Internship Report

A.A.Bhole (s1841467)

MSc. Systems and Control

UT Supervisor

Dr.Ir. Theo De Vries

Faculty of Electrical Engineering, Mathematics and Computer Science

University of Twente

Internship Supervisor

Dr. Luigi Villani

Department of Electrical Engineering and Information Technologies

University of Naples Federico II

Internship Advisor

Dr. Fanny Ficuciello

Department of Electrical Engineering and Information Technologies

University of Naples Federico II

Acknowledgements

This report marks the end of my five month internship program at **PRISMA Lab, University of Naples Federico II**, undertaken as a part of curriculum for MSc. Systems and Control at **the University of Twente**.

I got an opportunity to work on two different problem statements during my internship. The report is thus divided into two parts. The first part presents **An energy-based control approach to realize a variable impedance control task on a robotic manipulator** and the second part presents results on **Estimation of impedance parameters of a variable impedance controlled robotic manipulator.**

I would like to take this opportunity to express profound gratitude and deep regards to **Raffaella Carloni** for helping me to land on this great opportunity for an internship. **Fanny Ficuciello, Salvatore Strano and Luigi Villani** for their exemplary guidance, monitoring and constant encouragement throughout the course of this project. The environment created at the **PRISMA Lab** at University of Naples Federico II is probably the best a student interested in robotics could ever hope for.

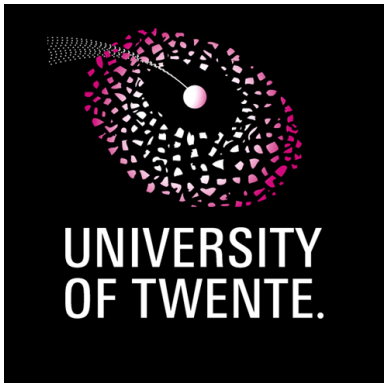
Working on this internship assignment was a challenging and an interesting adventure and I enjoyed having great colleagues and friends, **Jonathan Cacace, Fabio Vigoriti and Fabio Ruggiero** who were always there to help me with my work.

Keeping up the gradient!
Ajinkya Bhole

Energy based approach for Variable Impedance Control of Robotic Manipulators

By

Ajinkya Arun Bhole



Faculty of Electrical Engineering, Mathematics and
Computer Science (EEMCS)
University of Twente, The Netherlands

This report is submitted to University of Twente in accordance with the requirements of the degree of MSc Systems and Control.

January 2018

Energy based approach for Variable Impedance Control of Robotic Manipulators

Abstract

This work presents an energy-based control approach to realize a variable impedance control task on a robotic manipulator. Introducing an energy tank into the system and aptly regulating its dynamics, it is made certain that the overall system i.e. the manipulator and the environment remains passive, thus ensuring a stable interaction.
The presented method can be used to impose offline as well as online varying impedance profiles for a robotic manipulator. For example varying the impedance profile for a needle insertion task during a surgery, depending on the varying features of a tissue being operated.
Unlike other similar works which use tank-based approaches for variable impedance control, our method does not induce any chattering effects in the realized impedance and also avoids the singularity problem, in case the tank gets empty. Our method provides a smooth transition of the realized impedance to that of the desired one and tries to keep the realized impedance as close as possible to the desired impedance. To demonstrate this, simulations and experimental studies were performed on a 7-DOF KUKA LWR.
This work was performed in continuation of the work performed by Denny Geremia [9].

Contents

List of Figures

Chapter 1

Introduction

Humans can perform skillful movements like like sewing, hammering, catching, running, etc, which requires interacting with partially or completely unknown environment. Humans achieve this by adjusting the dynamic characteristics of their musculo-skeletal system. Taking this strategy as an inspiration, roboticists are trying to integrate such skillful capabilities into robotic manipulators. Such a goal can be advantageous for tasks like invasive surgery wherein one can overcome surgeon's error and thus improve surgery quality. The most commonly used surgical robotic systems like the Da Vinci or DLR Micro Surge (see Figure 1.1) are teleoperated by surgeon. One major disadvantage of teleoperated robots is the surgeon's need to perform the task manually and also the significant training effort that goes into this. This can increase the workload and the fatigue and consequently, decrease the concentration during the operation. For these reasons, a major goal of the robotics community is to make it a possibility to perform automatically some parts of the surgery by the robot.

A significant challenge in automating a surgery lies in interacting with partially or completely unknown environment. This brings in unknown dynamics into the system requiring the manipulator to change its interactive behavior online as the operating conditions may change: like a surgeon who needs to adapt stiffness of his/her arm during a needle insertion task depending on the characteristics of the tissue being operated. The surgical field is just one of the many domains in robotics requiring adaptation of interactive behavior of the manipulator with the environment. In his famous trilogy, Neville Hogan [10] has addressed the task of interaction

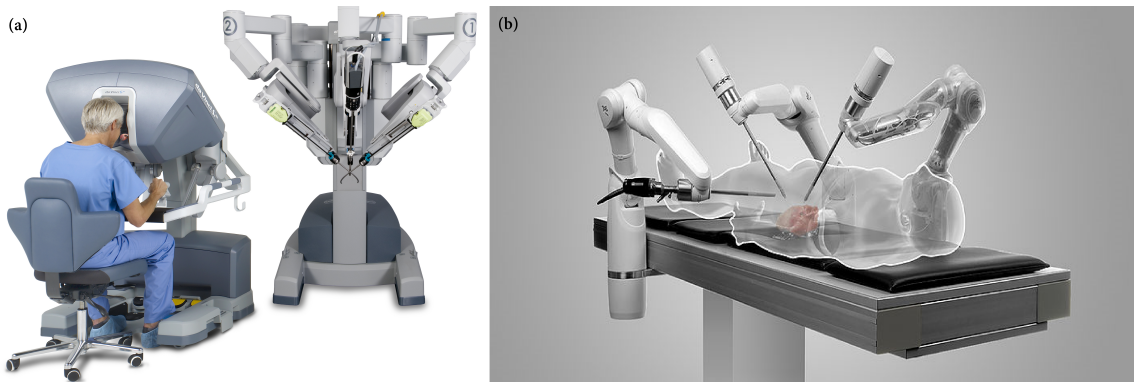


Figure 1.1: (a) Da Vinci Surgical System and (b) DLR Micro Surgical Robot

of robotic systems with environment by introducing the concept of Impedance control. The strategy allows the robotic systems to modify and shape its interaction behavior with the environment. This is a widely adapted paradigm when it comes to interacting with unknown environments using robots. Taking this idea forward, many researchers have also dived into the area of variable impedance control [1, 12, 2, 6, 5].

Variable impedance control provides an increased flexibility and performance in the task over fixed impedance controlled systems, but also comes with a few caveats. Varying the impedance parameters can land the system into instability as the system no longer remains passive [12]. Hence, the topic of ensuring stability in a variable impedance control task has gained a lot of attention recently. Very few works have addressed the issue of guaranteeing stability during a variable impedance task [12, 5, 3]. Klas and Aude [12] have provided an elegant solution to this problem, but, their method can only be used in case the variable impedance profiles are known a-priori. However, there exist numerous tasks in which there is an absence of pre-planned impedance profiles and thus an online variation is required. A surgery task as mentioned earlier is one of them.

In this work, our main focus is present a method which can be used for an offline as well as online variable impedance task. Based on energy interplay in the system, the tank-based approach first introduced in [4] is a widely used approach to tackle such a problem at hand [8, 14, 7, 15]. The performance of this method depends largely on the formulation of the dynamics of the energy tank and the adaptation of the impedance profile when the tank gets empty. The real impedance profile needs to be deviated from the desired impedance profile in case the tank gets empty so as to assure passivity of the overall system. The main contribution of this work lies in the formulation of the dynamics of this tank and the impedance profile adaptation strategy in case the tank gets empty. We show using simulations and experiments, that as compared to previously introduced formulations [5, 3], our method provides a smooth transition of the real impedance profile to the desired impedance profile when the tank gets empty. As a result of this, a betterment is also seen in the end-effector trajectory.

The remainder of the report is organized as follows. Chapter 2 briefly introduces the background on Cartesian Impedance Control. Chapter 3 performs a stability analysis of a variable impedance model. Chapter 4 and 5 introduce various existing methods and our proposed method which guarantee stability during a variable impedance control task. Chapter 6 presents experimental evaluations of the existing methods and our proposed method. Finally, concluding remarks are drawn in Chapter 7.

Chapter 2

Cartesian Impedance Control

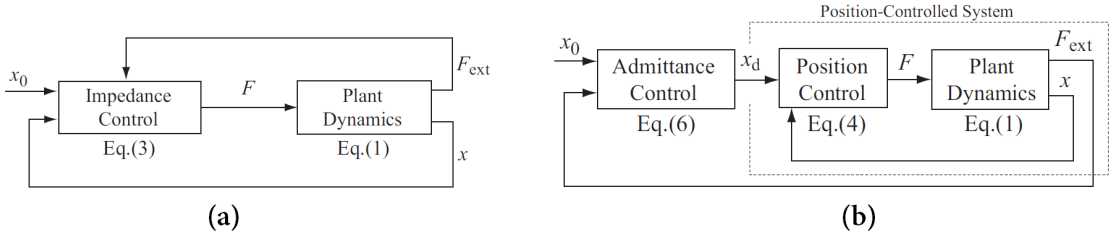


Figure 2.1: (a) Impedance Control Scheme and (b) Admittance Control Scheme [13]

Many applications in robotics require a robotic manipulator to interact with the environment. This interaction changes the overall dynamics of the system and usually this change is partially or completely unknown. Thus, it makes no sense to control either the force or the position of the end-effector, instead, a dynamic relationship can be set up between the motion of the manipulator and force applied by the environment. This is the core idea behind impedance control [10]. Thus, the control objective for impedance control is to design the control action F_c that will establish a given relationship between the external force F_{ext} and the error $e = (x - x_d)$ from a desired trajectory x_d.
Depending on the causality of the controller, there are two ways to implement impedance control: "Impedance Control" and "Admittance Control". Figure 2.1 shows the implementation of the two different schemes. The details on stability and performance of these schemes are well documented in [13]. In this work we have implemented the Admittance Control scheme, although as is usually done in literature, we will refer this to as impedance control in the remainder paper. A background of the Impedance control for a robotic manipulator has been presented in [6], but for the sake of completeness and readability we mention it in this section.
The Euler-Lagrange model of a n-DOF fully actuated robotic manipulator has the following form:

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) + \tau_f = \tau_c + J^T(q)F_{ext} \tag{2.1}$$

where $q \in R^n$, $M(q)$ is the inertia matrix, $C(q,\dot{q})$ is the vector of Coriolis/centrifugal torques, $g(q)$ is the vector of gravitational torques, τ_f is the vector of friction torques, τ_c is the vector of control torques, $J(q)$ is the robot Jacobian, $J^T(q)F_{ext}$ is the joint torque resulting from external force on the end-effector.

In order to design the impedance control, it is useful to derive the end effector dynamics in the operational space, considering only the translational motion:

$$\Lambda(q)\ddot{x} + \mu(q,\dot{q})\dot{x} + F_g(q) + F_f(q) = F_c + F_{ext} \tag{2.2}$$

where $x \in R^3$ is the Cartesian position vector of the end effector, $\Lambda = (JM^{-1}J^T)^{-1}$ is the (3×3) end effector inertia matrix, hereafter denoted as apparent inertia, while $\mu\dot{x} = \Lambda(JM^{-1}C - \dot{J})\dot{q}, F_g = J^{\dagger T}g, F_f = J^{\dagger T}\tau_f$ and $F_c = J^{\dagger T}\tau_c$ are the forces, reflected at the end effector, corresponding to the noninertial joint torques in equation 2.1.
Equation 2.2 describes only the end effector dynamics and does not include the so-called null space dynamics. Matrix $J^\dagger$ is the dynamically consistent generalized inverse of matrix J, defined as [11]

$$J^\dagger = M^{-1}J^T[JM^{-1}J^T]^{-1} \tag{2.3}$$

The goal of the impedance controller is to have a non-linear feedback law which translates the manipulator into an equivalent mass-damper-spring system. Let $x_d(t) \in R^3$ be a desired configuration for the end-effector. The relationship between $\tilde{x}(t) = x(t) - x_d(t)$ and F_{ext} is given by:

$$\Lambda_d(t)\ddot{\tilde{x}} + D_d(t)\dot{\tilde{x}} + K_d(t)\tilde{x} = F_{ext} \tag{2.4}$$

where $K_d(t), D_d(t) and M_d(t)$ are, respectively, the desired stiffness, damping and inertia matrices.

The above dynamics can be imposed to the closed loop controlled system by choosing F_c in as follows:

$$F_c = \eta(q,\dot{q}) + \Lambda(q)\Lambda_d^{-1}(D_d\dot{\tilde{x}} + K_d\tilde{x}) + (\Lambda(q)\Lambda_d^{-1} - I)F_{ext} \tag{2.5}$$

with $\eta(q,\dot{q}) = \mu(q,\dot{q})\dot{x} + F_g(q)$. In order to achieve (), it is necessary to feedback the external force F_{ext}, which can be measured by using a force/torque sensor mounted at the end-effector, or alternatively, force estimation techniques can be adopted. If the apparent inertia of the end effector is left unchanged, i.e., $\Lambda_d = \Lambda(x)$, the control law does not depend on the external force F_{ext}.

Chapter 3

Stability Analysis

Consider the following impedance model:

$$\Lambda_d \ddot{\tilde{x}} + D_d \dot{\tilde{x}} + K_d(t)\tilde{x} = F_{ext} \tag{3.1}$$

where, the desired stiffness varies with time. To analyze the stability properties of equation 3.1 , consider the following Lyaponov candidate function:

$$L_1(\tilde{x}, \dot{\tilde{x}}) = \frac{1}{2}\dot{\tilde{x}}^T \Lambda_d \dot{\tilde{x}} + \frac{1}{2}\tilde{x}^T K_d(t)\tilde{x} \tag{3.2}$$

Differentiating along the trajectories of 3.2 with $F_{ext} = 0$, we have:

$$\dot{L_1} = \left[\frac{1}{2}\tilde{x}^T \dot{(K_d)}(t)\tilde{x} - \dot{\tilde{x}}^T D_d \dot{\tilde{x}}\right] \tag{3.3}$$

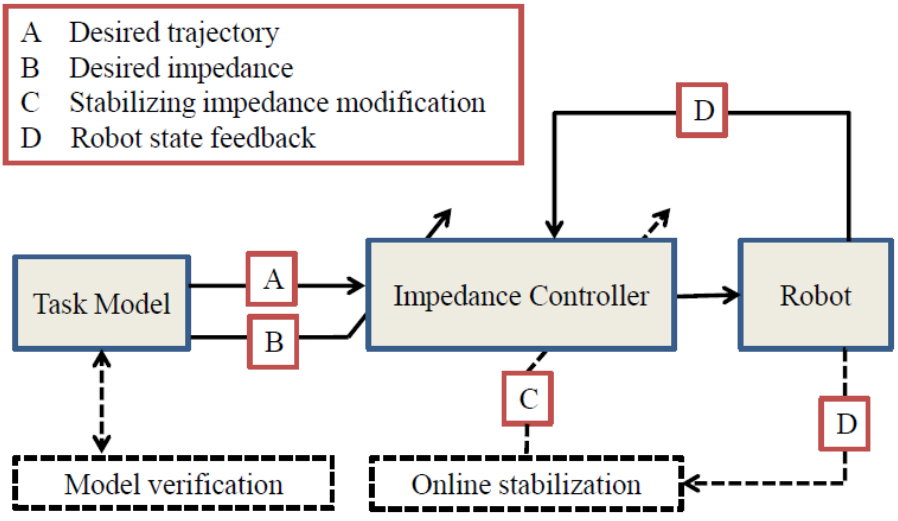


Figure 3.1: Approaches to guarantee stability in case of variable impedance task [12]

It can be seen that increasing stiffness eigenvalues injects potential energy into the system and can possibly lead to unstable behavior. Only a few have worked on this problem of tackling instability. Figure 3.1 shows the general approaches to guarantee stability. Model Verification: Klas [12] tackle this problem by introducing

a less conservative Lyapunov function than 3.2 , increasing the domain of attraction and allowing to include some positive stiffness variations although with specific constraints. We call this **Lyapunov Method** in the remainder of the paper. Online Method: Other elegant approach uses the theory of energy tanks to ensure passivity of the overall system and thus stability [5, 3]. These will be referred as **Tank Methods** in the remainder paper. All the above-mentioned methods are presented in the following chapters.

Chapter 4

Lyapunov Method

Taking inspiration from practices in adaptive control, Klas [12] define their Lyapunov function using weighted sum of velocity and position error of the state as follows:

$$L_2(\tilde{x}, \dot{\tilde{x}}) = \frac{1}{2}(\dot{\tilde{x}} + \alpha\tilde{x})^T \Lambda_d (\dot{\tilde{x}} + \alpha\tilde{x}) + \frac{1}{2}\tilde{x}^T \beta(t)\tilde{x} \tag{4.1}$$

where,

$$\beta(t) = K_d(t) + \alpha D_d(t) - \alpha^2 \Lambda_d \tag{4.2}$$

with some positive constant α chosen such that $\beta(t)$ is positive semidefinite for all $t > 0$. **Note that this method does not allow time varying inertia matrix** . Defining this Lyapunov function allows to establish sufficient constraints on stability which is formalized as follows:

Let Λ_d be a constant, symmetric and positive definite matrix. Let $K_d(t)$ and $D_d(t)$ be symmetric, positive definite and continuously differentiable varying stiffness and damping profiles. Then, the system in equation 3.1 with $F_{ext} = 0$ is globally uniformly stable if there exists an $\alpha > 0$ such that $\forall t \geq 0$:

1. $\alpha\Lambda_d - D_d(t)$ is negative semi-definite
2. $\dot{K}_d(t) + \alpha\dot{D}_d(t) - 2\alpha K_d(t)$ is negative semi-definite

If in 2) semi-definiteness is replaced with definiteness, the stability property is in addition asymptotic.

Chapter 5

Tank Methods

Another method to ensure stability is to make the overall system passive (Appendix A briefly describes Passivity Theory). For this, a virtual tank is integrated into the system [5]. The exchange of energy with the tank is set up such that the overall system remains passive with respect to the pair (F_{ext},$\dot{\tilde{x}}$). We adopt this method as it allows online variable impedance unlike Lyapunov Method, and can also be used to vary Inertia Matrix.

The performance of the Tank Method largely depends on the formulation of the dynamics of the tank. In the following subsections we present from the literature different formulations for the dynamics of the tank (Tank 1 and Tank 2) and our formulation (Tank 3).

5.0.1 Tank 1: Ferraguti et.al. [5]

The extended dynamics of the overall system along with the tank is given as follows:

$$\begin{cases} \Lambda_d \ddot{\tilde{x}} + D_d \dot{\tilde{x}} + K_c \tilde{x} - w x_t = F_{ext} \\ \dot{x_t} = \frac{\sigma}{x_t} (\dot{\tilde{x}}^T D_d \dot{\tilde{x}}) - w^T \dot{\tilde{x}} \end{cases} \tag{5.1}$$

where, $K_d(t) = K_c + K'(t)$, K_c being the constant part of the stiffness matrix and $K'(t)$ the varying part. $x_t \in R$ is the state of the tank and

$$T(x_t) = \frac{1}{2} x_t^2 \tag{5.2}$$

is the energy stored in the tank. The term $w(t)$ is a control input through which it is possible to control the exchange of energy between the main impedance model and the tank. σ is a design parameter that enables/disables the storage of dissipated energy, i.e.:

$$\sigma = \begin{cases} 1 & if \quad T(x_t) \leq \bar{T} \\ 0 & otherwise \end{cases} \tag{5.3}$$

where $\bar{T} \in R^+$ is a suitable, application dependent, upper bound that avoid excessive energy storing in the tank. The variable stiffness behavior can be implemented by the following setting:

$$w(t) = \begin{cases} \frac{-K'(t)\tilde{x}}{x_t} & if \quad T(x_t) > \epsilon \\ 0 & otherwise \end{cases} \tag{5.4}$$

where $\epsilon > 0$ is a threshold below which energy cannot be extracted by the tank for avoiding singularities. If $T(x_t) \leq \epsilon$, then the desired variable stiffness cannot be implemented passively and the constant stiffness K_c is implemented.

5.0.2 Tank 2: Cardenas et.al [3]

The formulation of Tank 2 is similar to that of Tank 1 except for the control action $w(t)$, which is given as follows:

$$w(t) = \begin{cases} \frac{-K'(t)\tilde{x}}{x_t} & if \quad T(x_t) > \epsilon \\ \frac{-K'(t)\tilde{x}x_t}{\gamma^2} & otherwise \end{cases} \tag{5.5}$$

where, $\gamma = \sqrt{2\epsilon}$. Note that here, unlike the method of Tank 1, the control action is not abruptly set to 0 but is gradually shut down when the tank energy goes below ϵ. The advantage of this formulation will be clearer in the following sections.

5.0.3 Tank 3: Our Formulation

Tank 1 and Tank 2 are based on the idea that interconnection of two passive systems (constant impedance system and tank) is also passive. In our method, instead of having two separate passive systems, we set up the tank dynamics such that the overall systems remains passive. The distinctiveness of our method comes from the formulation of the dynamics of the tank and the adaptation of the stiffness variation when the tank gets empty. The extended dynamics of our system are given as follows:

$$\begin{cases} \Lambda_d \ddot{\tilde{x}} + D_d \dot{\tilde{x}} + K(t)\tilde{x} = F_{ext} \\ \dot{T} = \alpha(T)\dot{\tilde{x}}^T D_d(t)\dot{\tilde{x}} - \frac{1}{2}\beta(\dot{K})\tilde{x}^T \dot{K}(t)\tilde{x} \end{cases} \tag{5.6}$$

$\alpha(s)$ and $\beta(\dot{K})$ are design parameter that enable/disable the storage of energy into the tank and are given as follows:

$$\alpha(T) = \begin{cases} 1 & 0 < T < \bar{T} \\ 0 & T > \bar{T} \end{cases} \tag{5.7}$$

$$\beta(\dot{K}) = \begin{cases} 1 & if \quad \dot{K} > 0 \\ 0 & if \quad \dot{K} \leq 0 \end{cases} \tag{5.8}$$

where, $\bar{T}$ is the upper limit for the energy storage in the tank to avoid practically unstable behaviors. The reason for the choice of β will be clear ahead in this section.

The varying stiffness $K(t)$ is given as follows:

$$K(t) = K(0) + \int_0^t \dot{k}(\tau)d\tau \tag{5.9}$$

where,

$$\dot{k}(\tau) = \gamma(T, K)(\dot{K}_d + Pe) \tag{5.10}$$

where, $\dot{K}_d$ is the time derivative of the desired stiffness, $\mathbf{e}$ is error between the desired stiffness $K_d(t)$ and the actual stiffness $K(t)$ realized in the impedance law in (), P is a proportional term acting on $\mathbf{e}$ and $\gamma(T, K)$ is given as follows:

$$\gamma(T,K) = \begin{cases} 0 & if \quad T \notin [0, \bar{T}] \quad or \quad K \notin [K_{min}, K_{max}] \\ 1 & otherwise \end{cases} \tag{5.11}$$

This procedure ensure that the stiffness remains constant at its last achieved value when the tank gets empty and gradually gets along the desired stiffness $K_d(t)$ (due to the term $\dot{K}_d + Pe$ in equation 5.10), when the tank starts filling up again. Morover, it can be seen that the issue of singularity does not arise here.

The proof for the overall system's passivity can be found in Appendix B.

Chapter 6

Evaluation

6.0.1 Regulation with perturbations:

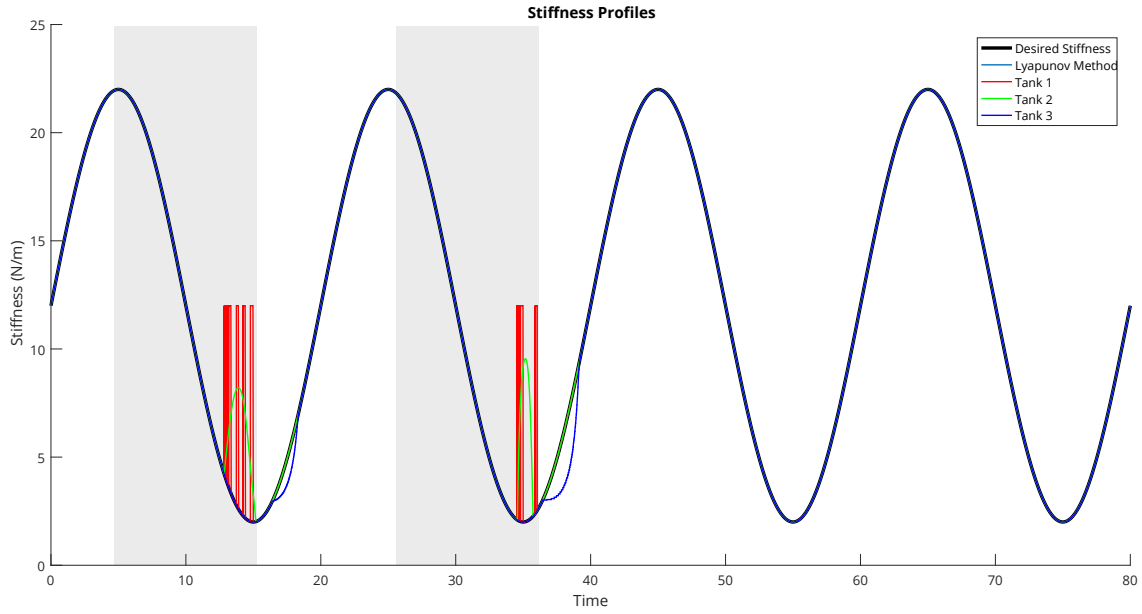


Figure 6.1: Comparison of Stiffness profiles of mentioned methods for regulation with perturbations experiment

To compare the performance of Lyapunov method with that of Tank Methods, an experiment is performed wherein, a known varying stiffness profile is given for a one DOF impedance law given as follows:

- $\Lambda_d = 10kg$
- $D_d = 4Ns/m$
- $K_d(t) = 12 + 10sin(\frac{\pi t}{10})$

The desired trajectory $x_d(t)$ is set as follows:

$$x_d(t) = 0.1sin(0.1t) \tag{6.1}$$

This impedance profile follows the two conditions given in the Chapter 3, therefore allowing use of Lyapunov Method. A constant perturbation force was applied in

intervals shaded gray in the plots. Giving perturbations is essential for having a comparison between Lyapunov method and Tank Methods as tank based methods are sensitive to perturbations (Perturbations change the state $\tilde{x}$ of the system, which in-turn changes the dynamics of the tank).

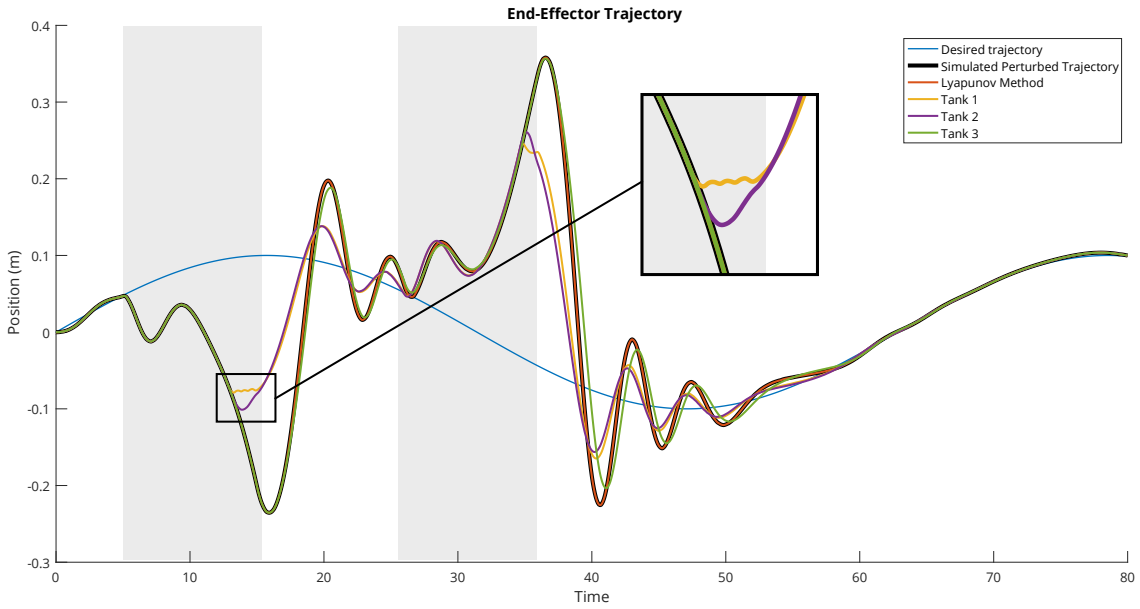


Figure 6.2: Comparison of End-Effector trajectory of mentioned methods for regulation with perturbations experiment

The results for the four methods is shown in Figure 6.1 and 6.2. It can be observed that Lyapunov Method provides the best performance by following the simulated trajectory perfectly as its stability criteria is independent of the states and thus perturbations. While in the case of tank based methods, the tanks get empty inside perturbation intervals as position and velocity errors can lead to inlet and outlet of energy in the tank. It can be clearly seen that stiffness profile of Tank 1 suffers from chattering as it switches from desired stiffness $K_d(t)$ and the constant part of the stiffness i.e. 12. The chattering effect in the stiffness profile in Tank 1 also affects the trajectory regulation as can be seen in Figure 6.2. The chattering is appreciably taken care of by Tank 2, but this method still tries to go towards the constant part of the stiffness. While in case of Tank 3, it can be seen that a smooth transition is observed between real stiffness and desired stiffness. The trajectory regulation is also better in case of Tank 3.

6.0.2 Sponge Insertion:

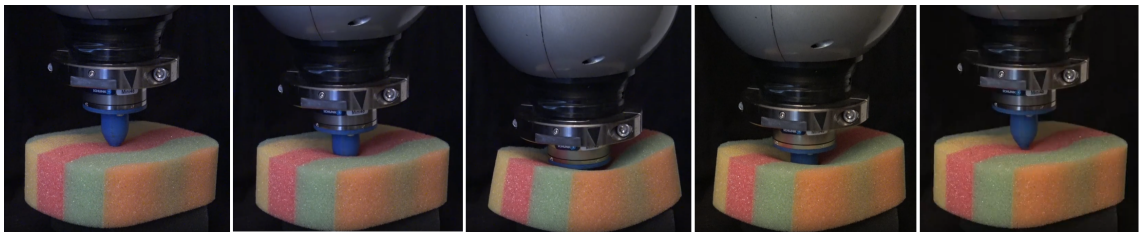


Figure 6.3: Sponge Insertion Experiment

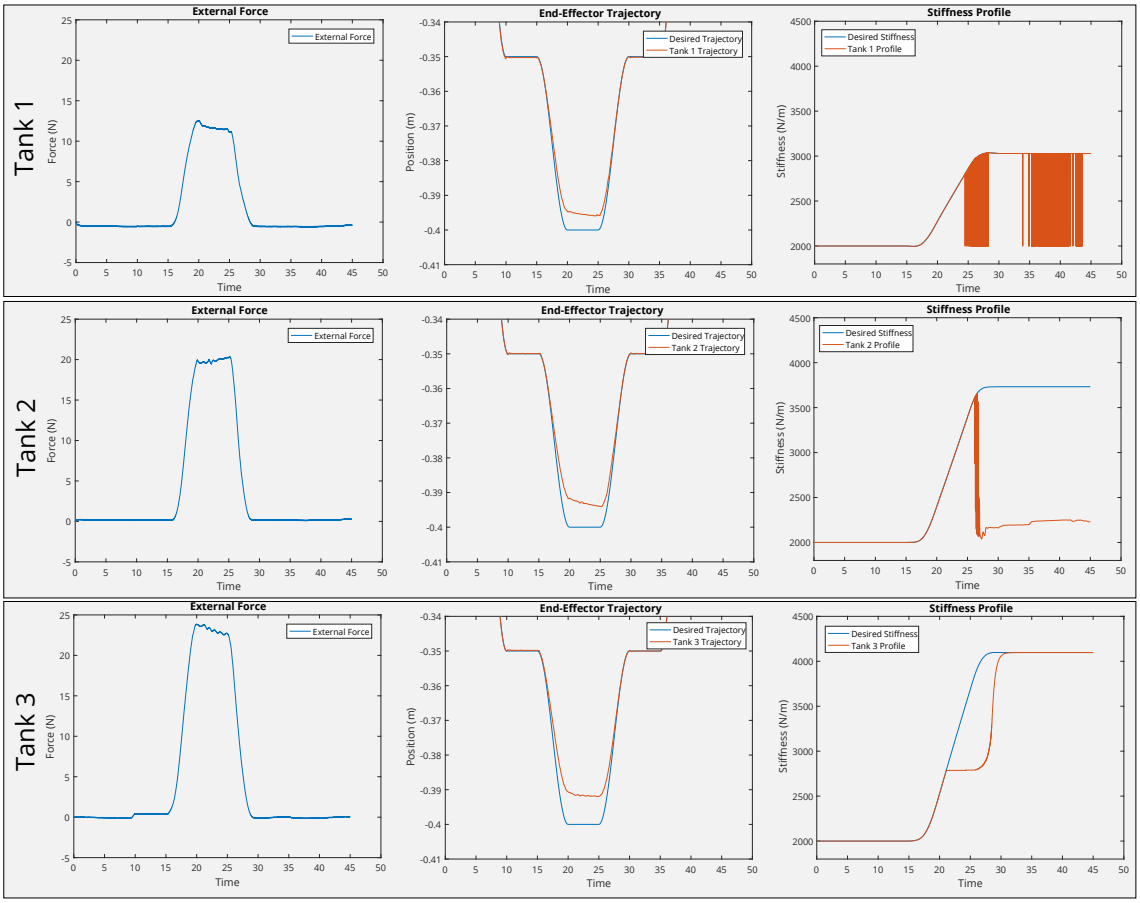


Figure 6.4: Comparison of End-effector trajectory and Stiffness profiles of mentioned methods for Sponge Insertion experiment

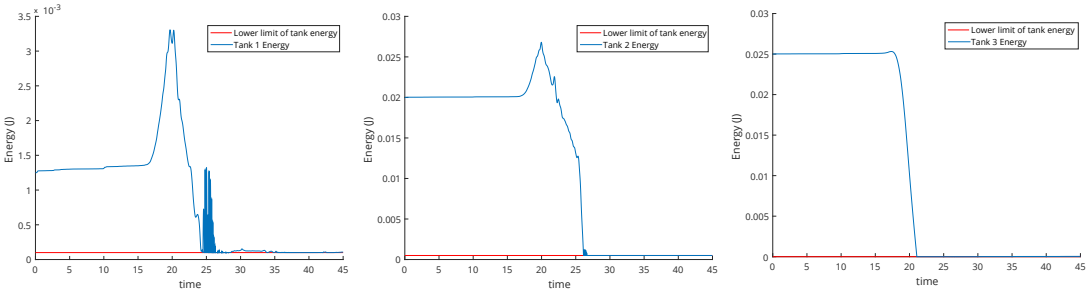


Figure 6.5: Energies of tanks during sponge insertion experiment

To check the performance of tank methods, an online variable impedance task was also set up. This aim of this experiment was to reproduce a surgical scenario wherein we have interaction between the manipulator's tool tip and a body tissue with unknown characteristics. As can be seen in Figure 6.3, this was done by inserting the manipulator's end-effector into a bath sponge. The desired stiffness in the impedance law was varied as follows:

$$K_d(t) = 2000 + \int_0^t 10 F_{ext} \tag{6.2}$$

where, F_{ext} is the force acting on the end-effector due to the insertion in sponge.

Figure 6.4 shows the stiffness and end-effector position plots. The superiority of Tank 3 can be well appreciated by observing the stiffness profile in this Figure. In this experiment, the constant part of stiffness is 2000. When the respective tanks get empty, Tank 1 and Tank 2 methods send the actual stiffness profile towards the constant part of the stiffness while Tank 3 maintain its last achieved value of stiffness. This helps Tank 3 to remain as close as possible to the desired value of stiffness. Moreover, the transition of the real stiffness value to the desired stiffness value is smooth for Tank 3 as compared to that for Tank 1 and Tank 2.

Chapter 7

Conclusion

In this work, the problem of ensuring stability for variable impedance controlled robotic manipulator has been addressed. The Lyapunov Method and the Tank based methods are widely adopted for this task. Although the Lyapunov method works best in case the varying impedance profile is known a-priori, it cannot be used to vary impedance online. Moreover, it does not allow the variation of Inertia in the impedance law. Although the tank based methods gives a sub-optimal performance as compared to Lyapunov Method, it can be used for varying all the impedance parameters offline as well as online and without any specific constraints on the impedance profile.This preserves the appealing robustness idea behind impedance control.
Among the tank based methods, Tank 1 suffers from chattering as it sends the actual impedance to a constant part of impedance profile abruptly when the tank gets empty. Tank 2 reduces these chattering effects but still suffers from the factor that it gradually tries to send the actual impedance profile to the constant part of impedance profile. This can be problem in cases when the desired stiffness is far from the constant part of stiffness for a long period of time. Unlike these methods, Tank 3 tries to maintain an impedance profile as close as possible to the desired one. This also in turn affects less the end-effector trajectory profile.

Appendix A

Passivity Analysis

Simply stated, a system is said to be passive when it cannot deliver more energy than what is stored in it i.e. it does not generate internal energy. A passive system satisfies the following energy balance equation:

$$V(x(t)) - V(x(0)) = \int_0^t u^T(s)y(s)ds - d(t) \tag{A.1}$$

$$V(x(t)) - V(x(0)) \leq \int_0^t u^T(s)y(s)ds \tag{A.2}$$

where, $V(x)$ is the total (free) energy function, $x \in R^n$ is the state vector, u and y are power conjugated variables, and $d(t)$ is a positive dissipation function describing (irreversible) energy conversion to the thermal domain. The system is said to be passive with respect to the pair (u,y) using the storage function $V(x)$.

Appendix B

Tank 3 Passivity Analysis

The following storage function of the extended system ():

$$\mathcal{W} = \mathcal{V} + T \tag{B.1}$$

where, V is the total energy of the mechanical impedance model, given as follows:

$$\mathcal{V}(\tilde{x}, \dot{\tilde{x}}) = \frac{1}{2}\dot{\tilde{x}}^T \Lambda \dot{\tilde{x}} + \frac{1}{2}\tilde{x}^T K(t)\tilde{x} \tag{B.2}$$

and T is the stored energy of the tank.
We therefore have:

$$\begin{aligned}
\dot{\mathcal{W}} &= \dot{\mathcal{V}} + \dot{T} \\
&= \dot{\tilde{x}}^T \Lambda_d \ddot{\tilde{x}} + \frac{1}{2}\tilde{x}^T \dot{K}\tilde{x} + \tilde{x}^T K\dot{\tilde{x}} + \dot{T} \\
&= -\dot{\tilde{x}}^T D_d \dot{x} - \dot{\tilde{x}}^T K\tilde{x} + \dot{\tilde{x}}^T F_{ext} + \tilde{x}^T K\dot{\tilde{x}} + \frac{1}{2}\tilde{x}^T \dot{K}\tilde{x} + \dot{T} \\
&= -\dot{\tilde{x}}^T D_d \dot{x} + \dot{\tilde{x}}^T F_{ext} + \frac{1}{2}\tilde{x}^T \dot{K}\tilde{x} + \alpha\dot{\tilde{x}}^T D_d \dot{\tilde{x}} - \frac{1}{2}\beta\tilde{x}^T \dot{K}\tilde{x} \\
&= \dot{\tilde{x}}^T F_{ext} - (1-\alpha)\dot{\tilde{x}}^T D\dot{\tilde{x}} + \frac{1}{2}(1-\beta)\tilde{x}^T \dot{K}\tilde{x}
\end{aligned} \tag{B.3}$$

Since $\alpha, \beta \in \{0, 1\}$, we have that

$$-(1-\alpha)\dot{\tilde{x}}^T D\dot{\tilde{x}} + \frac{1}{2}(1-\beta)\tilde{x}^T \dot{K}\tilde{x} \leq 0$$

and, therefore the augmented system is passive respect to the pair $(F_{ext}, \dot{\tilde{x}})$.

Appendix C

Regulation with Perturbations Experiment C++ Code

```
#include <LWRJointPositionController.h>
#include <FastResearchInterface.h>
#include <friComm.h>
#include <math.h>
#include <time.h>
#include <TooN/TooN.h>
#include <TooN/GR_SVD.h>
#include <TooN/LU.h>
#include "simulation.h"
#include <iostream>
#include <fstream>
#include <string.h>
#include <stdio.h>
#include <ATI_FTsensor.h>

using namespace std;
using namespace TooN;
#define NUMBER_OF_JOINTS          7
#define RUN_TRAJECTORY 80.0
#ifndef PI
#define PI       3.14159265358979323846264338327950
#endif

#define EOK                                    0
#define RUN_TIME                    80.0// choose appropriate
    size for Matrix dat[]

ofstream results;

// ------------------------------------ function
    ------------------------------------

Vector <3> erroreQuaternioni(Matrix<4,4> Tbe, Matrix<4,4>
    Tbe_d);
```

```
31 Vector <3> r2quat(Matrix<3,3> R_iniz, double &eta);
32 Matrix <3> skew(Vector<3> v);
33 //
   ------------------------------------------------------------------------

34 //================================== Trajectory Function
   ====================
35 inline void linear_trajectory(float *p,float *dp,float *ddp,
   Matrix<3,1> xi,Matrix<3,1> xf,float ti,float tf,float tik
   ){
36     Matrix<3,1> a3=10.0/pow(tf-ti,3)*(xf-xi);
37     Matrix<3,1> a4=-15.0/pow(tf-ti,4)*(xf-xi);
38     Matrix<3,1> a5=6.0/pow(tf-ti,5)*(xf-xi);
39
40     for(int i=0; i<3; i++){
41         p[i]=xi(i,0)+a3(i,0)*pow(tik-ti,3)+a4(i,0)*pow(tik-
               ti,4)+a5(i,0)*pow(tik-ti,5);
42         dp[i]=3.0*a3(i,0)*pow(tik-ti,2)+4.0*a4(i,0)*pow(tik-
               ti,3)+5.0*a5(i,0)*pow(tik-ti,4);
43         ddp[i]=6.0*a3(i,0)*(tik-ti)+12.0*a4(i,0)*pow(tik-ti
               ,2)+20.0*a5(i,0)*pow(tik-ti,3);
44     }
45 }
46
47 //-------------------- Data matrices --------------------
48 const int datasize=1000*RUN_TIME/2;
49 Matrix<datasize,8>   dat=Zeros;
50 Matrix<datasize,7>   DTorque=Zeros;
51 Matrix<datasize,4>   Dks=Zeros;
52 Matrix <datasize,3> Dp_e,Dw,Dpd_e,Dx_pos,Dxd_pos,DKp,DLambda
   ,DLp,Dmass_real,Dstiff_real,Dstiff_des,Dmass_des,Derr,
   Dderr,Dgamma,Dx_des,Dxd_des,Dxdd_des,DDamping,DF=Zeros;
53 Matrix <datasize,1> D_tank,Dst,Dd_st_num,Dd_st_an,Dalfa,
   Dbeta,Di_o=Zeros;
54 Matrix <datasize,2> Dteta=Zeros;
55 //------------------------------------------------------------

56
57 //
   ========================================================================

58 int main(int argc, char *argv[]){
59
60     results.open("/home/prisma/Desktop/Ajinkya/denny_tanks/
         results.csv",ios::app);
61
62     int ResultValue=0;
63     float JointValuesInRad[NUMBER_OF_JOINTS];
64     float jp[NUMBER_OF_JOINTS];
```

```
65      float tau[NUMBER_OF_JOINTS];
66      float MTorque[NUMBER_OF_JOINTS];
67      float InitialJointValuesInRad[NUMBER_OF_JOINTS
           ]={0.0,0.0,0.0,0.0,0.0,0.0,0.0};
68      double t=0.0;
69      double CycTime;
70
71      float *force;
72      force = new float[6];
73      for(int j=0;j<6;j++)
74          force[j] = 0.0;
75
76      float *POS;
77      POS= new float[12];
78      for (int i=0; i<12;i++)       POS[i]=0.0;
79
80      float **JacMatrix;
81      JacMatrix= new float*[FRI_CART_VEC];
82      for (int i=0; i<FRI_CART_VEC;i++){
83          JacMatrix[i]=new float [NUMBER_OF_JOINTS];
84      }
85      for (int i=0; i<FRI_CART_VEC;i++){
86          for (int j=0; j< NUMBER_OF_JOINTS ;j++)
87              JacMatrix[i][j]=0.0;
88      }
89
90      float **Mass;
91      Mass= new float*[NUMBER_OF_JOINTS];
92      for (int i=0; i<NUMBER_OF_JOINTS;i++){
93          Mass[i]=new float [NUMBER_OF_JOINTS];
94      }
95      for (int i=0; i<NUMBER_OF_JOINTS;i++){
96          for (int j=0; j< NUMBER_OF_JOINTS ;j++) Mass[i][j
               ]=0.0;
97      }
98      //
            ------------------------------------------------------------------------------------------
99
100         FastResearchInterface *FRI;
101         FRI = new FastResearchInterface("/home/prisma/
               Desktop/FRILibrary/etc/980039-FRI-Driver.init");
102
103     //------------------------------------------init FT sensor
            ------------------------------------------//
104     ATI_FTsensor *FTsensor;
105     FTsensor= new ATI_FTsensor((char*)"143.225.169.201"); //
           ATI FT sensor IP
```

```
106     /*Set a software Bias for Force and Torque*/
107     FTsensor->setBias();
108     // ---------------------------------------- Variables
          ----------------------------------------//
109
110     Vector<3> x,
111     x_old,
112     xp,xpp,
113     xi_old,
114     m_diag,
115     xp_old,
116     kpv_diag,
117     kp_diag,
118     e,
119     edot,
120     ep,
121     eo,
122     epp,
123     eop,
124     xpi,
125     f,
126     f_old,
127     fd,
128     xi=Zeros;
129
130     Matrix<3,1> err3d,
131     p_e,
132     F3_ext,
133     errpos,
134     pd_e,errvel,
135     pdd_e=Zeros;
136
137     //giunti
138     Vector<7> q,
139     qi,
140     qp,
141     qpi,
142     q_old,
143     qpp,
144     rid,
145     qp_old,
146     q_plot,
147     q2_o,
148     q_o=Zeros;
149
150     Matrix<7,1> qd,qdF,
151     qdd,
152     qdprim=Zeros;
```

```
153
154     Matrix <3,3> Re,
155     S_pe,
156     Da,
157     Lambda,
158     LambdaDes,
159     LambdaDes_inv,
160     Lambda_inv,
161     K,
162     stiff,
163     Rif,
164     Kf,
165     KI,
166     Mcost,
167     Mvar,
168     Rd=Zeros;
169
170     Matrix <4,4> Tbe,
171     Tbe_d=Zeros;
172
173     Vector <6> fsens,
174     e_e,
175     e_dot,
176     fsens_f = Zeros;
177
178     Matrix <7,3> Js,
179     Jso=Zeros;
180     Matrix <7,6> J_s;
181     Matrix <6,7> J=Zeros;
182     Matrix <6,6> A,B,T=Zeros;
183     Matrix <3,7> J3dof,Jprec,Jdot,Jo=Zeros;
184     Matrix <7,7> M=Zeros;
185     Matrix <7,7> M_inv=Identity;
186
187
188     B=Data(1,0,0,0,0,0,
189            0,1,0,0,0,0,
190            0,0,1,0,0,0,
191            0,0,0,0,0,1,
192            0,0,0,0,1,0,
193            0,0,0,1,0,0);
194     float Fext[6];
195
196
197     Vector <3> Kst,Kstf;
198     double phi1,beta1,gamma1,y_1,y1=0;
199     double phi2,beta2,gamma2,y_2,y2=0;
200     double phi3,beta3,gamma3,y_3,y3=0;
```

```
201     double phi1f,beta1f,gamma1f,y_1f,y1f=0;
202     double phi2f,beta2f,gamma2f,y_2f,y2f=0;
203     double phi3f,beta3f,gamma3f,y_3f,y3f=0;
204     double V1=5;
205     double V2=5;
206     double V3=5;
207
208     double V1f=5;
209     double V2f=5;
210     double V3f=5;
211     double kx=10;
212     double ky=10;
213     double kz=10;
214     double kxf=10;
215     double kyf=10;
216     double kzf=10;
217     double kzf_old=10;
218     double dkzf=10;
219
220
221
222     //========== VARIABLES TANK ============================
223     double upperT = 0.05;              //upper bound tank
224     double lowerT = 0.0;              //lower bound continuit
          tank
225     double xt  = 0.001;
226     double xt_old = xt;
227     double var=0.0;
228
229 double stiff_d = 12;
230 double stiff_a = 12;
231 double stiff_e = 0;
232 double stiff_deri=0;
233
234     double T_tank=0.001;
235     double sigma=0.01;
236     Matrix<3,1> w = Zeros;
237     Matrix<1,1> dxt=Zeros;
238     float d_storage_num=0.0;
239     Matrix<1,1> temp=Zeros,
240     T_temp=Zeros;
241     Matrix<1,1> ds,storage,i_o,s_temp,storage_old    = Zeros;
242     Matrix <1,1> d_storage_an = Zeros;
243     Matrix <3,3> stiff_var=Zeros,
244                  stiff_var_des=Zeros,
245                 stiff_cost=Zeros,
246                 stiff_des=Zeros,
247                 stiff_real=Zeros;
```

```
248     //==== ============================
249 Matrix<2,1> phi,teta,gamma_stima=Zeros;
250 Matrix<2,2>  GUAD=Zeros;
251     Matrix<1,1>  beta_stima,Fz,beta_stima_inv,modello_forza=
            Zeros;
252     Matrix<1,1>  temp_stima=Data(1.0);
253     GUAD=Data(1.2,0.0,
254               0.0,1.2);
255
256     //============================
257     float tiA,tAB,tBC,tCD,tDE,tEF,tFi,t0,tf,*p_,*dp_,*ddp_;
258     p_=new float[3];
259     dp_=new float[3];
260     ddp_=new float[3];
261     Vector<3> x_des,xd_des,xdd_des,delta_x,delta_dx=Zeros,
            x_init;
262     Matrix<3,1> pA,pB,pC,c2,pD,pE,pF;
263     int k;
264     tiA=10.0;
265     tAB=13.0;
266     tBC=31.0;
267     tCD=47.0;
268     tDE=55.0;
269     t0=15;
270     tf=31;
271     pA=Data(-0.671, 0.431,  0.042);
272     pB=Data(-0.67, 0.43,  0.042);
273     pC=Data(-0.67, 0.43,  0.0);
274     pD=Data(-0.67, 0.43,  0.05);
275     pE=Data(-0.5492, 0.0064,  0.4562);
276     //
            ------------------------------------------------------------
277
278     //========  M, D, K  ============================
279
280     LambdaDes = Data(10.0,0.0,0.0,
281                      0.0,10.0,0.0,
282                      0.0,0.0,10.0);
283     LambdaDes_inv=Data(1/LambdaDes(0,0),0.0,0.0,
284                        0.0,1/LambdaDes(1,1),0.0,
285                        0.0,0.0,1/LambdaDes(2,2));
286
287         Da=Data(4.0,0.0,0.0,
288                 0.0,4.0,0.0,
289                 0.0,0.0,4.0);
290
291     kp_diag=makeVector(12,12,12);
```

```
292     stiff_cost=Data(kp_diag[0],0.0,0.0,
293                     0.0,kp_diag[1],0.0,
294                     0.0,0.0,kp_diag[2]);
295
296     //=============================================================
297
298     fprintf(stdout, "RobotJointPositionController object
            created.\n Starting the robot...\n");
299     ResultValue=FRI-> StartRobot(FastResearchInterface::
            JOINT_POSITION_CONTROL,120);
300
301     if (ResultValue == EOK) fprintf(stdout, "Robot
            successfully started.\n");
302     else                    fprintf(stderr, "ERROR, could
            not start robot: %s\n", strerror(ResultValue));
303
304     fprintf(stdout, "Current system state:\n%s\n", FRI->
            GetCompleteRobotStateAndInformation());
305     CycTime=FRI->GetFRICycleTime();
306     FRI-> GetMeasuredJointPositions(InitialJointValuesInRad)
            ;
307
308
309     cout<< "\nThe cycle time is "<<CycTime<<endl;
310
311     //-----------------------------------------init clik
            -----------------------------------------//
312
313     qi=makeVector(InitialJointValuesInRad[0],
            InitialJointValuesInRad[1],InitialJointValuesInRad
            [2],InitialJointValuesInRad[3],
            InitialJointValuesInRad[4],InitialJointValuesInRad
            [5],InitialJointValuesInRad[6]);
314
315     q_old=qi;
316     qp=Zeros;
317     qp_old=qp;
318
319     FRI->GetMeasuredCartPose(POS);
320
321     Re=Data(POS[0], POS[1], POS[2],
322             POS[4], POS[5], POS[6],
323             POS[8], POS[9], POS[10]);
324
325     Tbe_d=Data(POS[0], POS[1], POS[2],POS[3],
326                POS[4], POS[5], POS[6], POS[7],
327                POS[8], POS[9], POS[10],POS[11],
328                0.0,0.0,0.0,1.0);
```

```
329
330     xi=makeVector(POS[3],POS[7],POS[11]);
331     x_init=makeVector(POS[3],POS[7],POS[11]);
332     xi_old=xi;
333     x=xi;
334     x_old=x;
335
336     xp=Zeros;
337     xp_old=xp;
338
339     FRI->GetCurrentJacobianMatrix(JacMatrix);
340
341     for (int i=0;i<FRI_CART_VEC;i++){
342         for (int j=0;j<NUMBER_OF_JOINTS;j++)
343             J[i][j]=JacMatrix[i][j];
344     }
345
346     //------------------------------------------ jacobian init
          ------------------------------------------
347
348     A=Data(Re(0,0),Re(0,1),Re(0,2), 0      ,0      ,0,
349            Re(1,0),Re(1,1),Re(1,2), 0      ,0      ,0,
350            Re(2,0),Re(2,1),Re(2,2), 0      ,0      ,0,
351            0      ,0      ,0      ,Re(0,0),Re(0,1),Re(0,2),
352            0      ,0      ,0      ,Re(1,0),Re(1,1),Re(1,2),
353            0      ,0      ,0      ,Re(2,0),Re(2,1),Re(2,2));
354
355
356     J=A*B*J;
357
358     J3dof=J.slice(0,0,3,7);
359     Jprec=J3dof;
360
361
362     //------------------------------------------  init control
          ------------------------------------------
363
364     double daz,day,dax=0.0;
365     double max,may,maz=0.0;
366
367
368     Vector<7> Pred=Zeros;
369
370
371
372     //
          ==========================================================
```

```
373     //============================== START LOOP
            ==========================================
374     //
            ================================================================================


375
376     int itr=0;
377     cout<<" xi= "<<xi<<endl;
378     while (t<=RUN_TIME){
379
380
381         FRI->WaitForKRCTick();
382
383         if (!FRI->IsMachineOK())
384         {
385             fprintf(stderr, "ERROR, the machine is not ready
                    anymore.\n");
386             break;
387         }
388
389
390         //------------------------------------------ VARIAZIONE
                STIFFNESS ------------------------------------------
391
392         if(t<=tAB)   var=0;
393         if(t>tAB && t<tAB+1) var=0.5*(1+cos((tAB+1-t)*PI/(1)
                ));
394         if(t>=tAB+1)   var = 1;
395
396         if(t>tAB+1 && t<=tCD)   var=1;
397         if(t>tCD && t<=tCD+1) var=0.5*(1+cos((tCD-t)*PI/(1))
                );
398         if(t>tCD+1)   var = 0;
399
400
401         kpv_diag=makeVector(2000*var,2000*var,2000*var);
402
403         stiff_var_des=Data(0.0,           0.0,
                0.0,
404                            0.0,   10*sin(PI*t/10),
                                          0.0,
405                            0.0,           0.0,      0.0);
406
407
408         stiff_des=stiff_cost + stiff_var_des;
409
410
411
```

```
412         x_des= x_init + makeVector(0,0.1*sin(0.1*t),0);
413         xd_des=makeVector(0,0.1*0.1*cos(0.1*t),0);
414         xdd_des=makeVector(0,-0.1*0.1*0.1*sin(0.1*t),0);
415
416         //recieve the pos, jacobian, mass at each cycle time
417         FRI->GetMeasuredCartPose(POS);
418         FRI-> GetMeasuredJointPositions(jp);
419         q=makeVector(jp[0],jp[1],jp[2],jp[3],jp[4],jp[5],jp
                [6]);
420         qpi=(q-q_old)/CycTime;
421         qd=Data(qpi[0],qpi[1],qpi[2],qpi[3],qpi[4],qpi[5],
                qpi[6]);
422         qdF   = 0.1*qd+(1.0-0.1)*qdF;
423         pd_e = J3dof*qdF;
424         Re=Data(POS[0], POS[1], POS[2],
425                                 POS[4], POS[5], POS[6],
426                                 POS[8], POS[9], POS[10]);
427
428         FRI->GetCurrentJacobianMatrix(JacMatrix);
429         for (int i=0;i<FRI_CART_VEC;i++){
430              for (int j=0;j<NUMBER_OF_JOINTS;j++)     J[i][j]=
                    JacMatrix[i][j];
431         }
432
433         A=Data(Re(0,0),Re(0,1),Re(0,2), 0.0      ,0.0
                ,0.0,
434              Re(1,0),Re(1,1),Re(1,2), 0.0      ,0.0
                    ,0.0,
435              Re(2,0),Re(2,1),Re(2,2), 0.0      ,0.0
                    ,0.0,
436              0.0      ,0.0      ,0.0      ,Re(0,0),Re(0,1)
                    ,Re(0,2),
437              0.0      ,0.0      ,0.0      ,Re(1,0),Re(1,1),
                    Re(1,2),
438              0.0      ,0.0      ,0.0      ,Re(2,0),Re(2,1)
                    ,Re(2,2));
439
440         J=A*B*J;
441         J3dof=J.slice(0,0,3,7);
442         Jdot=(J3dof-Jprec)/CycTime;
443         xi=makeVector(POS[3],POS[7],POS[11]);
444         xpi=J3dof*qp;
445         //
                ------------------------------------------------------------
446
447
448         //----------------------martice M
```

```
449         FRI-> GetCurrentMassMatrix(Mass);
450         for (int i=0; i<NUMBER_OF_JOINTS;i++){
451             for (int j=0; j< NUMBER_OF_JOINTS ;j++)  M[i][j
                ]=Mass[i][j];
452         }
453
454         LU<7> lu_M(M);
455         M_inv=lu_M.get_inverse();
456
457         //------------------------ Lambda ------------
458         LU<3> lu_J(J3dof*M_inv*J3dof.T());
459         Lambda      = lu_J.get_inverse();
460         Lambda_inv = J3dof*M_inv*J3dof.T();
461         //-------------------------------------------
462
463
464
465         S_pe=Data(0.0,-xi[2],xi[1],
466                   xi[2],0.0,-xi[0],
467                   -xi[1],xi[0],0.0);
468         S_pe=S_pe*Re;
469
470         T=Data(Re(0,0),Re(0,1),Re(0,2),0.0,0.0,0.0,
471                Re(1,0),Re(1,1),Re(1,2),0.0,0.0,0.0,
472                Re(2,0),Re(2,1),Re(2,2),0.0,0.0,0.0,
473                S_pe(0,0),S_pe(0,1),S_pe(0,2),Re(0,0),Re(0,1)
                   ,Re(0,2),
474                S_pe(1,0),S_pe(1,1),S_pe(1,2),Re(1,0),Re(1,1)
                   ,Re(1,2),
475                S_pe(2,0),S_pe(2,1),S_pe(2,2),Re(2,0),Re(2,1)
                   ,Re(2,2));
476
477         Tbe=Data(POS[0], POS[1], POS[2],POS[3],
478                                  POS[4], POS[5], POS[6],POS
                                     [7],
479                                  POS[8], POS[9], POS[10],POS
                                     [11],
480                  0.0,0.0,0.0,1.0);
481
482         //------------------------------------------ force sensor
                ------------------------------------
483         FTsensor->getFTData(force);
484
485         for(int i=0;i<6;i++)  fsens[i]=force[i];
486
487         fsens=T*fsens;
488
```

```
489                                     if(t>=5 && t<=15)
490         {f=makeVector(0,-1,0);}
491                                     else if(t>=26 && t<=36)
492         {f=makeVector(0,1,0);}
493                                     else
494         {f=makeVector(0,0,0);}
495         //
              ------------------------------------------------------------------------


496
497         //=========================================== TANK
              ========================================
498         errpos=Data(-x_des[0]+xi[0],-x_des[1]+xi[1],-x_des
              [2]+xi[2]);
499         T_tank=xt;
500
501
502         if(T_tank<=upperT)
503                                     {
504                                       if(T_tank<=upperT &&
                                            T_tank>upperT*0.9) {
                                            sigma = 0.5*(1+cos((
                                            T_tank-upperT*0.9)*PI
                                            /(0.1*upperT)));cout<<"
                                            inif= "<<endl;}
505           else {sigma =1;}
506         }
507         else
508         {sigma = 0;}
509
510 double temp_kder = PI*cos(PI*t/10);
511 double beta_k =0;
512 double gamma_k=0;
513
514 if(temp_kder>0) beta_k = 1;
515 else beta_k=1;
516
517 if(T_tank>=0.0 && T_tank<0.05) gamma_k=1;
518 else gamma_k=0;
519
520 stiff_e = 12+10*sin(PI*t/10) - stiff_a;
521
522 stiff_deri = (PI*cos(PI*t/10) + 20*stiff_e)*gamma_k;
523 Matrix<3,3> stiff_deri_mat = Data(0,0,0,
524
```

```
525                                         0,0,0);
526
527         dxt = (sigma)*(errvel.T()*Da*errvel) - 0.5*beta_k*
                errpos.T()*stiff_deri_mat*errpos;
528         xt =xt_old + CycTime*dxt(0,0);
529
530                                         if(xt>0.05) xt = 0.049;
531
532
533         GR_SVD<3,7> J_svd(J3dof);
534         Js = J_svd.get_pinv();
535         e=x-xi;
536         edot=xp-xpi;
537         errvel=Data(-xd_des[0]+xpi[0],-xd_des[1]+xpi[1],-
                xd_des[2]+xpi[2]);
538         delta_x= x-x_des;
539         delta_dx=xp-xd_des;
540
541                 //-------------------------------------controllo
                        ammettenza ------------------------------------
542         xpp = LambdaDes_inv*( -Da*delta_dx -stiff_a*delta_x
                + f );
543         xp  = xp_old+CycTime*xpp;
544         x   = x_old+CycTime*xp_old;
545
546 results<<xi[1]-x_init[1]<<","<<x_des[1]-x_init[1]<<","<<
    stiff_a<<","<<12+10*sin(PI*t/10)<<","<<xt<<","<<sigma<<",
    "<<gamma_k<<endl;
547
548 stiff_a = stiff_a + stiff_deri*CycTime;
549
550                 //-------------------------------------- redundancy
                        ---------------------------------------
551
552         eo=erroreQuaternioni(Tbe,Tbe_d);
553         Jo=Data(J(3,0),J(3,1),J(3,2),J(3,3),J(3,4),J(3,5),J
                (3,6),
554                 J(4,0),J(4,1),J(4,2),J(4,3),J(4,4),J(4,5),J
                    (4,6),
555                 J(5,0),J(5,1),J(5,2),J(5,3),J(5,4),J(5,5),J
                    (5,6) );
556
557         GR_SVD<3,7> Jor_svd(Jo);
558         Jso=Jor_svd.get_pinv();
559         eo=5.0*eo;
560         q_o=Jso*eo;
561         qpp=Js*(xpp+10.0*edot + 50.0*e -Jdot*qp) + (Identity
                (7) - Js*J3dof) * (-2.0*qp+1.7*q_o);
```

```
562
563
564         //------------------------------------ clik
                ------------------------------------
565         qp=qp_old + CycTime*qpp;
566         q=q_old + CycTime*qp_old;
567
568         //======================================== command
                ========================================
569         for(int i=0;i<NUMBER_OF_JOINTS;i++)   jp[i]=q[i];
570         FRI->SetCommandedJointPositions(jp);//
                JointValuesInRad
571         //
                ================================================================================


572
573
574         i_o = pd_e.T()*F3_ext;
575
576
577         //------------------------------------ write Matrices
                ------------------------------------
578         dat[itr]        = makeVector(t,jp[0],jp[1],jp[2],jp
                [3],jp[4],jp[5],jp[6]);
579         Dp_e[itr]       = makeVector(xi[0],xi[1],xi[2]);
580         Dpd_e[itr]      = makeVector(xpi[0],xpi[1],xpi[2]);
581         Dx_des[itr]     = makeVector(x_des[0],x_des[1],x_des
                [2]);
582         Dxd_des[itr]    = makeVector(xd_des[0],xd_des[1],
                xd_des[2]);
583
584         DF[itr]         = makeVector(f[0],f[1],f[2]);
585         Dst[itr]            = makeVector(storage(0,0));
586                 Dd_st_num[itr]    = makeVector(d_storage_num)
                        ;
587                 Dd_st_an[itr]     = makeVector(d_storage_an
                        (0,0));
588         Di_o[itr]           = makeVector(i_o(0,0));
589         D_tank[itr]         = makeVector(T_tank);
590         Dstiff_real[itr] = makeVector(stiff_real(0,0),
                stiff_real(1,1),stiff_real(2,2));
591                 Dstiff_des[itr]   = makeVector(stiff_des(0,0)
                        ,stiff_des(1,1),stiff_des(2,2));
592         DDamping[itr]       = makeVector(Da(0,0),Da(1,1),Da
                (2,2));
593                 DLambda[itr]        = makeVector(Lambda(0,0),
                        Lambda(1,1),Lambda(2,2));
594               //------------------------------------ update
```

```
595         q_old       = q;
596         qp_old      = qp;
597         qdprim      = qd;
598         xp_old      = xp;
599         x_old       = x;
600         xt_old      = xt;
601         Jprec       = J3dof;
602
603         t = t+CycTime;
604         itr = itr+1;
605
606     }
607
608     //
        ================================================================================

609     //============================ END LOOP
        ================================================
610     //
        ================================================================================


611
612     // stopping the robot...
613
614
615     ResultValue =       FRI->StopRobot();
616     if (ResultValue != EOK)     fprintf(stderr, "An error
        occurred during stopping the robot...\n");
617     else                        fprintf(stdout, "Robot
        successfully stopped.\n");
618
619
620     ofstream myfile;
621     myfile.open("mydata.m");
622     cout<<"open file..."<<endl;
623     myfile<<"\n s=["<<D_tank<<"];\n";
624     myfile<<"\n Stiff_des=["<<Dstiff_des<<"];\n";
625     myfile<<"\n Stiff_real=["<<Dstiff_real<<"];\n";
626     myfile.close();
627     cout<<"data wrote..."<<endl;
628
629     cout<<"data closed..."<<endl;
630     cout<<"\n
        ================================================================================\
        n";
631
632
```

```
633     delete FRI;
634
635
636     delete [] force;
637     delete [] POS;
638     delete [] Mass;
639     delete [] JacMatrix;
640
641     fprintf(stdout, "Object deleted...\n");
642
643     system("pause");
644     return(EXIT_SUCCESS);
645 }
646
647 /*
    ================================================================
    end main
    ================================================================
     */
648
649
650 /* ============================================ orientamento
    ========================*/
651
652
653 Vector<3> erroreQuaternioni(Matrix<4,4> Tbe, Matrix<4,4>
    Tbe_d)
654 {
655         double eta, eta_d;
656         Matrix<3> R = Tbe.slice<0,0,3,3>();
657         Vector<3> epsilon = r2quat(R, eta);
658         Matrix<3> R_d = Tbe_d.slice<0,0,3,3>();
659         Vector<3> epsilon_d = r2quat(R_d, eta_d);
660         Matrix<3> S = skew(epsilon_d);
661         Vector<3> eo = eta*epsilon_d-eta_d*epsilon-S*epsilon
                ;
662         return eo;
663 }
664 /*
    ----------------------------------------------------------------
    */
665
666 Vector<3> r2quat(Matrix<3,3> R_iniz, double &eta)
667 {
668         Vector<3> epsilon;
669         int iu, iv, iw;
670
671         if ( (R_iniz(0,0) >= R_iniz(1,1)) && (R_iniz(0,0) >=
```

```
            R_iniz(2,2)) )
672         {
673                 iu = 0; iv = 1; iw = 2;
674         }
675         else if ( (R_iniz(1,1) >= R_iniz(0,0)) && (R_iniz
               (1,1) >= R_iniz(2,2)) )
676         {
677                 iu = 1; iv = 2; iw = 0;
678         }
679         else
680         {
681                 iu = 2; iv = 0; iw = 1;
682         }
683
684         double r = sqrt(1 + R_iniz(iu,iu) - R_iniz(iv,iv) -
               R_iniz(iw,iw));
685         Vector<3> q = makeVector(0,0,0);
686         if (r>0)
687         {
688         double rr = 2*r;
689         eta = (R_iniz(iw,iv)-R_iniz(iv,iw)/rr);
690         epsilon[iu] = r/2;
691         epsilon[iv] = (R_iniz(iu,iv)+R_iniz(iv,iu))/rr;
692         epsilon[iw] = (R_iniz(iw,iu)+R_iniz(iu,iw))/rr;
693         }
694         else
695         {
696         eta = 1;
697         epsilon = makeVector(0,0,0);
698         }
699         return epsilon;
700 }
701
702 /*
       ------------------------------------------------------------
    */
703 Matrix<3> skew(Vector<3> v)
704 {
705         Matrix<3> S = Data(    0,          -v[2],     v[1],
                    //Skew-symmetric matrix
706                         v[2],          0,   -v[0],
707                         -v[1],      v[0],        0);
708         return S;
709 }
710
711 /*
       ------------------------------------------------------------
    */
```

Appendix D

Sponge Insertion Experiment C++ Code

```
#include <LWRJointPositionController.h>
#include <FastResearchInterface.h>
#include <friComm.h>
#include <math.h>
#include <time.h>
#include <TooN/TooN.h>
#include <TooN/GR_SVD.h>
#include <TooN/LU.h>
#include "simulation.h"
#include <iostream>
#include <fstream>
#include <string.h>
#include <stdio.h>
#include <ATI_FTsensor.h>

using namespace std;
using namespace TooN;
#define NUMBER_OF_JOINTS          7
#define RUN_TRAJECTORY 80.0
#ifndef PI
#define PI        3.14159265358979323846264338327950
#endif

#define EOK                                    0
#define RUN_TIME                    80.0// choose appropriate
    size for Matrix dat[]

ofstream results;

// ------------------------------------ function
    ------------------------------------

Vector <3> erroreQuaternioni(Matrix<4,4> Tbe, Matrix<4,4>
    Tbe_d);
```

```
31 Vector <3> r2quat(Matrix<3,3> R_iniz, double &eta);
32 Matrix <3> skew(Vector<3> v);
33 //
   ------------------------------------------------------------------------

34 //================================== Trajectory Function
   ====================
35 inline void linear_trajectory(float *p,float *dp,float *ddp,
   Matrix<3,1> xi,Matrix<3,1> xf,float ti,float tf,float tik
   ){
36     Matrix<3,1> a3=10.0/pow(tf-ti,3)*(xf-xi);
37     Matrix<3,1> a4=-15.0/pow(tf-ti,4)*(xf-xi);
38     Matrix<3,1> a5=6.0/pow(tf-ti,5)*(xf-xi);
39
40     for(int i=0; i<3; i++){
41         p[i]=xi(i,0)+a3(i,0)*pow(tik-ti,3)+a4(i,0)*pow(tik-
               ti,4)+a5(i,0)*pow(tik-ti,5);
42         dp[i]=3.0*a3(i,0)*pow(tik-ti,2)+4.0*a4(i,0)*pow(tik-
               ti,3)+5.0*a5(i,0)*pow(tik-ti,4);
43         ddp[i]=6.0*a3(i,0)*(tik-ti)+12.0*a4(i,0)*pow(tik-ti
               ,2)+20.0*a5(i,0)*pow(tik-ti,3);
44     }
45 }
46
47 //-------------------- Data matrices --------------------
48 const int datasize=1000*RUN_TIME/2;
49 Matrix<datasize,8>   dat=Zeros;
50 Matrix<datasize,7>   DTorque=Zeros;
51 Matrix<datasize,4>   Dks=Zeros;
52 Matrix <datasize,3> Dp_e,Dw,Dpd_e,Dx_pos,Dxd_pos,DKp,DLambda
   ,DLp,Dmass_real,Dstiff_real,Dstiff_des,Dmass_des,Derr,
   Dderr,Dgamma,Dx_des,Dxd_des,Dxdd_des,DDamping,DF=Zeros;
53 Matrix <datasize,1> D_tank,Dst,Dd_st_num,Dd_st_an,Dalfa,
   Dbeta,Di_o=Zeros;
54 Matrix <datasize,2> Dteta=Zeros;
55 //------------------------------------------------------------

56
57 //
   ========================================================================

58 int main(int argc, char *argv[]){
59
60     results.open("/home/prisma/Desktop/Ajinkya/denny_tanks/
         results.csv",ios::app);
61
62     int ResultValue=0;
63     float JointValuesInRad[NUMBER_OF_JOINTS];
64     float jp[NUMBER_OF_JOINTS];
```

```
    float tau[NUMBER_OF_JOINTS];
    float MTorque[NUMBER_OF_JOINTS];
    float InitialJointValuesInRad[NUMBER_OF_JOINTS
        ]={0.0,0.0,0.0,0.0,0.0,0.0,0.0};
    double t=0.0;
    double CycTime;

    float *force;
    force = new float[6];
    for(int j=0;j<6;j++)
        force[j] = 0.0;

    float *POS;
    POS= new float[12];
    for (int i=0; i<12;i++)     POS[i]=0.0;

    float **JacMatrix;
    JacMatrix= new float*[FRI_CART_VEC];
    for (int i=0; i<FRI_CART_VEC; i++){
        JacMatrix[i]=new float [NUMBER_OF_JOINTS];
    }
    for (int i=0; i<FRI_CART_VEC; i++){
        for (int j=0; j< NUMBER_OF_JOINTS ;j++)
            JacMatrix[i][j]=0.0;
    }

    float **Mass;
    Mass= new float*[NUMBER_OF_JOINTS];
    for (int i=0; i<NUMBER_OF_JOINTS; i++){
        Mass[i]=new float [NUMBER_OF_JOINTS];
    }
    for (int i=0; i<NUMBER_OF_JOINTS; i++){
        for (int j=0; j< NUMBER_OF_JOINTS ;j++) Mass[i][j
            ]=0.0;
    }
    //
     ------------------------------------------------------------

        FastResearchInterface *FRI;
        FRI = new FastResearchInterface("/home/prisma/
            Desktop/FRILibrary/etc/980039-FRI-Driver.init");

    //-----------------------------------------init FT sensor
     -----------------------------------------------------//
    ATI_FTsensor *FTsensor;
    FTsensor= new ATI_FTsensor((char*)"143.225.169.201"); //
        ATI FT sensor IP
```

```
106     /*Set a software Bias for Force and Torque*/
107     FTsensor->setBias();
108     // ---------------------------------------- Variables
          ----------------------------------------//
109
110     Vector<3> x,
111     x_old,
112     xp,xpp,
113     xi_old,
114     m_diag,
115     xp_old,
116     kpv_diag,
117     kp_diag,
118     e,
119     edot,
120     ep,
121     eo,
122     epp,
123     eop,
124     xpi,
125     f,
126     f_old,
127     fd,
128     xi=Zeros;
129
130     Matrix<3,1> err3d,
131     p_e,
132     F3_ext,
133     errpos,
134     pd_e,errvel,
135     pdd_e=Zeros;
136
137     //giunti
138     Vector<7> q,
139     qi,
140     qp,
141     qpi,
142     q_old,
143     qpp,
144     rid,
145     qp_old,
146     q_plot,
147     q2_o,
148     q_o=Zeros;
149
150     Matrix<7,1> qd,qdF,
151     qdd,
152     qdprim=Zeros;
```

```
153
154     Matrix <3,3> Re,
155     S_pe,
156     Da,
157     Lambda,
158     LambdaDes,
159     LambdaDes_inv,
160     Lambda_inv,
161     K,
162     stiff,
163     Rif,
164     Kf,
165     KI,
166     Mcost,
167     Mvar,
168     Rd=Zeros;
169
170     Matrix <4,4> Tbe,
171     Tbe_d=Zeros;
172
173     Vector <6> fsens,
174     e_e,
175     e_dot,
176     fsens_f = Zeros;
177
178     Matrix <7,3> Js,
179     Jso=Zeros;
180     Matrix <7,6> J_s;
181     Matrix <6,7> J=Zeros;
182     Matrix <6,6> A,B,T=Zeros;
183     Matrix <3,7> J3dof,Jprec,Jdot,Jo=Zeros;
184     Matrix <7,7> M=Zeros;
185     Matrix <7,7> M_inv=Identity;
186
187
188     B=Data(1,0,0,0,0,0,
189            0,1,0,0,0,0,
190            0,0,1,0,0,0,
191            0,0,0,0,0,1,
192            0,0,0,0,1,0,
193            0,0,0,1,0,0);
194     float Fext[6];
195
196
197     Vector <3> Kst,Kstf;
198     double phi1,beta1,gamma1,y_1,y1=0;
199     double phi2,beta2,gamma2,y_2,y2=0;
200     double phi3,beta3,gamma3,y_3,y3=0;
```

```
201     double phi1f,beta1f,gamma1f,y_1f,y1f=0;
202     double phi2f,beta2f,gamma2f,y_2f,y2f=0;
203     double phi3f,beta3f,gamma3f,y_3f,y3f=0;
204     double V1=5;
205     double V2=5;
206     double V3=5;
207
208     double V1f=5;
209     double V2f=5;
210     double V3f=5;
211     double kx=10;
212     double ky=10;
213     double kz=10;
214     double kxf=10;
215     double kyf=10;
216     double kzf=10;
217     double kzf_old=10;
218     double dkzf=10;
219
220
221
222     //========== VARIABLES TANK ==============================
223     double upperT = 0.05;              //upper bound tank
224     double lowerT = 0.0;              //lower bound continuit
          tank
225     double xt  = 0.001;
226     double xt_old = xt;
227     double var=0.0;
228
229 double stiff_d = 12;
230 double stiff_a = 12;
231 double stiff_e = 0;
232 double stiff_deri=0;
233
234     double T_tank=0.001;
235     double sigma=0.01;
236     Matrix<3,1> w = Zeros;
237     Matrix<1,1> dxt=Zeros;
238     float d_storage_num=0.0;
239     Matrix<1,1> temp=Zeros,
240     T_temp=Zeros;
241     Matrix<1,1> ds,storage,i_o,s_temp,storage_old    = Zeros;
242     Matrix <1,1> d_storage_an = Zeros;
243     Matrix <3,3> stiff_var=Zeros,
244                  stiff_var_des=Zeros,
245                 stiff_cost=Zeros,
246                 stiff_des=Zeros,
247                 stiff_real=Zeros;
```

```
248     //==== ============================
249 Matrix<2,1> phi,teta,gamma_stima=Zeros;
250 Matrix<2,2>  GUAD=Zeros;
251     Matrix<1,1>  beta_stima,Fz,beta_stima_inv,modello_forza=
            Zeros;
252     Matrix<1,1>  temp_stima=Data(1.0);
253     GUAD=Data(1.2,0.0,
254               0.0,1.2);
255
256     //============================
257     float tiA,tAB,tBC,tCD,tDE,tEF,tFi,t0,tf,*p_,*dp_,*ddp_;
258     p_=new float[3];
259     dp_=new float[3];
260     ddp_=new float[3];
261     Vector<3> x_des,xd_des,xdd_des,delta_x,delta_dx=Zeros,
            x_init;
262     Matrix<3,1> pA,pB,pC,c2,pD,pE,pF;
263     int k;
264     tiA=10.0;
265     tAB=13.0;
266     tBC=31.0;
267     tCD=47.0;
268     tDE=55.0;
269     t0=15;
270     tf=31;
271     pA=Data(-0.671, 0.431,  0.042);
272     pB=Data(-0.67, 0.43,  0.042);
273     pC=Data(-0.67, 0.43,  0.0);
274     pD=Data(-0.67, 0.43,  0.05);
275     pE=Data(-0.5492, 0.0064,  0.4562);
276     //
            ------------------------------------------------------------
277
278     //========  M, D, K  ============================
279
280     LambdaDes = Data(10.0,0.0,0.0,
281                      0.0,10.0,0.0,
282                      0.0,0.0,10.0);
283     LambdaDes_inv=Data(1/LambdaDes(0,0),0.0,0.0,
284                        0.0,1/LambdaDes(1,1),0.0,
285                        0.0,0.0,1/LambdaDes(2,2));
286
287         Da=Data(4.0,0.0,0.0,
288                 0.0,4.0,0.0,
289                 0.0,0.0,4.0);
290
291     kp_diag=makeVector(12,12,12);
```

```
292     stiff_cost=Data(kp_diag[0],0.0,0.0,
293                     0.0,kp_diag[1],0.0,
294                     0.0,0.0,kp_diag[2]);
295
296     //=============================================================
297
298     fprintf(stdout, "RobotJointPositionController object
            created.\n Starting the robot...\n");
299     ResultValue=FRI-> StartRobot(FastResearchInterface::
            JOINT_POSITION_CONTROL,120);
300
301     if (ResultValue == EOK) fprintf(stdout, "Robot
            successfully started.\n");
302     else                    fprintf(stderr, "ERROR, could
            not start robot: %s\n", strerror(ResultValue));
303
304     fprintf(stdout, "Current system state:\n%s\n", FRI->
            GetCompleteRobotStateAndInformation());
305     CycTime=FRI->GetFRICycleTime();
306     FRI-> GetMeasuredJointPositions(InitialJointValuesInRad)
            ;
307
308
309     cout<< "\nThe cycle time is "<<CycTime<<endl;
310
311     //-----------------------------------------init clik
            -----------------------------------------//
312
313     qi=makeVector(InitialJointValuesInRad[0],
            InitialJointValuesInRad[1],InitialJointValuesInRad
            [2],InitialJointValuesInRad[3],
            InitialJointValuesInRad[4],InitialJointValuesInRad
            [5],InitialJointValuesInRad[6]);
314
315     q_old=qi;
316     qp=Zeros;
317     qp_old=qp;
318
319     FRI->GetMeasuredCartPose(POS);
320
321     Re=Data(POS[0], POS[1], POS[2],
322             POS[4], POS[5], POS[6],
323             POS[8], POS[9], POS[10]);
324
325     Tbe_d=Data(POS[0], POS[1], POS[2],POS[3],
326                POS[4], POS[5], POS[6], POS[7],
327                POS[8], POS[9], POS[10],POS[11],
328                0.0,0.0,0.0,1.0);
```

```
329
330     xi=makeVector(POS[3],POS[7],POS[11]);
331     x_init=makeVector(POS[3],POS[7],POS[11]);
332     xi_old=xi;
333     x=xi;
334     x_old=x;
335
336     xp=Zeros;
337     xp_old=xp;
338
339     FRI->GetCurrentJacobianMatrix(JacMatrix);
340
341     for (int i=0;i<FRI_CART_VEC;i++){
342         for (int j=0;j<NUMBER_OF_JOINTS;j++)
343             J[i][j]=JacMatrix[i][j];
344     }
345
346     //------------------------------------------ jacobian init
          ------------------------------------------
347
348     A=Data(Re(0,0),Re(0,1),Re(0,2), 0      ,0      ,0,
349            Re(1,0),Re(1,1),Re(1,2), 0      ,0      ,0,
350            Re(2,0),Re(2,1),Re(2,2), 0      ,0      ,0,
351            0      ,0      ,0      ,Re(0,0),Re(0,1),Re(0,2),
352            0      ,0      ,0      ,Re(1,0),Re(1,1),Re(1,2),
353            0      ,0      ,0      ,Re(2,0),Re(2,1),Re(2,2));
354
355
356     J=A*B*J;
357
358     J3dof=J.slice(0,0,3,7);
359     Jprec=J3dof;
360
361
362     //------------------------------------------  init control
          ------------------------------------------
363
364     double daz,day,dax=0.0;
365     double max,may,maz=0.0;
366
367
368     Vector<7> Pred=Zeros;
369
370
371
372     //
          ==========================================================
```

```
373     //============================== START LOOP
            ==========================================
374     //
            ================================================================================


375
376     int itr=0;
377     cout<<" xi= "<<xi<<endl;
378     while (t<=RUN_TIME){
379
380
381         FRI->WaitForKRCTick();
382
383         if (!FRI->IsMachineOK())
384         {
385             fprintf(stderr, "ERROR, the machine is not ready
                    anymore.\n");
386             break;
387         }
388
389
390         //------------------------------------------ VARIAZIONE
                STIFFNESS ------------------------------------------
391
392         if(t<=tAB)   var=0;
393         if(t>tAB && t<tAB+1) var=0.5*(1+cos((tAB+1-t)*PI/(1)
                ));
394         if(t>=tAB+1)   var = 1;
395
396         if(t>tAB+1 && t<=tCD)   var=1;
397         if(t>tCD && t<=tCD+1) var=0.5*(1+cos((tCD-t)*PI/(1))
                );
398         if(t>tCD+1)   var = 0;
399
400
401         kpv_diag=makeVector(2000*var,2000*var,2000*var);
402
403         stiff_var_des=Data(0.0,          0.0,
                0.0,
404                            0.0,   10*sin(PI*t/10),
                                          0.0,
405                            0.0,          0.0,      0.0);
406
407
408         stiff_des=stiff_cost + stiff_var_des;
409
410
411
```

```
412         x_des= x_init + makeVector(0,0.1*sin(0.1*t),0);
413         xd_des=makeVector(0,0.1*0.1*cos(0.1*t),0);
414         xdd_des=makeVector(0,-0.1*0.1*0.1*sin(0.1*t),0);
415
416         //recieve the pos, jacobian, mass at each cycle time
417         FRI->GetMeasuredCartPose(POS);
418         FRI-> GetMeasuredJointPositions(jp);
419         q=makeVector(jp[0],jp[1],jp[2],jp[3],jp[4],jp[5],jp
                [6]);
420         qpi=(q-q_old)/CycTime;
421         qd=Data(qpi[0],qpi[1],qpi[2],qpi[3],qpi[4],qpi[5],
                qpi[6]);
422         qdF   = 0.1*qd+(1.0-0.1)*qdF;
423         pd_e = J3dof*qdF;
424         Re=Data(POS[0], POS[1], POS[2],
425                                 POS[4], POS[5], POS[6],
426                                 POS[8], POS[9], POS[10]);
427
428         FRI->GetCurrentJacobianMatrix(JacMatrix);
429         for (int i=0;i<FRI_CART_VEC;i++){
430              for (int j=0;j<NUMBER_OF_JOINTS;j++)     J[i][j]=
                    JacMatrix[i][j];
431         }
432
433         A=Data(Re(0,0),Re(0,1),Re(0,2), 0.0      ,0.0
                ,0.0,
434              Re(1,0),Re(1,1),Re(1,2), 0.0      ,0.0
                    ,0.0,
435              Re(2,0),Re(2,1),Re(2,2), 0.0      ,0.0
                    ,0.0,
436              0.0      ,0.0      ,0.0      ,Re(0,0),Re(0,1)
                    ,Re(0,2),
437              0.0      ,0.0      ,0.0      ,Re(1,0),Re(1,1),
                    Re(1,2),
438              0.0      ,0.0      ,0.0      ,Re(2,0),Re(2,1)
                    ,Re(2,2));
439
440         J=A*B*J;
441         J3dof=J.slice(0,0,3,7);
442         Jdot=(J3dof-Jprec)/CycTime;
443         xi=makeVector(POS[3],POS[7],POS[11]);
444         xpi=J3dof*qp;
445         //
                ------------------------------------------------------------
446
447
448         //----------------------martice M
```

```
449         FRI-> GetCurrentMassMatrix(Mass);
450         for (int i=0; i<NUMBER_OF_JOINTS;i++){
451             for (int j=0; j< NUMBER_OF_JOINTS ;j++)  M[i][j
                ]=Mass[i][j];
452         }
453
454         LU<7> lu_M(M);
455         M_inv=lu_M.get_inverse();
456
457         //------------------------ Lambda ------------
458         LU<3> lu_J(J3dof*M_inv*J3dof.T());
459         Lambda      = lu_J.get_inverse();
460         Lambda_inv = J3dof*M_inv*J3dof.T();
461         //-------------------------------------------
462
463
464
465         S_pe=Data(0.0,-xi[2],xi[1],
466                   xi[2],0.0,-xi[0],
467                   -xi[1],xi[0],0.0);
468         S_pe=S_pe*Re;
469
470         T=Data(Re(0,0),Re(0,1),Re(0,2),0.0,0.0,0.0,
471                Re(1,0),Re(1,1),Re(1,2),0.0,0.0,0.0,
472                Re(2,0),Re(2,1),Re(2,2),0.0,0.0,0.0,
473                S_pe(0,0),S_pe(0,1),S_pe(0,2),Re(0,0),Re(0,1)
                   ,Re(0,2),
474                S_pe(1,0),S_pe(1,1),S_pe(1,2),Re(1,0),Re(1,1)
                   ,Re(1,2),
475                S_pe(2,0),S_pe(2,1),S_pe(2,2),Re(2,0),Re(2,1)
                   ,Re(2,2));
476
477         Tbe=Data(POS[0], POS[1], POS[2],POS[3],
478                                  POS[4], POS[5], POS[6],POS
                                     [7],
479                                  POS[8], POS[9], POS[10],POS
                                     [11],
480                  0.0,0.0,0.0,1.0);
481
482         //------------------------------------------ force sensor
                ------------------------------------
483         FTsensor->getFTData(force);
484
485         for(int i=0;i<6;i++)  fsens[i]=force[i];
486
487         fsens=T*fsens;
488
```

```
489                                     if(t>=5 && t<=15)
490         {f=makeVector(0,-1,0);}
491                                     else if(t>=26 && t<=36)
492         {f=makeVector(0,1,0);}
493                                     else
494         {f=makeVector(0,0,0);}
495         //
              ------------------------------------------------------------------------


496
497         //=========================================== TANK
              ========================================
498         errpos=Data(-x_des[0]+xi[0],-x_des[1]+xi[1],-x_des
              [2]+xi[2]);
499         T_tank=xt;
500
501
502         if(T_tank<=upperT)
503                                     {
504                                       if(T_tank<=upperT &&
                                            T_tank>upperT*0.9) {
                                            sigma = 0.5*(1+cos((
                                            T_tank-upperT*0.9)*PI
                                            /(0.1*upperT)));cout<<"
                                            inif= "<<endl;}
505           else {sigma =1;}
506         }
507         else
508         {sigma = 0;}
509
510 double temp_kder = PI*cos(PI*t/10);
511 double beta_k =0;
512 double gamma_k=0;
513
514 if(temp_kder>0) beta_k = 1;
515 else beta_k=1;
516
517 if(T_tank>=0.0 && T_tank<0.05) gamma_k=1;
518 else gamma_k=0;
519
520 stiff_e = 12+10*sin(PI*t/10) - stiff_a;
521
522 stiff_deri = (PI*cos(PI*t/10) + 20*stiff_e)*gamma_k;
523 Matrix<3,3> stiff_deri_mat = Data(0,0,0,
524
```

```
525                                         0,0,0);
526
527         dxt = (sigma)*(errvel.T()*Da*errvel) - 0.5*beta_k*
                errpos.T()*stiff_deri_mat*errpos;
528         xt =xt_old + CycTime*dxt(0,0);
529
530                                         if(xt>0.05) xt = 0.049;
531
532
533         GR_SVD<3,7> J_svd(J3dof);
534         Js = J_svd.get_pinv();
535         e=x-xi;
536         edot=xp-xpi;
537         errvel=Data(-xd_des[0]+xpi[0],-xd_des[1]+xpi[1],-
                xd_des[2]+xpi[2]);
538         delta_x= x-x_des;
539         delta_dx=xp-xd_des;
540
541                 //-------------------------------------controllo
                        ammettenza ------------------------------------
542         xpp = LambdaDes_inv*( -Da*delta_dx -stiff_a*delta_x
                + f );
543         xp  = xp_old+CycTime*xpp;
544         x   = x_old+CycTime*xp_old;
545
546 results<<xi[1]-x_init[1]<<","<<x_des[1]-x_init[1]<<","<<
    stiff_a<<","<<12+10*sin(PI*t/10)<<","<<xt<<","<<sigma<<",
    "<<gamma_k<<endl;
547
548 stiff_a = stiff_a + stiff_deri*CycTime;
549
550                 //-------------------------------------- redundancy
                        ---------------------------------------
551
552         eo=erroreQuaternioni(Tbe,Tbe_d);
553         Jo=Data(J(3,0),J(3,1),J(3,2),J(3,3),J(3,4),J(3,5),J
                (3,6),
554                 J(4,0),J(4,1),J(4,2),J(4,3),J(4,4),J(4,5),J
                    (4,6),
555                 J(5,0),J(5,1),J(5,2),J(5,3),J(5,4),J(5,5),J
                    (5,6) );
556
557         GR_SVD<3,7> Jor_svd(Jo);
558         Jso=Jor_svd.get_pinv();
559         eo=5.0*eo;
560         q_o=Jso*eo;
561         qpp=Js*(xpp+10.0*edot + 50.0*e -Jdot*qp) + (Identity
                (7) - Js*J3dof) * (-2.0*qp+1.7*q_o);
```

```
562
563
564         //------------------------------------ clik
                ------------------------------------
565         qp=qp_old + CycTime*qpp;
566         q=q_old + CycTime*qp_old;
567
568         //======================================== command
                ========================================
569         for(int i=0;i<NUMBER_OF_JOINTS;i++)   jp[i]=q[i];
570         FRI->SetCommandedJointPositions(jp);//
                JointValuesInRad
571         //
                ================================================================================


572
573
574         i_o = pd_e.T()*F3_ext;
575
576
577         //------------------------------------ write Matrices
                ------------------------------------
578         dat[itr]        = makeVector(t,jp[0],jp[1],jp[2],jp
                [3],jp[4],jp[5],jp[6]);
579         Dp_e[itr]       = makeVector(xi[0],xi[1],xi[2]);
580         Dpd_e[itr]      = makeVector(xpi[0],xpi[1],xpi[2]);
581         Dx_des[itr]     = makeVector(x_des[0],x_des[1],x_des
                [2]);
582         Dxd_des[itr]    = makeVector(xd_des[0],xd_des[1],
                xd_des[2]);
583
584         DF[itr]         = makeVector(f[0],f[1],f[2]);
585         Dst[itr]            = makeVector(storage(0,0));
586                 Dd_st_num[itr]    = makeVector(d_storage_num)
                        ;
587                 Dd_st_an[itr]     = makeVector(d_storage_an
                        (0,0));
588         Di_o[itr]           = makeVector(i_o(0,0));
589         D_tank[itr]         = makeVector(T_tank);
590         Dstiff_real[itr] = makeVector(stiff_real(0,0),
                stiff_real(1,1),stiff_real(2,2));
591                 Dstiff_des[itr]   = makeVector(stiff_des(0,0)
                        ,stiff_des(1,1),stiff_des(2,2));
592         DDamping[itr]       = makeVector(Da(0,0),Da(1,1),Da
                (2,2));
593                 DLambda[itr]        = makeVector(Lambda(0,0),
                        Lambda(1,1),Lambda(2,2));
594                 //------------------------------------ update
```

```
595             q_old       = q;
596             qp_old      = qp;
597             qdprim      = qd;
598             xp_old      = xp;
599             x_old       = x;
600             xt_old      = xt;
601             Jprec       = J3dof;
602
603             t = t+CycTime;
604             itr = itr+1;
605
606         }
607
608         //
            ================================================================================
609         //============================ END LOOP
            ================================================
610         //
            ================================================================================
611
612         // stopping the robot...
613
614
615         ResultValue =       FRI->StopRobot();
616         if (ResultValue != EOK)     fprintf(stderr, "An error
               occurred during stopping the robot...\n");
617         else                        fprintf(stdout, "Robot
               successfully stopped.\n");
618
619
620         ofstream myfile;
621         myfile.open("mydata.m");
622         cout<<"open file..."<<endl;
623         myfile<<"\n s=["<<D_tank<<"];\n";
624         myfile<<"\n Stiff_des=["<<Dstiff_des<<"];\n";
625         myfile<<"\n Stiff_real=["<<Dstiff_real<<"];\n";
626         myfile.close();
627         cout<<"data wrote..."<<endl;
628
629         cout<<"data closed..."<<endl;
630         cout<<"\n
               ==========================================================================\
               n";
631
632
```

```
633     delete FRI;
634
635
636     delete [] force;
637     delete [] POS;
638     delete [] Mass;
639     delete [] JacMatrix;
640
641     fprintf(stdout, "Object deleted...\n");
642
643     system("pause");
644     return(EXIT_SUCCESS);
645 }
646
647 /*
    ================================================================
    end main
    ================================================================
     */
648
649
650 /* ============================================ orientamento
    ========================*/
651
652
653 Vector<3> erroreQuaternioni(Matrix<4,4> Tbe, Matrix<4,4>
    Tbe_d)
654 {
655         double eta, eta_d;
656         Matrix<3> R = Tbe.slice<0,0,3,3>();
657         Vector<3> epsilon = r2quat(R, eta);
658         Matrix<3> R_d = Tbe_d.slice<0,0,3,3>();
659         Vector<3> epsilon_d = r2quat(R_d, eta_d);
660         Matrix<3> S = skew(epsilon_d);
661         Vector<3> eo = eta*epsilon_d-eta_d*epsilon-S*epsilon
                ;
662         return eo;
663 }
664 /*
    ----------------------------------------------------------------
    */
665
666 Vector<3> r2quat(Matrix<3,3> R_iniz, double &eta)
667 {
668         Vector<3> epsilon;
669         int iu, iv, iw;
670
671         if ( (R_iniz(0,0) >= R_iniz(1,1)) && (R_iniz(0,0) >=
```

```
            R_iniz(2,2)) )
672         {
673                 iu = 0; iv = 1; iw = 2;
674         }
675         else if ( (R_iniz(1,1) >= R_iniz(0,0)) && (R_iniz
               (1,1) >= R_iniz(2,2)) )
676         {
677                 iu = 1; iv = 2; iw = 0;
678         }
679         else
680         {
681                 iu = 2; iv = 0; iw = 1;
682         }
683
684         double r = sqrt(1 + R_iniz(iu,iu) - R_iniz(iv,iv) -
               R_iniz(iw,iw));
685         Vector<3> q = makeVector(0,0,0);
686         if (r>0)
687         {
688         double rr = 2*r;
689         eta = (R_iniz(iw,iv)-R_iniz(iv,iw)/rr);
690         epsilon[iu] = r/2;
691         epsilon[iv] = (R_iniz(iu,iv)+R_iniz(iv,iu))/rr;
692         epsilon[iw] = (R_iniz(iw,iu)+R_iniz(iu,iw))/rr;
693         }
694         else
695         {
696         eta = 1;
697         epsilon = makeVector(0,0,0);
698         }
699         return epsilon;
700 }
701
702 /*
       ------------------------------------------------------------
    */
703 Matrix<3> skew(Vector<3> v)
704 {
705         Matrix<3> S = Data(    0,          -v[2],     v[1],
                    //Skew-symmetric matrix
706                         v[2],          0,   -v[0],
707                         -v[1],      v[0],        0);
708         return S;
709 }
710
711 /*
       ------------------------------------------------------------
    */
```

Bibliography

[1] Jonas Buchli et al. "Learning variable impedance control". In: *The International Journal of Robotics Research* 30.7 (2011), pp. 820–833.

[2] Sylvain Calinon, Irene Sardellitti, and Darwin G Caldwell. "Learning-based control strategy for safe human-robot interaction exploiting task and robot redundancies". In: *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*. IEEE. 2010, pp. 249–254.

[3] Carlos A. Cardenas. *Development of a safety-aware intrinsically passive controller for a multi-DOF manipulator*. Sept. 2017. URL: `http://essay.utwente.nl/73878/`.

[4] Vincent Duindam and Stefano Stramigioli. "Port-based asymptotic curve tracking for mechanical systems". In: *European Journal of Control* 10.5 (2004), pp. 411–420.

[5] Federica Ferraguti, Cristian Secchi, and Cesare Fantuzzi. "A tank-based approach to impedance control with variable stiffness". In: *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE. 2013, pp. 4948–4953.

[6] Fanny Ficuciello, Luigi Villani, and Bruno Siciliano. "Variable impedance control of redundant manipulators for intuitive human–robot physical interaction". In: *IEEE Transactions on Robotics* 31.4 (2015), pp. 850–863.

[7] Antonio Franchi et al. "Bilateral teleoperation of groups of mobile robots with time-varying topology". In: *IEEE Transactions on Robotics* 28.5 (2012), pp. 1019–1033.

[8] Michel Franken et al. "Bilateral telemanipulation with time delays: A two-layer approach combining passivity and transparency". In: *IEEE Transactions on Robotics* 27.4 (2011), pp. 741–756.

[9] Denny Geremia. "Energy-based approaches to the control of the physical interaction of robot manipulators". MA thesis. University of Naples Federico II, 2017.

[10] Neville Hogan. "Impedance control: An approach to manipulation". In: *American Control Conference, 1984*. IEEE. 1984, pp. 304–313.

[11] Oussama Khatib. "A unified approach for motion and force control of robot manipulators: The operational space formulation". In: *IEEE Journal on Robotics and Automation* 3.1 (1987), pp. 43–53.

[12] Klas Kronander and Aude Billard. "Online learning of varying stiffness through physical human-robot interaction". In: *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. Ieee. 2012, pp. 1842–1849.

[13] Christian Ott, Ranjan Mukherjee, and Yoshihiko Nakamura. "Unified impedance and admittance control". In: *Robotics and Automation (ICRA), 2010 IEEE International Conference on*. IEEE. 2010, pp. 554–561.

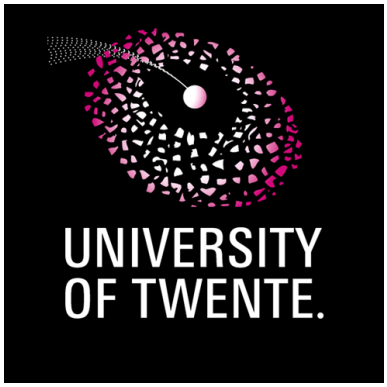
[14] Cristian Secchi, Stefano Stramigioli, and Cesare Fantuzzi. "Position drift compensation in port-hamiltonian based telemanipulation". In: *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*. IEEE. 2006, pp. 4211–4216.

[15] Cristian Secchi et al. "Bilateral teleoperation of a group of UAVs with communication delays and switching topology". In: *Robotics and Automation (ICRA), 2012 IEEE International Conference on*. IEEE. 2012, pp. 4307–4314.

Impedance Parameters Estimation for a Variable Impedance Controlled Robotic Manipulator

By

Ajinkya Arun Bhole



Faculty of Electrical Engineering, Mathematics and
Computer Science (EEMCS)
University of Twente, The Netherlands

This report is submitted to University of Twente in accordance with the requirements of the degree of MSc Systems and Control.

January 2018

Impedance Parameters Estimation for a Variable Impedance Controlled Robotic Manipulator

Abstract

This aim of this work is to estimate the impedance parameters, namely the damping and stiffness of a variable impedance controlled robotic manipulator.
The estimation is performed using a Constrained Extended Kalman Filter (CEKF). As compared to other non-linear estimation techniques Extended Kalman Filter shows a superiority with respect to speed of execution. This speed of execution was a major requirement as this estimation is to be used for an online variable impedance control task wherein the impedance law is varied according to the current estimated impedance parameters.
Experiments were performed on a 7-DOF KUKA LWR by giving the position of the end-effector as the measurement and the external force applied on it as an input to the CEKF. Without giving explicit information of the dynamics of the variable impedance law of the manipulator, the EKF was able to quite appreciably able to track the real impedance. The performance of the estimator however deteriorates as the real impedance variation too non-linear.

Contents

List of Figures

Chapter 1

Introduction

In various dynamic systems, online estimation of the process states forms an essential part of monitoring the process conditions and state-feedback based control. This is because processes are affected by various random disturbances and it is difficult to obtain regular and noise-free online measurements of many process states, such as concentrations, compositions, etc. The Bayesian state estimation approach is widely used because it provides a systematic and general approach to handle the effect of various random uncertainties on the process states and measurements. Bayesian state estimation algorithms use a first-principles based dynamic model and the statistical properties of the random disturbances and measurements to obtain the posterior distribution of the state estimates.

The Kalman Filter introduced by Rudolf E. Kalman [4] is a widely used Bayesian estimation technique. This technique works best in-case the model of the system under consideration is linear. In case the model is non-linear, which is generally the case, there is another approach with almost the same computational complexity, but with better performance. That approach is the Extended Kalman filter (EKF) [1]. In EKF the state distribution is approximated by a Gaussian Random Variable (GRV), which is then propogated through the first order linearization of the non-linear system. As compared to other non-linear estimation techniques, EKF provides a better performance with respect to execution time which is a necessary factor during an online estimation task.

The goal of this work is to estimate the impedance parameters of a variable impedance controlled robotic manipulator. The work in this report can be useful for example in a manually guided surgery task wherein the impedance parameters of the arm of the human operator can be estimated and based on this estimation, the impedance law of the manipulator can be varied so as to make the surgery easier for the operator.

The remainder of the report is organized as follows. Chapter 2 briefly introduces the background on bayesian online estimation. Chapter 3 introduces the Extended Kalman Filter. Chapter 4 introduces an extended version of EKF, the Constrained Extended Kalman Filter (CEKF). Chapter 5 briefly introduces the idea of Impedance Control. Chapter 6 shows the results of the CEKF for the estimation of impedance parameters for a Variable Impedance control task. Finally, concluding remarks are drawn in Chapter 7.

Chapter 2

Optimal Online Estimation of Linear Gaussian Systems: The Kalman Filter

Online estimation is the estimation of the present state using all the measurements that are available, i.e. all measurements up to the present time. This chapter sets up a framework for the online estimation of the states of time-discrete processes. Of course, most physical processes evolve in the continuous time. Nevertheless, it will be assumed that these systems can be described adequately by a model where the continuous time is reduced to a sequence of specific times.

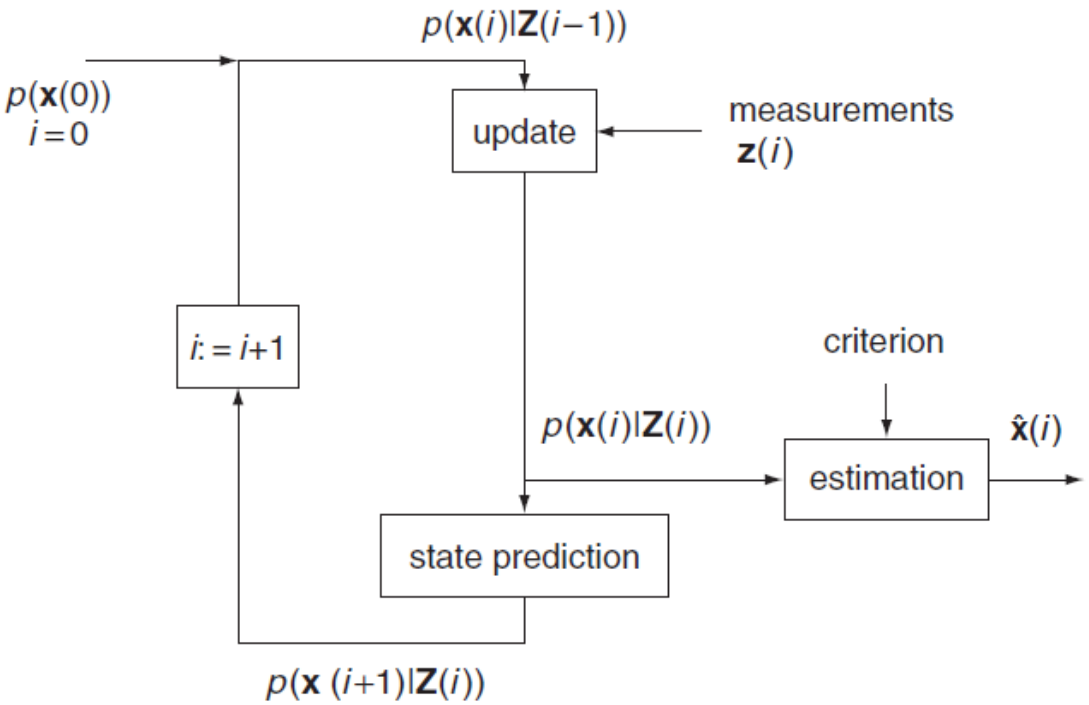


Figure 2.1: An overview of online estimation [6]

Figure 2.1 presents an overview of the scheme for the online estimation of the state. The connotation of the phrase online is that for each time index i an estimate $\hat{x}(i)$ of $x(i)$ is produced based on $Z(i)$, i.e. based on all measurements that are

available at that time. The crux of optimal online estimation is to maintain the posterior density $p(x(i) \mid Z(i))$ for running values of i. This density captures all the available information of the current state $x(i)$ after having observed the current measurement and all previous ones.

Most literature in optimal estimation in dynamic systems deals with the particular case in which both the state model and the measurement model are linear, and the disturbances are Gaussian (the linear-Gaussian systems). The main reason for the popularity is the mathematical tractability of this case.

The state model is said to be linear if the transition from one state to the next can be expressed by a so-called linear system equation (or: linear state equation, linear plant equation, linear dynamic equation):

$$x(i+1) = F(i)x(i) + L(i)u(i) + w(i) \tag{2.1}$$

$$y(i) = H(i)x(i) + v(i) \tag{2.2}$$

$F(i)$ is the system matrix. The vector $u(i)$ is the control vector (input vector). $L(i)$ is the gain matrix. $w(i)$ is the process noise (system noise, plant noise). The process noise represents the unknown influences on the system, for instance, formed by disturbances from the environment. The process noise can also represent an unknown input/control signal. Sometimes process noise is also used to take care of modelling errors. The general assumption is that the process noise is a white random sequence with normal distribution and has the following properties:

$$E[w(i)] = 0 \tag{2.3}$$

$$E[w(i)w^T(j)] = C_w(i)\delta(i,j) \tag{2.4}$$

$C_w(i)$ is the covariance matrix of $w(i)$. Since w(i) is supposed to have a normal distribution with zero mean, $C_w(i)$ defines the density of $w(i)$ in full.

The update steps, i.e. the determination of $p((x(i) \mid Z(i))$ shown in Figure 2.1 is given as follows:

$$\hat{z}(i) = H(i)\bar{x}(i \mid i-1) \tag{2.5}$$

$$S(i) = H(i)C(i \mid i-1)H^T(i) + C_v(i) \tag{2.6}$$

$$K(i) = C(i \mid i-1)H^T(i)S^{-1}(i) \tag{2.7}$$

$$\bar{x}(i \mid i) = \bar{x}(i \mid i-1) + K(i)(z(i) - \hat{z}(i)) \tag{2.8}$$

$$C(i \mid i) = C(i \mid i-1) - K(i)S(i)K^T(i) \tag{2.9}$$

The interpretation is as follows: $\hat{z}(i)$ is the predicted measurement. It is an unbiased estimate of $z(i)$ using all information from the past. The so-called innovation matrix $S(i)$ represents the uncertainty of the predicted measurement. The uncertainty is due to two factors: the uncertainty of $x(i)$ as expressed by $C(i \mid i-1)$, and the uncertainty due to the measurement noise $v(i)$ as expressed by $C_v(i)$. The matrix $K(i)$ is the Kalman gain matrix. This matrix has large, when $S(i)$ is small and $C(i \mid i-1)H^T(i)$ is large, that is, when the measurements are relatively accurate. When this is the case, the values in the error covariance matrix $C(i \mid i)$ will be much smaller than $C(i \mid i-1)$.

The prediction, i.e. the determination of $p(x(i+1) \mid Z(i))$ given $p(x(i) \mid Z(i))$, boils down to finding out how the expectation $x(i \mid i)$ and the covariance matrix $C(i \mid i)$ propagate to the next state. We thus have:

$$\bar{x}(i+1 \mid i) = F(i)\bar{x}(i \mid i) + L(i)u(i) \tag{2.10}$$

$$C(i+1 \mid i) = F(i)C(i \mid i)F^T(i) + C_w(i) \tag{2.11}$$

The above presented recursive equations are generally referred to as the discrete Kalman filter (DKF).

Chapter 3

Sub-Optimal Solution for Online Estimation of Non-Linear Systems: The Extended Kalman Filter

The general case of nonlinear systems and nonlinear measurement functions can be given as follows:

$$x(i+1) = f(x(i), u(i), i) + w(i) \tag{3.1}$$

$$y(i) = h(x(i), i) + v(i) \tag{3.2}$$

The vector $f(.,.,.)$ is a nonlinear, time variant function of the state x(i) and the control vector u(i).

Any Gaussian random vector that undergoes a linear operation retains its Gaussian distribution. A linear operator only affects the expectation and the covariance matrix of that vector. This property is the basis of the Kalman filter. It is applicable to linear-Gaussian systems, and it permits a solution that is entirely expressed in terms of expectations and covariance matrices. However, the property does not hold for nonlinear operations. In nonlinear systems, the state vectors and the measurement vectors are not Gaussian distributed, even though the process noise and the measurement noise might be. Consequently, the expectation and the covariance matrix do not fully specify the probability density of the state vector. The question is then how to determine this non-Gaussian density, and how to represent it in an economical way. Unfortunately, no general answer exists to this question.

assuming that the nonlinearities of the system are smooth enough to allow linear or quadratic approximations, Kalman-like filters become within reach. These solutions are suboptimal since there is no guarantee that the approximations are close. An obvious way to get the approximations is by application of a Taylor series expansion of the functions.

$$f(x+\epsilon) = f(x) + F(x)\epsilon + H.O.T. \tag{3.3}$$

$$h(x+\epsilon) = h(x) + hx)\epsilon + H.O.T. \tag{3.4}$$

where, H.O.T. represent the higher order terms.

The update steps then become:

$$\hat{z}(i) = H(x(i))\bar{x}(i \mid i-1) \tag{3.5}$$

$$S(i) = H(x(i))C(i \mid i-1)H^T(x(i)) + C_v(i) \tag{3.6}$$

$$K(i) = C(i \mid i-1)H^T(x(i))S^{-1}(i) \tag{3.7}$$

$$\bar{x}(i \mid i) = \bar{x}(i \mid i-1) + K(i)(z(i) - \hat{z}(i)) \tag{3.8}$$

$$C(i \mid i) = C(i \mid i-1) - K(i)S(i)K^T(i) \tag{3.9}$$

And the prediction step can be given as follows:

$$\bar{x}(i+1 \mid i) = f(x(i), u(i), i) \tag{3.10}$$

$$C(i+1 \mid i) = F(i)C(i \mid i)F^T(i) + C_w(i) \tag{3.11}$$

Chapter 4

Constrained Extended Kalman Filter

Many times, it is necessary to constrain the state estimation in the form of algebraic equality and/or inequality constraints. This can be necessary in case the estimated states are used in some feedback control law and unbounded states can lead to undesirable control.

For weakly nonlinear systems, the Extended Kalman Flter (EKF) has found numerous uses as a suboptimal state estimator. Unfortunately the structure of the filter does not include constraints on the states. This chapter presents analytical solutions to the state Constrained EKF (CEKF) for a class of linear constraints.

4.0.1 The Projection Approach

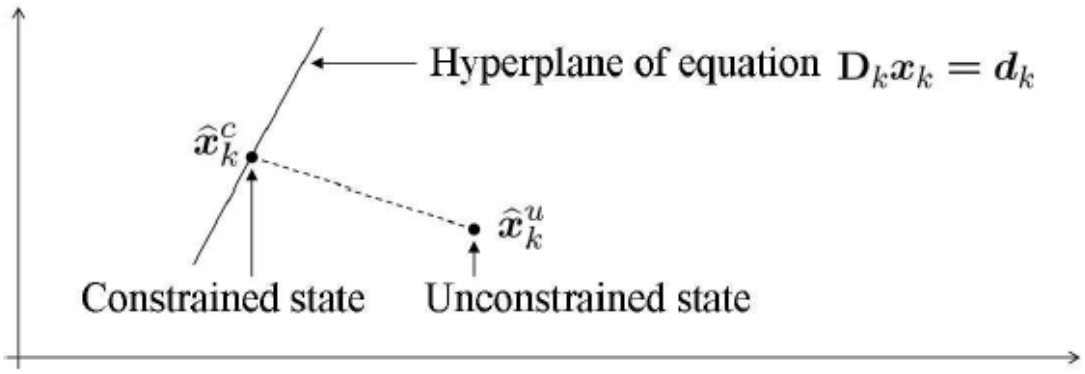


Figure 4.1: The Principle of Projection Approach [8]

Let us consider we have the following linearized model of the plant:

$$x(i+1) = F(i)x(i) + L(i)u(i) + w(i) \tag{4.1}$$

$$y(i) = H(i)x(i) + v(i) \tag{4.2}$$

Suppose at each time instant i the states are subject to following constraints:

$$D_i x(i) = d_i \tag{4.3}$$

Let us denote by $\hat{x}_i^u$ the state estimated at time i by an unconstrained estimator. $\hat{x}_i^c$ be the constrained estimate taking into account equation 4.3. The associated covariances of estimation error are denoted P_i^u and P_i^c. The principle of the projection approach is illustrated in Fig. 4.1. It consists in solving the following constrained optimization problem:

$$\min_{\hat{x}_i^c}((\hat{x}_i^c - \hat{x}_i^u)^T W_i^{-1}(\hat{x}_i^c - \hat{x}_i^u)) \qquad s.t. \quad D_i\hat{x}_i^c = d_i \tag{4.4}$$

W_i is a symmetric positive definite weighting matrix. The solution is obtained through the use of the Lagrange multiplier, and summarized by the following set of equations:

$$\hat{x}_i^c = \hat{x}_i^u + L_i(d_i - D_i\hat{x}_i^u) \tag{4.5}$$

$$P_i^c = (I_n - L_iD_i)P_i^u(I_n - L_iD_i)^T \tag{4.6}$$

$$L_i = W_i^{-1}D_i^T(D_iW_i^{-1}D_i^T)^{-1} \tag{4.7}$$

The constrained estimated state has the following properties:

- if $W_i = P_i^u$ then it results in the minimum variance filter
- if $W_i = I_n$ then it results in a constrained estimate that is closer to the true state than the unconstrained estimate

Now, suppose that at each time step i, state is subject to the following linear soft inequality constraint:

$$D_ix(i) \leq d_i \tag{4.8}$$

A way of dealing with such a problem is using the active set method []. It consists in testing at each time step i the scalar inequalities of 4.8. For the k^{th} inequality, two scenarii can occur:

- The inequality is satisfied, and so do not have to be taken into account.
- The inequality is not satisfied. Then, an equality constraint is applied to the boundary: $D_ix(i) = d_i$, where for any matrix M_i at time i, M_k, i denotes the k^{th} row of M_i

Consequently, dealing with soft inequality constraints reduces to the application at each time step of the active equality constraints.

Chapter 5

Cartesian Impedance Control

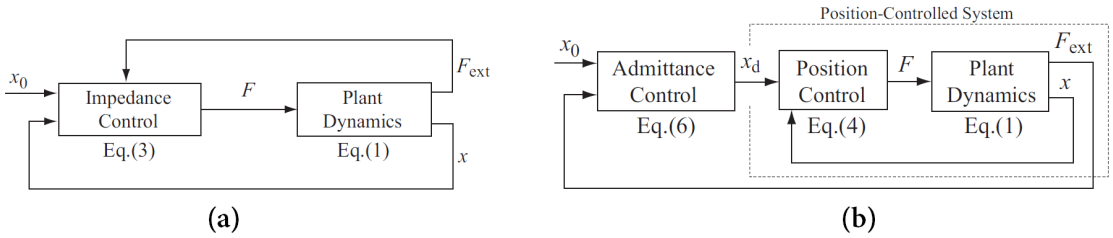


Figure 5.1: (a) Impedance Control Scheme and (b) Admittance Control Scheme [7]

Many applications in robotics require a robotic manipulator to interact with the environment. This interaction changes the overall dynamics of the system and usually this change is partially or completely unknown. Thus, it makes no sense to control either the force or the position of the end-effector, instead, a dynamic relationship can be set up between the motion of the manipulator and force applied by the environment. This is the core idea behind impedance control [3]. Thus, the control objective for impedance control is to design the control action F_c that will establish a given relationship between the external force F_{ext} and the error $e = (x - x_d)$ from a desired trajectory x_d.
Depending on the causality of the controller, there are two ways to implement impedance control: "Impedance Control" and "Admittance Control". Figure 5.1 shows the implementation of the two different schemes. The details on stability and performance of these schemes are well documented in [7]. In this work we have implemented the Admittance Control scheme, although as is usually done in literature, we will refer this to as impedance control in the remainder paper. A background of the Impedance control for a robotic manipulator has been presented in [2], but for the sake of completeness and readability we mention it in this section.
The Euler-Lagrange model of a n-DOF fully actuated robotic manipulator has the following form:

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) + \tau_f = \tau_c + J^T(q)F_{ext} \tag{5.1}$$

where $q \in R^n$, $M(q)$ is the inertia matrix, $C(q,\dot{q})$ is the vector of Coriolis/centrifugal torques, $g(q)$ is the vector of gravitational torques, τ_f is the vector of friction torques, τ_c is the vector of control torques, $J(q)$ is the robot Jacobian, $J^T(q)F_{ext}$ is the joint torque resulting from external force on the end-effector.

In order to design the impedance control, it is useful to derive the end effector dynamics in the operational space, considering only the translational motion:

$$\Lambda(q)\ddot{x} + \mu(q,\dot{q})\dot{x} + F_g(q) + F_f(q) = F_c + F_{ext} \tag{5.2}$$

where $x \in R^3$ is the Cartesian position vector of the end effector, $\Lambda = (JM^{-1}J^T)^{-1}$ is the (3×3) end effector inertia matrix, hereafter denoted as apparent inertia, while $\mu\dot{x} = \Lambda(JM^{-1}C - \dot{J})\dot{q}, F_g = J^{\dagger T}g, F_f = J^{\dagger T}\tau_f$ and $F_c = J^{\dagger T}\tau_c$ are the forces, reflected at the end effector, corresponding to the noninertial joint torques in equation 5.1.
Equation 5.2 describes only the end effector dynamics and does not include the so-called null space dynamics. Matrix $J^\dagger$ is the dynamically consistent generalized inverse of matrix J, defined as [5]

$$J^\dagger = M^{-1}J^T[JM^{-1}J^T]^{-1} \tag{5.3}$$

The goal of the impedance controller is to have a non-linear feedback law which translates the manipulator into an equivalent mass-damper-spring system. Let $x_d(t) \in R^3$ be a desired configuration for the end-effector. The relationship between $\tilde{x}(t) = x(t) - x_d(t)$ and F_{ext} is given by:

$$\Lambda_d(t)\ddot{\tilde{x}} + D_d(t)\dot{\tilde{x}} + K_d(t)\tilde{x} = F_{ext} \tag{5.4}$$

where $K_d(t), D_d(t) and M_d(t)$ are, respectively, the desired stiffness, damping and inertia matrices.

The above dynamics can be imposed to the closed loop controlled system by choosing F_c in as follows:

$$F_c = \eta(q,\dot{q}) + \Lambda(q)\Lambda_d^{-1}(D_d\dot{\tilde{x}} + K_d\tilde{x}) + (\Lambda(q)\Lambda_d^{-1} - I)F_{ext} \tag{5.5}$$

with $\eta(q,\dot{q}) = \mu(q,\dot{q})\dot{x} + F_g(q)$. In order to achieve (), it is necessary to feedback the external force F_{ext}, which can be measured by using a force/torque sensor mounted at the end-effector, or alternatively, force estimation techniques can be adopted. If the apparent inertia of the end effector is left unchanged, i.e., $\Lambda_d = \Lambda(x)$, the control law does not depend on the external force F_{ext}.

Chapter 6

Estimation Results

Experiments to estimate the impedance parameters were performed on a Variable Impedance Controlled 7-DOF KUKA LWR manipulator.

The following 1-DOF impedance model was used:

$$\Lambda\ddot{\tilde{x}} + D_d(t)\dot{\tilde{x}} + K_d(t)\tilde{x} = F_{ext} \tag{6.1}$$

where, $x_{des}(t) = 0$. A sinusoidal force was simulated on the end-effector for every experiment.

For the above model, the mass was considered known and was kept constant. Thus, we have four states to estimate: velocity (x_1), position (x_2), damping (x_3) and stiffness (x_4). The measurement is the end-effector position x_2 and the input to the model is F_{ext}. Following is the dynamics of the model:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} \frac{F_{ext} - D_d x_1 - K_d x_2}{\Lambda} \\ x_1 \\ 0 \\ 0 \end{bmatrix} \tag{6.2}$$

In the discrete form, we have:

$$\begin{bmatrix} x_1^{t+1} \\ x_2^{t+1} \\ x_3^{t+1} \\ x_4^{t+1} \end{bmatrix} = \begin{bmatrix} \frac{dt(F_{ext} - D_d x_1 - K_d x_2)}{\Lambda} + x_1^t \\ dt(x_1) + x_2^t \\ x_3^t \\ x_4^t \end{bmatrix} \tag{6.3}$$

The dyanmics of the damping and stiffness are not know. Hence, they are considered constant.

6.0.1 Estimation with constant Impedance Parameters

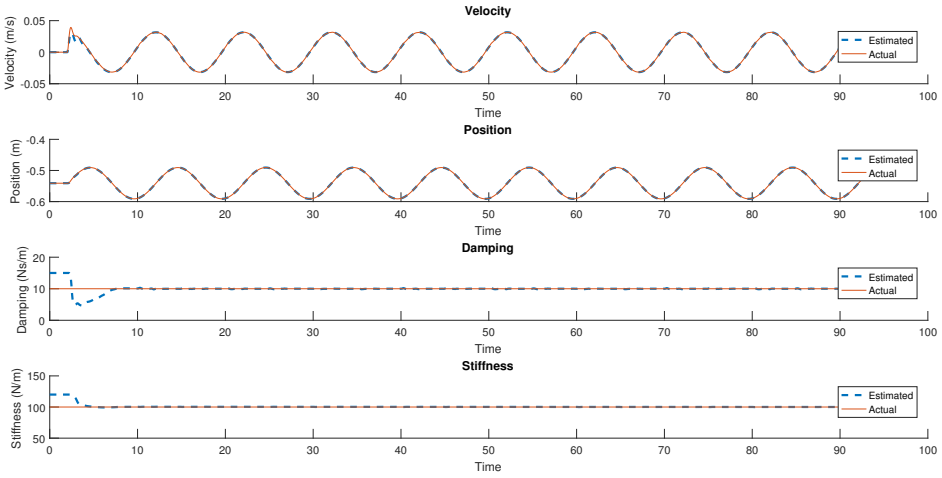


Figure 6.1: Estimation with constant impedance parameters: Mass-Damping-Stiffness Model

Figure 6.1 show the results of estimation of constant impedance parameters ($D_d = 10Ns/m$ and $K_d = 100N/m$). It can be seen that starting from initial guess different from that of the real parameters, the estimate converges to the real values.

6.0.2 Estimation with varying Impedance Parameters

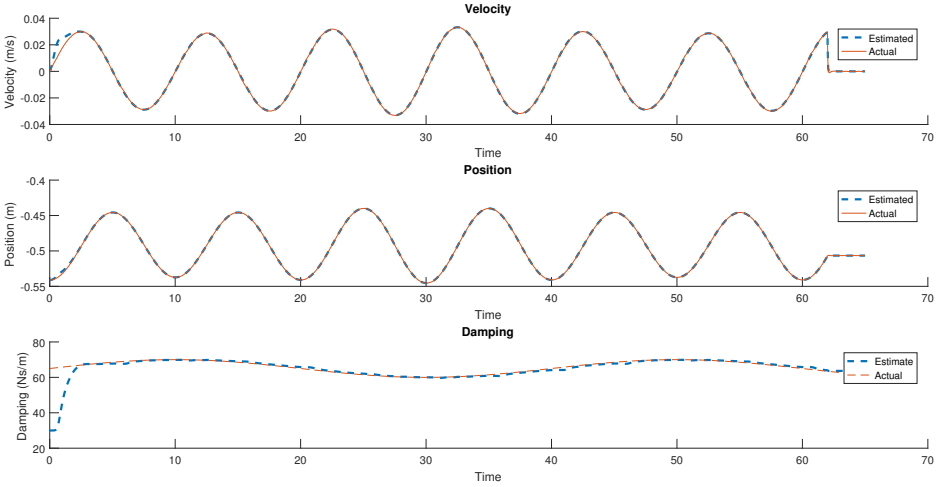


Figure 6.2: Estimation with varying impedance parameters: Mass-Damping Model

Figure 6.2 show the results of estimation of varying impedance parameter ($D_d(t) = 65 + 5\frac{sin(2\pi t)}{40}Ns/m$). It can be seen that starting from initial guess different from that of the real parameters, the estimate converges to the real values.

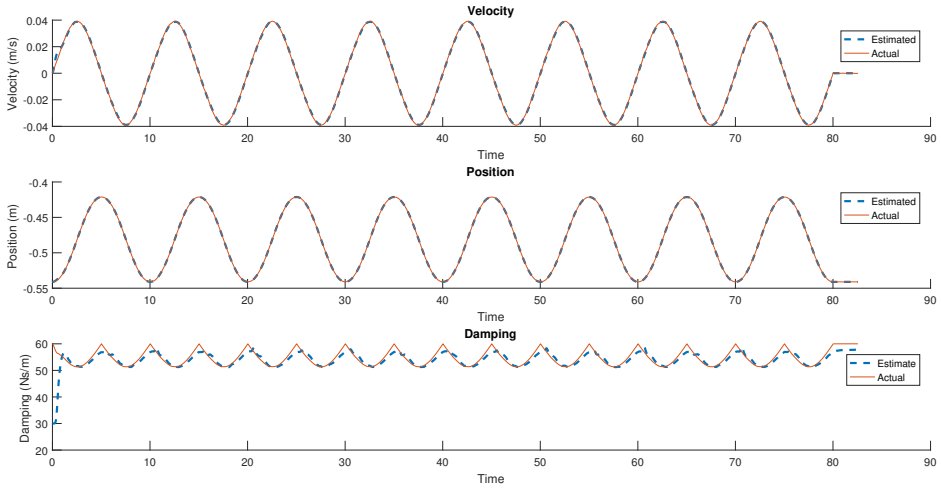


Figure 6.3: Estimation with varying impedance parameters: Mass-Damping Model

Figure 6.3 show the results of estimation of varying impedance parameter ($D_d(t) = 60e^{(-4|x_1|)}Ns/m$). It can be seen that starting from initial guess different from that of the real parameters, the estimate converges to the real values.

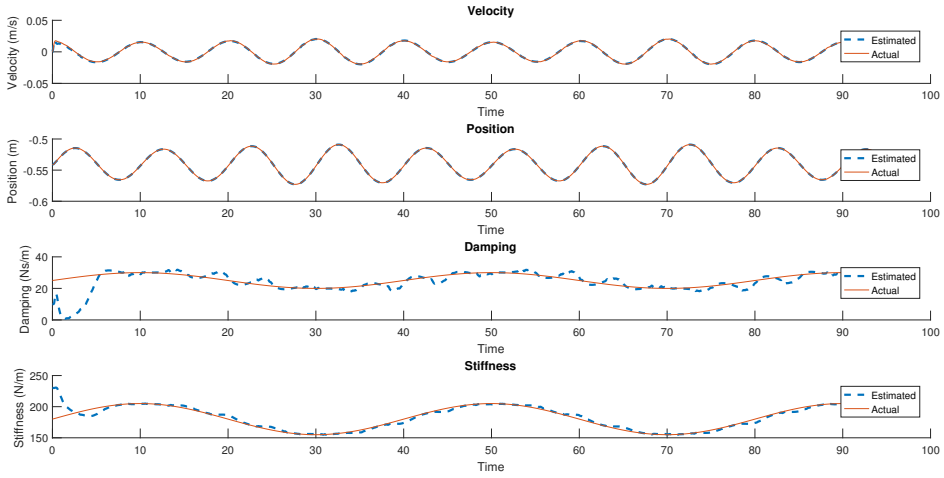


Figure 6.4: Estimation with varying impedance parameters: Mass-Damping-Stiffness Model

Figure 6.4 show the results of estimation of varying impedance parameter ($D_d(t) = 25 + 5\frac{sin(2\pi t)}{40}Ns/m$ and $K_d = 180 + 25\frac{sin(2\pi t)}{40}N/m$). It can be seen that starting from initial guess different from that of the real parameters, the estimate converges quite appreciably to the real values for stiffness but not damping.

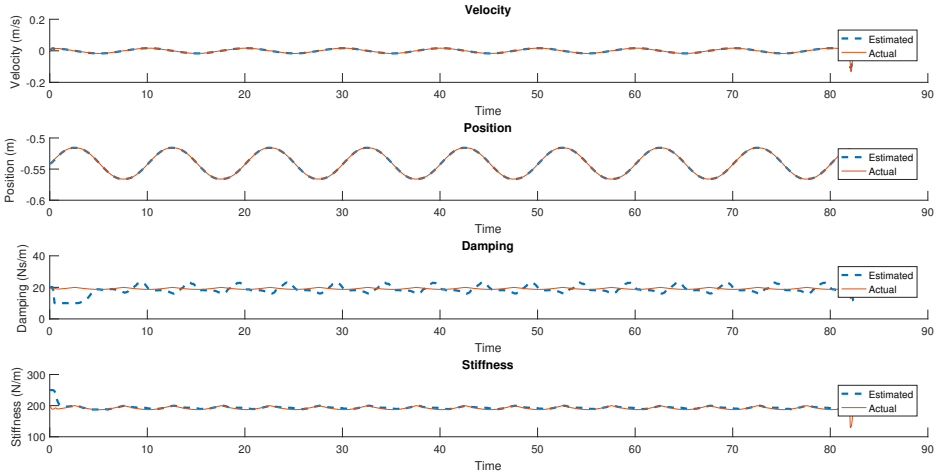


Figure 6.5: Estimation with varying impedance parameters: Mass-Damping-Stiffness Model

Figure 6.5 show the results of estimation of varying impedance parameter ($D_d(t) = 20e^{(-4|x_1|)}Ns/m$ and $K_d = 200e^{(-4|x_1|)}N/m$). It can be seen that starting from initial guess different from that of the real parameters, the estimate converges quite appreciably to the real values for stiffness but the results for damping are not good..

Chapter 7

Conclusion

In this work, the problem of estimating the impedance parameters of a variable impedance controlled robotic manipulator is addressed. The Extended Kalman Filter was used for this purpose because of its simplicity and speed of execution as compared to other non-linear estimation techniques.

The performance of the filter was tested for constant as well as variable impedance law. As can be expected, the estimation for the constant impedance law gives best result as this goes along with the state transition matrix used in the filter. In case of variable impedance law, a mass-damping and mass-damping-stiffness model was tested with inappreciable (sinusoidal) and appreciable (exponential) non-linear variation law.

In case of sinusoidal variation, the tracking is quite good but worsens in case of exponential variation of impedance law. This is because the EKF system model considers damping and stiffness as constants and goes ahead with the estimation based only on the error in the measurement data. Thus the poor performance is understandable.

If we compare the results of mass-damping and mass-damping-stiffness model, it can be observed that the parameter which most affects the dynamics is damping and stiffness respectively. Therefore, in case of mass-damping-stiffness model, the estimation of damping is poor as compared to that of stiffness. This can be remedied by using a sufficiently exciting input to the EKF which helps the damping term to become significant.

These work is assignment is done so that it can be applied to a manually guided surgery task wherein the input force is slowly varying. The mass was considered constant and known in the performed experiments as getting a good estimation of mass is difficult in case the force is slowly varying and thus not sufficiently exciting.

To obtain a better performance in the estimation, Adaptive EKF can be used, which can adapt the noise and state covariance matrices which can take care of the unknown model in this case.

Appendix A

C++ code for EKF Estimation

```
1
2
3  //========================================== Ajinkya EKF
       Initializations ==================
4
5                  Vector<2> x_d_EKF = makeVector(x_pos_ini[0],
                       x_pos_ini[1]);
6                  const int dim_state = 10, dim_obs = 2;
7      const int dim_con = 14; //4 for damping 4 for stiffness
           4 for mass 2 for k=d^2
8
9      //State Vectors
10     Vector<dim_state> x_p_EKF = makeVector(0.0,0.0,x_d_EKF
           [0],x_d_EKF[1],1.5,1.5,20,20,200,200);
11     Vector<dim_state> x_u_EKF = Zeros;
12     Vector<dim_state> X_EKF = x_p_EKF; //Updated
13     //State Covariance
14     Matrix<dim_state,dim_state> C_p_EKF=makeVector
           (0,0,0,0,0,0,0,0,0,0).as_diagonal();
15     //Initial velocity and position is known with good
           confidence
16     Matrix<dim_state,dim_state> C_u_EKF=Zeros;
17     Matrix<dim_state,dim_state> C_EKF=Zeros; //Updated
18
19
20     //State Transition Matrix
21     Matrix<dim_state,dim_state> A_EKF=Zeros;
22     //Observation Matrix
23     Matrix<dim_obs,dim_state> H_EKF=Zeros;
24     H_EKF.slice(0,dim_obs,dim_obs,dim_obs) = Identity(
           dim_obs);
25     Vector<dim_obs> z_p_EKF = Zeros;
26     Vector<dim_obs> z_meas_EKF;
27
28     //Innovation Matrix
```

```
29      Matrix<dim_obs,dim_obs> S_EKF = Zeros;
30      Matrix<dim_obs,dim_obs> S_EKF_inv = Zeros;
31      //Kalman Gain
32      Matrix<dim_state,dim_obs> K_EKF = Zeros;
33
34      Vector<dim_obs> F_EKF; //Force Input
35      double dt_EKF=CycTime;
36
37
38      double v_rndd = pow(0.00001,2);
39      double p_rndd = pow(0.000001,2);
40      double m_rndd = pow(1,2);
41      double d_rndd = pow(3,2);
42      double k_rndd = pow(10,2);
43      double r_rndd = pow(0.5,2);
44
45      //Noise Matrices
46      Matrix<dim_state,dim_state> Q_EKF=makeVector(v_rndd,
           v_rndd,p_rndd,p_rndd,m_rndd,m_rndd,d_rndd,d_rndd,
           k_rndd,k_rndd).as_diagonal();
47      Matrix<dim_obs,dim_obs> R_EKF=makeVector(r_rndd,r_rndd).
           as_diagonal();
48
49      //Variable impedance Parameters
50      int va = 20, vb = 4, vam = 5, vak = 200; // Constraint(
           vam,va) Parameters(va,vb)
51
52      //Active Method
53      Matrix<dim_con,dim_state> Constr= Zeros; //Linear
           Constraint Matrix Filler
54      Vector<dim_con> ConV = Zeros; //Constraint Vector
55      Vector<dim_con> g_of_x = Zeros; //Nonlinear Constraint
56      Matrix<dim_state,dim_con> L_ACT_out = Zeros; //
           Covariance Update Gain Matrix
57      Matrix<dim_con,dim_state> Act_out = Zeros; //Linear
           Constraint Matrix
58
59  //Adaptive
60                  Vector<dim_obs> res_AEKF; //Residual
61                  Vector<dim_obs> innov_AEKF; //Residual
62                  double alpha_AEKF = 1;
63
64  //=============================================================
       Ajinkya EKF Update ============================================//
65
66  z_meas_EKF = makeVector(xi[0],xi[1]);      //Note i
67
68  //Measurement Prediction
```

```
69 z_p_EKF = H_EKF*x_p_EKF;
70
71 S_EKF = H_EKF*C_p_EKF*H_EKF.T() + R_EKF;
72 Cholesky<dim_obs> chol_S_AEKF(S_EKF);
73 S_EKF_inv = chol_S_AEKF.get_inverse();
74 K_EKF=C_p_EKF*H_EKF.T()*S_EKF_inv;          // Kalman filter
       gain
75
76
77
78
79 C_EKF=C_p_EKF-K_EKF*S_EKF*K_EKF.T();       // Final state
       covariance matrix
80 //C_EKF = 0.5*(C_EKF+C_EKF.T());
81
82 //Adaptation
83 R_EKF = alpha_AEKF*R_EKF + (1.0 - alpha_AEKF)*(innov_AEKF.
       as_col()*innov_AEKF.as_row() + H_EKF*C_p_EKF*H_EKF.T() );
84 Q_EKF = alpha_AEKF*Q_EKF + (1.0 - alpha_AEKF)*(K_EKF*
       innov_AEKF.as_col()*innov_AEKF.as_row()*K_EKF.T());
85
86 //Start Active Set
87 int active_con =1; // Check number of active constraints
88
89 for (int ac=0; ac<1; ac++) // Run three times to get good
       estimate
90 {
91
92 Constr = Zeros;
93 ConV = Zeros;
94 g_of_x = Zeros;
95 active_con =0;
96 if(X_EKF[4]>mass+3) {Constr(active_con,4) = 1; ConV[
       active_con] = mass+3; g_of_x[active_con] = X_EKF[4];
       active_con = active_con+1;}
97 if(X_EKF[4]<mass-3)  {Constr(active_con,4) = -1; ConV[
       active_con] = -(mass-3); g_of_x[active_con] = -X_EKF[4];
       active_con = active_con+1;}
98 if(X_EKF[5]>mass+3) {Constr(active_con,5) = 1; ConV[
       active_con] = mass+3; g_of_x[active_con] = X_EKF[5];
       active_con = active_con+1;}
99 if(X_EKF[5]<mass-3)  {Constr(active_con,5) = -1; ConV[
       active_con] = -(mass-3); g_of_x[active_con] = -X_EKF[5];
       active_con = active_con+1;}
100 if(X_EKF[6]>va+10) {Constr(active_con,6) = 1; ConV[
       active_con] = va+10; g_of_x[active_con] = X_EKF[6];
       active_con = active_con+1;}
101 if(X_EKF[6]<vam)  {Constr(active_con,6) = -1; ConV[
```

```
    active_con] = -vam; g_of_x[active_con] = -X_EKF[6];
    active_con = active_con+1;}
102 if(X_EKF[7]>va+10) {Constr(active_con,7) = 1; ConV[
    active_con] = va+10; g_of_x[active_con] = X_EKF[7];
    active_con = active_con+1;}
103 if(X_EKF[7]<vam)  {Constr(active_con,7) = -1; ConV[
    active_con] = -vam; g_of_x[active_con] = -X_EKF[7];
    active_con = active_con+1;}
104 if(X_EKF[8]>vak+20) {Constr(active_con,8) = 1; ConV[
    active_con] = vak+20; g_of_x[active_con] = X_EKF[8];
    active_con = active_con+1;}
105 if(X_EKF[8]<vam)  {Constr(active_con,8) = -1; ConV[
    active_con] = -vam; g_of_x[active_con] = -X_EKF[8];
    active_con = active_con+1;}
106 if(X_EKF[9]>vak+20) {Constr(active_con,9) = 1; ConV[
    active_con] = vak+20; g_of_x[active_con] = X_EKF[9];
    active_con = active_con+1;}
107 if(X_EKF[9]<vam)  {Constr(active_con,9) = -1; ConV[
    active_con] = -vam; g_of_x[active_con] = -X_EKF[9];
    active_con = active_con+1;}
108
109
110 if(active_con>0)
111 {
112 // Act * X = Con
113 Matrix<Dynamic,dim_state> Act(active_con,dim_state);
114 Vector<> Con(active_con);
115 Vector<> gofx(active_con);
116 Matrix<> S_ACT(active_con,active_con);
117 Matrix<> S_ACT_inv(active_con,active_con);
118 Matrix<dim_state,Dynamic> L_ACT(dim_state,active_con);
119
120 Act = Constr.slice(0,0,active_con,dim_state);
121 Act_out.slice(0,0,active_con,dim_state) = Act;
122 Con = ConV.slice(0,active_con);
123 gofx = g_of_x.slice(0,active_con);
124
125 Con = Con - gofx + Act*X_EKF;
126
127 Matrix<> Dum(active_con,active_con);
128 Dum = 1e-10*Identity; // To avoid loss of rank
129 //Active Set Update
130 S_ACT = Act*Act.T() + Dum;
131 Cholesky<Dynamic> chol_S_ACT(S_ACT);
132 S_ACT_inv = chol_S_ACT.get_inverse();
133
134 L_ACT = Act.T()*S_ACT_inv;
135 L_ACT_out.slice(0,0,dim_state,active_con) = L_ACT;
```

```
136
137 X_EKF = X_EKF + L_ACT*(Con - Act*X_EKF);
138
139 }
140
141 } //Three times run end
142
143 if(active_con>0)
144 {
145 Matrix<dim_state,dim_state> eye_dim_state = Identity;
146 C_EKF = (eye_dim_state - L_ACT_out.slice(0,0,dim_state,
      active_con)*Act_out.slice(0,0,active_con,dim_state))*
      C_EKF*(eye_dim_state - L_ACT_out.slice(0,0,dim_state,
      active_con)*Act_out.slice(0,0,active_con,dim_state)).T();
147 //C_EKF = 0.5*(C_EKF+C_EKF.T());
148 }
149 //Active Set Ends
150
151 x_u_EKF = X_EKF;
152 estim[itr] = X_EKF;
153 C_u_EKF = C_EKF;
154
155
156
157 //=============================================================
      Ajinkya Endof EKF Update
      ============================================//
158
159
160 //=============================================================
      Ajinkya EKF Prediction
      ============================================//
161
162
163 //To take care of absolute values
164 int signv0 = 1, signv1 =1;
165 if (x_u_EKF[0]>0)
166 {signv0=1;}
167 else if (x_u_EKF[0]==0)
168 {signv0=0;}
169 else
170 {signv0=-1;}
171 if (x_u_EKF[1]>0)
172 {signv1=1;}
173 else if (x_u_EKF[1]==0)
174 {signv1=0;}
175 else
176 {signv1=-1;}
```

```
177
178 //Control Input
179 F_EKF=makeVector(f[0],f[1]); // Fx
180
181 //Predict the state ------ sp: state predict---------
182 double sp00 = (F_EKF[0]-x_u_EKF[6]*x_u_EKF[0]-x_u_EKF[8]*(
       x_u_EKF[2]-x_d_EKF[0])); //-------------------------------
        TTC -------------------------//
183 double sp0 = (dt_EKF/x_u_EKF[4])*sp00;
184 double sp11 = (F_EKF[1]-x_u_EKF[7]*x_u_EKF[1]-x_u_EKF[9]*(
       x_u_EKF[3]-x_d_EKF[1])); //-------------------------------
        TTC -------------------------//
185 double sp1 = (dt_EKF/x_u_EKF[5])*sp11;
186
187 x_p_EKF[0]=x_u_EKF[0]+sp0;
188 x_p_EKF[1]=x_u_EKF[1]+sp1;
189 x_p_EKF[2]=x_u_EKF[2]+dt_EKF*x_u_EKF[0];
190 x_p_EKF[3]=x_u_EKF[3]+dt_EKF*x_u_EKF[1];
191 x_p_EKF[4]=x_u_EKF[4];
192 x_p_EKF[5]=x_u_EKF[5];
193 x_p_EKF[6]=x_u_EKF[6];
194 x_p_EKF[7]=x_u_EKF[7];
195 x_p_EKF[8]=x_u_EKF[8];
196 x_p_EKF[9]=x_u_EKF[9];
197
198 //State Transition Matrix

    //------------------------------------ TTC in first two rows for
     k(x-xd)-------------------------//
199 Fill(A_EKF)= (1-(dt_EKF*x_u_EKF[6])/x_u_EKF[4]),0, -(dt_EKF*
    x_u_EKF[8])/x_u_EKF[4],0, (-dt_EKF*sp00)/(pow(x_u_EKF
    [4],2)),0, -(dt_EKF*x_u_EKF[0])/x_u_EKF[4],0, -(dt_EKF*(
    x_u_EKF[2]-x_d_EKF[0]))/x_u_EKF[4],0,
200             0,1-(dt_EKF*x_u_EKF[7])/x_u_EKF[5],0, -(dt_EKF*
                x_u_EKF[9])/x_u_EKF[5],0, (-dt_EKF*sp11)/(pow
                (x_u_EKF[5],2)),0, -(dt_EKF*x_u_EKF[1])/
                x_u_EKF[5],0, -(dt_EKF*(x_u_EKF[3]-x_d_EKF
                [1]))/x_u_EKF[5],
201              dt_EKF,0,1,0,0,0,0,0,0,0,
202              0,dt_EKF,0,1,0,0,0,0,0,0,
203                  0,0,                    0,0,

                     1,0,                    0,0,
                     0,0,
204                  0,0,                    0,0,

                     0,1,                    0,0,
                     0,0,
```

```
205                     0,0,                    0,0,
                            0,0,                    1,0,
                            0,0,
206                     0,0,                    0,0,
                            0,0,                    0,1,
                            0,0,
207                     0,0,                    0,0,
                            0,0,                    0,0,
                            1,0,
208                     0,0,                    0,0,
                            0,0,                    0,0,
                            0,1;
209
210
211 //Predict Covariance
212 C_p_EKF=A_EKF*C_u_EKF*A_EKF.T()+Q_EKF;
213 C_p_EKF = 0.5*(C_p_EKF+C_p_EKF.T());
214
215
216 //
    ================================================================================
      Ajinkya EKF
    ================================================================================//
```

Bibliography

[1] Brian DO Anderson and John B Moore. "Optimal filtering". In: *Englewood Cliffs* 21 (1979), pp. 22–95.

[2] Fanny Ficuciello, Luigi Villani, and Bruno Siciliano. "Variable impedance control of redundant manipulators for intuitive human–robot physical interaction". In: *IEEE Transactions on Robotics* 31.4 (2015), pp. 850–863.

[3] Neville Hogan. "Impedance control: An approach to manipulation". In: *American Control Conference, 1984*. IEEE. 1984, pp. 304–313.

[4] Rudolph Emil Kalman et al. "A new approach to linear filtering and prediction problems". In: *Journal of basic Engineering* 82.1 (1960), pp. 35–45.

[5] Oussama Khatib. "A unified approach for motion and force control of robot manipulators: The operational space formulation". In: *IEEE Journal on Robotics and Automation* 3.1 (1987), pp. 43–53.

[6] Bangjun Lei et al. *Classification, parameter estimation and state estimation: an engineering approach using MATLAB*. John Wiley & Sons, 2017.

[7] Christian Ott, Ranjan Mukherjee, and Yoshihiko Nakamura. "Unified impedance and admittance control". In: *Robotics and Automation (ICRA), 2010 IEEE International Conference on*. IEEE. 2010, pp. 554–561.

[8] Vincent Sircoulomb et al. "State estimation under nonlinear state inequality constraints. A tracking application". In: *Control and Automation, 2008 16th Mediterranean Conference on*. IEEE. 2008, pp. 1669–1674.