FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FEUP

Optimizing Simulated Humanoid Robot Skills

Luis Rei

Dissertation

Master in Informatics and Computing Engineering

Supervisor: Luís Paulo Reis (Ph.D)

Second Supervisor: Nuno Lau (Ph.D)

17^{th} January, 2010

Optimizing Simulated Humanoid Robot Skills

Luis Rei

Dissertation
Master in Informatics and Computing Engineering

Abstract

Controlling a humanoid robot with a large number of joints and thus, degrees of freedom, presents a complex problem that requires knowledge in multiple fields, including biology, mechanics, physics, electronics and computer engineering. Soccer, a task that was created specifically for humans, requires not just precise and complex motor skills from an individual, but also strategy, planning and cooperation between the elements of the same team. The technologies involved in allowing a team of robots to play soccer span the main areas of artificial intelligence such as design principles of autonomous agents, multi-agent collaboration, strategy acquisition, real-time reasoning and planning and intelligent robotics. The potential social and economic impact of significant developments in these areas coupled with the benefits of having a standard problem for researchers across the world to compete and collaborate, ultimately lead to the creation of the RoboCup initiative. This thesis aims to improve the performance of motor skills such as getting up, kicking the ball and walking developed by the FC Portugal3D team by applying to them an automated optimization process. The robot used in this work is the standard simulated RoboCup 3D robot, a simulated Nao. The advantages inherent to the use of a simulated robot, as opposed to a real robot, facilitate the development of this type of solution as well as the execution of subsequent experiments. Several different optimization algorithms, in particular Hill Climbing, Simulated Annealing, Tabu Search and Genetic Algorithms are adapted to this problem, used and compared. The skills optimized in this work were those responsible for getting the robot to stand after a fall, side-walking and rotating around a fixed point. They are optimized by each algorithm and compared to the original, unoptimized, skills as well as to those of other teams participating in the RoboCup simulated 3D league, where applicable. The achieved results are good, providing skills that performed, according to their specific performance measure such as time elapsed, speed and precision, much better than the original skills, additionally comparing favorably with other teams in the RoboCup 3D simulated league.

Resumo

Controlar um robô humanóide com um número grande de juntas e por isso, graus de liberdade, apresenta um problema complexo que requer conhecimento em múltiplas disciplinas, incluindo biologia, mecânica, física, electrónica e ciência de computadores. Futebol, uma tarefa que foi criada especificamente para humanos, requer não só uma boa coordenação motora individual mas também estratégia, planeamento e cooperação entre elementos da mesma equipa. As tecnologias envolvidas em permitir a uma equipa de robôs jogar futebol abrange as principais áreas da inteligência artificial tais como os princípios de agentes autónomos, colaboração multi-agente, acquisição de estratégia, raciocínio e planeamento em tempo real, robótica inteligente. O potencial impacto social e económico de desenvolvimentos significativos nestas áreas juntamente com os benefícios de ter um problema padrão para investigadores de todo o mundo competirem e colaborarem levou a criação da iniciativa RoboCup. Esta dissertação consiste na melhoria dos comportamentos do robô, tais como levantar-se, chutar a bola, e andar, desenvolvidos pela equipa FC Portugal através da aplicação de de um processo automático de optimização. métodos de optimização automáticos. O robô utilizado neste trabalho é o robô padrão da liga 3D simulado do RoboCup, um Nao simulado. As vantagens inerentes à utilização de um robô simulado, em vez de um robô real, facilitam o desenvolvimento deste tipo de solução e a execução de experiências subsequentes. Vários algoritmos de optimização, em particular Subir a Colina, Arrefecimento Simulado, Pesquisa Tabu e Algoritmos Genéticos, são adaptados ao problema, utilizados e comparados. Os comportamentos optimizados neste trabalho são os comportamentos responsáveis por fazer o robô levantar-se após uma queda, andar lateralmente e rodar em torno de um ponto fixo. Estes são optimizados por cada algoritmo e comparados aos comportamentos originais não-optimizados e aos de outras equipas que participam na liga 3D simulada do RoboCup, onde tal comparação pode ser aplicada. Os resultados alcançados são bons, tendo resultado em comportamentos que, de acordo com as suas métricas de desempenho específicas tais como tempo decorrido, velocidade e precisão, são melhores que os comportamentos originais e comparam-se favorávelmente com as outras equipas na liga 3D simulada do RoboCup.

Acknowledgements

There are far too many people, groups and institutions who have made this work and this moment in my life possible to list here and not have the reader skip this section entirely. Therefore, I'll keep this as short as possible by thanking my supervisors Luís Paulo Reis and Nuno Lau and also the other members of FC Portugal Robocup team, specially Bruno Pimentel and Nima Shafii. Finally, I would like to thank the Department of Informatics Engineering of the Faculty of Engineering, University of Porto for providing the financial means that allowed me to participate directly in the 2010 edition of RoboCup World Soccer championship.

Luís Rei

Agradecimentos

Demasiadas pessoas, grupos e instituições tornaram este trabalho e momento na minha vida possível para serem mencionados aqui sem correr o risco que o leitor passe à frente esta secção. Assim, vou manter os meus agradecimentos o mais curto possível, agradecendo aos meus orientadores Luís Paulo Reis e Nuno Lau e também aos outros membros da equipa do Robocup FC Portugal, em especial a Bruno Pimentel e Nima Shafii. Por fim, quero também agradecer ao Departamento de Engenharia Informática da Faculdade de Engenharia da Universidade do Porto por ter financiado a minha participação na edição de 2010 do campeonato RoboCup World Soccer.

Luís Rei

"It’s not enough that we do our best; sometimes we have to do what’s required."

– Sir Winston Churchill

Contents

1 Introduction **1**
- 1.1 Context and Motivation . . . 1
- 1.2 RoboCup . . . 2
  - 1.2.1 The 3D Simulation League . . . 3
- 1.3 Objectives . . . 4
- 1.4 Thesis Structure . . . 5

2 Simulated Robotic Soccer **7**
- 2.1 Introduction . . . 7
- 2.2 Advantages Of The Simulation . . . 8
- 2.3 The RoboCup 3D Simulation League . . . 9
  - 2.3.1 Server . . . 9
  - 2.3.2 The Monitor . . . 10
  - 2.3.3 Agents . . . 10
  - 2.3.4 Soccer Simulation . . . 13
- 2.4 Conclusions . . . 13

3 Humanoid Skills Specification **15**
- 3.1 Introduction . . . 15
- 3.2 Skill Specification Models . . . 16
  - 3.2.1 Step-based method . . . 17
  - 3.2.2 Sine interpolation . . . 18
  - 3.2.3 Central Pattern Generator . . . 19
- 3.3 Bipedal Walking in FCPortugal Based on Truncated Fourier Series . . . 20
  - 3.3.1 Movements in the Saggital Plane . . . 21
  - 3.3.2 Movements in the Coronal Plane . . . 22
- 3.4 Conclusions . . . 23

4 Optimization **25**
- 4.1 Introduction . . . 25
- 4.2 Hill Climbing . . . 26
- 4.3 Simulated Annealing . . . 27
- 4.4 Tabu Search . . . 28
- 4.5 Genetic Algorithms . . . 29
- 4.6 Particle Swarm Optimization . . . 31
- 4.7 Optimization of Humanoid Robot Skills . . . 33
- 4.8 Conclusions . . . 34

5 Optimizer Development and Implementation **35**
5.1 Parametrized Skills . . . 35
5.1.1 Changing Values . . . 35
5.1.2 Multiple Skills . . . 37
5.2 Transitions . . . 38
5.3 Overview of The Optimizer . . . 40
5.4 Main Optimizer Classes . . . 41
5.5 The Optimization Server . . . 41
5.5.1 Parameters . . . 42
5.5.2 Reading and Transmitting the Skills . . . 43
5.5.3 The Scripts . . . 44
5.5.4 Evaluation . . . 44
5.6 Implementation of The Optimization Algorithms . . . 45
5.6.1 Optimization Variables . . . 46
5.6.2 Hill Climbing . . . 46
5.6.3 Simulated Annealing . . . 47
5.6.4 Tabu Search . . . 48
5.6.5 Genetic Algorithms . . . 48
5.7 The Agent - Optimization Client . . . 49
5.7.1 RoboCup Server Modifications . . . 51
5.8 Conclusions . . . 51

6 Experiments and Results **53**
6.1 Introduction . . . 53
6.2 GetUpBack . . . 54
6.2.1 Skill Description . . . 54
6.2.2 Objective Function . . . 54
6.2.3 Optimization Options . . . 55
6.2.4 Results . . . 56
6.2.5 Analysis . . . 56
6.3 GetUpFront . . . 57
6.3.1 Skill Description . . . 57
6.3.2 Objective Function . . . 59
6.3.3 Optimization Options . . . 59
6.3.4 Results . . . 59
6.3.5 Analysis . . . 60
6.4 SideWalk . . . 61
6.4.1 Skill Description . . . 61
6.4.2 Objective Function . . . 62
6.4.3 Optimization Options . . . 63
6.4.4 Results . . . 64
6.4.5 Analysis . . . 65
6.5 RotateAround . . . 66
6.5.1 Skill Description . . . 66
6.5.2 Objective Function . . . 66
6.5.3 Optimization Options . . . 67
6.5.4 Results . . . 67

6.5.5 Analysis . . . 68
6.6 Comparison With Other Teams . . . 69

7 Conclusion and Future Work **71**
7.1 Conclusion . . . 71
7.2 Future Work . . . 72

References **75**

A Optimizer Configuration Files **79**
A.1 GetUpBack . . . 79
A.2 GetUpFront . . . 80
A.3 SideWalk . . . 80
A.4 SideWalk . . . 81

B Optimized Skills **83**
B.1 GetUpBack . . . 83
B.2 GetUpFront . . . 85
B.3 SideWalk . . . 87
B.4 RotateAround . . . 88

C User Manual **89**
C.1 Compiling the Optimizer . . . 89
C.2 Creating the Optimization Configuration File . . . 89
C.2.1 Common Parameters . . . 89
C.2.2 Algorithm Specific Parameters . . . 90
C.2.3 Behavior Type Specifc Parameters . . . 91
C.2.4 Equalities and Symmetries . . . 92
C.3 Creating the Start Script . . . 92

List of Figures

1.1 RoboCup 3D Simulation League in 2006. Adapted from [war10]. 3
1.2 SoccerBot, used in the RoboCup 3D Simulation League until 2008 and simulated Nao, used since. Adapted from [sim10]. 4

2.1 Diagram of SimSpark communicating with an agent . Adapted from [sim10]. 9
2.2 Simspark Monitor screenshot. Adapted from [dRA08]. 10
2.3 The Nao humanoid robot: real vs simulated. Adapted from [BDR+08]. . . 10
2.4 The joints of the simulated Nao robot. Adapted from [sim10]. 12
2.5 Field Dimensions and Layout. Adapted from [sim10]. 14

3.1 Coronal plane view of proposed walking Sequence. 22

4.1 The FC Portugal kick, developed using optimization. 33

5.1 Schematic of two Rotate Around skills with different radius. The front of the robot always points to the center of the circle. 36
5.2 Various stages of a kick. 37
5.3 A Param Behavior that contains a table of Slot Behaviors. The *get* method returns the appropriate Slot Behavior to be executed. 38
5.4 The ZeroWalk position 39
5.5 A diagram of the optimizer's configuration. Red arrows indicate data transmission with the tip of the arrows indicating the direction in which the connection is established. Black arrows indicate an execution call via a system execution command with the tip indicating which component is being executed. 40
5.6 A class diagram of the Optimizer class and its children, HillClimb, TabuSearch, SimulatedAnnealing and GeneticAlgorithm. 41
5.7 A schematic of a CPG skill in the optimizer's memory. 43
5.8 A diagram of the communication between the optimizer and an agent. . . 44
5.9 Screenshot of the monitor during an optimization, showing the agents executing experiments. 45

6.1 Sequence of images showing the execution of the GetUpBack skill. Each image corresponds to the final state of a slot. 54
6.2 Optimization of the GetUpBack skill: Time vs Iterations 57
6.3 Optimization of the GetUpBack skill: Score vs Iterations 58

6.4 Sequence of images showing the execution of the GetUpFront skill. Each image corresponds to the final state of a slot. 58
6.5 Optimization of the GetUpFront skill: Time vs Iterations 60
6.6 Optimization of the GetUpFront skill: Score vs Iterations 61
6.7 Sequence of images showing the execution of a SideWalk skill. 62
6.8 Optimization of the SideWalk skill: Speed vs Iterations 64
6.9 Optimization of the SideWalk skill: Score vs Iterations 65
6.10 Sequence of images showing the execution of a RotateAround skill. . . . 66
6.11 Optimization of the RotateAround skill: Position vs Iterations. X in blue, Y in green. 68
6.12 Optimization of the RotateAround skill: Score vs Iterations 69

List of Tables

2.1 Available joints for the Nao robot. 11

3.1 Variables that can be specified in a Step Behavior. 18
3.2 Variables that can be specified in a Slot Behavior. 19
3.3 Variables that can be specified in a CPG Behavior. 20

6.1 Algorithm parameters for GetUpBack. 55
6.2 Results for the optimization of GetUpBack - various algorithms compared. 56
6.3 Variables that were changed by the optimization of the GetUpBack skill using the HC algorithm. 59
6.4 Results for the optimization of GetUpFront - various algorithms compared. 60
6.5 Variables that were changed by the optimization of the GetUpFront skill using HC. 62
6.6 Algorithm parameters for SideWalk. 63
6.7 Results for the optimization of SideWalk - various algorithms compared. . 64
6.8 Variables that were changed by the optimization of the SideWalk skill using TS. 65
6.9 Results for the optimization of RotateAround skill - various algorithms compared. 67
6.10 Variables that were changed by the optimization of the RotateAround skill. 68
6.11 Elapsed time for the execution of GetUp skills - various teams compared. 70

Abbreviations

EBNF	Extended Backus–Naur Form
CMA-ES	Covariance Matrix Adaptation Evolution Strategy
CoM	Center of Mass
CoP	Center of Pressure
CPG	Central Pattern Generator
DOF	Degree of Freedom
GA	Genetic Algorithm
HC	Hill Climbing
PFS	Partial Fourier Series
PSO	Particle Swarm Optimization
SA	Simulated Annealing
ODW	Omnidirectional Walking
TS	Tabu Search
TFS	Truncated Fourier Series

Chapter 1

Introduction

1.1 Context and Motivation

The RoboCup international competition uses soccer as a standard problem to foster research in the fields of artificial intelligence and robotics [KAK+98]. FC Portugal team project was conceived as an effort to create intelligent players, capable of thinking like real soccer players and behave like a real soccer team [RL01] competing in the RoboCup simulation leagues. In the 2D simulation league, where the agents are circles that play in a two-dimensional plane, the team won the World championship in 2000 (Melbourne), reached third place in 2001 (Seattle) and won two European championships in 2000 (Amsterdam) and 2001 (Paderborn). In the 3D simulation league, where the agents were, from 2004 to 2006, spheres playing in a three-dimensional field, FC Portugal won the world championship in 2006 (Bremen) and the European championships in 2006 (Eindhoven) and 2007 (Hannover) [RLM09].

In 2007 a new 3D humanoid soccer simulation league was created in order to promote research in the necessary techniques in order to make humanoid robots play soccer. The topics of research include physics, biology, control theory and machine learning with the aim of developing stable biped skills such as walk, turn, get up and kick. The result of this kind of research may be extended to other domains, such as the use on real humanoid robots, which may be able to perform social tasks such as helping a blind to cross a street or elderly people to perform tasks that became impossible to do alone. The use of simulated environments is popular since it allows the developers to make arbitrary or complex tests in the simulator without using the real robot thus avoiding expensive material getting damaged [LM96].

1.2 RoboCup

The RoboCup initiative, which includes the World Cup Robot Soccer, aims to promote artificial intelligence and robotics research by providing a common task for evaluation of various theories, algorithms and agent architectures. The range of technologies spans the main areas of research, such as design principles of autonomous agents, multi-agent collaboration, strategy acquisition, real-time reasoning and planning, intelligent robotics, sensor-fusion and so forth [KAK+98]. The long-term goal of the RoboCup is stated as follows [rob10a]:

> "By 2050, a team of fully autonomous humanoid robot soccer players shall win a soccer game, complying with the official FIFA rules, against the winner of the most recent World Cup of human soccer."

Presently, there are several leagues in RoboCup:

- RoboCup Soccer:
  - Simulation Leagues (2D, 3D and Mixed Reality);
  - Small Size League (SSL);
  - Middle Size League (MSL);
  - Standard Platform League (SPL);
  - Humanoid League.
- RoboCup Rescue:
  - Simulation;
  - Virtual;
  - Real robots
- RoboCup Junior:
  - Dance;
  - Soccer;
  - Rescue;
  - Demonstration.
- RoboCup @Home.

1.2.1 The 3D Simulation League

The 2D Simulation League is one of the oldest leagues in the RoboCup. In it, two teams, of eleven autonomous agents each, play soccer in a two-dimensional virtual soccer stadium. It focus on artificial intelligence and team strategy. Introduced in 2003, the 3D simulation competition increased the realism of the simulated environment used in other simulation leagues by adding an extra dimension and more complex physics [sim10]. Without the necessity of maintaining any robot hardware, the RoboCup Simulation Leagues focus more heavily on artificial intelligence and team strategy. The idea of a simulation league is to develop a virtual agent capable of thinking and acting so that the acquired knowledge can be transferred to the real robots. To make this possible, it is necessary to construct accurate and reliable models of the real robots. In 2007, the 3D spheres (figure 1.1) previously used in the RoboCup 3D Simulation League were changed to simulated models of humanoid robots. The first simulated humanoid robot was the SoccerBot which was based on the HOAP-2 by Fujitsu and can be seen figure 1.2(a). Lastly, in 2008, it was replace by the simulated Nao robot, figure 1.2(b).

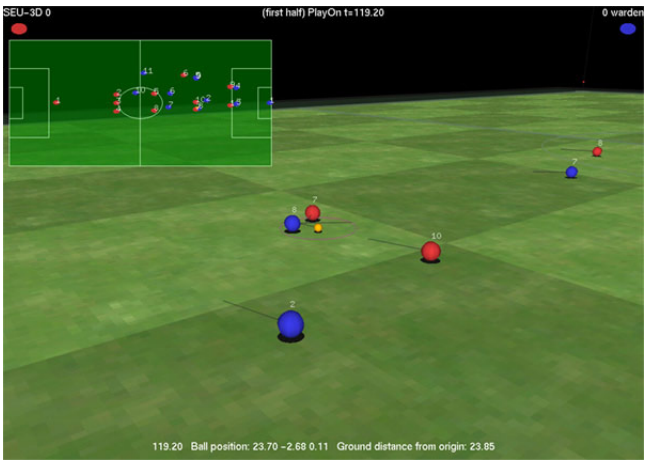


Figure 1.1: RoboCup 3D Simulation League in 2006. Adapted from [war10].

The change to humanoid robots shifted the aim of the 3D Simulation League from strategy towards the low-level control of humanoid robots and the creation of basic skills like walking, kicking, turning [sim10]. The teams currently consist of only six agents and thus, research in coordination has not been very important in the 3D league. Instead, developing efficient low-level skills has been the main decisive factor in the 3D league [LRPS10]. The introduction of the Nao robot as the 3D Simulation League model as well as the official robot for the Standard Platform League now allows researchers to test their algorithms and ideas in a simulated robot before trying them in real robots [sim10]. The two main drawbacks of using real robots is that they are relatively expensive and that the

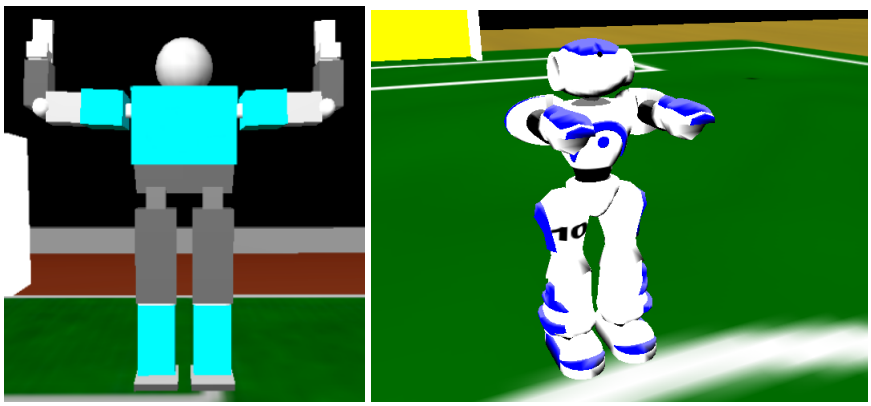


Figure 1.2: SoccerBot, used in the RoboCup 3D Simulation League until 2008 and simulated Nao, used since. Adapted from [sim10].

robots can easily get damaged. Therefore, the price to construct and repair real robots can be a limitation for improvements in the fields related to intelligent robotics.

1.3 Objectives

Robotic joints simulate biological joints, the location at which two bones make contact. They stand at the interception of two rigid bodies and allow them to move in respect to each other. In the context of this work, one movement consists simply of changing the angular position of a joint, which in turn changes the relative position of the two rigid bodies it connects. For example, in the Nao robot, the joint that connects the upper left arm to the lower left arm, designated "left elbow joint", allows the lower left arm to change its position relative to the upper left arm. A skill (or behavior) is composed by a sequence of movements of one or more joints. Examples of skills include walking or kicking ball, in each of these cases, when a human is performing the skill, it is easy to see that the angles of his knee joints assume several different values during the execution of the skill. These basic skills require the precisely timed movement of several joints to precise points. While a human learns those skills through imitation and training during their lives, robots currently require them to be specified.This is often done with the aid of humanoid skill specification models which create a set of angle values for the robot's joints to assume during a time interval, from their current angle values to a desired final angle value.

The main objective of this work is to create an optimizer that allows the optimization of any low-level skill (walk, kick, get up) and use it to develop the best possible low-level skills for the FC Portugal simulated humanoid team such as walk faster (without falling),

get up faster and kick the ball further. These skills are to be created using automatic processes for skill generation and improved with the aid of optimization techniques.

In the process of accomplishing the previously stated goal, different optimization techniques will be applied to each skill and the results of the various algorithms will be analyzed and compared to each other. The final resulting skills will be compared to those of the current FC Portugal skills as well as those of other teams.

1.4 Thesis Structure

Chapter 1 which is this introduction.

Chapter 2 describes the advantages of a simulation and provides more information about the RoboCup Simulated 3D League by detailing its components, the server, monitor and agents, focusing on the last.

Chapter 3 presents skills and the skill specification models used by the FC Portugal team and the files containing a skill specification.Effectively providing the details of the problem with which this work is concerned, namely the decision variables used in the optimization process.

Chapter 4 introduces automatic optimization and shows how the optimization algorithms used in this work, Hill Climbing, Simulated Annealing, Tabu Search and Genetic Algorithms, work.

Chapter 5 shows the initial work developed to enable optimizing skills, the developed optimizer, its different components, how they work together. It also shows how the optimization algorithms were applied to the problem of improving humanoid robot skills, specifically, to the skills of the FC Portugal agent.

Chapter 6 details the experiments performed such as the skills optimized and the options provided to the optimizer, including the parameters of the optimization algorithms, for each skill. It provides the the results of these experiments and compares the algorithms in terms of their performance and the results achieved. The resulting optimized skills are compared to the original, unoptimized, skill and, some, to their equivalent skills in other teams participating in the RoboCup 3D Simulation League.

Chapter 7 provides a conclusion to this work and suggests future improvements.

Chapter 2

Simulated Robotic Soccer

This chapter provides details of the RoboCup 3D Simulation League. It begins by discussing the advantages of a simulation over the use of real robots and then details the simulation environment, including the simulated robots, providing the schematics of the virtual model used. All the work developed in the course of this thesis uses this environment.

2.1 Introduction

A computer simulation, also known as a computer model, is a computer program that attempts to simulate an abstract model of a system. A system is a set of structured interacting or interdependent components that forms an integrated whole. It can be a natural system such as a Coastal Ecosystem [Per10] or the entire universe or it can be human made such as a fighter jet or a personal computer.

Computer simulations have become a useful part of mathematical modeling of many natural systems in physics (computational physics), astrophysics, chemistry and biology, human systems in economics, psychology, social science, and engineering. Simulations can be used to [RR10]:

- Model natural systems or human systems in order to gain insight into their functioning;
- Simulate a technology for performance optimization, safety engineering, testing, training and education - even before it is built;
- Decision making, what if analysis.

The official RoboCup 3D simulation server is Simspark [BDR+08] a generic physical multi-agent simulator system for agents in three-dimensional environments. It simulates the laws of physics in the real world in the context of a soccer game (i.e. the soccer field,

ball and rules of the game) as well as the robots (physical dimensions, look, sensors and joints).

2.2 Advantages Of The Simulation

The robotic platforms (either the most simple articulated arms or the most complex humanoid robots) are usually very expensive. The use of simulation environments for research, development and test in robotics provides many advantages over the use of real robots [Nic04, LM96, Pic08]. The main advantages of the simulation are:

- Less expensive than real robots;
- Easy development and testing of new models of robots;
- Easy testing of new algorithms;
- Less development and testing time;
- The problem can be studied at several different levels of abstraction;
- Possibility to easily add, remove, and test different components;
- Facilitates study of multi-agent coordination methods;
- All tests can be done without damaging the real robot;
- For repetitive tests (e.g. optimization processes), the use of a virtual model is better because the robot will not need assistance to reinitialize every iteration;
- It is possible to retrieve very detailed information from the simulation. It is possible to easily monitor physical measures such as CoM, CoP and ZMP;
- With the quality of simulation environments that exist today, is is possible to use the results obtained by simulation in the real robots with just a few changes;
- Control over the simulation time;
- Comparison of alternative designs, alternative implementations of the same skill and different strategies.

The obvious disadvantage, in this case, is that the simulated system is never equal to the real system. As such, software agents developed for the simulated system will require tweaks in order to be used in the real system. How much will have to change depends on the accuracy of the simulation.

2.3 The RoboCup 3D Simulation League

Simspark is a generic simulation platform for physical multi-agent simulations. This simulator was developed over a flexible application framework (Zeitgeist) to be a generic simulator, capable of simulating anything, since the launch of a projectile to a big soccer game. The framework facilitates exchanging single modules and extending the simulator [OORR05]. The simulation consists of three important parts [BDR+08, BA08, dRA08]: the server, the monitor and the agents.

2.3.1 Server

The SimSpark server manages the simulation, modifying the simulation state in a continuous run loop. Objects in the simulation can change one ore more of their properties like position, speed or angular velocity changes due to influences. The rigid body physical simulation created resolves collisions, applies drag, gravity etc. Agents possess effectors that can modify objects. For example, the agents can change the angular position of the robots joints, creating a movement in such a way that the robot's body touches the ball, creating a collision that changes the velocity of the ball. Another responsibility of the server is to keep track of connected agent processes and at each simulation cycle it sends to the agents their respective sensor information. Furthermore, it receives action sequences from the agents for changing their effectors [BA08]. SimSpark implements a simple internal event model that immediately executes every action received from an agent. It does not attempt to compensate for network latency or for changes in computing resources during the simulation. As a consequence, there is no guarantee that events are reproducible [sim10]. Figure 2.1 illustrates the possibility of desynchronization between the agent and the server: an action which the agent sent according to nth cycle is realized in the (n + 1)th cycle, i.e. the action has been delayed by one cycle [sim10].

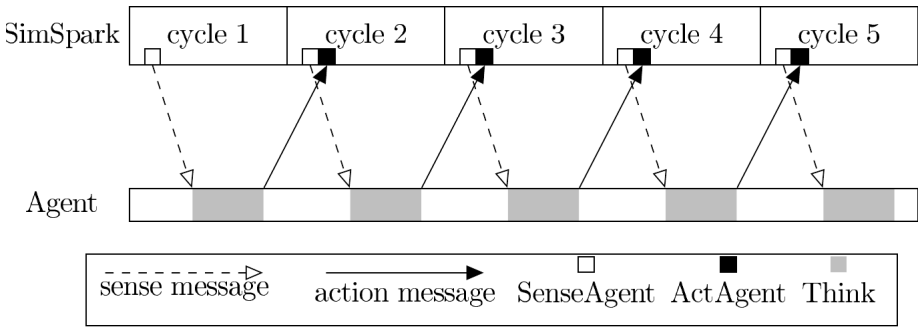


Figure 2.1: Diagram of SimSpark communicating with an agent . Adapted from [sim10].

2.3.2 The Monitor

The SimSpark monitor is responsible to render the current simulation. It connects to a running server instance from which it continuously receives a stream of update data that describes the simulation states either in full or as incremental updates relative to the preceding state. The monitor can further be configured to read a protocol of scene updates from a file and act as a logplayer. In this mode it does not connect to a server instance but replays a recorded game. The format of the logfile is identical to the monitor protocol used on the network.

Figure 2.2: Simspark Monitor screenshot. Adapted from [dRA08].

2.3.3 Agents

RoboCup 3D humanoid soccer league currently uses a virtual model of the Nao robot which is manufactured by Aldebaran Robotics. Its height is about 57cm and its weight is around 4.5Kg. Its biped architecture with 22 degrees of freedom allows Nao to have great mobility. The simulated version is quite similar to the real robot as can be seen in figure 2.3.

(a) real robot (b) virtual robot

Figure 2.3: The Nao humanoid robot: real vs simulated. Adapted from [BDR+08].

The simulated Nao robot model is equipped with a powerful selection of perceptors and effectors to provide a widespread information base for agent development [sim10]:

- Hinge joint effectors and perceptors, a complete list is presented in table 2.1 and their arrangement and relative orientation is shown in figure 2.4;
- A gyroscope and an accelerometer that keep track of radial as well as axial movement in the three dimensional space. Both located at the center of the torso and, therefore, identified by the name "torso";
- One force resistance perceptor in each foot indicates the actual pressure on it, possibly identified by "lf" and "rf", for the left and the right foot, respectively.
- A restricted vision perceptor at the center of the robot's head (limits the field of view to 120 degrees);
- For communication purposes it is equipped with a say effector and the corresponding hear perceptor;
- The gamestate perceptor is used to inform about the actual play time and play mode.

No.	Description	Hinge Joint	Perceptor name	Effector name
1	Neck Yaw	[0][0]	hj1	he1
2	Neck Pitch	[0][1]	hj2	he2
3	Left Shoulder Pitch	[1][0]	laj1	lae1
4	Left Shoulder Yaw	[1][1]	laj2	lae2
5	Left Arm Roll	[1][2]	laj3	lae3
6	Left Arm Yaw	[1][3]	laj4	lae4
7	Left Hip YawPitch	[2][0]	llj1	lle1
8	Left Hip Roll	[2][1]	llj2	lle2
9	Left Hip Pitch	[2][2]	llj3	lle3
10	Left Knee Pitch	[2][3]	llj4	lle4
11	Left Foot Pitch	[2][4]	llj5	lle5
12	Left Foot Roll	[2][5]	llj6	lle6
13	Right Hip YawPitch	[3][0]	rlj1	rle1
14	Right Hip Roll	[3][1]	rlj2	rle2
15	Right Hip Pitch	[3][2]	rlj3	rle3
16	Right Knee Pitch	[3][3]	rlj4	rle4
17	Right Foot Pitch	[3][4]	rlj5	rle5
18	Right Foot Roll	[3][5]	rlj6	rle6
19	Right Shoulder Pitch	[4][0]	raj1	rae1
20	Right Shoulder Yaw	[4][1]	raj2	rae2
21	Right Arm Roll	[4][2]	raj3	rae3
22	Right Arm Yaw	[4][3]	raj4	rae4

Table 2.1: Available joints for the Nao robot.

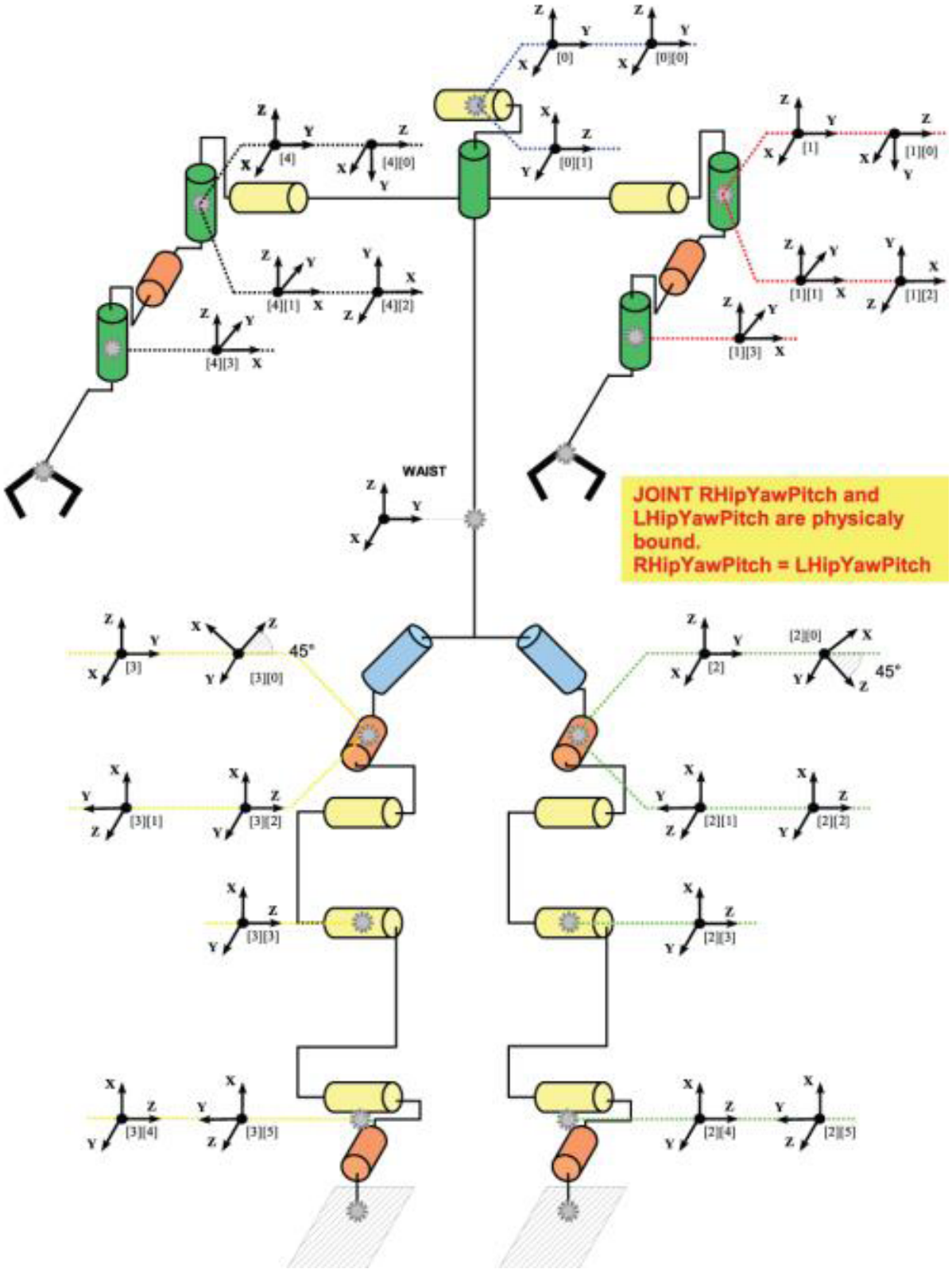


Figure 2.4: The joints of the simulated Nao robot. Adapted from [sim10].

2.3.4 Soccer Simulation

The simulated soccer game has a fixed format: matches are started automatically by a competition managing system which uses the binary agent program, start and kill scripts provided by the competing teams at the beginning of the competition or at the beginning of each round. Each match consists of two halves of five minutes each [Rob10b].

Many rules of human soccer apply to RoboCup matches, such as having a single designated goalkeeper, fouls such as obstructing the ball or deliberately impeding the progress of an opposing player by holding it, result in free kicks for the opposing team. Some rules are specific to the RoboCup, for example, crowding the ball with several players when an opposing player is near or having players immobile for long periods of time result in the repositioning of the infringing players. Additionally, teams that violate the fair play commitment of the RoboCup 3D simulated league by actions such as manipulating competition machines or jamming the simulator, will be disqualified. For each game, a referee is appointed to enforced these rules [Rob10b]. Most rules of the soccer game are judged by an automatic rule set that enforces the basic soccer rule set. However more involved situations like detection unfair behavior still require a human referee to intervene via a Monitor (see subsection 2.3.2).

The simulated soccer field has fixed dimensions of 18 by 12 meters. The center spot has a radius of 1.5 meters. Each goal is 2.1 by 0.6 meter with a height of 0.8 meters. The penalty area to each goal is 3.9 by 1.8 meter. The simulated soccer ball also has fixed physical properties: a radius of 0.04 meter and a mass of 26 grams [sim10]. Figure 2.5 shows the layout and dimensions of the simulated soccer field. It shows the length of the field as a distance defined along x-axis and the width of the field as a distanced defined along the y-axis. This definiton will be used in this work, thus, movement along length or the width of the field will be referred to as movement along the x-axis or y-axis, respectively.

2.4 Conclusions

Simulated environments provide many advantages over the use of real robots, of special relevance to this work is the advantages for the application of optimization and machine learning techniques. The RoboCup 3D Simulation League uses a sort of client-server architecture with the addition of a monitor to visualize their interactions and the resulting world state. An important fact is that it does not guarantee that events are reproducible, which can have consequences for an optimization process that relies on a completely deterministic system. The virtual agent chosen is closely modeled after a real humanoid robot, the Nao, which allows algorithms and ideas developed for the simulated agent to later be applied to the real robot. The simulated Nao has perceptors and effectors, the

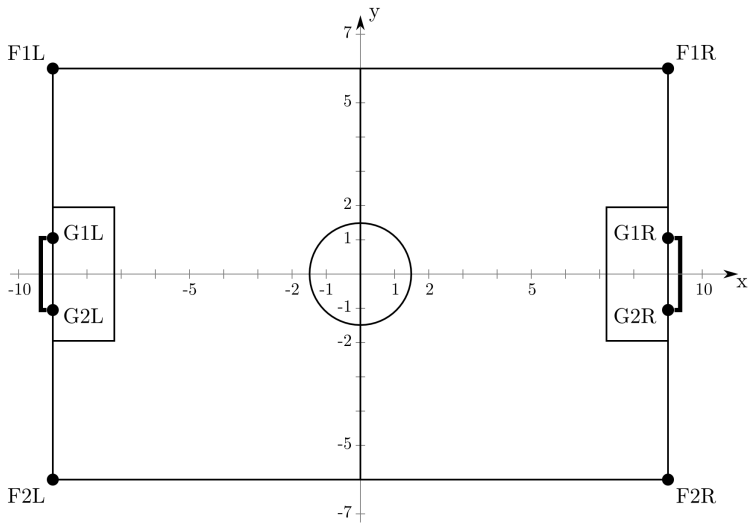


Figure 2.5: Field Dimensions and Layout. Adapted from [sim10].

more relevant of which for this work are the 22 joints and their respective perceptors. The joint effectors will be part of the decision variables later used in the optimization of skills.

Chapter 3

Humanoid Skills Specification

This chapter shows different models of humanoid skill specification which generate values for the robot's joints from a set of defined parameters. These are also known as methods of automatic skill generation and define the skill type to which a skill belongs to. They are used to create the low-level skills of a robot, such as walk, kick, turn, fall down and get up, by specifying parameters to functions which create a set of angle values for the robot's joints to assume during a time interval i.e. not just an initial and final values but also the values between those two. It is then shown how walking has been implemented in the FC Portugal team with the use of a specific model of skill specification.

3.1 Introduction

A skill or behavior consists in a purposeful, repeatable action taken by the executing agent such as getting up from a fall, kicking the ball or walking around the ball. These can be specified once and executed as many times as necessary. In the FC Portugal Simulated Humanoid agent, these skills are specified in an XML file which provides the parameters for a skill specification model.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<behavior name="GetupFront" type="SlotBehavior">
    <slot delta=".200">
        <move id="&lleg1;" angle="-40" />
        <move id="&rleg1;" angle="-40" />
        <move id="&lleg2;" angle="35" />
        <move id="&rleg2;" angle="-35" />
        <move id="&lleg6;" angle="-40" />
        <move id="&rleg6;" angle="40" />
        <move id="&lleg4;" angle="-100" />
        <move id="&rleg4;" angle="-100" />
        <move id="&lleg5;" angle="60" />
        <move id="&rleg5;" angle="60" />
    </slot>
```

```
    <slot delta=".280">
        <move id="&lleg3;" angle="100" />
        <move id="&rleg3;" angle="100" />
    </slot>
    <slot delta="1.500">
        <move id="&larm1;" angle="-90" />
        <move id="&rarm1;" angle="-90" />
        <move id="&lleg3;" angle="0" />
        <move id="&rleg3;" angle="0" />
        <move id="&lleg4;" angle="0" />
        <move id="&rleg4;" angle="0" />
        <move id="&lleg5;" angle="0" />
        <move id="&rleg5;" angle="0" />
    </slot>
    <slot delta=".100" />
</behavior>
```

Listing 3.1: Extract from a skill specification file (GetupFront Behavior of the FCP agent - GetupFront.xml) - December 2010

A skill specification file starts the same way as any other valid XML file, by defining the version of the XML standard used. Following that, the name of the skill and the type of skill specification model used are defined. In this case, the skill is a SlotBehavior which uses sine interpolation, described in subsection 3.2.2, as its specification model. Each robotic joint to be moved is assigned a final angle with its initial angle being its current angle. The specification model will then generate a specific set of intermediate angle values for that joint which will be based on the internal working of the model, the initial and final angles and any other parameters specified. In the case of listing 3.1, the only other parameter specified is the duration of the movement, from the current angle to the final angle, which corresponds to the slot *delta*. Any move tags defined inside a slot tag are affected by the respective *delta* parameter.

3.2 Skill Specification Models

A trajectory can be defined as the set of points followed by an object over an interval of time. In the case of robotic joints, trajectory planning consists of breaking the joint space into many start and end points during the time interval. A skill specification model can be used to specify skills (e.g. walk, turn, get up) by computing different joint trajectories [Pic08]. There are various models for specifying skills, some of those used by the FC Portugal team are described in this section.

3.2.1 Step-based method

This is the method originally used by FC Portugal team. The step-based method generates trajectories using step functions. A step function is a discontinuous function consisting of a series of constant functions, each one defined in some interval of time [Pic08]. The method used for the generation of the joint trajectories consists of a step function whose amplitude is the desired target angle for the joint on each interval of motion. The trajectory equation for each joint is:

$$f(t) = \sum_{i=0}^{n} \theta_i u_{A_i}(t)) \tag{3.1}$$

where n is the number of intervals, A_i is the interval i and θ_i is the desired target angle for the joint at the interval i, u_{A_i} is called the indicator function of A and

$$u_{A_i} = \begin{cases} 1 & t \in A_i \\ 0 & otherwise \end{cases} \tag{3.2}$$

3.2.1.1 Advantages and Disadvantages

The main advantages of the step-based method are [Pic08]:

- Simple to understand;
- Simple to implement;
- Simple to define target trajectories (target angles and tolerances).

The main drawbacks of the step-based method are [Pic08]:

- Time from current angle to target angle is unpredictable;
- No control over the angular velocity trajectory;
- The same gain is used for all joints;
- Sensitive to overshoot reactions at the control level;
- The syntax of the configuration file is not user-friendly;
- The model is not flexible.

3.2.1.2 Implementation

In the FCPortugal agent, skills that use this specification models are simply called Step Behaviors. A skill specification file consists of a series of steps, which will be executed sequentially. Each step can specify multiple moves, where a move is a joint-angle pair, as well as other variables. The variables in table 3.1 can be specified for a skill:

StepBehavior Variables	
Scope	**Parameter**
Step	wait Controller Type (normal or fast) power addPower
Joint	joint identifier Target angle (θ) Maximum speed

Table 3.1: Variables that can be specified in a Step Behavior.

3.2.2 Sine interpolation

Sine interpolation is meant to give better control over each joint trajectory by defining an interpolation of some smooth function for an interval of time between the current angle and the target angle. It allows defining not only the target angles, but also the time in which those angles should be achieved, as well as control the initial and final angular velocities. This is simplified version of the method proposed in [LGCM08].

The central concept of this method is the slot, an interval of time from 0 to δ, where several joints are moved in parallel. In each slot, the controller will interpolate between the current angle and the desired angle by performing a sine-like trajectory in a specified amount of time. Each joint follows a trajectory is given by:

$$f(t) = A\sin(\frac{\phi_f - \phi i}{\delta} + \phi_i) + \alpha_i, \forall t \in [0, \delta] \tag{3.3}$$

where $f(t)$ is the trajectory function, δ is duration of the slot in milliseconds, ϕ_i is the initial phase (which will influence the initial angular velocity), phi_f is the final phase (which will influence the final angular velocity), A is the amplitude and α is the offset. In order to interpolate between the current angle and the desired angle, taking into account the initial and final angular velocities, A and α must be calculated carefully. This is done using the following expressions:

$$A = \frac{\theta_f - \theta_i}{\sin(\phi_f) - sin(\phi_i)} \tag{3.4}$$

$$\alpha = \theta_i - Asin(\phi_i) \tag{3.5}$$

where $theta_i$ and $theta_f$ are the initial and final angles, respectively, and should be defined between $-\pi$ and π.

SlotBehavior Variables	
Scope	**Parameter**
Slot	delta (δ)
Joint	joint identifier target angle (θ) kp, ki, kd

Table 3.2: Variables that can be specified in a Slot Behavior.

3.2.2.1 Advantages and Disadvantages

The main advantages of the sine interpolation method are [Pic08]:

- Simple to understand and implement;
- Time from current angle to target angle is controlled;
- Some control over the angular velocities trajectories;
- The model is flexible;
- PID control allows for controlling the overshoot reactions;
- The motion description language3 is user-friendly and well structured.

The main drawbacks of the sine interpolation method are [Pic08]:

- It is not possible to define more complex sinusoidal shapes;
- The angular velocities trajectories are not completely controlled.

3.2.2.2 Implementation

In the FC Portugal agent, the skills that use this specification model are the Slot Behaviors. A skill specification file consists of a series of slots, which will be executed sequentially. Each slot can specify multiple moves as well as other parameters. The variables in table 3.2 can be specified for a skill:

3.2.3 Central Pattern Generator

Central Pattern Generator is a biologically inspired solution to the problem of skill specification that has been applied to omnidirectional walking (ODW). In nature, CPGs are biological neural oscillators (neural networks or neutral circuits) that generate periodic motor commands for rhythmic movements such as locomotion, under the control of simple input signals [RI06]. These neural oscillators drive the walking rhythm in vertebrates [Gri85] including humans [Nie03]. FC Portugal uses a generic CPG model implemented as a set of sinusoidal oscillators.

CPGBehavior Variables	
Scope	**Parameter**
Behavior	delta (δ)
Pattern	joint identifier target angle (θ) period phase amplitude

Table 3.3: Variables that can be specified in a CPG Behavior.

3.2.3.1 Advantages and Disadvantages

The main advantages of the CPG method are [Pic08]:

- More complex shapes are possible;
- Angular velocities trajectories are completely controlled;
- Allows several gaits with just one generator (forward walk, backward walk, sided walk, curved walk and turn on the spot);
- The generated gait is very similar to the natural human behavior.

The main drawbacks of the CPG method are [Pic08]:

- There is no supporting language;
- Requires complex knowledge about human behaviors;
- The shifting stage leads to slower movements.

3.2.3.2 Implementation

A skill specification file for a CPG skill consists of a series of patterns for each specified joint.The variables can be specified for a skill are presented in table 3.3.

3.3 Bipedal Walking in FCPortugal Based on Truncated Fourier Series

This section details the efficient and robust bipedal walk skill currently used by the FC Portugal team [LRPS10, SRL10]

3.3.1 Movements in the Saggital Plane

Trunk sagittal and coronal plane motion are fairly repeatable [FSG+] therefore Fourier series can be used to model the trajectory of the joints in these planes. In our approach, According to [YCPZ06] legs joint angular trajectories in sagittal plane are divided in two parts; the upper portion and the lower portion. The trajectories for both legs are identical in shape but are shifted in time relative to each other by half of the walking period. The Truncated Fourier Series (TFS) for generating each portion of hip and knee trajectories are formulated below.

$$\theta_h{}^{+} = \sum_{i=1}^{n} A_i \sin(iw_h t) + c_h, w_h = \frac{2\pi}{T_h} \tag{3.6}$$

$$\theta_h{}^{-} = \sum_{i=1}^{n} B_i \sin(iw_h t) + c_h, w_h = \frac{2\pi}{T_h} \tag{3.7}$$

$$\theta_k{}^{+} = \sum_{i=1}^{n} C_i \sin(iw_k t) + c_k, w_k = w_h \tag{3.8}$$

$$\theta_k{}^{-} = c_k \geq 0 \tag{3.9}$$

In these equations, the plus sign represents the upper portion of walking trajectory and the minus shows the lower portion. i = 1 and A_i, B_i, C_i are constant coefficients for generating signals. The h and k subscripts stands for hip and knee respectively. C_h, C_k are signal offsets and T_h is assumed as the period of hip trajectory.

In sagittal plane, during human walking, the arms normally swing in opposite manner to legs, which helps to balance the angular momentum generated in the lower body. The trajectory of arms uses a sinusoidal signal with the same frequency of the legs [Elf39]. Humans swing their arms close to 180^{o} out of phase with their respective legs during walking [CWR01]. It can be expected that the utilization of arm swing provides good performance to yaw moment stability, and recovery from stumbling. The effectiveness of this method is confirmed with an improvement of the accuracy of straight walking at different speeds. According to [SKAJ09], the trajectory of arms is a sinusoidal signal; therefore, to produce proper angular trajectories to arms swing, it is enough to obtain proper parameters for the following equation:

$$f(t) = A \sin(\omega_{arm} t) \tag{3.10}$$

In the above equation, A and ω are assumed as the amplitude and frequency of the signal, respectively. Since legs and arms have the same frequency, ω_{arms} can be considered equal to ω_{legs}. According to the mentioned equation only the proper value of parameter A must be obtained. According to [SKAJ09] , increasing the walking speed and amplitude

from standing to running the robot can walk more stable and faster. FC Portugal implemented a model for the robot to walk from smaller gait with lower amplitude to bigger gait with higher speed and acceleration. In this model a linear equation is used to lead the robot to increase the amplitude of trajectory linearly form zero (stop state) to desired angular trajectories. T is assumed as a parameter to determine how much time is needed for this increment algorithm to reach these desired trajectories.

3.3.2 Movements in the Coronal Plane

The range of motion in the coronal and transverse plane is less than that is seen in the sagittal plane [TRH08] but it has an important role to keep the balance of walking and reach the highest speed of walking. The range of its motion depends on the speed of the walking, at higher speeds this range is smaller. Coronal plane movements are periodic motions [FSG+]. Abduction and adduction are terms for movements of limbs relative to the coronal plane. To produce legs' motion in coronal plane and also considering keeping the balance of robot, we proposed a walking sequences and scenario. Figure 3.1 illustrates the walking sequences in a walking period. θ is assumed as the maximum of legs movement. Like in the previous section in coronal plane, feet were kept parallel to the ground in order to avoid collision and considering of the fact that just one of each hip's joint moves each time, the angle of ankle is equal to the hip's angle of the opposite leg.

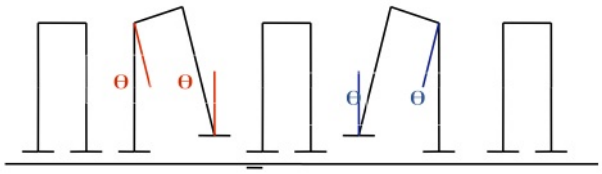


Figure 3.1: Coronal plane view of proposed walking Sequence.

According to the scenario, in order to produce proper angular trajectories to move the hips in coronal plane, proper parameters for the following equations must be obtained :

$$f(t) = H\sin(wt), t < \frac{T_h}{2} \tag{3.11}$$

$$f(t) = 0, t > \frac{T_h}{2} \tag{3.12}$$

In the above equation, H and ω are assumed as amplitude and frequency of signal respectively and T_h is assumed a period of hip. Ankle trajectories can be calculated from hip trajectories and as it is shown, left and right hip angular trajectories are the same but with a phase shift of -π. The period of walking in sagittal plane and coronal plane is equal.

Therefore T_h and ω is eliminated in this method and for producing the proper abduction and adduction, the proper value of H parameter must be found. In this approach the best parameters to generate angular trajectories for bipedal locomotion must be found. For this kind of optimization problem, some authors state that Particle Swarm Optimization can achieve very good results [SANS10].

3.4 Conclusions

There are different models of skill specification, each with advantages and disadvantages. A particular method may produce a more effective specific skill than another method but may not be better for all skills. Effectively, CPG Behaviors are used mostly for walking leg movements such as side walking and rotating while Slot Behaviors are used for everything else except forward and backward walking which is implemented using TFS. At this time, Step Behaviors are no longer widely used.

Chapter 4

Optimization

This chapter gives an overview of the different techniques that will be employed to determine the parameters of the low level skills designed with the use of the skill specification models discussed in chapter 3. The use of these techniques is expected to result in more effective skills than if those parameters were only defined by hand, which in some cases can be very difficult to do due to the complexity associated with the skill and the specification model.

4.1 Introduction

Several problems aim at finding the best configuration of a set of parameters to achieve a goal (or some goals). The goal is either to minimize or to maximize some quantity. This quantity is a function of one or more variables and is known as the **objective function**, f. The independent variables that can change while searching for an optimum are known as the **decision variables**. If the goal is to minimize f, then f is called the cost function (or sometimes, the penalty function). When the goal is maximization, f is referred as the utility function (or sometimes, the benefit function). For some problems, the values that decision variables can take are further specified through a number of conditions known as constraints.

The search space of a problem is defined as the set of all candidate solutions. A candidate solution corresponds to an instantiation of the decision variables, if it satisfies all the constraints of the problem then it is called a feasible solution or just a solution. The solution space is the set of all (feasible) solutions. An optimal solution is a solution for which the objective function evaluates to an optimum (minimum or maximum).

An optimization problem can be either a minimization or maximization problem. A maximization problem can be turned into a minimization problem by symmetry, i.e. maximizing an objective function f is the same as minimizing its symmetric $-f$. For simplicity optimization problems in this chapter will be assumed to be minimization problems,

except if otherwise stated.

Thus an Optimization Problem $P = (S, f, X, D, C)$ can be defined by:

- a set (or vector) of decision variables $X = \{x_1, ..., x_n\}$;
- variable domains $D = \{D_1, ..., D_n\}$;
- the set of constraints among variables, $C = \{C_1, ..., C_n\}$;
- an objective function f to be minimized, where $f : D_1 x ... x D_n \mapsto \Re_+$;
- where the solution space S is the set of all feasible solutions: $S = \{s = \{(x_1, v_1), ..., (x_n, v_n)\}, v_i \varepsilon D_i, s\text{satisfi}$

To solve an optimization problem one has to find a solution $s \in S$ for which the objective function takes the minimum value. This leads to the definition of a globally optimal (minimal) solution, s^* of P is such that $f(s^*) \leq f(s) \forall s \varepsilon S$. The set of all optimal solutions s^* is denoted by S^* and referred as the optimal set.

A candidate solution can also be referred to as an individual. Optimization problems may be broadly divided into two main categories: individual-based methods and population-based methods. Individual-based methods deal with only one current solution. Conversely, in population-based methods counts with a set of individuals (population) that are handled simultaneously [Wei07, Pic08]. The following sections explain some optimization algorithms.

4.2 Hill Climbing

The Hill Climbing (HC) algorithm [Bok81, JPY88] is one of the simplest algorithms to implement and performs well in several situations, specifically when the desired solution is close to the current solution in the search space and there are no local optima between them. Given an initial solution and a neighborhood relation, the hill climbing strategy runs over a graph whose states are treated as candidate solutions. Algorithm 1 shows the pseudo code of a generic HC [Pic08].

In algorithm 1, the procedure *GetInitialSolution* initializes the current solution either with a manually defined solution or a randomly generated one. Iteratively, if the procedure *TerminationConditionsNotMet* returns true, the algorithm keeps the iterative process. Otherwise, it stops the search. *GetNeighborwood* generates the neighbors by applying random variations to each parameter of the individual. *Evaluate* returns the score assigned to the state by the objective function. Each neighbor is evaluated and its score is compared with the score of the current solution. The algorithm chooses as the next state, the one which as a better score than the current solution. This algorithm easily gets stuck in a local optima solution, i.e., none of the neighbors has a better evaluation than the current solution, which might be a poor quality solution. A possible strategy to solve this

Algorithm 1 Hill Climbing

```
CS ⇐ GetInitialSolution()
E1 ⇐ Evaluate(CS)
while TerminationConditionsNotMet() do
  SS ⇐ GetNeighborwood(CS)
  for i = 1 to Length(SS) do
    E2 ⇐ Evaluate(SS[i])
    if E2 < E1 then
      CS ⇐ SS[i]
      E1 ⇐ E2
    end if
  end for
end while
return CS
```

problem is to, sometimes, accept worse solutions. Simulated Annealing [KGV83] aims to solve this problem and will be explained in the subsection 4.3.

4.3 Simulated Annealing

Simulated Annealing (SA) [KGV83] is a method of using a solution to find the lowest energy (most stable) orientation of a system. The method is based upon the procedure used to make the strongest possible glass. This procedure hits the glass to a high temperature so that the glass is a liquid and the atoms can move freely. The temperature of the glass is then slowly decreased so that, at each temperature, the atoms can move enough to begin adopting the most stable orientation. If the glass is cooled slowly enough, the atoms are able to achieve the most stable orientation. This slow cooling process is known as annealing.

SA was one of the first algorithms incorporating an explicit mechanism to escape from local optima. Analogous to the glass annealing problem, the candidate solutions of the optimization problem have correspondence with the physical states of the matter, where the ground state corresponds to the global minimum. The fitness function corresponds to the energy of the solid at a given state. The temperature is initialized to a high value and then decreased during the search process, which corresponds to the cooling schedule. The SA avoids local optima by accepting a solution worse than the current one with a probability that decreases along the optimization progress. Algorithm 2 shows the pseudo code the Simulated Annealing algorithm.

Algorithm 2 Simulated Annealing

```
CS ⇐ GetInitialSolution()
T ⇐ T_MAX
while TerminationConditionsNotMet() do
  NS ⇐ GetNeighbor(SS)
  if Evaluate(NS) < Evaluate(CS) then
    CS ⇐ NS
  else
    CS ⇐ AcceptanceCriterion(CS,NS,T)
  end if
  T = CoolingSchedule(T)
end while
return CS
```

The chance of getting a good solution is a trade off between the computation time and the cooling schedule. The slower is the cooling, the higher will be the chance of finding the optimal solution, but higher will be the time needed for optimization. The neighbor is evaluated and will be subject to acceptance if it gets a score worse than the score of the current solution. The acceptance criterion accepts the new solution with a probability, $p_{acceptance}$, which defined as follows:

$$p_{acceptance} = e^{\frac{Evaluate(CS)-Evaluate(NS)}{T}} \tag{4.1}$$

This probability depends on the temperature, T, and the difference of energy, $Evaluate(CS) - Evaluate(NS)$. The procedure *CoolingSchedule* decides how the cooling schedule is updated. A common approach follows the following rule:

$$T_{i+1} = \alpha T_i, \alpha \varepsilon (0,1) \tag{4.2}$$

where T_i is the current temperature level and T_{i+1} is the scheduled temperature level for the next iteration.

4.4 Tabu Search

In 1986, Fred Glover proposed the Tabu Search (TS) algorithm [Glo86]. TS improves the efficiency of the exploration process since it not only uses the local information (the result of the evaluation function), but also information related to the exploration process (solutions already evaluated). In this way, TS rejects already visited solutions. The main characteristic of TS is the systematic use of memory. While most exploration methods keep in memory essentially the best solution and its evaluation score, TS also keeps a list

of the last solutions visited. In essence, TS declares each node already visited as a *tabu*. Tabus are stored in a list, the *tabu list*, and the search in the neighborhood is restricted to the neighbors that are not in the *tabu list*. Algorithm 3 shows the pseudo code of a simple TS algorithm.

Algorithm 3 Tabu Search

```
CS ⇐ GetInitialSolution()
E1 ⇐ Evaluate(CS)
TabuList ⇐ Empty
while TerminationConditionsNotMet() do
  SS ⇐ GetNeighborwood(CS)
  for i = 1toLength(SS) do
    if NotMember(SS[i]), TabuList) then
      Add(TabuList, SS[i])
      E2 ⇐ Evaluate(SS[i])
      if E2 < E1 then
        CS ⇐ SS[i]
        E1 ⇐ E2
      end if
    end if
  end for
end while
return CS
```

TS will evaluate each member and will pick up the best one. During the search process, TS ignores already tested solutions which are recorded in the *TabuList*, avoiding the cost of evaluating them again.

4.5 Genetic Algorithms

A Genetic Algorithm (GA) [Hol75] is an optimization method inspired by the evolution of biological systems and based on global search heuristics. GA belongs to the class of population-based evolutionary algorithms. In spite of being different, evolutionary algorithms share common properties since they are all based on the biological process of evolution. Given an initial population of individuals (also called chromosomes), the environmental pressure, applied by the fitness (objective) function, causes the best fitted individuals to survive and reproduce more. Each individual (chromosome) is a set of variables (genes) and represents a possible solution to the optimization problem. The algorithm starts by creating a new population of individuals. Typically, this population is created randomly but any other creation function should be acceptable. The genes of each individual should be inside a range of acceptable values (variable domain) that is defined for each gene. The algorithm then starts the evolution which consists of creating a sequence of new populations. At each step, the algorithm uses the individuals in the current

population to create the next population by applying selection, crossover and mutation operators [Pic08].

These genetic operators are described as follows in [Mit98]:

- **Selection**: Specifies how the GA chooses parents for the next generation. The most common option is the roulette option which consists of choosing parents by simulating a roulette wheel, in which the area of the section corresponding to an individual is proportional to its fitness value;

- **Elitism**: Defines the number of individuals in the current generation that are guaranteed to survive in the next generation, usually the ones with the best fitness value;

- **Crossover**: A crossover function performs the crossover of two parents to generate a new child. The most common are the scattered function and uniform crossover, the first creates a random binary vector and selects the genes where the vector is a 1 from the first parent, and the genes where the vector is a 0 from the second parent. Moreover, the crossover function receives a parameter named as crossover fraction, p_c, which corresponds to the fraction of the population that is created by crossover, in the second, genes are of the two parents are swapped with a fixed probability;

- **Mutation**: The mutation function produces the mutation of children. The most common is the uniform mutation which applies random variations to the children using an uniform distribution. Uniform mutation receives a parameter, p_m which corresponds to the probability that an individual entry has of being mutated.

Algorithm 4 Genetic Algorithm

$Population \Leftarrow CreateInitialPopulation()$
$Evaluate(Population)$
while $TerminationConditionNotMet()$ **do**
  [Selection]$Parents \Leftarrow Selection(Population)$
  [Elistism]$Elite \Leftarrow Elitism(Population)$
  [Crossover]$Children \Leftarrow Crossover(Parents, p_c)$
  [Mutation]$Mutants \Leftarrow Mutation(Children, p_m)$
  $Population \Leftarrow Elite + Mutants$
  $Evaluate(Population)$
end while
return $Best(Population)$

In 4, *Best(Population)* returns the individual with the best evaluation (best value of the fitness function).

4.6 Particle Swarm Optimization

Particle swarm optimization (PSO) is based on the behavior of a swarm or colony of insects, such as ants and bees, a flock of birds, or school of fish. The particle swarm optimization algorithm mimics the behavior of these social organisms. The word particle denotes, for example, an ant in a colony. Each individual or particle in a swarm behaves in a distributed way using its own intelligence and the collective intelligence of the swarm. If one individual discovers a good path to food, the rest of the swarm will also be able to follow that path even if their location is far away in the swarm. Optimization methods based on swarm intelligence are called behaviorally inspired algorithms as opposed to evolution-based methods, such as genetic algorithms. The PSO algorithm was originally proposed by Kennedy and Eberhart in 1995 [KE95]. In the context of multivariable optimization, the swarm is assumed to be of specified or fixed size with each particle located initially at random locations in the multidimensional variable space, in this they are equivalent to the initial population of an evolutionary algorithm. Each particle is assumed to have two characteristics: a position and a velocity. Each particle wanders around in the problem space and remembers the best position (in terms of objective function value) it has discovered. The particles communicate information or good positions to each other and adjust their individual positions and velocities based on the information received.

As an example, consider the behavior of birds in a flock. Although each bird has a limited intelligence by itself, it follows the following simple rules:

1. It tries not to come too close to other birds.
2. It steers toward the average direction of other birds.
3. It tries to fit the "average position" between other birds with no wide gaps in the flock.

Thus the behavior of the flock or swarm is based on a combination of three simple factors:

- *Cohesion* - stick together.
- *Separation* - don't come too close.
- *Alignment* - follow the general heading of the flock.

The PSO is developed based on the following model:

1. When one bird locates a target or food (an optimum of the objective function), it instantaneously transmits the information to all other birds.
2. All other birds gravitate to the target or food (the optimum of the objective function), but not directly.

3. There is a component of each bird's own independent thinking as well as its past memory.

Thus the model simulates a random search in the variable space for the optimum value of the objective function: gradually over many iterations, the individuals converge on the or optimum of the objective function [Rao09].

Algorithm 5 Particle Swarm Optimization

```
for all P in S do
    P ⇐ InitializeParticle(P)
end for
while TerminationConditionNotMet() do
    for all P in S do
        E1 ⇐ Evaluate(P)
        if E1 < Evaluate(P_best[i]) then
            P_best[i] ⇐ P
            if E1 < Evaluate(G_best) then
                G_best ⇐ P
            end if
        end if
    end for
    for all P in S do
        UpdateVelocity(P)
        UpdatePosition(P)
    end for
end while
return G_best
```

In algorithm 5, S represents the swarm, P is a particle in the swarm, $P_{best}[i]$ is best fitness value of the ith particle in history and G_{best} is the best fitness value of any particle.

$UpdateVelocity()$ works according to the following formula:

$$V_j(i) = V_j(i-1) + c_1 r_1 \left[P_{best,j} - P_{i-1}\right] + c_2 r_2 \left[G_{best,j} - P_{i-1}\right] \tag{4.3}$$

where $V_j(i)$ is the velocity of particle j in the ith iteration, c_1 and c_2 are the cognitive (individual) and social (group) learning rates, respectively, and r_1 and r_2 are uniformly distributed random numbers in the range 0 and 1. The parameters c_1 and c_2 denote the relative importance of the memory (position) of the particle itself to the memory (position) of the swarm. The values of c_1 and c_2 are usually assumed to be 2 so that $c_1 r_1$ and $c_2 r_2$ ensure that the particles would overfly the target about half the time.

$UpdatePostion()$ works according to the following formula:

$$P_j(i) = P_j(i-1) + V_j(i) \tag{4.4}$$

4.7 Optimization of Humanoid Robot Skills

Optimization has been used for robotic skills successfully. FC Portugal used an optimized kick in the 2D Simulation League when it won the World championship in 2000 [RL01]. In both the RoboCup 3D Simulation League and SPL it is widely used in the humanoid walking. Within FC Portugal 3D team, PSO has been successfully to TFS walking [SRL10] and HC and GA to a previously used Partial Fourier Series (PFS) walking [Pic08]. The kick skill currently used, shown in figure 4.1, was also developed with the use of an optimization algorithm similar to HC [BPR10]. Additionally, instead of optimizing single skills, entire sequences of skills, such as walking forward followed by turning to a side followed by a kick, have been optimized. One of the main objectives in optimizing a sequence of skills, instead of just the individual skills that comprise the sequence, is to decrease the likelihood of the agent falling when switching between skills [UMK+10].

Figure 4.1: The FC Portugal kick, developed using optimization.

4.8 Conclusions

There are many different algorithms that can be used to find exact or approximate solutions to optimization problems. In this work, the optimization problem is that of finding parameters for the low-level skills specified with the use of skill specification models.

The major divisor between the algorithms detailed in this chapter is wether they maintain a single solution in memory on a set of many solutions (i.e. a population).

Different techniques may be more suitable than others for a particular problem, depending on the parameters being searched for, the types of values they hold (e.g. integers, reals,...) and on the properties of the search space.

Chapter 5

Optimizer Development and Implementation

This chapter begins by showing the initial work developed to enable optimizing skills. Skills were made more flexible by parameterizing them. By enabling the engine to dynamically modify the time specified for the execution of the initial slot or step of a skill, they become more efficient. Next, the different components of the optimizer that were developed and integrated with the simulation server and the FCPortugal agent to perform optimization of the agent's skills, are explained.

5.1 Parametrized Skills

Skills are specified by humans via an XML file as was explained in section 3.1. The agent loads the skills specification files at startup and executes them during a soccer match. They are thus inherently static by design: a skill should always perform the same way each time it is executed. There are, nonetheless, circumstances in which some measure of dynamic agent control over the skill is desired.

5.1.1 Changing Values

Skills such side walks, turn around (i.e. "spin") and rotate around a given point are implemented in the FC Portugal code as CPG Behaviors. As an example, consider the case of rotating around a given point: there is no way for the person that programs the skill to know the distance the agent is from the point since the skill is programmed prior to its execution by an agent. Thus, one can either specify a single skill that rotates at a fixed length (i.e. radius) which will often not be the distance to the point one wants the agent to rotate around in every situation or one can specify multiple rotate around skills, each with different radius for the movement, exemplified by figure 5.1. This last option would at least enable the rotate around movement to be specified, with an approximate radius,

for most useful situations. However this approach would be, as has been said, not perfect and furthermore creates a new problem: if the skill's designer needs to change the skill, he would have to change multiple files in the same way, creating the possibility that one or more of these files will either not be modified or be modified incorrectly.

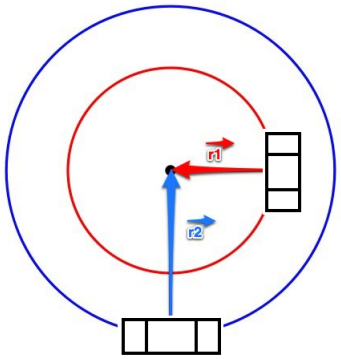


Figure 5.1: Schematic of two Rotate Around skills with different radius. The front of the robot always points to the center of the circle.

A solution to this problem is to allow the functions responsible for handling skills to accept parameters which modify values specified in the file. Continuing with the previous example, the rotate around skill, the radius can be modified by modifying a single value in the specification file. Skills are specified using XML files which are parsed by the agent. A value can be specified using an XML internal entity [BPSM$^+$er]:

```
<!ENTITY amp4 "24">
...
&lleg1;: -&amp4; &period; 1.570796327 0;
&rleg1;: -&amp4; &period; 1.570796327 0;
...
```

Listing 5.1: Example of an XML entity in a skill specification file: &amp4 will be replaced by the text 24 - adapted from a rotate around CPG skill

An entity is just an identifier that replaces a text string therefore, the previous code is equivalent to:

```
...
&lleg1;: -24 &period; 1.570796327 0;
&rleg1;: -24 &period; 1.570796327 0;
...
```

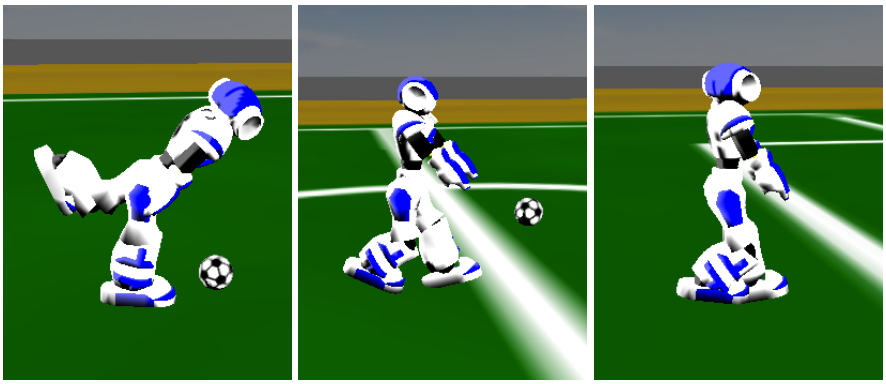
Listing 5.2: Equivalent XML file to Listing 5.1 without using an entity.

Parameters in skills work by replacing the text string in entities. Any value specified using an entity can be modified by evoking the skill with the name of the entity and the value to be attributed to it.

5.1.2 Multiple Skills

Simply changing one value is sufficient for some skills but not for more complex skills. As an example for this section, consider the skill that implements kicking the ball, hereafter referred simply as ”Kick”.

Kicking the ball is a very complex skill that entails several movements and requires compromises between power, reliability and stability. In short, the robot has to shift weight to the support leg, pull the kicking leg back while moving the rest of the body forward to distribute the weight in order not to fall, then move the kicking leg forward to hit the ball with some precision in its center with the center of the foot simultaneously moving the rest of the body back to compensate for the weight of the kicking leg finally, it has to put the kicking leg back on the floor and redistribute the weight between the two legs. In this movement, almost all of the robot's joints are involved.



(a) Pulling back the kicking leg

(b) Hitting the ball

(c) Returning to a walking position

Figure 5.2: Various stages of a kick.

Changing a single value within the kick skill, without changing all the values that are connected to it, would not result in an equally reliable, stable or powerful kick. Changing multiple values from a programming standpoint can be somewhat impractical. It would also require intimate knowledge of the kick skill.

A particular case where one would want to change the values of the kick is when the distance between the agent and the ball is different from the distance that the kick was designed for. The kick is very sensitive to this distance (specially on the y-axis) and while it can be made more robust, in the sense that it will work well, though not optimally, for different distances, there are limits for this flexibility. A solution to this problem is to

optimize the kick for the various distances, for both the x-axis and the y-axis. This can be implemented using a parametrized skill, which consists of a special class, ParamBehavior, which can contain a table of other skills, indexed by two values (e.g. xy distances). The underlying skills can be generated automatically (e.g. via the Optimizer) and must contain their particular indexes. A Param Behavior reads a directory of skill specification files in XML format. Each file contains the skill itself as well as its table index values. Executing a Param Behavior, means calling its *get* method with the index values as parameters (e.g. x and y distance) which returns the underlying skill associated with those index values, such as a Slot Behavior, witch can be executed the same way as any other skill (Figure 5.3). This way, many (e.g. hundreds of) different kick skills can be adequately used, each optimized for a particular situation.

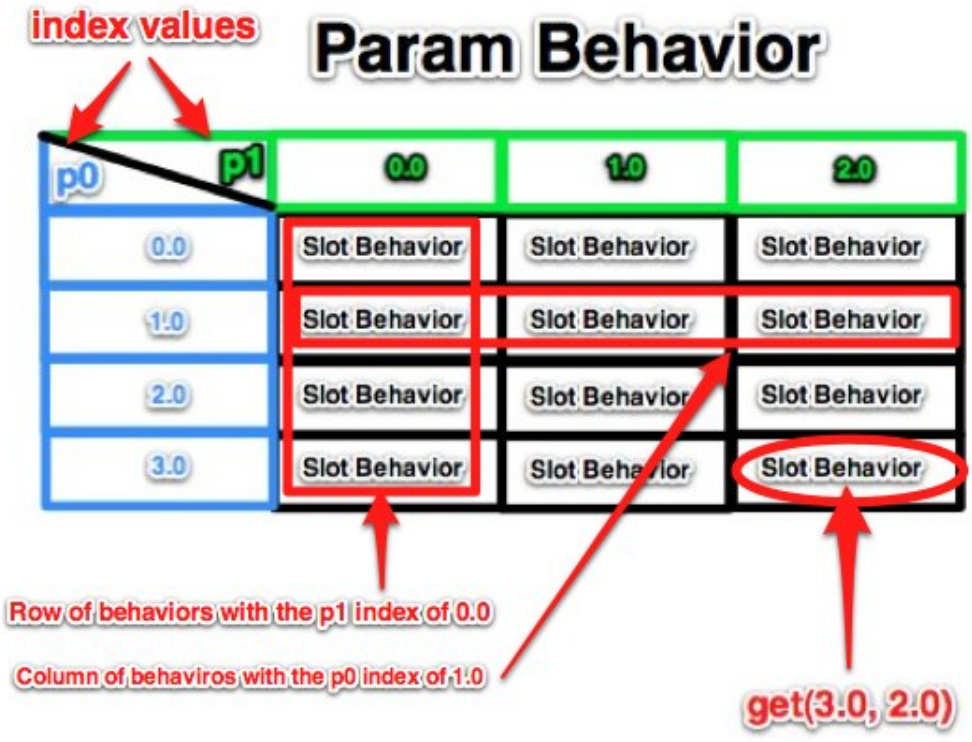
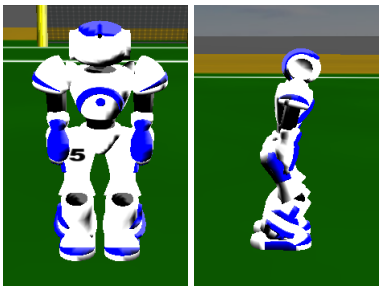


Figure 5.3: A Param Behavior that contains a table of Slot Behaviors. The *get* method returns the appropriate Slot Behavior to be executed.

5.2 Transitions

Most skills currently implemented that are not related to walking are implemented using Slot Behaviors. A Slot Behavior can be divided into multiple slots which in turn can be divided into multiple movements. Slots are executed sequentially and have time intervals

associated with them (subsection 3.2.2). All slots assume that the previous slot was executed and placed the joints in the positions it specified. If this condition isn't true, the skill will not have the result it was intended to have. In order to ensure this does not happen, most Slot Behaviors have a first slot which can be considered as a sort of "initial position" with a relatively long time to execute (δ_0) in order to guarantee that, regardless of the joints previous positions, they end up in that state. This state is for most skills the Zero Walk position (Figure 5.4) - the position from which the agent starts walking forward. In most cases, the agent will not require the allotted time interval to get to that position when transitioning from most other skills. By its nature, this time can not be optimized via the process described in the following chapters: for example, the final joint positions after a side walk will be different from the final joint positions of a back walk and will thus require different amount of time to get to the initial position of a kick. However because one of the advantages of Slot Behaviors is that time from current angle to target angle is controlled, it is possible to dynamically change the time interval of the initial slot of the skill to be executed based on the current joint positions.



(a) Front view (b) Side view

Figure 5.4: The ZeroWalk position

The current joint positions are approximately known. The positions specified in the initial slot of the next skill are also known (they are part of the specification file). Thus one can calculate the maximum distance between the current position and the position specified in the initial slot of the next skill for any joint:

$$maxDistance = \max_{k=0}^{n}(\theta_k^f - \theta_k^i) \tag{5.1}$$

where θ_k^f is the position specified in the next skill's initial slot for joint k and θ_k^i is its current position. Since we know that joints move approximately with a maximum speed of 6^o per simulation cycle and we know how long a simulation cycle takes (it is, by definition, constant) we can calculate a new time interval for the first slot:

$$\delta_{transition} = (maxDistance/6) * CycleTime \tag{5.2}$$

Thus, the first slot always executes with an optimum time of δ.

5.3 Overview of The Optimizer

The optimizer was developed to allow any skill of the agent to be optimized using either a single computer or multiple computers connected via a network. Figure 5.5 is a diagram representing the configuration of the optimizer.

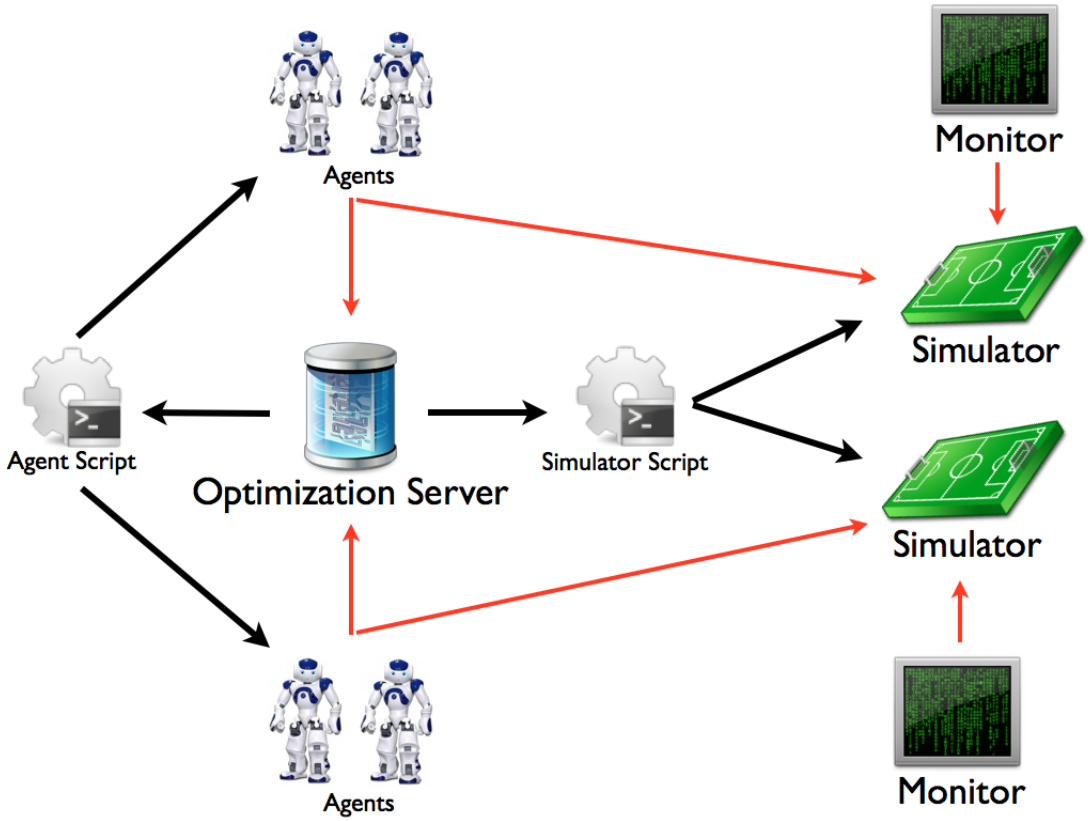


Figure 5.5: A diagram of the optimizer's configuration. Red arrows indicate data transmission with the tip of the arrows indicating the direction in which the connection is established. Black arrows indicate an execution call via a system execution command with the tip indicating which component is being executed.

Once started with the proper parameters, the optimizer executes the simulator script, a shell script which starts the simulator and the monitor. It may start multiple simulators and their respective monitors in different computers if specified. It also executes the agent script which in turn starts the FCP agents. The agents connect to a simulation server (see sub section 2.3.1) and to the optimizer (optimization server) which will provide the agents with the skill to execute and receives from the agents the performance data from that execution. Both these connections are made via TCP [CK74] sockets [Ste98].

5.4 Main Optimizer Classes

The central class of the optimizer is ***Optimizer***. It is responsible for reading the configuration file which contains the parameters for the optimization process, detailed in section 5.5.1 and executing the scripts. It also contains the method that generates neighbors for a solution. And crucially, it is the parent class for the optimization algorithms, which is represented in diagram form in figure 5.6.

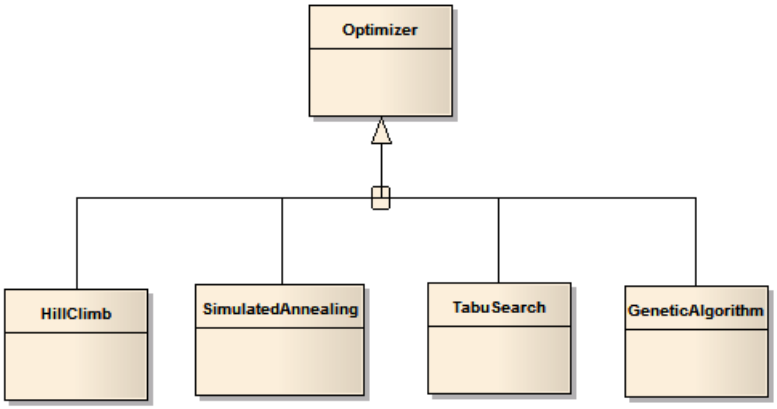


Figure 5.6: A class diagram of the Optimizer class and its children, HillClimb, TabuSearch, SimulatedAnnealing and GeneticAlgorithm.

The algorithm classes, ***HillClimb***, ***SimulatedAnnealing***, ***TabuSearch*** and ***Genetic Algorithm*** contain the implementation of their respective algorithms.

The ***Eval***, or Evaluation, class contains the methods responsible for creating the TCP server, handling connections from the agents and sending them the solution to the executed, receiving afterwards the experiment data from the clients and calling the respective objective function.

The ***Reader*** class contains the method that reads a skill specification file, creating a skill representation in memory. It is also responsible for copying a skill from memory to memory and for saving a skill in memory to a skill specification file. Additionally, it is responsible for converting the memory representation of a skill, a C++ array of arrays containing *double* values, into a string to be sent through the network, from the optimizer to the agent (and vice-versa).

5.5 The Optimization Server

The optimization server is the core of the optimizer. Apart from being responsible for the execution of the other components via the execution of scripts, It contains the functions that read a skill from its specification file, run the optimization algorithm, modify the skill according to the algorithm, send the modified skill (proposed solution) to the agents for

execution, receive the execution data from the agents, evaluate the data according to the specified objective function and, finally, terminates the optimization process via another script.

The optimization server is actually multi-threaded. A thread is started for each agent which creates a specific TCP server which provides an agent with the skill to optimize, waits for the agent to finish executing the skill, receives the experimental data sent by the agent and evaluates it, giving that generated skill a score according to an objective function.

5.5.1 Parameters

The parameters of the optimization process are passed to the optimization server via a configuration file that is passed as **an argument** of the optimizer. Each time a skill is to be optimized with a specific algorithm, a configuration file must be created. The most relevant parameters are the following:

- ***behavior*** - the location of the XML specification file of the skill to be optimized;
- ***type*** - the type of the skill to be optimized (e.g. SlotBehavior), this refers to the specification model used;
- ***objective*** - the name of the objective function to be used, both the function and its name must be defined in the optimizer's source code;
- ***script*** - the location of the script that starts the agents;
- ***algorithm*** - the algorithm to be used in the optimization process (e.g. "ga" for genetic algorithms);
- ***numExp*** - the number of experiments to perform for each skill generated by the optimization algorithm;
- ***nMax*** - the maximum number of iterations for the algorithms main loop;
- ***numThread*** - the number of agents to run simultaneously in each iteration of the optimization algorithm and, therefore, also the number of generated skills to be tested simultaneously;
- **algorithm parameters** - the parameters of the algorithm such as the population size for genetic algorithms or the size of the tabu list for a tabu search;
- **skill optimization parameters** - the specific parts of the skill to be optimized such as which slots or steps for Slot Behaviors or Step Behaviors, respectively, which joints and restrictions to enforce such as symmetries between joints;

- ***minChange*** - minimum value to change an optimization variable by adding to it to obtain a neighboring solution (this should be a negative number, otherwise changes will always be positive values) - does not apply to time intervals (e.g. slot execution time);
- ***maxChange*** - maximum value to change an optimization variable by adding to it to obtain a neighboring solution (this should be a positive number, otherwise changes will always be negative values) - does not apply to time intervals (e.g. slot execution time);
- ***minChangeDelta*** - minimum value to change an optimization variable that represents a time interval by adding to it to obtain a neighboring solution (this should be a negative number, otherwise changes will always be positive values);
- ***maxChangeDelta*** - maximum value to change an optimization variable that represents a time interval by adding to it to obtain a neighboring solution (this should be a positive number, otherwise changes will always be negative values).
- ***serverRestartTime*** - maximum amount of time the simulator can run without being restarted (in seconds).

5.5.2 Reading and Transmitting the Skills

In Chapter 3 it was explained that skills are specified and stored as XML files which are loaded when an agent starts. The optimization server has functions which read the skill to be optimized from its XML file and load the skill into an array of arrays in memory with each value being a C++ double. Figure 5.7 shows a schematic of this memory representation for CPG skills. It is this array representation of the skill that will be modified by the optimization algorithm. This representation is converted to a string and then transmitted to the agents, via a socket, to be executed. To perform this function, the optimization server creates a TCP server to which the agents connect. After performing the experiment (executing the skill), the agents will send the experiment data to the optimization server. Figure 5.8 shows a diagram of the communication sequence.

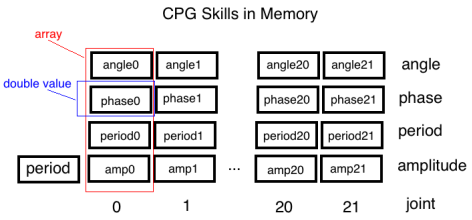


Figure 5.7: A schematic of a CPG skill in the optimizer's memory.

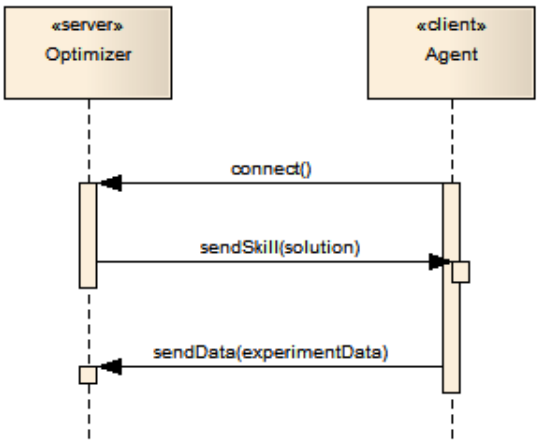


Figure 5.8: A diagram of the communication between the optimizer and an agent.

In the string containing the skill, sent from the optimizer to the agent, the values are space separated while the joints are separated by line breaks. In Extended Backus–Naur Form (EBNF) the message for a CPG skill is represented by:

```
skill = period, {``LB'', joint};
joint = amplitude, `` '', period, `` '',
          phase, `` '', angle;
```

5.5.3 The Scripts

An agent script, specific to each skill to optimize, is responsible for starting the FCP agents and providing them with their parameters. This allows the agents to be started on the local machine or in a remote machine. Another shell script is responsible for terminating the agents. In the current implementation, for convenience, the simulator script is started from the agent script. The simulator script starts both the simulation server and the its monitor.

The scripts exist for both flexibility and to isolate the optimization server from the specifics of running the other components. Recompiling the optimization server each time we wanted to change the parameters of the FCP agents such as the IP addresses of the simulation or optimization server changed would be a poor design choice.

5.5.4 Evaluation

Once an agent finishes a single experiment with a skill, it sends the experiment data (e.g. time taken, distance travelled by the agent, if the agent fell, etc) to the optimization server. For each modified skill, the agent may execute several experiments, defined by the *numExp* variable in the optimization configuration file (see 5.5.1). Once all the experiments

are finished, the optimizer calculates the score of each according to the objective function specified in the configuration file and defined in the code. It then calculates the average, without the worst scored experiment, and uses that as the score for the modified skill which is then used by the optimization algorithm to determine what to do with that proposed solution. Figure 5.9 shows the simulation while experiments are being performed and evaluated.

Figure 5.9: Screenshot of the monitor during an optimization, showing the agents executing experiments.

5.6 Implementation of The Optimization Algorithms

One of the most important factors to mention is that a single experiment can take a long time - it will take as long as the solution, a skill, takes for the agent to execute plus the amount of time that is necessary to wait after the skill is executed before being able to evaluate it. This amount of time is defined in the preparation function for the optimization of the skill. It exists because certain data of the execution of a skill can only be collected some time after the execution of the skill has finished. For example, the agent can fall after getting up or it might be necessary to wait for the ball to stop after a kick has been executed. Skills take in the order of seconds to execute. The following formula can be used estimate the time the optimization will take:

$$Time = AverageSolutionTime + WaitTime * numExp * nMax \tag{5.3}$$

Considering an average solution time of 3 seconds, with a 1 second wait time, with 3 experiments per solution and 300 solutions tested (one per optimization iteration) it would take the optimization process at least 3600 seconds or 1 hour to execute. Additional time is spent on various tasks: because of problems with the simulator, it has to be restarted at given intervals (e.g. 500 seconds), restarting a simulation server and, obviously the agents and monitor along with it, takes approximately 15 seconds on the computer used to develop the optimizer. Further, agents need to be reset (placed in the skill's start position) and stabilized before each execution. This time varies depending on the skill being tested.

In order to speed up this process, the optimizer tests various solutions simultaneously, that means various agents running in parallel thus implying that the algorithms generate and test multiple solutions per iteration.

5.6.1 Optimization Variables

The optimization variables are the parameters that can be specified in the skill specification file. These were described in Chapter 3 for each type of skill. In the optimizer configuration file it is possible to specify the subset of the skill's parameters to be used in the optimization process. It is possible to specify which steps or slots, which joints and which skill type parameters of the joint movement will be optimized.

5.6.1.1 Equalities and Symmetries

There are restriction enforced on the optimization process due to the nature of skills that reduce the size of the search space significantly. For example, for many skills, the movements on the robots left joints (arms and legs) are equal to the joint movements on the right side such as in the current GetUp movements (see figure 6.4). They can also be symmetrical, i.e. the angle of one joint is always equal to -angle of another joint. These restrictions can also be specified in the optimizer configuration file.

5.6.2 Hill Climbing

Most skills designed can easily benefit from some optimization. This is the simplest algorithm implemented. It was described in section 4.2. Two different implementations of this algorithm were developed. The first, denominated simply HillClimb, changes the selected optimization variable by a value that increases, by a constant factor called acceleration, with better solutions and decreases with worse solutions. The other, denominated HillClimbRandom, always changes the selected optimization variable by a random value in a specified range. The algorithm works in the following manner:

1. Assign the original skill to the current solution;

2. For *nMax* iterations:
   (a) Generate *numThread* neighboring solutions from the current solution by copying it and modifying a single random variable for each new solutions;
   (b) Evaluate the generated solutions;
   (c) If the best generated solution is better than the current solution, make it the current solution.

5.6.3 Simulated Annealing

As explained in section 4.3, SA is a local search algorithm that implements a mechanism for escaping from local optima in order to find global optima which it might still not find. The implementation follows the previously explained algorithm:

1. Assign the original skill to the current solution;
2. Assign the original skill to the best solution found;
3. Assign the initial temperature to the current temperature;
4. For *nMax* iterations:
   (a) Generate *numThread* neighboring solutions from the current solution by copying it and modifying a single random variable for each new solutions;
   (b) Evaluate the generated solutions;
   (c) If the best generated solution **is better** than the best solution found:
      i. assign the new solution to the best solution found;
      ii. make it the current solution;
      iii. cool down (reduce the current temperature by a percentage);
   (d) If the best generated solution **is not better** than the best solution found:
      i. Select a random solution from the ones generated in this iteration;
      ii. Calculate the probability of acceptance for the selected solution, using the current temperature;
      iii. If the probability of acceptance is higher than a randomly generated value, make the selected solution the current solution;
   (e) If a better solution hasn't been found in *saRestart* iterations, start searching from the best solution found (by assigning the best solution found to the current solution).

The optimizer parameters provided specifically for simulated annealing, via the skill's optimization configuration file, are the following:

- ***temperature*** - initial temperature;
- ***cool*** - percentage to reduce temperature by with every new best solution found;
- ***saRestart*** - maximum number of iterations without finding a new best solution, after those, the search is restarted at the best solution found;

5.6.4 Tabu Search

The key difference between the algorithm described in section 4.4 and the one described here is the use of an additional tabu list for the search direction (determined via the gradient) to prevent searching in the direction of previous, worse, solutions.

1. Assign the original skill to the current solution;
2. For *nMax* iterations:
   (a) Generate *numThread* neighboring solutions from the current solution by copying it and modifying a single random variable for each new solutions, **ensuring** that neither the generated solutions nor their directions are tabu;
   (b) Evaluate the generated solutions;
   (c) Add the generated solutions to the tabu list;
   (d) If the best generated solution is better than the current solution, make it the current solution.
   (e) Add the direction from the new best solution to the old best solution (inverse search direction) to the tabu list.

The specific parameters for tabu search are the following:

- ***listsize*** - the size of the tabu list for solutions.

5.6.5 Genetic Algorithms

Genetic algorithms are currently the only population based algorithm and the only global search algorithm implemented. Genetic algorithms were described in section 4.5. The implementation is the following:

1. The original skill is loaded;
2. The initial population is created by copying the original skill and making large (random) mutations on the copies;
3. For *nMax* generations:

(a) Evaluate the individuals in the population (*numThread* solutions at a time);

(b) Rank the individuals (proposed solutions) based on their fitness (objective function score from the evaluation), the top *nElite* solutions are considered the elite;

(c) Use roulette-wheel selection to select individuals for crossover - selecting a total of population size minus number of elite solutions individuals;

(d) Perform the crossover using uniform crossover and replacing the least fit individuals with the new individuals created;

(e) Perform single point mutation on the individuals;

4. Perform local optimization using hill climbing on the best *nElite* individuals of the final generation.

There is the assumption that even the best solutions generated after the final iteration of a genetic algorithm, a global search, are not the best local solutions. Hill climbing, a local search algorithm, is performed to obtain better optimized results. The specific parameters for genetic algorithms are the following:

- ***nElite*** - number of elite solutions (solutions to keep each generation);
- ***popSize*** - size of the population (total number of different solutions every generation);
- ***pMutate*** - probability of mutation for each gene;
- ***pCrossover*** - probability of selecting a given point for crossover.

5.7 The Agent - Optimization Client

In the context of this work, the agent is the FCPortugal agent, which was modified for this optimizer. The modification was, deliberately limited to adding a single class, ***Optimize*** and a few optional command line arguments. The operation of the agent is as follow:

1. Initialize the agent;
2. Connect to the optimization server and get the skill to execute;
3. Prepare the experiment: executes a skill specific function that sets the position, via beaming, of the agent in the simulation, executes any preparation skill (such as falling for getups) and initializes variables such as the timer;
4. Wait - waits before executing the experiment to ensure the preparation skill has placed the agent in the desired position (for instance, the fall skill is finished executing while the agent is still falling since all the joints are already in their desired

positions, even tough the agent isn't on the ground at that time) or that the agent is stable after beaming and switching to the ZeroWalk position;

5. Execute the experimental skill;

6. Wait - waits for a while after the skill executed, this is mostly to ensure the stability of the executed skill as the agent might fall after the skill has finished executing because it wasn't entirely stable, this waiting time is defined in the preparation function and might be zero;

7. Collect and send the experiment data to the optimization server.

The experiment data that is being sent to the optimization server by the agent (figure 5.8) currently consists of the following items:

- ***time*** - time taken to execute the skill, not that if the agent got "stuck" somehow (e.g. arms stuck behind the legs trying to get up), and a timeout on the experiment occurs, this variable will be assigned a very large value to indicate this occurrence (e.g. 1000 seconds);

- ***fall*** - a boolean value indicating if the agent was on the floor after the skill finished executing (for most skills, this would be a bad thing but not necessarily or maybe not important);

- ***xdist*** - distance travelled by the agent in the x-axis;

- ***ydist*** - distance travelled by the agent in the y-axis;

- ***direction*** - orientation of the agent after the skill finished executing;

- ***bdistx*** - distance travelled by the ball in the x-axis;

- ***bdisty*** - distance travelled by the ball in the y-axis.

It is simple to add more data to send as it would imply simply collecting it and modifying the structure that holds this data which is defined in a header file included by both the agent and the optimization server that uses it in the objective function for evaluation of the solution tested. Not all objective functions use all the data and the weight attributed to each datum is defined in the objective function, executed in the optimization server, not on the clients. Therefore, changes to the weigh associated with each datum do not require any changes to the agent.

The agent sends the experiment data to the optimizer as a string of space delimited values. In EBNF:

```
experiment data = time, `` '', ``fall'', `` '',
     xdist, `` '', ydist, `` '',
     direction, `` '', bdistx, `` '',
     bdisty;
```

5.7.1 RoboCup Server Modifications

To allow for the optimization process, some modifications to the simulation server had to be made. The first modification was to allow for the game to start automatically with a single agent and/or a single team on the field. Without this modification, the simulator would only start the game once both teams report ready. The second modification was to allow the beam skill, which instantaneously transports an agent to a location of its choosing, to be used during regular gameplay. The original simulator only allows agents to beam during special situations such as when the game begins or at halftime The final modification of the simulator source code was for the true ball position to be passed to the agent via a special state variable. Without this, the agent would only know the position of the ball if it could see where it is using its vision sensors which are subject to a random error that simulates the inaccuracy of human vision when it comes to accurate distance measurement.

5.8 Conclusions

The generic design of this optimizer allowed for the different algorithms to be implemented easily and new algorithms, wether individual or population based can be added in the future without any other changes to the existing optimizer. While the different components, such as the and evaluation functions, require each other in order for the optimization process to work, their modular and generic design allows for them to be swapped for others similarly designed components, such as new algorithms or evaluation functions, without additional changes other than their implementation. It would also be possible to switch to a new agent other than the FC Portugal agent or another simulation server without changing the other components. This, however, would require changes to the new agent and would probably also require changes to the simulation server to provide the agent with data and abilities it would normally not provide in a competition environment, as it did to both the FC Portugal agent and the SimSpark simulation server. The ability of easily specifying new evaluation functions, coupled with the ability to configure the optimization process with the use of a configuration file allows new experiments to be performed quickly and without the need to modify or understand the optimizer's code base. Furthermore, the use of start and kill scripts, coupled with the use of network communication protocols between the optimizer and the agents, allows for optimization

process to be performed in a distributed manner, across networked computers, with ease by simply modifying the start and kill scripts.

Chapter 6

Experiments and Results

This chapter shows the experiments performed, using the developed optimizer, for enhancement of different skills. It includes the description of the optimized skills, a comparison of the results obtained using different algorithms for each skill and finally provides a comparison between some of those skills and their equivalents implemented by other RoboCup teams.

6.1 Introduction

As already pointed out in section 5.6 it was explained that most of the time taken by the optimization process is spent in the execution of the skills. The same number of iterations will take the same amount of time, independently of the algorithm being used. The exception are population based algorithms which must test most or all the solutions in their population at each iteration. The total number of solutions tested by the single solution algorithms is:

$$SolutionsTested = numThread * iterations \tag{6.1}$$

While the total number of solutions tested by genetic algorithms is (approximately)

$$Solutionstested = (popsize - elite) * GA_{iterations} + elite \tag{6.2}$$

Elite solutions are not tested at each cycle since their scores have not changed since they were previously calculated. Thus we only need to consider them in one iteration. Another particularity of our implementation of genetic algorithms is that they will test more "bad" solution because many of the initially generated solutions and their descendants, will be very different, by choice, from the original human designed skill. These solutions will not only take longer to execute but will often never accomplish the goal of the experiment and will require termination by timeout.

This chapter shows the optimization process of four different skills: GetUpBack, GetUpFront, SideWalk and RotateAround. For each of these skills, the four implemented optimization algorithms, HC, SA, TS and GA, are applied. This means a total of sixteen optimizations were made. For each of these, the values of their final objective functions scores and of their most relevant measures, such as speed in the SideWalk, are shown in a table - one table for each skill. Furthermore, the evolution of these values, scores and measures, during the course of the algorithms iterations is shown.

Optimizations were performed using a computer with an Intel Core 2 Duo P8700 processor and 4GB of memory.

6.2 GetUpBack

6.2.1 Skill Description

The GetUpBack skill is a Slot Behavior used to get the robot up on his feet after it fell on its back, intentionally or, more likely, otherwise. This is a very important movement as falls are somewhat frequent during a match at this stage in the development of the FCP agent and other teams and the fact that a robot is mostly useless while on the floor. The execution of the GetUpBack skill is shown in Figure 6.1

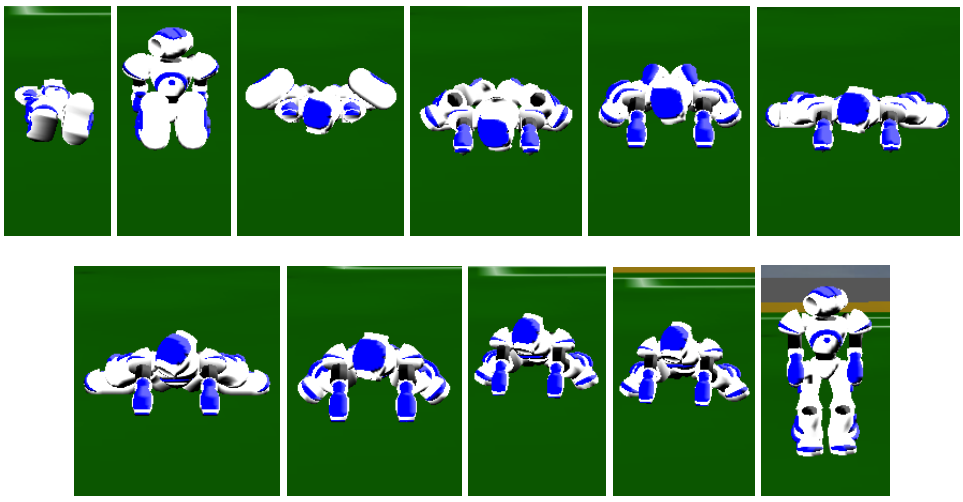


Figure 6.1: Sequence of images showing the execution of the GetUpBack skill. Each image corresponds to the final state of a slot.

6.2.2 Objective Function

There are two measures of the performance of the GetUpBack: time taken and stability. Stability, in this context, means that the robot reliably gets up and does not fall after or

while getting up. This measure is by far the most important. Accordingly, the objective function will give a very low score to experiments that result in falls. The time is measured by how long it takes to execute the GetUpBack skill. The objective function is the following:

$$f = 1000 - t * w_t - s * w_s \tag{6.3}$$

Where t is the time taken to execute the GetUpBack skill, $w_t = 10$ is the weight associated with the time and s, with its associated weight $w_s = 500$, is a binary measure of the stability of the agent and is set according to:

$$s = \begin{cases} 1 & \text{if the agents falls during execution of the skill} \\ 0 & \text{otherwise} \end{cases} \tag{6.4}$$

6.2.3 Optimization Options

The parameters in table 6.1 are described in subsections 5.5.1 and 5.6.1.

Algorithm	**Parameter**	**Value**
Common	Number of Experiments	14
	Threads	6
	Minimum Change for Angles	-5.0
	Maximum Change for Angles	5.0
	Minimum Change for Deltas	-0.4
	Maximum Change for Deltas	0.4;
HC	Number of Iterations	500
SA	Number of Iterations	500
	Temperature	450.0
	Cooldown Factor	0.95
	Restart	30
TS	Number of Iterations	500
	Tabu List Size	1000
Genetic Algorithm	Number of Generations	100
	Population Size	24
	Number of Elite Solutions	6
	Mutation Probability	0.5
	Crossover Probability	0.2

Table 6.1: Algorithm parameters for GetUpBack.

The skill consists of 11 slots (from 0 to 10), all slots where optimized except the first slot which serves as a ”reset slot”. For the last slot, which sets the position of the robot to the previously mentioned ZeroWalk position, only the delta was optimized. Because the skill was designed so that the right side and left side joints moved simultaneously to the

same angles, this constraint was enforced during the optimization process. A total of 47 different decision variables (see section 4.1), consisting of joint angles and slot deltas (see table 3.2), were used in the optimization of this skill. The large number of experiments executed for each proposed solution (14) is due to the importance given to the stability in an inherently unstable skill. The point is to ensure that, no matter how many times the skill executes, it never falls.

For the entire configuration file used to obtain the best result see Appendix A.1.

6.2.4 Results

The results of the optimization process using different algorithms are shown in table 6.2 with the best result highlighted.

Algorithm	Iterations	Elapsed Time	Score
Original	NA	3.00 (s)	816.26
Hill Climbing	500	1.95 (s)	980.72
Simulated Annealing	500	2.09 (s)	979.85
Tabu Search	500	1.96 (s)	980.34
Genetic Algorithm	150	3.00 (s)	816.26

Table 6.2: Results for the optimization of GetUpBack - various algorithms compared.

The graphics in figure 6.2 show the evolution of the skill's main performance measure, execution time, over the course of each algorithm's iterations while figure 6.3 shows the evolution of the objective function's score. The red dots show where better solutions where found.

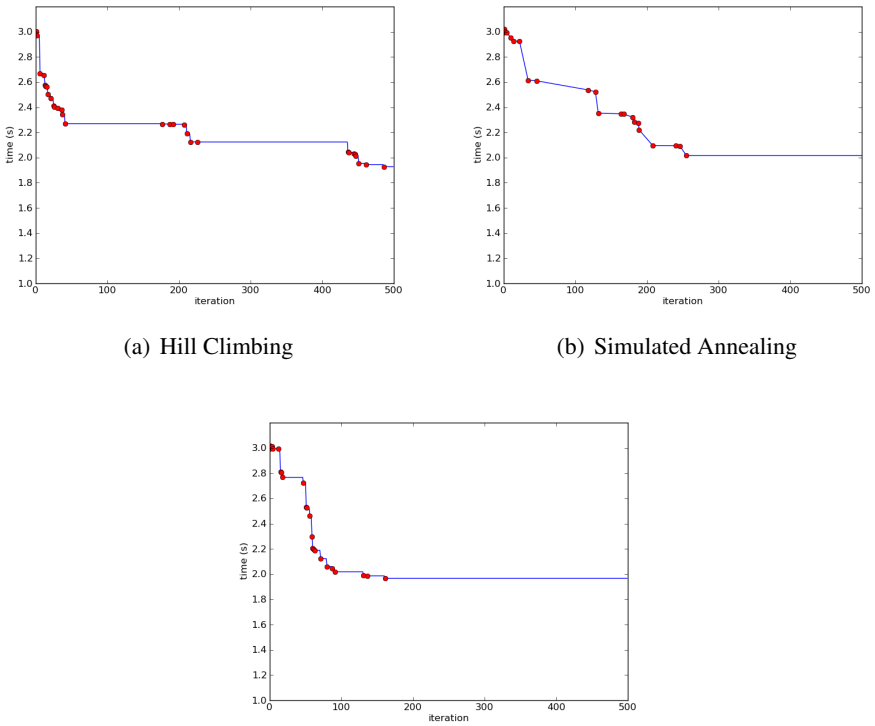
Table 6.3 compares the original skill to the skill optimized using HC in terms of its decision variables which are part of the skill specification file.

The full optimized skill is shown in Appendix B.1.

6.2.5 Analysis

Table 6.3 shows that the most relevant changes made by the optimization process to the decision variables are those made to the deltas, however some might have only been possible due to previous changes to related target angles, both in preceding and succeeding slots. Slot 4 was set to a negative delta which the generator assumes is a value of 0 and effectively means the slot will not be executed. According to the current implementation of the simulation server, the values of delta only have a different effect if they differ by more than 0.02 (s) which corresponds to the time of a simulation cycle.

It is important to note that while the results show the time taken for the robot to get up, that does not represent the score of the solution tested which also factors the stability of the skill (i.e. if it is on the floor after the skill executed). Thus a faster GetUpBack is not



(a) Hill Climbing

(b) Simulated Annealing

(c) Tabu Search

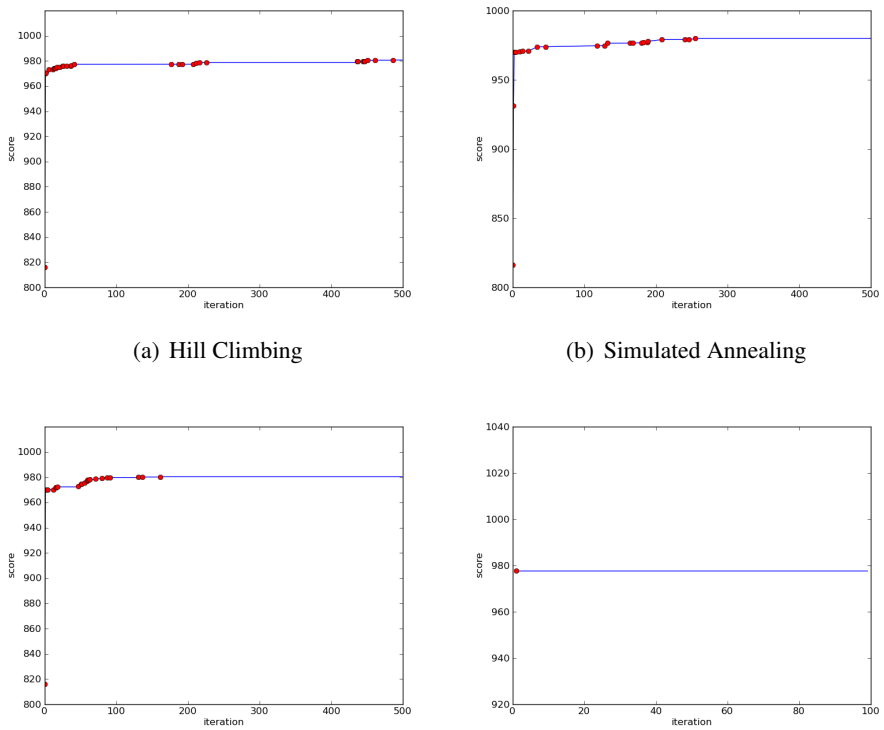
Figure 6.2: Optimization of the GetUpBack skill: Time vs Iterations

necessarily better than a slower one. This is evident in the "time vs iteration" graph for SA, figure 6.3(b), where peaks of faster solutions than the one chosen are clearly shown. It's also interesting to see that TS reached an equally performing solution much faster than HC. The use of the optimizer resulted in a skill that is approximately 30% better in terms of execution speed than the original skill which translates to a 1s difference. GA algorithms failed to find a single better solution, even after testing with various different parameters. The algorithm is made for a more global search than the rest of the algorithms which is clearly not suited for this problem - optimizing human designed robot skills in general and the complex GetUp skills in particular, which combine a large search space with a relatively high probability that a change will result in a very ineffective skill by making the robot lose stability and fall.

6.3 GetUpFront

6.3.1 Skill Description

The GetUpFront skill is a Slot Behavior used to get the robot up on his feet after it fell on its front. It is the equivalent of GetUpBack except it is used when the robot is fallen on



(a) Hill Climbing

(b) Simulated Annealing

(c) Tabu Search

(d) Genetic Algortihm

Figure 6.3: Optimization of the GetUpBack skill: Score vs Iterations

its front, rather than on its back. The execution of the GetUpFront skill is shown in figure 6.4.

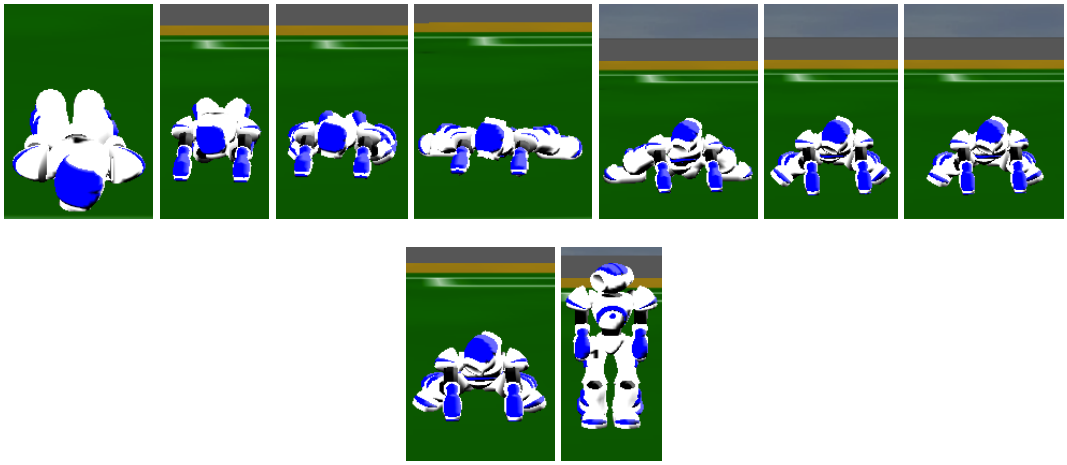


Figure 6.4: Sequence of images showing the execution of the GetUpFront skill. Each image corresponds to the final state of a slot.

Slot	Joint	Original	Optimized
Slot 1	delta	0.90	0.53274
Slot 2	delta	0.20	0.00624
	arm1	90	91.8835
Slot 3	leg5	45	48.5158
	arm1	80	79.3633
Slot 4	delta	0.100	*-0.01867*
	leg 1	-60	-56.9144
	leg 4	-120	-123.681
Slot 5	leg 4	-100	-96.5803
	leg 5	-45	-46.1314
Slot 7	delta	0.15	0.08525
Slot 8	leg 3	100	100.15
	leg 6	-45	-43.9971
Slot 9	delta	0.2	0.14617
	leg 1	-60	-58.0933
	leg 4	-120	-123.925
	arm 1	0	-4.75392
Slot 10	delta	0.30	0.05553

Table 6.3: Variables that were changed by the optimization of the GetUpBack skill using the HC algorithm.

6.3.2 Objective Function

The objective function used is the one described for the GetUpBack skill in sub section 6.2.2.

6.3.3 Optimization Options

This Slot Behavior consists of 9 different slots, with the first slot being a "reset" slot and the last one a ZeroWalk slot. A total of 39 different variables, consisting of joint angles and slot deltas, were used in the optimization of this skill after enforcing left to right side equality.

The configurations of the optimizer are the same as those chosen for GetUpBack in sub section 6.2.3 except for *nMax* which was set to the smaller value of 400, given the smaller search space for a similar skill.

For the entire configuration file used to obtain the best result see Appendix A.2.

6.3.4 Results

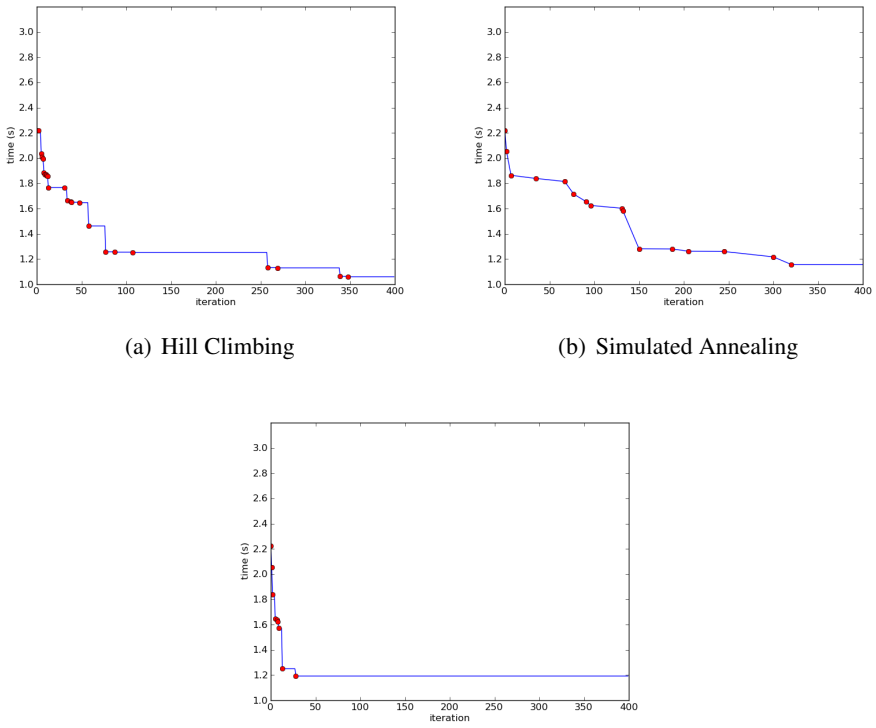
The results of the optimization process using different algorithms are shown in table 6.4.

Figures 6.5 and 6.6 show the evolution of the skill's execution time and score, respectively, over the course of each algorithm's iterations.

Algorithm	Iterations	Elapsed Time	Score
Original	NA	2.00 (s)	977.80
Hill Climbing	400	1.06 (s)	989.40
Simulated Annealing	400	1.36 (s)	988.42
Tabu Search	400	1.19 (s)	988.07
Genetic Algorithm	150	2.00 (s)	977.80

Table 6.4: Results for the optimization of GetUpFront - various algorithms compared.

The full optimized skill is shown in Appendix B.2.



(a) Hill Climbing

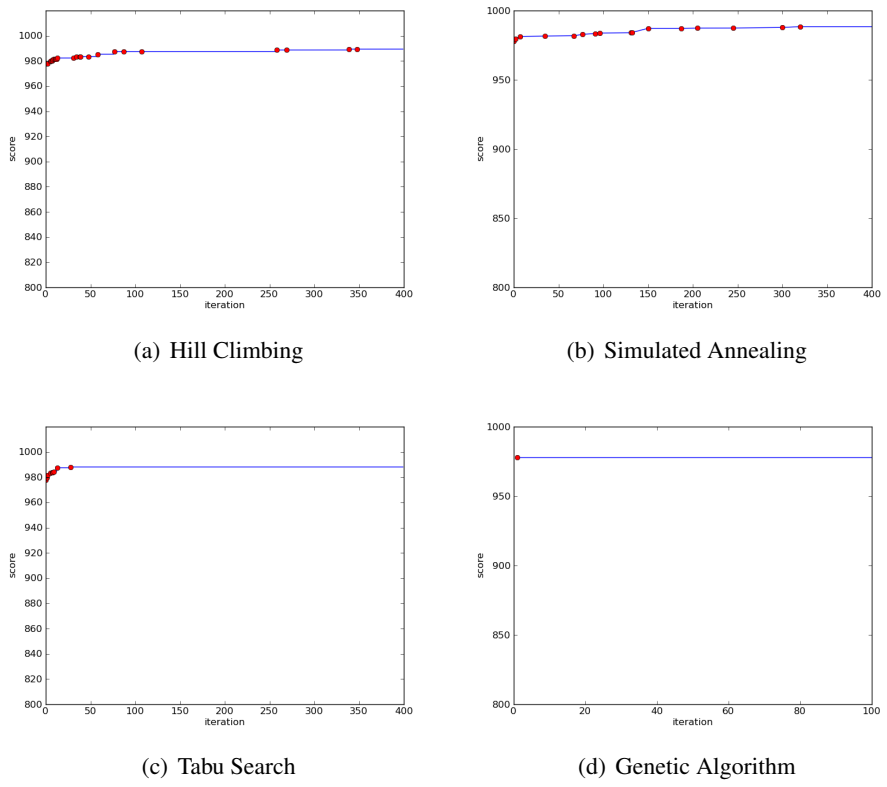
(b) Simulated Annealing

(c) Tabu Search

Figure 6.5: Optimization of the GetUpFront skill: Time vs Iterations

6.3.5 Analysis

Table 6.5 shows that 3 out of the 9 slots in the GetUpFront skill were disabled by the optimization process. This explains, in part, the 50% increase in performance (overall speed of execution) which means the skill became 1s quicker. In this skill, it's again possible to see that TS was much faster than HC reaching its solution. In this case, TS



(a) Hill Climbing

(b) Simulated Annealing

(c) Tabu Search

(d) Genetic Algorithm

Figure 6.6: Optimization of the GetUpFront skill: Score vs Iterations

reached the solution in under 50 iterations while HC took around 350. Again, GA failed to find a single better solution.

6.4 SideWalk

6.4.1 Skill Description

The SideWalk skills are a set of CPG Behaviors that are used to adjust the position of the robot, by walking left or right instead of forward or backward, before kicking the ball, they have also been used by the goalkeeper to defend the goal and could be used to avoid obstacles (i.e. other robots) before starting to run (forward walking). A total of 28 different parameters, consisting of joint angles, periods, phases and amplitudes, were used in the optimization of this skill. Rather than equalities, symmetry, equal absolute value but different sign, between left and right leg amplitudes were enforced. The skill optimized in this section is actually SideWalkLeft, shown in figure 6.7, which only moves the robot to the left however, SideWalkRight, which moves to the right, can be obtained from SideWalkLeft by simply switching the signals of certain amplitudes and performs just in the same way except it moves right instead of left.

Slot	Joint	Original	Optimized
Slot 1	leg 3	100	100.719
	leg 4	-130	-132.533
Slot 2	delta	0.10	0.07451
	leg 4	-130	-126.393
	arm 1	50	51.605
Slot 3	delta	0.200	0.12546
	leg 4	-130	-126.393
	arm 1	50	51.605
Slot 4	delta	0.200	*-0.09199*
Slot 5	delta	0.20	*0.19476*
Slot 6	delta	0.200	*-0.35815*
	leg 1	-60	-61.5235
	leg 4	-120	-119.439
	leg 5	45	46.9777
Slot 7	delta	0.3	*-0.25731*
	leg 1	-60	-57.2913
	leg 5	45	49.29
Slot 8	delta	0.50	0.15518

Table 6.5: Variables that were changed by the optimization of the GetUpFront skill using HC.

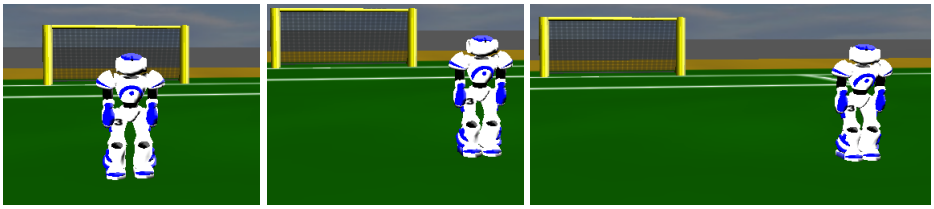


Figure 6.7: Sequence of images showing the execution of a SideWalk skill.

6.4.2 Objective Function

There are two measures of the performance of the SideWalk: speed and stability. Speed is measured by the time it takes from the start to finish of the SideWalk skill. The objective function is more complex than for the GetUp skills:

$$f = 1000 + \frac{-\Delta y}{\Delta t} * w_y - |\Delta x| * w_x - s * w_s - k * w_k \tag{6.5}$$

Speed here is defined as $\frac{distance}{duration}$, in particular, distance travelled sideways (negative y-axis). $w_y = 10$ is the weight associated with this speed. It is important that the robot moves as much as possible in a straight line, as such, forward or backward movement (x-axis) is penalized, $w_x = 5$, is the weight associated with movement in the x-axis. w_s remained with the same weight used for the GetUp skills. A condition, k, was inserted to

prevent step sizes smaller than a certain value. This could be changed to other values to purposefully allow for smaller steps but it should be noted that very small steps can result in less predictable behavior and higher instability when traversing distances of several meters. It was defined as follows:

$$k = \begin{cases} 1 & \text{if } -\Delta y < 0.1 \\ 0 & \text{otherwise} \end{cases} \tag{6.6}$$

With $w_k = 200$.

6.4.3 Optimization Options

The configurations of the optimizer are shown in table 6.6.

Algorithm	**Parameter**	**Value**
Common	Number of Experiments	6
	Threads	4
	Minimum Change for Angles	-5.0
	Maximum Change for Angles	5.0
	Minimum Change for Deltas	-0.4
	Maximum Change for Deltas	0.4;
HC	Number of Iterations	500
SA	Number of Iterations	500
	Temperature	150.0
	Cooldown Factor	0.95
	Restart	20
TS	Number of Iterations	500
	Tabu List Size	500
Genetic Algorithm	Number of Generations	100
	Population Size	24
	Number of Elite Solutions	4
	Mutation Probability	0.5
	Crossover Probability	0.2

Table 6.6: Algorithm parameters for SideWalk.

The skill consists of 14 patterns of which 10 were selected for optimization. The arms which are set static at a -90^{o} angle were not optimized. Final angles values were not optimized as these represent the ZeroWalk position, only amplitudes, phases and periods (table 3.3) with the pre-condition that there is only one period, common to all patterns. Thus, the total number of decision variables was just 11.

For the entire configuration file used to obtain the best result see Appendix A.3.

6.4.4 Results

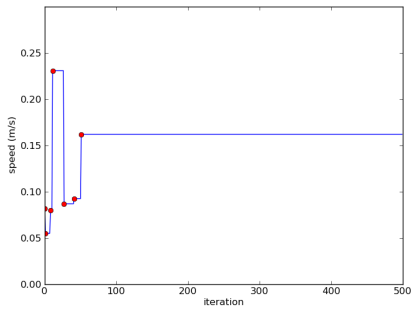
The results of the optimization process using different algorithms are shown in table 6.7. It is interesting to note that phases were not changed.

Algorithm	Iterations	Speed	$\lvert\Delta x\rvert$	Fall Percentage	Score
Original	NA	0.08 (m/s)	0.44 (m)	60 (%)	320.14
Hill Climbing	500	0.16 (m/s)	0.08 (m)	0 (%)	1001.24
Simulated Annealing	500	0.17(m/s)	0.11 (m)	0 (%)	1001.16
Tabu Search	500	0.24 (m/s)	0.12 (m)	0 (%)	1001.81
Genetic Algorithm	150	0.18 (m/s)	0.17 (m)	0 (%)	1000.93

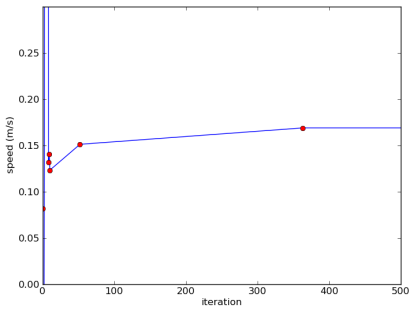
Table 6.7: Results for the optimization of SideWalk - various algorithms compared.

Figure 6.8 and 6.9 show the evolution of the skill's speed and score, respectively, over the course of each algorithm's iterations.

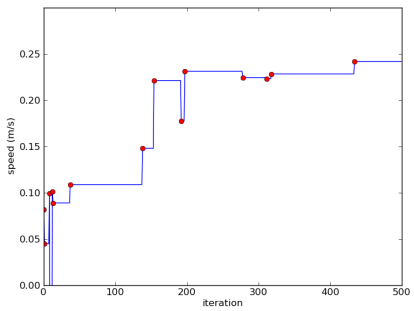
The full optimized skill is shown in Appendix B.3.



(a) Hill Climbing

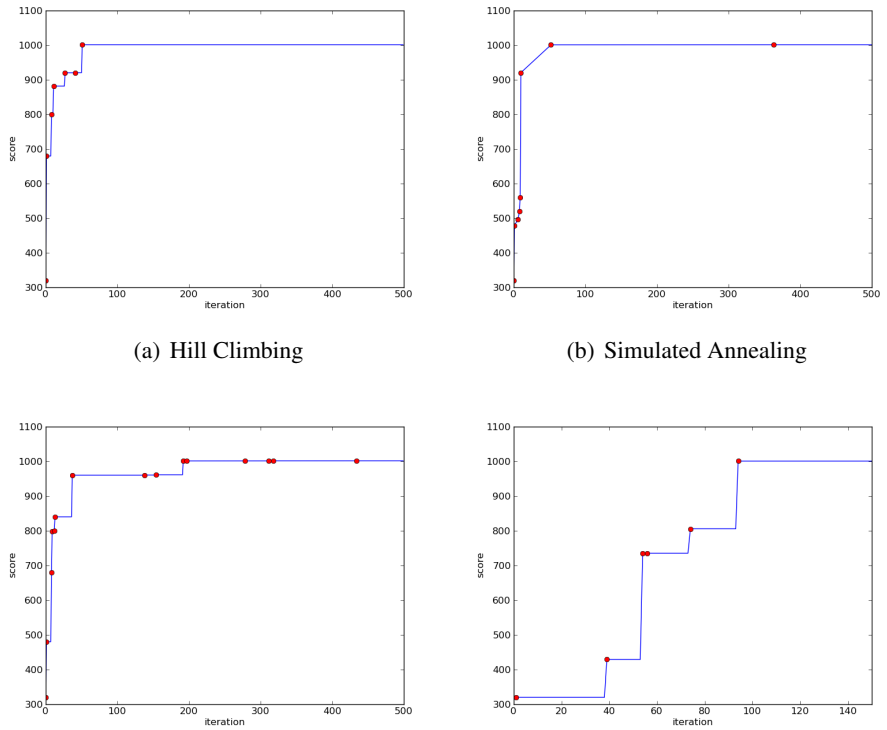


(b) Simulated Annealing



(c) Tabu Search

Figure 6.8: Optimization of the SideWalk skill: Speed vs Iterations



(a) Hill Climbing

(b) Simulated Annealing

(c) Tabu Search

(d) Genetic Algorithm

Figure 6.9: Optimization of the SideWalk skill: Score vs Iterations

Joint	**Amplitude (original)**	**Amplitude (optimized)**
leg 2	10	18.1656
leg 3	12	12.1866
leg 4	34	28.4987
leg 5	12	16.4808
leg 6	10	11.9486
Period (original)		**Period (optimized)**
0.25		0.488572

Table 6.8: Variables that were changed by the optimization of the SideWalk skill using TS.

6.4.5 Analysis

The original SideWalk skill was not optimized for speed at all, rather it was just meant to work. The improvements made by the TS optimization represent an increase in speed of approximately 300%. GA performed much better than previously but still had the worst result of all optimization algorithms used. HC and SA had very similar results in terms of objective function score. SA's solution had a higher speed but HC's solution had a much lower $|\Delta x|$. It's important to note that the original solution performed terribly

in this experiment, with a very high percentage of falls. Because of the very high weight attributed to stability in the objective function, all optimization algorithms chose solutions that did not result in any falls. The increase in speed was still very significant and there was also a decrease in the $|\Delta x|$ which was cut in half by the TS optimization.

6.5 RotateAround

6.5.1 Skill Description

The RotateAround skills are a set of CPG Behaviors that are used to adjust the position of the robot before kicking the ball. They consist of rotating around a central point at a fixed radius, left or right. They were mentioned in section 5.1.1. It is possible to develop this skill by optimizing from the SideWalk skill, described in section 6.4 with the objective function for RotateAround. The first implementation of this skill using CPG Behaviors was created in this way. While the starting point for this optimization was a human designed skill, it was created by modifying the SideWalk. The skill being optimized is RotateAroundLeft but as was the case with SideWalk, creating RotateAroundRight from this skill is just a matter of switching the signal in certain amplitudes. Multiple RotateAround exist, at least one for each different radius, which can be integrated into a single Parametrized Behavior as described in section 5.1. This particular RotateAround was optimized for a radius of 0.4m, rotating to the left, and is referred to as RotateAroundLeft40. While the skill for rotating around a point existed, it was not designed for the 0.4m radius therefore, the original skill has a very different radius.

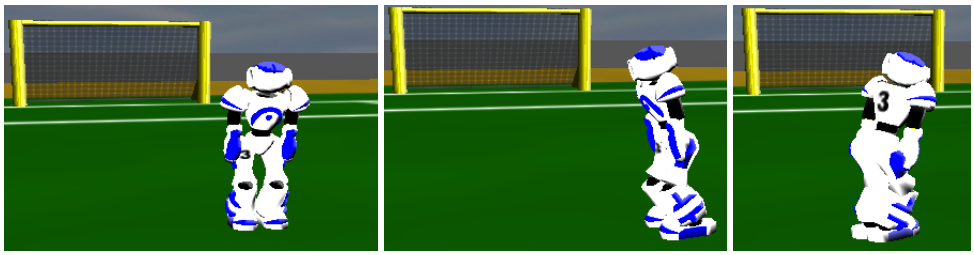


Figure 6.10: Sequence of images showing the execution of a RotateAround skill.

6.5.2 Objective Function

There are two measures of the performance of this RotateAround: final position in the x and y axis and direction the robot is facing. This results in the most complex objective function presented:

$$f = 1000 + ((1 - Error_y) * w_p + (1 - Error_x) * w_p + (1 - Error_d) * w_d)) * w_l - t * w_t - s * w_s \quad (6.7)$$

Where $Error_y$, $Error_x$, $Error_d$ are the absolute errors of the robot's x-axis position, y-axis position and direction, respectively, relative to the relative desired position, (0; 0,4), and direction, 90º. $w_x = 200$, $w_y = 200$ and $w_d = 1$ are the weights respectively associated with each of those measures and $w_l = 20$ is a final weight associated with all those measures. t is the time taken to execute the skill and has an associated weight $w_t = 10$. Finally, $w_s = 5000$.

This means that this RotateAroundLeft skill is being optimized to perform a quarter circle to the left with a radius of 0,4m, considering a relative starting position of (0,4; 0) with a relative starting direction of 0º.

6.5.3 Optimization Options

The configurations of the optimizer that are common to all algorithms are the same as those chosen for SideWalk skill in sub section 6.4.3 except for $nMax$ = 700 and the size of the tabu list, $listsize$ = 1000.

The skill is similar in composition, to the SideWalk skill (sub section 6.4.3) and the decision variables were the same except symmetries were not enforced because the skill is asymmetric in nature. The total number of decision variables was 21.

For the entire configuration file used to obtain the best result see Appendix A.4.

6.5.4 Results

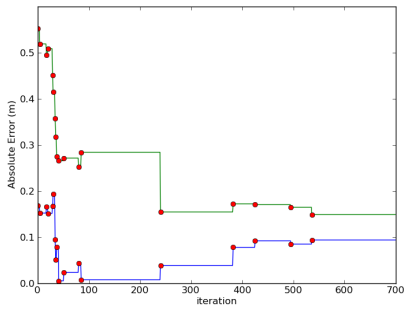
The results of the optimization process using different algorithms are shown in table 6.9.

Algorithm	Iterations	$\lvert Error_x \rvert$	$\lvert Error_y \rvert$	$\lvert Error_d \rvert$	Score
Original	NA	0.17 (m)	0.55 (m)	2.34 (º)	6079.64
Hill Climbing	500	0.09 (m)	0.15 (m)	4.58 (º)	7809.92
Simulated Annealing	500	0.01 (m)	0.37 (m)	4.52 (º)	7362.01
Tabu Search	500	0.02 (m)	0.06 (m)	4.86 (º)	8017.61
Genetic Algorithm	150	0.09 (m)	0.15 (m)	4.12 (º)	7850.41

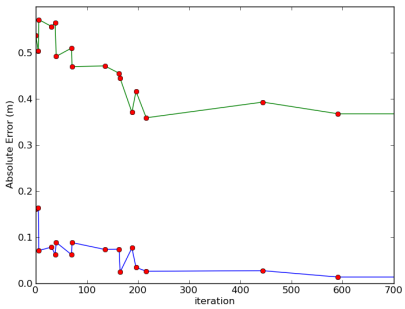
Table 6.9: Results for the optimization of RotateAround skill - various algorithms compared.

Figures 6.11 and 6.12 show the evolution of the skill's final position and score, respectively, over the course of each algorithm's iterations.

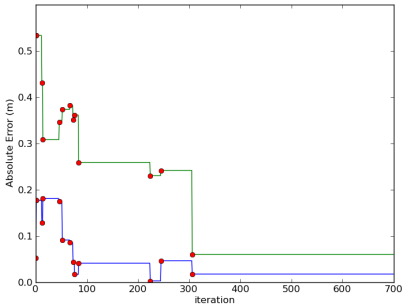
The full optimized skill is shown in Appendix B.4.



(a) Hill Climbing



(b) Simulated Annealing



(c) Tabu Search

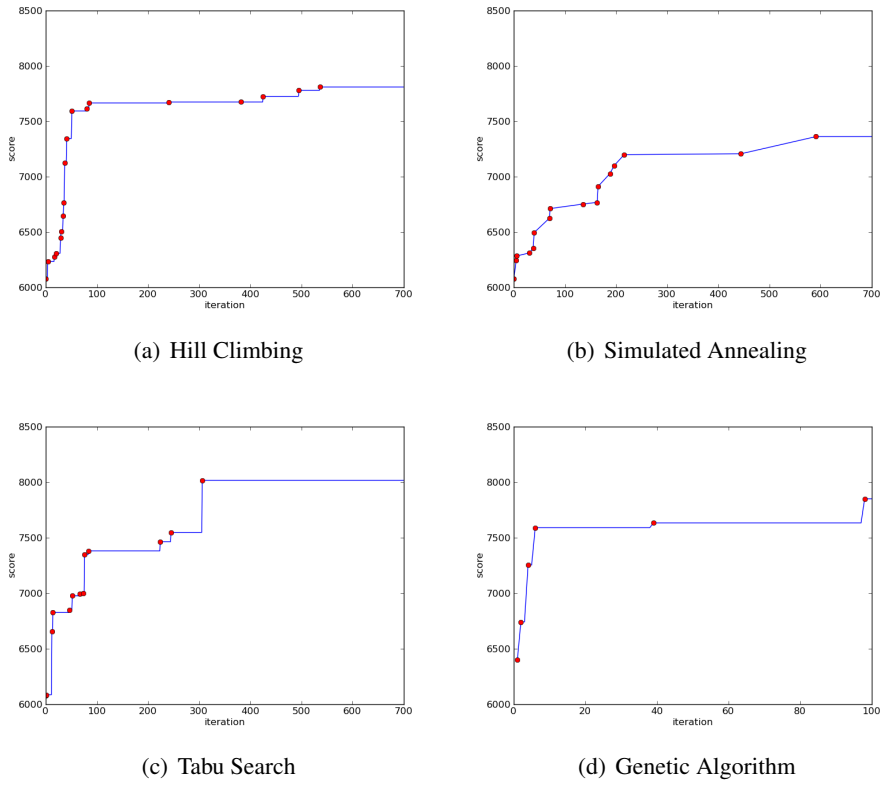
Figure 6.11: Optimization of the RotateAround skill: Position vs Iterations. X in blue, Y in green.

Joint	**Amplitude (original)**	**Amplitude (optimized)**	**Phase (original)**	**Phase (optimized)**
left leg 2	-20	-22.2925	1.570796327	2.15309
right leg 2	20	18.3508	1.570796327	2.00391
left leg 3	8	8	0	0.182294
right leg 3	-8	-11.95	0	0
left leg 4	-16	-16.9968	0	0
right leg 4	16	16	0	0.567934
left leg 5	8	14.3576	0	0
right leg 5	-8	-8	0	0
left leg 6	20	26.3867	1.570796327	1.5708
right leg 6	-20	-20	1.570796327	1.5708
Period (original)			**Period (optimized)**	
0.25			0.575333	

Table 6.10: Variables that were changed by the optimization of the RotateAround skill.

6.5.5 Analysis

The creation of a new RotateAround skill with a different radius was clearly successful. The performance of the different algorithms was, as expected, similar to that of the Side-



(a) Hill Climbing

(b) Simulated Annealing

(c) Tabu Search

(d) Genetic Algorithm

Figure 6.12: Optimization of the RotateAround skill: Score vs Iterations

Walk skill in sub section 6.4.5. TS was again the best algorithm. GA was the second best. The original skill ended at a distance of 0.721393m from the desired final position to a distance of just 0.078270m. A 920% improvement. A large improvement was to be expected since the original skill had not been designed for this specific radius.

6.6 Comparison With Other Teams

Finally, it is interesting to compare the effectiveness of the skills optimized with their equivalents in other teams participating in the RoboCup challenge. Due to the nature of the skills previously described, only the GetUp skills allow for relatively straightforward comparison with other teams. The teams selected for comparison are the 2010 champions, Apollo3D, the vice-champions, NaoTeamHumboldt and BoldHearts which took the 4th place.

The data for the performance of the other teams skill was obtained by observing the log files from the 2010 World Cup. Ten different executions of each skill were observed and an average was made.

Team	**GetUpBack**	**GetUpFront**
FC Portugal (optimized)	1.95 (s)	1.06 (s)
FC Portugal (original)	3.00 (s)	2.00 (s)
Apollo3D	2.70 (s)	2.00 (s)
NaoTeamHumboldt	3.00 (s)	3.00 (s)
BoldHearts	2.50 (s)	2.80 (s)

Table 6.11: Elapsed time for the execution of GetUp skills - various teams compared.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Humanoid robots are not currently up to par with humans in performing the most basic of tasks, such as getting up or walking sideways. This thesis offers an automated process of reducing this difference by improving existing skills with the aid of automatic optimization methods. The initial work was made to create an optimization framework for the skills of the FCPortugal3D humanoid agent which inhabits the simulated world of the RoboCup 3D simulation server. This required both modifications to the agent, the simulation server and the creation of the optimizer itself. The configuration of the optimizer allowed for easy distributed optimization using multiple agents and multiple simulations executing concurrently in multiple computers over a network. Subsequently, different optimization algorithms were implemented, namely Hill Climbing, Simulated Annealing, Tabu Search and a Genetic Algorithm. These algorithms were then used to optimize different skills: GetUpBack and GetUpFront, which use Sine Interpolation for automatic trajectory planing and SideWalk and RotateAround which use a Central Pattern Generator. The results were definitively good. The GetUpBack and GetUpFront improved their execution time by approximately 30% from around 3s to around 2s and 50% from around 2s to around 1s, respectively. SideWalk improved its speed, distanced travelled over time, by approximately 300% from around 0.08m/s to around 0.24 m/s while RotateAround improved the precision by approximately 920%. The two best algorithms, in terms of end results, were Hill Climbing which found the best solutions for the GetUp skills and Tabu Search which found the best solutions for the other skills. TS was usually faster than HC at finding solutions and even though the solutions it found for the GetUp skills were not as good, they were only marginally worse while the difference between the solutions found by HC and TS in SideWalk and RotateAround were bigger. Thus, TS stands as the best overall algorithm among those implemented. The implemented Genetic Algorithm failed to find better results for the GetUp skills while the results for SideWalk and RotateAround

were good though not as good as those obtained by TS.

Transitions between different skills, which, like Parametrized Behaviors, was preliminary work and not the main focus of this thesis, was the first feature finished and showed a slight but almost constant decrease in the time the agent took to switch between skills. This improves the overall effectiveness of the agent during a soccer game, specially when the time to switch between skills is important and makes it unnecessary to worry about providing an accurate delta for the so called "reset slot" since an arbitrarily large one can be provided by the skill's designer and the agent will always use the minimum necessary time.

7.2 Future Work

The most obvious work is to optimize more skills such as the kick and adapt the process to support FC Portugal's current walk gait generator. It would also be easy to implement more optimization algorithms such as Particle Swarm Optimization, a biologically inspired population based algorithm. Machine learning algorithms should also be experimented. Objective functions can be improved, specifically to use new measures or better ways to measure. For example, in the optimizations performed and detailed in this work, stability was determined simply by observing if the agent fell after performing the skill being optimized. By making the robot start walking after the execution of the skill and only after finishing the walk determining if it was still standing, stability could be improved. This would assume that the walk is perfect, which it currently isn't, and would require much more time per iteration but it could improve the stability of the skills optimized and perhaps enable the agent to switch between a given skill and walking.

Now that the optimizer is complete, a greater number of variations of existing skills could be created, as was the optimized RotateAround, for more flexibly dealing with the same situation with different conditions such as is the case with rotating around a point at different radius. Another case would be optimizing different kicks depending on the x and y distance to the ball. The kick is very sensitive to changes in its distance to the ball and to kick the ball into a predetermined location, the agent must be at an exact distance to the ball which usually requires it to adjust its position to the ball before performing the ball - a time consuming process which gives the opponent time to act. By optimizing the kick for different distances this precious time could be saved. The different rotate around, kicks and other skills optimized for different situations could each be integrated into its own Parametrized Behavior.

A more significant change would be to provide the optimization process with the ability to not just change and remove moves, slots and patterns but also to add new ones. This would increase the search space infinitely in theory but could be limited to a maximum number of new additions. In theory, this could ultimately be used to generate entirely new

skills given just an objective function and a lot of time. In this case, given such an extremely large search space and the relatively big difference between a randomly generated original skill and a usable skill, the problem would perhaps be better suited for population based algorithms and would require a relatively large number of computers working over a long period of time, more adequately measured in weeks or months instead of hours.

References

[BA08] Joschka Boedecker and Minoru Asada. Simspark - concepts and application in the robocup 3d soccer simulation league. In *SIMPAR-2008 Workshop on The Universe of RoboCup Simulators*, volume CD-ROM, 2008.

[BDR+08] Joschka Boedecker, Klaus Dorer, Markus Rollmann, Yuan Xu, and Feng Xue. *SimSpark User's Manual*, 1.1 edition, 2008.

[Bok81] Shahid H Bokhari. On the mapping problem. *IEEE Transactions on Computers*, C-30 (3):207–214, 1981.

[BPR10] Nuno Lau Bruno Pimentel and Luís Paulo Reis. Flexible movement planning in humanoid soccer. In *Encontro Cientifico do Robotica 2003 - Festival Nacional de Robotica*, 2010.

[BPSM+er] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml) 1.0 (fifth edition). http://www.w3.org/TR/2004/REC-xml-20040204, 2008 November.

[CK74] Vinton G. Cerf and Robert E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, Vol. 22(No. 5):637–648, May 1974.

[CWR01] Steven H. Collins, Martijn Wisse, and Andy Ruina. A 3-d passive dynamic walking robot with two legs and knees. *The International Journal of Robotics Research*, 20(7):607 – 615, July 2001.

[dRA08] Nuno Filipe dos Reis Almeida. Control agent architecture of a simulated humanoid robot. Master's thesis, Universidade de Aveiro, 2008.

[Elf39] H. Elftman. The function of the arms in walking. *Journal of Human Biology*, 11:529—535, 1939.

[FSG+] Stefania Fatone, Rebecca L. Stine, Aruna Ganju, Steven A .Gard, Stephen L. Ondra, and Regina J. Konz. A kinematic model to assess spinal motion during walking. *Journal of Biomechanics*, 31(24):E898–E906.

[Glo86] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 13(5):533–549, 1986.

[Gri85] Sten Grillner. Neurobiological bases of rhythmic motor acts in vertebrates. *Science*, 228:143–149, 1985.

[Hol75] John Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, 1975.

[JPY88] David S. Johnson, Christos H. Papadimtriou, and Mihalis Yannakakis. How easy is local search? *Journal of Computer and System Sciences*, 37(1):79–100, 1988.

[KAK+98] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, Eiichi Osawai, and Hitoshi Matsubara. RoboCup: a challenge problem for AI and robotics. In *RoboCup-97: Robot Soccer World Cup I*, pages 1–19. 1998.

[KE95] James Kennedy and Russell Eberhart. Particle swarm optimization. In *IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[LGCM08] José Lima, José Gonçalves, Paulo Costa, and António Moreira. Humanoid robot simulation with a joint trajectory optimized controller. In *Emerging Technologies and Factory Automation*, ETFA, pages 986–993. IEEE, 2008.

[LM96] Henrik Hautop Lund and Orazio Miglino. From simulated to real robots. In *International Conference on Evolutionary Computation*, pages 362–365, 1996.

[LRPS10] Nuno Lau, Luís Paulo Reis, Bruno Pimentel, and Nima Shafii. Fc portugal 3d simulation team: Team description paper. 2010.

[Mit98] Melanie Mitchell. *An Introduction to Genetic Algorithms*. The MIT Press, 1998.

[Nic04] Julien Nicolas. Artificial evolution of controllers based on non-linear oscillators for bipedal locomotion. Master's thesis, École Polytechnique Féderale de Lausanne, 2004.

[Nie03] Jens Bo Nielsen. How we walk: Central control of muscle activity during human walking. *Neuroscientist*, 9(3):195–204, June 2003.

[OORR05] Oliver Obst, Oliver Obst, Markus Rollmann, and Markus Rollmann. Spark - a generic simulator for physical multi-agent simulations. *Computer Systems Science and Engineering*, 20(5):347–356, September 2005.

[Per10] António Manuel Correia Pereira. *Intelligent Simulation of Coastal Ecosystems*. PhD thesis, Faculty of Engineering University of Porto, July 2010.

[Pic08] Hugo Picado. Development of behaviors for a simulated humanoid robot. Master's thesis, Universidade de Aveiro, 2008.

[Rao09] Singiresu S. Rao. *Engineering Optimization: Theory and Practice*. JOHN WILEY and SONS, 2009.

[RI06] Ludovic Righetti and Auke Jan Ijspeert. Programmable central pattern generators: an application to biped locomotion control. In *Proceedings IEEE International Conference on Robotics and Automation*, pages 1585–1590, Orlando, FL, USA, May 2006.

[RL01] Luís Reis and Nuno Lau. Fc portugal team description: Robocup 2000 simulation league champion. In Peter Stone, Tucker Balch, and Gerhard Kraetzschmar, editors, *RoboCup 2000: Robot Soccer World Cup IV*, volume 2019 of *Lecture Notes in Computer Science*, pages 29–40. Springer, 2001.

[RLM09] Luís Paulo Reis, Nuno Lau, and Luís Mota. Fc portugal 2009 - 2d simulation team description paper. In *Proceedings CD of Robocup 2009*, 2009.

[rob10a] January 2010.

[Rob10b] RoboCup Organizing Committee. *RoboCup Soccer Simulation League 3D Competition Rules and Setup for the 2010 competition in Singapore*, 2nd edition edition, May 2010.

[RR10] Rosaldo Rossetti and Luís Paulo Reis. Introduction to simulation and modeling. Slides Introduction to Simulation and Modeling, FEUP, 2010.

[SANS10] Nima Shafii, Siavash Aslani, Omid Nezami, and Saeed Shiry. Evolution of biped walking using truncated fourier series and particle swarm optimization. In *RoboCup 2009: Robot Soccer World Cup XIII*, volume 5949, pages 344–354. Springer, 2010.

[sim10] July 2010.

[SKAJ09] Nima Shafii, Ali Khorsandian, Abbas Abdolmaleki, and Bahram Jozi. An optimized gait generator based on fourier series towards fast and robust biped locomotion involving arms swing. In *2009 IEEE International Conference on Automation and Logistics*, pages 2018–2023, Shenyang, China, 2009.

[SRL10] Nima Shafii, Luís Paulo Reis, and Nuno Lau. Biped walking using coronal and sagittal movements based on truncated fourier series. RoboCup, July 2010.

[Ste98] W. Richard Stevens. *UNIX Network Programming: Networking APIs: Sockets and XTI*, volume Volume 1. Prentice Hall, 2nd edition, January 1998.

[TRH08] Dominic Thewlis, Jim Richards, and Sarah Jane Hobbs. The appropriateness of methods used to calculate joint kinematics. *Journal of Biomechanics*, 41(1):S320–S320, July 2008.

[UMK+10] Daniel Urieli, Patrick MacAlpine, Shivaram Kalyanakrishnan, Yinon Bentor, and Peter Stone. Optimizing interdependent skills for simulated 3d humanoid robot soccer. In *The Fifth Workshop on Humanoid Soccer Robots at Humanoids 2010*, Nashville, TN, 2010.

[war10] July 2010.

[Wei07] Thomas Weise. *Global Optimization Algorithms – Theory and Application*. Thomas Weise, july 16, 2007 edition, July 2007. Online available at http://www.it-weise.de/projects/book.pdf.

[YCPZ06] L. Yang, C. Chew, A. Poo, and T. Zielinska. Adjustable bipedal gait generation using genetic algorithm optimized fourier series formulation. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4435–4440, Beijing, China, 2006.

Appendix A

Optimizer Configuration Files

A.1 GetUpBack

```
behavior = "./movs/getup/SuperGetUpBack.xml";
type = "SlotBehavior";
objective = "getUpBack";
script ="optimization/sopt_getupback.sh";
algorithm = "hcr";
minChange = -5.0;
maxChange = 5.0;
minChangeDelta = -0.4;
maxChangeDelta = 0.4;
numExp = 14;
nMax = 500;
numThread = 6;
/* In a slot behavior,  0 is the delta. joints start at 1
i.e. Joint 5 in this file equals Joint 4 */
slots = [1,2,3,4,5,6,7,8,9,10];
slot1 = [];
sym1 = [7,8,9,10,11,12,15,16];
slot2 = [];
sym2 = [3,4,15,16];
slot3 = [];
sym3 = [7,8,9,10,11,12,15,16];
slot4 = [];
sym4 = [3,4,5,6,9,10,13,14,15,16];
slot5 = [];
sym5 = [3,4,9,10,11,12];
slot6 = [];
sym6 = [15,16];
slot7 = [];
sym7 = [3,4,9,10,11,12];
slot8 = [];
sym8 = [3,4,5,6,7,8,9,10,11,12,13,14,15,16];
slot9 = [];
```

```
sym9 = [3,4,5,6,7,8,9,10,11,12,13,14,15,16];
slot10 = [0]; /* delta in the "zeroWalk" slot */
sym10 = [];
serverRestartTime = 400;
```

A.2 GetUpFront

```
behavior = "./movs/getup/SuperGetUpFront.xml";
type = "SlotBehavior";
objective = "getUpBack";
script ="optimization/sopt_getupfront.sh";
algorithm = "hcr";
minChangeDelta = -0.4;
maxChangeDelta = 0.4;
minChange = -5.0;
maxChange = 5.0;
numExp = 9;
nMax = 500;
numThread = 6;
/* In a slot behavior,  0 is the delta. joints start at 1
i.e. Joint 5 in this file equals Joint 4 */
slots = [1, 2, 3, 4, 5, 6, 7, 8, 9];
slot1 = [0, 7, 8, 9, 10, 11, 12, 15, 16];
sym1 = [7, 8, 9, 10, 11, 12, 15, 16];
slot2 = [0, 7, 8, 9, 10, 11, 12, 15, 16];
sym2 = [7, 8, 9, 10, 11, 12, 15, 16];
slot3 = [0, 3, 4, 5, 6, 9, 10, 15, 16];
sym3 = [3, 4, 5, 6, 9, 10, 14, 15];
slot4 = [0, 3, 4, 9, 10, 11, 12, 13, 14];
sym4 = [3, 4, 9, 10, 11, 12, 13, 14];
slot5 = [0, 15, 16];
sym5 = [15, 16];
slot6 = [0, 3, 4, 9, 10, 11, 12];
sym6 = [3, 4, 9, 10, 11, 12];
slot7 = [0, 3, 4, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16];
sym7 = [3, 4, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16];
slot8 = [0, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16];
sym8 = [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16];
slot9 = [0]; /* delta in the "zeroWalk" slot */
sym9 = [];
serverRestartTime = 400;
```

A.3 SideWalk

```
behavior = "./movs/prepkick/SideLeft.xml";
type = "CPGBehavior";
```

```
objective = "sideLeft";
script = "optimization/sopt_sideleft.sh";
algorithm = "tabu";
listsize=500;
minChange = -8.0;
maxChange = 8.0;
minChangeDelta = -1.0;
maxChangeDelta = 1.0;
numExp = 3;
nMax = 500;
numThread = 4;
serverRestartTime = 500;
/* Joint 0 is the delta,
Joint 5 in this file is actually Joint 4 */
joints = [0, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14];
/* For CPGs sym
always changes the amplitude sign. */
sym = [5, 6, 7, 8, 9, 10, 11, 12, 13, 14];
joint0 = [0]; /* the delta/period */
joint5 = [0, 1, 2];
joint6 = [0, 1, 2];
joint7 = [0, 1, 2];
joint8 = [0, 1, 2];
joint9 = [0, 1, 2];
joint10 = [0, 1, 2];
joint11 = [0, 1, 2];
joint12 = [0, 1, 2];
joint13 = [0, 1, 2];
joint14 = [0, 1, 2];
```

A.4 SideWalk

```
behavior = "./movs/prepkick/RotAroundLeft.xml";
type = "CPGBehavior";
objective = "objectiveRotateLeft40";
script = "optimization/sopt_rotateleft.sh";
algorithm = "tabu";
listsize = 1000;
minChange = -5.0;
maxChange = 5.0;
minChangeDelta = -0.4;
maxChangeDelta = 0.4;
numExp = 6;
nMax = 500;
numThread = 4;
serverRestartTime = 500;
```

```
/* Joint 0 is the delta,
Joint 5 in this file is actually Joint 4 */
joints = [0, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14];
sym = [];
joint0 = [0]; /* the delta/period */
joint5 = [0, 1, 2];
joint6 = [0, 1, 2];
joint7 = [0, 1, 2];
joint8 = [0, 1, 2];
joint9 = [0, 1, 2];
joint10 = [0, 1, 2];
joint11 = [0, 1, 2];
joint12 = [0, 1, 2];
joint13 = [0, 1, 2];
joint14 = [0, 1, 2];
```

Appendix B

Optimized Skills

B.1 GetUpBack

```
<?xml version="1.0" encoding="ISO-8859-1"?>


<behavior name="Sol" type="ExpediteSlotBehavior" >

  <slot name="0" delta="0.5">
    <move id="2" angle="0" />
    <move id="3" angle="0" />
    <move id="4" angle="0" />
    <move id="5" angle="0" />
    <move id="6" angle="0" />
    <move id="7" angle="0" />
    <move id="8" angle="0" />
    <move id="9" angle="0" />
    <move id="10" angle="0" />
    <move id="13" angle="0" />
    <move id="14" angle="-90" />
    <move id="15" angle="-90" />
  </slot>
  <slot name="1" delta="0.53274">
    <move id="6" angle="100" />
    <move id="7" angle="100" />
    <move id="8" angle="0" />
    <move id="9" angle="0" />
    <move id="10" angle="30" />
    <move id="11" angle="30" />
    <move id="14" angle="-117.857" />
    <move id="15" angle="-117.857" />
  </slot>
  <slot name="2" delta="0.00624">
    <move id="2" angle="-90" />
    <move id="3" angle="-90" />
    <move id="14" angle="91.8835" />
    <move id="15" angle="91.8835" />
  </slot>
  <slot name="3" delta="0.05">
    <move id="6" angle="100" />
    <move id="7" angle="100" />
    <move id="8" angle="-120" />
    <move id="9" angle="-120" />
```

```
  <move id="10" angle="48.5158" />
  <move id="11" angle="48.5158" />
  <move id="14" angle="79.3633" />
  <move id="15" angle="79.3633" />
</slot>
<slot name="4" delta="-0.01867">
  <move id="2" angle="-56.9144" />
  <move id="3" angle="-56.9144" />
  <move id="4" angle="45" />
  <move id="5" angle="-45" />
  <move id="8" angle="-123.681" />
  <move id="9" angle="-123.681" />
  <move id="12" angle="-45" />
  <move id="13" angle="45" />
  <move id="14" angle="50" />
  <move id="15" angle="50" />
</slot>
<slot name="5" delta="0.2">
  <move id="2" angle="-90" />
  <move id="3" angle="-90" />
  <move id="8" angle="-96.5803" />
  <move id="9" angle="-96.5803" />
  <move id="10" angle="-46.1314" />
  <move id="11" angle="-46.1314" />
  <move id="12" angle="-26.4712" />
  <move id="13" angle="25" />
</slot>
<slot name="6" delta="0.1">
  <move id="14" angle="20" />
  <move id="15" angle="20" />
</slot>
<slot name="7" delta="0.08525">
  <move id="2" angle="-60" />
  <move id="3" angle="-60" />
  <move id="8" angle="-120" />
  <move id="9" angle="-120" />
  <move id="10" angle="45" />
  <move id="11" angle="45" />
</slot>
<slot name="8" delta="0.15">
  <move id="2" angle="-60" />
  <move id="3" angle="-60" />
  <move id="4" angle="45" />
  <move id="5" angle="-45" />
  <move id="6" angle="100.15" />
  <move id="7" angle="100.15" />
  <move id="8" angle="-120" />
  <move id="9" angle="-120" />
  <move id="10" angle="45" />
  <move id="11" angle="45" />
  <move id="12" angle="-43.9971" />
  <move id="13" angle="-43.9971" />
  <move id="14" angle="0" />
  <move id="15" angle="0" />
</slot>
<slot name="9" delta="0.14617">
  <move id="2" angle="-58.0933" />
  <move id="3" angle="-58.0933" />
  <move id="4" angle="45" />
  <move id="5" angle="-45" />
```

```
    <move id="6" angle="100" />
    <move id="7" angle="100" />
    <move id="8" angle="-123.925" />
    <move id="9" angle="-123.925" />
    <move id="10" angle="37.311" />
    <move id="11" angle="37.311" />
    <move id="12" angle="-45" />
    <move id="13" angle="45" />
    <move id="14" angle="-4.75392" />
    <move id="15" angle="-4.75392" />
  </slot>
  <slot name="10" delta="0.05553">
    <move id="2" angle="0" />
    <move id="3" angle="0" />
    <move id="4" angle="0" />
    <move id="5" angle="0" />
    <move id="6" angle="28.44" />
    <move id="7" angle="28.44" />
    <move id="8" angle="-46.33" />
    <move id="9" angle="-46.33" />
    <move id="10" angle="31" />
    <move id="11" angle="31" />
    <move id="12" angle="0" />
    <move id="13" angle="0" />
    <move id="14" angle="-90" />
    <move id="15" angle="-90" />
    <move id="16" angle="0" />
    <move id="17" angle="0" />
    <move id="18" angle="0" />
    <move id="19" angle="0" />
    <move id="20" angle="0" />
    <move id="21" angle="0" />
  </slot>

</behavior>
```

B.2 GetUpFront

```
<?xml version="1.0" encoding="ISO-8859-1"?>


<behavior name="Sol" type="SlotBehavior" >

  <slot name="0" delta="0.3">
    <move id="6" angle="0" />
    <move id="7" angle="0" />
    <move id="8" angle="0" />
    <move id="9" angle="0" />
    <move id="10" angle="0" />
    <move id="11" angle="0" />
    <move id="14" angle="50" />
    <move id="15" angle="50" />
    <move id="20" angle="0" />
    <move id="21" angle="0" />
  </slot>
  <slot name="1" delta="0.1">
    <move id="6" angle="100.719" />
    <move id="7" angle="100.719" />
```

```
  <move id="8" angle="-132.533" />
  <move id="9" angle="-132.533" />
  <move id="10" angle="75" />
  <move id="11" angle="75" />
  <move id="14" angle="50" />
  <move id="15" angle="50" />
</slot>
<slot name="2" delta="0.07451">
  <move id="2" angle="-60" />
  <move id="3" angle="-60" />
  <move id="4" angle="45" />
  <move id="5" angle="-45" />
  <move id="8" angle="-126.393" />
  <move id="9" angle="-126.393" />
  <move id="12" angle="-45" />
  <move id="13" angle="45" />
  <move id="14" angle="51.605" />
  <move id="15" angle="51.605" />
</slot>
<slot name="3" delta="0.12546">
  <move id="2" angle="-90" />
  <move id="3" angle="-90" />
  <move id="8" angle="-100" />
  <move id="9" angle="-100" />
  <move id="10" angle="-45" />
  <move id="11" angle="-45" />
  <move id="12" angle="-25" />
  <move id="13" angle="25" />
</slot>
<slot name="4" delta="-0.09199">
  <move id="14" angle="0" />
  <move id="15" angle="0" />
</slot>
<slot name="5" delta="0.19476">
  <move id="2" angle="-60" />
  <move id="3" angle="-60" />
  <move id="8" angle="-120" />
  <move id="9" angle="-120" />
  <move id="10" angle="45" />
  <move id="11" angle="45" />
</slot>
<slot name="6" delta="-0.35815">
  <move id="2" angle="-61.5235" />
  <move id="3" angle="-61.5235" />
  <move id="4" angle="45" />
  <move id="5" angle="-45" />
  <move id="6" angle="100" />
  <move id="7" angle="100" />
  <move id="8" angle="-119.439" />
  <move id="9" angle="-119.439" />
  <move id="10" angle="46.9777" />
  <move id="11" angle="46.9777" />
  <move id="12" angle="-45" />
  <move id="13" angle="45" />
  <move id="14" angle="0" />
  <move id="15" angle="0" />
</slot>
<slot name="7" delta="-0.25731">
  <move id="2" angle="-57.2913" />
  <move id="3" angle="-57.2913" />
```

```
    <move id="4" angle="45" />
    <move id="5" angle="-45" />
    <move id="6" angle="100" />
    <move id="7" angle="100" />
    <move id="8" angle="-130" />
    <move id="9" angle="-130" />
    <move id="10" angle="49.29" />
    <move id="11" angle="49.29" />
    <move id="12" angle="-45" />
    <move id="13" angle="45" />
    <move id="14" angle="0" />
    <move id="15" angle="0" />
  </slot>
  <slot name="8" delta="0.15518">
    <move id="2" angle="0" />
    <move id="3" angle="0" />
    <move id="4" angle="0" />
    <move id="5" angle="0" />
    <move id="6" angle="28.44" />
    <move id="7" angle="28.44" />
    <move id="8" angle="-46.33" />
    <move id="9" angle="-46.33" />
    <move id="10" angle="31" />
    <move id="11" angle="31" />
    <move id="12" angle="0" />
    <move id="13" angle="0" />
    <move id="14" angle="-90" />
    <move id="15" angle="-90" />
    <move id="16" angle="0" />
    <move id="17" angle="0" />
    <move id="18" angle="0" />
    <move id="19" angle="0" />
    <move id="20" angle="0" />
    <move id="21" angle="0" />
  </slot>

</behavior>
```

B.3 SideWalk

```
<?xml version="1.0" encoding="ISO-8859-1"?>


<behavior name="Sol" type="CPGBehavior" >

  <patterns>
    2: -15 0.423626 1.5708 -10;
    3: -15 0.423626 1.5708 -10;
    4: -14.1521 0.423626 1.5708 0;
    5: 10.8025 0.423626 2.45187 0;
    6: 9.05087 0.423626 0 28.44;
    7: -6.76724 0.423626 0 28.44;
    8: -29.7077 0.423626 0.804519 -46.33;
    9: 24 0.423626 0.94179 -46.33;
    10: 9.8671 0.423626 0.679823 31;
    11: -12 0.423626 0 31;
    12: 19.2081 0.423626 1.55665 0;
    13: -13.7167 0.423626 1.9362 0;
```

```
    14: 0 0.423626 0 -90;
    15: 0 0.423626 0 -90;
  </patterns>

  <delta>0.423626</delta>

</behavior>
```

B.4 RotateAround

```
<?xml version="1.0" encoding="ISO-8859-1"?>


<behavior name="Sol" type="CPGBehavior" >

  <patterns>
    2: -24 0.575333 1.5708 0;
    3: -24 0.575333 1.5708 0;
    4: -22.2925 0.575333 2.15309 0;
    5: 18.3508 0.575333 2.00391 0;
    6: 8 0.575333 0.182294 28.44;
    7: -11.95 0.575333 0 28.44;
    8: -16.9968 0.575333 0 -46.33;
    9: 16 0.575333 0.567934 -46.33;
    10: 14.3576 0.575333 0 31;
    11: -8 0.575333 0 31;
    12: 26.3867 0.575333 1.5708 0;
    13: -20 0.575333 1.5708 0;
    14: 0 0.575333 0 -90;
    15: 0 0.575333 0 -90;
  </patterns>

  <delta>0.575333</delta>

</behavior>
```

Appendix C

User Manual

C.1 Compiling the Optimizer

To compile the optimizer, go into the fcpagentHumanoid and type:

```
make test
```

C.2 Creating the Optimization Configuration File

The optimization configuration files are located in the *optimization/config/* directory. Parameters are defined by entering:

```
parameter = value;
```

The ";" at the end is required. String values must be enclosed in quotation marks while floating point parameters always need at least a decimal place, even if it is a zero. List parameters must be enclosed in square brackets with each value separated from the next by a comma. Comments can be used

C.2.1 Common Parameters

The following parameters must be specified for all optimizations:

- ***behavior*** **(String)** - the location of the XML specification file of the behavior to be optimized;
- ***type*** **(String)** - the type of the behavior to be optimized, the following values are currently allowed:
  - "StepBehavior";
  - "SlotBehavior";
  - "CPGBehavior";
- ***objective*** **(String)** - the name of the objective function to be used - defined in the source code;
- ***script*** **(String)** - the location of the script that starts the agents;

- ***algorithm*** **(String)** - the algorithm to be used in the optimization process, currently the following values are allowed:
  - "hc" for hill climbing;
  - "ga" for the genetic algorithm;
  - "sa" for simulated annealing;
  - "tabu" for tabu search;
- ***numExp*** **(Int)** - the number of experiments to perform for each behavior generated by the optimization algorithm;
- ***nMax*** **(Int)** - the number of iterations for the algorithms main loop;
- ***numThread*** **(Int)** - effectively the number of agents to run simultaneously in each iteration of the optimization algorithm and, therefore, also the number of generated behaviors to be tested simultaneously;
- **algorithm parameters** - the parameters of the algorithm such as the population size for genetic algorithms or the size of the tabu list for a tabu search;
- ***minChange*** **(Float)** - minimum value to change an optimization variable by adding to it to obtain a neighboring solution (this should be a negative number) - does not apply to time intervals (e.g. slot execution time);
- ***maxChange*** **(Float)** - maximum value to change an optimization variable by adding to it to obtain a neighboring solution (this should be a positive number) - does not apply to time intervals (e.g. slot execution time);
- ***minChangeDelta*** **(Float)** - minimum value to change an optimization variable that represents a time interval by adding to it to obtain a neighboring solution (this should be a negative number);
- ***maxChangeDelta*** **(Float)** - maximum value to change an optimization variable that represents a time interval by adding to it to obtain a neighboring solution (this should be a positive number).
- ***serverRestartTime*** **(Int)** - maximum amount of time the simulator can run without being restarted (in seconds).

C.2.2 Algorithm Specific Parameters

These parameters depend on the choice of algorithm and are required by their respective algorithm. Hill Climbing does not have any algorithm specific parameters.

C.2.2.1 The Genetic Algorithm

- ***nElite*** **(Int)** - the number of best solutions which will be preserved (unchanged) from the previous iteration;
- ***popSize*** **(Int)** - the total number of individuals in the population;

- ***pMutate*** **(Float)** - probability of mutation for the Single Point Mutation function;
- ***pCrossover*** **(Float)** - probability of crossover for the Uniform Crossover function;
- ***ChangeMultiplier*** **(Float)** - a positive number which multiplies the change parameters during the creation of the original population, the bigger it is, the more different will the initial population from the original solution.

C.2.2.2 Simulated Annealing

- ***temperature*** **(Float)** - initial temperature of the simulated annealing system;
- ***cool*** **(Float)** - cooldown factor, reduces the temperature at each better solution;
- ***restart*** **(Float)** - number of iterations without better solutions after which the current solution reverts back to the last best solution found.

C.2.2.3 Tabu Search

- ***listsize*** **(Int)** - size of the tabu list;

C.2.3 Behavior Type Specifc Parameters

C.2.3.1 Step Behaviors

- ***steps*** **(List)** - the list of steps, identified by number, to be optimized, if empty, all steps will be optimized;
- ***step(Number)*** **(List)** - the list of joints to be optimized for this step, if empty, all joints will be optimized, number is the number of the step that will be optimized;

C.2.3.2 Slot Behaviors

- ***slots*** **(List)** - the list of slots, identified by number, to be optimized, if empty, all slots will be optimized;
- ***slot(Number)*** **(List)** - the list of joints to be optimized for this slot, if empty, all joints will be optimized, number is the number of the slot that will be optimized;

C.2.3.3 CPG Behaviors

- ***joints*** **(List)** - the list of joints, identified by number, to be optimized, if empty, all joints will be optimized;
- ***joint(Number)*** **(List)** - the list of parameters to be optimized for this joint, where amplitude, phase, period and angle are 0, 1, 2, 3, respectively, if empty, all parameters will be optimized, number is the number of the joint that will be optimized;

C.2.4 Equalities and Symmetries

- ***sym* (List)** - the list of slots (for Slot Behaviors) or joints (for CPG behaviors) were equalities and inequalities (symmetries) are present. In the case of CPG behaviors, each pair of successive joint numbers specifies a symmetry between their amplitudes If a minus is specified before one of the joints, it will assume the amplitudes should be equal.

Slot behaviors require an additional parameter for each slot specified by the previous parameter:

- ***sym(Number)* (List)** - a list of joints for which each successive pair represents an equality between them.

C.3 Creating the Start Script

The simulation start script is similar to the normal FCP agent start script. The only difference is that the address of the optimizer and the name of the prepare function must be specified to the fcpagent via the "-o" argument. The following is an start script for the optimization of the GetUpFront skill:

```
#!/bin/bash
host=localhost
./start_simspark.sh &
sleep 5
./fcpagent -o localhost getupfront -u 1 -h $host -b -1.0 3.0 0.0 -t FCP>tttt2 2>&1 &
sleep 1.5
./fcpagent -o localhost getupfront -u 2 -h $host -b -1.0 -3.0 0.0 -t FCP>tttt2 2>&1 &
sleep 1.5
./fcpagent -o localhost getupfront -u 3 -h $host -b -4.0 3.0 0.0 -t FCP>tttt2 2>&1 &
sleep 1.5
./fcpagent -o localhost getupfront -u 4 -h $host -b -4.0 -3.0 0.0 -t FCP>tttt2 2>&1 &
sleep 1.5
./fcpagent -o localhost getupfront -u 5 -h $host -b -7.0 3.0 0.0 -t FCP>tttt2 2>&1 &
sleep 1.5
./fcpagent -o localhost getupfront -u 6 -h $host -b -7.0 -3.0 0.0 -t FCP>tttt2 2>&1 &
```