Development of an optimization tool for Myode

A case study of hopping

Bachelor-Thesis

submitted in partial fulfilment of the requirements for
the degree of Bachelor of Science (B. Sc.)

Humboldt-Universität zu Berlin
Faculty of Mathematics and Natural Sciences II
Department of Computer Science

Eidgenössische Technische Hochschule Zürich
Department of Mechanical and Process Engineering
Institute of Robotics and Intelligent Systems

Referees:
Prof. Dr. Verena V. Hafner
Prof. Dr. Fumiya Iida

Supervised by:
Dr. Hugo Gravato Marques
Dr. Surya Nurzaman
Utku Culha

Author: Cordt Voigt
Born on: May 10, 1988
in: Essen, Germany
Submitted on: July 4, 2013
Contents

Abstract iii

Acknowledgements v

1 Introduction 1

2 Simulation environment and leg model 3
  2.1 Caliper and Myode 3
  2.1.1 Installation and set-up 3
  2.1.2 The model 4

3 The optimization plug-in 6
  3.1 Software Engineering approach 6
  3.1.1 Design and functionality requirements on the optimization plug-in 7
  3.1.2 Design requirements on the controllers 8
  3.1.3 Software architecture 9
  3.2 Evolution component 10
  3.2.1 Evolution 10
  3.2.2 Parameters 11
  3.2.3 Population, Individual, Genotype 11
  3.2.4 Operators 11
  3.2.5 Nature 12
  3.2.6 Fitness 12
  3.3 Optimization component 12
  3.4 Functioning of the controllers 15

4 Methods 16
  4.1 The PD-Controller 16
  4.2 Tuning of the Evolutionary Algorithm 17

5 Results 19
  5.1 Results of the PD-Controller 19
  5.2 Results of the optimization process 21
    5.2.1 Manual parameter tuning 22

6 Additional Results 24
  6.1 The SMT-Controller 24
    6.1.1 Resulting matrices 25
  6.2 The Hopping-Controller 26
# 7 Conclusion and future work

7.1 The leg model ........................................... 28
7.2 The optimization plug-in ............................ 28
7.3 The controllers ......................................... 29

A Symbols, Indices, Acronyms and Abbreviations 31

B Screenshots from Caliper ............................... 33

Bibliography .................................................. 35

Statement of authorship .................................. 37
Abstract

The design, implementation, simulation or assembly of different types of robots often involves various optimization problems such as design- or controller-optimization. Whereas for certain tasks there exist well adapted methods, a generic technique to deal with different problems are evolutionary algorithms. This thesis presents the implementation of an optimization plug-in for Myode, a framework for the simulation of tendon-driven robots, based on the evolutionary algorithm approach. Furthermore three controllers will be used to demonstrate the optimization process. The results show the increasing fitness of the parametrized individuals, a comparison to manual parameter tuning and changes due to different settings of the model.
Acknowledgements

This work was carried out at the Bio-Inspired Robotics Laboratory at the Eidgenössische Technische Hochschule Zürich in collaboration with the chair of Kognitive Robotik at the Humboldt-Universität zu Berlin.

I want to thank Prof. Verena V. Hafner and Prof. Fumiya Iida for giving me the possibility to do a project during my stay abroad at the ETH Zürich and their support in every respect. Furthermore I want to thank my supervisors Dr. Hugo Gravato Marques, Dr. Surya Nurzaman and Utku Culha who gave me lots of input during the entire assignment.
Chapter 1

Introduction

One of the projects the Bio-Inspired Robotics Laboratory (BIRL) is currently working on, is the Myorobotics Project [2], which is a collaborative project of the Technische Universität München (TUM), the Eidgenössische Technische Hochschule (ETH), the Fraunhofer IPA, the University of Bristol (UB) and the University of West of England (UWE). It aims at creating a framework for musculoskeletal robot development. As a part of the framework a software tool is provided to analyse, design, control, simulate and optimize so called Myorobots. This Bachelor Thesis describes the modelling and implementation of a modular, easily extensible optimization plugin for Myode, that uses an evolutionary algorithm approach to be able to deal with a wide variety of different optimization problems. To show the performance of the plug-in three controllers are presented which will be introduced here.

Amongst others, the current research at the BIRL is concerned with self-organisation of reflexive behaviour. Basically the hypothesis is that reflexes can emerge by creating spontaneous motor activity on a musculoskeletal system by evaluating the resulting sensor and motor activity (See, e.g. [3]).

The framework[1] which is the basis of one of the controllers that will be described in this thesis, realises the learning process by single muscle twitches (SMTs). The correlation between the muscle contractions and the induced sensory information determines a set of reflex circuits. Twitching the muscles used in the model and obtaining weight matrices for this set of muscles will be realized by the so called SMT-Controller.

In previous work it has been shown(see [3]) that two types of reflex circuits are sufficient to achieve hopping in a musculoskeletal leg model. One type of circuit transforms muscle deformation, i. e. change in muscle length (received from muscle spindles) into motor signals. The second type transforms the rate of change in muscle length (also received from muscle spindles) into motor signals. In each reflex circuit a single parameter controls the extent to which each muscle is contracted. The learning process itself determines these two parameters, which is referred to as weights. One task of, for example, a leg model is stable hopping in presence of gravity. That is, without adjusting the learned weights and therefore the muscles active force function, the leg will in general not be able to level off to a stable hopping amplitude and frequency. In order to achieve, for instance, stable hopping, all the weights of the two groups are adjusted by a single parameter. Thus, only two parameters are necessary for all reflex circuits of the created model. Applying the active force on the muscles using a set of learned weights and the two adjustable parameters acting on them will be carried out by a second controller - the Hopping-Controller.

As discussed later, there are still a few uncertainties within the model and the simulator, such that the optimization plug-in cannot be applied to this Hopping-
Controller to show the performance of the plug-in. Therefore a third controller is introduced that aims at bringing the leg in a certain position using a proportional and a derivative gain, a so called Proportional-Derivative (PD)-Controller. Since this controller is not subject to the aforementioned uncertainties it can be used to demonstrate the performance of the optimization plug-in as well as reveal the behaviour of different muscle settings with respect to optimization strength (fitness), reference angle development and muscle activation behaviour.

In chapter 2 the simulation environment will be described in more detail as well as the leg model, that is the foundation of all the simulations that were carried out. Chapter 3 demonstrates the requirements and the implementation of the optimization plug-in, emphasizing the evolutionary algorithm at its core. In chapter 4 the methods that were used to evaluate the optimization plug-in are described by the example of the PD-Controller, due to the aforementioned uncertainties using the other controllers. Chapter 5 illustrates the application of the PD-Controller as well as the performance of the optimization plug-in applied to it. Chapter 6 sums up the results obtained from the SMT- and the Hopping-Controller whereas chapter 7 gives a conclusion of this work and names possible tasks for future work.
Chapter 2

Simulation environment and leg model

The following chapter shortly describes Caliper, a robot simulation framework for tendon-driven robots [4] that is mainly developed at the TUM and Myode, which is a software tool based on Caliper that will be provided with the Myorobotics framework. Finally the properties and features of the leg model that is used for the simulations as well as a reference model used in previous work will be illustrated.

2.1 Caliper and Myode

As discussed in [4] there is currently a lack of simulation tools that support tendon-driven robots, although this class of robots is becoming increasingly important. To overcome this problem Caliper was developed as an extensible robot simulation framework for tendon-driven robots within the Embodied Cognition In A Compliantly Engineered Robot (ECCEROBOT) Project [5].


The overall project in which this project was carried out is the Myorobotics project and the following descriptions are taken from its website [2]. As already mentioned the project aims at providing a tool-kit for developing musculoskeletal robots. It offers so called “design primitives”, which are low-cost production building blocks of the robots as well as tools that offer self-diagnosis and self-calibration. It finally also provides a software tool to design, analyse, control, simulate and optimize Myorobots. The latter refers to the Myorobotics Development Environment (Myode) - modelling, implementing and testing the optimization plug-in as well as three certain controllers will be subject to the following chapters.

2.1.1 Installation and set-up

As mentioned before the recommended setting is Ubuntu 12.04 LTS and Eclipse CDT, so the following description is based on this set-up. Basically the installation of Myode for development purposes consists of four steps:

- Adding the necessary repositories
- Installing the necessary packages
Chapter 2. Simulation environment and leg model

- Retrieving the latest versions from the Caliper and Myode Apache Subversion (SVN) [9] repositories
- Out-of-source build using `cmake` [10]

The Out-of-source build is recommended but not necessary. Detailed installation instructions can be found on the Myorobotics repository [11] and shall not be described here any further. This set-up provides five folders, all of which are represented in Eclipse:

1. caliper - contains the source code of Caliper. It is connected to the SVN and thus versioned.
2. caliper@build - contains the build of Caliper. It is not connected to the SVN, but is targeted by `make` [12] as the building target, containing the compiled files.
3. MYODE - contains the source code of Myode. It is connected to the SVN and thus versioned.
4. MYODE@build - contains the build of Myode. It is not connected to the SVN, but is targeted by `make` as the building target, containing the compiled files.
5. models - contains the models available for the simulation. The models can be described using Collaborative Design Activity (COLLADA) [13].

The process of creating and using the files mentioned in the latter in Caliper is described in Wittmeier et al. [4]. The model that was used in this project will be described qualitatively in the following section.

2.1.2 The model

Although the optimization plug-in, which is the main goal of this project, is designed to cover a wide variety of design- and controller-optimization problems using arbitrary models, this thesis presents only one model that is used for all simulations. It is based on the leg model proposed in Bharadwaj [1] with minor, but as discussed later, important changes.

The main difference between Bharadwaj and this work is the environment in which these simulations are carried out. Since the simulation of the leg model in the former is performed in Matlab [14] the aim is to compare the results produced by Myode qualitatively with them.

The two models use different muscle settings. In Bharadwaj there are six muscles attached - from the hip to the thigh and the thigh to the shank whereas the model used here is extended by three muscles and a foot. The muscle settings are illustrated in figure 2.1 and figure 2.2. One can argue that this change in the muscle setting decreases or even annihilates the comparability of the results, respectively, there is no certainty that a relation between the results exists. As a matter of fact it will become clear in 6.1 that the different setting does indeed change the system to a certain degree, but the two systems still have properties that allow for a comparison. One feature of the models that definitely needs to be identical in both in order to be able to compare them, is the employed muscle model. By now, Myode, respectively Caliper, offers two different muscle models: the Eccerobot tendon-driven muscle model [18] and the popular Hill-muscle model. Bharadwaj as well as this work continuously used the three-element Hill-muscle model, where each muscle’s behaviour is determined by three parameters which are directly adopted from Bharadwaj as follows:
• *Spring constant* $k = 140$
• *Damping constant* $B = 5$
• *Contractile Element* $CE = 75$

The influence of the single parameters on the muscle’s behaviour as well as the aforementioned choice are not part of this work.

With this model and the simulator the foundations for first investigations on, for example, properties and features of the muscle model or the behaviour of the leg-model in its environment are given. Two certain tasks will be described in detail in 4.1 and 6.1. The former will be used to illustrate the performance of the optimization plug-in.
Chapter 3

The optimization plug-in

In this chapter the design, the modelling and the implementation of the optimization plug-in for Myode will be described. The plug-in was meant to fulfill several aspects concerning its design and functionality. Starting from the general proceeding with respect to Software Engineering (SE) used here, these aspects will be explained subsequently, followed by an illustration of the chosen approach to match them. Since they are embedded by the plug-in and are subject to the actual optimization process, the design requirements on the controllers as well as their basic functionality are of concern in this chapter.

3.1 Software Engineering approach

One of the major steps in modern Software Engineering (SE) is the requirements specification. Due to its great importance during the development process of commercial software it is standardized by the Institute of Electrical and Electronics Engineers Standards Association (IEEE-SA) as ANSI/IEEE Std 830-1984 [15], where ANSI refers to the American National Standards Institute. Although it is one of the first steps in important software development process models like the Waterfall Model, as described in Royce [16], or the V-Model (see, e.g., [17]), the proceeding taken here neither uses one of these models nor follows the process of requirements specification as proposed, for example, by the IEEE-SA. Instead, starting from a small set of rather general requirements, the development is carried out using software prototyping (a comparison between specification and prototyping is given, e.g., in [19]). This course of action was chosen for several reasons. First of all, neglecting "common" software development techniques arises with the nature of this kind of software. Since Myode is developed within an European wide project, several Universities participate in the development process and are therefore responsible for different parts of the software. This does not necessarily exclude these parts to be created using a SE development process as described above. Furthermore the required functionalities of a module of Myode, respectively the optimization plug-in, are roughly defined but will be subject to changes during the development - a few of them due to misconceptions in the overall framework or the plug-in itself, others due to the design of the controllers, which underlie, similarly, changes during its development as discussed later.

In contrast to commercial software the Myorobotics project does not yield profit or fulfill customer demands and is accordingly not bound to requirement specifications on the level of the software modules. The general assumption is that this opens up more possibilities concerning the functionality and the usability of the product.

Finally, a strong argument to use software prototyping is that many implementa-
tions, even of the same module, are carried out by different persons - unlike in a project not together, but subsequently. Sticking to a certain software development process model would require every new engineer to reconstruct the entire planned software model.

The approach used here is depicted in figure 3.1. As can be gathered from the figure, a first draft of the architecture and the class diagrams was created after defining the general design and functionality requirements. These will be described in the sections 3.1.1 and 3.1.2. Moreover, during the prototype iterations, behavioural diagrams were created as necessary. The architecture is described in section 3.1.3, class diagrams and the behavioural diagrams of the final iteration are given in the according sections. After having given the proceeding with respect to SE, the next sections deal with the design and functionality requirements of the optimization plug-in with particular emphasis on evolutionary algorithms (EAs), a strategy that will be explained in section 3.2. Moreover the design requirements and the general functioning of the controllers will be described and finally the software architecture is discussed.

3.1.1 Design and functionality requirements on the optimization plug-in

The overall goal of the development of the optimization plug-in was to be able to apply it on arbitrary optimization problems, especially without having any knowledge about the certain problem. This suggests the usage of a generic optimization strategy that does not restrict the problem space. The strategy used here is known as evolutionary algorithms (EAs). The general cycle of this kind of algorithms is shown in figure 3.2. EAs itself already form a big class of different strategies - the general cycle and the algorithm used here will be explained in section 3.2. To fulfil the requirement of handling arbitrary optimization problems, not only a generic strategy as the EAs are necessary, but the design of the plug-in also must allow for any possible input. In this thesis only controllers, which are introduced in the next
section, are used for optimization purposes. Hence, restating the former, to be able to optimize arbitrary controllers, these should offer parameters that are changed in order to optimize the outcome - they will be denoted as optimizable parameters. Parameters that indicate whether the performance of the controller, respectively the model it is acting on, improved will be denoted as reference parameters.

The aforementioned requirements refer to the design of the plug-in. The functionality requirements basically draft the process of using the plug-in. It can be described roughly as follows:

- Load an arbitrary controller
- Select optimizable parameters passed by the controller
- Select reference parameters passed by the controller
- Tune the EA
- Start the optimization

A detailed description of this process based on the final interface design is given in section 3.3. It is assumed that this proceeding implementing the plug-in, makes it modular and easily extensible which is one of the main goals of this project.

### 3.1.2 Design requirements on the controllers

As described in the former section, one of the main goals was to design the optimization plug-in for arbitrary controllers which was taken into account of its implementation. In turn there are also requirements on the design of the controllers to guarantee interoperability with the plug-in. These requirements can be narrowed down and described as follows:

- The controllers have to offer a standardized way to receive and change its parameters, designated as optimizable or reference parameters
- The controllers must meet implementational requirements, such that they can be loaded in the user interface by the optimization plug-in

The latter refers to implementation details, e.g. inheritance and graphical user interface (GUI) programming, and will not be described any further. The former suggests the usage of an interface that offers the necessary functionality; it is the IController interface as depicted in figure 3.5. Its meaning in the overall context is described in the next part. This section puts emphasis on the parameters. The approach here was to implement the entire parameter functionality in the interface, such that the necessary methods the controller has to offer to the optimization plug-in, are available by simply implementing the interface. By doing so it is ensured, that both, the controllers and the optimization plug-in, deal with the parameters in the same way and can therefore be used together. Summing up, if the second requirement is neglected, which does not restrict the controller in any way, every controller that implements the IController interface can be used for optimization purposes by the optimization plug-in. The formalism, allowing the user to define the optimization problem as needed, is realized as follows: The parameters stored in an instance of an IController are basically designated as optimizable or reference. These parameters are then displayed by the optimization plug-in, giving the user the possibility to select them properly. The plug-in itself has no information about these parameters or their purpose. It just displays them and leaves the task of combining them properly to the user. This generic structure and formalism perfectly match the goal of a modular tool that is not restricted to a certain problem space. Having established the background of the design requirements the next part deals with the software architecture.
3.1.3 Software architecture

Caliper already provides a good overview of the architecture as well as detailed class descriptions provided by Doxygen [21]. Myode also makes use of Doxygen - at the current development stage the documentation is not published but can be created from the source code at any time. In this section only the parts of Caliper and Myode that are necessary to arrange the optimization plug-in and the controllers in the whole picture will be described. Figure 3.3 outlines these parts of the software architecture. Although this is already a simplification, only a few aspects shall be emphasized here. First of all the diagram shows accurately that Caliper is extended by Myode, respectively an Optimization and a Controller component, both of which implement the IPlugin interface provided by Caliper. This is how plug-ins have to be added to extend Caliper as described in [11]. Secondly, amongst others, the Myode components make use of three interfaces provided by Caliper:

1. ISimulator, offering access and control to the simulator plug-in
2. IMuscle, offering access and control to the Hill-muscles, present in a certain simulator
3. IModel, offering access and control to the model, present in a certain simulator

The relevance to the optimization plug-in and the controllers will become clear in the following sections. The fourth interface depicted in figure 3.3 is the IController interface. It offers access and control to the controllers, especially adding, accessing and updating all the associated parameters. The last important point here is, that the Optimization component not only contains the Optimization instance but also the Evolution component, that carries out the EA. It is described in detail in the next section.

---

Figure 3.3: Fragmentary picture of the software architecture based on the design of Caliper and Myode
3.2 Evolution component

The evolution component contains the EA, the optimization plug-in makes use of. Basically this class of algorithms follows the biological pattern of evolution, that is, it creates individuals that represent different parametrizations of the optimization problem. It then evaluates them according to certain characteristics using a so called fitness-function, determining which individuals, respectively parametrizations, are the fittest according to these characteristics. There are different possibilities to proceed with this population - it is the programmer’s challenge to offer different strategies and the researcher’s challenge to select one that fits the conditions of the problem. The general cycle of an arbitrary EA as depicted in 3.2 can be described as follows:

1. Initialization
   A population, consisting of individuals showing different characteristics is created

2. Evaluation
   The population is evaluated according to a certain fitness function

3. Replacement
   Individuals, that have shown weak behaviour are replaced by those who were fitter

4. Stop criteria
   If the individuals have reached a certain fitness value or a time- or iteration-constraint is reached, the EA stops

5. Selection
   Depending on the algorithm’s set-up, individuals are chosen to be subject to so called Genitors - in general mutation or crossover. Whereas a mutation changes the characteristics of an individual, intending to create randomly a fitter one, crossover combines characteristics of more than one parent individual, intending to unite their assets

6. Variation
   The population is changed by applying the Genitors on them, creating the offspring - the population for the next iteration. The population of every iteration is referred to as generation

As stated above, it is the programmer’s challenge to offer different strategies. One of the most important factors influencing the performance of the algorithm is the fitness function. Since it usually determines which individuals eventually will be dropped and which will be used for variation, it has to be chosen carefully. It takes a very sophisticated and generic user interface to allow the researcher himself creating it, therefore the approach in this project was to provide only pre-implemented fitness functions the user can choose and design the EA such that more functions can be added subsequently without considerable effort. Figure 3.4 shows a simplified class diagram of the EA. All classes are designed so that additional functionalities can be added easily. Since the nomenclature in the diagram is differing from the one given above, the diagram will be explained in detail in the following sections.

3.2.1 Evolution

The class “Evolution” basically represents the EA itself as it contains the individual parts of it. It offers the methods to create the initial population and every subsequent generation arising from it. Moreover, access to the population and to the parameters class is provided.
3.2.2 Parameters

The “Parameters” class is a very generic class, not bound to a certain data type. To access single parameters, a so called “access name” is used that has to be unique throughout this class. The class is used in two contexts within the evolution component. Firstly, as the name suggests, it parametrizes the EA. That is, information like the population size (number of individuals), the number of generations, the stop criteria and all information concerning the parameters to optimize as well as the reference parameters is stored in there. A reference to the class is passed to every component of the EA that makes use of these parameters. Secondly, it is used in the role of genes, containing all the attributes that belong to an individual. Due to the generality of the “Parameters” class the individuals can have any possible characteristics that can be expressed by data types of the programming language. The controller-interface manages and offers parameters in a very similar way. This suggest to implement an interface, designed for the purpose of managing and offering arbitrary parameters. Future work should deal with this issue.

3.2.3 Population, Individual, Genotype

The classes “Population”, “Individual” and “Genotype” represent the population, its individuals and their characteristics, respectively. Although the functionalities could be combined to one or two classes, the abstraction is chosen to clearly represent the features of an EA.

3.2.4 Operators

The “Operators” class implements and offers the Genitors that act on the population. Using them it also determines each individual’s fitness. As all other parts of the EA it is controlled by the “Parameter” class. The current development stage copes with the Genitor mutation in combination with a tournament. The tournament determines the portion of individuals that will be subject to a mutation.
3.2.5 Nature

The “Nature” class provides methods that rely on chance and are necessary within the EA. An example of their usage is the strength of a mutation. The underlying mechanism makes use of a so-called pseudorandom number generator. In computability theory, pseudorandomness refers to results that cannot be predicted by efficient algorithms. For the purposes of the EA described here, this generator is perfectly sufficient.

3.2.6 Fitness

The static “Fitness” class contains the available fitness functions. As stated above they have to be implemented by the programmer and cannot be generated in the user interface. The “Operator” class uses the “Fitness” class to evaluate the individuals, the actual fitness function is determined by a parameter. The usage of the EA within the optimization process will be described in the following section.

3.3 Optimization component

The Optimization component consists of three parts:

- The classes, belonging directly to the optimization plug-in
- The Evolution component, adding an EA to the plug-in
- The controller- and simulator-interface, connecting up to one controller and an arbitrary number of simulators to the optimization plug-in

Although it seems odd, the implementation allows for no connected simulator. Caliper is designed as a modular software that is extended by plug-ins. Thus, it is possible that no simulator is present. However, the optimization plug-in makes sure that a simulator is connected before it can be used. Figure 3.5 shows a class diagram, illustrating the aforementioned structure. Only very few attributes and methods are represented in the diagram. Especially the controller-interface additionally offers access to a group of class-members that represent its parameters as emphasized in section 3.1.2. Important to mention is that the optimization plug-in manages
the interaction of all the participants of the optimization process. Additionally, the “OptimizationThread” class is necessary to run the controller simultaneously with the simulation. An explicit explanation of single methods would exceed the scope of this work, for this the reader is referred to the Doxygen documentation or to the source-code itself. Instead a sequence diagram is given (Figure 3.6), sketching the optimization process. The diagram represents the logical course, rather than explicit function calls or statements. Once the user has started the optimization, all the parameters, e.g. population size, number of generations, choice of fitness function, etc., are stored in the parameters instance (to be distinguished from the parameters instance of every individual’s genotype in the role of genes) and the EA is started. Then, according to the set-up stored in the parameters instance, the EA creates the first generation. The optimization plug-in now retrieves the single individuals of the generation and simulates them, using the simulator’s interface (not depicted). Once all individuals have been simulated, the optimization plug-in tells the EA to create the child population, respectively the next generation. As already described in 3.2, the EA evaluates the individuals according to the chosen fitness function and applies the Genitors on those that were chosen for the variation. Finally the optimization plug-in begins simulating the next generation until the desired number of generations is reached.

There are two important aspects that shall be emphasized here: Firstly, the EA knows nothing about the simulation or the controller, none of which are even depicted in the diagram. The EA uses every proper data input given by the optimization plug-in and is therefore not restricted to certain problem settings as claimed in the design requirements. Secondly, the way the individuals are evaluated can easily be extended: The fitness function used is determined by an according parameter, which is retrieved from the parameter class. Afterwards this function is applied to every single individual. This proceeding simplifies the process of adding new functions by reducing it to two simple steps: A new fitness function has to be added to the “Fitness” class and the user must be given the opportunity to select this fitness function in the user interface. This approach copes with the requirement of easy extensibility.

Figure 3.6: Simplified sequence diagram of the optimization process
The concrete result of the implementation is shown in figure 3.7 and shall be described here: The “Optimization” tab (a); see appendix B, figure B.2) offers the user the possibility to start and stop the optimization process. It also shows the progress of the EA. The “Evolution” tab (displayed) allows to tune the EA. In the part on the top, general settings can be made. The last line, “Measurement duration per individual”, refers to the portion of the whole simulation, that is taken into account evaluating the fitness of a single individual. The lower part contains four tabs to tune the EA with respect to the certain problem:

- “Parameters” (c); see appendix B, figure B.1) allows to select the parameters that were passed as optimizable (see 3.1.1) for the optimization process
- “Fitness” (d); see appendix B, figure B.3) allows to select the reference parameter, its desired value and the fitness function that should be applied to this problem
- “Mutation” (e), displayed), allows to select the Genitor mutation, the maximum strength of the mutation on every optimizable parameter, the tournament pressure and two additional settings that will not be explained here
- “Crossover” (f), not displayed) is meant to allow for the Genitor crossover. It is not implemented yet.
During the optimization statistical data is recorded to analyse the process. Finally, the general functioning of the controllers with respect to the above listed requirements will be explained.

3.4 Functioning of the controllers

Additionally to the description of the formalism when used for the optimization (see 3.1.2), the general functioning of the controllers will be described here. As can be gathered from the class diagram in figure 3.5 the controller interface offers a “step” and a “resetController” method. These two methods are used to run and reset the controller. Since they are defined in the interface this is true for every controller and makes sure that they are used in the same way. As mentioned in section 3.3 the controller is “stepped” simultaneously to the simulation using a thread, therefore the “step” function is called very frequently. For the experiment a frequency of 500 Hertz was being used, which corresponds to the simulator’s frequency. This approach means that there is no message or method flow as it was described, for example, with the sequence diagram in figure 3.6. Instead the controller needs to keep an information, that determines its current behaviour. This can be achieved by using a state, usually a number, that represents the current setting and therefore the next step of the controller. Since it is unique to every controller it is referred to as an “intrinsic state”. The information concerning the behaviour is only necessary if the controller is more sophisticated and accordingly only used in the case of the SMT-Controller. This intrinsic state is implemented as a state machine as shown and described in 6.1.

The next chapter describes the methods that were used to verify the optimization plug-in.
Chapter 4

Methods

In this chapter the design, the modelling and the implementation of a so called Proportional-Derivative (PD)-Controller will be described. Starting from a motivation, illustrating its relevance in terms of developing tendon-driven robots, the need for optimization is explained. The second part points out how the EA can be tuned to meet the problem’s needs as good as possible, emphasizing a reasonable fitness function to determine the fitness of each individual.

4.1 The PD-Controller

A PD-Controller, where “P” designates “proportional” and “D” designates “derivative”, is a very generic controller that tries to minimize the error with respect to a certain setpoint, by comparing it to the actual value of this quantity and using this deviation to tune the mechanism that affects the setpoint. The application of this controller on the leg model, to realize the task of reaching a certain leg position, will be described here.

In a tendon-driven robot, moving parts of the robot is not as straight forward as having a motor right at the hinge that extends this part to the desired position. If the activation of the muscle is too strong it will overshoot the desired position. If it is too weak it will not arrive. The question is, if there is a certain activation, such that it arrives exactly at the desired position. Even if a value were found it would strongly depend on the current setting of the robot, that is, it depends on other attached muscles, gravity and its current position. An approach to solve this problem is to use a PD-Controller. A fairly simple task is to keep the leg at a certain angle with respect to one of the joints. Figure 4.1 shows the values of the hip joint angle in the simulator.

To apply the PD-Controller to this task, a setpoint

![Figure 4.1: Values of the hip joint angle in the simulator (given in radians)](image)

as well as the elements that affect it have to be defined. The former is the hip joint
angle and is denoted as $\alpha_{\text{ref}}$; it can only be controlled by determining the muscle activation with the aid of the PD-Controller:

$$e(t) = \alpha_{\text{ref}} - \alpha(t),$$  \hspace{1cm} (4.1)

$$\dot{e}(t) = \frac{e(t) - e(t-1)}{\Delta t},$$  \hspace{1cm} (4.2)

$$A_i(t) = P_i e(t) + D_i \dot{e}(t),$$  \hspace{1cm} (4.3)

where $e(t)$ is the error at time $t$, $\alpha_{\text{ref}}$, as already mentioned, denotes the hip joint angle, $\alpha(t)$ is the actual value of the hip joint angle at time $t$, $e(t)$ is the derivative of the error calculated with the time step of the simulator $\Delta t$, $A_i(t)$ is the activation of muscle $i$ at time $t$, $P_i$ is the proportional gain of the muscle $i$ and $D_i$ is the derivative gain of the muscle $i$.

Clearly, this controller can be regulated by tuning $P_i$ and $D_i$ properly. Talking about two parameters that can be adapted and the task to bring the leg in a certain position seems to pose an acceptable challenge. Indeed, as will be discussed in chapter 5, this can be done manually with sufficient results, but once the problems become more complex the need for an automated optimization arises.

### 4.2 Tuning of the Evolutionary Algorithm

The graphical user interface (GUI) of the optimization plug-in offers several possibilities to tune the entire process. The important parameters will be described in the following. EAs are characterized by the individuals with all their properties and the generation they belong to. Two important parameters are therefore the number of generations and the number of individuals. As a rule of thumb, knowing the problem space very well suggests a small number of individuals, if the range of possible solutions is mostly unknown, the number should be large. Whereas a well known problem space allows for very purposeful tuning, the other scenario needs the EA to probe different individuals in a very wide range. In most cases, the number of generations allows to determine the precision or quality of the solution. Furthermore, the GUI gives the user the possibility to determine the time, each individual is simulated, as well as the time, the reference values are recorded to be submitted to the fitness function. The latter allows, for example, to drop strong oscillations at the beginning of the simulation, evaluating the individual’s fitness. In the example given in section 5.1, the first two seconds could be neglected to focus on the development of the angle once the strong peaks are passed. The optimization plug-in offers the possibility to select any parameter that is designated as optimizable for the process and to decide in which range the values for the first generation should be. Consequently, the reference parameter, its value and the worst fitness that is possible, can be selected. By now the plug-in only allows for one reference parameter. The only implemented Genitor at this point in time is mutation with a tournament condition. That is, the user can choose the strength of the mutation for every optimizable parameter (i.e. the highest possible value at which the optimization parameter can be changed) and select how many individuals in each generation will be subject to a mutation. At the current development stage, the least fit individuals will always be selected for the mutation whereas the most fit individuals will always remain in the population. Finally, one of the implemented fitness functions can be chosen. The choice of it is crucial for the entire optimization process. A bad choice leads to a weak or even insufficient solution, constraints concerning the allowed solutions that are not taken into account flaw the optimization by allowing it to develop in a wrong direction. In the case of the task described in section 4.1 the choice of a sufficient fitness function is moderately easy. Since we want the leg
to arrive fast in the desired position oscillating as little as possible, using the mean of the absolute error fits the requirements. Mathematically spoken

$$F = \frac{\sum_{t=0}^{N} |\alpha_{ref} - \alpha_{t}|}{N},$$

(4.4)

where $F$ designates the fitness value and $N$ is the total number of discrete measurements.

This fitness function was used for all optimizations and led fast to good solutions as it will be described in the following chapter.
Chapter 5

Results

In this chapter the performance of the optimization plug-in is described, using the example of the PD-Controller. Starting from a discussion on the features of equation 4.1 and the signs of the gains, two simple settings and their scientific relevance are presented. These settings are then extended by an additional reference value and used for the optimization process. Finally, a comparison to manual parameter tuning is given.

5.1 Results of the PD-Controller

What behaviour can be expected according to equation 4.1? The proportional gain is directly connected to the error with respect to the desired position of the leg and thus accelerates it in the direction of the reference angle. The model does not allow for negative activation since the muscles can only be contracted (increasing activation) or relaxed (decreasing activation); the counteracting force that pushes the leg in its resting position is the gravitational force. If the formula results in negative values, they will be cut to zero. Having only this term would cause the muscle to oscillate heavily and the simulator to break down in most of the cases. The derivative gain acts on the rate of change of the error and works like a damper for the proportional term: The faster the activation due to the proportional term decreases the error, the stronger the derivative term decreases the force. Looking at the values of the Hip Joint angle it can be deduced, that in the case of a negative reference angle the gains have to be negative too. But this is not entirely correct. It rather has to be required that the gains of a muscle have the same sign as the direction they shall provide a push to, with respect to the reference value. To explain this requirement two settings will be described:

1. $\alpha_{ref} = -0.6$, Muscle: Iliacus
2. $\alpha_{ref} = -0.6$, Muscles: Iliacus, Gluteus maximus

The Iliacus muscle (figure 2.1) connects the hip with the femur and can lift the latter forward. Therefore, using no other muscle but the Iliacus, only negative reference values for the hip joint make sense. The Gluteus maximus muscle, simplified said, connects the hip with the femur on the back side of the leg and can lift the femur backward. Similarly, if it was the only muscle used, only positive reference values for the hip joint make sense. The signs of the gains have to be negative in the case of the Iliacus in combination with a negative reference value and positive in the case of the Gluteus maximus in combination with a positive reference value. Once both muscles are used, this choice of signs still makes sense: If the reference value is negative, the Gluteus maximus antagonizes the Iliacus once it overshoots $\alpha_{ref}$ and
vice versa. Otherwise the muscles would stop each other to reach the designated angle and relax, once the other muscle overshoots. Indeed, biology tells us that, simplified said, the antagonist of a muscle breaks its movement when, for example, one tries to lift his leg in a certain height. Figure 5.1 and 5.2 show the angle development of the Hip Joint in time of the two settings. It differs from the typical development of a value adjusted by a PD-Controller due to the one way activation in combination with the gravity. Quite more interesting is the development of the muscle activation in time as shown in figure 5.3 and 5.4. Whereas the one-muscle setting shows that the activation oscillates between zero and a value that is sufficient to bring the muscle to the desired angle, the two-muscle setting reveals the antagonizing behaviour of the Gluteus maximus: The Iliacus overshoots way stronger than in the former setting but the leg is then decelerated by the Gluteus maximus. In this section the application of a PD-Controller to the task of reaching a certain leg position was described as well as reasonable constraints on the choice of the gains were determined. But how can the values of the gains be determined properly and what characterizes a good choice? The latter is simple: the leg should reach the reference value as fast as possible and oscillate around it only very little. Determining the actual values of the gains can be done manually, but is not very accurate with respect to the claimed characteristics of a good choice and for the two-muscle setting already fairly time consuming. Applying the optimization plug-in to these tasks and finally a qualitative comparison to manual parameter tuning will be subject of the last two sections of this chapter.
5.2 Results of the optimization process

The first tested setting is, as described in the previous section, one or two muscles in combination with a reference value of -0.6 for the hip joint angle. The results are shown in figure 5.5 and 5.6. Both graphs show three curves that refer to the least fit and the most fit individual as well as the average of all individuals in each generation. The x-axis designates the generation, the y-axis the fitness value obtained from equation 4.2. The latter is log-scaled, since the values differ very little. The graphs have a few interesting features that will be emphasized in the following. First of all, the development of the most fit individual is quite similar in both muscle settings. That means that from this point of view, there is no advantage of one setting compared to the other. Also interesting is that the two-muscle setting seems to be more “stable”. Especially the least fit individuals show less peaks throughout the single generations. One unexpected feature of both graphs is, that the most fit individual’s fitness is able to decrease, although it was stated in section 4.2 that the most fit individual will always remain in the population. The simulator that is implemented in Culpier is deterministic. Nevertheless, there is a possibility for different episodes of subsequent simulations with the very same setting. It is most likely caused by the fact that different threads act on the same model. At least one thread carries out the simulation and another steps the controller. However, these aberrations can be neglected due to the fact that they are extreme small. Figure 5.7 shows the fitness values of subsequent simulations with the very same set-up. The simulator and the controller were both reloaded before every simulation. Finally,

Figure 5.5: Optimization of the PD-Controller; $\alpha_{ref} = -0.6$, Muscle: Iliacus

Figure 5.6: Optimization of the PD-Controller; $\alpha_{ref} = -0.6$, Muscles: Iliacus, Gluteus maximus

Figure 5.7: Deviation of the calculated fitness value during subsequent measurements with the very same set-up: $\alpha_{ref} = -0.6$, Muscle: Iliacus, $P = -3250.3$, $D = -572.518$
the graphs show that the first generation already had a very good result. This is
due to the aforementioned knowledge of the problem space. Without considering
the possible values for the P- and D-gain of each muscle and a rough idea of the
order of magnitude of the gains, we would have to allow for a way larger problem
space and the results of the first generation will very likely not be that accurate.
The second tested setting equals the first, apart from the reference angle, which
was set to 0.4. The results are given in figure 5.8 and 5.9. Basically the features of

![Figure 5.8: Optimization of the PD-Controller; $\alpha_{ref} = -0.4$, Muscle: Iliacus](image)

![Figure 5.9: Optimization of the PD-Controller; $\alpha_{ref} = -0.4$, Muscles: Iliacus, Gluteus maximus](image)

these graphs are the same, but there is one important difference: The fitness of the
most fit individual is better in the case of $\alpha_{ref} = -0.4$ for both, the one-muscle and
the two-muscle setting. Taking into account the small magnitude of the values it
can be considered remarkably better, also because this result is reproducible. The
question arising is, why can a smaller deflection of the leg simpler, or more stable,
be achieved? Indeed, one can observe that the fitness gets worse, the higher the
deflection is set. Manual probing showed that non-linear increasing of the propor-
tional gain is necessary to reach higher angles. The controller theory tells us that
we are lacking an integral-term, which is part of a so called PID-Controller (pro-
portional, integral, derivative). It should be considered in future work.

### 5.2.1 Manual parameter tuning

Similarly to the EA, the effort, respectively the time, to reach reasonable results
depends on the problem space. The PD-Controller set-up, especially the one-muscle
setting, describes a fairly simple task. As discussed, the optimization process al-
dready showed very good results after the simulation of the first generation. The
effort, to achieve equally good results manually will be described qualitatively in
the following, since data measurements of this process are not very revealing. It
turned out that for the one-muscle setting about 10-20 attempts were sufficient to
find gains that yield a good result. It also showed up, that the fitness values the
most fit individuals reached were nearly impossible to beat, although we are talking
about differences in the order of magnitude of $10^{-3}$. Once the reference angle is set
to rather high values, with respect to the deflection of the leg (e.g. -0.7 and smaller),
the manual tuning is clearly in favour, since fairly high values for the proportional
gain are necessary. Summing up, manual tuning makes sense to isolate the prob-
lem space and tune the EA with this information whereas the actual optimization
process can safely be performed by the plug-in. Of course, this has to be balanced
on every problem separately. As mentioned in the introduction the first approach
was the SMT-Controller, aiming to carry out reflex learning. This controller and
the Hopping-Controller, that was designed to apply activation on the muscles of the model according to the learned weights, will be discussed in the next chapter.
Chapter 6

Additional Results

In this chapter the design, the modelling and the implementation of two additional controllers will be described. Starting from a motivation, illustrating their relevance, the behaviour of the controllers is analysed. Finally, in the case of the SMT-Controller, the data resulting from its application on the leg-model is evaluated.

6.1 The SMT-Controller

Based on the work of Bharadwaj, the aim of the SMT-Controller is to carry out reflex learning by twitching the muscles of the model, determining a set of reflex circuits. It is designed, such that it is not bound to a certain model. However, this work only describes the leg model as proposed in section 2.1.2. The model is fixed in space to learn the reflexes. Accordingly, although it is subject to gravity, it is not able to fall down. Figure 6.1 shows a simplified state machine of the SMT-Controller. Noticeable is that the muscle twitching is not part of the state machine at all, which is due to pragmatic reasons. Primarily, it is entirely timer controlled. The learning rule that is implemented in the controller is not part of this work and can be comprehended in Bharadwaj.

Although the controller is a little bit more complex, for example by making use of more than one timer, the diagram is good enough to explain the twitching process qualitatively:

The controller starts in state zero; once its step function is called for the first time, a timer is started that determines the time until a twitch is performed and the state changes to state one. The controller then remains in this state and recalculates the weights in every single step. Simultaneously the first twitch occurs when the timer has exceeded. Now the twitch lasts for a certain duration and once it is finished the state is set to state two if the number of desired twitches is not reached yet, or remains in state one otherwise. Although it seems unnatural that the controller should stay in state one, this is justified by the design of the controller and the optimization plug-in: The controller does not stop itself, instead the optimization plug-in using it does. It makes sure that the controller is not anymore stepped and resets it to its initial state using the reset function of the controller interface. Moreover there are three open questions concerning the explanation just given: Firstly, why does the controller wait a certain time between single twitches? The reason is that unintended movement of the leg affects the weight learning process. To retrieve clean data the leg needs to be in rest. Since it is flexible and subject to gravity it will never be in a perfect rest, but without the waiting period the leg keeps swinging and biases the data of every subsequent calculation. Secondly, what does it
mean that the twitch “lasts”? The Hill-muscle model that is used here, allows to set an activation on the muscle. In simple words it can be determined how strong the muscle is contracted. The duration fixes how long this activation is subject to the muscle. Finally, how many twitches should be performed? Experiments, as explained in the following section, have shown that three twitches for every muscle were sufficient, although the data suggests that two twitches could also be sufficient. Crucial for the choice of the number of twitches is that the weights must converge to a certain value. A more detailed description is provided in Bharadwaj.

All the parameters given in the description above - the number of twitches, the twitch strength (respectively the muscle activation), the time between the twitches and the duration of a single muscle twitch - can be tuned in the GUI. Using the SMT-Controller one will also find an input for the so called “learning rate” \( \eta \) and an input for the matrix update interval. The former refers to a number used in the learning rule, the latter determines how often the matrices in the GUI are updated, it does not affect the process itself. The choice of all these parameters depends on the given model and is consequently left to the researcher.

6.1.1 Resulting matrices

As described in Bharadwaj, examining the learning rule, certain features of the resulting weights can be expected, if applied properly. A Hinton diagram, which is a diagram that illustrates the correlation between the magnitude and the sign of a matrix of values rather than explicit numbers, can be used to verify the correlation of the weights. Figure 6.2 shows a Hinton diagram for one type of weights that was obtained using the SMT-Controller. Basically it visualizes qualitatively how the contraction of a muscle affects features of other muscles. Again, neither the formula to calculate these weights nor the correlations shown by the diagram will be described here. The important point is, that using this type of diagram, flawed correlations in the muscle setting or even a wrong calculation can be revealed. In
the case of the leg-model it turned out that the foot causes problems. This means that the weights associated to the muscles, attached at the foot, are seemingly wrong. The reason for this is, that the foot has a way shorter degree of freedom and might therefore behave different compared to other parts of the model during the twitching process. These issues have to be verified and solved to continue the work with the SMT-Controller. The consequence due to this problem was to stop investigations at this point.

After obtaining the matrices, the GUI offers the user the possibility to store the matrices on the file system, so that they can be used by the Hopping-Controller, which will be investigated in the next part.

6.2 The Hopping-Controller

The Hopping-Controller can be considered as the second part of the task “Stable Hopping”. Once a set of reflex connectivities was obtained and stored on the file system it can be used by the Hopping-Controller. The Hopping-Controller does not implement a state machine, since its behaviour does not change on any trigger. In contrast to the SMT-Controller, the Hopping-Controller uses, of course, a model that is not fixed in space. Once the Hopping-Controller is loaded, a new environment is loaded where the hip is not fixed. If the simulation starts, the leg simply falls down due to the pull of gravity.

Basically the Hopping-Controller works as follows: Using a set of weight matrices obtained from the SMT-Controller, the activation of every single muscle in the model is calculated in every single step. This is principally everything what the controller does. The formula that determines the activation of the muscles is

\[ A_i = G_{II} \sum_j w_{ij}^I (l_j - l_{rest,j}) + G_{Ia} \sum_j w_{ij}^I \dot{l}_j, \]  

(adapted from [1]) where \( A_i \) is the activation of muscle \( i \), \( w_{ij}^I \) and \( w_{ij}^I \) correspond to the values of the weight matrices, \( l_j \) corresponds to the muscle length, \( l_{rest,j} \) to its resting length, \( \dot{l}_j \) to the rate of change in length and \( G_{II} \) and \( G_{Ia} \) are so called gains. Important here are the two gains. In previous work it was shown, that simply applying the activation formula using the weights is not sufficient to realize, for example, stable hopping (see e.g. [1]). The weights also have to be tuned by two parameters, that scale them in a proper order of magnitude. Correspondingly, \( G_{II} \) and

\[ G_{Ia} \]
and $G^{Ia}$ have to be chosen properly so that the formula yields to activations that are settled in a magnitude that fits the environment. This defines an optimization problem the optimization plug-in can be applied to. The two gains have to be tuned adequately to achieve stable hopping. Thus, how does the controller have to be designed to meet the requirements given in section 3.1.2? The optimizable and the reference parameters simply have to be designated as such. The former is quite easy, the only parameters in this scenario that can be tuned to change the outcome are $G^{II}$ and $G^{Ia}$. Implementing the controller they have to be designated as optimizable parameters and will appear in the according user interface of the optimization plug-in. Declaring the reference parameters needs a little bit more of thought: Since they have to be combined with a proper fitness function they should reveal useful information about the model. Bearing in mind that the overall goal is stable hopping, the height of the model in space seems to be a good choice. A sinus curved development of the model’s height in time with a more or less constant amplitude could be a first idea to determine the fitness. Mathematically spoken, the standard deviation of the model’s height has to be minimized.

If the order of magnitude, in which the parameters that will be optimized are located, is not known at all, probing different values can be used as a strategy to scale down the problem space. In this case the magnitude of the gains can be guessed quite properly comparing the gains and the weights that led to stable hopping in Bharadwaj. Even without using the optimization plug-in, several tests, tuning the gains manually, did not lead to a hopping like behaviour. Although this does not necessarily mean that no two gains can be found that lead to this behaviour, there are two problems that make it very unlikely to find gains that are fitting. Firstly, the aforementioned problem in terms of the weight matrices has to be addressed. Even if only the foot causes problems, it is possible that this disturbs the entire process and produces useless matrices. Secondly, by now the ground model does not work properly or is at least not adequately tested. That might exclude the possibility for hopping independently from the obtained reflex connectivities and accordingly makes further investigations on the SMT-Controller dispensable until the ground model is verified.

The last chapter concludes the results of this work and presents possible tasks that can be carried out in future projects.
Chapter 7

Conclusion and future work

This chapter summarizes the achievements of the different parts of the project, emphasizing the main goals and possible work and research that could be carried out subsequently.

7.1 The leg model

The leg model in combination with the Hill muscle model was chosen to be able to compare the results of the SMT-Controller with those accomplished in Bharadwaj. Although this opened the possibility for a lack of clear criteria to compare the results, it was decided to use a foot and three additional muscles in this work. As it turned out the foot, or rather the muscles attached to it, caused problems in the weight learning process and might be a reason that the task “stable hopping” could not be achieved. Further work that makes use of the leg model, should deal with this problem or even start with a model that is identical to the one proposed in Bharadwaj. Another argument that suggests this course of actions is to make sure that there exist no other problems in the weight learning process, like, for example, the problem with the ground model as restated in section 7.3.

7.2 The optimization plug-in

The main goals of the optimization plug-in were modularity, such that it can easily be extended, and generality, such that it is not restricted to certain problems. Both of these requirements, as it was described in detail, were accounted for since the first draft of the architecture. The former, by designing the software model accordingly, the latter by considering this requirement in the software model as well as making use of the EA approach. A certain application of this modularity is the process of adding new fitness functions. Different problems will often require a different fitness function. Although only two steps have to be taken to add a new one, both necessitate changes of the code. To enlarge the set of problems the plug-in can be used for, a next step could be an implementation of a GUI based fitness function definition. This definitely requires a very sophisticated user interface and constitutes a large project on its own. Moreover, the plug-in only offers mutation as a Genitor. It should definitely be extended by the Genitor crossover, which gives the plug-in a strong tool to adjust it to several problems. Also, there is an extensive range of literature and research on the topic of EAs and similar classes of algorithms. These domains should be examined and evaluated due to their relevance in the Myorobotics context. Afterwards the optimization plug-in should be extended by techniques that are in line with the domain of tendon-driven robots.
Independent of the possible extensions, the performance of the optimization plug-in was shown using the PD-Controller. In future work this has to be verified further, applying the plug-in to more sophisticated problems.

### 7.3 The controllers

The first idea was to implement the optimization plug-in and test it, using a SMT- and a Hopping-Controller. Although the SMT-Controller is fully implemented, the correctness of the values is not ensured. As described in section 7.1 this might be caused by the additional foot, leading to odd values in the weight matrices. Apart from that, the SMT-Controller offers all the functionality that it is designed for. The parameters, associated to the weight learning process, can be tuned in the GUI and the obtained matrices stored to the hard drive to use them in the Hopping-Controller. In future work it should be extended in such a way that the learning process can be applied to an arbitrary model, allowing to simulate and optimize other tasks. One example of a different task is stable hopping with an additional torso. It is already subject to research in the BIRL, using the Matlab environment. The Hopping-Controller is the simplest of the three controllers, only applying the activation to the muscles according to equation 6.2. Loading the necessary weight matrices as well as setting the two gains $G^{Ia}$ and $G^{II}$ can be done in the GUI. Once the controller is loaded in Caliper, the environment is automatically changed, such that the leg model is no longer fixed in space. This mechanism is tailored to this task and should be generalized to different environment settings. As pointed out, the leg does not show hopping like behaviour which is, apart from the aforementioned problem with the foot, most likely due to an insufficient ground model. It has to be verified before further investigations can be performed.

The third controller that was implemented is the PD-Controller. It was also designed with a focus on generality, that is, an arbitrary model can be used. All the muscles of the model will be offered as parameters that can be self controlled using the PD-Controller whereas all the hinges will be offered as reference parameters. The controller offers many opportunities for further investigation, for example using different muscle settings, reference parameters and values as well as other models and environments. A first insight on the behaviour of different muscle settings in the Caliper environment was already given in this work.
Chapter 7. Conclusion and future work
Appendix A

Symbols, Indices, Acronyms and Abbreviations

Symbols

\begin{align*}
A_i & \quad \text{activation of muscle } i \\
w_{ij}^{II} & \quad \text{weight of group } s^{II}, \text{ see [1]} \\
w_{ij}^{Ia} & \quad \text{weight of group } s^{Ia}, \text{ see [1]} \\
l_j & \quad \text{length of muscle } j \\
l_{rest,j} & \quad \text{resting length of muscle } j \\
l_j & \quad \text{rate of change in length of muscle } j \\
G^{II} & \quad \text{the gain, tuning the weights belonging to group } s^{II} \\
G^{Ia} & \quad \text{the gain, tuning the weights belonging to group } s^{Ia} \\
\alpha_{ref} & \quad \text{reference angle} \\
e(t) & \quad \text{error at time } t \\
\alpha(t) & \quad \text{actual value of the angle at time } t \\
e'(t) & \quad \text{derivative of the error} \\
\Delta t & \quad \text{time step of the simulator} \\
A_i(t) & \quad \text{activation of muscle } i \text{ at time } t \\
P_i & \quad \text{proportional gain of the muscle } i \\
D_i & \quad \text{derivative gain of the muscle } i \\
F & \quad \text{fitness function} \\
N & \quad \text{total number of discrete measurements} \\
\eta & \quad \text{learning rate}
\end{align*}

Indices

\begin{align*}
x & \quad \text{x-axis} \\
y & \quad \text{y-axis}
\end{align*}
### Acronyms and Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
</tr>
<tr>
<td>BIRL</td>
<td>Bio-Inspired Robotics Laboratory</td>
</tr>
<tr>
<td>COLLADA</td>
<td>Collaborative Design Activity</td>
</tr>
<tr>
<td>EA</td>
<td>Evolutionary algorithm</td>
</tr>
<tr>
<td>ECCEROBOT</td>
<td>Embodied Cognition In A Compliantly Engineered Robot</td>
</tr>
<tr>
<td>ETH</td>
<td>Eidgenössische Technische Hochschule</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical user interface</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>IEEE-SA</td>
<td>Institute of Electrical and Electronics Engineers Standards Association</td>
</tr>
<tr>
<td>Myode</td>
<td>Myorobotics Development Environment</td>
</tr>
<tr>
<td>PD</td>
<td>Proportional Derivative</td>
</tr>
<tr>
<td>SE</td>
<td>Software Engineering</td>
</tr>
<tr>
<td>SMT</td>
<td>Single muscle twitch</td>
</tr>
<tr>
<td>SVN</td>
<td>Apache Subversion</td>
</tr>
<tr>
<td>TUM</td>
<td>Technische Universität München</td>
</tr>
<tr>
<td>UB</td>
<td>University of Bristol</td>
</tr>
<tr>
<td>UWE</td>
<td>University of West of England</td>
</tr>
</tbody>
</table>
Appendix B

Screenshots from Caliper

Figure B.1: Evolution component of the optimization plug-in, tab “Parameters” - screenshot from Caliper.
Appendix B. Screenshots from Caliper

Figure B.2: Tab “Optimization” of the optimization plug-in - screenshot from Caliper.

Figure B.3: Evolution component of the optimization plug-in, tab “Fitness” - screenshot from Caliper.
Bibliography


Retrieved July 1, 2013


UbuntuDesktop


Statement of authorship

I declare that I completed this thesis on my own and that information which has been directly or indirectly taken from other sources has been noted as such. Neither this nor a similar work has been presented to an examination committee.

Zürich, July 4, 2013

........................