UNIVERSITY OF MIAMI

ADAPTIVE DYNAMIC WALKING AND MOTION OPTIMIZATION FOR HUMANOID ROBOTS

By

Andreas Seekircher

A PHD THESIS

Submitted to the Faculty
of the University of Miami
in partial fulfillment of the requirements for
the degree of Philosophiae Doctor

Coral Gables, Florida

July 2015

SEEKIRCHER, ANDREAS (Ph.D., Computer Science)
Adaptive Dynamic Walking and Motion Optimization for Humanoid Robots (July 2015)

Abstract of a PhD thesis at the University of Miami.

Thesis supervised by Professor Ubbo Visser.
No. of pages in text. (192)

An essential ability of a robot is to act in its environment by generating motions for locomotion or manipulation. This can be a challenging problem on a robot with high degrees of freedom. Although biped robots have shown drastic improvements with regard to motion skills over the past few years, many approaches for generating motions still need manual calibration due to variances in the hardware or inaccurate sensors and actuators. This manual fine-tuning can be very tedious and time-consuming. This thesis focuses on generating motions for humanoid robots automatically without manual calibration. The first approach discussed uses parameter optimization (e.g. CMA-ES, PSO) to directly optimize the joint angle trajectories for various motions and discusses the results and limits of this approach, the second part of this research describes an approach for a dynamically generated walk that optimizes the underlying physical model to improve the overall stability of the motion.

The first approach was used to create various motions used for a simulated NAO robot in the RoboCup 3D Soccer Simulation League. Directly optimizing the joint angles used during motions for specific tasks such as standing up or kicking a ball, already produces results that are far superior to hand tuned motions. This approach can yield good results in a short time by running multiple simulations in parallel. A similar approach is described to stabilize motions generated from noisy motion capture

data. The motion of a human is captured using a Microsoft Kinect and mapped to the robot's joint angles. Parameter optimization is used to find parameters for functions describing the angle trajectories of the captured motion without falling. Experiments show that this approach works also on physical NAO robots. However, even when using results from the simulation as a seed, it is still time-consuming on physical robots and stresses the hardware.

Furthermore, some tasks require more flexible motions. Especially a walking motion needs to be generated dynamically including a component for reactive balancing to react quickly to disturbances. These requirements and the deficiencies of the offline motion optimization led to the model-based approach for generating a walk motion dynamically described in the second part of this thesis. The developed walking engine generates a walking motion by planning steps and moving the supporting leg according to a linear inverted pendulum model (LIPM), which keeps the zero moment point (ZMP) within given constraints. The robot's state is observed and used to compensate for errors caused by disturbances and inaccuracies in the model. This is a common approach, but due to variances in the hardware and the environment it often still requires several parameters to be tuned manually. The objective of the described approach is to improve an LIPM/ZMP-based walk by optimizing parameters of the model using observations of the robots behavior while walking. Improving the model produces better predictions of the robots behavior which yields a more stable walk. The step planning is adjusted according to the modified parameters without requiring manual calibration. This approach will be integrated in the RoboCanes agent and tested using the RoboCup environment.

Contents

Chapter 1

Introduction

Humanoid robots have, theoretically, the capabilities to perform tasks usually done by a human in environments meant for humans. A wheeled robot for example can not climb stairs or a robot without a human-like arm might have difficulties to open a door. However, controlling a humanoid robot is difficult, for example due to the high degree of freedom and complex physics.

A lot of research has been done in the area of controlling humanoid robots in the past years. There is significant progress, but it is still a long way to go until an humanoid robots can act autonomously and reliably in difficult environments. Overall, controlling an autonomous robot involves a wide range of areas, e.g. computer vision, sensor fusion, decision making, planning or motion control. The goal of an autonomous robot is mostly to interact appropriately with its environment and perform certain tasks.

There can be different ways for a robot to interact with an environment. A robot can interact with different types of actuators. For example, a robot can interact with

people using speakers and text-to-speech software. However, most interaction requires using the robot's joints for locomotion and/or manipulation. Therefore, motions are a crucial component to performing many tasks. Whether a task is possible to be performed by a robot often depends primarily on (or is limited by) the robot's motion capabilities.

This also applies to the RoboCup soccer environment. The robots need to be able to walk and dribble or kick the ball in order to play. The deciding factor in many games at RoboCup is still the speed and stability of the motions.

1.1 Motivation

Generating motions on a humanoid robot that operates under the contraints of physics can be a difficult and time-consuming process. Attempts to create even simple motions by manually adjusting angles or parameters are very tedious and often end in failure [e.g., 88]. It is possible to create simple motions manually, such as arm gestures. However, motions that have to use the correct angles to keep the robot balanced are more difficult to create, since small changes in a few values can have a big effect on the dynamics of the complete robot.

Humanoid robots can have a high degree of freedom. The NAO robot has a total of 25 joints. These joints are controlled with 100 Hz, which means it is possible to set different target angles for all joints 100 times per second. Thus, there is a stream of 2,500 values per second controlling all the joints of the NAO. The large control signal space, complex physics and limited computational resources make it difficult

```
0    0  31  -60  29   0   0    0  31  -60   29   0   -90  10  -90  4   -90  -10   90  4  500
0  -10  20  -41  21  15   0  -10  20  -41   21  15   -90  10  -90  4   -90  -10   90  4  300
0  -10   8  -21  13  15   0  -10   8  -21   13  15   -90  10  -90  4   -90  -10   90  4  200
0  -10   8  -21  13  15   0  -18  20  -43   23  20   -90  10  -90  4   -90  -10   90  4  200
0  -10   8  -21  13  15   0  -18 -10  -65   40  20  -120  10  -90  4   -60  -10   90  4  200
0  -10   8  -21  13  15   0  -18 -10  -95   40  20  -120  10  -90  4   -60  -10   90  4  10
0  -10   8  -21  13  15   0  -18  30  -43   30  20   -90  10  -90  4   -90  -10   90  4  10
0  -10   8  -21  13  15   0  -18  90    0  -35  20   -60  10  -90  4  -120  -10   90  4  10
0  -10   8  -21  13  15   0  -18  30    0  -35  20   -60  10  -90  4  -120  -10   90  4  500
0  -10   8  -21  13  15   0  -18  20  -63   30  20   -90  10  -90  4   -90  -10   90  4  500
0  -10   8  -21  13  15   0  -10   8  -21   13  15   -90  10  -90  4   -90  -10   90  4  700
0    0   8  -21  13   0   0    0   8  -21   13   0   -90  10  -90  4   -90  -10   90  4  500
0    0  31  -60  29   0   0    0  31  -60   29   0   -90  10  -90  4   -90  -10   90  4  300
```

Figure 1.1: Manually created keyframes for a kick motion. Each line is one keyframe consisting of 20 joint angles and one time interval.

to create whole body motions. Furthermore, due to variance in the hardware and external disturbances, the robot never behaves exactly as it should according to the given robot model. Often simplified physical models are used to reduce the amount of computation, but they might add additional errors depending on the used model and parameters.

A common approach for generating robot motions is to create a smooth motion by interpolating between manually created keyframes. This approach is easy to implement, but creating the motions is still tedious and involves editing a lot of numbers which are usually not intuitive to edit.

Figure 1.1 shows the keyframes for a kick motion that was created manually for a simulated robot. This is an example for a motion that took a lot of effort to be created. This kick motion works, but the motion is several seconds long and the ball is not kicked very far.

Creating a motion this way for physical robots is even more tedious. Even when the same robot type is used, the same motion will always produce slightly different results for different robots due to hardware variances. One robot might fall to the left,

another robot to the right. Changing the angles slightly for each robot is a possible way to make the motion work, but maybe only until the robot fell a few times and the joints behave slightly different. Another approach is to compensate hardware variance by carefully calibrating each robot and fine-tuning several parameters, which is again time-consuming.

A better approach on physical robots is generating a motion dynamically such that it is able to balance and react to external disturbances. A kick could be generated by moving the foot along a trajectory, while setting the other joints in a way that the robot does not fall. Approaches like this can produce stable results. However, in practice these approaches still involve many parameters. Besides the basic parameters to generate the motion (coordinates for the kicking foot), there are also parameters to deal with errors in the model and variances in the hardware, such as offsets for the center of mass. Even with model based approaches, there is often a lot of manual fine-tuning required.

Some motions do work without adjusting the parameters, but are not as good as they could be. The fastest short-term solution is chosen very often and parameters are changed manually. However, some parameters then have to be updated and recalibrated regularly.

In summary, motions on humanoid robots always involve some fine-tuning of parameters, which can be direct joint angles, model parameters or calibration values. In practice, those values are often adjusted manually. This time consuming process should be avoided by adjusting motions automatically by using optimization or learning.

1.2 Goal

This thesis investigates approaches for generating motions for humanoid robots automatically using optimization methods. The overall objective is to generate stable and robust motions and a stable walk for a humanoid robot that do not require time-consuming manual parameter tuning. The motions should work on different robot models (simulated and physical robots). It should be possible to adjust motions automatically to deal with variances in the robots hardware, the model or the environment.

1.3 Overview

Chapter 2 describes the used environments, robots and some general concepts. Several approaches for generating motions using offline optimization are described in chapter 3. For better results on physical robots we describe a model-based approach for dynamic walking in chapter 4.

The appendix 4.4.1 provides information about the infrastructure and tools of the RoboCanes framework.

Chapter 2

Background

This chapter provides background information, describes commonly used methods and concepts and discusses related work.

Sections 2.1 and 2.2 describes the environments and robots used in this research and some basic methods for motion generation. Section 2.3 explains some commonly used models for dynamic walking with biped robots. Section 2.4 focuses on methods for learning or optimization in the context of motion control and describes the optimization methods used in the next chapters.

2.1 Robots and Environments

This section describes the used environments and robots. We used two different simulators and physical robots.

2.1.1 RoboCup

The RoboCup is an international competition and symposium to foster research in robotics and AI. The long term goal is to develop a team of autonomous robots by 2050 that can win again the human world champion in soccer. The main goal is not to play soccer. The main goal is to develop methods for controlling autonomous robots in a highly dynamic, partial observable, adversarial environment. Playing soccer serves as a standard problem that is publicly appealing and easy to understand, but at the same times provides many challenges and open problems for research.

Figure 2.1: Various RoboCup leagues.

RoboCup consists of several leagues that focus on different problems, use different robots and environments. There are several soccer leagues that use different robot types (e.g. the Standard Platform League, humanoid kid-size/teen-size/adult-size, small-size, middle-size, simulation leagues), but also leagues that focus on rescue scenarios or for example controlling service robots in a home environment.

This research uses the RoboCanes agent framework and is mainly based on work done for the RoboCanes team in the Soccer 3D Simulation League and the Standard Platform League (SPL). Since these leagues both concentrate on playing soccer with

humanoid robots, the motions used for this work include e.g. walking, kicking a ball and standing up, which are the most important motions for the used environment.

2.1.2 NAO Robot

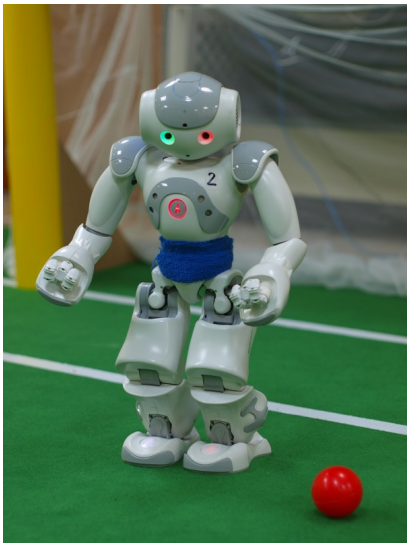


Figure 2.2: The NAO robot.

The robot model used for the experiments is the NAO ([18], figure 2.2). The NAO robot is manufactured by Aldebaran and is used as standard platform in the RoboCup SPL league. The NAO is a humanoid robot with 21 (or 25) degrees of freedom. All joints are controlled with 100 Hz. The joint angles and the current can be measured. It has an accelerometer and gyroscope in the torso and force sensors in the feet. Two cameras in the head provide images with 30 frames per second and sonars in the front of the torso can be used to detect obstacles.

The experiments we conducted on physical robots were done using five NAO V4 H21 robots and a NAO V4 H25 (25 degrees of freedom, additional wrist and hand joints). Although the H21 robots are supposed to be the same model, there are small

differences in the joints and sensor values that often make an additional calibration for each robot necessary.

2.1.3 Webots

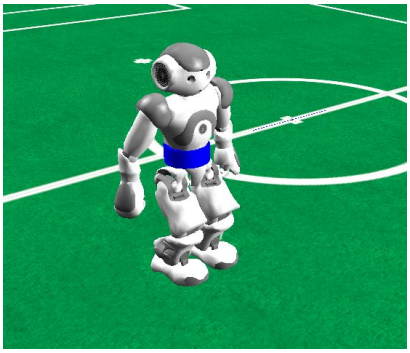


Figure 2.3: The NAO in Webots.

Webots [49] is a simulator for various robots in different environments. It also supports the Aldebaran NAO and is offered by Aldebaran as the NAO simulator. It simulates the same interfaces as on the NAO for reading sensor values and controlling the joints. As on the physical robot, the motion of the robot in Webots is controlled by setting desired joint angles at a 100 Hz frequency. We use a simulated NAO V4 H21 in Webots. The dimensions and joints of the simulated robot are the same as the physical robot, but the physical behavior is slightly different. The simulation adds noise and is not completely deterministic, but it simulates a perfect NAO that did not change from falling and general wear.

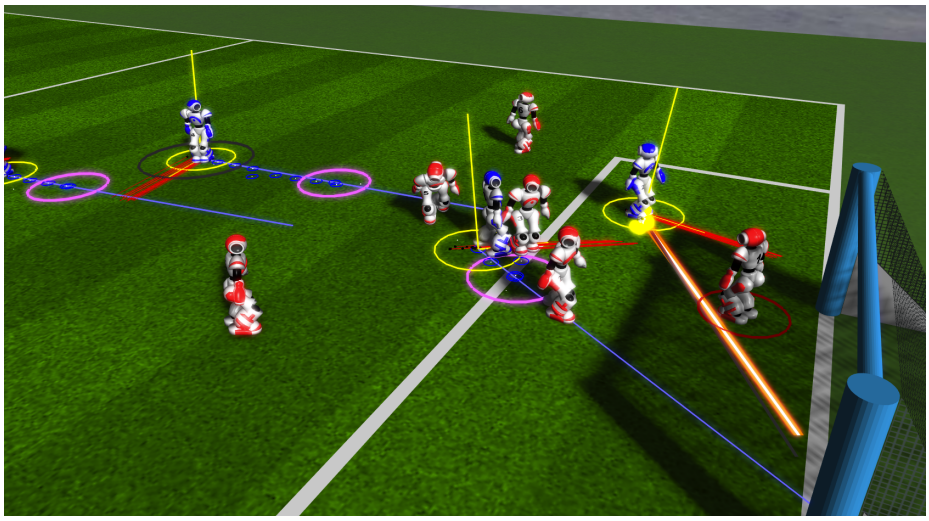


Figure 2.4: Scene from a RoboCup 3D Soccer Simulation game in SimSpark, visualized using RoboViz [81].

2.1.4 SimSpark

SimSpark (based on Spark [60]) is the simulator of the RoboCup 3D Soccer Simulation League. The simulated robot is a humanoid robot that is similar to the NAO [18]. It is equipped with 22 degrees of freedom, and receives sensor information and controls the joints with 50 Hz. The physical behavior of the robots in SimSpark is less realistic than the behavior of the NAO in Webots, but this simulator is very fast and allows us to do a large number of experiments.

2.2 Motion Control on Robots

Controlling the motion of a robot with 21 degrees of freedom with 100 Hz means setting the angles of 21 joints every 10 ms. This is a huge control space and looking at the control signal as a stream of 2100 joint angles is not very intuitive. There are several ways of representing motions by smaller and more intuitive representations.

2.2.1 Keyframe Interpolation

Some motions can simply be generated by keyframe interpolation. Each keyframe contains all joint angles at a given timestep. A smooth motion can then be generated by interpolating between all values from one frame to the next. Especially upper body motions can be created this way, since they do not have to be belanced and do not make the robot fall easily.

However, defining the angles manually is tedious. One way to simplify creating the keyframes, is to "knit" a physical robot to capture the joint angles for the different poses. But if the legs are moved and/or the movements are fast, it is difficult to manually create good keyframes and choose timesteps.

Nevertheless, this method is often used by RoboCup teams to create motions.

2.2.2 Trajectories in Cartesian Space

Instead of defining keyframes in joint space, the keyframes could contain positions of the end effectors (e.g. the feet) in cartesian coordinates. These values are much more intuitive to modify. At each timestep the foot positions only have to be transformed into joint angles using inverse kinematics [96, 83, 3].

Instead of a linear interpolation, there can then be different ways of describing the trajectory of the feet, different functions, splines or bezier curves. In the end the motion is defined by the parameters of the foot trajectories.

Similar approaches can already be used to create a walk motion [11, 98]. Feet trajectories can be used to create a parameterized walk motion. The parameters of

the motion can then be found by optimization or learning.

2.2.3 Model-based / Dynamically Generated

Motions generated by keyframe interpolation or static trajectories that are created offline can be tuned to work an a specific robot, but they are often not robust and can not react to disturbances.

A better approach is to dynamically generate motions based on a simplified physical model. For example balancing can be done by keeping the robots center of mass above the supporting polygon. Additionally, there are concepts such as the Zero Moment Point (ZMP) that can be used for constraints on the motion to keep it dynamically stable.

This can also be combined with predefined trajectories. A model-based balancing for the support leg can be combined with a static trajectory for the kicking foot.

2.3 Bipedal Walking

Bipedal walking with robots is a large research field. A variety of research is done in different areas such as balance controllers, step planning or the hardware design of robots.

Some research is very specific, e.g. Sekiguchi and Tsumaki [76] explore the effects of different foot shapes or Hanazawa et al. [25] flexible actuators. Other topics are biologically inspired designs [16] or designing robots to support heavy loads [8]. Another interesting area is passive dynamic bipedal walking to maximize the energy

efficiency [31] or streched-knee walking [61]. These are just a few examples of work done in the area of walking with robots.

We will focus more on generating motions for a robot such as the NAO robot. There are several approaches for generating a walk motion. One approach mentioned before is the optimization of a parameterized trajectories for the walk motion. Another approach is generating the oscillatory motion of the walk using a central pattern generator (CPG), which need to be trained or use parameter search [99, 42]. Hong et al. [28] also use a CPG to generate a walk with a changing height of the mass using a simulated robot. The approaches that involve training or learning to generate motions can be difficult to use on a physical robot. Random initial parameters can create motions that might damage the robot and the number of iterations has to be small. Some approaches are only evaluated using a simulated robot or at least an initial learning is done in a simulator. However, transferring results from a simulation to a physical robot can again cause problems, due to the different physical behavior of the robot in simulation and reality (the reality gap [33, 46]).

Other approaches are model-based and do not necessarily need learning or optimization for a stable walk. Model-based approached use simplified physical models, such as the inverted pendulum [35]. These models can also be used to define constraints on CoM/ZMP trajectories for a stable walk [36] or for preview control or model predictive control for walking [92, 82]. In Huang et al. [30] a ZMP-based approach is use for walking on slopes or stairs.

Additionally, there are many approaches for balance controllers, for example for lateral disturbance rejection [50], balancing through foot placement adjustment [53],

ankle-, hip- and step-strategies [1, 100] or balancing based on contact forces [65].

These are only a few examples of research done in the area of bipedal walking with robots. There is a lot of progress, however the results are often only evaluated in a simulated environment. There is a gap between the methods that work in simulations and the methods that have successfully been used on physical robots.

The simulation is a good environment for optimizing motions. However, the physical NAO can break easily and therefore a model-based approach has the benefit of directly generating a reasonably stable motion without additional learning. Model-based approaches use a simplified model to approximate the robots physical behavior. Calculating the full-body physics of the robot is computationally too expensive. Also, there are going to be errors from hardware variance and disturbances. Therefore, a simplified model is usually sufficient.

Many approaches use the inverted pendulum model or the Zero Moment point to calculate contraints for the motion to maintain stablility. Another approach is to directly use the model to generate the motion. In chapter 4 the inverted pendulum model is used for generating a walk for the NAO robot.

2.3.1 Zero Moment Point

The zero moment point occured very early in research about humanoid walking and was defined by: "The distributed floor reaction force can be replaced by a single force R that acts on the ZMP" (Vukobratovic and Stepanenko, 1972) [90] [91] [89].

As long as the robot is stable, the ZMP is equal to the center of pressure in the

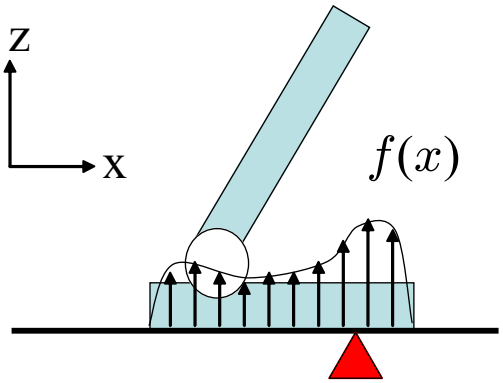


Figure 2.5: Supporting foot and the force $f(x)$ on different positions x. [37]

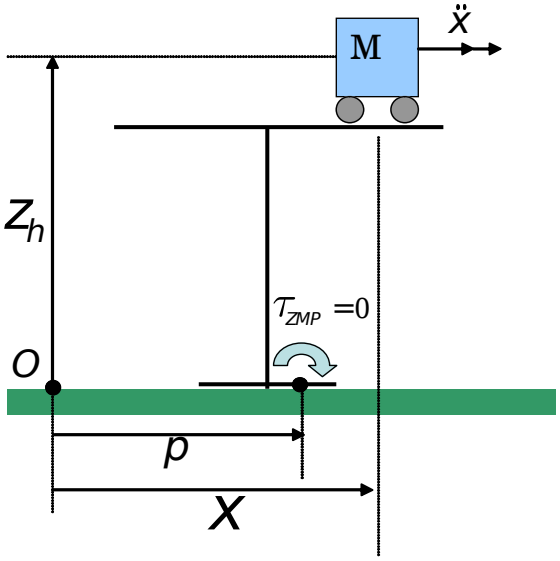


Figure 2.6: The cart-table model. [37]

support polygon. Figure 2.5 shows an example for forces on a supporting foot and the ZMP, which can be calculated by $p = \frac{\int x f(x) dx}{\int f(x) dx}$ where x is a position inside the supporting polygon and $f(x)$ the force applied at that position. The robot is stable as long as there is a ZMP inside the supporting polygon.

Figure 2.6 shows the cart-table model, which consists of a running cart on a massless table. This model can be used to calculate the position of the zero moment point for a given mass position and acceleration. The table is a rigid object. The cart

represents the mass, which applies a vertical force on the table from gravity. The acceleration of the mass applies an opposite reaction force on the table.

The zero moment point is the position in the foot where the torque equals 0, which yields equation 2.1. The torque at a position in the tables foot can be calculated as tthe torque caused by two levers, the gravity pushing on a lever with length $x - p$ and the reaction force caused by the acceleration pushing a lever with the length z_h which is the height of the mass.

$$\tau_{ZMP} = Mg(x - p) - M\ddot{x}z_h = 0 \tag{2.1}$$

$$p = x - \frac{z_h}{g}\ddot{x} \tag{2.2}$$

This can be rewritten as the ZMP equation (2.2) for calculating the ZMP position for a given x and $\ddot{x}$. Using this equation it is possible to decide whether an acceleration for a given mass position is safe and keeps the robot stable. If the ZMP would be outside the supporting polygon, the robot is not stable anymore and would start rotating around the edge closest to the ZMP.

There is a lot of related work in biped walking that uses the ZMP [101, 43, 64, 7, 79, 87]. Kajita et al. [36] generate a biped walking pattern by using a preview control of the zero-moment point (ZMP). This ZMP controller can then be used to compensate the ZMP error caused by the difference between a simple model and the precise multi-body model of the robot. Munirathinam et al. [55] describe a hybrid approach using ZMP constraints in an offline parameter optimization. A compilation of ZMP approaches for biped walking can be found in Vukobratović and Borovac [89].

2.3.2 Trajectory Generation

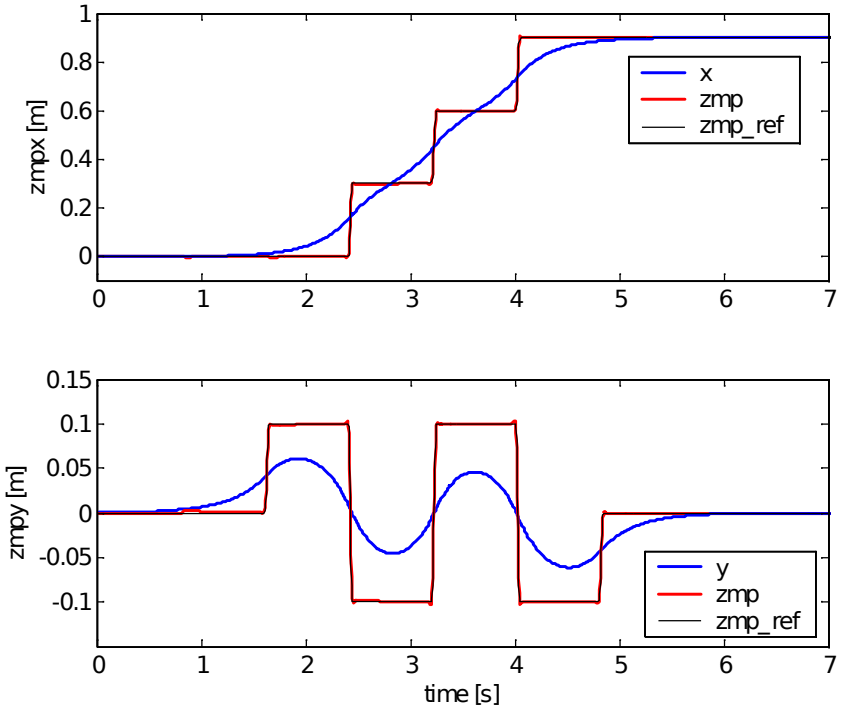


Figure 2.7: ZMP and matching center of mass trajectory [37].

Since the ZMP has to be inside the supporting foot, the steps of the robot define a desired ZMP trajectory (the reference ZMP trajectory). The robot has to move its center of mass such that the resulting ZMP matches the reference ZMP. Figure 2.7 shows an example for a ZMP trajectory and a matching mass trajectory.

2.3.3 3D Linear Inverted Pendulum Model

The carted-pendulum model in figure 2.8 is the dual of the cart-table model. It can be used to calculate the acceleration of the mass for a given position of the mass relative to the cart using equation 2.3 with the origin position p, mass position x, mass height z_h and gravity g.

A 3D inverted pendulum models the movement of the mass in a similar way.

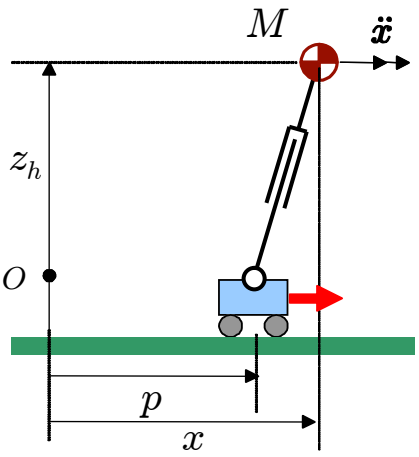


Figure 2.8: The carted-pendulum model. [37]

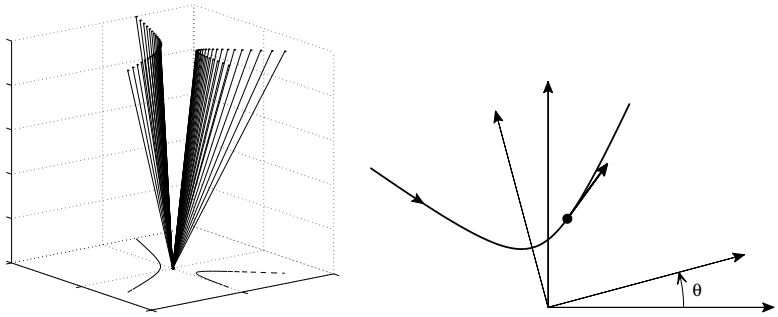


Figure 2.9: Example trajectories for the 3D inverted pendulum model and the linear inverted pendulum model projected onto the xy-plane. [35]

A given ZMP position can be used as origin of the pendulum which defines the acceleration of the mass. By assuming a constant mass height and zero input torque, the movement of the 3D inverted pendulum can be described by two independent linear functions for $\ddot{x}$ and $\ddot{y}$ [35].

$$\ddot{x} = \frac{g}{z_h}(x - p) \tag{2.3}$$

Figure 2.9 shows an example movement of the 3D linear inverted pendulum and the same trajectory shown in the xy-plane.

2.4 Optimization and Learning

Since creating a stable, complex motion manually is very difficult, there have been many approaches to use optimization or learning for generating motions.

Evolutionary strategies are used in [27] for learning a gait for a humanoid robot in a simulator and on a physical robot. Similarly, Chernova and Veloso [6] use evolutionary algorithms to learn a gait for a four legged robot.

Non-parametric methods such as dynamic bayesian networks and other probabilistic techniques have been used largely in [21, 20, 62, 67, 29].

Learning or optimization of motions often requires many repetitions. Usually the learning is first done in a simulation, before using a physical robot. Also, evaluating bad parameters can already damage a physical robot.

2.4.1 Parameter Optimization

The optimization of the motions in chapter 3 is done using black box optimization methods. The methods we used and compared are CMA-ES, xNES and PSO.

The Covariance Matrix Adaption Evolution Strategy (CMA-ES) algorithm [26, 41] is one of the most widely used algorithms for parameter optimization. The family of Natural Evolution Strategies (NES) [94, 93] algorithms are an alternative to CMA-ES in order to perform real-valued black box function optimization. For medium size dimensions, with highly correlated parameters, the exponential NES (xNES) [17, 72] empirically shows significant performance compatible with CMA-ES. Particle swarm optimizations [40] are simple and yet effective algorithms for optimizing a wide range

of functions.

There are several existing approaches for using parameter optimization to learn motions. For example, in Depinet et al. [13] CMA-ES is used for optimizing the keyframes of a kick motion. The initial values are obtaining by observing another robot.

Chapter 3

Optimization of Whole-Body Motions

The approach for generating motions automatically described in this chapter uses methods for parameter optimization to generate stable motions for humanoid robots. We generated several motions for the simulated NAO robots in SimSpark needed to play soccer and participate in the RoboCup 3D Simulation League (kick and stand-up motions) in section 3.1. Section 3.2 describes a motion learning for imitating captured human motions[1]. Section 3.3 describes the parallelization of these motions in the simulator[2]. Section 3.4 applies the optimization of the captured human motions on a physical robot and discusses the results.

[1]Section 3.2 is an extended version of "Motion Capture and Contemporary Optimization Algorithms for Robust and Stable Motions on Simulated Biped Robots" [75] which I wrote together with Justin Stoecker, Saminda Abeyruwan and Ubbo Visser.

[2]Section 3.3 is based on work from the Master Thesis of Piyali Nath, which benefited from my infrastructure and advice.

3.1 Optimization of Soccer Motions

The RoboCanes agent is used to participate in RoboCup 3D Soccer Simulation League. The simulator used in this league is SimSpark. For playing soccer, the agents have to be able to walk, kick the ball and stand up if they fall. The faster these motions are, the better is the performance of the agent.

We improved most motions of the RoboCanes agent using parameter optimization (CMA-ES). The disadvantage of this approach is the high amount of repetitions needed to find a good parameter set. However, the simulated robot can not break and SimSpark is a fast simulator that allows to run many evaluations in a short time. The results of these optimizations increased the performance of the RoboCanes team in the 3D Simulation significantly.

3.1.1 Special Action Optimization

The RoboCanes agent uses motions that are stored as "Special Actions", lists of keyframes consisting of joint angles and a timing value for the interpolation to the next keyframe.

Some attempts had been made of creating these motions manually, but the result motions are slow and unreliable. However, these motions consist of only a few keyframes and can be used as starting point for an optimization. We optimize these motions by directly using all angle and time values of the special action as parameters. The best parameter set found by the optimization can be saved as new special action and directly replace the manually created motion.

In [12] a similar optimization is used to optimize a kick motion that is defined by keyframes, but the state space is reduced significantly by fusing joint values between frames, if the angle does not change much. This reduction is neccessary for motions that consist of a larger number of keyframes.

Reducing the state space might also improve the performance of our optimization, but even without reducing the state space the optimization of the full keyframes is able to improve the motions significantly. However, without using a working motion as a seed, the optimization would get stuck in local minima and not be able to find a good motion in this huge state space (¿ 100 parameters). Nevertheless, the seed can be slow and unreliable and still allow the optimization to find better motions.

The RoboCup 3D Soccer Simulation League provides different heterogenous robot types with different body part dimensions (e.g. wider hip, longer legs) or slightly different joint behavior. To foster research in generating motions automatically, a rule was introduced enforcing teams to use several different robot types in competitions. Creating motions for each robot type can not be done manually. Instead, we can use the optimization of the special actions to increase the speed and reliability of the motions for each robot type.

Optimization of stand-up motions

In the 3D Soccer Simulation League, robots can not be damaged and there are many collisions during a game. Robots fall very often and have to be able to stand up quickly when they are lying on the front or on the back.

We manually created two special actions for standing up. These motions have been

used in several competitions, but they are not very fast. However, these motions can be used as seed in the optimization. The manually created "stand-up-front" motion consists of 10 keyframes. Each keyframe consists of 20 joint angles (head joints are fixed) and a time value for the interpolation. Therefore, the entire special action is defined by 210 values that we can use as parameters to optimize. The "stand-up-back" motion consists 8 keyframes and therefore requires 168 parameters to be optimized.

The objective of the stand up motions is to bring a lying or falling robot as fast as possible back to an upright pose from that it is able to start walking without falling. To evaluation a set of parameter, the robot is initially lying on the front or back. The motion is generated from the parameters and executed. After the execution of the motion finished, the robot immediately starts walking into a varying direction for 2 seconds. The walking phase in the end is important to find stand up motions that allow a fast transition into the walking motion.

We do not include the torso angle during the execution of the motion in the fitness function to avoid local minima. Only the torso angle after the motion when the robot starts walking is used. The smaller the sum over all measured torso angle errors during the walking phase is, the more stable was the robot after the stand up motion. If the robot falls while walking, there is a larger error. The earlier the robot falls, the higher the error. Even if the robot does not stand up at all, the walking phase is executed and the resulting large torso angle errors are used for the fitness. Additionally to minimizing the torso angle error, the duration of the motion has to be minimized to reduce the time needed for standing up.

These two criteria yield the fitness function shown in equation 3.1, where t_{action} is the time in milliseconds used for executing the motion, t_{total} is the total time for standing up and walking, $ObsTO_{t,up}$ is a vector pointing upwards for the observed torso orientation at time t and w_{time} defines the weighting between the torso angle error and the duration of the motion (in our experiments $w_{time} = 0.08$).

$$fitness_{standUp} = - \sum_{t \in [t_{action}, t_{total}]} \angle((0,0,1)^T, ObsTO_{t,up}) - t_{action} * w_{time} \qquad (3.1)$$

Since there is noise in the execution of the motions, we repeat the evaluation for each set of parameters 30 times and use the average fitness to find a reliable motion. Figure 3.1 shows the errors during the optimization of standing up from the front with robot type 0. The optimization uses CMA-ES with a population size of 30.

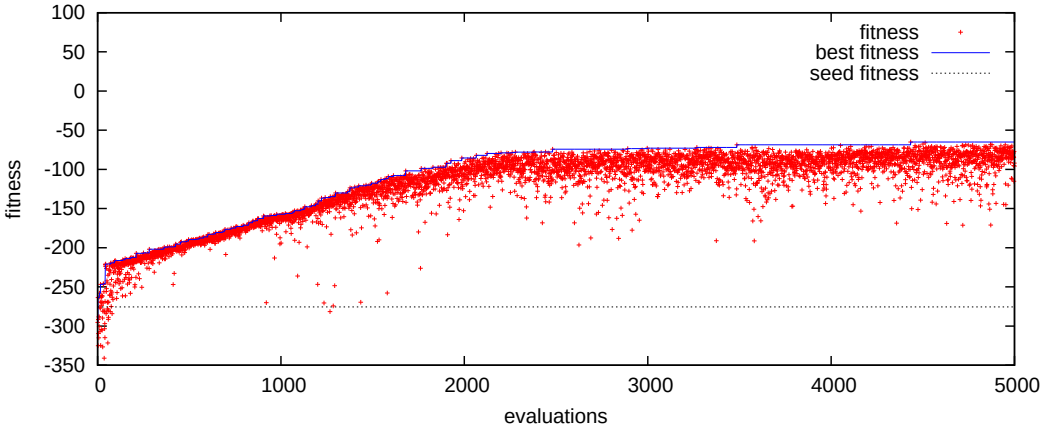


Figure 3.1: The dashed line shows the fitness for the manually created motion (stand up from front). The optimization slowly increases the fitness by adjusting the angles and timesteps in the keyframes of the motion.

Figure 3.2 shows how the duration of the motion and the error from the torso angle change during the optimization. The optimization first reduces the torso angle error and finds very stable, but still slow motions. It slowly reduces the duration,

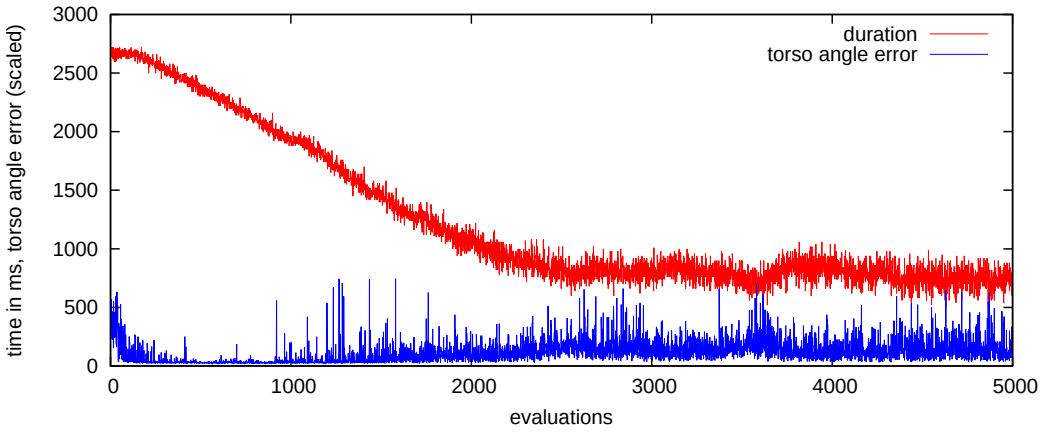


Figure 3.2: The duration of the stand up motion and the torso angle errors during the optimization for robot type 0.

which adds more variance to the torso angles error. But some parameters produce fast and stable motions and in the end only the best motion found will be used.

The optimization slowly reduces the time for the interpolation between the keyframes. When a time reaches 0, the keyframe can be completely removed from the motion. This is done in the end to keep the same number of parameters during the optimization. This shorter motion could also be used as seed with less paramters for a new optimization using the same or also other robot types.

Figure 3.3 and 3.4 show the minimum errors for both stand up motions on different robot types in SimSpark. Each of these optimizations ran for more than 40 hours on an Intel i7-5930K, but once a good motion is found it can be used in SimSpark for that robot type without further changes.

The tables 3.1 and 3.2 list results of the stand up motion optimization for the different robot types in SimSpark. These values are the average results for letting the robot 1000 times fall and stand up. As for the optimization, the robot starts to walk directly when the stand up motion finishes. After 2 seconds of walking the torso

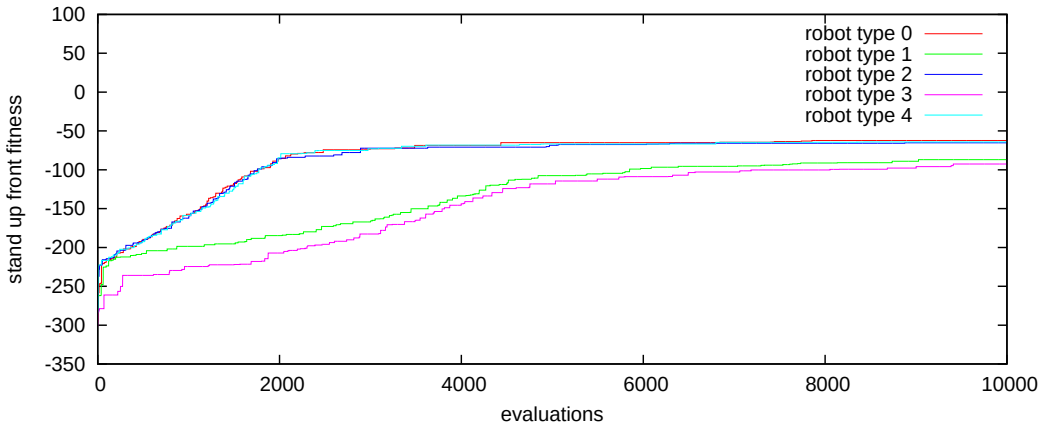


Figure 3.3: The maximum fitness during the optimization for the stand up front motion for all different homogeneous robot types in SimSpark.

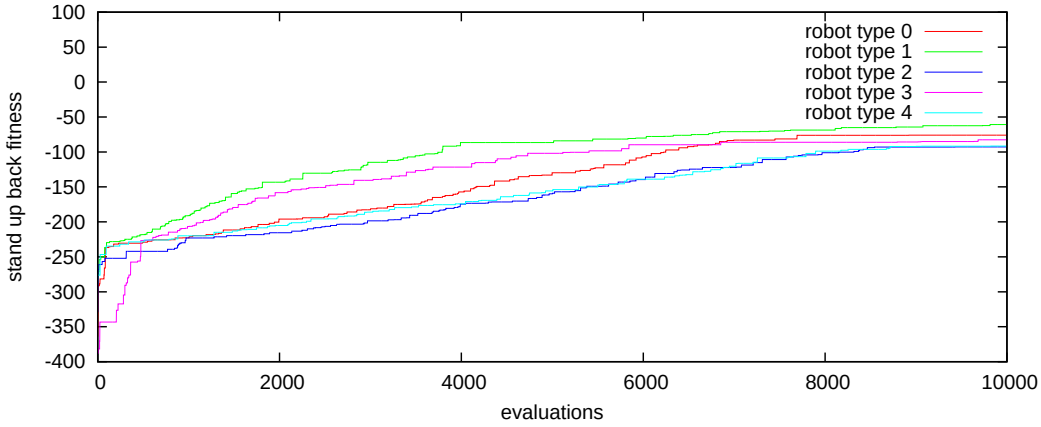


Figure 3.4: The maximum fitness during the optimization for the stand up back motion for all different homogeneous robot types in SimSpark.

angle is used to decide wether the stand up was successful (using a threshold of 30 degrees). If the robot was able to start to walk and is stable, the angle is most of the time less than 10 degress, if it falls it is about 90 degrees. The seed motion was created manually for robot type 0. Therefore, the success rate is lower for the other robot types.

The success rate is not much improved, but the time needed for the motion is reduced by 50% to 75%. Using a higher weight for the torso angle error in the fitness function would yield slower motions with a higher success rate. We chose a high weight

robot type	manual		optimized		improvement
	time	success	time	success	
0	2660	80.6%	620	86.4%	
1	2660	64.6%	960	90.0%	
2	2660	65.2%	720	91.6%	
3	2660	65.6%	1060	93.7%	
4	2660	77.0%	680	81.8%	

Table 3.1: The time needed and success rate for the manually created stand up from the front and the optimized motions for different robot types in SimSpark.

robot type	manual		optimized		improvement
	time	success	time	success	
0	2880	96.7%	860	95.7%	
1	2880	78.3%	680	84.5%	
2	2880	94.5%	1100	88.9%	
3	2880	77.7%	940	77.5%	
4	2880	79.5%	1200	90.1%	

Table 3.2: The time needed and success rate for the manually created stand up from the back and the optimized motions for different robot types in SimSpark.

on the duration, because in the simulation the robots can not break and if the first attempt to stand up fails the motion can be repeated very quickly. This configuration turned out to perform better in most games in the RoboCup 3D Simulation League.

Optimization of a kick motion

The same optimization as for the stand up motions can also be used to optimize a kick motion. The only differences are that the ball needs to be positioned before executing the motion and the fitness function has to be extended. Furthermore, we added two parameters to the optimization for the ball position relativ to the robot.

The errors used for the fitness of the stand up motions, can also be used for the kick, since the kick motion should not make the robot fall and it should be a short motion that can be executed quickly. Thus, these errors still have to be minimized.

Additionally, we have to maximize the distance the ball moved. The maximum height reached be the ball is also used to give a slightly higher fitness to kicks that make the ball fly, which reduces the friction and helps with finding long kicks faster.

$$\begin{aligned} fitness_{kick} &= kickDistance * w_{dist} + min(1.0, kickHeight) * w_h \\ &\quad - \sum_{t \in [t_{action}, t_{total}]} \angle((0,0,1)^T, ObsTO_{t,up}) - t_{action} * w_{time} \end{aligned} \tag{3.2}$$

The extended fitness function used for optimizing the kick is shown in equation 3.2, where $kickDistance$ is distance the ball traveled and $kickHeight$ is the height reached by the ball (limited to 1m).

The more values are included in the fitness function, the more difficult does it get to choose good factors to combine these values. If is duration of the motion is emphasized to much, the result might be a motion that only takes milliseconds and does not kick the ball at all. After some experiments we found the values $w_{dist} = 30$, $w_h = 120$ and $w_{time} = 0.08$ to produce fast, long kicks. Using these values, the results are not the longest kicks seen in the 3D Soccer Simulation League (e.g. [12]), but the kicks can be executed very fast.

Similar to the trade off between kick distance and the duration of the motion, we also added random noise to the initial ball position to find kicks that are more reliable. Optimizing the kick for an exact ball position might yield longer kick distances during the optimization, but the robot will not be able to position itself close enough to this position during a game, such that the result would be much shorter kicks. In our

experiments, we position the ball randomly in a 2 cm x 2 cm area, which will later allow the robot to position very quick and use the kick more often during a game.

The maximum fitness found during the kick optimizations for the different robot types is shown in Figure 3.5.

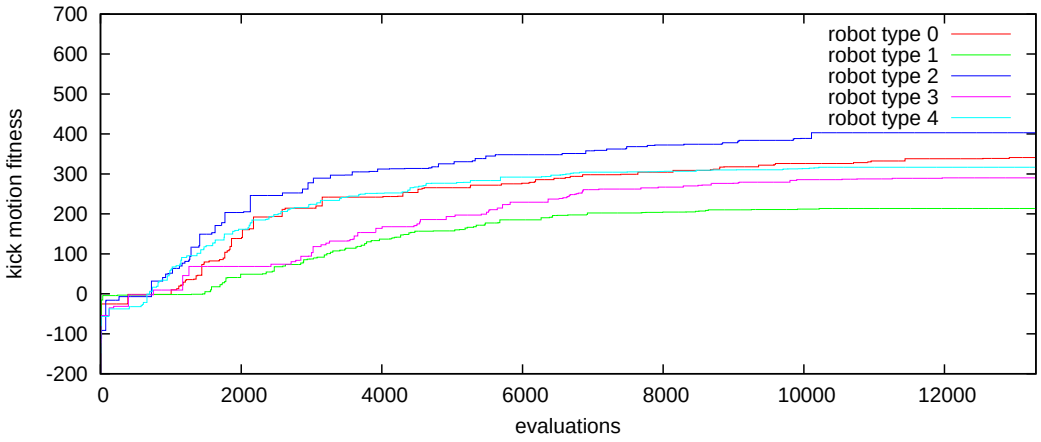


Figure 3.5: Maximum fitness values found by the optimizations of the kick special action using different robots types.

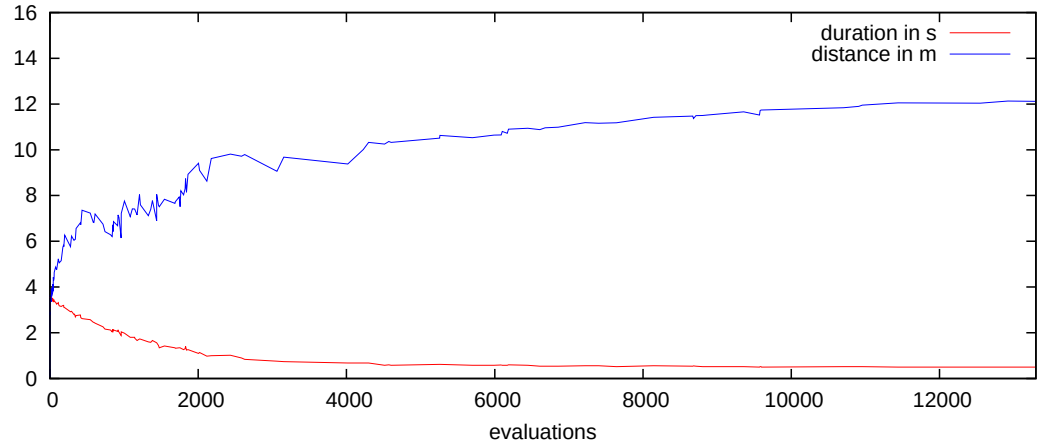


Figure 3.6: Kick distances and times of the kicks during the optimization using type 0.

Figure 3.6 shows the best kick distances and the duration of the motion during the optimization for robot type 0. The initially 3 meter long kick that takes more than 3 seconds is changed into a 12 m kick that only needs 0.7 seconds.

robot type	manual		optimized		improvement
	time	distance	time	distance	
0	3500	3.2	500	12.1	
1	3500	3.8	1300	11.2	
2	3500	0.4	460	13.7	
3	3500	3.3	1460	13.9	
4	3500	3.2	1320	14.2	

Table 3.3: The time needed and kick length for the manually created kick and the optimized kicks for different robot types in SimSpark.

The results for all robot types are summarized in table 3.3. The optimization improved the kicks for all robot types significantly. Most importantly, the time needed for the motion is much shorter. A 3 second motion can almost always be interrupted by an opponent. A kick motion that needs less than a second can be used in a lot more situations in a game.

3.1.2 Optimization of Low-level Behaviors

The kick optimization positioned the robot close to the ball and executed the motion that is optimized. However, the best kick motion does not make a difference in a game, if the robot can not position itself correctly at the ball or needs too much time for the positioning. This, we also optimized the kick positioning behavior similar to the special actions optimization.

The kick positioning behavior controls how the robot approaches the ball, how it walks around the ball if necesssary, how it positions to kick into a given direction and when to start the kick motion. We use 8 parameters: 5 for the walk target to approach the ball (distance to ball while walking around the ball, angle thresholds for walking around ball, speed factor to slow down) and 3 for the target position relative

to the ball (x and y coordinates, distance threshold to start kick). The direction of the kicks is not a parameter, but observed and saved such that the positioning can use the correct orientation to kick towards a target position during a game.

Since there are only 8 parameters, the values can be chosen manually. However, the results are usually either a too accurate which yields reliable kicks but takes too much time, or the positioning is fast but unreliable. The positioning might also perform better with individually tuned parameters for the different robot types in SimSpark. Therefore, we optimize the parameters automatically using CMA-ES for each robot type and for different kicks. The robot is positioned with a random orientation in 1.5 m distance from the ball. This is repeated with several different angles for each parameter evaluation.

The optimization uses the fitness function in equation 3.3 to increase the kick distance and reduce the time it takes to position and the kick angle standard deviation for fast and reliable kicks.

$$fitness_{standUp} = kickDistance * w_{dist} + kickAngleStdv * w_{angle} - t_{action} * w_{time} \quad (3.3)$$

Figure 3.7 and 3.8 show the results of an example run of the optimization using the optimized kick special action and robot type 0. Since the number of parameters for the kick positioning is small, it is easy to manually set values that work and let the robot kick. However, the positioning takes too much time and many kicks are short because the robot did not reach the best position. For the long kicks, the optimization using CMA-ES with a population size of 15 finds better parameters after only a few

iterations (Fig. 3.7). The average kick distance is increased from 5.7 m to 10 m, while the average time to position is decreased from an average of 8.2 to 5.4 seconds.

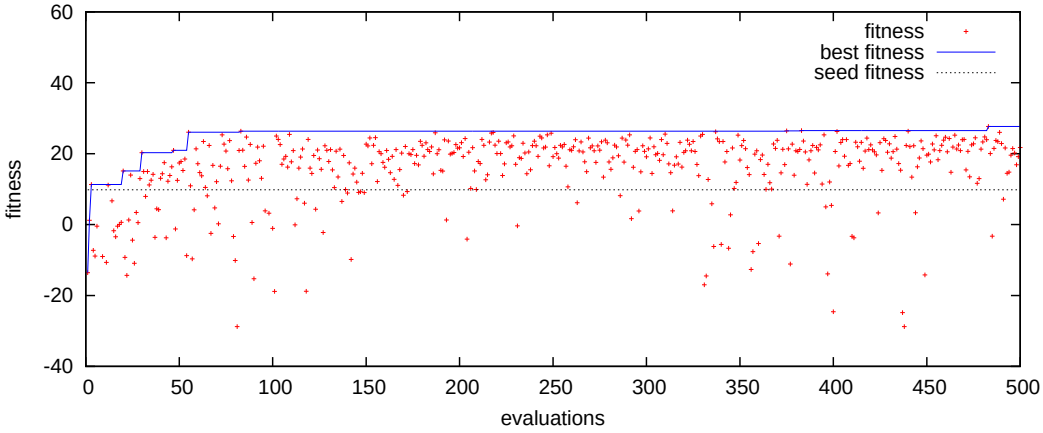


Figure 3.7: Fitness values during the kick positioning optimization for the long kick (the optimized special action) using robot type 0.

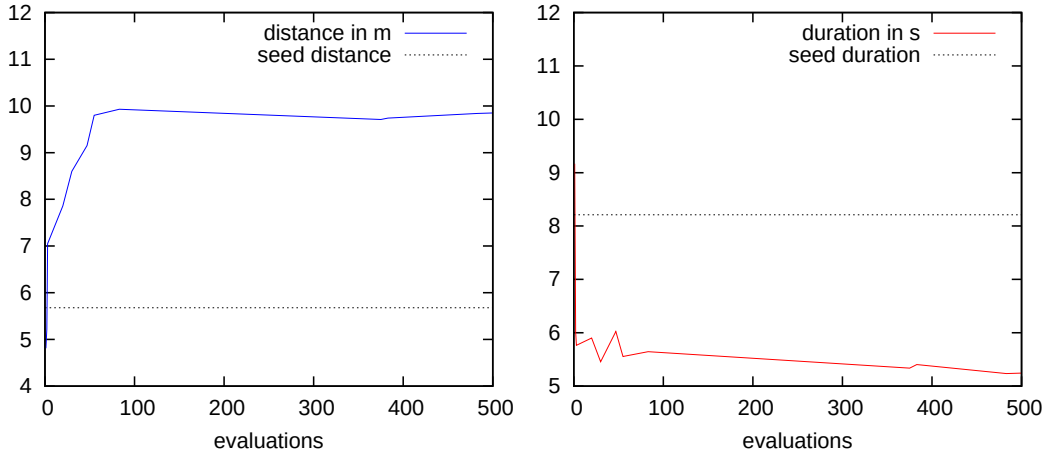


Figure 3.8: The best kick distances and positioning times during the optimization of the long kick compared to the values reached by the manually chosen parameters.

We optimized the positioning behavior for the optimized long kick special action, but also for a simple short kick that can be executed very quickly. The motion for this fast kick is generated only by setting a high torque for the hip joint in only a few frames (total time 60 ms). This trivial motion does not have to be optimized, but it depends a lot on the positioning. Manually selected parameters result in unreliable

kicks and the robot often misses the ball completely. Since the actual kick motion for this quick kick can be executed almost instantly, we apply small random velocities to the ball while the robot is positioning to optimize for situations in a game where the ball is moved by an opponent.

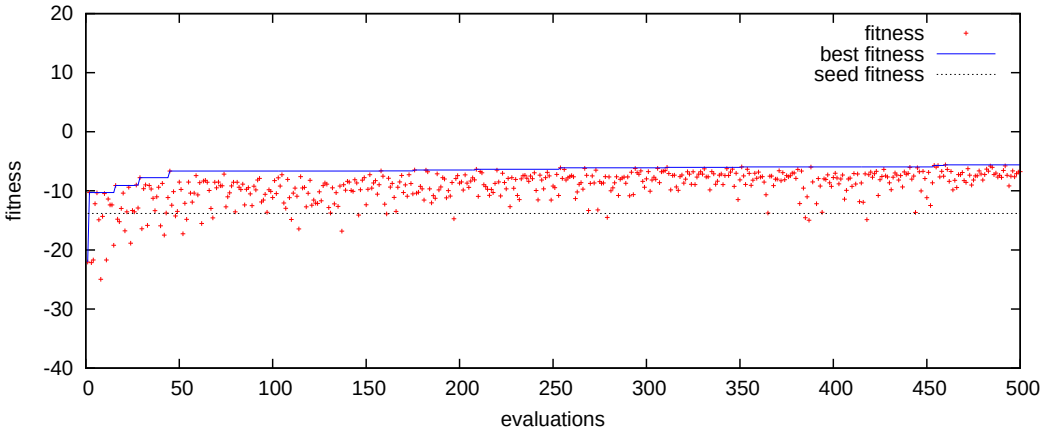


Figure 3.9: Fitness values during the kick positioning optimization for a quick forward kick using robot type 0.

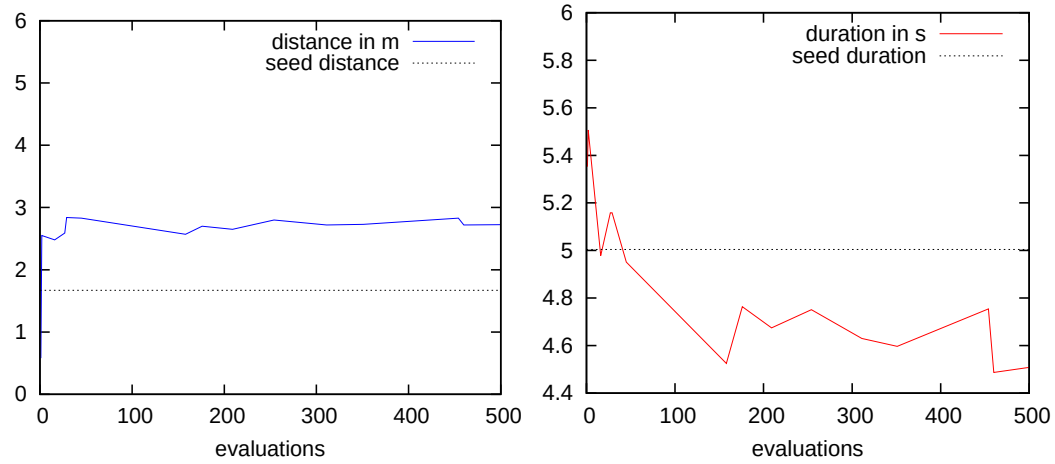


Figure 3.10: The best kick distances and positioning times during the optimization of the quick kick.

Figure 3.9 and 3.10 show the results of the kick positioning optimization for a quick forward kick. The distance is improved from 1.7 to 2.8 m. The time for the positioning is only slightly improved from 5 to 4.7 seconds, but before the kick fast

started quickly in a wrong position which caused the average distance to be only 1.7 m.

The special action and kick positioning optimizations clearly show the benefit of optimizing motions and low level-behaviors for a simulated robot. The optimization of the positioning makes it possible to use the very fast kicks in almost any situation during a game. The manually created kick special action has never been used in games. It is better to not kick at all, if the kick is to short and takes too much time. Using the special action optimization we were able to create a kick motion for long kicks that is fast enough to be used during games. The faster and more reliable motions created using the optimization increased the team's performance in the RoboCup 3D Soccer Simulation League significantly.

3.2 Kinect Motion Capturing and Optimization

In order to avoid the time-consuming process of manually generating motions, we propose a methodology and a framework to use motion capture with inexpensive equipment to record and map human motions to be deployed on humanoid robots. The immediate problem with this approach is that the humans and the humanoid robots do not share the same motor capabilities, range of motions, dimensions, masses, and other physical attributes. In order to alleviate these constraints, we assume that the dimensions and the body part masses of the humans and the robots are approximately equivalent. With this assumption, (1) the motion processing stage maps from human motion space to a specific robot motion space; and (2) the motion

optimization state optimizes the robot motions to be deployed on humanoid robots.

3.2.1 Related Work

There are numerous systems exist that enable effective human motion tracking. Perhaps the most familiar of these systems is marker-based optical motion capture [e.g., 4, 73]. A user typically wears a suit with several reflective markers that are recorded by several overhead cameras, and the positions are triangulated. An example of motion mapping using an optical marker system with the NAO robot is demonstrated in Setapen et al. [77]. There are other motion capture systems, some are based on inertial systems either partly or fully, [e.g. 102, 45, 80] or optical methods [23, 9]. Ziegler et al. [102] propose a tracking approach that aims to provide globally aligned full body posture estimates by combining a mobile robot and an inertial motion capture system. The mobile robot uses a laser scanner to anchor pose estimates of a person that is being tracked. Könemann and Bennewitz [45] use an inertial sensor system for whole body imitation. They actively balance the center of mass of the robot over the support polygon of the robot's feet to achieve stability. Optical motion tracking is used mainly in the video/gaming industry where numerous synchronized cameras are used to reconstruct the body posture of the performer [23]. Cole et al. [9] collect human body motions from a camera, map them onto a biped robot and then use a learning-based probabilistic dynamic balance model to dynamically obtain a stable sequence of motions on the robot. One major downside to these systems is that they usually require large labs with expensive equipments, software, and time.

The most related work to our approach has been introduced by [59]. Their work describe a system that has the ability to reproduce imitated motions on a DARwIn-OP humanoid robot continuously, on-line, and in real-time using the motions captured from a Kinect sensor. Our work also involves mapping from captured motions to humanoid robots, but differs in that we do not elicit any constraints on the dynamics of the captured motions, and we use off-line parametric optimization techniques.

Our motion capture experiments were performed with the first version of Microsoft's XBox 360 Kinect sensor[3]; while not as accurate as more expensive platforms, our hypothesis is that the Kinect provides a sufficient level of detail and is easily accessible to researchers without the funds or space for a dedicated motion capture lab. Even with a system that provides low-noise tracking, a significant challenge remains in stabilizing the robot when motors are actuated; mapping of human to robot joints, particularly in the legs, will often result in instability such that the robot falls over. After acquiring motions either manually or with other methods, the mapped robot motions need further optimization in order to achieve maximum performance and/or robustness [48, 12].

Evolutionary algorithms (EA) have been applied largely for this and have been proven to deliver acceptable results. Amor et al. [2], for instance, mention the use of EA to adjust the features of a mapped motion until the result is stable. Another examples has been carried out by Theeravithayangkura et al. [84] who use EA to find the most suitable posture for each virtual plane created during body compensation in adaptive biped robot gait control. Grimes et al. [22] learn a nonparametric model of

[3] `http://www.microsoft.com/en-us/kinectforwindows/`

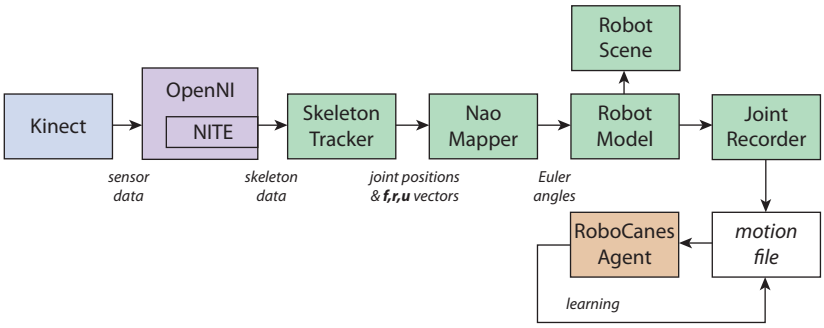


Figure 3.11: The motion capture processing pipeline. Each color designates a separate software or hardware component: the OpenNI framework is shown in purple; the motion capture mapping and recording is green; finally, the balancing is done on our RoboCup soccer simulation agent in brown.

forward dynamics from constrained exploration to infer actions and full-body imitation. Other authors do not attempt to solve the balancing problem and focus entirely or partly on imitating and mapping the upper body [47, 69].

In addition to these systems, there are popular model-based methods to plan trajectories of the motions. Kim et al. [44] have produced stable whole-body motions from motion capture by imitating a zero moment point (ZMP) trajectory of a simplified human model and dynamically adjusting the pelvis for balance. PETMAN [58] also uses motion capture data with model-based approaches for different activities.

We use the parameter optimization methods CMA-ES, PSO and xNES in the optimization stage (see section 2.4.1).

3.2.2 Human Motion Capture

The Kinect itself does not generate motion capture (MoCap) information, but it provides color and depth images (RGB-D) that can be used to track a user's body.

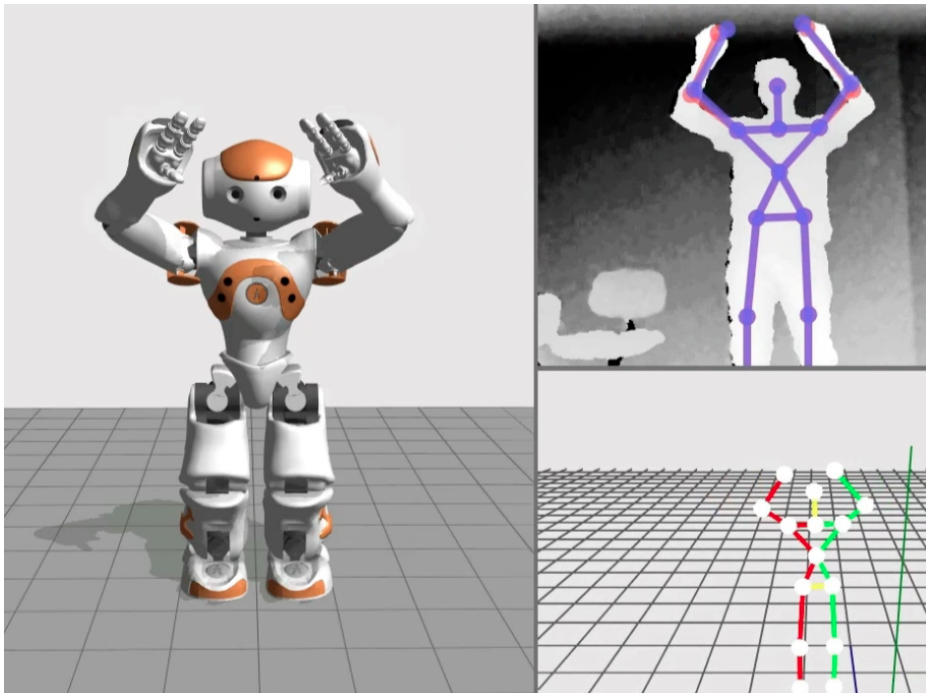
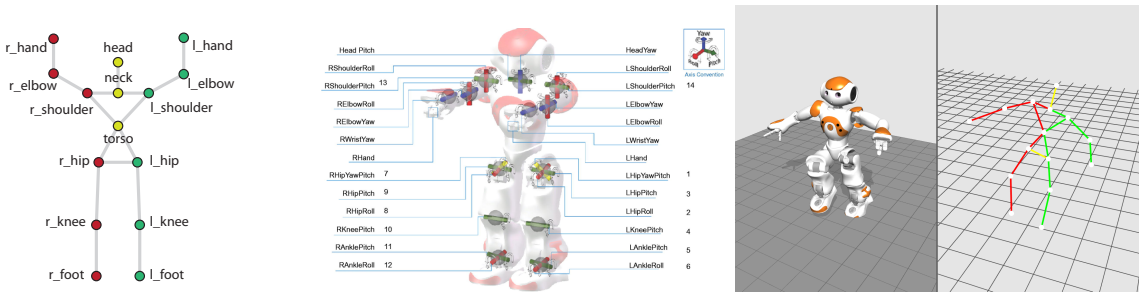


Figure 3.12: The human motion capturing using kinect and mapping to the NAO model.

Microsoft's Kinect SDK[4] and an open source alternative, OpenNI [63], both implement skeletal tracking algorithms. OpenNI is a framework that provides an interface to a variety of natural interaction (NI) devices, such as vision or audio sensors, that record motion and sound for the purpose of human-computer interaction. Rather than directly providing implementations for all imaginable sensors, both low-level and high-level features of the OpenNI API are enabled by middleware packages. We chose to use OpenNI over the Kinect SDK as it can be used with non-Windows operating systems and provides access to existing and future NI devices, such as the Xtion Pro[5]. The PrimeSense NITE [66] middleware enables skeleton tracking for the Kinect sensor.

[4] http://www.microsoft.com/en-us/kinectforwindows/develop/beta.aspx
[5] http://www.asus.com/Multimedia/Motion_Sensor/Xtion_PRO/



(a) (b) (c)

Figure 3.13: In (a), the OpenNI user skeleton model in the calibration pose. In (b)[6], the physical NAO model with all joints at 0 degrees rotation. Hinge joints in the NAO are represented as cylinders through the respective rotation axis. In (c), example pose mapped from the skeleton model to the robot model.

An overview of our motion capture processing pipeline is shown in Fig. 3.11. The Kinect is connected to a single machine by USB and provides raw sensor data to the OpenNI framework. The NITE middleware provides the skeletal tracking algorithms. Our client software uses the Java library and API to register as a listener to skeleton data updates from OpenNI. Immediately after receiving an update, the joint positions and calculated orientation vectors are packaged into a data structure that is passed on to the mapper module that modifies the state of the current robot model. For our experiments, a mapper was written specifically for the simulated NAO, though the robot model and mapper are interchangeable components and can be easily reconfigured for another robot. Finally, a real-time visualization displays the updated model (without physics) and the angles of the model are written to a motion file. This motion file serves as the input for our agents that adjust the angles to ensure a stable motion.

When OpenNI is configured to provide user tracking, data is presented as a skele-

ton model that approximates the motions of a human user. This skeleton model is defined by fifteen joints, each containing a position in sensor space; these joints are shown in Fig. 3.13a. To map the OpenNI user skeleton to the simulated NAO model, we first calculate two local coordinate systems for the user skeleton in terms of the vectors **f** (forward), **r** (right), and **u** (up); one coordinate system has the upper torso as the origin, and the second has the lower torso as the origin. For the upper body, which is used to calculate the arm angles: $\mathbf{f} = (\mathsf{l\_shoulder} - \mathsf{torso}) \times (\mathsf{r\_shoulder} - \mathsf{torso})$, $\mathbf{r} = \mathsf{r\_shoulder} - \mathsf{neck}$, and $\mathbf{u} = \mathbf{r} \times \mathbf{f}$. For the lower body orientation, which is used to calculate the leg angles: $\mathbf{f} = (\mathsf{r\_hip} - \mathsf{torso}) \times (\mathsf{l\_hip} - \mathsf{torso})$, $\mathbf{r} = \mathsf{r\_hip} - \mathsf{l\_hip}$, and $\mathbf{u} = \mathbf{r} \times \mathbf{f}$. Finally, the $\mathbf{f}, \mathbf{r}, \mathbf{u}$ vectors for both the upper and lower body are normalized to unit length.

Once the skeleton coordinate systems are established, Euler angles are computed for the joints in the NAO model. Our mapping approach uses the vectors between skeleton joint positions to calculate the NAO joint angles; this approach can be extended to any robot model consisting of revolute joints. Inverse kinematics could be used as an alternative approach to determine joint angles, although it introduces a degree of unpredictability: the trajectory of intermediate joints in a kinematic chain are not guaranteed to follow the motion of the human. Furthermore, our goal is not to position the end effectors of the robot, but instead to ensure the relative angles of body parts are correct; a 90° bend in the human's elbow should result in a 90° bend in the robot's elbow. For these reasons, a direct calculation of the joint angles is the

[6] `http://scienceblogs.com/startswithabang/2012/05/27/weekend-diversion-and-here-come-the-robot-zombies/nao-robot-dof/`

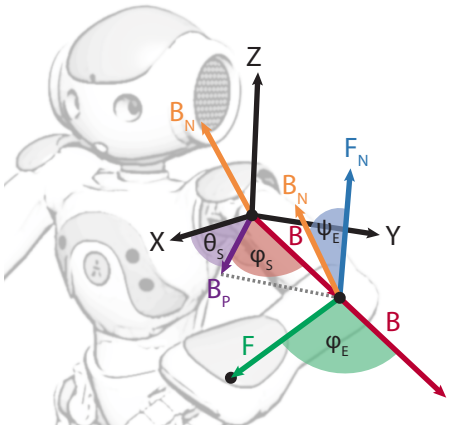


Figure 3.14: Using OpenNI skeleton joint vectors calculate to Euler angles for the NAO joints θ_s = l_shoulder_pitch, φ_s = l_shoulder_roll, ψ_e = l_elbow_yaw, and φ_e = l_arm_roll. The $\mathbf{X}, \mathbf{Y}, \mathbf{Z}$ vectors correspond to $\mathbf{f}, -\mathbf{r}$ and $\mathbf{u}$ for the upper body.

most appropriate. Figure 3.13c shows an example pose mapped from the skeleton model to the robot model. Unfortunately, the NAO's head and foot angles must be ignored, as the skeleton does not provide enough information to determine their orientations (see Fig. 3.13).

Each NAO arm has four joints that apply rotation in the following order: shoulder pitch (θ_s), shoulder roll (φ_s), elbow yaw (ψ_e), and elbow roll (φ_e). Fig. 3.14 illustrates the calculation of these angles for the left arm. Using the joints of the OpenNI skeleton, the bicep vector from l_shoulder to l_elbow, $\mathbf{B}$, is projected onto the plane spanned by $\mathbf{X}$ and $\mathbf{Z}$ to get $\mathbf{B_P} = \mathbf{X}(\mathbf{X} \cdot \mathbf{B}) + \mathbf{Z}(\mathbf{Z} \cdot \mathbf{B})$. From this, we also calculate the bicep normal $\mathbf{B_N} = \frac{\mathbf{B_P} \times \mathbf{B}}{|\mathbf{B_P} \times \mathbf{B}|}$. The shoulder pitch $\theta_s = \angle(\mathbf{X}, \mathbf{B_P}, \mathbf{Y})$, where the notation $\angle(\mathbf{U}, \mathbf{V}, \mathbf{R})$ means the angle that rotates vector $\mathbf{U}$ into $\mathbf{V}$ around $\mathbf{R}$. The shoulder roll $\varphi_s = \angle(\mathbf{B_P}, \mathbf{B}, \mathbf{B_N})$. After the shoulder joint angles are calculated, the elbow joint angles are found using the same process. The forearm vector $\mathbf{F}$ is found from l_hand to l_elbow, and the forearm normal $\mathbf{F_N} = \frac{\mathbf{F} \times \mathbf{B}}{|\mathbf{F} \times \mathbf{B}|}$. The elbow yaw

$\psi_e = \angle(\mathbf{B_N}, \mathbf{F_N}, \mathbf{B})$ and the elbow roll $\varphi_e = \angle(\mathbf{B}, \mathbf{F}, \mathbf{F_N})$.

For the legs, we observe that the hip, knee, and foot joints form a plane in space (see Fig. 3.15). The hip_yawpitch and hip_roll angles establish the orientation of this plane, and the hip_pitch and knee_pitch angles rotate the leg with in this plane. The current thigh vector **T** (knee - hip) and shin vector **S** (foot - kee) can be used to determine all angles for the leg. We initialize a lookup table to store the hip_yawpitch and hip_roll angles as well as the thigh vector used in the calculation: forward kinematics is used to iterate over possible combinations of these angles, and the normal vector of the leg plane is used as the key. To retrieve the hip_yawpitch and hip _roll angles during mapping, the current leg normal (the cross product of the thigh and tibia vectors from the skeleton) is compared with normals in the lookup table. The knee pitch $\theta_k = \mathbf{angle}(\mathbf{ST}, \mathbf{N})$. The hip pitch $\theta_H = \angle(\mathbf{T}, \mathbf{Z_H}, \mathbf{N})$ where $\mathbf{Z_H}$ is the thigh vector stored in the lookup table.

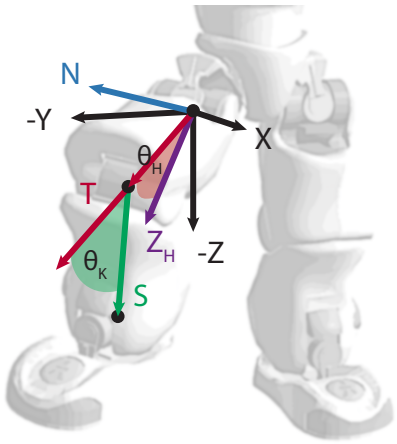


Figure 3.15: Calculating the NAO leg joint angles using the OpenNI joint vectors.

3.2.3 Optimization for Stabilizing Captured Motions

The MoCap framework provides a set of traces for each motion. These motions have variable durations, and are inherently noisy. The MoCap framework captures traces only for most of the angles, but any unknown angles (e.g., head, ankle pitch, and ankle roll) default to zero. The direct replay of the captured motions (interpolated to the robots control cycle frame rate) causes the agent to fall, since the mapping of the human motion to the robot does not consider physics or the masses and capabilities of the robot. The sequences of joint angles provided by the direct mapping have to be adjusted to obtain a stable motion. This is the main problem that we are addressing in this section. Given some motions as input, we extended our framework to (1) construct models of the motions; (2) initialize the model parameters using maximum likelihood and least squares methods; (3) optimize the prior parameters to follow the original motions; and (4) find the joint angle traces that can be replayed by the robot without falling.

Models and Initialization

A motion consists of a sequence of target angles for each joint. These sequences consist of e.g., 50 angles per second for the joint control. Instead of adjusting these angles directly, we create models for the movements of the joints. By approximating the given joint angle traces with functions, only a relatively small set of parameters has to be optimized to achieve the correct motions on the robot.

We use linear function approximation with fixed nonlinear basis functions of the

input variables to build the motion models. Each motion file contains 22 traces of target angles, which corresponding to 22 degrees of freedom available on the robot. Each trace has M sampling points. We select K traces from the motion file to build models depending on the complexity of the problem. We use linear function approximation to build motion models for each of the K traces. A trace comprises of M observations $\{x_m\}$, where $m = 1, \ldots, M$, x_1 is the trace start time while x_M is the trace end time such that $x_i < x_j$ for all $i, j \in [1, \ldots, M]$ and $i < j$, and $x_m \in \mathbb{R}_{\geq 0}$, together with the corresponding target angles $\{t_m\}$, where $t_m \in \mathbb{R}$. For each input pair, (x_m, t_m), we create a feature vector, $\phi(x_m) \in \mathbb{R}^N$, where $N \in \mathbb{N}_{>0}$, using a suitable form of basis function expansion. We use a model of the form $y(x_m, \boldsymbol{\theta}) = \sum_{j=0}^{N} \theta_j \phi_j(x_m) = \boldsymbol{\theta}^{\mathrm{T}} \phi(x_m)$, where $\phi_j(x_m)$ is a component of the the feature vector with $\phi_0(x_m) = 1$, $\boldsymbol{\theta} = (\theta_0, \ldots, \theta_N)^{\mathrm{T}}$, and $\phi(x_m) = (\phi_0, \ldots, \phi_N)^{\mathrm{T}}$. Therefore, there are $N+1$ number of parameters in the model. With the choice of suitable basis functions, we model arbitrary nonlinearities in the input traces. Basis functions take many forms, and we use polynomial basis functions of the form $\phi_j(x_m) = x_m^j$, or sigmoidal basis functions of the form $\phi_j(x_m) = \sigma(\frac{x_m - \mu_j}{s})$, where $\sigma(a)$ is the logistic sigmoidal function defined by $\sigma(a) = \frac{1}{1+\exp(-a)}$, μ_j fixes the location of the basis functions in the input space, s represents the spacial scale, and $j \in [1, \ldots, N]$. Polynomial basis functions are global functions of the input, which cause changes in one region to affect all the other regions. On the other hand, sigmoidal basis functions are local, and a small change to input only affect some of the nearby basis functions. The application of global and local basis function can have a significant influence on the optimization. We independently minimize the objective functions $\sum_{m=1}^{M}(t_m -$

$\boldsymbol{\theta}^{\mathrm{T}}\phi(x_m))^2$ using ES/PSO algorithms to find the maximum likelihood parameters for the K traces. These models use a total of $K \times (N + 1)$ parameters, which will be subjected to further optimization in subsection 3.2.3 to obtain stable motions.

Model Optimization

The evaluation of the models with the initial parameters provides approximately close enough traces to the original motions. A replay of a motion with the respective model initially fails to capture the desired outcome of the original motion. The joints are moved according to the input motion. However, depending on the specific robot model (e.g., the masses of body parts) it is necessary to change the motion slightly.

The initial optimization of the model parameters is used as a seed for the optimization of the motion for stability on the robot. This task is an optimization problem with two conflicting objectives. Following exactly the joint angles provided by the MoCap does not guarantee that the outcome is the correct motion. For instance, for a kick motion the robot could fall back and kick into the air, which might follow exactly the given joint angles. This can be avoided by including the captured torso orientation of the human for every time step in the motion capture data as a desired torso angle. This captured torso angle can be compared with the robots torso angle during the execution of the motion. The torso orientation of the simulated robot is provided by the simulator as ground truth. The distance to the desired torso orientation is one component of the fitness function (torso error). Additionally, the changes in the joint angles (joint error) have to be small to make sure that the final result is close to the captured motion, e.g., a kick motion should not be stabilized by

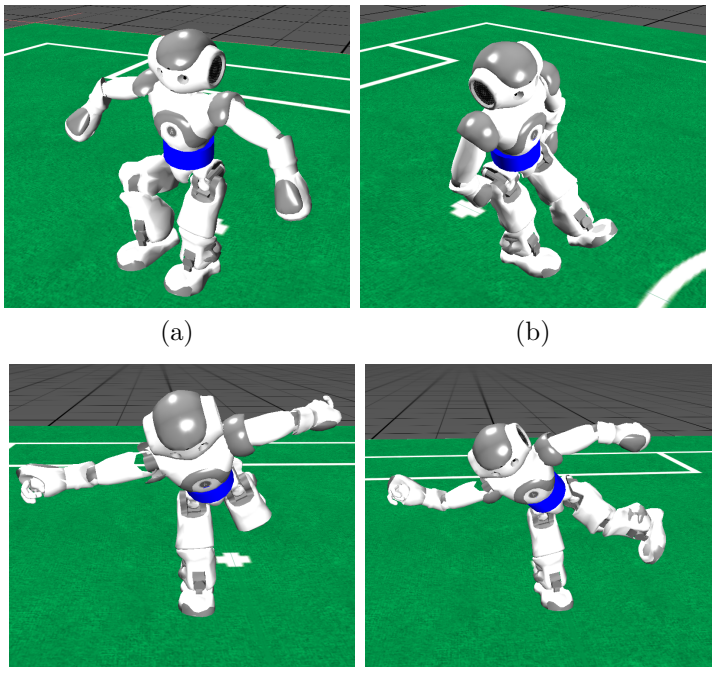
removing the actual kick from the motion. Thus, the differences between the joint angles provided by the MoCap and the measured joint angles from the robot are the second component of the fitness function. The Equation 3.4 calculates the torso angle errors while the Equation 3.5 calculates the joint angle errors:

$$\sum_{t=1}^{M} \max_{i \in \{forward,up,side\}} (\angle(MoCapTO_{t,i}, ObsTO_{t,i}))^2 \tag{3.4}$$

$$\sum_{t=1}^{M} \sum_{j=1}^{K} (MoCapAngle_{t,j} - ObsJointAngle_{t,j})^2 \tag{3.5}$$

where $\text{MoCapTO}_{t,i}$ represents the MoCap torso orientation and $\text{ObsTO}_{t,i}$ represents the observed torso orientation at time t for forward, up, and side angles, and $\text{MoCapAngle}_{t,j}$ represents the MoCap captured joint angle and $\text{ObsJointAngle}_{t,j}$ represents the observed joint angle for time t for joint j. The combination of the Equation 3.4 and the Equation 3.5 is the input to the parameter optimization algorithms.

We use ES/PSO algorithms again to optimize the model parameters until the desired motion is learned. In this phase, we optimize $K \times (N+1)$ parameters directly. We use the sum of the torso errors and the joint errors over all frames of the motion as the fitness to perform real valued black box function optimization. We have decided to optimize only the traces of the agent's legs. There are twelve such traces for each motion. Therefore, we directly optimize $12 \times (N+1)$ parameters (e.g., if we were to commit to a polynomial model with eight parameters, we optimize $12 \times (7+1) = 96$ parameters.



(a) (b)

(c) (d)

Figure 3.16: These screenshots show the motions we have used in the experiments: (a) lifting the leg, (b) a kick, (c) bending forward and balance, (d) leaning to the side. The poses in (c) and (d) look similar, but for the balance motion (c) the torso is moved only forward and not sideways as in the side balance (d).

3.2.4 Experiments using Simulated Robots

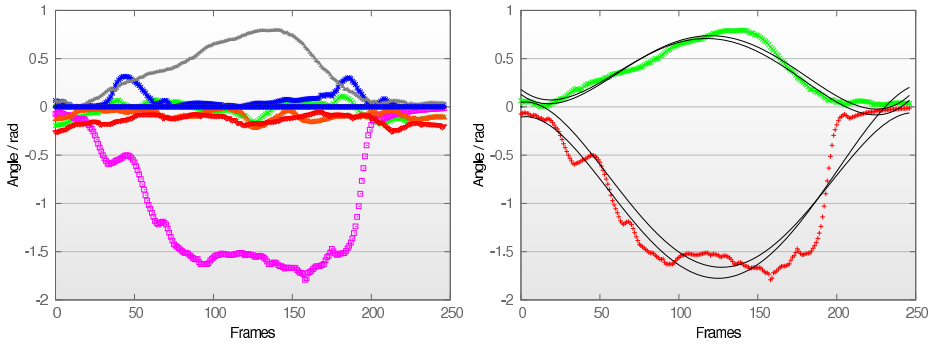
We use four different motions in the experiments (cf. Fig. 3.16) in which the robot (1) lifts the right leg for a few seconds (motion `leg`); (2) performs a simple kick motion (motion `kick`); (3) leans forward and balances on one leg while stretching the other leg back (motion `balance`); and (4) leans the torso to the side (motion `side`). The joint motions are modeled using two different function approximations, which are linear: polynomials and weighted sigmoidal basis functions. These models are initialized by minimizing the least squared error to the input angles. Using the initial parameters as a seed, the twelve leg joints are optimized by CMA-ES, xNES, and PSO using the fitness function based on the joint and the torso error explained

in subsection 3.2.3.

In [88], twelve parameters were learned with a population size of 30; since we optimize up to 96 parameters, we use a higher population size of 50 for all optimization methods. CMA-ES and xNES start with the parameters from the initialization of the models as the mean and a standard deviation of 0.06, which produces a suitable amount of exploration in the beginning of the optimization. The samples for PSO are initialized using the same mean and standard deviation. Our PSO implementation uses the parameters proposed in [5]. To calculate the fitness function; the robot is initialized to the pose in the first frame of the motion. While the motion is executed using the current parameters, all torso and joint errors are added; these errors are used as the fitness of the tested parameters.

We did not learn the transition from and to different motions in this paper. The evaluation of a parameter set starts with a short preamble phase during in which the robot moves all joints to the angles at frame zero of the motion. The robot is also moved to an initial torso position using an operator interface of the simulator. This way, the motion is always started from the same initial situation. At the end of the motion, the robot keeps the joints at the angles from the last frame for half a second. During this time, the torso and joint errors for the evaluation are still accumulated; this prevents learning of motions that are unstable in the end and would make the robot fall immediately after the motion is done. Transitions could be learned by using different angles in the first or last frame, which is planned as future work.

The first set of experiments were conducted using SimSpark. The fast simulation speed allowed us to run many experiments and obtain good average results for differ-



(a) (b)

Figure 3.17: (a) The captured leg joint angles of the balance motion. (b) Two examples for leg joint motions (l_knee_pitch and r_hip_pitch) during the balance motion with the corresponding models using the initial parameters and the adjusted models of the stabilized motion. A small difference in the joint angles is sufficient to stabilize the motion.

ent models and optimization algorithms. To prepare running the optimization on the physical Robot, we ran similar experiments in Webots, which runs slower, but has a more accurate NAO model. In the final experiments we use the physical NAO.

Experiments in Simspark

Table 3.4: The errors after learning for five hours simulated time using polynomials with five parameters as models (average over 30 runs).

Motion	Optimization	evals	min.err.	avg.err.	stddev	torso err.	joint err.	success
leg	CMA-ES	2989	0.009	0.016	0.004	0.006	0.009	83%
leg	PSO	2987	0.006	0.058	0.154	0.049	0.009	70%
leg	xNES	2990	0.013	0.017	0.003	0.007	0.010	86%
kick	CMA-ES	4926	0.018	0.027	0.005	0.011	0.016	60%
kick	PSO	4924	0.015	0.135	0.210	0.106	0.029	46%
kick	xNES	4931	0.023	0.037	0.047	0.020	0.017	60%
balance	CMA-ES	2970	0.051	0.096	0.076	0.046	0.050	60%
balance	PSO	2971	0.072	0.807	0.403	0.720	0.088	13%
balance	xNES	2971	0.047	0.089	0.153	0.050	0.039	60%
side	CMA-ES	1816	0.432	1.351	0.466	1.203	0.149	0%
side	PSO	1816	0.474	1.634	0.370	1.505	0.129	0%
side	xNES	1816	0.401	1.229	0.490	1.092	0.137	0%

For the model of the first experiment, we used polynomial basis functions with five parameters. Fig. 3.17a shows the joint angles of the balance motion, and Fig.

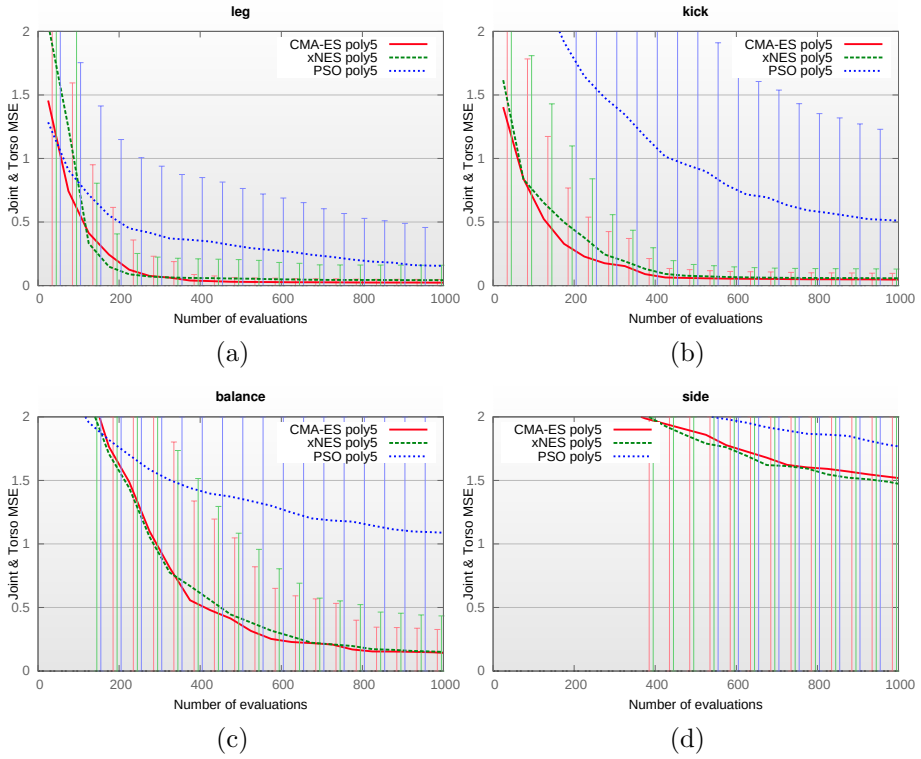
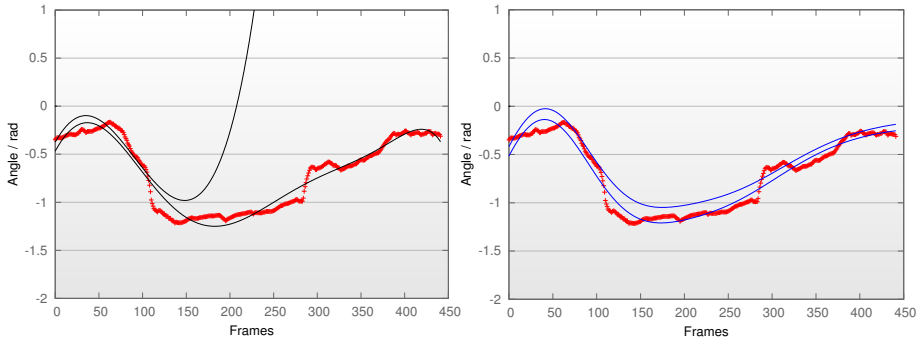


Figure 3.18: The learning curves for different motions using polynomials of degree four. The motions for the twelve leg joints were optimized using CMA-ES, PSO and xNES. The error is the minimum total error (joints MSE + torso angle MSE) averaged over 30 runs. The error bars represent one-standard deviation over these 30 runs.

3.17b shows the traces of two of the joints. The optimized motion slightly adjusts the initial values of the parameters to obtain stable motion, and the combination of all joint traces together needs to be stable. However, the experiments show that often only very small changes in the joint motion provide a stable and complete motion. In Fig. 3.17b the polynomial seems to be able to sufficiently approximate the motion of the joint.

We used the same polynomial approximation to learn four different motions, and Fig. 3.18 shows the average learning curves. In these experiments, both CMA-ES and xNES quickly learn solutions, with CMA-ES finding a solution with a slightly

smaller variance. Both algorithms perform better than PSO. It is possible that the results of PSO could be improved by tuning some internal parameters; CMA-ES and xNES do not require this.



(a) (b)

Figure 3.19: An example joint motion (l_knee_pitch) of the side balance and the initial and learned model using CMA-ES and (a) a polynomial with eight parameters or (b) sigmoidal basis functions with a sum of eight parameters.

There is a swift learning curve for leg motion. For the kick motion, PSO yields a very high variance, which indicates that the found motion is often unstable. For the balance motion in Fig. 3.18c, CMA-ES and xNES find good solutions, but the learning time increases. While the polynomials work for these three motions, the algorithms could not find a stable motion for the side balance motion in Fig. 3.18d. In fact, the results for the other motions are also often unsatisfactory. Polynomials cause these motions to be very smooth. Although the errors are often small and the motion is stable, there is a noticeable lack of detail, and the high-degree polynomials introduce numerical instabilities. In most motions, polynomials cause some joints to move unexpectedly towards the end.

Table 3.4 shows the error values of the experiments after five hours of simulated

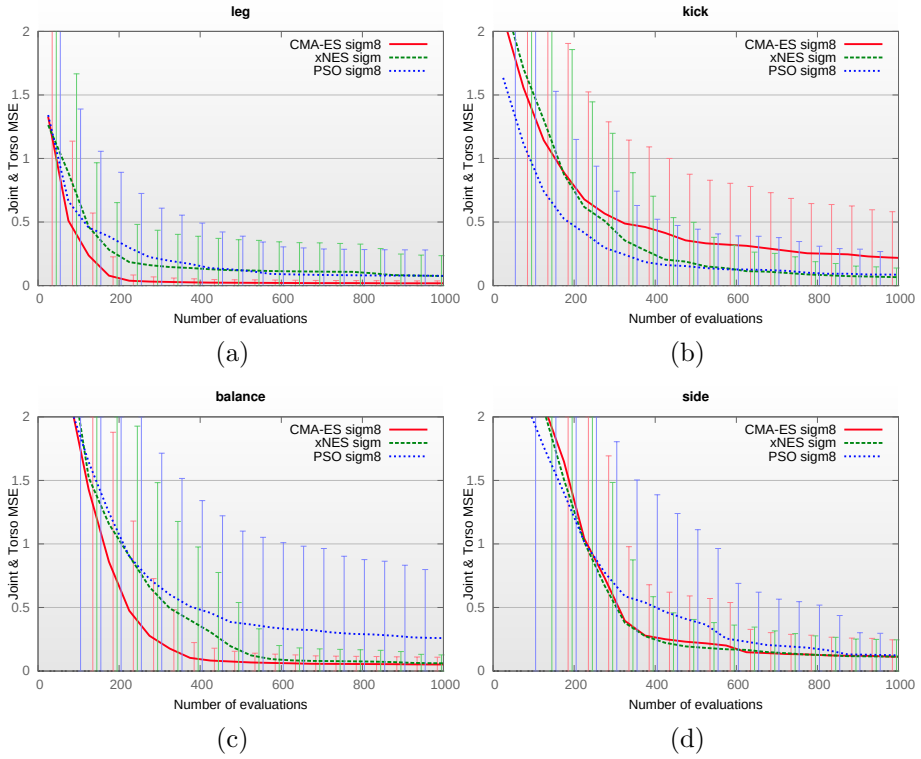


Figure 3.20: Using sigmoidal basis functions instead of polynomials improves the results of the learning. The learning curves of the same experiments as in Fig. 3.18 are shown (averages over 30 runs).

time. The success rates are created by manually evaluating how many result motions are stable and close enough to the original motion. A longer learning time improves the success rates. However, another reason for lower success rates, despite small average errors, is the noise in the fitness values. There is a chance that an unstable motion gets a small error once and never works again. Averaging the fitness over several runs could lower this noise and improve the learning, but each evaluation would need much more time.

As an attempt to improve the learning, we ran the optimization of the side balance motion again using polynomials, but increased the number of parameters to eight.

It still did not find a solution. Increasing the number of degrees only creates more instabilities. Fig. 3.19a shows that a reason for the unsatisfactory performance is the global influence of parameters on the function. Changing the first part of the motion can create completely wrong angles for the remaining motion.

Table 3.5: Results of the optimization using sigmoidal basis functions.

Motion	Optimization	evals	min.err.	avg.err.	stddev	torso err.	joint err.	success
leg	CMA-ES	1781	0.011	0.016	0.004	0.007	0.008	93%
leg	PSO	1782	0.005	0.051	0.149	0.044	0.007	76%
leg	xNES	1782	0.012	0.042	0.086	0.027	0.014	70%
kick	CMA-ES	2935	0.029	0.150	0.206	0.124	0.027	53%
kick	PSO	2935	0.012	0.038	0.068	0.029	0.009	43%
kick	xNES	2940	0.037	0.047	0.007	0.025	0.023	40%
balance	CMA-ES	2970	0.029	0.042	0.016	0.018	0.024	93%
balance	PSO	2971	0.027	0.119	0.256	0.092	0.027	73%
balance	xNES	2971	0.030	0.041	0.009	0.020	0.020	76%
side	CMA-ES	1816	0.041	0.099	0.060	0.073	0.026	70%
side	PSO	1816	0.024	0.093	0.085	0.064	0.029	60%
side	xNES	1817	0.038	0.102	0.062	0.078	0.024	40%

Since the side motion could not be learned at all and the other motions suffered from the high generalization and numerical instabilities of the polynomials, we ran the same experiments again with the linear weighted sigmoidal basis functions as the model. Fig. 3.19b shows that the local basis functions improve the optimization. All four motions can be adjusted to be stable on the robot using this model (Fig. 3.20). Although the number of parameters that are optimized has been increased from 60 to 96 (twelve joints, five or eight parameters per model), the optimization needs less time to find solutions for the balance motion and also works for the side balance. The average joint and torso errors are listed in Table 3.5. For the balance motion the improved models yield significantly smaller joint angle errors compared to the polynomials.

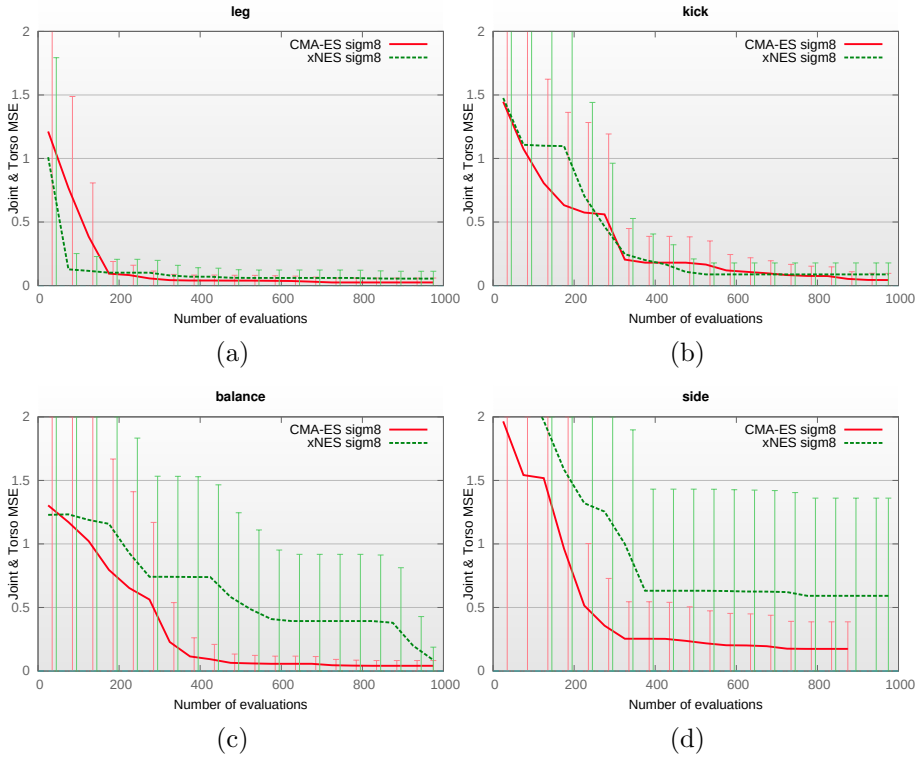


Figure 3.21: The same optimization works in a different simulation environments. These results from the optimization in Webots are very similar to the results using SimSpark (sigmoidal basis functions, CMA-ES, averages over three runs).

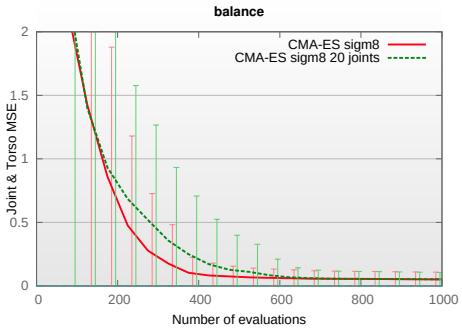


Figure 3.22: The optimzation of the balance motion using only the 12 leg joints compared to the results for the optimization of all 20 leg and arm joints (using sigmoidal basis functions, CMA-ES, averages over 30 runs).

In all previous experiments we only optimized the joint angle traces for the leg joints. The arm joints always followed the angle traces obtained during the initial-

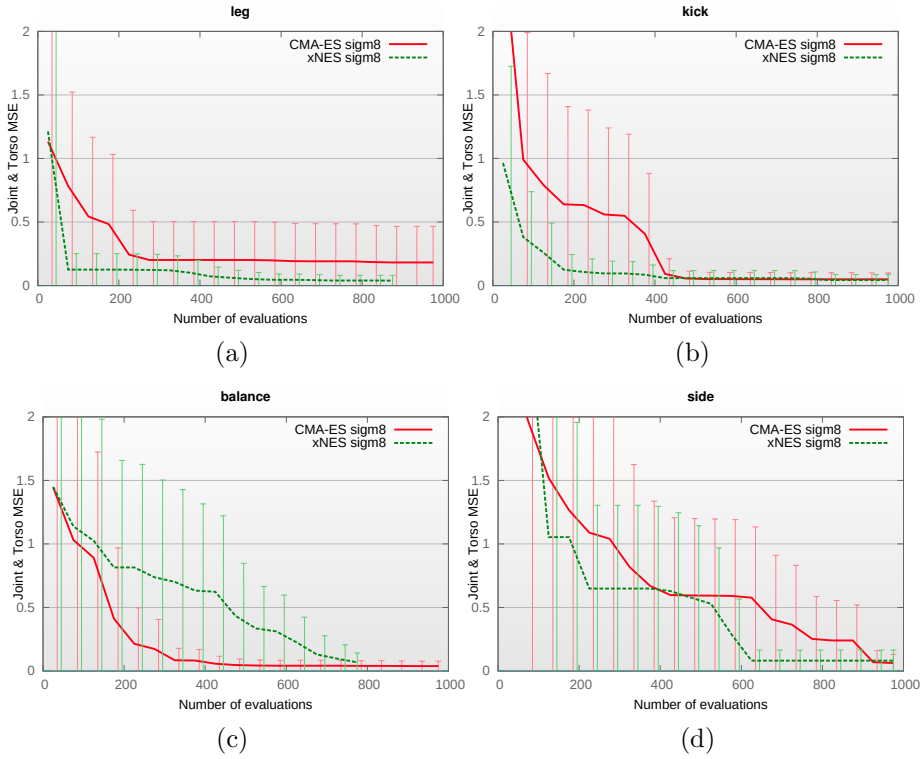


Figure 3.23: The results of the optimization in Webots without ground truth (sigmoidal basis functions, CMA-ES, averages over three runs).

ization. We assumed that small changes in the arm movements only have a minor influence on the overall motion. To verify this assumption we compared the previous results for the optimization of the balance motion when the optimization includes all 20 joints. Figure 3.22 shows that the optimization including the arms yields higher errors and variances without finding smaller errors in the end.

Experiments in Webots

The goal is to create motions for the physical robot. As the next step towards this goal, we have repeated some of the experiments in Webots simulator. The NAO model in Webots is slightly different from the model in SimSpark and the physics seem to

be more accurate. The behavior of the NAO on Webots is closer to the behavior of the physical NAO. Motions that were optimized and stable in Simspark are stable neither in Webots nor on the physical robot. In fact, the robot in Webots and the physical robot sometimes fall in a similar way.

However, the simulations in Webots can not run faster than real-time using the NAO model in Webots (since Webots and the NAOqi controller are not synchronized). Therefore, the evaluation of parameters takes significantly more time than in SimSpark and we are stipulated to reduce the number of experiments. In all experiments using Webots and on the physical robot, we used only sigmoidal basis functions and CMA-ES or xNES to optimize the angle traces of the 12 leg joints. The exhaustive experiments in SimSpark have shown that this configuration yields good results. Figure 3.21 shows the errors during the optimization in Webots for the same motions as in Fig. 3.20. The results are very similar to the optimization in SimSpark.

Overall, we have observed that CMA-ES and xNES are similar in performance. Also, the PSO algorithm shows partly smaller errors with a larger variance and it takes longer to learn. The optimization using models with sigmoidal basis functions yields good results for all three algorithms. Optimizing only the leg motions is sufficient for the used robot model.

Some small modifications have to be considered to apply the same optimization on a physical NAO. Since the physical robot can easily get damaged during experiments, we will have to stop the motion and measurement when the robot falls. Therefore, for the experiments in Webots, we already have added a threshold for the error. If the error exceeds this threshold, the motion is stopped and all remaining frames of

the motion get the last measured error is assigned. As the results show, this does not effect the optimization procedures and even speeds up the evaluations in the beginning when the robot falls significantly.

For the optimization in the physical robot, we will have to calculate the error without ground truth. The orientation of the torso will be estimated using the gyroscope and accelerometer of the NAO. Since the NAO has only a 2-axis gyroscope, rotations about the Z-axis (upright) can not be measured. Figure 3.23 shows the results of the optimization in Webots without the ground truth provided by the simulator. The optimization can still find joint angle traces with small errors. However, depending on the motion, the missing ground truth information causes the optimization to converge slower (motion side balance) and sometimes does not find a good solution at all (higher variance of motion leg).

3.3 Parallel Optimization

The motion optimizations discussed in the previous sections are done using simulated robots. Therefore, it is possible to run many iterations of the optimization without breaking the robot. Since the simulated robot always behaves the same in the simulation (the simulation always adds noise the same way), we can use multiple simulators to run the optimization in parallel.

There have been several approaches for parallelizing optimization algorithms. Hakkarinen et al. [24] presented a development of a parallel CMA-ES algorithm that reduces the runtime for a specific geophysical data analysis, dipole localization.

For problems with dimensions as high as 400, a cloud scale distributed covariance matrix adaptation based evolutionary strategy was developed and evaluated in [95]. Müller et al. [54] developed a computationally efficient, scalable and portable software library that implemented parallel CMA-ES and some other variants in Fortran. Wong [97] and Fok et al. [15] proposed to implement a parallel EA on consumer-level graphics cards which gave considerable speed up for a large population size. Parallelizing in Graphics Processing Unit (GPU) is always effective owing to its massively parallel architecture and many researchers have used it in various ways to parallelize both EA and PSO [86, 34, 74, 56, 57].

As the motion optimizations discussed in the previous sections use parameter optimization, they are all based on evaluating parameter sets. This is done by executing a behavior or motion to evaluate the parameters. The robot's behavior has to be observed to calculate the fitness using a predefined fitness function. In the methods used (PSO, CM A-ES or x-NES), the updates in the optimization algorithm to provide parameter sets are computationally much less expensive than evaluation of the parameter sets. Most of the CPU time is used by the physics simulation of the robot. Therefore the optimization algorithm itself does not have to be parallelized.

Optimization methods that work with populations of candidate parameter values can provide all individuals of a generation at once. These candidates can then be evaluated completely independent from each other. Therefore, an optimization using a population size k can reach a speed up factor close to k if the evaluation of the parameters needs most of the CPU time and assuming there is only a low overhead and no communication delays.

Section 3.3.1 describes the parallelization of the optimization of the captured motions as in section 3.2. Section 3.3.2 shows the result for the same parallelization infrastructure applied to the kick and stand up motions for SimSpark from section 3.1.1.

3.3.1 Parallel optimization of captured motions

We have chosen to implement the client-server model to achieve parallelism as it would need minimum modification of the existing code and all the different optimization algorithms can be used with the same model without any need to change the optimization algorithms code. The basic idea of the parallelization approach is to distribute the fitness evaluation phase of the sequential approach into multiple systems such that there can be multiple simulations for fitness evaluations of different parameter sets at the same time. In the following, each parameter set that needs to be evaluated is referred to as particle (as in PSO).

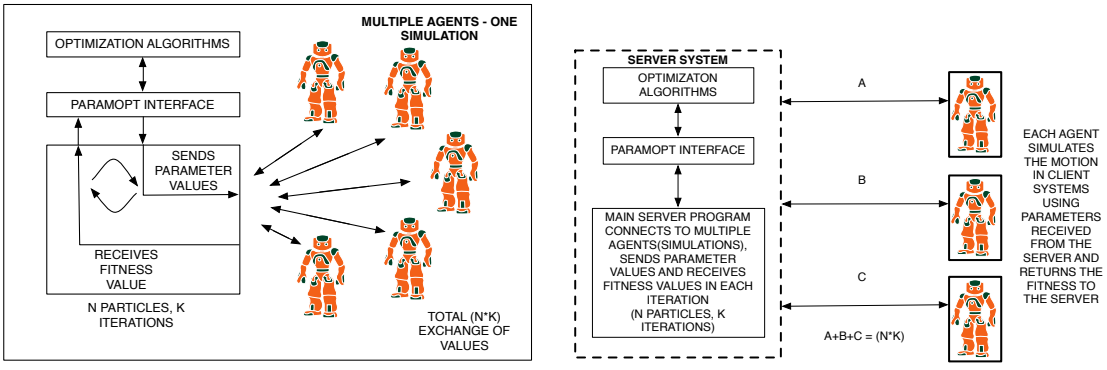


Figure 3.24: (a) Parallel approach using multiple (five shown here) agents and one soccer simulation. (b) Parallel approach using multiple agents and multiple simulations (three agents and three soccer simulations shown here).

The process starts with initialization and the resetting of the particle's parameters like the sequential approach. The only difference is that the server is now responsible for this part of the work. All the client system connect with the server using TCP connections. The sending phase of the parallel process starts where the server sends the particles to different clients. For this, it is required that multiple clients connect to the server through a port that the server listens on. After the client is connected, the server sends a particle to the client. The server keeps track of the particle number that was sent last and thus sends the next particle to the new client. For example, if there are k clients connected to the server at any point of time and the server has sent already n particles, then in the next sending phase, a total of $n+k$ particles are sent to the clients by the server. After each client receives a parameter set, it is responsible for evaluating the fitness of the particle after one simulation. Each of the simulations are independent of one another and after they are done, each of them sends the errors separately to the server. The server collect each error as it receives and sends it to the optimization algorithm method using the particle number as the index of the evaluation. The server then updates the count of the particles received until a certain point of time ($n = n + k$) and then checks to see if it has reached the population size, i.e. whether one iteration has completed or not. Similar to the sequential approach, the value of n can lead to two different cases. If the iteration is not complete then the server goes back to the sending phase and sends the rest of the particles left for the iteration to complete which is *POPULATION_SIZE* - n to the clients. If the value of n becomes equal to the population size then the server calls the *update* method of the optimization algorithm which updates the particles'

parameters. It also sets the counter n to 0 to restart the entire process.

One way to implement it is having all of the agents connect to a single soccer simulation (Figure 3.24a). The main drawback of this process is that this is not ideally parallel. The reason behind this statement is that when an agent connects to a soccer simulation server, it communicates with the server and exchanges values in order to play the motion. The soccer simulation can send and receive messages from only one agent at a time. Thus when multiple agents are connected to the soccer simulation server, then all the messages from the agents are sent to the server in sequence and received in a sequential order as well. The soccer simulation is quite fast and running multiple agents is not much of a problem but the behavior of the soccer simulation server is definitely not parallel and thus does not give us the maximum throughput.

Another way to parallelize the approach is to have one simulation corresponding to each agent where the server connects to multiple agents and sends the parameter values generated from the underlying optimization algorithms (Figure 3.24b). Each agent connects to its own soccer simulation server and exchanges messages to and from the soccer simulation server in order to run a simulation i.e. play a motion using the parameters received from the main server of the client-server model. They evaluate the fitness independently like the other approach and send it to the main server program and then wait to receive the next parameter set from the server to play the motion using the new values. Each of the agents along with their soccer simulation servers are independent of each other and thus can be said to be ideally parallel. Each client system can be envisioned as an individual binary of the agent running in

a separate processor. Each agent sends its errors to a client optimization interface. This client optimization instance forwards the errors to the server which actually interacts with the optimization algorithms and updates the particles thus generating a new set of particles. The server sends the particles to the agent via the client opt interface which forwards the values to the agent. The agent gets the parameter set and then exchanges the necessary values with its own soccer simulation values in order to play the motion. This kind of a setup enables the entire optimization process to be ideally parallel and also allows us to parallelize the approach to any number of clients. We can use multiple processors or systems to host multiple agents with their simulations and a single system running the server program. Since this entire setup is based on TCP connections, it is not necessary to have an interconnected set of computer systems to run this in a distributed manner. If we know the IP addresses of the different computer systems, we can run this on any number of systems available to us at any point of time. Since each agent is connected to its own soccer simulation server, there will be no delay in the simulation and it will be faster than the previous architecture. It is designed in a way that the amount of particles each client gets is not the same but depends on the speed of the system, then the mechanism is made dynamic i.e. a faster agent will get more parameter sets to evaluate rather than a slower agent.

Experiments & Results

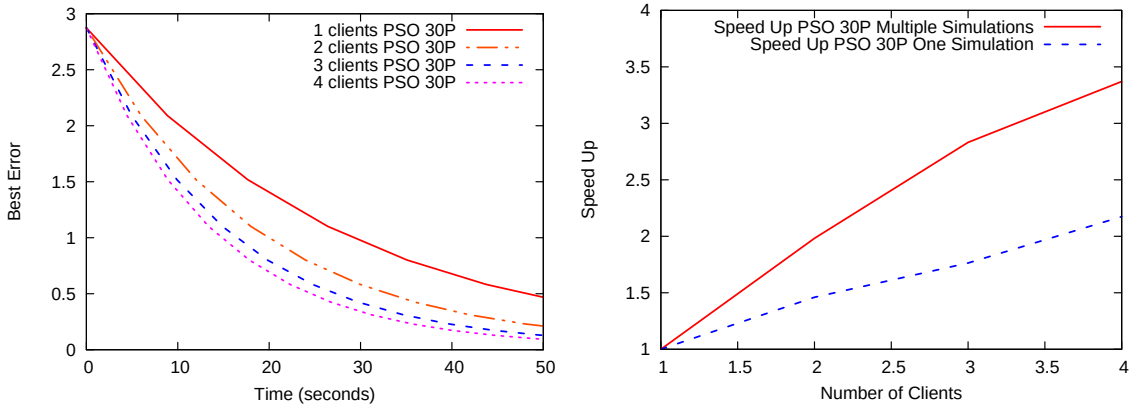
The experiments were performed using four different types of human motions and each of them were optimized by the three optimization algorithms: PSO, CMA-ES and

xNES. These motions were captured by *Kinect* and then processed further to produce four individual motion files (cf. figure 3.27 on page 70 to see the motions) consisting of the sequence of joint angles that can be played on the simulated humanoid robot.

The optimization experiments were run in three different experimental setups. The first setup is a single multicore system called the *RegenBase* system with 2.0 GHz, 15MB cache $Intel^R\ Xeon^R\ E5\text{-}2620$ processor. It has a total of 12 cores. The second setup is a network of 20 dual core computer systems/nodes where each of the system has a 2.4 GHz, 4MB L2 cache $Intel^R\ Core^{TM}\ 2\ Duo$ processor. The experiments used a total of 24 cores. The third setup is on *Pegasus* which is a parallel cluster of computers where the maximum number of cores used for a particular experiment was 30. The server and the clients communicate with each other using a combination of TCP and a special set of libraries, Message Passing Interface (MPI).

After multiple preliminary experiments we found that optimization using a population size of minimum 30 particles gave us a solution for the motion in an acceptable amount of time. Hence all the experiments of multiple agents, multiple simulations approach were performed with a population size of 30 particles. All the experiment results are averaged over 30 runs. It should be noted that a low error does not always guarantee a stable motion because of some noise in the fitness level. There is a chance that an unstable motion gets a small error once and never works again. The performance of the optimization process is measured in the form of time taken in milliseconds. Our experiments were successful in finding a solution for the optimization process and all the motions were *learned* i.e. when played on the simulated humanoid robot, these motions were found to be stable. From the performance point of view, it

was seen that there is considerable speed-up in the process as we increase the number of nodes for the optimization process.



(a) (b)

Figure 3.25: (a) Decrease in best error over time with optimization of a leg motion using PSO optimization algorithm with 1, 2, 3 and 4 clients. (b) Speed-up comparison between multiple agents, single simulation approach and multiple agents, multiple simulations approach for 1, 2, 3, and 4 agents.

It can be seen from figure 3.25a that with the parallel approach, the error decreased over time as the number of clients were increased. Figure 3.25b illustrates the difference in the approaches and shows that the approach using multiple agents, multiple simulations performs better and achieves higher speed-up as compared to the multiple agents, single simulation approach for experiments performed using 1, 2, 3 and 4 agents. The reason for this difference in results between the two approaches is explained in the previous section 3.3.1.

Since the approach with multiple agents and multiple simulations was found to yield higher speed-ups than multiple agents, single client approach, we performed the following experiments using the better approach in different experimental setups.

We observed in the first setup, which is the single multithreaded 12 cores *Regen-*

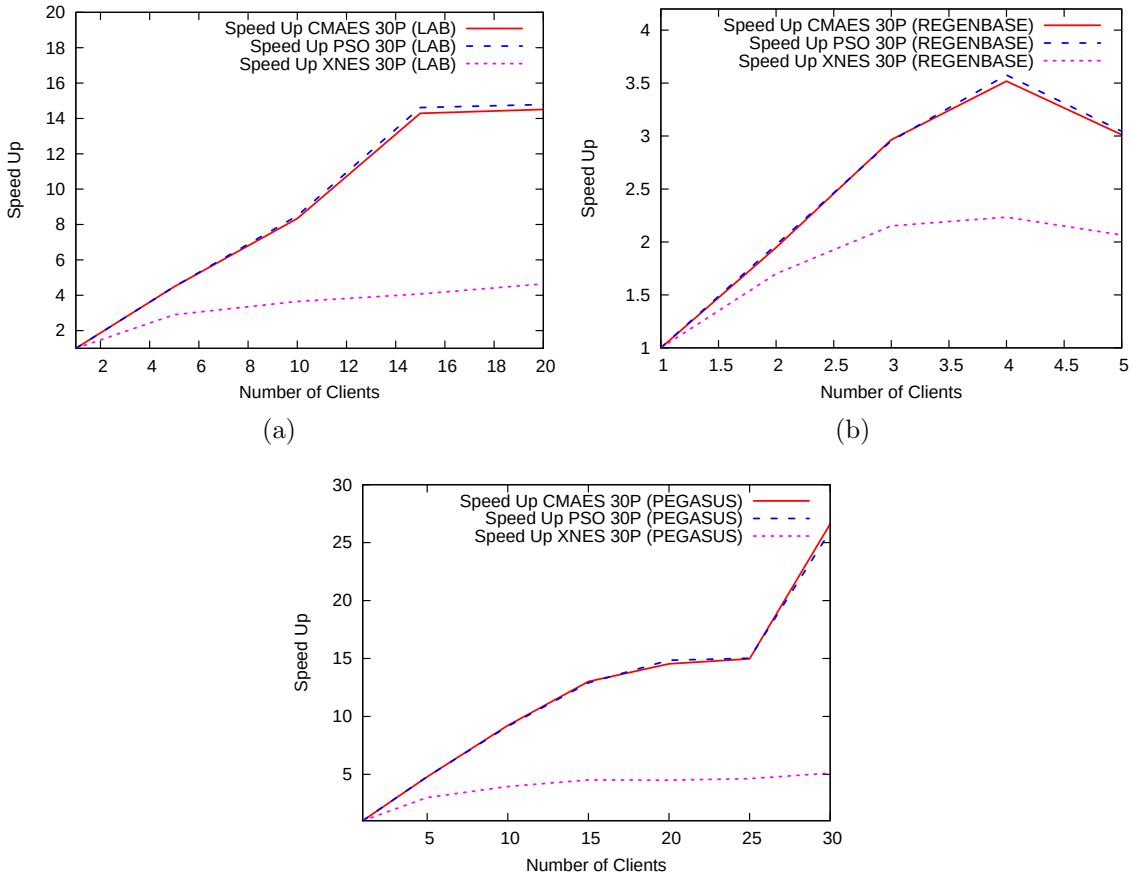


Figure 3.26: (a) Speed-up for all the optimization algorithms using 1, 2, 3, 4 and 5 clients (1,500 evaluations) - *RegenBase System* (b) Speed-up for all the optimization algorithms using 1, 5, 10, 15 and 20 clients (1,500 evaluations) - *Lab Systems* (c) Speed-up for all the optimization algorithms using 1, 5, 10, 15, 20, 25 and 30 clients (3,000 evaluations) - *Pegasus Cluster*.

Base system, on that the soccer simulator server (*SimSpark*) used up to 300% of the CPU i.e. used 3 cores and thus the experiments were made with 1, 2, 3, 4 and 5 clients to show the speedup in the process. In these experiments, the server and all the clients were running on the same system. Figure 3.26a shows almost linear increase in the speed-up of the optimization process with 2, 3 and 4 clients as compared to the one client optimization with an average speed-up in the range of 1 to 3.11 for

1 to 4 clients. Howerver, the speed-up value for 5 clients was not found to be better than for the optimization process running with 4 client simulations with an average speed-up value of 1.708 (Figure 3.26a) because the system did not have enough cores to support simultaneous running of 5 different *SimSpark* simulators considering the CPU usage percentage of one simulator. We can see that we never achieve a linear speedup. This is due to the fact that there is always an overhead of sending and receiving values using TCP and also an overhead due to certain parts of the algorithm which run sequentially. Since the number of clients range from 2 to 5 in these experiments, the overhead does not account for much considerable effect in the execution time of the program and we get almost linear speedup.

The next set of experiments were conducted with a group of 20 individual systems. Experiments were made with 1, 5, 10, 15 and 20 clients and 30 particles as the *POPULATION_SIZE*. It can be seen in figure 3.26b that the speed-up for the optimization experiments using clients 15 and 20 are almost similar. This is because of the fact that for the optimization algorithm using 30 particles as the population size, the sending phase of the algorithm can send the particles to 15 clients in two batches and even for 20 clients, it has to send the particles in two batches. Similar to the sending, the receiving phase also receives fitness values in two batches for both the cases and thus there is hardly any sizable difference in the time taken by the optimization process for 15 and 20 clients. Nevertheless, the optimization process using 20 clients does yield a slightly higher speed-up as it has some extra clients that can compensate for any slow clients in the set up, owing to the fact that the algorithm is implemented in a way such that the faster clients get a higher number of particles.

For the 15 clients optimization, a single slow client can become a bottleneck in the performance of the optimization process as there are no extra possibly faster clients to which the particles can be sent. Having extra clients over 15 helps the process to compensate for any slow clients present in the optimization process and thus helps in reaching a speed-up value of about 15.

The graph 3.26c for the next set of experiments, performed in *Pegasus* with a maximum number of clients of 30, shows similar nature of the plots. If we see figure 3.26c, we see a sudden jump in the speed-up with respect to the corresponding speed-up value for the 25 clients optimization experiment. In this case, the sending/receiving phases send/ receive values in one batch, thus almost reducing the time by half when compared to the time taken for sending and receiving for 15, 20 or 25 clients optimization. The speed-up increases with an increase in the number of clients as illustrated in figure 3.26c. This is the maximum level of parallelism that this optimization using 30 particles as population size can reach. If we use more than 30 clients, we can expect a slight increase in the speed-up and maybe achieve a speed-up value close to 30 as long as we can get most clients (at least 30) which are not busy running other programs and can run the individual simulations fast. Table 3.6 shows the results for the experiments performed with a particular leg motion on *Pegasus*.

After seeing all the results for the three different experimental set ups, it was observed that for XNES algorithm, the speed-up does not scale with the increase in the number of clients like the other algorithms, PSO and CMAES. We performed three experiments to investigate this observation by recording the execution times of the different parts of the server program. The percentage of the total execution time taken

Table 3.6: Results of parallel optimization experiments in Setup 3 with different numbers of clients and their respective speed-ups for the leg motion.

SETUP 3 EXPERIMENTS - Leg Motion

Opt.	Evals.	RMSE	Min. Error	Client(s)	Time (ms)	Std. Dev. (ms)	Speed-Up
PSO	990	0.90704	0.00669	1	415083	4321.66	1.000
PSO	990	0.88491	0.00552	10	47223	1650.44	8.790
PSO	990	0.84064	0.00630	20	29320	434.97	14.157
PSO	990	0.82386	0.00757	30	15724	279.67	26.398
CMAES	990	1.17818	0.01706	1	424373	5064.30	1.000
CMAES	990	1.19996	0.01805	10	51092	1636.81	8.306
CMAES	990	1.19247	0.01674	20	29295	858.86	14.486
CMAES	990	1.16812	0.01508	30	16155	620.07	26.269
XNES	990	1.18037	0.01951	1	524482	4653.31	1.000
XNES	990	1.16340	0.01679	10	154422	3159.07	3.396
XNES	990	1.21483	0.01742	20	138786	3833.12	3.779
XNES	990	1.19902	0.01993	30	121072	1176.77	4.332

Table 3.7: Percentage of the total execution time used in different phases of the server program.

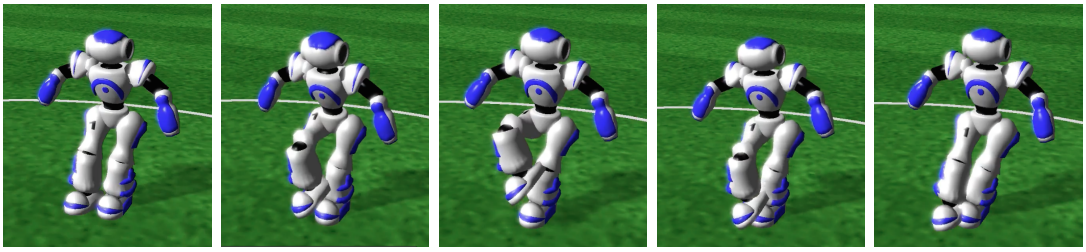
PERCENTAGE OF THE TOTAL EXECUTION TIME

Opt.	Clients	Update Phase (%)	Sending & Receiving Phases (%)
PSO	1	0.0126	99.9873
PSO	10	0.1048	99.8938
PSO	20	0.1538	99.8443
CMAES	1	0.0937	99.9062
CMAES	10	0.9119	99.0870
CMAES	20	1.3799	98.6184
XNES	1	13.8059	86.1939
XNES	10	61.4714	38.5279
XNES	20	71.2221	28.7771

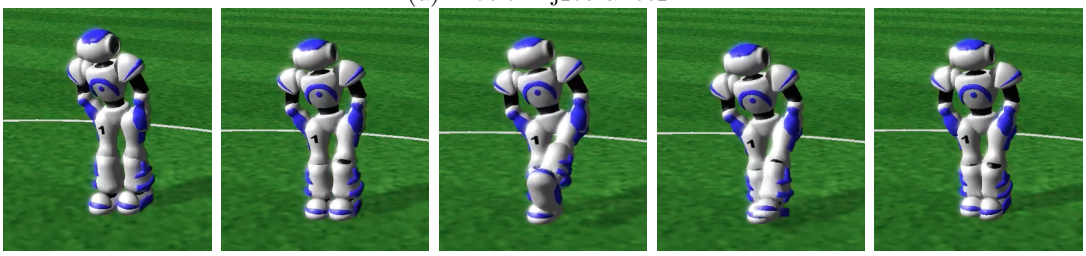
by the two major phases; update phase and the sending and receiving phases is shown in table 3.7. The results reveal that the execution time taken by the update phase of the algorithm, which is the only sequential phase in the parallel algorithm, was much higher than the time taken by the sending and receiving phases of the algorithm in case of XNES as compared to the other algorithms. As the number of clients increases in the parallel approach using the XNES algorithm, the execution time spent by the sending and receiving phases decreases. This is because these phases are meant to be parallel whereas the time taken by the sequential phase i.e. the update phase remains

the same for all the multiple client optimization experiments, thus accounting for a higher percentage of the total execution time thus limiting the speedup.

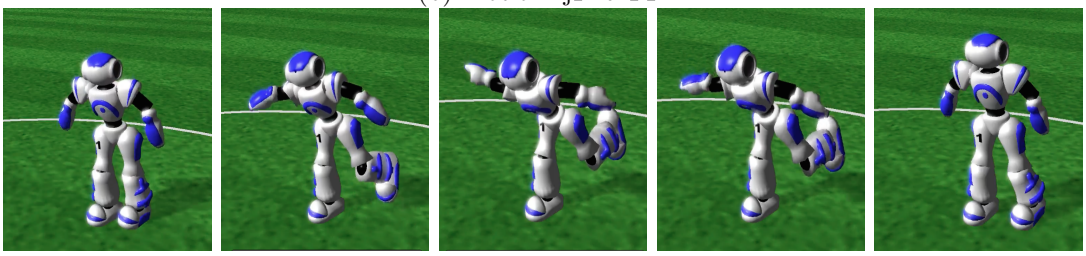
The four human motions as shown in figure 3.26 were optimized using the optimization algorithms and was played in the simulated humanoid robot. The optimization algorithms were successful in finding a stable motion for each of the four human motions which are shown in figure 3.27.



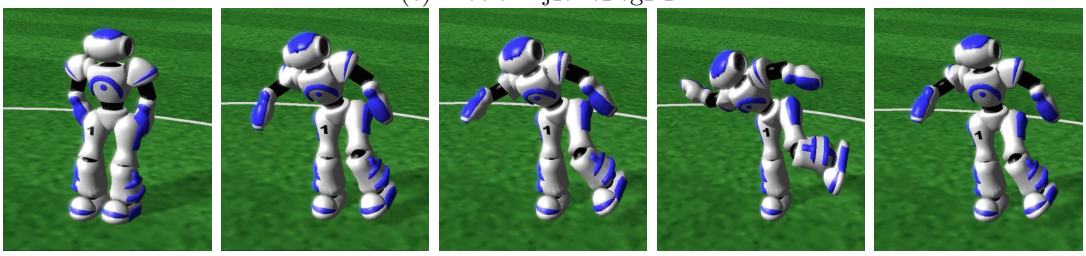
(a) Motion: j_balance_2



(b) Motion: j_kick_l_1



(c) Motion: j_one_leg_r_2



(d) Motion: j_side_balance_1

Figure 3.27: The optimized human motions played on the simulated humanoid robots

3.3.2 Parallel Optimization of Soccer Motions

The infrastructure for the parallel optimization of motions described in the previous section can be used for any optimization in the simulator, since it is only based on the independent evaluation of candidate solutions. The optimizations of soccer motions for the 3D simulation league described in section 3.1 uses the same parameter optimization classes and interfaces as the optimization of the captured motions. Therefore, it can run in parallel just by activating the optimization client module in the agent.

For the special action optimization, the time needed for each evaluation is even longer than for the motion optimization in the previous section. Therefore, there is still only a small overhead caused by the network communication and the parallelization yields the same good speed-up as for the parallel optimization of the captured motions show in section 3.3.1.

By running the optimization in parallel on a cluster, it is possible to run various optimizations quickly, for example to find weight factors for the components of the fitness functions that produce the desired results. We also ran the optimization using different fitness functions to create different kicks, such as a kick for a certain distance or a very high kick. Instead of running the optimization sequentially for one or two days, the same results can be produced by running it for an hour on an MPI cluster with 50 clients.

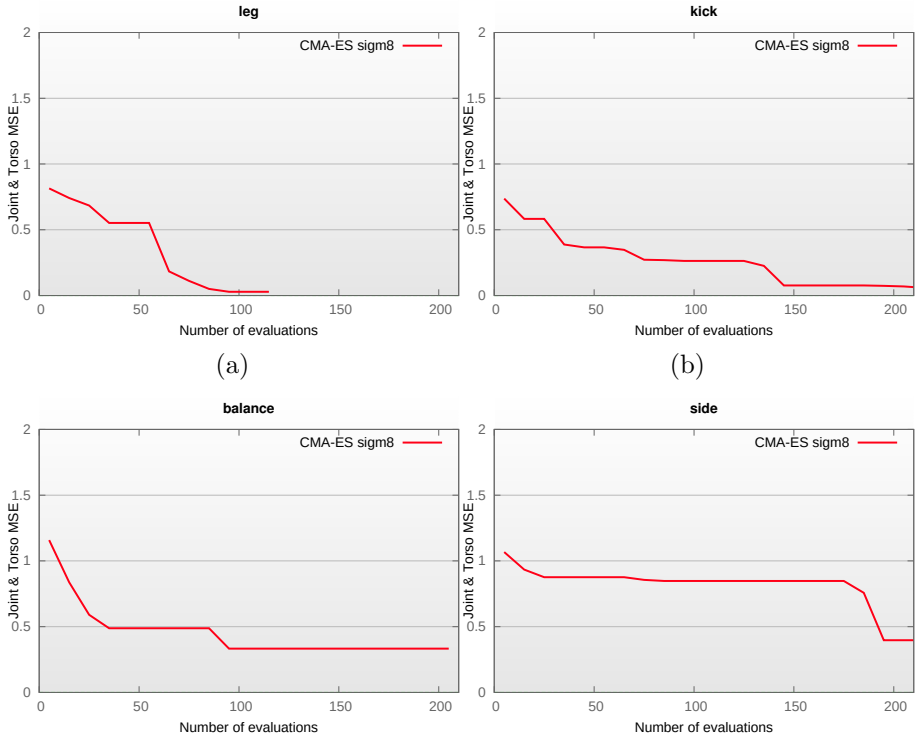


Figure 3.28: The results of the optimization for the physical robot using seeds from the simulation (sigmoidal basis functions, CMA-ES, single run).

3.4 Motion Optimization on a physical robot

This section shows the results of the motion optimization from section 3.2 on a physical robot.

3.4.1 Experiments on the Physical Robot

Parameter optimizations often need a lot of evaluations. On physical robots, some optimization problems are very challenging or even unfeasible. The approximately 2000 evaluations we executed for each experiment in SimSpark and Webots are already too much. It would stress the hardware and take too much time. In contrast to

the fully automated simulated experiments, now a human operator is needed, who positions the robot, starts the motion and catches the robot when it falls to avoid damaging the robot.

During the experiments we made sure not to hold the robot at any time during the evaluation of a motion. Only when the motion causes the robot to start falling, we let it fall but slowed it down before hitting the ground. This way we did not interfere with the learning process too much and were able to obtain useful error measurements.

According to the simulation result, the simplest motion in our experiments is the leg motion. However, even for this relatively simple motion we needed 200 to 300 evaluations to find stable results. These are still many evaluations, if they need to be executed using the NAO.

We reduced the number of evaluations by using simulation results as a seed for the optimization on the NAO. The parameter values, used to create the seed motion, are only used as the mean in the initialization of the individuals of the optimization algorithms. The original capture data is still the reference in the calculation of errors during the evaluation of parameters, since this is the motion we are trying to imitate. Additionally, we reduced the population size from 50 to 30. Figure 3.29 compares the errors for the leg motion learned with seeds from the simulators or without a seed. We stopped the optimization when a good solution was found or after at most 200 iterations. Even in the simulation 200 evaluations are not always sufficient to find a stable solution for the leg motion. Without a seed the learning did not find stable motions within 200 evaluations. If no seed is used, almost all parameters during the

first iterations make the robot fall immediately. This can be avoided by using results from the simulation as a seed. The simulated robot models are sufficiently accurate to provide seeds that are already close to stable solutions for the NAO. Skipping the unnecessary exploration in the beginning speeds up the learning significantly, which makes the method feasible for a physical robot.

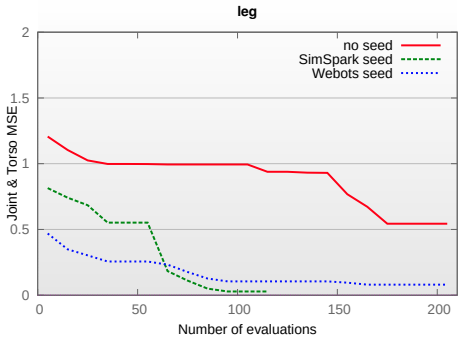


Figure 3.29: The optimization of the leg motion with different seeds / no seed (using sigmoidal basis functions, CMA-ES, single run).

Figure 3.29 shows that we found stable results already after 100 evaluations. Although the robot behavior in Webots seemed to be more realistic in the previous experiments, optimization results from Webots are not necessarily better seeds for the NAO. The differences of the simulators do not have a significant influence on the seed quality. There are other factors that can cause a solution from the simulation to be a less efficient seed. Since, we used black box optimization of the joint angle traces of the motion, sometimes small error values were reached by exploiting missing self-collision checks in the simulators. Therefore, some motions learned in the simulator have a small error value, but are less useful as a seed for the optimization on the NAO robot.

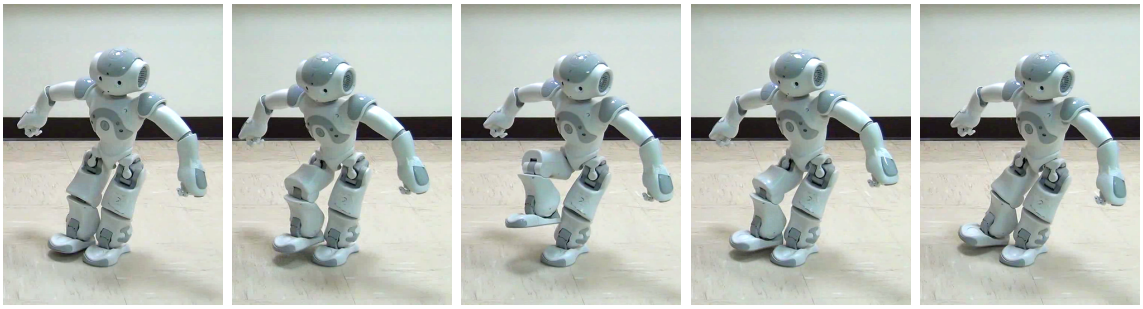
Using seeds from the simulation we were able to find solutions for three of the

four test motions that we used also in the simulation experiments. Figure 3.28 shows some example learning curves. The generated motions are shown in Figure 3.30. In general, our approach works on the NAO. As expected, the results are not as good as in the simulation and there are high variances in the results. As mentioned in section 3.2.4, the optimization does not try to start or end the motion with given poses. We do not learn transitions. Thus, the starting pose in the first frame and the last part of the motion can vary without increasing the error much. The objective function needs to be improved to yield more consistent results. Another problem for the optimization using the NAO is the backlash in the joints. The initial pose and foot positions can vary, especially on carpet, if the foot gets stuck in different positions. This complicates the optimization on the physical robot.

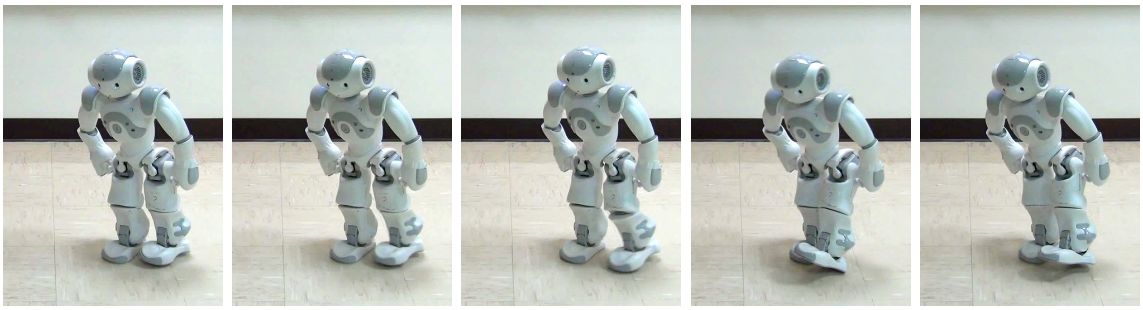
Some motions could not be stabilized completely. For example, the remaining error for the balance motion in 3.28c indicates that the best result found was still not completely stable. The side balance motion is challenging, too. The results in Webots (Fig. 3.23d) showed already that it often takes many iterations to find good parameters. The experiments showed that this is caused by a local minimum. The robot is stuck in a simple motion that is easy to balance, but from that it is difficult to reach the expected motion. Nevertheless, it was possible to find stable results for the other three motions. The MoCap motions are static joint angle traces, that can only be perfectly stable in a simulated environment. A dynamic balancing is needed to improve the results.

Overall, the results show that the optimization also works on a physical robot, but

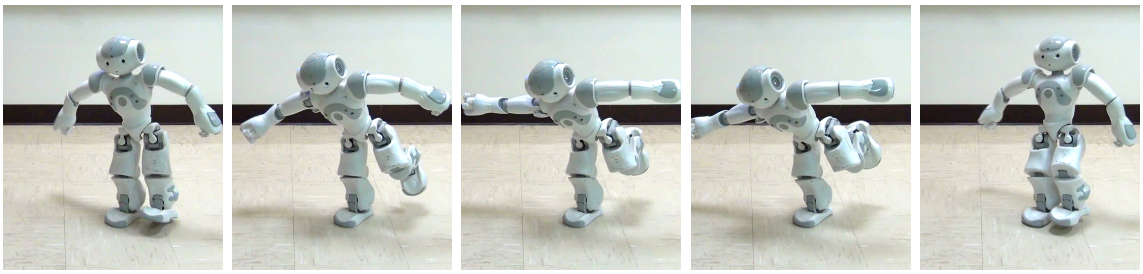
[7]Video material: `http://web.cs.miami.edu/home/aseek/motionlearn/`



(a) leg



(b) kick



(c) balance

Figure 3.30: The optimized motions on the NAO.[7]

the number of evaluations has to be reduced further. The optimization on a physical robot is possible, but still tedious. The generated motions are stable, but they need to be improved to be used e.g. in the soccer environment. The generated motions need defined poses before and after the learned motions and allow transitions from and to other motions. Additionally, the results on the physical robot are highly optimized motions, that are only stable under the exact conditions under that the optimization was done. Since the generated motions are static sequences of joint angle, we did not

expect the results to be perfectly stable using a physical robot.

3.5 Conclusion

The optimization of motions for the RoboCanes agent in the 3D Soccer Simulation League has shown the potential in optimizating motions. The resulting motions have increased the teams performance significantly.

Furthermore, this chapter described a system for creating human-like motions for a biped robot using inexpensive motion capture techniques combined with optimization. The Kinect sensor provides enough information to map complex and initially unstable poses, such as the balancing motions. Applying modern optimization algorithms to the mapped motions will lead to robust and stable motions on the robot; that said, we do not make the claim that our approach will hold for *all* possible motions. Nevertheless, our results are very promising, and it shows that we have found a process that can be used to produce a number of motions needed for humanoid soccer robots. We ran our parameter optimization on a desktop CPU with four cores and 2.27 GHz. After running simulation experiments with four motions, three algorithms, two model functions with different number of parameters and 30 runs each, we can say that the motions were mostly stable after 30 min (in average). Making use of parallel processing on a cluster of computers can bring the processing time down to 10-30 min per motion, including the motion capture.

Overall, the results are very good in a simulation. Very stable motions can be produced in a short time, but the learned motion can not directly be transferred to

a physical robot.

The complete optimization can not be done in real time using a physical robot, but we showed that the optimization in the simulator can be used as a preprocessing step, that produces a seed for the optimization on the robot. This reduces the required number of evaluations significantly and the optimization on the physical robot becomes possible.

However, the results of this approach are limited. Executing even only a hundred evaluations on the robot is still very tedious and the resulting motions are not as stable as in the simulation. The behavior of each physical NAO is slightly different from the others and optimizing the motions this way for each robot is again too time-consuming. Additionally, without balancing the motions dynamically, the results can not be robust enough considering the variances in the robots hardware and joint control and the possibility of external disturbances.

Chapter 4

Dynamically Generated Motions and Balanced Biped Walking

The results of the motion optimization in the previous chapter are very stable in the simulator they were optimized in, but the experiments on the physical robot show that a static motion is often not robust enough due to the hardware variances, loose joints or external disturbances.

Having a set of static motions is also too limiting for some types of motions. For example, a walk should be able to walk with different velocities in different directions. If there are only separate actions a walk might consist of a sequence of actions such as “walk forward”, “turn left” or “turn right“. However, this would be a very limited walk and the transition between the motions might be unstable or slow.

A better approach is to generate the motion dynamically. This also allows to react to measurements and adjust the motion to balance and compensate errors caused by hardware variances or external disturbances. One of the motions optimized in

the previous chapter lifts one leg and balances on the other. This motion could be generated using inverse kinematics and moving the supporting foot under the center of mass. When the center of mass is above the supporting foot, the other foot can be lifted. If the robot does not move too fast, it stays stable as long as the center of mass is above the supporting foot. This motion is statically stable.

Keeping the center of mass always above the supporting foot was used for walks [39, 78], but a dynamically stable walk can be much faster. However, a humanoid robot with a high degree of freedoms can have very complex physical dynamics. Calculating the exact dynamics for all body parts (more than 20 body parts for the NAO) is computationally too expensive. Therefore, usually a simplified physical model is used to predict the dynamics of the robot. Motions on humanoid robots are often generated based on an inverted pendulum model [35, 38, 19].

Since with this approach, the motion is generated online and can not be optimized directly. Instead, components of it can be optimized to improve the walk motion.

This chapter describes a new walking engine for creating a balancing, dynamic walk for the NAO with the focus on optimizing parameters of the used model from observations gathered while the robot is walking. The goal is a stable walk motion that does not require manual fine-tuning of parameters to be stable in different simulators or on different physical NAOs.

4.1 Related Work

Working on this.

4.2 LIPM-based Closed-Loop Gait Generation

This section describes the implementation of the new walking engine for a dynamically generated walk on the NAO. As a model for the motion of the center of mass of the robot, we use the linear inverted pendulum as described in section 2.3.3. A center of mass reference trajectory is generated and the steps are created according to this model. While walking the sensor measurements need to be used to adjust the estimated robot state and react to errors and disturbances.

4.2.1 The Pendulum Model and Walk State Representation

We use the linear inverted pendulum model as in Kajita et al. [35]. It assumes that the center of mass stays at a constant height h. The position and velocity of the center of mass in the plane at height h can be calculated by

$$x(t) = x_0 * cosh(k * t) + \dot{x}_0 * \frac{1}{k} * sinh(k * t) \tag{4.1}$$

$$\dot{x}(t) = x_0 * k * sinh(k * t) + \dot{x}_0 * cosh(k * t) \tag{4.2}$$

with $k = \sqrt{\frac{g}{h}}$, where g is the gravitation and $x_0, \dot{x}_0 \in \mathbb{R}^2$ are the position and velocity of the mass at $t = 0$ (see [19]).

Since this model describes the movement of the center of mass, the most important information about the state of the robot is the position of the supporting foot relative to the center of mass and the current velocity of the mass. For representing the foot position, we use a coordinate system with the center of mass as origin and the x/y-

plane aligned to the ground plane (CoM coordinate system).

The supporting foot always has to move according to the pendulum model. Assuming the models are correct, accelerating the mass this way keeps the ZMP centered in the supporting polygon (see section 2.3.1). If the foot would move differently, the mass would still be accelerated by the gravitation. If the difference is large enough the ZMP might reach the edge of the support polygon and the complete robot would rotate around that edge and falls.

Other components of the walk might need additional values (e.g. the balancing still needs the torso angles). Therefore, we represent the robots state by the positions of both feet and the torso in the CoM coordinate system and the mass velocity. We refer to these values as the walk state. When calculating the movement of the supporting foot and planning steps, the torso position in this representation can be ignored, since the foot positions are stored relative to the mass. The torso pose (torso angles and position in the CoM coordinate system) is only important for the balancing and to transform the foot positions back to positions relative to the torso for controlling the joints in the end.

For every step, we use the model to generate the movement of the mass relativ to the supporting foot. Given the duration of a step, the position of the supporting foot and the velocity can be predicted using the model. If we assume that the swing foot reaches the step target position, we can predict the complete walk state (both foot and the velocity) from the step target and duration. This information is stored in walk actions, which can represent left or right steps, but also double support phases. This representation using states and actions allows us to generate the walking motion

as a sequence of states with actions as transitions between the states.

In contrast to some other implementations, we did not derive equations for step positioning or balancing directly from the equations of the pendulum model. Instead, the model is basically used as a black box. We use the fact, that we can use the model to create the successor state for a given walk state and a step action. The step planning finds good actions through iterative methods and using the prediction. This allows us to change the model or add new parameters and offset without having to change all the other modules.

4.2.2 Reference Trajectory

The center of mass reference trajectory describes the required mass movement for a given walk velocity and step frequency. This trajectory describes the optimal movement according to the model for the current requested walk velocity. A full cycle of the walk consists of a left and a right step. If the walk speed does not change, the same two steps can be repeated.

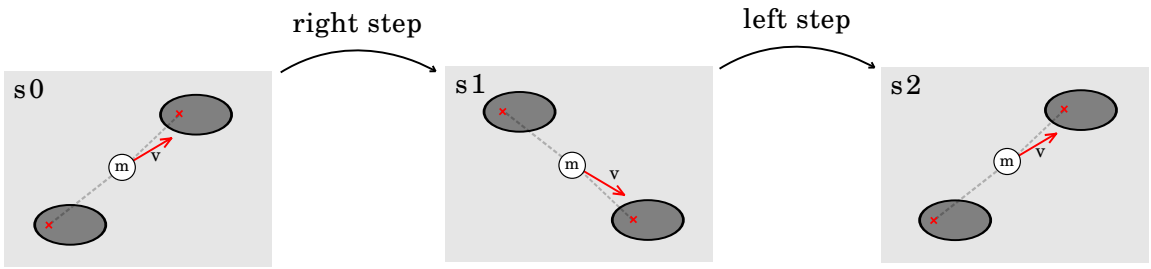


Figure 4.1: The reference trajectory can be defined by the two states when the supporting foot is changed. For the reference trajectory, v has to be the velocity that yields $s_2 = s_0$, which means that the steps of the reference trajectory can be repeated continuously and the robot maintains the requested average walk speed.

Figure 4.1 shows this cycle. The states shown are the states when a step is finished

and the support is changed. The states consist of foot positions and mass velocities. The actions define a duration and step target. This maintains the same velocity and can be repeated only if the state s_2 is equal to s_0. The mass velocity in those states depends on the walk model and is not equal to the requested walk velocity. The requested velocity is the average velocity over the complete cycle. Since the duration of the steps is given by the step frequency and the average velocity is given, we can calculate the distance that has to be covered by the two steps. Thus, the step length and the target positions for the swing legs can be calculated. Therefore, the requested walk velocity defines the walk actions in the sequence in figure 4.1.

The predicted mass position after a step and the swing target define the initial position of the next supporting foot. The only unknown values are the velocities in the support exchange states. There is only one mass velocity in s_0 that yields the same velocity in s_2 after executing the steps. If the velocity in s_0 is too high, the velocity will increase over the two steps. If the velocity is too small, it will decrease further. The velocity and foot positions in the support exchange states are sufficient to describe the complete reference trajectory for the mass.

It is possible to derive closed form equations for the velocities in the reference trajectory. However, we want to be able to change the model without updating other modules. Therefore, we use the model as a black box, such that the reference trajectory generation is independent of the exact model. This is less efficient, but more flexible. Using the predictions of the model and the known information about the steps from the requested walk request, we can use a simple iterative method to find a velocity that fits to the given steps and yields $s_0 = s_2$. The only requirement

is that the velocity error in $s2$ for a given velocity in $s0$ is a monotonic function.

Algorithm 1 Iterative method for finding the velocity in the support exchange states of the reference trajectory.

```
1: findVelocity(model, a0, a1):
2: s0.supportingFoot = a1.stepTarget
3: s0.velocity = (0, 0)
4: while vUpdate > threshold do
5:    s1 = model.predictState(s0, a0)
6:    s2 = model.predictState(s1, a1)
7:    prevError = error
8:    error = s2.velocity − s0.velocity
9:    if first iteration then
10:      vUpdate = error ∗ −0.0001
11:   else
12:      gradient = (error − prevError)/vUpdate
13:      vUpdate = −1.0 ∗ error/gradient
14:   end if
15:   s0.velocity = s0.velocity + vUpdate
16: end while
17: return s0.velocity
```

Algorithm 1 shows how to compute the velocity in the first state of the reference trajectory. The step time and step length and target positions for the requested walk speed are given as the actions $a0$ and $a1$. All other values are computed by the model. If the approximate gradient from the previous two iterations is used to update the velocity as in the Newton method, it takes only a few iterations to compute a very accurate velocity. In fact, without any modifications the velocity error is linear with respect to the changes in the velocity of $s0$, such that two iterations are needed to compute a gradient and the third iteration directly finds a good velocity. Only if the steps also include a a foot rotation for turning, another iteration is needed. With 4 to 5 iterations this method finds a velocity for $s0$ that produces a velocity error in $s2$ of less than 1 mm/s.

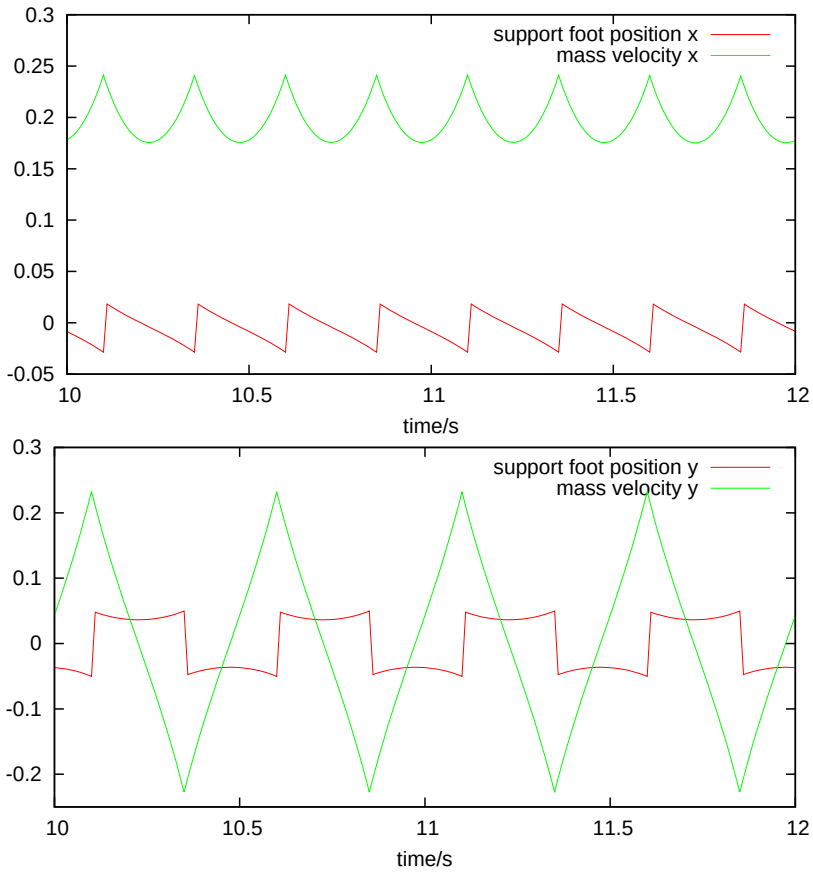


Figure 4.2: Example reference trajectories for walking forward with 0.2 m/s (in x and y direction). The position of the support foot is relative to the center of mass of the robot.

Figure 4.2 shows the result reference trajectories in x and y direction for walking forward with 0.2 m/s. This is generated by repeatedly executing the same two steps of the reference trajectory starting with the correct initial velocity.

4.2.3 Step Planning

The behavior of the robot can request different walk directions and velocities. The walking engine has to make sure that this velocity is reached as fast as possible without falling.

The previous walking engines used by the RoboCanes agent did not always gener-

ate stable motions. A sudden acceleration could make the robot fall. Additional limits on the acceleration had to be added, but they were chosen manually and probably limited the walking speed and acceleration more than necessary.

A better approach is to let the walking engine control the acceleration. The maximum acceleration depends on the physical behavior of the robot, which is modelled e.g. by the pendulum model. The walk should be generated based on the physical behavior of the model and guarantee a feasible acceleration. The maximum speed should only depend on the maximum step length.

The task of the step planning is to choose the current step, such that the robot can reach the reference trajectory. The duration and the swing target of the current step are chosen such that the predicted state after the next action is as close as possible to the support exchange state defined in the reference trajectory. The step target we choose is similar to the capture step foot placement strategies used e.g. in Missura and Behnke [51, 52].

We determine the step values similar to the calculation of the velocities in the reference trajectory iteratively using the physical model as a black box. This might not be the most efficient solution, but it is the most flexible solution for changing the model and again only a few iterations are needed.

Due to errors in the joint control or external disturbances, it can be necessary to adjust the step time and step target position for the current step. The step planning updates these values in every control cycle (e.g. with 100 Hz on the physical NAO). First, the remaining step time is calculated such that the lateral velocity at the end of the step is close to the velocity defined in the reference trajectory. Using this

time and the current position of the support foot, the mass position at the end of the step can be predicted using the model. The position of the supporting foot that is used beginning from this predicted state, depends on the step target for the current step. The current steps target position changes the outcome of the next step. Therefore, the step target position is chosen (again approximated numerically), such that the predicted state after that is close to the ideal support exchange state from the reference trajectory.

If the requested walking speed changes rapidly, the reference trajectory will immediately change to the new trajectory. However, the step planning will choose feasible steps that will not make the robot fall. The walk velocity is changed according to the model.

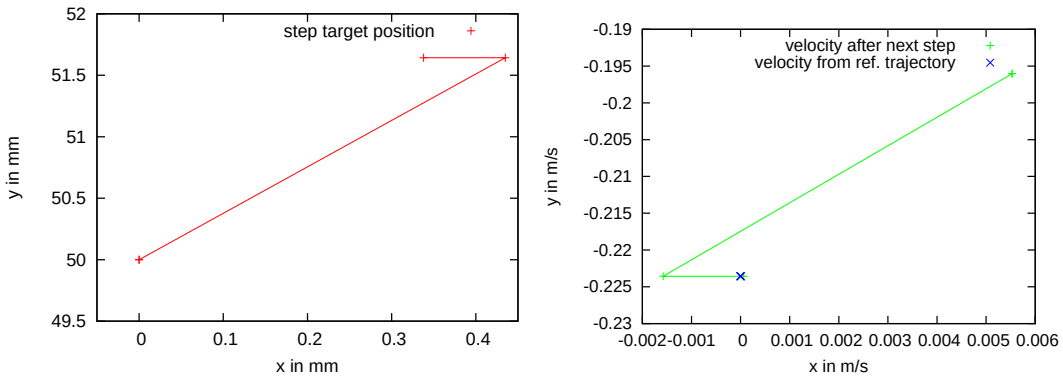


Figure 4.3: Step target positions and resulting velocities visited to find a good step target. The step target positions (left) produce the predicted velocities (right) after the next step.

Figure 4.3 shows an example for positions visited by the step planning to find a good target position. The robot was walking on a spot. It starts from the ideal step target defined by the reference trajectory ($x = 0m$, $y = 0.05m$). For the current estimated state this will result in a small forward velocity (positive x) and a too

small velocity to the right (right: negative y). The ideal velocity from the reference trajectory is $(0, -0.223)$ (the blue cross). Therefore the step planning adjusts the target position of the current step in two more iterations until the predicted velocity is close to the ideal velocity. Similar to the generation of the reference trajectory, this search needs only a few (most of the time 3) iterations to find the target position for the current step. This search is repeated every cycle to update the step target (needed when the current estimated state is changed by feedback in the closed loop walk).

By using different criteria as the error for setting the step target position, this search can be used in different situations, such as walking, standing or the transition from standing to walking.

Each robot state is either a "left support", "right support" or "double support" state. While walking only the left and right support states are used and the step time and target positions are chosen as described. When the robot is standing, it is in a double support state. In this case, the walk actions only define a center of pressure within the support polygon instead of a step target. In this state the robot moves according to the pendulum, too. By using the model also when the robot is standing, it is possible to create a smooth transition between standing and walking by choosing the right center of pressure in double support to reach the right initial velocity before the first step. It is not necessary to start stepping on a spot and then slowly accelerate.

Similarly, the model can be used when the robot stops walking to decide whether it is safe to stop and choose the parameters of the last step such that the remaining

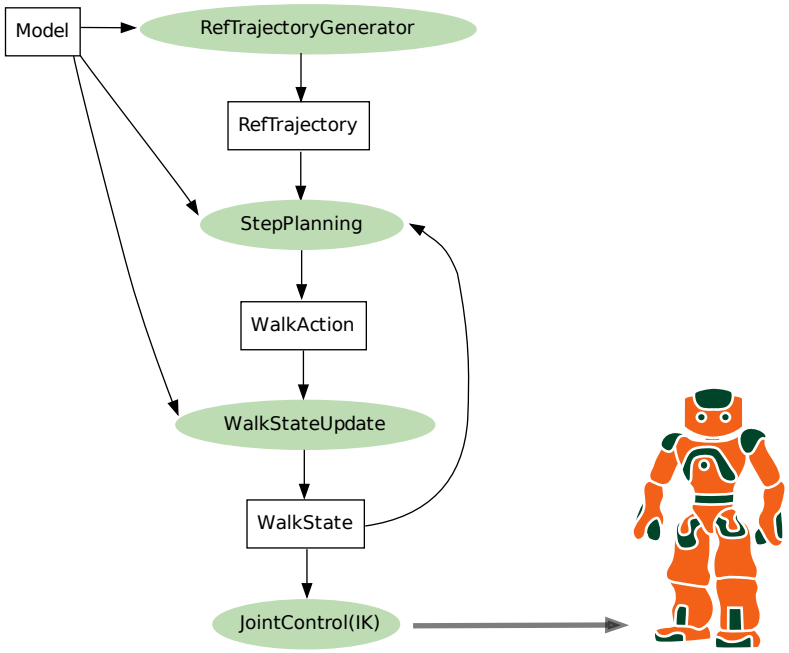


Figure 4.4: The modules and information passed between them for an open-loop walking engine. The white boxes represent information passed. The round objects are modules (see A.2.3).

mass velocity is small when the double support begins.

4.2.4 Open-Loop Walk

In addition to the reference trajectory and the step planning, only a simple module for predicting the state in the next timestep is needed for an open-loop walking engine.

Figure 4.4 shows the components described so far and how they can be used to create an open-loop walk. The state of the robot is only an internal state and is updated every time step using the model and the chosen action. No feedback from the robot's sensors is used. This can only generate a stable walk, if the robot's actual motion is similar enough to the model and there are no external disturbances.

This open-loop walk is already stable in the simulation since there are no distur-

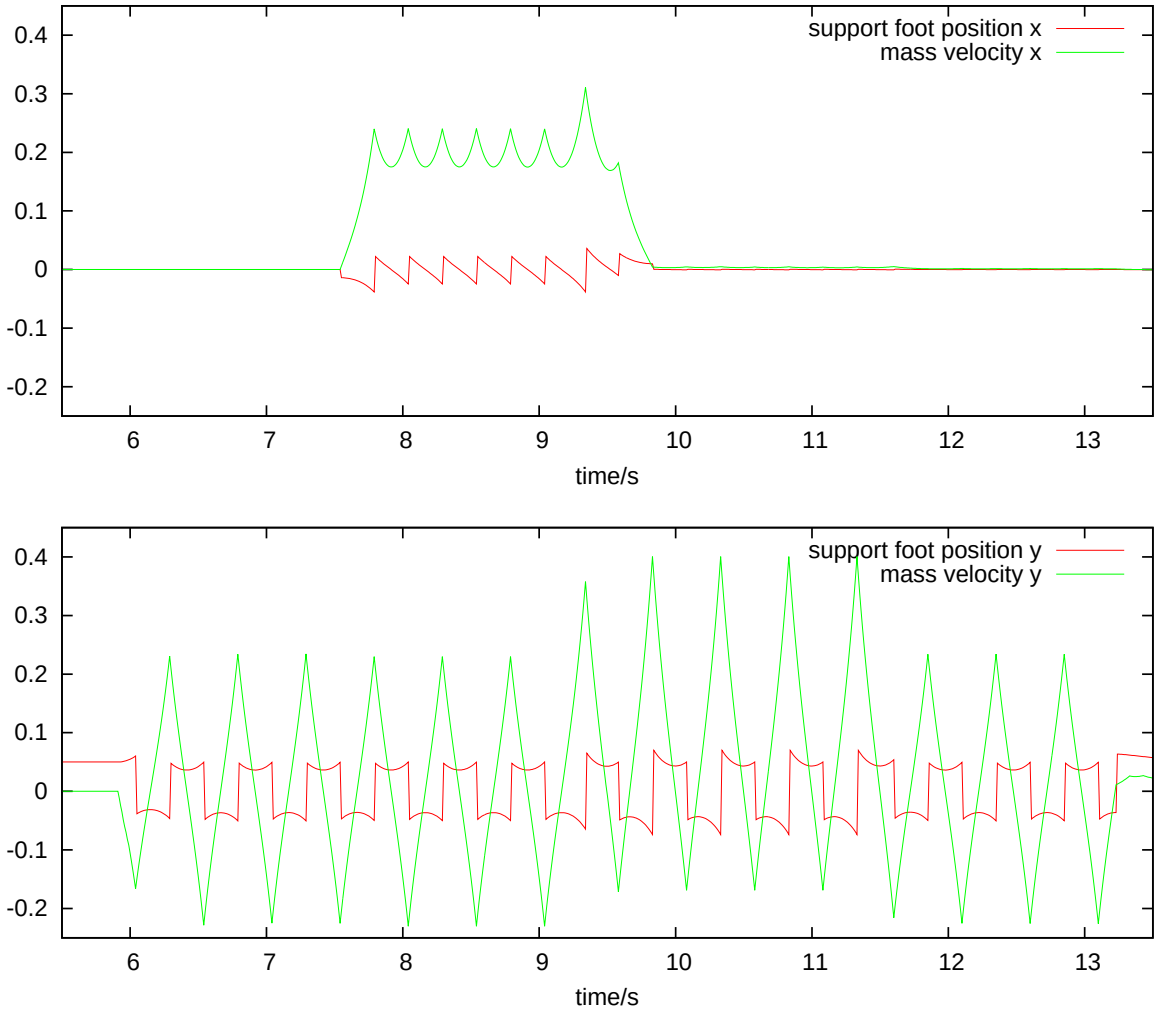


Figure 4.5: Test walk generated by the open loop walking engine that starts with speed 0, walks 0.2 m/s forward, then 0.1 m/s to the side, slows down to speed 0 and stops.

bances, the robot model is known exactly and the robot is walking on a perfect plane. Figure 4.5 shows a short test walk. Figure 4.6 also shows the prediction for the state at the end of the current step. Since the motion is generated only using the model, the prediction is perfect and the predicted state is reached after every step.

For the same model parameters, this walk will always be generated exactly the same way for the requested walk velocities. On a physical NAO the motion would be the same. If the robot's actual movement starts to differ from the model, the motion

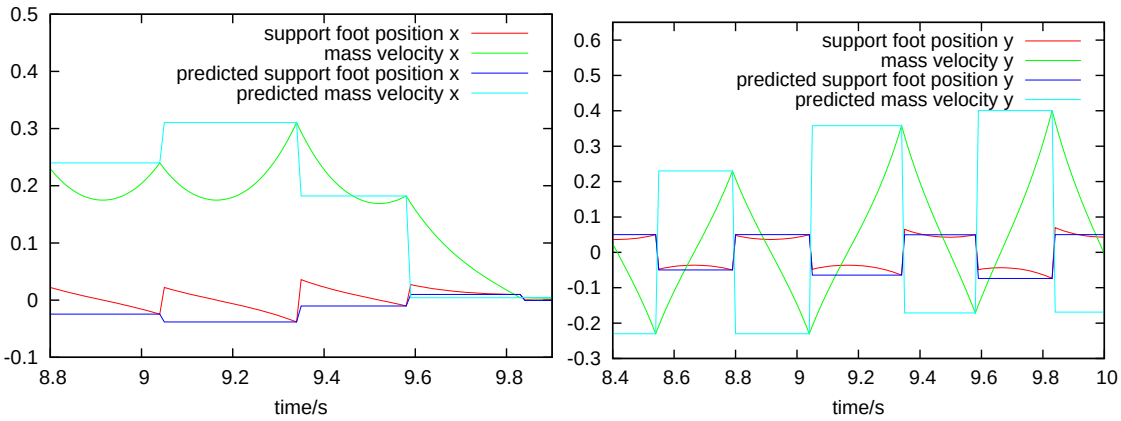


Figure 4.6: Foot positions and velocities each with predicted values for the end of the current step. Since the open loop walk follows exactly the model, there is no error in the predictions.

will just continue, the errors might increase and cause the robot to fall.

There are too many variances and disturbances on a physical robot for an open-loop walk to be stable. Fine-tuning the model parameters can make the motion stable under good circumstances, but as with the optimized static motions from chapter 3, the resulting motion will not be robust.

Errors in the motion have to be detected and the motion needs to change to compensate for the errors. Nevertheless, the better the model fits to the actual physical behavior of the robot, the smaller are the errors and only smaller adjustments in the steps are needed.

4.2.5 Closed-Loop Walk

The walk can only be robust if it reacts to unexpected errors in the motion. These can be caused by external disturbances, inaccurate joint control or also by the errors in the model.

The foot positions relative to the mass can directly be perceived by the measured

joint angles combined with the forward kinematics of the robot. The rotations of the torso can be measured using the gyroscope and the torso angle estimated using the gyroscope and accelerometer.

The velocity can not directly be measured and has to be estimated from other sensor values. The accelerometer can be very noisy and small errors in the torso angle can cause problems with distinguishing accelerations on the x/y plane from gravity.

Therefore, we do not integrate the accelerometer values to retrieve the current torso velocity. Instead, we use a particle filter [85] to estimate the velocity using the observed movement of the foot, the measured torso rotations and the pendulum model. The point of observing the state is to detect differences to the expected movement. That means differences to the model are expected and we need to use high noise values in the state estimation.

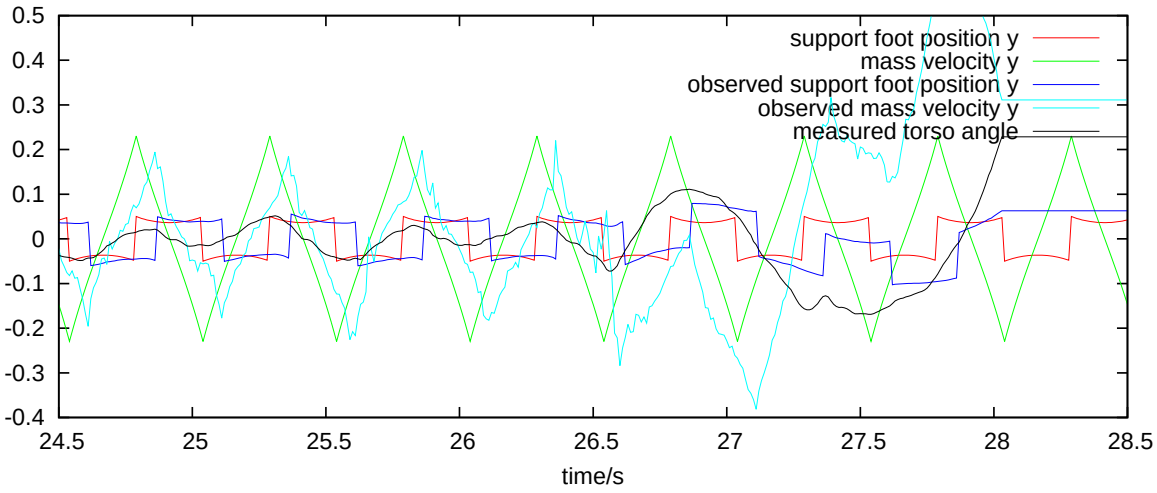


Figure 4.7: Motion generated by the open loop walk (speed 0) and observed state when the motion is executed on a physical NAO.

The estimated velocity, the measured foot positions and estimated torso angles are used as observed walk state. Figure 4.7 shows the internal state from the open-loop

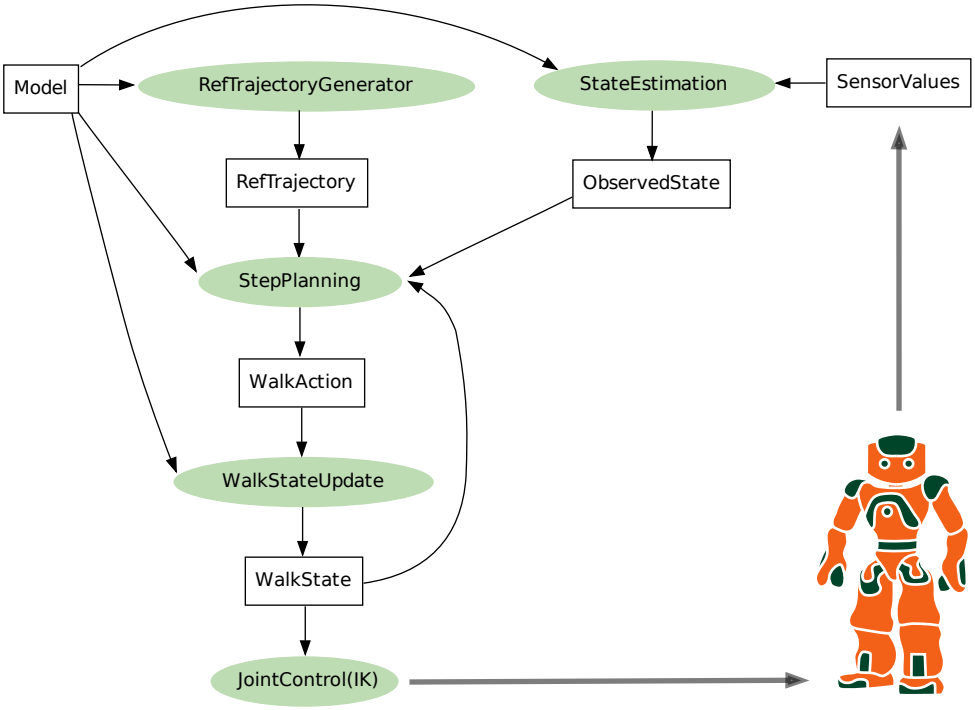


Figure 4.8: Components of the closed-loop walk.

walk and the estimated velocity and foot positions from the observed state when the motion is executed on a physical NAO. Without disturbances the motion is stable (speed 0, only walking in one position). At the end the NAO is pushed a little. Once the motion of the NAO is not synchronized to the walk motion, the errors increase and the robot falls.

The observed state can be noisy and directly using it as current state in the step planning would transfer this noise into the control of the joints. Responding too much to the observed state can also cause oscillations, e.g. if the steps are always a little too long, after one long step, the next step will be longer in the opposite direction, causing the step after that to be even longer in the first direction, and so on.

Additionally, there is a delay in the control loop of several cycles, which can also cause oscillations when the observed state is used directly. Using the model we predict

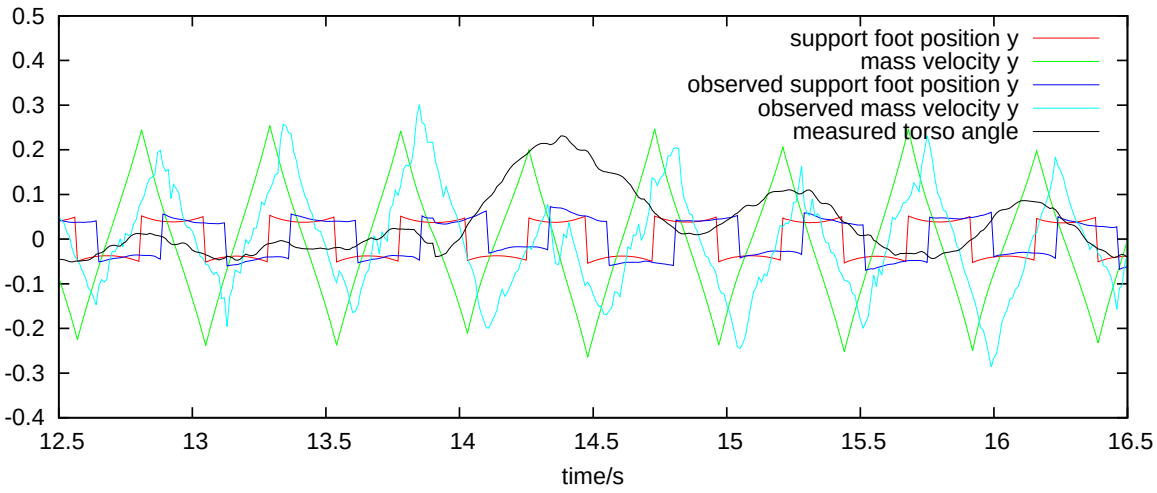


Figure 4.9: The support foot positions and velocities in y direction (lateral) using the closed loop walk. The support foot position and velocity set by the walk are now adjusted slightly, if the observed state differs from the expected values.

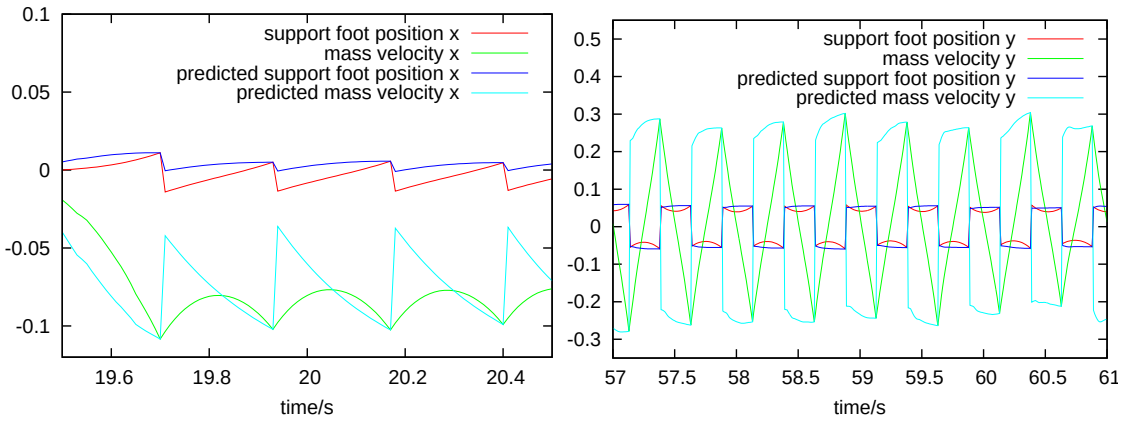


Figure 4.10: Foot positions and velocities each with predicted values for the end of the current step. There is a bias in x direction and the velocity in y direction is underestimated.

the state after the delay time from the observed state. This predicted state is then used to only slowly change the current estimated walk state which is used for the step planning and to move the feet.

Figure 4.8 shows the modules of the closed-loop walk. This walking engine can now produce a stable walk also on the physical NAO. The plot in figure 4.9 shows a stable walk, although the values do not reach the values of the reference trajectory.

Nevertheless, the error is not increasing and the walk is stable. Towards the end of the plot the robot was pushed from the side and is able to recover from that.

Figure 4.10 shows the predicted values that are expected to be reached at the end of the current step. The error in the prediction is small, but there are errors in every step and the predicted state at the end of each step (each walk action) change over time. During the experiment, the robot was leaning back slightly which caused it to also walk slowly backwards. The changes in the predicted positions will also change the step target position for the current action while the swing leg is already moving. The higher the error in the prediction, the higher is the error in the position reached by the swing leg. This can again cause errors and will prevent the robot from following the reference trajectory.

4.2.6 Joint Hardness

The feet never reach exactly the correct target position at the end of a step. There are errors in the control of the leg, the kinematic model (not perfectly calibrated joints) and the target position can change due to disturbances which update the estimated walk state and change the step target. All errors caused by model inaccuracies and measurement delays effect the stability of the walk, because the movement of the mass is not exactly the required movement to keep the ZMP in the center of the supporting polygon.

This might not be a problem for small errors, since they will only cause a small change in the center of pressure (or ZMP).

NAOqi, the software controlling the joints of the NAO, allows to set a target angle and a hardness value for each joint. The hardness value limits the maximum current used for that joint.

If all joints are controlled with a high stiffness/hardness, small errors can simply be ignored. They only move the center of pressure in the foot. As long as the errors are small and the center of pressure stays within the supporting polygon the walk is stable and the mass acceleration can be executed.

On the other hand, the high stiffness can stress the joints more and the errors are difficult to measure. Using a high stiffness the errors are basically hidden. The joints are forced to move in a certain way and only the center of pressure is moved slightly, which is are difficult to observe. For example, measuring the center of pressure can be difficult, since the force sensors in the feet of the NAO are noisy and the measurements can vary depending on the ground surface.

The experiments on the physical robot in this section were done using a lower hardness in the ankles and hip. A lower joint hardness does not only protect the joints. It allows us to observe if an acceleration was wrong. If the center of pressure moves, the lower stiffness allows the foot and torso to turn slightly, which is easier to measure. These angle changes the observed robot state, which will be used to adjust the walk and reduce the errors. However, this means that the walk has to be more responsive to the torso angle and it requires a better control of the torso angle that keeps the torso upright.

4.2.7 Torso Angle Control

The walk state described in 4.2.1 contains the foot positions relative to the mass using the orientation of the world coordinate system. Most modules of the walking engine do not use the torso angle. The torso angle is used in the transformation from the measured foot positions relative to the torso into the walk state and for the transformation back from the updated walk state to the foot positions used to calculate joint angles using inverse kinematics.

Since the step movements are created in the walk state reference frame which is aligned to the ground, the feet do not hit the ground when the torso is rotated slightly. Nevertheless, the torso rotations have an effect on the step planning, since the foot positions relative to the center of mass in the walk state depend on the torso angle. A rotation of the torso changes the observed walk state, which changes the current walk state used for the step planning.

An additional controller is needed to turn the torso back and keep it in an upright position. The torso can be rotated back by slowly turning the coordinate system used to generate the foot movements. However, that can quickly result in hitting the ground with the swing leg if the torso does not rotate as expected.

Therefore we added a controller that is completely independent from the other modules of the walking engine. The torso should be turned by a torque applied by the supporting leg. We use a simple PD controller that modifies the angles of the hip joints of the supporting leg. This controller is executed after the angles for the legs are calculated using the inverse kinematics. If this hip controller starts turning the

torso, the walking engine will measure the torso rotation and update the observed walk state accordingly to position the feet using the updated torso orientation.

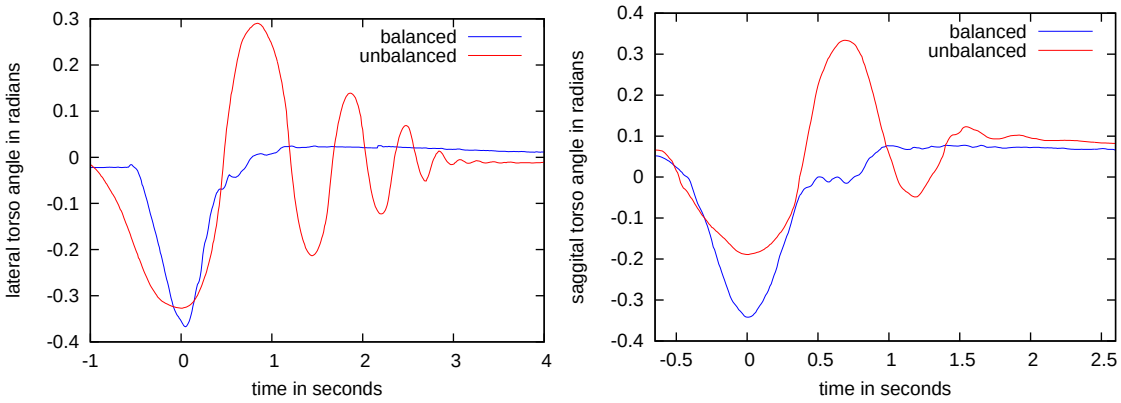


Figure 4.11: The torso angles of a standing NAO that is tilted to the side (left) or back (right) and is released.

The same torso angle control is used when the robot is standing and the walking engine uses the measured torso angle to balance. Figure 4.11 shows the measured torso angle of a standing physical NAO when it is tilted around the x-axis (pushed from side, lateral) and the y-axis (pushed from front, saggital) and falls back. If the joints positioned at fixed angles for the stand and stiff, the robot oscillates for seconds because the robot stands only on the edge of a foot and builds up a high angular momentum.

The balanced stand uses the closed-loop walking engine in the double support state and the hip controller. This stabilizes much faster than the robot with stiff joints. The oscillations are avoided mainly because the measured torso angle directly causes the robot to adjust the foot positions. Unlike the unbalanced robot, it does not fall back on the feet, since the feet are always aligned with the ground and it never stands only on the edge of a foot. The balancing robot can also be pushed back

further, because the feet are positioned depending on the measured center of mass. The stiff robot would fall back for larger angles. The fact that the robot is balancing and uses the measured torso angle already avoids the oscillations. The hip controller only moves the torso back to an upright pose.

4.2.8 Challenges on Physical Robots

This section gives some examples of problems with the physical robots that do not exist in the simulation. There are many obvious differences between the simulation and the physical robot (higher noise, errors and delays), however even seemingly small errors or different movements can cause unexpected problems.

IMU and Feet Angles

Since we use the estimated torso angle is used to generate foot movements in a coordinate system aligned to the ground and to set the ankle angles, a small difference between the expected foot angle and the measured angle can have a big effect.

If a physical robot sets the angle of the ankle joint such that the torso angle should be 0, the measured angle from the IMU is usually not 0 (see table 4.1). If the robot then uses the measured torso angle from the IMU to set the foot angle for the current torso pose, it will increase that torso angle, such that the IMU measure an even higher angle in the next step. This angle adds up very quickly. Directly using the measured torso angle from the IMU for the foot angle will make the NAO rotate the torso until the foot positions are not reachable any more in less than a second. A reactive torso angle control can keep the angle small, but it constantly has to compensate for

Robot	Standing	Walking
SimSpark type 0	0	-0.0001
Webots NAO	0.002	0.007
physical NAO 1	-0.08	-0.075
physical NAO 5	-0.06	-0.025
physical NAO 6	-0.003	0.03

Table 4.1: Average differences between the measured torso angle (in saggital plane) from the IMU and the expected torso angle from the leg joints of the supporting legs while standing and walking with speed 0 (angles in radians).

that error to avoid this wrong rotation. This could result in different behaviors when the robot is leaning in different directions. The control might be "softer" towards the direction of this wrong rotation. The hip controller that tries to keep the torso in an upright pose, constantly pushes the hip in one direction which can cause the balancing to start slowly forward or backwards.

We measure both torso angles, the angle from the IMU and the expected angle for the current joint positions, and use the average difference to reduce the effects of these errors. Table 4.1 shows the average torso angle measured using the IMU on different robots while standing and walking on a spot. This is only one example showing the difficulties on a physical robot when the balancing is too responsive to sensor values. Using a high hardness to move the legs and assuming a torso angle of 0 hides some problems, but it also less responsive to measurements.

In simulators as SimSpark or Webots this error is much smaller and causes less problems, since the simulated robot is perfectly calibrated and walks on a perfect plane.

On physical robots, these errors vary and there might be more unexpected errors. Therefore the robot should detect problems and adjust the motion.

Knee Currents

The used current and resulting joint temperatures are an important aspect of motions on physical robots, while it is not of any interest in simulations as the RoboCup 3D Soccer Simulation League.

For the new walking engine and also the previously used B-Human walking engine, the knee joints use the most current. However, for most robots one knee uses more current than the other. Reducing the current or at least balancing the knee currents is important to avoid overheating and damaging the joints.

However, the current can not directly be controlled (only target angle and hardness can be send to NaoQi) and several experiments have shown, that simply leaning to one side or shifting the mass does not change how much current is used in the knee. Also the speed or height of steps and walk direction only have a minor effect on the current.

Instead, the movement of the supporting leg has a significant effect on the current. Especially, changing the height of the torso can cause high currents, even if the reason for changing the torso height is keeping the center of mass height constant. Small changes in the torso angle and how the foot hits the ground when it becomes the new supporting foot also have a large influence on the current.

To reduce the torque in the hip joints, we intentionally tilt the torso towards the supporting leg while walking. However, these small rotations change the height of the hips slightly if the center of mass stays at a constant height. If the robot is leaning to the left, the left hip is moved down and the right hip up. When the torso

rotation starts moving towards the other side, the left hip will have to be moved up. Although the height of the center of mass is constant (according to the robot model), this movement of the hips causes high currents. The current is lower, if the hip height of the supporting leg is not changed by the torso rotations even if that changes the height of the center of mass slightly. Since the height of the center of mass is increase while leaning to the side, the overall mass motion is closer to the oscillations of the stiff robot in section 4.2.7 which might be the explanation for the improved energy efficiency.

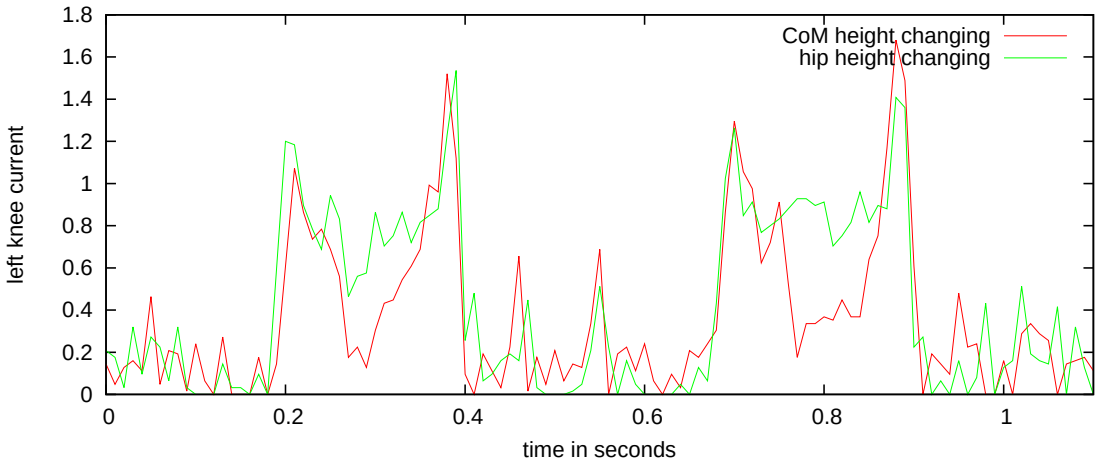


Figure 4.12: Example for measured knee currents during the walk. There is a lot of noise and the current varies a lot, but overall the supporting leg uses less current when the hip height is kept constant. There are similar peaks in the currents in the beginning and at the end of each step, but changing the hip height produces a more constant high current while the leg is supporting.

Figure 4.12 shows the currents used by the left knee joint while walking. There is a 1.5 mm oscillation either in the height if the supporting side of the hip or the center of mass caused small torso angles to the sides. The experiments on the physical robot have shown that the average knee current is about 15 to 20% lower if the hip height is constant. However, there are many factors that change the used current.

For example, small differences in the calibration of the joints on each robot can make a difference.

There are many similar situations were small changes in the movement have an unexpected effect on the stability or the joint currents. Automatically optimizing the walk motion for each individual robot should help to reduce these unpredictable effects on physical robots.

4.3 Walk Optimization

In a closed-loop walk the measured state of the robot is used to react to errors caused for example by model errors, inaccurate joint control or external disturbances. However, any change in the step planning for balancing can cause the robot to not walk the way it is supposed to. Therefore, the smaller the errors are, the more controlled is the walk.

If the observed walk state always differs from the expected behavior of the model, the model might be wrong.

A very common problem is a robot that leans to the back and constantly walks slowly backwards to balance. If the robot is supposed to walk slowly forward, it might not move forward at all. In a RoboCup environment this can prevent the robot from reaching the ball if the robots behavior tries to approach the ball slowly.

Using the observed walk state, we will optimize the parameters of the model to improve the prediction and change how the motion is generated. Since most components of the walk use the model, changing the model parameters will change

the complete motion (the step planning and the movement of the supporting foot).

4.3.1 Model Parameters

The movement of the inverted pendulum only depends on its length. Theoretical, the height of the center of mass of the NAO can be measured, but there are errors. A shorter or longer pendulum might model the motion of the robot better, since the exact center of mass position is not known. Similarly, an error in the center of mass in x and y direction might cause errors in the predictions. The predictions will always be noisy, but a bias should be avoided. For example, if the predicted forward velocity is always overestimated, each step chosen according to the model results in a lower velocity and the robot can not reach the desired walk speed. In this case the model should be changed to reduce this error.

The basic pendulum model will not be changed by the optimization, but we add parameters to the mapping from the estimated state of the robot to the model. We add offsets for the center of mass position in x, y and z direction, offsets for the estimated velocity, factors to adjust the measured foot positions in x direction and a time scaling factor.

Some of these parameters might not be physically reasonable and maybe some values will not help improving the predictions. However, the point of using a black box parameter optimization to fit the predictions to observations is to improve the model without knowing exactly what is causing the errors and which parameters need to be changed.

4.3.2 Model Optimization

The estimated state of the robot (positions of feet and center of mass) is updated every cycle using the model and measurements from the sensors. In the beginning of each step, the model can be used to predict the trajectory of the center of mass using the current supporting foot as origin.

After observing several steps, this data can be used to test how much the predicted movement using only the state from the beginning of each step differs from the observed movement (in position and velocity). These errors have to be minimized by changing the model parameters.

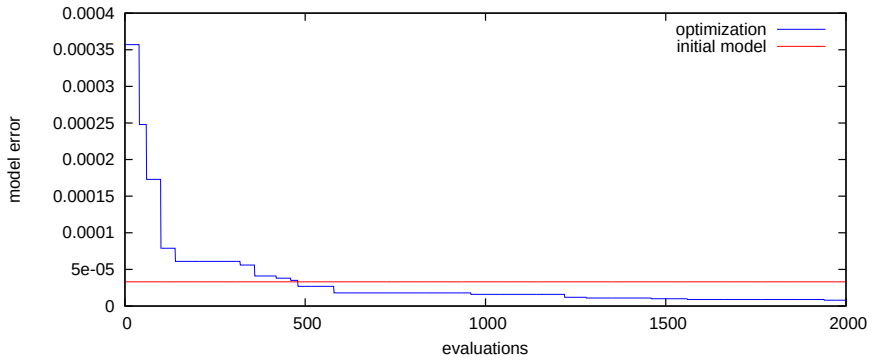


Figure 4.13: The minimum error during the model optimization using data from a NAO in Webots. This error is calculated by comparing the observed positions and velocities with the predicted motion for several steps.

After collecting data from several steps the optimization can be done offline. Each candidate solution (parameter set) is evaluated by calculating several positions on the trajectories using the model and comparing the result positions and velocities to the observed values to calculate an overall mean squared error for the used parameters. This optimization can run on a physical robot in the background with low priority. After a constant amount of iterations it stops and the robot can use the new

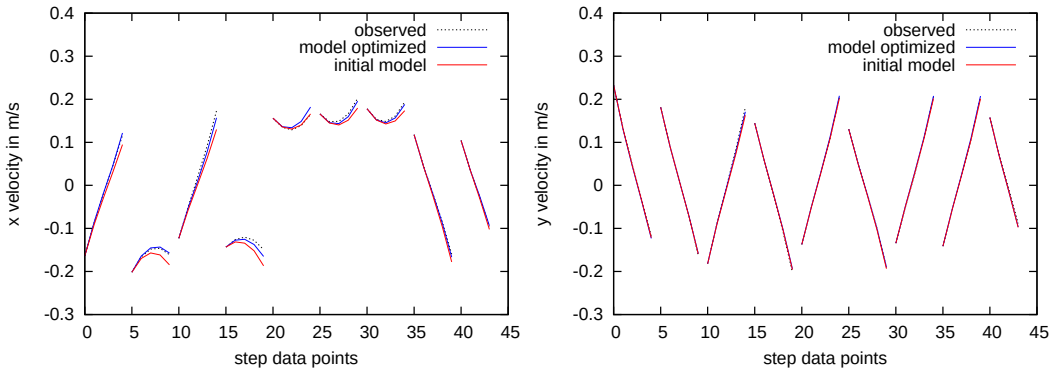


Figure 4.14: The velocities of the observed steps used for the optimization in Fig. 4.13.

parameters.

Figure 4.13 shows an example run of this optimization using data from simulated robot in Webots. The 10 model parameters were optimized using CMA-ES with a population size of 20. Figure 4.14 shows the velocities of the step data and the values predicted by the model before and after the optimization.

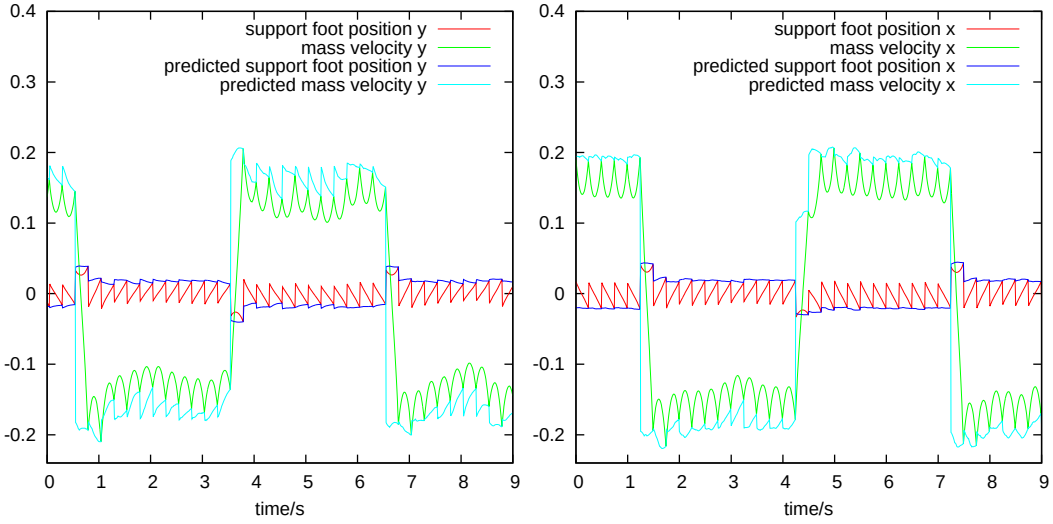


Figure 4.15: Examples for the estimated walk state and the predicted positions and velocities in Webots before (left) and after the optimization (right). The requested walk speeds are -0.15 m/s and 0.15 m/s. There is only a small difference, but the walk speed is a little more constant using the optimized values.

Since the optimization adds some random noise in the beginning, the optimiza-

tion starts with higher errors, but it quickly finds parameters that are better than the model without any modifications. Although the error is already small without modifying any model parameters, the small improvement by adjusting the parameters is noticable in the predictions and improves the step planning slightly. Figure 4.15 shows the predictions in webots for walking forward and backwards with 0.15 m/s with and without the optimization. The predictions are already quite accurate without the optimization, but the small change in the parameters causes the robot to walk with a more constant 0.15 m/s. Without the optimization, the velocity is predicted to high causing the average walk speed to be less than the desired 0.15 m/s.

Since the steps that are used as observations in the optimization vary a lot, the error is not always reduced as in figure 4.13. Sometimes the error is higher and can not be reduced much, but the error is never higher than the error for not changing the parameters. If the errors are already small and adjusting the parameters does not improve the predictions, it will stay close to neutral values (offsets set to 0, scaling factors set to 1).

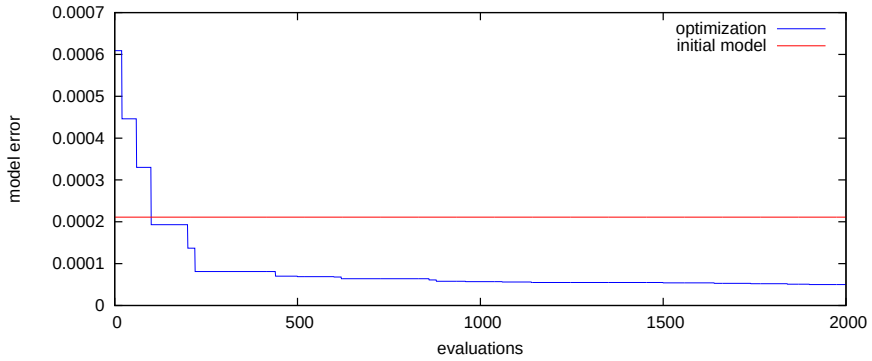


Figure 4.16: An example run for the model optimization on a physical NAO.

Figure 4.16 shows the error and figure 4.17 the step trajectories and predictions

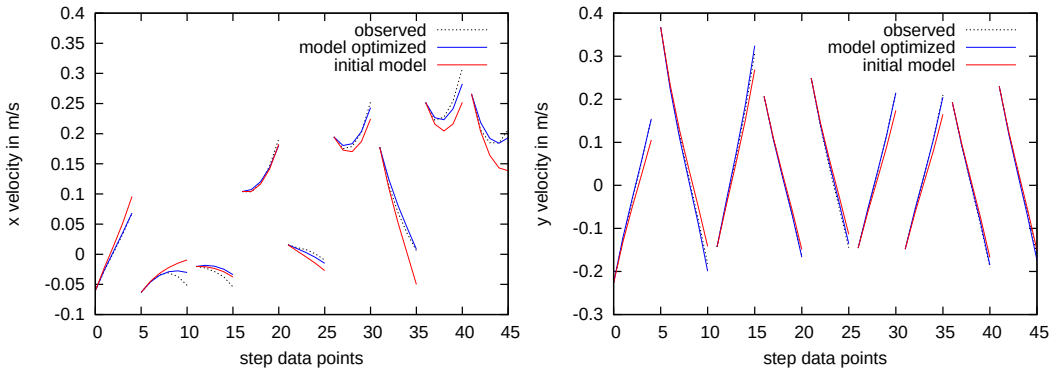


Figure 4.17: The velocities of the observed steps from the physical NAO used for the optimization in Fig. 4.16.

Parameter	Neutral parameters	Webots	physical NAO
mass offset x	0	-0.0014	0.0016
mass offset y	0	0.0001	0.0012
mass offset z (for x movement)	0	0.0046	0.015
mass offset z (for y movement)	0	-0.0029	-0.0154
velocity offset x	0	0.0026	-0.0144
velocity offset y	0	-0.0017	0.0008
foot position factor x	1	0.9938	0.9382
foot position factor y	1	0.9785	1.0578
foot position factor	1	0.981	1.0218
time factor	1	0.9989	0.9991

Table 4.2: The parameters found by the optimization in Webots and on a physical NAO.

for the same walk and optimization on a physical robot. The errors are not much larger than in Webots. This is only one run and the error is different depending on the observed steps, but in most cases the optimization is able to reduce the prediction error for the observed steps significantly on the physical NAO.

Table 4.2 shows the parameter values found in this two experiments the different environments produce different adjustments in the model parameters. This would be very tedious/impossible to do manually for multiple robots. The values also show, that Webots does not benefit much from these parameters, since the values from Webots stay close to the neutral values, that do not modify the model.

The optimization reduces the model error for only a few specific observed steps. More experiments are needed to verify that the adjusted parameters improve the walk in general.

4.3.3 Experiments

We conducted several experiments using different physical and simulated robots to evaluate the effect of the optimization on the walk.

The experiments executed using two different simulators and different physical NAOs (Nao1-Nao6). Five of the physical NAOs are the model NAO V4 H21. Nao6 is the model V4 H25 with additional wrist and hand motors.

The simulator Webots contains a NAO H21 model, which is very similar to the physical NAOs. The simulator SimSpark uses a robot model similar to the NAO, but with slightly different dimensions and masses. Additionally, it is possible to modify some parameters of the robot model (e.g. longer legs, wider hips). In our experiments, we use the robot types used by the RoboCup 3D Soccer Simulation League.

The optimization of the model parameters runs for 20 seconds, then gathers new steps data and restarts the optimization. In the experiments using physical robots, it directly run on the robot.

We compare some results in this chapter to the previously used walking engine which is based in the walk of the RoboCup team B-Human and described in [19]. The B-Human walk is a very stable closed-loop walk based on the inverted pendulum model and has been used by several RoboCup teams since it was published as part

of the B-Human code releases (e.g. [68]).

We integrated the B-Human walking engine into the RoboCanes agent. However, the B-Human walking engine depends highly on an accurate calibration of the NAO and a very accurate estimation of the torso angles. If some errors are too high, it does not walk at all. There are also almost 100 parameters (offsets, limits, thresholds) that can be set manually. Some parameters are easy to set or can always be 0, but many parameters are important, e.g. the maximum speed and acceleration. We found a configuration that is very stable, but the acceleration is limited and the robots can not react quick enough in many situations. This walking engine could probably be faster and stable, but we did not fine-tune all parameters. Nevertheless, this walk was used by the team RoboCanes in several RoboCup competitions (1st place at US Open 2015) and will be used in some comparisons.

Walking on a spot

In the first experiment, the robot walks on a spot (the requested speed is 0) for several minutes. The lateral movement is most important in this experiment, which depends on the step timing and selection of step target positions. The robot should make steps such that it maintains a stable oscillation from left to right without much variance.

The model parameters are optimized using information from only a few steps. If better parameters are found, the walking engine directly starts using these parameters, which decreases the average prediction error of the model. Table 4.3 shows the prediction errors in different environments with and without the optimization. Additionally, it shows the average of adjustments in the step targets during each step

		mean x	pred $\dot{x}$	err y	$\dot{y}$	mean x	step change y	support mean	foot err avg. abs. error
Webots	NAO	1.6	-10.3	1.0	-6.0	0.77	0.38	1.2	2.5
		0.9	-7.8	0.3	-0.14	0.47	0.28	0.5	2.6
SimSpark	type0	-0.03	0.2	0.1	0.3	-0.03	-0.02	0.1	4.5
		0.2	-0.4	0.3	-1.6	0.06	-0.4	0.5	4.7
physical	Nao2	1.9	-11.6	5.8	-38.7	0.68	3.94	8.3	8.9
		3.0	-23.9	2.1	0.9	1.9	0.31	3.2	5.8
	Nao6	5.8	-40.6	3.1	-20.3	2.83	2.29	5.5	8.4
		1.8	-23	0.3	12.1	1.08	-0.73	3.2	6.7

Table 4.3: Observed values while walking on a spot with different robots. The second row for each robot contains the results using the optimization. All values are averages over 5 minutes walking ($\approx$ 1200 steps).

and distance values of the supporting foot to the desired position from the reference trajectory.

The better the predictions for the movement of the mass are, the better are the step target positions that are chosen in the beginning of each step and the position does not have to be updated much during the step.

Since the robot did not walk forward or backward, the observed steps do not contain much movement along the x axis and therefore does not improve the predcitions in this direction. In some cases the errors increase, because the optimization overfits the model to small x values in the observed steps. However, these results show, that the optimization is able to reduces the prediction errors in y direction (lateral) in all experiments. This results in smaller adjustments in the step targets during the steps and the movement is closer to the reference trajectory.

The "support foot error" is calculated at every support exchange by comparing the position of the previous support foot to the reference trajectory. The table shows the distance of the mean position over all steps to the reference position and the average distance for the individual steps. There will always be variance, but the mean can be

close to the planned position. If there is a bias, the robot might walk slowly although the requested speed is 0.

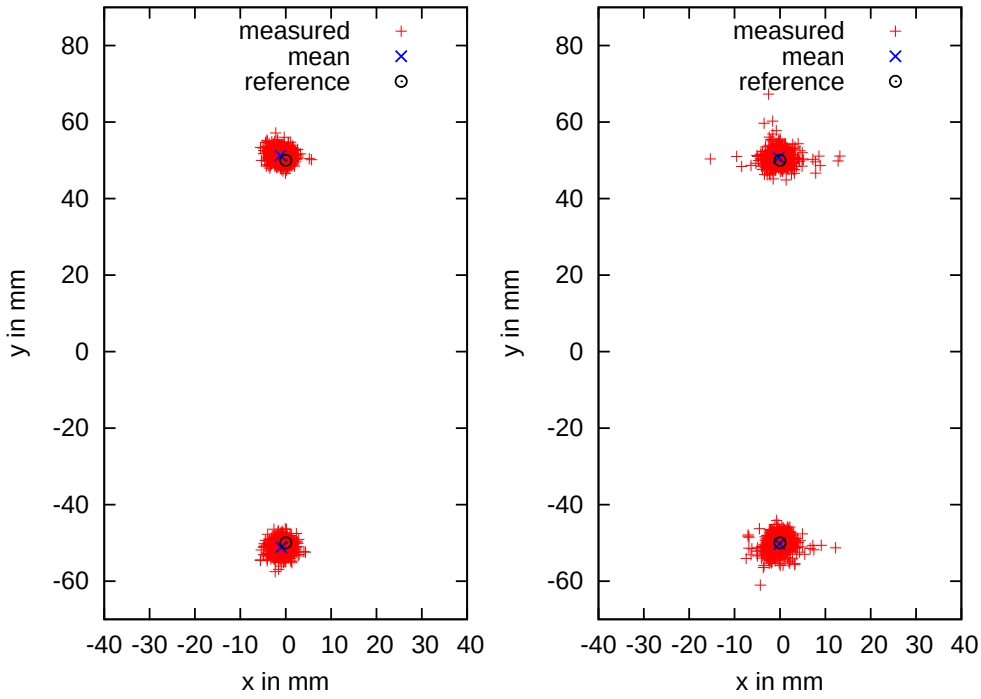


Figure 4.18: The positions of the support foot at the end of each step in Webots without (left) and with the optimization (right). The variance is not reduced, but the mean of the measured positions moves even closer to the position from the reference trajectory.

If the robot model would perfect, the robot would be able to choose each step such that the robot's state at the end of each step is the state defined by the reference trajectory. Figure 4.18 shows the variance in the support foot positions at the end of the steps made in Webots. There is only a small variance in Webots, but there is a small bias. Using the model optimization, the mean of the observed positions moves closer to the position defined by the reference trajectory.

Figure 4.19 shows similar results for a physical NAO. Without the optimization the robot walked slowly forward (the feet move toward negative x) and the feet are

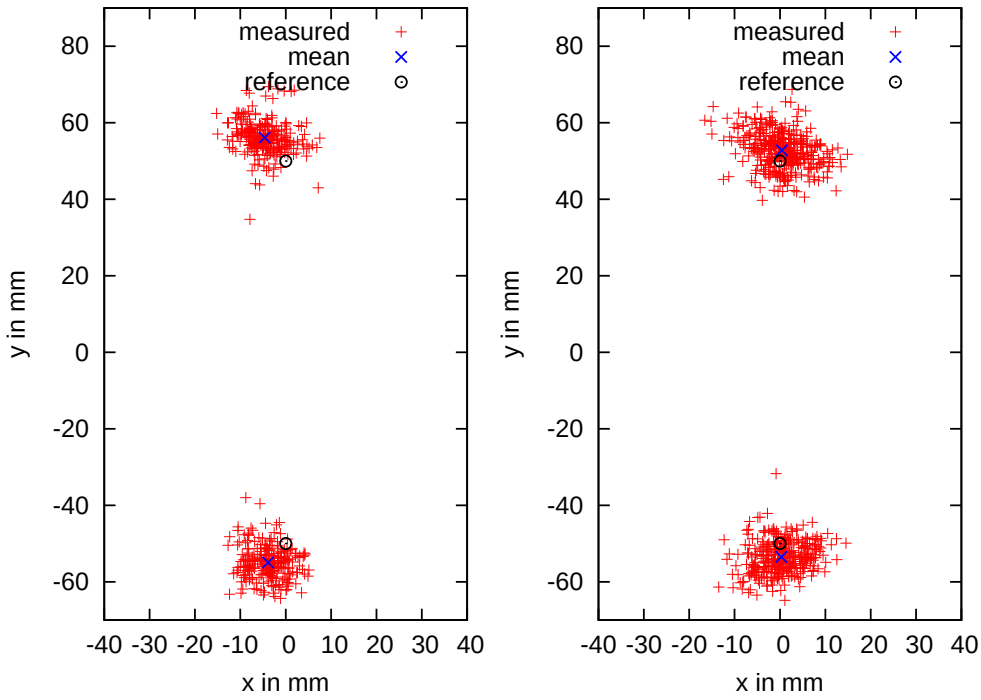


Figure 4.19: Variance in the positions of the support foot at the end of each step on a physical NAO without (left) and with the optimization (right).

further apart than the position from the reference trajectory (which is based on the 50 mm hip offset of the NAO's legs). Using the optimized model, the robot does not walk forward anymore (almost no error in x direction) and also mean y position is improved.

Walking forward/backward

For the next experiments the robots receive a walk requests that change from 150 mm/s forward to 150 mm/s backward every 3 seconds. The walking engine is expected to accelerate towards the requested walk speed as fast as possible, but without falling.

Table 4.4 shows that the optimization again reduces the overall prediction error in these experiments. The forward and backward steps provide better data for the movement along the x axis than the steps in the previous experiment. Most prediction

		mean x	pred $\dot{x}$	err y	$\dot{y}$	mean step x	change y	walk mean	speed err. std.dev.
Webots	NAO	-0.2	1.5	2.0	-12.0	0.25	0.81	-37.0	23.8
		-0.4	3.2	1.1	-3.7	-0.4	0.25	-28.5	18.2
SimSpark	type0	-0.2	1.4	0.8	-4.8	-0.1	0.4	-24.4	16.5
		0.4	-2	0.6	-5.6	0.08	0.5	-3.8	20.8
	type3	-0.1	0.4	0.9	-5.3	-0.08	0.51	-12.9	15.7
		0.2	-0.4	0.4	-1.1	0.04	-0.01	-8.6	19.1
physical	Nao1	2.7	-16.8	4.6	-29.8	1.05	2.48	61.0	106.3
		1.3	-4.7	1.4	-0.8	0.32	-0.16	38.0	45.7
	Nao2	1.6	-10.3	4.9	-32.0	1.26	2.94	64.8	102.9
		0.3	-1.2	0.5	8.8	0.32	-0.15	24.8	77.5
physical	old walk	-	-	-	-	-	-	-51.8	88.6

Table 4.4: Observed values for the forward/backward walk with 0.15 m/s with different robots. The second row for each robot contains the results using the optimization.

errors are now reduced by the optimization.

In this experiment, we use odometry information from the motion of the torso (position and angle) relative to the feet to calculate the approximate current walk speed. If errors in the predictions prevent the robot from walking with the desired speed, the robot measures that but consistently chooses wrong step target positions which do not change the walk speed correctly. Therefore, a smaller error in the predictions allows the robot to follow the requested walk speeds more accurately.

For 0.5 seconds before each change in the walk direction, we comparing the measured walk speed with the requested speed for the average speed error shown in table 4.4.

The joints in Webots react slow and the walk is too slow. The optimization improves it slightly, but it does not reach 150 mm/s. On the other hand, the physical robots move too fast and the speed varies. The optimization reduces the error in the walk speed and reduces the standard deviation of the speed, which means that the robot walks with a more constant velocity.

Without changing any parameters and without optimization, the new walking engine walks too fast backwards, such that it can not slow down and falls. For the experiments in the physical robot without the optimization, it was necessary to set a mass offset of 5 mm to shift the mass forward. This improves the stability slightly, but the robot still falls sometimes. Figure 4.20 shows measured walk speeds using this offset and no optimization on a physical robot. The walk speed varies and it walks too fast. Especially the velocity backwards is too high, such that it sometimes takes longer to slow down and change the direction.

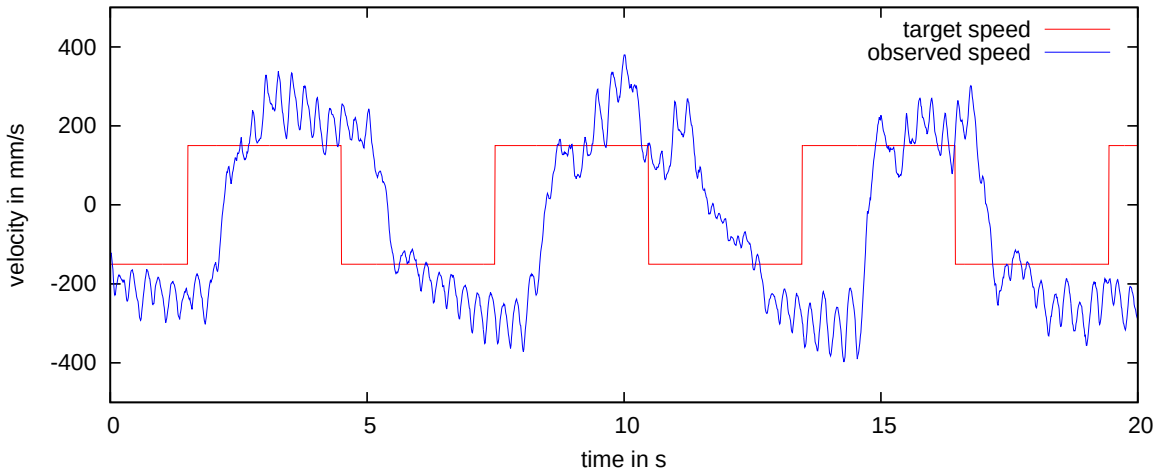


Figure 4.20: Forward/backward walk with 150 mm/s on a physical robot without the model optimization.

Figure 4.21 shows walk speeds measured on a physical NAO with optimized parameters. The improved prediction yields a more stable walk that maintains a more constant velocity. The backwards velocity is still too high, but not as much as without the optimization and the robot does not fall.

Since the observed walk speed does not depend on information from the walking engine, we can compare these results with the speeds of the previously used walking

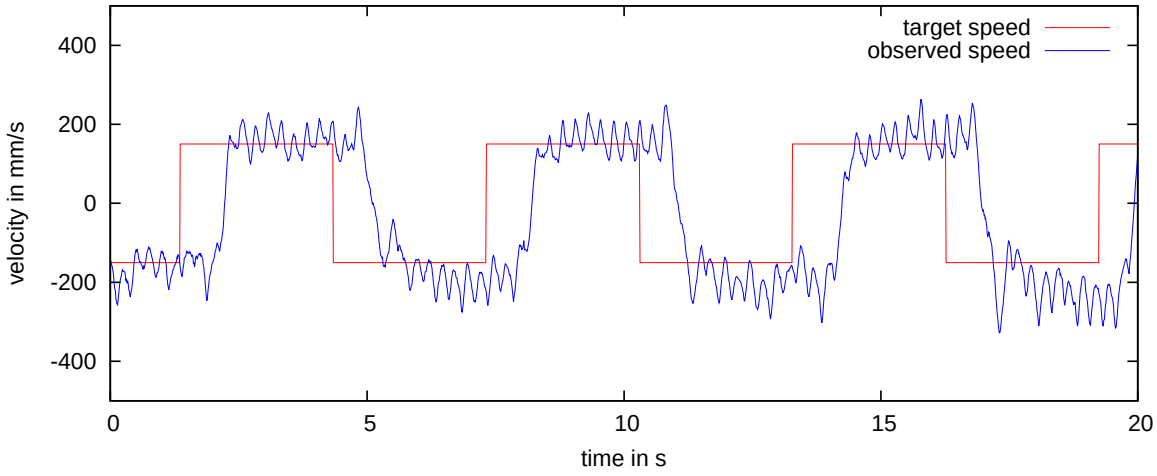


Figure 4.21: The new walk on a physical robot with optimized model parameters.

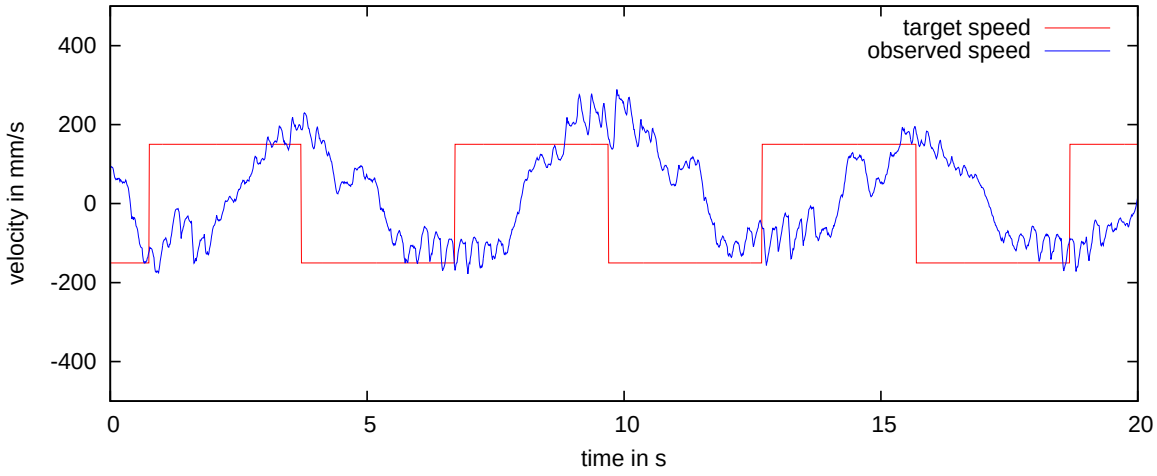


Figure 4.22: Results for the forward/backward walk using the B-Human based walking engine. Using our current parameters this walk accelerates slowly to be stable

engine (B-Human based walk). This walk is very stable, but figure 4.22 shows that it accelerates much slower and the forward velocity is faster than backwards. One of the parameters of this walk controls the maximum velocity change. However, increasing this parameter does not neccessarily improve the walk. With a higher allowed acceleration the walk is less controlled and the torso starts oscillating more. This walk uses a balancer module that is activated when certain thresholds are exceeded. When the walk is too unstable, the robot balances more often and ignores the walk request

as shown in figure 4.23.

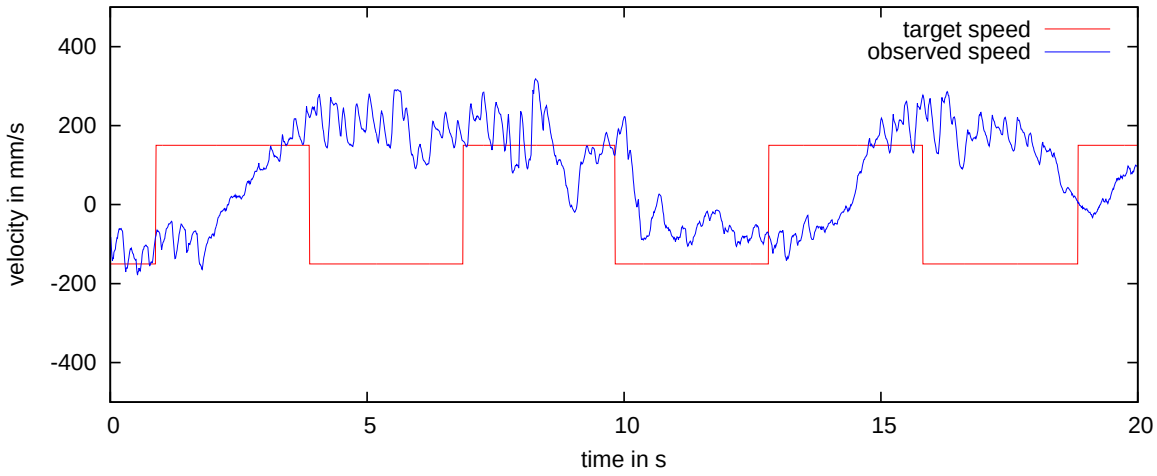


Figure 4.23: The old walk with a higher acceleration is unstable. It ignores the requested walk speed and only balances.

The B-Human RoboCup team showed that the walk can be very stable and fast. Therefore, the old walk can probably perform better. However, it depends a lot on the robot's calibration and the fine-tuning of many parameters.

Acceleration

Some walking motions on humanoid robots start to walk with a fixed first step to bring the robot into the correct oscillation. Once the robot made two or three steps on the spot it starts to accelerate and walk forward.

When the walk request is received, our walk engine already starts moving forward in the double support, such that it walks forward beginning with the first step. It accelerates as fast as possible according to the inverted pendulum model. In the following experiments the robot initially stands. The motion request is changed from "stand" directly to "walk forward with 200 mm/s". The walk is stopped after walking 0.5 m. We measure the total time it takes to reach 0.5m, the delay in the beginning

until the robot starts to move (first step forward), the average acceleration and the error in the velocity reached.

		mean x	pred $\dot{x}$	err y	$\dot{y}$	total time in ms	delay in ms	acc. in m/s^2	speed mean	error std.dev.
Webots	NAO	-2.1	15.6	1.8	-10.5	3758	402	0.44	-44.1	35.8
		-1.2	11.6	0.7	-0.1	3588	301	0.43	-38.0	24.7
SimSpark	type0	-0.6	4.7	0.5	-3	3000	296	15.0	18.8	
		-0.24	4.0	0.33	-0.8	2082	301	14.6	16.4	
physical	Nao1	4.6	-32.3	5.1	-33.5	2920	331	0.68	57.1	44.1
		1.4	-12.2	1.0	5.5	3030	354	0.75	15.3	43.3
physical	old walk	-	-	-	-	4287	856	0.12	-1.3	36.8

Table 4.5: Observed values for the acceleration test. The second row for each robot contains the results using the optimization. The results are average numbers over 30 runs.

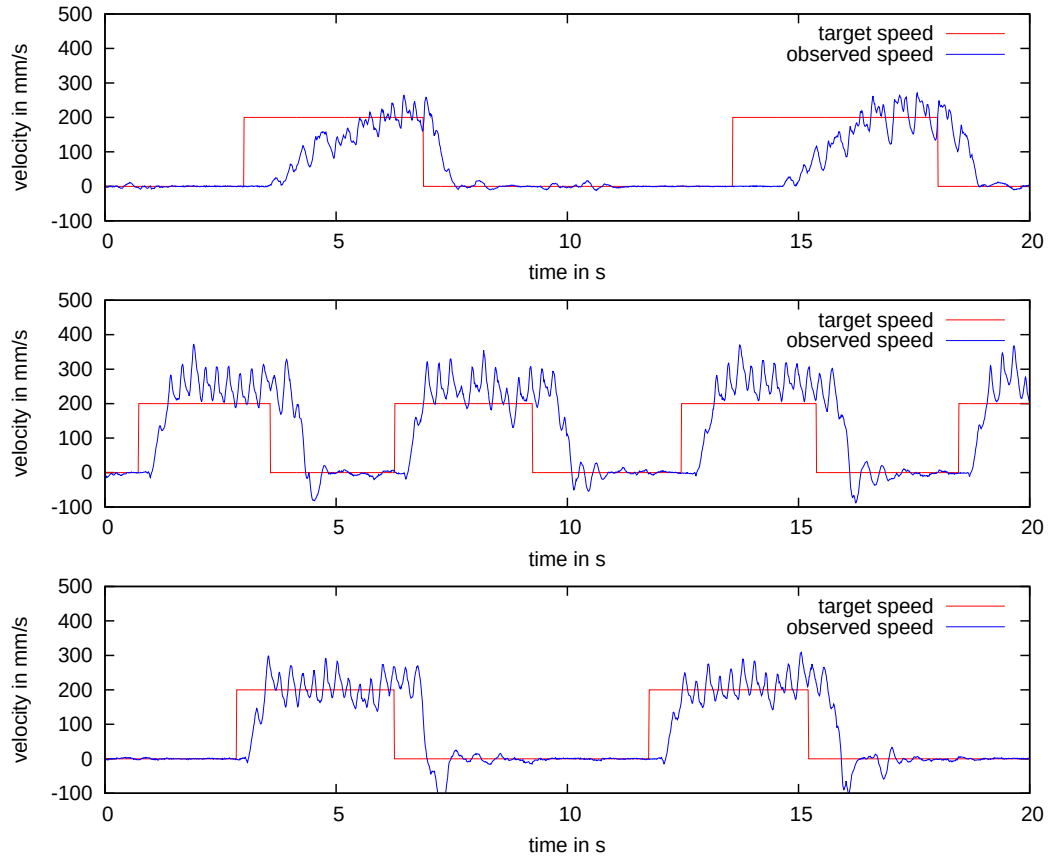


Figure 4.24: Velocities during the acceleration test using the B-Human based walk, the new walk without optimization and the new walk with optimized parameters. The optimization prevents the robot from walking too fast.

Table 4.5 summarizes the results for the acceleration test. As in the previous experiments, the new walk accelerates much faster than the B-Human based walk, but it is less controlled and walks too fast. The old walk accelerates slowly and reaches the requested velocity very accurately. The optimization improves the new walk and reduces the error in the reached velocity without slowing down the acceleration. Figure 4.24 shows examples for the observed velocities during the experiments.

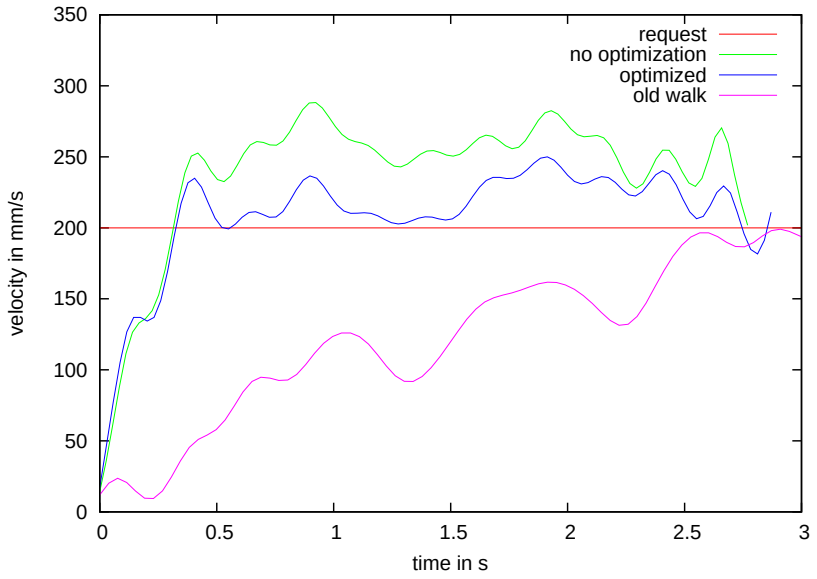


Figure 4.25: Walking speeds averaged and smoothed over all test runs.

Figure 4.25 compares the walking speed using the different configurations on a physical robot. At time 0 the walk request starts. The new walk accelerates much faster. Using the optimization, the accuracy of the walk is improved.

4.4 Conclusion

We have implemented a closed-loop LIPM-based dynamic gait for the NAO. Components such as the reference trajectory generation and step planning use the physical

model as black box to generate the corresponding motion. This allows to easily modify the model. We added several parameters to the inverted pendulum model that can be modified to improve the model predictions. These parameters are simple offsets and scaling factors that can compensate systematic errors caused by variances in the hardware of physical robots.

We optimize the model parameters using CMA-ES to fit the model to observations of the robot's movement. The experiments show, that the optimization reduces bias in the predictions by approximately 50% to 90% (varies depending on walk requests, robot, environment). It is able to improve the walk, even if it uses only a few observed steps to adjust the parameters and runs with limited computational resources, e.g. on the 1.6 GHz Atom CPU of of a physical NAO.

In simulations, the optimization is not able to improve the walk much, since the errors are already small. On physical robots, the errors are different from robot to robot. The optimization can quickly improve the model parameters for individual robots.

Approaches for motion generation using optimization (optimizing trajectories, training CPG's) work well, but often do random exploration and can be difficult to apply to physical robots. On the other hand, model-based approaches only generate feasible motions, but often need some fine-tuning of parameters and an efficient balancing to compensate for model errors additionally to external disturbances.

We successfully applied a parameter optimization to components of a model-based motion generation to combine the benefit of limiting the search space to safe motions that will not break a physical robot with an optimization such that the motion is

automatically adjusted for different robots (physical robots or different simulated robots) and no manual parameter-tuning is needed.

4.4.1 Future Work / Extensions

More components of the walking engine could be improved using parameter optimization. The movement of the mass in x- and y-direction does not have a big influence on the used current in the legs. It seems that very small changes in the hip height or the angle of the supporting foot can have a bigger effect on the current. The control of the supporting leg could be optimized to reduce the currents. Since a walking robot makes more than 1000 steps per minute, it should do enough repetitions to run the optimization on a physical robot. Another task that can probably be improved using optimization or learning is the torso angle control.

A possible general extension of the walk engine is the integration of a kick. A kick can be added as a walk action (a different step type). A kick action could be seens as a step with longer duration and maybe constraints on the swing target position after the kick, such that it is treated as a long step.

For a kick integrated in the walking motion, it would be useful to plan the previous steps such that a kick is possible at a certain position relative to the robot. Therefore, the most important change would be a step planning with a different objective. Instead of reaching the reference trajectory, the step planning would have to choose steps such that it reaches a given target state at a given relative position to the current state. This would also be helpful for just walking to a given position without

using arbitrary values for slowing down when approaching the target position.

Appendix A: The RoboCanes Software

This chapter gives an overview over the RoboCanes software, including the agent, various tools and scripts and how to use them. The described framework and tools were build from scratch to allow the RoboCanes team to work more efficiently with the simulated and physical NAOs and successfully participate in the RoboCup 3D Soccer Simulation and Standard Platform League. The RoboCanes framework is an important infrastructure for most of our research with the robots.

The focus is on information needed for working with the framework and understanding the general concepts. Some implementation details are discussed, but the information in those sections is not needed for only using the framework and tools.

After giving an overview over the included files and directories, section A.2 describes the agent framework and general module structure for the 3D Soccer Simulation and the SPL. Additionally, section A.5 and A.6 will describe various scripts and tools.

A.1 Repository Overview

The robocanes repository contains the following directories:

- **bin/binNao**

  The bin directory contains all files needed to run the agent. After compiling the agent, the executable file `robocanes` is copied there. Additionally there are config files, start scripts and all other files loaded by the agent at runtime (e.g. motion files for special actions).

- **build**

  The build directory is created automatically by the configure script. Usually, there is no reason to touch this directory. It only contains dependency files and objects created during the compilation.

- **doc**

  Directory for creating a doxygen documentation.

- **lib**

  Additional libraries used by the agent should be stored in `lib`. The configure script already adds `lib/include` to the include path and links the binary such that it will find the libraries in `lib/linux32`. In fact, the binary searches for libraries in ../lib/linux32, ./lib (local in the bin directory), and /root/lib (for the physical robot).

- **scripts**

  This directory contains several scripts, e.g. for running games against different

opponents automatically or several scripts for configuring and working with the physical NAO.

- **src**

  This directory contains all the source code for the agent (the `robocanes` binary). The configure script searches for all c and cpp files in this directory (unless a path is explicitly excluded in the configure settings). Everytime a cpp file is added or removed, the configure script has to be executed to update the makefiles in the build directory.

- **utils**

  Additional tools that run independent from the agent. These can be own tools written by the RoboCanes group or tools from external sources (e.g. the SPL-GameController provided by the RoboCup community).

A.2 The RoboCanes Framework

A.2.1 Build system

The RoboCanes agent does not use a build system such as CMake, SCons, etc. Some build systems require a lot of extra files within the source file directories (CMake-Lists.txt) that sometimes have to be updated manually when files are added or moved. For most build systems, everyone working on the project has to know how to use and maintain it. Some build systems are also slow or waste time, e.g. by unnecessarily updating all dependencies every time "make" runs. Since portability is not a priority,

the RoboCanes agent uses an own, simple configure script. The goal was to use a build system that is as easy to use as possible, does not require editing files, does not add extra files in the source directories and handles and updates dependencies correctly.

The RoboCanes build system consists of a single configure script that searches for all C and C++ files in the given source directories and creates a Makefile and a build directory. When a file is added, moved or deleted, the configure script has to be executed to update some files in the build directory, but no files have to be edited manually.

The configure script does not have to be executed to only update dependency information. Dependencies for existing files are updated by the Make files every time they are compiled.

There are several settings at the top of the configure script, such as the name of the binary, source directories (also directories to ignore), compiler flags or used libraries. Those settings can also be overwritten by command line arguments passed to the configure script.

The RoboCanes agent can be compiled with several different configurations (SimDebug, SimRelease, Nao, NaoLocal, NaoRelease), which use different compilation flags and ignore source files in some cases. The SimSpark agent for example ignores the Nao subdirectories which would require OpenCV, which is not used for the SimSpark agent. All these configurations can be initialized by different scripts that call the configure.sh with different arguments.

For example compiling the release binary for the SimSpark agent can be done by

executing `./configureSimRelease.sh` followed by `make`. This will write the binary directly to the `bin` directory. The configure scripts for different configurations only add make targets to the Makefile. The default target that is compiled with `make` is always the last configuration that was created/updated using the configure script. All other configurations can be compiled as separate targets, e.g. with `make SimDebug` or `make NaoRelease`. The target `all` compiles all configured versions of the agent.

More Details

The build directory is created to store the compiled object files and files needed for the dependecies. When the configure script is executed, it checks the dependencies for all source files, which takes several seconds. This should not be done every time the agent is compiled. Usually only a small number of source files is changed and only those files are compiled. Those files are also the only files that might have different dependencies. Therefore, it is sufficient to only update the dependency information for these files.

The configure script realizes this by creating a `dep` file for each object file which contains the Make target to compile this object. The target's dependencies are generated by gcc using the -MM flag. The configure script adds commands to these targets, such that they compile the objects and update the dep file. All these generated dep files are included in the main Makefile, such that all targets are checked for changes when "make" is executed.

Executing the configure script with a new target name does not overwrite the Makefile. It only adds another target to the Makefile and creates a new subdirectory

for the target in the build directory.

If the name of the output file (the linked binary) uses the extensions `.so` or `.a` the configure script will automatically add the correct compiler flags and compile a shared or static library. By defining the source directories or ignored directories correctly, parts of a project can be compiled as a library. The target for the main executable can be configured to use that library. When configuring the main target, the library target can be given as a dependency, such that compiling the main target will also check if there are changes in library files that need to be compiled to create a consistent complete build.

Some files in the RoboCanes project are generated automatically (the parsers using Flex/Bison). This is not hardcoded in the configure script. Instead, the configure script assumes by default, that non-existing files can be generated by additional targets. Those targets can be added by including additional Makefiles.

All extra files to include need to be listed in the configure script settings under "includedMakefiles". Additionally, generated cpp files might have to be listed under "generatedFiles" to make sure those files are compiled. A simple additional Makefile could contain just one target to generate a specific file:

```
myGeneratedHeader.h: inputFile
    @bash_command_to_generate_file inputFile
```

By including additional targets, it is also possible to provide generic targets for generating certain files. The follwing code automatically generates code using Flex and Bison:

```
BISON=‘which bison‘
```

```
FLEX=‘which flex‘

# --- targets for creating code with bison and flex
%.tab.c %.tab.h: %.y
        @echo $(TUQ)[BISON]$(NOR) $(notdir $*.y)
        @$(BISON) -d --file-prefix=$* $<

%.l.c: %.l %.tab.h
        @echo $(TUQ)[FLEX]$(NOR) $(notdir $*.l)
        @$(FLEX) -t $< > $@
```

A similar extension is used for the configure in the RoboCanes Manager (see section A.6.2) to gerate code for Qt using the Qt User Interface and Meta Object Compiler (uic and moc).

A.2.2 Agent Structure

An agent observes its environment and acts upon it. In other words, the interaction of an agent with its environment is limited by its sensors and actuators. The agent has to read sensor values, process them and produce a stream of outputs that control the actuators. Usually, this can be represented by a cycle as shown in Figure A.2.26. When the agent receives new perceptions, it needs to process these perceptions, update some internal states and in the end produce new control outputs that are send back to the environment. An agent can be implemented in complicated ways, including modeling and learning, but overall there is always the cycle that processes perceptions and produces control outputs.

Figure A.2.27a shows an example structure of a model-based reflex agent, that contains the same loop. Several types of agents are described in [70] using a control loop consisting of different components. The model-based reflex agent could also be

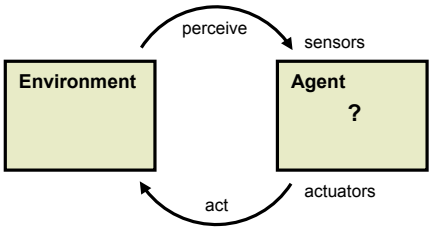
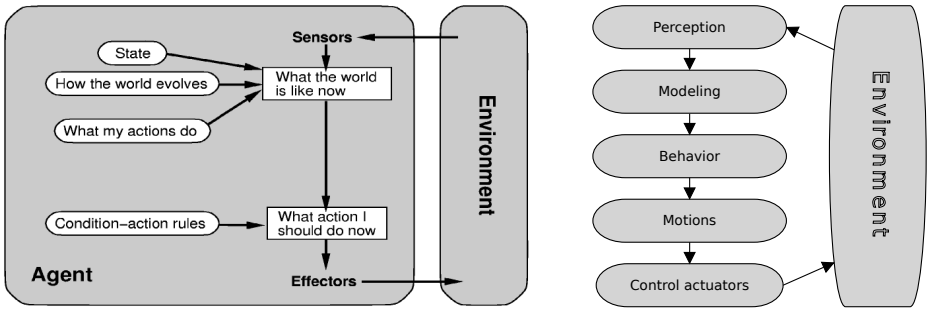


Figure A.2.26: General control loop of an agent acting in an environment.



(a) (b)

Figure A.2.27: The control loop of a model-based reflex agent (a) (from [70]) and a more general example for possible components of an agent (b).

split into two main components, the modeling and the behavior. Depending on the format of the perceptions and the outputs, the agent would require additional steps in the processing and can be split further into a sequence of more components as in Fig. A.2.27b. These components can be seen as categories of tasks. For an agent controlling an autonomous robot these can be split into a lot of much smaller tasks, e.g. the detection of a certain feature in a camera image, self-localization or the control of the robot's head. All these modules require some information and produce new information that might be required by other components. The RoboCanes module framework supports exactly this structure and helps to maintain it.

A.2.3 Module Framework

The RoboCanes module is a general framework that helps to organize the code of the agent. It guarantees that the various modules are executed in the correct order, helps avoiding hard to debug problems with side-effects and makes it easy to add or replace modules.

The RoboCanes framework is based on concepts of a blackboard architecture [32, 10]. In a blackboard architecture, all known information is gathered on a blackboard. The modules can be seen as experts or knowledge sources. Each module can perform a specific task, but it has to wait for all information it requires to be on the blackboard. Once the constraints of a module are fulfilled, it can use the information on the blackboard and add additional information to it. Each module solves only a small subtask, but they all work together to solve a complex problem. The requirements of the modules (the dependencies) can be used to find the correct execution order of the modules. After all modules have been executed, the blackboard contains all available information including the solution to the overall problem, e.g. the control outputs for the joints of the robot.

The RoboCup team B-Human uses the same architecture [71]. There are several similarities in the RoboCanes framework, especially in the way of creating modules and defining their dependencies. The RoboCanes framework followes several ideas of the B-Human framework, but it is implemented differently. It is much smaller and less complex than the B-Human framework and simplifies some aspects further, e.g. adding modules does not require to modify any framework code. It is not necessary to

know anything about the implementation of the framework to work on the modules of the agent.

The complete RoboCanes agent consists only of two types of classes: modules and representations. Most of the computation is done in the modules. Representations only store information and can be seen as data containers or interfaces between modules. Modules can require and provide representations. A module can not be executed before all required representations have been updated in the current cycle. When a module is executed it can read the information from the required representations and update other representations that might then allow more modules to be executed.

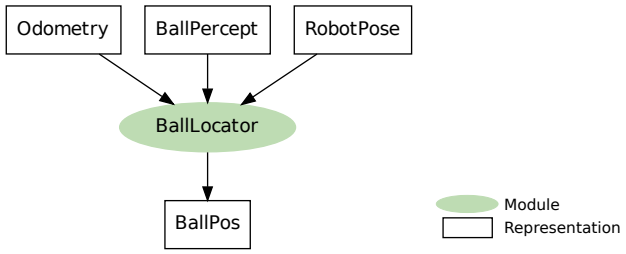


Figure A.2.28: Example for a module and representations. Here, three representations are required by the BallLocator. It provides the representation BallPos.

Modules have no direct access to other modules. Modules can only access representations that are declared as required or used representation in the class definition of the module. Each module also declares which representations it is able to provide. Figure A.2.28 shows an example for a module and its required and provided representations. Those are the only representations the module can access.

All modules declare the used representations and are registered in the framework. This way the framework knows all available modules and can activate and execute them. It is not necessary to call any methods of the module somewhere in a central

file to activate it. The framework controls the execution of modules.

The information about the required and provided representations is used by the framework to create the module graph (a directed acyclic graph) that contains all modules and representations and shows their dependencies. By sorting the modules topologically, the framework finds an execution order of the modules, such that all modules are only executed after the required representations where updated by other modules. The framework stops and shows an error message, if a required representation is never updated or if there are dependency cycles in the graph.

If the agent is compiled as debug version, the framework creates a `.dot` file when the agent starts to visualize the module graph using the Linux Graphviz tools [14]. This file can be used to generate a picture, e.g. with the command `"dot -Tpdf modulegraph.dot > modulegraph.pdf"`.

Creating representations and modules

Representations are classes that only store information. Therefore, most representations will only list some member variables. The framework needs to be able to access all representations through a common base class and some type dependent methods need to be defined for each representation. This is all done by the `REPRESENTATION` macro. This macro creates a base class for the given representation. The name of this base class is the name of the new representation with "Base" appended. By extending this base class, the new representation class inherits all bases and methods that are required by the framework. The following code example shows how to create a new representation with the name "ExampleRep":

```
#include "framework/Representation.h"

REPRESENTATION(ExampleRep)
class ExampleRep : public ExampleRepBase
{
  //...
};
```

Representations do not have to implement any methods and can only consist of the class definition in a header file.

Modules are created in a similar way. The macros `MODULE` and `END_MODULE` generate the code for the module base class. The following code is sufficient to define a module, that could be added to the framework:

```
#include "framework/Module.h"

MODULE(ExampleModule)
END_MODULE

class ExampleModule : public ExampleModuleBase
{
  public:
    void init() {  }
    void execute() {  }
};
```

There are abstract virtual methods in the base class. Some methods have empty default implementations, others might be abstract and have to be implemented in the module. The methods `init` and `execute` are optional. Those methods are called by the framework. The `execute` method is called in every cycle, the `init` only once in the beginning (all representations are available when init is called, which is not the case in the constructor). The macro `MAKE_MODULE` has to be used in the modules `.cpp` files to register a module in the framework (e.g. with the line

"`MAKE_MODULE(moduleName);`"). Otherwise, the framework would not know that this module exists, since it is not included or created in any other file. This is not required for the representation. The modules register all representations they use in the framework when they are created.

There are two macros for creating the module base classes, because the representations used by the module need to be defined between these macros. The macro `REQUIRES` defines that a module requires the information of a representation and can only be executed after that representation was updated. This macro automatically adds a pointer to the base class to access that representation. The name of the pointer is the name of the representation with the prefix "the". It only allow reading from the representation.

```
#include "framework/Module.h"
#include "representations/perception/BallPercept.h"

MODULE(ExampleModule)
  REQUIRES(BallPercept)
END_MODULE

class ExampleModule : public ExampleModuleBase
{
  public:
    void execute()
    {
      log << theBallPercept->pos.x;
    }
};
```

The macro `PROVIDES` defines that a module is able to provide a certain representation. An update method has to be implemented in the function for each provided representation. When the framework calls an update method, it passes a pointer to the representation that allows the module to write to it. The following code

shows an example of a module that provides `BallPos` based on information from `BallPercept`.

```
#include "framework/Module.h"
#include "representations/perception/BallPercept.h"
#include "representations/modeling/BallPos.h"

MODULE(ExampleModule)
  REQUIRES(BallPercept)
  PROVIDES(BallPos)
END_MODULE

class ExampleModule : public ExampleModuleBase
{
  public:
    void update(BallPos* theBallPos)
    {
       theBallPos->relativePos = theBallPercept->pos;
    }
};
```

Modules can provide multiple representations, but it is not necessary to always use all of them. It depends on the configuration of the framework, which update methods are used. Since writing to representations is only possible through the pointer that is passed to the update method, the framework controls which module can write to a representation. A faulty module can not write to a wrong representation by accident which otherwise might cause difficult to debug problems. The `execute` method is always called before any update methods.

In some cases, a module has to access a representation, but adding it as requirement would add circular dependencies. In that case, the macro `USES` can be used to just add the representation pointer to the base class. However, this should be used only if it is really necessary. A `USES` does not have any effect on the execution order of the modules. It can be used to read values from the previous cycle, when accessing

a representation that is updated later. The `USES` macro also does not guarantee that the representation is provided at all. The representation pointer can be NULL.

The macro `REQUIRES_IF_PROVIDED` works exactly as the `REQUIRES` macro, if the representation is provided by another module. However, if the representation is not provided, the framework does not show an error. In that case, the representation pointer in the module will be NULL. This macro can be useful for optional representations that should be updated in the correct order if they are used, but the agent would also work without them.

Module configuration (modules.cfg)

The framework does not always use all available modules. There can be different version of the same module or alternative modules for the same purpose. The modules configuration defines which modules are active and which representations each module provides. The default location for this config file is `bin/config/modules.cfg`.

Each line in the `modules.cfg` can contain a module name followed by a representation that this module provides. Listing only the module name means that the module is only executed, but it does not provide anything. If a module provides more than one representation there has to be one line for each provided representation. Figure A.2.29 shows a module graph that is created by the following modules configuration (part of the SimSpark agent configuration):

```
SimSparkConnection ServerMessage
Parser FrameInfo
Parser SensorData
Parser PlayerInfo
Parser JointData
```

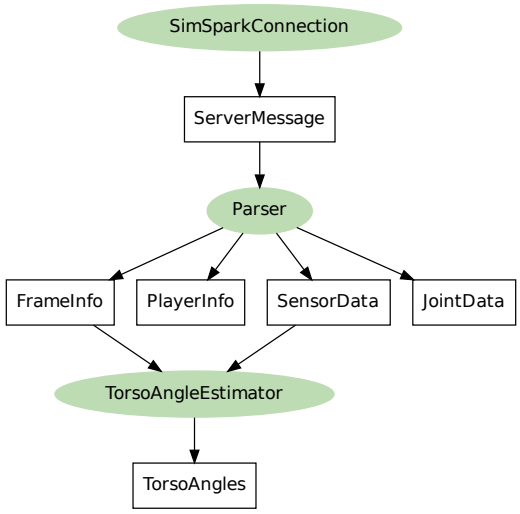


Figure A.2.29: Example module graph containing only a few modules from the RoboCanes agent.

If a modules configuration produces a modules graph with cycles or missing dependencies, the framework shows an error message and exits.

Threads

It is possible to define two or more graphs in the `modules.cfg` that run independently in threads. A thread is created by a line as for example "[cognition 5]". This defines the thread with the name "cognition" and priority 5. All modules listed after this line are added to that thread. Modules can only run in one thread. Multiple threads are not synchronized. A thread can not be blocked by other threads (unless a thread with the maximum priority 15 uses the complete CPU time).

Each thread has to contain a complete module graph and all representations required by modules in a graph have to be provided. Representations provided in

another thread can not directly be accessed (with USES or REQUIRES). However, representations can be copies from one thread to another to be provided there. For example, if a representation BallPos is provided in the cognition thread, but is required by modules in another thread, the line "cognition BallPos" can be used to provide a copy of BallPos in that thread. This copy is always the current state of the representation in the source thread. Since there is no synchronization or blocking, this copy might not get updated every frame or miss information depending on the frequencies the threads run with.

The following example configuration creates two threads:

```
[thread1 5]
Module1 Data

[thread2 1]
thread1 Data
Module2
```

The first thread executes a module that provides a representation `Data`. If `Module2` in the second thread requires `Data`, the line "thread1 Data" is needed to copy that representation from the first thread and provide it in the second thread.

Network streaming of representations

An agent can request representations from another agent running on a remote machine just by letting the module "network" provide that representation and setting the IP address where the remote agent is running in the file `config/NetworkModule.cfg`.

The example above could run in two agents, each executing the modules of one thread. The first agent only needs to execute the module that provides `Data` and the

network module:

```
[debug 1]
network
[thread1 5]
network
Module1 Data
```

The network module needs to be executed to run a server and accept connection of remote agents. If there are multiple threads, each thread should execute "network" to allow access to all representations in those threads (this module is a special module that can be executed in multiple threads). Additionally, there should be an extra thread for sending all information to avoid slowing down or blockign the other thread if the network is too slow. The configuration file defines the name of the thread used for sending (e.g. "debug"). Another agent can use the following configuration to request `Data` from a remote agent:

```
network Data
Module2
```

When this agent is started, it tries to connect to a remote agent using the IP and port defined in `config/NetworkModule.cfg` and requests the representation `Data`. If this representation exists in the remote agent, it starts sending it every cycle. If the bandwidth is limited it might not send the representation every time it is updated. If the receiving agent runs too slow it also skips frames to provide the most recent information instead of buffering all information which could create an increasing delay.

Copying representations from one thread to another or streaming over the network uses the stream method described in A.2.5.

Internal structure and plugins

This section describes more details about the internal structure of the framework and how to add functionalities or objects through plugins. This information is not needed to only work with the framework and write new modules or representations.

Figure A.2.30 gives an overview over the most important objects and how they interact with each other. All modules are implemented using the macros described in the previous sections. The `CREATE_MODULE` macro in each cpp file of the modules create a loader object that registers itself in the controller. For the controller this loader class is a generic module loader, such that the module's header file does not have to be included anywhere. This way the controller has access to loader classes for all available modules and representations. It creates a process object for each thread defined in the `modules.cfg`. These processes use the loader objects to create instances for all activated modules.

The macros used in the header file of each module create the module specific base classes (module name with "Base" appended), which defines the representation pointers used to access representations. All the object specific base classes extend a generic module base class, such that all modules can be treated as generic objects in the controller and process class.

When a module instance is created using a loader class, the module specific base class registers representation loader objects and information about the required and provided representations in the controller. This way the processes get access to representation loader objects from the used modules without including all representation

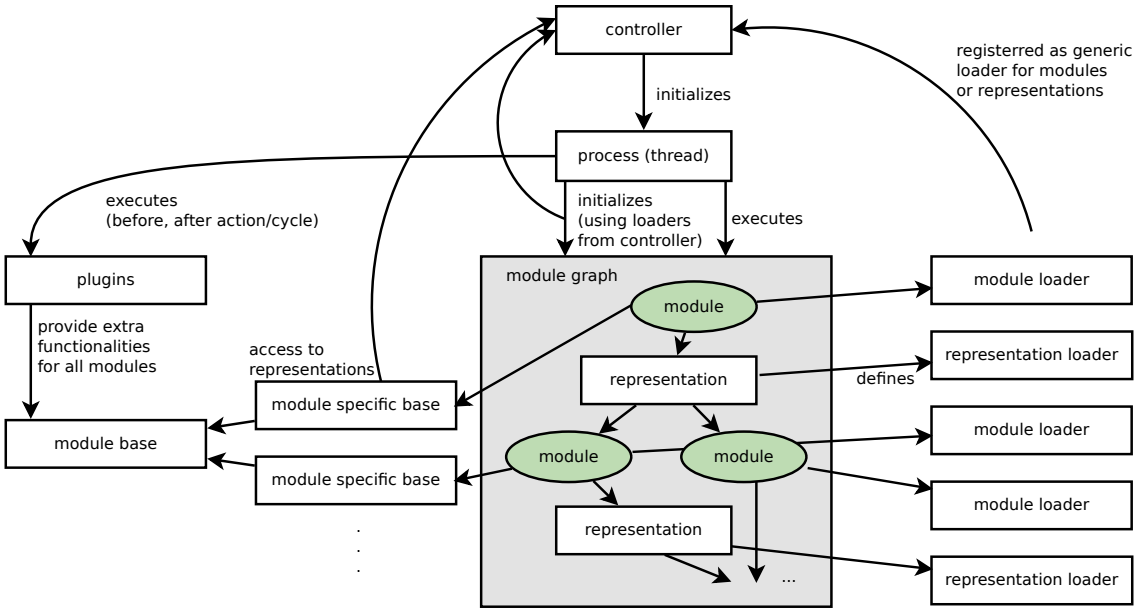


Figure A.2.30: All available modules register loader objects in the framework controller, which creates process objects that use the loader objects to create the module graph. Plugins can add objects to the module base class.

header files and listing the representations explicitly. The representation pointers defined in the module specific base class (pointers with the prefix "the") gain access to the representation instances through the controller.

The framework plugins are classes with methods that are called by the processes before and after every call of an execute or update method and at the end of each cycle. The plugins can add additional objects to the module base class (the module base extends a class PluginModuleBase). For example, all modules in the RoboCanes framework have access to a "debug" object through the module base class, which probides methods for debug drawings or time measurement. The plugin initializes the debug object, measures the time used by each module execution and representation update and sends a flush message for drawings at the end of each cycle.

Additional objects can be added to the module base class (via the PluginMod-

uleBase) without adding a plugin class. Similarly, the representation base class can be extended by adding variables to the class PluginRepresentationBase (not shown in Fig. A.2.30, but analoguos). Any actions that should be executed e.g. after every representation update or once every cycle need to be implemented in a plugin class. This avoids adding extra code for specifig extensions to the process class of the framework. Instead, all extensions can be integrated as plugins.

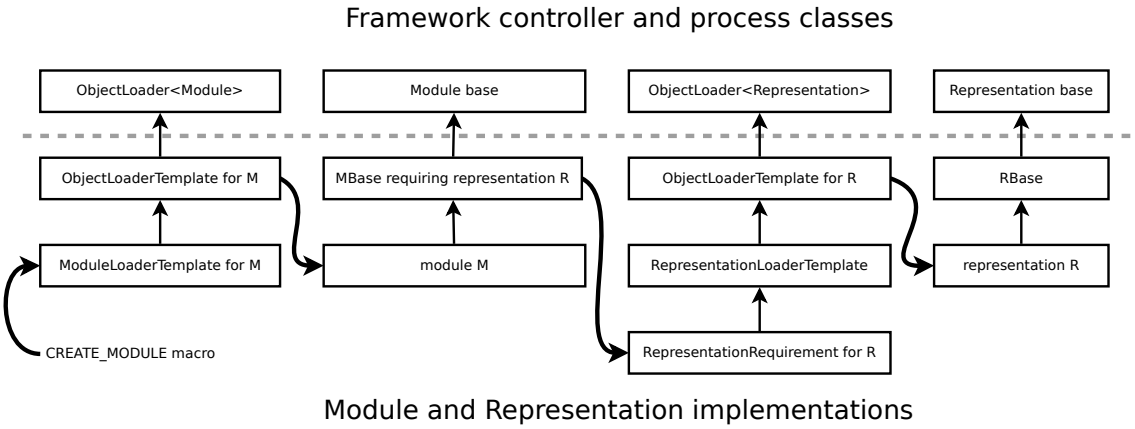


Figure A.2.31: Base classes and templates of modules and representations and loader classes. The thin arrows upwards point to the extended base class. The thick arrows show where instances of another class are created.

Figure A.2.31 shows a part of the framework's class hierarchy and how modules and representations can be added to the framework without modifying any framework code (adding header files, creating representation instances, etc). All classes below the dashed line are created by the implementation of the module and need the definition of the type of the module. The macro `CREATE_MODULE` in the cpp file of a module creates an instance of a loader class for this specific module. Several templates and base classes are used to register the object as a loader for a generic Module class in the framework, which is completely independent of the exact type of the module. However, this can be used to create an instance of the module. This module uses a

base class generated by the macros of the framework. It is known to the controller only as a generic Module. The module specific base class contains instances of representation loader classes (created by the macros, e.g. `REQUIRES`). These is registered in the controller as generic Representation loader objects, but using templates they are able to create instances of the representations.

From the view of the framework controller all classes are generic loader classes or generic rrepresentations and modules. All other classes that depend on the exact types of the modules and representations are created using templates and generated by the macros provided by the framework.

Framework modules (RepTransfer and NetworkModule)

Currently, two special modules can be used that are implemented without using the macros and behave different from usual modules. These modules are the module for copying representations from another thread and the network module. They have access to all representations known to the controller and they can provide any representation. They can be added to multiple threads.

Since these modules can use only representations that are registered in the controller, it is not possible to stream a representation to an agent, if that representation is not used by any other module (e.g. only for seeing the RoboViz drawings sent by the draw method in the representation).

Similar to the network module a log module could be implemented, that can save representations to a files and read from it later.

A.2.4 Framework Extensions in the RoboCanes Agent

In the RoboCanes agent, several tools have been added through plugins and modules in the module base class. The following objects are available in all modules of the agent: debug, config, log, monitor.

The debug object

The debug object provides methods for debug drawings and stopwatches. The drawings are send to the monitor RoboViz.

The config object

The config object provides a simple way of storing config values.

Log stream / LogProcess

The log stream can be used to write messages to an internal buffer. If the agent crashes (e.g. with a segmentation fault), another process writes the last log messages to the file "crash.log". When the agent is started with the option "-v", it additionally prints all log messages on the standard output.

SimSpark monitor commands

The monitor object is only useful for the simulation. It can be used to send commands to the monitor port of SimSpark. It can move the ball and robots or change the current game state.

A.2.5 Streamable Objects

The `Streamable` base class is important for sending representations from one thread to another, to remote agents through the NetworkModule and for sending values through the debug interface.

All basic data types are streamable (bool, char, int, float, double). Vectors of streamable objects are streamable (not vector of pointers to streamable objects!). Classes can be made streamable by extending the base class `Streamable`. This base class is an abstract class with a virtual stream method. Every streamable object needs to implement this stream method, which is used for serialization. Some classes used in the RoboCanes codebase are already streamable, e.g. the vector and pose classes in the `math` directory. The following code shows an example of a simple class that is streamable:

```
class Example : public Streamable
{
  public:
    int a, b;
    double v;
    Vector2<int> p;

    void stream(InStream *in, OutStream *out)
    {
      STREAM(a);
      STREAM(b);
      STREAM(v);
      STREAM("vec", p);
    }
};
```

In most cases, the stream method does not require any other code besides the macro STREAM. This macro generates the required code to serialize the class and

write to the out stream. It can also read from a stream and set modified values. Only one stream method is required for reading and writing. The variable passed to the STREAM macro has to be streamable. Optionally, the value get be given a label (e.g. shown in the debugger). If no label is defined, the variable name is automatically used as the label.

Two different types of streams are implemented: a binary stream and a text stream. The stream method of streamable objects is independent of the stream type. It works with both stream types. Using a binary stream, the example above will simply write on value after another to the stream without any overhead. The text stream will create a human readable string that also shows labels, e.g.

```
{[a] 0 [b] 3 [v] 5.820000 [vec] {[x] 2.000000 [y] -1.500000 }}.
```

The text stream is used for the debug interface. The purpose of the labels is only used to improve the readability of the outputs in the debugger. When the stream method reads from a string, it ignores the labels. The curly brackets and order of the values is important.

Representations have a default implementation of the stream method, that sends the representation as a block of binary data. It uses the `sizeof` operator to determine the size of the representation. This works correctly only if there are no pointers to variables that are not included in the representation. This default stream implementation is sufficient for sending representations from one thread to another or to another agent through the network, if the representation does not contain pointers. Those connections are always using a binary stream.

A.2.6 Debug Interface

This section describes the debug interface of the RoboCanes agent. It provides the possibility send commands to the agent to read or set variable values or activate debugging code.

Section A.6.1 will describe rdb, a separate tool that connects to the debug interface. However, the debug interface is independent from the debugger. It can be used by directly sending commands as strings, e.g. through simple tools as nc (netcat). The debugger rdb only provides a console with some additional functionalities. This section describes how to use the macros of the debug interface in the agent code.

Only streamable variables that were registered for debugging in the agent code can be accessed through the debug interface. Similarly, only code blocks that are declared as debug response can be switched on and off. Registering variables or debug code in the debugging interface can be done using macros in the agent. By default, all of these macros do not change the behavior of the code, since all debug requests are deactivated when the agent starts. The macros only add the possibility to access values or activate debug code later. Table A.2.6 gives an overview over the implemented macros.

The id string used to identify variables or debug reponses can be any string. However, meaningful prefixes should be used to avoid conflicts, e.g. "module:ExampleModule: someValue". The following shows examples on how to use the macros:

```
OUTPUT("module:Example:debug", myVar); //by reference, uses &myVar, no copying
OUTPUT_BY_VALUE("module:Example:debug", someFunction(x,2)*0.1);

DEBUG_RESPONSE("module:Example:test", createPlot());
```

Macro	Command	Description
OUTPUT(”id”, variable);	get id watch id	Read the variables value once. Read the value repeatedly.
MODIFY(”id”, variable);	set id x setOnce id x unset id x	Overwrite value of variable with x every cycle. Overwrite value only in the next cycle. Stop overwriting variable. Implies OUTPUT.
OUTPUT_BY_VALUE(”id”, value);		Same as OUTPUT, but value is copied.
DEBUG_RESPONSE(”id”, code);	dr id dr id off dr id once	Activates debug response. ”code” is executed every cycle. Deactivate debug response. Stop executing ”code”. Execute code only once (in next cycle).
DEBUG_RESPONSE_OUTPUT(”text”);		Can only be used inside DEBUG_RESPONSE.
DEBUG_RESPONSE_ONCE(”id”, code);		Debug response that always runs only once.

Table A.2.6: Debug interface macros and the corresponding debug commands.

```
DEBUG_RESPONSE("module:Example:test",
{
  //do
  //something
  DEBUG_RESPONSE_OUTPUT("Text printed in the debugger.");
});
```

The only requirement for the debug interface to work is the module “DebugModule”. It has to be activated in the agent (preferably in a low priority debug thread). This module needs to call on update method of the debug interface regularly to accept new connections and send or receive data. The debug interface itself is initialized as a singleton. There is always only one instance that can be accessed globally by the macros. Thus, the macros can be used anywhere in the code, not only in modules.

Some objects are automatically registered by the framework for debugging:

- MODIFY on representations, id “representation:NAME”.
- MODIFY on all config values, id “config:CONFIG:VALUENAME”.
- OUTPUT on all stopwatches (see debug object, section A.2.4) with the id “timing:STOPWATCHNAME”. This is for example used for all `execute` and `update` methods, e.g. “timing:NaoCamera:execute” or “timing:SelfLocator2:updateRobotPose”.

A.3 SimSpark Agent

In the 3D Soccer Simulation League, SimSpark simulates the environment and the robots. The RoboCanes agent has access to sensors and actuators of one simulated robot (through a TCP connection to the server) as shown in figure A.3.32. The monitor RoboViz connects to SimSpark to show the environment and the robots. The agent can send commands directly to RoboViz to show debug drawings.

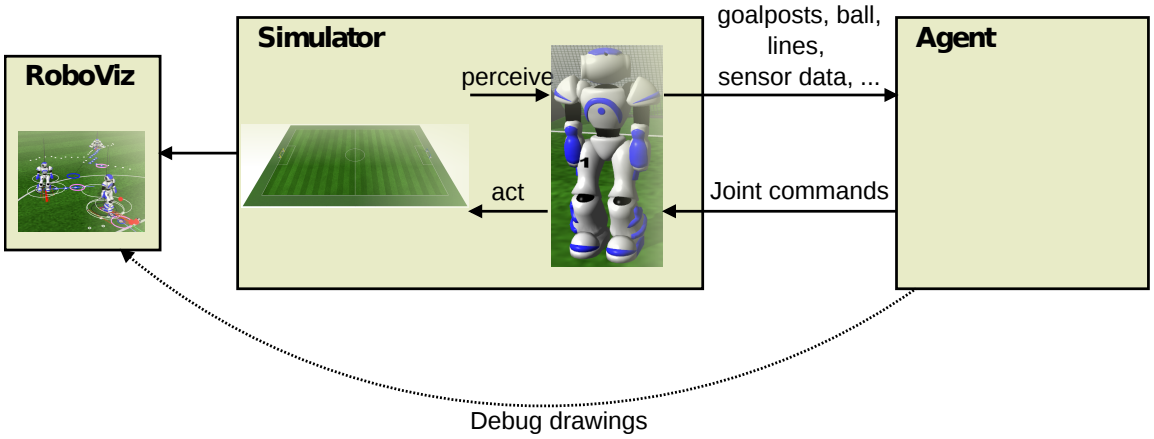


Figure A.3.32: The connections between SimSpark, the agent and RoboViz.

Figure A.3.33 shows the module graph for the SimSpark agent. The SimSpark-Connection module receives the messages from the server. It provides it as one string.

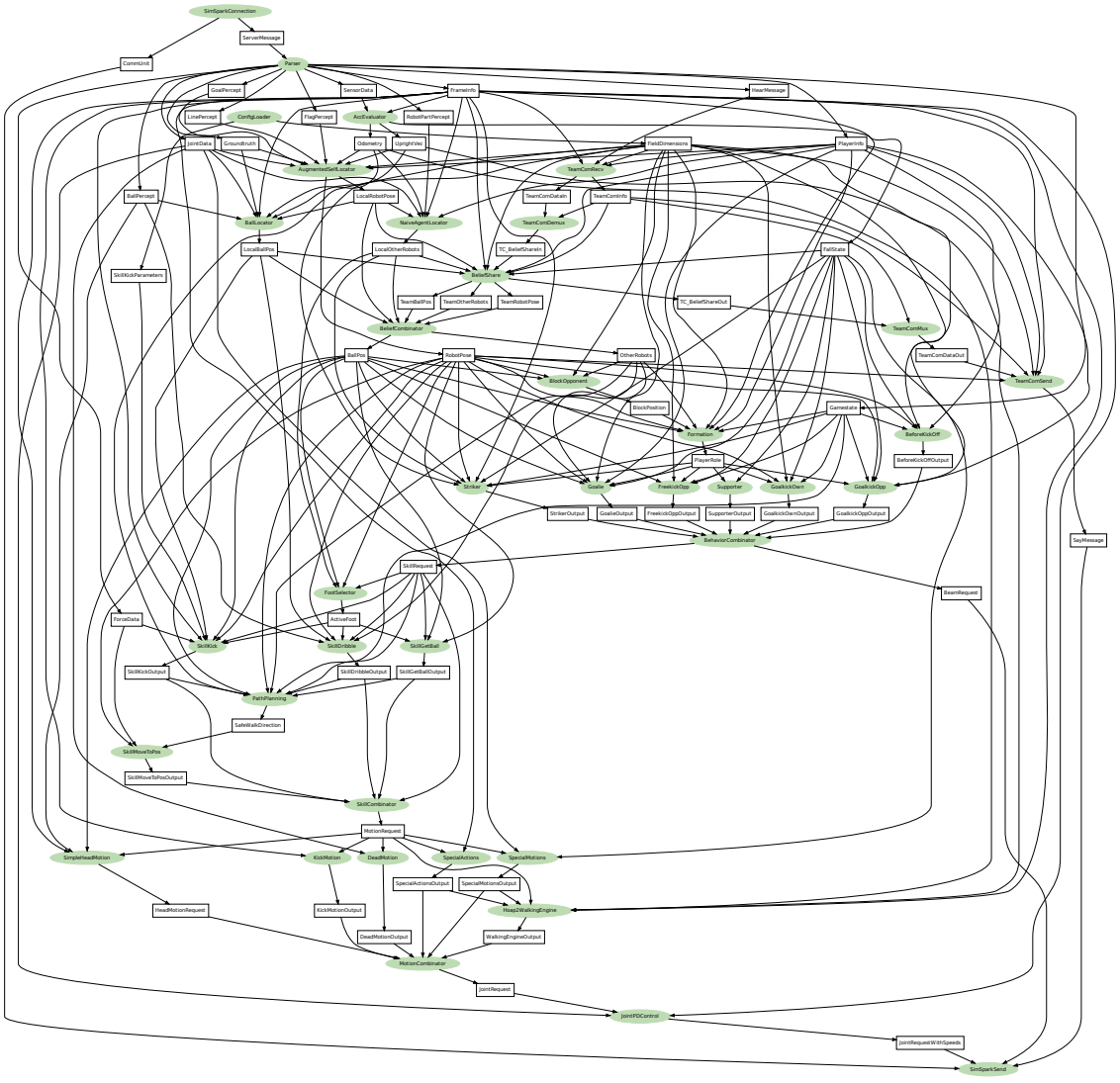


Figure A.3.33: The complete module graph of the SimSpark agent.

The Parser module reads out all information from the server message and provides separate representations. At the end of every cycle the SimSparkSend module uses the connection provided by the SimSparkConnection module to send the joint commands to SimSpark. Since the complete process only depends on the messages of the server that are received with 50 Hz, all modules of the SimSpark agent are in one thread. The complete processing from receiving the server message to sending the joint commands is split into modules for the seperate tasks:

Perception

- **CognitionConfigLoader** → FieldDimensions, SkillKickParameters)

  Provides representations that contain values from config files that never change while the agent is running (e.g. FieldDimensions). It only initializes the representations once.

- **SimSparkConnection** → ServerMessage, SimsparkInfo

  Connects to SimSpark and provides the received server messages as a string. Additionally it provides information about the connection (e.g. socket, time connected).

- **Parser** → FrameInfo, Gamestate, SensorData, GoalPercept, ...

  Parses the server message and provides all received information as separate representations. The parser is generated using flex and bison.

- **TeamComRecv** → TeamComInfo, TeamComDataIn

  Messages between players are send as strings through the server. The length and character set is limited. The Robocanes agent converts the binary data to send into base 64 to encode up to 120 bit information into a string with length 20 using 64 different characters (some more characters are allowed, but they cause problems in the parser).

  The module TeamComRecv decodes the message provides received binary data and for example information about the sender.

- **TeamComDemux** → TC_BeliefShareIn

  Splits messages from other players into separate parts (currently the communi-

cation is only for sharing robot and ball positions).

- **TeamComMux** → TeamComDataOut

  Combines all data to send in one buffer.

- **TeamComSend** → SayMessage

  Encodes the data to send into a 20 character string.

Modeling

- **AccEvaluator** → Odometry

  This module has been used to provide the upright vector and torso pose of the robot using the accelerometer and gyroscope. Most of this is replaced by the module TorsoAngleEstimator. Only the odometry is still provided using information from the gyroscope and walking engine.

- **TorsoAngleEstimator** → UprightVec, TorsoAngles

  Filter for estimating the torso angles using the accelerometer and gyroscope.

- **RobotModelProvider** → FKModel

  Uses forward kinematics to provide cartesian coordinates of all body parts and joints and the center of mass relative to the torso of the robot.

- **GroundContactDetector** → GroundContactState

  Detects whether the robot is on the ground using the force sensors in the feet.

- **TorsoPoseProvider** → TorsoPose

  Combines the FKModel and TorsoAngles to provide the pose of the robots torso in the world coordinate system relative to a reference point between the feet of the robot.

- **FallDetector** → FallState

  Sets the FallState (standing/fallen) using the torso angles. This module also detects whether the robot is sitting.

- **PolarBallPerceptConverter** → BallPercept

  Converts the ball perception in polar coordinates from sent by SimSpark into a position on the field relative to the robot using the TorsoPose.

- **AugmentedSelfLocator** → RobotPose, LocalRobotPose

  Self-localization using a particle filter.

- **BallLocator** → LocalBallPos

  Kalman filter for the ball tracking. This provides only the local ball belief. The overall BallPos representation combines the local belief (LocalBallPos) and received information from the team (TeamBallPos).

- **NaiveAgentLocator** → LocalOtherRobots

  Tracking of other robots from body part perceptions.

- **BeliefShare** → TC_BeliefShareOut, TeamRobotPose, TeamBallPos, Received-BallPos, TeamOtherRobots

  This module provides robot and ball positions received from teammates.

- **BeliefCombinator** → BallPos, OtherRobots

  Combines the local belief with received information. If the confidence is high enough, the local ball belief always has priority over the received ball position.

- **OppAnalyzer** → OppInfo

  This module uses observations of opponent players to obtain information about the opponent. Currently, it only provides an estimated average walk speed of

the opponent and the estimated time until an opponent reaches the ball.

Behavior

- **BlockOpponent** → BlockPosition

  Provides a position to block an opponent robot. This position can be used e.g. by the Formation module as position for supporting players.

- **Formation** → PlayerRole

  The formation sets the player role (goalie, striker, supporter). If the role is supporter, PlayerRole also contains the suggested walk target position.

- **BeforeKickOff** → BeforeKickOffOutput

  Positions the robots in the "before kickoff" game state using the SimSpark beam effector.

- **KickOffOwn** → KickOffOwnOutput

  Behavior for own kickoffs that chooses the direction for the kick.

- **FreekickOpp** → FreekickOppOutput

  Behavior for opponent freekicks, which overwrite the target position for the robots close to the ball to wait and block.

- **GoalkickOpp** → GoalkickOppOutput

  Behavior for opponent goalkick. This module overwrites the target positions for the robots close to the opponent penalty area to either block or get to a position the ball is moved to if the opponent goalie does not manage to kick the ball before the time out.

- **GoalkickOwn** → GoalkickOwnOutput

Behavior to kick the ball out of the own penalty area and avoid blocking the own kick with own robots.

- **CornerkickOwn** → CornerkickOwnOutput

  Behavior for corner kick that kicks towards the opponent penalty area and positions three robots in front of the opponent goal to score.

- **Striker2** → StrikerOutput

  This module chooses the action executed by the striker. If possible, it uses a long kick or kicks to the goal. Otherwise, it dribbles, blocks or uses quick, short kicks.

- **Supporter** → SupporterOutput

  Simple module that always uses the suggested supporter position of the formation modules as walk target.

- **Goalie** → GoalieOutput

  The goalie behavior.

- **BehaviorCombinator** → SkillRequest, BeamRequest

  All output representations of the behaviors contain a skill and beam request. This module checks which behavior outputs are active and provides the requests that have to be used. The skill request contains request as "walk to a given position" or "kick to a given position".

Low-level Behavior / Skills

- **FootSelector** → ActiveFoot

  This module chooses which foot should be used for dribbling based on opponent

positions and where the opponent goal is.

- **PathPlanning** → SafeWalkDirection

  The path planning module uses rapidly exploring random trees (RRT TODO cite) to avoid obstacles and find a path to the target position. Since this path is updated every frame, the SafeWalkDirection representation contains only the direction to walk to in the current cycle.

- **SkillMoveToPos** → SkillMoveToPosOutput

  This module sets walk requests (walk velocities relative to the robot) to reach a given position and orientation.

- **SkillGetBall** → SkillGetBallOutput

  Skill to position at the ball with a given orientation.

- **SkillKick** → SkillKickOutput

  For a given kick target in absolute field coordinates, this module walks to the ball, positions and kicks.

- **SkillDribble** → SkillDribbleOutput

  Module to dribble the ball to a given position on the field.

- **SkillCombinator** → MotionRequest

  Similar to the BehaviorCombinator, the SkillCombinator selects the active skill output and provides the MotionRequest that will be used by the motion modules. The motion

Motions

- **MotionSelector** → MotionSelection

This module selects the active motion based on the motion request and whether it is safe to stop the previously active motion. It can also set factors for each motion for interpolation.

- **WalkingEngine** → WalkingEngineOutput

  The walking engine.

- **SpecialActions** → SpecialActionsOutput

  Module for motions created by keyframe interpolation. Used for stand-up motions and long kicks.

- **KickMotion** → KickMotionOutput

  Module to set the joint request for quick kicks.

- **DeadMotion** → DeadMotionOutput

  Deactivates all joints. Never active in SimSpark agent.

- **SimpleHeadMotion** → HeadMotionRequest

  This module always sets the head angles. Depending on the role and position the robot chooses to look around for self-localization or look at the ball.

- **MotionCombinator** → JointRequest

  Each motion module provides an output that contains a joint request. The MotionCombinator copies the active joint request or interpolates between joint request depending on the MotionSelection.

- **JointPDControl** → JointRequestWithSpeeds

  This modules uses a PD controller for each joint to reach the requested joint angles.

- **SimSparkSend** Module to send the joint commands, messages for commu-

nication and beam commands to simspark using the socket provided by the SimSparkConnection module in the beginning.

The general structure of the RoboCanes agent is very similar for the 3D Simulation League and the Standard Platform League, but there are several additions that help with working with the RoboCanes agent in Simspark. The beam command for SimSpark can only move the robot that sends the beam command, therefore the agent is also able to connect to the monitor port of SimSpark (the same way as RoboViz) to send commands to move any robot or the ball. This is added as plugin in the framework such that it is available in all modules through the "monitor" object in the module base class. Furthermore, the parser can provide the representation "Groundtruth" with exact positions of all robots and the ball. The module "GroundtruthModule" can provide RobotPose or BallPos to let the complete behavior use groundtruth positions instead of the estimated positions.

The SimSpark connection module has several options in the config file. Very useful for learning tasks in SimSpark are the options to let the agent start SimSpark and reconnect automatically. The SimSpark instance started by the agent will randomly choose free ports for the communication such that multiple agents can run on one computer, each in an own instance of the simulator. If the simulator is killed or crashes, it restarts it and reconnects.

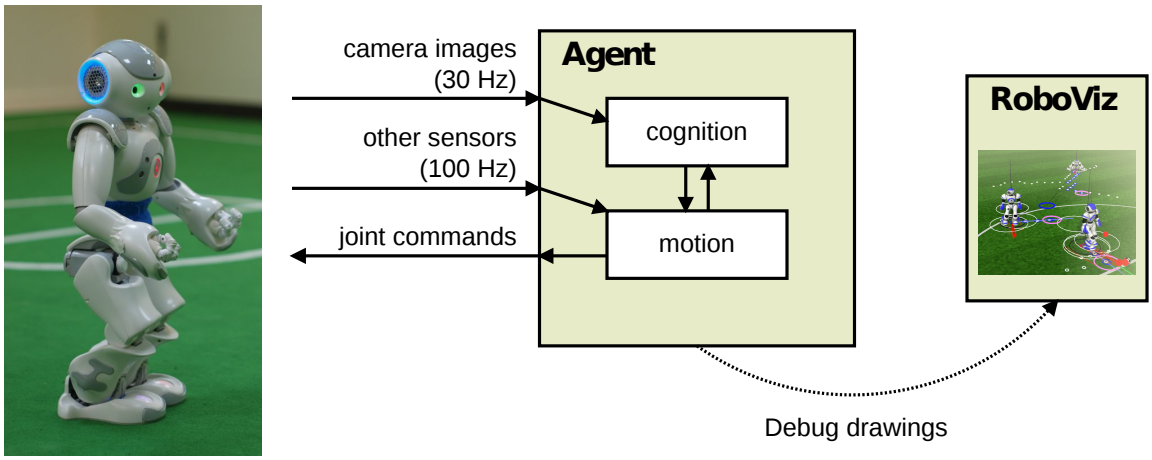


Figure A.4.34: The connection of the agent to the physical robot and RoboViz.

A.4 SPL Agent

The SPL agent runs on the robot and directly reads sensor values and controls the joints through NAOqi (software from Aldebaran) with 100 Hz. At the same time it needs to capture images from the NAO's camera for image processing with 30 Hz. Therefore, there are two threads running in the RoboCanes SPL agent, a cognition and a motion thread. Figure A.4.34 shows this setup. The agent can still connect to RoboViz to send drawings.

The streaming of representations from one agent to another through the network module implemented in the framework allows to connect an agent running for example on a laptop to a robot and stream representations as in figure A.4.35. This connection helps with debugging, since modules can show debug outputs or the complete agent can even run in a debugger (with stopping the agent or using single stepping) while it used images or sensor values directly from the robot.

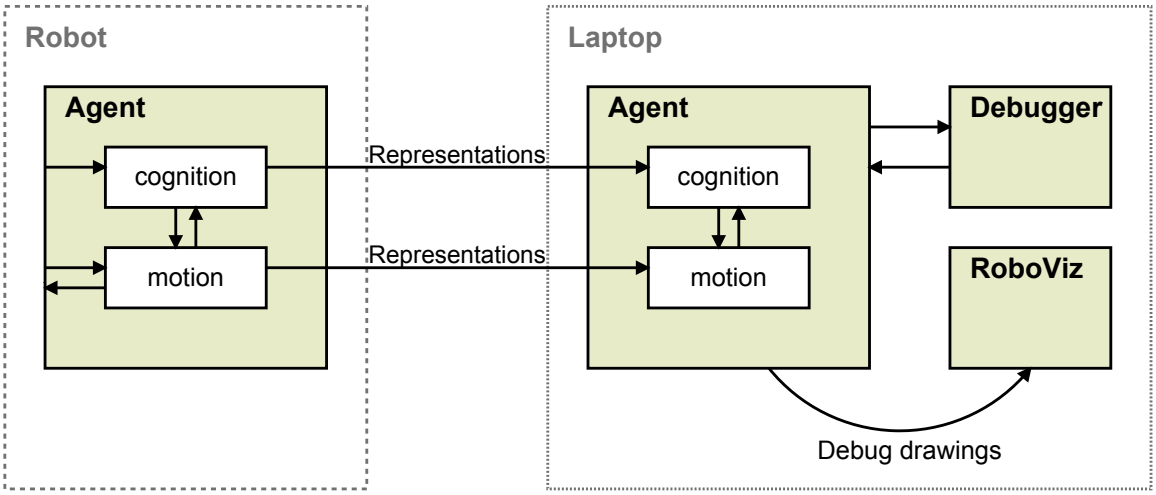


Figure A.4.35: The network module of the framework allows streaming representations from the robot to another agent.

A.4.1 Differences to SimSpark Agent

Since all sensor values from NaoQi are read with 100 Hz and the camera images are read with 30 Hz, the processing has to be done in two separate threads. The figures A.4.36 and A.4.37 show the module graphs for the two threads of the SPL agent.

The overall structure of the SPL agent is very similar to the SimSpark agent. Several modules are used for both leagues only with different values in the config files, e.g. the formation module, some behaviors, the path planning, belief share and combinator modules.

The cognition thread starts the execution with a module that provides camera images and there are several additional modules for the image processing and SPL specific tasks:

- **NaoCamera** → V4L2Image

  The camera images are read using Video4Linux2 and provided as raw image buffers in the V4L2Image representation.

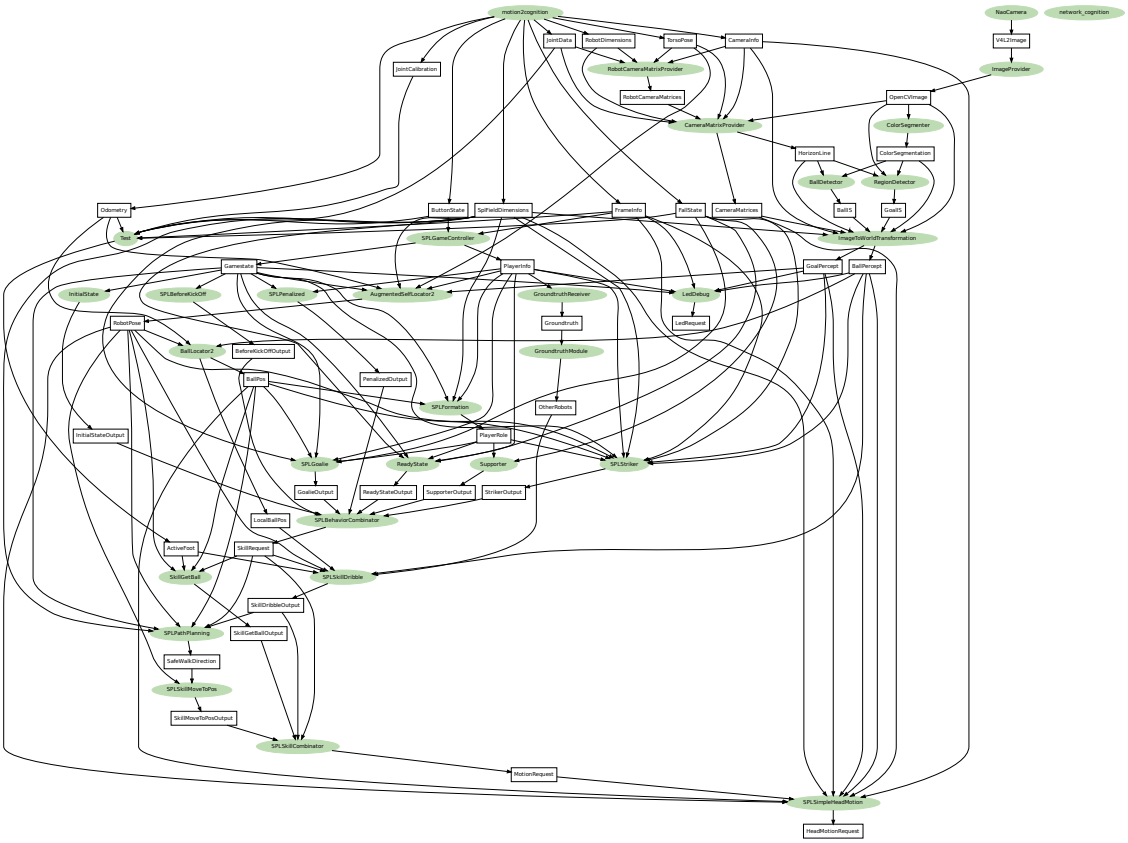


Figure A.4.36: Module graph for the cognition thread.

- **FullResolutionImageGridProvider** → ImageGrids

  Module that provides an image grid that can change the resolution used in different parts of the image (lower resolution for areas that are close, high resolution to see details far away). Currently, the full resolution is used for the entire image, since the image processing is fast enough.

- **RunLengthImageProvider** → RunLengthImages

  Module that provides the run length encoded images.

- **BallPerceptorRLE2** → BallPercept

  Detects the ball in the RLE image based on color, size and roundness.

- **GoalPerceptorRLE2** → GoalPercept

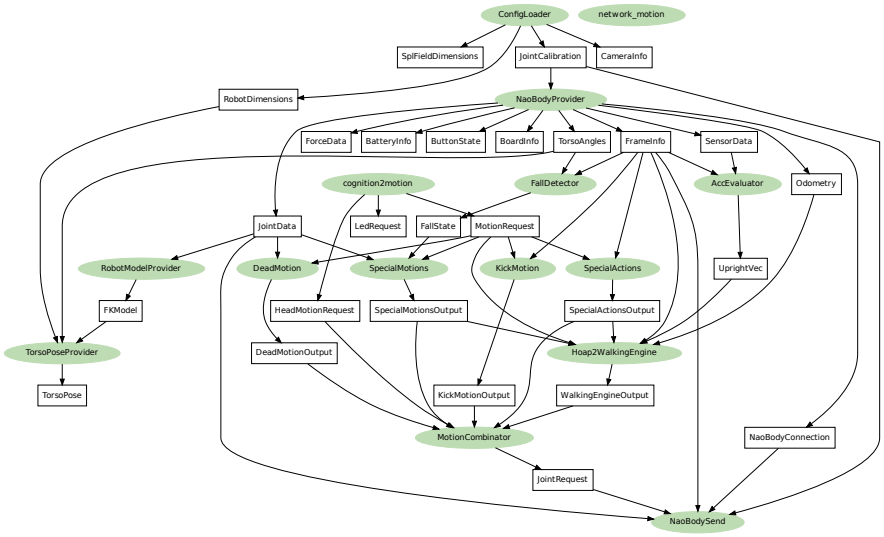


Figure A.4.37: Module graph for the motion thread.

Detects the goal posts in the RLE image.

- **LinePerceptorRLE2** → LinePercept, InvalidLines

  Line detection using the RLE images. Lines that were detected, but filtered out because they did not match the expected width are provided as InvalidLine.

- **RobotCameraMatrixProvider** → RobotCameraMatrices

  This module provides matrices for the transformation from the torso to the camera coordinate systems.

- **CameraMatrixProvider** → CameraMatrices

  The camera matrices represent the transformation from the reference point between the feet of the robot in the world coordinate system to the camera coordinate systems.

- **SPLTeamComRecv** → TeamComInfo, TeamComDataIn

  Replaces the module TeamComRecv in the SimSpark agent. No special encoding is needed for the SPL. Instead the message can be attached as binary data

to the SPL standard message. This module opens a UDP socket, receives messages from other robots and provides the data in the same way as the SimSpark agent.

- **SPLTeamComSend** → r

  This module attaches the data to send to a SPL standard message, fills in the mandatory values of the message and sends it to the other robots through a UDP socket (broadcast or multicast).

- **SPLMessageReader** → TeamBallPos, TeamRobotPose, TeamOtherRobots, GoalieBallPos

  This module has can be used in SPL drop-in games to use the information stored in the SPL standard messages of robots from other teams. The RoboCanes agents usually use only the data from the binary buffer at the end of the message.

- **SideModelProvider** → SideModel

  Module to keep track of the half the robot is in to avoid problems with the localization when robots get turned and start localizing on the wrong half.

- **LedDebug** → LedRequest

  This module sets the colors of all LEDs of the NAO. It reads information from several representations and shows for example whether the ball is seen or the team communication is working.

- **SPLGameController** → Gamestate, PlayerInfo

  This module receives packets from the SPL game controller and sets the GameState representation accordingly. If no packets are received, it switches to the manual button interface.

- **AudioFFT** → MicFrequencies

  Records audio data from the microphones, runs a fast fourier transformation and provides the frequencies.

- **AudioPredictionModule** → AudioPredictionRepresentation

  Module for whistle detection used for the kickoff.

- **RCMModule** → RCMData

  This module sends information about the robot's status to RCM (see section A.6.2). It sends the information over ethernet and provides a representation with the same data to be attached to the SPL standard message and sent over wifi.

- **JoystickModule** → JoystickData, JoystickBehaviorOutput

Once the image processing modules provide the perception representations, the remaining modules for the modeling, communication and behavior are structured similar to the SimSpark agent. However, the cognition thread ends after the MotionRequest is provided. All motion modules run in the motion thread. This thread accesses the motion request from the cognition thread, but runs independently with 100 Hz and a higher priority to guarantee smooth motions even if other threads use too much CPU time.

The sensor values and joints of the NAO can only be accessed through the software NaoQi from Aldebaran. To separate the RoboCanes framework completely from NaoQi and avoid dependecies to the NaoQi SDK, a single NaoQi module is added on the robot, which communicates with the RoboCanes agent through shared memory.

It writes all sensor values to the shared memory and reads the joint target angles written by the agent. In the agent the modules `NaoBodyProvider` and `NaoBodySend` read and write to this shared memory.

The following modules are added for the SPL agent in the motion thread:

- **NaoBodyProvider** → RawJointData, ForceData, ButtonState, BatteryInfo

  Reads all sensor values and provides separate representations, e.g. for joint positions, the force sensors, the battery information, or button states.

- **USControl** → USRequest

  This module controls the sonars. The USRequest representation contains the request to fire the sonars.

- **MultiObstacleModelProvider** → ObstacleModel

  Uses sonar perceptions to detect obstacles in front of the robot.

- **ArmContactProvider** → ArmContact

  Detects when an arm is pushed by an obstacles. The ArmContact representation is used to switch of the arm joints.

- **StandMotion** → StandMotionOutput

  This module interrupts the stand-up special actions as soon as the feet are below the center of mass. It completes the stand-up motion using inverse kinematics.

- **HeadMotionControl** → HeadJointRequest

  Module to move the head smoothly with a given maximum speed towards target pan and tilt angles provided by the cognition thread with a lower framerate.

- **IKArmMotion** → ArmMotionOutput

  Module for arm motions using interpolation in cartesian space and inverse kine-

matics.

- **NaoBodySend**

  Writes the values from the joint request to the shared memory for the NaoQi module.

A.4.2 Module Configurations and Agent Start Scripts

The directory `binNao` contains several scripts to start the agent with different configurations for connecting to the robot and streaming representations from the agent on the robot to the agent running locally. Executing "./testRLE.sh 192.168.1.101" for example will connect to an agent running at that IP and show several debug outputs of the image processing as in figure A.4.38. Only the RLE compressed image and representations important for the image processing (e.g. the camera matrices) are streamed. This allows the image processing modules to run locally with the same result as on the robot. The agent has to be compiled with the configuration set by "./configureNaoLocal.sh" in the source directory to be able to show the images. Another example is the script "testRemoteSensorsNoCam.sh" which streams most sensor values, but not the camera images. This modules configuration is sufficient to show for example debug drawings of the perceptions in RoboViz.

A.5 Scripts

This section describes the most important scripts in the RoboCanes repository, e.g. for automatically running 3D Simulation games in SimSpark or for configuring a NAO

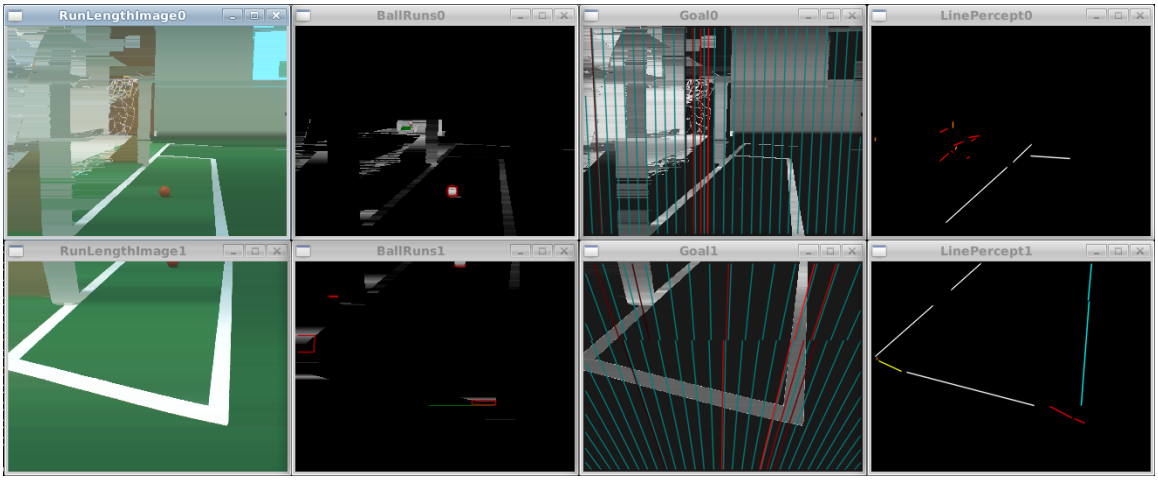


Figure A.4.38: Debugging outputs of the image processing modules.

robot to run the RoboCanes agent.

SimSpark Binary Package

The script `scripts/createBinaryPackage.sh` creates a tar.gz file in /temp/ that contains all files from the bin directory that are needed to run the agent. It removes unnecessary files, such as .svn directories or additional start scripts. Furthermore, it removes local files that are not under version control (never committed to the SVN). Those files are not shown as modifications by "svn st", but could change the behavior of the agent. The archive created by `scripts/createBinaryPackage.sh` contains a clean version of the agent, which corresponds exactly the current SVN version if the command "svn st" does not show local changes. This script should be used to create a clean agent package used in competitions of the 3D simulation league.

SimSpark Autoplay

The directory `scripts/autoplay` contains several scripts for running full games in SimSpark automatically. The script `startGame.sh` starts SimSpark and two teams (using the standard start script). It automatically sends a kick off signal to SimSpark to start the game, starts the second half and stops all processes after the game finished. The final score is written to `games.log`. If the server crashes or hangs, the script kills all processes. The `runOppList.sh` can be used to games against different opponent binaries automatically. These scripts can be used to run games continiuously on a headless system. The scripts `getAvgSteps.sh` and `getAvgForEachOpp.sh` read the results from the log file and show average scores.

NAO Robot Initialization

There are several configurations that need to be changed on a new robot or after a factory reset. The script `./scripts/nao/naorootInit.sh` helps with this process. The complete configuration can be done in a few minutes by executing the following steps:

- Make sure all files are updated (svn up) and run `./configureNaoRelease.sh` and `make` in the source directory to compile the release version of the agent for the NAO. This binary will by copied to the robot.
- Connect with the NAO directly with an ethernet cable.
- Choose the connection type “direct link” in the network manager (on the laptop) to get an IP address assigned. If the connection works, it should be possible to

ping the robot using “ping Nao.local”.

- Change to `trunk/scripts/nao` and execute `./naorootInit.sh Nao.local`. Follow the instructions. This script will connect to the robot several times to change passwords and allow SSH access as user root. It copies all files from the directory `scripts/nao/naoroot` to `/root/` on the robot. This includes the binary, libraries, wifi scripts and files needed for configuring the robot. The last step is to execute the script `configureNao.sh` which changes several configurations on the robot (use correct arguments for host name and IP!).

- Reboot the robot.

All changes in system configuration files or any other modification in the NAO’s system should be added to the `configureNao.sh`, such that it can be applied automatically to all robots and they are configured the same way. Currently, the script applies the following changes to the NAO:

- Setting the hostname.
- Link to /root/autostart.sh in /etc/local.d/.
- IP for wifi in /root/wifi_lab.sh.
- Changing autoload.ini of NaoQi and setting the permission to deactivate the NaoQi fall manager.
- Modifications in sudoer file.
- Installation of init.d scripts for robocanes agent in /root/bin/.
- Deactivate DNS lookup in ssh server config (causes delays when connecting through SSH).

- Adding a kernel module for joystick support (joydev.ko).
- Prevent NaoQi from opening microphones. Exclusiv access would prevent agent from using the microphones with Alsa).
- Installation of files from ./tools (e.g. the program "screen").
- IPv4 ethernet configuration.

The script is implemented such that executing it multiple times will not cause errors. Therefore, it can be executed to update robots and apply changes if when the script is changed.

NAO Helper Scripts

There are various helper scripts in `scripts/nao/` for stopping or restarting the RoboCanes agent on a robot, reboot or shutdown a robot or updating the agent. These scripts can be used for individual robots (e.g. `./agentRestart.sh nao1`, with nao1 defined in /etc/hosts) or they can be combined with the script `forall.sh`. For example, "./forAll.sh naorootUpdateBinary.sh -y" updates the binary in `/root/binNao` on all robots.

The script "copyBinary.sh" can be used to copy a binary from `binNao` for testing to a directoy in `/home/nao/`. It uses rsync to avoid copying the complete `binNao` directory for every test.

A.6 Utilities

A.6.1 RoboCanes Debugger (rdb)

The debugger rdb is a stand-alone program that connects through TCP to the debug interface of the RoboCanes agent. The module “DebugModule” has to be activated in the agents configuration to open the port that accepts rdb connections.

As mentioned before, debug commands can be send as a simple string to the agent (e.g. with nc). It is not required to use an extra debugger, but it can add extra functionalities.

Since the debug interface sends information already in a human-readable text format, the debugger does not need any information about the objects or data that is streamed. Therefore, rdb does not have to be recompiled when the available values or debug responses in the agent change.

The debugger rdb has to be started with ”rdb [ip-address]”. If no address is given, it connects to localhost. Figure A.6.39 shows the rdb console (using the ncurses library) with a plot window (using OpenCV). Commands typed into the rdb console are forwarded to the debug interface of the agent rdb is connected to. Additionally, the rdb console provides the following functions:

- The console stores a history of all previously executed commands, which is saved to a file when rdb is closed.

- The file ”alias.cfg” contains definitions of aliases in the form `"A|B"`. Every string ”A” in the input will be substituted by ”B”. For example, the line `" r:|`

`representation:"` makes it possible to write "get r:BallPos" instead of "get representation:BallPos".

- All watched variables are printed and updated in the top of rdb.
- There can be more complex commands implemented in rdb, that are using values received by the basic commands. The commands currently implemented are:
  - `plot value [windowName [numValues [repaintInterval]]]` The command "plot" sends a "watch" command to the agent and shows the received values in a plot window. The only required parameter is the value to show. Additional parameters control the title of the window, the number of values shown and how often the plot is updated. Multiple values can be plotted in one window by using the same window title.

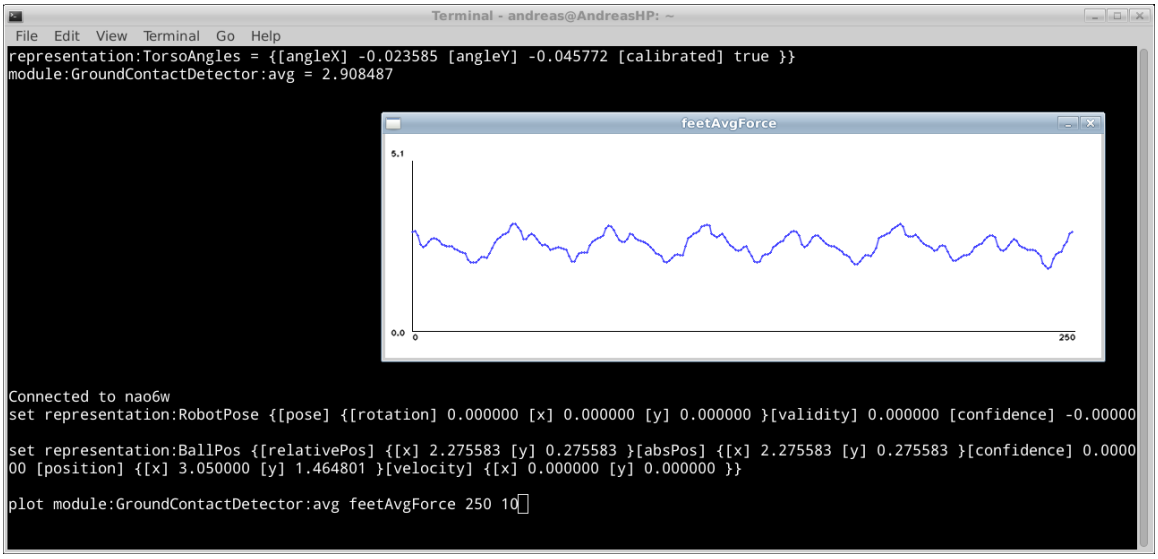


Figure A.6.39: RoboCanes debugger rdb that shows some watched variables received representation values and a plot.

Planned extensions include for example parsing more complex streamable variables and accessing only part of it, e.g. for plotting. Furthermore, an automatic completion in the console would be very helpful. Lists of all registered variables and debug responses are available in the agent. Sending these to the debugger would allow an automatic completion in rdb, similar to the behavior of the tab key in a linux shell. Available commands could be send to rdb including a syntax definition (e.g. as "get STREAMABLE"). Additionally, the agent could send all strings that can be used as STREAMABLE parameter.

A.6.2 RoboCanes Manager (rcm)

The RoboCanes Manager (rcm) is a tool, that receives status packets from the robot and shows an overview of those values for all robots in a Qt window. The agents running on the robots need to execute the module "RCMModule", which sends the status packets via UDP broadcasts or multicasts. All settings can be found in the config file for the RCMModule. In competitions, it might be important to change e.g. the broadcast ip address.

Since the communication between the agent is limited to the SPL standard message format, the RCM status packets need to be embedded in the binary buffer at the end of the standard messages.

Figure A.6.40 shows the rcm window, which shows some important values for each robot, such as the battery status, the temperature of the hottest joint, the team color, jersey number (ID of the robot in the game controller) and the current role. The box

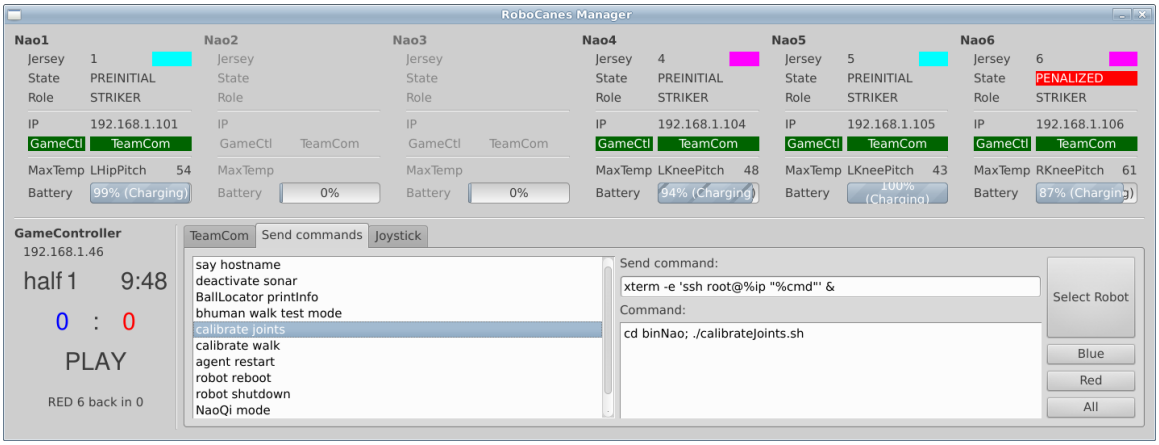


Figure A.6.40: RoboCanes Manager GUI. Four robots are running an agent, two in the red and two in the blue team.

"TeamCom" is only green, if communication packets from another robot are received. The left bottom corner shows the game controller state. Only if rcm receives game controller packets, the colored boxes for the robots are activated and indicate with green or red whether each robot receives game controller packets.

The area below shows on the left side the status received from an SPL GameController. The remaining space is used for showing the team communication between the agents and for a GUI for sending commands to the robots. It shows a list of commands that can easily be send to robots using the detected ip address, e.g. for activating debug modes or restarting the agents. The commands can be edited in a configuration file and new commands can be added. These commands can be shell command that are executed on the robot or debugger commands that are sent to the agent's debug interface. An example use command is "shutdown robot", which can quickly be send to all robots and executes the bash command "halt" everywhere.

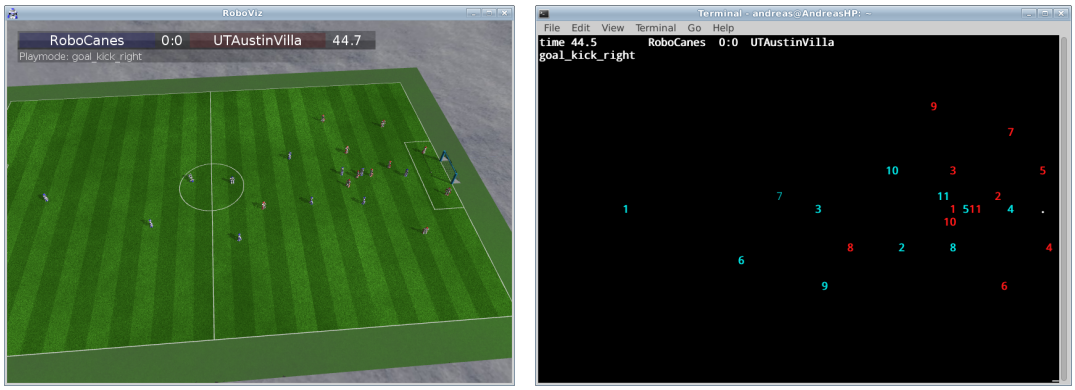


Figure A.6.41: A 11 vs 11 game in SimSpark shown in RoboViz (left) and the 2D monitor (right). Besides uses significantly less resources, the 2D view can also give a better overview over the team behavior.

A.6.3 2D Text-based SimSpark Monitor (simplemonitor)

Simplemonitor is a fast and lightweight SimSpark monitor that runs in a terminal. The robot and ball positions are shown using the terminal UI library ncurses (see Fig. A.6.41). This monitor was developed as a simple alternative to RoboViz that needs less resources and can even be started in an SSH session.

The default monitor connection sends a scene graph to the monitor (e.g. RoboViz). This scene graph is readable text and contains geometric information e.g. about separate body parts of the robots. Therefore, connecting a monitor such as RoboViz to SimSpark needs a fast network connection. Also, parsing and processing the received scene graph information in RoboViz needs significant CPU time.

The simplemonitor uses an own protocol. It receives very simple packets from SimSpark that only contain robot and ball positions. This could only be realized by adding a module to SimSpark that opens a port, accepts connections and sends all positions in the correct format. It sends all positions only five times per second.

(much slower than realtime)

An 11 vs 11 game running in real-time produces approximately 1.7 MB/s data sent through the monitor port of SimSpark to RoboViz. On an Intel i7-620m dual-core CPU, RoboViz uses about 85% CPU time (of one core) to parse the data of a full game and render the field and the robots. If the simulation is running on the same machine, RoboViz slows down the simulation.

Simplemonitor uses less then 1% CPU and receives 4 kB/s from SimSpark to show a full 11 vs 11 game.

Since the simplemonitor uses ncurses, it can directly be started in an SSH session to quickly check whether a simulation is running correctly or even to watch a full game. However, less bandwidth would be used by running simplemonitor locally and connecting to the server ("simplemonitor serverIP") or by tunneling the simplemonitor port from a remote server through SSH ("ssh -L3201:127.0.0.1:3201 serverIP" and "simplemonitor").

Installation instructions for the simplemonitor SimSpark patch can be found in `trunk/utils/SimpleMonitor/INSTALL`. Module in SimSpark has to be initialized in rcssserver3d.rb, which might be in `/.simspark` and not updated when applying the patch.

A.6.4 SimSpark Groundtruth

SimSpark can provide groundtruth to agents through options of the `RestrictedVisionPerceptor`. However, this only sends the own position of an agent and the ball position to each robot. It does not include other robots or

torso angles.

The RoboCanes SimSpark groundtruth is a plugin for SimSpark that adds all groundtruth information to the server messages. It the positions of all robots, the ball and own torso angles to every agent. Therefore, this can be used to evaluate the errors in the perception and modeling, such as the torso angle estimation or robot perception.

Installation instructions for the groundtruth pluging can be found in `trunk/utils/simsparkGroundtruth/INSTALL`. The plugin consists of one class that is added in the simspark sources in `rcssserver3d/plugin/soccer/`. The patch adds entries for these files to CMake, adds the plugin to the SimSpark framework and activates the groundtruth in `nao_hetero.rsg`. If the plugin works correctly, the server message contains the groundtruth information as shown in figure A.6.42. These values are parsed by the RoboCanes agent and written to the groundtruth representation.

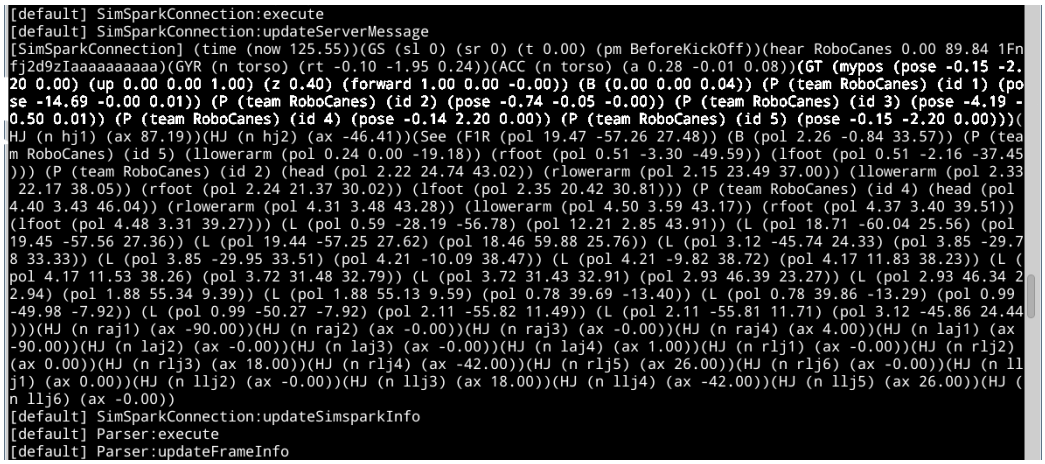


Figure A.6.42: The server message received by the RoboCanes agent (printed by "./robocanes -v"). The groundtruth information is in the parentheses starting with GT.

The groundtruth is only sent if the team name is "RoboCanes" to avoid causing

problems when playing against other teams that would not be able to parse the RoboCanes groundtruth.

A.6.5 Webots Extensions

Webots simulates a NAO robot that runs NaoQi, such that the agent used on the physical robots can directly connect to Webots as if it was running on the robot. This means that an agent for Webots needs to use completely different modules for the communication with the simulator. However, the framework and most available representations are the same. An experimental setup for an optimization or learning task in SimSpark can directly be used in Webots.

The only missing components are the groundtruth information and a beam command. These can be added to Webots using the RoboCanes supervisor controller for Webots. The directory `trunk/utils/webots` contains an example Webots world with an SPL soccer field and a NAO that starts the RCSupervisor from the controller directory.

The supervisor controller reads out the position and torso angles of the NAO. Those values are sent as a UDP multicast with TTL 0 (only within localhost), since the agent has to run on the same host to connect to the shared memory of librobocanes running in the NAOqi of the simulated NAO.

The agent module that receives this information has to be activated with "GroundtruthReceiver Groundtruth" in the module configuration of the agent. The same packets could also be sent by other groundtruth systems, e.g. an additional

robot detection for the physical robots using a high resolution camera over or next to the field.

A.6.6 MPI Tunnel

The MPI tunnel is a tool that forwards TCP connections through MPI. This allow programs with a server-client architecture that uses TCP sockets to run on a cluster and communicate through MPI. Figure A.6.43 shows the communication between different processes using the MPI tunnel. For the TCP server and client the tunnel is completely transparent. When a client connects to the MPI tunnel, the tunnel create a new connection to the server. All data sent by the client or server is forwarded. If the connection is closed on one side, the MPI tunnel closes the connection on the other side. The MPI tunnel is implemented in the

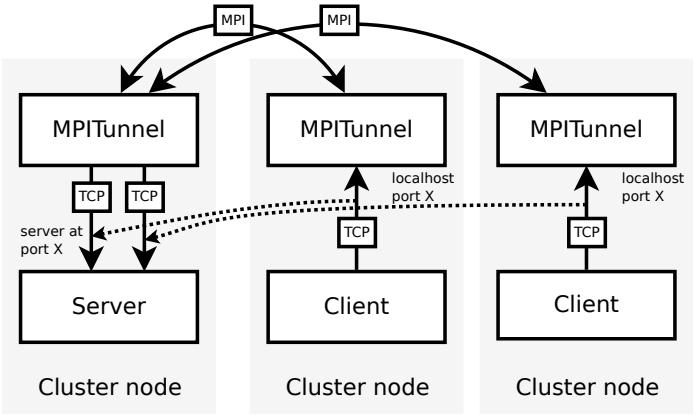


Figure A.6.43: The MPI tunnel forwards TCP connections from clients to the server. The clients can use localhost as server address. The tunnel is completely transparent. For the clients and server the connection seems to be a direct TCP connection (dashed arrows).

file `utils/OptimizationServerFiles/mpitunnel.cpp` and has to be compiled using `mpic++`. Starting the client with "mpirun ./mpitunnel ./server "./client localhost"

12345" will execute "./server" on the first node, "./client localhost" an every other node and forward connections to port 12345 through MPI.

A.6.7 Parallel Optimization

The directory `src/tools/` contains a base class used for different parameter optimizations in the agent. There are implementations of CMA-ES, PSO and xNES using the same parameter optimization interface, such that optimizations can easily switch between different algorithms. For parallel optimization an optimization client can be used in the agent. This client uses the same interface, but connects to a server that runs the actual optimization and distributes parameter sets to the clients. The clients evaluate the parameters and send the fitness value to the server.

The optimization server and several scripts for running an optimization on a MPI cluster using the server and MPI tunnel can be found in `utils/OptimizationServerFiles`.

References

[1] Zohaib Aftab, Thomas Robert, and Pierre-Brice Wieber. Ankle, hip and stepping strategies for humanoid balance recovery with a single model predictive control scheme. In *Proceedings of the IEEE-RAS International Conference on Humanoid Robots*, 2012.

[2] Heni Amor, Erik Berger, David Vogt, and Bernhard Jung. Towards Responsive Humanoids: Learning Interaction Models for Humanoid Robots. *International Conference on Machine Learning*, pages 1–4, 2011.

[3] Tamim Asfour and Rüdiger Dillmann. Human-like motion of a humanoid robot arm based on a closed-form solution of the inverse kinematics problem. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 2, pages 1407–1412. IEEE, 2003.

[4] Cristian Canton-Ferrer, Josep R. Casas, and Montse Pardàs. Marker-based human motion capture in multiview sequences. *EURASIP J. Adv. Signal Process*, 2010:73:1–73:11, 2010. ISSN 1110-8657.

[5] A. Carlisle and G. Dozier. An off-the-shelf PSO. In *Proceedings of the Workshop on Particle Swarm Optimization*, pages 1–6. Indianapolis, Purdue School of Engineering and Technology, IUPUI, 2001.

[6] Sonia Chernova and Manuela Veloso. An Evolutionary Approach to Gait Learning for Four-Legged Robots. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings.*, volume 3, pages 2562–2567. IEEE, 2004.

[7] Christine Chevallereau, Dalila Djoudi, and Jessy W Grizzle. Stable Bipedal Walking with Foot Rotation through Direct Regulation of the Zero Moment Point. *IEEE Transactions on Robotics*, 24(2):390–401, 2008.

[8] Hyeung-Sik Choi, Sam sang You, Sang-Jun Lee, Byung-Guk Kim, Geun-Wha Lim, Dong-Yeon Ko, and Woong-Ju Moon. A study on a new biped robot supporting heavy weight. In *SICE-ICASE, 2006. International Joint Conference*, pages 1180–1184, 2006. doi: 10.1109/SICE.2006.315362.

[9] Jeffrey B Cole, David B Grimes, and Rajesh PN Rao. Learning full-body motions from monocular vision: Dynamic imitation in a humanoid robot. In *In-*

telligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on, pages 240–246. IEEE, 2007.

[10] Daniel D Corkill. Blackboard systems. *AI expert*, 6(9):40–47, 1991.

[11] H. Dallali, P. Kormushev, Z. Li, and D. Caldwell. On global optimization of walking gaits for the compliant humanoid robot, coman using reinforcement learning. *Journal of Cybernetics and Information Technologies*, vol. 12, no. 3: pp. 39–52, 2012.

[12] Mike Depinet, Patrick MacAlpine, and Peter Stone. Keyframe Sampling, Optimization, and Behavior Integration: Towards Long-Distance Kicking in the RoboCup 3D Simulation League. In H. Levent Akin, Reinaldo A. C. Bianchi, Subramanian Ramamoorthy, and Komei Sugiura, editors, *RoboCup-2014: Robot Soccer World Cup XVIII*, Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, 2015.

[13] Mike Depinet, Patrick MacAlpine, and Peter Stone. Keyframe sampling, optimization, and behavior integration: Towards long-distance kicking in the robocup 3d simulation league. *RoboCup-2014: Robot Soccer World Cup XVIII*, 2015.

[14] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphvizopen source graph drawing tools. In *Graph Drawing*, pages 483–484. Springer, 2002.

[15] Ka-Ling Fok, Tien-Tsin Wong, and Man-Leung Wong. Evolutionary Computing on Consumer Graphics Hardware. *Intelligent systems, IEEE*, 22(2):69–78, 2007.

[16] I. Gaiser, S. Schulz, H. Breitwieser, and G. Bretthauer. Enhanced flexible fluidic actuators for biologically inspired lightweight robots with inherent compliance. In *Robotics and Biomimetics (ROBIO), 2010 IEEE International Conference on*, pages 1423–1428, 2010. doi: 10.1109/ROBIO.2010.5723538.

[17] T. Glasmachers, T. Schaul, and J. Schmidhuber. A natural evolution strategy for multi-objective optimization. In *Parallel Problem Solving from Nature–PPSN XI*, pages 627–636. Springer, 2011.

[18] David Gouaillier, Vincent Hugel, Pierre Blazevic, Chris Kilner, Jerome Monceaux, Pascal Lafourcade, Brice Marnier, Julien Serre, and Bruno Maisonnier. The nao humanoid: a combination of performance and affordability. *CoRR*, abs/0807.3223, 2008.

[19] Colin Graf and Thomas Röfer. A Center of Mass Observing 3D-LIPM Gait for the RoboCup Standard Platform League Humanoid. In *RoboCup*, pages 102–113, 2011.

[20] David B Grimes and Rajesh PN Rao. Learning Nonparametric Policies by Imitation. In *International Conference on Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ*, pages 2022–2028. IEEE, 2008.

[21] David B. Grimes, Daniel R. Rashid, and Rajesh P. Rao. Learning Nonparametric Models for Probabilistic Imitation. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*, pages 521–528. MIT Press, Cambridge, MA, 2006. URL `http://books.nips.cc/papers/files/nips19/NIPS2006_0615.pdf`.

[22] David B. Grimes, Daniel R. Rashid, and Rajesh P. N. Rao. Learning nonparametric models for probabilistic imitation. In *Advances in Neural Information Processing Systems (NIPS)*. MIT Press, 2006.

[23] Gutemberg B. Guerra-filho. Optical motion capture: Theory and implementation. *Journal of Theoretical and Applied Informatics (RITA)*, 12:61–89, 2005.

[24] Douglas Hakkarinen, Tracy Camp, Zizhong Chen, and Allan Haas. Reduced Data Communication for Parallel CMA-ES for REACTS. In *20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2012*, pages 97–101. IEEE, 2012.

[25] Y. Hanazawa, H. Suda, and M. Yamakita. Analysis and experiment of flat-footed passive dynamic walker with ankle inerter. In *Robotics and Biomimetics (ROBIO), 2011 IEEE International Conference on*, pages 86–91, 2011. doi: 10.1109/ROBIO.2011.6181267.

[26] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

[27] Matthias Hebbel, Ralf Kosse, and Walter Nistico. Modeling and Learning Walking Gaits of Biped Robots. In *Proceedings of the Workshop on Humanoid Soccer Robots of the IEEE-RAS International Conference on Humanoid Robots*, pages 40–48, 2006.

[28] Young-Dae Hong, Chang-Soo Park, and Jong-Hwan Kim. Human-like stable bipedal walking with a large stride by the height variation of the center of mass using an evolutionary optimized central pattern generator. In *IECON 2011-37th Annual Conference on IEEE Industrial Electronics Society*, pages 177–182. IEEE, 2011.

[29] Matthew Howard, Stefan Klanke, Michael Gienger, Christian Goerick, and Sethu Vijayakumar. Behaviour Generation in Humanoids by Learning Potential-Based Policies from Constrained Motion. *Applied Bionics and Biomechanics*, 5(4):195–211, 2009.

[30] Weiwei Huang, Chee-Meng Chew, Yu Zheng, and Geok-Soon Hong. Pattern generation for bipedal walking on slopes and stairs. In *Humanoid Robots, 2008.*

Humanoids 2008. 8th IEEE-RAS International Conference on, pages 205–210. IEEE, 2008.

[31] Yan Huang, Qining Wang, Guangming Xie, and Long Wang. Optimal mass distribution for a passive dynamic biped with upper body considering speed, efficiency and stability. In *Humanoid Robots, 2008. Humanoids 2008. 8th IEEE-RAS International Conference on*, pages 515–520, 2008. doi: 10.1109/ICHR.2008.4756006.

[32] Vasudevan Jagannathan. *Blackboard architectures and applications*, volume 3. Elsevier, 1989.

[33] Nick Jakobi, Phil Husbands, and Inman Harvey. Noise and the reality gap: The use of simulation in evolutionary robotics. In *Advances in artificial life*, pages 704–720. Springer, 1995.

[34] Jui-fang Chang and Shu-chuan Chu and John F. Roddick and Jeng-shyang Pan. A parallel particle swarm optimization algorithm with communication strategies. *Journal of Information Science and Engineering*, pages 809–818, 2005.

[35] S. Kajita, F. Kanehiro, K. Kaneko, K. Yokoi, and H. Hirukawa. The 3D linear inverted pendulum mode: a simple modeling for a biped walking pattern generation. In *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, volume 1, pages 239–246 vol.1, 2001. doi: 10.1109/IROS.2001.973365.

[36] S. Kajita, F. Kanehiro, K. Kaneko, K. Fujiwara, K. Harada, K. Yokoi, and H. Hirukawa. Biped walking pattern generation by using preview control of zero-moment point. In *Robotics and Automation, 2003. Proceedings. ICRA '03. IEEE International Conference on*, volume 2, pages 1620–1626 vol.2, 2003. doi: 10.1109/ROBOT.2003.1241826.

[37] Shuuji Kajita. Keynote presentation: Overview of ZMP-based Biped Walking. Dynamic Walking 2008. Delft, Netherlands.

[38] Shuuji Kajita, Fumio Kanehiro, Kenji Kaneko, Kiyoshi Fujiwara, Kazuhito Yokoi, and Hirohisa Hirukawa. Biped walking pattern generation by a simple three-dimensional inverted pendulum model. *Advanced Robotics*, 17(2):131–147, 2003.

[39] Dayal C Kar. Design of statically stable walking robot: a review. *Journal of Robotic Systems*, 20(11):671–686, 2003.

[40] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948. IEEE, 1995.

[41] S. Kern, S.D. Müller, N. Hansen, D. Büche, J. Ocenasek, and P. Koumoutsakos. Learning probability distributions in continuous evolutionary algorithms–a comparative review. *Natural Computing*, 3(1):77–112, 2004.

[42] Jeong-Jung Kim, Tae-Yong Choi, and Ju-Jang Lee. Experience repository based particle swarm optimization and its application to biped robot walking. In *Humanoid Robots, 2008. Humanoids 2008. 8th IEEE-RAS International Conference on*, pages 373–378, 2008. doi: 10.1109/ICHR.2008.4755980.

[43] Jung-Yup Kim and Young-Seog Kim. Human-Like Gait Generation for Biped Android Robot using Motion Capture and ZMP Measurement System. *International Journal of Humanoid Robotics*, 7(04):511–534, 2010.

[44] S. Kim, C.H. Kim, B. You, and S. Oh. Stable whole-body motion generation for humanoid robots to imitate human motions. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 2518–2524. Ieee, 2009.

[45] J. Könemann and M. Bennewitz. Whole-body imitation of human motions with a nao humanoid. In *Human-Robot Interaction (HRI), 2012 7th ACM/IEEE International Conference on*, page 425, 2012.

[46] Sylvain Koos, J-B Mouret, and Stéphane Doncieux. The transferability approach: Crossing the reality gap in evolutionary robotics. *Evolutionary Computation, IEEE Transactions on*, 17(1):122–145, 2013.

[47] P. Kormushev, D.N. Nenchev, S. Calinon, and D.G. Caldwell. Upper-body kinesthetic teaching of a free-standing humanoid robot. In *Robotics and Automation (ICRA), 2011 IEEE International Conference*, pages 3970–3975, 2011.

[48] Patrick MacAlpine, Daniel Urieli, Samuel Barrett, Shivaram Kalyanakrishnan, Francisco Barrera, Adrian Lopez-Mobilia, Nicolae Ştiurcă, Victor Vu, and Peter Stone. UT Austin Villa 2011: A champion agent in the RoboCup 3D soccer simulation competition. In *Proc. of 11th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2012)*, June 2012.

[49] Olivier Michel. Webots: Symbiosis between virtual and real mobile robots. In *Virtual Worlds*, pages 254–263. Springer, 1998.

[50] M. Missura and S. Behnke. Lateral capture steps for bipedal walking. In *Humanoid Robots (Humanoids), 2011 11th IEEE-RAS International Conference on*, pages 401–408, 2011. doi: 10.1109/Humanoids.2011.6100886.

[51] Marcell Missura and Sven Behnke. Lateral capture steps for bipedal walking. In *Humanoid Robots (Humanoids), 2011 11th IEEE-RAS International Conference on*, pages 401–408. IEEE, 2011.

[52] Marcell Missura and Sven Behnke. Omnidirectional capture steps for bipedal walking. In *Proceedings of IEEE Int. Conf. on Humanoid Robots (Humanoids)*, 2013.

[53] M. Morisawa, K. Harada, S. Kajita, K. Kaneko, J. Sola, E. Yoshida, N. Mansard, K. Yokoi, and J-P Laumond. Reactive stepping to prevent falling for humanoids. In *Humanoid Robots, 2009. Humanoids 2009. 9th IEEE-RAS International Conference on*, pages 528–534, 2009. doi: 10.1109/ICHR.2009.5379522.

[54] Christian L Müller, Benedikt Baumgartner, Georg Ofenbeck, Birte Schrader, and Ivo F Sbalzarini. pCMALib: A Parallel Fortran 90 Library for the Evolution Strategy with Covariance Matrix Adaptation. In *Proceedings of the 11th Annual conference on Genetic and Evolutionary Computation*, pages 1411–1418. ACM, 2009.

[55] K. Munirathinam, C. Chevallereau, and S. Sakka. Offline imitation of a human motion by a humanoid robot under balance constraint. In Aleksandar Rodic, Doina Pisla, and Hannes Bleuler, editors, *New Trends in Medical and Service Robots*, volume 20 of *Mechanisms and Machine Science*, pages 269–282. Springer International Publishing, 2014. ISBN 978-3-319-05430-8.

[56] Luca Mussi, Fabio Daolio, and Stefano Cagnoni. Evaluation of Parallel Particle Swarm Optimization Algorithms within the CUDA Architecture. *Information Sciences*, 181(20):4642–4657, 2011.

[57] Luca Mussi, Youssef SG Nashed, and Stefano Cagnoni. GPU-Based Asynchronous Particle Swarm Optimization. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, pages 1555–1562. ACM, 2011.

[58] Gabe Nelson, Aaron Saunders, B Swilling, J Bondaryk, D Billings, C Lee, R Playter, and M Raibert. PETMAN: A humanoid robot for testing chemical protective clothing. *Journal of the Robotics Society of Japan*, 30:372–377, 2012.

[59] Van Vuong Nguyen and Joo-Ho Lee. Full-body imitation of human motions with kinect and heterogeneous kinematic structure of humanoid robot. In *System Integration (SII), 2012 IEEE/SICE International Symposium*, pages 93–98, 2012.

[60] Oliver Obst, Markus Rollmann, and Markus Rollmann. Spark - a generic simulator for physical multi-agent simulations. In *Computer Systems Science and Engineering*, 2004.

[61] N. Oda and O. Yanada. Motion controller design of biped robot for human-like walking with stretched knee. In *IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society*, pages 5434–5439, 2012. doi: 10.1109/IECON.2012.6389523.

[62] Yuki Okuzawa, Shohei Kato, Masayoshi Kanoh, and Hidenori Ito. Imitative Motion Generation for Humanoid Robots based on the Motion Knowledge Learning and Reuse. In *IEEE International Conference on Systems, Man and Cybernetics, 2009. SMC 2009.*, pages 4031–4036. IEEE, 2009.

[63] OpenNI Organization. *OpenNI User Guide*, November 2010. URL `http://www.openni.org/documentation`. Last viewed 02-26-2012.

[64] Jimmy Or. A Hybrid CPG–ZMP Control System for Stable Walking of a Simulated Flexible Spine Humanoid Robot. *Neural Networks*, 23(3):452–460, 2010.

[65] C. Ott, M.A. Roa, and G. Hirzinger. Posture and balance control for biped robots based on contact force optimization. In *Humanoid Robots (Humanoids), 2011 11th IEEE-RAS International Conference on*, pages 26–33, 2011. doi: 10.1109/Humanoids.2011.6100882.

[66] PrimeSense. *Prime Sensor NITE 1.3 Algorithms notes*. Primsense Inc., 2010. URL `http://www.primesense.com`. Last viewed 02-26-2012.

[67] Lawrence Rabiner and Biing-Hwang Juang. An Introduction to Hidden Markov Models. *ASSP Magazine, IEEE*, 3(1):4–16, 1986.

[68] Thomas Röfer, Tim Laue, Judith Müller, Alexan der Fabisch, Fynn Feldpausch, Katharina Gillmann, Colin Graf, Thijs Jeffry de Ha as, Alexander Härtl, Arne Humann, Daniel Honsel, Philipp Kastner, Tobias Kastner, Carsten Könemann, Benjamin Markows ky, Ole Jan Lars Riemann, and Felix Wenk. B-Human Team Report and Code Release 2011. Available online, 2011. URL `http://www.b-human.de/downloads/bhuman11_coderelease.pdf`.

[69] Jos Rosado, Filipe Silva, and Vtor Santos. A kinect-based motion capture system for robotic gesture imitation. In Manuel A. Armada, Alberto Sanfeliu, and Manuel Ferre, editors, *ROBOT2013: First Iberian Robotics Conference*, volume 252 of *Advances in Intelligent Systems and Computing*, pages 585–595. Springer International Publishing, 2014. ISBN 978-3-319-03412-6.

[70] Stuart Russell and Peter Norvig. *Artificial Intelligence : a Modern Approach*. Prentice Hall, 3 edition, December 2010. ISBN 0136042597.

[71] Thomas Rfer and Tim Laue. On b-human's code releases in the standard platform league - software architecture and impact. In *RoboCup 2013: Robot Soccer World Cup XVII*, Lecture Notes in Artificial Intelligence. Springer, 2014.

[72] Tom Schaul, Tobias Glasmachers, and Jürgen Schmidhuber. High dimensions and heavy tails for natural evolution strategies. In *Genetic and Evolutionary Computation Conference (GECCO)*, 2011.

[73] Christian Schönauer, Thomas Pintaric, and Hannes Kaufmann. Full body motion capture - a flexible marker based solution. In *Proceedings of Workshop on*

Accessibility Engineering with user models, simulation and VR, 2012. Vortrag: Joint Virtual Reality Conference (JVRC 2011), Nottingham, UK; 2011-09-20 – 2011-09-21.

[74] Jaco F Schutte, Jeffrey A Reinbolt, Benjamin J Fregly, Raphael T Haftka, and Alan D George. Parallel Global Optimization with the Particle Swarm Algorithm. *International Journal for Numerical Methods in Engineering*, 61 (13):2296–2315, 2004.

[75] Andreas Seekircher, Justin Stoecker, Saminda Abeyruwan, and Ubbo Visser. Motion Capture and Contemporary Optimization Algorithms for Robust and Stable Motions on Simulated Biped Robots. pages 213–224, 2013.

[76] A. Sekiguchi and Y. Tsumaki. A prototype foot shape for human-like walk of humanoid robot. In *SICE Annual Conference 2010, Proceedings of*, pages 1804–1807, 2010.

[77] Adam Setapen, Michael Quinlan, and Peter Stone. MARIOnET: Motion Acquisition for Robots through Iterative Online Evaluative Training (Extended Abstract). In *The Ninth International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. International Foundation for Autonomous Agents and Multiagent Systems, May 2010.

[78] Ching-Long Shih and Chien-Jung Chiou. The motion control of a statically stable biped robot on an uneven floor. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 28(2):244–249, 1998.

[79] Ching-Long Shih, Wen-Yo Lee, and Chia-Pin Wu. Planning and Control of Stable Walking for a 3D Bipedal Robot. *International Journal of Advanced Robotic Systems*, 9(47), 2012.

[80] Zhibin Song and Shuxiang Guo. Development of a real-time upper limb's motion tracking exoskeleton device for active rehabilitation using an inertia sensor. In *Intelligent Control and Automation (WCICA), 2011 9th World Congress*, pages 1206–1211, 2011.

[81] Justin Stoecker and Ubbo Visser. RoboViz: Programmable Visualization for Simulated Soccer. In Thomas Röfer, Norbert Michael Mayer, Jesus Savage, and Uluç Saranli, editors, *RoboCup 2011: Robot Soccer World Cup XV*, 2012.

[82] Johannes H. Strom, George Slavov, and Eric Chown. Omnidirectional walking using zmp and preview control for the nao humanoid robot. In *RoboCup*, pages 378–389, 2009.

[83] Gaurav Tevatia and Stefan Schaal. Inverse kinematics for humanoid robots. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 1, pages 294–299. IEEE, 2000.

[84] Chayooth Theeravithayangkura, Tomohito Takubo, Kenichi Ohara, Yasushi Mae, and Tatsuo Arai. Adaptive gait for dynamic rotational walking motion on unknown non-planar terrain by limb mechanism robot asterisk. *Journal of Robotics and Mechatronics*, 25(1):172–182, 2013.

[85] Sebastian Thrun. Particle filters in robotics. In *Proceedings of the Eighteenth conference on Uncertainty in artificial intelligence*, pages 511–518. Morgan Kaufmann Publishers Inc., 2002.

[86] Kuo-Yang Tu and Zhan-Cheng Liang. Parallel Computation Models of Particle Swarm Optimization Implemented by Multiple Threads. *Expert Systems with Applications*, 38(5):5858–5866, 2011.

[87] Barkan Ugurlu and Atsuo Kawamura. Bipedal Trajectory Generation based on Combining Inertial Forces and Intrinsic Angular Momentum Rate Changes: Eulerian ZMP Resolution. *IEEE Transactions on Robotics*, 28(6):1406–1415, 2012.

[88] Daniel Urieli, Patrick MacAlpine, Shivaram Kalyanakrishnan, Yinon Bentor, and Peter Stone. On optimizing interdependent skills: A case study in simulated 3d humanoid robot soccer. In Kagan Tumer, Pinar Yolum, Liz Sonenberg, and Peter Stone, editors, *Proc. of 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, volume 2, pages 769–776. IFAAMAS, May 2011. ISBN 978-0-9826571-5-7.

[89] Miomir Vukobratović and Branislav Borovac. Zero-moment point—thirty five years of its life. *International Journal of Humanoid Robotics*, 1(01):157–173, 2004.

[90] Miomir Vukobratović and J Stepanenko. On the stability of anthropomorphic systems. *Mathematical biosciences*, 15(1):1–37, 1972.

[91] Miomir Vukobratovic, Branislav Borovac, and Dragoljub SurdiloviC. Zero-moment point-proper interpretation and new applications. In *Proceedings of the IEEE/RAS International Conference on Humanoid Robots*, volume 244. Piscataway, NJ, USA: IEEE, 2001.

[92] P. B Wieber. Trajectory free linear model predictive control for stable walking in the presence of strong perturbations. In *Humanoid Robots, 2006 6th IEEE-RAS International Conference on*, pages 137–142, 2006. doi: 10.1109/ICHR.2006.321375.

[93] D. Wierstra, T. Schaul, T. Glasmachers, Y. Sun, and J. Schmidhuber. Natural evolution strategies. Technical Report arxiv:1106.4487v1, arxiv.org, 2011.

[94] Daan Wierstra, Tom Schaul, Jan Peters, and Jürgen Schmidhuber. Natural evolution strategies. In *Proceedings of the Congress on Evolutionary Computation (CEC08), Hongkong*. IEEE Press, 2008.

[95] Dennis Wilson, Kalyan Veeramachaneni, and Una-May O'Reilly. Cloud Scale Distributed Evolutionary Strategies for High Dimensional Problems. In *Applications of Evolutionary Computation*, pages 519–528. Springer, 2013.

[96] William A Wolovich and H Elliott. A computational technique for inverse kinematics. In *Decision and Control, 1984. The 23rd IEEE Conference on*, pages 1359–1363. IEEE, 1984.

[97] Man Leung Wong. Parallel Multi-Objective Evolutionary Algorithms on Graphics Processing Units. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, pages 2515–2522. ACM, 2009.

[98] Shuaijun Wu, Gang Pan, and Ling Yu. Dynamic walking gait designing for biped robot based on particle swarm optimization. In *Control Engineering and Communication Technology (ICCECT), 2012 International Conference on*, pages 372–377, 2012. doi: 10.1109/ICCECT.2012.127.

[99] Woosung Yang, Nak Young Chong, Syungkwon Ra, ChangHwan Kim, and Bum-Jae You. Self-stabilizing bipedal locomotion employing neural oscillators. In *Humanoids*, pages 8–15, 2008.

[100] Seung-Joon Yi, Byoung-Tak Zhang, D. Hong, and D.D. Lee. Active stabilization of a humanoid robot for impact motions with unknown reaction forces. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 4034–4039, 2012. doi: 10.1109/IROS.2012.6385854.

[101] Zhangguo Yu, Xuechao Chen, Qiang Huang, Huaping Wang, Si Zhang, Wei Xu, Jing Li, Gan Ma, Weimin Zhang, and Ningjun Fan. Humanoid Walking Pattern Generation based on the Ground Reaction Force Features of Human Walking. In *2012 International Conference on Information and Automation (ICIA),*, pages 753–758. IEEE, 2012.

[102] J. Ziegler, H. Kretzschmar, C. Stachniss, G. Grisetti, and W. Burgard. Accurate human motion capture in large areas by combining IMU- and laser-based people tracking. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 86–91, 2011.