Learning Motor Control for Simulated Robot Arms

Djordje Mitrovic

Master of Science
Artificial Intelligence
School Of Informatics
University Of Edinburgh
2006

Abstract

Controlling a high degree of freedom humanoid robot arm to be dextrous and compliant in its movements is a critical task in robot control. The dynamics of such flexible and light manipulators have a highly non-linear nature, making analytical closed form solutions using rigid body assumptions inappropriate. In this thesis, we use Locally Weighted Projection Regression to learn online the inverse dynamics of a high degree of freedom dextrous robot arm in a physically simulated environment. The learned control task is based on a visual servoing scenario, which incorporates trajectory planning, inverse kinematics and motor control.

We build a powerful simulation framework for robot arms using the Open Dynamics Engine. The developed software places emphasis on flexible creation of any arbitrary robot arm, by incorporating polyhedral object geometry for optimised 3D rendering and physical parameter calculation. The extensible software architecture allows for easy incorporation of other learning algorithms and control paradigms.

The goodness of the learned inverse dynamics and the simulation model is verified using several experiments. Finally we learn the inverse dynamics model of a robot arm for a defined operating region, and show the goodness of the learned model by evaluating it against recorded human motion. We aim in the long run to transfer the gained insights and results of the reference arm to a real hardware implementation.

Acknowledgements

I would like to thank Dr. Sethu Vijayakumar, my supervisor, for his invaluable advice, support and the confidence he placed in me throughout this project.
I also would like to thank Dr. Heiko Hoffmann for his great support and patience during our discussions and debugging sessions.
I further would like to express my gratitude to Georgios Petkos for his friendly support and motivating words and to Dr. Marc Toussaint for his amazing MT-library, which I found invaluable.
Last but not least, I would like to thank my friend Siddhu Warrier for proof-reading my dissertation.

Djordje Mitrovic

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Djordje Mitrovic)

Table of Contents

1 Introduction

Over the past decades, the importance of robot arms in a wide range of applications like industrial manufacturing, and handling of heavy objects or hazardous materials has increased immeasurably. They are used for the fast and accurate execution of repeated tasks in an isolated industrial environment. This requires a functional design specialised for a specific task with stiff joints and strong actuators. Modern robot arms are becoming more and more compact, light and stable, and are often designed to operate in human environments. Such anthropomorphic robots must be capable of performing independent tasks in a human-like way, which means that its movements must be versatile, fast, accurate and energy efficient. In addition to that, the robot must be capable of reacting appropriately to external stimuli. This property is usually referred to as "compliance". To incorporate these desirable properties, a robot arm requires a high number of degrees of freedom.

It is desirable in many applications to have robot arms which can follow a specific trajectory. If a multi-joint robot arm tries to follow a desired trajectory, the appropriate motor commands must be applied to each joint of the arm at all times during the motion. Every joint is dependent on its parent joints, and wrong motor commands can lead to undesirable movements; this can result in unpredictable consequences which may have an adverse impact on the robot's task and its environment. Furthermore, the correct motor commands must be adapted over time in order to take into consideration changing physical parameters such as arm load, orientation, or joint friction.

Classic *feedback* control showed to be inappropriate for compliant movements since it requires high gains in order to guarantee accuracy. However, high gains can result in large correction forces which reduce compliance. In order to deal with this problem, a *feed-forward* component with an accurate inverse dynamics model of the robot arm can be used. Then, desired motions can be predicted and only small correction forces are required, which increases the compliance. Analytical approaches make rigid-body assumptions to calculate the inverse dynamics model. However, calculating the dynamics analytically is complex, and in many cases, precise dynamics parameters may not be known. Humanoid robots with large degrees of freedom, lightweight links and flexible joints do not meet rigid body assumptions.

1.1 Learning to Move

Recent approaches treat the solution of inverse dynamics as a function approximation problem. A robot arm produces a vast amount of data (joint angles, velocities, accelerations

and torques) during its movements, which can be used as training data for a learning algorithm. The learned function, representing the inverse dynamics model, can then be applied to a low gain composite controller with a learning component that allows a robot arm to make movements that are both precise and compliant at the same time, and also adapt "during operation" to changing situations. Learning of the system dynamics also seems to be biologically plausible. Humans are born with minimal motor skills, which they hone as they grow older. Their motor skills are also adaptive, to changes in the body's physical morphology, and the nature of the task being performed. Furthermore biological systems are able to cope with large sensor delays and still accomplish compliant and precise motion. Creating algorithms and robot systems that show such learning behaviour may help develop an understanding of how motor learning takes place in biological systems. The aim, in the long run, is the creation of robots with highly sophisticated and adaptive motor skills.

1.2 Simulation as a Tool

Parallel to the rapid development in robot hardware and its algorithms in recent years, the field of 3D computer graphics and virtual environments has also progressed by leaps and bounds. Researchers are no longer solely restricted to working with physical robotic systems. Operating real robots requires specialised knowledge, and limitations are imposed on the number of users who can use the system simultaneously. Furthermore, the construction of new robots is very complex, time-consuming, and could require interdisciplinary cooperation. This may not be financially feasible. Therefore, an alternative approach would be to use robot simulations which allow researchers to carry out experiments on the computer.

Simulations are not prone to defects, need no specific hardware knowledge, and are not dangerous in the least. Parameters like arm weight, form or number of joints can be easily changed in the simulation model. Ideally one would first prototype a robot arm and its control algorithms in a simulation. As a second step, the learned parameters and results could be transferred to the real hardware implementation of the robot arm.

During recent years, several robotics simulators have been developed. *Webots* (Michel, 2004) is probably the most popular robot simulator framework. The advantage of an existing graphics interface and a neat scene-tree is that it allows for comfortable object

and scene handling in the simulation. Webots uses the Open Dynamics Engine (ODE)[1] for its physics simulation and offers an interface library to ODE with C++. However the access to ODE is realised via a custom shared library loaded during runtime. Another, very powerful simulator, the *Yobotics Simulation Software*[2], was created with a strong focus on robot arm and leg systems. It has integrated functions for data generation, evaluation and reporting, which aid in performing exact analyses of rigid body control. Older robot simulation packages - such as Robotica for Mathematica (Nethery & Spong, 1994) and the Robotics Toolbox for Matlab (Corke, 1996) - do not have the capability to perform physical simulation. The following table lists some other well known robot (arm) simulators.

Camelot	Industrial robot arm simulation environment. Includes functionality for object modelling, test and transfer to real robots. www.camelot.dk
Dynawiz XMR	Multibody dynamics simulation for robot design and analysis. Simulates rigid and flexible bodies. Includes dynamics calculation. Interface to Matlab Simulink. www.concurrent-dynamics.com
EASY-ROB	3D robot simulation tool for producing high quality and high speed rendered images in 3D format. Special focus on industrial robots. www.easy-rob.de
Juice	3D robot simulation environment with realistic physics. Optimised for developing walking and rolling robots using a visual programming language. www.natew.com/juice
RoboWorks	Tool for modelling, simulation and animation of any mechanical system. Includes a physical simulation and a general interface to C++, Visual Basic, and Matlab among others. www.newtonium.com

Table 1.1: Robot (arm) simulators.

[1] http://www.ode.org

[2] http://yobotics.com/simulation/simulation.htm

1.3 Outcome

This dissertation is about realising a visual servoing for a robot arm in a virtual 3D environment. The visual servoing task was chosen because it encompasses three important phases of robotic control. These three phases are: The planning of a trajectory, its translation into joint angle space, and the execution of correct motor commands.

A major part of this project deals with the implementation of a simulator for robot arms. We create a physical robot simulation framework with ODE, which allows us to simulate robot arms that adhere to physical laws. These robot arms can have any arbitrary shape, and can interact with other objects in the environment. The simulator is extensible in that it also permits the addition of new control and learning algorithms. The implemented simulator has the ability to perform fast prototyping of different kinds of robot arms and its associated control algorithms.

An easy-to-use text-based interface is used to call functions for collecting data, following trajectories, tuning parameters, applying learning algorithms, and simulating the robotic system in a virtual 3D environment. The main purpose of the simulator is to create a set of base functions that can be utilised as-is, and can additionally be easily extended for individual experiments.

We utilise the implemented simulator framework to learn the inverse dynamics of a reference robot arm. This is done using Locally Weighted Projection Regression (LWPR) (Vijayakumar, D'Souza, & Schaal, 2005), an algorithm that performs particularly well on online learning control for robot systems with many degrees of freedom. Besides the theoretical concepts of LWPR, we will also discuss important practical issues, and present the learning success with experimental results. In the long run, it is aimed to use the collected results and insights from the simulation for the aforementioned reference robot, which is expected to be installed at the *Statistical Machine Learning and Motor Control Group (SLMC)*[3] in future.

1.4 Dissertation Outline

This project thesis is separated into three main parts.

The first part discusses the mechanisms required to perform robot arm control and learning in a simulated environment. Chapter 2 presents some fundamental concepts of

[3] http://www.ipab.inf.ed.ac.uk/slmc

robot (arm) control. The principles of trajectory planning, kinematics and dynamics are explained by taking into consideration a visual servoing task. We explain classic approaches to motor control and motivate the use of learning approaches for control problems. Chapter 3 describes the LWPR algorithm, and focuses on its desirable properties (in the context of learning robot arm control). This is followed by Chapter 4, wherein we delve into a brief discussion of the basics of physical simulations of rigid bodies, as it is used by ODE. In this chapter, an outline of different rigid body representations, body constraints and a universally valid physics engine architecture is presented. We additionally describe the basic components of a simulation loop in ODE.

The second part deals with the development of a flexible robot arm simulation framework. Chapter 5 explains the required components and their implementation. The manner in which all the components cooperate with each other is described using the simulator architecture.

The final part of the thesis shows how the simulator can be used to learn the inverse dynamics of a robot arm in practice. Chapter 6 explains the general learning approach using the simulator, and presents experimental results of the learned inverse dynamics model. The qualitative and quantitative evaluation of the learned inverse dynamics model is performed by applying a visual servoing task.

2 Robot Arm Control

This chapter reviews the fundamental principles of robot control, with a focus on the control of a robot arm. We create a reference robot arm model based on the construction plans and physical parameters of the *DLR Light Weight Robot III (DLW III)*. This is the third generation of a seven degree of freedom robot arm designed by the *Institute of Robotics and Mechatronics at the German Aerospace Centre (DLR)*. Like the human arm, the DLW III has seven degrees of freedom[4], allowing for highly flexible movements. Each joint has a position sensor, a joint velocity sensor and a joint torque sensor, which allows the robot arm to be operated by position, joint velocities or joint torques. The DLW III's reach is about 1 m, it weighs 13.5 kg, and can handle loads up to 15 kg, resulting in an excellent weight-to-load ratio. The arm is used not just in space applications, but also in terrestrial research environments. Throughout this work we shall refer to the DLW III as *robot manipulator, robot arm,* or simply *arm*.

Figure 2.1: The design-award winning DLW III, with the hand holding an egg (Reproduced from (Hirzinger, 2006)).

2.1 The Model of a Robot Arm

An idealised robot arm is represented as a system of articulated rigid bodies. The end-effector position can be represented in two different coordinate frames. One is the *proximal space* or *joint angle space*, which is defined by the joint positions and the joint angles of the robot arm. The other is the *distal space*, often referred to as *task space*, *Euclidean space* or *world coordinate system*. The distal space is better suited for task

[4] Without considering the hand.

definitions - for instance, to reach at an object - since it follows our natural understanding of positions in three-dimensional space. Since a robot arm is controlled in joint angle space (by applying torques at each joint), transformations between these two coordinate frames have to be performed.

The transformation from joint angle space to task space is called *forward kinematics*. Let $x \in \Re^n$ be some desired target position in task space, and $\alpha \in \Re^m$ the set of joint angles defining the end-effector position of the arm. We then can define a unique mapping k form joint angle space coordinates to task space coordinates as:

$$x = k(\alpha) \ .$$

Defining the forward kinematics function k is a straightforward geometric calculation, based on the arm's morphology which includes relative joint positions, joint angle ranges and link lengths. Inverse *kinematics* is the mapping from task space coordinates to joint angle space coordinates, and is represented as:

$$\alpha = k^{-1}(x)$$

It addresses the natural concept of *goal directed motion* (Korein & Badler, 1982), which corresponds to finding the correct joint angles of the arm from a given position in task space. It is not guaranteed that an inverse kinematics solution exists. For redundant manipulators, however, there can be infinite solutions. Solutions to the inverse kinematics problem are discussed in section 2.2.2

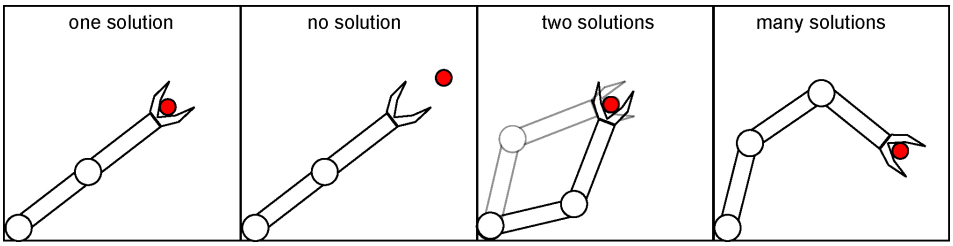


Figure 2.2: Different inverse kinematics scenarios in 2D.

Similar to forward and inverse kinematics, we can define *forward and inverse dynamics*. Unlike kinematics, dynamics take into account the physical behaviour of the arm system under the influence of joint torques and gravitational forces. This problem can be formalised as a state representation of the robot arm (Moore, 1990). This state representation contains enough parameters to model and therefore predict the behaviour of the arm system in the world. For a robot arm, we define a state s as a set of two vectors α and $\dot{\alpha}$ that hold all the joint angles and joint angle velocities respectively. The joint angle accelerations $\ddot{\alpha}$ represent the dynamic behaviour of the system, which is dependent on the current state of $s = (\alpha, \dot{\alpha})$ and the applied joint torques τ .

$$\ddot{\alpha} = d(s, \tau)$$

The function d is the *forward* or *direct* d*ynamics* of a robot arm. The *inverse dynamics* function d^{-1} calculates the required joint torques that have to be applied to the system in order to reach a desired acceleration $\tilde{\ddot{\alpha}}$. The inverse dynamic function can be formalised as:

$$\tau = d^{-1}(s, \tilde{\ddot{\alpha}})$$

Section 3.1 shows that an analytical solution to the inverse dynamics problem is non-trivial, and also specifies the advantages inherent in the application of learning algorithms to the problem.

2.2 Controlling an Arm

The control of an arm can be evaluated on many levels of logical abstraction. In the context of this work, we are mainly interested in the high-level task of following a desired trajectory defined in 3D task space. This task takes place in three phases:

1. Trajectory planning
2. Inverse kinematics
3. Motor Control

2.2.1 Trajectory Planning

The aim of trajectory planning is to enable a robot end-effector to move from an initial position to another position. This transition forms the basis for many high-level tasks like reaching objects or avoiding obstacles.

The generation of trajectories is often realised by means of an optimisation process for defined cost functions. Some approaches have analysed cost functions for the impulse, energy or peak-acceleration ("bing-bing") of movements (Nelson, 1983). Other simplistic cost functions involve shortest distance (Park & Brockett, 1994), minimum acceleration (Noakes, Heinzinger, & Paden, 1989), and minimum-time ("bang-bang") (Shiller, 1994). However, it has been observed that human motion is characterised by smooth trajectories with a bell-shaped velocity profile (Wolpert, Ghahramani, & Jordan, 1995). A smooth bell-shaped velocity profile is therefore an important criterion when evaluating cost functions. One way to achieve smooth motion is to use the *jerk* as an optimisation criterion (Flash & Hogan, 1985). The jerk is the change in acceleration over time, the third time derivative of the position, x. For a given start time t_s and end time t_e, the cost function to be minimised is defined as (Shadmehr & Wise, 2005):

$$C(x(t)) = \frac{1}{2}\int_{t_s}^{t_e} \dddot{x}(t)^2 dt$$

(Flash & Hogan, 1985) showed that a function with a sixth derivative equal to zero ($x^{(6)} = 0$) optimises the motion for minimum jerk. The general solution to this is a fifth order polynomial:

$$x(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5$$

By including known constraints to the motion, such as the motion duration d, position, velocity and acceleration at start and end of the motion, a general solution can be found. It states the smoothest possible transition with from start position x_s to end position x_e:

$$x(t) = x_s + (x_e - x_s)\left(10\left(\frac{t}{d}\right)^3 - 15\left(\frac{t}{d}\right)^4 + 6\left(\frac{t}{d}\right)^5\right)$$

The presented minimum jerk trajectory planning is dependent only on the kinematics of the arm system, i.e., it does not take into consideration any forces affecting the arm[5]. Other cost functions, for example the *minimum torque change* (Uno, Kawato, & Suzuki, 1989), take into account the dynamics of a robot arm. They cannot be calculated analytically and have to be solved by iterative solutions.

For more complex movements, multiple optimisation criteria are often incorporated at different times during the motion. Besides being smooth, a planned trajectory should not conflict with the robot's joint limits or violate the available motor performance and possible external restrictions. For the rest of this chapter we assume that the trajectory is optimised and known.

2.2.2 Inverse Kinematics

Many industrial robots are designed in a manner that makes it easier to compute an inverse kinematics solution in a mathematically closed form. But this often restricts design freedom, the robot's flexibility and application range, and the aesthetics of its design.

One popular technique is the so-called *Resolved Motion Rate Control (RMRC) (Whitney, 1969).* It makes use of the fact that the forward function k can be linearly approximated using the Jacobian $J(\alpha)$, which contains the first order partial derivatives of

[5] Independent of the dynamics.

the joint angle parameters:

$$J(\alpha) = \begin{pmatrix} \frac{\partial x}{\partial \alpha_1} & \cdots & \frac{\partial x}{\partial \alpha_n} \\ \frac{\partial y}{\partial \alpha_1} & \cdots & \frac{\partial y}{\partial \alpha_1} \\ \frac{\partial z}{\partial \alpha_1} & \cdots & \frac{\partial z}{\partial \alpha_1} \end{pmatrix}$$

The Jacobian relates changes in joint angle space in accordance with changes in task space. The forward kinematics, for a small time step Δt between actual and desired state can then be expressed as

$$\Delta x = J(\alpha)\Delta\alpha$$

By calculating the inverse of the Jacobian J, the inverse kinematics can be directly calculated for a small time step as follows:

$$\Delta\alpha = J^{-1}(\alpha)\Delta x$$

This makes the assumption that, for sufficiently small steps, the real function can be linearly approximated. Though J is not invertible in many cases, especially for redundant arms and special arm configurations. In cases where the Jacobian matrix is non-square (rectangular) the inverse can be replaced by a pseudoinverse J^+ (Klein & Huang, 1983; Liegeois, 1977; Siciliano, 1990).

$$J^+ = (J^T J)^{-1} J^T$$

But using the pseudoinverse does not solve the problems of singularities in the Jacobian. Singularities occur if, for example, multiple links are aligned in the same direction, which leads to identical derivatives for several joints of the robot arm. Inverses of nearly singular matrices will result in excessively large velocities and therefore cause unrealistic behaviour that oscillates around a singular configuration.

There are many alternative approaches that address the inverse kinematics problem, such as Jacobian transpose methods (Wolovich & Elliot, 1984), quasi-Newton and conjugate gradient methods (Wang & Chen, 1991; Zhao & Badler, 1994), and Levenberg-Marquardt damped least squares methods (Nakamura & Hanafusa, 1986; Wampler, 1986).

2.2.3 Motor Control

After having defined the desired trajectory and the inverse kinematics, the remaining task is to issue the correct motor commands to the joints of the arm. These motor commands produce torques that lead to desired accelerations and therefore to the planned movement

of the arm. Depending on the actual robot, many different actuators can be used to produce joint torques. Robots can be driven by electric, hydraulic or pneumatic actuators, depending on the robot's construction, size and application (McKerrow, 1991). The presented motor control principles from this section are generally valid for common robot arm controllers, independent of the type of actuator used. The following principles will be applied in order to control the DLW III in a simulated environment (Section 5.6).

The simplest control mechanism, *open loop control* (see Figure 2.3), generates motor torques τ based on an idealised function f. It transforms the robot over time (t), assuming constant system conditions c from its actual state α to a desired state $\tilde{\alpha}$.

$$\tau = f(\tilde{\alpha}, c, t)$$

This transition assumes an ideal, noise-free world and a perfect forward model as it does not incorporate the real resulting state after applying the torque τ. If there is a change in system conditions, the forward model behaves incorrectly, leading to erroneous transitions. To mitigate this problem, we can add a mechanism that is aware of the state that results from the application of the motor torque τ. This is referred to as *closed loop control* (Figure 2.4), as the resulting feedback is used to adapt the forward model. This new function which generates the torques now adapts, depending on the outcome α of the desired state as given below:

$$\tau = f(\tilde{\alpha}, \alpha, c, t)$$

Robots usually are equipped with several joint angle sensors, using which we can calculate the error err between a desired and actual state at each time step of the movement. We therefore use the sensor readings and feed the error into our system as feedback:

$$\tau = f(err, c, t)$$
$$err = (\tilde{\alpha} - \alpha)$$

Such a *negative feedback controller* (Figure 2.5) performs corrections on the basis of the error from the previous movement, and does not need to know the forward model anymore. The *PID (Proportional Integral Derivative)* controller, which is widely used in robotics and other engineering disciplines, is based on the principle of negative feedback. The general PID control policy is defined as:

$$\tau = K_p \cdot err + K_D \frac{derr}{dt} + K_I \int (err) dt$$

It introduces three terms, each one of which is multiplied by the gain factors K_P, K_I and K_D respectively. This brings about a change in the correction behaviour of the

feedback controller. The first summand is the *proportional* term. It controls the speed of the robot's correction. If the proportional gain K_P is too small, correction will be slow; a value of K_P that is too large will overcompensate for errors and destabilise the system by causing overshooting. The second term is a product of the gain with the first *derivative*, i.e., the velocities of the system are incorporated. This damping factor helps to reduce the overshoot caused by large position errors. The last summand of the PID control law is called the *integral* term, which integrates up those steady-state errors that occur in the system and cannot be corrected by the proportional and derivative components. The integral gain K_I is important in cases where the original calculated torque cannot lift the arm precisely to the correct position, due to friction and load on the arm. Finding the correct value for K_I is difficult in practice. This is because it accumulates errors over a long period of time. For immediate changes in the arm, this can lead to the application of large torques based on outdated information. This, in turn, can lead to catastrophic behaviour. Our simulation completely ignores the integral component because we have constant physical conditions, no friction in the joints and idealised motors with "infinite" power.

The PD-controller is tuned to strike a balance between accuracy and stiffness. If the robot gets disturbed by an external stimulus like a push, it will apply very strong forces to make corrections. This makes the robot very stiff and dangerous, i.e., not compliant. Small gains, on the other hand, will reduce stiffness and increase the robot's flexibility with respect to external stimuli, though in extreme cases, it may result in sluggish and imprecise movements.

As has been mentioned earlier, we would like to have movements that are accurate but still compliant; but these are conflicting properties in PD control. This problem can be resolved by using a *composite controller* (Figure 2.6) with a feed-forward component that predicts correct movements from the actual state to the desired state, and therefore produce errors of minimal magnitude. This can then be corrected by a low-gain PD component.

Though a combination of negative feedback and feed-forward control can result in compliant motion, a reliable inverse dynamics model is required in order to predict the correct torques. The next chapter addresses this problem and proposes a solution using a learning approach.

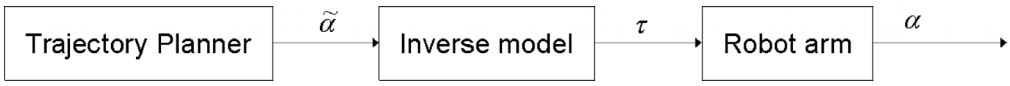


Figure 2.3: Open loop control diagram.

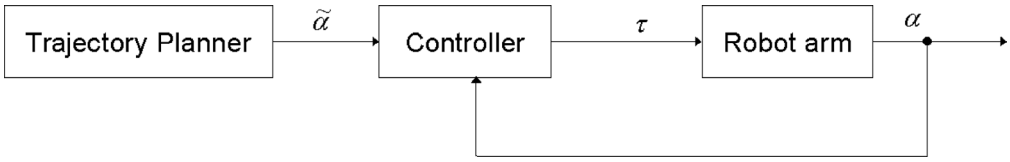


Figure 2.4: Closed loop control diagram.

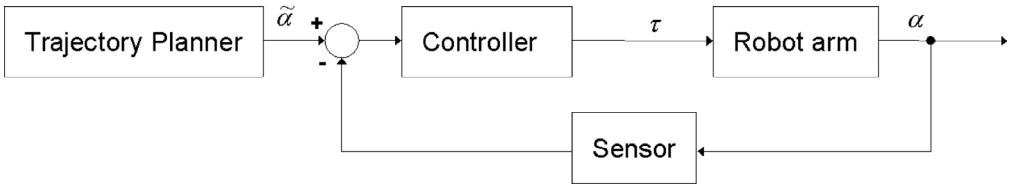


Figure 2.5: Negative feedback control diagram.

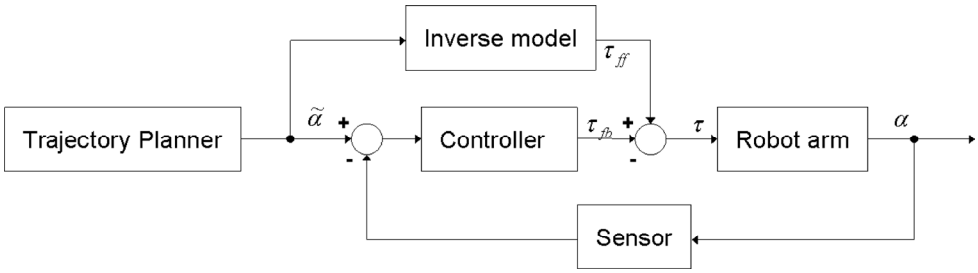


Figure 2.6: Composite control diagram.

3 Learning in Robot Control

In this chapter, we will discuss two approaches to solving the inverse dynamics problem. We will first present an analytical solution to the problem by making *rigid body dynamics assumptions*. Secondly we will motivate the use of learning approaches to solve the dynamics problem, and present the learning algorithm, LWPR.

3.1 Estimating the Dynamics Analytically

In order to estimate the dynamics of a robot system, it is common to make *rigid body assumptions* (Sciavicco & Siciliano, 2000), wherein the robot system is assumed to consist of single stiff bodies, not affecting any other bodies or joints. Each body is assumed to behave in an ideal fashion with frictionless joints and without positional inaccuracies. The description of the system's dynamics, often referred to as the *Joint Space Dynamic Model*, is solely based on the bodies' mass and shape. The joint behaviour is not incorporated into this model.

There are two basic formulations for the dynamics of general rigid body manipulators (Featherstone & Orin, 2000):

On the one hand, there are (Euler) Lagrangian formulations[6], which are widely used to describe the equations of motion (EOM) of a robot. They are based on the Lagrangian equations of the systems' total energy, and can be described in a closed mathematical form.

On the other hand are the Newton-Euler methods[7] , which are based on the balance of all forces that affect the bodies that constitute an arm. This can be described in a recursive formulation for each single body, which then can be assembled in order to describe the complete arm system. The Newton-Euler methods are computationally more efficient. However, parameter estimation in this method is more difficult than it is in the Lagrangian formulations. The Langrangian method is briefly discussed in this section. The discussion is based on (Sciavicco & Siciliano, 2000), unless explicitly mentioned otherwise. For further details on both formulations, including some examples, we refer the reader to (Featherstone & Orin, 2000; Sciavicco & Siciliano, 2000).

The Lagrangian of an arm system can be formulated as:

$$L = E_{kin} - E_{pot}$$

[6] Also called analytic formulations.

[7] Also called synthetic formulations.

Where E_{kin} is the kinetic energy and E_{pot} the potential energy of the system. The Lagrange equations then can be written as follows:

$$\frac{\partial}{\partial t}\frac{\partial L}{\partial \dot{\boldsymbol{\alpha}}_i} - \frac{\partial L}{\partial \boldsymbol{\alpha}_i} = \xi_i, \qquad i = 1,\ldots,n$$

The presented Langrange equations are given in joint angle coordinates, whereas a description in genaralised coordinates is also possible. The number of joints is denoted by n, and ξ_i describes the generalised force associated with the i[th] joint. These forces are non-conservative forces, i.e., joint torques, joint friction torques and joint torques resulting from interaction with the environment. In order to calculate the Lagrangian, the kinetic and potential energy of all the bodies need to be calculated. Consequently the Langrange equations can be computed, and the equations of motion can be derived in matrix form as follows:

$$\mathbf{B}(\boldsymbol{\alpha})\ddot{\boldsymbol{\alpha}} + \mathbf{C}(\boldsymbol{\alpha}, \dot{\boldsymbol{\alpha}})\dot{\boldsymbol{\alpha}} + \mathbf{g}(\boldsymbol{\alpha}) = \hat{\boldsymbol{\tau}}$$

Where:

- $\mathbf{B}_{n\times n}$: Inertia matrix - depending on the arm position.
- $\mathbf{C}_{n\times n}$: Matrix containing Coriolis forces - depending on arm position and velocities.
- $\mathbf{g}_{n\times 1}$: Gravitational forces vector - depending on position.
- $\hat{\boldsymbol{\tau}}_{n\times 1}$: Resulting joint torque vector.

The problem that still remains is to estimate the parameters $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{g}$. To achieve this, simplifications are usually performed by designing robots that have massive and stiff links with very high transmission ratios in the joint motors. This reduces the Coriolis and gravitational effects on the arm, and therefore $\mathbf{C}$ and $\mathbf{g}$ can be neglected. Industrial robot arms are built - as far as possible - to obey the rigid body assumptions and allow good parameter estimation. However, robots like the DLW III are designed to be dextrous and compliant. Its lightweight construction with flexible joints and large degrees of freedom introduce non-linearities in the system, which are not covered by the rigid body assumptions. In addition to that, accurate parameter (e.g. inertia tensor) information is unavailable in many cases, and system dynamics properties may change over time due to material wear or morphological changes to the robot. All these factors would lead to discrepancies between the model and the real dynamics of the robot. In summary, therefore, we can conclude that an analytical estimation of the dynamics is not feasible in the context of humanoid robots like the DLW III.

3.2 Learning the Dynamics

The previous section deliberated upon the issues faced when analytical solutions to the inverse dynamics problem are used. Another way to solve the problem is to use machine learning methods to learn the inverse dynamics model.

A robot arm produces a vast amount of data from sensor readings during its operation. This data can be used by a learning component to learn the dynamics. The learning component retrieves data pairs, consisting of the robot arm's state after executing a specific motor command. The composite control diagram (Figure 3.1) is extended by a learning component, which adapts the inverse dynamics model based on the system's operational behaviour.

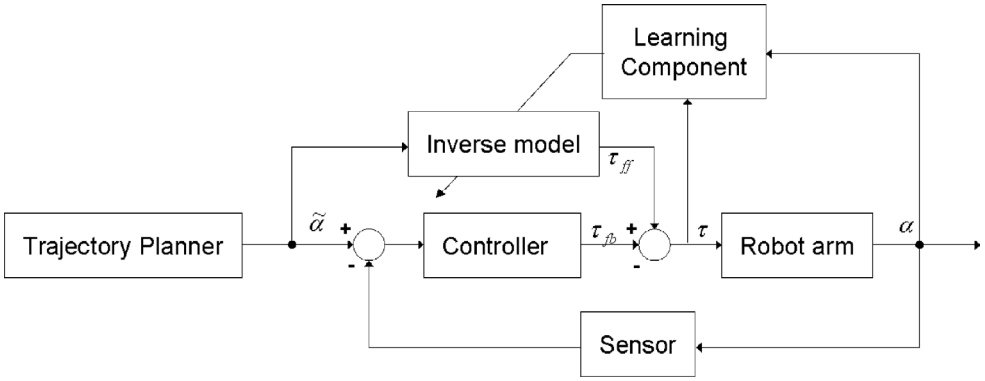


Figure 3.1: Composite control diagram with learning component.

The learning component tries to find a general mapping from a system state to a control vector, and implicitly makes the assumption that such a functional relationship exists between input and output. There is usually no prior information available about the underlying function that has to be learned. The learning thus relies only on the presented data.

The question that remains is what kind of learning approach should be used to learn the inverse model (which predicts the feed-forward commands).

Various machine learning algorithms have been applied to robot control learning problems, and a major distinction can be made between *global* and *local* learning approaches. An example of a global learning method is a neural network with a sigmoid activation function. Such neural networks are characterised by activation functions that respond to inputs from the whole input space, which generally leads to the problem of *negative interference* (Schaal & Atkeson, 1998). If a neural network is trained using data from one specific region in the high-dimensional input space, its future predictions will be accurate in these regions. However, if the system is then trained with new data from another region, the input distribution changes and the parameters are adjusted. This may

result in the loss of the previously learned regions. This problem could be solved by storing all the produced data, and always retraining the network using the complete data set. However, the amount of data produced from robot motion is too large to be stored, and the re-training of the network is computationally not feasible for real time applications. Another issue with global learning arises with the adjustment of meta-parameters such as the number of neurons or hidden layers, which cannot be adapted during learning. This makes global learning inflexible with respect to change in the dynamics of the system.

Local learning methods, in contrast, are characterised by activation functions that are non-zero only in a spatially restricted region of the input space. They represent a function using small simplistic patches - e.g. first order polynomials. The size of these patches is determined by local activation kernels, and the positions and number of the local patches is used to approximate a specific function. Because the input data activates only local patches, local learning algorithms are robust against global negative interference. This ensures the flexibility of the learned model towards changes in the dynamics properties of the arm (e.g. load, material wear, and different motion). A local learning strategy therefore seems appropriate for sequential data streams from different, previously unknown input regions, which is the case in robot motion systems.

3.3 Locally Weighted Learning

Locally Weighted Learning (LWL) approaches are a group of nonparametric learning algorithms that have in the past showed to perform well in the context of online motor learning scenarios (Schaal, Atkeson, & Vijayakumar, 2002).

Robot motion data is characterised by high-dimensional input spaces with many irrelevant and redundant input dimensions. The motion data is produced in high frequencies, and typically has highly non-linear characteristics. However there is usually little prior knowledge about the complexity of the underlying model that is approximated. LWL, with its local nature, allocates the required model resources in a data driven fashion, which makes it very useful for learning online motion data. Besides the beneficial properties of local learning, there are also problems when high-dimensional input data is used. The number of local models required to cover the whole input space "explodes" exponentially with the number of input dimensions. This *curse of dimensionality* (Scott, 1992) can be conquered by reducing the number of input dimensions. It has been shown in the past (Schaal, Vijayakumar, & Atkeson, 1998) that high-dimensional motion data has a low-dimensional distribution locally. Therefore, local models could be allocated on a low dimensional subspace and still make accurate predictions. LWPR extends the classic LWL

framework by finding efficient local projections that are used to perform local function approximation. LWPR thus uses all the beneficial properties of LWL, while at the same time solving the curse of dimensionality by reducing the number of input dimensions.

There are several established dimensionality reduction algorithms, including principle component regression (PCR), factor analysis (FA), and partial least squares regression (PLS).

PLS is a shrinkage method that calculates recursively orthogonal projections of the data and then performs a univariate regression on the residuals of the previous iteration. The big advantage of PLS compared to other dimensionality reduction techniques is that it takes into account the input as well as the output data, as opposed to considering just the input data. PLS achieves this by choosing projection directions on the basis of the maximal correlation between the residual error and the input. PLS further ensures that the projection directions are always orthogonal by performing an additional regression of the previous step against the projected inputs. This leads to optimal regression results with just one projection in cases where the input distributions are spherical. The PLS is known to be the most favourable technique for local dimensionality reduction (in terms of goodness of the results, computational load and robustness) (Vijayakumar et al., 2005). LWPR uses a specially adapted variant of the partial least squares method to reduce dimensionality.

3.4 Locally Weighted Projection Regression

Locally Weighted Projection Regression (LWPR) is an algorithm that belongs to the group of *locally weighted learning* methods (LWL). This section describes the LWPR algorithm based on the description in (Vijayakumar et al., 2005). Later, we will use an implementation of LWPR to learn the inverse dynamics of the DLW III for different motion tasks. For further details, we refer the reader to some of the various publications that discuss LWPR and its applications (Schaal & Atkeson, 1998; Vijayakumar et al., 2005; Vijayakumar, D'souza, Shibata, Conradt, & Schaal, 2002).

LWPR uses the concept of approximating a non-linear function by using many locally linear models. This local linear regression takes into account only query points from a local neighbourhood, which is defined by *receptive fields.* Depending on the complexity of the function region to be approximated, more of less local models are required for performing the approximation. Learning LWPR involves the determination of the right number of local models K, the hyperplane parameters in each local regression model (i.e., slopes), and the determination of the region of validity (i.e., receptive fields) in each local model. The process of allocating new receptive fields and adapting their width and shape is non-competing during training, as a result of which the other receptive fields are not

incorporated. New receptive fields are allocated as they are required to perform the local regression. Fields that are not required get pruned. The receptive field in an LWPR is modelled as a Gaussian kernel, and the associated weight of a query point x can be represented as follows:

$$w_k = \exp\left(-\frac{1}{2}(\mathbf{x}-\mathbf{c}_k)^T \mathbf{D}_k (\mathbf{x}-\mathbf{c}_k)\right)$$

where $\mathbf{c}_k$ is the centre of the k[th] linear model and $\mathbf{D}_k$ is the *distance metric.* $\mathbf{D}_k$ determines the shape and size of the region of validity. As mentioned earlier, no competition between the receptive fields takes place during training. However, during prediction, the total systems output $\hat{y}$ for a query point $\mathbf{x}$ is built using the normalised weighted sum of all K linear model predictions $\hat{y}_k(\mathbf{x})$:

$$\hat{y} = \frac{\sum_k w_k \hat{y}_k}{\sum_k w_k}$$

Figure 3.2 summarises the basic process of LWPR learning. During training, the high dimensional input is reduced by applying an incremental version of PLS. This happens independently between the local models, and the dimensionality reduction rate may vary in each model depending on the input data. In the next step, a linear regression is performed on the lower dimensional data. Furthermore, each local model is associated with a receptive field. The receptive field gives a corresponding weight, which defines the region of validity of the local regression for each local model prediction. The complete system output is then built by taking into consideration the weighted average of each model prediction.

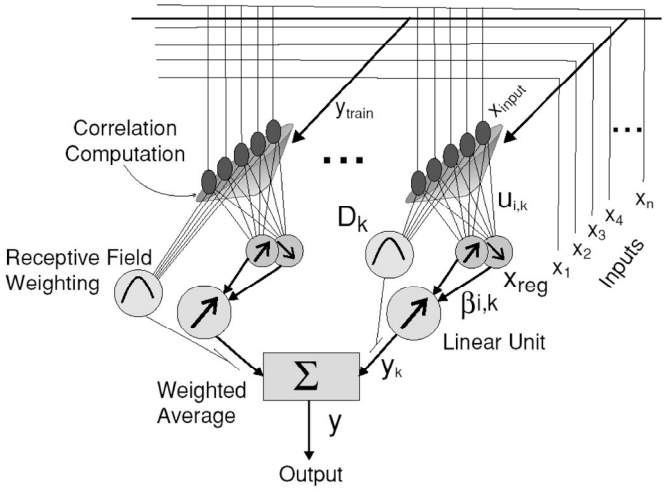


Figure 3.2: LWPR learning scheme (Reproduced from (Vijayakumar et al., 2005)).

It was previously mentioned during the course of this thesis that for online scenarios it is not desirable to store all of the received data. So the algorithm must be able to perform the projections, local regressions, and distance metric adaptations in an incremental fashion without storing the data. Therefore LWPR keeps a number of variables that hold *sufficient statistics* for the algorithm to perform the required calculations incrementally. These sufficient statistics also contain a forgetting factor ($\lambda \in [0,1]$), which allows forgetting of older received data.

The local validity of each model, i.e., the distance metric $\mathbf{D}_k$, can be updated by optimising a cost function J using a gradient descent based method.

$$\mathbf{D}^{n+1} = \mathbf{D}^n - \alpha \frac{\partial J}{\partial \mathbf{D}}$$

$$J = \frac{1}{\sum_1^M w_i} \sum_1^M w_i (y_i - \hat{y}_{i,-i})^2 + \frac{\gamma}{N} \sum_{i,j=1}^N D_{i,j}^2$$

M represents the number of training data points. The first term of the cost function J corresponds to the mean of the leave-one-out cross validation error of the local model. It is a generalisation factor that prevents overfitting. The second term is a regularisation term on the distance metric. It penalises large values in $\mathbf{D}$ (i.e., narrow kernels) and therefore prevents the receptive field from shrinking to an infinitesimally small width. The above presented cost function J can be reformulated by replacing the leave-one-out-cross validation error as PRESS residual errors, which allows direct (incremental) error calculation. Details about the incremental cost function can be found in (Vijayakumar et al., 2005).

During learning with LWPR it is necessary to add new receptive fields, for unknown regions of the input domain. For new incoming data LWPR decides, whether to add a new receptive field, depending on the activation of all allocated receptive fields. If, for a new input data point x, no existing receptive field is activated more than a minimal threshold (w_{gen}), then a new receptive field is allocated with centre at $c = x$ and a default distance metric $D = D_{def}$. The initial value of D_{def} has a major influence on the time performance of LWPR. Wide[8] initial receptive fields are desirable, because they span a wider range of the input domain. Therefore they will be activated by more input data points and shrink faster to an appropriate size.

[8] Not excessively wide.

The final outline of the LWPR algorithm is shown in the following pseudocode:

- Initialise the LWPR with no receptive field (RF)
- For every new training sample (x,y):
    - For k=1 to #RF:
        - Calculate the activation w_k
        - Update projections, regressions and distance metrics
        - Check if number of projections needs to be increased, by monitoring the change in the Mean Squared Error of the regression.
    - if no RF was activated by more than a minimal threshold (w_gen), then create a new RF

Table 3.1: LWPR algorithm outline (Adapted from (Vijayakumar et al., 2005)).

The discussed LWPR algorithm is capable of approximating functions between high dimensional input and output data, as it is produced from robot motion tasks. It can approximate non-linear functions without any prior knowledge, since it uses data from a local neighbourhood only, while not competing with other models. The major strength of LWPR is that it uses incremental calculation methods and therefore does not require the input data to be stored. This allows LWPR to learn over an unrestricted period of time. Furthermore the algorithm can cope with highly redundant and irrelevant data input dimensions, because it uses an incremental version of PLS, which automatically determines the required number of projections and the appropriate projection directions. In summary, we can say that LWPR is a suitable algorithm for learning the dynamics of a robot arm.

4 Physical Simulation

This chapter discusses the fundamental concepts of physical simulation, as the robot arm simulator implemented as part of this project uses physical simulation.

The goal of a physical simulation is to mimic the physical world using a computer. In the past few years, several proprietary and open-source physical simulation frameworks, more popularly known as *physics engines*, have been developed. They are widely used in 3D computer games, engineering and medical applications, and research simulations. The physics engine models the object's behaviour in the presence of gravitational and frictional forces, and other physical parameters. Depending on the actual implementation, particles, rigid bodies, and even deformable bodies like clothes or even liquids can be simulated. The two main areas where physics engines are used are scientific computing, and computer graphics and animation. In the former, real-time performance is not of critical importance, whereas in the latter, appearance and real-time performance are of primary importance. However, in scientific computing, the computational load can be enormous, and physical plausibility has to be ensured, which is not the case in computer graphics and animation. Developing an optimal solution requires striking a balance between the two aforementioned extremes.

We require a simulation that allows us to model correct physical behaviour on a higher level, which mainly means gravitation, friction and object collisions. For the robot arm, we use an articulated rigid body dynamics simulation by making the assumption that objects are non-deformable and cannot interpenetrate. In the real world, however, no perfectly rigid bodies exist as minimal distortions and bending are unavoidable. However this assumption simplifies the simulation drastically and makes the modelling of simple mechanical objects accurately in real-time possible.

A rigid body simulation consists of two main parts: the *dynamics simulation* itself and *collision detection*.

Linear and angular velocities and accelerations are modelled in the dynamics simulation. This could be an object accelerating due to gravitational forces or decelerating due to inertia.

The collision detection component prevents objects from interpenetration. *Open Dynamics Engine (ODE)* has both components built in one framework. This helps in the development of powerful rigid body simulations (Russell Smith, 2006). The rest of this chapter reviews the basics of rigid body physical simulation, as it is used by ODE.

4.1 Rigid Body Representation

Rigid bodies can, in general, be classified into non-polygonal and polygonal models. It is necessary to distinguish between non-polygonal (for instance, boxes and spheres) and polygonal objects (for instance, triangulated meshes), and whether they are convex or non-convex (see Figure 4.1).

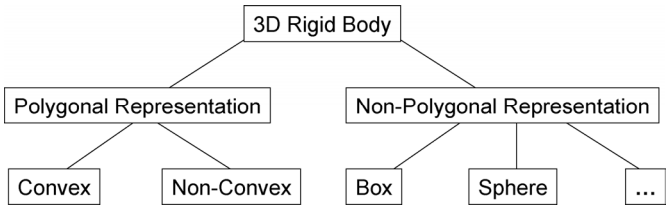


Figure 4.1: Basic distinction between 3D rigid body representations.

Either a polygonal or a non-polygonal rigid body representation is chosen, mainly on the basis of the purpose of the simulation. Basic primitives can be described mathematically in a closed form and therefore allow for faster computations like collision detections, motion and visualisation calculations. The variety in shapes - and consequently, the object modelling possibilities - is, however, restricted. In practice, the right trade-off between exactness and computational performance needs to be evaluated on a case-by-case basis. A rigid body has two internal representations in a computer: the first is used to calculate the physical behaviour, and the second is used to perform 3D visualisations. Though both representations can be treated as the same, but can also be setup separately. For example, one can choose a highly detailed polygonal object for visualisation, and a simple bounding box for collision detection and motion calculations.

4.2 Constraints

An idealised scenario is a rigid body simulation in an unconstrained world, in which object can move and do not have to care about collisions and spatial restrictions. Given some external forces, the motion caused by the object's response to the aforementioned forces can be calculated numerically by solving coupled differential equations. However this scenario is not realistic and does not consider interaction between objects. Therefore two constraints, namely the *collision constraint* and the *joints constraint*, are usually applied to rigid body simulations.

Object collision detection is an important concept as it is the only natural way to make objects interact with one another. Collisions can occur due to short contacts (e.g. colliding billiard balls) or due to resting contacts between bodies (e.g. an object lying on the floor).

Unlike collision constraints, joint constraints restrict the motion between bodies at all times during the simulation. The latter is a very powerful concept for creating connected rigid bodies, such as robot arms. One can imagine a joint constraint as a friction-free and zero-mass infinitesimally small mechanical unit, acting as real joints do (see Section 4.4).

4.3 Physics Engine Architecture

A physical simulation consists of many consecutive snapshots, also referred to as *frames,* which hold the physical state of the system at a given point in time. Each frame holds the actual state, usually position and velocity, of the rigid body system. These frames, taken in series, form a simulation that represents the dynamic behaviour of the system over time. The phase in between two physical frames is called a transition phase, during which the engine calculates the next states of the system. The time step T between two frames serves as the resolution parameter of the simulation. If the step size is small, the simulation becomes more precise, but also requires more calculations per "real world second" of the simulation. A perfect simulation would require an infinitesimally small value of T. In practice, we often choose T to be equal to 0.01. If faster motions of objects are expected, the time step should be reduced in order to make sure that object collisions are resolved.

The physical simulation process is usually formalised using the *simulation loop* (Mirtich & Canny, 1995), which consists of three states:

1. Discovery of new object positions.
2. Collision detection.
3. Response to collisions.

In practical implementations, step 1 and 3 are merged together in a simulation module, and step 2 is performed by a separate module. The modularisation of collision detection and simulation is done in order to make a clear separation between the purely geometric problem of collision detection and the simulation (which is responsible for computing the motion of the objects in the system).

Collision detection is computationally the most expensive part of the simulation loop. It is called when the simulation module returns the actual positions of all objects in the system after the last simulation step. In order to compute all the points of contact, the collision detection module examines the geometry of every object in the system.

The Simulation module itself can be subdivided into four modules (Erleben, 2004):

Time-Control Module: This is the central time coordination unit that initiates and synchronises each time step between the Motion Solver, the Constraint Solver and the Collision solver.

Constraint Solver: Calculates the constraints in the system that evolve from collisions and joints. It gets the contact points from the collision detection module and calculates the resulting object impulses by calling the *Collision Solver*. The results of the Constraint solver are then directed to the Motion solver, which incorporates the constraint forces in order to calculate the resulting motion of the system.

Motion Solver: Calculates the movement of every object in the system using the equations of motion, which are given as ordinary differential equations (Barraff & Wittkin, 1997).

Collision Solver: Is called by the Constraint solver to calculate the impulses of all colliding objects.

This presented architecture outline represents a generally valid approach, even if the actual algorithms used vary between different implementations.

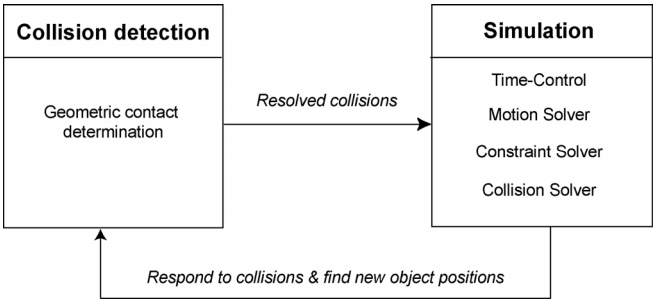


Figure 4.2: General physics engine architecture (Adapted from (Erleben, 2004)).

4.4 Rigid Body Simulation with ODE

Open Dynamics Engine (ODE), created by Russel Smith (R. Smith, 2001), is a free, industrial-quality library for simulating articulated rigid body dynamics. It is probably the best known and most commonly used free physics engine. ODE is especially well suited for real-time simulations, since it was designed particularly for real time purposes, with an emphasis on speed and stability rather than physical accuracy.

ODE consists of a large number of different objects and functions, the most important of which are summarised below (Russell Smith, 2006):

ODE uses four major objects to represent physical simulation - Worlds, Rigid bodies, Joints, and Spaces.

Worlds in ODE are object containers. Objects placed in different worlds cannot interact with each other. Therefore, most ODE simulations use only a single world. A world has global properties like gravity parameters and error reduction factors (see later) that can be set by the user.

Rigid bodies are the main construction block of a system of objects. ODE version 0.5

supports *boxes*, *spheres*, *capped cylinders* and triangulated *meshes*. A rigid body is characterised by a point of reference (x, y, z), which is the position of the body's centre of mass in world coordinates. It also contains a velocity vector (vx, vy, vz), an angular velocity vector $(\omega x, \omega y, \omega z)$, and the orientation of the body represented as a quaternion[9] (qs, qx, qy, qz). These state parameters may change over time. There are also fixed body parameters such as the body's mass m, the centre of mass relative to the body $(\tilde{x}, \tilde{y}, \tilde{z})$, and the inertia matrix (tensor) $I_{3\times3}$, which quantifies the rotational inertia of a rigid body.

The user also can specify constraints in the form of *joints*. Three joint types are available: *Universal joints, hinge joints,* and *slider joints* (see Figure 4.3). A fourth type is the fixed joint, which is a rigid link.

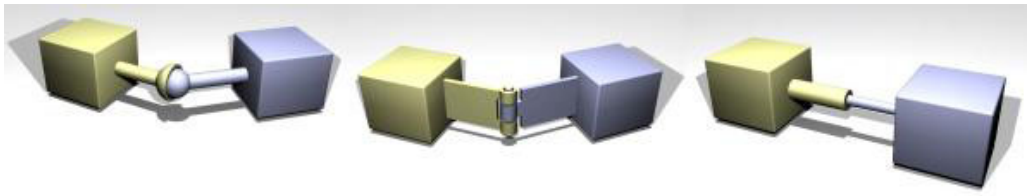


Figure 4.3: ODE joint types visualised as mechanical units. From left to right: universal joint, hinge joint, and slider joint. Note that joints in ODE have no spatial extent, friction or mass (Image reproduced from (Russell Smith, 2006))

For each simulation step, every joint applies *constraint forces* to the bodies they are connected to, forcing the system to move solely towards the allowed directions. These restrictions, and the numerical imprecision accumulated over time, can result in *joint errors* creeping into the simulation. ODE has a mechanism controlled by the *error correction parameter (ERP)* that reduces this error after each simulation step. The ERP, a value between 0 (no correction) and 1, must be evaluated and set explicitly by the programmer.

As stated before, ODE has an integrated collision detector that implements collision detection. The user has to specify in his program, which objects should be checked for collisions. A naïve approach would check each body with every other body in the system, and call the collision check function `dCollide` $O(n^2)$ times, where n is the number of bodies. *Spaces* are again a kind of container, into which the programmer can add bodies that he wants to check for collisions. So the `dCollide` function is called only once per collision group (i.e. space). Furthermore, this function is optimised to operate on spaces, and therefore speeds the collision detection process up. The ODE architecture also allows

[9] A more compact and accurate representation than a rotation matrix.

the integration of external collision detection systems.

The following pseudocode gives an example of a typical ODE simulation procedure (Russell Smith, 2006):

```
1. Create a world and set world parameters.
2. Create bodies in the world.
3. Set the state (position etc) of all bodies.
4. Create joints in the world.
5. Attach the joints to the bodies.
6. Set the parameters of all joints.
7. Loop:
       a. Apply forces to the bodies as necessary.
       b. Adjust joint parameters as necessary.
       c. Call collision detection.
       d. Take a simulation step.
8. Destroy the dynamics and collision worlds.
```

For robot arm control using joint torques, step "a" of the presented ODE loop is used to apply the appropriate forces directly to the joints. ODE also allows the user to set joint positions and velocities directly, whereas in our robot control paradigm they play a minor role. An important issue with setting joint torques directly is that ODE has no control mechanism to limit the applied forces. This can result in unrealistic joint constellations, which in turn often leads to joints gaps in rigid body chains, or even complete displacement of the rigid bodies all over the space. The ODE user must always keep this limitation in mind, and additionally remember that the joint can become "elastic" if the wrong forces are applied. This can produce "useless" data because it falsifies the real desired mechanics of the arm. Figure 4.4 demonstrates such a case, which shows a robot arm following a trajectory (compensating with PD gains that are too large).

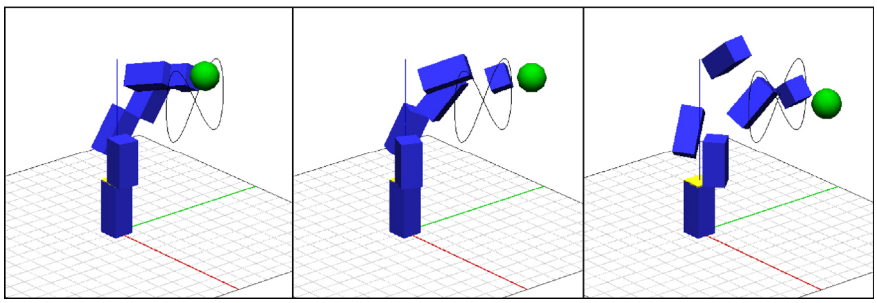


Figure 4.4: A robot arm performing a motion task with increasing joint forces (from left to right). If the forces get too large, the arm structure first builds gaps and finally breaks apart completely, leading to mechanically unrealistic results.

5 Implementation of a Robot Simulator

This chapter describes how the robot control mechanisms, LWPR and ODE are combined to develop a flexible simulation framework for robot arms. The chapter begins with a look at available software, and then defines the functional requirements of the simulator. This is followed by a discussion of the implemented components required for the final simulator developed as part of this project. This chapter also includes a brief overview of the other concepts, besides learning and motor control, that were required in order to realise the implementation. It concludes with a presentation of the simulator architecture that incorporates all the components presented in previous sections.

5.1 Existing Software

The *MT library* (Toussaint, 2005), originally developed by Marc Toussaint, served as the basis for the simulator. It is written in C++, and offers interfaces to *OpenGL* (Schreiner, 2004) and ODE. It contains the *Dynamic Configuration Graph* (*DCG),* a generic data structure describing the dynamic state of a connected rigid body system. The DCG-file format provides the programmer with the ability to define a flexible specification of rigid body systems and its dynamic state by using basic primitives (boxes, spheres and capped cylinders) and its joint connections (fixed, hinge, slider and universal joints). The DCG-file information then can be loaded into ODE and rendered using OpenGL. The MT library delivers a performance-optimised array container class for storage and maintenance of motion data, as well as for many useful functions such as setting and getting joint states, velocities and forces.

For this project many parts of the library, which has a substantial code base of 20,000 lines, had to be changed or improved; major extensions were made to the DCG structure, the 3D rendering and navigation. This modified MT library remains compatible with older versions.

5.2 Requirements and Functionality

The functionality of the simulator is mainly driven by the task of learning the inverse dynamics of an arbitrary robot arm:

Test mode - The user usually starts off by designing a robot arm in the DCG file that contains all the bodies, joints and physical parameters in the system. A general viewing mode enables the user to evaluate the basic setup, the initial arm position, and the reach of the robot arm. It remains the user's responsibility to somehow create a task-specific

trajectory, and to operate the arm within its kinematic boundaries. The test phase can be used to load trajectories from files and check them in terms of the arm's reach, tracking speed and other task specific properties. The trajectory can be drawn in task space as an orientation point.

PD tuning mode - After the geometry of the arm has been evaluated and a desired trajectory has been created and checked, the appropriate PD gains for motor control can be evaluated in the tuning mode. The PD gains are specified in a parameter file, together with other parameters for the robot arm and the 3D scene. The correct values of the PD controller can be adapted manually by observing the trajectory tracking performance during operation, or by storing the arm states into files (in joint angle space or task space) and analysing them in Matlab.

Creating LWPR models: This phase is used for LWPR batch learning. The user takes the recorded joint angles α, velocities $\dot{\alpha}$, accelerations $\ddot{\alpha}$, and torques τ from the PD tuning phase and creates a LWPR model. The LWPR parameters are stored in a separate *configuration* file, since these can be applied to several different robot arms. This configuration file is part of the MT library, and is by default called `MT.cfg`. The created LWPR model can also be saved as a *LWPR model file*.

Motor control and online learning: The final step in the simulation involves performing actual arm control using a composite controller with learned inverse dynamics model. During operation, this mode of control can be switched with keyboard inputs between open loop, feedback, and composite control. The user also has the option to define whether data recording should be done during operation, and if he wishes to learn LWPR online during operation. Furthermore, the user is given the ability to specify if he wants to follow a desired trajectory or if he wants to perform a visual servoing task using keyboard inputs to locate the target.

5.3 Triangulated Meshes

In order to be able to design arbitrarily complex robot arms of any shape it is not sufficient to use spheres, boxes and capped cylinders as objects. For building a precise model of the DLW III, it is essential to use triangulated meshes (trimeshes). This is because the single parts are non-convex, which cannot be approximated properly using boxes. It is helpful to use more complex objects with interesting shapes, especially for setting haptic experiments up in the future. We therefore took the effort to extend the basic primitives of the DCG by incorporating trimeshes which enable the accurate simulation of any robot. The integration of trimeshes affects all levels of the simulator (see Figure 5.1).

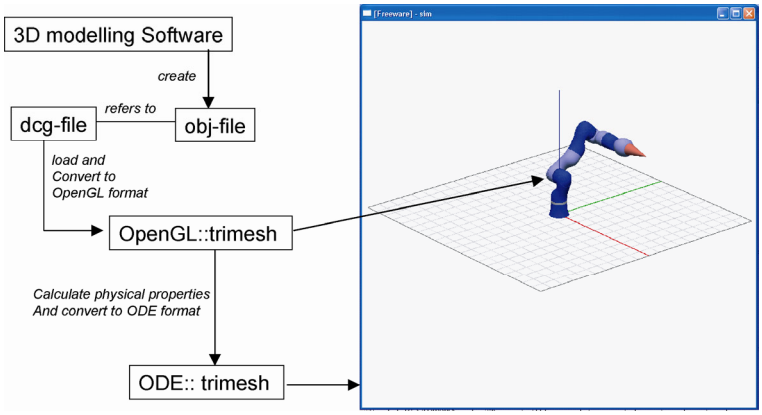


Figure 5.1: Components required for trimesh support. The modelling of the trimesh is left to the user.

5.3.1 Graphical Representation

The user is expected to create the required 3D models by using 3D modelling software and export it to the *Wavefront obj-format*. This is a widely used text-based 3D file format, supported by most modern graphics software packages. The format supports polygonal objects defined by points and faces, as well as free-form objects defined as parameterised curves and surfaces. Only polygonal objects representation will be used in the presented simulator. The obj-file syntax can be easily parsed, and errors can be detected and corrected efficiently.

In order to load the vertices and faces from an obj-file, we developed a file parser, based on a previous C implementation (Robins, 2000). The following example of a box with unit length given in obj serves the purpose of giving the reader an idea of the basic structure of an obj-file. The "v" tag defines a vertex followed by x, y and z coordinates. The "f" tag describes a triangle face, defined by three vertex indices. In the box example in Table 5.1 the first face is defined by the first, second and fourth vertex.

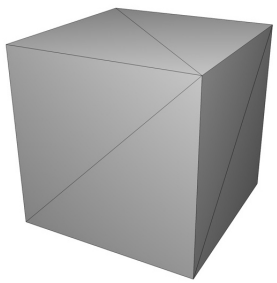
#vertex data	#face data	
v -0.5 -0.5 0.5	f 1 2 4	
v -0.5 0.5 0.5	f 3 4 6	
v 0.5 -0.5 0.5	f 5 6 8	
v 0.5 0.5 0.5	f 7 8 2	
v 0.5 -0.5 -0.5	f 2 8 6	
v 0.5 0.5 -0.5	f 7 1 3	
v -0.5 -0.5 -0.5	f 1 4 3	
v -0.5 0.5 -0.5	f 3 6 5	
	f 5 8 7	
	f 7 2 1	
	f 2 6 4	
	f 7 3 5	

Table 5.1: 3D Box example in obj-format

The complete obj specification covers many more features such as object groups, vertex normals, materials and texture coordinates that may be incorporated into the simulator at a

later point. A detailed specification of the Wavefront obj-file format can be found in (Reddy, 2006).

The parsed object is internally represented as a trimesh class, which consists of a set of vertices with corresponding face normals and vertex normals, as well as a set of triangles, each stored as a point indexed triplet. This class contains all the data structures and functions that are required to calculate parameters for rendering in OpenGL and physical simulation in ODE.

```
class TriMesh{
public:

  typedef struct _triangle {
       int vindices[3]; // array of triangle vertex indices
       int nindices[3]; // array of triangle normal indices
       int tindices[3]; // array of triangle texcoord indices
       int findex;      // index of triangle facet normal
       . . .
  }triangle;

  char*    pathname;    // file path to this model

  MT::Array<triangle> T; // triangles (given as index triplets)
  uint numtriangles;         // number of triangles
  MT::Array<Vector> V;  // vertices
  uint numvertices;   // number of vertices
  MT::Array<Vector> Vn; // vertices' normals
  uint numnormals;    // number of normals
  MT::Array<Vector> Tn; // face normals
  uint numfacenormals;      // number of face normals
  uint   numtexcoords;  // number of texcoords in model
. . .
}
```

In order to be able to draw the trimesh in OpenGL, we have to calculate the *face normals* as well as the *vertex normals* first. The face normals are used to calculate the shading of the object in flat shading mode, and more importantly, for collision detection in ODE. The flat shading is a very simplistic way of representing objects in 3D, and does not look very appealing. In order to achieve smooth surfaces a large number of vertices are required, which results in a drastic reduction in performance, especially for physical collision calculations. This is an important issue as we want the robot arm to be able to learn LWPR online, which is a computationally intensive task. Small triangles are documented to be prone to errors in the context of collision detection in ODE (Russell Smith, 2006). In other words, the bigger the vertices of a model, the faster and more stable the simulation. The OpenGL smooth rendering mode uses the calculated vertex normals to render in smooth shading mode, which in turn renders smooth surfaces even with low polygon meshes. OpenGL interpolates the colour values calculated at each vertex according to a modified version of the Phong illumination model (Blinn, 1977; Phong, 1975). For this algorithm a function that calculates every vertex normal of the trimesh is required. This is achieved, for

each vertex of the body, by building the normalised sum of all face normal-vectors that adjoin the specific vertex.

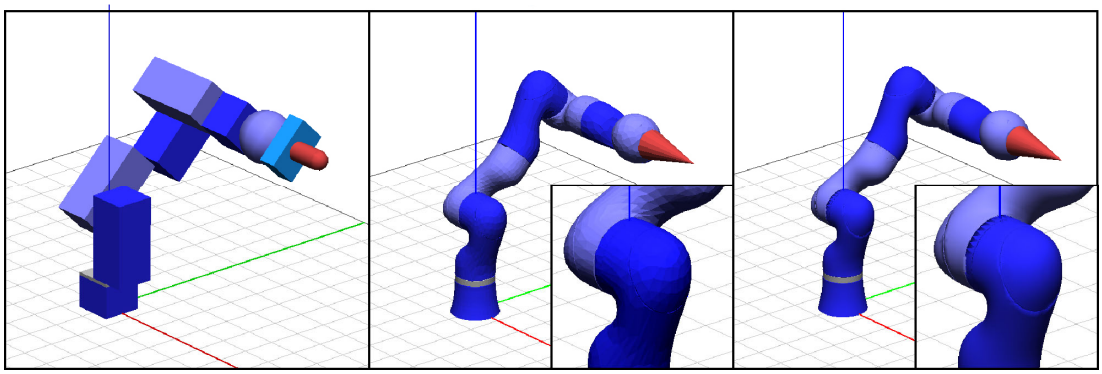


Figure 5.2: Different representations of objects. Left: DLW III made using basic primitives; Middle: Trimesh rendered using flat shading; Right: Same trimesh rendered using smooth shading.

Figure 5.2 shows the presented object representations rendered in the simulator window. From a performance point of view, the non-trimesh version of the DLW III (left) is clearly the fastest. It takes 12.5 seconds to track a reference trajectory, whereas with the flat-shaded trimesh arm (middle), it takes 15 seconds to track the same trajectory. On an average over multiple trajectories, the flat-shaded trimesh arm proved to be about 25% slower than the non-trimesh arm. The smooth shaded trimesh (right) does not perform significantly worse in comparison with the flat-shaded trimesh (performance is about 5% lower) and is therefore used throughout this project.

5.3.2 Physical Representation

In order to use the trimesh class in ODE representation, some transformations have to be performed because ODE has its own list-based representation that does not use triplets. However, the trimesh information is only used for collision detection calculations. To formulate the dynamics of the rigid body system, two key parameters have to be calculated - the linear momentum L and the angular momentum H, which are defined as:

$$L = mv$$
$$H = I\omega$$

Here v is the linear velocity and ω is the angular velocity of the body's centre of mass. The inertia tensor is represented by I. The key quantities to calculate are the object's centre of mass, its mass, and its inertia matrix. They can be easily calculated if the volume V of the trimesh is known.

$$V = \int_{trimesh} dV$$

ODE has no function that calculates V for a trimesh. As a result of this, we had to implement this functionality. The approach taken tries to successively reduce the volume integrals to simpler integrals (Mirtich, 1996). This is achieved by first using the Divergence Theorem (i.e., Gauss' Theorem) to reduce the volume integral to a sum of surface integrals for all triangles in a trimesh. These surface integrals are then treated as planar projections of the surface, on which Green's theorem can be applied. This further reduces each planar integral to a sum of line integrals around the edges of a polygon. Thus, this process reduces a volume integral to a set of line integrals, the values of which can be propagated back to the volume of the trimesh. Our implementation is based on a implementation that used a different polyhedral structure (Mirtich, 1996).

The presented approach requires closed trimeshes, which means they can have no holes or gaps on the surface. CAD constructions from robot parts do not usually satisfy these requirements. We had to deal with this issue while constructing our arm model from the original CAD of the DLW III. Unfortunately there is no automated method that can be used to close the gaps and reduce the number of polygons. This had to be done laboriously by hand using a 3D modelling tool.

A major limitation with the aforementioned calculation of the mass properties of a trimesh is that it assumes a uniform mass distribution. In reality, especially for complex robot parts with motor gears, this is, in many cases, a wrong assumption which falsifies the dynamic behaviour of the simulation. Therefore, we extended the trimesh class with a function that allows the user to set the above calculated parameters by specifying it directly in the DCG-file. For the DLW III, we received the detailed construction plans from DLR, which allowed us to use the correct physical parameters and build as close an approximation as possible to the real arm. During simulation, we found it remarkable that the real physical parameters lead to much more "realistic" behaviour. It was observed that in certain regions, the arm operate better than it does when in other regions. This was found to depend on the joint constellation and weight distribution. In contrast to the arm with uniform mass distribution, the motions of this arm were found to be much more "idealised". However it must be noted that the above observations are only qualitative, and are consequently hard to prove without access to the real robot arm.

There are caveats that must be borne in mind when trimeshes are used. ODE has a reported weakness in determining collisions when trimeshes are used. It is reported (as well as observed) that trimesh-to-trimesh collisions and trimesh-to-box collisions work comparatively reliably, whereas trimesh-to-plane collisions do not work at all. In the ODE

user guide (Russell Smith, 2006), it is stated that the trimesh class is under further development to improve performance. As a result of this, the decision on whether to use trimeshes, what resolution they should have, and with which objects they are likely to collide with, is one that must be considered carefully.

5.4 Dynamic Configuration Graph

It has been shown so far how individual body parts and objects can be represented either as polygonal or non-polygonal objects. These bodies have to be connected by joints. Each body has an initial frame (coordinate system), which is usually located at the centre of mass. This frame helps to identify the object's orientation in world coordinates (world frame), as well as the orientation between different objects. Joints also have an initial frame that defines the joint's position, orientation, and rotational axis. Figure 5.3 shows the building blocks of a simple, "one degree of freedom" robot arm. Each object has its own coordinate frame, which can be connected by defining a series of transformations.

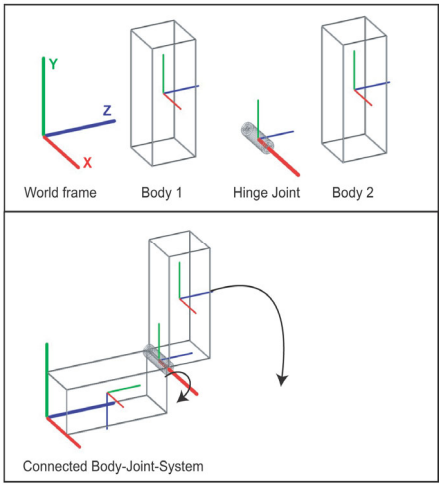


Figure 5.3: Two bodies and a hinge joint that rotates around its x-axis can create a model of a simple robot arm. Note that the joint (in ODE) has no spatial extent and is shown merely for the purpose of visualisation.

In order to construct and maintain such a body-joint system during simulation, it is beneficial to have a general description structure that is efficient and flexible to changes. The DCG is a data structure that enables the user to control the dynamic properties of rigid body systems. It is derived from a generalised template of a graph class that is built out of nodes and edges. Bodies and edges can be general objects such as structs or classes. In our case bodies (`DCBody`) are used as node objects, and joints (`DCJoint`) are used as edge objects, thus enabling the DCG graph to represent articulated body systems as presented above.

`DCBody` contains the required attributes to describe a rigid body in OpenGL and ODE. These are general properties like the body coordinate frame, mass properties, name, body ID, and colour, among others. All these attributes can be loaded from the DCG-file with a read function, which is developed in a manner that allows an easy extension to new attributes. The `DCJoint` class is built in the same way, just with joint specific properties and functions. The most important difference is that the joint object holds information on how and where the two bodies are connected with respect to each other. This is maintained using transformation tags. The whole robot arm structure and all body and joint attributes are specified in a DCG-file, a text-based tag file. The table in Appendix B explains the DCG tags. Additionally Appendix C lists the specification of the DLW III with trimeshes and correct physical parameters[10], as it was used throughout this project. The simulator replaces the hand present in the real DLW III with a cone as can be seen in Figure 5.2 on page 32.

5.5 Trajectories

A trajectory in a simulated environment is nothing but a sequence of points in space usually stored as an array, in which the rows indicate the different values at different times. ODE then reads the data out row-by-row and steps through the points of the path either kinematically, by setting the joint positions directly, or dynamically, by applying joint forces specified by some control law. Closed trajectories are often produced in order to create repetitive motion patterns, and to efficiently collect multiple data sets from the same motion. For other tasks like the visual servoing task, open-ended trajectories are employed, and direct connections to the target are aimed for.

Trajectory data may also be produced from different sources. It can be created artificially from functions, or it can be recorded from human motion. Independent of the data generation process, the resulting trajectory should satisfy some important criteria. The trajectory must be within the arm's reach at any time and the motion should preclude the possibility of self-collisions or collisions with other objects. This ensures that the recorded data is reliable in terms of the desired trajectory, except in cases where explicit collisions are desired. Furthermore, the trajectory must be as smooth as possible and contain no large positional gaps. Since the velocities and accelerations along a trajectory can be calculated by differentiation, alternating position jumps will lead to larger velocities and excessively

[10] Physical parameters are omitted due to copyright regulations.

large accelerations. Over each discrete time step in ODE, these values may vary greatly, thus resulting in data that is possibly "very noisy". The general rule is to create smooth trajectories with many data points, leading to smoother velocity and acceleration profiles.

At the lowest level, a robot arm is always controlled by joint angle motor commands. A common trajectory generation approach works directly using joint angle coordinates and creates artificially sinusoidal values of joint angles over time.

$$\alpha(t) = A\sin(ft + p)$$

The resulting motions are usually characterised by smooth position, velocity and acceleration profiles, and guarantee that the trajectory is moving always within its reach. By varying the frequency f, amplitude A and phase p over time in all joints, various movements in a large area of the arm's reach can be simulated.

Planning in task space on the other hand is intuitive. It, however, requires an inverse kinematics solution for the robot arm. A popular trajectory defined in task space is the 8-shaped trajectory. This trajectory is useful because it has a simple mathematical definition and covers a relatively large area in task space. It also contains regions with low and high velocity.

$$x = const$$
$$y = 0.5\sin(t\pi)$$
$$z = 0.5\sin(2\pi t)$$

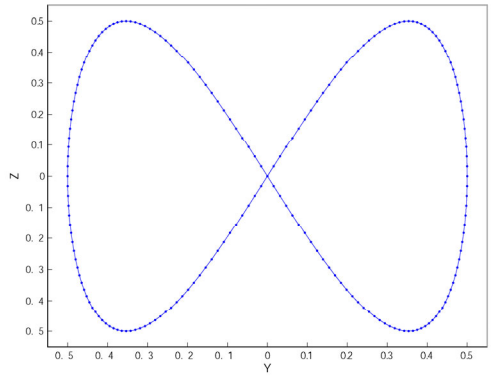


Figure 5.4: Figure eight (infinity) trajectory.

5.5.1 Inverse Kinematics in ODE

ODE itself delivers no inverse kinematics for articulated bodies. But it provides the information required to calculate such a solution. The implemented inverse kinematics in our simulator uses the mentioned method of inverting the Jacobian to approximate the solution for a small time step. This incremental method "translates" a given task-space

trajectory relative to a specified robot arm into a joint-angles trajectory. Following pseudocode explains the inverse kinematics algorithm.

```
1) Trajectory = load(trajectoryFile)
2) endEffector = getDesiredBody(bodyNumber)
3) endEffectorOffset = setDesiredBodyOffset(x,y,z)
4) X = getActualJointStates()

5) for (i=0; i<Trajectory.length();i++)
{
       a) J = calculateJacobian(X, endEffector, endEffectorOffset)
       b) invJ = inverse(J);
       c) deltaX = invJ* (Trajectory(i)- lastPosition);
       d) X = X + deltaX;
       e) lastPosition = forwardKinematics(X,endEffector,endEffectorOffset)
}
```

The user specifies a desired trajectory in task space, and the body of the arm that should act as end-effector[11]. The end-effector offset is used to define an offset relative to the body's coordinate frame, since it is initially located in the centre of mass. Finally, the user sets the initial joint angle positions, which are specified in the DCG-file.

The algorithm loops through the trajectory (step 5) and first calculates the Jacobian (step a) of the end-effector position with respect to every joint. In order to determine the Jacobian, the algorithm calculates the forward kinematics stepwise for each joint, starting with the lowest one. The algorithm applies small changes in each joint angle and relates it to changes in the end-effector position. This incremental walk through the bodies and joints can easily be achieved with the DCG data structure. In a next step the pseudoinverse is calculated (step b), which is multiplied with the end-effector velocity (step c). This relates the changes in the end-effector positions with changes in joint angles. The algorithm then simulates one time step, by adding the change to the current joint angle position (step d). The final step involves the application of forward kinematics to the arm, and the setting of the joint angles to their new position. The loop then begins its next iteration (step 5).

A big advantage of this inverse kinematics approach is that it is applicable to any robot arm, independent of its shape and number of joints. The incremental implementation reliably calculates an inverse kinematics solution of the DLW III and it can be used to translate task-space trajectories into joint-angle trajectories. In order to check the goodness of the inverse kinematics algorithm, we test the resulting position error in task space after having calculated the inverse kinematics:

We generated five test trajectories, all with the same shape ("eight-shaped", 2 meters long, located vertically parallel to the yz-plane) with different sampling rates. We

[11] Usually the last body.

then calculated the inverse kinematics solution of the DLW III separately for each trajectory. The resulting end-effector positions in task-space are compared to the original coordinates of the desired trajectory. The results showed that, even with the lowest sampling rate of 100 points, only a negligibly small mean absolute position error resulted. As expected, the error further decreases with higher sampling rates (i.e., smaller steps). This shows that we have a reliable inverse kinematics algorithm for a redundant robot arm.

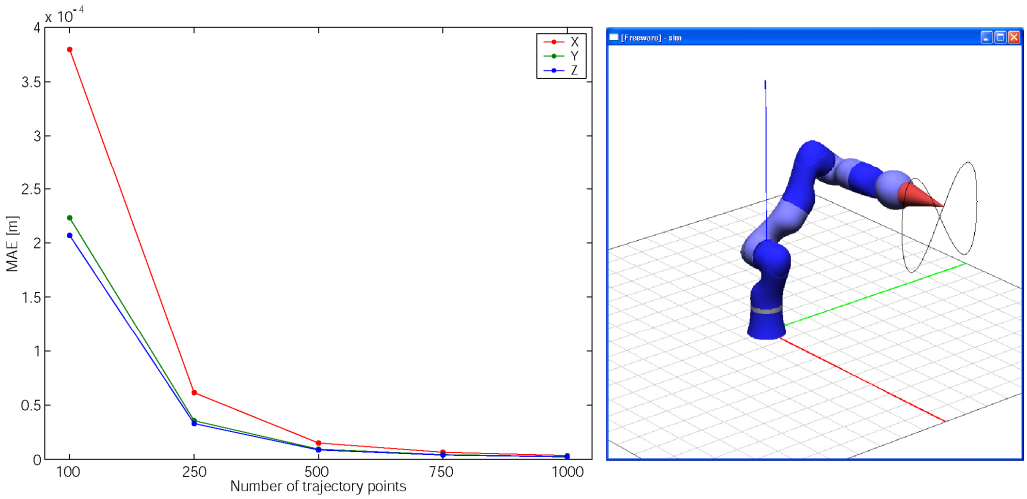


Figure 5.5: Left: Mean absolute errors (MAE) of end-effector positions in task space; Right: DLW III following a vertically aligned "eight-shaped" trajectory.

However, this algorithm has its limitations: Its iterative nature disallows a direct calculation of the inverse kinematics of any arbitrary point in task space. It depends on an initial state and the incremental transition to the new state. If the steps are too large, the algorithms will fails because it makes locally linear assumptions. These issues have a direct adverse effect on closed trajectories. The normal approach would be to calculate the joint angles for a trajectory once, and then work with joint angle coordinates. However, it can not be guaranteed that the joint angles at the starting position will be the same when the arm returns to the initial start position and starts the next trajectory loop. This effect shows up only in redundant arms, and is typically characterised by tiny "jumps" when the trajectory passes its start position. One can solve this problem by using an inverse kinematics that restricts null-space movements. Another method would be to calculate the inverse kinematics while following the trajectory. This approach will typically lead to different joint angle positions when the trajectory is tracked multiple times. This is disadvantageous if the objective is to learn the inverse dynamics, since it blows up the potential input space by delivering many kinematic solutions for a single point in space.

5.5.2 Servoing

The servoing task is a specific case of trajectory generation. The user can place a target object using the keyboard, and command the arm to reach towards the target. After this trajectory has been generated, the control paradigm does not change.

Section 2.2.1 illustrates the optimality of smooth, bell-shaped velocity profiles. Using a minimum jerk approach to trajectory planning is inapplicable. This is because our simulator does not yet have the capability to directly calculate the inverse kinematics of any arbitrary point in task space. Therefore, the constraints at the end of the motion (i.e. joint angles), which would be required for planning a minimum jerk trajectory, are not known a priori.

In order to still get a bell-shaped smooth velocity profile along the trajectory, we sample the points between the start and end points (in task space) following a sigmoid pattern. This is because the integral of a bell-shaped function can be approximated roughly by a sigmoid function (Hoffman, 2006).

$$S(t) = \frac{1}{1 + e^{-t}}$$

The speed of the reaching task can be modified, depending on the number of sampled points.

With this strategy, we should be able to produce smooth trajectories in task space and joint angle space. Even if this approach does not always lead to bell-shaped velocity profiles in all joints, it ensures that the joint velocity profiles are smooth.

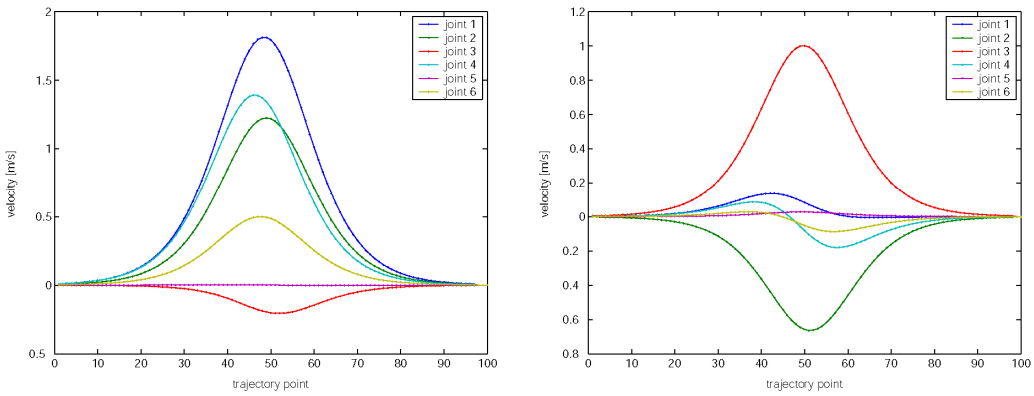


Figure 5.6: Recorded smooth velocity profiles for two separate reaching tasks. Depending on the movement, all joints have bell-shaped velocity profiles (left). In other cases, directional changes in joint-angle positions occur, resulting in non-bell-shaped profiles.

It is in general problematic to create an appropriate input mechanism to move a

target in 3D task space. The mouse is a 2D navigation tool, and is therefore not suitable for 3D tasks. This problem was solved by creating an interface to a motion tracking system in our simulator. We use the *Flock of Birds*[12] system, a magnetic field based motion tracker. It does not require cameras or sensor calibration, and is operated via an RS-232 interface. The user can produce motion data very quickly and comfortably by moving a small transmitter attached to a glove to its desired position. The visual servoing target can then be directly set by human arm motions. The motion tracker is a comfortable and efficient way to record 3D data that is biologically plausible, which can be very important depending on the experiment.

5.6 Motor Control and Learning Module

Thus far, the discussion has focused on important aspects of how a robot arm can be represented in the simulator, and how kinematics can be applied to it to follow trajectories and reach targets kinematically.

In order to be able to control the robot arm along trajectories as discussed in section 2.2.3 we implemented a class that provides a composite controller with open loop, feedback and composite modes. The class takes the defined gains from a parameter file to calculate the feedback commands based on desired joint positions and actual joint positions. The class contains a LWPR object which is responsible for learning and prediction of the feed-forward commands. LWPR is available as a C++ class, or in Matlab code. The LWPR class has a training mode for online and batch learning, as well as a prediction function.

All these components allow the controller to perform the required torque commands. The user can switch between open loop, feedback and feed-forward during operation. Additionally, he also has the option to switch on LWPR learning, and learn the dynamics of the arm for the actual task online. The control loop in Appendix A shows an example of how the control of the robot arm works in composite mode.

Experiments in motor learning are characterised by large amounts of produced data. The user therefore has the option to record and save all motion control data for further analysis in Matlab. Additionally, the LWPR model can be saved and loaded in an LWPR model file for later reuse.

---

[12] More details on www.ascension-tech.com.

5.7 User Interaction

The simulator is designed to be used by people with a strong background in robot control and machine learning, and some basic knowledge of programming in C. The simulator navigation takes place on two levels. One is the environment in which the robot arm is being simulated and rendered in 3D. The other is the control interface that is used to control the robot, such as loading trajectories, starting PD-control or storing recorded data. These two components were developed independently in order to allow changes and extensions at any point.

5.7.1 3D Navigation

In every 3D application, it is crucial that the user is able to navigate intuitively in order to view objects from different perspectives. Many 3D programs use fixed rotation axes (x, y, z) to transform user mouse motions into 3D camera rotations. This approach leads to confusing mirrored navigation once an axis has been rotated more than 180°. We therefore require a rotation mechanism that maps the user's mouse movements intuitively to rotations on the object. This was achieved by developing a trackball, a virtual hemisphere superimposed to the screen.

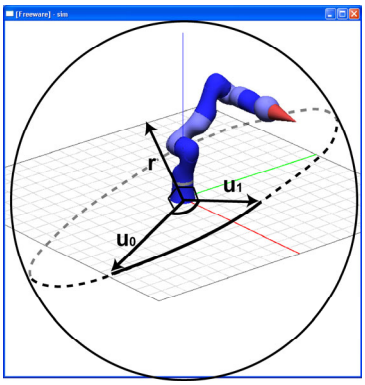


Figure 5.7: Visualisation of the trackball principle for a flexible and intuitive navigation.

When the user clicks on the screen at position (x, y) with the mouse pointer, this point is orthographically projected onto the hemisphere, thereby building a mouse vector $\mathbf{u}_0$. Whenever the pressed mouse is moved, a new mouse vector $\mathbf{u}_1$ is calculated in the same manner, and the actual rotation axis r is calculated by performing the cross product:

$$\mathbf{r} = \mathbf{u}_0 \times \mathbf{u}_1$$

The resulting rotation axis adapts dynamically to the direction in which the mouse is dragged. Thus, it is possible to perform a precise and flexible manipulation of the 3D scene.

5.7.2 Control Interface

The actual simulator uses the *QT toolkit* (Trolltech, 2006) for visualising the OpenGL window and handling all mouse and keyboard events. QT is a platform-independent GUI[13] toolkit that contains all elements to build a graphical user interface. However it is not expected that the standard C++ programmer has experience in QT, and therefore would have to learn QT in order to extend the simulator.

This leads us to a text-based standard console interface, in which the user navigates through the program using the keyboard (i.e. numbers and letters). For this purpose, we use a generic text-menu class (Glassborow, 2004). The key element of the menu class is a menu entry, which consists of a character that acts as a *trigger*, an *entry name* as well as a *function pointer* to the function that needs to be called for this menu entry.

```
typedef void (*actionFunction)();

struct menuEntry
{
       menuEntry( char trigger, string text, actionFunction action )
     : _trigger( trigger ),
       _text( Text ),
       _Action( Action )


  char _trigger;
  string _text;
  (*actionFunction)(); _action;

};
```

The menu class itself consists of three main parts: it has a C++ vector container for the menu entries, a function for adding new menu points, and a "perform" function that handles the user input and performs the desired action (i.e., calls the specified function).

```
class menu
{
  public:

       menu();
       void add( const menuEntry& Entry );

       void perform();

       // other useful functions
              . . .
  private:
       std::vector< menuEntry > m_Entries;
       char  m_title[256];
       int   m_width;

};
```

[13] Graphical User Interface

With the menu class, it is possible to build menu hierarchies that can be navigated very quickly, and more importantly, it allows the less experienced programmer to extend the program menu with new entries within minutes. Adding a new submenu, for example a help menu, can be realised with a few lines of standard C++ code.

```
void help()
{
       menu helpMenu;

       helpMenu.clearScreen();
       helpMenu.setTitle ("Help Menu");
       helpMenu.setMenuWidth (MENU_WIDTH);

       helpMenu.add( menuEntry( '1', "General help", generalHelp ) );
       helpMenu.add( menuEntry( '2', "Specific help", specificHelp ) );
       helpMenu.add( menuEntry( 'b', "Return", NULL ) );

       helpMenu.perform();
}
```

5.8 Simulator Architecture

In this chapter thus far, we have discussed the important components for creating a flexible robot arm simulator. One possible abstraction divides the architecture into three levels: *file level*, arm *control level*, and *simulation level.* The architecture of the simulator can be represented as shown in Figure 2.1.

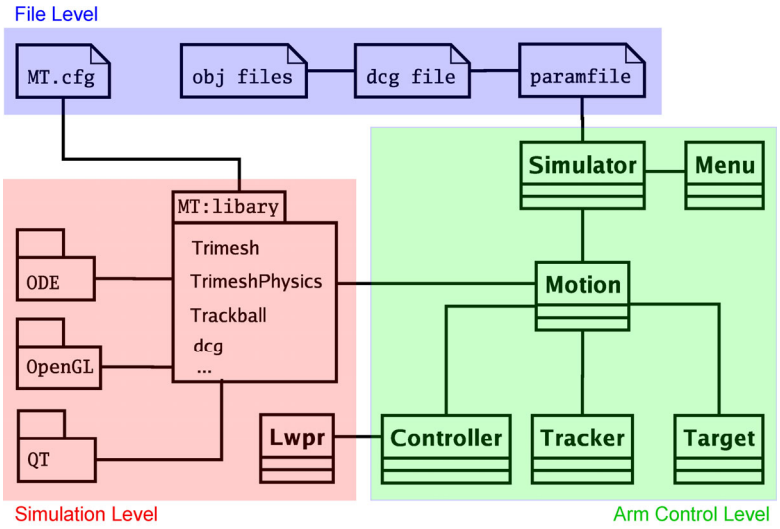


Figure 5.8: Basic architecture of the robot arm simulator (Note: This is not a UML specification).

In the file level, we have the `MT.cfg` file. This is a generic configuration file, in which any kind of parameters can be specified. These can be arrays, strings or single numbers. The problem with `MT.cfg` is that the name and location of the file cannot be changed, since it is hard coded in the MT library. This is not very convenient if, for example, the user wants to define the same parameter for different robot arms. Therefore we created a parameter file of our own (`paramfile`) that holds those simulator parameters that are likely to change from arm to arm or from test case to test case. The specification of the parameter file can be found in Appendix D.

The specification of the `paramfile` can be easily extended for new parameters at a later point. The `paramfile` also contains the path to a DCG file which specifies the physical robot arm setup. Depending upon whether trimesh objects are used or not, further references to obj files are set in the DCG file. This kind of convoluted file references are very comfortable for the practical use.

The simulation level mainly consists of the MT library. It delivers interfaces to ODE, OpenGL, and a basic window interface to QT. The simulation level also incorporates the extended components for trimesh visualisation, trimesh physical parameter calculation and

3D navigation, as well as extensions to the DCG format.

The arm control level delivers the functionality required to perform robot control and learning. The starting point of the program is the `Simulator` class. It is called with a parameter file as an argument. From there on, all specified parameters and required objects are initialised. Furthermore, in the initialisation process, a DCG graph is set up with the ODE objects and the required QT and OpenGL objects. The `Simulator` class contains a `Menu` object, which handles menu navigation and governs the interpretation of user inputs. The key object is the `motion` class, which is directly connected to the MT library that performs the simulation. `Motion` is responsible for trajectory planning, coordinate transformations and application of motor commands. It further contains other functions for handling the recorded motion data. In order to be able to control a robot arm, `motion` refers to a `Controller` object. The controller object itself uses an `Lwpr` object and delivers the different motor control modes to `Motion`, i.e., open loop, feedback and composite control. The `tracker` class implements the interface to the flock of birds' motion tracker. The target that has to be tracked is specified as a `Target` object, containing all position and visualisation parameters for the target. This concept enables the user to specify multiple targets in one scene.

6 Experimental Setup and Results

In this chapter, we describe how the implemented simulation framework was utilised to perform several learning tasks on the DLW III. We start with an explanation of how learning experiments with LWPR may be setup in practice. We then conclude with a discussion of the results of different motion tasks that were used to evaluate the performance of LWPR and the simulation framework.

6.1 LWPR in Practice

In practice, learning the inverse dynamics consists of three phases: defining the motion task, tuning the PD controller, and choosing the right LWPR parameters.

A significant consideration when learning the dynamics is the type of motion that has to be learned. Repeated motion across a given trajectory will learn well only in this region, and will generalise badly in other regions. In order to get a rich dataset, the velocities and accelerations must also be modulated. It is clear that the possible input space for all possible motions is enormous, taking into account all positions, velocities and accelerations. So, in general, one would start off with learning a single trajectory before tackling more complex learning tasks.

The general approach to learn a specific motion task is to make the robot arm follow some desired trajectory in feedback mode repeatedly and collect training data samples during operation from ODE. So we require a well-tuned PD controller that follows the trajectory accurately, and produces training data "near" the desired trajectory. Since the simulator does not simulate friction in joints, no steady state error is apparent, and the integral gains can be neglected. For the remaining PD controller, all gains are initially set to zero. Then the proportional gain which makes the system more responsive is increased until the system begins to overshoot and oscillate. Next, the derivative gain is increased until the system reduces the overshoot and behaves in the desired manner. The PD gains have to be set individually for each joint. Experiments with one general PD joint-setting do not lead to sufficiently accurate trajectory tracking. In the experiments we carried out, it was observed that the system was either too unresponsive, or the upper joints of the arm fell apart. This is plausible since joints near the base carry the other bodies' weight, and therefore require higher correction forces (gains) than joints located at the tip of the arm. The performance of the PD controller also depends on the actual joint constellation or arm -orientation, and one set of PD gains may not perform equally well in every region of the task space. In addition to the high gain controller, an additional controller with about 80% lower gains

was evaluated.

After having defined appropriate gains, the actual learning with LWPR begins. The LWPR parameters can be defined in the generic `MT.cfg` parameter file. The `Lwpr` class has a large number of different parameters that control the learning process. In order to identify the right LWPR parameters, one can start off by collecting some training data for batch learning. Test data should be separately collected and not be used for training. The LWPR parameters can vary very much between different motion tasks, and it is therefore hard to make predictions beforehand. The table in Appendix E provides an overview of the LWPR parameters and their meaning.

In practice, default values could be used for most LWPR parameters, whereas a few critical LWPR parameters have to be tuned in order to optimise the learning process:

The distance metric adaptation (`updateD`) can initially be switched off, because it is computationally very expensive. Not using distance metric adaptation mainly means that we are using spherical kernels with fixed size. Therefore it is important that the input data is allocated in similar value ranges. This is not the case for most motion data, because the velocity and acceleration input dimensions have much higher values. It is therefore recommended to normalise the input dimensions by its variance, which is less sensitive to outliers as compared to when normalisation is done with maximal values. The initial value for the distance metric (`InitD`) is the most important parameter in practice. In cases where distance metric adaptation is switched on, too large or too narrow kernels can drastically reduce convergence behaviour and it can take "very long" for the kernels to adapt their widths.

Without distance metric adaptation, the kernel width defines the smoothness of the learned model. Extremely large kernels will lead to smooth overall prediction patterns, since large areas are assumed to be linear. It will however fail to approximate well on highly non-linear regions. Tiny kernels, on the other hand, correspond to "memorising" every training point explicitly, which is also not desired. Another problem with tiny kernels is that it is likely to overfit if the data is noisy. In addition to that, smaller kernels will approximate large linear regions by using many small local models, instead of fitting very few large kernels.

Both the aforementioned situations are sub-optimal. In order to save resources, one could start off with larger kernels, allocating only few receptive fields. Then the kernels can be narrowed step by step, while monitoring the normalised mean squared error of the predictions on a test data set.

6.2 Experiments and Results

We have performed a number of tests in order to evaluate the goodness of the LWPR for learning the inverse dynamics of the DLW III.

In order to measure the goodness of a learned inverse model, several criteria can be used:

nMSE: The normalised means squared error (nMSE) is defined as $\frac{1}{n\sigma_y^2}\sum_{i=1}^{n}(y_i-\tilde{y}_i)^2$.

It takes into account the distribution of the output, which, for example, allows a representative comparison between data of different joints. Calculating the nMSE is in our case usually applied to predicted torque $\tilde{y}$ and the expected torque y.

Feed-forward to feedback signal ratio: In a composite controller, a well learned inverse dynamics model is expected to produce an accurate feed-forward command, and thus reduce the corrections required for the feedback component. This results in a high feed-forward to feedback ratio, which is a good indicator of the learning success.

Tracking error (nMSE) in *joint angles* or *end-effector positions*: This criterion measures the effective success of the motion in the simulator. However the tracking error is not ideal to compare different motion tasks. A high feed-forward to feedback ratio generally leads to a good tracking behaviour.

For the experiments, we did not incorporate the orientation of the end-effector. We merely used the final position of the end-effector-tip to follow a trajectory. Therefore our full model has six degrees of freedom. We found it useful to reduce the degrees of freedom at the beginning, in order to speed up learning and to be able to analyse the learning results and its resulting movements with greater ease.

6.2.1 Batch Learning with LWPR

We first started with a batch learning on the DLW III, reduced to 2 DOF in order to get fewer input dimensions. This reduces the amount of training data required, and expedites the initial learning process. We generated training data for a vertically aligned "eight" with 750 sampling points. We collected 40,000 data points, by using the simulator's PD-controller-mode. This dataset was split up into training data (75%) and test data (25%), which was not incorporated in the training phase. As discussed in the previous section, we started without distance metric adaptation and larger kernels, and then successively narrowed the width, while monitoring the nMSE.

Figure 6.1 shows how the nMSE is initially large, and how it decreases as the kernel

widths are narrowed. The test error decreases until the kernels become too small and fail to generalise, causing an increase in the nMSE. The general performance is worse for adaptive distance metrics, and more receptive fields are allocated (Figure 6.2). We therefore decided to use non-adaptive kernels for all further experiments, since it appeared to be more precise, and because LWPR performs much faster without distance metric adaptation.

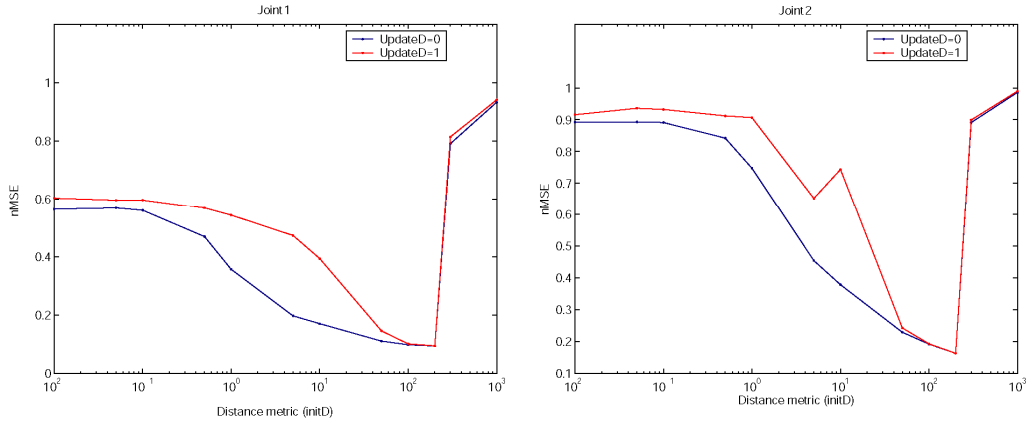


Figure 6.1: nMSE on a test set for joints 1 (left) and 2 (right), with and without distance metric adaptation.

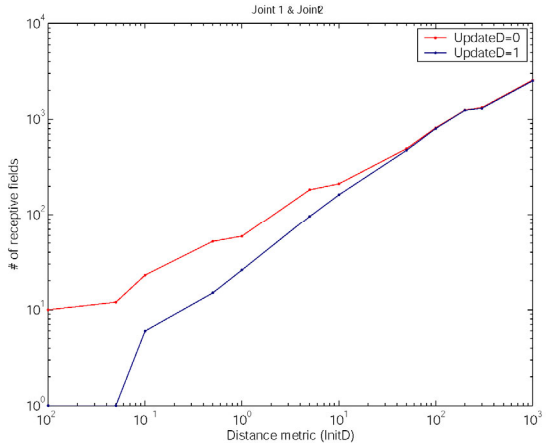


Figure 6.2: Number of allocated receptive fields, with and without distance metric adaptation.

The normalised mean squared error cannot be used as the only criterion to evaluate the learning success of LWPR. During the course of these experiments, we were mainly interested in the applied motor commands and its effects on the arm. We therefore ran a second experiment on the same trajectory with six degrees of freedom in order to analyse the motor commands and the tracking behaviour of a redundant arm.

For the redundant arm, we collected more data (200,000 data points), and proceeded as before. We then operated the arm in composite mode and recorded the applied joint torques for one loop through the trajectory. Figure 6.3 shows the applied

feed-forward signal (green) and the feedback signal (blue) of each joint during trajectory tracking in composite mode. Joints 2, 3 and 4 carry most of the robot arm's weight, and therefore produce the largest torques. Forces at joints 1, 5 and 6 are mainly rotational joints that are used very little for this motion task and consist mainly of noise, which is not learned by the inverse dynamics model. Not fitting to noise is desirable and the smooth feed forward signals indicate the use of correct kernel widths. The feed forward signal dominates over the whole range of the motion for the major joints (2, 3, 4). It shows that the inverse dynamics model has been learnt (Table 6.1).

As a last step, the learning result was checked by comparing the tracking performance on a low gain controller (all gains reduced by 80%) between feedback control and composite control. Figure 6.4 shows clearly that composite tracking outperforms low-gain feedback control. The tracking deviations in the top-left region occur because of the aforementioned inverse kinematics "jump" with closed trajectories. After some time, the controller stabilises for this major disturbance. Table 6.2 shows the trajectory tracking performance over the whole trajectory including the first unstable starting part. Once the arm has stabilised towards the desired trajectory, the composite controller performs clearly better than the low gain feedback controller (Table 6.3). The largest difference is visible in the z-axis which is directly affected by gravitational forces.

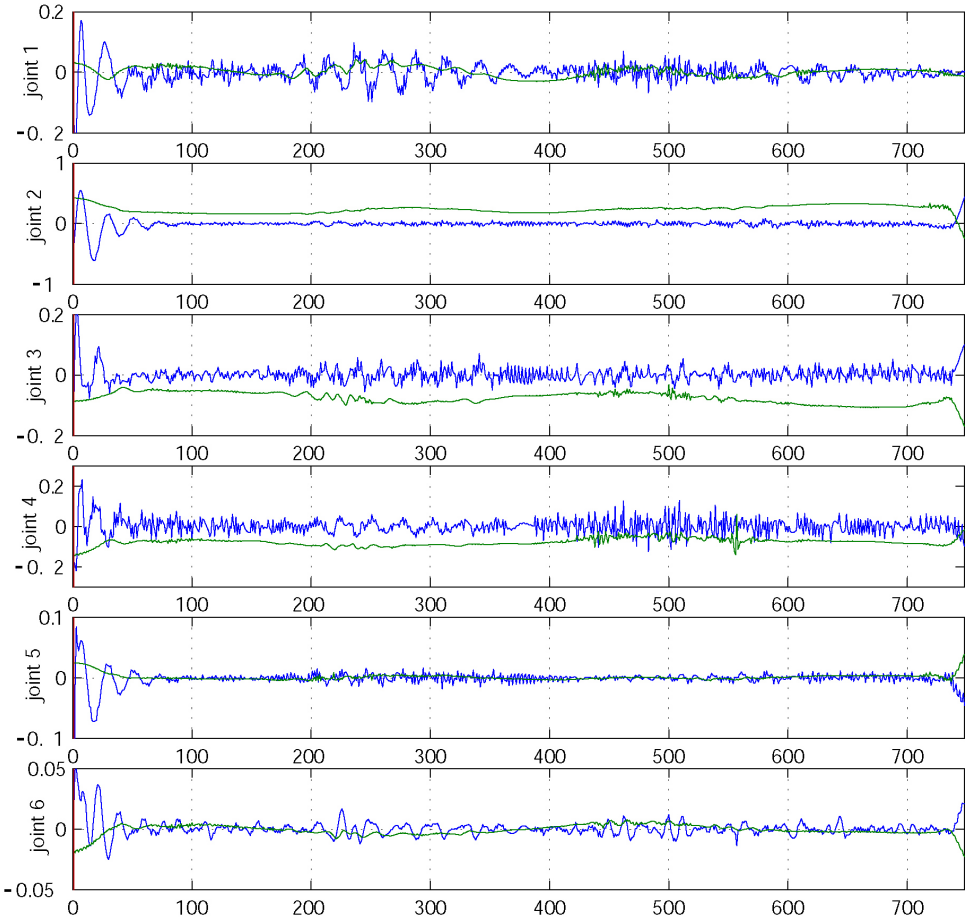


Figure 6.3: Feed-forward (green) and feedback component (blue). X-axis represents the point of the trajectory and the y-axis the applied joint torque. Note: The y-axes are individually scaled for each joint.

	figure eight motion task			
	avg(ff/fb)	**var(ff/fb)**	**avg(ff)**	**avg(fb)**
Joint 1	0.433344	0.063147	0.013400	0.023098
Joint 2	0.890025	0.014146	0.236670	0.038371
Joint 3	0.834231	0.013368	0.076202	0.017104
Joint 4	0.749425	0.026877	0.078172	0.030001
Joint 5	0.401416	0.057779	0.002988	0.007001
Joint 6	0.495602	0.056708	0.003132	0.004005

Table 6.1: Average feed-forward to feedback ratio avg(ff/fb); Variance of ff/fb; average of feedback signal avg(fb); Average feed-forward signal avg(fb).

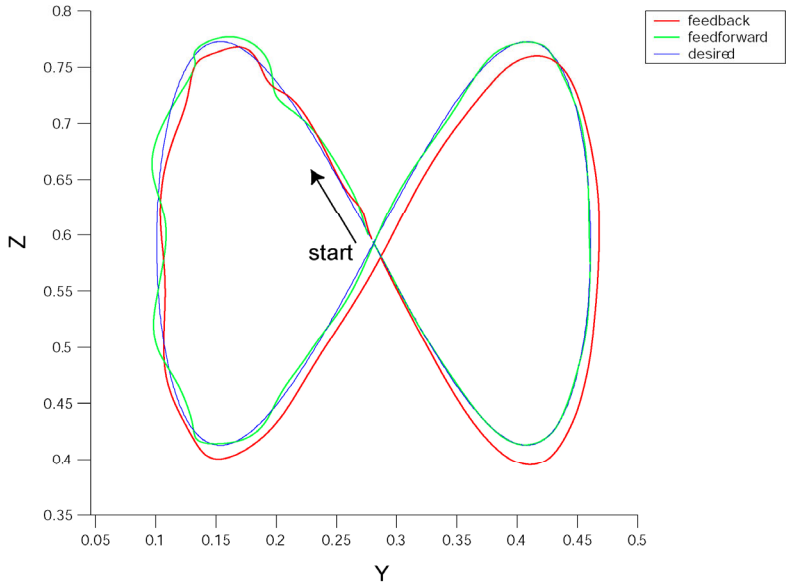


Figure 6.4: Tracking performance of a low-gain feedback controller (red) and a composite controller (green). "Start" indicates the beginning of the trajectory.

	Feedback control nMSE	**Composite control nMSE**
Y-axis [0-750]	1.95E-03	8.29E-04
Z-axis [0-750]	9.38E-03	2.63E-04

Table 6.2: Tracking performance over whole trajectory (750 trajectory points).

	Feedback control nMSE	**Composite control nMSE**
Y-axis [325-750]	8.96E-03	4.78E-04
Z-axis [325-750]	1.73E-02	3.25E-05

Table 6.3: Tracking performance for stable second part of trajectory (trajectory points 325 to 750).

In order to ensure the learning success of LWPR, without the aforementioned kinematic problems, we repeated the presented learning procedure with a trajectory that was generated by a sinusoidal joint angles sequence. According to the assumption stated earlier (in Chapter 5.5), we should be able to avoid any kinematic inaccuracies like those that occurred during the performance of the figure eight task. Figure 6.5 proves that these expectations hold good. The feedback signal over the whole trajectory is significantly smaller than the predicted feed-forward signal, which means that the motion is mainly performed by the learned inverse dynamics model. The feedback channel contains almost no noise, and the predictions are smooth. This shows that LWPR in batch learning predict very well on all six joints.

The mentioned noise in the task space planned trajectory occurs, because of the differentiation of the position to calculate the velocities and accelerations. Tiny positional changes (zig-zag patterns) will lead to large alternating velocities and huge alternating accelerations. This then results in noise in the feedback signal.

The signals from the sinusoidal task on the other hand contain almost no noise, which indicates overall smooth positional transitions. The learned model predicts very efficiently, which can be seen by the large feed-forward to feedback ratios (Table 6.4).

We conclude from these experiments that joint angle tasks, can be learned much more efficiently, due to the absence of noise and ideal kinematic arm constellations. We furthermore have seen that the feed-forward to feedback ratio is a reliable indicator for measuring the learning success, independent from the motion task itself. We will therefore use this criterion further to evaluate the quality of learning for more complicated motion tasks.

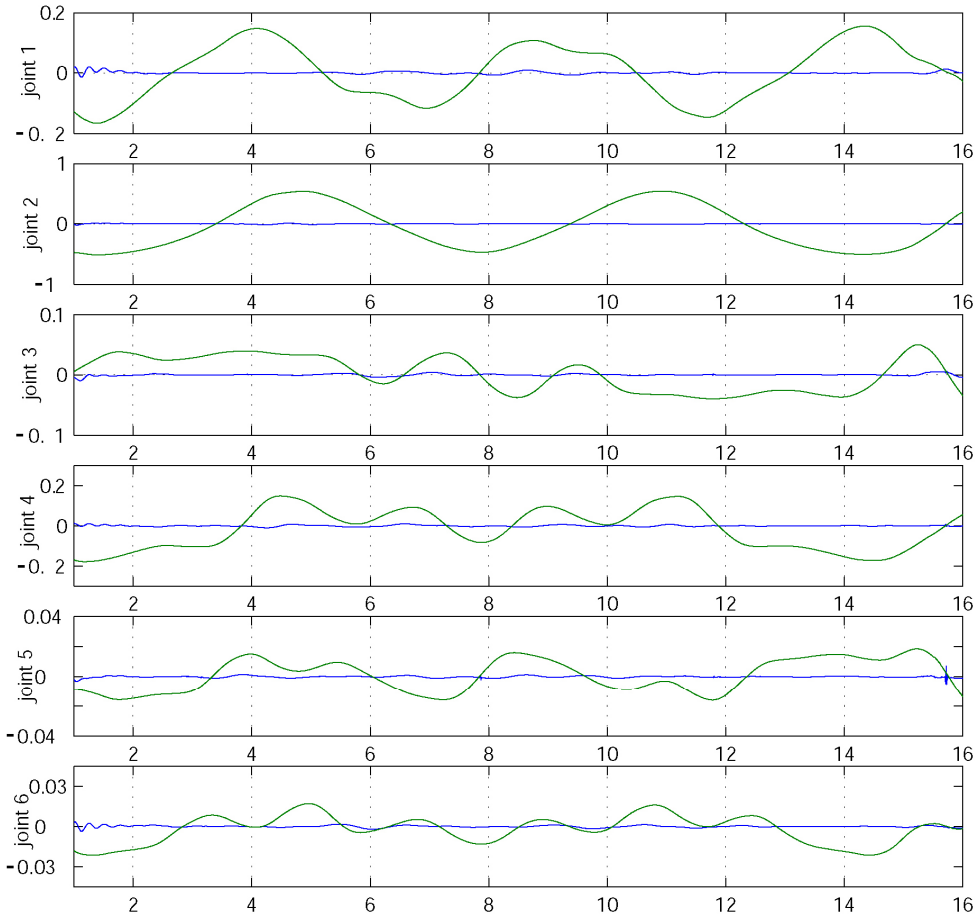


Figure 6.5: Feed-forward (green) and feedback component (blue). X-axis represents the point of the trajectory and the y-axis the applied joint torque. Note: The y-axes are individually scaled for each joint.

	Sinusoidal motion task			
	avg(ff/fb)	**var(ff/fb)**	**avg(ff)**	**avg(fb)**
Joint 1	0.912390	0.024673	0.086117	0.012260
Joint 2	0.955564	0.011608	0.329532	0.013430
Joint 3	0.855273	0.038965	0.029555	0.006173
Joint 4	0.894068	0.029307	0.091035	0.008033
Joint 5	0.893292	0.024647	0.010898	0.001425
Joint 6	0.803066	0.049541	0.006020	0.001557

Table 6.4: Average feed-forward to feedback ratio avg(ff/fb); Variance of ff/fb ; average of feedback signal avg(fb); Average feed-forward signal avg(fb).

6.2.2 Learning a Task Region Online

A visual servoing task is characterised by arbitrary trajectories in task space. We therefore aim to learn the dynamics for a specific operating area of the arm, in which we expect it to be operated. To learn the whole range of a seven degree of freedom arm is a utopian aim with the computational resources that are available to us. We therefore defined a specific operating region of the arm in which we want to perform visual servoing. The region covered mainly one quadrant of the arms reach, as shown below:

$$20 \text{ cm} < x < 70 \text{ cm}$$

$$20 \text{ cm} < y < 70 \text{ cm}$$

$$0 \text{ cm} < z < 50 \text{ cm}$$

We also reduced the number of the arm's degrees of freedom to three, in order to make sure that each position in task space has a unique joint angle combination. This resulted in a dramatic reduction in the size of the input space. We then simulated 1000 reaching tasks, while simultaneously learning LWPR online, in an automated mode of our simulator. The simulator plans trajectories with sigmoidal position pattern, which produces straight lines in task space with smooth velocity profiles. Whenever the target is reached, the simulator plans a trajectory towards a new randomly selected target. The trajectory has fixed length (100 sampling points), which means that a wide range of velocities and accelerations will be incorporated into the training depending on the start and end position of the trajectory.

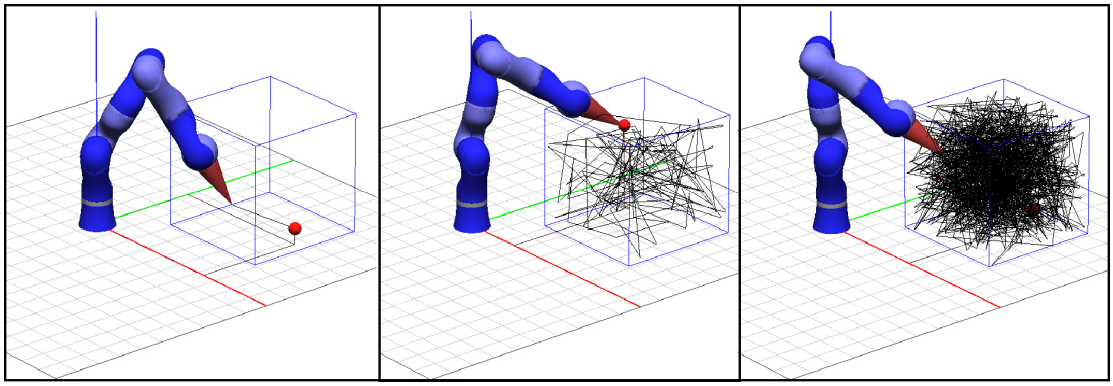


Figure 6.6: Visualisation of learning experiment. Left: Visual servoing towards a red target in the first iteration; middle: After 10 iterations; right: After 1000 iterations. The blue box indicates the target area.

Figure 6.7 shows the number of receptive fields allocated after reaching a target. Initially the number of newly allocated receptive fields increases quickly. After passing 200 trials (20,000 training points), the growth rate of the number of new receptive is reduced,

and receptive fields are added only when previously unseen dynamic situations arise. E.g.,a very fast trajectory with large accelerations. After 1000 trials (i.e. 100,000) training points we stop the learning.

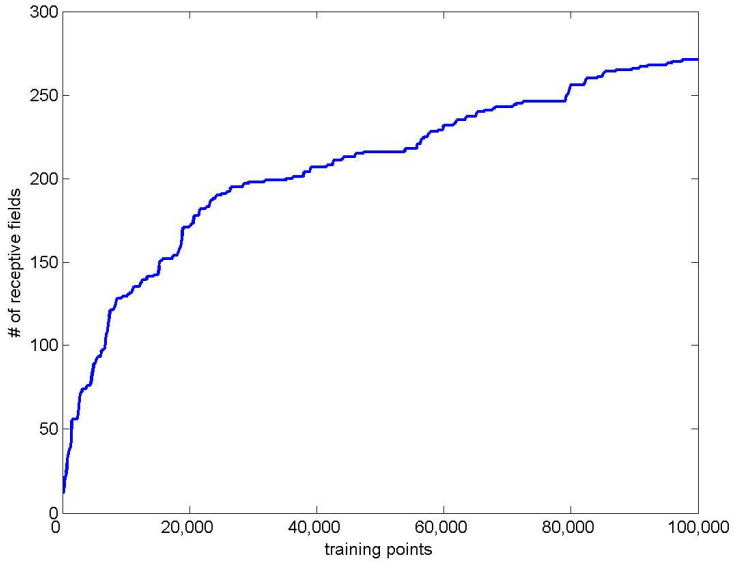


Figure 6.7: Number of receptive fields allocated during learning 1000 random reaching tasks.

We created at random 10 reaching tasks (9 transitions) to constitute a test set which was not used for training. The series of results on the next three pages show how the feed-forward signal improves during online learning. With only a few receptive fields, the prediction signals are weak. As the proportion of area covered in the task region of the arm increases, the feed-forward signal starts to dominate over the feedback signal, which indicates that the region has been learned successfully. This experiment clearly demonstrates the ability of LWPR to learn online and its robustness against negative interference, which reveals the ability to learn the dynamics of an entire operational area. However, to assure that "every" possible motion in the target area is trained more than 100,000 training points would be required.

- Figure 6.8 on page 57 shows the motor commands (torques) for the training set, after LWPR was trained with 1 reaching task. 12 receptive fields were allocated.
- Figure 6.9 on page 58 shows the motor commands (torques) for the training set, after LWPR was trained with 200 reaching task. 173 receptive fields were allocated.
- Figure 6.10 on page 59 shows the motor commands (torques) for the training set, after LWPR was trained with 1000 reaching task. 271 receptive fields were allocated.

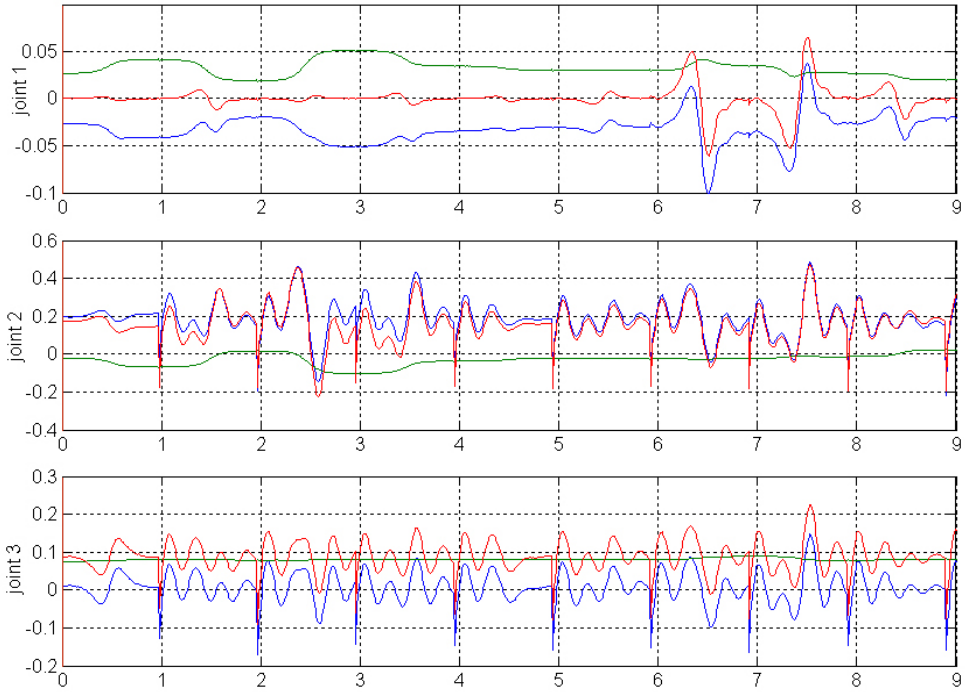


Figure 6.8: Motor command signals on 9 test reaching tasks. Blue: Feedback signal; Green: Feed-forward signal; Red: Composite signal applied to robot arm.

	Test set - model with 12 receptive fields			
	avg(ff/fb)	**var(ff/fb)**	**avg(ff)**	**avg(fb)**
Joint 1	0.503092	0.008983	0.031951	0.033166
Joint 2	0.160850	0.015050	0.033501	0.194204
Joint 3	0.753084	0.022452	0.080883	0.031719

Table 6.5: Numerical results from Figure 6.8.

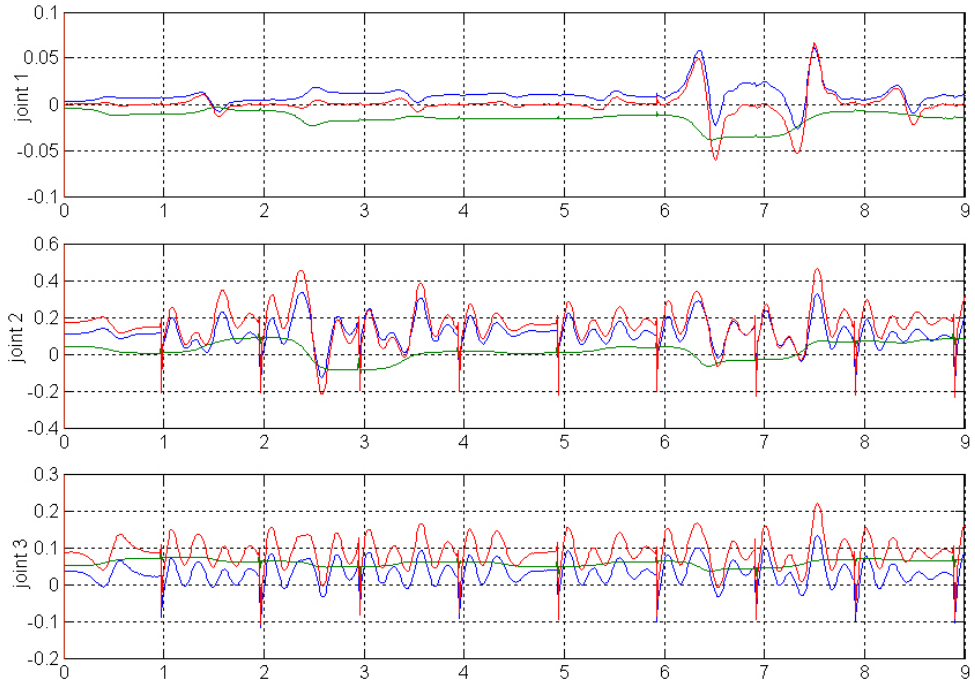


Figure 6.9: Motor command signals on 9 test reaching tasks. Blue: Feedback signal; Green: Feed-forward signal; Red: Composite signal applied to robot arm.

	Test set - model with 173 receptive fields			
	avg(ff/fb)	**var(ff/fb)**	**avg(ff)**	**avg(fb)**
Joint 1	0.578872	0.016613	0.014740	0.011543
Joint 2	0.291470	0.041543	0.044600	0.119798
Joint 3	0.656594	0.030334	0.058697	0.036514

Table 6.6: Numerical results from Figure 6.9.

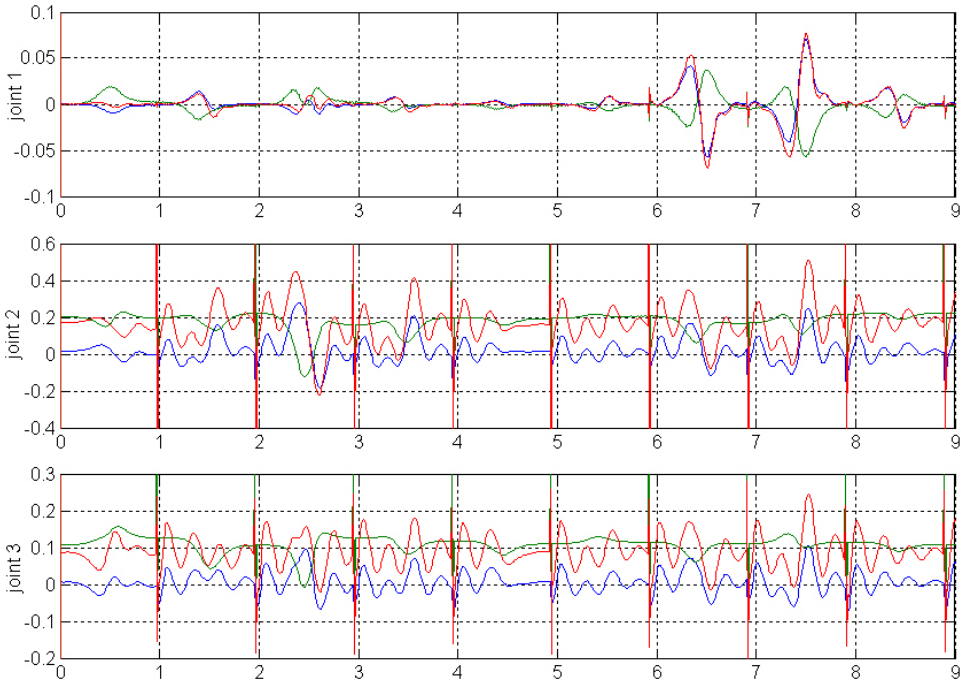


Figure 6.10: : Motor command signals on 9 test reaching tasks. Blue: Feedback signal; Green: Feed-forward signal; Red: Composite signal applied to robot arm.

	Test set - model with 271 receptive fields			
	avg(ff/fb)	**var(ff/fb)**	**avg(ff)**	**avg(fb)**
Joint 1	0.579187	0.043555	0.006484	0.006685
Joint 2	0.800530	0.030224	0.192755	0.051649
Joint 3	0.829835	0.022043	0.110323	0.023353

Table 6.7: Numerical results from Figure 6.10.

6.2.3 Verifying the Task Region with Human Motion

So far we have only verified our learned model using artificially created trajectories with the same motion pattern that was used for training. So the question arises as to whether the arm has learned the dynamics produced by movements other than the sigmoidal motion patterns. The expectation is that the learned inverse dynamics model will remain valid for any arbitrary motions within the learned target area, and the learned velocities and accelerations. This does not cover some unrealistic extreme cases, with huge acceleration for example[14].

In order to verify this assumption, we used the flock of birds to create test trajectories generated by humans. The first test trajectory was designed to simulate the task of reaching several positions on the floor, and jumping between these positions. This mimics some picking or displacing tasks which often occur in the everyday life of humans and robots. The second recorded trajectory is a 3D spherical arm movement, which is located about 30cm above the surface of the environment, so as to incorporate higher regions of the task region equally.

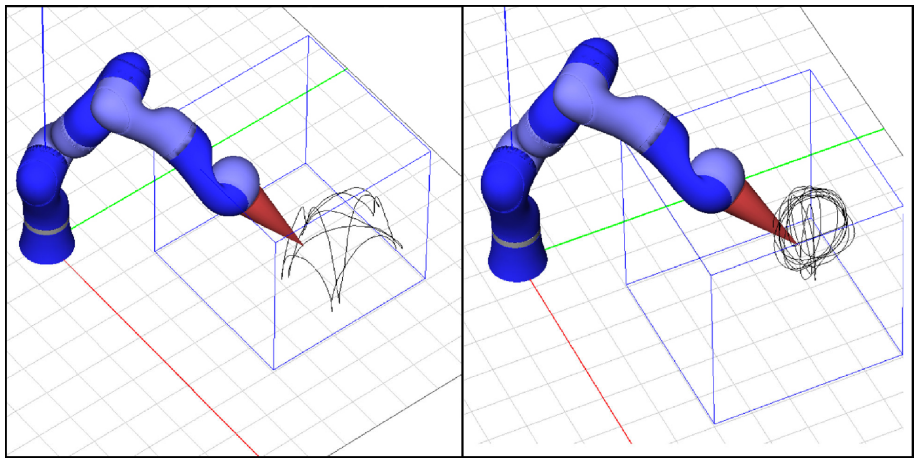


Figure 6.11: Two recorded human trajectories. Left: Picking task; Right: Smooth circular motion.

The flock of birds produces data points with a frequency of 100 Hz. The tracking data is scattered with noise. The tracker produces local oscillations that end up in tiny zig-zag pattern in the joint angle coordinates. This noise leads to very large alternating acceleration values, which in most cases are not covered by the learned inverse dynamics

[14] In real robots such cases should not occur, since they have a maximum speed and acceleration, which is defined by the joint motor specifications.

model. This problem was solved by smoothing the data in a pre-processing step in Matlab. The results on the next two pages demonstrate behaviour similar to the results obtained by using artificially created test data. The learned inverse dynamics model of the task region therefore predicts the dynamics well for a visual servoing tasks in this area.

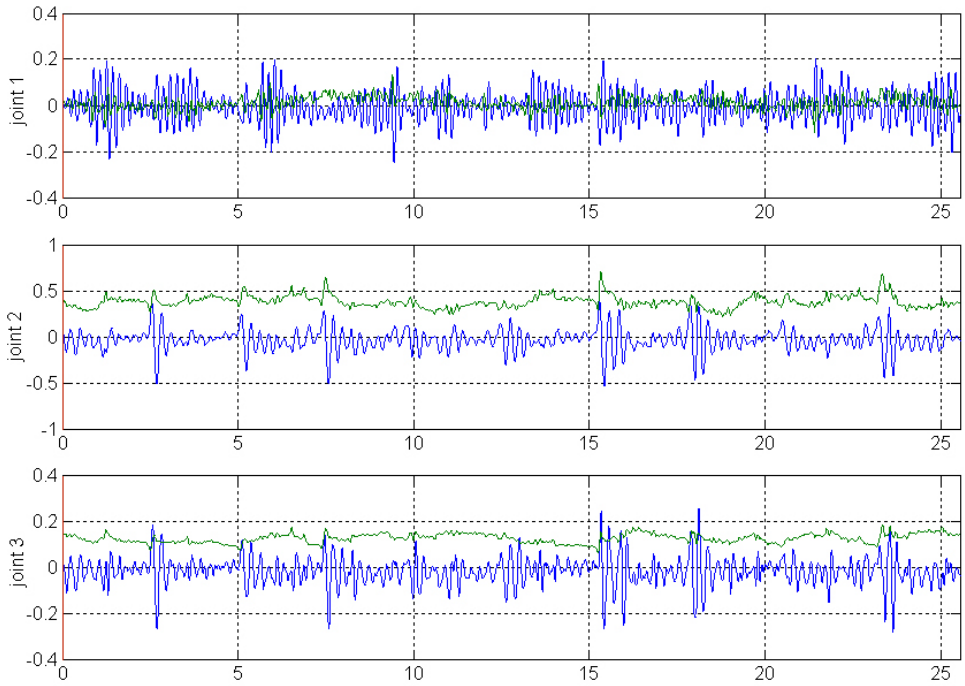


Figure 6.12: Motor command signals on picking task . Blue: Feedback signal; Green: Feed-forward signal.

	Picking Task - model with 271 receptive fields			
	avg(ff/fb)	**var(ff/fb)**	**avg(ff)**	**avg(fb)**
Joint 1	0.349488	0.062176	0.022983	0.053045
Joint 2	0.835001	0.014549	0.382137	0.086294
Joint 3	0.772227	0.022832	0.125111	0.045198

Numerical results from Figure 6.12.

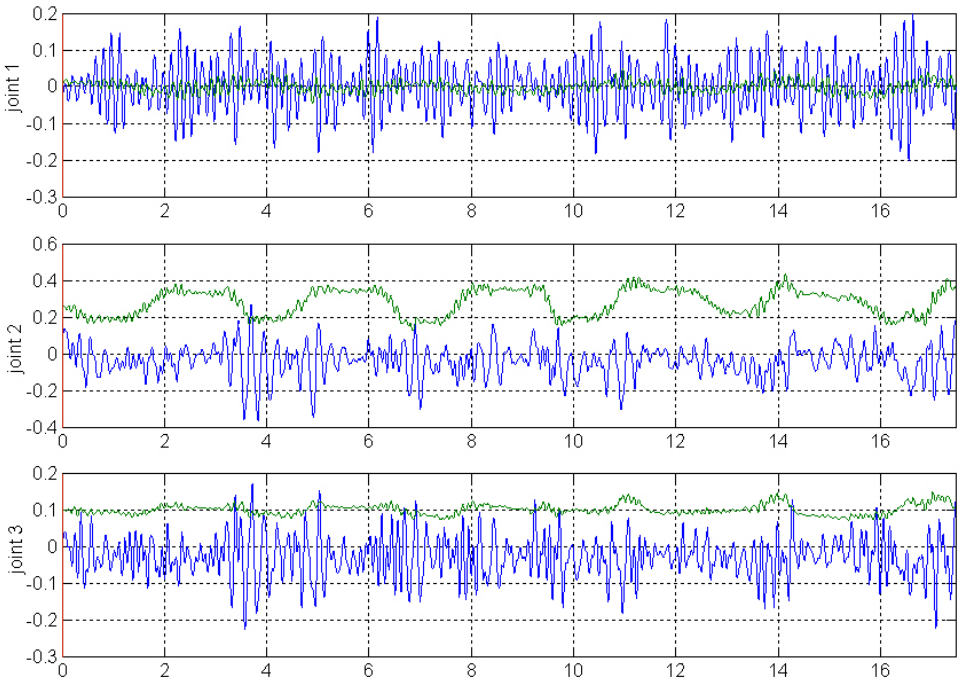


Figure 6.13: Motor command signals on sphere task. Blue: Feedback signal; Green: Feed-forward signal.

	Sphere Task - model with 271 receptive fields			
	avg(ff/fb)	**var(ff/fb)**	**avg(ff)**	**avg(fb)**
Joint 1	0.248233	0.047788	0.012071	0.054463
Joint 2	0.806663	0.020039	0.278921	0.071855
Joint 3	0.709195	0.026015	0.099873	0.049179

Numerical results from Figure 6.13.

6.3 Discussion

This chapter presented a series of experiments that showed LWPR's learning performance in the simulated environment. It was shown that the initialisation of the distance metrics is crucial for learning success, and that good learning results can be achieved without using distance metric adaptation. Trajectories generated in joint angle space have smoother position, velocity and acceleration profiles than task space based trajectories. The absence of noise in the data and optimal kinematic constellations allows LWPR to predict joint angle based trajectories much better than task space based trajectories. To deal with the noise in the generated data a processing of the data is required. This could, for instance, be a low pass filter. Alternatively one could use a more sophisticated approach for differentiating the joint angles (in order to get the joint velocities and accelerations). So far differentiation is done by calculating the difference between the actual joint angles and the previous joint angle values (α_{t-1}) and dividing them by the time step size:

$$\dot{\alpha}_t = \frac{\alpha_t - \alpha_{t-1}}{T}.$$

A calculation that incorporates multiple previous joint angle values would lead to smoother velocity and acceleration profiles and therefore reduce the noise problem.

The results further show the stability of LWPR in the face of negative interference. Independent of the transition direction and velocity, the predictions perform equally well in the learned task region of the robot arm. The DLW III successfully learned an operating region in which it can perform the visual servoing task. The learning results strongly differ between the joints. This is reasonable, because each task uses different joints, depending on the kinematic setup during the motion.

The computational and working memory resources that are required to run LWPR online with large degrees of freedom are significant. Especially in cases in which many local models are allocated, the working memory requirements are large. On the used system (a Personal Computer), real time online learning could be achieved only by reducing the number of degrees of freedom (input dimensions).

The parameter estimation in LWPR remains critical and requires a lot of "human intuition". Besides the LWPR parameters, the PD gains also play an important role. The joint concept in ODE does not allow operation with gains that are too large (see Section 4.4), and it can not be guaranteed that the PD gains are tuned optimally for every possible trajectory.

The simulator's inverse kinematics algorithm shows some negative properties. Apart

from the previously mentioned problem with closed trajectories, the iterative nature of the algorithm reveals instabilities when operating with the motion tracker. These occur because the tracker produces minimal positional oscillations, which are not yet filtered out by the simulator. In the actual simulator implementation, the inverse kinematics algorithm treats these minimal oscillations as new trajectories which have to be tracked. This leads to undesirable arm positions in the online tracking mode, in which the arm constellation is heavily affected by gravitational forces and downward drifts. These problems may be alleviated for by adding minimal position thresholds that prevent the generation of a new target position, if the changes are not above the threshold. A general smoothing of the tracker data could also mitigate this problem.

However, the iterative inverse kinematics algorithm has the major advantage of being universally applicable to any arbitrary robot arm, which was what we had aimed for when developing the simulator. It is therefore the right algorithm for a first evaluation of a new robot arm, but not necessarily the optimal solution for intensive and reliable use.

7 Conclusion

In this dissertation, we presented the planning, implementation and evaluation of a flexible robot arm physical simulator, and discussed how modern learning algorithms could be applied to create accurate and compliant movements.

We started off with a review of the background information that was required in order to realise the simulator. This covered classic robot control, learning inverse dynamics with LWPR, and the basics of physics engines. We further described the requirements and components of the simulation framework. The focus was set on creating a strong software base for intensive use and further development; a base which allows for the incorporation of novel robot arms of any shape and physical properties. We have also seen that this requires more sophisticated object geometry (trimeshes). To achieve this, we have discussed the issues with polyhedral geometry, 3D computer graphics, and practical modelling skills. We incorporated a modern tracking system so as to create an appropriate interface for 3D motion, and delved on the problems with the data it produces.

As a next step, we outlined the practical issues involved in applying LWPR to the robot arm in the simulator. We discussed the critical learning parameters, as well as restrictions on performance and learning. A number of representative tests were performed, which proved that the simulation environment is a feasible tool to perform motor learning experiments.

The implemented simulation environment showed very good usability characteristics during the experimental phase. We believe that we have produced a useful tool for performing experiments on simulated robots, which allows researchers to perform their learning experiments more efficaciously. The software design will allow the researchers to extend the existing simulator easily and make it an even more powerful tool in the future.

The results showed that LWPR is, as was expected, well suited for learning the dynamics of the DLW III in a simulated environment. We successfully learned the inverse dynamics model of an operating area of the DLW III, and verified the model with data recorded from human motion. We are positive that we will profit from the gained insights, when using the real DLW III in future.

Despite promising results, further research has to be carried out into simulated learning behaviour. This will also help to identify the simulator's boundaries and help to improve it. Further development of the inverse kinematics algorithm will reduce the problems with closed trajectories. Additionally, the simulator framework could be extended to allow for multiple context learning scenarios, and the DCG-file specification could be

extended to include support for haptic experiments. The latter will require an extension of the DCG by incorporating touch sensors in ODE and an interface for hardware touch-sensors.

From a simulator design point of view, it would be useful to make a clearer separation between OpenGL, ODE and QT. These components are not separated properly inside the MT library. Separation will lead to much better performance and maintainability. Also, the text based interface has limitations, as it is not linked to event handling in QT. Two solutions can be proposed for this problem.

One could either implement thread-based handling of the two separate units (GUI and menu), or port the menu completely to QT. As the functionality provided is expected to grow in the future, a Graphical User Interface would be a good direction forward.

For the LWPR implementation, a well-chosen aim would be to include multiprocessor support, such that each joint could be learnt on a separate processor. This would help distribute the computational load and allow more complex and longer lasting experiments.

Bibliography

Barraff, D., & Wittkin, A. (1997). Physically based modeling: Principles and practice. Paper presented at the ACM Siggraph '97.

Blinn, J. F. (1977). Models of light reflection for computer synthesized pictures. ACM SIGGRAPH Computer Graphics, 11(2), 192-198

Erleben, K. (2004). Stable, Robust, and Versatile Multibody Dynamics Animation. Unpublished Ph.D. Thesis, University of Copenhagen, Copenhagen.

Featherstone, R., & Orin, D. (2000). Robot Dynamics: Equations and Algorithms. Paper presented at the IEEE lntemational Conference on Robotics & Automation, San Francisco, CA.

Flash, T., & Hogan, N. (1985). The coordination of arm movements: an experimentally confirmed mathematical model. Journal of Neuroscience, 5, 1688-1703.

Glassborow, F. (2004). You Can Do It!: A Beginners Introduction to Computer Programming: John Wiley & Sons.

Hirzinger, G. (2006). Institute of Robotics and Mechatronics. Retrieved 28.07.2006, from http://www.dlr.de/

Hoffman, H. (2006). Approximating the integral of a bell-shaped function. Personal Communication.

Klein, C. A., & Huang, C.-H. (1983). Review of Pseudoinverse Control for Use with Kinematically Redundant Manipulators IEEE Transactions on Systems, Man, and Cybernetics, 13(3), 245-250

Korein, J. U., & Badler, N. I. (1982). Techniques for generating the goal-directed motion of articulated structures. IEEE Computer Graphics and Applications, 71-81.

Liegeois, A. (1977). Automatic supervisory control of the configuration and behavior of multibody mechanisms. IEEE Transactions on Systems, Man and Cybernetics, 7(12), 868-871.

McKerrow, P. J. (1991). Introduction to robotics. Boston, MA, USA: Addison-Wesley Longman Publishing Co.

Michel, O. (2004). Webots: Professional Mobile Robot Simulation. International Journal of Advanced Robotic Systems, 1(1), 39-42.

Mirtich, B. (1996). Fast and Accurate Computation of Polyhedral Mass Properties. Journal of Graphics Tools 1(2), 31-50.

Mirtich, B., & Canny, J. F. (1995). Impulse-based simulation of rigid bodies. Paper presented at the Symposium on Interactive 3D graphics, Monterey, California, United States.

Moore, A. W. (1990). Efficient Memory-based Learning for Robot Control. Unpublished PhD Thesis, Trinity Hall - University Of Cambridge.

Nakamura, Y., & Hanafusa, H. (1986). Inverse kinematics solutions with singularity robustness for robot manipulator control. Journal of Dynamic Systems, Measurement, and Control, 108, 163-171.

Nelson, W. L. (1983). Physical principles for economies of skilled movements. Biological Cybernetics, 46, 135-147.

Noakes, L., Heinzinger, G., & Paden, B. (1989). Cubic splines on curved surfaces. IMA Journal of Mathematical Control & Information, 6, 465–473.

Park, F. C., & Brockett, R. W. (1994). Kinematic Dexterity of robot mechanisms. International Journal of Robotic Research, 13(1), 1-15.

Phong, B. (1975). Illumination for computer generated pictures. Communications of the ACM, 18, 311-317.

Reddy, M. (2006). The graphics file format page. Retrieved 09.08.2006, from http://www.martinreddy.net/gfx/3d/OBJ.spec

Robins, N. (2000). Wavefront OBJ model file format reader/writer/manipulator.

Schaal, S., & Atkeson, C. G. (1998). Constructive incremental learning from only local information. Neural Computation, 10(8), 2047-2084.

Schaal, S., Atkeson, C. G., & Vijayakumar, S. (2002). Scalable techniques from nonparameteric statistics for real-time robot learning. Applied Intelligence, 17(1), 49-60.

Schaal, S., Vijayakumar, S., & Atkeson, C. G. (1998). Local Dimensionality Reduction. In M. I. Jordan, M. J. Kearns & S. A. Solla (Eds.), Advances in Neural Information Processing Systems (Vol. 10). Cambridge, MA: MIT Press.

Schreiner, D. (2004). OpenGL Reference Manual: The Official Reference Document to OpenGL (4 ed.): Addison Wesley Professional.

Sciavicco, L., & Siciliano, B. (2000). Modelling and Control of Robot Manipulators. New York: Springer.

Scott, D. W. (1992). Multivariate density estimation: Theory, Practice, and Visualization. New York: Wiley.

Shadmehr, R., & Wise, S. P. (2005). The Computational Neurobiology of Reaching and Pointing: A Foundation for Motor Learning. Cambridge, Massachusetts: MIT Press.

Shiller, Z. (1994). Time-energy optimal control of articulated systems with geometric path constraints. Paper presented at the Robotics and Automation, San Diego, CA, USA.

Siciliano, B. (1990). Kinematic control of redundant robot manipulators: A tutorial. Journal of Intelligent and Robotic Systems, 3(3), 201-212.

Smith, R. (2001, 03.08.2006). Q12.org - Personal web space of Russell Smith, from http://www.q12.org/

Smith, R. (2006). Open Dynamics Engine v0.5 - User Guide. Retrieved 02.08.2006, from http://www.ode.org/ode-latest-userguide.html

Toussaint, M. (2005). MT Code. Retrieved 07.08.2006, from http://homepages.inf.ed.ac.uk/mtoussai/source-code/doc/html/index.html

Trolltech. (2006). Qt Homepage. Retrieved 14.08.2006, from http://www.trolltech.com/

Uno, Y., Kawato, M., & Suzuki, R. (1989). Formation and control of optimal trajectories in human multijoint arm movements: minimum torque-change model. Biological Cybernetics, 61, 89-101.

Vijayakumar, S., D'Souza, A., & Schaal, S. (2005). Incremental Online Learning in High Dimensions. Neural Computation, 17(12), 2602-2634

Vijayakumar, S., D'souza, A., Shibata, T., Conradt, J., & Schaal, S. (2002). Statistical Learning for Humanoid Robots. Autonomous Robots, 12(1), 55-69.

Wampler, C. W. (1986). Manipulator inverse kinematic solutions based on vector formulations and damped least squares methods. IEEE Transactions on Systems, Man and Cybernetics, 16, 93-101.

Wang, L. C. T., & Chen, C. C. (1991). A combined optimization method for solving the inverse kinematics problem of mechanical manipulators. IEEE Transactions on Robotics and Automation, 7, 489-499.

Whitney, D. E. (1969). Resolved motion rate control of manipulators and human prostheses. IEEE Transactions on Man-Machine Systems, 10, 47-53.

Wolovich, W. A., & Elliot, H. (1984). A computational technique for inverse kinematics. Paper presented at the 23rd IEEE Conference on Decision and Control.

Wolpert, D. M., Ghahramani, Z., & Jordan, M. I. (1995). Are arm trajectories planned in kinematic or dynamic coordinates? An adaptation study. Experimental Brain Research, 103(3), 460-470.

Zhao, J., & Badler, N. I. (1994). Inverse kinematics positioning using nonlinear programming for highly articulated figures. ACM Transactions on Graphics, 13, 313-336.

Appendix A - Motor Control Loop

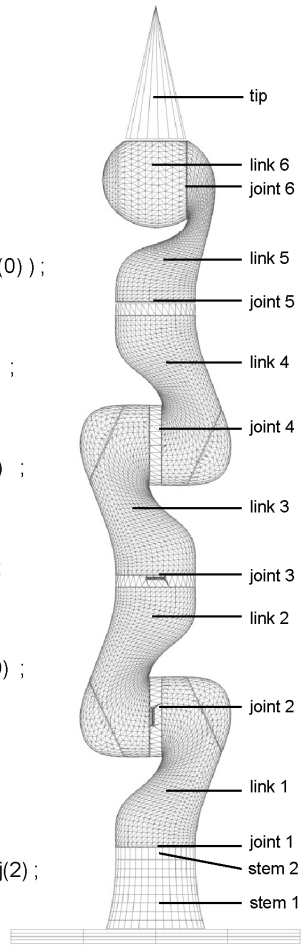
```
// phi:  vector with actual joint angles
// vel:  vector with actual joint velocities
// acc:  vector with actual joint accelerations
// _d:   indicates an ideal or desired state
// _old: indicates a state from the previous simulation step
// _TAU: ODE simulation time step size - default 0.01
// trajectory: desired circular trajectory given in joint angles

while (run)
{
        // increase trajectory index i and check if end reached
        i= i+1
        i= i MODULO (trajectory.length)

        // store desired joint angles from previous simulation step
        phi_d_old = phi_d
        vel_d_old = vel_d

        // get next desired trajectory point at position i, given in joint angles
        phi_d = trajectory.get(i)

        // Calculate desired joint angle velocity and joint angle acceleration
        // by differentiation;
        vel_d = (phi_d - phi_d_old)/_TAU
        acc_d = (vel_d - vel_d_old)/_TAU

        // calculate feedback command, by comparing desired positions (phi_d_old)
        // and actual positions (phi)
        feedback = controller.get_feedback_command(phi_d_old, phi)

        // predict feedforward command - internally using previously loaded LWPR class
        feedforward = controller.get_feedforward_command(phi_d_old, vel_d_old, acc_d)

        // calculate composite signal
        composite = feedback + feedforward

        // apply calaculated joint forces, correct joint errors
        // and simulate a next step in ODE
        ODE.setJointForce(Composite)
        ODE.clearJointErrors()
        ODE.step(_TAU)

        // store real joint angles and velocities from previous simulation step
        phi_old = phi
        vel_old = vel

        // update actual joint states and velocities from ODE
        ODE.getJointStates(phi,vel)

        // calculate real acceleration after having applied the composite signal
        acc = (vel-vel_old)/_TAU

        if (learn) // learning flag
        {
                // online learning
                // input: last real joint angles, joint velocities from the previous
                // simulation step
                // target: applied composite command leading to actual acceleration
                lwpr.learn(phi_old, vel_old , acc,  composite)
        }

        // Update rendering in OpenGL
        OpenGL.update()
}
```

Appendix B - DCG-File Specification

Tag	Purpose	Value(s)	Remarks
B	Defines a body object	Name	All the following tags are associated to the body until it reads a ";"
o	Defines the object type	0 = Sphere 1 = Box 2 = Capped cylinder 3 = Trimesh	
f	Defines a filename to a obj file	Name of the trimesh obj-file	Only used for trimeshes
s	Size of an object	Box = sx,sy,sz Sphere= radius Capped cylinder= radius, height	This tag has no effect on trimesh objects
m	Density of an object	Density	Is only used for objects that use no direct weight setting (see w tag)
c	Colour of the object	Red, green, blue	
fix	Defines if a body can be moved or is fixed	If set then body is "fixed" else mobile	
I	Inertia tensor	3x3 matrix	
X	Centre of mass	x,y,z	
w	Mass of an object	Mass	This tag is only used for trimesh objects.
J	Defines a joint object	Joint name	All the following tags are associated to the joint until it reads a ";"
a	Indicates the start angle at a joint	Degrees	Used to create specific arm start positions
p	Joint angle restrictions	Min , max	Restricts the joint limits to min and max angles. If p is not set, there are no joint angle restrictions.
t	Translation tag	x,y,z	Used to set the position of joints relative to a parent body and a child joint.
d	Rotation tag	rx, ry, rz	Same like "t" tag for rotational transformations.

Appendix C - DLW III in DCG-Format

```
<DCGraph>

B floor <t(0 0 -0.052)> o(0) m(10) s(2 2 0.1 2) c(0.2 0.2 1) fix;

B stem1 <t(0 0 0) > f[obj/base.obj] o(3) m(10) c(0.15 0.15 1) fix;
B stem2 <t(0 0 0)> f[obj/hat.obj] o(3) m(10) c(0.7 0.7 0.7) fix;

B link1 <> f[obj/dlr_link1.obj] o(3)  c(0.15 0.15 1)
                        l( ... ) X(...)       w( ... );
J stem2-link1 < t(0.0 0.0 0.310) d(90 0 -1 0) || d(90 0 1 0) t(0.0 0 0) >  a(0) ) ;

B link2 <>  f[obj/dlr_link2.obj] o(3) c(0.5 0.5 1)
                        l( ... ) X(...)       w( ... );
J link1-link2 < t(0.0 0.0 0.0) d(90 1 0 0) d(-90 1 0 0) || t(0 0 0.0) > a(-45)  ;

B link3 <> f[obj/dlr_link3.obj] o(3)) c(0.15 0.15 1)
                        l( ... ) X(...)       w( ... );
J link2-link3 < t(0.0 0.0 0.4)  d(90 0 -1 0) || d(90 0 1 0) t(0.0 0 0) >  a(90)   ;

B link4 <> f[obj/dlr_link4.obj] o(3) c(0.5 0.5 1)
                        l( ... ) X(...)       w( ... );
J link3-link4 < t(0.0 0.0 0.0) d(90 1 0 0)  d(-90 1 0 0) || t(0 0 0) >  a(90)  ;

B link5 <> f[obj/dlr_link5.obj] o(3)  c(0.15 0.15 1)
                        l( ... ) X(...)       w( ... );
J link4-link5 < t(0.00 0.00 0.39)  d(90 0 -1 0) || d(90 0 1 0) t(0 0 0) >  a(0)  ;

B link6 <> f[obj/dlr_link6.obj] o(3) c(0.5 0.5 1)
                        l( ... ) X(...)       w( ... );
J link5-link6 < t(0 0.0 0.0)  d(0 0 -1 0) || d(90 0 1 0)  t(0 0 0) > a(0)  ;

B link7 <> f[obj/tip.obj] o(3)   c(0.8 0.25 0.25)
                        l( ... ) X(...)       w( ... );
J link6-link7 < t(-0.06 0.0 0.0)  d(90 0 -1 0) || d(90 0 1 0)  t(0 0 0) > a(0) j(2) ;

</DCGraph>
```



Appendix D - Paramfile Specification

```
configfile                    dcg/dlrHiRes.dcg
-----------------------------------------------
winSizeWH                 700     700
camPosXYZ                 18.0    -15.0  10
camFocXYZ                 0.0     0.0    0.0
camHeightAngle            5
-----------------------------------------------
refreshRate               10
TAU                       0.01
-----------------------------------------------
bodyNumber                9
bodyOffsetXYZ             -0.19 0.0 0.0
-----------------------------------------------
posfile                   data/POS
anglefile                 data/PHI
velfile                   data/VEL
accfile                   data/ACC
posdfile                  data/POS_D
angledfile                data/PHI_D
veldfile                  data/VEL_D
accdfile                  data/ACC_D
torquefile                data/TOR
feedbackfile              data/F
feedforwardfile           data/FF
-----------------------------------------------
PID                       0 60  0.0  40
PID                       1 70  0.0  80
PID                       2 50  0.0  30
PID                       3 60  0.0  50
PID                       4 60  0.0  75
PID                       5 10  0.0  10
-----------------------------------------------
SINE_LENGTH               20
%SINE                     joint ampl freq phase
SINE                      0    1.0   1        0
SINE                      1    1.0   0.5      0
SINE                      2    0.5   0.25     0
SINE                      3    1.8   0.125    0
SINE                      4    2.0   0.0625   0
SINE                      5    2.0   0.0313   0
-----------------------------------------------
trajRefreshRate           10
trajScaling               0.45

lwprfile_online           data/model_online
lwprfile_batch            data/model_batch
keyboardStep              0.01
```

Appendix E - LWPR Parameters

Parameter	Explanation
add_proj	Allow(disallow) the addition of projections (to the initial number of projections specified by init_n_reg) if deemed necessary. This acts as a debugging variable.
updateD	Allow(disallow) the gradient descent update of the Distance Metric D
useNorm	Normalisation Parameter, Allow(disallow) normalisation
w_prune	If a training data elicits responses greater than w_prune from 2 local models, then, the one with the greater error is pruned.
w_gen	A new local model is generated if no existing model elicits response (activation) greater than w_gen for a training data.
initD	Initial values of distance metric
init_lambda	Initial forgetting factor
tau_lambda	Annealing constant for the forgetting factor
final_lambda	Final forgetting factor
add_threshold	The mean squared error of the current regression dimension is compared against the previous one. Only if the ratio of the nMSEr-1/nMSEr < add_threshold, a new projection is added. This criterion is used in conjunction with some other checks to ensure that the decision is based on enough data support.
meta	Allow (disallow) second order learning of the distance metric adaptation.
meta_rate	To be used if second order learning is enabled
initR	Number of projections to start regression with (Usually start with 2, one for regression and the other to compare whether an additional dimension should be used.
alpha	Distance Metric learning rate initialisation for gradient descent (currently implements only equal rates along all dimensions, i.e. initialises all the learning rates with the same parameters. (If you see lots of large update warnings or instability in convergence, you have too large a learning rate (alpha). If the Mean Squared Error is decreasing but the convergence is slow, you might try increasing the learning rate.)
gamma	Multiplication factor for the regularisation penalty term in the optimisation functional
blend	1: Allows weighted mixing of results from overlapping receptive fields. 0 : Winner-take-all prediction.

Parameters adapted from LWPR implementation (Toussaint, 2005).