secondly it makes it possible to modify the return data packets from the dsPIC, allowing, for example, the insertion of data into the packet stream from

[3]The signal path is Exynos SoC->20cm ribbon cable->Level conversion->FPGA->board-to-board connector->dsPIC

[4]70 cycles for high priority stepper motor step generation interrupt service routine, and 18 cycles for the SPI service routine

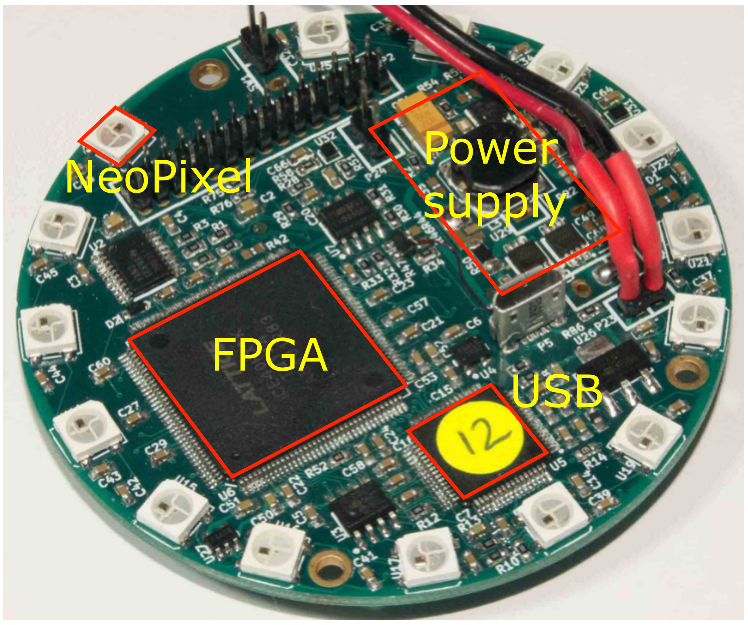


Figure 5.5: Interface board PCB, showing the boost converter PSU for the XU4 5v supply, the FPGA and USB interface, the VGA camera and SPI level shifting, and the 15 Neopixels.

possible future sensors attached to the FPGA.

The FPGA contains additional logic to interpret fields within the SPI packet for controlling the Neopixel LEDs. These data are stored in a buffer within the FPGA and used to generate the appropriately formatted serial stream to the LEDs.

5.1.5 Physical design

The interface board is 70 mm in diameter, the same as an e-puck. It sits on top of the base e-puck. Above this, the XU4 board is held vertically within an 75 mm diameter cylindrical 3D printed shell, which also holds the battery. Flying leads from the XU4 for the GPIO parallel and the USB interfaces, and for the power supply, connect to the interface board. Figure 5.5 show shows the interface PCB, and Figure 5.6 shows 16 completed Xpucks, and the major components of the assembly.

5.2 Software and infrastructure

The swarm operates within an infrastructure that provides tracking, virtual sensing, and message services. To facilitate this, the Xpucks run a full featured distribution of Linux and ROS, the Robot Operating System [Quigley *et al.* , 2009]. This gives access to much existing work; standard libraries, toolchains, and already existing robot software. Given the close dependence of ROS on Ubuntu we chose to use Ubuntu 14.04.4 LTS, running ROS Indigo.

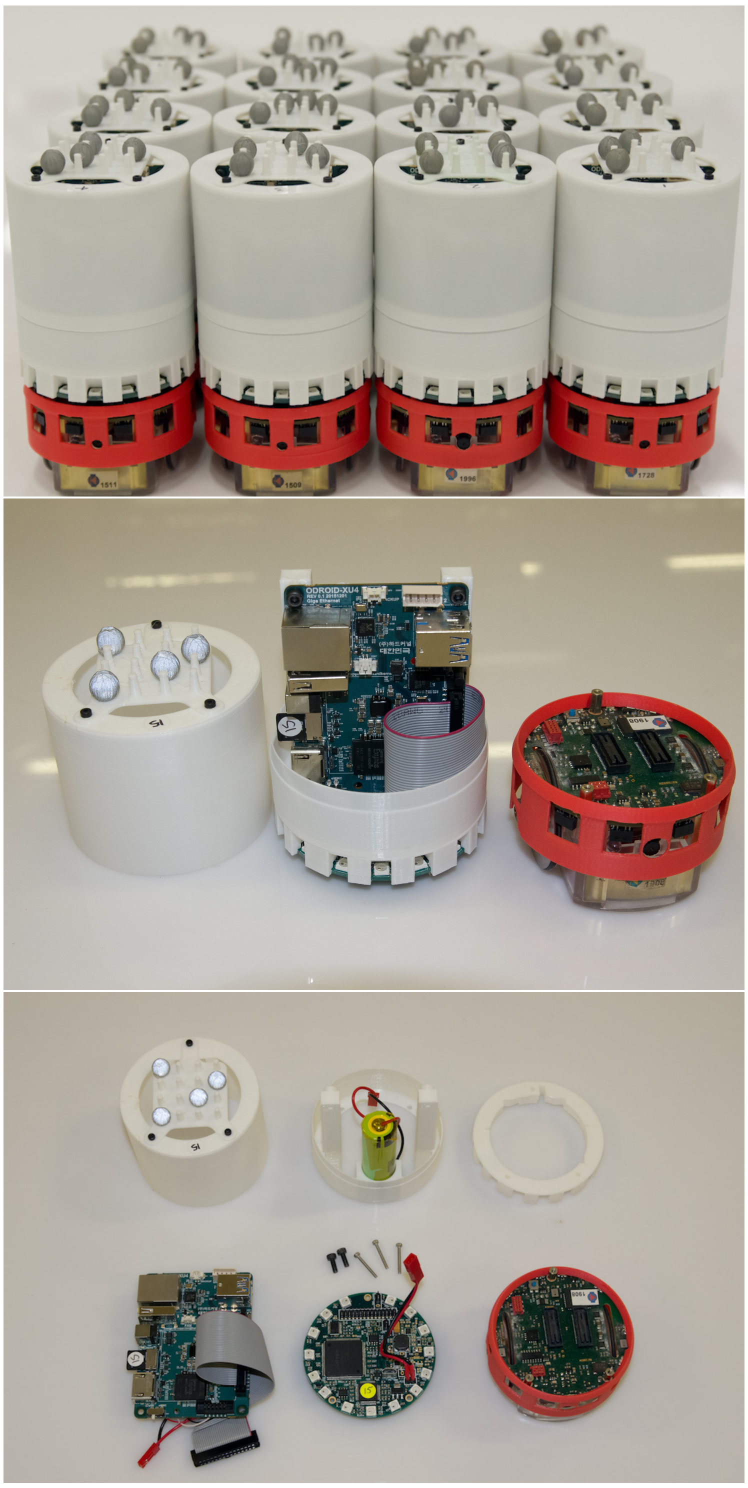


Figure 5.6: Xpuck swarm and construction. Top: 16 assembled Xpucks. Centre: Partially disassembled Xpuck, showing arrangement of components. Bottom: Major components, left to right, top to bottom. Outer 3D printed shell, showing Vicon tracking reflectors in unique pattern on top. Support chassis, which holds the XU4 single board computer and the LiION battery. Spacer ring, locating the chassis above the PCB and reflecting the upward facing LEDs outwards. XU4 computer, with leads for power and data. Interface PCB. Base e-puck, with red 3D printed skirt.

5.2.1 Real time kernel

The standard Linux kernel is not hard real-time, that is, it does not offer bounded guarantees of maximum latency in response to events. One of the tasks that is running on the XU4 that requires real-time performance is the low-level control loop comprising the SPI data message exchange with the e-puck. The maximum speed of the e-puck is about 130 mm/s. A distance of 5 mm corresponds to about 40 ms. It would be desirable to have a control loop with a period several times faster than that, one commonly used in e-puck experiments is 100 Hz, or $t_{control} = 10\,\text{ms}$. The minimum time for the control loop to respond to a proximity sensor is two SPI message lengths, so to achieve a 10 ms control period, we need an SPI message period $t_{period} < 5\,\text{ms}$. Assuming a 5 MHz SPI clock with a message comprising 32 16 bit packets and a 6.4 µs interpacket gap, the total time per message is $t_{message} = 307\,\mu\text{s}$. This gives a budget of $t_{period} - t_{message} = 4.7\,\text{ms}$ for processing and latency. Measurements using cyclictest[5] over 500000 loops of 1 ms, or about 8 minutes, with the *Server* preemption policy kernel while running SPI message exchange at 200 Hz showed figures of 13.9 ms, and even when running the *Low-Latency Desktop* preemption policy this was above 3.5 ms. This leaves little margin for processing.

We used the PREEMPT-RT patch, Rostedt & Hart [2007], which modifies the kernel to turn it into a real time operating system (RTOS), able to provide bounded maximum latencies to high priority real-time user tasks. With the RTOS kernel the measured latencies while running SPI message exchange never exceeded 457 µs over several hours running at 200 Hz.

Measurement of actual latencies within the SPI message code still showed some very high latencies, which we fixed with a patch[6] addressing this issue with the Samsung S3C64xx SPI peripheral. With this patch applied, measured latencies within the SPI message code never exceeded 457 µs over several hours running at 200 Hz.

5.2.2 Resilient filesystem

One of the important issues when making reliable Linux embedded systems is how to deal with unexpected power removal. Linux filesystems, in general, are likely to be corrupted if the power is removed while they are performing a write. Even journalling filesystems like ext4 are prone to this. This is why Linux needs to be properly shut down before power is removed, but this is simply not practical for an experimental battery-powered system. Disorderly shutdowns will happen, so this needs to be planned for.

We implement a fully redundant filesystem with error checking using BTRFS [Rodeh

[5]https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest

[6]Jeff Epler, [RFC 0/4] Improving SPI driver latency (vs v3.8.13.14-rt31), http://www.spinics.net/lists/linux-rt-users/msg12195.html

et al. , 2013] as described in a StackExchange answer[7]. BTRFS is modern journalling filesystem that supports on-the-fly compression and RAID, and is capable of self-healing, provided there are redundant copies of the data. The idea is that we create two partitions on the same SD card and mount them as a completely redundant RAID1 array. Any filesystem corruption will be seen as a mismatch between checksum and file, and the redundant copy on the other partition used to replace the corrupt version. This has proven to be very reliable so far, with no corrupted SD cards.

5.2.3 Arena integration

The Xpucks work within an arena which provides the infrastructure for experiment control, implementing virtual senses if needed, and for logging, see Figure 5.7. It is an area 2 m by 1.5 m equipped with a Vicon[8] tracking system and an overhead network webcam. Each Xpuck has a USB WiFi dongle, and the arena has a dedicated WiFi access point. For robustness, each Xpuck has a fixed IP address and the standard scripts are replaced with a script that continually checks for connectivity to the access point and attempts reconnection if necessary.

The motion tracking system is a Vicon MX equipped with four cameras and connected to a dedicated PC. The cameras are modified to use visible rather than infra-red light so as not to interfere with the IR proximity sensors of the e-pucks. The system is capable of tracking unique combinations of reflective markers that are fixed to the top of the robots to millimetre accuracy at a frame rate of 50 Hz. Pose data from the tracked markers is available on a network socket with a processing delay of three frames, giving a minimum latency from motion to data of 60 ms.

Software called the *switchboard* runs on the *Hub* server and is responsible for the distribution of experiments to the Xpucks, their initiation, and the logging of all experiment data. Each Xpuck automatically starts a ROS node at boot which connects to the Hub over ZeroMQ sockets [Hintjens, 2013] supplying a stream of telemetry about the physical state of the Xpuck, including battery levels and power consumption, temperature, sensor values and actuator settings. The switchboard sends timestamps, virtual sense data, and can command the ROS node to download and execute arbitrary experiment scripts, which would typically set up a more complex ROS graph for the robot controller, which in turn will run the experiment upon a trigger from the switchboard. Controllers are always run locally on the Xpucks. This is all controlled either from the command line on the Hub, or with a GUI giving visibility to important telemetry from the swarm.

[7]Corruption-proof SD card filesystem for embedded Linux? http://unix.stackexchange.com/questions/136269/corruption-proof-sd-card-filesystem-for-embedded-linux

[8]`https://www.vicon.com/`

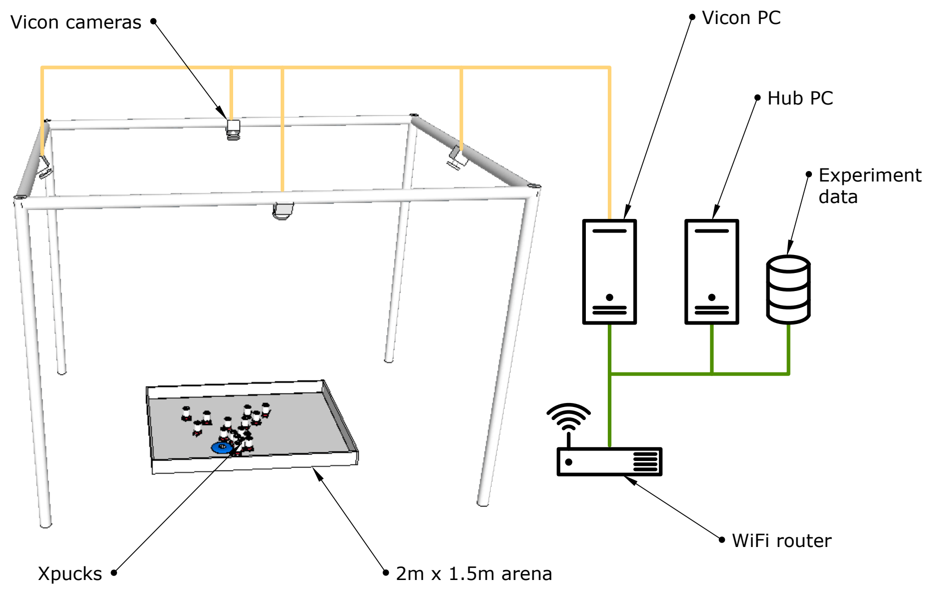


Figure 5.7: The Xpuck arena. Experiments take place within a 2 m x 1.5 m area surrounded by walls slightly higher than the height of the Xpucks. Each Xpuck has a unique pattern of spherical reflectors on their top surface to enable the Vicon motion tracking system to identify each individuals pose. The Vicon PC is dedicated to managing the Vicon system and makes available a stream of pose data. The Hub PC is responsible for all experiment management, data logging, and virtual sense synthesis.

Each Xpuck is marked with a unique pattern of reflectors recognised by the Vicon system. There are four reflectors arranged on a 4x4 grid with spacing of 10 mm. We used a brute force search to find unique patterns for each of the 16 Xpucks. Because of the size of the marker pattern and of the Xpucks themselves, there should be no ambiguous conditions when Xpucks are close to each other. This has proved effective, with unambiguous detection even when all 16 Xpucks were placed packed together in the arena.

The switchboard software connects to the Vicon system and receives pose messages at the update rate of 50 Hz. This is used to log the absolute positions of the Xpucks during experiments and also to synthesise virtual senses included in the outgoing streams of data from the switchboard to the Xpucks. Range and bearing is an important sense in swarm robotics experiments, which we can construct directly using the e-pucks IR proximity sensors or with additional hardware [Gutiérrez *et al.* , 2009a,b]. We can also synthesise range and bearing information from the Vicon data with behaviour defined by a *message distribution model*, which allows us to specify parameters such as range, noise, and directionality. There is the capability for Xpucks to send broadcast messages consisting of their ID, this is disseminated by the switchboard according to the message distribution model. Messages received have no content, but are an indication that the sender and the receiver can communicate, actual data transfer can take place point-to-point. In this we take inspiration from O'Dowd *et al.* [2014], who use IR communication between e-pucks to establish if contact is possible, data transfer then taking place over WiFi.

5.3 GPGPU robot simulator

In this section we describe the design and realisation of a fast parallel physics-based 2D multi robot simulator running on the Xpuck SoC GPU.

In order to perform on-board evolution of controllers or to evaluate multiple what-if scenarios, we need to be able to run many simulations much faster than real-time. A typical evolutionary algorithm might have a population of p potential solutions. Each of these needs to be evaluated for fitness by running r simulations with different starting conditions. Many generations g of evaluation, selection, combination, and mutation take place to produce fitter individuals. Typically, p, r, g might be $(50, 10, 100)$. One scenario we envisage is evolving a controller for the next fixed interval Δt of real time. During the current time interval, we need to complete $n_{sims} = prg$ simulations of that time Δt, or:

$$\frac{n_{sims} \cdot t_{real}}{t_{sim}} < 1 \tag{5.3}$$

where t_{sim} is the simulated time and t_{real} is the wall clock time for that simulated time. It is generally the case [Vaughan, 2008; Jones *et al.* , 2015] that multi robot simulation time is proportional to the number of agents being simulated. We define a simulator speed using the robot acceleration factor:

$$r_{acc} = \frac{n_{robots} \cdot t_{sim}}{t_{real}} \tag{5.4}$$

where n_{robots} is the number of robots, t_{sim} and t_{real} as above. With Eqn 5.3 we get a required r_{acc} of:

$$r_{acc} > n_{sims} \cdot n_{robots} \tag{5.5}$$

We can see that if we are using a single simulator, the required r_{acc} increases with the number of robots being simulated. But if we run a distributed evolutionary algorithm, and have a simulator embodied in each robot, the required r_{acc} simply becomes:

$$r_{acc} > n_{sims} \tag{5.6}$$

For the example above, we therefore require a simulator with $r_{acc} > 50000$.

There is a basic trade-off between simulator fidelity and speed. Typical values of

r_{acc} when running on a PC are 25 for a full 3D physics simulation like Gazebo, 1000 - 2000 for 2D[9] arbitrary shape simulators with relatively detailed sensory modelling like Stage [Vaughan, 2008], and ARGoS [Pinciroli *et al.* , 2011], and 50000 - 100000[10] for constrained geometry 2D physics game engines like Box2D [Catto, 2009]. There is also a cost to generality; the critical path in Stage is the ray-tracing operation for modelling of distance sensors, necessary in order to handle arbitrary object shapes in the simulated world. We show in Jones *et al.* [2016] that a constrained geometry 2D physics engine simulator is capable of being used to evolve swarm controllers which transfer effectively to the real world, so this motivates our simulator design.

For the experiment described in Jones *et al.* [2016] we wrote a robot simulator for swarms of kilobots [Rubenstein *et al.* , 2012] based on the 2D physics game engine Box2D [Catto, 2009]. This achieved an r_{acc} of 8×10^4 running on a 3.2 GHz iMac computer. Box2D models objects as simple convex shapes, with mass per unit area, and collisions between the objects in a physically realistic way. The speed comes with loss of generality; because shapes are not arbitrary, collisions and intersections can be computed geometrically. We made some initial experiments with the performance of the Box2D-based kilobot simulator on the XU4 with a view to accelerating the core engine on the GPU. Running purely on the CPU achieved an r_{acc} of around 1000, some 50 times too slow. Examining the code and that of an OpenCL accelerated port[11] showed that it would be hard to achieve GPU acceleration with the relatively low number of objects (10s) we expect within our simulated worlds.

In order to get good performance on an application running on a GPU, it is necessary that there is a large number of work items that can be performed in parallel. The Mali Midgard GPU architecture present in the Exynos 5422 SoC of the XU4 has six shader cores, each of which can run 256 threads simultaneously. In order to keep the cores busy it is recommended that a kernel be executed over hundreds or thousands of work items, depending on its resource usage. We therefore need to design our simulator to have parallelism at least in the hundreds to take advantage of the GPU, and be sufficiently constrained in scope that we avoid the costs of generality; by using only straight lines and circles in our simulation, collisions and sensor intersections can be calculated cheaply by geometry, rather than expensive ray-tracing.

As well as the physical aspects of the simulation, the controllers for each simulated robot also need to be implemented for good performance. We use behaviour trees as the controller architecture, for good performance on a GPU we needed to pay some attention to the characteristics of OpenCL.

[9]or "two-and-a-half D" with sensors having some awareness of Z but kinematics and dynamics modelled purely in 2D

[10]We achieved 80000 with our Box2D-based kilobot simulator [Jones *et al.* , 2016]

[11]https://download.tizen.org/misc/media/conference2014/.../tdc2014-webphysics.pdf

5.3.1 Simulation model

The simulation models up to 16 Xpuck robots within a 2 m x 1.5 m rectangular arena centred about the origin, with the edges of the arena surrounded with immovable walls. As well as the Xpuck robots, there can be other inert round objects that can be pushed by the Xpucks. The reference model for the robots is given in Table 6.1, this describes the sensors and actuators that are exposed to the robot controller.

We can divide the simulation into three sections; *physics*, *sensing*, and *control*. *Physics* handles the actual physical behaviour of the robots within the arena, modelling the dynamics of motion and collisions in a realistic way. *Sensing* constructs the input variables described in the robot reference model from the locations and attributes of the objects within the simulated world. *Control* runs a robot controller for each simulated robot, responding to the reference model inputs, and producing new output variables, resulting in the robot acting within the world.

There are three types of object within the world, the arena walls, the Xpucks, and inert objects. The walls are immoveable and are positioned horizontally and vertically symmetrically about the origin. Xpucks, which are round and coloured, can sense each other with their camera, proximity sensors and range and bearing, and can move with two-wheel kinematics. Inert objects, which are round and coloured, can be sensed by Xpuck cameras but not by the proximity sensors because they are low in height. They move only due to collisions.

Physics

The physics core of the simulation is based on work by Gaul [2012]. There are only circular bodies, which are rigid and have finite mass, and the walls, which have infinite mass. Interactions between bodies are governed by global attributes of coefficients of static and dynamic friction, and restitution. Interactions between the bodies and the arena floor are governed by individual attributes of mass and coefficient of friction. The physical state of each body i is described by the tuple $S_i(\boldsymbol{x}, \boldsymbol{v}, \theta, \omega)$ representing position, velocity, angle, angular velocity.

The equations of motion governing the system are:

$$\begin{aligned}
\dot{\boldsymbol{v}} &= \frac{1}{m}\boldsymbol{F} \\
\dot{\omega} &= \frac{1}{I}\tau \\
\dot{\boldsymbol{x}} &= \boldsymbol{v} \\
\dot{\theta} &= \omega
\end{aligned} \tag{5.7}$$

Where $\boldsymbol{F}$ is force, m is mass, τ is torque, I is moment of inertia. They are integrated using the symplectic Euler method [Niiranen, 1999] which has the same computational cost as explicit Euler but better stability and energy preserving properties.

Collisions between bodies are resolved using impulses. For each pair of intersecting bodies, a contact normal and relative velocity are calculated, producing an impulse vector which is used to instantaneously change the linear and angular velocities of the two bodies. This is iteratively applied multiple times to ensure that momentum is transferred in a physically realistic way between multiple contacting bodies.

Collision detection between pairs of bodies with a naive algorithm has $O(n^2)$ performance scaling[12] so most physics simulators handling a large number of bodies (100s upwards) use a two stage process with a *broadphase* step that eliminates a high proportion of pairs that cannot possibly be in collision, before the *narrowphase* step that detects and handles those bodies that are actually colliding. But we have only a maximum of 21 bodies (4 walls, 16 robots, 1 object) which means that any broadphase step must be very cheap to actually gain performance overall. We tried several approaches before settling on a simple binning algorithm: Each object is binned according to its x coordinate, with bins just larger than the size of the objects. A bin contains a bitmap of all the objects within it. Objects can only be in collision if they are in the same or adjacent bins so the or-combined bitmap of each two adjacent bins is then used to form pairs for detailed collision detection.

The two wheel kinematics of the robots are modelled by considering the friction forces on each wheel due to its relative velocity to the arena surface caused by the wheel velocity and the object velocity. Friction force is calculated as Coulomb but with μ reduced when the velocity v is close to zero using the formulation in Williams *et al.* [2002]:

$$\mu = \mu_{max} \frac{2 \cdot arctan(k * v)}{\pi} \tag{5.8}$$

With the same justification as Williams *et al.* [2002], we chose $k = 20$ empirically to ensure numerical stability. The forces on each body are resolved to a single force vector $\boldsymbol{F}$ and torque τ. Non-robot objects simply have zero wheel velocities with the wheelbase modified to reflect the physical behaviour.

The noise model is a simplified version of that described by Thrun *et al.* [2005]. Three coefficients, $\alpha_1, \alpha_2, \alpha_3$, control respectively velocity dependent position noise, angular velocity dependent angle noise, and velocity dependent angle noise. So position and angle are modified:

$$\begin{aligned} \boldsymbol{x'} &= \boldsymbol{x} + \boldsymbol{v} \cdot s(\alpha_1) \\ \theta' &= \theta + \omega \cdot s(\alpha_2) + |\boldsymbol{v}| \cdot s(\alpha_3) \end{aligned} \tag{5.9}$$

[12]Big O notation describes how the performance of an algorithm scales with increasing input size

where $s(\sigma)$ is a sample from a Gaussian distribution with standard deviation σ and mean of zero. Because the noise model is on the critical path of position update and the calculation of even approximate Gaussian noise is expensive, we use a pre-calculated table of random values with the correct distribution.

The physics integration timestep is set at 25 ms for an update rate of 40 Hz. This value was chosen as a trade-off performance and physical accuracy, meeting the simulator performance requirements while still giving realistic physical behaviour. The 40 Hz rate gives 4 physics steps per controller update cycle.

Sensing

There are three types of sensors that need to be modelled. Each Xpuck has eight IR proximity sensors arranged around the body at a height of about 25 mm. These can sense objects out to about 40 mm from the body. The reference model specifies that the real-valued reading varies from 0 when nothing is in range, to 1 when there is an object adjacent to the sensor. Similarly to the collision detection above, the maximum sensor range is used to set the radius of a circle about the robot which is tested for intersection with other objects. For all cases where there is a possible intersection, a ray is projected from the robot at each sensor angle and a geometrical approximation used to determine the location of intersection with the intersected body and hence the range. This process is actually more computationally expensive than collision detection, but only needs to take place at the controller update rate of 10 Hz.

The second and third types of sensor are the camera blob detection and the range and bearing sense. Blob detection splits the camera field of view into three vertical segments and within each segment, detects the presence of blobs of the primary colours. Range and bearing sense counts the number of robots within 0.5 m and produces a vector pointing to the nearest concentration. Together they are the most computationally expensive of the senses to model. They necessarily traverse the same data structures and so are calculated together.

To model the camera view, we need to describe the field of view subtended by each object, what colour it is, and whether is is obscured by nearer objects. We implement this by dividing the visual field into 15 segments and implementing a simple z-buffer. Each object is checked and a left and right extent derived by geometry. The segments that are covered by these extents have the colour of the object rendered into them, if the distance to the object is less than that in the corresponding z-buffer entry. As each object is checked, the distance is used to determine if the range and bearing information needs to be updated.

In the real robot arena, range and bearing is implemented as virtual sensing using a

Vicon system and communication over WiFi. There is significant latency of around 100 ms-200 ms between a physical position and an updated range and bearing count and vector reaching the real robot controller. Also the camera on each Xpuck has processing latency of a similar order. For this reason and due to the computational cost, this sensor information is updated at half the controller rate, or 5 Hz.

Controller

The controller architecture we use is behaviour tree based, discussed in detail in Chapter 3. A behaviour tree consists of a tree of nodes and a *blackboard* of variables which comprise the interface between the controller and the robot. At every controller update cycle, the tree of each robot is evaluated, with sensory inputs resulting in actuation outputs. Evaluation consists of a depth-first traversal of the tree until certain conditions are met. Each agent has its own blackboard, state memory and tree. Although our initial work is using homogeneous swarms, with all robots executing an instance of the same tree, we wanted the flexibility to support heterogeneous swarm simulation. Supporting both these forms requires separating the tree from the blackboard and state memory.

5.3.2 Implementation of simulator on GPU

To best exploit the available performance of the GPU, our implementation must have a high degree of parallelism. We achieve this by running multiple parallel simulations almost entirely within the GPU. We pay little penalty transferring data, since each thread is relatively long lived, and although we will have inevitable thread divergence, we should not suffer a performance penalty because of the particular characteristics of the Mali Midgard architecture. Because the total number of objects in the simulation is low, in the 10s, we cannot achieve the necessary parallelism within the simulation itself. The limit to parallelisation of running multiple simulations for an evolutionary algorithm is the number of simulations per generation; it is necessary to completely evaluate the fitness of the current generation in order to create the individuals that will make up the next generation. With the numbers given above, this would be 500 simulations, below what would normally be recommended to keep the GPU busy, but long lasting threads ensure the GPU is fully utilised.

As we implemented the simulator, it actually turned out that memory organisation was the most critical element for performance. Each of the four cores within the first core group of the GPU[13] has a 16 Kbyte L1 data cache and a 256 L2 cache shared between them. Ensuring that data structures for each agent were minimised, and that they fitted within and were aligned to a cache line boundaries resulted in large performance improvements. Memory barriers between different stages of the

[13]The six cores are divided into two core groups, one with four cores and one with two. These are presented as two separate OpenCL devices. For ease of coding, only one core group was used.

simulation update cycle ensured that data within the caches remained coherent and reduced thrashing. As performance improved and the memory footprint changed, the effect of workgroup size and number of parallel simulations was regularly checked. We used the DS-5 Streamline[14] tool from ARM to visualise the performance counters of the GPU which showed clearly the memory-bound nature of the execution. Profiling of OpenCL applications is difficult at anything finer than the kernel level, so there was much experimentation and whole application benchmarking.

5.3.3 Implementation of behaviour tree interpreter on GPU

The algorithm to execute a behaviour tree is given in Section 3.1.5. As we note, this is expressed most naturally using recursion. However, recursion is not necessarily the best way to implement a behaviour tree interpreter, since the use of the general call stack for storage is not as efficient as other methods, since (at least) the return address from each called function has to be stored, as well as any passed parameters.In addition, the amount of space available for the stack is relatively opaque.

More fundamentally, we need to implement the interpreter in OpenCL, and the language specification explicitly forbids function recursion.

The design requirements can be summarised as:

1. Separate tree and state storage.
2. Efficient use of memory
3. No recursion

For the initial experiment into evolving behaviour trees in Chapter 4, we used a single data structure for the tree and node state, where each node was a structure with pointers to any children, and embedded state. This works well when implemented on a single self-contained robot, as each kilobot was in that experiment, but is wasteful of space when we have a homogeneous swarm, since the tree is replicated. It also makes the Reset process in Algorithm 1 of tick evaluation more costly, involving a complete traversal of the tree to reach each item of state. For this reason, we have a static data structure for each tree, with fixed offsets for each node into state storage area. Each agent within the simulation has a pointer to its state space, and a pointer to the tree it is executing. This tree structure can be shared by any number of agents.

With up to 16 agents and 512 parallel simulations, we may need 8192 regions of memory for state storage. At every tick evaluation, Reset must change the state of each tree node according to Figure 3.7. Nodes that are in state *running* are moved

[14]https://developer.arm.com/products/software-development-tools/ds-5-development-studio/streamline

to state *active*, and nodes in any other state are moved to state *idle*. This process is relatively costly, since it applies to every node in the tree, but arranging the tree state as a flat structure means that it can be traversed in a simple linear fashion, with no pointer following.

The data structures are shown in Figure 5.8. The tree buffer consists of a header, with a pointer[15] to the root node of the tree, and fields holding the size of the blackboard and the total number of nodes. Following this are the tree nodes, each consisting of a type identifier, an offset to the node state in the state buffer, and node specific fields. Nodes with children will have 16 bit offset pointers to those child nodes. The state buffer of an agent consists of the current random number generator seed for this agent, the blackboard variables, and the state for each node. The per-node state is two bytes, the first is the current state of the node, the second is an index value, used by the sequence, selection, and repeat nodes. Not all node types use the index byte, but the regular structure is kept to allow easy traversal for the RESET process.

We have a fixed allocation of memory for each of the two buffers; 4096 bytes for the state buffer, and 16384 bytes for the tree nodes. Node sizes are variable, depending on the particular node, but average about 8 bytes. The fixed allocation thus allows for a maximum of 2048 nodes[16]. The total memory used for BTs when running 512 parallel simulations with 16 agents is about 160 Mbytes.

As noted above, OpenCL forbids recursion. In order to rewrite the algorithms in Chapter 3 in a non-recursive way, we have to provide an explicit stack to support the tree descent and evaluation; adding items while descending, removing them once evaluated. We also need to provide a way of returning values from lower nodes of the tree. Rather than using the explicit stack for this, we use the status bytes for each node that are necessary anyway. As well as supporting the state machine in Figure 3.7, they now also serve to store the final result of node evaluation. Higher level nodes can then read the result state of their children.

Finally, on the right in Figure 5.8 is an example of how a simple behaviour tree might appear in memory. The root offset points to the root `seq` node, highlighted in red. This node points to its state in the state buffer, and to its two children, a `movcv` and a `repeati`, which in turn points to its child `mulav`. Each node has a pointer to its state, and some parameters. The ordering in this example is an artefact of the recursive descent tree parsing process that generated the memory layout.

[15]Actually a 16 bit offset from the start of the buffer
[16]Since the limit is the number of 2 byte state entries

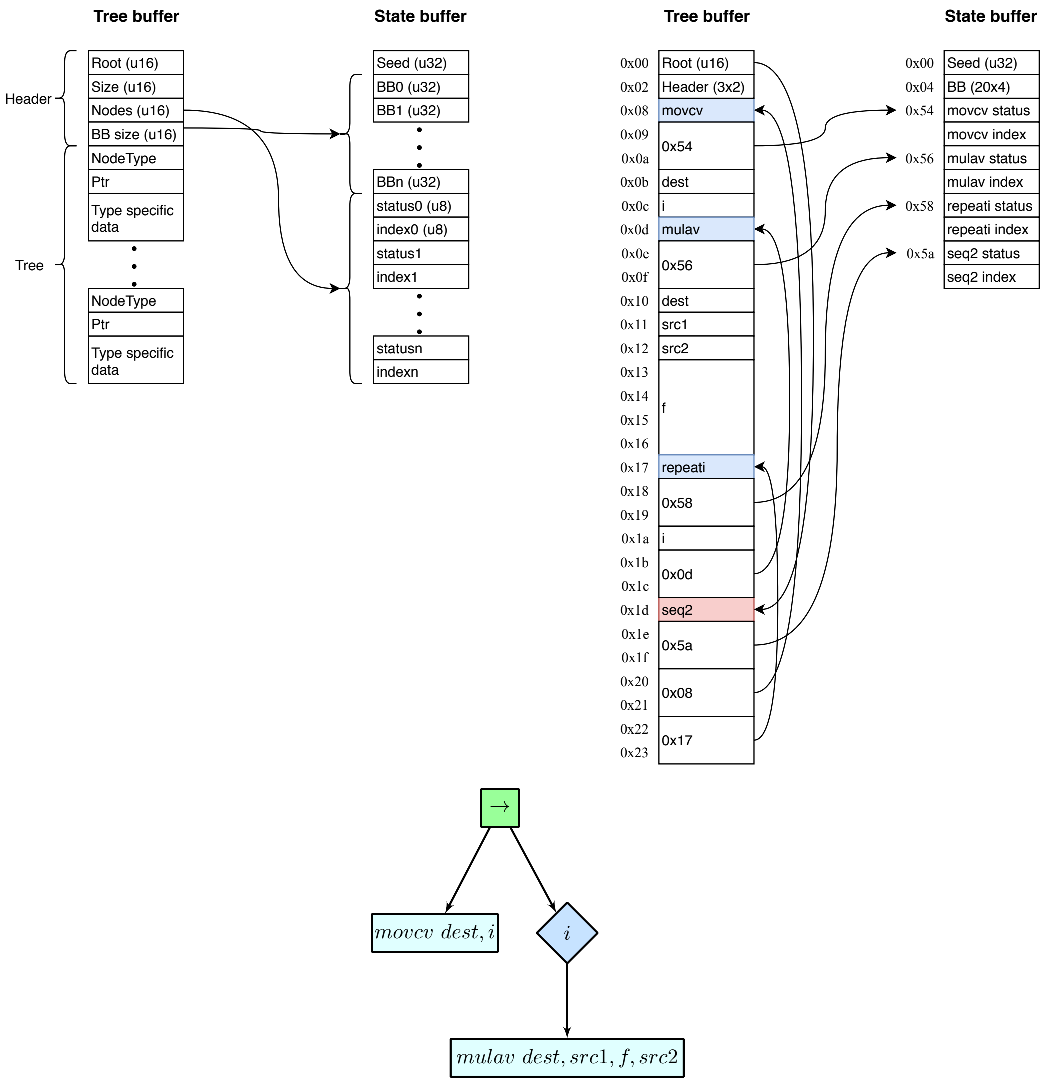


Figure 5.8: Behaviour tree interpreter data structures. On the left, the two columns are the tree structure and the state structure. On the right is a specific example of a simple tree, shown below in tree form.

5.3.4 Results: Performance of simulator

Table 5.3 shows the results of running parallel simulations for a simulated time of 30 seconds. Each simulation consists of 16 robots running a simple controller for exploration with basic collision avoidance, and one additional object that can be pushed by the robots. The effect of running different numbers of parallel scenes and with various different levels of functionality enabled are shown. t_{rss} is the time to simulate one robot second. $t_{rss} = \frac{1}{r_{acc}}$, so the required acceleration factor of 50000 corresponds to $t_{rss} = 20\mu s$. It can be seen that the requirement is met when running 256 simulations in parallel, with $t_{rss} = 17\mu s$. It is interesting to note that when running 512 simulations, the performance is better with all functionalities except the controller enabled. We surmise that, when running the controller, the total working set is such that there is increased cache thrashing with 512 parallel simulations.

Table 5.3: Speed of simulator with various functionalities enabled. 16 robots, 1 passive object, basic exploration and collision avoidance controller. Tested over five runs with 256 and 512 parallel simulations. t_{rss} is time (μs) per robot simulated second. With 256 parallel simulations, the physics functionality dominates at 40% of the processing time, but with 512 parallel simulations, controller processing is the largest proportion.

	256 simulations			512 simulations		
Functionality	t_{rss}	Δt_{rss}	%	t_{rss}	Δt_{rss}	%
Physics	6.9	6.9	40	6.7	6.7	31
+ Sensors	11	4.5	26	11	4.0	19
+ Camera and R&B	15	3.1	18	14	3.3	16
+ Controller (All functionality)	17	2.6	16	21	7.2	34

The performance of the simulator running on the Xpuck GPU is comparable to the same code running on the CPU of a much more powerful desktop system and at least ten times faster than more general purpose robot simulators such as Stage and ARGoS running on the desktop. Although later chapters will demonstrate the transferability of the evolved solutions from this simulator, we note at this stage that the fidelity of the simulator is similar to previous work [Jones *et al.* , 2016] which successfully transferred with only moderate reality gap effects.

5.4 Image processing demonstration: ArUco tag detection

The high computational capability of the Xpuck makes it possible to run camera image processing algorithms not possible on the e-puck on its own or enhanced with the Linux Extension Board. In order to demonstrate this and to evaluate the performance of the camera, we implement ArUco marker tracking [Garrido-Jurado *et al.* , 2014] and test it with the onboard camera. ArUco is a widely used library that can recognise square black and white fiducial markers in an image and generate camera pose estimations from them. In this demonstration, we use the marker recognition part of the library and test the tracking under different distances and Xpuck rotational velocities.

A ROS node was written to apply the ArUco[17] marker detection library function to the camera image stream and to output the detected ID and pixel coordinates on a ROS topic. Default detection options were used and no particular attention was paid to optimisation.

Two experiments were conducted. In both cases we used video from the Xpuck camera at a resolution of 320x240 and a frame rate of 15 Hz. First we measured the time taken to process an image with the detection function under conditions of no markers, four 100 mm markers in a 2x2 grid, and 81 20 mm markers in a 9x9 grid.

[17]Version 1.2, standard install from Ubuntu 14.04.4 ROS Indigo repository.

Frame times were captured for 60 seconds.

Second, we affixed four ArUco tags of size 100mm with different IDs to locations along the arena walls. An Xpuck was placed in three different locations within the arena and commanded to rotate at various speeds up to 0.7 rad/s. Data was collected for 31500 frames. Commanded rotational velocity, Vicon tracking data, and marker tracking data were all captured for analysis.

The data was analysed in the following way; each video frame is an observation, which may have markers present within it. Using a simple geometrical model, we predict from the Vicon data and the known marker positions whether a marker should be visible in a given frame and check this against the output of the detector for that frame. From this we derive detection probability curves for different rotation speeds.

5.4.1 Results: Performance of image processing task

For the computationally demanding image processing task, Table 5.4 shows the time taken for the Xpuck to process a 320x240 pixel frame using the ArUco library to search for markers. With four large markers, the 23 ms processing time is fast enough to sustain the full camera frame rate of 15 Hz. In the 81 marker case, detection speed slows to 94 ms, such that a 15 Hz rate is not sustainable. In both cases however, all the markers were correctly detected in each frame.

Table 5.4: Processing time for ArUco tag image recognition task under different conditions

Condition	Processing time (ms)	σ
No markers	12.4	2.7
4x 100 mm markers	23.3	4.5
81x 20 mm markers	93.8	0.25

The dsPIC of the e-puck would not be capable of running this code - it is only capable of capturing camera video at 40x40 pixels and 4 Hz with no image processing [Mondada *et al.* , 2009] and has insufficient RAM to hold a complete image. The Linux Extension Board processor could potentially run the detection code, but we estimate the processing time would be at least 50 times longer[18] giving a frame rate of less than 1 Hz.

The arena detection experiment collected 31500 frames, with 11076 marker detections possible in ideal circumstances. Actual detections numbered 8947, a total detection rate of 81%. Figure 5.9 shows the probability of detecting a marker under different conditions. With four markers around the arena, and the Xpuck capturing data at three locations within the arena, there are twelve distance/angle combinations. Distances vary from 0.5 m to 1.5 m, and angles from 0° to 70°. The grey envelope

[18]ARM926EJS @200MHz = 220DMIPS, A15 @800MHz = 2800DMIPS, 4x penalty for no floating point, single core only: 50x

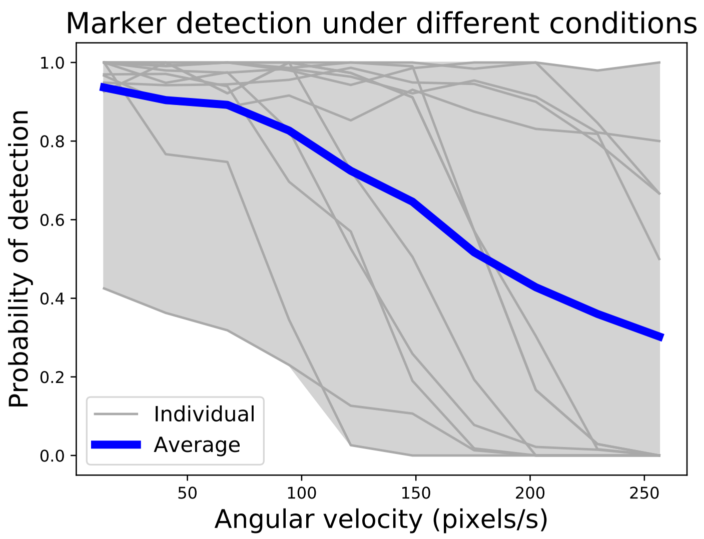


Figure 5.9: Probability of marker detection under different conditions. There are four markers around the arena, with data collected at three locations, giving twelve distance/angle combinations. Observations at a resolution of 320x240 pixels were made for 31500 frames, with 8947 marker detections out of a possible 11076, a detection rate of 81%. The number of detections compared to the maximum possible for each geometry were binned by angular velocity to give probability curves. Grey lines are individual distance/angle combinations, and the blue line is the average over all combinations. Generally, detection rate falls with increasing angular velocity, with a 50% detection rate at 180 pixels/s.

and lines show the individual distance/angle combinations against the angular velocity, with the blue line being the average over all observations. Angular velocity is expressed in pixels/s for better intuition about how fast a marker is traversing the field of view of the camera. Generally, the detection rate falls as the angular velocity increases, with a 50% detection rate at 180 pixels/s.

This shows that, even with unoptimised code, the Xpuck has sufficient computational performance, and the camera subsystem is of sufficient quality, that visual marker tracking is feasible. This would not be possible on the base e-puck.

5.5 In-swarm evolution demonstration

One of our motivations for moving computation into the swarm is to tackle the scalability of swarm controller evolution. To demonstrate both the computational capability of the Xpuck swarm and scalability, we implement an island model evolutionary algorithm and demonstrate performance improvement when running on multiple Xpuck robots.

The island model of evolutionary algorithms divides the population of individuals into multiple subpopulations, each of which follows its own evolutionary trajectory, with the addition of *migration*, where some individuals of the subpopulations are shared or exchanged with other subpopulations. Island model evolutionary algorithms enable coarse-grained parallelism, with each island corresponding to a different compute node, and sometimes outperform single population algorithms by maintaining diversity [Whitley *et al.* , 1999]. Even without that factor, the ability to scale the size of total population with the number of compute nodes hosting subpopulations is desirable for a swarm of robots running embodied evolution.

5.5.1 Implementation of island model

On each Xpuck, we run a genetic algorithm evolving a population of behaviour tree controllers similar to that described in Jones *et al.* [2016] using methods from Genetic Programming [Koza, 1992]. Evolution proceeds as follows; an initial subpopulation of $n_{sub} = 32$ individuals is generated using the Koza's *ramped_half_and_half* procedure, detailed in Poli *et al.* [2008], with a depth of $n_{depth} = 4$. Each individual is evaluated for fitness by running $n_{sims} = 8$ simulations with different starting conditions and averaging the individual fitnesses. The subpopulation is sorted and the top $n_{elite} = 3$ individuals are copied unchanged into the new subpopulation. The remaining slots are filled by tournament selection of two individuals with replacement followed by a tree crossover operation, with random node selection biassed to internal nodes 90% of the time [Koza, 1992], to create a new individual. Then, every parameter within that individual is mutated with probability $p_{mparam} = 0.05$, followed by mutating every node to another of the same arity with probability $p_{mpoint} = 0.05$, followed by replacing a subtree with a new random subtree with probability $p_{msubtree} = 0.05$. This new population is then used for the next round of fitness measurement.

The genetic algorithm is extended to the island model in the following way; after every n_{epoch} generations, each Xpuck sends a copy of the fittest individual in its sub-population to its neighbours. They replace the weakest individuals in their subpopulations. Currently, this is mediated through a *genepool server*, running on the Hub PC, although direct exchange of genetic material between individual Xpucks is also possible using local IR communication. This server maintains the topology and policy for connecting the islands. This may be physically based, drawing on the position information from the Vicon. It is important to note that server provides a way to abstract and virtualise the migration of individuals, in the same way we use the Vicon information to provide virtual sensing. When the server receives an individual from a node, it replies with a set of individuals, according to the policy. These are used to replace the least fit individuals on the requesting node. The process is asynchronous, not requiring that the nodes execute generations in lockstep. The policy for this experiment is to make a copy of each individual available to every other node, so with n_{nodes} nodes the migration rate is

$$r_{migration} = \frac{n_{nodes} - 1}{n_{sub} \cdot n_{epoch}} \tag{5.10}$$

Task and fitness function

We evolve a behaviour tree controller for a collective object movement task. The task takes place in a 2 m x 1.5 m arena with the origin at the centre and surrounded by walls greater than the height of the Xpucks. The walls and floor are white. A blue plastic frisbee of 210 mm diameter is placed at the origin. Nine Xpucks with red skirts are placed in a grid with spacing 100 mm centred at $(-0.8, 0)$ and facing rightwards. The goal is to push the frisbee to the left. Fitness is based on how far to the left the frisbee is after a fixed time. An individual Xpuck can push the frisbee, but at slower than the full Xpuck speed, so collective solutions have the potential to be faster. The swarm is allowed to execute its controller for 30 s. After this time, the fitness is given by Eqn 5.11.

$$f = \begin{cases} r_{derate} \frac{-x}{1 - l_{frisbee\_radius}}, & \text{for x} < 0 \\ 0, & \text{otherwise} \end{cases} \tag{5.11}$$

where x is the x-coordinate of the centre of the frisbee, and r_{derate} is a means of bloat control, proportionately reducing the fitness of behaviour trees which use more than 50% of the resources available.

In order to show scalability with increasing numbers of Xpucks, we compare two scenarios, firstly a single Xpuck running a standalone evolution and secondly six Xpucks running an island model evolution. In both cases the parameters are as above. With the island model, every $n_{epoch} = 2$ generations, a node sends to all its neighbours a copy of its fittest individual and receives their fittest individuals, using these to replace its five least fit individuals, giving a migration rate $r_{migration} = 0.078$. Each scenario is run ten times with different initial random seeds.

5.5.2 Results: Performance of island model evolution

The results are summarised in Figure 5.10. It is clear that the six node island model evolutionary system performs better than the single node. Maximum fitness reached is higher at 0.7 vs 0.5, and progress is faster. Of interest is the very low median fitness of the single node populations (shown with red bar in boxes), compared to the mean. This is because seven out of the ten runs never reached a higher fitness than 0.1 suggesting the population size or the number of generations is too small. Conversely, the median and mean of the island model population's maximum fitnesses are quite similar, showing a more consistent performance across runs. If we look at how fast the mean fitness rises, a single node takes 100 generations for the fitness to reach 0.15. The six node system reaches this level of mean fitness after 25 generation, four times faster.

Figure 5.11 shows a plot of the elapsed processing time per generation over ten runs. The variation is mostly due to the complexity and depth of the behaviour tree controllers within each generation, together with the trajectory of the robots in simulation. Each of the ten runs of both the island model and the single node systems completed in less than 10 minutes. For comparison, each evolutionary run in our previous work [Jones *et al.* , 2016] took several hours on a powerful desktop machine.

This demonstrates the Xpucks are sufficiently capable to host in-swarm evolutionary algorithms that scale in performance with the size of the swarm.

5.6 Experimental procedure

The arena infrastructure provides all the services required to run experiments with the Xpucks. In order to run an experiment, we use the following procedure.

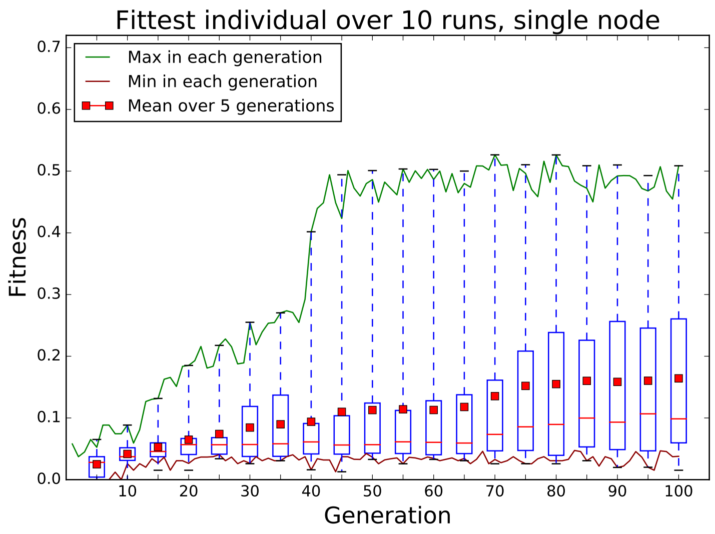
Each required Xpuck is turned on. During boot, ROS is automatically started[19] with a basic graph. This performs several functions. Firstly, it interfaces to the hardware of the Epuck and the Xpuck shim board. Secondly, it maintains a two-way communication link with the central hub computer. Thirdly it handles experiment sequencing, startup, and running. Finally, it presents an interface in the form of ROS topics that provide the senses and actuators detailed in the reference model, constructed from both real and virtual senses derived from the Vicon system.

On the hub PC, the *switchboard* program is started. This provides a graphical interface through which the state of the Xpuck robots can be monitored and controlled. When the robots have booted up, they establish communications with the *switchboard* and start exchanging telemetry. Important values are the battery voltage and the telemetry packet round-trip time. Once all the robots are running and showing good telemetry, the experiment can begin.

From the GUI, an experiment file is sent to all the robots. This is essentially a Python script that is executed on the robots. Typically, it will start the simulator and evolutionary algorithm, set up additional ROS nodes for executing behaviour tree controllers in real life, and enable logging of experiment data. All experiment files must be able to respond to `PAUSE`, `RUN`, and `STOP` commands from the *switchboard*. When all robots have indicated that they have received the experiment file and are

[19]Using `http://wiki.ros.org/robot_upstart`

A



B

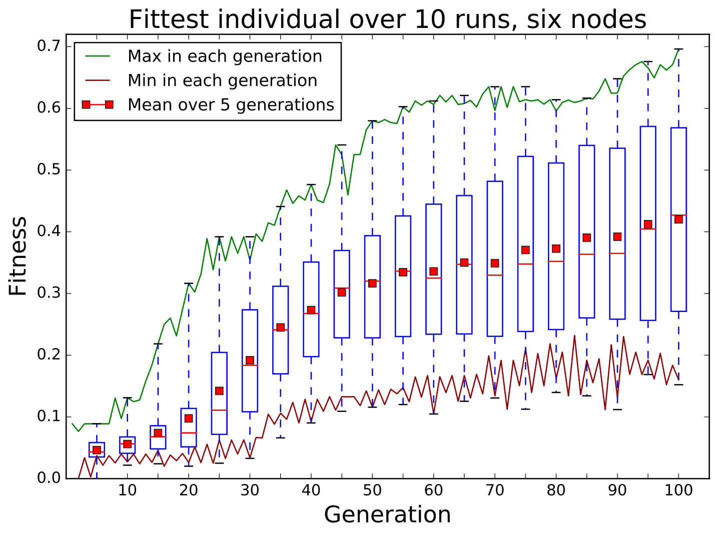


Figure 5.10: Single node vs six node island model performance. Comparison of 100 generations of evolution using a single node (A) and using an island model with six nodes (B). Each node has a population of 32 individuals, evaluated 8 times with different starting conditions for fitness. Each node in the six node system replaces is five least fit individuals with the fittest from the other five nodes every two generations. Boxes summarise data for that generation and the previous four. Red bar in boxes indicates median. The six node system clearly shows higher maximum fitness after 100 generations, and reaches the same mean fitness as the single node system in a quarter of the time. The large difference between mean and median in the single node system is due to seven of the ten runs not exceeding a fitness of 0.1.

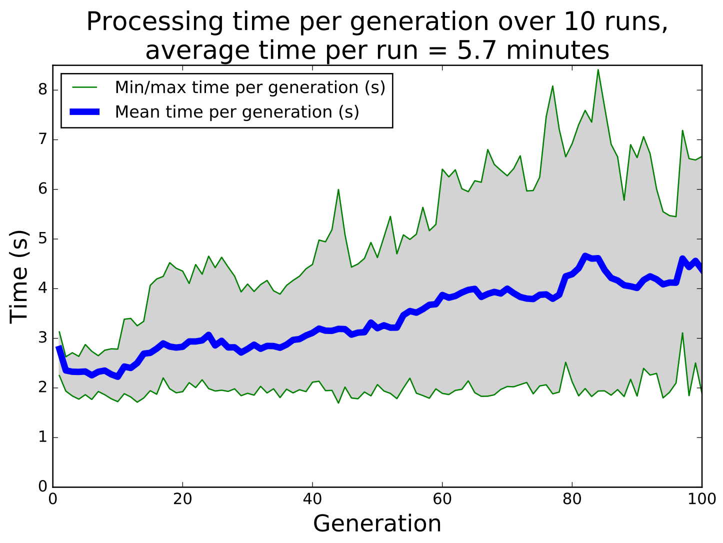


Figure 5.11: Time per generation of single node evolution, 10 runs of 100 generations each, with different starting conditions. The average length of a run is 5.7 minutes. The variation in processing time is due mostly to the size and complexity of the behaviour trees within the population.

in a ready state, the experiment is started. The robots run autonomously, except to listen for the above commands.

5.7 Conclusion

In this chapter we have described the design, development, tuning and programming of the complete Xpuck swarm robot system and accompanying infrastructure. We now have a behaviour tree controller architecture, a fast simulator capable of supporting the required high rate of simulation needed for evolutionary algorithms, the experimental infrastructure needed to provide virtual sensing and data logging, and a swarm of robots.

We next move on to the design of a behaviour tree architecture for the Xpuck robot that is suitable for use with evolutionary algorithms and tries to minimise *reality gap* effects by design, taking a more formal approach than that used for our initial explorations.

Chapter 6

Designing a behaviour tree architecture

In this chapter, we will describe in detail the design process for a behaviour tree architecture targeting a swarm of Xpuck robots, described in Chapter 5. As we note in Chapter 3 we are inspired by the approach and arguments of Francesca *et al.* [2014a] for reducing the effects of the *reality gap*.

A central part of the methodology Francesca *et al.* use is to define the capabilities of the robots as seen by the controller in terms of a *reference model*, and then to define *constituent* behaviour and conditions the controller may use in terms of the reference model. By formalising the model, we define the capabilities required of a minimal simulation and make explicit what might be implicitly assumed. We also define the conditioning of the real robot senses and actuators into a common format.

The use of these constituent behaviours and conditions is a means of motivating the design of the behaviour tree and blackboard; we should be able to express them easily and flexibly within our architecture.

6.1 Robot reference model

The reference model is an abstraction that formally describes the sensors and actuators available to the robot controller. The reference model for the Xpucks is given in Table 6.1. This is formulated using the standard senses and actuators of the base e-puck robot that the Xpuck extends, together with two virtual senses; a compass and a range-and-bearing sense. The input variables are inputs to the controller from the senses. The output variables are what the controller alters to act within the world. The constants define various aspects of the robot abstraction that do not change. The choices about what to abstract away, and what to idealise is driven by both the physical capabilities of the robot, and the necessity of writing a simulator to model these capabilities with sufficient fidelity and speed.

The proximity sensors P_i are a set of eight IR reflectance sensors spaced around the body of the e-puck at the angles given in $\angle q_i$. They are strictly short range, with a maximum range of around p_{max} or 3 cm. They work by turning on an IR emitter and measuring the received level from an IR sensor, and comparing this to the received level when the IR emitter is turned off. They are short-range, vary quite widely between individual sensors and behave differently with different materials. Only objects taller than 30 mm will be detected. In order that it is possible for Xpucks to detect each other it was necessary to 3D print 'skirts' of opaque material that reflect the emitted IR. The standard plastic body of the e-pucks is quite transparent to IR. The raw characteristics can be approximated by an inverse-fourth law relationship with distance, like radar (inverse square from emitter, inverse square from reflector)[1]. We abstract this to a real value between 0 and 1, ranging from a value of 0 when there is no object within the detection range of the sensor, to a value of 1 when when an object is adjacent to the detector.

The camera of the e-puck is VGA resolution and can run at 30 frames per second (FPS), although the base e-puck is only capable of sampling a small subset of each frame. The Xpucks are capable of much more intensive pixel processing such that we can run image processing algorithms. But if we make the camera image processing too complex, the question of how to model this well within the simulator arises. It then becomes a tradeoff between easy to simulate and yet still useful for a swarm robot. We settled upon a scheme of dividing the visual field into three equal vertical segments, within which we recognise the presence of primary coloured pixels above a threshold quantity.

We carefully control the real visual environment to simplify the task of modelling. The arena floor and sidewalls are white. The field of view (FOV) of the cameras is vertically truncated by ignoring pixels above the centre-line, this means that only objects within the arena are perceived. The robots have red 3D printed 'skirts' and are otherwise white. Plastic objects that can be used for foraging tasks are widely available in primary colours, and it is possible to mark areas of the arena walls with coloured paper. The camera sense is thus reduced to just 9 bits of information; the presence or not of red, green, or blue coloured objects within the left, centre, or right hand visual fields.

The compass sense gives the heading of the robot in the world frame. In the case of the Xpucks, this is synthesised virtually, using a Vicon object tracking system and

[1]The inverse square law for signal strength at a distance from a point source is a consequence of the surface area of a sphere $4\pi r^2$ being proportional to the square of the radius, the signal being spread over some fixed proportion of said sphere.

transmitted via WiFi, but this is not necessary, we have a set of BM055 Inertial Measurement Unit (IMU) sensor boards for future use which provide a 9 degree of freedom output (three gyroscope axes, three accelerometer axes, and three magnetic field axes).

A sense of where the neighbouring robots are is generally important for constructing swarm behaviours . One such sense is the range and bearing sense. This captures the number of neighbours n within half a metre r_{max} of the robot, and the distance and bearing to each of them. There are a number of ways that range-and-bearing has been implemented on e-pucks. Using the built-in IR sensors together with a specialised library *libircom* [Gutiérrez *et al.* , 2009a], the design of an additional sensor board [Gutiérrez *et al.* , 2009b], and virtual sensing. We chose to synthesise a virtual range and bearing sense for several reasons. Firstly, experiments showed that the range and bearing performance when using the built-in IR sensors was very poor. Secondly, we did not have any of the range-and-bearing sensor boards. Finally, using virtual sensing allowed us to control the characteristics of the sensor as we chose, setting the maximum range and allowing the injection of various degrees of noise. It is important to note, however, that nothing precludes the use of a real sensor to perform this task, none of the experiments used functionality that could not be constructed in a real sensor.

The only actuators described in the reference model are the two wheels, where the controller can specify their velocities in the range $[-v_{max}, v_{max}]$. When both are set to an equal positive value, the robot will move forward in a straight line. Given that the robot is circular in form with the wheels mounted symmetrically along an axis through the centre of the robot, with the addition of l, the wheelbase, or distance between the wheels, the kinematics are completely described:

$$\begin{aligned} \omega &= \frac{v_{right} - v_{left}}{l} \\ \dot{x}' &= \frac{v_{right} + v_{left}}{2} \end{aligned} \tag{6.1}$$

where $\dot{x}'$ is the velocity in the x' axis of the robot frame of reference. For a derivation, see, for example Siegwart *et al.* [2011].

The controller runs at an update period of $t_{update} = 100\,\text{ms}$. Every update period, the input variables from the sensors are presented to the controller, which then generates new values for the wheel actuator output variables.

Figure 6.1 shows the locations of the proximity sensors and their range, and the camera and its field of field, together with the wheelbase and robot body diameter. The world reference frame is labelled x and y, and the robot frame is shown with

Table 6.1: Robot reference model for the Xpucks

Input variables	Values	Description
$P_{i\in\{1,2,..,8\}}$	$[0,1]$	Proximity sensor i
$R_{i\in\{left,centre,right\}}$	$\{0,1\}$	Red blob detection
$G_{i\in\{left,centre,right\}}$	$\{0,1\}$	Green blob detection
$B_{i\in\{left,centre,right\}}$	$\{0,1\}$	Blue blob detection
θ	$[-\pi,\pi)$	Compass, giving pose angle in world frame
$n\in\mathbb{N}$	$\{0,...,15\}$	Number of neighbouring Xpucks
$(r,\angle b)_{i\in\{1,...,n\},n\neq 0}$	$([r_{min},r_{max}],[-\pi,\pi))$	Range and bearing of neighbour m
Output variables		
$v_{i\in\{left,right\}}$	$[-v_{max},v_{max}]$	Left and right wheel velocities
Constants		
t_{update}	100 ms	Sensor and controller update period
r_{min}	75 mm	Minimum range and bearing range
r_{max}	0.5 m	Maximum range and bearing range
v_{max}	$0.13\,\mathrm{ms}^{-1}$	Maximum wheel velocity
$\angle q_{i\in\{1,2,...,8\}}$	$0.297, 0.855, 1.571, 2.618, -2.618, -1.571, -0.855, -0.297$	Angle of proximity sensor i
p_{max}	30 mm	Proximity sensor maximum range
p_{height}	35 mm	Height of proximity sensors above ground
FOV	56°	Camera field of view
d	75 mm	Diameter of robot
l	53 mm	Wheelbase

axes names x' and y'.

6.2 Constituent behaviours and conditions

All possible tasks $\mathbb{T}$ that the robot may perform are defined implicitly by the reference model, it would not be possible for the robot with the reference model in Table 6.1 to move towards heat, for example, having no sensor capable of perceiving temperature. The set of tasks that the robot can actually perform, $\mathbb{T}'\subseteq\mathbb{T}$, is defined by the controller architecture, that is, the method of mapping of sensor inputs to actuator outputs. We might consider that we want $\mathbb{T}-\mathbb{T}'$ to be as small as possible, but this comes at the cost of increasing generality of the controller.

An important argument made in Francesca *et al.* [2014a] is the relevance of the *bias-variance tradeoff* concept from machine learning to the issue of the reality gap in automatically discovered controllers for swarm robotics. They assert that the reality gap is an example of overfitting due to the high expressiveness of, for example, neural networks as a controller architecture. In machine leaning, high *variance*. By

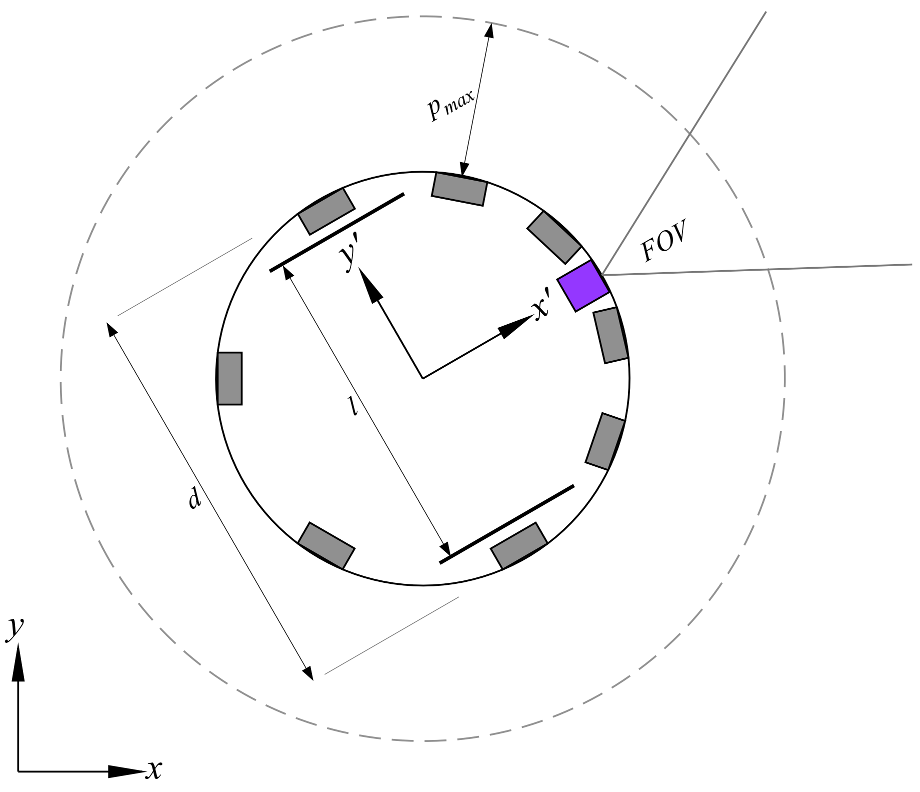


Figure 6.1: Illustration of some of the robot reference model parameters. Proximity sensors are the grey boxes, with a maximum range of p_{max}, the camera is shown with a purple box. The robot frame of reference is shown with the axes labelled x' and y'. The diameter of the robot is given by d and the wheelbase, the distance between the two wheels by l.

increasing the *bias*, or moving in the direction of a less expressive or general controller architecture, they assert that we can reduce the reality gap of the discovered controllers but still maintain the ability to express a useful subset of tasks.

To this end Francesca *et al.* define a controller architecture consisting of a set of constituent behaviours and a set of conditions utilising the reference model for the robot that are then used in the automatic discovery of state machine controllers. The definition of these is regarded as the domain of the *expert*.

We follow a similar path, but one advantage of the BT architecture is that its hierarchical nature allows for behaviours to be composed. In this, we are not limited to choosing explicitly the constituent behaviours. We can instead define a set of primitive action nodes and blackboard registers and then construct the constituent behaviours as subtrees. These subtrees could either be used as fixed elements within the evolutionary algorithm, or potentially be themselves evolvable, in the manner of Automatically Defined Functions (ADFs) [Koza, 1992]. By changing these constituent behaviours we can tune our location along the *bias-variance* tradeoff.

Francesca *et al.* define six behaviours and six conditions. The behaviours are: *Exploration* - The robot moves in a straight line until one of the front proximity detectors is triggered, when it turns on the spot away from the triggered sensor for a random number of timesteps. *Stop* - The robot stays still. *Phototaxis* - The robot moves towards any detected light or straight. Obstacles are avoided. *Anti-phototaxis* - The robot moves away from detected light or straight, with obstacle avoidance. *Attraction* - The robot moves towards other robots, with obstacle avoidance, and *Repulsion* - the robot moves away from other robots, with obstacle avoidance. *Attraction* and *Repulsion* have a parameter determining how strong the respective force is, and *Exploration* has a parameter determining the maximum number of timesteps the robot will turn for. The conditions are: *Black-floor*, *Gray-floor*, *White-floor*, *Neighbour-count*, *Inverted-neighbour-count*, and *Fixed-probability*. Each of these being true enables a state machine transition with probability controlled by a parameter.

Given the different reference model for the Xpuck, we obviously cannot replicate the same set of behaviours and conditions, but we hope that the ones we have chosen are in the same spirit, providing useful building blocks but with sufficient granularity to reduce *reality-gap* effects. Using these building blocks, we will then try and define a set of action nodes and blackboard registers with which to implement the building blocks as subtrees. The process of implementing the behaviours will inform the choice of action nodes, and that will also inform further iterations of constituent behaviours.

This is quite akin to the process of designing the Instruction Set Architecture (ISA) of a computer processor; there is a tension between more complex instructions that express ideas compactly, but at the expense of flexibility, and simpler, more regular instructions that can be used to construct the more complex instructions but perhaps take more space. Perhaps we are seeking elegance, simple enough but no simpler.

6.2.1 Constituent behaviours

The following is our starting point for the constituent behaviours, based heavily on Francesca *et al.* [2014a].

Exploration: The robot moves straight forward. Obstacle avoidance is performed in the following way: If the vector sum of the individual proximity sensors, given by

$$\boldsymbol{v_{prox}} = \sum_{i=1}^{8} (P_i, q_i) \tag{6.2}$$

faces forwards i.e. $\boldsymbol{v_{prox}} \cdot \hat{\boldsymbol{x}} > 0$ and $|\boldsymbol{v_{prox}}| > 0.1$, there is an obstacle in front of the robot, so the robot turns on the spot for a random time bounded by τ in the direction away from $\boldsymbol{v_{prox}}$.

Stop: Stop moving.

Upfield: Move towards the $+x$ end of the arena, with obstacle avoidance embedded. The robot follows the vector $\boldsymbol{v}$, where $k = 5$ and $\boldsymbol{v_{up}}$ is the unit vector away from the home end of the arena.

$$\boldsymbol{v} = \boldsymbol{v_{up}} - k\boldsymbol{v_{prox}} \tag{6.3}$$

Downfield: Move towards $-x$ end, with obstacle avoidance embedded. The robot follows the vector $\boldsymbol{v}$, where $k = 5$ and $\boldsymbol{v_{up}}$ is the unit vector away from the home end of the arena.

$$\boldsymbol{v} = -\boldsymbol{v_{up}} - k\boldsymbol{v_{prox}} \tag{6.4}$$

Attraction/Repulsion: Move towards or away from robots in the neighbourhood, as detected by the range and bearing data, with obstacle avoidance. The robot follows the vector $\boldsymbol{v}$. Degree of attraction tuned by the parameter α, when α is positive the attracted to neighbours, when negative, repulsed. When there are no neighbours, the robot moves forward.

$$\begin{aligned}\boldsymbol{v_{rb}} &= \begin{cases} \alpha \sum_{i=1}^{n} (\frac{r_{min}}{r_i}, \angle b_i), & \text{if } n > 0 \\ (1, \angle 0), & \text{otherwise} \end{cases} \\ \boldsymbol{v} &= \boldsymbol{v_{rb}} - k\boldsymbol{v_{prox}} \end{aligned} \tag{6.5}$$

6.2.2 Constituent conditions

The conditions defined by Francesca *et al.* enable probabilistic state transitions depending on certain sensor inputs. The equivalent with a BT is a subtree that probabilistically returns *success* based on sensor conditions. The Xpuck does not have ground greyscale sensors but does have a camera, this is used for the red, green, and blue conditions.

Red: If $R_i = 1$, for any $i \in \{left, centre, right\}$, return *success* with probability β.

Green: If $G_i = 1$, for any $i \in \{left, centre, right\}$, return *success* with probability β.

Blue: If $B_i = 1$, for any $i \in \{left, centre, right\}$, return *success* with probability β.

Neighbour-count: Return *success* with probability:

$$z(n) = \frac{1}{1 + e^{k(l-n)}} \tag{6.6}$$

where n is the number of robots in the neighbourhood, k is steepness of the function, and l is the number of robots to give $P = 0.5$.

Inverted-neighbour-count: Return *success* with probability $1 - z(n)$, with $z(n)$ defined as in Eqn. 6.6.

Fixed-probability: Return *success* with probability β.

Table 6.2: Behaviours, conditions, and their parameters

Behaviour	Param	Range	Description
Exploration	τ	$\{1, 2, .., 100\}$	Explore, turn away if obstacle for τ steps
Stop			Stop moving
Upfield			Move towards $+x$ end, with obstacle avoidance
Downfield			Move towards $-x$ end, with obstacle avoidance
Attraction	α	$[0, 15.875]$	Move towards neighbouring robots, with obstacle avoidance
Repulsion	α	$[0, -16]$	Move away from neighbouring robots, with obstacle avoidance
Condition			
Red	β	$[0, 1]$	Probabilistic success if red seen
Green	β	$[0, 1]$	Probabilistic success if green seen
Blue	β	$[0, 1]$	Probabilistic success if blue seen
Neighbour count	k	$[0, 15.875]$	Probabilistic success based on neighbour count
	l	$[0, 15]$	
Inv-neighbour count	k	$[-16, 0]$	
	l	$[0, 15]$	
Fixed probability	β	$[0, 1]$	Fixed probabilistic success

6.3 Blackboard and action nodes

Designing a behaviour tree architecture for a particular reference model involves defining the *blackboard* and the action nodes. Composition nodes are universal, we implement the ones detailed in Table 3.1. Since `seq*` and `sel*` nodes of arbitrary arity can be synthesised by nesting, we just provide small fixed variants.

There are similarities with designing an instruction set architecture for a CPU; balancing the desire for an elegant and orthogonal set with the demands of the specific behaviours we want to be able to express. Of course we can express any behaviour with a set of very simple behaviours that amount to effectively machine code, but this is probably not a good target for evolutionary algorithms (too many possibilities with no functionality), and also likely to be towards the wrong end of the bias-variance tradeoff, being capable of too high an expressiveness. So what we want is a set of orthogonal and meaningful[2] behaviours that can be combined to easily produce the constituent behaviours and conditions described Section 6.2. In some ways, the blackboard/action node split is reminiscent of the Move machine processor architecture, where an extremely minimal instruction set, even as little as just a single conditional move, achieves functionality by mapping that functionality onto addresses that are the source and destination of the move [Lipovski, 1976; Tabak & Lipovski, 1980]. The blackboard entries encapsulate certain basic functionality, and the action nodes are used to operate on that.

Many of the constituent behaviours detailed above involve 2D vectors, for motion and for sensing. For this reason, the first major design decision was to make the natural handling of vectors a central element of the nodes and blackboard.

The blackboard consists of a set of 32 bit floating point registers addressed by an integer index. Some blackboard entries represent two component vectors, and these entries have even indices, with B_{2i} representing the first (x) component of the vector, and B_{2i+1} representing the second (y) component of the vector. Some blackboard entries represent scalar quantities, and these entries can have any value indices. It is possible to access a vector component of the blackboard as a scalar by use of the appropriate index.

The register pair at index $0, 1$ will always read as zero. There is a scalar and a vector scratch register which can be written and read and serve as a memory. They have no direct effect on the environment.

Blackboard entries may be linked to physical actuators. These can only be updated by one action node per controller update cycle. The first action node to access a physically linked register will return `success`, any subsequent access within that update will return `running`, indicating that the register is not yet available. Any physically linked register not written in an update cycle will assume a default value, not maintain any previously set value. Currently the only physically linked entry is the $\boldsymbol{v_{goal}}$ register.

[2]in terms of xpuck behaviour

All vector quantities are considered to be in the robot reference frame, which defines the $+x$ axis as the forward facing direction i.e. the direction the camera points and the direction in which the robot will move if identical positive velocities are sent to the two motors. The vector $\mathbf{(1,0)}$ or $\hat{\boldsymbol{x}}$ is thus the unit vector pointing forwards, the vector $\mathbf{(0,1)}$ or $\hat{\boldsymbol{y}}$ points out of the left hand side of the robot, see Figure 6.1.

Table 6.3 summarises the blackboard, we will now talk in some detail about the design decisions involved.

6.3.1 Goal velocity, physically linked registers

Probably the most interesting register is the goal velocity vector, $\boldsymbol{v_{goal}}$. It is through this that the BT will actuate the robot. Fundamentally, we want a simple but flexible motion control. If we copy a vector quantity into this register, we are saying that we want the robot to move in that direction with a speed related to the magnitude of the vector.

We stated above that a physically linked blackboard register may be written to by only one node per update cycle, and if it is not written to, it should assume a zero default value. Why? Firstly, why can only one node per update cycle write to a physically linked node. Consider Figure 6.2. A memory sequence node with six

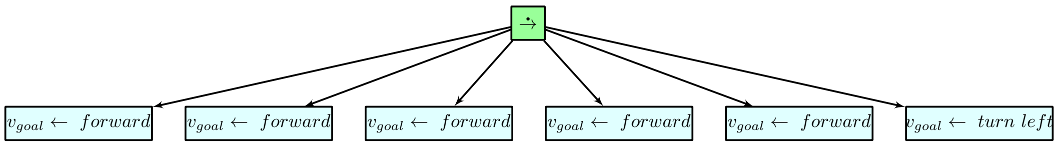

$\overset{*}{\rightarrow}$
$v_{goal} \leftarrow forward$
$v_{goal} \leftarrow forward$
$v_{goal} \leftarrow forward$
$v_{goal} \leftarrow forward$
$v_{goal} \leftarrow forward$
$v_{goal} \leftarrow turn\ left$

Figure 6.2: A behaviour tree to move in a polygon

children, five of which command forward move, and a final that commands a left turn. If there was no restriction on writing to the goal register in an update, the first child node would be ticked, then when that returned `success` the next node would be ticked and so on until all the nodes had been ticked in a single update cycle, with the final value for the update cycle being ‘turn left’. The robot would just continually spin anticlockwise on the spot. This is clearly not the intent, and it will generally be the case that registers linked to some physical process will require time to take effect. By ensuring that only the first node that writes to a physical register can return `success` and any subsequent writes within that cycle return `running`, the natural intent of the tree is produced; the robot moves forward for five update cycles, then turns left for one update cycle, repeating every six cycles.

Why use a zero default value? Lets consider a simple behaviour tree that makes the robot move forward if there is nothing detected by the proximity sensors. Two ways to write the tree are shown in Figure 6.3, the left hand tree assumes that the goal register has a default zero, and the right hand tree shows what is necessary if there

Table 6.3: Blackboard registers. Read-only registers can be used as a destination without error but will be unchanged.

Index	Name	Access	Description
0	$zero$	R	Zero. This register will always read as a zero vector or scalar and will not be affected by writes
2	$\boldsymbol{v_{goal}}$	RW	Goal velocity vector. Writing to this register determines how the robot will move. If the vector points backwards, the robot will turn on the spot in a direction determined by the sign of the y component of the vector. If it points forwards, the robot will move forwards while turning to minimise the angle between the vector and the forward direction of the robot. The speed the robot moves at will be determined by the magnitude of the vector, with any magnitude larger than 1 giving the maximum wheel velocities The motor velocities are determined as follows: At the start of a *tick*, $\boldsymbol{v_{goal}} \leftarrow (0,0)$ At the end of a *tick*, the vector may have been updated. If it is non-zero, it is transformed to wheel motor velocities by: $s = \begin{cases} 1, & \text{if } \lvert\boldsymbol{v_{goal}}\rvert < 1 \\ \lvert\boldsymbol{v_{goal}}\rvert, & \text{otherwise} \end{cases}$ $\boldsymbol{v'_{goal}} = \frac{1}{s}\begin{cases} (0, sgn(\boldsymbol{v_{goal}} \cdot \boldsymbol{\hat{y}})\lvert\boldsymbol{v_{goal}}\rvert), & \text{if } \boldsymbol{v_{goal}} \cdot \boldsymbol{\hat{x}} < 0 \\ \boldsymbol{v_{goal}}, & \text{otherwise} \end{cases}$ $\boldsymbol{v''_{goal}} = R(\angle 45) \times \boldsymbol{v'_{goal}}$ $(v_{left}, v_{right}) = v_{max}\boldsymbol{v''_{goal}}$
4	$\boldsymbol{v_{prox}}$	R	Proximity vector. Reading this register returns the vector sum of all the proximity sensors, where each sensor has magnitude varying from 0 if there is nothing within the sensor range p_{max}, to 1 if there is something directly adjacent to the sensor. $\boldsymbol{v_{prox}} = \sum_{i \in \{1,\ldots,8\}} (P_i, \angle q_i)$
6	$\boldsymbol{v_{up}}$	R	Upfield vector. This is the unit vector pointing upfield, that is, towards the $+x$ end of the arena. Given by: $\boldsymbol{v_{up}} = \boldsymbol{(1, -\angle\theta)}$ where θ is the pose angle of the robot in the world frame.
8	$\boldsymbol{v_{attr}}$	R	Attraction vector. The points towards the nearest concentration of neighbouring robots. It is formed from the range-and-bearing system information by: $\boldsymbol{v_{attr}} = \begin{cases} \sum_{i=1}^{n}(\frac{r_{min}}{r_i}, \angle b_i) & \text{if } n > 0 \\ (1, \angle 0) & \text{otherwise} \end{cases}$
10	$\boldsymbol{v_{red}}$	R	Red vector. This points towards the detected red blobs within the camera FOV. It is given by: $\boldsymbol{v_{red}} = (R_{left}, \angle 18.7) + (R_{centre}, \angle 0) + (R_{right}, \angle -18.7)$
12	$\boldsymbol{v_{green}}$	R	Green vector, calculated as with red
14	$\boldsymbol{v_{blue}}$	R	Blue vector, calculated as with red $\boldsymbol{v_{blue}} = (B_{left}, \angle 18.7) + (B_{centre}, \angle 0) + (B_{right}, \angle -18.7)$
16	s_n	R	Neighbour count. The number of neighbours detected by the range-and-bearing system
17	s_{scr}	RW	Scratchpad
18	$\boldsymbol{v_{scr}}$	RW	Scratchpad

is no default. Clearly, the left version feels more natural and is simpler.

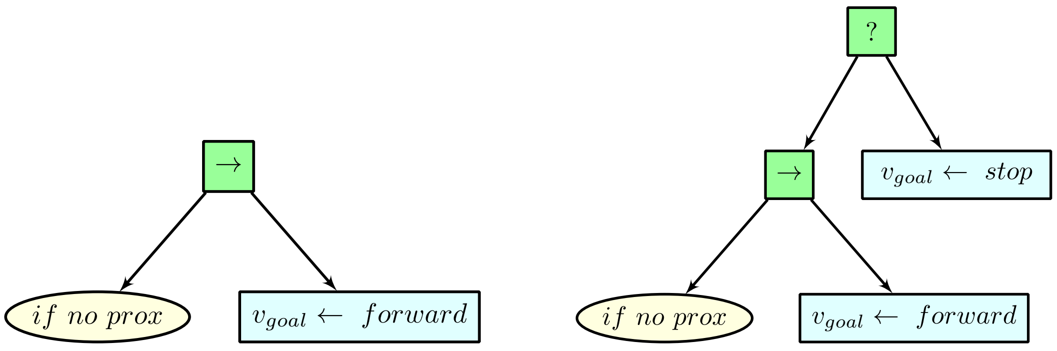

→
if no prox
$v_{goal} \leftarrow$ forward
?
→
$v_{goal} \leftarrow$ stop
if no prox
$v_{goal} \leftarrow$ forward

Figure 6.3: Behaviour tree to move forwards while there is nothing detected by the proximity sensors, with default to zero on left, and keep last value on right.

We can make the comparison to an FSM with multiple states, some of which move the robot. The state of the FSM dictates what is sent to the actuator, not the previous state of the actuator. It is the same for a behaviour tree, but here the state is implicit in the path to activation of a node writing the register for the first time in an update cycle.

6.3.2 Steering

Given a vector in the world frame to follow, how do we steer? The e-puck robot which the Xpucks are based on has two-wheel differential drive kinematics; it is non-holonomic and cannot just move in any direction, but needs a steering strategy to reach a goal velocity or position in the world frame. One simple strategy with a fixed goal vector for a two-wheel robot is to turn until the pose angle of the robot is the same as the angle of the vector, then to move in a straight line along the vector. But this does not handle changing vectors well since movement needs to be stopped for reorientation to take place.

We want the movement to be smooth. We also need a completely reactive strategy, we can't use a top-down approach of route-planning followed by constrained trajectory generation and play out since there is no overall endpoint target. Fajen & Warren [2003] studied human navigation towards a goal in the presence of obstacles and found that people tried to minimise the difference between their heading and the the goal heading but nearby obstacles in their path caused repulsive changes in heading away from the goal until the obstacle was passed. Routes emerged as a consequence, rather than being planned. Huang *et al.* [2006] take this insight and develop a steering law for non-holonomic robots that produces smooth paths. Park & Kuipers [2011] consider the case of smooth trajectory generation for wheelchairs, where obstacles are dynamically changing, and the comfort of the user requires that acceleration or jerk (change in acceleration) is limited.

This motivates an initial design where the assumption is that there will be a continuous updating of the velocity goal vector with the current goal in the robot frame. The trajectory followed should vary smoothly.

Let us suppose that we want the robot to move in the $+x$ world frame direction, represented by the blackboard register $\boldsymbol{v_{up}}$, and to achieve this, we have a simple behaviour tree consisting of a single action node that continually performs $\boldsymbol{v_{goal}} \leftarrow \boldsymbol{v_{up}}$.

All blackboard vectors are in the robot frame, so consider what that means for the simple example. Start with the robot facing in the $+y$ direction, having a angle in the world frame of $\frac{\pi}{2}$. The register $\boldsymbol{v_{up}}$ will have a value of $\boldsymbol{(1, -\angle\frac{\pi}{2})}$, a unit vector pointing to the right hand side of the robot. Intuitively, as humans, we would start to turn to the right, and also move forward, increasing our forward motion and reducing our turning as we became better aligned to the target.

The steering behaviour was originally written in an *ad-hoc* way to directly generate wheel velocities from the goal vector v_{goal} based on this intuition. We motivated this with the following requirements:

1. The magnitude of the vector controls the motor velocity. A zero magnitude vector will produce no motion, a vector of magnitude 1 produces the maximum motor velocity. Any vector larger than 1 is normalised to magnitude 1.
2. The direction of the vector controls the robot direction. A vector in the $\boldsymbol{(+x, 0)}$ direction will activate both motors identically, moving forwards. Increasing vector angle away from 0 will produce higher turning rate, ω, and lower forward velocity v.
3. If the vector points behind the robot, the robot will rotate in place, not move backwards. The idea behind this is to keep the direction of motion predominately forward, where the camera and most proximity sensors point.

These requirements were formalised as the following equations for wheel velocities v_{left} and v_{right}, given the input goal vector $\boldsymbol{v_{goal}}$:

$$s = \begin{cases} 1, & \text{if } |\boldsymbol{v_{goal}}| < 1 \\ |\boldsymbol{v_{goal}}|, & \text{otherwise} \end{cases} \tag{6.7}$$

$$\boldsymbol{v'_{goal}} = \frac{1}{s} \begin{cases} (0, sgn(\boldsymbol{v_{goal}} \cdot \boldsymbol{\hat{y}})|\boldsymbol{v_{goal}}|), & \text{if } \boldsymbol{v_{goal}} \cdot \boldsymbol{\hat{x}} < 0 \\ \boldsymbol{v_{goal}}, & \text{otherwise} \end{cases} \tag{6.8}$$

$$\begin{bmatrix} v_{left} \\ v_{right} \end{bmatrix} = v_{max} \begin{bmatrix} \cos\frac{\pi}{4} & -\sin\frac{\pi}{4} \\ \sin\frac{\pi}{4} & \cos\frac{\pi}{4} \end{bmatrix} \times \boldsymbol{v'_{goal}} \tag{6.9}$$

The goal vector is normalised to unit magnitude if it is greater magnitude than 1, and if the vector points to the back of the robot $x < 0$ it is rotated (keeping the same magnitude) to the nearest $x = 0$ location, Eqn 6.8. Noting that a unit $\hat{\boldsymbol{x}}$ vector should activate both motors equally, and rotating by $\frac{\pi}{4}$ gives x and y both equal to $\frac{\sqrt{2}}{2}$, we use this rotation, with a scaling to the maximum allowed motor velocity, to transform the goal into motor velocities.

In order to briefly but more formally analyse the steering law, it is best expressed in polar form. Let $\boldsymbol{v_{goal}} \equiv \boldsymbol{(r, \phi)}$ be the goal vector expressed as magnitude r and heading in the robot frame ϕ. l is the distance between the wheels. Firstly, get the wheel velocities:

$$\boldsymbol{v_{goal}} \equiv \boldsymbol{(r, \phi)} \tag{6.10}$$

$$r' = f(r) = \begin{cases} r, & \text{if } r < 1 \\ 1, & \text{otherwise} \end{cases} \tag{6.11}$$

$$\phi' = g(\phi) = \begin{cases} \frac{\pi}{2}, & \text{if } \phi > \frac{\pi}{2} \\ -\frac{\pi}{2}, & \text{if } \phi < -\frac{\pi}{2} \\ \phi, & \text{otherwise} \end{cases} \tag{6.12}$$

$$\begin{bmatrix} v_{left} \\ v_{right} \end{bmatrix} = v_{max} \begin{bmatrix} \cos\frac{\pi}{4} & -\sin\frac{\pi}{4} \\ \sin\frac{\pi}{4} & \cos\frac{\pi}{4} \end{bmatrix} \times \begin{bmatrix} r'\cos\phi' \\ r'\sin\phi' \end{bmatrix} \tag{6.13}$$

Then express as linear and angular velocities:

$$v = r' v_{max} \sqrt{2} \cos\phi' \tag{6.14}$$

$$\omega = \frac{r' v_{max} \sqrt{2}}{l} \sin\phi' \tag{6.15}$$

Figure 6.4 shows the commanded velocity and angular velocity resulting from Eqns 6.14 and 6.15. This shows similarity to the result of Fajen & Warren [2003] that the turning rate is roughly proportional to difference in heading of the robot and the goal.

We will implement this steering law using a discrete time controller, with an update rate of t_{update} as specified in the robot reference model. A basic rule-of-thumb for a discrete time controller to work correctly and be stable is that the highest system frequency is preferably less than half Nyquist, or a quarter the sampling rate. Given a controller update rate of 10 Hz, this gives a maximum frequency of ≈ 2.5 Hz. The system frequency in this case is given by ω and the maximum value from Eqn 6.15 of $\omega = \frac{0.13 \times 0.707}{0.053} = 1.7\,\text{rad/s} \approx 0.3\,\text{Hz}$, comfortably meeting the requirement. Another point to consider with our system is that there is a latency $t_{latency}$ associated with certain virtual senses. If this latency is included, is the system stable? We ran simulations to test the behaviour, using the discrete time equations below.

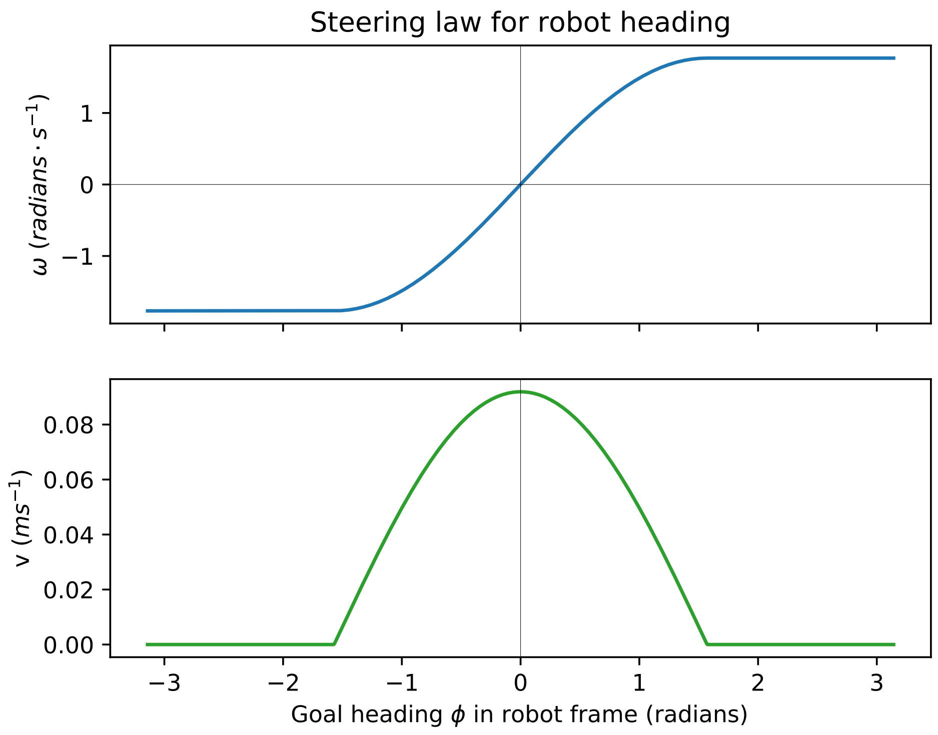


Figure 6.4: Steering law for robot. Velocity and angular velocity for a given goal vector heading angle in the robot frame. When the heading is zero, the robot moves forward, as the heading increases, the forward velocity smoothly decreases and the angular velocity smoothly increases, until the robot is turning on the spot at a vector angle of $\pm\pi/2$.

At every control step $s \in \{0, 1, ...\}$ where $t_s = s \cdot t_{update}$ we calculate new velocities, but the goal is based on the robot heading at time $t - t_{latency}$ to account for the latency between actual position of the robots and the supply of virtual senses. We also introduce the additional gain term $k_{\omega gain}$ to get a feel for the steering stability margin.

$$\phi' = g(\phi(t - t_{latency})) \tag{6.16}$$

$$v_s = \frac{v_{max} \cdot r' \cdot \sqrt{2} \cos \phi'}{2} \tag{6.17}$$

$$\omega_s = \frac{k_{\omega gain} \cdot v_{max} \cdot r' \cdot \sqrt{2} \sin \phi'}{l} \tag{6.18}$$

These are integrated to produce the robot pose using basic Euler integration with

$\Delta t = 1\,\text{ms}$.

$$x_{i+1} = x_i + v_s \cos\theta \Delta t \tag{6.19}$$

$$y_{i+1} = y_i + v_s \sin\theta \Delta t \tag{6.20}$$

$$\theta_{i+1} = \theta_i + \omega \Delta t \tag{6.21}$$

The world frame goal heading is switched between $-\frac{\pi}{4}$ and $\frac{\pi}{4}$ to give the system step response. We simulate with a virtual sensor latency $t_{latency} = 0.25\,\text{s}$, obtained from measuring the real system, see later chapters, and various rotational velocity gains $k_{\omega gain}$.

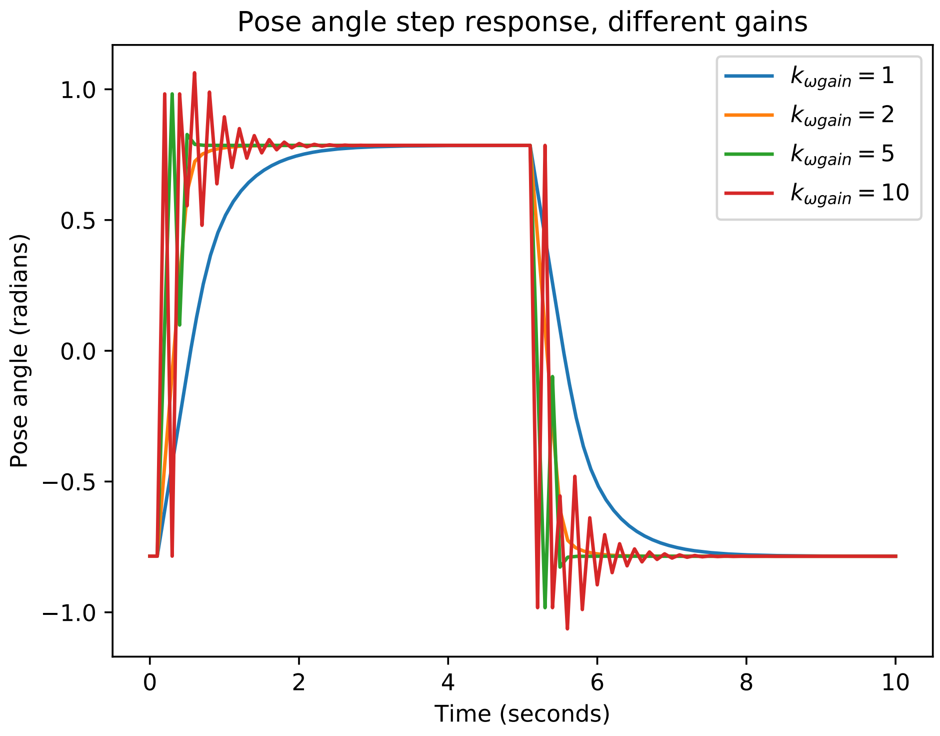
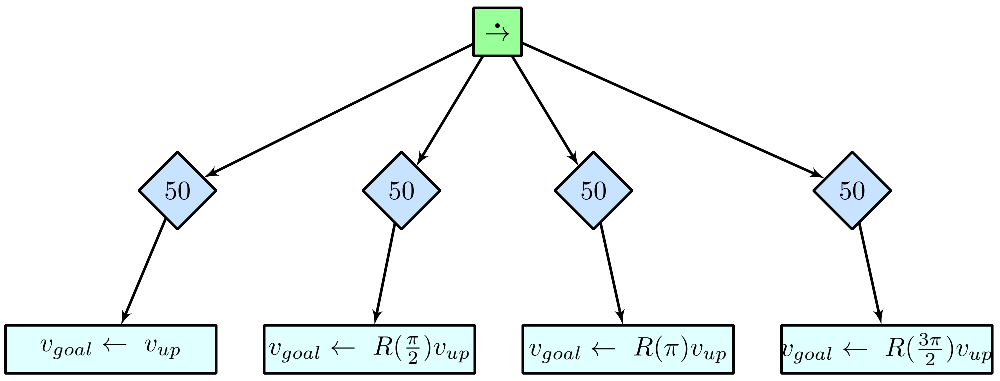


Figure 6.5: Robot pose angle step response with different gains

The simulated system behaviour is shown in Figure 6.5. The system is stable at a gain of 1, with critically damped behaviour somewhere between gains of 2 and 5. By a gain of 10, corresponding to a maximum $\omega = 17.7 = 2.8\,\text{Hz}$, the system is close to oscillation, broadly confirming the rule-of-thumb estimate above.

Finally, we simulate the system with a contrived but possible scenario, shown in Figure 6.6. This "square" behaviour tree sets $\boldsymbol{v_{goal}}$ to be the $+x$, $+y$, $-x$, and $-y$ directions for 50 updates or 5 seconds each, one after the other. The robot should trace an approximate square while moving in an anticlockwise direction. The results



$\dot{\rightarrow}$
50
50
50
50
$v_{goal} \leftarrow v_{up}$
$v_{goal} \leftarrow R(\frac{\pi}{2})v_{up}$
$v_{goal} \leftarrow R(\pi)v_{up}$
$v_{goal} \leftarrow R(\frac{3\pi}{2})v_{up}$

Figure 6.6: Move in an approximate square, taking 5 seconds per side. $R(\theta)$ represents a rotation by the angle θ.

of this simulation are shown in Figure 6.7, with the pose of the robot plotted every second and with the starting pose at location (0,0) facing $-y$. At each corner, see the zoomed figure, when the heading changes we can see linear velocity reduced to zero and angular velocity increased as the turn starts, with the linear velocity gradually increasing again as the heading starts to approach the goal heading. The steps in the lower graph also shows clearly the effect of the 100 ms control update cycle on the recalculation of new wheel velocities.

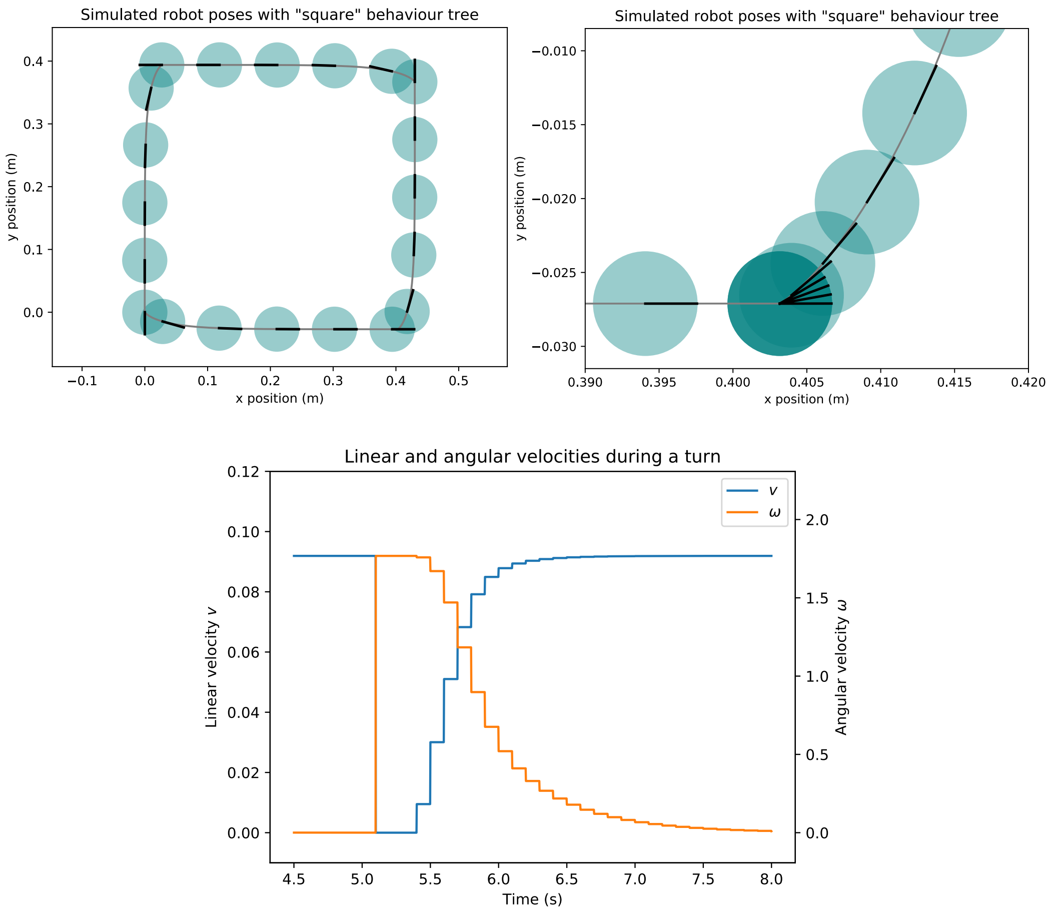


Figure 6.7: Simulation showing the pose of a robot while running the "square" behaviour tree. Pose position is the centre of the shaded circles, and pose angle is indicated by the bold line, with the grey line showing the trajectory. The robot starts with pose $(x, y, \theta) = (0, 0, -\frac{\pi}{2})$ and moves in an anticlockwise direction. The top graph shows the robot body actual size with poses each second, the centre graph shows the lower right corner at 0.1 second intervals with the pose indicator shrunk for clarity, and the bottom graph shows the linear and angular velocities during the turn. The steps in the curves are due to the controller update period of 100 ms.

6.3.3 Sensors

We now cover the remaining blackboard registers, which are mostly associated with sensor input conditioned in various ways. They are designed in order to easily construct the constituent behaviours and conditions detailed in Section 6.2. In summary, the behaviours are *explore*, *upfield/downfield*, and *attraction/repulsion*. The conditions related to sensors are *red, green, blue*, and *neighbour count*.

Exploration, as encapsulated in the *explore* behaviour, involves moving forward unless there is an obstacle in the way, in which case the robot turns away from the obstacle for a random amount of time. The sense required for this is a proximity sense. As defined by the reference model, each IR proximity sensor returns a value between 0, for nothing in range, to 1, for an obstacle adjacent to the sensor. We turn this into the blackboard vector $\boldsymbol{v_{prox}}$ by summing the vectors comprising the magnitude and angle of each of the individual sensors. This vector will tend to point towards the nearest obstacle, with a magnitude related to how close the obstacle is. We say 'tend' because it would be possible for there to be obstacles equally spaced on opposite sides of the robot, resulting in a zero magnitude vector. This approach of using the vector sum of the IR sensors is commonly used.

Moving *upfield* or *downfield* with embedded avoidance needs a blackboard register containing the orientation of the world frame relative to the robot, as well as the proximity register $\boldsymbol{v_{prox}}$. This is the register $\boldsymbol{v_{up}}$. The *upfield* direction corresponds to the $+x$ direction, or a world frame pose angle of zero. Since the base e-puck robot does not have a compass, we implement this sense virtually, using the known absolute pose of the robots from the Vicon tracking system. The pose angle θ is turned into a unit vector $\boldsymbol{(1, \angle - \theta)}$, transforming the pose angle to the robot reference frame. Due to the way the Vicon system processes data, there is a latency of approximately 200 ms between the real pose angle changing, and this being reflected in the value of the $\boldsymbol{v_{up}}$ register at the robot.

Attraction and repulsion towards or away from concentrations of neighbouring robots is done by constructing the vector $\boldsymbol{v_{attr}}$ from the range and bearing information detailed in the reference model. As with $\boldsymbol{v_{up}}$, the robots have no built-in range and bearing sense, this this is synthesised from the Vicon tracking system. The vector is constructed similarly to the method used by Francesca *et al.* [2014a]. One difference is as a result of a bug in a simplification implementing this. Francesca *et al.* [2014a] define the *attraction* behaviour to move towards the nearest neighbouring robots, or move forward if there are no neighbours, and the *repulsion* behaviour to move away from nearest neighbours or forward if no neighbours. Our intent was to use a single vector blackboard register, and multiply this by positive or negative constants to get attraction and repulsion behaviour but the vector included the 'move forward otherwise' unit $+x$ vector. This results in the repulsion behaviour actually using a unit $-x$ vector with no neighbours, resulting in 'rotate anticlockwise if no neighbours' behaviour. We consider the effects of this bug later.

Next, we have the constituent conditions of *red, green, blue* seen by the camera. Since the camera vision system, as described by the reference model, divides the field of view into three segments, *left*, *centre*, and *right* with detection or absence for each, we choose to regard each bit as a unit vector of different direction, based on the field of view of the camera, and then form the vector sum. A vector pointing through the centre of the left-hand segment will be at an angle of 18.7°, one third of the 56° FOV of the camera. Likewise, the centre segment is at $\angle 0$, and the right-hand segment at $\angle{-18.7°}$. This formulation means that to move towards a blue object, for example, a suitable behaviour tree could simply consist of $\boldsymbol{v_{goal}} \leftarrow \boldsymbol{v_{blue}}$.

Table 6.4 shows all the possible patterns of detection for a single colour, and the resultant vectors, and Figure 6.8 shows the same information diagrammatically.

Table 6.4: List of all possible vectors in polar and rectangular form for one colour from the camera system.

Left	Centre	Right	Vector (polar)	Vector (rect)
0	0	0	$(0.00, \angle 0.00)$	$(0.00, 0.00)$
0	0	1	$(1.00, \angle{-18.7°})$	$(0.947, -0.321)$
0	1	0	$(1.00, \angle 0.00°)$	$(1.00, 0.00)$
0	1	1	$(1.98, \angle{-9.35°})$	$(1.95, -0.321)$
1	0	0	$(1.00, \angle 18.7°)$	$(0.947, 0.321)$
1	0	1	$(1.89, \angle 0.00°)$	$(1.89, 0.00)$
1	1	0	$(1.98, \angle 9.35°)$	$(1.95, 0.321)$
1	1	1	$(2.95, \angle 0.00°)$	$(2.95, 0.00)$

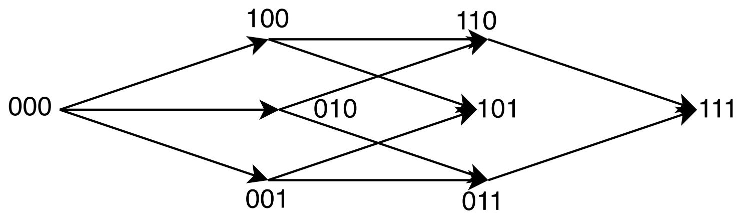


Figure 6.8: Diagrammatic view of all possible vectors from the camera system. Binary labels correspond to $\{left, centre, right\}$ colour detection.

Finally we have the *neighbour count* of nearby robots. This is a scalar value and as such simply occupies a single blackboard address. It is the total number of robots that are currently within range of the range-and-bearing system. As defined by the reference model, this is the count of all robots within a range of 0.5 m.

6.3.4 Zero and scratchpad

The last blackboard entries are the zero and scratchpad registers. The zero register $\boldsymbol{v_{zero}}$ is defined for similar reasons that a fixed zero is often defined in processor architectures, because it is a very commonly used value and means that more generic action nodes can be defined which with the use of $\boldsymbol{v_{zero}}$ become more specific. This essentially simplifies and regularises the set of action nodes that need to be defined in order to implement the desired constituent behaviours. It can be written to with no side effects.

The scratchpad registers, one scalar s_{scr} and one vector $\boldsymbol{v_{scr}}$ are memories. They are writeable, and maintain the last written value between update cycles. The intent of adding memory is to potentially allow the evolution of more complex behaviour tree controllers than might otherwise be possible. An analogy would be between a feed-forward and a recurrent neural network. We make no assumptions about what use evolution may make of these registers.

6.4 Action (leaf) nodes

The leaf nodes of the behaviour tree interface to the blackboard and can alter it or test conditions against it. As with all behaviour tree nodes, they return either *success*, *failure*, or *running*. We define a subset the action nodes called *query* nodes. A query node has no effect on the state of the blackboard, and never returns *running*. Action nodes may, but do not have to, result in a change to the blackboard, that is, they may act on the environment.

The complete set of ten action nodes is summarised in Table 6.5. The set of nodes above the line write a result into a destination blackboard register. The first two, `movcs` and `movcv`, write a constant value into either a scalar or a vector blackboard destination register respectively. They each take two parameters, a destination index d and an 8 bit constant i. In the case of `movcs` i is interpreted simply as a signed integer value in the range $[-128..127]$.

For `movcv`, what we choose to represent with i is a unit vector at a particular angle ranging from $-\pi$ for $i = -128$ to $\pi\frac{127}{128}$ for $i = 127$. The reasoning for choosing this restricted set of representable constant vectors, rather than allowing a two component formulation of arbitrary vectors is as follows. Firstly, we are working within the domain of evolutionary algorithms. If we allow arbitrary vectors, say by having two 32 bit floating point parameters, we vastly increase the search space for no gain; most numbers are probably not useful since the velocity goal blackboard entry $\boldsymbol{v_{goal}}$ saturates at a vector magnitude of 1. Secondly, the size of the nodes is an important design consideration; we intend to run many simulations using behaviour tree controllers in parallel, it takes 8 bytes for two floating point numbers versus 1 bytes for a unit vector at various angles. We want expressiveness, but not too much, and we want space efficiency as well. Again, the parallel to ISA design is apposite.

The next two leaf nodes, `mulas` and `mulav`, are used to manipulate blackboard registers. They are both of the form $d \leftarrow s_1 + f s_2$, where d, s_1, s_2 are blackboard registers, and f is a 32 bit floating point constant. `mulas` acts on scalar registers and `mulav` on vector registers. These could be regarded as rather complicated instructions, performing the addition of one source operand to the result of a constant multiplication of a second source operand, before writing to the destination register. Because we can use the $\boldsymbol{v_{zero}}$ or s_{zero} register as a source operand, together with different choices of constant we can perform addition, constant multiplication, or register move.

Finally the `rotav` instruction adds one source vector register s_1 to a second source vector register s_2 rotated by a constant angle defined by a signed 8 bit value i. The result is then stored in the destination vector register d. As with `movcv` the angle is defined as $\pi\frac{i}{128}$. Together, these five leaf nodes allow us to create, scale, translate and rotate vectors, and create, scale and add scalars.

The question of what these nodes should return deserves a brief consideration. It would be possible for these nodes to return *success* or *failure* based in some way on the result of the operation, for example, a zero result could return *failure*. Although superficially attractive, this is not necessarily useful, what does zero mean? We would be privileging a particular value and complicating the reasoning about node behaviour. Again, another point of reference is processor design, where the idea of flags being set by operations gave way to specific instructions to test state, in order to minimise side effects and regularise instruction sets[3]. For this reason, we make all nodes that may affect the blackboard **only** return *success* or *running*, and nodes that query the blackboard **only** return *success* or *failure*.

Thus, each of these nodes will return either *success* or *running*. If the destination register is a physically connected register, in our case this only applies to the $\boldsymbol{v_{goal}}$ register, and a previous action node within the current update cycle has written to that register, the node will return *running*, otherwise it will return *success*.

The second set of nodes, below the line in Table 6.5 are the *query* nodes. They have no side effects, and return only *success* or *failure*. The first two, `successl` and `failurel`, rather unsurprisingly always return *success* or *failure* respectively. We might ask why these are needed, and they are indeed syntactic sugar in that we can create equivalent effects with the use of the zero register and other condition nodes. It turns out that it is quite common to have constructs like that shown in Figure 6.9 where we want to try a number of options which may fail, but we don't what the entire subtree to fail if all the options do. By adding a final `successl` child we can ensure that `sel` always returns *success*.

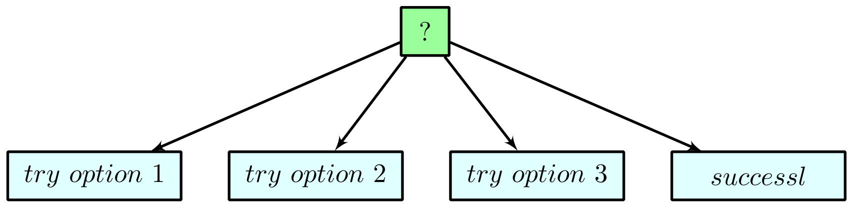


Figure 6.9: A selection node that always succeeds, even if all children fail.

The `ifprob` node provides a means of introducing probabilistic behaviour. This is necessary specifically in order that we can implement the *neighbour-count* constituent condition, see Section 6.2.2. More generally, we need a source of randomness to be able to implement probabilistic finite state machines in a BT. Depending on the value of a source register s_1, and two constants k and l, we return *success* with probability governed by the location of s_1 on the logistic curve defined by k (steepness) and l (location). The constants k, l are eight bit values interpreted as 5.3 signed fixed point numbers. They can represent numbers from -16.0 to 15.875 in steps of 0.125.

[3]An important part of superscalar processor design.

Figure 6.10 shows some curves with different k steepness values.

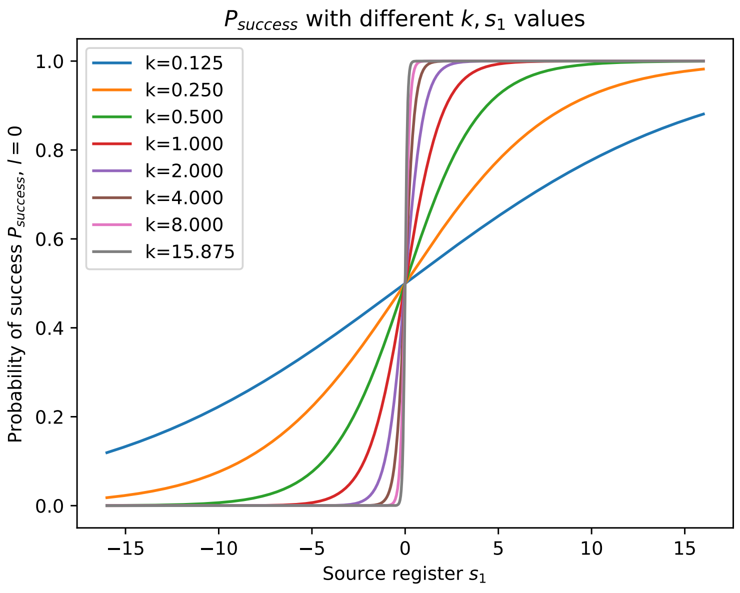
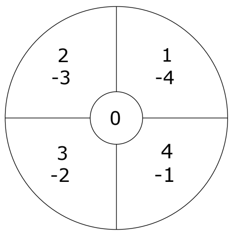
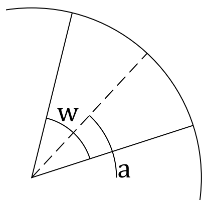


Figure 6.10: Different `ifprob` curves with different values of k and the source register s_1. The location parameter l is fixed at zero.

We can see that `ifprob` is quite a general purpose query node. With low values of k, or low steepnesses, we can have a smoothly varying range of probabilities depending on the source register value and the location l. See, for example, the $k = 0.25$ curve on Figure 6.10. Or, by specifying a high value of k, the curve is step-like, and can be used as an integer compare to x, with $l = x - 0.5$ specifying the value and the node returning *success* if $s1 >= x$ and *failure* if $s1 < x$. Another use is to specify fixed probabilities. By setting the source register to the zero register, the probability of success depends only on the values of k and l. By choosing the values appropriately, we can select a fixed probability between 0 and 1 with a maximum gap between available values of $\frac{1}{256}$.

Table 6.5: Action nodes. Scalar nodes have suffix s, vector nodes suffix v. $[n]$ is the nth blackboard entry. d and $s1, s2$ are 8 bit indices into the blackboard, with vector entries, the LSB is ignored and assumed zero. f is a 32 bit floating point. i, j are 8 bit signed integer. k, l are 5.3 fixed point signed, P is probability. $R(\theta)$ is a rotation matrix $\left[\begin{smallmatrix}\cos\theta & \sin\theta \\ -\sin\theta & \cos\theta\end{smallmatrix}\right]$

Name	Parameters	Behaviour	Description
`movcs`	d, i	$[d] \leftarrow i$	Set d scalar to constant
`movcv`	d, i	$[\boldsymbol{d}] \leftarrow (1, \angle\pi\frac{i}{128})$	Set d vector to unit vector constant
`mulas`	$d, s1, f, s2$	$[d] \leftarrow [s1] + f \times [s2]$	Scalar scale and add
`mulav`	$d, s1, f, s2$	$[\boldsymbol{d}] \leftarrow [\boldsymbol{s1}] + f \times [\boldsymbol{s2}]$	Vector scale and add
`rotav`	$d, s1, i, s2$	$[\boldsymbol{d}] \leftarrow [\boldsymbol{s1}] + R(\pi\frac{i}{128}) \times [\boldsymbol{s2}]$	Vector rotate and add
`successl`		*success*	Always return success
`failurel`		*failure*	Always return failure
`ifprob`	$s1, k, l$	$P_{success} = \dfrac{1}{1 + e^{k(l-[s1])}}$	Success with probability governed by location of $s1$ on logistic curve defined by k (steepness) and l (location). $P = 0.5$ when $l = [s1]$
`ifquad`	$s1, i$	$q = i \mod 5$ 	If $q \neq 0$ success if the vector $s1$ is in the quadrant q and $\lvert s1\rvert > 0.1$. If $q = 0$ success if $\lvert s1\rvert \leq 0.1$
`ifsect`	$s1, i, j$	$a = \pi\frac{i}{128}$ $w = \pi\frac{j}{256}$ *success* if $\lvert\angle[\boldsymbol{s1}] - a\rvert < w$ 	If $j \neq 0$ success if the vector $s1$ is in sector defined by angle of centre of sector i and sector width j and $\lvert s1\rvert > 0.1$. If $j = 0$ success if $\lvert s1\rvert < 0.1$

The final two nodes, `ifquad` and `ifsect` are designed to tell something about the direction of a vector, returning *success* if pointing in a particular direction and of sufficient magnitude. `ifquad` divides the 2D space into quadrants, numbered $1, 2, 3, 4$ in increasing positive angles, and $-1, -2, -3, -4$ in increasing negative angles. The node constant i specifies the quadrant and is interpreted *modulo* 5. Quadrant 0 is a special case. If the magnitude of the vector in blackboard register $s1$ is > 0.1 and the direction of the vector is in quadrant $q = i \mod 5$, the node returns *success*. If the quadrant is 0, the node returns success if the vector is of magnitude < 0.1, providing a way of identifying small near-to-zero vectors.

The node `ifsect` performs a similar function but in a more generic way. The node constants i and j define the angle and width respectively of a sector of the 2D space. If the vector in blackboard register $s1$ is of magnitude > 0.1 and is in the sector so defined, *success* is returned. If the width of the sector is specified as 0, *success* is returned if the magnitude < 0.1. The constant j is interpreted as an unsigned 8 bit number, so the sector width can be defined to subtend up to π radians.

The reason for these two similar nodes is historical; originally, we only implemented `ifquad` in order to write the constituent behaviours we defined above, but later work showed that the quadrant-based formulation is actually slightly awkward to use, consider for example the question of whether the robot is facing towards some coloured object. Using `ifquad` this involves checking if the vector, $\boldsymbol{v_{blue}}$, say, is quadrants 1 or 4. But maybe we want greater specificity; all non-zero vectors from the camera will fall within this complete semicircle. These problems arose when we started implementing behaviours beyond the original constituent behaviours, and motivated the design of `ifsect`. But by that point there was an already existing codebase making use of `ifquad`, so we decided to keep it.

6.5 Behaviours and conditions expressed as subtrees

Given the complete set of behaviour tree nodes and blackboard entries described in Tables 3.1, 6.3, and 6.5, we can now express each of the constituent behaviours and conditions described in Section 6.2 as subtrees. This section shows the trees as both listings and diagrams. The first line of a listing names the subtree and its parameters if it has them.

Behaviour `explore`

The robot explores the environment, moving forwards unless there is an obstacle detected in front of it, when it will turn away from the obstacle for a random amount of time up to a limit t. Figure 6.11 shows a behaviour tree that accomplishes this task.

All the composition nodes are memoried forms, meaning that, within the `explore` subtree, there is no preemption, actions once started are carried out to completion. At the root is a three-clause `selm` node, which tries each child subtree in turn until one succeeds. The first two subtrees consist of guarded sequences, first checking if the $\boldsymbol{v_{prox}}$ blackboard register is in the front left quadrant (quadrant 1) and if so,

```
explore(t):
selm3
  seqm2
    ifquad $\boldsymbol{v_{prox}}, 1$
    repeatr t
      movcv $\boldsymbol{v_{goal}}, -64$
  seqm2
    ifquad $\boldsymbol{v_{prox}}, -1$
    repeatr t
      movcv $\boldsymbol{v_{goal}}, 64$
  movcv $\boldsymbol{v_{goal}}, 0$
```

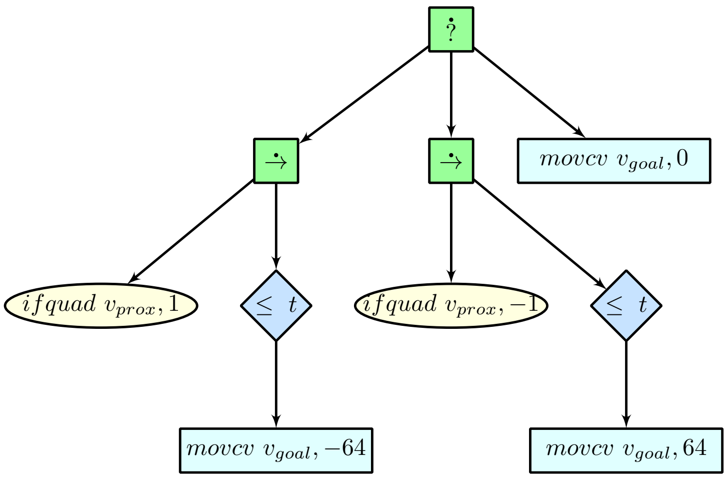


Figure 6.11: Subtree for behaviour `explore`

turning clockwise for up to t control steps, then checking if it is in the front right quadrant (quadrant -1) and if so, turning anticlockwise for up to t control steps, and if neither of these conditions are true, meaning there is no obstruction in front of the Xpuck, it moves straight forwards, remembering that `movcv` assigns a unit vector with direction given by the second parameter. This subtree will always return *running*, if it is performing a turn, or *success*.

It would be possible to encode forward movement with obstacle avoidance more simply by specifying the goal vector as:

$$\boldsymbol{v_{goal}} = \hat{\boldsymbol{x}} - k\boldsymbol{v_{prox}} \tag{6.22}$$

meaning: move forward (in unit x direction) while proximity vector is small, otherwise move in the opposite direction to the obstacle as indicated by the proximity vector. But this would only produce random movement necessary for exploration as a consequence of noise, compared to the given formulation above which deliberately introduces random amounts of turning at each obstacle.

6.5.1 Behaviour `stop`

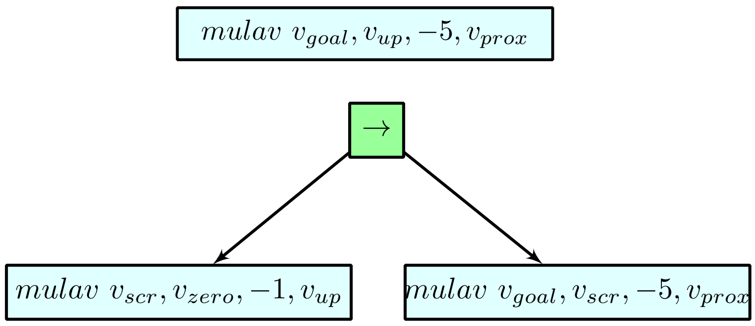
This behaviour is implicit when there are no actively ticked action nodes that write to the $\boldsymbol{v_{goal}}$ register.

6.5.2 Behaviours `upfield` and `downfield`

These behaviours are specified to move towards the $+x$ or $-x$ ends of the arena, with embedded obstacle avoidance. Here we are not concerned with explicitly embedding randomness, so the trees are very simple, shown in Figure 6.12, and we can use the

```
upfield:
mulav $\boldsymbol{v_{goal}}, \boldsymbol{v_{up}}, -5, \boldsymbol{v_{prox}}$
```

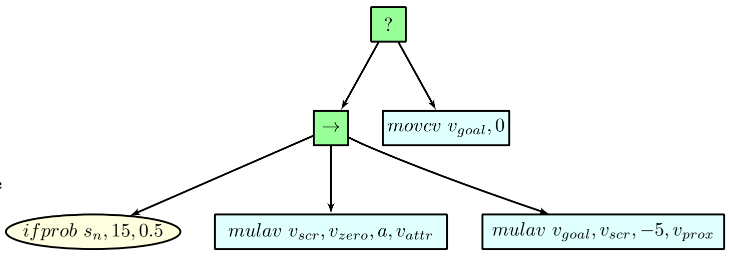
```
downfield:
seq2
  mulav $\boldsymbol{v_{scr}}, \boldsymbol{v_{zero}}, -1, \boldsymbol{v_{up}}$
  mulav $\boldsymbol{v_{goal}}, \boldsymbol{v_{up}}, -5, \boldsymbol{v_{prox}}$
```



mulav $v_{goal}, v_{up}, -5, v_{prox}$
→
mulav $v_{scr}, v_{zero}, -1, v_{up}$
mulav $v_{goal}, v_{scr}, -5, v_{prox}$

Figure 6.12: Subtrees for behaviours `upfield` and `downfield`

```
attraction(a):
sel2
  seq3
    ifprob $s_n, 15, 0.5$
    mulav $\boldsymbol{v_{scr}}, \boldsymbol{v_{zero}}, a, \boldsymbol{v_{attr}}$
    mulav $\boldsymbol{v_{goal}}, \boldsymbol{v_{scr}}, -5, \boldsymbol{v_{prox}}$
  movcv $\boldsymbol{v_{goal}}, 0$
```



?
→
movcv $v_{goal}, 0$
ifprob $s_n, 15, 0.5$
mulav $v_{scr}, v_{zero}, a, v_{attr}$
mulav $v_{goal}, v_{scr}, -5, v_{prox}$

Figure 6.13: Subtree for `attraction` and `repulsion`

form:

$$\boldsymbol{v_{goal}} = \boldsymbol{direction} - k\boldsymbol{v_{prox}} \tag{6.23}$$

where the goal vector is the sum of the desired direction and the some negative multiple of the proximity vector. The factor k describes how strong the 'aversion' to obstacles is, Francesca *et al.* [2014a] use a value of 5, which we follow. Since the upfield vector register $\boldsymbol{v_{up}}$ is a unit vector, when the robot gets close enough to an obstacle, the resultant will always be large enough to cause avoiding action.

`Upfield` can be specified with a single `mulav` node, but for `downfield`, we need to make use of the scratch register $\boldsymbol{v_{scr}}$ and the zero register $\boldsymbol{v_{zero}}$ to calculate the downfield direction by inverting $\boldsymbol{v_{up}}$ before subtracting the proximity component.

Behaviours `attraction` and `repulsion`

The robot moves towards or away from the nearest concentration of robots in the neighbourhood, with the strength of attraction or repulsion being governed by the parameter α, negative values producing repulsion. Obstacle avoidance is embedded, and when there are no neighbours, the robot moves forward. As noted in Section 6.3.3, the original intent of the $\boldsymbol{v_{attr}}$ blackboard register was that it could simply be multiplied by a positive or negative constant in order to get the attraction or repulsion, with forward movement in the absence of any neighbouring robots, but by including the unit x vector for forward movement when there are no neighbours this results in incorrect behaviour in the repulsion case. For this reason, we use a

```
red(b)
seq2
  sel2
    ifquad $v_{red}, 1$
    ifquad $v_{red}, -1$
  ifprob $s_{zero}, -0.25, b$
```

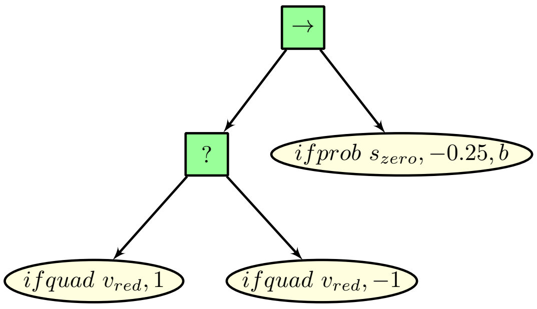


Figure 6.14: Subtree for `red` (similar for `green` and `blue`)

slightly more complicated tree, shown in Figure 6.13.

Here we have a guarded sequence that calculates the goal vector as:

$$\boldsymbol{v_{goal}} = \alpha \boldsymbol{v_{attr}} - 5\boldsymbol{v_{prox}} \tag{6.24}$$

providing the attraction and repulsion functions, together with obstacle avoidance. The guard is interesting though, using `ifprob` as a means of testing whether there are any robot neighbours. Recall that `ifprob s1,k,l` returns *success* with probability based on the position of the blackboard register *src* on the logistic curve defined by k (steepness) and l (position). Here, a value of $k = 15$ is almost a step function, and $l = 0.5$ means that if the blackboard register s_n, corresponding to the number of neighbours, is zero then *success* will be returned with $P = 0.00055$, and if the number of neighbours is one, *success* with $P = 0.99945$.

If there are neighbours, the guard will almost certainly pass, and the goal vector will be as above, but if there are no neighbours, the sequence fails and the second clause of the select is activated, assigning the unit x vector to the goal vector and giving forward movement.

6.5.3 Conditions `red`, `green`, and `blue`

These conditions as defined in Section 6.2.2 take one parameter, the probability of *success* β. If the camera detects the given colour within its field of view, the tree will return *success* with probability $P = \beta$. We implement this functionality as shown in Figure 6.14 for the colour red. Other colours make the obvious changes. If the camera resultant vector for the given colour is in either quadrant 1 or quadrant -1, which is true for any of the cases in Table 6.4 except for no colour detected, the selection is satisfied and so the `ifprob` node is triggered returning *success* with a probability based on b. Note that we cannot specify the probability directly, but by using a point on the logistic curve with steepness 0.25, see Figure 6.10. At this steepness, by varying b between the possible values of -16 to 15.875 we can choose one of 256 probability values from $P = 0.018$ to $P = 0.981$. Of course, we could use

`neighbour(k,l)`
`ifprob` s_n, k, l

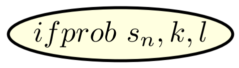


Figure 6.15: Subtree for `neighbour` and `invneighbour`

`fixedprob(b)`
`ifprob` $s_{zero}, -0.25, b$

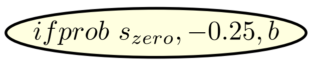


Figure 6.16: Subtree for `fixedprob`

other steepness settings to access a different range of probabilities.

6.5.4 Conditions `neighbour` and `invneighbour`

These conditions return *success* probabilistically, based on the number of neighbours. They take two parameters, k, the steepness of the logistic curve, and l, the number of neighbours to give $P = 0.5$. We can see that this condition can be written purely as a specific usage of `ifprob`, as shown in Figure 6.15. There is also no need for a separate `invneighbour` node, since this functionality can be achieved by negating the steepness parameter k which is equivalent to inverting the probability.

6.5.5 Condition `fixedprob`

Finally, the fixed probability node `fixedprob`. Again, this is simply a specific use of the `ifprob` node, as shown in Figure 6.16. As with the colour detection condition, we use a shallow sloped logistic curve to give smoothly varying probabilities for *success* based on the value of b varying over its range between -16 and 15.875.

6.6 Conclusion

In this chapter, we have shown how we have approached the problem of designing a behaviour tree architecture, starting with a reference model for the robot which we wish to control. This gives us a formal abstraction of the robot capabilities and constrains the behaviours that are possible. We then define several behaviours and conditions that we consider are useful for a robot to have, both from the swarm robotics perspective, but also from the perspective of serving as concrete endpoints for the design process. By keeping these design endpoints in mind, we iteratively seek to produce a set of action nodes and blackboard registers that are capable of elegantly expressing them. By demonstrating how these concrete endpoints can actually be written, we feel that this design process ends up with an architecture that is capable of expressivity and generalisation.

It is important to note that the methodology used here is by no means limited to a 2D robot, which we use because of our familiarity with both designing 2D simulators and with the availability of a large number of e-puck robots that could be enhanced into Xpucks for this work. To apply the methodology to flying drones, for example, the reference model would obviously be different but would still present a simplified abstraction of the real robot. The blackboard would still be used to present the abstracted robot senses and actuators to the BT leaf nodes, and the leaf nodes would be designed with the behavioural modalities of the drone in mind. Conceptually, the process would be very similar, and an interesting work to undertake in the future.

Chapter 7

Controller transferability

A central aim of this work is to move towards the idea of an autonomous swarm. By this, we mean a swarm of robots performing useful tasks in the real world that is not reliant on exterior computational resources, and can potentially adapt to a changing environment. Much conventional swarm robotics relies on the discovery of controllers, for example by evolution, in a computationally expensive offline reality simulation process.

In the previous two chapters, we have described the behaviour tree architecture we will use for our swarm robotics experiments, and the design of the Xpuck robot, which makes up the swarm of robots. We have talked about the design of the parallel robot simulator capable of running on the robots and detail some results demonstrating the processing power of the swarm, both for running a distributed evolutionary algorithm, and for running a tag-tracking image processing task. In this chapter we examine the problems of transferring automatically discovered controllers from the simulated environment to the real world and describe the steps we have taken to mitigate these problems.

A fundamental issue with the use of simulation in the automatic discovery of swarm robotics controllers is the *reality gap.* We noted in Chapter 3 the work of Francesca *et al.* [2014a] as an inspiration for the design of our behaviour tree architecture, enabling representation of behaviours at different granularities. The question of simulator fidelity is closely linked, the more minimal the simulator representation of reality, the more robust to reality gap effects it is necessary for our controller architecture to be. Finding a good balance between simulator fidelity, behaviour representation, and evolutionary algorithm that best mitigates the reality gap problem while still providing timely production of new controllers is the central theme of this chapter.

The structure of this chapter is as follows; firstly, we outline the collective task we aim to have the swarm perform. We calibrate and validate the simulator by making measurements of physical parameters of the robots and their behaviour, and incorporating these into the simulator, noting problematic areas and proposing mitigations. The calibrated simulator together with these mitigations is then used to evolve controllers for the task, which are transferred to the real robots. We measure the performance against the task of the controllers in simulation and in reality, showing no significant difference in performance, thus demonstrating successful mitigation of the reality gap.

7.1 Benchmark task

We need a benchmark task for the swarm of robots that is non-trivial and has relevance to possible real-world applications. In the field, *foraging* is regarded as canonical problem [Winfield, 2009a] in that it encapsulates the solution of many sub-problems, such as navigation, object recognition, and transport, and that it is a direct analogue to many real-world problems, such as harvesting, pollution control, search and rescue and many others.

There are many forms that foraging takes in swarm robotics experiments, often using virtual resources. Our version requires the swarm to continuously move a stream of objects in a particular direction, a collective transport task. We feel that the direct manipulation of objects is an important part of the problem, so we use a real object, in our case a round blue plastic frisbee. The Xpuck robots have no manipulators, so the frisbee can only be moved with pushing actions. We define the task thus:

The blue frisbee is placed approximately in the centre of the arena, shown in Figure 7.1. The size of the arena is 2 m by 1.5 m, with the origin in the centre. The swarm must move the frisbee in the $-x$ direction. If the frisbee contacts either the $+x$ or the $-x$ walls of the arena, at the far right and left sides respectively, the robots are stopped in place and the frisbee relocated back to the approximate centre before the robots are started again. The fitness of the swarm is the x component of the velocity of the frisbee in the $-x$ direction, normalised to the maximum Xpuck velocity and averaged over some time period.

This task is interesting because the robots of the swarm must locate the frisbee, move towards it, position themselves so that they are on the correct side, and then push it towards the $-x$ end of the arena, and continue to do this when the frisbee is relocated back to the centre. In order that the robots of the swarm actively pushing the frisbee are effective, there must also be little obstruction of the frisbee by other robots. This requirement for multiple different types of sub-behaviour and switching between them depending on the current state of the swarm and arena makes this task non-trivial and open to different possible solutions. Many swarm

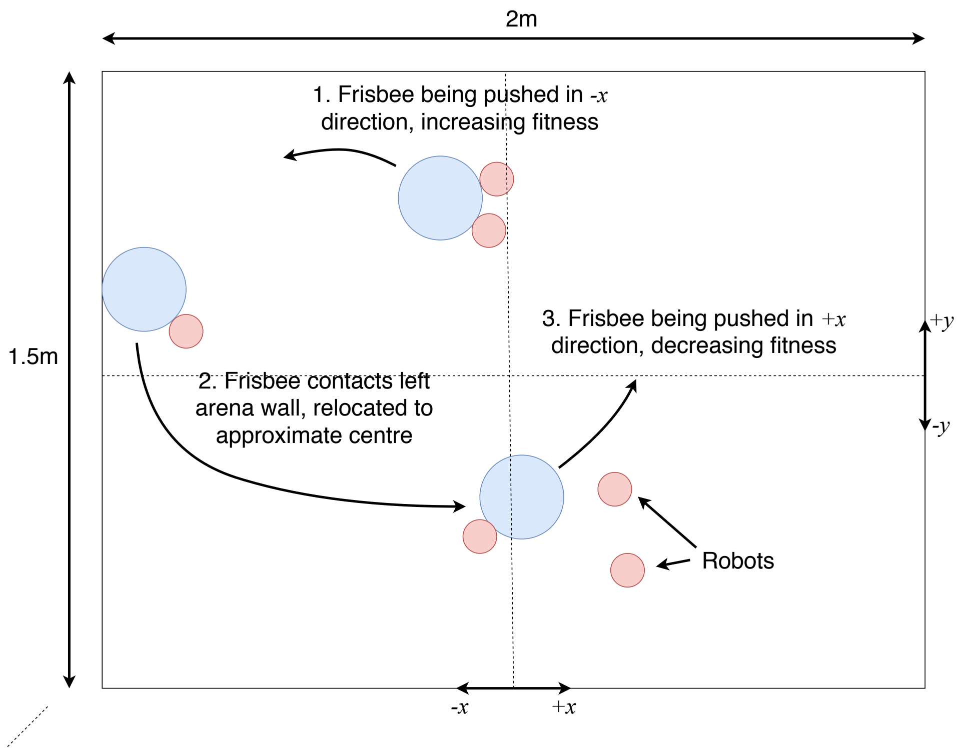

2m
1.5m
1. Frisbee being pushed in -x direction, increasing fitness
2. Frisbee contacts left arena wall, relocated to approximate centre
3. Frisbee being pushed in +x direction, decreasing fitness
Robots
+y
-y
-x
+x

Figure 7.1: Diagrammatic overview of the benchmark foraging task. Origin at the centre. Within the 2m x 3m arena, the robot swarm (shown as red discs) must try and move the frisbee (shown in blue) to the left of the arena. If the frisbee contacts the left or right walls of the arena, it is relocated back to the approximate centre.

robotics benchmarks are quite simple. For example, Francesca *et al.* [2014b] pit human designers and several automatic methods against each other with a number of different swarm tasks, consisting of forms of aggregation, dispersal and coverage. The most complex task is SCA - Shelter with Constrained Access, that must maximise the aggregation of robots on a particular area which is protected by walls on three sides, with environmental cues. Here the robots must seek the opening to the shelter, then aggregate there. Many foraging tasks in the literature are virtual, a robot is deemed to have picked up 'food' once it reaches an appropriate region and retrieved it once it returns to a 'nest' region, with no physical interaction with objects [Hoff *et al.* , 2013]. We argue that the requirement to find and move a real object increases the difficulty of the task and makes it more representative of real-world applications, such as warehousing and logistics.

7.2 Simulator physics calibration

The design of a robot simulator for use in evolutionary algorithms, or to ask *what-if* questions about possible scenarios, has to balance a fundamental trade-off between fidelity and performance - the more detail with which you simulate the real world, the longer it will take at a constant rate of calculation. But why does fidelity matter?

One of the fascinating things about evolutionary algorithms and other automatic discovery processes is the way in which solutions to the problem posed by the objective function are often unexpected and surprising. Lehman *et al.* [2018a,b] collate examples of this, with one class being the exploitation of simulator bugs and edge cases. These edge cases are the obvious corollary of a lack of simulator fidelity and potentially represent the ability to defy the laws of physics. An evolved controller solution that relies on this is not going to transfer well to reality.

A standard approach to lack of fidelity is to mask the deficiencies in noise [Jakobi *et al.* , 1995] but how much noise is sufficient? Too much noise makes it hard for evolution to find solutions. And the lack of fidelity is not a fixed thing - certain aspects of the simulation will be much truer to reality than others. This implies a possible strategy of avoiding or masking more the particularly problematic areas of simulation if possible, by design or by modifying the objective function [Koos *et al.* , 2013]. It also suggests that we can spend more of the calculation budget in certain important areas, if we know what they are.

We therefore face a multifaceted trade-off; what our simulation representation is, how we spend our simulation calculation budget within that representation, how we characterise the most problematic areas of simulation, how we avoid or forbid behaviours in those areas, how we use noise to mask infidelities. The approach we use is as follows. Firstly, in the previous chapter, we decided upon a 2D constrained geometry simulator as able to provide the desired raw performance, Eqn 5.3, and detail its implementation. In this section, we calibrate the various parameters of the simulation by measuring the real robots, and their senses, both real and virtual. We then run identical simple controllers on both simulated and real robots, gathering large amounts of sensor and actuator data from both in order to visualise and discover important points of difference. Differences are addressed by modifying simulator behaviour if that is possible without significant performance effects, or masking and avoiding the problematic areas otherwise.

7.2.1 Simulator physical parameters

There are a number of physical parameters used in the simulation, shown on Table 7.1. These should reflect reality as closely as possible. Some are straightforward to measure, such as mass. Others less so. We next describe each of these parameters, and the way they can be measured. Where direct measurement is not possible, we highlight this, and in the subsequent section describe the methodology by which we chose suitable values.

Object masses

These are the masses of the Xpuck robots, and the passive object which can be pushed by the Xpuck, in reality a blue frisbee of about 210 mm diameter. It is obviously straightforward to measure their masses.

Table 7.1: Simulator parameters

Parameter	Value	Description
m_{xpuck}	0.3 kg	Mass of Xpuck
m_{object}	0.07 kg	Mass of object
μ_{xpuck}	0.65	Coefficient of friction between Xpuck and floor
$\mu_{frisbee}$	0.5	Coefficient of friction between the frisbee and floor
α_1	0.1	Velocity dependent position noise
α_2	0.1	Angular velocity dependent angle noise
α_3	0.1	Velocity dependent angle noise
μ_{bodies}	0.15	Coefficient of friction between bodies
e	0.1	Coefficient of restitution between bodies

Coefficients of friction with floor

The simulator calculates friction forces in two domains. Firstly, as a 2D physics simulation, there are the friction forces that apply within the plane of simulation, that is, disregarding the notion of a ‘floor’ and gravity. Objects collide, and in the process of collision transfer momentum which may be linear or angular. The coefficients of restitution e and friction between objects μ_{bodies} governs the process of momentum transfer. Secondly, there is the interaction of the objects with the arena floor. There are no collisions to resolve in this case, purely forces and torques as a result of object velocities, wheel velocities, and normal forces due to mass and gravity. These are governed by two coefficients μ_{xpuck} and $\mu_{frisbee}$.

Coefficients of friction between Xpuck and floor, and frisbee and floor were measured with a Sauter FK50 force gauge, with a resolution of 0.02 N. A thread was attached to the Xpuck or frisbee at the lowest possible position and the force gauge pulled across the arena surface, pulling the Xpuck or frisbee with it. While in motion, the reading was taken. Ten readings were taken, the top and bottom discarded, and the remaining eight used to calculate a mean value of μ. We also measured static friction with the Xpuck by noting the highest reading just before movement started. With the frisbee, the mass was low enough that the force was only about ten times the minimum resolution of the gauge, making it very difficult to get meaningful static readings. The simulator only models a single component of friction for movement of bodies over the arena floor, we averaged the static and dynamic measured values for the Xpuck, giving values of $\mu_{xpuck} = 0.65$ and $\mu_{frisbee} = 0.5$.

Coefficients of friction and restitution between objects

Measuring e, μ_{bodies} is much more complicated, and highlights an important area of difference between the simulator and reality. The simulator is a 2D physics simulation. Interactions between bodies within the plane of the simulation, that is, collisions between bodies, use velocity and mass, together with the coefficients of restitution e, and friction μ_{bodies} to calculate the impulse vectors, which is then resolved to force and torque components. But these objects in simulation are ideal 2D circles contacting at a single point. The real robots are three dimensional, have protrusions, rough edges, 3D printed plastic cases, and are decidedly non-ideal in characteristics. Looking up standard values for μ for plastics suggests figures of $0.3 - 0.5$, but using numbers like this within the simulator results in very unrealistic looking behaviour.

Likewise, typical values for the coefficient of restitution for plastics are quite high, in the region of 0.8, but this is for a perfect sphere of material dropped onto a perfectly hard surface. Using 0.8 as a value in simulation resulted in absurd collisions, which bore no resemblance to actually observed collisions in the real world. Obviously, a robot is not a perfect sphere of plastic, and what is actually contacting is flexible layers of the 3D printed skirts and surrounds, which act more like compliant dampers than pure solids.

Noise model

The simulator noise model is a simplified form of that presented in Thrun *et al.* [2005]. Three coefficients, $\alpha_1, \alpha_2, \alpha_3$, control respectively velocity dependent position noise, angular velocity dependent angle noise, and velocity dependent angle noise. So position and angle are modified at each timestep:

$$\boldsymbol{x'} = \boldsymbol{x} + \boldsymbol{v} \cdot s(\alpha_1) \tag{7.1}$$

$$\theta' = \theta + \omega \cdot s(\alpha_2) + |\boldsymbol{v}| \cdot s(\alpha_3) \tag{7.2}$$

where $s(\sigma)$ is a sample from a Gaussian distribution with standard deviation σ and mean of zero.

7.2.2 Choosing appropriate parameter values

As we have seen, there are a number of parameters for which there is no direct measurements possible. These are the coefficients of restitution e and friction μ_{bodies}, and the noise model parameters $\sigma_1, \sigma_2, \sigma_3$. The approach we follow is to run the real robots over some defined trajectory, which may include collisions, multiple times. We use the Vicon system to accurately measure their true poses. We then construct the same scenario in simulation and run multiple trials, and adjust the parameters until

Table 7.2: Measurement of coefficient of friction of Xpuck and frisbee on perspex arena surface. High and low bracketed values discarded.

Reading	Xpuck static	Xpuck dynamic	Frisbee
1	(2.06 N)	(1.40 N)	0.30 N
2	2.06 N	1.64 N	0.34 N
3	1.78 N	1.58 N	0.38 N
4	2.06 N	1.70 N	0.32 N
5	(1.74 N)	1.72 N	0.36 N
6	1.92 N	(1.80 N)	(0.40 N)
7	2.00 N	1.66 N	(0.28 N)
8	1.88 N	1.80 N	0.34 N
9	1.90 N	1.76 N	0.28 N
10	2.06 N	1.80 N	0.32 N
$\bar{F}$	1.96	1.71	0.33
$\mu = \frac{F}{gm}$	0.67	0.58	0.5

we have achieved a good correspondence between the simulator object trajectories and the real trajectories.

The noise of the real robots for simple movements is actually quite low, commanded to move a fixed distance of 1 m, the standard deviation of the positional error was 8 mm in both x and y. This corresponds to $\alpha_1 \approx 0.01$ and $\alpha_3 \approx 0.015$. Values for α_2 were harder to measure, we make the assumption it is similar to $alpha_3$ and thus $alpha_2 \approx 0.015$. These figures represent the real noise of a single robot. In fact, each Xpuck has a differing bias, caused by wheel diameter variations, which caused position differences of a similar order, around 10 mm. Imperfections and dirt in the arena contribute further to the noise over all robots. These real factors probably contribute to at least doubling the numbers above.

We used 0.1 for all three α, representing a degree of noise masking, equivalent to between three and five times the actual noise including inter-robot variation. Jakobi *et al.* [1995] use double the measured noise but used a simulator with a higher degree of fidelity than ours, with quite carefully modelled light sensors, for example. The effect of noise masking is to deny the evolutionary algorithm access to areas of difference between simulator and reality, it cannot 'see' them and exploit them to produce controllers relying on unphysical behaviours, which will obviously not transfer well. But too much noise will mean the evolutionary algorithm will not be able to advance. The shape of the problem must still be apparent. There is obvious scope for further work to examine the best level of masking.

The coefficient of restitution seemed to make little difference to Xpuck-frisbee collisions below a value of about 0.3, but Xpuck-Xpuck collisions looked most realistic at a lower value, so this was fixed at 0.1.

The coefficient of friction μ_{bodies} was most significant in effect. Figure 7.2 shows an example collision with parameter sweep. The coloured lines show the trajectories of the Xpuck and frisbee in real life before and during a collision, and the grey lines show simulated trajectories with over a sweep of parameter values. In this case there is good agreement between simulation and reality at $\mu_{bodies} \approx 0.15$. Other examples are not quite so clear cut but this value seemed a reasonable compromise.

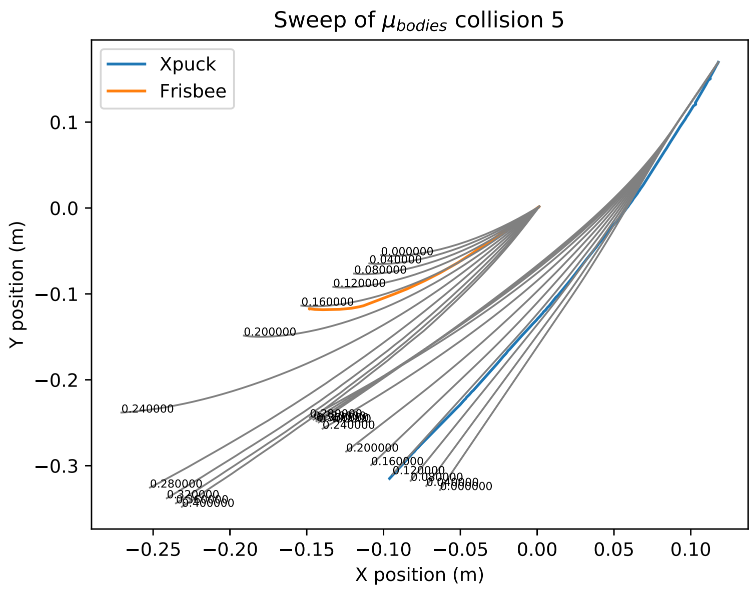


Figure 7.2: Example parameter sweep of a collision between Xpuck and frisbee. Coloured lines show trajectories in reality, grey lines are simulated trajectories with different values of the parameter μ_{bodies} the coefficient of friction between bodies.

7.2.3 Observations and mitigation

Throughout this process of hand tuning, guided by parameter sweeps, recorded real collisions, and observation, it became obvious that collisions between objects were the most problematic area of the simulation to model with good fidelity. This is not a surprise, the physics is more complicated. The dynamics are more 'three dimensional'; consider a collision between two Xpucks, they contact at a point above the arena floor which tends to result in each Xpuck slightly rocking back and applying less normal force to their wheels, meaning less friction between the Xpucks and floor, and a lower apparent friction between the Xpucks. This cannot be modelled easily in a 2D simulator. Conversely, the collisions between an Xpuck and the frisbee are at a physical location very close to the arena floor and are more faithfully modelled.

Given that the cost of more accurately simulated collisions is a simulator performance impact we cannot afford, we choose instead to minimise the most problematic collisions. An unconstrained evolutionary algorithm working within the simulator will find novel solutions that exploit these infidelities. We mitigate this with a subsumption controller architecture, making collision avoidance using the IR proximity sense the highest priority behaviour, before the evolved behaviour. Figure 7.3 shows this. We define a behaviour subtree called `avoiding` which returns *success* if an obstacle in front of the Xpuck has been detected and collision avoidance is taking place. The top of the tree only uses the evolved behaviour if `avoiding` returns *failure*.

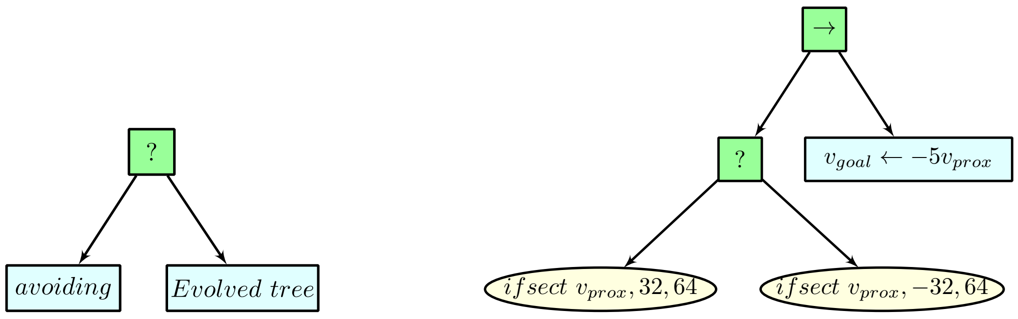

?
avoiding
Evolved tree
→
?
$v_{goal} \leftarrow -5v_{prox}$
$ifsect\ v_{prox}, 32, 64$
$ifsect\ v_{prox}, -32, 64$

Figure 7.3: Implementing collision avoidance subsumption architecture. The top level tree is shown on the left, if the 'avoiding' behaviour is not happening, them the evolved tree will be executed. The 'avoiding' subtree is shown on the right. If there is an obstacle detected to the front left or front right of the robot, it will move in the opposite direction to the proximity vector.

By evaluating evolved trees within this top level tree we ensure that the ability of the evolutionary algorithm to exploit the poorly modelled collision dynamics is minimised and thus we reduce one of the causes of reality gap effects.

7.3 Sensor calibration

So far, we have only considered the physics aspects of the simulation, but the simulator also has to run the robot controller. In this section, we look at the modelling and conditioning of the robot sensors. Recall that we define the abstract reference model of the robot senses, in Table 6.1. This is the basis from which we construct the behaviour tree blackboard, Table 6.3, and action nodes, Table 6.5. It defines how we represent the real sensor data to the controller, and also what a simulator model of the senses has to provide. From the perspective of the robot controller, it sees no difference between the simulator and reality, with regards to the format of the presented sensory data.

We can see that in both the real robot and in simulation, for a given physical arrangement of robots and objects within the arena, there should be an identical representation of the senses that is presented to the controller. The methodology we use in this section is as follows. We run real robots in various scenarios and log all blackboard data, together with positional information from the Vicon system. We then take this data and use it to construct many frames, consisting of the physical locations of all objects within the arena over time, which are presented to the simulator. For each frame, we run the simulator for one timestep in order to get the simulated senses for that frame. We then have a time series of real senses, and a corresponding time series of simulated senses for identical positions. We can then compare these two sets of data to find problematic areas.

7.3.1 IR proximity sensors

The reference model states that the eight IR proximity sensors, spaced around the robot at a height of about 35mm above ground level, respond with 0 when there is no object within range, and 1 when there is an object immediately adjacent to the sensor. As described in Section 5.3.1 within the simulator these are modelled by casting a ray from the sensor position out to the maximum sensor range and using a fast approximation to get any intersection distance with other objects.

The real sensors have an IR emitter and an IR detector. By measuring the detected level of IR with the emitter both turned off and turned on, the contribution due to IR reflection can be estimated. The level of detected IR reflection is dependent on the distance to the object, but also on the angle, and the materials of the object. The behaviour with distance is highly non-linear and approximates inverse fourth power, as with radar. In order to derive a reasonable transformation from this raw data to the desired real-valued distance range of $[0, 1]$ we measured the response r_{sensor} (ADC counts) of eight different sensors to plain white card at multiple distances $r_{distance}$ in metres.

The raw data all had similar characteristics as distance increased, with an initial slow drop until a threshold distance, beyond which the signal fell sharply, flattening out towards zero. We aligned all the curves with distance and value offsets, and a scaling factor, then found a best fit curve with two regions. Figure 7.4 shows the aligned curves, with a fitted idealisation.

The fitted curve takes the form:

$$r_{sensor} = 3500 \cdot \begin{cases} (40r_{distance} + 0.7)^{-4} - 0.06 & \text{if } r_{distance} > 0.007\,\text{m} \\ -5r_{distance} + 1.04 & \text{otherwise} \end{cases} \tag{7.3}$$

Real sensor data was conditioned by creating an eight entry table populated from this fitted curve and using this to perform piecewise-linear interpolation of the raw sensor data to give a linear distance measurement. This was converted to the required $[0, 1]$ range.

7.3.2 Camera

The camera sense divides the field of view into three segments, each one third of the width of the frame. Within each segment, we may detect red, green, or blue. The arena floor and walls are white, the only coloured items are the red Xpuck skirts, the blue frisbee, and potentially green coloured items affixed to the walls. The real

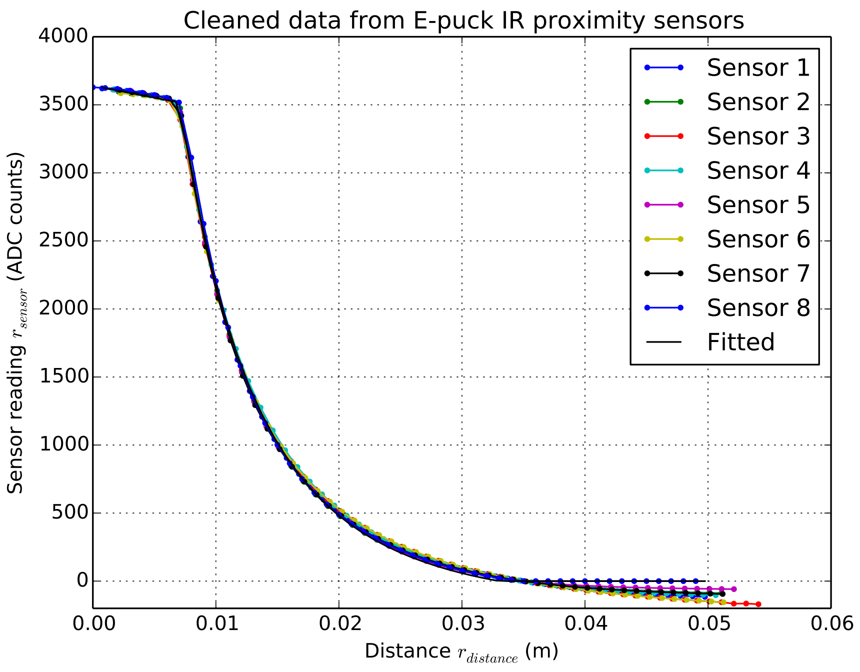


Figure 7.4: IR proximity sensor data, scaled and shifted to align, with the fitted curve.

camera feed is processed by classifying pixels as red, green, or blue if they fit within defined regions of the HSV colourspace, shown in Table 7.3, then checking if the proportion of pixels within a segment exceeds a threshold.

Table 7.3: Colour regions within HSV colourspace used to classify pixels

Colour	Hue	Saturation	Value
Red	$[320°, 20°]$	$[0.4, 1.0]$	$[0.4, 1.0]$
Green	$[80°, 160°]$	$[0.4, 1.0]$	$[0.4, 1.0]$
Blue	$[180°, 260°]$	$[0.2, 1.0]$	$[0.4, 1.0]$

Initial experiments suggested that the range of the camera for detecting coloured blobs was limited. This was due to setting a detection threshold of 5%, which was necessary to eliminate false positives when the visual background was cluttered. Within the controlled environment of the arena, we were able to reduce the detection threshold to 0.2%, equivalent to 50 pixels at a resolution of 320×240. This made it possible for an Xpuck to reliably detect the frisbee from the opposite end of the arena.

We were also concerned that the detection ability would be adversely affected by the angular velocity of the Xpucks. Four Xpucks were placed in the arena, together with the blue frisbee. The Xpucks were at distances varying from 0.5 m to 1.8 m from the frisbee. Each Xpuck was commanded to rotate at ten different angular velocities of 10%, 20% etc up to 100% of the maximum angular velocity of about 3.5 radians/s.

Camera blob detection data was captured, along with Vicon ground truth data, and simulated senses calculated as described above. The length of the vector difference between real blackboard vector v_{blue} and the simulated version was measured for each controller update cycle. Only measurements taken when the angular velocity was stable were used. There were a total of 1357 measurements. Table 6.4 shows the possible values of a camera vector. The maximum length of a vector difference between any of the possible values is 2.95. The length of difference between the simulated v_{blue} and the real one can be regarded as a measure of the camera sense reality gap:

$$r_{gap} = \frac{|\boldsymbol{v_{blue\_sim}} - \boldsymbol{v_{blue\_real}}|}{2.95} \tag{7.4}$$

We plot the normalised vector difference length r_{gap} against the angular velocity for each of the measurement points.

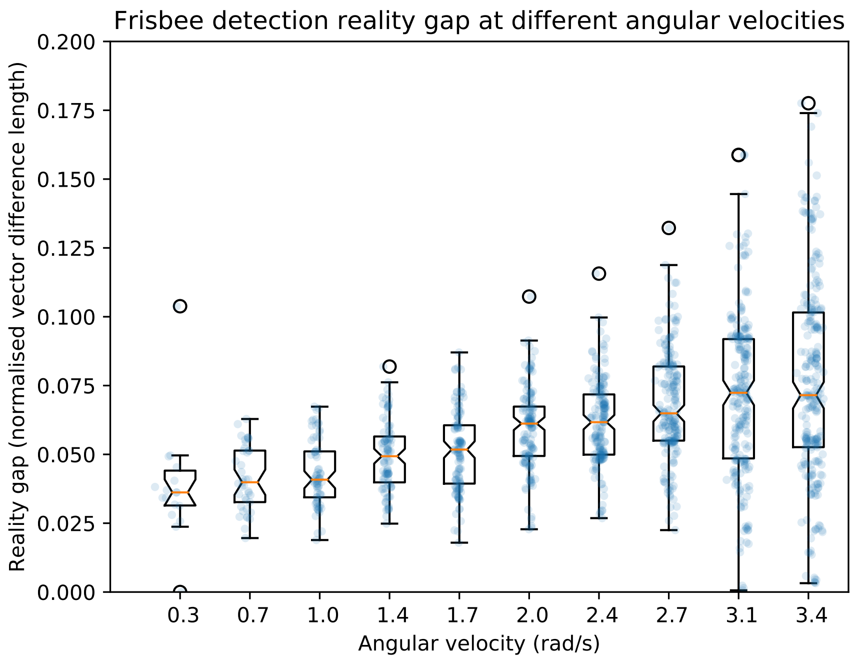


Figure 7.5: Camera frisbee detection reality gap at different angular velocities. Each point is a single measurement of the difference between the simulated and real values of v_{blue}, normalised to the maximum possible difference, at a particular angular velocity. Measurements are binned to the nominal angular velocity to give the boxplots. In general, real detection accuracy falls as angular velocity increases.

Figure 7.5 shows the results. Generally, as the angular velocity increases, the reality gap increases too, but remains below 10% on average. It is worth considering what the reality gap consists of in this case. The simulator camera sense is perfect, it does not model pixels and thresholds but directly uses the geometry to calculate if a coloured object is visible within a segment of the camera. The real camera has processing latency, shearing effects from lateral movement[1], and blurring. All of these factors increase the error with increasing robot angular velocity, when compared to a perfect camera.

7.3.3 Virtual Senses

The virtual senses of range-and-bearing, giving the blackboard registers of $\boldsymbol{v_{attr}}$ and s_n, and the compass, giving $\boldsymbol{v_{up}}$, are both derived from the Vicon motion tracking system. This has a latency of a certain number of frames, and in addition there is other overhead in the packet exchange with the robots. In order to quantify this, we set up an experiment where a robot would execute a behaviour tree that would move forward and pause every few seconds. All sensor and actuator data is timestamped and logged.

Figure 7.6 shows once of the pauses. We can clearly see the latency effect, about five or six telemetry ticks between a change in commanded velocity and the Vicon system registering the change. This amounts to about 200 ms latency for these senses.

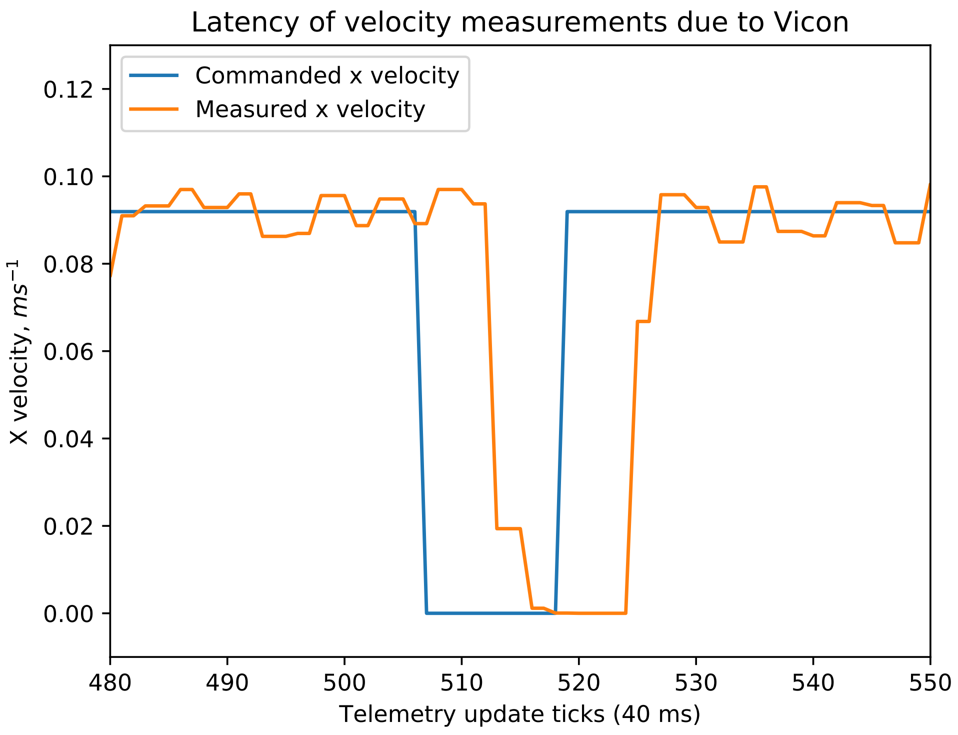


Figure 7.6: Time delay between commanded velocity change and the Vicon tracking reported velocity.

As noted earlier, in the design of the simulator we perform range-and-bearing calculations only every other controller update of 100 ms for reasons of computational

[1]The camera uses a rolling shutter, so the time of exposure of a pixel depends on the vertical position of the pixel within the frame.

complexity. This is not exactly the same as the Vicon latency since the delay will vary from control step to step but it is similar.

7.4 Testing controller transferability

A major step towards full on-board evolutionary experiments is the evolution of controllers in simulation that transfer well to a swarm of real robots, demonstrating similar performance. In other words, with a low *reality gap.* In this section, we take the benchmark task outlined above and evolve a controller to perform this task using the simulator with calibrations and mitigations as described above. We then transfer the controller to the swarm of Xpucks and measure their fitness in reality.

7.4.1 Detailed task

The task is broadly as described in Section 7.1. Nine robots are used. The arena is 2 m x 1.5 m surrounded with 180 mm high walls, with the origin at the centre of the arena. The robots are placed with random pose at least 100 mm from any others in the region bounded by $x \in [-0.5\,\mathrm{m}, -0.9\,\mathrm{m}]$ and $y \in [-0.5\,\mathrm{m}, 0.5\,\mathrm{m}]$. A blue frisbee of 210 mm diameter is placed randomly in the region bounded $x \in [0.8\,\mathrm{m}, 0\,\mathrm{m}]$ and $y \in [-0.2\,\mathrm{m}, 0.2\,\mathrm{m}]$. The task for the swarm to move the frisbee as far in the $-x$ direction of the arena as possible from its starting point within 1 minute. The fitness is expressed as a normalised velocity:

$$f_{raw} = -\frac{x_{finish} - x_{start}}{t_{sim} v_{max}} \tag{7.5}$$

and can range from $[-1, 1]$.

For the purpose of the evolutionary algorithm, we make two modifications to the raw fitness. Firstly, we penalise solutions that fail to move the frisbee at all. If the start and finish positions are identical, we subtract 1 from the raw fitness. The value was chosen arbitrarily to be greater than any possible negative fitness value from movement of the frisbee. This was motivated by the observation that any movement of the frisbee at all, even in the wrong direction is a prerequisite for a proto-solution; if the robots move randomly we have a better controller than if the robots are static.

Secondly, the fitness value is derated by factor r_{derate} based on the resource usage, or *parsimony* of the controller. Parsimony $r_{parsimony}$ is the proportion of the total available memory space that is not occupied by the behaviour tree:

$$\begin{aligned}
k_{penalty} &= \begin{cases} 1 & \text{if } x_{finish} = x_{start} \\ 0 & \text{otherwise} \end{cases} \\
r_{derate} &= \begin{cases} 2r_{parsimony} & \text{if } r_{parsimony} < 0.5 \\ 1 & \text{otherwise} \end{cases} \\
f_{evo} &= r_{derate} \cdot (f_{raw} - k_{penalty})
\end{aligned} \tag{7.6}$$

Once the free space falls below 50%, the fitness is reduced, reaching zero when all memory is occupied. This has the effect of controlling bloat within the evolution of the behaviour tree, an important issue within Genetic Programming [Koza, 1994].

7.4.2 Behaviour tree nodes and allowed parameters

It is important for the purpose of thinking about applying evolutionary methods to behaviour trees to separate behaviour tree nodes and subtrees into different sets. Genetic Programming, when creating new trees or mutating existing trees must have a set of inner nodes and a set of leaf nodes from which it can choose. The inner nodes are some subset of the composition nodes $C' \subset C$ shown in Table 3.1. The leaf nodes are some subset of the base action nodes A shown in Table 6.5, but we may also define a set of subtrees T that can be used by the evolutionary algorithm *as if* they were leaf nodes. This is possible because of the modularity of behaviour trees. So the leaf nodes are given by $L \subset A \cup T$.

As well as the sets of allowable inner and leaf nodes, the evolutionary algorithm requires a set of allowed ranges for all possible parameters within these nodes. These can be divided into blackboard registers that are sources, registers that are destinations, and various types of integer, fixed point, and floating point constants.

The complete set of standard composition nodes described in Table 3.1 are used, with fixed arities of 2, 3, and 4 for the `seq*` and `sel*` types. Table 7.4 shows the action nodes and subtrees we make available to the evolutionary algorithm. The action nodes are as outlined in Table 6.5 except for the nodes `rotav`, which was not implemented at the time of this experiment, and `ifquad`, since `ifsect` provides the same functionality in a more flexible way.

We use all the subtrees defined in Section 6.5 (`explore`, `upfield`, `attract`, `neighbour`, `fixedprob`) with the addition of `avoiding`, described above, and `bleft`, `bright`, and `bfront`. These are very simple subtrees that are special cases of `ifsect`, using the blackboard register v_{blue}. They return *success* if there is a blue object within the left, right, or central field of view of the camera, corresponding to the possible $\boldsymbol{v_{blue}}$ vector angles of $18.7, 9.35°$ for left, $0°$ for centre, and $-9.35, -18.7°$ for right. The sector locations and widths shown in Table 7.4 classify these vectors. The motivation for this is that since the task involves moving a blue frisbee, by providing these specific cases of `ifsect` we can narrow the search space of the evolutionary algorithm and speed evolution.

Note: While documenting `bleft` and `bright` we discovered that we had in fact specified the sector width for the special case `ifsect` nodes incorrectly as $\angle 81°$ rather than $\angle 162°$. This means that both `bleft` and `bright` will only ever return *failure*.

Table 7.4: Action nodes and subtrees used for transferability experiment. Notation $S \equiv$ *success*, $F \equiv failure$, r maximum number of timesteps turning, g strength of movement towards or away from $+x$ or neighbours, k and l steepness and location of logistic curve.

Node	Parameters	Description
`movcs`	d, i	Set scalar to constant
`movcv`	d, i	Set vector to constant
`mulas`	$d, s1, f, s2$	Multiply and add scalar
`mulav`	$d, s1, f, s2$	Multiply and add vector
`ifprob`	$s1, k, l$	S wth probability based on operands
`ifsect`	$s1, i, j$	S if vector in sector
`successl`		S always
`failurel`		F always
Subtree		
`explore`	r	Move forward until obstacle, then turn randomly $< r$
`upfield`	g	Move in $+x$ or $-x$ direction
`attract`	g	Move towards or away from other Xpucks
`neighbour`	k, l	S with probability based on number of neighbours
`fixedprob`	k	S with fixed probability
`avoiding`		S if proximity collision avoiding
`bleft`		S if blue in left of camera FOV
		`ifsect` $\boldsymbol{v_{blue}}, \angle 90°, \angle 162°$
`bright`		S if blue in right of camera FOV
		`ifsect` $\boldsymbol{v_{blue}}, \angle -90°, \angle 162°$
`bfront`		S if blue in centre of camera FOV
		`ifsect` $\boldsymbol{v_{blue}}, \angle 0°, \angle 8°$

The allowed parameter ranges are shown in Table 7.5. We don't allow the evolutionary algorithm to generate nodes that write to read-only blackboard registers, although this would have no effect except to increase the search space. The restriction of the floating point range deserves a little further consideration. The complete range of a 32-bit floating point number is $(-3.4 \times 10^{38}, 3.4 \times 10^{38})$. This is huge, and much of this range will produce useless outputs, in the sense that the velocity goal vector saturates at length 1. In addition, by allowing unrestricted ranges, the multiplies in `mulas` and `mulav` readily result in infinities and NaNs, which tend to propagate through further calculations[2]. If these reach the physics computation part of the simulation, it not longer functions correctly. We adopted a two-fold approach to this problem, firstly restricting the range of floating point values allowed as parameters. This in itself does not stop infinities or NaNs within the controller, so secondly, we check and enforce that the controller outputs of motor velocity are sensible, replacing infinities and NaNs with zeros so that the physics simulation does not explode.

Table 7.5: The allowed ranges for parameters within the evolutionary algorithm. B is the set of all blackboard registers

Parameter	Range	Description
d	$\{zero, \boldsymbol{v_{goal}}, s_{scr}, \boldsymbol{v_{scr}}\}$	Destination registers. Only zero, the goal vector and the scratchpads
$s1, s2$	B	Source registers. All registers
i, j	$[-128..127]$	8-bit signed integer constant, all values valid
k, l	$\frac{[-128..127]}{8}$	5.3 signed fixed point, all values valid, equivalent to $[-16, 15.875]$
r	$[1..100]$	8-bit integer, restricted range, for `repeat*`
f	$[-32.0, 32.0]$	32 bit floating point, restricted range
g	$[-5.0, 5.0]$	32 bit floating point, restricted range, for attraction/repulsion strength

7.4.3 Evolutionary algorithm

The evolutionary algorithm proceeds in the following way using the parameters in Table 7.6. A random population of size n_{pop} is created using the Koza's *Ramped half-and-half* procedure [Poli *et al.* , 2008], shown in Algorithm 11, with a maximum tree depth of n_{depth}. The fitness of the population is measured in simulation with a simulated time of t_{sim}. Multiple evaluations n_{eval} carried out on each individual, with different starting conditions and the average of the modified fitness function f_{evo} given in Equation 7.4.1 is used.

A new population of individuals is then formed from this population; the fittest n_{elite} individuals are transferred across unchanged. The remaining individuals in the new population are created by using standard tournament selection of size n_{tsize} to select two parents which are combined with a tree crossover operation biassed to choose inner node crossover points at probability $p = 0.9$ [Koza, 1992]. These individuals then undergo parameter mutation with $p = p_{mutparam}$ for each parameter in the tree. This is followed by point mutation over all nodes, which with $p = p_{mutpoint}$ may replace a node with another of the same arity. Then follows subtree mutation, which with $p = p_{mutsubtree}$ per individual may replace a randomly chosen node biassed to

[2]IEEE754 standard for floating point has rules for how Not a Number and infinity propagate; any operation with a NaN yields a NaN, and various operations involving infinities produce a NaN.

inner nodes with $p = 0.9$, with a new random subtree generated using `full` with a depth randomly chosen up to a maximum of n_{depth}.

Table 7.6: Evolutionary algorithm parameters

Parameter	Value	Description
n_{pop}	64	Population
n_{gen}	1000	Generations
n_{eval}	8	Evaluations
n_{elite}	3	Number of elite
n_{tsize}	3	Tournament size
n_{depth}	4	Tree depth
t_{sim}	60 s	Simulated time
$p_{mutparam}$	0.05	Probability of parameter mutation
$p_{mutpoint}$	0.05	Probability a node may be replaced by a node of the same arity
$p_{mutsubtree}$	0.1	Probability an individual may have a node replaced by a new subtree

This new population replaces the original population and the process repeats for n_{gen} generations. At each generation only the non-elite individuals are re-evaluated, the elites keep their previously measured fitness.

The choice of parameter values was initially guided by Poli *et al.* [2008] and the parameters that worked in Chapter4, but several changes were made based on experience. Mutation was simplified to just three operators, with parameter and node mutation changed to a fixed rate per node, rather than probability per tree, but with values roughly equivalent to those in Table 4.3 for a ten node tree. Population was increased to 64, more typical in the literature, and number of evaluations reduced to 8, these numbers based on the parallel simulator having highest performance with power-of-two numbers of evaluations of at least 256. We ran some trials varying the mutation rates by factors of two in each direction with little observed change in evolutionary trajectory so left them at the values shown. Varying initial tree depth below four resulted in slower progress and lower fitness, above seemed to make little difference.

This evolutionary algorithm was run ten times with different random seeds. Table 7.7 shows the fitness and the parsimony of the fittest individual of each run, together with the average fitness of the entire population.

7.4.4 Transfer to reality

The fittest controller from Run 7, with a fitness of 0.10, was measured using 500 simulations with different starting conditions. It was then transferred to the swarm of real robots and run 20 times.

For each real run, the robots were positioned to the left of the $x = -0.7$ line and the

Algorithm 11 Ramped half-and-half. Use both `full` and `grow` methods to generate a population of p of individuals with a maximum depth varying from 0 to $d-1$. The depth of a node is the number of edges traversed to reach it. C is the set of composition nodes, A is the set of action nodes.

```
1: function RampedHalfAndHalf($p$,$d$)
2:     for $i \leftarrow 0$ to $int(p/2)-1$ do
3:         $depth \leftarrow int(2*i*d/p)$
4:         $pop[2*i] \leftarrow$ Full($depth$)
5:         $pop[2*i+1] \leftarrow$ Grow($depth$)
6:     return $pop$
7: function Full($d$)
8:     if $d = 0$ then
9:         $node \leftarrow$ choose_random_element($A$)
10:    else
11:        $inner \leftarrow$ choose_random_element($C$)
12:        for $i \leftarrow 1$ to $arity(inner)$ do
13:            $arg.i \leftarrow$ Full($d-1$)
14:        $node \leftarrow (inner, arg.1, arg.2, ..)$
15:    return $node$
16: function Grow($d$)
17:    if $d = 0$ or $rand() < \frac{|A|}{|A|+|C|}$ then
18:        $node \leftarrow$ choose_random_element($A$)
19:    else
20:        $inner \leftarrow$ choose_random_element($C$)
21:        for $i \leftarrow 1$ to $arity(inner)$ do
22:            $arg.i \leftarrow$ Grow($d-1$)
23:        $node \leftarrow (inner, arg.1, arg.2, ..)$
24:    return $node$
```

frisbee placed in the right hand half of the arena, with $x > 0.2$. The experiment was started, and data was collected for 60 seconds starting at the point the robots start to execute their behaviour tree controllers.

The data collected was analysed by capturing the position of the frisbee at the time that controller execution started, and again 60 seconds later. Two runs were discarded due to battery failure during the run. Table 7.8 summarises the real robot runs, showing the start and finish positions of the frisbee and the fitness of the swarm. It is worth noting that in two of the runs, 9 and 15, the end position of the frisbee is at -0.88 m. Given the radius of the frisbee, this is its maximum $-x$ location, suggesting that the fitness of these two runs could potentially have been higher if the arena was longer or the start position more negative. Examining these runs further, in Run 9 the frisbee reached the arena boundary at 57 s and in Run 15 at 39 s, implying that we have reached a ceiling in performance and that the fitness function needs to modified to account for this. We revisit this in the next chapter.

Table 7.7: Results of ten evolutionary runs, showing the fitness and parsimony of the fittest individual, together with the mean fitness of the entire population.

Run	Fitness	Mean fitness	Parsimony
1	0.069	0.009	0.933
2	0.049	-0.001	0.555
3	0.068	0.007	0.838
4	0.057	0.008	0.963
5	0.068	0.001	0.925
6	0.081	0.006	0.873
7	0.100	0.020	0.798
8	0.044	0.009	0.883
9	0.049	0.007	0.704
10	0.089	0.003	0.501

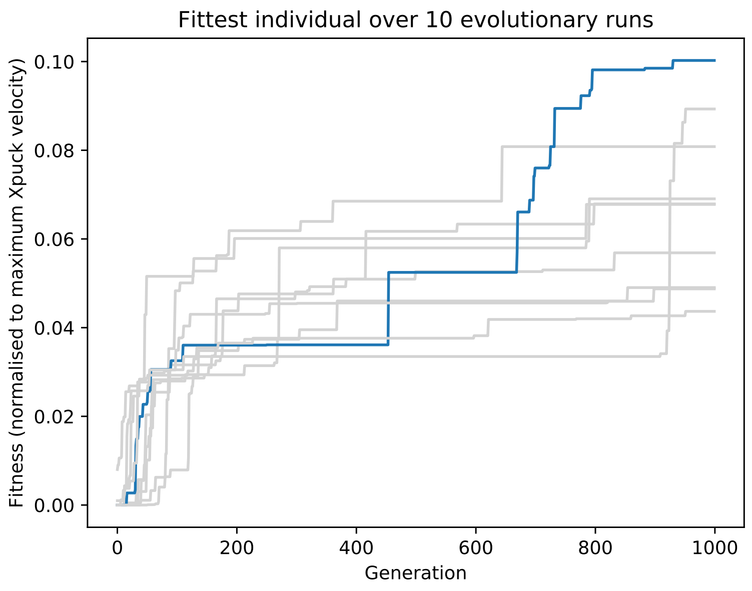


Figure 7.7: Evolutionary trajectory over ten runs, showing the fittest individual in each population. Blue line highlights the final fittest individual and the one transferred to the real robots.

In Figure 7.8 we show the results of simulating the fittest controller 500 times, and the results of running the same controller in real robots 18 times. We can see that the controller has transferred well from simulation to reality, with no significant loss in performance. This is also shown using independent two sample t-test; the null hypothesis of identical fitness cannot be rejected, $p = 0.36$.

7.4.5 Conclusion

In this chapter, we discuss controller transferability, or how to minimise the *reality gap*.

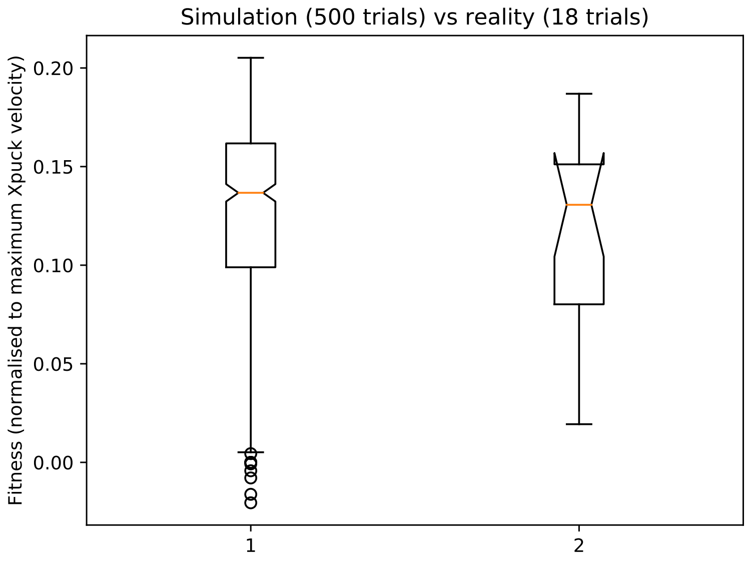


Figure 7.8: Fitness of fittest controller in simulation (500 trials) versus in reality (18 trials). There is no significant difference in controller performance.

In order to achieve this, we used multiple methods. Firstly, following Francesca *et al.* [2014a], we designed the behaviour tree architecture with a coarser granularity of possible behaviours than might exist with a neutral network controller. Reducing the representational power of the controller reduces the freedom of the automatic design process to exploit infidelities in the simulation. We also carefully tuned the simulator such that differences between it and reality were minimised as much as possible. We added noise to the simulation to mask infidelities. Finally, for the particularly problematic area of modelling collisions, we used the hierarchical structure of behaviour trees to impose collision avoidance as a default high priority behaviour, minimising time spent in these hard-to-model areas.

The results show that we have achieved a system which is capable of evolving controllers that then transfer successfully to the swarm of real robots, with no significant reality gap.

Table 7.8: Results from runs on real robots, showing the start and finish position of the frisbee, the distance moved and subsequent fitness. 20 runs were conducted, two were discarded due to battery failure during the run. Simulation mean and standard deviation also shown.

Run	Start (x)	Finish (x)	Distance	Fitness
1	0.638	-0.166	0.804	0.103
2	0.394	-0.109	0.502	0.064
3	0.469	-0.517	0.986	0.126
4	0.509	0.358	0.151	0.019
5	0.513	-0.588	1.101	0.141
6	0.458	-0.484	0.941	0.121
7	0.463	-0.698	1.161	0.149
8	0.692	-0.360	1.052	0.135
9	0.574	-0.884	1.458	0.187
10	0.430	-0.787	1.217	0.156
11	0.534	-0.650	1.185	0.152
12	0.562	-0.003	0.566	0.073
13	0.701	-0.357	1.058	0.136
14	0.295	-0.059	0.353	0.045
15	0.528	-0.881	1.409	0.181
16	0.559	-0.810	1.369	0.176
17	0.537	-0.015	0.552	0.071
18	0.590	-0.285	0.875	0.112
			$\bar{x}$	0.119
			σ	0.047
			Simulation $\bar{x}$	0.129
			Simulation σ	0.046

Chapter 8

In-swarm evolution

This chapter contains work that was published as *Onboard Evolution of Understandable Swarm Behaviors* in Advanced Intelligent Systems [Jones *et al.* , 2019].

A central aim of this work is to move towards the idea of an autonomous swarm. By this, we mean a swarm of robots performing useful tasks in the real world that is not reliant on exterior computational resources, and can potentially adapt to a changing environment. Much conventional swarm robotics relies on the discovery of controllers, for example by evolution, in a computationally expensive offline reality simulation process. We now move this offline process into the swarm, taking advantage of the processing power available in the Xpuck robot platform, and create a swarm system capable of generating a variety of complex behaviours autonomously and rapidly, with no external computing resources.

In the previous chapters, we have described the behaviour tree architecture we will use for our swarm robotics experiments, and the design of the Xpuck robot, which makes up the swarm of robots. We have talked about the design of the parallel robot simulator capable of running on the robots and detailed some results demonstrating the processing power of the swarm, both for running a distributed evolutionary algorithm, and for running a tag-tracking image processing task. We then looked at the reality gap and methods to overcome it, calibrating the simulator and mitigating areas of low fidelity to achieve good transference of evolved controllers from simulation to reality.

In this chapter we describe a system consisting of a swarm of Xpucks that is capable of evolving successively fitter controllers in simulation within the swarm that then run on the real robots for their collective transport task. Over the course of 15 minutes the swarm goes from having zero fitness to performing effectively in the real world. We then demonstrate one of the advantages of behaviour trees as a control architecture, that of human readability, by analysing some of the evolved trees, explaining how they work, and modifying them for performance improvements.

The structure of this chapter is as follows; firstly, we outline the collective task we aim to have the swarm perform. Since evolutionary algorithms are critically dependent on simulator speed, we look at possible alternatives to the conventional approaches used in the presence of noisy fitness functions, and devise a modified evolutionary algorithm that more effectively uses the available simulation budget. We next describe the incorporation of this modified algorithm into a distributed island model evolutionary algorithm, running on the robots of the swarm. The performance of this in-swarm evolutionary approach is evaluated, with successive controllers evolved in simulation being instantiated to run the real robots. Particular behaviour trees are analysed for understanding, and the implications of the heterogeneous swarm controllers is examined.

8.1 Benchmark task

The benchmark task is as described in the previous chapter, in Section 7.1.

8.2 Evolution with a noisy objective function

A major difficulty with evolving controllers for swarm robotics tasks are the twin problems of robustness and noise. Consider the scenario above, where the task is to move the frisbee to the $-x$ end of the arena. The objective function is noisy, in that slightly different starting conditions may result in quite different distances moved by the frisbee. The standard approach to combat this during the evolutionary process is by averaging over multiple simulation runs with different starting conditions, which we do. But a related issue is robustness; a robust controller solution will produce similar fitnesses over different starting conditions. This motivates some measure to grade the quality of the solution based on the variance of fitness, for example. But what if the solutions with high variability contain useful proto-behaviour that we do not wish to see prematurely optimised out of the population?

Observing many evolutionary runs, an early type of controller which produces non-zero fitnesses (both positive and negative) , and thus bootstraps into a regime where selection pressure can lead to better solutions, is simply to move forward. Combined with the default collision avoidance (`sel2 avoiding`) master tree, this inevitably leads to an xpuck eventually colliding with the frisbee and moving it, either in the $+x$ or $-x$ direction. These simple controllers show very high variability in fitness, but it is a prerequisite of any fitter controller that it does indeed move the robots.

At later stages, when fitter controllers have evolved, high variability seems to denote an overfitted controller; if the starting conditions are just right, we might get very performant behaviour, but the response to environmental perturbations is poor.

It would seem, then, that a modified evolutionary algorithm that was robust to noise and aware of variability may make better use of the available simulation budget. Jin & Branke [2005] survey approaches to the problem of evolution in uncertain environments. They divide uncertainty into four areas, noisy fitness functions, robustness of solutions[1], fitness approximation, and time-varying fitness functions. Clearly, we have a noisy fitness function, it is non-deterministic due to environmental noise and also depends on the starting conditions. But we are also using a fitness approximation; the true fitness function is that evaluated on the real swarm of robots, the fitness in simulation is an approximation that we use because it is much cheaper to evaluate than the impractical case of running millions of real robot experiments.

Common approaches to noisy fitness use explicit averaging, as we do in Chapters 4 and 7, increased population size, consideration of variance as an objective, modifications to the selection process to consider the effects of noise. We take inspiration from these, considering the particular characteristics of the simulator, which performs at its fastest when executing many (at least 256) simulations in parallel.

8.2.1 Comparison

The standard approach in evolutionary swarm robotics to dealing with the noisiness of the fitness function is to conduct multiple fitness evaluations of each individual in the solution population and taking the mean. But is this the best use of the simulation budget? Note that we use powers-of-two for repeats and population numbers because the parallel simulator described elsewhere runs most efficiently in this way. We compare two approaches using the benchmark problem fitness function which both use the same number of simulations, $n_{totsim} = 153600$, in total.

8.2.2 Modified evolutionary algorithm

The motivation for this algorithm was to increase the population as much as possible such that noise in the fitness function was implicitly averaged [Jin & Branke, 2005]. But to avoid potentially useful individuals being prematurely removed from the population due to a single poor evaluation, we reduce the replacement ratio to about 20% compared to the 95% for the old algorithm. This allows the accumulation of multiple fitness evaluations, increasing the confidence of the measure. We also introduce a modified tournament selector that is variance-aware.

We use the parameters in Table 8.1. The algorithm proceeds as follows. An initial population is created using *Ramped half-and-half* with a depth of n_{depth}. The fitness of the entire population is evaluated once and sorted. To form the new population,

[1]This is a different usage of robustness than above, applying to perturbations of the solution, rather than the environment.

the top $r_{elite} \cdot n_{pop} = 64$ are copied across untouched. The remaining individuals are either copied across unchanged, or with probability $p_{replace} = 0.25$ replaced with an individual generated either by crossover using a modified tournament selector from two elite parents with probability p_{xover} followed by the three mutation operators, or by a freshly generated individual. So, on average, the new population will have 25% elites, 56% non-elite but unchanged, 9.4% crossover and mutation, and 9.4% newly generated.

The fitness of the entire population is now evaluated. Many of these individuals will be unchanged from the previous generation, 81% in fact. We track the number of times an individual has been evaluated and maintain statistics of average fitness and variance over those multiple evaluations.

As the algorithm starts, the fittest individuals sometimes have only a single or a few evaluations, their high fitness due to chance in the noisy evaluation process, but as the algorithm proceeds, individuals with multiple evaluations and high average fitness tend to dominate. For a fair comparison with the old algorithm, when reporting the fittest individual within a generation, if the number of accumulated evaluations is below the old algorithm $n_{eval} = 8$ we scale the fitness by

$$f' = f \cdot \frac{number\_of\_evaluations}{n_{eval}} \tag{8.1}$$

The modified tournament selector uses instead of the average fitness, the 95% likelihood fitness, or if there is only one fitness evaluation of an individual chosen for the tournament, we halve the fitness. This exerts some selection pressure towards lower variability fit individuals.

The values of parameters for the comparison between the old and new algorithms are kept the same where this makes sense, for example the mutation, depth, and tournament size, and the values chosen are broadly what was used for the experiments in Chapter 7. As noted, population, evaluations and generations are kept to the same total simulation budget for a fair comparison. The values for the three new parameters r_{elite}, $p_{replace}$, and p_{xover} were chosen to ensure the replacement ratio was quite low and an average individual will accumulate at least several fitness evaluations.

We conducted ten evolutionary runs of each algorithm, with different starting seeds. Figure 8.1 shows the mean fitness across the ten runs for each algorithm. Because the new algorithm uses half as many simulations per generation, we show simulations as the x-axis for a fair comparison, since we want higher performance, and each simulation is approximately constant time. The new algorithm clearly performs

Table 8.1: Evolutionary algorithm comparison parameters

Parameter	Old	New	Description
n_{totsim}	153600		Total simulation budget
t_{sim}	30 s		Simulated time
n_{pop}	64	256	Population
n_{eval}	8	1	Evaluations per generation
n_{gen}	300	600	Generations
r_{elite}		0.25	Ratio of elite
n_{elite}	3	$(64)^1$	Number of elite
$p_{replace}$		0.25	Probability of non-elite replacement
p_{xover}		0.5	Probability of replacement by crossover
n_{tsize}	3		Tournament size
n_{depth}	6		Tree depth
$p_{mutparam}$	0.05		Probability of parameter mutation
$p_{mutpoint}$	0.05		Probability a node may be replaced by a node of the same arity
$p_{mutsubtree}$	0.05		Probability an individual may have a node replaced by a new subtree

^{1}Controlled by r_{elite}

better in this case, reaching the same fitness faster, with a higher final mean fitness across the runs after the complete simulation budget is used.

8.3 In-swarm evolution

We now have most of the necessary components to perform in-swarm evolution. We have a swarm of Xpuck robots, together with a simulator capable of running on the robot, using the GPU for performance acceleration. We have an evolutionary algorithm that can use the simulator to evolve controllers that transfer effectively to the real robots.

In this section, we describe how we turn the evolutionary algorithm into a distributed Island Model algorithm that runs on the whole swarm of nine Xpucks, previously touched on in Chapter 5. We use the island model in order that each robot can have a separate population, loosely coupled by limited migration of individuals between these separate populations. We then describe the experimental setup and protocol whereby the swarm evolves controllers in simulation in a distributed fashion, and periodically instantiates fit controllers to run the real robots. We modify the fitness function to better deal with the boundary conditions of the ends of the arena, to allow longer continuous runs to have meaningful measures of performance. By these means, after a short period of time (within 15 minutes) the real swarm has often evolved to perform well in the required task, with no external computation required.

A large variety of different evolved behaviours are observed. We examine some of

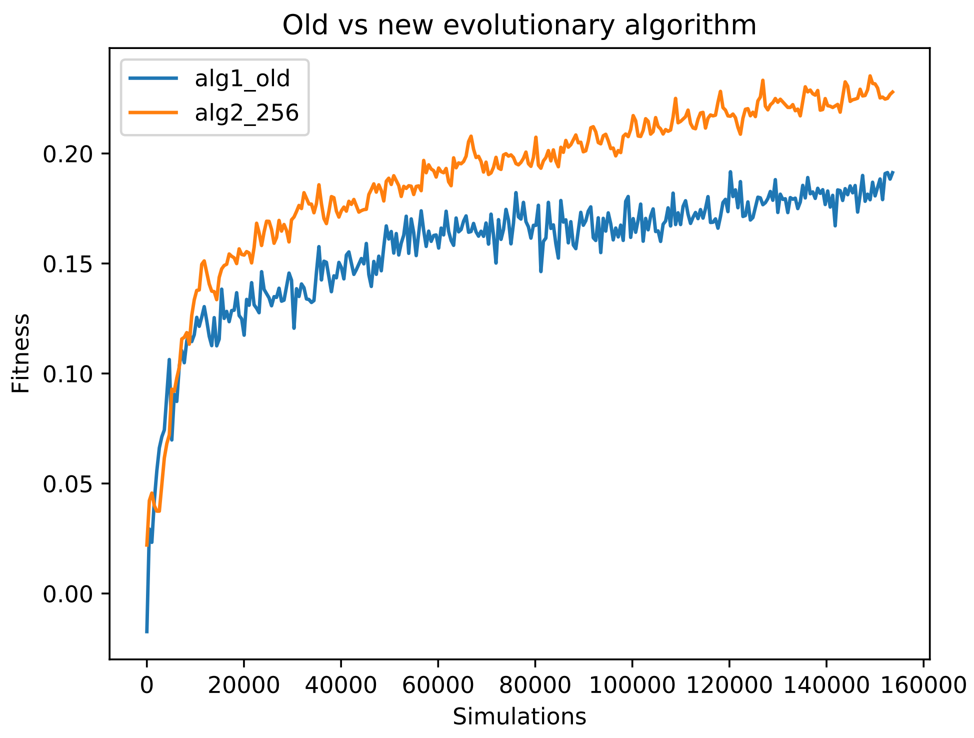


Figure 8.1: Old vs new evolutionary algorithm. In each case, ten evolutionary runs were conducted, with different starting seeds, plotting the mean of the fittest individuals in each of the ten runs. The old algorithm used a population of 64, with 8 evaluations per generation over 300 generations. The new algorithm used a population of 256 over 600 generations. In each case, the total number of simulations was identical.

the evolved controllers, using analysis techniques outlined in Chapter 3 and consider questions arising from the Island Model algorithm, such as the impact of heterogeneity on the swarm performance.

8.3.1 Island Model evolutionary algorithm

An Island Model evolutionary algorithm takes inspiration from theoretical biology [Wright, 1943] and investigations into the trajectory of natural evolution on islands, say, in the Pacific [Benzie & Williams, 1997]. Each island hosts a population of evolving individuals with its own evolutionary trajectory. In addition, there is some degree of interchange of genetic material between the island, a *migration* rate. The separation into sub-populations can result in higher performance than a single panmictic population of the same size due to niching effects and the maintenance of diversity [Whitley *et al.* , 1997]. In addition, by separating the total population into sub-populations with only a small amount of communication between then, we enable coarse-grained parallelism.

By hosting a sub-population on each Xpuck robot, we achieve scalable parallelism of the evolutionary algorithm; as the size of the swarm increases, so does the size of population hosted and thus performance.

We use the modified evolutionary algorithm with a population $n_{pop} = 256$. This is converted into an Island Model form in the following way. Within the Hub PC that logs experimental data and constructs the virtual senses based on the Vicon tracking data, we run a program called the *geneserver*. This mediates the migration of individuals between islands, or Xpucks. We can construct whatever connection topology we desire using the geneserver, for example physically based, so that individuals can only migrate between closely located Xpucks. For these runs, we use a fully connected topology so migration can occur between any two Xpucks.

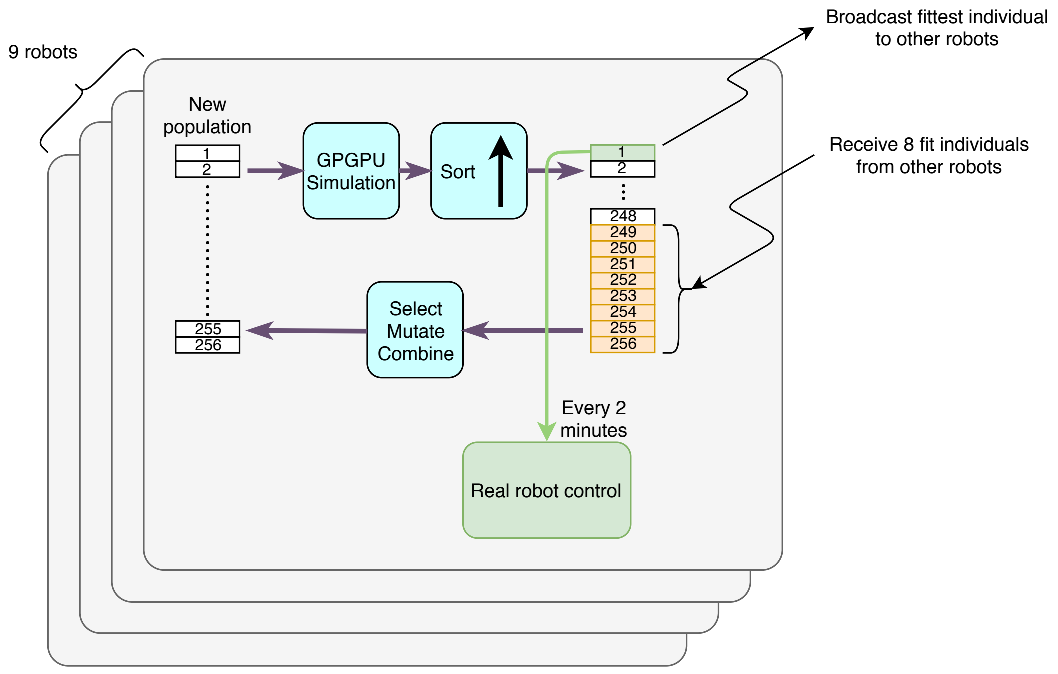


Figure 8.2: Illustration of the island model algorithm. A population of 256 individuals is maintained on each of the nine robots. This is evaluated for fitness in simulation and sorted. The fittest individual is broadcast, and the least fit eight individuals are replaced by fit individuals received from other robots. This population is then used to create a new population by selection, mutation, and combination. Every two minutes, the fittest individual is copied to take control of the real robot.

Once the experimental run has started, each Xpuck is running its own evolutionary algorithm. After each generation has been tested for fitness and sorted, the Xpuck broadcasts its fittest individual. This is the fittest individual with $n_{eval} = 8$ within the top half of the elite, or failing that, with $n_{eval} = 7$ and so on until a fittest has been found. This individual is kept within the geneserver. In return, from the geneserver, it receives eight fit individuals from the other Xpucks, which are used to replace the eight least fit individuals in its sub-population. This process of broadcast and replacement occurs just before the next generation is generated. In order that the process does not fluctuate too rapidly, the geneserver maintains a list of the eight most recently broadcast individuals from each Xpuck, and it is from these lists, excluding the originator, that the migrating eight fittest individuals are selected. Thus, the inward migration to an Xpuck will not necessarily contain individuals from all the other Xpucks. The complete cycle is illustrated in Figure 8.2.

The total population over the swarm is $9 \times n_{pop} = 2304$. The migration rate is the proportion of the individuals in the population that are replaced by migration each generation:

$$\begin{aligned} r_{migration} &= \frac{8}{n_{pop}} \\ &= 0.031 \end{aligned} \tag{8.2}$$

It should be noted that the process is asynchronous, each node or island broadcasts and receives individuals at the rate at which it completes generations, they do not run in lockstep. Also should be noted that migration does not remove individuals from their source, but instead copies them. This is common in implementations of Island Model evolutionary algorithms.

8.3.2 Fitness function

The fitness function encodes the benchmark foraging task, given in Section 8.1. In some of the earlier sections, we have referred to the task of the swarm pushing the blue frisbee in the $-x$ direction. For initial experiments and development of the evolutionary algorithms, what happened if or when the frisbee reached the border of the arena was not considered. In Section 7.4.4, this actually occurred in two of the 18 real-life runs.

The benchmark describes that if the frisbee reaches either the $+x$ or $-x$ boundary of the arena, i.e. the far right or left walls respectively, the robots will be temporarily halted[2] and the frisbee relocated to approximately the middle of the arena, before restarting the robots. The accumulated distance moved, not counting relocations, is used to calculate a normalised frisbee velocity.

Measuring fitness in this way means it is not dependent on the length of time the simulation or real experiment runs for, so fitnesses can be compared. But more importantly, we want to run long-lasting experiments, certainly longer than the 30 seconds or 1 minute experiments run so far. During the initial experiments, one particular failure mode was observed multiple times. If the frisbee ended up (by chance) at the $+x$ end of the arena, it was very difficult for the swarm to recover, even if it had a generally performant controller, the frisbee would often end up jammed in one of

[2]Including all evolutionary processes.

the corners of the arena surrounded by Xpucks ineffectually trying to get behind the frisbee. This didn't really matter when the test duration was limited to a minute or less, but for long lasting tests, this attractor, which is also at a point of large differences between simulation and reality (collisions between Xpucks, friction with floor and walls, many bodies physically interacting at the same time), made it hard to get reasonable behaviours. As with the solution to the problem of poor fidelity in Xpuck collision modelling, we adopted the approach of essentially removing this situation from the experiment.

By relocating the frisbee to the centre the instant it contacts one of the $+x$ or $-x$ boundary walls, we effectively transform the arena into an endless surface in the x direction, 1.5 m wide in the y direction. Analogously, the problem could be viewed as collective transport of objects within a pipe or corridor of which we can see a small segment.

The fitness function then becomes:

$$\begin{aligned}
f_{raw} &= -\frac{\sum \Delta x_{frisbee}}{t_{sim} \cdot v_{max}} \\
k_{penalty} &= \begin{cases} 1 & \text{if } \sum \Delta x_{frisbee} = 0 \\ 0 & \text{otherwise} \end{cases} \\
r_{derate} &= \begin{cases} 2 \cdot r_{parsimony} & \text{if } r_{parsimony} < 0.5 \\ 1 & \text{otherwise} \end{cases} \\
f_{evo} &= r_{derate} \cdot (f_{raw} - k_{penalty})
\end{aligned} \tag{8.3}$$

With the raw fitness value modified as described above. It is penalised for zero frisbee motion and derated to control bloat when the resource usage exceeds 50%.

One interesting problem emerged when we first started to use controllers evolved over a short period of simulation time (e.g. 1 minute) in runs that lasted long enough for the frisbee to be relocated. Apparently fit controllers were unable to respond to the relocation of the frisbee, seeming to 'lose' sight of it. This seemed to occur mostly when the Xpucks were more spread out than the initial configuration of all robots within $xy_{xpuck} : ([-0.5, -0.9], [-0.6, 0.6])$. For this reason, during the evolution of controllers, the initial configuration was randomly varied between the above, Scenario A, and Scenario B, with the robots spread over the whole arena: $xy_{xpuck} \in ([-0.8, 0.8], [-0.6, 0.6]$. Controllers evolved using this mix of starting scenarios were able to respond to relocations more effectively.

8.3.3 Behaviour tree architecture

The behaviour tree architecture is as used in the previous experiments. The set of leaf nodes L and blackboard registers B' we make available to the evolutionary algorithm are given in Table 8.2. Compared to the previous experiments, we have simplified it further, we no longer make available the blackboard registers associated with colours other than blue, and we remove the subtrees `explore`, `neighbour`, `fixedprob`, and `avoiding`. This was motivated by a desire to simplify, and we conducted some simple evolutionary tests with and without these subtrees with no clear difference in apparent evolutionary trajectories.

We create one new subtree, called `bsearch`, which returns *success* if there is blue visible in the central segment of the camera, otherwise moves slowly in a way controlled by the parameter i. The tree is shown in Figure 8.3. As well, we now have the `rotav` action node.

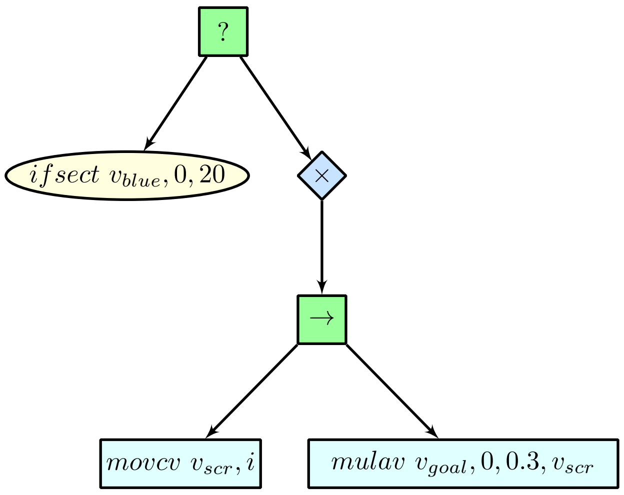


Figure 8.3: `bsearch` subtree. Move slowly in direction controlled by i until blue visible in central camera segment.

8.3.4 Experimental protocol

The nine Xpucks are placed in the arena at the left hand ($-x$) end with random orientation and with position as described in Section 8.3.2 scenario A. The blue frisbee is placed approximately in the centre of the arena. From the Hub PC, we monitor the state of the Xpucks and trigger the upload of an experiment file. At this point, the experimental run can be started. During the run, we can manually set the command state s_{cmd} of the swarm of robots to `PAUSE` and `RUN` in order to relocate the frisbee from either end of the arena back to the centre. During the `PAUSE` state, the robots do not move and the evolutionary algorithm does not advance.

While in the `RUN` state, the evolutionary algorithm proceeds on each Xpuck, with the fittest individual of each generation being advertised via the geneserver, and the

Table 8.2: Behaviour tree architecture subset. Blackboard registers addressed by index. Scalar instructions can access components of vector registers. d destination, s source, f float, i,j 8 bit signed, k,l 5.3 fixed

Name	Parameters	Description
Blackboard		
$\boldsymbol{v_{zero}}$	0 (R)	Zero. This register will always read as zero
$\boldsymbol{v_{goal}}$	2 (RW)	Goal velocity. Move in direction of vector when written
$\boldsymbol{v_{prox}}$	4 (R)	Proximity. Returns the vector sum of all the proximity sensors. $\boldsymbol{v_{prox}} = \sum_{i\in\{1,\ldots,8\}}(P_i, \angle q_i)$
$\boldsymbol{v_{up}}$	6 (R)	Upfield. Point to $+x$ end of the arena.
$\boldsymbol{v_{attr}}$	8 (R)	Attraction. Points to the nearest concentration of neighbouring robots. $\boldsymbol{v_{attr}} = \sum_{i=1}^{n}(\frac{r_{min}}{r_i}, \angle b_i)$
$\boldsymbol{v_{blue}}$	10 (R)	Blue. Points to blue blobs within the camera FOV. $\boldsymbol{v_{blue}} = (B_{left}, \angle 18.7) + (B_{centre}, \angle 0) + (B_{right}, \angle -18.7)$
s_n	12 (R)	Number of neighbours
s_{scr}	13 (RW)	Scalar scratch register
$\boldsymbol{v_{scr}}$	14 (RW)	Vector scratch register
Action nodes		
`movcs`	d, i	Set d scalar to constant $d \leftarrow i$
`movcv`	d, i	Set d vector to unit vector constant $d \leftarrow (1, \angle\pi\frac{i}{128})$
`mulas`	$d, s1, f, s2$	Scalar scale and add $d \leftarrow s1 + f \times s2$
`mulav`	$d, s1, f, s2$	Vector scale and add $\boldsymbol{d} \leftarrow \boldsymbol{s1} + f \times \boldsymbol{s2}$
`rotav`	$d, s1, i, s2$	Vector rotate and add $\boldsymbol{d} \leftarrow \boldsymbol{s1} + R(\pi\frac{i}{128}) \times \boldsymbol{s2}$
`ifprob`	s, k, l	S with probability governed by location of s on logistic curve defined by k, l. $P_{success} = \frac{1}{1+e^{k(l-bb[src])}}$
`ifsect`	s, i, j	S if the vector s is in sector defined by angle of centre of sector i and sector width j and length $>$ 0.1
`successl`		S always
`failurel`		F always
Subtrees		
`upfield`	g	Move towards or away from the $+x$ end of the arena at depending on sign of g
`attract`	g	Move towards or away from the nearest group of robots, depending on the sign of g
`bleft`[1]		S if blue in left of visual field
`bright`[1]		S if blue in right of visual field,
`bfront`		S if blue in centre of visual field `ifsect` $\boldsymbol{v_{blue}, \angle 0°, \angle 14°}$
`bsearch`	i	Turn at speed and direction based on i if there is no blue visible in centre of field of view. `success` if blue seen

[1] As noted in Section 7.4.2, due to an error, these subtrees are equivalent to `failurel`

least fit 8 individuals being replaced by the fittest eight individuals from the other Xpucks that are currently on the generserver. Every two minutes, the behaviour tree execution engine of each Xpuck loads the latest, fittest controller that the local evolutionary algorithm has generated. This controller takes over the running of the robot in the real world from that point until the next controller is loaded. After 16 minutes, the experiment is complete and the Xpucks are halted. During that 16 minutes, seven different controllers have run on each Xpuck.

All Vicon data, telemetry, and evolutionary algorithm data are logged for analysis. This includes the heritage and measured simulation fitness of every single individual within the whole island model evolutionary system, and the full behaviour trees of the fittest individuals of each island for every generation. Because the power consumption when running the simulator for the EA is high, the Xpuck battery life is about 1.5 hours, sufficient for about five runs.

As well as the control software synthesising the virtual senses and logging positional, telemetry and sensory data, we also run the *geneserver* on the Hub PC. This maintains a list of the eight most recent BT controllers sent by each Xpuck, and uses this set of lists to return back the eight fittest to each Xpuck after they send. As well as this migration function supporting the Island Model, each Xpuck sends abbreviated information about its entire population, which the geneserver logs. This makes it possible to reconstruct the entire lineage of any controller over the entire experimental run.

8.4 Data analysis

Data was processed in the following way. The files containing the position, telemetry, and geneserver logs are read in. Each line has a timestamp, quantised at 25 Hz. The lines of data are used to fill in fields of a large Pandas[3] dataframe, with one row for each discrete timestamp, and multiple columns for the frisbee and each Xpuck. the frisbee columns describe the pose tuple (x, y, θ), and the Xpuck columns describe the pose (x, y, θ), the command state s_{cmd}, the message round trip time t_{ping}, and a tuple giving some information on the currently fittest controller within the local island $(f, i_{gen}, i_{eval}, uid)$; fitness in simulation, generation, number of evaluations in simulation, and unique identifier.

We performed a total of 29 runs. Three runs suffered robot failure due to battery exhaustion and were dropped. Because virtual sense data is supplied over the WiFi connection and interference was present at some times of the day, we applied a quality test of no more than 5% of telemetry packets having a round trip time > 100 ms. Seven runs failed this test, one of which also suffered battery failure, leaving 20 runs which we analysed. The overall performance of the swarm is summarised in Table 8.3, which shows the final average fitness of the island model evolutionary algorithm,

[3]https://pandas.pydata.org/

the real fitness of the swarm in each two minute segment that it was running an evolved controller. It also shows the average round trip telemetry ping times and number of dead robots.

Table 8.3: Summary of all in-swarm evolutionary runs. D is number of dead robots during run, P is 1 if disqualified for more than 5% of ping times above 100 ms. $\overline{t_{ping}}$ is the mean ping time over the whole run in ms, %BP is the percentage of bad pings in the run. f_{EA} is the average final fitness across the swarm of the island model evolutionary algorithm, and $f_{real1}..f_{real7}$ are the real fitnesses of the swarm over each two minute segment of the run.

Run	D	P	$\overline{t_{ping}}$	$\%BP$	f_{EA}	f_{real1}	f_{real2}	f_{real3}	f_{real4}	f_{real5}	f_{real6}	f_{real7}
1	0	0	27.60	0.29	0.219	0.001	-0.113	0.152	0.098	-0.004	-0.081	0.174
2	0	0	58.17	3.43	0.163	0.023	0.036	-0.035	0.016	0.020	0.043	0.028
3	0	0	28.08	0.22	0.220	0.048	-0.086	0.106	-0.024	0.141	0.123	0.116
4	2	0	55.32	3.01	0.184	0.027	0.034	0.077	0.002	-0.030	-0.015	-0.017
5	0	0	27.95	0.15	0.233	0.034	0.017	0.038	0.042	0.039	-0.023	0.085
6	0	0	28.15	0.13	0.203	0.065	-0.063	0.040	0.108	0.148	-0.077	0.097
7	0	0	28.81	0.76	0.259	-0.025	0.030	-0.031	-0.019	0.015	0.125	0.240
8	0	0	28.32	0.27	0.266	-0.080	-0.032	0.175	0.188	0.110	0.073	0.204
9	0	0	27.92	0.23	0.213	-0.005	0.172	0.072	0.126	0.079	0.015	0.171
10	0	1	46.29	7.96	0.221	0.042	-0.018	-0.080	-0.023	0.028	0.032	-0.017
11	0	1	73.44	12.81	0.189	-0.039	0.052	0.105	-0.003	0.054	0.056	-0.013
12	0	1	46.87	7.31	0.155	-0.050	0.030	-0.019	0.087	-0.002	0.029	0.074
13	1	0	49.49	2.93	0.187	-0.047	0.044	0.027	0.076	0.000	0.090	0.120
14	0	0	27.76	0.49	0.154	0.093	0.014	0.092	0.039	0.000	0.062	0.038
15	0	0	28.17	0.51	0.221	-0.043	-0.015	0.191	0.132	0.062	0.059	0.005
16	0	0	34.25	1.56	0.299	-0.004	0.043	-0.025	0.105	0.114	0.194	0.200
17	0	0	37.04	3.16	0.227	0.009	0.038	-0.096	0.121	0.089	0.146	0.154
18	0	0	37.45	3.75	0.180	-0.036	-0.016	-0.139	-0.001	0.101	-0.017	0.199
19	0	1	66.79	9.80	0.190	-0.063	0.027	-0.021	0.003	0.055	0.012	-0.014
20	0	0	31.49	2.36	0.164	0.070	-0.009	0.074	0.097	0.085	0.028	0.108
21	0	0	35.87	3.60	0.167	0.054	-0.061	-0.023	0.127	-0.055	0.063	-0.069
22	0	1	43.83	6.31	0.183	-0.100	-0.049	-0.003	0.048	0.072	0.088	0.113
23	0	0	31.24	2.51	0.184	-0.104	0.048	0.107	0.075	0.049	0.076	-0.082
24	0	0	28.22	2.18	0.189	0.022	-0.024	0.082	-0.069	0.098	-0.035	0.160
25	0	0	27.81	2.07	0.183	-0.047	0.013	0.022	0.053	0.063	0.004	0.124
26	0	0	28.19	2.23	0.180	-0.056	-0.026	0.038	-0.025	-0.021	-0.088	-0.197
27	0	1	37.40	6.34	0.212	0.058	0.053	0.086	0.062	0.018	0.016	0.084
28	0	0	27.40	1.84	0.182	0.064	0.086	0.085	0.008	0.094	-0.035	-0.061
29	1	1	79.94	6.95	0.183	0.022	0.157	0.115	0.094	-0.013	0.097	0.063

Table 8.4 summarises the results for all good runs.

8.4.1 Island model

The distributed island model evolutionary algorithm running on the swarm completed an average of 84, $\sigma = 11.1$ generations per run giving a mean generation time of 11.4 s. The swarm ran a total of 3.9 million simulations[4]. Figure 8.4 shows the mean fitness of the island model genetic algorithm running on the swarm for each of the 20 good runs. This is calculated by taking the fitness of the fittest individual on each island, or Xpuck, at a particular time, derating the fitness if there had been less than 8 evaluations, and taking the average across the swarm.

The runs that ended with the highest and lowest mean fitness are highlighted, as is the average over all runs. What is noteworthy is the high effectiveness of the island model, the average final fitness is 0.20 in 84 generations, compared to the single node

[4]20 good runs x 84 generations x 9 robots x 256 simulations per generation

Table 8.4: The 20 good runs, showing f_{EA} is mean fitness of the island model evolutionary algorithm, and f_{realx} the real fitness for each segment of the 16 minute runs. Mean and standard deviation over all runs shown at bottom of table. Real fitness values $f > 0.1$ shown bolded.

Run	f_{EA}	f_{real1}	f_{real2}	f_{real3}	f_{real4}	f_{real5}	f_{real6}	f_{real7}
1	0.219	0.001	-0.113	**0.152**	0.098	-0.004	-0.081	**0.174**
2	0.163	0.023	0.036	-0.035	0.016	0.020	0.043	0.028
3	0.220	0.048	-0.086	**0.106**	-0.024	**0.141**	**0.123**	**0.116**
5	0.233	0.034	0.017	0.038	0.042	0.039	-0.023	0.085
6	0.203	0.065	-0.063	0.040	**0.108**	**0.148**	-0.077	0.097
7	0.259	-0.025	0.030	-0.031	-0.019	0.015	**0.125**	**0.240**
8	0.266	-0.080	-0.032	**0.175**	**0.188**	**0.110**	0.073	**0.204**
9	0.213	-0.005	**0.172**	0.072	**0.126**	0.079	0.015	**0.171**
14	0.154	0.093	0.014	0.092	0.039	0.000	0.062	0.038
15	0.221	-0.043	-0.015	**0.191**	**0.132**	0.062	0.059	0.005
16	0.299	-0.004	0.043	-0.025	**0.105**	**0.114**	**0.194**	**0.200**
17	0.227	0.009	0.038	-0.096	**0.121**	0.089	**0.146**	**0.154**
18	0.180	-0.036	-0.016	-0.139	-0.001	**0.101**	-0.017	**0.199**
20	0.164	0.070	-0.009	0.074	0.097	0.085	0.028	**0.108**
21	0.167	0.054	-0.061	-0.023	**0.127**	-0.055	0.063	-0.069
23	0.184	-0.104	0.048	**0.107**	0.075	0.049	0.076	-0.082
24	0.189	0.022	-0.024	0.082	-0.069	0.098	-0.035	**0.160**
25	0.183	-0.047	0.013	0.022	0.053	0.063	0.004	**0.124**
26	0.180	-0.056	-0.026	0.038	-0.025	-0.021	-0.088	-0.197
28	0.182	0.064	0.086	0.085	0.008	0.094	-0.035	-0.061
$\overline{x}$	0.205	0.004	0.003	0.046	0.060	0.061	0.033	0.085
σ	0.037	0.053	0.062	0.084	0.066	0.054	0.076	0.113

shown in Figure 8.1 which only reaches a similar average fitness after 600 generations. This should be expected with the much larger population of 2304 (9×256) vs 256, but does demonstrate that the island model partitioning into a parallel system with low migration rate has scaled quite effectively. We define a scaling coefficient β as:

$$\begin{aligned}\beta &= \frac{f_{island} \cdot n_{gen\_single} \cdot n_{sim\_single}}{f_{single} \cdot n_{gen\_island} \cdot n_{sim\_island} \cdot n_{nodes}} \\ &= \frac{0.20 \cdot 600 \cdot 256}{0.21 \cdot 84 \cdot 256 \cdot 9} \\ &= 0.76 \end{aligned} \tag{8.4}$$

Which is a measure of the ratio of achieved fitness per simulation between the single node and the island model. With perfect scaling, β would be 1.

8.4.2 Real life behaviour

The behaviour of the swarm in real life shows a clear increase in fitness over the runs. As Figure 8.5 shows, in the early stages of the experimental runs, the two segments starting at 2 minutes and 4 minutes, the fitness is not significantly above zero, and

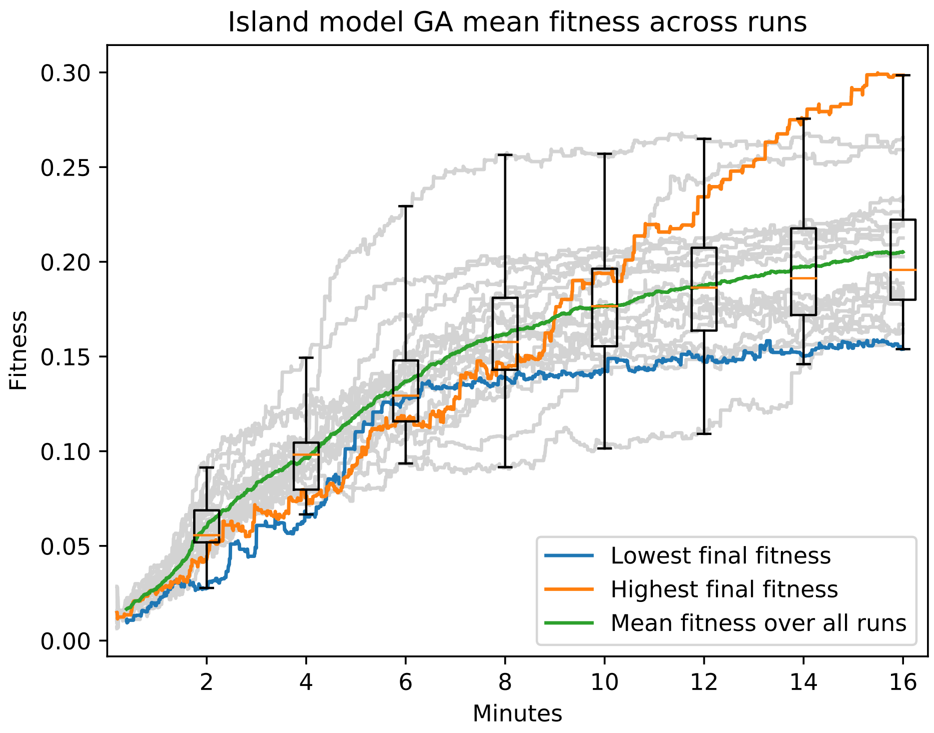


Figure 8.4: Mean fitness of the island model evolutionary algorithm over each of the 16 runs. Boxplot whiskers cover full range

in the final segment starting at 14 minutes, the fitness at a mean of about 0.085 is significantly above zero. There is an overall gain in fitness between the first segment starting at 2 minutes and the final segment starting at 14 minutes (independent T-test null hypothesis $p = 0.008$).

But there are very large variations in the fitness between runs. One problem with analysing the data is that, conventionally, evaluating a swarm controller in a real swarm involves multiple runs with the same controller. Here each run has a different, heterogeneous, and varying mix of controllers. We don't know whether the variation in fitness is due to chance, some property of the controllers affecting transference, or some effect of heterogeneity. We return to this in Section 8.7.

8.5 Behavioural analysis

When watching videos of the real swarm over multiple runs, it is apparent that there is a rich variety of behaviours that solve the problem of collective movement of the frisbee, not captured by the bottom line fitness measure. The approach we take to analyse and gain insight is as follows. Firstly, we define several behavioural metrics, which we can automatically calculate from the captured trajectory data of the swarm. We then associate these metrics with individual two-minute segments

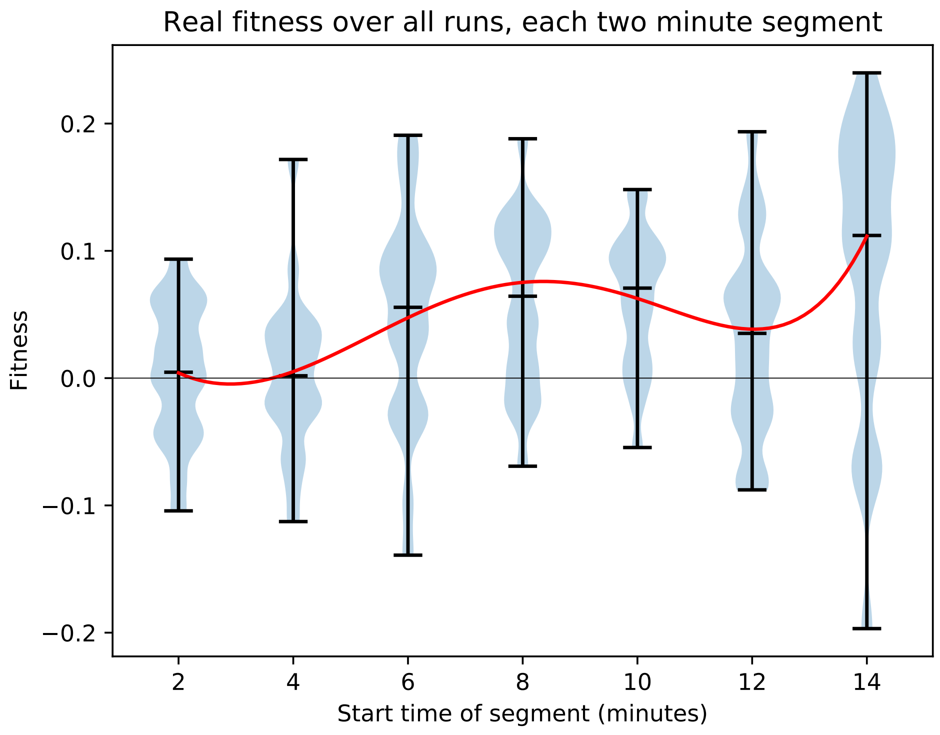


Figure 8.5: Real fitness of the swarm over time across all runs. Violin plots show distribution of fitnesses over runs, with ticks at median and extrema. Red line is 5^{th}-order polynomial fitted to medians of each segment

during which the swarm is executing a fixed set of behaviour tree controllers. In general, the segments near to the end of an experimental run will have greater real fitness, but we have already seen that there is wide variance in this measure. Some of this may be due to chance, some may be due to discovery then loss of high-performing behavioural traits.

We take all the segments that have a reasonable segment fitness, defined here to be $f > 0.1$, and shown bolded in Table 8.4, and perform a behavioural cluster analysis motivated by the subjective impression of quite different solution strategies. From different clusters, representing a different solution style, we can then analyse the behaviour tree controllers themselves, using the identities defined in Chapter 3 to simplify the trees such that we can gain understanding of their functionality. In doing this, we hope to discover useful or interesting behavioural traits, or subtrees, of the behaviour tree controllers.

The metrics we define are:

1. Energy $m_{energy} = \sum_{i \in robots} |v_{mleft}(i)| + |v_{mright}(i)|$. The total use of the motors.

2. Pushing m_{push}, the average proportion of the robots that are within 1 frisbee

radius plus 1.5 robot radii of the centre of the frisbee.

3. Loitering m_{loit}, the average proportion of robots that are within 3 frisbee radii but not in the pushing zone.

4. Cooperation $m_{coop} = \frac{1}{n \cdot r_{pushing}} |\sum_{i \in pushing}(1, \angle\theta_i)|$, the degree to which the robots in close proximity to the frisbee are facing in the same direction, thus can push cooperatively.

5. Acceleration $m_{acc} = \sum |\frac{\Delta v_{mleft}}{\Delta t}| + |\frac{\Delta v_{mleft}}{\Delta t}|$. The sum of all motor absolute motor accelerations, a measure of how jerky the motion is.

A total of 33 segments in the good runs have a fitness $f > 0.1$. We cluster these using a Self Organising Map [Kohonen, 1982] which tries to arrange the data in cells such that topological relations in the high dimensional feature space are somewhat preserved in the two-dimensional representation.

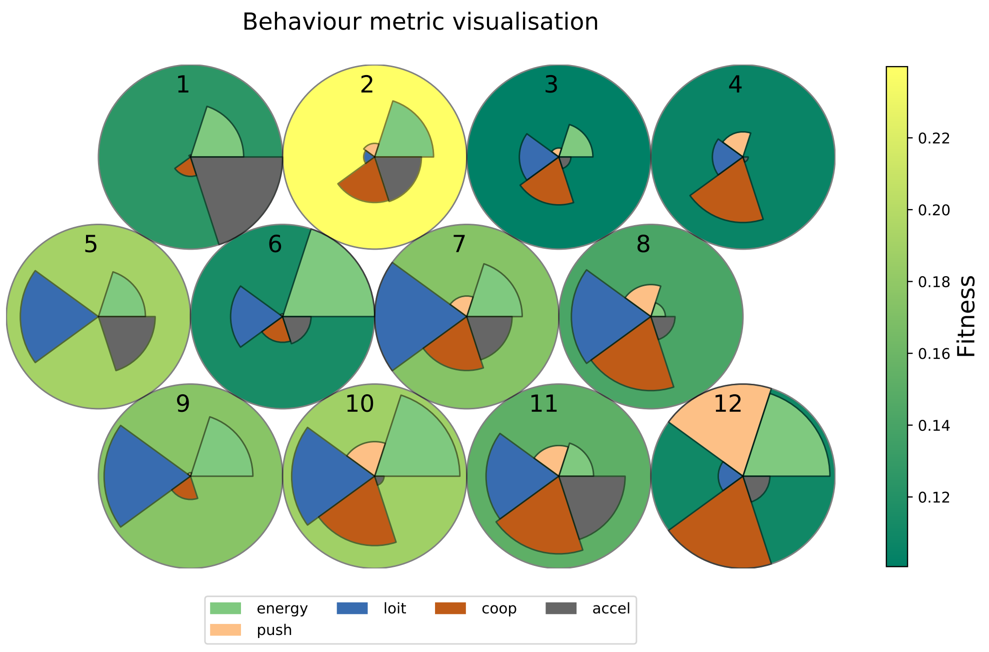


Figure 8.6: Self Organised Map of the 33 segments of swarm behaviour with fitness $f > 0.1$. Each segment is clustered into a cell with other segments with similar behaviour. Background colour shows the average fitness of the cell. The wedges within the cells shows the relative values of the five different behavioural metrics, with larger radii representing higher values. The metrics are m_{energy} the total motor use, m_{push} robots close to the frisbee, m_{loit} robots near the frisbee, m_{coop} alignment of robots close to frisbee, and m_{accel} the jerkiness of motion.

Table 8.5: Cell map, showing which segments are present in each cell. Segments are denoted r, s where r is run number, s is segment, from 1 to 7.

Cell	Segments				
1	7,6	9,2	15,4	17,4	24,7
2	7,7	9,4	16,4	18,7	
3	18,5	23,3			
4	3,3	6,4	6,5	25,7	
5	15,3				
6	16,5				
7	1,7	9,7	17,6	17,7	
8	3,5	3,6	20,7		
9	8,3	16,7	21,4		
10	8,4	8,7	16,6		
11	1,3	3,7			
12	8,5				

Figures 8.6 shows the map and Table 8.5 the experimental run segments associated with each cell in the map. The background colour of the Cells shows the fitness, ranging from dark green in Cell 4 representing $f = 0.11$ to light yellow in cell 2, representing $f = 0.24$. The radii of the segments within the cells show the values of each of the behavioural metrics. The three fittest runs in real life are Run 7, Run 8, and Run 16. Run 8 and Run 16 are in adjacent cells, so would be expected to have more similar behaviour than to Run 7. Run 16 is interesting for another reason, that its final population of controllers is dominated by a single controller.

For these reasons of interestingness and potentially different solution styles, we will examine behaviour trees from Run 16 final segment, present in Cell 9, and from Run 7 final segment, present in Cell 2.

8.6 Analysis of trees

We have previously stated that one advantage of behaviour trees is their human readability, and thus the potential to analyse an evolved behaviour tree to see how it works and gain inspiration and understanding. In this section, we perform this analysis. In the previous section, we used a number of behaviour metrics to cluster the numerous segments of runs in order to select behaviour trees which we wish to analyse and potentially gain insight from. We will examine Run 16 final segment, and Run 7 final segment. From the clustering, we might expect that Run 7 would have more jerky movement with more clustering around the frisbee, while Run 16 would have smoother, more spread-out behaviour.

Table 8.6 shows the unique identifiers of the behaviour trees controlling each robot in the final segment of each run. These are integers in the form `rriiiii` where `rr` is the robot index where it was created, and `iiiii` is the monotonically increasing index of individuals created on a given island, or robot.

Table 8.6: All behaviour tree unique IDs in the final segment of each run. Least significant five digits are a monotonically increasing ID on each island, most significant one or two digits are the host ID where the tree originated. R is the run number, U is the number of unique behaviour trees in the swarm

R	U	xp03	xp04	xp05	xp06	xp07	xp08	xp09	xp11	xp15
1	9	308064	1506821	507640	1105859	707946	807872	907786	1108289	1508281
2	9	305755	406009	505113	605148	704998	805299	905512	800225	1502169
3	5	504991	504991	506545	504991	706215	902792	504991	504991	405581
4	9	307807	408586	507203	608188	707639	808227	908015	1108614	1507138
5	9	308324	306624	508080	307405	707717	808170	908240	307405	505420
6	8	302593	302593	1106509	607880	1105527	807734	908185	1106442	1505266
7	8	308016	408292	905166	1506491	707625	905166	908257	305404	1507078
8	9	307600	407477	507291	306004	707923	1506320	908505	1108214	505628
9	9	307683	407955	507595	706960	708113	807513	1106746	1107776	1508083
10	9	306608	407092	605935	606461	705612	807112	1106288	1107194	1506976
11	7	903272	606246	903272	608503	806366	606246	401770	906612	906484
12	8	307462	306185	306272	606982	305341	807244	907772	306185	1507621
13	9	306001	605230	507214	606792	707513	404924	906577	1107160	1506860
14	9	307169	706805	507372	306008	707259	705436	704194	1107182	1507469
15	8	307974	407849	603515	606957	708261	603515	908100	1107788	1507850
16	3	906737	806768	806768	906737	806768	807914	806768	806768	806768
17	7	305274	1506957	706956	1506957	307278	606884	1506957	1108241	1507591
18	9	805433	407752	1106209	606027	707490	1505063	907057	1107408	804212
19	9	306400	406626	507422	607151	1503917	804932	906282	1107006	1506660
20	7	803974	407398	508277	608033	803974	808233	803974	803974	507386
21	8	702114	606104	506235	505572	702114	705968	1105525	705968	1507265
22	9	307557	408022	508100	404458	707508	1106431	907640	1506587	1507577
23	9	306817	407149	506735	606893	603987	806397	907093	1107085	1506847
24	5	502488	403680	502488	502488	1503399	803837	502488	1103697	502488
25	5	803424	803253	902478	902426	803253	803621	803253	803424	803253
26	8	304226	403850	503810	503384	703634	1503195	904215	503384	403644
27	7	1503270	1502624	504096	702823	1503270	1502624	904529	1103852	302326
28	9	403614	404053	504186	604074	1503664	803867	602348	1103811	1503957
29	8	1500868	502233	502233	603296	703586	803703	903506	1103408	1503275

We start with Run 16, which has the highest final fitness in the EA of 0.299, and a final segment fitness in reality of 0.2, with steadily increasing real fitness over most of experimental run. From Table 8.6 we can see that there are only three unique behaviour trees; `806768, 906737, 807914`. BT `806768` is interesting because it is fit enough to have migrated from its origin on robot xp08 to robots xp04, xp05, xp07, xp09, and xp11, which we can see from the prefix 8, and then to have survived and remained the fittest in those destination robots. This suggests that it is *consistently* fit.

We then move on to Run 7 final segment, which has the third highest fitness in the EA and the highest in reality of 0.259 and 0.24 respectively. This segment has one repeated tree, so there are eight unique trees, numbered `308016, 408292, 905166, 1506491, 707625, 908257, 305404, 1507078`.

8.6.1 Automatic tree reduction

Recall from Section 3.1.7, we state a series of identities that apply to behaviour trees. These require that subtrees are classified as containing only query leaf nodes Q, that is, node that do not alter the state of the blackboard, or subtrees containing action leaf nodes C that can affect the state of the blackboard. We also need to know whether we can guarantee that a subtree will return success or failure.

Table 8.7 shows the node types, expressed using the notation from Section 3.1.7. Trees are expressed as recursive functions. C is a subtree that may change the state of the blackboard, Q is a subtree or node that does not. Action nodes may change blackboard state and can be regarded as a subtree C. Success may be a leaf node S returning success and not changing state, or a function enclosing a subtree $S(t)$ returning success. Similarly for failure F and $F(t)$. A tree C with a known value of *success* or *failure* is denoted $S(C)$ or $F(C)$ respectively.

Table 8.7: Node types and side effects

Node	Type	Side effect
`movcs`	$S(C)$	yes
`movcv`	$S(C)$	yes
`mulas`	$S(C)$	yes
`mulav`	$S(C)$	yes
`rotav`	$S(C)$	yes
`ifprob`	Q	no
`ifsect`	Q	no
`successl`	$S(Q)$	no
`failurel`	$F(Q)$	no
`upfield`	$S(C)$	yes
`attract`	$S(C)$	yes
`bleft`	Q	no
`bright`	Q	no
`bfront`	Q	no
`bsearch`	C	yes

The reduction rules are summarised below and are applied in a depth-first recursive manner. Any rule applying to a subtree C also applies to Q:

1. Replace decorated queries $S(Q) \rightarrow S$ and $F(Q) \rightarrow F$
2. Replace decorated known trees $S(S(C)) \rightarrow S(C)$ and $F(F(C)) \rightarrow F(C)$
3. For `seq` nodes, remove any children to the right of $F(C)$ because they will never be ticked
4. For `sel` nodes, remove any children to the right of $S(C)$ because they will never be ticked

5. For `seq` nodes, remove any $S(Q)$ child that is not the rightmost since it cannot affect the result

6. For `sel` nodes, remove any $F(Q)$ child that is not the rightmost since it cannot affect the result

7. `seq` or `sel` with a single child are replaced by the child

8. Collapse multiple levels of `seq` and `sel` nodes to a single wide level

Sometimes one pass through the list of transformations exposes further opportunities for reductions. We therefore repeat the transformations until there are no more changes to the tree. Also, Rule 8 is useful for producing clearer diagrams but can result in trees that cannot be executed, due to the fixed arity sizes that are supported for `seq*` and `sel*` in the behaviour tree interpreter code. We disable this rule for the purpose of generating executable reduced trees.

Since the reduction rules are identities, the execution of a correctly reduced tree must result in identical behaviour, anything else indicating bugs in the process. To validate our reductions, we run a simulation with nine robots executing the original and reduced tree for 60 simulated seconds, in each case producing a log file containing the poses of all objects at every timestep, together with all sensor inputs and actuator outputs. Any differences in the logfiles indicates non-equivalence.

8.6.2 Run 16 overview

Before analysing the trees `806768, 807914, 906737`, we first perform a pairwise functional comparison between trees to eliminate those which are differently labelled but functionally identical. This shows that tree `807914` is functionally identical to `806768`. There are differences in the tree, but these are in never triggered branches. Tree `806768` is present in seven of the nine robots, and tree `906737` in the remaining two. Table 8.8 summarises the unique trees of Run 16 and their characteristics.

Table 8.8: Run 16 tree characteristics. Fitness evaluated over 1000 runs of 60 s with different starting conditions.

Tree	`806768`	`906737`
Original nodes	134	235
Reduced nodes	20	36
Reduction	85%	85%
Fitness $\bar{x}$	0.27	0.23
Fitness σ	0.044	0.093

8.6.3 Analysis of Run 16 tree `806768`

Behaviour tree reduction

We apply the reduction algorithm to tree `806768`. The original and reduced versions are shown in Figure 8.7. It is clear that there is a large amount of redundancy. The original has 134 nodes and the reduced form has 20, an 85% reduction. Even so, it is not immediately obvious what the tree does. Let us analyse it. From here, we refer to the listing form in Listing 8.1. The left-most[5] subtrees are more significant to behaviour, so we start there.

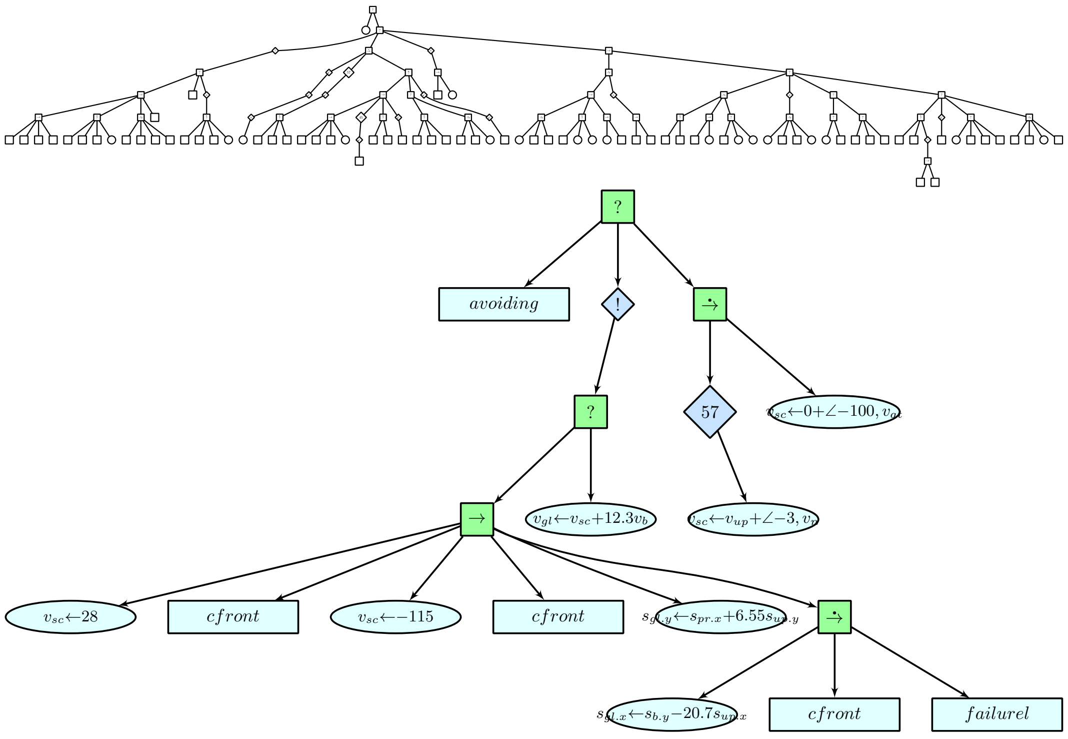

?
avoiding
!
$\dot{\rightarrow}$
?
57
$v_{sc} \leftarrow 0 + \angle -100, v_{at}$
$\rightarrow$
$v_{gl} \leftarrow v_{sc} + 12.3 v_b$
$v_{sc} \leftarrow v_{up} + \angle -3, v_n$
$v_{sc} \leftarrow 28$
$cfront$
$v_{sc} \leftarrow -115$
$cfront$
$s_{gl.y} \leftarrow s_{pr.x} + 6.55 s_{up.y}$
$\dot{\rightarrow}$
$s_{gl.x} \leftarrow s_{b.y} - 20.7 s_{up.x}$
$cfront$
$failurel$

Figure 8.7: Tree `806768` shown in the original and automatically reduced form.

[5]or top-most in the listing

Listing 8.1: Reduced behaviour tree `806768`

```
 1  sel
 2      avoiding
 3      invert
 4          sel
 5              seq
 6                  movcv $v_{scr}$  28
 7                  bfront
 8                  movcv   $v_{scr}$ -115
 9                  bfront
10                  mulas   $v_{goal.y}$ $v_{prox.x}$ 6.545845 $v_{up.y}$
11                  seqm4
12                      mulas $v_{goal.x}$ $v_{blue.y}$ -20.683918 $v_{up.x}$
13                      bfront
14                      failurel
15              mulav $v_{goal}$ $v_{scr}$ 12.297516 $v_{blue}$
16      seqm
17          repeati    57
18              rotav      $v_{scr}$ $v_{up}$ -3 $(s_n, s_{scr})$
19          rotav      $v_{scr}$  0   -100      $v_{attr}$
```

The `sel avoiding` is the standard prefix that we use for all evolved trees to perform basic collision avoidance before any other behaviours. If the robot is not performing the `avoiding` action, then the subsequent trees to the right are ticked. There are two further subtrees of the top level `sel`, starting at lines 3 and 16 of the listing. The `invert` subtree will always fail, although it will produce side effects in the process. We can see this by noting that the `sel` clause below the `invert` has a final child of `mulav`, which will always return *running* or *success*.

If we look at the third subtree, starting at line 16, we can see that it only affect the scratch blackboard register $\boldsymbol{v_{scr}}$. But if we look at the second subtree, starting at line 3, $\boldsymbol{v_{scr}}$ is always written, at line 6. Thus the third subtree is redundant and we can remove it. By removing that subtree, the `invert` become redundant, since it makes no difference what the return values of the last subtree of the entire tree is. Also, note that `seqm` only behaves differently from `seq` if it has children that can return *running*. This is not the case here, since the write to the blackboard goal register component $s_{goal.x}$ is the first such within the tree. The multiple `bfront` queries are redundant after the first. Finally, the second `movcv` at line 8 targeting $\boldsymbol{v_{scr}}$ is redundant, since the goal vector $\boldsymbol{v_{goal}}$ will be busy after the writes in lines 10 and 12, meaning the `mulav` at line 18 will not take effect (will return *running*) if `bfront` is *success*. This also makes the `failurel` at line 14 redundant. We can therefore further simplify the tree, verified as behaviourally identical, and shown in Listing 8.2.

Listing 8.2: Reduced simplified behaviour tree `806768`

```
1 sel
2     avoiding
3     sel
4         seq
5             movcv $v_{scr}$  $(1, \angle 39°)$
6             bfront
7             mulas $v_{goal.y}$  $v_{prox.x}$  6.55  $v_{up.y}$
8             mulas $v_{goal.x}$  $v_{blue.y}$  -20.7 $v_{up.x}$
9         mulav $v_{goal}$  $v_{scr}$ 12.3 $v_{blue}$
```

Single robot behaviour

We can now see that there are two behaviours, depending on whether the blackboard register $\boldsymbol{v_{blue}}$ is non-zero and pointing forwards, that is, there is something blue in the centre of the field of vision of the robot. If the robot is directly facing something blue, it will perform one behaviour (lines 7 and 8) labelled B1, otherwise it will perform the behaviour of lines 5 and 9, labelled B2. We can restate this as:

$$\boldsymbol{v_{goal}} = \begin{cases} \begin{bmatrix} v_{blue.y} & -20.7 \cdot v_{up.x} \\ v_{prox.x} & +6.5 \cdot v_{up.y} \end{bmatrix} & \text{if directly facing frisbee (B1)} \\ (1, \angle 39°) + 12.3 \cdot \boldsymbol{v_{blue}} & \text{otherwise (B2)} \end{cases} \tag{8.5}$$

If not directly facing the frisbee, the behaviour B2 is quite simple to state; the robot will move forward in an anticlockwise circular fashion until something blue enters the visual field, at which point it will move forward while turning in that direction until the frisbee is in the centre of the visual field, at which point the other behaviour B1 takes control. Figure 8.8 visualises a simulation of the behaviour tree with the robot starting at pose $(0.3, 0, 0)$, so facing in the $+x$ direction away from the frisbee.The location of the robot is shown for each timestep of 100 ms over a period of 3 s. The colour of the trail indicates which behaviour is executing, yellow indicating B2 and green B1. We can see that the robot circles in an anticlockwise direction until the blue frisbee comes into view, at which point the robot heads more towards the frisbee. Finally the behaviour switches to B1.

If robot is directly facing the frisbee, triggering behaviour B1, the goal vector is formed of several components and the meaning is not immediately clear. By observing that the components of the $\boldsymbol{v_{up}}$ vector dominate the maximum values that might be seen from $s_{prox.x}$ and $s_{blue.y}$ of ≈ 2 and ± 0.32 respectively, the majority of $\boldsymbol{v_{goal}}$ is formed from the $\boldsymbol{v_{up}}$ vector reflected in the robot y-axis and anisotropically scaled. If the pose angle θ of the robot is in the range $|\theta| < \pi/2$ this will cause the robot to rotate to face in the $+x$ direction and stop. With angles greater than this,

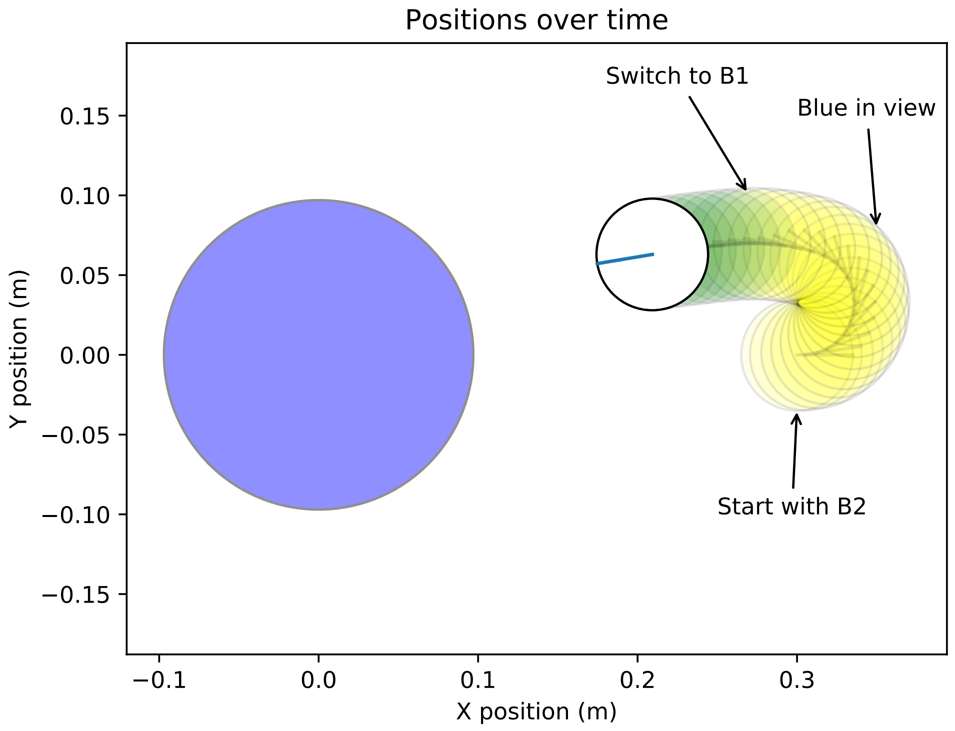


Figure 8.8: Visualisation of the path of a single Xpuck following behaviour B2 then B1. The robot starting pose is $(0.3, 0, 0)$, facing in the $+x$ direction away from the frisbee. Colour of trail is yellow for B2 and green for B1, each plot of trail is one control cycle of 100 ms

$|\theta| > \pi/2$, i.e. facing in the $-x$ direction, the robot will move forward while turning to face the $+x$ direction. The rate of turning is dependent on the angle of the robot, so at an angle of exactly π the robot will move forward with no turning, but any deviation will result in an accelerating turn towards the $+x$ direction.

Consider two scenarios, both with the frisbee at $(0, 0)$, and the robot with pose $(0.2, 0, \pi)$ in the first, and $(-0.2, 0, 0)$ in the second, shown in the diagram in Figure 8.9. In the first scenario, the robot will move forwards until it contacts the frisbee, then pushing the frisbee in the $-x$ direction. In the second scenario, the robot will not move. We can see that randomly choosing between these two scenarios will on average result in an increase in fitness, since the frisbee will only ever move in the $-x$ direction. If we perturb the first scenario slightly, with a robot starting pose of $(0.2, 0, \pi - 0.1)$, the robot will move forward while turning clockwise. The frisbee will again be pushed in the $-x$ direction, but as the robot continues to rotate, it will reach the situation where the vector $\boldsymbol{v_{blue}}$ no longer has zero angle (from the possible angles of $-18.7, -9.35, 0, 9.35, 18.7$) and thus `bfront` will return *failure* and behaviour B2 will occur. This will tend to make the robot move forward and turn anticlockwise towards the frisbee, while pushing it. If the turning rate is fast enough then the robot will end up fully facing the frisbee again, such that the first behaviour is again activated. We can see that we might have a switching of behaviours of clockwise and anticlockwise forward movement such that the frisbee is on average moved in the $-x$ direction.

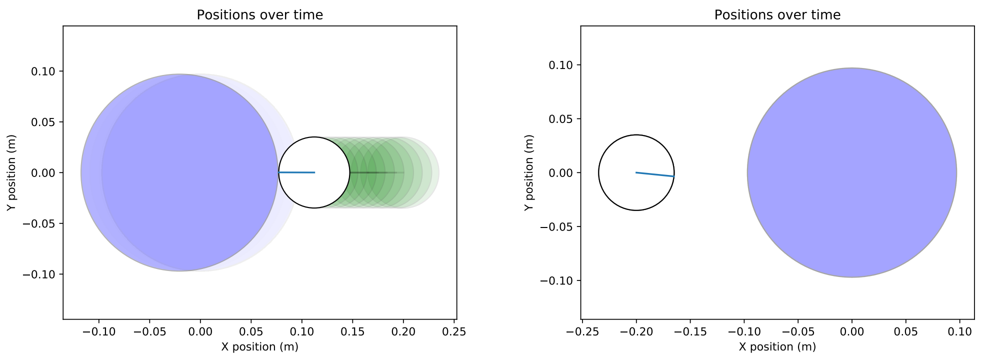


Figure 8.9: Visualisation of the path of a single Xpuck following behaviour B1 in two scenarios

In actual fact, a single robot does not reliably turn far enough that the frisbee becomes centred in the field of view, so sometimes the frisbee will get pushed in a circular path and sometimes on an erratic path towards $-x$ depending on the exact starting condition. Figure 8.10 shows the evolution of the perturbed first scenario, with B1 resulting in a slow clockwise turn intially, then a period of rapid switching between B1 and B2, a further period of B1 turning anticlockwise this time, then finally a stable situation running B2 with the robot pushing the frisbee in a circular path.

Interaction of two robots

What is interesting is if we change the scenario to have two robots in contact with the frisbee. In this case, although neither individually can stably push the frisbee, with two robots their interactions produce an emergent stable pushing behaviour. It is important to realise that these interactions now include the default collision avoidance behaviour, not shown in Equation 8.6.3. This usually causes a robot to turn on the spot away from the object detected with the IR proximity sensors and is visible on the trail visualisation as denser outlines at points where the robots are not moving forwards.

Figure 8.11 shows an example of this. The initial configuration of the system was the frisbee at location $(0.65, 0)$ and the robots at poses $(0.8, 0.05, \pi)$ and $(0.8, -0.05, \pi)$. The track of the frisbee is not straight, but never degenerates into a stable orbit. The two robots use varying amounts of B1 and B2, depending on the system state. By inspection, when the frisbee path is tending upwards too much, the top xpuck starts spending more time in B2 and the less in B1, causing the system of frisbee and robots to turn back downwards, and likewise in the opposite situation, with collision

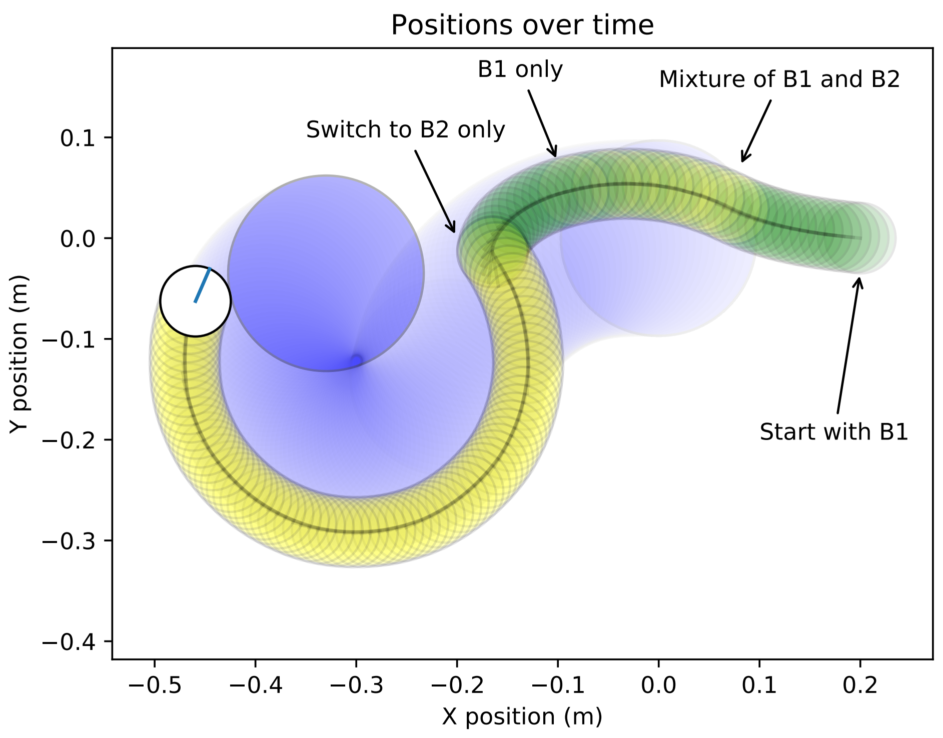


Figure 8.10: Visualisation of the path of a single Xpuck following behaviour B1 at the start, then combinations until ending in a stable orbit in behaviour B2.

avoidance ensuring that the other robot is turned to maintain some separation.

To show that this behaviour is active, rather than a chance interaction, we compare what happens with a simple tree that performs forward movement with collision avoidance, shown in Listing 8.3.

Listing 8.3: Tree for forward motion with collision avoidance: `forward`

```
sel
    avoiding
    movcv $v_{goal}$ $(\mathbf{1}, \angle\mathbf{0})$
```

Figure 8.12 shows the initial configuration and the results for trees `806768` and `forward`. The starting configuration was frisbee at $(0.65, 0.5)$ and the robots at $(0.6, 0.65, -\pi/2)$ and $(0.7, 0.65, -\pi/2)$. This is the same relative positions and poses as previously, but rotated so the Xpucks are starting out facing in the $-y$ direction. The simple forward movement behaviour tree quite successfully pushes the frisbee, because they stay relatively aligned, but in the wrong, $-y$, direction. The tree `806768` on the other hand actively corrects its heading such that the frisbee is pushed in the $-x$ direction.

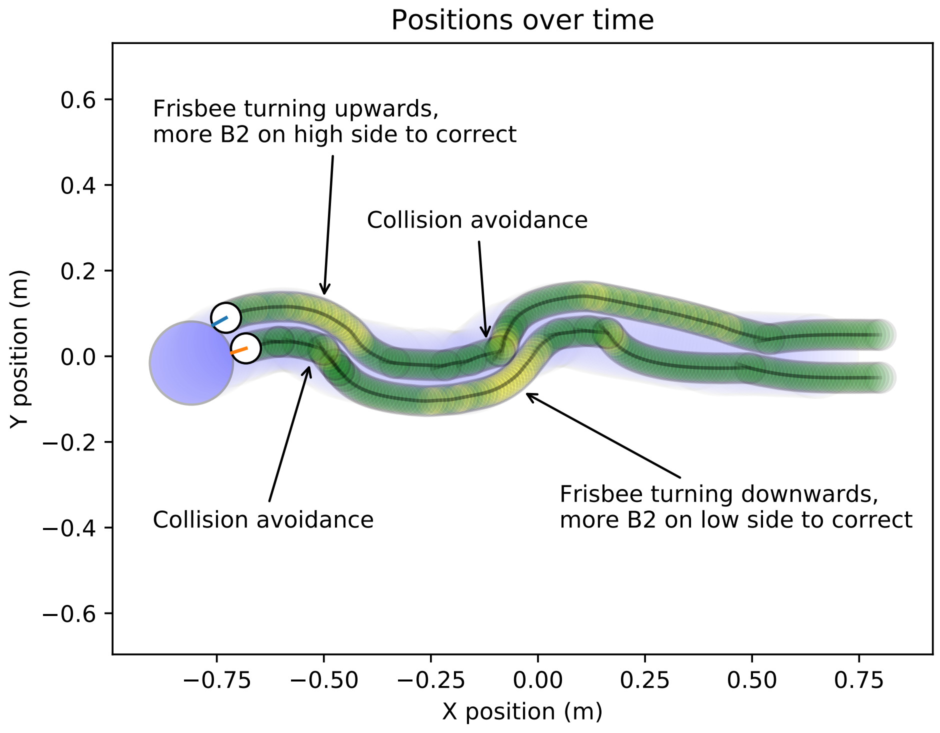


Figure 8.11: Visualisation of the path of two Xpucks with a starting position close to the right of the frisbee

Resilience to perturbation

We want swarm controllers to be resilient, because they will be more general, and able to perform well whatever the starting conditions. How resilient is this system to perturbations in the starting poses? We approach this in the following way, comparing the performance of the tree `806768` with the simple `forward` tree described above. We start with the frisbee at position $(0, 0)$ and the two robots are placed with random poses centred on $(0, 0, 0)$ with added Gaussian noise of standard deviations $(0.2, 0.2, 1.5)$. The robots and frisbee are not allowed to overlap. Valid configurations are simulated for a time of 8 s, just less than the time for a perfect attempt to push the frisbee to an x boundary. A total of 100000 simulations were run for each tree. Each tree has a best configuration, that having the highest fitness. The mean starting distance of the robots from this configuration is measured for each run and this data is binned and plotted against the fitness of that run.

Figure 8.13 shows the results. there are clearly two quite different types of behaviour here. As you might expect, if you run a lot of trials of essentially a random walk (move forward with collision avoidance), there will be some that are quite fit, but the majority will not be. The data shows this, with the `forward` tree having most runs clustering around a zero fitness. The `806768` tree, on the other hand, maintains

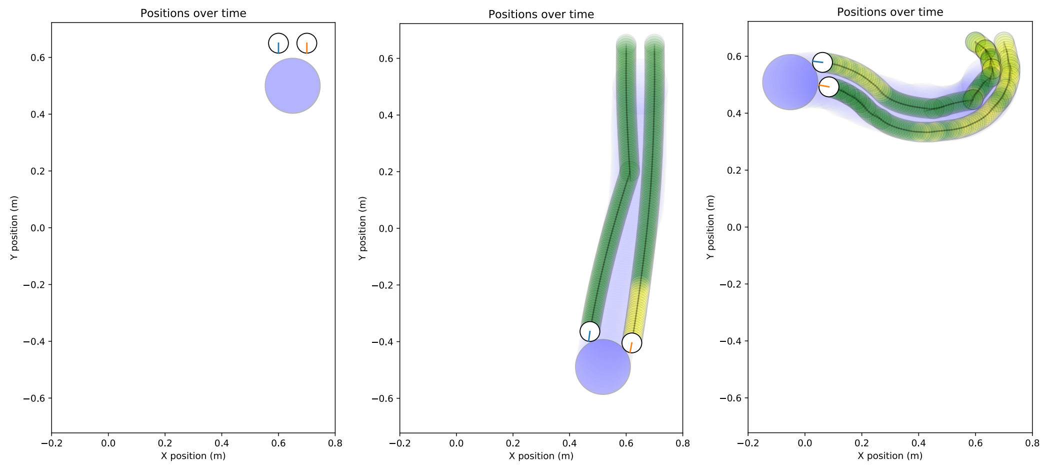


Figure 8.12: Visualisations of the path of two Xpucks. The starting configuration is shown on the left, the centre shows the effect of a simple behaviour tree performing just collision avoidance and forward movement, and figure on the right shows tree `806768` actively steering towards the correct $-x$ direction.

a consistent median fitness, which falls gradually as the mean starting distance from the frisbee increases, as you would expect since the robots have to reach the frisbee before pushing it. The important indicator of an effective controller is that the median fitness is maintained even over a quite large increase in the distance away from the frisbee, implying the active movement towards and then pushing of the frisbee.

Effect of swarm size

One interesting question about swarm controllers is whether they produce emergent behaviour. It is not obvious how to answer this, but one approach would be to measure the fitness of the controller when running in different sized swarms. If there was no emergent behaviour, we would expect a single agent to have a certain degree of fitness $f = f_{agent}$, then n agents to have a higher fitness $f = k \cdot f_{agent}$ but with $k < n$ since multiple agents with no cooperation or emergent behaviour may interfere, and for our task there is a physical limit on how many agents can actually interact with the frisbee. We expect sublinear scaling, in other words. Conversely, with emergent cooperation in the swarm we may see superlinear scaling when cooperation outweighs interference. Superlinear performance scaling has been observed in swarm robotics systems, [Mondada *et al.* , 2005], and [Hamann, 2012] develop a simple model of swarm performance comprising two components of cooperation and interference.

We simulated the tree `806768` at different swarm sizes up to $n = 16$. Figure 8.14 shows the results. There is clearly superlinear performance scaling up to a swarm size of $n = 7$. Above seven robots, the performance scaling is sublinear as the system

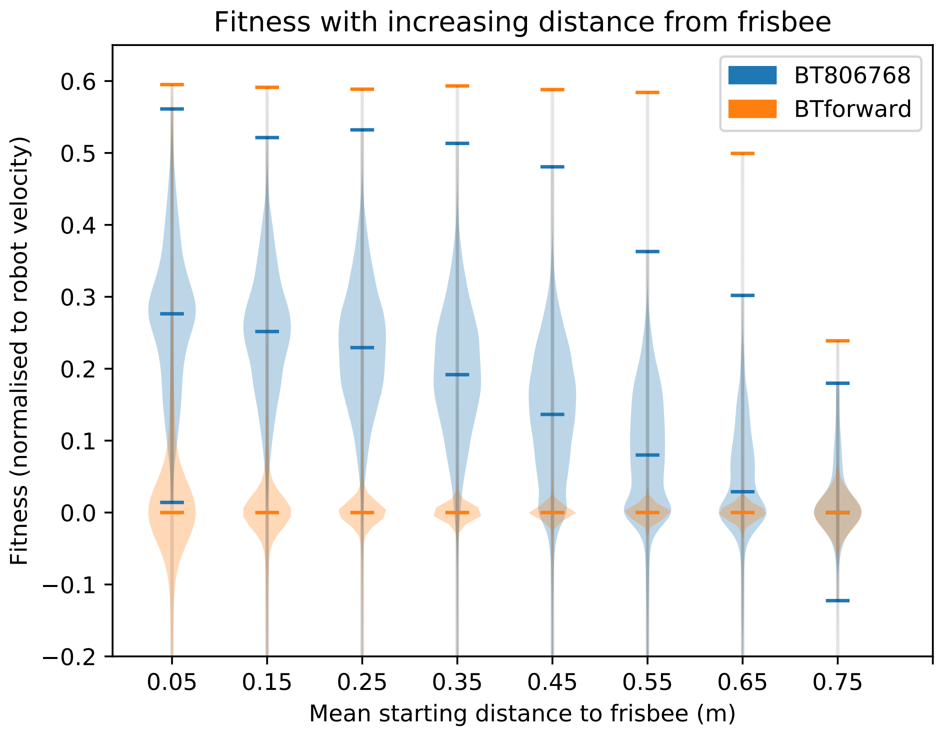


Figure 8.13: Distribution of fitness of two robots over 100000 runs of each of two trees against mean starting distance from frisbee. Ticks show extrema and medians.

performance reaches a plateau of around $f = 0.3$. Above a certain number of robots, there can be no improvement in performance, since there is only a single frisbee and the robots have a maximum velocity. We regard the superlinear scaling as evidence of emergent collective behaviour.

8.6.4 Analysis of Run 16 tree `906737`

We have looked in detail at the single tree `806768` because it dominates the swarm, being present as the controller in seven out of the nine robots. But what of the other tree? Is it a small variation on `807678` or does it have have quite different behaviours? What is the effect of this heterogeneity on the performance of the swarm?

Tree `906737` is present in two of the nine robots. We apply the reduction algorithm and similar manual transformations as used above to give thhe tree shown in Listing 8.4, verified as functionally identical.

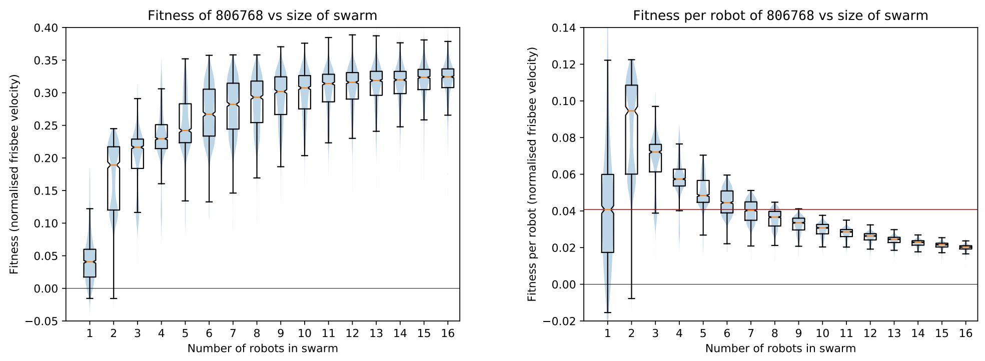


Figure 8.14: Scalability of swarm performance with increasing swarm size. Each point measured with 60 s simulated time over 1000 simulations with different starting conditions.

Listing 8.4: Reduced simplified behaviour tree `906737`

```
1   sel2
2       avoiding
3       sel2
4           seq2
5               seq2
6                   movcv    $v_{scr}$      $(1, \angle -162°)$
7                   bfront
8               seqm4
9                   mulas    $v_{goal.y}$   $v_{prox.x}$   30.8   $v_{up.y}$
10                  mulas    $v_{goal.x}$   $v_{blue.y}$   -31.1  $v_{scr.x}$
11                  mulas    $v_{goal.x}$   $v_{scr}$      12.6   $v_{goal.x}$
12                  failurel
13          mulav    $v_{goal}$   $v_{scr}$   11.1   $v_{blue}$
```

Single robot behaviour

There are some interesting subtleties to understanding this tree. We can see similarities with the previous tree in that the `bfront` node at line 7 controls the behaviour. If the robot is directly facing the frisbee we get one behaviour we will again call B1, and if not, we get behaviour B2, governed by lines 6 and 13. B2 causes a clockwise rotation until there is blue within the field of view, then movement towards the direction of the blue. This is similar to tree `806767` except that the direction of rotation is reversed.

When directly facing the frisbee, the following occurs. In the first control cycle the nodes on lines 9 and 10 result in successful writes to both components of the v_{goal} vector. As a result, the node in line 11, which also wants to write to a component of the v_{goal} vector cannot complete and returns *running*. Because this node is the child of a `seqm`, a node with memory, at the next control step the nodes at lines 9 and 10 are ignored and the node at 11 is *tick*ed. This time the node can succeed, and a write to v_{goal} occurs. Subsequently, the `failurel` means that the node at line 13 is ticked, which attempts to write to v_{goal} again. Ordinarily, this should return *running*, since a previous node has already written to it in this cycle.

Here, the effect of an implementation choice of the behaviour tree interpreter is exposed. In order to tell if the v_{goal} register has been written in a cycle, it is initialised to NaN[6] (Not-a-Number) at the beginning of a tick update. The code of any writing node checks if the destination contents are still NaN, and if they are, this flags that an update is still allowed in this cycle. However, the node at line 11 reads from this register to form the result to write back into it. Any arithmetic operation involving a NaN as one of the operands will result in a NaN, meaning that a NaN is written back into v_{goal}, giving it the *appearance* of not having been written this cycle. Hence, in this case, the node at line 13 *will* succeed in writing.

$$\boldsymbol{v_{goal}} = \begin{cases} \begin{bmatrix} \frac{1}{2}(v_{blue.y} & +26.7 + 11.1 \cdot v_{blue.x}) \\ \frac{1}{2}(v_{prox.x} & +6.5 \cdot v_{up.y} + 11.1 \cdot v_{blue.y} - 0.3) \end{bmatrix} & \text{if directly facing frisbee (B1)} \\ (1, \angle - 162^\circ) + 11.1 \cdot \boldsymbol{v_{blue}} & \text{otherwise (B2)} \end{cases} \tag{8.6}$$

The overall effect of this is that any time directly facing the frisbee (behaviour B1) results in cycle-by-cycle alternation between writes from lines 9 and 10, and writes from line 13. We show this in Eqn 8.6.4 as the linear combination of the two. We can further simplify this by noting that the only values of $v_{blue.x}$ that can occur in behaviour B1 are $1, 1.89, 2.95$. The x component of v_{goal} will therefore always be 18.9 or higher. $v_{blue.y}$ will always be zero in behaviour B1, and v_{prox} will be low, otherwise collision avoidance will have been triggered. A simplified approximation is thus:

$$\boldsymbol{v_{goal}} = \begin{cases} \begin{bmatrix} 20 \\ 3.3 \cdot v_{up.y}) \end{bmatrix} & \text{if directly facing frisbee (B1)} \\ (1, \angle - 162^\circ) + 11.1 \cdot \boldsymbol{v_{blue}} & \text{otherwise (B2)} \end{cases} \tag{8.7}$$

Behaviour B1 then becomes essentially moving forward, with a maximum deviation from that of about 10° ($\tan^{-1} \frac{3.3}{20}$), controlled by the v_{up} vector. When the Xpuck is facing in the $-x$ direction, the effect is positive feedback causing the robot to turn increasingly away from the frisbee, until behaviour B2 is triggered, causing it to turn back towards the frisbee again. So we expect to see quite similar behaviour to `806768` when facing the frisbee in the $-x$ direction, but in the $+x$ direction we

[6]The behaviour of NaN is defined in the floating point standard IEEE754-2008, followed by compliant OpenCL implementations.

will instead see forward movement, tending to push the frisbee the wrong way.

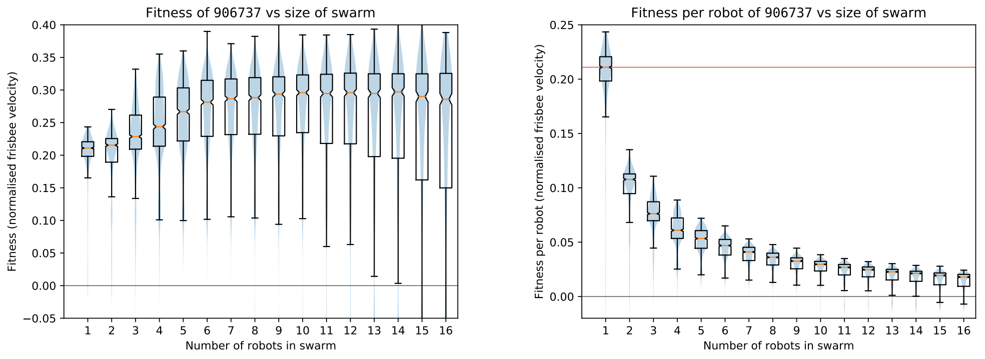


Figure 8.15: Tree `906737` scalability of swarm performance with increasing swarm size. Each point measured with 60 s simulated time over 1000 simulations with different starting conditions.

When looking at simulations of different swarm sizes, we see that the single robot performance of `906737` is significantly better than `806768` but as the swarm size increases, performance does not scale nearly as well, plateauing earlier and with much higher variance. There is no superlinear scaling, due to the high performance of a single robot.

8.6.5 Effect of heterogeneity

An inevitable effect of the distributed island model evolutionary algorithm is a heterogeneous set of controllers running the swarm. We see from this analysis that in Run 16 there are only two trees present. Tree `806768` on seven of the nine robots, and tree `906737` on the remaining two. Tree `806768` has better performance in larger swarms, whereas tree `906737` performs well at an individual level. How do they behave in a mixed swarm? We measured the fitness of each possible mixture of the two trees in simulation for 60 s, with 1000 simulations at each point. Figure 8.16 shows the results. On the left, all the robots are running `806768` and on the right `906737`. We can see a smooth variation between the characteristics of the two. On the left, there is higher fitness and smaller variance. As we introduce greater numbers of tree `906737` the fitness falls and the variability rises.

8.6.6 Analysis of Run 7

Run 7 final segment consists of the trees `305404, 308016, 408292, 707625, 905166, 908257, 1506491, 1507078`. We first conducted pairwise functional comparisons between all trees to eliminate duplicates. This showed that `305404` and `707625` were identical, leaving seven unique trees. The overall characteristics of these trees are shown in Table 8.9

Six of the seven trees have fundamentally the same structure. This is shown in

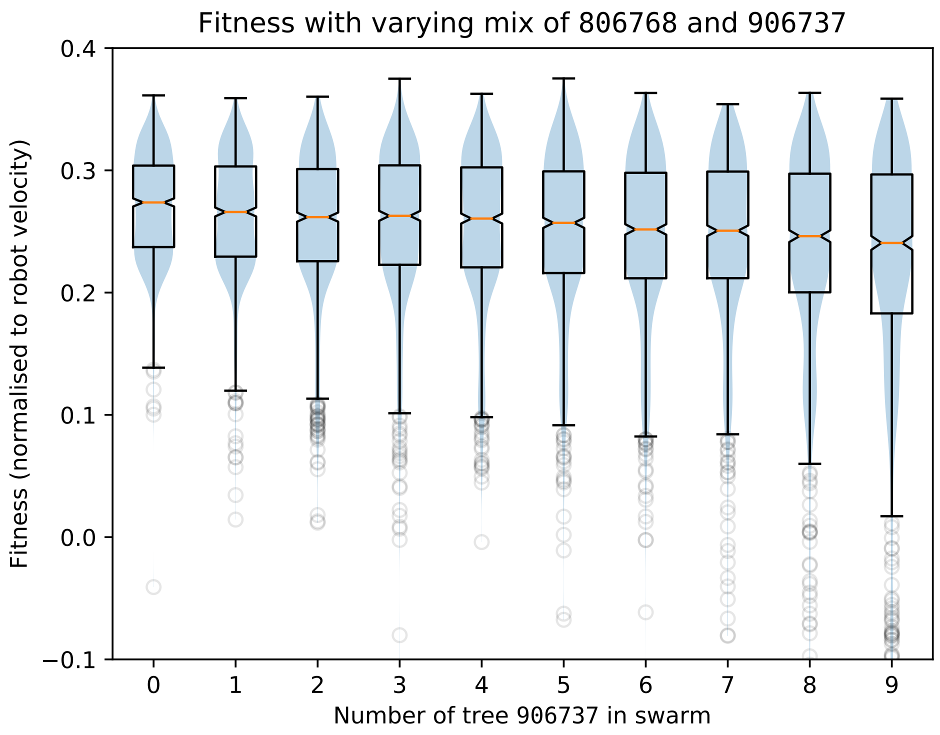


Figure 8.16: Effect on fitness of a variable mixture of trees `806768` and `906737`. 1000 simulations, each of 60 seconds per boxplot.

Table 8.9: Run 7 tree characteristics. Fitness evaluated over 1000 runs of 60 s with different starting conditions.

Tree	`305404`	`308016`	`408292`	`905166`	`908257`	`1506491`	`1507078`
Original nodes	74	68	67	68	74	56	112
Reduced nodes	13	15	15	15	15	15	46
Reduction	82%	78%	78%	78%	80%	73%	60%
Fitness $\bar{x}$	0.20	0.21	0.20	0.20	0.20	0.20	0.21
Fitness σ	0.090	0.090	0.093	0.090	0.085	0.094	0.099

Listing 8.5, with the particular values of a, b, c, d shown in Table 8.10.

Listing 8.5: Common tree structure

```
sel
    avoiding
    sel
        seq
            bfront
            rotav $v_{goal}$  $v_{blue}$  $a$    $v_{up}$  $(B1)$
        seq
            rotav $v_{scr}$   $b$    $c$   $(s_n, 0)$        $(B2)$
            mulav $v_{goal}$  $v_{scr}$  $d$  $v_{blue}$          $(B2)$
```

Table 8.10: Variations on common structure

Tree	a	b	c	d
`305404`	$7°$	v_{up}	$-90°$	23.4
`308016`	$24°$	$(s_n, 0)$	$148°$	23.4
`408292`	$24°$	v_{up}	$148°$	30.1
`905166`	$24°$	v_{up}	$148°$	23.4
`1506491`	$-20°$	v_{up}	$148°$	23.4
`1507078`	$24°$	v_{up}	$148°$	23.4

As with Run 16, we see major behaviours, B1 and B2, depending on whether the Xpuck is directly facing the frisbee or not (line 5). If it is, line 6 is triggered, otherwise, lines 8 and 9 are. The structure is slightly different to that of Run 16, where the individual elements of the v_{goal} vector are written. Here, there are two separate and complete writes to v_{goal}, depending on behaviour.

When the frisbee is directly in front of the robot (B1), the robot will move forward, but with some tendency to turn towards the $+x$ end of the arena. Behaviour B2 has several sub-behaviours, depending on whether there is any blue visible, and if not, whether and how many neighbours it has. If there is blue visible, the robot will move towards it. This combined with the B1 behaviour will tend to push the frisbee towards $-x$

With B2 and no visible frisbee, the robot will move to the $+x$ end of the arena if it has no neighbours. If there is a single neighbour, the robot will turn until its angle is $180° - c$, except for `308106` which will continue to turn. With more neighbours it will continue to turn. From a global perspective, this should result in a tendency for the swarm members that cannot see anything blue to congregate towards the $+x$ end of the arena and rotate to search for blue.

We can see these behaviours illustrated in Figure 8.17, which visualises a robot running tree `905166`. At the right of the figures are one or two robots that are stationary[7]. As the active robot approaches $+x$, it is executing B2 until it comes within range-and-bearing range of the stationary robots, that is 0.5 m. With a single neighbour, shown in the top figure, the robot just turns slightly and stops. With two neighbours, it continues turning until the blue frisbee becomes visible, at which point a blend of B1 and B2 behaviours causes the robot to reach and then actively push the frisbee in the $-x$ direction.

The seventh tree, `908257` has a slightly different structure. This is shown in Listing 8.6.

[7]Executing a null behaviour tree

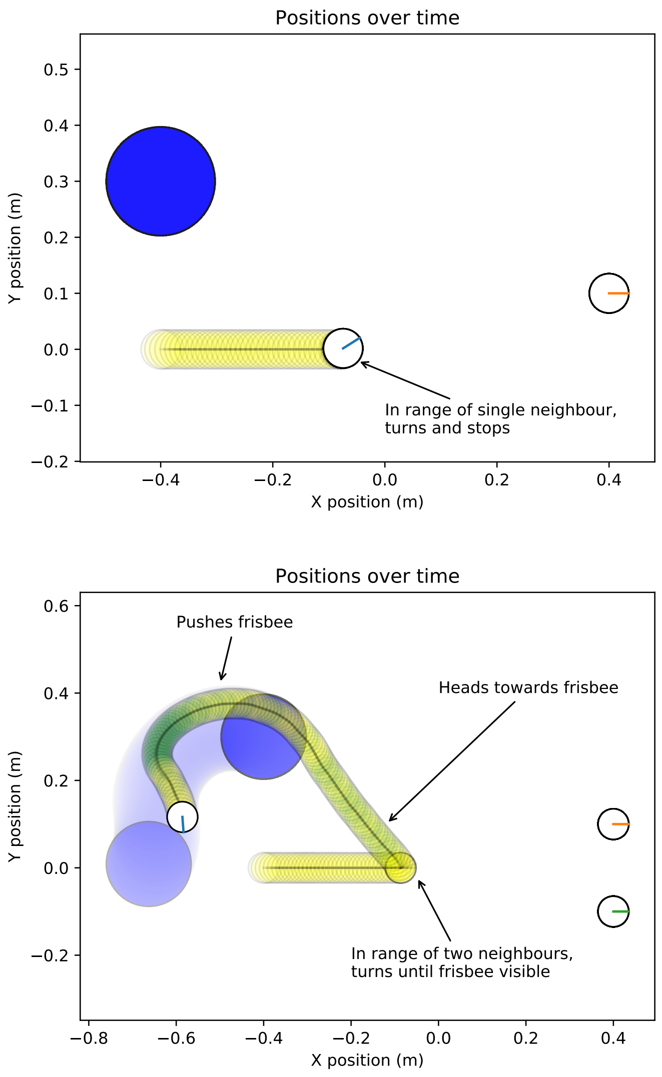


Figure 8.17: Visualisation of tree `905166` in the presence of one and two neighbours

Listing 8.6: Tree `908257`

```
sel
    avoiding
    failured
        sel
            seq
                bfront
                movcv $v_{scr}$  $-169°$
                rotav $v_{goal}$  $v_{blue}$  $24°$  $v_{up}$
            seq
                mulas $s_{scr}$  $v_{blue.y}$  $-23.1$  $v_{up.y}$
                mulav $v_{goal}$  $v_{scr}$  $23.4$  $v_{blue}$
    seqm
        rotav $v_{scr}$  $(s_n, s_{scr})$  $11°$  $v_{prox}$
        rotav $v_{goal}$  $v_{prox}$  $-56°$  $v_{prox}$
```

This again behaves in a similar way to the trees described above, with the presence of blue directly ahead controlling execution of lines 7 and 8 if yes, and lines 10 and 11 if not.

Once only execution What is interesting is the `seqm` construct at line 12. Due to the `failured` decorator at line 3, whatever the results of the check for blue, it will always be executed. Due to both `seq` clauses at lines 5 and 9 ending in a write to v_{goal}, any second write to v_{goal} in an update cycle will return *running*. The first ever update cycle will result in the node at line 13 executing, followed by that at line 14, which returns *running*. Since we are in a memoried sequence, and v_{goal} has always previously been written, all subsequent updates cycles resume at line 14, meaning that line 13 is guaranteed to only ever execute once.

8.6.7 Engineering higher performance

Given the ability we now have to deconstruct evolved behaviour trees and understand how they work, can we use this knowledge to engineer higher performance? We observed in Section 8.6.3 that a single robot using tree `806768` was unable to reliably push the frisbee, while more than one robot could. We also saw that tree `906737` was better at stably pushing the frisbee with a single robot, motivating the possibility of improving the single robot performance of `806768`. The tree is shown in Listing 8.7 with particular parameters denoted a, b, c, and d. We decided to try and hand tune the parameters to optimise single robot pushing stability and overall fitness.

Listing 8.7: Tree `806768`

```
sel
    avoiding
    sel
        seq
            bfront
            mulas      $v_{goal.y}$   $v_{prox.x}$   $a$   $v_{up.y}$
            mulas      $v_{goal.x}$   $v_{blue.y}$   $b$   $v_{up.x}$
        seq
            movcv      $v_{scr}$      $c$
            mulav      $v_{goal}$     $v_{scr}$      $d$   $v_{blue}$
```

Table 8.11: Hand tuned parameters for tree `806768`

	Original	Optimised
a	6.55	2
b	-20.7	-48
c	39°	70.3°
d	12.3	48
Fitness $\bar{x}$	0.27	0.30
Fitness *sigma*	0.044	0.041

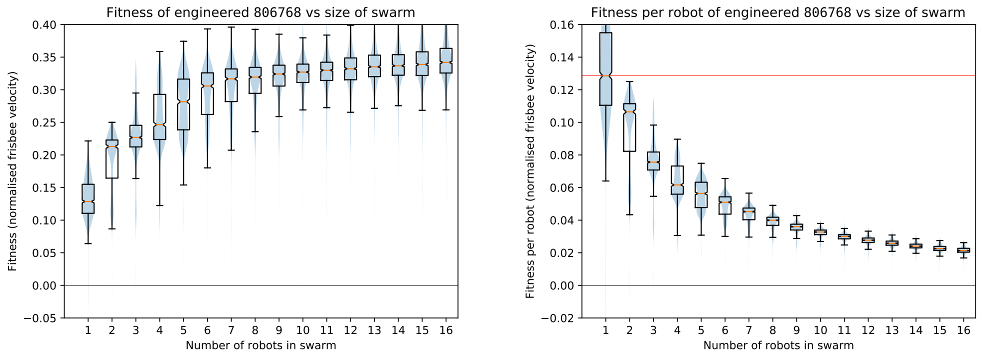


Figure 8.18: Scalability of swarm performance with increasing swarm size. Each point measured with 60 s simulated time over 1000 simulations with different starting conditions.

We can see from the data in Table 8.11 that we have achieved a useful performance improvement of about 10% from what was already a quite fit controller. But this has been achieved at the cost of superlinear scaling in performance as the swarm size increases. Figure 8.18 shows that the single agent performance has increased considerably from the unaltered tree (from 0.039 to 0.12) as we intended when optimising for more stable single robot pushing behaviour.

8.7 Explaining the difference between simulated and real fitness

Some runs produced fit behaviours in reality and some did not, despite having evolved fit controllers within simulation in the distributed evolutionary algorithm. This could be the result of several causes; firstly, it could just be a sampling problem. Each evolutionary run is unique, rather than multiple trials in reality of a particular swarm controller or controllers, we only have a single trial. Whereas multiple trials of the controller has been carried out in simulation, variability will sometimes result in large differences in fitness when transferred. There could also be certain controllers that are more susceptible to reality gap effects. One intriguing possibility though, is that we are seeing the effects of heterogeneity in the swarm.

8.7.1 Effect of sampling

To investigate whether sampling error could explain the differences, we have to make some assumptions. We carried out an independent two sample T-test for each run, where one sample was the simulated heterogeneous controller mix of a run, with 1000 observations, and the second sample was the real measured fitness of that controller mix. This gives a second sample size of one, which means we cannot say anything about the variance. We make the assumption that the variance is the same as the first sample, justified by reference to the results in Table 7.8 showing virtually identical

σ when evaluating the same controller in simulation and reality. The T-test also assumes that the samples are normally distributed, although this is less important for large samples. The simulated data fails the Shapiro-Wilk normality test, so these results should thus be treated with caution.

Table 8.12: Independent two sample T-test on each run between simulated heterogeneous fitness over 100 simulations and the real measured fitness. Only in runs 26 and 28 (bolded) does the real fitness differ significantly from the simulated fitness ($p < 0.05$).

Run	Sim $\bar{x}$	Sim σ	f_{real}	T-test p
1	0.171	0.084	0.174	0.977
2	0.118	0.077	0.028	0.243
3	0.184	0.063	0.116	0.276
5	0.162	0.083	0.085	0.355
6	0.148	0.090	0.097	0.568
7	0.186	0.099	0.240	0.582
8	0.202	0.098	0.204	0.983
9	0.158	0.078	0.171	0.874
14	0.102	0.075	0.038	0.399
15	0.162	0.086	0.005	0.067
16	0.258	0.056	0.200	0.306
17	0.174	0.070	0.154	0.778
18	0.060	0.130	0.199	0.284
20	0.113	0.081	0.108	0.950
21	0.075	0.125	-0.069	0.252
23	0.077	0.129	-0.082	0.217
24	0.084	0.128	0.160	0.555
25	0.114	0.068	0.124	0.882
26	0.071	0.127	-0.197	**0.035**
28	0.134	0.068	-0.061	**0.004**

The results are shown in Table 8.12. With the caution above in mind, we can see that, except for runs 26 and 28, so in 90% of the runs, we cannot reject the hypothesis ($p < 0.05$) that the samples are the same. In most cases, therefore, there is no difference between simulated fitness and performance in reality that needs to be explained.

8.7.2 Effect of controller heterogeneity in real swarm

We have seen above in the two runs that we analysed in detail that many trees that appeared different at first were actually functionally similar. In Run 16, we had two different trees, simulating a heterogeneous mix of the two at different ratios resulted in a smooth variation of performance of the swarm between the two homogeneous performances. But is that always the case? We can imagine scenarios where two or more styles of solution could conflict, resulting in worse performance than either type alone, or even the opposite, with synergistic interactions resulting in better performance.

From the statistical argument above, showing that the differences between simulation and real results are explainable from sampling error, we hypothesise that we should see little difference due to heterogeneity. To test this, we take each heterogeneous mixture of controllers that were present in the last segment of each run and simulate them to measure the mixture fitness. We then measure the fitness of each controller of the mixture individually in homogeneous simulation and take the average over the controllers. For each data point, we use the same simulation budget; 1000 simulations for each heterogeneous mix and 111 simulations for each of the nine trees present in the mix. Figure 8.19 shows the results.

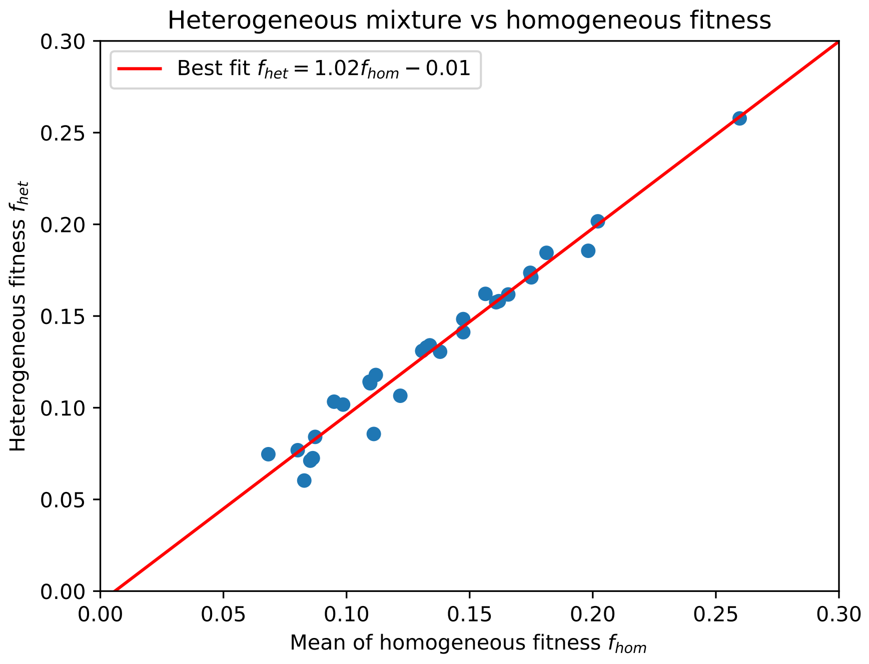


Figure 8.19: Effect of heterogeneity within runs on performance. Each point is a run. We measure the homogeneous performance of each controller within a run and take the average, and plot this against the heterogeneous performance of that set of controllers. There is little effect from heterogeneity.

The correspondence is remarkably high and there appears to be virtually no performance loss due to the heterogeneity of the controllers in a given run. This implies that, like the two runs examined in detail above, the trees of a given run are all quite similar in actual behaviour. This confirms the statistical argument of Section 8.7.1. We should expect if the island model evolutionary algorithm has a sufficient degree of migration to ensure good mixing. The actual migration level is $r_{migration} = 0.031$ (Eqn 8.3.1), so this raises the question of how low we can take the migration level before we start to see performance loss due to heterogeneity.

8.7.3 Effect of unrelated controller heterogeneity

In the previous section, we show that the heterogeneous mix of controllers in the real swarm does not cause a performance degradation compared to a homogenous swarm. We attribute this to the island model having a high enough migration rate to ensure good mixing. But we can test the effect of a zero rate by running simulations of heterogeneous swarms composed of controllers randomly from different runs. Although the fitness function in each case is the same, there will be no common lineage.

We performed 100 tests where the nine controllers for the heterogeneous swarm were selected randomly with replacement from the set of controllers present in the final segment of all runs. Each test was simulated 1000 times. We plotted the average of the individual trees homogenous fitness, as measured in the previous section, against the heterogeneous fitness. Because selection was random with replacement, with no knowledge of duplicates, some tests will contain multiple trees from the same run, but most will have trees from a mixture of runs.

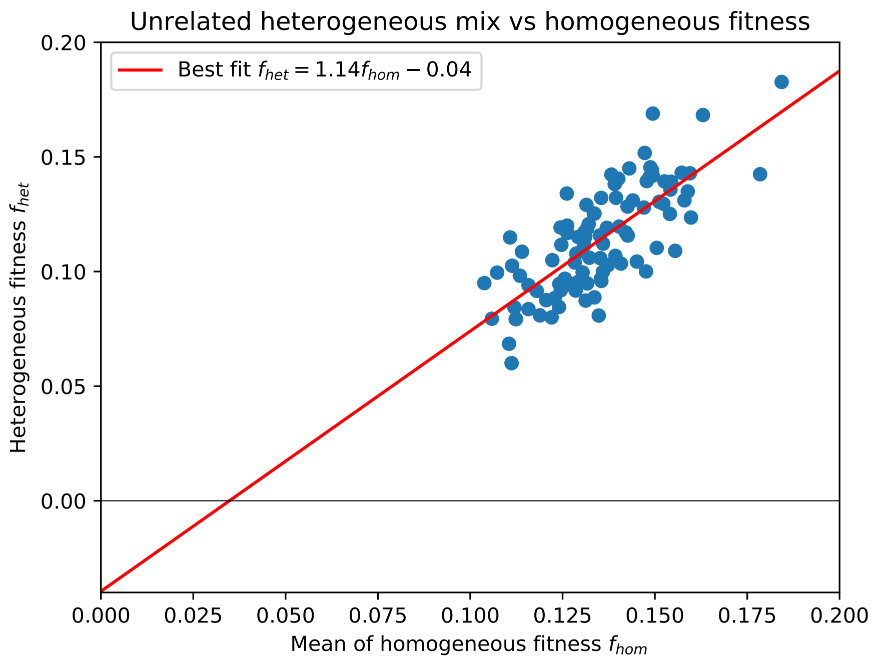


Figure 8.20: Effect of heterogeneity with unrelated trees on performance. We randomly select trees from across all runs and measure the heterogeneous fitness and plot this agains the mean homogeneous fitness. There is a cost to heterogeneity since the fitness reaches zero when the homogeneous fitness is still positive.

The results are shown in Figure 8.20. We can now see that there is a cost to heterogeneity, since the best fit line intersects the x-axis at a positive point. What is interesting is that the cost is relatively small - we are taking completely unrelated trees and mixing them in a swarm, and it is still performing reasonably.

8.8 Conclusions

In this chapter, we have demonstrated a swarm that is capable of evolving new controllers within the swarm itself, removing the tie to offline processing power. The in-swarm computational power is able to run an island model evolutionary algorithm that can produce fit and effective swarm controllers within 15 realtime minutes, far faster than has been possible previously. This is due to careful attention to several elements; the writing of a fast simulator that makes maximal use of the GPU processing power available, tuning the simulator parameters and controller architecture to minimise and mitigate reality gap effects, using the available simulator budget more effectively by improving the evolutionary algorithm, and finally using the island model to scale the evolutionary performance with the size of the swarm.

One overarching theme of this work has been the desirable properties of behaviour trees as a controller architecture, particularly as the target of evolutionary algorithm. The modularity, human understandability, and natural extendability should mean that we can analyse and understand evolved controllers for insight. In this chapter, we demonstrate this by using automatic methods to simplify evolved trees, then further human analysis to describe in detail how a selection of trees actually function. The understandability, or explainability, is an important characteristic for future systems created by machine learning in order that they can be verified to be safe.

Due to the decentralised nature of the island model, the real swarm is running an heterogeneous mixture of controllers. We explore the effect that this has on performance, and find that, in this case, the effect is low. There are further interesting questions to answer about how much we could reduce the current migration rate before diversity of behaviour produced much degraded performance, given how relatively well unrelated heterogeneous mixtures perform. Indeed, it may be the case that the objective function, the design of the robots, and the design of the behaviour tree architecture is constraining the solution space such that we have convergent behaviours.

Chapter 9

Conclusions

9.1 Overview

In this work, we have been concerned with moving swarm robotics as a discipline out of the laboratory and into the real world. There is tremendous potential for robot swarms in the wild to solve such problems as disaster recovery, mapping, pollution control, and exploration but there are so far are no real world examples. We identified two impediments to this.

Firstly, the limited computational power of existing swarms means that the automatic discovery of swarm controllers that produce a desired emergent collective behaviour has to take place outside the swarm. This means that the swarm is reliant on outside infrastructure, such as computers and communications links, making swarms difficult to deploy where these are not easily available, for example, in space, or remote areas.

Secondly, the usual controller architectures are opaque and hard to understand. In order to deploy swarm robots in the real world, we have to confront issues of understandability, explainability and safety. Without the ability to analyse the automatically discovered controllers, we cannot be confident on how a swarm will behave in a given situation, and cannot give guarantees of safety.

9.1.1 Behaviour trees

In Chapter 2 we looked at commonly used controller architectures for swarm robotics. We saw that there are difficulties with the commonly used approaches. Neural networks are fundamentally black boxes, only understandable by actually running them. Finite state machines suffer from scalability issues, as the number of nodes increases, the number of connections increases exponentially, making them either limited or hard to understand. The desire to tackle the disadvantages of both of these architectures lead us to consider behaviour trees. Because they are modular and hierarchical, they can be easily extended without exponential explosion of connections; a tree can encapsulate a complete useful sub-behaviour, that can be reused as a component within a larger tree. This hierarchical character also makes them amenable to analysis by decomposition; we can descend the tree hierarchy until the size of subtree is understandable. It could be argued that a very large tree would compromise this understandability, and it is true that the task would be harder. But the task is harder still for a neural network or state machine of similar complexity, e.g number of nodes or neurons. The ability to subdivide the problem, provided by the hierarchical structure, is unique among these three architectures and suggests that a behaviour tree will always be easier to analyse than either a neural network or state machine of comparable complexity.

We examined the theory of behaviour trees and described in detail their semantics. Various works have slightly differing terminologies and ambiguities, we have specified the exact behaviour and given a complete algorithm for their interpretation. We described various identities which can be used to transform trees in order to automatically simplify them. Behaviour trees have two components; the inner nodes, common across different works and applications, and the leaf nodes of a behaviour tree that interact with the environment as abstracted in the blackboard. We have shown that behaviour trees and finite state machines are equivalent and can be mechanically transformed from one to the other. We have also shown that behaviour trees can be Turing complete, by implementing the RASP Universal Turing Machine. The common subsumption robot control architecture was shown to be easily implementable using behaviour trees.

Since we used behaviour trees as the controller architecture, we designed a simple experiment to demonstrate the feasibility of using artificial evolution to discover fit controllers. We used a swarm of kilobots and an off-line evolutionary process to generate controllers for a simple foraging problem. The tree structure makes behaviour trees more amenable to the techniques of Genetic Programming [Koza, 1992], we discussed the various crossover and mutation operators required. The resultant evolved controllers transferred well from simulation to real robots, and were possible to analyse and understand. This showed that the evolvable behaviour tree approach was viable, and was published in Jones *et al.* [2016].

Finally, we looked in detail at the design of a behaviour tree architecture suitable for the Xpuck robots that constitute our swarm. We abstracted the sensor and actuator capabilities of the robot into a robot reference model, that defines both the requirements any robot simulator must fulfil, and the interface that was used in the design of the behaviour tree leaf nodes and blackboard. By making this as simple as possible, but no simpler, we eased the design of a fast simulator. By considering useful swarm behaviours that should be simple to describe in the behaviour tree architecture, we motivated the design of the leaf nodes and the blackboard entries.

9.1.2 Xpuck design

In Chapter 5 we covered the design of the robots that make up our custom-built swarm. The two primary motivators for the design were cost and processing performance. It is obviously important that cost per robot is minimised when building a swarm, where larger numbers are better. We based our robot on the e-puck, of which we had many available to the Bristol Robotics Laboratory. On top of that, we added an additional Single Board Computer that gave us the increased processing power we needed. Surveying the available platforms, we chose the Odroid XU4 for the very high computational power available from its Samsung Exynos 5422 SoC due to the on-chip GPU and the ability to use OpenCL for GPGPU, and for its low cost. The requirements for battery life of several hours, and full use of the e-puck VGA camera constrained the system design. We detailed that design, the trade-offs required and the process of tuning the operating points of the CPU and GPU that gave the best power and performance points.

The ability of the Xpuck robot to make good use of the e-puck VGA camera was demonstrated by implementing the ArUco tag image tracking library and testing the tracking accuracy in multiple scenarios with high success. The original e-puck processor, and even the Linux Extension Board were not capable of running this image processing application, showing the need for the extra processing power of our design.

Because evolutionary algorithms are dependent on large numbers of simulations, we looked next at the construction of a fast simulator. By using GPGPU techniques and carefully constraining the design to only the minimal required capability, we built a fast parallel 2D physics simulator capable of an aggregate robot simulation speed more than 50000 times faster than reality. We combined this with a behaviour tree interpreter and an evolutionary algorithm to demonstrate the processing performance of the swarm. By distributing the evolutionary algorithm over the swarm using the island model, we showed scalable and fast in-swarm evolution.

Some of this chapter was published in Jones *et al.* [2015].

9.1.3 Controller transferability

One important problem that affects robot controllers that have been discovered automatically in simulation is how well they transfer to reality. Known as the reality gap, or the sim-to-real problem, this is a consequence of the lack of fidelity of the simulated reality compared to the actual. Because evolutionary approaches will exploit any feature of the environment in order to effect fitness improvements, the controllers can become overfitted to the simulated environment. Chapter 7 introduced the benchmark task we used for the rest of this work and discussed the approaches we used to tackle this issue.

Firstly, we injected noise into the simulation to mask its deficiencies [Jakobi *et al.* , 1995]. The real robot motion noise was measured and much greater levels are injected into the simulation. Secondly, we carefully measured physical properties of the real robots and frisbee within the arena, such as the mass and coefficients of friction. This only took us so far though, due to the simulator being 2D and objects composed of perfect circles and lines, we could never truly reflect the complexities of actual collisions. We staged multiple collisions between the robots and the frisbee, recording the trajectories of each, then constructed identical scenarios within the simulator and optimised various simulator parameters to minimise the differences in the object trajectories.

The third aspect of the strategy was encapsulated in the robot reference model. Here we abstracted the sensor capabilities of the robot, but we did so in such a way as to both simplify the simulator modelling and minimise the difference between simulated senses and conditioned senses in the real robot. For example, the camera sensor in the reference model was abstracted as simply the presence of blue in the left, centre, or right third of the field of vision. An image with 300k pixels was represented as 3 bits. In the real robot, we classified the colour of pixels and count them. In the simulation model, we performed a low resolution ray-tracing operation to identify visible objects. We measured the actual sensor data after this abstraction for each sense in both real robots and in simulated robots in identical scenarios, adjusting the simulator to reduce the difference.

Finally, some aspects of reality are particularly difficult to model well in a 2D simulation, principally collisions between robots, and between robots and arena walls. The simulator has robots and walls as ideal surfaces, the real robots are rough, with protrusions and imperfections. The point of collision between two robots is above ground level, resulting in moments that reduce the wheel friction in non-linear and unmodelled ways. Our solution was to try and avoid these situations at the behavioural level, with collision avoidance as a default base level behaviour, taking priority over the evolved controller. The hierarchical structure of behaviour trees made this example of subsumption architecture trivial to implement.

We demonstrated the effectiveness of our mitigation strategies by evolving a behaviour tree controller for the benchmark task and measuring its fitness many times in simulation, and 19 times in reality. There was no difference between simulation and reality.

9.1.4 In-swarm evolution

In Chapter 8 we brought all the different components of the work together, running the evolution of new controllers entirely within the swarm in a distributed and autonomous way. As fitter controllers were generated in simulation, they were periodically instantiated to run the real robots of the swarm, resulting in a real swarm that after only 15 minutes was good at the task of pushing the frisbee.

Firstly we modified the evolutionary algorithm used in Chapter 7 to make better use of the available simulation budget in the context of a noisy fitness function. Rather than performing a fixed number of simulations per individual, we used a much larger population with accumulating fitness evaluations and a tournament selector that has a bias towards high fitness with low variance. This reached a higher fitness with the same simulation budget. We then extended this to an island model distributed evolutionary algorithm by having each robot evolve its own population and allowing a small amount of migration of the fittest individuals between the subpopulations. This showed good scaling in performance, compared to a single node.

Given this distributed in-swarm evolutionary system, we performed a set of 16 minute runs where, every two minutes, each robot instantiated its local fittest controller to run itself in the real world. The controllers of the swarm thus followed the discretised trajectory of the evolutionary system. In many cases, after only 10 minutes, the real swarm was performing the benchmark task with high fitness.

A central theme of this work has been the desirability of behaviour trees as a controller architecture, because automatically generated trees can be analysed and explained. We took the many evolutionary runs performed and cluster them by behavioural metrics in order to highlight interesting controllers to analyse. We looked at several different controllers, automatically simplifying them and then applying further simplifications by hand. The behaviour was explained in detail with visualisations of the different sub-behaviours. We characterised the swarm scaling performance. Using the insight gained, we engineered higher performance in one controller.

Finally, we looked at the difference in performance between the swarm in reality and in simulation, examining the effects of a heterogeneous but related mixture of controllers, a completely heterogeneous mixture of controllers, and the effect of sampling. We concluded that most of the difference in our set of runs from simulation was due to sampling, and there was no significant controller transference performance loss.

9.2 Conclusions and future work

We have successfully built a computationally powerful swarm of robots and demonstrated the autonomous in-swarm evolution of fit behaviour tree controllers in short periods of time. The controllers could be, and some were, analysed, explained and even improved. These are necessary conditions to move swarm robots into the real world.

Clearly, though, there is further work that can be done. The following sections sketch some of the areas we are particularly interested in pursuing.

9.2.1 Adaptivity

Firstly, and perhaps most importantly, the system as described was not adaptive to changes in the environment. By adaptivity, we mean the quality of the system being able to adapt its behaviour automatically in response to change, rather than adaptability, the quality of the system being easily adapted by some external actor to suit a change in the environment [Oppermann & Rasher, 1997; Reinecke *et al.* , 2010]. Controller solutions were evolved entirely in simulation on the robots, then transferred to run them in reality. The flow of control was one-way only. We have sketched out a tentative way forward; consider a changeable element E of the environment that can affect the fitness of the swarm. To be adaptive, the swarm has to a) detect the current state of E, and b) has to evolve new controllers within a simulated environment incorporating the new value of E. One approach is the co-evolution of simulator parameters and controllers, used by O'Dowd *et al.* [2014], with real-world fitness used as a proxy for simulator alignment with the environment. Another is the more direct measurement of the environment. We assume that after sufficient running time in an unchanged environment, the distributed evolutionary system reaches a plateau of performance. If we are at this quasi steady-state and a change in E is detected in the real world, we alter the simulated environment similarly. This should cause a drop in the fitness within the evolutionary system, followed by a recovery as new innovations are produced in response to the perturbation.

These new adapted controllers are eventually instantiated to run the real robots, making the swarm adaptive to the changed environment. The detection of the state of some element E of the environment could be called *reality probing*. The are many possible ways this could be done. For example, suppose the change was the mass of frisbee. A heavy frisbee is harder to push, and the driving wheels slip more. The difference between the commanded velocity vs the actual velocity when pushing the frisbee is one form for reality probe.

In accordance with swarm principles, this reality probe information is local, but we want the entire swarm evolutionary system to have access to the measured value of

E. Here we could use some form of distributed decision making, as used by bees, for example, when choosing a nest site, to arrive at a consensus value for E based on local knowledge and agent interactions [Crosscombe *et al.* , 2017]. At this point, we have detected an environmental change and altered the simulation environment used by the evolutionary algorithm in an entirely distributed way.

9.2.2 Reality gap and the effect of architecture

The central thesis of Francesca *et al.* [2014a] is that the high representational power of neural networks makes them prone to overfitting the simulation environment of an automatic discovery process and thus such controllers perform poorly when transferred to the real world. By reducing the representational power, by limiting available behaviours to a larger granularity, the reality gap could be reduced. Their work was one inspiration for our use of behaviour trees, and their modular and hierarchical structure make it convenient to test this thesis. We can define useful sub-behaviours that perform similarly both in simulation and reality and encapsulate them. If the evolutionary algorithm is only allowed to use these robust sub-behaviours as leaf nodes in evolved controllers, we hypothesise that such controllers will be more robust themselves to reality gap effects than if the evolutionary algorithm is allowed to use the full range of much lower level leaf nodes. Of course, by limiting the range of behaviours available to the evolutionary algorithm, we may also limit the ultimate level of performance we can reach, but the shape of this trade-off is not known and merits further investigation.

On a related note, we have discussed in detail the design of the behaviour tree architecture. What impact does this have on evolved solutions? One choice we made which in retrospect might be worth revisiting is the form of the blackboard vectors. They are currently $(\boldsymbol{x}, \boldsymbol{y})$ pairs of real numbers. What if they were represented in polar form $(\boldsymbol{r}, \boldsymbol{\theta})$? Because of other decisions, the individual elements of the vector are accessible to scalar leaf nodes, indeed this capability is exploited in one of the trees we examine in Chapter 8. Would the polar form produce easier to understand evolved controllers? Or different behaviour styles? It should always be possible to represent a controller that performs identically with either vector style, but the search space is differently shaped, so we could expect different results.

9.2.3 Moving into three dimensions

This work was focussed entirely on robots operating on a 2D surface. The reasons for this were the previous experience of the author in writing fast 2D physics simulators, and the availability within the lab of a large supply of e-puck robots that could be enhanced at relatively low cost in order to build the Xpuck swarm. There is, however, no reason why the methodology used here could not be applied to a swarm of robots working in three dimensions, flying drones, for example.

The current system is capable of evolving fit solutions within 15 minutes. This is within the flying time of current drones. The XU4 single board computer is small and light, weighing only 60 g, and drones have large capacity batteries to power the propeller motors, so the power requirements should be easy to meet. It seems quite feasible to equip a swarm of drones with the same computational capacity as the Xpuck swarm. Given such a swarm, the question then is how to go about designing the robot reference model, and, given that abstraction, the associated blackboard and suitable BT leaf nodes. In order to run the evolutionary algorithm, it would also be necessary to write an appropriately fast simulator. We argue that it is important to construct the reference model in a minimalistic way; abstracting away as much detail as possible without compromising the ability to perform the required tasks. This simplification makes possible a sufficiently performant simulator and motivates granular behaviours, potentially helping reduce the reality gap. It would be very interesting to test the methodology of this work in such a way.

Appendix A

Additional Material

A.1 Xpuck Open Source

We make the hardware and software design of the Xpuck robot and associated simulator freely available with a permissive MIT licence:

All files necessary to build the Xpuck hardware and run OpenCL accelerated simulations are available at the repository `https://bitbucket.org/siteks/xpuck_open_` `source`.

A.2 Videos

Some videos of the experiments are available. Each run has a visualisation of the telemetry and Vicon data and some runs have actual videos. They are available at `https://www.youtube.com/user/simonj23/videos`.

References

Abiyev, Rahib H, Bektaş, ŞENOL, Akkaya, Nurullah, & Aytac, Ersin. 2013. Behaviour Trees Based Decision Making for Soccer Robots. *Page 102 of:* Kanarachos, Andreas (ed), *Recent Advances in Mathematical Methods, Intelligent Systems and Materials: (mamectis '13)(materials '13).* Mathematics and computers in science and engineering. Wseas LLC.

Bäck, Thomas. 1996. *Evolutionary algorithms in theory and practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* Oxford university press.

Bäck, Thomas, Fogel, David B, & Michalewicz, Zbigniew. 2000. *Evolutionary Computation 1. Basic Algorithms and Operators, chapter Permutations.*

Bagnell, J Andrew, Cavalcanti, Felipe, Cui, Lei, Galluzzo, Thomas, Hebert, Martial, Kazemi, Moslem, Klingensmith, Matthew, Libby, Jacqueline, Liu, Tian Yu, Pollard, Nancy, *et al.* . 2012. An integrated system for autonomous robotics manipulation. *Pages 2955–2962 of: 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems.* Vilamoura, Portugal: IEEE, for IEEE.

Baldassarre, Gianluca, Nolfi, Stefano, & Parisi, Domenico. 2003. Evolving mobile robots able to display collective behaviors. *Artificial life*, **9**(3), 255–267.

Bedau, Mark A. 1997. Weak emergence. *Noûs*, **31**, 375–399.

Benzie, John AH, & Williams, Suzanne T. 1997. Genetic structure of giant clam (Tridacna maxima) populations in the west Pacific is not consistent with dispersal by present-day ocean currents. *Evolution*, **51**(3), 768–783.

Beyer, Hans-Georg, & Schwefel, Hans-Paul. 2002. Evolution Strategies - A comprehensive introduction. *Natural computing*, **1**(1), 3–52.

Birattari, Mauro, Stützle, Thomas, Paquete, Luis, Varrentrapp, Klaus, *et al.* . 2002. A Racing Algorithm for Configuring Metaheuristics. *In: Gecco*, vol. 2.

Birattari, Mauro, Delhaisse, Brian, Francesca, Gianpiero, & Kerdoncuff, Yvon. 2016. Observing the effects of overdesign in the automatic design of control software for robot swarms. *Pages 149–160 of:* Dorigo, Marco, Birattari, Mauro, Li, Xiaodong, López-Ibáñez, Manuel, Ohkura, Kazuhiro, Pinciroli, Carlo, & Stützle, Thomas (eds), *International Conference on Swarm Intelligence (ANTS 2016).* Brussels, Belgium: Springer.

Birattari, Mauro, Ligot, Antoine, Bozhinoski, Darko, Brambilla, Manuele, Francesca, Gianpiero, Garattoni, Lorenzo, Garzón Ramos, David, Hasselmann, Ken, Kegeleirs, Miquel, Kuckling, Jonas, Pagnozzi, Federico, Roli, Andrea, Salman, Muhammad, & Stützle, Thomas. 2019. Automatic Off-Line Design of Robot Swarms: A Manifesto. *Frontiers in Robotics and AI*, **6**, 59.

Bjerknes, Jan Dyre, & Winfield, Alan FT. 2013. On fault tolerance and scalability of swarm robotic systems. *Pages 431–444 of:* Martinoli, A., Mondada, F., Correll, N., Mermoud, G., Egerstedt, M., Hsieh, M.A., Parker, L.E., & Støy, K (eds), *Distributed Autonomous Robotic Systems. The 10th International Symposium (DARS 2010).* Springer.

Blum, Christian, Winfield, Alan FT, & Hafner, Verena V. 2018. Simulation-Based Internal Models for Safer Robots. *Frontiers in Robotics and AI*, **4**, 74.

Bohr, Mark. 2007. A 30 year retrospective on Dennard's MOSFET scaling paper. *IEEE Solid-State Circuits Society Newsletter*, **12**(1), 11–13.

Bongard, Josh, Zykov, Victor, & Lipson, Hod. 2006. Resilient machines through continuous self-modeling. *Science*, **314**(5802), 1118–1121.

Bongard, Josh C. 2013. Evolutionary robotics. *Communications of the ACM*, **56**(8), 74–83.

Brambilla, Manuele, Ferrante, Eliseo, Birattari, Mauro, & Dorigo, Marco. 2013. Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, **7**(1), 1–41.

Bredeche, Nicolas, Montanier, Jean-Marc, Liu, Wenguo, & Winfield, Alan FT. 2012. Environment-driven distributed evolutionary adaptation in a population of autonomous robotic agents. *Mathematical and Computer Modelling of Dynamical Systems*, **18**(1), 101–129.

Bredeche, Nicolas, Haasdijk, Evert, & Prieto, Abraham. 2018. Embodied Evolution in Collective Robotics: A Review. *Frontiers in Robotics and AI*, **5**, 12.

Brooks, Rodney. 1986. A robust layered control system for a mobile robot. *IEEE journal on robotics and automation*, **2**(1), 14–23.

Brooks, Rodney A. 1991. Intelligence without representation. *Artificial intelligence*, **47**(1-3), 139–159.

Cantú-Paz, Erick. 1998. A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systems repartis*, **10**(2), 141–171.

Cao, Y Uny, Fukunaga, Alex S, Kahng, Andrew B, & Meng, Frank. 1995. Cooperative mobile robotics: Antecedents and directions. *Pages 226–234 of: Intelligent Robots and Systems 95.'Human Robot Interaction and Cooperative Robots', Proceedings. 1995 IEEE/RSJ International Conference on*, vol. 1. IEEE.

Catto, Erin. 2009. Box2D: A 2D Physics Engine for Games. *World Wide Web electronic publication, http://box2d.org/about/.*

Champandard, Alex. 2007. Behavior trees for next-gen game AI. *In: Game developers conference, audio lecture.*

Clancy, Daniel J, & Kuipers, BJ. 1993. Behavior abstraction for tractable simulation. *Pages 57–64 of: Proceedings of the Seventh International Workshop on Qualitative Reasoning about Physical Systems.* Citeseer.

Clune, Jeff, Mouret, Jean-Baptiste, & Lipson, Hod. 2013. The evolutionary origins of modularity. *Proceedings of the Royal Society of London B: Biological Sciences*, **280**(1755), 20122863.

Colledanchise, Michele. 2017. *Behavior Trees in Robotics.*

Colledanchise, Michele, & Ogren, Petter. 2014. How Behavior Trees modularize robustness and safety in hybrid systems. *Pages 1482–1488 of:* Burgard, Wolfram (ed), *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2014).* IEEE, Chicago, Illinois, USA.

Colledanchise, Michele, Parasuraman, Ramviyas, & Ögren, Petter. 2015. Learning of Behavior Trees for Autonomous Agents. *arXiv preprint arXiv:1504.05811.*

Cook, Stephen A, & Reckhow, Robert A. 1973. Time bounded random access machines. *Journal of Computer and System Sciences*, **7**(4), 354–375.

Crosscombe, Michael, Lawry, Jonathan, Hauert, Sabine, & Homer, Martin. 2017. Robust distributed decision-making in robot swarms: Exploiting a third truth state. *Pages 4326–4332 of:* Maciejewski, Tony (ed), *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* IEEE, Vancouver, Canada.

Cutumisu, Maria, & Szafron, Duane. 2009. An Architecture for Game Behavior AI: Behavior Multi-Queues. *In:* Darken, Christian J., & Youngblood, G. Michael (eds), *Fifth Artificial Intelligence for Interactive Digital Entertainment Conference (AIIDE 2009).* Palo Alto, California, USA: AAAI.

Davis, Lawrence. 1991. Handbook of genetic algorithms.

De Wolf, Tom, & Holvoet, Tom. 2004. Emergence versus self-organisation: Different concepts but promising when combined. *Pages 1–15 of: International workshop on engineering self-organising applications.* Springer.

Deneubourg, Jean-Louis, & Goss, Simon. 1989. Collective patterns and decision-making. *Ethology Ecology & Evolution*, **1**(4), 295–311.

Dennard, Robert H, Gaensslen, Fritz H, Rideout, V Leo, Bassous, Ernest, & LeBlanc, Andre R. 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, **9**(5), 256–268.

Dill, Kevin, & Lockheed Martin. 2011. A game AI approach to autonomous control of virtual characters. *In: Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC).*

Doncieux, Stephane, Bredeche, Nicolas, Mouret, Jean-Baptiste, & Eiben, Agoston E Gusz. 2015. Evolutionary robotics: what, why, and where to. *Frontiers in Robotics and AI*, **2**, 4.

Dorigo, Marco, Tuci, Elio, Groß, Roderich, Trianni, Vito, Labella, Thomas Halva, Nouyan, Shervin, Ampatzis, Christos, Deneubourg, Jean-Louis, Baldassarre, Gianluca, Nolfi, Stefano, *et al.* . 2004. The swarm-bots project. *Pages 31–44 of:* Şahin, Erol, & Spears, William M. (eds), *International Workshop on Swarm Robotics.* Springer, Santa Monica, CA, USA.

Dorigo, Marco, Floreano, Dario, Gambardella, Luca Maria, Mondada, Francesco, Nolfi, Stefano, Baaboura, Tarek, Birattari, Mauro, Bonani, Michael, Brambilla, Manuele, Brutschy, Arne, *et al.* . 2013. Swarmanoid: a novel concept for the study of heterogeneous robotic swarms. *IEEE Robotics & Automation Magazine*, **20**(4), 60–71.

Došilović, Filip Karlo, Brčić, Mario, & Hlupić, Nikica. 2018. Explainable artificial intelligence: A survey. *Pages 0210–0215 of: 2018 41st International convention on information and communication technology, electronics and microelectronics (MIPRO).* IEEE.

Dromey, R Geoff. 2003. From requirements to design: Formalizing the key steps. *Pages 2–11 of: First International Conference on Software Engineering and Formal Methods.* IEEE, Brisbane, Queensland, Australia.

Duarte, Miguel, Oliveira, Sancho Moura, & Christensen, Anders Lyhne. 2014. Hybrid control for large swarms of aquatic drones. *Pages 785–792 of: Proceedings of the 14th International Conference on the Synthesis & Simulation of Living Systems (ALIFE 2014).* New York, NY, USA: MIT Press.

Duarte, Miguel, Gomes, Jorge, Costa, Vasco, Oliveira, Sancho Moura, & Christensen, Anders Lyhne. 2016. Hybrid Control for a Real Swarm Robotics System in an Intruder Detection Task. *Pages 213–230 of:* Squillero, Giovanni, & Burelli, Paolo (eds), *Applications of Evolutionary Computation: 19th European Conference (EvoApplications 2016)*. Porto, Portugal: Springer, Cham.

Dyckhoff, Max, & Bungie LLC. 2008. Decision making and knowledge representation in Halo 3. *In: Presentation at the Game Developers Conference*. GDC.

Fajen, Brett R, & Warren, William H. 2003. Behavioral dynamics of steering, obstable avoidance, and route selection. *Journal of Experimental Psychology: Human Perception and Performance*, **29**(2), 343.

Fletcher, Desmond, & Goss, Ernie. 1993. Forecasting with neural networks: an application using bankruptcy data. *Information & Management*, **24**(3), 159–167.

Floreano, Dario, Mitri, Sara, Magnenat, Stéphane, & Keller, Laurent. 2007. Evolutionary conditions for the emergence of communication in robots. *Current biology*, **17**(6), 514–519.

Floreano, Dario, Dürr, Peter, & Mattiussi, Claudio. 2008. Neuroevolution: from architectures to learning. *Evolutionary intelligence*, **1**(1), 47–62.

Fortin, Félix-Antoine, Rainville, De, Gardner, Marc-André Gardner, Parizeau, Marc, Gagné, Christian, *et al.* . 2012. DEAP: Evolutionary algorithms made easy. *The Journal of Machine Learning Research*, **13**(1), 2171–2175.

Francesca, Gianpiero, & Birattari, Mauro. 2016. Automatic Design of Robot Swarms: Achievements and Challenges. *Frontiers in Robotics and AI*, **3**, 29.

Francesca, Gianpiero, Brambilla, Manuele, Brutschy, Arne, Trianni, Vito, & Birattari, Mauro. 2014a. AutoMoDe: A novel approach to the automatic design of control software for robot swarms. *Swarm Intelligence*, **8**(2), 89–112.

Francesca, Gianpiero, Brambilla, Manuele, Brutschy, Arne, Garattoni, Lorenzo, Miletitch, Roman, Podevijn, Gaëtan, Reina, Andreagiovanni, Soleymani, Touraj, Salvaro, Mattia, Pinciroli, Carlo, *et al.* . 2014b. An experiment in automatic design of robot swarms. *Pages 25–37 of: Swarm Intelligence*. Springer.

Francesca, Gianpiero, Brambilla, Manuele, Brutschy, Arne, Garattoni, Lorenzo, Miletitch, Roman, Podevijn, Gaëtan, Reina, Andreagiovanni, Soleymani, Touraj, Salvaro, Mattia, Pinciroli, Carlo, *et al.* . 2015. AutoMoDe-Chocolate: automatic design of control software for robot swarms. *Swarm Intelligence*, **9**(2-3), 125–152.

Fujita, Masahiro, Kuroki, Yoshihiro, Ishida, Tatsuzo, & Doi, Toshi T. 2003. Autonomous behavior control architecture of entertainment humanoid robot SDR-

4X. *Pages 960–967 of: Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)(Cat. No. 03CH37453)*, vol. 1. IEEE.

Garrido-Jurado, Sergio, Munoz-Salinas, Rafael, Madrid-Cuevas, Francisco J., & Marin-Jimenez, Manuel J. 2014. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, **47**(6), 2280 – 2292.

Gaul, Randy. 2012. Impulse Engine 2D physics simulator. *World Wide Web electronic publication.*

Gonzalez-Perez, Cesar, Henderson-Sellers, Brian, & Dromey, Geoff. 2005. A meta-model for the behavior trees modelling technique. *Pages 35–39 of: Third International Conference on Information Technology and Applications (ICITA'05)*, vol. 1. IEEE.

Gorges-Schleuter, Martina. 1990. Explicit parallelism of genetic algorithms through population structures. *Pages 150–159 of: International Conference on Parallel Problem Solving from Nature.* Springer.

Grasso, Ivan, Radojkovic, Petar, Rajovic, Nikola, Gelado, Isaac, & Ramirez, Alex. 2014. Energy Efficient HPC on Embedded SoCs: Optimization Techniques for Mali GPU. *Pages 123–132 of: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International.* IEEE, Phoenix, AZ, USA.

Gronqvist, Johan, & Lokhmotov, Anton. 2014. Optimising OpenCL kernels for the ARM Mali-T600 GPUs. *GPU Pro 5: Advanced Rendering Techniques*, 327–357.

Gurrum, Siva P, Edwards, Darvin R, Marchand-Golder, Thomas, Akiyama, Jotaro, Yokoya, Satoshi, Drouard, Jean-Francois, & Dahan, Franck. 2012. Generic thermal analysis for phone and tablet systems. *Pages 1488–1492 of: IEEE 62nd Electronic Components and Technology Conference (ECTC 2012).* IEEE, San Diego, USA.

Gutiérrez, Álvaro, Tuci, Elio, & Campo, Alexandre. 2009a. Evolution of neuro-controllers for robots' alignment using local communication. *International Journal of Advanced Robotic Systems*, **6**(1), 6.

Gutiérrez, Álvaro, Campo, Alexandre, Dorigo, Marco, Donate, Jesus, Monasterio-Huelin, Félix, & Magdalena, Luis. 2009b. Open e-puck range & bearing miniaturized board for local communication in swarm robotics. *Pages 3111–3116 of: Robotics and Automation, 2009. ICRA'09. IEEE International Conference on.* IEEE, Kobe, Japan.

Hamann, Heiko. 2012. Towards swarm calculus: Universal properties of swarm performance and collective decisions. *Pages 168–179 of:* Dorigo, Marco, Birattari, Mauro, Blum, Christian, Christensen, Anders Lyhne, Engelbrecht, Andries P., Groß, Roderich, & Stützle, Thomas (eds), *International Conference on Swarm Intelligence (ANTS 2012).* Springer, Brussels, Belgium.

Hauert, Sabine, Winkler, Laurent, Zufferey, Jean-Christophe, & Floreano, Dario. 2008. Ant-based swarming with positionless micro air vehicles for communication relay. *Swarm Intelligence*, **2**(2), 167–188.

Hauert, Sabine, Zufferey, Jean-Christophe, & Floreano, Dario. 2009. Evolved swarming without positioning information: an application in aerial communication relay. *Autonomous Robots*, **26**(1), 21–32.

Hauert, Sabine, Leven, Severin, Varga, Maja, Ruini, Fabio, Cangelosi, Angelo, Zufferey, J-C, & Floreano, Dario. 2011. Reynolds flocking in reality with fixed-wing robots: communication range vs. maximum turning rate. *Pages 5015–5020 of: Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on.* IEEE.

Haykin, Simon. 1994. *Neural networks: a comprehensive foundation.* Prentice Hall PTR.

Hintjens, Pieter. 2013. *ZeroMQ: Messaging for Many Applications.* O'Reilly Media, Inc.

Ho, Joshua, & Smith, Ryan. 2015. NVIDIA Tegra X1 Preview and Architecture Analysis. *AnandTech.*

Hoff, Nicholas, Wood, Robert, & Nagpal, Radhika. 2013. Distributed colony-level algorithm switching for robot swarm foraging. *Pages 417–430 of: Distributed Autonomous Robotic Systems.* Springer.

Holland, John Henry, *et al.* . 1975. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* MIT press.

Hoshino, Yukiko, Takagi, Tsuyoshi, Di Profio, Ugo, & Fujita, Masahiro. 2004. Behavior description and control using behavior module for personal robot. *Pages 4165–4171 of: IEEE International Conference on Robotics and Automation (ICRA'04)*, vol. 4. IEEE, New Orleans, LA, USA.

Huang, Wesley H, Fajen, Brett R, Fink, Jonathan R, & Warren, William H. 2006. Visual navigation and obstacle avoidance using a steering potential function. *Robotics and Autonomous Systems*, **54**(4), 288–299.

Hutchison, David C. 2005. Introducing BrilliantColor^{TM} Technology. *Texas Instruments white paper.*

Isla, Damian. 2005. Handling complexity in the Halo 2 AI. *In: Game Developers Conference (GDC 2005)*, vol. 12. GDC.

Jakobi, Nick. 1998. Running across the reality gap: Octopod locomotion evolved in a minimal simulation. *Pages 39–58 of: Evolutionary Robotics.* Springer.

Jakobi, Nick, Husbands, Phil, & Harvey, Inman. 1995. Noise and the reality gap: The use of simulation in evolutionary robotics. *Pages 704–720 of: Advances in artificial life.* Springer.

Jin, Yaochu. 2011. Surrogate-assisted evolutionary computation: Recent advances and future challenges. *Swarm and Evolutionary Computation*, **1**(2), 61–70.

Jin, Yaochu, & Branke, Jürgen. 2005. Evolutionary optimization in uncertain environments-a survey. *IEEE Transactions on evolutionary computation*, **9**(3), 303–317.

Jones, Simon, Studley, Matthew, & Winfield, Alan FT. 2015. Mobile GPGPU Acceleration of Embodied Robot Simulation. *In:* Headleand C., Teahan W., Ap Cenydd L. (ed), *Artificial Life and Intelligent Agents: First International Symposium (ALIA 2014).* Bangor, UK: Springer.

Jones, Simon, Studley, Matthew, Hauert, Sabine, & Winfield, Alan FT. 2016. Evolving behaviour trees for swarm robotics. *In:* Groß, Roderich, Kolling, Andreas, Berman, Spring, Frazzoli, Emilio, Martinoli, Alcherio, Matsuno, Fumitoshi, & Gauci, Melvin (eds), *13th International Symposium on Distributed Autonomous Robotic Systems (DARS 2016).* London, UK: Springer.

Jones, Simon, Studley, Matthew, Hauert, Sabine, & Winfield, Alan FT. 2018. A Two Teraflop Swarm. *Frontiers in Robotics and AI*, **5**, 11.

Jones, Simon, Winfield, Alan FT, Hauert, Sabine, & Studley, Matthew. 2019. Onboard Evolution of Understandable Swarm Behaviors. *Advanced Intelligent Systems*, **1**.

Keckler, Stephen W, Dally, William J, Khailany, Brucek, Garland, Michael, & Glasco, David. 2011. GPUs and the future of parallel computing. *IEEE Micro*, **31**(5), 7–17.

Khronos OpenCL Working Group, *et al.* . 2010. *The OpenCL Specification, Version 1.1.*

Klöckner, Andreas. 2013a. Behavior Trees for UAV Mission Management. *Pages 57–68 of: INFORMATIK 2013 Informatik angepasst an Mensch, Organisation und Umwelt.* Koblenz, Germany: Springer.

Klöckner, Andreas. 2013b. Interfacing behavior trees with the world using description logic. *In: AIAA conference on Guidance, Navigation and Control.* AIAA, Boston, MA, inUSA.

Kohonen, Teuvo. 1982. Self-organized formation of topologically correct feature maps. *Biological cybernetics*, **43**(1), 59–69.

Konfrst, Zdenek. 2004. Parallel genetic algorithms: Advances, computing trends, applications and perspectives. *Page 162 of: 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* IEEE.

Koos, Sylvain, Mouret, J-B, & Doncieux, Stéphane. 2013. The transferability approach: Crossing the reality gap in evolutionary robotics. *Evolutionary Computation, IEEE Transactions on*, **17**(1), 122–145.

Koza, John R. 1992. *Genetic programming: on the programming of computers by means of natural selection.* Vol. 1. MIT press.

Koza, John R. 1994. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, **4**(2), 87–112.

Kushleyev, Alex, Mellinger, Daniel, Powers, Caitlin, & Kumar, Vijay. 2013. Towards a swarm of agile micro quadrotors. *Autonomous Robots*, **35**(4), 287–300.

Langdon, William B. 2000. Size fair and homologous tree crossovers for tree genetic programming. *Genetic programming and evolvable machines*, **1**(1-2), 95–119.

Lehman, Joel, Clune, Jeff, & Misevic, Dusan. 2018a. The Surprising Creativity of Digital Evolution. *Pages 55–56 of:* Ikegami, Takashi, Virgo, Nathaniel, Witkowski, Olaf, Oka, Mizuki, Suzuki, Reiji, & Iizuka, Hiroyuki (eds), *The 2018 Conference on Artificial Life (ALIFE 2018).* MIT Press, Tokyo, Japan.

Lehman, Joel, Clune, Jeff, Misevic, Dusan, Adami, Christoph, Beaulieu, Julie, Bentley, Peter J, Bernard, Samuel, Belson, Guillaume, Bryson, David M, Cheney, Nick, *et al.* . 2018b. The surprising creativity of digital evolution: A collection of anecdotes from the evolutionary computation and artificial life research communities. *arXiv preprint arXiv:1803.03453.*

Lim, Chong-U, Baumgarten, Robin, & Colton, Simon. 2010. Evolving behaviour trees for the commercial game DEFCON. *Pages 100–110 of: Applications of evolutionary computation.* Springer.

Lipovski, G Jack. 1976. The architecture of a simple, effective control processor. *Microprocessing and Microprogramming.*

Liu, Wenguo, & Winfield, Alan FT. 2011. Open-hardware e-puck Linux extension board for experimental swarm robotics research. *Microprocessors and Microsystems*, **35**(1), 60–67.

Liu, Wenguo, Winfield, Alan FT, & Sa, Jin. 2007. Modelling swarm robotic systems: A case study in collective foraging. *Towards Autonomous Robotic Systems (TAROS 07)*, 25–32.

Luke, Sean, & Panait, Liviu. 2006. A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, **14**(3), 309–344.

Mack, Chris A. 2011. Fifty years of Moore's law. *IEEE Transactions on semiconductor manufacturing*, **24**(2), 202–207.

Manocha, Dinesh. 2005. General-purpose computations using graphics processors. *Computer*, **38**(8), 85–88.

Marques, Hugo Gravato, & Holland, Owen. 2009. Architectures for functional imagination. *Neurocomputing*, **72**(4), 743–759.

Marzinotto, Alejandro, Colledanchise, Michele, Smith, Colin, & Ogren, Petter. 2014. Towards a unified behavior trees framework for robot control. *Pages 5420–5427 of: IEEE International Conference on Robotics and Automation (ICRA 2014)*. IEEE, Hong Kong, China.

Mataric, Maja J. 1993. Designing emergent behaviors: From local interactions to collective intelligence. *Pages 432–441 of: Proceedings of the Second International Conference on Simulation of Adaptive Behavior.*

Mateas, Michael, & Stern, Andrew. 2002. A behavior language for story-based believable agents. *IEEE Intelligent Systems*, **17**(4), 39–47.

McLurkin, James, Lynch, Andrew J, Rixner, Scott, Barr, Thomas W, Chou, Alvin, Foster, Kathleen, & Bilstein, Siegfried. 2013. A low-cost multi-robot system for research, teaching, and outreach. *Pages 597–609 of:* Martinoli, A., Mondada, F., Correll, N., Mermoud, G., Egerstedt, M., Hsieh, M.A., Parker, L.E., & Støy, K (eds), *Distributed Autonomous Robotic Systems. The 10th International Symposium (DARS 2010).* Springer.

Menzel, Randolf, & Giurfa, Martin. 2001. Cognitive architecture of a mini-brain: the honeybee. *Trends in cognitive sciences*, **5**(2), 62–71.

Millard, Alan G, Timmis, Jon, & Winfield, Alan FT. 2013. Towards exogenous fault detection in swarm robotic systems. *Pages 429–430 of:* Natraj, Ashutosh, Cameron, Stephen, Melhuish, Chris, & Witkowski, Mark (eds), *Towards Autonomous Robotic Systems (TAROS 2013).* Springer, Oxford, UK.

Millard, Alan G, Timmis, Jon, & Winfield, Alan FT. 2014. Run-time detection of faults in autonomous mobile robots based on the comparison of simulated and real robot behaviour. *Pages 3720–3725 of:* Burgard, Wolfram (ed), *IEEE/RSJ*

International Conference on Intelligent Robots and Systems (IROS 2014). IEEE, Chicago, Illinois, USA.

Millard, Alan G, Joyce, Russell Andrew, Hilder, James Alan, Fleseriu, Cristian, Newbrook, Leonard, Li, Wei, McDaid, Liam, & Halliday, David Malcolm. 2017. The Pi-puck extension board: a Raspberry Pi interface for the e-puck robot platform. *In:* Maciejewski, Tony (ed), *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2017).* IEEE, Vancouver, Canada.

Minkovich, Kirill, Thibeault, Corey M, O'Brien, Michael John, Nogin, Aleksey, Cho, Youngkwan, & Srinivasa, Narayan. 2014. HRLSim: a high performance spiking neural network simulator for GPGPU clusters. *IEEE transactions on neural networks and learning systems*, **25**(2), 316–331.

Mitri, Sara, Floreano, Dario, & Keller, Laurent. 2009. The evolution of information suppression in communicating robots with conflicting interests. *Proceedings of the National Academy of Sciences*, **106**(37), 15786–15790.

Mondada, Francesco, Bonani, Michael, Guignard, André, Magnenat, Stéphane, Studer, Christian, & Floreano, Dario. 2005. Superlinear physical performances in a SWARM-BOT. *Pages 282–291 of:* Capcarrere, M., Freitas, A.A., Bentley, P.J., Johnson, C.G., & Timmis, J (eds), *The 8th European Conference on Artificial Life (ECAL 2005).* Springer, Canterbury, UK.

Mondada, Francesco, Bonani, Michael, Raemy, Xavier, Pugh, James, Cianci, Christopher, Klaptocz, Adam, Magnenat, Stéphane, Zufferey, Jean-Christophe, Floreano, Dario, & Martinoli, Alcherio. 2009. The e-puck, a robot designed for education in engineering. *Pages 59–65 of:* Gonçalves, P.J.S., Torres, Paulo, & Alves, C.M.O. (eds), *Proceedings of the 9th conference on autonomous robot systems and competitions (ROBOTICA 2009)*, vol. 1.

Mouret, Jean-Baptiste, & Chatzilygeroudis, Konstantinos. 2017. 20 Years of Reality Gap: a few Thoughts about Simulators in Evolutionary Robotics. *In: Workshop "Simulation in Evolutionary Robotics", Genetic and Evolutionary Computation Conference (GECCO 2017).* ACM, Berlin, Germany.

Nelson, Andrew L, Barlow, Gregory J, & Doitsidis, Lefteris. 2009. Fitness functions in evolutionary robotics: A survey and analysis. *Robotics and Autonomous Systems*, **57**(4), 345–370.

Niiranen, Jouko. 1999 (09). Fast and accurate symmetric Euler algorithm for electromechanical simulations NOTE: The method became later known as "Symplectic Euler". *Pages 71–78 of: Proceedings of the 6th International Conference ELECTRIMACS '99: Modelling and Simulation of Electric Machines, Converters and Systems*, vol. 1.

Nolfi, Stefano, Floreano, Dario, & Floreano, Director Dario. 2000. *Evolutionary robotics: The biology, intelligence, and technology of self-organizing machines.* MIT press.

Nvidia. 2007. *NVIDIA CUDA, Compute Unified Device Architecture Programming Guide.* NVIDIA.

O'Dowd, Paul J, Studley, Matthew, & Winfield, Alan FT. 2014. The distributed co-evolution of an on-board simulator and controller for swarm robot behaviours. *Evolutionary Intelligence*, **7**(2), 95–106.

Ogren, Petter. 2012. Increasing modularity of UAV control systems using computer game behavior trees. *In: AIAA Guidance, Navigation and Control Conference.* AIAA, Minneapolis, MN, USA.

Olfati-Saber, Reza. 2006. Flocking for multi-agent dynamic systems: Algorithms and theory. *IEEE Transactions on automatic control*, **51**(3), 401–420.

Oppermann, Reinhard, & Rasher, R. 1997. Adaptability and adaptivity in learning systems. *Knowledge transfer*, **2**, 173–179.

Park, Jong Jin, & Kuipers, Benjamin. 2011. A smooth control law for graceful motion of differential wheeled mobile robots in 2d environment. *Pages 4896–4902 of: IEEE International Conference on Robotics and Automation (ICRA 2011).* IEEE, Shanghai, China.

Peng, Xue Bin, Andrychowicz, Marcin, Zaremba, Wojciech, & Abbeel, Pieter. 2018. Sim-to-real transfer of robotic control with dynamics randomization. *Pages 1–8 of: IEEE International Conference on Robotics and Automation (ICRA 2018).* IEEE, Brisbane, Australia.

Perez, Diego, Nicolau, Miguel, O'Neill, Michael, & Brabazon, Anthony. 2011. Evolving behaviour trees for the mario ai competition using grammatical evolution. *Pages 123–132 of: Applications of evolutionary computation.* Springer.

Pinciroli, Carlo, Trianni, Vito, O'Grady, Rehan, Pini, Giovanni, Brutschy, Arne, Brambilla, Manuele, Mathews, Nithin, Ferrante, Eliseo, Di Caro, Gianni, Ducatelle, Frederick, *et al.* . 2011. ARGoS: a modular, multi-engine simulator for heterogeneous swarm robotics. *Pages 5027–5034 of:* Amato, Nancy M. (ed), *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2011(.* IEEE, San Francisco, USA.

Poli, Riccardo, Langdon, William B, McPhee, Nicholas F, & Koza, John R. 2008. *A field guide to genetic programming.* Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`.

Polilov, Alexey A. 2012. The smallest insects evolve anucleate neurons. *Arthropod structure & development*, **41**(1), 29–34.

Quigley, Morgan, Conley, Ken, Gerkey, Brian, Faust, Josh, Foote, Tully, Leibs, Jeremy, Wheeler, Rob, & Ng, Andrew Y. 2009. ROS: an open-source Robot Operating System. *Page 5 of: IEEE International Conference on Robotics and Automation (ICRA 2009) Workshop on Open Source Robotics*, vol. 3. IEEE, Kobe, Japan.

Reinecke, Philipp, Wolter, Katinka, & Van Moorsel, Aad. 2010. Evaluating the adaptivity of computing systems. *Performance Evaluation*, **67**(8), 676–693.

Reynolds, Craig W. 1987. Flocks, herds and schools: A distributed behavioral model. *Pages 25–34 of: ACM SIGGRAPH Computer Graphics*, vol. 21. ACM.

Rodeh, Ohad, Bacik, Josef, & Mason, Chris. 2013. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, **9**(3), 9.

Rostedt, Steven, & Hart, Darren V. 2007. Internals of the RT Patch. *Pages 161–172 of:* Lockhart, John W., Ozen, Gurhan, Feeney, John, DiMaggio, Len, & Poelstra, John (eds), *Ottawa Linux Symposium*, vol. 2.

Rubenstein, Michael, Ahler, Christian, & Nagpal, Radhika. 2012. Kilobot: A low cost scalable robot system for collective behaviors. *Pages 3293–3298 of:* Parker, Lynne (ed), *IEEE International Conference on Robotics and Automation (ICRA 2012).* IEEE, St. Paul, MN, USA.

Rusu, Andrei A, Vecerik, Mel, Rothörl, Thomas, Heess, Nicolas, Pascanu, Razvan, & Hadsell, Raia. 2016. Sim-to-real robot learning from pixels with progressive nets. *arXiv preprint arXiv:1610.04286.*

Şahin, Erol. 2005. Swarm robotics: From sources of inspiration to domains of application. *Pages 10–20 of:* Şahin E., Spears W.M. (ed), *International Workshop on Swarm robotics (SR 2004).* Santa Monica, CA, USA: Springer.

Samek, Wojciech, Wiegand, Thomas, & Müller, Klaus-Robert. 2017. Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models. *arXiv preprint arXiv:1708.08296.*

Scheper, Kirk YW, Tijmons, Sjoerd, de Visser, Cornelis C, & de Croon, Guido CHE. 2016. Behavior Trees for Evolutionary Robotics. *Artificial life*, **22**(1), 23–48.

Schwefel, Hans-Paul Paul. 1993. *Evolution and optimum seeking: the sixth generation.* John Wiley & Sons, Inc.

Shoulson, Alexander, Garcia, Francisco M, Jones, Matthew, Mead, Robert, & Badler, Norman I. 2011. Parameterizing behavior trees. *Pages 144–155 of: International Conference on Motion in Games.* Springer.

Siegwart, Roland, Nourbakhsh, Illah Reza, & Scaramuzza, Davide. 2011. *Introduction to autonomous mobile robots.* Cambridge, MA: MIT press.

Stanley, Kenneth O, & Miikkulainen, Risto. 2002. Evolving neural networks through augmenting topologies. *Evolutionary computation*, **10**(2), 99–127.

Tabak, Daniel, & Lipovski, G. Jack. 1980. MOVE architecture in digital controllers. *IEEE Journal of Solid-State Circuits*, **15**(1), 116–126.

Texas Instruments. 2013. *INA231 High- or Low-Side Measurement, Bidirectional Current and Power Monitor With 1.8-V I^2C Interface.*

Thrun, Sebastian, Burgard, Wolfram, & Fox, Dieter. 2005. *Probabilistic Robotics.* Intelligent robotics and autonomous agents. MIT Press.

Tobin, Josh, Fong, Rachel, Ray, Alex, Schneider, Jonas, Zaremba, Wojciech, & Abbeel, Pieter. 2017. Domain randomization for transferring deep neural networks from simulation to the real world. *Pages 23–30 of:* Maciejewski, Tony (ed), *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2017).* IEEE, Vancouver, Canada.

Trianni, Vito. 2008. *Evolutionary swarm robotics: evolving self-organising behaviours in groups of autonomous robots.* Vol. 108. Springer.

Trianni, Vito, Tuci, Elio, Ampatzis, Christos, & Dorigo, Marco. 2014. Evolutionary Swarm Robotics: a theoretical and methodological itinerary from individual neuro-controllers to collective behaviours. *The Horizons of Evolutionary Robotics*, 153.

Usui, Yukiya, & Arita, Takaya. 2003. Situated and embodied evolution in collective evolutionary robotics. *Pages 212–215 of: 8th International Symposium on Artificial Life and Robotics (AROB 8th 2003).* ACM, Beppu, Oita, Japan.

Vanderelst, Dieter, & Winfield, Alan. 2018. An architecture for ethical robots inspired by the simulation theory of cognition. *Cognitive Systems Research*, **48**, 56–66.

Vargas, Patricia A, Di Paolo, Ezequiel A, Harvey, Inman, & Husbands, Phil. 2014. *The horizons of evolutionary robotics.* MIT press.

Vásárhelyi, Gábor, Virágh, Cs, Somorjai, Gergo, Tarcai, Norbert, Szörényi, Tamás, Nepusz, Tamás, & Vicsek, Tamás. 2014. Outdoor flocking and formation flight with autonomous aerial robots. *Pages 3866–3873 of:* Burgard, Wolfram (ed), *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2014).* IEEE, Chicago, Illinois, USA.

Vaughan, Richard. 2008. Massively multi-robot simulation in Stage. *Swarm Intelligence*, **2**(2-4), 189–208.

Wang, F. 1994. The use of artificial neural networks in a geographical information system for agricultural land-suitability assessment. *Environment and planning A*, **26**(2), 265–284.

Wang, Minghui, Ma, Shugen, Li, Bin, Wang, Yuechao, He, Xinyuan, & Zhang, Liping. 2005. Task planning and behavior scheduling for a reconfigurable planetary robot system. *Pages 729–734 of: IEEE International Conference Mechatronics and Automation, 2005*, vol. 2. IEEE.

Watson, Richard A, Ficici, Sevan G, & Pollack, Jordan B. 2002. Embodied evolution: Distributing an evolutionary algorithm in a population of robots. *Robotics and Autonomous Systems*, **39**(1), 1–18.

Weber, Ben G, Mawhorter, Peter, Mateas, Michael, & Jhala, Arnav. 2010. Reactive planning idioms for multi-scale game AI. *Pages 115–122 of: IEEE Conference on Computational Intelligence and Games (CIG 2010)*. IEEE, Copenhagen, Denmark.

Whitley, Darrell. 1994. A genetic algorithm tutorial. *Statistics and computing*, **4**(2), 65–85.

Whitley, Darrell. 2001. An overview of evolutionary algorithms: practical issues and common pitfalls. *Information and software technology*, **43**(14), 817–831.

Whitley, Darrell, & Starkweather, Timothy. 1990. Genitor II: A distributed genetic algorithm. *Journal of Experimental & Theoretical Artificial Intelligence*, **2**(3), 189–214.

Whitley, Darrell, Rana, Soraya, & Heckendorn, Robert B. 1997. Island model genetic algorithms and linearly separable problems. *Pages 109–125 of:* Corne, David, & Shapiro, Jonathan L. (eds), *AISB International Workshop on Evolutionary Computing*. Springer, Manchester, United Kingdom.

Whitley, Darrell, Rana, Soraya, & Heckendorn, Robert B. 1999. The island model genetic algorithm: On separability, population size and convergence. *CIT. Journal of computing and information technology*, **7**(1), 33–47.

Williams, Robert L, Carter, Brian E, Gallina, Paolo, & Rosati, Giulio. 2002. Dynamic model with slip for wheeled omnidirectional robots. *IEEE transactions on Robotics and Automation*, **18**(3), 285–293.

Wilson, Sean, Gameros, Ruben, Sheely, Michael, Lin, Matthew, Dover, Kathryn, Gevorkyan, Robert, Haberland, Matt, Bertozzi, Andrea, & Berman, Spring. 2016.

Pheeno, a versatile swarm robotic research and education platform. *IEEE Robotics and Automation Letters*, **1**(2), 884–891.

Winfield, Alan FT. 2009a. Foraging robots. *Pages 3682–3700 of: Encyclopedia of Complexity and Systems Science.* Springer.

Winfield, Alan FT. 2009b. Towards an engineering science of robot foraging. *Pages 185–192 of:* Asama, H, Kurokawa, H, Ota, J, & Sekiyama, K (eds), *9th International Symposium on Distributed Autonomous Robotic Systems (DARS 2008).* Tsukuba, Japan: Springer.

Winfield, Alan FT. 2015. Robots with internal models: a route to self-aware and hence safer robots. *Pages 237–252 of: The computer after me: Awareness and self-awareness in autonomic systems.* World Scientific.

Winfield, Alan FT, Liu, Wenguo, Nembrini, Julien, & Martinoli, Alcherio. 2008. Modelling a wireless connected swarm of mobile robots. *Swarm Intelligence*, **2**(2-4), 241–266.

Winfield, Alan FT, Blum, Christian, & Liu, Wenguo. 2014. Towards an ethical robot: internal models, consequences and ethical action selection. *Pages 85–96 of:* Mistry, Michael, Leonardis, Aleš, Witkowski, Mark, & Melhuish, Chris (eds), *Towards Autonomous Robotic Systems (TAROS 2014).* Birmingham, UK: Springer.

Wright, Sewall. 1943. Isolation by distance. *Genetics*, **28**(2), 114.

Xianyi, Zhang, Qian, Wang, & Yunquan, Zhang. 2012. Model-driven level 3 BLAS performance optimization on Loongson 3A processor. *Pages 684–691 of: IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS 2012).* IEEE, Singapore.

Zagal, Juan Cristóbal, Ruiz-del Solar, Javier, & Vallejos, Paul. 2004. Back to reality: Crossing the reality gap in evolutionary robotics. *In: 5th IFAC Symposium on Intelligent Autonomous Vehicles (IAV 2004).*