SoftBank Robotics

Learning By Demonstration

- Internship Report -

Author:
Noëlie RAMUZAT
SPID - Robotic option
Promotion year 2018

Supervisors:
Maxime BUSY
Benoît ZERR

August 23, 2018

ENSTA Bretagne

Acknowledgements

I would first like to thank my company supervisor Maxime Busy who has enabled me to realize this internship and provided me suggestions and encouragements, helped me to coordinate my project especially in writing this report. Furthermore I would also like to acknowledge with much appreciation the ProtoLab team, Maxime Caniot, Edouard Lagrue, Axel Lefrant and Jean-Marc Montanier, for their comments and advices, and for having integrated me to their team.

Finally I would like to thank M. Jaulin and M. Zerr to have answered my questions and provided their help during my internship.

Résumé

La réalisation de tâches adaptables à différentes situations constitue une problématique actuelle en robotique. Afin d'éviter de programmer un comportement décisionnel pouvant se révéler incomplet ou faillible, des techniques d'apprentissage utilisant le Machine Learning (ML) ont été développées. Ce rapport présente une solution réalisée dans le cadre du stage *Learning by Demonstration* au sein de la société *SoftBank Robotics Europe*.

Le but de ce projet est d'apprendre au robot humanoïde *Pepper* un mouvement approchant une de ses mains d'un objet à attraper, tout en évitant les obstacles sur la trajectoire. Les méthodes de Learning from human Demonstration (LfD) permettent au robot de s'adapter à de nombreuses situations, comme des déplacements du but à atteindre, des changements de position de départ ou la présence d'obstacles.

Ces méthodes permettent à un utilisateur d'enseigner simplement un mouvement à un robot par le biais de démonstrations visuelles ou physiques. Dans le premier cas l'utilisateur effectue la tâche à enseigner devant les caméras du robot, tandis que dans le second la tâche est réalisée en manipulant directement les parties du robot.

Le stage se concentre sur les techniques d'apprentissage physiques, dit *kinesthésiques*. Après une étude des méthodes permettant d'obtenir les modèles des tâches à partir de leur démonstrations, la solution développée emploie des Dynamic Movement Primitives (DMP).

Les DMPs modélisent la trajectoire du mouvement démontré par des équations différentielles, qui peuvent être complétées par un terme permettant l'ajout de comportements d'évitement obstacles. La mise en place d'une solution de détection d'obstacles, réalisée grâce à la caméra de profondeur du robot *Pepper*, a été nécessaire afin de déterminer ce terme. L'image de profondeur obtenue est segmentée afin d'isoler les obstacles, qui sont représentés par un nuage de points en trois dimensions. Ceux-ci sont finalement pris en compte par les équations des DMP dans le terme additionnel, qui leur associe un caractère répuslif.

Abstract

The performance of a task in different situations represents an actual issue in the robotic field. To avoid programs creating a decisional behavior that can be incomplete, some techniques using the Machine Learning (ML) have been developed. This report presents a solution realized during the internship *Learning by Demonstration*, proposed by the *SoftBank Robotics Europe* company.

The aim of this project is to make the humanoid *Pepper* robot learn an arm movement, allowing it to approach its hand to an object, while avoiding obstacles. The Learning from human Demonstration (LfD) methods allow the robot to be adaptable to different situations, such as goal movements, changes of starting position or obstacles avoidance.

These techniques provide a way for the user to easily teach a movement to a robot by giving it visual or physical demonstrations. In the first case, the user realizes the task in front of the cameras of the robot. In the second case, the task is realized by moving the different body parts of the robot.

The internship focuses on the physical techniques of learning, called *kinesthetics*. After a study of the different techniques that can be used to obtain the tasks models from their demonstrations, the selected one uses the Dynamical Movement Primitives (DMP) method.

This method models the trajectory of the demonstrated task by differential equations, which can be completed by a term containing information on obstacles to be avoided. The implementation of a solution of obstacles detection, realized with the depth camera of the robot *Pepper*, has been necessary in order to determine this term. The depth image obtained is segmented in order to extract the obstacles, which are then represented as a three-dimensional point-cloud. Finally, these points are taken into account in the equations of the DMP in the additional term, which assigns them a repulsive behavior.

Table of Contents

Introduction

Subject Presentation

The aim of this internship is to make the humanoid robot *Pepper* learn an arm movement, allowing the robot to approach its hand to an object, while avoiding obstacles. The two main points of the subject are the creation and learning of an adaptive trajectory, and the detection of the obstacles on this trajectory.

The first point focuses on the methods of Programming by Demonstration (PbD), in which the robot is given a physical guidance of the task by an operator. This operator executes a demonstration of the movement to be learned by moving the different parts of the robot, such as its arm. This approach is called *kinesthetic teaching* and is part of the Learning from human Demonstration (LfD) techniques[2].

These methods have to face key issues raised by the learning by demonstrations problems, which can be described by the following questions[2]:

- What to imitate ?
- How to imitate ?
- Who to imitate ?
- When to imitate ?

The two last questions have not really been explored so far and are not within the scope of the internship. Currently, in most cases, the robot enters a learning phase and registers the movements given by any operator. Furthermore the robot does not perform the acquired tasks autonomously, but when the user decides it.

On the contrary, the two first questions are inherent in the problematic of the subject: Which data are relevant to represent a movement and should be registered ? How can the robot learn these data appropriately in order to reproduce the movement autonomously, in different situations ? To answer these points, the methods of LfD can be split into three stages: the tasks representation, their learning and their autonomous and adaptive realization. After a state of the art on these techniques, the DMP method (described in part 2.2.2) has been chosen.

Concerning the second point, several methods exist to detect obstacles and they greatly depend on the available systems in the scene. In this project the obstacles to be detected

are the ones close to the robot, which can impeach its arm to reach an object, and are different depending on the situations. Thus, this report will not explain how to detect and avoid obstacles during a navigation task or in a monitored environment. To this extend, the presented methods will only rely on the robot's embedded depth camera, which presents different possibilities explained in Section 3.5.

Objectives

The main objective of this internship is to implement a solution which registers kinesthetic demonstrations and creates a general trajectory from them, taking into account a goal and obstacles. A second objective is to efficiently detect the obstacles with the depth camera of *Pepper* and to insert them in the previous solution.

Report Outline

In the following part is first described the activities of the company and its departments. Next, the background researches that have been realized in the scope of the internship are presented. These researches cover possible ways of collecting the demonstration data; the different methods of LfD and procedures to improve them.

Then, the implementation of the chosen solution is described, containing the recording of the movements, the obstacles detection, the creation and learning of the trajectory, its performance and its optimization.

Finally the results of the implemented solution are presented, followed by a conclusion presenting the perspectives for a future work.

Chapter 1

Company Presentation

1.1 General Description

Softbank Robotics is one of the world leading companies in the field of humanoid robotics. It was originally named *Aldebaran Robotics* in 2005, and was the first French firm industrializing humanoid robots. The first robot is *NAO*, created for the educational and research fields. It has a small size and possesses a wide range of possible moves. The second created robot is *Romeo*, taller in size than *NAO*. It is designed to help the elderly, and is used as an internal research platform within the firm. *Aldebaran Robotics* has been acquired by the Japanese group *SoftBank* after their partnership on the creation of the robot *Pepper* (in 2014) and renamed *Softbank Robotics* in 2016.

Pepper has been designed to welcome the customers in the shops of *SoftBank* in Japan. It possesses three wheels used to navigate. Recently, the *Android* tablet has been replaced on its torso, allowing the development of different applications directly accessible on the robot, such as sending emails or displaying the website of a shop.

The group possesses four antennas in the world, in China, America, Japan and Paris. The Paris antenna is the biggest one in terms of employees and is responsible of most of the research and development projects. The center is divided into different departments:

- The Customer Services
- The Human Resources
- The Finance and Legal
- The Marketing, Sales and Communication
- The Business Development
- The Software
- The Hardware
- The Innovation

I realize my internship within the Innovation department, in the software section, in the *Protolab* team.

1.2 Innovation Department

The Innovation department focuses on the research and implementation of the future functionalities of the robots. Almost all of the collaborative projects are handled by this department, in partnership with European and worldwide laboratories. These projects are part of the EU framework programmes to support Research and Innovation. The Innovation can be subdivided in different sections:

- The Mechatronics
- The Software
- The Design

Each profession necessary to develop a prototype is represented in this department and the sections collaborate with each others on common projects.

1.2.1 Mechatronics

The Mechatronics section handles low-level implementation, involving the hardware areas of the robotics field such as the electronic, electromagnetism, systems engineering, mechanical and control engineering. They study the new possibilities brought by the researches in mechanics and control to improve the hardware and mechanical parts of the robots. For instance, they test new motors and actuators in order to reduce the heat spread by them. They attempt to increase the capabilities of the robots without creating important changes in the design, by adding new cameras or sensors for instance.

1.2.2 Design

The Design section is in charge of the appearance of the robots and their new external characteristics. They are in charge of the aestheticism of the robots and have to create new shapes that respect the identity of the previous ones. They create the sketches of the shape and once it is validated they build models, integrating some hardware parts to validate them.

1.2.3 Software

The Software section handles high-level implementations, developing algorithms adding functionalities to the robots. The activities are divided into two sections, the AILab and the ProtoLab. The first effectuates researches on the current advanced papers in robotics to determine the new promising algorithms. The second one works on exploitable concepts given by the research to implement and integrate into the robot's software.

AILab

It is the main research center of the company working on fundamental algorithms in the artificial intelligence field. Currently the researches are oriented on unsupervised learning applied on robotics development. In particular they focus on the Sensorimotor Contingencies Theory (SMCT) which aims to account for the phenomenal character of perceptual

experience[6]. They publish papers about their work and, when they identify a technology that can be adapted to the industry, they transmit their researches to the Protolab which transforms the concept into an implementation.

ProtoLab

This team has three main goals:

- Extract research concepts that can be converted in exploitable technologies and can be integrated to industry
- Find new concepts of robots or prototypes
- Handle the Collaboratives Projects

They deal with the actual hardware system of the robot to address complex subjects such as grasping, detect human skeletons, Internet of Things (IoT)... My internship is part of the "grasping" project of the ProtoLab: the robot has to reach a graspable object with its hand before realizing the grasping movement. Currently this reaching movement is hard-coded, the solution proposed by this internship would allow the arm of *Pepper* to reach a relevant position while dynamically avoiding obstacles.

Chapter 2

Background Researches

2.1 Interfaces for Demonstration

In order to collect the movement information, different interfaces are available on robots. In this section, different methods using these interfaces are presented, in particular the kinesthetic one (used in the project).

2.1.1 Kinesthetic Method

This method is based on a physical demonstration of the movement, the human has to move the robot parts to realize the desired task. The motion is recorded directly by registering the position's variation of the robot joints. In contrary to the following methods, this method has no correspondence problem, there is no need to adapt the recorded solution to the robot. Indeed, the demonstrator uses the robot capabilities in its own referential to obtain the movement[10]. Moreover with this approach it is intuitive to teach a movement to the robot, the operator has just to make the robot do it (see Figure 2.1).

Figure 2.1: Kinesthetic Teaching.

The main drawback of this method is that the human can have some difficulties to execute the movement[2]. If a task necessitates the use of several parts of the robot, the tutor may experience some troubles to move them simultaneously, especially to reach the desired orientation. To achieve a good precision, the demonstrator will most of the time use more degrees of freedom than the robot for the motion. As a result complex tasks cannot be done

with this method, except if they are sequenced into smaller simplest ones and then combined. This technique has been chosen for *Pepper*'s learning because of the simplicity of the task and of the small amount of computing power and algorithms needed to obtain a good accuracy[10].

2.1.2 Other Methods

Human Motions Recording and Tracking

This method is based on the visual sensors of the robot, skeleton tracking systems or external motion sensors the demonstrator has to wear[2]. The robot will track the human movements with its cameras, registering all the human body or only the skeleton. Another form of teaching can be performed with body's keypoints, given by motion tracking sensors worn by the demonstrator.

The use of this type of recording necessitate a robust tracking system , but it has the advantage to be external, thus giving a precise measurement of the movement. The movement first has to be extracted from the environment (for the vision and tracking) and then to be adapted to the robot. Indeed the adaptability of the solution to the kinematic model of the robot (as articular joint position for instance) is the main problem of this solution. *Pepper* has no legs, and only one degree of freedom in its hand. The demonstration's mapping will be hard to perform. Moreover an important limitation will be the computational power needed to have a robust solution.

Teleoperation

This method is based on teleoperation systems, such as joysticks or others remote control devices, allowing to remotely control the robot. Some external devices can be used to collect precise demonstration data, for instance sensitive gloves can be used to gather effort informations. Moreover the tutor executes the demonstration externally, no difficulties should be encountered to achieve the task, it can be done from a distance.

The disadvantages of this method is the understanding of the remote control solution and its configuration to map the kinematic model of the robot. This is the correspondence problem of this method[10]. The solution has to be simple to use but also covers all the complexity of the model of the robot's chain. In the case of *Pepper* such solution exists, but its accuracy is not optimal: there is slack in the robot's arms leading to imprecision in the kinematic solver, the workspace of the robot is limited and the considered kinematic chain owned many singularities (see Section 3.1). The solution is much less precise than what could be obtained through kinesthetic teaching, that is why this method has been excluded.

2.2 Learning from Demonstration Algorithms

After recording the movement to be learned, the first step is to obtain the mathematical representation of this one. This model should not be a copy of the trajectory but an adaptation of this one in order to be stable, robust against perturbations and adaptive to new goals, changes of starting position and obstacles.

2.2.1 Gaussian Mixture Model

This method extracts a set of primitive behaviors or actions from the given task and classifies them. Then it learns how to reproduce the movement from the individual behavior list and how to generalize it to new situations. This approach can be combined with a trajectory level encoding which builds models operating in continuous spaces. For instance it encodes the angular position of the robot's joints or the Cartesian position, speed or torque of an end-effector by mapping the sensory inputs to motor outputs and velocities[5, 10].

The Gaussian Mixture Regression (GMR) technique is the most used trajectory level encoding, using a Gaussian Mixture Model (GMM) tuned with an Expectation Maximization (EM) algorithm[4] (see Appendix A). It extracts the underlying task constraints from several demonstrations which take the form of the desired velocity profiles for the joint angles of the robot chains. This representation takes into account the changing correlations across the movement variables and the variations observed among multiple demonstrations. Moreover, this method creates a statistical representation of the problem, easily composed with others machine learning algorithms (see Chapter 2.2.3).

One drawback of this approach is the difficulty to add an obstacle avoidance behavior to the method since this representation is not designed to deal with perturbations without replaning the whole movement[5].

System Implementation

The kinematic chain of the robot is represented at each time step by its joint angles, in the variable $\xi(t)$. The method consists in the extraction of the key features of several demonstrations $[\dot{\xi^{demo}}(t)]$, and in the adaptation of the computed trajectory to different initial and final conditions. It first generalizes each inputed movements thanks to the GMM, then the parameters of the model are selected with the EM algorithm (see Appendix A). Finally the GMR is applied to reconstruct a general form of the task $\dot{\xi}^m(t)$.

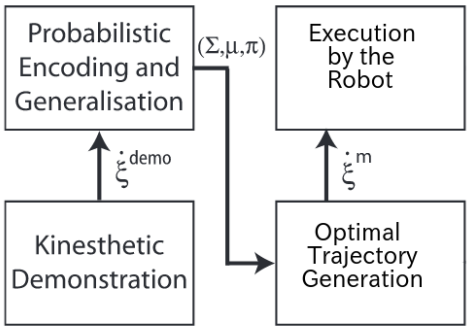

Figure 2.2: Conceptual sketch of the GMM system, using GMR[4].

Probabilistic Encoding and Generalization of the trajectories

To generalize the demonstrations ξ^{demo}, the joint distribution is modeled as a GMM. Mixture modeling is mostly used for density approximation of continuous or discrete data. It searches for a compromise between model complexity and variations of the training data, allowing flexibility. A mixture model of K components is defined by a probability density function [4]:

$$p(\xi_j) = \sum_{k=1}^{K} p(k)\, p(\xi_j|k) \tag{2.1}$$

- ξ_j is a datapoint
- $p(k)$ is the prior
- $p(\xi_j|k)$ is the conditional probability density function

In the case of the system of Figure 2.2, the input variable is the time t relative to the registered trajectory and the output variables are the velocities $\dot{\xi}$ of the joints of a robot arm[10]. By joining these variables in a vector v, the probability density function of the GMM can be expressed as:

$$\begin{aligned}
v &= [t\ \dot{\xi}^T]^T \\
p(v) &= \sum_{k=1}^{K} \pi_k\, N(v;\mu_k;\Sigma_k) \\
N(v;\mu_k;\Sigma_k) &= \frac{1}{\sqrt{(2\pi)^D\,|\Sigma_k|}}\, exp(-\frac{1}{2}(v-\mu_k)^T\Sigma_k^{-1}(v-\mu_k)) \\
\mu_k &= [\mu_{k,t}^T\ \mu_{k,\dot{\xi}}^T]^T \\
\Sigma_k &= \begin{bmatrix} \Sigma_{k,t} & \Sigma_{k,t\dot{\xi}} \\ \Sigma_{k,\dot{\xi}t} & \Sigma_{k,\dot{\xi}} \end{bmatrix}
\end{aligned} \tag{2.2}$$

- π_k is the prior, the weighting factor
- $N(v;\mu_k;\Sigma_k)$ is a Gaussian function of mean μ_k and covariance matrix Σ_k
- K is the number of Gaussian in the mixture
- D is the dimensionality of the Gaussians

This GMM is then used to train the EM algorithm with the demonstrations as training set. The EM algorithm iteratively estimates the *Maximum A Posteriori* of the parameters. It returns a joint probability density function for the input and the output (see Appendix A).

Optimal Trajectory Generation

Once the GMM has been trained by the EM algorithm it is possible to recover the expected output variable $\hat{\dot{\xi}}_m$ given the observed input variable t[4, 8]:

$$\begin{aligned} \dot{\xi}^m(t) &= \sum_{k=1}^{K} h_k(t)\,(\mu_{k,\dot{\xi}} + \Sigma_{k,\dot{\xi}t}\,\Sigma_{k,t}^{-1}(t-\mu_{k,t})) \\ h_k(t) &= \frac{\pi_k\,\mathcal{N}(t;\mu_{k,t};\Sigma_{k,t})}{\sum_{k=1}^{K}\pi_k\,\mathcal{N}(t;\mu_{k,t};\Sigma_{k,t})} \\ \Sigma_{\dot{\xi}}(t) &= \sum_{k=1}^{K} h_k^2(t)\,(\Sigma_{k,\dot{\xi}} - \Sigma_{k,\dot{\xi}t}\,\Sigma_{k,t}^{-1}\Sigma_{k,t\dot{\xi}}) \end{aligned} \tag{2.3}$$

To conclude on this method, after training, the GMM is used to generate a movement by taking the expected velocities $\dot{\xi}^m(t)$ conditioned on time t. This movement is then reproduced by the robot.

In the Figure 2.3 is presented the GMM method applied on the arm of an humanoid robot[8, 4]. The GMM is trained with 26 demonstrations of an arm movement putting a ball in a box. $\dot{\xi}_1^m$ to $\dot{\xi}_4^m$ represents the four joint angles of the robot arm. On the left, the thin lines represent the demonstrations and the thick lines the generalization $\dot{\xi}^m$ retrieved by using the GMR. The ellipses represent the Gaussian components of the joint probability distribution. On the right, the reproduced trajectories (dash-dotted line) are qualitatively similar to the modulation trajectory $\dot{\xi}_1^m$ (first joint, solid line), although they reach the goal from different initial positions.

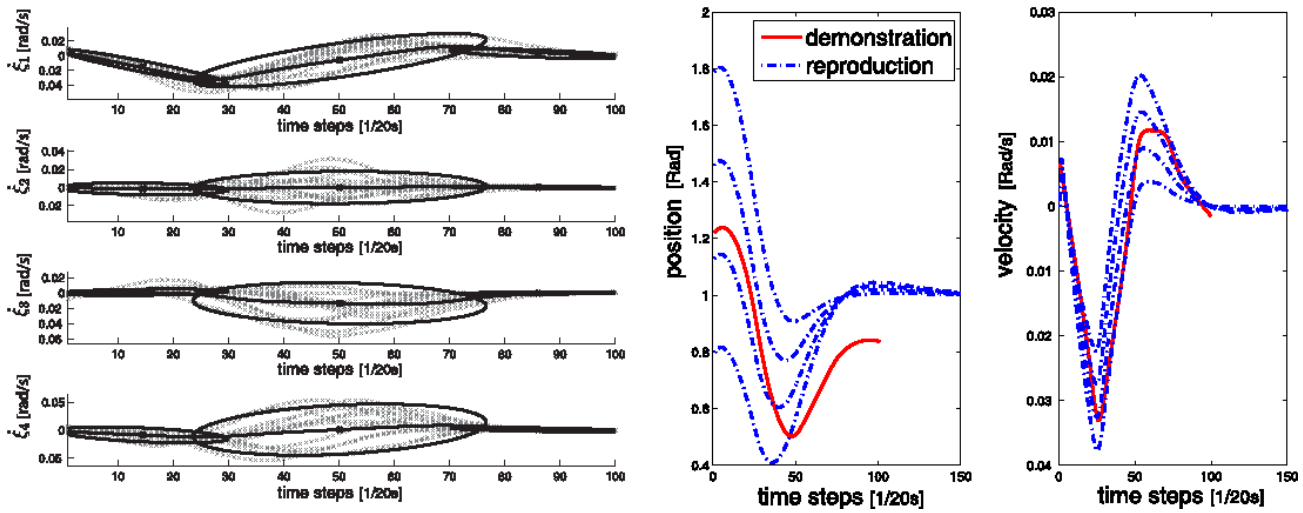

Figure 2.3: Gaussian Mixture Model and Generated trajectories.

2.2.2 Dynamic Movement Primitives

This method decomposes a task into movements primitives, that is to say in mathematical representations of the basic elements of an action, its primitives. They first have been introduced by Ijspeert et al.[13, 11] who were searching for a way to create Control Policy (CP) for movement planning based on attractor dynamics. This CP leads to DMPs which are autonomous nonlinear differential equations involving the positions, velocities and accelerations of a given joint.

This dynamical system encodes a trajectory from its initial state to its final state. This method does not require time-indexing and is robust against perturbations, thanks to the characteristics of the differential equations[11, 12]. Finally, "forcing" terms are added to this model, allowing the learning of complex movements. DMPs are also very simple to learn by a robot, because the weights of the forcing terms are learned separately and independently of each other to reduce the state space. Therefore, learning DMPs can be done very quickly and efficiently even if a movement involves multiple degrees of freedom.

The idea behind the DMP is to use simple formulations of attractor equations to encode the basic behavioral patterns and then to use statistical learning to adjust the obtained system to the task. This learning is obtained with the forcing terms.

In summary, DMPs anchors a linear learning system in the phase space of a canonical dynamic system. The linear system contains nonlinear basis functions to characterize the spatiotemporal path of the DMPs. The dynamical system has attractor properties to ensure the stability of the solution. It allows to learn complex attractor landscapes of non-linear differential equations without endangering the asymptotic convergence to the goal state.

The DMPs have several properties such as[11, 12]:

- Stability: due to the use of a critically damped system.
- Invariance: conservation of the scale of the movement and adaptation to changes of start, end point and/or duration.
- Robustness against Perturbations: differential equations are easily modifiable and can include additional obstacles avoidance terms.

However, on the contrary of the GMM, this method does not take into account the changing correlations between the movement variables and the variations observed among multiple demonstrations.

Controller

The kinematic variables of the model are converted to angular position through inverse kinematics solver. In this way, standard control techniques taking kinematic trajectory plans as input can be used for the execution of DMPs.

Attractor Equations

Any dynamical system can be used to determine the attractor equations, as long as it is stable and can be adjusted with non linear terms[13, 23]. The most commonly used system is the spring damper system, which is flexible enough to fit complex motor behaviors and to avoid instability.
The stable system for a basic attractive point g can be expressed in the second order dynamics as follow:

$$\begin{aligned} \tau \dot{z} &= \alpha_z \left(\beta_z \left(g - y\right) - z\right) \\ \tau \dot{y} &= z \end{aligned} \tag{2.4}$$

- g is the goal state
- α_z and β_z are time constants
- τ is a temporal scaling factor
- y and $\dot{y}$ are the desired position and velocity generated

The system 2.4 is a stable linear dynamic system with a unique attractor point and converges exponentially. With $\beta_z = \alpha_z/4$ the system is critically damped and converges without oscillations.
To achieve a more complex behavior, a non-linear function f representing the forcing terms can be added to the equation[12]:

$$
\begin{aligned}
\tau \dot{z} &= \alpha_z \left(\beta_z \left(g - y\right) - z\right) + f \\
\tau \dot{y} &= z \\
\tau \dot{x} &= -\alpha_x x \\
f(x) &= \frac{\sum_{i=1}^{N} \psi_i(x)\omega_i}{\sum_{i=1}^{N} \psi_i(x)} x \left(g - y_0\right) \\
\psi_i &= exp(-\frac{1}{2\sigma_i^2} \left(x - c_i\right)^2))
\end{aligned}
\tag{2.5}
$$

- ψ_i are Gaussian basis functions of width σ_i and center $c_i \in [0, 1]$
- $\alpha_x = \frac{\alpha_z}{3}$ constant, to keep a critically damped system
- ω_i are the weights of the basis function ψ_i
- $x \in [0, 1]$ represents the time evolution of the movement, it is equal to zero at the beginning and at the end of the task in order to suppress the action of the forcing function

In the system 2.5, the number of Gaussian basis functions ψ_i are chosen by the user and depends on the movement to imitate. The system is stable and asymptotically converges to the unique attractor point g. Moreover the scaling term $(g - y_0)$ in the function f assures the invariance properties of the trajectory.

There are other formulations of the DMPs depending on the system to express (discrete such as accelerator DMP or rhythmic such as oscillator DMP)[24] but this one is the most appropriate for the system of the project.

Obstacles Avoidance

To take into account the obstacles, a coupling term C_t can be added in the transformation equation in order to affect the spatial evolution of the system and create an avoiding behavior. The system becomes[11, 22]:

$$
\begin{aligned}
\tau \dot{z} &= \alpha_z \left(\beta_z \left(g - y\right) - z\right) + f + C_t \\
\tau \dot{y} &= z
\end{aligned}
\tag{2.6}
$$

This coupling term creates a movement perpendicular to the trajectory's direction when an obstacle is detected. In 3D, this behavior can be modeled by the following system when an obstacle is detected at a position o:

$$
\begin{aligned}
y &= [y_1\ y_2\ y_3]^T \\
g &= [g_1\ g_2\ g_3]^T \\
o &= [o_1\ o_2\ o_3]^T \\
C_t &= [C_{t,1}\ C_{t,2}\ C_{t,3}]^T \\
C_t &= \gamma\ R\ \dot{y}\ \theta\ exp(-\beta\theta) \\
\theta &= arccos(\ \frac{(o-y)^T \dot{y}}{|o-y||\dot{y}|}\ ) \\
r &= (o-y)\ \dot{y}
\end{aligned}
\tag{2.7}
$$

- y is the robot's end-effector position in 3D
- g is the goal position in 3D
- o is the obstacle position in 3D
- θ is the angle between the velocity vector $\dot{y}$ and the difference vector $(o-y)$ between the current position and the obstacle
- r is the vector perpendicular to the plan spanned by $\dot{y}$ and $(o-y)$
- R is the rotational matrix of 90 degrees about r to avoid the obstacle
- γ and β are constants parameters to adapt the movement

In the Figure 2.4 the simulated behavior of this system is represented with different starting positions (around the origin $y_i = [0,0,0]^T$) and the same goal ($y_g = [1,1,1]^T$). The obstacle is a red sphere (at the position $y_o = [0.5,0.5,0.5]^T$) and the initial trajectory is in green. The scales are expressed in meters. When the starting position is far from the obstacle, the avoiding movement is less curved around the obstacle, which is an intuitive behavior[11].

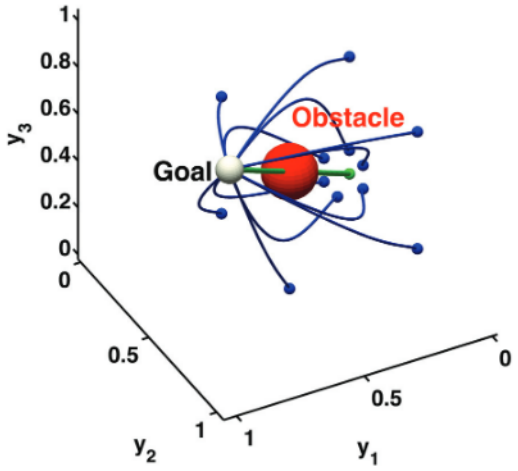

Figure 2.4: Obstacle avoidance with a coupling term[11].

Imitation Learning

To be able to adjust the obtained system to the demonstrated task it is necessary to learn the weights ω_i of the non linear function f. They will characterize the spatiotemporal path of the DMP. For a given trajectory $y_{demo}(t)$, $\dot{y}_{demo}(t)$ and $\ddot{y}_{demo}(t)$ with a duration T, the target function f can be expressed based on the equation 2.5 as [24, 11]:

$$\begin{aligned} f_{target} &= \tau^2 \ddot{y}_{demo} - \alpha_z \left(\beta_z \left(g - y_{demo}\right) - \tau \dot{y}_{demo}\right) \\ g &= y_{demo}(T) \end{aligned} \tag{2.8}$$

With this system, the function approximation problem is simpler, it suffices to adjust the terms of f so that this function fit as well as possible f_{target}. Moreover the number of basis function ψ_i have to be chosen by the user or determined by the approximation function. The more basis functions are put, the closest to the demonstrated movement the computed trajectory will be. A smaller number of these functions will create a smoother trajectory, which can be preferable depending on the movement to be performed.

There are several approximation functions which exist to find the weights ω_i, such as the Locally Weighted Regression (LWR) function, the Locally Weighted Projection Regression (LWPR) function or the Radial Basis Function Network (RBFN) function (see Appendix B).

Others methods can be implemented from these two principal techniques, some solutions using combinations of methods have been studied during this internship and are briefly presented in Appendix C.

2.2.3 Reinforcement Learning

When the final trajectory is obtained, it can present differences with the demonstrated ones. It is not an incorrect characteristic, the movement can be slightly different depending on the situation, as the movement has to be adaptable. The computed solution allows it but to some extends, it can be necessary to improve its adaptability. Algorithms can be used to generate different solutions from an initial one, in order to give the robot the ability to handle several types of situation. These algorithms are called Reinforcement Learning (RL) and aim to train the robot and improve the obtained solution. Thanks to these methods, the robot will react in an optimal way to different situations. The robot is able to learn by exploring different options obtained by scaling the demonstration's trajectory and thus discover new solutions.

These algorithms are based on Markov Decision Process (MDP): the robot is in a state and can perform actions, it has a probability to realize an action and a reward is associated to each performed action. At each time step, the robot observes the environment, choses an action to perform and the reward is determined. The goal of RL is to collect as much reward as possible. The reward function is adapted depending on the goal of the task to perform.

The general algorithm is expressed as follow:
The selection of the robot's action is modeled by a policy π, giving the probability to perform the action a in state s with parameters θ [20, 19, 16].

$$\begin{array}{lcl} \pi & : & S \times A \rightarrow [0,1] \\ \pi(a|s;\theta) & = & P(a_t = a \,|s_t = s,\, \theta) \end{array} \tag{2.9}$$

The system yields a scalar reward $r_t = r(a_t, s_t)$ after each action. For each policy π_θ, a state-value function $V^\pi(s)$ and a state-action value function $Q^\pi(a,s)$ are defined as:

$$\begin{array}{lcl} V^\pi(s) & = & E_\tau[\, \sum_{t=0}^{T} \gamma^t r_t \,|\, s = s_0 \,] \\ Q^\pi(a,s) & = & E_\tau[\, \sum_{t=0}^{T} \gamma^t r_t \,|\, s = s_0,\, a = a_0 \,] \end{array} \tag{2.10}$$

- T is the set of all possible paths
- $\gamma \in [0,1[$ is the discount factor

These functions estimate the amount of reward earned by being in a given state. The general goal of RL is to optimize the normalized expected return of the policy π with parameters θ defined by:

$$J(\theta) = E_\tau[(1-\gamma) \sum_{t=0}^{T} \gamma^t r_t \,|\, \theta] \tag{2.11}$$

The two most commonly used RL algorithms in problems dealing with multiple degrees of freedom systems are expressed in Appendix D.

Chapter 3

Implementation

In this part is presented the implementation of the solution on the *Pepper* robot in it's realization order. This section also covers the experiments that have been implemented and tested but discarded in the ultimate solution.

3.1 Recording the Movement

The first objective was to register the movement during a kinesthetic demonstration. To achieve this goal, a program in *Python* is implemented, using the *NAOqi* framework[1]. The *NAOqi* framework is the main software that runs on the *SoftBank Robotics*' robots and controls it. It owns an API allowing to interact with the robots.

In the program, the stiffness of the robot's body parts concerned by the demonstration is set to the minimum in order to allow the user to move them. These body parts are stored into an ordered array representing the kinematics chain of the robot involved in the movement, which will be used by the kinematic solver in Section 3.2. In this project the movement is an arm trajectory and the kinematics chain starts from the robot torso to its hand as shown in Figure 3.1 in *RVIZ*.

During the task demonstration, the position of the end effector (then end of the kinematics chain, one of the hand of *Pepper*) is retrieved in the Cartesian space in 6D (translation and rotation) with *NAOqi*. After this stage, the velocities and accelerations of these translations and rotations are computed. Finally all these data are saved in two text files, the first column corresponds to the time and the nine following columns correspond to the variables (position in 3D, velocity in 3D, acceleration in 3D, for the translations and rotations, see the algorithm 1).

The time step between two registered positions has to be balanced, if it is too big, the trajectory will be irregular and not precise, but if it is too small, the computational power needed to perform the DMP algorithms will be too large (because the number of data to process will be high).

[1] http://doc.aldebaran.com/2-1/dev/naoqi/index.html

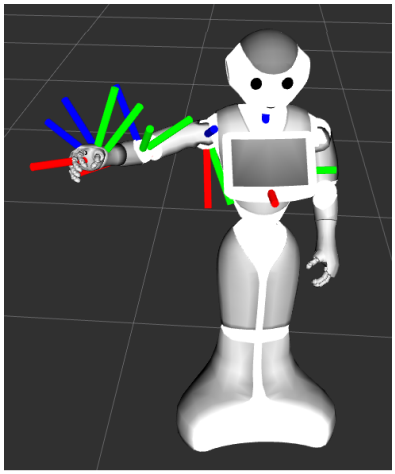

Figure 3.1: Kinematics chain of *Pepper* used in the project. The used joints are: RightShoulderPitch, RightShoulderRoll, RightElbowYaw, RightRElbowRoll, RightWristYaw, RightHand (or left).

Algorithm 1: Movement Recording Algorithm

Input: bodyParts, RobotIP
Output: ∅

```
1:  if not NAOqi.ConnectionSucessful(RobotIP) then
2:      return ERROR
3:  end if
4:  NAOqi.setStiffnesses(bodyParts, 0)
5:  timeStep ← chosenTimeStep
6:  translations ← []
7:  rotations ← []
8:  endEffector ← bodyParts[-1]
9:  endMove ← 0
10: while endMove == 0 do
11:     if actualTime - previousTime >= timeStep then
12:         // In reality in a background scheduler to have a constant time step
13:         cartesian, time ← NAOqi.getPosition(endEffector)
14:         translations.append( [ time, cartesian[0,3] ])
15:         rotations.append( [ time, cartesian[3,6] ])
16:     end if
17:     endMove ← userInput // non-blocking call
18: end while
19: total_translation = getVelocityAndAcceleration(translations)
20: total_rotation = getVelocityAndAcceleration(rotations)
21: saveFile("translation.txt", total_translation)
22: saveFile("rotation.txt", total_rotation)
```

There are two important choices that have been made here, the usage of the Cartesian space and the separation of the translations and rotations of the positions.

Firstly, the DMP algorithm can process the data, independently of its space representation. At the beginning, the movement was registered in the joints space of the robot. For an arm demonstration, the joint angles of the arm were saved relatively to the time step. The DMP algorithm returned a trajectory in the joint space which can be realized by the robot. But a problem appears when the obstacles have to be added to the equations: they cannot be expressed in the joint space of the robot with the detailed Equation 2.7. That is why the Cartesian space has been chosen afterward.

Secondly, the DMP algorithm can process the position in 6D directly but it has been chosen to implement two DMP, one on the translations and one on the rotations. This choice is once again due to the addition of the obstacle avoidance term C_t in the equations: indeed the orientation of the obstacles does not matters for the considered problem. The obstacles are implemented as a summation of small repulsive points with a fixed translation and these points have to be avoided independently of the object orientation. Thus, instead of setting a null rotation for them and process the translation and rotation together (which may leads to errors), only the DMP for the translation is completed with the obstacle avoidance term C_t.

3.2 Performing the Movement

Once the trajectory has been computed by the DMP algorithm, it is registered in two files: one for the translations and one for the rotations. With the *NAOqi* framework it is possible to apply angular positions onto the robot body parts. On *Pepper* there is no function that allows to directly set the Cartesian positions of an end-effector. Thus, unlike the previous algorithm, it is not possible to work exclusively in the Cartesian space.

To retrieve the desired angular coordinates from a Cartesian position, the usage of an inverse kinematics solver is necessary. The ProtoLab team had already created a solver as a service on the *Robot Operating System (ROS)* and the implemented program uses it. Finally, to perform the movement, the implemented program is in *Python* and uses the solver service to get the angular coordinates from the Cartesian positions stored in the files (see the algorithm 2). The choice of text files was made to examine the computed trajectory after the reproduction of the movement.

It is interesting to notice that the files containing the translations and the rotations will be modified during the execution of this algorithm. Indeed, the DMP algorithm is called periodically to take into account the moves of the goal (the object to reach with the hand of *Pepper*), thus the files will be updated to fit the computed trajectory. Therefore, the algorithm 2 is performed in an external thread, while the principal program checks the position of the goal and tracks it.

Algorithm 2: Movement Performance Algorithm

Input: bodyParts, fileTranslations, fileRotations, RobotIP
Output: $\emptyset$

```
1: if not NAOqi.ConnectionSucessful(RobotIP) then
2:     return ERROR
3: end if
4: timeStep ← chosenTimeStep
5: translations, time ← parseFile(fileTranslations)
6: rotations, _ ← parseFile(fileRotations)
7: for i = 0 ; i < length(time); i++ do
8:     if actualTime - previousTime >= timeStep then
9:         // In reality in a background scheduler to have a constant time step
10:        cartesian ← translations[i, :].append(rotations[i, :])
11:        joints ← serviceROSJointsFromCartesian(cartesian, bodyParts)
12:        NAOqi.setAngles(joints, bodyParts)
13:    end if
14: end for
```

3.3 Creation and Learning of the Trajectory

To compute the final trajectory, the DMP algorithm need the two files containing the movement variables, the goal position to reach (if it exist) and the obstacles positions (if they exist).

The goal, in this project, is a point near an object on which an *Aruco* marker[7] has been added in order to be detected by the robot *Pepper*. This detection is realized with the *aruco-ros* package, a *ROS* package of the *Aruco* Augmented Reality marker detector library[2]. The marker is retrieved in an image of a given camera by segmenting it, and its 6D position is returned in the camera frame thanks to the intrinsic and extrinsic parameters of this camera. Finally this position is transformed into the torso frame of the robot (see Figure 3.1), which is the Cartesian space used for the DMP.

Then, the goal is tracked by *Pepper* with the function *lookAt* of the *NAOqi* framework. When the goal position changes, it is given to the DMP algorithm which updates the attractor point of the dynamic system. It also adds a delay to the trajectory duration, proportional to the movement of the goal, to ensure that the task can be finished in time.

The obstacle detection is described in Section 3.5 and they are represented as a repulsive vector in three dimension in the torso frame of the robot.

Once all these data have been retrieved the algorithm can be processed, following the representation explained in Section 2.2.2.

[2] https://github.com/pal-robotics/aruco_ros

3.3.1 DMP Algorithm

This program is implemented in *C++* and uses the github library *dmpbbo*[3]. It follows the mathematical implementation described in Section 2.2.2, the files containing the translations and rotations are parsed to extract the variables y, $\dot{y}$, z and $\dot{z}$ of equation 2.5. Two DMP are created with these variables, one for the translations and one for the rotations, following the next associations:

- y is filled with the three coordinates of the Cartesian translation (or rotation)
- z takes the three coordinates of the Cartesian velocity
- $\dot{y}$ is equal to z/τ
- $\dot{z}$ is equal to $(-spring\_constant * (y - goal) - damping\_coefficient * z) / \tau$

It is necessary to choose the type of approximation function (and its number of basis functions) that will be used to adjust the dynamical system (see Section 2.2.2 and Appendix B). This function determines the forcing term to be added to $\dot{z}$, in order to fit the demonstrated trajectory. First the basis functions are initialized accordingly to the chosen approximation function (LWR, RBFN, ...). Then, the spring system is created: as a dynamic system, it is composed of two periods, the phasing and the gating one (the created DMP is separated in two depending on the duration). The approximation function is computed on the phasing part, using the Gaussian basis functions activation. It is then applied on the gating part, giving the forcing term which are lately added to $\dot{z}$.

In the Figure 3.2 is presented the work-flow of the DMP algorithm, used in the solution. As aforementioned, this algorithm is periodically called during the execution of the movement, to keep the computed trajectory updated, in function of the goal's and robot's positions. This algorithm is always initialized with the demonstration files rather than the computed trajectories, to avoid the addition of uncertainty and the deformation of the movement.

[3] https://github.com/stulp/dmpbbo

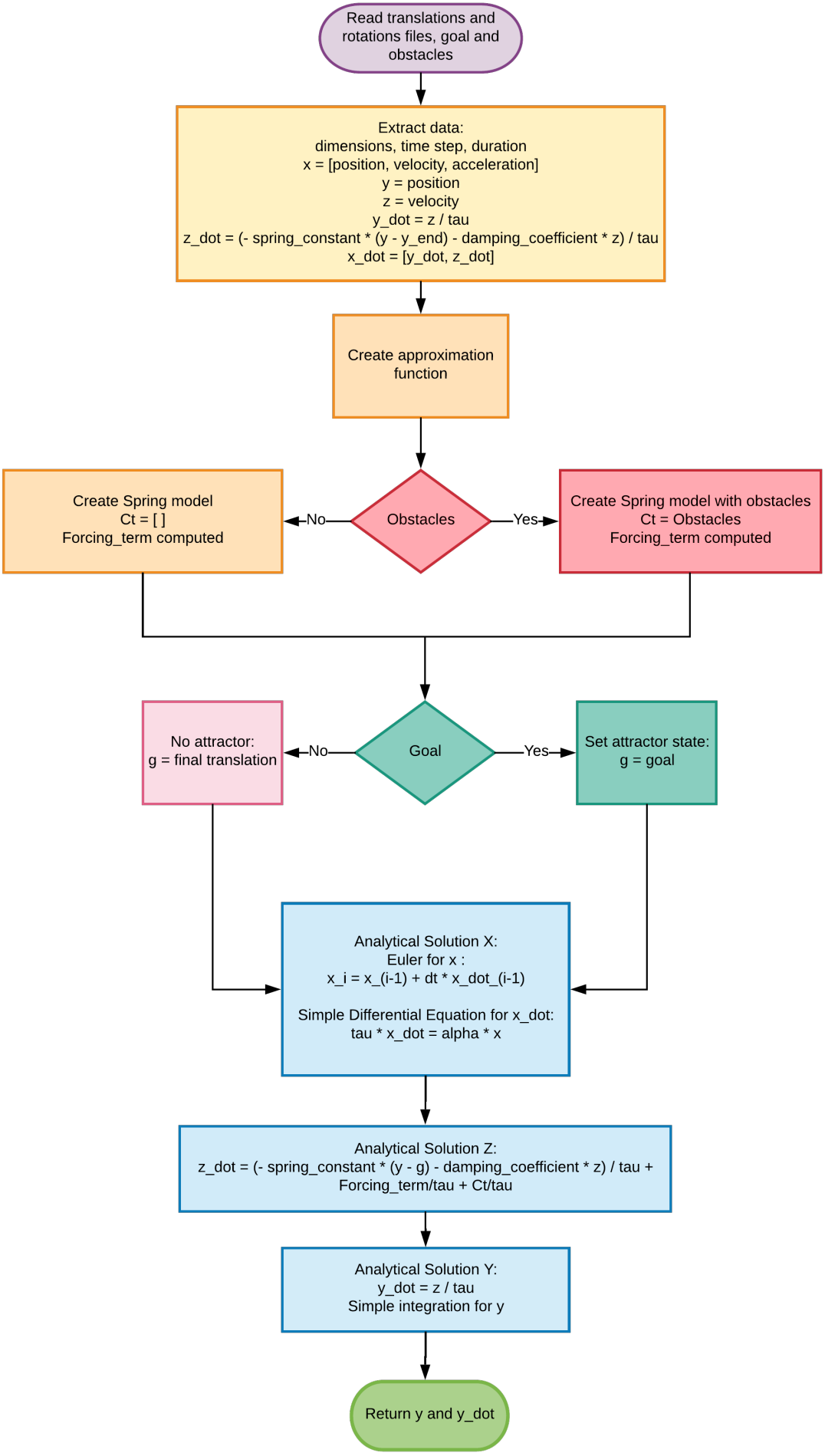

Figure 3.2: Work-flow of the DMP Algorithm.

3.3.2 Obstacles Addition

As explained in Section 2.2.2, obstacles can be added to $\dot{z}$ in the equations (see Equation 2.6 and Figure 3.2) in the C_t term, a vector representing a repulsive force allowing to avoid the obstacles. This vector creates a perpendicular movement to the trajectory direction when an obstacle is detected. The repulsive force depends on the norm of the vector, directly linked to the distance between the obstacles and the robot. The furthest the obstacle is, the smallest the norm will be. This vector is simply added during the analytical solution of Z as shown on the work-flow (Figure 3.2) of the algorithm, divided by the time constant τ.

The perpendicular vector is computed periodically, with a smaller period than the DMP algorithm to be up-to-date before the computation of the trajectory. It is realized in a background scheduler, in the *Python* main program (see the Figure 3.4), using the positions of the detected obstacles (see in Section 3.5). The algorithm creating the vector exactly follows the mathematical Equation 2.7 and performs a summation of all the obstacle vectors to obtain one general repulsive force.

3.4 Optimization

The optimization program has been experimented before implementation of the obstacle avoidance behavior. Thus, a simple algorithm is used for the optimization, based on evolutionary strategies. It is implemented with the github library *dmpbbo*[4].

From the parameters of the computed DMP are generated new parameters which act in the DMP equations to reduce the cost function. This cost function is defined by the user as well as a covariance matrix (representing the exploration capability of the algorithm). The general algorithm can be expressed as follow:

Algorithm 3: Optimization Algorithm

Input: θ, Σ, J
Output: $\emptyset$

1: **while** no convergence **do**
2: Generate λ samples from $\mathcal{N}(\theta\ ;\ \Sigma)$
3: Evaluate λ by J
4: Normalize the obtained costs
5: Keep the μ best samples
6: Update θ with these μ samples
7: Update Σ
8: **end while**

- θ initial parameters of the DMP
- Σ covariance matrix defined by the user

[4] https://github.com/stulp/dmpbbo

- J cost function defined by the user
- $\mathcal{N}(\theta\ ;\ \Sigma)$ is a Gaussian function of mean θ and of covariance matrix Σ

The convergence is evaluated by comparing the computed covariance matrix Σ_i and its predecessor Σ_{i-1}, a convergence threshold and a distance measurement (such as the Euclidean norm) are chosen to realize this convergence evaluation.
To update θ a simple rule is applied: from the costs of the μ selected samples are computed their weights ω_i and the new θ are obtained by performing a weighted averaging on them:

$$\begin{array}{lcl} \omega_i & = & exp(c_i) \\ c_i & : & \text{cost of sample } \mu_i \\ \theta & = & \displaystyle\sum_{i=1}^{N} \omega_i \,.\, \theta_i \end{array} \tag{3.1}$$

The Σ matrix defines the exploration and is simply defined here. The user chooses an exploration rate r and a covariance decay factor d, respectively representing the amount of solution that will be explored at each update and the reduction of this exploration at each update. Thus, the covariance matrix is initialized and updated as follow :

$$\begin{array}{lcl} \Sigma & = & r \times I \\ \Sigma_{new} & = & d^2 \times \Sigma \end{array} \tag{3.2}$$

- r exploration rate
- d covariance decay factor
- I Identity matrix

In the first implementation the cost function J was a simple euclidean distance between the hand of the robot and the goal. In future implementation the function will additionally take into account the distance between the obstacles and the hand during the execution of the movement.

3.5 Obstacle Detection

The robot *Pepper* has RGB and depth cameras which can be used for the obstacle detection. For this reason the implemented approach uses a point cloud and a segmentation of depth images, allowing to extract the objects depending on their distance to the camera. Exploration performed on this topic during the internship has raised solutions to detect obstacles during the execution of a movement, covering the case of moving obstacles. But a limitation quickly appeared, the depth camera cannot detect objects closer than eighty centimeters. During the movement, the robot is too close to the obstacles to detect all of them which is an issue because the aim is precisely to detect them at short distance. A first solution was then to use a version of *Pepper* possessing stereo camera, presenting a closer range of detection.

3.5.1 Stereo Vision Experiments

This process uses two cameras to create the depth image from the disparity between them. To extract the depth information of the images the cameras have to be calibrated accordingly. When an object is projected in both images there is a disparity δ between the two pictures coordinates (x_1, y_1) and (x_2, y_2) due to the different positions of the cameras. Thanks to the calibration, this disparity can be found[1, 9]:

$$\begin{aligned} x_2 &= x_1 + \delta(x_1, y_1) \\ y_2 &= y_1 \\ Z &= \frac{b\,f}{\delta} \\ b &= \begin{cases} 0, & \text{if it is the left camera} \\ \text{Baseline between the cameras}, & \text{if is the right camera} \end{cases} \end{aligned} \tag{3.3}$$

- δ is the disparity between the two pictures
- Z is the object distance
- f is the focal length of the cameras
- b is the baseline, the distance between the two cameras

The Equation 3.3 shows that the depth of a point Z in a scene is inversely proportional to the distance between corresponding image points and their camera centers. This equation allows to compute the depth of all pixels in the image pair. The depth image is generated by detecting the image features which also exist in the real scene and by computing their disparity. Then, by grouping all the disparity values in the left picture and converting it into gray-scale, an image representing the objects depth is obtained[9].

In this project, the disparity image is computed using the *OpenCV* library[5] in *Python*, employing the function $StereoSGBM$. This image is then segmented to retrieve the objects close to the robot and to remove the body parts of the robot that can appear on the image. The segmentation is realized with the *OpenCV* library, using the functions of find-contours[6] and watershed[7].

The creation of the mask for the body parts of the robot uses the inverse process of the point-cloud generation, from coordinates in three dimensions the algorithm finds the corresponding pixel in the depth image. In the project, the arm of *Pepper* is seen in the camera during the execution of the movement; its position is retrieved with the *NAOqi* framework and transformed into the depth camera frame with the *Transform* class of *ROS*. Finally its position in pixel (u, v) is retrieved from its coordinates (x, y, z) with the following formula:

$$\begin{aligned} u &= (f\,.\,x + b)/z + c_x \\ v &= (f\,.\,y)/z + c_y \end{aligned} \tag{3.4}$$

[5] https://opencv.org/
[6] Find-contours function
[7] Watershed function

The obtained mask is added to the segmented depth image and the result is represented as a point-cloud, using the parameters of the camera:

$$\begin{array}{rcl} z & = & (f \,.\, b) \,/\, \delta(u,v) \\ x & = & ((u - c_x) \,.\, z + b) \,/\, f \\ y & = & (v - c_y) \,.\, z \,/\, f \end{array} \tag{3.5}$$

- x, y, z are the coordinate of the pixel (u, v) in the frame of the camera in the point-cloud
- c_x and c_y are the coordinates of the focal center of the left camera

This point-cloud is published and displayed in *RVIZ* and used to create an array of points, representing the obstacles. This array is then given to the main program which creates the repulsive vector avoiding the obstacles. But the point-cloud created is translated on the x-axis and presents some inconsistent points due to the quality of the disparity image. This result is due to the parameters selected to create the disparity image in *OpenCV* and to the calibration of the cameras. Moreover a problem that have not been anticipated appeared, the field of view was too narrow when the robot was close of the obstacles. A large obstacle such as a table was not perceived in its entirety when the robot was in front of it. The robot had to look at its hand during the movement to successfully detect all of the obstacles. But with this implementation the robot could not follow the goal anymore.

For these reasons, it has been decided that the case of moving obstacles will not be considered. With this hypothesis a simpler obstacles detection solution has been chosen.

3.5.2 Implemented Solution

In the adopted solution the obstacles are only detected once, at the beginning of the program. The robot is positioned at a distance of at least half a meter from the goal and the depth image is directly taken from the depth camera of the robot (a XTION camera, not a stereo). This solution was not usable before because of the quality of the obtained image at short distance. Then a point-cloud is created from this image, using the *ROS* package *depth_image_proc*[8]. This point-cloud is reduced, only keeping the point at a distance of 1.5 meters or less from the robot and removing the points representing the goal (by comparing the points positions to the location of the *Aruco* marker).

This final point-cloud is published in *ROS* and displayed in *RVIZ* to have feedbacks. The trajectories demonstrated and computed by the DMP algorithm are also published and displayed (as *ROS path*) and allow to have a clear representation of the impact of the repulsive force on the trajectory. The final point-cloud is sampled to reduce the number of points to be treated, reducing the computational time (almost without any loss of accuracy). Finally, these points are expressed in the torso frame of the robot and transmitted to the main program which creates the repulsive vector avoiding the obstacles.

[8] http://wiki.ros.org/depth_image_proc

3.6 Architecture Schemes

In this part are presented two graphs, the first one presents the *ROS* nodes implementation and the second one the architecture of the program implementation.

In the Figure 3.3, the nodes *create_depth_cloud* and *record_player_manager* compute the depth image of *Pepper* to create the original point-cloud. This one is then used by the *DMP_node* to create the final point-cloud with the selected points. The *service_node* is the node containing the inverse kinematics solver, allowing to translate Cartesian coordinates into angular ones. The *move _base_simple* node is used by the *NAOqi* framework to move toward a particular location. The *look_at* node is the one used to track the *Aruco* marker, it uses the *NAOqi* framework to rotate the head of the robot in the direction of the goal. The *tf* node is the transform node of *ROS* which effectuates transformations between the frames.

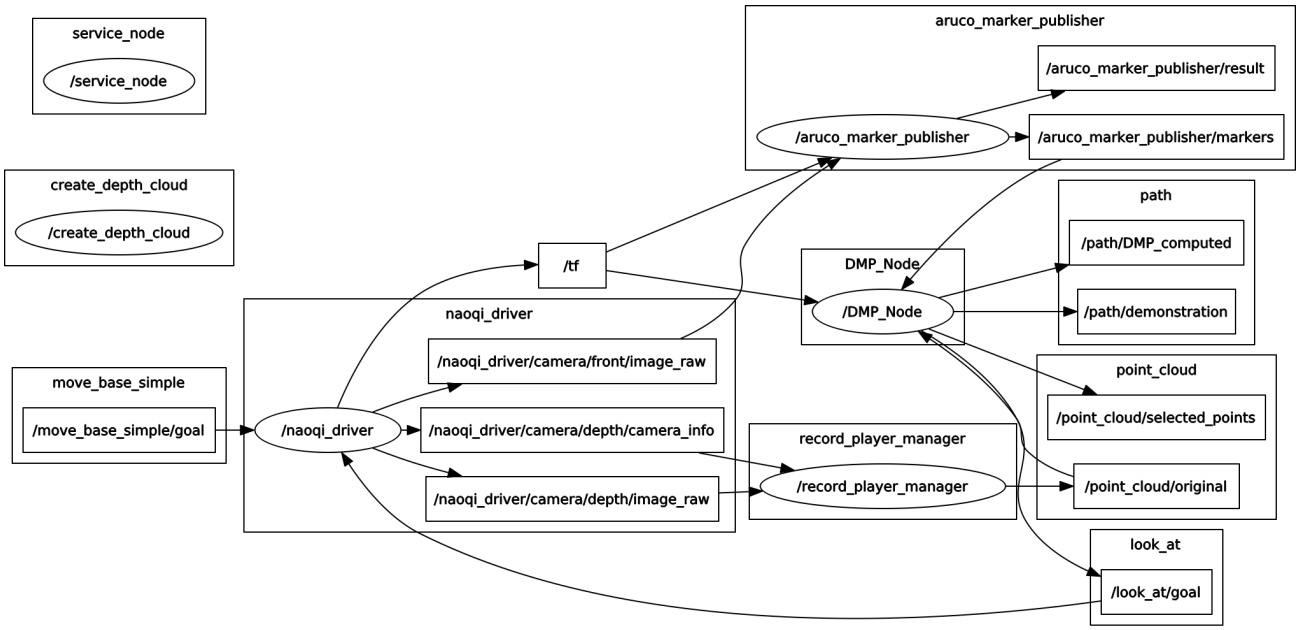

Figure 3.3: ROSGraph: Representation of the links between the *ROS* nodes. The arrows represent topics of the *ROS* architecture.

The Figure 3.4 decomposes the architecture of the implementation between the registering and managing programs (in *Python*), the DMP ones (in *C++*), the *ROS* nodes and the *NAOqi* functions used. The arrows represent the associations and calls between these blocks, grouped by functionalities with colors:

- Blue for the registration of the movement
- Green for the obstacles detection
- Orange for the performance of the movement
- Red for the goal tracking
- Black for the execution of the DMP algorithm
- Purple for the paths creation

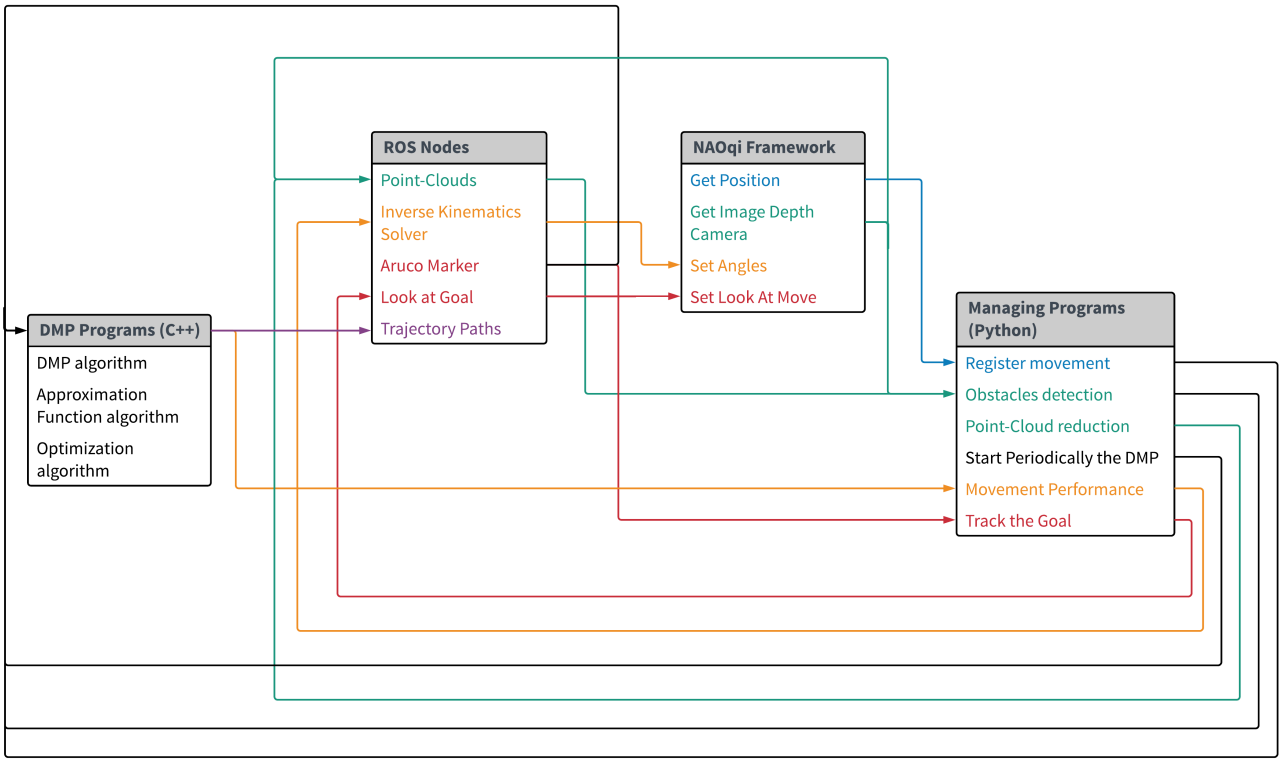

Figure 3.4: Architecture of the implementation.

Chapter 4

Results

In this chapter are presented the different results obtained during this internship. The first part develops the work that have been realized on the obstacles detection, comparing the quality of the obtained point-clouds. The second part renders the outcome obtained with the implemented solution.

4.1 Depth Point-cloud Models

On the Figure 4.1 the scene that have been used for the following tests and results is represented. In this picture the goal, the gray cylindrical shaped object that can be seen in the foreground, has an *Aruco* marker placed on it. The obstacles are the table, the mug and the bottle. In the Figure 4.2 are displayed two depth images, the first is computed directly on the robot by *NAOqi* and is published on a *ROS* topic, and the second one is created with *OpenCV* on an external computer with the stereo image given by the robot.

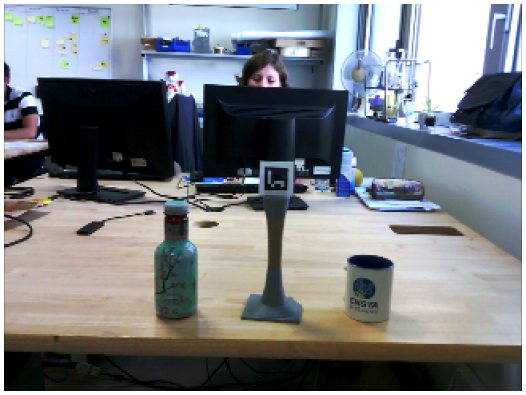

Figure 4.1: Scene used for the obstacles detection.

As expected the result with the stereo leads to a higher rate of object identification, for a scene close to the camera. It is because the image used for the custom reconstruction has a better resolution (1280 × 720px) than the image used by *NAOqi*'s reconstruction algorithm (320 × 240px). Moreover, before the construction of the point-cloud, the stereo depth image is segmented to retrieve only the obstacles (as shown on Figure 4.2). As a consequence,

the stereo point-cloud is more precise and presents only the objects of interest whereas the *NAOqi*'s comuted point-cloud only shows a surface (see Figure 4.2).

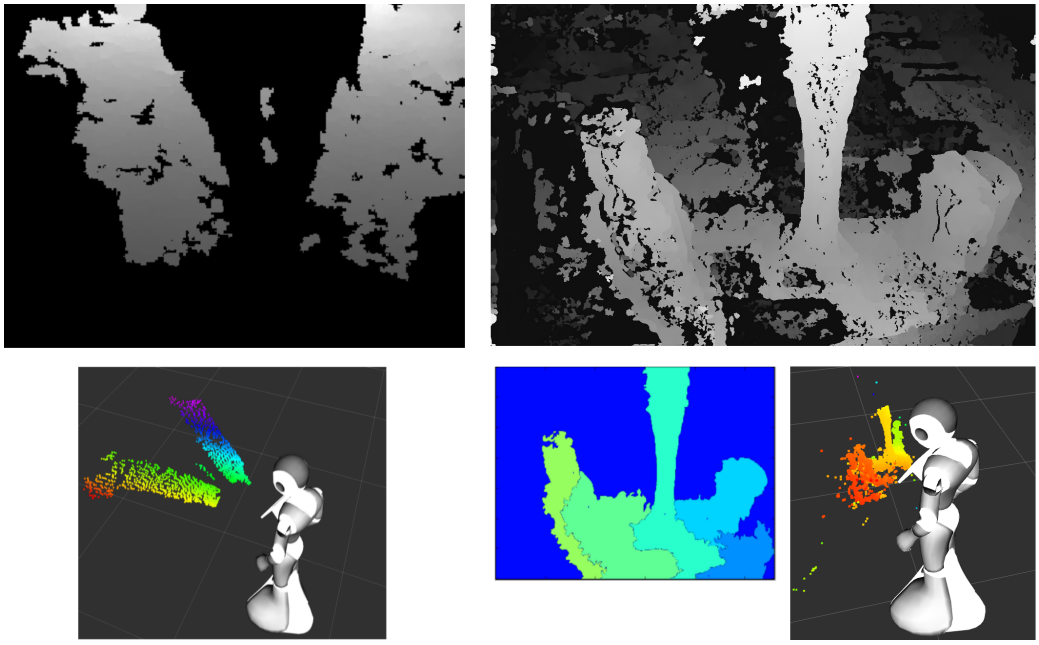

Figure 4.2: On the left: depth image and point-cloud computed with *NAOqi*
On the right: stereo depth image, segmentation and point-cloud computed with *OpenCV*.

In the adopted solution, the robot is at a distance of at least half a meter from the goal. In the Figure 4.3 is displayed the depth image created with *OpenCV*. The result is exploitable and it is possible to extract the objects of the scene as it can be seen on the segmentation picture. But the point-cloud created from the segmentation is translated on the x-axis. Some inconsistent points can be noticed, their presence is due to the quality of the disparity image. As a consequence, when the obstacles repulsive vector is created with these points and is put in the DMP equation, the behavior obtained is inaccurate. The error in translation creates a trajectory which avoid prematurely the obstacles and the inconsistent points alter this trajectory by adding obstacles although they do not exist.

Nevertheless, by using the depth camera of *Pepper* (the XTION camera, not the stereo), the result obtained with the depth image of *NAOqi* (Figure 4.4) is accurate enough to obtain all the obstacles of the scene in a complete point-cloud. The shapes of the objects are clear and are exactly positioned in the frame of the robot. This is due to the preprocessing realized by *NAOqi* on the 3D image returned by the camera, which provides improved image quality compared to the image raw (used in stereo). Finally, when these obstacles points are computed to create the repulsive vector and added to the DMP equation, the trajectory obtained allows the robot to avoid the obstacles.

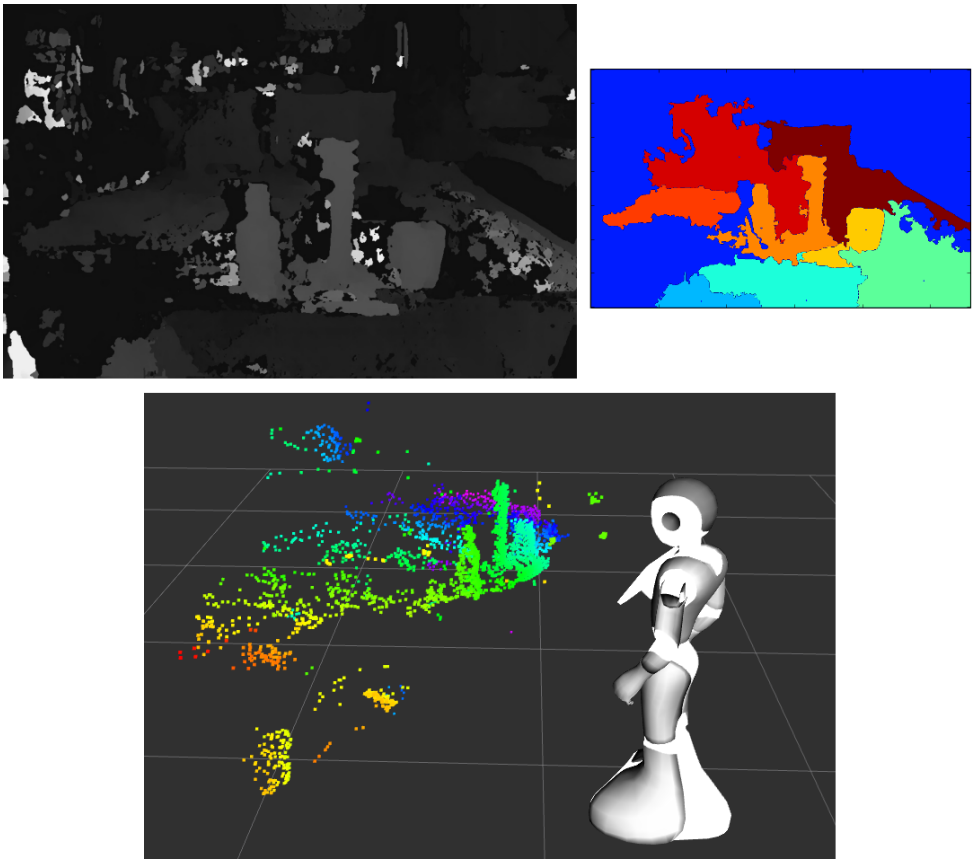

Figure 4.3: Depth image from the stereo camera of *Pepper*, its segmentation and the point-cloud computed with *OpenCV*.

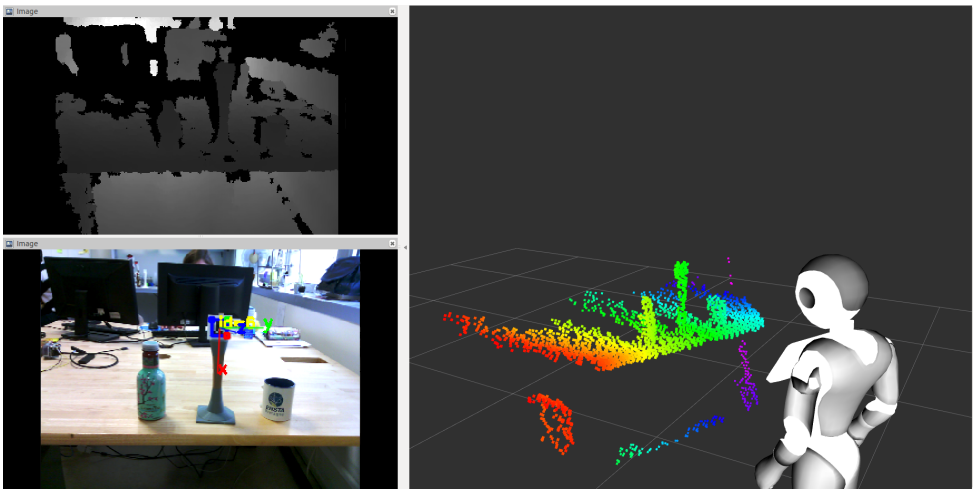

Figure 4.4: Depth image from the depth camera of *Pepper* and point-cloud computed with *NAOqi*.

4.2 Comparison of Approximation Functions in DMP

To adjust the dynamical system of the DMP to the demonstration it is necessary to choose the approximation function and its number of basis functions (see Equations 2.5, 2.8). Three methods have been experimented during this internship, the LWR, the RBFN and the GMR. In the Table 4.1 are presented a comparison of the methods based on the euclidean distance between the obtained trajectories and the demonstration. Then a mean is calculated on the coordinates:

$$\begin{aligned} distance_x &= \sqrt{\sum_{i=0}^{N} (\, x(i)_{demo} - x(i)_{DMP})^2 \,)} \\ distance &= \frac{1}{3} \sum_{j=x,y,z} distance_j \end{aligned} \tag{4.1}$$

- x, y, z the Cartesian coordinates
- N the number of positions in the trajectory

Number of Basis	Position Distance (m)	Velocity Distance (m/s)
	LWR	
20	0.059	0.471
30	0.029	0.436
40	0.026	0.422
50	0.022	0.404
60	0.024	0.398
	RBNF	
20	0.042	0.460
30	0.047	0.445
40	0.044	0.427
50	0.033	0.420
60	0.085	0.405
	GMR	
20	0.033	0.431
30	0.035	0.409
40	0.042	0.398
50	0.074	0.406
60	0.048	0.372

Table 4.1: Comparison of the approximation functions based on the euclidean distance between the demonstration and the computed trajectory.

In the Table 4.1 are presented the mean results obtained on ten demonstrations, on the right and left arm of *Pepper*. The fidelity of the trajectory is quite accurate, for each methods the difference in position is less than ten centimeters (in 3D). The result must have been better without the slack in the robot's arms leading to imprecision in the kinematic solver (which takes here 5 joints in the kinematic chain). The best method according to these experiments is the LWR method (see Annexe B) with a number of basis between 30 and 60. It creates a smooth trajectory fitting the demonstration with the best accuracy (see Figure 4.5). Thus, in the adopted implementation, the DMP algorithm is employed with the LWR approximation function with a number of basis of 40. Indeed, the more number of basis functions are used, the more the computational time increases because it adds calculation, thus 40 is a good compromise to keep a great accuracy and a small computational time (around 0.1 seconds).

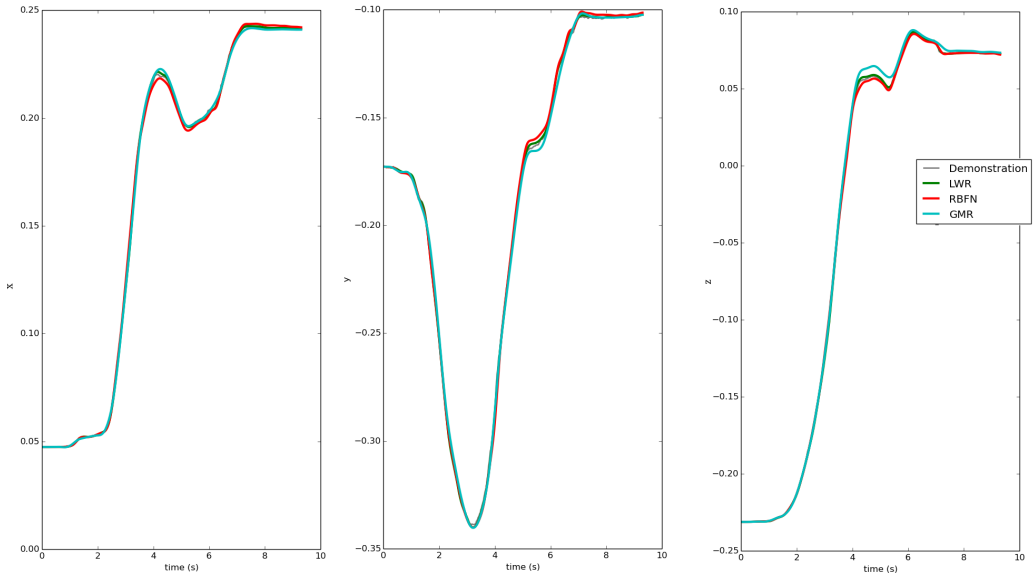

Figure 4.5: Comparison of the LWR, the RBFN and the GMR approximation functions on an arm demonstration.

4.3 DMP Trajectories

4.3.1 Optimization Without Obstacles

In this section are presented the results obtained when the Algorithm 3 is used to update the parameters of the DMP. The obstacles are not taken into account in this implementation. The demonstration is a movement where the arm of *Pepper* is raised and directed to avoid a table (where the goal is put), and it finishes with the hand of *Pepper* in front of its torso. The chosen cost function expresses the distance between the hand of the robot and the goal, at the end of the movement.

In the Figure 4.6 is presented the impact of the optimization algorithm on the computed trajectory, the hand of *Pepper* is approaching the goal (the *Aruco* marker) to reduce the cost function of the algorithm. After seven updates of the parameters, each updates realizes ten

samples (thus the movement is realized 70 times), the final cost is around 3.5cm (in 3D), meaning it has almost been divided by three in comparison to the first computed trajectory (see Figure 4.7).

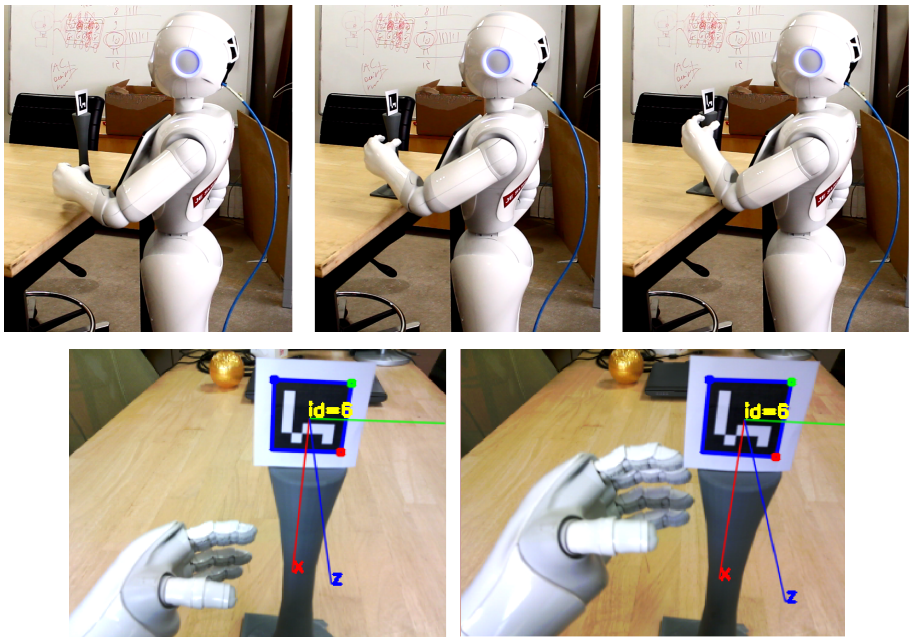

Figure 4.6: Evolution of the hand position of *Pepper* during optimization.
On the left the hand position before optimization, on the right the hand position after optimization

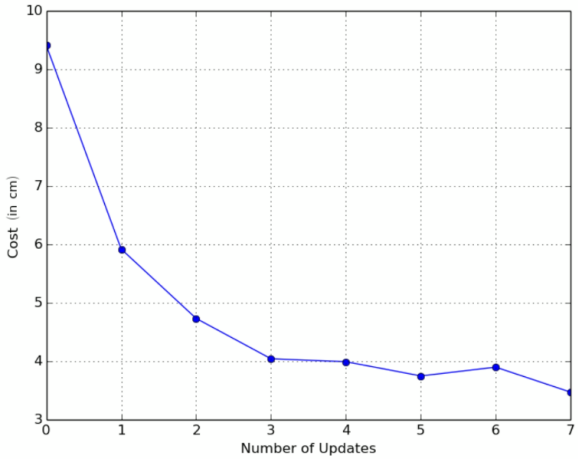

Figure 4.7: Evolution of the cost function during optimization.

It is interesting to point out the fact that the optimization needs the trajectories to be as close as possible to each others to be the most efficient. Thus, because the starting point can be different and the movement can change a little because of the DMP, the cost function will not reach zero. But in this project a perfect optimization is not necessary although it may be achieved with a different cost function.

4.3.2 Implementation With Obstacles

In this section is presented the result of the DMP algorithm with the obstacles avoidance behavior. In the following example, the demonstration trajectory is a circular arm movement which ends with the hand of *Pepper* in front of its torso. At the end of the demonstration the wrist of *Pepper* is turned to face its torso. It has been realized without any obstacle.

In the Figure 4.8, the demonstrated and computed trajectories are represented as *ROS path* in *RVIZ*. The black line is the demonstration and the white is the computed one. The wrist of *Pepper* follows this second line which is updated during the execution of the movement. In this figure can be seen the difference between the two trajectories: the computed one avoids the obstacles (in particular, the table) represented in the point-cloud whereas the demonstrated one pass into it. The last part of the movement is the rotation of the wrist which is represented as a small circular circle in both of the trajectories. It is noticeable that the goal has been removed of the point-cloud to avoid the algorithm to create a trajectory going back and forth.

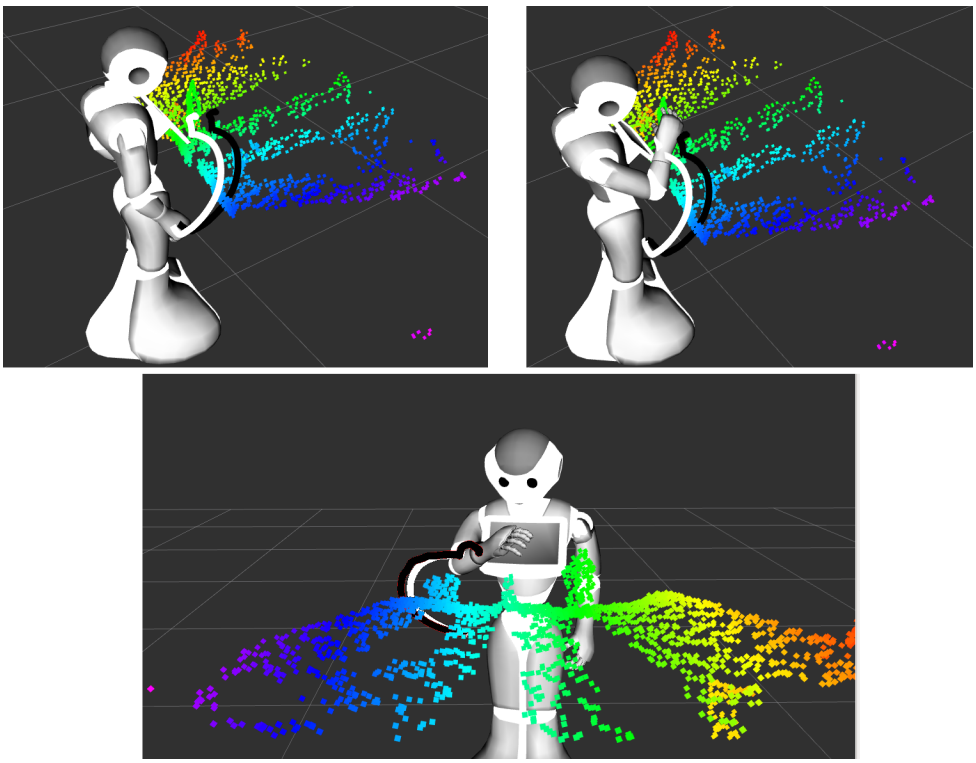

Figure 4.8: Representation of the trajectories in *RVIZ*, the demonstration is in black and the DMP result in white.

The figure 4.9 presents sequenced pictures of the above computed trajectory performed by *Pepper*. Compared to the initial demonstration (see the black line in Figure 4.8), the arm moves differently and adapt its course to avoid the table. Then, keeping its distance from the table it avoids the mug. This example validates the implementation on the *Pepper* robot of DMP creating trajectories with obstacles avoidance.

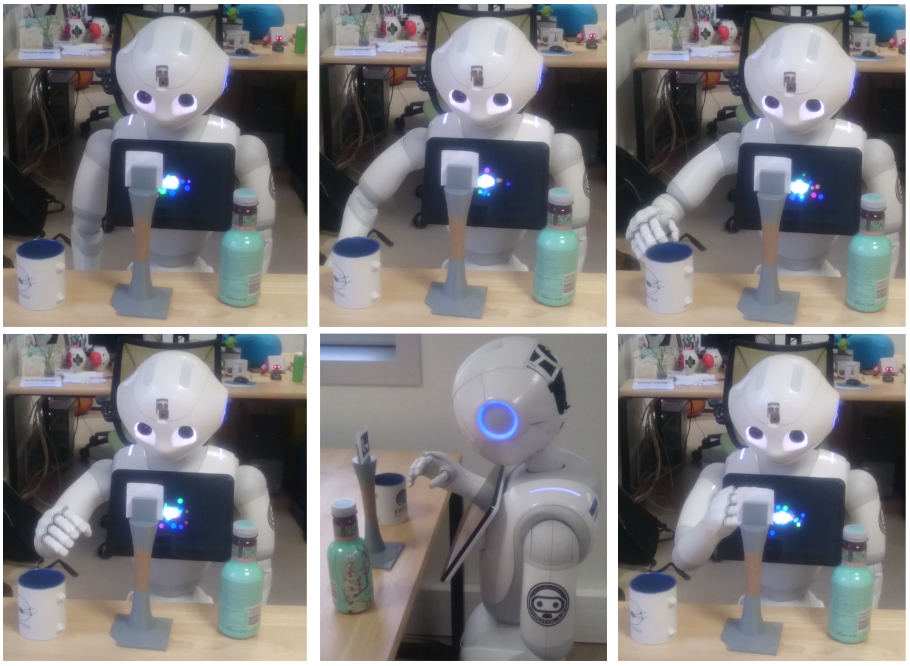

Figure 4.9: Realization of the trajectory computed by the DMP.

4.3.3 Implementation With Goal Movements

The last result concerns the adaptivity of the DMP to the goal's motions, which is an important point to validate in the implementation. The DMP computed trajectory has to take into account the obstacles and the movements of the goal, to reach the internship expectations. In the implementation, each time the goal moves more than 0.3 cm, its position is updated in the DMP algorithm and this one updates the attractor state of the dynamical system. With this implementation the computed trajectory has always the goal as final position. In the Figure 4.10 is presented the *ROS path* created during an example where the goal is displaced upward.

The obtained result shows that the computed trajectory takes into account this change, the blue line effectuates a higher circle compared to Figure 4.8 although the demonstrations are the same. Moreover the trajectory still avoid the obstacles as it is shown with the table, the blue line goes back and higher than the black just as the previous example.

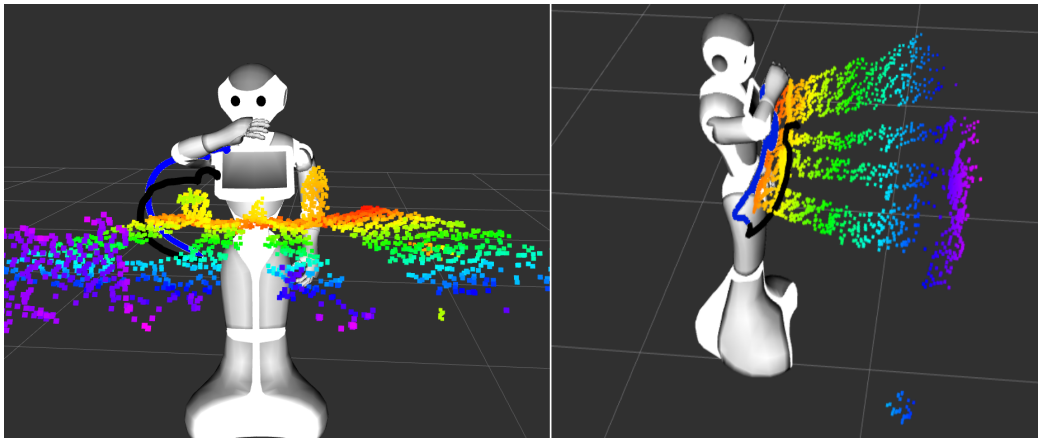

Figure 4.10: *RVIZ* representation of the adapted trajectory in function of the goal's moves (goal displaced upward). The demonstration is in black and the DMP result in blue.

The figure 4.11 presents sequenced pictures of the computed trajectory during the displacement of the goal. The arm of *Pepper* is moving upward to reach the goal, while avoiding the obstacles. This example achieves the validation of the implementation on the *Pepper* robot, using the DMP to create trajectories from a demonstration in an adaptive and dynamic way.

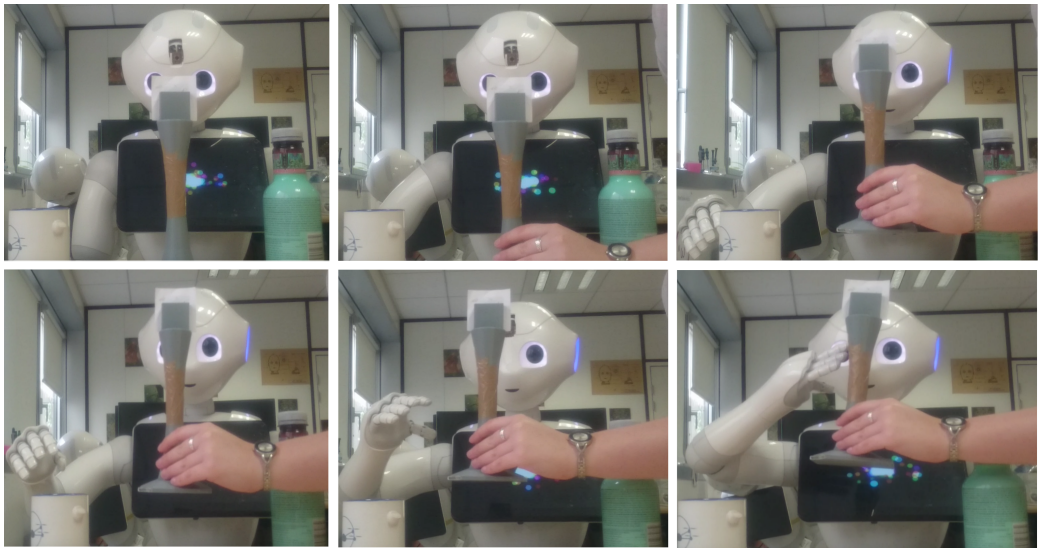

Figure 4.11: Realization of the DMP's trajectory adapted in function of the goal's movements (here goal displaced upward).

Conclusion and Future Work

The solution developed during this internship enables to teach to *Pepper* new movements and tasks without programming them. The user effectuates the demonstration by moving the body parts of *Pepper* and the task is recorded. At the beginning of the reproduction of the movement, the robot acquires the position of the obstacles in the scene. The DMP algorithm finally computes the trajectory tacking into account the obstacles and the goal. This implementation allows *Pepper* to move its hand to reach a graspable object, while avoiding obstacles. The solution is flexible and allows a great adaptability to changing situations because of the characteristics of the differential equations.

As a future work, the optimization process can be improved to take into account the obstacles. The cost function in the RL algorithm can be computed after the DMP update of the trajectories and use the distance from the hand to the goal and from the hand to the obstacles (using a Weighted Average).
Moreover the *ROS paths* will completely replace the file used to register the computed trajectory. Indeed writing and reading data in files can be time consuming and create conflicts. This choice was made to examine the computed trajectory after the reproduction of the movement. Finally feedbacks algorithms, such as speech or physical controls, may be added to allow the cooperation between Humans and robots.

Allowing the communication with the robot can provide a way for the tutor to rate the reproduction of the task by the robot[3]. This is really useful in the scope of classifying the computed movements and reaching the solution quicker with fewer demonstrations. Moreover the robot may interact directly with the demonstrator by asking precisions on the movement, to prioritize specific parameters of the motion. To make this interaction possible, this requires some natural language processing (NLP) researches to make sure the robot understand well the answers of the tutor and to implement questions directly linked to the desired feedbacks.

If physical interactions are allowed, the demonstrator will not only perform the movement to be learned by the robot but also interact with it during its attempts to accomplish the task. The actions of the demonstrator are taken into account in the equations of the trajectory to adapt the movement to these interventions. This method allows to directly correct the gesture and position of the robot, allowing the improvement of its trajectory (in particular to adapt it while avoiding obstacles).

List of Figures

List of Tables

List of Algorithms

Appendices

Appendix A

Expectation Maximization Algorithm

EM is a simple local search technique used when the data is incomplete, has missing data points, or has unobserved (hidden) latent variables. It guarantees a monotone increase of the training set's likelihood during optimization[18]. It chooses random values for the missing data points, and using those guesses, it estimates (E) a second set of data. The new values are used to create a better guess (M) for the first set, and the process continues until the algorithm converges on a fixed point. To avoid being trapped in a local minima when choosing the random values, a k-means clustering technique can be applied on the dataset .

From the probability density function of the GMM expressed as:

$$\begin{aligned} v &= [t\ \dot{\xi}^T]^T \\ p(v) &= \sum_{k=1}^{K} \pi_k\, N(v;\mu_k;\Sigma_k) \\ N(v;\mu_k;\Sigma_k) &= \frac{1}{\sqrt{(2\pi)^D\,|\Sigma_k|}}\, exp(-\frac{1}{2}(v-\mu_k)^T\Sigma_k^{-1}(v-\mu_k)) \\ \mu_k &= [\mu_{k,t}^T\ \mu_{k,\dot{\xi}}^T]^T \\ \Sigma_k &= \begin{bmatrix} \Sigma_{k,t} & \Sigma_{k,t\dot{\xi}} \\ \Sigma_{k,\dot{\xi}t} & \Sigma_{k,\dot{\xi}} \end{bmatrix} \end{aligned} \tag{A.1}$$

The probability density function can be reformulated as conditional, depending on $\theta = (\pi_k, \mu_k, \Sigma_k)$[18]:

$$p(v|\theta) = \sum_{k=1}^{K} \pi_k\, p_k(v \mid \mu_k, \Sigma_k) \tag{A.2}$$

Where p_k is defined as the Gaussian distribution in equation A.1. The log of Likelihood function is then expressed as:

$$L(V|\Theta) = \sum_{k=1}^{K}\sum_{i=1}^{N-1} h_{k,i}\, ln(\pi_k\, p_k(v_i \mid \mu_k, \Sigma_k)) \tag{A.3}$$

- N is the sample size

- $h_{k,i} = p(k \mid v_i)$ is the conditional expectation of p_k given the observation v_i. So the posterior probability that v_i belongs to the kth component.

Expectation step computes the conditional expectation probability $h_{k,i}$ and then $L(V|\Theta)$ with $\Theta^{(n)}$:

$$h_{k,i}^{(n)} = \frac{\pi_k \, p(v_i \mid \mu_k^{(n)}, \Sigma_k^{(n)})}{\sum_{r=1}^{K} \pi_r \, p(v_i \mid \mu_r^{(n)}, \Sigma_r^{(n)})} \tag{A.4}$$

Maximization step computes the maximization of the log-likelihood function given $h_{k,i}^{(n)}$ and $\Theta^{(n)}$. Thus, the estimation of the mixture Gaussians parameters $(\pi_k^{(n+1)}, \mu_k^{(n+1)}, \Sigma_k^{(n+1)})$ is possible:

$$\begin{aligned}
\hat{\Theta}^{(n+1)} &= arg\max_{\theta} L(V|\Theta^{(n)}) \\
\hat{\Theta}^{(n+1)} &= (\pi_k^{(n+1)}, \mu_k^{(n+1)}, \Sigma_k^{(n+1)}) \\
\hat{\pi_k}^{(n+1)} &= \frac{1}{N} \sum_{i=1}^{N} h_{k,i}^{(n)} \\
\hat{\mu_k}^{(n+1)} &= \frac{\sum_{i=1}^{N} h_{k,i}^{(n)} v_i}{\sum_{i=1}^{N} h_{k,i}^{(n)}} \\
\hat{\Sigma_k^2}^{(n+1)} &= \frac{\sum_{i=1}^{N} h_{k,i}^{(n)} (v_k - \hat{\mu_k}^{(n)})^2}{\sum_{i=1}^{N} h_{k,i}^{(n)}}
\end{aligned} \tag{A.5}$$

Appendix B

Approximation Functions

LWR is a memory-based method that performs a regression around a point of interest using only training data that are "local" to that point. Thus, points are weighted by proximity to the current x using a Gaussian kernel (see Figure B.1). A regression is finally computed using the weighted points, it minimizes the locally weighted quadratic error criterion[11]:

$$\sum_{t=1}^{T} \psi_i(t) \left(f_{target}(t) - \omega_i(x(t)\,(g - y_0))\right)^2 \tag{B.1}$$

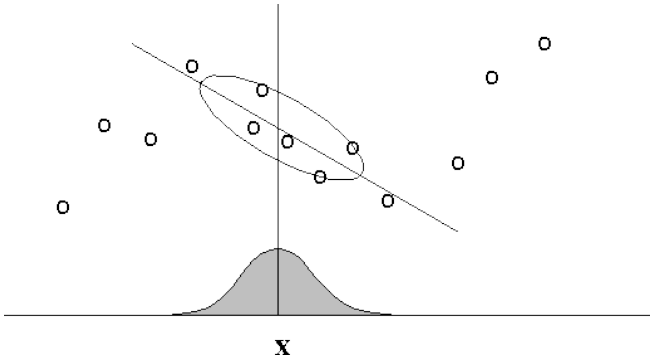

Figure B.1: Locally Weighted Regression.

LWPR is a nonparametric regression technique. It determines the number of basis function ψ_i, their center c_i and their width σ_i automatically. It uses the LWR method, but keeps each computed model for further predictions in memory. It uses multiple locally weighted linear models which are combined for approximating non-linear function. These updates allow to chose the best number of basis function ψ_i.

RBFN is an artificial neural network that uses radial basis functions (RBF) as activation functions. It possesses three layers: an input layer, a hidden layer with a non-linear RBF activation function and a linear output layer. The RBF can be expressed as follow:

$$y(x) = \sum_{i=1}^{N} \omega_i \, \phi(\|x - c_i\|) \tag{B.2}$$

- N the number of radial basis function ψ_i associated with a center c_i and a weight ω_i
- ϕ the radial function, usually a Gaussian
- ||.|| the norm, usually the Euclidean distance

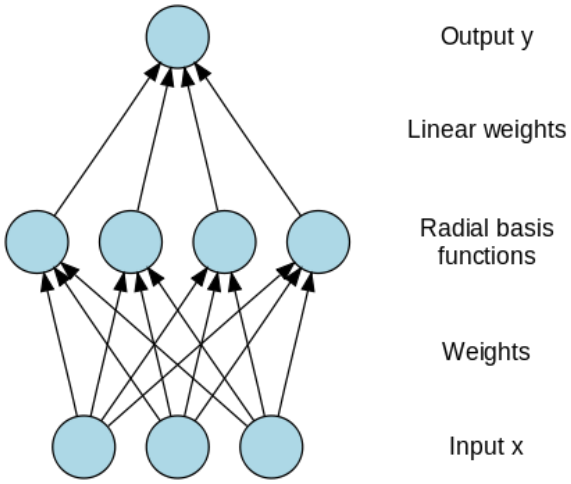

Figure B.2: Radial Basis Function Network.

This figure represents the architecture of a RBFN. An input vector Z (see the equation B.3) is used as input to all radial basis functions, each with different parameters. The output of the network is a linear combination of the outputs from radial basis functions. The weights ω_i of the matrix W are estimated using one of the linear least squares methods, for instance Weighted Least Squares (WLS) which solves the following quadratic minimization problem (with Y the position matrix):

$$
\begin{aligned}
\hat{W} &= arg\min_{\omega} S(\omega) \\
S(\omega) &= \sum_{j=1}^{M} |y_j - \sum_{i=1}^{N} \omega_i Z_{ij}|^2 = \|Y - WZ\|^2 \\
Y &= [y_1\, y_2 \ldots y_M]^T \\
W &= [\omega_1\, \omega_2 \ldots \omega_N]^T \\
Z &= \begin{bmatrix}
\phi(\|X_{11} - C_1\|) & \phi(\|X_{12} - C_2\|) & \ldots & \phi(\|X_{1M} - C_M\|) \\
\phi(\|X_{21} - C_1\|) & \phi(\|X_{22} - C_2\|) & \ldots & \phi(\|X_{2M} - C_M\|) \\
\ldots & \ldots & \ldots & \ldots \\
\phi(\|X_{N1} - C_1\|) & \phi(\|X_{N2} - C_2\|) & \ldots & \phi(\|X_{NM} - C_M\|)
\end{bmatrix}_{(N,M)}
\end{aligned}
\tag{B.3}
$$

This minimization problem has a unique solution, because the M columns of the matrix Z are linearly independent, given by solving the following normal equation. Finally $\hat{W}$ is the coefficient vector of the least-squares hyperplane, expressed as followed:

$$\begin{array}{lcl} (Z^T Z)\hat{W} & = & Z^T Y \\ \hat{W} & = & (Z^T Z)^{-1} Z^T T \end{array} \tag{B.4}$$

For these three methods, the learning of the forcing terms weights is computed independently for each degree of freedom. Thus, the learning of the model is quite quick.

Appendix C

Combination of Methods

C.1 DS-GMR

This model combines the advantages of the GMM and of the DMP[5]. The DMP is modulated as a probabilistic model thanks to the usage of GMR during the learning. In this way, DMP systems can easily be extended to task-parameterized models such as Parametric Hidden Markov Models (PHMM), which allows the modulation of movements with respect to task parameters such as positions of objects.

The movement is represented with the DMP as a superposition of virtual spring-damper systems. These ones are described statistically such that the attractor points acts in different reference frames. The GMR is used to determines which frames are the most important in the given trajectory. Thus the spring-damper systems are activated in function of the movement.

A noticeable advantage of DS-GMR over the WLS or LWR approaches, described in Appendix B, is that DS-GMR automatically adapts the span and position of the activation weights while learning the movement. In DS-GMR, the system learns how to partition the activation weights ω_i (see equation 2.8) together with the search of force components. It should allow smoother transitions between the force components even in movements of varying complexity.

Moreover this method does not need to modify the DMP representation, it only replaces the imitation learning mechanism by a GMR. On the following figure a comparison between WLS and GMR is presented.

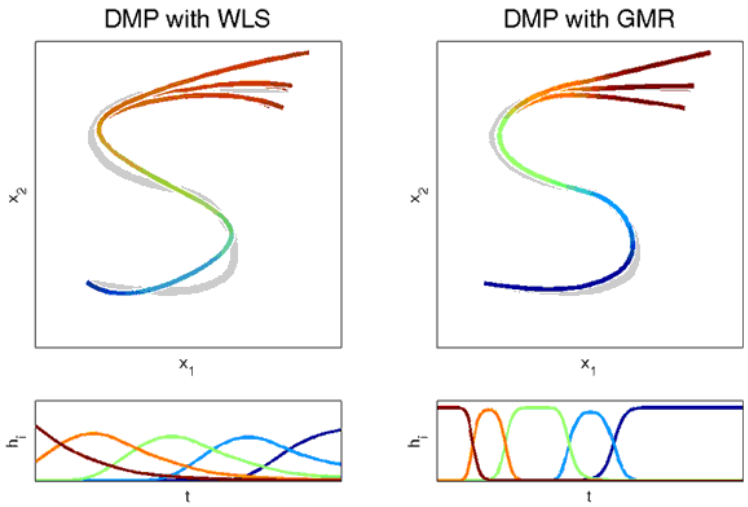

Figure C.1: DS-GMR VS DMP with WLS[5].

The first line of blocks represents the result of the DMPs after WLS (on the right) and GMR (on the right) learning process. The demonstration is in gray and the result in color, there are three starting points in each case and one goal. The representation of the movement is the representation of one joint in two-dimensions (x_1 and x_2). The movement has five spring-damper systems (represented by the different colors) and their activation in function of the time are represented in the second line of blocks (function h_i). The GMR produces activation functions that are more locally defined (Gaussian functions) than the WLS ones. That is why the DS-GMR provides a more accurate approximation of the non-linear perturbing force.

C.2 GMM for modulating DMP

This method, as opposed to the previous one, to the one before, first computes the GMM and then uses it to modulate a dynamical system expressed with the DMP. In the example described in *Learning Dynamical System Modulation for Constrained Reaching Tasks*[10], the dynamical system has an attractor point considered as a target given by the stereo-vision tracking of the robot. Their system can be represented as followed:

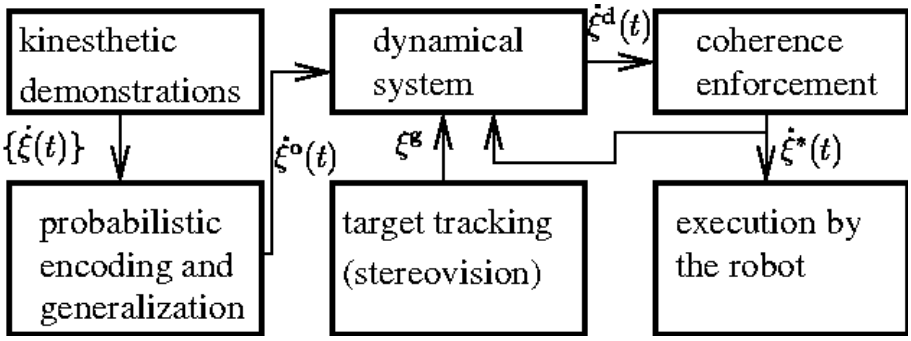

Figure C.2: Conceptual sketch of the GMM for modulating DMP system[10].

The first two blocs are created like the two first blocks described in the part of the GMM (see Section 2.2.1), the dynamical system is then expressed similarly to the DMP one (see Section 2.2.2, Equation 2.4), with Gaussian expressions:

$$\ddot{\xi}^s(t) = \alpha\left(-\dot{\xi}^s(t) + \beta\left(\xi^g - \xi^s(t)\right)\right) \tag{C.1}$$

This system is modulated by the generalized trajectory $\dot{\xi}^o(t)$ created by the GMM and EM algorithms. A weighted average between the velocities of the demonstrations, $\dot{\xi}^o(t)$, and of the dynamical system, $\dot{\xi}^s(t)$, is then computed. Finally this solution is corrected to achieve consistent end-effector positions and joint angle configuration (coherence enforcement bloc in figure C.2) to give the final result $\dot{\xi}^*(t)$.

Likely to the DS-GMR method, this representation combines the advantages of the GMM and DMP methods. The DMP allows variability in the initial and target states, such as their displacements. The GMM gives to the system its generalization and scalability which gives it the ability to perform many different tasks.

Appendix D

Reinforcement Learning Algorithms

D.1 Natural Actor-Critic (NAC)

This method is an improved actor-critic method, it is based on two stages: the actor improvement, which improves the current policy, and the critic evaluation, which evaluates the current policy [20, 21, 8].

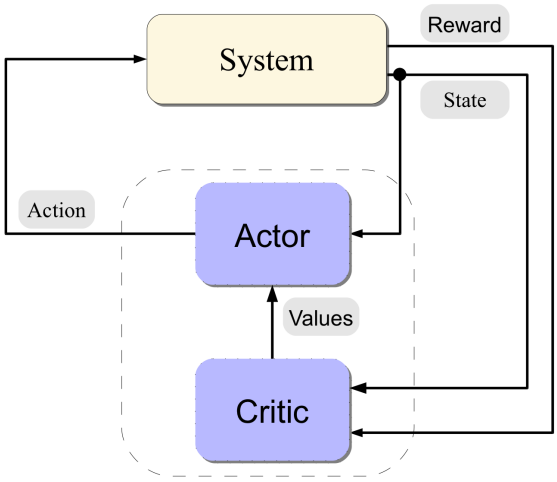

Figure D.1: The Actor-Critic principle.

The critic part is evaluated by approximating the state-value function $Q^{\pi}(s,a)$ with a Least Square Temporal Difference algorithm. The actor part in charge of the policy improvement uses policy natural gradient descent to evaluate the state-value function $V^{\pi}(s)$. The natural gradient can be expressed as followed:

$$\tilde{\nabla}_{\theta} J(\theta) = G^{-1}(\theta) \nabla_{\theta} J(\theta) \tag{D.1}$$

The Fisher information $G(\theta)$ is a way of measuring the amount of information that an observable random variable X carries about an unknown parameter θ upon which the prob-

ability of X depends. Here X can be assimilated to π.

$$G(\theta) = E[(\frac{\partial}{\partial\theta}\log\pi(a|s;\theta))^2 \mid \theta] \tag{D.2}$$

This method updates the policy parameterization according to the gradient update rule, based on a learning rate α:

$$\theta_{t+1} = \theta_t + \alpha_t \nabla_\theta J|_{\theta=\theta_t} \tag{D.3}$$

One major drawback of gradient-based approaches is the learning rate, which is an open parameter that can be hard to tune. This parameter is essential to obtain good performance. The EM algorithm described in Section 2.2.1 can be used to resolve this problem.

D.2 Policy Learning by Weighting Exploration with the Returns (PoWER)

This method uses the EM algorithm to resolve the problem of learning rate. Similarly to the previously described method, it uses parameterized policy to find values for the parameters maximizing the expected return of the policy. This method is adapted to the DMP method because it can take the basis functions ψ_i into account in the policy ϕ[17, 19, 14, 15]:

$$\pi = \theta^T \psi(s,t) \tag{D.4}$$

This equation is completed with an additive exploration $\epsilon(s,t)$, usually a Gaussian, in order to make model-free reinforcement learning possible. As a result the explorative policy is:

$$a = \theta^T \psi(s,t) + \epsilon(\psi(s,t)) \tag{D.5}$$

This method performs many roll-outs $h = [1,...,H]$ to create samples in order to computes the state-action value $Q^\pi(s,a)$. Then it updates the policy parameterization with:

$$\theta^n_{k+1} = ((\psi^n)^T Q^\pi \psi^n)^{-1} (\psi^n)^T Q^\pi A^n \tag{D.6}$$

With $A^n = [a_1^{1,n},...,a_T^{1,n},...,a_1^{H,n},...,a_T^{H,n}]$ the actions computed during roll-outs.

The drawback of this method is that the exploration ϵ is unstructured and lead to perturbations on the actions (acts as low pass filter) and causes a large variance in parameters updates. To solve this problem the exploration function must be changed to become state-dependent with $\epsilon(\psi(s,t)) = \epsilon_t^T \psi(s,t)$ and ϵ_t Gaussian for instance.

Glossary

CP Control Policy. 15

DMP Dynamic Movement Primitives. 1, 2, 5, 15–17, 19, 21, 23–25, 27, 30, 32, 34, 36–43, 53–55, 57

EM Expectation Maximization. 12, 14, 55, 57

GMM Gaussian Mixture Model. 12–14, 16, 48, 53–55

GMR Gaussian Mixture Regression. 12, 14, 36, 37, 43, 53, 54

IoT Internet of Things. 9

LfD Learning from human Demonstration. 1, 2, 5, 6

LWPR Locally Weighted Projection Regression. 19, 50

LWR Locally Weighted Regression. 19, 25, 36, 37, 43, 50, 53

MDP Markov Decision Process. 20

ML Machine Learning. 1, 2

PbD Programming by Demonstration. 5

PHMM Parametric Hidden Markov Models. 53

RBFN Radial Basis Function Network. 19, 25, 36, 37, 43, 50, 51

RL Reinforcement Learning. 20, 42

ROS Robot Operating System. 23, 24, 29–32, 43

SMCT Sensorimotor Contingencies Theory. 8

WLS Weighted Least Squares. 51, 53, 54

Bibliography

[1] Franz Andert and Gordon Strickert. Depth image processing for obstacle avoidance of an autonomous VTOL UAV. page 8, 2006.

[2] Rüdiger Dillmann Aude G. Billard, Sylvain Calinon. Learning from humans. In Bruno Siciliano and Oussama Khatib, editors, *Springer Handbook of Robotics*, chapter 74. Springer-Verlag, Berlin Heidelberg, 2008.

[3] Maya Cakmak and Andrea L. Thomaz. Designing robot learners that ask good questions. page 17. ACM Press, 2012.

[4] Sylvain Calinon, Florent Guenter, Aude Billard, and LASA Laboratory. On Learning, Representing and Generalizing a Task in a Humanoid Robot. *PART B*, page 12.

[5] Sylvain Calinon, Zhibin Li, Tohid Alizadeh, Nikos G. Tsagarakis, and Darwin G. Caldwell. Statistical dynamical systems for skills acquisition in humanoids. pages 323–329. IEEE, November 2012.

[6] J. Degenaar and J. Kevin O'Regan. Sensorimotor theory of consciousness. *Scholarpedia*, 10(5):4952, 2015. revision #149109.

[7] Sergio Garrido-Jurado, Rafael Muñoz-Salinas, Francisco José Madrid-Cuevas, and Manuel Jesús Marín-Jiménez. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, 47(6):2280–2292, 2014.

[8] Florent Guenter, Micha Hersch, Sylvain Calinon, and Aude Billard. Reinforcement Learning for Imitating Constrained Reaching Movements. page 23.

[9] Akkas Uddin Haque and Ashkan Nejadpak. Obstacle Avoidance Using Stereo Camera. *arXiv:1705.04114 [cs]*, May 2017. arXiv: 1705.04114.

[10] Micha Hersch, Florent Guenter, Sylvain Calinon, and Aude Billard. Learning Dynamical System Modulation for Constrained Reaching Tasks. pages 444–449. IEEE, December 2006.

[11] Auke Jan Ijspeert, Jun Nakanishi, Heiko Hoffmann, Peter Pastor, and Stefan Schaal. Dynamical Movement Primitives: Learning Attractor Models for Motor Behaviors. *Neural Computation*, 25(2):328–373, February 2013.

[12] Auke Jan Ijspeert, Jun Nakanishi, and Stefan Schaal. Learning Attractor Landscapes for Learning Motor Primitives. page 8.

[13] Auke Jan Ijspeert, Jun Nakanishi, and Stefan Schaal. Movement imitation with nonlinear dynamical systems in humanoid robots. In *In IEEE International Conference on Robotics and Automation (ICRA2002*, pages 1398–1403, 2002.

[14] J. Kober and J. Peters. Learning motor primitives for robotics. In *2009 IEEE International Conference on Robotics and Automation*, pages 2112–2118, May 2009.

[15] Jens Kober. Reinforcement Learning for Motor Primitives. page 89.

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

[17] Petar Kormushev, Sylvain Calinon, and Darwin G Caldwell. Robot motor skill coordination with EM-based Reinforcement Learning. pages 3232–3237. IEEE, October 2010.

[18] Ould Mohamed Mahmoud Mohamed and Jaidane Meriem. On the parameters Estimation of The Generalized Gaussian Mixture Model. page 5.

[19] Jan Peters and Stefan Schaal. Policy Gradient Methods for Robotics. pages 2219–2225. IEEE, October 2006.

[20] Jan Peters and Stefan Schaal. Natural Actor-Critic. *Neurocomputing*, 71(7):1180–1190, March 2008.

[21] Jan Peters, Sethu Vijayakumar, and Stefan Schaal. Reinforcement Learning for Humanoid Robotics. page 20.

[22] Peter Pastor Sampedro. Imitation Learning using Dynamic Movement Primitives. page 21, 2009.

[23] S. Schaal, A. Ijspeert, and A. Billard. Computational approaches to motor learning by imitation. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 358(1431):537–547, March 2003.

[24] Stefan SCHAAL, Jan PETERS, and Jun NAKANISHI. Control, Planning, Learning, and Imitation with Dynamic Movement Primitives. page 21.