Dynamic movement primitives and reinforcement learning for adapting a learned skill

Jens Lundell

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo August 3, 2016

Thesis supervisor:

Prof. Ville Kyrki
Prof. George Nikolakopoulos

Thesis advisor:

M.Sc. Murtaza Hazara

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Jens Lundell

Title: Dynamic movement primitives and reinforcement learning for adapting a learned skill

Date: August 3, 2016 Language: English Number of pages: 9+81

Department of Electrical Engineering and Automation

Professorship: Automation Technology (AS3004)

Supervisor: Prof. Ville Kyrki, Prof. George Nikolakopoulos

Advisor: M.Sc. Murtaza Hazara

Traditionally robots have been preprogrammed to execute specific tasks. This approach works well in industrial settings where robots have to execute highly accurate movements, such as when welding. However, preprogramming a robot is also expensive, error prone and time consuming due to the fact that every features of the task has to be considered. In some cases, where a robot has to execute complex tasks such as playing the ball-in-a-cup game, preprogramming it might even be impossible due to unknown features of the task. With all this in mind, this thesis examines the possibility of combining a modern learning framework, known as Learning from Demonstrations (LfD), to first teach a robot how to play the ball-in-a-cup game by demonstrating the movement for the robot, and then have the robot to improve this skill by itself with subsequent Reinforcement Learning (RL). The skill the robot has to learn is demonstrated with kinesthetic teaching, modelled as a dynamic movement primitive, and subsequently improved with the RL algorithm Policy Learning by Weighted Exploration with the Returns. Experiments performed on the industrial robot KUKA LWR4+ showed that robots are capable of successfully learning a complex skill such as playing the ball-in-a-cup game.

Keywords: Learning from Demonstrations, Dynamic Movement Primitives, Reinforcement Learning

AALTO-UNIVERSITETET
HÖGSKOLAN FÖR ELEKTROTEKNIK

SAMMANDRAG AV
DIPLOMARBETET

Författare: Jens Lundell

Titel: Dynamiska rörelseprimitiver och förstärkande inlärning för att anpassa en lärd färdighet

Datum: August 3, 2016 Språk: Engelska Sidantal: 9+81

Instutionen för elektroteknik och automation

Professur: Automationsteknologi (AS3004)

Övervakare: Prof. Ville Kyrki, Prof. George Nikolakopoulos

Handledare: M.Sc. Murtaza Hazara

Traditionellt sett har robotar blivit förprogrammerade för att utföra specifika uppgifter. Detta tillvägagångssätt fungerar bra i industriella miljöer var robotar måste utföra mycket noggranna rörelser, som att svetsa. Förprogrammering av robotar är dock dyrt, felbenäget och tidskrävande eftersom varje aspekt av uppgiften måste beaktas. Dessa nackdelar kan till och med göra det omöjligt att förprogrammera en robot att utföra komplexa uppgifter som att spela bollen-i-koppen spelet. Med allt detta i åtanke undersöker den här avhandlingen möjligheten att kombinera ett modernt ramverktyg, kallat inläraning av demonstrationer, för att lära en robot hur bollen-i-koppen-spelet ska spelas genom att demonstrera uppgiften för den och sedan ha roboten att själv förbättra sin inlärda uppgift genom att använda förstärkande inlärning. Uppgiften som roboten måste lära sig är demonstrerad med kinestetisk undervisning, modellerad som dynamiska rörelseprimitiver, och senare förbättrad med den förstärkande inlärningsalgoritmen Policy Learning by Weighted Exploration with the Returns. Experiment utförda på den industriella KUKA LWR4+ roboten visade att robotar är kapabla att framgångsrikt lära sig spela bollen-i-koppen spelet.

Nyckelord: inlära av demonstrationer, dynamiska rörelseprimitiver, förstärkande inlärning

Preface

To my uncle who recently passed away. May you rest in peace.

Otaniemi, August 3, 2016

Jens Lundell

Contents

Symbols and abbreviations

Symbols

$\mathbf{A}$	differencing matrix
c_i	centre of the Gaussian or von Mises basis function i
$\mathbf{f}_{target}$	stacked force vector for all demonstrations
f	forcing function
g	goal of a trajectory
N_w	number of basis functions
$\mathbf{S}$	stacked scaling vector for all demonstrations
t	time
t_{learn}	learning time
w_i	weighting factor i
$\dot{x}$	temporal derivative of the phase variable
x	phase variable
x_0	initial value of the phase variable
y	position
$\dot{y}$	velocity
$\ddot{y}$	acceleration
α_x	decay factor of the discrete canonical system
α_z	stiffness of the transformation spring-damper-system
β_z	damping of the transformation spring-damper-system
$\mathbf{\Gamma}_i$	compound diagonal basis function matrix for weighting factor i
ϵ	exploration rate
θ	policy parameter
Ψ_i	basis function i
$\mathbf{\Psi}_i^j$	diagonal basis function matrix for demonstration j and weighting factor i
Σ	covariance matrix
σ	standard deviation of the Gaussian function
τ	time scaling factor (duration of movement)

List of acronyms

BIC Bayesian Information Criterion

DMP Dynamic Movement Primitives

DoF Degrees of Freedom

DTW Dynamic Time Warping

DP Dynamic Programming

EM Expectation-Maximization

eNAC episodic Natural Actor Critic

eRWR episodic Reward Weighted Regression

FRI Fast Research Interface

F/T Force/Torque

GMM Gaussian Mixture Model

GMR Gaussian Mixture Regression

GP Gaussian Process

GPIC Generalized Path Integral Control

GPR Gaussian Process Regression

HMM Hidden Markov Model

HSV Hue Saturation Value

HW Hardware

KCP KUKA Control Panel

KL Kullback-Leibler

KRC KUKA Robot Controller

KRL KUKA Robot Language

LfD Learning from Demonstration

LGP Local Gaussian process

LOOCV Leave One Out Cross Validation

LSOGP Localized Sparse Online Gaussian Process

LWL Locally Weighted Learning

LWPR Locally Weighted Projection Regression

LWR Light-Weight Robot

LWR Locally Weighted Regression

ML Machine Learning

MNS Mirror Neuron System

MW Middleware

NLP Non-Linear Programming

OPS Orocos Program Script

Orocos Open Robot Control Software

PbD Programming by Demonstration

PDE Partial Differential Equation

PI^2 Policy Improvement with Path Integral

PoWER Policy Learning by Weighted Exploration with the Returns

PTP Point-To-Point

RGB Red Green Blue

RL Reinforcement Learning

RMS Root Mean Square

ROS Robot Operating System

SEDS Stable Estimator of Dynamical Systems

SOC Stochastic Optimal Control

SW Software

VPG 'Vanilla' Policy Gradient

1 Introduction

Robots are used in human environments today more than ever before. Examples of such robots are simple vacuum cleaning or lawn cutting robots. Albeit robots becoming more popular, they are still mainly used in industrial settings as welders, assemblers and much more. The reason why robots are excelling in industrial settings is because the environment is for the most part static and that the task at hand is known a priori. Because of this, the robot can be preprogrammed to precisely follow specific trajectories. However, in an constantly changing environment, as in our homes, or with complex tasks, as playing ping pong [61], the robot cannot be preprogrammed as before. Thus, for a robot to learn in these situations, one possible and frequently used solution is to learn from a human demonstrator.

The concept of having a human to teach the robot by interacting with it is known as *Learning from Demonstration* (LfD), as well as *Programming by Demonstration* (PbD) or *imitation learning*. LfD has enabled robots to acquire complex tasks such as playing the ball-in-a-cup game [50], playing ping pong [61] and much more [10, 63]. For the robots to acquire a movement, they first and foremost have to record the demonstrated movement either by their internal sensors or by an external monitoring system. Then the responsibility is shifted over to the robot, which in turn is expected to learn a reasonable representation of the movement from the recorded information. The learning is fulfilled with the guidance of a learning algorithm. However, as the reproduced movement by the robot can face perturbations, the learning algorithm has to be able to produce such a representation of the movement that perturbations can be handled without risking failure.

LfD has attracted significant attention in the last couple of years where the number of papers written in the subject is a significant indicator [5, 60, 64]. There are several reasons behind the increased interest [3, 53]. The first reason is the reduced learning time compared to traditional approaches where a human had to program the robot off line. The second reason is that by demonstrating the movement to the robot, the user can also predict the behaviour in advance when the robot executes the learned skill as this should follow the demonstrated movement quite accurately. This, in turn, increases safety and is exceptionally important when integrating robots in human environments. Finally, the acquired movement is in line with human workspace and thus critical parts of the movement can be modelled very accurately.

Albeit all the positive sides of learning from demonstrations, problems as: improving and generalizing the learned skill still remains. The former of these problems originates from the basic idea of learning from a demonstration is not enough to master complex tasks as flipping a pancake [52] or playing the ball-in-a-cup game [50]. Hence, to really master a learned skill subsequent practise is needed. The method of practising as to improve the skill has already been implemented in robotics and is known as *Reinforcement Learning* (RL), where the robot tries to improve the learned movement by interacting with the environment and receiving some responses in form of rewards [48]. Based on these rewards, the learned movement will be adapted as to increase future rewards. This resembles how humans acquire a new skill: first we imitate it from another person and then we subsequently practice the skill where the

underlying idea is to fine-tune the movement until we can master it.

Although reinforcement learning nowadays is the standard approach to improve a taught skill, one problem still remains and that is how to safely explore new trajectories. To explore new trajectories, noise is added to the learned representation of the skill, resulting in a slightly new trajectory. However, generating the noise is not a trivial task and problems as: the magnitude of the generated noise and if it should be correlated or not is still an open question. For example in [50] and [49] the generated noise is dependent on the learned model. Intuitively this makes sense; however, it does not take into consideration the constraints of the executing system, which in both cases is a robotic arm. Recent studies [44], on the other hand, concentrates on generating noise which focuses on the constraints of the actual system. However, this has not yet been applied to improving a learned model represented as a *Dynamic Movement Primitives* (DMP). Hence, this thesis will study how to safely explore new trajectories for a skill modelled as a DMP? To answer this question the following objectives are to be achieved:

- the skill has to be modelled as a dynamic movement primitive,
- an RL algorithm for improving the modelled skill has to be selected,
- the exploration has to be safe.

To test how well the new approach of generating safe trajectories works, a challenging enough real world experiment is needed. One experiment which can be seen as a benchmark problem in robotics is the ball-in-a-cup game [50]. The game consist of a ball attached to a string which in turn is attached to the bottom of a cup. The objective of the game is to get the ball into the cup. This is achieved by inducing a movement on the ball by quickly moving the cup back and forth and then pulling it up and moving the cup under the ball. The game is simple by its nature but not that easy to learn as it requires a fast and precise movement, where even small changes of the movement can have a drastic impact on the trajectory of the ball. The experiment is of interest as a robot cannot get the ball into the cup by merely executing the movement produced by the initially learned model, and thus subsequent RL is needed to successfully master the skill.

This thesis is organized as follows: the second chapter introduces and compares several state-of-the-art LfD approaches. Based on the results, DMP is chosen to model movements. Chapter 3, in turn, is devoted to explaining DMP in more detail. As reinforcement learning is used to improve the learned movement, Chapter 4 is dedicated to explaining this. First the basics behind RL are explained, and later the state-of-the-art algorithms are introduced. In Chapter 5 the testbed is introduced. This consist of both the hardware and the already implemented and newly developed software supporting the experimental part in this thesis. The experiments and results are presented in Chapter 6. These are also critically discussed in the same chapter. Conclusions and ideas for future work are presented in Chapter 7.

2 Learning from Demonstrations

For a robot to perform a task, the traditional approach has been direct programming, where the desired position of the robot is manually specified by the user [54]. This, however, is time consuming, error prone and requires full knowledge of the task, forcing the programmer to be an expert in the field. Moreover, it is also not possible to scale or reuse the program to new tasks. Therefore, to remedy these shortcomings the LfD framework has been developed. This framework allows a robot to acquire new skills in a simple and fast manner by transferring it from a human to a robot through demonstrations. Hence, as the input data to the robot is labelled, for example data points of the demonstrated trajectory, the learning belongs to the supervised learning class of machine learning [5]. In supervised learning the problem is learning the input-output mappings from the provided data [76].

For acquiring the new skill, the problem can be split into two parts: a teaching and a learning part [5]. The teaching part consists of methods for teaching the robot, *i.e.* how to transfer the skill from a human to the robot. In the learning part, the responsibility is on the robot to learn a representation of the movement based on the demonstration.

This chapter is devoted to explaining both the learning and teaching parts. As the robot acquire a new skill by first having it demonstrated and then learning the representation of the movement, teaching will be covered before learning in this chapter.

2.1 Methods for Teaching a Skill

Teaching a robot a new skill deals with the problem of how to transfer the skill from a human demonstrator to a robot, or more exactly how to transfer the information known by the teacher, for example the arm movement when flipping a pancake, to the robot in a correct way. The problem with transferring information is finding a mapping from the teacher to the learner, or vice versa. This problem is known as the correspondence problem [64], which deals with the diversity in morphology between the demonstrator and the imitator. To actually transfer a skill from a demonstrator to a robot can be realized based on three different methods [54]: *kinesthetic teaching*, *teleoperation*, and *observational learning*.

In kinesthetic teaching the demonstrator grabs the robot and moves it manually. Unfortunately, for this to be possible the robot has to be both small and able to compensate for its own weight through active control such as gravity-compensation control [53]. Despite meeting all these requirements, kinestethic teaching still has its shortcomings as the human demonstrator can have problems teaching a complex movement to a robot consisting of multiple *Degrees of Freedom* (DoF) as this requires moving several joints simultaneously. However, the advantages kinesthetic teaching provides are that the demonstrator can effectively demonstrate the whole movement or only a part of it. Moreover, the speed of the demonstration can easily be adjusted by the demonstrator. Applications of kinesthetic teaching have been, for example, wood planing [59], playing the ball-in-a-cup game [51] and flipping pancakes [49].

In teleoperation, the teacher operates the robot at a distance by using a controller. The controller can, for instance, be a joystick or a haptic device. Teleoperation provides a simple mean of teaching the robot and the teacher is not bound to be in contact with the robot. However, being too far from the robot to control is not feasible either due to the time delay between commanding the robot until it receives the command. This is of concern when controlling robots on planets other than Earth. Other disadvantages are restrictions in sensing the limitations and possibilities of the robot, performing fast movements, and controlling a robot consisting of multiple DoF. Despite these limitations, teleoperation has been utilized in many applications such as robotic kicking motions [14], flying a robotic helicopter [65], robotic assembly tasks [24], and object grasping [74].

In contrast to the two previous methods, which forced the demonstrator to either be in direct contact with the robot or teleoperating it directly, another teaching method, observational teaching, which shadows a demonstrator, is also a possibility [5]. To enable observational learning, data has to be externally collected from the demonstrator, either by using sensors on the teacher or by external observations through cameras. Benefits of using observational learning is that the demonstrator uses his own body to show the movements, alleviating the problem of controlling a complex robot. However, with observational learning the correspondence problem is significant as the the demonstration needs to be mapped from the body of the demonstrator to the joints of the robot. Moreover, the whole setup, in contrast to the other teaching methods, is far more complicated as it requires either a motion capture system, video cameras, or sensors directly attached to the demonstrator. Despite the shortcomings, observational learning has resulted in applications such as object manipulation [98].

As for now, three different methods for teaching a robot have been considered. From the human demonstrations the robot receives data which will be used as input for the learning part, where the robot will have to learn how to successfully reproduce the taught demonstration on its own. This will be covered in the next section.

2.2 Methods for Learning A Skill

After the demonstrations have been conducted, the robot has to be able to transform the provided data, represented as a set of state-action pairs, into a policy which can be used by the robot to reproduce the demonstrated movement [5]. The learning can be split into two categories [11]: symbolic learning and trajectory learning, where their individual learning schemes are visualized in Figure 1. The former is used to learn at a more abstract level, where a skill is composed of several primitive actions represented as symbols. In the latter, the movement is learned at a low level and is represented as a trajectory. Next the symbolic learning will be introduced, and later the focus will be shifted towards trajectory learning.

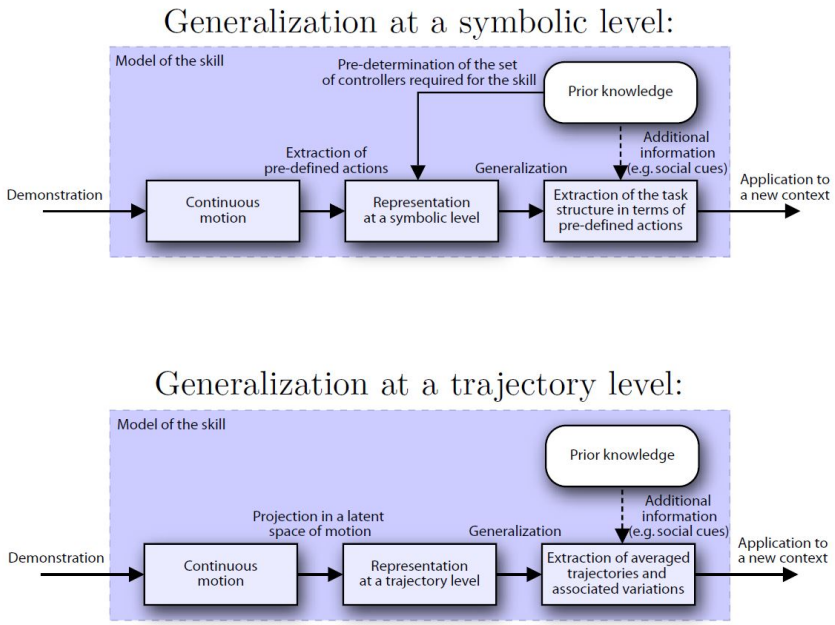


Figure 1: The skill can either be represnted at a symbolic level (upper figure) or at a trajectory level (lower figure). The fundamental difference is that at a symbolic level pre-defined actions are extracted, while on a trajectory level the motion is projected on a latent space. (Source: [11]).

2.2.1 Symbolic Learning of Skills

In symbolic learning the actions are represented as symbols. Thus, to learn a task, one possibility is to segment it and then encode it as a sequence of predefined symbols. In [28] actions are represented in a hierarchical manner, and learning is based on extracting symbolic rules for handling objects. An example of this is setting up a table, where the soup plate has to be on top of the main dish plate; thus, the order of placing objects becomes important. The resulting system consists of several building blocks depicted in Figure 2.

To encode a skill into a set of symbolic rules, the first thing is to demonstrate the task at hand. Thus, the first building block, named *Demonstrations*, provides one or several demonstrations of the task to the system. From the demonstrated tasks, the second building block, *Segmentation*, is responsible for composing the actions as primitives and mapping them to individual states. A state can be seen as the impact an action has on the current world state, e.g. "the main dish plate was set 10 cm below the glass". As several of these states correlates, the *State Generation* block search for states representing similar subtasks. Then in the *Task Generalization* block, the task is generalized by determining in what order the states have to occur, and if some states can be removed due to their direct impact on the final result is neglectable. One example of a symbolic rule is that a dishwasher has to be opened before filling it. As the constraints are extracted from the demonstrations, more demonstrations lead to more constraints, depicted in Figure 3. With both the task

space and task restrictions in mind, the robot should create a plan to reach the goal state without violating the predefined rules, which is done in the *Planning* block. The plan should tell which object to move and where to move it as to fulfil the task. This plan is later executed in the *Execution* block. This step is in charge of grasping the object and manipulating it; thus, it is connected to the *Grasp Planning* and *Visual Servoing* block. The *Perception* block, which is connected to several other blocks, is of importance as it is used for pose estimations of the object, where the gathered information is especially utilized in the planning phase. The sensory perception and modelling, used for pose estimation, is usually very difficult and can be seen as the bottleneck of the system, affecting how complicated tasks the robot can perform [28].

All in all, symbolic learning is effective at learning high-level tasks such as setting up a table and placing boxes [28]. However, to effectively learn a skill a lot of prior knowledge is needed to avoid unnecessary encoding of conditions. This, on the other hand, is time consuming as knowledge is gathered from demonstrations and several demonstrations are needed to provide enough knowledge. In contrast to symbolic learning, there also exist another learning method called trajectory learning which learns, as the name suggests, the skill on a trajectory level. This learning method is introduced in the next section.

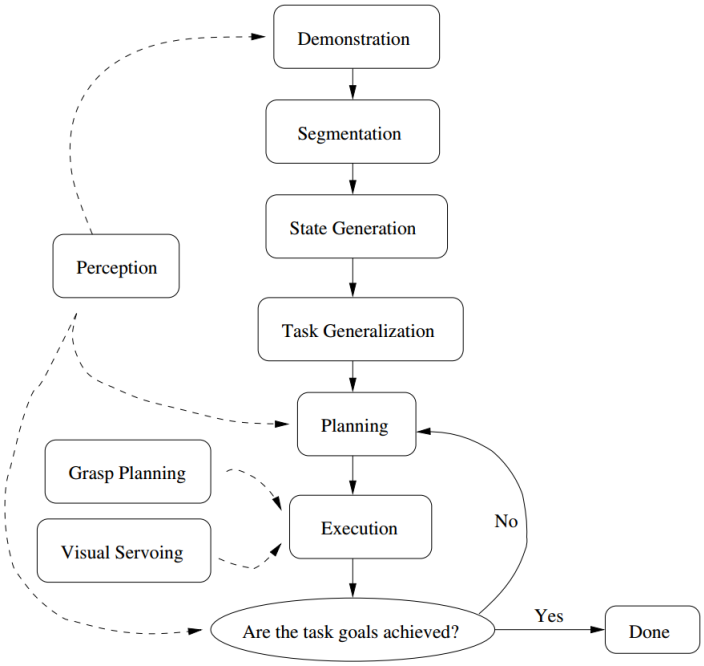


Figure 2: A symbolic learning flowchart. (Source: [28]).

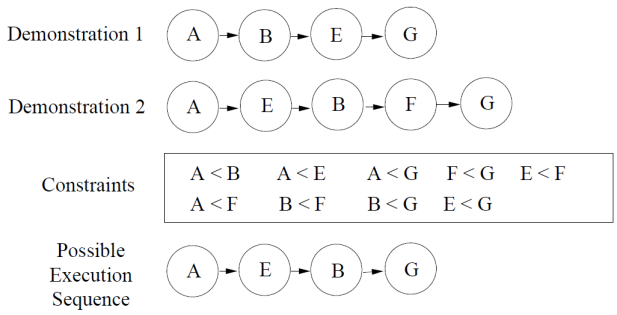


Figure 3: From Demonstrations (1 and 2) several constraints can be generated and state F can be neglected. (Source: [28]).

2.2.2 Trajectory Learning of Skills

From the symbolic learning, a high level policy could be derived. In contrast to this, trajectory learning seeks to find a policy able to reproduce the low level of the taught movement, such as joint positions, motor commands, or Cartesian position of the end effector. Traditionally the encoding of a motion has been direct by using *splines* and *Bézier curves* [36] [97]. However, this approach of learning a skill on trajectory level is limited to only learning the position over time and excludes other potentials variables, such as velocity and acceleration, from the task space. More recent research focuses on representing the motion as a statistical model or by a dynamical system [11]. Moreover, promising studies [17, 37, 56, 62, 84] point toward encompassing the dynamics of a human movement into the actual encoding of the trajectory.

Statistical modelling of skills

The idea behind exploiting statistical learning for learning a model of the taught skill is to cope with the high variability originating from the demonstrations [11]. The foundation block in statical learning is a mathematical concept known as *regression*. Regression analysis seeks to estimate the relationship between the input and output data. For trajectory learning the input are robot states and the output are robot actions [5]. In the following sections several well known regression methods, used to encode trajectories for a robot, are presented.

Locally weighted learning

Locally Weighted Learning (LWL) refers to a family of non-parametric regression methods [6]. For fulfilling the regression and subsequently obtaining the output, LWL only concentrates on data points which are close (based on some metric) to the input value, hence the prefix “local”. This is visualized in Figure 4.

In the LWL family, two algorithms exist: *Locally Weighted Regression* (LWR) and *Locally Weighted Projection Regression* (LWPR) [25]. The main difference between these two are that the former method, LWR, is memory based, whereas the latter

is not. Memory based means that before the regression can take place all the data points have to be stored. Thus, the computational complexity is high, more exactly it is $\mathcal{O}(n^2)$. Therefore, to lower the computational complexity, LWR has been extended to deal with incremental regression, making it non-memory based. The extension resulted in the LWPR algorithm with a linear computational complexity of $\mathcal{O}(n)$, which is much lower in comparison to the LWR algorithm. Thus, if the input data (number of samples) is high LWPR should be used, otherwise LWR suffices.

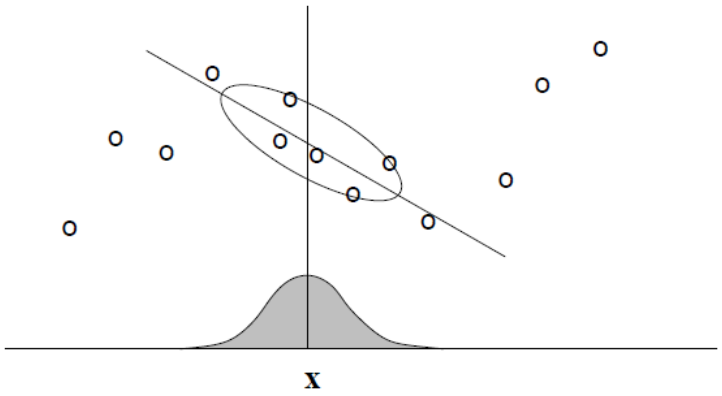


Figure 4: In LWL, all data points are weighted with a kernel according to their closeness to the current value of interest, given here as **x**. (Source: [25]).

Gaussian process regression

A *Gaussian Process* (GP) is defined as a set of random variables, where any finite set of this random variables is itself a joint Gaussian distribution [76]. Thus, a GP is made up of a mean and a covariance function, where the input-output mapping is learned with *Gaussian Process Regression* (GPR) (See Figure 5). The advantages with GPR is that there are less open parameters to tune in comparison to other regression methods such as LWPR and *Gaussian Mixture Regression* (GMR).

On the other hand, GPR requires inverting the covariance matrix resulting in a computational complexity of $\mathcal{O}(n^3)$, where n is the number of input points. This computational complexity is much higher in comparison to both LWR and LWPR, making it useless for online learning. Because of this fact, effort has been invested into reducing the computational complexity for a GPR, and two main solutions have been proposed in [66] and [70].

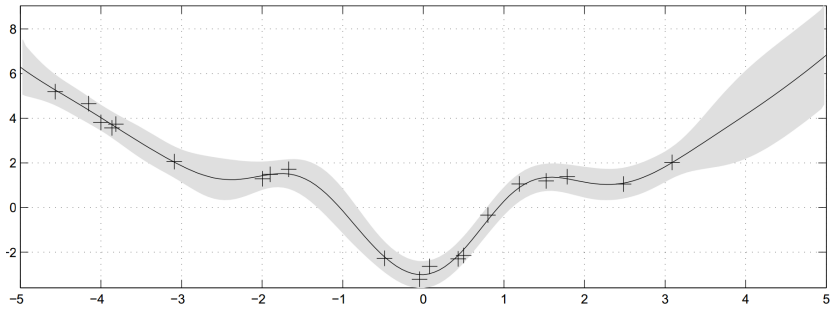


Figure 5: This figure shows the learned continues mean function (solid line) and variance function (gray area) in a GP given the input data (crosses). The learning is fulfilled with GPR. (Source: [76]).

In [66] the computational complexity of the GP is reduced by introducing similar approximations as in LWL effectively reducing the usage of data points. This reduction of data points resulted in what is know as the *Local Gaussian process* (LGP) which performs faster than the original GPR. Although the reduction in data point usage, the results showed that accuracy of the regression was similar for LGP and GPR.

Another approach for speeding up the GPR is presented in [70]. This approach utilizes sparse approximation techniques, where the number of data points are reduced by discarding all points which are not considered salient according to a specific metric. In their approach they use the *Leave One Out Cross Validation* (LOOCV) algorithm to discard all non-salient points which, in combination with regression, resulted in the *Localized Sparse Online Gaussian Process* (LSOGP).

Gaussian mixture regression

To use GMR for encoding a trajectory as a statistical model, some initial steps needs to be fulfilled. All the steps are depicted in Figure 6, where it is clearly seen that GMR is only used as the final step. In the first steps, the data obtained from one or several demonstrations is encoded as a statistical model, either as a *Hidden Markov Model* (HMM) or a *Gaussian Mixture Model* (GMM) [22]. To retain any reliable information from the statistical models they also have to be trained. Once this is fulfilled, the statistical model, together with the input query, acts as the input to the GMR which produces a sought output to the given input. For example the input query could be a Cartesian position and the output from the GMR could be the needed joint torques to move the end effector of the robotic arm to the specific input position.

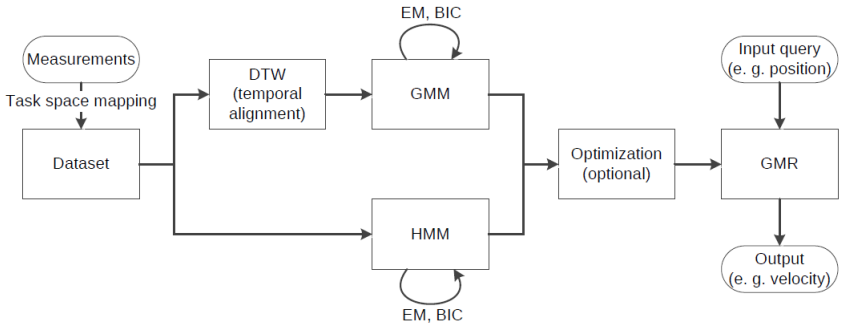


Figure 6: To encode a trajectory with GMR it is required to do some preprocessing steps, and either encode the initial data as a HMM or a GMM. (Source: [87]).

A GMM consists of several Gaussian distributions, each with their own mean and covariance matrix [25]. This model is applied when the data set consists of several clusters, where one Gaussian distribution per cluster is needed to model the data set in a correct way (see Figure 7). In this thesis, the data set consists of a trajectory, hence each Gaussian models a part of that trajectory. To initialize a GMM, each Gaussian in the mixture is assigned an individual mean and covariance matrix as well as a prior which describes the initial likelihood of one Gaussian w.r.t. the others in the mixture.

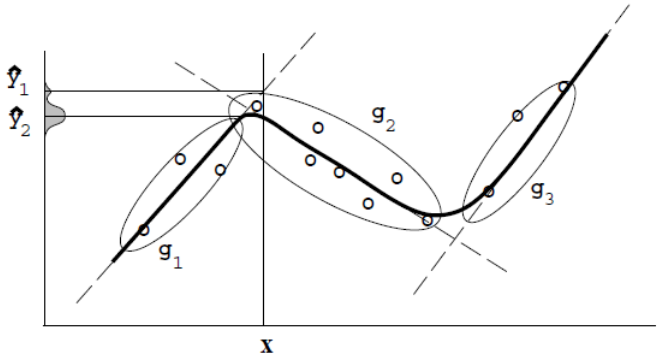


Figure 7: The predicted output $\hat{y}$ is produced as a weighted mixture of the Gaussians g_1, g_2 and g_3. The Gaussians should completely model the density of the input data. (Source: [25]).

As can be seen in Figure 6, there is a temporal alignment step before the actual encoding of the GMM. This is a consequence of the fact that GMM also encodes the temporal information as any any other variable, and as demonstrations most probably vary in time, they need to be temporally normalized to minimize the variance between all demonstrations [23]. Otherwise, using only the raw data, the resulting model would be inaccurate due to a high variance (see Figure 8). Aligning the demonstrations is realized by using an algorithm known as *Dynamic Time Warping* (DTW) [60]. This algorithm takes all the data samples of the trajectories

and aligns them in a pairwise fashion as to minimize the distance between them according to some metrics. To align them, as to achieve an optimal solution, *Dynamic Programming* (DP) has been utilized [9]. An example of the whole process from aligning different demonstrations with the DTW algorithm, to encoding the data as a GMM, to extracting the wanted output through a GMR is presented in [19]. Their approach, called Time-dependent Gaussian Mixture Regression (TDGMR), resulted in a learning algorithm able to generalize from already known manipulation tasks to new ones.

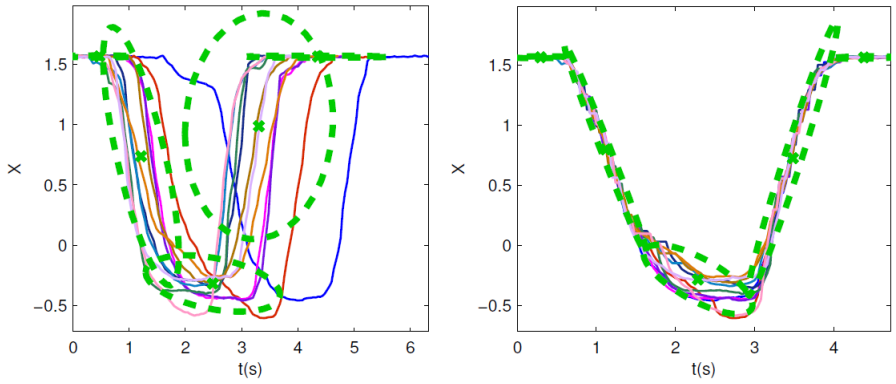


Figure 8: The left image depicts the GMM without DTW, whereas the right with DTW. The mean and covariance of the learned trajectory is depicted, respectively, as the green line and the ellipses. (Source: [60]).

Another possibility to using GMM in the GMR, is to use a HMM. A HMM consists of several distinct hidden states which all are responsible for generating the observation [75]. Moreover, the hidden variables satisfy the Markov property, meaning that future state is independent of the past states as long as the current state is known [31]. In our case, the states in the HMM are modelled as Gaussian distributions with their individual mean vector and covariance matrix. As the HMM usually consists of several coupled states, the transition from one state to another is modelled as a transition probability. In contrast to GMM, time itself is encoded as the probability to transition from one state to another, and thus the temporal alignment is usually embedded inside the actual encoding of the HMM [101].

By only assigning a number of Gaussians to the data set in hope to efficiently cover it is not enough, hence post training is necessary. To post train a GMM, the *Expectation-Maximization* (EM) algorithm is usually used [60]. Unfortunately the EM algorithm is not applicable for post trainig an HMM, but instead the Baum-Welch algorithm, which is a specific EM algorithm, is used [20]. Both algorithms, however, iteratively adapts the means and covariance matrices of the Gaussian distributions in order for them to effectively model the data points.

As the number of states in the HMM and number of Gaussians in the GMM is a free parameter, it is up to the user to manually choose the number he thinks will cover the data set in the best way. However, this is not a trivial task, where too many Gaussians will result in an over fitted model and too few Gaussians in an under fitted

model [18]. Hence, finding models which are neither substantially over nor under fitted is of great interest. To find potential candidates for the best model, a criterion known as *Bayesian Information Criterion* (BIC) has been developed [22, 60]. This criterion finds the model which best compromises between complexity and accuracy. The idea is to initially generate and train several models with varying number of Gaussians. The BIC value are then computed for each model based on the number of parameters and the maximized log-likelihood of the model, where a lower BIC value indicates a better fitted model.

As training several Gaussians with the EM algorithm can be cost intensive a faster approach was proposed in [60] where speed is traded for accuracy by applying a fast *k-Mean* clustering instead of training each Gaussian individually. When the training was over the BIC values were calculated as usually. The proposed approach, however, usually resulted in an over fitted model, but the total computational time was significantly less. When the model with the lowest BIC value was selected it was subsequently trained with the EM algorithm.

When the correct model has been finalized, it is passed on, potentially through an optimization step, to the GMR which is responsible to generate the desired output from the given input. The chosen model consists of Gaussians which can be visualized as a system of attractors resulting in a compact representation of the movement. Different inputs are required for the different mixture models; GMM requires time as input [22, 60], whereas HMM requires the current task space configuration [20]. The output from the mixture models, and thus input to the GMR, is their spatial information along with their covariance matrices. The resulting output obtained through regression is calculated as a linear combination of the conditional expectations of the input of each Gaussian component. The linear combination is weighted based on the probability of each Gaussian for a GMM. However, for a HMM the weight is also influenced by the transition probability from the previous state. The output from the regression can be interpreted for instance as the commanded Cartesian velocity.

2.2.3 Dynamical system modelling of skills

In contrast to encoding trajectories as statistical models, mainly two other methods where the trajectory is encoded as a dynamical system exist. One method encodes the trajectory as a *Stable Estimator of Dynamical Systems* (SEDS) [46] and the other as a DMP [40].

Stable Estimator of Dynamical Systems

The idea behind SEDS is to learn globally stable discrete movements such as *Point-To-Point* (PTP) movements as a dynamical system [46]. To be able to learn a feasible representation of the movement several demonstrations are needed as these acts as the input for the statistical encoding of the movement. For modelling the movement GMM is used. The global stability ensured by SEDS means that as long as the starting point of the movement lies within the task space it will converge to the given goal vector. The reproduced movements from the previous mention models

(HMM, GMM, etc.) could not ensure global stability, and in most cases not even local stability, where the starting point of the movement lies within a subspace of the task space.

As stipulated before, SEDS consist of a GMM and thus learning the unknown parameters of the model (priors, covariance matrices and mean vectors) is required. When Learning the unknown parameters the global stability criteria has to be fulfilled and thus the problem has to be formulated as a *Non-Linear Programming* (NLP) problem [46]. The NLP problem can always be solved using standard optimization techniques. However, due to the fact that the NLP problem is non-convex, the optimization step cannot ensure that the solution is globally optimal.

Dynamic Movement Primitive

A DMP represents a stable known dynamical system expressed as a spring-damper system modulated with a nonlinear function to enable reproduction of complex trajectories. The building block in the DMP framework is a *transformation system*, a *canonical system*, and a *forcing function*. The DMP framework is explained in more detail in Chapter 3.

2.3 Comparison

Up to this point several different methods, all used by the LfD framework, have been presented. These methods all have their individual advantages and disadvantages. However, in this thesis the skill to learn is how to play the ball-in-a-cup game (discussed in Chapter 6) which sets some requirements on the learned movement. First of all, learning the movement from one demonstration only is necessary. Secondly, the reproduced movement has to be smooth. With all this in mind, a thorough comparison between the discussed learning algorithms has to be conducted as this will opt as a reference for selecting the best method for learning the ball-in-a-cup game.

For learning to play the ball-in-a-cup game the symbolic learning approach is not applicable as this learns the task on a high level. The game can only be successfully played by the robot if it can execute accurate movements and for this to be possible controlling it on a low level is the only way.

As symbolic learning is not applicable for learning the skill of playing the ball-in-a-cup game, the skill has to be learned on trajectory level. The traditional trajectory learning techniques *splines* or *Bézier curves* is very limited and lacks flexibility as only position is modelled over time and because of this they are not considered either. Hence, the only LfD approach left is to represent the skill as a statistical model or as a dynamical system. However, as there exist several methods for both statistical and dynamical encodings of a skill, not many studies have been conducted to comparing all of these. The only comprehensive comparison is presented in [20], where HMM, LWL, LWPR, DMP and TDGMR are compared. The evaluation is based on five metrics comparing the similarity between the reproduced movement and the demonstrated one. The metrics used in the comparison were: the *Root Mean*

Square (RMS) error, the RMS error after DTW , norm of Jerk, learning time, and retrieval duration.

The RMS error depicts how well the reproduced trajectories follows the demonstrated one. It takes into consideration both the spatio and temporal information. The extended RMS error after DTW aligns all the demonstrations in time before calculating the RMS error. By doing this, emphasis is given to the spatial information, *i.e.* instead of comparing the trajectory along time the actual path which the robot follows is used as the metric. The norm of jerk evaluates how smooth the reproduced movement is. The metric is based on the RMS jerk quantification and is calculated from the derivation of acceleration, which sets a good standard for human motion smoothness [35]. The learning time evaluates the time it takes for the learning algorithms to finish its computations, whereas the retrieval duration is the time it takes for one iteration to finish the retrieval process.

In the comparison step in [20], three different movements were generated. From this set of movements, the learning algorithms have three attempts to reproduce the movement. The reproduction procedure was then repeated for different amount of states, ranges of perturbation, and dimensionalities. In Figures 9 and 10 the results are presented, where the former figure presents the influence with variable number of states and the latter with variable number of dimensionalities. The states are represented with the variable K which, in case of a DMP, also represent the number of basis functions and the dimensionalities by the variable D.

From Figures 9 and 10 the RMS error both with and without DTW produce seemingly equal results. Both for a high number of states and dimensionalities the error is extremely small, which in terms of the reproduction accuracy means that the deviation from the demonstrated trajectory is almost negligible. However, in Figure 9 the reproduction from a DMP is high with for low number of states, but is almost unaffected after a certain number. Hence, the reproduction accuracy for a DMP is highly dependant on the number of basis functions.

For a variable set of states HMM produces the largest norm of jerk. HMM is also, together with LWPR, among the worst performers when considering the norm of jerk with a variable number of dimensions. The lowest norm of jerk, on the other hand, is achieved by the DMP, and this, in comparison to the highest norm of jerk in respective figure, is about five times lower. An interesting point regarding the smoothness of the reproduced movement is that a DMP is able to produce a smoother movement than the actual demonstration. Furthermore, the number of states does only affect the reproduction smoothness by LWPR in a larger extend, whereas a larger number of dimensions seems to mostly impair the smoothness of HMM and LWPR, but it also affects the other in a smaller scale.

The learning time for a DMP and LWR is unaffected by the number of states, whereas they almost increases linearly for the rest of the methods. Although the same seems to apply for the retrieval time, this is actually not the case as the retrieval time for the LWR algorithm is so high, over $7 \times 10^{-2} s$, that it is outside the graphs. However, for the rest of the methods the low retrieval time is favourable because it allows online learning. The high variation in learning time for both HMM and TGMR originates from the inherited behaviour of the EM algorithm which always initializes

the training randomly and stops when a local maximum is found. Therefore, nothing can be said with certainty how many iterations it takes before a local maximum is found.

As a side note, the total learning time for the LWPR algorithm in Figure 9 is over 10 iterations with the dataset randomly shuffled at each iteration. Hence, for only a single iteration the computational time can be trimmed down with one order of magnitude.

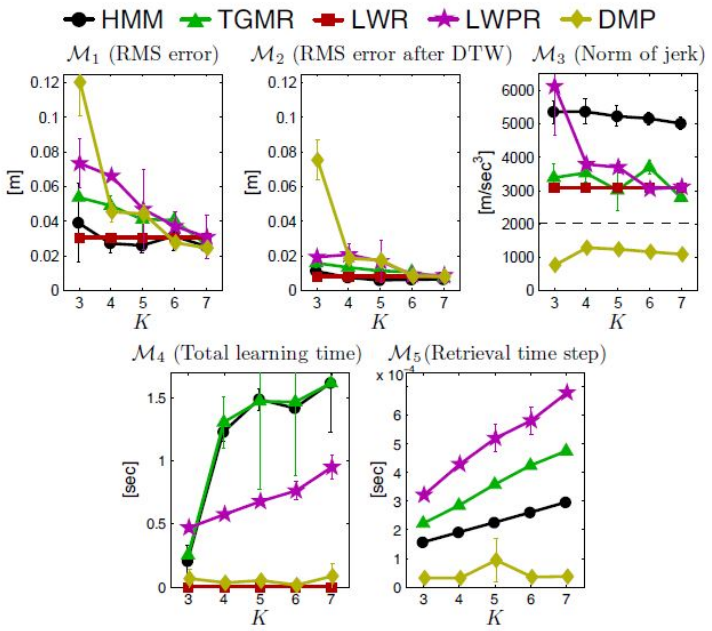


Figure 9: The evaluation results for different learning approaches over a variable amount of states but a fixed set of dimensionalities (D=7). The dashed line in the graph indicates the mean RMS jerk of the demonstrations. (Source: [21]).

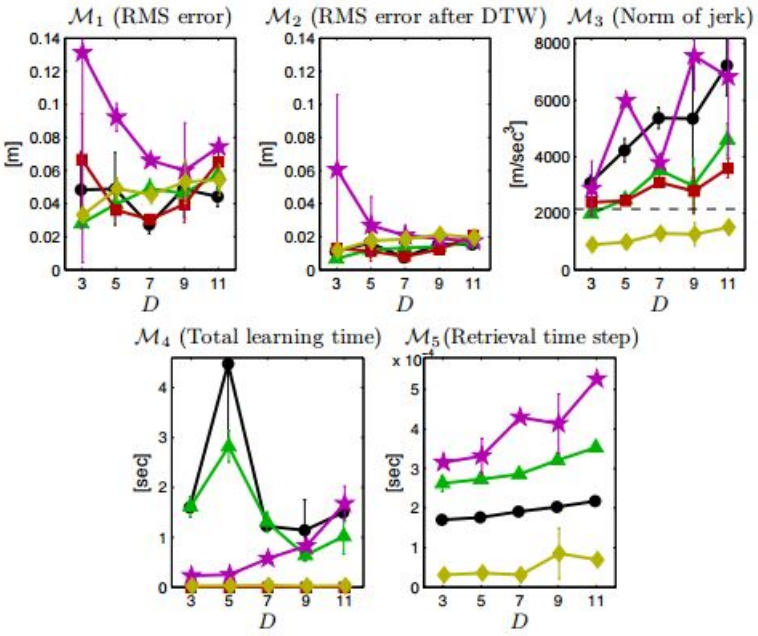


Figure 10: The evaluation results for different learning approaches over a variable amount of dimensionalities but a fixed number of states (K=4). The dashed line in the indicates the mean RMS jerk of the demonstrations. (Source: [21]).

2.4 Discussion

LfD is indeed a versatile approach for teaching a robot a new skill in relation to preprogramming. Not only is it more versatile but it is also significantly cheaper and faster. However, it is not a trivial task to first teach the robot and then have it learn the new skill. Several approaches exists for both teaching and learning. Thus, their individual advantages and disadvantages need to be considered before selecting one over the other.

As previously mentioned, the skill to be learned by the robot in this thesis is how to play the ball-in-a-cup game. To teach the movement three different methods exists: kinesthetic teaching, observational learning and teleoperation. Out of these three methods one has to be chosen for demonstrating how to play the game to a KUKA robotic arm used in the experimental part. To select the best possible teaching methods the advantages and disadvantages of the individual methods had to be compared.

Observational learning is indeed a good approach as it does not require controlling the robot at all for teaching the skill. However, the correspondence problem in observational learning is such a significant problem to solve that different methods are preferred. For teaching the robot directly the demonstrated movement has to be both accurate and executed in high speed. With teleoperation neither of these requirements can be fulfilled, thus excluding this mean of teaching. With all this in mind, kinesthetic teaching avoids the correspondence problem as the demonstration is performed directly on the robot by grabbing it and showing the

movement. This also ensures that the movement can be taught with high speed and precision. Thus, kinesthetic teaching is chosen as the teaching method to be used later in the experimental part.

As the movement has been transferred through kinesthetic teaching, the responsibility is shifted over to the robot to learn a representation of this movement. Based on the results from Section 2.3, the selected method for learning is DMP. The decision was mostly based on the ability for the DMP to reproduce smooth trajectories, which is crucial when playing the ball-in-a-cup game. Furthermore, has the same skill also previously been modelled as a DMP [50] . Moreover, a DMP was able to both learn and reproduce the movement fast, which is important for online learning.

2.5 Alternative Approaches to Learning from Demonstration

The learning from demonstration framework is not the only approach for a robot to acquire new skills. Another view of learning is inspired by humans and animals, hence the name biologically-inspired learning. This method takes advantages on the modern science of artificial neural network which imitates *Mirror Neuron System* (MNS). A thorough introduction to MNS is presented in [68]; however, a more recent version of the same article is presented in [69].

In contrast to LfD, direct programming and MNS, a skill can also be learned by reinforcement learning. The benefits of this approach is that learning is fully autonomous and the learned skill can be adapted to varying dynamics. On the other hand, in a high dimensional space, where both actions and states are continuous, finding an optimal policy can take infinite amount of time due to the "curse of dimensionality" [9]. However, in comparison to LfD, where a skill which cannot be demonstrated is impossible to learn, RL can still learn this skill provided that the search space is small. RL is explained in more details in Chapter 4.

3 Dynamic Movement Primitives

Movements in the animal kingdom can be seen as combining motor primitives, which has been witnessed in despinalized frogs [32] where the leg ended up moving in a point-attracting movement when the spinal cord was stimulated. The same movement primitives were also found in rats [96], cats [55] as well as for vertebrates and invertebrates [29]. All these studies point toward complex movements being built from a library of movement primitives.

This idea has later been applied to robotics where primitives is the linkage between the perceptual system and the motor system and is the primary factor for complex imitation [57]. For modelling these motor primitives a framework known as dynamic movement primitive was presented in [83]. This framework enable robots to learn complex movements on trajectory level as a dynamical system. The replicable movements can either be discrete, for example in reaching tasks [38], or rhythmic, for example in biped locomotion [63]. Another definition of the discrete and rhythmic movements are point attractor and limit cycle movements [39]. The following sections in this chapter will explain the whole DMP framework starting from modelling a DMP.

3.1 Modelling the DMP

The theory behind DMP is well established in [38] where the heart of the model is a point-attractor system modulated with a nonlinear function to enable generation of complex movements. One point-attractor system is the spring-damper system

$$\tau\ddot{y} = \alpha_z(\beta_z(g - y) - \dot{y}) \tag{1}$$

where τ is the duration of the movement, α_z and β_z are constants chosen in order to ensure that the system becomes critically damped, g is the goal state and $y, \dot{y}, \ddot{y}$ is respectively the desired position, velocity and acceleration. Equation (1) can also be written in a first-order notation as

$$\begin{aligned} \tau\dot{z} &= \alpha_z(\beta_z(g - y) - z) \\ \tau\dot{y} &= z. \end{aligned} \tag{2}$$

The intuition behind using a spring-damper system is that the spring, after excitation, always converges to the goal state in a finite amount of time. Hence, the end state will always converge to the goal state (See Figure 11). However, such generated trajectories are trivial as they always form similar shapes and thus more complex patterns are needed for controlling complex movements. This is realized by modulating Equation (2) with a nonlinear function f, resulting in [38]

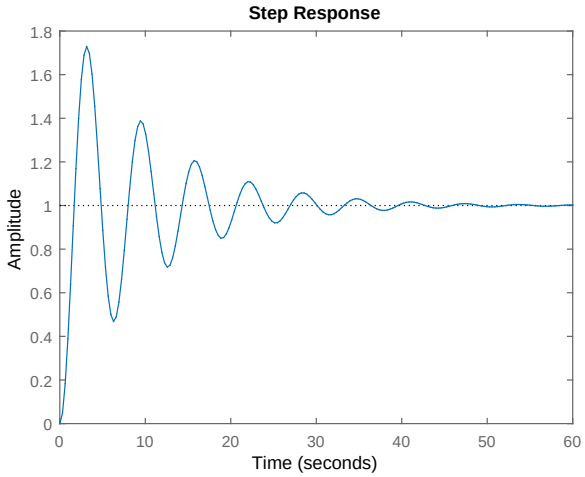


Figure 11: This figure shows the trajectory generated form an excited spring-damper system. The trajectory oscillates around the set-point, which in this case is 1, in a damped fashion.

$$\begin{aligned} \tau \dot{z} &= \alpha_z(\beta_z(g - y) - z) + f, \\ \tau \dot{y} &= z. \end{aligned} \tag{3}$$

Equation (3) is called for the *transformation system* and the nonlinear function f for the *forcing function*.

The forcing function is defined as a linear combination of basis functions

$$f(t) = \frac{\sum_{i=1}^{N} \Psi_i(t) w_i}{\sum_{i=1}^{N} \Psi_i(t)}, \tag{4}$$

where Ψ_i denotes a basis function and w_i represents the corresponding adjustable weight. It is interesting that the complex nonlinear function is represented as a linear combination of simple basis function, which is of common praxis in *Machine Learning* (ML) [13]. The forcing function is also responsible to either generate discrete or repeating movements. The former movement is realized by having a phasic forcing function and the latter by having a periodic one. The following derivations are based on discrete movements, whereas the rhythmic movements will be introduced later.

As can be seen in Equation (3) the forcing function, f, is time dependent. As a consequence the basis functions becomes broader as time moves forward (see Figure 13a), resulting in shorter activation time for early kernels and longer for latter ones. Therefore, to equispace the kernels equally in time a variable x, called *phase variable*, is introduces as a replacement for time. The variable is defined as a first order differential equation

$$\tau \dot{x}(t) = -\alpha_x x(t), \tag{5}$$

where α_x is a fine-tuned constant which enforces a stable system, and τ is the same time constant as previously. This equation is called the *canonical system* and has the solution

$$x = x_0 \exp(-\alpha_x \frac{t}{\tau}). \tag{6}$$

Equation (6) will, whatever positive initial value x_0 is (usually $x_0 = 1$), converge exponentially to zero as the execution time t converges to infinity (see Figure 12). By replacing time in Equation (4) with the phase variable in Equation (6) the nonlinear forcing function can be rewritten in a time independent form

$$f(x) = \frac{\sum_{i=1}^{N} \Psi_i(x) w_i}{\sum_{i=1}^{N} \Psi_i(x)} x(g - y_0), \tag{7}$$

where g is the goal, y_0 is the initial state and $\psi_i(x)$ are defined as Gaussian kernels

$$\Psi_i(x) = \exp\left(-\frac{1}{2\sigma_i^2}(x - c_i)^2\right), \tag{8}$$

where σ_i is the standard deviation (width) and c_i is the mean (centre) of a kernel.

It is interesting to note the differences between the time dependant and independent forcing functions in Equations (4) and (7). The latter equation is, in addition to being time independent, also scaled with the phase variable x and the factor $g - y_0$. The reason behind scaling Equation (7) with the phase variable x is that this ensures global stability of the system because as x converges to zero so does f and with it its influence, resulting in the same unique point attractor $(z, y) = (0, g)$ as the regular spring-damper in Equation (2). The extra scaling factor $g - y_0$, on the other hand, provides a possibility to scale the movement according to changes in its amplitude [38]. Additionally a time independent forcing function enables to either speed up or slow down the execution of the movement, something which is otherwise impossible. All in all, the whole system converges to the globally stable point-attractor $(z, y, x) = (0, g, 0)$ as time converges to infinity.

Up till now the focus has been on generating discrete movements. However, rhythmic movements are also of great interest as many human motions such as biped locomotion is of this kind [63]. To adapt the point-attractor system to a periodic motion, periodicity is either inserted in the basis function or in the canonical system. In [38] the emphasis lies on the latter proposal where a simple phase oscillator

$$\tau\dot{\phi} = 1 \tag{9}$$

is used as the canonical system. The constant ϕ is the phase angle of the oscillation defined within the region $[0, 2\pi]$ and r is the amplitude of the oscillation. By inserting the phase angle ϕ and amplitude r of the movement into the forcing and basis functions they can be rewritten into the following form

$$\begin{aligned} f(\phi, r) &= \frac{\sum_{i=1}^{N} \Psi_i(\phi) w_i}{\sum_{i=1}^{N} \Psi_i(\phi)} r, \\ \Psi_i(\phi) &= \exp\left(-h_i\left(\cos(\phi - c_i) - 1\right)\right), \end{aligned} \tag{10}$$

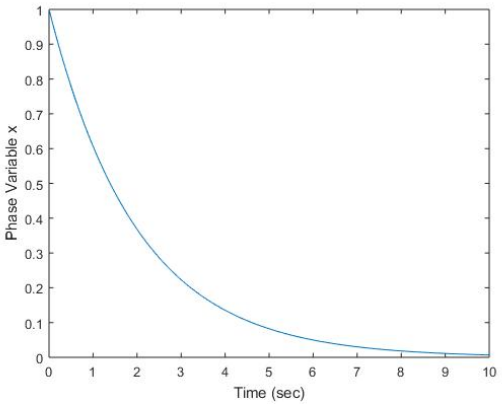
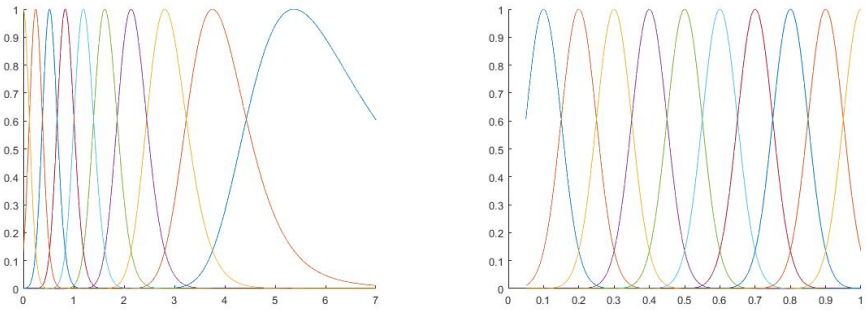


Figure 12: In the discrete system the canonical system converges to zero as time converges to infinity. Here $\tau = 1$, $\alpha = 2$ and $x_0 = 1$.



(a) The kernels are not equispaced in time and hence they get broader and broader as time converges to infinity. This causes problems as the first kernels are not activated for as long as the kernels to the end.

(b) The phase variable allows every kernel to be eqispaced and hence all of them are active for an equal amount of time.

Figure 13: In the figure to the left the kernels are time dependent whereas in the right figure they are phase dependent.

where Φ_i is a von Mises basis function [41]. This formulation provides the possibility to change the amplitude and period in real time by changing r and ϕ respectively. The goal g in the transformation system defines the set or anchor point of the oscillatory trajectory and can be adapted to the desired oscillation baseline.

3.2 Learning the DMP

In the previous section, the attractor landscape for both discrete and rhythmic movements were established. It is worth noting that the forcing function includes both unknown weights and basis functions. The role of the basis functions is to enable a smooth reproduction of the movement and the number of such functions is

defined in advanced by trial and error. The role of the weights, on the other hand, is to define the spatio temporal path by weighing the basis function differently. The values of the weights, in contrast to the basis functions, have to be learned and cannot be set in advance.

For learning an initial representation of the parameters w_i supervised learning is used, which is possible as these parameters are linear [38]. In supervised learning some initial input and output data is provided [13]. In the case of learning a DMP the interesting parameters are the positions, velocities and accelerations $(y_{demo}(t), \dot{y}_{demo}(t), \ddot{y}_{demo}(t))$, sampled at different time instances $t \in [1, \ldots, P]$. For discrete movements the goal g is the position at time P, $g = y_{demo}(t = P)$, and the start position is the position at time 0, $y_0 = y_{demo}(t = 0)$. In contrast to the discrete movement, the rhythmic one defines the goal, duration and amplitude differently. The goal g is the midpoint of the movement $g = 0.5(min_{t\in[1,\ldots,P]}(y_{demo}(t))+ max_{t\in[1,\ldots,P]}(y_{demo}(t)))$, the duration τ is the period of the motion divided by 2π and the amplitude r is set to the value 1.0.

The duration of the movement τ can be extracted from the demonstration. However, to sense a movement onset or offset, a threshold value should be set, for instance at 2% of the maximum velocity of the trajectory [38]. Then, at the end, the total duration τ should be multiplied by 1.05 to compensate for the thresholding.

When all the required parameters are set, the weights in Equation (7) can be calculated. This is done by first rearranging Equation (2) to get an explicit form for the forcing function f

$$f = \tau\dot{z} - \alpha_z(\beta_z(g - y) - z). \tag{11}$$

Knowing the desired values from the demonstration and inserting them in the previous equation yields

$$f_{target} = \tau^2\ddot{y}_{demo} - \alpha_z(\beta_z(g - y_{demo}) - \tau\dot{y}_{demo}). \tag{12}$$

Now the issue is how to set the parameters in f to be very close to f_{target}. This is a function approximation problem where existing methods like Gaussian mixture models [47] and LWR [81] can be applied, both discussed in Chapter 2. The one chosen is LWR as it is fast and enable kernels to learn their values individually.

The goal in LWR is to minimize the weighted quadratic error

$$J_i = \sum_{t=1}^{P} \Psi_i(t)(f_{target}(t) - w_i\varepsilon(t))^2, \tag{13}$$

where $\varepsilon(t) = x(t)(g - y_0)$ for discrete movements and $\varepsilon(t) = r$ for rhythmic ones. This regression model finds corresponding weights w_i to the individual kernels Ψ_i such that the minimization is fulfilled. The solution to the regression problem is

$$w_i = \frac{\mathbf{S^T\Gamma_i f_{target}}}{\mathbf{S^T\Gamma_i S}}, \tag{14}$$

where

$$\mathbf{S} = \begin{pmatrix} \varepsilon(1) \\ \varepsilon(2) \\ \vdots \\ \varepsilon(P) \end{pmatrix} \quad \mathbf{\Gamma}_i = \begin{pmatrix} \Psi_i(1) & 0 & \dots & 0 \\ 0 & \Psi_i(2) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \Psi_i(P) \end{pmatrix} \quad \mathbf{f}_{target} = \begin{pmatrix} f_{target}(1) \\ f_{target}(2) \\ \vdots \\ f_{target}(P) \end{pmatrix}.$$

The regression performed in Equation (14) is called batch regression where the idea is to perform the regression once all the data is gathered. Another approach is LWPR where the regression is updated incrementally as new data becomes available. Both LWR and LWPR were discussed in Chapter 2 and the main difference between the two approaches was the computational complexity, which for LWR was polynomially $\mathcal{O}(n^2)$ and for LWPR linear $\mathcal{O}(n)$. Therefore, if one expects the data set n to be large, LWPR is the main choice. Although, in our experiment, this is not the case and LWR will suffice.

As the weights, which can have both positive and negative values (see Figure 14), have been learned they are used by the transformation system to generate a representation of the demonstrated trajectory (see Figure 19). Unfortunately, the initially learned representation of the movement is not always capable of reproducing a taught skill successfully. For instance in [50] the demonstrated skill is how to play the ball-in-a-cup game. Even tough the ball went into the cup in the demonstration part, the initially learned DMP could not generate such a movement where the ball went into the cup. Thus, for successfully learning the skill, the shape parameters of the DMP needed to be fine-tuned with subsequent reinforcement learning. How this is done will be explained in Chapter 4. However, before proceeding to that the learned movement has to be scaled to multiple DoF, which will be explained next.

3.3 From One To Multiple Degrees of Freedom

Until now the focus has been on learning a DMP for a single DoF. However, a real robotic manipulator usually consists of several degrees of freedom, which is also the case for the robot used in this thesis (see Figure 19). Hence, the basic idea of the transformation and canonical system needs to be extended to cope with multiple DoF.

There are basically three different alternatives for extending the DMP framework to multiple DoF. The first option is for all DoF to have their own transformation and canonical system. The second option is for all DoF to be coupled together with a special coupling term. While the third option is for all DoF to have their own transformation system but share an internal canonical system as is presented in Figure 16.

The disadvantages with the first alternative is synchronisation, as there are no part where information is shared between any degree of freedom they can, in the long run, diverge numerically far from each other. On the other hand, information shared can also be a potential problem. This occurs in the second alternative where the system grows too complex, and tuning the coupling terms is difficult. With this in mind, the third option seeks to solve these problems by combining the simplicity

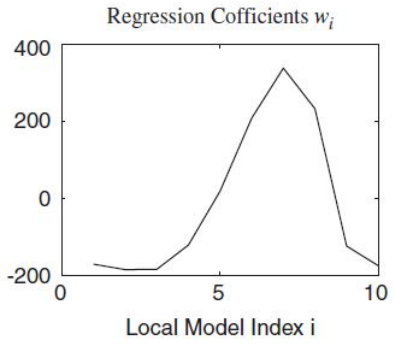


Figure 14: This figure shows an example of the learned weights from one demonstration. (Source [82]).

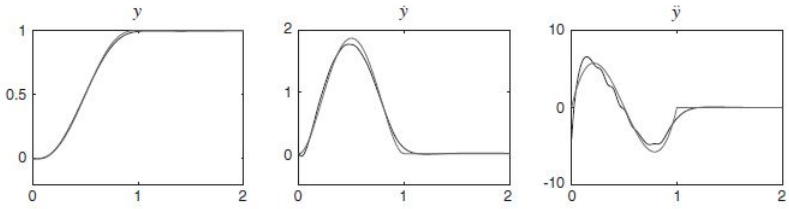


Figure 15: This figure show the reproduced trajectory by a DMP with the weights presented in Figure 14. The superimposed smooth line is the ideal movement which minimizes the integral of squared jerk along the trajectory. Noteworthy is that the learned trajectory is not far from the ideal one. (Source [82]).

of the first and the complexity of the second into a mixture of both. This solution will force the canonical system to act as a global clock to couple all DoF together. However, there are no definite rules on how the coupling has to be done. For example in a humanoid robot with several arms, each arm might have its own canonical system which then has to be coupled together to provide synchronising movements of both arms [38].

In conclusion, the extended DMP formulation allows for learning complex movements in multiple DoF by having a simple attractor landscape perturbed by a nonlinear forcing function which decreases monotonically as to preserve the global stability of the whole system.

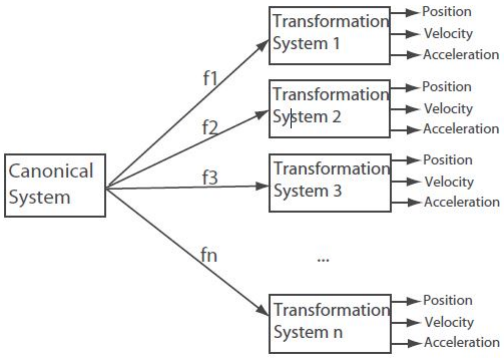


Figure 16: An overview for scaling the DMP to multiple DoF. Each DMP has its own transformation system while they share a common canonical system acting as a global clock to sync the different transformation systems together. (Source [38]).

4 Reinforcement Learning

In RL the goal is to learn an optimal policy, resembling what actions to choose as to maximize the gained rewards [90]. Hence, in contrast to supervised learning, no prior information is given about the task, and learning it is based on an iterative approach of choosing actions and observing the rewards. Some classical approaches to solve RL problems is Q-learning and SARSA, both mentioned in [90]. Reinforcement learning is a powerful learning method when the states and actions are few and discrete; however, with continuous states and actions, common in robotics, the search space grows too large and is bounded by the "curse of dimensionality"[9]. Hence, RL alone is not applicable for learning a robot how to perform tasks.

Up to this point, both RL and DMP have been proven impossible as a learning framework for complex tasks. However, when combining the two approaches their individual problems disappear, and learning complex tasks is made possible. The idea behind combining RL and DMP is to first encode the demonstrated skill as a DMP and then subsequently fine-tune the shape parameters of the DMP by an RL algorithm. As the output from a DMP can be a suboptimal policy, the goal with RL is to search for an optimal policy, hence the name *policy search*.

Skills can either be defined as episodic or nonepisodic with a finite or infinite time horizon. As many skills are episodic with a finite time horizon, for example reaching tasks, these set of skills will be the focus in this chapter, while the rest are discarded.

In this chapter the basics of reinforcement learning is introduced. This concept will then be extended into the policy search space where two approaches are presented: one based on policy gradient [91, 100], and another based on expectation maximization [27, 72]. All in all, two algorithms per approach will be presented. In the policy gradient case the basic *'Vanilla' Policy Gradient* (VPG) algorithm [100] and *episodic Natural Actor Critic* (eNAC) algorithm [71, 73] are presented. In the EM case the basic *episodic Reward Weighted Regression* (eRWR) algorithm [72] and the improved *Policy Learning by Weighted Exploration with the Returns* (PoWER) algorithm [51] are presented.

The last mentioned algorithm, PoWER, has previously been applied as a policy search algorithm to the ball-in-a-cup game [50], and can currently be seen as one of the best policy search algorithms. However, there also exist a competing one, based on *Stochastic Optimal Control* (SOC), called *Policy Improvement with Path Integral* (PI^2) [95]. Thus, PI^2 will also be discussed. Based on the individual performance of both PoWER and PI^2 one will be chosen and implemented for improving a policy in the ball-in-a-cup game.

4.1 Problem Statement and Notation

RL was introduced in [90] where the idea is to learn by interaction with the environment. Learning through interactions is common among humans as we do this throughout our lives. For example, an infant learns how to walk through a repetitive trial and error interaction with the environment. Thus, reinforcement learning, in its essence, is something everyone has experienced.

Before going into detail about reinforcement learning some basic notations has to be defined. First we have to assume there exist an agent in an environment which at time t chooses an action a_t. This action transfers the agent from the current state s_t to the next state s_{t+1} while receiving a reward r_t. The reward can either be positive or negative, depending if a certain behaviour should be penalized or encouraged. The actions are chosen as one amongst several possibilities, resulting in a stochastic policy $\pi(a_t|s_t, t)$, where the current action depends on the current state and time. By having a stochastic policy different actions are tested all the time, a concept known as exploration.

As stipulated before, this thesis concentrate on episodic finite time horizon cases, where the ultimate goal is to improve an initial parametrized policy for a complex motor task. The parametrized policy is assumed to be stochastic with parameters $\theta \in \mathbb{R}^n$. An episodic environment is defined by having some absorbing states, also called end states, from where an agent cannot escape. If such a state is visited, the agent is transferred back to the start state, while the previously visited states, actions chosen, and accumulated rewards are stored. The whole process from start to end state is called a 'Monte Carlo rollout' or simply a 'rollout', and is defined as $\Omega = [s_{1:T+1}, a_{1:T}]$, where $s_{1:T+1} = [s_1, s_2, \ldots, s_{T+1}]$ are the visited states and $a_{1:T} = [a_1, a_2, \ldots, a_T]$ are the chosen actions.

The goal with RL is to find a policy π parametrized by the policy parameters θ such that the expected returns defined as $J(\theta)$ are optimized. This results in the following Equation

$$J(\theta) = \int_T p_\theta(\Omega) R(\Omega) d\Omega, \tag{15}$$

where $P(\Omega)$ is the probability of rollout Ω, and $R(\Omega)$ is the corresponding accumulated reward form that specific rollout. The integration is taken over all possible rollouts.

Equation (15) can be simplified if the Markov property holds, and if the rewards are additive and cumulative. The Markov property refers to future states being independent of past states if the current state is known [16]. These assumptions simplifies $p_\theta(\Omega)$ and $R(\Omega)$ to

$$\begin{aligned} p_\theta(\Omega) &= p(s_1) \prod_{t=1}^{T} p(s_{t+1}|s_t, a_t) \pi(a_t|s_t, t), \\ R(\Omega) &= T^{-1} \sum_{t=1}^{T} r(s_t, a_t, s_{t+1}, t), \end{aligned} \tag{16}$$

where $p(s_1)$ is the distribution of the initial state, $p(s_{t+1}|s_t, a_t)$ is the next state distribution at time $t+1$ given the current state and action at time t, and $r(s_t, a_t, s_{t+1}, t)$ is the immediate reward received when transferring from state s_t to state s_{t+1} by choosing action a_t.

4.2 Episodic Policy Learning

The focus in this section is on RL for optimizing policies which are episodic and have a finite time horizon. Before deriving the state-of-the-art episodic RL algorithm

PoWER several steps defined in [51] have to be discussed. First a lower bound on the expected reward has to be set for reassuring improvements of the policy update steps. The derivation of the lower bound follows [27] which, unfortunately, only considers immediate rewards. Thus, in the second step the lower bound of the update step based on the immediate reward is extended to also work for episodic RL. Based on this result a general rule for updating policies can be derived. Finally, based on this update rule, the policy gradient theorem [92] and the PoWER algorithm can be derived.

4.2.1 Defining the Lower Bound of the Policy

In contrast to classic RL, the EM algorithm aims to optimize the lower bound of the cost function [58]. Intuitively, this makes sense as long as the lower bound can be used as a sampling policy, then by maximizing it the resulting policy will also be improved.

To prove the previous statement, we extend the work in [27] to the episodic scenario. First, assume an existing policy with parameters θ from which rollouts Ω can be collected and rewards $R(\Omega)$ received. The goal is then to match the known policy with parameters θ weighted with the rewards $R(\Omega)$ to a new policy with parameters θ'. The matching is realized by minimizing the *Kullback-Leibler* (KL) divergence $D(p_\theta(\Omega)R(\Omega) \parallel p_{\theta'}(\Omega))$ between the current reward weighted distribution and the new one. The KL divergence is defined in machine learning as a distance measure between two probability distributions [8][99]. It is, however, not strictly a distance measure because it is non-symmetric, *i.e.* $D(p_\theta(\Omega)R(\Omega) \parallel p_{\theta'}(\Omega)) \neq D(p_{\theta'}(\Omega) \parallel p_\theta(\Omega)R(\Omega))$, but this does not impose any major problems and can therefore be used. A lower bound on the policy is then obtained by maximizing the KL divergence by utilizing Jensen's equality and the concavity of the logarithm [27, 72]. This results in

$$\begin{aligned} \log J(\theta') &= \log \int_T p_{\theta'}(\Omega)R(\Omega)d\Omega = \log \int_T \frac{p_\theta(\Omega)}{p_\theta(\Omega)} p_{\theta'}(\Omega)R(\Omega)d\Omega \\ &\geq \int_T p_\theta(\Omega)R(\Omega) \log \frac{p_{\theta'}(\Omega)}{p_\theta(\Omega)} d\Omega + \text{const} \\ &\propto -D(p_\theta(\Omega)R(\Omega) \parallel p_{\theta'}(\Omega)) = L_\theta(\theta'), \end{aligned} \tag{17}$$

where the KL divergence is denoted as

$$D(p(\Omega) \parallel q(\Omega)) = \int p(\Omega) \log \frac{p(\Omega)}{q(\Omega)} d\Omega.$$

The constant is Equation (17) is essential for reassuring a tight bound.

The lower bound $L_\theta(\theta')$ can be optimized in two ways, either by policy gradient search or with the EM algorithm. For clarification, both methods are presented, where the latter one results in the previously mentioned PoWER algorithm.

4.2.2 Policy Gradient Search

In the previous section, the idea of maximizing the lower bound $L_\theta(\theta')$ to improve the policy was introduced. Furthermore, two methods for realizing this was mentioned,

where one was to use policy gradient search. In this section, two policy gradient optimization algorithms, the vanilla policy gradient [100] and episodic Natural Actor Critic [7, 71], are presented.

For maximizing a function the basic idea is to calculate its gradient. This is applied to form the VPG algorithm (see Algorithm 1) where the function to differentiate is the lower bound $L_\theta(\theta')$, resulting in

$$\partial_{\theta'} L_\theta(\theta') = \int_T p_\theta(\Omega) R(\Omega) \partial_{\theta'} \log p_{\theta'}(\Omega) d\Omega = E\left\{ \left(\sum_{t=1}^{T} \partial_{\theta'} \log \pi(a_t|s_t, t) \right) R(\Omega) \right\}, \tag{18}$$

where

$$\partial_{\theta'} \log p_{\theta'}(\Omega) = \sum_{t=1}^{T} \partial_{\theta'} \log \pi(a_t|s_t, t)$$

is the log-derivative of the state distributions. It is worth noting that the log-derivative is only dependent on the policy π. Thus, estimating the gradient can be done from performing rollouts by replacing the expectation in the previous equation by a sum. If θ is close to θ' the partial derivative of the lower bound can be rewritten as

$$\lim_{\theta' \to \theta} \partial_{\theta'} L_\theta(\theta') = \partial_\theta J(\theta).$$

This estimator is the same one which is used in the widely known episodic REINFORCE algorithm [100].

The problem with the policy gradient algorithm is the non-existing restriction in the amount the new policy can differ from the old one. This can, in the worst case, lead to large infeasible changes. Therefore, large changes should be avoided, something which is fulfilled in the eNAC algorithm (see Algorithm 2). In this algorithm large differences are penalized. The penalization is realized by adding an additional constraint to the KL divergence and approximating it as a second-order expansion as

$$D(p_\theta(\Omega) \parallel p_{\theta'}(\Omega)) \approx 0.5(\theta' - \theta)^T F(\theta)(\theta' - \theta) = \delta,$$

where $F(\theta)$ is the Fisher information matrix [8, 83]. Moreover, the estimator in the eNAC algorithm utilizes the fact that future actions are uncorrelated with previous rewards. Thus, when combining Equations (16) and (18), the expectation of the product between r_t and $\partial_{\theta'} \log \pi(a_{t+\delta t}|s_{t+\delta t}, t + \delta t)$ vanishes for all positive δ [71]. This simplification reduces the estimator in Equation (18) to

$$\partial_{\theta'} L_\theta(\theta') = E\left\{ \sum_{t=1}^{T} \partial_{\theta'} \log \pi(a_t|s_t, t) Q^\pi(s, a, t) \right\}, \tag{19}$$

where Q is the state-action value function defined as

$$Q^\pi(s, a, t) = E\left\{ \sum_{\widetilde{t}=t}^{T} r(s_{\widetilde{t}}, a_{\widetilde{t}}, s_{\widetilde{t}+1}, \widetilde{t}) | s_t = s, a_t = a \right\}.$$

If the dependence on time is removed from Equation (19), then as time goes to infinity $\theta' \to \theta$ which is identical to the policy gradient theorem [91].

4.2.3 Policy Search by Expectation-Maximization

In both the VPG and eNAC algorithms there is an open tunable parameter α called the learning-rate parameter. For performance issues this parameter has to be fine-tuned, which is both tedious and hard to accomplish. To overcome this problem, policy search with EM is employed. This method utilizes the EM algorithm, which in supervised learning avoids the learning parameter while providing a faster convergence rate [58].

The idea behind the EM algorithm is to iteratively find the maximum likelihood of a latent variable. In this case the latent variable is θ, and its next value corresponds to maximizing the lower bound defined in Equation (17) as

$$\theta_{t+1} = \arg\max_{\theta'} L_\theta(\theta').$$

Solving the maximization, and subsequently obtaining the next θ value, is fulfilled by setting the gradient of the lower bound w.r.t. θ' to zero, and then solving for θ'. This can be done analytically, provided that the stochastic policy $\pi(a_t|s_t, t)$ is an exponential function, by either setting Equations (18) or (19) to zero, and solve for θ'. Choosing the latter equation results in

$$E\left\{\sum_{t=1}^{T} \partial_{\theta'} \log \pi(a_t|s_t, t) Q^\pi(s, a, t)\right\} = 0. \tag{20}$$

Note that different stochastic policies results in different solutions and subsequently different learning algorithms.

Generating the actual stochastic policy has so far not been discussed. First, we have to define a deterministic mean policy a as $\bar{a} = \theta^T \phi(s, t)$, where ϕ are basis functions and θ the parameters. To transform this into a stochastic policy, noise or exploration defined as $\epsilon(s, t)$ is added to the mean policy, which enables model-free reinforcement learning. The resulting stochastic policy $\pi(a_t|s_t, t)$ can then be written as

$$a = \theta^T \phi(s, t) + \epsilon(\phi(s, t)). \tag{21}$$

The exploration term ϵ can be chosen freely, but in previous studies [71, 72] it has been defined as a state-independent Gaussian distribution $\epsilon(\phi(s, t)) \sim \mathcal{N}(\epsilon|0, \Sigma)$. With the Gaussian exploration, Equation (20) is solved for θ' by reward-weighted regression, as in Algorithm 3. Several applications are based on this solution including T-Ball batting [71] and Peg-In-Hole [34].

This approach seems to work well in many applications. However, for the ball-in-a-cup game it is not applicable because of the problems with unconstrained exploration at every time step. The drawbacks are: 1) a large variance in the policy parameter update which grows with the amount of time steps, 2) the system works as a low pass filter, thus the effect of high frequency perturbation of actions averages out, and in the end the effect vanishes, and 3) the system executing the trajectory can be damaged. The third problem occurs as sudden instantaneous changes in the policy might force instantanious movements of the robot which is not possible as the motors

and links have to overcome inertia induced by previous actions, and the controller needs time to react.

Therefore to enable policy search for the ball-in-a-cup game, an extension to the current framework is needed. This extension results in the PoWER algorithm [51], which will be presented in the next section.

Algorithm 1 'Vanilla' Policy Gradients (VPG). (Source: [51]).

Input: initial policy parameters $\theta_{\mathbf{0}}$

repeat

Sample: Perform $\mathbf{h} = \{\mathbf{1}, \ldots, \mathbf{H}\}$ rollouts using $\mathbf{a} = \theta^{\mathbf{T}}\phi(\mathbf{s}, \mathbf{t}) + \epsilon_{\mathbf{t}}$ with $[\epsilon_t^n] \sim \mathcal{N}(0, (\sigma^{h,n})^2)$ as a stochastic policy and collect all $(\mathbf{t}, \mathbf{s_t^h}, \mathbf{a_t^h}, \mathbf{s_{t+1}^h}, \epsilon_{\mathbf{t}}^{\mathbf{h}}, \mathbf{r_{t+1}^h})$ for $\mathbf{t} = \{\mathbf{1}, \mathbf{2}, \ldots, \mathbf{T+1}\}$.

Compute: Return $\mathbf{R^h} = \sum_{\mathbf{t=1}}^{\mathbf{T+1}} \mathbf{r_t^h}$, eligibility

$$\psi^{\mathbf{h,n}} = \frac{\partial \log \mathbf{p(\Omega^h)}}{\partial \theta^{\mathbf{n}}} = \sum_{\mathbf{t=1}}^{\mathbf{T}} \frac{\partial \log \pi(\mathbf{a_t^h}|\mathbf{s_t^h}, \mathbf{t})}{\partial \theta^{\mathbf{n}}} = \sum_{\mathbf{t=1}}^{\mathbf{T}} \frac{\epsilon_{\mathbf{t}}^{\mathbf{h,n}}}{(\sigma_{\mathbf{h}}^{\mathbf{n}})^{\mathbf{2}}} \phi^{\mathbf{n}}(\mathbf{s_t^{h,n}}, \mathbf{t})$$

and baseline

$$b^n = \frac{\sum_{h=1}^{H} (\psi^{h,n})^2 R^h}{\sum_{h=1}^{H} (\psi^{h,n})^2}$$

for each parameter $\mathbf{n} = \{\mathbf{1}, \ldots, \mathbf{N}\}$ from rollouts.

Compute Gradient:

$$\mathbf{g_{VP}^n} = E\left\{\frac{\partial \log p(\Omega^h)}{\partial \theta^n}(R(\Omega^h) - b^n)\right\} = \frac{1}{H}\sum_{h=1}^{H} \psi^{h,n}(R^h - b^n).$$

Update policy using

$$\theta_{\mathbf{k+1}} = \theta_{\mathbf{k}} + \alpha \mathbf{g_{VP}}$$

until Covergence $\theta_{\mathbf{k+1}} \approx \theta_{\mathbf{k}}$

Algorithm 2 episodic Natural Actor Critic (eNAC). (Source: [51]).

Input: initial policy parameters $\theta_{\mathbf{0}}$

repeat

Sample: Perform $\mathbf{h} = \{\mathbf{1}, \ldots, \mathbf{H}\}$ rollouts using $\mathbf{a} = \theta^{\mathbf{T}}\phi(\mathbf{s}, \mathbf{t}) + \epsilon_{\mathbf{t}}$ with $[\epsilon_t^n] \sim \mathcal{N}(0, (\sigma^{h,n})^2)$ as a stochastic policy and collect all $(\mathbf{t}, \mathbf{s_t^h}, \mathbf{a_t^h}, \mathbf{s_{t+1}^h}, \epsilon_{\mathbf{t}}^{\mathbf{h}}, \mathbf{r_{t+1}^h})$ for $\mathbf{t} = \{\mathbf{1}, \mathbf{2}, \ldots, \mathbf{T+1}\}$.

Compute: Return $\mathbf{R^h} = \sum_{\mathbf{t=1}}^{\mathbf{T+1}} \mathbf{r_t^h}$, eligibility $\psi^{\mathbf{h,n}} = \sum_{t=1}^{T} (\sigma_h^n)^{-2} \epsilon_t^{h,n} \phi^n(s_t^{h,n}, t)$ for each parameter $\mathbf{n} = \{\mathbf{1}, \ldots, \mathbf{N}\}$ from rollouts.

Compute Gradient:

$$[\mathbf{g_{eNAC}^T}, \mathbf{R_{ref}}]^{\mathbf{T}} = (\mathbf{\Psi^T \Psi})^{-\mathbf{1}} \mathbf{\Psi^T R}$$

with $\mathbf{R} = [R^1, \ldots, R^H]^T$ and $\mathbf{\Psi} = \begin{bmatrix} \psi^1, \ldots, \psi^H \\ 1, \ldots, 1 \end{bmatrix}^T$ where $\psi^{\mathbf{h}} = [\psi^{h,1}, \ldots, \psi^{h,N}]^T$.

Update policy using

$$\theta_{\mathbf{k+1}} = \theta_{\mathbf{k}} + \alpha \mathbf{g_{eNAC}}$$

until Covergence $\theta_{\mathbf{k+1}} \approx \theta_{\mathbf{k}}$

Algorithm 3 episodic Reward Weighted Regression (eRWR). (Source: [51]).

Input: initial policy parameters $\theta_{\mathbf{0}}$

repeat

Sample: Perform $\mathbf{h} = \{\mathbf{1}, \ldots, \mathbf{H}\}$ rollouts using $\mathbf{a} = \theta^{\mathbf{T}}\phi(\mathbf{s}, \mathbf{t}) + \epsilon_{\mathbf{t}}$ with $[\epsilon_t^n] \sim \mathcal{N}(0, (\sigma^{h,n})^2)$ as a stochastic policy and collect all $(\mathbf{t}, \mathbf{s_t^h}, \mathbf{a_t^h}, \mathbf{s_{t+1}^h}, \epsilon_{\mathbf{t}}^{\mathbf{h}}, \mathbf{r_{t+1}^h})$ for $\mathbf{t} = \{\mathbf{1}, \mathbf{2}, \ldots, \mathbf{T+1}\}$.

Compute: State-action value function $\mathbf{Q_t^{\pi,h}} = \sum_{\tilde{\mathbf{t}}=\mathbf{t}}^{\mathbf{T}} \mathbf{r_{\tilde{t}}^h}$

Update policy using

$$\theta_{\mathbf{k+1}}^{\mathbf{n}} = \left((\mathbf{\Phi^n})^{\mathbf{T}}\mathbf{Q}^{\pi}\mathbf{\Psi^n}\right)^{-\mathbf{1}} (\mathbf{\Psi^n})^{\mathbf{T}}\mathbf{Q}^{\pi}\mathbf{A^n}$$

with basis functions

$$\mathbf{\Psi^n} = [\phi_1^{1,n}, \ldots, \phi_T^{1,n}, \phi_1^{2,n}, \ldots, \phi_1^{H,n}, \ldots, \phi_T^{H,n}]^T,$$

where $\phi_{\mathbf{t}}^{\mathbf{h,n}}$ is the value of the basis function of rollout $\mathbf{h}$ and parameter $\mathbf{n}$ at time $\mathbf{t}$, actions

$$\mathbf{A^n} = [a_1^{1,n}, \ldots, a_T^{1,n}, a_1^{2,n}, \ldots, a_1^{H,n}, \ldots, a_T^{H,n}]^T,$$

and return

$$\mathbf{Q}^{\pi} = \text{diag}(Q_1^{\pi,1}, \ldots, Q_T^{\pi,1}, Q_1^{\pi,2}, \ldots, Q_1^{\pi,H}, \ldots, Q_T^{\pi,H})$$

until Covergence $\theta_{\mathbf{k+1}} \approx \theta_{\mathbf{k}}$

4.3 Policy Learning by Weighted Exploration with the Returns

The previous section introduced the policy search through EM, and resulted in the eRWR algorithm, an algorithm with which the policy can be iteratively improved. However, the drawbacks made it impossible to apply it as a policy search algorithm for the ball-in-a-cup game. Hence, another algorithm which overcome these problems is needed. This algorithm is the PoWER algorithm, and will be introduced next.

For solving the problems which the previous eRWR algorithm had, it is necessary to redefine the exploration from a state-independent into a structured, state-dependent one, as [78]. This results in an exploration defined as

$$\epsilon(\phi(s,t)) = \epsilon_t^T \phi(s,t),$$

where $\epsilon_t \sim \mathcal{N}(0, \hat{\Sigma})$. The $\hat{\Sigma}$ is called a meta-parameter, and can be optimized in a similar manner as the θ parameter. By changing the exploration, the basic stochastic policy introduced in Equation (21) becomes

$$a \sim \pi(a_t|s_t,t) = \mathcal{N}(a|\theta^T\phi(s,t), \phi(s,t)^T\hat{\Sigma}\phi(s,t)).$$

This policy is then inserted into Equation (20), which generates the optimal θ at that specific time step. The policy parameter is then updated as

$$\theta' = \theta + E\left\{\sum_{t=1}^{T} \mathbf{W}(s,t)Q^{\pi}(s,a,t)\right\}^{-1} E\left\{\sum_{t=1}^{T} \mathbf{W}(s,t)\epsilon_t Q^{\pi}(s,a,t)\right\}, \qquad (22)$$

where $\mathbf{W}(s,t) = \phi(s,t)\phi(s,t)^T(\phi(s,t)^T\hat{\Sigma}\phi(s,t))^{-1}$. The derivation of Equation (22) and maximization of $\hat{\Sigma}$ follows from Appendix A.3 in [51].

For an on-policy scenario, as presented here, the amount of rollouts needs to be reduced to keep the variance of the expectation in Equation (22) low. To reduce the rollouts, a concept known as importance sampling is used. This concept has been introduce for RL in [4, 90]. To achieve it, the expectation in Equation (22) is changed to the importance sampler denoted as $\langle\cdot\rangle_{w(\Omega)}$. There exist, however, a problem with the importance sampler in RL, and that is fragility. In this case, the consequence is a biased update, occurring when the parameters are updated with the accumulated mass of several low rewarded rollouts. To avoid this problem, low weighted rollouts are discarded. A good practise is to keep the j best rollouts, a number which is of the same magnitude as the the number of parameters N.

PoWER is presented in pseudo code in Algorithm 4. This algorithm is robust regarding the reward function. It is, however, required to model the reward function as an improper probability distribution, thus restricting the individual rewards to only positive values. If it is possible for the rewards to sum up to one, as for a proper probability distribution, learning speed can be increased [51].

The convergence rate of the PoWER algorithm depends upon the amount of open parameters, a low value improves convergence rate while a high value worsens it. However, in comparison to non-EM inspired algorithms, this problem is of less significant implication [51].

Algorithm 4 EM Policy Learning by Weighted Exploration with the Returns (PoWER). (Source: [51]).

Input: initial policy parameters $\theta_{\mathbf{0}}$
repeat
Sample: Perform rollout(s) using $\mathbf{a} = (\theta + \epsilon_{\mathbf{t}})^{\mathbf{T}}\phi(\mathbf{s}, \mathbf{t})$ with $\epsilon_{\mathbf{t}}^{\mathbf{T}}\phi(\mathbf{s}, \mathbf{t}) \sim \mathcal{N}(\mathbf{0}, \phi(\mathbf{s}, \mathbf{t})^{\mathbf{T}}\hat{\mathbf{\Sigma}}\phi(\mathbf{s}, \mathbf{t}))$ as stochastic policy and collect all $(\mathbf{t}, \mathbf{s_t^h}, \mathbf{a_t^h}, \mathbf{s_{t+1}^h}, \epsilon_{\mathbf{t}}^{\mathbf{h}}, \mathbf{r_{t+1}^h})$ for $\mathbf{t} = \{\mathbf{1}, \mathbf{2}, \ldots, \mathbf{T+1}\}$.
Estimate: Use unbiased estimate
$$\hat{Q}^{\pi}(s, a, t) = \sum_{\widetilde{t}=t}^{T} r(s_{\widetilde{t}}, a_{\widetilde{t}}, s_{\widetilde{t}+1}, \widetilde{t}).$$
Reweight: Compute importance weights and reweight rollouts, discard low-importance rollouts.
Update policy using
$$\theta_{\mathbf{k+1}} = \theta_{\mathbf{k}} + \langle \textstyle\sum_{\mathbf{t=1}}^{\mathbf{T}} \mathbf{W}(\mathbf{s}, \mathbf{t})\mathbf{Q}^{\pi}(\mathbf{s}, \mathbf{a}, \mathbf{t})\rangle_{\mathbf{w(\Omega)}}^{-\mathbf{1}} \langle \textstyle\sum_{\mathbf{t=1}}^{\mathbf{T}} \mathbf{W}(\mathbf{s}, \mathbf{t})\epsilon_{\mathbf{t}}\mathbf{Q}^{\pi}(\mathbf{s}, \mathbf{a}, \mathbf{t})\rangle_{\mathbf{w(\Omega)}}$$
with $\mathbf{W}(\mathbf{s}, \mathbf{t}) = \phi(\mathbf{s}, \mathbf{t})\phi(\mathbf{s}, \mathbf{t})^{\mathbf{T}}(\phi(\mathbf{s}, \mathbf{t})^{\mathbf{T}}\hat{\mathbf{\Sigma}}\phi(\mathbf{s}, \mathbf{t}))^{-\mathbf{1}}$.
until Covergence $\theta_{\mathbf{k+1}} \approx \theta_{\mathbf{k}}$

4.4 Policy Improvement with Path Integrals

In contrast to the policy search methods based on maximizing the lower bound, there also exist another method based on SOC and path integrals, where the goal is rather to define and minimize the cost function. This theory is employed in [94], and results in the PI^2 algorithm. This algorithm, in its essence, is extremely simple as the only open tuning parameter is the exploration noise. However, before presenting the algorithm some theory behind optimal control needs to be established. The theory

and derivation of the main equations are from [89], but for a full coverage the reader is referred to [94].

In its essence SOC defines a control system as

$$\dot{\boldsymbol{x}}_t = \boldsymbol{f}(\boldsymbol{x}_t) + \boldsymbol{G}(\boldsymbol{x}_t)(\boldsymbol{u}_t + \epsilon_t) = \boldsymbol{f}_t + \boldsymbol{G}(\boldsymbol{u}_t + \epsilon_t), \tag{23}$$

where $\boldsymbol{x}_t$ is the state vector of the system, $\boldsymbol{G}_t = \boldsymbol{G}(\boldsymbol{x}_t)$ is the control matrix, $\boldsymbol{f}_t = \boldsymbol{f}(\boldsymbol{x}_t)$ is the passive dynamics, $\boldsymbol{u}_t$ is the control vector and ϵ_t is noise sampled from a zero mean Gaussian distribution with a covariance matrix $\boldsymbol{\Sigma}$.

The value function is defines as

$$V(\boldsymbol{x}_{t_i}) = V_{t_i} = \min_{u_{t_i:t_N}} E_{\boldsymbol{\Omega}_i}[R(\boldsymbol{\Omega}_i)], \tag{24}$$

which is minimized by finding the controls $\boldsymbol{u}_t$ over the expectation of the cost R. The cost is accumulated from a trajectory Ω_i as

$$R(\boldsymbol{\Omega}_i) = \phi_{t_N} + \int_{t_i}^{t_N} r_t dt, \tag{25}$$

where Φ_{t_N} is the reward received at the end of the trajectory at time t_N, and r_t is the immediate reward obtained at time t. From the definition of the reward function in Equation (25), the trajectory has a finite time horizon starting at state x_{t_i} at time t and ending at state x_{t_N} at time t_N.

The minimization of the value function in Equation (24) is coupled to the solution of the stochastic Hamilton-Jacobi-Bellman (HJB) equation [30, 88], and results in

$$\partial_t V_t = q_t + (\nabla_{\boldsymbol{x}} V_t)^T \boldsymbol{f}_t - \frac{1}{2}(\nabla_{\boldsymbol{x}} V_t)^T \boldsymbol{G}_t \boldsymbol{R}^{-1} \boldsymbol{G}_t^T (\nabla_{\boldsymbol{x}} V_t) + \frac{1}{2} trace\left((\nabla_{\boldsymbol{xx}} V_t) \boldsymbol{G}_t \boldsymbol{\Sigma}_\epsilon \boldsymbol{G}_t^T\right) \tag{26}$$

$$\boldsymbol{u}(x_{t_i}) = \boldsymbol{u}_{t_i} = -\boldsymbol{R}^{-1} \boldsymbol{G}_{t_i}^T (\nabla_{x_{t_i}} V_{t_i}), \tag{27}$$

where u_{t_i} is the optimal control at time t_i. With these equations in mind, the PI^2 algorithm seeks to find a solution to the nonlinear second order partial differential equation (26), and subsequently apply the solution to improve the already obtained policy. To realize this, the stochastic optimal control is transformed into a *Generalized Path Integral Control* (GPIC), of which the PI^2 algorithm happens to be a special case.

To derive the GPIC from the SOC three main steps, defined in [94], are necessary:

1. Transform the value function into a logarithmic function defined in [45] as $V_t = -\lambda \log \psi_t$, and simplify $\lambda \boldsymbol{R}^{-1} = \Sigma$. This results in a linear *Partial Differential Equation* (PDE) called the Chapman Kolmogorov partial differential equation.

2. Transforming the result obtained in the previous step, by the Feynman-Kac theorem, into a path integral [45].

3. The path integral obtained in the second step needs to be generalized by separating the control transition matrix $\boldsymbol{G}_t$ into an controllable $\boldsymbol{G}_t^{(c)}$ and uncontrollable part $\boldsymbol{G}_t^{(m)}$ [94].

These steps render the following equations

$$V_t = -\lambda \log \psi_t, \tag{28}$$

$$\psi_{t_i} = \lim_{dt \to 0} \int \exp(-\frac{1}{\lambda} \widetilde{S}(\Omega_i)) d\Omega_i^{(c)}, \tag{29}$$

where

$$\widetilde{S}(\Omega_i) = \psi_{t_N} + \sum_{j=1}^{N-1} q_{t_j} dt + \boldsymbol{C}_{t_i}. \tag{30}$$

The integral in Equation (29) is performed over the rollouts $d\Omega_i^{(c)} = (d\boldsymbol{x}_{t_i}, \ldots, d\boldsymbol{x}_{t_N})$. The cost function $\widetilde{S}(\Omega_i)$ can be seen as the cost-to-go from the current time step i to the end. This way of interpreting the cost function is widely known in dynamic programming [9].

To get a global optimal solution for the value function in Equation (28) all possible trajectories needs to be executed. This, however, is not possible for a high-dimensional problem because the possible trajectories are too many. Thus, to solve the problem a subset of the trajectories are selected. This, on the other hand, makes it problematic to find low cost trajectories as many specific trajectories are not performed. Moreover, the dynamics of the system might bias possible trajectories, avoiding large state spaces and possible solutions. However, this is the price of having an relative easy path integral representing the value function rather then the extremely difficult HJB equation.

To obtain the optimal control minimizing the cost function, Equation (28) is inserted into Equation (27), resulting in

$$\boldsymbol{u}_{t_i} = \lambda \boldsymbol{R}^{-1} \boldsymbol{G}_{t_i} \frac{\nabla_{x_{t_i}} \boldsymbol{\Psi}_{t_i}}{\boldsymbol{\Psi}_{t_i}}, \tag{31}$$

where Ψ_{t_i} is known from Equation (29). By inserting the expression for Ψ_{t_i} into the previous equation and simplifying it, the final expression for the optimal control is reduced to

$$\boldsymbol{u}_{t_i} = \int P(\boldsymbol{\Omega}_i) \boldsymbol{D}(\boldsymbol{\Omega}_i) \boldsymbol{\epsilon}(\boldsymbol{\Omega}_i) d\boldsymbol{\Omega}_i^{(c)}, \tag{32}$$

where $P(\Omega_i)$, $D(\Omega_i)$ and $\epsilon(\Omega_i)$ defines, respectively, the probability of a trajectory, the local control, and the perturbed trajectory, all at time step i. The probability term $P(\Omega_i)$ is calculated as

$$P(\boldsymbol{\Omega}_i) = \frac{\exp\left(-\frac{1}{\lambda} \widetilde{S}(\boldsymbol{\Omega}_i)\right)}{\int \exp\left(-\frac{1}{\lambda} \widetilde{S}(\boldsymbol{\Omega}_i)\right) d\boldsymbol{\Omega}_i}, \tag{33}$$

and the local control as

$$\boldsymbol{D}(\boldsymbol{\Omega}_i) = \boldsymbol{R}^{-1}\boldsymbol{G}_{t_i}^{(c)T}(\boldsymbol{G}_{t_i}^{(c)}\boldsymbol{R}^{-1}\boldsymbol{G}_{t_i}^{(c)T})\boldsymbol{G}_{t_i}^{(c)}. \tag{34}$$

Is is worth noticing that the probability of a trajectory is inversely proportional to the cost-to-go given by the cost function in Equation (30). Hence, a lower cost trajectory entails a higher probability, and vice versa for a higher one.

The local control $D(\Omega_i)$, which is needed for calculating the optimal control, depends on the control matrix $G(x_t)$ and the passive dynamics $f(x_t)$. This dependency causes GPIC to be model-based and not applicable to improving parametrized policies, which in this thesis are rendered by the DMP. Therefore, to apply the GPIC to improving trajectories learned by DMP, it has to be made model-free. This is realized by replacing the passive dynamics $f(x_t)$ with the spring-damper system, and the control matrix with a basis function g_t defined as

$$u_t = g_t^T(\theta + \epsilon_t), \tag{35}$$

where ϵ_t is Gaussian exploration noise $\mathcal{N}(0, \boldsymbol{\Sigma})$, and θ is the current policy parameter. The basis function is equivalent with the one defined in Equation (7), but with ϵ_t added as noise to the already known weights w_i as to explore a new policy. The variance $\boldsymbol{\Sigma}$ is a user defined variable, and by changing it the exploration also changes. These substitutions yields what is known as the PI^2 algorithm, presented in pseudo code in Algorithm 5.

Algorithm 5 Policy Improvement with Path Integrals (PI^2). (Source: [94])

Input:

– An immediate cost function $rt = q_t + \boldsymbol{\theta}_t^T \boldsymbol{R} \boldsymbol{\theta}_t$

– A terminal cost term ϕ_{t_N}

– A stochastic parameterized policy $a_t = g_t^T(\theta + \epsilon_t)$

– The basis function g_{t_i} from the system dynamics

– The variance $\Sigma_{\boldsymbol{\epsilon}}$ of the mean-zero noise $\boldsymbol{\epsilon_t}$

– The initial parameter vector $\boldsymbol{\theta}$

repeat

– Create K rollouts of the system from the same start state x_0 using stochastic parameters $\boldsymbol{\theta} + \boldsymbol{\epsilon}_t$ at every time step

– **For** $k = 1, \ldots, K$, compute:

* $\frac{e^{-\frac{1}{\lambda}\widetilde{S}(\boldsymbol{\Omega}_i)}}{\sum_{k=1}^{K}[e^{-\frac{1}{\lambda}\widetilde{S}(\boldsymbol{\Omega}_i)} d\boldsymbol{\Omega}_i]}$

* $S(\boldsymbol{\Omega}_{i,k}) = \phi_{t_N,k} + \sum_{j=1}^{N-1} q_{t_j,k} + \frac{1}{2}\sum_{j=i+1}^{N-1}(\boldsymbol{\theta} + \boldsymbol{M}_{t_j,k}\boldsymbol{\epsilon}_{t_j,k})^T \boldsymbol{R}(\boldsymbol{\theta} + \boldsymbol{M}_{t_j,k}\boldsymbol{\epsilon}_{t_j,k})$

* $\boldsymbol{M}_{t_j,k} = \frac{\boldsymbol{R}^{-1}\boldsymbol{g}_{t_j,k}\boldsymbol{g}_{t_j,k}^T}{\boldsymbol{g}_{t_j,k}^T \boldsymbol{R}^{-1}\boldsymbol{g}_{t_j,k}}$

– **For** $i = 1, \ldots, N-1$, compute:

* $\delta\boldsymbol{\theta}_{t_i} = \sum_{k=1}^{K}[P(\boldsymbol{\Omega}_{i,k})\boldsymbol{M}_{t_i,k}\boldsymbol{\epsilon}_{t_i,k}]$

– $[\delta\boldsymbol{\theta}]_j = \frac{\sum_{i=0}^{N-1}(N-i)w_{j,t_i}[\delta\boldsymbol{\theta}_{t_i}]_j}{\sum_{i=0}^{N-1} w_{j,t_i}(N-i)}$

– Update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \delta\boldsymbol{\theta}$

– Create one noiseless rollout to check the trajectory cost $R = \phi_{t_N} + \sum_{i=0}^{N-1} r_{t_i}$.

until until convergence $R_{t+1} \approx R_t$

4.5 Incorporating Reinforcement Learning with Dynamic Movement Primitives

So far all the algorithms presented have been used to improve an already known policy. The policy, in this thesis, is encoded as a DMP as explained in Chapter 3. However, nothing has been said about the connection between the DMP and RL, or more exactly between the DMP and the parameter θ which is to be optimized to improve the already known policy. Therefore, a connection has to be established, which in the case of having only one DMP is quite trivial as the policy parameter θ can be interchanged with the shape parameters (weights) in the forcing function given in Equation (7). The reason behind this is that the weights are responsible for defining the spatio temporal path of the movement. Thus, by adding noise to the learned weights a new path will follow. By repeatedly performing rollouts in this manner, and subsequently updating the weights according to the algorithms the known policy will improve.

4.6 Evaluation of the Policy Improvement Algorithms PoWER and PI^2

In this section the algorithms previously presented will be compared. Based on the results one algorithm will be chosen to improve the initial policy generated from a DMP and performed on a robot for the sake of playing the ball-in-a-cup game.

First of all, the PoWER and PI^2 algorithms cannot be easily compared. This follows from the different definitions of the reward function. In the PoWER algorithm the immediate reward function has to be an improper probability, restricting it to only positive rewards which also needs to integrate to a constant value, whereas no such restrictions exists for the PI^2 algorithm. Therefore, the reward function in the PoWER algorithm needs to be transformed, usually in a nonlinear fashion, to be able to handle immediate rewards as defined for the PI^2 algorithm. Otherwise, both algorithms update the policy in a similar manner, and use interchangeable policy perturbation methods.

One case where both the PoWER and the PI^2 algorithms were compared was in [95]. The task to execute was to move from one point to another through a predefined via point. The movement was generated from a one DoF DMP, and the algorithms role were to optimize the policy parameters. The results can be seen in Figure 17. From this figure it can easily be seen that both the PoWER and PI^2 algorithm perform better than the other algorithms discussed in this chapter. However, it is extremely difficult to distinguish between the performance of the PoWER and PI^2 algorithm because the resulting policies have almost the same values. Thus, in this case, it is impossible to determine which is better. With this in mind, selecting one over the other based on their performance alone is not possible in this case. The PI^2 algorithm could be chosen for the simpler reward function definition. However, as mention previously in this chapter, the PoWER algorithm has been used before for policy search in the ball-in-a-cup game [50] and is therefore chosen to be implemented to see if comparable results can be obtained.

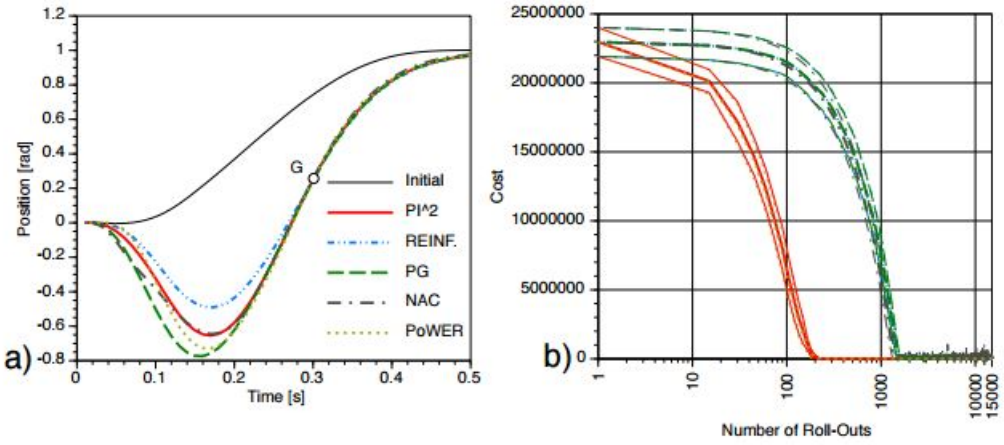


Figure 17: The performance of the PoWER algorithm drawn as a dashed yellow line and the PI^2 drawn as a red line are indistinguishable. In the left image a) both algorithms follows a policy which fulfils the via point restriction at 300 ms and ends up at the correct position at about 0.5 s. In the right image b) the learning curve is presented. The number of rollouts are averaged over 10 runs per algorithm. (Source :[95]).

5 Testbed

The experiments outlined in this thesis will be evaluated on a real robot. Thus, this chapter will be devoted to explaining the *Hardware* (HW), *Middleware* (MW) and *Software* (SW) used to realize the experiment. The HW, MW and SW parts consists, respectively, of the robot and all its peripheral, the already available software, and the chosen software architecture and its functions. Combined, these parts completes the testbed capable of robot sensing and control in hard real-time.

5.1 Overview

The testbed, from HW to SW, used in this thesis constitutes of three components, a *Kuka Light-Weight Robot* (LWR), a *KUKA Robot Controller* (KRC), and an external computer. These components are all integrated together as depicted in Figure 18. The reason behind the chosen infrastructure is to allow for hard real-time control of the robot. As the HW, MW, and SW all are important for the overall success of the system they will be explained in more detail in the following sections, starting with HW, then MW and last SW.

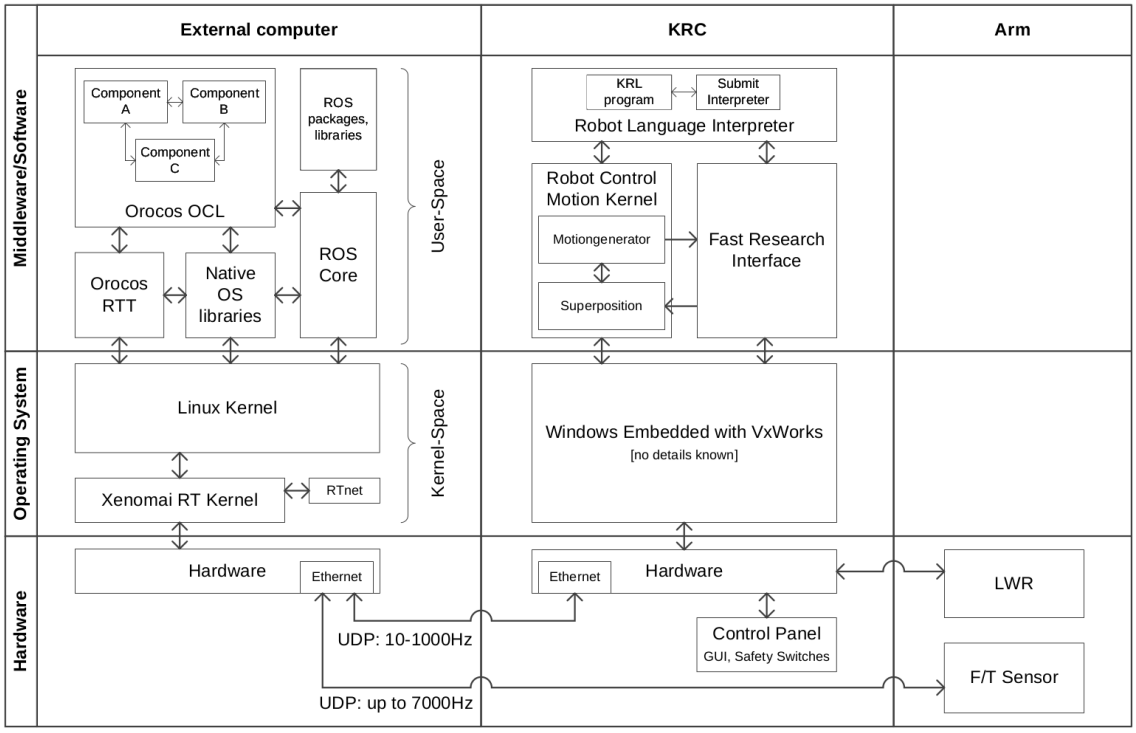


Figure 18: The overall view of the system. As is depicted is the three components: KRL, KRC and the external computer, as well as the intra-connection between each component. (Source: [87]).

5.2 Hardware

Industrial robots are gaining popularity each year mostly due to their high usability and fairly low cost. The essence of an industrial robot is their ability to repetitively perform accurate motions at high speeds, while at the same time be robust and durable [2]. To uphold all of these characteristics, the industrial robot is designed to have a high stiffness. The high stiffness unfortunately affects the mass to payload ratio in a negative way. Furthermore, they also require the environment to be more or less static, requiring predefinition of all parts and excluding potential collision with other objects and humans. With all of these advantages and disadvantages in mind, the industrial robot surly have its place in the modern world. However, as more and more attention is placed on human-robotics interaction, which the learning from demonstration framework is a prime example of, a safe and highly controllable robot is required. Regarding safety, the robot needs to be able to detect and react to collisions, whereas controllability enables a human to easily interact with the robot by, for instance, grabbing it and guiding it to the right location.

As a result of the need of human-robot interaction, new robots have been developed to overcome the limits presented by the industrial robots. One of these robots is the KUKA LWR. What sets KUKA aside from industrial robots is the integrated *Force/Torque* (F/T) sensors in every joint [12]. With the aid of these F/T sensors collisions can be detected in real-time and the robot can perform complementary actions, as completely stopping the movement, before any harm is done. The collision detection in combination with a light weight and kinematic redundancy enables the robot to be used cooperatively by humans without endangering their safety. This enables cutting edge research such as learning from demonstrations. Albeit the state of the art characteristics presented in the KUKA LWR, it is still far from as highly functional as the human arm which can detect vibration, temperature, pressure, and much more thanks to its multimodal senses.

As the KUKA LWR 4+ possesses all great features for human-robot interaction it was chosen as the HW performing the experiments in this thesis. It was designed with research in mind [12]. The robotic arm has a slim body and seven elastic joints enabling it to effectively move in 7 DoF (see (1) in Figure 19). Moreover, each joint is integrated with an F/T sensors and the overall repeatability of the arm is ± 5 mm [12]. It can handle a payload of 7 kg while its own mass is only 15 kg. Another important characteristic of the robot is the ability to actively set the compliance on either joint or Cartesian level. By setting the robot compliant in a specific Cartesian direction it can be moved or pushed in that direction without resistance. In this thesis the movement was restricted to a plane, thus the robot was compliant in two Cartesian directions but non-compliant in the third. All in all, the slim and round body, collision detection and adaptive stiffness settings makes the KUKA LWR 4+ a suitable HW to implement the LfD framework on.

However, the HW setup does not only consist of the LWR but also on six other components, namely the KRC, the ball-in-a-cup game, the external computer, the *KUKA Control Panel* (KCP), and the kinect sensor. These components and their mutual connection are illustrated in Figure 19.

The ball-in-a-cup game was built from an aluminium can with a height of 5.0 cm and a diameter of 8.5 cm. The string was attached through a hole in the bottom of the can. A wooden ball with a weight of 10.35 grams was attached to the other end of the string.

The KRC includes a power unit, a control PC, a connection panel, and safety logic. The control PC is the main user interface to the robot and runs Windows XP. The user interface is made up of a preinstalled editor for writing programs in the *KUKA Robot Language* (KRL). The main purpose of the KRC is to ensure safe operation when controlling the robot.

The external computer facilitates an 8-core processor and runs Ubuntu[1] extended with the real-time kernel Xenomai. The external computer eases the software development as it provides the ability to write the software which controls the robot in another programming language than KRL (mainly C++ or Python). This is realized by utilizing the *Fast Research Interface* (FRI) developed by KUKA and The Deutche Luft und Raumfarth.

For the tracking of the ball and the cup a Kinect sensor[2] was used. This stereo vision camera can capture both RGB images and the point cloud of the scene, thus enabling object identification through colour segmentation and distance calculation from the point cloud. The frame rate of the camera depends on the resolution of the image. As tracking of the object at high speed is needed the highest frame rate was chosen which was 30 frames per second. At this frame rate the image resolution was 640×480 pixels.

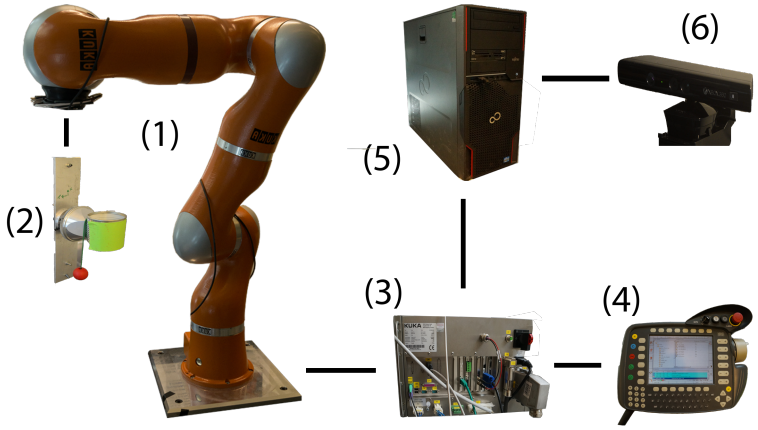


Figure 19: The hardware included in the testbed. (1) depicts the KUKA LWR4+, which is attached to the ball-in-a-cup game (2). The LWR is also connected to the KRC (3) which in turn is connected to both the KCP (4) and the external computer (5). The external computer is in turn connected to the Kinect sensor (6).

[1] Version 12.04 LTE
[2] Version 1, model 1414

5.3 Impedance Controller

To enable both teaching and execution of the movement in a plane, the Cartesian impedance controller of the KUKA LWR is used. The Cartesian impedance controller steers the arm by working in conjunction with the individual torque controllers in each joint. The controller is implemented as a spring/damper system along each axis of rotation and translation, where the stiffness and damping factor can be dynamically set for each of these [33].

The control law of the Cartesian impedance controller implemented in the LWR defined in [85] is the following

$$\tau_{cmd} = J^{'} \left(k_c(x_{cmd} - x_{msr}) + D(d_c) + F_{cmd}\right) + f_{dynamics}(q, \dot{q}, \ddot{q}),$$

where $J^{'}$ is the Jacobian matrix responsible for converting the forces at the end-effector into joint torques (τ_{cmd}) commanded to the robot. The spring is represented with the factor $k_c(x_{cmd} - x_{msr})$, where k_c is the stiffness factor, x_{cmd} is the commanded Cartesian position of the end-effector and x_{msr} is the measured position of the end-effector. The damping is represented by $D(d_c)$, where d_c is the damping factor. The commanded Cartesian position, stiffness factor, damping factor as well as the superposed Force/Torque factor F_{cmd} can be dynamically set by the user to the robot. The last factor $f_{dynamics}(q, \dot{q}, \ddot{q})$ counteracts the gravity torques, centrifugal and Coriolis forces [10].

The control law basically integrates two different control modes: one for position and one for force. If no commanded force, represented as F_{cmd}, but only a commanded position of the end-effector is sent to the robot the commanded position of the end-effector is reached as a spring/damper. On the other hand, if only the commanded force is sent to the robot, the robot arm is controlled by moving it in the same direction as the commanded force vector and exerts a counteracting force if some external force is applied to the robot. In some cases the commanded force F_{cmd} and the force generated from the spring system $F_k = k_c(c_{cmd} - x_{msr})$ can cancel out each other. In such a case, it can happen that neither the positional or the force profiles can be followed with high accuracy resulting in a poor reproduction of the learned task.

If both the commanded force and stiffness factor is set to zero the robot can be freely moved in space as the gravitational and frictional forces are counteracted by the robot. In the teaching part in this thesis, the commanded forces are set to zero and the stiffness is set high in all directions except in the y- and z-directions as this enables the teaching of the robot in only a plane. On the other hand, when the learned movement is to be executed the stiffness is set to be high in all direction as the relative displacement $k_c(x_{cmd} - x_{msr})$ between the desired Cartesian position and the actual position is influenced by the stiffness parameter, where a higher value will result in higher accuracy and vice versa for a lower value [33]. However, setting the stiffness value over or close to the maximum values indicated in [33] can result in a jerky execution.

5.4 Middleware

The MW can be visualized as a bridge between the software already developed or to be developed and the actual hardware. The main point of utilizing MW in this thesis is that it eases the development of new software by providing a simple approach to communicating with the robot in hard real-time. Furthermore, the MW also consist of a large library of already developed code, for example image processing tools, which reduces the need to develop new code. In this thesis the MW consists of the *Robot Operating System* (ROS)[3], the *Open Robot Control Software* (Orocos)[4] and the FRI.

5.4.1 ROS and OROCOS

ROS and OROCOS eases the development of new software by providing a lot of already built in robotic software. The reason for using OROCOS is the hard real-time constraints which it can handle but ROS cannot. On the other hand, ROS comes pre-packed with a lot more developed software than OROCOS.

The general goal with the OROCOS project is to develop robot control software which are flexible, modular and not bound a specific platform or robotic device [15]. OROCOS facilitates an event triggered generic core [86]. A program written in OROCOS consist of one or several components and one deployer. The structure of a component can be seen in Figure 20. It facilitates input and output ports to communicate with other OROCOS components as well as operations and operation callers. As the name suggests, input ports are ports where data is written into the component and output ports are ports to which a component can write data to for other components to read from. Operation callers calls an operation from another component. A component is executed in a single thread written in plain C/C++ or in the *Orocos Program Script* (OPS) [86]. The deployer file can be visualized as the bridge between all the components. In this file all the OROCOS components which are to be used are specified and started. Furthermore, the individual activities of each component alongside their connection to the other components are defined. The activities can be defined as periodic, non-periodic or slave activities where the execution starts when other activities starts.

In contrast to OROCOS, the middleware also depends on ROS for non hard real-time tasks. The goal with ROS is to ease the development of robotic software by providing an extensive library of already developed software, not requiring the programmer to "reinvent the wheel" [67]. ROS has gained an immersive popularity in the robotics society and can be considered the de facto MW to be used in robot programming [26]. ROS programs are implemented as nodes with a publish-subscribe communication scheme and intra communication with other ROS nodes are realized by writing messages of a specific type to a predefined topic name [67]. Other ROS nodes, even within the same network, can then read the data from the topic names by subscribing to them.

[3]The ROS version used was Groovy
[4]The version used was 2.6

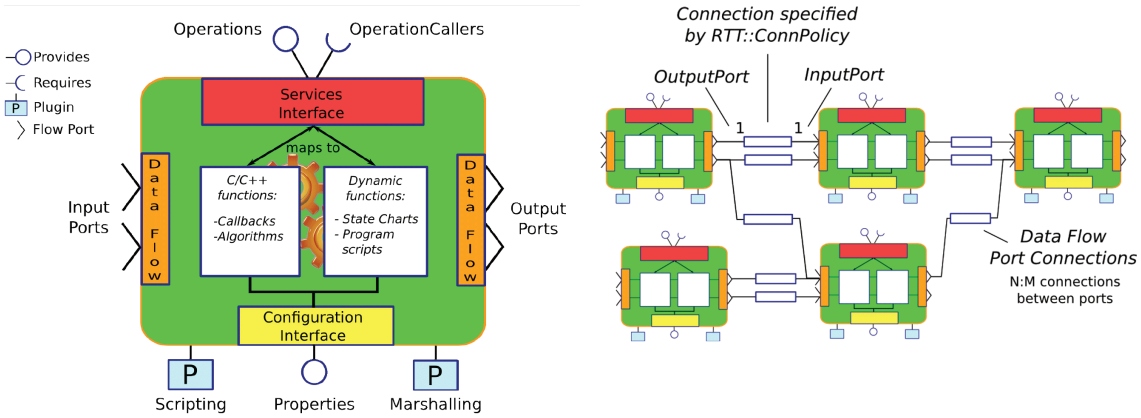


Figure 20: An OROCOS program is called a component. Each component provides operations and operation callers, which can be thought of as methods. A component also facilitates input and output ports to communicate with other orocos components. (Source: [86]).

To integrate the hard real-time functionality of OROCOS with ROS, topics from ROS nodes can not only be read by other ROS nodes but can also be streamed into OROCOS ports or vice versa. This feature is still limited, however they do fulfil the requirements for the software used in this thesis as it is mainly developed in ROS. The reason for implementing most of the software as ROS nodes was that all the components with hard real-time requirements had mostly already been developed.

5.4.2 Fast Research Interface

The FRI was implemented as a result of a requirement analysis together with the robotic society in [85]. The questionnaire revealed what main purpose the FRI should fulfil. These were: a low-level real-time connection with the KRC at rates up to 1 kHz, freedom to integrate it with an operating system of choice, and a possibility to use the already developed controller in the KUKA LWR. Functions which has already been integrated in the controller are teaching, control mode selection (position or impedance), safety systems and much more [85]. As these functions were implemented in the KRL, FRI grants access to them from another programming language, effectively removing the need to reprogram them. An overview of the system architecture of the FRI is presented in Figure 21 where the connection between the FRI running on the KRC and the FRI Remote running on the external computer is established though a UDP connection.

The interface which grants access to a low level control of the robot is implemented as a state machine and accommodates two different modes: the command and the control mode. In the control mode the external computer can monitor the robot but, as the names suggests, the controlling of the robot is only possible in the commanding mode. In this mode the remote computer can receive sensor readings and status reports from the robot, set specific parameters (e.g. stiffness and damping), request the control mode of the robot, and command the robot to a specific position.

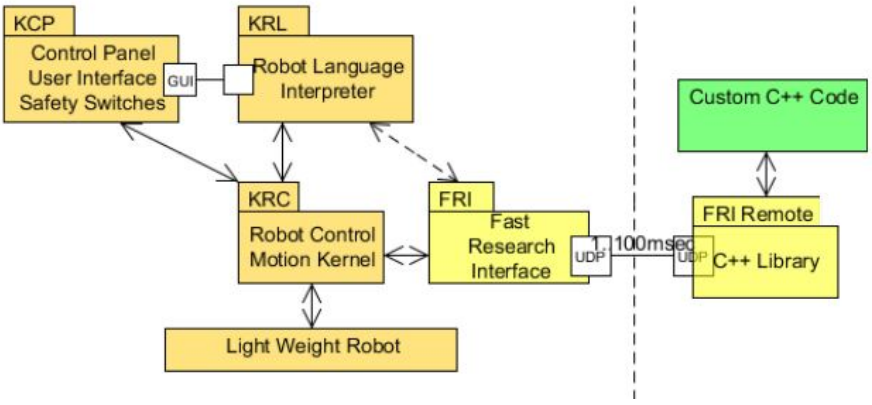


Figure 21: An overview of the system architecture of the FRI. (Source: [85]).

When the FRI is initially started the mode is set to control mode. If the user wants to switch from the control mode to the commanding mode a handshaking between the remote computer and the FRI needs to be fulfilled. The handshaking is realized by following some specific steps outlined next. As the FRI is started it enters the monitor mode and while being in this mode packages are sent from the KUKA controller to the remote computer. The remote computer then has to answer to these packages by sending answer packages within a predefined time frame. If the answer packages are received by the FRI within the predefined time frame the handshaking is considered successful and the mode can be switched.

5.5 Software Architecture

The software in this thesis was designed as a modular framework as this will ease the possibility of reusing it for other applications. Most of the software used in this thesis had already been developed by Franz Steinman for his master thesis [87]. However, others have been improving the code since the first release, for example, the initial software design did not set the kernels of the DMP equally in time, but this is solved in the improved software. Although others have contributed and improved the code, the basic architecture has not changed much. As the architecture is thoroughly explained in [87], it will not be covered in great details here. However, some modifications were needed to adapt the already implemented code to work for the application defined in this thesis. Thus, the focus of the next chapters are to discuss these changes as well as to give a thorough explanation of the software which were developed in this thesis. The developed software comprehends the following classes: `vision`, `Logic`, `PoWER`, `Reward`, and `MultivariateGaussian`. The first three classes are implemented as ROS nodes while the latter two are standalone C++ classes.

The whole chain for acquiring a new skill can be split into four phases: demonstration of the skill, learning the skill, executing the skill, and improving the skill. This chain can be seen in Figure 22. The already developed software comprehends the recording, learning and execution phase. Thus, these phases together with the basic components, enabling the actual communication with the KUKA LWR, are explained

in the next two chapters. After this the newly developed software is introduces.

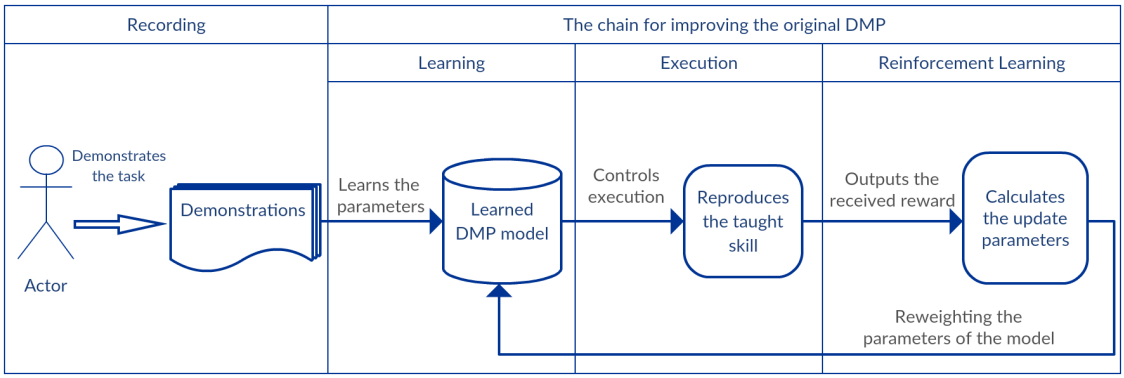


Figure 22: For the robot to learn a trajectory and subsequently improve upon it four phases needs to be completed in sequence. The first two parts, demonstration and learning, is only done once, whilst the last two, execution and reinforcement learning is executed iteratively as to improve the learned model.

Basic components

The basic components can be seen in Figure 23. The OROCOS components `FT Sensor`, `KUKACommander`, `KUKACommanderROS`, `FRI ServerRT`, and the KRC component `FRI Control` all enables straightforward control of the LWR. First of all the `FRI SeverRT` acts as a bridge between the OROCOS components and the KRC, and is responsible to send the correct data to the correct ports of the KRC over a UDP protocol. To enable easy activation of the basic services of the LWR, for instance activating gravity compensation, the `KUKACommander` is used. To simplify the sharing of data between ROS nodes and OROCOS components the `KUKACommanderROS` ROS node was developed in [87]. This ROS node is foremost responsible of exposing the services of the `KUKACommander` to other ROS nodes. These services are activated by simply publishing or subscribing to predefined ROS topics set in the `KUKACommanderROS`.

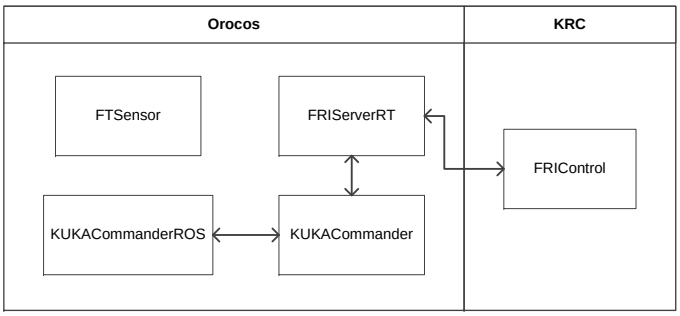


Figure 23: These OROCOS components were already implemented as to ease the communication with the LWR.(Source: [87]).

Recording Phase

To acquire a skill through a human demonstration only one component of the basic framework is used, namely the ROS package `Recorder_Jens`. The workflow of the recorder node is visualized in Figure 24. When the node is initiated it asks if the movement should be recorded on trajectory level, meaning that the node starts subscribing to topics which contains data corresponding to the position and orientation of the end effector. When the setup is finished, the KUKA LWR then moves to a previously specified position where the demonstration will start from. As this position is reached, the arm is set stiff in the x-direction as well as the orientation around x, y, and z. This restricts the robotic arm to only be moved along the y- and z-directions. As a result, the movement is only learned in a plane. The actual recording of the movement is started when the user presses `ENTER`. Now the arm can be moved and data is published by the `FRIServer` at a rate of 100 Hz and stored into a ROS bag file. When the demonstration is finished the recording is terminated by pressing `ctrl+C`. The user then has the opportunity to either record another demonstration or terminate the whole node. If another recording is of interest, the whole procedure is reset from the point of moving the arm to a predefined position. Each demonstration is stored in ROS bag files and will later be used for learning a representation of the movement.

In comparison to the previously implemented `Recorder` in [87] the `Recorder_Jens` differs in two aspects. First, instead of activating the gravity compensation mode to allow the user to move the arm to an arbitrary starting position, it instead moves to a previously defined one. Secondly, the arm is restricted to move in only a plane. The reason for restricting the movement to a plane is that this alleviates the need for using two cameras, one above and one perpendicular to the cup. As will be explained in Chapter 6, the feedback given to the reinforcement learning algorithm is based on the distance between the ball and the rim of the cup when the ball crosses it with a downward motion. This distance can only be measured in one direction when one camera is used. If instead the movement had been free in space, two cameras would be necessary, one for tracking the movement of the ball and the other one for recording the distance in, for example, both x- and y-directions.

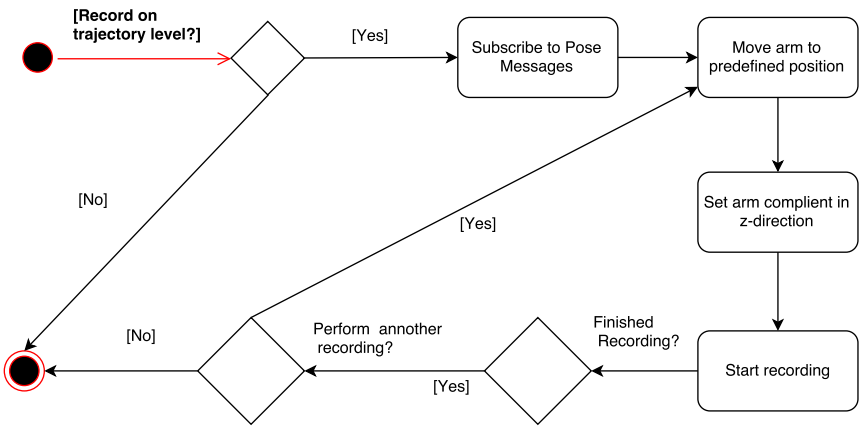


Figure 24: This figures shows the steps to be fulfilled when recording a demonstration.

Logic

To incorporate the reinforcement learning and execution of the DMP in a simple fashion the `Logic` class was developed. This class is implemented as a ROS node and is responsible for learning the initial DMP, then executing the DMP and subsequently calling the RL for reweighing the DMP. The functionality can be seen in Figure 22. The function of the node can be thought of as a state machine which follows the workflow depicted in Figure 25. The first step performed is to read the correct input parameters, such as the number of kernels, the demonstration time τ, the stiffness parameter α_x, along with the file path to the ROS bag file where the demonstration is stored. The bagfile contains the pose of the end effector at different time instances. This file name is then passed on to the `Learner` class which learns the DMP according to Equation 14. As the learning is finished, the initial parameters for initializing the covariance matrix used for sampling noises is created. After this the ROS node `vision` is started. This node is responsible for tracking the ball and the cup and calculate the distance between the ball and the cup. When these steps have been performed, the noises are sampled. The noise together with the learned DMP parameters are then passed on to an OROCOS component in charge of calculating and sending the next state to the FRI.

When one execution of the DMP is finished, the reward class `exp_reward` is called with the distance from the `vision` node as input. This reward class then returns the calculated reward which is passed on together with the sampled noises to the reinforcement learning node `PoWER`. This nose is then responsible for calculating the updated parameter values which are to be added to the current weights of the DMP as well as the optimization parameter β, which will be explained in more detailed in Chapter 6. The whole procedure is then either restarted from the execution phase or terminated if either the user terminates the node or a predefined amount of executions has been reached.

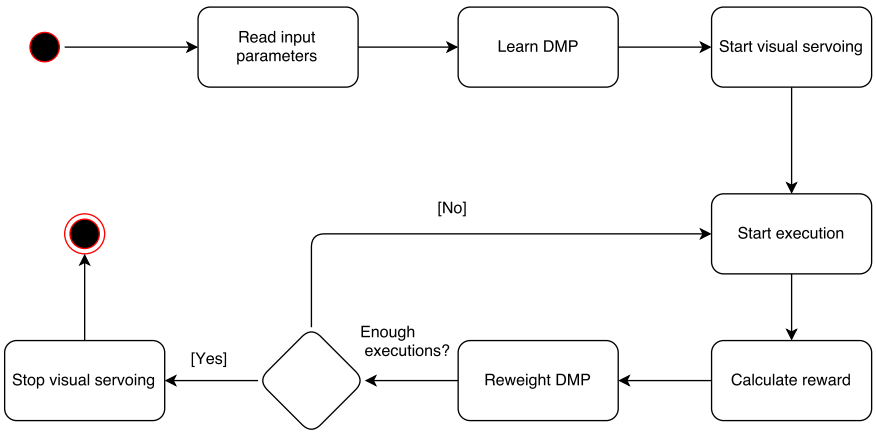


Figure 25: This figure shows the work flow of the `Logic` node.

Reinforcement Learning

The recording and learning phase of the software produces an initial DMP with all the necessary parameters. However, in this thesis the initial DMP has to be improved with subsequent reinforcement learning. Thus, the executor interface and the reinforcement learning has to work together. As discussed in the previous section the `Logic` node executes the initial DMP once and then the new updated parameter values are calculated in the `PoWER` node.

As the reinforcement learning node `PoWER` is only called after an execution of the movement it was sensible to implement is as an ROS action server node. Thus, the node only computes the necessary updates if it is being called from the `Logic` node. The message to the node is the recently sampled noise and received reward. The node then calculates the update parameters according to Algorithm 4 and sends it back as the request. In this way the user can choose when and when not to call the power algorithm. Thus, executing evaluation trials without adding noise or reweighing the parameters is possible.

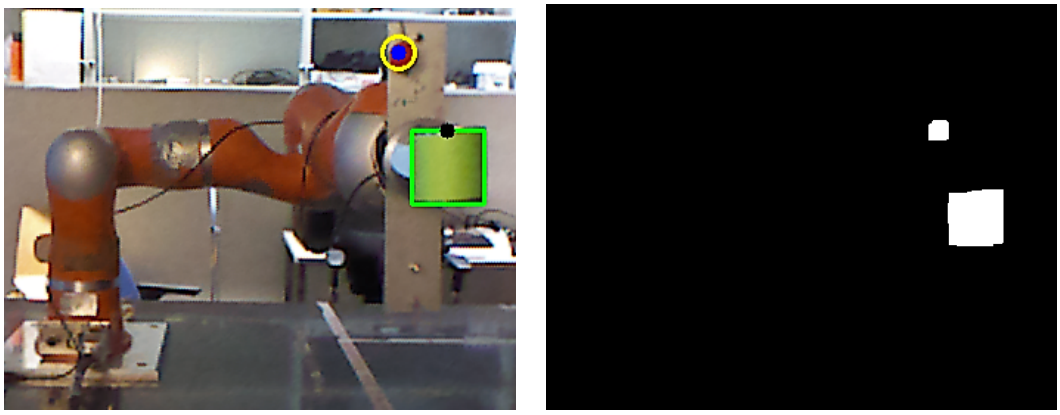
Visual Tracking

The reward given to the reinforcement learning is the distance from the centre of the ball to the rim of the cup when the ball crosses it with a downward motion. This sets up two constraints for the system: first it has to be able to track the ball to know when it is in a downward motion, and secondly it needs to calculate the distance from the ball to the cup when it crosses the rim of the cup.

As previously defined, the setup only consisted of one camera and the movement was restricted to a plane. The visual tracking was implemented as a ROS node called `vision`. This node subscribes to the RGB and depth images published by the Kinect camera. It then publishes the distance from a red object to a green object as soon as the ball crosses the cup with a downward motion. When the `vision.py` node is launched it first calculates the depth to a green object (the cup). This depth is needed to convert the pixel distance between the cup and the ball to SI units.

To enable the distinction of the ball and the cup from each other as well as from the background, colour segmentation is used. First of all, the images are read in as *Red Green Blue* (RGB) images. These images are then converted into *Hue Saturation Value* (HSV) space to simplify the colour segmentation. To further simplify the colour segmentation, the ball is coloured in a sharp red and the cup in a sharp green colour. The colour segmentation can be seen in Figure 26.

When the two objects has been segmented they are tracked with a blob detector. The blob detector is responsible for calculating the topmost centred pixel value of the cup and the centre of the ball in the current and in a previous frame. Based on this information a crossing can be found by checking if the centre of the ball crosses the rim of the cup in the two to fourteen consecutive frames. The reason for keeping track of the centre of the ball in more than just the previous fame is that the camera might loose track of the ball for a couple of frames and thus an interpolation is done between several frames. If a crossing has been found, the distance between the ball and the cup is calculated as shown in Figure 27.



(a) In the figure the blue dot is the center of the red ball and the black dot is the center of the rim of the cup.

(b) This figure is a colour segmented version of the same figure visualized to the left. The segmentation is based on the hsv colours of both the red ball and the green cup.

Figure 26: A example of the colour segmentation of the ball and the cup.

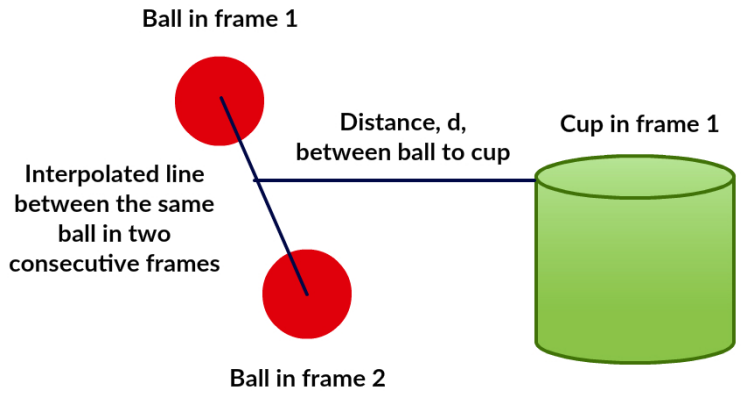


Figure 27: This figure shows how the distance d is calculated. As frame 1 depicts a previous instance in time this means that the ball is falling with a downward motion and also crosses the rim of the cup. Then a line is interpolated between the centre of the balls, and the distance between the cup and the ball is the distance between the interpolated line and the point of the rim of the cup in frame 1.

Helper classes

To ease some of the computations, two helper classes the `MultiVariateGaussian` and the `reward` were implemented. These are implemented as stand alone C++ classes. Hence to use them, an object of each class needs to be created inside the ROS nodes.

The first class, `MultiVariateGaussian`, implements a multivariate Gaussian distribution. To sample noise. the class is called together with a predefined covariance

matrix. The second class, `reward` is a base class for different reward functions. In this thesis the reward is the inverted exponential distance from the ball to the rim of the cup, and thus a derived class `reward_exp` is created from the base class `reward`. For different problems, where the reward function will vary from the one used here, a new derived class with its own reward function needs to be implemented.

5.6 Discussion

The main focus of the implementation is to provide a framework for both learning, executing and improving the learned skill. However, the software was also developed to be as modular as possible. This is foremost recognized as the reinforcement learning (`PoWER`), visual tracking (`vision.py`) and `Logic` are all implemented as separate ROS nodes. The `Logic` node is the main part of the software as this routes all the information in a sensible way. The best practice to reuse the code is to modify the `Logic` node. If another reward function is to be used, one can easily be derived from the base class `reward`. However, the system is known to have the following three significant limitations: the visual tracking of the ball, the restriction of the learned movement to a plain, and the inaccuracy of the impedance controller.

The first limitation originates from visually tracking the ball. The ball can sometimes move so fast that the system do miss the ball completely. The reason for this is that the camera is not physically able to capture sharp enough images of the fast moving ball. This problem can be solved in two ways: either a Kalman filter can be set on the ball to help the tracking, or a camera with a higher framerate can be used.

The second limitation is bound to the restriction of the movement to a plane. As stipulated previously, to allow the movement to be free in space two cameras have to be used, as this enables for calculating the reward based on the distance from the ball to the cup in more than one direction. Steps were taken to alleviate this problem, and an improved ROS node of the original `vision` was actually implemented. This node is supposed to read data from two cameras at the same time. However, the problem was the testing part as apparently reading data from two Kinect cameras at the same time was impossible with the used ROS version. This problem can be solved by updating ROS.

The last limitation is bound to the inaccuracy of the Cartesian impedance controller. As will be presented in the next Chapter, the movement was only learned in two directions and the desired position of the third direction was kept constant. However, the difference between the measured and the actual position of the end-effector showed that the controller was not able to perfectly follow a position set to a constant value. This inaccuracy could sometimes lead to a resulting optimal policy where the generated trajectory induced such a movement on the ball that it was thrown either in front or behind the cup. To alleviated this problem the joint position controller should be used instead, but for this to work the movement has to either be learned in joint space or inverse kinematics has to be used to transform the desired Cartesian position of the end-effector into positions of each joint.

All in all, the developed software made it possible for the aforementioned eval-

uation presented in the next chapter. Moreover, the software is modular and the reinforcement learning is easy to integrate for other applications as well. The limitations of the system are known and can be solved by improving the used hardware and further developing the software.

6 Experiments and Results

To study if the proposed PoWER algorithm can improve an initially taught skill (in this case how to play the ball-in-a-cup game) encoded as a DMP, experiments on actual hardware are needed. Thus, this chapter will be devoted to the experiments. However, before conducting the experiments, the parameters (α_z, β_z, α_x, and the number of kernels) have to be set. Furthermore, to prevent execution of potentially harmful policies on the actual hardware, for example jerky trajectories or trajectories which require high instantaneous acceleration, a thorough simulation was conducted. In this simulation the initial motor primitives were learned, based on data from one demonstration on real hardware, and the parameters were perturbed to see the effect on the reproduced trajectory. Based on the results from the simulation, the exploration rate for the subsequent RL is defined.

One experiment on the actual hardware was conducted. In this experiment the goal was to reproduce the results obtained in [50], as well as study the effect of using several initial rollouts or none at all before starting to reweigh the parameters. To be able to examine this, an initial demonstration was showed to the robot which in turn needed to learn a representation of the movement as a DMP and then subsequently apply RL to improve the learned skill.

In the next section the game, its consecutive states as well as reasoning for choosing this game in the first hand is discussed. After this the initial motor primitive is learned and tested in a simulation built in MATLAB. This is conducted as a feasibility study, and based on the results the actual experiments on the real robot can be conducted. The chapter is concluded with a thorough discussion of the results.

6.1 Ball-In-A-Cup Game

The Ball-in-a-cup game, also known as Balero, Bilboquet or Kendama [1], consists of a cup, a ball and a string . In this thesis, the cup had a height of 5 cm and a diameter of 8.5 cm, while the ball weighed 10.3 grams. The ball hanged down vertically from the cup through a string which was attached to the bottom of the cup. The length of the string was set to 42 cm.

To play the game, the cup should be held in one hand only (or attached to the end effector of the robot) and in a single movement the player should move the cup to induce a movement of the ball through the sting, then pull the cup upwards and catch the ball with the cup. For the movement to be successful, both speed and accuracy has to be very high and precise.

The game has been used in many previous studies in robotics [50, 79, 80, 93] and can thus be seen as a benchmark problem in robotics. Moreover, learning to play the game is even challenging for children who usually succeed in bringing the ball into the cup for the first time after 35 trials [48]. With all this in mind, this game sets a standard for how well the implemented system is able to learn a complex task. Because of these reasons, the ball-in-a-cup game was selected as the game to be learned in the experimental parts in this thesis.

The state of the system is defined as the pose (orientation and position) of the end effector and the position of the cup, both in Cartesian coordinates. For controlling the robot the Cartesian impedance controller (explained in Section 5.3) was used. The robot can move in six degrees of freedom, three for position (x, y and z) and three for orientations (R_x, R_y and R_z). As the movement was restricted to a plane the commanded position in x-direction was set to constant. The other positions and orientations were represented by an individual DMP. Noteworthy is that orientation is represented as quaternions and thus the number of learned DMP was six altogether: one for the y- and z-direction and one for the four quaternions q_x, q_y, q_z and q_w.

The final reward, defined in [50], was calculated as the distance between the rim of the cup and the ball when the ball is in a downward motion

$$r(t) = \begin{cases} \exp(-\alpha d^2), & \text{if } t = t_c, \\ 0, & \text{otherwise} \end{cases} \tag{36}$$

where t_c is the time instant when the ball crosses the rim of the cup with a downward motion, and d is the distance between the rim of the cup and the ball (see Figure 27). The scaling parameter α was set to 100. This reward was then passed on as one of the input parameters to the PoWER algorithm. The higher the reward is the closer the ball is to the rim of the cup and vice versa. If, however, the ball does not cross the rim of the cup with a downward motion the received reward is zero. If this was not a criteria, the robot might learn a policy where it hits the bottom of the cup with the ball.

The number of kernels had to be sufficient to produce an initial policy which got the ball over the rim of the cup and thus receive a reward. From trial and error, 55 kernels were enough to fulfil this requirement. As the movement was restricted to a plane, the primitives which were responsible for the movement in this plane, namely the ones in y- and z-direction, were perturbed. Therefore, the number of parameters (weights) which needed to be fine-tuned by the RL algorithm was 110. This amount is almost half of what was used in [50], where they used 233.

6.2 Evaluation in Simulation

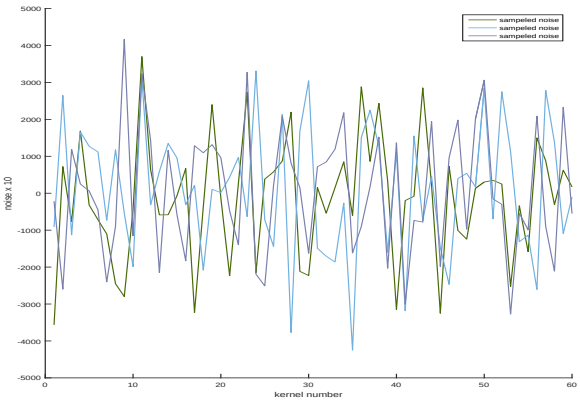
Before implementing the reinforcement learning algorithm on the actual hardware, a simulation of the initially learned representation of the movement was conducted. The simulation was performed in MATLAB. However, for initializing the simulation, data had to be collected from one demonstration on the real hardware. Thus, one successful demonstration, where the ball went into the cup, was performed on the KUKA LWR4+ and the data gathered was the Cartesian positions and orientation of the end effector at each time instant. Based on the collected data, both velocity and acceleration was evaluated, and the initial DMP was learned as in Equation (14). As the initial DMP was learned, the unperturbed and perturbed policies were simulated. To perturb the policy, noise was added to each weight of the DMP. In this way the new policy will differ from the original one by the amount of the added noise. However, as noises can be generated differently, the resulting perturbed policy

will also differ. Thus, the main task of the simulations is to find out safe approaches for generating noise.

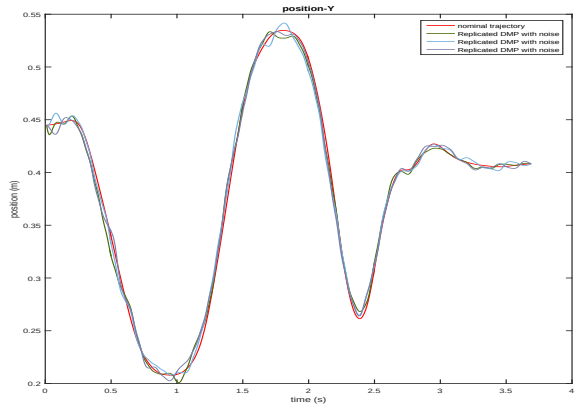
6.2.1 Simulation with Uncorrelated Noise

In the first simulation noise was sampled from a diagonal covariance matrix Σ where each entry on the diagonal $\sigma_{i,j}$ corresponded to the DMP index i and kernel index j. As there are 55 kernels in each DMP the covariance matrix is a 55×55 matrix. The number of different covariance matrices is directly dependent on the number of motor primitives which are to be perturbed, and in this case it was two (one in y- and another one in z-direction). Each variance $\sigma_{i,j}$ was initialized as the median of the weights corresponding to the i-th motor primitive. The noises ϵ were then sampled from a multivariate zero mean Gaussian distribution $\epsilon \sim \mathcal{N}(0, \Sigma)$ and can be seen in Figure 28a. The resulting perturbed trajectories in y-direction can be seen in Figure 28b. From the generated trajectories it can clearly be seen that there exist large deviation in the desired position which, in turn, will lead to large accelerations and a jerky movement. The large accelerations are not possible to execute on a real robot due to the physical limitations of its actuators and the jerky movement will make the robot unable to play the game successfully as the overall movement to be executed has to be very smooth for the ball to end up in the cup. In addition to this problem, the large deviations in position in both the beginning and end of the trajectory is unfavourable as the overall smoothness of the generated trajectory is dependent on the trajectory being smooth in these parts.

Based on these results another approach where the sampled noises generated smooth trajectories was needed. The proposed approach was to sample correlated noise and in the next section it is tested.



(a) These noises are sampled from a diagonal covariance matrix which explains the uncorrelated noise.



(b) The red trajectory is the originally demonstrated one. The other trajectories are simulated with uncorrelated noise added to each parameter. For better visualization the noises are multiplied by 10. However, large deviations in position due to uncorrelated noise can clearly be seen.

Figure 28: Figure (a) visualize the noise which were added to each corresponding kernel for one rollout and (b) is the corresponding trajectory. It is worth pointing out that the noise is multiplied by 10 for visualization reasons.

6.2.2 Simulation with Correlated Noise

In the second simulation the selected covariance matrix had to solve the problems which are inevitably inherited from uncorrelated noise. The new covariance matrix had to fulfil the following characteristics: the sampled noise have (to some extent) be correlated, large accelerations are to be avoided, and perturbations in both the initial and final part of the trajectory should be low. One possibility to generate such noise is presented in [43, 44, 77] where they force the covariance matrix Σ to be

inversely dependent to the sum of squared accelerations along the trajectory $\boldsymbol{\theta}^T\mathbf{R}\boldsymbol{\theta}$, where $\boldsymbol{\theta}$ are the shape parameters and $\mathbf{R}$ is a quadratic control cost matrix. The quadratic control cost matrix $\mathbf{R}$ is defined as

$$\mathbf{R} = \sum_{k=1}^{K} w_k \left\|\mathbf{A}_k\right\|^2 \tag{37}$$

where $\mathbf{A}_k$ is a differencing matrix, w_k is a weight factor, and k is the order of differentiation. As stipulated before, we are interested in squared accelerations; thus $k = 2$ and all weights, except $w_2 = 1$, are set to zero. The resulting $\mathbf{R}$ is then

$$\mathbf{R} = \left\|\boldsymbol{A}_2\right\|^2 = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ -2 & 1 & \cdots & 0 \\ 1 & -2 & \cdots & 1 \\ \vdots & \vdots & \ddots & -2 \\ 0 & 0 & \cdots & 1 \end{bmatrix}. \tag{38}$$

The noises ϵ are then sampled from a zero mean multivariate Gaussian distribution with the covariance matrix $\Sigma = \mathbf{R}^{-1}$ as $\epsilon \sim \mathcal{N}(0, \mathbf{R}^{-1})$. The noises sampled from this distribution are displayed in Figure 29a. By structuring the covariance matrix in this fashion, the sampled noises depends on both the previous and next value. Moreover, this structure also penalize large accelerations and keeps the noises added to the first and last weights considerably small. Although the sampled noises follows the defined characteristics, one main problem still remains – how to reduce the impact of the noise when the policy has converged? One possible solution, which was proposed in this thesis, was to add an optimization parameter β to the covariance matrix Σ as

$$\Sigma = \beta\mathbf{R}^{-1}. \tag{39}$$

As the parameter β increases the amount of added noise decreases. Hence, the magnitude of the added noise is only dependant on one parameter, namely β. Fine-tuning this parameter differs from application to application. In this thesis it proved to be favourable to set the magnitude of the β parameter as

$$\beta = \exp \sum_{i=0}^{N} r_i^2, \tag{40}$$

where r_i belongs to the i-th reward and the sum is taken over the N best rewards. In this way, the added noise is effectively reduced several order of magnitudes as β increases (Figure 29b).

Now as there exist a method for reducing the effect of the added noise as the policy converges, one problem still remains and that is initializing the variance of the covariance matrix. The initialization of the variance is an important factor as this regulates the magnitude of the initially sampled noise when the β parameter is kept low which is the case when the learning starts. The sampled noises, which are to be added to the weights, needs to be considerably large from time to time to actually

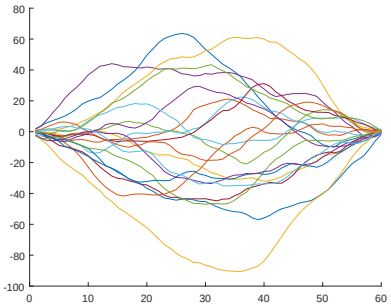
change the resulting trajectory, otherwise the policy will either never converge or only converge after a large number of trials, which is not an effective strategy. On the other hand, the sampled noise cannot be too large either as this might result in large deviations in the trajectory and subsequently in large accelerations which the robot cannot physically execute.

Intuitively, with all this in mind, the initialization of the variance for every covariance matrix should be bound, in some sense, to the weights of each motor primitive. This reasoning resulted in the complete covariance matrix

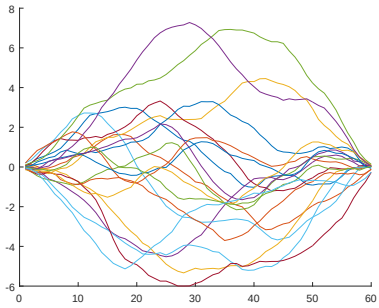
$$\Sigma_i = \gamma_i \beta \mathrm{R}^{-1}, \tag{41}$$

where γ_i is the initialization parameter. The magnitude of the γ_i parameter differs from application to application. In this thesis a proper value was found through trial and error and were 1.4 times the inverse of the standard deviation of the weights in y-direction, and 0.6 times the inverse of the standard deviation of the weights in z-direction.

As previously mentioned, these noises are also sampled from a multivariate zero mean Gaussian distribution with the covariance matrix Σ_i, i.e. $\mathcal{N}(0, \Sigma_i)$. The sampled noises corresponding to Equations (38) and (41) are displayed, respectively, in Figures 29a and 30. In the latter figure the sampled noises do indeed fulfil all the characteristics which were previously defined. Possible trajectories which can be governed by this form of sampled noise are visualized in Figure 31. It can clearly be seen that the resulting trajectories are smooth, i.e. no sudden deviations in position, and the initial and final positions do not differ much from the demonstrated trajectory. With all this in mind, the trajectories in Figure 31 shows that sampling the noises according to Equation (41) results in trajectories which are feasible for the robot to execute. Thus, this was implemented and executed on the actual hardware and the acquired results are presented in the following section.



(a) The figure resembles 20 samples drawn from a zero mean Gaussian distribution with the covariance matrix Σ. It can clearly be seen how correlated the noises are.



(b) The figure resembles 20 samples drawn from a zero mean gaussian distribution with the covariance matrix $\beta\Sigma$ where $\beta = 100$. In comparison to Figure 29a the added noise is almost 10 times smaller, thus an increase of β reduces the added noise.

Figure 29: Figures (a) and (b) both depict correlated noises which has been sampled from a multivariate Gaussian distribution. However, they differ with respect to the β parameter.

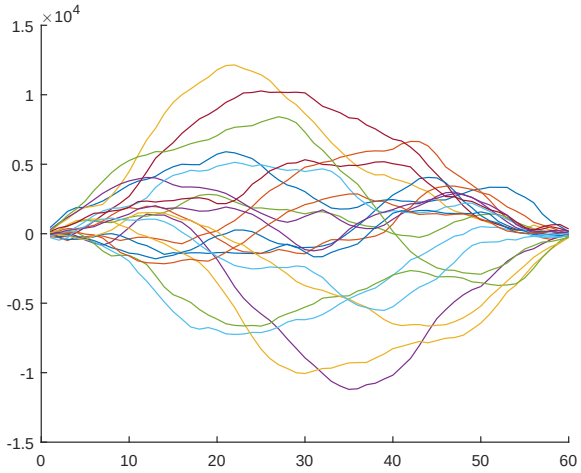


Figure 30: The figure resembles 20 samples drawn from a zero mean Gaussian distribution with the covariance matrix $\beta\Sigma_i$ where $\beta = 1$ and Σ_i is initialized according to the weights of motor primitive in y-direction. The noises are several magnitudes larger than the original in Figure 29a. This noise corresponds in larger extent to the size of the weights of each motor primitive, and by increasing the β parameter the effect of the noises washes out as can be seen in 29b.

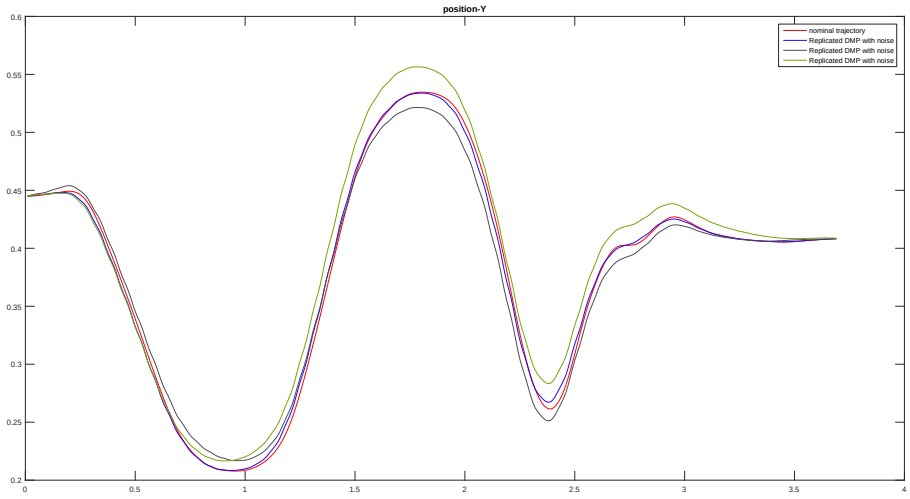


Figure 31: The red trajectory is the original demonstrated one. The other trajectories are simulated with correlated noise added to each parameter, where $\beta = 1$ and Σ_i is initialized for the motor primitive in y-direction.

6.3 Imitation Learning for the Ball-in-a-cup Game

In this section, experiments conducted on the actual hardware are presented. Based on the previous results, the generated noise, which are added to the weights of the original DMP, was sampled from a multivariate zero mean Gaussian distribution with a covariance matrix as in Equation (41). As previously established the number of kernels was 55 and the importance sampling for calculating the new weights used the 8 best previous rollouts.

Only one experiment was conducted and it examined if a robot can learn how to play the ball-in-a-cup game by imitation learning with subsequent reinforcement learning. It was split into two parts, case A and case B, where case A used eleven initial rollouts before starting to reweigh the DMP and case B used zero. By comparing the number of initial rollouts the following characteristics are compared: does the number of initial rollouts affect how often the robot learns the optimal policy as well as the convergence rate, *i.e.* how fast an optimal policy can be found? To be able to answer these questions reliably, fourteen trials for each case were conducted. The policy was said to converge if the ball went into the cup five times in a row. On the other hand, the robot was said to be unable to learn for one trial, if in ten consecutive rollouts not even one made it up to the top eight rollouts, which in this case was the number used in the importance sampling, or if the learned policy threw the ball either in front or behind the cup.

6.3.1 Imitation Learning with Varying Initial Rollouts

As the ball-in-a-cup game is a complex skill, the learned DMP is not able to exactly reproduce the demonstrated trajectory, indicated in Figure 32, which, in turn, leads

to the ball not entering the cup, indicated in Figure 33. Thus, for learning a representation of the taught skill which can generate a policy where the ball goes into the cup, subsequent reinforcement learning is needed. The performance of the learning algorithm for two different cases, case A with 11 initial rollouts and case B with no initial rollouts, is indicated in Figure 34a and 34b. These results are also listed in Table 2. These results are slightly worse than the ones presented in [50] where it took on average about 100 rollouts to fine tune the 231 shape parameters used. However, the main difference between the results is when considering how many trials it took for the optimal policy to be found after the ball went into the cup for the first time. In [50] they mention that it on average got the ball into the cup after 43 rollouts but it took an additional of about 57 trials before the policy converged to the optimal one. In comparison, it took, on average, case A 11 trials and B 14 trials to find the optimal policy after the ball went in for the first time.

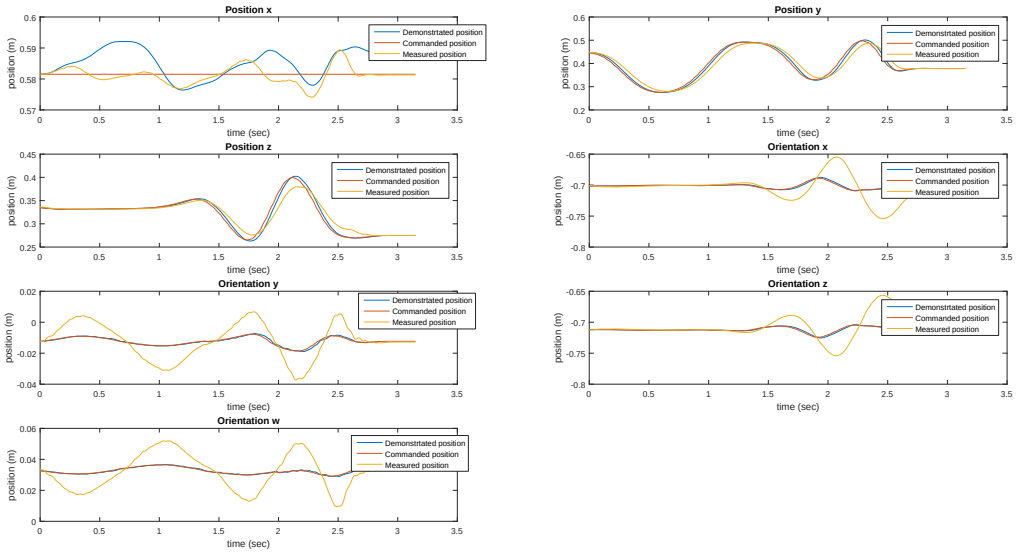


Figure 32: The different plots depicts the demonstrated, desired and commanded position and orientation of the end tool.

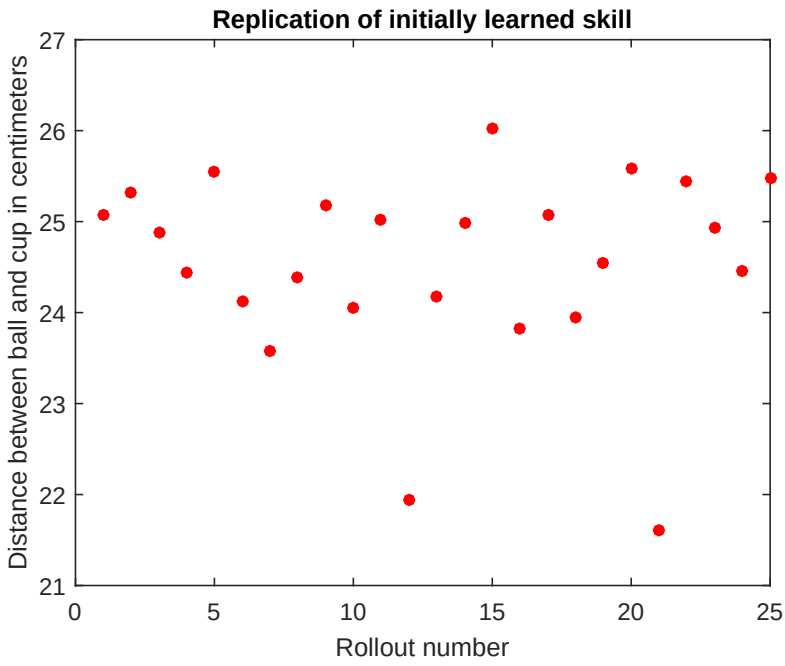
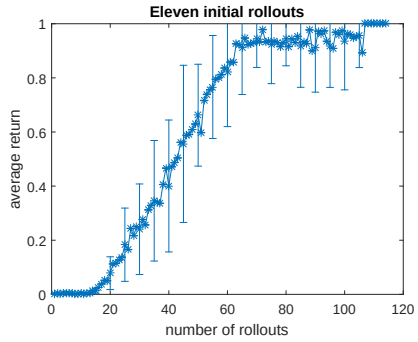
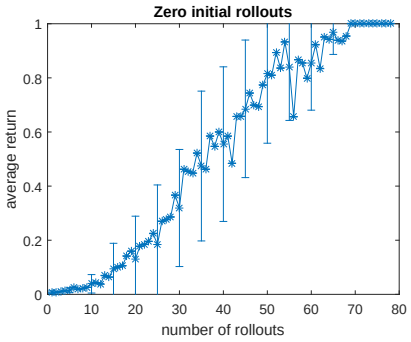


Figure 33: This figure shows the distances between the cup and the ball when the skill is only replicated by the DMP. The mean distance from these 20 rollouts is 24.5 cm.



(a) This figure shows the expected reward averaged over twelve successful trials for the policy learned in case A.



(b) This figure shows the expected reward averaged over five successful trials for the policy learned in case B.

Figure 34: Figure (a) and (b) both depicts the expected reward averaged over their individual number of successful trials. Rollouts when the vision system failed are discarded. Noteworthy is that the standard deviation is only plotted for every third rollout.

Table 2: The performance of the learning algorithm for the two cases

Case	Initial rollouts	Number of successfully learned policies out of totally 14 trials each	Average number of rollouts before converging	Average number of rollouts after ball went into the cup for the first time
A	11	12	79.9	11
B	0	6	62.8	14

Based on the results presented in Table 2 both case A and B differs to some extent when considering the number of successfully learned policies and their individual convergence rate. 14 trials were conducted in each case, and in case A the number of successful trials was 12, whereas the same results for case B was 6. The mean of the convergence rate, on the other hand, was 79.9 in case A and 62.8 in case B. Based on these numbers, there is a need for a statistical analysis to be able to tell if there exist a statistical difference between the two cases.

However, to do the statistical analysis one has to differentiate between a trial where the robot did not learn the optimal policy at all and one where it learned a suboptimal one, namely where the ball got thrown either in front or behind the cup. The suboptimal policy is actually an optimal policy considering the reward function defined in this thesis; however they are still discarded as the actual meaning of the game is to throw the ball into the cup and not in front or behind it. These restriction reduces the overall number of unsuccessful trials to only one in case A and seven in case B.

With these simplifications, two null and alternative hypotheses were examined. The first null and alternative hypotheses considered the number of successfully learned policies and was stated as

H_0: There are no differences in the mean number of successfully learned policies when using eleven initial rollouts to using zero.

H_a: The mean number of successfully learned policies is higher when using eleven initial rollouts than using zero.

While the second null and alternative hypotheses considered the convergence rate of the successfully learned policies and was stated as

H_0: There are no differences in the mean convergence rate when using eleven initial rollouts to using zero.

H_a: The mean convergence rate is higher when using eleven initial rollouts to using zero.

The results from the statistical comparison can be seen in Table 3. Based on the p-value listed for $\mu_{A,0}<\mu_{B,0}$ (0.004647) the first null hypothesis, stating that there are no difference in the mean number of successfully learned policies when using eleven initial rollouts to zero, can be rejected. Thus, the alternative hypothesis, stating that

using eleven initial rollouts will increase the number of successfully learned policies than using zero, holds. On the other hand, the p-value listed for $\mu_{A,1}<\mu_{B,1}$ (0.0964) is not low enough for rejecting the null hypothesis. All in all, the probability of successfully learning policies when 11 initial rollouts were used is higher than when 0 initial trials were used, whereas nothing can be said about the convergence rate.

Table 3: The summary of the results from the statistical analysis for learning to play the ball-in-a-cup game with different number of initial rollouts

Case	mean and standard deviation of the number of successfully learned policies	mean and standard deviation of the number of rollouts before converging	p-value for $\mu_{A,0}<\mu_{B,0}$ (significance =1%)	p-value for $\mu_{A,1}<\mu_{B,1}$ (significance =1%)
A	$\mu_{A,0} = 0.92$ $\sigma_{A,0} = 0.23$	$\mu_{A,1} = 79.9$ $\sigma_{A,1} = 20.5$	0.004647	0.0964
B	$\mu_{B,0} = 0.46$ $\sigma_{B,0} = 0.52$	$\mu_{B,1} = 62.8$ $\sigma_{B,1} = 10.6$		

Next an example of the demonstrated, learned and replicated trajectories along with the sampled noise from one successful trial in case A is presented in Figures 35 and 36 respectively. In Figure 35 it can be seen that the trajectories where the learning actually tunes the shape parameters, namely in y- and z-direction, changes in a smooth fashion, exactly as intended with the chosen sampling method. It is worth pointing out the error between the commanded position in x-direction and all the orientations and the actual measured positions. This relatively large error clearly indicated the limitations regarding the position accuracy of the Cartesian impedance controller presented in Section 5.3. Figure 36, on the other hand, shows that the sampled noises are correlated and that the initial and final magnitude of the sampled noise is low, which is all in line with the characteristics set up by the covariance matrix in Equation (38). Furthermore, the influence of the β parameter can also be depicted, where a high value effectively lowers the magnitude of the noise.

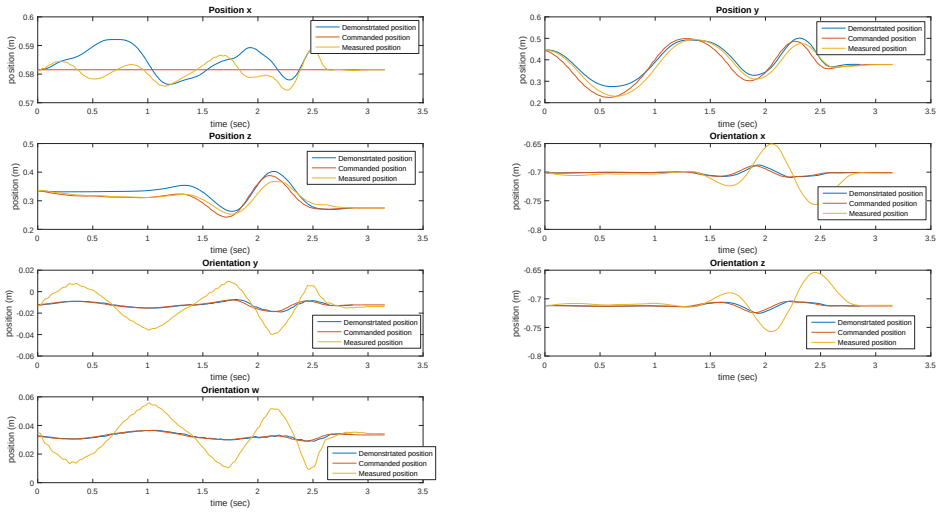


Figure 35: The different plots depicts the position and orientation of the end tool after a specific amount of trials. Worth pointing out is that the same states are replicated for all the orientations and all positions except in y- and z-direction where noises are applied to the weights. The final trajectories for these directions is clearly different from the original ones, thus the reinforcement learning does indeed change the overall shape of the trajectories.

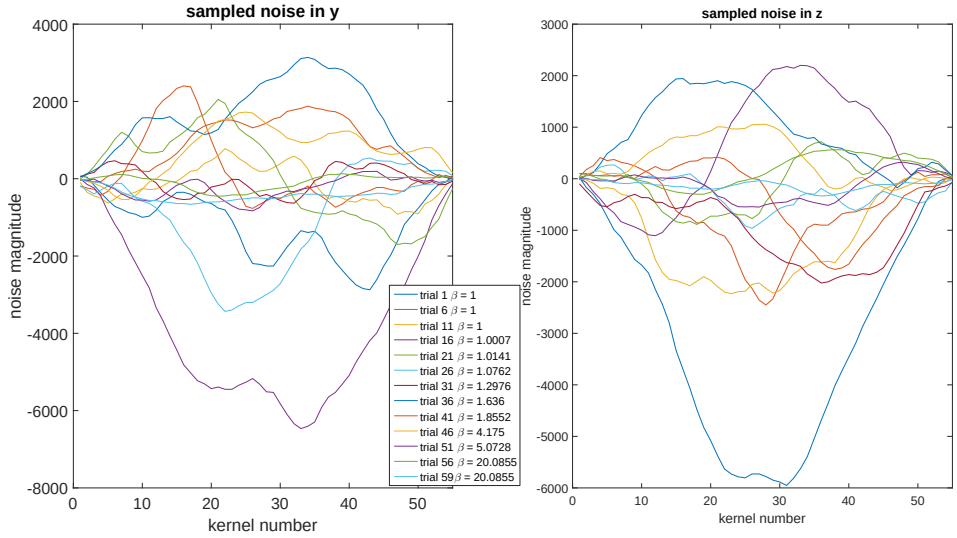


Figure 36: This figure depicts the sampled noises in both y- and z-direction after 1, 11, 21 and 31 rollouts. It can clearly be seen that the noises, sampled from $\mathcal{N}(0, \mathbf{R}^{-1})$, is correlated and low at the start and end of the trajectory. Furthermore, the influence of the optimization parameter β is also recognizable.

6.4 Discussion

This Chapter consisted of both a simulation (Section 6.2) and a real-life experiment (Section 6.3). The simulations were mainly fulfilled as to guide the implementation on the real hardware by pointing out potential errors and to find an appropriate method for selecting the covariance matrix from which the noises were sampled. From the simulations the covariance matrix was selected as a differencing matrix as this produced correlated noise with low magnitude of the initial and final values. The problem with this approach was the initialization of the covariance matrix. In this thesis the initialization of the covariance matrix was set individually for every motor primitive which was to be disturbed. This approach took a considerable amount of time as it was basically tuned through trial and error. In comparison to the initialization, the approach for diminishing the influence of the noises as the policy converged to an optimal one was easy to develop and worked considerably well.

To study if the covariance matrix used in the simulations worked well also in real-life, one experiment on a KUKA LWR was conducted. This experiment proved that to successfully play the ball-in-a-cup game the initially produced policy from the learned DMP was not enough. Thus, to successfully learn the skill the shape parameters of the DMP were fine-tuned with the RL algorithm PoWER. The results indicated that this RL algorithm was able to fine-tune the parameters which enabled the robot to successfully learn the movement and get the ball into the cup. When considering the overall convergence rate, the result from both case A and B were slightly worse than the results presented in [50]. The difference in convergence rate might originate from a lower exploration rate compared to [50]. The exploration rate can be increased by increasing the value of the initialization parameter γ which in turn will produce larger values for the sampled noise. This, On the other hand, can also affect how often the robot will be able to learn the optimal policy as large exploration can lead to more policies which never converges or are infeasible for the robot to physically execute. Thus, improving convergence rate can impair the success rate for learning the optimal policy and impairing the convergence rate can have the opposite affect.

However, in contrast to the lower convergence rate, the obtained results in comparison to [50] indicated a faster convergence rate once the ball went into the cup for the first time. Here it only took about 11-14 rollouts for the policy to converge once the ball went in while in [50] it took up to 30 trials. This improvement probably originate from the exploration rate. As the sampled noises are highly correlated new policies are explored in a safe manner and does not differ in such a large extent from previous trajectories. Moreover, the impact of the noise diminishes as the optimal policy converges. Thus, as the ball starts to get close to the cup, for example hitting the side of it, the magnitude of the added noise is considerably small as only small perturbations of the shape parameters are sufficient to finally get the ball into the cup. Therefore, the PoWER algorithm reweighs the parameters based on rollouts which are slowly but safely improving towards the optimal policy.

Furthermore, the statistical analysis of cases A and B indicated that using more initial rollouts increased the probablility for the robot to learn an optimal policy.

This result makes sense as the system is stochastic, some rollouts will always be considered bad. If the reweighing then immediately starts from such a bad rollout the shape parameters will already be affected by this rollout, and subsequently the learning algorithm can have problems overcoming this reweighing and turn the learning back towards a better policy. This problem, however, is alleviated by doing several initial rollouts and then reweighing only the top N of these rollouts. In this way, bad rollouts are washed out and only good ones are used for the initial reweighing. Another interesting fact from the statistical analysis is that even though case A basically used 11 rollouts before reweighing the shape parameters, the average number of rollouts before converging were not statistically different from doing zero initial trials. Thus, the convergence rate also seems to be improved by doing several initial trials and only using the top N for reweighing. This is probably a result of reweighing several good rollouts at once, which will change the shape parameters more than only using one rollout.

Finally, the experiment also proved the inaccuracy of the Cartesian impedance controller. Although the stiffness was set to the maximum allowed value, the controller was unable to exactly replicate the desired position. This comes from the fact that the controller replicates a spring/damper system and no matter how high the stiffness is set, the system will always produce an error.

All in all, both the simulation and the experiment proved that DMP with subsequent reinforcement learning, where the noises are sampled from a multivariate zero mean Gaussian distribution with the covariance matrix initialized as a differencing matrix, was able to successfully learn how to play the ball-in-a-cup game. As this game can be seen as a benchmark problem in robotics, the approach used in this thesis, where exploration of new trajectories is considerably safe, can most probably also work for other tasks where reinforcement learning is needed or has been used before, for instance as in the pancake flipping task [52]. However, this solution also has its drawbacks. First of all, the covariance matrix has to be initialized in such a way that it produces large enough noises for the policy to converge in a reasonable amount of rollouts. Moreover, the robot did, in some trials, not learn the optimal policy. This number, however, can probably either be eliminated or at least be made smaller if the initialization of the covariance matrix is fine-tuned. The fine-tuning of the β parameter, on the other hand, worked well. However, fine-tuning it as is done in this thesis, exponentially increasing as the rewards gets higher, is probably not applicable for all applications.

7 Conclusions

The goal of this thesis was to study how a robot can acquire a learned skill from a human teacher and subsequently improve this skill by reinforcement learning. However, as robots also have their physical limitations, the reinforcement learning must follow a safe path as to ensure safety for both the robot and the user. With all this in mind, the thesis sought to answer the following question:

Q1. How can the exploration rate be made safe for exploring new trajectories as to improve the current policy?

Answering this question is a step towards showing that robots are able to learn in the same manner as humans, something which is of great significance if robots are to be integrated safely into a human environment.

This thesis was split into a theoretical part consisting of three chapters *Learning From Demonstrations*, *Dynamic Movement Primitives* and *Reinforcement Learning* and a more practical part consisting of the two chapters *Testbed* and *Experiments and Results*.

The empirical findings in the first three chapters were crucial to argue for a solution to Q1. First of all, the skill had to be demonstrated with a sensible approach and subsequently the learned representation by the robot had to be very smooth. In the *Learning from Demonstration* chapter kinestethic teaching was chosen as the teaching method as this effectively avoided the correspondence problem and was easy to execute on the KUKA LWR. For learning a representation of the skill DMP was chosen as this produced even smoother representations than the actual demonstrations.

However, only reproducing the learned movement encoded as a DMP was not enough to successfully bring the ball into the cup. Thus, reinforcement learning had to be incorporated as to improve the initially learned skill. As discussed in the *Reinforcement Learning* chapter, classical RL approaches were not applicable to learn the skill as the search space was infinite. To alleviate this shortcoming, reinforcement learning was applied to an already learned skill encoded as a DMP, thus reducing the search space effectively. The resulting state-of-the art policy search algorithm, which was also implemented, was the PoWER algorithm.

For both testing the overall performance of the PoWER algorithm and finding a feasible approach for selecting the exploration rate (subsequently answering Q1) everything was tested on a KUKA LWR4+ robotic arm. The approach of selecting a covariance matrix as a differentiating matrix resulted in correlated noise with low initial and final values. By utilizing this noise, an optimal policy could be found in a safe manner. Furthermore by adding another scaling parameter which tuned the influence of the added noise, the policy converged to the optimal one after about 11 rollouts after the ball went into the cup for the first.

The implemented system in this thesis also had its limitations. Although the learned movement should theoretically only be executed in a plane, it did indeed happen that the learned policy by the robot reproduced a movement where the ball was thrown either in front of or behind the cup. However, as the reward function

was based on the horizontal distance in y-direction between the ball and the cup, the learned policy was actually an optimal one. Hence, the robot did indeed optimize the reward, but not learn the skill as intended. Another limitation of the actual system was the visual tracking of the ball as it often malfunctions due to not being able to track the red ball at all. With all this in mind, future work to improve the system built in this thesis should concentrated on:

- allowing the skill to be learned in free space, thus also learning a DMP in x-direction as well as reformulating the reward function to take into consideration the distance from the ball to the cup in both x- and y-direction;
- improving the visual tracking of the ball by either using a camera with a higher frame rate, or by using a Kalman filter on the ball to ease the tracking of it.

Other possibilities for future work, which do not include the limitations of the implemented system, are to:

- learn an initial representation of a complex skill and then generalize this to a new situation;
- learn an in-contact task which is subsequently improved with RL;
- incorporate apprenticeship learning (inverse reinforcement learning) instead of classical RL.

The generalization of learned movements has previously been studied; however, there is still a gap between generalization and subsequent reinforcement learning. In the scope of this thesis, a new situation would be a longer or shorter string which would affect the dynamics of the system. Nevertheless, if several representations of the skill have been learned for varying situations, these might be combined as to provide an initially good guess of how to perform the skill and then be subsequently improved by reinforcement learning.

For in-contact tasks the problem is to also learn a representation for the forces. Although modelling such skills have been studied, it has not yet been combined to much extent with reinforcement learning. In these kind of applications, for instance wood planing, the magnitude of the applied force will affect the overall performance of the system and should be explored as to find a good representation of the movement.

For apprenticeship learning a reward function does not exist, but has to be learned. The problem with reinforcement learning is that the user has to provide the reward function, hence the performance of the system is dependent on this reward function. However, in situations such as driving, an overall working reward function does not exists and for robots to learn how to drive they also needs to learn this reward function.

All in all, this thesis showed that a robot can learn a complex task, such as playing the ball-in-a-cup game, by learning an initial representation of the movement from a human demonstration and then subsequently improving the learned skill until mastering it. If, or when, also the proposed future work can be solved, the whole

framework of teaching a robot a new skill simply by demonstrating it will drastically reduce development costs for robots which, in turn, might lead to more robots in human environments.

References

[1] Bilboquet: Cup and ball or ring and pin games. http://www.gamesmuseum.uwaterloo.ca/VirtualExhibits/bilboquet/pages/. Accessed: 22- Jun-2016.

[2] Alin Albu-Schäffer, Sami Haddadin, Christian Ott, Andreas Stemmer, Thomas Wimböck, and Gerd Hirzinger. The dlr lightweight robot: design and control concepts for robots in human environments. *Industrial Robot: an international journal*, 34(5):376–385, 2007.

[3] Jacopo Aleotti and Stefano Caselli. Robust trajectory learning and approximation for robot programming by demonstration. *Robotics and Autonomous Systems*, 54(5):409–413, 2006.

[4] Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I Jordan. An introduction to mcmc for machine learning. *Machine learning*, 50(1-2):5–43, 2003.

[5] Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483, 2009.

[6] Christopher G Atkeson, Andrew W Moore, and Stefan Schaal. Locally weighted learning for control. In *Lazy learning*, pages 75–113. Springer, 1997.

[7] J Andrew Bagnell, Sham M Kakade, Jeff G Schneider, and Andrew Y Ng. Policy search by dynamic programming. In *Advances in neural information processing systems*, page None, 2003.

[8] J Andrew Bagnell and Jeff Schneider. Covariant policy search. IJCAI, 2003.

[9] Richard Bellman. *Dynamic Programming*. Dover Books on Computer Science Series. Dover Publications, 2003.

[10] Aude Billard, Sylvain Calinon, Rüdiger Dillmann, and Stefan Schaal. *Robot Programming by Demonstration*, pages 1371–1394. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[11] Aude Billard, Sylvain Calinon, Ruediger Dillmann, and Stefan Schaal. Robot programming by demonstration. In *Springer handbook of robotics*, chapter 59, pages 1371–1394. Springer, 2008.

[12] Rainer Bischoff, Johannes Kurth, Günter Schreiber, Ralf Koeppe, Alin Albu-Schäffer, Alexander Beyer, Oliver Eiberger, Sami Haddadin, Andreas Stemmer, Gerhard Grunwald, et al. The kuka-dlr lightweight robot arm-a new reference platform for robotics research and manufacturing. In *Robotics (ISR), 2010 41st international symposium on and 2010 6th German conference on robotics (ROBOTIK)*, pages 1–8. VDE, 2010.

[13] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[14] Brett Browning, Ling Xu, and Manuela Veloso. Skill acquisition and use for a dynamically-balancing soccer robot. In *AAAI*, pages 599–604, 2004.

[15] Herman Bruyninckx. Open robot control software: the orocos project. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 3, pages 2523–2528. IEEE, 2001.

[16] Arthur Earl Bryson. *Applied optimal control: optimization, estimation and control*. CRC Press, 1975.

[17] Daniel Bullock and Stephen Grossberg. Vite and flete: Neural modules for trajectory formation and postural control. *Volitional action*, pages 253–298, 1989.

[18] Kenneth P Burnham and David R Anderson. *Model selection and multimodel inference: a practical information-theoretic approach*. Springer Science & Business Media, 2003.

[19] Sylvain Calinon and Aude Billard. Statistical learning by imitation of competing constraints in joint space and task space. *Advanced Robotics*, 23(15):2059–2076, 2009.

[20] Sylvain Calinon, Florent D'halluin, Eric L Sauser, Darwin G Caldwell, and Aude G Billard. Learning and reproduction of gestures by imitation. *Robotics & Automation Magazine, IEEE*, 17(2):44–54, 2010.

[21] Sylvain Calinon, Florent D'halluin, Eric L Sauser, Darwin G Caldwell, and Aude G Billard. Learning and reproduction of gestures by imitation. *Robotics & Automation Magazine, IEEE*, 17(2):44–54, 2010.

[22] Sylvain Calinon, Florent Guenter, and Aude Billard. On learning, representing, and generalizing a task in a humanoid robot. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 37(2):286–298, 2007.

[23] Sylvain Calinon, Florent Guenter, and Aude Billard. On learning, representing, and generalizing a task in a humanoid robot. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 37(2):286–298, 2007.

[24] Jason Chen and Alex Zelinsky. Programing by demonstration: Coping with suboptimal teaching actions. *The International Journal of Robotics Research*, 22(5):299–319, 2003.

[25] David A Cohn, Zoubin Ghahramani, and Michael I Jordan. Active learning with statistical models. *Journal of artificial intelligence research*, 1996.

[26] Steve Cousins. Exponential growth of ros [ros topics]. *IEEE Robotics & Automation Magazine*, 1(18):19–20, 2011.

[27] Peter Dayan and Geoffrey E Hinton. Using expectation-maximization for reinforcement learning. *Neural Computation*, 9(2):271–278, 1997.

[28] Staffan Ekvall and Danica Kragic. Learning task models from multiple human demonstrations. In *RO-MAN 2006 - The 15th IEEE International Symposium on Robot and Human Interactive Communication*, pages 358–363. Affiliation: Computational Vision and Active Perception, Centre for Autonomous Systems, Royal Institute of Technology, Stockholm, Sweden; Correspondence Address: Kragic, D.; Computational Vision and Active Perception, Centre for Autonomous Systems, Royal Institute of Technology, Stockholm, Sweden; email: kragic@nada.kth.se, 6 September 2006 through 8 September 2006 2006.

[29] Tamar Flash and Binyamin Hochner. Motor primitives in vertebrates and invertebrates. *Current opinion in neurobiology*, 15(6):660–666, 2005.

[30] Wendell H Fleming and Halil Mete Soner. *Controlled Markov processes and viscosity solutions*, volume 25. Springer Science & Business Media, 2006.

[31] Zoubin Ghahramani. An introduction to hidden markov models and bayesian networks. *International Journal of Pattern Recognition and Artificial Intelligence*, 15(01):9–42, 2001.

[32] Simon F Giszter, Ferdinando A Mussa-Ivaldi, and Emilio Bizzi. Convergent force fields organized in the frog's spinal cord. *Journal of Neuroscience*, 13(2):467–491, 1993.

[33] KUKA Laboratories GmbH. *KUKA System Software 5.6 lr: Operating and Programming Instrictuions for System Integrators*. KSS 5.6 lr SI v5 en edition.

[34] Vijaykumar Gullapalli, Judy A Franklin, and Hamid Benbrahim. Acquiring robot skills via reinforcement learning. *Control Systems, IEEE*, 14(1):13–24, 1994.

[35] Neville Hogan. Adaptive control of mechanical impedance by coactivation of antagonist muscles. *Automatic Control, IEEE Transactions on*, 29(8):681–690, 1984.

[36] Jung-Hoon Hwang, Ronald C Arkin, and Dong-Soo Kwon. Mobile robots at your fingertip: Bezier curve on-line trajectory generation for supervisory control. In *Intelligent Robots and Systems, 2003.(IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 2, pages 1444–1449. IEEE, 2003.

[37] Auke Ijspeert, Jun Nakanishi, and Stefan Schaal. Learning control policies for movement imitation and movement recognition. In *Neural information processing system*, volume 15, pages 1547–1554, 2003.

[38] Auke Jan Ijspeert, Jun Nakanishi, Heiko Hoffmann, and Stefan Pastor, Schaal. Dynamical movement primitives: learning attractor models for motor behaviors. *Neural computation*, 25(2):328–373, 2013.

[39] Auke Jan Ijspeert, Jun Nakanishi, and Stefan Schaal. Learning attractor landscapes for learning motor primitives. Technical report, 2002.

[40] Auke Jan Ijspeert, Jun Nakanishi, and Stefan Schaal. Movement imitation with nonlinear dynamical systems in humanoid robots. In *2002 IEEE International Conference on Robotics and Automation*, volume 2, pages 1398–1403. Affiliation: University of Southern California, Los Angeles, CA 90089-2520, United States; Correspondence Address: Ijspeert, A.J.; University of Southern California, Los Angeles, CA 90089-2520, United States; email: ijspeert@usc.edu, 11 May 2002 through 15 May 2002 2002.

[41] Rick L Jenison and Kate Fissell. A comparison of the von mises and gaussian basis functions for approximating spherical acoustic scatter. *Neural Networks, IEEE Transactions on*, 6(5):1284–1287, 1995.

[42] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, pages 237–285, 1996.

[43] Mrinal Kalakrishnan, Sachin Chitta, Evangelos Theodorou, Peter Pastor, and Stefan Schaal. Stomp: Stochastic trajectory optimization for motion planning. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 4569–4574. IEEE, 2011.

[44] Mrinal Kalakrishnan, Ludovic Righetti, Peter Pastor, and Stefan Schaal. Learning force control policies for compliant manipulation. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4639–4644. IEEE, 2011.

[45] Hilbert J Kappen. Path integrals and symmetry breaking for optimal control theory. *Journal of statistical mechanics: theory and experiment*, 2005(11):P11011, 2005.

[46] S Mohammad Khansari-Zadeh and Aude Billard. Learning stable nonlinear dynamical systems with gaussian mixture models. *IEEE Transactions on Robotics*, 27(5):943–957, 2011.

[47] S Mohammad Khansari-Zadeh and Aude Billard. Learning stable nonlinear dynamical systems with gaussian mixture models. *Robotics, IEEE Transactions on*, 27(5):943–957, 2011.

[48] J. Kober and J. Peters. Policy search for motor primitives in robotics. *Machine Learning*, 84(1-2):171–203, 2011.

[49] Jens Kober, Katharina Mülling, Oliver Krömer, Christoph H Lampert, Bernhard Schölkopf, and Jan Peters. Movement templates for learning of hitting and batting. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 853–858. IEEE, 2010.

[50] Jens Kober and Jan R Peters. Policy search for motor primitives in robotics. In *Advances in neural information processing systems*, pages 849–856, 2009.

[51] Jens Kober and Jan R Peters. Policy search for motor primitives in robotics. In *Advances in neural information processing systems*, pages 849–856, 2009.

[52] Petar Kormushev, Sylvain Calinon, and Darwin G Caldwell. Robot motor skill coordination with em-based reinforcement learning. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 3232–3237. IEEE, 2010.

[53] Petar Kormushev, Sylvain Calinon, and Darwin G Caldwell. Imitation learning of positional and force skills demonstrated via kinesthetic teaching and haptic input. *Advanced Robotics*, 25(5):581–603, 2011.

[54] Petar Kormushev, Sylvain Calinon, and Darwin G Caldwell. Reinforcement learning in robotics: Applications and real-world challenges. *Robotics*, 2(3):122–148, 2013.

[55] Michel A Lemay and Warren M Grill. Endpoint force patterns evoked by intraspinal stimulation - ipsilateral and contralateral responses. In Enderle J.D, editor, *22nd Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, volume 2, pages 918–920, 2000.

[56] Perry Y Li and Roberto Horowitz. Passive velocity field control of mechanical manipulators. *Robotics and Automation, IEEE Transactions on*, 15(4):751–763, 1999.

[57] Maja J Mataric. Sensory-motor primitives as a basis for imitation: Linking perception to action and biology to robotics. In *Imitation in Animals and Artifacts*. Citeseer, 2000.

[58] Geoffrey McLachlan and Thriyambakam Krishnan. *The EM Algorithm and Extensions*. Wiley Series in Probability and Statistics. Wiley, 1996.

[59] Alberto Montebelli, Franz Steinmetz, and Ville Kyrki. On handing down our tools to robots: Single-phase kinesthetic teaching for dynamic in-contact tasks. In *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 5628–5634. IEEE, 2015.

[60] Manuel Mühlig, Michael Gienger, Sven Hellbach, Jochen J Steil, and Christian Goerick. Task-level imitation learning using variance-based movement optimization. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 1177–1184. IEEE, 2009.

[61] Katharina Mülling, Jens Kober, Oliver Kroemer, and Jan Peters. Learning to select and generalize striking movements in robot table tennis. *The International Journal of Robotics Research*, 32(3):263–279, 2013.

[62] Ferdinando A Mussa-Ivaldi. Nonlinear force fields: a distributed system of control primitives for representing and learning movements. In *Computational Intelligence in Robotics and Automation, 1997. CIRA'97., Proceedings., 1997 IEEE International Symposium on*, pages 84–90. IEEE, 1997.

[63] Jun Nakanishi, Jun Morimoto, Gen Endo, Gordon Cheng, Stefan Schaal, and Mitsuo Kawato. Learning from demonstration and adaptation of biped locomotion. *Robotics and Autonomous Systems*, 47(2):79–91, 2004.

[64] Chrystopher L Nehaniv and Kerstin Dautenhahn. *Imitation and social learning in robots, humans and animals: behavioural, social and communicative dimensions*. Cambridge University Press, 2007.

[65] Andrew Y Ng, Adam Coates, Mark Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger, and Eric Liang. Autonomous inverted helicopter flight via reinforcement learning. In *Experimental Robotics IX*, pages 363–372. Springer, 2006.

[66] Duy Nguyen-Tuong and Jan Peters. Local gaussian process regression for real-time model-based robot control. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 380–385. IEEE, 2008.

[67] Jason M O'Kane. A gentle introduction to ros, 2014.

[68] Erhan Oztop, Mitsuo Kawato, and Michael Arbib. Mirror neurons and imitation: A computationally guided review. *Neural Networks*, 19(3):254–271, 2006.

[69] Erhan Oztop, Mitsuo Kawato, and Michael A Arbib. Mirror neurons: functions, mechanisms and models. *Neuroscience letters*, 540:43–55, 2013.

[70] Sooho Park, Shabbir Kurbanhusen Mustafa, and Kenji Shimada. Learning-based robot control with localized sparse online gaussian process. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 1202–1207. IEEE, 2013.

[71] Jan Peters and Stefan Schaal. Policy gradient methods for robotics. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 2219–2225. IEEE, 2006.

[72] Jan Peters and Stefan Schaal. Reinforcement learning by reward-weighted regression for operational space control. In *Proceedings of the 24th international conference on Machine learning*, pages 745–750. ACM, 2007.

[73] Jan Peters, Sethu Vijayakumar, and Stefan Schaal. Reinforcement learning for humanoid robotics. In *Proceedings of the third IEEE-RAS international conference on humanoid robots*, pages 1–20, 2003.

[74] Polly K Pook and Dana H Ballard. Recognizing teleoperated manipulations. In *Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on*, pages 578–585. IEEE, 1993.

[75] Lawrence R Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[76] Carl Edward Rasmussen. Gaussian processes for machine learning. 2006.

[77] Nathan Ratliff, Matt Zucker, J Andrew Bagnell, and Siddhartha Srinivasa. Chomp: Gradient optimization techniques for efficient motion planning. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 489–494. IEEE, 2009.

[78] Thomas Rückstieß, Martin Felder, and Jürgen Schmidhuber. State-dependent exploration for policy gradient methods. In *Machine Learning and Knowledge Discovery in Databases*, pages 234–249. Springer, 2008.

[79] Takeshi Sakaguchi and Fumio Miyazaki. Dynamic manipulation of ball-in-cup game. In *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*, pages 2941–2948. IEEE, 1994.

[80] Shuichi Sato, Takeshi Sakaguchi, Yasuhiro Masutani, and Fumio Miyazaki. Mastering of a task with interaction between a robot and its environment('kendama' task). *Nippon Kikai Gakkai Ronbunshu, C Hen/Transactions of the Japan Society of Mechanical Engineers, Part C*, 59(558):179–185, 1993.

[81] Stefan Schaal and Christopher G Atkeson. Assessing the quality of learned local models. *Advances in neural information processing systems*, pages 160–160, 1994.

[82] Stefan Schaal, Peyman Mohajerian, and Auke Ijspeert. *Dynamics systems vs. optimal control - a unifying view*, volume 165. 2007.

[83] Stefan Schaal, Jan Peters, Jun Nakanishi, and Auke Ijspeert. *Learning movement primitives*, volume 15. 2005.

[84] Gregor Schöner and Cristina Santos. Control of movement time and sequential action through attractor dynamics: A simulation study demonstrating object interception and coordination. 2001.

[85] Günter Schreiber, Andreas Stemmer, and Rainer Bischoff. The fast research interface for the kuka lightweight robot. In *IEEE Workshop on Innovative Robot Control Architectures for Demanding (Research) Applications How to Modify*

and Enhance Commercial Controllers (ICRA 2010), pages 15–21. Citeseer, 2010.

[86] Peter Soetens. *The Orocos Component Builderś Manual*. The Orocos Project, 2.6.0 edition, 2012. Accessed: 22- Jun- 2016.

[87] Franz Steinmetz. Programming by demonstration for in-contact tasks using dynamic movement primitives.

[88] Robert F Stengel. *Optimal control and estimation*. Courier Corporation, 2012.

[89] Freek Stulp and Olivier Sigaud. Policy improvement methods: Between black-box optimization and episodic reinforcement learning. 2012.

[90] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 1998.

[91] Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, volume 99, pages 1057–1063, 1999.

[92] Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, volume 99, pages 1057–1063, 1999.

[93] Kazuki Takenanka. Dynamical control of manipulator with vision(" cup and ball" game demonstrated by robot). *TRANS. JAPAN SOC. MECH. ENG.*, 50(458):2046–2053, 1984.

[94] Evangelos Theodorou, Jonas Buchli, and Stefan Schaal. A generalized path integral control approach to reinforcement learning. *Journal of Machine Learning Research*, 11:3137–3181, 2010.

[95] Evangelos Theodorou, Jonas Buchli, and Stefan Schaal. Learning policy improvements with path integrals. In *International Conference on Artificial Intelligence and Statistics*, pages 828–835, 2010.

[96] Matthew C Tresch and Emilio Bizzi. Responses to spinal microstimulation in the chronically spinalized rat and their relationship to spinal systems activated by low threshold cutaneous stimulation. *Experimental Brain Research*, 129(3):401–416, 1999.

[97] Aleš Ude. Trajectory generation from noisy positions of object features for teaching robot paths. *Robotics and Autonomous Systems*, 11(2):113–127, 1993.

[98] Aleksandar Vakanski, Farrokh Janabi-Sharifi, and Iraj Mantegh. Robotic learning of manipulation tasks from visual perception using a kinect sensor. *International Journal of Machine Learning and Computing*, 4(2):163, 2014.

[99] Laurens Van Der Maaten, Eric Postma, and Jaap Van den Herik. Dimensionality reduction: a comparative. *J Mach Learn Res*, 10:66–71, 2009.

[100] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[101] Andrew D Wilson and Aaron F Bobick. Hidden markov models for modeling and recognizing gesture under variation. *International Journal of Pattern Recognition and Artificial Intelligence*, 15(01):123–160, 2001.