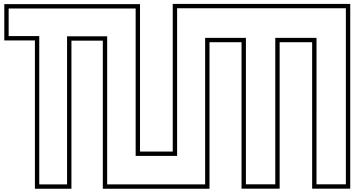


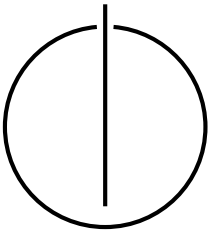
FAKULTÄT FÜR INFORMATIK

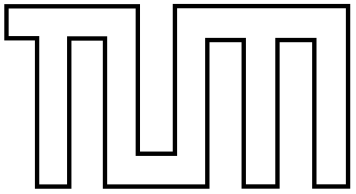
der Technischen Universität München

Master's Thesis in Informatik

Extracting the Muscle Jacobian for Anthropomimetic Robot Control using Machine Learning

Christian Schmaler





FAKULTÄT FÜR INFORMATIK

der Technischen Universität München

Master's Thesis in Informatik

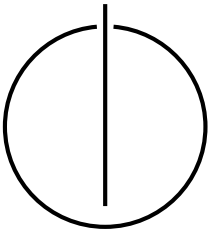
Extracting the Muscle Jacobian for Anthropomimetic Robot Control using Machine Learning

—

Bestimmung der Muscle Jacobian zur Steuerung eines Anthropomimetischen Roboters mittels Maschinellen Lernens

Christian Schmaler

Supervisor:	Prof. Dr.-Ing. habil. Alois Knoll
Advisor:	Dipl.-Ing. Michael Jäntsch
Submission Date:	March 31, 2011



Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I assure the single handed composition of this thesis only supported by declared resources.

München, den 31. März 2011 Christian Schmaler

Abstract

The anthropomimetic principle introduced a new design paradigm in humanoid robotics as it does not just imitate the human form, but also the biological structures and functions. By mimicking the mechanisms of the musculoskeletal system—like bones, joints, and muscles—the structure of the robot becomes more compliant and allows a safer operation in a human centered environment. However, the control of such a robot is still a relatively unexplored problem. It requires a model which describes the functional relationship between joints and muscles, e.g. in form of the so-called muscle jacobian [41]. The derivation of such a relationship for humanoid robots with many DoF can be difficult. This work proposes a general method for the extraction of the muscle jacobian. It uses machine learning techniques for function approximation and is therefore in principle applicable for arbitrary tendon-driven anthropomimetic robotic systems. For the generation of training data a bio-inspired method based on random limb movements is suggested among others. The developed extraction method allows a partitioning of robotic systems in independent subsystems and can therefore be easily extended and scaled to robots with many DoF. It has been evaluated using a simplified simulation of an anthropomimetic robot arm developed in the ECCEROBOT project. It consists of a hinge elbow joint as well as a spherical shoulder joint and is actuated by 11 muscles. As machine learning methods, feedforward neural networks (FNN) and locally weighted projection regression (LWPR) were examined.

Kurzfassung

Mit dem anthropomimetischen Prinzip wurde ein neues Paradigma in der Konstruktion von humanoiden Robotern geschaffen, das nicht nur die menschliche Gestalt sondern auch die biologischen Strukturen und Funktionen imitiert. Durch das Nachahmen des muskuloskelettales Systems (wie Knochen, Gelenke und Muskeln) wird die Struktur des Roboters nachgiebiger und erlaubt somit einen sichereren Betrieb im menschlichen Umfeld. Die Regelung eines solchen Roboters ist allerdings nach wie vor eine recht unerforschte Problemstellung. Diese erfordert ein Modell, das den funktionalen Zusammenhang zwischen Gelenken und Muskeln beschreibt, etwa in Form der sog. Muscle Jacobian [41]. Für humanoide Roboter mit vielen Freiheitsgraden kann die Bestimmung dieses Zusammenhangs allerdings problematisch sein. In dieser Arbeit wird ein allgemeines Verfahren zur Bestimmung der Muscle Jacobian vorgestellt. Dieses verwendet maschinelle Lernverfahren zur Funktionsapproximation und ist daher im Prinzip für beliebige, durch Sehnen bewegte anthropomimetische Roboter anwendbar. Zur Erzeugnung von Trainingsdaten wird u.a. eine biologisch inspirierte Methode auf Grundlage von zufälligen Bewegungen von Gliedmaßen vorgeschlagen. Das entwickelte Bestimmungsverfahren erlaubt die Zerlegung von Robotern in voneinander unabhängige Teilsysteme und kann daher einfach erweitert und auf Roboter mit vielen Freiheitsgraden skaliert werden. Es wurde mit einer vereinfachten Simulation eines im ECCEROBOT Projekt entwickelten anthropomimentischen Roboterarms getestet. Dieser besteht aus einem Drehgelenk im Ellenbogen, einem Kugelgelenk in der Schulter und wird von 11 Muskeln bewegt. Als maschinelle Lernverfahren wurde die Verwendung von vorwärtsgerichteten neuronale Netzen (FNN) und Locally Weighted Projection Regression (LWPR) untersucht.

Acknowledgements

First of all, I would like to thank my supervisor Professor Alois Knoll, Dr.-Ing. habil, head of the chair for Robotics and Embedded Systems at Technische Universität München, for giving me the opportunity to work on such an interesting topic.

Most importantly, I wish to express my deep and sincere gratitude to my advisor Michael Jäntsch, Dipl.-Ing., for his patience and continuous friendly support. His understanding and encouraging guidance have been an important basis for the present thesis and enabled me to develop an understanding of the subject.

I equally would like to extend my thanks to Steffen Wittmeier, Dipl.-Inf. (FH), for his assistance and the welcoming cooperation. His valuable advice constantly contributed to my work and he was a great help in issues of programming.

My sincere thanks also goes to Frank Sehnke, Dipl.-Inf., Christian Osendorfer, Dipl.-Inf., and Thomas Rückstieß, Dipl.-Inf. They kindly supported me in all questions related to machine learning.

I am grateful to Konstantinos Dalamagkidis, Dr., for providing insightful comments.

Furthermore, I thank all other bachelor, master and Ph.D. students at the branch of the chair for Robotics and Embedded Systems in Garching-Hochbrück for providing a pleasant environment and nice chats during the breaks.

Last but not least, I would like to thank my parents and my brother for their support and encouragement during all my studies.

Contents

List of Figures

List of Tables

1. Introduction

1.1 Motivation

Most current humanoid robots try to solely imitate the outer human form, while under the surface still conventional robotics mechanisms are used. For instance, they are usually constructed with rather simple joints of a single degree of freedom and are actuated by motors positioned directly in the joints. Needless to say, these mechanisms are radically different from the human structure and lead to some drawbacks. For one thing, the structure of these robots results in movements which seem very unnatural. For another thing, a safe interaction with humans becomes problematic due to the stiff and unyielding structure and the high forces they can exert. But especially in the case of humanoid robots it is important that they are able to work flexibly and safely in a human centered environment. One approach to circumvent these problems was introduced by the anthropomimetic principle [8]. This new design paradigm for humanoid robots also mimics the inner structures and mechanisms of the human body, not only its outer shape. In a musculoskeletal humanoid the anatomical prinicple of bones, joints, muscles and tendons is duplicated in an abstract form.

The Embodied Cognition In A Compliantly Engineered Robot (ECCEROBOT) project, in whose scope this work is set, has developed several prototypes of anthropomimetic robots [16]. One of its long-term objectives is to create a robot torso which moves and interacts with the physical world similarly to the way we humans do. The much higher level of biological inspiration might help in the understanding of human cognition and action as the structural limitations of standard humanoid robots affect their ability to perceive and internalize the world around them [7]. The picture of the prototype ECCE3 in Fig.1.2 shows that the construction is very similar to the human anatomy. The bones are made by hand from a thermoplastic known as polymorph which can be molded at 60° C. Not only the skeleton is close to the human model, the actuation methods have been imitated as well. The robot is equipped with artificial muscles consisting of a kiteline that is wound around a spindle driven by a DC motor and gearbox (see Fig. 1.1a). To mimic the flexibility of a biological muscle the kiteline is connected to a piece of shock cord. Depending on the direction of motor rotation, the artificial muscle can be innervated or relaxed. The joints of an anthropomimetic robot can be much more complex than in conventional humanoids as they are not limited to simple hinge or universal joints but can also be spherical with multiple degrees of freedom. Limbs have to be moved through the antagonist-protagonist-principle in which the effect of one muscle is opposed by that

of one or several others (see Fig. 1.1b). The setup of the complete robot is explained in detail in [8].

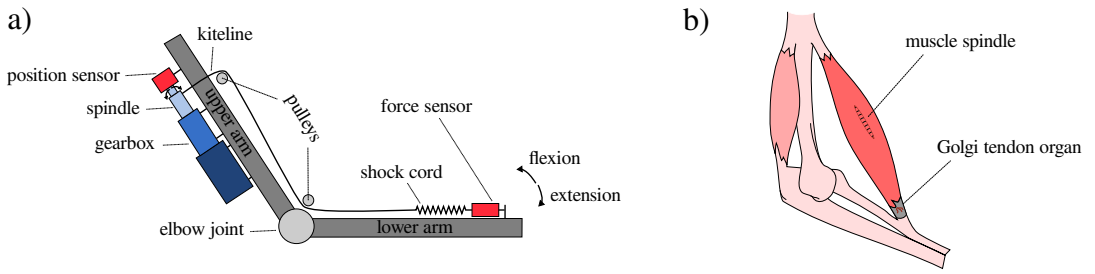


Figure 1.1: (a) The actuation principle of an anthropomimemtic robot with artificial muscles. By winding the kiteline round a spindle driven by a DC motor and gearbox a force can be exerted on the bones of the robot. The elasticity of a biological muscle is imitated by a piece of shock cord. (b) The corresponding human structure shows the principle of antagonistic muscles. The muscles of mammals contain sensor systems for measuring muscle tension (via the Golgi tendon organ) and the length of a muscle (via the muscle spindle). (Graphics adapted from [12].)

Due to the compliant muscular linkages in the skeletal structure all movements and disturbances are transmitted through the whole robot body. Anthropomimetic robots have shown that this flexible structure leads to quite human-like movements which are also much safer in the interaction with humans. The drawback of this concept is, however, that it is much harder to control since most of the control methods already extensively examined for standard robots are not directly applicable. Even simple movements like lifting an arm requires the actuation of several muscles to retain the body posture. In the ECCEROBOT project it was tried to copy the structure of the human motor control system and a distributed control infrastructure was developed that reduces the complexity of the control task by distributing subtasks into the robot's limbs [12]. This way it is possible to specify certain properties of each muscle, like the force it exerts, from a global control level while the compliance of these reference values is controlled locally via the motor current of the muscle.

However, if the robot is to be driven in a given posture it is not sufficient to be able to control each muscle individually. It is also necessary to have some information about the joint angles and how the muscles contribute to them. A mapping between the muscles and the skeleton describes the relationship between the lengths of all muscles and the angles of all joints at a certain configuration of the robot. Chapter 2 presents two forms of such a mapping that enable different variants of whole body control for an anthropomimetic robot. One is the direct mapping from skeletal angles to muscles lengths, the other uses derivatives in the so-called muscle jacobian [41]. In case of ECCEROBOT, the determination of this relationship is a difficult task. Because of the robot's complex structure an analytic derivation seems almost impossible. Therefore, this master's thesis pursues the approach to extract this mapping in form of the muscle jacobian using machine learning techniques.

1.2 Related Works

The usage of machine learning techniques in the domain of robotics has a long tradition. Learning approaches have been applied on all levels in robot control in a vast number of works. It is not possible to enumerate all so far examined applications, but [14] and [21]

Figure 1.2: ECCE3, the third prototype of an anthropomimetic robot in the ECCEROBOT project. It consists of a torso, head and two arms and is actuated by 60 artificial muscles.

can provide a first rough overview of machine learning usages in robotics. The area that is most resemblant to our application is the learning of models for control, like learning inverse kinematics. This has been successfully tried many times for conventional robots, e.g. using neural networks in [22]. The authors of the LWPR algorithm, which is also used in this work, have learned an inverse kinematic mapping for a 30 degree of freedom humanoid robot [5].

As only very few anthropomimetic robots exist up to now, there have not been many opportunities for the usage of machine learning techniques with theses kinds of robots. Apart from the ECCEROBOT project there are most notably the musculoskeletal robots Kotaro [19] and its successor Kojiro [18]. The latter is used in [20] to develop joint proprioception for a tendon-driven robot, which is roughly similar to the task of this work. The paper describes a method that allows the estimation of joint postures based only on information about relative changes in muscle lengths. However, their algorithm requires that the mapping from joint angles to absolute muscle lengths is known. For Kojiro this mapping can be retrieved from a geometric model of the robot which describes the position of all joints, the sizes of the bones and the position of the muscle attachment points. Unfortunately, the structure of ECCEROBOT is much more complicated and such a model does not exist. Therefore, this mapping has to be extracted in another fashion.

1.3 Overview of Thesis

This thesis applies machine learning techniques in order to extract first the relationship between joint angles and muscle lengths and second the muscle jacobian for an anthropomimetic robot.

First of all, Chapter 2 outlines the characteristics of the task and motivates the use of machine learning to solve it. The muscle jacobian is introduced and a control approach is described for which it can be used. Afterwards, the platform for the experiments in this work is described. In Chapter 3 the used machine learning techniques are presented. It gives a short summary of supervised learning in general and briefly reviews feedforward neural networks (FNN) and locally weighted projection regression (LWPR). Subsequently, Chapter 4 is concerned with some general considerations about the solution of the task and the learning process. It also describes the used methods for the acquisition of training and testing data. Chapter 5 and 6 then thoroughly present the results achieved with this approach for a simulated anthropomimetic robot arm. At first, the used methods for the evaluation of performance are described. Then the results for a simplified version of the arm are shown which consists only of an elbow joint and three muscles. Eventually, the results for the complete robot arm are described. Concluding remarks and an outlook to future work are given in Chapter 7. Finally, Appendix A documents the software component developed in the course of this work and Appendix B derives the relationship of time derivatives of XYZ-Euler-Angles and rotational velocities.

2. Task

As already mentioned, the current possibilities to control the robot are still very limited. For each artificial muscle the motor voltage, motor position and exerted force can be specified. The respective control variables are adjusted to these reference values via PID-controllers using the sensory feedback. Each artificial muscle is equipped with several sensors that measure position and current of the motor and the tendon strain. The length of a muscle is composed of the lengths of its kiteline and its shock cord. It is possible to indirectly calculate the length of the kiteline via spindle radius and spindle revolution as measured by the motor position sensor. The length of the shock cord can be obtained using the measured force and the knowledge about the expansion characteristics of its material.

For controlled movements and interactions with the environment it is essential for the robot to know at all times how its limbs and other parts of the body are oriented in relation to each other. This sense of perceiving the posture of the joints is called joint proprioception. However, the robot is not equipped with joint angle sensors. Especially in the case of more complicated joints, such as spherical joints in the shoulder, it is difficult to embed appropriate sensors into the joint structure. It should also be mentioned that the human body is equally not equipped with sensors to measure joint angles directly. The joint proprioception is rather done by integrating information about the body motion, like muscle tension and muscle length. This information is provided by embedded sensors in the human muscle. The neurotendinous spindle measures changes in the muscle length and the Golgi tendon organ is sensitive to the muscle tension (see Fig. 1.1). As this information is also available in our robot we can try to mimic the human proprioception and obtain the angles of a joint from the lengths of all muscles that are spanned over it.

There is a function $f : \mathbb{R}^n \to \mathbb{R}^m_+$ defined by $f(\boldsymbol{q}) = \boldsymbol{l}$ which describes the relationship between all n joint variables $\boldsymbol{q}$ and the lengths of all m muscles $\boldsymbol{l}$. This function depends on the setup of the robot, like the attachment points of the muscles and the pathway of the tendons (in our case the kitelines). In the simplest case the length of a given muscle depends on only one joint. Aside from that, muscles may be also spanned over more than a single joint and therefore depend on several joint angles. The output of function f is defined for all joint configurations $\mathfrak{Q} \subset \mathbb{R}^n$ that the robot can take within the physical limits of its joints. A vector of muscle lengths can not contain arbitrary values as the lengths of the muscles are restricted by the skeleton and the arrangement of the muscles. In general, a certain muscles introduces some constraints on the other muscles spanned over the same joint. Moreover, the mapping from joints to muscles is only unique under

the condition that there are no slack muscles. If a joint position is supported by only some of the muscles spanned over it one of the other muscles can become slack without changing the joint angles. The arbitrary length of such a slack muscle would associate infinitely many vectors $\boldsymbol{l}$ to one joint configuration $\boldsymbol{q}$. Considering only the valid muscle length vectors in $\mathfrak{L} \subset \mathbb{R}_{+}^{m}$, the mapping from $\mathfrak{Q}$ to $\mathfrak{L}$ is bijective and therefore invertible.

This mapping can be used for whole body control. One way to do so is to use it directly to obtain the muscle lengths corresponding to a desired configuration of the robot. Provided that there is some kind of muscle length controller to deploy these reference values, this is a very simple method to move the robot in a given posture. Such a muscle length controller can be implemented as common feedback controller (e.g. PID controller), similarly to the motor position or force controllers already integrated for each muscle.

Another possibility to use this mapping for whole body control is its application in *computed-torque control* [30]. This is a standard robot motion control scheme that transforms nonlinear robotic systems into simple decoupled linear closed-loop systems by applying feedback linearization. The control design for the resulting systems is a well-known problem. The application for anthropomimetic robots requires some modifications that were developed by Jäntsch et al. and are described in detail in [11]. As the realization of this controller scheme was one of the main motivations for this work it is briefly presented here.

Computed-torque control is based on a dynamic model of the robot as described by the equation of motion in its joint-space formulation [30]:

$$\boldsymbol{\tau} = \boldsymbol{H}(\boldsymbol{q})\,\ddot{\boldsymbol{q}} + \boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})\,\dot{\boldsymbol{q}} + \boldsymbol{\tau}_g(\boldsymbol{q}) \tag{2.1}$$

For a robot with n links this yields a set of n decoupled linear systems, relating the vector of joint variables $\boldsymbol{q}$ and the corresponding torques $\boldsymbol{\tau}$. Using Eq. (2.1) it is possible to calculate the joint torques required to move the robot with a given joint acceleration $\ddot{\boldsymbol{q}}$ starting from a specific system state $\boldsymbol{q}$ and $\dot{\boldsymbol{q}}$. Since the actuation of an anthropomimetic robot is not achieved over motors positioned in the joints, it is not possible to control the torques directly. However, this information can still be used in an indirect way. As described in [41] the joint torques $\boldsymbol{\tau}$ can also be calculated over the vector of muscle forces $\boldsymbol{f}_m$:

$$\boldsymbol{\tau} = \boldsymbol{J}_m^{\mathsf{T}}\,\boldsymbol{f}_m \tag{2.2}$$

Matrix $\boldsymbol{J}_m$ is called *muscle jacobian* and indicates how the lengths of the muscles change for small joint angle changes. It is therefore similar to the chain jacobian widely used in robotics which specifies how the position of an endpoint changes in relation to small joint angle changes. The muscle jacobian is defined as the partial derivatives of f:

$$\boldsymbol{J}_m = \begin{bmatrix} \frac{\partial l_1}{\partial q_1} & \frac{\partial l_1}{\partial q_2} & \dots & \frac{\partial l_1}{\partial q_n} \\ \frac{\partial l_1}{\partial q_1} & \frac{\partial l_1}{\partial q_2} & \dots & \frac{\partial l_1}{\partial q_n} \\ \dots & \dots & \dots & \dots \\ \frac{\partial l_m}{\partial q_1} & \frac{\partial l_m}{\partial q_m} & \dots & \frac{\partial l_1}{\partial q_n} \end{bmatrix} \tag{2.3}$$

By solving Eq. (2.2) for $\boldsymbol{f}_m$ it is possible to obtain the muscle forces for the joint torques as calculated with Eq. (2.1). As there are generally more muscles than degrees of freedom, the inversion of Eq. (2.2) is underdetermined and has to be formulated as an optimization problem. An additional constraint of this optimization is to keep the muscle forces

as small as possible. The retrieved forces can eventually be used in the control of the anthropomimetic robot.

2.1 The Extraction of the Muscle Jacobian

The subject of this thesis is the extraction of the muscle jacobian for the anthropomimetic robots of the ECCEROBOT project. The great number of muscles and joints in the considered platform is not the only reason why this is not a trivial task. In principle, it is possible to obtain the mapping between joint angles and muscle lengths analytically from a geometric representation of the robot. However, the complex structure of the ECCEROBOT robots makes an analytic derivation extremely difficult. The kitelines of some muscles have complex pathways that collide with the skeleton in certain poses. Tendons influence and deflect each other depending on the posture of the robot. They may even exert forces on bones and joints and therefore change the angles of the joints. Furthermore, the robots are completely built by hand, so there are no exact CAD models of them and each one is different. It is therefore not possible to get the necessary information out of a blueprint.

For these reasons, the approach in this work is to use machine learning techniques to extract an approximation of the muscle jacobian. This has the advantage that once the method has been developed and tested it can be applied to any anthropomimetic robot. In addition, it is assumed that the bones made of a thermoplastic may slightly distort over time. This could change the relationship between joint angles and muscle lengths. Using a suitable learning method, the retrieved approximation can be easily adapted to these changes without the need to start from scratch.

The training data to learn this approximation is retrieved by moving the robot and recording the joint angles and muscle lengths. In this work we will examine the application of two machine learning methods for this function approximation task, i.e. feedforward neural networks (FNN) and locally weighted projection regression (LWPR). These will be introduced in Chapter 3.

2.2 The Anthropomimetic Robot Arm

Needless to say, the ultimate goal is to apply this approach to an entire robot. But in order to reduce the complexity during development and evaluation, this work will use a reduced test rig that is located at Technische Universität München. This test rig is an anthropomimetic robot arm which uses the same mechanisms as the entire robot and can be seen in Figure 2.1. It consists of an elbow and a shoulder joint.

While the elbow is just a conventional hinge joint with a single DoF, the shoulder of the test rig is a slightly simplified replication of the human shoulder which is one of the most complex joints in our body. The shoulder unites the shoulder blade (scapula), the collarbone (clavicle) and the upper arm (humerus) and actually consists of three joints (the glenohumeral, acromioclavicular, and the sternoclavicular joint) [17]. The main joint (glenohumeral joint) is a ball and socket joint and provides the upper arm with all three degrees of freedom. The ball is part of the upper arm and rests rather loosely in the dish-shaped portion of the shoulder blade, held in place by the tendons and muscles spanned over it. As all these elements of the biological example have also been modelled in a simplified manner in the robotic arm it becomes even more difficult to embed angle sensors directly into the joint than it already is for standard spherical joints. The complete test rig is actuated by the 11 muscles listed together with their primary function for movements in Table 2.1. After all, 8 of these muscles are exclusively associated with the shoulder and

there is also a muscle—the Biceps Brachii muscle—that is spanned over both shoulder and elbow joints.

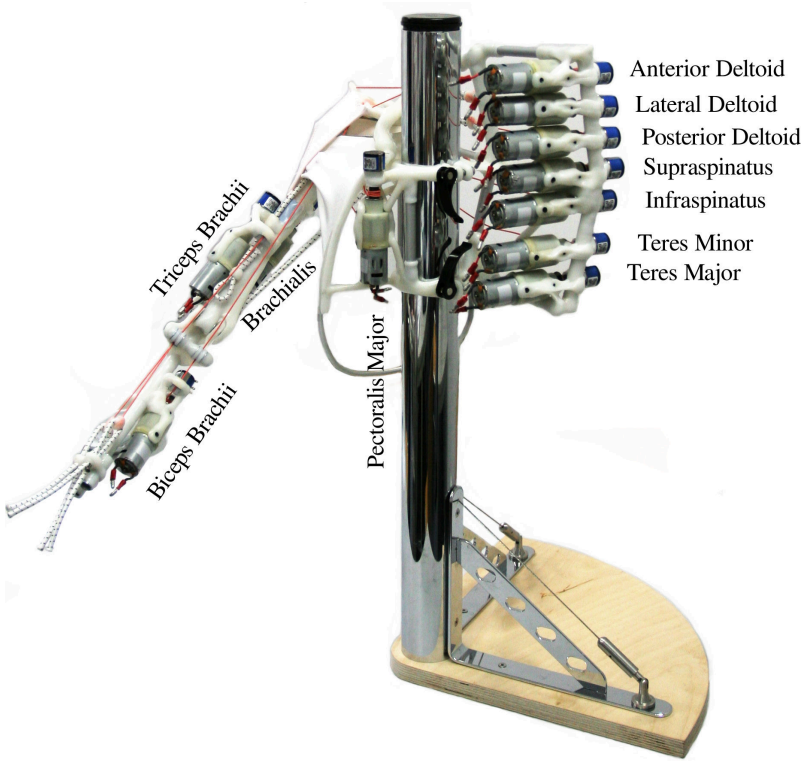


Figure 2.1: The test rig: An anthropomimetic robot arm consisting of elbow and shoulder joint with 11 muscles.

No	Joint	Muscle name	Main Function
1	S	Anterior Deltoid	Shoulder abduction when externally rotated
2	S	Lateral Deltoid	Shoulder abduction when internally rotated
3	S	Posterior Deltoid	Transverse extension and external rotation
4	S	Supraspinatus	Shoulder abduction and stabilization
5	S	Infraspinatus	External rotation, transv. abduction and stabilization
6	S	Teres Minor	External rotation and transverse abduction
7	S	Teres Major	Arm Extension, internal rotation and adduction
8	S	Pectoralis Major	Transverse adduction and internal rotation
9	S&E	Biceps Brachii	Elbow flexion and transverse shoulder flexion
10	E	Brachialis	Elbow flexion
11	E	Triceps Brachii	Elbow extension

Table 2.1: The 11 muscles of the test rig with their main function for movements. Most muscles have multiple functions depending on the posture of the arm. The muscles are divided in three groups subject to whether they contribute to shoulder movements (S), elbow movement (E) or both (S&E). (Descriptions taken from [26] and adapted to the test rig.)

The reduced number of DoF and muscles is without question a massive simplification compared to an entire robot. But it is a necessary step during implementation and testing. Since the test rig is still complex enough to study all requirements of this task the results retrieved on the test rig can hopefully be applied to the full robot in the end.

2.3 The Simulation

In this work only a model of the test rig is used in simulation for the time being. There are mainly two reasons for this. On the one hand, the development is simplified by means of the simulation. It allows an easier, faster and more accurate acquisition of the muscle lengths and joint angular values required as training and testing data. On the other hand, although it was mentioned before, the muscles of the real test rig are not fully equipped with force sensors yet as these are still in development. It is therefore not possible to calculate the length of a muscle based on the motor position and the force that stretches the shock cord. The second part necessary to record data for the real robot is the capability to measure all joint angles. For this purpose, a stereo vision based tracking system has been implemented and tested in another work over the last months and is almost ready to use.

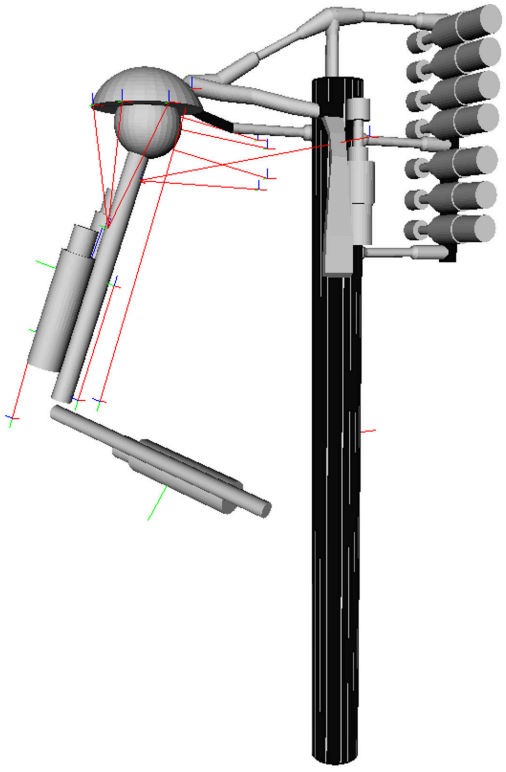


Figure 2.2: The model of the anthropomimetic robot arm in simulation. The muscles are indicated by red lines.

Prior to this work, the test rig was scanned with a 3D scanner in order to obtain a model for the simulation. Some simplifications had to be made afterwards to the scanned 3D model (Fig. 2.2). In addition to the geometrical model, a physical model contains estimates of all the relevant properties of the robot arm, like the masses and inertia of the limbs or the friction in the joints. Both were used in a physics simulation (see Appendix A.1) throughout this work.

Like the real robot, the model is actuated by 11 muscles which have roughly the same function as in the real test rig. However, it must be noted that the model is currently still a massive simplification. The attachment points of the muscles are only vague estimations and, most importantly, the tendons do neither interfere with the skeleton nor with each other. Muscles are simulated as straight connections between two attachment points. It has turned out that it is very hard to simulate the deflection of the kitelines physically correctly, especially in the case of colliding kitelines. It will be the objective of a future work to adjust the structure and behavior of the simulated model as close as possible to the real robot.

Since the CAD model is known and the muscles are represented in this simple fashion the muscle jacobian could easily be obtained analytically for this case. Nevertheless, this is still an arduous task for a higher number of muscles, especially for muscles spanned over more than just one joint. But what is more important, it is the task of this work to find a method applicable for the real robot after all.

3. Background

This chapter provides the theoretical background to the machine learning techniques relevant for this work. The first section is dedicated to the basic principle of supervised sequence learning in general, independently of any specific machine learning method. Subsequently, the methods and architectures considered in this thesis are introduced. Section 3.2 presents feedforward neural networks like multilayer perceptrons and artificial neural networks in general. Eventually, a further learning architecture for non-linear function approximation, locally weighted projection regression, is described in section 3.3

3.1 Supervised Learning

Machine learning tasks can usually be roughly divided in the categories of *supervised learning*, *unsupervised learning*, and *reinforcement learning*. The task in this thesis is a common supervised learning problem and this principle is therefore introduced in this section while the other two are only mentioned. For a more detailed introduction of the different machine learning tasks and various popular methods see [1]. Supervised learning refers to a class of problems in which a *teacher* dictates what the learning system is supposed to learn, the so-called *targets*. In contrast, in unsupervised learning there is no such specification and the learning algorithm seeks to determine the structure of the data on its own. In reinforcement learning merely a system of rewards and punishments is provided and the *agent* learns by trial and error to optimize its profit.

The purpose of supervised learning is to adapt a function from training data. The used training data is divided in two or three sets. In all cases there is a *training set* S and a disjoint *test set* T which both consist of pairs $(\boldsymbol{x}, \boldsymbol{t})$ of inputs and desired output targets, where $\boldsymbol{x}$ is an element of the input space $\mathfrak{X}$ and $\boldsymbol{t}$ is an element of the target space $\mathfrak{T}$. The dimensions of input and target space are arbitrary and determined by the specific task. Similarly to polynomial curve fitting, the pairs of the training set can be thought of as supporting points of the function that shall be learned. The whole purpose of supervised learning is to use the training set to train an approximator $h : \mathfrak{X} \to \mathfrak{T}$ in a way that minimizes the difference between the output of the function approximator and the real values when used on the test set. In order to achieve a good performance on the test set, the learner has to generalize the relationships in the presented training data and transfer them to unseen situations. In general, it is important that the data points of every set are independent and identically distributed (i.i.d.), which means they are drawn independently from the same distribution $D_{\mathfrak{X}\times\mathfrak{T}}$ of input-target pairs.

Dependent on whether the task is a regression or classification problem, there are different error measures to evaluate the approximation accuracy. As the task of this work is a regression problem, the case of classification will not be further considered. Typically, in regression tasks the real-valued N-dimensional input space $\mathfrak{X} = \mathbb{R}^N$ is mapped to the M-dimensional output space $\mathfrak{T} = \mathbb{R}^M$. In this case, a very popular error measure is the mean square error. For the output dimension k, the mean square error MSE_k with respect to all samples in a test set T is defined as:

$$MSE_k = \frac{1}{|T|} \sum_{i}^{|T|} (h(\boldsymbol{x}_i)_k - t_{i_k})^2 \tag{3.1}$$

where $|T|$ is the size of the test set, $h(\boldsymbol{x}_i)_k$ is the k-th element of the approximated output for input $\boldsymbol{x}_i$ and the t_{i_k} is the k-th element of the i-th target vector. To account for the scale of the outputs this value can be normalized to the variance of all target values in the n-th dimension. The normalized mean square error for output dimension n is defined as:

$$nMSE_k = \frac{MSE_k}{var_k} \tag{3.2}$$

Sometimes an additional third set, the *validation set* V, is split off from the training set. Its purpose is to validate the perfomance of the learning algorithm during the training process and decide when to stop the training in order to avoid *overfitting*. Overfitting could be described as "learning by heart" in the context of a human learner. The algorithm merely learns the presented input-target-pairs in the training set but does not abstract from the exemplary data points to the general relationship behind them. Usually, there is a point in the training process from which the performance on the validation set starts to get worse again while the performance on the training set is still increasing, often to an exact mapping of inputs to targets in the training set. In order to maximize the performance on an unseen test set this is a reasonable point to stop the training. This method is called *early stopping*. Figure 3.1 shows an example of this behaviour.

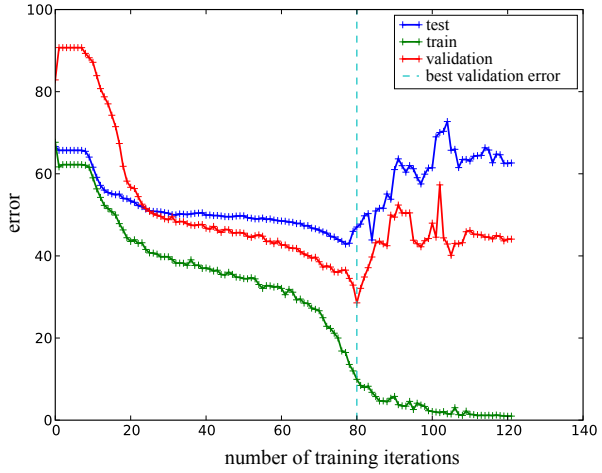


Figure 3.1: An example of early stopping in order to avoid overfitting during neural network training. For a good generalization property the model parameters of the marked training step are chosen.

3.2 Feedforward Neural Networks

An artificial neural network (ANN) is a connectionist model for processing information by—very abstractly—simulating the structure and aspects of a real biological neural network. It can be used to recognize and classify patterns and to model complex relationships in all kinds of data. This section will describe the basic principles behind neural networks and a popular training algorithm. A more detailed introduction can be found in [3].

An ANN consists of a number of processing units, the artificial neurons, which are linked by directed weighted connections with one or more other processing units. Figure 3.2 shows a juxtaposition of the schematic structure of a biological and an artificial neuron.

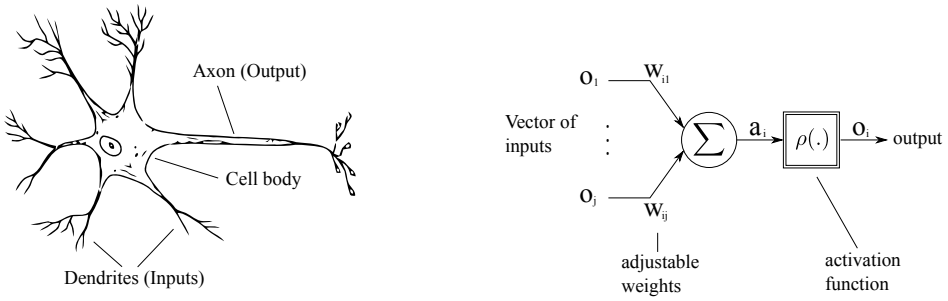


Figure 3.2: A juxtaposition of the schematic structure of a biological and an artificial neuron. (Figure of the biological neuron adapted from [13].)

Like its biological example the artificial neuron displays an activity (in most cases a real value o) which depends on the activity of the neurons connected to it. The connections between these units can have an amplifying or inhibiting effect on the activity depending on their connection weight. A weight is simply a numerical value associated with a connection that scales the input on this connection and represents the strength of a synapse between two biological neurons. The sign of the weight indicates amplifying or inhibiting connection behaviour. The weight of the connection from neuron j to neuron i is denoted as w_{ij}. The network input a_i to a neuron i is the sum of arriving activations. The activation o_i, which is the output of neuron i, is computed by applying an *activation function* $\rho_i(x)$ to its network input a_i.

$$o_i = \rho_i(a_i) = \rho_i(\sum_j w_{ij} o_j) \tag{3.3}$$

These activation functions are typically nonlinear which is an important criterion for the powerfulness of a neural network. Nonlinear neural networks are able to model nonlinear relationships and find nonlinear classification boundaries while a network with linear activation functions is after all still a linear operator and therefore only suitable for linear problems. The most common activation functions include the hyperbolic tangent $\tanh(x)$ and the logistic sigmoid function $\sigma(x)$ [24]:

$$\rho(x) = \tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{3.4}$$

$$\rho(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.5}$$

In the later section on network training it will become apparent that it is important that these functions are differentiable. Their first derivatives are:

$$\tanh'(x) = 1 - \tanh(x)^2 \tag{3.6}$$
$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{3.7}$$

Both functions can be transformed into each other by the following relationship:

$$\tanh(x) = 2\sigma(2x) - 1 \tag{3.8}$$

Thus the choice between the two activation functions has essentially no influence on the set of functions that can be modelled by the network. Due to their characteristic to squeeze an infinite input domain to a finite codomain, see Eq. (3.10) for $\tanh(x)$ and the logistic sigmoid $\sigma(x)$, these functions are also called *squashing functions* [24].

$$\tanh(x): \quad \mathbb{R} \to (-1, 1) \tag{3.9}$$
$$\sigma(x): \quad \mathbb{R} \to (0, 1) \tag{3.10}$$

Figure 3.3 shows both functions and further often used classes of activation functions.

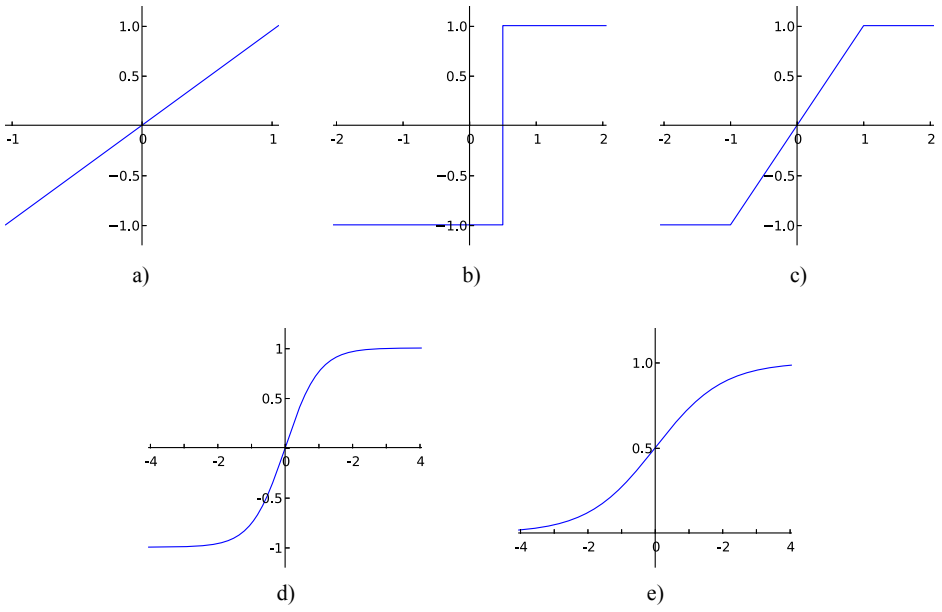


Figure 3.3: Common activation functions: a) linear, b) threshold, c) piecewise linear, d) tanh, e) logistic sigmoid $\sigma(x)$

The most common type of ANN is the feedforward network (FFN). It is distinguished by its acyclic structure of connections between the neurons. The types of ANNs that form cycles are called recurrent, feedback, or recursive, neural networks. They have their benefits especially in the handling of sequences of data. Popular examples of the great variety of FNNs are radial basis function networks, Hopfield nets, Kohonen maps (self organizing maps) and the most widely spread *multilayer perceptron (MLP)*. Since MLPs are used in this work this section will further focus on this versatile and well examined architecture. They are easy to use and implement and there are already many library implementations

available for all kinds of programming languages. For the implementation choice of this work see Appendix A.

MLPs consist of one or more layers of processing units that are arranged linearly with weighted connections between each layer. All connections point in the same direction, so that a layer feeds forward into the next. The input data is presented to the input layer from where the information flows through the so-called *hidden layers* to the last, the output layer as the activations are propagated through the network. This propagation is called the *forward pass*. Figure 3.4 shows a multilayer perceptron with one hidden layer.

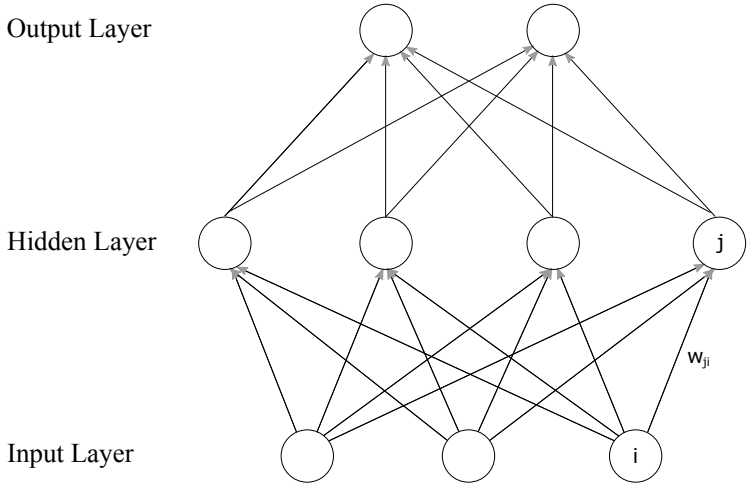


Figure 3.4: An exemplary multilayer perceptron with three inputs, a single hidden layer of four neurons and two outputs units.

A neural network instantiates a function $f(x, w) = y$ that is parameterised by the connection weights w and, due to the size of the parameter space, is capable of implementing many mappings. It has been shown that a multilayer perceptron with only a single hidden layer containing enough nonlinear units can approximate any continuous transformation [9]. Unfortunately, there is no general algorithm to determine the number of hidden units required for a given task. On the one hand, too few neurons reduce the representation capabilities of the neural network. It might not be able to represent a desired mapping at all. On the other hand, too many hidden units diminish the generalization property, it might come to overfitting. It is therefore necessary to determine the optimal structure of a neural network for a specific task via a trial and error approach. As every new network structure requires a training procedure this can become computationally very expensive.

3.2.1 Network Training

This subsection introduces a procedure for training networks in order to approximate a suitable mapping from a given data set using *error backpropagation* and *gradient descent* [37]. The goal is to determine adequate weights by minimizing an appropriate error function. The backpropagation algorithm became popular through Rumelhart et al. [25] and provides an efficient method for evaluating the gradient of this error function. It is suitable for arbitrary feedforward neural networks with differentiable activation functions and a differentiable error function. The schematic procedure is given in Algorithm 3.1. The single steps are explained in more detail below.

This subsection describes the *online* version of the algorithm in which the weights are updated after each presentation of an input vector. There is also a *batch* method in which

Algorithm 3.1 The Backpropagation algorithm [37]

```
1: Initialization of all weights w^(0)
2: repeat
3:   for all input vectors x_n in the training set do
4:     /* Error Backpropagation /*
5:     Forward pass: present input vector x_n to the network and calculate successively
       the activations of all the hidden and output units
6:     Compare the network's output y to the desired output t_n
7:     Calculate the error δ_k for all the output units
8:     Backpropagate the δ's to obtain the errors δ_j for each hidden unit in the net
9:     Evaluate the derivatives of the error function E with respect to each weight
10:
11:    /* Weight Updates /*
12:    Update weights w^(i) to w^(i+1) using the evaluated derivatives
13:  end for
14: until all weights have converged
```

the derivative of the total error E is obtained by repeating the above steps for each pattern in the training set first and then the weights are updated according to the sum over all derivatives.

It is expected here that the error E_n for input vector x_n in the online variant can be expressed as a differentiable function of the network output variables y_k.

$$E_n = E_n(y_1, \ldots, y_m) \tag{3.11}$$

For the batch method it is also assumed that the total error can be written as a sum of all errors E_n for each pattern in the training set.

$$E = \sum_n E_n \tag{3.12}$$

These assumptions are appropriate for nearly all error functions. For example, the error for pattern n could be the standard sum-of-squares error $E_n = \frac{1}{2}\sum(y_k - t_k)^2$. This would lead to the error function

$$E = \frac{1}{2}\sum_n \sum_k (y_{nk} - t_{nk})^2 \tag{3.13}$$

Due to the relationship

$$\frac{\partial E}{\partial w_{ij}} = \sum \frac{\partial E_n}{\partial w_{ij}} \tag{3.14}$$

the further derivation will only consider the online case with one input pattern at a time.

Without loss of generality it is henceforward assumed that each hidden or output unit in the network has the same activation function $\rho(x)$. The equations can be easily adapted for different activation functions $\rho_i(x)$ for each unit.

Initialization of weights

To use a standard gradient descent algorithm it is necessary that the weights are initialized with small random numbers. The convergence speed of the training algorithm depends strongly on the right choice of this initialization method. For this reason, Thimm and Fiesler [32] have experimentally evaluated a variety of weight initialization methods for

feedforward neural networks like multilayer perceptrons on real-world benchmark data sets. They found that for multilayer perceptrons with one hidden layer and a hyperbolic tangent as activation function, the best performance is on average achieved if the initial weights are drawn from a continuous uniform distribution centered around 0 with a variance of 0.2. This leads to a weight range of $[-0.77, 0.77]$. An important consequence of any random initialization is that the training process of each experiment has to be repeated several times in order to determine the significance of the results.

Forward pass

For each pattern the corresponding input vector is supplied to the network. As already mentioned before the activation o_i of a unit i is computed by building a weighted sum of all its inputs and transforming it by a nonlinear activation function $\rho(x)$.

$$o_i = \rho(a_i) = \rho(\sum_j w_{ij} o_j) \tag{3.15}$$

If the inputs o_j in the sum are inputs of the network they are denoted by input x_j and in the case of o_k being the activation of an output unit it is denoted by y_k.

Calculation of the error δ_k for all the output units

The purpose of the backpropagation algorithm is to evaluate the derivative of E_n with respect to some weight w_{ij}. As E_n depends on an arbitrary weight w_{ij} over the summed input a_i to unit i we can use the chain rule for partial derivatives:

$$\frac{\partial E_n}{\partial w_{ij}} = \frac{\partial E_n}{\partial a_i} \frac{\partial a_i}{\partial w_{ij}} \tag{3.16}$$

Using the notation

$$\delta_i := \frac{\partial E_n}{\partial a_i} \tag{3.17}$$

and with

$$\frac{\partial a_i}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_k w_{ik} o_j = o_j \tag{3.18}$$

we get

$$\frac{\partial E_n}{\partial w_{ij}} = \delta_i o_j \tag{3.19}$$

This means that the searched derivative with respect to weight w_{ij} can be determined by multiplying the value of δ_i for unit i at the output end of the weight with the value of o_j for the unit at the input end of the weight. The unit outputs o were already calculated in the forward pass. Only the δ's for all hidden and output units have to be computed.

The δ's of the output units are:

$$\delta_k = \frac{\partial E_n}{\partial a_k} = \frac{\partial E_n}{\partial y_k} \frac{\partial y_k}{\partial a_k} = \frac{\partial E_n}{\partial y_k} \rho'(a_k) \tag{3.20}$$

For example, if the standard sum-of-squares error $E_n = \frac{1}{2} \sum (y_k - t_k)^2$ was used, Eq. (3.20) leads to:

$$\delta_k = \frac{\partial E_n}{\partial y_k} \rho'(a_k) = (y_k - t_k) \rho'(a_k) \tag{3.21}$$

Backpropagation of the δ_k's to obtain the errors δ_j for each hidden unit

Now the evaluation of the δ's for hidden units is considered. At first, the focus is on the weight w_{ij} of a connection leading to a unit i in the last hidden layer, which means all connections from this unit end in an output unit. Thus, the output of unit i contributes to all network outputs y_k. In general, a variation in the a_i of any hidden unit i leads to a variation in all net inputs a_k of the units to which i has an outgoing connection. By using the chain rule for partial derivatives again this can be expressed as:

$$\delta_i = \frac{\partial E_n}{\partial a_i} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_i} \tag{3.22}$$

Substituting Eq. (3.17) and

$$\frac{\partial a_k}{\partial a_i} = \frac{\partial}{\partial a_i} \sum_j w_{kj} \rho(a_j) = w_{ki} \rho'(a_i) \tag{3.23}$$

in Eq. (3.22) the back-propagation formula is obtained:

$$\delta_i = \rho'(a_i) \sum_k w_{ki} \delta_k \tag{3.24}$$

The central principle of back-propagation is that the value of δ for a particular hidden unit is calculated by propagating the δ's backwards from units higher up in the network. Using Eq. (3.24) the δ's for all units in the network can be recursively determined.

Updating weights

After all derivatives of the error function are evaluated using Eq. (3.19) they are used to compute the adjustments to be made to the weights. In many cases a simple gradient descent technique is used. However, there are other, more complex gradient descent algorithms and a variety of different optimization schemes available, which are possibly more efficient, robust and faster than the simple gradient descent considered here. The point in time at which the weights are updated depends on the learning method. In the case of online learning the weights are updated after presentation of each pattern, while in batch learning the adjustments are made after first summing the derivatives over all the patterns in the training set. In each iteration step (denoted by i) the weights are updated according to

$$\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} + \Delta \mathbf{w}^{(i)} \tag{3.25}$$

The derivatives give the gradient $\nabla E(\mathbf{w})$ of the error function[1]. This vector points in the direction of greatest rate of increase of the error function. The simplest approach to using gradient information is to update the weights by a small step in the direction of the negative gradient. This is known as *steepest descent*. The weight change in the batch gradient descent is

$$\Delta \mathbf{w}^{(i)} = -\eta \nabla E(\mathbf{w}^{(i)}) \tag{3.26}$$

The parameter $0 < \eta \ll 1$ is called *learning rate*. In online gradient descent $\nabla E_n(\mathbf{w}^{(i)})$ is used instead. After each iteration step the gradient has to be evaluated again for the

[1] analogous use of $\nabla E_n(\mathbf{w})$ for online gradient descent

new weight vector. The whole process is, theoretically, repeated until the weights have converged to a local or global minimum of the error function. In practice, the steepest descent method often runs into local minima. Several runs of the algorithm with different randomly initialized weights can be helpful to find a sufficiently good minimum [24]. Apart from often converging faster than batch learning the online gradient descent is also less likely to get stuck in local minima as a stationary point of the error function for the whole data set will generally not be a stationary point for each data point individually [15]. A further used method that has often shown to improve learning performance is the randomization of the training set (i.e. the random permutation of its elements) before each *epoch*[2] [15]. The implementation used for this work provides a further modification to the weight change called *momentum* [38]:

$$\Delta \mathbf{w}^{(i)} = \epsilon \Delta \mathbf{w}^{(i-1)} - \eta \nabla E(\mathbf{w}^{(i)}) \tag{3.27}$$

where $0 < \epsilon < 1$ is the momentum parameter. The new term literally adds momentum to the motion of the algorithm through weight space, which helps to escape from local minima and also speeds up convergence.

3.2.2 Input and Output Normalization

Whenever neural networks with sigmoid activation functions are used, the components of the input data should be normalized to the range $[-1, 1]$ [15]. This procedure has the purpose of moving the input values in the unsaturated range of the standard sigmoid activation functions. In the case of gradient-descent learning this can have a considerable effect on network performance as saturated sigmoids lead to vanishing derivatives. In case sigmoid activation functions are also used for the output units of the network it is also necessary to normalize the output values to the codomain of the respective activation function. Otherwise the outputs of the neural network can not reach to values outside of this codomain. A normalization is also recommended if the output values are very small in order to avoid numerical problems with too small network weights.

This normalization procedure requires knowledge about the values the inputs and outputs can take on. In some cases these lower and upper bounds are known, e.g. if there are any physical restrictions. If these are unknown, it is a popular approach to calculate the mean and standard deviation for every input (and output) component over the training set and standardise all data according to these values. All further data like the validation and test sets are standardised with the mean and standard deviation of the training set. Needless to say, the information contained in the data is not changed by this normalization step.

3.3 Locally Weighted Projection Regression

A completely different approach for nonlinear function approximation than neural networks is locally weighted projection regression (LWPR) [34]. This algorithm finds approximations by means of piecewise linear models, even in high dimensional spaces. As it increases the required resources in a purely incremental and data driven way it is suitable for online training and does not require that the data is collected beforehand. This is important if the model may require adaption since the target mapping may change over time. As with neural networks, it is not necessary to have any prior knowledge necessary about the approximated function. The learning algorithm automatically determines the required number of locally linear models and their parameters. One interesting property is that each local model is trained for itself without information about the others. Therefore, this method can be interpreted as a mixture of expert systems in which the experts are trained independently and combine their knowledge to make predictions[10].

[2] a whole pass through the training set

For clarifications, the term *model* is used in two meanings in this section. On the one hand, the model of an approximated function and the trained learning system as a whole is denoted as *LWPR model*. On the other hand, the LWPR model consists of many linear models. These are called *locally linear models* [34], which might be misleading as each model is not only locally but overall linear. If just the term model is used in this section it refers to the these linear models.

3.3.1 Receptive Fields

LWPR is a derivative of receptive field weighted regression (RFWR) [27]. However, the fundamental idea is still the same. The function approximation is achieved by a system of locally linear models. Each linear model is basically of the following form:

$$y_k(\boldsymbol{x}) = \beta_{k,0} + \sum_{i=1}^{N} \beta_{k,i}\, x_i = \boldsymbol{\beta}_k^{\mathsf{T}}\, \hat{\boldsymbol{x}}, \quad \text{with} \quad \hat{\boldsymbol{x}} = (1,\, \boldsymbol{x}^{\mathsf{T}})^{\mathsf{T}} \tag{3.28}$$

where N is the dimensionality of the input space, $\beta_{k,0}$ is the offset in the k-th locally linear model and $\beta_{k,i}$ denotes the parameters of the hyperplane. As also seen in Eq. (3.28), these parameters can be combined in the vector $\boldsymbol{\beta}_k$ by generating the extended input vector $\hat{\boldsymbol{x}}$. A linear model is only valid in a certain region that is called *receptive field* (RF). Each receptive field uses a *Gaussian kernel* to compute to which degree its associated locally linear model is relevant for a given data point $\boldsymbol{x}$:

$$w_k = \exp(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{c}_k)^{\mathsf{T}}\, \boldsymbol{D}_k\, (\boldsymbol{x} - \boldsymbol{c}_k)) \tag{3.29}$$

The weight w_k corresponds to the activation of the receptive field k by the data point $\boldsymbol{x}$. The size and shape of a receptive field is determined by a positive definite distance metric $\boldsymbol{D}_k$, while its location in the input space is specified by the center $\boldsymbol{c}_k$. Therefore, each receptive field consists of a linear model and a gaussian kernel. The total output y of the learning system for a query point $\boldsymbol{x}$ is calculated from the normalized weighted sum of the outputs y_k of all K receptive fields:

$$y = \frac{\sum_{k=1}^{K} w_k\, y_k}{\sum_{k=1}^{K} w_k} \tag{3.30}$$

In the implementation of the algorithm only receptive fields with sufficient activation contribute to the prediction in order to speed up the computations.

Receptive fields are added by the learning algorithm as needed, so the number of receptive fields will automatically adjust to the learning problem. It is important to note that the parameters of each RF, such as the center $\boldsymbol{c}_k$, the distance metric $\boldsymbol{D}_k$ and the parameters of the locally linear model $\boldsymbol{\beta}_k$ are determined independently. No information about the other receptive fields is used. This is an important difference to competitive learning or global function approximators like neural networks. Usually, the forgetting of useful knowledge while focusing on learning from new data is a common problem in incremental learning. By learning each model independently, the robustness towards this so-called *negative interference* is increased [28].

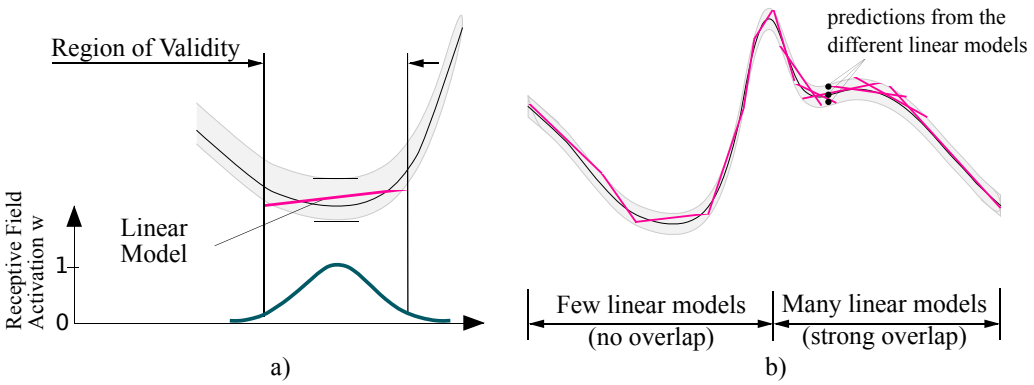


Figure 3.5: A one dimensional example of function approximation with LWPR. a) presents a single locally linear model and its region of validity as determined by the Gaussian kernel of the receptive field. In b) the coverage of a whole function with linear models is shown. Dependant on the curvature few or many linear models are required. (Figure taken from [27].)

Figure 3.5 shows an one dimensional example for function approximation with this concept. It should be mentioned that this section only considers systems with a single output. However, it is possible to approximate functions with multi-dimensional outputs with LWPR as well. Basically, this is done by using separate LWPR models for each output dimension, decomposing the system in several functions with a single output value. These independent models are then combined in one large LWPR model.

3.3.2 Locally Weighted Partial Least Squares

The big difference of LWPR compared to RFWR is that it employs a technique to reduce the dimensionality of the inputs. This way, it has the advantage that it is able to cope with high dimensional input spaces which even might contain redundant and irrelevant dimensions. In order to stay computationally efficient and numerically robust this dimensionality reduction is performed for each local model. In each receptive field a local projection regression is applied to fit the hyperplane of the locally linear model. High dimensional inputs are handled by decomposing multivariate regressions into a superposition of single variate regressions along a couple of selected directions in input space. In order to detect these directions and the corresponding regression parameters a method called locally weighted partial least squares (LWPLS) [35]—a localized variant of the partial least squares (PLS) algorithm [40]—is used. PLS computes recursively orthogonal projections along which a univariate regression is performed. In each iteration step a new projection is added as long as it contributes significantly to the prediction accuracy. The direction of the new projection is the direction of maximal correlation between the residual error and the input data. By choosing the projection direction from correlation between the input and output data irrelevant input dimensions are automatically excluded. On account of this expansion, Eq. (3.28) for the locally linear model changes to:

$$y_k(\boldsymbol{x}) = \beta_{k,0} + \sum_{i=1}^{R_k} \beta_{k,i}\, s_i \tag{3.31}$$

where R_k is the number of local PLS projections, $\beta_{k,i}$ the corresponding regression coefficients and s_i are the elements of the projected input $\boldsymbol{x}$. As the LWPR algorithm places models only where they are needed the allocation of new models can be restricted to the

local dimensionality detected through PLS and only a small part of the whole high dimensional space needs to be filled with local models. However, a effective reduction of high dimensional input spaces is only possible if the data actually lies on low dimensional manifolds. The authors of the LWPR algorithm have shown in [35] that this assumption holds for a large class of real world data. In the case of the task in this work, an analysis for low dimensional distributions is not necessary because the input spaces already have quite few dimensions.

The elements of a LWPR model are shown in a network-like illustration in Figure 3.6. The inputs are routed to all receptive fields. Each one consists of a Gaussian kernel, a projection and a linear regression unit. The overall output of the system is built as weighted average from the outputs of every receptive field.

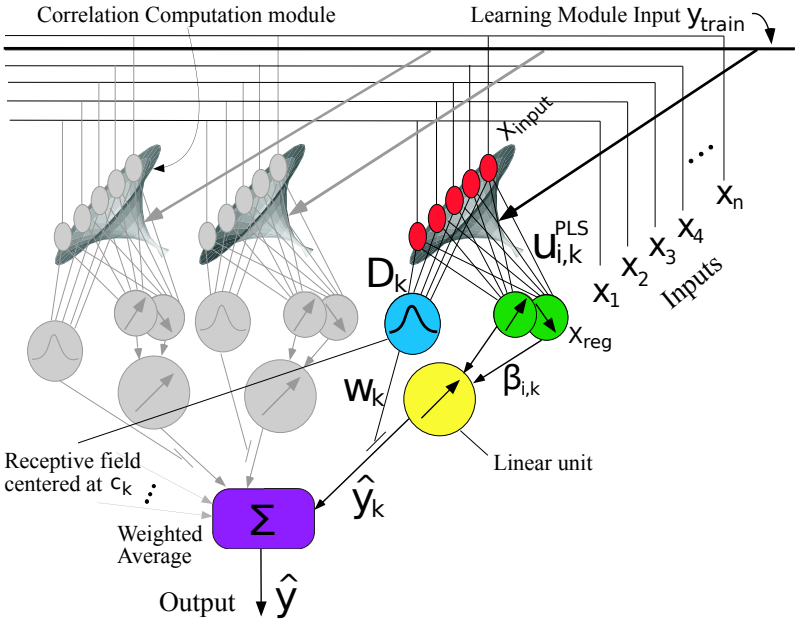


Figure 3.6: An overview over the concept of an entire LWPR model in a network-like illustration. (Figure taken from [35].)

3.3.3 Learning with LWPR

In this subsection the learning procedure of the LWPR algorithm will be introduced. It is designed for an incremental learning scenario in which a data point can be discarded after it is incorporated in the learning system. Data points can be recorded gradually and are presented successively to the algorithm. Therefore, there is no need to collect data beforehand in a training set and equally it is not necessary to store the already used training samples. Also, no validation set is required. The input and output distributions of the data points can be unknown and may even change over time. Usually, the LWPR model adapts fast to these changes in the target mapping as the placement of receptive fields, their size and the local PLS directions can often be kept and only the regression parameters must be calculated again. The LWPR model has a built in forgetting factor that can be used to adjust the time scale expected for changes to occur.

There are basically three main steps that have to be considered in the learning procedure. The first one is the addition of new receptive fields and the decision when this is necessary. Secondly, the update of size and shape of each receptive fields. And finally, the update

of the linear model parameters $\boldsymbol{\beta}$ which involves the projection and regression analysis by means of PLS. As already mentioned, all updates of receptive fields and their locally linear models are done independently. The schematic procedure of the learning algorithm is outlined in Algorithm 3.2 while each step is explained in more detail below.

Algorithm 3.2 The LWPR learning algorithm [34]

```
Initialization of the LWPR model with no receptive field
for all new training samples (x, y) do
    Calculate the activation w_k from Eq. (3.29) for every receptive field
    if no receptive field was activated by more than a threshold w_gen then
        Create a new receptive field with c_k = x, D = D_def and R = 2
    end if
    for all receptive fields (k = 1 to K) do
        Update distance metric D_k of gaussian kernel
        Update projections and regression parameters β_k
        Check if number of projections must be increased
    end for
end for
```

Adding new receptive fields

In the beginning the LPWR model is empty and new locally linear models are only allocated as needed. A new receptive field is created if the input $\boldsymbol{x}$ of a training sample does not activate any of the existing receptive fields by more than a threshold w_{gen}. Its center $\boldsymbol{c}_k$ is set to the current training input $\boldsymbol{x}$ and is never modified afterwards in order to fulfill the independence property of local learning. By this means, negative interference during incremental learning is minimized that could occur due to changing input distributions. The distance metric $\boldsymbol{D}_k$, which determines the size and shape of the gaussian kernel, is initialized to a manually chosen default value $\boldsymbol{D}_{def}$ and the number of projections is set to $R = 2$. The LWPR algorithm also provides a facility to prune a receptive field if it overlaps too much with another receptive field. However, this happens very rarely in real applications and is therefore not implemented in the current version of the algorithm.

Updating the distance metric

The distance metric D_k can be tuned for each local model individually. This way the shape and size of each receptive field is adapted according to the local curvature of the function. This adaption is done by minimizing the following penalized leave-one-out cross validation cost function via stochastic gradient descent:

$$J = \frac{1}{\sum_{i=1}^{M} w_i} \sum_{i=1}^{M} w_i (t_i - y_{i,-i})^2 + \frac{\gamma}{N} \sum_{i,j}^{N} D_{ij}^2 \tag{3.32}$$

where N is the dimensionality of the input space, M the number of used training samples and γ a tradeoff parameter. The subscript $i, -i$ refers to the leave-one-out cross validation that is used to ensure a reasonable generalization. For that reason, it is not necessary to have a validation set. The first term in Eq. (3.32) is the mean cross validation error while the second term is a penalty term that prevents indefinitely shrinking of the receptive field in case of very much training data. It should be mentioned that this cost function can be minimized in an incremental fashion that does not require to store all training data [28]. As usual in gradient descent a learning rate α is used for updates to the distance metric. The learning rate is adjusted automatically using meta learning.

Update of projections and regression parameters

In order to perform the univariate regressions it is necessary for every local model to keep track of some statistical information as the learning algorithm does not store any training data. The already mentioned forgetting factor enables an exponential forgetting of older data in the statistics. Given a large enough number of training samples the algorithm is able to accumulate enough statistics to detect the required number R of PLS projections and the scale on which the function is locally linear. If there is only little training data the samples should be presented to the learning algorithm several times in random order. In order for PLS to find the optimal projection direction—which corresponds to the gradient of the local linearization parameters of the function to be approximated—all the input variables should be statistically independent and have equal variance. As the algorithm allows a specification of component-wise input normalization factors it is not necessary to manually normalize the inputs to equal variance beforehand. For a derivation of all the update rules required in this part of the algorithm and detailed information about the necessary statistics, see [36].

Increasing the number of projections

The algorithm recursively increases the number of projections R in each local model depending on the progression of the approximation error. Beginning with $R = 2$ it adds projections as long as the approximation error of a further projection decreases by more than a certain percentage of the previous error.

4. Acquisition of the Muscle Jacobian

This chapter presents some general considerations regarding the acquisition of the muscle jacobian which are important before the actual realization of the task is approached. First of all, it is reasoned how the muscle jacobian can be extracted and at which stage the machine learning techniques are applied best. The second and third section discuss the most suitable form of representation for rotations in three dimensional space and how the muscle jacobian can be extended to joints with multiple degrees of freedom. Afterwards, the proposed techniques for movement generation and data acquisition are described. The last section explains the approach to decompose the function approximation task in multiple subtasks.

4.1 Extraction Procedure

Chapter 2 already introduced the task of this thesis, the extraction of the mapping from joint angles to muscle lengths for an anthropomimetic robot arm in form of the muscle jacobian. For this purpose, the machine learning techniques introduced in the previous chapter shall be applied. At a first closer inspection of this task the question arises whether the muscle jacobian should be approximated directly or if an intermediate step by means of learning the direct mapping $\boldsymbol{l} = f(\boldsymbol{q})$ would be better.

The most important factor in this consideration is that it is not possible to collect the partial derivatives of the muscle jacobian directly, neither for the simulation nor for the real robot. Only the joint angular values and the muscle lengths can be recorded. Therefore, in order to obtain the muscle jacobian it is necessary to compute it by numerical derivation either before or after the learning process.

The partial derivatives can be obtained from the mapping $f : \boldsymbol{q} \rightarrow \boldsymbol{l}$ between joint angles and muscle lengths using the central difference quotient [4]. For a given angular configuration $\boldsymbol{q}$ the muscle jacobian can be computed as follows:

$$\boldsymbol{J}_m(\boldsymbol{q}) = \frac{f(\boldsymbol{q} + \Delta\boldsymbol{q}) - f(\boldsymbol{q} - \Delta\boldsymbol{q})}{2\Delta\boldsymbol{q}} \tag{4.1}$$

Eq. (4.1) is actually an abbreviated notation. As f is a multivariable function the central

difference formula extends in this case to (see [33]):

$$[\boldsymbol{J}_m(\boldsymbol{q})]_{ij} = \frac{f_i(q_1, \ldots, q_j + \Delta q, \ldots, q_n) - f_i(q_1, \ldots, q_j - \Delta q, \ldots, q_n)}{2\Delta q} + O(\Delta q^2) \quad (4.2)$$

where f_i is the i-th output of mapping f. The subscript j refers to the joint, while i refers to the corresponding muscle. As always in numerical derivation there is an estimation error involved. For the central difference formula this error is in the magnitude of ϵ^2. It can be easily seen that the number of sampling points required for the computation of the muscle jacobian increases linearly with the number of muscles and joints.

However, function f is unknown and as described before it might not be possible to determine it analytically. A machine learning technique can be applied in order to train a function approximator $\hat{f} \approx f$ on previously recorded joint angles and muscle lengths. This approximator can then be used instead of f in Eq. (4.2) to estimate the elements of the muscle jacobian around a given position $\boldsymbol{q}$ for small angular changes $\Delta q \ll 1$:

$$[\boldsymbol{J}_m(\boldsymbol{q})]_{ij} \approx \frac{\hat{f}_i(q_1, \ldots, q_j + \Delta q, \ldots, q_n) - \hat{f}_i(q_1, \ldots, q_j - \Delta q, \ldots, q_n)}{2\Delta q} \quad (4.3)$$

An alternative to this approach would be to perform the numerical derivation before the application of the function approximation. At first, the required values of Eq. (4.2) are collected. This means that the muscle lengths have to be recorded for small changes in each joint angle around given positions, respectively. This data is afterwards used to compute a training set of muscle jacobians on which eventually a function approximator $\hat{g} : \boldsymbol{q} \rightarrow \boldsymbol{J}_m$ can be trained. This method would require the ability to change each joint angle precisely and individually. However, the motivation for this work was that the robot can not be specifically controlled to a given pose in the first place. This approach is therefore only possible in the simulation which allows the setting of joint angles. The extraction technique developed in this work is supposed to be eventually applied to the real robot, though. Apart from that, it is planned to extend the learning method in the future by an online scheme in which the recording of data and the learning process is continued during the normal operation of the robot (see Chapter 7.1.2). The collection of necessary values to compute the muscle jacobian before learning would be impossible for this strategy.

For these reasons, it was decided to pursue the first approach which learns an approximation of the direct mapping and numerically derives it afterwards in order to obtain the muscle jacobian. If f is known the muscle jacobian can be easily retrieved from it while the reverse direction is not possible. Therefore, learning an approximation of f also enables further applications for whole body control. By means of higher order methods for the numerical derivation which evaluate f for more than just two points—for example the five-point stencil [4]—the order of the estimation error can be increased. However, the increase in evaluation points also leads to a proportional increase in computation time. As the muscle jacobian shall be used for whole body control and the calculation of the partial derivatives is necessary in every pass of the control loop the computation time is certainly a critical factor. It was therefore decided to use the central difference quotient for the time being.

4.2 Representation of Rotations in 3D Space

The angles of joints with three DoF, like a spherical shoulder joint, need to be represented by multiple values. This section discusses the question of the most suitable representation

of a rotation in three dimensional space for this task. The minimum number of required parameters for an orientation in 3D space is three and there are many different forms for this like ZXZ-Euler-Angles or XYZ fixed angles. However, it can be shown that a unique and continuous representation without singularities is only possible with more than three parameters. The most compact form with these properties are unit quaternions [30] with four scalar parameters.

A unit quaternion $\boldsymbol{\epsilon} = [\epsilon_0, \epsilon_1, \epsilon_2, \epsilon_3]^\mathsf{T}$ is defined as:

$$\boldsymbol{\epsilon} = \epsilon_0 + \epsilon_1\, i + \epsilon_2\, j + \epsilon_3\, k \tag{4.4}$$

where the components ϵ_0, ϵ_1, ϵ_2, and ϵ_3 are scalars and i, j, and k are operators. The components of a unit quaternion are constrained to satisfy the following condition:

$$\epsilon_0^2 + \epsilon_1^2 + \epsilon_2^2 + \epsilon_3^2 = 1 \tag{4.5}$$

Unit quaternions or rotation matrices are virtually always the representation of choice for control applications. For this work it was decided to use unit quaternions due to their reduced number of parameters compared to rotation matrices.

4.3 Muscle Jacobian for Joints with Multiple DoF

The muscle jacobian as defined in Eq. (2.2) on page 6 is only valid for joints with a single degree of freedom. However, the anthropomimetic robots considered in this work feature also joints with three DoF. During the data acquisition the angles of all joints in the robot are measured and stored in a vector $\boldsymbol{\alpha}$. This vector combines the angular values of all hinge and spherical joints in a particular representation. As previously described there are various forms to represent an orientation in three dimensional space. The vector $\boldsymbol{\alpha}$ is therefore not the same as the previously used pose vector $\boldsymbol{q}$. Using the vector $\boldsymbol{\alpha}$ as angular inputs of the learning algorithm the function $\boldsymbol{l} = f_\alpha(\boldsymbol{\alpha})$ is approximated. Building the difference quotient of this function as in Eq. (4.2) leads to

$$\boldsymbol{J}_m^\alpha(\boldsymbol{\alpha}) \;=\; \left[\frac{\partial \boldsymbol{l}_m}{\partial \boldsymbol{\alpha}}\right] \tag{4.6}$$

In order to use this jacobian matrix $\boldsymbol{J}_m^\alpha(\boldsymbol{\alpha})$ in Eq. (2.2) it needs to be transformed to $\boldsymbol{J}_m(\boldsymbol{q})$. Although it was decided to use unit quaternions in this work this section will derive how this transformation can be performed for an arbitrary representation as other forms will be used on another occasion later on.
The muscle jacobian $\boldsymbol{J}_m(\boldsymbol{q}) = [\partial \boldsymbol{l}_m / \partial \boldsymbol{q}]$ can be expressed through $\boldsymbol{J}_m^\alpha(\boldsymbol{\alpha})$ by solving Eq. (4.6) for $\partial \boldsymbol{l}_m$ and dividing by $\partial \boldsymbol{q}$:

$$\boldsymbol{J}_m(\boldsymbol{q}) \;=\; \left[\frac{\partial \boldsymbol{l}_m}{\partial \boldsymbol{q}}\right] \;=\; \boldsymbol{J}_m^\alpha(\boldsymbol{\alpha}) \left[\frac{\partial \boldsymbol{\alpha}}{\partial \boldsymbol{q}}\right] \tag{4.7}$$

Expressing $[\partial \boldsymbol{\alpha} / \partial \boldsymbol{q}]$ as a matrix $\boldsymbol{A}(\boldsymbol{\alpha})$:

$$\boldsymbol{J}_m(\boldsymbol{q}) \;=\; \boldsymbol{J}_m^\alpha(\boldsymbol{\alpha}) \boldsymbol{A}(\boldsymbol{\alpha}) \tag{4.8}$$

Matrix $\boldsymbol{A}(\boldsymbol{\alpha})$ in turn can be expressed using the time derivatives $\dot{\boldsymbol{\alpha}}$ and $\dot{\boldsymbol{q}}$. The transformation from time derivatives to rotational velocities $\boldsymbol{\omega} = \dot{\boldsymbol{q}}$ is a well known problem for

almost any representation form of three dimensional rotations. Expanding Eq. (4.8) with ∂t results in:

$$\dot{\boldsymbol{\alpha}} = \frac{\partial \boldsymbol{\alpha}}{\partial t} = \boldsymbol{A}(\boldsymbol{\alpha})\frac{\partial \boldsymbol{q}}{\partial t} = \boldsymbol{A}(\boldsymbol{\alpha})\dot{\boldsymbol{q}} \tag{4.9}$$

For the quaternions used in this work this relationship between their time derivatives $\dot{\boldsymbol{\epsilon}}$ and the corresponding rotational velocities $\boldsymbol{\omega} = \dot{\boldsymbol{q}}$—which is necessary to built the matrix $\boldsymbol{A}(\boldsymbol{\alpha})$ in Eq. (4.9)—is [30]:

$$\dot{\boldsymbol{\epsilon}} = \boldsymbol{B}_\epsilon(\boldsymbol{\epsilon})\,\boldsymbol{\omega} \quad \text{, with} \quad \boldsymbol{B}_\epsilon(\boldsymbol{\epsilon}) = \frac{1}{2}\begin{bmatrix} -\epsilon_1 & -\epsilon_2 & -\epsilon_3 \\ \epsilon_0 & \epsilon_3 & -\epsilon_2 \\ -\epsilon_3 & \epsilon_0 & -\epsilon_1 \\ \epsilon_2 & -\epsilon_1 & \epsilon_0 \end{bmatrix} \tag{4.10}$$

4.4 Data Acquisition

While recording data for training or testing it is important that all muscles are at least under a minimal strain. Otherwise, the mapping from joint angles to muscle lengths is not uniquely defined. This can already be seen by considering a simple arrangement of two antagonistic muscles spanned over a joint with one DoF, like in the case of Brachialis and Triceps muscles. If the Brachialis is contracted the arm bends and the joint angle takes a certain value. As long as the antagonistic Triceps muscle does not exert any force this joint angle does not change. This means that the Triceps can be extended arbitrarily, associating one joint angular value with infinitely many muscle length values. Therefore, the occurrence of slack muscles must be avoided during acquisition of data.

However, as we are only using the simulation of a simplified model in this work this problem does not occur. The muscle lengths are so far calculated as the geometric distance between their two attachment points. This way, the length of slack muscles is calculated as if they were tense, simplifying the acquisition of data very much. The gathering of data can be done independently and asynchronously to the generation of robot movements. Hence it was decided to record samples in predefined time intervals as long as the configuration of the robot changes.

The generation of robot movements is another challenge. Since it is not possible to simply control the robot in a given position it is necessary to find other ways of positioning the robot during the acquisition of training data. These should be able to cover the whole input space as good as possible. The following two methods have been used in this work.

4.4.1 Setting of Joint Angles

The used simulation software supports the explicit setting of each joint angle in the model. This allows to walk in small steps through the entire space of possible angle configurations and record the corresponding muscle lengths. Just setting a angular configuration might result in slack muscles but as mentioned above this is not a problem with the type of muscles currently used in the simulation model. This method has the advantage that it guarantees a even coverage of the input space which can be adjusted by means of the step size. However, it is restricted to the simulation with this special type of muscles.

4.4.2 Generation of Random Movements

The second method developed to produce data is comprised of the generation of random movements. For this purpose, the control of motor positions in each muscle is used. At

very short time intervals (every 300 ms) a muscle is chosen randomly and its motor position is set to a random value in the range of possible values. This range is specific for the chosen muscle and should comprise the complete radius of movements in which it is involved. The described approach results in a completely uncoordinated interaction of many muscles which persistently change between cooperation and working against each other. A synchronization is automatically achieved through the compliant anthropomimetic structure of the robot which balances the effects of all contributing muscles.

It was discovered that the motion sequence can be improved by setting the force reference value of one or more muscles to zero from time to time, also chosen randomly. This results in an instant relaxation of the corresponding muscles and avoids that the robot gets stuck in some configurations for a longer period of time.

The biggest problem of this approach is the coverage of the space of possible angle configurations. It has been observed that the robot arm is moved much more in certain areas of the input space than in others if every muscle is chosen with the same probability. The reason for this seems to be the uneven arrangement of muscles. There are several muscles contributing to movements in the front, especially front left area, while only a few lead to movements to the right side or backwards. Therefore, the option to manually adjust the probability with which the muscles are chosen was introduced. As a result, the arm movements are distributed more evenly over the configuration space if muscles leading to movements to the right side or backwards are chosen more often than others.

On a side note, it is assumed that the concept of generating random movements and observing the results is also a behavior in humans and other mammals. Developmental psychology suspects that infants learn to control their body through systematic exploratory movements [23]. For instance, they apparently randomly wave their arms as they learn to control their body and reach for objects. Since the ECCEROBOT project tries to mimic the way humans perceive and interact with their environment this approach of movement generation would certainly be one more step towards this goal.

4.5 Decomposition in Multiple Models

A basic property resulting from the setup of the robot is that not all muscle lengths depend on all joint angles. If a single approximator is trained for the mapping between all joint angles and all muscle lengths, the learning algorithm has to determine which inputs are relevant for which outputs. In addition, the higher the number of DoF and muscles in the robot gets the more complicated this model becomes and the probability gets higher that it is less accurate. However, the structure of the robot introduces a segmentation in independent subsystems. By manually specifying this structure the task of learning the mapping from joint angles to muscles lengths for an entire robot can be divided in the training of several simpler models.

In a first step, the mapping can be decomposed into groups of muscles that depend on the same combination of joint angles. For instance considering the test rig, Table 2.1 on page 8 shows that there are eight muscles depending on the angles of the shoulder joint, two muscles depending on the angle of the elbow joint, and one muscle depending on both. This suggests a manual partition in those three groups. Furthermore, as all muscle lengths are independent of each other the mapping can be split according to the outputs in models for one or more muscles. This concept is exemplarily illustrated in Figure 4.1.

Such a segmentation would have to be specified manually and therefore has the drawback that it requires the provision of knowledge about the setup of the robot. However, the experimentation results as presented in the following chapter will show whether a suitable decomposition will lead to better approximation results than as expected.

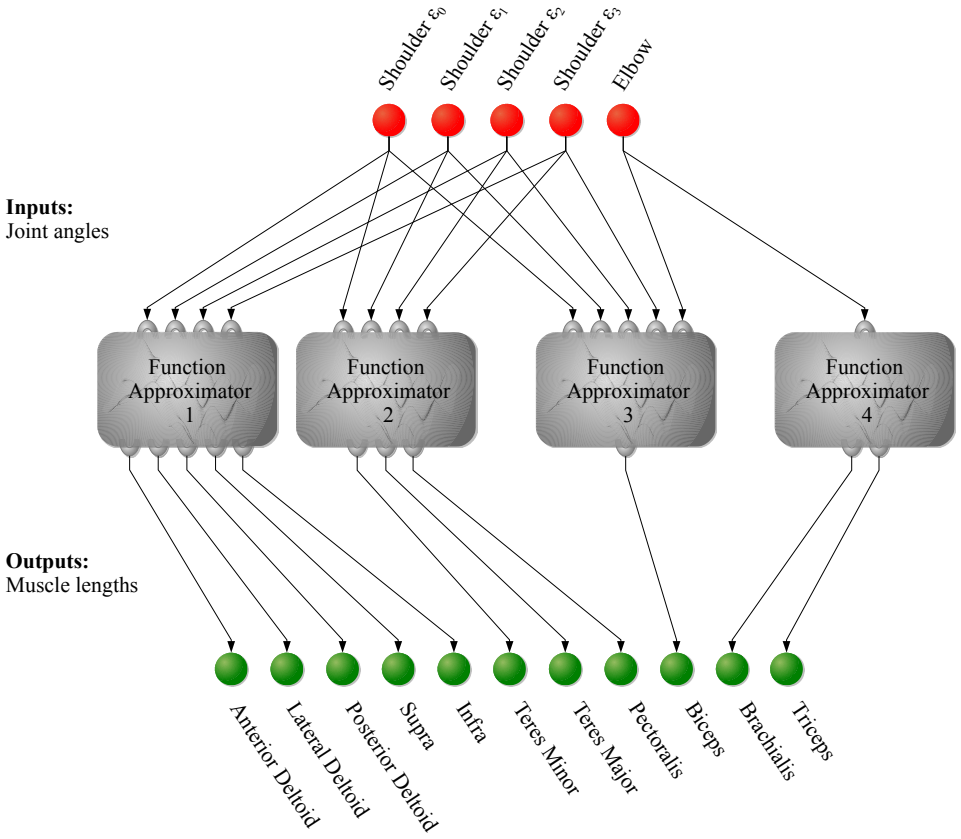


Figure 4.1: This figure shows a decomposition of the function approximation task for the test rig. Two approximators are used for the muscles spanned over the shoulder joint and get only the quaternion parameters as input values. A third one is concerned with the Biceps and accesses the shoulder and elbow angular parameters. Finally, the fourth approximator is associated with the Brachialis and Triceps which depend only on the elbow joint. The allocation of muscles as well as the choice to use four approximators was done arbitrarily.

5. Experimental Results I: Test Rig with Elbow only

This is the first chapter presenting the various results achieved during the experiments with both neural networks and linear weighted projection regression. For all obtained machine learning models the approximation accuracy of the mapping from joint angles to muscle lengths as well as of the resulting muscle jacobian was evaluated. The first section describes the methods that were used for these evaluations. Afterwards, the results attained with a very simple robot arm are presented.

5.1 Performance Evaluation

5.1.1 Evaluation of the Direct Mapping

The evaluation of the direct mapping from joint angles to muscle lengths can be done in a straight forward manner. Some test data can be gathered using the previously described techniques (see Chapter 4.4). This data should not have been used for previous training and cover the input space as good as possible. By applying the trained approximators on the inputs of the test data the previously introduced normalized mean squared error can be calculated for every output dimension, i.e. every muscle. Additionally, the absolute mean error for each muscle can be determined which gives the mean deviation of the estimated muscle lengths in meters:

$$\overline{err}_k = \frac{1}{|T|} \sum_{i}^{|T|} |h(\boldsymbol{x}_i)_k - t_{i_k}| \tag{5.1}$$

where $|T|$ is the size of the test set, $h(\boldsymbol{x}_i)_k$ is the k-th element of the approximated output for input $\boldsymbol{x}_i$ and the t_{i_k} is the k-th element of the i-th target vector.

5.1.2 Evaluation of the Muscle Jacobian

The evaluation of the muscle jacobian is more challenging. In order to compare the estimate, a reference value is required which can only be obtained by numeric derivation. One method to receive such a reference for the muscle jacobian is presented in the following. Starting from the pose for which the muscle jacobian is to be computed the changes in muscle lengths must be determined for slight variations of the inputs in each dimension.

This original robot configuration is described by the pose vector $\boldsymbol{\alpha}$ which contains the angles of all hinge and spherical joints in a particular representation. New configurations $\boldsymbol{\alpha}_i^{\pm}$ are generated by adding—and substracting if the central difference quotient is used—small δ's to dimension i of $\boldsymbol{\alpha}$ and leaving all the other dimensions the same:

$$\boldsymbol{\alpha}_i^{\pm} = \boldsymbol{\alpha} \pm \delta\, \boldsymbol{\eta}_i \tag{5.2}$$

The robot is moved into all of these configurations and the corresponding muscle lengths are recorded. The gathered data can eventually be used in Eq. (4.1) on page 25 to compute a reference for the muscle jacobian at configuration $\boldsymbol{\alpha}$. However, as it is not possible to specifically control the robot to a given pose this method can only be used in the simulation which allows the setting of joint angles.

In order to get a meaningful statement about the accuracy of the approximated muscle jacobian this generation of reference values has to be repeated for as many configurations as possible, distributed over the entire input space. Naturally, the more DoF the robot has—and therefore the more entries there are in $\boldsymbol{\alpha}$—the more data points must be collected in a single execution of this procedure. It has turned out that the precise setting of joint angles in the used simulator is a quite time consuming issue. Therefore, the higher the number of input dimensions is the more expensive becomes this process and the fewer evaluations can be obtained in a reasonable period of time. Needless to say, numeric derivation using the difference quotient is always error-prone. Apart from that, it was found that it is not possible to change a joint angle in the simulator without slightly changing other angles as well. Therefore, the reference values calculated with this method might not exactly represent the real muscle jacobian. Unfortunately, no way around these disadvantages could be found.

This method is not applicable for arbitrary representations of three dimensional rotations the pose vector $\boldsymbol{\alpha}$ might contain. For instance, as the parameters of quaternions have to fulfill certain constraints it is very likely that the change of one parameter leeds to an invalid representation. This problem does not occur for Euler-Angles. Additionally, the used simulator requires the specification of angles for spherical joints in the representation of XYZ-Euler-Angles. Therefore, it was decided to use this representation in the pose vector $\boldsymbol{\alpha}$ for all spherical joints during the procedure described above. The muscle jacobian obtained this way is as a consequence also based on XYZ-Euler-Angles and needs to be transformed as described in Chapter 4.3. This requires the knowledge of the relationship between the time derivatives $\dot{\boldsymbol{\gamma}}$ of XYZ-Euler-Angles $\boldsymbol{\gamma} = [\gamma_x,\, \gamma_y,\, \gamma_z]^{\mathsf{T}}$ and the corresponding rotational velocities $\boldsymbol{\omega}$. As deduced in Appendix B this relationship is:

$$\dot{\boldsymbol{\gamma}} = \boldsymbol{B}_{\mathrm{XYZ}}(\boldsymbol{\gamma})\, \boldsymbol{\omega} \quad \text{, with} \quad \boldsymbol{B}_{\mathrm{XYZ}}(\boldsymbol{\gamma}) = \begin{bmatrix} 1 & \frac{\sin(\gamma_x)\sin(\gamma_y)}{\cos(\gamma_y)} & \frac{\cos(\gamma_x)\sin(\gamma_y)}{\cos(\gamma_y)} \\ 0 & \cos(\gamma_x) & \sin(\gamma_x) \\ 0 & -\frac{\sin(\gamma_x)}{\cos(\gamma_y)} & \frac{\cos(\gamma_x)}{\cos(\gamma_y)} \end{bmatrix} \tag{5.3}$$

Another method one could think of to examine the estimated muscle jacobian is its use in whole body control. The obtained muscle jacobian is employed in the controller approach described in Chapter 2 and it can thus be examined how well the robot can be controlled. Of special interest in this context would be how accurately the controller moves the robot's pose to the specified angular reference values. Needless to say, this is an indirect and unreliable method. As already shown in the introduction of the controller scheme it is quite complex and consists of many computation steps. This makes it very difficult to deduce the influence of the muscle jacobian compared to all the other controller components from

these observations. This method can therefore not be considered as a real alternative to the first described approach.

5.1.3 Evaluation of the Runtime

It has already been mentioned several times that the estimation of the muscle jacobian is supposed to be used in whole body motion control. This requires an evaluation of the function approximator(s) and the computation of the muscle jacobian in every pass of the control loop. As a result, the computation time of these steps is a critical factor for the functionality and stability of the controller. The runtime of these computations should therefore be evaluated as well. For this purpose, the runtime for the computation of the direct mapping using a particular function approximator was averaged over 10000 executions. The same procedure was also repeated for the computation of the muscle jacobian. As a matter of fact, the results are only valid for the specifically used software implementation and hardware. But it does still provides a general benchmark and allows a comparison of the different used machine learning methods.

The implementation of this work (see Appendix A) was compiled for these runtime tests with all the standard optimization flags of the GNU G++ Compiler. The tests were run under Ubuntu 10.10 Linux on a Intel Core i5 2.4 GHz machine with 4GB of memory.

5.2 Test Rig with Elbow Joint only

Before trying to extract the muscle jacobian for the entire test rig it was focused on a simplified case which consists only of the elbow joint. It is in principle the same model as the full test rig but there are only three muscles, Biceps, Brachialis and Triceps. Since the shoulder joint was removed the Biceps spans only the elbow joint. The data required for training with the machine learning algorithms was collected in simulation by sampling the full range of the elbow from 0° to almost 180° in steps of 0.005 radians. This resulted in an extensive training set of 3657 samples consisting of the elbow angle in radians and the corresponding muscle length values in meters. The reference values for the evaluation of the muscle jacobian were retrieved by the method described in Section 5.1.2. These values were computed for 293 evenly distributed angular positions of the elbow joint by numerical derivation of the values returned by the simulator.

The simulation provides very exact values for the angles of the joints and the lengths of the muscles. Regarding the eventual application for the real robot this is a quite unlikely scenario. In order to gather training and test data for a real robot both joint angles and muscle lengths have to be measured. For the joint angles it is planned to use a stereo-vision tracking system with an expected accuracy of 1-2 degrees. It is assumed that the measurements of the muscle lengths will be distinctly more inaccurate because they have to be composed of the motor position and the extension of the shock cord. The latter is computed via noisy force sensor values and an estimation of the suspension rate of the shock cord. In order to take account for these measurement errors, this work also examined the performance of the used machine learning methods when trained on data containing artificial noise added to both inputs and outputs. All function approximators were trained on two cases of noisy data. In the first case, zero-mean Gaussian noise with a standard deviation of one degree for the joint angles and one millimeter for the muscle lengths was added. This is supposed to be very low noise. In the second case, the standard deviation of the noise was increased to two degrees for the joint angles and five millimeters for the muscle lengths. These values are expected to be much more realistic for the measurements on the real robot.

5.2.1 Analytic Examination

First of all, it is examined of which kind the relationship between the angle of the elbow joint and the lengths of the three muscles is. For this purpose the length l_{Br} of the Brachialis muscle is derived exemplarily in dependence of the elbow angle α (see Fig. 5.1).

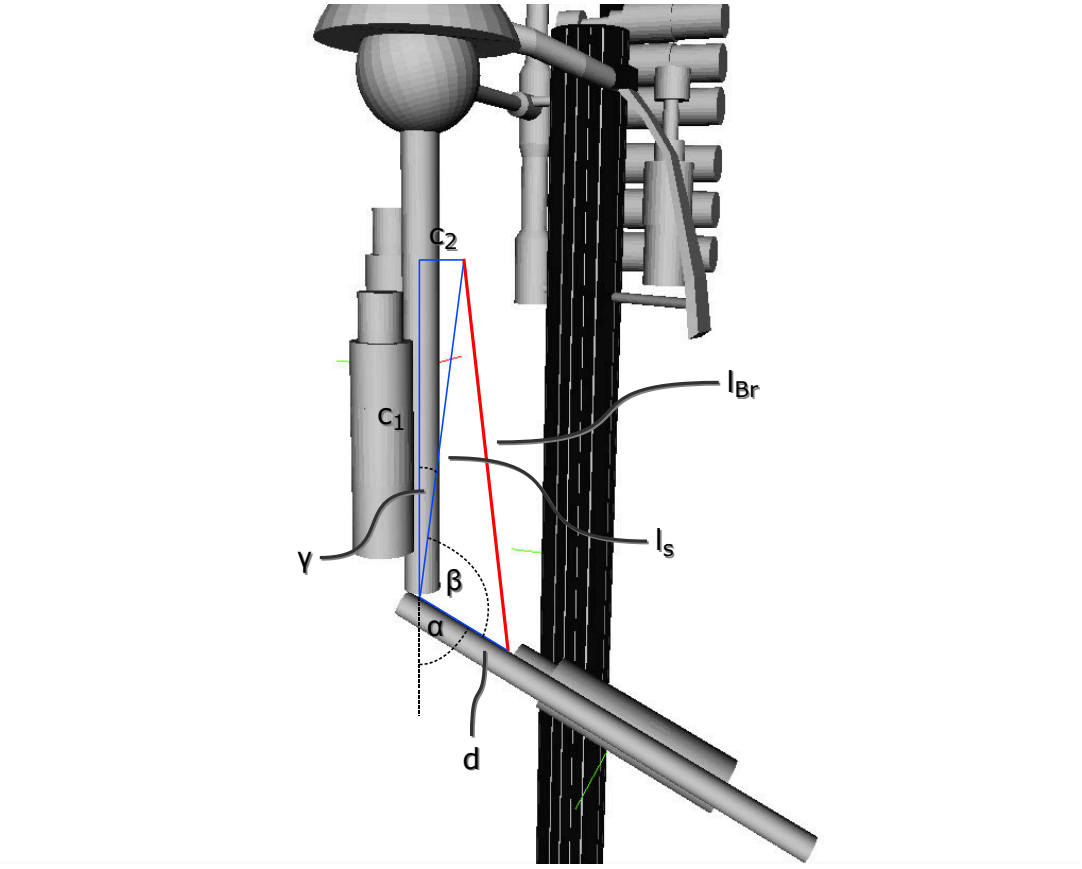


Figure 5.1: This figure shows all the variables used in the analytic derivation of the Brachialis length l_{Br} in dependence of the elbow angle α. The Brachialis muscle is indicated as red line.

It is recognizable from the figure that the elbow angle is defined to have value zero for a completely stretched out arm. Thus, $\alpha = 180° - \beta - \gamma$. Using the the law of cosines in trigonometry one gets:

$$l_{Br}^2 = l_s^2 + d^2 - 2\, l_s\, d\, \cos(180° - \alpha - \gamma) \tag{5.4}$$

Also it is known that:

$$l_s^2 = c_1^2 + c_2^2 \tag{5.5}$$

$$\cos\gamma = \frac{c_1}{l_s}, \qquad \sin\gamma = \frac{c_2}{l_s} \tag{5.6}$$

Using the substraction theorems for cosine and Eq. (5.5) and Eq. (5.6):

$$l_{Br}^2 = c_1^2 + c_2^2 + d^2 + 2\,c_1\,d\,\cos\alpha - 2\,c_2\,d\,\sin\alpha \tag{5.7}$$

Expressions of similar form can be derived for the Biceps and Triceps muscles but are omitted at this point.

5.2.2 Neural Networks

Several experiments were carried out with single neural networks which consist of one input neuron for the elbow angle and three output neurons for the muscle lengths. It turned out that a hidden layer of five neurons is already enough to achieve very good results. While the hidden neurons are equipped with the tanh, the output units have a linear activation function. Separate networks were trained on the original training data without noise, on the training data with added low noise and again with high noise. In the further text, low noise will always mean the addition of zero-mean Gaussian noise with a standard deviation of one degree for the joint angles and 1 mm for the muscle lengths, while high noise means a standard deviation of two degrees for the joint angles and 5 mm for the muscle lengths. The networks were trained using the incremental update version of the backpropagation algorithm with a learning rate of $\eta = 0.01$. It was also experimented with other learning rates but only worse or similarly good results could be achieved. As the results of [32] suggest all weights were randomly initialized in the range $[-0.77, 0.77]$. However, it should be mentioned that it was also experimented with Gaussian distributions and different standard deviations but no significant difference in the results could be noticed. The random weight initialization required that the training in each experiment was repeated several times and eventually the best network was chosen. In order to avoid overfitting 15% of the training set was split off and the development of the error on this validation set was observed during training. Additionally, all input as well as all output values were scaled to the interval $[-1, 1]$. This is especially important for the muscle length values because the simulator returns these values in meters but the lengths vary only in the scope of a few centimeters or even millimeters. This normalization is completely done internally by the used software classes and therefore the data from the simulator can be used without further processing.

Initially, it was also tried to use a version of the cascade-correlation training algorithm [6]. Standard neural networks have the problem that the network size has to be tuned manually. The cascade-correlation algorithm overcomes this drawback by starting with an empty network and gradually adding new neurons as long as needed. It is also supposed to solve some other common problems in network training, like getting stuck in local minima or having to manually tune the step size in gradient descent. However, even after extensive testing no satisfactory results could be achieved with this approach. It is a common observation that not all problems are equally well suited for the cascade-correlation algorithm. Therefore, this was not further pursued.

5.2.2.1 Results for the Direct Mapping

Table 5.1 shows the error measures of the neural networks if evaluated on test data gathered with the simulator. The results are grouped according to the training of the networks without, with low and with high noise. For each of these cases, the normalized mean squared error as well as the absolute mean error are given for each output dimension separately, i.e. for each muscle. The mean error was scaled to millimeters for easier legibility.

	No Noise		Low Noise		High Noise	
	nMSE	$\overline{\text{error}}$ [mm]	nMSE	$\overline{\text{error}}$ [mm]	nMSE	$\overline{\text{error}}$ [mm]
Biceps	1.96e-5	6.37e-2	7.14e-5	1.34e-1	3.61e-4	2.89e-1
Brachialis	1.48e-5	5.68e-2	8.48e-5	1.51e-1	1.95e-4	2.47e-1
Triceps	8.88e-6	8.63e-2	6.40e-4	4.93e-1	4.97e-4	4.55e-1
Average	1.45e-5	6.89e-2	2.65e-4	2.59e-1	3.51e-4	3.30e-1

Table 5.1: The error values for the best neural networks trained on the original training data and with addition of low and high noise. The normalized mean square error and the mean absolute error is given separately for each muscle and in average.

It becomes obvious from the table that the network trained on the original data exhibits negligible errors. The direct mapping from the elbow angle to the lengths of the three muscles is approximated fairly exactly. As expected, the accuracy decreases if noise is added to the training data. The differences in the approximation error between low and high noise are relatively small. In both cases, the average error in the estimated muscle lengths is below half a millimeter and therefore still in a very acceptable range. This fact is also illustrated by Figure 5.2. It shows the part of the training data that belongs to the Triceps muscle, without and with addition of high noise. The overlaid outputs of the neural networks show that the graph of the network trained on the original training data coincides completely with the target. The outputs of the networks trained on the data with noise deviate only slightly on the lower end of the angular range.

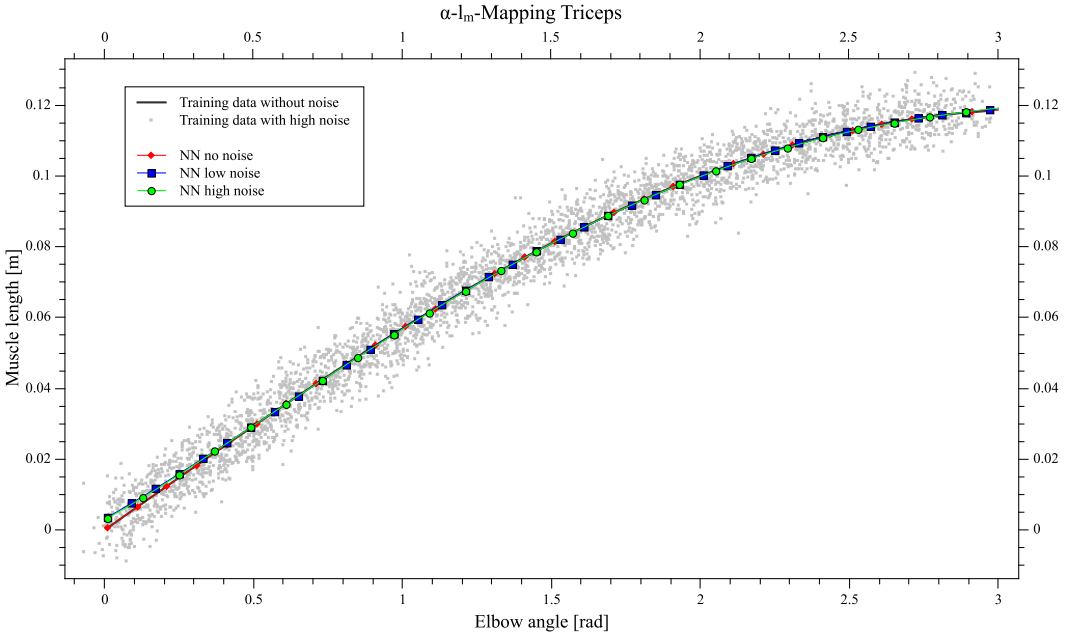


Figure 5.2: The training data belonging to the Triceps muscle with and without high noise overlaid with the outputs from the best neural networks.

The runtime measurement according to the system specifications in Section 5.1.3 returned in average 0.0014 ms for a single evaluation of a neural network.

5.2.2.2 Results for the Muscle Jacobian

At this point, it was examined how accurate the values of the estimated muscle jacobian are if computed by numeric derivation from the previously trained neural networks. The

error values for the three networks are listed in Table 5.2. As before, the normalized mean squared error and the average absolute error are given for each output dimension. The unit of the mean error is meter/radian and therefore this value does not have such an illustrating meaning as before for the direct mapping.

	No Noise		Low Noise		High Noise	
	nMSE	$\overline{\text{error}}$	nMSE	$\overline{\text{error}}$	nMSE	$\overline{\text{error}}$
Biceps	2.46e-5	2.16e-5	1.58e-2	7.03e-4	2.68e-2	1.02e-3
Brachialis	1.45e-5	1.80e-5	1.76e-2	7.05e-4	1.88e-2	7.40e-4
Triceps	9.85e-6	2.51e-5	2.78e-2	1.39e-3	2.77e-2	1.36e-3
Average	1.63e-5	2.16e-5	2.04e-2	9.31e-4	2.44e-2	1.04e-3

Table 5.2: The errors for the best neural networks with respect to the computation of the muscle jacobian. The results are grouped according to the addition of noise to the training data of the networks. The normalized mean square error and the mean absolute error is given separately for each muscle and in average.

The evaluation for the muscle jacobian reflects the results for the direct mapping. If the network is trained on the original data without additive noise the muscle jacobian provides perfect results. In the other two cases, the error values are significantly higher but the difference between low and high noise is only small.

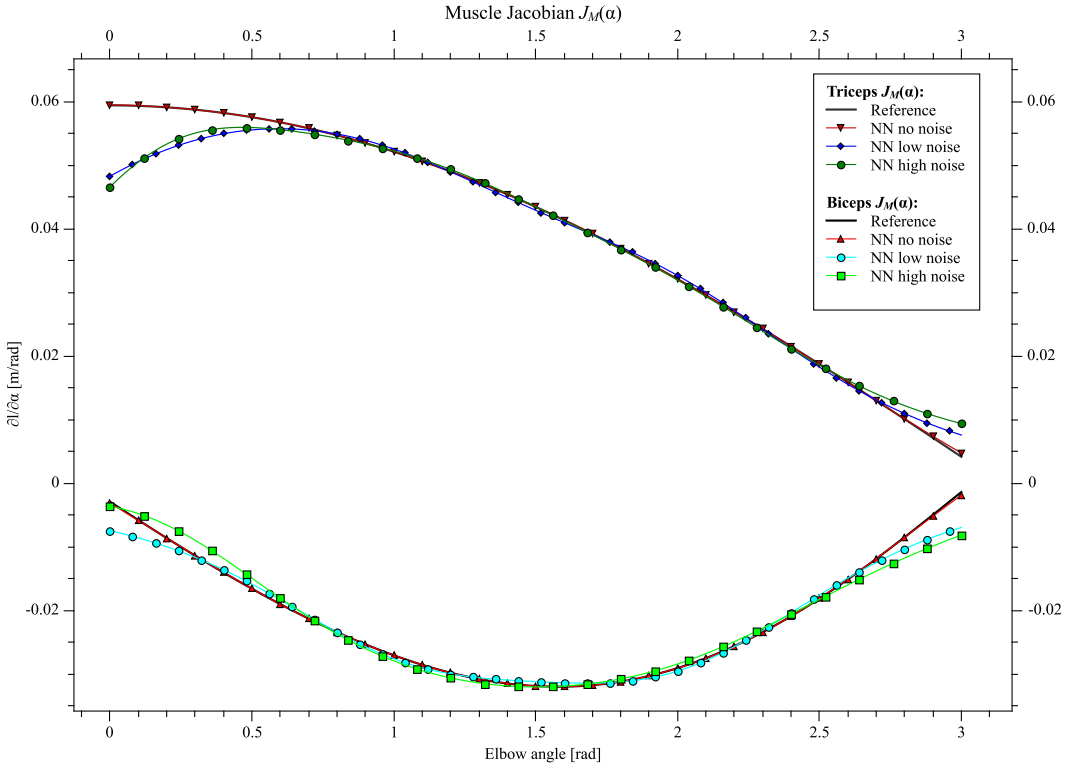


Figure 5.3: The estimated muscle jacobian for the Biceps and Triceps muscle from the neural networks. The reference values were obtained using the method described in Section 5.1.2.

In Figure 5.3 the reference values of the muscle jacobian are plotted for the two muscles Biceps and Triceps. The muscle jacobian retrieved from the neural network trained on the original data leads to almost the same graphs. While the outputs of the other two

networks show a good coincidence in the middle of the angular range, there are more distinct deviations in the marginal regions.

A single computation of the muscle jacobian using neural networks needs 0.0031 ms on average. Although the computation of the muscle Jacobian needs six evaluations of a network, the runtime of this implementation is only slightly more than twice as high as for the direct mapping.

5.2.3 Locally Weighted Projection Regression

The same datasets as for neural networks were used to train three LWPR models. Instead of normalizing the data beforehand the models were provided with the range of possible input and output values and all scaling was left to the algorithm. Meta learning was enabled during the training process to allow an automated adaption of the learning rate, which was initially set to $\alpha = 50$. After some experimentation the initial distance metric of the gaussian kernel was set to $D_{init} = \text{diag}(100)$ and the threshold for the creation of new receptive fields to $w_{gen} = 0.3$. This resulted in a coverage of the input space with about 10 receptive fields. Due to the size of the training set, the best results were achieved if the data was presented at least ten times in random order to the algorithm. Unlike with neural networks the training process for each LWPR model has to be done only once as there is no random initialization of the models. Although the order of presentation should be randomized if the same samples are shown multiple times to the same LWPR model, the actual order had almost no effect on the results.

After some training with the original data and various parameter settings it became evident that the resulting LWPR models did not provide such accurate outputs as the neural networks. As a first measure of improvement the generation of additional input features was examined. From the analytic analysis in Section 5.2.1 it is known that the muscle lengths are a function of sine and cosine of the elbow angle. Much better result could be achieved after these were added as further input features. The difference between the LWPR models trained on the original data with and without generation of features can be seen in Figure 5.4. Due to this improvements, the appendage of sine and cosine as additional input dimensions was implemented as a transparently performed step in the used software component and applied in all further experiments.

5.2.3.1 Results for the Direct Mapping

As for the neural networks, Table 5.3 shows the errors of the LWPR models trained on the data without, with low and with high noise. In all three cases, the generation of additional input features as described above was enabled. Again, the normalized mean squared error as well as the mean absolute error is given for each muscle separately.

	No Noise		Low Noise		High Noise	
	nMSE	$\overline{\text{error}}$ [mm]	nMSE	$\overline{\text{error}}$ [mm]	nMSE	$\overline{\text{error}}$ [mm]
Biceps	1.96e-5	6.42e-2	3.29e-5	8.39e-2	1.54e-4	2.03e-1
Brachialis	1.48e-5	5.76e-2	2.46e-5	8.19e-2	1.24e-4	1.77e-1
Triceps	8.85e-6	8.60e-2	1.29e-5	1.12e-1	1.08e-4	2.67e-1
Average	1.44e-5	6.93e-2	2.35e-5	9.28e-2	1.28e-4	2.16e-1

Table 5.3: The error values for the LWPR models trained on the original training data and with addition of low and high noise. The generation of additional input features was enabled for all cases. The normalized mean square error and the mean absolute error is given separately for each muscle and in average.

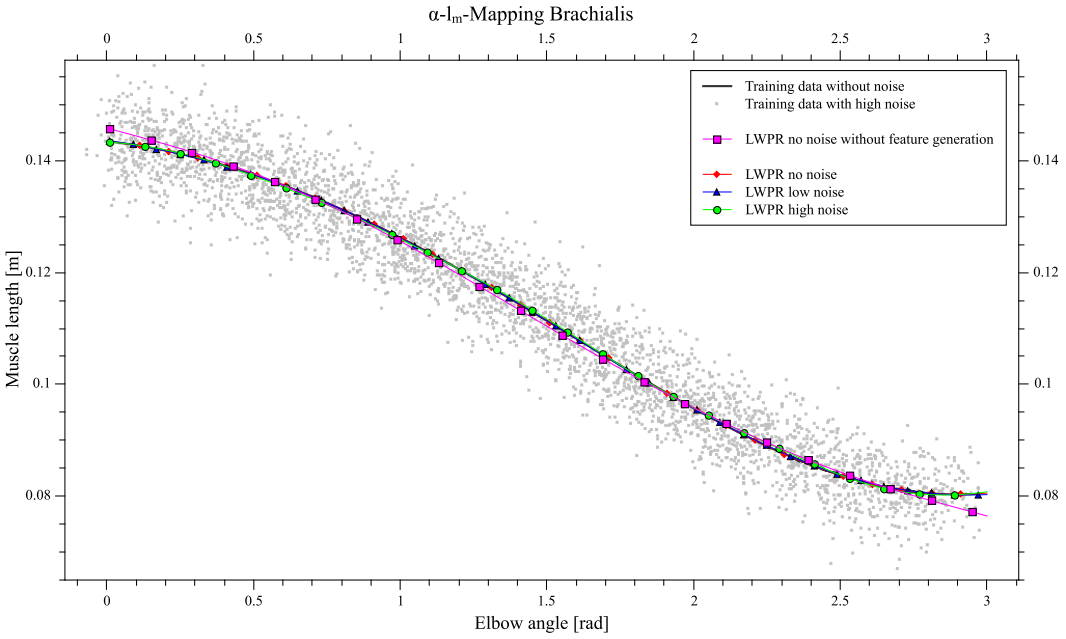


Figure 5.4: The training data of the Brachialis muscle without and with high noise overlaid with the outputs from the LWPR models. Furthermore, the graph of the LWPR model trained on the original data without generation of additional input features is shown.

It can be seen that the LWPR model trained on the original data has very low approximation errors. There is almost no difference to the training on data with addition of low noise and in the case of high noise the errors are only slightly higher. While there were small deviations in the graphs for neural networks trained on noisy data, Figure 5.4 shows an excellent coincidence of the outputs of all LWPR models with the target.

A single evaluation of a LWPR model took in average 0.0037 ms for the used implementation.

5.2.3.2 Results for the Muscle Jacobian

Finally, Table 5.4 lists the estimation errors of the muscle jacobians which were computed from the LWPR models. As before, the normalized mean squared error and the average absolute error are given for each output dimension.

	No Noise		Low Noise		High Noise	
	nMSE	$\overline{\text{error}}$	nMSE	$\overline{\text{error}}$	nMSE	$\overline{\text{error}}$
Biceps	5.34e-6	1.03e-5	1.71e-3	2.50e-4	2.93e-2	9.78e-4
Brachialis	7.89e-6	1.55e-5	2.14e-3	3.14e-4	2.87e-2	1.30e-3
Triceps	2.11e-6	1.39e-5	1.83e-3	3.68e-4	1.70e-2	1.10e-3
Average	5.11e-6	1.32e-5	1.89e-3	3.10e-4	2.50e-2	1.13e-3

Table 5.4: The errors for the best LWPR models with respect to the computation of the muscle jacobian. The results are grouped according to the addition of noise to the training data of the models. The normalized mean square error and the mean absolute error is given separately for each muscle and in average.

For the LWPR model trained on the original data without additive noise the muscle jacobian is estimated almost precisely. As can be seen in Figure 5.5, the accuracy decreases

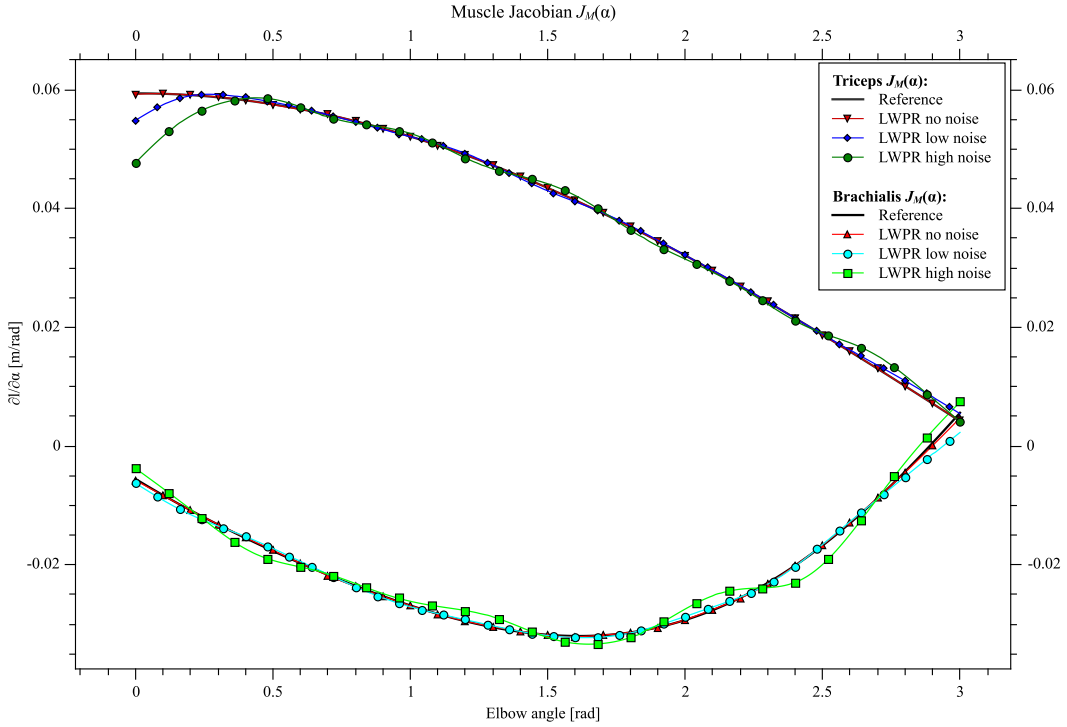


Figure 5.5: The reference values of the muscle jacobian for the Brachialis and Triceps muscle overlaid with the estimation retrieved from the LWPR models.

with increasing noise. This is especially evident in the lower end of the angular range for the Triceps muscle in the case of high noise.

Using the LWPR models, a single computation of the muscle jacobian needs 0.0077 ms on average with the specific implementation and hardware.

5.2.4 Discussion

This section showed that both neural networks and locally weighted projection regression achieved very good results for the simplified test rig consisting only of the elbow joint and three muscles. For the LWPR models apparently the generation of additional non-linear features, which occur in the functional relationship that shall be learned, leads to a significant improvement. LWPR seems to cope slightly better with noisy data than neural networks.

Both variants were also tested in the computed torque controller described in Chapter 2 and it could be shown that the concept works. The angle of the elbow joint could be controlled with high precision in the simulation using this approach. Although the LWPR implementation is only half as fast as the one of the neural networks, the runtime for the computation of the muscle jacobian is more than sufficient for the stable integration in a control loop.

6. Experimental Results II: The Complete Test Rig

After the successful experiments with the reduced test rig it was tried to transfer the gained experience and apply this method to the full robot arm. As described in Chapter 2.2 the complete test rig consists of a shoulder and an elbow joint and is actuated by 11 muscles. The function approximation task becomes distinctly more difficult in this case. In the former experiments the entire input space, i.e. the range of elbow angles, had been sampled in steps of 0.005 radians. This is not even nearly possible for the complete robot arm with four DoF, which can be seen in the following rough calculation. Assuming that the three DoF of the shoulder are each limited to the range $[-60°, 60°]$ and the angle of the elbow is as before limited to $[0°, 180°]$. Then a sampling of the input space in steps of one degree would already lead to approximately $3 \cdot 10^8$ data points. The gathering of training data and its processing in the learning algorithms is not possible with such many samples. Apparently, it is necessary to rely on much less training data than this calculation suggests and leave much larger parts of the input space for the machine learning methods to interpolate. It becomes all the more important that the learning algorithms are provided with the right training data, which means that it is drawn from the entire input space and is distributed as evenly as possible.

The collection of training data for the following experiments was done in two steps. Initially, the DoF of the shoulder were sampled in the range $[-1.0, 1.0]$ and the elbow angle in the range $[0.0, 3.0]$ in steps of 0.15 radians. This was done by setting joint angles as described in Chapter 4.4.1 in order to achieve a basic coverage of the input space. The data set was eventually extended by generation of random movements (see Chapter 4.4.2) to an overall number of 103055 samples. Its 5-dimensional input vectors consist of the four quaternion parameters for the shoulder joint and the angle of the elbow. The lengths of the 11 muscles—again in meters—are collected in the output vectors. All following experiments involved again the training of the respective machine learning methods on the original training data and with additive low and high noise. Low noise stands for zero-mean Gaussian noise with a standard deviation of one degree for the joint angles and one mm for the muscle lengths and high noise a standard deviation of two degrees for the joint angles and five mm for the muscle lengths.

As before, reference data for the evaluation of the muscle jacobian was collected using the method described in Chapter 5.1.2. The mere extension from one to four DoF already

makes this a very time-consuming task. The collection of the 11272 reference values took several days of real time simulation.

6.1 Analytic Examination

Analogously to the simplified model of the elbow joint, the relationship between the angle of the shoulder joint and the lengths of the muscles spanned over it was examined analytically before the running of the experiments. For this purpose, this subsection derives the length of a muscle attached to the upper arm in dependence of the angle of the spherical shoulder joint. The setup and all required variables are shown in Figure 6.1. The pose of the upper arm with reference to the torso is given as unit quaternion $\boldsymbol{\epsilon} = (\epsilon_0, \epsilon_1, \epsilon_2, \epsilon_3)^\mathsf{T}$.

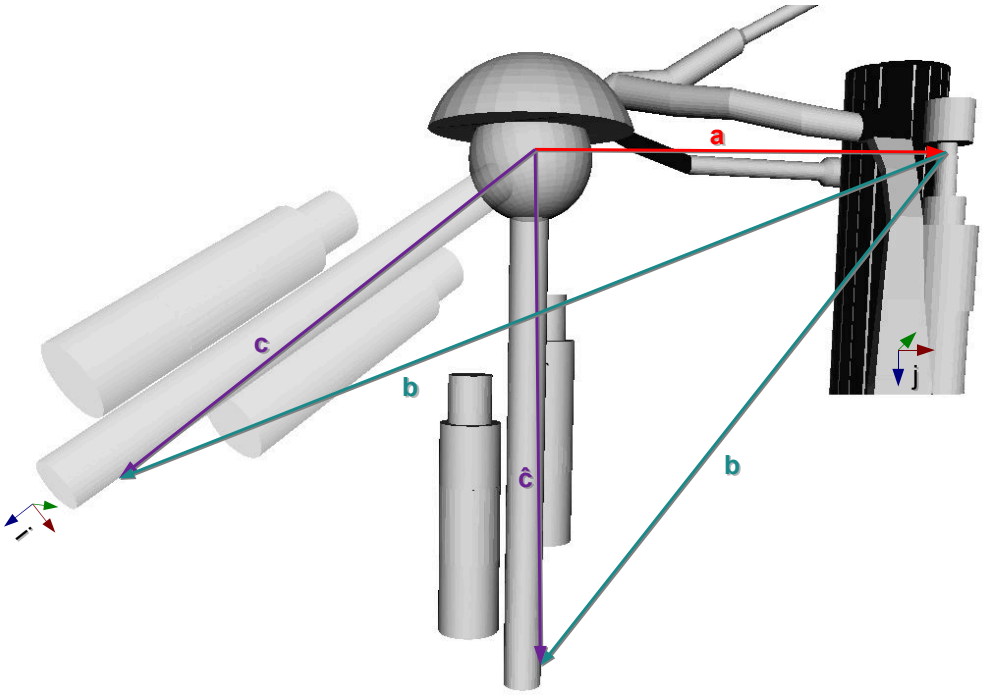


Figure 6.1: This figure indicates all vectors used in the analytic derivation of the length of one muscle spanned over the shoulder joint in dependence of the quaternion describing the orientation of the upper arm.

According to [31] the quaternion $\boldsymbol{\epsilon}$ can be transformed in an equivalent rotation matrix ${}^j\boldsymbol{R}_i$ which describes the orientation of the upper arm specified by the coordinate frame i with respect to the coordinate frame j of the torso:

$$ {}^j\boldsymbol{R}_i = \begin{bmatrix} 1-2(\epsilon_2^2+\epsilon_3^2) & 2(\epsilon_1\epsilon_2-\epsilon_0\epsilon_3) & 2(\epsilon_1\epsilon_3+\epsilon_0\epsilon_2) \\ 2(\epsilon_1\epsilon_2+\epsilon_0\epsilon_3) & 1-2(\epsilon_1^2+\epsilon_3^2) & 2(\epsilon_2\epsilon_3-\epsilon_0\epsilon_1) \\ 2(\epsilon_1\epsilon_3-\epsilon_0\epsilon_2) & 2(\epsilon_2\epsilon_3+\epsilon_0\epsilon_1) & 1-2(\epsilon_1^2+\epsilon_2^2) \end{bmatrix} \tag{6.1} $$

The position vector $\boldsymbol{a} = (a_x, a_y, a_z)^\mathsf{T}$ points from the center of the spherical joint to the first attachment point of muscle m, while vector $\boldsymbol{c} = (c_x, c_y, c_z)^\mathsf{T}$ points from the same origin to the second attachment point. While $\boldsymbol{a}$ does never change, $\boldsymbol{c}$ corresponds to the

current pose of the upper arm. Vector $\hat{\boldsymbol{c}}$ indicates the zero position of the arm in which the coodinate frame i and j are the same. Using the rotation matrix from Eq. (6.1) the altered pose of the arm can be expressed as $\boldsymbol{c} = {}^{j}\boldsymbol{R}_i\,\hat{\boldsymbol{c}}$. Being defined between the two attachment points, vector $\boldsymbol{b} = \boldsymbol{c} - \boldsymbol{a}$ always coincides with the path of muscle m. Therefore, the length of the muscle can be computed as length of vector $\boldsymbol{b}$:

$$l_m = ||\boldsymbol{b}|| = ||{}^{j}\boldsymbol{R}_i\,\hat{\boldsymbol{c}} - \boldsymbol{a}|| \tag{6.2}$$

This can be further evaluated to gain the muscle length l_m in dependence of the parameters of quaternion $\boldsymbol{\epsilon}$ and the fixed attachment points of the muscle which are described by $\boldsymbol{a}$ and $\hat{\boldsymbol{c}}$:

$$\begin{aligned} l_m^2 = \quad & \big(\hat{c}_x(1-2(\epsilon_3^2+\epsilon_2^2)) + 2\hat{c}_z(\epsilon_1\epsilon_3+\epsilon_0\epsilon_2) + 2\hat{c}_y(\epsilon_1\epsilon_2-\epsilon_0\epsilon_3) - a_x\big)^2 \\ & + \big(\hat{c}_y(1-2(\epsilon_3^2+\epsilon_1^2)) + 2\hat{c}_z(\epsilon_2\epsilon_3-\epsilon_0\epsilon_1) + 2\hat{c}_x(\epsilon_0\epsilon_3+\epsilon_1\epsilon_2) - a_y\big)^2 \\ & + \big(2\hat{c}_y(\epsilon_2\epsilon_3+\epsilon_0\epsilon_1) + 2\hat{c}_x(\epsilon_1\epsilon_3-\epsilon_0\epsilon_2) + \hat{c}_z(1-2(\epsilon_2^2+\epsilon_1^2)) - a_z\big)^2 \end{aligned} \tag{6.3}$$

6.2 Neural Networks

The experiments with neural networks are divided into the ones with single neural networks and the ones which decomposed the function approximation task into multiple subtasks. For all cases the same procedures and parameters as with the network training before in the simplified case were adopted. This means that all networks are multilayer perceptrons with a single hidden layer. The hidden neurons are equipped with the tanh and the output units with a linear activation function. For every experiment the training was repeated several times—because of the random weight initialization in the range $[-0.77, 0.77]$—using the incremental backpropagation algorithm with a learning rate of $\eta = 0.01$. All input and output values were scaled to the interval $[-1, 1]$ and the error on the validation set, which this time comprised 20% of the gathered training data, was observed during the training process in order to avoid the effects of overfitting.

6.2.1 Single Neural Networks

In the first instance, some experiments were carried out with single neural networks for the approximation task of the function of all muscle lengths with respect to all joint angles. Therefore, these multilayer perceptrons have five input neurons for the elbow angle and the quaternion parameters of the shoulder joint and 11 output neurons for the muscle lengths. After some trials with different network sizes it was found that the best results could be achieved with 60 neurons in the hidden layer. As before, separate networks were trained on the original training data without noise and again with additive low and high noise.

6.2.1.1 Results for the Direct Mapping

The best results for these network configurations are summarized in Table 6.1. The normalized mean squared error and the absolute mean error with respect to the test data gathered with the simulator are given for each output dimension. As before, the results are grouped according to the training of the networks without, with low and with high noise.

The inspection of the numbers shows that the lengths of some muscles are approximated slightly better than the ones of others. The set of muscles affected by this disparity is

	No Noise		Low Noise		High Noise	
	nMSE	$\overline{\text{error}}$ [mm]	nMSE	$\overline{\text{error}}$ [mm]	nMSE	$\overline{\text{error}}$ [mm]
Anterior Deltoid	8.18e-5	1.60e-1	7.69e-5	1.52e-1	8.13e-5	1.59e-1
Biceps	4.65e-5	1.75e-1	3.91e-5	1.63e-1	4.39e-5	1.64e-1
Brachialis	1.38e-5	5.52e-2	1.33e-5	5.27e-2	1.36e-5	5.43e-2
Infra	4.36e-5	5.41e-2	4.51e-5	6.13e-2	4.71e-5	5.79e-2
Lateral Deltoid	7.27e-5	1.37e-1	6.00e-5	1.27e-1	7.29e-5	1.38e-1
Pectoralis	4.97e-5	1.05e-1	4.76e-5	1.01e-1	4.69e-5	1.04e-1
Posterior Deltoid	7.07e-5	1.15e-1	5.73e-5	1.03e-1	5.29e-5	9.84e-2
Supra	2.99e-5	4.43e-2	3.00e-5	4.23e-2	2.84e-5	4.33e-2
Teres Major	3.21e-5	8.63e-2	3.58e-5	8.83e-2	3.37e-5	8.00e-2
Teres Minor	1.41e-5	3.36e-2	1.41e-5	3.52e-2	1.72e-5	3.89e-2
Triceps	3.60e-6	6.68e-2	2.62e-6	5.21e-2	2.71e-6	5.45e-2
Average	4.17e-5	9.38e-2	3.83e-5	8.90e-2	4.00e-5	9.02e-2

Table 6.1: The error values for the best single neural networks trained on the original training data and with additive low and high noise. The normalized mean square error and the mean absolute error is given separately for each muscle and in average.

apparently always the same and does not depend on the network initialization or the addition of noise to the training data. In any case, the errors for all muscles are relatively low and the average deviation never exceeds 0.2 millimeters. The addition of noise to the training data has in this case no effects on the network performance. In fact, the network trained on the high noise data has in some cases even lower error values than the one trained on the original training data without noise. However, these effects are very small and it is assumed that they can be attributed to the random initialization of the network weights. The accuracy of approximation even in spite of noisy training data is also illustrated in Figure 6.2a and Figure 6.3a. These graphs show the approximated length of a single exemplary muscle plotted over two shoulder angles. The remaining other shoulder angle as well as the elbow angle are set to a fixed value. In order to facilitate the illustration the shoulder angles were converted from quaternions to XYZ-Euler-Angles. Obviously there is no difference visible between the output of the neural networks and the reference values as given by the training data.

The average runtime according to the specifically used implementation and hardware, as described in Chapter 5.1.3, was measured as 0.0075 ms for a single evaluation of a neural network.

6.2.1.2 Results for the Muscle Jacobian

It was also evaluated how accurate the values of the estimated muscle jacobian are if obtained using the previously presented single neural networks. The normalized mean squared error and the average absolute error values for the three networks are presented in Table 6.2.

These numbers reflect the results seen for the direct mapping, there is no difference apparent in the results for the networks trained on data with and without noise. Typically, after the numerical derivation the error values are higher than for the direct mapping. Although the task is significantly more difficult compared to the simplified test rig, the results differ not that much, at least compared to the previous case of highly noisy training data. The high numbers of variables makes an illustration of the muscle jacobian for the complete test rig very difficult. It is at best possible to give some snapshots for single muscles by plotting their partial derivatives of the muscle jacobian $\boldsymbol{J}_m(\boldsymbol{q})$ over a single

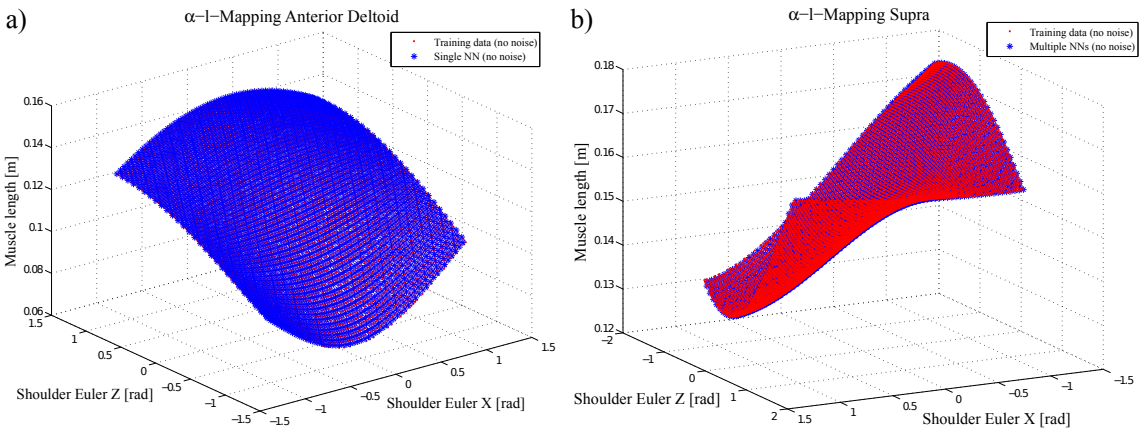


Figure 6.2: Single muscle lengths plotted over the x- and z-component of the XYZ-Euler-Angles of the shoulder joint. The y-component was set to the fixed value of 0.1 rad and the elbow angle to 0.5 rad. Figure a) shows the length of the Anterior Deltoid muscle using the single neural network, while Figure b) shows the length of the Supra muscle using a decomposition in three neural networks. All networks were trained on the original training data without noise.

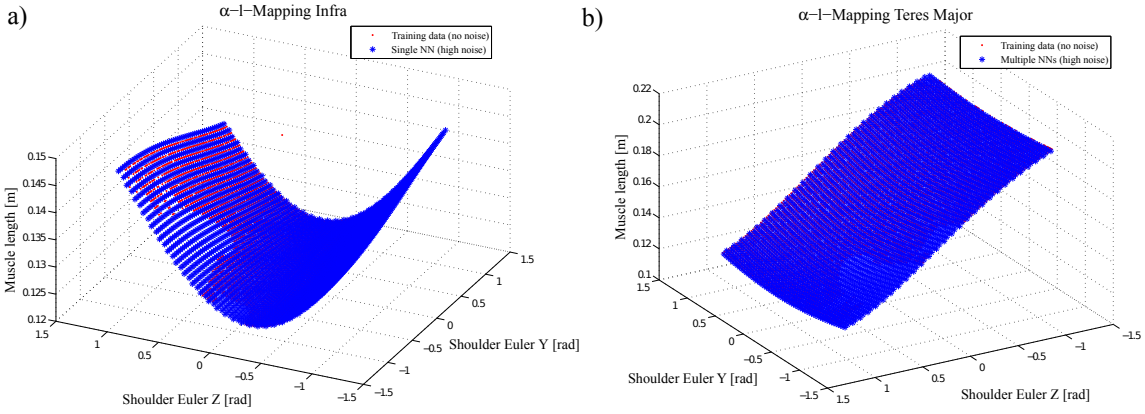


Figure 6.3: Another example of approximated muscle lengths, plotted over the y- and z-component of the shoulder angles this time. As the x-component was set to 1.1 rad this in a sense shows a marginal area of the movement space of the robot arm. The elbow angle was once more set to 0.5 rad. Figure a) shows the length of the Infra muscle using a single neural network trained on the highly noisy training data and Figure b) presents the Teres Major muscle using a decomposition in three neural networks trained on the same data. The gaps in the graph of the training data indicate joint configurations in which the robot arm would collide with the rest of the body and which therefore can not be taken on. Nevertheless, the neural networks have covered these regions nicely as well.

input dimension. Figure 6.4a shows the partial derivatives of the Teres Minor muscle over each shoulder angle, respectively. The other input dimensions are set to fixed values. It can be seen that the muscle jacobian is estimated very accurately in the shown region for this muscle. Only the partial derivative with respect to the z-dimension of the shoulder joint in the second row of the figure shows no resemblance at all to the reference values.

The computation of the muscle jacobian for a single joint configuration of the robot using these neural networks takes on average 0.075 ms. This number is very consistent with

	No Noise		Low Noise		High Noise	
	nMSE	$\overline{\text{error}}$	nMSE	$\overline{\text{error}}$	nMSE	$\overline{\text{error}}$
Anterior Deltoid	1.15e+0	3.63e-3	1.01e+0	3.64e-3	1.02e+0	3.63e-3
Biceps	2.45e-1	3.38e-3	2.45e-1	3.36e-3	2.44e-1	3.34e-3
Brachialis	1.42e+0	5.54e-5	1.20e+0	5.54e-5	1.28e+0	5.09e-5
Infra	5.88e-1	4.39e-3	5.59e-1	4.39e-3	5.81e-1	4.39e-3
Lateral Deltoid	1.33e+0	3.75e-3	8.76e-1	3.73e-3	1.13e+0	3.74e-3
Pectoralis	7.02e-1	4.52e-3	6.30e-1	4.51e-3	7.66e-1	4.52e-3
Posterior Deltoid	8.25e-1	3.82e-3	8.89e-1	3.80e-3	6.65e-1	3.78e-3
Supra	5.42e-1	3.64e-3	6.08e-1	3.63e-3	6.74e-1	3.64e-3
Teres Major	6.89e-1	3.96e-3	5.51e-1	3.96e-3	5.51e-1	3.96e-3
Teres Minor	5.65e-1	3.93e-3	5.96e-1	3.94e-3	6.30e-1	3.94e-3
Triceps	9.99e-1	1.24e-4	1.00e+0	1.01e-4	1.03e+0	1.07e-4
Average	8.23e-1	3.20e-3	7.43e-1	3.19e-3	7.79e-1	3.19e-3

Table 6.2: The errors for the best single neural networks with respect to the computation of the muscle jacobian. The normalized mean square error and the mean absolute error is given separately for each muscle and in average.

the runtime for the direct mapping as in this case the computation of the muscle jacobian requires ten evaluations of the neural network.

6.2.2 Decomposition in Multiple Neural Networks

As an alternative to using a single network the decomposition of the function approximation task in three subtasks was examined. This decomposition was done according to the set of joint parameters the particular muscle lengths depend on. A first network was used for the muscles spanned over the elbow joint, namely Brachialis and Triceps. It has only the elbow angle as input and similarly to the case of the simplified test rig gets along with five neurons in the hidden layer. The Biceps is the only muscle spanned over both shoulder and elbow joint and is therefore represented by a neural network of its own. This network with 20 neurons in the hidden layer takes the quaternion parameters of the shoulder joint and the angle of the elbow as an input. Although it covers only one muscle it has turned out that a few more hidden units than for the elbow network are needed since the functional relationship of the Biceps with its dependence on five inputs is more complex. A third MLP of 30 hidden neurons incorporates the remaining eight muscles spanned over the shoulder joint and consequently has an input neuron for each quaternion parameter. The training of each network was performed separately and repeated several times in order to find the best results.

6.2.2.1 Results for the Direct Mapping

The error values achieved with the best networks of this decomposition are presented in Table 6.3.

Apparently in this case as well, the addition of noise to the training data does not have any observable effects on the performance of the networks. One of the reasons to introduce a decomposition into multiple function approximators was to relief the learning algorithm from the task of determining which muscles (outputs) depend on which joints (inputs). If such a manual input allocation is not performed the function approximator might model a relationship which in fact does not exist. In order to find out if this is the case it should be examined whether the output for a muscle changes when only the angle of a joint not spanned by this muscle is varied. For instance, it was checked whether the outputs of the Brachialis or Triceps muscle lengths change for variations in the shoulder angles.

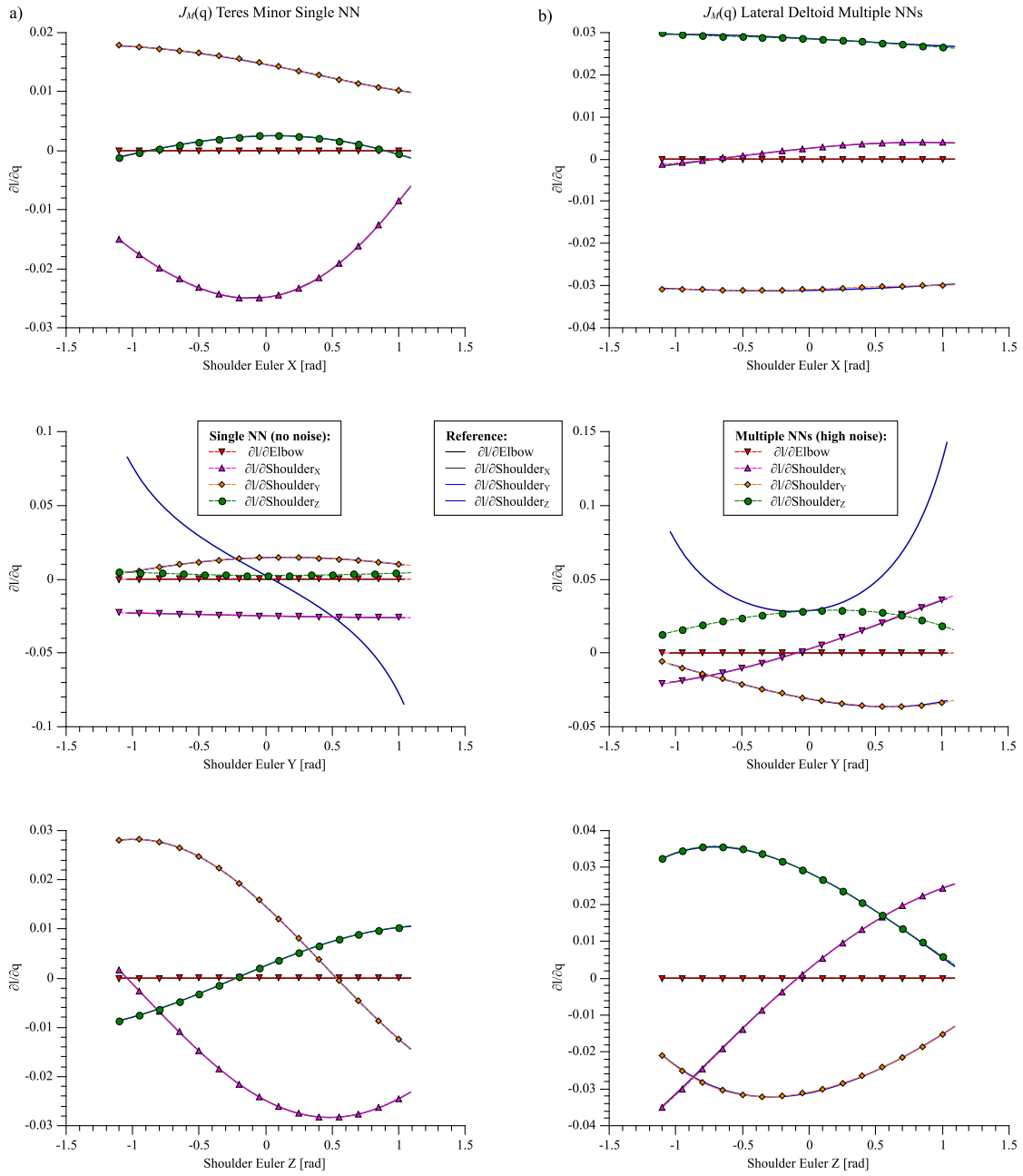


Figure 6.4: Some snapshots of the estimated muscle jacobian, plotting the partial derivatives of one muscle over single input dimensions. The three plots for each muscle in one column have a different angle of the shoulder joint as abscissa. All input dimensions not shown have been set to zero. Figure a) shows the segment of the muscle jacobian for the Teres Minor muscle as estimated by the single neural network trained on the original training data without noise. In Figure b) the same is illustrated for the Lateral Deltoid muscle using the decomposition in three neural networks trained on highly noisy data.

One way to do so is to sample the entire input space of the approximator and plot the corresponding muscle outputs over the elbow angle, projecting all inputs to the dimension of the elbow angle. This has been done in Figure 6.5 for the single neural network trained on the original training data. It is obvious that the muscle lengths show a variation for a certain elbow angle. The figure also shows the same plot for the previously presented

	No Noise		Low Noise		High Noise	
	nMSE	$\overline{\text{error}}$ [mm]	nMSE	$\overline{\text{error}}$ [mm]	nMSE	$\overline{\text{error}}$ [mm]
Anterior Deltoid	9.30e-5	1.73e-1	1.09e-4	1.65e-1	9.11e-5	1.63e-1
Biceps	3.76e-5	1.57e-1	3.37e-5	1.49e-1	4.07e-5	1.58e-1
Brachialis	1.41e-5	5.96e-2	1.35e-5	5.23e-2	1.34e-5	5.13e-2
Infra	5.07e-5	6.18e-2	5.78e-5	6.51e-2	5.27e-5	6.15e-2
Lateral Deltoid	9.03e-5	1.51e-1	8.20e-5	1.40e-1	1.51e-4	1.67e-1
Pectoralis	5.75e-5	1.08e-1	6.77e-5	1.10e-1	5.90e-5	1.16e-1
Posterior Deltoid	1.02e-4	1.31e-1	6.37e-5	1.13e-1	9.87e-5	1.30e-1
Supra	7.96e-5	6.76e-2	4.42e-5	5.43e-2	3.80e-5	5.29e-2
Teres Major	5.44e-5	1.07e-1	4.07e-5	9.05e-2	5.99e-5	1.06e-1
Teres Minor	3.13e-5	5.30e-2	2.52e-5	4.69e-2	1.84e-5	4.14e-2
Triceps	2.44e-6	5.26e-2	2.89e-6	6.05e-2	2.97e-6	5.97e-2
Average	5.57e-5	1.02e-1	4.91e-5	9.51e-2	5.69e-5	1.01e-1

Table 6.3: The error values for a decomposition using three neural networks trained on the original training data and with additive low and high noise. The normalized mean square error and the mean absolute error is given separately for each muscle and in average over all muscles. For details about the decomposition see text.

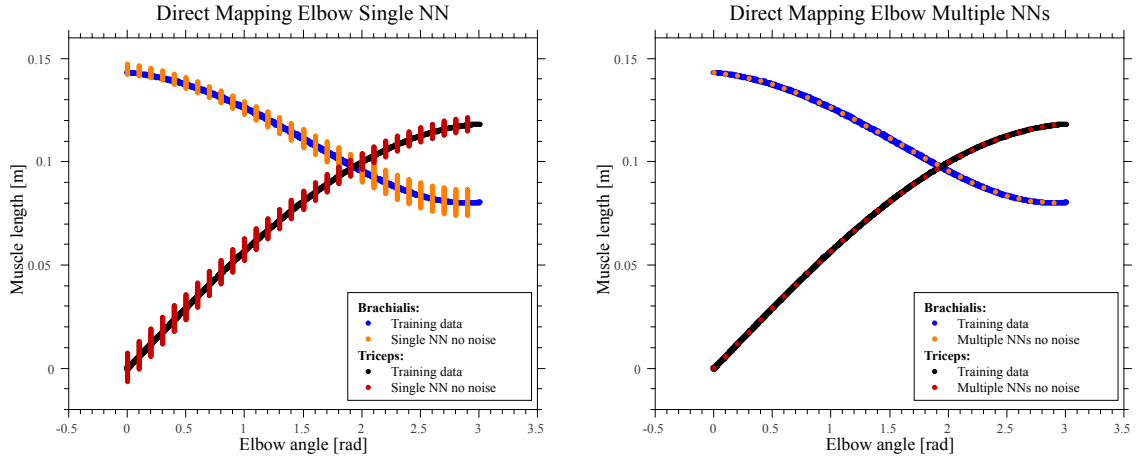


Figure 6.5: The output for the Brachialis and Triceps muscle of the single neural network (left) and the decomposition in three neural networks (right), both trained on the original training data without noise. The input space of the approximators was sampled in steps of 0.1 (which is the reason why the output is not continuous over the entire range of elbow angles) and all the inputs were projected to the dimension of the elbow angle. The muscle lengths of the single network show a distinct variance over a certain elbow angle which indicates a dependence on the shoulder angles.

decomposition in multiple networks. Naturally, the described effect cannot occur for the decomposed case and the reference curves as given by the training data are fitted exactly.

Apart from that, the differences in the errors compared to the single networks are very small. If anything, the performance of the single networks seems slightly better. It is speculated that this can be attributed to a better exploitation of similarities in the functional relationship between muscles in the case of the single networks. Figure 6.2b and Figure 6.3b show exemplarily the approximated muscle lengths of two muscles involved in shoulder movements. The outputs of both networks trained on data with and without noise show that there is no difference to the reference values visible to the naked eye.

A single evaluation of the direct mapping as decomposed in the described three neural networks takes in average 0.0093 ms. As a matter of fact, the application of multiple

networks is more efficient in its consumption of network resources as it overall uses about half as many network weights as the single neural network. Nevertheless, the evaluation of three smaller networks as compared to just one large network takes a slightly longer.

6.2.2.2 Results for the Muscle Jacobian

The estimation errors for the muscle jacobian obtained from the three neural networks are presented in Table 6.4. Analogically to the direct mapping, these results are very similar to those of the single neural networks.

	No Noise		Low Noise		High Noise	
	nMSE	$\overline{\text{error}}$	nMSE	$\overline{\text{error}}$	nMSE	$\overline{\text{error}}$
Anterior Deltoid	6.18e-1	3.63e-3	6.17e-1	3.64e-3	6.18e-1	3.65e-3
Biceps	2.44e-1	3.33e-3	2.45e-1	3.32e-3	2.44e-1	3.32e-3
Brachialis	7.90e-1	1.06e-4	7.90e-1	3.44e-5	7.90e-1	3.44e-5
Infra	5.05e-1	4.40e-3	5.05e-1	4.40e-3	5.05e-1	4.41e-3
Lateral Deltoid	5.72e-1	3.76e-3	5.72e-1	3.73e-3	5.73e-1	3.76e-3
Pectoralis	5.39e-1	4.53e-3	5.39e-1	4.53e-3	5.39e-1	4.53e-3
Posterior Deltoid	5.26e-1	3.84e-3	5.24e-1	3.80e-3	5.26e-1	3.84e-3
Supra	4.84e-1	3.65e-3	4.84e-1	3.66e-3	4.84e-1	3.65e-3
Teres Major	4.83e-1	3.99e-3	4.83e-1	3.97e-3	4.84e-1	3.99e-3
Teres Minor	5.01e-1	3.95e-3	5.02e-1	3.94e-3	5.02e-1	3.95e-3
Triceps	8.46e-1	1.06e-4	8.46e-1	9.87e-5	8.46e-1	9.81e-5
Average	5.55e-1	3.21e-3	5.55e-1	3.19e-3	5.56e-1	3.20e-3

Table 6.4: The errors for a decomposition in the best three neural networks with respect to the computation of the muscle jacobian. The normalized mean square error and the mean absolute error is given separately for each muscle and in average.

The snapshots of the estimated muscle jacobian for the Lateral Deltoid muscle in Figure 6.4b show a similarly good overlap with the reference data as already seen for the case of single neural networks. It is interesting that just as with the single network the figure in the second row reveals a great error in the partial derivative with respect to the z-dimension of the shoulder joint. Unfortunately, no descriptive explanation for this effect could be found. The reference values were obtained using the method described in Chapter 5.1.2. It is unclear whether the different observed courses of the curves should be attributed to problems of this method, i.e. flawed reference values, to an incorrect approximation of the machine learning models, or even both.

On a side note, all the plots in Figure 6.4 illustrate the independence of the two shown muscles (Teres Minor and Lateral Deltoid) from the elbow joint since the partial derivatives with respect to the elbow angle are constantly zero. This fact has been modelled by the neural networks very well. Apparently, the learning algorithm has recognized the irrelevant inputs in case of the single networks better than in the case of the elbow joint.

The computation of a muscle jacobian using the decomposition takes on average 0.093 ms.

6.3 Locally Weighted Projection Regression

The experiments with locally weighted projection regression were structured as with the neural networks. They are also divided into the ones with single LWPR models and the ones which decomposed the function approximation task into multiple subtasks. All LWPR models were provided with the range of possible input and output values in order to avoid manual normalization. As there is no need for a validation set the entire collected data from the simulation could be used for training. The learning rate of the algorithm was initially set to $\alpha = 50$ but was adapted by means of meta learning during the training process.

Extensive trials with different values for the distance metric of the gaussian kernel led to the initialization with $D_{init} = \text{diag}(20)$. The threshold for the creation of new receptive fields was set to $w_{gen} = 0.3$. Usually, if the training set is large enough it is sufficient for the LWPR algorithm to present each sample only once. However, in all the training cases it was observed that the errors were still decreasing after 15 to 20 presentations.

From the results with the simplified test rig it was already known that it is likely to improve the approximation accuracy for locally weighted projection regression by means of further generated input features. The analytic examination of the functional relationship between the angles of the shoulder joint and the length of the muscles spanned over it in Section 6.1 suggests the introduction of the square of the quaternion parameters as new features. As a matter of fact, some minor improvements could be observed after this measure. Since the elbow angle is still present in the complete test rig the generation of sine and cosine as further input features was kept as well.

6.3.1 Single LWPR Model

The first set of experiments used single LWPR models to approximation the entire mapping from joint angles to muscle length. With the parameters described before, this resulted in LWPR models with 11 submodels—one for each output dimension—and each submodel consisted of between 230 and 250 receptive fields.

6.3.2 Results for the Direct Mapping

The error values for the single LWPR models trained on data with different kinds of noise are presented in Table 6.5.

	No Noise		Low Noise		High Noise	
	nMSE	$\overline{\text{error}}$ [mm]	nMSE	$\overline{\text{error}}$ [mm]	nMSE	$\overline{\text{error}}$ [mm]
Anterior Deltoid	1.10e-2	2.00e+0	1.04e-1	2.94e+0	1.04e-1	3.05e+0
Biceps	5.31e-3	1.90e+0	2.55e-1	3.66e+0	8.96e-2	3.03e+0
Brachialis	2.03e-4	3.05e-1	1.97e-2	4.91e-1	1.63e-2	4.89e-1
Infra	1.18e-2	1.15e+0	6.75e-1	2.33e+0	1.80e-1	1.89e+0
Lateral Deltoid	1.29e-2	2.00e+0	3.49e-2	2.26e+0	2.79e-2	2.32e+0
Pectoralis	1.92e-2	2.14e+0	9.34e-1	4.26e+0	2.36e-1	3.16e+0
Posterior Deltoid	2.11e-2	1.84e+0	2.58e-1	2.94e+0	2.01e-1	2.88e+0
Supra	2.08e-2	1.33e+0	2.76e-1	1.79e+0	1.70e-1	1.91e+0
Teres Major	1.56e-2	2.06e+0	4.69e-1	3.37e+0	9.67e-2	2.63e+0
Teres Minor	1.18e-2	1.11e+0	3.68e-1	1.94e+0	2.23e-1	1.92e+0
Triceps	3.72e-5	2.38e-1	7.30e-3	4.59e-1	2.34e-3	4.72e-1
Average	1.18e-2	1.46e+0	3.09e-1	2.40e+0	1.23e-1	2.16e+0

Table 6.5: The error values for the single LWPR models trained on the original training data and with additive low and high noise. The normalized mean square error and the mean absolute error is given separately for each muscle and in average.

The approximation accuracy was for all muscles roughly the same. If noise was added to the training data the errors are slightly higher, but there is no distinct difference between low and high noise. However, compared to the results achieved with neural networks these errors are considerably higher with an average deviation between one and three millimeters. This can also be seen in Figure 6.6a which shows as an example the approximated length of the Lateral Deltoid muscle over two of the shoulder angles. It is obvious that the shape of the curve described by the reference values has been roughly approximated but there are clearly some deviations visible.

The average runtime according to the specifically used implementation and hardware was measured as 3.22 ms for a single evaluation of a LWPR model.

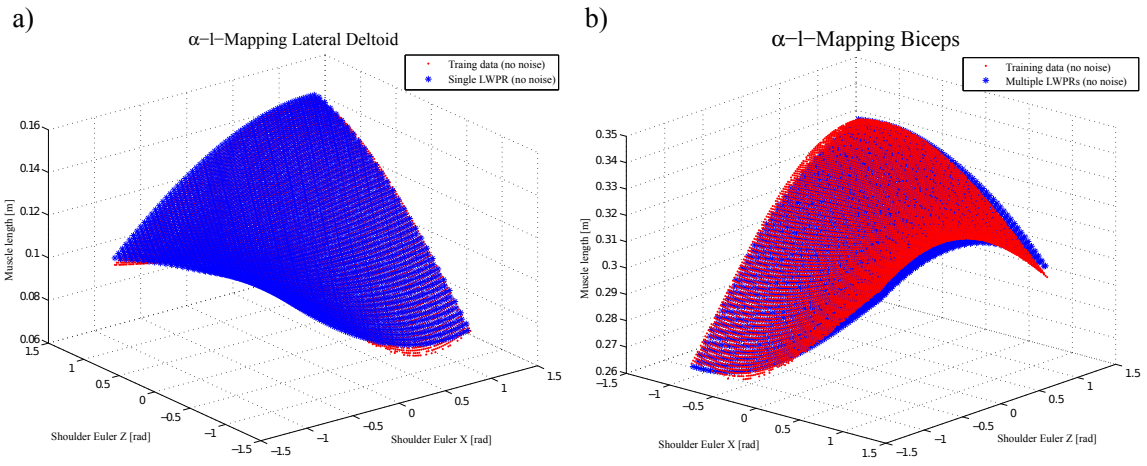


Figure 6.6: The approximated muscle lengths of two exemplary muscles involved in shoulder movements plotted over the x- and z-component of the XYZ-Euler-Angles of the shoulder joint. The y-component was set to the fixed value of 0.1 rad and the elbow angle to 0.5 rad. Figure a) shows the length of the Lateral Deltoid muscle using a single LWPR model, while Figure b) shows the length of the Biceps muscle using a decomposition in three LWPR models. All models were trained on the original training data without noise.

6.3.2.1 Results for the Muscle Jacobian

It is of course interesting how these higher deviations in the direct mapping affects the accuracy of the muscle jacobian. The normalized mean squared error and the average absolute error values for the single LWPR models are presented in Table 6.6.

	No Noise		Low Noise		High Noise	
	nMSE	$\overline{\text{error}}$	nMSE	$\overline{\text{error}}$	nMSE	$\overline{\text{error}}$
Anterior Deltoid	1.33e+2	5.58e-3	1.08e+2	5.59e-3	2.07e+2	5.75e-3
Biceps	3.68e-1	6.38e-3	3.40e-1	5.95e-3	3.65e-1	6.30e-3
Brachialis	1.24e+1	6.07e-4	1.37e+1	6.49e-4	2.42e+1	7.10e-4
Infra	1.46e+2	5.90e-3	1.50e+2	5.99e-3	2.16e+2	6.15e-3
Lateral Deltoid	2.71e+2	5.96e-3	2.62e+2	5.87e-3	3.62e+2	6.15e-3
Pectoralis	2.14e+2	7.28e-3	2.62e+2	7.20e-3	2.89e+2	7.43e-3
Posterior Deltoid	1.99e+2	6.03e-3	2.14e+2	5.97e-3	3.67e+2	6.07e-3
Supra	1.54e+2	5.04e-3	1.68e+2	5.04e-3	2.00e+2	5.32e-3
Teres Major	1.82e+2	6.66e-3	1.71e+2	6.50e-3	2.35e+2	6.72e-3
Teres Minor	1.61e+2	5.47e-3	2.83e+2	5.49e-3	2.43e+2	5.69e-3
Triceps	4.05e+0	5.34e-4	1.06e+1	5.82e-4	2.78e+1	8.02e-4
Average	1.34e+2	5.04e-3	1.49e+2	4.99e-3	1.97e+2	5.19e-3

Table 6.6: The errors for the best LWPR models with respect to the computation of the muscle jacobian. The normalized mean square error and the mean absolute error is given separately for each muscle and in average.

While the mean errors of Table 6.6 are comparable to the values seen for the neural networks, the normalized mean square errors are three orders of magnitude higher. This indicates the different levels of significance of those two error measures. A comparison of the different results according just to the mean error would certainly be fallacious. The snapshots of the estimated muscle jacobian using a single LWPR model in Figure 6.7a also show that there are distinct differences to the reference values. The LWPR model has still recognized relatively well that the illustrated Pectoralis muscle is independent of the elbow joint. It is already a little bit more difficult to correlate the courses of the partial

derivatives with respect to the shoulder angles to the corresponding desired values. As already seen with the neural networks the partial derivative with respect to the z-dimension of the shoulder joint in the second row of the figure shows no resemblance to the reference values.

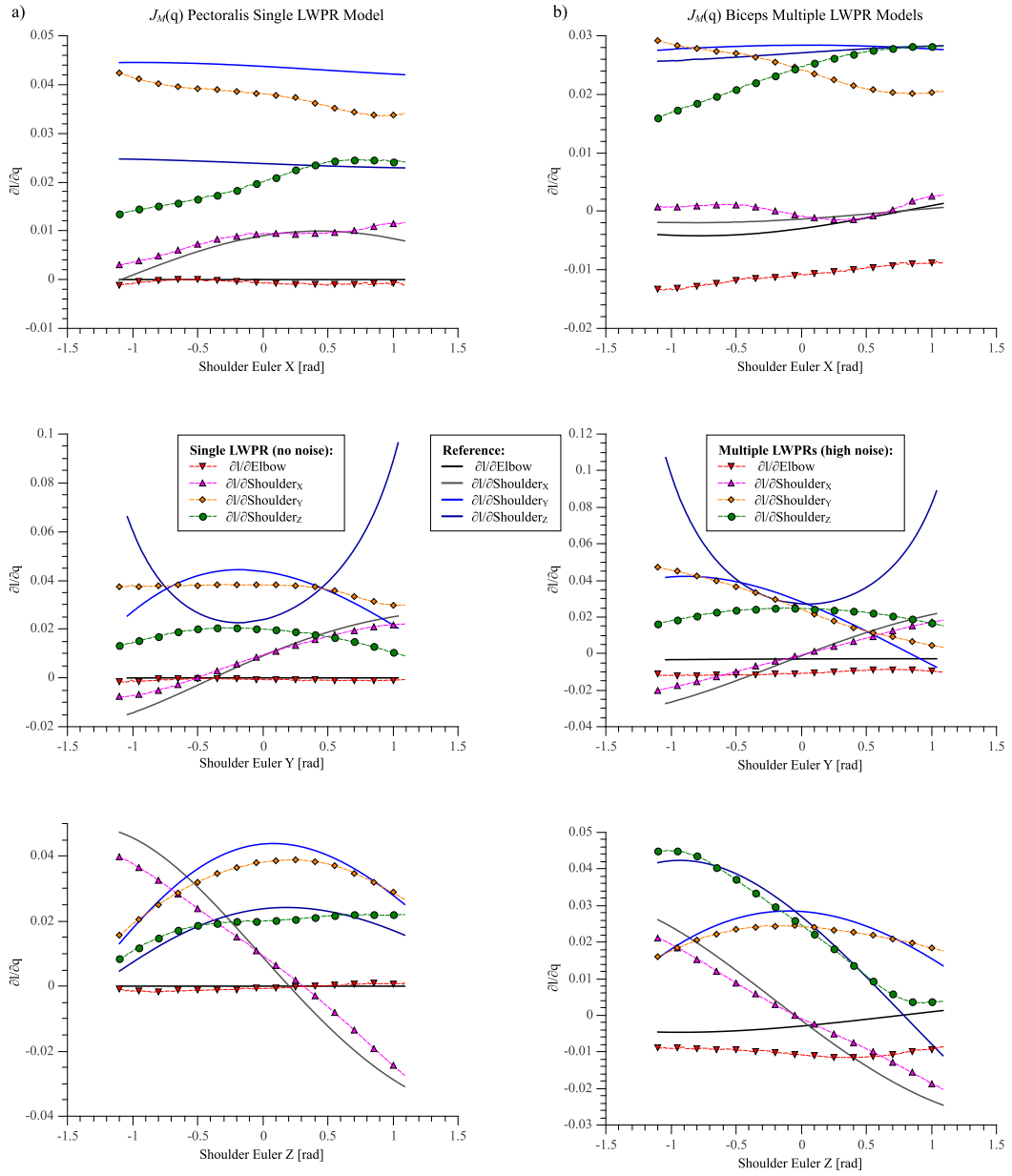


Figure 6.7: The partial derivatives of the estimated muscle jacobian for one muscle. Each of the three figure in a column is associated with one exemplary muscle and gives a different snapshot for each of the shoulder angles. All input dimensions not shown have been set to zero. Figure a) shows the partial derivatives of the Pectoralis muscle as estimated by the single LWPR model trained on the original training data without noise. Figure b) gives an example of the Biceps muscle using the decomposition in three LWPR models trained on highly noisy data.

It should also be noted that the computation of the muscle jacobian for a single joint configuration of the robot using single LWPR models takes on average 31.97 ms.

6.3.3 Decomposition in Multiple LWPR Models

In the second part of the experiments with locally weighted projection regression the function approximation task of the direct mapping was divided in three subtasks. The allocation of inputs and outputs was the same as before with the decomposition in multiple neural networks. One LWPR model was used for the Brachialis and Triceps muscle and therefore consisted of two submodels, each with just six receptive fields. The LWPR model for the Biceps muscle allocated 241 receptive fields. The remaining muscles of the shoulder joint were again incorporated in a further model. Each of its eight submodels had between 48 and 52 receptive fields. Every model was trained separately and apart from the appropriate joint angles also provided with generated features as further inputs.

6.3.3.1 Results for the Direct Mapping

The performance of this decomposition is summarized in Table 6.7.

	No Noise		Low Noise		High Noise	
	nMSE	$\overline{\text{error}}$ [mm]	nMSE	$\overline{\text{error}}$ [mm]	nMSE	$\overline{\text{error}}$ [mm]
Anterior Deltoid	9.05e-3	1.75e+0	9.02e-3	1.75e+0	9.04e-3	1.75e+0
Biceps	5.26e-3	1.89e+0	5.29e-3	1.90e+0	5.30e-3	1.90e+0
Brachialis	1.29e-5	4.74e-2	1.29e-5	4.74e-2	1.29e-5	4.74e-2
Infra	6.87e-3	8.77e-1	6.90e-3	8.81e-1	6.86e-3	8.75e-1
Lateral Deltoid	8.81e-3	1.66e+0	8.78e-3	1.66e+0	8.78e-3	1.66e+0
Pectoralis	9.56e-3	1.56e+0	9.57e-3	1.56e+0	9.52e-3	1.55e+0
Posterior Deltoid	1.27e-2	1.48e+0	1.29e-2	1.48e+0	1.26e-2	1.48e+0
Supra	1.28e-2	1.09e+0	1.27e-2	1.09e+0	1.27e-2	1.09e+0
Teres Major	7.76e-3	1.58e+0	7.75e-3	1.58e+0	7.74e-3	1.58e+0
Teres Minor	7.11e-3	9.01e-1	7.15e-3	9.02e-1	7.12e-3	9.02e-1
Triceps	3.20e-6	5.94e-2	3.20e-6	5.94e-2	3.21e-6	5.94e-2
Average	7.27e-3	1.17e+0	7.28e-3	1.17e+0	7.25e-3	1.17e+0

Table 6.7: The error values for a decomposition using three LWPR models trained on the original training data and with additive low and high noise. The normalized mean square error and the mean absolute error is given separately for each muscle and in average over all muscles. For details about the decomposition see text.

The error values show that the addition of noise to the training data did not make any difference in the performance of the LWPR models. The errors of the model incorporating the Brachialis and Triceps muscle are negligible and can be compared with the results achieved with the simplified test rig. In all other cases the errors are considerably higher and more or less of the same magnitude. Figure 6.6 shows exemplarily the output of the LWPR model for the Biceps muscle over the entire range of two shoulder angles. The deviation from the target is plainly visible.

One evaluation of the direct mapping using the three described LWPR models takes in average 0.559 ms.

6.3.3.2 Results for the Muscle Jacobian

It was eventually evaluated how accurate the values of the estimated muscle jacobian are if obtained using the previously presented decomposition in LWPR models. The corresponding normalized mean squared error and the average absolute error values are presented in Table 6.8.

Interestingly, at the first glance these numbers indicate much better results than in the case of the single LWPR models. In fact, the numbers seem comparable with the results

	No Noise		Low Noise		High Noise	
	nMSE	$\overline{\text{error}}$	nMSE	$\overline{\text{error}}$	nMSE	$\overline{\text{error}}$
Anterior Deltoid	7.63e-1	5.07e-3	7.63e-1	5.07e-3	7.64e-1	5.07e-3
Biceps	3.68e-1	6.37e-3	3.64e-1	6.32e-3	3.68e-1	6.37e-3
Brachialis	7.90e-1	3.78e-5	7.90e-1	3.78e-5	7.90e-1	3.78e-5
Infra	5.69e-1	5.48e-3	5.69e-1	5.48e-3	5.69e-1	5.48e-3
Lateral Deltoid	6.29e-1	5.10e-3	6.29e-1	5.10e-3	6.29e-1	5.10e-3
Pectoralis	6.31e-1	6.25e-3	6.32e-1	6.25e-3	6.32e-1	6.25e-3
Posterior Deltoid	5.90e-1	5.35e-3	5.90e-1	5.34e-3	5.90e-1	5.35e-3
Supra	5.31e-1	4.68e-3	5.32e-1	4.68e-3	5.32e-1	4.68e-3
Teres Major	5.58e-1	5.77e-3	5.58e-1	5.77e-3	5.58e-1	5.77e-3
Teres Minor	5.61e-1	5.07e-3	5.61e-1	5.07e-3	5.61e-1	5.07e-3
Triceps	8.46e-1	1.53e-4	8.46e-1	1.53e-4	8.46e-1	1.53e-4
Average	6.21e-1	4.48e-3	6.21e-1	4.48e-3	6.22e-1	4.49e-3

Table 6.8: The errors for a decomposition in three LWPR models with respect to the computation of the muscle jacobian. The normalized mean square error and the mean absolute error is given separately for each muscle and in average.

seen for neural networks. However, the results for the direct mapping already indicated that this might not be the case. Figure 6.7b shows some excerpts of the estimated muscle jacobian for the Biceps muscle. As before with the single LWPR models there are distinct differences to the desired values. Just as a footnote, the Biceps muscle is not independent of the elbow angle as also clearly displayed by the corresponding partial derivatives.

The computation of the muscle jacobian using the described decomposition in three LWPR models takes on average 5.69 ms.

6.4 Discussion

This section showed a clear superiority of the neural networks over locally weighted projection regression in all examined categories for this specific function approximation task. All experiments with neural networks attested a very good performance for the approximation of the direct mapping as well as the computation of the muscle jacobian. A decomposition in multiple subtasks did not provide much advantage in case of the neural networks, it even increased the runtime although it solved the task with less resources. However, it relieved the learning algorithm from recognizing the independence of the elbow muscles from the shoulder angles. The used implementation also performed very well regarding the runtime.

In contrast, the LWPR models performed in general not as good. The generation of further non-linear input features did not have such drastic effects as in the case of the simplified test rig. A decomposition of the task in three LWPR models showed some small improvements and reduced the runtime considerably. In fact, the time for a computation of the muscle jacobian in case of a single LWPR model would make a stable integration into a control loop very difficult. Even a control frequency of 50 Hz would need to have all computations finished in 20 ms. With a decomposition in multiple LWPR models the compliance to such a time limit seems more realistic.

Finally, the neural networks were also tested in the whole body controller described in Chapter 2. The results using the decomposed neural networks trained on the original data are also presented in [11]. It was shown that the control scheme performs well for the simulated test rig. Figure 6.8 illustrates how the controller followed a given trajectory involving movements of the elbow and shoulder. It can be seen that the system was able to follow the trajectory very closely and all disturbances disappeared quickly after the robot arm reached the targeted position.

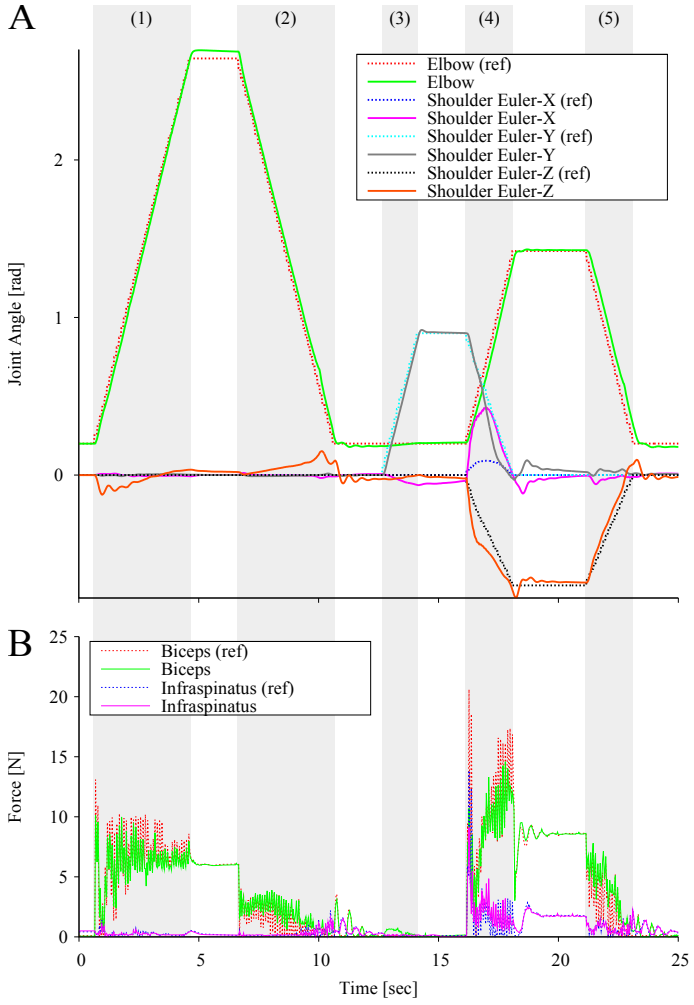


Figure 6.8: The usage of the learned muscle jacobian in a whole body control scheme to follow a given trajectory. Figure A shows the joint angles of test rig during a movement of only the elbow (1-2), a shoulder adduction (3) and a movement involving all joints (3-5). Figure B gives the corresponding forces for two involved muscles. (Figure taken from [11])

7. Conclusions and Outlook

This work successfully developed a technique for the extraction of the muscle jacobian for tendon-driven anthropomimetic robots. It examined the use of the two machine learning methods feedforward neural networks and locally weighted projection regression for the approximation of a mapping between the joint angles and the muscle lengths. Afterwards, the muscle jacobian was obtained by means of numeric derivation of this approximation. The approach was initially tested with a very simple robot arm consisting of a single hinge joint and three muscles spanned over it. Both neural networks and LWPR performed very well in this case. LWPR benefited considerably from the generation of additional non-linear input features. Eventually the method was applied to a more advanced anthropomimetic robot arm which also comprised a spherical joint and a muscle spanning multiple joints. Neural networks have proved to be a very suitable method for the task and showed an overall good performance. With LWPR models the results were in general distinctly worse.

Additionally, this work proposed two methods for the generation of training data required in the learning process. A large training set was collected by moving the simulated joints to different joint angles in combination with the generation of random movements. The training data was apparently comprehensive enough and covered the entire input space in order to allow the learning method a good generalization to the unseen regions. It would be interesting to examine how much the number of training samples can be reduced without a decrease in performance. In order to test the approach under more realistic conditions it was also examined how the performance develops if noise is added to both inputs and outputs of the training data. The accomplished results provide confidence that this concept will not only work in simulation but also with a real robot which will inevitably generate noisy measured data.

The decomposition of the function approximation task in multiple subtasks did not show a great advantage in the experiments of this work. However, if this approach is going to be applied to a robot with more joints and muscles, e.g. the upper-torso developed in the ECCEROBOT project, the benefits of a decomposed system will probably become more apparent. The more DoF a robotic system has the more important becomes the distribution in simpler subtasks. Apart from that a decomposition also provides a more efficient use of model resources and allows an easy exchange of single approximators for parts of the body or the combination of different machine learning techniques.

The main motivation for this work was the use of the muscle jacobian in a whole body control scheme based on computed torque control. Jaentsch et al. [11] could show that the

muscle jacobian obtained with the technique developed in this work led to a convincing performance with the proposed control scheme.

7.1 Future Works

Finally, this section presents a few future developments that will possibly follow this work and discusses some problems that might occur as well as how these could be solved.

7.1.1 Further Applications

Now that this approach has been developed and tested for a simplified simulation model of the robot it would be the next step to transfer these results to a more realistic model or the real test rig. It is the objective of another currently ongoing work to create an accurate simulation model of the test rig. This involves a physically correct simulation of the artificial muscles, especially with regard to the more complicated paths and colliding tendons, and the optimization of various model parameters, like the attachment points of the muscles, to adapt the simulation to the behavior of the real test rig. At the same time, the efforts continue to equip the test rig with everything necessary for the collection of real training and test data. On the one hand, this requires a facility to get the angles of all joints. A marker based stereo vision tracking system has already been successfully implemented for this purpose. On the other hand, the lengths of the muscles need to be measured. At the moment it is planned to infer the muscle length from he motor position and the elongation of the shock cord, which requires force sensors currently in development. It will yet have to turn out whether the accuracy of the values retrieved by this method are actually in the scope assumed in this work.

The application of the method developed in this work on the new simulation model or the real test rig will especially require some changes to the acquisition of data. In both cases, it becomes necessary to avoid the problem of slack muscles. Newly recorded training samples are only valid if the strain on all muscles is greater than zero. For this purpose, it is suggested to set the reference value of the force controllers in each muscle to a minimum value. Furthermore, in the case of the real robot, the generation of data has to be reconsidered. It will have to be examined whether the random movement method introduced in this work can be applied to the real robot. One proposed alternative is the passive generation of movements by setting the force controllers of all muscles to a minimum value and moving the limbs of the robot by hand in order to cover the input space as good as possible. Another suggestion is to train an initial model with data from the simulation. This approximation is probably not very accurate for the real robot but it still might be used to move the robot and generate some real training data that can eventually be used to adapt the model trained from simulated data or to train a new one. However, this at least requires that the simulation and the real robot have a sufficient level of resemblance in their setup and behavior.

7.1.2 Online Learning

Another desirable progression would be the implementation of a online learning scheme. In this case, this would mean the continuation of the learning process during the normal operation of the robot in oder to improve the learned model in previously badly explored regions of the input space or to adapt it to changes in the structure of the robot. Such changes could arise from distortions of the plastic bones that might occur slowly over time. This learning scheme could mean that a trained model is used in some way to operate the robot while continuously collecting data about the joint angles and muscle lengths and using them immediately in the learning algorithms to improve the model.

This poses a few new challenges. One could be the occurrence of negative interference which would mean while the model gets adapted in the environment of the current robot movements the approximation becomes worse again in other parts of the input space. Locally weighted projection regression is supposed to be more prone to those effects than a global function approximator like neural networks as it is designed for such an incremental learning scheme. This could already be slightly observed in a little experiment with the data from the simplified simulation model consisting only of the elbow joint and three muscles. The approximators for the original data without noise presented in Chapter 5 were used in the further training with data coming only from the angular range between 0.1 and 1.2 rad. It has been shown before that these models approximated the target function in the complete range of the elbow joint very precisely. However, Figure 7.1 shows that after the continued training the neural network has developed a very small but noticeable deviation in the upper marginal region of the elbow angle. In contrast, the LWPR model did not show any changes.

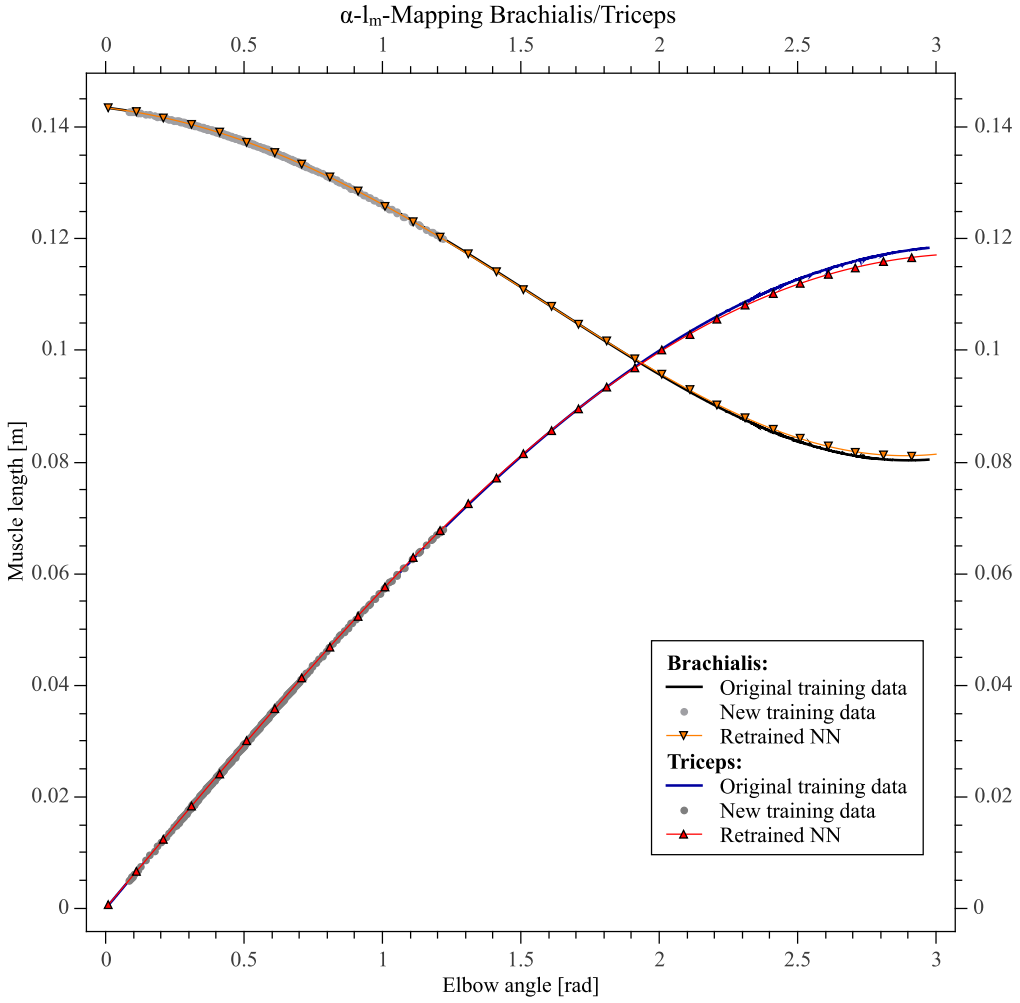


Figure 7.1: The output of the previously best neural network for the original data without noise after it was retrained on data of only a small segment of the input space. The targeted and estimated muscle lengths of the Brachialis and Triceps muscle are plotted. It can be seen that the output of the retrained network deviates slightly from the targeted function in the upper marginal region of the elbow angle.

Another difficulty is the decision when to stop the training on the newly gathered data. If the samples are presented too few times to the learning algorithm they have almost no effect and it takes a very long time until the approximator adapts to changes. On the other hand, if the samples are used too often for training the approximator might tend to overfitting. Additionally, as the training is supposed to take place during the normal operation of the robot this is to some extent also a time-critical application as continuously new training samples are recorded. Again, this seems rather a problem with neural networks than with locally weighted projection regression. It is difficult to decide for how many epochs a neural network should be trained for optimal results. Standard techniques like the early stopping method for the automated termination of the learning process in order to avoid overfitting have proven very unreliable most of the time in the training of neural networks for this work. For LWPR it seems to be a practicable method to present the incoming training samples only once or twice to the algorithm in order to achieve a slow adaption to changes. Furthermore, LWPR is supposed to be less prone to the problem of overfitting. It is very unfortunate that despite these promising properties of LWPR the results achieved in this work did not look so promising.

A. Software Documentation

This appendix describes the software that has been developed in the course of this work. First of all, the most important parts of the existing software framework are briefly introduced, on which the new software is established and had to be integrated into. Afterwards, the `MuscleJointManager` component developed in this work is presented along with its interfaces and classes.

A.1 The Existing Software Framework

A detailed description of the complete ECCEROBOT software framework would certainly go beyond the scope of this section. Therefore, only the parts most important for this work and the newly developed software are briefly introduced at this point.

The ECCEROBOT software is subdivided in various components in order to allow uncomplicated extensibility and customizability. The component-oriented development of software for robotic systems is a key principle of the underlying robotic middleware OpenRTM-aist [2] used here. Communication between all components is handled via the operation system and programming language independent Common Object Request Broker Architecture (CORBA).

A very important part for this work is the universal robot simulation framework *CALIPER* [39] consisting of several components. It mainly comprises the open source physics engine Bullet Physics[1], which simulates the dynamics of the robot and its environment, and the OpenGL-based graphics engine Coin3D[2] that renders the current state of the simulated scene. Furthermore, it provides import and export facilities for CAD robot models in the open COLLADA-format as well as additional tools for the investigation and control of the simulation.

The `MuscleControl` and `JointControl` interfaces provide all the functionality to control the robot and obtain the sensor values necessary for the task in this work. Both interfaces are (or will eventually be) implemented for the simulation as well as the real robot, so that the same methods can be used in both cases. The `MuscleControl` interface allows among other things to control the exerted force, motor voltage and motor position for each muscle individually. Additionally, all the sensor values of the muscles like lengths, force and motor position can be accessed via this interface. The `JointControl` interface

[1] http://bulletphysics.org
[2] http://www.coin3d.org/

provides the methods to get the angular values of each joint. In case of the simulation, it also allows the setting of joint angles.

A.2 The MuscleJointManager Component

The newly developed `MuscleJointManager` component is written in C++ and comprises all functionality for the acquistion of data and the management and training of machine learning methods. Figure A.1 gives a rough overview of the component and shows its internal structure as well as the interfaces for communication with the rest of the framework.

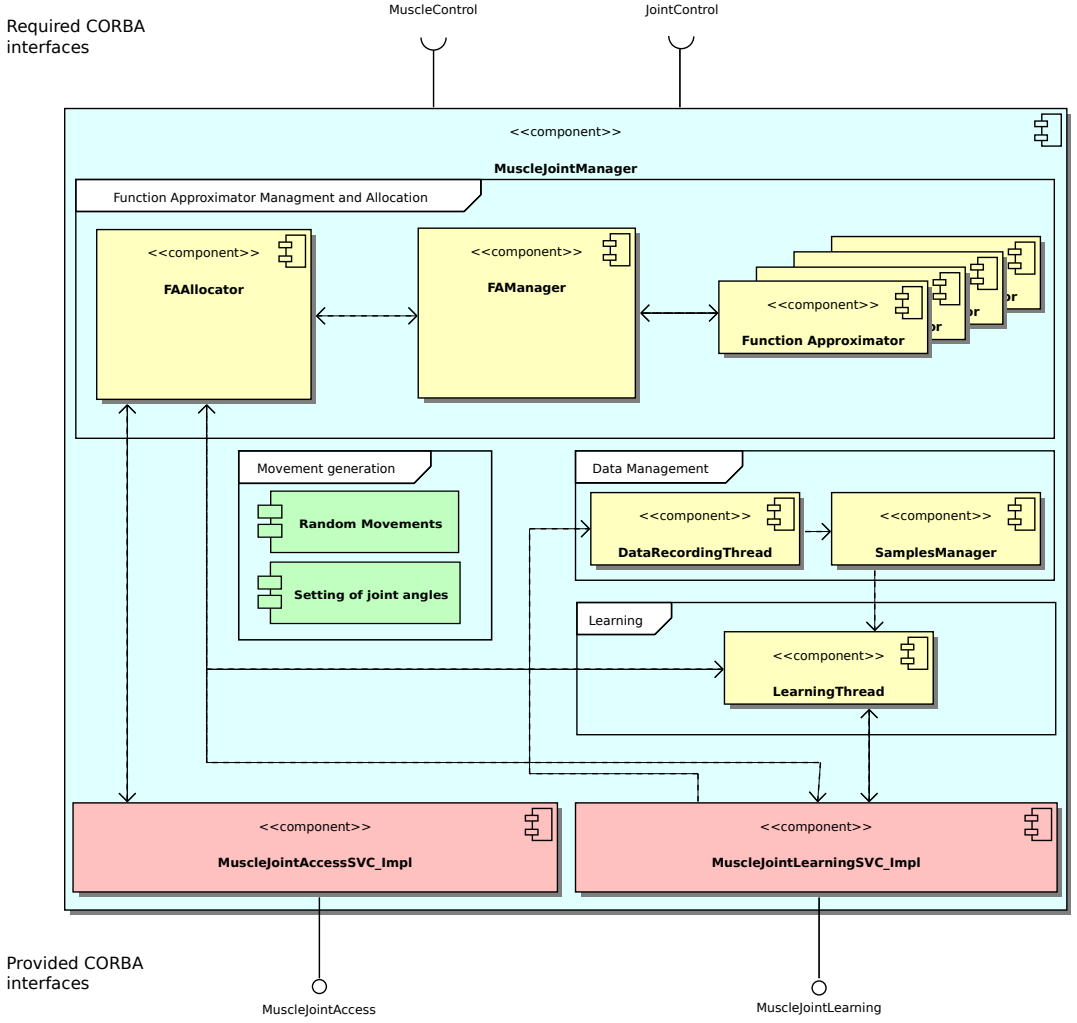


Figure A.1: An overview over the `MuscleJointManager` component. It shows the internal structure and the required as well as the provided CORBA interfaces.

The individual parts are described in more detail in the following subsections. The UML diagrams for the developed classes contain only the most relevant member attributes and methods. The focus is more on providing a general description of the functionality rather than a detailed documentation of each variable and method.

A.2.1 Interfaces

For all tasks concerned with the generation of movements and the collection of measured data the `MuscleJointManager` component has to be connected to the `MuscleControl` and `JointControl` interfaces.

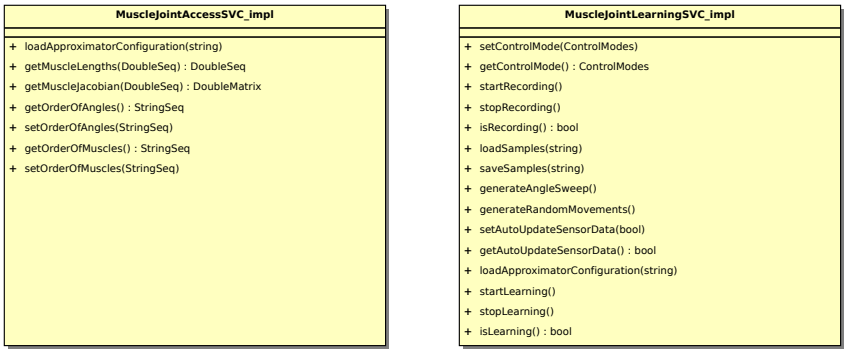


Figure A.2: The provided `MuscleJointAccess` and `MuscleJointLearning` interfaces.

The component itself provides two CORBA interfaces for the communication with the rest of the framework. Figure A.2 shows the two classes which implement these interfaces.

The `MuscleJointAccess` interface supplies a method for the loading of function approximators which yield an approximation of the direct mapping from joint angles to muscle lengths for a certain robot. The configuration of these approximators is defined in a XML file (see Section A.2.6). The method `getMuscleLengths(angles)` allows the retrieval of the estimated muscle lengths for a configuration of the robot specified by a vector of joint angles. Analogously, there is another method which returns the muscle jacobian by means of numeric derivation of the approximated direct mapping. Additionally, it is possible to define the order in which the joint angles have to occur in the input vectors and in which the muscle length are returned in the output vectors. These orders also affect the structure of the muscle jacobian.

While the `MuscleJointAccess` interface only allows the evaluation of an already trained set of approximators, the `MuscleJointLearning` interface is concerned with all tasks related with the learning process. It supplies the possibility to turn the recording of joint angles and muscle lengths on and off and to switch between the real robot and the simulator as source of this data. Already recorded samples can be saved to a file and loaded back into the system again. For the generation of data there are currently the two methods of setting the joint angles and random movements. If there is another component frequently requesting sensory data the automatic update of the `MuscleJointManager` can be turned off. Again, there is a method for the loading of function approximators. If the specified files do not exist new ones are created. And most importantly, this interface provides the control of the learning process.

A.2.2 Function Approximators

One of the most fundamental elements of the `MuscleJointManager` is component is the function approximator. A function approximator is in general a construct that somehow maps a vector of real-valued inputs to a corresponding vector of outputs. These vectors have to be of certain dimensions. In this implementation, a function approximator has also a name and is stored in a file. In order that it approximates the desired function it has to be trained before it is used. For this purpose, it has to be supplied with matching input-output-pairs (samples). Afterwards, the approximator can be evaluated by providing an input vector to it and it returns the approximated outputs. This basic concept is independent of the algorithm that is actually used to learn the desired function and that somehow generates the output values from the inputs. Therefore, the `FunctionApproximator` class (see Fig. A.3) is an abstract base class and has yet to be fully implemented by a specific

machine learning algorithm. Additionally, the `FunctionApproximator` class offers facilities for the transparent scaling of inputs and outputs. By providing the range of possible input and output values, all data going to and coming from the actual approximation algorithm is scaled internally. Thus, there is no need to normalize any values manually beforehand.

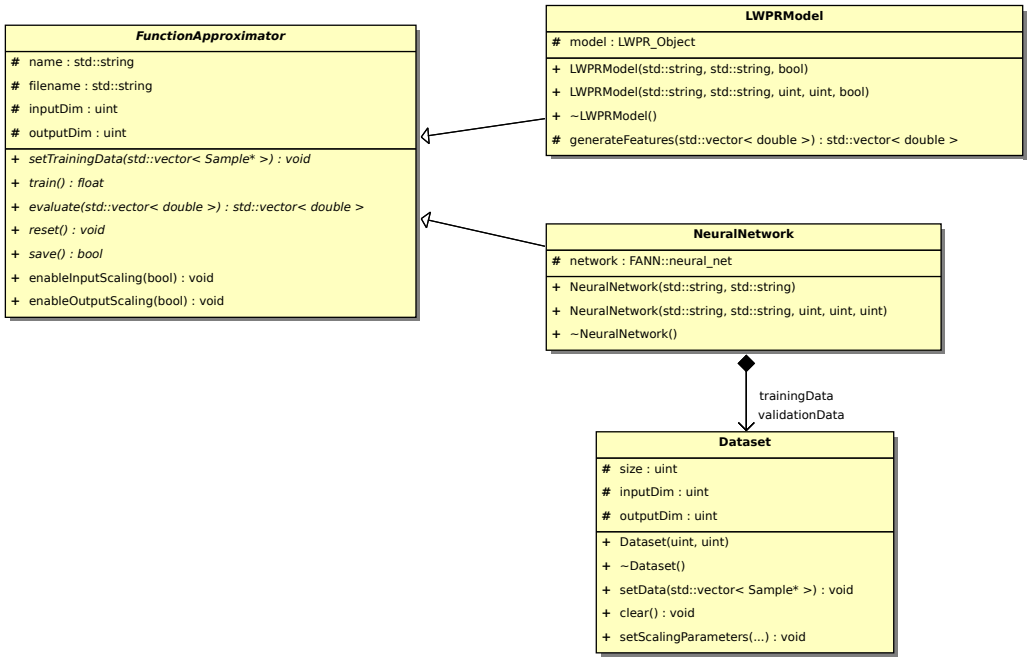


Figure A.3: UML diagram of the abstract `FunctionApproximator` class. The inherited classes `LWPRModel` and `NeuralNetwork` provide full implementations of function approximators.

There are currently two classes inheriting from the `FunctionApproximator` class. The `NeuralNetwork` class implements a multilayer perceptron (MLP) with a single hidden layer. For this purpose it uses the Fast Artificial Neural Network Library (FANN)[3] written in C. Upon creation of a new object an existing network is either loaded from a file or a new untrained network with a given number of input, hidden and output neurons is created. In the latter case the weights of the network are initialized with random values. All neurons of the hidden layer have the tanh and the output neurons a linear activation function. The training procedure uses the incremental update version of the backpropagation algorithm and implements the early stopping method. The data of the training and validation sets are stored in objects of the `Dataset` class. A dataset has the same input and output dimensions as the neural network and a certain number of samples. It also handles the potential scaling of the data. For neural networks, all input and output values are scaled internally to the range $[-1, 1]$.

The other implemented machine learning technique is locally weighted projection regression. The `LWPRModel` class uses the C++ library of the inventors of the algorithm[4]. This class mainly passes all data through to the `LWPR_Object` provided by the library. Upon creation of a new object an existing LWPR model is either loaded from a file or an empty model with given input and output dimensions is created. During the training process, the samples are presented in random order to the LWPR algorithm for a certain number

[3] http://leenissen.dk/fann/
[4] http://www.ipab.inf.ed.ac.uk/slmc/software/lwpr/

of times. The class also gives the opportunity to transparently generate further input features. These features are then automatically generated for the training data as well as for the inputs of a single evaluation of the model. Instead of scaling the input and output data the LWPR algorithm is directly provided with the range of possible inputs.

A.2.3 Function Approximator Management and Allocation

All function approximators in use by the `MuscleJointManager` are managed by the class `FAManager`. This class merely keeps track of all existing `FunctionApproximator` objects. It can process a configuration file (see Section A.2.6) and caters for the generation of function approximators, i.e. currently `NeuralNetwork` or `LWPRModel` objects, by either loading them from a file or creating new ones.

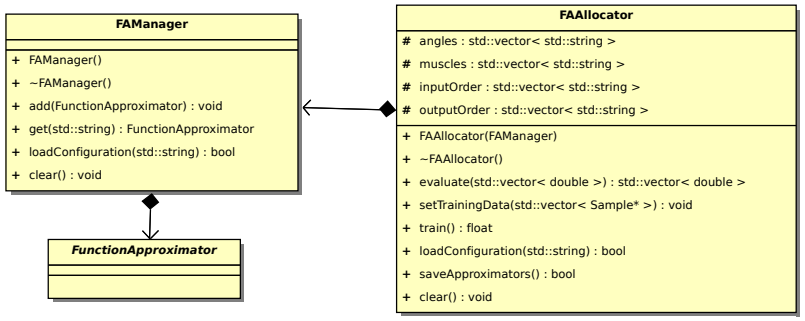


Figure A.4: UML diagram of the `FAManager` and `FAAllocator` classes.

The `FAAllocator` class provides the functionality to decompose a function approximation task into multiple subtasks which are each solved by an own function approximator. To the outside it acts as if it were a single function approximator. But internally, the input data is passed to the actual approximators and the individual outputs are afterwards assembled to a single output vector. The order of the components in the input and output vectors can be specified. The same holds for the passing of training data. The allocation of inputs and outputs to function approximators can be declared in a configuration file (see Section A.2.6).

A.2.4 Data Management

The `DataRecordingThread` handles independently of the other components the task of recording joint angles and muscle lengths. For this purpose, it has access to the `MuscleControl` and `JointControl` interfaces. Dependent on the specified control mode it accesses the corresponding implementations of the simulator or the real robot. In intervals of dt µs the thread checks whether the configuration of the robot has changed. If this is the case, it stores the current joint angle and muscle length values as a `Sample` in the `SamplesManager` (see Fig. A.5). If the control mode is set to the real robot, new samples are only recorded if the force of all muscles exceed a minimum value.

A `Sample` simply stores a pair of input and output vectors. The components of these vectors can be associated with labels, i.e. the angle and muscle names. All samples are combined in the `SamplesManager` which also provides the functionality to save and load all stored samples to a file.

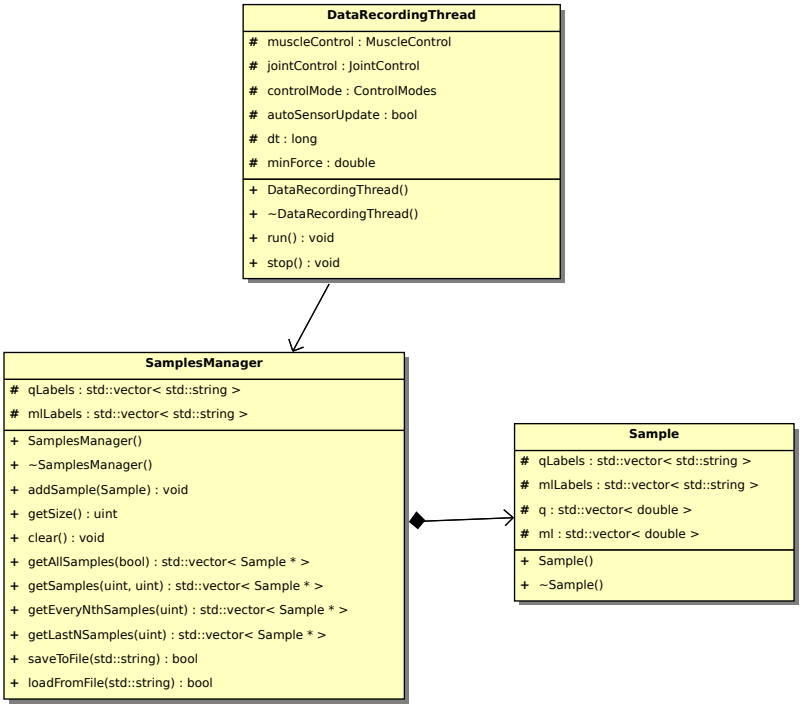


Figure A.5: The `DataRecordingThread` saves the recorded data in `Sample` objects managed by the `SamplesManager`.

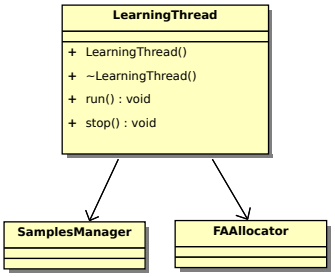


Figure A.6: The UML diagram of the `LearningThread` class.

A.2.5 Learning

The `LearningThread` (Fig. A.6) handles the training process. During its execution, it simply receives the training data from the `SamplesManager` and passes it to the learning algorithms of the function approximators comprised by the `FAAllocator`.

A.2.6 XML Configuration File

All information concerning the configuration of a set of function approximators is specified in a XML file. It allows the allocation of angles (inputs) and muscles (outputs) to the corresponding function approximators. Additionally, the range of possible input and output values is declared here. The specification of a function approximator requires its name and the file in which it is stored, as well as in some cases further type-specific options. Listing A.1 gives an example which allocates five joint angles and ten muscles to a neural network and a LWPR model.

Listing A.1: An exemplary XML configuration file as read by the MuscleJointManager component. It defines an allocation of the inputs (angles) and outputs (muscles) to a neural network and a LWPR model.

```
1
2  <?xml version="1.0" encoding="UTF-8"?>
3  <!DOCTYPE MJApproximatorConfiguration SYSTEM "dtd/MuscleJoint.dtd">
4
5  <MJApproximatorConfiguration>
6  <Angles>
7   <Angle name="ShoulderJoint_W" minValue="-1.0" maxValue="1.0"/>
8   <Angle name="ShoulderJoint_X" minValue="-1.0" maxValue="1.0"/>
9   <Angle name="ShoulderJoint_Y" minValue="-1.0" maxValue="1.0"/>
10  <Angle name="ShoulderJoint_Z" minValue="-1.0" maxValue="1.0"/>
11  <Angle name="ElbowJoint" minValue="0" maxValue="3.0"/>
12 </Angles>
13
14 <Approximators>
15
16  <NeuralNetwork name="elbow" file="shoulder_nn/elbowNetwork.net"
17     description="elbow only" hiddenUnits="5">
18   <Inputs>
19    <AngleRef name="ElbowJoint"/>
20   </Inputs>
21
22   <Outputs>
23    <Muscle name="R_Brachialis" minValue="0.08" maxValue="0.15"/>
24    <Muscle name="R_Triceps" minValue="0.0" maxValue="0.12"/>
25   </Outputs>
26  </NeuralNetwork>
27
28  <LWPRModel name="shoulder" file="shoulder_nn/shoulderNetwork.net"
29     description="shoulder only" featureGeneration="on" >
30   <Inputs>
31    <AngleRef name="ShoulderJoint_W"/>
32    <AngleRef name="ShoulderJoint_X"/>
33    <AngleRef name="ShoulderJoint_Y"/>
34    <AngleRef name="ShoulderJoint_Z"/>
35   </Inputs>
36
37   <Outputs>
38    <Muscle name="R_Anterior_Deltoid" minValue="0.06" maxValue="0.16"/>
39    <Muscle name="R_Infra" minValue="0.12" maxValue="0.18"/>
40    <Muscle name="R_Lateral_Deltoid" minValue="0.05" maxValue="0.16"/>
41    <Muscle name="R_Pectoralis" minValue="0.15" maxValue="0.25"/>
42    <Muscle name="R_Posterior_Deltoid" minValue="0.06" maxValue="0.16"/>
43    <Muscle name="R_Supra" minValue="0.12" maxValue="0.19"/>
44    <Muscle name="R_Teres_Major" minValue="0.11" maxValue="0.21"/>
45    <Muscle name="R_Teres_Minor" minValue="0.12" maxValue="0.18"/>
46   </Outputs>
47  </LWPRModel>
48
49 </Approximators>
50
51 </MJApproximatorConfiguration>
```

B. Time Derivatives of XYZ-Euler-Angles and Rotational Velocities

In this appendix the relationship of the time derivatives $\dot{\boldsymbol{\gamma}}$ of XYZ-Euler-Angles and the rotational velocities $\boldsymbol{\omega}$ is deduced. The analogue derivation for ZXZ-Euler-Angles is presented in [29] and will be be transferred to XYZ-Euler-Angles at this point.

XYZ-Euler-Angles $\boldsymbol{\gamma} = (\gamma_x,\, \gamma_y,\, \gamma_z)$ describe the orientation of the body fixed coordinate frame $\hat{i}$ labelled $(\hat{\boldsymbol{e}}_x,\, \hat{\boldsymbol{e}}_y, \hat{\boldsymbol{e}}_z)$ relative to the space fixed coordinate frame i denoted by $(\boldsymbol{e}_x,\, \boldsymbol{e}_y,\, \boldsymbol{e}_y)$. Each Euler-Angle represents a rotation about an axis of a moving coordinate frame. Initially, the moving frame $j = i$, at this step labelled $(\boldsymbol{e}_\xi,\, \boldsymbol{e}_\eta,\, \boldsymbol{e}_\zeta)$, is aligned with the fixed frame i. The sequence starts by rotating the frame j by an angle γ_x about the $\boldsymbol{e}_\xi = \boldsymbol{e}_x$ axis. The resulting coordinate frame j' is labelled $(\boldsymbol{e}_{\xi'},\, \boldsymbol{e}_{\eta'},\, \boldsymbol{e}_{\zeta'})$. In a second step, this new coordinate frame is rotated by an angle γ_y about the rotated $\boldsymbol{e}_{\eta'} = \boldsymbol{e}_{y'}$ axis producing the intermediate coordinate frame j'' denoted by $(\boldsymbol{e}_{\xi''},\, \boldsymbol{e}_{\eta''},\, \boldsymbol{e}_{\zeta''})$. And finally, this is rotated by an angle γ_z about the the twice rotated $\boldsymbol{e}_{\zeta''} = \boldsymbol{e}_{z''}$ axis to produce the body fixed coordinate frame $\hat{i}$ labelled $(\hat{\boldsymbol{e}}_x,\, \hat{\boldsymbol{e}}_y, \hat{\boldsymbol{e}}_z)$.

The angular velocities $\boldsymbol{\omega} = (\omega_x,\, \omega_y,\, \omega_x)$ expressed in the space fixed coordinate frame i are obtained by transforming the rates of the Euler-Angles—which are the angular velocities about the rotation axes $\boldsymbol{e}_\xi$, $\boldsymbol{e}_{\eta'}$ and $\boldsymbol{e}_{\zeta''}$—to the space fixed coordinate frame:

$$\begin{bmatrix}\omega_x\\ \omega_y\\ \omega_z\end{bmatrix} = \begin{bmatrix}\dot{\gamma_x}\\ 0\\ 0\end{bmatrix} + \boldsymbol{R}_\xi(\gamma_x) \begin{bmatrix}0\\ \dot{\gamma_y}\\ 0\end{bmatrix} + \boldsymbol{R}_\xi(\gamma_x)\, \boldsymbol{R}_{\eta'}(\gamma_y) \begin{bmatrix}0\\ 0\\ \dot{\gamma_z}\end{bmatrix} \tag{B.1}$$

The rotation matrix $\boldsymbol{R}_\xi(\gamma_x)$ transforms a vector expressed in coordinate frame j' to a vector expressed in coordinate frame i and $\boldsymbol{R}_{\eta'}(\gamma_y)$ analogously transforms from coordinate frame j'' to coordinate frame j':

$$\boldsymbol{R}_\xi(\gamma_x) = \begin{bmatrix}1 & 0 & 0\\ 0 & \cos(\gamma_x) & -\sin(\gamma_x)\\ 0 & \sin(\gamma_x) & \cos(\gamma_x)\end{bmatrix} \quad , \quad \boldsymbol{R}_{\eta'}(\gamma_y) = \begin{bmatrix}\cos(\gamma_y) & 0 & \sin(\gamma_y)\\ 0 & 1 & 0\\ -\sin(\gamma_y) & 0 & \cos(\gamma_y)\end{bmatrix} \tag{B.2}$$

The terms of Eq. (B.1) can be expanded to:

$$\boldsymbol{\omega} = \boldsymbol{A}\,\dot{\boldsymbol{\gamma}} \quad \text{, with} \quad \boldsymbol{A} = \begin{bmatrix} 1 & 0 & \sin(\gamma_y) \\ 0 & \cos(\gamma_x) & -\sin(\gamma_x)\cos(\gamma_y) \\ 0 & \sin(\gamma_x) & \cos(\gamma_x)\cos(\gamma_y) \end{bmatrix} \begin{bmatrix} \dot{\gamma_x} \\ \dot{\gamma_y} \\ \dot{\gamma_z} \end{bmatrix} \tag{B.3}$$

This can be solved for the vector $\boldsymbol{\gamma}$ by inverting the matrix $\boldsymbol{A}$:

$$\dot{\boldsymbol{\gamma}} = \boldsymbol{B}_{\mathrm{XYZ}}(\boldsymbol{\gamma})\,\boldsymbol{\omega} \quad \text{, with} \quad \boldsymbol{B}_{\mathrm{XYZ}}(\boldsymbol{\gamma}) = \begin{bmatrix} 1 & \frac{\sin(\gamma_x)\sin(\gamma_y)}{\cos(\gamma_y)} & \frac{\cos(\gamma_x)\sin(\gamma_y)}{\cos(\gamma_y)} \\ 0 & \cos(\gamma_x) & \sin(\gamma_x) \\ 0 & -\frac{\sin(\gamma_x)}{\cos(\gamma_y)} & \frac{\cos(\gamma_x)}{\cos(\gamma_y)} \end{bmatrix} \tag{B.4}$$

Matrix $\boldsymbol{A}$ becomes singular at the representation singularities. In these configurations, it is not possible to describe an arbitrary angular velocity with a set of Euler-Angle time derivatives. The representation singularities occur for all kinds of Euler-Angles if the first and the last axes of rotation in the sequence lie along the same direction. For XYZ-Euler-Angles this is the case if the second rotation angle is $\gamma_y = \pm 90°$. One can see that the cosine terms in the divisors in matrix $\boldsymbol{B}_{\mathrm{XYZ}}(\boldsymbol{\gamma})$ become zero for these angles.

Bibliography

[1] E. Alpaydin. *Introduction to machine learning.* Jan 2004.

[2] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W. Yoon. Rt-middleware: distributed component middleware for rt (robot technology). *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2005)*, Jan 2005.

[3] C. M. Bishop. *Pattern recognition and machine learning.* Jan 2006.

[4] R. Burden and J. Faires. *Numerical analysis.* 2005.

[5] A. D'Souza, S. Vijayakumar, and S. Schaal. Learning inverse kinematics. *Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Jan 2001.

[6] S. E. Fahlman and C. Lebiere. The cascade-correlation learning architecture. *Advances in Neural Information Processing Systems*, 2:524–532, Jan 1990.

[7] O. Holland. A strongly embodied approach to machine consciousness. *Journal of Consciousness Studies*, Jan 2007.

[8] O. Holland and R. Knight. The anthropomimetic principle. *Proceedings of the AISB06 Symposium on Biologically Inspired Robotics*, Jan 2006.

[9] K. Hornik, M. Stinchombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, Jan 1989.

[10] R. Jacobs, M. Jordan, S. J. Nowlan, and E. H. Geoffrey. Adaptive mixtures of local experts. *Neural Computation*, 3(1), Jan 1991.

[11] M. Jaentsch, C. Schmaler, S. Wittmeier, K. Dalamagkidis, and A. Knoll. Joint-space control for an anthropomimetic robot. 2011. submitted.

[12] M. Jaentsch, S. Wittmeier, and A. Knoll. Distributed control for an anthropomimetic robot. *The 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Aug 2010.

[13] E. Kandel, J. Schwartz, and T. Jessell. *Principles of neural science.* 2000.

[14] J. Kreuziger. Application of machine learning to robotics-an analysis. *Proceedings of the Second International Conference on Automation, Robotics, and Computer Vision*, Jan 1992.

[15] Y. LeCun, L. Bottou, G. Orr, and K. R. Mueller. Efficient backprop. *Neural Networks: Tricks of the Trade*, Jan 1998.

[16] H. G. Marques, M. Jaentsch, S. Wittmeier, O. Holland, C. Alessandro, A. Diamond, M. Lungarella, and R. Knight. Ecce1: the first of a series of anthropomimetic musculoskeletal upper torsos. *Proceedings of the IEEE-RAS International Conference on Humanoid Robots 2010*, Dec 2010. accepted.

[17] M. McKinley and V. O'Loughlin. *Human Anatomy.* 2009.

[18] I. Mizuuchi, Y. Nakanishi, Y. Sodeyama, Y. Namiki, T. Nishino, N. Muramatsu, J. Urata, K. Hongo, T. Yoshikai, and M. Inaba. An advanced musculoskeletal humanoid kojiro. *7th IEEE-RAS International Conference on Humanoid Robots*, Jan 2009.

[19] I. Mizuuchi, T. Yoshikai, Y. Sodeyama, Y. Nakanishi, A. Miyadera, T. Yamamoto, T. Niemelae, M. Hayashi, J. Urata, Y. Namiki, T. Nishino, and M. Inaba. Development of musculoskeletal humanoid kotaro. *Proceedings of the 2006 IEEE International Conference on Robotics and Automation*, Jan 2006.

[20] Y. Nakanishi, K. Hongo, I. Mizuuchi, and M. Inaba. Joint proprioception acquisition strategy based on joints-muscles topological maps for musculoskeletal humanoids. *IEEE International Conference on Robotics and Automation (ICRA 2010)*, Jan 2010.

[21] J. Peters. *Machine learning for robotics: learning methods for robot motor skills.* 2008.

[22] F. Pourboghrat. Neural networks for learning inverse-kinematics of redundant manipulators. *International Joint Conference on Neural Networks*, Jan 2002.

[23] P. Rochat. Self-perception and action in infancy. *Experimental Brain Research*, 123(1):102–109, 1998.

[24] R. Rojas. *Neural networks: a systematic introduction.* 1996.

[25] D. E. Rumelhart, G. E. Hintont, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[26] K. Saladin. *Anatomy and Physiology: The Unity of Form and Function.* 2009.

[27] S. Schaal and C. G. Atkeson. Receptive field weighted regression. *ART Human Inf. Process. Lab.*, Jan 1997.

[28] S. Schaal and C. G. Atkeson. Constructive incremental learning from only local information. *Neural Computation*, Jan 1998.

[29] A. L. Schwab and J. P. Meijaard. How to draw euler angles and utilize euler parameters. *Proceedings of IDETC/CIE*, Jan 2006.

[30] B. Siciliano and O. Khatib. *Springer handbook of robotics.* Jan 2008.

[31] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo. *Robotics: modelling, planning and control.* 2009.

[32] G. Thimm and E. Fiesler. High-order and multilayer perceptron initialization. *IEEE Transactions on Neural Networks*, 8(2), Jan 1997.

[33] D. Trim. *Multivariable Calculus.* 1993.

[34] S. Vijayakumar, A. D'Souza, and S. Schaal. Incremental online learning in high dimensions. *Neural Computation*, Jan 2005.

[35] S. Vijayakumar, A. D'Souza, and S. Schaal. Lwpr: A scalable method for incremental online learning in high dimensions. *Neural Computation*, Jan 2005.

[36] S. Vijayakumar and S. Schaal. Locally weighted projection regression : An o(n) algorithm for incremental real time learning in high dimensional space. *Proc. of 17th International Conference on Machine Learning (ICML2000)*, pages pp. 1079–1086, Sep 2000.

[37] P. Werbos. *The roots of backpropagation: from ordered derivatives to neural networks and political forecasting.* 1994.

[38] W. Wiegerinck, A. Komoda, and T. Heskes. Stochastic dynamics of learning with momentum in neural networks. *Journal of Physics A: Mathematical and General*, 27(13), Jan 1994.

[39] S. Wittmeier, M. Jaentsch, K. Dalamagkidis, M. Rickert, H. G. Marques, and A. Knoll. Caliper: A universal robot simulation framework. Aug 2010. submitted.

[40] H. Wold. Soft modeling by latent variables: the nonlinear iterative partial least squares approach. *Perspectives in probability and statistics*, Jan 1975.

[41] V. M. Zatsiorsky. *Kinetics of human motion.* Jan 2002.