Mixture of Interaction Primitives for Multiple Agents

A Python framework

by

Ashish Kumar

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved July 2017 by the
Graduate Supervisory Committee:

Hani Ben Amor, Chair
Yu Zhang
Yezhou Yang

ARIZONA STATE UNIVERSITY

December 2017

ABSTRACT

In a collaborative environment where multiple robots and human beings are expected to collaborate to perform a task, it becomes essential for a robot to be aware of multiple agents working in its work environment. A robot must also learn to adapt to different agents in the workspace and conduct its interaction based on the presence of these agents. A theoretical framework was introduced which performs interaction learning from demonstrations in a two-agent work environment, and it is called Interaction Primitives.

This document is an in-depth description of the new state of the art Python Framework for Interaction Primitives between two agents in a single as well as multiple task work environment and extension of the original framework in a work environment with multiple agents doing a single task. The original theory of Interaction Primitives has been extended to create a framework which will capture correlation between more than two agents while performing a single task. The new state of the art Python framework is an intuitive, generic, easy to install and easy to use python library which can be applied to use the Interaction Primitives framework in a work environment. This library was tested in simulated environments and controlled laboratory environment. The results and benchmarks of this library are available in the related sections of this document.

ACKNOWLEDGMENTS

First of all, I would like to thank my thesis advisor Dr. Heni Ben Amor for accepting me at the Interactive Robotics Lab. It was the most significant opportunity offered to me during the span of my graduate studies at the Arizona State University. Without the guidance, support, availability, and strictness of my advisor, my goal of completing my thesis could not have been possible. His doors were always open to my questions, plans or suggestions. Working at the Interactive Robotics Lab has been a great learning experience for me, which may not have been possible otherwise.

I would also like to thank Joe Campbell, who is a colleague at the lab, for co-advising my work and helping me recognize key problems in my approach and helping me solve them. I would also like to thank my other colleagues at the lab, Kevin Sebastian Luck, Yash Rathore, Simon Stepputtis and Trevor Barron for helping me out with my theoretical or practical questions.

It was a mentally and emotionally strong decision for me to start my Masters at Arizona State University USA which is far away from my homeland, India. This dream of studying at a privileged International Institution could not have been realized without the constant support from my parents, my wife, and my elder sister.

Lastly, I would like to thank my friends at the university for constantly supporting me towards my goal to complete my thesis and helped in every way possible to stay focused on my goals.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Robotics and artificial intelligence have found numerous applications including applications in medical science, automobile industry, domestic help, security, drone technologies and others. In these scenarios, robots tend to interact with human beings or other robotic agents. Existing Interaction Primitives framework based Dynamic Movement Primitives by Heni Ben Amor (2014) and Interaction Primitives framework based on Probabilistic Movement Primitives (Interaction ProMPs) by Guilherme Maeda (2014) aims at creating a framework for learning interactions between two agents for a single task. Interaction Primitives framework (ProMPs) by Guilherme Maeda (2014) is an improvement over the original IP framework based on DMP in a way that it is based on probabilistically modeling the interactions between two agents performing a single collaborative task where the movements of the agents are encoded using ProMPs. It is a form of Interaction Learning where the framework learns from demonstrations. This framework uses trajectories encoded as probabilistic movement primitives to learn distribution of weights and generate interaction primitives for the two agents to encode a correlation between the agents in a single task work environment. The framework creates a model by learning a prior of the weights from multiple demonstrations of both the agents and applies the primitives to recognize intentions of the observed agent. This model then predicts a trajectory for an unobserved agent.

A mixture of Interaction Primitives by Marco Ewerton and Maeda (2015) aims at creating a framework to learn a mixture of interaction primitives for multiple collaborative tasks between two agents. The framework aims at learning several interaction patterns and on a certain observation of the observed agent, recognize the action and then predict remaining trajectory for the unobserved agent. The movement primitives are generated for the observed partial action.

This new state of the art Python library implements the original Mixture of Interaction Primitives paper. This library aims at achieving benchmarks which will be described in the later sections and has been tested on simulated experiments and in a controlled experiment in a laboratory. This Python library has been developed on the footprints of the theory of mixture of Interaction Primitives and the new theory which allows the Interaction Primitives framework to be applied in a work environment with more than two agents. This report contains a description on how to use this library in a later section. This library has been designed to be used in a real-time scenario. In order to make it faster and more efficient, an approximation of integrals and Bayesian inferences have been introduced. This library introduces an application of Particle Filter for phase estimation. The original theory on Interaction Primitives uses Dynamic Time Warping as a methodology to estimate a correct phase of the movement of the agents. DTW is another elegant method to estimate phase, but it's application is limited to demonstrations with less variance. It's application is also limited to a single task work environment and not in a work environment where multiple tasks are possible.

Chapter 2

MOTIVATION

Human-Robot Collaboration has been one of the topics of progressive discussion and research in the field of robotics and artificial intelligence. With the growing demand for industrial and commercial automation, a framework that assists in collaborative task between multiple agents will be a blessing in disguise. A theoretical framework already exists and will be explained in the next section. The absence of a state of the art software framework which is open-source and can be easily used was one of the key motivation for me to work on this project and take the existing theory a step above to handle collaboration between more than two agents.

A demand of automation has already opened up windows of its application in various sectors like the food industry, assembling, manufacturing, health sector where multiple agents (including human) are interacting, and the work environment is dynamic and stochastic. For example, let's portray the work situation at a Coffee Shop. A customer walks in the shop, places an order, the order is prepared by humans and robotic arms in a collaborative way and the order is finally served to the customer. Due to the involvement of a human agent, there will be a variation in the way a task is repetitively done. This variation and correlation between the agents can be easily captured by the Interaction Primitives framework. It's exciting to know the scope of this state of the art library in a world which is moving towards automation.

Chapter 3

INTERACTION PRIMITIVES

The original theory on Interaction Primitives Heni Ben Amor (2014) is based on capturing the correlation of movements between observed degree of freedoms(DOFs) and unobserved/controlled DOFs of a single or two agents. This theory can also be applied to learning correlations between the movements of an observed agent and an unobserved/controlled agent where the movements were captured using Dynamic Movement Primitives.

3.1 Interaction ProMPs

Let, P be the total number of degrees of freedom of the observed agent and Q be the total number of degrees of freedom of the controlled agent. Let, D be the combined number of degree of freedom for the observed and the controlled agent. At any time t, the position and velocity of these agents at a time step will be represented as combined vectors where

Let, P be the total number of degrees of freedom of the observed agent and Q be the total number of degrees of freedom of the controlled agent. Let, D be the combined number of degree of freedom for the observed and the controlled agent. At any time t, the position and velocity of these agents at a time step will represented as combined vectors where, $\boldsymbol{q}_t = [(\boldsymbol{q}^o)^T, (\boldsymbol{q}^c)^T]^T \in \mathbb{R}^D$ and $\dot{\boldsymbol{q}}_t = [(\dot{\boldsymbol{q}}^o)^T, (\dot{\boldsymbol{q}}^c)^T]^T \in \mathbb{R}^D$, where $(\cdot)^o$ and $(\cdot)^c$ refer to observations of the observed and controlled agents, respectively, and $\boldsymbol{q}$ and $\dot{\boldsymbol{q}}$ are position and velocity, respectively. Positions of these agents in a time-series can be represented as a trajectory $\boldsymbol{y} = \{\boldsymbol{q}_t\}_{t=0..T} \in \mathbb{R}^{D \times T}$. Observations for a single DOF of an agent can be encoded using the Probabilistic

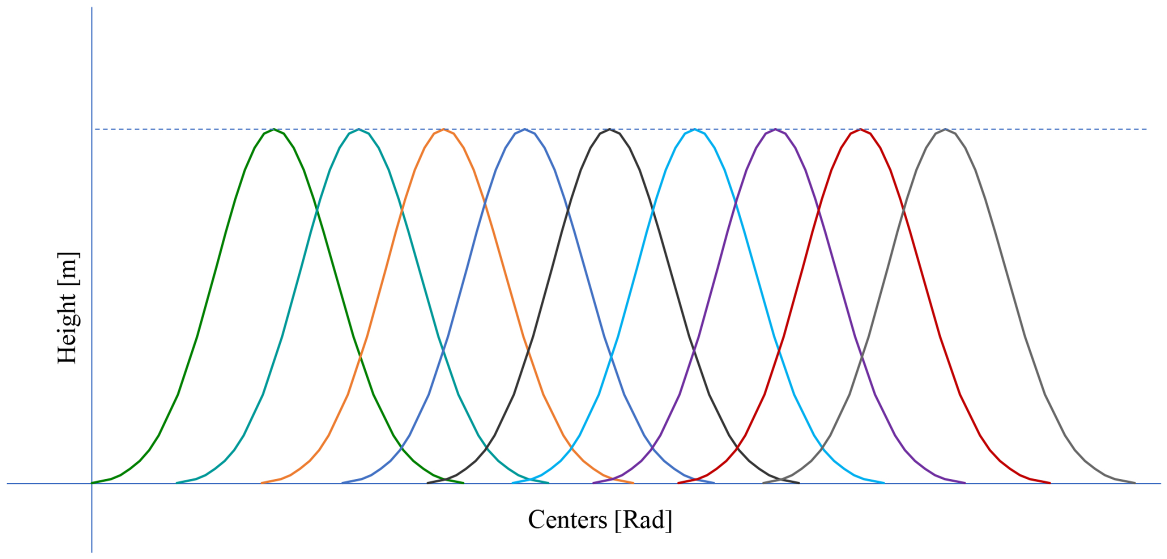


Figure 3.1: Gaussian Basis Functions

Movement Primitives Alexandros Paraschos and Neumann (2013).

$$\boldsymbol{q}_t = \boldsymbol{\phi}_t^T \boldsymbol{w} + \epsilon_y \tag{3.1}$$

where $\boldsymbol{\phi}_t \in \mathbb{R}^N$ is a time-dependent basis row vector of length N, where N is the number of basis functions, and ϵ_y is a zero-mean i.i.d Gaussian noise represented by $\epsilon_y \sim \mathcal{N}(0, \boldsymbol{\Sigma}_y) \in \mathbb{R}$. The figure 3.1 shows Gaussian basis functions been centered at a uniform distance and have the same height. If both the position and velocities have to be encoded, $\boldsymbol{\phi}_t$ becomes a $N \times 2$ basis matrix. The weight $\boldsymbol{w}$ can be learned in a least squares way. This library supports only single observation for a single degree of movement. This means that this library can either support a time-series data of only positions, joint angles, velocities, force, acceleration or any other metric. In my experiments, I have captured only the joint angles of agents in a work environment. Let's define the cost function,

$$J(\boldsymbol{w}) = \frac{1}{2} \sum_{t=1}^{T} (\boldsymbol{\phi}_t^T \boldsymbol{w} - \boldsymbol{q}_t)^2 \tag{3.2}$$

The weight $\boldsymbol{w}$ for a single trajectory can either be learned using the Least Mean Squares(LMS) Algorithm which uses gradient descent or can be calculated analytically using the least squares method,

$$\boldsymbol{w} = (\boldsymbol{\Phi}^T\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^T\boldsymbol{y}. \tag{3.3}$$

where $\boldsymbol{\Phi} \in \mathbb{R}^{T\times N}$ is a matrix of column vectors formed by transpose of basis row vectors, $\boldsymbol{\phi}_t^T$ for each time step. Our Python library calculates weights for a trajectory in the least squares sense. Weights for the all the DOF of agents can be represented by a combined row vector $\boldsymbol{w} \in \mathbb{R}^{ND}$. For an observed agent with P DOFs and a controlled agent of Q DOFs, weight $\boldsymbol{w}_d$ can be used to represent combined weights for the d-th demonstrations.

$$\boldsymbol{w}_d = \left\{[\boldsymbol{w}_1^T, ...\boldsymbol{w}_p^T, ...\boldsymbol{w}_P^T]^o, [\boldsymbol{w}_1^T, ...\boldsymbol{w}_q^T, ...\boldsymbol{w}_Q^T]^c\right\} \tag{3.4}$$

where $\boldsymbol{w}_d$ is row vector with weights $[]^o$ of observed agent and weights $[]^c$ of controlled agent. For a single demonstration, a single weight vector is learned. This weight vector activates different basis functions at each time steps to a certain proportion. The weighted sum of the basis functions is an approximating the observation at that time step. For a single demonstration and for a single DOF, the learned weight vector can be used to approximate the observed joint angles at any time step. As shown in figure 3.2, a weight vector activates each of the Gaussian Basis function and the weighted sum of the Gaussian Basis function is the approximate value of the observed joint angle at that time step. In order to learn a distribution of the weight vector, multiple varied demonstrations will be required. Figure number 3.3 shows how two agents can interact at different Interaction Points for a single task or multiple tasks. Trajectories of multiple varied demonstrations can now be recorded and a distribution of the weights $p(\boldsymbol{w}; \theta)$ can then be learned from D different demonstrations. AS shown

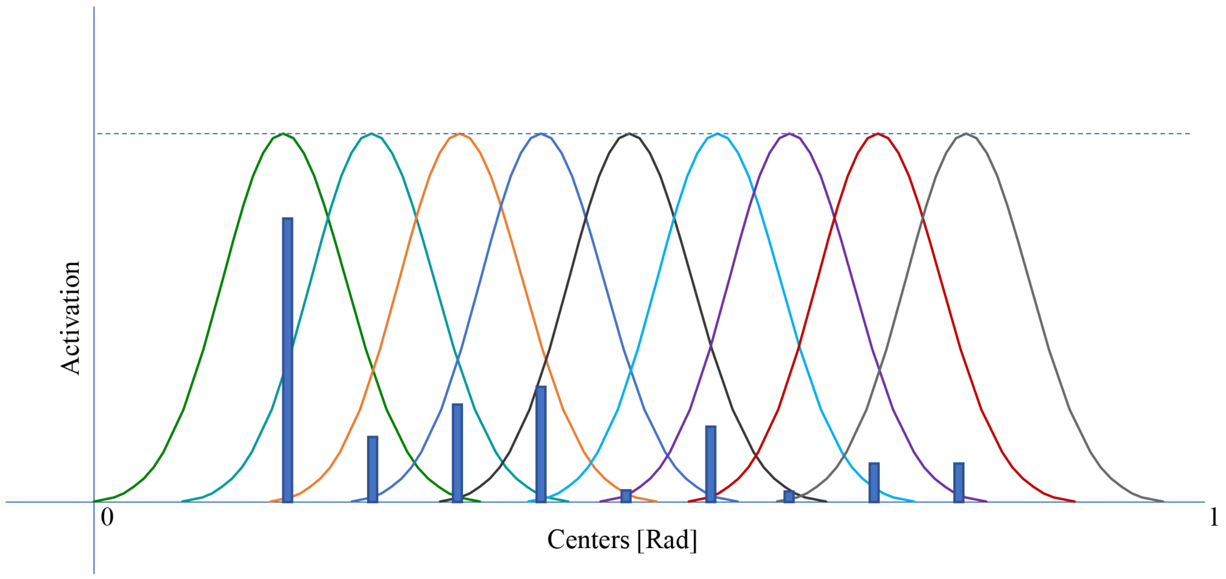


Figure 3.2: Learning Of Weights For A Single DOF

in figure 3.3, multiple demonstrations vary in a sense that they may be performing different tasks or the interaction points are varied even for a single task.

$$p(\boldsymbol{w};\theta) \sim \mathcal{N}(\boldsymbol{w};\boldsymbol{\mu}_w,\boldsymbol{\Sigma}_w), \tag{3.5}$$

$$\boldsymbol{\mu}_w = mean([\boldsymbol{w}_1, ...\boldsymbol{w}_d, ...\boldsymbol{w}_D]), \tag{3.6}$$

$$\boldsymbol{\Sigma}_w = Cov([\boldsymbol{w}_1, ...\boldsymbol{w}_d, ...\boldsymbol{w}_D]) \tag{3.7}$$

The weights can also be updated with each observation using on-line training in the form

$$\boldsymbol{\mu}_w^+ = \boldsymbol{\mu}_w^- + \boldsymbol{K}(\boldsymbol{q}_t^* - \boldsymbol{H}_t^T\boldsymbol{\mu}_w^-) \tag{3.8}$$

$$\boldsymbol{\Sigma}_w^+ = \boldsymbol{\Sigma}_w^- - \boldsymbol{K}(\boldsymbol{H}_t^T\boldsymbol{\Sigma}_w^-) \tag{3.9}$$

$$\boldsymbol{K} = \boldsymbol{\Sigma}_w^-\mathbf{H}_t^T(\boldsymbol{\Sigma}_q^* + \mathbf{H}_t^T\boldsymbol{\Sigma}_w^-\mathbf{H}_t)^{-1} \tag{3.10}$$

Here, $\boldsymbol{K} \in \mathbb{R}^{ND}$ is the Kalman gain matrix, $\boldsymbol{q}_t^*$ is the observed joint angles vector for DOFs of all agents, $\boldsymbol{\Sigma}_{\boldsymbol{q}}^* \in \mathbb{R}^{D\times D}$ is the measurement noise matrix. $()^+$ and $()^-$

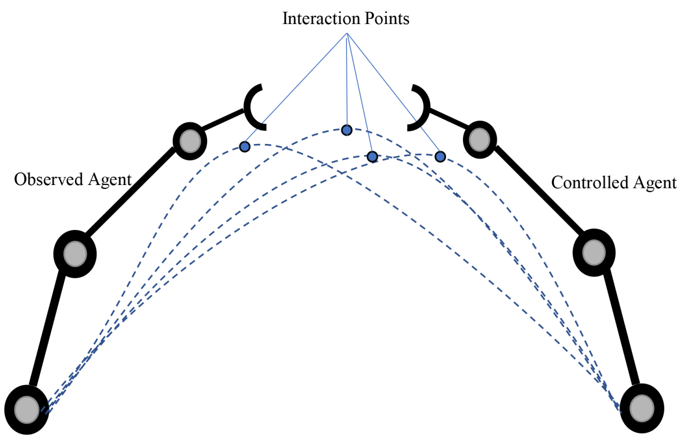


Figure 3.3: Multiple Demonstration At Different Interaction Points

represent new and old values.The observation matrix $\boldsymbol{H}_t^T \in \mathbb{R}^{D\times ND}$ is generated for each time step.

$$\mathbf{H}_t = \left[\begin{array}{cc:cc} (\boldsymbol{\phi}_t^o)^T_{(1,1)} & 0 & 0 & 0 \\ 0 & (\boldsymbol{\phi}_t^o)^T_{(P,P)} & 0 & 0 \\ \hdashline 0 & 0 & (\boldsymbol{\phi}_t^c)_{(1,1)} & 0 \\ 0 & 0 & 0 & (\boldsymbol{\phi}_t^c)_{(Q,Q)} \end{array}\right] \tag{3.11}$$

where $(\boldsymbol{\phi}_t^o)_{(p,p)}$ is a column vector of length n which contains the n basis functions at time step t.

3.1.1 Basis Functions

This python library can be configured to either use Gaussian basis functions, b_i^G, for rhythmic movements or Von-Mises basis functions, b_i^{VM}, for stroke-based moved. The choice of basis function is inspired from the work on Probabilistic Movement Primitives byAlexandros Paraschos and Neumann (2013). If there are n basis functions with centers $\boldsymbol{c} = \{c_1, ...c_i, ...c_n\} \in \mathbb{R}^N$ and height $h \in \mathbb{R}$,

$$b_i^G(\boldsymbol{q}_t) = exp\bigg(-\frac{(\boldsymbol{q}_t - c_i)^2}{2h}\bigg), \tag{3.12}$$

$$b_i^{VM}(\boldsymbol{q}_t) = exp\bigg(\frac{cos(2\pi(\boldsymbol{q}_t - c_i))}{h}\bigg) \tag{3.13}$$

For a trajectory given by $\boldsymbol{y} = \{\boldsymbol{q}_t\}_{t=0..T}$, the probability of observing any trajectory can then be calculated by

$$p(\boldsymbol{y}|\boldsymbol{w}) = \prod_{t=0}^{T} \mathcal{N}(\boldsymbol{q}_t|\boldsymbol{\phi}_t^T\boldsymbol{w}, \boldsymbol{\Sigma}_y) \tag{3.14}$$

The trajectory distribution can then be calculated by marginalizing out the weight.

$$p(\boldsymbol{y};\theta) = \int p(\boldsymbol{y}|\boldsymbol{w})p(\boldsymbol{w};\boldsymbol{\theta})d\boldsymbol{w} \tag{3.15}$$

For a given set of D demonstrations, θ can be estimated as described in Equations (3.6) and (3.7). It follows that,

$$p(\boldsymbol{y}) = \int p(\boldsymbol{y}|\boldsymbol{w})p(\boldsymbol{w})d\boldsymbol{w} \tag{3.16}$$

Once the Interaction primitives, $\boldsymbol{\mu}_w$ and $\boldsymbol{\Sigma}_w$, have been learned by using this framework on data from D demonstrations, these interactives can be used to predict the trajectory of a controlled agent by observing an observed agent and this prediction can be performed at each time step t using the updation method specified by equations (3.9) and (3.10).

3.1.2 Time Scaling

Different agents can move at different speeds and the number of observations for the agents will vary in a certain period of time. Due to the possibility of agents executing movements at various speed, our system decouples the movement from the time signal by introducing a phase variable τ, which varies from 0 to 1. This way, movements executing at different speeds can be encoded between these phase values.

3.1.3 Prediction based on Interaction ProMPs

During the training phase and for a single demonstration d, an observation is captured at each phase value for all DOFs, $\boldsymbol{q}_\tau = \{\boldsymbol{q}_\tau^o,\ \boldsymbol{q}_\tau^c\}$, where $\boldsymbol{q}_\tau^o$ are the observations of the joint angles of an observed agent and $\boldsymbol{q}_\tau^c$ are the observations of the joint angles of a controlled agent. Once training data has been collected and the Interaction Primitives have been generated, this model can then be used to perform prediction of the remaining trajectory of the controlled agent based on partial observation of the observed agent, $\boldsymbol{q}_\tau^{*o}$ at each phase value τ. During the prediction phase, at each phase value τ, a column vector $\boldsymbol{q}^*$ to represent the combined joint angles of observed and controlled agent is generated, where $\boldsymbol{q}^* = \{\boldsymbol{q}^{*o}, 0_{Q,1}\}$. The conditional distribution $p(\boldsymbol{w}^+|\boldsymbol{q}_\tau^*)$ is also Gaussian and the new mean and covariance can be calculated by

$$\boldsymbol{\mu}_w^+ = \boldsymbol{\mu}_w + \boldsymbol{K}(\boldsymbol{q}_\tau^* - \boldsymbol{H}_\tau^T\boldsymbol{\mu}_w) \tag{3.17}$$

$$\boldsymbol{\Sigma}_w^+ = \boldsymbol{\Sigma}_w - \boldsymbol{K}(\boldsymbol{H}_\tau^T\boldsymbol{\Sigma}_w) \tag{3.18}$$

$$\boldsymbol{K} = \boldsymbol{\Sigma}_w\mathbf{H}_\tau^T(\Sigma_q^* + \mathbf{H}_\tau^T\boldsymbol{\Sigma}_w\mathbf{H}_\tau)^{-1} \tag{3.19}$$

where,

$$\boldsymbol{q}_\tau^* = [\boldsymbol{q}_t^{*o}; 0_{Q,1}], \text{no observation for controlled agent.} \tag{3.20}$$

$$\mathbf{H}_t = \left[\begin{array}{cc:cc} (\boldsymbol{\phi}_\tau^o)^T_{(1,1)} & 0 & 0 & 0 \\ 0 & (\boldsymbol{\phi}_\tau^o)^T_{(P,P)} & 0 & 0 \\ \hdashline 0 & 0 & (0^c_{N,1})_{(1,1)} & 0 \\ 0 & 0 & 0 & (0^c_{N,1})_{(Q,Q)} \end{array}\right] \tag{3.21}$$

$$\boldsymbol{\Sigma}_q^* = \left[\begin{array}{cc:cc} \alpha & 0 & 0 & 0 \\ 0 & \alpha & 0 & 0 \\ \hdashline 0 & 0 & \beta & 0 \\ 0 & 0 & 0 & \beta \end{array}\right] \tag{3.22}$$

where α is a small noise value which reflects our confidence in the observation and β is a large noise value which reflects low confidence since there is no observation made for the controlled agents. An important point to note here is that the Interaction Primitives library generates a predicted trajectory for the controlled agent based on the mean and covariance of the conditional distribution. As shown in figure number 3.4, there are still numerous possibilities for a controlled agent. These possibilities can be harnessed by randomly choosing weight vectors from the normal distribution, $\mathcal{N}(\boldsymbol{\mu}_w^+, \boldsymbol{\Sigma}_w^+)$.

3.1.4 Phase Estimation

Due to the decoupling of movements from the time signal and use of phase variable τ, it is important to do a refined phase estimation for better accuracy in the conditioning step. Our system uses Particle Filtering to estimate the phase of the

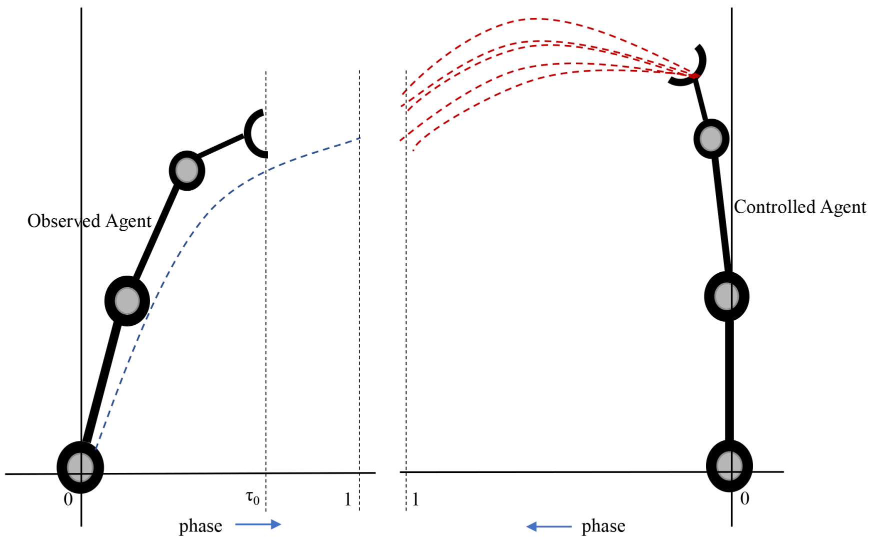


Figure 3.4: Multiple Possibilities For The Controlled Agent

observed agent for a correct conditioning. Particle Filters are based on Bayesian Filters which is, in turn, views the latent state space as a Markov Chain. Let, τ_1, τ_2,...,τ_t,...,τ_T be the different latent states(or phases) of a dynamic system made up by work environment with multiple agents, where the system transitions from one state to another depending on an observation made. Let, $\boldsymbol{q}_1$, $\boldsymbol{q}_2$,...,$\boldsymbol{q}_t$,...,$\boldsymbol{q}_T$ be the set of observation made at different time steps. As per the Markov Chain assumption, the probability of the current hidden state of a system is only dependent on the last state and conditionally independent of any previous states. The following derivation is inspired from Turner (2013) That is,

$$p(\tau_t|\tau_{0:t-1}) = p(\tau_t|\tau_{t-1}) \tag{3.23}$$

Also, at any time step t, the observation $\boldsymbol{q}_t$ only depends on the latent state τ_t at that time step and is conditionally independent of any previous states. That is,

$$p(\boldsymbol{q}_t|\tau_{0:t}) = p(\boldsymbol{q}_t|\tau_t) \tag{3.24}$$

The distribution of the current state, given all observation can then be calculation using marginal probability in the following way:

$$p(\tau_{0:t}|\boldsymbol{q}_{0:t}) = p(\tau_{0:t-1}|\boldsymbol{q}_{0:t-1})\frac{p(\tau_t|\tau_t - 1)p(\boldsymbol{q}_t|\tau_t)}{p(y_t|y_{0:t-1})} \tag{3.25}$$

$$p(\tau_t|\boldsymbol{q}_{0:t}) = \frac{p(\tau_t|\boldsymbol{q}_{0:t-1})p(\boldsymbol{q}_t|\tau_t)}{p(\boldsymbol{q}_t|\boldsymbol{q}_{0:t-1})} \tag{3.26}$$

$$p(\tau_t|\boldsymbol{q}_{0:t}) = p(\boldsymbol{q}_t|\tau_t)\int p(\tau_{t-1}|\boldsymbol{q}_{0:t-1})\frac{p(\tau_t|\tau_{t-1})}{p(\boldsymbol{q}_t|\boldsymbol{q}_{0:t-1})}d\tau_{t-1} \tag{3.27}$$

where,

$$p(\boldsymbol{q}_t|\boldsymbol{q}_{0:t-1}) = \int p(\boldsymbol{q}_t|\tau_t)\left\{\int p(\tau_{t-1}|\boldsymbol{q}_{0:t-1})p(\tau_t|\tau_{t-1})d\tau_{t-1}\right\}d\tau_t \tag{3.28}$$

In the above equations, it is not computationally tangible to calculate the integrals. To make an approximate Bayesian inference, the concept of Importance Sampling is used. This method of approximate Bayesian inference for the calculation of the probability of current state using Importance Sampling is called Particle Filters. In our implementation, the Sequential Monte Carlo Algorithm has been used. An overview of the algorithm is as follows:

(i) **Sampling** - Draw n samples $\tau_{0:t-1}$ from the previous set and generate n new samples using the distribution: $p(\tau_t|\tau_{t-1})$.

(ii) **Importance Sampling** - Assign each sample an importance weight according to the likelihood of the observation.

(iii) **Re-Sampling** - Multiply/Discard samples by drawing samples with replacement according to the distribution defined through the importance weight.

In this implementation, the Re-Sampling Wheel Algorithm has been used for the Re-Sampling step. This library performs importance sampling by calculation the probability of current observation given the value of a particle. This probability is calculated by fitting an observation in the normal distribution $\mathcal{N}(\boldsymbol{\phi}_{\boldsymbol{\tau}}^{\boldsymbol{T}}\boldsymbol{w}, \boldsymbol{\Sigma}_{\boldsymbol{y}})$, where $\boldsymbol{\Sigma}_{\boldsymbol{y}}$ is the covariance of the demonstration data given by, $\boldsymbol{\Sigma}_{\boldsymbol{y}} = \boldsymbol{H}_{\tau}^{T}\boldsymbol{\Sigma}_{w}\boldsymbol{H}_{\tau}$. Probabilities of each particles are calculated and are normalized. At the last observation, and after importance sampling, the phase decided by choosing the value of the particle with the highest weight.

3.2 Mixture of Interaction Primitives

Using a mixture of Interaction Primitives, interactions ranging over multiple tasks between two agents can also be used to learn weights for these interactions. This arises an intuition of a possible clustering in the weight space. A Gaussian Mixture Model in the weight space is learned to capture the possibility of multiple clusters. With the mixture of Interaction Primitives theory in place, it is now important to determine the most probable action performed by the observed agent in a partial action and then the generate the remaining trajectories of the controlled agent using conditioning over the interaction primitives.

3.2.1 Gaussian Mixture Model

With the application of Interaction ProMPs on a set of observations, a distribution of weights can be learned. In the case where training is done over multiple tasks or highly varied tasks, it becomes intuitive to expect the presence of mixture models in the learned distribution. The Gaussian Mixture Model Expectation-Maximization algorithm can now be applied to learn the mixture parameters in the weight space. The algorithm to calculate the parameters has already been explained in the original

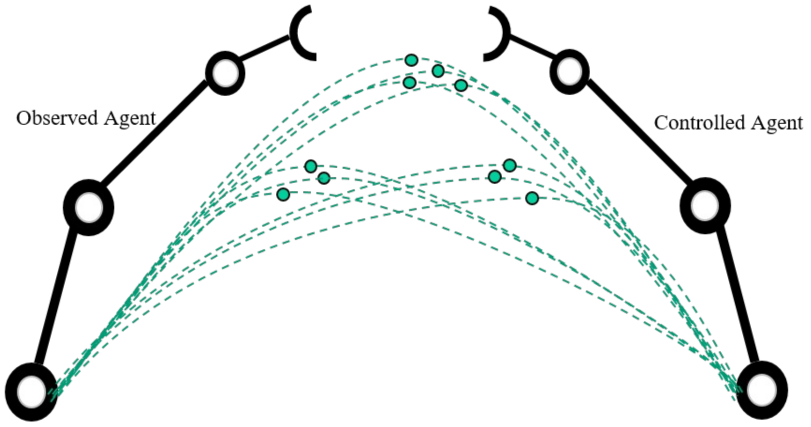


Figure 3.5: Multiple Collaboration For A Mixture Of Interaction Primitives

work on Mixture of Interaction Primitives by Heni *et al* 2015. The following algorithm has been taken from the original work:

The integration in the inference step is complex and in order to efficiently perform this integration, I have used an approximation for this Integration. The approximation works by fitting a partial observation into a normal distribution of the approximated trajectory from the mean of weights of a mixture and the covariance of the trajectories. That is, for a partial observation $\boldsymbol{y}^*$ up to a certain phase value τ_0,

$$p(D|k) = \int p(D|\bar{\boldsymbol{w}})p(\bar{\boldsymbol{w}}|k)d\bar{\boldsymbol{w}} \tag{3.36}$$

$$p(D|k) \approx \sum_{\tau=0}^{\tau_0} \mathcal{N}(\boldsymbol{q}_\tau; \boldsymbol{\phi}_\tau^T \boldsymbol{\mu}_{\boldsymbol{k}}, \boldsymbol{\Sigma}_{\boldsymbol{y}}) \tag{3.37}$$

where,

$$\boldsymbol{\Sigma}_{\boldsymbol{y}} = \boldsymbol{H}_\tau^T \Sigma_w \boldsymbol{H}_\tau \tag{3.38}$$

Algorithm 1 Training

1) **Parameterize demonstrated trajectories**: Find vector of weights $\bar{\boldsymbol{w}}$ for each trajectory, such that $\mathbf{q}_t \approx \boldsymbol{\phi}_t^T \bar{\boldsymbol{w}}$.

2) **Find GMM in parameter space, using EM**: Initialize GMM parameters $\alpha_{1:K}$, $\mu_{1:K}$ and $\Sigma_{1:K}$ with k-means clustering.

repeat

E step

$$r_{ik} = p(k|\bar{\boldsymbol{w}}_{\boldsymbol{i}}) = \frac{\mathcal{N}(\bar{\boldsymbol{w}}_{\boldsymbol{i}}; \boldsymbol{\mu}_{\boldsymbol{k}}, \boldsymbol{\Sigma}_{\boldsymbol{k}})\alpha_k}{\sum_{l=1}^{K} \alpha_l \mathcal{N}(\bar{\boldsymbol{w}}_{\boldsymbol{i}}; \boldsymbol{\mu}_{\boldsymbol{l}}, \boldsymbol{\Sigma}_{\boldsymbol{l}})} \tag{3.29}$$

M step

$$n_k = \sum_{i=1}^{n} r_{ik}, \alpha = \frac{n_k}{n} \tag{3.30}$$

$$\boldsymbol{\mu}_{\boldsymbol{k}} = \frac{\sum_{i=1}^{n} r_{ik} \bar{\boldsymbol{w}}_{\boldsymbol{i}}}{n_k} \tag{3.31}$$

$$\boldsymbol{\Sigma}_{\boldsymbol{k}} = \frac{1}{n_k} \left(\sum_{i=1}^{n} (\bar{\boldsymbol{w}}_{\boldsymbol{i}} - \boldsymbol{\mu}_{\boldsymbol{k}})(\bar{\boldsymbol{w}}_{\boldsymbol{i}} - \boldsymbol{\mu}_{\boldsymbol{k}})^T \right) \tag{3.32}$$

until $p(\bar{\boldsymbol{w}}_{\boldsymbol{i}}; \boldsymbol{\alpha}_{\mathbf{1:K}}; \boldsymbol{\mu}_{\mathbf{1:K}}; \boldsymbol{\Sigma}_{\mathbf{1:K}})$ converges.

Algorithm 2 Inference

1) **Find most probable cluster given observation:**

$$p(k|D) \propto p(D|k)p(k) \tag{3.33}$$

$$k^* = \arg\max_x p(k|D) \tag{3.34}$$

2) **Condition on observation, using cluster** k^*:

$$p(\boldsymbol{q}_{1:T}|D) = \int p(\boldsymbol{q}_{1:T}|\bar{\boldsymbol{w}}) p(\bar{\boldsymbol{w}}|k^*, D) d\bar{\boldsymbol{w}} \tag{3.35}$$

3.2.2 Mixture of Probabilistic PCA

This library further extends the formation of mixture models, as suggested in the original research, in a way that it also provides the ability to select the Mixture of Probabilistic Principal Component Analysis for the mixture model parameter calculation. This implementation of a mixture of PPCA has been done on the guidelines of the original research by Tipping and Bishop (1999).

Latent Variable Model

Let's first define a latent variable $\boldsymbol{l} \in \mathbb{R}^L$ which has a dimension $L < D$. Let's also define a function f(;) of the latent variable with parameters $\boldsymbol{p}$. This new function f(;) serves the purpose of reconstruction from a latent space to weight space.

$$\boldsymbol{w} = f(\boldsymbol{l}; \boldsymbol{p}) + \epsilon$$

where, ϵ is an isotropic reconstruction error, that is, $\epsilon \sim \mathcal{N}(0, \sigma^2\mathbf{I})$.

Factor Analysis

The theory on statistical factor analysis by Bartholomew (1987) can be used to define a linear reconstruction function in the following way,

$$\boldsymbol{w} = \mathbf{W}\boldsymbol{l} + \boldsymbol{\mu}_{\boldsymbol{w}} + \epsilon \tag{3.39}$$

where, the latent variables $\boldsymbol{l}$ have a normal distribution with zero-mean and unit variance, that is, $\boldsymbol{l} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. The noise has a normal distribution given by $\epsilon \sim \mathcal{N}(0, \sigma^2\mathbf{I})$. The *weight* matrix has to be approximated using parameterized learning in case of probabilistic PCA, as will be shown in later section. The weight vector has a normal distribution, given the above formulation and is given by $\boldsymbol{w} \sim \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{w}}, \boldsymbol{\Sigma}_{\boldsymbol{w}})$.

Probabilistic PCA

As can be seen in the statistical factor analysis model, reconstruction from the latent space to the weight space an Gaussian error associated with it. This gives rise to the existence of a probability distribution over the weight space given the latent variable $\boldsymbol{l}$. The equation (3.39) can be used to establish this conditional probability distribution and is given by,

$$p(\boldsymbol{w}|\boldsymbol{l}) = (2\pi\sigma^2)^{-D/2} exp\Big\{ -\frac{1}{2\sigma^2}||\boldsymbol{w} - \mathbf{W}\boldsymbol{l} - \boldsymbol{\mu}||^2 \Big\} \tag{3.40}$$

Since, $\boldsymbol{l} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. Hence by marginalization,

$$p(\boldsymbol{w}) = \int p(\boldsymbol{w}|\boldsymbol{l})p(\boldsymbol{l})d\boldsymbol{l} \tag{3.41}$$

$$or, p(\boldsymbol{w}) = (2\pi)^{-D/2}|\boldsymbol{\Sigma}_{\boldsymbol{w}}|^{-1/2} exp\Big\{ -\frac{1}{2}(\boldsymbol{w} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}_{\boldsymbol{w}}^{-1}(\boldsymbol{w} - \boldsymbol{\mu}) \Big\} \tag{3.42}$$

The sample covariance matrix $\boldsymbol{S}$ and $\boldsymbol{\mu}$of the weights can be calculated using maximum likelihood from the set of weights generated for D demonstrations using the equation (3.6) and the following formula,

$$\boldsymbol{S} = \frac{1}{D}\sum_{d=1}^{D}(\boldsymbol{w}_d - \boldsymbol{\mu}_{\boldsymbol{w}})(\boldsymbol{w}_d - \boldsymbol{\mu}_{\boldsymbol{w}})^T \tag{3.43}$$

It is also shown in the original work that,

$$\boldsymbol{W}_{ML} = \boldsymbol{U}_L(\boldsymbol{\Lambda}_L - \sigma^2\mathbf{I})^{1/2}\mathbf{R} \tag{3.44}$$

where, the matrix $\mathbf{U}_L$ contains the first L Eigen vectors of $\mathbf{S}$. $\boldsymbol{\Lambda}_L$ is a $L \times L$ diagonal matrix with the L Eigen values. $\mathbf{R}$ is an arbitrary rotation matrix. σ^2 can also be estimated using maximum likelihood as described in the original research,

$$\sigma^2_{ML} = \frac{1}{D-L}\sum_{j=L+1}^{D}\lambda_j \tag{3.45}$$

where, $\sum_{j=L+1}^{D}\lambda_j$ is the sum of the smallest Eigen values of $\mathbf{S}$.

Mixture of PPCA

As discussed earlier, it's only intuitive that a mixture model exists in the weight space when the weights are learned for multiple tasks or highly varied tasks. The original theory of a mixture PPCA suggests a different approach to the formulation of mixture models and their parameter calculation. An iterative EM algorithm for the calculation of the calculation of the model parameters $\alpha_{1:K}$, $\boldsymbol{\mu}_{1:K}$, $\mathbf{W}_{1:K}$, $\Sigma_{1:K}$ and $\sigma_{1:K}$ is as follows:

Algorithm 3 EM Algorithm for Mixture of PPCA

Initialize the parameters α_k and $\boldsymbol{\mu}_k$ using K-means clustering.

E step

$$r_{ik} = \frac{p(\boldsymbol{w}_i|k)\alpha_k}{p(\boldsymbol{w}_i)} \tag{3.46}$$

M step

$$\alpha_k = \frac{1}{N}\sum_{i=1}^{D} r_{ik} \tag{3.47}$$

$$\boldsymbol{\mu_k} = \frac{\sum_{i=1}^{D} r_{ik}\boldsymbol{w}_i}{\sum_{i=1}^{D} r_{ik}} \tag{3.48}$$

$$\boldsymbol{S}_k = \frac{1}{\alpha_k D}\sum_{i=1}^{D} r_{ik}(\boldsymbol{w}_i - \boldsymbol{\mu}_k)(\boldsymbol{w}_i - \boldsymbol{\mu}_k)^T \tag{3.49}$$

$$\boldsymbol{\Sigma}_k = \sigma_i^2\mathbf{I} + \mathbf{W}_i\mathbf{W}_i^T \tag{3.50}$$

σ_i^2 and $\mathbf{W}_i$ are calculated using equations (3.44) and (3.45).

3.2.3 Number of cluster determination

One of the key parameters in any approach towards mixture model/clustering is to determine the parameter K, the number of clusters, as correctly as possible. One of the approach suggested in Marco Ewerton and Maeda (2015) is to use K-Fold cross validation to determine a decent value of K. Although, this method gives fairly good results, it is a time-complex approach and can sometimes be computationally very expensive. A faster approximation of the value K can be done by applying the G-Means Algorithm suggested in Hamerly and Elkan (1999).

3.3 Controlled arm halting at anomaly detection

The original framework has been extended to make the controlled arm halt at its current joint angles whenever it detects an anomaly with the position of a tool/object in the work environment. This feature is advised to be only used when there is a presence of a tool/object in the work space. The framework captures multiple trajectories $\boldsymbol{\tau}^o = \{\boldsymbol{\tau}_1^o, \boldsymbol{\tau}_2^o, ..., \boldsymbol{\tau}_t^o, ..., \boldsymbol{\tau}_T^o\}$. The framework then learns a normal distribution of these trajectories, $\mathcal{N}(\boldsymbol{\mu^o}, \boldsymbol{\Sigma^o})$. The framework then fits a prediction of the object/tool joint angles in this normal distribution and if the probability is less than a threshold γ, the controlled arm halts.

3.4 Mixture of Interaction Primitives for Multiple Agents

My framework extends the original theory of Mixture of Interaction primitives for more than two agents. The framework adopts two approaches towards to handle the case of more than two agents:

(i) **Learning one to one correlation** - This framework defines the approach towards learning correlations or Interaction Primitives between two pairs of agents. In either approach, there is a single observed agent and multiple controlled agents. Upon a partial observation of the observed agent, the trajectories of the controlled agents are predicted.

(ii) **Learning one to all correlation** - In this approach, a single correlation or

Interaction Primitives is generated between the DOFs of the observed agent and the combined DOFs of all the controlled agent.

3.4.1 Learning one to one correlation

Let's assume there are m agents given $A = \{A_1, A_2, ..., A_m\}$. In this approach we aim at learning (m - 1) interaction primitives between each two pairs of interacting agents. The first agent, A_1, is the primary observed agent. The goal is to generate predicted trajectories for the remaining (m - 1), $\{A_2, A_3, ..., A_m\}$, controlled agents on a partial observation of A_1 at each value of phase. Let, P be the total DOFs of A_1 and $\{Q_1,\ Q_2,..,Q_{M-1}\}$ be the DOFs of each of $\{A_2, A_3, ..., A_m\}$. Let, D be the total number of combined DOFs. For a single demonstration and for a single phase value, the joint angles of the agents can be represented by a combined vector $\boldsymbol{q}_t$. This vector is created in the same way as created in the case of two-agents by combining the observation. An entire demonstration can be captured using a matrix, $\boldsymbol{y} = \{\boldsymbol{q}_\tau\}_{\tau=0..1}$ $\in \mathbb{R}^{D\times T}$. Interaction primitives between two agents can then be learned by learning individual primitives between each interacting pair. This is illustrated in the figure number 3.6. During learning the primitives for an interacting pair, one of the agent is made the observed agent and the other is made a controlled agent. The theory defined in section (3.1) can then be used to learn (m -1) different interaction primitives given by, $\{(\boldsymbol{\mu}_w^1, \boldsymbol{\Sigma}_w^1),\ (\boldsymbol{\mu}_w^2, \boldsymbol{\Sigma}_w^2),...,\ (\boldsymbol{\mu}_w^m, \boldsymbol{\Sigma}_w^m)\}$.

During the prediction phase, upon a partial observation, $\boldsymbol{q}_\tau^{*A_1}$ of the primary observed agent, predicted trajectories of the controlled agents are calculated. This is performed by chaining the prediction step and using the predicted trajectory of the last step as an observation for the next step.

Algorithm 4 Learning one to one correlation

```
(i) Collection training demonstrations and learning IP between each interaction
    pair $(A_1, A_2)$, $(A_2, A_3)$,...,($(A_1, A_2)$, $(A_{m-1}, A_m)$).
(ii) These interaction primitives can be represented by $\{(\boldsymbol{\mu}_w^1, \boldsymbol{\Sigma}_w^1)$, $(\boldsymbol{\mu}_w^2, \boldsymbol{\Sigma}_w^2)$,...,
    $(\boldsymbol{\mu}_w^m, \boldsymbol{\Sigma}_w^m)\}$.
(iii) During prediction phase, chain the predictions. Let, $\boldsymbol{q}_\tau^{*A_1}$ be the observation of
    the agent, $A_1$. The IPs learned for the pair $(A_1, A_2)$ can be used to predict
    trajectory for $A_2$. Let the predicted trajectory be $\boldsymbol{q}_t^{*A_2}$. This predicted trajec-
    tory can be used as an observation for the IPs for the second pair $(A_2, A_3)$ to
    generated a predicted trajectory $\boldsymbol{q}_t^{*A_3}$ for the agent, $A_3$.
(iv) Perform this chaining for all the remaining agents.
(v) Perform step(iii) and step(iv) for all remaining phase values.
```

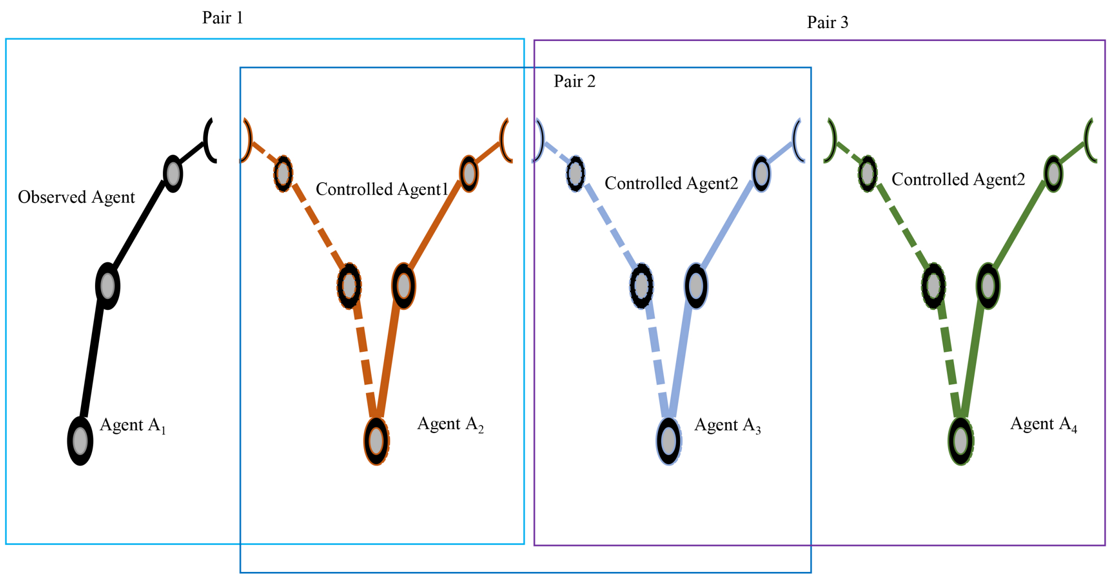


Figure 3.6: Learning Interaction Primitives Between Interacting Pairs

3.4.2 Learning one to all correlation

Under this approach, the underlying assumptions about the agents and interactions are the same. The difference lies in the fact that in this approach a single Interaction Primitive is learned. Under this approach, the first agent is the observed agent and the degrees of freedoms for all the controlled agents is treated in a combined way. That is, for a single demonstration and for a single time step, $\boldsymbol{q}_\tau = [(\boldsymbol{q}_\tau^o)^T, (\boldsymbol{q}_\tau^c)^T]^T$, where $(\boldsymbol{q}_\tau^c)$ is the combined observation vector for all the controlled agents. Also for each demonstration d, the learned weight vector is given by,

$$\boldsymbol{w}_d = \{[\boldsymbol{w}_1^T, ...\boldsymbol{w}_p^T, ...\boldsymbol{w}_P^T]^o, [\boldsymbol{w}_1^T, ...\boldsymbol{w}_{q_2}^T, ...\boldsymbol{w}_{Q_2}^T]^{c_2}, ..., [\boldsymbol{w}_1^T, ...\boldsymbol{w}_{q_m}^T, ...\boldsymbol{w}_{Q_m}^T]^{c_m}\} \tag{3.51}$$

where c_j represents a controlled agent A_j. This is illustrated in the figure number 3.7.

During the prediction phase, on a partial observation $(\boldsymbol{q}_\tau^o)$ of the observed agent, a predicted trajectory of all the controlled agent is generated at once.

The major difference between the two approaches is primarily based on how the approaches handle space-time complexity. The first approach performs matrix operations on (m - 1) smaller matrices as compared to a big correlation matrix when all the controlled agents are treated in a combined way. The first approach is ideal when a faster speed is required but each agent has high values of DOFs. The second approach is ideal when a fast speed is required given that agents have a small value of DOFs.

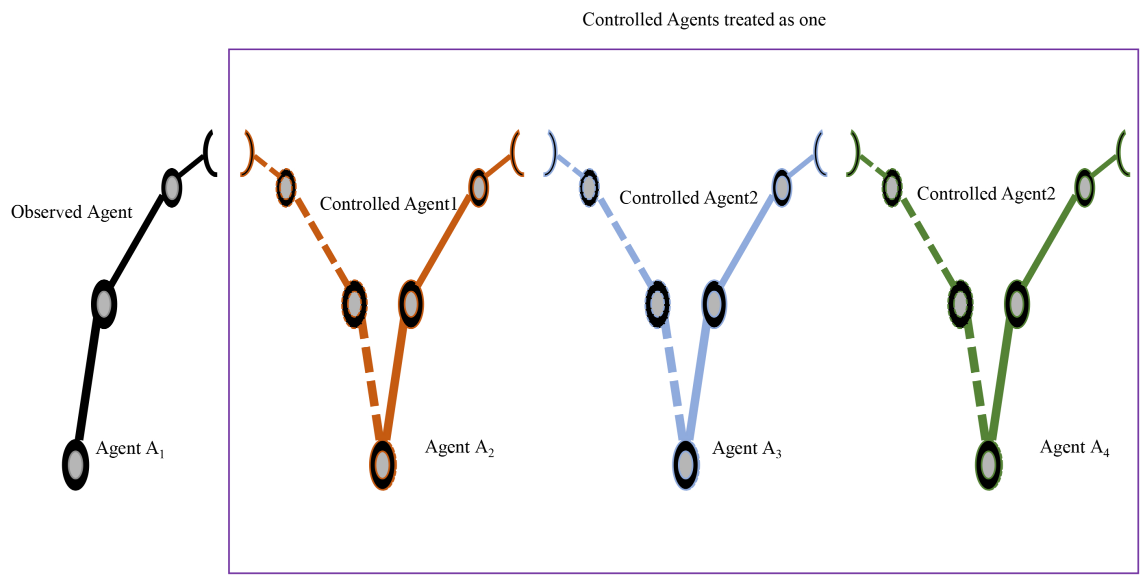


Figure 3.7: Learning Interaction Primitives Between The Observed Agent And Combined Controlled Agents

Chapter 4

PYTHON FRAMEWORK

4.1 Overview

The Interaction Primitives Python library created for the theoretical framework of the Mixture of Interaction Primitives for multiple agents is a state of the art Python library than can be applied to a work environment where there is a collaboration between multiple agents. This framework is an open-source library and is developed completely on Python 2.7. This library has been tested on Windows 10 platform as well as Ubuntu 14.04 platform, and no other platforms. Some salient features of this library are as follows:

(i) **Open Source** - This software library can be used free of charge and is currently licensed under the MIT Open Source Initiative.

(ii) **Easy to Install and Use** - There are two core files of this library and other extended files which can be used on a need basis. Importing the core files is fairly easy and will be explained in the later section.

(iii) **Fast** - This library uses several forms of Bayesian approximation and approximation of big integrals to speed up the library in general.

(iv) **Customizable** - The library provides various ways to customize the core framework as well as several functionalities related to this library.

(v) **Well Tested** - This library has been well-tested in different simulated environments and one human experiment.

4.2 Design Pattern

The framework has been created using object-oriented programming and follows guidelines of object-oriented programming. The library has been developed using *Factory* - *Creational* design pattern, *Chain of Responsibility* - *Behavioral* design pattern and *Decorator* - *Structural* design pattern.

4.3 Libraries Used

In order to create this library, several python libraries has been used to make it robust and efficient. The list of python libraries used are:

(i) Numpy.

(ii) Scipy.

(iii) Scikit-Learn.

(iv) Matplotlib

(v) StatsModels

4.4 Usage

The library can be used to either be applied in a two-agent work environment or a more-than-two-agent work environment. The core system of this library has two classes *InteractionPrimitives* and *InteractionPrimitivesMulti*, which can be used either in a two-agent setup or multiple agents set up. The library has an inbuilt implementation of the G-Means clustering algorithm which can be used to calculate an approximate number of clusters based on the G-Means algorithm. The software package has various examples of calculating the number of clusters using different testing methods for K-Fold cross-validation.

4.4.1 *Object Instantiation*

This step creates an appropriate core object to perform data fitting, clustering and prediction for Mixture of Interaction Primitives. The core classes have similar APIs for these tasks and are described more in the later sections.

(i) Two Agent Work Environment. The design patterns used by the library provides a decent level of abstraction from the background programming done to create the main objects which allow generating Interaction Primitives. For a two-agent work environment, the following factory method can be used to create an instance of the *InteractionPrimitives* class. An example usage of the instantiation step,

```
dof_observed_agent = x #x is the number of DOFs of an observed
    agent
dof_controlled_agent = y #y is the number of DOFs of a controlled
    agent
dof_agents = [dof_observed_agent, dof_controlled_agent]
number_of_gaussians = n #n is the number of basis functions
number_of_cluster = c #if known, else should be -1
noise_error_value = e #e is the gaussian noise
environment_type = 'single' #the other allowed value is multi

#Following are some optional parameters than can be used to
    enable/disable optional features and can also be used to
    customize a model.
disable_log = False # to disable any form of logging,
other allowed value is True
disable_plot = False # to disable/enable the plotting of the
    weights after performing a dimensionality reduction using PCA.
basis_function_type = "Gaussian" #the other possible value is
```

```
    Von-Mises
ip = InteractionPrimitivesFactory.get_instance(dof_agents,
    environment_type, gaussians, number_of_cluster,
    noise_error_value, disable_log, disable_plot,
    basis_function_type)}
```

In the above snippet, *InteractionPrimitivesFactory* class is used to get an instance of the *InteractionPrimitives* class. The *InteractionPrimitivesFactory* class has a function, *getInstance*, which acts as a factory function and can either return either an object of the *InteractionPrimitives* class or an object of the *InteractionPrimitivesMulti* class. In the above code snippet, the first argument of the *getInstance* function is an array containing the number of DOFs of the two agents. As shown in the code snippet below, this array contains the number of DOFs of all the agents in a multi-agents work environment.

The second argument of the *getInstance* function is a String literal, which can either take the value *single* for a two-agent work environment or the value *multi* for a multi-agent work environment.

The third argument of the *getInstance* function is an integer having the value as the total number of basis functions to be used for the regression. In our example, we are using *Gaussian* basis functions and this argument defines the total number of *Gaussian* basis function to be used.

If the number of clusters is known in advance, by K-Fold cross validation or any other method, the fourth argument can be used to specify that. Otherwise, it can be specified later on.

Since our library uses Probabilistic Movement Primitives, it becomes important to specify a value for the Gaussian noise. This can be specified using the fifth argument.

In the above code snippet, rest of the arguments are optional and intuitive. The sixth argument is used to enable/disable logs. The library has a feature which can be used to perform a plot of the learned weights in a 3-D space and this dimensionality reduction is performed using PCA. The seventh argument is used to enable/disable plotting.

As mentioned in the section (3.1), the framework supports two types of basis functions: *Gaussian* and *Von-Mises*. The eighth argument is used to specify the type of basis function to be used.

(ii) More than two agent work environment. The core *InteractionPrimitivesMulti* object can be fetched using the factory class using a similar way described above,

```
dof_observed_agent = x #x is the number of DOFs of an observed
    agent
dof_controlled_agent1 = y1 #y is the number of DOFs of first
    controlled agent
dof_controlled_agent2 = y2 #y is the number of DOFs of second
    controlled agent
...

dof_controlled_agentn = yn #y is the number of DOFs of nth agent
dof_agents = [dof_observed_agent,
    dof_controlled_agent1,..dof_controlled_agentn]
number_of_gaussians = n #n is the number of basis functions
number_of_cluster = c #if known, else should be -1
noise_error_value = #e is the gaussian noise
environment_type = 'multi'

#optional parameters
```

```
method = 'oto' #should only be used in case of multi, other
    allowed value is 'all'
disable_log = False # to disable any form of
logging, other allowed value is True
disable_plot = False # to disable/enable the plotting
basis_function_type = "Gaussian" #the other possible value is
    Von-Mises
of the weights after performing a dimensionality reduction using
    PCA.

ip = InteractionPrimitivesFactory.get_instance(dof_agents,
    environment_type, gaussians, number_of_cluster,
    noise_error_value, disable_log, disable_plot,
    basis_function_type, method)
```

The above code snippet is similar to the code snippet for the instantiation of the *InteractionPrimitives* class, except that the first argument is now an array containing the number of DOFs for the multiple agents in the work environment.

The ninth argument is used to specify the type of algorithm to be used for the multi-agent case. The type can either be *"ono"* or *"all"*, which represent the *one to one* and the *one to all* algorithms, resp

4.4.2 Data Fitting

This step is used to learn the weights of the demonstration data and the algorithm learns the weight in a least-squared sense as shown in the equation (3.3). The abstraction designed as a part of this library makes the data fitting step similar for both the scenarios.

```
data_dir_prefix = 'a path to the folder with data files'
```

```
file_prefix = 'data files must have and numbers following'
start_index = i # start index of numbering
num_demo = j # total number of demonstrations
ip.fit(data_dir_prefix, file_prefix, start_index, num_demo)
```

The above code snippet is a demonstration for the two-agent work environment and multiple-agent work environment scenario, with method specified as *all*.

The first argument to the *fit* function is an absolute or a relative path to the folder containing the data file.

The library expects a standard way of structuring the data files themselves and their naming inside the given folder. The data files must have a common prefix and they must be numbered continuously. For example, if the prefix of the files is *"jointAngles"* and there are three data files, they must be named as, *"jointAngles0.txt"*, *"jointAngles1.txt"*, and *"jointAngles2.txt"*. In this example, the second argument of the *fit* function will be a String literal *"jointAngles"*. Regarding the internal structure of the data files, each row of the data file should be a combined column vectors of the position/joint angles of all the DOFs of all the agents. In granular terms, a datafile must be structured as a *numpy* matrix so that it can be imported using the *loadtxt* function of *numpy*.

The third argument is the start index of data files, which in the current example will be 0. The fourth argument is the total number of data files, which in our current example will be 3.

For the multiple agent work environment with method specified as *ota*, the start and end indices of the combined weights of the agents must be specified in the data file. Also, the weights in the data files must be arranged linearly to one after another depending on how the interactions are taking place. For example, if there are 3 agents with 6 DOFs each, then the total number of columns present in the data file must be

18. Let's assume that these agents are $\{A_1, A_2, A_3\}$ where interacting pairs are (A_1, A_2) and (A_2, A_3). In that case, the first 6 columns should have the joint angles of the agent A_1, the next 6 columns should have the joint angles of A_2 and so on.

In the below code snippet, the *start_index_df* and *end_index_df* are used to specify the span of the indices of the columns for an interacting pair. This means that for (A_1, A_2), *start_index_df* will be 0 and *end_index_df* be 11.

```
data_dir_prefix = 'a path to the folder with data files'
file_prefix = 'data files must have and numbers following'
start_index = i # start index of numbering
num_demo = j # total number of demonstrations
start_index_df = k # start index of combined weights
end_index_df = l # end index of combined weights
ip.fit(data_dir_prefix, file_prefix, start_index, num_demo,
    start_index_df, end_index_df)
```

4.4.3 Clustering parameter

To determine the number of clusters, various examples in the test files can be referred to which are primarily based on K-Fold cross validation. If the number of clusters has to be calculated on the fly, the G-Means algorithm can be used in the following way:

```
ipc = InteractionPrimitivesClustering(ip.weights.T)
sig_level = s #value which represents the significant level or
    confidence. Higher value means higher confidence and less number of
    clusters.
K = ipc.learn_k_by_g_means(sig_level)
```

The mixture parameters can then be calculated using,

```
method = 'gmm' # the other allowed value is ppca
ip.calculate_mixture_parameters(number_of_cluster, method)
```

4.4.4 Prediction

The most important step is to perform the prediction as described in the original Mixture of Interaction Primitives framework. The software package has several examples on how to perform the predictions. In a nutshell, the following steps are required,

```
t = index #index of the current reading
test_data = ['2D array of observations']
phase = z # phase value for the current reading, if phase value is -1,
    the phase will be calculated using particle filter
ip.assign_cluster(test_data[0:t,:], phase) #if the phase value is -1,
    the programs calculates the phase based on measurements. Returns
    phase value

# next step is to calculate predicted trajectory till the phase that
    has already passed.
ip.predict_till_phase(test_data[0:t,:], phase, number_of_samples) #if
    the phase value is -1, the programs calculates the phase based on
    measurements. Return an array of joint_angles and phase value

#subsequently, the prediction can be performed at each time step.
reading = r # reading of the observed agent
ip.predict_at_phase(phase, reading) #if the phase value is -1, the
    programs calculates the phase based on measurements.
```

In the above code snippet, the first step performed is to perform a cluster assignment of the partial observation. This step must be done before prediction as it determines the cluster to use for performing predictions.

Since, the prediction may start at a phase value, τ_0 0, it becomes evident to move the controlled agent to match the partial observation of the observed agent. In order to achieve this task, the *predict_till_phase* function is used. This function takes an argument an array of joint angles of the observed agent, the current phase, and number of samples of data required to control the controlled agent. After this stage, predictions can take place at each time step using the *predict_at_phase* function.

4.4.5 Anomaly Detection

This library includes a safety feature which lets the controlled arm detect an anomaly. The library has a two functions for this safety measure, the first function is *learn_weights_for_tool* which takes as an argument a prefix of the directory where the data files are kept, prefix of the file name, an array of indices of the columns which contain tool positions, start index number for the data file and total number of demonstrations. This function learns weights for the trajectory of an object and returns mean and covariance of the distribution. The second function is the *test_anamoly* which takes as an argument, the current position, mean and covariance of the distribution, threshold value, and the current phase value. It either returns True or False.

```
directory_prefix = 'directory where data files are kept'
file_name_prefix = 'prefix_file_names'
indices = [0, 1, 2]
start_index = 0
total_demonstrations = 200

mu, cov = ip.learn_weights_for_tool(directory_prefix, file_name_prefix,
    indices, start_index, total_demonstrations)
```

```

# at each time step, an anomaly detection can be done
anomaly = ip.test_anamoly(current_position, mu, cov, threshold, phase)
```

Chapter 5

EXPERIMENTS

For testing this Python library, four simulated work environments were created on the V-Rep simulator tool. Out of these four work environments, three were created to the test this library in a two-agent work environment, and one of them was set up to test the multi-agent capabilities of this library. This library was also tested in a controlled laboratory environment. More details about these work environments and the architecture of the project will follow up in the later sections.

5.0.1 V Rep

V Rep is a software tool which is used to created simulated work environments or scenes. A simulated work environment can be set up on V Rep using many available robots, physical objects, infrastructure objects, people, and vehicles. A scene is customizable because of various choices of robots, choices of different types of physical objects; choice on how to mock a real-world environment and adjusting dynamical properties, which can include weight, gravity, light, etc., of a scene, are also customizable. V Rep supports *Lua* as an internal scripting language to add programming features to a scene. Most of the scenes in my experiments have been programmed using *Lua*. V Rep has an extensive API for its scripting language and can be used to control a scene and fetch information about a scene dynamically. V Rep has an inbuilt TCP(Transmission Control Protocol) server which is used to connect to V Rep over TCP using the supported programming language. Since this library has been developed entirely using Python, the software package has programs which connect to the V Rep using its Python API.

5.0.2 Kinect Version 2

Kinect is a motion sensor which can be used to capture depth as well as color images. In my experiment, I have used a Kinect version 2 sensor for skeleton tracking. To capture skeleton tracking data, I have used the J4K(Java for Kinect) library.

5.0.3 Simulated Work Environments

Programs in the Python scripting language were written to apply this Python library on different Simulated environments. An overview of the architecture of the testing programs follows in the next two sections. Python programs for these work environments have been programmed to collect training data at ~ 50 Hz. The choice of data collection rate is not due to any scientific or technical constraint, but this allowed me to keep the number of observations in a single trajectory at a decent number to allow faster computations.

Training Phase

During the collection of training data, Python programs for each of these simulated environments connect to the V Rep API and make the appropriate API calls to collect the joint angles of the DOFs of each agent. Since this framework is based on interaction learning and learning by demonstration, it requires a decent number of sample to perform accurate prediction. The minimum number of sample needed for testing is decided by how varied the task(s) are. An overview of the training phase can be seen in the figure number 5.1.

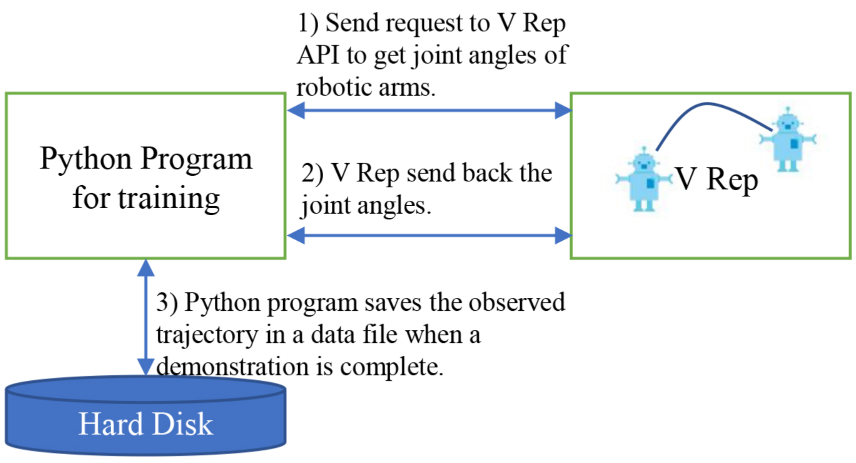

1) Send request to V Rep API to get joint angles of robotic arms.
Python Program for training
2) V Rep send back the joint angles.
V Rep
3) Python program saves the observed trajectory in a data file when a demonstration is complete.
Hard Disk

Figure 5.1: Overview Of Training Data Collection

Testing Phase

The testing phase is when this Python library is applied to predict trajectories of a controlled after partially observing trajectory of an observed agent. To perform prediction, data files are used to learn the interaction primitives. Once the primitives have been generated, they can be used to perform predictions at each time step. The figure number 5.2 and algorithm 5 will give an intuitive overview of the testing phase.

Work Environment 1

The first work environment is shown in figure number 5.3. This simulated work environment has two UR10 robotic arms mounted on two cubicle stands. The agent on the left-hand side is an observed agent and the agent on the right-hand side is a controlled agent. The interaction between these two agents is limited to handing over a tool on the rack on left by the observed agent to the controlled agent. Ultimately,

Algorithm 5 Testing predictions from the Interaction Primitives Python library

(i) Observe a partial trajectory of an observed agent till a phase value, $\tau = \tau_0$. The accuracy of cluster assignment improves as the value of τ_0 increases since more observation leads to more data and better cluster assignment.

(ii) Use the Interaction Primitives library to learn weights and cluster parameters.

(iii) Use the prediction capabilities of this library to predict the trajectory for the controlled agent up to the current phase value, $\tau = \tau_0$.

(iv) Now, at each time step, a single observation of the joint angles of the DOFs of observed is made and each time step, a predicted trajectory for the controlled agent is generated.

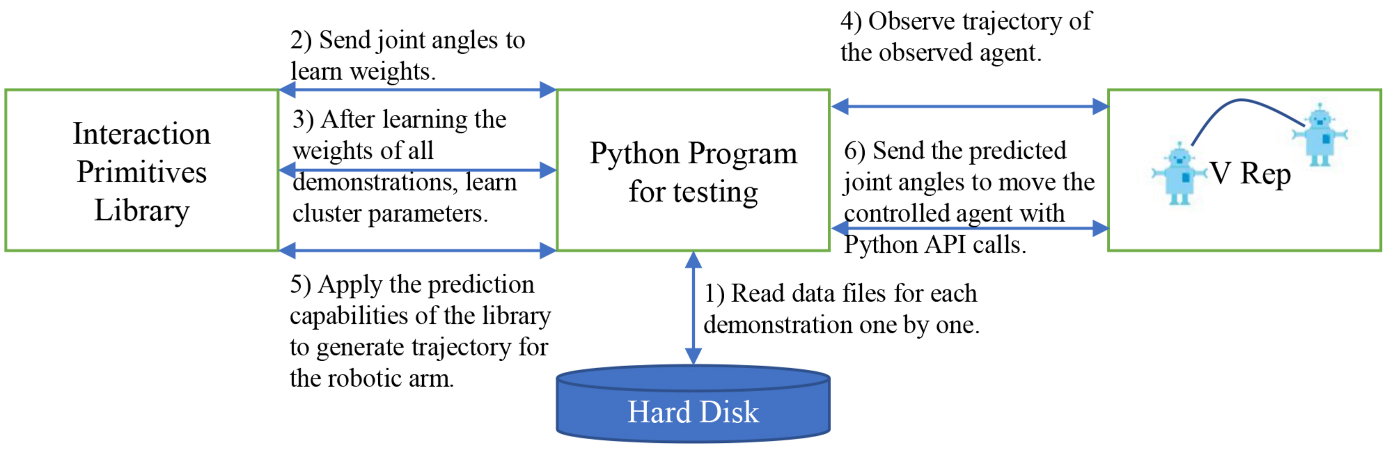

Interaction Primitives Library
2) Send joint angles to learn weights.
3) After learning the weights of all demonstrations, learn cluster parameters.
5) Apply the prediction capabilities of the library to generate trajectory for the robotic arm.
Python Program for testing
1) Read data files for each demonstration one by one.
Hard Disk
4) Observe trajectory of the observed agent.
6) Send the predicted joint angles to move the controlled agent with Python API calls.
V Rep

Figure 5.2: Overview Of Testing Predictions Of The Interaction Primitives Library

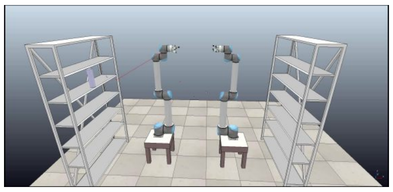
the controlled agent places this tool on the rack on right. This is an example of high variation in a single task. The demonstrations vary from each other in a way that the interaction point is randomly chosen from a uniform distribution of points inside a unit hemisphere. For the demonstrations, the joint angles of the 6 DOFs of both the UR10 robots are recorded. During the prediction phase, a partial trajectory comprising of joint angles of the observed agent is recorded and used for cluster assignment and conditioning.

Work Environment 2

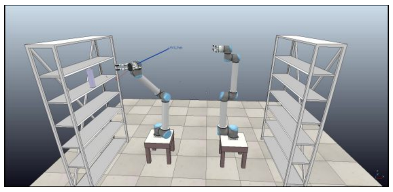
The first work environment is shown in figure number 5.4. This simulated work environment has two UR10 robotic arms mounted on two cubicle stands, and they are performing multiple tasks. The multiple tasks are defined by handing over multiple tools as shown in the simulated V Rep scene. The rack on the left has two tools, and the table on the left has a single tool kept on top of it. The agent on the right is the controlled agent, and the agent behind it on the left is the observed agent. Variation in the demonstrations is achieved by randomizing the interaction point. The interaction points for the three tasks are far apart, but for each of these tasks, training data is collected from interactions at uniformly random points inside a semi-unit sphere. Training and prediction data is same as for the work environment number 1.

Work Environment 3

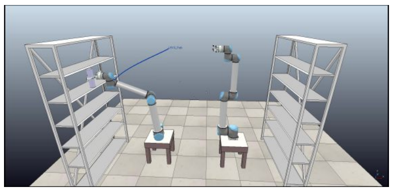
The purpose of the third work environment shown in figure number 5.5 is to test this library on a mobile robot. The simulated work environment has two YouBots. The YouBot on the backsides is the observed agent, which picks up a tool from the platform on the back and meets a random position with the controlled agent for an interaction. The controlled agent then picks up this tool from the platform of the



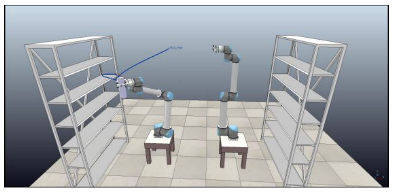
(a)



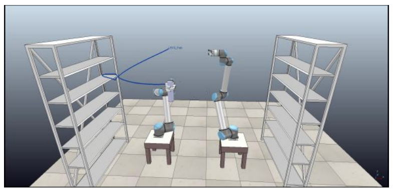
(b)



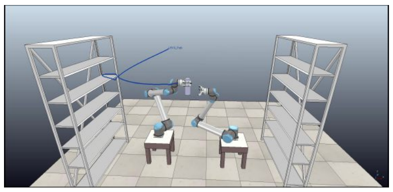
(c)



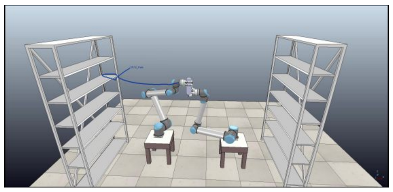
(d)



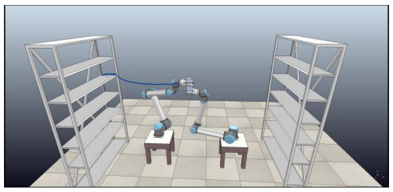
(e)



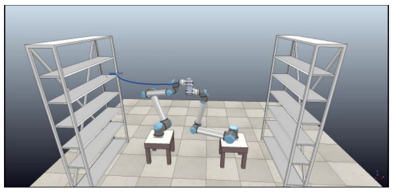
(f)



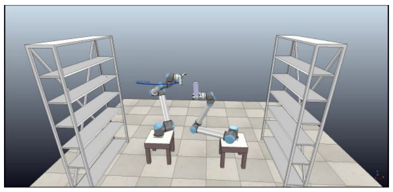
(g)



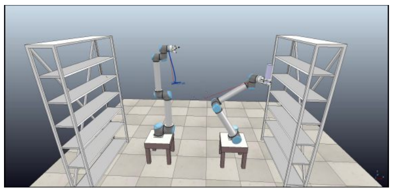
(h)



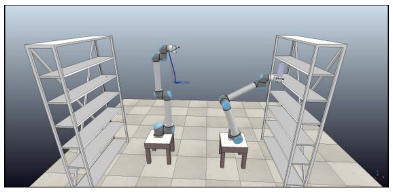
(i)



(j)

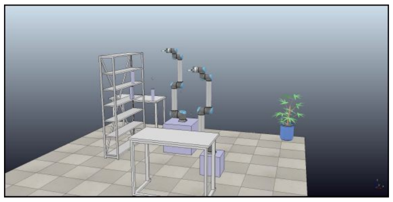


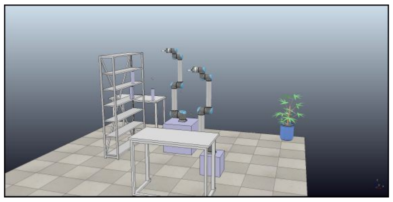
(k)

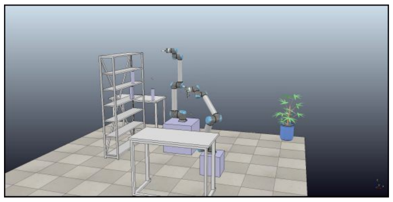


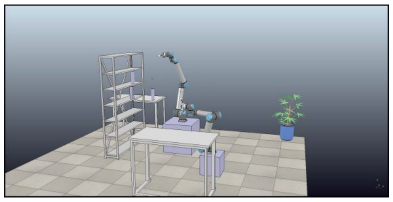
(l)

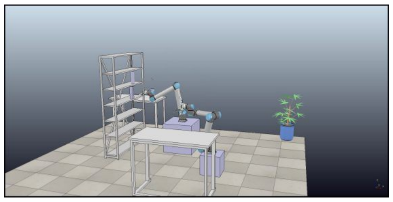
Figure 5.3: Work Environment 1

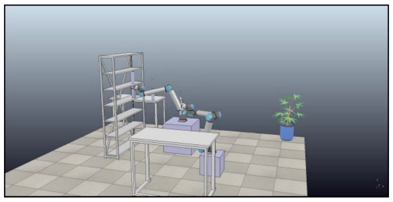

(a)

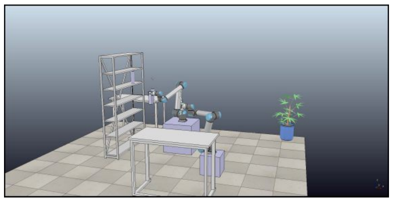

(b)

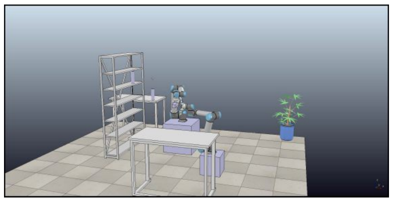

(c)

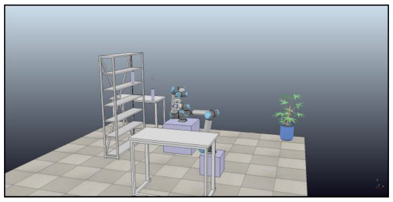

(d)

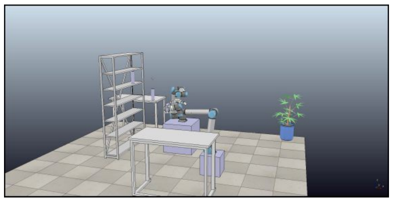

(e)

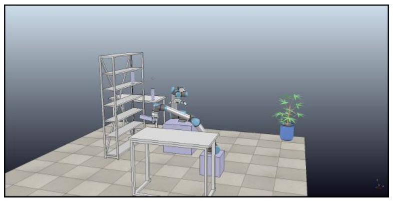

(f)


(g)


(h)


(i)


(j)


(k)

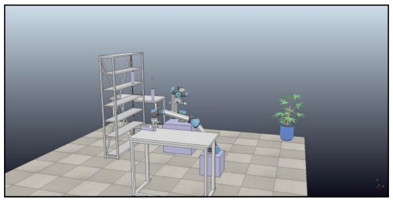

(l)

Figure 5.4: Work Environment 2

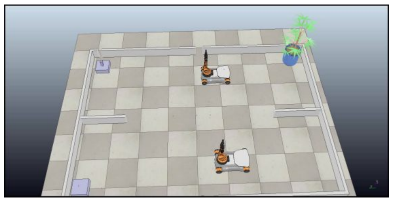
observed YouBot and moves towards the platform in front on the left and places the tool on top of the platform. Multiple demonstrations are captured by randomizing the interaction point. The interaction points are chosen at uniformly random points from a circle of radius 1.5 meters. During training, joint angles of the arms of both the YouBots are captured, the state of the gripper for both the YouBots are captured. The movement of the YouBots is controlled through motion planning. The destinations of the YouBots are also captured at each time steps.

Work Environment 4

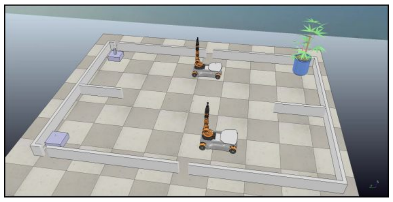
The fourth work environment is shown in figure number 5.6. This simulated work environment was created to test the library in a multi-agent work environment. The goal of this work environment is to mock a bucket brigade. Instead of a bucket, this simulated scene has several YouBots working in collaboration to pick up a tool from the platform on the left and ultimately place it on the top of the platform on the right. Interactions take place in a way that one YouBot places the tool on the top of its platform and the other YouBot picks up the tool and then places the tool on its own platform. The next YouBot then picks up the tool and chain continues till the last robot places the tool at the destination. During the training phase, multiple demonstrations are created by randomizing the location of the platform on the left. This leads chain forming at different locations in each demonstration. As for the training data, the joint angles of all the DOFs of a YouBot's arm, it's gripper state and the location of its destination is captured at each time step.

5.0.4 Human Robot Environment

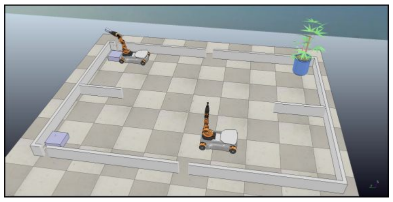
This controlled laboratory environment was setup to test the Interaction Primitives library in a real-world environment. In this setup, a human subject performs a



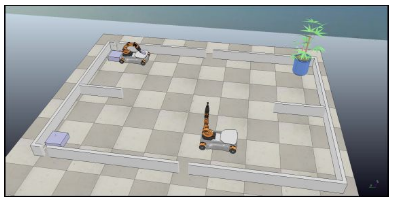
(a)



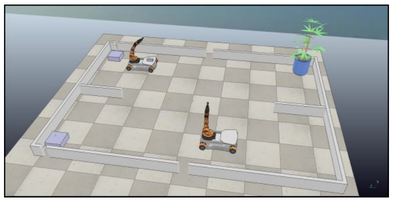
(b)



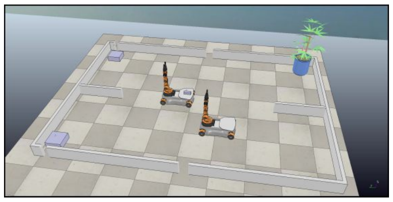
(c)



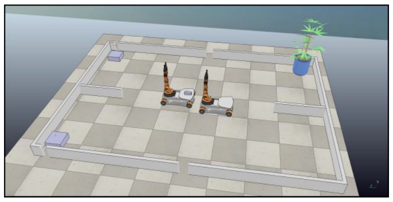
(d)



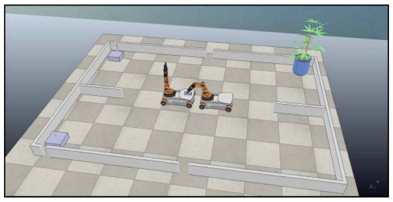
(e)



(f)

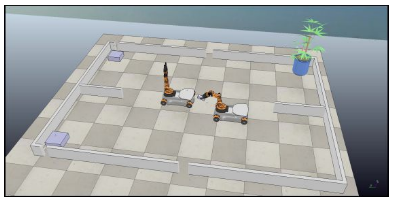


(g)

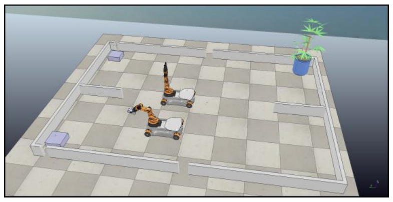


(h)

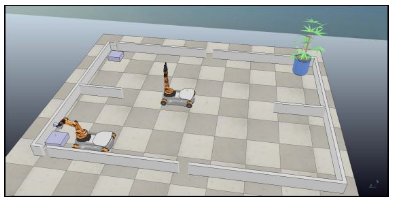
(i)



(j)

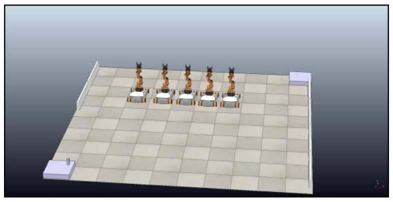


(k)

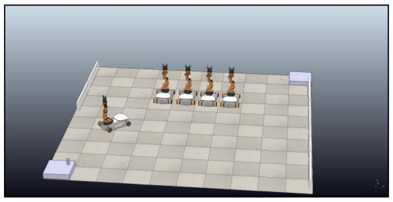


(l)

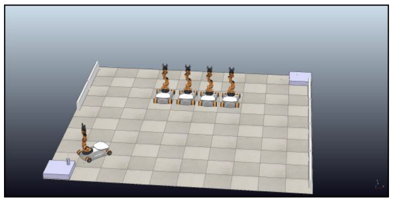
Figure 5.5: Work Environment 3



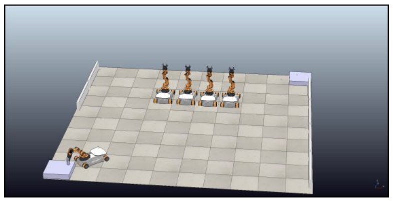
(a)



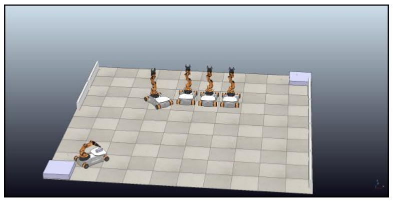
(b)



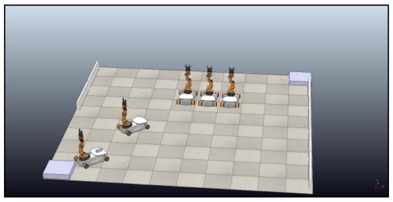
(c)



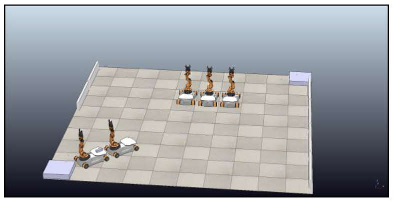
(d)



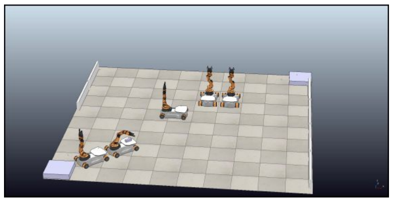
(e)



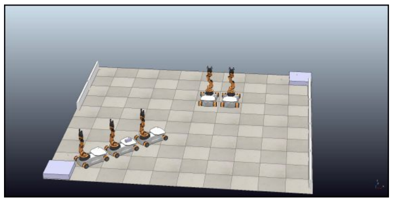
(f)



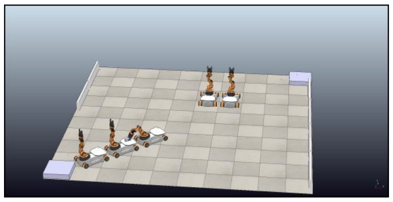
(g)



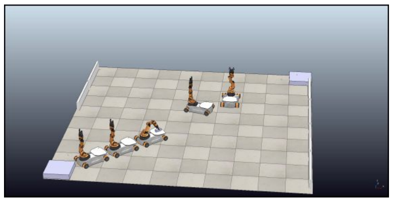
(h)



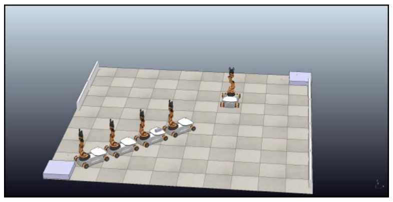
(i)



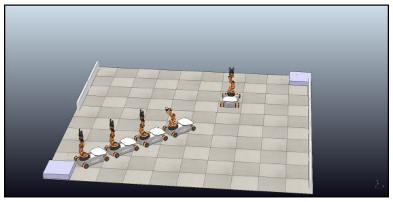
(j)



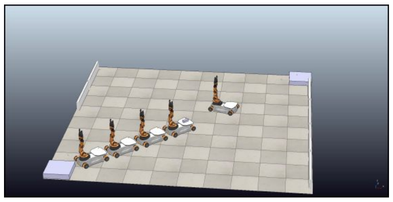
(k)



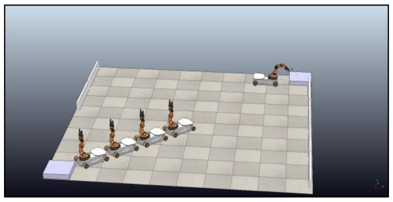
(l)



(m)



(n)



(o)

Figure 5.6: Work Environment 4

high-five with a UR5 robot. The joint angles of the UR5 robot is captured by communicating with its TCP API. The movement of the human subject is captured using a Kinect Version 2 sensor which captures the skeleton data of the subject. One of the challenges faced for capturing the skeleton data is an absence of a decent Python library that works with Kinect Version 2. In order to overcome this challenge, a Java Project was developed which uses the J4K(Java for Kinect) library to capture the skeleton data and sends position data of the human right hand on a UDP port. The J4K library is developed to capture the data at 30Hz and that's the speed of the Java UDP server. With the presence of a TCP server which sends the UR5 movement data and a Java UDP server which sends the movement data of a human hand, Python scripts were written to collect training data and test the library by controlling the UR5 arm on observations of the human subject. The training data was recorded at 30Hz due to a bottleneck of the J4K library and the testing is also performed at 30Hz.

Architecture

The figures 5.7 and 5.8 show overall architecture of the human-robot experiments. Description of the stages are specified in the respective figures.

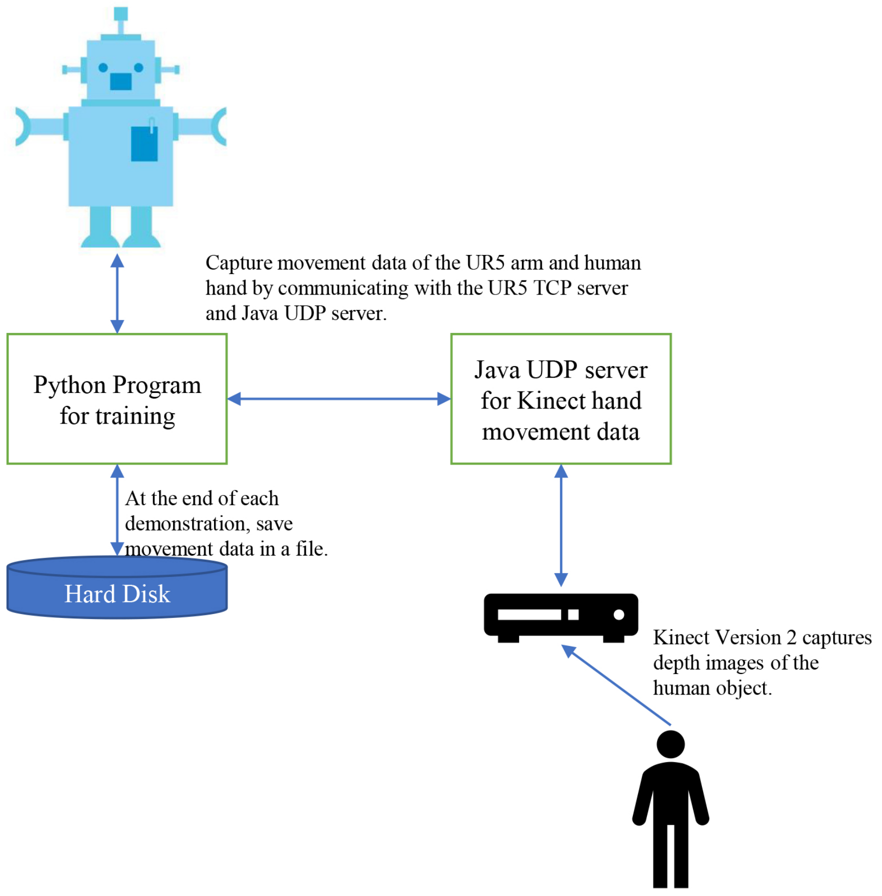

Capture movement data of the UR5 arm and human hand by communicating with the UR5 TCP server and Java UDP server.
Python Program for training
Java UDP server for Kinect hand movement data
At the end of each demonstration, save movement data in a file.
Hard Disk
Kinect Version 2 captures depth images of the human object.

Figure 5.7: Overview of Human Robot Experiment - Training

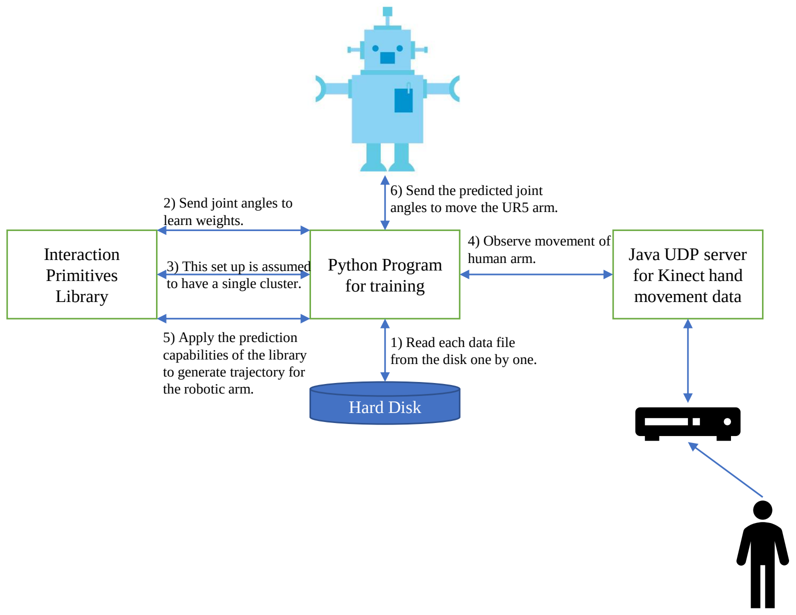

6) Send the predicted joint angles to move the UR5 arm.
2) Send joint angles to learn weights.
Interaction Primitives Library
3) This set up is assumed to have a single cluster.
Python Program for training
4) Observe movement of human arm.
Java UDP server for Kinect hand movement data
5) Apply the prediction capabilities of the library to generate trajectory for the robotic arm.
1) Read each data file from the disk one by one.
Hard Disk

Figure 5.8: Overview of Human Robot Experiment - Testing

Chapter 6

RESULTS AND BENCHMARKS

6.1 Benchmarks

The benchmarks laid down for the Python library so that it is well suited to be used in real world scenarios with multiple robots and multiple sensors are as follows:

(a) This library should have a performance to enable posterior prediction within 20 Hz.

(b) This library needs to be accurate in the predictions.

(c) This library should give accurate results on small training data.

(d) This library needs to be efficient.

6.2 Tests Performed

The following tests were done to check if the benchmarks were meet and the behavior of the library for other scenarios mentioned below:

(a) The library can meet the benchmark of 20 Hz.

(b) Euclidean distance between the predicted and the actual trajectories.

(c) Success rate of tool pickup.

(d) Performance vs number of clusters.

(e) Accuracy vs number of clusters.

(f) Accuracy vs number of training data.

6.3 Results

(a) The library met the benchmarks for frequency. While interacting with VRep, the library could meet at frequency of approximately 19Hz on an average. Whereas, when not interacting with VRep, the library attained a frequency of 3554 Hz on an average. Table 6.1 shows a detailed analysis.

Table 6.1: Frequency Comparison.

	With VRep	Without VRep
Average	19.4Hz	3553.5Hz
Median	18.27Hz	352.4Hz

(b) Different plots for the variation of Euclidean distances can be seen in figures 6.1, 6.2 and 6.3. These diagrams show how the euclidean distance between the predicted trajectory and actual trajectory varied by varying the number of clusters, number of gaussians and number of training samples, respectively.

(c) Success rate was determined by calculating the percentage of successful tool handover over all the testing data. The overall success rate was 51%. Changes in the value of success rate by varying the number of clusters, number of gaussians and number of training samples, respectively, can be seen in figures 6.4, 6.5 and 6.6.

(d) One of a computationally intensive step in the overall flow of the library is the cluster assignment step. This step was chosen as an indicator of the performance of the library. The way the performance varies with change in the number of clusters and the number of gaussians, can be seen in figures 6.7 and 6.8, respectively.

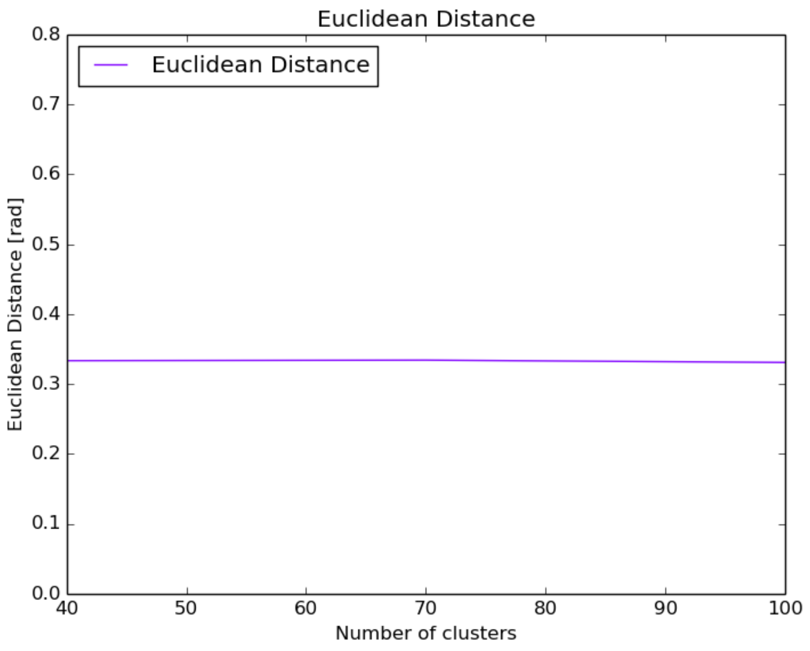


Figure 6.1: Euclidean Distance Vs Number Of Clusters

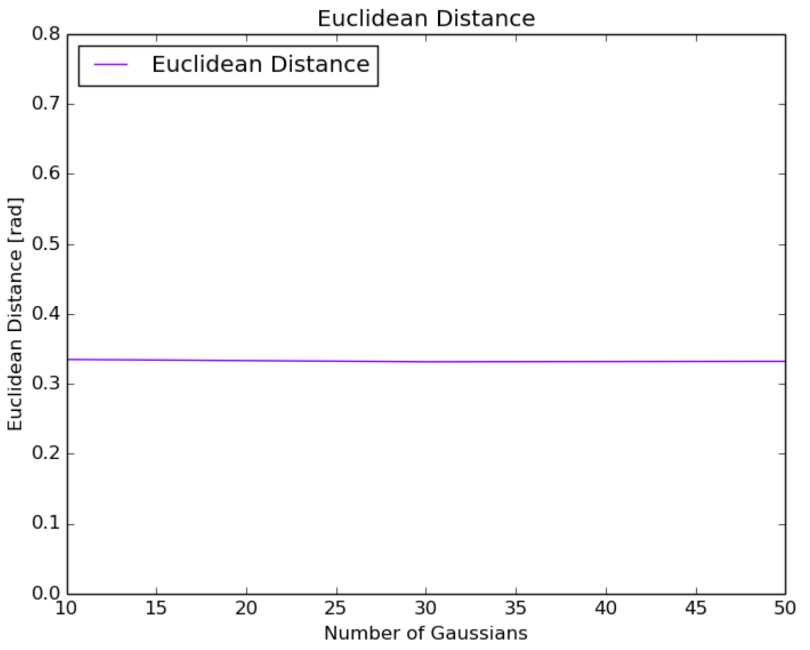


Figure 6.2: Euclidean Distance Vs Number Of Gaussians

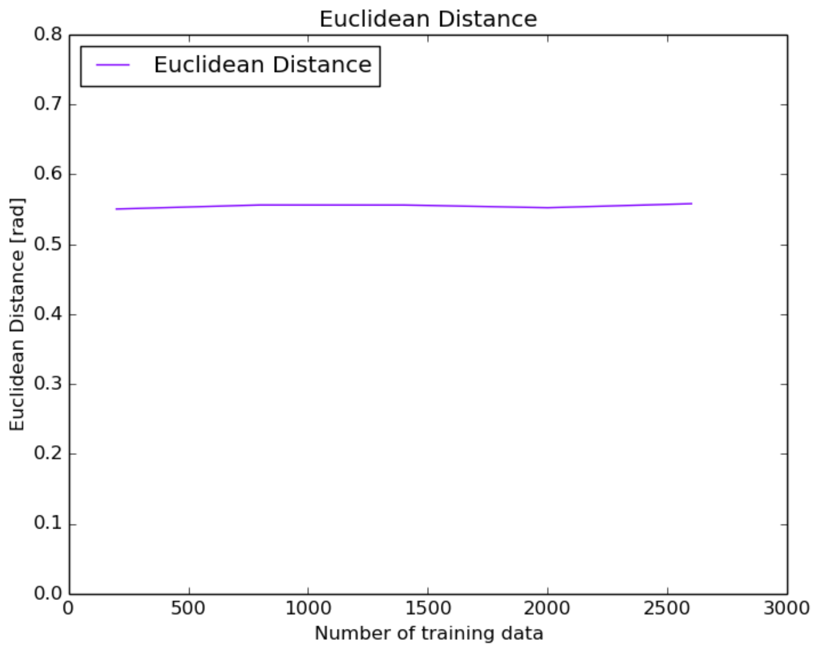


Figure 6.3: Euclidean Distance Vs Number Of Training Samples

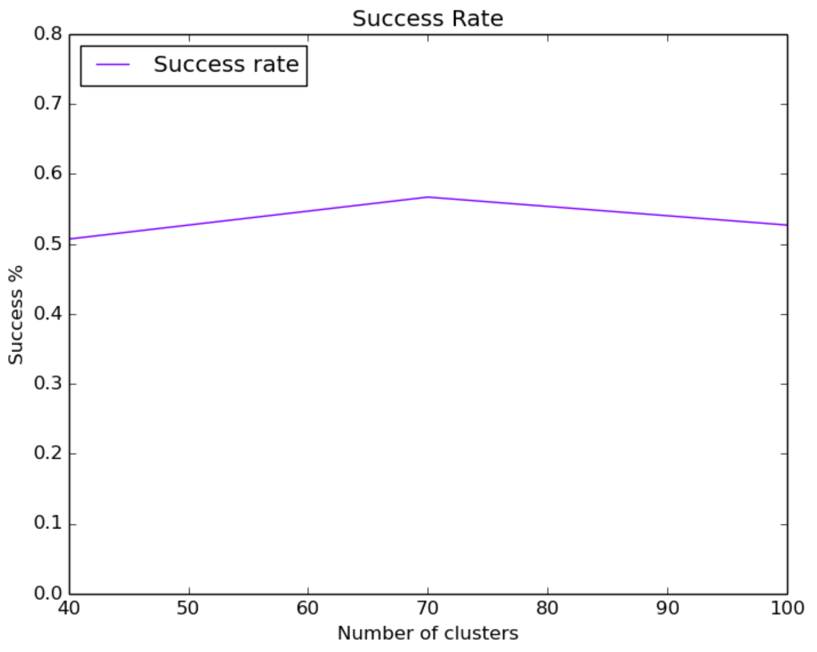


Figure 6.4: Success Ratio/Accuracy Vs Number Of Clusters

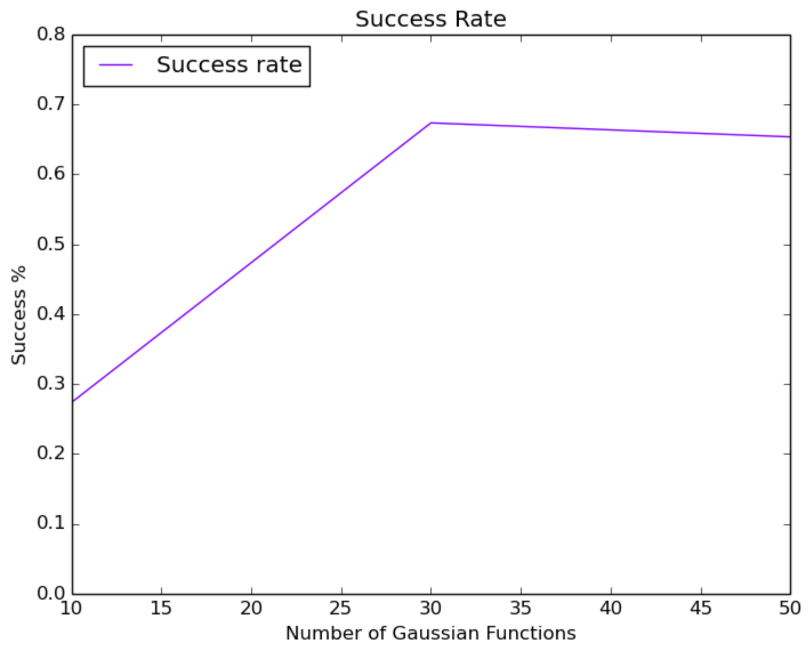


Figure 6.5: Success Ratio/Accuracy Vs Number Of Gaussians

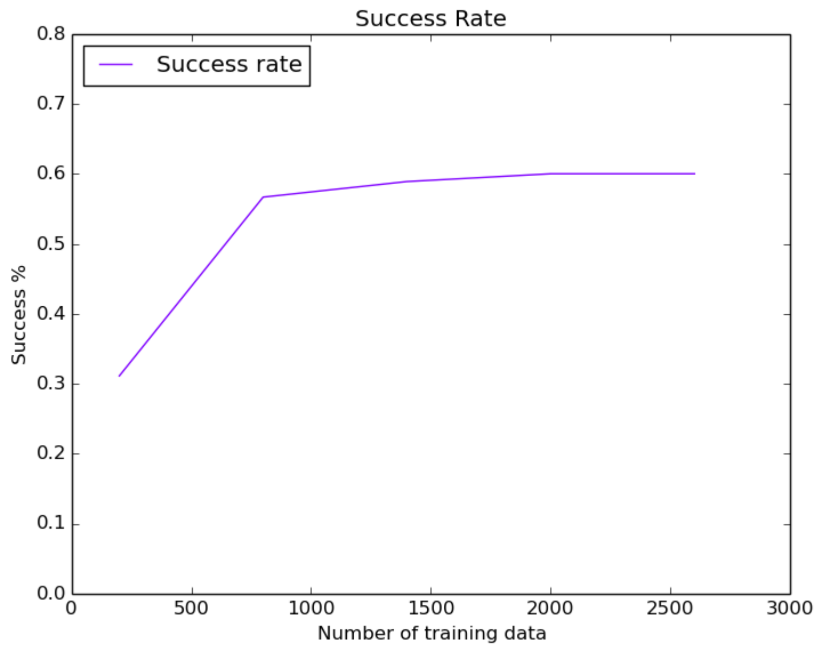


Figure 6.6: Success Ratio/Accuracy Vs Number Of Training Samples

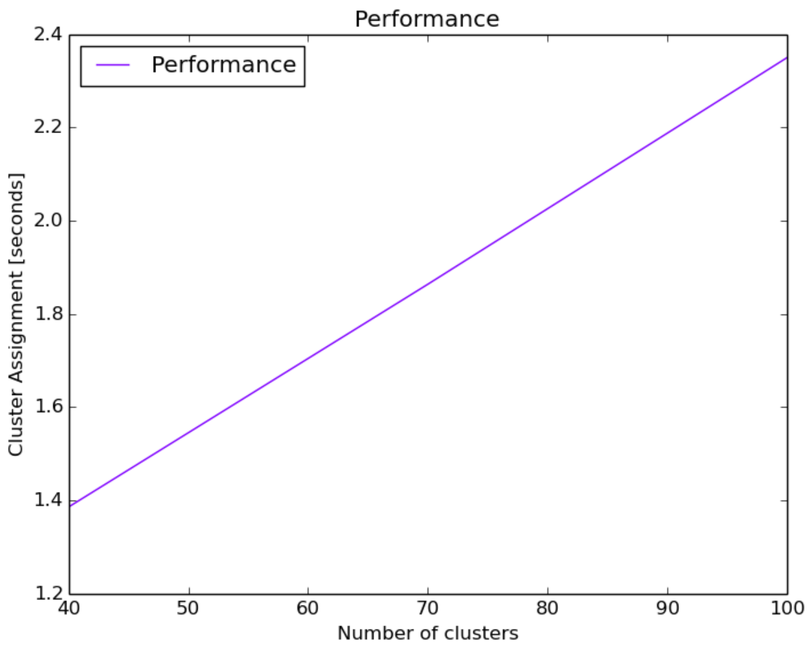


Figure 6.7: Performance Vs Number Of Clusters

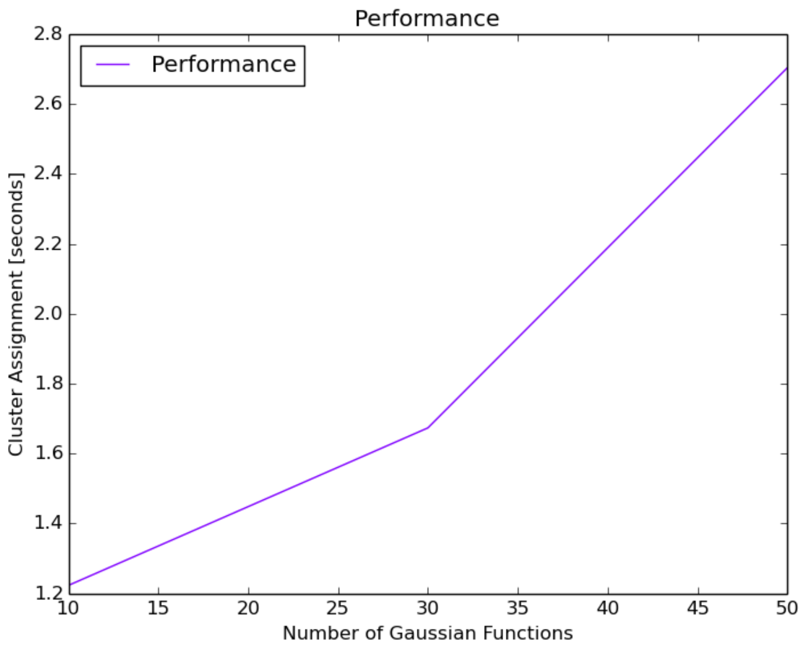


Figure 6.8: Performance Vs Number Of Gaussians

Chapter 7

CONCLUSION

The objective of my thesis work is to develop a state of the art Python library which is based on the Interaction Primitives framework and achieves the benchmarks laid down for the library. This library can be used in both a two-agent and a multi-agent setup and shows good results for a single task or multi task set up in a two agent work environment, and a single task set up in a multi-agent work environment.

My experiments and the library documentation documents show this library is easy to use and easy to be deployed and has been developed on tight guideline laid down in the original research work on the theory of Interaction Primitives. To achieve better results with the framework, there are some considerations which can be made while working in a new setup.

1. Number of training sample: Although this library gives good results with a small number of training data, the accuracy increases with larger training data. This has been shown in the results.

2. Number of clusters: Carefully choosing the number of clusters is important for better results. This parameter should not be very small, which may lead to different tasks being grouped in a common cluster, and should not be too large which could make computation slower.

3. Examples: The software package has several test files, which can be used as a starting point before starting a new project. Some examples are also available in the Appendix section.

Chapter 8

FUTURE WORK

This state of the art Python library has desirable qualities and has excellent benchmarks for a two-agent multi-task work environment and a multi-agent single-task work environment. This library has been tested on both real-world as well as simulated experiments. Even with good results, this library has a scope for further research and work.

1. Phase Estimation: The Interaction Primitives library uses the theory behind Particle Filtering for phase estimation. Although this library shows good results for a two-agent setup, the phase estimation is not accurate in multi-agent multi-task setup. Also, as the number of particles is increased, it makes the computation slower. A framework which performs phase estimation as a part of the Interaction Primitives conditioning is highly desirable.

2. Cluster Parametric Estimation and Assignment: Determining the best value for the number of clusters (considering computation) was one of the challenges. This framework suggests two possible ways for the number of cluster determination. Research can still be done in finding a framework to determine the best number of clusters.

3. Collision Detection: This framework has been developed for a stochastic work environment. Although the framework has an anomaly detection feature, it still lacks a collision detection feature. Collision detection will have to be externally programmed if required.

REFERENCES

Alexandros Paraschos, J. P., Christian Daniel and G. Neumann, “Probabilistic movement primitives”, in “Advances in Neural Information Processing Systems 26 (NIPS 2013)”, (2013).

Bartholomew, D. J., “Latent variable models and factor analysis”, (1987).

Guilherme Maeda, R. L. H. B. A. J. P. G. N., Marco Ewerton, “Learning interaction for collaborative tasks with probabilistic movement primitives”, in “Proceedings of the International Conference on Humanoid Robots (HUMANOIDS)”, (2014).

Hamerly, G. and C. Elkan, “Learning the k in k-means”, (1999).

Heni Ben Amor, S. K. O. K. J. P., Gerhard Neumann, “Interaction primitives for human-robot cooperation tasks”, in “Proceedings of 2014 IEEE International Conference on Robotics and Automation (ICRA)”, (2014).

Marco Ewerton, R. L. H. B. A. J. P., Gerhard Neumann and G. Maeda, “Learning multiple collaborative tasks with a mixture of interaction primitives”, in “Proceedings of 2015 IEEE International Conference on Robotics and Automation (ICRA)”, (2015).

Tipping, M. E. and C. M. Bishop, “Mixtures of probabilistic principal component analysers”, (1999).

Turner, L., “An introduction to particle filtering”, (2013).