Evolutionary Legged Robotics

von Malte Langosz

Dissertation

zur Erlangung des Grades eines Doktors der Ingenieurwissenschaften
- Dr.-Ing. -

Vorgelegt im Fachbereich 3 (Mathematik & Informatik)
der Universität Bremen
im November 2018

Datum des Promotionskolloquiums: 14. Februar 2019

Gutachter: Prof. Dr. Frank Kirchner (Universität Bremen)
Prof. Dr. Joachim Hertzberg (Universität Osnabrück)

Abstract

Due to the technological advance, robotic systems become more and more interesting for industrial and home applications. Popular examples are given by robotic lawn mower, robot vacuum cleaner, and package drones. Beside the toy industry, legged robots are not as popular, although they have some clear advantages compared to wheeled systems. With their flexibility concerning the locomotion, they are able to adapt their walking pattern to different environments. For instance they can walk over obstacles and gaps or climb over rubble and stairs. Another possible advantage could be a redundancy for locomotion. A faulty motor in one limb could be compensated by other motors in the kinematic chain. As well, multiple failing legs can be compensated by an adapted walking pattern. Compared to this, the more complex mechatronic systems represent a major challenge to the construction and the control.

This thesis is dedicated to the control of complex walking robots. Genetic algorithms are applied to generate walking patterns for different robots. The evolutionary development of walking patterns is done in a simulation software. Results of various approaches are transferred and tested on existing systems which have been developed at RIC/DFKI. Different robotic systems are used to evaluate the generality of the applied methods. Eventually, a method is developed that can be utilized, with a few system specific modifications, for a variety of legged robots.

As basis for the development and investigation of several methods, software tools are designed to generalize the application of applying genetic algorithms to legged locomotion. These tools include a simulation environment, a behavior representation, a genetic algorithm and a learning and benchmark framework. The simulation environment is adapted to the behavior of real robotic systems via reference experiments. In addition, the simulation is extended by a foot contact model for loose surfaces. The evaluation of the genetic algorithm is done on several benchmark problems and compared to three existing algorithms.

This thesis contributes to the state of the art in many areas. The developed methodology can easily be applied to several complex robotic systems due to its transferability. The genetic algorithm and the hierarchical behavior representation provide a new opportunity to control the generation of the offspring in an evolutionary process. In addition, the developed software tools are an important contribution for their respective research fields.

Zusammenfassung

Robotische Systeme erhalten durch den technologischen Fortschritt mehr und mehr Einzug in das alltägliche Leben. Rasenmähroboter, Staubsaugerroboter und auch Paketdrohnen sind bekannte Beispiele. Laufroboter sind außerhalb der Spielzeugindustrie allerdings wenig verbreitet, obwohl sie gegenüber radgetriebenen Systemen einige Vorteile haben. Ihre große Stärke ist die Flexibilität – sie können sich über ihr Laufverhalten an die Umgebung anpassen und so über Hindernisse oder Spalten laufen, über Geröll klettern oder Treppen steigen. Ein weiterer Vorteil kann eine Redundanz bei der Fortbewegung sein. Ein defekter Motor in einer Extremität kann zum Beispiel durch andere Motoren in der kinematischen Kette kompensiert werden. Ebenso können sogar mehrere ausfallende Beine durch ein angepasstes Laufmuster ausgeglichen werden. Die im Vergleich komplexeren mechatronischen Systeme der Laufroboter stellen jedoch große Herausforderungen an Konstruktion und Steuerung dar.

Diese Arbeit widmet sich der Steuerung komplexer Laufroboter. Es werden genetische Algorithmen eingesetzt, um Laufverhalten für verschiedene Roboter zu generieren. Die Entwicklung der Laufmuster findet ausschließlich in einer Simulationsumgebung statt. Die Ergebnisse unterschiedlicher Ansätze werden auf drei am RIC/DFKI entwickelten Robotersysteme übertragen und bewertet. Um Aussagen über die Allgemeingültigkeit der verwendeten Methoden zu ermöglichen werden Roboter mit unterschiedlicher Komplexität eingesetzt. Daraus folgt die Entwicklung einer Methodik, die mit geringfügigen, roboterspezifischen Anpassungen für eine Vielzahl von Systemen einsetzbar ist.

Als Grundlage für die Entwicklung und Untersuchung verschiedener Methoden werden Software-Werkzeuge entwickelt, die die Anwendung genetischer Algorithmen auf Laufroboter weitestgehend modularisieren und abstrahieren. Zu den Werkzeugen gehören eine Simulationsumgebung, eine Verhaltensrepräsentation, ein genetischer Algorithmus und ein Lern- und Benchmarking-Framework. Die Simulationsumgebung wird über Vergleichsexperimente an das Verhalten der realen Roboter angeglichen. Zudem wird die Simulation um ein Modell für die Fuß-Boden-Interaktion auf losem Untergrund erweitert. Die Bewertung des genetischen Algorithmus' erfolgt über mehrere Benchmarkprobleme mit drei vergleichbaren Algorithmen.

Diese Arbeit erweitert den Stand der Technik in diversen Bereichen. Die entwickelte Methodik lässt sich durch ihre Übertragbarkeit auf verschiedene komplexe Robotersysteme anwenden. Der genetische Algorithmus und die hierarchische Verhaltensrepräsentation bieten eine neuartige Möglichkeit, die Generierung der Nachfahren im evolutionären Prozess zu kontrollieren. Auch die entwickelten Software-Werkzeuge stellen in ihrem jeweiligen Gebiet einen wertvollen Beitrag dar.

Danksagung

Seit ich als Schüler angefangen habe mich mit Computern und Technik zu beschäftigen, treibt mich der Gedanke voran, die technischen Geräte, die uns heutzutage zur Verfügung stehen, intelligenter zu machen. Als ich dann in der AG Robotik die Möglichkeit bekam an der Laufrobotik mitzuarbeiten, bot sich mir eine großartige Gelegenheit viel über künstliche Intelligenz zu lernen. Meine Motivation technische Geräte intelligenter zu machen wurde für die nächsten Jahre konkretisiert: Ein Laufroboter sollte in der Lage sein die Koordination seiner eigenen Beine ohne menschliches Eingreifen zu erlernen.

Daher möchte ich an erster Stelle meinem Doktorvater Prof. Dr. Frank Kirchner danken, der mich über seine Robotik Vorlesungen für das Thema begeistert hat. Während ich meiner Forschung nachgekommen bin hat er meinen Weg mit wertvollen Diskussionen, Ratschlägen aber auch Perspektivwechseln mitgelenkt. Unter seiner Leitung hat die AG Robotik zusammen mit dem RIC/DFKI eine herausragende Umgebung bereitgestellt um an meinem Thema zu forschen. Neben einer Vielzahl verschiedener Robotersysteme konnte ich jederzeit auf kompetente Unterstützung in allen Bereichen zurückgreifen. Besonders hervorheben möchte ich hier die Hardware Teams (Konstruktion und Elektronik) die mit ihrer Expertise komplexe und robuste Laufroboter entwickelt haben auf denen ich auch mal nicht ganz "optimale" Laufverhalten ausführen konnte.

Während meiner Forschung für meine Doktorarbeit haben mich viele Freunde und Kollegen mit Diskussionen, Fachwissen, kreativen Ideen und mit der Hilfe bei der Durchführung von zahlreichen Experimenten unterstützt. Die Liste wäre zu lang um alle namentlich zu erwähnen, aber Einigen gebührt besonderer Dank, da sie mich über viele Jahre begleitet haben: Dr. Yohannes Kasssahun, Lorenz Quack, Kai von Szadkowski, Alexander Dettman, Dr. Daniel Kühn, Dr. Berthold Bongart, Dr. Sebastian Bartsch, Michael Rohn und Familie Niebuhr.

Zuletzt geht mein größter Dank an meine Frau Julia Langosz, die mich durch alle Höhen und Tiefen, Erfolge und Misserfolge begleitet hat. Sie hat mir, mit unendlich viel Verständnis, auch privat den Raum geschaffen meiner Forschung und meiner Arbeit nachzugehen. Zudem hat sie immer ein offenes Ohr für mich gehabt, was sie jetzt wahrscheinlich zu einer mir gleichgestellten Expertin in dem Gebiet macht.

Contents

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Goal . . . 4
- 1.3 Structure of the Thesis . . . 5
- 1.4 Contributions . . . 6

2 Background and Related Work 9
- 2.1 Legged Locomotion . . . 9
  - 2.1.1 Model-Based Control . . . 9
  - 2.1.2 Bio-Inspired Control . . . 10
- 2.2 Evolution and Optimization . . . 15
  - 2.2.1 Natural Evolution . . . 15
  - 2.2.2 Artificial Evolution . . . 16
  - 2.2.3 Particle Swarm Optimization . . . 21
- 2.3 Robotic Systems . . . 22
  - 2.3.1 SPOT . . . 22
  - 2.3.2 SpaceClimber . . . 23
  - 2.3.3 Charlie . . . 27

3 Simulation MARS 31
- 3.1 Software Architecture . . . 31
- 3.2 Open Dynamics Engine . . . 35
- 3.3 Comparison of Simulated and Real Robot . . . 36
- 3.4 Ground Interaction Model by Neural Network . . . 39
  - 3.4.1 Passive Experiments . . . 39
  - 3.4.2 Active Experiments . . . 44
  - 3.4.3 Conclusion . . . 49

4 Learning Framework BOLeRo 51
- 4.1 Definition . . . 51
- 4.2 Implementation . . . 53
  - 4.2.1 Environments . . . 57
  - 4.2.2 Learning Algorithms . . . 60

4.2.3 Parameter Optimizer . . . 62

5 Behavior Representation BAGEL 63
5.1 Definition . . . 64
5.2 Implementation . . . 67
5.2.1 C Execution Library . . . 67
5.2.2 Extern Nodes . . . 69
5.2.3 Unit Test . . . 70
5.2.4 C++ Execution Wrapper . . . 71
5.2.5 Simulation Module . . . 71
5.2.6 Graphical Design Software . . . 72
5.3 Conclusion . . . 73

6 Genetic Algorithm SABRE 75
6.1 Definition . . . 76
6.1.1 Structural Mutation . . . 78
6.1.2 Parameter Mutation . . . 79
6.1.3 Default Configuration . . . 80
6.2 Implementation . . . 81
6.2.1 Integration Into Learning Framework . . . 81
6.3 Validation Benchmarking . . . 82
6.3.1 Black Box Function Approximation . . . 83
6.3.2 Analysis on Transfer Functions . . . 87
6.3.3 CEC13 Test-Functions . . . 92
6.3.4 Torcs . . . 92
6.3.5 HyperRobot . . . 93
6.3.6 Conclusion . . . 94

7 Central Pattern Control Approach 95
7.1 Central Pattern Generator . . . 95
7.2 Joint Pattern Representation . . . 97
7.2.1 Sine-Based Pattern . . . 97
7.2.2 Gaussian Pattern . . . 98
7.2.3 Pendulum Pattern . . . 101
7.2.4 Fourier Series . . . 102
7.2.5 Neural Oscillator . . . 103
7.2.6 Sigmoid ANN Pattern . . . 104
7.3 Experiments . . . 104
7.3.1 Sine-Based Test-Functions . . . 105
7.3.2 Gauss-Based Wave . . . 110
7.3.3 Model-Based Joint Pattern . . . 113

7.3.4 Human Joint Pattern 114
7.4 Conclusion 115

8 Evolving Legged Locomotion 117
8.1 Simulated Robot 117
8.1.1 Evolving Open-Loop Control 117
8.1.2 Navigation Controller 125
8.1.3 Stabilize Controller 131
8.2 Experiments with Real Systems 134
8.2.1 SPOT 135
8.2.2 SPACECLIMBER 138
8.2.3 CHARLIE 151
8.2.4 Conclusion 160

9 Conclusion and Outlook 163

A Appendix 169
A.1 Configurations 169
A.1.1 SPACECLIMBER 169
A.1.2 CEC13 Benchmark Functions 172
A.1.3 NEAT 173
A.1.4 Locomotion Environment 178
A.2 Experiment Results 179
A.2.1 Evolving Legged Locomotion 179
A.2.2 Benchmark Analysis 180

Terms and Definitions 181

Acronyms 183

Math Notations 185

List of Tables 187

List of Figures 189

Bibliography 195

Chapter 1

Introduction

1.1 Motivation

Robotics is an interdisciplinary field with an increasing complexity of robotic systems due to the progress of technology. This includes how a robotic system can perceive its environment through an increasing sensor density and the opportunity to implement many degrees of freedom without significantly scaling up the overall system's size and weight. The resulting complexity of robotic systems also increases the flexibility of their applications. Due to this flexibility the smart robotics field becomes more and more interesting for industrial (Bogue (2012); Hvilshøj et al. (2012)) and home applications (Iocchi et al. (2015); Riek (2017)). For example a fictitious complex bipedal robot could perform all tasks in a home environment that a human can execute.

Flexibility concerning the locomotion of a robot in many environments is a quality shared by legged systems. Some examples can be found in Klaassen et al. (2002); Sakagami et al. (2002); Spenneberg et al. (2005b); Albiez (2007); Raibert et al. (2008); Zucker et al. (2010); Bartsch (2014); Kühn (2016). In contrast to wheeled systems they can adapt better to different environmental settings like stairs of different step sizes and also overcome obstacles by performing big steps or climbing above them. The main disadvantage of legged systems is the amount of control necessary for coordinating the legs to move the robot in a desired direction or climb on objects by keeping the system stable. The control generally gets more complex with an increasing system complexity and resulting flexibility.

The complexity of locomotion control is affected by the structural complexity of the system, like the number of legs, the degrees of freedom per leg, and the mass distribution. Other factors influencing the complexity are the dynamic properties of the system. This includes, amongst others, properties maximal torque and velocity of the actuators, electromechanical compliance of the system that could compensate for non-optimal control inputs, and properties concerning the interaction with the environment, mainly represented by the contact behavior of the feet on the ground surface.

The problem of controlling walking robots is tackled by many different approaches. These approaches can represent any combination of three main classes. The first class uses pure mathematical models to compute the actuator input by modeling the system and its environment and accounting for the static and dynamic stability of the system, referred to as *model-based* control in this work. A comprehensive introduction is given in Wieber et al. (2016). The second class, labeled *bio-inspired* control, represents a model of locomotion control observed and analyzed from natural examples like stick insects, sala-

manders, or cats. Prominent examples are given in Cruse et al. (1998); Ijspeert (2008); Manoonpong et al. (2008); Spenneberg and Kirchner (2007). A third alternative is to generate the locomotion control by machine learning techniques or by utilizing evolutionary principles, referred to as *generated* control. Examples can be found in Beer and Gallagher (1992); Reil and Husbands (2002); Ito and Matsuno (2002); Clune et al. (2009).

Setting up the model-based or bio-inspired approaches for a new robotic system without utilizing optimization techniques requires expert knowledge depending on the system's complexity, either for setting up and configuring the models or for tuning the bio-inspired control parameters to fit the control to the robotic morphology and dynamic properties. Generated control approaches, on the other hand, have the potential to produce solutions for any morphology without the need of an expert. Additionally, they might be able to solve the problem with less computational effort while intrinsically utilizing the system properties. However, the purely evolved control approaches do not scale well with an increasing system complexity due to the massive explosion of the search space, known as *curse of dimensionality*. Another difficulty arises from the structure of the fitness landscape, which is spanned by the evaluation of the behaviors while moving in the search space. An ideal fitness landscape guides the learning algorithm from every point in the search space to a global optimum. The more noise and local optima the fitness landscape includes, the more complex is the search for the global optimum. If the fitness landscape reaches a certain degree of complexity, the search problem cannot be solved with a feasible number of trials (test of single behaviors). Thus, the complexity for a learning algorithm is defined by the structure of the fitness landscape which depends on the search space defined by the kinematic complexity and the behavior representation, the dynamic properties of the robotic system and the evaluation function.

Although some approaches successfully generated locomotion behaviors directly on a real system (as done by Lewis et al. (1992); Gomi and Ide (1998); Kirchner (1998, 1999); Zykov et al. (2004); Röfer (2005)) there is a relatively high risk of breaking the system when performing non-optimal actions. Thus many approaches are using a model for the learning phase, utilizing a simulation tool (Reil and Husbands (2002); Valsalam and Miikkulainen (2009); Peng et al. (2017); Gay et al. (2013)). But these models cannot be perfect and therefor introduce a *simulation-reality gap*. This means a controller that performs well in simulation can have a poor performance on the real system due to properties that are not modeled. Unfortunately, these problems imply a learning expert that understands the search space and the simulation-reality gap to be able to adapt the setup for a given robotic system to allow the generation of usable controllers. Examples of transferred behaviors from simulated robots to real systems are given in Jakobi et al. (1995); Boeing et al. (2004); Lee et al. (2006); Spröwitz et al. (2013).

The power of evolution is demonstrated by the evolution of life on earth. Natural evolution generated billions of specialized life forms that can survive in different and also very rough environments. Figure 1.1 depicts a few examples of the variety of life forms living on earth. Under the correct circumstances evolution can produce complex fine-tuned results as presented in the examples. The very core principle of evolution is the continuous adaptation of populations of agents or individuals to an environment. The fittest individuals of a population are selected to generate a new population through reproduction. In an ideal artificial world an artificial evolution could possibly come up with an intelligent agent that has the ability to learn within its lifetime via an evolved controller. This is a visionary view on artificial evolution, but the evolutionary principles can also be applied on much simpler optimization problems, like parameter tuning of a developed controller.

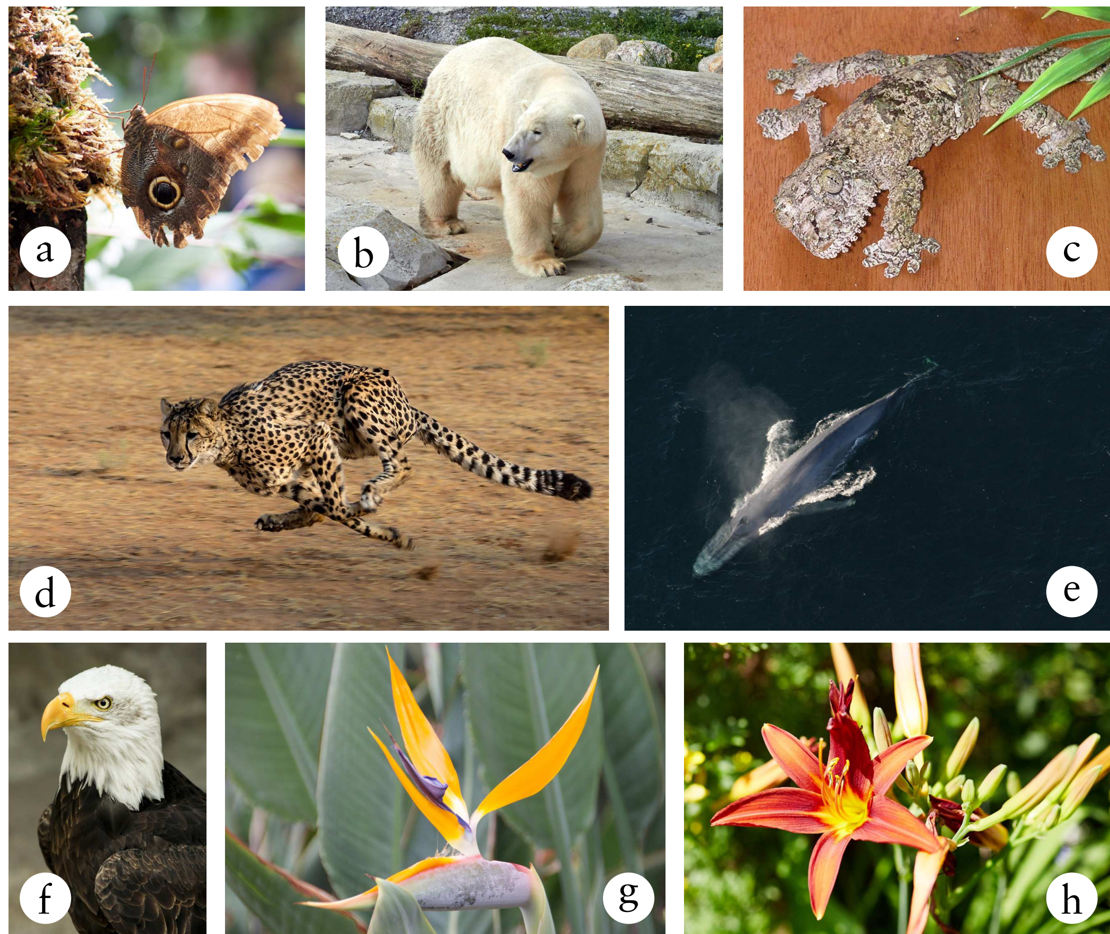


Figure 1.1: The sub figures depict a few examples of specialized animals and plants. (a) Shows a moth of the caligo species. It is camouflaged when sitting on a tree trunk, while its coloring of the wings imitates the eyes of an owl and could deceive a possible predator. (b) Shows a polar bear in a zoo. The polar bear can survive in the very cold desert of the arctic. With the white coloring of its fur it is well camouflaged for approaching possible prey. In contrast, the skin underneath the fur is black, absorbing the sun light and keeping the bear warm. (c) Depicts a leaf-tailed gecko, which is nearly invisible when sitting on a tree trunk. (d) The cheetah is another example of the fastest terrestrial animal. With its dynamic sprinting skills it can reach up to 100 kilometers per hour. (e) Depicts a blue whale, which is, of about 170 tonnes of weight, the biggest creature known that ever lived on earth. The blue whale is part of the marine mammal group, which adapted back from land to the sea. (f) Shows an eagle which has one of the best eyes, being able to catch sight of a mouse while gliding more than 100 m above the ground. (g) and (h) depict plants that are also specialized to attract insects or birds for transporting their seeds.

(a), (b), (g), and (h) Pictures are taken by the author

(c) `https://commons.wikimedia.org/wiki/File:UroplatusSikoraeSameiti.png`

(d) `https://pixabay.com/de/gepard-afrika-namibia-katze-laufen-2859581`

(e) `https://commons.wikimedia.org/wiki/File:Anim1755_-_Flickr_-_NOAA_Photo_Library.jpg`

(f) `https://pixabay.com/de/weißkopfseeadler-vogel-raubtier-142685`

A general methodology that allows to evolve a locomotion controller in simulation, either offline or while running on the robot, would reduce the effort required to develop legged robotic systems. The same method could be used to adapt the robot's behavior to new environments or new circumstances. An evolutionary method that adapts the behavior step by step could also be used continuously on a robotic system in the sense of long-time learning.

1.2 Goal

As outlined in Section 1.1, the control of complex legged robots is a challenging task. Evolutionary methods are able to solve the problem, but due to the enormous search space and the resulting fitness landscape, the learning setup has to be tuned to allow the evolution to succeed. This reduction of the search space is often done specifically for one robotic system, for instance by limiting or reducing selected degrees of freedom of the system. The main goal of the thesis focuses on this problem:

> **Goal:** Reduce the search space for evolving locomotion controllers by: 1. predefining control modules that limit the control possibilities to a useful range and can still be used by all systems. 2. defining an iterative method with increasing complexity instead of dealing with the whole search space at once.

Due to the fact that the search space and the fitness landscape cannot be defined analytically a series of experiments is necessary to achieve the main goal. For these experiments it must be possible to control how the evolution operates in the search space. Therefore, a set of subgoals is defined:

Subgoal 1: Develop a behavior representation that allows the definition of any kind of control structure and that can be configured to control where and how genetic mutations occur.

Subgoal 2: Develop a genetic algorithm that operates on the resulting behavior structure. The new algorithm should have at least the same performance as common genetic algorithms. In contrast to existing algorithms, it has to allow the modification of selected behavior modules simultaneously on different levels of the control hierarchy.

Subgoal 3: In order to evolve controllers for different real robotic systems, a simulation framework has to be developed that fulfills the requirements of optimization applications. The simulation has to be precise enough to allow the transfer of evolved behaviors to the real systems.

Subgoal 4: In order to perform the needed experiments and to verify the new algorithms, a benchmarking framework has to be developed.

All tools developed for this thesis should be available for the research community to allow a comparison of different approaches mainly for locomotion control on the exact same problems implemented in the benchmarking framework.

1.3 Structure of the Thesis

Figure 1.2 depicts the structure of the thesis which starts with Chapter 1 and 2 giving the motivation and goals of this thesis and providing an overview on the research fields correlated to this work. Chapter 3 to Chapter 6 present the methods and implementations done for this thesis. The chapters include single experiments to improve and benchmark the implemented methods. Since the simulation quality is a very important part when designing and optimizing behaviors in simulation for real systems, a framework is introduced in Chapter 3, which is designed to balance the trade-off between computational effort and simulation precision. Especially in the field of evolutionary robotics no standard benchmarks and frameworks are available at this point in time and it is complicated to compare results from different publications. Therefore a learning framework is designed in Chapter 4 with special focus on performing automated benchmarks and providing learning problems for the community.

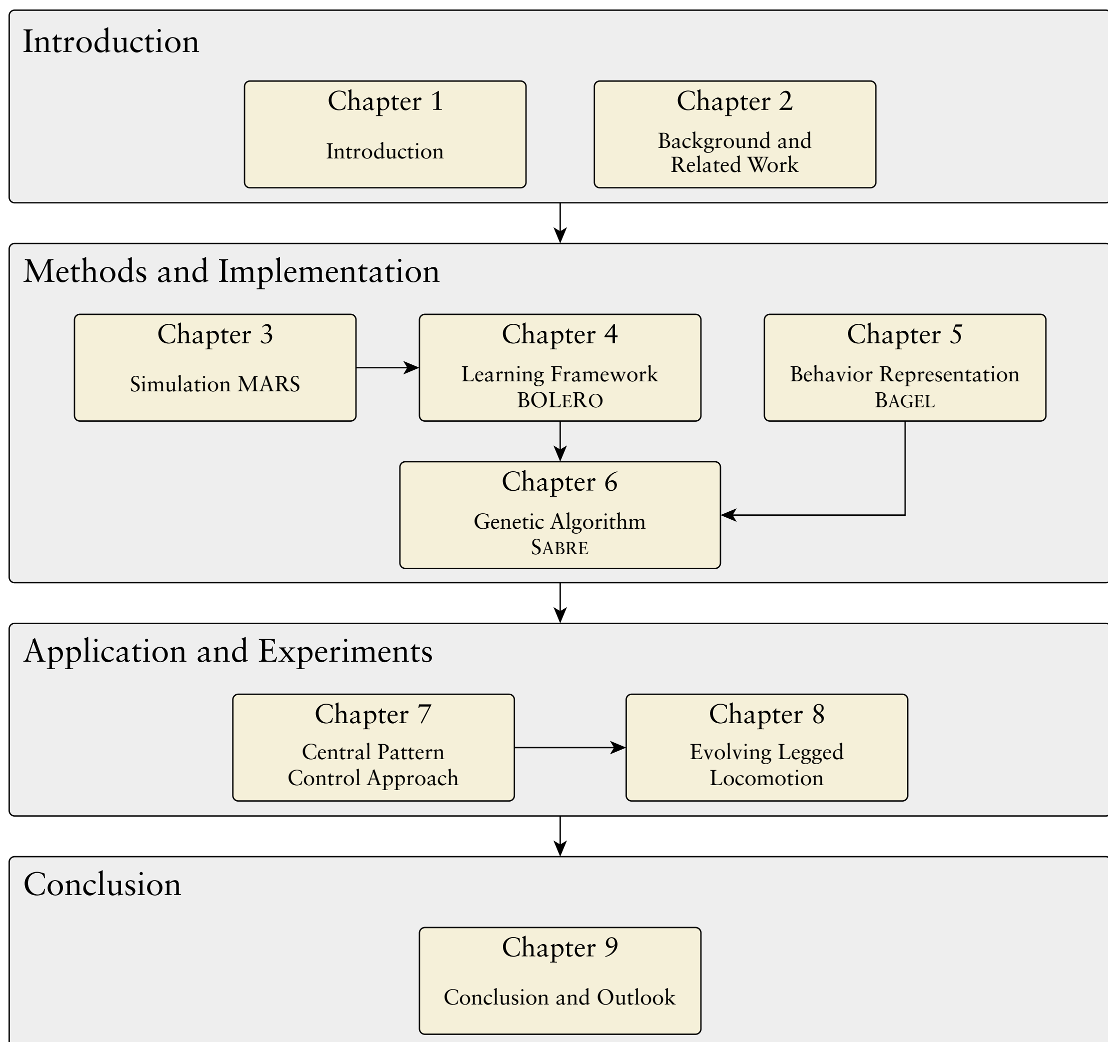


Figure 1.2: Graphical overview showing the structure of the thesis.

For most robotic control frameworks a large implementation effort is needed to optimize one or multiple components simultaneously with evolutionary methods. In particular, this becomes complicated if evolutionary methods are used to modify control

structures on multiple control layers. To allow a high flexibility in applying evolutionary methods on robotic control problems, Chapter 5 introduces a novel graph-based multi-layer control framework. Since the same flexibility has to be provided by the genetic algorithm, Chapter 6 presents a novel genetic method and the interfacing of the control framework with the genetic algorithm and the learning framework. The chapter also includes some benchmark results that compare the performance of the new method with the most popular state of the art algorithm.

Chapter 7 and 8 present the methods and experiments to evolve the locomotion control for complex legged robots. First Chapter 7 experimentally analyses different joint pattern representations. The focus of the chapter lies on the representation strength and on how well the different approaches are suited for applying evolutionary methods to modify the patterns. Afterwards Chapter 8 presents the main experiments performed to generate locomotion control for different target systems. The chapter also includes different approaches to deal with complex locomotion problems.

The last chapter (Chapter 9) summarizes the results of this thesis and the contribution to the state of the art. Additionally, some ideas are presented in the chapter on how the research can be continued.

1.4 Contributions

All publications listed in this section are peer-reviewed conference or journal papers. Until 2012 the author published his results with his family name Malte Römmermann. The simulation-reality gap is published in the following paper:

> Römmermann, M., Bartsch, S., and Haase, S. (2010). Validation of simulation-based morphology design of a six-legged walking robot. In *Emerging Trends in Mobile Robotics*, pages 895–902. World Scientific.

The simulation is extended by a leg-soil interaction model based on real experiments. The extension is done to simulate a more realistic contact behavior. Different approaches, experiments, and results are published in:

> Yoo, Y.-H., Abdenebaoui, L., Rohn, M., Römmermann, M., and Kirchner, F. (2009b). Modelling and simulation of mobile six-legged robot with leg-soil contact. In *Proceedings of the 11th European Regional Conference of the International Society for Terrain-Vehicle Systems*, ISTVS.

> Römmermann, M., Kühn, D., Cordes, F., Yoo, Y.-H., and Kirchner, F. (2009a). Concept evaluation of modeling terrain mechanics by a neural network. In *Proceedings of the 11th European Regional Conference of the International Society for Terrain-Vehicle Systems*, ISTVS.

> Yoo, Y.-H., Jung, T., Römmermann, M., Rast, M., Rossmann, J., and Kirchner, F. (2010). Developing a virtual environment for extraterrestrial legged robot with focus on lunar crater exploration. In *Proceeding of 10th International Symposium on Artificial Intelligent, Robotics and Automation in Space*, iSAIRAS.

> Ahmed, M., Quack, L., Römmermann, M., and Yoo, Y.-H. (2011). Development of a real and simulation testbed for legged robot soil interaction. In *Proceedings of the 17th International Conference of the International Society for Terrain-Vehicle Systems*, ISTVS.

Römmermann, M., Ahmed, M., Quack, L., and Kassahun, Y. (2011). Modeling of leg soil interaction using genetic algorithms. In *Proceedings of the 17th International Conference of the International Society for Terrain-Vehicle Systems*, ISTVS.

Römmermann, M., Ahmed, M., Quack, L., and Kassahun, Y. (2011). An application of genetic algorithms to model leg–soil interaction. In *Proceedings of the 4th International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems*, ERLARS 2011.

The simulation tool was used to support the design of robotic systems and to optimize controllers. In the following list the publications dealing with locomotion control are excluded, since they are listed separately.

Kassahun, Y., de Gea, J., Römmermann, M., and Kirchner, F. (2009). On applying neuroevolutionary methods to complex robotic tasks. In *IEEE IROS Workshops on Exploring new horizons in Evolutionary Design of robots*, pages 26–30.

Yoo, Y., Ahmed, M., Römmermann, M., and Kirchner, F. (2009a). A simulation-based design of extraterrestrial six-legged robot system. In *2009 35th Annual Conference of IEEE Industrial Electronics*, pages 2181–2186.

Kühn, D., Sauthoff, N., Grimminger, F., Römmermann, M., and Kirchner, F. (2009b). Towards a biologically inspired ape-like robot. In *Mobile Robotics*, pages 165–172.

Bartsch, S., Birnschein, T., Cordes, F., Kühn, D., Kampmann, P., Hilljegerdes, J., Planthaber, S., Römmermann, M., and Kirchner, F. (2010). SpaceClimber: Development of a six-legged climbing robot for space exploration. In *ISR 2010 (41st International Symposium on Robotics) and ROBOTIK 2010 (6th German Conference on Robotics)*, pages 1–8.

Straube, S., Rohn, M., Römmermann, M., Bergatt, C., Jordan, M., and Kirchner, E. A. (2011). On the closure of perceptual gaps in man-machine interaction: Virtual immersion, psychophysics and electrophysiology. In *Perception - ECVP Abstract Supplement. European Conference on Visual Perception (ECVP-2011)*, volume 40, pages 177–177.

Dettmann, A., Römmermann, M., and Cordes, F. (2011). Evolutionary development of an optimized manipulator arm morphology for manipulation and rover locomotion. In *2011 IEEE International Conference on Robotics and Biomimetics*, pages 2567–2573.

The novel behavior representation is published in the following papers:

Langosz, M., Quack, L., Dettmann, A., Bartsch, S., and Kirchner, F. (2013). A behavior-based library for locomotion control of kinematically complex robots. In *Nature-Inspired Mobile Robotics*, pages 495–502.

Dettmann, A., Langosz, M., von Szadkowski, K. A., and Bartsch, S. (2014). Towards lifelong learning of optimal control for kinematically complex robots. In *Workshop on Modelling, Estimation, Perception and Control of All Terrain Mobile Robots . IEEE International Conference on Robotics and Automation (ICRA-2014)*. IEEE.

Bartsch, S., Manz, M., Kampmann, P., Dettmann, A., Hanff, H., Langosz, M., von Szadkowski, K., Hilljegerdes, J., Simnofske, M., Kloss, P., Meder, M., and Kirchner, F. (2016). Development and control of the multi-legged robot MANTIS. In *ISR 2016: 47st International Symposium on Robotics.*, pages 379–386, Berlin, Offenbach. VDE VERLAG GmbH.

The genetic algorithm was used for the model development of the leg-soil interaction listed above, while the algorithm itself is published in:

Langosz, M., von Szadkowski, K. A., and Kirchner, F. (2014). Introducing particle swarm optimization into a genetic algorithm to evolve robot controllers. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO Comp '14, pages 9–10, New York, NY, USA. ACM.

Publications with the focus on evolving the locomotion control of legged robots:

Römmermann, M., Edgington, M., Metzen, J. H., de Gea, J., Kassahun, Y., and Kirchner, F. (2008). Learning walking patterns for kinematically complex robots using evolution strategies. In Rudolph, G., Jansen, T., Beume, N., Lucas, S., and Poloni, C., editors, *Parallel Problem Solving from Nature – PPSN X*, pages 1091–1100, Berlin, Heidelberg. Springer.

Kühn, D., Römmermann, M., Sauthoff, N., Grimminger, F., and Kirchner, F. (2009a). Concept evaluation of a new biologically inspired robot "LittleApe". In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 589–594.

Römmermann, M., Kühn, D., and Kirchner, F. (2009b). Robot design for space missions using evolutionary computation. In *2009 IEEE Congress on Evolutionary Computation*, pages 2098–2105.

Bartsch, S., Birnschein, T., Römmermann, M., Hilljegerdes, J., Kühn, D., and Kirchner, F. (2012). Development of the six-legged walking and climbing robot SpaceClimber. *Journal of Field Robotics*, 29(3):506–532.

Chapter 2

Background and Related Work

This chapter introduces the basics of legged locomotion and evolution. While targeting at the control and development of artificial legged systems, also the biological background and modeling is introduced. Since the development of the robotic systems, used to test the results of this thesis, is not part of this work, the systems are explained in detail within this chapter.

2.1 Legged Locomotion

Controlling the legs of a walking system poses a complex problem. The legs have to support the robot's weight and push it forward without acting against each other. At the same time, they have to adapt their movements to the properties of the environment, for instance the roughness, softness, or friction of the surface. This control problem can be solved by a mathematical approach, referred to as *model-based* control in this work. Alternatively, the control can be achieved by imitating the behaviors observed in nature, referred to as *bio-inspired* approach. The two main methods have their pros and cons, as will be presented in the following section. For many applications a combination of both approaches represents an adequate solution.

2.1.1 Model-Based Control

In a purely model-based approach, mass distribution and leg loads are calculated based on the dynamic model of the system, which is fed with the motor and posture sensors. Based on the calculated center of mass, the zero moment point, and the feet contact points, the space of possible next foot positions can be calculated. The exact next foot positions are calculated by a planner, which needs a rather precise map of the environment and the correct location of the robot in the map. The mapping and location is provided by SLAM[1] algorithms, which are widely used for navigation planning of autonomous systems. Since a leg is only allowed to be lifted up if the robot remains stable, a main problem can be the decision which leg to move next. A multi-legged system can get stuck in a state where no legs are allowed to move. Beside the challenge of coordinating the legs, the model-based approaches need high computation effort and are generally limited by the precision of the model and the perception of the environment needed for step planning. For details on the model-based approaches see Garcia et al. (2003); Bretl et al. (2003); Vernaza et al. (2009); Bombled (2011); Righetti et al. (2013); Herzog et al. (2016); Xin et al. (2018).

[1] SLAM: Self-localization and mapping

2.1.2 Bio-Inspired Control

A large effort is spent in understanding how locomotion works in nature to optimize motion sequences in sports, for rehabilitation practice, or to model and imitate locomotion principles with artificial systems. To transfer locomotion concepts from nature to artificial systems, two approaches are commonly used. The first one models the holistically observed behavior, mostly of insects, by reflex chains. A common model of this approach is the WALKNET as introduced in Cruse et al. (1998). Since it was initially introduced WALKNET was adapted and extended in several ways. It is used on different robots, as presented in Weidemann et al. (1994); Lewinger et al. (2006); Paskarbeit et al. (2010). However, it is based on the locomotion of insects and it is unknown how well it could represent the dynamic locomotion of mammals, like a cheetah, a horse or a human, especially if sensory feedback is too slow for fast gaits (Schilling et al. (2013)).

On the other hand, locomotion patterns could be observed by experiments with different animals, leading to a central pattern generator (CPG) model for locomotion control Minassian et al. (2017); Ijspeert (2008); Guertin (2009); Marder and Bucher (2001). In the CPG approach, some base movement patterns are generated, that can, for example, be modulated by the amplitude, the frequency, or an offset. These modifications allow, for instance, to adapt the walking speed or provide a turning behavior. These base patterns can be generated by simple recurrent artificial neural networks composed of only two interconnected neurons, see review Ijspeert (2008) which lists many examples. Other approaches, like Spenneberg (2006); Spröwitz et al. (2013), use modulated sine waves or Bezier curves representing the patterns. The patterns are executed on or provided to the legs with a defined phase shift resulting in a basic locomotion control. This phase shift can be modeled by a rather static shift or through a phase coupling, which allows a temporary adaption of the phase shift through sensory feedback. An additional adaptation to the environment can be achieved by reflexes to correct the basic pattern if the sensory readings do not fit well enough to the expectations.

Both approaches, the WALKNET and the CPG, are combined with the model-based control, at least by using a kinematic model of the robot to control the robots in Cartesian space instead of joint space, e. g. Schilling et al. (2012); Bartsch et al. (2012); Kühn et al. (2009b); Kühn (2016). This makes the tuning of the approaches to a target system more intuitive for a human developer and allows a reduction of the parameter space. The basic concepts of both approaches are explained more in detail in this section. The last part of this section discusses the bio-mechanical system (biomechanics), which plays an important role for natural locomotion.

WALKNET

Schilling et al. (2013) provides a helpful overview of related publications and the concept, development, and different versions of WALKNET. For the WALKNET model, the leg movement is split into a swing and a stance phase. Two positions of the leg define where the leg switches between the two states. The anterior extreme position *AEP* defines the end position of the swing phase where the leg switches to the stance phase. The position where the leg switches from stance to swing is defined by the posterior extreme position *PEP*. The most important modules of WALKNET are three networks: the Stance-net, the Swing-net, and the Target-net. For each leg one instance of the Stance- and Swing-net is used, while multiple Target-nets can be used for different walking modes, for instance forward or backward walking. The Stance- and Swing-net implement the leg control to perform the movement according to the respective state. The Target-net defines the *AEP*

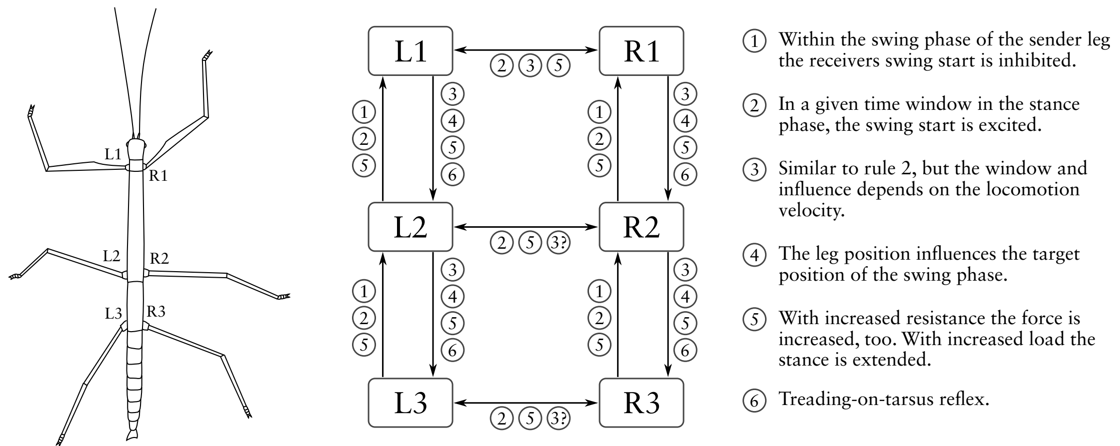


Figure 2.1: The figure depicts the coordination rules of the WALKNET model for a stick insect. The modules L1 to L3 and R1 to R3 are the single-leg controllers implementing the swing and stance movement. The connections between the modules modify the transition between the leg phases. [The figure is based on Schilling et al. (2013)].

position of the controlled leg. The *PEP* is modified via three coordination rules defined between the legs, while *Rule 4* influences the *AEP*. *Rule 5* and 6 relay on force and tactile sensory information and are not used in the default implementations of WALKNET. As a result of the coordination rules a walking gait, a phase shift between the legs, is generated. The six rules are defined as follows (compare Figure 2.1):

- *Rule 1:* As long as a leg is in the swing phase or within a velocity-dependent time delay after the start of the stance phase, it inhibits the beginning of the swing phase of connected legs by moving the *PEP* to the rear. The rule ensures that connected legs do not start the swing phase until the controlling leg can support the body.
- *Rule 2:* For a defined time window in the stance phase the *PEP* of the connected leg is moved forward. This rule excites the start of the swing movement of connected legs.
- *Rule 3:* Similar to rule 2 but the “critical” window and the influence on the *PEP* depends on the locomotion velocity. Additionally, the influence is different between ipsilateral and contralateral legs.
- *Rule 4:* The rule is implemented by a specific version of the Target-net, forwarding the current leg position to the Target-net of the connected leg, and thus defining the *AEP* of the connected leg.
- *Rule 5:* Depending on the propulsion force or load on the leg, the forces of connected legs is increased to distribute the overall load.
- *Rule 6:* The rule implements the treading-on-tarsus reflex, by performing a short back step if the leg in front is touched at the end of the swing phase.

Central Pattern Generator

The research of central pattern generators is closely related to the research of neural networks. In contrast to reflex-based models, the focus lies on identifying and understanding the neural circuit for generating locomotion motor commands. This section will only give

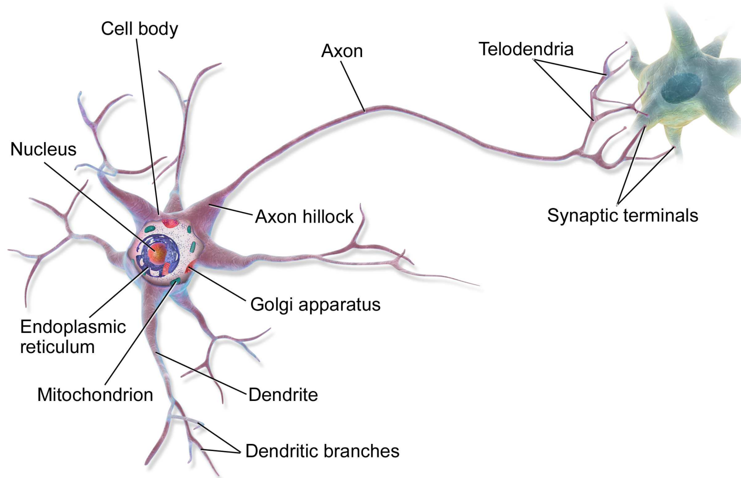


Figure 2.2: The picture depicts the structure of a multipolar neuron. The *dendrites* transfer incoming electrical or chemical signals that influence the voltage of the cell. On a high enough voltage change an electrochemical pulse is generated and transferred via the *axon*. A neuron can have multiple *dendrites* but only one *axon* that is connected at the *axon hillock*. However, the *axon* can split into multiple branches connecting multiple neurons. The synaptic signals to a neuron can either be excitatory or inhibitory.
[Source: `https://commons.wikimedia.org/wiki/File:Blausen_0657_MultipolarNeuron.png`]

a rough overview for a basic understanding of neural pattern generation, for details see Minassian et al. (2017); Ijspeert (2008); Guertin (2009); Marder and Bucher (2001). The very basic functionality of a neural cell is depicted in Figure 2.2. Different neurons, contributing to pattern generation, were found that show different membrane properties, see Figure 2.3 as examples. Beside synaptic control by other neurons, cell behaviors can also be activated, deactivated, or modulated by the chemical environment of the cells.

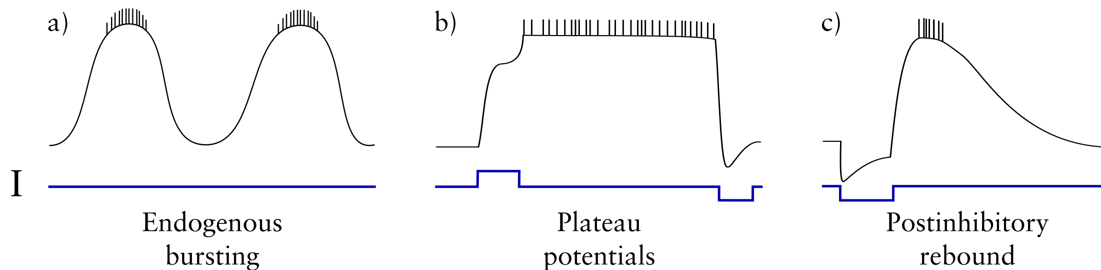


Figure 2.3: Examples of different cell properties that can contribute to generate or modulate a CPG. The upper graphs depicts the cell voltage, while the lower graphs show the current applied to the cell. a) The cell intrinsically produces a pattern without being triggered from outside. b) The cell voltage reacts to depolarizing current pulses with an increase of voltage (plateau potential). The voltage is kept even if the depolarization is stopped. If hyperpolarizing current pulses are applied, the voltage decreases to the rest state. c) A cell that reacts to inhibition with a rebound voltage after the inhibition stops. [The drawing is based on Marder and Bucher (2001).]

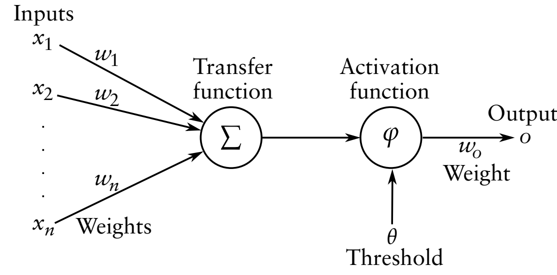


Figure 2.4: The architecture of an artificial neuron with weighted inputs, a transfer function, threshold based activation function, and a weighted output.

A commonly used model of a neuron in artificial neural networks is shown in Figure 2.4. The model summarizes the weighted input signals and produces an output if the sum is above a defined threshold (P_{th}), as defined by:

$$x' = \sum_{i=0}^{n} x_i w_i \tag{2.1}$$

$$f(\boldsymbol{x}) = \begin{cases} x', & \text{if } x' \geq P_{th} \\ 0, & \text{else,} \end{cases} \tag{2.2}$$

where x_i are the input signals, w_i the input weights, and th defines the threshold. The model is far simpler than its biological template, since in nature the processes of generating a potential are more complex and many factors can influence and modulate the cells behavior. One possibility to produce two alternating patterns as needed for cross gait locomotion, is to combine two non-bursting neurons that are interconnected by inhibiting connections. Another possible network structure found is a "pacemaker" combination where one endogenous bursting neuron generates a pattern and another neuron, which does not intrinsically produce a pattern, shows an alternating pattern if connected to the first one (see Figure 2.5).

It is accepted that pattern-generating networks can be found in vertebrates and invertebrates, but especially for mammals it is still an open question how these patterns contribute to the final locomotion control. In some examples, the motor neurons, responsible for muscle contraction and relaxation, forward or modify incoming CPG patterns or they even represent a part of the pattern producing network. Especially for mammals, modulation of patterns have to be quite complex to allow an adaption to the environment. It was shown that different networks can be responsible for different behaviors, where single neurons can also contribute to more than one of these networks. There is strong evidence that the locomotion control of mammals is much more complex than the control of insects. This is also to be expected, as the evolutionary lineages of insects/arthropods and mammals split before legs were evolved.

Biomechanics

Another important element of natural locomotion is the complex biomechanics involved. A joint, such as a knee, is driven by multiple muscles. A muscle is attached to bones via tendons which are elastic and thus can store energy. Muscles can connect two or multiple bones. They are composed of multiple segments, each consisting of multiple muscle fibers. The muscle fibers are the part that can contract the muscle to produce a desired force. To allow a bidirectional joint movement, an agonist can move the joint in

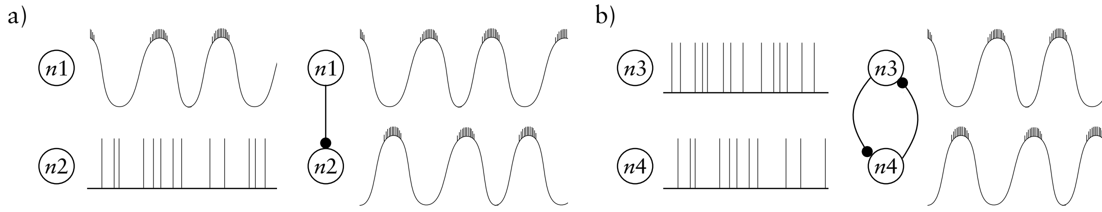


Figure 2.5: The schematic figure depicts two possibilities to generate alternating pattern by coupling two neurons via inhibitory connections. In a) a pattern generating neuron ($n1$) is used coupled with an non-bursting neuron ($n2$). To the left is the unconnected output, while the right side depicts the coupled one. In b) two non-bursting neurons ($n3$ and $n4$) are coupled by two inhibiting connections. [The drawing is based on Marder and Bucher (2001).]

one direction while the antagonist moves the joint in the opposite direction. By producing a force with both muscles a stiffness of the joint can be achieved. The elasticity of the muscles and tendons adds the compliance needed to passively adapt to the environment. This allows for example to perform a smooth touchdown by absorbing the impulse forces. The force to support the body weight can be generated after the main impulse forces are absorbed. A commonly used model to simulate muscle forces is to split the muscles into single strands between the attachment points of a muscle (see Lee et al. (2009); Pandy and Andriacchi (2010)). How these segmented muscles contribute to the torque developed at the joints is defined by a matrix. The equation for calculating the joint torques is given by:

$$\begin{bmatrix} T_1 \\ T_2 \\ \cdot \\ T_n \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & \cdot & r_{1m} \\ r_{21} & r_{22} & \cdot & r_{2m} \\ \cdot & \cdot & \cdot & \cdot \\ r_{n1} & r_{n2} & \cdot & r_{nm} \end{bmatrix} \begin{bmatrix} F_1^M \\ F_2^M \\ \cdot \\ F_m^M \end{bmatrix}, \tag{2.3}$$

where n is the number of joints, m the number of muscles, T_j is the torque of joint j, F_i^M is the force produced by muscle i, and r_{ji} is the moment arm of muscle i about joint j. A muscle can only produce a positive force, and its current force is calculated by its length, contraction or strain of the muscle in parallel direction, and the change in length (velocity). The contractile force is multiplied with the activation level of the muscle.

This kind of biomechanics modeling is also used to control characters and animals in a physical rigid body simulation. The characters are controlled by applying torques to the joints, while the model is fed back with the joint position to calculate the current muscle length. This way, the characters show very compliant behaviors, stabilizing the whole control. In comparison to a robotic system with classical electrical motors, a position or speed command is used and the motor generates the torque needed to achieve the desired target value. Compared to the muscles, the motors behave like agonist and antagonist muscles producing as much stiffness as possible. However, many robotic systems nowadays measure the torque at the actuators and allow to control the torque generated by the motor. With this torque controlled actuators the load distribution can be optimized allowing to compensate for an uneven surface. But still the actuators have a relatively high damping due to the mechanical gearing and the electrical coupling of the coils. A mechanical behavior more similar to natural systems can be achieved with serial elastic actuators combined with torque measurement. This way, a torque applied to the spring (the elastic element in the actuator) can be used to compensate the damping of the actuator. These actuators can absorb the impulse force at the touchdown phase and behave more similar to the biological counterpart.

2.2 Evolution and Optimization

2.2.1 Natural Evolution

A modern introduction of the theory of evolution is given by the book *Almost Like a Whale: The Origin of Species Updated* authored by Steve Jones (Jones (2000)). This section provides a short summary, roughly introducing the theory of evolution. A timeline with key events in the development of life is shown in Figure 2.6. How life began is still an unanswered question. The earth is approximately 4.5 billion years old and life started about 3.8 billion years ago by creating first one-cellular life-forms which were able to reproduce themselves. It took another 1.7 billion years until first multicellular life-forms evolved. The next milestone was about 1.8 billion years ago, when first cells with a nucleus evolved. The first arthropods evolved 570 Mya followed by fish 530 Mya. The Cambrian explosion took place in the period from 545 to 495 Mya. Within this period all major forms of life evolved including animals that swam, crawled, hunted, and hid away. Dinosaurs populated the earth for more than 150 million years within the period of 225 to 65 Mya. The human species evolved only 200 thousand years ago and thus settled on the earth 0.13 % the time dinosaurs lived and only 0.004 % of the whole time the earth exists.

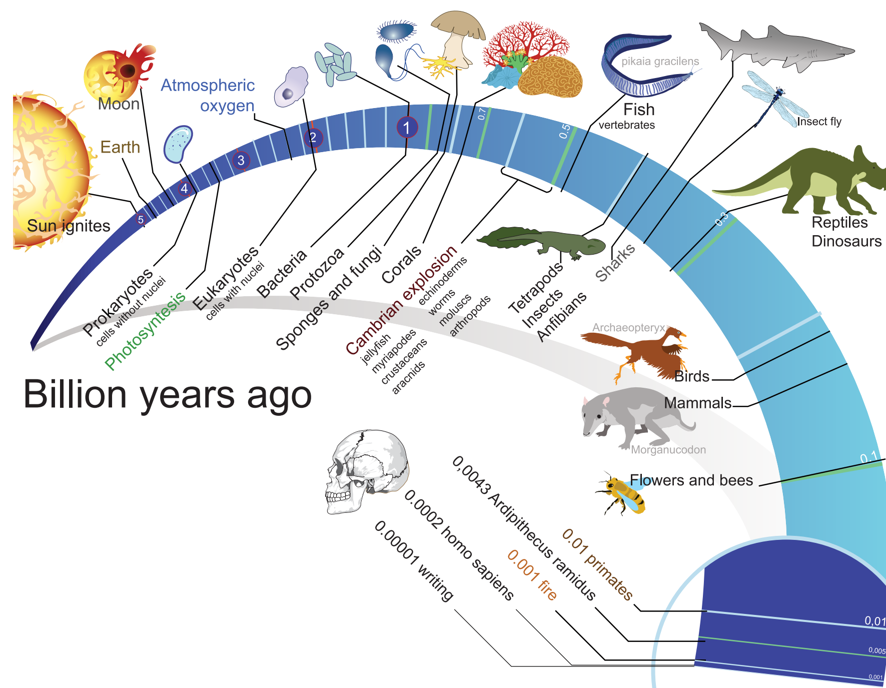


Figure 2.6: The timeline of evolution of life.
[Source: `https://commons.wikimedia.org/wiki/File:Timeline_evolution_of_life.svg`]

The basis of all life is the genetic encoding (*DNA*) that came into place within the first 0.7 billion years. A bacterial cell is the simplest organism defined by *DNA*, which already includes thousands of different protein molecules that have to work in a coordinated way to enable the cell to survive. In contrast to bacterial cells, the cells of multi-cellular systems, like mammals, include a nucleus embedding the *DNA*. More than 300 different types of cells can be found in the human body. Some cells produce or represent the body, like bones, hair, or organs, while others produce substances the organism needs.

The substances are produced or processed by enzymes which are a collection of proteins performing specific jobs. Proteins are molecules composed by a combination of twenty different amino acids. Up to a few hundred amino acids can be combined into the structure of a protein. The sequence of the sub-units is defined by the *DNA* of the gene that produces the protein.

DNA is made of four types of bases, which are aligned in a paired double-string in a double-helix structure. The fruit-fly (Drosophila melanogaster), used for much research in genetics, has about 14,000 genes with about 120 million letters of *DNA*. Human *DNA* has about 25 times as many letters, while the number of different genes is still unknown. Less than 2 % of the human *DNA* is responsible for coding the proteins and it is unknown how much of the rest is important for the organism to live. Some of the non-coding *DNA* is responsible for starting and stopping the production of proteins. The whole *DNA* defines the plan of how the organism is created out of one single cell through the cell division process, whereby each cell includes a full copy of the original maternal and paternal *DNA* set. During the cell division process the *DNA* is first replicated within the nucleus. Since the replication process is a chemical one, it can produce errors in the copy. Within the process, enzymes verify the new *DNA* string and can correct errors. After replication the copy is transferred out of the nucleus and the cell division process continues. The errors that are produced and not corrected within the *DNA* replication are the source of mutation. In the end, the probability of mutation with an influence on the development of the organism differs between different functions. For instance, the probability for albino mutations is much higher than the probability for an additional extremity growing somewhere at the corpus. The difference of the same gene strings of different people is less then 0.1 % of the *DNA* letters, compared to about 1 % difference between human and chimpanzee. The principle of the complex machinery of the cells with its *DNA*, proteins, and cell division is shared between all multi-cellular life on earth.

Adaptation of a species is mainly driven by natural selection or survival of the fittest. The individuals that produce more offspring either by reproducing themselves faster or by living longer than others pass more of their genes to the next generation. This also includes the ability to protect themselves from being caught by a predator and vice versa. Another factor is genetic drift, which occurs if a mutation that has no influence on fitness and is thus not selected against, spreads within a species over multiple generations. Altogether, the adaptation of a population to its environment is a very slow process of many small adaptations over many generations.

2.2.2 Artificial Evolution

The research on artificial evolution is done with different purposes. On one hand, models can be used to explain or understand the processes observed in nature. On the other hand, evolutionary models can be used to solve many problems concerning design or control . It has been shown for several applications that evolutionary methods can produce human competitive results, many examples are summarized by Koza (2010). The umbrella term for the whole research field is commonly *Evolutionary Computation* or *Genetics Computation*. The field can be split into four sub-groups, as shown in Figure 2.7. However, the subgroups are not clearly separated and a lot of research can be ordered in between different groups, illustrated by the overlapping areas in the overview. While the subfields of *Evolutionary Strategies* and *Genetic Programming* focus predominantly on solving given design or control problems, the fields of *Genetics Algorithms* and *Neuroevolution* have many contributions for understanding natural processes. What all

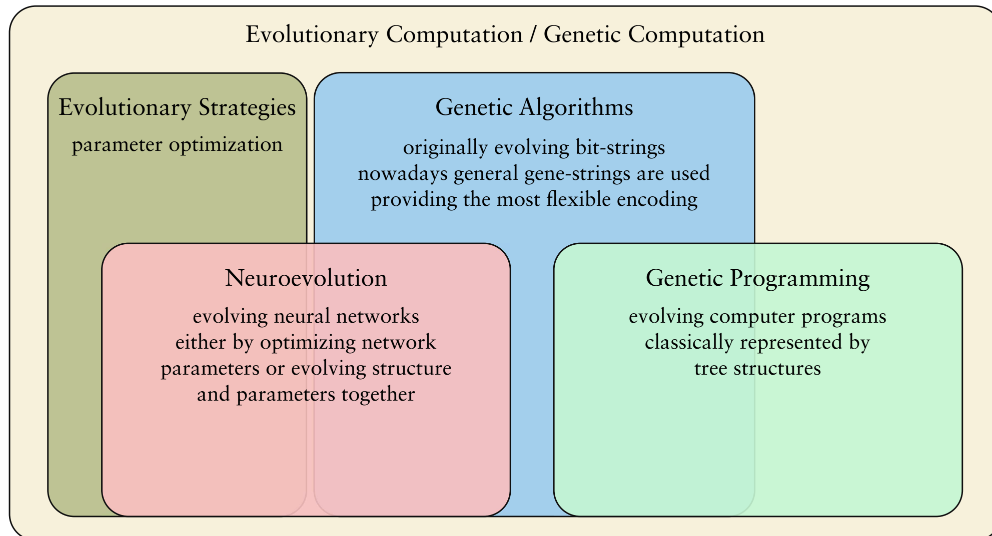


Figure 2.7: Overview of research fields of Evolutionary Computation.

subfields have in common is the basic principle of managing one or more populations where every individual is evaluated and a subset of individuals is selected for generating the followup populations. Whether one or more populations are managed and how many parents are selected depends on the specific methods used and is part of the research in all subfields. The following subsections introduce the four groups more in detail. Finally an overview of the application of artificial evolution applied to legged robotic systems is given.

Evolutionary Strategies

Evolutionary Strategies were introduced by Rechenberg (1964, 1973) and are designed to optimize real-valued vectors. They represent a heuristics search in the parameter space, the search space for parameter optimization, able to find adequate global solutions for many problems. In most implementations a step width σ is used to control how different the offspring can become from its parents. Either the step width is constant or it is adapted from one generation to the next. The most basic algorithm is given by the code example Listing 1. More advanced implementations handle a separated step width for

```
1  Input: population P, step width σ
2  for x_i in P do
3    x_i ~ U(0,1)   # sample initial parameters from uniform distribution
4  end for
5  # evaluate parameter with f : R^N → R
6  while True do
7    # select best individual as parent for the next population
8    p ← x_0
9    for x_i in P do
10     p ← (f(x_i) < f(p)) ? x_i : p
11   end for
12   # generate next population from selected parent
13   for x_i in P do
14     x_i ~ x_i + N(0,σ()^2)
15   end for
16 end while
```

Listing 1: Basic optimization process of Evolutionary Strategies.

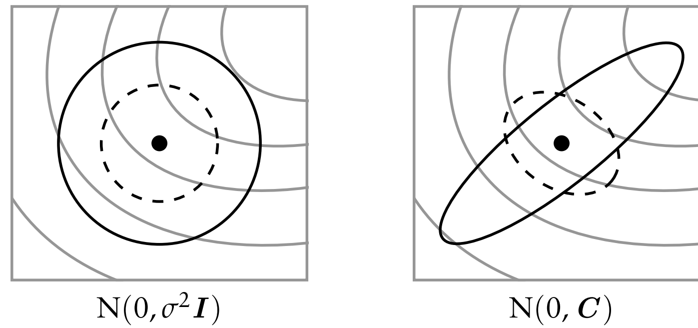


Figure 2.8: The two pictures depict the behavior of different mutation approaches. The grey lines in the background illustrate the fitness landscape, which is improving in the direction of the upper right corner. The first image shows two samples of a normal distribution with $\sigma \in \mathbb{R}^+$. The second example depicts two samples using the covariance matrix.
[The drawing is based on Hansen and Ostermeier (2001).]

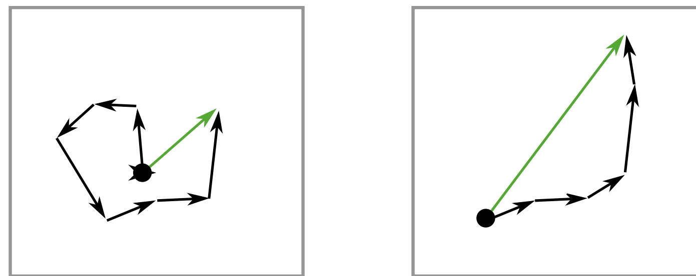


Figure 2.9: The two pictures illustrate two examples of a path taken in the search space. The black arrows depict the single step taken by the search algorithm while the green arrow represents the cumulative path which can be used to optimize the step width adaptation.
[The drawing is based on Hansen and Ostermeier (2001).]

every dimension of the parameter vector or even generate and update a covariance matrix to learn parameter correlations. A very popular implementation is the Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) introduced in Hansen and Ostermeier (2001). The implementation adds two main features influencing how the algorithm operates in the search space. The first feature uses a covariance matrix to learn parameter correlations and influence the distribution of the offspring. Figure 2.8 depicts the influence of the covariance matrix in comparison to normal-distributed offspring. The second feature makes use of a cumulative step width calculation to adapt the step width used to generate a new offspring. This principle is illustrated in Figure 2.9.

Genetic Algorithms

Genetic Algorithms became popular with the work of Holland (1992) and represent the most generic subfield of evolutionary computation. While the research originally focused on evolving bit-strings representing the genotype, today it covers any kind of gene string with a non-restrictive conversion of the genotype to the phenotype. The other subfields of Evolutionary Computation differ mainly in the more restricted usage of the genotype, like Evolutionary Strategies which use real-valued vectors as genotypes to perform parameter optimization.

Common operations on the genotype used for reproduction are the *mutation* and the *crossover* operations. In mutation, the properties of a single gene string that are stored in the genes are varied randomly, mostly by adapting parameters similar to the ES meth-

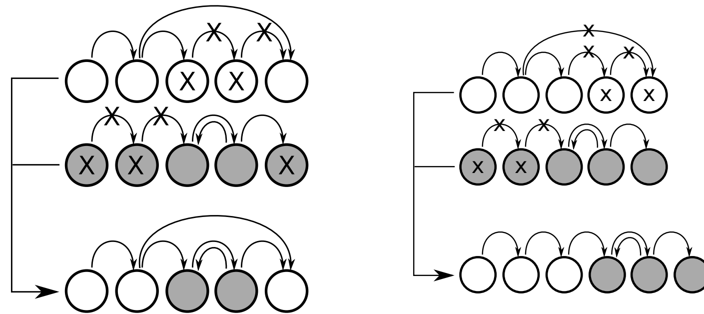


Figure 2.10: Two examples of the crossover operation. In each projection the upper two strings are combined into the lower new ones by eliminating the marked edges and nodes.

ods. The crossover operation combines the genes of two parent strings. Two examples of the crossover operation are illustrated in Figure 2.10. The genes in the genotype can, for instance, decode the morphology and the control of a virtual creature as done a lot in artificial life research, as done by Sims (1994); Mazzapioda et al. (2009); Auerbach and Bongard (2011). Many contributions to the field use genetic algorithms in a biological scientific context as done by Parsons et al. (1993); Izquierdo and Lockery (2010); Custódio et al. (2014); Luque-Baena et al. (2014).

Genetic Programming

The field of *Genetic Programming* was established by Koza (1992). Genetic Programming has its focus on evolving computer algorithms. Therefore, the classical representation is a tree structure composing mathematical operations (see Figure 2.11). Each node in the tree can have two sub-trees and connects to one node on the above tree layer. Nodes that have no sub-trees define inputs to the algorithm or can introduce constant parameters. The root node in the tree represents the output of the overall algorithm. To allow multiple outputs of an algorithm one tree can be evolved per output. Many contributions to Genetic Programming are listed in Koza (2010).

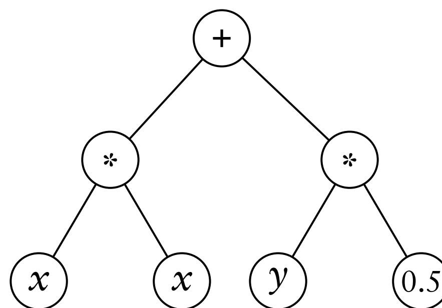


Figure 2.11: An example of a possible GP-tree representing the function $f(x) = x^2 + 0.5y$.

Neuroevolution

In the field of *Neuroevolution* (Floreano et al., 2008; Yao, 1999) the phenotype is represented by an artificial neural network (ANN). The parameters, or structure, or both of the ANN are developed by an evolutionary algorithm. Generally, neuroevolutionary methods are comparable to reinforcement learning algorithms, where the fitness value of an individual is the accumulated reward received by the individual after it has operated in a

given environment (Whitley et al., 1993). Therefore, the ANN input is the current state of the reinforcement agent while the output of the network represents the action selected by the current policy represented by the neural network. This way, *Neuroevolution* can be applied to any episodic reinforcement learning tasks. Additionally, *Neuroevolution* is one possible approach to apply reinforcement learning to continuous problems.

Many contributions successfully evolve only the parameters of an ANN by keeping the structure (i. e. network topology) fixed (Wieland, 1991; Saravanan and Fogel, 1995; Moriarty and Miikkulainen, 1996; Gomez and Miikkulainen, 1997; Igel, 2003; Gomez et al., 2008; Heidrich-Meisner and Igel, 2009). However, the performance of these approaches highly depends on the human-designed network structure, which has to fit more or less precisely to the specific application. On the other hand, methods that also evolve the network structure can be subdivided by the mapping from genotype to phenotype used, distinguishing between “external”, “explicit” or “implicit”. For external mapping each gene in the genotype directly represents an element in the phenotype. The genotype could consist of node and edge genes representing ANN nodes and connections as used by (Angeline et al., 1994; Stanley, 2004; Kassahun et al., 2007b). In case of explicit encoding, the genotype represents a blueprint to generate the phenotype, while this development process is specified in the genotype, as done by Kitano (1990); Sendhoff and Kreutz (1999); Gruau (1994); D’Ambrosio and Stanley (2007); Koutník et al. (2013). Implicit mapping has the strongest resemblance of natural evolution, where the phenotype emerges from the interaction and activation patterns of the genes in the genotype (Vaario et al., 1997; Nolfi and Parisi, 1991; Dellaert and Beer, 1996; Jakobi, 1995; Bongard and Pfeifer, 2001; Reisinger and Miikkulainen, 2007; Bentley and Kumar, 1999; Mattiussi, 2005).

A commonly-used implementation for evolving ANNs is the NEAT algorithm which can be applied to many optimization problems without the need to configure it to the specific problem. NEAT stands for “NeuroEvolution of Augmenting Topologies” and was developed to improve previous neuroevolutionary methods by creating network topologies in parallel with the parameter optimization starting with minimal topologies (see Stanley and Miikkulainen (2002)). NEAT uses a direct encoding for its ANNs, managing node genes and connection genes separately. Network structure is built up starting with no hidden nodes at all and adding both nodes and connections between random nodes through mutation. Connection weights are subject to mutations, too, thus allowing established connections to be modified in order to optimize the network’s performance. A specialty of NEAT are historical markers (*innovation numbers*) that are used to track the generation of new genes, thereby allowing correct alignment upon recombination of genomes. Using these gene identifiers, the population of networks is subdivided into species by genome similarity, thus allowing structural innovations to evolve in a semi-protective environment with less competition than in the whole population.

Since its initial conception, NEAT has been used in various ways to develop robot controllers. An extension of NEAT is HYPERNEAT, which focuses on generating patterns in the outputs of the evolved ANNs. HYPERNEAT uses a fixed-structure neural network where the parameters are approximated by a compositional pattern-producing network (CPPN) evolved with NEAT. The main advantage of CPPNs are their tendency to produce symmetric patterns in the weights of the control network. That this can be useful for the control of walking machines was demonstrated by Clune et al. (2009), who evolved a simple quadruped robot controller.

Evolution Applied on Walking Control

One of the first papers on the evolution of walking patterns is the work of Beer and Gallagher (1992). In their work, artificial evolution was used to evolve a static gait for a simulated, kinematically simple six-legged robot. Each leg was controlled by a five-neuron, fully-connected, continuous-time neural network, and the corresponding neurons of the controllers of the neighboring legs were connected. The weights of the six controllers were constrained to be identical, yielding a total of 50 parameters to be optimized. The used genetic algorithm was able to evolve a tripod gait in all conducted runs.

Lewis et al. (1992) transferred the approach of Beer and Gallagher onto a physical robot for the first time. While Lewis et al. did not perform the evolution itself in the real world (but only tested the evolved gait on a physical robot), Gomi and Ide (1998) evolved a gait for an octopod robot completely on-line. While the control architecture is fixed in most approaches, Gruau and Quatramaran (1996) used the cellular encoding to evolve both the weights and the structure of a controller for an octopod robot. Bongard and Pfeifer (2002) investigated a method to measure the influence of morphology for the development of behaviors, in which the simulated agents have the same neural network architecture, number of sensors, and actuators. Bongard et al. (2006) described a method based on self-models to assemble new behaviors to recover from damage.

In contrast to the approaches discussed above, in which a static gait was evolved, Hornby et al. (1999) and Röfer (2005) evolved a dynamic gait for AIBO [2] robots. They used a genetic algorithm to optimize the parameters, and evolution was performed online on real robots as in the work of Gomi and Ide. Zhang and Chen (2007) achieved the fastest gait for these robots by combining a learning method with evolutionary techniques. Hornby used a central pattern generator (CPG) model for the controller, which allowed oscillatory patterns to be generated in the absence of external stimuli. This model was also used by Reil and Husbands (2002), who used evolutionary methods to optimize the parameters of CPGs on a simulated bipedal robot and showed that the quality of an evolutionary method depends heavily on the chosen fitness function. Ito and Matsuno (2002) used evolutionary techniques in a redundant, multi-legged robot, although only simulated results were presented.

Related work using a "co-evolution" approach was done by Ventrella (1994) for animated characters. In a simulation environment Sims (1994) evolved creatures that compete in a three-dimensional world. The winner received a higher fitness value which allowed it to survive and reproduce itself. Unlike in this thesis, the applications were not meant to build real robots or were developed on less complex systems. Endo et al. (2003) used co-evolution to develop the morphology and walking pattern for a humanoid robot. Their aim was to walk on flat ground and the development of one walking pattern. Altogether, little research deals with more than two or three degrees of freedom per leg.

2.2.3 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a heuristic method to optimize a set of parameters. It was introduced by Kennedy and Eberhart (1995). It is quite robust and can deal with an unstructured, noisy, and even time-dependent fitness landscape. PSO is inspired by the behavior of swarm of birds and shoal. Individual particles move within the search space and their direction is influenced by the best parameters of the individual particle (best local position) and the best parameters of the whole swarm (best global position). The

[2] AIBO: Very popular quadruped robot from Sony.

algorithm itself is rather simple including four strategy parameters, the number of particles N, the damping of the particle velocities ω, and the weights of the local and global attraction points φ_p and φ_g. The parameters allow to control the exploration versus exploitation. Since PSO does not store any information about the fitness landscape other than the local and global best positions, it can adapt to a dynamically changing fitness landscape. Like in most heuristic methods, the strategy parameters have to be tuned to the application to prevent the algorithm from converging too early to local optima. The core of the PSO algorithm is the velocity update of a particle, which is defined by:

$$\boldsymbol{v}_i = \omega \boldsymbol{v}_i + \varphi_p \boldsymbol{r}_p \cdot (\boldsymbol{p}_i - \boldsymbol{x}_i) + \varphi_g \boldsymbol{r}_g \cdot (\boldsymbol{g} - \boldsymbol{x}_i)\,, \tag{2.4}$$

where $\boldsymbol{p}_i$ is the local best position of the *ith* particle and and $\boldsymbol{g}$ is the global best position. $\boldsymbol{x}_i$ is the current position of the particle and $\boldsymbol{v}_i$ its current velocity. $\boldsymbol{r}_p$ and $\boldsymbol{r}_g$ are normal-distributed random vectors. The algorithm itself is shown in Listing 2.

```
1  Input: particle swarm P, damping ω, local and global attraction weight φ_p and φ_g
2  for x_i in P do
3    x_i ~ U(0,1)   # sample initial position from uniform distribution
4    v_i ~ U(0,1)   # sample initial velocity from uniform distribution
5    p_i ← x_i   # init local best
6  end for
7  # evaluate particles with f : R^N → R
8  g ← argmin_i f(P[i])   # init global best with best particle in P
9  while True do
10   for x_i in P do
11     r_p, r_g ~ U(0,1)   # sample two uniform distributed vectors
12     v_i ← ωv_i + φ_p r_p · (p_i − x_i) + φ_g r_g · (g − x_i)   # update particle velocity
13     x_i ← x_i + v_i   # update particle position
14     p_i ← (f(x_i) < f(p_i)) ? x_i : p_i
15     g ← (f(x_i) < f(g)) ? x_i : g
16   end for
17 end while
```

Listing 2: Core algorithm of PSO.

2.3 Robotic Systems

This section describes the existing robotic systems that are used for the experiments executed on real systems. The three systems are SPOT, SPACECLIMBER, and CHARLIE. For all three systems locomotion controllers are evolved in simulation and tested on the real robots. The SPACECLIMBER system is also used to measure the quality of the simulation MARS. The systems are selected due to their increasing complexity, which is also the order of the subsections introducing them. The complexity is increasing concerning the system's number of *dof* as well as by the precision of the actuators and the consequential required controller quality to prevent actuators working against each other.

2.3.1 SPOT

The four-legged robotic system SPOT was developed in the robotics laboratory at the University of Bremen. The robot has a weight of about 35 *kg* and a size of 70 *cm* length, 45 *cm* width, and 60 *cm* height. Each of its four legs has six joints combined with two neck and one hip joint, powered by two motors, amounting to 27 independently controllable joints. The toe joints (joint 6 in Figure 2.12) have a diameter of 22 *mm* and a

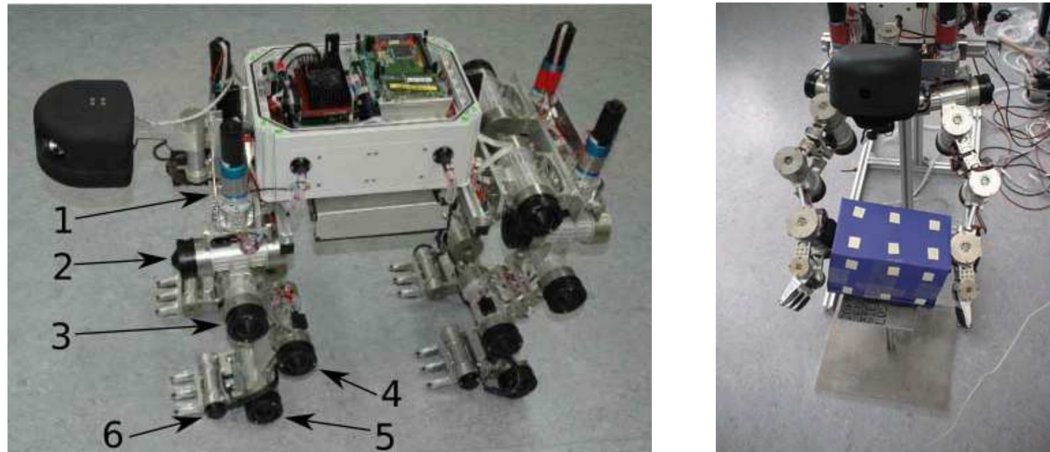


Figure 2.12: *Left*: The four-legged robotic system SPOT. The joints of the front left leg are labeled with numbers for reference in the experiment chapter. *Right*: SPOT performing dual-arm manipulation to recognize objects with different weights and visual appearance. Source left: Römmermann et al. (2008); Source right: Kassahun et al. (2007a)

peak torque of 3 *Nm*. All other joints have a diameter of 38 *mm* and a peak torque of 17 *Nm*. The robot shares many similarities with the ARAMIES robot Spenneberg et al. (2005a), with regard to the physical construction and the software architecture. The main difference in the kinematic model is the introduction of the hip joint and the different dimensions of the robot's corpus. The hip joint enables the robot to bend at the waist and makes it possible, for instance, to sit on its hind legs or to climb steep slopes. For the control of the system, the robot's micro-kernel M.O.N.S.T.E.R.[3] (Spenneberg et al. (2005a)) is used, integrating the PCR control approach (Spenneberg (2006)). M.O.N.S.T.E.R. was originally developed in the ARAMIES project as an operating system running on various micro-controllers. It mainly combines properties of real-time operating systems with behavior-based programming by featuring hard and soft periodic processes, preemptive reflexes, behavior processes on different execution frequencies and a background process. Within the PCR control approach, the CPG is realized by designing Bezier-Splines to define rhythmic patterns of the joints. However, due to the six degrees of freedom (*dof*) per leg of the SPOT, manually designing the Bezier-Splines is extremely complex and time-consuming. On the other hand, due to the legs' complexity, they can be used not only for traversing a variety of terrains, but also for manipulating objects.

In Kassahun et al. (2007a) the system was used to distinguish between objects with same shape but different visual appearance and of different weight. To compare the performance of the classifier using exteroceptive and/or proprioceptive data to recognize objects, different scenarios were used. To perform the experiments the robots corpus was mounted on a frame to allow a reproducible test setup for the dual-arm manipulation as shown in Figure 2.12 on the right hand side.

2.3.2 SPACECLIMBER

The SPACECLIMBER system is a six-legged walking robot developed at the RIC/DFKI within the SPACECLIMBER project that was finished in 2010. SPACECLIMBER, shown in Figure 2.13, has a size of approximately 92.6 *cm* length, 94 *cm* width, and 36.5 *cm* height in it's default standing posture. It has four rotary actuators per leg with a repeatable peak

[3]**M**icro**k**er**n**el for **s**cabrous **t**errain **e**xploring **r**obots

Figure 2.13: The SPACECLIMBER system in an environment simulation the Mars surface.

torque of 28 *Nm* at 50 *rpm*. The actuators have a length of 11 *cm* and a diameter of 6.4 *cm*. The morphology development of SPACECLIMBER was supported by an optimization within the simulation. Therefore, a parameterized simulation model was set up by simulating three leg pairs connected by bars as shown in Figure 2.14. For each pair of legs the bar length and position together with the upper and lower leg lengths was adapted in a given parameter range.

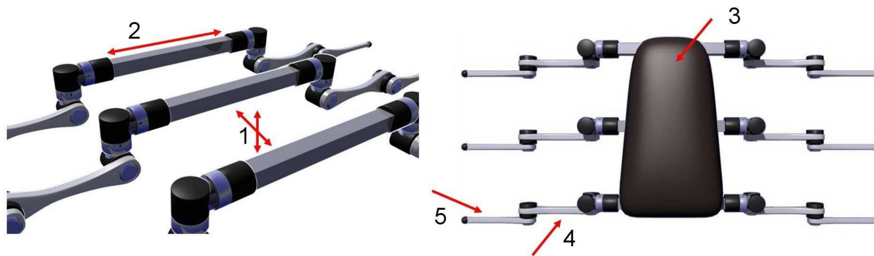


Figure 2.14: Illustration of the SPACECLIMBER morphology parameters to adapt it's structure. The parameters are the position (1) and the length (2) of each bar, the center of mass position (3), and the link length of the first (4) and second leg segment (5). [Images source: Römmermann et al. (2009b)]

Each morphology configuration (parameter set) was evaluated in three environments, shown in Figure 2.15. The robot had to move forward in the test environments that only differ in the inclination of the planar surface. In case of the flat environment the friction properties of the surface changed every 2 *m*. An individual was tested on the slopes for 10 *s* and on the flat surface for 30 *s*. A model-based parameterized central pattern approach was used to control the robot. A set of 13 control parameters were optimized for every test environment separately, resulting in 17 parameters for the morphology and $13 \cdot 3$ parameters for the control. The overall 56 parameters that represented one individual are shown in Figure 2.16.

Five parameters were measured in the simulation to evaluate a parameter set: the distance travelled d, the average torque ϑ and the average load l of the joints, a value g indicating if parts of the robot other than the feet had ground contact, and the center of mass of the robot m_c. The fitness function was defined as the sum of four terms. The first term $\frac{-\vartheta}{d}$ assessed the energy efficiency. The second term $\frac{-l}{d}$ was based on the load incurred on the joints over the covered distance. The third term $-c_n\tau$ checks whether parts of the robot other than the feet came into contact with the ground and a constant negative reward was added in that case ($c_n = 1000$; in case of a contact τ is 1 otherwise it is 0). The last term m_c^{out} was based on the percentage of time in which the center of mass

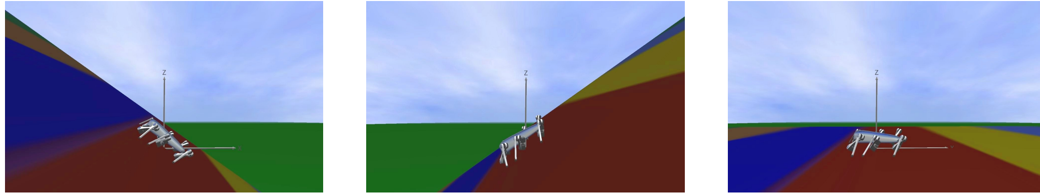


Figure 2.15: To optimize the SpaceClimber morphology every parameter set was evaluated in three environments with planar surfaces. The three setups differed in the angle of the surface with -30°, 30°, and 0°. In the last 0° setup, the colored areas represented different friction properties. [Images source: Römmermann et al. (2009b)]

Morphology	Walking pattern 1	Walking pattern 2	Walking pattern 3

Leg segment length
Bar length
Bar position on x and z
Body center on x and z
Height of the body
Offset at first joint
Offset at second joint
Scaling factor of amplitude
Distance between foot and body
Scaling factor of swing height
Scaling factor of walking frequency

Figure 2.16: Graphical illustration of the parameter set, which was optimized. [Images source: Römmermann et al. (2009b)]

was out of the stability margin, as depicted in Figure 2.17. The resulting fitness function is described by the following equation:

$$fitness = \frac{-\vartheta}{d} + \frac{-l}{d} - c_n g - m_c^{out}. \tag{2.5}$$

Over 85 single evolutions were performed. Every evolution produced a result which was able to travel about 0.3 m/s on the planar surface, about 0.3 m/s on the 30° slope, and more than $0.5 m/s with an inclination of -30°. The results were classified and the resulting morphology of SpaceClimber was derived from an average morphology of the best group.

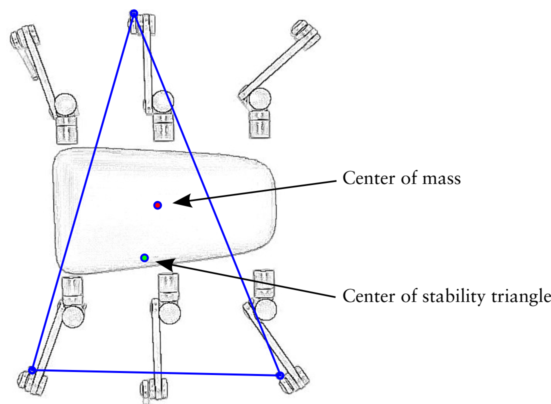


Figure 2.17: Using the stability margin as evaluation criteria. [Images source: Römmermann et al. (2009b)]

Only the relevant parts of the SpaceClimber controller that are used to generate locomotion behaviors in the experiment chapter of this work are described in detail. A comprehensive description of the control concept is given in Bartsch (2014). The SpaceClimber controller is a combination of different modules including, amongst others, a posture controller, a state pattern generator, and a number of reflex behaviors. The

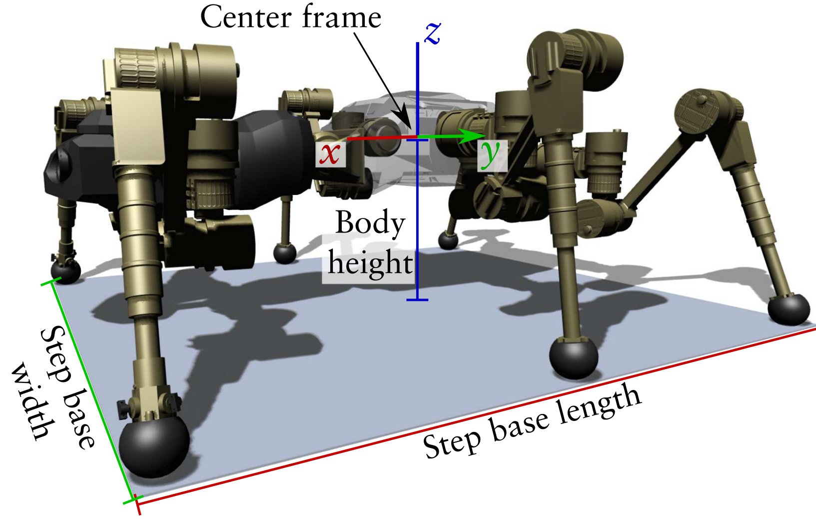


Figure 2.18: Posture parameters of the SPACECLIMBER controller.

posture controller generates the initial posture of the system by defining the height of the center frame and a base width and length of the polygon defined by the feet positions. Additionally, the center frame can be shifted on the xy-plane (forward and lateral) and orientations on all axes can be defined. Figure 2.18 depicts the defined base posture and its parameters. The center frame of the robot is also used as a measurement point for evaluation functions like tracking the robots speed or stability. The output of the posture controller defines the foot positions relative to the center frame. An inverse kinematics module is used to calculate the corresponding joint positions.

The model-based central pattern approach is implemented by a state machine running for every leg. The values generated by the pattern generator are added to the values of the posture controller, adapting the desired foot positions. The states implemented are a *stance*, *lift*, *shift*, and a *touchdown* phase. The timings T_{Lift}, T_{Shift}, T_{Down}, and T_{Step} are defined in milliseconds by parameters of the controller. The swing time T_{Swing} is the sum of the lift, shift and touchdown time, while the stance time T_{Stance} is given by the step time minus the swing time according to Equation 2.6 to 2.7.

$$T_{Swing} := T_{Lift} + T_{Shift} + T_{Down} \tag{2.6}$$
$$T_{Stance} := T_{Step} - T_{Swing}. \tag{2.7}$$

During the stance phase, all feet are moved with the same velocity in the xy-plane for linear movements and they are rotated by the same angle for rotations around the z-axis of the robot's center frame. The xy-velocities are defined by a forward and lateral step size R_x and R_y given in meters, while R_{yaw} defines the orientation velocity in degrees per step cycle. The calculations of the velocities are defined in Equation 2.8 to 2.10.

$$\dot{v}_x := \frac{R_x}{T_{Step} - T_{Shift}} \tag{2.8}$$
$$\dot{v}_y := \frac{R_y}{T_{Step} - T_{Shift}} \tag{2.9}$$
$$\dot{\gamma} := \frac{R_{yaw}}{T_{Step} - T_{Shift}}. \tag{2.10}$$

The swing phase is split into the lift, shift, and touchdown sub-phases. Within the lift phase the movement of the stance phase is continued to prevent a slipping behavior before the leg leaves the ground. A z-movement is added to the leg via linear interpolation of the lift time and the desired step height R_{height}. Within the shift phase, the leg is moved to the next stance start position, the anterior extreme position AEP, given in Equation 2.11 to 2.13.

$$AEP_x := FO_x + 0.5R_x \tag{2.11}$$
$$AEP_y := FO_y + 0.5R_y \tag{2.12}$$
$$AEP_{yaw} := 0.5R_{yaw}. \tag{2.13}$$

The touchdown phase performs an inverted linear interpolation on the z-axis and applies the stance velocity on the xy-plane as done in the lift phase. By already applying the stance velocity, a smooth transition between the swing and stance phase can be achieved. The phase shift between the legs is defined by the parameter PS used to calculate the phase displacement for the legs PD_0 - PD_5 as defined in Equation 2.14 to 2.19.

$$PD_0 := 0 \tag{2.14}$$
$$PD_1 := 0.5 \tag{2.15}$$
$$PD_2 := 0.5 + \frac{1}{6}PS \tag{2.16}$$
$$PD_3 := \frac{1}{6}PS \tag{2.17}$$
$$PD_4 := \frac{2}{6}PS \tag{2.18}$$
$$PD_5 := 0.5 + \frac{2}{6}PS. \tag{2.19}$$

If the phase displacement between two legs i and j is zero ($PD_i = PD_j$) the leg phases are aligned. The phases are maximally misaligned if $|PD_i - PD_j| = 0.5$, resulting in an alternating gait between the two legs. The controller includes exception handling, which is mainly needed when changing the locomotion parameters and thus performing a transition between different walking gaits. This exception handling is mainly based on a neighboring leg rule and it is not further explained in this work, since no gait transitions are evaluated in the experiment chapter. The default parameters of the SPACECLIMBER controller are given in the appendix Section A.1.1.

2.3.3 CHARLIE

The CHARLIE robot is a four-legged walking system developed at the RIC/DFKI within the iSTRUCT [4] project that was finished in 2013. CHARLIE, shown in Figure 2.19, has a size of approximately 66 *cm* length, 43 *cm* width, and 77 *cm* height in its default standing posture. The overall mass is about 21.5 *kg*, whereby a single front leg has a mass of about 3.4 *kg* and a rear leg about 4.1 *kg*. The morphology of the system is biologically inspired, displaying ape-like properties in its body proportions and rear feet design. The system introduces a 6 *dof*-spine that allows to increase the workspace of the legs and thus supports the locomotion possibilities. The rear feet shape is inspired by the feet of apes including two ankle joints, a passive heel joint, and two toe joints. Additionally, each

[4] iSTRUCT: `https://robotik.dfki-bremen.de/de/forschung/projekte/istruct.html`

Figure 2.19: The CHARLIE robot climbing on the crater of the space exploration hall at the RIC/DFKI.

rear foot can provide its own support polygon via pressure sensors in the sole and local data processing. The actuator type in the hip, shoulders, and knees is a successor of the SPACECLIMBER actuator providing about 25 *Nm* with a maximal speed of about 60 *rpm*.

Figure 2.20 shows the degrees of freedom of the legs and the spine. To control the parallel kinematics of the spine, an inverse kinematics solution is used that allows to command the spine in all six *dof* at a reference coordinate system (spine frame). The maximal range of motion of the spine is shown in Table 2.1. Also the head can be moved by a similar parallel kinematic structure of the neck (neck frame). Since the *dof* of the neck are not used for evolving a locomotion control in this work, the neck frame is not shown in Figure 2.20. The frames of the force torque sensors at the feet and the frame of

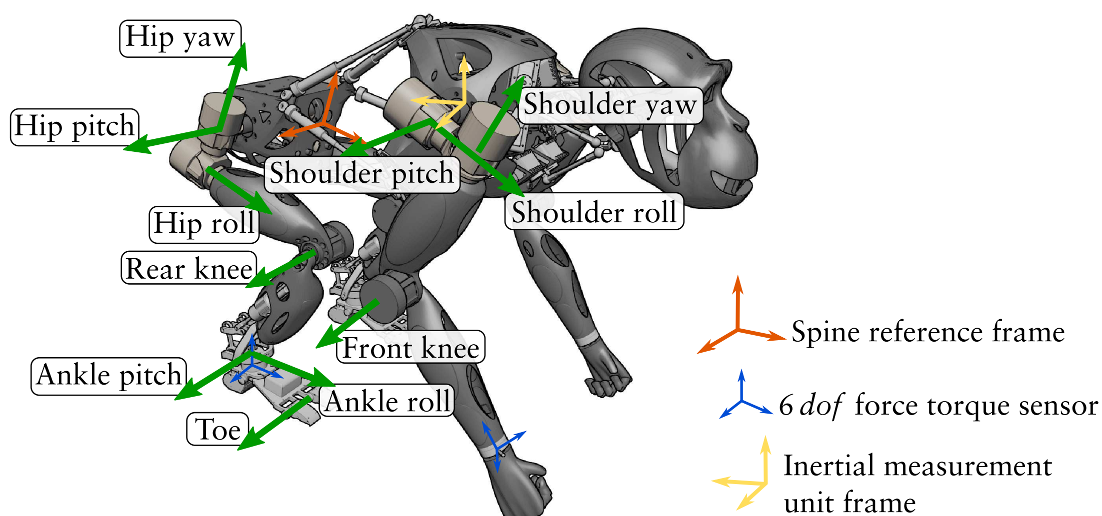


Figure 2.20: Illustration of the CHARLIE's degrees of freedom including the spine frame and the force torque sensor frames.

the inertial measurement unit (IMU) are displayed in Figure 2.20. The IMU provides the orientation of the system's front body and is used together with the force torque sensors to evolve an adaptive behavior. The other sensors of CHARLIE, like the cameras or the feet contact sensors, are not used within this thesis.

Table 2.1: The maximal range of motion of CHARLIE's spine. The values on the single *dof* are further limited by the current pose on the other axes.

Axis	Translation (*mm*) min / max	Rotation min / max
x	0 / 44	-28° / 28°
y	-60 / 60	-18° / 18°
z	-52 / 56	-16° / 16°

For the control of the system, a similar concept is used as described in the introduction of SPACECLIMBER. Only static locomotion behaviors are implemented, meaning the robot is statically stable in any phase within the walking pattern. Therefore, a body shift is used, moving the center of mass of the system towards the center of the support polygon of the next three legs performing the stance phase, while the fourth leg can perform the swing movement. Another difference is an interpolation used to smoothing the swing trajectories of the legs. The control details are not further explained because the original control is not used for the experiments of this thesis. For further details on the mechanical design or the control concept see Kühn (2016).

Chapter 3

Simulation MARS

As introduced in Chapter 1, a simulation is a powerful tool for testing and developing approaches to generate locomotion behaviors for legged robots. Many experiments are required to test and compare different learning methods and behavior representations, which would be too time consuming on a real system. Additionally, due to the risk of breaking the real system, the optimization process would be bounded to the mechanical limits of the system. Whereby, in a simulation environment the mechanical stress could be bigger at the beginning of an optimization, while being reduced at the end by the fine tuning of the evolved behavior. Another possibility enabled by simulation is to separately optimize behaviors for single elements of the robot.

If the simulation is used to generate results for real systems, it is important to minimize the difference of the simulated robot and the real one. Optimizing this simulation-reality gap allows to transfer simulation results to real systems. To fulfill this requirements the software *Simulation Machina Arte Robotum Simulans* (MARS) is developed with special focus on performance and flexibility for configuring optimization setups. Besides the usage of MARS for this thesis, it is used in many projects at the RIC/DFKI. The applications reach from generating virtual environments for head mounted displays or multi-screen projections to launch multi-robot simulations or interactive scenarios where the virtual environment is used as a user interface to generate robot commands or create whole robotic missions.

This chapter will first introduce the simulation software itself which is a key software tool for the experiments of this thesis. The transferability of behaviors is explicitly evaluated by comparing the holistic behavior of the simulated SPACECLIMBER with the real behavior measured via a motion tracking system and the logged sensor data of the robot. Another part of this chapter is the extension of the simulation by an leg-soil interaction model. To generate this model multiple experiments performed in a soil test environment are presented. The extended leg-soil interaction is used in the final experiment chapter – evolving the controller for legged robots – to increase the quality of resulting behaviors (see Chapter 8).

3.1 Software Architecture

MARS is a modular and flexible open-source simulation and visualization framework. It mainly combines and manages three very commonly used libraries. It uses the Open

Dynamics Engine library (ODE[1]) for rigid body simulation, QT[2] for providing a graphical user interface (GUI), and OPENSCENEGRAPH (OSG[3]) for 3D visualization. The whole functionality of MARS is split into several modules (libraries). An overview of these modules and the main architecture (grouping of modules) is presented in Figure 3.1. Except for some small utility libraries, all libraries inherit from the `lib_manager` class which provides functionality to dynamically load and unload modules. Additionally, it provides detailed information about a module, containing version information of the source (address and exact version that was build) and whether the source has local modifications or not. This information can be exported when experiments are performed to allow reproducibility of the results. Every module loaded by the `lib_manager` gets a reference to it, which enables the module to load further components or get references to already loaded ones. This way an application setup can be constructed in a very modular way and its behavior is defined by the modules loaded. Beside the modular concept, all core modules of MARS are described by interfaces that are collected in the `mars_interfaces` library. These interfaces define the MARS API for plugins and other modules. Plugins are special modules that inherit from a `PluginInterface` and are managed in the MARS update loops of the graphical and physical simulation thread.

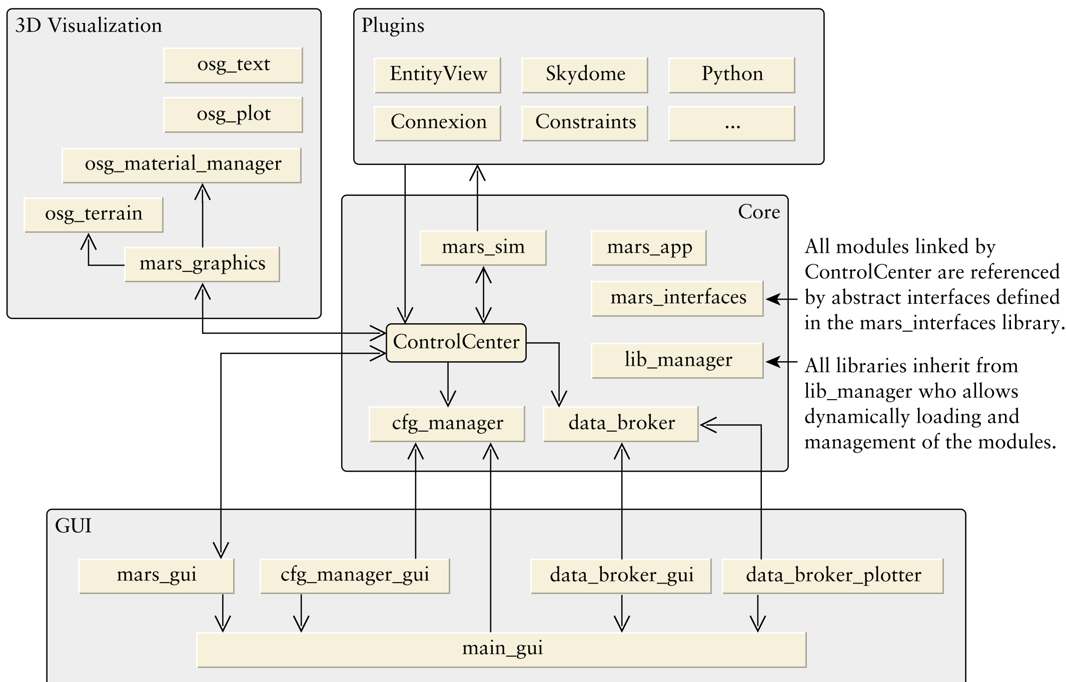


Figure 3.1: Main architecture overview of the MARS framework.

MARS Core

The core of MARS consists of the modules: `ControlCenter`, `mars_sim`, `cfg_manager`, `data_broker`, `mars_interfaces`, `lib_manager`, and `mars_app`. `ControlCenter` is the central class that is provided to any MARS plugin and the main MARS libraries. It holds interfaces to the simulation core, the graphical user interface, 3D graphics, and other

[1] Open Dynamics Engine by Smith (2005): `http://www.ode.org`
[2] QT by Qt Group (2018): `https://www.qt.io`
[3] OPENSCENEGRAPH by Osfield (2018): `http://www.openscenegraph.org`

modules. The interfaces themselves are defined in the mars_interfaces library that has no further dependencies. Thus, MARS components can be built without depending on other modules and their corresponding dependencies. Especially, the dependencies of the 3D visualization and the graphical user interface can be excluded to decrease computational requirements, e.g. for running multiple simulations in parallel.

The mars_sim module is the main simulation library which manages the models, performs the simulation updates and events, and includes the wrapping of the physics library ODE. cfg_manager is a module that generically stores and provides configuration information like preferences. It provides the functionality to store and load configurations to and from files. Additionally, the cfg_manager includes a callback mechanism that allows other modules to get notified if a configuration has changed.

The last core module is the data_broker, which is designed as a fast library providing all modules required simulation data. Every module can push to or read data form the data_broker. Also, modules can register themselves at the data_broker to periodically produce or read data. This registration can be done on trigger or timer events that are distinguished by labels. For example, one can register to the _REALTIME_ timer provided by data_broker itself or to mars_sim/simTimer provided by mars_sim representing the simulation time. The modules registered to the mars_sim/simTimer behave always correctly with respect to the simulation behavior independent of whether the simulation is calculating faster or slower than real time. Modules, that update the graphical user interface, may better synchronize with real time to avoid slowing down the physical simulation itself. Another functionality of data_broker is the possibility to connect data items. By connecting data items, the data flow of simulation modules can be defined without the need to hard wire the data flow directly within the modules.

The mars_app module implements the setup and initialization of MARS. It starts the GUI event loop and the simulation thread depending on whether a GUI is desired or not. The module includes an executable to start MARS as a stand-alone application or it can be used as a library to extend other applications with MARS, like it is done in the learning framework presented in Chapter 4.

Graphical User Interface

A base library for the graphical user interface is main_gui which creates the QT main window, manages the application menu and toolbar, and provides functionality to save and restore window positions for GUI modules that inherit from main_gui. Additionally, through inheriting main_gui, a docking and window mode is available. Most of the simulation-specific GUI elements are implemented in the mars_gui module. The modules cfg_manager_gui, data_broker_gui, and data_broker_plotter provide generic access to cfg_manager and data_broker, respectively.

3D Visualization

The 3D visualization is mainly implemented in the mars_graphics module, which setups the default scene and configurations for OSG. Amongst others, it provides interfaces for loading objects, managing materials through osg_material_manager, handling text and image overlays like camera overlays, and setting up multiple views. Since the 3D visualization is not critical for the experiments in this thesis, it is not described in detail. Figure 3.3 shows an example live rendering of the MARS visualization.

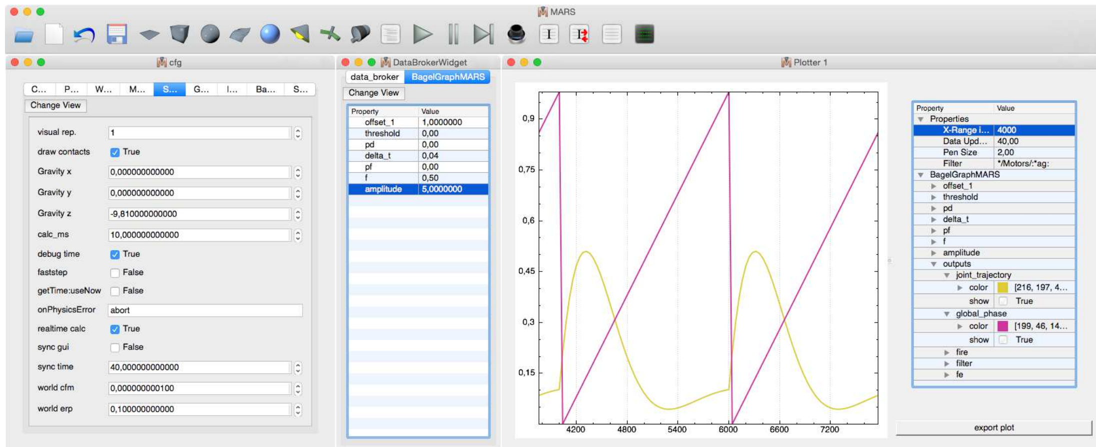


Figure 3.2: Example components of the MARS Gui. Most of the toolbar icons are generated by `mars_gui` to load or save models, start, stop or step the simulation, and quickly add objects or lights to the environment. The first of the three lower windows is the generic `cfg_manager_gui` which allows to set configuration values, like the simulation update time or whether the simulation should try to run in real time or faster. The window in the middle is the generic `data_broker_gui` where the user can check the current simulation state or write values like control parameters. The last of the three windows is the plotting tool for `data_broker`. The current content of the plotter can be exported into a folder. The raw data is exported together with the chosen configuration, like color or line width, and a Python script is exported too, that can be used to generate a plot image.

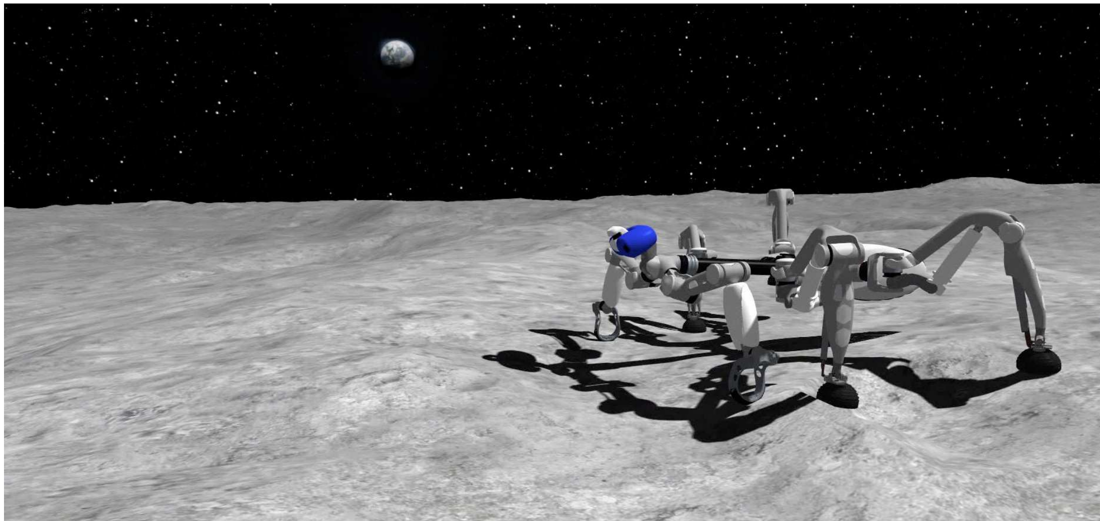


Figure 3.3: Example screenshot of the MARS 3D visualization showing the MANTIS robot (Bartsch et al. (2016)) simulated on the moon. To create a more realistic rendering the background is rendered by the `Skydome` plugin, shadows are generated, normal mapping is used on the moon surface, and a noise filter is applied on the overall scene to generate more realistic face renderings.

Plugin Architecture

A MARS plugin can extend the simulation core, the GUI, and the 3D visualization. The `EntityView` for example extends the GUI to provide user access to all entities loaded in the simulation. One can review the entities or even change parameters of the entities. `Skydome` is an example that extends the 3D visualization by a rendering of a sky. Within the initialization of the plugin it adds itself to the graphics update loop to refresh the sky rendering. The `Constraints` plugin is an example of a core extension, that adds constraints to a loaded model and allows to change these constraints through `cfg_manager`. An example constraint can be the position of the legs with respect to the corpus length of a robot model. If the corpus length is changed via the `cfg_manager` constraint property, the leg positions are updated, too. Thus morphology optimizations are possible, like it was presented for the SPACECLIMBER in Section 2.3.2.

PYTHON Interface

The simulation is extended with a PYTHON interface via the `Python` plugin. The plugin loads a given PYTHON script and expects two global functions in that script, which are called from the `Python` plugin. The first function is an `init()` function and it is called in the initialization process of the `Python` plugin or if the script is reloaded by the user. The second function is `update()`, called by the `Python` plugin from within the MARS update loop. The `update()` function of the script gets a PYTHON dictionary with requested simulation data. The MARS PYTHON interface provides functionalities such as requesting simulation data, setting motor values, changing `cfg_manager` or `data_broker` values, printing messages to the console, and rendering points, lines, and text into the 3D visualization. One can also push new data to the `data_broker` which allows to debug the scripted behavior via `data_broker_plotter`.

Configuration

Since MARS is very flexible, it needs to load a desired configuration for a target application from a defined folder. A default folder is provided by the installation, which starts a blank simulation with graphical user interface and some core plugins. Within the default simulation a robot model and/or an environment can be loaded and the simulation can be extended by dynamically loading other modules. Additionally, the PYTHON interface can be used to write and test robot behaviors directly in the running simulation. However, once the simulation is used in a bigger project scope, a specific configuration folder that sets up MARS for the project saves development time and allows non-project experts to use the software.

3.2 Open Dynamics Engine

The Open Dynamics Engine (ODE) is a rigid body dynamics simulation library. The library uses a first-order integrator which is fast and quite stable. A joint defines constraints between two bodies, for example in case of a fixed joint the connected bodies should keep the relative position and orientation to each other while calculating the next simulation state. Also, contact points are internally handled as joints, adding the constraint that the bodies should not penetrate each other. However, due to the integration and accumulation of forces arises by other constraints, an error is produced. How big this error can become depends on the simulation step size and the complexity and mass distribution of

the model. The stability of ODE can be influenced by two parameters that define how ODE handles these errors. The first parameter is the error reducing parameter (*erp*) and defines how much force a joint can add to its connected bodies to reduce the error. An *erp* value of zero means no correction forces are added and the bodies can potentially drift apart. With a value of 1 the error is reduced in a single simulation step. If the value is chosen too large the simulation can become unstable due to an oscillating behavior. A second parameter, the constraint force mixing parameter (*cfm*), defines a tolerated error. The tolerated error is proportional to the force required to fulfill the constraint. Thus a value of zero would not allow an error while a value of one would neglect the constraint. For both parameters a default value can be defined to fine-tune the trade-off of simulation stability and precision. However, due to the fact that the simulation step size and the simulated model define the "expected" errors, the default parameters have to be customized respectively. On the other hand, these two parameters can be defined for every joint individually to create "soft" constraints, like soft collision materials or spring-damper joints. The "softness" can be controlled by a traditional spring constant k_p and a damping factor k_d by calculating the corresponding *cfm* and *erp* values with the following formula:

$$erp := \frac{hk_p}{hk_p + k_d}, \tag{3.1}$$

$$cfm := \frac{1}{hk_p + k_d}, \tag{3.2}$$

where h is the simulation step size. MARS uses default parameters for the step size, *cfm*, and *erp* that are a good starting point for robotic simulations with models in an order of 2 *kg* to 90 *kg* and up to 50 joints. However, tuning these parameters for a given application is beneficial in most cases, especially for an application such as distributed evolutionary optimization. As friction model, ODE uses a Coulomb friction model with an approximation of the friction cone by a pyramid aligned with the first and second friction directions. The friction directions are calculated by ODE or can be defined externally, which is useful for simulating wheels. For each friction direction a friction coefficient ca be defined. As default, MARS takes one friction coefficient μ for both directions.

ODE allows to control a joint by either setting a joint velocity or defining the torque or force of a joint. When setting the joint velocity, a maximum torque/force can be defined limiting the joint velocity and acceleration depending on the load. In this case ODE calculates the torque/force produced by a joint, which is in general more stable than setting it directly. The default modeling in MARS uses the peak torque of a motor as maximum torque of the joint in combination with a PID controller getting a target position and producing the joint velocity.

3.3 Comparison of Simulated and Real Robot

To evaluate the simulation, the behavior of the simulated and real SPACECLIMBER robot is compared. For this comparison, three experiments are performed where the robot has to move a fixed period of time longitudinally, laterally, and rotationally. In each experiment it has to move for the same period of time in the reverse direction. The step cycle time is kept constant at 5 *s*. The time period for the movements is 60 *s* longitudinally and laterally and 55 *s* turning. Within the first and last 10 *s* of each period, the target distance to be covered in one step cycle is linearly increased and decreased, respectively. The target distances per step cycle are 20 *cm* longitudinal, 10 *cm* lateral, and 10° turning. In the first

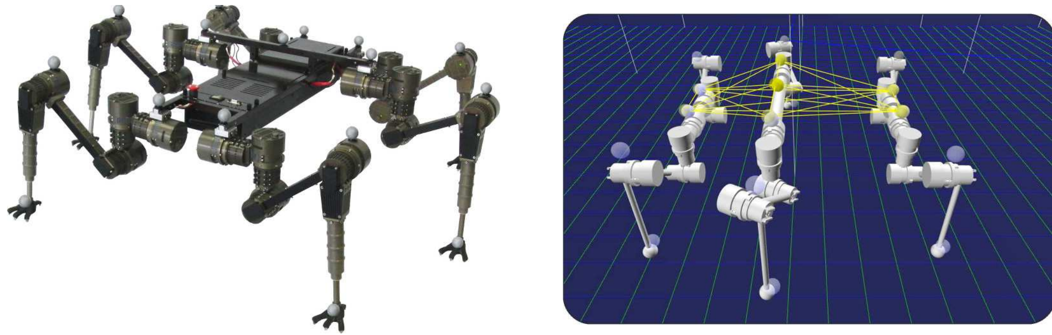


Figure 3.4: Left: The SPACECLIMBER integration study equipped with markers for the motion tracking system. Right: The simulation model with balls at the marker positions to compare the real behavior with the simulated one. The yellow wireframe represents the pose of the real system.

experiment the robot should move approximately 2 *m*, in the second 1 *m* sideways, and in the last experiment it should turn about 90°. SPACECLIMBER is equipped with reflecting markers (see Figure 3.4) which are recorded by a motion tracking system. The simulation model is equipped with markers at the same positions to allow a direct comparison. To align the real and simulation data, a starting sequence is used, in which the robot has to lift the front left leg up and down. The data alignment is done at the highest foot position of the starting sequence.

Figure 3.5 shows the resulting positions and orientations measured in real and simulation. Table 3.1 summarizes the distances traveled during the different experiment phases. The main elements that influence the behavior of the real system are the joint controller, the spring and damping properties of the lower legs, the contact behavior and the elasticity in the whole mechanical structure resulting from the materials and the assembling.

Table 3.1: Covered distances within the three experiments.

Experiment	Forward (real)	Forward (simulated)	Backward (real)	Backward (simulated)
Longitudinal (x)	185 *cm*	186 *cm*	198 *cm*	203 *cm*
Lateral (y)	97 *cm*	97 *cm*	95 *cm*	91 *cm*
Turning (*yaw*)	89°	86°	85°	85°

In simulation, the parameters of the joint controller are not related to the real ones due to much lower control frequencies. Thus, the parameters of the simulation have to be fitted to produce a similar joint behavior on the same higher level joint inputs. The behavior of two step cycles of the front left thorax joint is compared in Figure 3.6. Figure 3.7 shows a 20 *s* extract of the robot's pitch and roll angle while walking. The angles are mainly influenced by the characteristics of the lower leg suspension and weight distribution.

The ground friction coefficient and the spring and damping constants of the lower legs are manually adapted for the simulation model to produce a similar behavior of the simulated robot as the tracked behavior. Since the lower leg of the real system could get stuck, due to lateral forces, the spring and damping properties of the overall lower leg compartment differ to the properties of the installed springs. This behavior is not modeled in the simulation, therefore the spring and damping properties that approximate the real behavior best cannot directly be derived and have to be tuned manually.

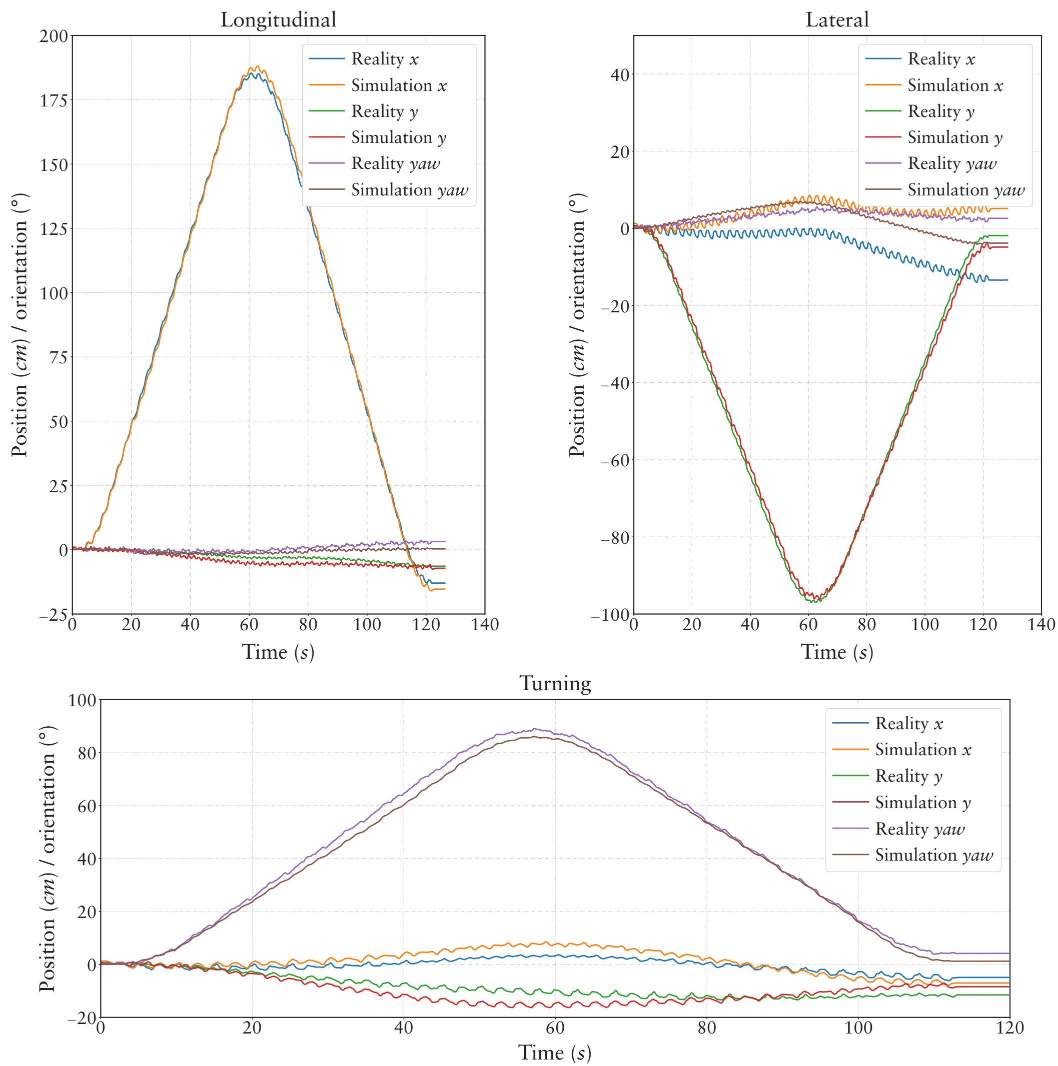


Figure 3.5: Position and orientation of the real and simulated robot over the experiment period.

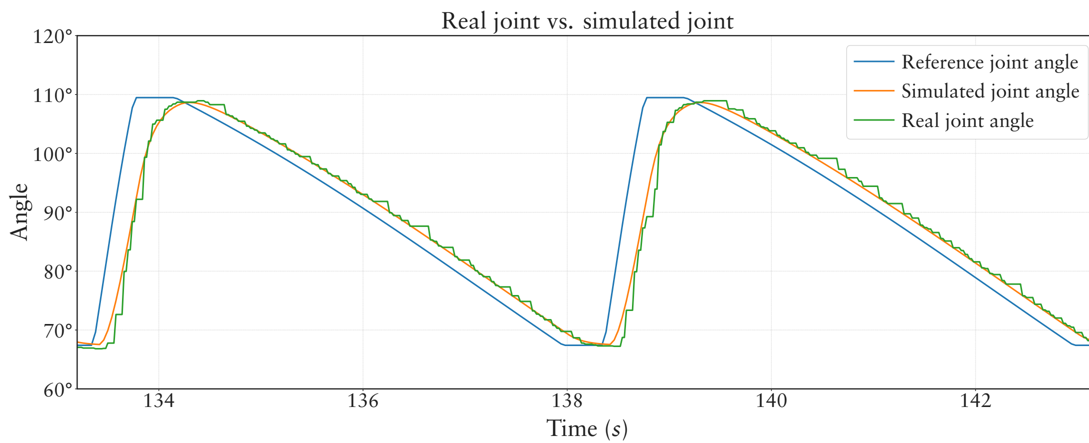


Figure 3.6: Comparison of the reference joint angle and the measured angle of the real actuator and the simulated one.

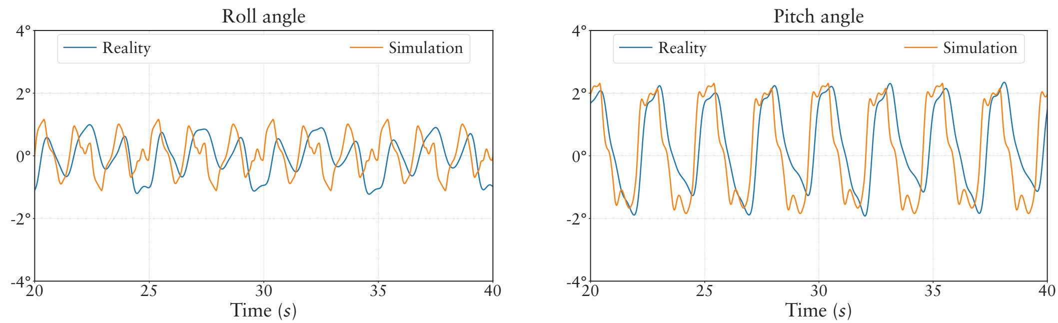


Figure 3.7: Pitch and roll orientation of the real and simulated robot over 20 *s*.

Additionally, the default *cfm* and *erp* parameters, introduced in Section 3.2, are tuned to compensate for the elasticity in the real system. Due to mechanical asymmetry, the real robot produces a drifting behavior to the right side. This effect is not analyzed in detail and it is reproduced in the simulation by shifting the main body 5 *cm* to the right. This does not represent the asymmetry correctly, and thus produces a different drifting behavior of the robot. This difference is especially observed in the roll behavior of the robot (see Figure 3.7). However, the overall behavior is closer to the real system with the asymmetric adaptation and the simulated behavior is expected to be close enough to the real behavior for transferring learning results from the simulation onto the real system.

3.4 Ground Interaction Model by Neural Network

To generate a more challenging and realistic simulation environment, the ground interaction of ODE is extended by a soft soil contact model. Therefor, a series of experiments are performed in a soil test-bed. Two "passive" experiments are performed where cylindrical objects are dropped into a test soil referred to as the "Drop Experiment" and another, the "Pull Experiment", where the very same objects are pulled through the soil. The passive experiments evaluate the general idea of adapting the contact model through a learned adaptation of the ODE contact parameters. Two additional "active" experiments are performed where a SPACECLIMBER foot is moved by an industrial robotic arm into another more granular soil ("Push Experiment") while a force/torque sensor measures the soil interaction. The second active experiment moves again the test object through the soil ("Move Experiment") to measure the lateral ground interaction. In case of the passive experiments the choice of the experiment parameters is related to the SCORPION system (Spenneberg and Kirchner (2007)) which legs and feet have a cylindrical shape as can be seen on the left hand side of Figure 3.8. The active experiments, on the other hand, are designed to represent the soil interaction of the SPACECLIMBER system with ball shaped feet, see right hand side of Figure 3.8.

3.4.1 Passive Experiments

Experiment Setup

The two parameters that define the softness of contact in the simulation are *erp* and *cfm* introduced in Section 3.2. These two parameters and the friction coefficient can be generated by a neural network to model a more realistic soil interaction. The cylinder position is tracked and the simulation is used to train a neural network to generate the same be-

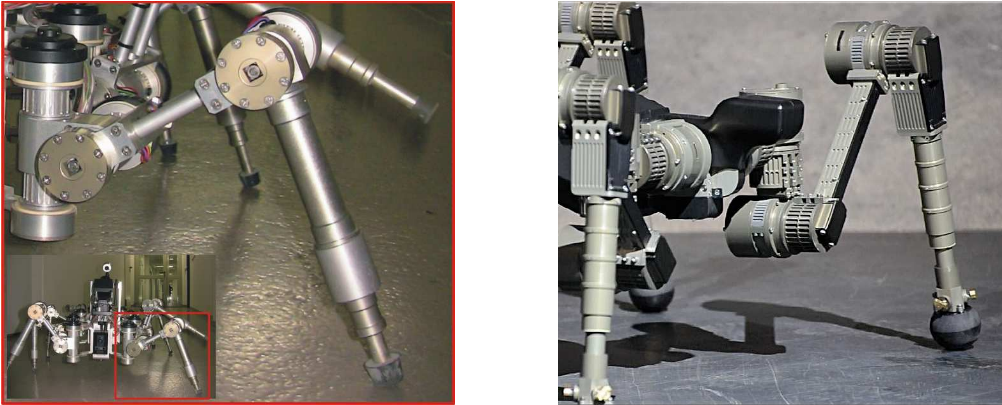


Figure 3.8: Left: The SCORPION robot with the leg zoomed in, illustrating the cylindrical shape of the leg and feet. Right: The SPACECLIMBER leg with the default ball shaped foot.
[Source of the left picture: Römmermann et al. (2009b)]

havior in simulation as measured in the real reference experiments. The input values of the neural network are diameter, velocity vector, and contact depth of the cylinder within the substrate. The structure of the neural network is fixed. Four neurons are used that receive all five inputs each. All four neurons are connected to three output neurons. The first output neuron is linked to the *erp* parameter, the second to the *cfm*, and the third output defines the friction coefficient. The neural network is a simple perceptron-based model (see Section 2.1.2) with a weight for every edge and a threshold for the outputs of the neurons. Figure 3.9 shows the structure of the neural network. To adapt the net-

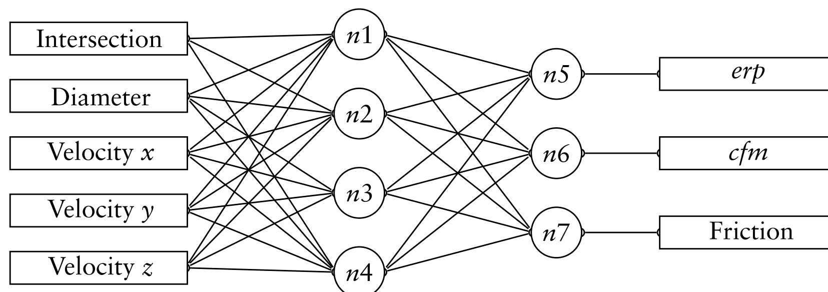


Figure 3.9: Illustration of the neural network architecture used to model the cylinder behavior of the two passive experiment setups.

work to the obtained experiment data, the weights and the thresholds of the neurons are optimized by the CMA-ES algorithm (see Section 2.2.2), resulting in 39 parameters. A simulation model consisting of a cylinder, a ground plane, and a prismatic joint is used for this optimization. In both experiment setups, the cylinder is mounted in the world by the prismatic joint. The diameter and weight of the cylinder are defined by the parameters of the real experiment setup. The ground is located on the *xy*-plane and is used to simulate the immersion of the cylinder into the soil. The contact parameters of the plane/cylinder collision are defined by the neural network. To evaluate one network, the position error of the virtual cylinder at defined time points is taken and the percentage of the error to the position of the real cylinder is summarized while the network is tested for all variations of cylinder diameter and test setups. Afterwards, the sum is divided by the number of measured distances and the number of single tests, which gives the average percentage position error that is used as a fitness value for the optimization algorithm.

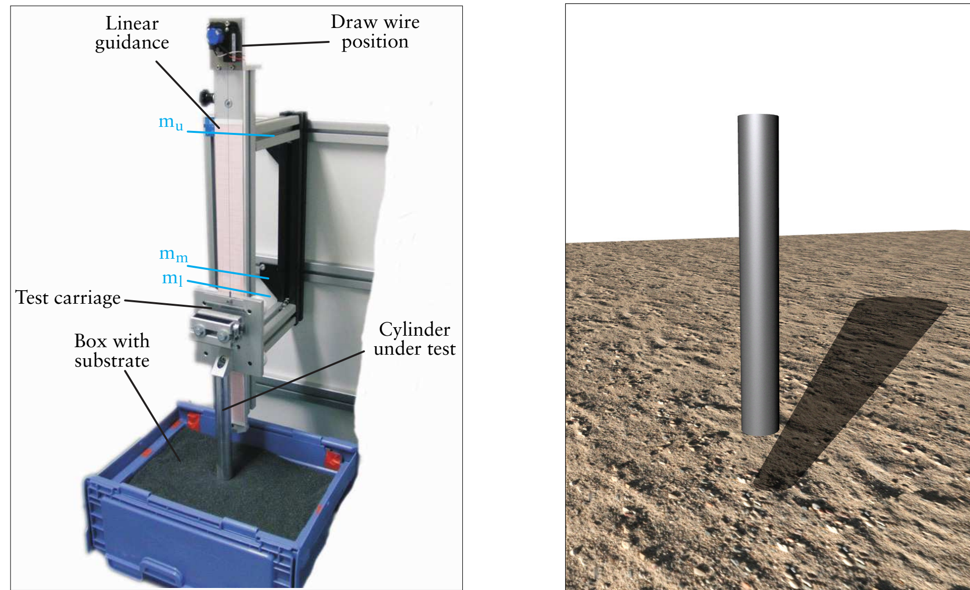


Figure 3.10: On the left hand side the drop experiment setup is shown and on the right hand side a simple rebuild of the setup in simulation by a cylinder and a plane. The cylinder is mounted to the world by a prismatic joint that is not visualized.
[Source of left picture: Römmermann et al. (2009b)]

Drop Experiment In this experiment, aluminum cylinders are dropped on a linear guidance rail into the substrate. Five different cylinder diameters in the range of 10 *mm* to 30 *mm* with 5 *mm* steps are used. The chosen diameters of the test cylinders resemble the diameters in the SCORPION leg, see Figure 3.8, which is used as a reference system for this experiment. For each cylinder an experiment series is performed including four different initial heights and ten repetitions per height. Thus, altogether $5 \cdot 4 \cdot 10 = 200$ data sets are recorded. The experimental setup is shown in Figure 3.10.

The cylinders are dropped into a box filled with basalt chippings as test substrate. The dimensions of the box are $40\ cm \times 30\ cm \times 25\ cm$. The basalt chippings is compacted by shaking and giving shocks to the box. This procedure is repeated after every fifth single experiment executions. The different weights of the cylinders are compensated with a box on the test carriage, filled with additional load to align the mass with the heaviest cylinder. A draw wire position transducer is fixed to the linear guidance rail. The wire is extended when the test carriage, with the cylinder, moves down. After calibrating the position transducer, three virtual markers are defined. The upper marker m_u is used to begin the time measurement of the experiment. A stop watch implemented on a microcontroller starts when the test carriage passes this marker. The two markers m_m and m_l are used to estimate the impact velocity of the cylinder into the substrate. In Figure 3.10 the blue lines depict the position of the markers for an experiment performed from the highest drop position. The marker m_l is adapted for each single experiment to represent the contact position of the cylinder with the substrate. Additionally, the depth of the immersion into the substrate is measured, referred to as the immersion marker m_i, and logged with the respective timestamp. The Equations 3.3 to 3.8 define the parameters that are measured with the four markers.

$$t_d = t_{m_l} - t_{m_u} = t_{m_l} \quad t_d : \text{drop time} \tag{3.3}$$
$$t_2 = t_{m_l} - t_{m_m} \quad t_2 : \text{time from } m_m \text{ to } m_l \tag{3.4}$$
$$d_d = m_u - m_l \quad \text{drop distance} \tag{3.5}$$
$$d_i = m_l - m_i \quad \text{immersion depth} \tag{3.6}$$
$$t_i = t_{m_i} - t_{m_l} \quad \text{immersion time} \tag{3.7}$$
$$v_i = \frac{m_l - m_m}{t_2} \quad \text{impact velocity} \tag{3.8}$$

In the simulation setup (see Figure 3.10) used to optimize the network parameters, the axis of the prismatic joint is identical to the global z axis. To generate a fitness value of the simulated behavior four measuring points are defined. The first one is taken at the timestamp when the cylinder reaches the deepest position in the real experiment. The second measurement is taken 100 ms after the first one. The third and fourth measurement points are defined with 100 ms offset to the second and 600 ms offset to the third point. The additional three points ensure that the cylinder is not passing the desired depth and comes to a halt there.

Pull Experiment The same cylinders are used in this experiment as in the previous one, starting with the diameter of 15 mm. Resulting in four test cylinders with the diameters 15, 20, 25, and 30 mm. The aluminum cylinders are pulled horizontally through the box on a linear guidance rail. A weight of $m = 0.5\ kg$ is connected to the test carriage via a deflection pulley. The weight generates a constant pulling force to the carriage in order to accelerate it continuously. Figure 3.11 shows the setup of this experiment.

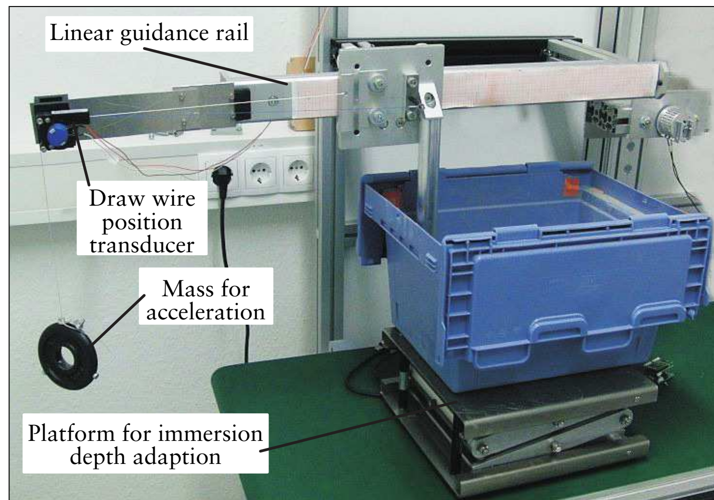


Figure 3.11: Traction Test Setup [Source: Römmermann et al. (2009b)]

To measure the experiment data, a draw wire position transducer in combination with a micro-controller is used. The covered distance and the time from the starting position to the end position is recorded. One starting position is used for all cylinders. The carriage is moved through the box until the border of the box is reached. At fixed way points, timestamps are measured to determine the velocity and acceleration of the carriage. The experiments are performed with three immersion depths per cylinder and twenty repetitions per depth, resulting in $4 \cdot 3 \cdot 20 = 240$ data sets available for the

analysis. The three immersion depths are $d_{i1} = 13$, $d_{i2} = 20$, and $d_{i3} = 25\,mm$. For the optimization in the simulation, the prismatic joint is aligned with the global x-axis and the cylinder touches the ground plane. To simulate the acceleration of the cylinder a positive constant force F is applied to the prismatic joint. Corresponding to the pulling weight of 0.5 kg in the real experiment a constant force $F = 9.81 \cdot 0.5\,Nm = 4.905\,Nm$ is used. To evaluate one neural network the *RMSE* at the defined way points is calculated.

Results

Altogether, twenty different setups are available from the drop experiment and twelve different setups are measured by the pull experiment. All repetitions performed for each setup are averaged to generate comparative data for the network evaluation. However, due to too much variance in the measured data of the pull experiment only eight setups could be used for training the neural network.

Table 3.2: The table shows the average result of 20 evolutions done for every experiment (column 1). A lower fitness value represents a better result. The data shown is the average fitness value (column 2), the standard deviation (σ) of the average fitness value (column 3), the average of the average distance error between the simulated cylinder and the real cylinder (column 4), and the standard deviation of this error (column 5).

Experiment	⊘ Fitness (best)	σ	Error (m) (best)	σ
$drop_a$	3.66e-5 (6.275e-6)	7.42 (-5)	1.55e-8 (2.656e-9)	3.14 (-8)
$drop_b$	11.181207 (8.26564)	1.986089	0.006775 (0.004771)	0.001212
$pull_a$	9.444777 (9.0202)	0.530627	0.001794 (0.001582)	0.000106
$pull_b$	13.94812 (10.6407)	3.714403	0.008933 (0.00526)	0.003151
$drop_{pull}$	16.28644 (10.692)	7.330551	0.011553 (0.006652)	0.008295

Five different network optimizations are performed in simulation. Every optimization is repeated twenty times to calculate an average performance. Two optimizations are done for the drop experiment ($drop_a$ and $drop_b$), two for the pull experiment ($pull_a$ $pull_b$) and one network is generated representing both recorded behaviors ($drop_{pull}$). The first network ($drop_a$) is generated by using only one data set, representing one cylinder diameter falling from a defined height. This network does not have to generalize and it is used to get an idea of how precise the simulation combined with the neural network can become. The second network ($drop_b$) is generated by twelve data sets. Three sets are used to test the performance of the network for input values that are not used for the training. A similar procedure is used for the optimization of the networks for the pull experiment ($pull_a$ and $pull_b$). However, because only eight data sets are available, this time all sets are used for the optimization of the network and no data set is available to test the performance for input data that is not used by the training process. For the last optimization ($drop_{pull}$) all data sets are used to train a network representing the drop and the pull behavior. Table 3.2 shows the average and the best fitness values together with the average and best position error of the cylinder in meters. Additionally, the standard deviation (σ) for both values is given.

The best resulting network of the second drop experiment ($drop_b$) is used on three test data sets to test how well the neural network generalizes for the input data. Table 3.3 shows the fitness and the resulting average distance error for the three test cylinders. The average of the three fitness values is higher than the average fitness value reached by the network training (average test fitness: 10.7319, training fitness: 8.26564). However

the average test fitness value is still lower than the average fitness value over all optimized networks (average optimization fitness: 11.181207), and thus, it is still in a reasonable range to approximate the drop behavior.

Table 3.3: Resulting fitness value (column 2) and average position error (column 3) of three test cylinders (column 1) that are not used to optimize the parameters of the neural network.

Test Cylinder	Fitness value	⌀ Position error (m)
Cylinder A	10.647	0.00609
Cylinder B	12.789	0.006387
Cylinder C	8.75972	0.008545

3.4.2 Active Experiments

Experiment Setup

Regarding the two active experiments, a `sigmoid` transfer function is used instead of a threshold and the network structure is generate by a genetic algorithm implementing a basic mutation and cross-over operation as introduced in Section 2.2.2. The genotype is implemented by a graph structure, which is evolved by the genetic algorithm. The phenotype is the network encoded by the nodes and connections. Each neuron has three inputs: the penetration depth of the foot, a bias, and a variation parameter, thus all nodes get a set of the whole network input. The variation parameter is used to model different soil behaviors measured during the collection of real data. One node of the genotype represents a neuron with the parameters: *used transfer function, input bias, and weights of the input and output signals*. A connection of the genotype adds a new output to the first referenced neuron and connects this output to a new input of the second referenced neuron. The connection itself includes the references of the two nodes (neurons) that are connected and two parameters that are used for the construction of the neural network:

1. The weight of the new output of the first node
2. The weight of the new input of the second node

Additionally, the connections define whether they connect to an existing node or if the second referenced node is duplicated by the decoding of the genotype. Thus, also an indirect decoding is possible as shown in Fig. 3.12. After the generation of the neural

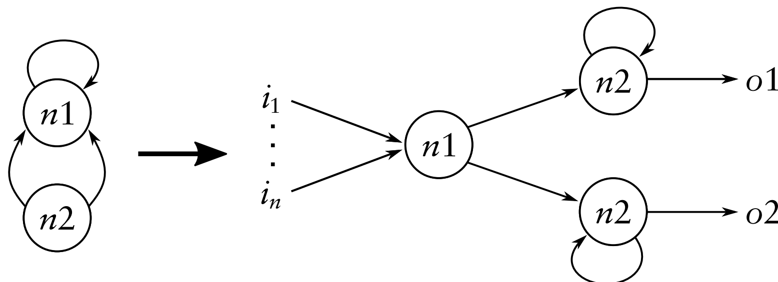


Figure 3.12: A possible indirect decoding from a genotype into a phenotype.

network, the last neuron created by the encoding gets an additional output, which represents the overall network output and is used in the simulation as the normal force of the foot-soil contact, or the lateral resistance respectively.

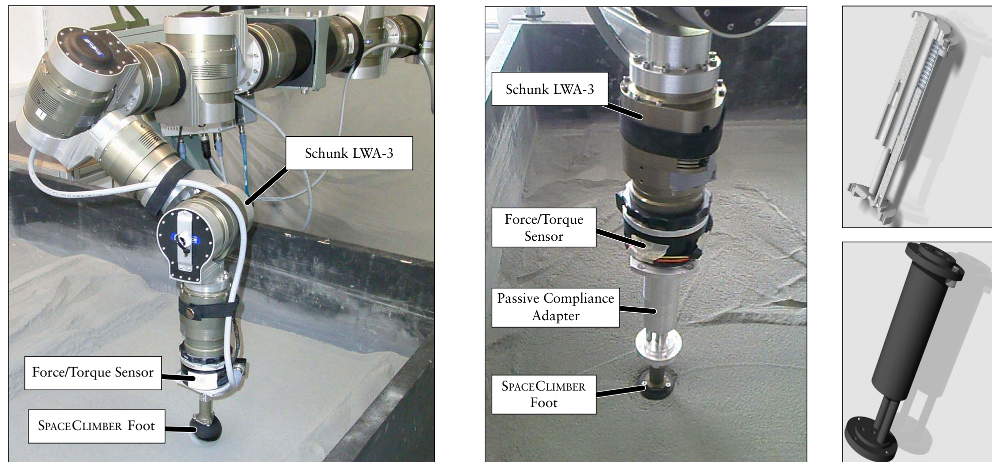


Figure 3.13: Left: The experiment setup for measuring the real foot-soil behavior while pushing into the soil. Middle: the spring mounted between the force/torque sensor and the foot. Right: The spring construction composed of three single spring-loaded cylinders.

One network is evolved for the Push Experiment generating the normal force of the contact depending on the immersion depth. Another network is generated for the Move Experiment representing the friction, by mapping the immersion depth and the normal force to a maximal lateral resistance force. A friction coefficient for the simulation can be determined by the lateral resistance force and the normal force. For the friction network a subset of the recorded data is used as training data and the root mean square error (*RMSE*) of the recorded data and the network outputs has to be minimized by the evolutionary algorithm evolving the network. Concerning the normal force, all experiment data is used for the training because of the small amount of recorded data. Again, the *RMSE* have to be minimized by the evolutionary process. Since the networks are evaluated directly with the recorded data, no simulation is needed to perform the optimization.

Push Experiment To actively push a SpaceClimber foot into the basalt chippings, the Schunk LWA-3 robotic arm is used. A six axes force/torque sensor is mounted at the end-effector of the arm, following by a cylinder with a SpaceClimber foot attached at the lower end. The experimental setup is illustrated in Figure 3.13. After positioning the foot above the substrate it is moved perpendicular to the surface into the soil at a constant speed of $1\,mm/s$. In between the individual experiment cycles, the substrate is manually loosened and leveled, as reproducible as possible with a rake for loosening and a board for leveling. Nevertheless, there is still a noticeable variance in the resulting forces due to slightly different preconditioning of the substrate. However, this different preconditioning is expected in a real environment too, when penetrating the soil in different locations. Thus, the resulting model has to contain some kind of variance.

Move Experiment To record ground interaction data for the SpaceClimber foot while moving through the substrate, the experiment setup is extended by a force controller that keeps a desired normal force simulating a constant load. To support this controller, a spring is introduced between the force/torque sensor and the SpaceClimber foot, see Figure 3.13. The experiment procedure is as follows: First the foot is placed above the substrate. Then the force controller is activated with a target load of $50\,N$. This results in a movement down into the substrate until the target load is reached. The force con-

troller has a time slot of five seconds to reach it's final state. Afterwards, with a speed of $5\,mm/s$, the foot is moved $15\,cm$ sideways through the substrate. Along the way, the force controller is enabled to maintain a constant target load. For all experiments, the joint angles, the force/torque sensor data, and the coordinates of the end-effector are recorded with a corresponding timestamp.

Results

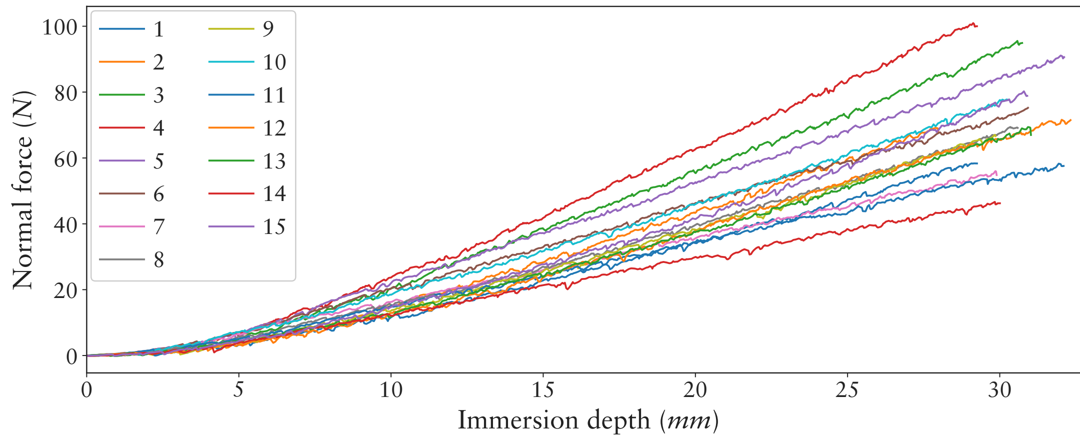


Figure 3.14: The resulting measured forces of the performed fifteen experiment repetitions.

Push Experiment The data measured from the real experiments are shown in Figure 3.14. To obtain a variance parameter for every experiment run, the forces of the curves at the maximum depth are linearly projected to a value between one and three. The minimum force value is projected to one and the maximum value to three. The value range is empirically chosen to fit to an expected default value range of a neural network. Multiple individual evolutions are performed to tune the parameters of the genetic algorithm. One network is generated with the resulting configuration representing a single possible result for this experiments. All experiment data sets are used for the network generation. The resulting force curve for the specified variance value is shown in Figure 3.15, where every experiment data set is plotted together with the corresponding network output. The generalization properties of the network are shown in Figure 3.16, where the whole network output is plotted within the defined variance value range with a discretization into one hundred steps.

Move Experiment The recorded data of five experiments are shown in Figure 3.17. Every experiment data set is split into individual measurements. Six of the data sets are selected to generate a neural network to project an immersion depth and a normal force to a friction value. Figure 3.18 depicts the measurements of all six experiments, used for training, including the immersion depth, the normal force, the friction value labeled as desired friction, and the friction value generated by one resulting neural network. The average percentage error per data point is about 3.67 %, which is a feasible generalization compared to the variance in the measured data. Figure 3.19 includes a comparison of the network output with the test data set not used for training. The average percentage error of about 3.69 % for the test data is virtually identical compared to the training data set.

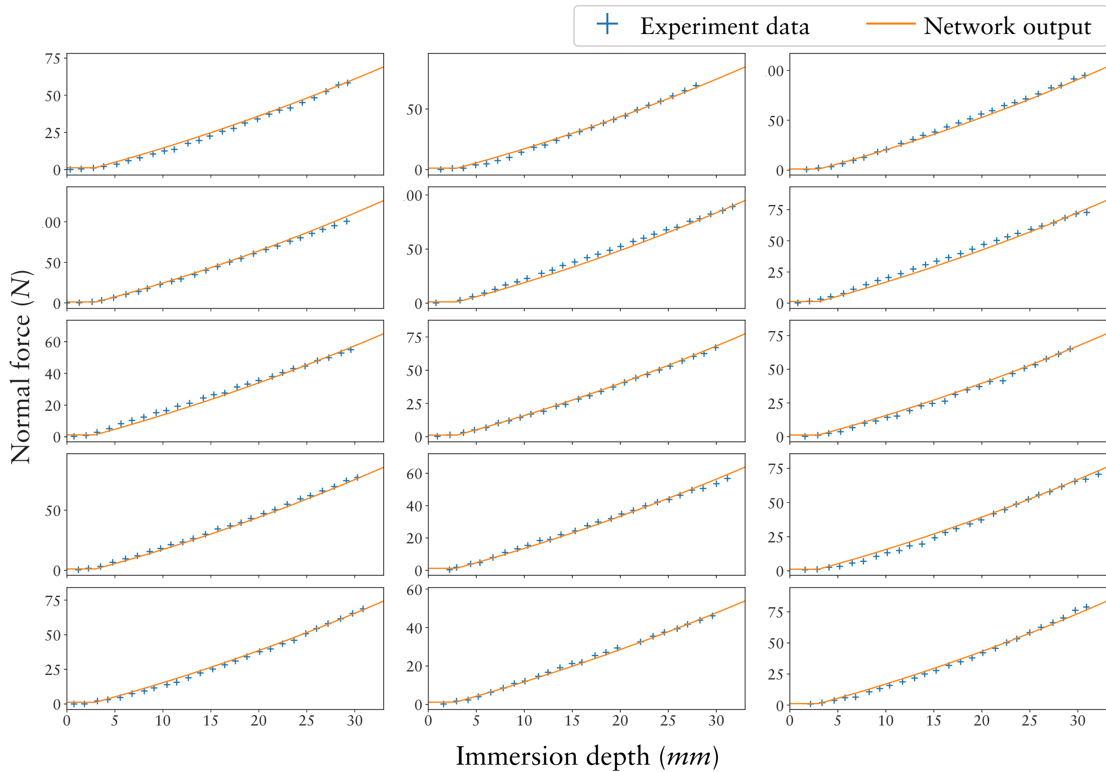


Figure 3.15: Comparison between experiment data and output of one generated neural network. The different plots are distinguished by a parameter obtained at the maximum depth of the experiment run. The parameter is used as network input and the network generalizes not only over the single runs but also over all experiments.

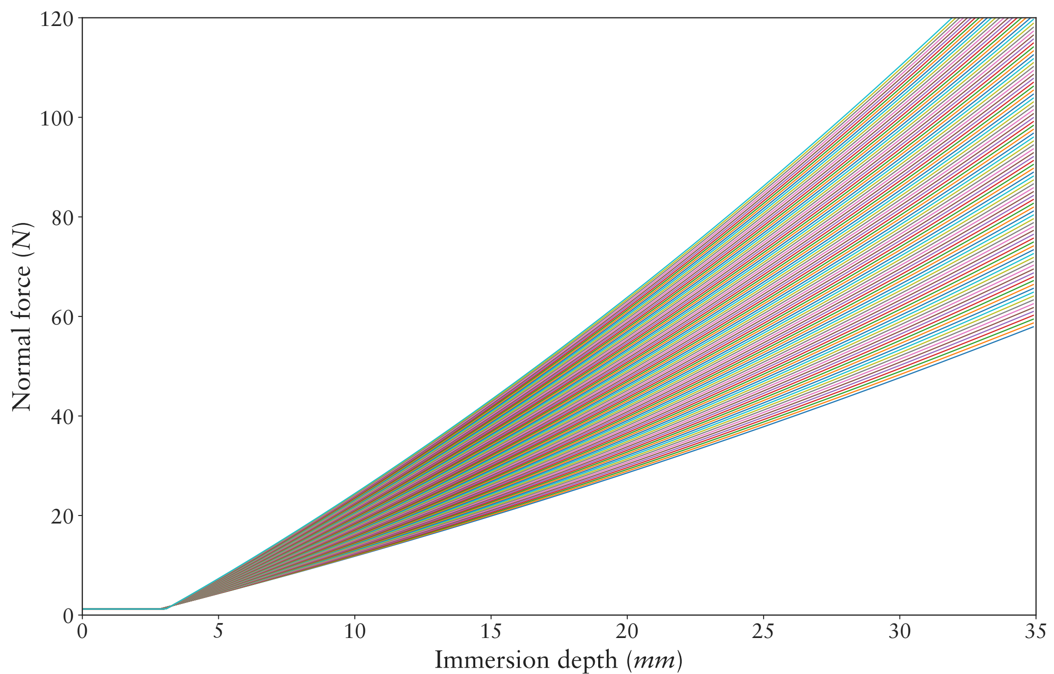


Figure 3.16: The resulting network output for variance parameters from one to three in 100 steps. Each curve depicts the output of a corresponding variance value, whereby the lowest curve correlates to a variance value of one and the highest curve to a value of three.

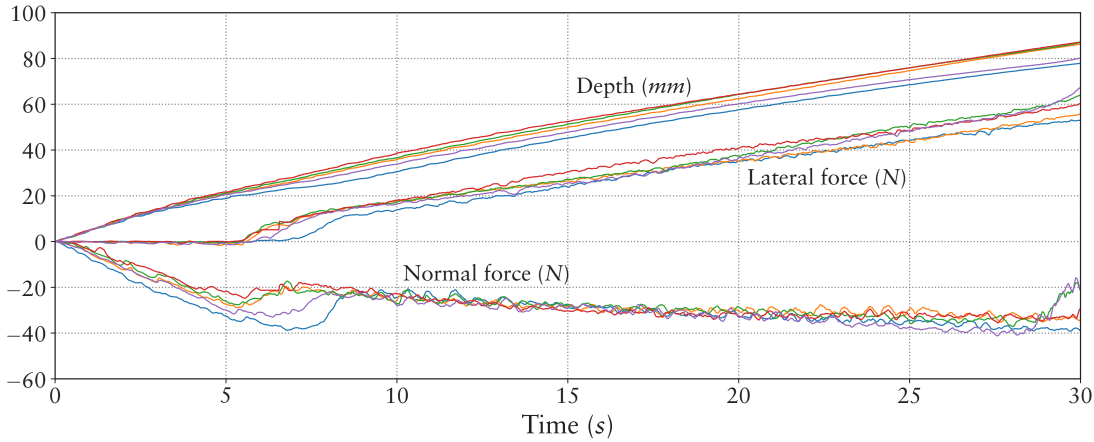


Figure 3.17: Measured data of five performed horizontal foot-soil experiments. To obtain the data for the network generation, individual measurements within the more or less smooth area roughly between 7,5 s and 27 s are taken for a training and test data set.

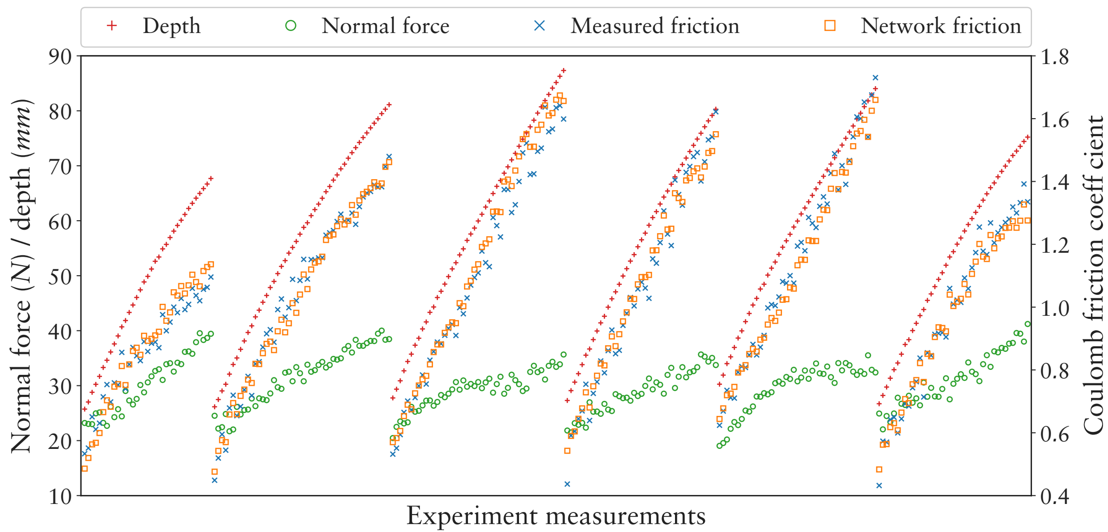


Figure 3.18: The graph compares the training data set with the resulting output of an evolved neural network. The six individual segments in the plot correlate to the data measured in six of the real experiments. Since the network cannot distinguish between the single experiment runs, it has to approximate the input-output mapping for the whole data set. The average percentage difference of the network output to the measured friction is 3.67 %.

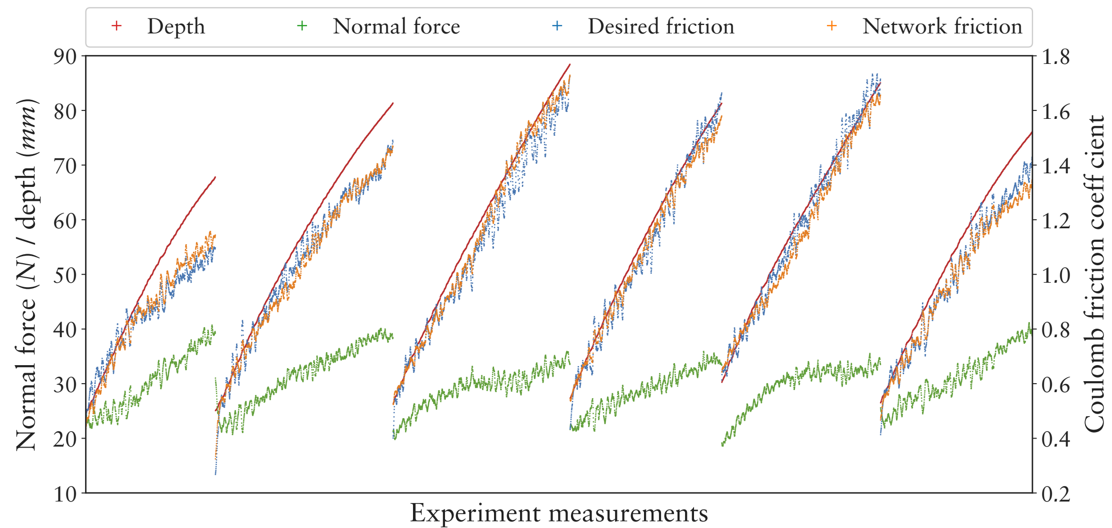


Figure 3.19: Comparison of six experiments of the test data set (not used for evolving the network). Compared to the training data, set the average percentage error of 3.69 % is virtually identical for the test data set.

3.4.3 Conclusion

The passive experiments have shown that using a neural network is a feasible concept to map soil parameters to standard contact parameters in specific situations. The positioning error of about 0.008 m is a good starting point to extend the approach. The relative high positioning error of about 8 % – 13 % is due to the fact that the percentage error includes measurements where a position error of 0.008 m corresponds to a percentage error of about 40 %. Regarding the active experiments, the neural network approach is able to approximate the measured foot-soil behavior for the contact normal force and friction. The neural network represents a generalization of the whole measured variance range for the normal force. The network approach can model the lateral forces, too. However, the measured data is not sufficient to model a comprehensive amount of use cases. For instance, no data is measured for immersion depths about 6 cm and normal forces of about 20 N, which could correlate to a situation where the load of one SpaceClimber leg is reduced within the stance phase. To compensate for this lack of data

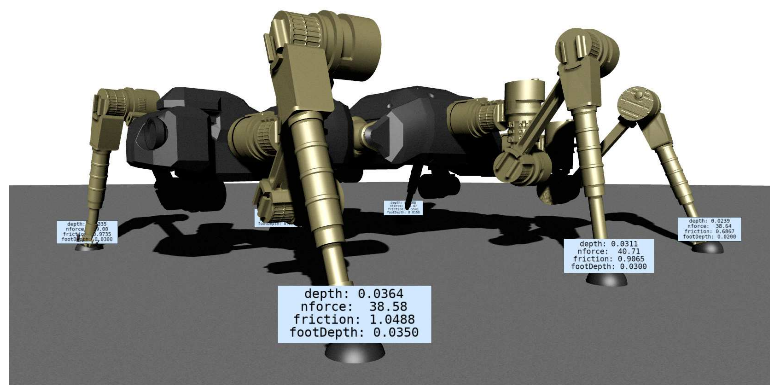


Figure 3.20: The SpaceClimber robot in the simulation with the new developed foot-soil interaction.

in the simulation application, the immersion depth is linearly projected to the measured normal force for the friction approximation, effectively mapping an immersion depth to a friction value. Additionally, by scaling the network outputs it can be adapted to represent different conditions, such as feet size or soil compaction, while the general model behavior is still based on real measured data. The calculation time of the networks is fast enough to be used in an interactive real-time simulation on a standard desktop PC. Both networks are implemented in a plugin, the `neural_soil` plugin, to extend the MARS simulation. Whenever a new soil collision is detected by the plugin, it generates a random variance parameter and saves the footprint information. If a collision is detected within a footprint, the saved properties (variance parameter and displacement) are used to calculate the contact behavior. Figure 3.20 shows the SPACECLIMBER in the simulation with random variance parameters for the footprints. In the future, this approach could be extended by using one neural network for different foot sizes, using the foot size as additional network input. The passive experiments have already shown the capability to generalize for different object sizes. The same might be possible for different soil properties, e. g. density and granularity. Therefore, the experiment setup should be improved by a more specialized actuation and a more automated experiment execution, which should allow to take more complete measurements.

Chapter 4

Learning Framework BOLERO

Slightly changing the simulation setup or parameters can significantly influence the fitness landscape of a control problem. If simulation software, used for obtaining published results is not available it is next to impossible to reproduce the results and to access the scientific relevance of them. The goal of the learning framework Behavior Optimization and Learning for Robots (BOLERO[1]) is to provide learning and benchmark problems, used in this work and other publications, within a *Open Source* framework. A special focus lies on learning and optimization of robotic controllers in simulation.

The BOLERO software defines interfaces between a problem specification and a learning algorithm. The problem specification defines the environment, including the evaluation function. The learning algorithm searches for a behavior that optimizes the evaluation function for the environment. A default learning application loads the environment and the learning algorithm from a configuration file and handles the main application loop connection both. The core of BOLERO are the generalized interfaces allowing to comfortable connect new learning algorithms and benchmark problems to BOLERO. Additionally, they allow to exchange learning algorithms for a problem and thus enable automated benchmarks.

4.1 Definition

In the following the core modules of BOLERO are introduced. Beside `Controller`, all modules are only abstract interfaces used to encapsulate specific implementations. The interfaces are `Behavior`, `Environment`, `BehaviorSearch`, and `Optimizer`. Figure. 4.1 depicts an overview of these modules and their interface definition.

Behavior

The `Behavior` interface represents the mapping of input values to output values. The number of input and output are defined by a specific application. Thus `Behavior` is the part that is generated or optimized within the learning application.

Environment

`Environment` defines the interface to a learning problem. It defines the number of in- and outputs available for `Behavior`. It provides methods to set action values, defined

[1]The core of BOLERO is open source at: `https://github.com/rock-learning/bolero` .

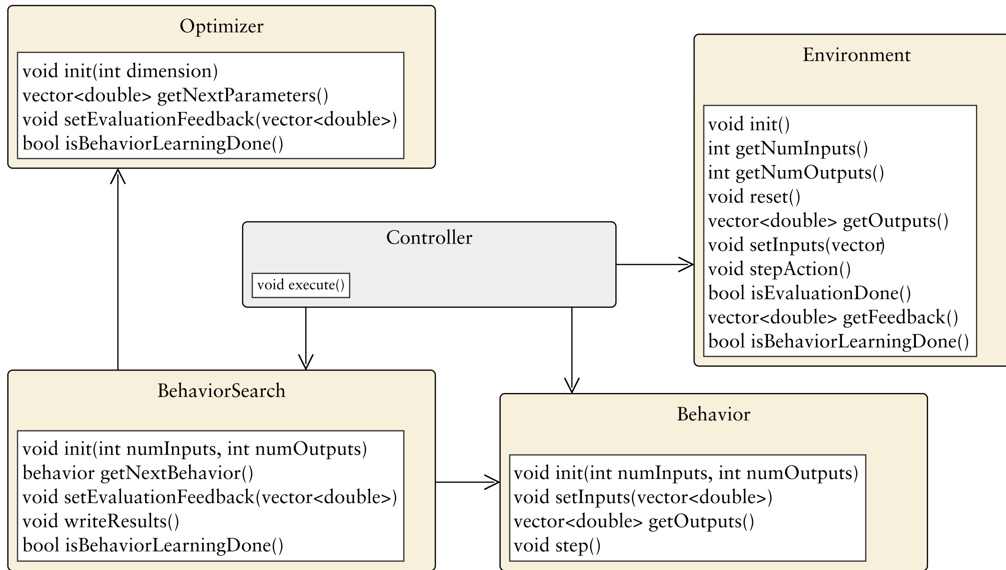


Figure 4.1: `Controller` is the main application of BOLERO that dynamically loads the environment and the learning algorithm from a configuration file at application startup. The colored boxes represent the interfaces defined by BOLERO. A `BehaviorSearch` implementation have to provide a mapping of input to output values through the `Behavior` interface. `BehaviorSearch` implementations can make use of a parameter optimizer which is decoupled and exchangeable through the `Optimizer` interface as well. `Controller` manages the interaction of `Environment` and `Behavior`.

as inputs of the environment. The output values of `Environment` can be read and piped to the inputs of `Behavior`, in general interpreted as sensor values. Other methods are defined to step and reset the environment. The reset method has to be used after the evaluation of `Behavior` is finished. Whether the evaluation is finished from perspective of `Environment` can be checked by another method. Last but not least, the `Environment` interface defines a method to return a feedback list, which allows to include evaluation values for multiple criteria. An `Environment` implementation can for instance start a robotic simulation, load a robot model and learning scenario, and interface the in- and output values to the simulated motors and sensors.

BehaviorSearch

The `BehaviorSearch` interface defines the communication with a learning algorithm. It has to provide a `Behavior` instance, gets the evaluation feedback from `Environment` and generates a new `Behavior` instance by modifying the existing. This way `Behavior` is adapted in an iterative process. `BehaviorSearch` can also manage a set of behaviors that are evaluated one after the other and a new set is generated based on the performance of the behaviors in the previous set, which correlates to the evolutionary concept.

Optimizer

Since in many applications the control structure is predefined and only control parameters have to be optimized, the `Optimizer` interface is defined. In that case `BehaviorSearch` is implemented including a fixed control structure and `BehaviorSearch` can use an optimizer for the control parameters. Parameter optimization is a research field on its own

and by using the interface different optimizers can be applied to a learning and control problem. The interface methods are similar to the one of `BehaviorSearch` just by replacing `Behavior` with a parameter set.

Controller

`Controller` is the main application and manages the learning process. It has to load the specific implementations and does the interfacing between `Environment`, `BehaviorSearch` and `Behavior`. The main control procedure is shown in Listing 3.

```
1  environment.init()
2  environment.getNumInOut(numInputs, numOutputs) # Inputs are actions; Outputs are sensors
3  bahaviorSearch.init(numOutputs, numInputs)
4  while True:
5    behavior = behaviorSearch.getNextBehavior()
6    while not environment.isEvaluationDone():
7      behavior.setInputs(environment.getOutputs())
8      behavior.update()
9      environment.setInputs(behavior.getOutputs())
10     environment.stepAction()
11   environment.reset()
```

Listing 3: Initialization and main loop of BOLeRo controller application.

Depending on the implementation, `Controller` can also include logging functionality, such as managing a folder structure, logging fitness process of learning results, and saving information about the exact modules used by logging repository information. A learning configuration file in YAML format is defined as a unified configuration source for all implementations of controllers or modules implementing the interfaces. The top level of the configuration file is a dictionary with keys correlating to the core modules of BOLeRo, as can be seen in Listing 4. The keys listed in the example are the fixed ones

```
1  Environment:
2    type: name of the environment to load
3    ... : can contain any environment specific configurations
4  BehaviorSearch:
5    type: name of the learning algorithm to load
6    ... : can contain any specific configurations of the loaded algorithm
7  Optimizer:
8    type: name of the parameter optimizer to load
9    ... : can contain any specific configurations of the loaded optimizer
10 Controller:
11   MaxEvaluations: limits the number of evaluations until the learning is finished
12   ... : controller implementation specific configurations
```

Listing 4: Top level configuration sections of BOLeRo controller.

defined by BOLeRo, while every section can contain implementation specific configurations. All configurations stored in this file can be automatically processed by the BOLeRo tools.

4.2 Implementation

The BOLeRo interfaces are implemented in C++ and Python. For all interfaces wrappers are implemented that allow to use an implementation in one language in an application of the other language. A `Controller` implementation exists in both languages. However,

through the wrapper the C++ controller can be used as the main controller, also interfacing a PYTHON environment with a PYTHON learning algorithm. A default learning controller is implemented that executes one learning application within one process in a single-threaded manner by evaluating one behavior after the other. A PYTHON framework is provided that allows to manage multiple default controllers to perform a defined number of learning executions for one application. The framework distributes the learning application on multiple processes and multiple computers. Additionally, it collects the resulting log data in a defined folder structure and provides basic analysis scripts to automatically create a summary including default statistics and plots. Additionally, two specialized controllers are implemented to allow the distribution of a single learning application on multiple processes or computers via REST communication. These controllers can be used if the learning algorithm manages a set of behaviors that can be evaluated in parallel.

bl_loader

The `bl_loader` library is a utility to dynamically load C++ or PYTHON BOLERO modules. For loading C++ modules `bl_loader` uses `lib_manager` as described in Section 3.1. The application can ask `bl_loader` for a module name defined by the type in the BOLERO configuration file and `bl_loader` will return a corresponding BOLERO interface on success. The PYTHON wrapper is part of `bl_loader` and allows to load PYTHON modules into a generic class implementing the corresponding C++ interface. If `bl_loader` cannot find a C++ library matching the desired module name it automatically searches for a matching PYTHON module. If, for instance, an environment has to be loaded, the module is found in the PYTHON path, and the PYTHON module implements the PYTHON environment interface, it can be loaded into the generic environment wrapper class implementing the C++ environment interface.

bolero_controller

The default controller implemented in C++ extends the core controller functionality, defined in the previous section, with features useful to perform benchmarks. On one hand, three different log configurations are provided to create a log of the fitness process, logging the learning result, and for debugging purpose, logging of all behaviors. Additionally, the controller allows to replay or evaluate an experiment result by loading the previously logged learning result. Another feature allows to handle a training and test phase, whereby after a given number of evaluations the best behavior is evaluated again while the environment is switched into a test mode. In case of function approximation, by using this feature, a fitness process on a training and test data set can be directly generated by performing the learning application.

Benchmark Distribution

A PYTHON based benchmark framework is implemented that is configured by a meta configuration file (benchmark configuration). Figure 4.2 depicts the framework architecture. The benchmark configuration can define multiple experiments including all information needed to generate the BOLERO configuration file. Additionally, the configuration contains information to a binary file provided by an FTP server and the number of repetitions to generate an average performance. The section used to generate the BOLERO configuration can include a list of parameters that are combined and split into individual

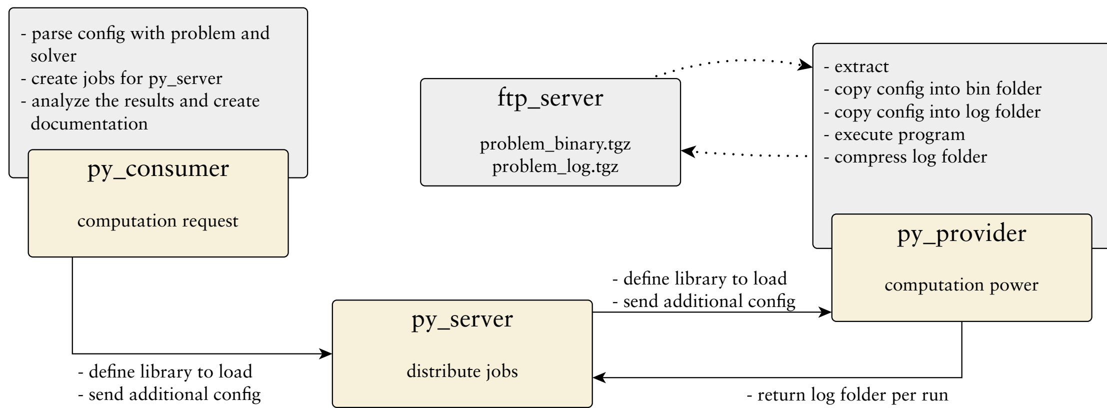


Figure 4.2: Architecture of the benchmark distribution. The `py_server` is the central part managing the distribution of jobs and results. The `py_provider` can execute a job and can be started multiple times on one machine and/or on multiple computers. A `py_consumer` loads a job definition and sends the request to the server.

experiment versions. For instance, by combining one environment with multiple learning algorithms, results are generated and collected for all combinations. Two methods can be used to combine the parameters, defined by the `parameterMix` entry. The first one (`mix`) creates all possible combinations. The second one (`aligned`) extends all parameter lists to the size of the larges one and creates all combinations element-wise. An example configuration is shown in Listing 5. A `py_consumer` application reads the bench-

```
1  Experiment Definition 1
2    archive: LIMESBenchmarkCollection.tgz
3    executablePath: behavior_learning/install/configuration/behavior_learning
4    configurationPath: .
5    executable: run.sh
6    runs: 100
7    MaxEvaluations: 1000
8    Environment:
9      - name: simple_test_problem
10       parameterMix: aligned
11       parameters:
12         CEC13TestFunction: { values: [1, 11] }
13   BehaviorSearch:
14     - name: c0n_behavior_search
15       parameterMix: aligned
16       parameters:
17         Optimizer: { values: [pso_optimizer, cmaes_optimizer] }
18         Sigma: { minValue: 0.1, maxValue: 0.5, steps: 3 }
```

Listing 5: Example configuration for distributed benchmarking with BOLERO.

mark configuration and creates the resulting jobs which are requested at the benchmark server. `py_server` pipes the jobs to free `py_provider` instances, that try to load the corresponding binary, fitting to the provider computer architecture. If a binary is loaded it is executed on the provider and the result is send to the server. The server collects the individual results and send them to the client when the results for one job are completed, i. e. the number of desired results is reached. The experiment binaries and results are communicated through a FTP server that is opened by the `py_server`. The resulting folder architecture is shown in Figure 4.3. The main concept is to split the experiment data from the configuration and the analysis tools. This can be done on different levels in the folder architecture. For instance the benchmark configuration is stored on the high-

est level, while the generated BOLERO configurations are stored in the single experiment sub-folders. All analysis tools and information should be saved at the corresponding places within the folder architecture. This allows to reconstruct how statistics and plots are generated. Again, the analysis folders can be stored in different levels of the folder architecture depending on the data they operate on. After a benchmark is finished, an au-

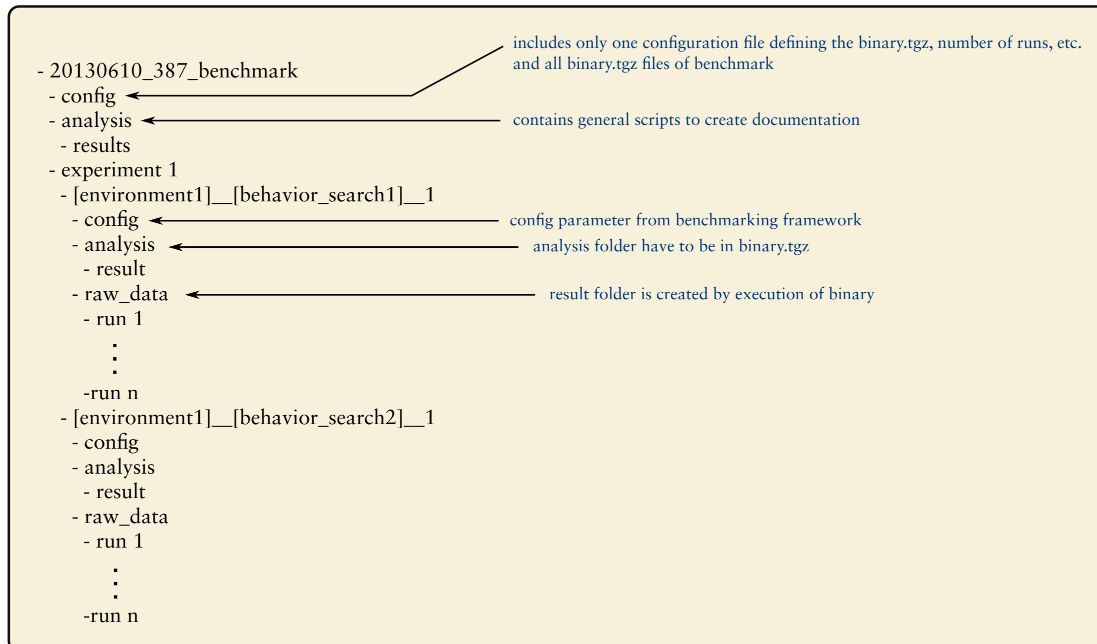


Figure 4.3: The designated folder structure for results generated with BOLERO. The benchmark framework automatically generates this folder structure and adds default scripts into the analysis folders.

tomated analysis is performed, generating statistics, significant tests, histograms, fitness plots, and a table of the results. The analysis results are summarized in a PDF document. An example is given in the appendix, see Figure A.1.

Parallel Execution

The parallel execution framework is especially designed for complex evolutionary applications with a large amount of required evaluations. The main architecture is similar to the benchmark framework with a server, an FTP server, a client, and a consumer. The server is implemented as web server with REST communication. Special client and consumer cloud controllers are implemented for the communication with the server. The consumers and clients register at the server and have to send a keep-alive message. If these messages do not arrive they are removed from the server's job management. A consumer pushes individual evaluation requests to the server which are stored server-side in the file system. The clients check whether the executable for the job is available for their hardware architecture. This is done via an FTP server as it is used for the benchmark framework. The log mechanism, the result folder structure, and the default analysis scripts are similar to the benchmark framework, too.

4.2.1 Environments

Function Approximation

A black-box function approximation environment is implemented that loads the function to approximate from an input file. The file contains a small header including the number of function in- and outputs followed by a list of function points. Which input file to load is defined in the environment section of the BOLERO configuration. Two input files can be defined, one for the training phase and another as test data set. Evaluation is done by comparing all function samples and calculating the root mean square error (*RMSE*).

MARS

A MARS environment is implemented which can be used as base for generating MARS environments for BOLERO. The library contains the main functionality to set up MARS and handle events and update procedures. For a concrete application one has to implement another library that inherits from the MARS environment and implements the loading of the simulation scenario and the evaluation function.

Locomotion

The locomotion environment (`locomotion_environment`) inherits from the MARS environment and provides functionality to load a locomotion scenario, connect sensors and motors to a behavior, and multiple possibilities to define the evaluation function. The locomotion environment includes a large number of possibilities to set up and evaluate a robotic system in MARS. These possibilities can be configured in the environment section of the BOLERO configuration. The configuration includes the maximal evaluation time period, the robot model to load, the environment to load, or a plane can be configured without the need of a special environment file to load. Predefined evaluation objectives can be configured that are the distance traveled, the energy efficiency (average motor torque over distance traveled), the load on the mechanical structure, feet slippage (f_s), average feet height, and any value that is available in the simulation can be minimized or maximized. The feet slippage is defined by the average percentage feet velocity on ground contact in relation to the robot's velocity. For every evaluation objective, a weight can be defined to tune the overall fitness function. A configuration parameter (*AbortOnContact*) can be used to abort the individual evaluation if other parts of a robot than its feet are involved in a collision. In that case the fitness is defined by a constant value (100,000,000) subtracted by the simulation time in milliseconds until the collision is detected. Additionally, the library allows to add variance to the starting position of the robot as well as apply random forces to the robot body while the evaluation is performed. The evaluation function can also be defined or combined with an external algorithm implemented in BAGEL (see Chapter 5). All locomotion experiments performed in this thesis are done using this BOLERO environment.

TORCS

The TORCS car racing simulation, as depicted in Figure 4.4, is used in a CEC competition where artificial drivers compete against each other. One approach to create an artificial driver is to use learning algorithms, e. g. the winner of 2008 Cardamone et al. (2009) used an adapted NEAT version to evolve a controller. For the TORCS benchmark a patched

Figure 4.4: TORCS car racing simulation.

version of TORCS v-1.3.4 is used. Due to the software architecture of the TORCS simulation it cannot be wrapped into a BOLERO environment directly. Instead a TORCS driver is created representing a BOLERO controller. This allows to use the TORCS environment in the benchmark framework but not with the cloud setup of BOLERO.

The predefined track used is `g-track-3`, which is not too long and contains difficult curves. The car used is `pw-evoviwrc` from the category `Offroad-4WD-GrA`. The

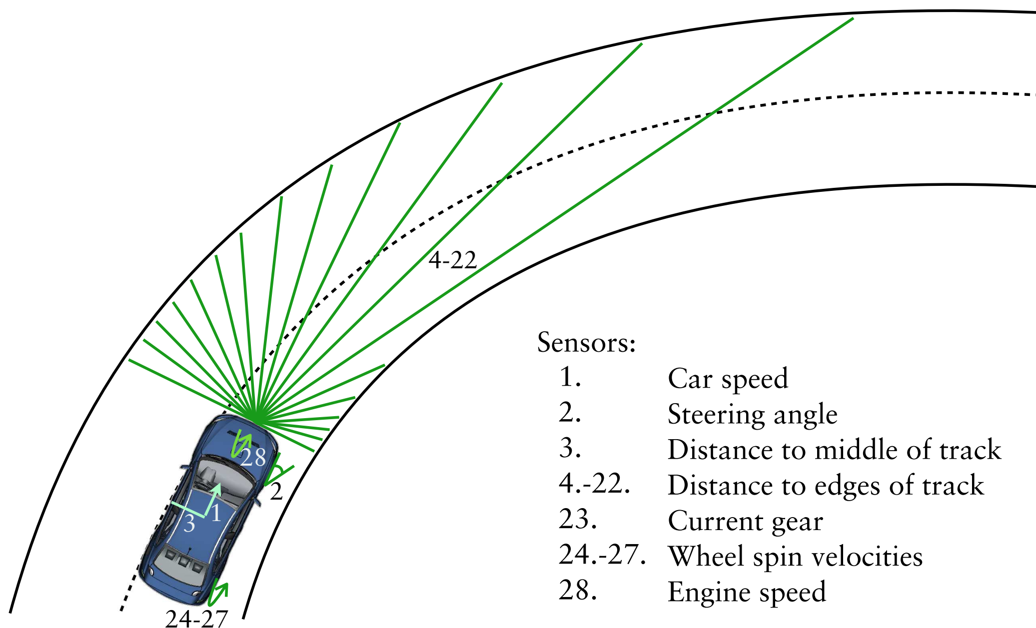


Figure 4.5: Torcs car racing setup for SABRE benchmark.

driver/behavior gets 28 sensor values and creates two output values (see Figure 4.5). The sensor values are the current speed, current steering angle, the position to the middle of the track, a scan of the track edges in a range from -90° to 90° with a resolution of 19 measurements, the current gear, the wheel spin velocities and the engine speed. The output values are the steering value and the acceleration that is used for braking if the value is smaller than zero.

A single evaluation is aborted if a lap is finished, car damage is above a threshold, the car is outside the track, the car is not moving for a given time, or the lap time exceeds 80 *s*. If the behavior does not manage a full lap the fitness is calculated by:

$$fitness = -2d + 80 - t + d_m, \tag{4.1}$$

while the fitness for a finished lap is defined by:

$$fitness = -2d - 5(80 - t) + d_m. \tag{4.2}$$

Where d is the lap distance reached, t the time the behavior was evaluated, and d_m the damage of the car. In case of the chosen setup the damage does not have a big influence since an evaluation is aborted if the car leaves the track.

CEC13 Parameter Test Functions

The test function set was used in the parameter optimization competition at the CEC13 conference. It is wrapped to the BOLERO framework as parameter optimization benchmark. Therefor, a behavior is executed only once and the output of the behavior define the parameter set used to calculate the fitness. The benchmark includes 30 functions which can be used with parameter dimensions 2, 5, 10, and up to 100 by increments of 10. The test functions mathematically define the fitness landscape. The functions and their minima are summarized in the appendix in Table A.1. A detailed description is provided in Liang et al. (2013). For the integration of the test functions into BOLERO the input values are projected from an expected range $[0, 1]$ to value ranges fitting to the individual test functions.

HyperRobot

For this benchmark a small robot with four legs and three degrees of freedom is used, similar to the work of Clune et al. (2009). The robot has a height of 25 *cm*, a width of 24 *cm* and a length of 34 *cm*. The body height is 5 *cm*, thus the length of the legs is 20 *cm* with the knee joint (the third joint) in the middle of the leg. The first joint swings the leg sideways while the second and third one swings the leg to the front and back. The

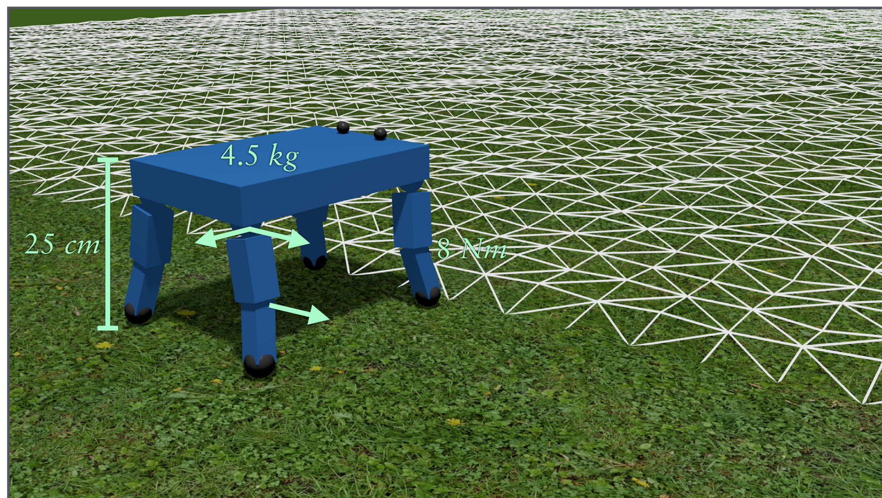


Figure 4.6: A comparatively simple robot with four legs and three *dof* per leg.

first two joints are located at the mounting of the leg at the body (the shoulder or hip respectively). The mass of the robot is 4.5 *kg* and the maximal torque of the actuators is 8 *Nm* with a maximal speed of 2 *rad/s*. The robot has two spheres (eyes) at the front of the main body without any functionality. The input to the controller to be evolved are the motor positions in radian, roll, pitch, and yaw rotation of the corpus in radian, four Boolean inputs (each input can be zero or one) indication whether a foot has contact or not, and a sine wave input given by $f(t) = sin(t * 6.28)$ where t is the simulation time in seconds. In simulation the robot can move efficiently forward by oscillating one or more legs with a high frequency. To reduce these kind of evolution artifacts, a rough terrain is added in front of the robot. The main evaluation criterion of the fitness function is the distance traveled in the forward direction of the initial robot orientation. The frequency of the left eye movement in the up axis is used as another criterion to reduce the results providing locomotion with high frequency oscillations of the legs. The last objective is the average difference of the single motor speeds to the average motor speeds. This objective is designed to evolve behaviors where almost all motors are equally used to provide forward locomotion.

4.2.2 Learning Algorithms

NEAT

The well-established NEAT algorithm (Section 2.2.2) is integrated into BOLERO. The standard NEAT implementation of the MultiNEAT package[2] is used. The BOLERO wrapper of NEAT allows to use NEAT in the default mode or in the HYPERNEAT configuration. The NEAT implementation with the default sigmoid transfer function produces only network outputs between zero and one. For every behavior output two NEAT outputs can be generated which are subtracted to generate the final value. This allows output values in the range of $[-1, 1]$. Additionally, a scaling of the output can be defined to tune the output range. Alternatively, a linear transfer function can be used for the output nodes to not limit the output values. NEAT itself provides many parameters that can be defined in a separate configuration file. Within the behavior search section of the BOLERO configuration a NEAT configuration file can be defined. A complete list of the default NEAT parameters is given in appendix Section A.1.3.

The additional parameters added by the `NeatBehaviorSearch` wrapper of BOLERO are shown in Table 4.1. The NEAT algorithm performs a strategy adaptation based on the fitness values of the population. It expects positive fitness values, which is not guaranteed by the BOLERO environment definition. Therefore, a maximal and minimal expected fitness value should be configured to scale the fitness range into the expected range of NEAT.

Optionally, the neural network can be reset after each input value process to generate behaviors that purely maps the input values to output values despite of recurrent connections allowed. To make use of recurrent connections in that configuration, the number of net iterations to process one set of input values can be defined. The last option is to enable a bias input that allows to produce a net output without any input or input values of zero.

[2] Source of the NEAT implementation used: `http://multineat.com/`

Table 4.1: Default parameters of the NEAT integration into BOLERO.

Parameter	Description	Value
MaxFitness	Max possible fitness of environment	1.0
MinFitness	Min possible fitness of environment	0.0
HyperNeat	Use HYPERNEAT approach	false
ResetNet	Reset net after each input values processing	true
NumberNetIterations	Net iterations for processing one input set	2
OutputScale	Scaling of the output values	1.0
DiffOutputs	Using two net output for one behavior output	true
PipeOutput	Use linear transfer function for output nodes	false
BiasInput	Add a constant input value of 1.0	true

Genetic Programing (GP)

A classic GP algorithm is implemented based on the BEAGLE library, see Gagné and Parizeau (2006). To enable the development of programs with multiple outputs the function set operates on a vector of real valued numbers. For every input node the input value is allocated on all dimensions of the program. While creating the functional nodes, whether a function operates on a specific dimension or not is randomly chosen. If a function does not operate on a dimension the corresponding input value is piped to the output vector. Figure 4.7 illustrates an example genetic programming tree operating on two dimensions and calculating two outputs (o_1 and o_2). The corresponding definition of the example program is given in Equation (4.3 - 4.4).

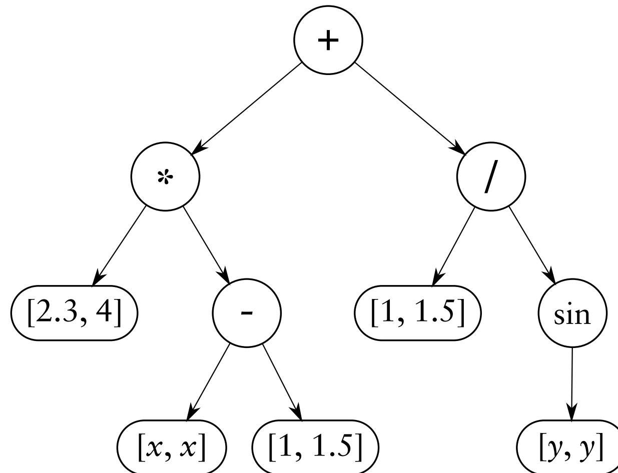


Figure 4.7: An example of a genetic programming tree encoding a program which gets two inputs (x and y) and calculates two outputs (o_1 and o_2). Each node operates on a vector of decimal numbers with the dimension identical to the number of outputs of the desired program.

$$o_1 = (2.3 \cdot (x - 1)) + \frac{1.0}{\sin(y)} \tag{4.3}$$

$$o_2 = (4 \cdot (x - 1.5)) + \frac{1.5}{\sin(y)} \tag{4.4}$$

The primitives implemented in the GP algorithm are ADD, SUB, MULTIPLY, DIVIDE, POWER, SIGMOID, and RANDOM. ADD, SUB, MULTIPLY, DIVIDE, and POWER take two inputs each and compute the respective mathematical functions. SIGMOID calculates a sigmoid function for one input and RANDOM generates a random real value with the uniform distribution $U(-1, 1)$.

SABRE

The SABRE algorithm is a genetic algorithm developed for this thesis and it is described in chapter 6.

4.2.3 Parameter Optimizer

CMA-ES

For the implementation of the CMA-ES parameter optimizer, the open source c_cmaes[3] implementation is integrated into BOLERO. The implementation does not provide any means of bounding the search space (i. e. the object variables are optimized in $\mathbb{R}$). To deal with parameter-value limits imposed by the robotic hardware or control software, a *triangle wave* function is used that projects the entire space of real values onto the range of zero to one. The triangle wave function $P : \mathbb{R} \longrightarrow [0, 1]$ is defined by the chain $x \longrightarrow x' \longrightarrow x''$ of the following transformations:

$$x' = x \,\%\, 2 \tag{4.5}$$

$$x'' = \begin{cases} x', & \text{if } x' \leq 1 \\ 2 - x', & \text{else.} \end{cases} \tag{4.6}$$

This function is continuous and it is chosen such that the absolute value of its derivative is constant wherever this derivative exists. This is done to avoid negative effects on the performance of CMA-ES that can result from non-linearly projecting the values learned onto the target-value space. Such a nonlinear projection like a modulo function could cause the fitness landscape to become more fragmented, making it more difficult to find a global maximum. Additionally, default start parameters of CMA-ES are defined that match the search space of $[0, 1]$. By scaling the resulting parameters of the optimizer to an application-specific target space, the default start parameters do not have to be adapted for the application itself.

Particle Swarm Optimization

The PSO optimizer is implemented as introduced in Section 2.2.3 with two small adaptations regarding search space and particle velocities. Both are limited to a range of $[0, 1]$, with the same projection used in the CMA-ES optimizer. This way both optimizers operate in the same parameter range and can be exchanged for an application without the need to adapt any parameters.

[3] Source of c_cmaes implementation: http://cma.gforge.inria.fr/cmaes_sourcecode_page.html

Chapter 5

Behavior Representation BAGEL

The design goal of the Biologically inspired Graph-Based Language (BAGEL) is to create a software representation that is as manageable for a human developer as for a computer program, allowing an algorithm to generate or optimize BAGEL components. Additionally, it has to provide a structure to design a modular locomotion control for kinematic complex robots. This modularity has to allow the development or optimization of single or multiple modules via ML methods, while they are evaluated by testing the whole controller.

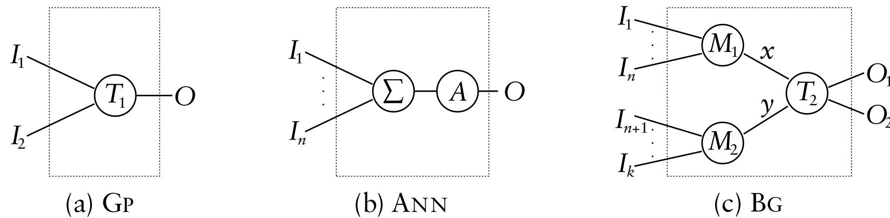


Figure 5.1: Different representations of nodes used in (a) a GP tree, (b) ANNs such as used in NEAT, and (c) BAGEL. A GP node has to define a fix number of inputs and one output to be used in the tree structure of the GP phenotype (e.g. the node T_1 gets two inputs and calculates $f(x, y) = x^y$). An ANN node uses a transfer function $\sum$ to merge an undefined (variable) number of inputs to one value that is forwarded to an activation function A. The activation function calculates the output of the node. The BAGEL node mixes both concepts by defining a transfer function T_2 which replaces the activation function of the ANN and defines a fixed number of inputs like the GP node. Additional to the GP node the transfer function can also have more than one output. For each input a merge function M_1 and M_2 is defined similar to the transfer function of the ANN node. In the depicted example, T_2 could be defined as $O_1(x, y) = x^y$ and $O_2(x, y) = y^x$.

The main concept of BAGEL is to combine the properties of artificial neural networks with behavior-based control architectures. ANNs are a suitable representation for ML algorithms but they are not that comfortable for a human developer compared with a programming language. Therefore, BAGEL extends the concept with hierarchical structuring and by providing a set of basic functions. A classical ANN node consists of an activation and a transfer function. The transfer function merges the incoming data to one value representing the neuron state. The activation function defines the output of the neuron depending on its state. Thus the activation function of an ANN node is a projection $t_n : \mathbb{R} \rightarrow \mathbb{R}$. BAGEL combines the node properties of ANNs with the node representation used in genetic programming (see Figure 5.1). The resulting transfer function used in BAGEL correlates to the activation function of ANNs and is defined as projection $t_b : \mathbb{R}^k \rightarrow \mathbb{R}^l$, where k and l are defined by the node type.

For every input of the transfer function, a port with a merge function is defined. The classic `sigmoid` transfer function has one input and one output. Thus a BAGEL node using the `sigmoid` transfer function and a `sum` merge is identical to a classic ANN node. In contrast to existing behavior representations for evolutionary methods, BAGEL offers the possibility to define more complex transfer functions that can be used as building blocks within BAGEL graphs. More specifically, a whole BAGEL graph can be used as transfer function on a higher level of a control architecture. This way, learning algorithms that can optimize BAGEL graphs can be applied on one or multiple selected graphs on different levels without application-specific implementation effort. Figure 5.2 depicts the interface of a bagel node. The `Meta Data` section in the figure is a placeholder for any kind of additional data that has no influence on the execution of the graphs. For example, the data can contain tags that allow to apply search filters to support a user while developing new graphs. Another application for the `Meta Data` is to store information for a ML algorithm directly within the graph, which could be used e. g. to prevent specific parameters from being adapted by a ML algorithm.

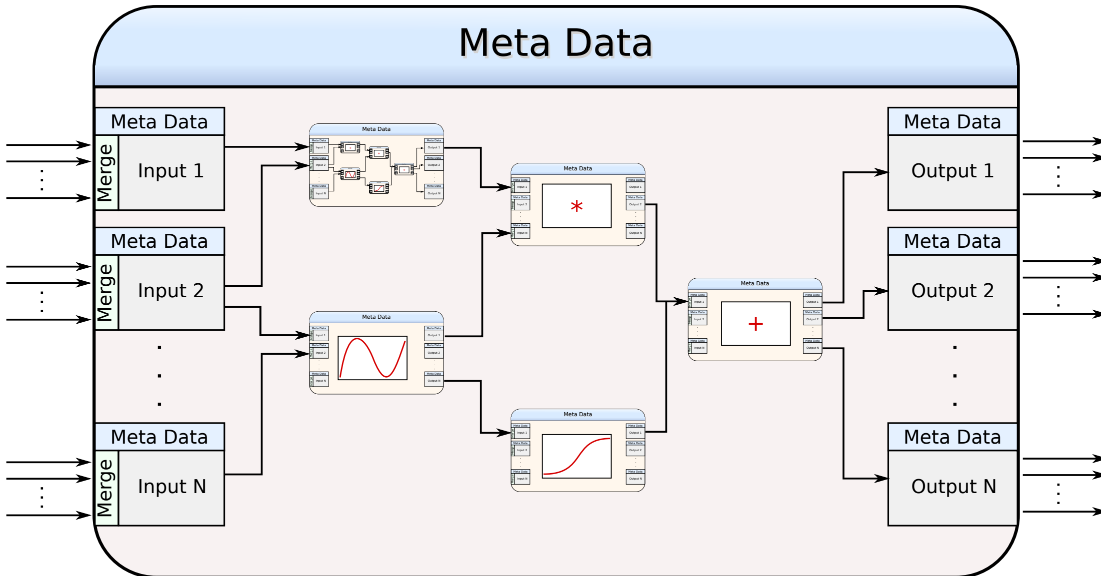


Figure 5.2: Graphical illustration of the interface and hierarchical concept of BAGEL. Beside "atomic" functions a whole BAGEL graph can be used within a node of another graph.

5.1 Definition

The BAGEL graph is a multi-graph including loops. The nodes of the graph have in- and output ports and define a transfer function from in- to output values. The edges define the data flow within the graph including a weight to scale the output of a node.
Thus the graph (G) is defined as a tuple

$$G = (V, \boldsymbol{v}', \boldsymbol{v}'', E, i, j), \tag{5.1}$$

consisting of a finite set of nodes V and edges E and two vectors of in- and output nodes (inputs: $\boldsymbol{v}'$, outputs: $\boldsymbol{v}''$) with:

$$\boldsymbol{v}' \in V^{N_i}, \boldsymbol{v}'' \in V^{N_o}, \tag{5.2}$$

where N_i is the number of input nodes and N_o the number of output nodes. One function defines the source of an edge and one the target, by a projection of the edge to a node and the index of the port to connect to:

$$i, j : E \rightarrow (V, \mathbb{N}). \tag{5.3}$$

e_i is used to reference the source tuple of edge e as replacement of the notation $i(e)$ and e_j for the target tuple. The notation $e_i^{(1)}$ is used to address the first element of the tuple, in this case the node of the source of the edge. An edge is defined as the tuple:

$$E = (x, w, r | x, w \in \mathbb{R}, r \in \{0, 1\}), \tag{5.4}$$

where x is the current value of the edge, w is the weight, and r defines whether the edge is a recurrent or a forward edge. If r is zero the edge is a forward edge, otherwise it is a recurrent one. The notations e_x, e_w, and e_r are used to reference the single elements of the edge 3-tuple. The edges connecting to one input port are merged to one input value of the transfer function. The available set of merge functions is defined by:

$$M \subseteq \{m(\boldsymbol{x}, \boldsymbol{w}, d, b) : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \,|\, \boldsymbol{x}, \boldsymbol{w} \in \mathbb{R}^N, d, b \in \mathbb{R}\}, \tag{5.5}$$

where n is a varying number for each merge function and represents the number of connected edges. Thus each merge function has to be valid for all $n \in \mathbb{N}$. The last two parameters of the merge function are a default and a bias value defined in the node. The first two input vectors of the merge function are defined as:

$$(\boldsymbol{x}, \boldsymbol{w}) = \{(x_l, w_l) = v_t(e_l) | 0 \leq l < n\}, \tag{5.6}$$

the function $v_t(e)$ defines the current tuple of value and weight depending on whether the edge is a recurrent one or not. The function is defined by:

$$v_t(e) = \begin{cases} (e_{x_t}, e_w) & \text{if } e_r = 0 \\ (e_{x_{t-1}}, e_w) & \text{else.} \end{cases} \tag{5.7}$$

The values of an edge at time t is defined by:

$$(e_{x_t}) \leftarrow eval(e_i), \tag{5.8}$$

with $eval((n, i))$ reading the current edge value from node n and output port at index i. The available set of transfer functions is defined by:

$$T_{kl} \subseteq \{t(\boldsymbol{y}) : \mathbb{R}^k \rightarrow \mathbb{R}^l | \boldsymbol{y} \in \mathbb{R}^k\}. \tag{5.9}$$

A node is defined as the tuple:

$$V = (m_1, \ldots, m_k, \boldsymbol{d}, \boldsymbol{b}, f, \boldsymbol{o} | m_1, \ldots m_k \in M, f \in T_{kl}, \boldsymbol{o} \in \mathbb{R}^l, \boldsymbol{d}, \boldsymbol{b} \in \mathbb{R}^k), \tag{5.10}$$

where $m_1, \ldots, m_k$ are the merge functions for the input ports, f is the transfer function, $\boldsymbol{o}$ is the output vector representing the output ports, and $\boldsymbol{d}$ and $\boldsymbol{b}$ are default and bias vectors for the merge functions.
Definition of node function v_n:
Let $t : \mathbb{R}^k \rightarrow \mathbb{R}^l$ and $m_1 : \mathbb{R}^{n_1} \times \mathbb{R}^{n_1} \rightarrow \mathbb{R}, \ldots, m_k : \mathbb{R}^{n_k} \times \mathbb{R}^{n_k} \rightarrow \mathbb{R}$.

Then v_n is defined by:

$$v_n(\boldsymbol{x}_1, \boldsymbol{w}_1, \ldots, \boldsymbol{x}_k, \boldsymbol{w}_k) = t(m_1(\boldsymbol{x}_1, \boldsymbol{w}_1, \boldsymbol{d}^{(1)}, \boldsymbol{b}^{(1)}), \ldots, m_k(\boldsymbol{x}_k, \boldsymbol{w}_k, \boldsymbol{d}^{(k)}, \boldsymbol{b}^{(k)})) \quad (5.11)$$

The transfer function for input and output nodes is given by $f(x) = x$. Since the input nodes are the interface to a graph, the node state (value) is defined from outside the graph and the value is transferred to the output port of the node. Thus, the whole graph function is given by $g : \mathbb{R}^{|v'|} \to \mathbb{R}^{|v''|}$ and can be used as a single transfer function with $g \in T_{|v'||v''|}$. No recursion is permitted, meaning no graph can use itself as a transfer function in one of its nodes or the nodes included throughout the hierarchy.

To define an algorithm with BAGEL, a set of merge and transfer functions has to be provided and the nodes and edges have to be defined. An example is given by:

```
1  Merge    (sum, product)
2  Transfer (input(1,1), output(1,1), pipe(1,1), pow(2,1))
3  N In1    (input,   (sum))
4  N P1     (pipe,    (product))
5  E 1      ((In1, 0), (P1, 0),   1.0)
6  N In2    (input,   (sum))
7  E 2      ((In2, 0), (P1, 0),   1.0)
8  N Pow1   (pow,     (sum, sum))
9  E 3      ((P1, 0), (Pow1, 0), 1.0)
10 E 4      ((In2, 0), (Pow1, 1), 1.0)
11 N O1     (output,  (sum))
12 E 5      ((Pow1, 0), (O1, 0), 0.5)
```

The syntax used in this example is defined by:

- *Merge* $(m1, ..., m_n)$:
  Defines a set of available merge functions M.
- *Transfer* $(t_1\ (n_1, m_1), ..., t_l\ (n_l, m_l))$:
  Defines a set of available transfer functions T and the number of in- and output ports for each function. Transfer function t_i has n_i input and m_i output ports.
- *N id* $((t, (m_1, ..., m_n))$:
  - N determines that the row decodes a node.
  - *id* is a unique identifier of the node within that graph.
  - t defines the transfer function for the node with $t \in T$.
  - $m_1, ..., m_n$ defines the merge function for every input of t with n equals the number of inputs of t.
- *E id* $((n_s, i_s), (n_t, i_t), w)$:
  - E determines that the row decodes an edge.
  - *id* is a unique identifier of the edge within that graph.
  - (n_s, i_s) defines the source node and the output port index, where n_s refers to the node id.
  - (n_t, i_t) defines the target node and the output port index.
  - w defines the weight of the edge.

The graph definition can contain multiple rows of any of the defined sections. Every referenced node, merge, or transfer function has to be defined before its first reference. For this example the definition order of the nodes correlates to the calculation order in

which the node states are updated. Thus, if the source node of an edge ($e_i^{(1)}$) is defined after the target node ($e_j^{(1)}$), the edge represents a recurrent connection. Figure 5.3 depicts the constructed example in a graph representation.

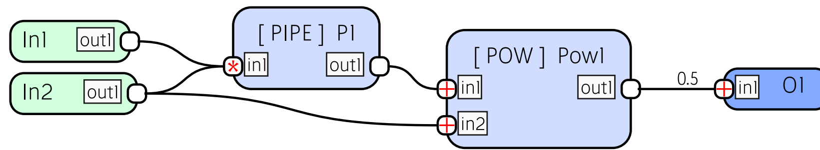


Figure 5.3: Example of BAGEL graph implementing the function: $f(In1, In2) := 0.5(In1 \cdot In2)^{In2}$.

5.2 Implementation

Software tools for the design, verification and execution of BAGEL graphs are developed. The implementation is already used on MANTIS (shown in Figure 3.3), SPACECLIMBER, and CHARLIE robot. The tools include a C execution library, PYTHON utility scripts, and a graphical user interface to view and edit BAGEL graphs. The tools are introduced in the following sections.

5.2.1 C Execution Library

The C execution of BAGEL graphs is done by the `c_bagel` library. It is restricted to the ANSI C89 standard[1], which allows an extensive compatibility to different target platforms. For instance, it can be compiled for micro-controllers or integrated in any application that allows to be extended by plugins, like Matlab or the MARS simulation. Nearly the whole API is covered by a unit test application to verify and protect the library. This is important, as this library is designed to be responsible for the reactive locomotion control of complex robotic systems. The library provides file import and export with YAML as file format. An example for the graph shown in Figure 5.3 is given in Listing 6.

The file lists the nodes and edges representing a BAGEL graph. The nodes and their ports are identified by names which are used as references in the edge list. The node type defines how many in- and outputs are expected, while the type in the input lists of the nodes defines the merge function to use. Amongst other checks, the `c_bagel` file parser returns an error if for example an identifier is not unique or a given type is unknown. Since YAML syntax is used, additional data that is ignored by the `c_bagel` parser can be stored in the files. This data correlates to `Meta Data` defined in the BAGEL interface. For instance, the tag `pos` of the nodes is used by the graphical user interface of BAGEL.

Since the files only define the graph structure but not the execution order of the nodes, the TSORT algorithm introduced in Knuth (1997) is used to determine the execution order and to identify the recurrent connections. However, in some special cases, the user still has to define the recurrent connections. Figure 5.4 depicts an examples in which the execution order cannot be clearly derived from the graph structure. In such cases the user can mark an edge to be recurrent and this edge will be ignored by the `tsort` algorithm. Due to the execution order, some recurrent edges transfer the node state of the previous iteration. It is not yet implemented that any edge can be explicitly defined as a delayed edge, which can make graphs with complex recurrent structures more complicated (again see Figure 5.4).

[1] Information on the ANSI C standards can be found at: `http://www.iso-9899.info/wiki/The_Standard`

```
1  nodes:
2    - type: INPUT
3      name: In1
4      outputs: [{name: out1}]
5      pos: {x: 897, y: 652}
6    - type: INPUT
7      name: In2
8      outputs: [{name: out1}]
9      pos: {x: 897, y: 629}
10   - name: P1
11     type: PIPE
12     inputs:
13       - {name: in1, default: 0, bias: 1, type: PRODUCT}
14     outputs: [{name: out1}]
15     pos: {x: 995, y: 654}
16   - name: Pow1
17     type: POW
18     inputs:
19       - {name: in1, default: 0, bias: 0, type: SUM}
20       - {name: in2, default: 0, type: SUM, bias: 0}
21     outputs: [{name: out1}]
22     pos: {x: 1089, y: 644}
23   - name: O1
24     type: OUTPUT
25     inputs:
26       - {name: in1, default: 0, bias: 0, type: SUM}
27     pos: {x: 1211, y: 631}
28 edges:
29   - {weight: 1, smooth: true, toNode: P1, toNodeInput: in1, fromNode: In1, fromNodeOutput: out1}
30   - {weight: 1, smooth: true, toNode: P1, toNodeInput: in1, fromNode: In2, fromNodeOutput: out1}
31   - {weight: 1, smooth: true, toNode: Pow1, toNodeInput: in1, fromNode: P1, fromNodeOutput: out1}
32   - {weight: 1, smooth: true, toNode: Pow1, toNodeInput: in2, fromNode: In2, fromNodeOutput: out1}
33   - {weight: 0.5, smooth: true, toNode: O1, toNodeInput: in1, fromNode: Pow1, fromNodeOutput: out1}
```

Listing 6: Example of BAGEL graph defined in a YAML file for `c_bagel`.

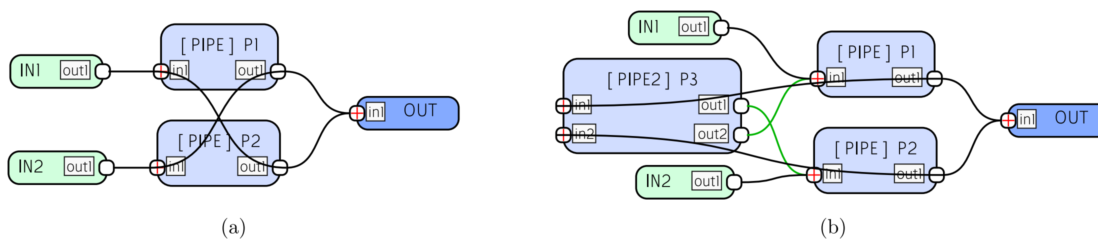


Figure 5.4: Example (a) shows a graph where the execution order cannot be derived clearly from the structure. Node P1 could be executed before P2 or vice versa. If P1 is executed first, the connection from P1 to P2 is treated as a regular forward edge while the connection from P2 to P1 becomes a delayed edge. Without additional effort, it is not possible to define both edges as delayed ones. Example (b) allows this by using an additional node P3 and marking the outgoing edges to be recurrent (green edges). This way, the edges are ignored for calculating the execution order and both P1 and P2 are executed before P3, while the outgoing edges become delayed edges.

The node types implemented in `c_bagel` are `PIPE`, `DIVIDE`, `SIN`, `ASIN`, `COS`, `TAN`, `ACOS`, `ATAN2`, `POW`, `MOD`, `ABS`, `SQRT`, `FSIGMOID`, `>0`, `==0`, and `TANH`. The available merge functions are `SUM`, `WEIGHTED_SUM`, `PRODUCT`, `MIN`, `MAX`, `MEDIAN`, `MEAN`, and `NORM`. The node types can be extended using graphs as node type. To do so, the node type has to be `SUBGRAPH` and the graph has to be provided via a file or the `c_bagel` API. Another option is to extend the node types via a plugin mechanism. In that case the node type is `EXTERN` and one has to reference the dynamic C-library providing symbols for `init_nodes` and `node_info`.

5.2.2 Extern Nodes

A PYTHON script is installed by the C execution library that creates the interface for a new external node. The input is a YAML file defining the interfaces of the new node. An example that pipes two inputs directly to the outputs is given by:

```
1  name: Pipe2
2  inputs:
3    - name: in1
4    - name: in2
5
6  outputs:
7    - name: out1
8    - name: out2
```

The corresponding interface created by the tool is:

```
1  /**
2   * \file Pipe2.h
3   *
4   * \brief
5   *   This file is auto-generated! Do not touch it, it will be overwritten
6   *   while reinstalling this library!!!
7   *
8   **/
9
10 #include <c_bagel/bagel.h>
11 #include <c_bagel/bg_impl.h>
12 #include <c_bagel/bg_node.h>
13 #include <assert.h>
14 #include <stdlib.h>
15
16 #ifdef __cplusplus
17 extern "C" {
18 #endif
19
20 /**** private structs ****/
21
22 typedef struct Pipe2_input_data {
23   bg_real in1;
24   bg_real in2;
25 }Pipe2_input_data;
26
27 typedef struct Pipe2output_data {
28   bg_real out1;
29   bg_real out2;
30 }Pipe2_output_data;
31
32 typedef struct Pipe2_private_data {
33   Pipe2_input_data inputs;
34   Pipe2_output_data outputs;
35   void *intern_data;
36 }Pipe2_private_data;
37
38 /**** private functions ****/
39
```

```
40 void install_Pipe2(void);
41 bg_error init_Pipe2(bg_node_t *node);
42 bg_error deinit_Pipe2(bg_node_t *node);
43 bg_error evaluate_Pipe2(bg_node_t *node, Pipe2_private_data *p_data);
44
45 #ifdef __cplusplus
46 }
47 #endif
```

All function names are a combination of the function and the external type name. Thus the type name have to be unique, like the node types. Beside a header file, defining this interface, a source file is automatically created managing the registration of the new node type while loading the extension with `c_bagel`. The functions declared by the interface file have to be implemented by the user. For instance, the `evaluate_Pipe2` could look like:

```
1 bg_error evaluate_Pipe2(bg_node_t *node,
2                         Pipe2_private_data *p_data) {
3   (void)node;
4   p_data->outputs.out1 = p_data->inputs.in1;
5   p_data->outputs.out2 = p_data->inputs.in2;
6   return bg_SUCCESS;
7 }
```

This possibility allows to easily extend the functionality of BAGEL with more complex algorithms; even C++ libraries can be integrated this way. However, with this possibility the protection against memory corruption cannot be guaranteed.

5.2.3 Unit Test

A unit test program for BAGEL graphs is developed. For every graph, a YAML file can be created, defining test inputs and expected outputs within a given precision. Another YAML file lists all graphs to be tested. In case of the `Pipe2` example, the definition file could look like the following block on the left hand side, while the unit test output is shown on the right hand side:

```
1 unit_tests:
2   - inputs:  [1.0, 2.0]
3     outputs: [1.0, 2.0]
4     epsilon: [0.0001]}
5   - inputs:  [3.0, 2.5]
6     outputs: [3.0, 3.0]
7     epsilon: [0.0001]}
```

⟹

```
1 testing: pipe2Test ...
2 bg_graph_from_yaml: pipe2Test.yml
3 #inputs: 2
4 #outputs: 2
5 test inputs: 1 2
6 test inputs: 3 2.5
7 expected output (index: 1): 3.0   graph output (id: 5): 2.5
8 ... fail
```

Note that the expected output defined in line 6 is wrong on purpose to generate a faulty output of the test program. By increasing `epsilon` of the last test the output is accepted and the result is:

```
1 unit_tests:
2   - inputs:  [1.0, 2.0]
3     outputs: [1.0, 2.0]
4     epsilon: [0.0001]}
5   - inputs:  [3.0, 2.5]
6     outputs: [3.0, 3.0]
7     epsilon: [ 1.0001 ]
```

⟹

```
1 testing: pipe2Test ...
2 bg_graph_from_yaml: pipe2Test.yml
3 #inputs: 2
4 #outputs: 2
5 test inputs: 1 2
6 test inputs: 3 2.5
7 ... fine
```

If no failure occurs, the return value of the program is zero, otherwise the number of failed tests is returned. Due to the return value the program can be used in an automated way for a whole collection of BAGEL graphs.

5.2.4 C++ Execution Wrapper

The predominant programming language for robotic frameworks is C++. To integrate c_bagel into these frameworks, a C++ wrapper is developed, which can be used to wrap a BAGEL graph in a class instance. The usage is shown by the following example:

```
1  #include <behavior_library/BehaviorLibrary.h>
2
3  int main(int argc, char **argv) {
4    BehaviorLibrary bLib(NULL);
5    BehaviorGraph *graph = bLib->loadBehaviorGraph("test.yml", "path/to/extern/libs");
6    std::vector<std::string> inputNames, outputNames;
7    std::vector<double> inputValues, outputValues;
8    inputNames = graph->getInputNames();
9    outputNames = graph->getOutputNames();
10   inputValues.resize(inputNames.size();
11   outputValues.resize(outputNames.size();
12   inputValues[0] = 1.0; // or any value
13   graph->setInputValues(inputValues);
14   graph->evaluate();
15   graph->getOutputValues(&outputValues);
16   // to reset the intern state if desired
17   graph->reset();
18   return 0;
19 }
```

The BehaviorLibrary class is the factory which can be used to load a BAGEL graph from a file together with a path to the correlated external nodes. This way, the external nodes can be delivered together with the graph files. The BehaviorGraph class has an interface to receive the input and output names of the loaded graph. The input and output interface is implemented by vectors of double values with the same order as the previously returned names. The execution interface is defined by setting the input values, evaluating the graph, and getting the output values. These three steps are only performed once in this example but can be integrated into an application main loop. The last interface function is the reset method that restores the initial state of the graph. This library is used to integrate BAGEL into the ROCK [2] framework and other applications.

5.2.5 Simulation Module

A MARS-Plugin is developed that allows to load, execute, and debug BAGEL graphs directly in simulation. All inputs and outputs of the loaded graph can be accessed through the data_broker component which enables the use of the MARS plotting and logging tools for debug purposes. Additionally, if the graph output names match a defined naming scheme, they are automatically connected with the motors available in the loaded simulation scene. A name matches if the motor name correlates directly with an output name or if the motor name with suffix "/des_pos" or "/des_angle" is found in the output name list. Since the input values are marked as readable in data_broker, data_broker_gui can be used to change control parameters. To connect simulation data with inputs of the graph, similar as with sensor values, again the data_broker tools can be utilized. This way, complex locomotion behaviors can already be developed and tested in the simulation without the need of an entire robotics software framework. The plugin extends the application menu with the possibility to reset the graph or reload it from file. Thus, the simulation does not have to be restarted when the BAGEL graph is updated.

[2] ROCK (the Robot Construction Kit) is used and developed for the robotic systems at the DFKI (see https://www.rock-robotics.org).

5.2.6 Graphical Design Software

Due to the fact that no existing tool could be found that fulfills the requirements concerning the graph structure, a graphical user interface (GRAPHGUI) to view and design BAGEL graphs is developed. The main architecture of the GRAPHGUI is shown in Figure 5.5. The GRAPHGUI makes use of some general libraries of the MARS framework,

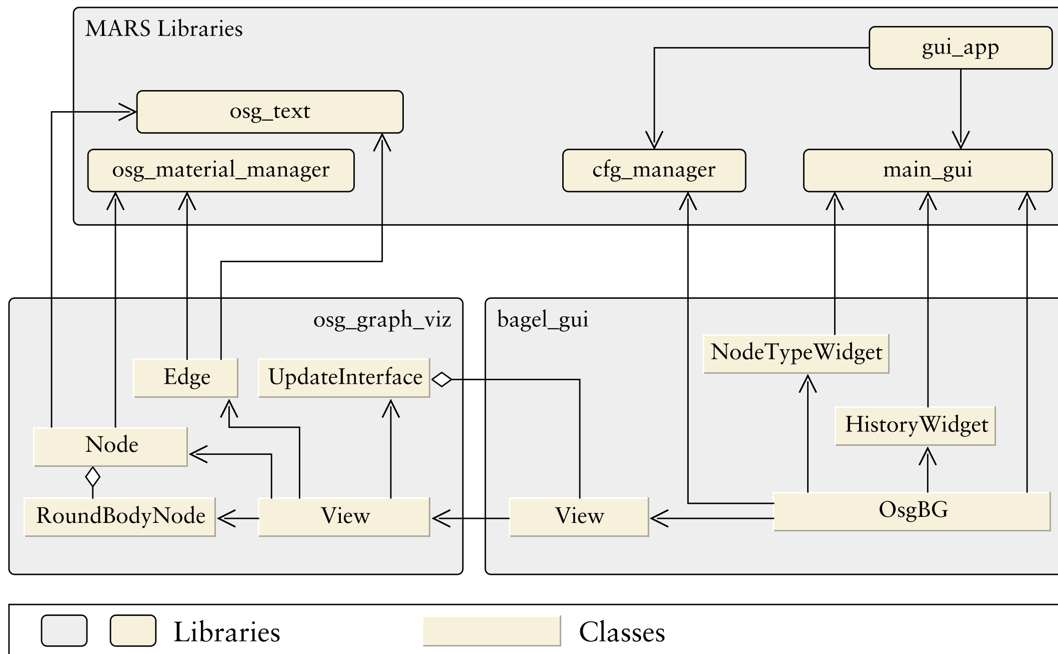


Figure 5.5: Main architecture overview of GRAPHGUI.

including the application management through the `gui_app` and the `main_gui` libraries. Also the `cfg_manager` is used to store and load user preferences. The GRAPHGUI itself is split into two libraries. One providing the graph drawing and user interaction within the graphs – `osg_graph_viz`, one used for the organization of the application – `bagel_gui`. `bagel_gui` sets up the application by loading all possible node types from a configuration folder and enables to load and save BAGEL graphs in the defined YAML format. `OsgBG` is the main class of the application, managing multiple `View` instances stored in different tabs of the main window. `NodeTypeWidget` lists all possible node types available for the graph displayed in the selected tab. The node types available for a tab can be extended by loading sub-graphs into `NodeTypeWidget`. By double-clicking the name of a node type in the list, a new node is added to the current view.

Edges can be created via drag and drop from a node output port to an input port, handled by the drawing library. The `UpdateInterface` class is used to update the `View` class of `bagel_gui`. Deleting nodes and edges is done by the drawing library and forwarded to `bagel_gui`. Every event is first processed in the main library via the `UpdateInterface` class and only performed in the graph on a positive feedback. Every `View` of the main library is coupled via `ModelInterface` with a model instance where the events are processed. Beside BAGEL other models can be implemented via `ModelInterface`, which allows to extend GRAPHGUI. Currently, GRAPHGUI is also used in the D-Rock[3] project

[3] Website of the D-Rock project: `https://robotik.dfki-bremen.de/en/research/projects/d-rock.html`

for software, hardware, electronics, and assembly models in a model-based design approach of robotic systems. Additionally, GRAPHGUI is used to develop visual shaders for the MARS simulation.

All node and edge information is stored in `ConfigMap` dictionaries provided by the `config_map` library. The library is developed as a generic data structure compatible to JSON and YAML. A generic `ConfigMap` widget is developed to view and edit the values stored in `ConfigMap`. By selecting a node or edge in the graphics view, the corresponding `ConfigMap` data is loaded into the generic data widget.

Another widget provided by the `bagel_gui` library is `HistoryWidget` which stores a history of changes for every view. The history allows to revert and redo changes of the graphs. See Figure 5.6 for an impression of the graphical user interface. The GUI includes some useful features such as duplicating a selection of edges and nodes, repositioning input and output nodes to the connected ports, or creating input nodes for selected nodes. For the latter feature, two modes are implemented. One that creates input nodes for all ports and one that creates only one input node for ports with the same name. Another feature allows to create a graph interface (inputs and outputs) by parsing the robot description used by the MARS simulation. The robot description can also reference default BAGEL graphs that are suitable for control. In that case, the graphs are loaded as individual nodes. For instance, if a control graph for a leg is referenced in the model and the robot has four legs, four nodes representing the leg control graphs are created together with edges to the correlating output nodes that represent the motors of the robot.

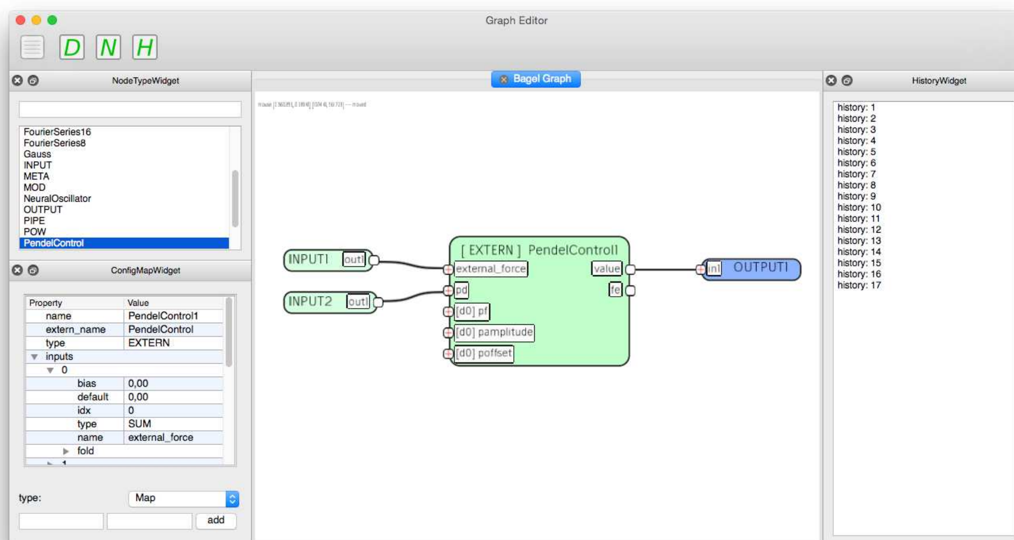


Figure 5.6: GRAPHGUI developed to design BAGEL graphs.

5.3 Conclusion

In general, the implementation is decoupled from the control logic and classical C mistakes such as wrong memory access are minimized. Additionally, as long as only "atomic" and sub-graph nodes are used, an interval arithmetic can be applied to check for possible singularities, e. g. division by zero or possible output values outside the operation range

of a target control value. In the current execution implementation, a graph representation is used internally to compute the BAGEL graphs. This is an obvious choice, but results in graph management overhead that could be reduced by translating the graphs online into individual C functions. A similar ongoing work is to translate the graphs into VHDL code to provide an execution of BAGEL on FPGA. As mentioned before, an additional extension is explicitly delayed edges, which are formalized, but not implemented yet. Beside the possibility to create BAGEL graphs with the GRAPHGUI or writing the YAML files by hand, it is possible to convert PYTHON functions into BAGEL graphs. For this, the PYTHON function has to fulfill a number of requirements, such as not using global variables. Due to the YAML interface, the BAGEL graphs can be easily serialized. Additionally, the sub-graph architecture allows to replace the graph structure within a sub-graph node. These features are for example used by the learning framework BOLERO, but one could also replace a single sub-graph on a robotic system to modify its behavior while it is operating.

Chapter 6

Genetic Algorithm SABRE

The genetic algorithm Swarm-Assisted Evolutionary Algorithm (SABRE) is designed to operate on graph structures such as those used by BAGEL. The core elements are nodes, edges, and ports, the later defining the interfaces to the nodes where the edges are attached. The interfaces of a node are separated into input ports and output ports. Edges can connect an output port with an input port of the same or another node and ports can have multiple edges connected. Thus, the genotype can fully represent artificial neural networks or common genetic programming structures. Figure 6.1 shows where the SABRE algorithm in combination with BAGEL fits into the sub-fields of evolutionary computation. SABRE indirectly adds a hierarchical structure, due to the node interface defined in BAGEL. In combination with BAGEL it allows to simultaneously develop and optimize multiple controllers on different or the same layers of the control hierarchy.

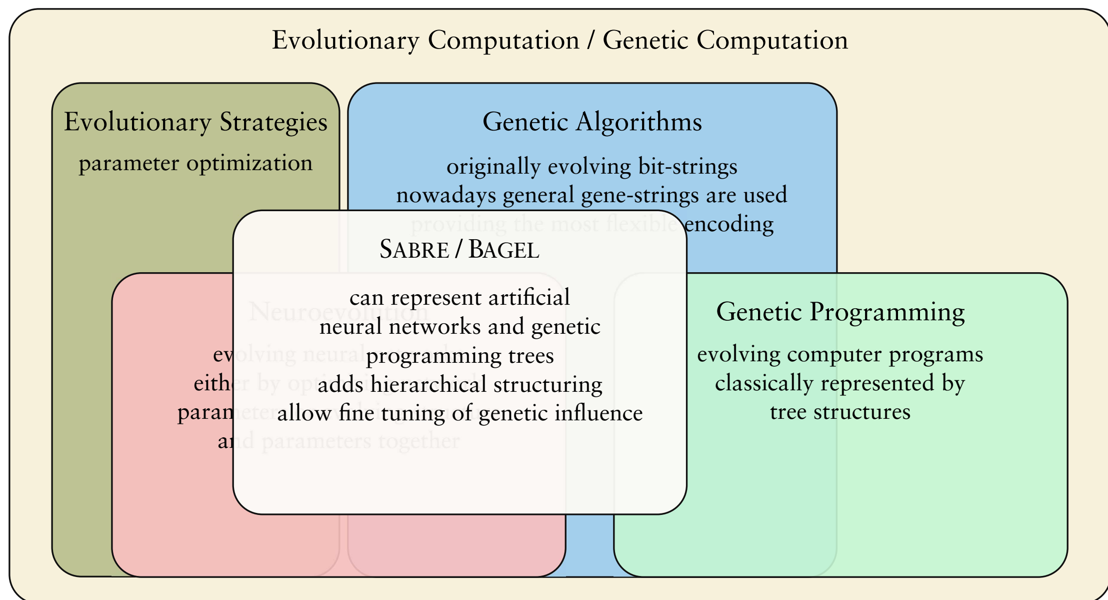


Figure 6.1: SABRE added to the overview of evolutionary computation.

Individuals, nodes, ports, and edges can have an application-specific number of parameters that are optimized in parallel to structural development. In SABRE, offspring is generated by performing mutations of a selected parent. The structure mutations are separated from the parameter mutations by a management of sub-populations or swarms. For each structure generated through SABRE a swarm of individuals is managed varying

in the parameters of the elements in the corresponding structure. For this parameter optimization a PSO algorithm or an ES method can be used. The following subsections include the definition of SABRE, the implementation including the integration into BOLERO, and some benchmark experiments to verify the method and it's implementation.

6.1 Definition

SABRE operates on gene strings containing genes that encode nodes and edges. Node genes are defined by node types that include a specific number of input and output ports. Two core types are input and output genes, including one output port or input port, respectively. Other types are defined by the specific application. The same type mechanism is used for the ports of nodes, which is used, e. g. for the different merge types of BAGEL if SABRE is used to generate BAGEL graphs. Connection genes define which nodes are linked in the phenotype through their ports. Whenever a new gene is developed, it is attached to the end of the gene string, thereby effectively sorting the genes by age. Additionally, every gene gets a unique identifier which is used to map the PSO parameters to the genes. Through the reproduction process, the genes are inherited by the offspring and can appear in multiple gene strings. An example gene string encoding a BAGEL graph is shown in Figure 6.2.

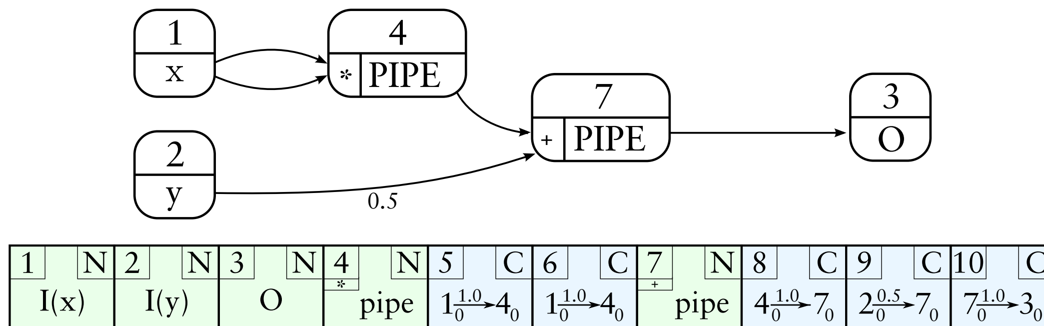


Figure 6.2: The function $f(x, y) = x^2 + 0.5y$ represented in a BAGEL graph and as gene string (underneath the graph).

Every gene contains a list of parameters that is defined by the application. A global number of parameters for nodes, ports, and edges can be defined. Node genes contain the number of parameters defined for nodes together with the number of parameters defined for input and output ports multiplied with the number of ports for the specific node type. Edge genes contain the number of parameters defined for edges.

The default behavior of SABRE is to initialize parameters of newly created edge or node genes with a uniform distribution defined by the current mutation step-size ($\mathrm{U}(-\sigma, \sigma)$). However, input ports of node types can be configured as special ports that initialize with parameters all set to zero or with a uniform distribution in $\mathrm{U}(-1, 1)$ instead of the default. Additionally, ports can be marked as hidden, which disables them for connecting edge genes.

If, for example, a BAGEL graph is developed, a BAGEL node can be used with multiple inputs where some inputs are disabled for connecting edges and thus only define parameters for the implemented transfer function instead of signals passing to the node. Optionally, SABRE produces only fully-connected individuals. For this, whenever a new gene string is created, edge genes are added that connect all unconnected ports to random ports in the graph.

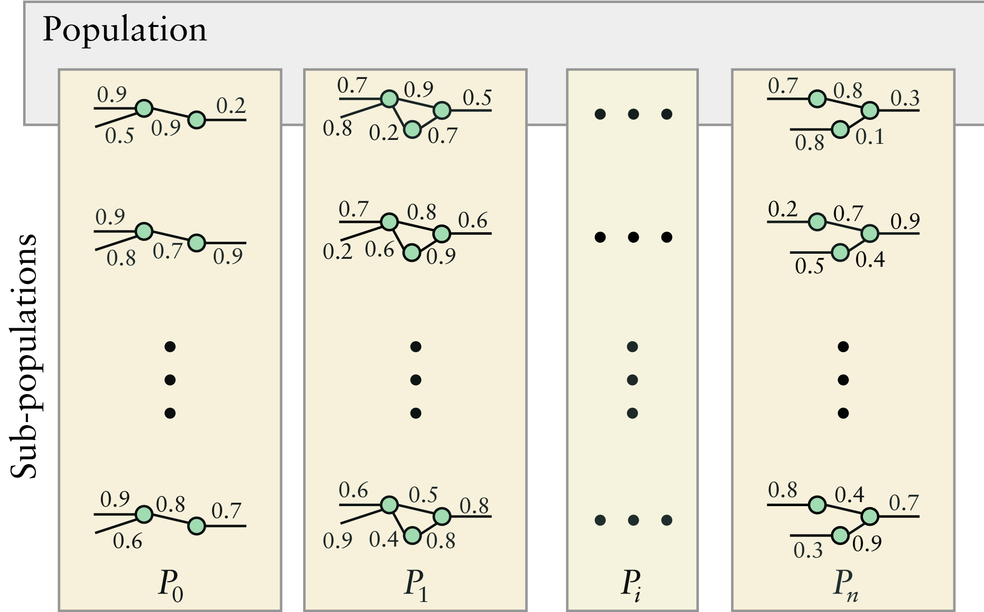


Figure 6.3: The two nested loops of SABRE. The population of different structures is depicted on the horizontal axis. The sub-populations altering the parameters of the structures are depicted on the vertical axes.

As a starting point of the evolution, a graph can be provided representing the initial parent individual. For this initial parent, a number of selected or all parameters and structure elements can be protected, preventing the genes from being changed or removed by the reproduction process. These configuration possibilities allow to precisely tune how the evolution will evolve new solutions and many undesired configurations of the resulting graphs can be eliminated beforehand instead of being filtered by the evaluation and selection procedure.

In SABRE, the structure adaptation is separated from the parameter optimization. For every gene string representing a structure, a fixed number of λ individuals are generated varying the genes' parameters. Figure 6.3 depicts this architecture. The number of *structure individuals* (κ) and λ are parameters of SABRE and define the final population size $\Lambda := \kappa\lambda$. The individuals containing the parameter variations are stored in sub-populations P_1 through P_κ, where every sub-population represents a structure individual.

The main population is given by the sum of the sub-populations $P := \bigcup_{i=0}^{\kappa} P_i$. After the evaluation of a population is finished, either the structure and parameters or only the parameters of the individuals can be mutated. The parameter e_λ defines how often parameters-only mutations occurs until also the structure mutation takes place for the next reproduction. This allows to tune the speed of the parameter development versus the structure development. A high value of e_λ for instance protects new structures and evaluates them until the defined number of parameter optimizations is performed.

The parameter mutation is done by the PSO algorithm or via an ES method. For the ES, a (1,λ)-strategy is used, meaning every generation the best individual of the sub-population is used as the parent for the new sub-population. Additionally, the parent is kept in the new population. The PSO implementation cycles through its swarm iteration to update the particle positions and define the parameters of the new sub-populations. The general evolutionary procedure is shown in Listing 7.

```
1  Input: number structure individuals κ, sub-population size λ, number parameter iterations e_λ
2  P ~ createStartPopulation(κ, λ)   # initialize start population
3  g ← 1   # initialize generation count
4  while True do
5    for x in P do
6      χ[x] ~ f(x)  # sample individual performance, where f is application specific
7    end for
8    parent ← argmin_x χ[x]   # select population parent
9    for i ← 0 to κ do
10     P_i ~ sort(P_i, χ)   # sort sub-population by fitness, best individual becomes first
11     if g % e_λ = 0 and parent not P_i[0] then
12       child ~ mutateStructure(parent)   # generate new structure individual
13       P_i[0] ← child   # assign new parent to first individual
14     else
15       child ← P_i[0]
16     end if
17     for x in P_i do
18       x ~ child not x ? mutateParameter(child) : x   # either apply ES mutation or PSO particle update
19     end for
20   end for
21   g ← g + 1   # increase generation count
22 end while
```

Listing 7: The core development process of SABRE.

6.1.1 Structural Mutation

The individual of the best performing sub-population is selected as parent for the entire new population. The parent is kept as parent individual of the first sub-population, while $\kappa - 1$ structural mutations are performed to generate the parents of the other sub-populations. Afterwards, the parameter mutation is performed to generate the new individuals of the sub-populations. To apply the structure mutation, different operations are defined. For every operation to emerge, a probability can be configured. This also allows to disable an operation by setting its probability to zero. Two modes can be used on how the operation probabilities are considered. The first mode chooses one operation by a roulette wheel selection where the probabilities define the representing area of the operation in the wheel. The second mode applies all operations with their individual probabilities for occurrence. Another option is to use both modes combined, where the mode to be used is selected for every single reproduction. In that case, the modes are selected with equal probability of fifty percent. Beside the operation probabilities the maximum number of hidden nodes can be defined to limit the size and computation effort needed. The structure is evolved based on five mutation operations (see Figure 6.4 for the first four). The operations are defined as follows:

- **`AddNode`:** The operation creates a new random node gene together with edge genes connecting the input ports of the new node with output ports of existing nodes. Similarly, edge genes are created for the output ports of the new node. If the number of nodes in the gene string reaches a predefined limit, the `RemoveNode` operation is called first.

- **`AddNodeAtEdge`:** The operation selects an edge that is replaced by a node. It creates a new node gene and an edge connecting the start port of the selected edge to a random input port of the new node and the same for the end port of the replaced edge. Since the number of node genes is increased with this operation, the check whether the maximum number of hidden nodes is reached is done beforehand, like for the `AddNode` operation.

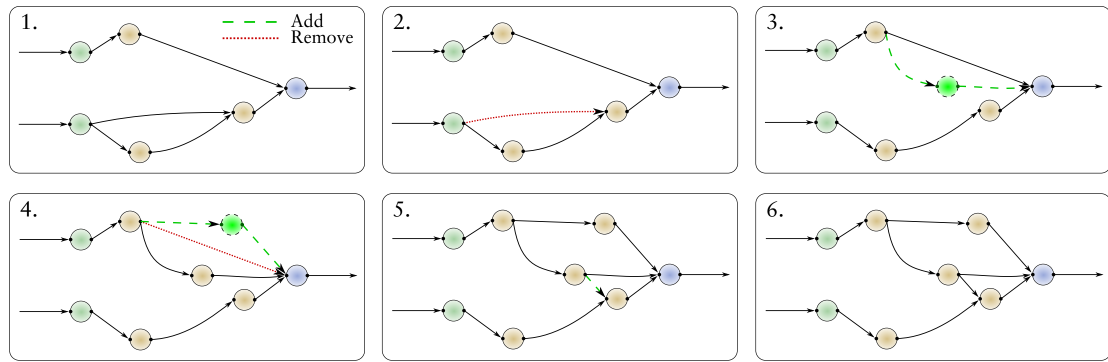


Figure 6.4: Example of structure development. For simplification the port handling of the nodes is not illustrated. Starting with the first graph the four structure mutations `RemoveEdge` (2.), `AddNode` (3.), `AddNodeAtEdge` (4.), and `AddEdge` (5.) are applied. 6. depicts the resulting structure.

- **`RemoveNode`:** This operation removes a random node gene and all connected edge genes.
- **`AddEdge`:** This operation connects a random output port with a random input port by a new edge gene.
- **`RemoveEdge`:** This operation removes an edge gene from the string.

6.1.2 Parameter Mutation

The number of individual, node, port, and edge parameters are globally set for an evolution setup. A maximal and minimal mutation step size σ_{max} and σ_{min} can be defined to control how fast the individuals can adapt to a given environment. Optionally the maximal mutation step size of the last sub-population can be set to $3\sigma_{max}$. This option is referred to as *EvoJumping* and configures one sub-population to search more explorative.

Particle Swarm Optimization (PSO)

For the PSO method, every particle of the swarm is given one segment per unique gene it maps to. The segments encode as many parameters as required by the particular gene. Thus, the parameter dimensions of all particles is equal and every particle includes parameters for all genes of the whole population. The mapping of PSO parameters to gene parameters is done by the unique identifier of the genes (see Figure 6.5). If a gene is completely removed from the population, the dimension of all particles is reduced accordingly by removing the corresponding parameter segment from all particles. For the mapping of the parameters, segments that are linked to genes not represented in the target gene string can be ignored. The particle positions are projected to a defined target space by a triangle wave function (as introduced in Section 4.2.3). If a particle's velocity is below a defined minimum (σ_{min}), the velocity of the particle is randomly reset. The maximal particle speed is limited by the maximum mutation step size σ_{max}.

Evolutionary Strategies (ES)

In the ES parameter mutation the gene parameters are changed either by a uniform distribution $U(-\sigma, \sigma)$, with $\sigma_{min} < \sigma < \sigma_{max}$ or by the following equation referred as *GaussCircle* adaption:

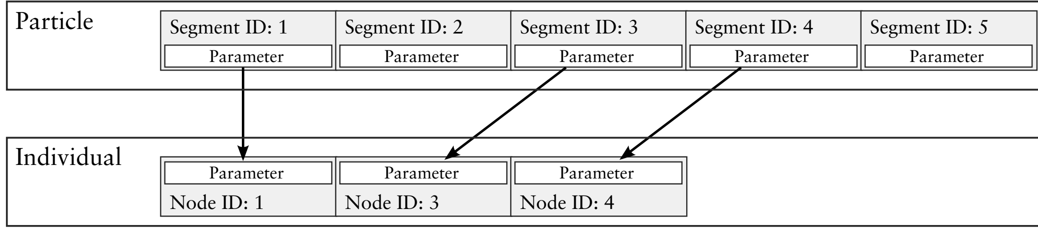


Figure 6.5: The figure depicts the mapping of a PSO particle to a gene string of SABRE. The particle parameters are separated into segments whereby every segment stores the parameters of one specific gene. The particles include segments for all available genes in the population. To map the parameters of a particle to a gene string, the corresponding segments of the genes appearing in the string are read.

$$\dot{x} = \begin{cases} \mathrm{N}(\sigma, \sigma) & \text{with probability } 0.5 \\ -\mathrm{N}(\sigma, \sigma) & \text{else.} \end{cases} \tag{6.1}$$

The step size is adapted every generation by the fitness development of the populations. If the fitness is improved from one population to the next, σ is increased by:

$$\sigma = \begin{cases} \frac{\sigma}{\beta} & \text{if } \frac{\sigma}{\beta} \leq \sigma_{max} \\ \sigma_{max} & \text{else,} \end{cases} \tag{6.2}$$

where β controls how fast the step size is adapted with $0 < \beta \leq 1$. Similarly, if the fitness does not improve from one generation to the other the adaption is done vice versa and σ is decreased by:

$$\sigma = \begin{cases} \sigma\beta & \text{if } \sigma\beta \geq \sigma_{min} \\ \sigma_{min} & \text{else.} \end{cases} \tag{6.3}$$

The parameter mutation is either performed for only one random gene (*single gene mutation*), the parameters of the genes are mutated with a probability of 0.1, or both modes can be used combined with a selection probability of 0.5 each. The single gene mutation can be used to produce a slow step-by-step adaptation, which is generally a feasible approach. However, in more complex scenarios, a fitness improvement may only be possible by adapting multiple genes at once.

Each gene string keeps its last parameter mutation which is applied again in case of a fitness improvement. This especially increases the adaptation process when a slow mutation step size is used. In contrast to PSO, the parameter range is not limited to a target space.

6.1.3 Default Configuration

The default parameters are listed in Table 6.1. The parameters are empirically chosen to fit as good as possible to a set of multiple benchmark scenarios. In the following sections and chapters only the parameters deviating from the default values are mentioned in the corresponding experiment descriptions.

Table 6.1: Default parameters of SABRE.

Parameter	Value	Parameter	Value
NumSubPopulations (κ)	5	MutateRemoveNodeProb	0.5
SubPopulationSize (λ)	10	MutateAddEdgeProb	0.5
NumSubIterations (e_λ)	40	MutateRemoveEdgeProb	0.5
NumStartHiddenNodes	4	UsePSO	disabled
NumMaxHiddenNodes	80	EvoJumping	enabled
NumMaxConnections	160	MutateSingleParameter	both modes
WorkFullyConnected	enabled	MutateGaussCircle	enabled
MutateStructureMode	both modes	StartSigma (σ_{max})	0.15
MutateAddNodeProb	0.5	ResetSigma (σ_{min})	0.000005
MutateAddNodeAtEdgeProb	0.5	MultiplySigma (β)	0.9

6.2 Implementation

The implementation is done in a C++ library providing an interface to integrate SABRE into any target application. The library includes some experimental features that are not further mentioned since they are not relevant for the experiments in this work. The main classes are shown in Figure 6.6. Due to the direct mapping of the gene string to a repre-

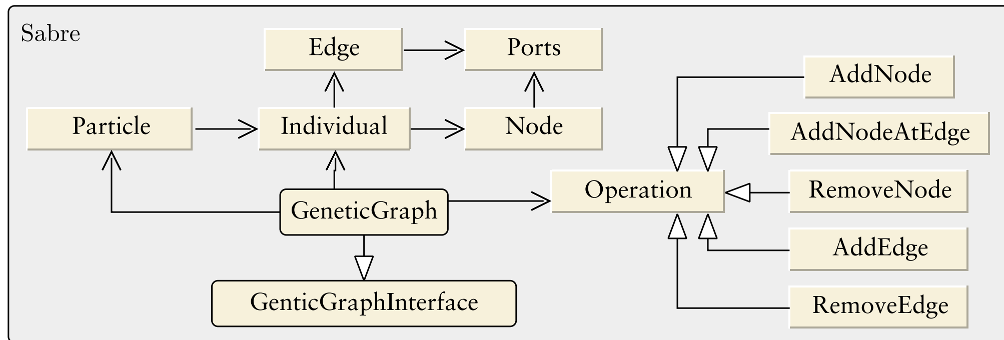


Figure 6.6: Software overview of SABRE.

senting graph, the gene string is directly managed as graph and stored in the `Individual` class. As introduced in the definition, the elements of the graph are represented by the `Node`, `Port`, and `Edge` classes. The PSO integration is done in the `Particle` class while `GeneticGraph` manages the main population, sub-populations, and the core development process of SABRE as previously introduced in Listing 7. The elements of the graph are still abstract and `Individual` in the implementation is representing the genotype. The phenotype is generated by the application. The interface allows to set up the available node and port types of the application, define a start graph by adding start nodes and edges, and to process the development through the methods `getNextIndividual` and `setFitness`.

6.2.1 Integration Into Learning Framework

A `SabreBehaviorSearch` module is implemented as an extension of the BOLERO framework and makes use of SABRE to generate and optimize BAGEL graphs for learning problems defined via BOLERO (see Figure 6.7). The module provides extensive configuration

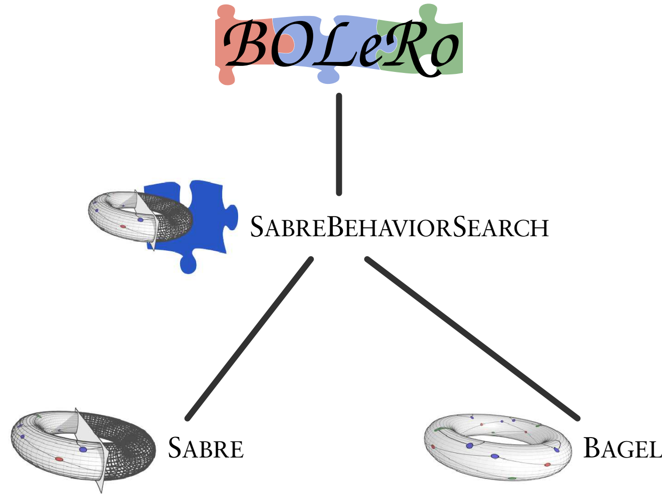


Figure 6.7: Integration of SABRE with BAGEL into BOLERO.

possibilities to define in detail which parts of a BAGEL graph can be evolved and optimized and how. Multiple BAGEL graphs can be evolved simultaneously and are evaluated by combining them via a top-level graph that can also be the result of an evolution. For this parallel development, the individual SABRE instances are independent from each other but should have the same configuration concerning the population sizes and number of parameter iterations. Every SABRE instance is configured individually. The configuration includes the SABRE parameters, the available BAGEL transfer and merge functions, the starting point of the evolution, and the option to add a fitness criterion to minimize the size of the resulting graphs. Additionally, every SABRE instance can be configured to use the graphs of other instances as node types for its own graph development. The starting point for every evolution is given by a BAGEL graph with optional SABRE specific annotations. The annotations can for instance hide a port – exclude it from being connected by edges –, configure how parameters are treated, or protect parts of the graph against mutations. The default transfer function of input and output nodes of BAGEL is `PIPE`. `SabreBehaviorSearch` allows to use the `SIGMOID` function for the output nodes to represent neural networks purely composed of sigmoid-based neurons. The main difference is that the `SIGMOID` function output value range is $[0, 1]$ while the the output range of the `PIPE` function is not limited. Another option is to represent one behavior output by two BAGEL outputs that are subtracted from one another. This is needed when using `SIGMOID` outputs to create negative output values. To generate behaviors that purely map the input values to output values without a memory feature, `SabreBehaviorSearch` can be configured to reset the BAGEL behavior after every set of input values is processed. To still make use of recurrent connections within the graph, a number of BAGEL graph evaluations per input value set can be defined. The reset handling and the usage of two output nodes representing one behavior output is handled similar as described for the BOLERO NEAT module in Section 4.2.2.

6.3 Validation Benchmarking

All benchmarks are performed with the BOLERO framework. Several setups are chosen for the evaluation of the novel introduced encoding and optimization method. The first

setup consists of black box function approximation for a set of mathematically defined functions. In the second setup the evolutionary algorithms are used to solve parameter optimization problems where the fitness landscapes are mathematically defined. The third setup is to evolve a controller for the car racing simulation TORCS. The controller has to complete a full lap while minimizing lap time. In the last setup a forward locomotion behavior of a simulated four legged walking robot have to be evolved.

The benchmarks are performed with NEAT, a GP algorithm, and SABRE, as described in Section 4.2.2. The algorithms are used in different configurations depending on the individual benchmark problems. However, to allow fair comparison, the phenotype representation of the NEAT and at least one SABRE configuration are the same for the single benchmark problems. Furthermore, the strategy parameters of the SABRE algorithm are kept constant and are not tuned to the individual problems. In case of the NEAT algorithm, the parameters are empirically tuned per problem because no well performing default parameters are known.

The HYPERNEAT and SABREPRE configurations of NEAT and SABRE respectively are only used in the last benchmark, since they are both explicitly designed for the legged robot benchmark. The HYPERNEAT approach has already been introduced in Section 2.2.2, while the SABREPRE configuration is described in the experiment section of the HyperRobot benchmark.

6.3.1 Black Box Function Approximation

This benchmark is used to analyze the performance of the PSO integration in comparison to a standard ES parameter mutation of SABRE. Furthermore, the impact of the available transfer functions is evaluated by providing different sets of functions. In total, four configurations of SABRE are tested, using a full set of transfer functions (SABRE_{full}) and only the sigmoid function (SABRE_{σ}) both with and without PSO usage: $\text{SABRE}_{full/PSO}$, SABRE_{full}, $\text{SABRE}_{\sigma/PSO}$, SABRE_{σ}. Since the results of this benchmark led to the chosen default parameters of SABRE, the parameters used in this benchmark differ and are listed in Table 6.2. Especially the default behavior of the ES parameter mutation is based on the results of this experiment.

Similar to SABRE, GP is used in two configurations: with a full and a reduced (only `SIG` and `ADD`) set of transfer functions. NEAT and the SABRE configurations use the linear transfer function for output nodes to be able to produce negative output values. Each combination of algorithm and benchmark function is replicated 100 times on a training data set of inputs drawn randomly from a uniform distribution $U(-1, 1)$. Every evolution experiment consists of 500,000 single evaluations and fitness is computed as *RMSE*. The optimization is done on the training data set, while an independently equally-sized created test data set is used to track fitness development and to compare the fitness distribution of the algorithms.

Benchmark Functions

In total six benchmark functions f_1-f_6 are used, chosen such that a succession of complexity both in types of computations and structure is realized, enabling more differentiated results than merely comparing the algorithms on a single arbitrary problem. f_1 has one input and one output, functions f_2-f_5 have three inputs and two outputs defined by separate equations and f_6, encoding a problem from robot kinematics, possesses four inputs and four outputs.

Table 6.2: Parameters used for the SABRE configurations using PSO and ES parameter mutation. Only the parameters that differ from the default configuration are listed.

SABRE Parameter	SABRE_{PSO}	SABRE_{ES}
NumSubPopulations (κ)	10	10
NumSubIterations (e_λ)	10	10
NumStartHiddenNodes	0	0
MutateStructureMode	multi	multi
MutateAddNodeProb	0.1	0.1
MutateAddNodeAtEdgeProb	0.1	0.1
MutateRemoveNodeProb	0.2	0.2
MutateAddEdgeProb	0.1	0.1
MutateRemoveEdgeProb	0.2	0.2
UsePSO	enabled	disabled
EvoJumping	disabled	disabled
MutateSingleParameter	multi	multi
MutateGaussCircle	disabled	disabled
StartSigma (σ_{max})	0.5	1.0
ResetSigma (σ_{min})	0.00005	0.00005
SabreBehaviorSearch Parameter		
ResetNet	true	true
NumberNetIterations	2	2

The benchmark functions are explained in detail in the following list:

- $f_1(x) = x^2$
  The square of the input x is a mathematical function that cannot be expressed directly by ANNs using the weighted sum for the inputs and the sigmoid function as activation function. Thus, it is expected that GP and SABRE including the `MULTIPLY` function perform better than their reduced configurations and the NEAT algorithm.

- $f_{2.1}(x_1, x_2, x_3) = \sigma(\sigma(x_1 + x_2) + \sigma(x_2 + x_3))$
  $f_{2.2}(x_1, x_2, x_3) = \sigma(x_1 + \sigma(x_1 + x_2))$
  This function is a combination of a nested sigmoid function and sums. It can be represented by ANNs with three input nodes, two hidden nodes and two output nodes. All tested algorithm variants have the possibility to accurately represent this function.

- $f_{3.1}(x_1, x_2, x_3) = \sigma(0.1\sigma(0.25x_1 + 0.4x_2 - 1.2) +$
  $0.3\sigma(x_2 + x_3 - 0.215) + 0.1123)$
  $f_{3.2}(x_1, x_2, x_3) = \sigma(x_1 + 0.0314)$
  This function is similar to f_2, the differences being the constants which have to be represented as additional nodes in GP and can be approximated by the other approaches with the weights and bias parameters.

- $f_{4.1}(x_1, x_2, x_3) = \frac{(x_1+x_2+x_3)}{1.4+x_3}$
  $f_{4.2}(x_1, x_2, x_3) = (x_1 + x_3) + (x_2x_3) - (x_1x_2)$
  Function f_4 utilizes all mathematical base operations and can be accurately represented by GP and SABRE making use of all available transfer functions.

- $f_{5.1}(x_1, x_2, x_3) = \frac{0.1(0.5x_1 + x_2 + x_3 + 1.3)}{1.4 + x_3}$
  $f_{5.2}(x_1, x_2, x_3) = 0.1((x_1 + 2.7x_3) + (-0.3 + x_2 x_3) - (1.34 x_1 x_2))$
  f_5 increases complexity in comparison to f_4 by adding constant parameters in the fashion of f_3 building on f_2.

- $f_6 : (j_\alpha, x, y, z) \mapsto (j_\alpha, j_\beta, j_\gamma, j_\delta)$
  This function represents the inverse kinematics calculation of a leg of SPACECLIMBER. The accurate solution to the underlying linear equation system can theoretically be represented by SABRE, but requires use of all available transfer functions and approximately 40 hidden nodes.

Results

Table 6.3 list all results with no significant difference, tested with Mann–Whitney U-test. For all other results the achieved fitness on the test data set is significantly different between the algorithms.

Table 6.3: Results with Mann–Whitney U-test with $p > 5\%$, showing no significant difference in the fitness distribution.

Test Function	Algorithm	vs.	Algorithm
f_2	SABRE$_{\sigma/ES}$		SABRE$_{full/ES}$
f_3	SABRE$_{full/PSO}$		GP$_{full}$
	SABRE$_{full/PSO}$		NEAT
	GP$_{full}$		NEAT
f_4	SABRE$_{\sigma/PSO}$		NEAT
	SABRE$_{full/PSO}$		GP$_{full}$
	SABRE$_{\sigma/ES}$		SABRE$_{full/ES}$
f_5	SABRE$_{full/ES}$		GP$_{full}$

For the simplest problem f_1, GP including the `MULTIPLY` function is the only algorithm always finding the correct solution (see Figure 6.8 f_1}, which is the only case of any algorithm to do so in any of the benchmarks. GP with only the `SIG` operation performs worst. Except for f_2 it performs worse in all other functions, as well. For f_2 it performs best due to its focus on structure search and no parameter tuning being needed for f_2. NEAT and SABRE always introduce parameters for the weights of edges and have to tune them. The best results for the more complex functions f_3 to f_6 are produced by NEAT and SABRE. For f_4 the difference between NEAT and SABRE is not significant, in case of f_3 and f_6 SABRE performs better, and for f_5 NEAT performs slightly better. The best version of the four SABRE configurations is SABRE$_{\sigma/PSO}$ which is always significantly better than the other three SABRE configurations.

The results indicate that SABRE in its standard form using PSO outperforms the variants using ES. It is noteworthy that SABRE performed better with a reduced function set, which may be explained by the comparably small incremental steps and thus the evolutionary stability resulting from the use of a sigmoid transfer function only. As for the overall performance of SABRE, one can conclude that for certain types of complex problems, such as inverse kinematics of real robots, SABRE is able to produce superior results compared to standard GP and NEAT. It has to be noted that it may be possible to find con-

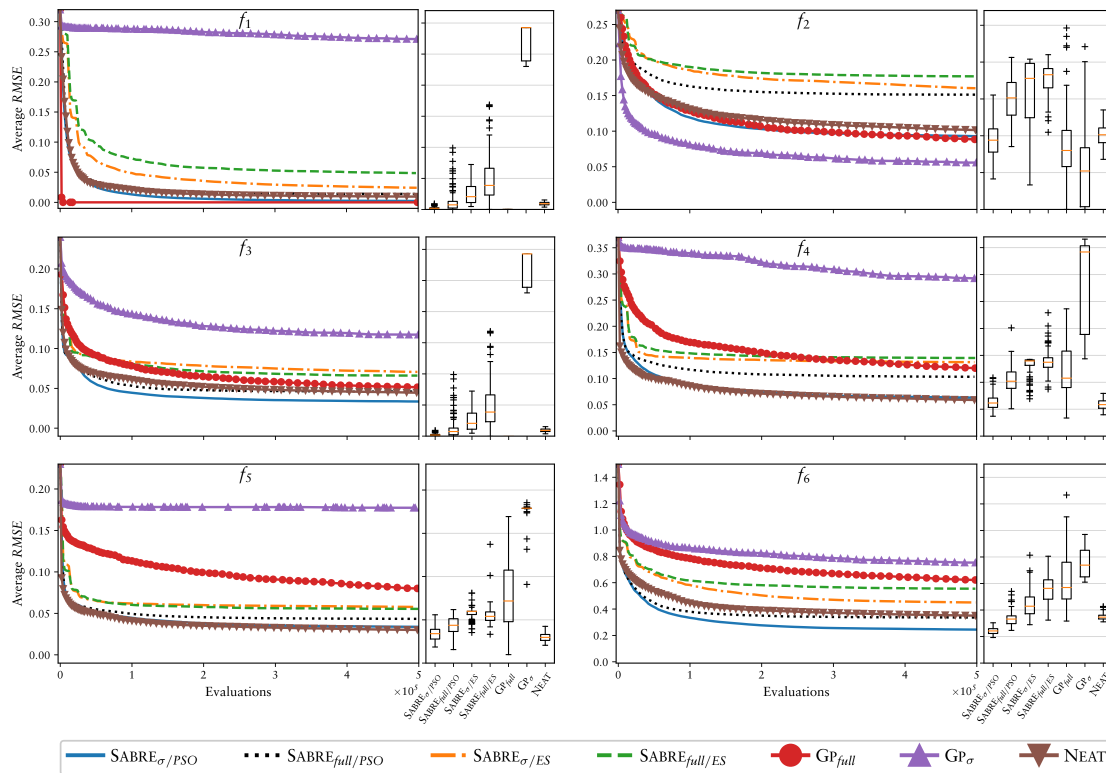


Figure 6.8: Development of average fitness over run-time of the tested algorithms for all benchmark cases. The corresponding fitness distribution in the final generation is displayed as box plots next to each development plot. Note that fitness is calculated as root-mean-square error, resulting in lower values representing higher fitness. Best-performing algorithms are GP for f_1, GP_σ for f_2, NEAT for f_5, and $\text{SABRE}_{\sigma/PSO}$ for f_3 and f_6. For test function f_4 NEAT and $\text{SABRE}_{\sigma/PSO}$ produce the best result without a significant difference (Mann–Whitney U-test, $p > 5\%$).

figuration parameters for the GP and NEAT algorithms changing these results, however despite various tests no such parameter sets could be found.

6.3.2 Analysis on Transfer Functions

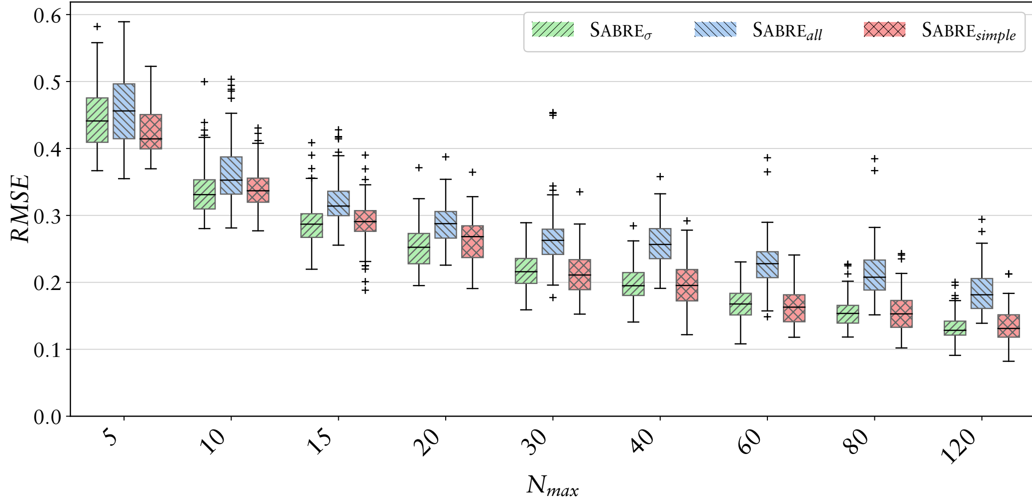


Figure 6.9: Minimum achieved *RMSE* distribution of the 100 runs each configuration of SABRE in setting $N_{max} = 120$. SABRE$_{all}$ has the highest median *RMSE* in all settings, while the other two configurations behave similarly.

To gain more insight into the adaptation behavior of SABRE, selected configurations of SABRE, possessing different sets of transfer functions, are tested on the test function f_6. These configurations are SABRE$_{all}$ with a full set of transfer functions, SABRE$_\sigma$ with only the sigmoid function, and SABRE$_{simple}$ with only `SUM` and `PRODUCT`. For the parameter optimization only the PSO method is used. All other parameters of SABRE are identical to the previous experiment if they are not explicitly introduced here. For each configuration, nine different *settings* are tested, in which the number of nodes is restricted to $N_{max} = (5, 10, 15, 20, 30, 40, 60, 80, 120)$ respectively, and the number of connections (or *edges* of the graph) to $E_{max} = 2N_{max}$. Each configuration of the algorithm is replicated 100 times on the training data set. For this experiment the analysis is done directly on the results of the training data set. Even though the absolute results between the training and test data sets are different, the ranking of the algorithms tested are the same on both data sets. In every setting, the algorithm is allowed to compute 1,000,000 fitness evaluations, resulting in 10,000 generations – given by the chosen parameters of SABRE.

In all settings, SABRE$_\sigma$ and SABRE$_{simple}$ achieve significantly better fitness than SABRE$_{all}$ with its full function set. The resulting distributions are depicted in Figure 6.9 which shows the achieved fitness for the 100 runs of each of the settings defined by limited node and edge numbers and respective algorithm configurations. The performance of each configuration can very accurately be predicted by fitting a rational function to the medians of the data, as is done in Figure 6.10, illustrating that a further increase in allowed network size will affect the achieved fitness to a diminishing degree. Performance of SABRE$_\sigma$ and SABRE$_{simple}$ differs only slightly. A significant difference between the configurations with reduced function sets can only be found in two out of nine settings using a Wilcoxon rank sum test with 95% confidence level, which yields p-values of 0.0128 ($N_{max} = 5$) and 0.0084 ($N_{max} = 80$) respectively.

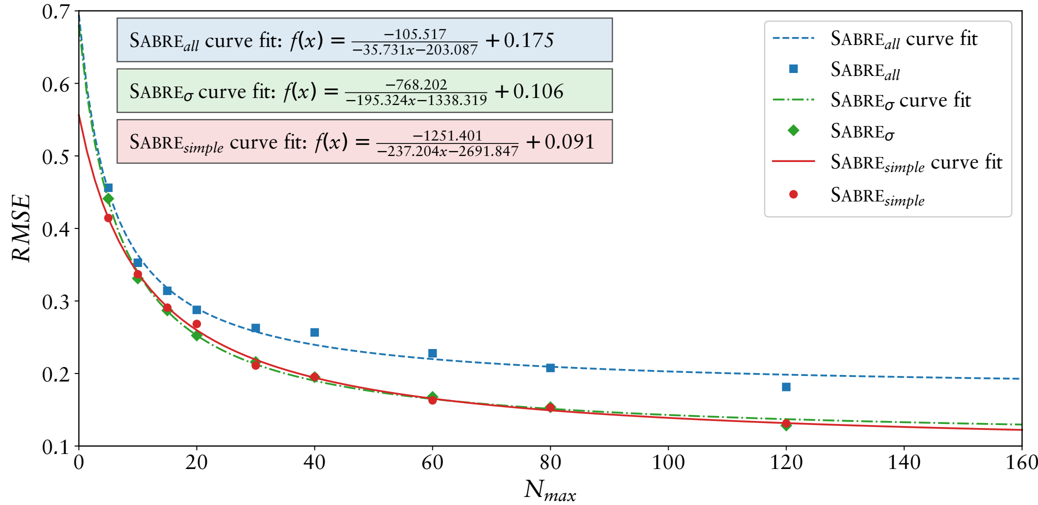


Figure 6.10: Fitted rational functions over median of minimum *RMSE* distributions of the 100 runs for all configurations and across all settings.

Despite the similar fitness ranges achieved by the σ and *simple* configurations, the algorithms' behavior over evolution time is quite different. In Figure 6.11, the development of maximum, average and minimum fitness over time is shown for the three variants of SABRE in the setting $N_{max} = 120$, plotted in two different scales.

It becomes clear that both SABRE$_{all}$ and SABRE$_{simple}$ show much larger variation around their 100-generation moving window means for both maximum and mean fitness than SABRE$_\sigma$. SABRE$_{simple}$ differs from SABRE$_{all}$ in that it starts out with low variance, but increases and finally surpasses SABRE$_{all}$. Nevertheless, the minimum values achieved by SABRE$_{simple}$ appear in the same range as those of SABRE$_\sigma$, despite its mean and maximum *RMSE* being very similar in range to SABRE$_{all}$; the latter only holds true until after generation 8000, when a comparably sharp increase in frequency of very large maximum *RMSE* values occurs and thus the mean *RMSE* of SABRE$_{simple}$ is also shifted. For SABRE$_\sigma$ on the other hand, maximum fitness climbs to a maximum value around halfway through the evolution and then slowly drops again, while mean fitness constantly decreases, resembling very closely the shape of minimum fitness values decrease in all three algorithm configurations. Here, it is almost impossible to distinguish between SABRE$_\sigma$ and SABRE$_{simple}$.

The difference in variability of SABRE$_\sigma$ compared to its competitors can be explained by how network parameters influence the overall network output in the three configurations. Independently of the input weights of a node, the sigmoid function produces outputs scaled from 0.0 to 1.0, thus changing the parameters can only produce a change of output of a node within that range. Other transfer functions can be much more sensitive towards parameter changes. An example is `DIV`, which approaches infinity for input values close to zero, resulting in a possibly large variance in the node output. Accordingly, the variance of the network output can be expected to be large over the whole evolution for SABRE$_{all}$. SABRE$_{simple}$ on the other hand needs some evolutionary time to concatenate multiple behaviors in such a way as to allow a cascade effect manifesting in even larger variability and values of the network output. This may also account for the observed increasing frequency of high *RMSE* values in SABRE$_{simple}$, as in large graphs, exponential effects and thus even small changes of a single finely tuned input weight may produce a large variance of the network output. Figure 6.12 depicts the number of fitness

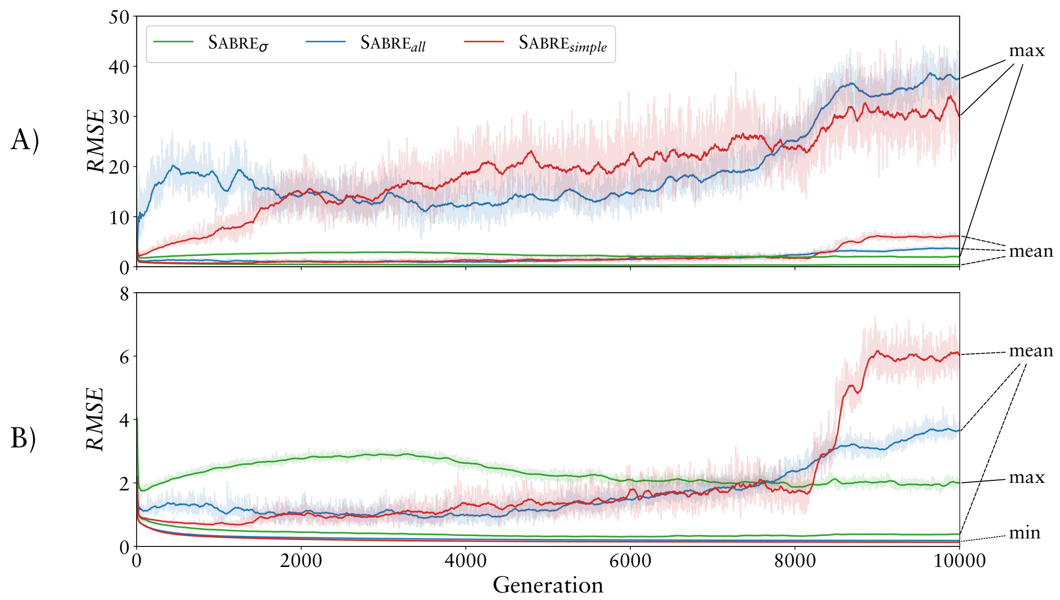


Figure 6.11: Maximum, mean and minimum fitness development of the 100 runs of setting $N_{max} = 120$ for all configurations over generation number. B) shows the lower part of A) with a larger scale, illustrating the differences in scale of the variation between the three estimators.

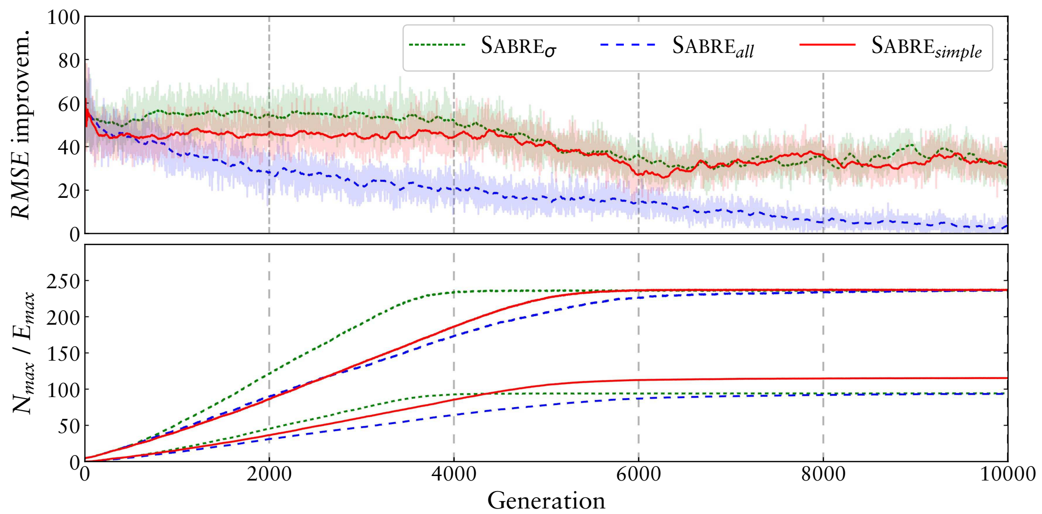


Figure 6.12: Frequency of individuals with improved fitness occurring per generation within the 100 runs of each configuration over generation number for setting $N_{max} = 120$. The probability of an improvement occurring in any one generation constantly deteriorates for SABRE$_{all}$ and is far more stable in the other two configurations. Plotted in parallel is the development of the number of edges (upper curves) and nodes(lower curves) over time.

improvements achieved per generation ($RMSE_{min}(t+1) < RMSE_{min}(t)$) across the 100 experiment runs, again for each configuration in the setting $N_{max} = 120$. While the three configurations start out similar, with high initial improvement rates, the latter then continuously decrease in SABRE$_{all}$, while SABRE$_{\sigma}$ and SABRE$_{simple}$ exhibit comparably constant behavior with on average about 55 and 45 improvements in every generation across the 100 runs up until generation 4000, from which on they decrease as well, which correlates with the maximum size of the networks being reached around that time. This correlation is also observed in other settings, with both the maximum size and the decrease of the number of improvements co-occurring earlier in the evolution, corresponding to the respective size limit. Nevertheless, the improvement rates stabilize again for both configurations from around generation 6000, staying at a lower level until the end of the experiments. Despite the similarity of development in SABRE$_{\sigma}$ and SABRE$_{simple}$, the overall number of fitness improvements is larger in the former. It also becomes clear why SABRE$_{all}$ stays at higher error values than the other two configurations.

Yet the difference between SABRE$_{simple}$ and SABRE$_{\sigma}$ shows that there must be another effect leveling the two configurations' fitness distributions. This effect can be identified when looking at the sizes of fitness improvements, plotted in Figure 6.13. The distributions of fitness improvements are here compared by classes of fitness values at which they were achieved, dividing the data in 16 classes with a step size of 0.1. It becomes evident that within each of the classes, the three SABRE configurations have very similar potentials for fitness improvement – given a beneficial mutation occurs – across most of the fitness range. However, SABRE$_{simple}$ has a slight advantage in the very low regions. This difference becomes slightly more obvious when looking at the overall frequency of fitness improvements across the range of already-achieved fitness, see Figure 6.14. While the two reduced configurations SABRE$_{\sigma}$ and SABRE$_{simple}$ have a similar distribution, SABRE$_{\sigma}$ shows more improvements at the lower end of the fitness range.

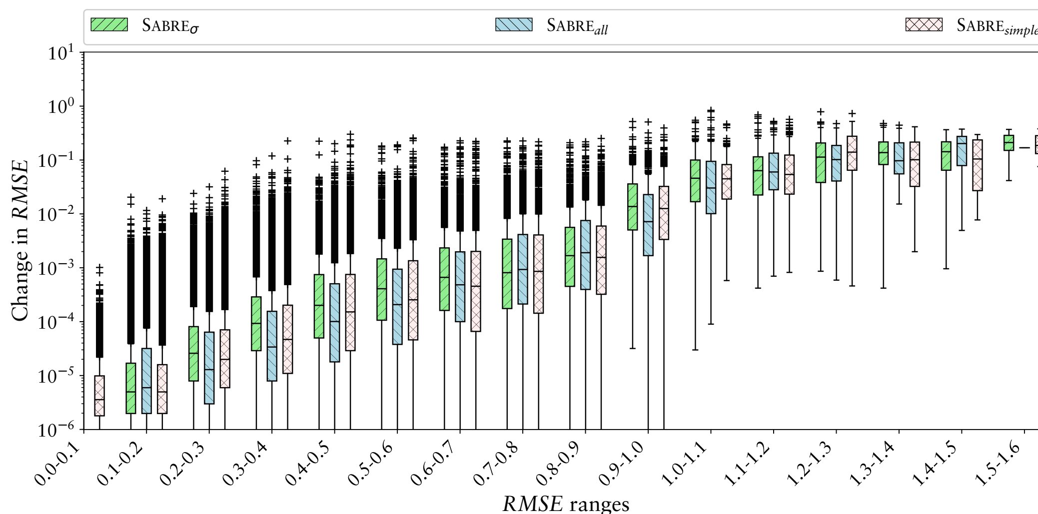


Figure 6.13: The box plot depicts the distribution of the changes of the *RMSE* within given *RMSE* ranges, without a significant difference between the three configurations of SABRE.

For the analyzed test function, the three different node representations possess roughly the same potential in improving the fitness for a given reached fitness over most of the observed fitness range, with SABRE$_{simple}$ exhibiting a wider range than the other two configurations. What differs clearly between the algorithms, however, is the probability of a beneficial mutation occurring in the first place.

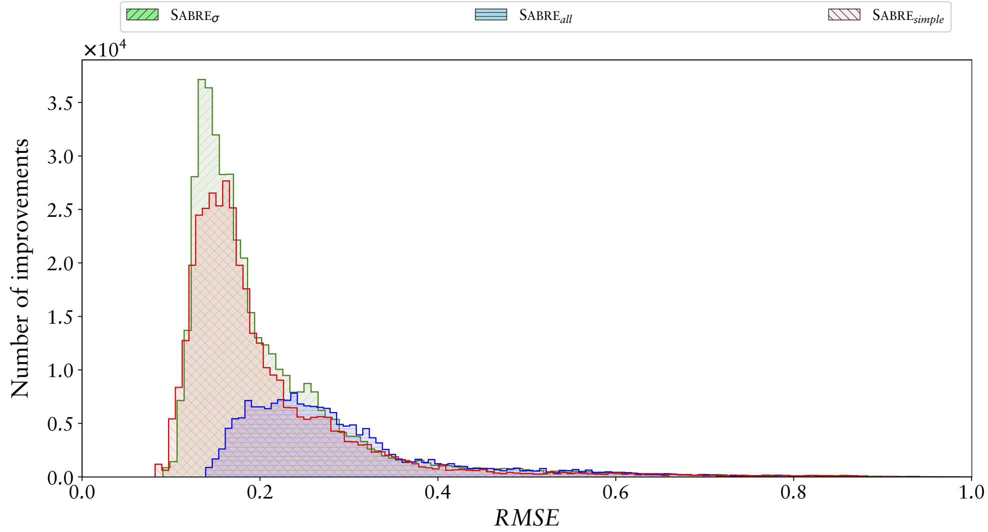


Figure 6.14: The histogram shows the number of improvements over the reached *RMSE*.

SABRE$_{all}$ failed to get any closer to the perfect solution than its two competitors, in fact, it is lagging behind them, as the number of fitness improvements constantly decreased over the evolution progress, indicating that the evolutionary process is running into local minima more rapidly in this configuration of SABRE. Additionally, SABRE$_{all}$ behaves much more unstable at the beginning of an evolution.

As result of this analysis, SABRE is adapted to perform parameter mutations of a single randomly selected gene, as already presented in the SABRE definition. Additionally, this analysis leads to the default parameters, which are chosen such that SABRE in principle tries to adapt slowly to a solution. For the resulting default configuration the original experiment with 500,000 evaluations per single optimization is repeated. Except function f_1 and f_6, the new configuration of SABRE outperformed all other algorithms and configurations. Table 6.4 compares the results of the new default configuration with the best algorithm of the previous experiments and the SABRE$_{\sigma/PSO}$ configuration. In case of test function f_6 the average fitness development of SABRE$_{default}$ indicates that the configuration adapts more slowly than the SABRE$_{\sigma/PSO}$ configuration. Due to that observation the approximation of f_6 is repeated performing 1,000,000 evaluations for each evolution. While SABRE$_{\sigma/PSO}$ with a median result of 0.245 does not further improve after the 500,000 evaluations, SABRE$_{default}$ continues with a median result of 0.225. Thus, in this experiment, the new default configuration can benefit from performing more evaluations.

Table 6.4: Comparison of new default SABRE configuration with best results from the benchmark. The best results are printed with bold and italics text.

Function	Previous best median (best)	SABRE$_{\sigma/PSO}$ median (best)	SABRE$_{default}$ median (best)
f_1	***0.0 (0.0)***	0.0018 (0.0003)	0.0030 (0.0010)
f_2	0.0522 (0.0)	0.0940 (0.0414)	***0.0504 (0.0202)***
f_3	0.0342 (0.0032)	0.0342 (0.0032)	***0.0015 (0.0003)***
f_4	0.0615 (0.0370)	0.0615 (0.0370)	***0.0369 (0.0193)***
f_5	0.0297 (0.0206)	0.03401 (0.0186)	***0.01661 (0.0122)***
f_6	***0.2426 (0.1929)***	***0.2426 (0.1929)***	0.2608 (0.1853)

6.3.3 CEC13 Test-Functions

The CEC13 parameter environment (see Section 4.2.1) is used to compare the algorithms in a well defined fitness landscape. However, the benchmark is not a typical control problem as only the first output of a behavior is evaluated as a parameter set. The first five functions are used with the dimensions 10 and 30. For every combination of algorithm, test function, and parameter dimension, 50 individual evolutions are performed. The evolutions with a parameter dimension of 10 are terminated after 500,000 evaluations and the ones with 30 parameters after 1,500,000 evaluations. To enable the behaviors to produce the output, a virtual input with a constant value is used as bias. Two net-iterations are used by all algorithms to create the output parameters, allowing recurrent connections to influence the parameter set.

Table 6.5: Parameter optimization results: The bold italics cells highlight the best results.

Function No. (Dim.)	SABRE median (best)	NEAT median (best)	PUPPY median (best)
1 (10)	***-1400.0*** (***-1400.0***)	***-1400.0*** (***-1400.0***)	4618.26 (-1338.0)
1 (30)	***-1400.0*** (***-1400.0***)	***-1400.0*** (***-1400.0***)	39266.8 (25328.7)
2 (10)	***25396.1*** (***3122.09***)	368776.5 (39858.6)	9.1e+7 (7581370.0)
2 (30)	***1.09e+6*** (***327696.0***)	1.38e+7 (4.8e+6)	2.49e+9 (1.1e+9)
3 (10)	***376870.0*** (***-1199.99***)	1.67e+10 (2.49e+9)	1.94+11 (4.27+10)
3 (30)	***6.51e+10*** (***2.02e+10***)	2.78e+11 (1.31e+11)	5.91+18 (8.65+14)
4 (10)	***-1099.99*** (***-1100.0***)	7911.22 (1232.15)	99688.0 (18140.2)
4 (30)	***-1100.0*** (***-1100.0***)	45987.8 (11760.7)	240972.5 (85850.6)
5 (10)	***-1000.0*** (***-1000.0***)	***-1000.0*** (***-1000.0***)	1209.65 (-765.94)
5 (30)	***-1000.0*** (***-1000.0***)	-999.99 (***-1000.0***)	5832.43 (3167.72)

In case of the first and fifth test function, NEAT and SABRE are able to find the global minima and thus the difference in the results is not significant. Since the PUPPY algorithm has no explicit mechanism to optimize parameters in the genotype, it performs worse. In all cases with significant differently results, the best overall and median results are produced by the SABRE algorithm.

This benchmark is very interesting because of the directly defined fitness landscape. However, due to the structure development, the three algorithms do not operate directly in the fitness landscape and the result give just a rough idea about the performance of the algorithms. Nevertheless, in the following benchmarks the rank of the algorithms does not vary much, which indicates that a behavior search algorithm that performs well in this benchmark – if it is not explicitly designed for parameter optimization – has a good chance to perform good in control problems, too.

6.3.4 Torcs

The TORCS environment described in Section 4.2.1 is used for this benchmark problem. For each test algorithm 50 individual evolutions are executed. An evolution is terminated after 500,000 performed evaluations. For all algorithms the sigmoid function is used for the output nodes and two network outputs are used to generate one behavior output as introduced in Section 4.2.2. The networks perform two iterations on one input parameter

set and are reset afterwards. The results, including the best and median fitness and lap time, are given in Table 6.6. The SABRE algorithm with its default configuration produces the best and median best results in this benchmark.

Table 6.6: Results of the car racing benchmark. The bold italics cells highlight the best results. The first row includes the fitness values reached while the second row shows the corresponding lap times.

	SABRE median (best)	NEAT median (best)	PUPPY median (best)
Fitness	***-5830.45*** (***-5870.23***)	-5801.0 (-5836.72)	-24.15 (-3913.88)
Lap time	***91.27*** (***83.59***)	97.24 (90.188)	– (–)

6.3.5 HyperRobot

The HyperRobot environment described in Section 4.2.1 is used for this benchmark problem. The learning algorithms and their variants applied are NEAT, HYPERNEAT, SABRE, SABREPRE, and GP. 50 evolutions are performed for every learning algorithm, including 500,000 evaluations each.

SABREPRE Configuration

SABREPRE makes use of the possibility to integrate foreknowledge by defining a control structure and simultaneously evolve a main graph and two sub-graphs used for the front and rear legs control. The predefined control structure uses the evolved front leg sub-graph as a node for the left and right front leg and the rear leg sub-graph for the rear legs respectively. A time counter is added as input to these graphs and a phase shift on the leg patterns can be defined by adding an offset to the counter. The two sub-graphs are evolved by two additional evolution processes running in parallel to the evolution of the main control graph. The sensor readings are only available to the top graph with the predefined structure where the evolution can create additional nodes and edges to connect the sensor readings to the motor outputs. Due to this configuration, a motor pattern is very likely to be produced by the front and rear leg sub-graphs which is applied symmetrically on the left and right legs, while the main evolution is responsible for closing the control loop. Additionally, a predefined sub-graph can be used as a transfer function by SABRE. The predefined module implements a parameterized Gaussian pattern, where the phase and the width of the Gaussian bell can be adapted through input values. The Gaussian function is defined by $f(x) = e^{-50b(x+a)^2}$, while x is mapped into the range $[0, 1]$ by the *fmod* function and defines the position in the Gaussian pattern. The parameter b changes the width of the Gaussian bell. The parameter a, an offset on the x value, shifts the Gaussian pattern within the input phase. By overlapping multiple shifted Gaussian patterns, a resulting joint pattern can be achieved. The idea of the Gaussian patterns is basically derived by Dominici et al. (2011) and is a precursor of the presented approach in the next chapters.

Results

Table 6.7: The results of the legged robot benchmark. The bold italics cells highlight the best results. SABREPRE produced the best median result while the pure SABRE configuration produces the best overall result.

	SABRE	SABREPRE	NEAT	HYPERNEAT	PUPPY
Median	1.34	***1.3***	1.96	2.06	4.88
Best	***0.797***	0.899	1.379	1.739	2.83

The results are shown in Table 6.7. SABREPRE produced the best median result while the pure SABRE configuration produces the best overall result. In this setup the SABRE algorithm performs significantly better than the NEAT versions. This might be due to the complex configuration possibilities of NEAT and there might be a configuration that performs significantly better than the one used.

6.3.6 Conclusion

Overall, the default SABRE configuration performs well on a large set of different benchmark problems. It is expected that it can find adequate solutions if the fitness landscape allows a “step by step” adaption. Furthermore, in combination with BAGEL, SABRE allows to tune the search space by defining start structures or building blocks for more complex problems. This is roughly shown in the HyperRobot benchmark and is used extensively in the following experiment sections of this thesis to analyze different joint pattern representations and to perform evolutions for walking behaviors of different legged robots.

Chapter 7

Central Pattern Control Approach

Learning the control of kinematically complex legged robots without predefining a main control concept generally results in a too large search space and complex fitness landscape, as discussed in Chapter 2. Predefining components such as the pattern representation limits the search space and increases the probability to generate locomotion behaviors that can be used on real systems. This chapter presents a control approach based on central pattern generators (CPG). This control approach is designed to be suitable for legged robots in general and it is applied on several robotic systems in Chapter 8. The main architecture – the CPG itself – is responsible for generating step phases, the phase shift between legs, and allows the modulation of the step frequencies and phases based on sensory input. How the step phases are transferred to appropriate joint patterns is a critical question when designing a controller. In this chapter, different methods are presented and experimentally compared with respect to their evolvability of defined joint patterns. To perform the experiments, the whole control approach is implemented in BAGEL which enables the optimization of the different joint pattern representations via SABRE.

7.1 Central Pattern Generator

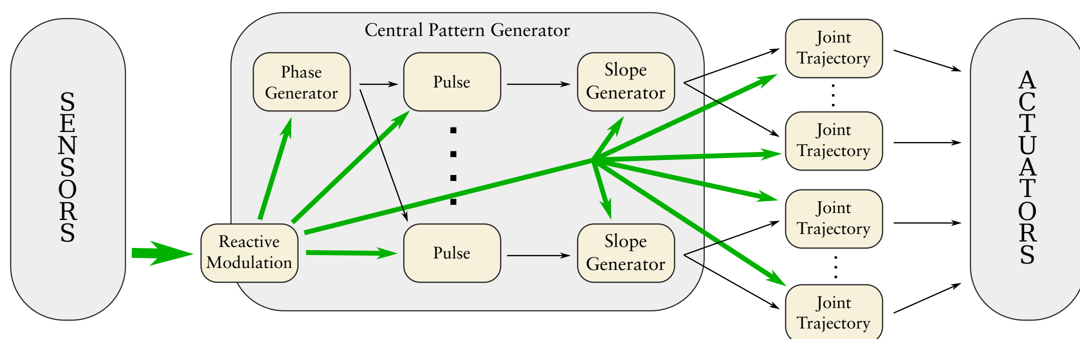


Figure 7.1: Basic control concept of the CPG based approach.

The CPG produces a global phase ϕ with $0 \leq \phi \leq 1$, multiple phase shifts φ_i and local phases ϕ_i. It is used as a base pattern generator, whereby the projection $j(\phi)$ from the local phases to the angular joint patterns is explored in the following section. Figure 7.1 depicts the general control architecture. The `Reactive Modulation` component is

a black-box whose inner structure is not further defined for this chapter. However, in Chapter 8 different reactive modules are evolved to produce a navigation behavior, walk over rough surface, and compensate for disturbances. The `Phase Generator`, `Pulse`, and `Slope Generator` modules are parts of the CPG. The `Joint Trajectory` module varies through the different joint pattern representations. The resulting phase-shifted slope – the output of the `Slope Generator` module – represents the individual leg phases and is transferred into joint positions by the `Joint Trajectory` module. An example combination of these modules is shown in Figure 7.2.

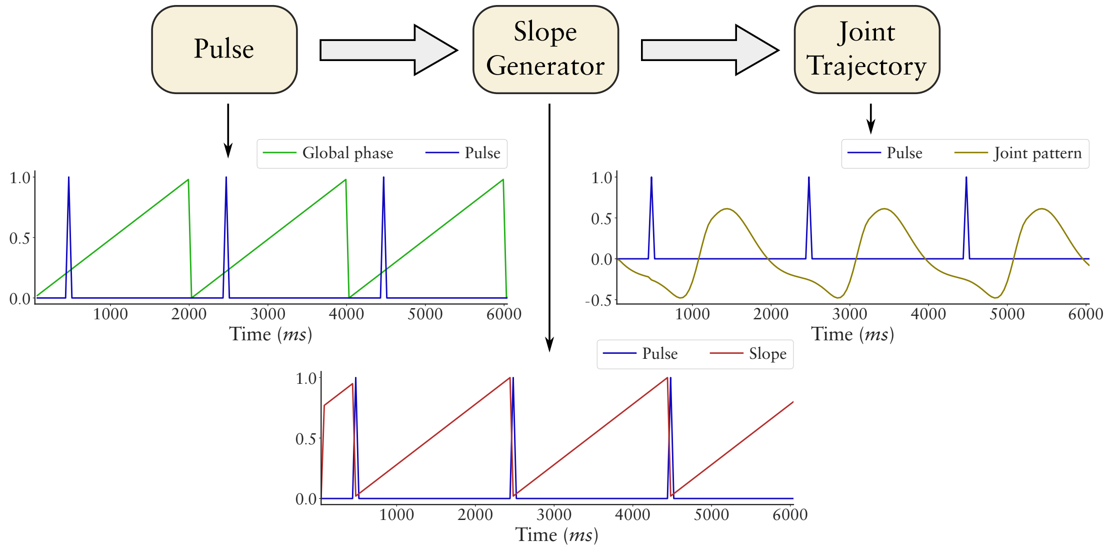


Figure 7.2: Basic concept for the CPG-based joint trajectory generation. The global `Phase Generator` module is not depicted though its output (Global phase) is plotted in the first graph. Based on the global phase the `Pulse` module generates a pulse pattern that is transferred to a "local" phase in the `Slope Generator` module. The local phase is aligned by the pulse pattern and it is transferred to a joint pattern in the `Joint Trajectory` module.

The global phase pattern is represented by a saw-tooth function. The corresponding BAGEL module generates the global phase ϕ_G based on the update period (t) and the desired frequency (f). It is defined as:

$$\phi_G(0) := 0, \tag{7.1}$$
$$\phi_G(t + \Delta t) := (\phi_G(t) + \Delta t f)\%1, \tag{7.2}$$

where Δt is defined by the execution clock. Figure 7.3 depicts the module in the BAGEL representation visualized in GRAPHGUI.

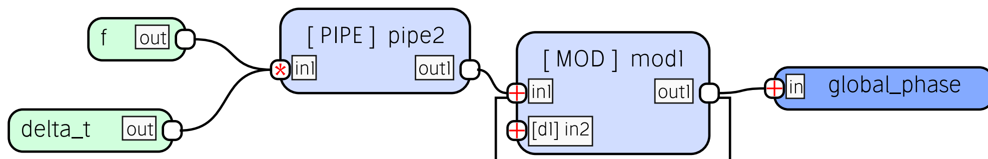


Figure 7.3: The global phase module modeled in GRAPHGUI.

A pulse function $p(\phi_G, \varphi)$ is used to create a trigger signal for a shifted slope. The input of the pulse function is ϕ_G and a desired phase shift φ with $0 \leq \varphi < 1$. The function is defined by:

$$p(\phi_G, \varphi) = \begin{cases} 1 & \text{if } 1 - \Delta t < \phi_G + 1 - \varphi < 1 + \Delta t, \\ 0 & \text{else.} \end{cases} \tag{7.3}$$

`Slope Generator` produces a phase ϕ for the leg cycle control. The phase starts whenever the trigger input ρ has a value of 1.0 and ϕ maintains its value once it reaches 1.0 until the next trigger input is given. The component is defined by:

$$\phi(t + \Delta t, \rho) = \begin{cases} 0 & \text{if } \rho > 0, \\ \phi(t) + \Delta t f & \text{if } \phi(t) < 1 \\ 1 & \text{else.} \end{cases} \tag{7.4}$$

7.2 Joint Pattern Representation

In this section six different approaches to generate joint patterns are presented. The *Sine-Based*, *Gaussian*, and *Pendulum* approaches are designed for this thesis with the goal of optimizing the pattern representation to evolutionary generate stable locomotion patterns. The *Fourier*, *Neural Oscillator*, and *Sigmoid* ANN approaches are commonly used methods from literature.

7.2.1 Sine-Based Pattern

Let o, a, p_1, and p_2 be constants defining the shape of the joint trajectory, with:

$$o, a, p_1, p_2 \in \mathbb{R} \mid -\pi \leq o, a, p_1, p_2 \leq \pi, \tag{7.5}$$

where o and a define the offset and amplitude of the sine wave, p_1 represents the start position in the sine wave of a virtual swing phase, and p_2 defines the start position of a virtual stance phase. The constants s_p determine the scale of the first phase as:

$$s_p \in \mathbb{R} \mid 0 \leq s_p < 1. \tag{7.6}$$

Then let:

$$\delta \in \mathbb{R} \mid \delta := \begin{cases} p_2 - p_1 & \text{if } p_2 > p_1, \\ p_2 - p_1 + 2\pi & \text{else.} \end{cases} \tag{7.7}$$

Using these constants the joint function is defined by:

$$j(\phi) := \begin{cases} o + a(\sin(\frac{\delta\phi}{s_p} + p_1)) & \text{if } \phi \leq s_p, \\ o + a(\sin(\frac{(2\pi - \delta)(\phi - s_p)}{1 - s_p} + p_2)) & \text{else.} \end{cases} \tag{7.8}$$

To use the Sine-Based pattern approach with SABRE, a BAGEL node is implemented receiving the phase as first input and using three more inputs that are hidden for connecting edges. The bias and default parameters of the three additional inputs define the parameters $p1$, $p2$, and sp of the modified sine wave. The amplitude and offset are defined by the

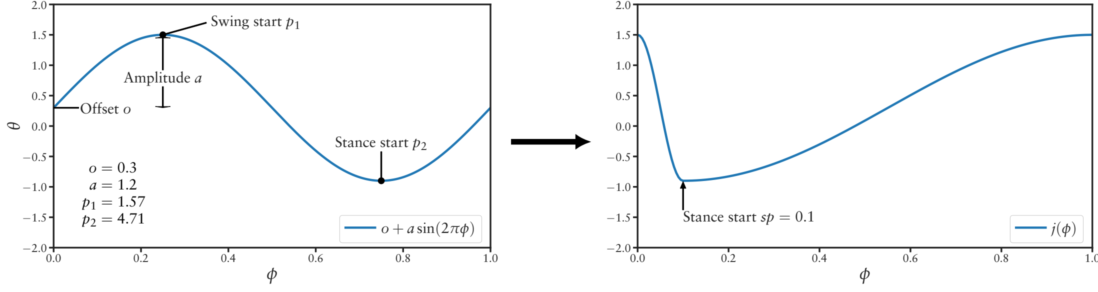


Figure 7.4: Example transformation of a sine curve into the joint trajectory generated by the Sine-Based pattern approach.

evolved network through the weight of the connected output edges and the bias of output nodes. The approach is originally designed to utilize one modified sine wave per joint trajectory. The modified sine waves are able to represent basic joint patterns required to control a legged robot. Therefore, two configurations of the Sine-Based pattern approach are proposed. The first one (SINEPG$_1$) uses one wave per trajectory, which results in a parameter optimization problem with $5N$ parameters to optimize, where N is the number of joint trajectories to approximate. The second configurations (SINEPG$_2$) allows SABRE to use the modified sine wave as a building block and thus can combine multiple waves to one joint trajectory, whereby the number of parameters to optimize is $5M$, where M is the number of building block instances.

7.2.2 Gaussian Pattern

The Gaussian pattern approach (GAUSS) is similar to a Gaussian process regression, but optimized to generate a periodic function and to support asymmetric shapes. A parameterized Gaussian pattern is used as a building block for the joint-based control. The parameters are defined by the constants a, b, c, d, o with $a, b, c, d, o \in \mathbb{R}$. a defines the amplitude of the peak, b influences the width of the Gaussian bell, c with $0 \leq c < 1$ defines the position of the bell within the step phase, d influences the slope of the incline and decline, and o adds an offset to the pattern. First, a function for the bell phase shift is defined considering the slope constant d:

$$b(\phi) := ((\phi + c)\%1)d - 0.5. \tag{7.9}$$

Then the Gaussian pattern function is defined by:

$$g(b(\phi)) := \begin{cases} o + ae^{-50b\left(\frac{0.5\phi}{d-0.5}\right)^2} & \text{if } \phi > 0, \\ o + ae^{-50b\phi^2} & \text{else.} \end{cases} \tag{7.10}$$

Since the influence of the slope constant d is nonlinear and asymmetric with respect to the incline and decline slope (see Figure 7.5), a correction is defined for the optimization process:

$$d := 0.56(0.2\pi + 0.4984 o_d)^{2\pi}, \tag{7.11}$$

where o_d is the tuning parameter for the optimizer to adapt the slopes. The result is shown in Figure 7.6. Additionally, d influences the phase of the Gaussian bell which is

adapted by a phase offset. Then, c is defined by:

$$c := \begin{cases} o_c - 0.55|o_d - 0.5|^{0.83} & \text{if } o_d > 0.5, \\ o_c + 0.55|o_d - 0.5|^{0.83} & \text{else.} \end{cases} \tag{7.12}$$

Where o_c is the tuning parameter for the optimizer to adapt the phase of the Gaussian bell. The phase adaptation is shown in Figure 7.7. A similar adaption is done for the width of the Gaussian bell (see Equation 7.13 and Figure 7.8). The final parameters o_b, o_c, and o_d are defined for parameter range of $[0, 1]$. Similar to the Sine-Based approach, a BAGEL node is implemented providing an input port for the phase and three parameter ports for o_b, o_c, and o_d hidden for evolving edges. The offset and the amplitude parameters are again handled by the evolved network.

$$b := 0.14\pi + \pi(0.1\pi + o_b)^{\pi}. \tag{7.13}$$

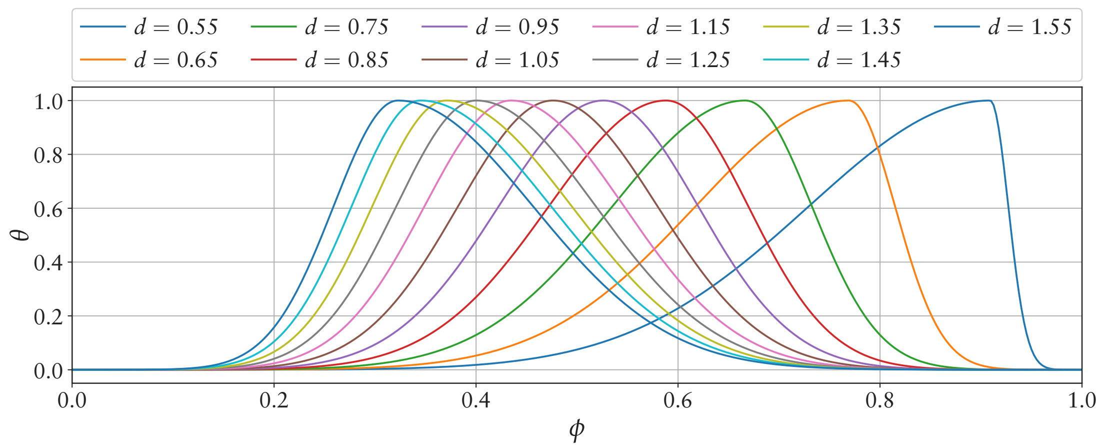


Figure 7.5: Non-linear influence of the incline parameter d to the slope and phase shift of the Gaussian bell.

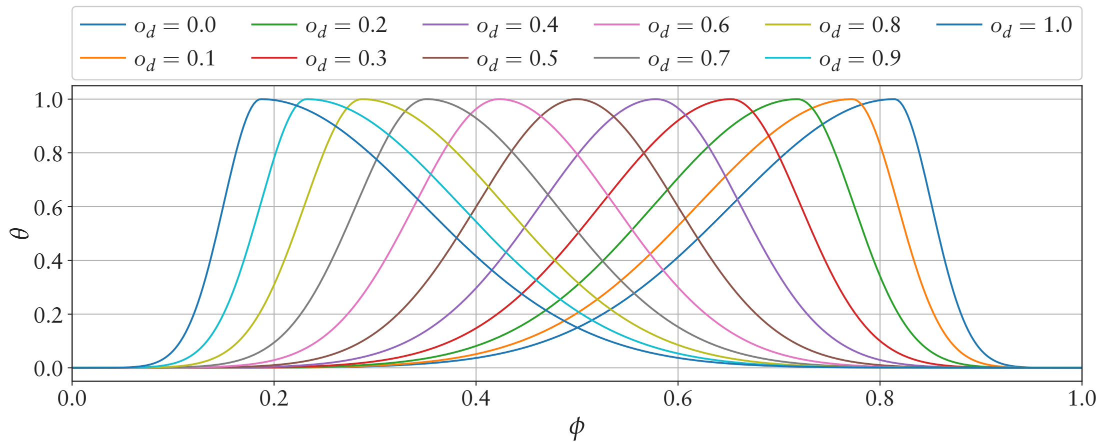


Figure 7.6: The graph depicts the linear influence of the parameter o_d that is transformed to parameter d.

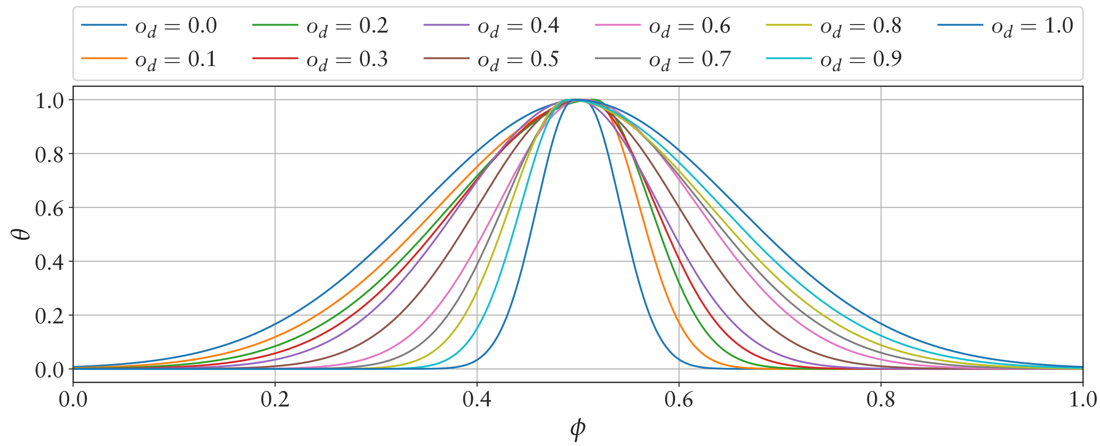


Figure 7.7: The resulting influence of o_d to the shape of the curve. To minimize the influence of o_d to the phase shift the final parameter c is calculated based on o_c and o_d.

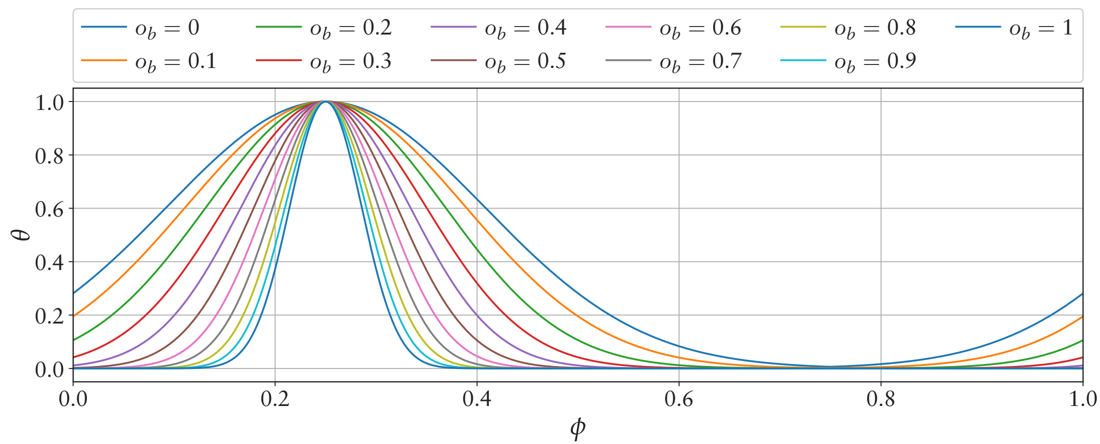


Figure 7.8: Similar to parameter o_d the width of the Gaussian bell is influenced linearly by o_b in a range of $[0, 1]$.

7.2.3 Pendulum Pattern

The Pendulum pattern approach is designed to partially compensate for the disadvantage of position-controlled electric actuators without any compliance control (see Section 2.1.2). The main idea is to create a controller that produces a virtual force F_p to drive the motor to a fixed neutral position. The more the motor differs from the neutral position the higher is the force to drive the motor back. Thus the force is defined through a model of a pendulum. The virtual force is applied to the pendulum influencing its virtual angular velocity ω_p. The virtual velocity is used to determine the pendulum position which defines the new position for the motor controller. The pendulum position for the next iteration of the controller can be updated with the sensory information of the real motor. The pendulum controller gets an external force input F_e which is added to the virtual force – always applied on the tangential direction. The virtual force is calculated by:

$$F_p = g \cdot \sin(\theta) + F_e, \tag{7.14}$$

where g represents the gravity. For this abstract model g is defined by $g := -1$. After applying the external force to the pendulum, the velocity is dampened by a parameter d_p. Thus, the angular pendulum velocity (ω_p) is given by:

$$\omega_p(t + \Delta t) = d_p\Big(\omega_p(t) + kF_p\Delta t\Big), \tag{7.15}$$

where k is the length of the pendulum divided by the moment of inertia. However, for this abstract model k is set to $k := 1$. The new motor position is given by:

$$\theta(t + \Delta t) = \theta(t) + f_p\omega_p(t + \Delta t)\Delta t, \tag{7.16}$$

where the parameter f_p controls the eigenfrequency of the virtual pendulum. Additionally, the controller simulates the virtual pendulum with a fixed time step of 1 s independently of the outer control frequency.

A short rhythmic impulse on the external force input can already create pendulum swinging as can be seen in Figure 7.9.

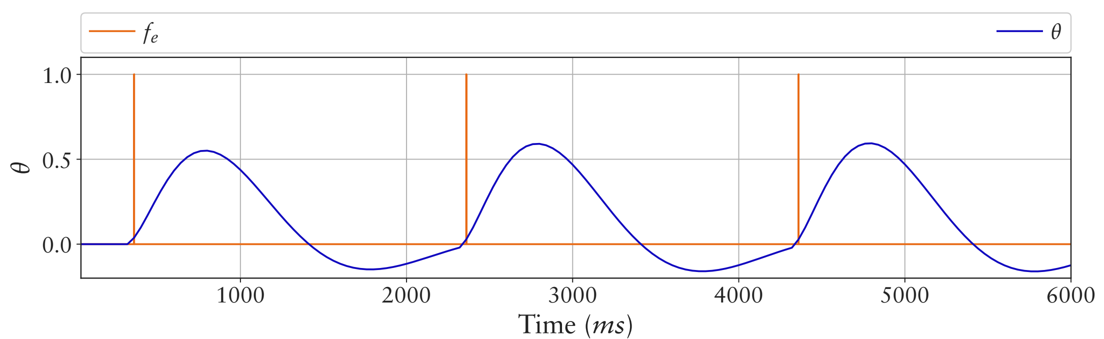


Figure 7.9: Example of a joint pattern created by the pendulum controller giving just an impulse based rhythmic force input.

The pendulum controller needs several cycles to approach a stable pattern, while receiving an input pattern for F_e. How fast the approach merges to the final shape depends on the input pattern. An example behavior is shown in Figure 7.10. To compensate the start phase of the pendulum controller, each test function should be iterated until the pendulum controller converges. The error of the last iteration is used for the evaluation.

As waiting for the pendulum controller to converge would inflate the evaluation time, a compromise is chosen by optimizing the initial values of $\omega_p(t_0)$, $\theta(t_0)$, and $F_p(t_0)$.

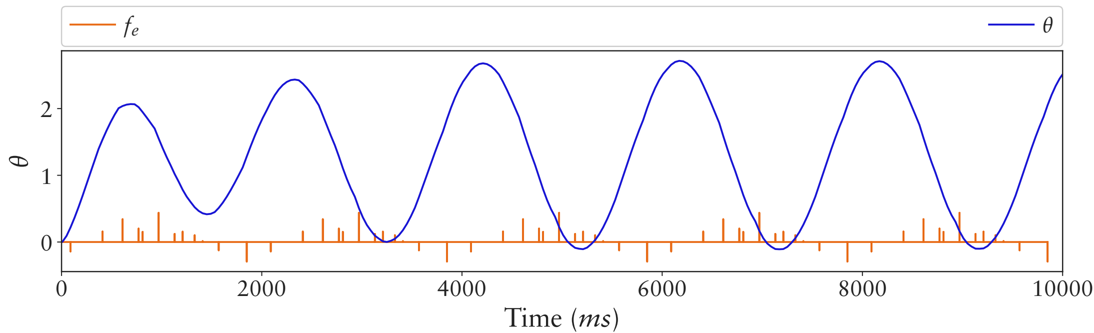


Figure 7.10: Example of the fading behavior of the pendulum controller.

The pendulum controller is implemented as a BAGEL node providing one input port for the pulse and four parameter ports for f_p, d_p, a gain for the amplitude, and an offset parameter. To create the pulse input pattern, aligned with the step period, a *fire neuron* is implemented getting two inputs and generating one pulse output. The default output is zero representing an inactive state of the neuron. The parameters of the first input port are protected and it is always connected to the step period with a connection weight of 1.0. Thus, the first input is the original unmodified step period. The second input is never connected and its bias defines the point in the step period when a firing output of 1 is generated. Internally, the bias input is projected to a value in the range of $[0, 1]$. The weights of the connected edges at the output can scale the activity of the neuron. With these constraints SABRE is creating only one forward-connected hidden layer composed of firing neurons. To have a comparison to this rather simple pattern, a second configuration is implemented using the Gaussian modules to generate the input signal for the pendulum controller. The two configurations are referred to as PENDULUM$_{Fire}$ and PENDLUM$_{Gauss}$ respectively.

7.2.4 Fourier Series

A Fourier series is a common mathematical method for representing periodic functions. It combines the sine and cosine functions with weights and varying frequencies. By using it as a pattern generator, it ensures that all frequencies are a multiple of 1. Thus, the function is defined by:

$$fs(\phi) = a_0 + \sum_{i=1}^{i<N} (a_i \sin(2\pi i\phi) + b_i \cos(2\pi i\phi)). \tag{7.17}$$

In this function, a and b are the coefficients that define the shape of the pattern. The function is used with an order of 8 ($N = 8$) resulting in 17 parameters for one joint trajectory. The order of 8 was empirically chosen via preliminary experiments and has proven to be well-suited for the test functions. A BAGEL node (Fourier node) is developed implementing $fs(\phi)$ and providing one input for the phase and 17 inputs for the parameters. For every joint pattern to approximate, one single Fourier node has to be generated whose input parameters are optimized by SABRE.

7.2.5 Neural Oscillator

Based on Pasemann et al. (2003), a neural network oscillator is implemented, modeling two interconnected non-linear neurons. The output of the neurons a_1 and a_2 are defined by:

$$a_1(t + \Delta t) := \alpha\Big(\cos(\gamma)\tanh(a_1(t)) + \sin(\gamma)\tanh(a_2(t))\Big), \tag{7.18}$$

$$a_2(t + \Delta t) := \alpha\Big(-\sin(\gamma)\tanh(a_1(t)) + \cos(\gamma)\tanh(a_2(t))\Big), \tag{7.19}$$

with $a_1(0) := 1$ and $a_2(0) := 0$. The parameter γ mainly influences the frequency of the pattern and α adapts the slope. Both influence the amplitude and α changes the frequency as well. With $\alpha = 1 + \epsilon$ and $\epsilon << 1$ the waveform becomes almost sine-shaped. The joint function also has the parameters a and o to adapt the amplitude and offset. In contrast to the other approaches it is not defined over phase ϕ but over time t. The function is given by:

$$j(t) = o + a * a_1(t). \tag{7.20}$$

Two example patterns that can be produced with this approach are shown in Figure 7.11.

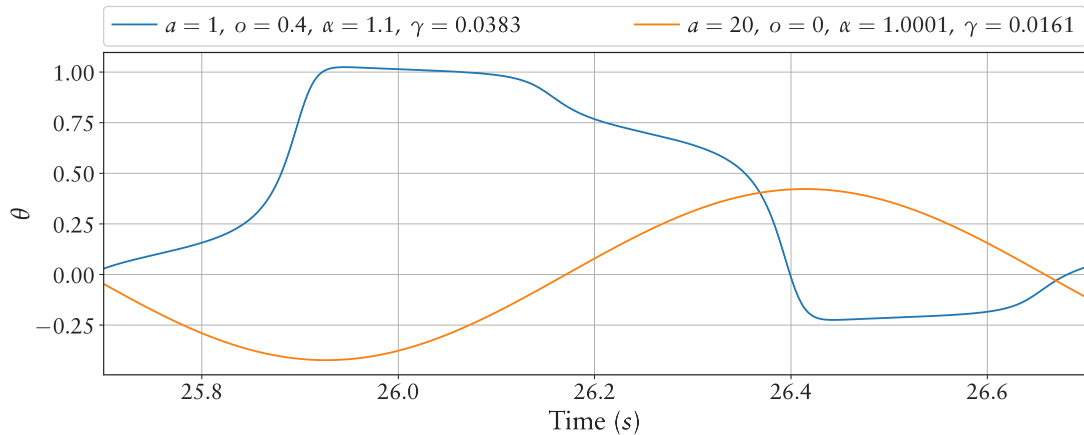


Figure 7.11: Two example patterns that can be generated by the Neural Oscillator approach.

The main disadvantage of this approach is that the phase cannot be defined directly by a parameter. To generate a phase shift, the frequency has to be adapted until a desired offset is reached. Additionally, the two neurons need some time to converge to a resulting oscillation based on the chosen parameters α and γ. For the validation part of this section, the network is calculated for a defined and constant initialization time t_{init} until the evaluation process starts. Due to this initialization, the two parameters γ and α also influence the phase by changing the frequency. Therefor, the phase shift parameter φ is added to t_{init} with a constant factor p_{scale}. For the experiments in this chapter the parameters are set to $t_{init} = 10000\Delta t$ and $p_{scale} = 2000\Delta t$. Two configurations, NEURALOS$_1$ and NEURALOS$_2$ are used in the experiment section. NEURALOS$_1$ uses one neural oscillator per joint pattern and only adapts the parameters a, o, φ, α, and γ. NEURALOS$_2$ uses the neural oscillator as a building block in SABRE to generate a joint pattern.

7.2.6 Sigmoid ANN Pattern

For the sigmoid pattern approach, SABRE is used to develop a recurrent neural network, based on the sigmoid transfer function. The overall configuration is similar to the validation part in the SABRE section 6.3.1, where it is used for general function approximation, mapping input values to desired output values. Additionally, in this section, two configurations are added in which the previous output values are additional input values of the network, simulating the concept of the network receiving the phase and joint position sensors of a robot. Through the generalization proprieties of neural networks, this configuration could already include an adaptive behavior for external impacts on the robot. The three different resulting configurations are:

- **SIGMOID$_1$:** Directly maps the phase ϕ to the joint trajectories.
- **SIGMOID$_2$:** For N trajectories N networks are developed with $j_i(\phi, \theta_i) \mid i \in N$.
- **SIGMOID$_3$:** For N trajectories one network is developed with $j(\phi, \boldsymbol{\theta}) \mid |\boldsymbol{\theta}| = N$.

The main disadvantage of ANNs with respect to optimization is that ANNs do not necessarily produce patterns. Thus it is part of the optimization process to match the value at the end of the phase to the one at the start.

7.3 Experiments

This section investigates the capability of the different joint pattern representations to generate pre-defined patterns while being optimized with SABRE. Even though a joint pattern representation is able to represent a given trajectory theoretically it does not necessarily perform well in an optimization application. These experiments are performed to select appropriate representations for the experiments with legged robots in Chapter 8. Altogether, eight experiment setups are designed with different complexities in terms of the pattern shape and number of patterns that have to be evolved simultaneously.

For all test setups, evaluation is performed by minimizing the fitness defined by the root mean square error (*RMSE*) of the approximation and the test function. For each representation an optimization is performed 100 times to generate a distribution used to compare the different representations. Other aspects of the different representations than the fitness value reached are discussed and shown in individual experiments. Due to computation cost, the test patterns are not split into training and test data sets. Thus, the generalization properties of the representations concerning function regression are not evaluated in general, but are discussed based on individual results and representations. To decide whether a pattern representation is precise enough for a possible use on a real system, the following condition is used:

$$e^{max} \leq 0.04a, \tag{7.21}$$

where a is the amplitude of the test pattern and e^{max} is the maximal error between the median optimized result and the target function:

$$e^{max} := max(|f(x) - OPT_{median}(x)|) \cdot a. \tag{7.22}$$

7.3.1 Sine-Based Test-Functions

A single sine function can be used to generate simple and smooth patterns, which can already be used to generate legged locomotion for proper robotic systems. Overlapping sine functions can produce more complex patterns, which is the basis of the Fourier series approach. For comparing the different approaches, three different sine patterns, shown in Figure 7.12, are used with increasing complexity by adding more terms to the function definitions.

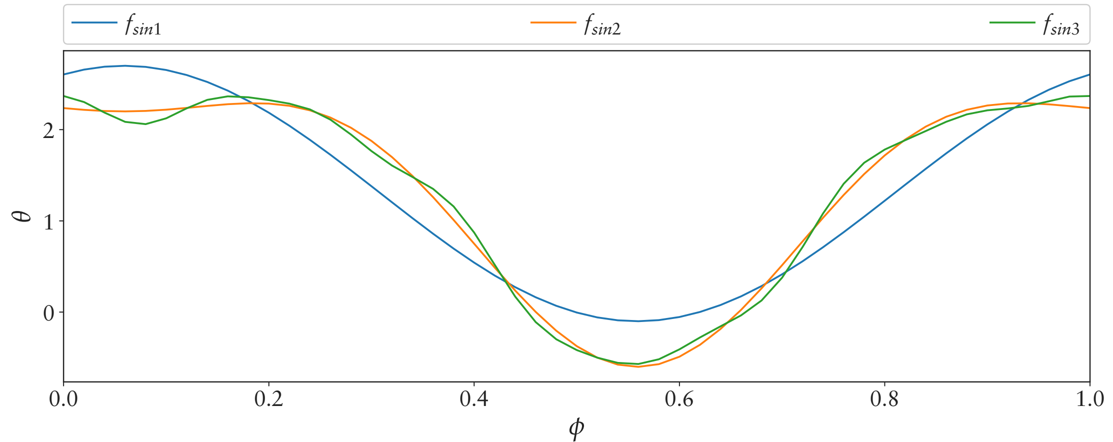


Figure 7.12: Different sine test functions for the pattern generators.

Single Sine Wave

The most simple shape used in this validation is a single sine wave modified by an offset, a scaling and a phase shift. The parameters are chosen as follows:

$$f_{sin1}(\phi) = 1.3 + 1.4 sin(1.2 + 2\pi\phi). \tag{7.23}$$

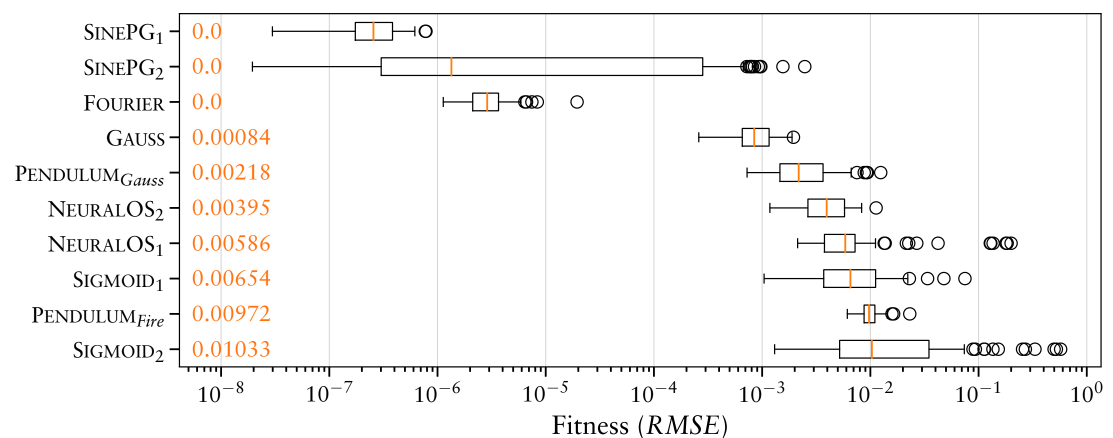


Figure 7.13: Fitness distribution of 100 performed repetitions for all approaches on the single sine curve.

The resulting fitness distribution of the different representations, optimized for this sine pattern, are shown in Figure 7.13. The two SINEPG representations and the Fourier series approach perform best on this test function. Which is due the fact that they are based on the sine function. However, all representations could be optimized to approximate the sine test function according to Equation 7.21 precisely enough to enable the usage of a representation as a joint pattern generator, provided such a basic pattern is sufficient for a target system. The worst median result of all optimizations, produced by the SIGMOID_2 representation, is shown in Figure 7.14. The best result of the SIGMOID_2

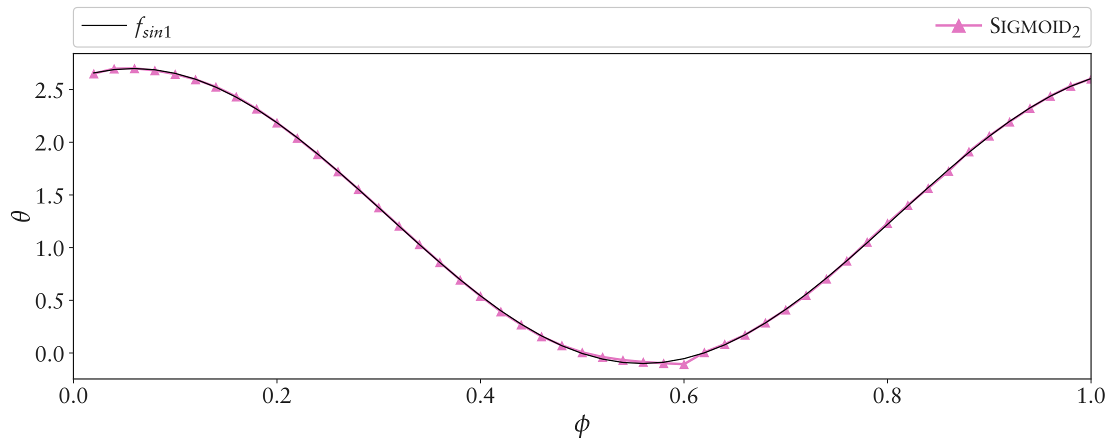


Figure 7.14: Median result of the SIGMOID_2 approach on the single sine curve, which is the worst median result of all approaches. The approximation is still close enough to the target function that it could be used to control a real system.

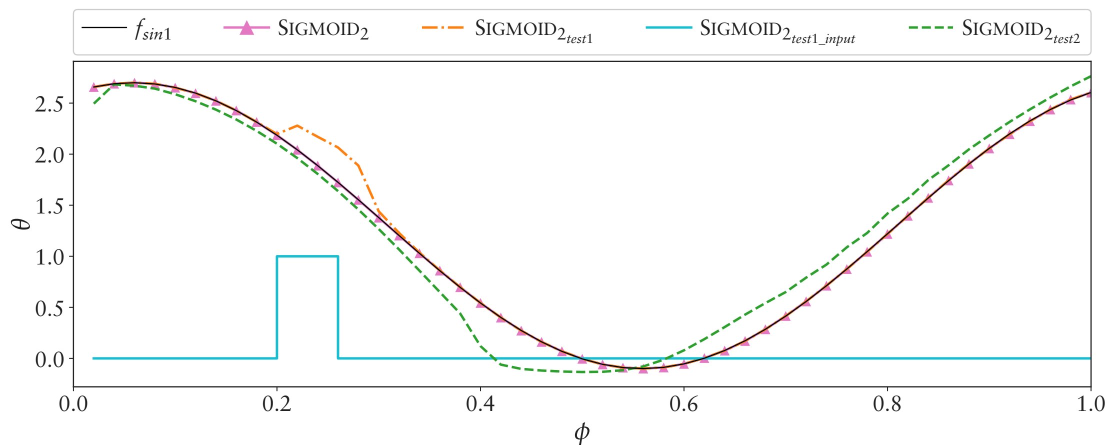


Figure 7.15: Example modulation of the SIGMOID_2 approach. The approach maps the phase and the last output value, representing a position sensor, to the target value. For this example $\text{SIGMOID}_{2_{test1}}$ gets an additional input for the last value shown by $\text{SIGMOID}_{2_{test1_input}}$. $\text{SIGMOID}_{2_{test2}}$ depicts the result if a constant offset is added to the last value, simulating a motor controller not reaching the target position.

approach on the f_{sin1} function was used to show an example behavior of the approach to an additional input value given to the sensor (last output) input. Figure 7.15 shows the resulting behavior. The first test adds an offset on a given interval while the second test adds a constant offset for the whole function. The results show that the shape of the pattern can be influenced differently, which could be used by a reflex or modulation

behavior. However, how the additional input influences the shape of the pattern strongly depends on the underling structure and it is not guaranteed that there is an influence at all. A similar example is shown for the best result of the PENDULUM$_{fire}$ approach see Figure 7.16. The figure shows the learned pendulum input $I_{pattern}$ that defines the shape of the resulting wave. As a test, a single additional input value I_{extern} is created that could represent a value generated by a reactive behavior. The plot PENDULUM$_{fire2}$ shows the resulting shape with the additional force applied, while PENDULUM$_{fire3}$ depicts the wave form one cycle later where the shape comes back to the original one.

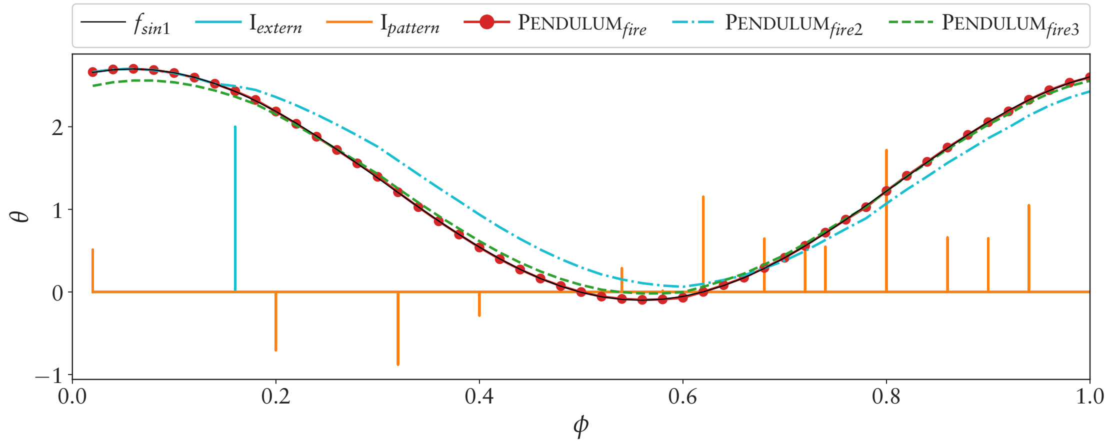


Figure 7.16: Similar to the SIGMOID$_2$ modulation an additional pulse input is generated for the PENDULUM$_{fire}$ approach. I$_{pattern} is the original force pattern evolved to generate the sine curve. I_{extern}$ depicts the single external input, PENDULUM$_{fire}$ the original curve, PENDULUM$_{fire2}$ the modulated curve resulting by the additional input, and PENDULUM$_{fire3}$ depicts the next cycle of the pattern fading back to the original curve.

Two Overlapping Sine Waves

The previous single wave is extended by adding a second sine wave with different frequency, phase shift and amplitude, given by:

$$f_{sin2}(\phi) = 1.3 + 1.4 sin(1.2 + 2\pi\phi) + 0.5 sin(0.5\pi + 2.4 + 4\pi\phi). \tag{7.24}$$

The resulting distribution of the different representations, optimized for this overlapping sine pattern, is shown in Figure 7.17. The Fourier series can easily represent this combined sine pattern and thus produces the best result. The SINEPG$_1$ and NEURALOS$_1$ approaches cannot represent the wave form, limiting how close the optimization can fit the target function. Figure 7.18 depicts the median results of the two approaches together with the median result of the SIGMOID$_2$, which is the worst performing approach of the ones that can theoretically represent the waveform. At least, theoretically, the SIGMOID$_2$ approach can be as good as SIGMOID$_1$. However, dealing with the larger number of inputs decreases the optimization performance. The SIGMOID$_2$ approach already fulfills the accuracy property defined in Equation 7.21.

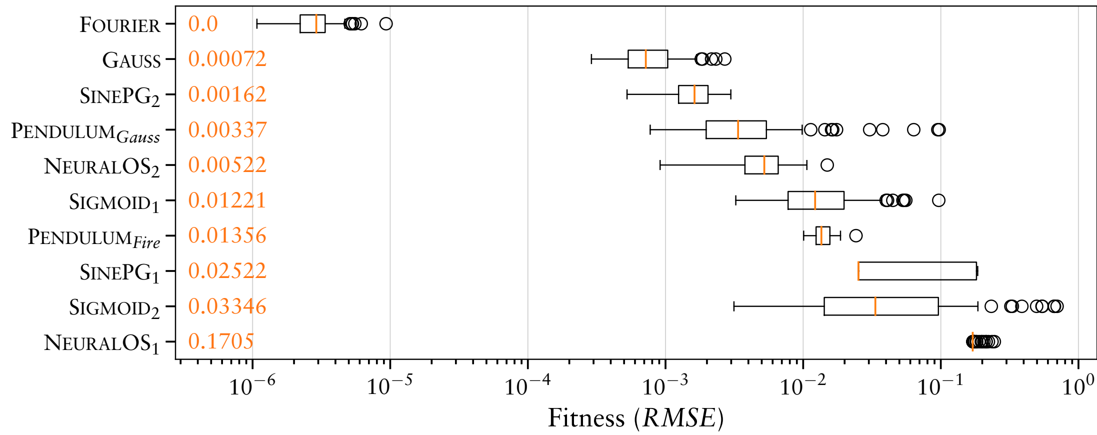


Figure 7.17: Fitness distribution of all tested approaches for the two overlapping sine waves.

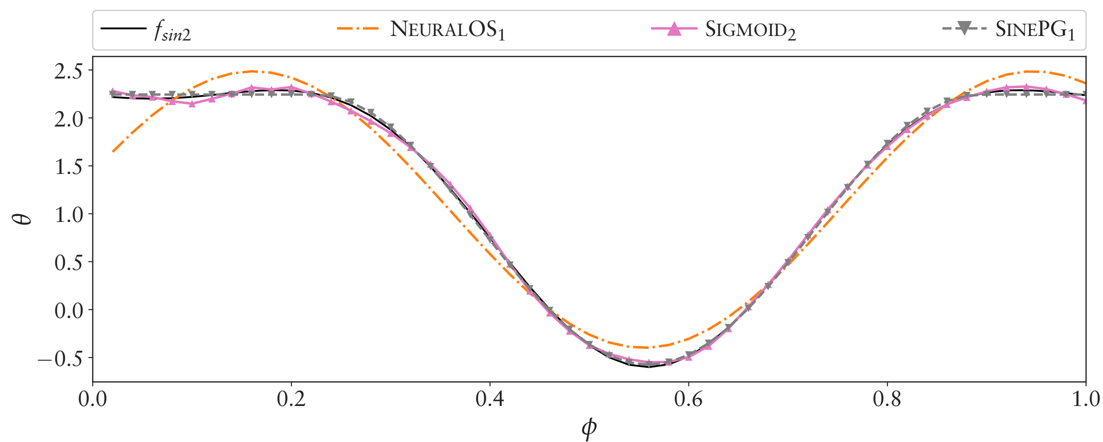


Figure 7.18: Median result of SINEPG$_1$, NEURALOS$_1$, and SIGMOID$_2$ configuration for the two overlapping sine waves. Since all other representation fit closer than SIGMOID$_2$ they are not plotted.

Multiple Overlapping Waves

For this test function two more sine terms are added to the previous function, resulting in a more detailed shape. The resulting target function is defined by:

$$\begin{aligned} f_{sin3}(\phi) = & 1.3 + 1.4sin(1.2 + 2\pi\phi) + \\ & 0.5sin(0.5\pi + 2.4 + 4\pi\phi) + \\ & 0.1sin(0.75\pi + 6 + 10\pi\phi) + \\ & 0.05sin(1.3\pi + 9.6 + 16\pi\phi). \end{aligned} \tag{7.25}$$

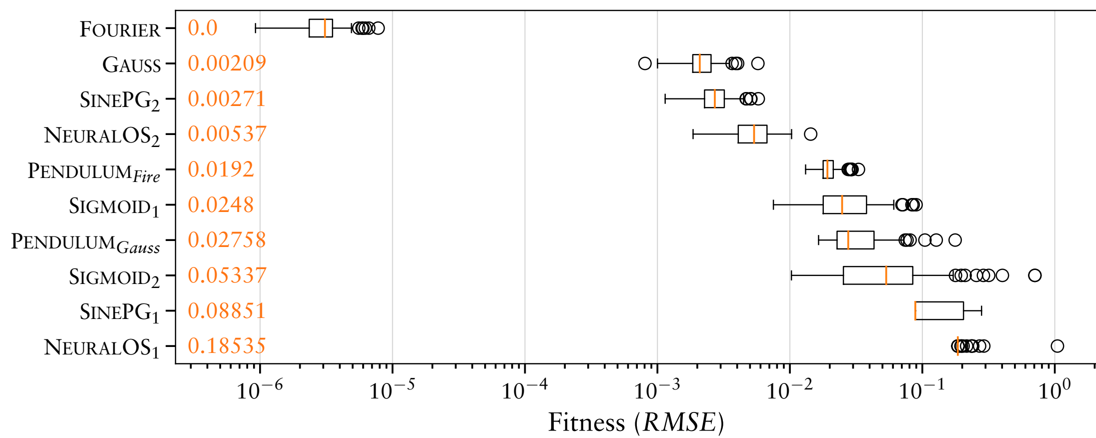


Figure 7.19: Fitness distributions of the approaches for the multiple overlapping sine waves.

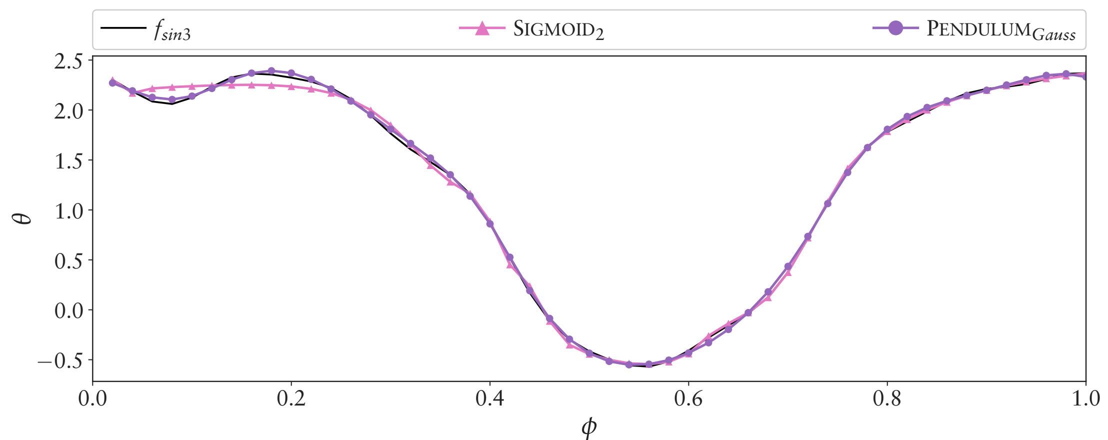


Figure 7.20: Median result of SIGMOID$_2$ and PENDULUM$_{Gauss}$ approaches for the multiple overlapping sine waves.

The resulting distribution of the different representations, optimized for this sine pattern, are shown in Figure 7.19. Again the Fourier series can easily represent this pattern and produces the best result, while the SINEPG$_1$ and NEURALOS$_1$ approaches cannot represent the wave form and thus perform worst. Similar to the previous experiments, the worst fit of the other approaches is achieved by SIGMOID$_2$. The median results of the SIGMOID$_2$ and PENDULUM$_{Gauss}$ approaches are shown in Figure 7.20. This time, at the beginning of the test pattern ($\phi \leq 0.25$), the SIGMOID$_2$ does not approximate the shape

well. The median result of the next best approach, the PENDULUM$_{Gauss}$, approximates the shape of the entire pattern close enough according to Equation 7.21.

7.3.2 Gauss-Based Wave

For this experiments, different Gaussian bells are used as test functions. The approaches are tested on a single pattern, four patterns at once, and 20 patterns at once. Optimizing multiple test functions at once increases the complexity. Some approaches for example can learn a common base pattern for all or for a subset of the target patterns and then adapt the common pattern to the different target ones. This can be an advantage if a common base pattern is contained in the target patterns. On the other hand, this can lead to generating "artificial" local optima. However, on the target application, the evolution has to generate multiple joint patterns at once to allow evolving the locomotion control of a legged robot. Therefore, test setups are designed with multiple target patterns as well. The patterns are generated by the equations 7.9 and 7.10 by varying the parameters a, b, c, and d.

Single Wave

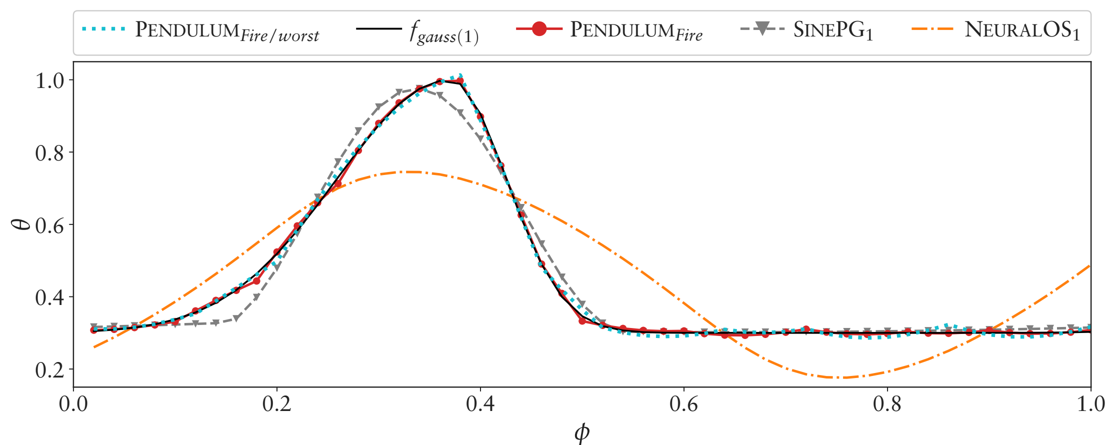


Figure 7.21: Example median results of the optimization of the single Gaussian test function.

The single Gaussian test function is shown in Figure 7.21. With the constant phase, it could represent a stance phase of a joint responsible for lifting the leg in the swing phase. The part between a ϕ of 0.1 and 0.5 could represent a swing phase by lifting up and down a leg respectively. With the exception of NEURALOS$_1$ and SINEPG$_1$, all approaches perform well, as can be seen in Figure 7.22. The GAUSS approach performs best because it is based on the same mathematical function as the test function. The PENDULUM$_{Fire}$ approach is on the fourth last rank and its worst result approximates the pattern already close enough.

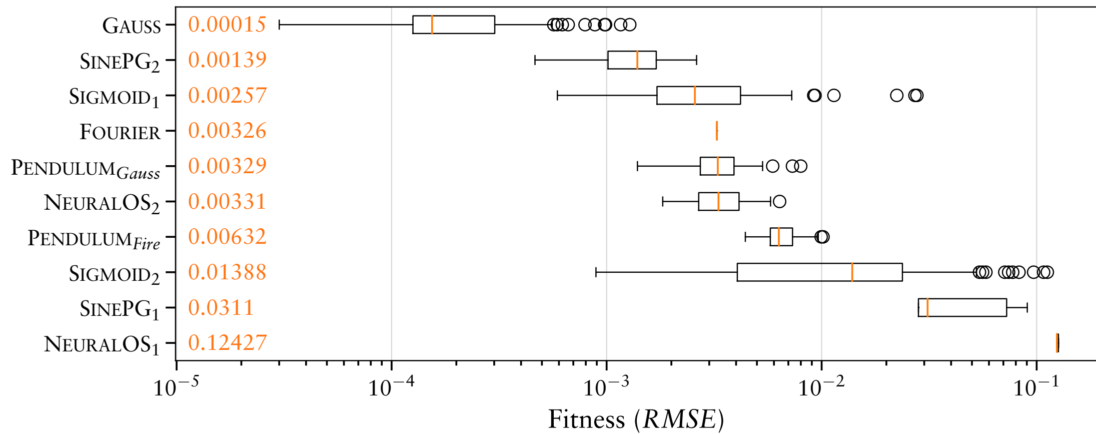


Figure 7.22: Fitness distribution of all approaches on the single Gaussian test function.

Four Waves

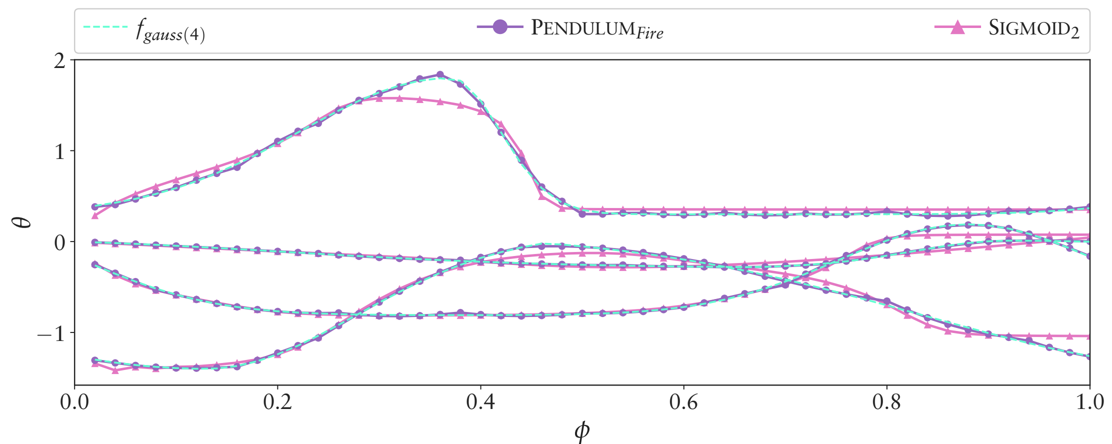


Figure 7.23: Example median results of the optimization of four Gaussian test functions optimized simultaneously.

In this setup four Gaussian test patterns are optimized simultaneously. The patterns are shown in Figure 7.23. This setup is similar to a joint pattern optimization of a single leg. The SIGMOID$_2$ approach is one of the four approaches that does not approximate all four curves well enough. The PENDULUM$_{Fire}$ approach, ranking in the middle, fits good enough again. Since more than one curve has to be approximated, the SIGMOID$_3$ approach is the first time included in the experiment, ranking at the second last position. The only difference to the SIGMOID$_2$ approach is that it learns one network getting all inputs at once. With this configuration it has a probability to cross-link the curves resulting in more complex structures and parameter relations that have to be tuned.

Twenty Waves

Optimizing twenty Gaussian-based test functions at once represents the case of generating joint patterns for an entire robotic system. The patterns are shown in Figure 7.25 together with the median result of the PENDULUM$_{Fire}$ approach. In this experiment only the FOURIER and the GAUSS approaches fit all curves quit close. The median result of the GAUSS approach is shown in Figure 7.26. The PENDULUM$_{Fire}$ approach has problems

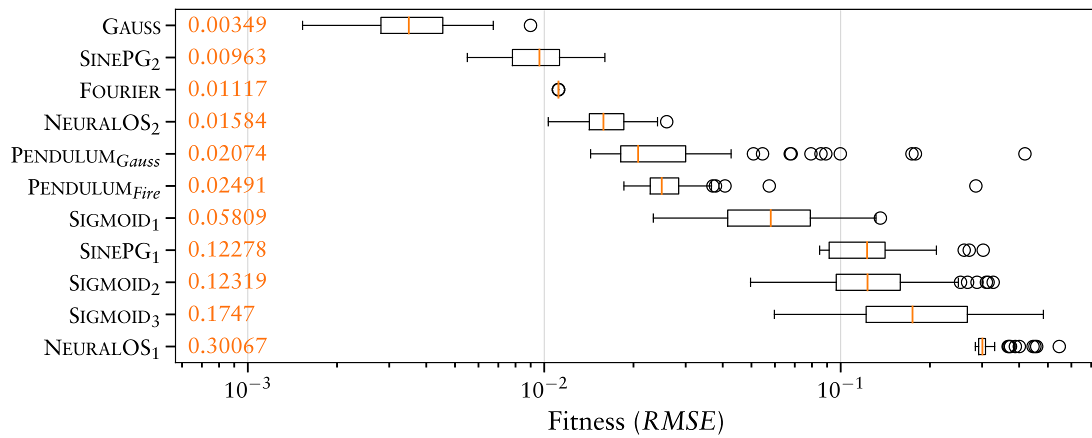


Figure 7.24: Fitness distribution of all approaches on the four Gaussian test function optimized simultaneously.

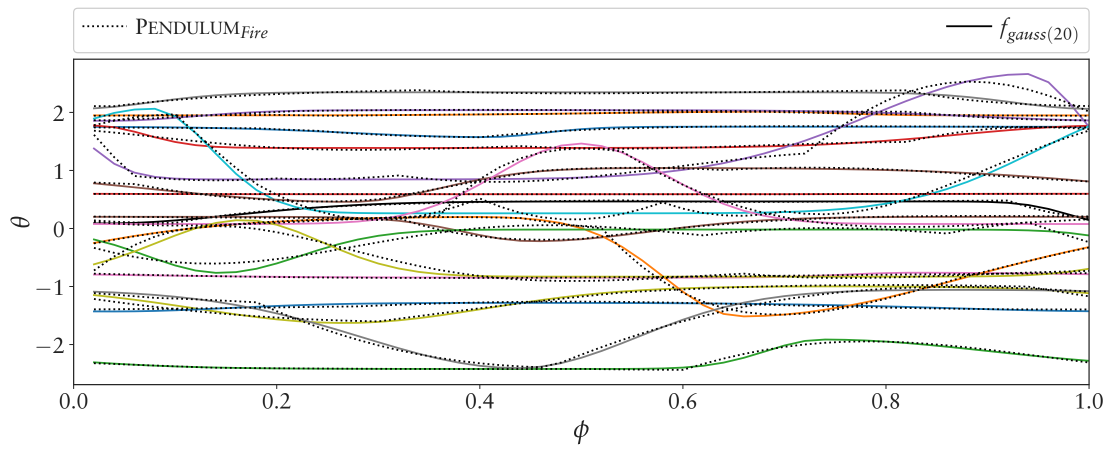


Figure 7.25: Median result of the PENDULUM_{Fire} approach on the twenty Gaussian test functions optimized simultaneously.

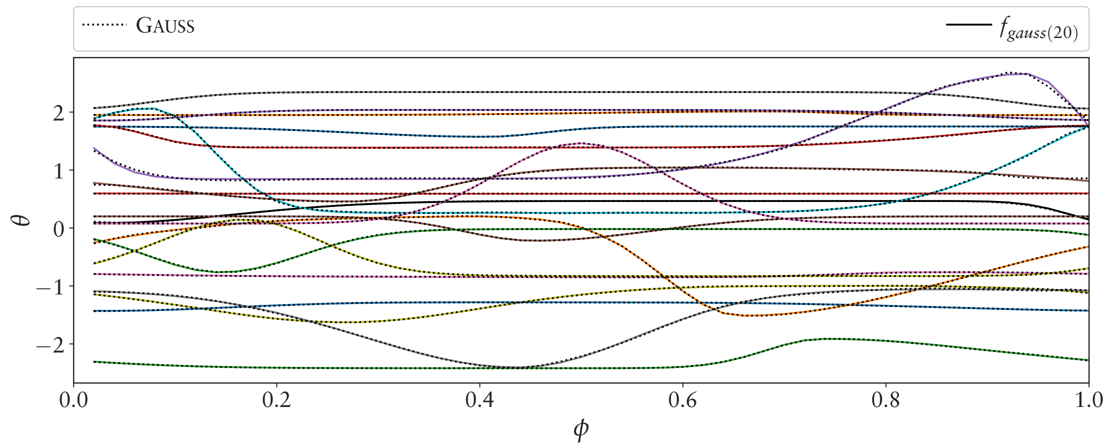


Figure 7.26: Median result of the GAUSS approach on the twenty Gaussian test functions optimized simultaneously.

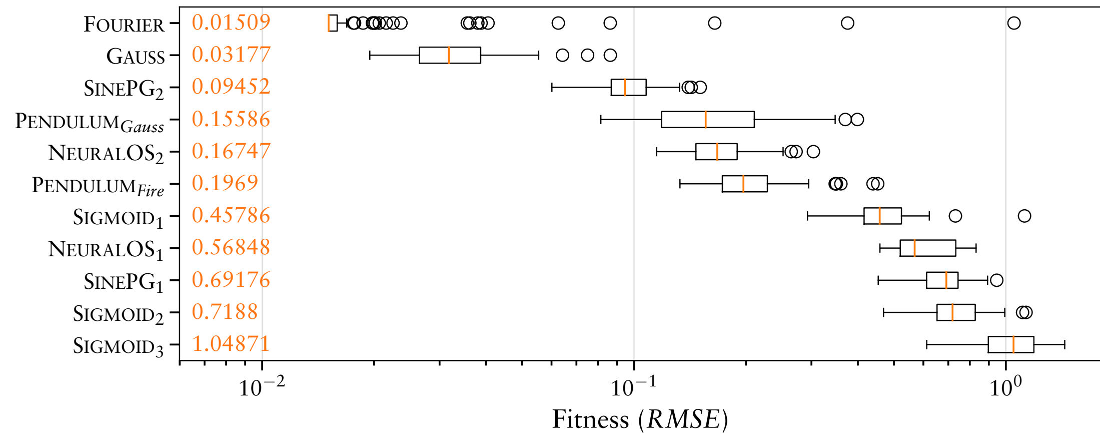


Figure 7.27: Fitness distribution of all approaches on twenty Gaussian test functions optimized simultaneously.

fitting all curves. It cannot fit well to some of the curves included in the twenty test functions. Even when approximating only one of the curves, were the approach does not fit well in the result, does not improve the approximation result. Thus this approximation result is due to a limitation of the representation strength of the Pendulum approach and how it is configured.

7.3.3 Model-Based Joint Pattern

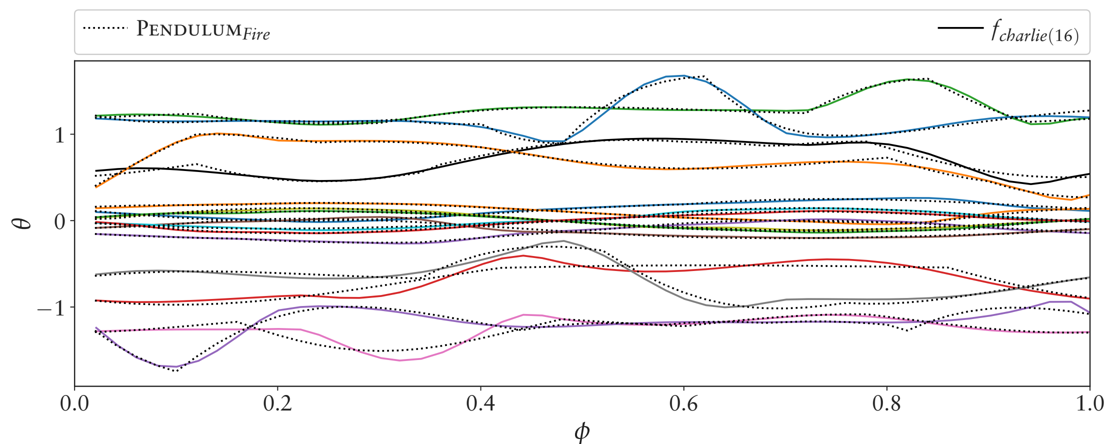


Figure 7.28: Median result of the PENDULUM$_{Fire}$ approach on the sixteen CHARLIE joint patterns.

To test patterns that are used on a real robotic system, this setup uses data from the CHARLIE robot. The data was produced using the model-based CPG approach. Since these patterns produce stable locomotion for an artificial legged system, the joint pattern approaches should be able to represent them.

The result is comparable to the experiment with the twenty Gaussian test functions. The PENDULUM$_{Fire}$ approach cannot represent the shape of two patterns precisely, see Figure 7.28. This representation weakness could limit the search space resulting in an advantage for evolving locomotion patterns as long as joint patterns can be represented that provide a stable walking gait for the target system.

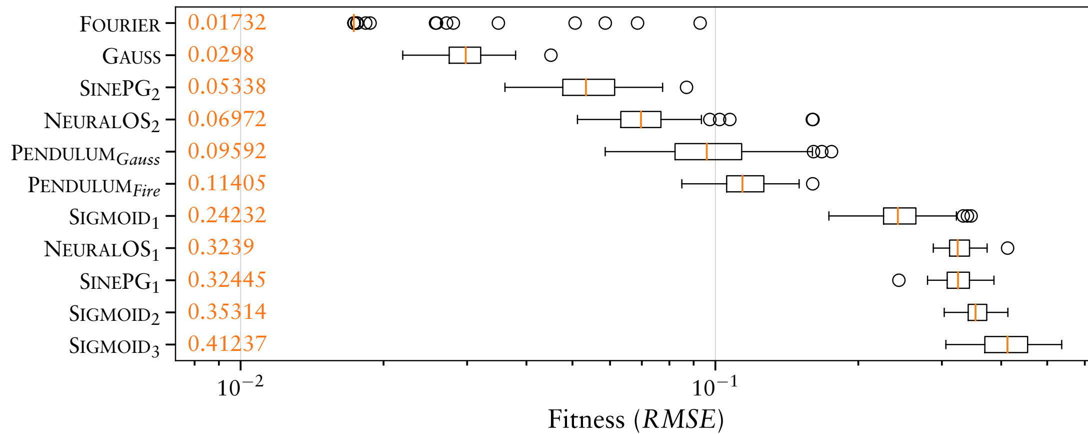


Figure 7.29: Fitness distribution of all approaches on the sixteen CHARLIE joint patterns.

7.3.4 Human Joint Pattern

To compare the performance on a natural created test pattern, the last data set is used from a human locomotion behavior recorded via a motion tracking system. The recorded data is translated into a hip, knee, and ankle angle. It was generated and used for a master thesis presented in Wiedemann (2017). Figure 7.30 depicts the medium results of PENDULUM$_{Fire}$ and FOURIER. Even though the maximal error of PENDULUM$_{Fire}$, with a value of 2.4°, is in a plausible range, e^{max} with a value of 0.074 is above the defined threshold for a precise solution. On one hand, this is caused by the fact that the recorded data has an offset between the first and the last data point which cannot be represented by the periodic pattern representations. On the other hand, the higher relative error is a result of the low amplitude in the pattern. Only the Gaussian approach is able to represent the pattern precise enough according to Equation 7.21. The resulting distributions of all approaches are shown in Figure 7.31.

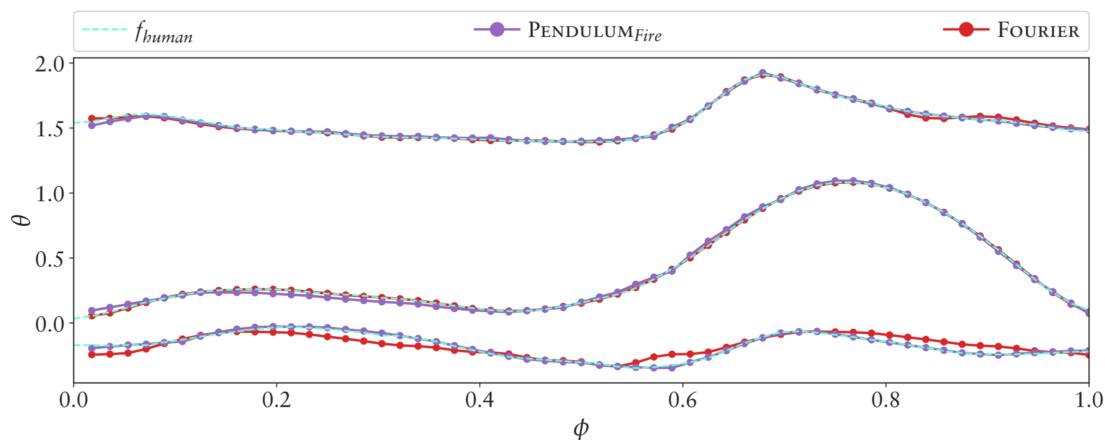


Figure 7.30: Medium results of PENDULUM$_{Fire}$ and FOURIER on a set of joint patterns measured from human locomotion.

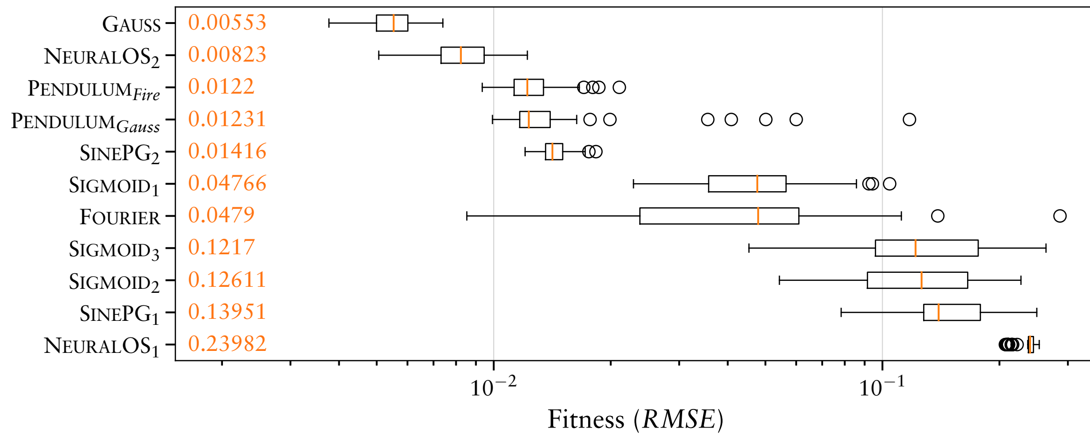


Figure 7.31: Fitness distribution of all approaches on the human locomotion pattern.

7.4 Conclusion

Different pattern representations are compared in respect to their evolvability by the genetic algorithm SABRE. The comparison is done by evolving defined patterns with characteristics as expected to create walking patterns for legged robots. In contrast to the target application of evolving walking patterns for robotic systems, by for instance, evaluating the distance traveled, the fitness landscape resulting by the *RMSE* evaluation of this experiments is expected to be more directed and less fragmented. How the joint pattern representations influence the search space for the target application might be an important factor and cannot be derived from this experiments. The experiments give insights into the representation strength of the approaches and how well they can be tuned if the target pattern is known.

The core Sine-Based pattern approach (SINEPG_1) can generate well tuned joint patterns with a small amount of parameters. However, it limits the possible shape of the patterns and cannot represent more complex joint trajectories as extracted from the model-based CPG control of CHARLIE. Composing multiple Sine-Based patterns (SINEPG_2) eliminates this limitation resulting in a similar concept and performance as the Gaussian approach. In contrast, the Gaussian approach is more specifically designed for this heterodyning concept and thus outperforms the Sine-Based pattern approach in all test experiments.

The Neural Oscillator approach has a similar representation strength as the Sine-Based one. It introduces the most disturbed influence of the parameters to the resulting pattern and thus the two Neural Oscillator configurations perform similar to the Sine-Based patterns ranking in most cases behind it. An exception for this tendency is observed for the human joint pattern where the NEURALOS_2 configuration produces the second best performance.

The Sigmoid ANN approach performs quite well on the benchmarks performed in Section 6.3. However, compared to the other joint pattern representations even for simple patterns it has to generate more complex structures. Additionally, it is the only approach not specifically designed and limited to produce patterns. In fact, the Sigmoid-based ANN should be able to precisely approximate the given joint patterns. Consequently, the resulting optimization performance must be caused by the optimization running into local optima as result of the specific combination of the ANN and SABRE. The approach in its first configuration (SIGMOID_1) ranks in the middle for most experiments. The other two

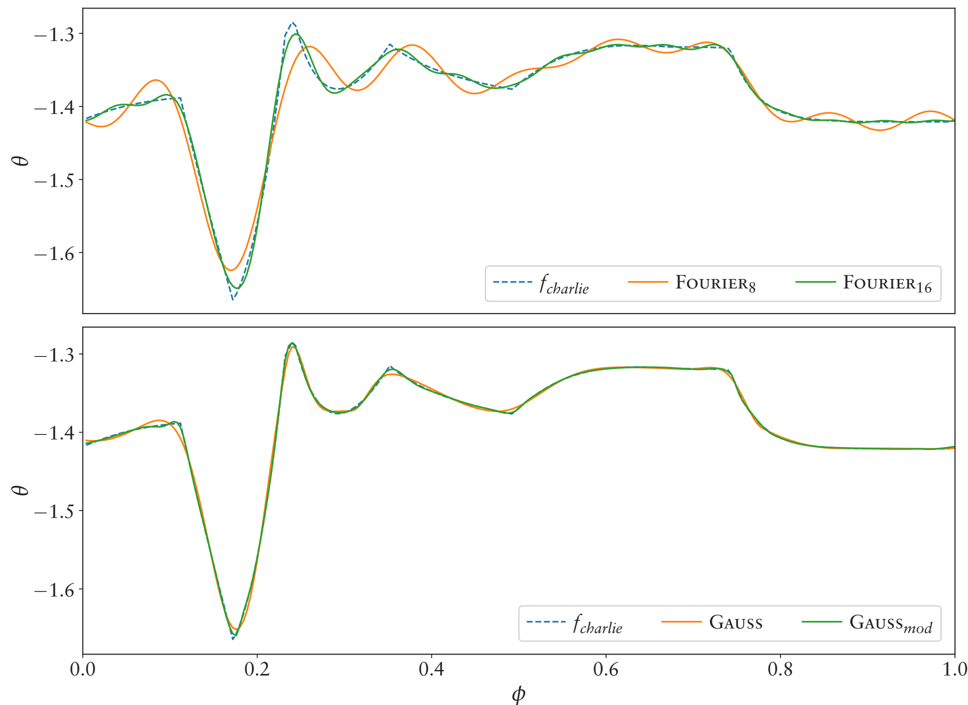


Figure 7.32: Single experiment to compare the Fourier approach with the Gaussian one on more artificial joint patterns.

configurations processing the global phase and the last network output perform worse in all experiments compared to the first configuration. Even though they provide a potential sensor coupling by replacing the last network output with the current motor positions in the target application, it is not clear whether this sensor coupling can compensate for the tendency to run into local optima.

The Pendulum approach does not produce the best results, but it can approximate the most test functions precisely enough to represent a feasible joint pattern generator. The approach is especially designed with the idea to provide a sensor coupling for reactive behaviors as exemplarily presented in Figure 7.16.

The Fourier series approach performs best for many test pattern. However it is limited by the order used as illustrated in an additional example shown in Figure 7.32. For that example a single less smooth joint trajectory of the Charlie is approximated by the Fourier and the Gaussian approach. The approximation is done with the original configuration of both approaches and with an adapted one. Such a pattern is more challenging for the Fourier series approach, even with an order of $N = 16$, while the Gaussian approach performed better in general, especially with a small adaptation of equation 7.13 by multiplying the width parameter o_b with 3.0. How well the Gaussian approach can adapt to the hard direction changes in the test pattern is limited by the minimal possible width of the Gaussian bell. The same is true for the Fourier approach, where a higher order is needed to create harder value changes. Another advantage of the Gaussian approach compared to the Fourier approach might be the parametrization that allows to tune the patterns more punctual, like drawing or tuning a Bezier spline.

Chapter 8

Evolving Legged Locomotion

This chapter presents experiments for evolving the locomotion control of legged robots. The first section deals with a purely simulated robot allowing to "design" the system complexity. With that robot different experiments are performed helping to setup the experiments for the second section presenting results on the real robotic systems SPOT, SpaceClimber, and Charlie. The behaviors for the three robots are evolved in simulation and selected results are evaluated on the real systems.

8.1 Simulated Robot

Within this section simulation experiments are presented to compare selected joint pattern representations introduced in Chapter 7. The four legged test robot "Easy4" is designed with the goal to create a test system that has a relatively low simulation cost while providing enough complexity to allow a comparison of the approaches. The robot is shown in Figure 8.1. With its four degrees of freedom per leg it has a lot of capabilities, while the resulting 16 controllable joints can be calculated rather fast by a simulation. The mass and motor properties are loosely based on values that can be accomplished with a real robotic system. The motors can be controlled by setting a desired position and a PID controller in the simulation generates a motor velocity by taking the maximal torque and motor speed into account. Sensors represent the robot's motor positions, corpus orientation, and feet contact forces. Three successive experiments are performed. The first experiment *"Evolving Open-Loop Control"* is used to generate an open-loop controller with the goal to perform a forward walking behavior. The second experiment *"Navigation Controller"* extends a selected result of the first experiment with a reactive controller, allowing the robot to reach a goal position by modifying the open-loop controller via a robot-to-goal orientation sensor. The last experiment *"Stabilize Controller"* extends the resulting navigation controller to increase the locomotion performance on a rough terrain by processing additional sensor inputs measuring the stability of the system.

8.1.1 Evolving Open-Loop Control

In the first experiment, a selection of the previously introduces control approaches (Chapter 7) is used to generate an open-loop control pattern of the four legged robot Easy4. One selected result of this open-loop optimization is the foundation of the following experiment sections, where the behaviors are further optimized by evolving reactive behaviors processing sensory information.

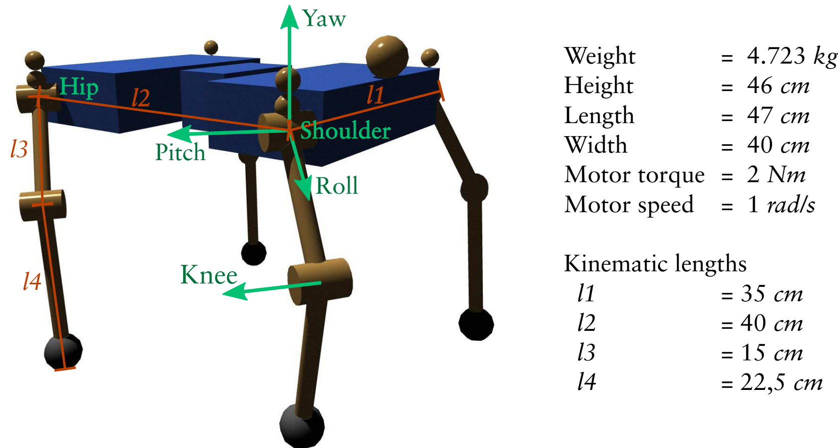


Figure 8.1: Simulation robot EASY4 used to evaluate different joint pattern representations and a reactive control concept.

Environment

The robot has to move on a planar surface with and without the soil contact extension presented in Section 3.4. By using the soil contact extension, the feet contacts are less hard and variance is included, so that the very same pattern does not always produce the same result. Due to the soft and varying contact properties it is expected that the robot has to lift up its feet higher and the evolution generates more stable and general patterns. Moving the feet as low as possible above the ground in the swing phase is energy efficient but would reduce the transferability as a small difference in the model can result in an undesired ground contact of a real system. Without the soil extension, a Coulomb friction value of 0.8 is used in the contact points.

Evaluation

The evaluation function is defined by a BAGEL graph which is connected to the simulation data and the feet slippage criterion measured by `locomotion_environment` (Section 4.2.1). The BAGEL graph defining the additional criteria is shown in Figure 8.2. Two fitness terms ft_1 and ft_2 are implemented in the graph, which are summarized by `locomotion_environment`.

The first term includes the average motor torques ϑ and joint loads ϱ (torques applied to the joints not on the motor axis) divided by the final forward position of the robot p_x. The robot position is limited to a value of $3\,m$. Thus, after the optimization creates behaviors that reach the $3\,m$ mark, it continues to optimize stability instead of producing faster individuals. Furthermore, to prevent to large fitness values or a division by zero, the robot position is cut to positive values starting from $0.01\,m$. To include the starting position at zero meters an offset of $10\,cm$ is added previously to the robot position.

The second term represents the *stability fitness* and includes the standard deviation of the motor torques σ_ϑ, the joint loads σ_ϱ, the roll σ_α and pitch σ_β angle of the robot, the robot's velocity σ_{v_x}, and the height σ_{p_z} of the corpus. The first term represents the general goal to move the robot forward with minimal effort and minimal stress on the mechanics. The second term increases the focus on steady behaviors to reduce the probability for the optimization to converge in local optima with very dynamic, fast and unstable behaviors. This focus is relevant for this work, because the target systems do not include sufficiently compliant components to support high dynamic locomotion behaviors. The two fitness

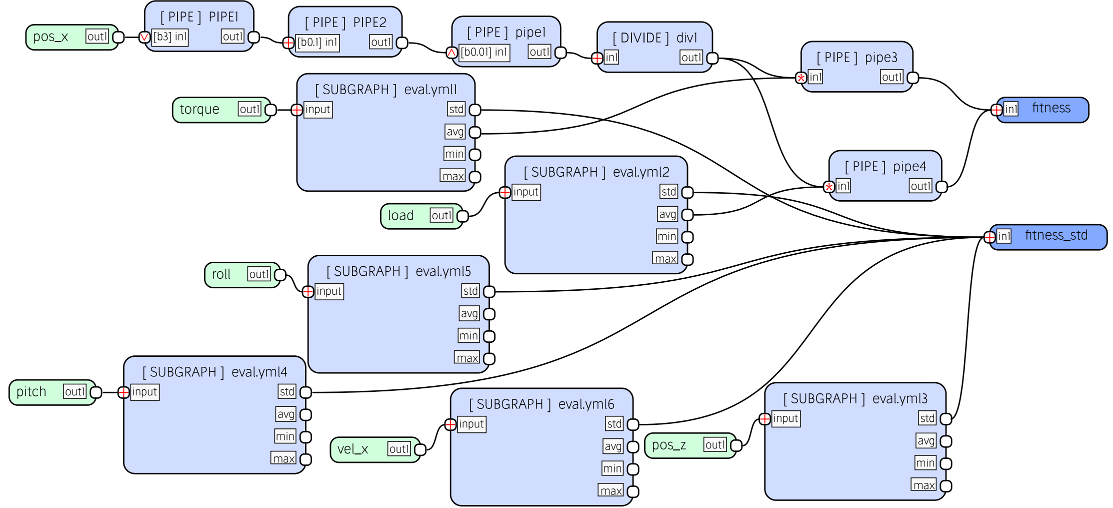


Figure 8.2: Evaluation function for the locomotion benchmark setup defined by a BAGEL graph processing simulation data provided by `data_broker`.

terms are defined by:

$$ft_1 = \begin{cases} \frac{\vartheta+\varrho}{3.1} & \text{if } p_x \geq 3.0 \\ \frac{\vartheta+\varrho}{p_x+0.1} & \text{else if } p_x \geq -0.09 \\ \frac{\vartheta+\varrho}{0.01} & \text{else} \end{cases} \tag{8.1}$$

and

$$ft_2 = \sigma_\vartheta + \sigma_\varrho + \sigma_\alpha + \sigma_\beta + \sigma_{v_x} + \sigma_{p_z}. \tag{8.2}$$

Each generated behavior is tested for 30 s simulation time. The first second is ignored for calculating the fitness value, removing the first acceleration phase of the robot from the stability evaluation. The evaluation is aborted if other parts of the robot than the feet have collisions ($ngc > 0$) – AbortOnContact parameter of `locomotion_environment`. In that case, the fitness is defined by a constant value (100,000,000) subtracted by the simulation time in milliseconds until the collision is detected. The combined evaluation function is given by:

$$f_{ind} = \begin{cases} 100000000 - t, & \text{if } ngc \geq 0 \\ f_s + 0.5(ft_1 + ft_2), & \text{else} \end{cases} \tag{8.3}$$

Configuration

A main control graph, shown in Figure 8.3, is defined which is not changed by the optimization, while the `phase_shift`, `front_leg`, and `rear_leg` sub-graphs are, depending on the used approach, generated or optimized by SABRE. To generate the leg phases the concept of combining a global `phase` with one `pulse` and one `slope` module per leg is used as introduced in Chapter 7. The *control cycle* (global phase) is configured to a time period of 2 s. The `phase_shift` sub-graph includes three unconnected outputs and no structure elements are allowed to be added. Eventually, only the `bias` parameter of the outputs are optimized which define the phase shift of the front right and rear legs relative to the front left leg. As presented in Section 7.2 for the Fourier and Pendulum approaches, the inner structure of the leg's sub-graphs is predefined, while the Gaussian

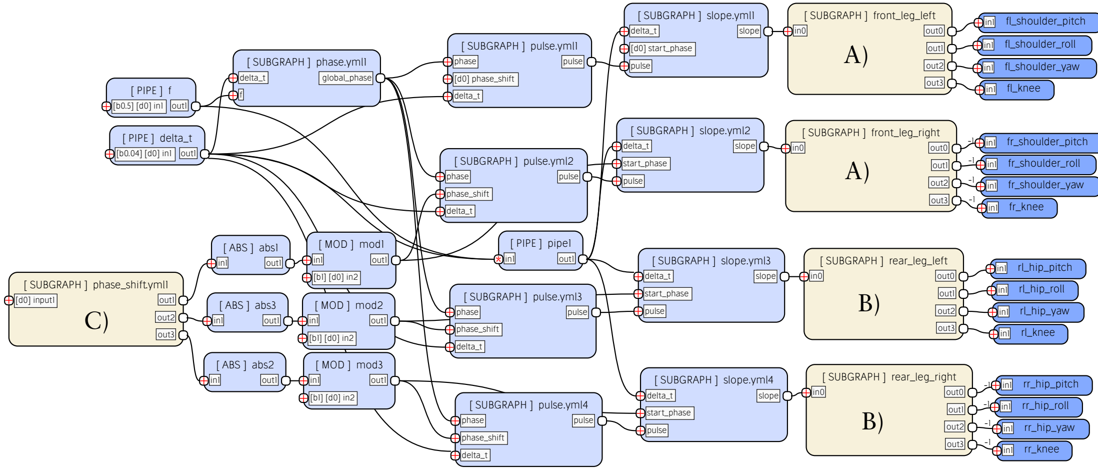


Figure 8.3: EASY4 main control architecture for open-loop locomotion. The sub-graphs A), B), and C) are evolved by SABRE, while this main architecture is not touched by the optimization. Sub-graph A) generates the patterns for the front legs, B) for the rear legs, and sub-graph C) generates the phase shift for the right front, rear left, and rear right leg.

approach directly evolves the sub-graphs by composing Gaussian modules. Figure 8.4 depicts the structure used for the leg control by the Fourier series approach. In this setup only the parameters for the single Fourier joint pattern are optimized. Figure 8.5 depicts

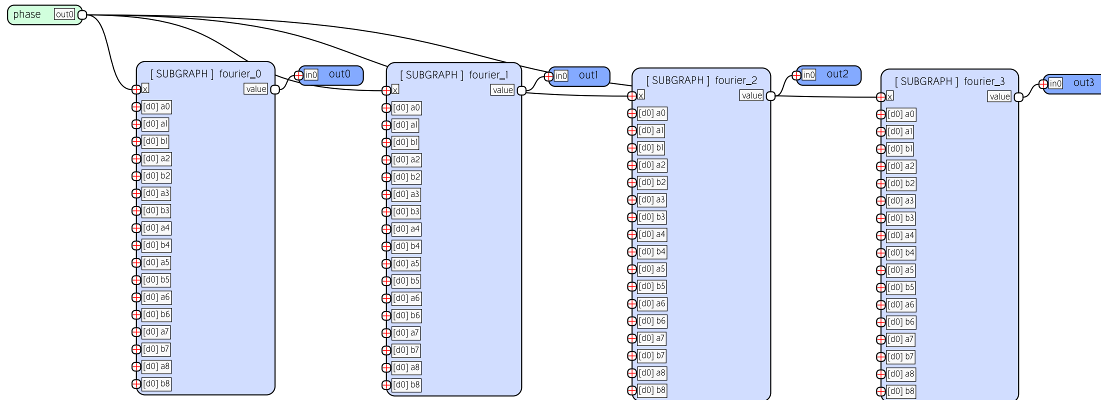


Figure 8.4: Predefined leg control structure for the Fourier series approach. In this configuration only parameter optimization for the inputs of the Fourier nodes is performed.

the control structure of the legs for the Pendulum approach. `fire_network` generates the pulse pattern and the parameters (damping, frequency, amplitude, and offset) of the pendulum module for the four outputs. The `fire_network` sub-graph is generated by SABRE. For every joint one output node is used to generate the pulse pattern and four output nodes (*parameter outputs*) are used to define the parameters of `pendel_control`. The parameter outputs are excluded from the structural development and thus represent constant parameters of the graph that are optimized by SABRE.

For all approaches two configurations are created, one that optimizes all 16 joint patterns independently and one that generates one sub-graph (*leg control graph*) for the front leg pair and one for the rear leg pair. The later configuration reduces the search space by enforcing symmetric patterns of the left and right legs. For every leg one instance of the corresponding leg control graph is generated transferring the individual leg phase to the joint trajectories. Since the motor directions are inverted between the sides of the

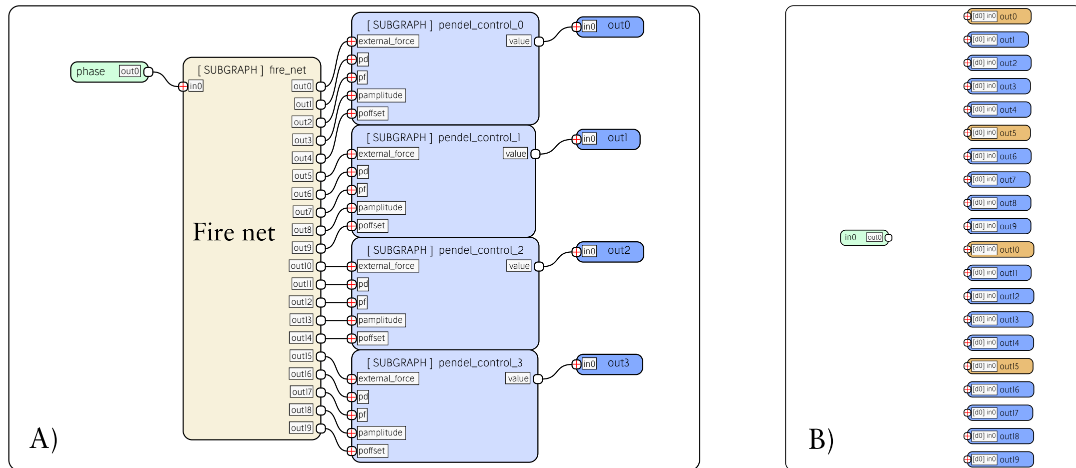


Figure 8.5: A) Predefined leg control structure for the Pendulum approach. The structure and parameters of this graph are fixed while the fire net is evolved. B) The configuration of the fire net. Only the orange output nodes can be connected by edges, while the other nodes define parameters for the pendulum controller of the higher layer.

robot, the weights for the right legs in the main control graph are set to -1, allowing the same output of the leg control graphs for both sides. Together with the soil and hard ground contact simulation each approach is processed in four resulting configurations of the setup ($\text{config}_{soil/sym}$, $\text{config}_{hard/sym}$, $\text{config}_{soil/full}$, and $\text{config}_{hard/full}$).

Preliminary optimizations have shown, that the Pendulum pattern approach with the same configuration as in the previous chapter is not able to generate locomotion patterns. The main reason is the small mutation step size. In the experiments of Chapter 7, this small adaptation of a single pattern already produces a fitness improvement that guides the evolution in the right direction. In the locomotion setup, a fitness improvement for the oscillation is only reached by generating a pattern on multiple joints that together produce a continuous forward movement. If, for instance, the shape of the patterns is well suited but the amplitude is too small, the feet do not lift up the ground and no forward movement can be produced. By adapting the overall mutation step size, the evolution produces patterns that provide a forward movement, but it starts with a lot of obstructive postures because the joint offsets are too big and the posture is varying nearly randomly instead of slowly adapting to a beneficial posture. Due to this observation, the mutation step sizes of the individual properties of the patterns are empirically chosen to fit to the locomotion setup. This is done by monitoring single optimizations and adapting the step sizes until the adaption process looks feasible. The same procedure is performed for the other control approaches, too.

Results

Table 8.1 includes the results of 50 evolutions performed for every combination of configuration and approach. The best results – lowest fitness values – in two of the four configurations are generated by the Pendulum patterns. The best median results vary between the different configurations. The feet and body heights are measured in world coordinates while executing the resulting behaviors. In case of the configurations with the soil model the feet dig about 3 *cm* into the surface, which has to be taken into account when interpreting the feet heights measured. By adding this offset, feet are lifted about 2 *cm* higher in the configurations with the soil model.

Table 8.1: Results of the open-loop experiment for EASY4 sorted by the configurations.

Config	Approach	Median fitness (best)	⊘ Step height (σ)	⊘ Body height (σ)
$\text{config}_{soil/sym}$	Fourier	4.54 (3.69)	1.5cm (0.4cm)	33.9cm (1.6cm)
	Pendulum	4.79 (3.19)	1.5cm (0.1cm)	32.8cm (2.8cm)
	Gaussian	5.07 (3.97)	1.7cm (1.5cm)	31.1cm (3.2cm)
$\text{config}_{hard/sym}$	Fourier	5.1 (2.67)	2.4cm (0.3cm)	34.3cm (2.1cm)
	Pendulum	3.91 (2.88)	2.5cm (0.3cm)	37.1cm (0.25cm)
	Gaussian	4.54 (2.88)	2.7cm (1.1cm)	33.0cm (3.4cm)
$\text{config}_{soil/full}$	Fourier	5.43 (4.45)	1.4cm (0.5cm)	34.2cm (1.0cm)
	Pendulum	6.7 (5.49)	1.4cm (0.5cm)	35.5cm (0.4cm)
	Gaussian	5.45 (4.24)	1.6cm (1.2cm)	33.6cm (2.9cm)
$\text{config}_{hard/full}$	Fourier	7.29 (4.94)	2.8cm (1.6cm)	35.4cm (1.3cm)
	Pendulum	6.32 (3.58)	2.5cm (0.3cm)	37.3cm (0.2cm)
	Gaussian	5.49 (4.03)	2.8cm (1.7cm)	32.1cm (4.1cm)

The fitness distribution between the approaches in the different configurations are significantly different (Mann–Whitney U-test $p < 5\%$) in most cases. Only two pairs of results are not significantly different. The first pair are the results of the Gaussian and Pendulum pattern in the first configuration and the second are the results of the Gaussian pattern and the Fourier series in the third configuration.

The calculated fitness represents the performance in the simulation setup and cannot be used as a metric for transferability to a real system. Especially behaviors with high frequency patterns can produce good fitness values, but on a real system would consume a lot of energy or fail due to a different contact behavior. This difference is known as simulation-reality gap, as introduced in Section 1.1. Reducing this gap by improving the simulation model is possible but can easily result in high additional computation load. To prevent this kind of behavior, either the evaluation function can be extended or the behavior representation can inhibit them or decrease the probability that they are evolved. Extending the evaluation function can result in an undesired tuning of evaluation criteria. In the worst case the evolution could tend to go into the direction of only one criterion, while changing the influences does not mix them but at some point switches the focus of the evolution to another criterion. The more opposed the criteria of the evaluation function are the more likely do such problems occur.

The Fourier series approach has a high probability to generate high frequency patterns at the beginning of the evolution and finally tends to evolve these behaviors. Figure 8.6 depicts the best result of the approach in the symmetric configuration combined with the soil model. The result represents a typical behavior generated by the Fourier approach for this experiment.

In case of Gaussian patterns the results vary to a large degree. The best solution is shown in Figure 8.7. The solution have a single stance phase within the defined control cycle, which does not represent the general results produced with the Gaussian patterns. However, the high movement of the body is presented in most of the results and is only lower if the step frequencies are higher. Most result have two or three stance phases within the control cycle, while some few results are similar to most results of the Fourier series approach producing higher frequencies.

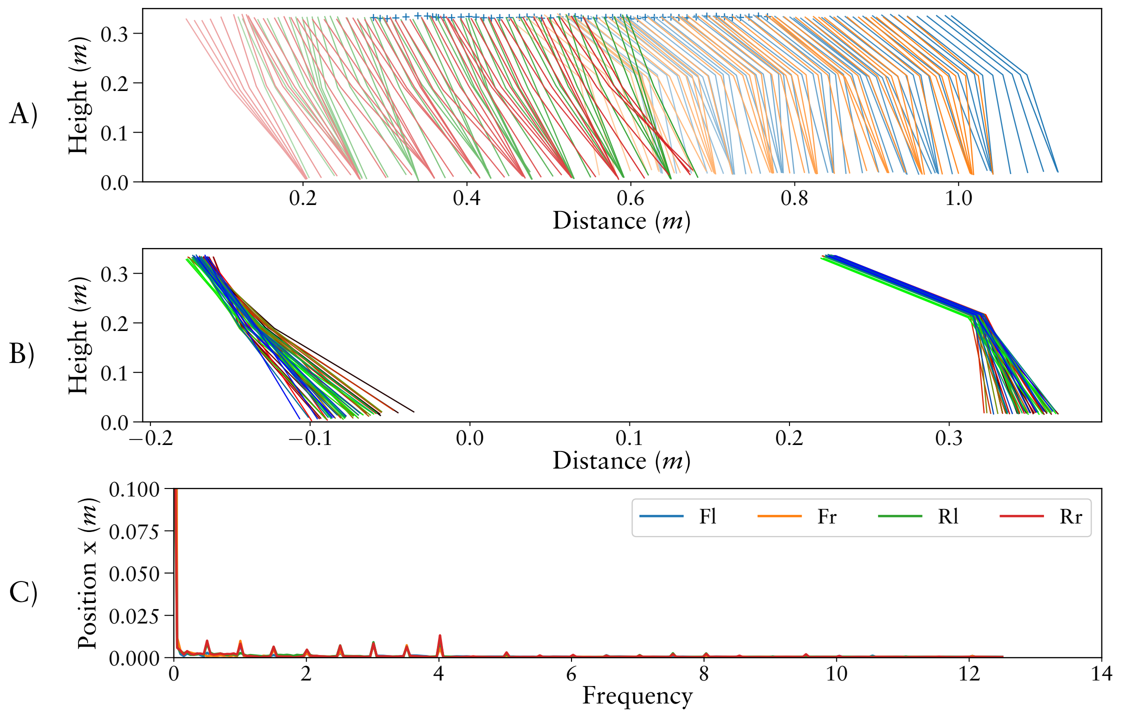


Figure 8.6: Best behavior generated with the Fourier series approach for the first configuration config$_{soil/sym}$. Graph A) shows the corpus center position (blue marker) and the legs from the side view performing one control cycle of 2 s. The legs are separated by the colors (front left = blue, front right = orange, rear left = green, rear right = red). Graph B) plots the left front leg and left rear leg in a side view relative to the corpus center position. The color gradient (red-green-blue) represents the state within the control cycle. Graph C) depicts the Fourier transformation of the feet positions on the forward axis representing the step frequencies (Fl = front left, Fr = front right, Rl = rear left, Rr = rear right). A peak at 0.5 Hz correlates to one step (forward-backward movement) per control cycle, while a peak at 4 Hz relates to 8 single steps within the control cycle. In the used setup, the Fourier approach has a clear tendency to evolve tiny steps with high frequencies.

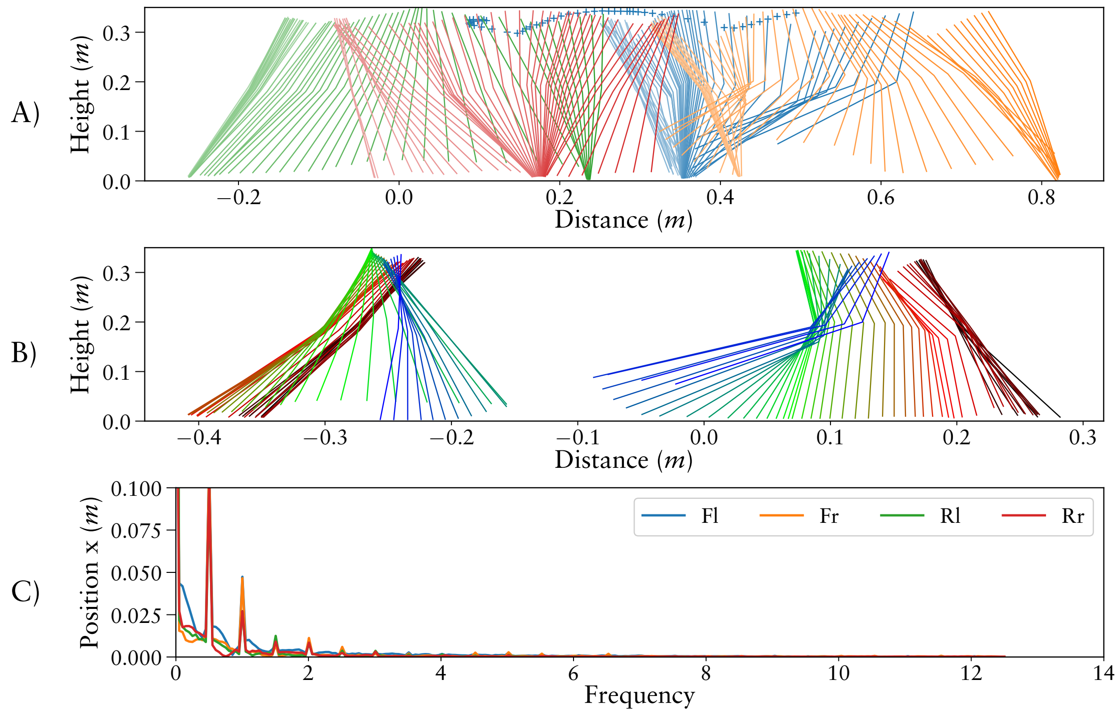


Figure 8.7: Best behavior generated with the Gaussian pattern approach for config$_{soil/sym}$. Compared to the other approaches the used Gaussian configuration produces the most dynamic behaviors concerning the movement of the robot's corpus.

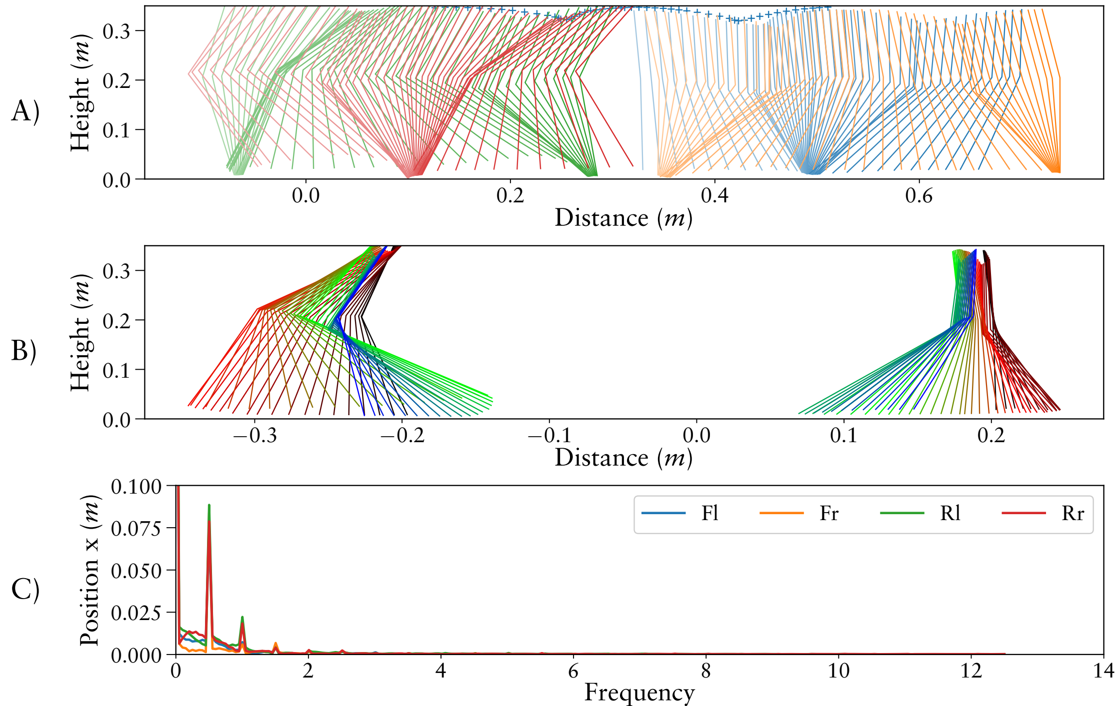


Figure 8.8: Best behavior generated with the Pendulum pattern approach for config$_{soil/sym}$. The used Pendulum pattern configuration has a clear tendency to produce pattern with a single stance phase within the control cycle. The resulting behaviors are more steady compared to the Gaussian results.

The best result of the Pendulum pattern approach is shown in Figure 8.8. The result has one stance phase within the control cycle similar to the best result produces with the Gaussian pattern. It is also the best result over all optimizations performed for the setups including the soil model. Compared with the best result of the Gaussian approach, the body is more stable which can also be seen in the shoulder and hip position in plot B) showing the left front and rear leg movements relative to the center position of the robot's corpus.

Table 8.2 lists the number of results sorted by the number of ground contacts within the defined control cycle time for the symmetric configuration with soil simulation. All results of the Fourier series generate more than two contacts. Most of the results of the Gaussian pattern approach generate two stance phases within the control cycle time. The most results with only one contact are produced by the Pendulum approach.

Table 8.2: Number of results per approach sorted by number of ground contacts in the control cycle time for the experiment with the soil model and symmetric left and right leg patterns.

Approach	One contact	Two contacts	More then two contacts
Fourier			50
Pendulum	30	19	1
Gaussian	11	30	9

Even though the Pendulum approach has some limitations regarding the representation strength of joint patterns, as discussed in the Chapter 7, it produces superior results in this experiment. On one hand, the pendulum controller seems to limit and form the search space in a way that more desirable patterns are produced. On the other hand, this also limits the representable shapes of the patterns and might does not allow enough fine tuning for a complex robotic system.

8.1.2 Navigation Controller

The previous section focuses on generating open-loop walking patterns. This experiment is designed to learn a modulation of the previously generated open-loop patterns by processing a direction input. The Pendulum approach produces the most desirable open-loop patterns and it is designed to allow the modulation of the patterns by individual sensor driven impulses. Therefore, an empirically selected result produced with the Pendulum approach is selected as open-loop behavior for this section. To verify that the finally evolved controller can generally be used for a navigation task, the controller is eventually evaluated in an additional experiment where the robot has to follow a defined path. The experiment of this section is designed to prove that, based on the Pendulum approach, a sensory driven modulation of the open-loop pattern can be evolved to control the robot's walking direction. Therefore, a single result is generated proving this statement.

Environment

To evolve a behavior that enables the robot to reach a defined target position, a simulation setup is designed with three start positions and one goal position, see Figure 8.9. The robot's initial orientation is the same in all start positions. Thus, depending on the start position, the robot has to move, to the right, straight ahead, or to the left to reach the goal. A MARS plugin is developed calculating the angle between a robot's orientation and the goal position. The plugin resets the robot to the next starting position after 20 s

simulation from each position. Altogether, each individual is tested for a time period of 60 s.

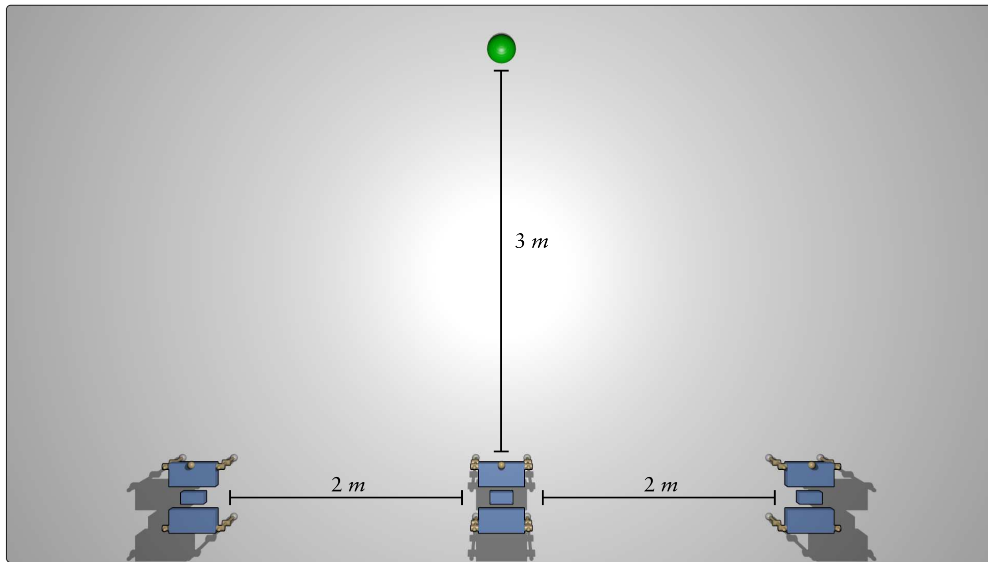


Figure 8.9: Simulation setup of the navigation experiment for EASY4.

Evaluation

For the evaluation of the tested behavior, the distance to the goal position is the main criterion of the fitness function. Since the objective is to reach the target position, the distance measurement is stopped if the robot gets closer than 0.25 m to the goal position. This allows the robot to pass the target position without decreasing the fitness. As result, the three resulting distances are summarized for the final fitness value. A constant value $c_n := 1000$ is added to the fitness if the robot does not reach the target positions from all three start positions. Otherwise, the stability is evaluated as long as the stability fitness (8.1.1) is lower than c_n. The stability term is the same as in the open-loop control experiment, see Equation 8.2.

Configuration

An empirically selected open-loop result of the Pendulum approach is chosen from the previous experiment, providing a stable locomotion behavior as starting point for this experiment. The control architecture of the open-loop controller is extended with the goal to process a direction command without changing the underlying open-loop patterns if no command is given. Additionally, the extension should allow to slowly adapt the resulting behavior through the optimization process of SABRE. To fulfill these requirements three modules that process the direction command are designed (see Figure. 8.10). Two of the modules receive the direction command and the global phase as input to generate a pulse output for the leg control sub-graphs. The latter are extended by piping the additional pulse inputs to the `external_force` inputs of the `pendel_control` graphs (see to Figure 8.5). One module is evolved for the front and one module for the rear legs. The third module only receives the direction command and can modulate the leg phase frequencies. For instance the step frequencies on one side of the robot could be increased to produce a direction change of the robot. Therefore, the global phase module is extended providing

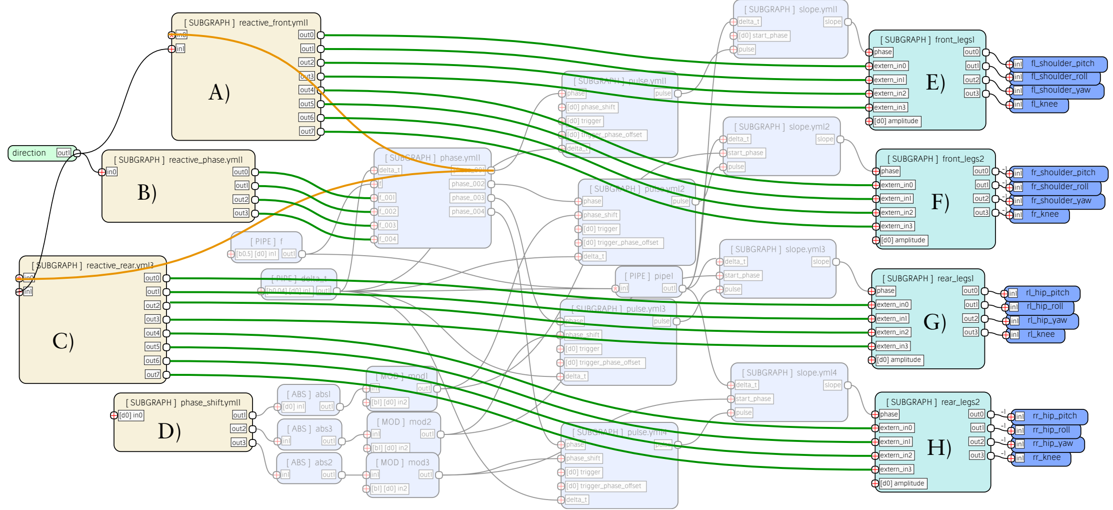


Figure 8.10: Controller extension for the navigation experiment. The nodes E) to H) are the leg control graphs evolved in the open-loop experiment. They are extended by providing an additional input for every joint. These inputs are added to the external force inputs of the `pendel_control` graphs. The nodes A) to D) are new nodes which inner structure is evolved. The nodes A) and C) generate a modulation force for the leg control graphs and process the global phase and the direction command. Node A) is connected to both front leg control graphs and C) to the rear leg control graphs. Node B) processes only the direction command and can modulate the phase frequencies of the individual legs. Therefore, the global phase module is extended to process the modulation signals of node B). Additionally, node D) can be adapted to optimize the phase shift of the legs for this experiment setup.

a separated phase for each leg. Four new inputs can modulate the phase frequencies, while the phases are aligned by a phase coupling. Without an input signal the phases are identical. By applying an input signal the phase frequencies are adapted, which also generates an additional phase shift between the legs. If the input signal is removed, the phases fade back to an identical phase. The previously introduced `phase_shift` module is applied after the processing of the phase frequencies and still defines the default phase shift between the legs. The phase coupling is done by the following equation:

$$\dot{\phi}_j := 0.01 \sum_{i=1}^{i<N} \sin(2\pi(\phi_i - \phi_j)), \tag{8.4}$$

with N representing the number of legs. Due to a possibly asymmetric phase shift the evolved open-loop controller might prefer walking curves in one direction, while walking curves into the other direction might need more complicated modulations of the joint patterns. To compensate for this possible preference, the phase shift is part of the optimization in this experiment, too.

Similar to the fire neuron introduced to generate the input pulse pattern for the pendulum controller (Section 7.2.3), a `fire2_neuron` is implemented to process one phase input and one threshold input. Altogether, the module has four inputs. The first and second inputs are treated just as the two inputs of the firing neuron. The first input is connected to the phase input of the reactive module and together with the bias of the second input defines the first condition for the neuron to become active. The third input is connected to the second input of the reactive module, where the target direction command is connected on the highest control layer. The last input defines a threshold

for the third input and thus the second condition. For the threshold test, the absolute value of the bias is used. If both conditions are fulfilled, the neuron produces an output of 1. With this configuration, the reactive module can only influence the open-loop pattern if an input signal is given for the direction command. The two reactive sub-graphs connected to the leg controllers can evolve a network of only `fire2_neurons`. For the reactive module influencing the phase frequencies, a similar implementation is done using only *threshold neurons* (`threshold_neuron`). The threshold neurons process two inputs similar to the third and forth input of `fire2_neuron`. The neurons are inactive without an input signal, as well.

Result

The design process of the setup finally used includes multiple individual evolutions with variations of starting positions, goal position, fitness calculations and wiring of the reactive modules. Without the limitation to purely reactive extensions many results of the test evolutions change the open-loop behavior to always walk a curve into one direction and walk into the other depending on the sensor input. With this kind of solutions the evolved controller does not walk straight forward without a direction command and it reacts only to commands for one direction. This is a valid solution for the setup but not necessarily a preferred behavior for a real system.

Finally, a single evolution is performed using the parallel distribution of BOLERO producing a navigation controller. The evolution is started without an evaluation limit and is manually aborted after 355.000 evaluations with a resulting controller providing the features in question. After 8.040 evaluations the generated controller reaches the goal from all three start positions for the first time. From that point on, the evolution is optimizing the stability of the walking pattern. The trajectory of the final evolved controller is shown in Figure 8.11.

Figure 8.12 depicts how the controller influences the open-loop pattern. Depending on the direction command it produces additional pulse inputs modulating the joint pattern. The evolved controller operates on two joints of the front left and rear right leg. The pitch and yaw joint of the the front left shoulder and the roll joint of the rear right hip are adapted to walk a left turn. Only the rear right knee is used to turn right.

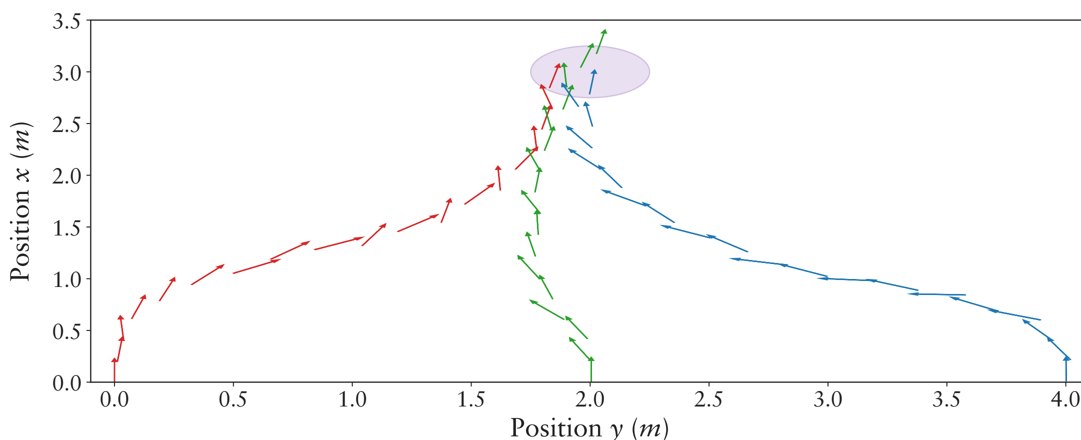


Figure 8.11: The resulting path taken by the evolved controller navigating to the target position from the three defined start positions. The target area, that have to be reached, is marked by the filled circle. The arrows plot the position and direction of the robot. For each starting position a different color of the arrows is used.

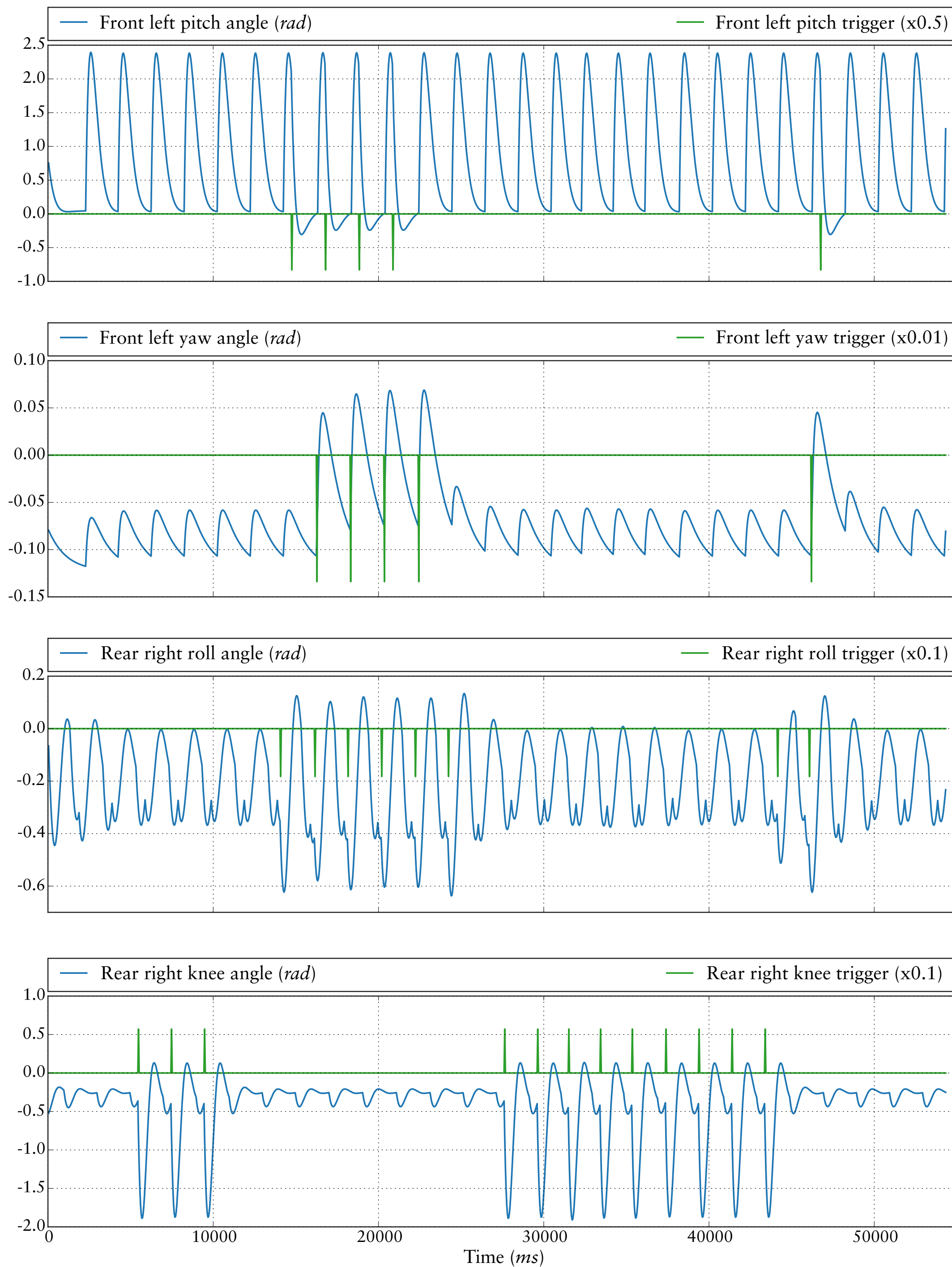


Figure 8.12: The plots depict how the joint patterns are modified by the reactive modules. Each plot shows the signal generated by the reactive module and the influence of the signal on the joint pattern.

To test the evolved controller in a navigation application, a second setup is designed, where the controller has to follow a defined path. The setup is shown in Figure 8.13 and the resulting behavior of the controller is shown in Figure 8.14. The controller manages to follow the path with a maximal distance of 70 *cm* to the path. The controller steers more intensely the left side then to the right resulting in a position error in the middle part of the path. At the end of the path the error is compensated. The result of this experiment proves the potential to evolve a modulation of the open-loop pattern that allows the robot to steer to a given target direction. However, the result produced is limited concerning the shape of possible paths the robot can follow. This can mainly be explained by the rather simple setup used to evolve the navigation controller. How this setup can be enriched or optimized to evolve controllers that can follow a given path more precisely is an open question.

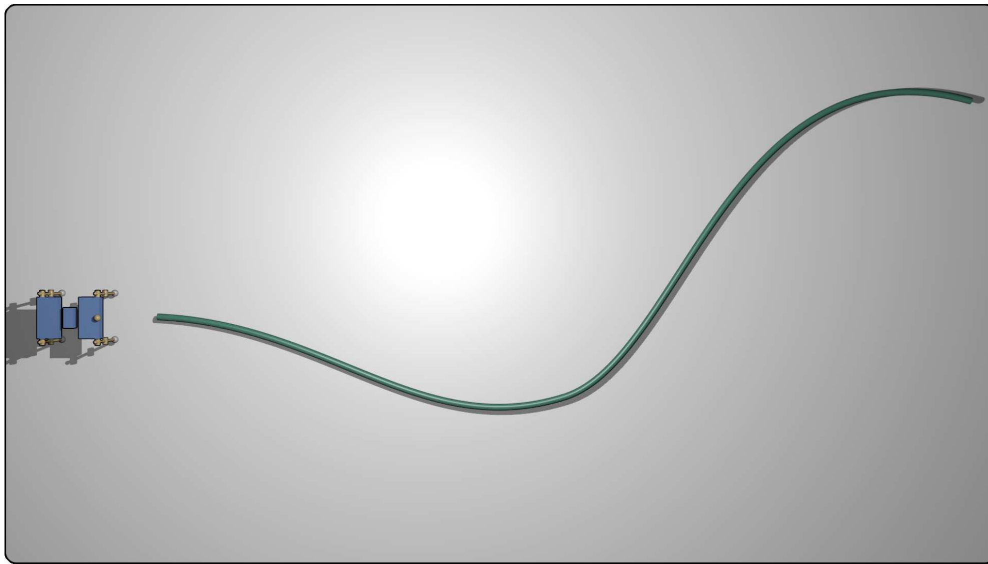


Figure 8.13: A test setup to evaluate the evolved navigation controller of EASY4.

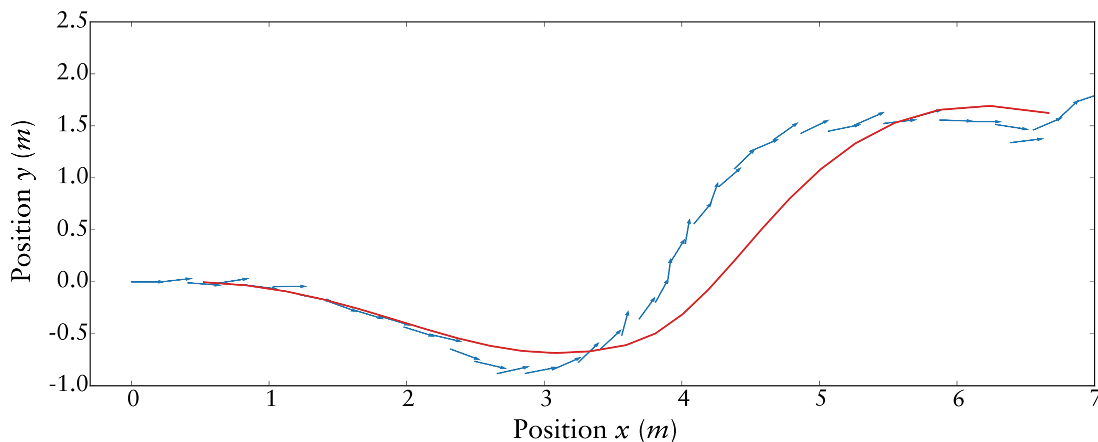


Figure 8.14: The result of the evolved navigation controller of EASY4 used to follow a defined path.

8.1.3 Stabilize Controller

In the previous section it is shown that a modulation of the open-loop pattern can be used to adapt the robot's walking direction. The next and last step in this series of experiments is to evolve a modulation that stabilizes the robot in an unstructured environment. As for the navigation experiment, the objective of this experiment is to show that a modulation can stabilize the robot and provide an adaptation to an environment. Therefore, the resulting controller of the navigation experiment is further extended in this section. Individual evolutions are performed to design the final setup and fitness function. Finally a single result is generated and is compared to the performance of the navigation controller in the rough environment designed for this experiment.

Environment

The simulation setup is shown in Figure 8.15. The environment becomes more challenging towards the edges, which rewards the usage of the navigation controller keeping the robot in the track's center. The boxes at the track's end introduce obstacles with sharp contours increasing the environment complexity once the evolved controller manages to operate on the hilly ground. Also the boxes height is increasing towards the borders allowing the robot to pass the obstacles only in the middle of the track.

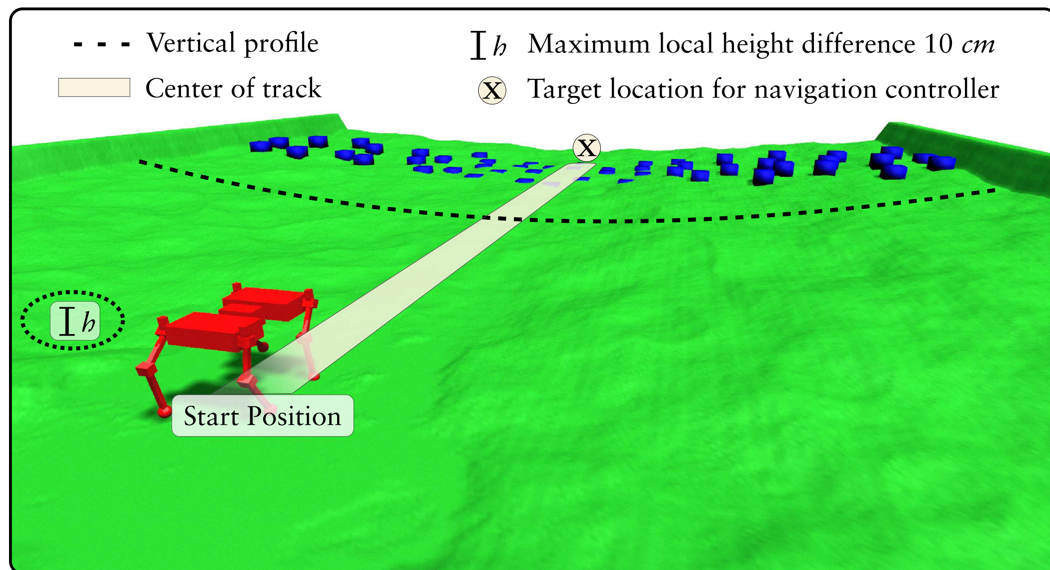


Figure 8.15: Simulation setup to evolve a stabilizing controller for EASY4. The previous evolved behavior provides an open-loop gait for traversing on a planar surface and a navigation controller allowing the robot to reach a target position. The surface roughness and slope increases to the sides of the environment to increase the fitness of behaviors walking straight ahead through the track. The navigation controller enables the robot to steer back to the middle of the track if the robot becomes misaligned due to collisions with the surface. The robot is visualized in its physical collision representation.

Evaluation

Each behavior is evaluated for a period of 60 s. To generate more generic solutions the lateral start position of the robot is varied randomly in a range of 0.6 cm by $U(-0.3, 0.3)$. The fitness function is defined in Equation 8.5. In this challenging environment the needed

energy and the mechanical stress are not taken into account for the fitness of a behavior. The evaluation of a behavior is aborted on contact of other elements than the feet ($ngc \geq 0$) similar to the previous experiments. The function combines the reached position of the behavior in the environment (x and y), the feet slippage percentage f_s as explained in 4.2.1, and in case of a collision abort, the time the behavior was operating. Thus the main focus of this evaluation is to enable the robot to operate in the environment.

$$f_{ind} = \begin{cases} 100000000 - t, & \text{if } ngc \geq 0 \\ \frac{y}{x^2} + f_s, & \text{else} \end{cases} \tag{8.5}$$

(8.6)

Configuration

The previous evolved navigation controller is extended similar to the navigation experiment (Section 8.1.2). Four new modules are introduced processing the robots z forces at the feet and the *roll* and *pitch* rotation of the robots body. The top level control architecture is shown in Figure 8.16 was really modified by this setup? If so describe in caption}. For the development of the new modules modifying the leg patterns, the evolution is configured to compose networks of `fire2_neuron` modules as used in the navigation experiment. The new module modulating the phase frequencies is evolved by composing a network of `threshold_neuron` modules.

Result

A single experiment result is generated and evaluated by a comparison with the initial navigation controller that is used as the basis for the evolved stabilize controller. For this comparison, 1000 individual evaluations of both controllers are performed in the simulation setup – including the random start position. The result of the comparison is shown in Figure 8.17. All test evaluations done with the pure navigation controller are aborted due to unintended collisions – the robot falls over. In case of the evolved stabilize controller 26.9% passed the evaluation without a collision abortion. The minimum, maximum, and median distances reached by the two controllers are shown in Table 8.3. For the failed and passed trials of the stabilize controller the median distances reached are above 4 m and the maximal distance is above 6 m. The median distance of the initial navigation controller is at 1.49 m, while the best trial reaches 4.32 m until the robot falls over. Altogether, the evolved controller successfully improves the navigation controller and increases the performance in the rough environment.

Table 8.3: Statistical results of the comparison of EASY4's navigation and stabilize controller. The navigation controller (start point for the evolution) and the stabilize controller are tested in 1000 trials in the rough environment. The results are separated by successful and failed trials – if the robot falls over within the evaluation time it fails the test.

Evaluation		*X*-axis			*Y*-axis	
	Max	Min	Median	Max	Min	Median
Initial failed	4.32 m	0.26 m	1.49 m	1.07 m	-0.56 m	0.12 m
Learned failed	6.35 m	0.41 m	4.1 m	0.8 m	-1.18 m	-0.17 m
Learned passed	6.01 m	3.02 m	4.75 m	0.54 m	-2.2 m	-0.52 m

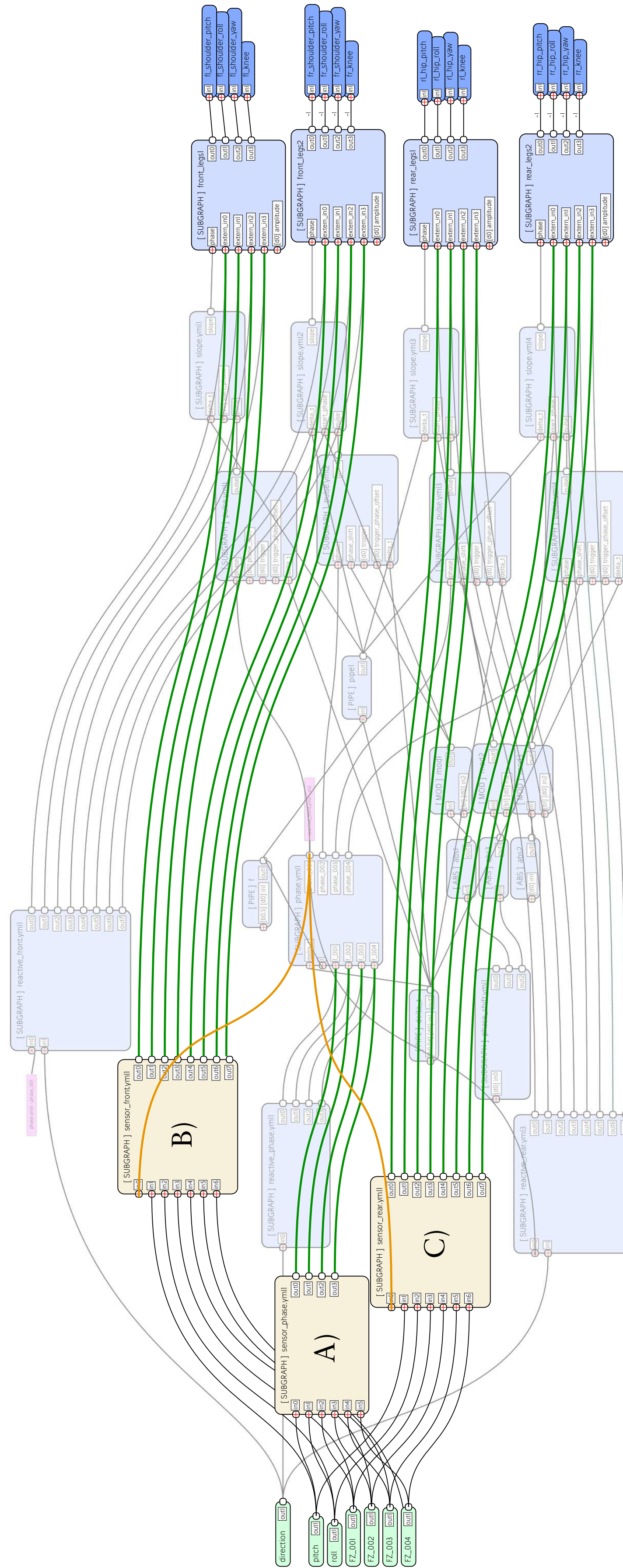


Figure 8.16: The control architecture of the navigation controller extended by the reactive modules (A), B), C)) processing the sensory data to stabilize the robot. Sub-graph A) modifies the phase frequencies, while B), and C) influences the open-loop pattern of the front and rear legs.

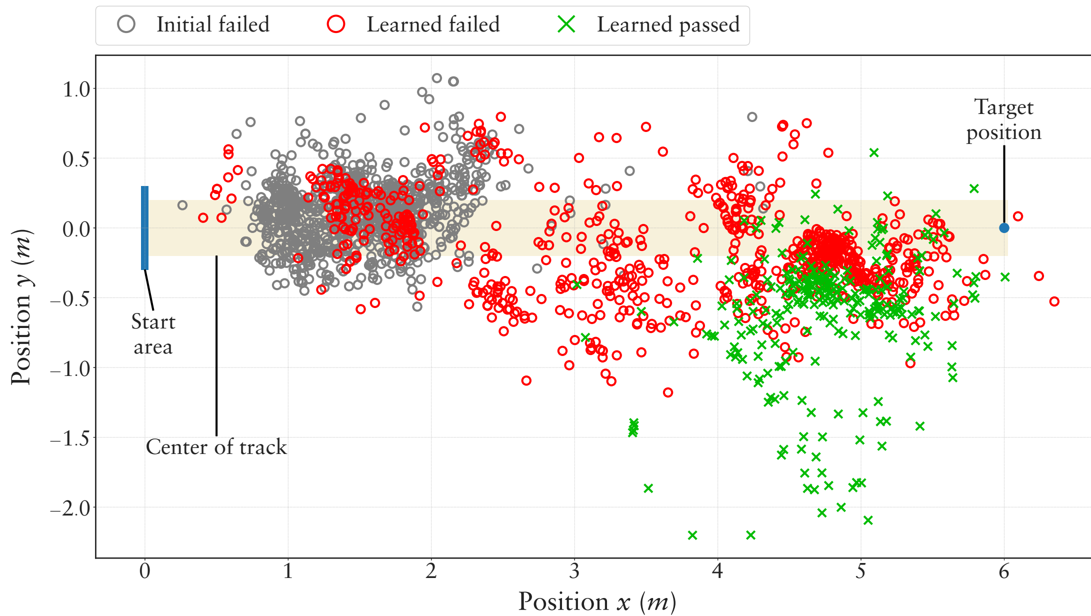


Figure 8.17: Comparison of the navigation controller and the evolved stabilize controller of EASY4 in the rough environment. The plot shows the resulting positions reached by controllers, separated by failed and succeeded trials.

8.2 Experiments with Real Systems

In the following subsections the focus is on evolving locomotion pattern for complex real robotic systems. For all systems the locomotion patterns are generated in the MARS simulation software and selected results are executed on the real systems. Two four-legged systems and one six legged robot are used. The systems count 20 or more *dof* each. Different methods are applied to derive the final proposed methodology for generating the locomotion control of complex legged robots. The first subsection presents two experiments performed for SPOT (Section 2.3.1), where the Sine-Based approach is applied to generate joint pattern. The first experiment limits the *dof* used by the optimization while the second deals with evolving patterns for all *dof*. The second subsection presents experiments with the SPACECLIMBER system (Section 2.3.2). In a first experiment the Sine-Based patterns are used by defining parameter limits fitting to the leg kinematics of SPACECLIMBER and reducing the *dof*s used for the optimization. In a second experiment the Pendulum approach is used to design patterns for all *dof* without limiting the possible leg movement. Beforehand a single leg behavior is evolved on a treadmill for every leg pair. The results are used to define the starting point for evolving walking patterns for the whole system. The last robotic system used is CHARLIE (Section 2.3.3). The experiment with CHARLIE is similar to the second one for SPACECLIMBER. For this experiment a starting behavior is evolved on a treadmill for the front and rear leg pairs. The results are used to generate the patterns for the whole system, including a pattern for the 6*dof* spine of CHARLIE. Finally, a reactive controller is evolved for CHARLIE based on the stabilize experiment for EASY4 8.1.3.

8.2.1 SPOT

The content of this section is mainly based on Römmermann et al. (2008). In this section two learning configurations utilizing the Sine-Based approach are presented and their results are discussed. The first configuration investigates the feasibility of the approach, and the possibility of transferring the simulated behaviors to the real robot. In the first configuration, the complexity of the learning problem is reduced by using empirical knowledge. In the second configuration, the learning algorithm is used to optimize a larger number of parameters, which results in a larger search space for the evolutionary method and a higher potential for resulting behaviors that are not executable on the real system. In these experiments the parameters of the Sine-Based controller are optimized using the CMA-ES algorithm (see 4.2.3).

Environment

The robot has to move on a planar surface with a Coulomb friction value of 0.8 of the contact points. At the beginning of each evaluation, the simulated robot starts at the same initial position.

Evaluation

To evaluate an individual, it is tested in the simulation for a period of 10 s. The evaluation time is chosen empirically and found to be a good balance between computational cost and adequate evaluation of individuals. In order to assess the evolved walking patterns, a fitness value is used based on the following variables measured in the simulation: The distance traveled d, the average torque ϑ and average load ϱ of the joints, a Boolean value τ' that indicates that the maximal torque of the toe joints is above a certain threshold, and a Boolean value τ that indicates if parts of the robot other than the feet has ground contact ($ngc > 0$). The fitness function is defined as sum of four terms. The first term $\frac{-\vartheta}{d}$ assesses the energy efficiency, i. e. the consumed energy for the traveled distance. The second term $\frac{-\varrho}{d}$ is based on the load incurred on the joints over the traveled distance. The third term $-c_n\tau$ adds a strong negative reward if parts of the robot other than the feet come into contact with the ground. The last term $c_n\tau'$ produces a strong negative reward if the maximal torque of one toe joint exceeds a certain threshold.

The resulting fitness function is described by the following equation:

$$fitness = \frac{-\vartheta}{d} + \frac{-\varrho}{d} - c_n\tau - c_n\tau', \tag{8.7}$$

where c_n is a constant value with $c_n = 1000$. Each evolution process proceeds until the `sigma` [1] value of a population is less than 0.01.

Configuration

In both learning configurations, each left leg is made to execute the same patterns that the corresponding right leg executes – with a phase shift between the legs. This restriction is made due to the symmetry of the robot architecture, and has the advantage that patterns for only one side of the robot need to be learned. As explained in Section 7.2.1, each pattern is defined by the four parameters offset (o), amplitude (a), start swing (p_1), and start stance (p_2), which must be optimized. The other parameters that affect the walking

[1] Sigma is an internal value of CMAES which is equivalent to the convergence of an evolution process.

behavior are the CPG waveform period T_{Step}, the duration of the swing phase T_{Swing}, and the phase shifts between the walking patterns of each leg.

Configuration 1 The hand-designed walking behaviors for SPOT only make use of the third shoulder (swinging the leg forward and backward) and the knee joint. During the execution of the walking behavior, the foot is aligned parallel to the body of the robot by a more or less passive control. The upper two joints of the shoulder are not used and are held constant to minimize the complexity of designing the walking patterns for a human developer. This configuration is also an appropriate basis for the first experiment and allows a practical comparison between the human designed and the learned walking behaviors.

Therefore, only the patterns of the third shoulder and the knee joint are learned in this configuration. In this case, the learning algorithm would normally need to optimize a total of eight joint angle patterns, two per leg. However, after taking into account the symmetry constraints, the learning algorithm has to only optimize the patterns of four joints (front-shoulder, front-knee, rear-shoulder, rear-knee) to learn a walking behavior. These four patterns result in a 16-dimensional object variable vector used by the learning algorithm.

The other parameters are held constant in order to minimize the dimensionality of the learning problem. The step period T_{Step} is set to $3\,s$, and T_{Swing} is set to $0.4\,s$. The phase shifts are chosen such that the robot will execute one swing movement after another, starting with the rear right leg, followed by the front left, the rear left, and finally the front right leg. These empirical values are based on experience gained from manually designed walking patterns of the robot. The step period of $3\,s$ results in approximately three steps per leg during the time period evaluated.

Configuration 2 In this second configuration, the evolutionary algorithm is configured to manage all degrees of freedom of the legs. Making use of all joints by the learning algorithm increases the possible space of solutions considerably. Initial tests of various fitness functions have shown that many solutions of this space perform well in the simulation but would not be transferable to the real SPOT system. This is because the simulated robot possesses certain physical possibilities which would cause the real robot to be damaged (e. g. the way the robot could contact the ground). The last two criteria of the fitness function are used to shape the evolutionary process to find a transferable behavior. In this experiment the algorithm has to learn a total of 12 joint patterns (6 for the front legs, and 6 for the rear legs), resulting in 48 parameters. The step period T_{Step} and the swing period T_{Swing} are set to the same values as in the first experiment (Configuration 1). In contrast to the first experiment, the phase shifts are also learned by the evolutionary algorithm, beginning with the values used in the first configuration.

Result

The resulting behaviors perform very well in the simulation and the fitness criteria lead the evolutionary process to produce several results that are transferable. The non-transferable results typically make too much use of the toes and their joints, which are not as stable on the real robot as on the simulated one. Table 8.4 shows some important variables that are calculated over 50 independent runs of both experiments. The standard deviations of most variables shown in Table 8.4 are very small, and thus the values generated by the fitness function also have a low standard deviation.

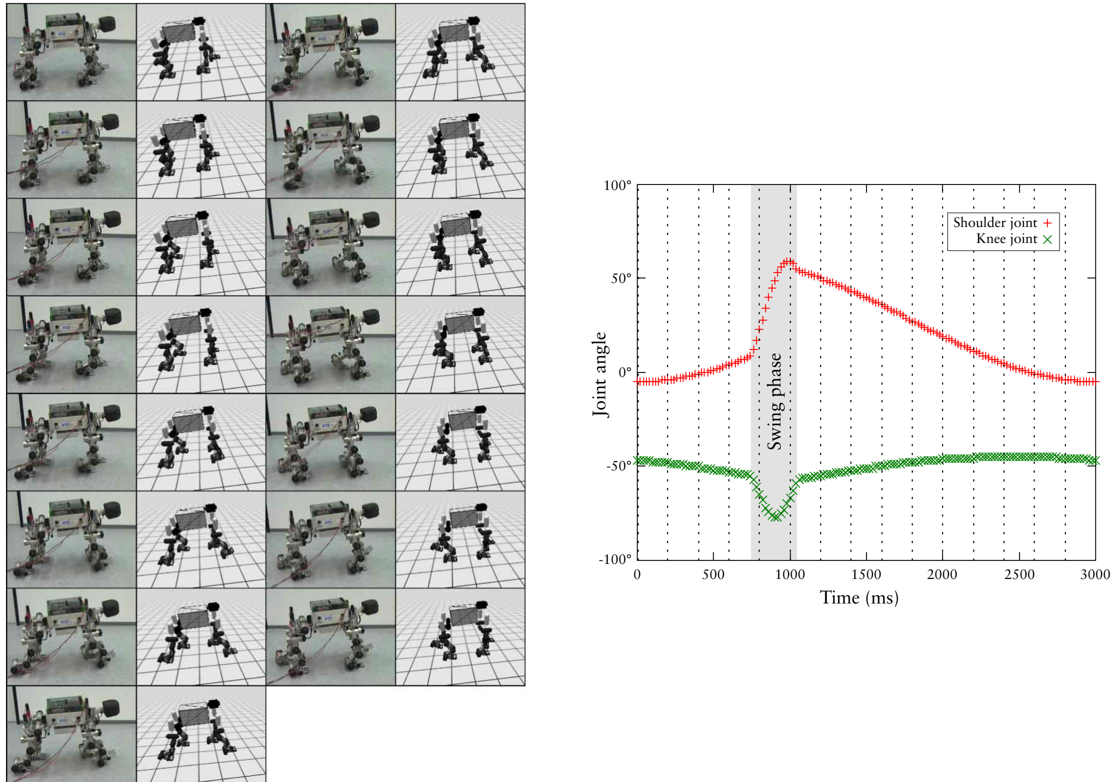


Figure 8.18: A time-progression of snapshots showing an evolved walking behavior that is executed on both the simulated and real robot. The images are arranged in a chronological order from top to bottom, starting with the left column. The joint trajectories executed by the front legs are shown in the plot on the right. These angle values are recorded on the real robot during the execution of the evolved walking patterns. The dotted vertical lines indicate the points at which the snapshots are taken.

The results of the second configuration have significantly ($p < 5 \times 10^{-8}$)[2] better fitness values than the first results of the first configuration. These better fitness values are primarily due to the fact that the robot reaches a significantly higher speed (i. e. a further distance is traveled during a fixed period of time) ($p < 2 \times 10^{-7}$).

Some walking behaviors that have a good fitness in both experiments are transferred directly and without modifications to the real robot. One transferred walking behavior of the first configuration is shown in Figure 8.18. When executing on the real robot, it proves to be stable, closely approximating the behavior of the simulated robot. The plot in Figure 8.18 shows the joint trajectories for the shoulder and the knee of the front right leg. The dotted vertical lines indicate the exact moments at which the pictures in this Figure are taken. It can be seen from this image sequence that the real robot's behavior is very similar to the behavior of the simulated robot. As compared to the manually-designed gaits, the evolved gaits result in a faster and more balanced dynamic walking behavior. The main difference between most of the gaits learned and the manually-designed behavior lies in that during particular stages of the walking process of the learned gaits two legs are in the air at the same time, while in the manually-designed behavior only one leg is in the air.

[2] Significance levels are computed using an one-sided Student's t-test with.

Table 8.4: Statistics of various important variables over 50 independent evolution processes of both experiments with SPOT.

Variable	Optimum		Average		Std. Deviation	
	Expt1	Expt2	Expt1	Expt2	Expt1	Expt2
Fitness	4.98	4.28	8.66	6.43	1.60	1.26
Num. of Evaluations	2292.00	7590.00	4187.52	11838.60	1020.86	1423.78
Distance in m	3.17	4.05	2.10	2.84	0.59	0.35
Avg. Height in m	0.59	0.61	0.56	0.54	0.02	0.04
Avg. Torque in N-m	2.64	2.71	3.61	4.03	0.30	0.49
Avg. Load in N-m	10.32	10.62	16.93	15.79	2.47	2.56
Avg. Pitch in degrees	0.74	0.86	2.86	6.60	1.02	4.51
Avg. Roll in degrees	0.52	0.10	2.86	4.28	1.00	2.19

8.2.2 SpaceClimber

It is unlikely to generate a desired locomotion behavior of a morphologically complex robotic system without limiting the search space or starting close to the desired solution. In case of the six-legged SpaceClimber system, the probability to evolve a four-legged locomotion behavior with the middle legs hanging above the ground can easily be higher than evolving a behavior that coordinates all six legs. Two experiments are performed to generate a locomotion behavior for SpaceClimber. The first starts the evolution from a predefined behavior and evaluates the local optimization capabilities utilizing a modified Sine-Based approach. Since it is desired that no robot-specific foreknowledge is needed, the second experiment – using the Pendulum approach – evolves the starting point of the final optimization by generating a controller for a single leg pair. Results of both experiments are transferred to the real system.

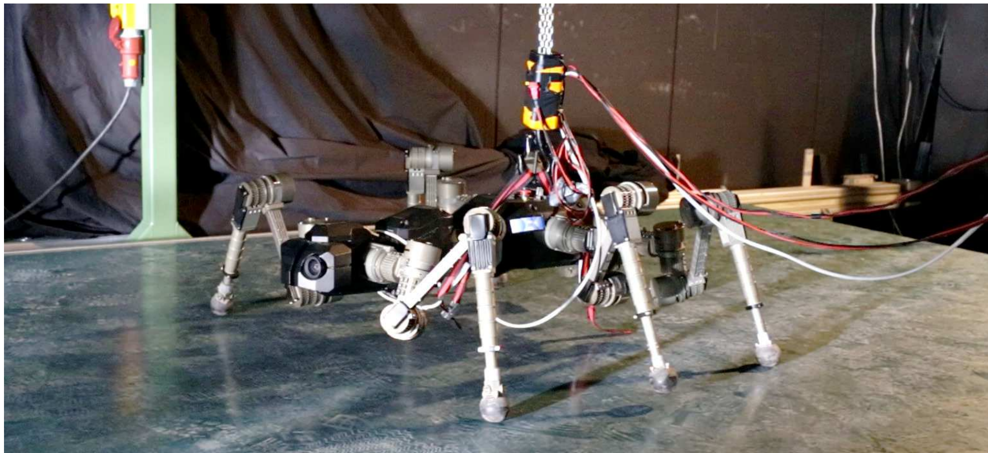


Figure 8.19: The real SpaceClimber system on the locomotion test-bed. Distance sensors are used to control the speed of the treadmill, which keeps the robot in the middle of the test-bed.

All behaviors tested on the real system are performed with SpaceClimber walking on a treadmill as shown in Figure 8.19. The treadmill test-bed includes a distance measurement of the robotic system in the forward direction, which is used to control its speed. A PID controller sets the speed of the treadmill by using the difference of a target distance and the actual robot distance as control parameter. The robotic system is secured by a

motor driven chain that can be used by the operator to lift the robot up. For a general classification of the evolved behaviors, different experiments with the classic control of SPACECLIMBER are performed with the real system – defining a set of reference behaviors. Table 8.5 lists the parameters and the general evaluation parameters consisting of the speed and mean / max power consumption of the whole system.

The parameters are measured for a period of $10\,s$. The target speed is calculated according to Equation 2.8 introduced in Section 2.3.2. Since the speed measurement of the treadmill is not precise and many evolution results are only tested and compared in simulation, the overall system speed of the reference behaviors is evaluated with the simulation model. The measured speed is higher than the target speed mainly due to the fact that the model calculates the foot positions in the middle of the foot sphere, thus the real leg length is approximately $2\,cm$ longer. Compared to the presented properties of SPACECLIMBER in Bartsch (2014), the measured data represents a plausible extrapolation of the system characteristics for higher speeds.

Table 8.5: SPACECLIMBER control parameters used used for the model-based control to create comparable data. Only the parameters that are varied from the default are listed. See Section 2.3.2 for a description of the model-based control and its parameters. The value applied on the step *y* parameter for IK_4 and IK_4 is used to compensate for a drifting behavior of the robot to the sides of the track.

Config	Step period	Shift	Step y	Step length	φ	Target speed	Speed in simulation	Power mean	Power peak
IK_1	$2.0s$	12%	$0cm$	$35cm$	0.5	$0.20\frac{m}{s}$	$0.22\frac{m}{s}$ (110%)	$142W$	$276W$
IK_2	$2.0s$	12%	$0cm$	$35cm$	0.0	$0.20\frac{m}{s}$	$0.21\frac{m}{s}$ (105%)	$173W$	$374W$
IK_3	$1.5s$	20%	$5cm$	$35cm$	0.0	$0.29\frac{m}{s}$	$0.31\frac{m}{s}$ (107%)	$203W$	$634W$
IK_4	$1.5s$	20%	$-5cm$	$30cm$	0.0	$0.25\frac{m}{s}$	$0.27\frac{m}{s}$ (108%)	$236W$	$576W$
IK_5	$2.0s$	20%	$0cm$	$45cm$	0.0	$0.28\frac{m}{s}$	$0.31\frac{m}{s}$ (110%)	$191W$	$399W$

Sine-Based Approach

Environment For this experiment the SPACECLIMBER have to move on a planar surface with a Coulomb friction of 0.8.

Evaluation One parameter set is tested in the simulation for a period of $25\,s$. Two different evaluation function are used. The first function Fitness_1 divides the average motor currents by the distance traveled, whereby the distance taken into account is limited to $5\,m$. The second function Fitness_2 uses the same fitness calculation as previously used for the pattern generation of the simulation robot EASY4 (compare Figure 8.2). Additionally, a feet slippage factor, provided by `locomotion_environment`, is added as a fitness criterion with a weight of 4.0 for the second configuration. The first configuration is simpler and is designed to optimize stability indirectly by minimizing the motor currents. The parameter ranges for the optimization are empirically chosen to produce good results on the first fitness configuration. If the search space becomes too complex, either nontransferable behaviors are more likely or even the parameter optimizer CMA-ES does not converge, in which case `sigma` increases, resulting in a random search if `sigma` becomes too large. This behavior of CMA-ES might be effected by the mapping of the CMA-ES parameter space to the target space, introduced in Section 4.2.3. Nontransferable behaviors are mostly a result of mechanical stress that is only very roughly approximated in the simulation model, similarly observed in the experiments with SPOT.

Configuration Instead of optimizing the whole control system from scratch, an empirically predefined pattern fitting to the robot morphology is used as a starting point. For this configuration the Sine-Based pattern is used allowing, due to the pattern representation, to define the parameters for a start configuration intuitively. Additionally, the representation enables to control the maximal offset to the start pattern by limiting the parameter range optimized for the offset parameter. Thus the whole control architecture is predefined and the CMA-ES optimizer is used to adapt the pattern. The first joint of the leg is not used, similar to the model-based pattern approach for SPACECLIMBER as introduced in 2.3.2. The joints optimized are shown in Figure 8.20. To allow a more

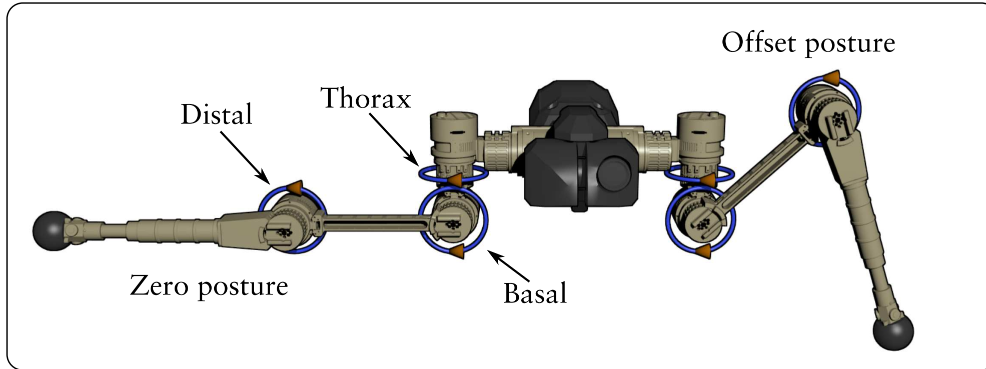


Figure 8.20: SPACECLIMBER kinematic leg model and joints used for the Sine-Based pattern optimization. The leg on the left hand side is in the zero posture, while the leg on the right hand side depicts the start posture resulting from the defined offset values for the sine pattern approach. All leg pairs are handled equally concerning the kinematical configuration.

precise shaping of the joint pattern the original Sine-Based approach is altered by using one parameterized sine wave for the swing phase and another for the stance phase. The input signal for this modified module is the position in the stance (ϕ_1) and swing (ϕ_2) phase where zero marks the start of the corresponding phase and one the end. The input signal, scaled by a percentage factor (p_1 for the stance and p_2 for the swing phase) plus a phase offset (φ_1 and φ_2), is used as input for the sine function. The output of the sine function is scaled with an amplitude factor a_1 or a_2. A state input (*state*) defines which parameters are combined with the offset o to generate the joint angle. The following equations defines the adapted approach:

$$\theta = \begin{cases} o + sin(\varphi_1 + \phi_1 p_1)a_1 & \text{if } state = 0 \\ o + sin(\varphi_2 + \phi_2 p_2)a_2 & \text{else.} \end{cases} \tag{8.8}$$

On one hand, more complex patterns can be generated with this approach. On the other hand a smooth transition between swing and stance phase is not guaranteed and is left to the optimization process. The parameter ranges defined for the optimization are given in Table 8.6. The offset parameter of the basal and distal joints are excluded from the optimization, resulting in 19 parameters to define a leg pattern. One leg pattern is optimized for the front, one for the middle, and a third for the rear legs. Overall, 57 parameters define the walking pattern in this experiment. Step period, swing/stance ratio and phase shift are predetermined and not further optimized. The step period is set to 2 *s*, the swing phase covers 10% of the step period, and the phase shift defines a cross gait coupling the swing phase of the front left, middle right, and rear left leg to one swing movement and the other legs to the second swing movement executed with 1 *s* time offset.

Table 8.6: Parameter ranges defined for the Sine-Based approach to evolve a SPACECLIMBER open-loop walking controller.

Joint	p_1	φ_1	a_1	p_2	φ_2	a_2	o
Thorax	2 – 4	0.5 – 1.7	0.25 – 0.75	2 – 4	4 – 5.2	0.25 – 0.75	-1 – 1
Basal	2 – 4	0 – 0.6	0.2 – 1.5	2 – 4	1 – 4	0.1 – 0.6	-0.75
Distal	2 – 4	0 – 0.6	-1.5 – 0.5	2 – 4	1 – 4	-0.6 – 0.1	-2.0

Result For each of the configurations one result is presented in this section. Since these results are produced by a series of individual performed optimizations, no substantiated statement can be given concerning the probability of successfully generating a feasible walking pattern for the real system. However, all tested behaviors perform very close to the simulated behaviors after further adapting the simulation model compared to Section 3.3. This adaption consists of making the joint constraint softer to compensate for mechanical compliance, which is increasing over time, mainly at mounting points in the structure. Additionally, the center of mass is moved towards the front of the system. As a side effect of this adaptation, the resulting executed joint patterns are smoother in the simulation than on the real system but all behaviors transferred show very similar characteristics compared to the simulated behaviors. Additionally, as result of the adaptation, producing a steady walking pattern is even harder in simulation then on the real system.

Fitness$_1$ Figure 8.21 presents one evolved joint pattern for the whole system executed on the simulation model used for optimization. The right and left legs are inverted to adapt to the underlying kinematic model. For all legs the foot trajectory is defined by the thorax and basal joint patterns. The desired position of the basal joints is given by a constant value and the pattern observed in the actual angle is produced by the soft joint properties of the simulation model.

The speed of the robot executing the evolved pattern in simulation is 0.26 m/s. The pattern executed on the real SPACECLIMBER system results in a power consumption of 177 W in average and 395 W maximal. With this characteristics the evolved pattern poses a plausible result to be used on the real system. The speed and power consumption fit best to the IK_2 reference behavior with a speed of 0.21 m/s, an average power consumption of 173 W, and a peak power consumption of 374 W. The IK_2 behavior is compared to the evolved pattern to present the potential difference. The comparison is done based on data generated with the simulation. The first test executes both behaviors on a planar surface. The simulation is executed until the behaviors become stable, reaching their maximum forward speed. Then the robot is positioned back to the start frame while keeping its current dynamic proprieties and joint angles. Afterwards, logging is started for a period of 10 s providing the data sets. For the second data set, the robot's main body is fixed in the simulation and the behaviors are executed without ground interaction, thus representing only the result of the forward kinematic.

Figure 8.22 compares the foot height of the left legs between the two behaviors. The middle left leg is used to align the recorded data by its swing phase. All following analyses of the data set with the robot walking on the ground, are aligned by the same offset. The evolved patterns produces a more expanding swing phase compared to the IK_2 pattern. The resulting step period of the evolved pattern is 2.077 s as result of the update rate used in the control graph. Figure 8.23 depicts the body velocity in the forward and lateral direction. The average forward velocity of the evolved pattern is larger. Also the amplitude of the velocity change in both directions, forward and lateral, is larger, while

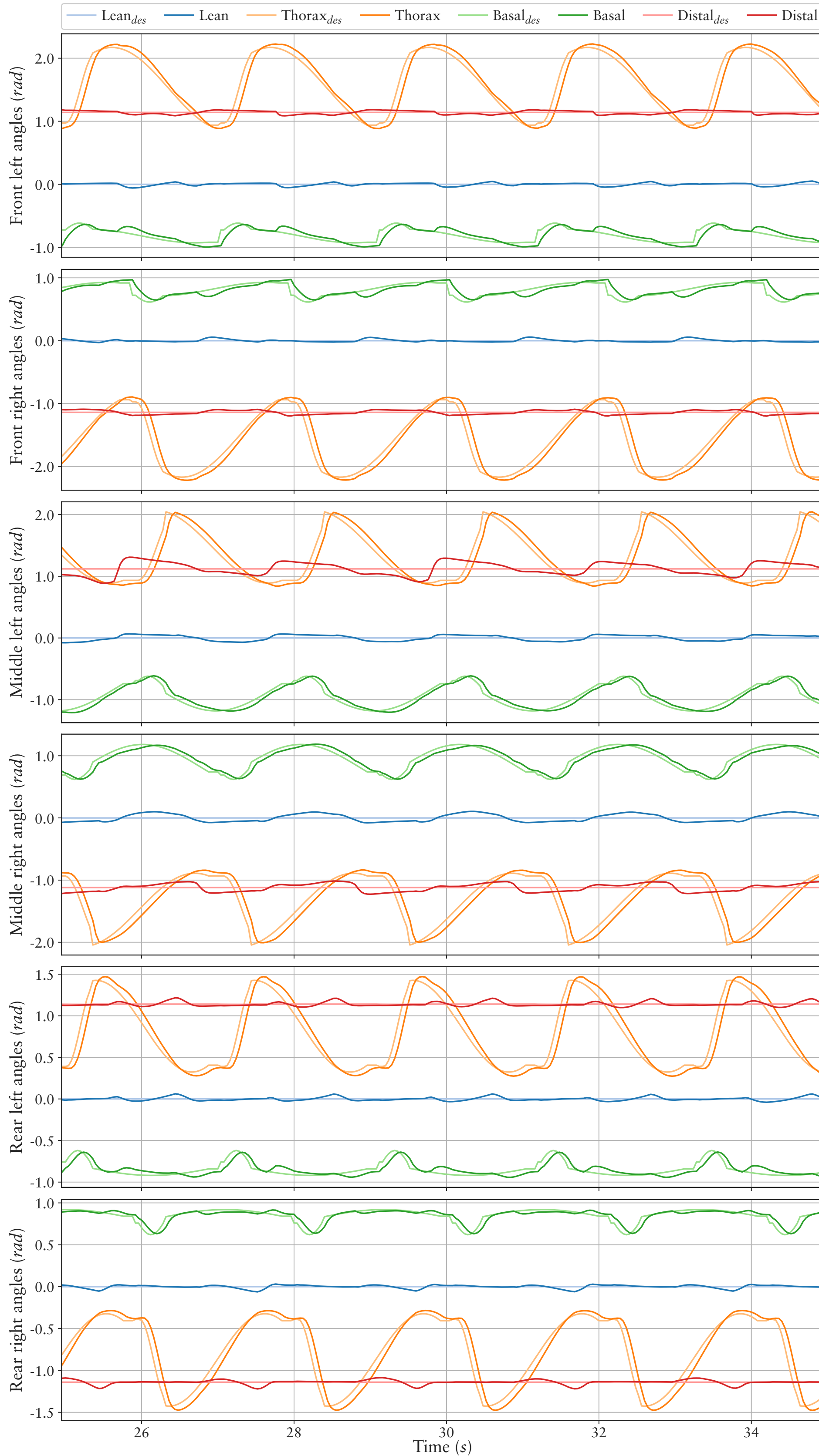


Figure 8.21: Joint patterns created by the Sine-Based approach for SPACECLIMBER with the Fitness_1 configuration. The desired joint angles are compared with the joint angles executed by the simulation model.

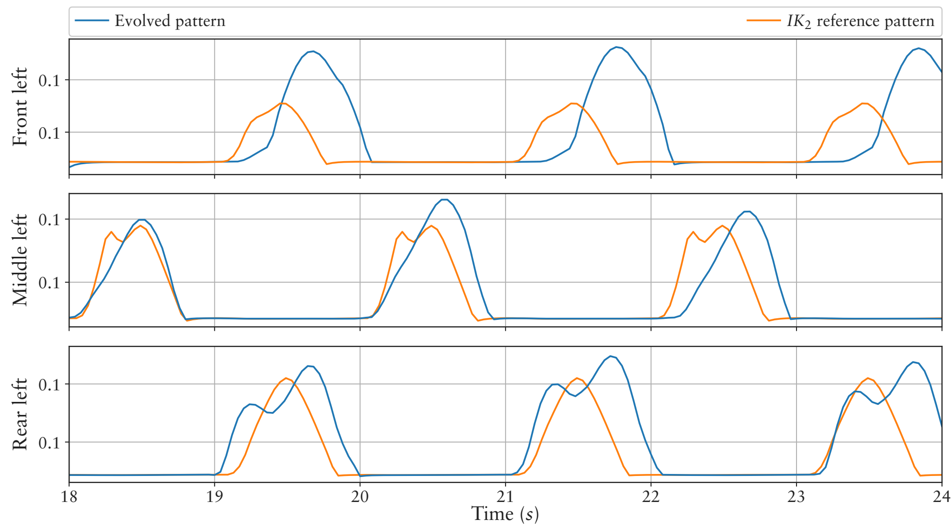


Figure 8.22: Comparison of the feet z position between the left legs of SPACECLIMBER. The patterns are aligned by the offset of the middle left leg (leg 2).

the frequencies are lower, representing a less steady but smoother movement of the robot's body. Comparing the height of the robot's body (see Figure 8.24) shows no real difference of the frequency. The evolved pattern has a higher main posture and a bigger amplitude on the height curve produced by the walking pattern. Figure 8.25 compares the foot trajectories resulting from the forward kinematic model of the left legs between the two behaviors. Again the phase is aligned by the middle leg, while this time the phase time of the evolved pattern is scaled to 2 s to allow a better comparison. The evolved pattern shows a trajectory of the feet on the lateral axis which produces the lateral movement of the robot's body. The general shape of the rear leg movement on the vertical-axis is quite similar, with a smaller and shorter swing phase of the evolved pattern. The higher and longer swing phase of the evolved rear feet patterns, while walking on ground, is a result of the overall tilt movement of the robot. Thus, the evolved pattern uses the robot's tilt movement to support the swing phase of the rear legs. A similar result can be observed for the front legs. This tilting behavior is mainly produced by the non-parallel movement on the vertical-axis of the middle legs in the stance phase combined with the resulting center of mass shift. The amplitude of the forward movement of the legs is similar between the behaviors matching to the similar forward speed. While the movement of the IK_2 behavior shows a continuous motion, the evolved behavior settles for a short period of time before switching into the swing phase, explaining the higher standard deviation of the robot's speed.

As can be seen from this comparison, the evolved pattern makes use of the dynamics of the system to support the locomotion. Additionally, it produces a stable control of the real system with a comparable power consumption. However, the possible results produced by the evolution are highly influenced by the limited parameter ranges defined, and by adapting these ranges the quality and main characteristics of the evolved patterns could be further tuned.

Fitness$_2$ The pattern evolved with the second fitness configuration is tested with three different activities, whereby the activity parameter linearly scales the amplitude of the joint patterns. An activity of zero results in no movement, while an activity of 1.0 produces the originally evolved pattern. For the evolved pattern, also lower activities

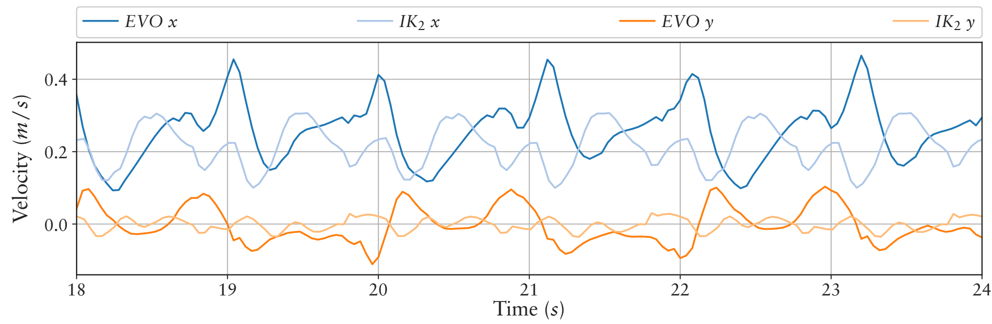


Figure 8.23: Comparison of the linear velocity on the x and y axis of the robot main body for the result of the Fitness_1 configuration and the IK_2 reference behavior.

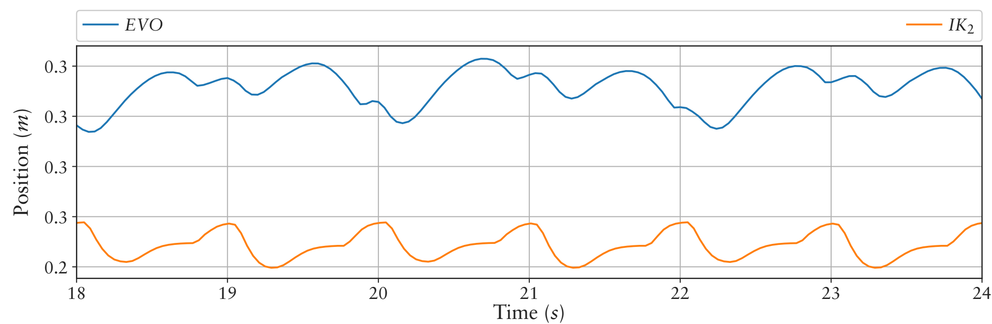


Figure 8.24: Comparison of the robot height.

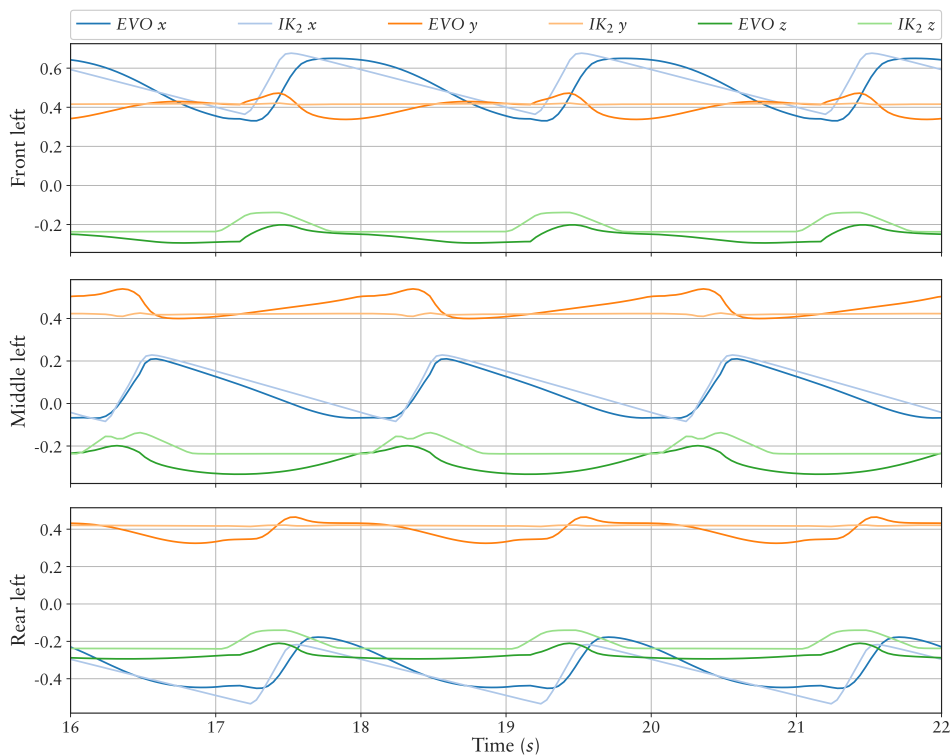


Figure 8.25: Feet positions of the robot fixated in the simulation (forward kinematics) with the result of the Fitness_1 configuration compared to the IK_2 reference behavior.

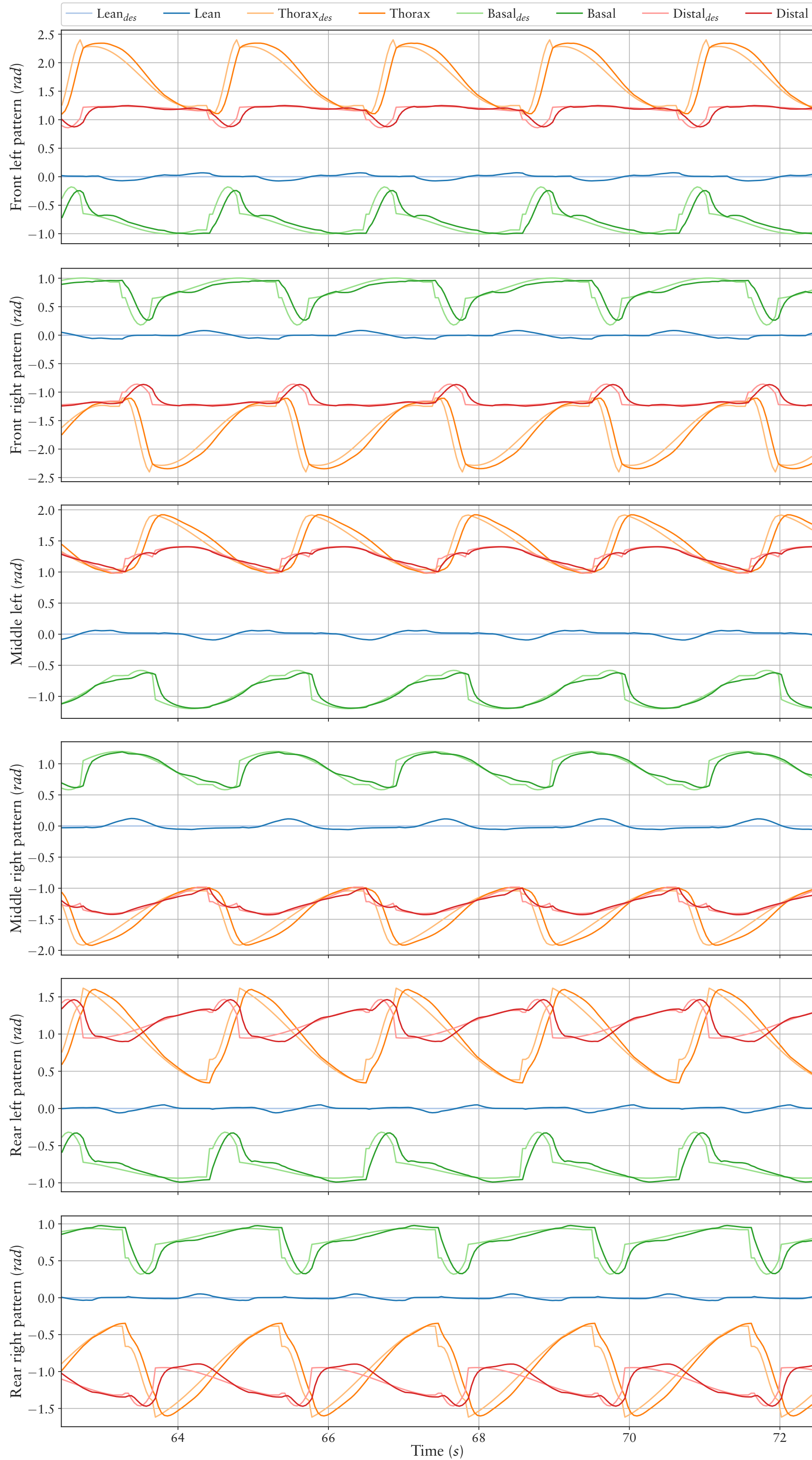


Figure 8.26: Joint patterns created by the Sine-Based approach for SPACECLIMBER with the Fitness$_2$ configuration. The desired joint angles are compared with the joint angles executed by the simulation model.

Table 8.7: Characteristics of one result for the Fitness$_2$ configuration tested on the real SpaceClimber system and in simulation with different activity.

Activity	Simulation speed	Power mean	Power peak
0.5	$0.11 m/s$	$124 W$	$181 W$
0.8	$0.19 m/s$	$159 W$	$337 W$
1.0	$0.26 m/s$	$193 W$	$381 W$

produce stable walking behaviors by varying robot speed. Table 8.7 lists the resulting characteristics of the evolved patterns. Again, the transferred behavior proved to represent a feasible control strategy on the real system. Figure 8.26 presents the joint angles produced by the original evolved pattern (an activity of 1). Compared to the previous result of Fitness$_1$, for all joints, configured for the evolution, a pattern is generated. Figure 8.27 presents the feet positions given by the forward model (Fk_x, Fk_y, and Fk_z) of the robot plotted together with the vertical movement while the robot is walking on ground (walking z). For comparison the vertical movement (walking z is shifted down by the mean body height of the walking pattern. For all legs the upper part of the swing phase matches very well with the forward model and the measured walking data.

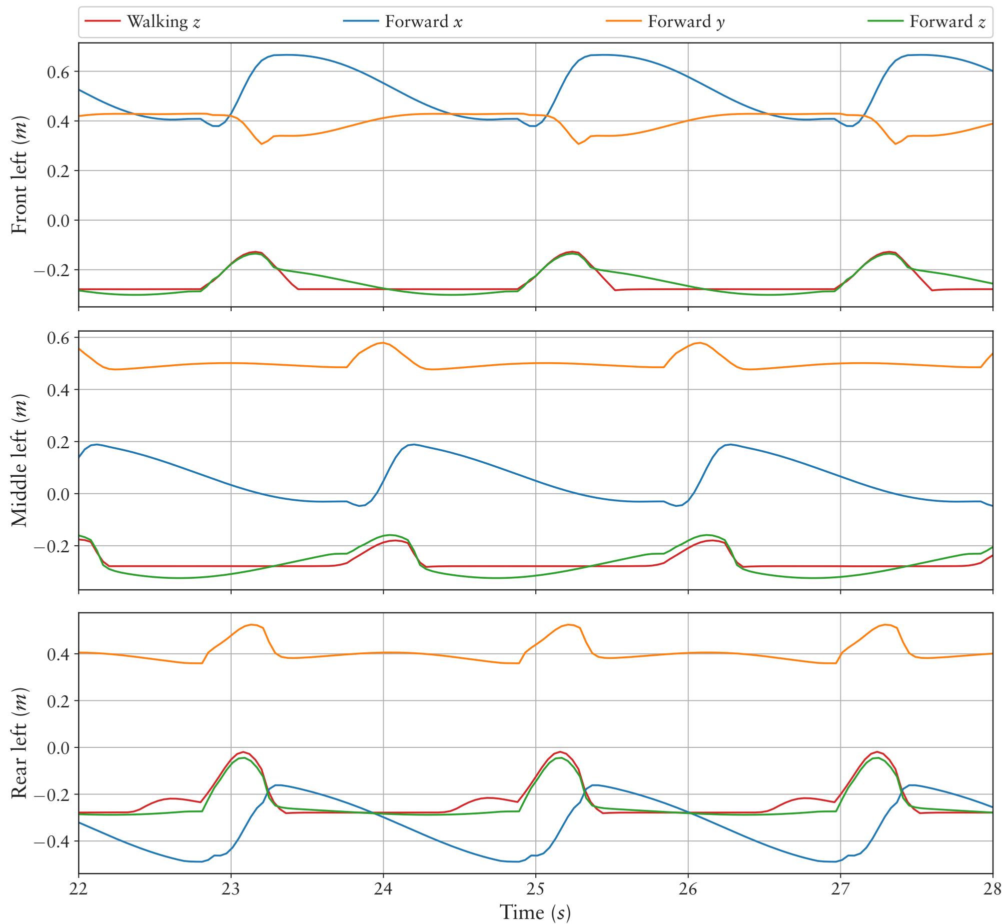


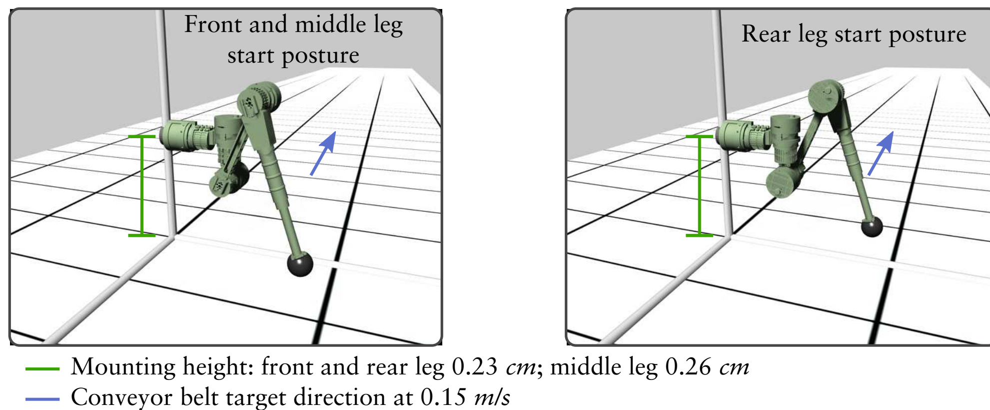
Figure 8.27: Feet positions of the robot fixated in the simulation (forward kinematics) and comparison of feet z movement while walking on the planar surface with the result of the Fitness$_2$ configuration.

As result, the leg swing movements on the vertical axis are not supported by a center of mass shift like observed in $Fitness_1$. This might be due to limiting the robot's body movement by the fitness function. Another result of this restriction combined with the limited parameter ranges is that the evolved pattern produces a short swing phase of the legs and hard touchdown impacts. At least adding the swing time as parameter to the optimization could further improve the results of this experiment.

Bottom-Up Pendulum Approach

The Sine-Based pattern approach needs a lot of parameter range tuning to the specific target robot morphology if the control problem becomes more complex. For the next experiment the initial walking pattern for the whole system is generated by evolving patterns beforehand for each of the leg pairs – representing a bottom-up approach.

Environment A single front, middle, and rear leg of SpaceClimber is mounted above a treadmill in simulation. Figure 8.28 depicts the setup, whereby the start posture for the front and middle leg is the same while the start posture of the rear leg differs in a backward orientation of the thorax joint. Additionally, leg length and mounting height is different between the middle leg and the read and front legs. The kinematic model used in this section is the same as described in Section 2.3.2. The simulation setup to generate the controller for the whole SpaceClimber is the same as used for the experiment evolving the Sine-Based controller.



— Mounting height: front and rear leg 0.23 *cm*; middle leg 0.26 *cm*
— Conveyor belt target direction at 0.15 *m/s*

Figure 8.28: The single leg simulation setup for SpaceClimber.

Evaluation The fitness for the single leg controllers is calculated by evaluating the average treadmill speed, the center position of the foot on the forward axis, and the velocity difference of the foot and the treadmill on contact, representing a slippage criterion. Figure 8.29 depicts the bagel graph implementing the evaluation function. The treadmill speed is only considered within the stance phase of the leg, to prevent evolving a poking behavior. While the fitness graph represents a general evaluation function for a single leg control on a treadmill, the parameters marked orange have to be defined for the specific leg morphology and start configuration. The first parameter, the upper one in the graph, defines the target center position of the foot, which is set to 10 *cm* for the front and middle legs and to -10 *cm* for the rear legs correlating to the start posture. This criterion is used to prevent evolving behaviors that produce collisions between the legs if the results are used for the whole system. The second parameter defines the target speed

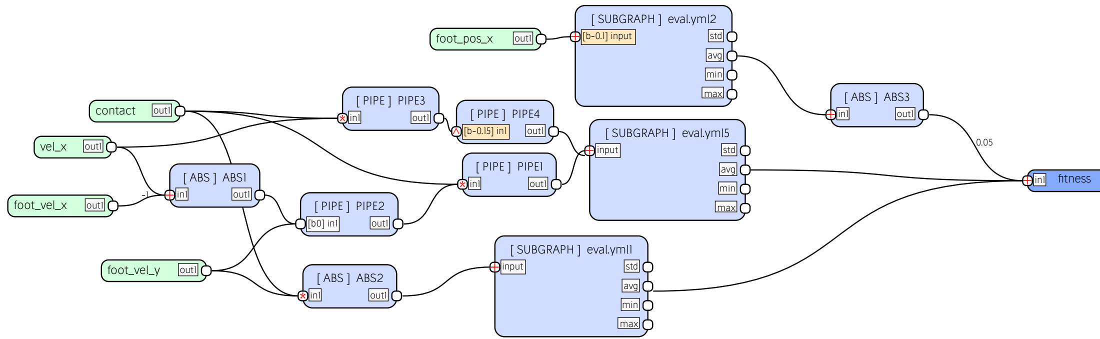


Figure 8.29: The fitness graph used for the evolution of the SPACECLIMBER's leg control.

of the treadmill. Defining a target speed instead of moving as fast as possible increases the compatibility of the resulting solutions for the different legs. Also, later on, the target speed of the whole system is set to the same value, to focus more on stability than on speed or energy efficiency.

Each behavior is tested for a period of 20 *s*, while the first second is not evaluated. Again, individual evolutions are performed to design the fitness function and to produce and manually select individual results for the next iteration, evolving the whole open-loop control for SPACECLIMBER. The evolved pulse networks are used as a starting point for SABRE to evolve the whole SPACECLIMBER control. Additionally, the phase shift is evolved, similar to the EASY4 open-loop setup. The evaluation function for the controller of the whole SPACECLIMBER is the same as described for the Fitness_2 experiment evolving the Sine-Based controller.

Configuration For the pattern representation the Pendulum approach is used, since it is able to generate satisfying walking controllers for the EASY4 robot. The control concept is similar to the one used for the open-loop control of EASY4 reduced to one leg, see Figure 8.30. The control graph for the whole SPACECLIMBER system is shown in Figure 8.31. Beside the leg patterns, the phase shift of the legs are adapted starting the evolution with a cross-gait.

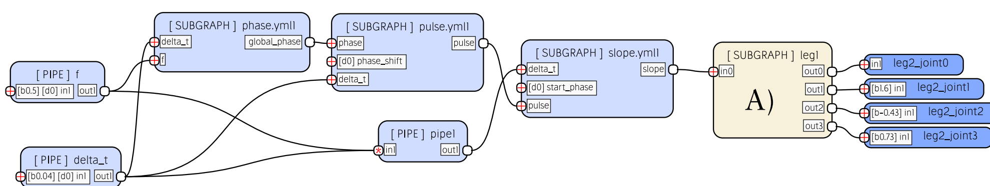


Figure 8.30: The control graph for a SPACECLIMBER leg. The parameters of the output nodes define the initial posture of the leg. The values shown are used for the front and middle legs, while the values are inverted for the rear legs. The main control graph is not adapted, while the inner structure of node A) is evolved.

Result Figure 8.32 depicts the evolved patterns for the individual legs compared to the result of the second phase, adapting the pattern for the whole system. The general shape of the front leg patterns are kept, only a small shift in the phase and an amplitude adaptation is done on the lean and basal joint. The thorax and distal patterns remain very close to the initial pattern evolved on the treadmill. In case of the middle leg, the thorax and basal patterns are not modified a lot, while the other two show a phase shift, an

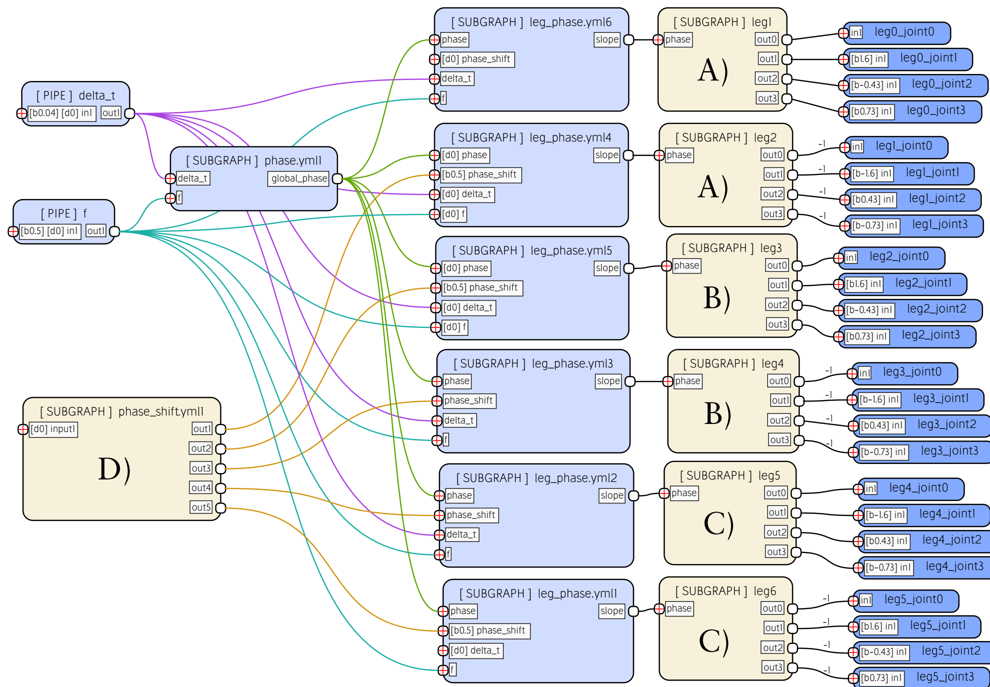


Figure 8.31: The control graph for the whole SPACECLIMBER system. Similar to the configuration for the single legs, the subgraphs A), B), C), and D) are evolved while the main graph is not adapted. The subgraphs defining the front, middle, and rear leg patterns are used twice with inverted outputs for the right legs. The starting point for the leg subgraphs is derived from the results of the single leg experiments.

adaptation of the shape, and a reduction of the amplitude of the lean pattern. The largest adaption can be observed for the rear legs. In relation to the thorax pattern the phase of the other joints is shifted by up to 25%. The shape of the thorax joint pattern is adapted, while the other joint patterns are mainly modified in the amplitude and offset.

Figure 8.33 shows the resulting foot position of the forward kinematic model and compares the vertical movement with the recorded foot position while walking on ground. The recorded foot height is shifted down by the average body height of 0.23 m of the walking behavior. The swing phase of the front and rear leg match well between the forward model and the recorded data. As a result, except for the middle legs, the dynamic effects do not take much influence on the resulting pattern. The middle legs show a very small amplitude on the vertical axis and the swing phase differs between the forward model and the resulting dynamic behavior. The walking swing phase has to be a result of the dynamic system properties.

Overall, from all evolved joint patterns, the resulting characteristics concerning the movement of the body are the closest compared to the *IK* generated patterns. The main difference is the use of the lean joint, resulting in smaller joint pattern amplitudes of the thorax joints. The behavior produces a forward velocity in the simulation of 0.17 m/s with an average power consumption on the real system of 148 W and a peak power consumption of 263 W. The measured performance is in a reasonable range for SPACECLIMBER compared with the results of the other experiments.

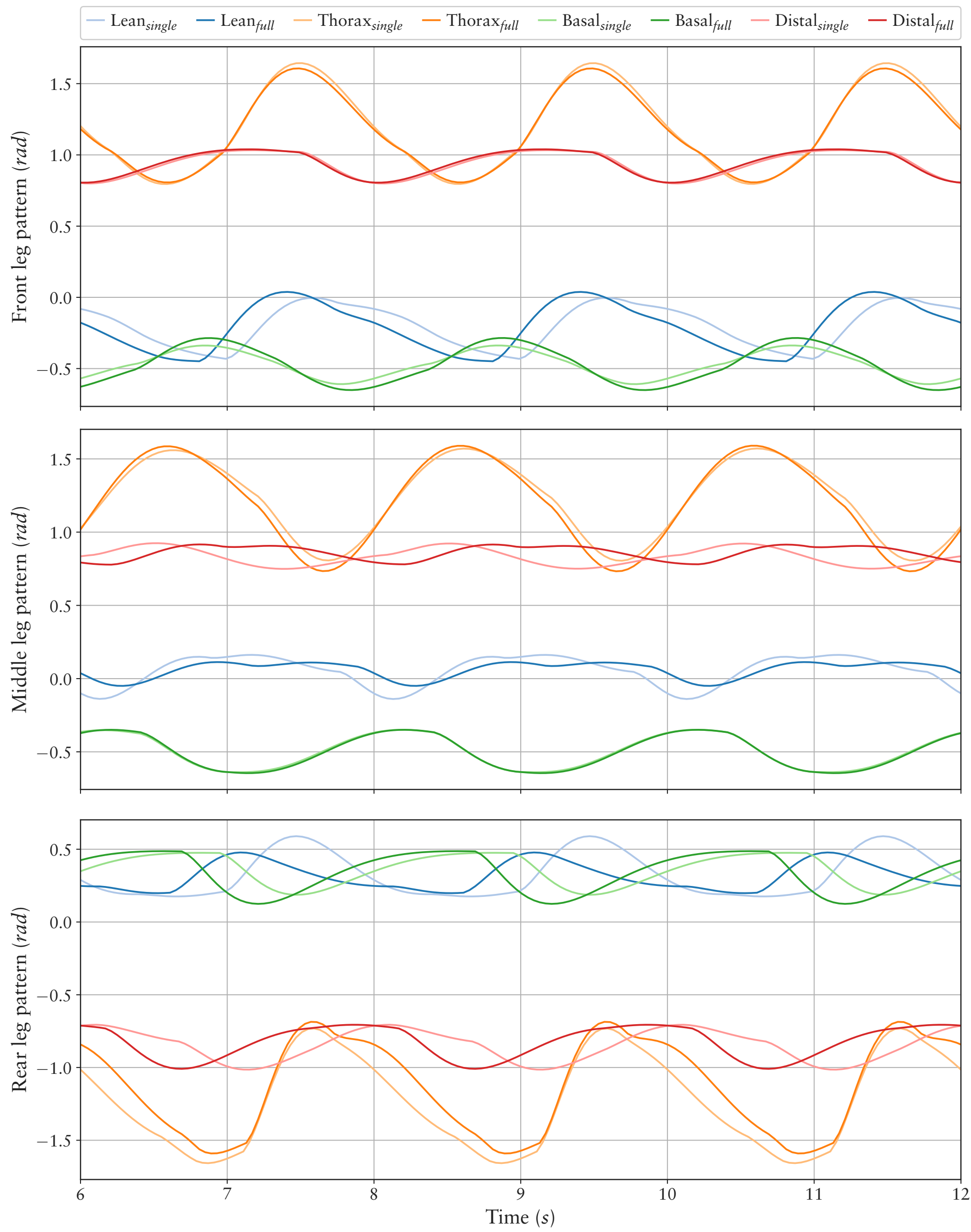


Figure 8.32: Comparison of the leg patterns generated by the evolution of the single leg control and the adaption of the patterns to the whole system. The leg patterns are aligned by the positive movement of the thorax joint.

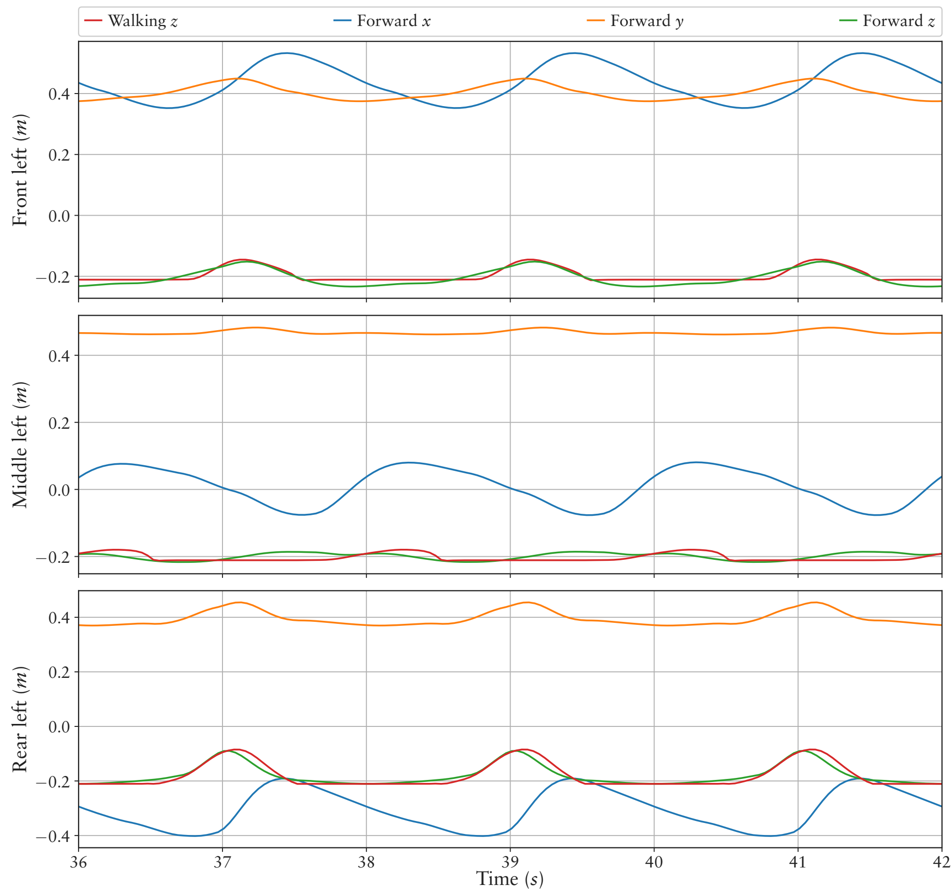


Figure 8.33: SPACECLIMBER foot positions of the bottom-up result generate by the forward kinematic model compared to the foot height measured while walking on the ground.

8.2.3 CHARLIE

The last target system is CHARLIE as described in Section 2.3. To evolve a controller a similar approach is set up as described in Section 8.2.2 for SPACECLIMBER. In a first setup (FRONTLEGS) a behavior for the front legs is generated, while the front body is mounted by a prismatic joint above a treadmill. Similarly, a behavior is generated for the rear legs (REARLEGS). OPENLOOP combines the results of FRONTLEGS and REARLEGS as starting behavior to generate a behavior for the entire CHARLIE robot. In the last setup (CLOSEDLOOP the controller is extended by a reactive module compensating forces applied to the front body of the system.

FRONTLEGS

Environment To generate a walking pattern for the front leg pair of CHARLIE a simulation model is used including only the forepart of the robot. In contrast to the setup used for SPACECLIMBER a leg pair is used instead of a single leg. The model is mounted by a prismatic joint (support joint) above a treadmill preventing the robot from falling over. Since the axis of the support joint is along the vertical axis (z-axis), the legs still have to support the robots weight while moving the treadmill backwards. The setup is depicted in Figure 8.34.

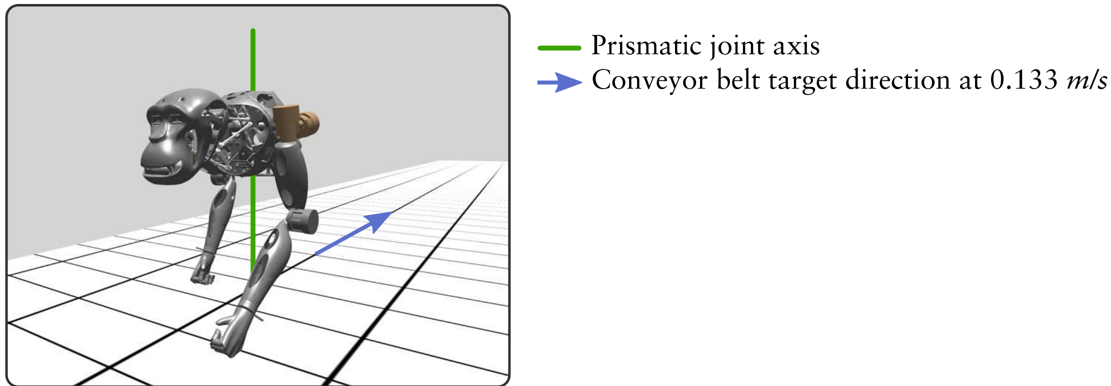


Figure 8.34: The simulation setup to generate a walking behavior for the front leg pair of CHARLIE.

Configuration The top-level graph defining a 2 s step time and a cross gait phase shift is shown in Figure 8.35. The nodes `front_pendel1` and `front_pendel2` are instances of a `front_pendel` module that is evolved as explained in Section 8.1.1 and shown in Figure 8.5. A start posture of the front legs is configured in the control graph based on the posture of the model-based control concept of CHARLIE.

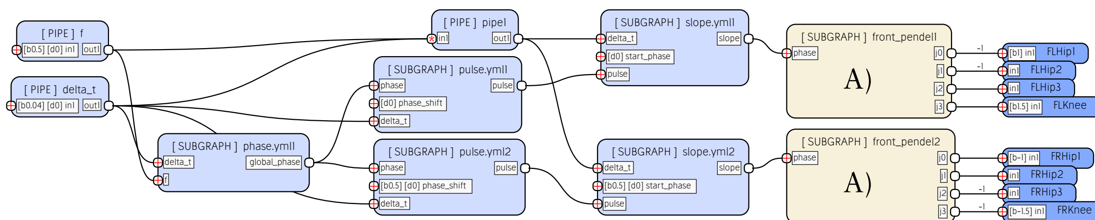


Figure 8.35: The top-level BAGEL graph defined for evolving walking patterns of the front leg pair of CHARLIE. The function of the sub-graph A) – used for both legs – is generated by the evolution configured similar to Figure 8.5. The bias of the output nodes defines a start posture of the legs inspired from the starting posture used by the original model-based controller of CHARLIE.

Evaluation Due to the fact that the legs have to support the robot's front body in the stance phase the evaluation is based on the function used in Section 8.1.1 instead of the function used in Section 8.2.2 for the single SPACECLIMBER legs. Due to the decreased degrees of freedom of the robot's front body by the support joint, two modification of the evaluation function are introduced. A first modification is added by not evaluating the roll and pitch angle of the robot. A second modification is done by evaluating the feet position on the lateral axis (y-axis) with the intention to generate behaviors with the feet operating below the shoulders on the lateral position. A possible adaptation of the lateral feet position should be generated in the third setup where the robot's degrees of freedom in the world are not limited. The final fitness graph used is shown in Figure 8.36. Similar to the experiments performed with SPACECLIMBER the feet slippage factor of the `locomotion_environment` is used (4.2.1). Each individual is tested in the simulation for a time period of 30 s.

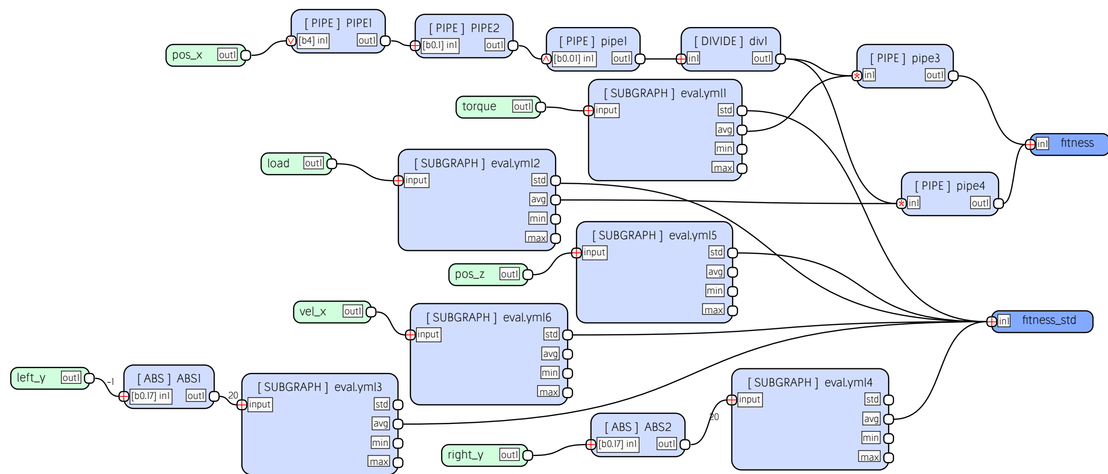


Figure 8.36: The evaluation graph used to generate a locomotion pattern for the front leg pair of CHARLIE.

Result The finally evolved joint patterns, selected as starting point for the generation of the whole walking controller, are shown in Figure 8.37. The evolution generated patterns for the shoulder pitch and the knee joint, while not utilizing the shoulder's roll and yaw joints.

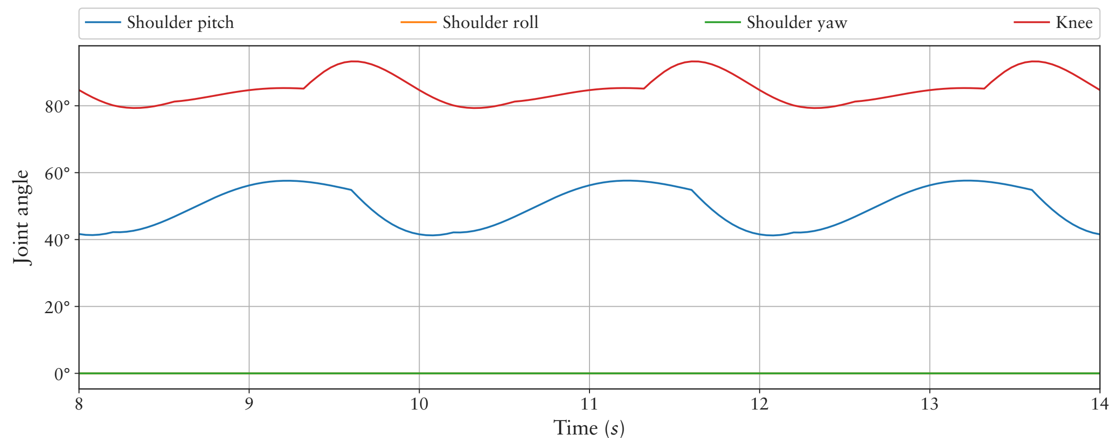


Figure 8.37: The evolved patterns of the front legs selected as start point for the whole CHARLIE system.

REARLEGS

The setup to generate a starting behavior for the rear leg pair of CHARLIE is designed similar to the one for the front legs.

Environment The simulation setup is similar to the setup used for the front legs (see Figure 8.38).

Evaluation Preliminary performed optimizations to test the setup have identified a tendency of producing behaviors where only the tip or the heel of the foot is used in the stance phase instead of the whole foot. To eliminate this tendency the pitch angle of the

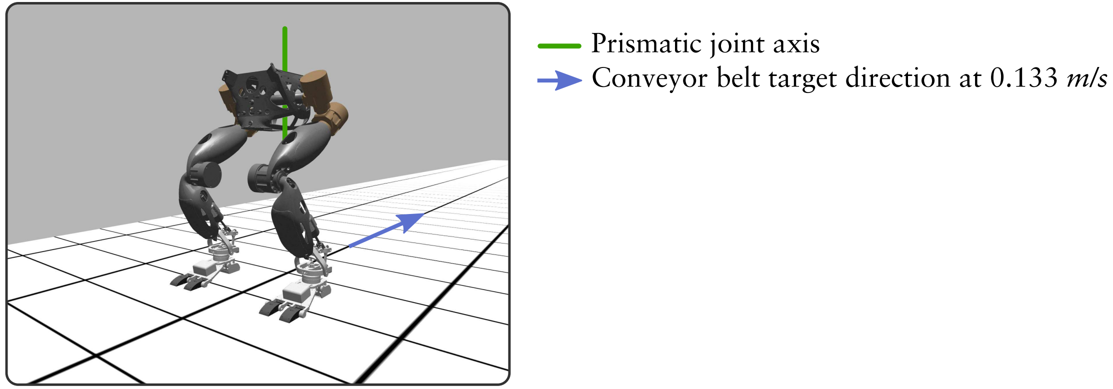


Figure 8.38: The simulation setup to generate a walking behavior for the rear leg pair of CHARLIE.

left foot orientation is minimized in the contact phase by an additional evaluation objective. Due to the symmetry constraint evaluating only one foot is sufficient. A further empirically chosen adaptation, based on the preliminary performed experiments, is done by increasing the weights of the objectives evaluating the distance reached. The resulting evaluation graph is shown in Figure 8.39. Each individual is tested in the simulation for a time period of 30 *s*.

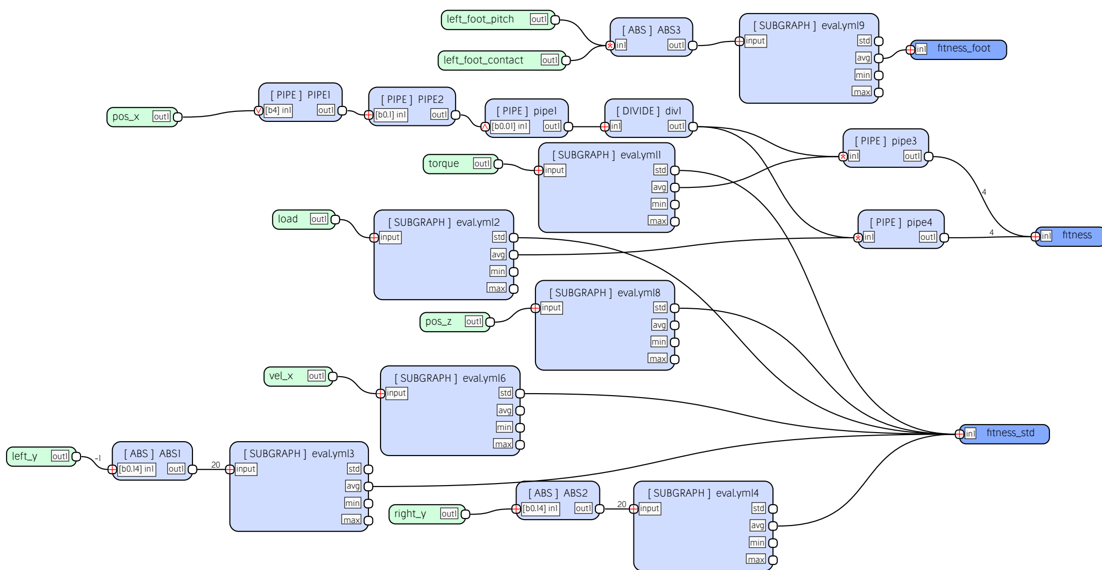


Figure 8.39: The evaluation graph used to generate a locomotion pattern for the rear leg pair of CHARLIE.

Configuration The general behavior configuration is the same as used in FRONTLEGS. The predefined BAGEL modules of FRONTLEGS are extended by the ankle pitch and roll joints of the rear legs.

Result The resulting joint patterns of a finally performed evolution are presented in Figure 8.40. Similar to FRONTLEGS the resulting controller utilizes the pitch and knee joints of the legs to move the treadmill.

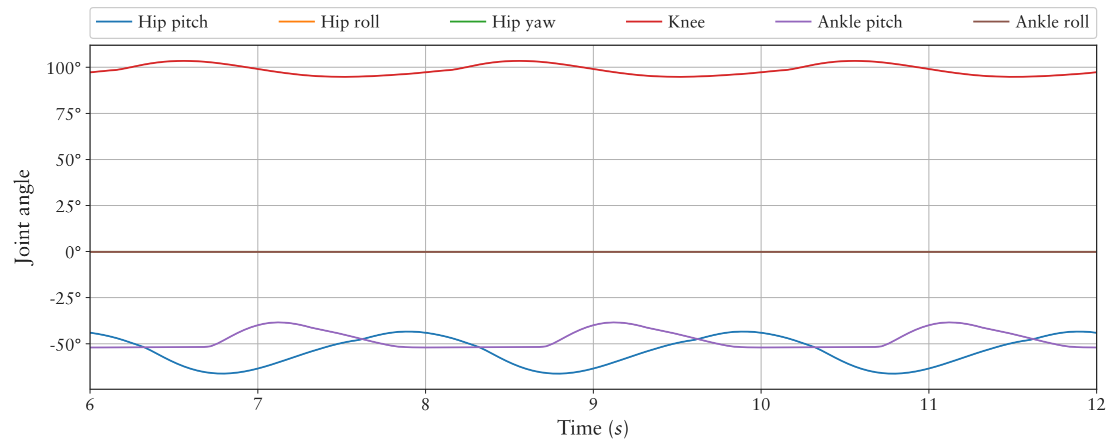


Figure 8.40: The evolved patterns of the rear legs selected as start point for the whole CHARLIE system.

OPENLOOP

This setup combines the evolved front and rear leg behaviors of FRONTLEGS and REARLEGS for a control of the whole CHARLIE system. The previous evolved patterns are used as starting point for the legs while the phase sift and patterns for the six *dof* of the spine are evolved from scratch. Again individual preliminary experiments are performed to test and adapt the overall setup.

Environment The robot has to walk in forward direction (*x*-axis) on a planar surface with a Coulomb friction of 0.8.

Evaluation In preliminary performed experiments is could be observed that the stability fitness (see Section 8.1.1), that is limiting the exploration of pattern, is counterproductive for generating a controller for CHARLIE. The final evaluation graph is shown in Figure 8.41. Additionally to the reduced objectives in the evaluation graph the influence of the feet slippage evaluation is reduced by a factor of 0.1. As done in the experiments with SPACECLIMBER and SPOT the evaluation is aborted if other elements of the robot than the feet are involved in a collision – AbortOnContact configuration of `locomotion_environment`. Each individual is evaluated for a time period of 30 *s* if the evaluation is not aborted due to an unintended collision.

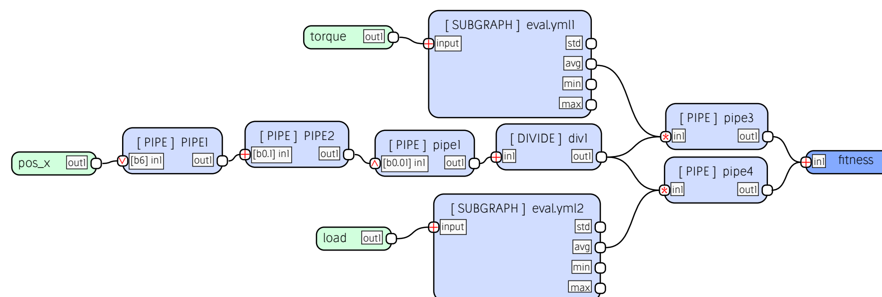


Figure 8.41: The evaluation graph used to generate the controller of the whole CHARLIE system.

Configuration The configuration is based on the experiment for the development of the open-loop controller of EASY4 (Section 8.1.1). Additionally to that configuration a control structure for the spine *dof* of CHARLIE is added. The top-level control graph is shown in Figure 8.42. To introduce the symmetry information of CHARLIE into the spine control, the spine pattern is executed twice within the control cycle, while the spine roll, yaw, and y movement is inverted for the second execution. To compensate for an possible hard value change, due to inverting three of the spine *dof* within the pattern, a filter smoothing the spine pattern is implemented. Beside the filter for the spine patterns, boundaries are integrated for the spine *dof* and the ankle pitch joints.

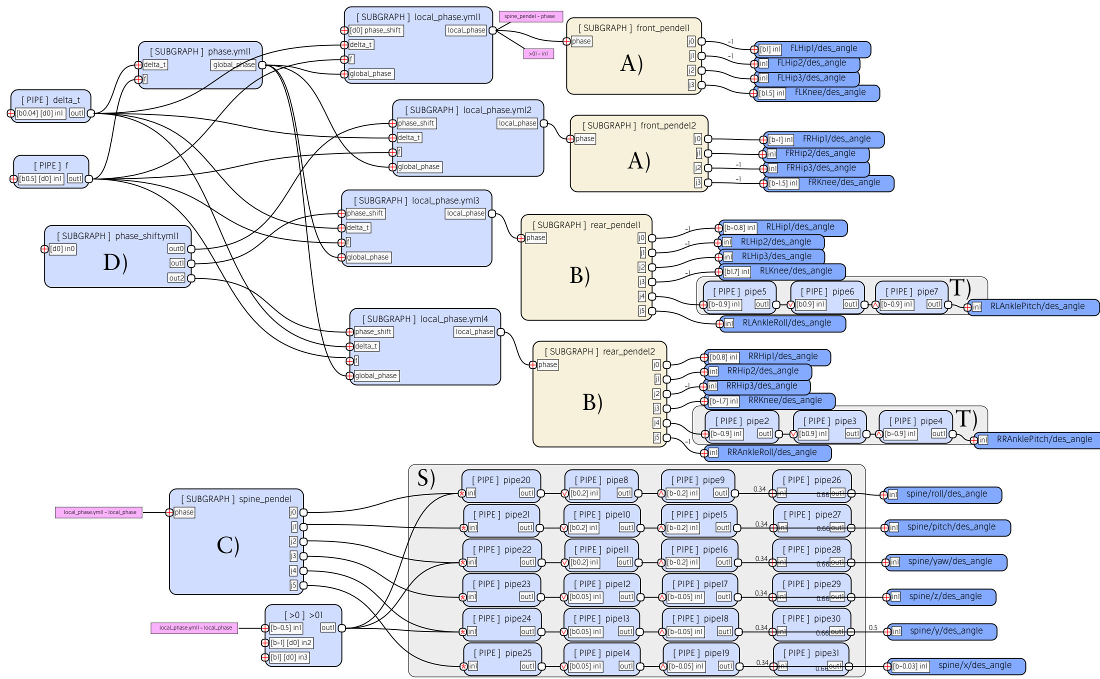


Figure 8.42: The top-level control graph for the open-loop controller of CHARLIE. The nodes A) and B) are evolved starting the evolution with the results of FRONTLEGS and REARLEGS. Node C) evolves a pattern for CHARLIE's spine *dof*. The parameters defining the phase shift are evolved in node D). The nodes included in group T) implement an offset and boundaries of the ankle pitch joints. The nodes of group S) handle boundaries of the spine *dof* and add a filter to smoothen the spine patterns.

Result A final result is generated and tested on the real system. Figure 8.43 compares the resulting joint patterns of the legs with the initial patterns generated in FRONTLEGS and REARLEGS. The results show, that not only small adaptations to the starting patterns are performed but also new features, such as utilizing the hip roll and yaw joint, can be evolved. Furthermore, the starting pattern are evolved for a target speed of $0.133\,m/s$ while the evaluation graph for the whole system is configured for a speed of $0.2m/s$. As result of this difference the amplitude of the starting pattern is increased through the optimization process. The resulting pattern produced for the *dof* of the spine is shown in Figure 8.44. Even though the patterns of the spine have a low amplitude, they significantly influence the workspace of the legs and the resulting performance of the locomotion behavior. A resulting image sequence of the walking pattern executed on the real system is shown in Figure 8.45. The patterns produce a stable forward locomotion of CHARLIE utilizing the system properties similarly as in the simulation.

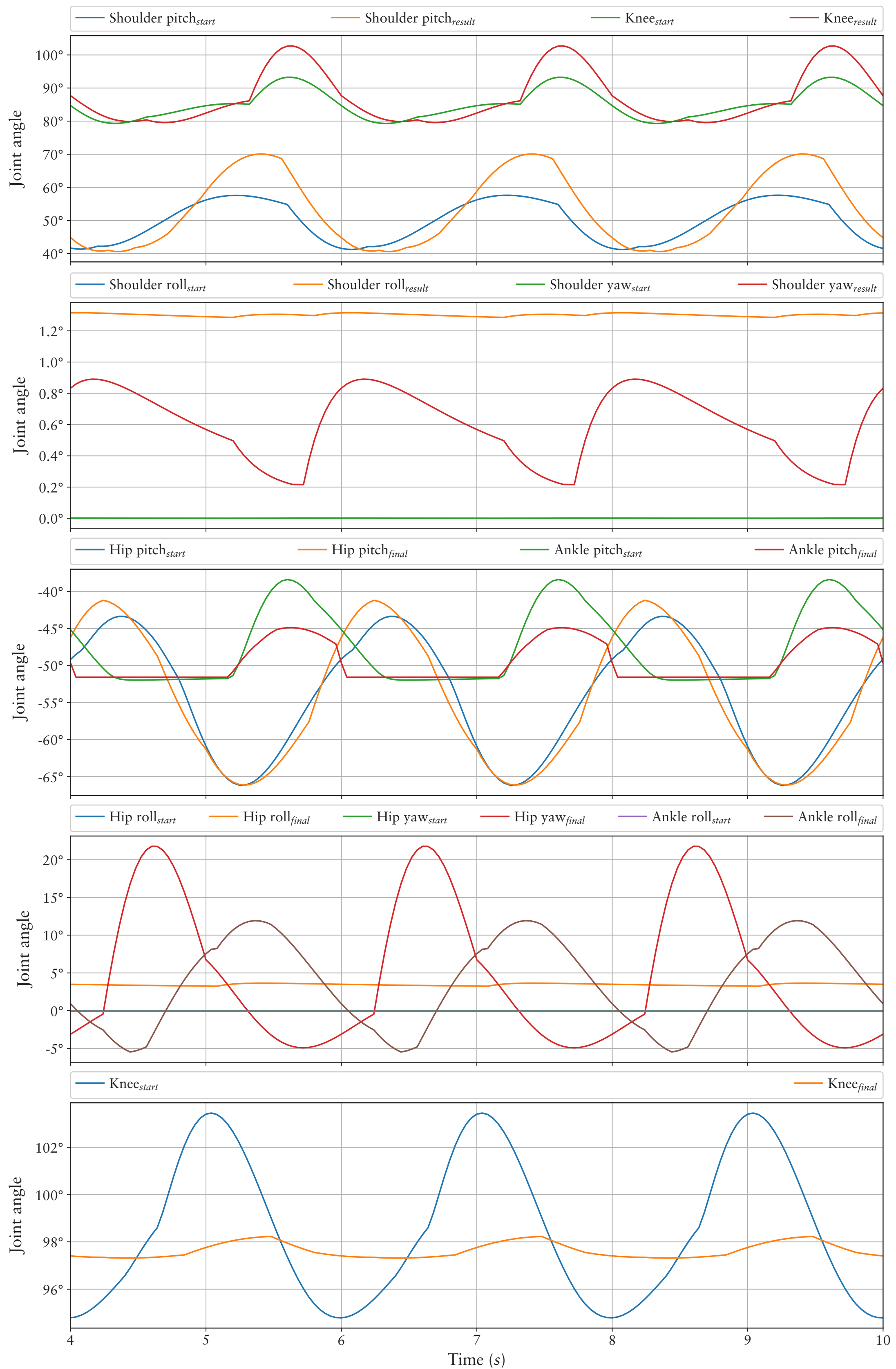


Figure 8.43: Comparison of the start patterns with the resulting adapted patterns of the whole CHARLIE system. For the front legs (the two upper graphs) the amplitude of the shoulder pitch and knee patterns is increased and a pattern with a low amplitude is generated for the shoulder roll and yaw joint. In case of the rear legs (the lower three plots) the amplitude of the knee pattern is strongly reduced while a pattern for the hip roll and yaw joint is generated. These roll and yaw patterns influence the swing movement lifting the feet to the outer side and thus generating a momentum supporting the lift phase of the diagonal front leg. The amplitude of the pattern for the ankle pitch joint is reduced while the amplitude of the hip pitch joint is only slightly adapted.

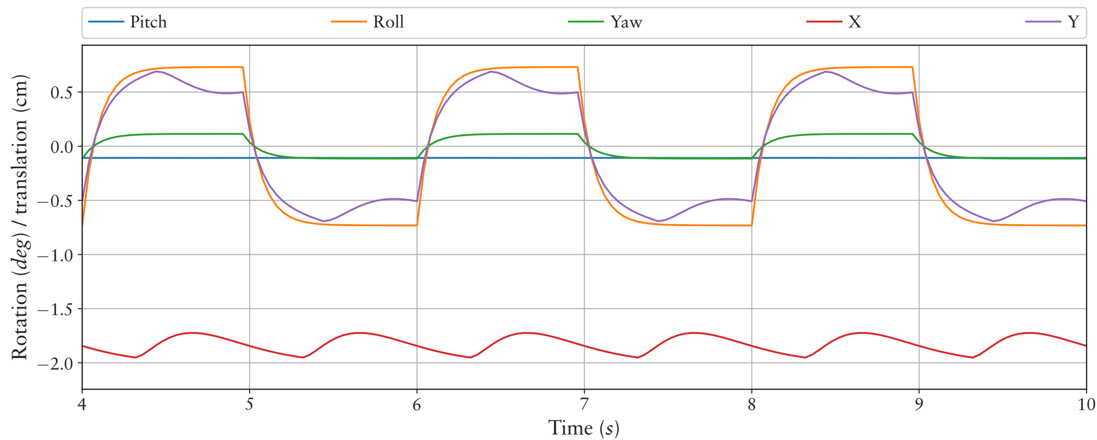


Figure 8.44: The resulting patterns generated for CHARLIE's spine *dof*.

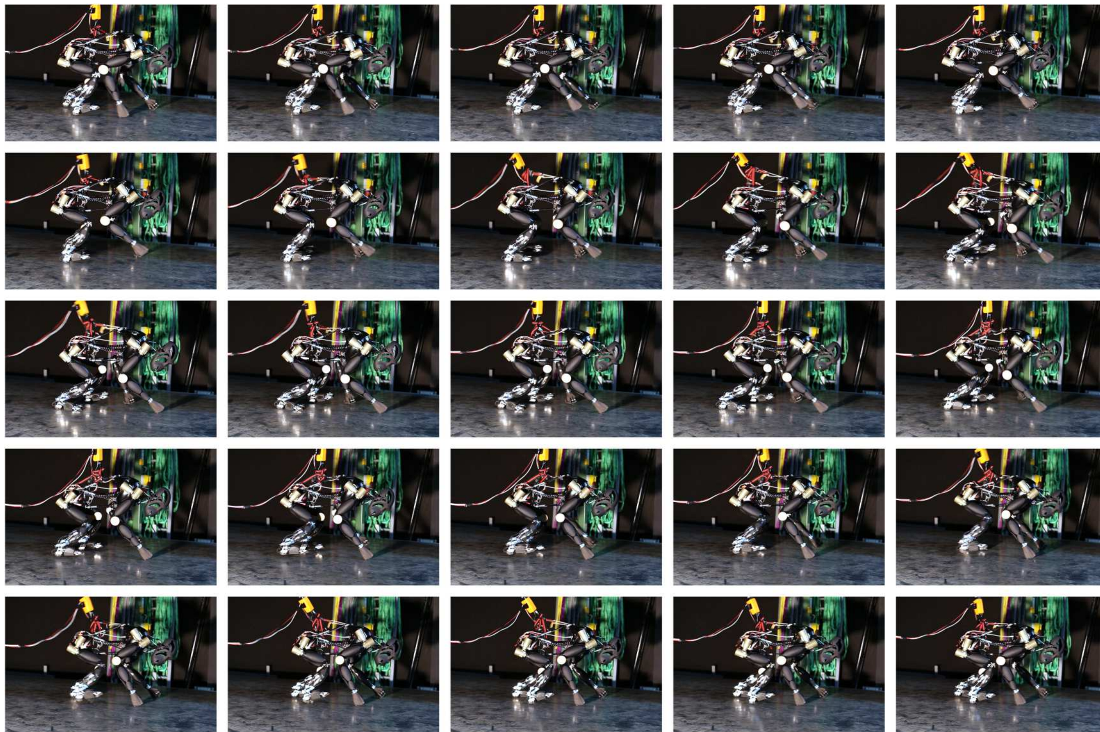


Figure 8.45: The image sequence depicts the resulting walking pattern of CHARLIE transferred to the real system on the locomotion test-bed.

CLOSEDLOOP

This experiment is designed to show the possibility of stabilizing CHARLIE by a sensory driven modulation of the evolved open-loop controller. The general concept is similar to the stabilizing controller evolved in Section 8.1.3. In contrast to that experiment the CHARLIE robot is more complex and it has to be shown that the concept can deal with the additional complexity.

Environment The robot has to walk in forward direction (x-axis) on a planar surface similar to the previous experiment in OPENLOOP. The environment is extended by a

MARS plugin applying a force pattern to the robot's front body. The disturbing forces are defined in the robot coordinate system along the lateral (y) axis. The disturbing forces are applied to the robot for a period of 0.5 s. A first disturbance of 50 N is generate 8 ± 0.2 s in simulation time after the evaluation of an individual is started. A second disturbance into the opposite direction (-50 N) is generate at the time of 12 ± 0.2 s.

Evaluation The evaluation is done with the same function and parameters as in the previous experiment evolving the open-loop controller in OPENLOOP. Each individual is tested in the simulation for a time period of 30 s.

Configuration The configuration for this experiment is based on the configuration for the stabilize controller of EASY4 (8.1.3). The top-level control graph is shown in Figure 8.46. In contrast to the stabilize controller for EASY4 the phase frequency adaptation is removed from the configuration and for each leg an individual reactive module is set up. The adaptations are done to restrict also the stabilize controller to symmetric behaviors.

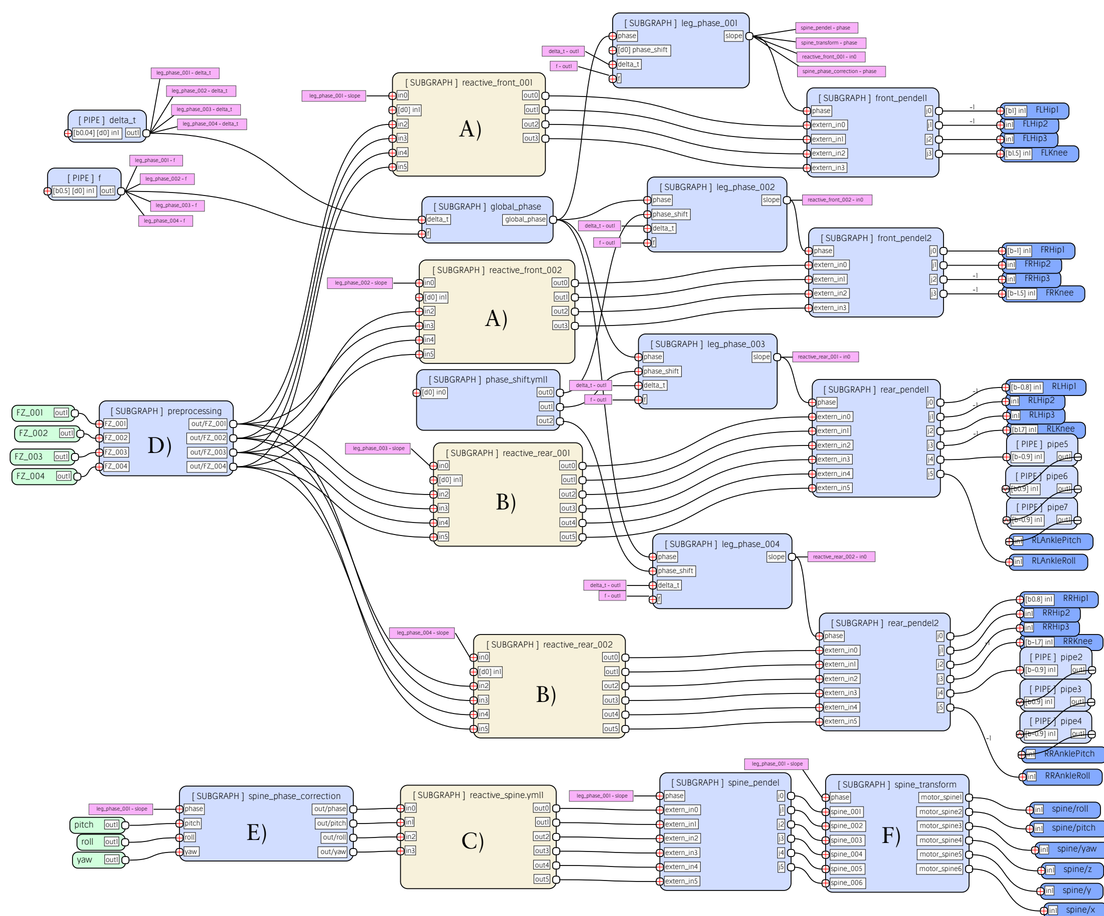


Figure 8.46: Top-level control graph for the stabilize controller of CHARLIE. The sub-graphs A), B), and C) are evolved to process the leg phase and sensory data to stabilize CHARLIE by modifying the open-loop patterns. Node D) multiplies the force sensor values with a factor of 0.01 to normalize them into a range of $[-1, 1]$. Node E) handles the doubled spine pattern frequency similar as described for the spine pattern of OPENLOOP. The spine pattern transformation introduced for OPENLOOP is encapsulated in node F).

Result A single result is generated for this experiment. The evolved controller is compared to the original open-loop controller which is the basis of the evolved reactive controller. Therefore, each controller is tested for a period of 60 *s* in the simulation with an adapted force profile shown in Table 8.8. This test is repeated 1000 times for both controllers providing a statistical comparison of their performances. The result of this comparison is summarized in Table 8.9. While the initial open-loop controller does not persist more than one disturbance, the evolved controller persists the whole force pattern in 37% of the trials. In average the evolved controller is evaluated for 32.1 *s* which is in between the last two disturbances.

Table 8.8: The force profile used to compare the evolved stabilize controller of CHARLIE with the initial open-loop controller.

Time (s)	Duration (s)	Lateral force (N)
5 ± 0.2	0.5	50
8 ± 0.2	0.5	50
12 ± 0.2	0.5	-50
14 ± 0.2	0.5	-50
20 ± 0.2	0.5	50
24 ± 0.2	0.5	50
28 ± 0.2	0.5	50
36 ± 0.2	0.5	50

Table 8.9: Statistics of the comparison between the initial open-loop controller and the resulting reactive controller of CHARLIE. Each controller is tested for a period of 60 *s* with a force profile of eight disturbances. An individual test is aborted if the controller does not persist a disturbance – if the robots falls over. A test is successful if the controller persists all disturbances, resulting in an evaluation time of 60 *s*.

	Initial open-loop controller	Evolved reactive controller
Successful test	0%	37%
Average evaluation time (s)	6.7	32.1
Max evaluation time (s)	8.8	60.0
Min evaluation time (s)	6.3	7.1

Figure 8.47 plots the modulation of the shoulder generated by the reactive module. The reactive module produces a linear mapping between the force sensor and the modulation signal, resulting in a continuous adaptation of the behavior to the environment. A similar adaptation is generated for the z-axis of the spine, while the hip roll angle and x-axis of the spine is modulated by the single pulses of generated `fire2_neuron` modules.

8.2.4 Conclusion

The experiments presented in Section 8.1.1 have shown that the Pendulum pattern approach produces superior results compared to the Gaussian and the Fourier approaches. Additionally, a navigation and stabilizing controller was evolved for the simulated robot EASY4. The relatively simple Sine-Based approach was used to generate locomotion pattern for the SPOT and the SPACECLIMBER system. However, it was configured especially for the target systems and thus cannot be used as general method. For the SPACECLIMBER

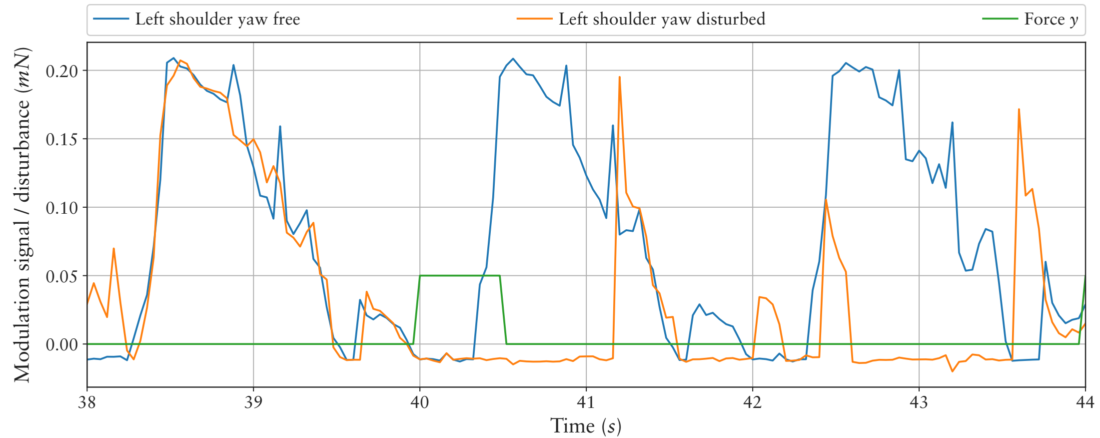


Figure 8.47: Comparison of the modulation signal for the left shoulder generated with and without a disturbing force applied to CHARLIE's front body.

system also the Pendulum approach was used. The Pendulum approach was configured by a more general method, while first evolving control patterns for the single legs and afterward the control of the whole system by starting the optimization with the single leg patterns. A similar method was applied to the CHARLIE system. The main difference in the configuration setup for CHARLIE was the fact that the starting pattern were evolved for the leg pairs and not for the single legs. For all three test robots control patterns could be generated in the simulation software that could be transferred to the real systems. At least in the simulation setup a sensory feedback loop was used to improve the locomotion control for the complex CHARLIE system. Even though, the general method used for all systems was similar, small adaptations mainly to the evaluation function were necessary to improve the transferability of the evolved controllers.

Chapter 9

Conclusion and Outlook

As described in the introduction (Section 1.1) the control of legged robots poses a complex problem. Solving this problem by utilizing optimization algorithms is an interesting approach and provides insights into two main questions. The first question *Q1* is how well optimization algorithms can deal with the complexity of the problem. The second question *Q2* is whether a learned controller can have the same or even better quality then an human-designed controller. This thesis contributes to the answer to these questions by investigating how to achieve the main goal of this thesis:

> **Goal:** Reduce the search space for evolving locomotion controllers by: 1. predefining control modules that limit the control possibilities to a useful range and can still be used by all systems. 2. defining an iterative method with increasing complexity instead of dealing with the whole search space at once.

To reach the defined goal, a set of tools is developed to perform reproducible and comparable experiments. The first tool is the modular simulation software MARS that focuses to optimizing the trade-off between accuracy and computation cost. Additionally, its software architecture is especially designed for the use in evolutionary applications. For several robotic systems the behaviors developed in the simulation could be transferred to the real systems resulting in a behavior with similar quality regarding energy efficiency. The second tool is the benchmarking framework BOLERO that is designed to perform reproducible benchmarking experiments for the evolutionary development of robotic controllers. Together with BOLERO a set of tools and benchmarks is developed with focus on the generation of controllers for legged robots. While BOLERO is used to define the learning environment (the problem description) the behavior representation BAGEL is developed to provide a graph-based data flow language that can be used to predefine control structures and modules for the behavior development. The genetic algorithm SABRE is developed to make use of the foreknowledge and to precisely configure how the evolution adapts to the behaviors. It is shown that the concept of SABRE to manage multiple populations for different behavior structures performs as well as comparable existing methods for a set of selected benchmark problems.

The MARS simulation and the BOLERO framework are already available as open source projects (see Langosz (2018); Fabisch and Langosz (2018)). The open source projects for the tools BAGEL and SABRE are in preparation, too. Once BAGEL and SABRE are also published as open source projects, all tools will be available as single "binary" application, that can be installed via drag-and-drop to be used by none computer science experts. Additionally, the benchmarks performed for this thesis will be included in the binary package.

Several experiments are presented to finally define a suitable control structure, building blocks, and an iterative process to evolve the control of legged locomotion. It is shown that the general methodology can be used to generate the locomotion control for different four-legged systems (two real systems and one simulated test system) and one real six-legged system. With the simulated test system it is shown that modifying the locomotion direction and stabilizing the robot by processing sensory data is possible with the purposed control structure. However, for the real systems only the forward locomotion is generated and further investigations are required to generate a holistic locomotion controller for a kinematically complex system. Due to the structural complexity and the required controller precision of SPACECLIMBER (six legs, 24 *dof* for walking) and CHARLIE (four legs, 32 *dof* for walking) it is expected that the developed methodology can be applied to other legged systems as well.

Coming back to the first question *Q1*, how well the optimization algorithms can deal with the complexity of the locomotion problem, different challenges are involved that can be nailed down to the structure and size of the search space and fitness landscape. Thereby, the choice of a representation for the controller defines the search space. On one hand, the controller, which parameters are optimized by a local search algorithm, can be developed for a specific system. On the other hand, a representation such as ANNs is a very generic controller whose structure and parameters can be optimized. A controller designed for a specific system allows to design the complexity for a local search algorithm and therefore does not pose a generic solution. Thus such an approach is less important concerning the two questions *Q1* and *Q2*. By evolving control patterns only based on ANNs without further limiting the search space via predefined structures it is not possible to generate adequate solutions for kinematically complex robots. The solution presented in this thesis lies in between these two approaches by defining general control structures that require only the knowledge about symmetry properties of the target system. This solution generates walking patterns that have a similar performance compared to the originally developed model-based controllers of the three robots SPOT, SPACECLIMBER, and CHARLIE.

Considering question *Q2*, whether a generated controller can have the same or even better quality than an implemented controller, the results presented in this thesis indicate that at least a similar quality can be achieved. However, for the comparison of the performance only a sub-part of the locomotion control – walking straight ahead – is compared and it still have to be shown that a holistic controller can be generated by ML techniques with a quality comparable to model-based controllers. The holistic controller would have to deal with desired direction changes of the robot (needed for navigation), transitions of different walking gaits, and an adaptation to the environment.

With the predefined structures and the iterative method presented in this thesis the evolutionary approach is able to deal with the complexity of the three existing robots SPOT, SPACECLIMBER, and CHARLIE. To allow the evolutionary approach to generate adequate solutions without the need of the predefined structures one have to come up with a setup in which these structures can be evolved before the evolutionary process faces the complexity of solving the locomotion of the whole system. A very important difference of natural evolution compared to the locomotion problem of existing robots is the fact, that the predefined structures in nature were evolved together with the morphology development. This concept might be copied by subdividing the morphology of the target system in a hierarchical structure starting with a single actuator on the lowest level. On every level the evolution could generate skills that are based on the evolved skills of the next lower layer. This hierarchical concept could generate the predefined structures automati-

cally. However, there are many possibilities how to define the hierarchical structures and the evolution of skills. On one hand, the predefined structures of this thesis help understanding the requirements on such a setup. On the other hand, performing a hierarchical evolution could give insights into how to adapt these predefined structures to increase the performance of the method presented in this thesis, especially for evolving more holistic solutions providing more integrated sensor coupling.

Two main challenges arise when generating locomotion controllers via ML methods in simulation for real systems. The first challenge is to define the evaluation function. On one hand, when using a rather simple evaluation function it is pretty likely that many solutions exist that perform well in relation to the evaluation function but would stress the real system. On the other hand, more complex evaluation functions penalizing these solutions could limit the exploration of the search algorithm and result in solutions with poor performance. Another option is to limit the search algorithm via the behavior representation and the search step size to generate desired solutions. However, depending on the evaluation function and behavior representation, the search algorithm might have to overcome a plateau with poor fitness to find the desired optimum, which stays in contrast to limiting the search algorithm. The second challenge is to deal with the simulation-reality gap. The mechanical stress on a system is normally only roughly simulated and thus cannot be treated with the evaluation function directly. Eventually, to apply ML methods to the problem on generating locomotion controller for complex legged robots, a deep understanding of these challenges is required. The tools, controller modules, and methods developed for this thesis should pose a good starting point to tackle these challenges for a new robot.

If the goal is to develop a holistic locomotion controller for an existing robot, in most cases designing a model-based solution is the more promising approach. It still has to be shown that a generated controller is able to provide all features of a model-based approach. These features include, amongst others, different locomotion gaits and an adaptation to the context given by the environment and target properties of the locomotion behavior such as desired speed, stability, or energy efficiency.

To continue the research on evolving locomotion controllers for real complex legged robotic systems different objectives could be tackled. The following list includes selected recommended objectives:

- **Statistically Results**: Many experiments on the development of legged locomotion only show the general potential of the method by providing a single positive result. The experiments could be repeated to give an idea about the probability that these results can be generated.
- **Pendulum Approach**: The Pendulum approach is finally presented as the most promising approach. Further investigation could improve this approach.
  - Further investigations could increase the representation strength concerning the joint pattern approximation.
  - The parameters of `pendel_control`, such as pendulum length and mass, could be derived from the kinematic properties of the system.
  - The damping defined for the pendulum could be replaced by spring-damper properties.
  - By using force-torque measurements of the robot the virtual gravity used in the pendulum controller could be replaced by the measured gravity, which already

provides a coupling of the controller with the environment without the need of explicitly evolving this coupling.

- **Sensor Coupling**: Reacting to sensory input is feasible with the Pendulum approach but needs more investigation in setting up scenarios as generally as possible.
  - It could be interesting to define a set of "sandbox" experiments for the sensory coupling, similar to the experiments in Chapter 7.
  - For the experiments performed in this thesis, constant reactions to sensory values measured at defined points in the step phase are composed. If, for instance, the sensors have to be evaluated over the whole stance phase, many measurement points have to be generated. The sensory coupling could be improved if a time slot is defined for the measurement instead of the exact time point. Similarly, if for example a linear reaction force to the sensory value is required multiple measurement points have to be composed, discretizing this function. These measurements could be merged to a single one if a projection from the measured sensor value to the reaction strength is possible.
  - Learning the "open-loop" pattern and the sensory coupling at once could improve the base pattern (behavior without disturbances) and already provides an adaptation to the environment even without enforcing it by the simulation setup.
  - It could be interesting to use the profile of the local environment as sensory input to modulate the locomotion pattern. It would allow the controller to adapt the touchdown position of the feet and already adapt the pattern to the environment before it is measured by force or orientation sensors. To learn this modulation deep reinforcement learning could be considered as a learning method.
- **Simulation**: The simulation could be enhanced to increase the probability of transferable results.
  - Joints that break on a defined load could be generated at potential breaking points of the real system.
  - The force distribution on the rigid bodies could be measured / simulated.
  - Simulating the motor currents and gear friction would influence the energy evaluation.
  - Simulating "pain" could allow to filter undesired behaviors in a more direct way than optimizing for mechanical stress in general.
- **Character Animation**: A great progress is done in the field of character animation, including model-based, optimization, and deep learning approaches. It should be evaluated how the problem of evolving locomotion controller for complex real legged robots can benefit from this progress.
  - Apply the Pendulum approach on character animation problems to compare it to results in this area.
  - Apply the methods of the character animation area to a simulation of a real robotic system.
- **Fitness Landscape**: Different methods could be used to influence the fitness landscape in order to improve the locomotion behaviors that can be evolved.

- The evolution could start with a less challenging environment, which for example can be achieved with a reduced gravity. Then the gravity could be increased over time or depending on the performance of the evolved behaviors.
- Randomly changing the weights of a multi-objective evaluation function in relation to the evolutionary progress could generate more generic behaviors.
- The evaluation function could be composed of a set of subgoals. For instance the following subgoals could be considered:
  1. The distance reached can be the first evaluation objective.
  2. If a defined distance is reached, the body movement is optimized (low movement on height axis and constant on forward axis).
  3. If the second goal has a defined performance, the evolution proceeds by optimizing the energy efficiency.

With the current state the question arises why to continue researching on ML methods to solve the locomotion problem of legged robots. Regarding this question different aspects have to be considered. One aspect is the possibility to generate locomotion solutions for specific requirements that can be implemented and used beside a holistic model-based control. This becomes interesting if for instance no sufficient model of the environment can be generated. Another aspect is the possibility to adapt the locomotion behavior while the system is operating. Given a legged robotic system that is operating on a day-by-day basis and can measure its own performance, the techniques presented in this thesis could be used for an online adaptation of the controller. A further aspect is the general research on ML methods on complex problems. If the performance of a ML method can be improved on a problem, even though the problem can be solved with a different approach, it might help improving the understanding of the ML method and to increase the performance also on other problems.

Appendix A

Appendix

A.1 Configurations

A.1.1 SPACECLIMBER

The default parameters of the SPACECLIMBER model-based control:

```
# ======= General Notes =========
# 1. The motion control is based on central pattern generator (cpg), which
#    generates a pulse which is shifted for every leg, so that cyclic walking
#    pattern is generated. A Cartesian posture is defined and superposed by a
#    cyclic motion around. This generated open loop trajectories are adapted by
#    reactive behaviors or reflexes to adapt to the environment.
# 2. The robot frame is fixed to the robot, x pointing to the front, z upwards,
#    and y creating a right-handed coordinate system.
# 3. All parameters are defined in the locomotion frame which is the the center
#    of footprint thus parallel to an ideal plane ground (which can also be
#    inclined).
#    -> So a tilted robot is still walking parallel to the ground.
# 4. All units are SI units (m, rad, N, ...) or explicitly stated if not.

# ======= Constant Parameters (which should stay constant) =========

# update period in s
time_diff: 0.04

# thresholds for contact estimation -> 2 of 3 must be true
# force in N, torque in Nm, pressure has no unit
# for sim
#contact/pressure_th: 2.0
#contact/force_th: 4.0
#contact/torque_th: 2.0
# for real
contact/pressure_th: 12.0
contact/force_th: 20.0
contact/torque_th: 3.0


# ====== Posture Parameters =======

# foot print in longitudinal and lateral direction in m (will be applied during
# swing phase to avoid counteracting forces)
step_base/x: 0.9
step_base/y: 0.85

# translation of robot frame in locomotion frame in m
body_shift/x: 0.0
body_shift/y: 0.0
body_shift/z: 0.23
```

```
43
44 # rotation between locomotion frame and robot frame in rad
45 body_rot/roll: 0.0
46 body_rot/pitch: 0.0
47 body_rot/yaw: 0.0
48
49 # rotation of lean joint of each limb in rad (around y-axis of robot frame)
50 lean: 0.0
51
52 # allows to modify each limb position in locomotion frame in m (applied during
53 # swing phase)
54 limb0/pos_offset/x: 0.0
55 limb0/pos_offset/y: 0.0
56 limb0/pos_offset/z: 0.0
57 limb1/pos_offset/x: 0.0
58 limb1/pos_offset/y: 0.0
59 limb1/pos_offset/z: 0.0
60 limb2/pos_offset/x: 0.0
61 limb2/pos_offset/y: 0.0
62 limb2/pos_offset/z: 0.0
63 limb3/pos_offset/x: 0.0
64 limb3/pos_offset/y: 0.0
65 limb3/pos_offset/z: 0.0
66 limb4/pos_offset/x: 0.0
67 limb4/pos_offset/y: 0.0
68 limb4/pos_offset/z: 0.0
69 limb5/pos_offset/x: 0.0
70 limb5/pos_offset/y: 0.0
71 limb5/pos_offset/z: 0.0
72
73 # ====== Motion Parameters =======
74 # each step cycle is divided into a swing (lift + shift + down) and a stance
75 # phase (which moves the robot over the ground)
76
77 # if one of the following three params is not zero, movement is desired and the
78 # internal clock of the cpg is ticking turn angle during one step cycle in rad
79 turn_rate: 0.0
80
81 # length of a step during one step cycle in m
82 step_length/x: 0
83 step_length/y: 0
84
85 # height during shift phase in m
86 step_length/z: 0.1
87
88 # step cycle time in s
89 step_time: 5.0
90
91 # time for each phase in percent/100 of a step cycle
92 lift_time: 0.1
93 shift_time: 0.12
94 down_time: 0.16
95
96 # linear or non-linear (first fast, at the end slow) movement in downphase
97 non_linear_down: 0.0
98
99 # phase shift between leg movement in percent/100 (1.0 -> single leg movement,
100 # 0.0 -> tripod or cross gait)
101 phase_shift: 0.5
102
103 # gait which defines the order of leg movement (1-4 for 6 limbs, 5 for 4 limbs)
104 gait: 2.0
105
106
107 # ====== Hole Reflex ===============
108 # and early touchdown (or elevation and depression reflex)
109 # stops the down movement when an obstacle is detected or moves further when in
110 # stance phase no ground is detected
111
112 # 0.0 -> deactivated, 1.0 -> activated
```

```
113 hole_reflex/active: 0.0
114
115 # speed of stretching the leg, if no ground was detected in m per time step
116 hole_reflex/sensitivity: 0.003
117
118 # speed of recovering to original trajectory in m per time step
119 hole_reflex/retraction: 0.008
120
121 # stretching is delayed by x time steps before beginning to stretch
122 hole_reflex/touchdown_delay: 2.0
123
124 # limits the stretch to this value in m
125 hole_reflex/max_foot_dist_z: 0.6
126
127 # does not stop down movement even when an obstacle was detected until this
128 # distance in m was moved
129 hole_reflex/min_amp_z: 0.02
130
131
132 # ====== SpaceClimber constants ===========
133 kinematics/shank_front: 0.333
134 kinematics/shank_middle: 0.355
135 kinematics/shank_rear: 0.336
```

A.1.2 CEC13 Benchmark Functions

Table A.1: Benchmark functions used in the parameter optimization competition at the CEC13 conference.

	No.	Function	$f_i^* = f_i(x^*)$
Unimodal Function	1	Sphere	−1400
	2	Rotated High Conditioned Elliptic	−1300
	3	Rotated Bent Cigar	−1200
	4	Rotated Discus	−1100
	5	Different Powers	−1000
Basic Multimodal Functions	6	Rotated Rosenbrock's	−900
	7	Rotated Schaffers F7	−800
	8	Rotated Ackley's	−700
	9	Rotated Weierstrass	−600
	10	Rotated Griewank's	−500
	11	Rastrigin's	−400
	12	Rotated Rastrigin's	−300
	13	Non-Continuous Rotated Rastrigin's	−200
	14	Schwefel's	−100
	15	Rotated Schwefel's	100
	16	Rotated Katsuura	200
	17	Lunacek Bi_Rastrigin	300
	18	Rotated Lunacek Bi_Rastrigin	400
	19	Expanded Griewank's plus Rosenbrock's	500
	20	Expanded Schaffer's F6	600
Composition Functions	21	Composition Function 1 (n=5, Rotated)	700
	22	Composition Function 2 (n=3, Unrotated)	800
	23	Composition Function 3 (n=3, Rotated)	900
	24	Composition Function 4 (n=3, Rotated)	1000
	25	Composition Function 5 (n=3, Rotated)	1100
	26	Composition Function 6 (n=5, Rotated)	1200
	27	Composition Function 7 (n=5, Rotated)	1300
	28	Composition Function 8 (n=5, Rotated)	1400

A.1.3 NEAT

The default parameters used for the NEAT algorithm.

```
##########
# My parameters
##########

outputScale: 1.0

##########
# Basic parameters
##########

# Size of population
PopulationSize: 130

# If true, this enables dynamic compatibility thresholding
# It will keep the number of species between MinSpecies and MaxSpecies
DynamicCompatibility: true

# Minimum number of species
MinSpecies: 5

# Maximum number of species
MaxSpecies: 20

# Don't wipe the innovation database each generation?
InnovationsForever: true

AllowClones: true



################
# GA Parameters
################

# Age threshold, meaning if a species is below it, it is considered young
YoungAgeThreshold: 5

# Fitness boost multiplier for young species (1.0 means no boost)
# Make sure it is > 1.0 to avoid confusion
YoungAgeFitnessBoost: 1.1

# Number of generations without improvement (stagnation) allowed for a species
SpeciesDropoffAge: 15

# Minimum jump in fitness necessary to be considered as improvement.
# Setting this value to 0.0 makes the system to behave like regular NEAT.
StagnationDelta: 0.0

# Age threshold, meaning if a species is above it, it is considered old
OldAgeThreshold: 30

# Multiplier that penalizes old species.
# Make sure it is < 1.0 to avoid confusion.
OldAgePenalty: 1.0

# Detect competetive coevolution stagnation
# This kills the worst species of age >N (each X generations)
DetectCompetetiveCoevolutionStagnation: false
# Each X generation..
KillWorstSpeciesEach: 15
# Of age above..
KillWorstAge: 10

# Percent of best individuals that are allowed to reproduce. 1.0  100%
SurvivalRate: 0.15
```

```
66
67 # Probability for a baby to result from sexual reproduction (crossover/mating). 1.0  100%
68 # If asexual reproduction is chosen, the baby will be mutated 100%
69 CrossoverRate: 0.25
70
71 # If a baby results from sexual reproduction, this probability determines if mutation will
72 # be performed after crossover. 1.0  100% (always mutate after crossover)
73 OverallMutationRate: 0.25
74
75 # Probability for a baby to result from inter-species mating.
76 InterspeciesCrossoverRate: 0.05
77
78 # Probability for a baby to result from Multipoint Crossover when mating. 1.0  100%
79 # The default is the Average mating.
80 MultipointCrossoverRate: 0.6
81
82 # Performing roulette wheel selection or not?
83 RouletteWheelSelection: false
84
85
86
87
88
89 #################/
90 # Phased Search parameters   #
91 #################/
92
93 # Using phased search or not
94 PhasedSearching: false
95
96 # Using delta coding or not
97 DeltaCoding: false
98
99 # What is the MPC + base MPC needed to begin simplifying phase
100 SimplifyingPhaseMPCThreshold: 20
101
102 # How many generations of global stagnation should have passed to enter simplifying phase
103 SimplifyingPhaseStagnationThreshold: 30
104
105 # How many generations of MPC stagnation are needed to turn back on complexifying
106 ComplexityFloorGenerations: 40
107
108
109
110
111
112
113 ##################/
114 # Novelty Search parameters       #
115 ##################/
116
117 # the K constant
118 NoveltySearch_K: 15
119
120 # Sparseness threshold. Add to the archive if above
121 NoveltySearch_P_min: 0.5
122
123 # Dynamic Pmin?
124 NoveltySearch_Dynamic_Pmin: true
125
126 # How many evaluations should pass without adding to the archive
127 # in order to lower Pmin
128 NoveltySearch_No_Archiving_Stagnation_Threshold: 150
129
130 # How should it be multiplied (make it less than 1.0)
131 NoveltySearch_Pmin_lowering_multiplier: 0.9
132
133 # Not lower than this value
134 NoveltySearch_Pmin_min: 0.05
135
```

```
136 # How many one-after-another additions to the archive should
137 # pass in order to raise Pmin
138 NoveltySearch_Quick_Archiving_Min_Evaluations: 8
139
140 # How should it be multiplied (make it more than 1.0)
141 NoveltySearch_Pmin_raising_multiplier: 1.1
142
143 # Per how many evaluations to recompute the sparseness of the population
144 NoveltySearch_Recompute_Sparseness_Each: 25
145
146
147
148
149 #################/
150 # Structural Mutation parameters
151 #################/
152
153 # Probability for a baby to be mutated with the Add-Neuron mutation.
154 MutateAddNeuronProb: 0.01
155
156 # Allow splitting of any recurrent links
157 SplitRecurrent: true
158
159 # Allow splitting of looped recurrent links
160 SplitLoopedRecurrent: true
161
162 # Probability for a baby to be mutated with the Add-Link mutation
163 MutateAddLinkProb: 0.15
164
165 # Probability for a new incoming link to be from the bias neuron
166 # This enforces it. A value of 0.0 doesn't mean there will not be such links
167 MutateAddLinkFromBiasProb: 0.0
168
169 # Probability for a baby to be mutated with the Remove-Link mutation
170 MutateRemLinkProb: 0.01
171
172 # Probability for a baby that a simple neuron will be replaced with a link
173 MutateRemSimpleNeuronProb: 0.0
174
175 # Maximum number of tries to find 2 neurons to add/remove a link
176 LinkTries: 128
177
178 # Probability that a link mutation will be made recurrent
179 RecurrentProb: 0.01
180
181 # Probability that a recurrent link mutation will be looped
182 RecurrentLoopProb: 0.01
183
184
185
186
187
188 #################/
189 # Parameter Mutation parameters
190 #################/
191
192 # Probability for a baby's weights to be mutated
193 MutateWeightsProb: 0.5
194
195 # Probability for a severe (shaking) weight mutation
196 MutateWeightsSevereProb: 0.5
197
198 # Probability for a particular gene's weight to be mutated. 1.0  100%
199 WeightMutationRate: 0.15
200
201 # Maximum perturbation for a weight mutation
202 WeightMutationMaxPower: 1.0
203
204 # Maximum magnitude of a replaced weight
205 WeightReplacementMaxPower: 1.0
```

```
206
207 # Maximum absolute magnitude of a weight
208 MaxWeight: 2.0
209
210 # Probability for a baby's A activation function parameters to be perturbed
211 MutateActivationAProb: 0.0
212
213 # Probability for a baby's B activation function parameters to be perturbed
214 MutateActivationBProb: 0.0
215
216 # Maximum magnitude for the A parameter perturbation
217 ActivationAMutationMaxPower: 0.0
218
219 # Maximum magnitude for the B parameter perturbation
220 ActivationBMutationMaxPower: 0.0
221
222 # Activation parameter A min/max
223 MinActivationA: 4.924273
224 MaxActivationA: 4.924273
225
226 # Activation parameter B min/max
227 MinActivationB: 0.0
228 MaxActivationB: 0.0
229
230 # Maximum magnitude for time constants perturbation
231 TimeConstantMutationMaxPower: 0.0
232
233 # Maximum magnitude for biases perturbation
234 BiasMutationMaxPower: 1.0
235
236 # Probability for a baby's neuron time constant values to be mutated
237 MutateNeuronTimeConstantsProb: 0.0
238
239 # Probability for a baby's neuron bias values to be mutated
240 MutateNeuronBiasesProb: 0.5
241
242 # Time constant range
243 MinNeuronTimeConstant: 0.0
244 MaxNeuronTimeConstant: 0.0
245
246 # Bias range
247 MinNeuronBias: -2.0
248 MaxNeuronBias: 2.0
249
250
251
252
253
254 # Probability for a baby that an activation function type will be changed for a single neuron
255 # considered a structural mutation because of the large impact on fitness
256 MutateNeuronActivationTypeProb: 0.0
257
258 # Probabilities for a particular activation function appearance
259 ActivationFunction_SignedSigmoid_Prob: 0.0
260 ActivationFunction_UnsignedSigmoid_Prob: 1.0
261 ActivationFunction_Tanh_Prob: 0.0
262 ActivationFunction_TanhCubic_Prob: 0.0
263 ActivationFunction_SignedStep_Prob: 0.0
264 ActivationFunction_UnsignedStep_Prob: 0.0
265 ActivationFunction_SignedGauss_Prob: 0.0
266 ActivationFunction_UnsignedGauss_Prob: 0.0
267 ActivationFunction_Abs_Prob: 0.0
268 ActivationFunction_SignedSine_Prob: 0.0
269 ActivationFunction_UnsignedSine_Prob: 0.0
270 ActivationFunction_SignedSquare_Prob: 0.0
271 ActivationFunction_UnsignedSquare_Prob: 0.0
272 ActivationFunction_Linear_Prob: 0.0
273
274
275
```

```
276
277 ##################/
278 # Speciation parameters
279 ##################/
280
281 # Percent of disjoint genes importance
282 DisjointCoeff: 1.0
283
284 # Percent of excess genes importance
285 ExcessCoeff: 1.0
286
287 # Average weight difference importance
288 WeightDiffCoeff: 1.5
289
290 # Node-specific activation parameter A difference importance
291 ActivationADiffCoeff: 0.0
292
293 # Node-specific activation parameter B difference importance
294 ActivationBDiffCoeff: 0.0
295
296 # Average time constant difference importance
297 TimeConstantDiffCoeff: 0.0
298
299 # Average bias difference importance
300 BiasDiffCoeff: 0.0
301
302 # Activation function type difference importance
303 ActivationFunctionDiffCoeff: 0.0
304
305 # Compatibility threshold
306 CompatThreshold: 3.0
307
308 # Minumal value of the compatibility threshold
309 MinCompatThreshold: 0.2
310
311 # Modifier per generation for keeping the species stable
312 CompatThresholdModifier: 0.2
313
314 # Per how many generations to change the threshold
315 # (used in generational mode)
316 CompatThreshChangeInterval_Generations: 1
317
318 # Per how many evaluations to change the threshold
319 # (used in steady state mode)
320 CompatThreshChangeInterval_Evaluations: 10
```

A.1.4 Locomotion Environment

Possible configuration parameters that can be defined for the locomotion environment:

Parameter	Description
`RobotScene`	The file that is loaded including the robot. The file could also include the environment, especially if the smurfs format is used.
`CreatePlane`	If no environment is loaded from file the locomotion_environment can create a simple infinite plane.
`ConveyorSpeed`	This option defines a contact speed for the plane created if the `CreatePlane` option is set. Setting the contact speed can be used to simulate a treadmill.
`PlaneCFM`	The /cfm/ parameter influences the softness of a contact with the plane (floor conditions). A ridged contact is something like 0.00000001, while a soft contact is in the area of 0.001. The softness depends also on the mass of the robot and other simulation parameters. So it might be necessary to tune the parameter for the every setup.
`FeetNodeNames`	Contains a list with the feet names of the robot. Is used to place the robot directly above the environment.
`FootPosZ`	An offset to the foot position for setting the robot into the environment. In case of sphere feet it is normally the radius.
`EvaluationCurrentFactor`	The influence of the motors current efficiency to the fitness value. **Note**: Currently, no current is created in MARS if a motor is not moving. A new motor model will be added soon.
`EvaluationTorqueFactor`	The influence of the motors torque efficiency to the fitness value.
`EvaluationNodeName`	The simulation node which defines the position of the robot.
`AbortOnContact`	Aborts the evaluation if the contact sensor returns a collision. Bad collisions can produce wrong simulation behaviors with robots jumping around. To prevent this, one should setup a contact sensor for all parts excluding the feet and set this parameter to true. It will also shorten the evaluation of undesired behaviors.
`ContactSensorID`	The sensor id in mars which provides the contact information for AbortOnContact. This sensor is a list of all bodies that shall not have contact.
`MaxTime`	Set the time in ms the robot will be evaluated if not aborted.
`StillFailureTime`	If this time in ms is reached without a change in the simulation the evaluation is aborted. This shortens the evaluation of behaviors that produces no or constant output.
`enableGUI`	Start the learning with or without GUI, without the GUI is much faster, while running with GUI allows to debug the optimization process.
`graphicsStepSkip`	The given number of simulation steps is not displayed. Makes the GUI run faster.
`DisplayTime`	With this option the simulation time can be displayed in 2D overlay.
`DisplayGrid`	Enables the visualization of a grid 4m x 4m x 2m grid.
`CreateSupportJoint`	With this option a slider joint can be created that fixes the `SupportNodeName` in the world with a slider joint with the axis in z direction. This option is mostly used if one wants to optimize behaviors of a robot or single legs on a treadmill.
`SupportNodeName`	See `CreateSupportJoint`.
`SliderMinPos`	Can be used to define a low stop of the support slider joint.
`SliderRange`	The range plus the min pos defines the high stop of the support slider joint.
`SliderD`	Defines a damping constant of the slider joint.
`SliderK`	Defines a spring constant of the slider joint. If the damping and spring constants are not set the slider if freely movable.
`VarySliderPosZ`	This options lowers the slider min position on every reset by a random value in the range of 0-5cm.
`UseSupportMotor`	This option create a support motor that holds the support slider in the initial position. The motor is activated and deactivated in an interval of one second. Whether the motor is active or not is a sensor information for the behavior. The sensor information is inserted in front of the robot sensor list. This option disables the five previous options.
`UseController`	MARS support controllers that define sensor and actuator lists. If this option is set the controller is used to define the in- and outputs of the behavior. Also the order of the in- and outputs is then defined by the order of the sensor and actuators in the controller.
`UsePWMLimit`	This option creates a second input for every motor that allows to limit the torque of the motor. It can be used for approaches where torque control is used instead of position control.
`FilterInfluence`	The filter influence value can be used to filter the behavior outputs. The value is used by: $motorValue = (lastValue * (1 - filterInfluence) + behaviorOutput * filterInfluence)$
`UseStartPosYOffset`	If set, the robot start position is varied on every reset on the y axis in a range of -0.5m to 0.5m.
`TimeOutput`	Adds a sensor value including the simulation time in front of the robot sensor list.
`SineOutput`	Adds a sensor value including a sine curve with an frequency of 1Hz.
`BagelGraphFitness`	A bagel graph can be used to define the fitness function.

A.2 Experiment Results

A.2.1 Evolving Legged Locomotion

Simulated Robot - Open Loop

Table A.3: Significance by Mann–Whitney U-test.

Config	Approach	vs	Approach	Significance
Soil	*Fourier*		*Gaussian*	$p \ll 5\%$
symmetric	*Fourier*		*Pendulum*	$p < 5\%$
	Pendulum		*Gaussian*	$p > 5\%$
Hard	*Fourier*		*Pendulum*	$p \ll 5\%$
symmetric	*Pendulum*		*Gaussian*	$p \ll 5\%$
	Fourier		*Gaussian*	$p < 5\%$
Soil	*Gaussian*		*Pendulum*	$p \ll 5\%$
full	*Fourier*		*Pendulum*	$p \ll 5\%$
	Gaussian		*Fourier*	$p > 5\%$
Hard	*Gaussian*		*Fourier*	$p \ll 5\%$
full	*Gaussian*		*Pendulum*	$p < 5\%$
	Fourier		*Pendulum*	$p < 5\%$

A.2.2 Benchmark Analysis

Environment list:

- env_0001: simple_test_problem [CEC13TestFunction: 1]
- env_0002: simple_test_problem [CEC13TestFunction: 11]

BehaviorSearch list:

- beh_0001: c0n_behavior_search [Optimizer: pso_optimizer]
- beh_0002: c0n_behavior_search [Optimizer: cmaes_optimizer]

env_0001:
beh_0001 vs. beh_0002 $p << 5\%$ $p : 2.98769040667e - 11$

env_0002:
beh_0001 vs. beh_0002 $p << 5\%$ $p : 1.44186279911e - 09$

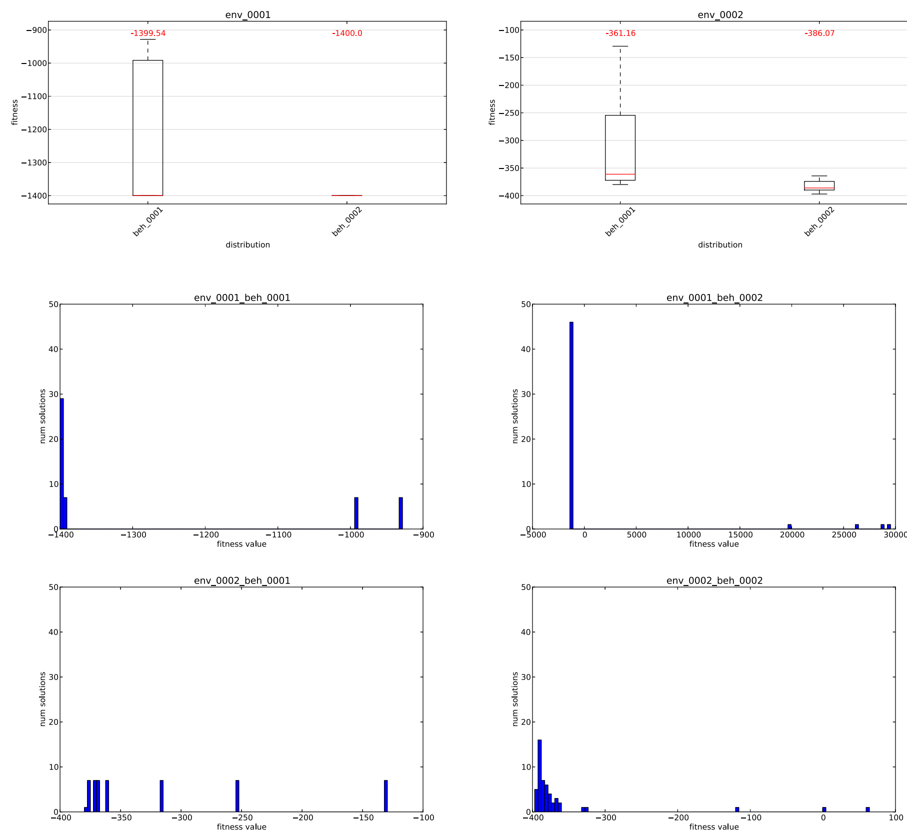


Figure A.1: Example output of automatically generated benchmark analysis.

Terms and Definitions

bias Some fixed value given into a control module.

contralateral Legs on the other side of the corpus.

crossover Combining the *DNA* of two parents into a new *DNA*.

fitness Fitness means the fitness of an individual of an evolutionary process. The selection of the parents for the next generation is done based on that fitness. Thus, the fitness correlates to a reward in the RL domain or a general evaluation.

fitness value The fitness value describes the fitness of an individual by a number. Due to often used fitness calculations, like the *RMSE*, a low fitness value represent a good individual fitness.

generation A generation is a collection of all individuals of one species with the same amount of ancestors.

genotype The genetic representation (*DNA*) of an individual.

individual An individual is a single entity underlying the process of fitness evaluation.

ipsilateral Legs on the same side of the corpus.

mutation The non-perfect process of copying a *DNA*. Due to errors in the process the copied *DNA* differs from the prototype.

phase A value between 0 and 1 defining the current position in a rhythmic pattern.

phenotype The final representation of an individual used for the fitness evaluation.

population A population is a collection of all simultaneously living individuals of one species. In an abstract model population and generation are often the same.

pulse A peek pattern being shortly 1 every x seconds and 0 otherwise.

Acronyms

ANN Artificial Neural Network

API Application Programming Interface

BAGEL Biologically Graph-Based Language

BEAGLE C++ Evolutionary Computation (EC) framework

BOLERO Behavior Optimization and Learning for Robots

CMA-ES Covariance Matrix Adaptation Evolutionary Strategy

CEC Congress on Evolutionary Computation

cfm Constraint Force Mixing

CPG Central Pattern Generator

DNA Deoxyribonucleic acid is the molecule that encodes the development of a life form

EC Evolutionary Computation

erp Error Reducing Parameter

ES Evolutionary Strategies

FPGA Field Programmable Gate Array

FTP File Transfer Protocol

GA Genetic Algorithms

GP Genetic Programming

GUI Graphical User Interface

IMU Inertial Measurement Unit

JSON Javascript Object Notation

MARS Simulation Machina Arte Robotum Simulans

ML Machine Learning

M.O.N.S.T.E.R. **M**icr**o**ker**n**el for **s**cabrous **t**errain **e**xploring **r**obots

NEAT Neuroevolution of Augmenting Topologies

ngc Number of ground contacts of robot's parts other than the feet

ODE Open Dynamics Engine

OSG OpenSceneGraph

PCR **P**osture, **C**entral pattern generator, and **R**eflex controller

PID Proportional–Integral–Derivative

PSO Particle Swarm Optimization

PYTHON Script-based programming language

QT An open source development framework for graphical user interfaces

REST Representational State Transfer

RL Reinforcement Learning

RMSE Root mean square error

ROCK The Robot Construction Kit

SABRE Swarm-Assisted Evolutionary Algorithm

TORCS The Open Racing Car Simulator

YAML YAML ain't markup language

Math Notations

General notations:

$\boldsymbol{x}$ Bold letters in equations are used to denote vectors.

σ_x A sigma followed by a subscript letter denotes the standard deviation of the corresponding evaluation criterion.

% The modulo as used in programming languages: $a\%b = \begin{cases} a - b\lfloor \frac{a}{b} \rfloor & \text{if } a > 0 \\ a - b\lceil \frac{a}{b} \rceil & \text{else} \end{cases}$.

$\mathrm{N}(\mu, \sigma^2)$ Normal distribution with μ as mean and σ^2 as variance.

$\mathrm{U}(a, b)$ Uniform distribution with $a \leq \mathrm{U}(a, b) \leq b$.

$argmin_i f(X[i])$ i'th element from X with minimum function value $f(x)$.

$j(\phi)$ Function defining the joint angle (θ) over the phase (ϕ).

$\oslash$ Symbol used to denote the average.

Symbols used:

ϕ	Phase of a pattern
ϕ_G	Global phase of the CPG
φ	Phase shift
ρ	Short trigger signal
θ	Joint angle
f	CPG frequency
t	Time
F_p	Force generated by the pendulum controller
F_e	Force added to the pendulum controller through input
ω_p	Virtual velocity of the pendulum controller
d_p	Damping of the virtual velocity of the pendulum controller
f_p	Frequency parameter of the pendulum controller
$\mathbb{R}^+$	Definition: $\mathbb{R}^+ := x \in \mathbb{R} \mid x > 0$
s_p	Percentage of the swing part within a step phase
μ	Friction coefficient
ϑ	Average motor torque used for evaluation
ϱ	Average joint load used for evaluation
p	Robot position
α	Robot roll angle, normally measured at the corpus
β	Robot pitch angle, normally measured at the corpus
γ	Robot yaw angle, normally measured at the corpus

v	Robot velocity
σ	Mutation step size
σ_{min}	Minimum mutation step size
σ_{max}	Maximum mutation step size
β	Step size adaption
$\dot{x}$	Derivative of x
T_{Stance}	Stance time of locomotion pattern
T_{Lift}	Lift time of locomotion pattern
T_{Shift}	Shift time of locomotion pattern
T_{Down}	Down time of locomotion pattern
T_{Swing}	Swing time of locomotion pattern
T_{Step}	Step time of locomotion pattern
R_x	Step length in x for one step cycle
R_y	Step length in y for one step cycle
R_{yaw}	Orientation change within one step cycle
R_{height}	Lift height of the leg in the swing phase
P_{th}	Threshold value
k_p	Spring constant
k_d	Spring damping factor
h	Simulation step size
d	Distance
d_m	Damage
P	Population
P_i	I'th sub-population
λ	Number of individuals in one population (in case of SABRE sub-population)
κ	Number of sub-population of SABRE
Λ	Number of individuals in one population of SABRE
N_{max}	Maximum number of nodes in a BAGEL graph generated by SABRE
E_{max}	Maximum number of edges in a BAGEL graph generated by SABRE
o	Offset
a	Amplitude
c_n	Constant negative reward used for Boolean evaluation criteria
f_s	Feet slippage percentage (slip velocity over robot velocity)
m_c	Robot's center of mass
τ	Boolean value that is true if $ngc > 0$ and false otherwise
τ'	Boolean value that is true if a joint load is above a given threshold
p_1	Start position of the swing phase in radian for the Sine-Based CPG
p_2	Start position of the stance phase in radian for the Sine-Based CPG

List of Tables

2.1 The maximal range of motion of CHARLIE's spine. The values on the single *dof* are further limited by the current pose on the other axes. 29

3.1 Covered distances within the three experiments. 37
3.2 Results of passive soil experiments. 43
3.3 Results of passive soil validation. 44

4.1 Default parameters of the NEAT integration into BOLERO. 61

6.1 Default parameters of SABRE. 81
6.2 Parameters of SABRE for black box function approximation. 84
6.3 Results of black box function approximation with no significant difference. 85
6.4 Comparison of new default SABRE configuration with best results from the benchmark. 91
6.5 Parameter optimization results. 92
6.6 TORCS benchmark results. 93
6.7 HyperRobot benchmark results. 94

8.1 Results of the open-loop experiment for EASY4 sorted by the configurations.122
8.2 Number of results per approach sorted by number of ground contacts in the control cycle time for the experiment with the soil model and symmetric left and right leg patterns. 125
8.3 Statistical results of the comparison of EASY4's navigation and stabilize controller. 132
8.4 Statistics of various important variables over 50 independent evolution processes of both experiments with SPOT. 138
8.5 SPACECLIMBER control parameters used for the model-based control to create comparable data. 139
8.6 Parameter ranges defined for the Sine-Based approach to evolve a SPACECLIMBER open-loop walking controller. 141
8.7 Characteristics of one result for the Fitness_2 configuration tested on the real SPACECLIMBER system and in simulation with different activity. . . . 146
8.8 The force profile used to compare the evolved stabilize controller of CHARLIE with the initial open-loop controller. 160
8.9 Statistics of the comparison between the initial open-loop controller and the resulting reactive controller of CHARLIE. 160

A.1 Benchmark functions used in the parameter optimization competition at the CEC13 conference. 172
A.3 Significance by Mann–Whitney U-test. 179

List of Figures

1.1 Examples of natural evolution. 3
1.2 Graphical overview showing the structure of the thesis. 5

2.1 Illustration of WALKNET rules. 11
2.2 Structure of a multipolar neuron. 12
2.3 Examples of neural membrane properties. 12
2.4 Architecture of an artificial neuron. 13
2.5 Schematic examples of neural pattern generators. 14
2.6 The timeline of evolution of life. 15
2.7 Overview of research fields of Evolutionary Computation. 17
2.8 Influence of covariance matrix on CMA-ES mutation. 18
2.9 Cumulative step width calculation of CMA-ES. 18
2.10 Examples of crossover operation. 19
2.11 An example of a possible GP-tree representing the function $f(x) = x^2 + 0.5y$. 19
2.12 The SPOT robot. 23
2.13 The SPACECLIMBER system in an environment simulation the Mars surface. 24
2.14 SPACECLIMBER morphology parameters for structure optimization. 24
2.15 SPACECLIMBER structure optimization setup. 25
2.16 SPACECLIMBER parameters for structure optimization. 25
2.17 Illustration of stability margin of SPACECLIMBER 25
2.18 Posture parameters of the SPACECLIMBER controller. 26
2.19 The CHARLIE robot climbing on the crater of the space exploration hall at the RIC/DFKI. 28
2.20 Illustration of the CHARLIE's degrees of freedom including the spine frame and the force torque sensor frames. 28

3.1 Main architecture overview of the MARS framework. 32
3.2 Graphical user interface of MARS 34
3.3 3D visualization example of MARS 34
3.4 Setup for comparison of simulated and real behavior of SPACECLIMBER. . 37
3.5 Position and orientation of the real and simulated robot over the experiment period. 38
3.6 Comparison of the reference joint angle and the measured angle of the real actuator and the simulated one. 38
3.7 Pitch and roll orientation of the real and simulated robot over 20 s. . . . 39
3.8 Example feet for foot-soil interaction. 40
3.9 Illustration of the neural network architecture used to model the cylinder behavior of the two passive experiment setups. 40
3.10 Soil drop experiment setup. 41
3.11 Soil pull experiment setup. 42
3.12 A possible indirect decoding from a genotype into a phenotype. 44

3.13 Setup of active soil experiments. 45
3.14 The resulting measured forces of the performed fifteen experiment repetitions. 46
3.15 Comparison between active soil experiment data and output of one generated neural network. 47
3.16 Generalization of active soil experiment result over soil variance parameter. 47
3.17 Measured data of five performed horizontal foot-soil experiments. 48
3.18 Result of active horizontal foot-soil experiments. 48
3.19 Validation of active horizontal foot-soil experiments. 49
3.20 The SPACECLIMBER robot in the simulation with the new developed foot-soil interaction. 49

4.1 Architecture overview of BOLERO. 52
4.2 Architecture of benchmark distribution. 55
4.3 Result folder structure of BOLERO benchmarks. 56
4.4 TORCS car racing simulation. 58
4.5 Torcs car racing setup for SABRE benchmark. 58
4.6 A comparatively simple robot with four legs and three *dof* per leg. 59
4.7 GP tree handling two output dimensions. 61

5.1 Different representations of nodes used in a GP tree), ANNs, and BAGEL . 63
5.2 Interface illustration of BAGEL. 64
5.3 Example of BAGEL graph implementing the function: $f(In1, In2) := 0.5(In1 \cdot In2)^{In2}$. 67
5.4 Example for execution order of BAGEL nodes. 68
5.5 Main architecture overview of GRAPHGUI. 72
5.6 GRAPHGUI developed to design BAGEL graphs. 73

6.1 SABRE added to the overview of evolutionary computation. 75
6.2 The function $f(x, y) = x^2 + 0.5y$ represented in a BAGEL graph and as gene string (underneath the graph). 76
6.3 The two nested loops of SABRE. 77
6.4 Example of structure development in SABRE. 79
6.5 Mapping of a PSO particle to a gene string of SABRE. 80
6.6 Software overview of SABRE. 81
6.7 Integration of SABRE with BAGEL into BOLERO. 82
6.8 Average fitness over run-time on black box function approximation. . . . 86
6.9 *RMSE* distribution of transfer function analysis. 87
6.10 Fitted rational functions over median of minimum *RMSE* distributions of the 100 runs for all configurations and across all settings. 88
6.11 Max, mean, and min fitness development of transfer function analysis. . . 89
6.12 Analysis on fitness improvements of transfer function analysis. 89
6.13 Distribution of the changes of the *RMSE* of transfer function analysis. . . 90
6.14 The histogram shows the number of improvements over the reached *RMSE*. 91

7.1 Basic control concept of the CPG based approach. 95
7.2 Basic concept for the CPG-based joint trajectory generation. 96
7.3 The global phase module modeled in GRAPHGUI. 96
7.4 Example transformation of a sine curve into the joint trajectory generated by the Sine-Based pattern approach. 98

7.5 Non-linear influence of the incline parameter d to the slope and phase shift of the Gaussian bell. 99
7.6 The graph depicts the linear influence of the parameter o_d that is transformed to parameter d. 99
7.7 The resulting influence of o_d to the shape of the curve. 100
7.8 Similar to parameter o_d the width of the Gaussian bell is influenced linearly by o_b in a range of $[0, 1]$. 100
7.9 Example of a joint pattern created by the pendulum controller giving just an impulse based rhythmic force input. 101
7.10 Example of the fading behavior of the pendulum controller. 102
7.11 Two example patterns that can be generated by the Neural Oscillator approach. 103
7.12 Different sine test functions for the pattern generators. 105
7.13 Fitness distribution of 100 performed repetitions for all approaches on the single sine curve. 105
7.14 Median result of the SIGMOID$_2$ approach on the single sine curve. 106
7.15 Example modulation of the SIGMOID$_2$ approach. 106
7.16 Example modulation of the PENDULUM$_{fire}$ approach. 107
7.17 Fitness distribution of all tested approaches for the two overlapping sine waves. 108
7.18 Median result of SINEPG$_1$, NEURALOS$_1$, and SIGMOID$_2$ configuration for the two overlapping sine waves. 108
7.19 Fitness distributions of the approaches for the multiple overlapping sine waves. 109
7.20 Median result of SIGMOID$_2$ and PENDULUM$_{Gauss}$ approaches for the multiple overlapping sine waves. 109
7.21 Example median results of the optimization of the single Gaussian test function. 110
7.22 Fitness distribution of all approaches on the single Gaussian test function. 111
7.23 Example median results of the optimization of four Gaussian test functions optimized simultaneously. 111
7.24 Fitness distribution of all approaches on the four Gaussian test function optimized simultaneously. 112
7.25 Median result of the PENDULUM$_{Fire}$ approach on the twenty Gaussian test functions optimized simultaneously. 112
7.26 Median result of the GAUSSapproach on the twenty Gaussian test functions optimized simultaneously. 112
7.27 Fitness distribution of all approaches on twenty Gaussian test functions optimized simultaneously. 113
7.28 Median result of the PENDULUM$_{Fire}$ approach on the sixteen CHARLIE joint patterns. 113
7.29 Fitness distribution of all approaches on the sixteen CHARLIE joint patterns.114
7.30 Medium results of PENDULUM$_{Fire}$ and FOURIERon a set of joint patterns measured from human locomotion. 114
7.31 Fitness distribution of all approaches on the human locomotion pattern. . 115
7.32 Single experiment to compare the Fourier approach with the Gaussian one on more artificial joint patterns. 116

8.1 Simulation robot EASY4 used to evaluate different joint pattern representations and a reactive control concept. 118

8.2 Evaluation function for the locomotion benchmark setup defined by a BAGEL graph processing simulation data provided by `data_broker`. . . . 119
8.3 EASY4 main control architecture for open-loop locomotion. 120
8.4 Predefined leg control structure for the Fourier series approach. 120
8.5 Predefined leg control structure for the Pendulum approach. 121
8.6 Best EASY4 behavior generated with the Fourier series approach for the first configuration $config_{soil/sym}$ 123
8.7 Best EASY4 behavior generated with the Gaussian pattern approach for $config_{soil/sym}$ 124
8.8 Best EASY4 behavior generated with the Pendulum pattern approach for $config_{soil/sym}$ 124
8.9 Simulation setup of the navigation experiment for EASY4. 126
8.10 Controller extension for the navigation experiment for EASY4. 127
8.11 The resulting path taken by the evolved navigation controller of EASY4. . 128
8.12 The plots depict how the joint patterns are modified by the navigation modules of EASY4. 129
8.13 A test setup to evaluate the evolved navigation controller of EASY4. 130
8.14 The result of the evolved navigation controller of EASY4 used to follow a defined path. 130
8.15 Simulation setup to evolve a stabilizing controller for EASY4. 131
8.16 The control architecture of the stabilize controller of EASY4 133
8.17 Comparison of the navigation controller and the evolved stabilize controller of EASY4 in the rough environment. 134
8.18 A time-progression of snapshots showing an evolved walking behavior that is executed on both the simulated and real SPOT robot. 137
8.19 The real SPACECLIMBER system on the locomotion test-bed. 138
8.20 SPACECLIMBER kinematic leg model and joints used for the Sine-Based pattern optimization. 140
8.21 Joint patterns created by the Sine-Based approach for SPACECLIMBER with the $Fitness_1$ configuration. 142
8.22 Comparison of the feet z position between the left legs of SPACECLIMBER for the result of the $Fitness_1$ configuration. 143
8.23 Comparison of the linear velocity on the x and y axis of the robot main body for the result of the $Fitness_1$ configuration and the IK_2 reference behavior. 144
8.24 Comparison of the robot height. 144
8.25 Feet positions of the robot fixated in the simulation (forward kinematics) with the result of the $Fitness_1$ configuration compared to the IK_2 reference behavior. 144
8.26 Joint patterns created by the Sine-Based approach for SPACECLIMBER with the $Fitness_2$ configuration. 145
8.27 Feet positions of the robot fixated in the simulation (forward kinematics) and comparison of feet z movement while walking on the planar surface with the result of the $Fitness_2$ configuration. 146
8.28 The single leg simulation setup for SPACECLIMBER. 147
8.29 The fitness graph used for the evolution of the SPACECLIMBER's leg control. 148
8.30 The control graph for a single SPACECLIMBER leg. 148
8.31 The control graph for the whole SPACECLIMBER system. 149

8.32 Comparison of the leg patterns generated by the evolution of the single leg control and the adaption of the patterns to the whole system. 150
8.33 SPACECLIMBER foot positions of the bottom-up result generate by the forward kinematic model compared to the foot height measured while walking on the ground. 151
8.34 The simulation setup to generate a walking behavior for the front leg pair of CHARLIE. 152
8.35 The top-level BAGEL graph defined for evolving walking patterns of the front leg pair of CHARLIE. 152
8.36 The evaluation graph used to generate a locomotion pattern for the front leg pair of CHARLIE. 153
8.37 The evolved patterns of the front legs selected as start point for the whole CHARLIE system. 153
8.38 The simulation setup to generate a walking behavior for the rear leg pair of CHARLIE. 154
8.39 The evaluation graph used to generate a locomotion pattern for the rear leg pair of CHARLIE. 154
8.40 The evolved patterns of the rear legs selected as start point for the whole CHARLIE system. 155
8.41 The evaluation graph used to generate the controller of the whole CHARLIE system. 155
8.42 Top-level open-loop control graph of CHARLIE. 156
8.43 Comparison of the start patterns with the resulting adapted patterns of the whole CHARLIE system. 157
8.44 The resulting patterns generated for CHARLIE's spine *dof*. 158
8.45 The image sequence depicts the resulting walking pattern of CHARLIE transferred to the real system on the locomotion test-bed. 158
8.46 Top-level control graph for the stabilize controller of CHARLIE. 159
8.47 Comparison of the modulation signal for the left shoulder generated with and without a disturbing force applied to CHARLIE's front body. 161

A.1 Example output of automatically generated benchmark analysis. 180

Bibliography

Ahmed, M., Quack, L., Römmermann, M., and Yoo, Y.-H. (2011). Development of a real and simulation testbed for legged robot soil interaction. In *Proceedings of the 17th International Conference of the International Society for Terrain-Vehicle Systems*, ISTVS.

Albiez, J. C. (2007). *Verhaltensnetzwerke zur adaptiven Steuerung biologisch motivierter Laufmaschinen*. Doctoral dissertation, Karlsruher Institut für Technologie, Germany.

Angeline, P. J., Saunders, G. M., and Pollack, J. B. (1994). An evolutionary algorithm that constructs recurrent neural networks. *IEEE Transactions on Neural Networks*, 5(1):54–65.

Auerbach, J. E. and Bongard, J. C. (2011). Evolving complete robots with CPPN-NEAT: The utility of recurrent connections. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 1475–1482, New York, NY, USA. ACM.

Bartsch, S. (2014). *Development, Control, and Empirical Evaluation of the Six-Legged Robot SpaceClimber Designed for Extraterrestrial Crater Exploration*. Doctoral dissertation, University of Bremen, Germany.

Bartsch, S., Birnschein, T., Cordes, F., Kühn, D., Kampmann, P., Hilljegerdes, J., Planthaber, S., Römmermann, M., and Kirchner, F. (2010). SpaceClimber: Development of a six-legged climbing robot for space exploration. In *ISR 2010 (41st International Symposium on Robotics) and ROBOTIK 2010 (6th German Conference on Robotics)*, pages 1–8.

Bartsch, S., Birnschein, T., Römmermann, M., Hilljegerdes, J., Kühn, D., and Kirchner, F. (2012). Development of the six-legged walking and climbing robot SpaceClimber. *Journal of Field Robotics*, 29(3):506–532.

Bartsch, S., Manz, M., Kampmann, P., Dettmann, A., Hanff, H., Langosz, M., von Szadkowski, K., Hilljegerdes, J., Simnofske, M., Kloss, P., Meder, M., and Kirchner, F. (2016). Development and control of the multi-legged robot MANTIS. In *ISR 2016: 47st International Symposium on Robotics.*, pages 379–386, Berlin, Offenbach. VDE VERLAG GmbH.

Beer, R. D. and Gallagher, J. C. (1992). Evolving dynamical neural networks for adaptive behavior. *Adaptive Behavior*, 1(1):91–122.

Bentley, P. and Kumar, S. (1999). Three ways to grow designs: A comparison of embryogenies for an evolutionary design problem. In *Proceedings of the 1st Annual Conference*

on Genetic and Evolutionary Computation - Volume 1, GECCO'99, pages 35–43, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Boeing, A., Hanham, S., and Braunl, T. (2004). Evolving autonomous biped control from simulation to reality. In *Proceedings of the Second International Conference on Autonomous Robots and Agents*, pages 440–445, Palmerston North, New Zealand.

Bogue, R. (2012). Robots in the laboratory: a review of applications. *Industrial Robot: An International Journal*, 39(2):113–119.

Bombled, Q. (2011). *Modeling and Control of Six-Legged Robots Application to AMRU5*. Doctoral dissertation, University of Mons, Belgium.

Bongard, J., Zykov, V., and Lipson, H. (2006). Resilient machines through continuous self-modeling. *Science*, 314(5802):1118–1121.

Bongard, J. C. and Pfeifer, R. (2001). Repeated structure and dissociation of genotypic and phenotypic complexity in artificial ontogeny. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, GECCO'01, pages 829–836, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Bongard, J. C. and Pfeifer, R. (2002). A method for isolating morphological effects on evolved behaviour. In *Proceedings of the Seventh International Conference on Simulation of Adaptive Behavior on From Animals to Animats*, ICSAB, pages 305–311, Cambridge, MA, USA. MIT Press.

Bretl, T., Rock, S., and Latombe, J. . (2003). Motion planning for a three-limbed climbing robot in vertical natural terrain. In *2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422)*, volume 3, pages 2946–2953.

Cardamone, L., Loiacono, D., and Lanzi, P. L. (2009). Evolving competitive car controllers for racing games with neuroevolution. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 1179–1186, New York, NY, USA. ACM.

Clune, J., Beckmann, B. E., Ofria, C., and Pennock, R. T. (2009). Evolving coordinated quadruped gaits with the HyperNEAT generative encoding. In *2009 IEEE Congress on Evolutionary Computation*, pages 2764–2771.

Cruse, H., Kindermann, T., Schumm, M., Dean, J., and Schmitz, J. (1998). Walknet - a biologically inspired network to control six-legged walking. *Neural Networks*, 11(7):1435–1447.

Custódio, F. L., Barbosa, H. J. C., and Dardenne, L. E. (2014). A multiple minima genetic algorithm for protein structure prediction. *Applied Soft Computing*, 15:88–99.

D'Ambrosio, D. B. and Stanley, K. O. (2007). A novel generative encoding for exploiting neural network sensor and output geometry. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 974–981, New York, NY, USA. ACM.

Dellaert, F. and Beer, R. D. (1996). A developmental model for the evolution of complete autonomous agents. In *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, pages 393–401. MIT Press.

Dettmann, A., Langosz, M., von Szadkowski, K. A., and Bartsch, S. (2014). Towards lifelong learning of optimal control for kinematically complex robots. In *Workshop on Modelling, Estimation, Perception and Control of All Terrain Mobile Robots . IEEE International Conference on Robotics and Automation (ICRA-2014)*. IEEE.

Dettmann, A., Römmermann, M., and Cordes, F. (2011). Evolutionary development of an optimized manipulator arm morphology for manipulation and rover locomotion. In *2011 IEEE International Conference on Robotics and Biomimetics*, pages 2567–2573.

Dominici, N., Ivanenko, Y. P., Cappellini, G., d'Avella, A., Mondì, V., Cicchese, M., Fabiano, A., Silei, T., Di Paolo, A., Giannini, C., Poppele, R. E., and Lacquaniti, F. (2011). Locomotor primitives in newborn babies and their development. *Science*, 334(6058):997–999.

Endo, K., Maeno, T., and Kitano, H. (2003). Co-evolution of morphology and walking pattern of biped humanoid robot using evolutionary computation:designing the real robot. In *2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422)*, volume 1, pages 1362–1367.

Fabisch, A. and Langosz, M. (2018). BOLeRo, https://github.com/rock-learning/bolero.

Floreano, D., Dürr, P., and Mattiussi, C. (2008). Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62.

Gagné, C. and Parizeau, M. (2006). Genericity in evolutionary computation software tools: Principles and case study. *International Journal on Artificial Intelligence Tools*, 15(2):173–194.

Garcia, E., Galvez, J. A., and Gonzalez de Santos, P. (2003). On finding the relevant dynamics for model-based controlling walking robots. *Journal of Intelligent and Robotic Systems*, 37(4):375–398.

Gay, S., Santos-Victor, J., and Ijspeert, A. (2013). Learning robot gait stability using neural networks as sensory feedback function for central pattern generators. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 194–201.

Gomez, F. and Miikkulainen, R. (1997). Incremental evolution of complex general behavior. *Adaptive Behavior*, 5(3-4):317–342.

Gomez, F., Schmidhuber, J., and Miikkulainen, R. (2008). Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, 9:937–965.

Gomi, T. and Ide, K. (1998). Evolution of gaits of a legged robot. In *1998 IEEE International Conference on Fuzzy Systems Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98CH36228)*, volume 1, pages 159–164.

Gruau, F. (1994). *Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm*. PhD thesis, Ecole Normale Superieure de Lyon, Laboratoire de l'Informatique du Parallelisme, France.

Gruau, F. and Quatramaran, K. (1996). Cellular encoding for interactive evolutionary robotics. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands.

Guertin, P. a. (2009). The mammalian central pattern generator for locomotion. *Brain research reviews*, 62(1):45–56.

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

Heidrich-Meisner, V. and Igel, C. (2009). Neuroevolution strategies for episodic reinforcement learning. *Journal of Algorithms*, 64(4):152–168.

Herzog, A., Rotella, N., Mason, S., Grimminger, F., Schaal, S., and Righetti, L. (2016). Momentum control with hierarchical inverse dynamics on a torque-controlled humanoid. *Autonomous Robots*, 40(3):473–491.

Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA.

Hornby, G. S., Fujita, M., Takamura, S., Yamamoto, T., and Hanagata, O. (1999). Autonomous evolution of gaits with the sony quadruped robot. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2*, GECCO'99, pages 1297–1304, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Hvilshøj, M., Bøgh, S., Nielsen, O. S., and Madsen, O. (2012). Autonomous industrial mobile manipulation (AIMM): past, present and future. *Industrial Robot: An International Journal*, 39(2):120–135.

Igel, C. (2003). Neuroevolution for reinforcement learning using evolution strategies. In *The 2003 Congress on Evolutionary Computation*, volume 4 of *CEC '03*, pages 2588–2595.

Ijspeert, A. J. (2008). Central pattern generators for locomotion control in animals and robots: A review. *Neural Networks*, 21(4):642–653. Robotics and Neuroscience.

Iocchi, L., Holz, D., del Solar, J. R., Sugiura, K., and van der Zant, T. (2015). RoboCup@Home: Analysis and results of evolving competitions for domestic and service robots. *Artificial Intelligence*, 229:258–281.

Ito, K. and Matsuno, F. (2002). A study of reinforcement learning for the robot with many degrees of freedom - acquisition of locomotion patterns for multi-legged robot. In *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*, volume 4, pages 3392–3397.

Izquierdo, E. J. and Lockery, S. R. (2010). Evolution and analysis of minimal neural circuits for klinotaxis in caenorhabditis elegans. *Journal of Neuroscience*, 30(39):12908–12917.

Jakobi, N. (1995). Harnessing morphogenesis. In *International Conference on Information Processing in Cells and Tissues*, pages 29–41.

Jakobi, N., Husbands, P., and Harvey, I. (1995). Noise and the reality gap: The use of simulation in evolutionary robotics. In Morán, F., Moreno, A., Merelo, J. J., and Chacón, P., editors, *Advances in Artificial Life*, pages 704–720, Berlin, Heidelberg. Springer.

Jones, S. (2000). *Almost Like a Whale: The Origin of Species Updated*. Black Swan.

Kassahun, Y., de Gea, J., Römmermann, M., and Kirchner, F. (2009). On applying neuroevolutionary methods to complex robotic tasks. In *IEEE IROS Workshops on Exploring new horizons in Evolutionary Design of robots*, pages 26–30.

Kassahun, Y., Edgington, M., De Gea, J., and Kirchner, F. (2007a). Exploiting sensorimotor coordination for learning to recognize objects. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, IJCAI'07, pages 883–888, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Kassahun, Y., Edgington, M., Metzen, J. H., Sommer, G., and Kirchner, F. (2007b). A common genetic encoding for both direct and indirect encodings of networks. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 1029–1036, New York, NY, USA. ACM.

Kennedy, J. and Eberhart, R. (1995). Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948.

Kirchner, F. (1998). Q-learning of complex behaviours on a six-legged walking machine. *Robotics and Autonomous Systems*, 25(3):253–262. Autonomous Mobile Robots.

Kirchner, F. (1999). *Hierarchical Q-Learning in Complex Robot Control Problems*. Doctoral dissertation, Department. of Computer Science, University Bonn, Germany.

Kitano, H. (1990). Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4:461–476.

Klaassen, B., Linnemann, R., Spenneberg, D., and Kirchner, F. (2002). Biomimetic walking robot SCORPION: Control and modeling. *Robotics and Autonomous Systems*, 41(2):69–76. Ninth International Symposium on Intelligent Robotic Systems.

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Koutník, J., Cuccu, G., Schmidhuber, J., and Gomez, F. (2013). Evolving large-scale neural networks for vision-based reinforcement learning. In *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, pages 1061–1068, New York, NY, USA. ACM.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

Koza, J. R. (2010). Human-competitive results produced by genetic programming. *Genetic Programming and Evolvable Machines*, 11(3):251–284.

Kühn, D. (2016). *Design and development of a hominid robot with local control in its adaptable feet to enhance locomotion capabilities*. Doctoral dissertation, University of Bremen, Germany.

Kühn, D., Römmermann, M., Sauthoff, N., Grimminger, F., and Kirchner, F. (2009a). Concept evaluation of a new biologically inspired robot "LittleApe". In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 589–594.

Kühn, D., Sauthoff, N., Grimminger, F., Römmermann, M., and Kirchner, F. (2009b). Towards a biologically inspired ape-like robot. In *Mobile Robotics*, pages 165–172.

Langosz, M. (2018). MARS, https://github.com/rock-simulation/mars.

Langosz, M., Quack, L., Dettmann, A., Bartsch, S., and Kirchner, F. (2013). A behavior-based library for locomotion control of kinematically complex robots. In *Nature-Inspired Mobile Robotics*, pages 495–502.

Langosz, M., von Szadkowski, K. A., and Kirchner, F. (2014). Introducing particle swarm optimization into a genetic algorithm to evolve robot controllers. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO Comp '14, pages 9–10, New York, NY, USA. ACM.

Lee, H., Shen, Y., Yu, C.-H., Singh, G., and Ng, A. Y. (2006). Quadruped robot obstacle negotiation via reinforcement learning. In *Proceedings 2006 IEEE International Conference on Robotics and Automation*, ICRA 2006, pages 3003–3010.

Lee, S.-H., Sifakis, E., and Terzopoulos, D. (2009). Comprehensive biomechanical modeling and simulation of the upper body. *ACM Transactions on Graphics*, 28(4):99:1–99:17.

Lewinger, W. A., Branicky, M. S., and Quinn, R. D. (2006). Insect-inspired, actively compliant hexapod capable of object manipulation. In Tokhi, M. O., Virk, G. S., and Hossain, M. A., editors, *Climbing and Walking Robots*, pages 65–72, Berlin, Heidelberg. Springer.

Lewis, M. A., Fagg, A. H., and Solidum, A. (1992). Genetic programming approach to the construction of a neural network for control of a walking robot. In *Proceedings 1992 IEEE International Conference on Robotics and Automation*, volume 3, pages 2618–2623.

Liang, J. J., Qu, B. Y., Suganthan, P. N., and Hernández-Díaz, A. G. (2013). Problem definitions and evaluation criteria for the cec 2013 special session on real-parameter optimization. Technical Report 201212, Zhengzhou University and Nanyang Technological University, Zhengzhou, China and Singapore.

Luque-Baena, R., Urda, D., Claros, M. G., Franco, L., and Jerez, J. (2014). Robust gene signatures from microarray data using genetic algorithms enriched with biological pathway keywords. *Journal of Biomedical Informatics*, 49:32–44.

Manoonpong, P., Pasemann, F., and Wörgötter, F. (2008). Sensor-driven neural control for omnidirectional locomotion and versatile reactive behaviors of walking machines. *Robotics and Autonomous Systems*, 56(3):265–288.

Marder, E. and Bucher, D. (2001). Central pattern generators and the control of rhythmic movements. *Current Biology*, 11(23):R986–R996.

Mattiussi, C. (2005). *Evolutionary synthesis of analog networks*. Doctoral dissertation, Lausanne, Switzerland.

Mazzapioda, M., Cangelosi, A., and Nolfi, S. (2009). Evolving morphology and control: A distributed approach. In *2009 IEEE Congress on Evolutionary Computation*, pages 2217–2224.

Minassian, K., Hofstoetter, U. S., Dzeladini, F., Guertin, P. A., and Ijspeert, A. (2017). The human central pattern generator for locomotion: Does it exist and contribute to walking? *The Neuroscientist*, 23(6):649–663. PMID: 28351197.

Moriarty, D. E. and Miikkulainen, R. (1996). Efficient reinforcement learning through symbiotic evolution. *Machine Learning*, 22(1):11–32.

Nolfi, S. and Parisi, D. (1991). Growing neural networks. Technical Report PCIA-91-15, Institute of Psychology, Rome.

Osfield, R. (2018). OpenSceneGraph, http://www.openscenegraph.org.

Pandy, M. G. and Andriacchi, T. P. (2010). Muscle and joint function in human locomotion. *Annual Review of Biomedical Engineering*, 12(1):401–433. PMID: 20617942.

Parsons, R., Forrest, S., and Burks, C. (1993). Genetic algorithms for DNA sequence assembly. In *Proceedings of the first International Conference on Intelligent Systems for Molecular Biology*, pages 310–318.

Pasemann, F., Hild, M., and Zahedi, K. (2003). SO(2)-networks as neural oscillators. In Mira, J. and Álvarez, J. R., editors, *Computational Methods in Neural Modeling*, IWANN 2003, pages 144–151, Berlin, Heidelberg. Springer.

Paskarbeit, J., Schmitz, J., Schilling, M., and Schneider, A. (2010). Layout and construction of a hexapod robot with increased mobility. In *2010 3rd IEEE RAS EMBS International Conference on Biomedical Robotics and Biomechatronics*, pages 621–625.

Peng, X. B., Berseth, G., Yin, K., and Van De Panne, M. (2017). DeepLoco: Dynamic locomotion skills using hierarchical deep reinforcement learning. *ACM Transactions on Graphics*, 36(4):41:1–41:13.

Qt Group (2018). Qt, https://www.qt.io.

Raibert, M., Blankespoor, K., Nelson, G., and Playter, R. (2008). BigDog, the rough-terrain quadruped robot. *IFAC Proceedings Volumes*, 41(2):10822–10825. 17th IFAC World Congress.

Rechenberg, I. (1964). *Kybernetische Lösungsansteuerung einer experimentellen Forschungsaufgabe*. Hermann-Föttinger-Institut für Strömungstechnik der Technische Universität Berlin, Germany.

Rechenberg, I. (1973). *Evolutionsstrategie; Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog, Stuttgart, Germany.

Reil, T. and Husbands, P. (2002). Evolution of central pattern generators for bipedal walking in a real-time physics environment. *IEEE Transactions Evolutionary Computation*, 6(2):159–168.

Reisinger, J. and Miikkulainen, R. (2007). Acquiring evolvability through adaptive representations. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO ’07, pages 1045–1052, New York, NY, USA. ACM.

Riek, L. D. (2017). Healthcare robotics. *Commun. ACM*, 60(11):68–78.

Righetti, L., Buchli, J., Mistry, M., Kalakrishnan, M., and Schaal, S. (2013). Optimal distribution of contact forces with inverse-dynamics control. *International Journal of Robotics Research*, 32(3):280–298.

Röfer, T. (2005). Evolutionary gait-optimization using a fitness function based on proprioception. In Nardi, D., Riedmiller, M., Sammut, C., and Santos-Victor, J., editors, *RoboCup 2004: Robot Soccer World Cup VIII*, pages 310–322, Berlin, Heidelberg. Springer.

Römmermann, M., Ahmed, M., Quack, L., and Kassahun, Y. (2011). Modeling of leg soil interaction using genetic algorithms. In *Proceedings of the 17th International Conference of the International Society for Terrain-Vehicle Systems*, ISTVS.

Römmermann, M., Edgington, M., Metzen, J. H., de Gea, J., Kassahun, Y., and Kirchner, F. (2008). Learning walking patterns for kinematically complex robots using evolution strategies. In Rudolph, G., Jansen, T., Beume, N., Lucas, S., and Poloni, C., editors, *Parallel Problem Solving from Nature – PPSN X*, pages 1091–1100, Berlin, Heidelberg. Springer.

Römmermann, M., Ahmed, M., Quack, L., and Kassahun, Y. (2011). An application of genetic algorithms to model leg–soil interaction. In *Proceedings of the 4th International Workshop on Evolutionary and Reinforcement Learning for Autonomous Robot Systems*, ERLARS 2011.

Römmermann, M., Bartsch, S., and Haase, S. (2010). Validation of simulation-based morphology design of a six-legged walking robot. In *Emerging Trends in Mobile Robotics*, pages 895–902. World Scientific.

Römmermann, M., Kühn, D., Cordes, F., Yoo, Y.-H., and Kirchner, F. (2009a). Concept evaluation of modeling terrain mechanics by a neural network. In *Proceedings of the 11th European Regional Conference of the International Society for Terrain-Vehicle Systems*, ISTVS.

Römmermann, M., Kühn, D., and Kirchner, F. (2009b). Robot design for space missions using evolutionary computation. In *2009 IEEE Congress on Evolutionary Computation*, pages 2098–2105.

Sakagami, Y., Watanabe, R., Aoyama, C., Matsunaga, S., Higaki, N., and Fujimura, K. (2002). The intelligent ASIMO: system overview and integration. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pages 2478–2483.

Saravanan, N. and Fogel, D. B. (1995). Evolving neural control systems. *IEEE Expert*, 10(3):23–27.

Schilling, M., Hoinville, T., Schmitz, J., and Cruse, H. (2013). Walknet, a bio-inspired controller for hexapod walking. *Biological Cybernetics*, 107(4):397–419.

Schilling, M., Paskarbeit, J., Schmitz, J., Schneider, A., and Cruse, H. (2012). Grounding an internal body model of a hexapod walker control of curve walking in a biologically inspired robot. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2762–2768.

Sendhoff, B. and Kreutz, M. (1999). Variable encoding of modular neural networks for time series prediction. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, volume 1, pages 259–266.

Sims, K. (1994). Evolving 3D morphology and behavior by competition. *Artificial Life*, 1(4):353–372.

Smith, R. (2005). Open Dynamics Engine, http://www.ode.org.

Spenneberg, D. (2006). *Bioinspirierte Kontrolle von Laufrobotern*. Doctoral dissertation, University of Bremen, Germany.

Spenneberg, D., Albrecht, M., and Backhaus, T. (2005a). M.O.N.S.T.E.R.: A new behavior based microkernel for mobile robots. In *Proceedings of the 2nd European Conference on Mobile Robots*.

Spenneberg, D. and Kirchner, F. (2007). The bio-inspired SCORPION robot: Design, control & lessons learned. In Zhang, H., editor, *Climbing and Walking Robots*, chapter 9. IntechOpen, Rijeka.

Spenneberg, D., Strack, A., Hilljegerdes, J., Zschenker, H., Albrecht, M., Backhaus, T., and Kirchner, F. (2005b). ARAMIES: A four-legged climbing and walking robot. In *Proceedings of 8th International Symposium iSAIRAS*.

Spröwitz, A., Tuleu, A., Vespignani, M., Ajallooeian, M., Badri, E., and Ijspeert, A. J. (2013). Towards dynamic trot gait locomotion: Design, control, and experiments with Cheetah-cub, a compliant quadruped robot. *The International Journal of Robotics Research*, 32(8):932–950.

Stanley, K. O. (2004). *Efficient Evolution of Neural Networks through Complexification*. Doctoral dissertation, Artificial Intelligence Laboratory, The University of Texas at Austin.

Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127.

Straube, S., Rohn, M., Römmermann, M., Bergatt, C., Jordan, M., and Kirchner, E. A. (2011). On the closure of perceptual gaps in man-machine interaction: Virtual immersion, psychophysics and electrophysiology. In *Perception - ECVP Abstract Supplement. European Conference on Visual Perception (ECVP-2011)*, volume 40, pages 177–177.

Vaario, J., Onitsuka, A., and Shimohara, K. (1997). Formation of neural structures. In *Proceedings of the Fourth European Conference on Artificial Life, ECAL97*, pages 214–223. MIT Press.

Valsalam, V. K. and Miikkulainen, R. (2009). Evolving symmetric and modular neural network controllers for multilegged robots. In *Exploring New Horizons in Evolutionary Design of Robots: Workshop at the 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IROS.

Ventrella, J. (1994). Explorations in the emergence of morphology and locomotion behavior in animated characters. In *Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 436–441, Cambridge, MA, USA. MIT Press.

Vernaza, P., Likhachev, M., Bhattacharya, S., Chitta, S., Kushleyev, A., and Lee, D. D. (2009). Search-based planning for a legged robot over rough terrain. In *2009 IEEE International Conference on Robotics and Automation*, pages 2380–2387.

Weidemann, H.-J., Pfeiffer, F., and Eltze, J. (1994). The six-legged TUM walking robot. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 2 of *IROS'94*, pages 1026–1033.

Whitley, D., Dominic, S., Das, R., and Anderson, C. W. (1993). Genetic reinforcement learning for neurocontrol problems. *Machine Learning*, 13(2):259–284.

Wieber, P.-B., Tedrake, R., and Kuindersma, S. (2016). Modeling and control of legged robots. In Siciliano, B. and Khatib, O., editors, *Springer Handbook of Robotics*, pages 1203–1234, Cham. Springer International Publishing.

Wiedemann, H. (2017). *Untersuchung des Übertragungspotentials menschlicher Gelenkwinkelverläufe zur Erreichung eines stabilen Gehverhaltens am Beispiel des Humanoiden RH5*. Master thesis, Hochschule Bremen, Germany.

Wieland, A. (1991). Evolving controls for unstable systems. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN-91)*, pages 667–673.

Xin, G., Lin, H., Smith, J., Cebe, O., and Mistry, M. (2018). A model-based hierarchical controller for legged systems subject to external disturbances. In *2018 IEEE International Conference on Robotics and Automation*, ICRA, pages 4375–4382.

Yao, X. (1999). Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447.

Yoo, Y., Ahmed, M., Römmermann, M., and Kirchner, F. (2009a). A simulation-based design of extraterrestrial six-legged robot system. In *2009 35th Annual Conference of IEEE Industrial Electronics*, pages 2181–2186.

Yoo, Y.-H., Abdenebaoui, L., Rohn, M., Römmermann, M., and Kirchner, F. (2009b). Modelling and simulation of mobile six-legged robot with leg-soil contact. In *Proceedings of the 11th European Regional Conference of the International Society for Terrain-Vehicle Systems*, ISTVS.

Yoo, Y.-H., Jung, T., Römmermann, M., Rast, M., Rossmann, J., and Kirchner, F. (2010). Developing a virtual environment for extraterrestrial legged robot with focus on lunar crater exploration. In *Proceeding of 10th International Symposium on Artificial Intelligent, Robotics and Automation in Space*, iSAIRAS.

Zhang, J. and Chen, Q. (2007). Learning based gaits evolution for an AIBO dog. In *2007 IEEE Congress on Evolutionary Computation*, pages 1523–1526.

Zucker, M., Bagnell, J. A., Atkeson, C. G., and Kuffner, J. (2010). An optimization approach to rough terrain locomotion. In *2010 IEEE International Conference on Robotics and Automation*, pages 3589–3595.

Zykov, V., Bongard, J., and Lipson, H. (2004). Evolving dynamic gaits on a physical robot. In *Genetic and Evolutionary Computation —- GECCO 2004*, volume 4, Berlin Heidelberg. Springer.