School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Using 3D Simulation to Develop Robot Code

by

Jeremy James Collette

Thesis submitted as a requirement for the degree of

Bachelor of Science (Computer Science)

Submitted:	11th August 2017	Student ID:	z5105377
Supervisor:	B. Hengst, T. Wiley	Topic ID:	3723

Abstract

In the last decade, many UNSW Sydney students have researched simulation, and even built their own simulators. However, the resulting simulators are either too divergent from the physical world, too focused in their simulation target, or they require major architectural changes to *rUNSWift* code for compatibility. There exists the need for a simulator that can accurately simulate internal modules of the rUNSWift software, without requiring major changes to the codebase itself.

This thesis presents *simswift*, a new build target of the rUNSWift source code. simswift uses the same internal modules as rUNSWift, but replaces the hardware facing *libagent* module with the *Simulation* module. The Simulation module connects to the already available RoboCup 3D Simulation League Simulator and controls a simulated NAO using the existing rUNSWift *Perception*, *Behaviour*, and *Motion* modules. As the Simulation module provides the same interface as libagent, the two can be used interchangeably and neighbouring modules are unaffected. The rUNSWift codebase can be built as simswift for testing in simulation before being re-built as rUNSWift and uploaded to a robot.

An important part of robotic soccer is searching for the ball. To demonstrate its efficacy, simswift was used to develop a new ball searching component in complete simulation. This thesis presents the new component, and compares it with its predecessors both in simulation and on the NAO robot. The results are assessed to evaluate the utility of developing robot code in simswift, and the simulated and real-life tests are compared to assess the fidelity of the simulation.

Acknowledgements

I'd like to thank my supervisor Bernhard Hengst, for his guidance and wisdom throughout my thesis. I'd also like to thank Tim Wiley, who went beyond his call of duty to offer his advice and support.

I owe all my achievements to my family and friends. Thanks to my parents Lynne and Glen, and my brother Justin, for always being there. Thanks to my life-coach Mark, for helping me solve the world's problems by the pool table. Thanks to my friend Adam, for always making me feel better by comparison.

And finally, I owe my most sincere thanks to my best friend and mentor, Karl. Without your support and encouragement, I wouldn't be where I am today. Thanks.

Contents

1 Introduction **1**

1.1 Contributions 3

2 Background **5**

2.1 *rUNSWift* 5

2.1.1 Perception 7

2.1.2 Behaviour 9

2.1.3 Motion 10

2.1.4 *Blackboard* 11

2.1.5 *libagent* 11

2.1.6 *offnao* 12

2.2 Simulation 13

2.2.1 The RoboCup 3D League Simulator 15

2.2.2 Related work 16

2.3 Finding the Ball 20

2.3.1 Ball Detection 21

2.3.2 Ball Searching 21

2.3.3 Related work 29

3 Simulation using *simswift* **32**

3.1 Overview . . . 32

3.2 Simulation module . . . 35

3.2.1 Where it comes together: *SimulationThread* . . . 35

3.2.2 Connecting the 3DL Simulator: *SimulationConnection* . . . 37

3.2.3 Simulating vision: *SimVisionAdapter* . . . 37

3.2.4 Simulating the X/Y angle: *AngleSensor* . . . 38

3.2.5 Simulating the sonars: *SonarSensor* . . . 39

3.2.6 Running an experiment: *ExperimentControllerInterface* . . . 39

3.3 Divergences between the simulator and SPL . . . 40

3.3.1 The NAO . . . 40

3.3.2 The environment . . . 49

4 Finding the Ball **50**

4.1 Overview . . . 50

4.2 New ball searching component implementation . . . 51

4.2.1 Estimating probability: *BallDistributionGenerator* implementations . . . 53

4.2.2 Where to go: *SimpleStrategy* class . . . 57

4.2.3 Getting there: *FindBall* Python behaviour . . . 59

4.2.4 *offnao* support . . . 61

5 Evaluation **63**

5.1 Searching the field . . . 63

5.1.1 Tests in simulation . . . 66

5.1.2 Tests on the NAO V5 . . . 67

5.2 New ball searching component . . . 68

5.2.1 Simulation results evaluation 69
5.2.2 Real-life results evaluation 71
5.3 *simswift* VS the world 73
5.3.1 Vision 73
5.3.2 Motion 74

6 Conclusion **77**
6.1 Future Work 78

Bibliography **81**

List of Figures

1.1 A UNSW Sydney NAO V5 robot, after just kicking the ball at the SPL world championship in Nagoya, 2017. 1

1.2 UNSW Sydney's 2017 RoboCup SPL team: Team rUNSWift. 2

2.1 A diagram illustrating the perception, planning, and actuation workflow. 5

2.2 A diagram illustrating the rUNSWift architecture at a high level, following the pattern described in Figure 2.1. 6

2.3 An example goal observation. 8

2.4 The two possible robot poses that could produce the frame in figure 2.3. 9

2.5 A screencap of offnao connected to a virtual NAO being controlled by *simswift*. 12

2.6 An abstraction of the architecture of simulating generic autonomous systems, based on figure 2.1. 13

2.7 The static search locations for the 2016 FindBall behaviour, described in Table 2.1 22

2.8 The static search locations for the 2016 FindBall behaviour, with the surrounding corners of the field annotated. Based on 2.7. 24

2.9 The static search locations for the 2016 FindBall behaviour, and their distances to surrounding edges of the field. Based on 2.8. 24

2.10 The theoretical maximum coverage of the static search points in the 2016 FindBall behaviour, with an assumed reliable ball detection distance of 2764mm. 25

2.11 The theoretical complete maximum coverage of the static search points in the 2016 FindBall behaviour, assuming that robots travel between points in a straight line. Also with an assumed reliable ball detection distance of 2764mm. 26

2.12 The theoretical complete coverage of the 2016 FindBall behaviour with a reliable ball detection distance of 2500mm. 27

2.13 A screencap of a game between rUNSWift and Berlin United at RoboCup in Leipzig, 2016. A rUNSWift robot (in yellow) is depicted standing next to the ball but is unable to detect it. The left eye of robots running rUNSWift turn red when they are able to see the ball. Here it has no colour, indicating no ball has been seen. 28

2.14 The theoretical complete coverage of the 2016 FindBall behaviour with a reliable ball detection distance of 1000mm, which is likely greater than the ability of the detector at RoboCup in Leipzig, 2016. 29

3.1 Five copies of simswift running simultaneously simulating a team of robots, as visualised by *RoboViz*. 33

3.2 The simswift architecture, based on the rUNSWift architecture seen in figure 2.2. Note that the *Vision* component of Perception has been changed to a simulated variant. 34

3.3 The Simulation Module architecture, showing the interactions between components and neighbouring modules. One level more detailed than Figure 3.2. 35

3.4 The difference between the gyroscope axes on the NAO V5 and the simulated NAO. 42

4.1 The proposed ball searching component architecture. 51

4.2 The implementation of the architecture described in Figure 4.1. 52

4.3 The theoretical coverage of the presented ball searching component, given the search points listed in Table 4.1. 54

4.4 The complete theoretical coverage of one possible route that could be taken by the presented ball searching component, given the search points listed in Table 4.1. 55

4.5 The architecture surrounding the TeamDistributionGenerator described in Section 4.2.1.2. It is an implementation of the BallDistributionGenerator. 57

4.6 The architecture of the distributed SimpleStrategy, including the DistributedUnityCalculator defined in Section 4.2.2.2. 59

4.7 A more in-depth visualisation of the architecture shown in Figure 4.6. . 60

4.8 The *FindBall* tab on offnao. 62

5.1 The ball positions and robot starting positions used for experiments. Balls are shown in green, and starting positions are shown in other colours. Based on Tables 5.2 and 5.1. 65

5.2 Shows the search routes of robots throughout 100 simulated tests using the current ball searching component (2016). 70

5.3 Shows the search routes of robots throughout 100 simulated tests using the new ball searching component (2017). 70

List of Tables

2.1 2016 ball searching points 22

3.1 Noise in 3DS Simulator visual observations [BDR+10] 41

3.2 Joint mapping from physical NAO V5 interface to simulated NAO interface 44

3.3 An example of poor joint actuation using the simulator 46

3.4 Differences between the RoboCup SPL and 3DSL environment 49

4.1 The points used for the whole field search test. 53

5.1 The starting points for each robot number used. 64

5.2 The points used for the whole field search test. 65

5.3 The data from, 100 tests in simulation using the 2016 ball searching component. 66

5.4 The data from, 100 tests in simulation using the new ball searching component (2017). 66

5.5 The data from 25 tests on a single NAO V5 using the existing ball searching component (2016). 67

5.6 The data from 25 tests on a single NAO V5 using the new ball searching component (2017). 67

5.7 The data from 25 tests using three NAO V5s with the existing ball searching component (2016). 68

5.8 The data from 25 tests using three NAO V5s with the new ball searching component (2017). 68

5.9 A summary of all the balls found by the existing ball searching component (2016) and the new ball searching component (2017). 69

5.10 A summary of the average time it took the existing ball searching component (2016) and the presented ball searching component (2017) to find the ball in simulation with three robots. 70

5.11 A summary of all the balls found by the existing ball searching component (2016) and the new ball searching component (2017) in real-life single and multi-robot tests. 71

5.12 A summary of the average time it took the existing ball searching component (2016) and the presented ball searching component (2017) to find the ball using three real NAO V5 robots. 72

5.13 A summary of all the tests that were run with three robots. 73

5.14 The comparitive find rate of ball positions between simulated and real-life tests. 74

5.15 The real speed of a simulated robot programmed to walk at 250mm/s. . 74

5.16 The comparitive average time to find balls at each position between simulated and real-life tests, adjusted for speed bias. 75

5.17 The difference in adjusted average search times for ball positions, between simulated and real experiments. 75

Chapter 1

Introduction

RoboCup is an international robotics competition, where teams compete in varying challenges using robots. The Standard Platform League (SPL) is an autonomous soccer competition, where competing teams each write their own software for a team of standardised robots. At the time of writing, the SPL uses the NAO V5 robot made by SoftBank.

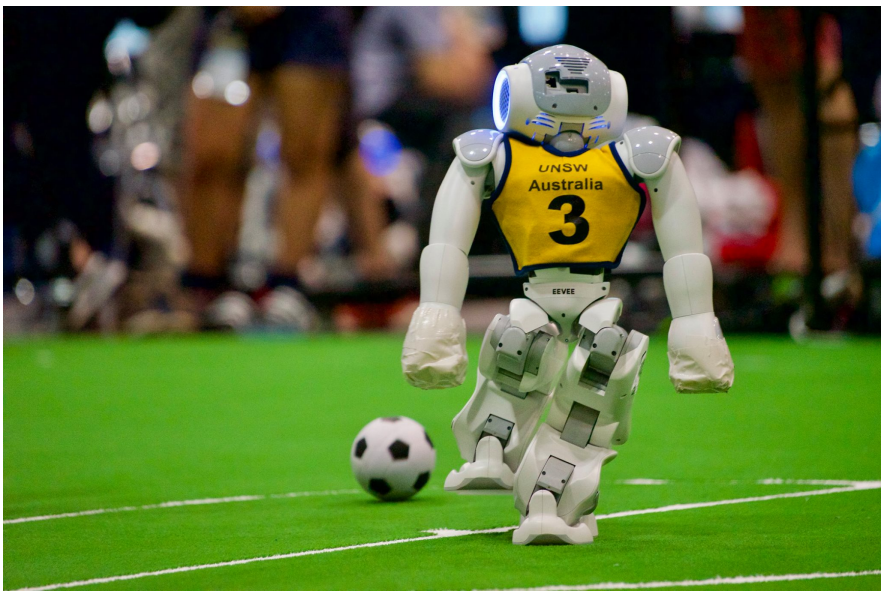

Figure 1.1: A UNSW Sydney NAO V5 robot, after just kicking the ball at the SPL world championship in Nagoya, 2017.

UNSW Sydney has been competing in RoboCup since 1999, and has quite the legacy. The team has won five world championships, three of which were in the Four-Legged League (2000, 2001, and 2003), and two back-to-back SPL championships in 2014 and 2015. In 2017, the team placed in the quarter finals at the SPL championship in Nagoya, Japan.

Figure 1.2: UNSW Sydney's 2017 RoboCup SPL team: Team rUNSWift.

UNSW Sydney's robotic software, termed *rUNSWift*, is the cumulative effort of many different iterations of team members. With each year, SPL rules changes precipitate additions and modifications to the rUNSWift codebase. Additionally, new team members bring fresh perspective and ideas, which results in improvements to the code. A consequence of these factors is that rUNSWift is perpetually in active development. The team works all year round to preserve the competency of the codebase. The majority of this work is developing and testing code on the NAO robots.

However, developing code on physical robots is often difficult, and there are many obvious benefits to developing robot code in simulation instead. As such, the team finds a need for an accurate and versatile simulation tool, that enables the simulation of rUNSWift code on a personal computer, instead of running on a physical NAO. The simulation tool should have minimal divergence from real-life, and should be compatible with the rUNSWift codebase without any major changes. This thesis presents such a

simulation tool, termed *simswift*. simswift is a new build target of the rUNSWift codebase that utilises the already existing RoboCup 3D Simulation League Simulator, *SimSpark*, to simulate how rUNSWift itself would perform on a physical robot.

To evaluate the effectiveness of simswift, it was used to develop robot code. The recent introduction of the SPL black and white ball has severely reduced the rUNSWift ball detection accuracy, especially at long distances. As the current rUNSWift ball searching component was developed with the assumption of a greater ball detection distance, it does not perform optimally with the current ball detector. A new ball searching component architecture and implementation was developed using simswift, in complete simulation. This thesis presents the new ball searching component, which is compared with previous infrastructure in both simulation (using simswift) and real-life. The tests in simulation are compared and the tests in real-life are compared to assess the utility of the new ball searching component. Then, the tests in simulation are compared with the equivalent tests in real-life, to assess the fidelity of the simswift simulation.

In this thesis, Chapter 2 explores the Background of rUNSWift, simulation in SPL, and ball searching. Chapter 3 presents simswift, a simulator-friendly build target of the rUNSWift codebase. Chapter 4 presents a new ball searching component architecture and implementation, developed in complete simulation. Chapter 5 evaluates the new ball searching component, and the effectiveness of using simswift to develop robot code. Chapter 6 discusses the conclusions that can be drawn from the evaluation, and suggests future work that could complement this thesis.

1.1 Contributions

In Chapter 3, this thesis presents *simswift*, a simulator-friendly build target of the same codebase as *rUNSWift*. *simswift* uses the same core components as rUNSWift, but replaces the hardware-facing component *libagent* with the *Simulation* component, which communicates with simulated NAO hardware instead.

In Chapter 4, this thesis presents a new (2017) distributed ball searching component

(BSC). The 2017 BSC uses probability estimation to perform a distributed search of the SPL field. The presented component is configurable depending on the reliable detection distance of the ball detector in use, and the size of the field being searched.

Chapter 2

Background

2.1 *rUNSWift*

rUNSWift is the name of the software written by UNSW Sydney for the RoboCup SPL robot, NAO V5. rUNSWift is an autonomous real-time system, where each update cycle consists of perception, planning, and actuation. Perception senses the current state of the environment, planning determines the action to take, and actuation carries out the action accordingly. This is a generalised workflow that can be used to describe many autonomous systems that interact with their environment. Figure 2.1 illustrates this relationship.

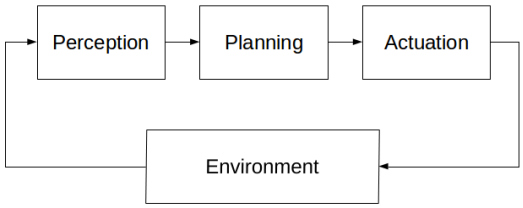

Figure 2.1: A diagram illustrating the perception, planning, and actuation workflow.

rUNSWift implements the perception, planning, and actuation processes with over 200,000 lines of C++ and Python source code. Each process is handled by separate modules, namely Perception, Behaviour, and Motion. Each of these modules are made

up of components, which are used in conjunction with other components to provide functionality. Figure 2.2 gives an abstracted overview of this architecture.

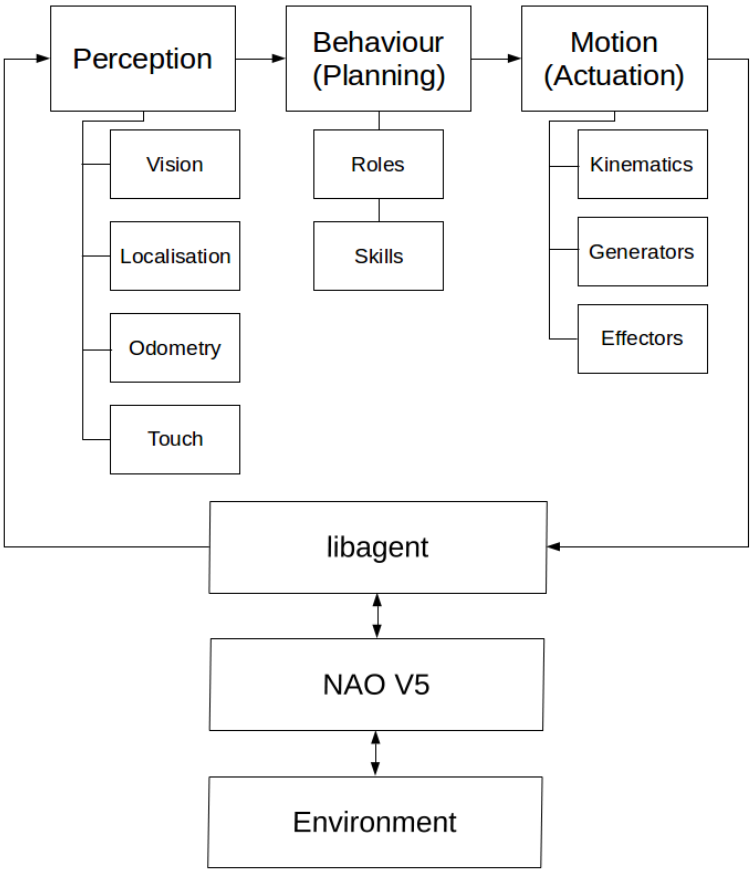

Figure 2.2: A diagram illustrating the rUNSWift architecture at a high level, following the pattern described in Figure 2.1.

Components are able to communicate with each other using a *Blackboard* shared memory component. Furthermore, components are able to communicate directly with *libagent*, the module which interfaces with robot hardware, using another separate shared memory component. This is used for the perception of sensors and joint angles, and the actuation of joints.

2.1.1 Perception

The *Perception* module senses the environment of the robot, and uses this information to maintain a world model. A world model is essentially the robot's estimation of the environment around it. The NAO V5 has two cameras, four microphones, and two sets of sonars, which it uses to perceive the environment. [Ald] Additionally, a gyrometer, an acceleromoter, an X/Y angle sensor, joint angle sensors, and eight force sensitive resistors (FSR) are used to track the robots pose throughout the constructed world model.

Vision is a core component of the Perception module. The Vision component uses the two cameras, one on the NAO's forehead and one just above the would-be chin, as the main sensors for the perception of field features. Field features are predefined physical objects that we expect to find on the SPL field, such as lines, goal posts, the ball, or other robots. Each of these field features are found by purpose-built detectors, that use computer vision to find them in image frames from the cameras. Once a detector has found a field feature, the information from the two-dimensional image is transposed to estimate the object's pose relative to the robot, and the object is added to the robot's world model.

Localisation is arguably of equal importance to Vision for the purpose of Perception. Given the field features detected by Vision, the Localisation component builds a world model of the robot's environment, and estimates the pose of the robot inside the environment. As the SPL environment is well defined, the robot can make estimations about its position given the field features it has observed. Unfortunately, as the field itself and the placement of field features are both symmetrical, the mapping from the observation state space to the robot pose state space is not bijective. This means that, given any singular frame of field feature observations, there are multiple poses on the field that could produce the same observations. For example, consider the case of observing two goal posts in a single frame. Even given our distance and heading to the posts, we can still be at two points on the field, which are mirrors of each other. Figure 2.3 illustrates an example of such a frame, and Figure 2.4 illustrates the two possible

Figure 2.3: An example goal observation.

robot poses that could produce such a frame.

To localise in such an environment, Sushkov researched and implemented a *Distributed Multi-Modal Extended Kalman Filter* [Sus06]. The filter keeps track of multiple positional hypotheses, and increases or decreases their probability of being correct depending on their conformance with sensed field features.

Odometry is the estimation of change in position over time using the information read from sensors. Using the gyroscopes and accelerometers, the Odometry component calculates the movement of the robot since the last update cycle. It assists the Localisation module in keeping accurate estimations of the robot's pose by providing movement information.

The *Touch* component provides information from sensors that are not visual, including the sonars, FSRs, and joint angles. This gives the robot a better understanding of its environment and its pose within the environment.

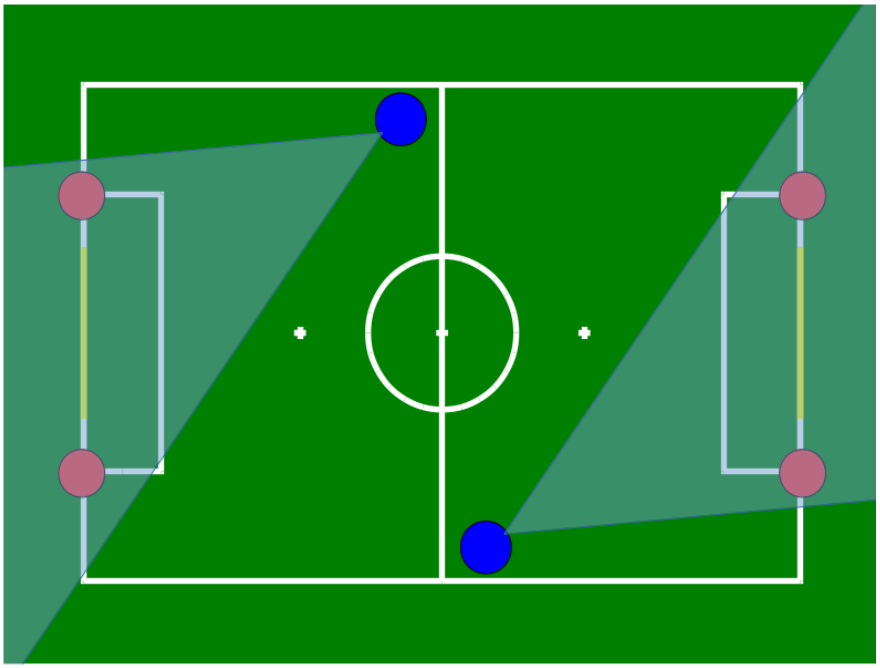

Figure 2.4: The two possible robot poses that could produce the frame in figure 2.3.

2.1.2 Behaviour

The *Behaviour* module, given the current state of the robot and its sensed environment from the Perception module, will determine the action to take until the next update cycle. It is the Planning stage of the autonomous workflow described in figure 2.1, and can be thought as of the control centre of the robot, where behavioural decisions are made.

The rUNSWift behavioural architecture has evolved over time, with the last major changes occurring in 2010. [RHH+10] As it stands, the Behaviour module is organised in to a hierarchy, where lower-level reusable components, *Skills*, are built upon and combined to create versatile behavioural policies, known as *Roles*. A few examples of Behaviour Skills at the time of writing are *ApproachBall*, *FindBall*, and *TurnDribble*. These are a sample of some of the Skills used to create Roles including *Striker*, *Upfielder*, *Midfielder*, *Defender*, and *Goalie*.

During an SPL game, each robot is dynamically assigned a Role, which is determined

depending on certain criteria. For example, the closest robot to the ball is delegated the *Striker* role. Each Role is only assigned to one robot at any given time. To assist the allocation process, robots share information between each other, including their estimated pose on the field and whether or not they can see the ball. As the game progresses, Role assignments are reallocated to different robots as their suitability for each Role changes. As an exception, the *Goalie* role is statically allocated, unless the assignee becomes incapacitated.

The behavioural Roles and Skills are written in Python, as behaviour development can benefit from the rapid development promoted by Python. Python boasts a fast build cycle, as code is byte-code interpreted. This avoids the long compile times typical of C/C++, and allows source code to be edited and interpreted directly on the robot. Furthermore, the 2010 rUNSWift team implemented automatic reloading for behaviours, which meant any changes to behavioural Python source files are automatically detected and loaded during execution. [RHH+10]

The Python components of the Behaviour module are executed using a Python interpreter which is embedded in to the rUNSWift C++ executable. This allows the C/C++ rUNSWift modules and the Python components of the Behaviour module to communicate with each other.

2.1.3 Motion

The rUNSWift *Motion* module, given the current request from the Behaviour module, will actuate the joints of the robot to fulfil the request. The Motion module is how rUNSWift moves the NAO to interact with its environment. It is the Actuation stage of the autonomous workflow described in Figure 2.1.

The Motion module is made up of *Generators* and *Effectors*. Generators read the current request from the Behaviour module and the current state of the robot to generate the joint angles required to fulfil the request. Effectors are supplied the desired joint angles from the selected Generator, and actuate the robot to move each joint to the

corresponding angle.

There are different Generators for different kinds of motion. For example, the *Walk2014Generator* is used to fulfil Walk or Kick commands, as requested by Behaviour requests. Meanwhile, the *ActionGenerator* is used to replicate static actions, such as front or back getups, also requested by Behaviour requests.

Each implementation of the Effector class is used for actuating a specific target. In the case of SPL, the only physical target we actuate is the NAO robot, but this could change. It is beneficial to have an abstract interface that can be implemented for any new actuation targets that may be introduced.

2.1.4 *Blackboard*

The *Blackboard* module is a 'blackboard' shared memory component, which is part of the blackboard behavioural software design pattern. A *blackboard* provides a central point of knowledge for cooperating modules that allows them to interface without directly communicating. This is appropriate for systems with large, complex modules that have changing interface requirements, or for systems where modules are separated by threads. In the case of rUNSWift, both of these are true.

2.1.5 *libagent*

libagent is a module which is used to communicate with the NAO hardware, via the provided *Device Communication Manager* (DCM) module. It is the interface between rUNSWift and the NAO.

libagent shares a block of memory with rUNSWift, where it stores sensory information to be read. rUNSWift uses the same block of shared memory to write joint commands for libagent to actuate, via DCM. It should be noted that this shared memory is separate to the Blackboard, for performance reasons.

2.1.6 *offnao*

offnao is a rUNSWift debugging tool that was developed by the team in 2009 [RHH+10]. offnao communicates with robots running rUNSWift over a network connection, and provides a visual display of the state of the robot.

offnao is made up of a number of tabs, each with different information about the robot. The main tab, *Overview*, displays an overview of the state of the robot. It plots the estimated position of the robot on an image of the SPL field, alongside the estimated position of teammates and any field features, balls, or robots that are currently being detected. On the right side of the tab, saliency images from the top and bottom camera are displayed, with detections highlighted. Next to the camera frames, data from the *Blackboard* is displayed, such as the details of visual detections that have been made. Figure 2.5 is a screencap of offnao connected to a simulated NAO controlled by *simswift*. Note that, as the simulated robot does not use a simulated camera for perception, the top and bottom image frames are blank.

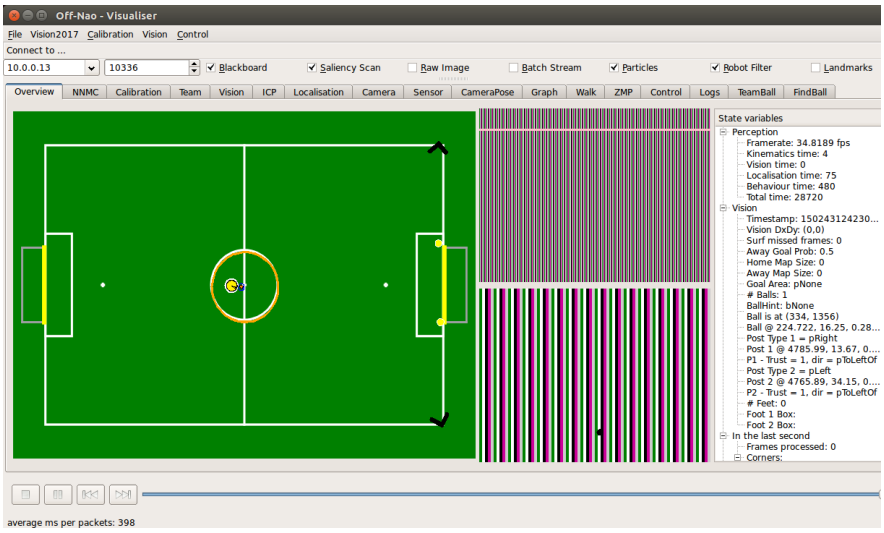

Figure 2.5: A screencap of offnao connected to a virtual NAO being controlled by *simswift*.

2.2 Simulation

In the context of robotics, simulation entails the careful modelling of a robot, its environment, and their relationship. Then, changes can be made to either entity to observe its affect on the other. Figure 2.6 shows an abstraction of the architecture of a generic autonomous system being simulated, based on figure 2.1. Note that the *Environment* entity has been replaced with the *Simulator*.

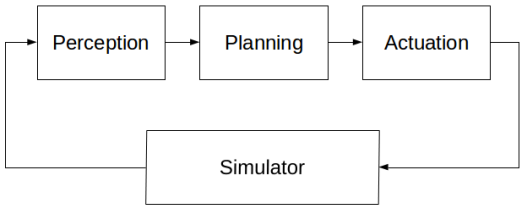

Figure 2.6: An abstraction of the architecture of simulating generic autonomous systems, based on figure 2.1.

In RoboCup, simulation can be used to predict how a real NAO would interact with the SPL field without actually using a physical robot. More specifically, simulation can be used to observe how a NAO robot running rUNSWift plays soccer, without actually playing a physical game.

Developing software for robots introduces considerations that are not applicable to traditional desktop programming. These considerations negatively impact the effectiveness of engineers, and slow down the development cycle. Simulation is useful because it negates or reduces many of these difficulties. A few examples of the problems associated with robot development are: slower build cycles, hardware inconsistencies, difficulty debugging, and difficulty testing.

Slower build cycles are a symptom of embedded programming. In a typical development environment, the engineer will maintain a local copy of source code on their machine. When they would like to run their code on the embedded system, the developer will compile the code locally (often using target specific compilers) and upload it to the embedded system using some kind of file transfer protocol. This is quite inconvenient,

especially for big codebases such as rUNSWift. Fortunately, simulation allows for code to be maintained, compiled, and run locally.

Hardware inconsistencies are a side-effect of robot software development. As robots are made up of many moving and sensitive parts (especially sensors), there may be many minute inconsistencies between two robots of the same model. This can be exacerbated by wear and tear. An example of this is the camera placement on the NAOs. The NAOs feature two cameras, which are fixed on the head. However, the position and orientation of these cameras inside their fixtures deviate slightly from robot to robot [RLM+09] [RHH+10]. Code tested on one robot may act slightly differently on another given hardware inconsistencies. This is especially dangerous if a robot with (possibly unknown) significant hardware inconsistencies is being extensively used for development. Conversely, simulated robots are consistent. In some cases, hardware inconsistencies may intentionally be modelled in simulation, to increase the fidelity of the simulation.

Anoter issue is difficulty debugging. As embedded systems run software locally on their own hardware, it can sometimes be very difficult to debug uploaded binaries. In the case of the NAO V5, developers are able to use SSH (*Secure Shell*) to remotely connect to the robot and run an instance of GDB (*GNU Debugger*) to debug code. However, plain old GDB is hardly sufficient for debugging a codebase as big and complicated as rUNSWift. In 2009, UNSW Sydney developed *offnao*, a rUNSWift debugging tool that assists debugging over the local network [RHH+10]. Simulation allows for code to be run locally, which means developer-preferred debuggers can be used.

Difficulty testing is likely the most notable disadvantage of embedded programming. Embedded systems are designed to be run in the real world, and their software often relies on sensors and actuators that interact with their environment in real-time. A core concept of testing is being able to replicate the state of a system, so that given a certain state, the next state of the system can be tested to meet some requirement. The presence of real-time sensors often make this difficult to achieve. For example, consider testing the rUNSWift ball detector using a camera on the NAO. If the ball is

in-front of the camera, but it is not detected in a specific frame, it is extremely difficult (if not impossible) to naturally reproduce the exact same frame to test the detection again at a later point. This increases the difficulty of testing, especially qualitative testing. One of the main benefits of simulation is the ability to control the state of the simulated robot or its environment. This allows for reproducible game states whereby the developer can test the states following to meet some kind of criteria.

Another issue introduced by real-time sensors and actuators is their dependence on time. Consider the NAOs motion actuators. Say, for example, a developer wanted to test how long it takes for the robot to walk 100 laps around the field. To do this experiment on an actual Nao, the developer would have to wait for the robot to do 100 laps. The actuators are bounded by the speed at which they can move, relative to time. This increases the time it takes to test robot code, further increasing the difficulty of testing. As simulation controls the environment of the robot, it also controls the speed of time. This allows for faster or slower than real-time testing, which greatly improves the efficacy of using machine learning approaches to optimise components of rUNSWift.

From the short list of reasons given above, it can be seen that programming for embedded systems and robots is often obfuscated by the embedded and real-time nature of the targets. However, it is possible to negate many of these difficulties using *simulation*.

2.2.1 The RoboCup 3D League Simulator

The Standard Platform League (SPL), is but one of many different autonomous robotic challenges that are hosted by RoboCup. The RoboCup 3D Simulation League (3DL) is another popular challenge, where teams write their own software (known as *agents*) to control a virtual NAO in a three-dimensional environment.

RoboCup uses *SimSpark* as the official simulator of the 3DL [BDR+10]. SimSpark simulates two teams of NAO robots competing on a virtual soccer field. Teams connect to the simulator via TCP/IP by which they can perceive and actuate the virtual robots.

A monitor can be used to watch the simulation. SimSpark provides one, called *rcssmon-*

itor3d. However, the provided monitor is quite bland, and a third-party open-source monitor called *RoboViz* looks much more realistic.

2.2.2 Related work

Given the benefits of using simulation for robotic software development, it not surprising that the idea of using computer simulations to develop and test rUNSWift code is not entirely novel. Over the years, UNSW Sydney students and staff have investigated and built their own simulators.

In 2011, Yusmanthia researched the development of robot behavioural skills using a simulator [Yus11]. Yusmanthia wrote a 2D behavioural simulator in Python, using the PyBox2D physics engine. He then used the simulator to develop robot behaviours, namely the positioning of the Midfielder and Defender roles. Yusmanthia also researched using machine learning to optimise behaviours, but was unable to outperform intuitive approaches.

Yusmanthia found that using a simulator "speeds up the whole development process" of robot behaviours. However, he also found that the simulator's dependency on a consistent rUNSWift interface was problematic, and postulated this would present compatibility issues in the future.

Yusmanthia's work could be improved upon. Firstly, the physics of the simulator developed was quite divergent from the physical world. The simulated environment was only in two-dimensional space, and only the feet of the NAOs were modelled. A more effective approach would be to simulate the robots in three-dimensional space, and to model the entire NAO.

Secondly, Yusmanthia's simulator was written in Python. This meant that only Python components of rUNSWift, the behaviour module, could be simulated. Motion, localisation, kinematics, touch, sonar, vision, and networking could not be (natively) simulated. Furthermore, while Python is a powerful language with lots of functionality, it is often

considered a poor choice for real-time applications, due to its slow execution times in comparison with traditional programming languages.

A more suitable approach would be to write a simulator in C/C++, as it is well-known for fast execution times and is the native language of the rUNSWift codebase (except for behaviours which are written in Python [1]). In fact, Yusamthia even suggests that future work could include using a "very generic simulator that uses [a] faster language, like C or C++".

Thirdly, as noted by Yusmanthia, the simulator developed was "hugely dependent on the structure of the rUNSWift code to stay consistent". This is not ideal, as rUNSWift is in constant development and the interface is ever-changing. A more versatile approach would consist of a simulator that is flexible, and able to cope with rUNSWift changes.

In 2012, Tam researched the simulation of robot motion using the RoboCup 3D Simulation League (3DL) Simulator [Tam12]. The 3DL simulator is driven by *agents* which, given the sensory information and current state of a virtual NAO, actuate the joints to perform a desired action. Agents communicate with the simulator via a TCP connection, which allows developers to write agents using any programming language or platform, given it has TCP capabilities.

Tam used an open source and freely available C++ agent library *libbats* to develop an agent for research purposes. Using libbats, Tam wrote an agent which was able to read and actuate rUNSWift *'pos'* files on the virtual NAO. 'pos' files are text files that describe a number of static robot positions, represented as sets of joint positions. Each position is actuated in order, over a predetermined timeframe. The result of this is a static and predictable movement of the robot, that can be reused whenever it is applicable. An example of the application of 'pos' files are the rUNSWift front and back getups, which are used when the robot has fallen on it's front or back, respectively.

[1] Although the behaviours are executed using a Python interpreter which is "embedded" in to the C/C++ executable using *libpython* [RHH+10].

Tam's simulation code was able to successfully simulate 12 of 20 'pos' files from the rUNSWift codebase at the time of writing. Five of the 'pos' files that were unable to be simulated were inhibited by the fact that they rely on a different starting position to that of the simulator. The remaining three were most likely affected by the diversion of physics and modelling between the real world and the simulator. Tam found that simulators were, in some cases, able to simulate the rUNSWift 'pos' files. It follows that simulators could possibly be used to develop and test motion code, especially in cases where high precision is not necessary.

Tam's work could be extended. One might note that he only simulated the movement of joints using 'pos' files, and did so using a third-party agent. A more encapsulating simulator would simulate motion using rUNSWift code, preferably the entire rUNSWift *Motion* module. This would give a more realistic simulation of motion, instead of just 'pos' reading and actuating. Tam suggests that machine learning could be used to develop more sophisticated actions, such as walking and kicking.

Echoing the suggested future work of Yusmanthia (2011) and Tam (2012), a machine learning through simulation approach was taken by Hengst in 2013 [Hen13]. Hengst built a 3D simulator using the Open Dynamics Engine (ODE) to simulate NAO motion, and used reinforcement learning to optimise the transition between a finite number of robot motion states. These states included side-to-side rocking, standing still, and balancing on one leg. Additionally, Hengst used reinforcement learning to optimise disturbance response.

Hengst was able to learn a transition function between a finite set of states, and an adequate response to disturbance. Hengst found that machine learning through simulation was a "promising approach" for bipedal locomotion problems.

Simulation research by Hengst could be supplemented in a number of ways. While Hengst focused on solving locomotive problems, there would be many benefits of using similar techniques to solve perceptive or behavioural problems. Unfortunately, the tools that Hengst used would most likely not be appropriate for this task. The first reason for this is because the simulator that Hengst used was purpose-built for locomotive

development. As a consequence, it does not simulate the robotic sensors or the field used in the SPL. A more applicable simulation should include these features.

Secondly, the simulator used by Hengst does not simulate the rUNSWift codebase itself, but rather simulates a ported section of the motion code, transcribed from C++ to Java. For an accurate simulation, it is imperative to run the same code on the simulator as the physical robot being simulated.

Also in 2013, Crane & Sherratt researched rUNSWift behavioural simulation [CS13]. The pair attempted to build their own 2D simulator using Python with *Pygame* and *pybox2d*. Pygame is a popular game design library, and pybox2d is a popular physics library.

Crane & Sherratt proposed a simulator that could simulate the behaviours of a ported version of rUNSWift. Essentially, the simulator was designed to run modified rUNSWift Python behaviours, with additional modules developed to simulate perception and actuation. The perception and actuation modules would communicate with the simulator, acting as the physical robot and environment. However, the pair were unable to use the simulator to develop behaviours that could be tested on the physical NAOs.

Their work could be improved in a number of ways. As noted previously, a more accurate simulation would be in three dimensions, rather than two. Further, a versatile simulator should be able to simulate the entire rUNSWift codebase, not just specific modules. The simulator should require minimal changes to the codebase to be functional. Finally, for a simulator to be worthwhile, it must allow for the development of useful behaviours that can be run on the physical NAO without major changes.

In 2017, Hanna & Stone used Grounded Simulation Learning (GSL) to optimise the rUNSWift walk engine in simulation, and increase its speed on a physical NAO by over 43% [HS17]. It should be noted that while Hanna & Stone demonstrated an improvement of the walk from 1952mm/s to 2797mm/s, the initial speed of 1952mm/s is not the maximum walk speed of the unoptimised rUNSWift walk engine, which can be configured to walk at 3000mm/s without optimisation. Nevertheless, while the speed

of the optimised walk may not be record-breaking, the use of GSL to optimise the walk in simulation, and port it to a physical robot is certainly noteworthy.

Reinforcement learning through simulation is a convenient way of optimising robot code without using physical robots. However, the divergences between simulators and the real-world often result in optimisations that do not perform well in the real world. Grounded Simulation Learning (GSL) is a reinforcement learning framework that attempts to mitigate the inaccuracies of the simulator being used. By comparing the effect of robot code in simulation and real-life, the GSL framework is able to learn a subset of the divergences between the simulator and the physical world, and account for them. This results in optimised robot code which is more congruent in its performance in simulation and the real world.

Hanna & Stone optimised the rUNSWift walk engine by using a ported version with the UT Austin Villa Three-Dimensional League codebase. Once optimised, the code would have to be re-ported to a physical robot. To complement their work, GSL could be used to optimise native rUNSWift code, that does not have to be ported. Furthermore, while GSL was used to optimise the rUNSWift walk, it could also be used to optomise other robot code, such as behaviours or localisation.

2.3 Finding the Ball

As the SPL is a robotic soccer competition, it follows that finding the ball is quite an important task. If a team cannot find the ball, they cannot score. If a team cannot score, they cannot win.

The task of finding the ball can be broken up in to two sub-tasks: ball detection and ball searching. Both of these tasks must be completed satisfactorily to find the ball, as they depend on each other.

2.3.1 Ball Detection

Ball detection is the act of using robot sensors to detect a ball in an environment. In the context of SPL, the NAO V5 robots are equipped with two HD cameras, that are used to look for the ball on the SPL field.

A basic overview of rUNSWift ball detection is as follows. Each update cycle, *Perception* reads the latest NAO camera frames. Within the frames, computer vision algorithms are used to detect regions of interest. These are areas that could possibly contain a ball, as they meet some kind of criteria that suggests this. Within the regions of interest, computer vision algorithms are used to try and detect the ball.

In previous years, the SPL ball was bright orange in colour, and thus was easy to detect in images at the pixel level. However, 2016 saw the introduction of the black and white ball [MSS+16], which significantly increased the difficulty of ball detection. The black and white ball is difficult to detect as many objects on and around the SPL field are comprised of white (lines, goal posts, robots), which greatly increases the amount of noise in detections. This has reduced the reliable ball detection distance of rUNSWift in recent years.

2.3.2 Ball Searching

Ball searching is the intelligent placement of robot sensors to effectively sense the ball, using ball detection, in a timely fashion. In the case of SPL, this means actuating the NAO in a way that has the maximum probability of detecting the ball using the cameras, as fast as possible.

The rUNSWift 2016 ball searching component is a Python behaviour called *FindBall*. Upon losing the ball, FindBall instructs each robot to walk through a hard-coded set of points on the field, in search of the ball. The point that is closest to the last known ball position is selected first, and the remaining points are repeatedly visited in order. Upon finding the ball, the search is terminated and play proceeds as normal. The hard-coded

points that the 2016 ball searching component visits are described in Table 2.1. The points are visualised on an SPL field in Figure 2.7.

Number	X Coordinate	Y Coordinate
1	-3000	0
2	-2000	2000
3	0	1000)
4	2000	2000
5	3000	0
6	2000	-2000
7	0	-1000
8	-2000	-2000

Table 2.1: 2016 ball searching points

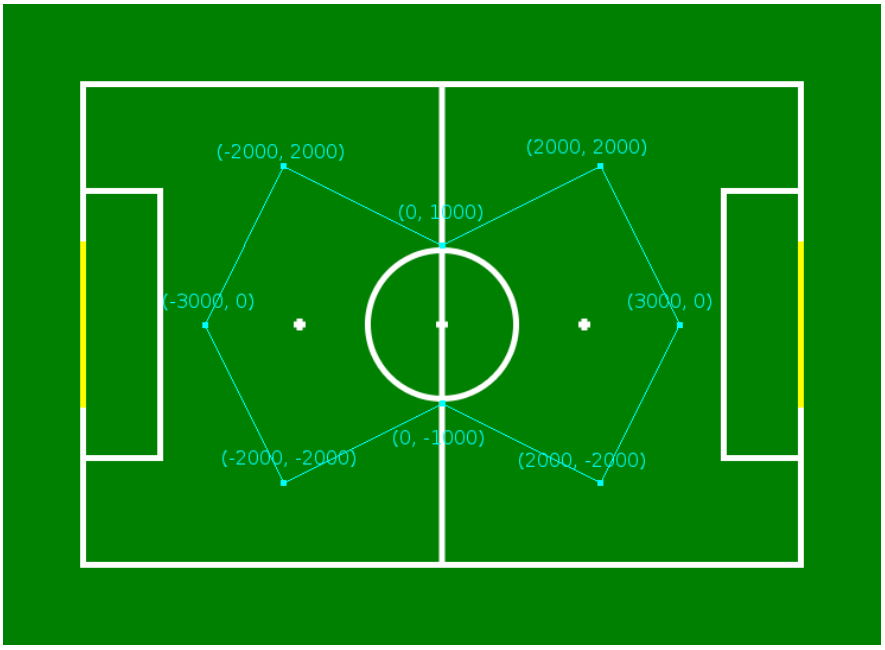

Figure 2.7: The static search locations for the 2016 FindBall behaviour, described in Table 2.1

However, it could be argued that finding the ball using hard-coded locations is not necessarily optimal, and difficult to maintain. For a set of search points to be effective, any legal ball must be observable from at least one of the locations (or during transit between them). Otherwise, it is possible to walk between every single search point without detecting the ball, rendering the behaviour ineffective.

For any legal ball to be observable from a set of points, the union of visible area from

each point in the set (and the routes between them) should at least contain the SPL field. In this context, the *visible area* of a search location is the surrounding ground that is within the estimated reliable range of the ball detector. It follows that the visible area from a search location is directly, and unconditionally dependent on the reliable detection range of the ball detector in use at the time of execution. Therefore, the union of visible area, and thus the overall effectiveness of the set of points is dictated by the reliable detection range of the ball detector.

The *reliable detection distance* of a detector is the expected detection distance of the detector for average balls in the majority of regulation environments. The reliable detection distance of a detector can be estimated by taking the *local detection distance* of a detector in a laboratory environment and reducing it by some amount to account for unknown and untested environments.

The *local detection distance* of a ball detector is the maximum distance that the detector can reliably detect an average ball in a specific environment. In this context, *reliably* detecting a ball means observing it consistently, over numerous frames. Lighting conditions, ball type, field colours, and field surroundings all affect the reliable detection distance of a detector.

As the effectiveness of a set of search points is dependent on the reliable ball detection distance, it follows that the performance of a ball searching component using a set of hard-coded points is unlikely to perform optimally. A static behaviour may perform well at the time of initial authorship as it will be implemented with search points that were optimised for the ball detector at the time of writing. However, as the ball detector and its environment changes, so will its reliable detection distance and thus the performance of the static behaviour will slowly diverge from optimal.

The 2016 FindBall behaviour is a good example of this. Figure 2.8 illustrates the position of the static points used in the behaviour, and the surrounding corners of the field. As mentioned earlier, for a set of static search points to be effective, the union of their search area must at least contain the SPL field. Figure 2.9 illustrates the distance from the field edge to the closest search point.

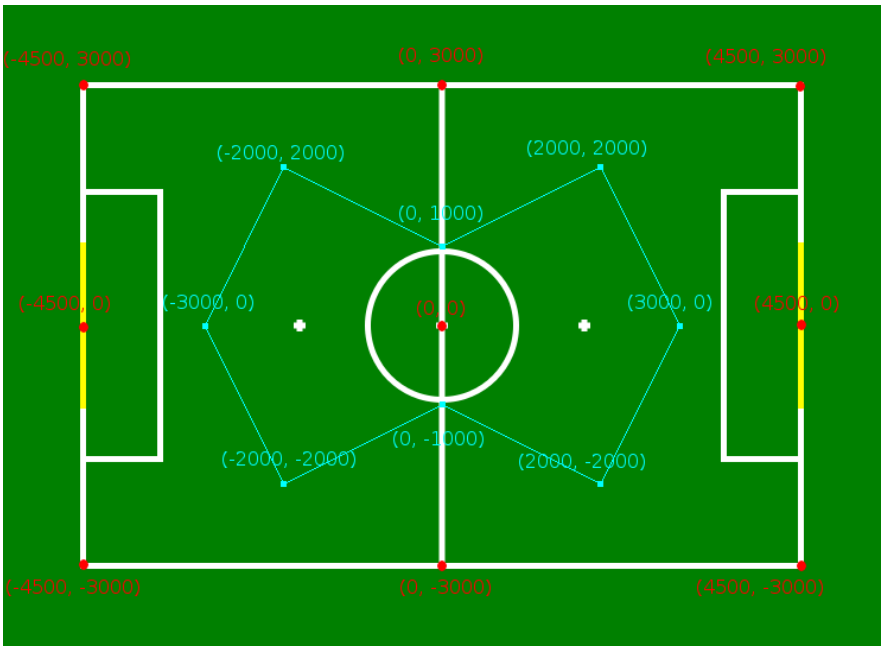

Figure 2.8: The static search locations for the 2016 FindBall behaviour, with the surrounding corners of the field annotated. Based on 2.7.

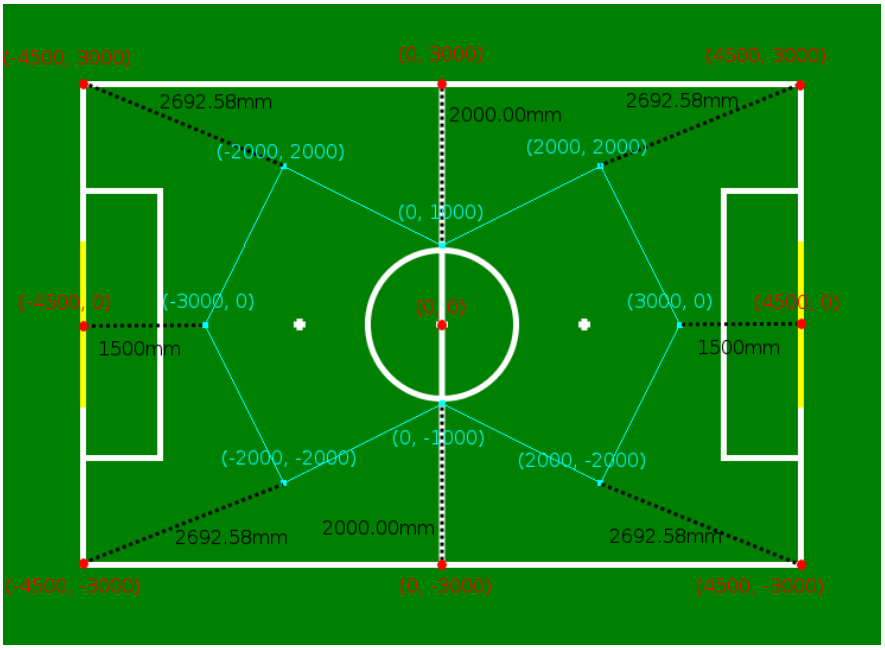

Figure 2.9: The static search locations for the 2016 FindBall behaviour, and their distances to surrounding edges of the field. Based on 2.8.

As can be seen in Figure 2.9, the furthest distance from any coordinate on the field to its closest search point is found on each corner, at 2692.58mm. Taking in to account the

SPL 2017 line width of 50mm [Com17], the length from the corner of the field to the edge of the corner can be calculated as the hypotenuse of a triangle with two equal sides of 50mm, which is $\sqrt{5000}$mm. Therefore, a legal ball may be placed at most $2692.58 + \sqrt{5000} = 2763.29$mm from the set of search locations. It follows that, for the visible area of the static set of points in the 2016 FindBall behaviour to cover the SPL field, the reliable ball detection distance of the ball detector being used should be equal or greater than 2763.29mm. If this is true, then theoretically the 2016 FindBall behaviour is able to reliably search the entire SPL field. Figure 2.10 shows the theoretical maximum coverage of static search points in the 2016 FindBall behaviour, with a reliable ball detection distance of 2764mm.

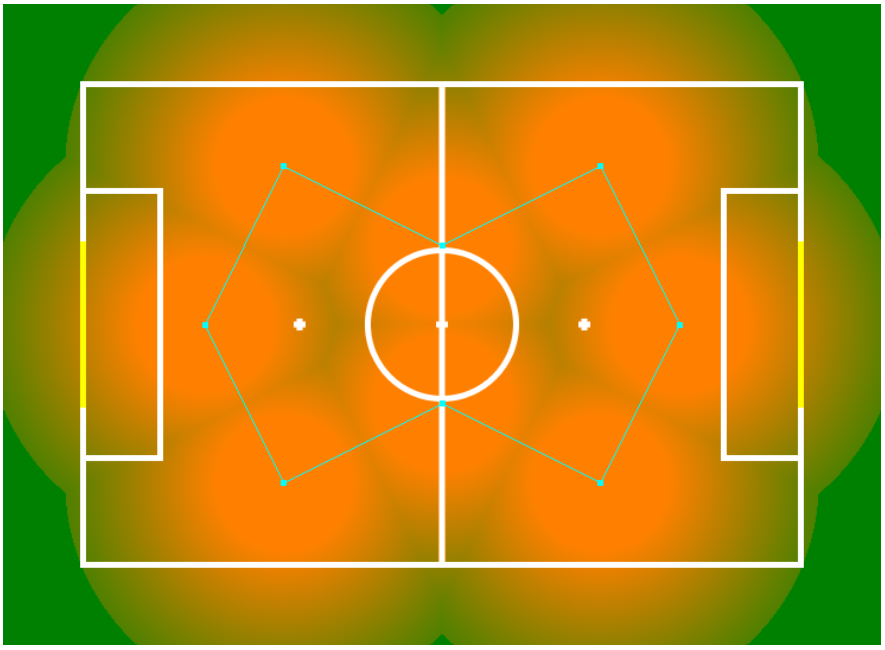

Figure 2.10: The theoretical maximum coverage of the static search points in the 2016 FindBall behaviour, with an assumed reliable ball detection distance of 2764mm.

Note that the maximum coverage does not take in to account field-of-view restrictions of the robot, assuming that they can perceive 360 degrees. As this is not the case in real life, the actual coverage during play is likely to be less than the theoretical maximum. The longer a robot patrols a set of points, the actual coverage will converge upon the maximum coverage. Also note that this coverage does not include the traversal between each static point, as this is behaviour is dynamic. Assuming that robots travel in a

straight line between search points, the estimated complete maximum coverage of the static set of points is depicted in Figure 2.11.

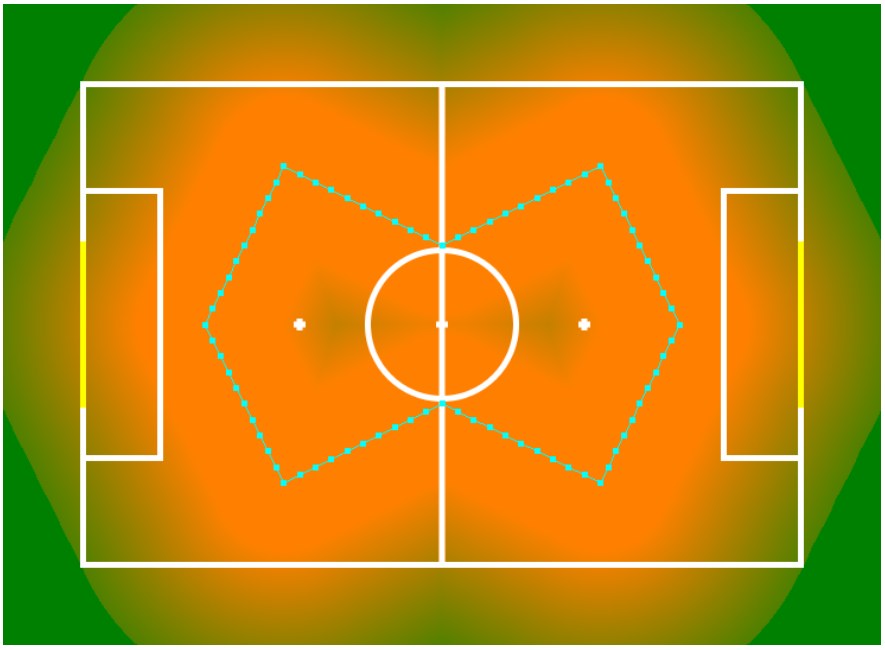

Figure 2.11: The theoretical complete maximum coverage of the static search points in the 2016 FindBall behaviour, assuming that robots travel between points in a straight line. Also with an assumed reliable ball detection distance of 2764mm.

The original FindBall behaviour was written when the orange ball detector was in use. This ball detector could reliably detect the orange ball up to 6 meters away [Cha11], which is equivalent to the width of the competition field. It follows that the original hard-coded search positions were selected for use in conjunction with a ball detector that has a similar reliable detection range.

Unfortunately, at the time of writing, the reliable ball detection distance for the black and white ball detector is less than 2763.29mm. During testing, a robot running rUNSWift could detect the black and white ball reliably at around 2500mm in the UNSW Robolab. In reality, the true reliable ball detection distance would most likely be less than this, as the UNSW Robolab is quite well lit compared to most regulation SPL fields. However, 2500mm is an acceptable heuristic.

With an estimated reliable ball detection distance of 2500mm, the maximum coverage

of the 2016 FindBall behaviour does not contain the entire SPL field. As depicted in Figure 2.12, the static set of hard-coded points do not cover the corners of the field.

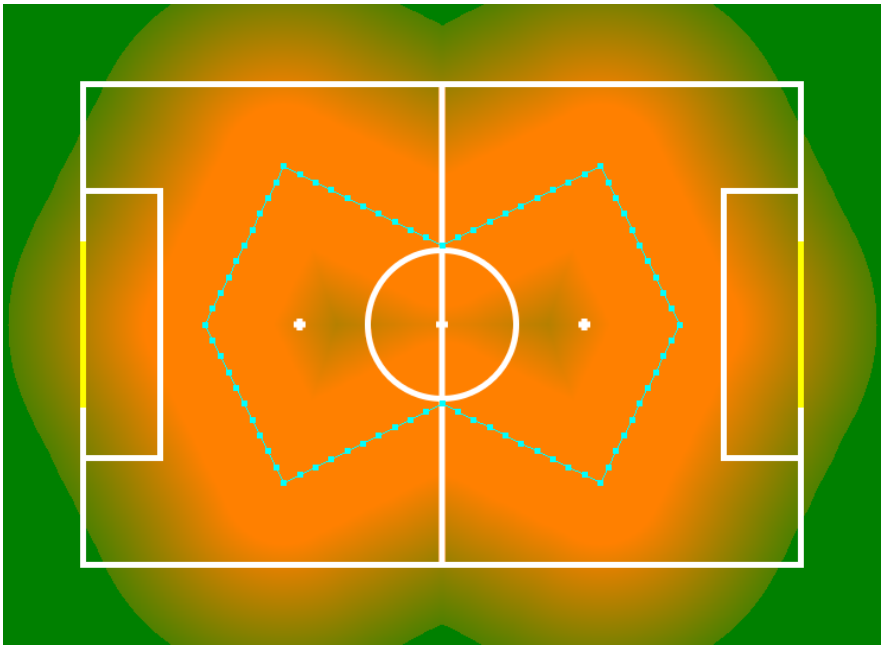

Figure 2.12: The theoretical complete coverage of the 2016 FindBall behaviour with a reliable ball detection distance of 2500mm.

Furthermore, during 2016, the reliable ball detection distance for the black and white ball detector was much less. The performance of the detector is not documented, but from the recordings of games at the competition in Leipzig, it appears to be less than 1000mm. Figure 2.13 shows a screencap from a game between rUNSWift and Berlin United, where a rUNSWift robot is unable to detect a ball that is much closer than 1000mm. Even if the reliable ball detection distance was 1000mm, the field coverage of the 2016 FindBall behaviour is quite poor. The majority of the field is not covered. This is depicted in Figure 2.14.

A more robust ball searching component would take a different approach. Instead of hard-coded points, it would be more effective to use a minimal dynamic set of search locations that are generated to maximise field coverage, given the current reliable distance of the ball detector in use. Ideally, each search location should have a mutually exclusive visible area, so that the disjunctive union of visible area from the set of search points covers the field. However, as it is difficult to correctly estimate the reliable

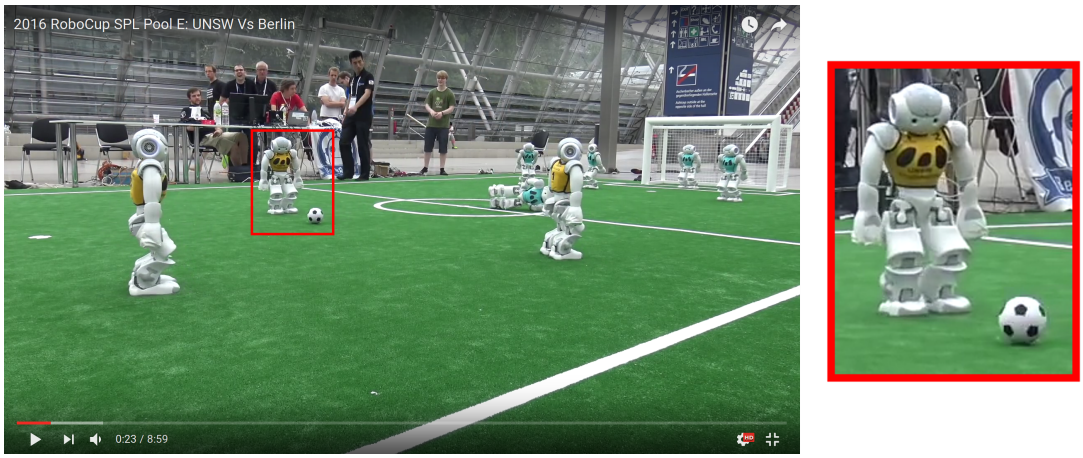

Figure 2.13: A screencap of a game between rUNSWift and Berlin United at RoboCup in Leipzig, 2016. A rUNSWift robot (in yellow) is depicted standing next to the ball but is unable to detect it. The left eye of robots running rUNSWift turn red when they are able to see the ball. Here it has no colour, indicating no ball has been seen.

distance of a ball detector, it would suffice for the search locations to have partially overlapping visible areas to account for errors in estimation.

Furthermore, the sensor actuation process of the 2016 FindBall behaviour is questionable. The behaviour patrols the hard-coded points in a pre-defined order, from a dynamically selected starting point. When the ball is lost, the behaviour selects the static point that is closest to the last known ball position. Upon arriving at the selected point, the robot visits the remaining points in clockwise order.

It could be argued that this is folly, as except for the initial point selection, the visiting order and direction of the patrol is unchanging. Consider the case where a robot on the right-side of the field selects an initial point on the left-side of the field. If the robot fails to find the ball, they will walk straight back to the points on the right side of the field. Another issue with a pre-defined route is the prospect of robot cramping and collisions. Robots that are on the same route are likely to impede each other, or even collide. This slows down the ball finding process, and increases the entropy of robot world models.

A more efficient behaviour would walk to the points in an order that optimises the time to find the ball. The points could be ranked by some kind of criteria that dictates their

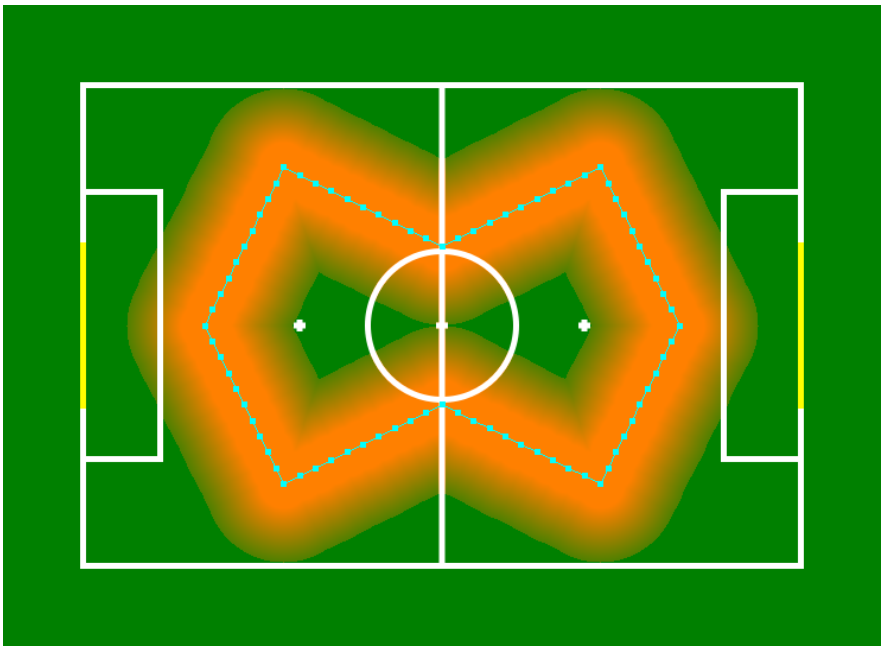

Figure 2.14: The theoretical complete coverage of the 2016 FindBall behaviour with a reliable ball detection distance of 1000mm, which is likely greater than the ability of the detector at RoboCup in Leipzig, 2016.

chance of helping the robot to find the ball. Points that have recently been visited should be selected after other points.

2.3.3 Related work

Sarmiento et al. (2003) published research outlying an efficient method of searching for objects using robots [SMH03]. Given a set of locations L, where the union of their visibility regions is equal the the environment W, they define a method of selecting the order in which each location in L should be visited, minimising the average time it would take to detect the target object. As calculating the optimal route is demonstrated to be NP-hard by reduction, Sarmiento et al. present a heuristic that can be used to provide an estimation of the optimal route at comparatively low cost.

In the case of RoboCup SPL, L would be a set of equidistant points that cover the SPL field, W. The distance between each point in L would be the distance at which the ball (or the object being searched for) can be reliably be detected.

The method presented by Sarmiento et al. relies on the assumption that, within the environment being searched, the probability of the object being detected from each point is not uniform, and as such there is a higher chance of detecting the object at some points instead of others. In such a case, it would make sense to check these locations before those with a lower chance of detection.

In the case of RoboCup, the probability distribution of the location of the ball may initially be uniform, but this does not hold true as more information about the environment is perceived. This is especially so if complimentary data from teammates is considered too.

Sarmiento et al. also note that the distance of each point from the robot also affects their usefulness. In general, the further a search location is from the robot, the lower its usefulness should be. Furthermore, the closer a search location is to the robot, the higher its usefulness should be.

Sarmiento et al. present a heuristic to estimate the usefulness of a search location, which they refer to as *utility*. They define the utility of a search location as ratio of the probability of object detection divided by the distance of the search location. This can be generalised as:

$$U(L_j, L_k) = \frac{P(L_k)}{Time(L_j, L_k)}$$

Where $U(L_j, L_k)$ is the utility of moving to search location L_k from position L_j.

The paper goes on to discuss the use of a tree to find partial paths. Then, the robot can select the partial path with the highest utility, instead of a single point.

Sarmiento et al. tested their path finding method in simulation, with good results. In comparison with the optimal solution, the search paths found using their utility heuristic performed only slightly worse, but were calculated magnitudes faster.

The work by Sarmiento could by supplemented by testing their heuristic on an ac-

tual robot, finding an actual object. For example, the utility heuristic presented by Sarmiento et al. could be used on the NAO V5 to find the ball in SPL.

Chapter 3

Simulation using *simswift*

3.1 Overview

simswift is a new build target of the *rUNSWift* codebase that runs on a Linux PC and controls a virtual NAO on the RoboCup 3D League (3DL) Simulator. More specifically, simswift is a rUNSWift build target with preprocessor definitions that disables components that interface with NAO hardware and replaces them with components that interface with the 3DL Simulator instead. To be clear, simswift itself is not a simulator. The 3DL Simulator is the software which handles the physical simulation of NAOs and the SPL environment. simswift is the simulator-friendly build target of the rUNSWift codebase, that is able to interact with the 3DL Simulator. Figure 3.1 shows five copies of *simswift* (being run on one machine) simulating a team of robots, as visualised by *RoboViz*.

Instead of interfacing with a NAO via libagent, simswift interacts with the 3DL Simulator using the Simulation module. If activated, the Simulation module and 3DL Simulator effectively replace the physical NAO and its environment in the system architecture. Instead of receiving sensory information from libagent, simswift receives sensory information from the Simulation module. Instead of sending joint commands to libagent, they are sent to the Simulation module. In turn, the Simulation mod-

Figure 3.1: Five copies of simswift running simultaneously simulating a team of robots, as visualised by *RoboViz*.

ule receives sensory information from the 3DL Simulator via TCP connection, and in response, sends the joint commands to be actuated on the virtual NAO.

For compatibility, the Simulation module provides the same interface and exhibits the same behaviour that libagent does. That is, the Simulation module writes sensory information to the shared memory at the same location and in the same format as libagent. Further, the Simulation reads joint commands from the same location in shared memory and in the same format as libagent. This means that the rUNSWift code that interacts with libagent remains unchanged. In essence, the libagent or Simulation modules are interchangeable and are selected depending on the build target being compiled (rUNSWift or simswift). Figure 3.2 illustrates the system architecture of a simswift build.

Each update cycle, the workflow of the Simulation module is as follows. Firstly, the latest sensory information is received from the 3DL Simulator, over a TCP connection. The sensory information is processed, and converted to a format that rUNSWift understands. The transformed sensory information is written to shared memory, where it is

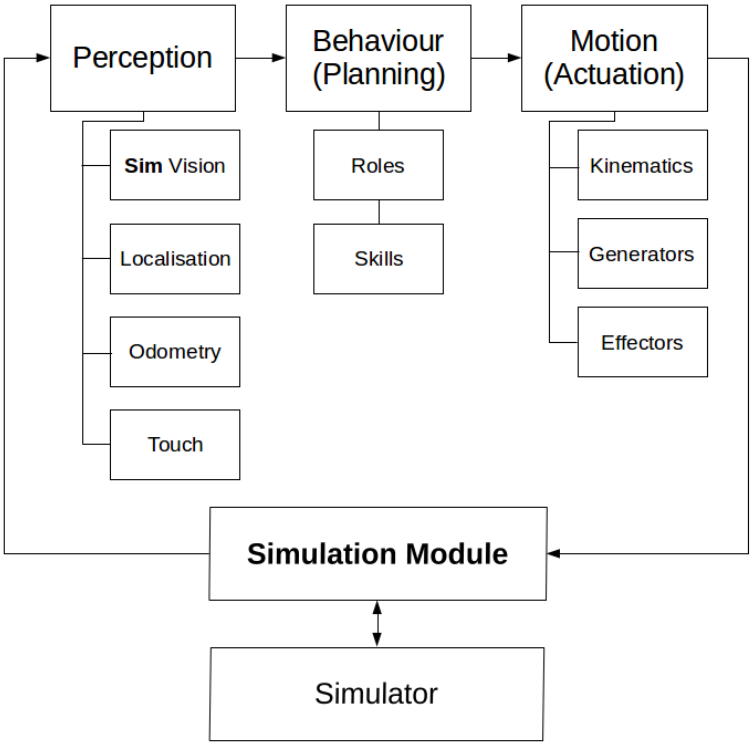

Figure 3.2: The simswift architecture, based on the rUNSWift architecture seen in figure 2.2. Note that the *Vision* component of Perception has been changed to a simulated variant.

read by Perception. The Simulation module then waits for joint angle commands from Motion. Once they are received, the Simulation module converts them to a format that the 3DL Simulator understands. Then, the translated joint commands are sent to the 3DL Simulator, where they are actuated on the virtual NAO. The cycle continues until simswift is killed or the simulation finishes.

For a simulation to be accurate, the divergence between simswift and rUNSWfit code must be minimal. As such, the majority of code differences introduced by simswift are confined to hardware-facing components. Higher-level components and behaviours are left essentially unchanged. The consequence of this is that simswift behaves (approximately) the same way on a virtual Nao as rUNSWift behaves on a physical Nao.

Careful consideration was made to keep simswif and rUNSWift in the same codebase.

The result of this is a multi-target codebase that can build either rUNSWift or simswift from a common set of source files plus a small set of target specific source files. This means that simswift and rUNSWift can be built interchangeably from the same source, and any changes made to their common source will affect both targets.

3.2 Simulation module

The Simulation module is the interface between simswift and the 3DL Simulator. It is made up of various C/C++ components that perform different tasks. These components are *SimulationThread*, *SimulationConnection*, *AngleSensor*, *SonarSensor*, and *SimVisionAdapter*. The interactions between these components and neighbouring modules is shown in Figure 3.3.

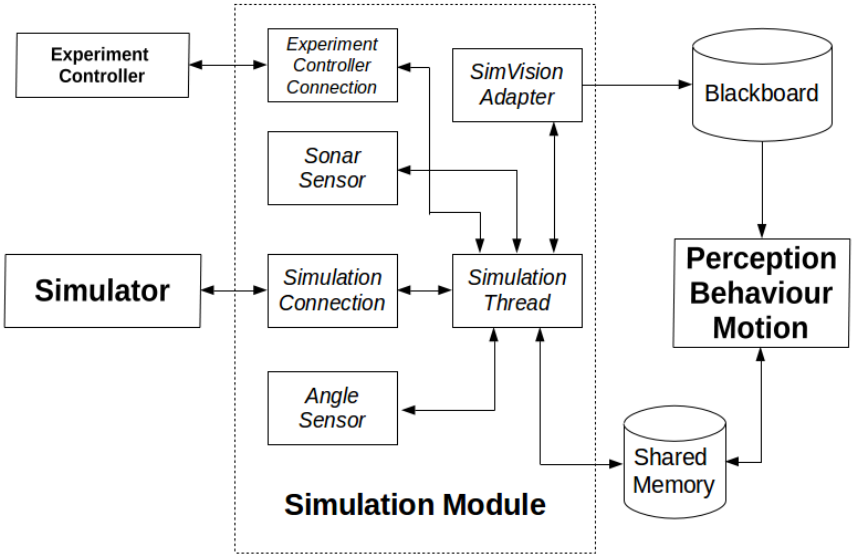

Figure 3.3: The Simulation Module architecture, showing the interactions between components and neighbouring modules. One level more detailed than Figure 3.2.

3.2.1 Where it comes together: *SimulationThread*

SimulationThread is the highest level of the Simulation Module. The SimulationThread contains and controls all other components of the Simulation Module. Simulation-

Thread is, as the name suggests, run on a separate thread, instantiated from the simswift entry point, where other thread-based modules are started. The workflow described in Section 3.1, at its highest level, is undertaken here.

Each tick, SimulationThread receives the latest perceptor information from *SimulationConnection*. Firstly, the *SimVisionAdapter* class is provided with information from the visual perceptors on the virtual robot. The information from the visual sensors is converted to collections of field features that have been observed, which are provided to the Perception module through the Blackboard.

The non-visual sensory information is then converted to a format consumable by Perception and Motion. This is a careful process, as the simulator makes abstractions and assumptions about the sensors and actuators. In many cases, the interface and behaviour of actuators on the virutal NAO is divergent from the physical NAO which it is modelled on. This is discussed in depth in section 3.3.1.

SimulationThread then provides perceptor information to *AngleSensor*, and *SonarSensor*, which simulate the X/Y angle and sonar sensors, as the 3DL Simulator does not simulate them itself.

Next, SimulationThread reads the current X/Y angle and sonar information estimated by the simulated AngleSensor and SonarSensor classes. This information, along with the Perception and Motion friendly joint angle readings, are then written to shared memory, so that neighbouring modules can process the information.

If *simswift* is connected to an experiment through the *ExperimentControllerConnection* class, then it sends the experiment controller information about the virtual robot, such as whether or not it can see the ball, and the estimated distance from the ball. Any commands from the experiment controller are also processed, such as penalise or game state commands.

The SimulationThread then reads the joint angle commands from the Motion module. These angles are converted back to radians, with the inverted joints being flipped again. The joint angles are then converted from an absolute angle to a motor velocity, as this

is the interface provided by the 3DL Simulator.

Once the joint commands have been converted to the specifications of the 3DL Simulator, the SimulationThread sends them to the simulator using the SimulationConnection class.

3.2.2 Connecting the 3DL Simulator: *SimulationConnection*

The *SimulationConnection* class handles communication between simswift and the 3DL Simulator. The 3DL Simulator provides a TCP/IP interface, which allows for platform agnostic agents to connect and control a virtual NAO. In this context, simswift is a C/C++ (and Python) agent.

Information is exchanged between the 3DL Simulator and simswift using a simple text protocol. Each message is proceeded by a 32-bit unsigned integer (in network order), and is made up of a string of field value pairings, delimited by parentheses. Values may contain their own field value pairings, which represent objects.

When SimulationConnection receives a packet from the server, it parses the text and stores the relevant information in a *PerceptorInfo* instance, which is consumable by other Simulation module components.

Similarly, SimulationConnection is passed information to send to the server in a *EffectorCommand* instance. This is converted back to the 3DL text protocol, and sent to the simulator.

3.2.3 Simulating vision: *SimVisionAdapter*

The *SimVisionAdapter* class, given the latest visual sensory information from the 3DL Simulator, will convert this in to a format that Perception can understand.

The 3DL Simulator provides the distance, horizontal angle, and vertical angle of simulated visual observations. Objects are detected when they are within a certain distance

and field-of-view of the virtual robot's camera. The simulator also provides the type of observation detected, being either a line, goal post, ball, or robot.

Using this information, SimVisionAdapter is able to transform each simulator observation in to a field feature understood by the Perception module. The distance, horizontal angle, and vertical angle are converted to a robot relative coordinate, and the field feature type is derived from the type of observation provided by the simulator.

Special care was taken to make sure that the distance and heading of the field feature observations are correct. The distance provided by the simulator is measured from the camera of the NAO, not from the centre between both feet as is expected by robot relative coordinates [RHH+10]. Given the height of the NAO [Ald], Pythagorean theorem is used to correctly convert the distance from the head of the virtual NAO to the distance from the feet of the virtual NAO.

SimVisionAdapter replaces the rUNSWift Vision component in simswift builds. As rUNSWift Vision relies on interfacing with the NAO camera, it is not suitable for use in a simulation build. As SimVisionAdapter maintains the same interface and behaviour as the rUNSWift Vision component, they can be used interchangeably without affecting neighbouring components.

3.2.4 Simulating the X/Y angle: *AngleSensor*

The *AngleSensor* class provides a simulation of the NAO X/Y angle sensor, as it is not provided by the 3DL Simulator. The X/Y angle sensor is used to detect if the robot has fallen over, and if so, which way it has fallen. If a fall is detected, Motion will attempt to get the robot back on its feet, using a *getup*.

The AngleSensor assumes an initial standing pose, and uses the simulated gyroscope and accelerometer to update this estimation as the robot moves. Gyroscopes are useful for measuring rotational change, but they drift over time. Fortunately, accelerometers can be used to compliment the data of the gyroscope, which reduces drift. This is known as a *complimentary filter*.

Each tick, the AngleSensor is provided the latest readings from the gyroscope and accelerometer. These are used in combination to update the X/Y angle estimation of the AngleSensor. The angle estimations are then read by SimulationThread, which passes them on to Perception.

3.2.5 Simulating the sonars: *SonarSensor*

Unfortunately, the 3DL Simulator does not simulate the sonar sensors that are present on the physical NAOs. As such, the SonarSensor class simulates sonar sensors itself, using the visual perceptor information provided by the 3DL Simulator.

According to the documentation, NAO V5 robots are equipped with two sets of sonars, that have a reliable detection range from 20cm to 80cm. [Ald] The sonars detect the distance of objects that are within this range and within a 60 degree field of detection.

The SonarSensor attempts to simulate this by deriving sonar detections from visual perceptions that meet the requirements above. Each tick, the SonarSensor receives sensory information from SimulationThread. Visual perceptions are iterated, and those that are within a distance and heading threshold are added as a sonar detection.

3.2.6 Running an experiment: *ExperimentControllerInterface*

The *ExperimentControllerInterface* provides a TCP interface for an experiment controller to communicate with simswift. An experiment controller manages a group of simswift processes and the simulator server to orchestrate some kind of process, such as an experiment.

For the purpose of evaluating *simswift*, a find ball experiment controller was developed, termed *findballexp*. findballexp moves connected agents to starting positions on the field and puts them in a penalised state. Then, findballexp moves the ball to a location on the field, unpenalises the robots, and times how long it takes for the robots to find the ball. This is logged, and repeated indefinitely.

3.3 Divergences between the simulator and SPL

3.3.1 The NAO

There are many differences between the virtual NAO, and the real NAO it is attempting to model. This includes both the physical modelling of the NAO, the virtual sensors that are simulated, and their exposed interfaces. Unfortunately, these deviations are undocumented, so they were only discovered through much trial and error.

It is possible that the simulator is simulating an older version of the NAO, which would result in some divergence from the NAO V5. However, the documentation does not state which version of NAO is being simulated, and the simulated NAO has mismatches across all versions of the physical NAO.

3.3.1.1 Sensors

All of the virtual sensors on the simulated NAO have divergences from those on the physical NAO V5. In most cases, at least the interface to the sensors differ to that on the physical NAO. However, in worse cases, sensors that exist on the NAO V5 are not even simulated on the virtual NAO.

The most obvious difference between the physical NAO V5 and its simulated counterpart is the vision system. The NAO V5 has two high-definition cameras, through which the NAOqi API provides 1280 x 960 images at 30 frames per second [Ald]. Meanwhile, the simulated visual perceptor interface does not provide any images, instead opting for a text-based description of *detected* objects. The real NAO V5 cameras have a horizontal field of view of 60.9°, whereas the simulated vision perceptor has a horizontal field of view of 120°[BDR+10]. To account for this, simswift discards balls that are outside a 60.9° field of view.

A frame based vision API, similar to the physical NAO V5, would be difficult to simulate realistically. Furthermore, simulated camera frames would be unlikely to work

effectively with existing computer vision infrastructure, developed to process images of real-life environments. As such, the developers' choice to instead provide a text-based API is understandable. However, the capabilities of the "smart image processing software" [BDR+10] simulated on the virtual NAO are far superior to the vision processing of rUNSWift, and most likely other teams in the SPL.

The simulated system can seemingly detect objects from any distance on the simulated SPL field. This is likely because the 3DL Simulator was designed with the larger 3DL field in mind, as opposed to the smaller SPL field. In comparison, rUNSWift requires much closer distances to reliably detect visual objects. For example, at the time of writing, rUNSWift can reliably detect the ball from, at most, approximately 2500mm distance. As such, simswift discards balls that are detected past 2500mm.

Furthermore, the simulated vision system has a very high detection accuracy, and is not susceptible to false positives or false negatives. Fortunately, the position of detected objects is affected by noise, but this is minimal. Table 3.1 illustrates the noise of perceptions from the simulated vision system.

Component	Noise
Camera position	Between -0.005 and 0.005m (calculated once per robot)
Object distance	Distributed around 0 where sigma = 0.0965
Horizontal angle	Distributed around 0 where sigma = 0.1225
Vertical angle	Distributed around 0 where sigma = 0.1480

Table 3.1: Noise in 3DS Simulator visual observations [BDR+10]

In comparison, the rUNSWift vision system is nowhere near as accurate. False positive detections are common, especially in changing lighting conditions and environments with white and/or green backgrounds. Furthermore, the estimated position of detections is also subject to more noise. This has not been handled in simswift.

The NAO V5 is equipped with a three-axis gyroscope, which measures rotational movement in radians. The simulated gyroscope measures rotational movement in degrees, the labelling and positioning of the axes does not correlate with that on the physical NAO. The 'X' axis on the simulated gyroscope actually represents the 'Y' axis on the physical NAO, whereas the 'Y' axis on the simulated gyroscope actually represents the

'X' axis on the physical NAO.

Furthermore, the direction of the axes labelled 'X' and 'Z' on the virtual gyroscope are inverted in comparison with their counter parts, 'Y' and 'Z' on the physical NAO. This is due to a different positioning of the gyroscope on the virtual NAO in comparison with the physical NAO. 3.4 illustrates the divergence between the simulated and real gyroscopes. simswift handles this by swapping the axes and directions to match the NAO.

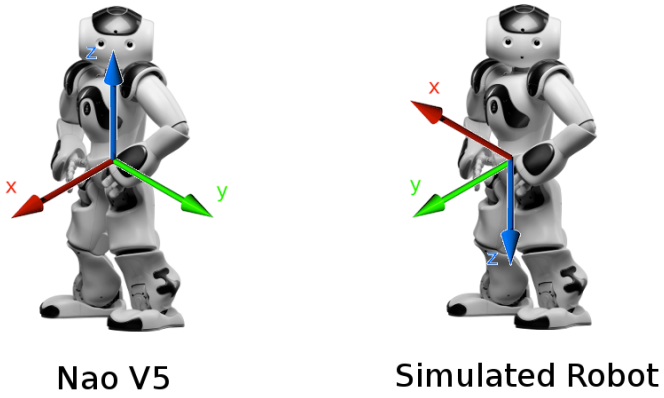

Figure 3.4: The difference between the gyroscope axes on the NAO V5 and the simulated NAO.

The NAO V5 is also equipped with a three-axis accelerometer, which measures proper acceleration. Again, the position and labelling of the axes on the virtual accelerometer does not match the NAO V5. It appears that the 'X' and 'Y' axes on the virtual accelerometer correspond to the 'Y' and 'X' axes on the NAO V5 accelerometer. Furthermore, the 'Y' and 'Z' axes on the accelerometer appear to be inverted in comparison to their physical counterparts, the 'X' and 'Z' axes. simswift also swaps the accelerometer axes and directions to match the NAO.

The physical NAO V5 is outfitted with an X/Y/Z angle sensor, which measures the robots orientation in three-dimensional space. However, the simulator does not simulate any angle sensors on the virtual Nao. simswift uses gyroscope and accelerometer

readings to calculate the X/Y angles instead. As the Z angle is not used by other modules, it is not calculated. See section 3.2.4 for more information.

The NAO V5 boasts two sets of sonars, each set made up of a sending and receiving unit. However, the simulator does not simulate any sonar sensors on the virtual NAO. Instead, simswift uses visual perceptions to simulate the sonars itself. See section 3.2.5 for more information.

3.3.1.2 Joints

Both the joints of the virtual NAO and the interface to perceive and actuate them are divergent from the real NAO V5. The physical NAO V5 has 26 hinge-joints, with 25 degrees of freedom. [Ald]. Each joint has its own degree of freedom, except the *LHipYawPitch* and the *RHipYawPitch*, which are connected to the same motor and share one degree of freedom.

However, the simulated NAO only has 22 hinge-joints, with 22 degrees of freedom. The simulated NAO does not model the joints in the wrists (*LWristYaw*, *RWristYaw*) and hands (*LHand*, *RHand*) of the NAO V5. Furthermore, the simulated NAO does not fuse the LeftHipYawPitch and RightHipYawPitch, meaning they both have their own degree of freedom.

The joints modelled in the simulator are also named differently to the joints of the NAO V5, making the mapping of interfaces difficult. Table 3.2 shows the mapping of joints from the API of the simulated NAO to the API of the physical NAO V5. *N/A* is an indication that the joint is not available.

Furthermore, it was discovered that nine joints on the virtual NAO were inverted compared to their physical equivalent. Actuating an inverted joint on both the simulated and physical NAO with the same angle command would result in the two joints moving to directly opposite positions. The joints that were found to be inverted on the virtual NAO are:

NAO V5	**Virtual NAO**
HeadYaw	NeckYaw
HeadPitch	NeckPitch
LShoulderPitch	LShoulderPitch
LShoulderRoll	LShoulderYaw
LElbowYaw	LArmRoll
LElbowRoll	LArmYaw
LWristYaw	*N/A*
LHand	*N/A*
LHipYawPitch	LHipYawPitch
LHipRoll	LHipRoll
LHipPitch	LHipPitch
LKneePitch	LKneePitch
LAnklePitch	LFootPitch
LAnkeRoll	LFootRoll
N/A	RHipYawPitch
RHipRoll	RHipRoll
RHipPitch	RHipPitch
RKneePitch	RKneePitch
RAnklePitch	RFootPitch
RAnkleRoll	RFootRoll
RShoulderPitch	RShoulderPitch
RShoulderRoll	RShoulderYaw
RElbowYaw	RArmRoll
RElbowRoll	RArmYaw
RWristYaw	*N/A*
RHand	*N/A*

Table 3.2: Joint mapping from physical NAO V5 interface to simulated NAO interface

- HeadPitch
- LShoulderPitch
- LHipPitch
- LKneePitch
- LAnklePitch
- RHipPitch
- RKneePitch
- RAnklePitch

- RShoulderPitch

To handle this, actuation commands for inverted joints are converted to their additive inverse before they are sent. Furthermore, joint angles that are perceived from inverted joints are also converted to their additive inverse before they are consumed by rUNSWift.

Not only are the modelling of the joints divergent from the physical NAO, but so is the interface to perceive and actuate them. The documentation for the NAO interface, NAOqi, states that the joint positions are read in radians. [Ald]. However, on the 3DL Simulator, joint angles read from the virtual perceptors are in degrees.

Furthermore, the joints of the physical NAO are actuated by supplying an absolute joint angle. However, on the 3DL Simulator, joints are actuated by supplying a relative angle, or motor velocity. This velocity will be maintained until a different velocity is supplied.

This makes actuation to an absolute target angle difficult. The problem is exacerbated by the fact that joint effector commands are delayed by one tick. If a joint command was issued during tick 1, the perceived joint angle does not change until tick 3. This means that by the time the effector command is actuated, the joint positions that were used to calculate the angle of the command are out of date.

Consider the following pseudocode snipped to actuate a simulated joint to a desired angle:

```
target_angle = 1.0
joint_angle = joint.getAngle()

while joint_angle != target_angle:
        diff = joint_angle - target_angle
        joint.setVelocity(diff)
        thread.waitOneTick()
        joint_angle = joint.getAngle()
```

Such a code snippet would run infinitely, alternating just above and just below the target angle. This is because of the one tick delay in actuation. Table 3.3 shows the perceived joint angle tick-by-tick.

Tick	**Approx. Perceived Angle**	**Actuation Command**
1	0.0	1.0
2	0.0	1.0
3	1.0	0.0
4	2.0	-1.0
5	2.0	-1.0
6	1.0	0.0
7	0.0	1.0
8	0.0	1.0
9	1.0	0.0
10	2.0	-1.0

Table 3.3: An example of poor joint actuation using the simulator

A more effective approach is to predict the joint angle at the time of actuation, which is the next tick, and find the desired velocity by subtracting this from the target angle. To predict the angle of the joint next tick, take the current joint angle plus the previous joint velocity command. As the actuation of joints is delayed by one tick, the previous command will be actuated this tick, affecting the joint reading of next tick.

This is reflected in the following code snipped:

```
target_angle = 1.0
previous_command = 0.0
joint_angle = joint.getAngle()

while joint_angle != target_angle:
        next_tick = joint_angle + previous_command
        diff = target_angle - next_tick
        joint.setVelocity(diff)
        previous_command = diff
        thread.waitOneTick()
        joint_angle = joint.getAngle()
```

However, there is one more piece to the puzzle. The code and table above assume that the joint angles change at a magnitude of their velocity once every tick. This, unfortunately, is untrue. Through trial and error, it was discovered that the 3DL Simulator actually actuates joints at a speed of approximately 1.22173 $*$ velocity per tick.

If we incorporate this in to our algorithm above, we get:

```
VEL_PER_TICK = 1.22173
target_angle = 1.0
previous_command = 0.0
joint_angle = joint.getAngle()

while joint_angle != target_angle:
        next_tick = joint_angle + (previous_command * VEL_PER_TICK)
        diff = (target_angle - next_tick) / VEL_PER_TICK
        joint.setVelocity(diff)
        previous_command = diff
        thread.waitOneTick()
        joint_angle = joint.getAngle()
```

The code above will correctly actuate a joint to a desired position.

Another divergence between the actuation of the virtual NAO and the physical NAO is the LHipYawPitch and RHipYawPitch joints. As can be seen in 3.2, the NAO V5 hardware interface does not allow the actuation of the RHipYawPitch, as it just mirrors the angle of the LHipYawPitch. This behaviour is not shared by the simulator, which allows the independent actuation of both joints. To circumnavigate this, simswift provides the same joint commands for both joints.

The clear differences between the joints of the real NAO V5 and the simulated NAO were initially concerning, as it appeared that the rUNSWift Motion module may not perform well, or work at all. Of most concern was the *Walk2014Generator*, which has

been finely tuned to produce a fast walk for the real NAO V5. However, once the 3DL Simulator API was carefully and correctly mapped to interface with the rUNSWift Motion module, the results were surprising.

The Motion module performs well on the simulated robot. The Walk2014Generator is simulated well, the strafing and ball approach routines work, the kicks and goalie dives work. There are only a few caveats. One is that the traditional back and front getups do not correctly transform the robot from a fallen to a standing position, as the robot falls back over. This is most likely due to minute divergences in the modelling of the robot, or its environment. For instance, on the real NAO V5, the rUNSWift slow back getup moves the robot's hands behind its back, and uses them as leverage to tip the robot on to its feet, in to a crouched position. This pivoting motion did not work in the simulation. Fortunately, a recently implemented fast front getup does work in simulation. To make use of this, the slow back getup was modified to tip the robot on to its front, from where it can execute a fast front getup.

Another caveat is that the max speed of the walk must be limited, or the robot tips over. Robots using the rUNSWift 2014 walk are able to walk at a top speed of 300mm/s. However, it appears that the simulated walk becomes unstable if the robot walks faster than 250mm/s. As such, the simswift walk max speed is limited to 250mm/s, instead of the 300mm/s rUNSWift max speed. It is hard to know the cause for this, but again it is most likely caused by minuscule modelling abstractions of either the robot or the environment.

Another possible cause is the difference in tick speed between the rUNSWift *Motion* module and the simulator. The rUNSWift Motion joint has a tick rate of 10ms, while the simulator server has a 50ms tick rate. Such a delay in perception and actuation may cause the limited speed of the walk in simulation.

3.3.2 The environment

The 3DL Simulator is used primarily in the RoboCup 3D Soccer Simulation League. As such, it is configured out of the box to model the 3DL environment. While the general features are the same, the proportions of the 3DL environment is quite different to that of the SPL.

Table 3.4 highlights the differences in the environment between the SPL and the 3DSL.

Item	SPL	3DSL
Field	9m x 6m	30m x 20m
Goals	1.6m x 0.5m	2.1m x 0.5m
Goal box	0.6m x 2.2m	1.8m x 3.9m
Centre circle	0.75m radius	2m radius

Table 3.4: Differences between the RoboCup SPL and 3DSL environment

These differences were surmounted by modifying the 3DSL server configuration files to suit the SPL environment.

Chapter 4

Finding the Ball

4.1 Overview

As discussed in section 2.3, the 2016 ball searching component is not optimal. It is static in nature, and fails to take in the account the ever-changing probability distribution of the position of the ball. This thesis presents a new ball searching component architecture and implementation, which is later compared with existing infrastructure.

The proposed ball searching architecture is made up of three main processes: *probability estimation*, *search location selection*, and *sensor placement*. The probability estimation process estimates a probability distribution of the position of the ball on the SPL field; the search location selection process recommends a position on the field to search for the ball; and the sensor placement process actuates the robot to search for the ball.

Each update cycle, the workflow of the new architecture is as follows. The probability estimation component is updated with new information from the environment. It uses this information to estimate an updated probability distribution. The updated distribution is provided to the search location selector, where it is used in conjunction with strategy to select a search location, a position on the field to search for the ball. Finally, the sensor placement component receives the latest search location, which is

taken in to consideration for the actuation of the robot and thus, placement of sensors. This architecture is illustrated in Figure 4.1.

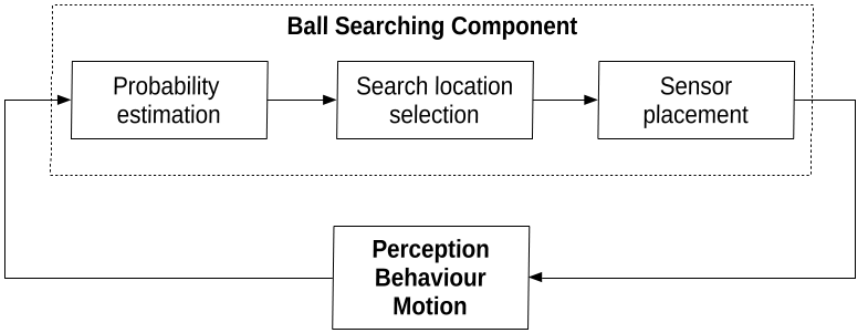

Figure 4.1: The proposed ball searching component architecture.

4.2 New ball searching component implementation

This section presents an implementation of the architecture described in Figure 4.1, the majority of which is handled in the new C++ *FindBall class*, and the new Python *FindBall behaviour*.

Upon instantiation, the *FindBall class* is provided with a *BallDistributionGenerator* instance and a *FindBallStrategy* instance. BallDistributionGenerator and FindBallStrategy are abstract base classes, which guarantees a partial interface for derived types. Classes that implement the BallDistributionGenerator interface receive information about the state of the robot and provide probability estimations regarding the location of the ball. Classes that implement the FindBallStrategy receive the latest ball positional probability estimation, and suggest a search location to look for the ball. This compartmentalised architecture allows for BallDistributionGenerator and FindBallStrategy derivations to be easily replaced with different implementations, or omitted entirely.

Each tick, the FindBall class reads the current sensory information from the robot. If a

valid pointer to a BallDistributionGenerator instance has been provided, the information is forwarded to the derived type. The BallDistributionGenerator instance returns a *PositionalDistribution*, which defines an estimation of the probability distribution of the position of the ball on the SPL field.

If a valid FindBallStrategy pointer has been provided, it is forwarded the PositionalDistribution. The FindBallStrategy instance returns an *AbsCoord*, which is a two-dimensional Cartesian coordinate that represents a point on the SPL field, deemed a suitable location to search for the ball. This point is written to the Blackboard.

Outside of the C++ FindBall class, the search location is read by the similarly named *FindBall* Python behaviour. To avoid confusion with its C++ namesake, the FindBall Python behaviour will be referred to as the 'FindBall *behaviour*', while the FindBall C++ class will be referred to as the 'FindBall *class*'. The FindBall behaviour walks the robot towards points on the field where the ball might be found. This implements the *sensor placement* process.

The new implementation is illustrated in Figure 4.2.

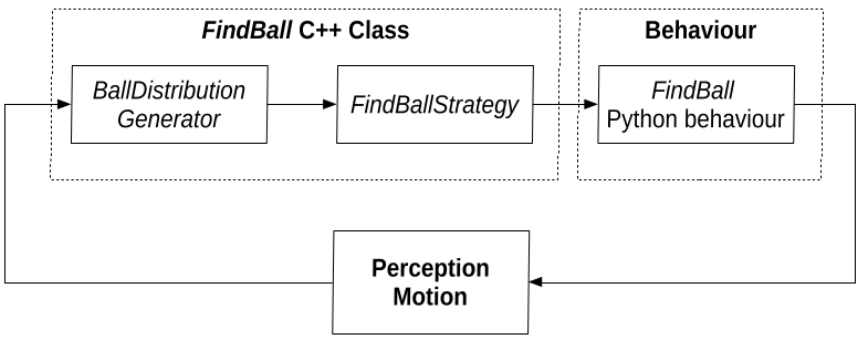

Figure 4.2: The implementation of the architecture described in Figure 4.1.

4.2.1 Estimating probability: *BallDistributionGenerator* implementations

This section presents two implementations of the BallDistributionGenerator interface, as described in Section 4.2.

BallDistributionGenerators use the *PositionalDistribution* class to provide an estimation of the probability distribution of the position of the ball. The SPL field is broken up in to J segments, each representing a mutually exclusive equal area of the SPL field, so that the disjunctive union of their area is equal to the field.

The centre of each segment in the *PositionalDistribution* is used as a search location for the ball. As such, J should be chosen so that the distance from the centre of each segment to its corners is equal to the reliable detection distance of the ball detector in use, minus some kind of error threshold to account for error in the reliable detection distance estimation. The result is a grid of segments, where each segment is a search location to be visited in search of the ball. If configured correctly, the grid should cover the entire SPL field.

At the time of writing, the PositionalDistribution class is configured to use 9 field segments. This creates a grid of segments with search locations at the points defined in Table 4.1.

Segment No.	**Middle X Coord.**	**Middle Y Coord.**
1	-3000	2000
2	0	2000
3	3000	2000
4	-3000	0
5	0	0
6	3000	0
7	-3000	-2000
8	0	-2000
9	3000	-2000

Table 4.1: The points used for the whole field search test.

With a reliable ball detection distance of 2500mm, this provides a theoretical coverage

that covers the entire SPL field. Figure 4.3. Note that this coverage does not include the traversal between points. As the order in which the points are visited is dynamic, complete coverage can only be estimated. Figure 4.4 shows the theoretical complete coverage for one possible route that the search points could be visited in.

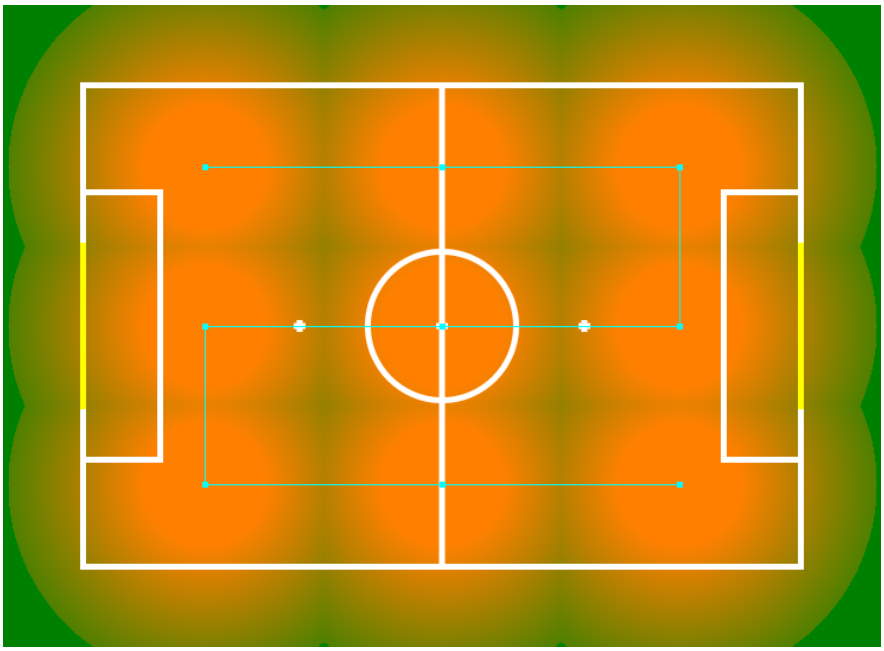

Figure 4.3: The theoretical coverage of the presented ball searching component, given the search points listed in Table 4.1.

Each segment has an associated estimated probability, which is an estimation of the chance that the segment contains the SPL ball. Initially, this is a uniform distribution, where each segment is estimated to have equal chance of containing the ball. Each time the probability of a segment is updated, the distribution is normalised, meaning the following holds true:

$$\sum_{j=1}^{J} P(S_j) = 1$$

Where J is the number of segments in the distribution, S_j is the jth segment of the distribution, and $P(S_j)$ is the estimated probability of segment S_j containing the ball.

At the time of writing, the PositionalDistribution instances in use break the field up in to 9 segments. The probability of each segment is determined by implementation specific metrics.

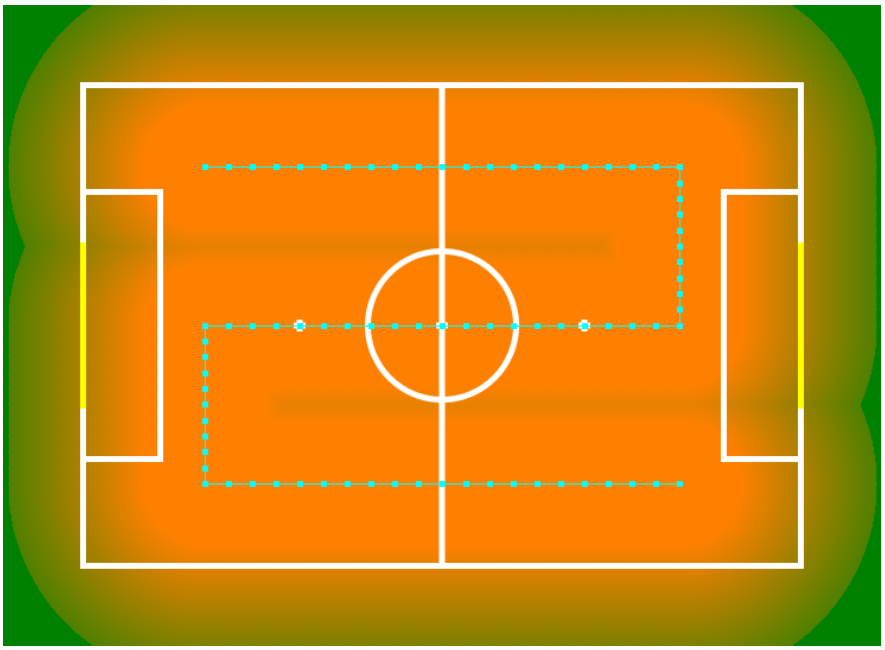

Figure 4.4: The complete theoretical coverage of one possible route that could be taken by the presented ball searching component, given the search points listed in Table 4.1.

4.2.1.1 Estimating local probability: *CompositeDistribuitonGenerator*

The *CompositeDistributionGenerator* class is an implementation of the BallDistributionGenerator interface, described in Section 4.2. It provides an estimation of the probability distribution of the position of the ball using instances of *DistributionComponent* components. Every tick, each component, given a PositionalDistribution of the ball, will transform the distribution in some way. Below is a list of the components used in the CompositeDistributionGenerator at the time of writing:

- *BallPosComponent*: Increases the probability of the ball being in the field segment where the ball is currently sighted.
- *EntropyComponent*: Converges the probability distribution towards a uniform distribution over time.
- *FlipFilterComponent*: Resets the probability distribution to a uniform distribution upon detection of mislocalisation.

- *RobotPosComponent*: Lowers the probability of the ball being in the same segment as the robot if it cannot see the ball.
- *PenalisedComponent*: Resets the probability distribution to a uniform distribution when the robot is penalised.

4.2.1.2 Sharing with friends: *TeamDistributionGenerator*

The *TeamDistributionGenerator* class is a distributed implementation of the BallDistributionGenerator abstract base class described in Section 4.2. Each tick, an underlying local BallDistributionGenerator instance is updated, and is used to generate a local probability distribution of the position of the ball. At the time of writing, the local distribution is generated using a *CompositeDistributionGenerator* instance, described in Section 4.2.1.1. The local distribution is sent to each teammate.

Similarly, a local distribution is received from each teammate. These are combined to form a distributed probability distribution of the position of the ball. Distributions are combined by multiplying the local probability estimation of each segment by the sum of all robot local estimations of the same segment (note that this includes the robot's own local estimation). The estimation is multiplied by the local probability estimation so that the local probability estimation is weighted more heavily. The combined distribution is normalised before being returned.

The pre-normalised probability for each segment of the distributed distribution is calculated as follows:

$$P(D_j) = P(S_j) \times \sum_{n=1}^{N} P(R_{nj})$$

Where $P(S_j)$ is the local probability estimation of segment j, N is the number of robots, and R_{nj} is the local probability estimation of the jth segment from the nth robot. Note that the sum of robot estimations includes the robot's own estimation.

This architecture of the TeamDistributionGenerator class is depicted in Figure 4.5.

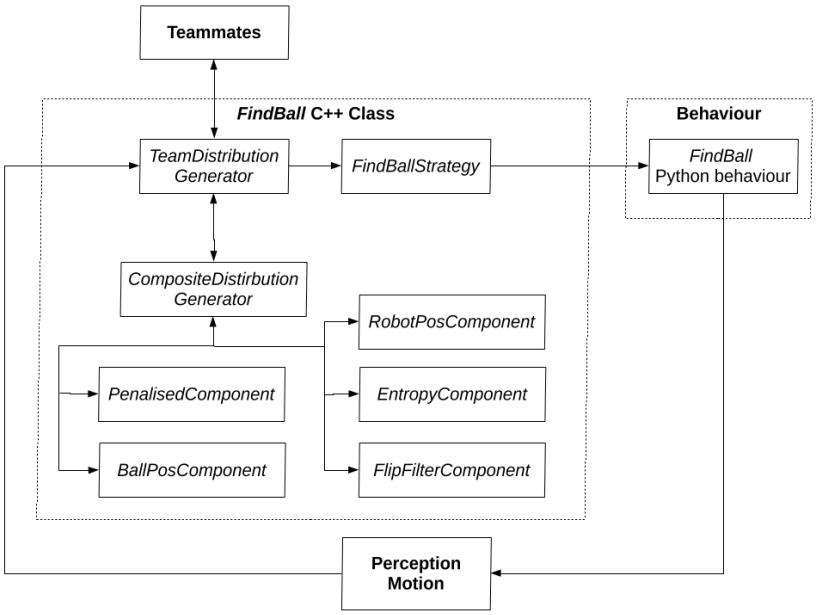

Figure 4.5: The architecture surrounding the TeamDistributionGenerator described in Section 4.2.1.2. It is an implementation of the BallDistributionGenerator.

4.2.2 Where to go: *SimpleStrategy* class

The SimpleStrategy class is an implementation of FindBallStrategy, described in Section 4.2. SimpleStrategy uses a *UtilityCalculator* instance to calculate the utility of each segment on the field. UtilityCalculator is an abstract base class, where derived types use implementation specific metrics to calculate the *utility* of visiting a certain field segment to search for the ball. The segment with the highest utility is chosen as the search location to find the ball. At the time of writing, the *DistirbutedUtilityCalculator* is used to calculate utility.

4.2.2.1 *SimpleUtilityCalculator* class

The *SimpleUtilityCalculator* class derives from the UtilityCalculator interface described in Section 4.2.2. Given the current state of the robot and its environment, this class defines the utility of visiting a certain segment on the field as a benefit-cost ratio.

The benefit-cost ratio is calculated as the estimated probability of the segment con-

taining the ball (the *benefit*) divided by the time it would take to visit the square (the *cost*). It follows that as the estimated probability of a segment containing the ball increases, so does its utility. Conversely, as the distance from the robot to the segment increases, the utility decreases.

For each segment, the utility is calculated using the following algorithm:

$$U(S_j) = \frac{P(S_j)}{Time(pos, centre(S_j))}$$

Where S_j is the segment being considered, $Time(a, b)$ is the time it takes for the robot to travel from a to b, pos is the current position of the robot, and $center(S_j)$ is center of the segment j.

This is based upon the heuristic presented by Sarmiento et al. (2003) [SMH03], discussed in Section 2.3.3.

4.2.2.2 *DistributedUtilityCalculator* class

The DistributedUtilityCalculator is a distributed implementation of UtilityCalculator, described in Section 4.2.2. The class combines local utility calculation with information received from teammates to provide a distributed utility calculation.

Each tick, the ball search position selected by teammates is received. When the utility of a segment is requested, the local utility is firstly calculated using another derivation of UtilityCalculator. At the time of writing, this is the *SimpleUtilityCalculator* class. Then, the local utility is divided by the number of robots that are already visiting the segment that is being considered.

The formula for distributed utility calculation is as follows:

$$U_d(S_j) = \frac{U_l}{V_j}$$

Where U_d is distributed utility and $U_l(S_j)$ is local utility, S_j is the segment being considered, and V_j is the number of teammates already visiting S_j.

This lowers the utility of search locations that are in the process of being visited by other robots. This is helpful for two reasons. Firstly, search locations that have other robots intending to visit them are already going to be assessed by those robots. Assuming that each robot has equal abilities of perception, the robot already visiting the search location will provide the same information as any other robot visiting the location. Any additional robots searching the same location at the same time would only provide redundant information, reducing the benefit of visiting the location. Instead, robots could head to search locations that no other robots are investigating, where they can provide new and important information.

Secondly, robots heading to the same place on the field may collide with each other, possibly causing robots to fall over. Furthermore, robots in close proximity may impede sensory perception, which might lead to a close-range ball being undetected. This further reduces the benefit of multiple robots visiting the same location at once.

A depiction of the architecture surrounding the DistributedUtilityCalculator is shown in Figure 4.6. A more in-depth depiction of the architecture is described in Figure 4.7.

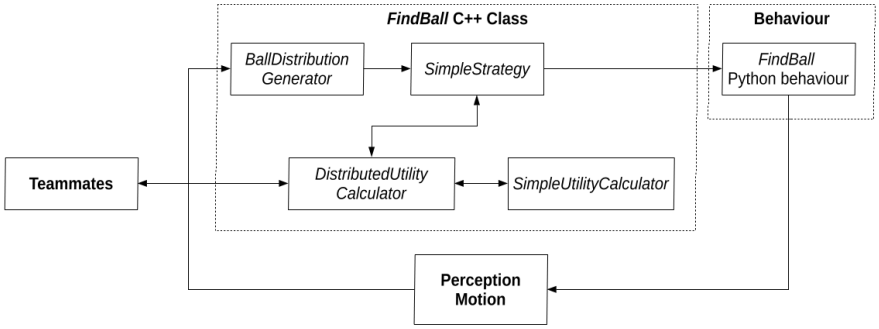

Figure 4.6: The architecture of the distributed SimpleStrategy, including the DistributedUnityCalculator defined in Section 4.2.2.2.

4.2.3 Getting there: *FindBall* Python behaviour

Once the FindBall C++ class has found the most useful position to search for the ball, the FindBall Python behaviour (not to be confused with the FindBall C++ class

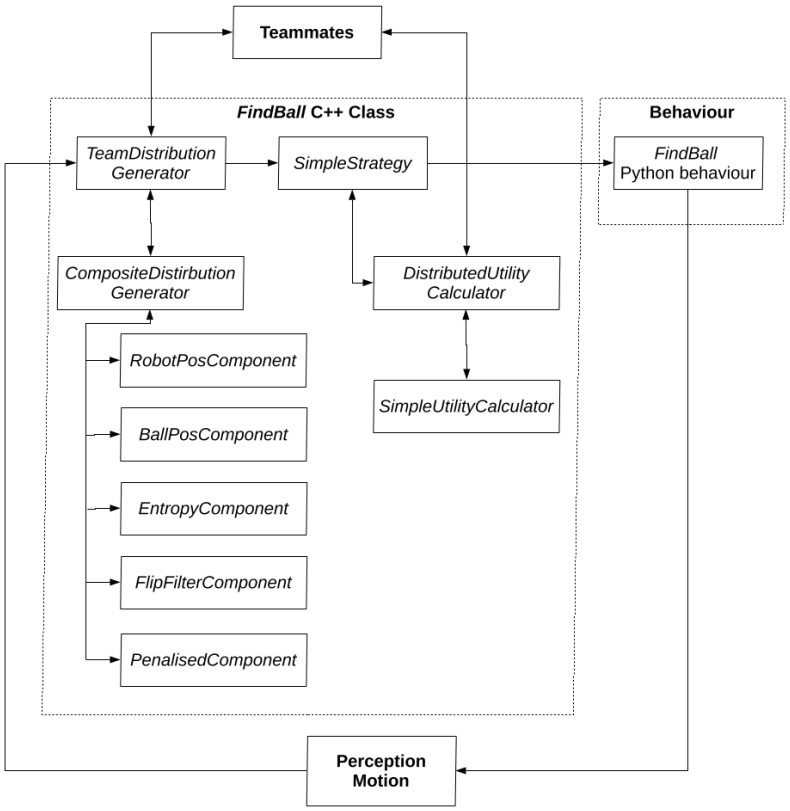

Figure 4.7: A more in-depth visualisation of the architecture shown in Figure 4.6.

described in Section 4.2) uses the position to actuate the robot in a way that effectively places its sensors.

Each update cycle, the FindBall Python behaviour reads the state of the robot. If the behaviour does not yet have a point to walk to to find the ball, it reads the position suggested by the FindBallStrategy instance in use. The behaviour walks to the suggested position.

Upon arriving at the ball finding position, the FindBall behaviour scans the robot head to look for the ball. Failing to find the ball, the FindBall behaviour spins the robot around, again looking for the ball. If the robot can still not see the ball, the FindBall behaviour reads the next suggested position from the FindBallStrategy. This continues until the robot, or a team member, detects the ball.

4.2.4 *offnao* support

The implementation includes a new *FindBall* tab for *offnao*, the debugging tool introduced in Section 2.1.6.

During play, the *FindBall tab* (not to be confused with the *FindBall class* or *FindBall behaviour*) displays ball finding information, overlaid upon an image of the SPL field.

The current position of the robot and the current suggested search location are depicted as small circles. The probability distribution of the ball is depicted using a number of rectangles, one for each segment of the distribution.

Each segment is drawn on-top of its correlating section of the SPL field, matching the area. Segments that have a higher estimated probability to contain the ball appear progressively red in colour, where segments with a lower estimated probability appear green. Figure 4.8 depicts this.

At the time of writing, the depicted probability distribution of the position of the ball is a distributed probability, generated by the *TeamDistributionGenerator* class.

Furthermore, if a distributed *FindBallStrategy* instance is in use (as is at the time of writing), the most recent search location of teammates is also depicted, alongside the teammates themselves. Robots are associated with their search location by the use of an arrow. Figure 4.8 depicts this.

This allows for the debugging and testing of the new Ball Search Component architecture and implementation.

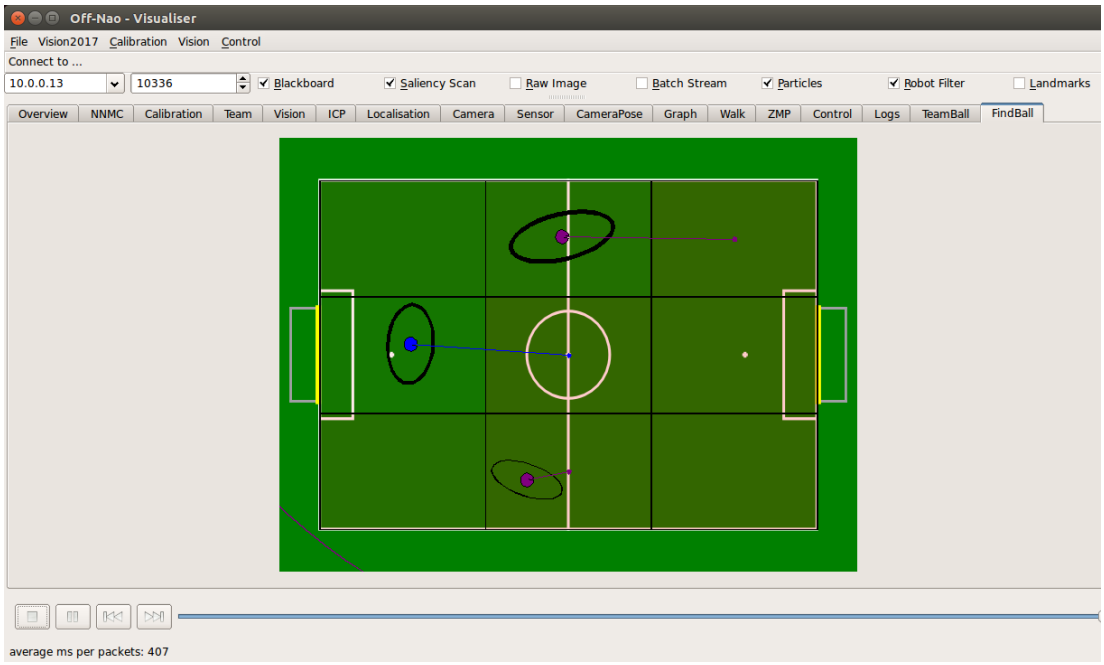

Figure 4.8: The *FindBall* tab on offnao.

Chapter 5

Evaluation

In Chapter 3, this thesis presented *simswift*, a rUNSWift build target that can interface with the RoboCup 3DL simulator, *rcssserver3d*. This allows rUNSWift code to be run in simulation, on a personal computer instead of a physical NAO V5. In Chapter 4, this thesis presented a new ball searching component architecture and implementation, which has been developed solely in simulation, using *simswift*.

This chapter evaluates the utility of the presented ball searching component (2017) by comparing it with the current infrastructure (2016) through both simulated and physical experiments. Prior to the experiments in this chapter, the presented ball searching component had not been tested on a physical NAO, even for calibration purposes. Secondly, this section evaluates the usefulness of using *simswift* for developing robot code. This is done by assessing the success of the new ball searching component, and by comparing the outcomes of simulated experiments with recreations on physical robots.

5.1 Searching the field

This experiment timed the duration it took a team of robots to find the ball with no indication of previous ball positions. This experiment was chosen to demonstrate

ball finding capabilities in situations where there is little or no information to narrow down the search domain. This experiment assesses the effectiveness of implementations conducting a whole field search. This experiment was chosen because of its simplicity, which meant it could be run on both virtual and physical NAOs under controlled conditions.

All tests began with all robots in a penalised state, in a starting position on the side of the field. The starting positions are listed in Table 5.1. They are also overlaid on an image of the field in Figure 5.1. Note that there are no entries for robot numbers 1 and 5, as only 2, 3, and 4 were used. The robot numbers used start at 2 because number 1 is reserved by rUNSWift to be the goalkeeper. Robots are in a penalised state at the beginning of an SPL game, when they commit a foul, or when they leave the field. If one robot was used, it was placed in the top left corner of the field. If three robots were used, two were placed in the top left corner, and one was placed in the bottom left corner. This is consistent with the starting positions during SPL games. Before each test, the ball information stored by each robot is reset. This means that each test is effectively a clean slate and is not affected by previous tests.

Robot Number	Starting X Coordinate	Starting Y Coordinate
2	-3250	3000
3	-2000	-3000
4	-1750	3000

Table 5.1: The starting points for each robot number used.

This experiment was first ran in simulation. Both the new ball searching component and implementation and existing infrastructure were tested 100 times each. The ball was placed in 10 different pre-selected positions, and each position was tested 10 times. A ball was considered to be found when at least one of the robots could both see the ball, and was within 300mm (inclusive) of the ball. If the ball was not found within 180 seconds, the test was ended and the ball was considered to have not been found. The tested ball positions are shown in Table 5.2. The positions are also overlaid on an SPL field in Figure 5.1.

If a robot became incapacitated, it was teleported to the sideline. If a robot continued

Number	X Coordinate	Y Coordinate	Description
1	2250	0	Right middle
2	4000	-2500	Right bottom corner
3	4000	2500	Right top corner
4	4500	3000	Right top corner (on line)
5	2250	3000	Right bottom middle (on line)
6	-4500	3000	Left top corner (on line)
7	4500	0	Right goal box (on goal line)
8	-3500	0	Left goal box (on goal box line)
9	4500	-3000	Right bottom corner (on line)
10	-4500	-1000	Left boundary next to goal (on line)

Table 5.2: The points used for the whole field search test.

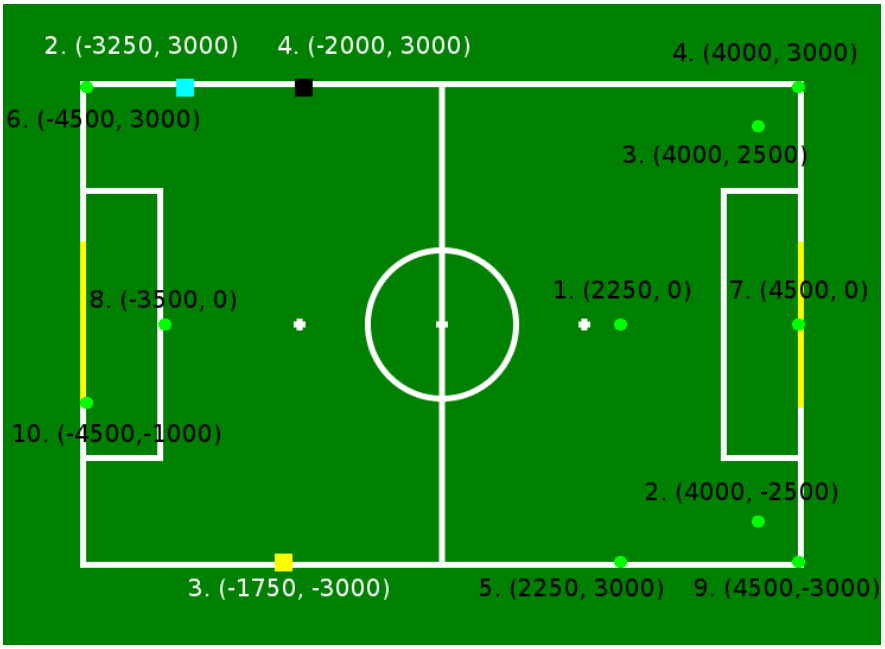

Figure 5.1: The ball positions and robot starting positions used for experiments. Balls are shown in green, and starting positions are shown in other colours. Based on Tables 5.2 and 5.1.

to be incapacitated and this was deemed to significantly affect the outcome of the test, the test was restarted. If the incapacitation did not significantly affect the test, the virtual robot was restarted for the next test. The simulations were ran with a reliable ball detection distance of 2500mm, to attempt to simulate the current vision infrastructure at the time of testing.

5.1.1 Tests in simulation

Firstly, the 2016 FindBall behaviour was tested with three simulated robots as a baseline. The results are shown in Table 5.3.

Position	Tests	Found	Not Found	Total Seconds	Avg. Seconds	Std. Dev.
1	10	10	0	485.00	48.50	4.54
2	10	10	0	803.00	80.30	5.44
3	10	10	0	592.00	59.20	5.23
4	10	1	9	100.00	100.00	0.00
5	10	10	0	870.00	87.00	21.94
Sub-Total:	50	41	9			
Sub-Average:	10.00	8.20	1.80	570.00	75.00	7.43
6	10	8	2	565.00	70.63	44.29
7	10	10	0	613.00	61.30	7.18
8	10	8	2	1119.00	139.88	32.86
9	10	5	5	496.00	99.20	5.39
10	10	8	2	925.00	115.63	57.70
Sub-Total:	50	39	11			
Sub-Average:	10.00	7.80	2.20	743.60	97.33	29.49
Grand Total:	100	80	20			
Grand Average:	10.00	8.00	2.00	656.80	86.16	18.46

Table 5.3: The data from, 100 tests in simulation using the 2016 ball searching component.

Then, the new ball searching component was tested against this baseline with three simulated robots. The results are shown in Table 5.4.

Position	Tests	Found	Not Found	Total Seconds	Avg. Seconds	Std. Dev.
1	10	10	0	579.00	57.90	7.40
2	10	10	0	782.00	78.20	9.41
3	10	10	0	938.00	93.80	29.64
4	10	10	0	1075.00	107.50	23.41
5	10	10	0	754.00	75.40	30.97
Sub-Total:	50	50	0			
Sub-Average:	10.00	10.00	0.00	825.60	82.56	20.16
6	10	9	1	437.00	48.56	33.66
7	10	10	0	821.00	82.10	15.69
8	10	10	0	266.00	26.60	1.28
9	10	7	3	773.00	110.43	20.79
10	10	10	0	306.00	30.60	1.56
Sub-Total:	50	46	4			
Sub-Average:	10.00	9.20	0.80	520.60	59.66	14.60
Grand Total:	100	96	4			
Grand Average:	10.00	9.60	0.40	673.10	71.11	17.38

Table 5.4: The data from, 100 tests in simulation using the new ball searching component (2017).

5.1.2 Tests on the NAO V5

This experiment was complimented with a recreation using physical NAOs. Due to the complexity of testing on physical robots, only the first 5 of the 10 positions were tested, and they were each tested 5 times instead of 10.

5.1.2.1 Single NAO

Firstly, both components were compared using one NAO V5. Table 5.5 shows the results using the existing ball searching component.

Position	Tests	Found	Not Found	Total Seconds	Avg. Seconds	Std. Dev.
1	5	5	0	196.53	39.31	4.41
2	5	5	0	244.65	48.93	8.92
3	5	5	0	201.20	40.24	3.01
4	5	0	5	0.00	0.00	0.00
5	5	5	0	358.39	71.68	50.60
Total:	25	20	5			
Average:	5.00	4.00	1.00	200.15	40.03	13.39

Table 5.5: The data from 25 tests on a single NAO V5 using the existing ball searching component (2016).

Position	Tests	Found	Not Found	Total Seconds	Avg. Seconds	Std. Dev.
1	5	5	0	285.64	57.13	11.40
2	5	5	0	540.16	108.03	21.49
3	5	5	0	545.50	109.10	26.19
4	5	4	1	438.22	109.56	44.27
5	5	5	0	315.25	63.05	6.27
Total:	25	24	1			
Average:	5.00	4.80	0.20	424.95	89.37	21.92

Table 5.6: The data from 25 tests on a single NAO V5 using the new ball searching component (2017).

5.1.2.2 Three NAO V5s

Both components were then tested using three NAO V5s. The results using the existing ball searching component are shown in Table 5.7.

Position	Tests	Found	Not Found	Total Seconds	Avg. Seconds	Std. Dev.
1	5	5	0	161.78	32.36	4.35
2	5	4	1	414.34	103.59	44.38
3	5	5	0	414.39	82.88	32.45
4	5	1	4	50.61	50.61	0.00
5	5	5	0	212.56	42.51	2.55
Total:	25	20	5			
Average:	5.00	4.00	1.00	250.74	62.39	16.75

Table 5.7: The data from 25 tests using three NAO V5s with the existing ball searching component (2016).

Table 5.8 shows the results of using the new ball searching component on three NAO V5s.

Position	Tests	Found	Not Found	Total Seconds	Avg. Seconds	Std. Dev.
1	5	5	0	121.69	24.34	1.29
2	5	5	0	191.43	38.29	2.34
3	5	5	0	354.52	70.90	11.70
4	5	4	1	229.58	57.40	23.25
5	5	5	0	216.28	43.26	6.39
Total:	25	24	1			
Average:	5.00	4.80	0.20	222.70	46.84	9.00

Table 5.8: The data from 25 tests using three NAO V5s with the new ball searching component (2017).

5.2 New ball searching component

To assess the presented ball searching component (2017), it is important to compare the performance of the component in comparison to the 2016 ball searching component, both in simulation and real-life. In Section 5.1.1, the results of 100 tests in simulation for both implementations were presented. In Section 5.1.2, the results of 25 tests in real-life for both implementations were presented, both conducted on singular and sets of three NAO V5 robots.

5.2.1 Simulation results evaluation

Looking at the results in simulation, the 2016 ball searching component (the existing infrastructure), detected 80 out of 100 balls, with a *find rate* of 80%. The 2017 ball searching component (the presented architecture and implementation) detected 96 out of 100 balls, with a find rate of 96%. The number of detections by both components for each ball position is depicted in Table 5.9.

	Existing BSC (2016)	**New BSC (2017)**
Ball Pos 1	10	10
Ball Pos 2	10	10
Ball Pos 3	10	10
Ball Pos 4	1	10
Ball Pos 5	10	10
Ball Pos 6	8	9
Ball Pos 7	10	10
Ball Pos 8	8	10
Ball Pos 9	5	7
Ball Pos 10	8	10
Total:	80	96

Table 5.9: A summary of all the balls found by the existing ball searching component (2016) and the new ball searching component (2017).

Furthermore, of the balls found, the 2016 ball searching component detected them in an average of 86.16 seconds, while the 2017 ball searching component detected them on average in 71.11 seconds. A comparison of the average detection time for each ball position is in Table 5.10.

It is interesting to also look at the player positions throughout the tests. By overlaying positional data throughout the 100 tests on to an image of the SPL field, the overall search area of each player can be assessed. Figures 5.2 shows the search area of the three robots during the testing of the 2016 ball searching component. Note that the complete search area of the three robots looks quite similar. This is because the current ball searching component (2016) is not a distributed approach, so robots will search the same area. This is quite wasteful, as robots searching the same area will make the same observations.

	Existing BSC (2016)	**New BSC (2017)**
Ball Pos 1 Avg. Time	48.50	57.90
Ball Pos 2 Avg. Time	80.30	78.20
Ball Pos 3 Avg. Time	59.20	93.80
Ball Pos 4 Avg. Time	100.00	107.50
Ball Pos 5 Avg. Time	87.00	75.40
Ball Pos 6 Avg. Time	70.63	48.56
Ball Pos 7 Avg. Time	61.30	82.10
Ball Pos 8 Avg. Time	139.88	26.60
Ball Pos 9 Avg. Time	99.20	110.43
Ball Pos 10 Avg. Time	115.63	30.60
Average:	86.16	71.11

Table 5.10: A summary of the average time it took the existing ball searching component (2016) and the presented ball searching component (2017) to find the ball in simulation with three robots.

In comparison, the complete search areas of the robots using the new ball searching component (2017) looks much different. It can be seen in Figure 5.3. Note that the search areas of each robot look different. This is because of the distributed approach that the new ball searching component (2017) takes. This is more efficient than the current searching component (2016) as robots are collecting complementary data, instead of redundant data.

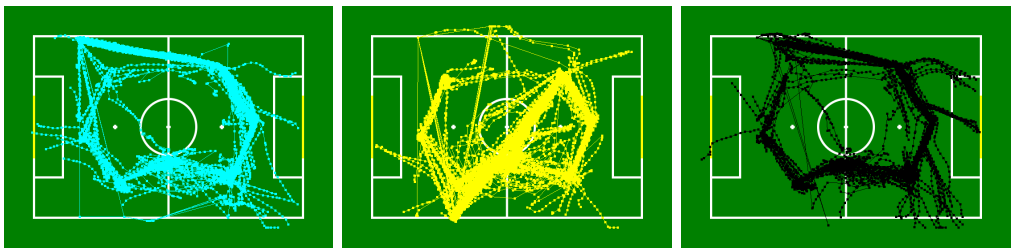

Figure 5.2: Shows the search routes of robots throughout 100 simulated tests using the current ball searching component (2016).

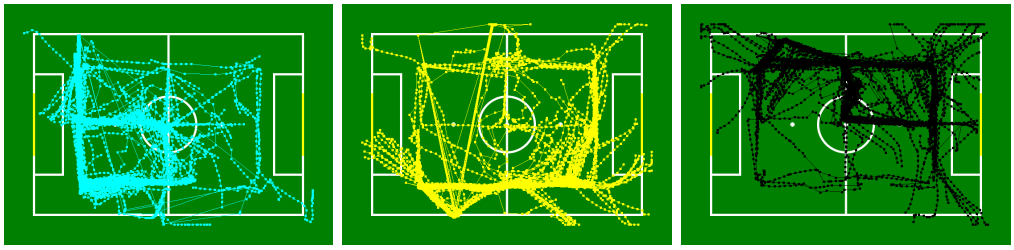

Figure 5.3: Shows the search routes of robots throughout 100 simulated tests using the new ball searching component (2017).

5.2.2 Real-life results evaluation

In simulation, the new ball searching component found more balls in less time than the 2016 predecessor. However, it's important to also compare the implementations in real life, as this is where performance really counts.

In comparison to the existing ball searching component (2016), the new ball searching component (2017) found more balls in more places. The data from the physical experiments on single NAO V5 robots show that the current ball searching component (2016) found 20 out of 25 balls, with a find rate of 80%. The data from experiments on three NAO V5 robots show the same overall find rates, with the current ball searching component (2016) finding 20 out of 25 balls (80% find rate) and the new ball searching component finding 24 out of 25 balls (96% find rate).

The number of detections from each component on both one and three robots is compared in Table 5.11.

	Existing BSC	**New BSC**	**Existing BSC**	**New BSC**
Robots	single	single	three	three
Ball Pos 1	5	5	5	5
Ball Pos 2	5	5	4	5
Ball Pos 3	5	5	5	5
Ball Pos 4	0	4	1	4
Ball Pos 5	5	5	5	5
Total:	20	24	20	24

Table 5.11: A summary of all the balls found by the existing ball searching component (2016) and the new ball searching component (2017) in real-life single and multi-robot tests.

Looking at the average find time, it appears that the existing ball searching component finds the ball faster than the new ball searching component in the single robot case. The 2016 ball searching component found balls in an average of 40.03 seconds, while the 2017 ball searching component found them in 89.37 seconds on average. This is more than double the existing component, but the result is not surprising. While the existing ball searching component walks around a hard-coded set of points on the field,

the proposed ball searching component strategically searches the entire field for the ball.

If there is only one robot on the field, then that robot will search the entire field by itself, which is expected to take a longer time than the existing ball searching component. The strength of the proposed ball searching component is in its distributed nature, which allows a team of NAOs to search separate areas of the field and combine search data.

Looking at the three robot case, the 2016 ball searching component detected balls in an average of 62.39 seconds, while the 2017 ball searching component detected them on average in 46.84 seconds. A comparison of the average detection time for each ball position are shown in Table 5.12.

	Existing BSC	**New BSC**	**Existing BSC**	**New BSC**
Robots	single	single	three	three
Ball Pos 1 Avg. Time	39.31	57.13	32.36	24.34
Ball Pos 2 Avg. Time	48.93	108.03	103.59	38.29
Ball Pos 3 Avg. Time	40.24	109.10	82.88	70.90
Ball Pos 4 Avg. Time	0.00	109.56	50.61	57.40
Ball Pos 5 Avg. Time	71.68	63.05	42.51	43.26
Average:	40.03	89.37	62.39	46.84

Table 5.12: A summary of the average time it took the existing ball searching component (2016) and the presented ball searching component (2017) to find the ball using three real NAO V5 robots.

An interesting fact of note here is that the existing ball searching component actually took longer with three robots instead of a single robot. This is most likely from the fact that multiple robots crowd the shared static search path, which causes collisions, robots falling over, and thus longer search times.

From the data collected in real life tests, the new ball searching component (2017) found more balls than the current ball searching component (2016). This is congruent with the data found in simulation. In the single robot case, the existing ball searching component (2016) finds balls faster, while in the multi-robot case, the new ball searching component (2017) is the fastest.

5.3 *simswift* VS the world

The success of the new ball searching component, as described in Section 5.2, provides an indirect evaluation about the capabilities of *simswift* for the development of robot code. However, it is also important to compare the results of the simulated experiments, and their real-life repetitions. Note that only the first five ball positions were tested on the physical NAOs. For a fair comparison, only the simulation tests that use the same ball positions will be considered (the first 50 tests).

The results of experiments involving three robots are shown in Table 5.13.

	Simulated	**Real**	**Simulated**	**Real**
Component	2016 Ball Search	2016 Ball Search	2017 Ball Search	2017 Ball Search
Robots	three	three	three	three
Tests	50	25	50	25
Total Found	41	20	50	24
Total Not Found	9	5	0	1
Find Rate	0.82	0.80	1.00	0.96
Avg. Avg. Seconds	75.00	40.03	82.56	67.46

Table 5.13: A summary of all the tests that were ran with three robots.

5.3.1 Vision

Looking at the data in Table 5.13, the most interesting statistic is the similarities between the *find rate* data of simulated experiments and their equivalent physical recreation. The find rate is the percentage of tests where the ball was successfully found within the time limit. A similar find rate indicates that the *simswift* vision system has a good estimation of rUNSWift ball detector in real-life.

However, it is important to look in more detail, and consider the find rate of individual ball positions, as the simulated and real-life experiments may have undetected balls in different positions. Table 5.14 compares the find rate of each specific ball position.

From 5.14, it can be seen that the majority of ball positions have a similar find rate, indicating that the *simswift* vision system has a good estimation of rUNSWift ball

Type	**Simulated**	**Real**	**Simulated**	**Real**
Component	2016 Ball Search	2016 Ball Search	2017 Ball Search	2017 Ball Search
Ball Pos 1 FR	1.00	1.00	1.00	1.00
Ball Pos 2 FR	1.00	0.80	1.00	1.00
Ball Pos 3 FR	1.00	1.00	1.00	1.00
Ball Pos 4 FR	0.10	0.25	1.00	0.80
Ball Pos 5 FR	1.00	1.00	1.00	1.00
Combined FR	0.82	0.80	1.00	0.96

Table 5.14: The comparitive find rate of ball positions between simulated and real-life tests.

detection, at least for the ball positions that were tested. In both the simulated and real NAO test for the 2016 ball searching component, only one ball was found at position 4.

5.3.2 Motion

Another interesting statistic from Table 5.13 is the *average average seconds* that experiments took to find the ball. As noted in Section 3.3.1.2, due to instability the simulated robot has a capped max speed at 250mm/s instead of the physical robots 300mm/s. Further to this, it seems that the simulated robot walks slower than the requested speed, even when operating at the capped speed. Table 5.15 shows the speed of a simulated robot walking the width of the SPL field, with a target speed of 250mm/s.

Attempt	**Simulated Speed (mm/s)**
1	177.42
2	175.94
3	176.42
4	180.86
5	181.91
6	177.89
7	175.16
8	181.21
9	180.5
10	182.06
Average:	178.937

Table 5.15: The real speed of a simulated robot programmed to walk at 250mm/s.

It is uncertain why this behaviour is exhibited by the *simswift*. It is possibly a side-

effect of the actuation delay and tick speed mismatch between the simulation server and simswift, slowing down the entire operating speed of Motion as it waits for delayed joint information from the server. As mentioned in 3.3.1.2, the simulation server ticks every 50ms, whereas Motion ticks every 10ms.

Regardless of the cause, it is undoubtable that the walk is much slower in simulation than in real life. Looking at Table 5.15 and given a physical robot speed of 300mm/s, the simulator appears to walk on average 40.35% slower than the real NAO. This affects the accuracy of the *average average seconds* data in Table 5.13. Table 5.16 compares the average find times for each ball point that have been adjusted to take the simulator speed difference in to account.

Type	**Simulated**	**Real**	**Simulated**	**Real**
Component	2016 Ball Search	2016 Ball Search	2017 Ball Search	2017 Ball Search
Adjusted Ball Pos 1 Avg. Time	28.93	32.36	34.54	24.34
Adjusted Ball Pos 2 Avg. Time	47.90	103.59	46.64	38.29
Adjusted Ball Pos 3 Avg. Time	35.31	82.88	55.95	70.90
Adjusted Ball Pos 4 Avg. Time	59.65	50.61	64.12	57.40
Adjusted Ball Pos 5 Avg. Time	51.89	42.51	44.97	43.26
Adjusted Average:	44.74	62.39	49.24	46.84
Adjustment:	0.5964666667	1.00	0.5964666667	1.00

Table 5.16: The comparitive average time to find balls at each position between simulated and real-life tests, adjusted for speed bias.

The adjusted average time to find the ball at each position is a more reasonable comparison. Table 5.17 compares the differences between the adjusted average times of each ball position in simulation and real-life, and finds the average difference.

	2016 Ball Searching	**2017 Ball Searching**
Ball Pos 1 Difference	-3.43	10.20
Ball Pos 2 Difference	-55.69	8.36
Ball Pos 3 Difference	-47.57	-14.96
Ball Pos 4 Difference	9.04	6.73
Ball Pos 5 Difference	9.38	1.72
Average:	-17.65	2.41

Table 5.17: The difference in adjusted average search times for ball positions, between simulated and real experiments.

However, while the average time difference data is interesting, it is difficult to draw meaningful conclusions from it. The first reason for this is the noise in the data collected. The nature of robotic experiments with real-time sensory information and the physical actuation of joints introduces noise. The second reason is that the robot is not necessarily taking the same route in each experiment, making time comparisons less valuable. As the positional data of physical robots was not captured, the overlapping of robot routes in simulated and physical tests is unavailable.

Chapter 6

Conclusion

From the results presented in Chapter 5, it is clear that robot code can be reliably developed and tested using *simswift*, the simulator-ready rUNSWift build target presented in Chapter 3. The ball searching component (BSC) that was developed in complete simulation using *simswift*, and presented in Chapter 4, was shown to find more balls than its predecessor in both simulated and real-life tests. Furthermore, in tests with multiple robots, the presented ball searching component found balls faster than its predecessor too. Of note is the accuracy of the simulated Vision system in *simswift*, which was congruent with the performance of the rUNSWift Vision system on physical robots.

On the other hand, it was found that the rUNSWift walk engine is much slower in simulation than in real-life. This is most likely due to a tick rate mismatch and actuation delay on the simulator server. However, this divergence from the physical robots is measurable and predictable, and its effect can be accounted for in measurements. Nevertheless, this divergence will affect the fidelity of simulated robot code that is dependent on the speed of Motion, and this should be considered during simulation.

Following this, it could be suggested that *simswift* is not appropriate for developing environment facing modules and components such as Motion and Vision. Such problems are better solved by development on physical robots, where they can be finely tuned

to suit the real world.

Conversely, *simswift* has been shown to be appropriate, if not preferable, for developing code that does not directly interact with the environment. Insular code such as Behaviour components, or the Localisation module, are able to be developed using *simswift* without the constant use of physical robots. This is of great benefit to developers, as it negates many of the difficulties associated with embedded development. However, developers should be aware of the limitations of simulation, and should finely tune their simulated designs on physical robots.

6.1 Future Work

While it has been demonstrated that robot code can be developed using *simswift*, there are improvements that can be made to the simulator code.

Firstly, the simulated vision system should be re-factored to be highly customisable, allowing developers to nominate minimum / maximum distances that objects can be detected, with configurable amounts of noise for different observations. Any missing field features that are not currently implemented should be. More specifically, a highly accurate and configurable model of the ball detector should be created, which increases the realism of detections. Troublesome balls, such as those on lines or behind other robots, should only be detected if they would be in real-life.

Secondly, the Motion delay that was observed on the simulator in Chapter 5 should be further investigated. It is imperative that the Motion divergence between simulated and physical robots is minimised to maximise the fidelity of simulation.

Once *simswift* has been improved, there are many research opportunities that present themselves. Machine learning could be used to optimise Behaviours, or even Localisation. Previous UNSW students [Yus11] [Tam12] have suggested that machine learning could be used to develop robot code. With *simswift*, the opportunity now presents itself. Controlled problems such as kick-off or penalty strategies, to more complex

problems such as positioning, no Wi-Fi behaviour, or Localisation tuning are possible projects using *simswift*.

However, anyone pursuing such a project should be aware of the divergences between the simulator and the physical world. For any machine learning projects, it would be advised to take a Grounded Simulation Learning (GSL) approach, as discussed in Chapter 2 [HS17]. In fact, using a GSL approach has possible applications for machine learning Motion code, as was demonstrated in [HS17]. This begets projects that optimise current Motion routines or designing entirely new ones, such as a chip kick.

The new ball searching component presented in this thesis also provides the opportunity for future work. The implementation presented uses an absolute grid on the field, which is traversed to search for the ball. Initially, this design used a robot-relative grid, which was centred around the robot. As the robot moved, its odometry calculations were used to move the robot inside the grid. This meant that when the estimation of the robot pose changed, the ball searching data was unaffected, as it was based on robot relative information, not absolute positioning.

However, this proved to be ineffective due to the inaccuracies of the odometry calculations, and so absolute positioning was used, tracking searched areas with the robot pose from Localisation. The problem with absolute positioning is that if a robot is mislocalised, it is marking incorrect sections of the field as searched. To combat this, ball searching history is reset when a robot detects a major change in pose estimation. This is not optimal. More research in to robot relative methods is encouraged to try and find an implementation that does not rely on absolute positioning. Alternatively, a component could be designed for the current implementation that does not reset history on mislocalisation, but instead transforms the search history of the field so that it is consistent with the new robot pose.

Another possible application of *simswift* is the implementation of a continuous integration server. Periodically, the server would pull the latest version of robot code from the rUNSWift repository, compile it as a simswift build, and test the code in simulation.

The tests would range from simple single robot *Striker* tests, to more complicated team tests. This could be used to track the progression and milestones of the code base.

Bibliography

[Ald] Aldebaran. NAO - Technical overview. `http://doc.aldebaran.com/2-1/family/robots/index_robots.html`. Online; accessed 11-July-2017.

[BDR+10] J. Boedecker, Klaus Dorer, Markus Rollmann, Yuan Xu, Feng Xue, Marian Buchta, and Hedayat Vatankhah. *SimSpark User Manual*. RoboCup Soccer Server 3D Maintenance Group, 2010.

[Cha11] Carl Chatfield. rUNSWift 2011 Vision System: A Foveated Vision System for Robotic Soccer, 2011.

[Com17] RoboCup Technical Committee. RoboCup Standard Platform League (NAO) Rule Book, 2017.

[CS13] B. Crane and S. Sherratt. UNSWift 2D Simulator; Behavioural Simulation Integrated with the rUNSWift Architecture, 2013.

[Hen13] B. Hengst. Reinforcement Learning of Bipedal Lateral Behaviour and Stability Control with Ankle-Roll Activation. In *16th International Conference on Climbing and Walking Robots, Sydney Australia (CLAWAR2013)*. World Scientific Publishing Company, 2013.

[HS17] J. Hanna and P. Stone. Grounded Action Transformation for Robot Learning in Simulation. In *Proceedings of the Conference on Artificial Intelligence (AAAI)*, 2017.

[MSS+16] D. McKinnon, P. Schmidt, H. Smith, K. Ng, J. Collette, G. Katz, S. Harris, B. Hall, B. Hengst, M. Pagnucco, and C. Sammut. rUNSWift Team Report 2016, 2016.

[RHH+10] A. Ratter, B. Hengst, B. Hall, B. White, B. Vance, C. Sammut, D. Claridge, H. Nguyen, J. Ashar, M. Pagnucco, S. Robinson, and Y. Zhu. rUNSWift Team Report 2010, 2010.

[RLM+09] T. Röfer, T. Laue, J. Müller, O. Bösche, A. Burchardt, E. Damros, K. Gillmann, C. Graf, T. J. de Haas, A. Härtl, A. Rieskamp, A. Schreck, I.Sieverdingbeck, and J.-H. Worch. B-Human Team Report and Code Release 2009. In *RoboCup 2009: Robot Soccer World Cup XIII Preproceedings*, 2009.

[SMH03] A. Sarmiento, R. Murrieta, and S.A. Hutchinson. An Efficient Strategy for Rapidly Finding an Object in a Polygonal World. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*, 2003.

[Sus06] O. Sushkov. Robot Localisation Using a Distributed Multi-Modal Kalman Filter, and Friends, 2006.

[Tam12] C. Tam. Developing Agent Behaviour in the SimSpark Simulator, 2012.

[Yus11] Y. Yusmanthia. Simulation and Behaviour Learning for RoboCup SPL, 2011.