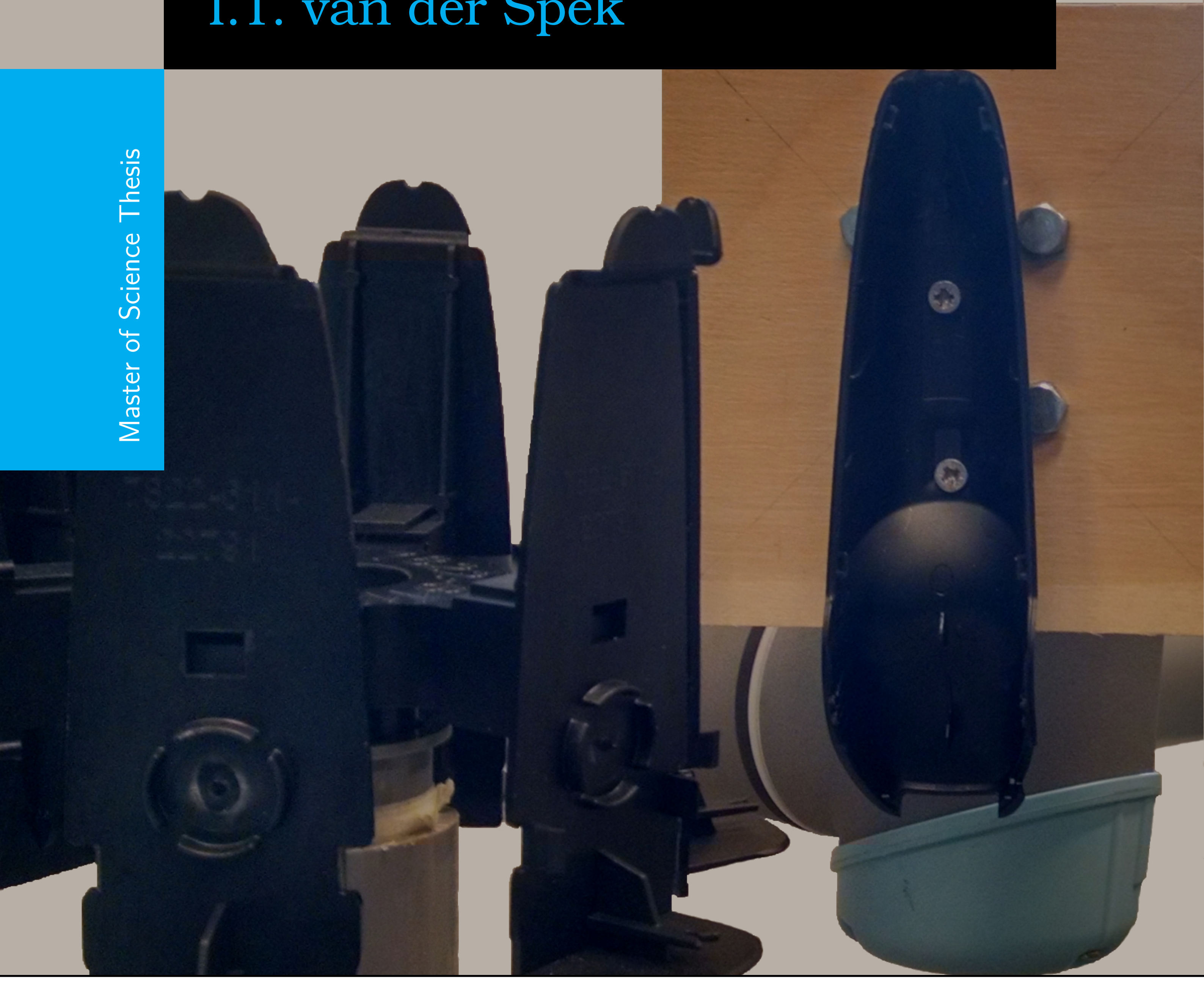
Imitation learning for a robotic precision placement task

I.T. van der Spek

Master of Science Thesis



Delft Center for Systems and Control

Imitation learning for a robotic precision placement task

Master of Science Thesis

For the degree of Master of Science in Embedded Systems at
Delft University of Technology

I.T. van der Spek

September 4, 2014

Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

ALTEN

The work in this thesis was performed at Alten Nederland

TU Delft Delft University of Technology

DCSC

Delft University of Technology
Department of
Delft Center for Systems and Control (dcsc)

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled

Imitation learning for a robotic precision placement task

by

I.T. van der Spek

in partial fulfillment of the requirements for the degree of

Master of Science in Embedded Systems

Dated: September 4, 2014

Supervisor(s):

prof.dr. R. Babuška

ing. J. Kuijpers MSc

Reader(s):

dr.ir. C.J.M. Verhoeven

dr.ir. S. Baldi

Abstract

In industrial environments robots are used for various tasks. At this moment it is not feasible for companies to deploy robots for productions with a limited batch size or for products with large variations. The use of robots for such environments can become feasible through a new generation of robots and software which can adapt quickly to new situations and learn from their mistakes while being programmable without needing an expert.

A concept that can enable the transition to flexible robotics is the combination of imitation learning and reinforcement learning. The purpose of imitation learning is to learn a task by generalizing from observations. The power of imitation learning is that the robot is programmed in an intuitive way while the insight of the teacher is incorporated in the execution of the task.

This research studies the combination of imitation and reinforcement learning, the research is applied to an industrial use-case. The research question of this study is: "Can imitation learning be combined with reinforcement learning to achieve a successful application in an industrial robotic precision placement task?"

To imitate the demonstrated trajectories, Dynamic Movement Primitives (DMPs) are used. The DMPs are used to encode the observed trajectories. DMPs can be seen as a spring-damper like system with a non-linear forcing term. The forcing term is a sum of Gaussian basis functions with each its corresponding weight. Reinforcement learning can be applied to these weights to alter the shape of the trajectory created by a DMP. Policy Gradients with Parameter based Exploration (PGPE) is used as reinforcement learning algorithm to optimize the recorded trajectories.

Experiments done on a UR5 show that without the learning step, the DMPs are able to provide a trajectory that results in a successful execution of a robotic precision placement task. The experiments also show that the learning algorithm is not able to remove noise from a demonstrated trajectory or complete a partial demonstrated trajectory. Therefore it can be concluded that the PGPE algorithm is not suited for reinforcement learning in robotics in its current form. It is therefore recommended to apply a data-efficient version of the PGPE algorithm in order to achieve better learning results.

Table of Contents

Preface

The master thesis in front of you is the written result of the research I performed for my master graduation project in the last nine months.

The idea of doing my thesis on this subject came after I was hired as a graduate student at Alten Nederland. The global idea of doing my thesis on ‘Programming by Example’ was proposed by my colleagues Heico Sandee and Simon Jansen. During my literature study and after some fruitful discussions with Simon and Jeroen, my daily supervisor, the project got the direction and shape it now has. After taking a tour trough the factory at Philips Consumer Lifestyle a suitable use-case was found regarding one of the processes in the factory. This all together gave enough challenge and material for a master graduation project.

For me, this was the first time working with (industrial) robot arms and the first real contact with the industry outside the academic world. I learned an incredible amount about robotics and the applications in industry. This made that my enthusiasm and interest for the field of robotics has grown strongly and is still growing.

I’m also very pleased that the abstract about the first findings of this research has been accepted at the DSPE Conference 2014. For the interested reader the extended abstract that will be published in the proceedings of the conference can be found in Appendix D. The abstract and associated talk are titled: ‘Flexible industrial robotics through imitation learning.’

I hope you will become as enthusiastic about robotics as I am while reading this thesis. Enjoy all the hard work I put in creating this document!

Acknowledgements

I would like to express my gratitude to my supervisor prof.dr. R. Babuška for his comments, ideas and guidance during the writing of this thesis.

Furthermore I would like to thank Alten Nederland for giving me the opportunity to do this graduation project. I really liked the freedom and trust that was given to me by the management. I especially would like to thank my daily supervisor Jeroen Kuijpers for all the advice and keeping me sharp and motivated through the whole process of writing this thesis. My thanks also goes to my colleagues Simon Jansen and Mark Geelen for their support in learning ROS and the fruitful discussions on all kinds of topics. I also like to thank all the colleagues and other graduates at the office for their support and the fun we made.

I thank Philips Consumer Lifestyle for providing me with a use-case for this graduation project. You have been very helpful in supplying the needed materials.

My deepest gratitude goes to my parents who gave me the opportunity to study and always kept me motivated during these five years of study.

Last but surely not least I would like to thank my sister and friends, you made that there was also something else apart from studying like love, fun and holidays.

I would like to end this acknowledgements with the words of the famous composer Johann Sebastian Bach: *"Soli Deo gloria"*.

Delft, University of Technology
September 4, 2014

I.T. van der Spek

“Only one who devotes himself to a cause with his whole strength and soul can be a true master. For this reason mastery demands all of a person.”

— *Albert Einstein*

“Only those who attempt the absurd will achieve the impossible. I think it’s in my basement... let me go upstairs and check.”

— *M.C. Escher*

Chapter 1

Introduction

In industrial environments robots are used for various tasks. At this moment it is not feasible for companies to deploy robots for productions with a limited batch size or for products with large variations. The current generation of robots is highly inflexible; therefore many repetitive tasks are still performed by humans. Current robots essentially have an insufficient level of intelligence: the robots are programmed at controller level and not at task level. Typically, a fixed sequence of actions is programmed, and robots do not learn from their mistakes or optimize their behaviour over time. Furthermore, a robot expert is still needed to program the robot. A solution to these challenges would be a new generation of robots and software which can adapt quickly to new situations and learn from their mistakes while being programmable without needing an expert.

A concept that can enable the transition to flexible robotics was already introduced in the 1980s: 'Programming by Example' [1]. Which is also called *Programming by Demonstration* (PbD), *Learning from Demonstration* (LfD) or *Imitation Learning.* These terms cover methods "by which a robot learns new skills through human guidance and imitation" [2]. Imitation learning is derived from how humans monitor actions of other individuals to attain knowledge [3]. The purpose of imitation learning is to perform a task by generalizing from observations [4]. Throughout this thesis the term imitation learning will be used.

When introduced in the 80s, imitation learning was called *guiding* [5]. In guiding the user moves the robot(arm) to desired positions while the trajectory is recorded. In this way an exact copy of the demonstrated trajectory is reproduced. Today this method is most referred to as teach-in [6]. Teach-in, also called kinesthetic teaching, is the input method for the state-of-the-art robots like the UR5 from Universal Robotics or Baxter from Rethink Robotics. In most systems teach-in is only used for storing way-points or trajectories, like the aforementioned guiding. Although the task is learned trough a form of imitation, the robot is not able to generalize and thus it is doubtful to speak about imitation learning. Baxter is an exception to this in the sense that it uses vision to cope with uncertainties in the environment e.g. coping with different locations of the object that has to be picked up. The downside of teach-in as input method is, that it is hard to move all the joints of the robot in the right way especially with bigger robots like Baxter. This might result in a trajectory that is suboptimal or includes unnecessary moves. In most industrial robots there is even no teach-in mode available because the robot is too big for doing teach-in, this might even lead to safety issues.

Besides teach-in there are currently more input methods used in research labs like teleoperation where the robot is moved by means of a remote control, motion-sensors or, vision-based methods to track the movements of the body of the demonstrator.

When looking into an input method for imitation learning, the question of *What to imitate?* is treated. According to [2] three other questions can also be asked regarding imitation learning: *How to imitate?*, *When to imitate?* and *Whom to imitate to?*. The how question focusses on how to execute actions such that the robot reproduces the behavior that was demonstrated. The other two questions respectively treat at which moment to imitate and who the target of the learned task is. Only the first two questions are addressed in research up till now [2]. This thesis will mainly focus on the *How to imitate?* question.

As indicated in the beginning, imitation learning sounds promising regarding flexible robots in industrial settings. However when we look at what is happening in industrial practice, there is no significant role played by imitation learning. Most researches are also not focussed on industrial applications but on humanoid robots for personal applications. An exception to this is some work of Kober and Calinon e.g. [7] and [8]. This thesis will focus on imitation learning in an industrial setting and more specifically imitation learning for a robotic precision placement task.

Important concerns when applying imitation learning in an industrial task are: an operator has to be able to teach the robot, the robot should improve the demonstration in order to achieve a more efficient execution of the task, and preferably the learning should need as few demonstrations as possible. The advantage of doing few demonstrations is that few human interactions are needed. To achieve a more efficient execution of the task, reinforcement learning is combined with the aforementioned imitation learning in order to improve on the demonstration. The research question for this thesis is: "Can imitation learning be combined with reinforcement learning to achieve a successful application in an industrial robotic precision placement task?"

However, learning a robot skill is a complex and high-dimensional task [9]. In this thesis three methods are combined to reduce the complexity of learning a robot skill. The first method is reducing the dimensionality of the task by using Dynamic Movement Primitive (DMP) to encode the skill [10]. These DMPs are initialized by imitation learning through giving a demonstration, which is the second method. The third method is the use of a policy gradient method: Policy Gradients with Parameter based Exploration (PGPE) which is suited for complex, high dimensional reinforcement learning problems [11]. The purpose of this thesis is to combine these methods to achieve an easy way to program a robot for a precision placement task.

This thesis has two main contributions to the existing work. The first contribution of this thesis is the policy gradient algorithm which has not been combined with imitation learning before. The policy gradient algorithm is modified to be used in combination with imitation learning and to suit the use case. The second contribution is the combination of imitation and reinforcement learning being applied in an industrial use-case.

In the remainder of this thesis, the above concepts are explained and applied to a use-case. Therefore this thesis is structured as follows: in Chapter 2 background information is given on movement primitives, imitation and reinforcement learning. Chapter 3 explains how DMPs can be used for encoding trajectories. Subsequently in Chapter 4 the policy gradient algorithm is elaborated on. In Chapter 5 it is explained how imitation learning is used to initialize the policy search algorithm. Then in Chapter 6 the use-case that forms the basis of this thesis is explained. The theory of the above mentioned chapters is then applied to the use-case in Chapter 7. The results of the experiments that are done in relation to the use-case are given in Chapter 8. Finally in Chapter 9 the conclusions are drawn and some directions for future work are given.

Chapter 2

Background information

This chapter presents the work that is described in literature related to imitation and reinforcement learning. This chapter starts in Section 2.1 by explaining the theory behind movement primitives which is a principle that is used as a basis for imitation learning in many papers. Subsequently in Section 2.2 the work on imitation learning is reviewed. Finally in Section 2.3 the work in reinforcement learning (for robotics) will be elaborated on.

2.1 Movement primitives

Movement primitives is a concept used as a basis in several studies concerning imitation learning. A movement primitive, also called motor primitive, is derived from the complex movement skills that are found in nature. Complex movement skills are combinations of movement primitives [12]. These movement primitives can be seen as units of actions, basis behaviors or motor schemas [13]. In 2003 Schaal recognized that movement primitives were a possibility for robots to cope with the complexity of motor control and motor learning [13]. Therefore the concept of Dynamic Movement Primitive (DMP) was introduced, a concept that can be used for imitation learning.

There are two kinds of DMPs: point attractive systems and limit-cycle systems. These are used to represent discrete and rhythmic movement plans respectively. A DMP consists of two sets of equations a canonical system and a transformation system. The transformation system is given by:

$$\begin{aligned} \tau\dot{v} &= \alpha_v(\beta_v(g-x)-v)+f(s) \\ \tau\dot{x} &= v \end{aligned} \tag{2-1}$$

Which is a damped spring model with nonlinear terms. Here τ is a time constant and α_v and β_v are positive constants. They represent spring stiffness and damping respectively. g is the generalized goal position, x is the generalized current position and v is the generalized velocity. Depending on the choice of $f(s)$ the system can be made point attractive or oscillatory. This function $f(s)$ is called the forcing term.

Discrete movement plans For discrete movement plans the canonical system is defined by:

$$\tau \dot{s} = -\alpha_s s \tag{2-2}$$

Here α_s is a predefined constant and s is the phase variable. This phase variable is used instead of time to make the dependency of the resulting control policy on time more implicit. This canonical system is used to replace the time component and models a generic behavior of the model equations. The forcing term f is defined as:

$$f(s) = \frac{\sum_{i=1}^{\mathcal{N}} \Psi_i(s) w_i}{\sum_{i=1}^{\mathcal{N}} w_i} s(g - x_0) \tag{2-3}$$

Where $\Psi_i(s)$ is a basis function given by:

$$\Psi_i(s) = \exp\left(-\frac{1}{2\sigma_i^2}(s - c_i)^2\right) \tag{2-4}$$

Here σ_i and c_i are constants that determine the width and centers of the basis functions. x_0 in Equation (2-3) is the initial state. The parameters w_i in Equation (2-3) can be adjusted using learning algorithms in order to produce complex trajectories [10, 13].

Rhythmic movement plans For the rhythmic movement plans the canonical system, the forcing term and the basis functions differ from the ones given for the discrete movement plans. The canonical system is given by:

$$\tau \dot{\phi} = 1 \tag{2-5}$$

here ϕ is the phase angle of the oscillator in polar coordinates and the amplitude of the oscillation is r. The forcing term and the basis function are given by:

$$f(\phi, r) = \frac{\sum_{i=1}^{N} \Psi_i w_i}{\sum_{i=1}^{N} \Psi_i} r \tag{2-6}$$

$$\Psi_i = \exp(h_i(\cos(\phi - c_i) - 1)) \tag{2-7}$$

The basis functions are now gaussian-like functions that are periodic. The amplitude and period of the oscillations can be modulated by varying r and τ [10, 13].

DMPs have however several drawbacks when they are generalized to other contexts than the context that was used to learn the DMP [14]. The first drawback is that if the start and goal position, x_0 and g, of a movement are the same, the system will stay at x_0. The second drawback is that if $g - x_0$ is close to zero, a small change in g may lead to huge accelerations. The third drawback is that when a movement adapts to a new goal and the new goal is in the opposite direction of the old goal, the resulting generalization is mirrored. Therefore [14] modifies the DMPs by replacing the structure of the aforementioned transformation system. In order to prevent the huge accelerations, first order filtering can be applied to the goal change [10].

A different approach to movement primitives was taken in [15] where the behaviors underlying in human motion data are extracted. Peformance-Derived Behavior Vocabularies (PDBV) are used to design a repertoire of skills. Here a skill can be either a primitive or a meta-level behavior. A primitive behavior has the ability to provide a prediction about the vector direction given a certain point in the joint angle space. A meta-level behavior is a sequential model of transitioning between a set of primitives. A PDBV assumes that there is a spatial-temporal structure in the data in order to extract behaviors. So the problem of finding behaviors can be seen as a problem of finding a spatio-temporal structure in the data.

2.2 Imitation learning

As indicated in Chapter 1, in the literature several terms are used to describe imitation learning. The most common terms are *Programming by Example*, *Programming by Demonstration*, *Learning from Demonstration* and *Imitation Learning*. Furthermore the term *apprenticeship learning* is used, this term has however multiple meanings in the literature, see also Section 2.3.2.

This section will first describe imitation learning based on the DMPs described in Section 2.1 in Section 2.2.1. Subsequently in Section 2.2.2 imitation learning based on Gaussian Mixture Models is explained. Then in Section 2.2.3 Hidden Markov Models are used as a basis for imitation learning. Finally in Section 2.2.4 other techniques that are used for imitation learning are described.

2.2.1 Imitation Learning based on Dynamic Movement Primitives

There are two levels where learning can be performed: learning the DMP itself and learning a task using DMPs. The first can be achieved by finding the parameters w_i of the non-linear function f that is part of every DMP [16] as can be seen in Equation (2-6). The goal is to reproduce the demonstration of the teacher [17]. The movement primitive is used as a parameterized policy $\pi(\theta)$ where the parameters θ are given by the weights w_i. The parameters are learned through Locally Weighted Regression (LWR) [16, 18, 19, 20]. The weights are estimated such that the error $J = \sum_{t=1}^{T}(f_{target}(x(t)) - f(x(t)))^2$ is minimized [18, 21].

For the learning of tasks using DMPs it is assumed that there is already some kind of library of DMPs available [17, 18, 19]. In [18] the library is used to recognize which of the primitives in the library is present in the observed trajectory. If there is no primitive found in the library that matches a segment of the observed trajectory, it is added to the library.

Another way of using the library is generating a new movement by computing a weighted average of all movement primitives in a library. The movement primitive is saved as a motor policy $\pi_i(\theta)$ in a library, where θ is the augmented state as mentioned above. The Mixture of Motor Primitives generates a new movement according to the motor policy given by:

$$\pi(x) = \frac{\sum_{i=1}^{L} \gamma_i(\delta)\pi_i(x)}{\sum_{i=1}^{L} \gamma_i(\delta)} \tag{2-8}$$

Here $\gamma_i(\delta)$ generates the weights and $x = [s, \delta]$ is the augmented state. Here s is the state and δ are additional parameters like final state and duration of the movement. All weights $\gamma_i(\delta)$

together form a gating network. This gating network gives high weights to relevant movement primitives while bad-suited movement primitives get low weights. So the mixture of motor primitives selects primitives from the library based on the state x [17, 19].

A third approach is proposed by [22] which automatically creates Petri nets based on observed demonstrations. Petri nets consist of places and transitions. Places represent states in the system while transitions represent actions. The action recognition is done by using DMPs in combination with affinity propagation. The affinity propagation algorithm is used to cluster the weight parameters of the DMPs to identify similar motions and selects one motion to represent a cluster. When a motion has finished, the motion is modeled by a DMP and then identified using the affinity propagation algorithm. If the motion is new, it is added as a new transition in the Petri net. The state of each object is represented by a place in the Petri net. By using a Petri net, new ways of performing a task can be included [22, 23].

2.2.2 Imitation Learning based on Gaussian Mixture Models

This subsection will describe the use of Gaussian Mixture Model (GMM) in imitation learning. Therefore first is explained what GMMs are, then how they are used and finally some variations on GMM will be shown.

Gaussian Mixture Models

A GMM is a method to model the joint distribution of an 'input' and 'output' variable [24]. It is a parametric probability density function represented as a weighted sum of Gaussian component densities [25]. A dataset of N datapoints of D dimensions is encoded in a GMM of M Gaussian densities [25, 26]. This can be written as [27]:

$$p(x_i) = \sum_{k=1}^{M} \pi_k p(x_i|k) \quad \text{with} \quad p(x_i|k) \sim \mathcal{N}(x_i; \mu_k, \Sigma_k) \tag{2-9}$$

such that $\sum_{k=1}^{M} \pi_k = 1$ and

$$\mathcal{N}(x_i; \mu_k, \Sigma_k) = \frac{1}{\sqrt{(2\pi)^D |\Sigma_k|}} \exp(-\frac{1}{2}(x_i - \mu_k)^T \Sigma_k^{-1} (x_i - \mu_k)) \tag{2-10}$$

which is a Gaussian function with mean vector μ_k and covariance matrix Σ_k [24, 25, 26, 27, 28]. Where x_i is the time series of all demonstrations defined in the chosen task space and π_k is the prior matrix. The mean vector μ_k and covariance matrix Σ_k can be separated into their respective input (ζ) and output(ξ) components:

$$\mu_k = [\mu_{k,\zeta}^T \quad \mu_{k,\xi}^T]^T \tag{2-11}$$

$$\Sigma_k = \begin{bmatrix} \Sigma_{k,\zeta} & \Sigma_{k,\zeta\xi} \\ \Sigma_{k,\xi\zeta} & \Sigma k, \xi \end{bmatrix} \tag{2-12}$$

The GMM can be trained by using demonstration data as training data [24]. After training it is possible to recover the expected output variable $\tilde{\xi}$ given the input variable ζ [28].

$$\tilde{\xi} = \sum_{k=1}^{M} h_k(\zeta)(\mu_{k,\xi} + \Sigma_{k,\xi\zeta}\Sigma_{k,\zeta}^{-1}(\zeta - \mu_{k,\zeta})) \tag{2-13}$$

where

$$h_k(\zeta) = \frac{\pi_k \mathcal{N}(\zeta; \mu_{k,\zeta}, \Sigma_{k,\zeta})}{\sum_{k=1}^{M} \pi_k \mathcal{N}(\zeta; \mu_{k,\zeta}, \Sigma_{k,\zeta})} \tag{2-14}$$

Thus given a certain output, the GMM is able to produce an estimate of the output whereby this last step is called Gaussian Mixture Regression (GMR).

How GMMs are used

As is explained above, GMMs are used to make an estimate of the output based on the input. This process which is called GMR, can be used to retrieve smooth generalized trajectories. In the case of trajectory learning the regression process estimates a generalized trajectory based on a set of trajectories used to train the model [26, 27]. The power of GMR is that it offers a way of extracting a single generalized trajectory from a set of trajectories that are used to train the model where the generalized trajectory is not part of the dataset but encapsulates all of its essential features [26].

One way to encode a trajectory or motion in a GMM is to take the time as input variable ζ and the velocity as output variable ξ [28]. Then the movement is modeled as a velocity profile:

$$\dot{x}^m = \tilde{\mathcal{F}}_{\dot{x}}(t) \tag{2-15}$$

Whereby $\dot{x}^m$ is the velocity of the end-effector. $\tilde{\mathcal{F}}_{\dot{x}}$ is obtained by applying GMR. Another way is to encode the trajectory to take the position x and velocity $\dot{x}$ as inputs and the acceleration $\ddot{x}$ as an output. The trajectory can now be seen as a second order system of the form

$$\ddot{x}^m = \tilde{\mathcal{F}}_{\ddot{x}}(x, \dot{x}) \tag{2-16}$$

Then also GMR is applied to obtain $\tilde{\mathcal{F}}_{\ddot{x}}$. These obtained values are used to modulate a spring-and-damper dynamical system. This spring-and-damper system is used to enable a robotic arm with n joints to reproduce a task with sufficient flexibility [24, 28].

The demonstrations can also be assumed to be instances of a stable dynamical system [29, 30], which is almost the same as Equation (2-16):

$$\dot{x} = f(x) \quad \text{with} \quad f(x^*) = 0 \tag{2-17}$$

Where $f(x)$ can be encoded using a GMM using the Stable Estimator of Dynamical Systems (SEDS) algorithm [31]. Once the GMM is learned, performing GMR yields an estimate $\hat{f}$ of the dynamical system.

Donut Mixture Model

In order to perform learning from failure the Donut distribution is introduced by [32]. The motivation for learning from failure is that that the assumption that the demonstration is successful is very strong. A Donut distribution is a difference between two Gaussians:

$$\mathcal{D}(x|\mu_\alpha, \mu_\beta, \Sigma_\alpha, \Sigma_\beta, \gamma) = \gamma \mathcal{N}(x|\mu_\alpha, \Sigma_\alpha) - (\gamma - 1)\mathcal{N}(x|\mu_\beta, \Sigma_\beta) \tag{2-18}$$

where $\gamma > 1$ and the priors must sum to 1. The aim of a Donut distribution is to find a good trajectory in the covariance space between the two Gaussian distributions [33]. This leads to the Donut Mixture Model (DMM). The DMM is used to generate novel trajectories [33].

A GMM is built from human demonstrations and the maximum of the corresponding DMM is found. There is no analytical solution to finding the maximum of a DMM therefore an optimization technique has to be used. In [32] gradient ascendant is proposed to find a maximum around an initial guess. An improvement is introduced by [33] which uses the Broyden-Fletcher-Goldfarb-Shanno algorithm. This is a quasi-Newton optimization method that iteratively approximates the Hessian to converge to the solution. This method assures fast convergence in several situations and it avoids calculation of second-order information.

In order to be able to use more than one demonstration for learning, a MultiDonut distribution is introduced [34]. This MultiDonut distribution replaces the individual Donut distributions. Thus instead of using k distributions with k holes just one distribution with k holes is used. Whereby the holes model the failed demonstrations. The MultiDonut can eliminate regions of parameters that lead to failure and remains in the region that leads to success.

2.2.3 Imitation Learning based on Hidden Markov Models

In [35] two Hidden Markov Models (HMMs) are used to capture and learn movements. These HMMs act at two levels. One HMM is used to segment a set of sliding windows. Each state of the HMM corresponds to a window, each change of state corresponds to a segment point. This HMM decomposes the incoming data stream into short segments of time series data that are potential motion primitives. The second HMM is used to group these segments and extract a model for each group of similar segments. The HMM is used to abstract the observation sequences. Thus each node in the HMM represents a group of similar motion primitives. This results in a motion primitive graph as can be seen in Figure 2-1.

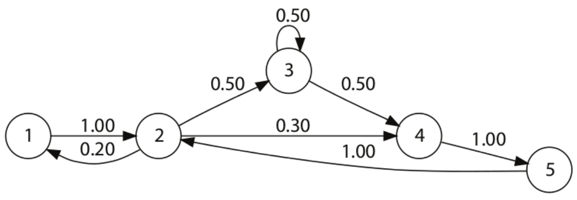


Figure 2-1: Example motion primitive graph [35].

Another approach is to encode the joint distribution $\mathcal{P}(x, \dot{x})$ of position and velocity in a HMM. The output distribution of each state is represented by a Gaussian locally encoding variation and correlation information. The parameters of the HMM are Π, a, μ, Σ. Where Π_i is the initial probability of being in state i, a_{ij} is the probability of going from state i to state j, μ_i and Σ_i represent the center and the covariance matrix of the i-th Gaussian distribution of the HMM. A desired velocity is estimated through GMR. This estimation is extended to recursively compute a likelihood through the HMM thereby also taking into account the sequential information [36].

2.2.4 Imitation Learning based on other techniques

There are a lot of other techniques that can be used to learn by imitation. This subsection will only highlight a few of them.

Iterative Improvement An algorithm is proposed that learns user preferences over trajectories through interactive feedback from the user [37]. Because it learns in an iterative manner, the user does not need to demonstrate an optimal trajectory. It is assumed that the user has a scoring function $s^*(x, y)$ that reflects how much a certain trajectory y is valued for context x. However the user cannot provide this scoring function directly, the user provides preferences that reflect the scoring function. The robot has to learn an approximation of the scoring function of the user. There are two ways to give feedback to the robot: 1) the robot displays a ranking of trajectories, the user can identify which is the best one or 2) through an interface that allows the user to correct one of the waypoints.

Probabilistic trajectory matching The key idea of probabilistic trajectory matching is to find a robot-specific policy such that the observed expert trajectories and the predicted robot trajectories match. Probability distributions are used over the trajectories these represent both the uncertainty about robot dynamics and the the variability of the demonstrations. The similarity between these distributions is measured by the Kullback-Leibler (KL) divergence. The objective is to find a policy that minimizes the KL divergence. This objective can be rewritten in such a manner that reinforcement learning algorithms can be applied to find the policy [38].

Approximate Policy Iteration In this method a mix of expert (LfD) and interaction (RL) data is used. The method is formulated as a coupled constraint convex optimization in which expert demonstrations define a set of linear constraints [39]. This optimization allows mistakes in the demonstrations. The expert samples are used to shape the value function. If there are few demonstrations, still a good generalization can be achieved due to the RL data. The key advantage of this method is that it is not assumed that the expert demonstrations are optimal.

Bayesian Network This approach incorporates both the concepts of GMM and DMP [40]. The GMM is used for autonomously segmenting the motion trajectories. The segmentation points are estimated in temporally overlapping points between two consecutive Gaussians. The set of segmented motion trajectories is then used to learn DMPs. Based on the situation and the goal, the most appropriate DMP has to be selected. To learn the relationship between task-relevant entities and motion primitives a Bayesian Network (BN) is used. The BN is learned by clustering task-relevant entities based on criteria of the motion primitives. The criterion used here is that the motion primitives have the same effect. The BNs are then used to infer motions in a given situation.

2.3 Reinforcement learning

This section will start of with work related to reinforcement learning in robotics in Section 2.3.1. In Section 2.3.2 and Section 2.3.3 inverse reinforcement learning and preference-based reinforcement learning, which are two special forms of reinforcement learning, will respectively be presented. Finally in Section 2.3.4 an introduction to data-efficient reinforcement learning will be given.

2.3.1 Reinforcement learning in robotics

This subsection will focus on Reinforcement Learning (RL) in robotics and thus not on RL in general. The principle is not different but in robotics some challenges occur when applying RL. A couple of challenges are: learning is applied to high-dimensional, continuous states and actions; every trial run is costly; simulations are hard to use due to the big effect of small modeling errors [6].

Theoretical background

RL is basically the process of learning from trial-and-error. By giving a reward or a punishment, the performance of the robot with respect to the desired goal is expressed. RL adds three missing abilities to the existing techniques: learning new tasks which even a human teacher cannot demonstrate or program, learning to achieve an optimization goal of a difficult problem by using only a cost function, and learning to adapt a skill to a new version of a task [6]. The basis of RL can be written as finding a policy $\pi(s, a)$ that gathers maximal rewards $R(s, a)$. Here s is the state and a the action. The policy can be written as $\pi(s, a) = f(a|s, \theta)$ with parameters θ. RL then focusses on finding an optimal policy π^* which maximizes the average return $J(\pi)$

$$J(\pi) = \sum_{s,a} \mu^{\pi}(s)\pi(s, a)R(s, a) \tag{2-19}$$

where μ^{π} is the stationary state distribution generated by policy π acting in the environment $T(s, a, s') = P(s'|s, a)$. This results in an optimization problem for $J(\pi)$. It can be shown that such policies that map states to actions are sufficient to ensure optimality. A policy does not need to remember previous states visited, actions taken or the particular time step. Optimizing in primal formulation is called policy search in RL while optimizing in the dual formulation is called a value function-based approach [6]. The condition for optimality in the dual formulation is:

$$V^{\pi^*} = \max_{a^*}[R(s, a^*) - \bar{R} + \sum_{s'} V^*(s')T(s, a^*, s')] \tag{2-20}$$

Where $\bar{R}$ and $V(s')$ are Lagrange multipliers.

Value function-based approaches scale poorly to high dimensions, like robotics, as a good representation of the value function becomes intractable. Furthermore even finding an optimal solution can become a hard problem due to the high dimensionality. Therefore policy search is more used in RL for robotics [6, 41]. In policy search RL a policy space which contains all possible policies is used. This reduces the dimensionality and increases the convergence speed compared to value function-based approaches [42]. There are three kind of policy search approaches: 1) policy gradient approaches based on likelihood-ratio estimation, 2) policy updates inspired by expectation maximization, and 3) path integral methods [6].

Learning Algorithms

In this subsection an overview of learning algorithms in robotics is given. This overview is a small selection and by no means exhaustive.

Natural Actor-Critic The Natural Actor-Critic algorithm exploits the natural gradient formulation [43]. The natural gradient is a gradient that is based on the Fisher information metric, a metric in Riemann spaces. By proving that the Fisher information and all-action matrix are the same, [43] derives the natural actor-critic algorithm. Therefore the Bellman Equation is rewritten as a set of linear equations. Then the least-squares policy evaluation method, LSTD(λ), is adapted to obtain value function parameters and the natural gradient. This natural gradient is used to update the policy parameters. This algorithm guarantees that it converges with probability 1 to the next local minimum of the average reward function [43].

Expectation-Maximization based RL Expectation-Maximization algorithms choose the next policy parameters θ_{n+1} by maximizing the lower bound $L_\theta(\theta')$. Hereby Equation (2-21) is solved for θ' using weighted regression with the state-action values $Q^\pi(s,a,t)$ as weights.

$$E\left\{\sum_{t=1}^{T} \partial_{\theta'} log\pi(a_t|s_t,t)Q^\pi(s,a,t)\right\} = 0 \tag{2-21}$$

However the exploration at every step is now unstructured. Therefore a form of structured exploration is introduced. By using state-dependent exploration, this results in a policy $a \sim \pi(a_t|s_t,t) = \mathcal{N}(a|\theta^T\phi(s,t),\hat{\Sigma}(s,t))$ [44]. This algorithm is called Policy learning by Weighting Exploration with the Returns (PoWER). The algorithm has two advantages over policy-gradient-based approaches: no learning rate is needed and it can make use of the previous experience of the agent [45]. In summary PoWER estimates the policy parameters to maximize the lower bound on the expected return from the following policy.

Probabilistic Inference The algorithm using probabilistic inference is called Probabilistic Inference for Learning Control (PILCO). To minimize the cost function J^π three components are used: 1) a probabilistic Gaussian process dynamics model, 2) deterministic approximate inference for long-term predictions and policy evaluations, and 3) analytic computation of the policy gradient for policy improvement. The long term predictions are done by cascading one-step predictions. By incorporating model uncertainty into long-term planning this approach reduces the effects of model errors [46].

Relative Entropy Relative entropy policy search (REPS) aims at finding the optimal policy that maximizes the expected return based on all observed series of states, actions and rewards. At the same time the loss of information has to be bounded using relative entropy between the observed data distribution and the data distribution generated by the new policy. The information loss bound is expressed using the Kullback-Leibler divergence:

$$D(p^{\pi}||q) = \sum_{s,a} \mu^{\pi}(s)\pi(a|s) log \frac{\mu^{\pi}\pi(a|s)}{q(s,a)} \leq \epsilon \tag{2-22}$$

Where μ_{π} is a stationary state distribution, $q(s, a)$ is the observed data distribution and $\mu^{\pi}(s)\pi(a|s)$ is the data distribution geneared by the new policy π. This results in an extra constraint for the maximization of the expected reward $J(\pi)$. This method can be used in a generalized policy iteration algorithm which makes it applicable for RL [47].

2.3.2 Inverse reinforcement learning

Inverse Reinforcement Learning (IRL) is learning a reward function from the demonstrations of an expert [48, 49, 50, 51, 52]. This reward function encodes the observed state-action pairs of the agent. IRL thus assumes that the expert's intent is driven by rewards. This reward function that has to be discovered is the reward function of a Markov Decision Process (MDP). This can also be written as a MDP without a reward specified, denoted by MDP\R. This is a tuple $(\mathcal{S}, \mathcal{A}, \{T^{\alpha}\}, \alpha, \gamma)$ where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, T^{α} is a transition matrix, α is the initial state distribution and γ is a discount factor [51, 53, 54, 55, 56]. Demonstrations are provided as a set of state-action pairs (s_i, a_i). From these demonstrations IRL tries to find a reward function that captures these observed demonstrations in a relation. The corresponding optimal policy π^* should match the demonstrations. The IRL problem is ill-posed [51, 54].

To resolve this ill-posedness [53] proposes to assume that there is some vector of features $\phi : S \rightarrow [0, 1]^k$ over states. There is also a true reward function $R^*(s) = w^*\phi(s)$. Where w^* is a weighting vector with unknown weights bounded by 1. Based on the assumption that the reward function is expressible as a linear combination of the features ϕ, the feature expectations $\mu(\pi)$ for a given policy π completely determine the expected sum of the discounted rewards. An estimate of the expert's feature expectations $\mu_E = \mu(\pi_E)$ is needed. This estimate is calculated from a set of trajectories $\{s_0^{(i)}, s_1^i, \dots\}_{i=1}^m$. The estimate is then given by:

$$\hat{\mu}_E = \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{\infty} \gamma^t \phi(s_t^{(i)}) \tag{2-23}$$

An algorithm then iterates over randomly picked policies and tries to find the weights that result in the best match between the expert policy and the picked policy in terms of the feature expectations.

Another approach is that the reward function is written as a combination of basis reward functions: $R_{sa} = \sum_i w_i^* R_{sa}^i$ where the unknown weights satisfy w_i^* and $\sum_i w_i^* = 1$ [56]. Each basis reward function has a corresponding basis value function $V^i(\pi)$. This results in $V(\pi) = \sum_i w_i^* V^i(\pi)$ therefore the smallest possible difference between $V(\pi)$ and $V(\pi^E)$ is $\min_i V^i(\pi) - V^i(\pi^E)$. The goal then becomes to find a policy π^A that solves

$$v^* = \max_{\pi} \min_{i} V^i(\pi) - V^i(\pi^E) \tag{2-24}$$

This can be solved by computing the weights of R_{sa}.

There are many other ways to resolve the ill-posedness or to perform IRL. In [49] maximum entropy is used, [50] first uses a classification step to obtain a score function followed by a regression step that provides a reward function. The reward function can also be approximated thereby dropping the assumption that the demonstrations are globally optimal [57]. Other techniques are using a Bayesian nonparametric mixture model [58], score-based classification [52] or bootstrapping [55].

Apprenticeship learning The term *apprenticeship learning* is used with different meanings. Sometimes it is used as a synonym for imitation learning [4] but in most literature it is associated with IRL. This is the case because the term apprenticeship learning was introduced in a paper that uses IRL to solve the problem [53].

2.3.3 Preference-based reinforcement learning

Preference-based Reinforcement Learning (PBRL), also referred to as Preference-based Policy Learning (PPL), consists essentially of four steps [59]:

1. Demonstration phase: the agent (robot) demonstrates a candidate policy.
2. Teaching phase: the expert ranks this policy compared to already demonstrated policies based on preferences.
3. Learning phase: a policy return estimate is learned based on the expert ranking.
4. Self-training phase: new policies are generated, using an adaptive trade-off between the policy return estimate and the policy diversity.

A candidate policy is selected to be demonstrated and the process is iterated. PPL has two main advantages: it does not require an expert to know how to solve the task or to demonstrate an optimal behavior [59, 60] and an expert does not need to design a reward function [60].

As is the case in Section 2.3.2 a MDP without a reward function forms the basis. It is assumed that there exists a utility function U^* that is linear in terms of features: $U^*(s,a) = W^*\phi(s,a)$ [59, 60, 61]. This utility function can be seen as the reward function in classic RL however with the difference that the utility of a trajectory can change with the occurrence of new preferences [61]. The utility function is used to select actions that will be discovered [62]. The utility function of a policy can be written in terms of feature expectations: $U(\pi) = W\mu(\pi)$ [61].

To formally represent preferences, [61] suggests a notation based on sequences of state-action pairs $C(t) : \mathbb{N} \to S \times A$. A preference is a relation between two sequences C_1, C_2. If C_1 is preferred over C_2 this is noted by $C_1 \succ C_2$. The learning process can now be seen as identifying a policy π^* which generates trajectories that agree with the observed preferences ζ.

Two approaches for comparing trajectories can be considered: 1) relational approaches that try to define a preference relation on policies and 2) value-based approaches which attempt to associate numerical values with policies. The value-based approaches use the aforementioned utility function to obtain a value for the policy [59, 60, 61]. This value is then used to indicate how good it is to follow policy π.

New policies are in [59] generated by using a stochastic optimization method. From these policies the best policy is selected and demonstrated to the expert. This selection is done by using a criterion that enforces a trade-off between the exploration of the policy space and the exploitation of the utility function. After the demonstration the process iterates.

2.3.4 Data-efficient reinforcement learning

In contrast to most RL methods which require many trials, data-efficient RL methods try to keep this number of trials to a minimum [63, 64]. Keeping the trials to a minimum has advantages in robotics because trials mean physical interactions with the environment which are often infeasible and cause wear-and-tear. In general model-based methods are more promising for efficiently extracting information from the available data than model-free methods [63]. Model-free methods update the policy by executing roll-outs on the system and collecting the corresponding rewards while no model of the robot dynamics is needed [65]. The downside of model-free methods in this context however is that a large number of interactions is needed. Model-based methods learn a dynamics model of the environment and use this model to compute the rewards [65]. Model-based methods are not very common in RL because they might suffer from model bias which is especially a problem when using only a few samples [63, 64, 65].

The best known data-efficient reinforcement learning algorithm is PILCO: probabilistic inference for learning control [63]. PILCO uses Gaussian processes to express model uncertainty in the dynamics model. Analytic gradients are used for policy improvement. This results in an order of magnitude less trials needed for a cart-pole task, compared to conventional algorithms.

Another algorithm that takes the same direction is GPREPS [65]. Here Gaussian processes are used to learn forward models. These models are used to predict the reward. The policy updates are based on the REPS algorithm [47]. For a 4-link pendulum balancing task, GREPS needs about the same amount of experience as PILCO for achieving the same reward limit. However the computation time of a sample trajectory for PILCO is 300 times higher than the time required by GREPS. This shows that PILCO is data-efficient but it comes at the cost of computation time.

A different approach is taken in [66] where reinforcement learning is seen as a reward-weighted self-imitation. Therefore the reward weighted regression is reformulated in a Cost-regularized Kernel Regression. This algorithm is compared to a policy gradient algorithm (finite difference) and reward-weighted regression. The CrKR outperforms these algorithms significantly.

Summary

Within imitation learning there are two main movements for encoding demonstrations. The first is Dynamic Movement Primitives which is a mathematical spring-damper system. These DMPs virtually pull a goal position to a goal state. The second movement is based on Gaussian Mixture Models which encode the demonstration in such a model.

Reinforcement learning is the process of learning from trial-and-error. The basis of RL is finding a policy that gathers maximal rewards. In robotics policy search is the most used RL technique, this technique reduces the dimensionality and increases the convergence speed.

Two different RL techniques are inverse reinforcement learning and preference-based reinforcement learning which base the reward function on the demonstrations or the feedback of an expert respectively.

In order to reduce the number of iterations that is needed for the learning process, research is done in data-efficient reinforcement learning. These techniques try to keep the number of trials to a minimum.

Chapter 3

Encoding trajectories in Dynamic Movement Primitives

The trajectories representing the movement that are received in terms of coordinates and orientations have to be encoded in such a way that the movements can be generalized and learning can be applied. In Chapter 2 several methods to encode the trajectories have been mentioned. In this thesis Dynamic Movement Primitive (DMP) will be used. There are two main reasons why DMPs are used: First in this thesis ideally one demonstration of the movement is given which can be captured in a DMP while a Gaussian Mixture Model (GMM) is more useful if multiple demonstrations of the same movement are given, second DMPs are proven to be suitable for generalization and learning [14, 67]. Apart from GMM there are also other methods but the most used and mentioned in the literature are DMPs and GMM.

This chapter will start with formally describing the DMPs that are used in this thesis in Section 3.1. In Section 3.2 will be explained how an observed movement can be encoded in a DMP. Finally in Section 3.3 it is described how to combine various DMPs.

3.1 Description of Dynamic Movement Primitives

DMPs were already described in Section 2.1 but there exist several types of DMPs. This section will describe the type of DMPs that will be used throughout this thesis.

As described in Section 2.1 a DMP consists of a transformation system, a canonical system and a forcing term. The original DMPs have some drawbacks: if start and goal position of a movement are the same, the system will remain at the start position; if the start and goal position are very close together a small change in the goal position can lead to huge accelerations; if a movement adapts to a new goal g_{new} such that $(g_{new} - x_0)$ changes its sign compared to $(g_{original} - x_0)$ the resulting generalization is mirrored [10, 14]. To overcome these drawbacks an adjusted DMP is proposed [14]. In Figure 3-1 a comparison of the trajectories generated by the different types of DMPs is given.

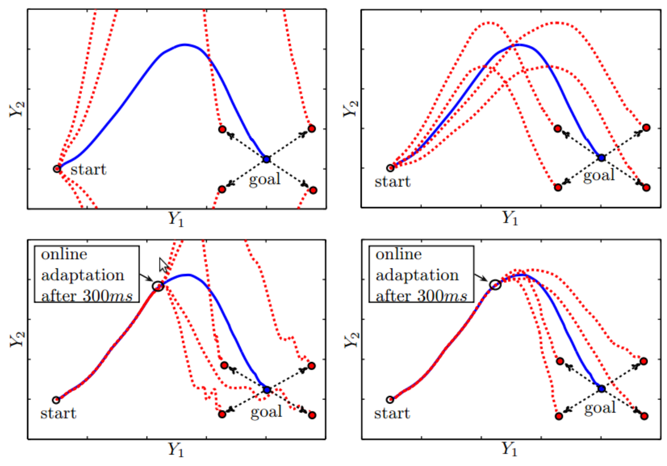


Figure 3-1: Comparison of goal-changing results between old (Left) and new (Right) DMP formulation in operational space (Y1, Y2) with one transformation system for each dimension. The same original movement (solid line) and goals are used for both formulations. The dashed lines show the result of changing the goal before movement onset (Top) and during the movement (Bottom) [14].

The transformation system is given by:

$$\begin{aligned} \tau\dot{v} &= K(g-x) - Dv - K(g-x_0)s + Kf(s) \\ \tau\dot{x} &= v \end{aligned} \tag{3-1}$$

Here K and D are a spring and damping constant which are chosen such that the system is critically damped; x and v are the generalized position and generalized velocity; x_0 and g are the start and goal position; τ is a temporal scaling factor; and $f(s)$ is given by:

$$f(s) = \frac{\sum_{i=1}^{N} w_i \psi_i(s) s}{\sum_{i=1}^{N} \psi_i(s)} \tag{3-2}$$

where $\psi_i(s) = \exp(-h_i(s-c_i)^2)$ is a Gaussian basis function with center c_i and width h_i and w_i are adjustable weights. This function depends on the phase variable s which changes from 1 to 0 during a movement and is given by the canonical system:

$$\tau\dot{s} = -\alpha s \tag{3-3}$$

Here α is a predefined constant. The term $K(g-x_0)s$ in Equation (3-1) is required to avoid jumps at the beginning of the movement. These sets have some favourable characteristics: convergence to the goal is guaranteed, the equations are spatial and temporal invariant, and the formulation generates movements that are robust against perturbations [14].

In order to apply DMPs on multiple Degrees of Freedom (DoFs) there are three approaches: 1) each DOF can be encoded in a separate DMP, 2) create coupling terms between different DoFs, or 3) share one canonical system among all DoFs while each DoF has its own transformation system [10, 14]. This last approach is used in both [10] and [14] and will also be used in this thesis. An example for a 7-DoF system is given in Figure 3-2 here the orientation of the end effector is encoded in quaternions.

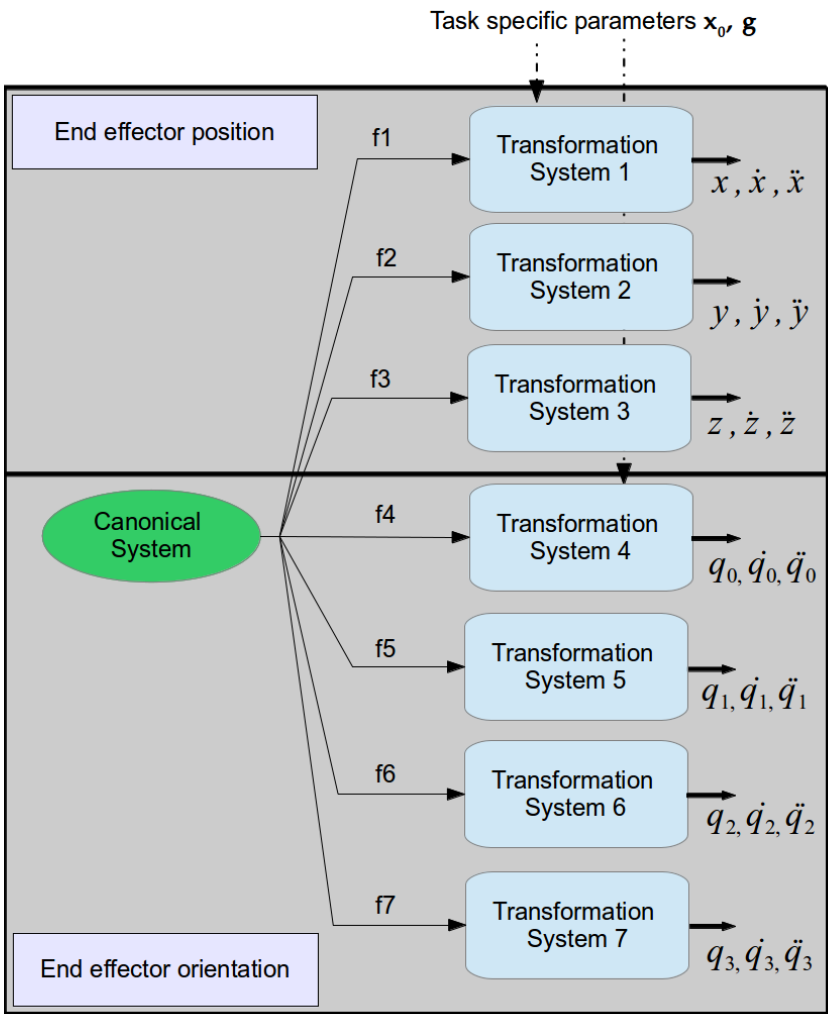


Figure 3-2: Overview of a 7-DoF DMP, derived from [14].

3.2 How to encode a trajectory in a DMP

Let for simplicity take a one dimensional case to explain how to encode an observed trajectory in a DMP. The desired behavior is given in terms of position which is differentiated to obtain velocity and acceleration. This results in the triple $(x_{demo}(t), \dot{x}_{demo}(t), \ddot{x}_{demo}(t))$ where $t \in [1, \ldots, P]$. Encoding the movement is then done in two phases: first the high-level parameters g, x_0 and τ are determined and then learning is performed for the weights w_i [10]. This learning of the weights will be elaborated on in Section 5.2. Here we only give the starting point.

Obtaining the high-level parameters is relatively easy. The parameter g is the position at the end of the movement, $g = x_{demo}(t = P)$. In the same way is x_0 the start position of the movement, $x_0 = x_{demo}(t = 0)$. The parameter τ is the duration of the demonstration. In [10] it is suggested that some tresholding is applied to obtain τ. For example a velocity treshold of 2% of the maximum velocity in the movement may be used and τ can be chosen as 1.05

times the duration of this tresholded trajectory piece. This factor is to compensate for the treshold.

In order to learn the weights w_i, Equation (3-1) is rewritten to:

$$f_{target}(s) = \frac{\tau^2 \ddot{x} + D\tau\dot{x}}{K} - (g - x) + (g - x_0)s \tag{3-4}$$

Where s is obtained by integrating the canonical system. Then regression techniques can be applied to find for instance the weights that minimize the difference between $f(s)$ and $f_{target}(s)$. This results in the cost function $J = \sum_s (f_{target}(s) - f(s))^2$. There are however all kind of cost functions possible.

3.3 Combination of DMPs

More complex movements can be generated by combining several DMPs. Starting a DMP after another DMP has finished is straight forward because the boundary conditions of a DMP are that the velocity and acceleration are zero. However, complete stops of the system are undesirable. Avoiding stops of the system can be done by starting the execution of the successive DMP before the preceding DMP has finished. In this case the velocities and accelerations are not zero. Jumps in the acceleration can be avoided by properly initializing the succeeding DMP with the velocities and positions of the preceding DMP [14].

Summary

Dynamic Movement Primitives are a spring-damper like system with a forcing term that can be used to encode each kind of trajectory. The spring-damper system virtually pulls the start state to the goal state while the forcing term determines the shape of the trajectory between these two. DMPs for multiple degrees of freedom are formed by using one canonical system while each degree has its own transformation system. One demonstration is used to encode a DMP, the weights of the forcing term are determined using regression techniques.

Chapter 4

Policy Gradients with Parameter based Exploration

This chapter starts by describing the relation of Dynamic Movement Primitive (DMP) to parameterized policies in Section 4.1. From this section it can be seen that Dynamic Movement Primitives can be optimized using policy search algorithms. In Section 4.2 and Section 4.3 two versions of the parameter-exploring policy gradients algorithm will be explained. This is the policy search algorithm that will be used in this thesis. Some extensions to the algorithms are given based on the findings in this thesis in respectively Section 4.2 and Section 4.4.

4.1 Dynamic Movement Primitive as a parameterized policy

In reinforcement learning a policy π maps states to actions. This policy can be optimized to find the optimal policy π^*. The optimal policy optimizes the cumulative discounted reward over time, also referred to as expected reward [11, 68]. When the action and state space become continuous, the problem of finding an optimal policy becomes hard to solve. Therefore parameteric policies π_θ are used to find the optimal policy π^*. This is done by finding the optimal policy parameters θ^*. These parameters θ^* thus optimize the cumulative discounted reward [68].

One way to parameterize a stochastic policy is to linearly parameterize it. The action a_t is then written as:

$$a_t = g_t^T(\theta + \epsilon_t) \tag{4-1}$$

Here g_t is a vector of basis functions and θ still denotes the parameter vector. The additive exploration ϵ_t is used to keep the policy stochastic [44, 69].

Following this line of thought, DMPs can be seen as a special case of parameterized policies.

This becomes more clear if we rewrite Equation (3-1) slightly:

$$\begin{aligned}
\tau\dot{v} &= f_t + K g_t^T(\theta + \epsilon_t)\\
\tau\dot{x} &= v\\
\tau\dot{s} &= -\alpha s\\
f_t &= K(g - x) - Dv - K(g - x_0)s
\end{aligned} \tag{4-2}$$

Equation (3-2) is then transformed to:

$$[g_t]_i = \frac{\psi_i(s)s}{\sum_{i=1}^N \psi_i(s)} \tag{4-3}$$

The DMP equations are now written in the same form as Equation (4-1). Hereby the parameters θ of the policy correspond to the weights w of the basis functions [69]. It can be concluded that a DMP is a special form of a parameterized policy.

As mentioned before, the policy can be optimized to find the optimal cumulative or expected reward. The expected reward of the policy π with parameters θ is defined by

$$J(\theta) = \int_T p_\theta(\tau)R(\tau)d\tau \tag{4-4}$$

Where T is the set of all possible paths and τ a possible path which is a series of states and actions [44]. $R(\tau)$ is the reward over the path τ. Policy search algorithms try to find a parameter vector $\theta^* = argmin_\theta J(\theta)$ which minimizes the expected cost (or maximize the reward) [70]. In the next sections a policy search algorithm will be elaborated on.

4.2 Basic parameter-exploring policy gradients

The Policy Gradients with Parameter based Exploration (PGPE) algorithm is a policy search algorithm. In this section a summary of the derivation in [11] is given. The algorithm is derived from the general framework of episodic reinforcement learning in a Markovian environment.

We start by assuming that a stochastic policy suffices: the distribution over actions only depends on the current state and the agent parameters θ: $a_t \sim p(a_t|s_t, \theta)$. Given a history h of state-action pairs, a cumulative reward can be computed: $r(h) = \sum_{t=1}^T r_t$. The goal of reinforcement learning is then to optimize the agent's expected reward:

$$J(\theta) = \int_H p(h|\theta)r(h)\mathrm{d}h \tag{4-5}$$

One way to optimize this is to estimate the gradient $\nabla_\theta J$ and apply gradient ascent optimization. The gradient can be written as:

$$\nabla_\theta J(\theta) = \int_H p(h|\theta)\sum_{t=1}^T \nabla_\theta log(p(a_t|s_t,\theta))r(h)\mathrm{d}h \tag{4-6}$$

Note that there is an error in the original paper [11] at this point, accidentally the logarithm was left out. Because integrating over the entire space of histories is infeasible, sampling methods have to be used

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T} \nabla_\theta log(p(a_t^n|s_t^n,\theta))r(h^n) \tag{4-7}$$

where the histories h^i are chosen according to $p(h^i|\theta)$.

The above derivation relates PGPE to classical policy gradients. However it is now the question how to model $p(a_t|s_t,\theta)$. In policy gradient methods such as REINFORCE [71] the parameters θ are used to determine a probabilistic policy. Sampling from this policy each time step however leads to a high variance in the sample over histories which leads to a noisy gradient estimate.

PGPE decreases the variance by using a probability distribution over the parameters θ instead of a probabilistic policy

$$p(a_t|s_t,\rho) = \int_\Theta p(\theta|\rho)\delta_{F_\theta(s_t),a_t}\mathrm{d}\theta \tag{4-8}$$

where ρ are the parameters determining the distribution over θ, $F_\theta(s_t)$ is the deterministic action chosen by the model with parameters in state s_t. The actions are now deterministic and therefore an entire history can be generated from a single parameter sample. An extra benefit is that the parameter gradient is estimated by direct parameter perturbations.

The expected reward, given a ρ, now becomes:

$$J(\rho) = \int_\Theta \int_H p(h,\theta|\rho)r(h)\mathrm{d}h\mathrm{d}\theta \tag{4-9}$$

This then leads, by using sampling methods, to the gradient estimator

$$\nabla_\rho J(\rho) \approx \frac{1}{N}\sum_{n=1}^{N} \nabla_\rho log(p(\theta|\rho))r(h^n) \tag{4-10}$$

By making the assumption that ρ consists of a set of means μ_i and standard deviations σ_i that determine a normal distribution for each parameter θ_i, the derivative of $log(p(\theta|\rho))$ can be written with respect to μ_i and σ_i.

The reward gradient of Equation (4-10) requires for each sampling an entire state-action history. A cheaper gradient estimate can be made by drawing a single sample θ and comparing its reward r to a baseline reward b. This baseline is a moving average over previous samples. Using the baseline the parameters are updated according to the following equations (using a step size $\alpha_i = \alpha\sigma_i^2$ in the direction of the positive gradient):

$$\Delta\mu_i = \alpha(r-b)(\theta_i-\mu_i) \qquad \Delta\sigma_i = \alpha(r-b)\frac{(\theta_i-\mu_i)^2-\sigma_i^2}{\sigma_i} \tag{4-11}$$

In Algorithm 4.1 the resulting pseudo code of the algorithm can be found.

Algorithm 4.1 Basic PGPE algorithm

```
procedure BASICPGPE($\mu_{init}, \sigma_{init}$)
    $\mu \leftarrow \mu_{init}$
    $\sigma \leftarrow \sigma_{init}$
    while True do
        for n=1 to N do
            draw $\theta^n \sim \mathcal{N}(\mu, I\sigma^2)$
            evaluate $r^n = r(h(\theta^n))$
        end for
        $T = [t_{ij}]_{ij}$ with $t_{ij} := (\theta_i - \mu_i)$
        $S = [s_{ij}]_{ij}$ with $s_{ij} := \frac{t_{ij}^2 - \sigma_i^2}{\sigma_i}$
        $r = [(r^1 - b), \ldots, (r^N - b)]^T$
        update $\mu = \mu + \alpha T r$
        update $\sigma = \sigma + \alpha S r$
        update baseline $b$                ▷ An update rule for $b$ can be: $b = 0.9b + 0.1r$
    end while
end procedure
```

Limiting the randomness

In practice the values for $r - b$ were relatively big which resulted in huge values for the updates. This had as a result that the samples were drawn from a bigger range than necessary. Which finally led to obtaining very big rewards which amplified the effect, resulting in infinite rewards. To limit the randomness or the range whereof the samples were drawn, the $r - b$ values had to be scaled. Inspired by the algorithm in Section 4.3 it was decided to keep track of the maximum (or minimum) reward m received so far. Then the baseline b is subtracted from m. This value is then finally used to scale the $r - b$ value resulting in $\frac{r-b}{m-b}$ as value to replace $r - b$.

4.3 Symmetric sampling parameter-exploring policy gradients

An improvement of the algorithm in Section 4.2 is PGPE with symmetric sampling. This gradient approximation is more robust because it measures the difference in reward between two symmetric samples on either side of the current mean.

This is done by picking a perturbation ϵ from the distribution $\mathcal{N}(0, \sigma)$, using this perturbation two parameter samples are created $\theta^+ = \mu + \epsilon$ and $\theta^- = \mu - \epsilon$. Then r^+ is the reward of θ^+ and r^- of θ^-. Both rewards are then used in the update equations:

$$\Delta\mu_i = \frac{\alpha\epsilon_i(r^+ - r^-)}{2m - r^+ - r^-} \qquad \Delta\sigma_i = \frac{\alpha}{m-b}\Big(\frac{r^+ + r^-}{2} - b\Big)\Big(\frac{\epsilon^2 - \sigma_i^2}{\sigma_i}\Big) \tag{4-12}$$

Where m is the maximum (or minimum) reward an agent can receive or when this is unknown the maximum (or minimum) reward received so far. For the complete derivation of the update equations see [11]. In Algorithm 4.2 the pseudo code for the symmetric sampling algorithm is given. Note that this pseudo code includes the reward normalization which was omitted in

[11]. It was decided to include it in this pseudo code because otherwise the algorithm is not presented in accordance with the update equations of Equation (4-12).

Algorithm 4.2 Symmetric sampling PGPE algorithm

```
procedure SYMMETRICPGPE($\mu_{init}, \sigma_{init}$)
    $\mu \leftarrow \mu_{init}$
    $\sigma \leftarrow \sigma_{init}$
    while True do
        for n=1 to N do
            draw $\epsilon^n \sim \mathcal{N}(0, I\sigma^2)$
            $\theta^{+,n} = \mu + \epsilon^n$
            $\theta^{-,n} = \mu - \epsilon^n$
            evaluate $r^{+,n} = r(h(\theta^{+,n}))$
            evaluate $r^{-,n} = r(h(\theta^{-,n}))$
        end for
        $T = [t_{ij}]_{ij}$ with $t_{ij} := \epsilon_i^j$
        $S = [s_{ij}]_{ij}$ with $s_{ij} := \frac{t_{ij}^2 - \sigma_i^2}{\sigma_i}$
        $r_T = [(\frac{r^{+,1}-r^{-,1}}{2m-r^{+,1}-r^{-,1}}), \dots, (\frac{r^{+,N}-r^{-,N}}{2m-r^{+,N}-r^{-,N}})]^T$
        $r_S = [(\frac{1}{m-b}(\frac{r^{+,1}+r^{-,1}}{2} - b)), \dots, (\frac{1}{m-b}(\frac{r^{+,N}+r^{-,N}}{2} - b))]^T$
        update $\mu = \mu + \alpha T r_T$
        update $\sigma = \sigma + \alpha S r_S$
        update baseline $b$                    ▷ An update rule for $b$ can be: $b = 0.9b + 0.1r$
    end while
end procedure
```

4.4 Dealing with small initial costs and a final reward

After doing a lot of testing, which is described in Chapter 8, it appeared that the algorithm can not handle the combination of small costs in the beginning of the learning and a (binary) final reward. If in the beginning small costs are rewarded and then a big cost is rewarded, the algorithm crashes. This can be explained as follows:

Let's assume we have received costs, both the positive and the negative cost, in the order of 0.01 in the first two cycles. The baseline b is then also in the order of 0.01. In the third cycle we get a positive cost in the order of 100.0 and a negative cost in the order of 0.01. Take for simplicity $r^+ = 100.0$ and $r^- = 0.01$. Then the mean of the two costs becomes $mean = 50.005$. The baseline is the result of the average of the two cycles before so it is in the order of 0.1, take $b = 0.2$. The minimum value was received in the first cycle and is $m = 0.05$. What you then get is: $\frac{1}{m - baseline} = -6.67$, $mean - b = 49.805$. This results in $\sigma_\delta = -332.20$. Where σ_δ is the first part of the update equation for σ of Equation (4-12).

The above illustrates that this value can become very big in one cycle. This results in a big growth of the σ value. This σ determines the width of the normal distribution whereof the perturbations are drawn. So the chance of drawing a big perturbation has grown extremely in just one cycle. A big perturbation then leads to a big change in θ. Which results in a huge cost in most cases. Thereby in a few cycles the algorithm runs to costs that are infinity.

The above sketched situation happens when a final cost or reward is used. Especially when a binary final cost is given and this final cost is important and has therefore a high weight. Because this behaviour is undesirable, a solution has to be found. As illustrated above the combination of a relatively big $\frac{1}{m-baseline}$ and a relatively big $mean - b$ results in a big value for σ_δ. Therefore action has to be taken to limit the σ_δ when it grows to big. First it was looked into if the maximum value of σ_δ could be based on the values that are available to the algorithm. However, no good relation was found between the values available and a maximum value for the σ_δ. For this thesis the maximum value was therefore determined empirically. From observations it could be seen that the value was always below 20.0. Therefore it was decided to reset all values of σ_δ that are higher than 20.0 to 0.0. This means that the current σ value is not changed in the current iteration.

Summary

The policy search algorithm used in this thesis is Policy Gradients with Parameter based Exploration. There are two versions of the algorithm: a basic version and a symmetric sampling version.

The basic PGPE algorithm is a policy search algorithm that samples perturbations from a normal distribution. The mean and standard deviation of the normal distribution are updated according to the reward of the iteration step. Relatively high update values were received for the normal distribution therefore it was decided to add a scaling factor to the original algorithm. This resulted in a more stable algorithm.

The symmetric sampling version of the algorithm is the same as the basic algorithm apart from the fact that it uses two samples on either side of the current mean. This results in two rewards for the two samples which are both used to update the current mean and standard deviation. The symmetric sampling version of the algorithm is more robust and needs less samples than the basic version even though the algorithm needs twice as much samples per iteration.

The symmetric sampling version of the algorithm is adjusted such that it can deal with small initial costs and a final reward. Therefore a simple clipping measure was taken based on empirical values.

Chapter 5

Combination of policy search and imitation learning

In this chapter it is explained how the trajectories that are encoded in a Dynamic Movement Primitive (DMP) (Chapter 3) are combined with the policy gradient algorithm in Chapter 4. Therefore, in Section 5.1 it is explained how imitation learning initializes the policy search. In Section 5.2 we improve the learning by splitting the demonstrated trajectory in multiple sub-trajectories. Troughout this chapter a 2-DoF example will be used to illustrate the differences between the approaches that are explained. This 2-DoF example can be seen as a simplified version of the use-case described in Chapter 6. The same principles apply however the number of DoFs is reduced.

5.1 Initializing policy search with imitation learning

5.1.1 DMPs and policy search

As explained in Section 4.1, DMPs can be seen as a parametrized policy. The weights of the DMPs can be used as the parameters of the policy that has to be learned. In Section 3.2 it was described how the weights of the DMPs can be obtained by using regression techniques. The idea of initializing policy search with imitation learning is to first obtain an initial approximation of the demonstrated trajectory by using regression techniques to obtain the initial weights. These initial weights are then used as a starting point for the Policy Gradients with Parameter based Exploration (PGPE) algorithm. This is done by setting μ_{init} to w_{init} where w_{init} are the initial weights and μ_{init} can be found in Algorithm 4.1. These weights can then be optimized according to a pre-defined cost function which calculates the cost. By using the initial weights the policy search gets a good initial starting point for exploring.

5.1.2 2-DoF example

In order to illustrate the use of initializing the policy search with imitation learning, a 2 Degree of Freedom example is used. Hereby we start with a trajectory in the x, y-plane where the x and y trajectory are encoded in a DMP with both their own transformation system. The original trajectories and the resulting DMPs without learning can be found in Figure 5-1

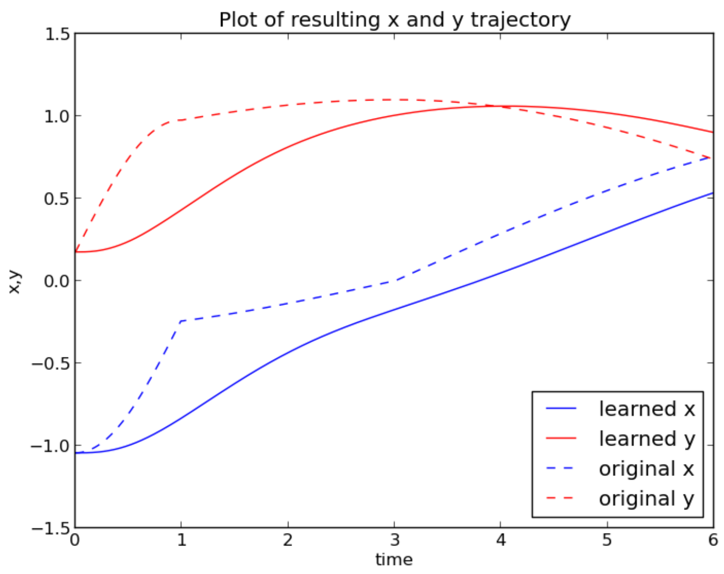


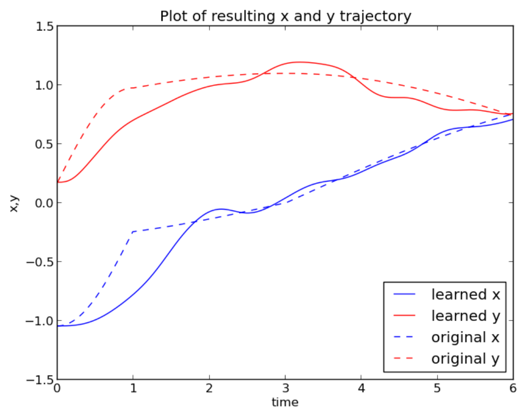
Figure 5-1: x, y-plot of original trajectories and trajectories generated by learned DMPs of the 2-DoF example

These trajectories are then optimized using the basic and symmetric PGPE algorithm according to the cost function defined in Equation (5-1) where N is the number of trajectory points. Here the learned trajectory has to follow the demonstrated trajectory as best as possible but at least it should reach its goal.

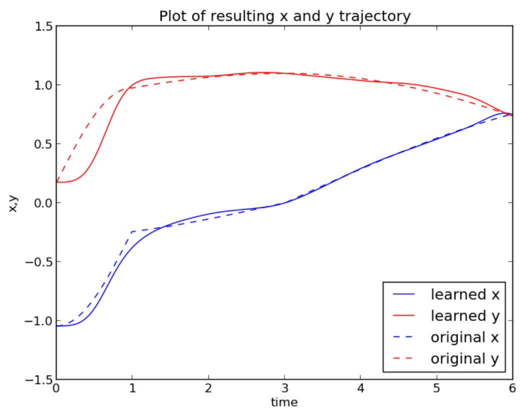
$$\begin{aligned} J = & \sum_{i=0}^{N-1} [-(x_{demo}(i) - x_{act}(i))^2 - (y_{demo}(i) - y_{act}(i))^2] \\ & -\kappa(x_{demo}(N) - x_{act}(N))^2 + (y_{demo}(N) - y_{act}(N))^2 \end{aligned} \tag{5-1}$$

Where κ is a variable that determines the importance of the last term and was chosen to be 1000. The parameters that are used for learning are the weights and the goals of the two DMPs.

To give an impression about the results after applying the two PGPE algorithms, in Figure 5-2 the resulting trajectories can be found. In Figure 5-3 the corresponding cycle-cost plot can be found. It can be seen that the symmetric algorithm needs more cycles to converge but that it obtains a lower cost.

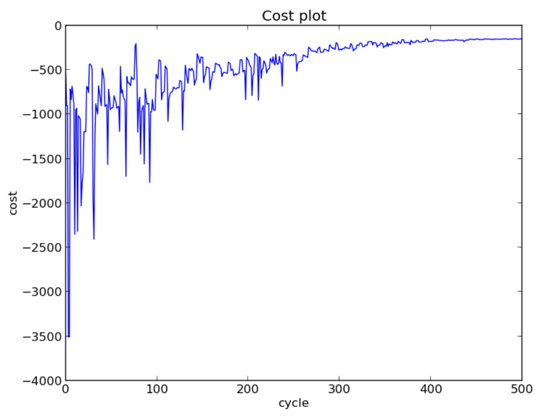


(a) Basic PGPE algorithm

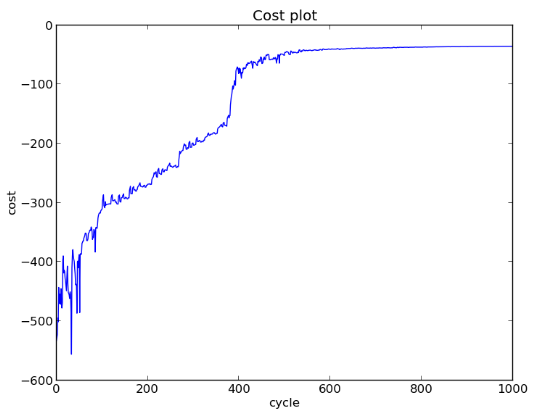


(b) Symmetric sampling PGPE algorithm

Figure 5-2: Resulting trajectories after learning



(a) Basic PGPE algorithm



(b) Symmetric sampling PGPE algorithm

Figure 5-3: Cycle-cost plot

5.2 Optimizing sequences of Dynamic Movement Primitives

Until now it was assumed that the whole demonstration is captured in just one DMP. However as suggested by [72] a movement can also consist of a sequence of DMPs. This is simply done by scheduling the DMPs after each other where the next DMP takes as x_0 the goals of the current DMP. An algorithm can then be used to optimize the sequence of DMPs. The algorithm that is used in [72] is PI^2 which is a policy improvement algorithm.

In this thesis also a sequence of DMPs will be used. However, because we are using a different algorithm we have to slightly adjust the way that a sequence of DMPs is optimized. In the PI^2 algorithm costs can be rewarded during the execution of the trajectory, this is visually shown in Figure 5-4. It can be seen that the weights of the DMPs are adjusted during execution while the goals are adjusted at the end of the execution. It can also be seen that the costs of the next DMPs are also used for improving the current goal.

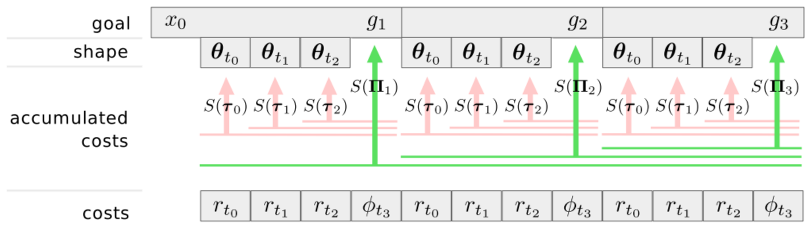

goal
shape
accumulated costs
costs
x_0
g_1
g_2
g_3
θ_{t_0} θ_{t_1} θ_{t_2}
$S(\Pi_1)$ $S(\Pi_2)$ $S(\Pi_3)$
$S(\tau_0)$ $S(\tau_1)$ $S(\tau_2)$
r_{t_0} r_{t_1} r_{t_2} ϕ_{t_3}

Figure 5-4: Cost rewarding of [72]

The aforementioned idea of incorporating the costs of the next DMPs is also used in this thesis. However there is just one kind of cost because it is not possible to reward costs while executing a DMP due to the PGPE algorithm. Therefore only the total cost of a DMP is used. For updating the weights only the total cost of the trajectory of the corresponding DMP is used while for updating the goals the total cost for the current trajectory and all next trajectories is used. The reason behind using the accumulated cost for the goals is that the goals of the current DMP also form the starting point of the next DMP and thus influences the costs of the next primitive(s). When we redraw Figure 5-4 according to the above mentioned changes, this results in Figure 5-5.

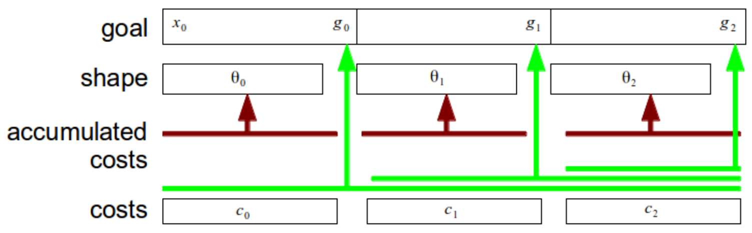


Figure 5-5: Cost rewarding used in this thesis

Sequence of DMPs in 2-DoF example

We applied the above theory to the same case as described in Section 5.1.2. But instead of using one DMP for the whole trajectory, the trajectory is split in three intervals which all get their own DMP. The result, without learning, can be seen in Figure 5-6 where the chosen intervals are indicated by the vertical lines.

To also show the difference with one DMP in learning, in Figure 5-7 the result after 500 iterations of the symmetric sampling PGPE can be found. It can be concluded that using a sequence of DMPs leads to a lower cost and a faster convergence for a 2-DoF case.

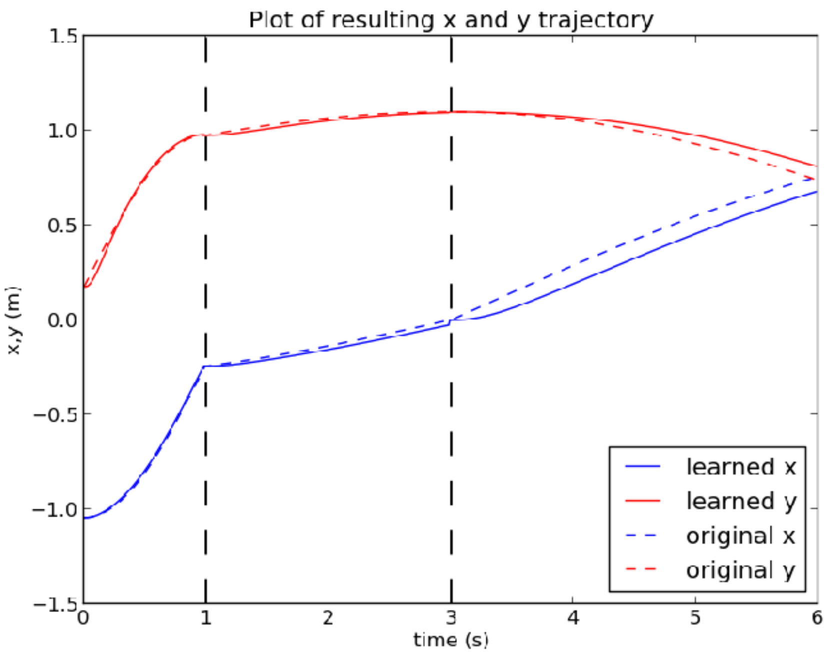
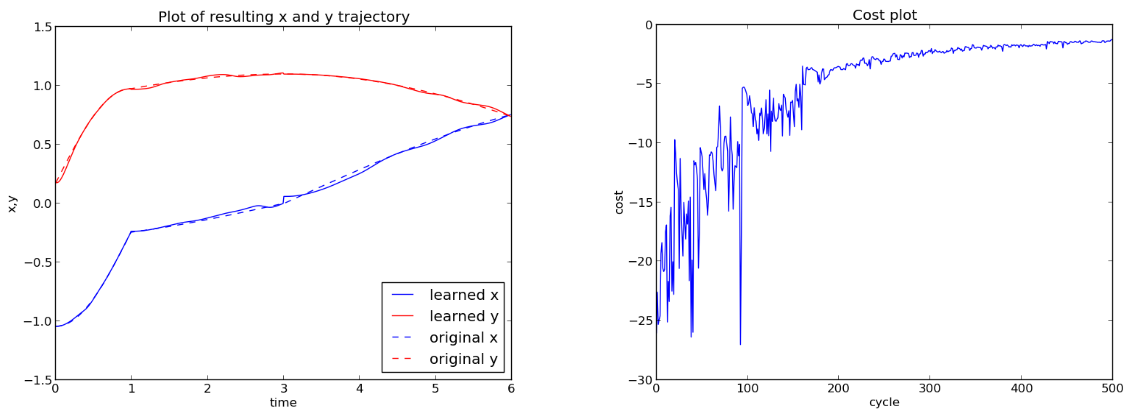


Figure 5-6: x, y-plot of original trajectories and trajectories generated by learned DMPs of the 2-DoF example using a sequence of DMPs



(a) Resulting trajectories **(b)** Cycle-cost plot for symmetric PGPE

Figure 5-7: Results after 500 iterations for learning of a sequence of DMPs

Conclusion

The weights of the DMPs can be used as a parametrized policy for the policy search algorithm. Then the algorithm can be used to alter the trajectories according to a cost function. From the 2 DoF case it can be concluded that optimizing a sequence of DMPs using the symmetric sampling version of the algorithm leads to a lower cost and a faster convergence compared to using a single DMP.

Chapter 6

Use-case description: precision placement task

In this chapter the use-case that forms the basis for the research in this thesis is explained. In Section 6.1 the setting of the use-case is described. Subsequently in Section 6.2 the problem is described. Finally in Section 6.3 the benefits of using imitation learning for this use-case are given.

6.1 Setting

This use-case is derived from a process in a consumer product factory. For a specific product parts are painted/lacquered in a paintshop. About 40 different parts of the final product are handled in this process. All parts are supplied in trays which have separate compartments for each separate part, a schematic overview is given in Figure 6-1. In the current situation these parts are picked by a human and placed on a jig. These jigs are attached to a carousel which can be rotated to place parts on all jigs. These carousels are on poles which are attached to a chain conveyor. This conveyor moves with intervals of a certain amount of time. Once in around half an hour the kind of part that is placed on the carousels is changed. Because the jigs on the carousels are made for a specific part, the carousels have to be changed.

The aforementioned placing of the part on the jig is not a trivial movement. Most parts have to be attached to the jig with a movement that is specific for that part. In Figure 6-2 an example of a movement of placing a part on a jig that is attached to a carousel is shown. This movement depends greatly on the part that is handled. Most movements result in a 'click'.

After the placing of the parts on the carousels the parts move through the whole painting process and finally come to a stage where the parts are removed from the jigs and placed back in the trays.

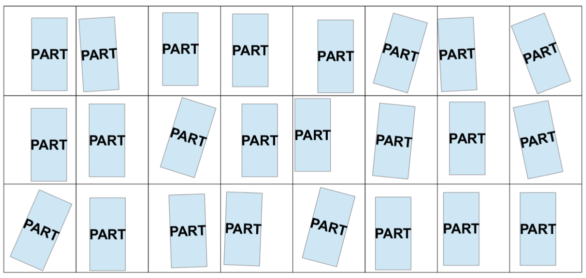


Figure 6-1: Overview of supply of parts in a tray

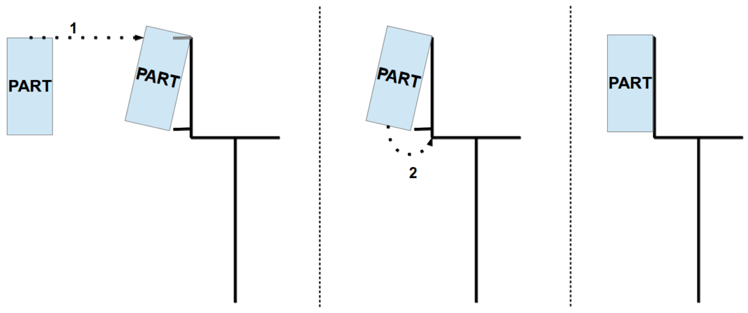


Figure 6-2: Schematic of placing a part on a jig that is attached on a carousel

6.2 Problem description

6.2.1 Desired situation

A desired situation for the situation described in Section 6.1 is that the process of placing and removing the parts to and from the jigs is robotised. Because there are multiple sub processes in this process, it is decided to focus on the sub process of picking the object from the trays and placing them on the jig. As explained before the placement on the jig is not a trivial place movement. Therefore we call this use-case a precision placement task.

This thesis thus focusses on the case of picking a part from the tray, moving it to the jig and placing it on the jig. This should all be done with a robot arm. The 'programming' of the robot should be done with the use of imitation learning. The robot arm should be able to handle each of the about 40 parts. An extra requirement is that new parts can be easily introduced in the system through imitation learning.

6.2.2 Assumptions

In order to make this case fit for this thesis, some assumptions have to be made. The assumptions that are made are:

- The parts in the trays are not flipped. The only variance in the configurations is as depicted in Figure 6-1.

- The parts are already recognized by some means and the Cartesian coordinates are known.
- The parts are assumed to be in the gripper of the robot arm at the start of the movement, when the arm is at the above mentioned coordinates.
- The demonstrated trajectory is available in Cartesian coordinates with corresponding time stamps or sampling rate.
- The Cartesian coordinates of the jig are known.
- The carousel does not have to be rotated
- Only jigs that are reachable without rotating the carousel have to be filled.
- The demonstrated trajectory is not necessarily optimal with relation to energy consumption, execution time and joint trajectories.

To summarize these assumptions: this use-case focusses purely on the trajectory. No work is done in object recognition, grasping techniques, robot grippers etcetera.

6.3 Benefits of imitation learning

The reason that imitation learning is chosen to solve the described problem is that it has some benefits over other solutions. The biggest benefit is that new parts can be introduced easily: a new trajectory for a part can be taught by a non-robot expert. As a result there is no longer a need for (expensive) robot experts. Another benefit is that the robot is more flexible and, if the algorithm is implemented successfully, the robot can also be applied to other tasks than precision placement.

Summary

As use-case a precision placement task is chosen. A part has to be placed on a jig that is attached to a carousel. The use-case purely focusses on learning the trajectory that is needed for the successful placement of the part.

Chapter 7

Implementation for use-case

In this chapter the implementation of the previously described methods and techniques is explained. Therefore it is first described how the trajectories are captured and subsequently encoded and used (Section 7.1). Then the application of Policy Gradients with Parameter based Exploration (PGPE) to the use-case is explained in Section 7.2. Finally in Section 7.3 it is elaborated on how the robot is controlled. For the application the UR5 robot from Universal Robots [73] is used.

7.1 Encoding the trajectories

7.1.1 Gathering trajectory data

In order to gather the data that is needed for encoding the trajectories, the aforementioned UR5 [73] is used. The UR5 robot can be put in teach-in modus which enables the user to move all the joints of the robot by gripping the joint that has to be moved and moving it in the desired state. Putting the robot in teach-in modus is done by holding a button at the back of the control panel of the robot (see Figure 7-1). This has as disadvantage that only one hand can be used to move the robot around.

The data is gathered by using a MATLAB driver for the UR5 that is available at Delft University of Technology (TU Delft). This driver enables you to specify the sample frequency and the time it has to capture data. All data that is available at the control panel of the robot can be read out e.g. joint positions, joint speeds and tool position. The data that was gathered in this thesis are the position of the tool in `x`, `y`, `z` coordinates and for the rotation `rx`, `ry` and `rz`. The `x`, `y`, `z` coordinates are in the reference frame of the `base_link` of the robot and `rx`, `ry`, `rz` is an axis-angle representation of the rotation. The reference frame can be found in Figure 7-2. Note that the robot in itself is symmetric but that the side of the wire can help as orientation point. In Figure 7-2 the wire is in the green `y` axis.

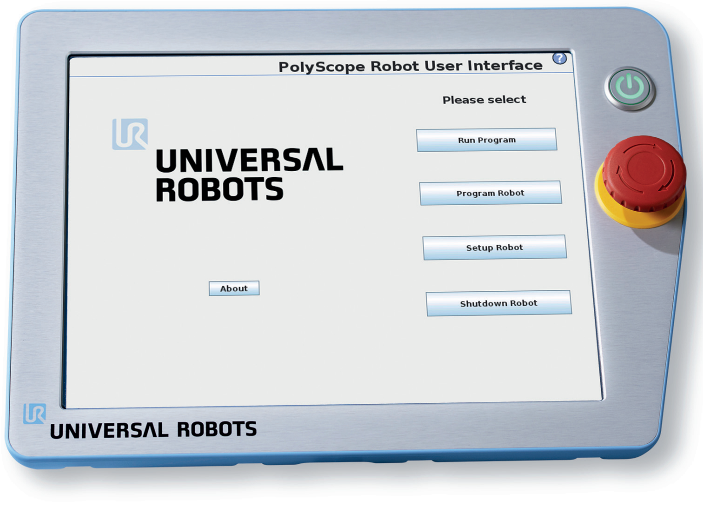


Figure 7-1: Control panel of the UR5 [73]

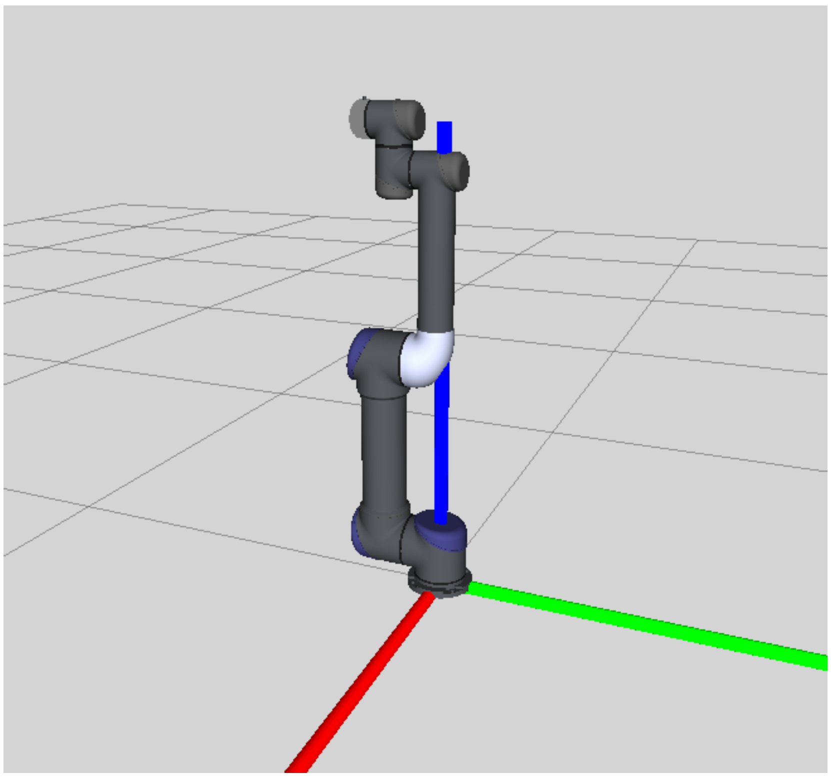


Figure 7-2: Coordinate frame as defined in MATLAB and on the UR5: x (red), y (green), z (blue)

7.1.2 Conversion from axis-angle representation to roll, pitch, yaw

`Roll`, `pitch` and `yaw` are a common representation for the orientation of an object. The MATLAB driver is able to output these values. Therefore, and because `roll`, `pitch`, `yaw` is an intuitive representation, the implementation of the algorithms was based on using `roll`, `pitch` and `yaw` as representation for the rotation. However after testing it appeared that the `roll` value was not recorded correctly, it was always 0.0.

Therefore it was decided to convert the `rx`, `ry`, `rz` representation to `roll`, `pitch`, `yaw` representation. Here `rx`, `ry` and `rz` are an axis-angle representation for the rotation. This representation is also used by the UR5 on the control panel. Equation (7-1) describes this angle-axis representation. This can also be written as a normalized vector $\hat{\boldsymbol{r}}$ times an angle θ as can be seen at the right in Equation (7-2), the angle θ is equal to the length of $\boldsymbol{r}$. A visualization of a rotation represented by the axis-angle notation can be found in Figure 7-3

$$\boldsymbol{r} = \begin{bmatrix} rx \\ ry \\ rz \end{bmatrix} \tag{7-1}$$

$$\boldsymbol{r} = \theta\hat{\boldsymbol{r}} \tag{7-2}$$

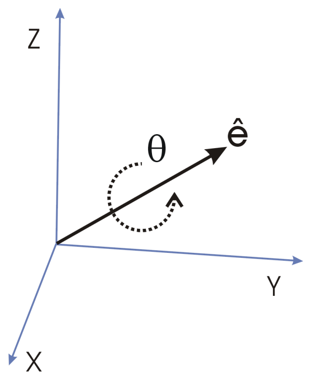


Figure 7-3: Axis-angle representation of a rotation [74]

The conversion to `roll`, `pitch` and `yaw` can be done through writing down the rotation matrices for both representations and use this to convert one into the other. This is explained in Appendix C, here only the resulting formulas are given. It should be noted that there are singularities in the conversion, in Appendix C it is explained how these can be prevented. In Equation (7-3), Equation (7-4) and Equation (7-5) the formulas for `roll`, `pitch` and `yaw` can be found respectively. Here f, g and h are the normalized values of rx, ry, rz and θ is the above mentioned angle.

$$\texttt{roll} = \text{atan2}((1-\cos(\theta))gh + f\sin(\theta), (1-\cos(\theta)h^2 + \cos(\theta)) \tag{7-3}$$

$$\texttt{pitch} = \text{asin}(-(1-\cos(\theta))fh + g\sin(\theta)) \tag{7-4}$$

$$\texttt{yaw} = \operatorname{atan2}((1-\cos(\theta))fg + h\sin(\theta), (1-\cos(\theta))f^2 + \cos(\theta)) \tag{7-5}$$

7.1.3 Conversion to ROS format

The algorithms and control of the robot are all implemented in the Robot Operating System (ROS) framework. This is an open-source framework for writing robot software, see Appendix A for more information on ROS. In order to use the data from MATLAB in ROS, the `x`, `y`, `z`, `roll`, `pitch` and `yaw` values are exported to a `.csv`-file. Which can be imported in ROS. The data is then converted to a data-type that is used by a package for encoding Dynamic Movement Primitive (DMP)s. This data-type is called `DMPTraj` which consists of `points` and `times`. In one `point` all the trajectory data at a certain time are stored while the corresponding timestamp is stored in `times`. So in this case a `point` entry in `points` consists of six 'positions' namely `x`, `y`, `z`, `roll`, `pitch` and `yaw` while the corresponding timestamp is stored at the same entry in `times`.

7.1.4 Applied DMPs

For the application to the use-case a 6 DoF DMP is used. In Section 3.1 it was already explained how such a DMP is constructed. The used variables for the DMP are `x`, `y`, `z`, `roll`, `pitch` and `yaw` as mentioned above.

For encoding trajectories an existing ROS package was used [75]. However this package was written for a previous ROS version and had to be updated to ROS Hydro (the ROS version used in this thesis). Working with the package, it was discovered that there were some bugs and short comings. The most missing feature was that there were no Gaussian basis functions used as basis functions for the DMPs. It was decided to improve this package and implement Gaussian basis functions. The Gaussian basis function used is of the form in Equation (7-6).

$$\psi_i = \exp\left(\frac{-0.5(x-c_i)^2}{w_i^2}\right) \tag{7-6}$$

Here the centers c and widths w are determined using a function that is used in [76] for determining the meta parameters of Locally Weighted Regression (LWR). A specified number of basis functions is placed between a given `min` and `max` value thereby taking into account the height at which two neighbouring basis functions should intersect. Thus the input of the function is: `num_basis_functions`, `min_value`, `max_value`, `intersection_height`. The `num_basis_functions` are linearly spaced between `min_value` and `max_value` to get the centers c. The width w_i is determined by:

$$w_i = \sqrt{\frac{(c_{i+1}-c_i)^2}{-8log(h)}} \tag{7-7}$$

Here h is the `intersection_height`. In this thesis there is chosen for an `intersection_height` of 0.5. The number of used basis functions is 15.

Locally Weighted Regression is used to determine the weights of the basis functions. LWR is very similar to simple regression the only difference is that a weight matrix is introduced. The simple regression is written as:

$$\beta = (X^T X)^{-1} X^T y \tag{7-8}$$

In LWR a weight matrix is added to the equation [77, 78]. The formula then becomes:

$$\beta = (X^T W X)^{-1} X^T W y \tag{7-9}$$

The basis functions are evaluated for each input and used to determine the weight matrix W.

7.1.5 Splitting the trajectories

As explained in Section 5.2 the trajectories can be split in multiple sub-trajectories and for each sub-trajectory a DMP can be learned. Then the total trajectory is constructed by a sequence of DMPs. The splitting is now done by plotting the captured data and setting the splits such that resulting sub-trajectories can be easily approximated by the DMPs. The reason that this is done by hand is that splitting the trajectory automatically is very hard. By doing it by hand, and because the trajectories are relatively simple, the human intuition on smart sub-trajectories can be incorporated.

7.2 Application of PGPE

7.2.1 Instances of PGPE

For each DMP, and thus sub-trajectory, two instances of the PGPE algorithm are created. One is for learning the weights of the DMP and one for learning the goals of the DMP. As mentioned in Section 7.1.4 a 6 DoF DMP is used. This means that the number of weights is six times the number of basis functions. It was decided to learn all these weights using one instance of the PGPE algorithm. There are two reasons for that: first from a software point of view it is easier to learn all weights using one instance; second the 2 DoF example of Chapter 5 did not show any difference in performance of the algorithm. The six goals that the 6 DoF DMP has, are learned by the other instance of the PGPE algorithm.

An important parameter in the instantiation of the PGPE algorithm is the σ_{init} value. As can be seen in Algorithm 4.2, this value has to be set by hand. In the literature there is no procedure described how to determine this value. Therefore in this thesis these values were determined empirically. This value determines the width of the initial normal distribution where the perturbations ϵ are drawn from. When σ_{init} is chosen very high, the normal distribution becomes very wide and there is a wide range of values with a high chance of being drawn. In the opposite case when this value is very small, the range of values that have a high chance of being drawn is very narrow.

The σ_{init} in this thesis are chosen as follows: for the trajectories σ_{init} is 0.02 and for the goals σ_{init} is 0.01. These values are chosen based on the experiments, they can be explained when looking what the σ_{init} determines. The σ_{init} determines the width of the initial normal distribution and thus the range of values that can be picked. Because the weights are in the order of

0.1 to 1.0, small perturbations can cause a big change. Furthermore the initial demonstration gives a good first approximation of the desired trajectory. Based on the combination of these two facts it can be seen that small values for the σ_{init} are reasonable.

Because the perturbations ϵ are drawn from a normal distribution, there is a small chance of getting an extreme value. An extreme value results in a huge perturbation which might lead to an infinite cost. Therefore it was decided to limit the drawing of the ϵ to a range of plus and minus three times the standard deviation σ. If the drawn perturbation ϵ is outside this range, a new value is drawn. A range of three times the standard deviation is based on the fact that hereby 99.7 % of the values of the normal distribution are covered as illustrated in Figure 7-4.

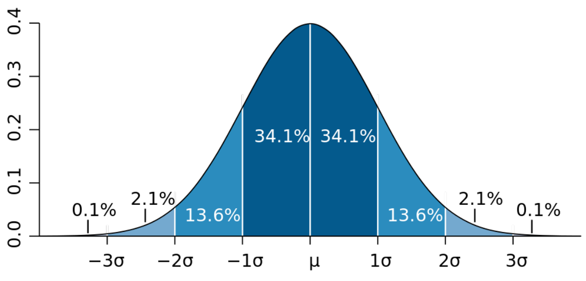


Figure 7-4: Normal distribution curve that illustrates standard deviations [79]

7.2.2 Cost function

The cost function used for the use-case consists of three sub-costs. The cost function was designed based on what is important from the practical view. This resulted in three factors and hence three sub-costs: 1) in the beginning of the movement the demonstrated trajectory has to be followed but deviations are allowed, 2) in the end of the movement the demonstrated trajectory is very critical to follow and only small deviations are allowed and 3) the part should be placed on the jig. To summarize this in a cost equation:

$$J = -\sum_{i=0}^{K-1}[\delta_a(\boldsymbol{f_{demo}}(i) - \boldsymbol{f_{act}}(i))^2] - \sum_{i=K}^{K+N-1}[\delta_b(\boldsymbol{f_{demo}}(i) - \boldsymbol{f_{act}}(i))^2] \\ - \delta_c[\kappa + \lambda(\boldsymbol{f_{demo}}(K+N) - \boldsymbol{f_{act}}(K+N))^2)] \quad (7\text{-}10)$$

Here $\boldsymbol{f}$ contains the values of the trajectory, in this case `x`, `y`, `z`, `roll`, `pitch` and `yaw`. The values of the demonstrated and actual trajectory are compared. The difference is squared for three reasons: the number should always be positive, the larger the difference the bigger the cost should be and it is most seen in literature e.g. [72]. K is the number of points in the sub-trajectories that are not the final sub-trajectory. N is the number op points in the final trajectory. The δ's represent a condition in this case. When the learned trajectory deviates more than 15% from the demonstrated trajectory, δ_a is 1 otherwise it is 0. The same holds for δ_b but there is the percentage 5%. For the last δ_c it holds that it is 1 when the part is not placed, otherwise it is 0. The variables κ and λ are weighting factors. For this use-case κ was 1000 and λ 300. These numbers were chosen because the placement and the deviation from the final position are seen as important.

7.3 Controlling the UR5

7.3.1 Control of the UR5 in ROS

The UR5 can be controlled in ROS by using the motion planner of ROS, *MoveIt!* [80]. *MoveIt!* is state of the art software for mobile manipulation, more information on *MoveIt!* can be found in Appendix B. A robot can be controlled through *MoveIt!* in several ways: providing a desired joint configuration, providing a desired position and quaternion of the end effector or providing a sequence of desired waypoints. In the first two cases *MoveIt!* uses a stochastic planner to determine a path that reaches the desired configuration. In the last case the planner tries to fit a path trough the waypoints, this will be further explained in Section 7.3.3.

The first two ways of controlling the UR5 are used for initializing the robot to a standard configuration and moving it to the start pose for learning and executing. When providing a desired position and quaternion (for the rotation) of the end effector the corresponding reference frame has to be set. Otherwise the planner does not know relative to what the position is provided. In order to take safety into account, each planned trajectory is first shown in simulation. Then the user is asked if the shown trajectory has to be executed, if the answer is 'no' the planner plans a new trajectory. If the answer is 'yes' the planner will execute the shown trajectory on the real robot.

7.3.2 Rotation conversion from UR5 to ROS

In Section 7.3.1 it was explained that a robot can be controlled by passing a `Pose` to the motion planner. This `Pose` consists of a `Position` and `Orientation`. The quaternion that is needed for the `Orientation` can also be set by using `roll`, `pitch` and `yaw`. ROS then converts these values to a quaternion. After simulation and testing it appeared that setting an orientation in this way does not result in the same orientation as desired while the same reference frame was used. We could compare this because we know both the quaternions in ROS and the measured values for `rx`, `ry` and `rz` from the demonstrations.

After research and testing it appeared that the conversion from `roll`, `pitch`, `yaw` to a quaternion is correct in ROS. However the problem is in how the rotation for the **0** vector is defined. If we take $rx = 0$,$ry = 0$, $rz = 0$, this is converted to the quaternion $1 + 0i + 0j + 0k$. If we set the following values on the control screen of the UR5: $x = 0$, $y = -0.4$, $z = 0.5$, $rx = 0$,$ry = 0$, $rz = 0$ this results in the pose of Figure 7-5a. While if we use ROS to go to the same position and use the quaternion $1+0i+0j+0k$ to set the orientation this results in the pose of Figure 7-5b. So where the zero orientation on the UR5 is defined looking up along the z-axis, in ROS it defined as looking forward along the x-axis.

Without a right conversion the application that was built, is worthless. Because all code was in the ROS framework while the demonstrations could only be captured using the MATLAB driver. The problem could not be solved by performing transformations between different reference frames. Therefore it was decided to look into the resulting quaternions. The orientation of the $rx = 0$,$ry = 0$, $rz = 0$ pose is in ROS seen as $0.5 + 0.5i - 0.5j + 0.5k$.

The problem can be solved by multiplying the quaternion $q = 1 + 0i + 0j + 0k$ with another quaternion $r = r_0 + r_1i + r_2j + r_3k$ to get $t = 0.5 + 0.5i - 0.5j + 0.5k$. We then get $t = q\text{x}r$,

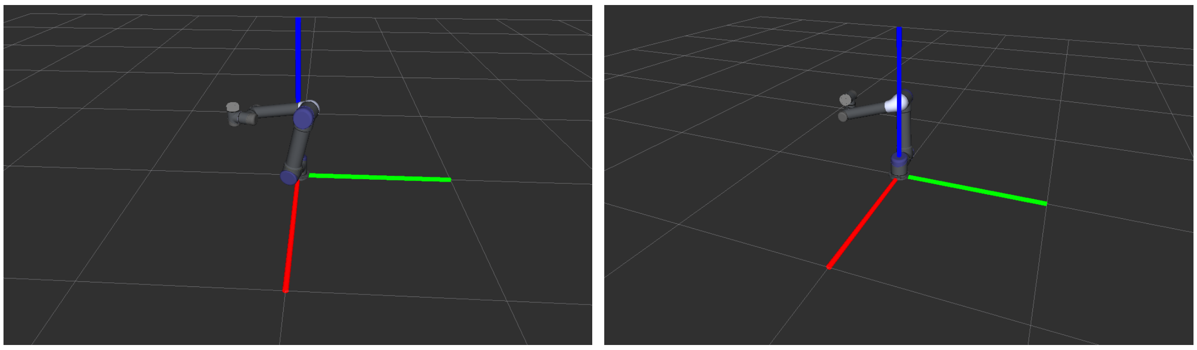

(a) UR5 **(b)** ROS

Figure 7-5: Zero orientation

with t:

$$\begin{aligned}
t_0 &= r_0q_0 - r_1q_1 - r_2q_2 - r_3q_3\\
t_1 &= r_0q_1 + r_1q_0 - r_2q_3 + r_3q_2\\
t_2 &= r_0q_2 + r_1q_3 + r_2q_0 - r_3q_1\\
t_3 &= r_0q_3 - r_1q_2 + r_2q_1 + r_3q_0
\end{aligned}$$

Because we know t and q, we can compute r. We then get $r = 0.5 - 0.5i + 0.5j + 0.5k$. Multiplying the quaternion that results from the conversion of the UR5 values with r gives the correct pose in ROS.

7.3.3 Executing a path

The aforementioned learned DMPs are used to generate a path that can be executed by the robot. The earlier mentioned ROS package is used to generate this path, also called plan. This is done by just filling in all values in Equation (3-1). This results in a vector with `points`, `velocities` and `times`.

For the control of the UR5 we can only use the `points` at this moment, due to limitation in the implementation of *MoveIt!*. The points of the DMPs are converted to `positions` and `orientations` which together form a `Pose`. These `Poses` are put in a vector. The vector can then be used as input for the function `computeCartesianPath()` which converts the `Poses` to a path for the end effector in the Cartesian space. This path can then be supplied to *MoveIt!* which executes the path on the robot.

To prevent that a lot of unusable trajectories are executed on the UR5, a check is done before sending the path to the `computeCartesianPath()` function. The end points of the path that has to be sent to `computeCartesianPath()` are checked for being within a certain margin of the desired end points. Because if the end points are outside this margin, it can be sure that the placement is unsuccessful. In this specific case the values in Table 7-1 are chosen as margin values. These have been determined empirically.

Table 7-1: Margins used for precision placement application

Variable	Margin
`x`	0.013 [m]
`y`	0.013 [m]
`z`	0.013 [m]
`roll`	0.05 [rad]
`pitch`	0.05 [rad]
`yaw`	0.07 [rad]

Summary

The trajectory data of the demonstrations is recorded by means of a MATLAB driver for the UR5. The `x`, `y`, `z`, `rx`, `ry` and `rz` values are recorded. Where `rx`, `ry` and `rz` is an axis-angle representation of the rotation which is converted to `roll`, `pitch` and `yaw`. These values are used to encode the DMPs, LWR is used as regression technique for determining the initial weights.

For each sub-trajectory two instances of the PGPE algorithm are created: one for the weights of the DMP and one for the goals. The perturbations are drawn in a limited region to create a more stable algorithm. The cost function that is used for learning is based on following the demonstrated trajectory, the deviation from the final position and if the part is placed.

The UR5 is controlled using the motion planner of ROS, *MoveIt!*. The path created by the DMPs are supplied to the path planner as input which tries to convert this to an executable path for the robot.

Chapter 8

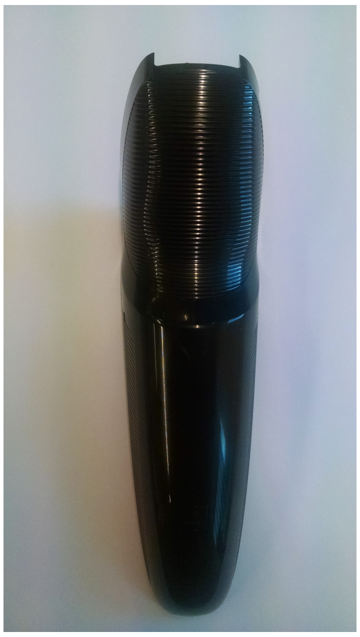
Experiments

Before starting to describe the results of the experiments, in Section 8.1 the test setup will be described. Then in Section 8.2 the results of the placement of the cover part will be given. Subsequently in Section 8.3 the results of the placement of the connection cover part will be given.

8.1 Test setup

Two of the parts supplied by Philips Consumer Lifestyle are used for the test of the precision placement task. These parts are chosen because the movement that is needed for the placement is non-trivial and because these parts can be easily mounted on the robot without blocking elements of the part that are needed for the placement. The parts that are used for the placement, are a cover and a part that covers a connection. In Figure 8-1 the cover part is shown and in Figure 8-2 the connection cover part is shown. In Figure 8-3 it is shown how the parts are attached to a wooden board which on its turn can be attached to the end effector of the robot.

The parts need to be placed on a jig which is attached to a carousel. These carousels for the two different parts are shown in Figure 8-4. The carousel with the jigs, is attached to a PVC pipe which is attached to a wooden block. This wooden block is then clamped to the table by two (glue) clamps. An overview of the whole setup can be found in Figure 8-5, a close-up can be found in Figure 8-6.


(a) Front

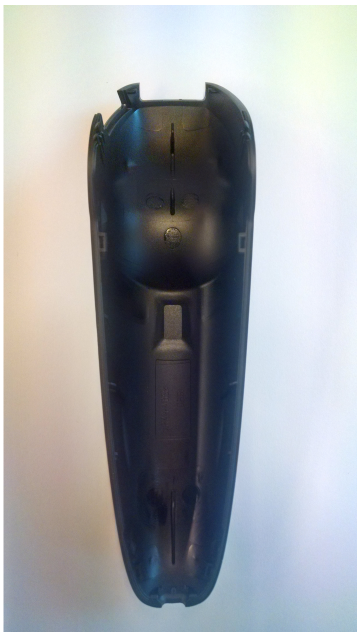
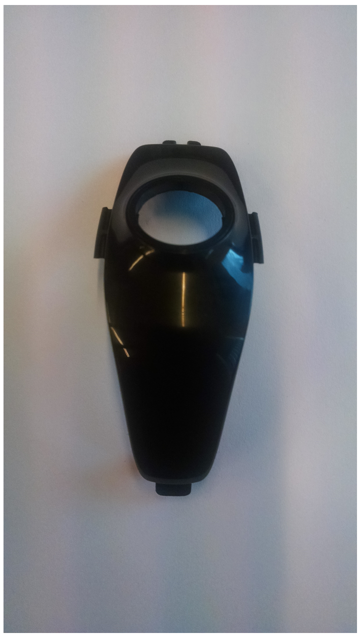
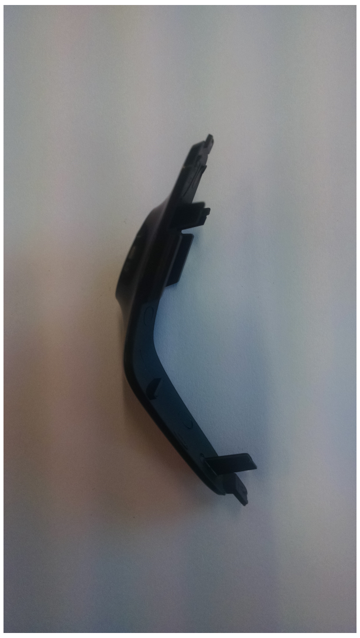

(b) Back

Figure 8-1: Cover part


(a) Front


(b) Side

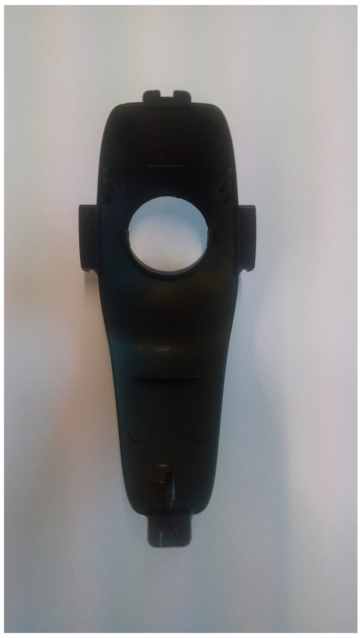
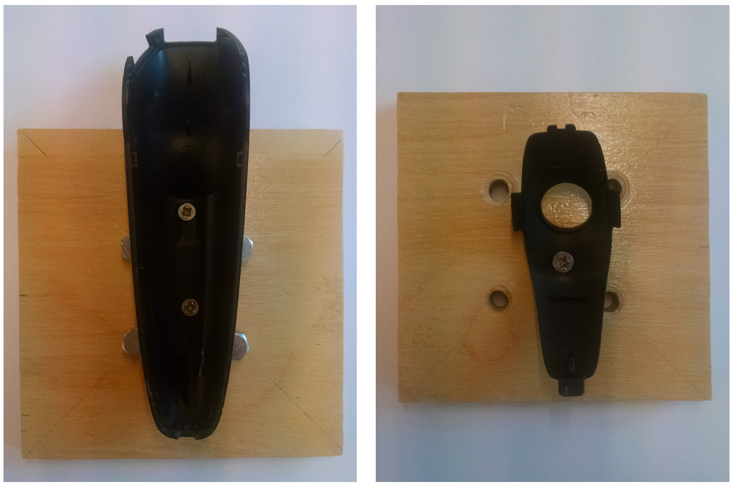
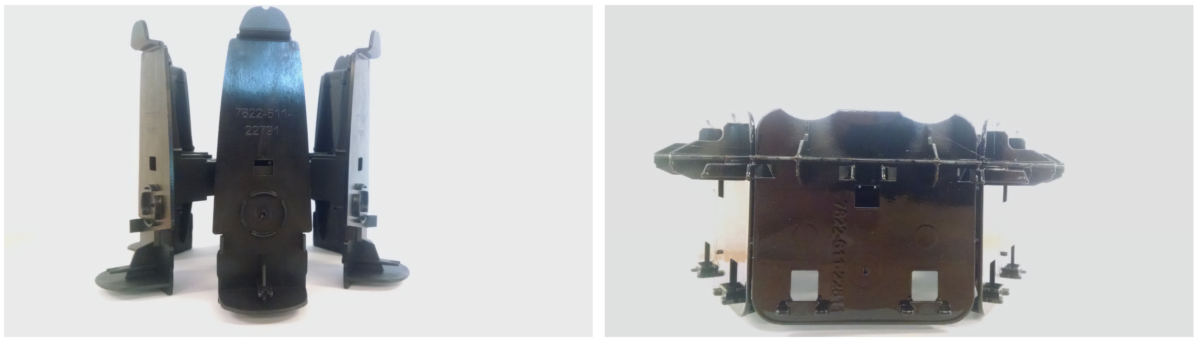

(c) Back

Figure 8-2: Connection cover part



(a) Cover **(b)** Connection cover

Figure 8-3: Parts attached to wooden board



(a) Cover **(b)** Connection cover

Figure 8-4: Carousels with jigs

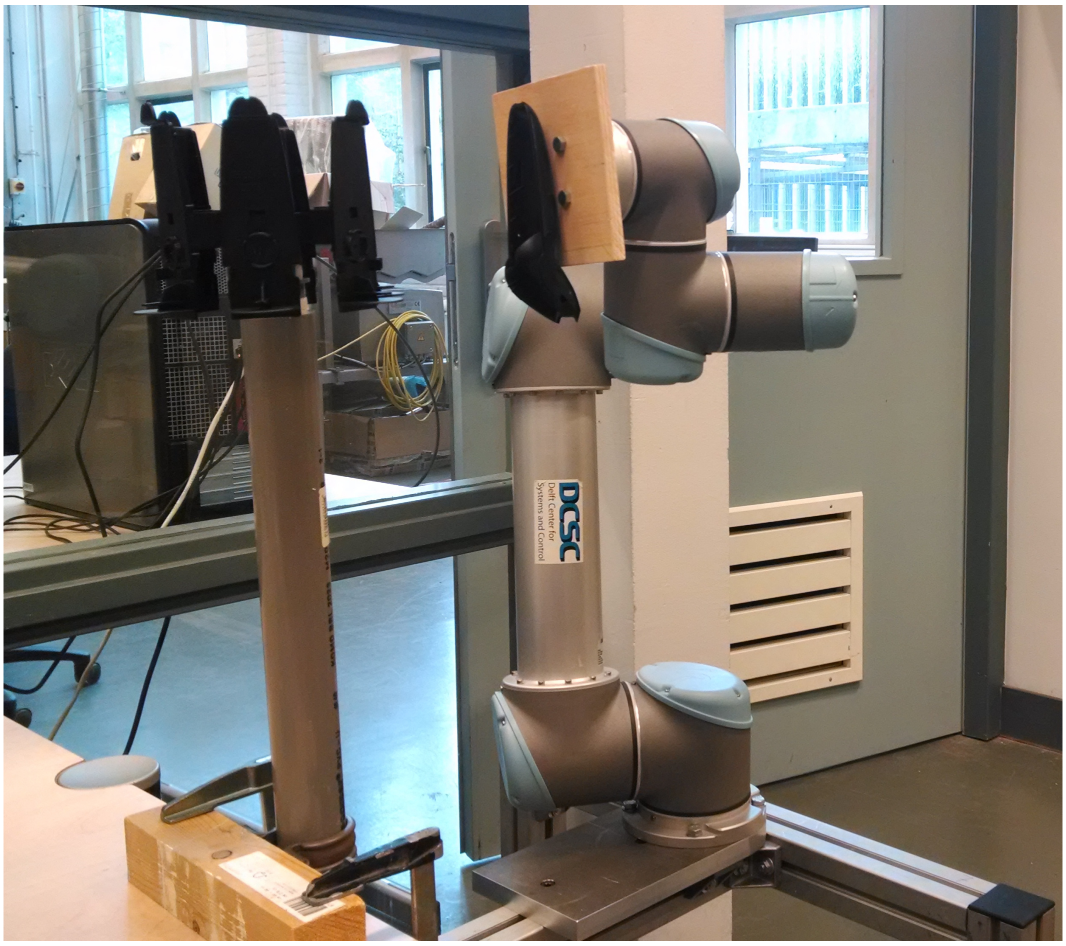


Figure 8-5: Overview of the test setup

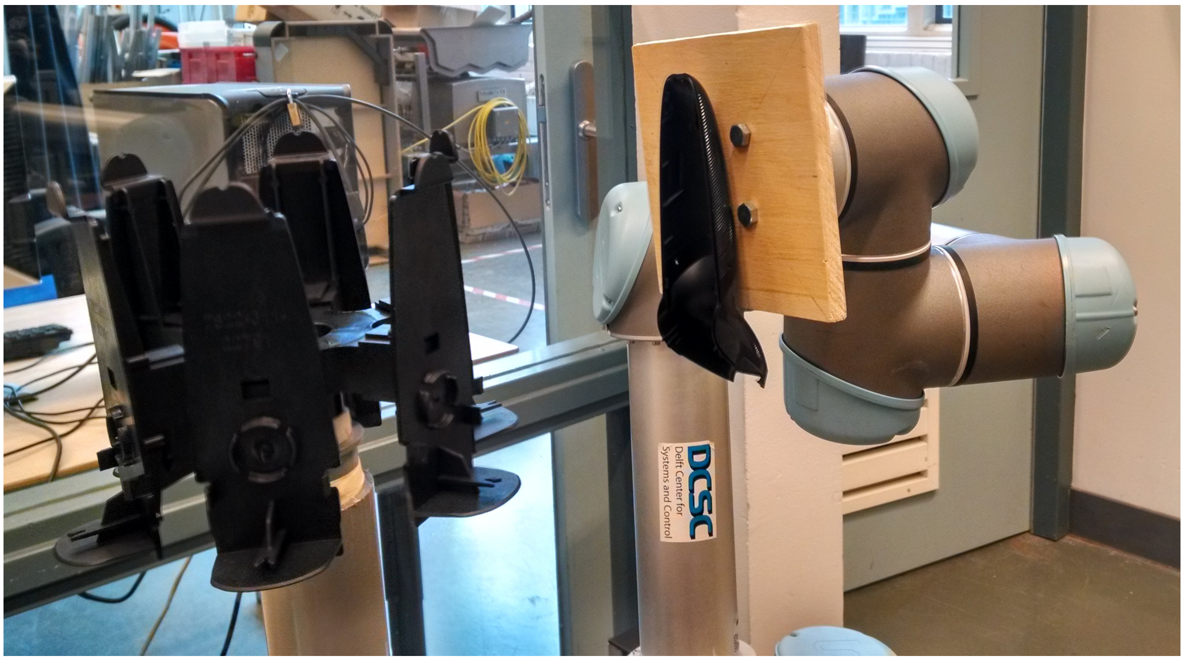


Figure 8-6: Close-up of the setup

8.2 Placement of cover part

8.2.1 Test method

The software that is described in Chapter 7 is tested in this section. Therefore first a demonstration of the placement is given using the teach-in mode of the UR5. The cover part is placed as is shown in Figure 6-2. First the bottom of the cover has to be placed over the top of the jig and then a downward turning movement has to be made to 'click' the top of the part into place. It should be noted that the part is placed upside down on the jig so top and bottom refer to the orientation as shown in the figures. The demonstration is given such that the movement is easy to perform by the demonstrator and is only based on human intuition. The demonstration is not necessarily fluent however the movement always results in a successful placement. The demonstration is then split in sub-trajectories and given as input to the software.

The demonstrated trajectory was split in four sub-trajectories based on the plot of the values. The resulting sub-trajectories are as depicted in Figure 8-7. 1) is chosen because there is a non-trivial change in `yaw`, 2) is chosen because there is a change in `roll`, 3) and 4) are chosen because there is again a big change in `roll`. The last part is not used because the object was already placed but the recording was still running.

8.2.2 Results of normal demonstration

A sequence of Dynamic Movement Primitive (DMP) results from the above mentioned sub-trajectories. A surprising observation was that the initial DMPs that were learned from these sub-trajectories already resulted in a successful placement. It was not expected before hand that the initial DMPs would be sufficiently shaped to do a successful placement. A plot of the trajectories that were demonstrated and the trajectories that were created by the initial DMPs can be found in Figure 8-8. Since the placement is already successful with the initial DMPs the learning step is not needed in this case.

In order to figure out if the learning step could improve the trajectory of a successful placement, the cost function was altered. A term was included that accounts for the jerk. Jerk is the third derivative of the position. To move smoothly from one point to another, the sum of the squared jerk along the trajectory should be minimized [81]. The jerk was computed for each sub-trajectory. However it appeared that the jerk could not be minimized further. The last two sub-trajectories can not become minimum jerk because the specific 'click' movement has to be made which is not a smooth movement. The first two sub-trajectories appear to be relatively minimum jerk already which can be explained by the fact that humans move with minimum jerk and thus a demonstration might already be minimum jerk. Overall the learning algorithm could not reduce the initial cost any further. This can be seen in Figure 8-9 where the cost is shown of a 100 cycle run. The algorithm is not able to reach the minimal cost that was reached in cycle 0. The most seen cost plot is that of Figure 8-10a where there is one peak with a very huge cost. If there is only looked at the first 20 cycles, a plot like Figure 8-10b would be found.

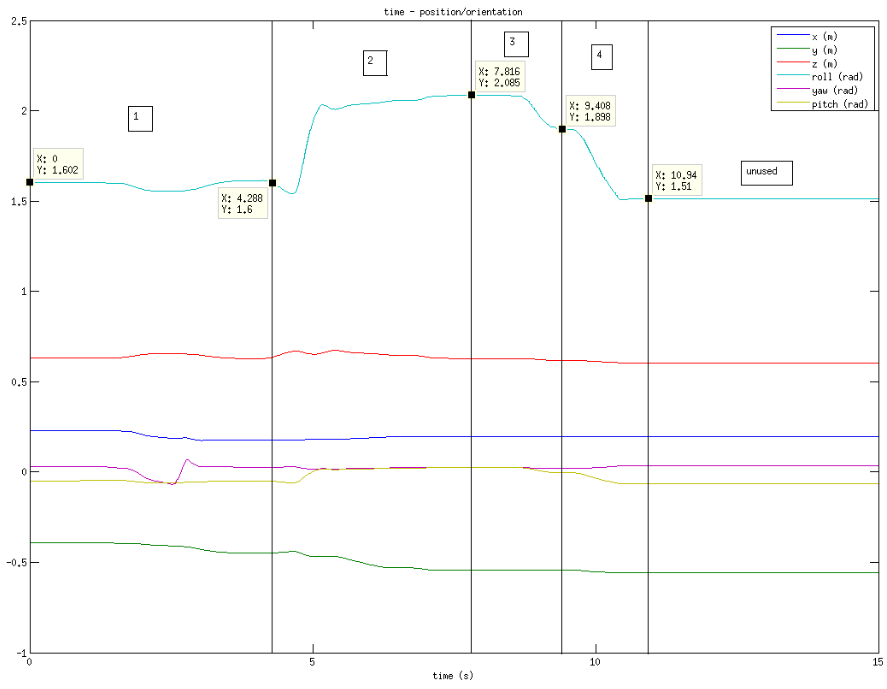


Figure 8-7: Demonstrated placement of cover with sub-trajectories shown

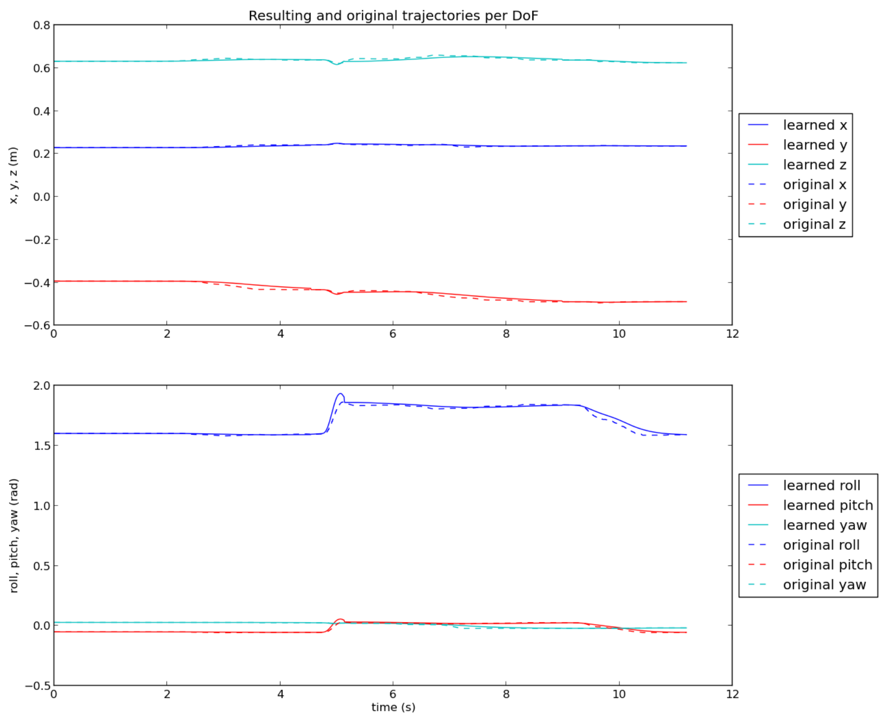


Figure 8-8: Demonstrated trajectories and resulting trajectories of initial DMPs

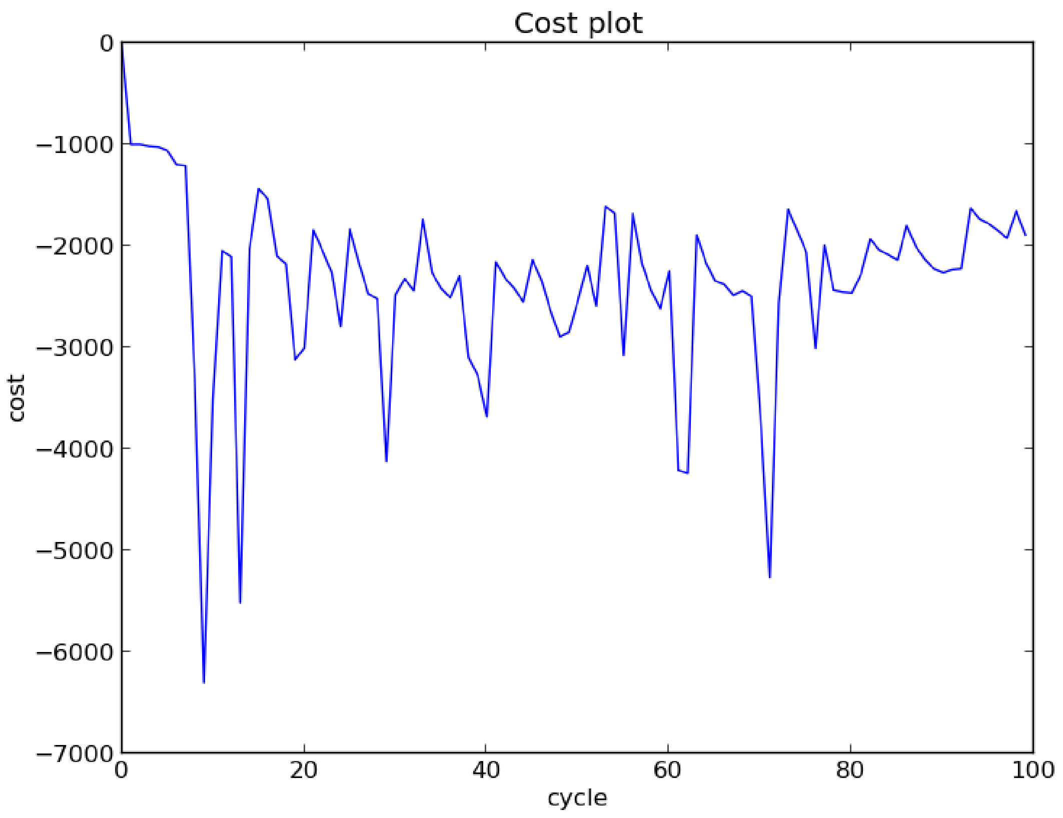
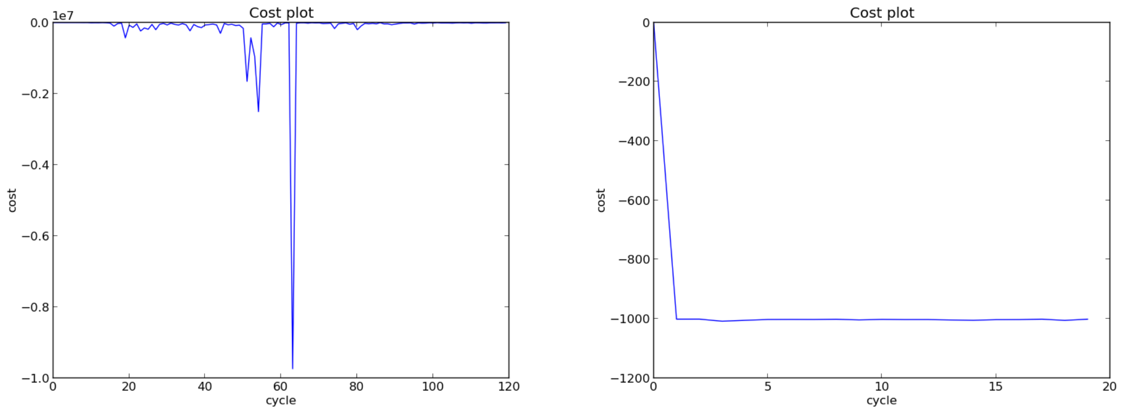


Figure 8-9: Cost plot of the placement of a cover



(a) Complete run

(b) up to iteration 20

Figure 8-10: Cost plot of learning step

8.2.3 Results of noisy demonstration

In order to test the learning algorithm, a demonstration is given that does not result in a placement by the initial DMPs. This is done by adding white Gaussian noise to the trajectories that were used in Section 8.2.2. These trajectories result in initial DMPs that do not place the part. Furthermore the noisy trajectories can be seen as a simulation of an input method that is noisier than teach-in e.g. based on vision. In practice vision-based systems will be used to capture the demonstrations because teach-in is not available or feasible on all robots. The trajectories that are given as input are shown in Figure 8-11, the same sub-trajectories as in Section 8.2.2 are used. The initial DMPs result in the trajectory shown in Figure 8-12.

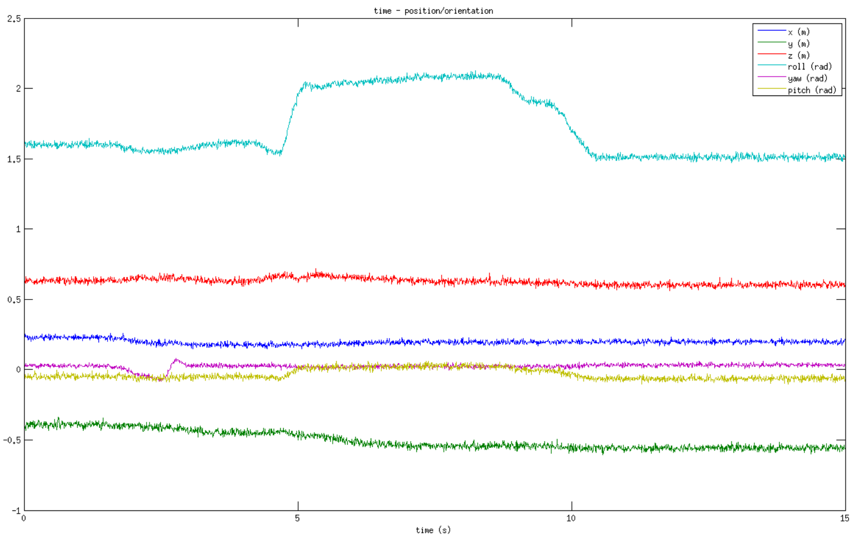


Figure 8-11: Demonstrated placement of cover with added noise

As these trajectories do not result in a successful placement, learning is needed to improve the trajectories to achieve a successful placement. From Figure 8-12 it can be seen why no placement is achieved. This is caused by both the `x` and `z` trajectory. When looking at the final position it can be seen that, due to the noise, there is a spike in the end of the trajectory. Thus the trajectory moves away from the desired value. Furthermore some spikes can be seen halfway the movement. These are caused by the change from one DMP to another. In practice the software does not execute the initial trajectory because the final points are outside the margins of the desired end points, as defined in Section 7.3.3. The goals that are given to the DMPs are the desired final positions. After around 70 iterations the learning algorithm finds a solution within the margins and thus can be executed. However due to unknown reasons the `computeCartesianPath()` function of *MoveIt!* is not able to convert 100% of the trajectory that is given as input. This results in a execution that stops before the complete trajectory is finished. Therefore it can not be determined if the placement is successful or not. In Figure 8-13 such a probably successful placement can be seen. Again it can be noticed that the change from one DMP to another is not fluent. The trajectories that can be converted for more than 90%, which is enough for complete execution, do not result in a successful placement.

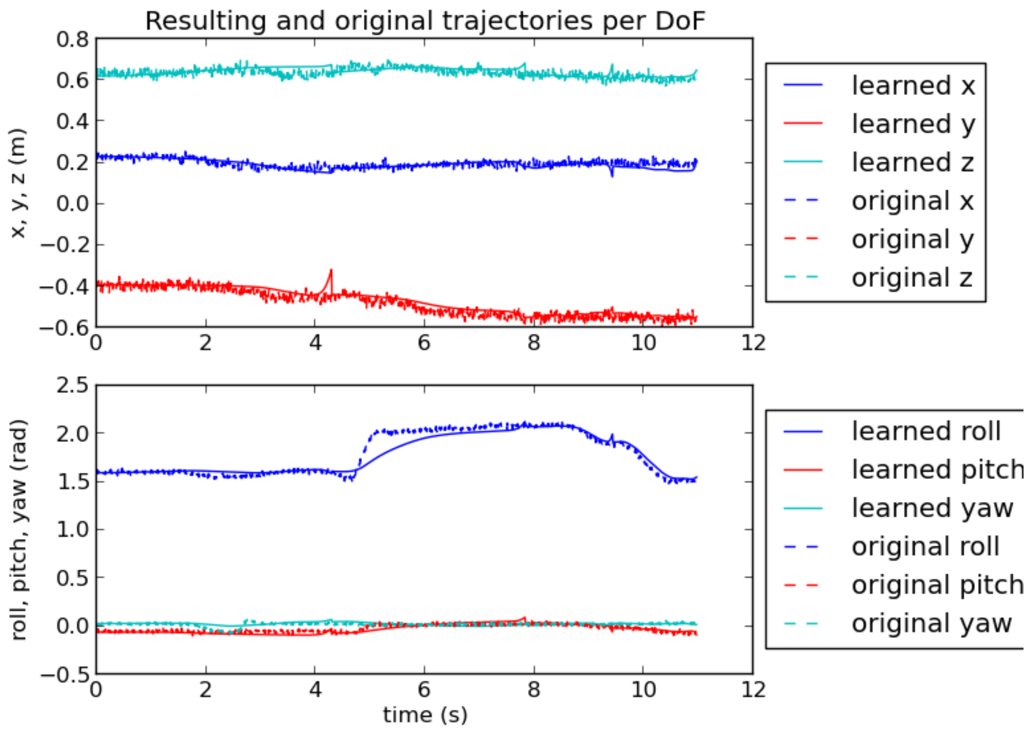


Figure 8-12: Demonstrated trajectories and resulting trajectories of initial DMPs of placement of cover with added noise

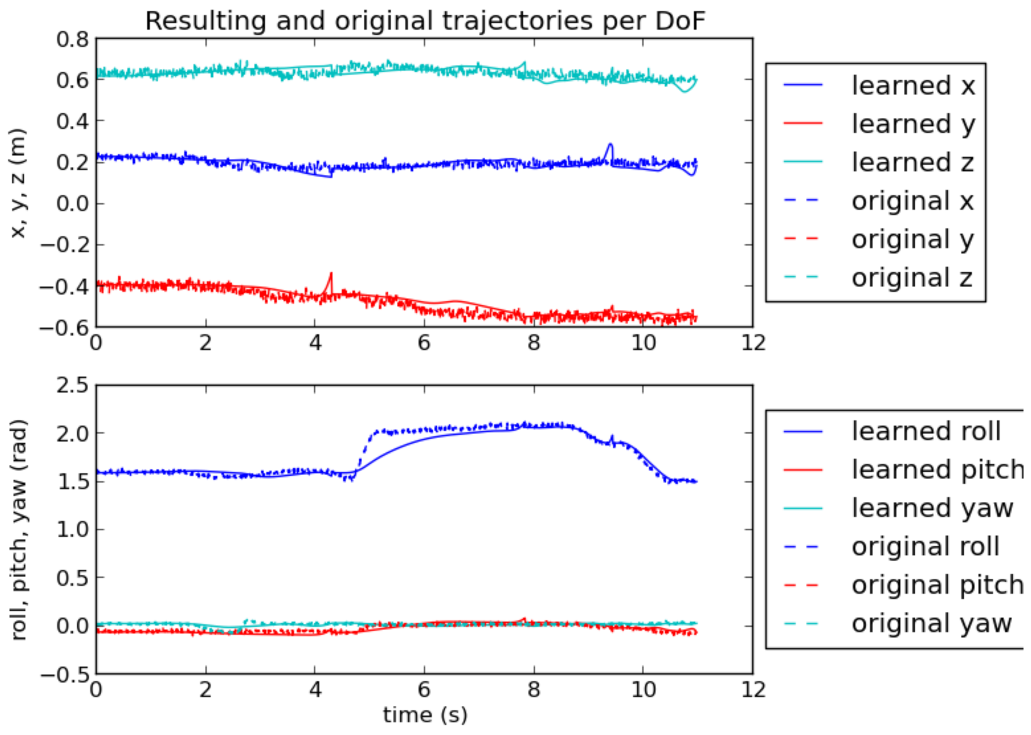


Figure 8-13: Trajectories of a probably successful placement

When looking into the plots of the learned trajectories, it can be seen why the part is not successfully placed. In Figure 8-14 the plot of iteration 76 is shown where the final position is within the margins and thus is executed. Especially the last section of the trajectories is important because that determines the 'click' movement. It can be seen that the spike in the end of the `x` trajectory is not flattened out but has grown. Furthermore a new spike is found in the `roll` trajectory. This means that the algorithm is increasing the weights of the basis functions that create the spike, presumably as part of a random action. These two facts combined make that the part can not be placed successfully. For a successful placement the trajectories should follow a flattened version of the noisy trajectories from second 8 to the end, especially the `roll` trajectory is very important. It might be possible that a suitable solution is found after more than 150 iterations however no more than 150 iterations could be done due to time constraints. From iteration 70 the algorithm finds solutions within the margins and these have to be executed which takes a couple of minutes per execution so the learning process takes a lot of time. Another observation that is done, is that the algorithm can't find an executable trajectory at all. In Figure 8-15 a cost plot can be found of a run of 2000 cycles which did not find a trajectory within the margins.

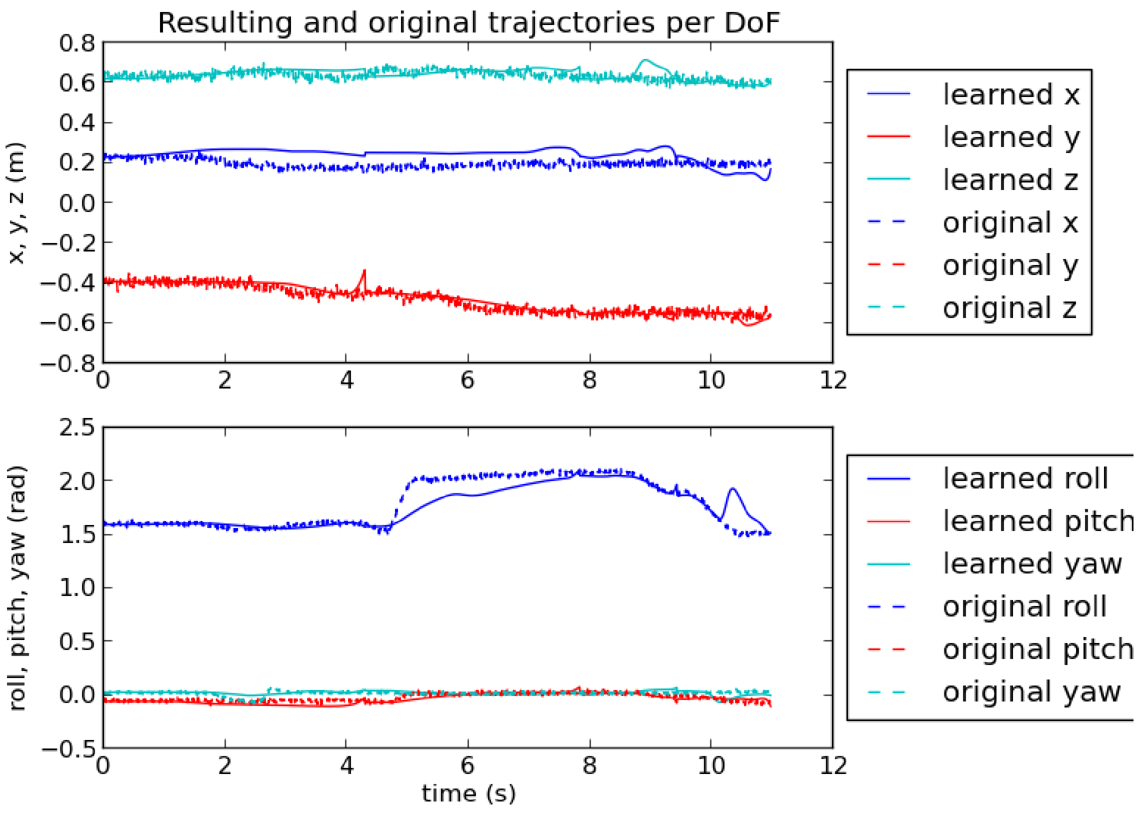


Figure 8-14: Demonstrated trajectories and resulting trajectories of iteration 76

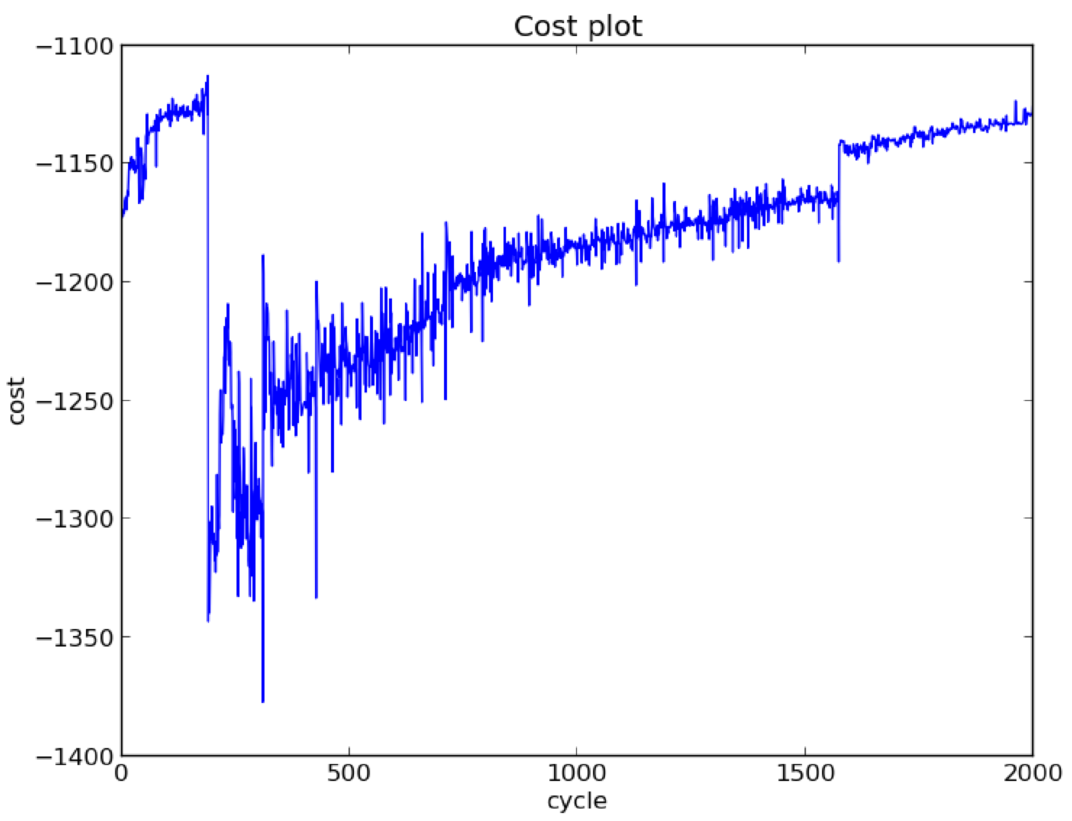


Figure 8-15: Cost plot of the noisy placement of a cover

8.2.4 Results of partial demonstration

Another experiment to test the learning step is done by simulating a partial demonstration. A partial demonstration requires the learning algorithm to find the missing part of the demonstration. The partial demonstration is simulated by taking the demonstration of Section 8.2.2 and setting the boundary of the last sub-trajectory to a value before the demonstration is actually finished. The goals of the DMPs are then set to the desired final position. Furthermore the final place cost is compared to the desired final position. The resulting initial DMPs and the original demonstrated trajectory are shown in Figure 8-16. Especially the `roll` trajectory suffers from this partial demonstration, it goes steep to its final value instead of following the demonstrated curve.

The task for the learning step is to find a solution that places the part successfully. Therefore it needs to learn to follow the curve. From iteration 45 the algorithm starts to find solutions that are within the margin for execution. In Figure 8-17 the plots of iterations 47 and 75 are shown. It becomes clear that the learning algorithm is not able to find a solution that can successfully place the part. This is caused by the fact that the learning algorithm does not create the original curve. Changing the margins of the cost function of Section 7.2.2 does not change the outcome. Furthermore it can be seen that the resulting learned trajectories become less smooth.

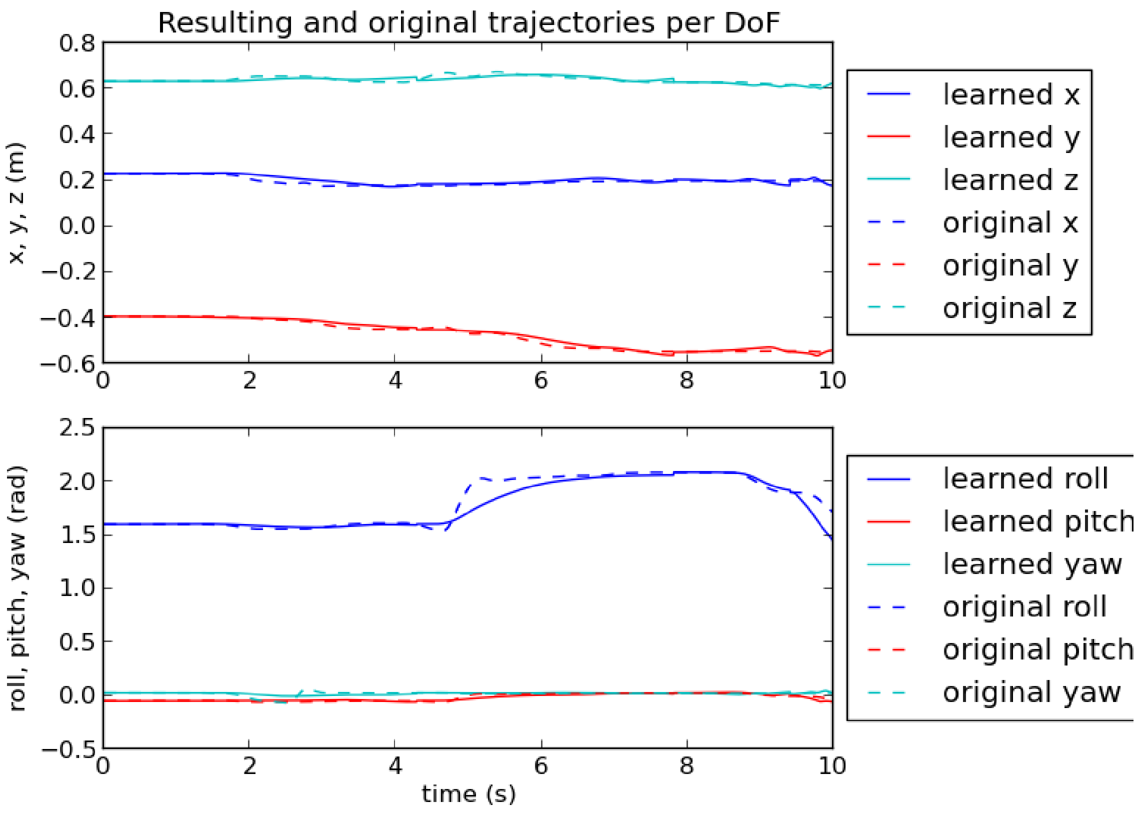
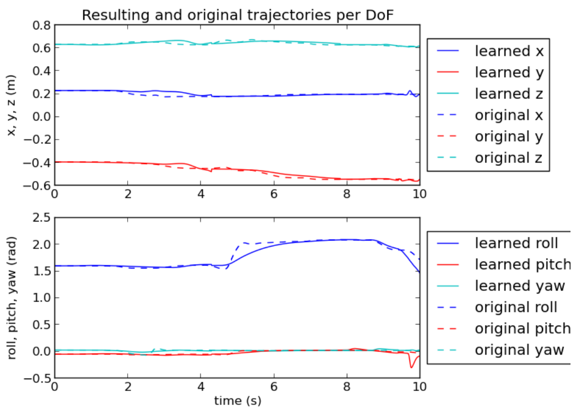
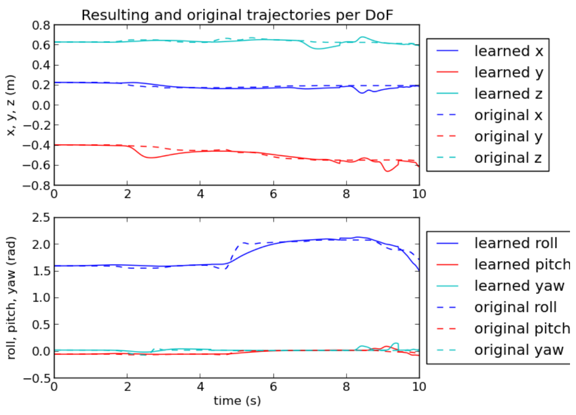


Figure 8-16: Demonstrated trajectories and initial trajectories of partial demonstration



(a) Iteration 47



(b) Iteration 75

Figure 8-17: Learning of partial demonstration

8.2.5 Results of shifted goal

In order to test the working of DMPs itself, a different goal than the demonstrated goal was set. According to the description of the DMPs this should also result in a successful placement. Therefore the carousel was shifted 2 cm in the `x`-direction and the goals of the DMP of the final trajectory were adjusted. This did not result in a successful placement. This is caused by the fact that the 2 cm shifted is needed earlier in the movement. From tests it appeared that the goal needs to be shifted for all sub-trajectories except for the first one to result in a successful placement. In the case of a failing initial placement, the learning step was again not able to learn a successful placement.

8.3 Placement of connection cover part

For the placement of the connection cover part the same procedure is used as in Section 8.2.1. However the placement needs a different movement. For this part the bottom of the part has to be placed in the gap at the bottom of the jig. Then a turning movement has to be made to 'click' the part on the jig at the top. Two clamps hold the part there while the pin of the jig sticks through the hole of the part.

The demonstration resulted in the plot that can be found in Figure 8-18. The sub-trajectories are shown by the vertical lines. The reasoning behind this split is as follows: 1) is chosen because of the big changes in `roll` it is no problem if the movement is smoothed a bit by the initial weights, 2) is chosen for the big change in `pitch` and 3) is chosen because of the change in both `roll` and `yaw`. In Figure 8-19 the resulting initial trajectories are shown. The trajectories fit less with the demonstrated trajectories than that was the case for the cover part.

After demonstration it appeared that the trajectories could not be executed by *MoveIt!*. The `computeCartesianPath()` function could only convert up to 30% of the received trajectory to an executable trajectory. Because this represents only the beginning of the movement, it could not be determined if the trajectories achieve a successful placement. It was tried to get the `computeCartesianPath()` function to convert a higher percentage of the received trajectories by supplying a less dense trajectory however without any effect. Because this is an built-in function of *MoveIt!* it could not be told what the cause of the error is.

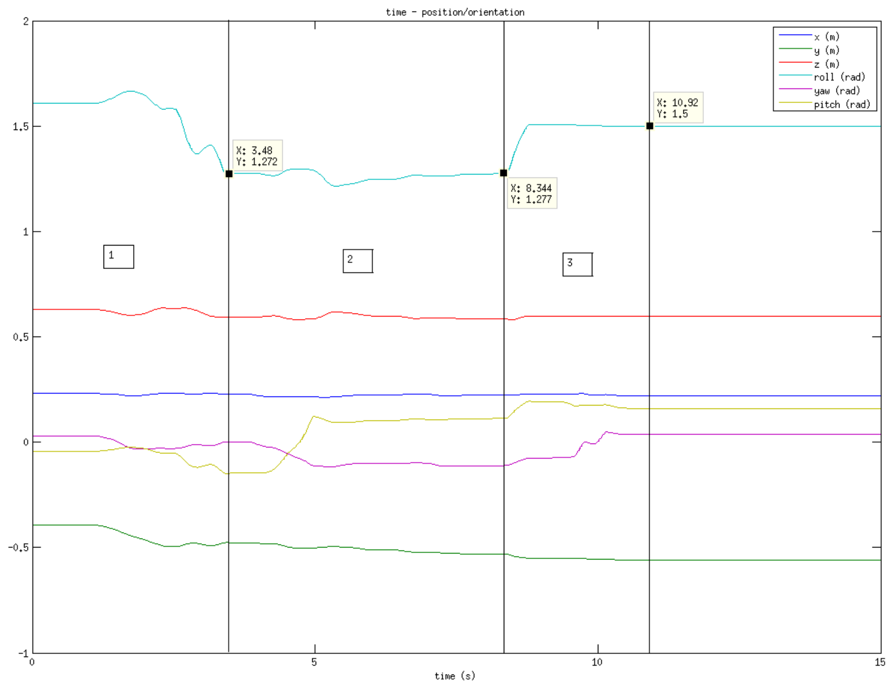


Figure 8-18: Demonstrated placement of the connection cover and the chosen sub-trajectories

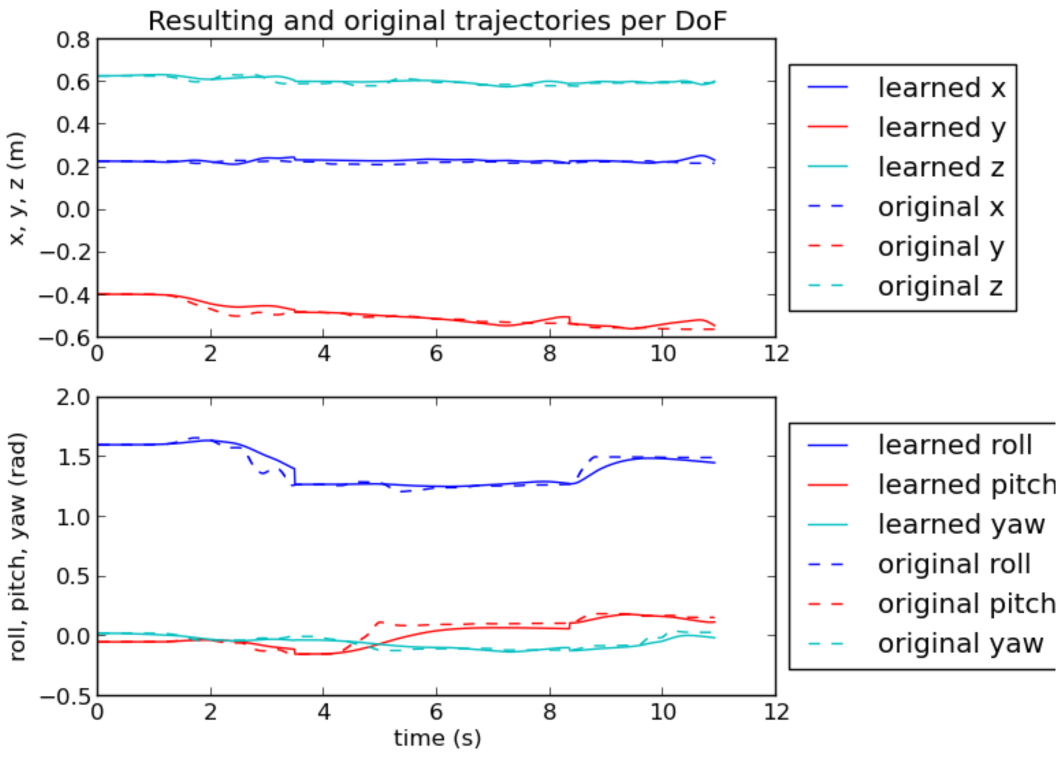


Figure 8-19: Demonstrated trajectories and resulting trajectories of initial DMPs of the placement of the connection cover

Conclusion

This chapter described the experiments that were done for the use-case. From these experiments several conclusions can be drawn. From the first experiment of the placement of the cover part it can be concluded that the initial DMPs performed better than expected. It was expected that a learning step was needed however the initial DMPs were able to achieve a successful placement. The other experiments showed three main things: 1) the learning step did not perform as good as expected, 2) *MoveIt!* needs improvement and 3) data-efficient Reinforcement Learning (RL) is almost required for RL in robotics. The third observation partly follows from the first observation.

There are several reasons why the learning step did not perform as good as expected. The third and most important reason is that this algorithm needs already over 500 iterations for a 2 Degrees of Freedom (DoFs) case. Because for the use-case execution is needed to determine the cost, a lot of executions are needed. It is infeasible to do more than 50 executions in a reasonable time. Therefore the number of iterations that could be reached was most of the time below 200. That 200 iterations could be reached was because of the smart implementation of the margin. This proves the third observation: for a successful application of RL to robotic tasks a data-efficient learning algorithm is needed. By using a data-efficient algorithm the number of executions can be limited. The second reason is that it is very hard to design a good cost function which is the function that determines the performance of the algorithm in the specific case. The cost function is build purely based on human intuition and insight however the effect of the algorithm might not always be as expected. The third reason is that due to the binary final cost, the cost function can be very fluctuating. This also has effect on the gradient. In the case of a smooth cost function and smooth gradient, minimization is done by following the function. However, due to the binary final cost, the algorithm depends more on its random actions for finding the optimal solution. Without random actions the algorithm is not able to find a successful placement and thus exploration is very important in the case of a binary cost.

The second observation is based on the performance of the `computeCartesianPath()` function of *MoveIt!*. The function was not able to convert all provided trajectories to a trajectory for execution on the robot. Why this was the case is not clear because the *MoveIt!* path planner is kind of a black-box application. Actually it is not expected that this happens because the trajectories are recorded using teach-in and are thus feasible.

Chapter 9

Conclusions & future work

In this chapter the conclusions that can be drawn from the research in this thesis are given. Furthermore some directions for future work are given.

9.1 Conclusions

From the research some general conclusions can be drawn and also some conclusions specifically related to the use-case can be drawn. First the general conclusions will be given and then the use-case specific conclusions. Finally the research question will be answered.

The Dynamic Movement Primitives that were used in this thesis performed as expected. From the experiment in Section 8.2.5 it became clear that shifting the goal indeed results in a scaled version of the original movement. Furthermore it is confirmed that a sequence of DMPs is more suitable to describe a movement that consist of multiple sub-movements. Where we, as humans, see a lot of movements as one movement; from an engineering point of view these movements can be brought down to a sequence of movements. Encoding each sub-movement in a separate DMP gave better results than expected. It was expected that the initial DMPs, obtained through regression, would not be able to do a successful placement. However during the experiments it became clear that the sequence of initial DMPs was able to do a successful placement of the part on the jig. Therefore the conclusion can be drawn that a sequence of DMPs is perfectly able to encode a challenging trajectory if it is split in sub-trajectories.

During the testing of the algorithm it became clear that it was not very stable; it crashed very randomly. When looking into the cause of the crash and reasoning about the algorithm it appeared that it had to do with the perturbations that were drawn. Therefore it was decided to limit the size of the perturbation to a deviation of three times the standard deviation from the current mean. This resulted in a more stable algorithm in the sense that infinite costs became very rare. The algorithm was also improved to handle binary final costs in combination with small initial costs. In the original form the algorithm crashed within 20 iterations when this combination was given as an input. In the improved form, which posed limitations on the update equations, the algorithm could run stable.

From the application of the PGPE algorithm to the DMPs it can be concluded that reinforcement learning can be combined with imitation learning. Where imitation learning is represented by DMPs and reinforcement learning by the PGPE algorithm. In itself this is nothing new as the derivation of the parametrized policy showed each DMP can be seen as a parametrized policy and used for reinforcement learning. The PGPE algorithm however had not been combined with DMPs before. The combination was successful in the sense that they could be combined without any adjustments to the algorithm or DMPs itself. The downside of the PGPE algorithm became very clear when the experiments were done: in the number of iterations that is feasible to do on a robot the algorithm was not able to improve upon the given trajectories. This fact leads to the conclusion that data-efficient reinforcement learning algorithms are required for the application of reinforcement learning to robotics in a practical situation. The number of trials for achieving a minimal cost should be kept to a minimum.

Regarding the Robot Operating System (ROS) framework it can be said that the framework is very suited for the easy development of robot applications. In the six months that were used to develop the software, the framework was learned from scratch and the resulting application was built. Due to the core components that are provided by the framework e.g. simulator, communication layer and path planner, the focus can be purely on the software that has to be added. Building further on the DMP package also shows the strength of the framework: by using software that is shared by others, relatively little effort has to be put in creating parts of the application. Another big advantage is that applications can first be tested in simulation and subsequently on the real robot. This helps discovering bugs and errors before the real robot is needed which is very helpful when the access time to the robot is limited.

The first conclusion that can be drawn related to the use-case is that the initial Dynamic Movement Primitives are very well suited for imitating a demonstrated trajectory. As already said above it was not expected that the initial DMPs were able to place the part successfully. By choosing smart sub-trajectories the initial DMPs were however able to place the part successfully. Thereby successfully completing the task that was described by the use-case.

Another aspect that was shown by the experiments for the use-case was that the learning step did not perform as expected. The learning step was not able to improve on the demonstrated trajectory. The PGPE algorithm was not able to optimize a complete demonstration or improve a partial or noisy demonstration. There are several reasons why the learning step did not perform as expected. The first reason is that is very hard to design a suitable cost function. In summary the cost function represents what the designer, mostly based on intuition, thinks is important. Especially the weights that are given to sub-costs are quite arbitrary. For the use-case it is important to follow the demonstrated trajectory but also deviations are allowed especially in the first part of the trajectory. These constraints are very intuitive to a human but hard to express in a mathematical formula. The second reason is, as mentioned above, the PGPE algorithm is not data-efficient and therefore requires a lot of iterations which was not feasible to do on the real setup. The maximal feasible number of iterations that could be done on the setup was around 200. The 2-DoF case showed that for that case already up to 500 iterations were needed. It can be imagined that the real setup, which is 6-DoF, needs more iterations to find a minimal cost because it is a harder to solve problem due to the number of extra weights. The third reason is the binary final cost which makes the cost function very fluctuating. Because the algorithm is based on gradient ascent, and a fluctuating cost results in a rough gradient landscape, it depends more on its random actions to find a minimal cost. The random actions are needed to find a successful placement.

The last conclusion is that the `computeCartesianPath()` function of *MoveIt!* is, at this moment, not suited to convert all provided trajectories. Because this is a complicated function, the user can not easily see what is going on. Therefore it is more like a black-box function that does or doesn't work.

To finalize this conclusion section the research question that was posed in the introduction is answered. The research question was:

"Can imitation learning be combined with reinforcement learning to achieve a successful application in an industrial robotic precision placement task?"

To answer this question, the above conclusions and derivations can be summarized. The first insight is that the robotic precision placement task could be achieved by only using imitation learning. Thus purely based on the results of this thesis the answer to this research question should be 'no'. This answer would then be based on the fact that the reinforcement learning step is not giving good results. However it is clear that reinforcement learning and imitation learning can be combined. Adjustments to the current algorithm or a change of algorithm might result in a working combination of imitation learning and reinforcement learning for a robotic precision placement task. Which has as advantage for the current application, which full fills the task, that it is able to improve on the demonstrated and initial trajectory. Where in this thesis the focus was on following the trajectory, minimizing the jerk, smoothing a noisy trajectory or completing a partially demonstrated trajectory other objectives are also possible. Other minimization objectives might be: energy, execution time, etcetera.

9.2 Future work

Based on the conclusions above and the experience gathered during the research in this thesis some directions for future work can be given. Most directions are tailored to improve the application designed in this thesis. The directions or the results thereof however might also be applicable to other cases.

The first direction for future research is the combination of multiple demonstrations in one DMP. In GMM already multiple demonstrations are used to create the model. It will be an interesting research direction to see if a hybrid model can be created that combines the strengths of both DMP and GMM. A starting point for this direction might be the research of [82] which combines multiple demonstrations in a DMP like system. The advantage of combining multiple demonstrations is that a more general movement is learned.

The second improvement is the splitting of the demonstrated trajectory into multiple sub-trajectories. In this thesis this is done by hand based on the intuition of the researcher. However the ideal case is that no expert is needed and therefore the division in sub-trajectories should become automatic. The application is even not industry ready when this is not implemented and is therefore a necessary extension.

For the recording of the trajectories, the MATLAB driver has to be used which runs on Windows while the ROS framework runs on Linux. Therefore the ROS UR5 driver should be improved such that the teach-in mode of the UR5 can be used in combination with ROS such that the trajectories can be directly recorded using the ROS framework. This extensions is also needed if the application runs in an industrial environment; the application should become one software package that incorporates all functionality. Furthermore there should be looked into the input method that is used, currently it is assumed that this is teach-in. However, for industrial robots a teach-in mode is not always available. Vision seems to be the most promising and intuitive input method for giving the demonstrations. It would be a nice challenge to incorporate this as input method in the current application.

Until now the application has only be tested on the UR5 robot. However the application should be suited for each robot for which a ROS driver exists. A relatively simple but interesting case would be to test the application on different robots.

As concluded a data-efficient reinforcement learning algorithm is actually needed for reinforcement learning in robotics. While the work of this thesis was already in progress a data-efficient version of the PGPE algorithm was proposed in a paper [83],[84]. It would be interesting to see what the results of the data-efficient version of the algorithm are.

The last direction of future work is to overcome the problem of creating a suitable cost function. This might be done by doing research in the direction of inverse reinforcement learning or preference-based reinforcement learning which were mentioned in Chapter 2. A good starting point might be a recent paper about direct policy search using preference-based reinforcement learning [85]. Another starting point can be a new framework called programming by feedback [86].

Appendix A

Robot Operating System

The Robot Operating System (ROS) is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behaviour across a wide variety of robotic platforms. ROS was built to encourage collaborative robotics software development [87]. In this appendix some basis principles of ROS will be explained.

A.1 Filesystem

The functional parts of ROS are formed by nodes. A node is a standalone executable file, but it also refers to the source code behind it. Functionalities are grouped in packages. A package can contain several nodes. A package is the most atomic build item and release item in ROS. Besides source code a package can also contain messages and services. Which contain the definition of a data structure used for a message or the definition of a request and response data structure respectively [88].

A.2 Communication

The communication in ROS is done over topics. A node sends out a message by publishing it to a given topic. The topic is a name that is used to identify the content of the message. A node that wants information about certain kind of data, subscribes to the appropriate topic. In general publishers and subscribers are not aware of each others' existence. The idea is to decouple the production of information from its consumption.

In order to support a request/reply manner of interaction the *service* was introduced. A service is a defined pair of message structures: one for the request and one for the reply. A providing node offers a service under a name and another node uses the service by sending the appropriate request message. From a programming point of view this can be seen as a remote procedure call [88].

A.3 Plugins

There are some plugins or software packages that are available upon install of ROS. The three most important ones are Gazebo, RVIZ and MoveIt!. MoveIt! will be explained in Appendix B.

A.3.1 Gazebo

Gazebo is a robot simulation tool [89]. It offers the ability to efficiently simulate robots in complex indoor and outdoor environments. Gazebo enables the user to build its own elements in a simulation like robots, lights, sensors and objects. Gazebo is a real physics simulator which simulates the real world. An impression of a UR5 in Gazebo can be found in Figure A-1.

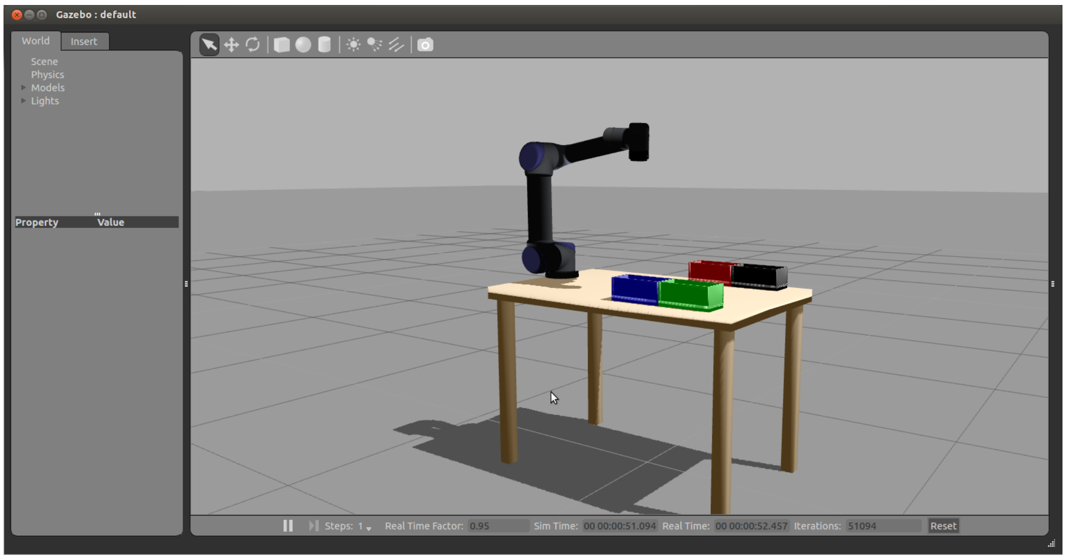
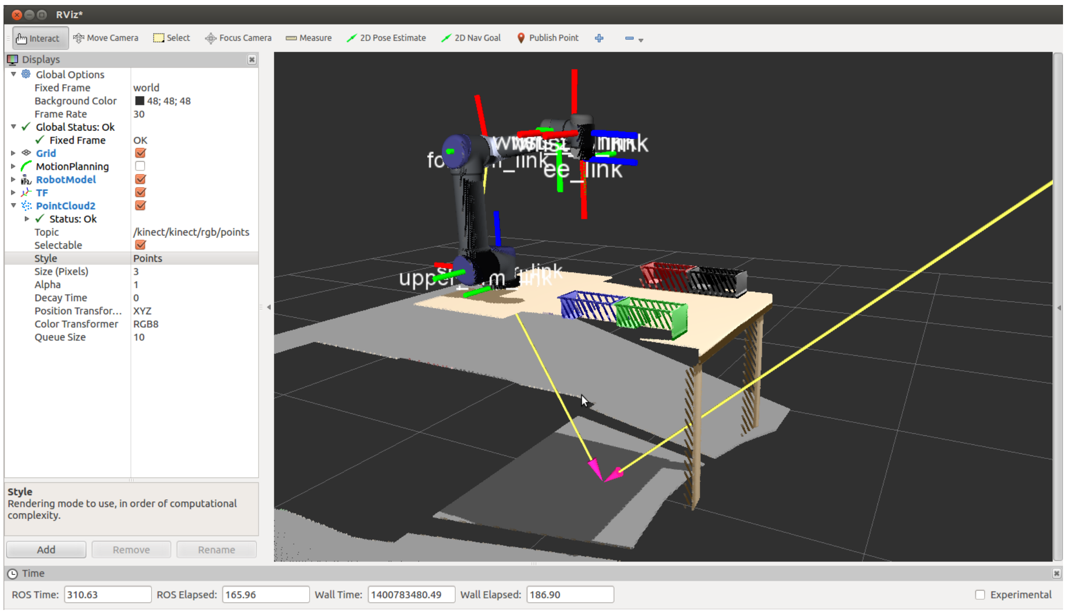


Figure A-1: UR5 in Gazebo

A.3.2 RVIZ

RVIZ stands for Robot Visualiser, it visualizes all information that is available in the ROS system [90]. RVIZ can for example be used to visualize the data of a 3D camera but also all the known reference frames or the model of the robot. A picture of RVIZ displaying a point cloud received from a 3D camera can be found in Figure A-2



Global Options	
Fixed Frame	world
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
MotionPlanning	☐
RobotModel	✓
TF	✓
PointCloud2	✓
✓ Status: Ok	
Topic	/kinect/kinect/rgb/points
Selectable	✓
Style	Points
Size (Pixels)	3
Alpha	1
Decay Time	0
Position Transfor...	XYZ
Color Transformer	RGB8
Queue Size	10

Style
Rendering mode to use, in order of computational complexity.

ROS Time: 310.63 ROS Elapsed: 165.96 Wall Time: 1400783480.49 Wall Elapsed: 186.90

Figure A-2: UR5 as seen by a 3D camera in RVIZ

Appendix B

MoveIt!

MoveIt! is a state of the art software package for mobile manipulation, incorporating the latest advances in motion planning, manipulation, kinematics, control and navigation [80]. In combination with Robot Operating System (ROS) *MoveIt!* is mostly used to control the robot by planning its movements. *MoveIt!* integrates with the Open Motion Planning Library (OMPL) which primarily implements randomized motion planners. These motion planners are used by MoveIt! as its default set of planners. The planners in OMPL have no concept of a robot. *MoveIt!* configures OMPL and provides the back-end to work with problems in robotics. *MoveIt!* is able to take the environment into account and plan a collision free path for multiple degrees of freedom.

OMPL

This section gives a short introduction into OMPL and its sampling-based motion planning.

Sampling-based motion planning is a powerful concept that employs sampling of the state space of the robot in order to quickly and effectively answer planning queries especially for systems with many degrees of freedom [91]. Sampling arises out of the need to quickly cover a large and complex state space. Sampling-based motion planning reasons over a finite set of configurations in the state space.

The goal of a sampling-based motion planning query is the task of finding a collision free path in the state space of the robot from a distinct start state to a specific goal state. Hereby utilizing a path composed of configurations connected by collision free paths. OMPL implements several different sampling-based motion planning algorithms.

The default motion planner used by *MoveIt!* is the `LBKPIECE1` which stands for Lazy Bi-directional KPIECE with one level of discretization. It is a tree-based planner that uses a discretization to guide the exploration of the continuous space. This implementation uses two trees of exploration with lazy collision checking [92].

Appendix C

Axis-angle to roll, pitch, yaw

This appendix describes the conversions from the axis-angle representation as used by the UR5 to the roll, pitch yaw representation as used by Robot Operating System (ROS). A lot of knowledge for the conversion is derived from [93]. The conversion is done by writing down the rotation matrix for both the axis-angle representation and the roll, pitch, yaw representation. Because these rotation matrices should be the same, one can write terms of one matrix in terms of the other matrix which is done for the conversion.

C.1 Axis-angle rotation matrix

The complete derivation of the rotation matrix of the axis-angle representation can be found at [93]. Here only the result is given. It should be noted that the 3x3 matrix that is used for representing the axis-angle is converted from vector algebra to matrix algebra using the skew symmetric or 'tilde' matrix. This results in:

$$[\mathrm{R}] = [\mathrm{I}] + \sin(angle)[\sim axis] + (1 - \cos(angle))[\sim axis]^2 \tag{C-1}$$

If we write out the rotation matrix R we get the following:

$$[R] = \begin{bmatrix} tx^2 + c & txy - zs & txz + ys \\ txy + zs & ty^2 + c & tyz - xs \\ txz - ys & tyz + xs & tz^2 + c \end{bmatrix} \tag{C-2}$$

Where $c = \cos(angle)$, $s = \sin(angle)$, $t = 1 - c$, x is the normalized axis x coordinate, y is the normalized axis y coordinate and z is the normalized axis z coordinate.

C.2 RPY rotation matrix

The rotation matrix for roll, pitch, yaw is derived from [94]. For each of the rotation a separate rotation matrix can be written down.

Roll:

$$R_x(\gamma) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\gamma) & -\sin(\gamma) \\ 0 & \sin(\gamma) & \cos(\gamma) \end{bmatrix} \tag{C-3}$$

Pitch:

$$R_y(\beta) = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{bmatrix} \tag{C-4}$$

Yaw:

$$R_z(\alpha) = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{C-5}$$

The roll, pitch, yaw rotation matrices can be multiplied to obtain a single rotation matrix. It should be noted that $R(\alpha, \beta, \gamma)$ first performs the roll, then the pitch and then the yaw. If the order is changed, a different rotation matrix would result.

$$R(\alpha, \beta, \gamma) = R_z(\alpha)R_y(\beta)R_x(\gamma) = \begin{bmatrix} c\alpha c\beta & c\alpha s\beta s\gamma - s\alpha c\gamma & c\alpha s\beta c\gamma + s\alpha s\gamma \\ s\alpha c\beta & s\alpha s\beta s\gamma + c\alpha c\gamma & s\alpha s\beta c\gamma - c\alpha s\gamma \\ -s\beta & c\beta s\gamma & c\beta c\gamma \end{bmatrix} \tag{C-6}$$

Where c is cos and s is sin.

C.3 Converting

To get the value for β we take the term $R_{3,1}$. This results in:

$$\beta = \text{asin}(-txz + ys) \tag{C-7}$$

To get the value for γ we divide $R_{3,2}$ by $R_{3,3}$ which results in:

$$\gamma = \text{atan2}(tyz + xs, tz^2 + c) \tag{C-8}$$

To get the value for α we divide $R_{2,1}$ by $R_{1,1}$ which results in:

$$\alpha = \text{atan2}(txy + zs, tx^2 + c) \tag{C-9}$$

There are two singularities at $\beta = \pi/2$ and at $\beta = -\pi/2$. When $\beta = \pi/2$ we can rewrite the rotation matrix using some trigonometric identities. We then can write:

$$\frac{R_{1,2}}{R_{1,3}} = \frac{\sin(\gamma - \alpha)}{\cos(\gamma - \alpha)} \tag{C-10}$$

Then choose $\alpha = 0$. This results in

$$\gamma = \text{atan2}(txy - zs, txz + ys) \tag{C-11}$$

The same can be done for $\beta = -\pi/2$. Then we also take $alpha = 0$ and we get:

$$\gamma = \text{atan2}(txy - zs, txz + ys) \tag{C-12}$$

Appendix D

DSPE Conference 2014 proceedings

On the next pages the extended abstract that is published in the proceedings of the DSPE Conference 2014 on precision mechatronics is shown. The theme of the conference is 'Revolution vs. Evolution' [95].

Flexible industrial robotics through imitation learning

Mark Geelen, Simon Jansen, Jacco van der Spek
Alten Mechatronics, Eindhoven, The Netherlands
Jacco.van.der.Spek@alten.nl

INTRODUCTION

In industrial environments robots are used for various tasks. However, it is not feasible for companies to deploy robots for productions with a limited batch size or for products with large variations. The current generation of robots is highly inflexible; therefore many repetitive tasks are still performed by humans. Current robots essentially have an insufficient level of intelligence: the robots are programmed at controller level and not at task level. Typically, a fixed sequence of actions is programmed, and robots do not learn from their mistakes or optimize their behaviour over time. Furthermore, a robot expert is still needed to program the robot. A solution to these challenges would be a new generation of robots and software which can adapt quickly to new situations and which learns from their mistakes while being programmable without needing an expert.

The concept that we propose as enabler to more flexible robotics is the combination of imitation learning and reinforcement learning. Imitation learning is a method "by which a robot learns new skills through human guidance and imitation" [1]. The purpose of imitation learning is to learn a task by generalizing from observations. The power of imitation learning is that the robot is programmed in an intuitive way while the insight of the teacher is incorporated in the execution of the task.

In this work, a combination of imitation and reinforcement learning is applied to a robotic precision placement task in an industrial setting. The robot generalizes movements that are demonstrated through an advanced form of teach-in by human operators. Reinforcement learning is then used to optimize the demonstrated trajectory in order to make sure that a correct placement is achieved. This optimization step is important because the initial learned movement cannot be assumed to be perfect.

This research contributes to the existing solutions in two ways. The first contribution of this work is the algorithm that is used for the optimization which has not been combined with imitation learning before.

The second contribution is the combination of imitation and reinforcement learning being applied in an industrial use-case.

This extended abstract is structured as follows: first, the concept of Dynamic Movement Primitives (DMPs) [2] are explained, this forms the basis of imitation learning. Next it is shown that a DMP can be seen as a parameterized policy. After that, the used algorithm is shortly explained. Furthermore the results of experiments are discussed. Finally, conclusions are presented.

DYNAMIC MOVEMENT PRIMITIVES

A movement primitive is derived from the complex movement skills that are found in nature. Dynamic Movement Primitives (DMPs) can be seen as a spring-damper like system with a non-linear forcing term [2]. This is also called the transformation system and is given by:

$$\tau\dot{v} = K(g-x) - Dv - K(g-x_0)s + Kf(s)$$
$$\tau\dot{x} = v$$

The forcing term *f* is a sum of Gaussian basis functions with each its corresponding weight w_i. The *K* and *D* parameters are the spring and damper constants respectively. *x* and *v* are the position and velocity; x_0 and *g* are the start and goal position; and τ is a temporal scaling factor. The phase variable *s* changes from 1 to 0 during the movement. The term $K(g-x_0)$ is required to avoid jumps at the beginning of the movement.

In order to encode a demonstrated trajectory in a DMP, first the high level parameters *g*, x_0 and τ are determined. Where τ is simply the duration of the demonstrated movement. The weights w_i are then determined by Locally Weighted Regression (LWR) [3] which minimizes the error between the demonstration and the trajectory created by the DMP.

DMP AS A PARAMETERIZED POLICY

In reinforcement learning, a policy π maps states to actions. This policy can be optimized to find the optimal policy π*. When the action and state space become continuous, the problem of finding an optimal policy becomes hard to solve. Therefore a parametric policy π_θ is used to find the optimal policy. This is done by finding the optimal policy parameters θ^* [4]. A DMP can be seen as a special form of a parameterized policy where the weights w_i of the forcing term are taken as parameters θ. Policy search algorithms can be used to find a parameter vector θ^* which minimizes the expected cost. Here, the cost is a function which is determined beforehand and incorporates terms that have to be minimized. In the next section the reinforcement learning algorithm that is used to minimize this cost is explained.

POLICY GRADIENTS WITH PARAMETER BASED EXPLORATION

The Policy Gradients with Parameter based Exploration (PGPE) algorithm is a policy search algorithm [5]. The algorithm is derived from the general framework of episodic reinforcement learning in a Markovian environment. The goal of reinforcement learning is to optimize the agent's expected reward. One way to optimize this is by estimating the gradient of the cost-function $J(\theta)$ which has to be optimized.

PROOF OF PRINCIPLE

In order to prove the principle of combining the methods mentioned above, a 2 Degree of Freedom (DoF) case is taken. The x-y trajectory of a two joint robot is taken as demonstration. This demonstration is used to encode a sequence of 2 DoF DMPs; one for each DoF. By this encoding the aforementioned weights w are obtained. These weights w are used as an initialization of the PGPE algorithm: μ is set to the initial weights. Then

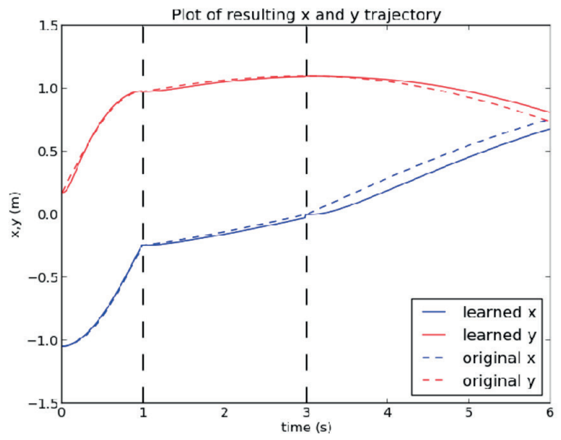


Figure 1
Demonstrated trajectory and initial DMP trajectory

the PGPE algorithm tries to optimize the trajectories that are formed by the DMPs according to a cost function. The cost function is focused on following the demonstrated trajectory.

The demonstrated trajectory and the resulting trajectory of the initial DMPs can be found in Figure 1. It should be noted that the trajectory is split, by hand, in sub-trajectories as can be seen in the figure by the black dotted lines. For each sub-trajectory a DMP is learned. After applying learning to the weights of the DMPs the resulting trajectory can be found in Figure 2 while the corresponding cycle-cost plot can be found in Figure 3. It can be concluded that after 300 cycles the cost starts to converge and that the resulting trajectory after 500 cycles has significantly improved.

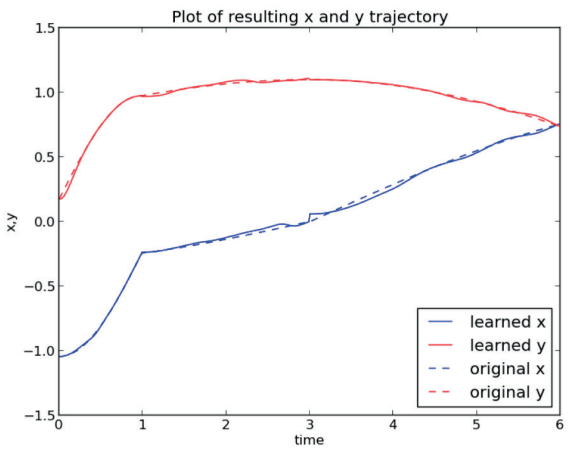


Figure 2
Resulting trajectory

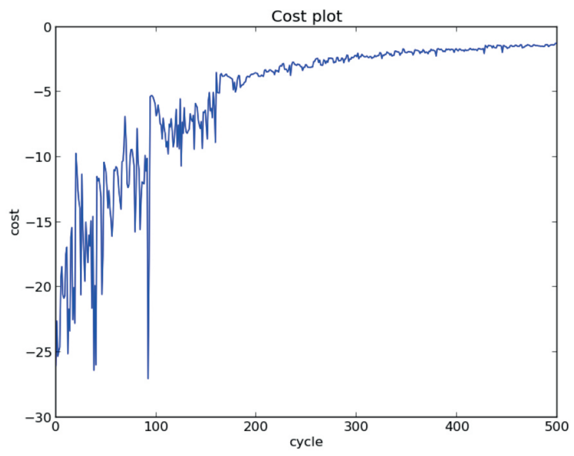


Figure 3
Cycle-cost plot

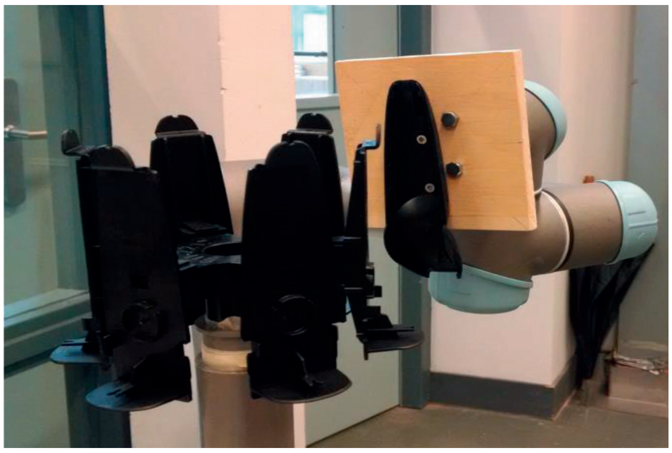


Figure 4
Setup before placement

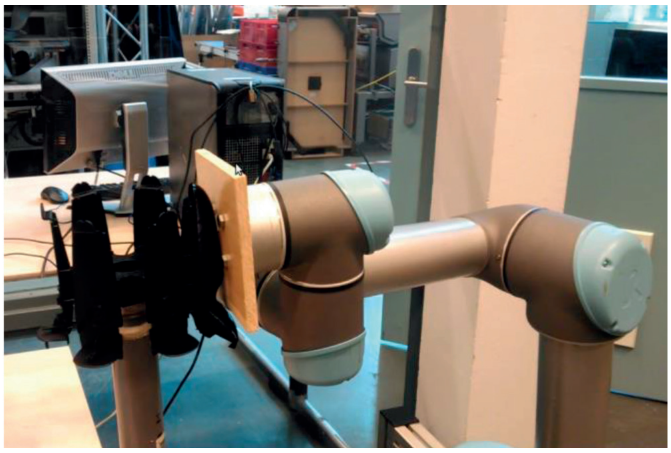


Figure 5
Setup after successful placement

USE-CASE

For the use-case an UR5 robot from Universal Robots [6] was used in teach-in modus to record demonstrated trajectories. These demonstrated trajectories are then used to encode a 6 DoF DMP: the trajectory is described using the **x, y, z** positions and **roll, pitch, yaw** rotations of the tooltip. These variables were chosen because by operating in Cartesian space, the operation becomes robot independent. The performed task is a precision placement of a part on a jig which is attached to a carrousel. In Figure 4, the setup, and the part attached to the tooltip and carrousel can be seen. The placing of the part is derived from a process at Philips Consumer Lifestyle where a part of a Philips consumer product has to be placed on a painting carrousel manually.

All the above mentioned algorithms and techniques were implemented using the Robot Operating System -Industrial (ROS-I) [7]. ROS-I is an open source frame-work for robotic applications which incorporates many existing software packages. Therefore, the built-in path planning package of ROS-I, MoveIt!, was used to control the UR5 robot arm.

First experiments already show promising results. Figure 6, presents the demonstrated trajectories and the produced trajectories. The produced trajectories are generated by the initial sequence of DMPs which are used to encode the demonstrated trajectories. When executing the generated trajectories, this results in the placement of the part on the jig as can be observed in Figure 5.

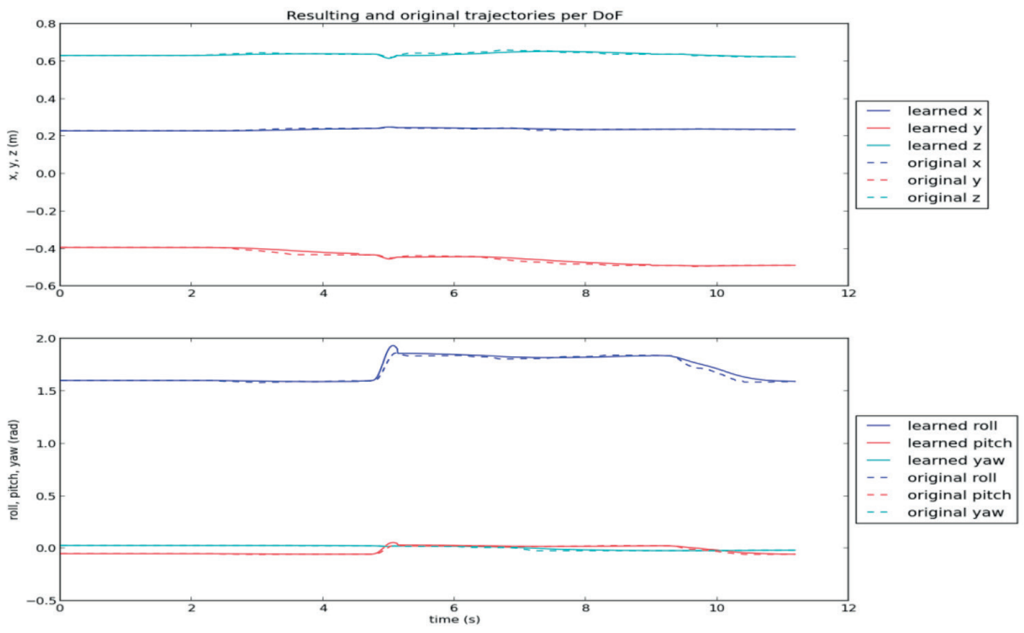


Figure 6
Demonstrated trajectories and initial DMP trajectories

CONCLUSION

The initial tests in which the combination of imitation and reinforcement learning were used are very promising. Even without learning, the robot was able to perform a successful precision placement of the part on the jig.

The proof of principle showed that a combination of imitation learning and reinforcement learning is able to improve a demonstrated trajectory. From this, we can conclude that the chosen optimization algorithm can be successfully combined with imitation learning. Furthermore, the results of this proof of principle enable the future application of this research in an industrial use-case.

Therefore, we can conclude that a combination of imitation and reinforcement learning is a promising direction toward more flexible robotics. In this way, a robot can be programmed by a non-expert operator, without the need for perfect demonstrations.

This work is part of the R5-COP project and was funded by the ECSEL-JU.

REFERENCES

[1] M. A. Armada, A. Sanfeliu and M. Ferre, "Programming by Demonstration: A Taxonomy of Current Relevant Methods to Teach and Describe New Skills to Robots," in ROBOT2013: First Iberian Robotics Conference, 2014, pp. 287-300.
[2] A. J. Ijspeert, J. Nakanishi, H. Heiko, P. Pastor and S. Schaal, "Dynamical movement primitives: learning attractor models for motor behaviors," Neural computation, vol. 25, no. 2, pp. 328--373, 2013.
[3] C. G. Atkeson, "Using locally weighted regression for robot learning," in Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on, pp. 958–963, IEEE, 1991.
[4] F. Stulp, O. Sigaud, et al., "Policy improvement methods: Between black-box optimization and episodic reinforcement learning," 2012.
[5] F. Sehnke, C. Osendorfer, T. Rückstieß, A. Graves, J. Peters, and J. Schmidhuber, "Parameter-exploring policy gradients," Neural Networks, vol. 23, no. 4, pp. 551–559, 2010.
[6] "UR5," Universal Robots, 2014. [Online]. Available: http://www.universal-robots.com/GB/Products.aspx.
[7] "ROS Industrial," SwRI, 2014. [Online]. Available: http://rosindustrial.org/.

Appendix E

Software variables

This appendix gives an overview of the variables that are used for the software packages. Most of the variables are also mentioned in the thesis but this appendix collects them in one place.

E.1 DMP package

Table E-1: Variables used in the DMP package

Variable	Value
`D`	100.0
`K`	20.0
`number_basis_functions`	15
`intersection_height`	0.5
`goal_tresh`	0.05
`integrate_iter`	1
`alpha`	$-\log(0.01)$

E.2 PGPE package

Table E-2: Variables used in the PGPE package

Variable	Value
`t_sigma`	0.02
`g_sigma`	0.01
max `delta_sigma`	20.0
max `random_number`	$\mu \pm 3\sigma$

E.3 Precision placement executive

Table E-3: Margins used for precision placement application

Variable	Margin
`x`	0.013 [m]
`y`	0.013 [m]
`z`	0.013 [m]
`roll`	0.05 [rad]
`pitch`	0.05 [rad]
`yaw`	0.07 [rad]

Bibliography

[1] I. H. Witten, "Programming by example for the casual user: a case study," 1981.

[2] J. Bautista-Ballester, J. Vergés-Llahí, and D. Puig, "Programming by demonstration: A taxonomy of current relevant methods to teach and describe new skills to robots," in *ROBOT2013: First Iberian Robotics Conference*, pp. 287–300, Springer, 2014.

[3] F. Chersi, "Learning through imitation: a biological approach to robotics," *Autonomous Mental Development, IEEE Transactions on*, vol. 4, no. 3, pp. 204–214, 2009.

[4] A. Billard and D. Grollman, "Robot learning by demonstration," *Scholarpedia*, vol. 8, no. 12, p. 3824, 2013.

[5] T. Lozano-Perez, "Robot programming," *Proceedings of the IEEE*, vol. 71, no. 7, pp. 821–841, 1983.

[6] J. Kober and J. Peters, "Reinforcement learning in robotics: a survey," in *Reinforcement Learning*, pp. 579–610, Springer, 2012.

[7] S. Manschitz, J. Kober, M. Gienger, and J. Peters, "Learning to unscrew a light bulb from demonstrations," in *ISR/Robotik 2014; 41st International Symposium on Robotics; Proceedings of*, pp. 1–7, VDE, 2014.

[8] A. Billard, S. Calinon, R. Dillmann, and S. Schaal, "Robot programming by demonstration," in *Springer Handbook of Robotics* (B. Siciliano and O. Khatib, eds.), pp. 1371–1394, Springer Berlin Heidelberg, 2008.

[9] F. Stulp and O. Sigaud, "Robot skill learning: From reinforcement learning to evolution strategies," *Paladyn, Journal of Behavioral Robotics*, vol. 4, no. 1, pp. 49–61, 2013.

[10] A. J. Ijspeert, J. Nakanishi, H. Hoffmann, P. Pastor, and S. Schaal, "Dynamical movement primitives: learning attractor models for motor behaviors," *Neural computation*, vol. 25, no. 2, pp. 328–373, 2013.

[11] F. Sehnke, C. Osendorfer, T. Rückstieß, A. Graves, J. Peters, and J. Schmidhuber, "Parameter-exploring policy gradients," *Neural Networks*, vol. 23, no. 4, pp. 551–559, 2010.

[12] T. Luksch, M. Gienger, M. Mühlig, and T. Yoshiike, "A dynamical systems approach to adaptive sequencing of movement primitives," in *Robotics; Proceedings of ROBOTIK 2012; 7th German Conference on*, pp. 1–6, VDE, 2012.

[13] S. Schaal, J. Peters, J. Nakanishi, and A. Ijspeert, "Control, planning, learning, and imitation with dynamic movement primitives," in *Proceedings of the Workshop on Bilateral Paradigms on Humans and Humanoids, IEEE 2003 International Conference on Intelligent RObots and Systems (IROS)*, pp. 27–31, 2003.

[14] P. Pastor, H. Hoffmann, T. Asfour, and S. Schaal, "Learning and generalization of motor skills by learning from demonstration," in *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pp. 763–768, IEEE, 2009.

[15] O. C. Jenkins and M. J. MATARIĆ, "Performance-derived behavior vocabularies: Data-driven acquisition of skills from motion," *International Journal of Humanoid Robotics*, vol. 1, no. 02, pp. 237–288, 2004.

[16] J. Nakanishi, J. Morimoto, G. Endo, G. Cheng, S. Schaal, and M. Kawato, "Learning from demonstration and adaptation of biped locomotion," *Robotics and Autonomous Systems*, vol. 47, no. 2, pp. 79–91, 2004.

[17] J. Peters, J. Kober, K. Mülling, O. Krämer, and G. Neumann, "Towards robot skill learning: From simple skills to table tennis," in *Machine Learning and Knowledge Discovery in Databases*, pp. 627–631, Springer, 2013.

[18] F. Meier, E. Theodorou, and S. Schaal, "Movement segmentation and recognition for imitation learning," in *International Conference on Artificial Intelligence and Statistics*, pp. 761–769, 2012.

[19] K. Mülling, J. Kober, O. Kroemer, and J. Peters, "Learning to select and generalize striking movements in robot table tennis," *The International Journal of Robotics Research*, vol. 32, no. 3, pp. 263–279, 2013.

[20] A. J. Ijspeert, J. Nakanishi, and S. Schaal, "Learning attractor landscapes for learning motor primitives," in *Advances in neural information processing systems*, pp. 1523–1530, 2002.

[21] J. Kober and J. Peters, "Imitation and reinforcement learning," *Robotics & Automation Magazine, IEEE*, vol. 17, no. 2, pp. 55–62, 2010.

[22] G. Chang and D. Kulic, "Robot task learning from demonstration using petri nets," in *RO-MAN, 2013 IEEE*, pp. 31–36, IEEE, 2013.

[23] G. Chang and D. Kulic, "Robot task error recovery using petri nets learned from demonstration," in *IEEE International Conference on Advanced Robotics*, 2013.

[24] F. Guenter, M. Hersch, S. Calinon, and A. Billard, “Reinforcement learning for imitating constrained reaching movements,” *Advanced Robotics*, vol. 21, no. 13, pp. 1521–1544, 2007.

[25] D. A. Reynolds, “Gaussian mixture models,” in *Encyclopedia of Biometrics*, pp. 659–663, 2009.

[26] S. Calinon and A. Billard, “Statistical learning by imitation of competing constraints in joint space and task space,” *Advanced Robotics*, vol. 23, no. 15, pp. 2059–2076, 2009.

[27] M. Mühlig, M. Gienger, and J. J. Steil, “Interactive imitation learning of object movement skills,” *Autonomous Robots*, vol. 32, no. 2, pp. 97–114, 2012.

[28] M. Hersch, F. Guenter, S. Calinon, and A. Billard, “Dynamical system modulation for robot learning via kinesthetic demonstrations,” *Robotics, IEEE Transactions on*, vol. 24, no. 6, pp. 1463–1467, 2008.

[29] D. H. García, C. A. Monje, and C. Balaguer, “Framework for learning and adaptation of humanoid robot skills to task constraints,” in *ROBOT2013: First Iberian Robotics Conference*, pp. 557–572, Springer, 2014.

[30] K. Kronander, M. Khansari-Zadeh, and A. Billard, “Learning to control planar hitting motions in a minigolf-like task,” in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pp. 710–717, 2011.

[31] S. M. Khansari-Zadeh and A. Billard, “Learning stable nonlinear dynamical systems with gaussian mixture models,” *Robotics, IEEE Transactions on*, vol. 27, no. 5, pp. 943–957, 2011.

[32] D. H. Grollman and A. G. Billard, “Robot learning from failed demonstrations,” *International Journal of Social Robotics*, vol. 4, no. 4, pp. 331–342, 2012.

[33] S. Michieletto, A. Rizzi, and E. Menegatti, “Robot learning by observing humans activities and modeling failures,” in *IROS workshops: Cognitive Robotics Systems (CRS2013)*, IEEE, 2013.

[34] A. Rai, G. de Chambrier, A. Billard, *et al.*, “Learning from failed demonstrations in unreliable systems,” in *Humanoids 2013*, no. EPFL-CONF-188281, 2013.

[35] D. Kulic and Y. Nakamura, “Incremental learning of human behaviors using hierarchical hidden markov models,” in *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pp. 4649–4655, IEEE, 2010.

[36] S. Calinon, F. D’halluin, E. Sauser, D. Caldwell, and A. Billard, “Learning and reproduction of gestures by imitation: An approach based on Hidden Markov Model and Gaussian Mixture Regression,” *IEEE Robotics and Automation Magazine*, vol. 17, pp. 44–54, 2010.

[37] A. Jain, T. Joachims, and A. Saxena, “Learning trajectory preferences for manipulators via iterative improvement,” *CoRR*, vol. abs/1306.6294, 2013.

[38] P. Englert, A. Paraschos, J. Peters, and M. P. Deisenroth, “Model-based imitation learning by probabilistic trajectory matching,” in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2013.

[39] B. Kim, A.-m. Farahmand, J. Pineau, and D. Precup, “Learning from limited demonstrations,” *Advances in Neural Information Processing Systems (NIPS)*, 2013.

[40] S. H. Lee and I. H. Suh, “Skill learning and inference framework for skilligent robot,” in *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pp. 108–115, 2013.

[41] J. Kober, J. A. Bagnell, and J. Peters, “Reinforcement learning in robotics: A survey,” *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[42] P. Kormushev, S. Calinon, and D. G. Caldwell, “Reinforcement learning in robotics: Applications and real-world challenges,” *Robotics*, vol. 2, no. 3, pp. 122–148, 2013.

[43] J. Peters, S. Vijayakumar, and S. Schaal, “Reinforcement learning for humanoid robotics,” in *Proceedings of the third IEEE-RAS international conference on humanoid robots*, pp. 1–20, 2003.

[44] J. Kober and J. Peters, “Policy search for motor primitives in robotics,” *Machine Learning*, vol. 84, no. 1-2, pp. 171–203, 2011.

[45] P. Kormushev, S. Calinon, and D. G. Caldwell, “Robot motor skill coordination with em-based reinforcement learning,” in *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pp. 3232–3237, IEEE, 2010.

[46] M. Deisenroth, D. Fox, and C. Rasmussen, “Gaussian processes for data-efficient learning in robotics and control,” 2013.

[47] J. Peters, K. Mülling, and Y. Altun, “Relative entropy policy search.,” in *AAAI*, 2010.

[48] A. Billard, A. K. Tanwani, *et al.*, “Transfer in inverse reinforcement learning for multiple strategies,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems, 2013*, no. EPFL-CONF-187590, 2013.

[49] B. D. Ziebart, A. L. Maas, J. A. Bagnell, and A. K. Dey, “Maximum entropy inverse reinforcement learning.,” in *AAAI*, pp. 1433–1438, 2008.

[50] E. Klein, B. Piot, M. Geist, and O. Pietquin, “A cascaded supervised learning approach to inverse reinforcement learning,” in *Machine Learning and Knowledge Discovery in Databases*, pp. 1–16, Springer, 2013.

[51] B. Michini, M. Cutler, and J. P. How, “Scalable reward learning from demonstration,” in *IEEE International Conference on Robotics and Automation, Karlsruhe, Germany*, 2013.

[52] M. Geist, E. Klein, B. PIOT, Y. Guermeur, and O. Pietquin, “Around inverse reinforcement learning and score-based classification,” in *1st Multidisciplinary Conference on Reinforcement Learning and Decision Making (RLDM 2013)*, October 2013.

[53] P. Abbeel and A. Y. Ng, “Apprenticeship learning via inverse reinforcement learning,” in *Proceedings of the twenty-first international conference on Machine learning*, p. 1, ACM, 2004.

[54] A. Boularias, O. Krömer, and J. Peters, "Structured apprenticeship learning," in *Machine Learning and Knowledge Discovery in Databases*, pp. 227–242, Springer, 2012.

[55] A. Boularias and B. Chaib-Draa, "Bootstrapping apprenticeship learning," in *Advances in Neural Information Processing Systems*, pp. 289–297, 2010.

[56] U. Syed, M. Bowling, and R. E. Schapire, "Apprenticeship learning using linear programming," in *Proceedings of the 25th international conference on Machine learning*, pp. 1032–1039, ACM, 2008.

[57] S. Levine and V. Koltun, "Continuous inverse optimal control with locally optimal examples," *arXiv preprint arXiv:1206.4617*, 2012.

[58] B. Michini and J. P. How, "Bayesian nonparametric inverse reinforcement learning," in *Machine Learning and Knowledge Discovery in Databases*, pp. 148–163, Springer, 2012.

[59] R. Akrour, M. Schoenauer, and M. Sebag, "Preference-based policy learning," in *Machine Learning and Knowledge Discovery in Databases*, pp. 12–27, Springer, 2011.

[60] R. Akrour, M. Schoenauer, and M. Sebag, "Preference-based reinforcement learning," in *Choice Models and Preference Learning Workshop at NIPS*, vol. 11, 2011.

[61] C. Wirth and J. Fürnkranz, "Preference-based reinforcement learning: A preliminary survey," in *Proceedings of the ECML/PKDD-13 Workshop on Reinforcement Learning from Generalized Feedback: Beyond Numeric Rewards*, 2013.

[62] C. A. Rothkopf and C. Dimitrakakis, "Preference elicitation and inverse reinforcement learning," in *Machine Learning and Knowledge Discovery in Databases*, pp. 34–48, Springer, 2011.

[63] M. Deisenroth and C. E. Rasmussen, "Pilco: A model-based and data-efficient approach to policy search," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 465–472, 2011.

[64] M. P. Deisenroth, C. E. Rasmussen, and D. Fox, "Learning to control a low-cost manipulator using data-efficient reinforcement learning," in *Robotics: Science and Systems*, 2011.

[65] A. Kupcsik, M. Deisenroth, J. Peters, and G. Neumann, "Data-efficient generalization of robot skills with contextual policy search," in *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2013.

[66] J. Kober, E. Oztop, and J. Peters, "Reinforcement learning to adjust robot movements to new situations," in *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Three*, pp. 2650–2655, AAAI Press, 2011.

[67] J. Kober and J. Peters, "Learning motor primitives for robotics," in *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pp. 2112–2118, IEEE, 2009.

[68] F. Stulp, O. Sigaud, *et al.*, "Policy improvement methods: Between black-box optimization and episodic reinforcement learning," 2012.

[69] E. Theodorou, J. Buchli, and S. Schaal, "A generalized path integral control approach to reinforcement learning," *The Journal of Machine Learning Research*, vol. 9999, pp. 3137–3181, 2010.

[70] E. Rückert and A. d'Avella, "Learned parametrized dynamic movement primitives with shared synergies for controlling robotic and musculoskeletal systems," *Frontiers in computational neuroscience*, vol. 7, 2013.

[71] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[72] F. Stulp, E. A. Theodorou, and S. Schaal, "Reinforcement learning with sequences of motion primitives for robust manipulation," *Robotics, IEEE Transactions on*, vol. 28, no. 6, pp. 1360–1370, 2012.

[73] Universal Robots, "Collaborative Robot Solutions." http://www.universal-robots.com/GB/Products.aspx. [Online; accessed January 2014].

[74] D. Malan, "Representation of how the axis and angle rotation representation can be visualized." http://commons.wikimedia.org/wiki/File:Euler_AxisAngle.png. [Online; accessed July 2014].

[75] S. Niekum, "ROS DMP package." http://wiki.ros.org/dmp. [Online; accessed January 2014].

[76] F. Stulp, "DmpBbo – a c++ library for black-box optimization of dynamical movement primitives.," 2014.

[77] C. G. Atkeson, "Using locally weighted regression for robot learning," in *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*, pp. 958–963, IEEE, 1991.

[78] D. Ruppert and M. P. Wand, "Multivariate locally weighted least squares regression," *The annals of statistics*, pp. 1346–1370, 1994.

[79] Mwtoews, " Normal distribution curve that illustrates standard deviations." http://commons.wikimedia.org/wiki/File:Standard_deviation_diagram.svg. [Online; accessed July 2014].

[80] I. A. Sucan and S. Chitta, "MoveIt!." http://moveit.ros.org. [Online; accessed August 2014].

[81] R. Shadmehr, *The computational neurobiology of reaching and pointing: a foundation for motor learning*. MIT press, 2005.

[82] T. Matsubara, S.-H. Hyon, and J. Morimoto, "Learning parametric dynamic movement primitives from multiple demonstrations," *Neural Networks*, vol. 24, no. 5, pp. 493–500, 2011.

[83] V. Tangkaratt, S. Mori, T. Zhao, J. Morimoto, and M. Sugiyama, "Model-based policy gradients with parameter-based exploration by least-squares conditional density estimation," *Neural Networks*, 2014.

[84] T. Zhao, H. Hachiya, V. Tangkaratt, J. Morimoto, and M. Sugiyama, "Efficient sample reuse in policy gradients with parameter-based exploration," *Neural computation*, vol. 25, no. 6, pp. 1512–1547, 2013.

[85] R. Busa-Fekete, B. Szörényi, P. Weng, W. Cheng, and E. Hüllermeier, "Preference-based reinforcement learning: evolutionary direct policy search using a preference-based racing algorithm," *Machine Learning*, pp. 1–25, 2014.

[86] M. Schoenauer, R. Akrour, M. Sebag, and J.-c. Souplet, "Programming by feedback," in *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pp. 1503–1511, 2014.

[87] Open Source Robotics Foundation, "About ROS." http://www.ros.org/about-ros/. [Online; accessed August 2014].

[88] Open Source Robotics Foundation, "ROS Concepts." http://wiki.ros.org/ROS/Concepts. [Online; accessed August 2014].

[89] Open Source Robotics Foundation, "Gazebo." http://gazebosim.org/. [Online; accessed August 2014].

[90] Open Source Robotics Foundation, "Rviz." http://wiki.ros.org/rviz. [Online; accessed August 2014].

[91] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, pp. 72–82, December 2012. http://ompl.kavrakilab.org.

[92] Rice University, "LBKPIECE1 Class Reference." http://ompl.kavrakilab.org/classompl_1_1geometric_1_1LBKPIECE1.html#gLBKPIECE1. [Online; accessed August 2014].

[93] M. J. Baker, "Maths - Rotation conversions." http://www.euclideanspace.com/maths/geometry/rotations/conversions/. [Online; accessed July 2014].

[94] S. M. LaValle, "Yaw, pitch, and roll rotations." http://planning.cs.uiuc.edu/node102.html. [Online; accessed July 2014].

[95] Dutch Society for Precision Engineering, "DSPE Conference 2014." http://www.dspe-conference.nl/. [Online; accessed August 2014].

Glossary

List of Acronyms

BN	Bayesian Network
DMM	Donut Mixture Model
DMP	Dynamic Movement Primitive
DoFs	Degrees of Freedom
GMM	Gaussian Mixture Model
GMR	Gaussian Mixture Regression
IRL	Inverse Reinforcement Learning
LWR	Locally Weighted Regression
MDP	Markov Decision Process
PBRL	Preference-based Reinforcement Learning
PGPE	Policy Gradients with Parameter based Exploration
PPL	Preference-based Policy Learning
RL	Reinforcement Learning
ROS	Robot Operating System
SEDS	Stable Estimator of Dynamical Systems
TU Delft	Delft University of Technology

List of Symbols

π	Policy
π^*	Optimal policy
τ	Temporal scaling factor
θ	Policy parameters
θ^*	Optimal policy parameters
D	Damping constant
$f(s)$	Forcing term
g	Generalized goal position
$J(\pi)$	Average return
K	Spring constant
$R(\tau)$	Reward
s	Phase variable
v	Generalized velocity
w_i	Adjustable weights of the DMP
x	Generalized position
x_0	Generalized start position

ALTEN