Optimisation of a Walk Engine for a Humanoid Robot

by

Brandon Rhys Zahn

A thesis submitted in partial fulfilment of the requirements for the degree of Bachelor of Engineering in Computer Engineering at The University of Newcastle, Australia.

2018

Abstract

This project's purpose was to optimise the humanoid walk engine used by the NUbots team by means of Multi-Objective Evolutionary Algorithms (MOEA). The team lacked physics-based simulation capabilities with the NUbots software and this capability was a fundamental requirement for this project.

Thus, the project starts with the research and development of such tools and middlewares for MOEA optimisation by simulation. This involved firstly establishing new communication protocols between an external robotics simulator and the NUbots proprietory software. Following, a generic MOEA framework that utilised the communications middlewares to run evaluations in a simulated environment was implemented.

To test the robustness of the MOEA framework, a kick script was optimised for maximising kicking distance while minimising torso sway. The resulting kick had approximately doubled the original kicking distance with a 50% reduction in torso sway, indicating that the MOEA framework and the middlewares were successfully realised. Similar results were observed when optimising the parameters of the humanoid walk engine for a faster and more stable walk, with a speed increase of 30% and stability increase of 40%.

The middlewares and communications developed in this project have now become a significant contribution to the tools and utilities used by members of the NUbots team for physics-based simulation and optimisation and can be used for the following years in upcoming international RoboCup competitions.

Acknowledgements

This project has been made possible thanks to all the members of the NUbots team that have assisted me along the way and I would like to acknowledge them here.

Firstly, I would like to thank my academic supervisor, Alexandre Mendes. He has helped and guided me with experience and advise at all key times during this project. Following, I would like to thank the team members Trent Houliston, Alex Biddulph and Jake Fountain, for their invaluable assistance and support for this project.

And finally, I'd like to thank the rest of the NUbots team for providing me with the tools, facilities, guidance, support and access to the NUgus robot to complete this project.

Contributions

My contributions to this project are listed below:

- Researching, developing and integrating the communications middleware between the NUbots proprietory software with the open source Gazebo Robotics Simulator,
- Creating the generic MOEA framework and integrating it into the NUbots codebase,
- And finally developing the model plugins used to work in the Gazebo Robotics Simulator.

Mr. Brandon Zahn

Dr. Alexandre Mendes

All materials used in this thesis have been cited to acknowledge the work of original authors. All figures that are not the original work of the author of this thesis have been reproduced in accordance with Section 107/108 of the United States Copyright Act 1976, the Australian Copyright Act 1968, or the permission of the publishers, and have been cited in the figure's caption.

Contents

List of Figures

List of Tables

List of Algorithms

Nomenclature

DARwIn Dynamic Anthropomorphic Robot with Intelligence

DoF Degrees of Freedom

DSL Domain Specific Language

EA Evolutionary Algorithm

EC Evolutionary Computation

ER Evolutionary Robotics

GRM Gazebo Reactor Module

GRS Gazebo Robotics Simulator

HOP Humanoid Open Platform

HP Head Pan

HT Head Tilt

IgnTrans Ignition Transport library

Ign The Ignition Robotics C++ libraries

IMU Inertial Measurement Unit

Ind Individual

LAN Local Area Network

LAP Left Ankle Pitch

LAR Left Ankle Roll

LE Left Elbow

LHP Left Hip Pitch

LHR Left Hip Roll

LHY Left Hip Yaw

LK Left Knee

LSP Left Shoulder Pitch

LSR Left Shoulder Roll

MOEA Multi-Objective Evolutionary Algorithm

MoI Moment of Inertia

NAT Network Address Translation

NSGA-II Non-dominating Sorting Genetic Algorithm II

OS Operating System

PID Proportional Integral Differential

Pop Population

RAP Right Ankle Pitch

RAR Right Ankle Roll

RE Right Elbow

RHP Right Hip Pitch

RHR Right Hip Roll

RHY Right Hip Yaw

RK Right Knee

ROS Robot Operating System

RSP Right Shoulder Pitch

RSR Right Shoulder Roll

RTF Real-Time Factor

SBX Simulated Binary Crossover

SDF Simulation Description Format

VM Virtual Machine

ZMP Zero Moment Point

1 Introduction

1.1 History and motivation

After several years of study and an invaluable overseas exchange experience, I have had a growing interest in optimisation by evolutionary algorithms. It began with a research project that I undertook while on exchange in Japan in 2015. I was lucky enough to be given the opportunity with Tokyo Metropolitan University's robotics lab and I learnt as much as I could in the matter of a few months time. The project involved optimising the parameters of a proportional-summation-difference controller of a line-tracing robot by a bacterial memetic algorithm, a type of evolutionary algorithm. After a lot of hard work with fellow researchers, I had finalised my first academic research paper [19] that was later peer reviewed and published by Springer for the 9$^{\text{th}}$ International Conference on Intelligent Robotics and Applications, held in Tokyo, Japan.

After returning to Australia in 2017, I was once again given another opportunity to conduct research involving evolutionary algorithms. I accepted an offer to work with Professor Pablo Moscato as a research assistant at the Hunter Medical Research Institute. I was tasked with testing a MOEA framework for community detection. I started to realise how powerful they can be if implemented correctly. During my final year of studies in 2018, I was introduced to the NUbots, a team of undergraduate and postgraduate students working alongside professors at UoN, who compete in the annual RoboCup, an international competition of soccer-playing robots. After joining the team, I found a good opportunity to keep with the trend of optimisation and evolutionary algorithms to further my knowledge on the topic. Thus, I decided on this project.

The results of this project have large consequences for the NUbots team, since no physics-based simulation environment exists at the time of writing and one is required in which the optimisation algorithms shall be run. Once complete, this project will eventually become a major contribution to the proprietary tools and utilities of the team.

1.2 Project description and objectives

The main objective of this project is to optimise the parameters of a walk engine for a humanoid robot for specific fitness functions such as torso stability and walk speed. The optimisation will ideally be done using an evolutionary algorithm that will be investigated in the literature review. An optimisation method such as this requires plenty of iterations, or generations, of tests to be serially executed and would ideally be run in a physics-based simulation environment. Since no such tool exists within the NUbots team, the primary objective of this project is to create the middlewares and tools which the algorithm shall use to run tests in simulation. The igus operating system is encapsulated into a virtual machine with certain hardware configurations in order to emulate the physical constraints of the robot, so the communications protocol with an external simulation environment would be the main focus.

Following, a generic MOEA framework will be developed and integrated into the NUbots software that uses the communications established with the simulation environment. Then, after running some tests using the framework, optimised parameters will be installed onto the real robot to see if the improvements are translated congruously with simulations. To summarise, the project is comprised of the following main objectives, ordered by descending priority and difficulty:

- Setting up a simulation environment and communications with the NUbots' software
- Implementing a generic MOEA framework to optimise the walk engine
- Testing the optimised parameters on the robot to test their validity in the real world

Proceeding this point is a description of the NUbots team along with the team's software architecture. Then Section 2 follows with a literature review on evolutionary optimisation by methods of simulation. Section 3 goes through the preliminary findings with the final communications configuration detailed in Section 4. Section 5 explains the MOEA framework and how it integrates with the middlwares. Sections 6 and 7 present the findings of tests with a Script engine and the walk engine respectively. Finally, Section 8 concludes the report with

possible expansions and future works. Appendix A then follows with the specifications of the igus robot. Appendix B lists all the MOEA framework's input parameters and Appendix C shows the protobuf messages used. Appendices D and E then show additional results analysis and the walk engine parameter values respectively.

1.3 The NUbots team

The NUbots team is part of the Newcastle Robotics Laboratory, an interdisciplinary research and training initiative of several robotics related research teams from many different research areas including computer science, electrical engineering and software engineering. The NUbots team have been an integral part of the lab for more than 12 years.

Before 2017, the team has used the Kid-size Humanoid robot soccer league domain as a well-defined test bed to develop solutions for wider applications in other research and application areas. From 2018, the team has moved on to the Teen-size Humanoid league, which uses the igus robot, shown in Figure 1.1. The NUbots team is constantly developing software for robotic soccer and competes every year in the international RoboCup competition, having won the world title in the standard platform league in 2006 and 2008. RoboCup is an international research and education initiative with the intent to foster artificial intelligence and robotics research by providing a standard problem where a wide range of technologies and concepts can be integrated and examined in comparison to other teams. The ultimate goal of the RoboCup initiative is stated as, *"By the year 2050, develop a team of fully autonomous humanoid robots that can win against the human world soccer champion team"*.[1] RoboCup counts as one of the most significant events of its kind, with each annual

Figure 1.1: The NUbots' igus robot

[1] Kitano et al. (1997)

RoboCup involving about 3000 people from about 40 countries and a world-wide support of over 100,000 people.

1.3.1 The igus Humanoid Platform and ROS

The robot being used for this project is the NUbots' igus robot, which is based on the igus Humanoid Open Platform, a child-sized 3D printed open-source robot.[1] The igus robot was based on the smaller framed humanoid platform DARwIn-OP.[9] Since 2017, the team have moved on to the bigger igus robot from the DARwIn-OP, for participation in the RoboCup's Teen-sized Humanoid League. The hardware, electronics and sensors of the igus Humanoid Platform are centred around an Intel i7-5500U processor. It runs with a 64-bit Linux-based OS, Ubuntu, and also maintains all of the robot control software. The DC power is provided via a power board and a 4-cell lithium polymer battery can be connected.

The PC communicates with a Robotis CM730 subcontroller board, whose main purpose is to electrically interface the twelve MX-106 and eight MX-64 actuators, which are all connected on a single Dynamixel bus. A summary of the hardware specifications of the robot is shown in Table A.1. The images captured by the camera at 30 Hz are converted into a HSV colour space and then processed. For the target application of soccer, vision processing tasks include field, ball and goal detection, shown in Figure 1.2.

The software running in the PC is a middle-ware called the Robot Operating System[2] (ROS) and comes default in the igus Humanoid Open Platform. This fosters modularity, visibility, reusability and to some degree it offers independence of platform. An overview of the ROS architecture is shown in Figure 1.3a. It was developed with humanoid robot soccer in mind, but the platform can be used for virtually any other application. This is possible because of the strongly modular way in which the software was written, due to the natural modularity of ROS and the use of plugin schemes.

[2] ros.org

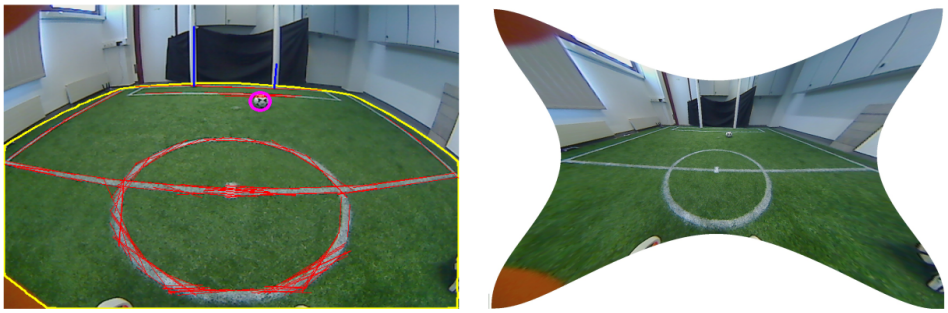
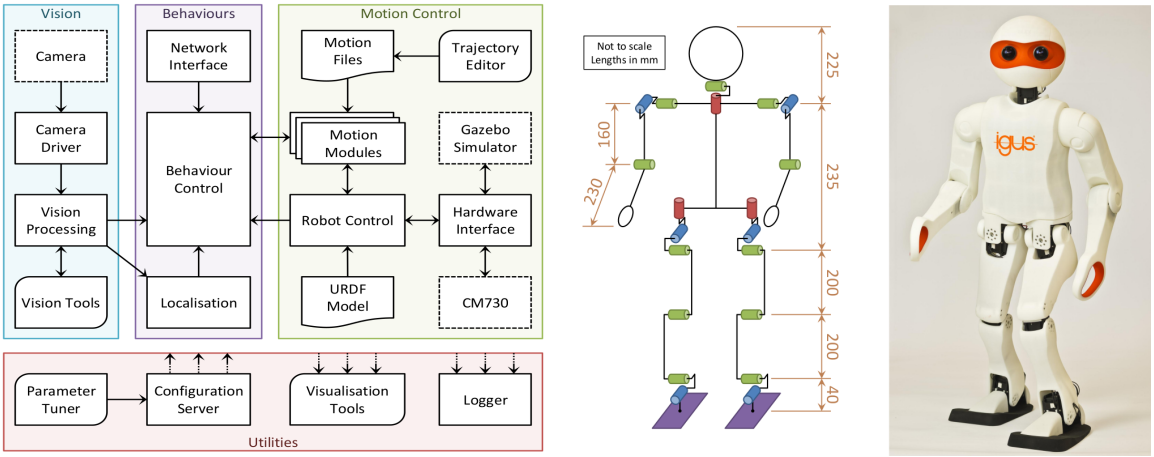


Figure 1.2: A captured image from the igus vision system, with the processing tasks highlighted in red, yellow and pink with the raw image on the right



(a) Architecture of the ROS software

(b) The igus dimensions

Figure 1.3: The igus Humanoid Open Platform

1.3.2 The NUbots' software architecture

The default ROS package that comes with the igus Humanoid Platfrom is replaced by the NUbots' original proprietary software architecture along with NUClear, a co-messaging system that is a fundamental component. NUClear is a software framework designed to aid in the development of real time modular systems and is built from a set of C++ template meta-programs which control the flow of information through the system. These meta-programs reduce the cost of routing messages between modules, resulting in faster communication than other similar systems. Since NUClear utilises a co-messaging system, it allows for simple event callback functions through an expressive domain specific language (DSL). The DSL is

highly extensible and provides several attachment points to develop new DSL keywords as needed. NUClear has been successfully applied in several projects for robotics and virtual reality and also in the igus Humanoid Platform. [10] The NUbots' architecture is based on a blackboard architecture and is shown in Figure 1.5a. It is a form of global store architecture where modules within such a system communicate with each other through the manipulation of data elements stored on the blackboard.

The NUClear framework is designed to utilise the advantages and address the problems that exist with the architectural styles of message passing and blackboards. The most significant components of the system are interpreted as the PowerPlant, Reactors, Reactions and Tasks, illustrated in Figure 1.4. The main components are described below:

The PowerPlant can be interpreted as the central messaging system used by Reactors for communication. Whenever a Reactor emits data, the PowerPlant takes control of the data and executes any Reactions which are subscribed to the message. Since NUClear is a multithreaded environment, the PowerPlant handles all threading logic and is also responsible for assigning Tasks to threads. The transparent multithreading uses a threadpool with enough threads to saturate every CPU core in the system, which allows for Tasks the freedom to execute on different threads.

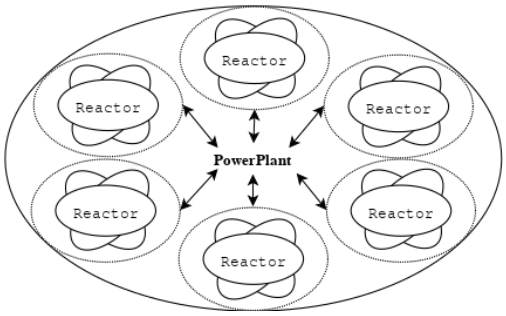


Figure 1.4: The nuclear reactor interpretation of the NUClear framework

Reactors can be considered as a module, as there are modules in ROS. All modules in the NUClear framework are an extension of the NUClear::Reactor namespace. During the Startup Process, the Reactors are installed into the PowerPlant and are then primarily responsible for two functions. Firstly, defining the Reactions and conditions of execution and secondly, emitting data to the PowerPlant.

Reactions provide definitions of the Tasks that need to run when data or conditions that are required for the reaction become available. Then a Reactor can use the functionality provided by the NUClear::Reactor namespace to set up a reaction and subscribe to the

PowerPlant. These functions are bound by the NUClear framework as lambda functions, known as callbacks. The execution of these callbacks are to then assign Tasks to a thread. Subscriptions to the PowerPlant are made using DSL On Statements. The conditions for a request can be defined using the keywords Trigger, With, Every and Always.

Tasks represent the singular executions of a defined reaction within the system.

The primary advantages of the NUClear framework are the high data availability in a blackboard system and the excellent decoupling properties of a message-passing system. A message-passing system must be used as the primary architectural paradigm to achieve loose coupling. Figure 1.5b shows the basic use of NUClear. The latest version of each message is stored and shows a module requesting a primary data type, along with the most recent version of a secondary data type. When these primary data are generated, the most recent copy of this co-message is bound into the callback. This affords the module a more expressive interface for gathering messages.

1.3.3 The NUClear co-messaging framework

NUClear uses keywords to implement the lambda callback functions, where the keyword would determine how the function should be executed. Some common keywords are given below: **on** `on<...>(runtime...).then(function)` is the wrapper for every subscription in NUClear and is used to wrap the template descriptions of the subscription's purpose. Other DSL words are entered as template arguments to this function, with any runtime arguments passed as function arguments.

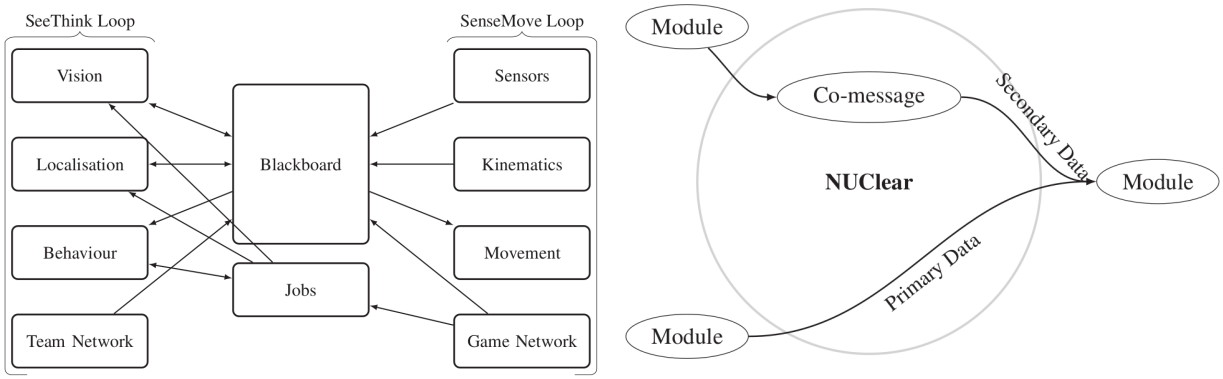
emit `emit(message)` handles the publish part of the system by taking data and forwarding it to the functions that have subscribed to it, which is routed at compile time.

Trigger `on<Trigger<Type>>()` statements set up callbacks and execute when the type is emitted by flagging the used data type as a triggering primary data type. When executed, it will pass the data that was emitted.

With `on<Trigger<TypeA>, With<TypeB>>()` describes the additional information used by a callback. The provided function will not be executed when these data are emitted. However, when this function is executed, the latest copy of this data will be provided.

Startup and Shutdown `on<Startup>()` and `on<Shutdown>()` functions with this word will execute at startup and execute at shutdown, respectively.

Configuration `on<Configuration>("File.yaml")` allows a program to watch a configuration file in a directory and be provided with the latest version when it changes.



(a) The overall architecture

(b) The NUClear co-messaging system

Figure 1.5: The NUbots' software architecture

1.4 Development software and Gazebo

Referring to Figure 1.3a, it can be seen that the ROS architecture has built-in compatability with Gazebo, an open-source physics-based robotics simulator[3]. It is very popular with the simulation-league of RoboCup, being utilised by all teams to run their robots in simulations. A lot of support is available online to help with any ROS-based robotic system to integrate smoothly with the Gazebo simulator. This, however, is not the case if any proprietary software, such as NUClear, is being used. Since Gazebo has such strong connections with RoboCup and resources such as 3D models and data for the igus Humanoid Platform are readily available, it would be a logical decision for both the team and this project to choose

[3] gazebosim.org

Gazebo to integrate with the team's software architecture and NUClear.

Integrating ROS into the NUbots' software architecture has been opposed by the team for numerous reasons, but mainly for the fact that both systems perform the same job as robotic operating systems. Doing so would force NUClear to operate in a multi-processed architecture and it's DSL is extensible whereas ROS is not as modular. An example is the vision module of ROS, which is comprised of a single node. NUClear's vision modules, in contrast, consist of many submodules for various object classifications, such as ball, goal-post, field detection and localisation, meaning that modifications are easily implemented. Another setback would be a language incoherency between the two systems with ROS forcing the C++ '07 standard, whereas the NUbots' software is based on the C++ '14 standard. Basically, there are no benefits to integrating ROS with NUClear and the number of setbacks and package dependancies introduced to the NUClear system would hinder it's performance drastically. Consequently, an alternative method of integration and communications protocol between Gazebo and NUclear will be investigated to create the required simulation environment for the team.

The main development software programs being used for the project are the NUbots' software and the NUClear framework, the Gazebo robot simulator and one of it's dependent packages, the Ignition Transport library[4]. Other tools include CMake[5], Vagrant[6], Oracle's VirtualBox[7] and Git version control[8].

The ignition Transport library is a dependent package of Gazebo, providing the simulator with an intra and inter-process co-messaging communication protocol for robot simulation, which works in a similar way to NUClear. After investigating the best method, it has been selected for the communications between NUClear and Gazebo. It would require soft real-time synchronicity and should be reliable even in the case of lost data, since communications would be inter-process through TCP protocol across the local network. Keeping with the modularity of NUClear, the Gazebo extension could be built upon in the future, to create even more immersive simulations.

[4] ignitionrobotics.org
[5] cmake.org
[6] vagrantup.com
[7] virtualbox.org
[8] git-scm.com

2 Evolutionary Computation

Robotic walk optimisation can be classified under the research field of Evolutionary Robotics (ER), where evolutionary algorithms (EA) are applied to solve optimisation problems in the branch of robotic systems, in the case of this project, bipedal locomotive robotics.[8] Following is an introduction to the field of ER, some examples of recent research on multi-pedal gait optimisation and then some issues relevant to the reality gap.

2.1 The research field of Evolutionary Robotics

Darwin's theory of evolution provides the fundamental foundation for the research field of ER, which endeavours to transfer the efficiency and richness of living organisms to robotic systems. The main goal of ER research is to create a process that is able to design and build an optimal robot, given only the specification of a task. Then by doing so, it would allow a robotic system to exploit the non-linear dynamics offered by its structure and surrounding environment without having it be modelled explicitly. Darwin's theory provides the best source of inspiration for ER, since nature has demonstrated its efficiency over many millennia. So EA are designed in such a way to best mimic the evolutionary process and then are integrated into robots to find optimal control. ER has its roots in Evolutionary Computation (EC), where genetic algorithms formed the basis of currently used EA.

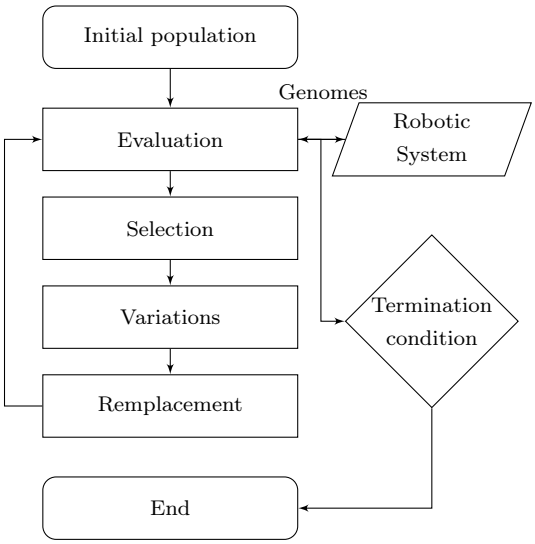


Figure 2.1: The standard setup of an evolutionary algorithm for ER

Figure 2.1 shows the standard configuration of the evolutionary process. The initial population consists of randomly generated individuals. Each is then evaluated based on some set fitness functions. Based on the outcome of this evaluation, individuals are selected to move on to the next generation, from which a new population is generated using two kinds of variation operators, mutation and recombination. Mutation involves creating a new individual as a modified clone of a previous one. Recombination involves creating a new individual by merging several individuals from the previous generation. The evolutionary process goes on until a termination criterion is reached, such as a specific number of generations. From the viewpoint of the EA, the interaction with the robotic system is seen as a generic black-box that is used to evaluate a set of parameters to a fitness value of a genome.

EA are used in a few key ways during the creation of a robot system, including parameter tuning, evolutionary aided design, online evolutionary adaptation and more recently automatic synthesis. Since the objective of this project is to optimise a walk engine, parameter tuning will be required and thus focused on in discussion.

2.2 Simulation-based optimisation

Optimising the gait of a walking robot requires the calibration of the controllers that control walking. There are several techniques to do so, such as simulation-based and robot-based parameter tuning. Combining these two techniques should make finding optimal solutions easier. Figure 2.2 shows how a simulation-based EA is implemented. Parameter tuning occurs based on simulations. The core of the EA produces parameters that are used to update some robotic controller. After some conditions are met, such as a time limit, the simulator ends its test then sends the results to the EA core as a fitness score. From this point, a generation is passed and new parameters are generated. Then, after some termination condition, the simulated optimal parameters are transferred to the real robot and tested for validity.

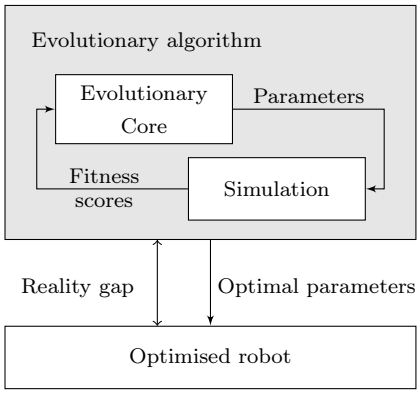


Figure 2.2: Tuning parameters with simulation-based EA

There are many possible objective functions for the simulation to test fitness scores. In some cases, multi-objective algorithms are used to find a pareto-front of solutions. One famous algorithm is the NSGA-II algorithm.[6] It is a multi-objective genetic algorithm used widely in many different fields of research including community detection. It has also been used in gait optimisation experiments for speed and stability as objective functions. The number of objective functions used to evaluate solutions can also be set to single-objective.[15]

There is a C++ framework designed to help researchers in the field of EC to make their code run as fast as possible on a multi-core computer.[13] It is called the Sferes framework and has seen use in ER for walking robot controllers.[4] This framework has several components. Firstly, the Sferes framework, which consists of several evolutionary algorithms, genotypes, fitness functions and others. Secondly, optional modules that may be experimental or require complex dependencies and thirdly, the user experiments that may include new genotypes, fitness functions and so on. The Sferes framework could be implemented to run with the rest of the system. It could also provide useful guidance when trying to implement the EA core, since several EA are already integrated into this framework, including NSGA-II.

2.3 Bridging the reality gap

Especially relative to this project is the reality gap. It refers to EAs being run in simulations during the optimisation process, which may lead to optimised controllers that cannot be applied to the real target robot. When a robot is simulated, real-world phenomena such as lighting and friction are approximated by software models which in most cases are over simplified interpretations of physical mechanics. This can cause the robot to rely on badly simulated behaviours which do not translate well to reality. This error, or gap, can then be exemplified in size after several thousand generations of an optimisation process using EA, resulting in a controller that is over optimistic about its environment. In some cases, this can lead to controllers that are even worse than the starting generation when applied on the real robot.[2] [16]

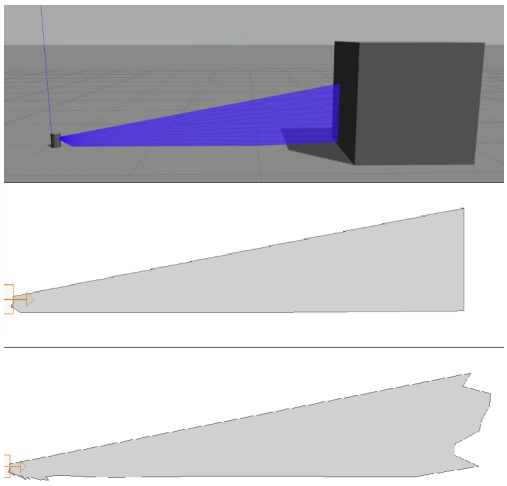


Figure 2.3: Gaussian noise being added to a laser distance sensor

A perfectly accurate simulation is implausible at this time, but some techniques can reduce the reality gap in simulation approximations. One technique is adding artificial noise to sensors. In reality, almost every sensor has output noise, such as lasers reading incorrect distances. Gazebo has built-in noise models that can apply some sort of noise to most sensors. Gaussian noise serves as a good basic interpretation of noise. In Figure 2.3, a laser distance sensor in the Gazebo simulator has a Gaussian noise model appended to roughen up the boundaries of detected distances, creating a more realistic, irregular result. Another technique to reduce the gap consists of estimating how a controller transfers to reality, based on several preliminary experiments on the real robot. Then the results can become an objective to approach solutions that do transfer well to the real robot.[12] Overcoming this gap could be one of the major challenges of the project, as not all simulations provide accurate results in ER. To help with making the bridge across the gap, there are some specific criterion to meet.[14] These requirements include:

Computation speed and Startup time Most ER experiments spend most of their time in evaluating fitness functions, meaning running the simulator for many time-steps. It is then critical that simulations are as fast as possible. The fitness function will be evaluated millions of times, therefore starting a new simulation should take as little time as possible.

Fully re-entrant and thread-safe A common strategy to reduce the duration of an ER experiment is to parallelise the fitness evaluations. However, to do so, the simulators need to be programmed without any hidden state in the form of static or global variables, and allow many instances to run at the same time, ideally in the same process.

Stability and accuracy Evolution, like other optimisation algorithms, tends to push a simulator to its limits and over-exploit bugs and instability. It is common in ER to observe simulated multi-body robots explode, where constraints of joints break. This is due to the main body travelling at a high speed. More accuracy is better and reduces the reality gap.

3 Preliminary Findings

As mentioned in the introduction, the NUbots team had no physics-based simulation environment for the NUClear framework so as a result, an appropriate method of integration and communication with the Gazebo robot simulator needed to be investigated before the project could advance. In this section, the findings that led to the final configuration are introduced and discussed, focusing on the communication protocol between the two systems.

3.1 Fundamentals of model control in Gazebo

The fundamentals of how models are controlled in Gazebo had to be understood before moving on to communications. The goal was to create a simple environment representative of the RoboCup soccer field, consisting of a field, two goal posts, a ball and most importantly, an igus robot. The robot model would be controlled in some way by an external application running in a separate process. Doing this would mimic the basic configuration of how the NUClear framework would interact with Gazebo, besides the communication protocol.

Gazebo uses model plugins to control models from external sources. In the plugin, anything related to the model can be manipulated. Gazebo uses its own interpretation of the Ignition Transport library, under the gazebo simulator namespace. This means that topics can be advertised in one process and then subscribed to in another, as long as the two processes are running on the same machine. When a topic message is recieved by a subscribe node, it initiates a callback function to do something with the recieved message. In this case, the plugin was programmed to subscribe to an advertisement coming from an external process about a particular topic and apply a velocity to one of the joints of the igus model when some message was recieved via a callback function.

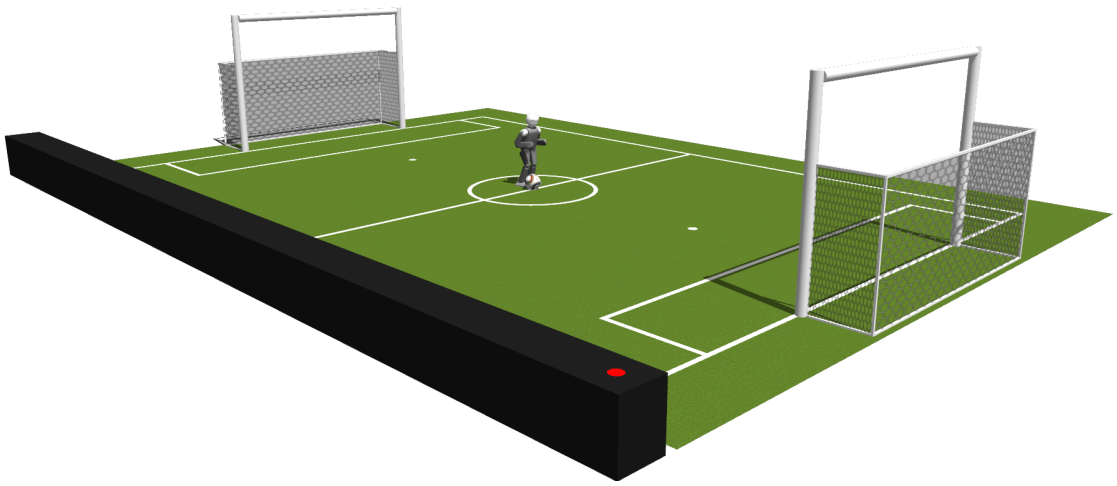


Figure 3.1: A screen shot of the environment in Gazebo, with two goals, a ball, the field and the NUbots' igus robot in the centre

Figure 3.1 shows the models instantiated in the simulation and Figure 3.2 shows the various components that make up the Gazebo plugin and RoboCup soccer environment. The `environment.world` file describes the environment and how to instantiate the models. It also creates a link between the igus model and the NUbots' igus plugin library. Then the plugin file tells the main Gazebo process to load the plugin's library along with the model itself. The models have two important components, the meshes and the SDF file that describes the physical properties of joints and how the they should link with one another.

The Gazebo Transport namespace has a `Node` class that implements the TCP communication protocol by using the `Advertise()` and `Subscribe()` public member functions of the instantiated node. Once an advertise message is recieved by a node that is subscribed to the "*foosball*" topic, the callback function is called. The node does not have the ability, however, to communicate with any nodes that exist in different domains or machines.

This is where the main issue for communications begins. The NUbots' software is run in a virtual machine that encapsulates the system completely from its host so that from the perspective of the virtual machine, it exists as a physically separate machine. So using a transport node from the Gazebo namespace to try and communicate would not work. Also, since the virtual machine has no user interface, there is no need to integrate all of Gazebo, ideally just a subset of its dependancies should provide the right tools.

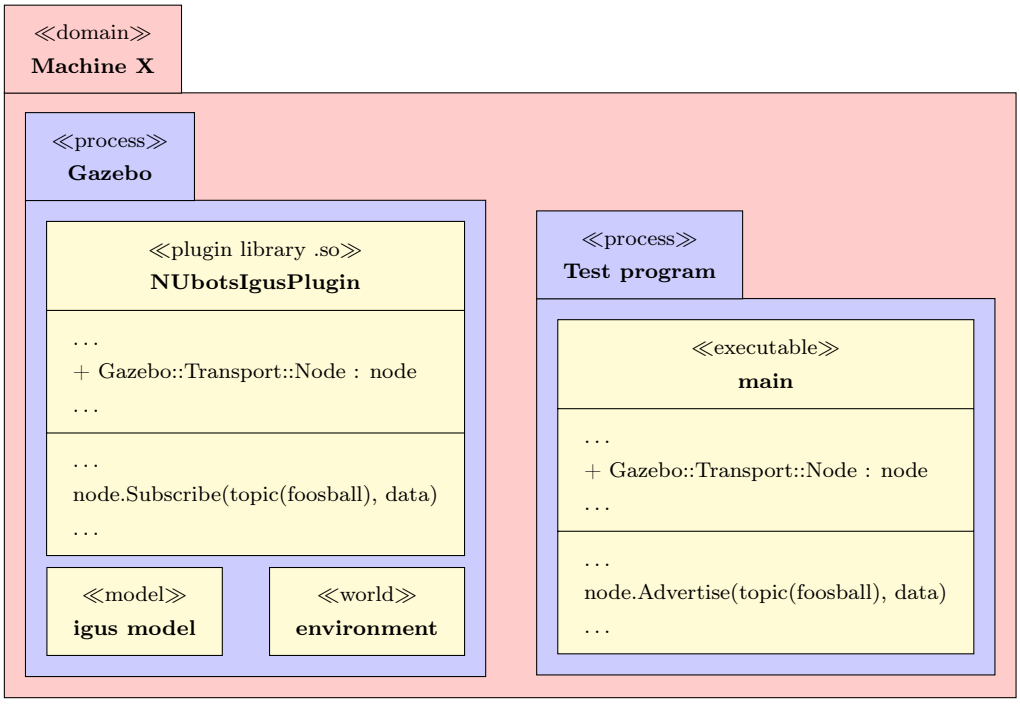


Figure 3.2: The main components of a Gazebo plugin using nodes to communicate with inter-process programs, running in the domain of a single machine named X

So after experimenting with the external application and successfully moving the igus model by an external input, the basic requirements on model control were understood. To completely control a model would involve expanding the same concept to all joints in a model. This process is reasonably well documented online and not too technical to implement. However, this control cannot occur if there is no working communication. Thus, the next step would focus on integrating NUClear with Gazebo, starting with the Gazebo development libraries, then investigating an appropriate communication protocol.

3.2 Integrating Gazebo with the NUClear framework

Integrating Gazebo with the NUClear framework involved a trial and error approach, until a configuration that was able to compile using the NUbots' toolchain was found. Since this is an experimental process, reducing overhead is possible at a later time, such as removing unnecessary packages.

When initially installing Gazebo from Ubuntu's online repositories and trying to run it inside the virtual machine, compilation errors were occurring due to a package conflict. The culprit was one of the dependencies of Gazebo, google's Protocol Buffers package[1], a language-neutral, platform-neutral extensible mechanism for serialising structured data. The reason was that a newer version of the same package was already installed in the toolchain. The `.proto` files are required to be compiled by a protocol buffer compiler and if a file is used by a differing version of the Protocol Buffers package, a version error will be thrown.

To solve this compilation error, the Gazebo package had to be built from source. Doing so would give full control over what packages are installed. A shell executable file was writen to achieve this by firstly downloading packages required for Gazebo, with the exception of the Protocol Buffers package from Ubuntu's repositories. Then setting up path variables using CMake to integrate Gazebo with the NUbots' toolchain. Then finally building Gazebo using the NUbots' version of Protocol Buffers. After completing this procedure, it was then possible to include code from Gazebo's development source files when compiling binaries with NUbots' software and NUClear. Compilation configuration was done by using CMake, which gives the user control over how compilers compile binaries. In this case, it was used along with the shell executable file to set the configuration paths for installs and during compilation to be linked to the NUbots' toolchain configuration path directory, so that when trying to compile and install, the Protocol Buffers package in NUbots' toolchain would be used.

It then became possible to use Gazebo source code in the virtual machine. A new module was made within NUClear for the purpose of communicating with an instance of the Gazebo simulator running externally on the host of the virtual machine. The structure of the Gazebo module is much the same to other modules in NUClear. Firstly, a Gazebo class is declared that has a `<nuclear>` include, along with any member functions and variables. Then, in the class declaration, a Reactor is declared that includes all of the lambda callback functions controlled by NUClear's DSL keywords. Within the Gazebo Reactor, there are four primary DSL callbacks declared:

on `<Configuration>("Gazebo.yaml").then([this](const Configuration& config)` to configure the module with configuration settings that are found in the `Gazebo.yaml`. This can be used to change settings in realtime for the module, since this callback function

[1] developers.google.com/protocol-buffers

is called each time there is an update to the `.yaml` file. The Gazebo module's Member functions and variables can be referenced to in this function, since there is reference passed that points to the instantiated object where the function was declared using the `this` keyword.

on `<Startup>().then([this]()` to set up the module when NUClear starts, such as instantiating member variables.

on `<Shutdown>().then([this]()` to be used when NUClear is shutting down, such as to perform any cleanup of dynamically allocated memory.

on `<Trigger<Sensors»().then([this](const Sensors& sensors)` to retrieve sensory data and send it to the simulator. Since the `Trigger` keyword is being used, each time the sensors are updated, this function will be called with updated sensor values. The sensory data to be used will mainly be servo positions and commands.

Now, the Gazebo Reactor module had the right foundation for setting up a communication protocol. The next task was to choose the appropriate tools to work around the situation shown in Figure 3.3. Now an inter-domain communication was required, meaning that a connection through the network between the simulator and the Gazebo Reactor in NUClear was required. Since Gazebo and NUClear already use TCP for communications, any tool using the same protocol would seem appropriate.

3.3 Establishing a communication protocol

Including the Gazebo development libraries in the Gazebo Reactor module brought along many dependancies that had to be discoverable in the virtual machine. Another route was decided upon after experiencing many compilation errors when trying to include these development libraries. The errors came from the Gazebo program using its own Protocol Buffer compiler, which caused version conflicts again with NUClear.

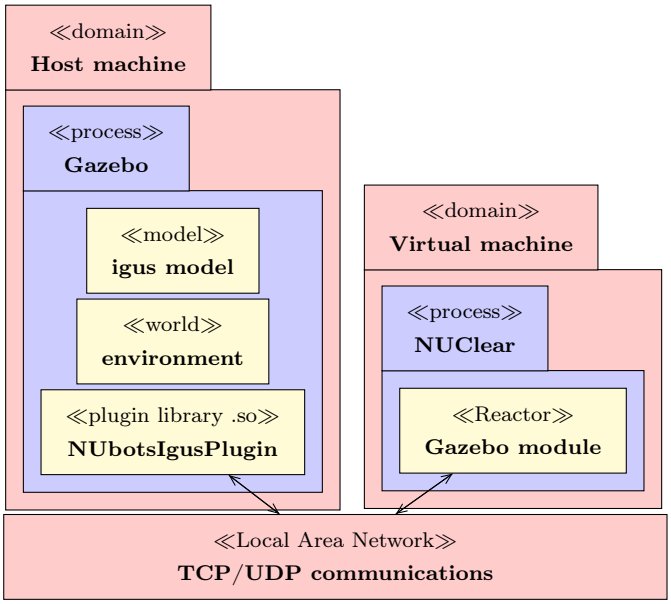


Figure 3.3: The conditions of inter-domain communication between host and virtual machine

While searching for useful packages that were integrated, some helpful libraries that originated from within Gazebo were discovered, shown in Figure 3.4. These libraries are evolutions of libraries which were at a stage built within Gazebo itself. The developers have extracted them from the main package to make the functionality offered by these libraries available for other projects and to make Gazebo more modular. These features are being made accessible through the set of libraries under the Ignition name.

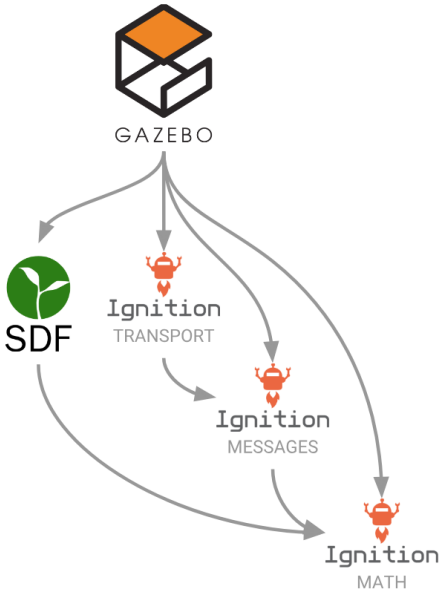


Figure 3.4: Some dependencies that originated from Gazebo

Ignition Transport is an open source communication library that allows sharing data between nodes, which could be running within the same process in the same machine or in machines located on the same network. It discovers, serialises and delivers messages to the destinations using ZeroMQ[2]. It also uses the Protocol Buffers package for data communication, but since it was built using the NUbots' version of Protocol Buffers, no version

[2] zeromq.org

conflicts occurred when using this package within NUClear. It seemed to fit the requirements very well for setting up the communications. Initially, ZeroMQ was investigated to check if it would be suitable for use but was tedious and somewhat complex to initialise. So it was decided to use the Ignition Transport library, since it was simpler to understand and originated from Gazebo.

3.4 The Ignition Transport library

Finally a lightweight communications tool had been discovered and could be easily integrated into NUClear by simply including the development libraries in the source code of the Gazebo Reactor module. It works in a similar way to the nodes from the Gazebo::Transport namespace in Figure 3.2. A class diagram of some useful classes is shown in Figure 3.5. Some key concepts and components of the Ignition Transport library are introduced below:

Topics are a name for grouping a specific set of messages or services, such as the "*foosball*" topic shown in Figure 3.2. Nodes that subscribe to the same topic will receive messages when they are broadcast by the topic's publisher. A topic can be advertised with a scope, which allows the user to set it's visibility. The three scopes available are *PROCESS*, *HOST* and *ALL*. The default scope is *ALL*.

Nodes are the main data structure and allow communication to follow a purely distributed architecture, where there is no central process. All nodes in a network can act as publishers, subscribers, provide services and request services. Publishers are nodes that produce information and subscribers are nodes that consume information.

Messages are one of the two methods of communication where a publisher node advertises a topic and then publishes periodic updates. Subscriber nodes can then subscribe to the topic and receive these updates.

Services are the other method for communication where nodes subscribe to the same topic and register a function that will be executed each time a new message is received. Services have two main components, a service provider and a service consumer.

The physical location of services are hidden and cannot be seen or retrieved directly by subscriber nodes. There is a discovery service in the library that is in charge of discovering and maintaining an updated list of services available on the network, that can help nodes in different processes or machines find each other. This service is described in the following Section.

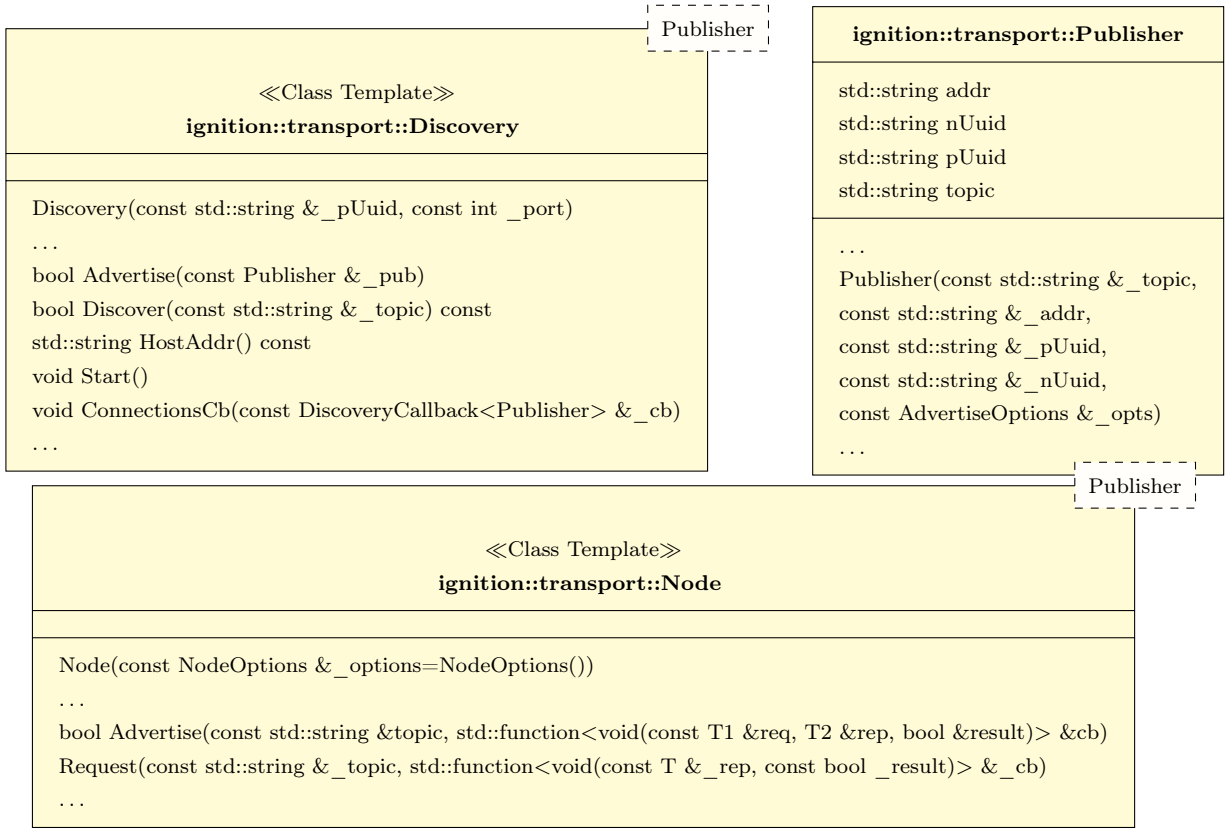


Figure 3.5: A class diagram of some Ignition Transport classes and some regularly used member functions

3.5 The Discovery service

The Discovery service is responsible for keeping an updated list of active topics ready to be queried by other entities and to discover other nodes and topics in a network. It learns about the network interfaces available and creates a container of ZeroMQ sockets, one per local IP address. These sockets are used for sending discovery data over the network and reaching other potential Discovery instances. When outbound data is pending from a publisher,

the list of sockets are iterated through and the message is sent over each one, flooding all interfaces with a discovery request. When a request is recieved by another Discovery instance, it gets all the location information of the topic's publisher. The internal architecture of the Discovery service is illustrated in Figure 3.6. Three main types of nodes are shown:

Node is the main interface with the user. The Node class contains all the functions required for communication between nodes.

Shared is a singleton instance of a NodeShared class and is shared between all the Node objects running inside the same process. The NodeShared instance contains all the ZeroMQ sockets used for sending and receiving data for topic and service communication to share resources between groups of nodes.

Discovery is a service layer required in each process to learn about the location of topics and services, since topics and services don't have any location information. Discovery uses a custom protocol and UDP multicast for communicating with other Discovery instances. These instances can then be located on the same machine or different machines over the same network.

The initialisation procedure for the Discovery service consists of the following steps:

1. Firstly, a Discovery instance needs to be instantiated by using the Discovery class constructor, which requires a UDP port for the discovery sockets and a Universally Unique ID (UUID) of the process in which the Discovery instance is running. This UUID will be used when announcing a local topic.
2. Then, the `Start()` function is used to create an additional internal thread that sees to updating topics by receiving discovery messages. This thread also answers with an `ADVERTISE` message when a `SUBSCRIBE` message is received and there are local topics present in the same process.
3. The `Advertise()` function is then required to register a local topic and announce it over the network and `Discover()` is used to learn about a given topic immediately.
4. Finally, to avoid blocking any other threads, the notification of discovery events is

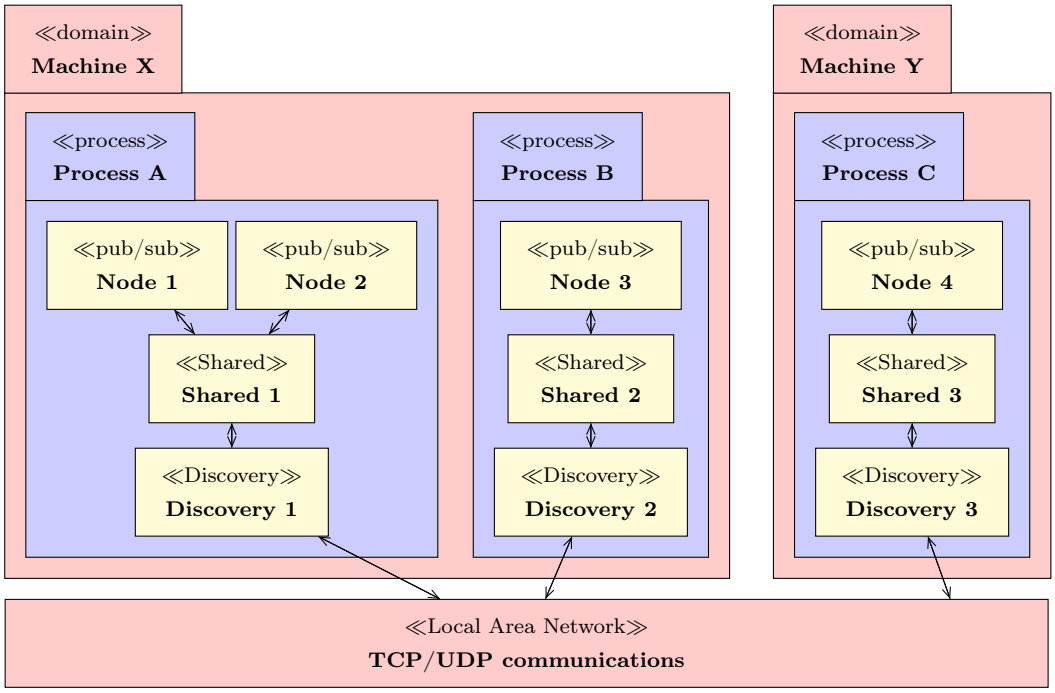


Figure 3.6: The internal architecture of the Discovery service and how it interacts with IgnTrans nodes in different processes and domains

achieved by using callback functions, one for receiving notifications when new topics are available and another for notifying when a topic is no longer active. The functions `ConnectionsCb()` and `DisconnectionsCb()` set these two notification callback functions.

After working with the library for several weeks and implementing the Discovery service with the above process, a connection was finally established between the Discovery layers of NUClear and the Gazebo simulator. With this accomplishment, this library has shown to be the appropriate communication tool required for this project.

From this point onward, the services and messages advertised by nodes need to integrate correctly with the Discovery layer, which would then allow for true inter-domain communication. Then, once these two elements are working together, the data being sent and requested by the service can be customised to contain all joint information about the NUbots' igus robot model in the simulator. This would then allow NUClear to control the robot model, request updates on the model's position and feed them back into the NUClear system.

4 Communications Summary

This Section describes in more detail the working communications configuration for this project. It also serves as a guide for members of the team that wish to use Gazebo with NUClear and improve the Gazebo Reactor module. An online repository[1] has been created in the NUbots GitHub where anyone can find these files and use them. The NUbots codebase repository[2] contains all the required source code for setting up the VM. The configuration for communications with GRS and the VM is shown in Figure 4.1.

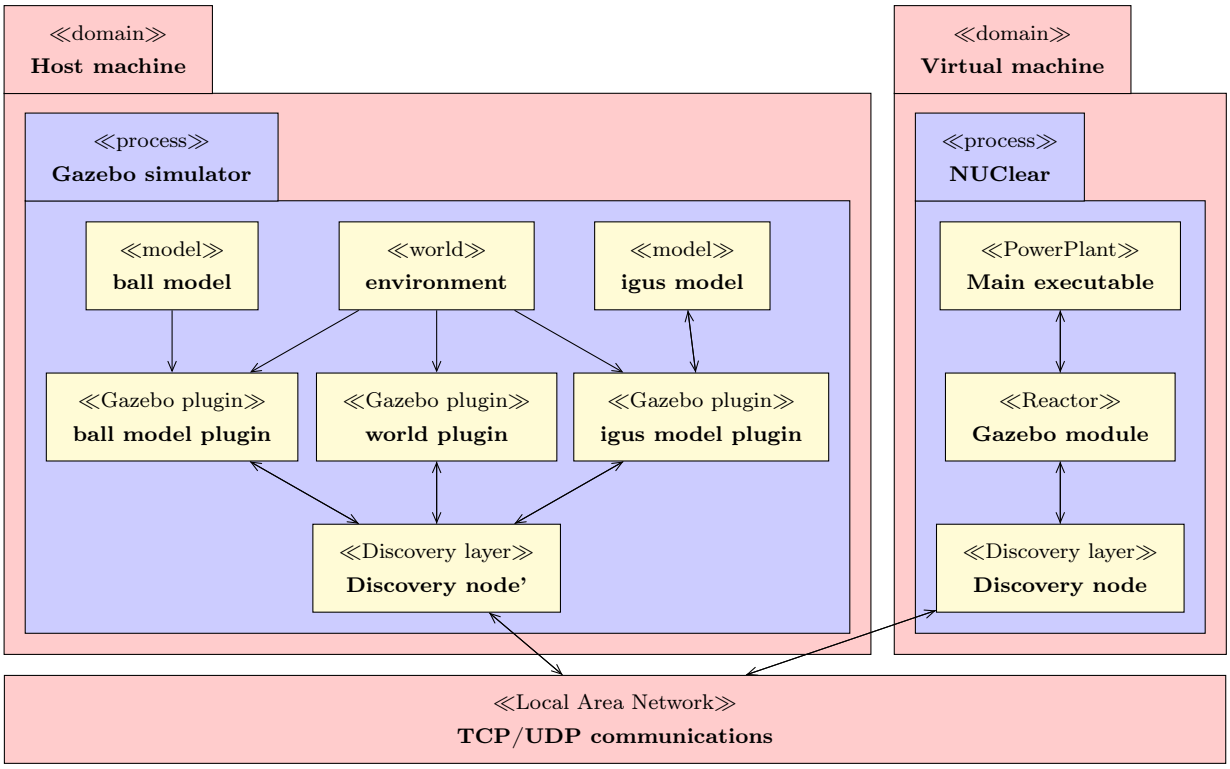


Figure 4.1: The configuration for communications with GRS

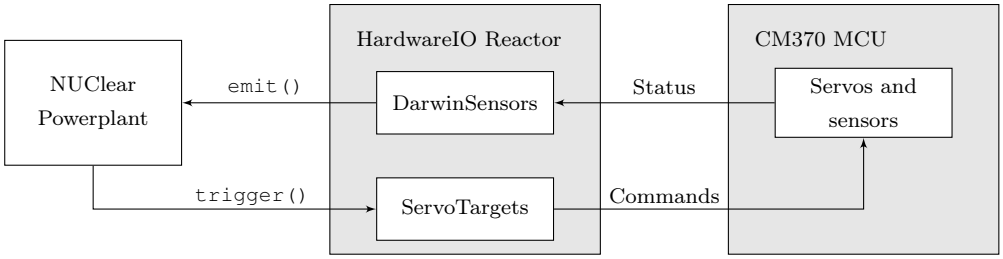
Following, the Gazebo Reactor module will be described and then the model plugins. Lastly, some networking issues that were encountered are presented. The source code of both the Reactor module and the plugins can be found in the NUbots team GitHub repository.

[1] github.com/NUbots/Gazebo
[2] github.com/NUbots/NUbots

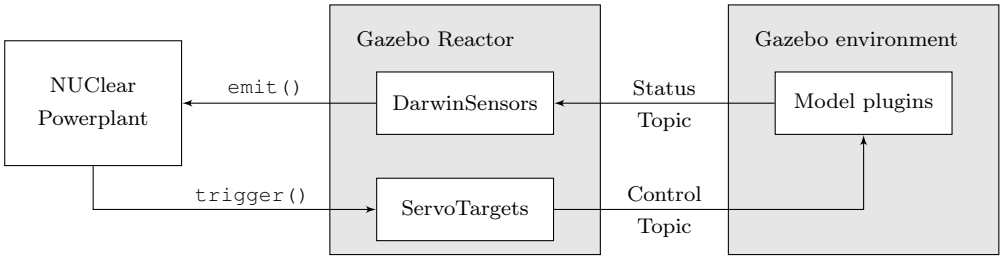
4.1 The Gazebo Reactor module

The Gazebo Reactor module follows the design of other simulation modules in the NUbots codebase. The general goal of these modules is to receive servo commands being emitted from some other modules and then replace the data that would usually come from the robot's sensors with simulated data. In the case of operating the real robot, a Reactor module named HardwareIO would emit a struct called DarwinSensors to the rest of the system, containing the robot's joints and sensor status data. It also triggers on ServoTarget messages to send commands to the robot's servos. It exchanges data to and from the CM370 micro-controller on board the igus robot, as in Figure 4.2a. Each of the servo commands in ServoTargets would then be transferred through the micro-controller to the appropriate Dynamixel servo.

In the case of this project, the Gazebo Reactor module takes place of the HardwareIO module and sends and retrieves data to and from the simulated model in GRS, shown in Figure 4.2b, including some extra plugins for ball and world information. It emits and triggers in much the same way as the HardwareIO module, emitting joint status data with DarwinSensors messages to NUClear and triggering on ServoTarget messages to move a joint as required. The emission of DarwinSensors is initiated by the Status topic callback function.



(a) The HardwareIO Reactor module used to interpret hardware data



(b) The design of the Gazebo Reactor module

Figure 4.2: Comparison of the Gazebo module with the HardwareIO module

The Gazebo Reactor module contains some fields that can be edited by the user to change the way that Ignition Transport communicates. These fields are available in the configuration `.yaml` file of the module and are listed in Table 4.1. It is used in the `on<Configuration>("Gazebo.yaml")` to set up the module during runtime. The following is very important and easy to miss: *The partition and namespace fields of nodes must be equal to the partition and namespace fields given to any other nodes using the same topic in order for the nodes to communicate using the Discovery service. Also, these two fields must have some value and* ***cannot be blank***. If no namespace or partition is assigned, they will not discover each other. The control address of a message publisher is used by any subscriber nodes to notify the message publisher about a new subscription. The IP address that IgnTrans uses should be set to whatever is being used by the virtual machine's LAN network device, usually device *enp0s3*. This topic is explained in more detail in Section 4.4.

Table 4.1: The Gazebo Reactor's configuration fields

Field	Example	Description
IGN_PARTITION	NubotsIgus	environment variable used to define partition names
IGN_IP	10.1.0.92	environment variable used to force a single IP address
Host address	10.1.0.92	The ZeroMQ address of the host
Control address	10.1.0.92	The ZeroMQ control address
Status topic	NubotsIgusStatus	The topic name used for joint status data
Control topic	NubotsIgusCtrl	The topic name used for joint commands
World status topic	NubotsWorldStatus	The topic name used for world data
World control topic	NubotsWorldCtrl	The topic name used for world commands
Ball status topic	NubotsBallStatus	The topic name used for ball status data
Node namespace	NUbots	The namespace given to all nodes in the module
igus nodes partition	Igus	The partition name given to the igus nodes
World nodes partition	World	The partition name of the World nodes
Ball node partition	Ball	The partition name of the ball status node

The module and its communication protocols are shown in Figure 4.3. The messages that It consists of a few NUClear keyword functions:

on`<Configuration>("Gazebo.yaml")` takes the parameters defined in Table 4.1 to set up five IgnTrans nodes; igus control, igus status, world control, world status and ball status. It then creates a new Discovery node and sets up the connection callback functions

and starts the Discovery service. Then it advertises the control nodes to the control topics and subscribes the status nodes to the status topics.

on`<Trigger<ServoTarget>, With<DarwinSensors> >()` triggers on a Servo Target message with the servo status data from DarwinSensors. It takes the servo commands and translates them into a string to be advertised by the igus control publisher. Since the ServoTarget message is a vector of commands, the order of commands is not constant. Since the message being advertised is of string type, it is nescessary to arrange the order of servo commands with some constant order. The `ParseServos()` procedure performs this operation and the order used is the enumeration shown in Table A.2.

on`<Trigger<GazeboWorldCtrl> >()` triggers on a GazeboWorldCtrl message. This message is used to reset the Gazebo world and reset the time.

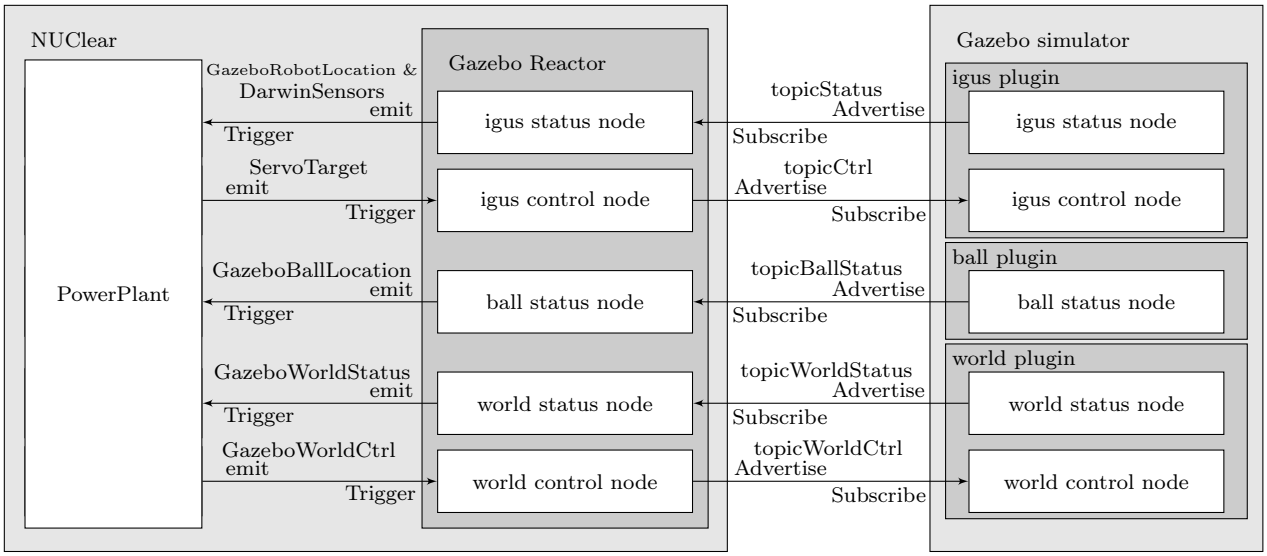


Figure 4.3: The IgnTrans nodes used in the Gazebo Reactor module act as the data translation point from Ignition Messages on the right to protobuf messages used for internal communications between modules within NUClear on the left. The arrow direction is indicating the flow of data

The content of the NUClear protobuf messages shown in Figure 4.3 is listed in Tables C.1. The source code can be found in the online repository on GitHub.

4.2 The Gazebo world and plugins

The `environment.world` file creates a new world environment for GRS to simulate. It instantiates the models listed and sets up the physics engine with specific parameters. Along with models, it also links models with plugin libraries. Here, the world, ball and the igus models have their own plugin. Each model uses a SDF file to describe the physics of the model in more detail and link any extra resources such as 3D meshes and textures. Figure 4.4 shows how the world file links all the components together.

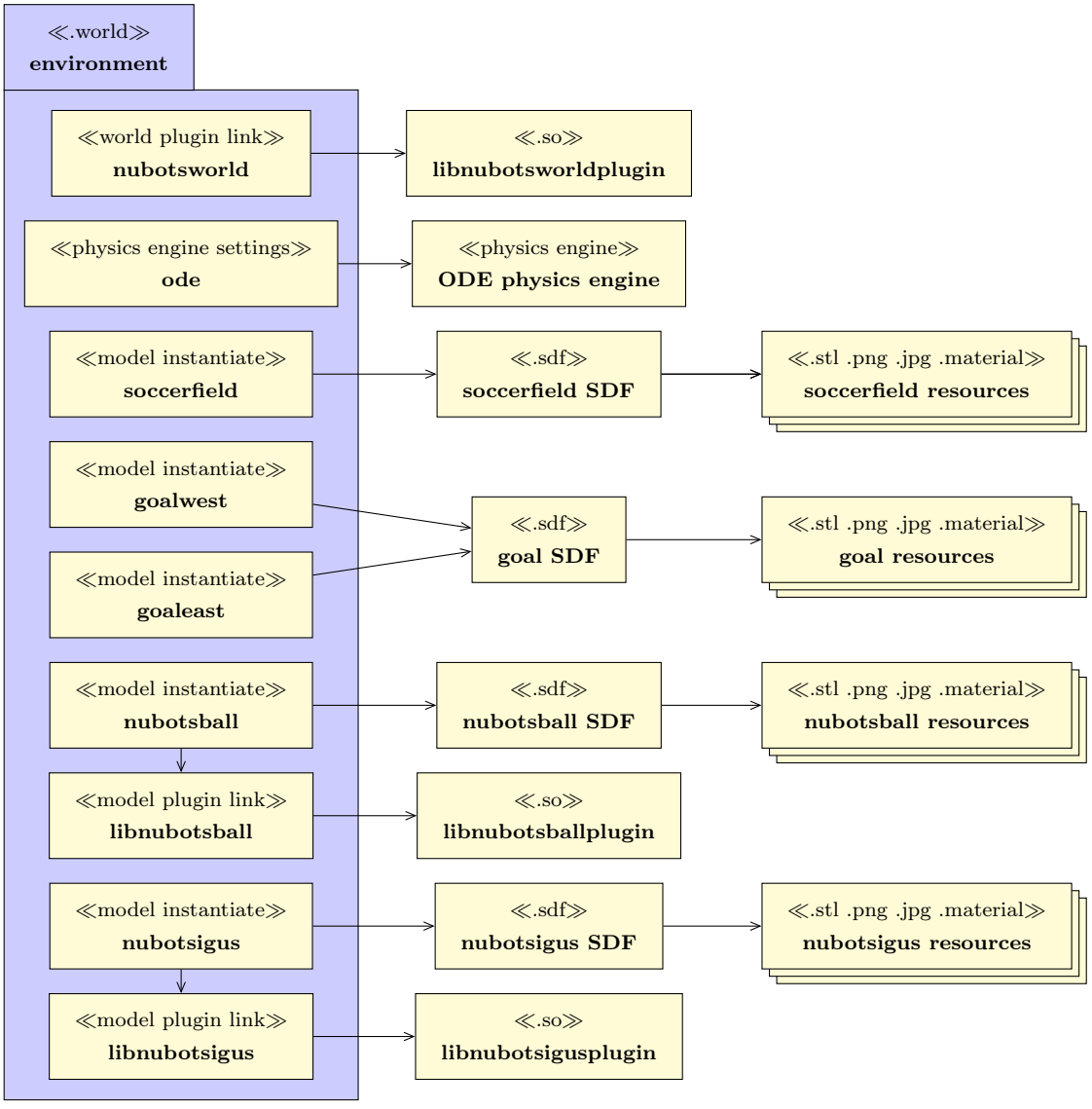


Figure 4.4: The structure of the `environment.world` file

4.2.1 The NUbots igus model plugin

The NUbots igus model plugin works with the 3D igus model during runtime in GRS. It controls the model's 20 joints and handles all communications with the IgnTrans nodes in the GRM in NUClear. The model is shown in Figure 4.5 with all the joint positions set to zero. The SDF file linked to the igus model describes the physical properties of the robot. These physical parameters have a big affect on how the robot reacts to its environment in simulation, and are listed in Table 4.2. Each link of the robot also has its own mass and moment of inertias defined here. These parameters were adjusted until the model behaved similarly to the real robot when running a getup script to make the robot stand up from lying down.

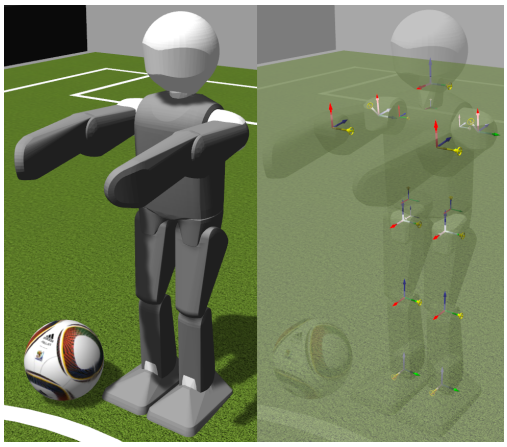


Figure 4.5: The igus robot model with a transparent view on the right showing the joints and their locations

Table 4.2: The physical properties of the igus model's joints

Field	Value	Description
max velocity	5.236	Joint velocity limit (rads per sec)
friction	0.5	Joint friction coefficient
damping	1	Joint damping
spring stiffness	3.25	Joint stiffness
spring stiffness damping	3.25	Equivalent to deadzone

All the joints are controlled by using a JointController. It is possible to control a joint directly in GRS by just setting the joint angle to whatever is required at a given time, but this is an unrealistic approach, since does not account for any physics phenomena such as momentum. The JointController works by setting either a target velocity or position for a joint rotation about some axis. The JointController requires a PID controller to be set, to define the manner in which the joint should approach its set target.

This is how the Dynamixel servos are controlled on the real robot, by setting a target position, velocity and gain to a PID controller. This data is all contained in the ServoTarget message, shown in Table 4.3. The JointController's PID controller has been tuned to model the Dynamixel MX-64 and MX-106 servo PID controller.

Table 4.3: The contents of the ServoTarget protobuf message

Field	Description
id	*uint32* The servo id
time	*Timestamp* A target time to get to the target position
position	*float* The target position in rads
gain	*float* The gain to apply to servo PID controller
torque	*float* The torque limit for the servo

Figure 4.6 shows two states of the right elbow pitch servo with the servo's current state on the left side and the state of the servo after issuing a Servo Target command on the right side. A Dynamixel servo requires a position to rotate to, referred to as θ_{tar}, a gain to apply to the PID controller and an angular velocity, v_{ang}, to get to the target position. v_{ang} is calculated as

$$v_{ang} = \frac{\theta_{tar} - \theta_{curr}}{time} rad/s \qquad (4.1)$$

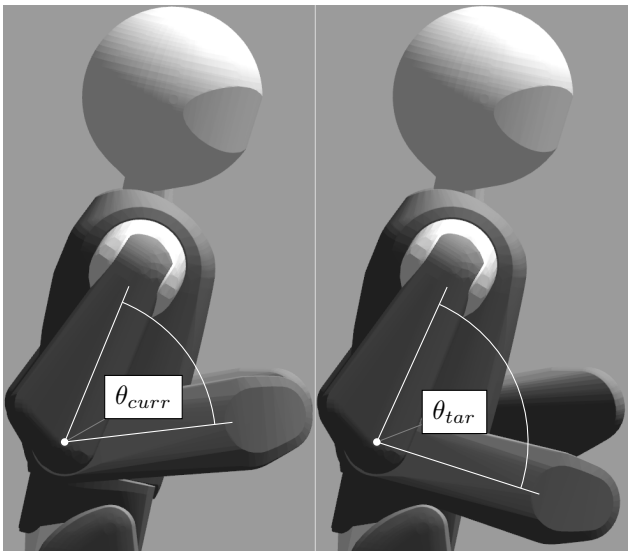


Figure 4.6: Controlling the right elbow pitch, with θ_{curr} as the current position and θ_{tar} as the target position

where the $time$ and θ_{tar} is taken from the time and position fields in the Servo Target message. θ_{curr} is retrieved from the latest DarwinSensors message NUClear has recieved. In this example, $\theta_{curr} = -2.443\ rad$ and $\theta_{tar} = -2.000\ rad$. If the command time value was set to be $time = 0.5s$, then $v_{ang} = 0.886\ rad/s$. The plugin implements this process by scaling the gain field in ServoTarget by four and setting it to the JointController's PID controller, setting a joint velocity limit to that of v_{ang} and setting the joint's target position to that of the position field in ServoTarget

message. The scaling of the gain field was nescessary since the joints were too weak in simulation to even keep the robot standing upright with the original gain values. Tuning these parameters requires a lot of time and could be tuned further to be more accurate.

Lastly, the plugin creates two IgnTrans nodes, one for the igus commands and another for status data, to communicate with NUClear. It links up a joint command callback function for ServoTarget messages and includes a function that is called on every simulation frame called `OnUpdate()`. When this function is run, the joint status and location of the robot is retrieved, concatenated into a string and advertised.

4.2.2 The NUbots ball and world plugins

The plugins are separated based on their purposes and models that they work with. Doing so helps with simplifying communications and synchronisation. Although the NUbots ball model plugin is a simple plugin that only retrieves the location in world space of the ball and advertises this data to its own topic, it is isolated from any other plugin or IgnTrans node. Since communications occur using callback functions and protobuf messages, this approach should be used even when implementing new plugins, such as adding some additional functionality for the goals in the field to register goal scoring for example. Such functionality should have its own plugin and IgnTrans nodes linked to the goal models.

Lastly, the world plugin retrieves the simulation time and real time elapsed during a simulation and advertises this to its own topic. It also subscribes to the world control topic to reset the world and time when the related command is recieved. Again, each plugin has its own `OnUpdate()` function, used to send periodic messages across the relative status topic. Models can be updated at a different rate to the world, so isolating libraries this way allows for isolated slowdowns if networking issues occur.

4.3 Custom NUClear clock for synchronisation

When a world is being simulated in GRS, the simulation time can lag behind the real time. This is mostly caused by a physics engine that is too demanding on the CPU, causing a slower simulation. A Real-Time Factor (RTF) is used to define this slowdown and is calculated as

$$RTF = \frac{simulation\ time}{real\ time} \tag{4.2}$$

Since NUClear is a completely separate system, a time desynchronisation between GRS and NUClear would occur at the rate of the RTF. Synchronising the two systems was required, but NUClear has never previously had the capability of running at a customisable execution time, nor has there been a requirement to do so until this project.

Such a feature required implementing a custom clock function that could give a time duration as a factor of the RTF recieved from GRS using the world plugin. Figure 4.7, shows the clock function used to get time points by NUClear. By using a *lastUpdate* time point, the following time point of the system would be a factor of the RTF in addition to the last updated point.

```
1 namespace NUClear {
2 clock::time_point clock::now() {
3   auto now = std::chrono::steady_clock::now();
4   lastUpdate = clock::time_point(lastUpdate + std::chrono::steady_clock
       ::duration((now - lastUpdate) * custom_rtf));
5   return lastUpdate;
6 }
```

Figure 4.7: Custom clock used for dynamic time points in NUClear

Now, anytime within NUClear a clock time point is required, this function will return a time point such that it would synchronise with the simulation time in GRS. With the v_{ang} for servo velocities, for example, this custom clock function would return a time value slowed down by the RTF, making simulation more realistic.

4.4 VirtualBox and packet headers networking issues

The NAT network device in the VM has to be disabled since VirtualBox alternates headers of outgoing packets of data causing them to be rerouted to another address. Once this happens, the IgnTrans nodes in an external domain will not receive any messages. Usually, this device name is *enp0s3*. To fix this issue, the following process needs to be followed:

1. Run the command `ifconfig` in the vm to check the configurations of all network devices.
2. Copy the IP address of the device below the first listed device. The first listed device is the NAT device so note the name because it will be disabled in the next step, so an alternative access point is required.
3. Run the command `sudo ifdown [NAT adapter]`, replacing the contents of the brackets with the name of the NAT adapter noted in the step before.
4. Once the NAT adapter is disabled, the vm will become unresponsive. To get out of the vm, press the keys `Enter`, `~` then `.` and the terminal should return to the host machine.
5. Then to return to the vm through the IP address copied in the first step, run the command `ssh vagrant@[public address of the bridged adapter]`, replacing the contents of the brackets with the IP address copied in the first step.

Once this process is followed, VirtualBox will not interact with any packets, allowing for IgnTrans nodes to communicate inter-domain as required.

5 The MOEA Framework

The design of the MOEA framework created for this project is detailed in this Section. Figure 5.1 shows the structure of the framework along with external dependencies. The EA core was selected to be Non-dominated Sorting Genetic Algorithm II (NSGA-II).[6] It is a well-known and tested algorithm being applied in a wide range of fields, including ER.[11] [3] There are several strong points that NSGA-II has over some other well-known MOEAs, such as Strength Pareto Evolutionary Algorithm (SPEA).[20] These strengths include a minimal distance of the final set of solutions to the real optimal solutions when converging, requiring a small number of fitness evaluations to converge and most notably, maintaining a strong diversity of solutions in the objective search space. Another advantage of NSGA-II is its ability to reduce clustering of solutions about local minima by using a non-dominated sort. Following, NSGA-II is described and then the integration of the framework into NUClear is detailed.

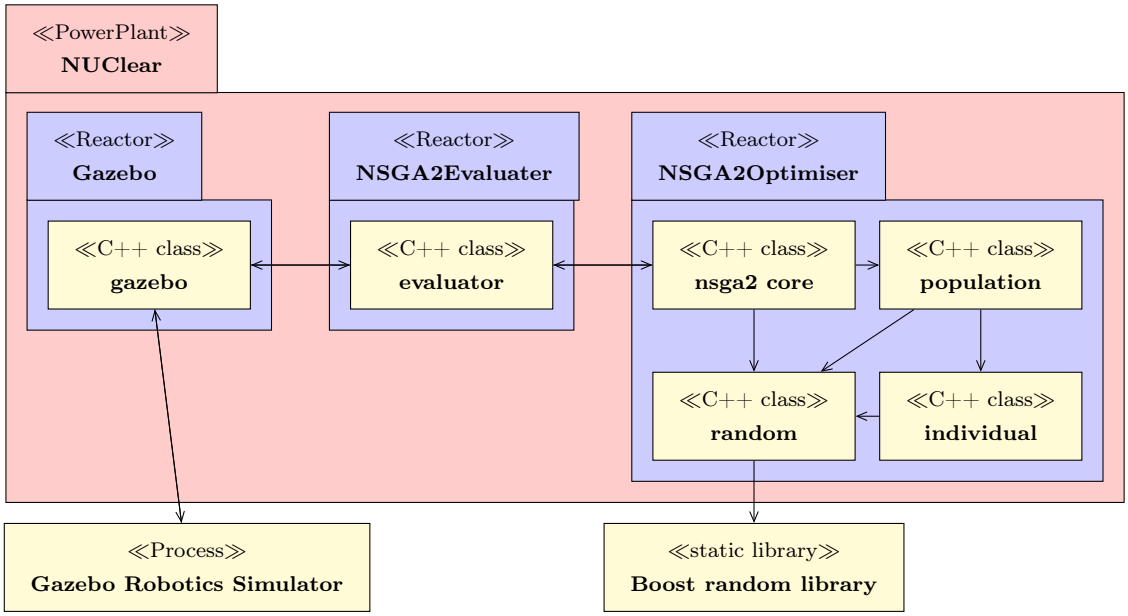


Figure 5.1: The structure of classes and dependencies of the MOEA framework

5.1 Elitist Non-dominated Sorting Genetic Algorithm

MOEAs classically have the structure of a single-objective EA at its core, but with some additional operators, such as the assigning of fitness scores to population individuals based on some non-dominated sorting and preserving diversity among solutions of the same non-dominated front. Such is the case with NSGA-II. It also implements the elitist mechanic, used to enhance the speed of convergence. Elitism involves copying individuals with good fitness scores from a parent population and directly advancing them to the parent population of the following generation, unmutated. Doing so ensures that good solutions are not lost when mutations occur. In order to explain in full detail the inner workings of NSGA-II, its two auxillary operators, a fast non-domination sorting method and a crowding distance method, will be introduced first. Then, a description of the main procedure follows.

5.1.1 A fast non-dominated sorting method

One of NSGA-II improvements on the original NSGA is the implementation of a fast non-dominated sort. It involves assigning a rank to each individual (Ind) in a population (Pop), relative to all other Inds in the Pop. NSGA-II advances generations by only selecting Ind from a Pop that score the lowest rank, thus stipulating minimisation of fitness scores.

For an Ind p and another Ind q, domination is determined by their objective scores. For the case of two objective functions, p dominates q if p has a lower score for both objectives than q. This domination check is done for each Ind against every other Ind in the Pop. Once all Ind have been checked, they are arranged into a series of fronts. The front that contains all Ind with a rank of one is referred to as the Pareto Front, or F_1. Following fronts, F_2, F_3 and so on hold Ind with the respective rank value. Algorithm 1 details the `FastNDS()` procedure. Each Ind holds a list of other Ind that it dominates, referred to as *dominations*, and another list which references all the other Ind that it is dominated by, referred to as *dominatedBy*.

5.1.2 Crowding distance

Crowding distance is defined as the average distance of surrounding Ind along each objective about a particular Ind in the search space. Crowding distance can be thought of as the density of solutions for a given point in the search space. After all Ind in a Pop are assigned a crowding distance, a comparison can be made between two Ind to determine their diversity. An Ind that has a smaller value of crowding distance is more crowded and thus less diverse than an Ind that has a higher crowding distance. Using this diversity metric along with the Ind rank, a crowded comparison operator is used to guide the selection process for a uniformly spread-out Pareto-optimal front.

Algorithm 2 shows the crowding distance comparator. It is used to compare all Ind in the same front against each other. Given an Ind, i being compared to another Ind j, if i has a lower non-domination rank than j, the comparator returns true. Other-wise, if both Ind belong to the same front, then the solution that is located in a lesser crowded region is selected.

Algorithm 1: Fast Non-dominated Sort

```
Input: Evaluated population, Pop
Output: Fronts F where F_i represents the i^th
        front
foreach Ind p in Pop do
    foreach Ind q in Pop do
        if p > q then
            p.dominations.add(q)
        else
            Increment p.dominatedBy
    if p.dominatedBy = 0 then
        p.rank = 1
        F_1.add(p)
    i = 1
    while F_i is not empty do
        foreach Ind p in F_i do
            foreach Ind q in p.dominations
             do
                Decrement q.dominatedBy
                if q.dominatedBy = 0 then
                    q.rank = i + 1
                    F_{i+1}.add(q)
        Increment i
output F
```

Algorithm 2: Crowding distance operator

```
Input: Ind i, Ind j
if i.rank < j.rank then
    return true
else
    if i.rank = j.rank & i.crowdDist >
     j.crowdDist then
        return true
    else
        return false
```

5.1.3 The main procedure

The main loop of NSGA-II is shown in Algorithm 3. It includes selection, mutation and evaluation functions that will be described later. The selection contains some crossover operation.

NSGA-II starts by creating a parent, child and mixed Pop. The parent Pop is initialised with randomly generated Ind. The variables that each Ind hold that are used when evaluating fitness scores is called its genome. These randomly generated genomes are then evaluated to get some fitness scores and then the first non-domination sort is applied and crowding distance completing the initial generation. The following generations follow a similar process. The child Pop is created by a selection and applying a crossover operator on some Ind from the parent. The child is then mutated and evaluated to attain some fitness scores. Following, the parent and child Pop are merged together to form a mixed Pop, that is then sorted into fronts and checked for crowding distance. From the mixed Pop, the Pareto-optimal front, Ind with rank 1, make up the new parent for the next generation, if there is enough space. If there still remains some vacancy in the new parent, the next non-dominated front is added.

Algorithm 3: NSGA-II structure

```
Input: N_gens, N_popsize
Create Pop Parent(N_popsize)
Create Pop Child(N_popsize)
Create Pop Mixed(2N_popsize)
Initialise(Parent)
Evaluate(Parent)
FastNDS(Parent)
CrowdingDistance(Parent)
while t < N_gens do
    Child = Selection(Parent)
    Mutate(Child)
    Evaluate(Child)
    Mixed = Merge(Child, Parent)
    Fronts = FastNDS(Mixed)
    CrowdingDistance(Fronts)
    Clear(Parent)
    foreach front f in Fronts do
        if Parent.size + f.size < N_max
         then
            Parent.add(f)
        else
            Remember f_next
            break
    while Parent.size < N_max do
        Parent.add(f_next.next)
    Increment t
```

Figure 5.2 shows the procedure visually. The parent and child Pop, shown as boxes, contain evaluated Ind, shown as coloured circles, where the colour indicates its fitness, with darker meaning the most minimised fitness score. The Ind that are beneath the dotted horizontal line and placed into fronts that are not added to the new parent are rejected and deleted from the algorithm. This procedure implements elitism to ensure no good solutions from the parent are lost when advancing to the next generation.

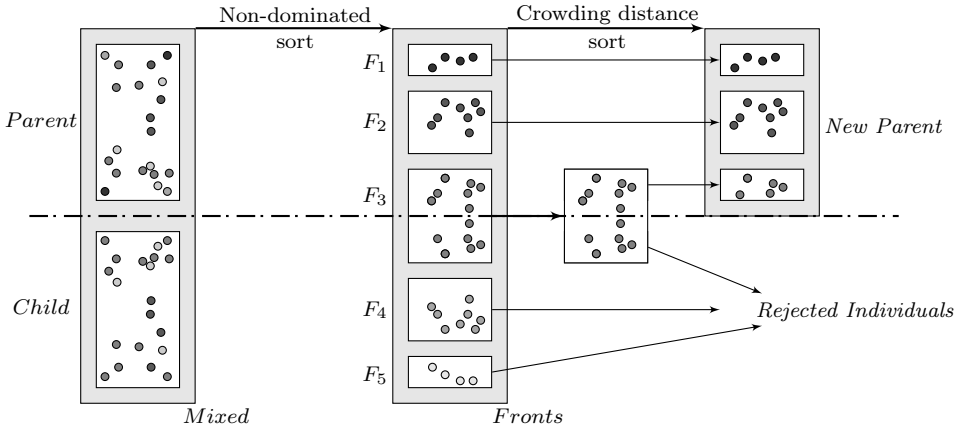


Figure 5.2: The NSGA-II procedure

5.1.4 Selection with simulated binary crossover

The selection process is used to create unique Ind for the child Pop. A tournament is held between four Ind at a time, where the dominant Ind are used in a Simulated binary crossover (SBX) to create a recombination offspring. SBX simulates the single-point crossover operator for binary strings.[7] A single-point crossover is shown in Figure 5.3 and involves selecting a point in the two parents' chromosones where all the data that is beyond that point is swapped between the parents. The resulting chromosones are the offspring. The decoded binary values are shown to the right of each chromosone.

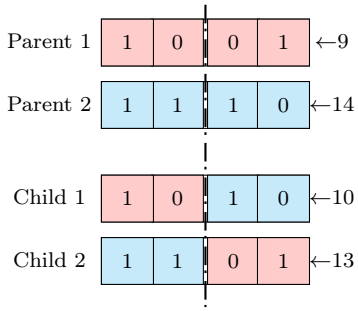


Figure 5.3: The single-point binary crossover operator

There are two important properties of binary single-point crossover that SBX keeps true. Firstly, the average value of the parents and children are the same, such as the example in the figure where

$$Parent_{ave} = \frac{9+14}{2} = child_{ave} = \frac{10+13}{2} = 11.5 \tag{5.1}$$

Secondly, a spread factor, β, is defined as the ratio of the spread of offspring points to that of the parent points along the gene value axis,

$$\beta = \left| \frac{c_1 - c_2}{p_1 - p_2} \right| \tag{5.2}$$

where p_1 and p_2 are the same gene from parent 1 and 2, and c_1 and c_2 are the same gene from child 1 and 2 respectively. If $\beta \leq 1$, a contracting crossover has occurred, with the offspring points being enclosed by the parent points. If $\beta > 1$, an expanding crossover has occurred with the offspring points enclosing the parent points. With real-coded chromosones, it is not possible to use binary-coded operations, so in order to simulate binary crossover, the two properties mentioned above are implemented to handle real-coded chromosones in SBX.

To reserve the first property of average value, the real-coded offspring values are calculated as

$$c_1 = \bar{x} - \frac{1}{2}\beta(p_2 - p_1) \tag{5.3}$$

$$c_2 = \bar{x} + \frac{1}{2}\beta(p_2 - p_1) \tag{5.4}$$

where $\bar{x} = \frac{1}{2}(p_1 + p_2)$ is the average of parent points and $p_2 > p_1$. Thus, $\bar{c} = \bar{x}$.

Following is the spread factor, β, with the probability distribution calculated as

$$c(\beta) = 0.5(n+1)\beta^n, \beta \leq 1 \tag{5.5}$$

for contracting crossovers and

$$c(\beta) = 0.5(n+1)\frac{1}{\beta^{n+2}}, \beta > 1 \tag{5.6}$$

for expanding crossovers. It can be seen that the power of distribution can be controlled with the n parameter, and is shown in Figure 5.4. A larger value of n gives a higher probability for creating solutions that are closer to the parent. SBX uses the spread factor β as a random number that follows the probability distribution shown in Figure 5.4.

A unified random number u is generated between 0 and 1, and along the β axis, a $\bar{\beta}$ value is retrieved that makes the region from 0 to $\bar{\beta}$ under the distribution curve equal u. This retrieved value is then used to calculate the offspring values as

$$c_1 = \bar{x} - \frac{1}{2}\bar{\beta}(p_2 - p_1) \tag{5.7}$$

$$c_2 = \bar{x} + \frac{1}{2}\bar{\beta}(p_2 - p_1) \tag{5.8}$$

This operator is used in the framework for every gene in the parents' genome until the childrens' genome is completed. Algorithm 5 shows the SBX procedure.

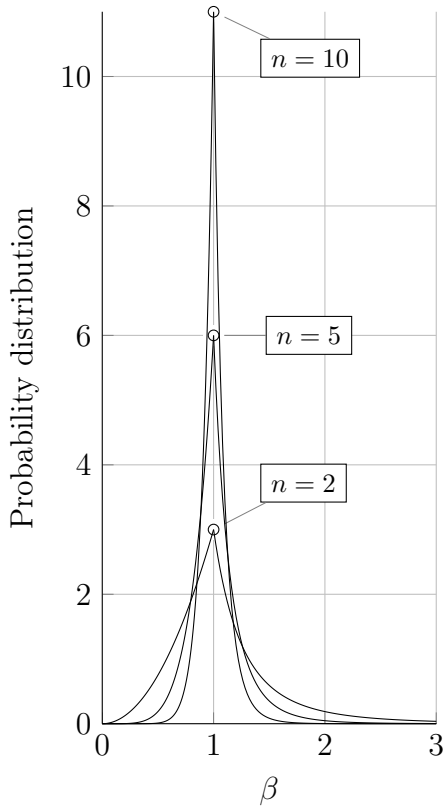


Figure 5.4: The probability distribution of β for offspring values relative to the original parent values

The tournament in the selection process is used to select the parents to participate in a SBX operation from the parent Pop. Two parents at random from the parent Pop are selected and are subjected to a tournament function. The winner of the tournament becomes the first parent and a second tournament is held to decide the second parent. There are four tournaments that select four parents for two crossover operations that produce four new Ind for the child Pop. The selection procedure is shown in Algorithm 4, that is run in groups of four for all Ind in a Pop.

Algorithm 4: Selection procedure

Input: 8 randomly selected Ind from parent Pop, Ind^{par}

Output: 4 Ind for child Pop, $Ind_1^{chi}, Ind_2^{chi}, Ind_3^{chi}, Ind_4^{chi}$

$Parent_1 = Tournament(Ind_1^{par}, Ind_2^{par})$

$Parent_2 = Tournament(Ind_3^{par}, Ind_4^{par})$

$Parent_3 = Tournament(Ind_5^{par}, Ind_6^{par})$

$Parent_4 = Tournament(Ind_7^{par}, Ind_8^{par})$

output $Ind_1^{chi}, Ind_2^{chi} = Crossover(Parent_1, Parent_2)$

output $Ind_3^{chi}, Ind_4^{chi} = Crossover(Parent_3, Parent_4)$

Due to this tournament, the population size must be a multiple of four. The tournament result is decided on the crowding distance operator shown in Algorithm 2. If they are equivalent, one is chosen at random. When a crossover is performed, there is a chance that the SBX procedure will not be executed and the parents will be copied to become the offspring, without any changes. This probability is referred to as $realCrossProb$ and is an input into the framework. Another important input to the framework is $etaC$, which is the distribution index used as n, which gives the user control on the power of the SBX operator spread. All random operations are handled by the random support class, that links with the Boost Random static library.

Algorithm 5: Simulated Binary Crossover

Input: Ind $Parent_1$ and Ind $Parent_2$, $realCrossProb$, $etaC$
Output: Ind $Child_1$ and Ind $Child_2$
if *random uniform number* $\leq realCrossProb$ **then**
  foreach p_1^i *in* $Parent_1$ *&* p_2^i *in* $Parent_2$ **do**
    gene p^i in parent genome
      Generate random uniform number u
      Calculate c_1^i & c_2^i using Eq. 5.7 and 5.8
      if *random uniform number* ≤ 0.5 **then**
        gene $Child_1^i = c_2^i$
        gene $Child_2^i = c_1^i$
      else
        gene $Child_1^i = c_1^i$
        gene $Child_2^i = c_2^i$
else
  foreach $Parent_1^i$ *&* $Parent_2^i$ **do** gene i in parent genome
    gene $Child_1^i = Parent_1^i$
    gene $Child_2^i = Parent_2^i$
output $Child_1$
output $Child_2$

5.1.5 Polynomial mutation

The mutation operator in the framework is based on a polynomial mutation for real-coded genomes.[5] Algorithm 6 shows the mutation procedure for a given Ind. For a given parent gene, p, the modified result p' is calculated by using a polynomial probability distribution to deviate the solution within the parent's local area. p also has a maximum and minimum limit, p^{upper} and p^{lower}. So the probability distribution left and right of p is adjusted so that no value outside p^{upper} and p^{lower} is created by the mutation operator. The modified gene p' is calculated with a random uniform number between 0 and 1 u as

$$p' = \begin{cases} p + \bar{\delta_L}(p - p^{lower}), & \text{for } u \leq 0.5 \\ p + \bar{\delta_R}(p^{upper} - p), & \text{for } u > 0.5 \end{cases} \tag{5.9}$$

where the two parameters $\bar{\delta_L}$ and $\bar{\delta_R}$ are calculated with a distribution index parameter η_m that is set by the user.

Similarly to SBX's n distribution index parameter, η_m alters the probability distribution of p'. A higher η_m value would mean a modified value closer to its original parent value. These two parameters are calculated as

$$\bar{\delta_L} = 2u^{\frac{1}{1+\eta_m}} - 1, \text{ for } u \leq 0.5 \qquad (5.10)$$

and

$$\bar{\delta_R} = 1-(2(1-u))^{\frac{1}{1+\eta_m}}, \text{ for } u > 0.5 \quad (5.11)$$

Algorithm 6: Polynomial mutation

Input: Ind real-coded genome, g, where g_i is the i^{th} gene, $realMutProb$, $etaM$

foreach g_i *in genome* g **do**
  if *random uniform number* $\leq realMutProb$ **then**
    if *random uniform number* ≤ 0.5 **then**
      Calculate g'_i using Eq. 5.9 & 5.10
    else
      Calculate g'_i using Eq. 5.9 & 5.11
    if $g'_i < g_i^{lower}$ **then**
      $g'_i = g_i^{lower}$
    if $g'_i > g_i^{upper}$ **then**
      $g'_i = g_i^{upper}$
    $g_i = g'_i$

5.2 Integration into NUClear

The NSGA-II core of the framework is dependant only on Boost random library for all random number generators and is already installed in the NUbots toolchain. There are several support classes for the core class, individual, population and random, with the source code avaliable for viewing on the Github repository.

The Framework requires input parameters to be defined in the NSGA2Optimiser Reactor module's configuration `.yaml` file. These parameters are listed in Appendix B along with recommended values.

5.3 Evaluating individuals

The evaluation procedure of a Pop is done serially, with one Ind being evaluated in GRS at a time. In order for an Ind to be evaluated, some fitness functions need to be defined in the NSGA2Evaluator Reactor module. All the data from the simulation environment and robot

is brought together in one Reactor module for the users convenience when defining fitness functions and evaluating Ind, separate from the core of the framework.

Figure 5.5 shows the intra-modular communication protocol within NUClear with the three Reactors involved described below.

NSGA2Optimiser Reactor contains the core of the algorithm. Its main task is to set up the framework with the input parameters, initiate the roundabout calls for Ind evaluations and process the fitness scores. Finally, when the final generation has been evaluated, it sends a termination message to stop the roundabout calls. It sends an evaluation request message for each Ind and then when the evaluation has completed, it recieves a fitness scores message containing the evaluated Ind fitness scores, generation number and Ind ID. During the evaluation process, the NSGA-II core will not advance. When a fitness score message is recieved, a Trigger is used to resume the core process.

NSGA2Evaluator Reactor brings all the necessary data together to one convenient place for setting up evaluation parameters, constraints, fitness functions and calculating fitness scores. It recieves an evaluation request, which contains the Ind genome, generation number and Ind ID. When it recieves this message, it resets the simulation environment and begins the evaluation process by emitting a robot command, amended using the Ind genome, for the robot to do something, such as start walking or kick. Once some termination criteria is met, such as the robot falling over for example, the evaluation is terminated, fitness scores calculated and sent to the NSGA2Optimiser Reactor module.

Gazebo Reactor is the same Reactor described in Section 4.1. It simply acts as the communication bridge between the NSGA2 Reactor modules and GRS.

In order to use the framework, this process must be followed:

1. Firstly, ensure that the plugin libraries for GRS are built and all IgnTrans nodes are properly defined, with matching namespace, partition and topic values with the VM. There is a user guide on the installation process for the plugins.[1]

[1] github.com/NUbots/Gazebo

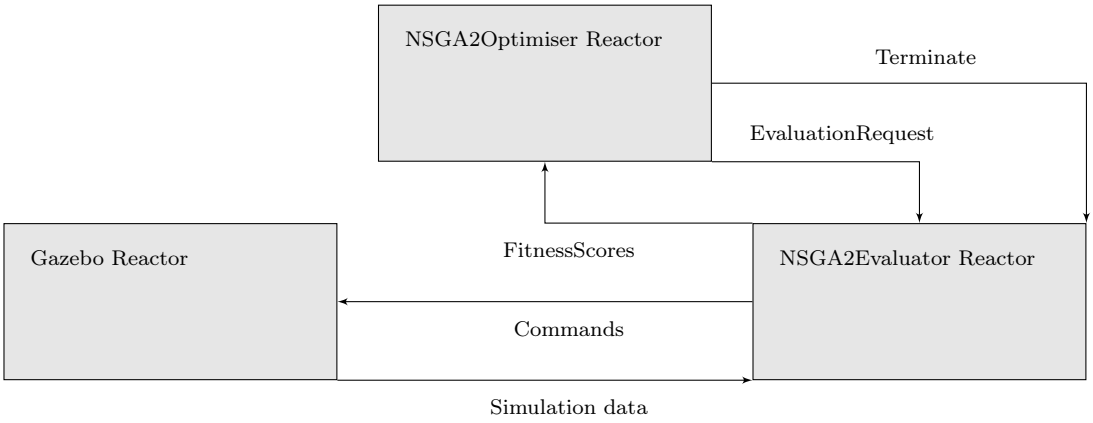


Figure 5.5: Inter-modular communications in the MOEA framework

2. Start GRS by running the `run_plugin.sh` script file found in the user guide. The soccer field and goals should appear with the NUbots igus robot in the centre.

3. Within the VM, follow the process in Section 4.4 to stop VirtualBox from interfering with outbound data packets.

4. Ensure that all objective functions are correctly defined within the NSGA2Evaluator Rector module and that the input parameters for the framework are correct.

5. Build and run the nsga2optimisation role within CMake.

6. The VM should now be communicating with GRS and the evolutionary optimisation process should now begin!

If any unexpected issues occur, take note of any error messages or warnings coming from the terminal window used to run the `run_plugin.sh` script file, such as incorrect world files and connection errors.

6 Optimising a Kick Script

Before moving on to optimising the walk engine, a simpler problem was used to test the framework's effectiveness. As an initial test, a kick script was optimised using the framework, with some interesting results. The NUbots Script engine is described next, then a generic script optimisation method is presented with an analysis of the results.

6.1 The NUbots Script engine

The NUbots script engine is used to move the robot in a scripted way, such as kicking, or getting up from lying on the ground. It is an open-loop control system that does not account for the robot's state when a script is executed. It does so by creating waypoints for the servos in the robot from pre-written script files. Each script consists of several waypoints, called frames, that list a ServoTarget command message for all 20 servos. Each frame has a time duration for the servos to reach their target positions. Figure 6.2 shows the structure of the kick script. It contains six frames, with each frame containing its own list of ServoTarget commands, θ^i_{RSP} and τ^i_{RSP}, where i is the frame number, θ is the target position and τ is the gain. The servo ID is an acronym of the IDs listed in Table A.2. Figure 6.1 shows how the original script controls the igus robot, frame by frame.[1]

Frame 1 *Frame 2* *Frame 3* *Frame 4* *Frame 5* *Frame 6*

Figure 6.1: The NUbots' igus robot kicking the ball with the original kick script

[1] NUbots RoboCup Qualification Video 2018

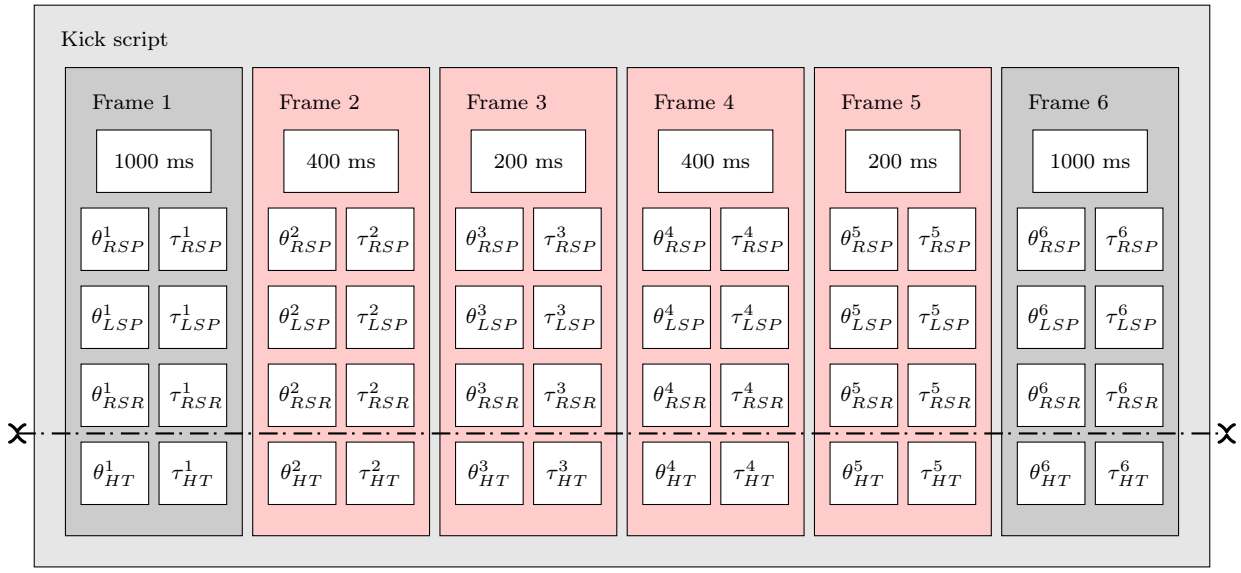


Figure 6.2: The structure of the kick script

6.2 Evaluation method for kicks

In order to evaluate a kick using the MOEA framework, firstly a genome for the Ind must be created. This is done by creating a gene for each target position in the script, with the exception of the first and last frames. In Figure 6.2, the frames shown in red contain the target positions that are used in the genome. The first and last frames of the script are omitted from the genome to keep the robot in the same state in which it started executing the script when it finishes a script. Figure 6.3 shows the genome for the kick script. The genome contains 80 real-coded genes, where each gene value is initialised with its original relative target position. Then, a unique search space for each gene is defined by setting upper and lower limits oriented about the original value with a δ value. For the case of gene 1, it is initialised to $\theta^2_{RSP} = -0.029\ rad$ and has an upper limit of $\theta^2_{RSP} + \delta = 0.471\ rad$ and a lower limit of $\theta^2_{RSP} - \delta = -0.529\ rad$, where $\delta = 0.5\ rad$. The original kick can now be mutated and evolved to optimise for some objective functions.

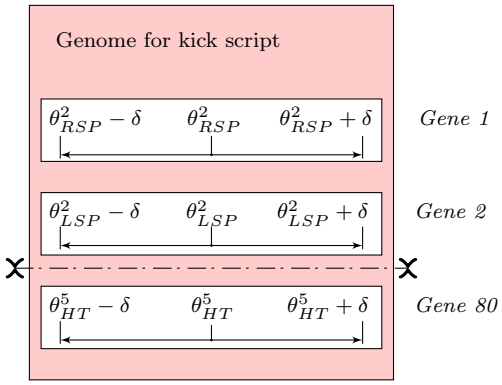


Figure 6.3: The kick script genome

Two objective functions, Obj_1 and Obj_2, were chosen to evaluate a kick's effectiveness. Obj_1 was set to assess the robot's torso stability during a kick. By using the IMU sensor in the torso, the accelerometer data was recorded during a kick and a maximum field plane sway, $||\vec{fps}_{max}||$, value was obtained. The field plane sway, $||\vec{fps}||$, is the 2D vector magnitude of the x and y components of the accelerometer sensor, calculated as

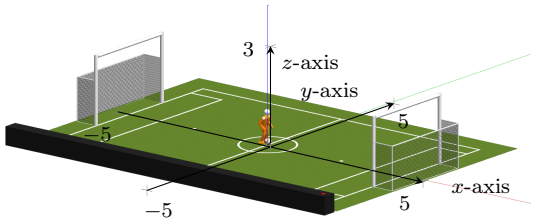


Figure 6.4: The 3D space orientation in NUClear and GRS, units in metres

$$||\vec{fps}|| = \sqrt{\vec{x_{accel}}^2 + \vec{y_{accel}}^2} \tag{6.1}$$

$||\vec{fps}||$ is calculated each time a new sensors message arrives, which is 90 times per second. If the newly calculated $||\vec{fps}||$ is greater than any previous values, $||\vec{fps}_{max}||$ is updated with this value. Now a stability metric is established, Obj_1 can be defined as

$$Obj_1 = ||\vec{fps}_{max}|| \ (ms^{-2}) \tag{6.2}$$

Since torso sway is to be minimised, lower scores for Obj_1 will advance. The world coordinate system used is shown in Figure 6.4 for reference.

Obj_2 is maximisation of the ball distance from the kick location. Since a minimisation function is required, Obj_2 is defined as

$$Obj_2 = \frac{1}{ball\ distance} \ (m^{-1}) \tag{6.3}$$

With Obj_1 and Obj_2 and using the kick script in Figure 6.1 as a starting point for the evolutionary process, a more stable, stronger kick will evolve. Some termination conditions must now be defined to know when an evaluation of an Ind should be stopped. Firstly, if the robot has fallen over, the evaluation should be terminated and objective scores set to a relatively high value. This is done by checking the z component of the accelerometer sensor for values where $z > -2\,ms^{-2}$. Also, if by a certain time the ball has not moved from its spawn location, the evaluation will terminate.

6.3 Kick script optimisation results analysis

The initial test ran the framework for 500 generations with a Pop size of 40. Additional parameters are listed in Table 6.1 with the results shown in Figure 6.5.

Table 6.1: Framework parameters used in the kick optimisation tests

Test	**Seed**	*popSize*	*gens*	*mutProb*	*crossProb*	*etaC*	*etaM*
Initial test	123	40	500	0.2	0.33	18	50
Soft ground test	650	40	300	0.025	0.8	15	40

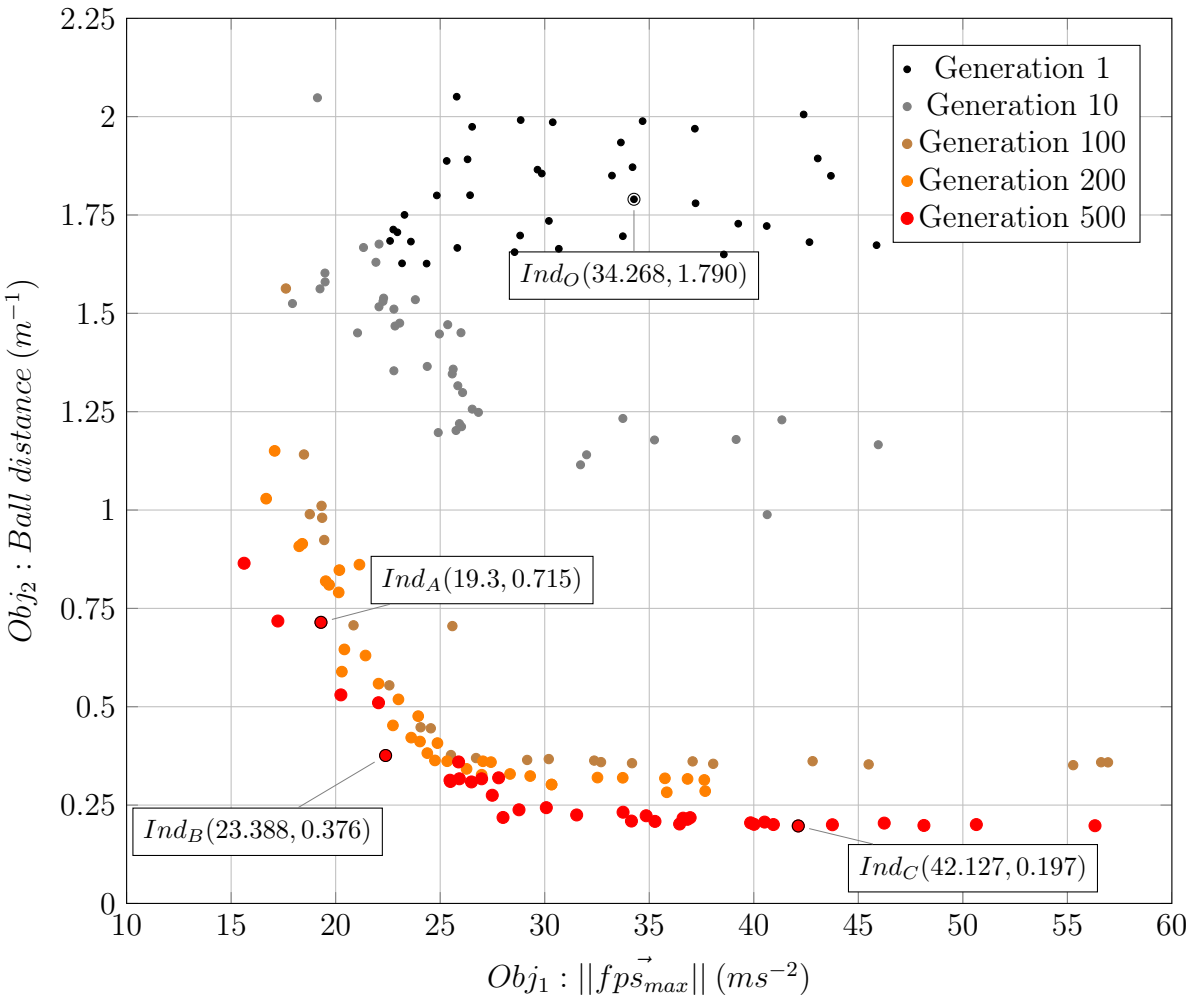


Figure 6.5: The results after 500 generations kicking the ball on a solid surface

In the initial test, a high mutation rate and relatively low crossover rate gave a large spread of solutions across the search space. The parent Pop of the initial generation consisted of all Ind initialised with the original kick, but with 20% of genes mutated. There is one Ind that has not been mutated, Ind_O, which is the unmodified original kick. It can be seen that generation 1 has a large distribution of solutions centred about Ind_O, indicating a strong mutation power. A low rate of recombination in the Selection process keeps this distribution high as the evolution advances.

Three Ind from the Pareto-optimal front, the front with rank of 1, have been selected for analysis from the final generation, Ind_A, Ind_B and Ind_C. Ind_A with $||\vec{fps}_{max}|| = 19.3\ ms^{-2}$ and ball distance of 1.4 m, is the most stable kick with the lowest kick distance of the selection. ***Ind_A has optimised for close to half of Ind_O torso sway, but with twice the kick distance.*** Ind_B is a more aggressive kick, with $||\vec{fps}_{max}|| = 23.39\ ms^{-2}$ and ball distance of 2.65 m. Ind_C is a super effective kick but at the cost of low torso stability, with $||\vec{fps}_{max}|| = 42.13\ ms^{-2}$ and ball distance of 5.08 m.

Figure D.3 shows a frame by frame analysis of Ind_A, Ind_B and Ind_C, relative to Ind_O. The robot has learnt to scoop the ball up and, similar to a throwing action, lob it off the ground to increase kicking distance. This is more evident in Ind_C, with a massive distance of 5 m. Ind_C seems rather implausible as a real solution, indicating that the reality gap is possibly rather large. Ind_A, however, is definitely a plausible solution for the real robot, as shown in Figure 6.6, an indication that the reality gap is not unrealistically vast.

Figure 6.6: Frame analysis of Ind_A and Ind_B being executed on the real robot

Ind_B did not run successfully, with the robot falling over, most likely due to too much momentum in the kicking foot and losing balance. Ind_C was not even attempted on the real robot for this reason, since the risk of damage would be high. It was clear that the reality gap needed to be reduced and the simulation required some improvements, one being improved modelling of ground contact. Another important point is the RTF had a significant impact on the evaluation of an individual and was unstable at the start of every evaluation. One reason for this instability is unnecessarily highly detailed collision meshes being used, with some parts of the robot's mesh having over 1.5 million faces. Since the ODE physics engine has to do collision calculations for each face in a mesh, it can slow down simulation time tremendously if there are a lot of faces. So before any more tests were done, a simpler igus model was created from removing 95% of the faces from the original model, shown in Figure 6.7. The new model has also had a visual change to the colour orange.

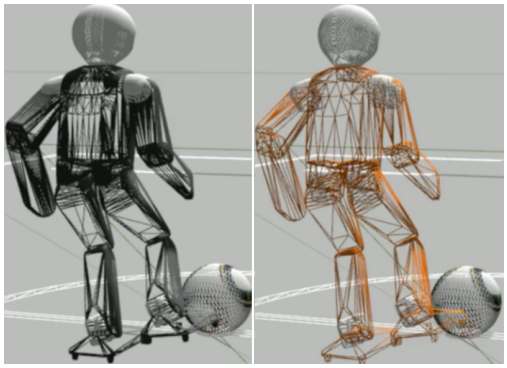


Figure 6.7: Wireframe view of the updated orange model

After some simple tests with the new model, a much more stable RTF improved simulation quality dramatically. Another major update was to the ground contact simulation. The real robot has 2cm rubber studs under its feet to give it grip in a soft field with artificial grass. In RoboCup, the grass length can be up to 3cm, so this soft surface definitely needs to be accounted for in simulation. Figure 6.8 shows the solution. Studs were extruded from underneath the feet meshes of the new model and rather than using a solid surface for the soccer field, shown by the black line, a soft collision surface was added 2cm above the solid surface to simulate a soft grassy plane. Notice how the ball is resting above the solid surface, on top of the grassy collision plane, indicated by the green line.

A second test was run using these new updates and different framework parameters, listed in Table 6.1 under the soft ground test entry. It was evident from the initial test that with using a Pop size of 40, the framework started converging on an optimal-solution set after generation 200. So in the soft ground test, the generation count was set to 300. Also, to reduce the spread of solutions across the search space, the crossover rate was increased

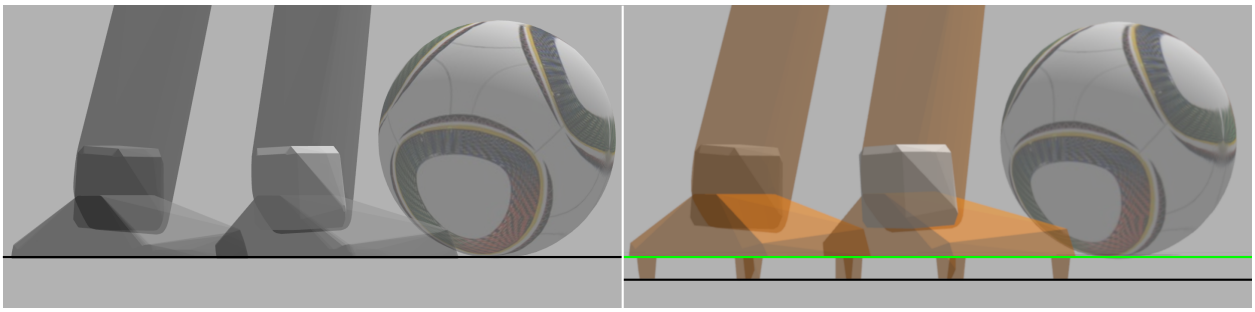


Figure 6.8: The changes made to the field to simulate grass seen in RoboCup

to 80%, with a much lower mutation rate of 2.5%. The results are shown in Figure D.1. From looking at the low spread of solutions of the initial generation, it can be seen how the lower mutation rate and higher crossover rate has decreased the entropy of solutions. As the generations advance, this is still evident, right until the final generation. Surprisingly, all the mutated Ind of the initial generation have a gradually more stable kick than Ind_O, possibly due to the newly added soft grass collision surface increasing stability.

As with the initial test, three Ind from the Pareto-optimal front have been selected for analysis from the final generation, Ind_A, Ind_B and Ind_C. Ind_A with $||\vec{fps}_{max}|| = 13.923\ ms^{-2}$ and ball distance of 2.23 m, is an improvement in both torso stability and kick effectiveness than the initial test results. Ind_B and Ind_C also have improved stability scores. Kick distance has decreased from more than 5 m down to a more realistic 3.55 m. Figure D.4 shows the frame by frame analysis of the selected Ind set. The original kick script was modified to use the arms in manner which is more comparable to a real human and also to make it visually easier to see differences in kick styles. Ind_A, Ind_B and Ind_C all have somewhat more realistic kicks than the initial test results, indicating that the reality gap has narrowed, thanks to the improvements made.

Some videos are available that show the real tests for Ind_A[2] and Ind_B[3], as shown in Figure 6.6. There is also a side-by-side comparison of the simulated kicks in the first simulation test.[4]

[2] Kick Test with Optimised Kick Script A
[3] Kick Test with Optimised Kick Script B
[4] A Comparison of 4 Kicks

7 Optimising the NUbots Walk Engine

After optimising the kick script, it was clear that the framework was working. Optimising the walk engine involved tuning 46 parameters that describe how the walk engine should generate waypoints for the joints during a walk cycle and were used to create the genomes of Ind for the walk optimisations.

The walk engine uses these parameters to calculate a list of waypoints for the servos to create a gait.[18] Unlike the script engine, the walk engine creates a waypoint for every time frame while running. A complete list of the parameters used are listed in Table E.1 and E.2. Figure 7.1 shows some of the parameters that define the spatial properties of the resulting walk. They are very sensitive and can drastically alter the gait of the robot. Along with these spatial parameters, there are other important time constraining parameters, such as the step time and Zero Moment Point (ZMP) time.[17]

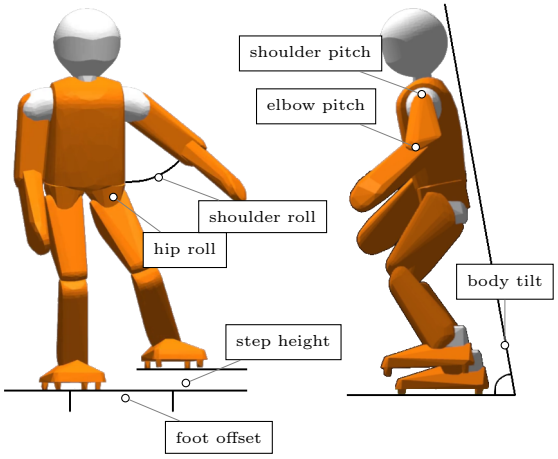


Figure 7.1: A visual representation of some important spatial walk parameters used in the walk engine

ZMP time is a stability criterion which defines how a walk should control its centre of mass above the support foot. The original walk had an inhibiting double-tap motion in the walk cycle, due to a de-sync between the step time and ZMP time.[1] Since the parameters are so sensitive, tuning the walk engine by hand is immensely time consuming with risk of damaging the robot if an infeasible parameter value is used, so by using the framework to optimise the parameters, optimal values for different walking conditions and strafes can be found in a matter of hours.

[1] Walking Test with Original Walk Parameters

7.1 The genome and walk fitness functions

The genome for the walking tests consisted of one gene for each of the 46 parameters in the walk engine. In a very similar way that the script engine was evaluated, each gene was given a minimum and maximum value, roughly 50% in each direction of the original value. For example, the ZMP time parameter was originally 0.4, so the minimum and maximum values were set to be 0.2 and 0.6 respectively.

The two fitness functions for walk evaluation were minimisation of the time required to walk from a standing pose a distance of 2.5 m in simulation, Obj_1, and the minimisation of the maximum magnitude of all three components of the accelerometer, Obj_2, where

$$Obj_2 = ||\vec{accel}_{max}|| \, (ms^{-2}) \tag{7.1}$$

and $||\vec{accel}||$ is calculated as

$$||\vec{accel}|| = \sqrt{\vec{x_{accel}}^2 + \vec{y_{accel}}^2 + \vec{z_{accel}}^2} \tag{7.2}$$

In the same way as the kick tests, $||\vec{accel}||$ is calculated each time a new sensors message arrives. If the newly calculated $||\vec{accel}||$ is greater than any previous values, $||\vec{accel}_{max}||$ is updated with this value.

7.2 Walk results analysis

The framework was set up with the input parameters listed in Table 7.1, it was run for 50 generations. The results are shown in Figure 7.2.

Table 7.1: Framework parameters used in the walk optimisation test

Seed	*popSize*	*gens*	*mutProb*	*crossProb*	*etaC*	*etaM*
300	40	50	0.025	0.8	13	20

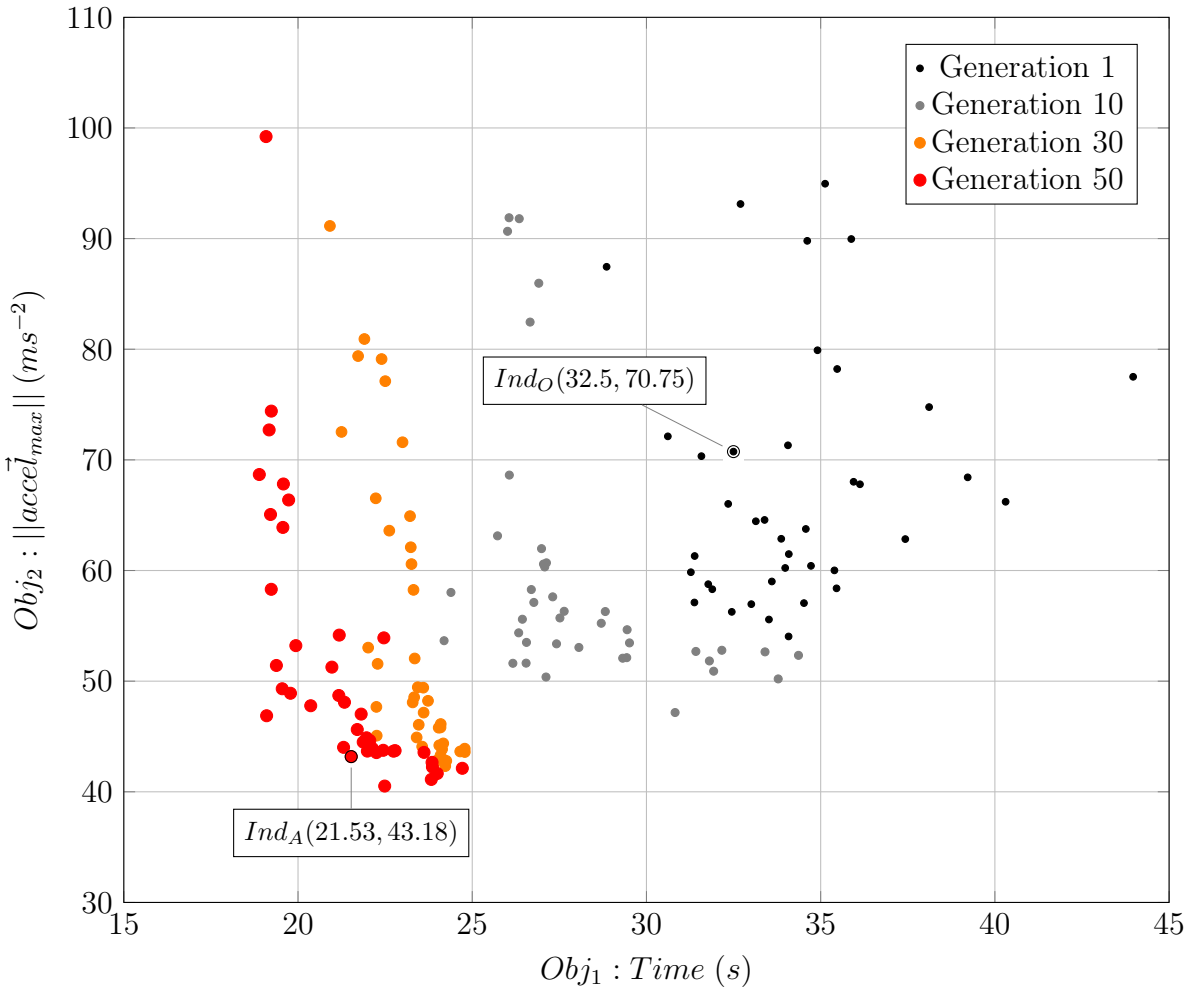


Figure 7.2: The results after 50 generations optimising the walk engine parameters

Although the generation count of 50 is comparatively lower to the 300 or so generations run for the kick tests, it can be seen that substantial improvements in the walk quality are being produced. Ind_O, the Ind of the walk with the original gene scored 32.5 s and a $||\vec{accel}_{max}||$ value of 70.75 ms^{-2}. It is also evident that the initial generation has a wide spread of solutions across the solution space. After generation 30, Obj_1 begins to converge on 20 s, due to the walk engine receiving a static velocity command which determines the maximum speed of the walk. By generation 50, a speed increase of 30% and stability increase of 40% is observed when comparing to an Ind in the Pareto-front, marked as Ind_A. The relative improvements in the gait cycle are immediately obvious in the following video, with a nice smooth transition between supporting and stepping feet.[2] Figure D.2 shows the accelerometer values during an unoptimised walk on the left, and optimised on the right.

[2] Walking with Optimised Walk Parameters Ind_A

Figure 7.3 shows the gait cycle of Ind_O on the top, with Ind_A on the bottom. Both walks start the gait cycle with their left foot leading and becoming the support foot with the right foot stepping forward, away from the camera. A original walk had a hindering double stepping motion, most evident in the frames highlighted in black. This is due to a desynchronisation between the ZMP time and step time, throwing the robot off balance when taking a step. It is clear that these two parameters have the most significant impact on the walks generated by the engine.

The values of ZMP time and step time for Ind_O were 0.85 and 0.4 respectively. The values found for the optimised walk Ind_A were 0.848 and 0.281 respectively. The other notable mutated gene values include shoulder roll, step height and hip roll compensation. Together with the new ZMP and step time parameters, they combine for a fast efficient gait.

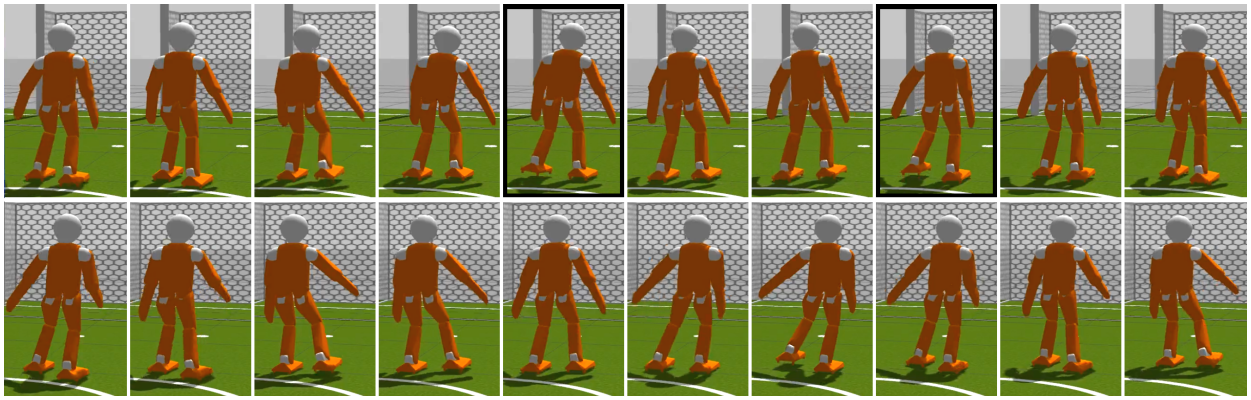


Figure 7.3: Discrete time analysis of the original walk cycle, Ind_O, on the top compared to an optimised walk cycle shown on the bottom, Ind_A

The parameters found during this test are optimised for the given forward walk command. There are of course different ways of walking and strafing, with each case having different optimal parameter values. From this point, more tests will be run with the framework to find the optimal parameters for different walks, such as walking backwards and then strafing left and right. The robot at the time of testing was unfortunately suffering from hardware issues, which halted any chance of testing the optimised parameters on it.

8 Conclusion

8.1 Project objectives summary

In summary, the three objectives set out at the start of the project have been completed successfully, supported by the promising results observed when testing with different parameterised engines in the NUbots software. Those three objectives were:

- To Set up a simulation environment and communications middleware with the NUbots' software and GRS,
- To Implement a generic MOEA framework to optimise the NUbots walk engine,
- To test the optimised parameters on the robot to validate the framework's credibility.

The communications middleware between the robot controlling software and the simulation environment was a crucial step in this project, with all following objectives solely relying on it. After researching possible solutions to the problem, an open source communications library, called Ignition Transport, was used to design and implement the middleware, allowing for easy communications between the two systems. The library also provides the possibility for distributed setups with multiple computers, which is being planned for a future expansion to the communications module.

Following, the MOEA framework has shown its usefulness as a generic tool that can be used in many different parameterised optimisation problems, such as optimising scripts and walks. The framework consists of two main parts, a EA core and the supporting infrastructure allowing it to be integrated into the NUbots software. Finally, the improvements observed in both the kick and walk indicate that the communications and the framework have worked together and validate the future use and expansion of these tools within the team.

8.2 Expansions

An expansion to the communications module is planned for a distributed simulation environment. Since the communications occur over the LAN, instances of GRS running on multiple machines on the same network can communicate easily with each other using the IgnTrans libraries. This expansion would greatly increase the speed of evaluations, by running multiple evaluations in parallel, rather than in series. Other expansions include integrating the visual system sub modules with the communications module to allow for the robot to be able to use its localisation and vision capabilities within GRS.

8.3 Future work

The immediate future work of this project is to find optimal parameters of the walk engine for various different types of walks and strafes. Other scripts that the team rely heavily on for controlling the robot, such as the get up script, are planned to be optimised.

As mentioned in the literature review, one of the biggest challenges of this project was dealing with the reality gap. It is clear from the results that the reality gap is present but not impossible to overcome. Since this project mainly focused on the software aspect of evolutionary computing, naturally a proceeding long term goal is to more accurately simulate how the robot behaves in simulation. There are tools available within GRS that can help with this, but it would take a lot of time testing and detailing the physical properties of the robot, then modelling them in the simulator.

Perhaps the most governing behaviour in the case of the igus robot is the joints. In order to accurately simulate any script running with the Script engine, the simulation of joint motion in between waypoint commands is vital to the credibility of the results. If the simulation of the joints becomes accurate, the reality gap will become even smaller, and will complement the robustness of the MOEA framework.

Bibliography

[1] Allgeuer, P., Farazi, H., Ficht, G., Schreiber, M., Behnke, S.: The igus humanoid open platform: A child-sized 3d printed open-source robot for research. German Journal on Artificial Intelligence 30 (July 2016)

[2] Antonova, R., Rai, A., Atkeson, C.G.: Sample efficient optimization for learning controllers for bipedal locomotion. In: 2016 IEEE-RAS 16th International Conference on Humanoid Robots (Humanoids). pp. 22–28 (November 2016)

[3] Conesa-Muñoz, J., Ribeiro, A., Andujar, D., Fernandez-Quintanilla, C., Dorado, J.: Multi-path planning based on a nsga-ii for a fleet of robots to work on agricultural tasks. In: 2012 IEEE Congress on Evolutionary Computation. pp. 1–8 (June 2012)

[4] Cully, A., Mouret, J.B.: Evolving a behavioral repertoire for a walking robot. Evolutionary Computation 24(1), 59–88 (2016)

[5] Deb, K., Deb, D.: Analysing mutation schemes for real-parameter genetic algorithms. International Journal of Artificial Intelligence and Soft Computing 4(1), 1–28 (2014)

[6] Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: Nsga-ii. IEEE transactions on evolutionary computation 6(2), 182–197 (2002)

[7] Deb, K., Sindhya, K., Okabe, T.: Self-adaptive simulated binary crossover for real-parameter optimization. In: Proceedings of the 9th annual conference on Genetic and evolutionary computation. pp. 1187–1194. ACM (2007)

[8] Doncieux, S., Mouret, J.B., Bredeche, N., Padois, V.: Evolutionary robotics: Exploring new horizons. In: Doncieux, S., Bredèche, N., Mouret, J.B. (eds.) New Horizons in Evolutionary Robotics. pp. 3–25. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)

[9] Ha, I., Tamura, Y., Asama, H., Han, J., Hong, D.: Development of open humanoid platform darwin-op. SICE Annual Conference pp. 2178 – 2181 (September 2011)

[10] Houliston, T., Fountain, J., Lin, Y., Mendes, A., Metcalfe, M., Walker, J., Chalup, S.: Nuclear: A loosely coupled software architecture for humanoid robot systems. Frontiers in Robotics and AI 3 (April 2016)

[11] Huang, Y., Fei, M.: Motion planning of robot manipulator based on improved nsga-ii. International Journal of Control, Automation and Systems 16(4), 1878–1886 (August 2018)

[12] Koos, S., Mouret, J.B., Doncieux, S.: Crossing the reality gap in evolutionary robotics by promoting transferable controllers. In: Proceedings of the 12th annual conference on Genetic and evolutionary computation. pp. 119–126. ACM (2010)

[13] Mouret, J.B., Doncieux, S.: SFERESv2: Evolvin' in the multi-core world. In: Proc. of Congress on Evolutionary Computation (CEC). pp. 4079–4086 (2010)

[14] Mouret, J.B., Chatzilygeroudis, K.: 20 years of reality gap: a few thoughts about simulators in evolutionary robotics. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion. pp. 1121–1124. ACM (2017)

[15] Nygaard, T.F., Torresen, J., Glette, K.: Multi-objective evolution of fast and stable gaits on a physical quadruped robotic platform. In: 2016 IEEE Symposium Series on Computational Intelligence (SSCI). pp. 1–8 (December 2016)

[16] Raj, M., Semwal, V.B., Nandi, G.C.: Multiobjective optimized bipedal locomotion. International Journal of Machine Learning and Cybernetics (March 2017)

[17] Song, S.: Development of an omni-directional gait generator and a stabilization feedback controller for humanoid robots. Ph.D. thesis, Virginia Tech (2010)

[18] Yi, S., Hong, D., Lee, D.D.: A hybrid walk controller for resource-constrained humanoid robots. In: 2013 13th IEEE-RAS International Conference on Humanoid Robots (Humanoids). pp. 88–93 (October 2013)

[19] Zahn, B., Ucherdzhiev, I., Szeles, J., Botzheim, J., Kubota, N.: Optimization of a psd controller using bm algorithms. International Conference on Intelligent Robotics and Applications pp. 362–372 (April 2016)

[20] Zitzler, E., Laumanns, M., Thiele, L.: Spea2: Improving the strength pareto evolutionary algorithm. Tech. rep. (2001)

A Robot Specifications

Table A.1: The igus Humanoid Open Platform specifications

Type	Specification	Value
General	Height & Weight	92 cm, 6.6 kg
	Battery	4-cell LiPo (14.8 V, 3.8 Ah)
	Battery Life	15–30 min
	Material	Polyamide 12 (PA12)
PC	Product	Gigabyte Brix GB-BXi7-5500
	CPU	Intel i7-5500U, 2.4–3.0 GHz
	Memory	4 GB RAM, 120 GB SSD
	Network	Ethernet, Wi-Fi, Bluetooth
	Other	4 USB 3.0, HDMI, MiniDP
CM730	Microcontroller	STM32F103RE (Cortex M3)
	Memory	512 KB Flash, 64 KB SRAM
	Other	3 Buttons, 7 LEDs
Actuators	Total	8 MX-64, 12 MX-106
	Head	6 MX-106
	Each Arm	2 MX-64
	Each Leg	3 MX-64
Sensors	Encoders	4096 ticks/rev
	Gyroscope	3-axis (L3G4200D chip)
	Accelerometer	3-axis (LIS331DLH chip)
	Magnetometer	3-axis (HMC5883L chip)
	Camera	Logitech C905 (720p)
	Camera Lens	Wide-angle lens, 150 FOV

Table A.2: The igus servo enumeration order and the model joints order

Darwin Sensors Name	**Order**	**Igus model joints**
Right Shoulder Pitch	1	Left Hip Yaw
Left Shoulder Pitch	2	Left Hip Roll
Right Shoulder Roll	3	Left Hip Pitch
Left Shoulder Roll	4	Left Knee
Right Elbow	5	Left Ankle Pitch
Left Elbow	6	Left Ankle Roll
Right Hip Yaw	7	Left Shoulder Pitch
Left Hip Yaw	8	Left Shoulder Roll
Right Hip Roll	9	Left Elbow
Left Hip Roll	10	Head Pan
Right Hip Pitch	11	Head Tilt
Left Hip Pitch	12	Right Hip Yaw
Right Knee	13	Right Hip Roll
Left Knee	14	Right Hip Pitch
Right Ankle Pitch	15	Right Knee
Left Ankle Pitch	16	Right Ankle Pitch
Right Ankle Roll	17	Right Ankle Roll
Left Ankle Roll	18	Right Shoulder Pitch
Head Pan	19	Right Shoulder Roll
Head Tilt	20	Right Elbow

B MOEA Framework Input Parameters

Table B.1: The input parameters for the MOEA framework, with recommended values

Parameter	**Value**	**Description**
Seed	Any int	A seed for the random number generator
popSize	$\%4$	Number of Ind in a Pop
generations	> 1	Number of generations to elapse before termination
objectives	> 1	Number of objectives
constraints	≥ 0	Number of constraints
realVars	≥ 0	Number of real-coded genes in the genome
realLimits	$realVars$	A list of each real gene's max and min values
realCrossProb	$50 - 90\%$	The probability of crossover operation for real-coded genes in the genome
realMutProb	$\frac{1}{popSize}\%$	The probability of mutation operation for a real gene in the genome
etaC	$2 - 20$	Defines the power of crossover operation
etaM	$5 - 50$	Defines the power of mutation operation
binVars	≥ 0	Number of binary-coded genes in the genome
binBits	≥ 0	Number of binary bits for each binary genes in the genome
binLimits	$binVars$	A list of each binary gene's max and min values
binCrossProb	$50 - 90\%$	The probability of crossover operation for a binary gene in the genome
binMutProb	$\frac{1}{popSize}\%$	The probability of mutation operation for a binary gene in the genome

C protobuf Messages

Table C.1: DarwinSensors protobuf message

Message	Component	Description
Accelerometer	x	*float* x component of the accelerometer
	y	*float* y component of the accelerometer
	z	*float* z component of the accelerometer
Gyroscope	x	*float* x component of the gyroscope
	y	*float* y component of the gyroscope
	z	*float* z component of the gyroscope
Servo	Error Flags	*uint32* Lists any error flags
	TorqueEnabled	*bool* if torque is enabled or not
	p Gain	*float* Proportional component of the servo
	i Gain	*float* Integral component of the servo
	d Gain	*float* Differential component of the servo
	Goal Position	*float* The target position
	Torque	*float* The current torque being exerted
	Present Position	*float* The current position of the servo
	Present Speed	*float* The current speed
	Load	*float* The current load
	Voltage	*float* The current voltage of the servo
	Temperature	*float* The current temperature of the servo
Servos	Servo[20]	*Servo* A list of 20 servo messages for each servo

D Additional Results Analysis

D.1 Soft grass kick test results

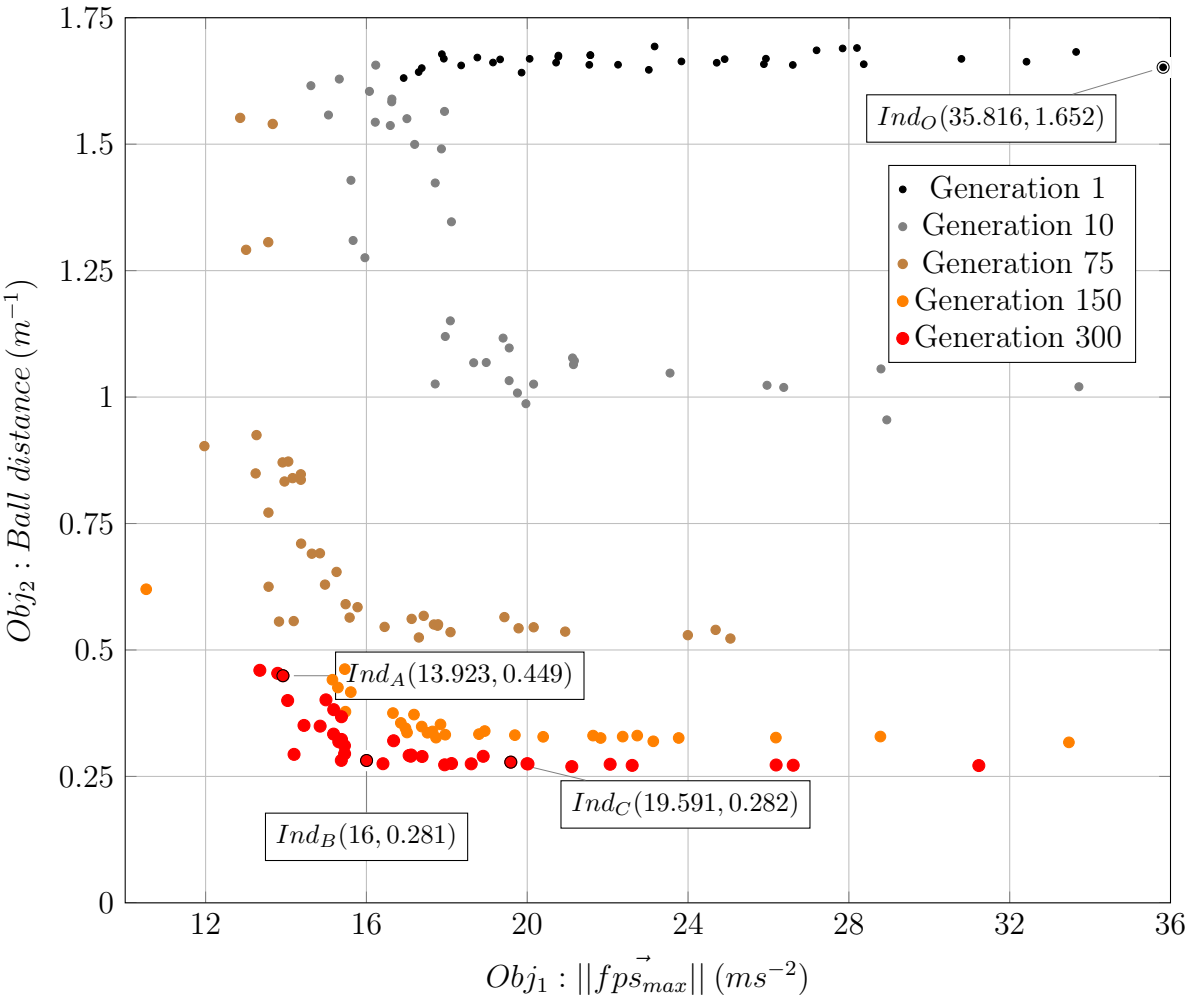
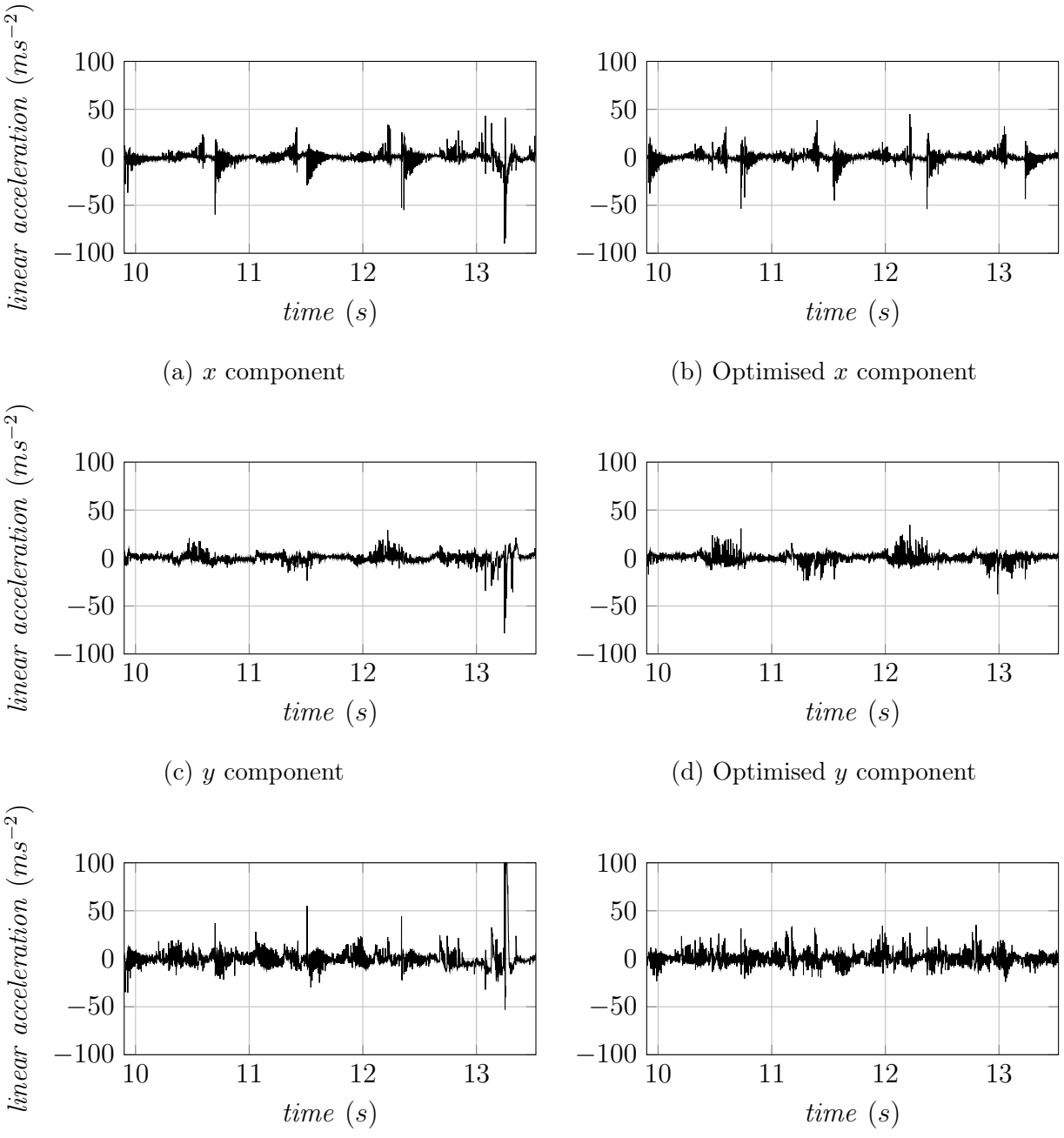


Figure D.1: The results after 300 generations kicking the ball on a grassy surface

D.2 Accelerometer data during a walk cycle

.



(a) x component

(b) Optimised x component

(c) y component

(d) Optimised y component

(e) z component

(f) Optimised z component

Figure D.2: The accelerometer data values during a single walk cycle

D.3 Individual Frame Analysis

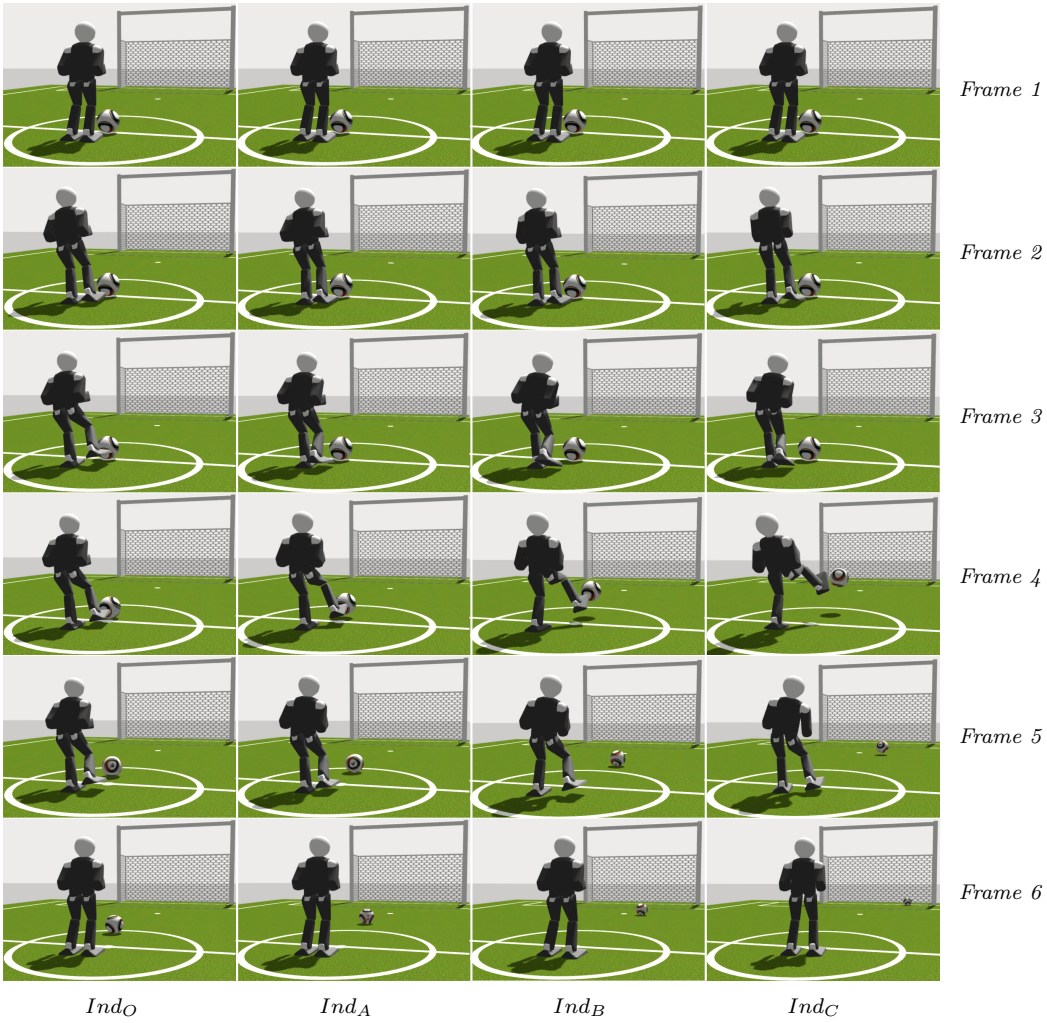


Figure D.3: Frame analysis of Ind_A, Ind_B and Ind_C compared to the original script, Ind_O, from initial kick test

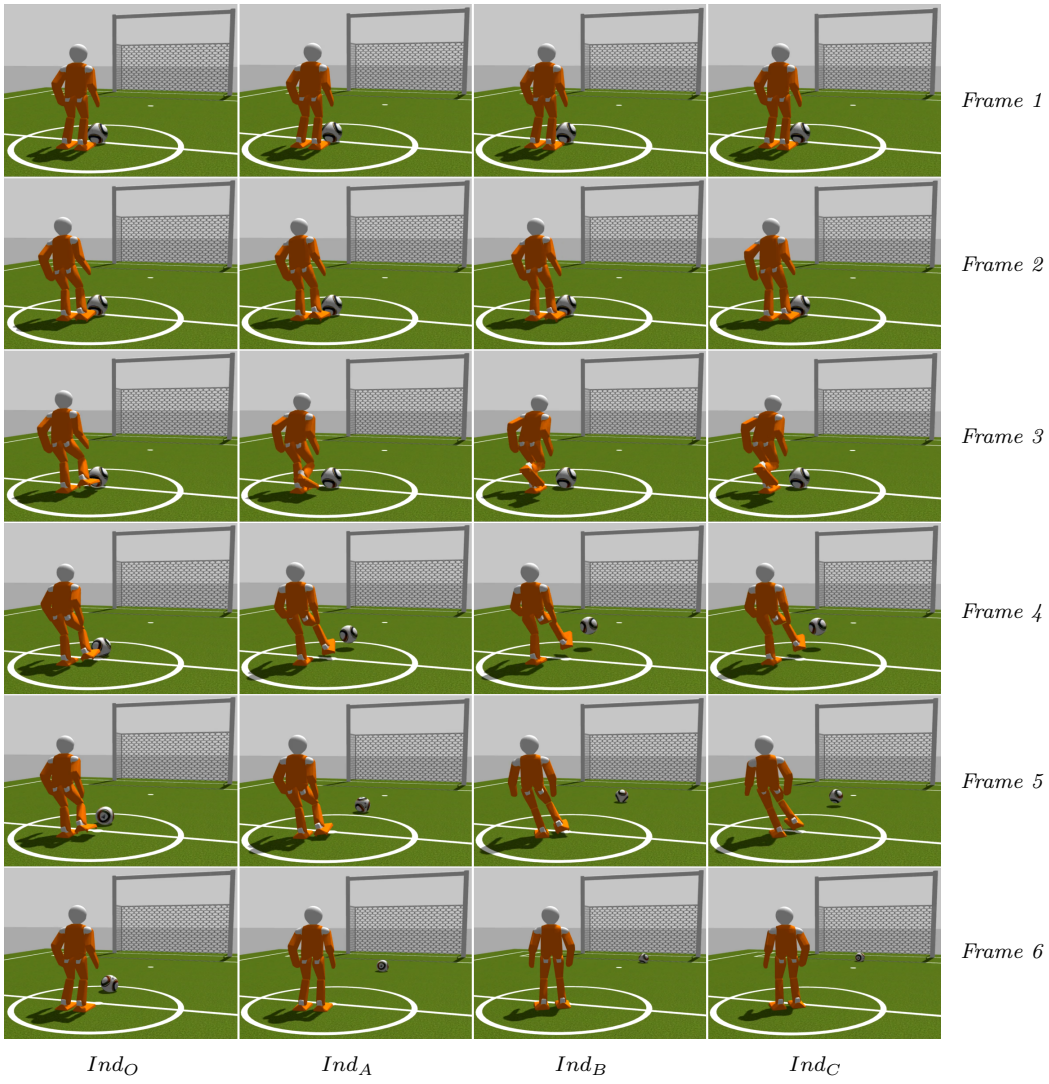


Figure D.4: Frame analysis of Ind_A, Ind_B and Ind_C compared to the original script, Ind_O, with soft grass kick test

E Walk Engine Parameters

Table E.1: Walk parameters for the walk engine shown for the original values, Ind_O and the optimised genome of Ind_A, parameters 1 - 20

Name	Ind_O	Ind_A
Body Tilt	0.174	0.174
Left Arm Start Pitch	1.570	1.570
Left Arm Start Roll	0.000	0.000
Left Arm Start Elbow	-0.523	-0.523
Left Arm End Pitch	1.570	1.571
Left Arm End Roll	0.000	0.000
Left Arm End Elbow	-0.523	-1.225
Right Arm Start Pitch	1.570	1.571
Right Arm Start Roll	0.000	0.000
Right Arm Start Elbow	-0.523	-1.221
Right Arm End Pitch	1.570	1.570
Right Arm End Roll	0.000	0.002
Right Arm End Elbow	-0.523	-0.523
Foot Offset 1	-0.072	-0.068
Foot Offset 2	-0.050	-0.051
Gain Arms	10.000	9.996
Gain Legs	15.000	14.878
Limit Margin Y	-0.050	-0.100
Step Time	0.850	0.848
ZMP Time	0.400	0.281
Hip Roll Compensation	0.024	0.035
Step Height	0.070	0.052

Table E.2: Walk parameters for the walk engine shown for the original values, Ind_O and the optimised genome of Ind_A, parameters 21 - 46

Name	Ind_O	Ind_A
Step Limits min X	-0.100	-0.100
Step Limits max X	0.100	0.108
Step Limits min Y	0.000	0.000
Step Limits Max Y	0.300	0.300
Step Limits min Angle	0.000	0.013
Step Limits max Angle	0.262	0.262
Step Height Slow Fraction	0.500	0.525
Step Height Fast Fraction	1.500	1.494
Velocity Limits min X	-0.027	-0.025
Velocity Limit smax X	0.300	0.298
Velocity Limits minY	-0.010	-0.005
Velocity Limits maxY	0.010	0.010
Velocity Limits min Angle	-0.800	-0.837
Velocity Limits max Angle	0.800	0.810
Velocity High	0.060	0.058
Aacceleration Limits 1	1.500	1.511
Acceleration Limits 2	0.045	0.045
Acceleration Limits 3	5.000	5.005
Acceleration Limits High 1	0.080	0.082
Acceleration Limits High 2	0.050	0.051
Acceleration Limits High 3	5.000	5.003
Acceleration Turning Factor	0.600	0.628
Single Support Phase Start	0.120	0.118
Single Support Phase End	0.880	0.879