ABSTRACT

Title of Document:	FORCE ESTIMATION BASED COMPLIANCE CONTROL OF A TWO LINK HARMONICALLY DRIVEN ROBOTIC MANIPULATOR. Leon Michael Aksman Master of Science, 2006
Directed By:	Professor David L. Akin Department of Aerospace Engineering

The estimation of external forces exerted on a robotic manipulator with harmonic drive gearing without a force/torque sensor is considered. Manipulator dynamics, together with motor current feedback are used to estimate external joint torques, which are transformed into estimated external end effector forces using knowledge of the manipulator's kinematics. Adaptive control is used to tune the parameters of the robot's modeled dynamics, while adaptive radial basis function (RBF) neural networks are used to learn the friction dynamics. Admittance control without force sensing is attempted on a two degree of freedom manipulator. Readings from a six-axis force/torque sensor mounted on the manipulator are used to validate the force estimates during the estimation phase.

FORCE ESTIMATION BASED COMPLIANCE CONTROL OF A TWO LINK HARMONICALLY DRIVEN ROBOTIC MANIPULATOR

By

Leon Michael Aksman

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Master of Science
2006

Dedication

To my family and Grace.

Acknowledgements

First and foremost I'd like to thank Dave Akin for taking me as one of his grad students and letting me joint the SSL. The lab could not have been a better fit for me both academically and personally – I got to work with a lot of great people. Some of those are the members of the SSL's world-class research staff who consistently sacrifice their time and efforts to help both the grads and undergrads in the lab. In particular, JM Henriette and Stephen Roderick have spent more time than should be legal helping me with all manner of woes related to my research. JM and Stephen's wealth of practical knowledge, part of what kept the lab running for the past decade or so, is the reason I ever got any hardware or software to work at all.

I'd like to thank the rest of my committee, Rob Sanner and Craig Carignan. Discussions with Dr. Sanner often clarified the finer points of adaptive control and always served to improve the focus of my research. I would not have completed my thesis without the help of Dr. Craig Carignan, who has helped every step of the way. From choosing the topic to helping me understand its scope to proofreading every draft I sent his way, Craig has been a tireless advisor.

I'd also like to thank my fellow SSL grad students, particularly officemates past and present: Max Ransan, Tim Wasserman, Mike Liszka, and Enrico Sabelli. Without them I would have probably finished my thesis but I definitely not have enjoyed the process nearly as much.

Finally it goes without saying that my family supported and helped me in every way they could have. Along with Grace I am very lucky to have people like that in my life.

Table of Contents

Chapter 1: Introduction

1.1 Motivation

The most common robotic manipulator control schemes are those that attempt to control strictly position. Such schemes ultimately require that a manipulator track a time varying joint trajectory specified for each of its degrees of freedom. Position control is an intuitive and often effective means by which to accomplish tasks. Its major drawback is that a manipulator will attempt to track its desired trajectory even if that brings damage to itself and objects in its way.

As a result, force control schemes have been developed to deal with controlling interactions between the manipulator and its environment. Compliance control attempts to combine position and force control by enforcing a mass-spring-damper relationship between external force and the manipulator's desired position, velocity and acceleration.

Robotic manipulators typically use force/torque sensors to realize force or compliance control. However force/torque sensors have several well-known drawbacks in the form of their cost, size and the complexity they introduce into a manipulator's mechanical, electrical, and software design. Force/torque sensors provide their most accurate and stable results when placed as close as possible to the end effector, constraining a system's mechanical design. The gravitational term of the end effector itself must then be compensated for in software. Also there is a need to incorporate the sensors' output signals into the system poses problems electrically as well.

Another, less common drawback is that force/torque sensors saturate due to high water pressure, rendering them ineffective in deep-sea robotic sampling tasks. The Space Systems Laboratory (SSL) of the University of Maryland is partnering with the Woods Hole Oceanographic Institute (WHOI) to create a system that will autonomously collect samples from the floor of the Artic Ocean. The project, part of NASA's Astrobiology Science and Technology Experiment Program (ASTEP), aims to be the first expedition to sample the hydrothermal vents in the Gakkel Ridge region of the Artic. The system will consist of the SSL's Subsea Artic Manipulator for Underwater Retrieval and Autonomous Interventions (SAMURAI) arm mounted on JAGUAR, a WHOI autonomous underwater vehicle (AUV). Figure 1.1 (a) depicts the manipulator mounted on the AUV while Figure 1.1 (b) depicts an actual prototype of the AUV. For such a system using force estimation instead for compliance control can enable safer interaction between the manipulator and its external environment.

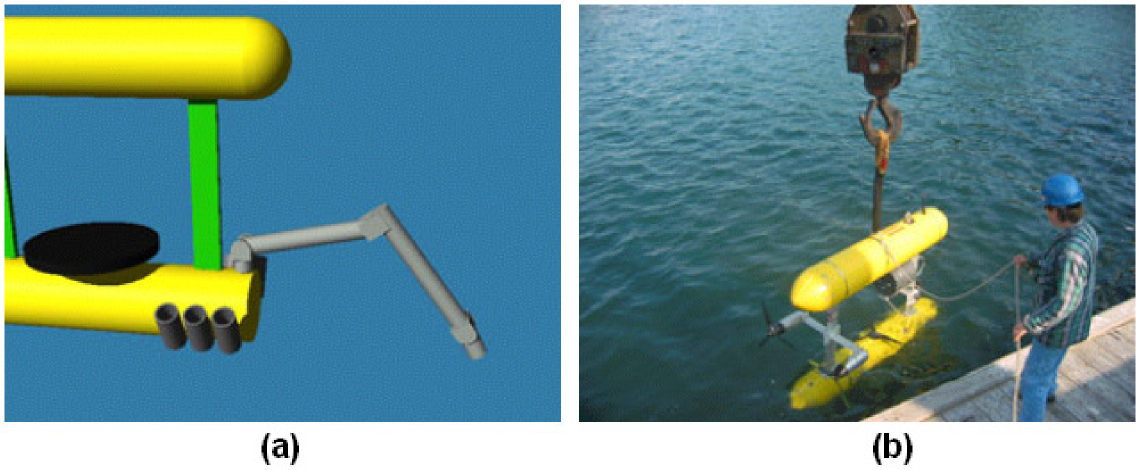


Fig 1.1 (a): JAGUAR with SAMURAI arm and sample containers (Model by Stephen Roderick, SSL).
(b): SeaBED, the prototype of JAGUAR (Photograph by Mike Naylor, SSL).

Many space and underwater manipulators, including SAMURAI, use electric motors, which provide high speed but low torque. They are therefore geared with harmonic drives. Gearing reduces a motor's speed but increases its torque. Harmonic drives are a popular method of providing gearing as they enable high gear ratios and cause little backlash. However, they tend to greatly increase manipulators' joint friction, especially in static situations. Stiction, short for static friction, is a major source of error in force estimation due to the difficulty in modeling its behavior. This work will attempt to characterize such difficulties while attempting compliance control based on force estimation in harmonically driven manipulators.

1.2 Previous Work

Force estimation as applied to robotic manipulators has been a topic of interest since the early 1990's. Murakami et. al., (1993) proposed a decoupled disturbance observer based approach. Hacksel and Salcudean (1994) presented a coupled force observer based on accurate knowledge of a robot's dynamics. Both observer based approaches demonstrated good results on direct drive manipulators with negligible unmodeled friction dynamics.

More recently, dynamics learning has been used in force estimation. Simpson and Hashtrudi-Zaad (2005) used a neural network to learn the entire dynamical model of their 3 degree of freedom (DOF) haptic device offline. Their system contained little friction however and the dynamics of the system were assumed to be unchanging after the initial neural network training. Zhan et al. (1998) showed that force sensorless hybrid force/position control was possible in a geared, though not harmonically driven, manipulator. They used a simplified model of robot dynamics, consisting of a known gravity term and a learned friction term. Adaptive neural networks, discussed shortly, were used for online friction learning though adaptation of the modeled dynamics was not performed.

Simpson et al. (2002) used motor current to estimate external forces for robots with harmonic drive gearing. The approach involved subtracting modeled dynamics from motor torque, assumed to be proportional to motor current, to form the estimated external torque. The estimated torque thus obtained contained significant unmodeled position-dependent friction. Filtering the estimated external torque in the position domain greatly improved the estimates. The technique is based on the friction modeling work of Popovic and Goldberg (1998) which involved using spectral analysis in the joint position domain, rather than the time domain, to model friction. The filtering was done offline however, when the entire position history of the estimated external torque was known. Therefore the force estimates are not suitable for use in real-time control.

All of these techniques have relied on well-known, unchanging parameters of the manipulator's dynamics. In reality, the parameters of the manipulator's dynamics are usually not known precisely. This is especially true under changing end effector load. For these reasons an adaptive control law for robotic manipulators was originally developed by Slotine and Li (1987). It relied on knowledge of the manipulator's full dynamical model with no unmodeled dynamics assumed. The parameters of the model were tuned online while maintaining closed loop control.

In addition to learning parameters of the modeled dynamics, it is often desirable to learn unmodeled dynamical terms. A control law involving the use of radial basis function (RBF) adaptive networks (a.k.a "neural networks") to learn a manipulator's unmodeled friction term, assumed to be velocity-dependent, was introduced by Sanner and Slotine (1992). In the work the function approximation abilities of such networks were investigated and bounds on tracking error were given based on the number of nodes in the network and their inverval. Sanner and Slotine (1995) combined the online learning of unmodeled dynamics with the online adaptation of modeled dynamics in a stable control law for manipulators. Liu (1997) performed further experiments using Sanner and Slotine's controller on an experimental one DOF manipulator. The ability of the controller's adaptive networks to approximate friction under variations in temperature was successfully demonstrated. In addition, the adaptive networks were able to learn unmodeled dynamics such as joint velocity dependent hydrodynamic forces in dynamic motion underwater. The work presented in this thesis builds on the ability of this adaptive learning controller to learn both modeled and unmodeled dynamics while maintaining closed loop control of the manipulator. A real-time force estimation technique is presented that relies on learning the dynamical model using the controller. The technique also allows for relearning of the dynamics at certain points in time chosen by either an operator or higher-level autonomy. This feature can enable good force estimation ability despite changes due to loading, temperature or even more exotic disturbances such as unmodeled hydrodynamics.

Real-time force estimation leads to the feasibility of performing compliance control without a force/torque sensor. Compliance control blends strict force control and strict position control by modifying the manipulator's desired trajectory based on external forces. Compliance control was first introduced by Salisbury (1980) in the form of the stiffness controller and Hogan (1985) in the form of the impedance controller. Under impedance control a manipulator is viewed as an object that accepts deflection in position, velocity and acceleration due to contact and responds by exerting force on the environment. The dual to this concept is admittance control, in which a manipulator is viewed as accepting force due to contact with the environment and responding with modification of its trajectory. The compliance controller used in this work is a modification of an admittance control used by Guion (2003) where it was termed “position-based impedance control”. Position-based impedance control was first introduced by Maples and Becker (1986).

1.3 Objectives

The primary objective of this thesis is to investigate the ability to estimate external forces exerted on a highly geared, harmonically driven manipulator with considerable friction. The goal will be realized using online adaptation and friction learning control applied to a two DOF manipulator both in simulation and hardware experiments. The approach relies on modeling manipulator dynamics and using motor current to estimate external joint torques, which are transformed into estimated external end effector forces using knowledge of the manipulator’s kinematics. The secondary objective of this work is to investigate the use of the force estimates to demonstrate compliance control on the experimental system without a force/torque sensor.

1.4 Thesis Outline

This thesis is divided into five chapters. Chapter 2 discusses the theoretical background of the control law used and derives the force estimation technique being introduced using control, dynamics, and kinematics. Chapter 3 details the particular dynamics and kinematics of the two DOF manipulator used in both the simulations and hardware experiments. The details of the hardware, electronics, and software used are given as well as a description of how the three aspects are integrated. Chapter 4 provides the results of the simulation and hardware experiments performed. Chapter 5 describes the compliance controller using force estimation. Chapter 6 draws conclusions based on the results and outlines future work.

Chapter 2: Control, Dynamics, and Force Estimation

Before discussing the force estimation technique in this chapter a review of robot dynamics as well as both position and compliance control will be presented. Dynamics and position control will be directly associated with force estimation, which will be used in compliance control in Chapter 5.

2.1 Robot Dynamics

The following well-known equation describes the dynamics of a rigid N link robotic manipulator:

$$\mathrm{H}(\mathrm{q})\ddot{\mathrm{q}} + \mathrm{C}(\mathrm{q},\dot{\mathrm{q}})\dot{\mathrm{q}} + \mathrm{g}(\mathrm{q}) + \mathrm{f}_{\mathrm{v}}(\dot{\mathrm{q}}) = \tau \tag{2-1}$$

Equation (2-1) describes the relationship between $\boldsymbol{\tau}$, the $\mathrm{N}\times 1$ vector of input torques at the manipulator's N joints and the vectors $\mathbf{q}$, $\dot{\mathbf{q}}$ and $\ddot{\mathbf{q}}$ - the $\mathrm{N}\times 1$ vectors describing the resulting position, velocity and acceleration of the manipulator's N joints. The generalized variable $\mathbf{q}$ has been used rather than $\boldsymbol{\theta}$ to allow for the possibility of prismatic joints. Here $\mathbf{H(q)}$ is a matrix of size $\mathrm{N}\times\mathrm{N}$ that describes the position dependant inertial term. The $\mathrm{N}\times\mathrm{N}$ matrix $\mathbf{C(q,\dot{q})}$ describes the torque due to Coriolis and centripetal effects while the $\mathrm{N}\times 1$ vector $\mathbf{g(q)}$ describes the torque due to gravity. Additionally, the $\mathbf{f}_{\mathrm{v}}(\dot{\mathbf{q}})$ term is an $\mathrm{N}\times 1$ vector describing the friction torque at the manipulator's joints, assumed to be strictly a function of velocity and decoupled between the joints.

When external torques are exerted on the manipulator equation (2-1) becomes

$$H(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) + f_v(\dot{q}) = \tau + \tau_{ext} \tag{2-2}$$

where $\boldsymbol{\tau_{ext}}$ is the $N\times 1$ vector of external torques experienced at the manipulator's joints. For geared manipulators the following hold for each joint

$$q = G^{-1}q_m \tag{2-3}$$

$$\tau = G\tau_m \tag{2-4}$$

where **G** is a diagonal $N\times N$ matrix of gear ratios for each joint, $\mathbf{q_m}$ is the joint velocity of the rotor and $\boldsymbol{\tau_m}$ is the motor torque. Using equation (2-2) and generalizing the development of Craig (2005) to N degrees of freedom, a torque balance at the rotor can be written as

$$I_m\ddot{q}_m + B_m\dot{q}_m + G^{-1}\left(H(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) + f_v(\dot{q})\right) = \tau_m + G^{-1}\tau_{ext} \tag{2-5}$$

using relations (2-3), (2-4) this can be rewritten as

$$GI_m\ddot{q} + GB_m\dot{q} + G^{-1}\left(H(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) + f_v(\dot{q})\right) = G^{-1}\left(\tau + \tau_{ext}\right) \tag{2-6}$$

Multiplying both sides by **G** and arranging terms leads to

$$\left(H(q) + G^2 I_m\right)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) + f_v(\dot{q}) = \tau + \tau_{ext} \tag{2-7}$$

where $\mathbf{G}^{-1}$ is the diagonal $N\times N$ matrix of inverted gear ratios, $\mathbf{G}^2$ is the square of **G** and $\mathbf{I_m}$, $\mathbf{B_m}$ are the diagonal $N\times N$ matrices containing rotor inertias and viscous friction coefficients respectively. The term $\mathbf{G^2 B_m \dot{q}}$ that results in the step from (2-6) to (2-7) has been incorporated into the velocity-dependent friction $\mathbf{f_v}$. The Equation (2-7) is the complete model of the dynamics of the system used in this thesis.

Equations (2-1), (2-2) and (2-7) describe the manipulator's dynamics in joint space, another way to frame manipulator dynamics is in Cartesian space, also called task space. In that case the dynamical equations will describe the relationship between the input torque and the position, velocity and acceleration of the manipulator's end effector in Cartesian space. Due to the fact that the desired trajectories used in this thesis were framed in joint space, the task space description will not be discussed further.

2.2 Position and Compliance Control

As discussed in Chapter 1, position control and compliance control are two different classes of control algorithm used with robotic manipulators. Position control attempts to track a time varying joint trajectory without controlling contact forces. Position control laws typically use position sensor feedback and occasionally velocity sensors (tachometers) to form a control law based on the error between desired and actual joint trajectory. Often model feedforward is used in such laws when some or all of the manipulator's dynamics is known. Compliance control schemes attempt to enforce a mass-spring-damper relationship between the deflection of the manipulator's trajectory and the force due to contact with its environment. The class of compliance control algorithms is broken up into two subclasses – impedance controllers and admittance controllers. Impedance controllers accept deflection in trajectory away from the commanded due to contact and respond with a contact force based on the desired system compliance. Admittance controllers work in the opposite way, accepting sensed force as input and outputting the modified desired trajectory based on the desired system compliance.

Proportional-Derivative control, usually termed PD control, is the most fundamental control scheme. Despite the vast amount of literature and research related to more advanced robotic control, PD control continues to be widely used in practical applications. This is due to the ease of implementation and good results that are attainable when the scheme is properly applied. Its simplicity and consequently fast rate allows for high bandwidth applications. PD control involves the knowledge of four terms related to a manipulator's position and velocity – the desired and actual joint position vectors, $\mathbf{q_d(t)}$ and $\mathbf{q(t)}$ respectively, and the desired and actual velocity, $\mathbf{\dot{q}_d(t)}$ and $\mathbf{\dot{q}(t)}$ respectively. Two error terms can then be formed as

$$e(t) = q(t) - q_d(t) \tag{2-8}$$

$$\dot{e}(t) = \dot{q}(t) - \dot{q}_d(t)\,. \tag{2-9}$$

To simplify the notation, the dependence of all terms on time will henceforth not be made explicit, though it should be kept in mind. The PD control law is

$$\tau = -K_p e - K_d \dot{e} \tag{2-10}$$

where $\mathbf{K_p}$ and $\mathbf{K_d}$ are positive definite matrices ($\mathbf{K_p > 0}$, $\mathbf{K_d > 0}$) and typically diagonal. This control law can be rewritten as

$$\tau = -K_d s \tag{2-11}$$

$$s = \dot{e} + \Lambda e \tag{2-12}$$

where $\mathbf{\Lambda} = \mathbf{K_d}^{-1}\mathbf{K_p}$. This formulation of the PD control law is useful because it introduces the term **s**, which will be used throughout the remaining theoretical development. The reader should note that **s** is not the Laplace variable.

2.3 Adaptive Control with Friction Learning

The next step towards force estimation involves modeling the manipulator's dynamics and learning the parameters of that model. This is a crucial step because force estimation can only follow from an accurate model of the manipulator's dynamics. Otherwise the estimator will not be able to distinguish between torque needed to move the manipulator through free space and torque due to external force. To accomplish this goal, adaptive control is used because of its ability to rapidly learn the parameters of a dynamical model in real time while maintaining closed loop control. Assuming zero friction and external torque for now, (2-1) can be rewritten as

$$H(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) = \tau \tag{2-13}$$

The following control law for the system described by (2-13) has been shown to yield asymptotically convergent tracking of a desired time-varying trajectory $\mathbf{q_d(t)}$

$$\tau = H(q)\ddot{q}_r + C(q,\dot{q})\dot{q}_r + g(q) - K_D s \tag{2-14}$$

$$\dot{q}_r = \dot{q}_d + \Lambda e \tag{2-15}$$

$$\ddot{q}_r = \ddot{q}_d + \Lambda \dot{e}. \tag{2-16}$$

where (2-15) and (2-16) define two new terms - the reference velocity and reference acceleration respectively. The controller attempts to “linearize” the closed loop dynamics using knowledge of the manipulator’s open loop behavior.

If the exact values of all the system’s physical parameters were known, the first three terms on the right hand side of (2-14) could be rearranged as follows

$$\mathrm{H}(\mathrm{q})\ddot{\mathrm{q}}_{\mathrm{r}} + \mathrm{C}(\mathrm{q},\dot{\mathrm{q}})\dot{\mathrm{q}}_{\mathrm{r}} + \mathrm{g}(\mathrm{q}) = \mathrm{Y}(\mathrm{q},\dot{\mathrm{q}},\dot{\mathrm{q}}_{\mathrm{r}},\ddot{\mathrm{q}}_{\mathrm{r}})\mathrm{a} \qquad \text{(2-17)}$$

where $\mathbf{Y}(\mathbf{q},\dot{\mathbf{q}},\dot{\mathbf{q}}_{\mathbf{r}},\ddot{\mathbf{q}}_{\mathbf{r}})$ is an $\mathrm{N}\times\mathrm{M}$ matrix containing known functions that are parameterized by M constants, arranged in the $\mathrm{M}\times 1$ vector **a**.

When the values in **a** are not known exactly or unknown, the adaptive control law of Slotine and Li (1987) can be applied to the system described by (2-13)

$$\tau = \hat{\mathrm{H}}(\mathrm{q})\ddot{\mathrm{q}}_{\mathrm{r}} + \hat{\mathrm{C}}(\mathrm{q},\dot{\mathrm{q}})\dot{\mathrm{q}}_{\mathrm{r}} + \hat{\mathrm{g}}(\mathrm{q}) - \mathrm{K}_{\mathrm{d}}\mathrm{s} = \mathrm{Y}(\mathrm{q},\dot{\mathrm{q}},\dot{\mathrm{q}}_{\mathrm{r}},\ddot{\mathrm{q}}_{\mathrm{r}})\hat{\mathrm{a}} - \mathrm{K}_{\mathrm{d}}\mathrm{s} \qquad \text{(2-18)}$$

$$\dot{\hat{\mathrm{a}}} = -\Gamma \mathrm{Y}^{\mathrm{T}}\mathrm{s} \qquad \text{(2-19)}$$

where $\hat{\mathbf{a}}$ is the approximation of **a**, just as $\hat{\mathbf{H}}(\mathbf{q})$, $\hat{\mathbf{C}}(\mathbf{q},\dot{\mathbf{q}})$ and $\hat{\mathbf{g}}(\mathbf{q})$ are approximations of the terms in (2-13). Equation (2-19) gives the adaptation law for $\hat{\mathbf{a}}$ where $\boldsymbol{\Gamma}$ is a positive definite $\mathrm{M}\times\mathrm{M}$ matrix of learning gains, typically diagonal. The reason this particular adaptation law is used is that it eliminates a term in the derivative of the Lyapunov function chosen in the proof of the convergence of the error measure **s** of the system (2-13) under control law (2-18)/(2-19) provided in (Slotine and Li, 1988). To gain some intuition into this adaptation law it can be rewritten as

$$\dot{\hat{\mathrm{a}}} = -\Gamma \frac{\partial \tau}{\partial \mathrm{a}}^{\mathrm{T}} \mathrm{s} \qquad \text{(2-20)}$$

$$\mathrm{Y} = \frac{\partial \tau}{\partial \mathrm{a}} = \begin{bmatrix} \frac{\partial \tau_1}{\partial \mathrm{a}_1} & \cdots & \frac{\partial \tau_1}{\partial \mathrm{a}_M} \\ \vdots & \ddots & \vdots \\ \frac{\partial \tau_N}{\partial \mathrm{a}_1} & \cdots & \frac{\partial \tau_N}{\partial \mathrm{a}_M} \end{bmatrix}. \qquad (2\text{-}21)$$

From (2-21) the N×M matrix **Y** is revealed to be the matrix of partial derivatives of the joint torque with respect to the parameter vector **a**. In effect the adaptation law is a gradient descent procedure that demands the estimate of the parameter vector change in a particular direction in the M×1 parameter space at each point in time. That direction is the result of M dot product operations between the M column vectors on the right hand side of (2-21) and the direction of the tracking error, represented by the vector **s**. The result of each dot product is the magnitude of the projection of the tracking error onto the direction in joint space that represents the greatest change in torque due to a change in a parameter. The adaptation proceeds opposite to this direction in parameter space with $\mathbf{\Gamma}$ acting as a gain matrix dictating the speed of adaptation. The goal of the adaptation law is to reach the point in parameter space at which any change in the torque due to a change in the parameter vector will be orthogonal to the tracking error – in other words, the point at which the estimate of the parameters cannot be changed to improve tracking.

Several important properties of (2-13) are used in the proof of the stability of the control law (2-18)/(2-19). They are given in Table 2.1.

TABLE 2.1
IMPORTANT PROPERTIES OF MANIPULATOR DYNAMICS

Property 1	Linearity in Parameters (LIP) of dynamics.
Property 2	Symmetry and Positive Definiteness of $\mathbf{H(q)}$.
Property 3	Skew Symmetry of $\dot{\mathbf{H}}(\mathbf{q}) - \mathbf{C}(\mathbf{q},\dot{\mathbf{q}})$.

Property 1, the LIP property, means that the unknown parameters in the dynamics appear only as constants multiplying fixed functions of joint angles, velocities or accelerations or are added in as constant but unknown offsets. The LIP property allows the manipulator dynamics to be rearranged into the following form

$$H(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) = Y(q,\dot{q},\ddot{q})a \qquad (2\text{-}22)$$

where **Y** is an $N \times M$ matrix containing known functions that are parameterized by M constants, arranged in the $M \times 1$ vector **a**. The LIP property enables the controller introduced in (2-14) to be rearranging into the form on the right hand side of equation (2-17). It is especially important because it greatly simplifies adaptive control. The property will be further clarified in the dynamical model of the experimental system presented in Appendix A.4. Property 2 is a consequence of the kinetic energy $\dot{\mathbf{q}}^{\mathbf{T}}\mathbf{H}\dot{\mathbf{q}}$ always being positive, while Property 3, noted by Koditschek (1984), holds in general for rigidly linked manipulators. By definition it means that

$$\left(\dot{H}(q) - C(q,\dot{q})\right)^T = -\left(\dot{H}(q) - C(q,\dot{q})\right) \qquad (2\text{-}23)$$

An important remark is made in (Slotine and Li, 1988) about the distinction between the convergence of $\hat{\mathbf{a}}$ to **a** versus the convergence of the tracking error **s** to zero. It may be possible that the tracking error converges to zero without the estimated parameters $\hat{\mathbf{a}}$ converging to the actual values in **a**. However, under "persistent excitation" $\hat{\mathbf{a}}$ converges to **a** with the tracking error converging to zero. Persistent excitation is a term that describes the need to have the desired trajectories excite all the terms of the manipulator's dynamics through a diverse combination of accelerations, velocities and positions. Mathematically, the condition is satisfied if the matrix **Y** from the controller in (2-18), evaluated at the desired trajectory, meets the following condition

$$\alpha_1 I \leq \int_{t_1}^{t_1+\delta} Y_d^T Y_d dt \leq \alpha_2 I \tag{2-24}$$

where $\mathbf{Y_d = Y(q_d, \dot{q}_d, \dot{q}_d, \ddot{q}_d)}$, **I** is the M×M identity matrix and $\boldsymbol{\alpha_1, \alpha_2, \delta}$ are positive constants. From (2-24) persistency of excitation can be thought of as a condition calling for **Y** to span the entire M-dimensional parameter space over some time period $\boldsymbol{\delta}$. In practice checking the persistency of excitation of a trajectory via (2-24) is a difficult computation. Section 4.1 will discuss the training trajectory chosen, through trial and error, to persistently excite the system.

Slotine and Li's adaptive controller can be applied to the geared case (2-7) as long as the three properties still hold. Rewriting (2-7) without the friction and external torque terms yields

$$\left(H(q) + G^2 I_m\right)\ddot{q} + C(q, \dot{q})\dot{q} + g(q) = \tau \tag{2-25}$$

Because $\mathbf{G^2 I_m}$ is diagonal (2-25) can be put in an LIP form by simply taking the LIP form of (2-22) and adding the rotor inertia terms multiplying the component of $\ddot{\mathbf{q}}$. Because $\mathbf{G^2 I_m}$ and $\mathbf{H(q)}$ are both symmetric and positive definite their sum is symmetric and positive definite as well so Property 2 holds. Since $\mathbf{G^2 I_m}$ is also a constant matrix, the derivative of $\mathbf{H(q)} + \mathbf{G}^2\mathbf{I_m}$ is still $\dot{\mathbf{H}}\mathbf{(q)}$ so the skew-symmetric property, Property 3, holds. Because the gearing does not alter the three properties the proof outlined in (Slotine and Li, 1988) can be directly applied to the geared case with the new inertia matrix $\mathbf{H(q)} + \mathbf{G}^2\mathbf{I_m}$. As a result, the control law (2-18) and adaptation law (2-19) can be applied not only to system (2-13), but also to the geared system (2-25). The control and adaptation laws for the geared case are given by

$$\tau = \left(\hat{H}(q) + G^2\hat{I}_m\right)\ddot{q}_r + \hat{C}(q,\dot{q})\dot{q}_r + \hat{g}(q) - K_d s = Y(q,\dot{q},\dot{q}_r,\ddot{q}_r)\hat{a} - K_d s \quad (2\text{-}26)$$

$$\dot{\hat{a}} = -\Gamma Y^T s \quad (2\text{-}27)$$

where the $\mathbf{Y}$, $\mathbf{a}$ and $\mathbf{\Gamma}$ have been redefined because of the gearing.

It is important to note that an actual acceleration term is not used in this control law. As a practical matter, acceleration estimates tend to be quite noisy, especially when derived by twice differentiating encoder measurements of joint angles. The estimates often need to be heavily filtered to reduce the noise, which in turn adds delay and may cause system instability. Another advantage of this controller is that it does not require the inversion of the estimated inertia matrix.

Up to this point, the issue of friction has been ignored. A simple yet powerful adaptive control law has been given for the geared, frictionless system described by (2-25). The next logical step is to extend the adaptive control scheme to (2-7), the model that also includes friction.

An adaptive radial basis function (RBF) neural network will be used to learn the viscous friction $\mathbf{f_v}$, which is assumed to depend strictly on velocity and be decoupled between the joints. The assumption is made that nothing else is known about the shape of this function but that it can be approximated by a sum of the outputs of the nodes of the RBF network.

Each node of the neural network is characterized by a function and a coefficient multiplying it. The coefficients of the nodes are tuned online using a learning rule similar to the one used to tune the physical parameters of the known dynamics. For an N DOF manipulator the assumption is made (as in (Sanner and Slotine, 1992)), that the friction term is continuous and can be approximated by an RBF network as

$$\hat{f}_v(\dot{q}) = \begin{bmatrix} \sum_{k=k_{MIN}}^{k=k_{MAX}} \hat{c}_{1,k} g(h\dot{q}_1 - k) \\ \vdots \\ \sum_{k=k_{MIN}}^{k=k_{MAX}} \hat{c}_{N,k} g(h\dot{q}_N - k) \end{bmatrix} \tag{2-28}$$

where $\hat{\mathbf{c}}_{\mathbf{i,k}}$ represents the estimate of the coefficient of node k for the ith joint. Here each DOF has a neural network with (**$\mathbf{k_{max}}$ - $\mathbf{k_{min}}$ + 1**) nodes. The function **g** is the radial basis function of the neural network, in this case chosen to be the "hat" function. It should not be confused with the gravitational term of the robot dynamics.

$$g(x) = \begin{cases} 1 - |x|, & \text{if } |x| < 1 \\ 0, & \text{otherwise} \end{cases} \tag{2-29}$$

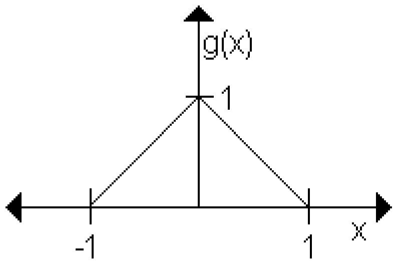


Figure 2.1 Hat basis function

Note that this function has radial symmetry about its center at zero, hence it qualifies as a radial basis function in one dimension. Each node's function is a shifted and scaled version of this basis function. For example, the **k**th node's function is **g(hx – k).** The parameter **h** determines the spacing between the centers of consecutive nodes, in effect the input resolution of the network. The center of each node is at $\mathbf{h^{-1}k}$ and the node's output is zero outside of $\mathbf{h^{-1}k \pm h^{-1}}$. Appendix A.5 describes the number of nodes used in this research as well as their spacing. The coefficients of the neural network are updated as follows

$$\dot{\hat{c}}_{i,k} = -\gamma_c g(hq_i - k)s_i \tag{2-30}$$

where $\boldsymbol{\gamma_c}$ is a scalar constant similar to the constant matrix $\boldsymbol{\Gamma}$ and $\mathbf{s_i}$ is the ith term of **s**.

The estimates of the derivatives of **a** and $\mathbf{c_{i,k}}$ generated by (2-19) and (2-30) respectively were numerically integrated every control cycle using the techniques shown in Appendix A.1. The use of a deadzone for **s** in the practical implementation of these adaptation laws is discussed when the experimental manipulator's controller is given in Appendix A.5. The final version of the adaptive friction learning control law is given in Section 2.5. The force estimation technique discussed in Section 2.4 will be broken up into a training and estimation/testing mode. During training the adaptation laws (2-19), (2-30) will be enabled while a special training trajectory is tracked. An important caveat is that during training, zero external force is assumed to be acting on the manipulator. After the training phase, the control law stops updating its estimates of $\mathbf{a}$ and $\mathbf{c}_{i,k}$ and the system switches into estimation mode. In practice, switching between training mode and estimation mode can be done on the fly by simply enabling or disabling the integrations implied by (2-19) and (2-30). If the updates were to continue in estimation mode, the friction learning neural networks would learn the joint torques needed to overcome the external torque in addition to the actual friction torque of the system, causing incorrect estimation.

2.4 Force Estimation

Before force estimation is presented, the relationship between external forces applied at the end effector and external torque at the joints will be discussed. The following well-known relationship holds between the joint velocities, $\dot{\mathbf{q}}$, and the end effector's Cartesian velocity, the 6×1 vector ${}^{A}\dot{\mathbf{x}}$:

$$ {}^{A}\dot{x} = {}^{A}J(q)\dot{q} \qquad (2\text{-}31) $$

where the $6\times N$ matrix ${}^{A}\mathbf{J(q)}$ is called the Jacobian of the manipulator. The left superscript "A" that appears twice in (2-31) signifies that the Cartesian velocity ${}^{A}\dot{\mathbf{x}}$ and Jacobian ${}^{A}\mathbf{J(q)}$ are expressed with respect to reference frame A, in this case an arbitrary frame. It is important to note that in general the Jacobian is configuration dependent, as indicated by its dependence on **q.** Note that **q** is not expressed with reference to a frame, since it represents relative joint displacements. In (2-31) the Cartesian velocity ${}^{A}\dot{\mathbf{x}}$ is broken up as follows:

$$^{A}\dot{x} = \begin{bmatrix} ^{A}\dot{p} \\ ^{A}\omega \end{bmatrix} \tag{2-32}$$

where $^{A}\dot{\mathbf{p}}$ is the 3×1 vector of linear velocity of the manipulator in Cartesian space and $^{A}\boldsymbol{\omega}$ is the 3×1 vector of angular velocity of the manipulator in Cartesian space. To compute the $6\times N$ Jacobian it is broken up as follows

$$^{A}J(q) = \begin{bmatrix} ^{A}J_{trans}(q) \\ ^{A}J_{rot}(q) \end{bmatrix} \tag{2-33}$$

where $^{A}\mathbf{J}_{\mathbf{trans}}$ is the $3\times N$ translational part of the Jacobian transforming joint velocity into end effector linear velocity and $^{A}\mathbf{J}_{\mathbf{rot}}$ is the $3\times N$ rotational part of the Jacobian transforming joint velocity into end effector angular velocity.

The Jacobian can now be used to describe the relationship between the 6×1 vector of external generalized forces (force and moment) acting on the end effector expressed in an arbitrary frame A, $^{A}\mathbf{F}_{\mathbf{ext}}$, and the $N\times1$ vector of torques seen at the manipulator's joints due to the external generalized force, $\boldsymbol{\tau}_{\mathbf{ext}}$ as derived in Craig (2005)

$$\tau_{ext} = {}^{A}J^{T}\,{}^{A}F_{ext}. \tag{2-34}$$

The external generalized force vector can be partitioned as follows

$$^{A}F_{ext} = \begin{bmatrix} ^{A}f_{ext} \\ ^{A}n_{ext} \end{bmatrix} \tag{2-35}$$

where $^{A}\mathbf{f}_{\mathbf{ext}}$ is the 3×1 external force vector and $^{A}\mathbf{n}_{\mathbf{ext}}$ is the 3×1 external moment vector. The opposite of (2-34) – namely the transformation from the external torque

$\boldsymbol{\tau}_{\mathbf{ext}}$ to the external generalized force ${}^{\mathbf{A}}\mathbf{F}_{\mathbf{ext}}$ is needed in force estimation. If the manipulator in question has six degrees of freedom, N = 6, the solution is to simply invert the transpose Jacobian because it is a square 6×6 matrix, to yield

$$ {}^{A}F_{ext} = ({}^{A}J^{T})^{-1}\tau_{ext}. \tag{2-36}$$

When $N \neq 6$ the manipulator is either under-constrained (N < 6) or over-constrained (N > 6) and this inversion cannot be performed. Instead the pseudo-inverse of the transpose Jacobian must be performed. The two cases are treated separately, following the example of Sabes (2001), who proves and justifies pseudo-inversion using optimization and Singular Value Decomposition (SVD) methods. In the under-constrained case the "right pseudo-inverse" is used. The right pseudo-inverse of a matrix **M,** denoted with the "plus" symbol, is given as

$$M_{R}^{+} = M^{T}\left(MM^{T}\right)^{-1} \tag{2-37}$$

substituting ${}^{\mathbf{A}}\mathbf{J}^{\mathbf{T}}$ for **M** yields

$$\left(J^{T}\right)^{+} = J\left(J^{T}J\right)^{-1} \tag{2-38}$$

where the frame of reference has been dropped to simplify notation. The inversion of the N×N matrix $\mathbf{J}^{\mathbf{T}}\mathbf{J}$ in this context is acceptable because the transpose Jacobian is assumed to have full row-rank of N. Because the Jacobian is configuration dependent (it depends on **q**) this assumption will break down if the manipulator is at or near a singular configuration and the pseudo-inverse will no longer be calculable.

In the over-constrained case, the "left pseudo-inverse" is used

$$M_L^{+} = \left(M^T M\right)^{-1} M^T \tag{2-39}$$

substituting ${}^{\mathbf{A}}\mathbf{J}^{\mathbf{T}}$ for **M** yields

$$\left(J^T\right)^{+} = \left(JJ^T\right)^{-1} J \tag{2-40}$$

The inversion of the 6×6 matrix $\mathbf{JJ^T}$ in this context is acceptable because the transpose Jacobian is assumed to have full column-rank of 6 unless at or near a singularity, where pseudo-inversion is not possible.

Equation (2-36) can be expanded to hold for all types of manipulators by writing

$${}^{A}F_{ext} = \mathrm{inv}({}^{A}J^T)\tau_{ext} \tag{2-41}$$

where **inv( )** is defined as

$$\mathrm{inv}(\ ) = \begin{cases} M_R^{+}, & N < 6 \\ M^{-1}, & N = 6\,. \\ M_L^{+}, & N > 6 \end{cases} \tag{2-42}$$

To find the external torque, equation (2-7) is rearranged to yield

$$\tau_{ext} = \left(H(q) + G^2 I_m\right)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) + f_v(\dot{q}) - \tau \tag{2-43}$$

Because the terms inside the parentheses on the right hand side of (2-43) are not known exactly, the estimates provided by the adaptive control laws (2-19) and (2-30) are used instead.

$$\hat{\tau}_{ext} = \left(\hat{H}(q) + G^2 \hat{I}_m\right)\ddot{q} + \hat{C}(q,\dot{q})\dot{q} + \hat{g}(q) + \hat{f}_v(\dot{q}) - \tau_{actual} \tag{2-44}$$

The external torque becomes an estimate due to the use of the estimates of the dynamical terms on the right hand side. Additionally, the term $\boldsymbol{\tau}$ is replaced with $\boldsymbol{\tau}_{\mathbf{actual}}$

$$\tau_{actual} = GK_m i_m \tag{2-45}$$

which is the actual motor torque at the manipulator's joints attained by converting the measured motor current $\mathbf{i_m}$ to torque using the gear ratios contained in $\mathbf{G}$ and the motor constants contained in the diagonal matrix $\mathbf{K_m}$. The force estimation equation can now be written by making use of (2-41) and (2-44)

$$^{A}\hat{F}_{ext} = inv(^{A}J^{T})\left(\left(\hat{H}(q) + G^{2}\hat{I}_{m}\right)\ddot{q} + \hat{C}(q,\dot{q})\dot{q} + \hat{g}(q) + \hat{f}_{v}(\dot{q}) - \tau_{actual}\right). \tag{2-46}$$

This equation provides the estimate of the external generalized force acting on the end effector given knowledge of the manipulator's kinematics, through the use of the Jacobian matrix, and its dynamics, including friction. It also makes use of the measurements of motor current, which is a feature often provided by the motor drivers, discussed in the next chapter. There are several issues related to the implementation of this equation in practice that are discussed in section 3.1.2. Namely (2-46) is broken up into a series of steps to filter noise added in by the motor current measurements and the calculation of acceleration.

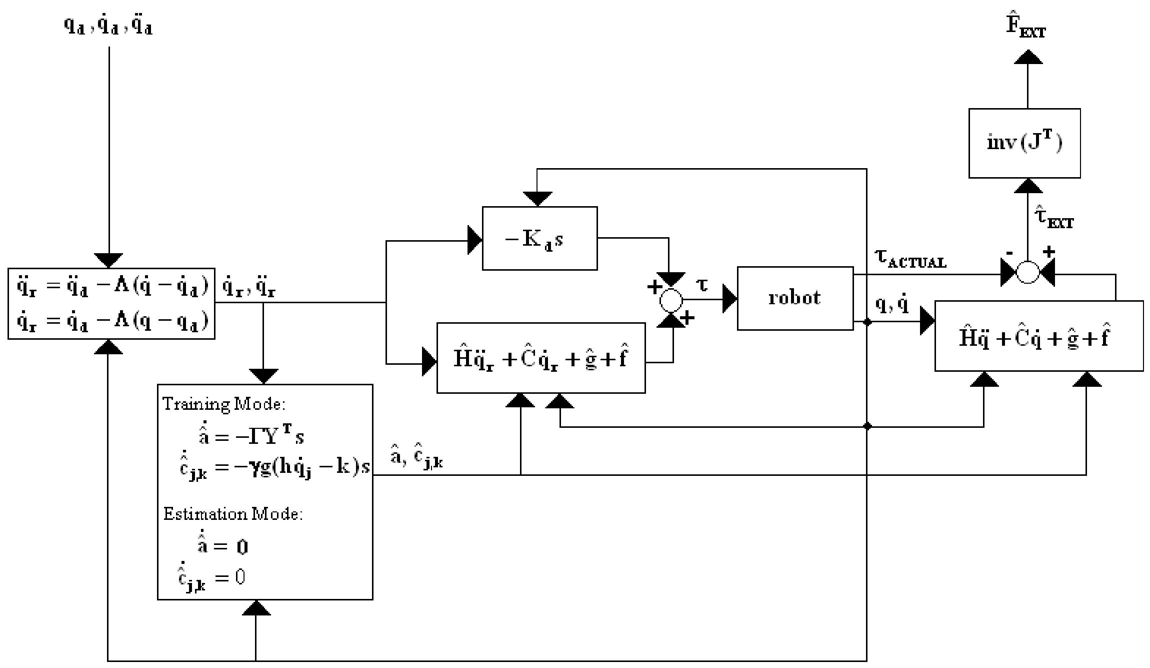

$q_d, \dot{q}_d, \ddot{q}_d$
$\ddot{q}_r = \ddot{q}_d - \Lambda(\dot{q} - \dot{q}_d)$
$\dot{q}_r = \dot{q}_d - \Lambda(q - q_d)$
$\dot{q}_r, \ddot{q}_r$
$-K_d s$
$\hat{H}\ddot{q}_r + \hat{C}\dot{q}_r + \hat{g} + \hat{f}$
τ
robot
τ_{ACTUAL}
$q, \dot{q}$
$\hat{H}\ddot{q} + \hat{C}\dot{q} + \hat{g} + \hat{f}$
$\hat{\tau}_{EXT}$
$inv(J^T)$
$\hat{F}_{EXT}$
Training Mode:
$\dot{\hat{a}} = -\Gamma Y^T s$
$\dot{\hat{c}}_{j,k} = -\gamma g(h\dot{q}_j - k)s$
Estimation Mode:
$\dot{\hat{a}} = 0$
$\dot{\hat{c}}_{j,k} = 0$
$\hat{a}, \hat{c}_{j,k}$

Figure 2.2 Block diagram of adaptive, friction learning position controller and force estimation based on learned dynamical model.

2.5 Summary

This chapter began by introducing the general form of a serial link manipulator's dynamics, including friction and gearing. A force estimation scheme was then presented based on using an adaptive control law to learn the manipulator's dynamical model. Equations for the adaptation of both the modeled and unmodeled parameters were also given. The following table summarizes the geared manipulator model, the controller with adaptation laws and the force estimation scheme. These equations will be referred to often in subsequent chapters.

TABLE 2.2
SUMMARY OF MANIPULATOR MODEL, CONTROL, AND FORCE ESTIMATION DISCUSSED

Manipulator Dynamical Model:

$$\left(H(q)+G^2 I_m\right)\ddot{q}+C(q,\dot{q})\dot{q}+g(q)+f_v(\dot{q})=\tau+\tau_{ext} \tag{2-47}$$

Adaptive, Friction Learning Control Law:

$$\tau=\left(\hat{H}(q)+G^2\hat{I}_m\right)\ddot{q}_r+\hat{C}(q,\dot{q})\dot{q}_r+\hat{g}(q)+\hat{f}_v(\dot{q})-K_d s= \\ Y(q,\dot{q},\dot{q}_r,\ddot{q}_r)\hat{a}+\hat{f}_v(\dot{q})-K_d s \tag{2-48}$$

$$\hat{f}_v(\dot{q})=\begin{bmatrix}\sum_{k=k_{MIN}}^{k=k_{MAX}}\hat{c}_{1,k}g(h\dot{q}_1-k)\\ \vdots \\ \sum_{k=k_{MIN}}^{k=k_{MAX}}\hat{c}_{N,k}g(h\dot{q}_N-k)\end{bmatrix} \tag{2-49}$$

Adaptation Laws in Training Mode:

$$\dot{\hat{a}}=-\Gamma Y^T s \tag{2-50}$$

$$\dot{\hat{c}}_{i,k}=-\gamma_c g(h\dot{q}_i-k)s_i \tag{2-51}$$

Adaptation Laws in Estimation Mode:

$$\dot{\hat{a}}=0 \tag{2-52}$$

$$\dot{\hat{c}}_{i,k}=0 \tag{2-53}$$

Force Estimation:

$${}^A\hat{F}_{ext}=inv({}^A J^T)\left(\left(\hat{H}(q)+G^2\hat{I}_m\right)\ddot{q}+\hat{C}(q,\dot{q})\dot{q}+\hat{g}(q)+\hat{f}_v(\dot{q})-\tau_{actual}\right) \tag{2-54}$$

$$\tau_{actual}=GK_m i_m \tag{2-55}$$

The force estimation technique presented consists of a training phase and testing phase. During the training phase, while the training trajectory is being tracked, zero external force is assumed. After the training phase is completed, the parameter adaptation laws are disabled and the manipulator enters the testing phase. The force estimation can then be used until a user or higher-level autonomy chooses to relearn the dynamical model.

So far the force estimation technique described has been developed for any general serial manipulator. The next chapter will describe the dynamics, kinematics, and controller of the specific manipulator used as well as both hardware and simulation details. The remaining chapters give the results of experiments involving force estimation and compliance control using the manipulator.

Chapter 3: Manipulator Case Study

Up until this point the theoretical aspects of this thesis have been discussed. Now the focus shifts to describing the set-up of the system that was used both in simulations and hardware experiments. The kinematics of the chosen manipulator is detailed as well as the hardware used, both mechanical and electrical, and the software used. Later chapters will go on to detail the results of the simulations and experiments performed on the hardware described here.

3.1 Manipulator Model

The manipulator used in this thesis was originally designed by the SSL as part of the Defense Advanced Research Projects Agency (DARPA) Modular On-Orbit Reconfigurable co-oPerative High-dexterity roBOT (MORPHbots) project (Akin, 2004). The goal of the project was to design and implement a set of small, light-weight robotic actuators that could be pieced together as needed by astronauts to perform a variety of tasks on-orbit.

The manipulator originally consisted of a MORPHbots 2 DOF "pitch-roll" module shown in Figure 3.1. A force/torque sensor and bar was later mounted onto the manipulator for verification of force estimates. The force/torque sensor was mounted on top of the second degree of freedom, the "roll". The final version of the manipulator is shown in Figure 3.2.

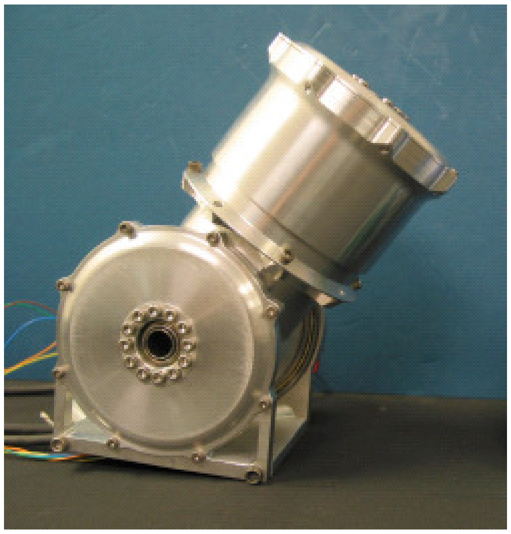


Fig. 3.1 MORPHbots two DOF module.

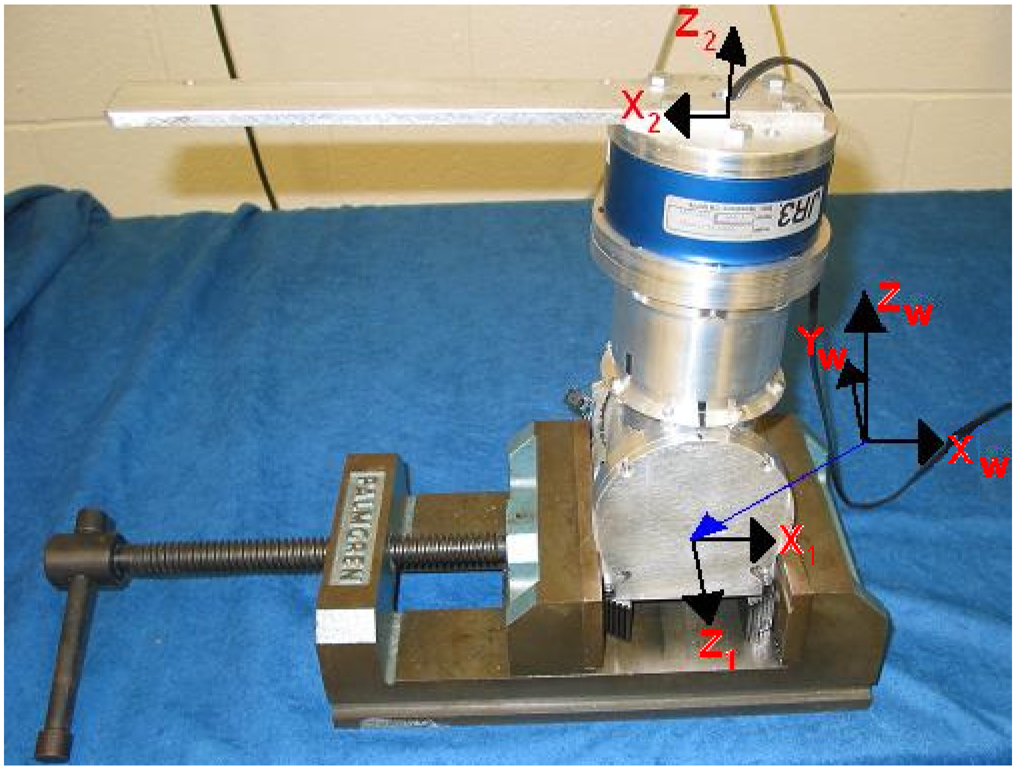


Fig. 3.2 The final version of the manipulator used, shown with frame definitions. Frame 1 is the pitch DOF frame, Frame 2 is the roll DOF frame. World frame origin is the same as Frame 1's, both at the center of joint 1. JR3 force/torque sensor shown is used to confirm force estimates.

3.1.1 Kinematics

A kinematics model of the manipulator shown in Figure 3.2 was formed using the Denavit-Hartenberg (D-H) convention (Craig, 2005). Figure 3.2 shows the two joint frames, Frame 1 and Frame 2, as well as the world frame. The world frame – the frame into which the force/torque sensor's data was transformed and in which all force estimates were made, was chosen so that its X-Y plane was parallel to the plane of the table top. The D-H parameters of the manipulator are given in Table 3.1. The ninety-degree value of the link twist α_0 comes about due to the choice of having the world frame and manipulator's 0 frame be the same for simplicity. Note that Frame 0, Frame 1 and the world frame all have the same origin – the point of intersection of Frame 1's z axis and Frame 2's z axis. The origin of the second frame is the center of the hole in the bar bolted into the plate atop the force/torque sensor (the hole was made for the sensor's data cable). Figure 3.2 shows the manipulator in the $\theta_1 = 0$, $\theta_2 = 180$ degrees configuration.

TABLE 3.1
MANIPULATOR DENAVIT-HARTENBERG (D-H) PARAMETERS

i	α_{i-1}	a_{i-1}	d_i	θ_i
1	90°	0	0	θ_2
2	-90°	0	L_1	θ_2

The forward kinematics, which is the transformation from joint angle configuration to Cartesian end effector position, is derived from this frame assignment. It is given by

$$ {}^{W}p = \begin{bmatrix} L_2 \cos\theta_1 \cos\theta_2 - L_1 \sin\theta_1 \\ L_2 \sin\theta_2 \\ L_2 \sin\theta_1 \cos\theta_2 + L_1 \cos\theta_1 \end{bmatrix} \tag{3-1} $$

where L_1 is the distance along the world frame's z-axis from the world frame's origin to Frame 2's origin. L_2 is the distance from frame 2 to the end effector, in this case the end of the bar shown in Figure 3.2. All forces were applied by hand to this point.

Because only force estimation was performed, only the translational part of the $6\times N$ Jacobian was used. The translational Jacobian matrix of a manipulator is in general derived by taking the partial derivative of the forward kinematics with respect to each joint variable (in this case θ_1 and θ_2).

$$
{}^{W}J_{trans} = \begin{bmatrix} \frac{\partial\, {}^{W}p}{\partial\theta_1} & \frac{\partial\, {}^{W}p}{\partial\theta_2} \end{bmatrix} =
$$

$$
\begin{bmatrix} -L_2\sin\theta_1\cos\theta_2 - L_1\cos\theta_1 & -L_2\cos\theta_1\sin\theta_2 \\ 0 & L_2\cos\theta_2 \\ L_2\cos\theta_1\cos\theta_2 & -L_2\sin\theta_1\sin\theta_2 \end{bmatrix} \qquad (3\text{-}2)
$$

The inverse of this matrix, used in (2-41) to transform estimated external torque into estimated external force, is given as

$$
\mathrm{inv}({}^{W}J^{T}_{trans}) = ({}^{W}J^{T}_{trans})^{+} = J(J^{T}J)^{-1} =
$$

$$
\frac{1}{L_1^2+L_2^2}\begin{bmatrix} -(L_1\cos\theta_1 + L_2\sec\theta_2\sin\theta_1) & -L_2\cos\theta_1\sin\theta_2 + L_1\sin\theta_1\tan\theta_2 \\ -L_1\tan\theta_2 & L_2^2\cos\theta_2 + L_1^2\sec\theta_2 \\ L_2\cos\theta_1\sec\theta_2 - L_1\sin\theta_1 & -L_2\sin\theta_1\sin\theta_2 + L_1\cos\theta_1\tan\theta_2 \end{bmatrix} \qquad (3\text{-}3)
$$

where the right pseudo-inverse is used instead of the true inverse because $N < 3$.

3.1.2 Force Estimator

The force estimator given by (2-54) was not used directly in practice. Instead, the equation was broken up into three parts. The first part involves estimating the external torque, formed by multiplying the $\mathbf{Y}$ part of the LIP form of the geared version for the modeled dynamics given in (2-25) by the estimated parameter vector $\hat{\mathbf{a}}$, forming the estimate of the torque due to the modeled dynamics. The actual torque at the joints, formed by converting motor current via (2-55), is then subtracted from this torque. The estimated torque is given by

$$\hat{\tau}_{ext} = Y(\ddot{q},\dot{q},q)\hat{a} + \begin{bmatrix} \sum_{k=k_{MIN}}^{k=k_{MAX}} \hat{c}_{1,k} g(h\dot{q}_1 - k) \\ \sum_{k=k_{MIN}}^{k=k_{MAX}} \hat{c}_{2,k} g(h\dot{q}_2 - k) \end{bmatrix} - GK_m i_m \tag{3-4}$$

In Section 2.3 it was noted that actual acceleration is not used in the adaptive, friction-learning controller. Instead the inertia matrix multiplies the reference acceleration signal. But equation (3-4) requires the use of actual acceleration so it is calculated by taking the second derivative of the position measurement provided by the encoders. Unfortunately this method is well known for yielding very noisy results. Digital low pass filtering can improve the signal at the expensive of adding delay. Typically higher order filters, which offer better results, are not used because delay can easily lead to instability in high bandwidth closed loop control. Fortunately, because the acceleration signal is not used directly in the position controller, but rather in generating force estimates for the lower bandwidth compliance controller, the use of a higher order filter is acceptable. The filter chosen was a fifth order elliptic low pass digital filter with 20 Hz cut-off frequency under 3 kHz sampling frequency (due to use within 3 kHz control frequency, discussed in section 3.3), 0.01 dB passband ripple and 40 dB attenuation in the stopband. Its coefficients were generated using the following MATLAB command: **[b, a] = ellip(5, .01, 40, 20/1500)**, where **b** is the vector of coefficients multiplying previous unfiltered samples and **a** is the vector of coefficients multiplying previous filtered samples (refer to the documentation of the **filter( )** function in the MATLAB Function Reference for further clarification of these vectors and digital filtering in general).

The next step involves filtering the estimated torque obtained in the first step. This is a crucial step because the estimated torque contains significant noise due to the motor current measurements. The motor current measurements, provided by the motor drivers, contain high frequency noise due to the high switching frequency (usually 20+ kHz) of their current-controlling transistors. The filter chosen for this step was the same fifth order elliptic filter used for filtering the actual acceleration, described in the previous paragraph. In chapters 4 and 5 it will be shown that additional thresholding and filtering of the torque were added at this step.

In the last step the force estimator is formed using the filtered version of (3-4) and the pseudo-inverse of the transposed translational Jacobian given in (3-3) as follows

$$ {}^{W}\hat{f}_{ext} = ({}^{W}J_{trans}^{T})^{+}\hat{\tau}_{ext\_filtered}. \tag{3-5}$$

The bandwidth of the force estimate thus obtained is limited to the bandwidth of the filter used on the estimated torque, in this case 20 Hz. Higher bandwidths may be possible though this aspect was not investigated to any great extent because the chosen bandwidth was deemed acceptable for the compliance control experiments. Note again that moment estimation was not performed.

3.2 Manipulator Hardware

3.2.1 Mechanical

The MORPHbots module used two Kollmorgen 01810-A brushless DC motors with integrated Hall effect sensors. The motor torque constant is $k_m = .0855$ N-m/A, the maximum continuous and peak current is rated at 5.28 A and 21.3 A respectively. The inertia of the housed motor was given as $3.74*10^{-5}$ kg*m^2 in the manufacturer's specifications.

Position was sensed using RS 40.4/25/1800 incremental encoder discs produced by Numerik Jena which provided 1800 encoder counts per revolution (CPR). The /1/2/B/040.4/1800/L/S encoder disk reading head increased this resolution by a factor of 5 using signal interpolation. The resulting 9000 CPR resolution was then quadrupled to 36000 CPR input pulses and sent to the counters on the DAQ board, discussed in Section 3.2.2.

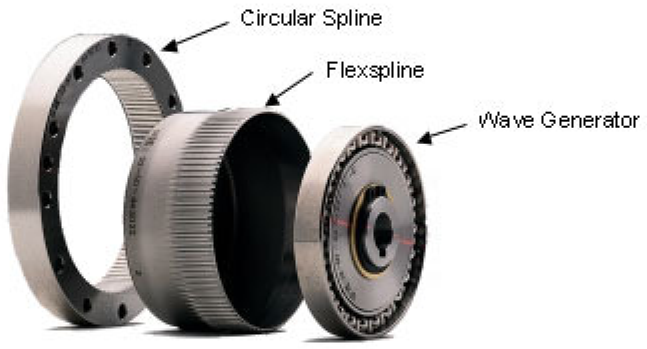


Figure 3.3 Harmonic drive components - from (Harmonic Drive, LLC, 2006).

Harmonic drives were originally developed in the late 1950's as an efficient method of enabling high gear ratios in motors. Harmonic drives have three components: a wave generator, a flexspline, and a circular spline shown in Figure 3.3. On the input end is the wave generator that is attached to the motor. The circular flexspline conforms to the elliptical outer surface of the wave generator when it is fit inside the flexspline. The flexspline is in turn fit inside the circular spline, which is a rigid circular steel ring. Both the circular spline and flexspline have teeth that mesh with each other. The elliptic shape of the flexspline induced by the wave generator causes the teeth to mesh in two opposite regions when the wave generator freely rotates inside the flexspline. High gearing ratios are possible because the flexspline has two less teeth than the circular spline so for each rotation of the wave generator the flexspline is moved by two teeth with respect to the circular spline. The output of the harmonic drive is attached to the flexspline with the gear ratio depending on the number of teeth in the circular spline (Harmonic Drive, LLC., 2006). Harmonic drives are widely used in space and underwater manipulators because they are compact, light, powerful, and offer very little backlash.

The harmonic drive used was model CSD-20-160 by Harmonic Drive, LLC. The gear ratio of the pitch DOF was 161:1 while the gear ratio of the roll DOF was 160:1. Because of the gearing, the encoder resolution of the manipulator increased to 161*36000 = 5796000 CPR for the pitch DOF and 160*36000 = 5760000 CPR for the roll DOF, which represents a resolution of about a millionth of a radian in both joints.

3.2.2 Electronics

The data acquisition electronics used in this thesis consisted of a National Instruments PCI-6025e data acquisition (DAQ) board. It has 16 analog inputs that are digitized to 12-bit resolution, 2 analog outputs that can supply +/-10 V to 12-bit resolution, 8 digital I/O lines and 2 24-bit counter/timers. The DAQ board allowed information to be passed between the control computer and the external electronic components: the encoder electronics and the current drivers. Because the DAQ board's onboard counters accepted signals in a different form than was provided by the encoder electronics embedded in the manipulator a special chip was used to convert the signals. The details of these signals and the chip's operation are provided in Appendix A.2.

A motor driver was used to power each of the two motors. Though two slightly different models were used, both were made by Advanced Motion Controls and their behavior was essentially identical. Both models were designed to drive brushless motors, meaning that they were capable of brushless motor commutation: reading the magnetic state via the Hall effect sensor inputs from the motor and controlling the desired level of current in the three phase motors. The B15A8 and B30A8 current driver models were used. The B15A8 model is capable of driving +/-7.5 A of continuous current at a switching frequency of 33 kHz with a DC supply of 20-60V while the B30A8 model is capable of driving +/-15 A of continuous current at a switching frequency of 22 kHz on a DC supply of 20-80V. They were powered at 30V in this work. Both models have peak current ratings of double their continuous current ratings and can operate in open loop mode, current mode and tachometer mode though the B30A8 model can also operate in Hall velocity mode. Because the drivers' purpose in this research was to control current, assumed to be proportional to the commanded torque through the motors' torque constant, the drivers were both put into current mode. Importantly, both motor drivers provided real-time measurements of the actual current in the motors.

3.2.3 Force/Torque Sensor

The force/torque sensor used to verify force estimates was a JR3 100M40A 100 mm diameter, 40 mm thickness with a maximum load of 200 lbs in the Z direction (direction perpendicular to sensor face) and 100 lbs in the X and Y directions. The moment ratings are about 66 ft-lbs in the Z direction and 33 ft-lbs in the X and Y directions

Because the manipulator is a 2 DOF non-planar type, the 3×1 estimated force vector has only two true directions of estimation, which vary according to the manipulator's configuration. To ensure that the force/torque sensor's force vector was only along those two directions, the following transformation was performed on its 3×1 force vector:

$$^{W}\mathrm{f}_{\mathrm{ext}} = \mathrm{J}(\mathrm{J}^{\mathrm{T}}\mathrm{J})^{-1}\mathrm{J}^{\mathrm{T}}\,{}^{W}\bar{\mathrm{f}}_{\mathrm{ext}} \qquad (3\text{-}6)$$

where $\mathbf{J}(\mathbf{J}^{\mathrm{T}}\mathbf{J})^{-1}\mathbf{J}^{\mathrm{T}}$ is a 3×3 matrix with rank at most 2 and ${}^{W}\bar{\mathbf{f}}_{\mathbf{ext}}$ is the original 3×1 force vector provided by the force/torque sensor in the world frame. This matrix is the result of transforming the force vector to joint torque using the transpose Jacobian, then using the pseudo-inverse of the transpose Jacobian. It can be found by multiplying the transpose of (3-2) by (3-3).

Before (3-6) can be applied the force/torques sensor's readings must be transformed from the sensor (FTS) frame, which depends on both joint positions, to the world frame (W) shown in Fig. 3.2, in which frame all the force estimates made later in Chapters 4 and Chapter 5 are set. The transformation is given as

$$ {}^{W}\bar{f}_{ext} = {}^{W}_{0}R \; {}^{0}_{1}R \; {}^{1}_{2}R \; {}^{2}_{FTS}R \; {}^{FTS}\bar{f}_{ext} = {}^{0}_{1}R \; {}^{1}_{2}R \; {}^{FTS}\bar{f}_{ext} = $$

$$ \begin{bmatrix} \cos q_1 \cos q_2 & -\cos q_1 \sin q_2 & -\sin q_1 \\ \sin q_2 & \cos q_2 & 0 \\ \sin q_1 \cos q_2 & -\sin q_1 \sin q_2 & \cos q_1 \end{bmatrix} {}^{FTS}\bar{f}_{ext} \tag{3-7} $$

Equation (3-7) uses the fact that the transformation of forces between frames depends strictly on the rotation matrix between the two frames. This does not hold for transforming both forces and moments between frames – see (Craig, 2005). The two rotation matrices ${}^{0}_{1}\mathbf{R}$, ${}^{1}_{2}\mathbf{R}$ used are found directly from the D-H parameters given in Table 3.1. Also used is the fact that the rotations between the world frame and Frame 0 as well as between the force/torque sensor's frame (FTS) and Frame 2 both equal the identity matrix.

3.3 Manipulator Software

The control program was written in C and run on a Dell Dimension^{TM} 8400 computer containing a 3.6 GHz Pentium^{TM} 4 processor and 1 GB of RAM. The computer was running distributed Timesys real-time Linux kernel 2.6.16.9. Coding was done on an Apple iMac^{TM} G5 with a 2.1 GHz PowerPC processor and 1 GB RAM using the Xcode editor. The NI DAQ board was used with Comedi drivers - Comedi is a set of Linux open source drivers for various commercial DAQ boards.

The Linux driver for the JR3 force/torque sensor was written by Mario Prats at UJI (Spain).

The control program was broken up into two threads – one a real-time thread used to realize the digital controller at the desired 3 kilohertz control frequency and the other a data logging thread not operating in real-time. The real-time control thread was responsible for reading the DAQ board's inputs to the computer, generating the desired voltage, proportional to desired torque calculated by the control law, and sending it back to the DAQ board within the 333 microsecond control period. Error in waking from sleeping at the end of the previous cycle was tolerated to within ± 50 microseconds of the desired time. The real-time kernel enabled the high control frequency with tight timing.

Data generated by the control thread cannot be saved directly to file because of potential buffer overflow that could hang the thread and cause it to miss timing deadlines. Instead the control thread would push a structure containing the current state information (actual and desired trajectory, sensed and estimated force, etc.) onto a queue at 100 Hz. A queue is a first in, first out data structure meaning that data that is pushed (added) onto the queue earlier is popped (removed) off of it sooner. The data logging thread consisted of an infinite loop that would continuously attempt to pop the data off of the queue. Occasionally (every twenty seconds) the control thread would also push the current values of the adapted parameters into another queue, which the same data logging thread would also continuously check. The communication between these threads using the queues is illustrated in Figure 3.4. Because the data logging thread operates at a lower priority, if it gets hung due to file buffer overflow the control thread's higher priority enables it to regain the processor's attention and avoid missing deadlines.

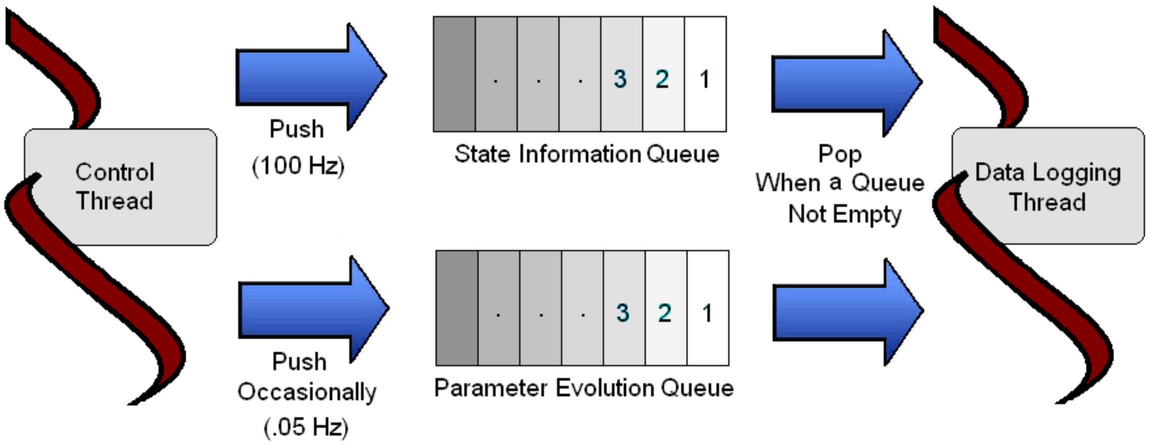


Figure 3.4 Data logging using inter-thread communication through queues.

3.4 System Integration

The fully integrated system is shown in Figure 3.5. It serves to illustrate the relationships between the various components of the experimental set-up. The computer, responsible for executing the control code, communicates with the data acquisition (DAQ) board via its PCI bus. The DAQ board sends the computer two analog input voltages, proportional to actual motor current, and two counter readings and receives two analog output voltages, proportional to desired motor current. The computer also communicates with the force/torque sensor's receiver board via the PCI bus. These signals are all exchanged during each control cycle.

The current monitoring feature of the motor drivers feeds the DAQ board's analog input voltages. The voltages from the drivers' current monitor output lines are proportional to the actual current in the motors (1V = 2A). An analog resistorcapacitor low pass filter is placed between the drivers and the DAQ board to act as an anti-aliasing filter. The DAQ board's two onboard counters are fed by the output lines of the LS7184 chips, described above in Section 3.2.2 and further in Appendix A.2, whose inputs are fed by the encoder electronics embedded in the manipulator.

Each of the two motor driver receive the output of one of the DAQ board's two digital to analog converter (DAC) output voltages proportional to desired current. The motor drivers also receive the input from the Hall effect sensors mounted on the motors. From these input signals the motor drivers output voltage into the manipulator's motors at their switching frequency.

Figure 3.6 shows the actual hardware in the fully integrated state. In the next chapter the force estimation ability of the system described in this chapter will be demonstrated using both the hardware and a simulation of the model described in this chapter.

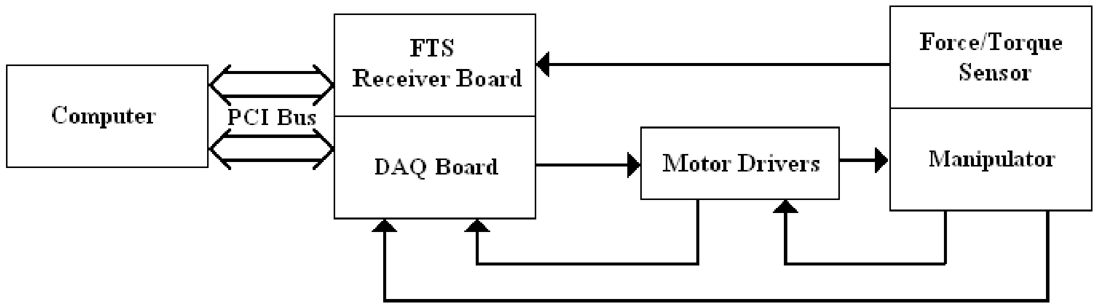


Figure 3.5: Fully integrated system - block diagram.

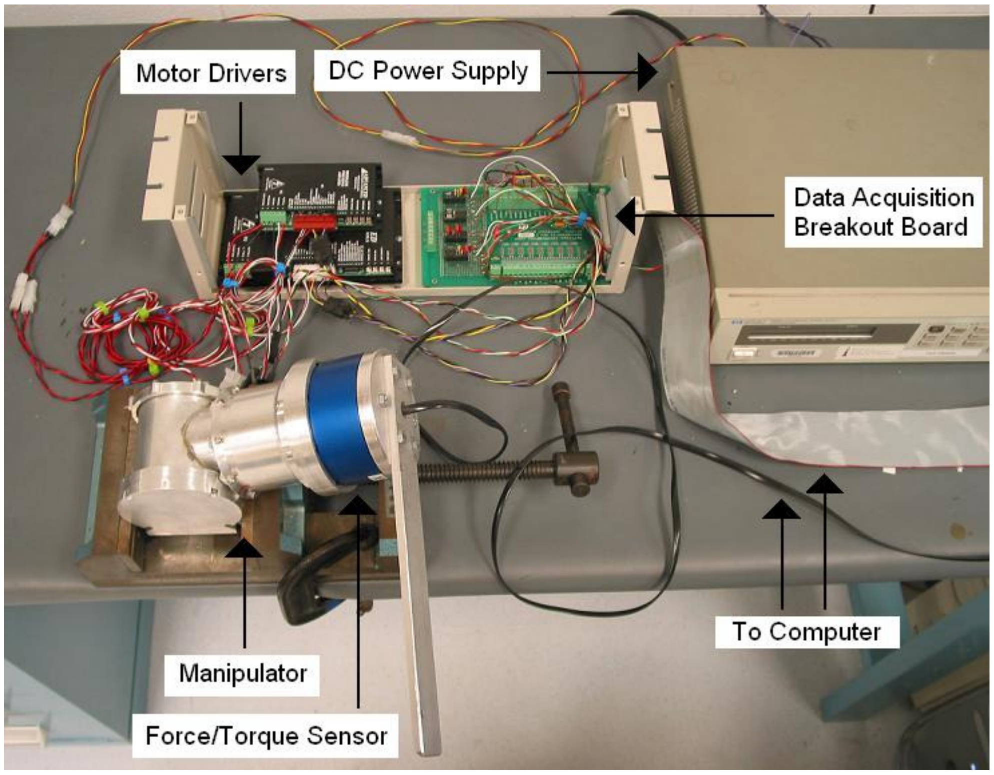


Figure 3.6: Fully integrated system - hardware.

3.5 Simulation Set-up

For the simulation, the dynamics and kinematics of the manipulator were described by (A-17) and (3-1) respectively. The same control and adaptation laws given in (A-18) were used during trajectory tracking. Just as in the case of the hardware, the parameter values were assumed to be unknown to the controller, i.e. the estimated value of the parameter vector **a** was equal to zero initially.

The difference between the simulation and hardware comes about when friction is considered. In the case of the hardware, the specifics of the friction function of each joint are unknown. The friction is simply assumed to be velocity-dependent and decoupled between joints. In the case of the simulation, the friction is specified and the accuracy of the adaptive networks' estimate of it is explicitly known. The friction function used in simulation is given here as

$$f_{v-SIM}(\dot{q}) = \begin{bmatrix} .25\left(1-e^{-|\dot{q}/.15|}\operatorname{sign}(\dot{q})\right)+.25\dot{q} \\ .25\left(1-e^{-|\dot{q}/.15|}\operatorname{sign}(\dot{q})\right)+.25\dot{q} \end{bmatrix} \tag{3-8}$$

The friction function, chosen to be the same in both joints, is a version of the well-known coulomb plus viscous model (Olsson et al., 1998) where the discontinuity through zero velocity has been smoothed. The function is plotted in red in Figure 4.4. The smoothing was added because, as explained below, the input resolution of the networks was worse in simulation than in the hardware control code.

As in hardware case, 41 nodes were used in the joints' adaptive networks but in the simulation case the range of the joints' velocity was chosen to be [-1.9 rad/s, 1.9 rad/s] instead of [-1 rad/s, 1 rad/s] as in the hardware case, due to the use of a training trajectory with larger amplitude for the simulation (discussed further in section 4.1). This led to having $\mathbf{h = 10.5}$ in simulation, which is worse input resolution than $\mathbf{h = 20}$. The poorer resolution was due to the fact that adding more nodes to the networks meant greatly increasing the running time of the simulation.

Appendix A.6 presents the constants used both in the simulation and the hardware experiments.

3.6 Summary

In this chapter the specific manipulator used in latter chapters' experiments was introduced and its kinematics were given. The specifics of the adaptive, friction-learning controller for this manipulator were discussed. The details of the force estimator's practical implementation were given in a three-step procedure intended to filter noise due to the current measurements. Filtering of the actual acceleration signal for use by the force estimator was also discussed. The hardware, software, and their integration were then described. Finally, the details of the simulation were given.

Chapter 4: Force Estimation Experiments

This chapter details the experiments used to demonstrate the force estimation technique presented in Chapter 2 with the system described in Chapter 3. The choice of training trajectory used to learn the parameters of the manipulator's modeled dynamics, discussed in Appendix A.4, as well as the coefficients of the adaptive networks for the unmodeled friction will be discussed. The evolution of both types of parameters during the course of this training trajectory will then be presented. Finally the results of force estimation during simulation as well as hardware experiments will be given for both stationary and dynamic testing trajectories.

4.1 Training Trajectories

Different training trajectories were used for the simulation and hardware experiments. To keep the running time of the simulation reasonable, the input resolution of the adaptive networks was lower than in the hardware experiments. As a result, a smoother friction model was used that could be learned more easily by the less densely spaced nodes. In the case of the simulation, a training trajectory consisting of a single sinusoidal signal was sufficient for both the modeled and unmodeled parameters to converge. The simulation trajectory is given as

$$q_d(t) = \begin{bmatrix} -2.2\cos\left(2\pi(0.1)t\right) + 2.2 \\ -2.2\cos\left(2\pi(0.1)t\right) + 2.2 \end{bmatrix} + q_0 \tag{4-1}$$

$$\dot{q}_d(t) = \begin{bmatrix} 2.2\left(2\pi(0.1)\right)\sin\left(2\pi(0.1)t\right) \\ 2.2\left(2\pi(0.1)\right)\sin\left(2\pi(0.1)t\right) \end{bmatrix} \tag{4-2}$$

$$\ddot{q}_d(t) = \begin{bmatrix} 2.2(2\pi(0.1))^2\cos(2\pi(0.1)t) \\ 2.2(2\pi(0.25))^2\cos(2\pi(0.1)t) \end{bmatrix} \quad (4\text{-}3)$$

where the initial position of the joints is given as $\mathbf{q}_0^T = [-1.1 \quad -\pi]$.

In the case of the hardware experiments, the nodes of the adaptive networks were more densely spaced but the actual friction was known to be less smooth than in the simulation. This is because harmonically driven manipulators have a high amount of "stiction", i.e. static friction. As a result, a more persistently exciting training trajectory was chosen to enable learning of both the more diverse friction dynamics. The first joint's trajectory was composed of the superposition of two sinusoidal signals – each of different amplitude and frequency. The second joint's trajectory was composed of one sinusoidal signal. The desired joint position, velocity, and acceleration for both joints were as follows:

$$q_d(t) = \begin{bmatrix} -0.8\cos(2\pi(0.1)t) - 0.2\cos(2\pi(0.4)t) + 0.8 + 0.2 \\ -0.5\cos(2\pi(0.25)t) + 0.5 \end{bmatrix} + q_0 \quad (4\text{-}4)$$

$$\dot{q}_d(t) = \begin{bmatrix} 0.8(2\pi(0.1))\sin(2\pi(0.1)t) + 0.2(2\pi(0.4))\sin(2\pi(0.4)t) \\ 0.5(2\pi(0.25))\sin(2\pi(0.25)t) \end{bmatrix} \quad (4\text{-}5)$$

$$\ddot{q}_d(t) = \begin{bmatrix} 0.8(2\pi(0.1))^2\cos(2\pi(0.1)t) + 0.2(2\pi(0.4))^2\cos(2\pi(0.4)t) \\ 0.5(2\pi(0.25))^2\cos(2\pi(0.25)t) \end{bmatrix} \quad (4\text{-}6)$$

where the initial position of the joints is given as $\mathbf{q}_0^T = [-1.1 \quad -\pi]$. The complete desired training trajectory was composed of (4-4) to (4-6). Joint 2's initial position is the same as that shown in Figure 3.2, while joint 1's initial position is rotated 68.8 degrees clockwise (-1.2 radians about its z-axis) from the position shown in Figure 3.2. Figure 4.1 depicts (4-4) graphically for the first 40 seconds of the training – the total time this training trajectory was applied for was 600 seconds, which corresponds to 60 cycles of joint one's periodic waveform and 150 cycles of joint two's periodic waveform.

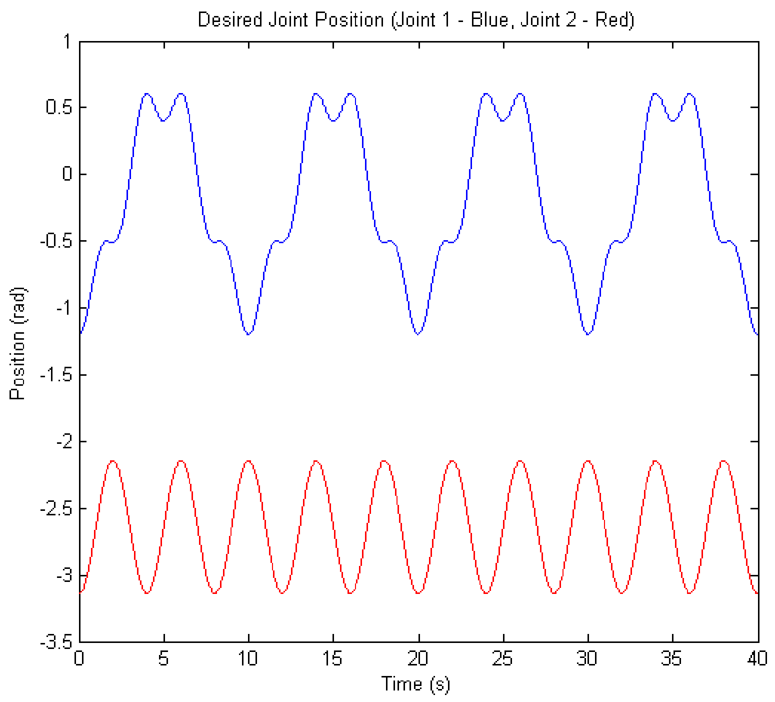


Figure 4.1 Desired joint position versus time for the hardware case.

From Figure 4.1 it can be seen that the first joint's trajectory is the superposition of two sinusoids of differing frequencies – a 0.1 Hz signal and a 0.4 Hz signal of smaller amplitude. The superposition forces the first joint to accelerate and decelerate under a more diverse set of gravitational loads, helping with the adaptation of the three parameters of joint 1 that significantly affect the dynamics (see Appendix A.4). The three parameters consisted of two gravitational parameters and one inertial parameter. It was determined experimentally that a training trajectory consisting of a single sinusoidal frequency for joint 1 did not did not meet the persistent excitation condition as stated in chapter two.

The second joint's modeled dynamical parameters were not as hard to adapt because only two were significant – its inertia due to gearing and the gravitational term that depends on both joints' position. The situation was further simplified by the fact that the gravitational term was also being adapted by the first joint, as can be seen from equation (A-29). As a result a simpler training trajectory consisting of a 0.25 Hz sinusoidal signal was sufficient for this joint's parameters to converge to the modeled values.

Note that at the beginning of the trajectory (t = 0), the desired position for both joints in both the simulation and hardware cases equaled the initial position and the desired velocity was zero. Because the manipulator was assumed to be stationary at the beginning of training, using an initial desired velocity of zero ensured that there was no initial velocity error, which would have caused a sharp spike in the commanded torque via the PD term.

4.2 Parameter Evolution During Training

The training trajectories just described were chosen so that the four parameters, discussed in Section 3.1.3, converged to within 20% of their theoretical values in both the simulation and hardware experiments. The other parameters were considered to have converged if their values remained small. Sections 4.2.1 and 4.2.2 give the adapted parameter vector for the simulation and hardware experiments respectively. In both cases the estimated parameter vector as well as the coefficients of the adaptive networks were set to zero at the start of training. The estimated parameter vector after training in both cases will be compared to (A-17).

4.2.1 Parameter Evolution - Simulation

After the simulation's training trajectory was tracked for 600 seconds the value of the vector was:

$$\hat{a}_{SIM}^{T} = [1.012 \quad 0.036 \quad 1.004 \quad 0.031 \quad -0.009 \quad 0.038 \quad 0.012 \quad -3.063 \quad 0.284] \qquad (4\text{-}7)$$

As can be seen by comparing (4-7) with (A-17), the four parameters discussed in Appendix A.4 – the first, third, and last two, converged to within 0%, 3%, 1%, and 29%. The last parameter, the smallest, did not converge to within 20% of the actual value due to error in its hundredth place. The progress of the parameters over the course of their adaptation during training is plotted in Figure 4.2. From the figure it can be seen that the first and eighth parameters take the longest to converge.

The progress of the coefficients of the adaptive networks over the course of the training trajectory can be seen in Figure 4.3. The second joint's coefficients converge quickly to their correct values while the first joint's coefficients evolve incorrectly at first before eventually converging correctly. The final estimate of the each joint's friction is seen in Figure 4.4. The error in the friction estimate of joint 2 is less than 5% of the maximum value of the friction over the range of joint velocities trained on. The error of joint 1 is less accurate but still less than 10% of the maximum value of the friction over the range of joint velocities trained on. The reason for poorer friction learning in joint 1 is that the joint is subject to higher gravitational dynamics (the eighth term) during training, which it had to adapt along with the other shared gravitational term (the ninth term, also being adapted by the first joint). Therefore the friction mapping provided by the first joint's adaptive network did not converge to the actual model until the eighth parameter converged, which can be seen from Figure 4.2 to happen late in the training period.

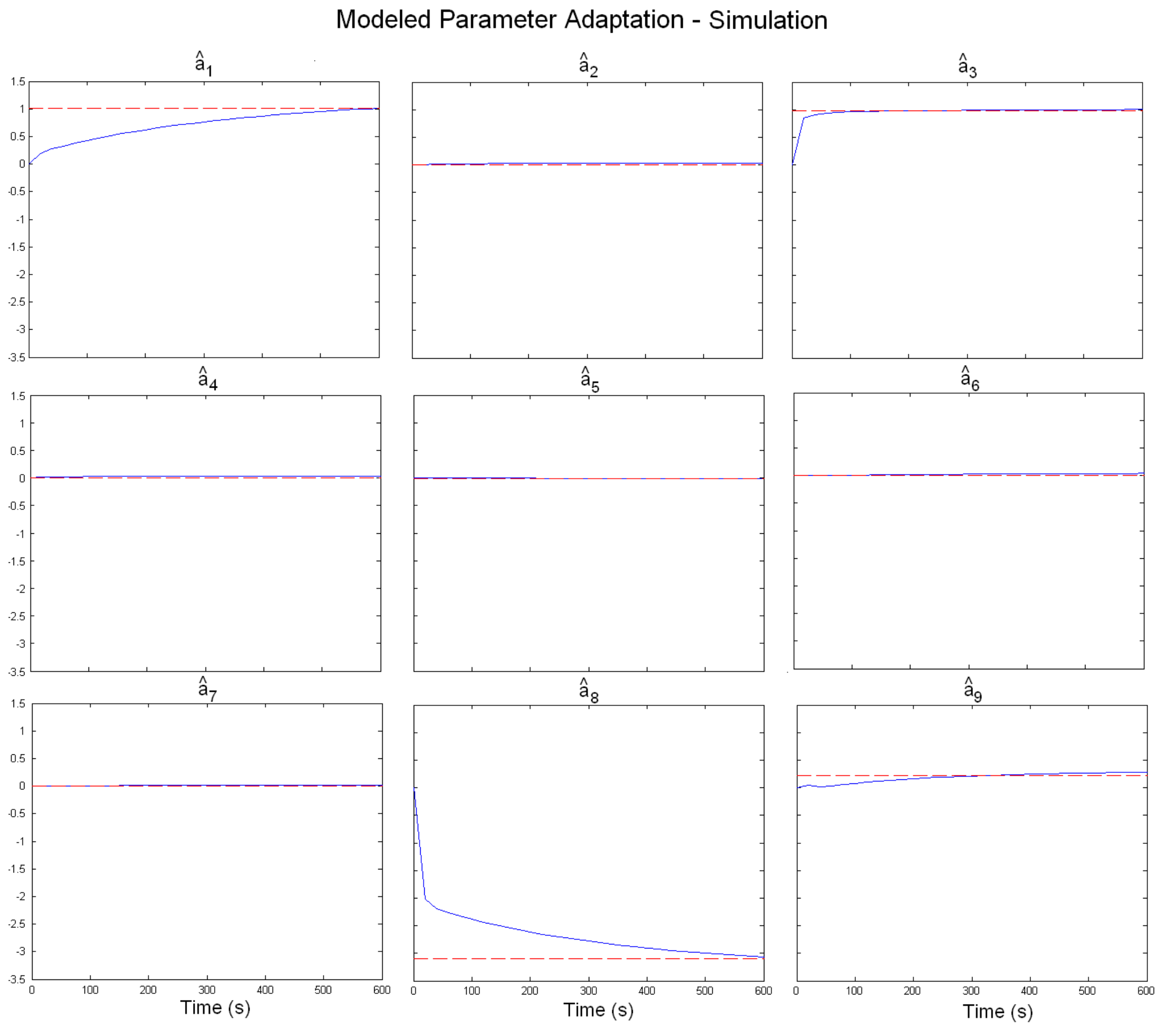


Figure 4.2 Adaptation of the modeled parameters over the course of the training trajectory for the simulation case.

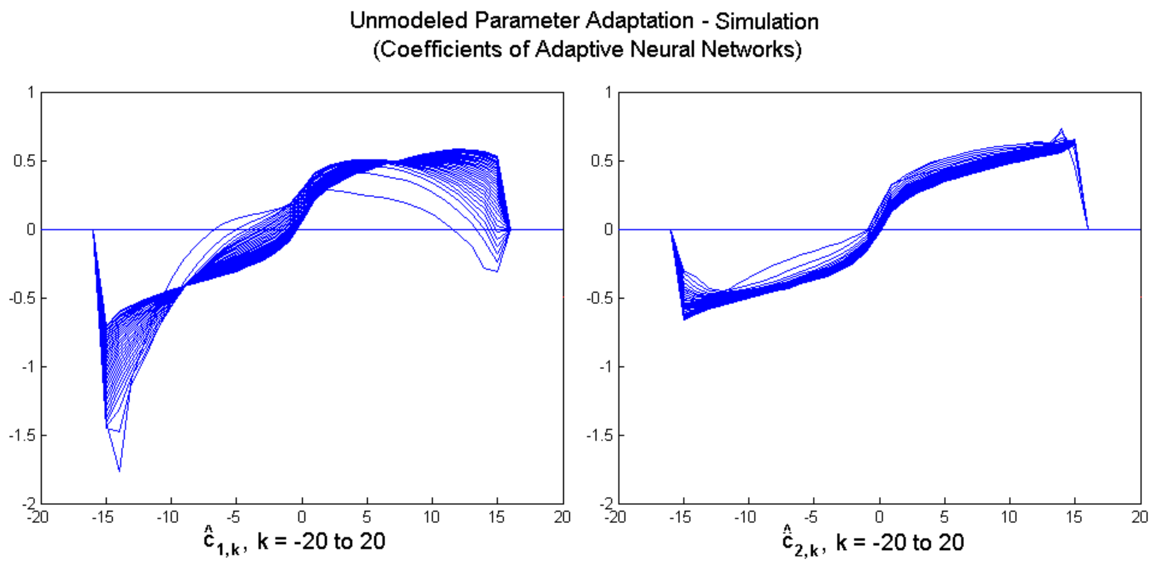


Figure 4.3 Adaptation of the unmodeled parameters of the dynamics over the course of the training trajectory for the simulation case.

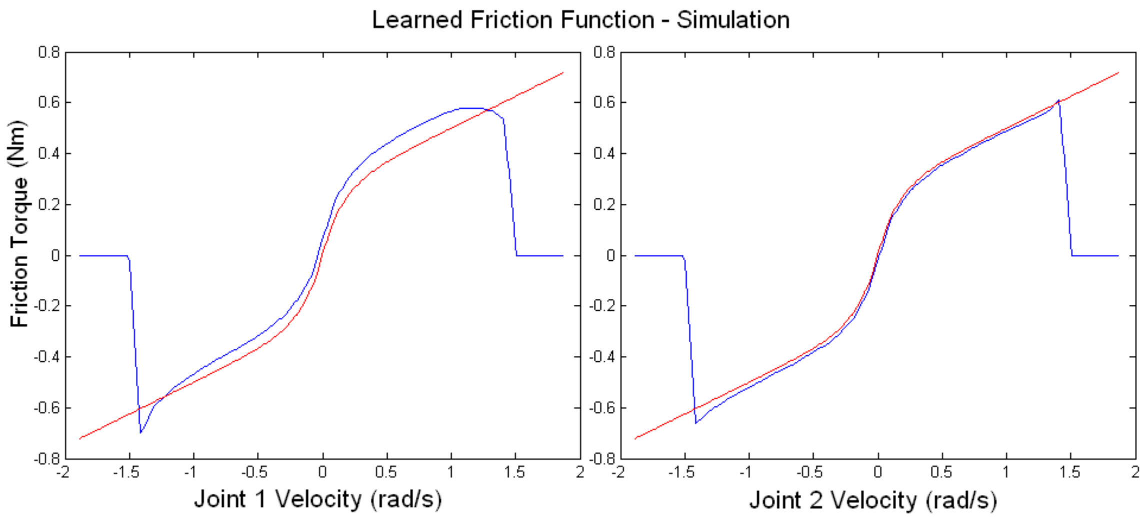


Figure 4.4 Mapping from velocity to friction torque learned by each joint's adaptive network for the simulation case.

4.2.2 Parameter Evolution - Hardware Experiment

After the training trajectory for the hardware was tracked for 600 seconds the value of the vector was:

$$\hat{a}_{HW}^{T} = [1.166 \quad 0.012 \quad 1.144 \quad -0.004 \quad 0.002 \quad -0.005 \quad -0.006 \quad -3.574 \quad 0.202] \qquad (4\text{-}8)$$

As can be seen by comparing (4-8) with (A-17), the parameters the four parameters discussed in Appendix A.4 – the first, third, and last two, converged to within 15%, 17%, 15%, and 8%. With the exception of the last parameter these results are not as close to the parameter vector (A-17) as they were in the simulation case. On the other hand the actual value of these parameters is not known – the parameter vector these values are being compared to is based on modeling, which is merely the best guess at the values based on what is assumed to be known. The inertia due to gearing, for example, which accounts for the majority of the first two significant parameters, is based on the manufacturer's specifications of the inertia of the housed motors, which may be incorrect by as much as 10-20%.

The progress of the parameters over the course of their adaptation during training is plotted in Figure 4.5. From the figure it can be seen that the parameters converge more quickly to near their final values than in the simulation case. This implies that a shorter training period could probably have been used in the hardware case, though it was kept at 600 seconds for comparison with the simulation.

The adaptation of the coefficients of the adaptive neural networks describing the mapping from joint velocity to friction torque is shown in Figure 4.6. From the figure two things are clear: the first is that both functions quickly converge close to their final form. Since the coefficients of the networks were saved every twenty seconds during training, most of the convergence occurs sometime in the initial twenty seconds. It is also clear that the first joint is subject to larger amounts of friction than the second. At $k = \pm 15$, which corresponds to a velocity of ± 0.75 rad/s ($\pm 15/h$, $h = 20$) the friction torque for the first joint is about ± 8 Nm while it is only about ± 5

Nm for the second joint at the same velocities. This result corresponds well with the qualitatively observed behavior: joint 1 appeared to have more friction than joint 2 when forces were exerted on it manually. The final estimate of the each joint's friction is seen in Figure 4.7.

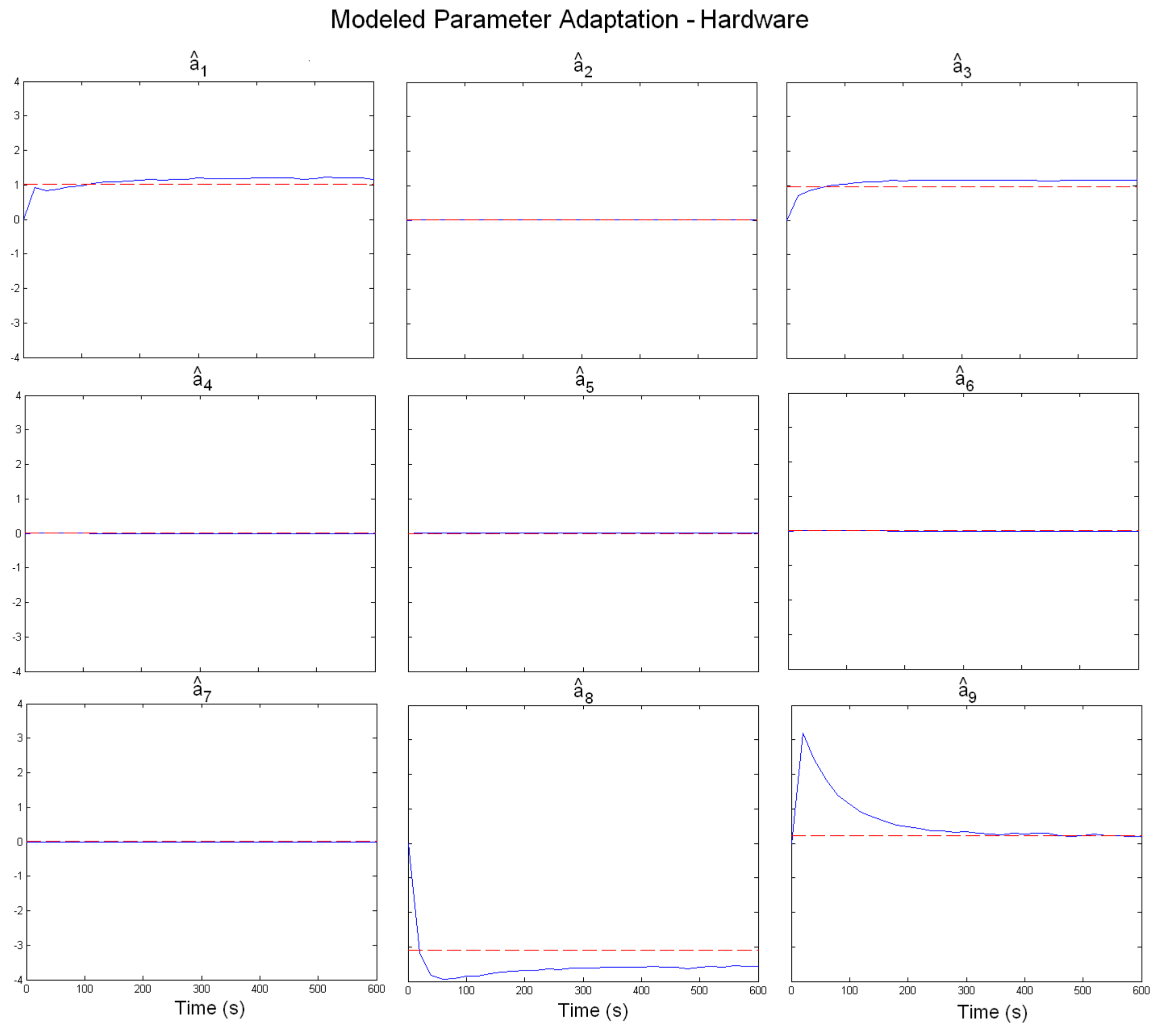


Figure 4.5 Adaptation of the modeled parameters over the course of the training trajectory for the hardware case.

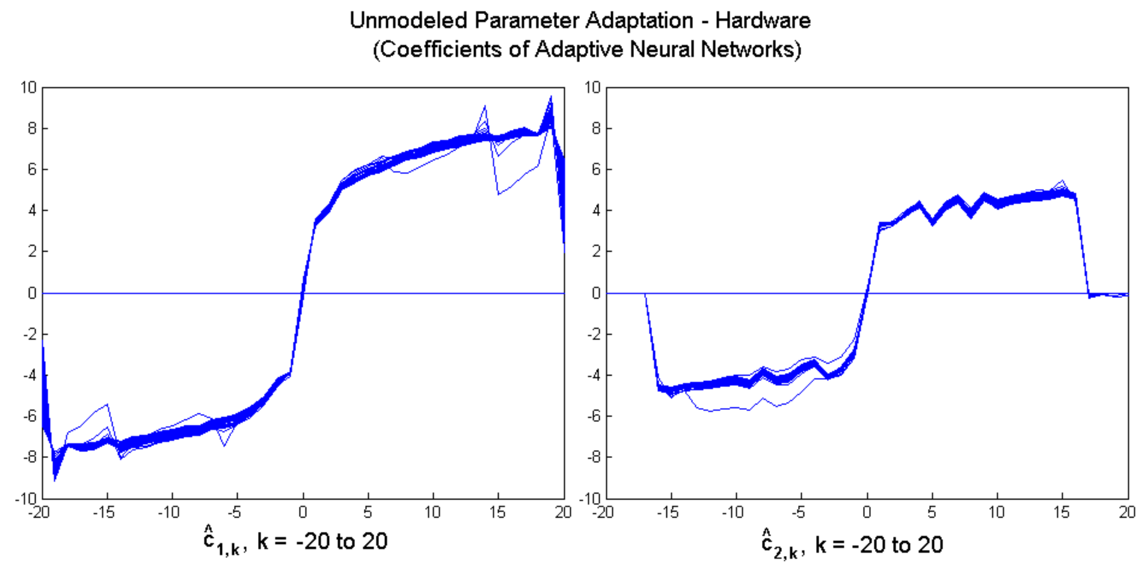


Figure 4.6 Adaptation of the unmodeled parameters of the dynamics over the course of the training trajectory for the hardware case.

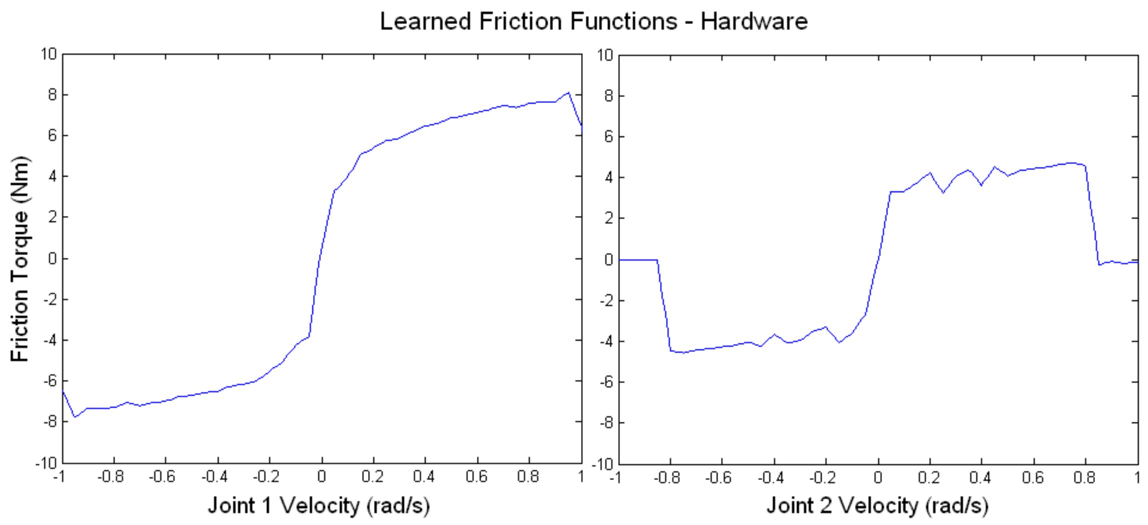


Figure 4.7 Mapping from velocity to friction torque learned by each joint's adaptive network for the hardware case.

4.3 Force Estimation Results for Stationary Testing

The ability to estimate forces while maintaining a fixed position is shown in the next two sections. For the position chosen in the stationary test, the same as the one shown in Figure 3.2, which is expressed in joint variables as $\mathbf{q}^T = [0 \quad -\pi]$, the direction of estimation provided by the first joint clearly has components in both the x and z directions of the world frame. This because its moment arm, the vector from the end of the bar to the joint's center, is a vector in the x-z plane and its axis of rotation is entirely in the y direction. The direction of estimation provided by the second joint is the y direction because the second joint's moment arm is entirely in the x direction and its axis of rotation is entirely in the z direction. Therefore the manipulator in that configuration can estimate forces in all three directions of the world frame although the x and z estimates are coupled because they come from the single estimate of external torque acting on the first joint.

In the case of the hardware experiment, forces were exerted by hand on the manipulator's end effector while it was maintaining the position described above. Section 4.3.2 provides the actual forces measured by the force/torque sensor and the results of force estimation during this test. In the case of the simulation, described in Section 4.3.1, the actual forces from the hardware experiment were logged and fed into the simulation of the system maintaining the same position.

4.3.1 Stationary Testing - Simulation

Force estimates versus actual force, as measured by the force/torque sensor, are shown in Figure 4.8 for the case of a stationary desired trajectory in the simulation case. The force estimates are can be seen to be very close to the actual force. The force estimation error, generally at or below 5 Newtons (N), can be attributed to two factors. The first is imperfectly learned parameters of both the modeled dynamics and the joint friction. The second factor is more significant – it is the delay in the force estimates due to the filtering of the estimated torque before it is converted to force (described in section 3.1.4). The spikes in the error are due to this delay - they can be seen to occur during large changes in the actual force, when the derivative of the force is close to an impulse. Furthermore, the filter only keeps frequencies at or below 20 Hz so the high frequencies in the force signal due to the fast change are filtered out. When the actual force does not change as quickly, as in the time period around 10 seconds in the x and z directions, the force error is very small.

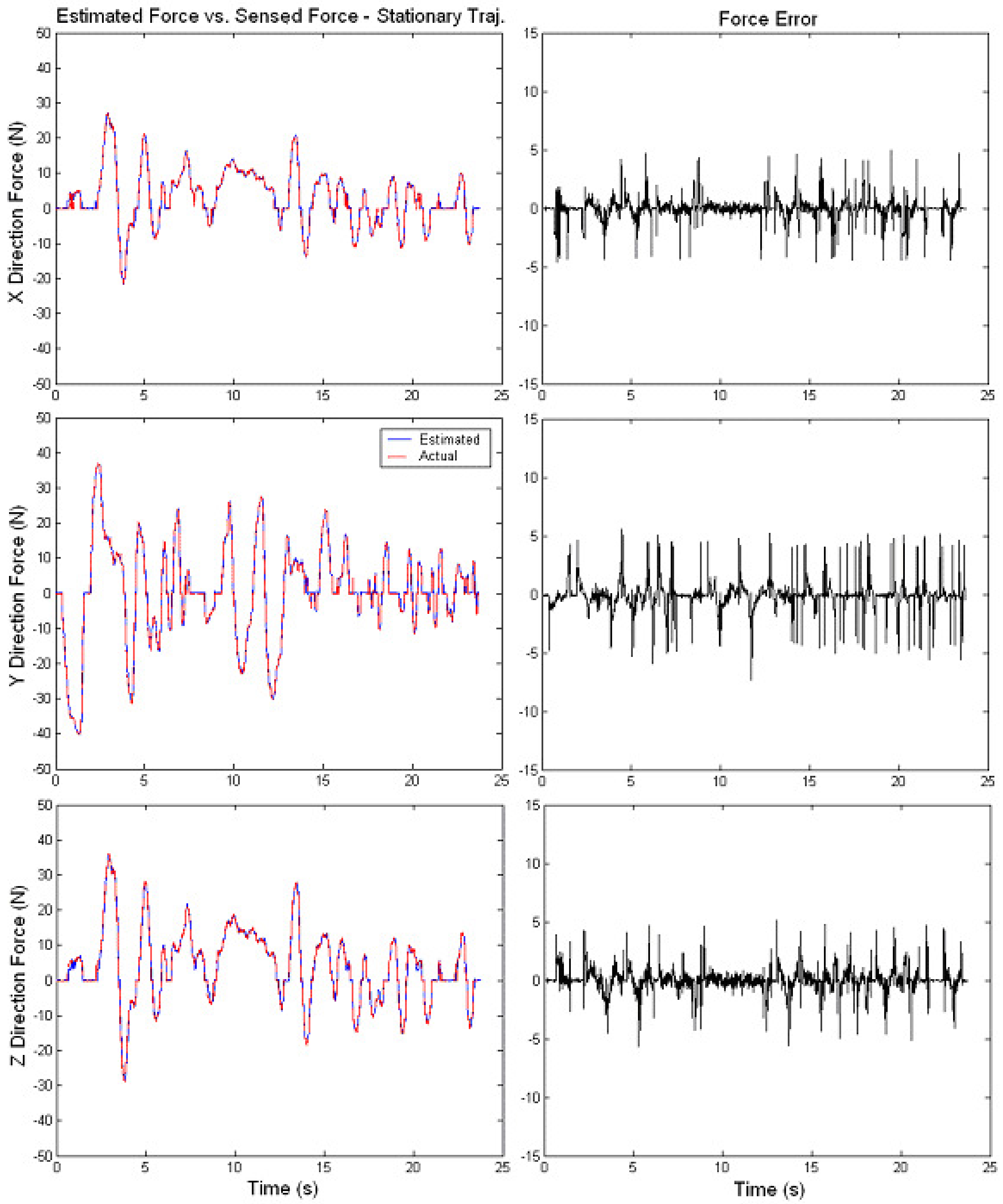


Figure 4.8 Estimated force versus sensed force and associated error in the case of a stationary testing trajectory for the simulation case.

4.3.2 Stationary Testing - Hardware Experiment

Force estimates versus actual force are shown in Figure 4.9 for the case of a stationary desired trajectory in the hardware case. The force estimation error is generally at or below 7 N. The two sources of error mentioned in Section 4.3.1 for the simulation case also apply here with the added comment that the larger amount of error versus the simulation case is due to the manipulator being stationary and hence being in the region of velocity most affected by stiction. Fortunately in the stationary case, when the desired velocity is exactly zero, the commanded torque due to the high PD gains discussed in section 3.1.3 is able, for the most part, to overcome the offsets in estimated torque caused by stiction.

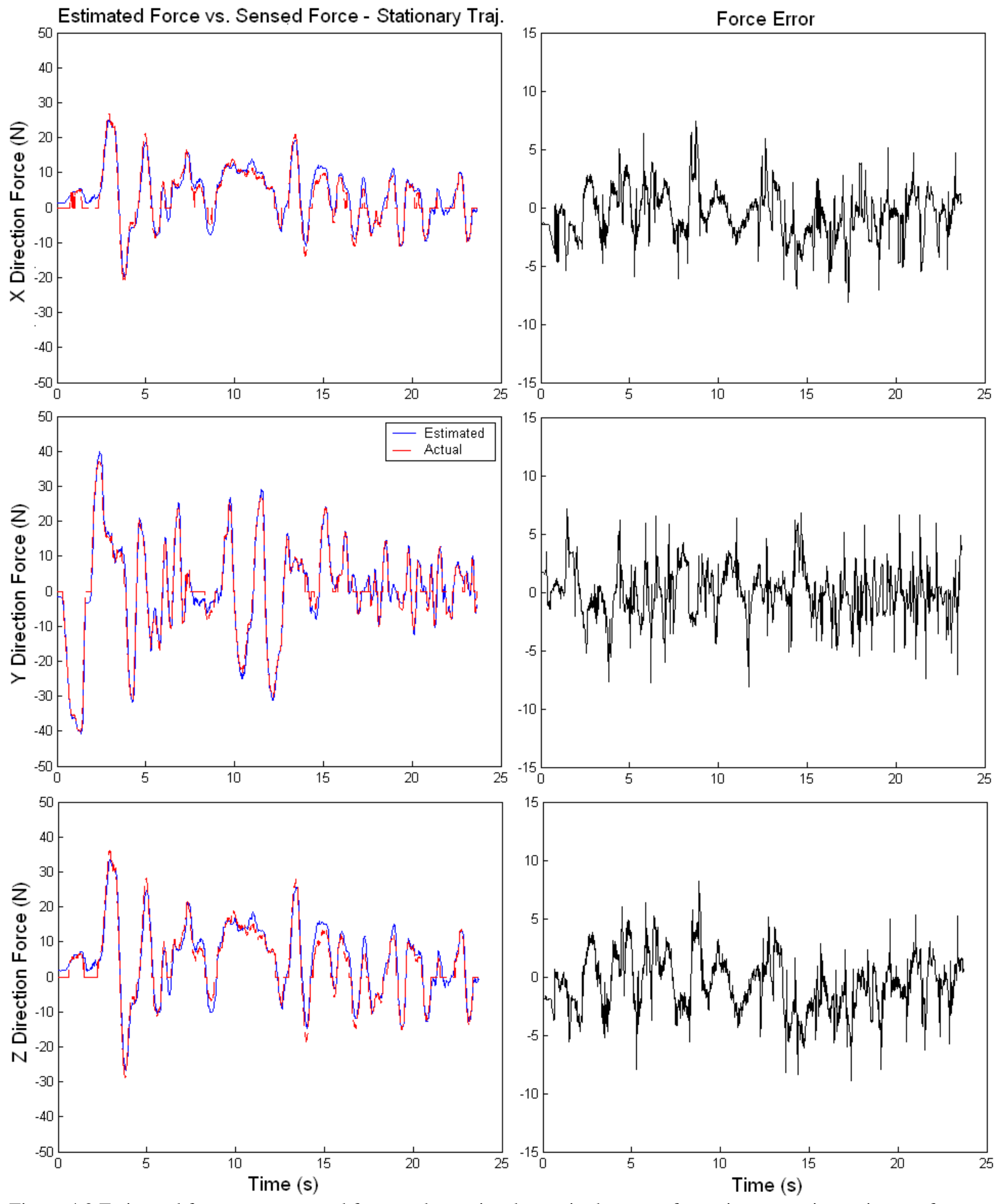


Figure 4.9 Estimated force versus sensed force and associated error in the case of a stationary testing trajectory for the hardware case.

4.4 Force Estimation Results for Dynamic Testing

Force estimation was also performed during a non-stationary testing trajectory, referred to here as a dynamic testing trajectory both in simulation and with the hardware. The trajectory used during this test was the same in both cases, with the desired position given as

$$q(t) = \begin{bmatrix} -0.4\cos(2\pi(0.05)t) + 0.4 \\ -0.4\cos(2\pi(0.05)t) + 0.4 \end{bmatrix} + q_0 \qquad (4\text{-}9)$$

where the dependence of the joint position on time has been made explicit. It is shown in Figure 4.10. The trajectory is sinusoidal of frequency 0.05 Hz for both joints with initial position $\mathbf{q}_0^T = [-0.6 \quad -\pi]$.

In the case of the hardware experiment, forces were exerted by hand on the manipulator's end effector while it was tracking the trajectory described above. Section 4.4.2 provides the actual forces measured by the force/torque sensor and the results of force estimation during this test. In the case of the simulation, described in Section 4.4.1, the actual forces from a hardware experiment were logged and fed into the simulation of the system tracking the same trajectory.

4.4.1 Dynamic Testing - Simulation

Force estimates versus actual force are shown in Figure 4.10 for the case of the dynamic desired trajectory associated with (4-10) in the simulation case. The force estimation error is generally at or below 10 N due to spikes from the filtering delay, slightly larger than in the stationary case for the simulation, shown in Figure 4.8. These spikes in the error are larger amplitude in the stationary testing trajectory but that may be attributed more to the fact that the actual force has larger amplitude and changes faster in the dynamic test. Besides these spikes the force estimates are very close to actual force and the estimated force during zero actual force is also zero as evidenced by the force error at those times. Note also that there is also no velocity dependence to the force error, something that will be seen in the hardware case. This means that the velocity-dependent friction was well learned by the adaptive networks, as was shown in Figure 4.4.

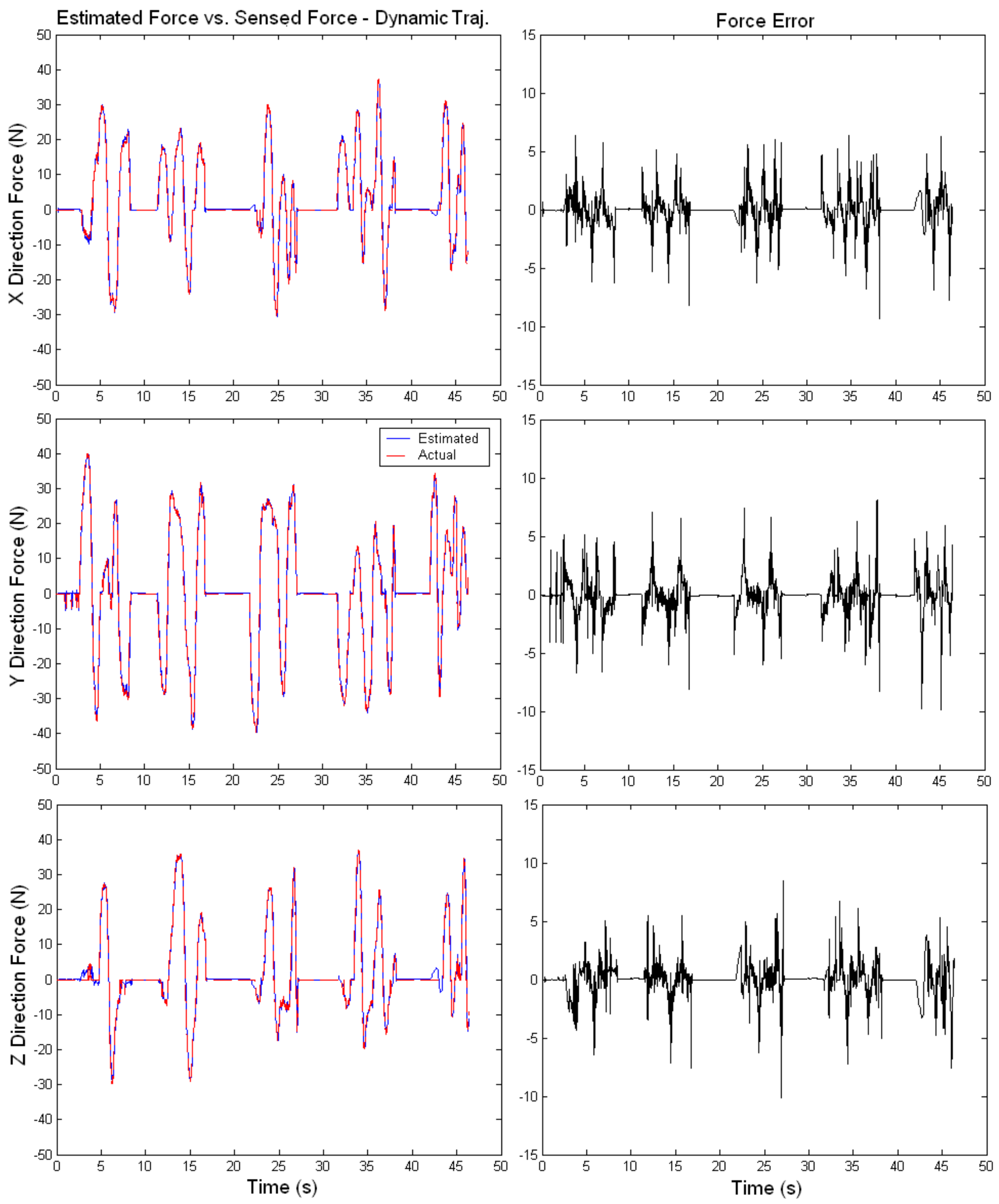


Figure 4.10 Estimated force versus sensed force and associated error in the case of a dynamic testing trajectory for the simulation case.

4.4.2 Dynamic Testing - Hardware Experiment

In the case of hardware experiments, force estimation during a dynamic trajectory is more challenging than during the stationary case. This is because dynamic trajectories test the accuracy of the viscous friction model obtained during training, which in the case of harmonically driven manipulators often represents a significant proportion of the overall joint torque. In simulation this did not present a problem because the actual friction of each joint was known, so a training trajectory was chosen such that both the modeled and unmodeled parameters converged. In hardware, each joint's mapping from velocity to friction torque is unknown so it was assumed that if the modeled dynamics' parameters converged to the neighborhood of their theoretical values, then the friction was well represented by the estimate provided by the adaptive networks. In other words, if the trajectory was clearly persistently exciting for the adaptation of the modeled parameters it was assumed to be persistently exciting for the parameters of the unmodeled dynamics.

Force estimates versus actual force are shown in Figure 4.13 for the case of the dynamic desired trajectory associated with (4-9) in the hardware case. The force estimation error is generally at or below 10 N due in part to spikes from the filtering delay. After observing the figure, two problems are evident that were not present in the simulation case. The first is that the force error seems to be velocity dependent, as evidenced by its 20 s period, the same as the trajectory being tracked. One possible explanation for this dependency is that the constants associated with the motors and motor drivers are not perfectly known. On the motor end are the torque constant and inertia due to the motor, which becomes significant through the effect of gearing. On the motor driver end, the current monitoring is accomplished through reading a proportional voltage with the DAQ board. Variations of 10-20% in this proportion or the torque constant are not unlikely. Another explanation of this problem is that it is the result of the adaptive networks not learning a fully accurate model of the friction. In that case the solution would be to seek a more persistently exciting trajectory or train longer with the current trajectory.

The second problem is that the force estimates are not zero when the actual force is zero. This is a more serious problem because the force estimates will be fed back to a compliance controller as described in Chapter 5. The problem can be alleviated by imposing a threshold upon the estimated external torque before it is converted into force (see section 3.1.2). The thresholding is imposed on the filtered torque, after step two in the procedure discussed in that section.

To choose the threshold the evolution of the estimated external torque during training must first be examined. Graph A of Figure 4.11 shows the progress of the filtered estimated external torque of the first joint over the first 50 seconds of training. Ideally, when the actual external torque is zero, the adaptation should drive this estimate to zero. However, since the friction is not learned perfectly, there is about 1 Nm unlearned at velocities with absolute value above 0.002 rad/s and 2 Nm unlearned below 0.001 rad/s, as graph B of Figure 4.11 shows. This graph illustrates the error in the estimated external torque (simply the estimated external torque since the actual is zero) between 300 seconds and 500 seconds of the training, when the modeled parameters and networks' coefficients have converged close to their final values. For higher velocities the error is likely due to position-dependent friction that cannot be captured by adaptive networks that learn the mapping from velocity to friction torque. The higher error near zero velocity is due to stiction that was not captured by the adaptive networks. Such networks tend to have difficulty learning discontinuities even when the nodes are densely spaced. A friction model incorporating stiction in harmonic drives has been suggested in (Ghandi et al., 2002) based on the dynamic model of friction proposed in (Canudas de Wit et al., 1995), but the model contains a number of parameters whose values are hard to determine in practice. An adaptive version of the dynamic friction model introduced in (Canudas de Wit et al., 1995) has been presented in (Misovec and Annaswamy, 1999).

The thresholding function is shown in graph A of Figure 4.12. It simply zeroes any torque with absolute value below the chosen threshold torque, $\boldsymbol{\tau}_{\mathbf{THRESHOLD}}$, and decreases the absolute value of any torque above that by the threshold torque. The threshold torque was chosen to be velocity dependent, as shown in graph B of Figure 4.12, due to the velocity dependent error in graph B of Figure 4.11. $\boldsymbol{\tau}_{\mathbf{THRESHOLD}}$ was chosen to be 2 Nm for velocities with absolute value at or below 0.001 rad/s and 1 Nm for actual velocity with absolute value at or above 0.002 rad/s with a linear transition in between. Joint 2's threshold is the same because its estimated external torque is driven down to a similar velocity-dependent error.

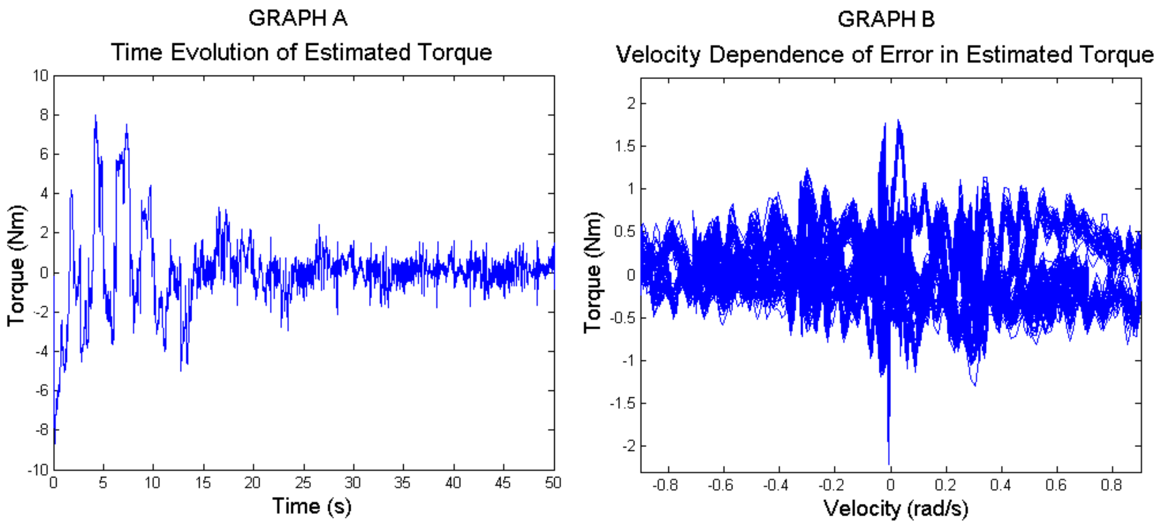


Figure 4.11 Time evolution of the estimated torque and velocity dependence of the error in the estimated torque between (between 300 s and 500 s of training time) for joint 1.

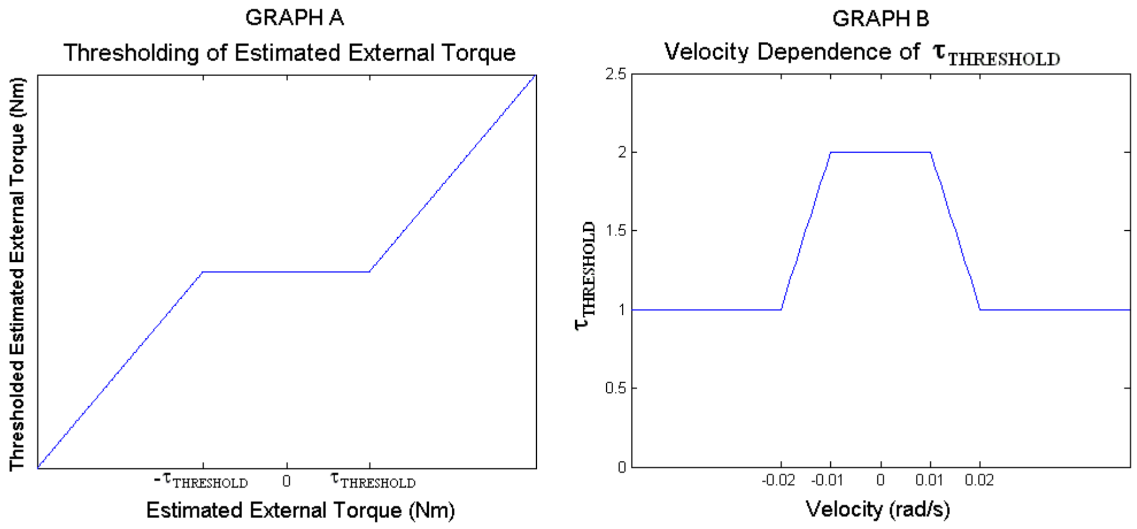


Figure 4.12 Velocity dependent thresholding of external estimated torque for hardware experiments.

Figure 4.14 depicts the force estimates versus actual force after the estimated external torque is thresholded using the velocity dependent technique described above. There is a definite improvement in the force estimates when the actual force is zero as can be seen in x and z direction estimates around 16-17 s and the y direction estimates around 15 s compared to the same time periods in Figure 4.13.

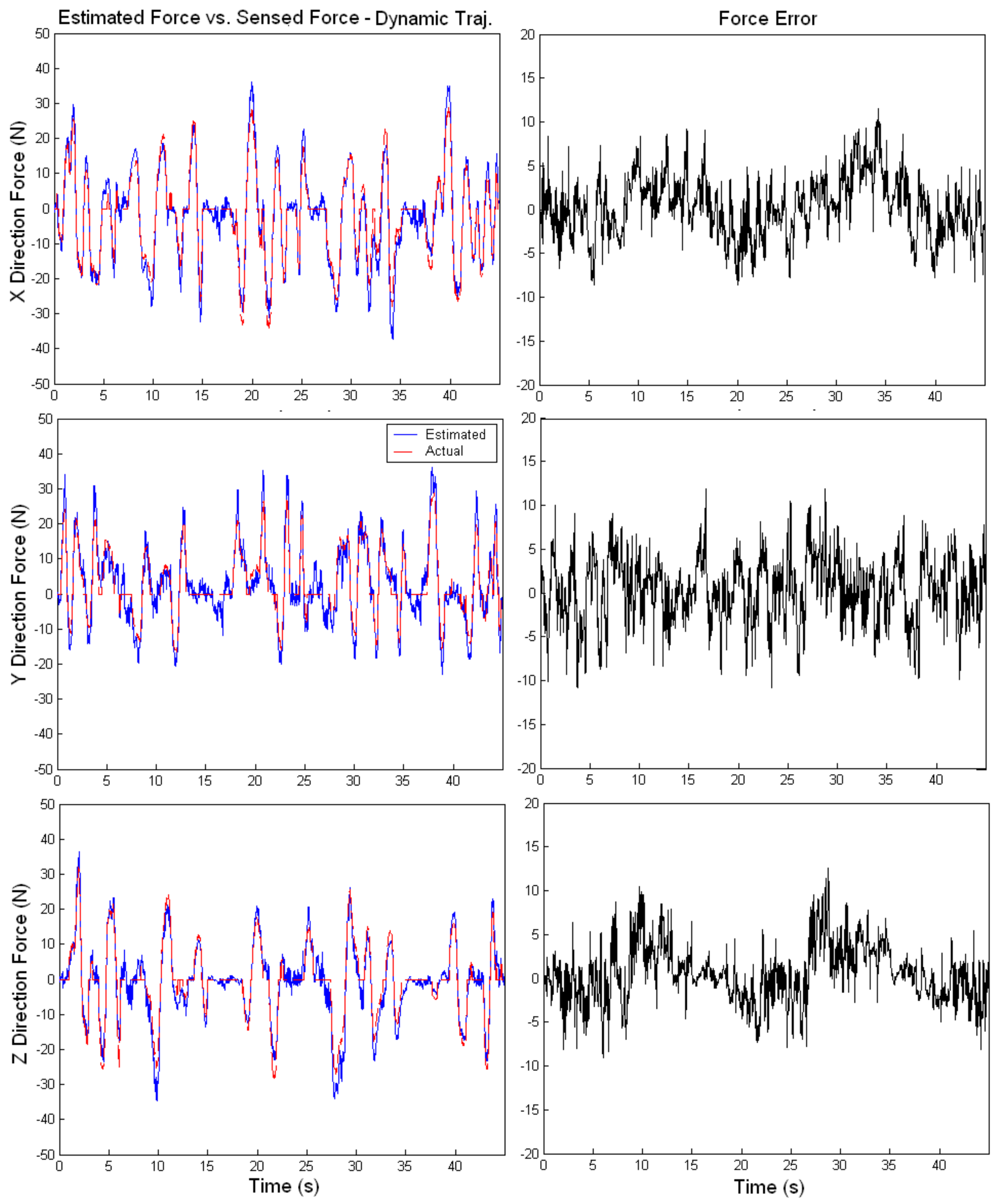


Figure 4.13 Estimated force versus sensed force and associated error in the case of a dynamic testing trajectory for the hardware case.

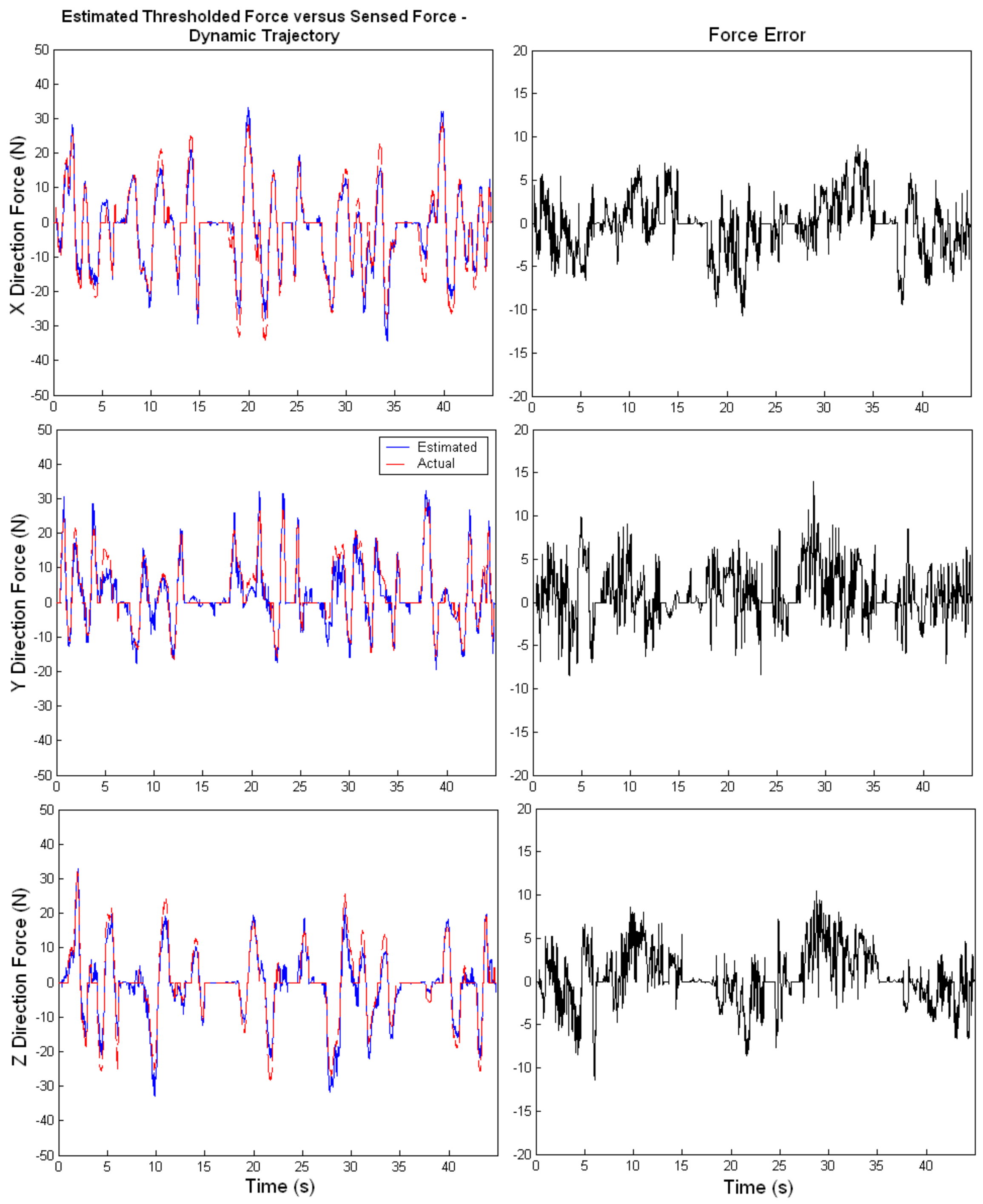


Figure 4.14 Thresholded estimated force versus sensed force and associated error in the case of a dynamic testing trajectory for the hardware case.

4.5 Summary

This chapter detailed the training trajectories used in both simulation and hardware experiments. The evolution of the modeled dynamics' parameters as well as the coefficients of the adaptive networks learning the unmodeled friction were then presented for both the simulation and hardware. Force estimation using the adapted parameters was then presented for a stationary testing trajectory in the simulation case and sources of error in the estimates were discussed. Briefly, the increased error in the experiment stemmed from delays caused by the filtering of the motor current. Force estimation for the stationary trajectory in the hardware case had similar error as it was able to overcome any unmodeled effects of stiction with high PD gains.

In the case of a dynamic testing trajectory, the simulation exhibited slightly larger amounts of error can be attributed to the larger amplitude and faster changing actual forces that happened to be used in the dynamic test. For the hardware experiments the dynamic testing trajectory revealed velocity dependence in the force estimation error that was attributed to the adaptive networks not having perfectly learned the friction. As a consequence, the problem of having non-zero estimated force during periods of zero actual force was identified. The problem was remedied by using a velocity-dependent threshold of the estimated external torque that is converted to estimated external force. This thresholding greatly reduced the force estimation error during zero actual force.

Chapter 5: Compliance Control Experiment

To illustrate the feasibility of using force estimates for force-based control, a compliance controller was implemented on the hardware test bed. First, the controller will be described then an experiment demonstrating the traditional approach using a force sensor will be presented. Finally, the same controller will be used with force estimation input instead of force sensing.

5.1 Compliance Controller

The compliance controller chosen was the “position-based impedance control” scheme used in (Guion, 2003). The controller, shown in Figure 5.1, is dual-loop; an inner loop controls joint position and an outer loop modifies the nominal desired Cartesian position to form the modified desired Cartesian position. The controller uses the external force it accepts to deflect the manipulator's trajectory in Cartesian space. The modified desired Cartesian position is converted to modified desired joint position through inverse kinematics for use by the inner loop. The adaptive friction-learning controller introduced in Chapter 2 is used as the inner position control loop. The compliance controller is the outer loop around the position control loop. The desired behavior between external force and the deflection of the trajectory from the nominal desired trajectory is represented by the following equation

$$F_{ext} = M_s \ddot{\tilde{x}} + C_s \dot{\tilde{x}} + K_s \tilde{x} \tag{5-1}$$

where $\tilde{\mathbf{x}} = \mathbf{x_d} - \mathbf{x_n}$, $\mathbf{x_n}$ is the nominal desired Cartesian position, and $\mathbf{x_d}$ is the modified desired Cartesian position due to the enforcement of the desired impedance behavior specified by the mass, spring, and damping matrices $\mathbf{M_s}$, $\mathbf{C_s}$, and $\mathbf{K_s}$ respectively. The difference between (5-1) and the compliance control law used in (Guion, 2003) is that the left side of (5-1) is $\mathbf{f_{ext}}$ rather than $\mathbf{K_f f_{ext}}$ which simply means that the three matrices on the right hand side of (5-1) have been multiplied by $\mathbf{K_f^{-1}}$. The modified trajectory is formed by rearranging (5-1) as follows

$$\ddot{x}_d = M_s^{-1}\left(F_{ext} - C_s\dot{\tilde{x}} - K_s\tilde{x}\right) + \ddot{x}_n \tag{5-2}$$

with the modified desired Cartesian velocity and position formed by integrating (5-2). These integrals were performed numerically using the technique given in Appendix A.1.

Because the modified desired trajectory issued by the compliance control law is formed in Cartesian space rather than joint space, the following numerical computation of the inverse kinematics was performed to form the input for the joint space control law (2-48) to use.

$$q_d = \int_0^t \dot{q}_d dt \tag{5-3}$$

$$\dot{q}_d = \mathrm{inv}(J)\dot{x}_d \tag{5-4}$$

$$\ddot{q}_d = \frac{d\dot{q}_d}{dt} \tag{5-5}$$

The integral in (5-3) and derivative in (5-5) were computed numerically also using Appendix A.1.

In training mode the modified desired trajectory due to the compliance control was not used – instead the desired trajectory was specified directly in joint space. The desired trajectory may also be specified in Cartesian space and converted into joint space. During estimation mode, the nominal desired trajectory was still specified in joint space although it was converted into Cartesian space for the compliance controller as follows

$$x_n = k(q_n) \tag{5-6}$$

$$\dot{x}_n = J\dot{q}_n \tag{5-7}$$

$$\ddot{x}_n = \frac{d\dot{x}_n}{dt} \tag{5-8}$$

where **k** is the manipulator's forward kinematics – the transformation from joint position and orientation to end effector Cartesian position and orientation.

A block diagram of the compliance and position controllers is shown in Fig. 5.1. Note that the learning rules shown in the figure are disabled when the force estimation based compliance controller is active. Also note that the control law (2-48) is unaware of the modification to its desired trajectory by the compliance controller block just as the compliance controller block is unaware of the use of force estimation instead of actual force sensing. This last point can be made explicit by substituting (2-54) instead of $\mathbf{f_{ext}}$ in (5-1) and (5-2).

In both the sensor based and force estimation based compliance control experiments, the three matrices in (5-1) that specify the desired impedance were set to

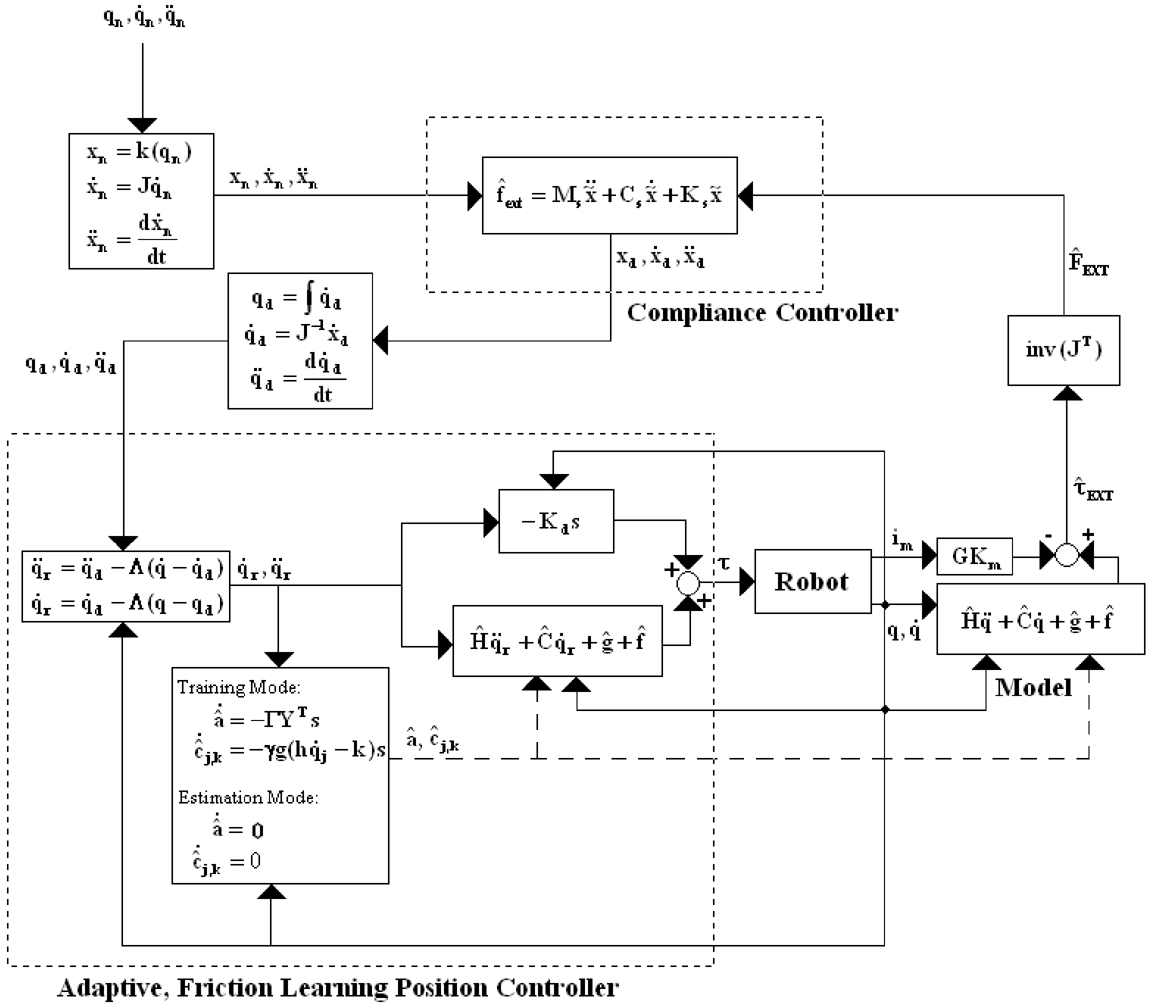
$$\begin{aligned} M_s &= m_s I_3 \\ C_s &= c_s I_3 \\ K_s &= k_s I_3. \end{aligned} \tag{5-9}$$

where $\mathbf{m_s = 100}$, $\mathbf{c_s = 2000}$, $\mathbf{k_s = 200}$ are scalar values. The damping ratio along each Cartesian direction for this choice of admittance is

$$\varsigma = \frac{c_s}{2\sqrt{k_s m_s}} \approx 7 \tag{5-10}$$

This high damping ratio will highlight problems encountering at low velocity. Typically overdamped behavior ($\boldsymbol{\varsigma > 1}$) is desired in a compliance controller because of well-known issues involving instability between a manipulator and the environment. The phenomenon, termed "contact instability" can occur when a compliance-controlled manipulator comes in contact with a very stiff environment, a wall for example. If the nominal desired position is past the border of the wall the manipulator will hit the wall and the contact force will deflect the trajectory in an attempt to enforce the desired stiffness relationship between the nominal and modified desired position. If the manipulator's damping ratio is not high enough the manipulator will move out to the modified desired position too quickly.

The problem is that this deflected position is likely to occur before the border of the wall, in a place where there is zero contact force, so the trajectory reverts back to the nominal and the manipulator moves back into the wall. A cycle of "chattering" begins in which the manipulator moves in and out of the wall, alternately making and breaking contact in its attempt to enforce the stiffness term of its desired admittance. A high damping ratio ensures that the damping term of the desired admittance is large enough relative to the mass and stiffness terms. A large damping term ensures the manipulator moves slowly as a result of contact. Therefore when it hits the wall, provided the wall has finite stiffness, the manipulator has a chance to settle out into a location that maintains contact and therefore enforces its steady state desired stiffness.



$q_n, \dot{q}_n, \ddot{q}_n$
$x_n = k(q_n)$
$\dot{x}_n = J\dot{q}_n$
$\ddot{x}_n = \frac{d\dot{x}_n}{dt}$
$x_n, \dot{x}_n, \ddot{x}_n$
$\hat{f}_{ext} = M_s\ddot{\tilde{x}} + C_s\dot{\tilde{x}} + K_s\tilde{x}$
$x_d, \dot{x}_d, \ddot{x}_d$
Compliance Controller
$\hat{F}_{EXT}$
$inv(J^T)$
$q_d = \int \dot{q}_d$
$\dot{q}_d = J^{-1}\dot{x}_d$
$\ddot{q}_d = \frac{d\dot{q}_d}{dt}$
$q_d, \dot{q}_d, \ddot{q}_d$
$\hat{\tau}_{EXT}$
$-K_d s$
$\ddot{q}_r = \ddot{q}_d - \Lambda(\dot{q} - \dot{q}_d)$
$\dot{q}_r = \dot{q}_d - \Lambda(q - q_d)$
$\dot{q}_r, \ddot{q}_r$
τ
Robot
i_m
GK_m
$q, \dot{q}$
$\hat{H}\ddot{q}_r + \hat{C}\dot{q}_r + \hat{g} + \hat{f}$
$\hat{H}\ddot{q} + \hat{C}\dot{q} + \hat{g} + \hat{f}$
Model
Training Mode:
$\dot{\hat{a}} = -\Gamma Y^T s$
$\dot{\hat{c}}_{j,k} = -\gamma g(h\dot{q}_j - k)s$
$\hat{a}, \hat{c}_{j,k}$
Estimation Mode:
$\dot{\hat{a}} = 0$
$\dot{\hat{c}}_{j,k} = 0$
Adaptive, Friction Learning Position Controller

Fig. 5.1 Block diagram of adaptive, friction learning position controller. Compliance control is based on the force estimates generated using the controller's adapted model parameters. Nominal desired trajectory, inputted at the top left, is specified in joint space, then converted to Cartesian space for the compliance controller to use.

5.2 Force Sensor Based Compliance Control

In this section compliance control will be demonstrated for the case in which force sensed by the force/torque sensor described in section 3.2.3 is used to generate the desired compliant behavior specified in (5-1). In the force-based compliance control experiment the manipulator was maintaining the same nominal desired joint position as in the stationary tests of Chapter 4, $q^T = [0 \quad -\pi]$. Forces were exerted by hand on the manipulator's end effector to demonstrate the deflection from the nominal desired joint position due to the enforcement of the desired impedance. Figure 5.2 shows the force sensed during the experiment. Note that the forces shown have been transformed by equation (3-6), limiting the full three degree of freedom force provided by the sensor to the two configuration-dependent degrees of freedom of the manipulator. Figure 5.3 shows the deflection of the nominal trajectory due to the forces shown in Figure 5.2. Due to the high damping ratio the modified position of both joints gradually returns to the nominal position when there is no contact force, as is clearly the case between 30 and 50 seconds and after 80 seconds.

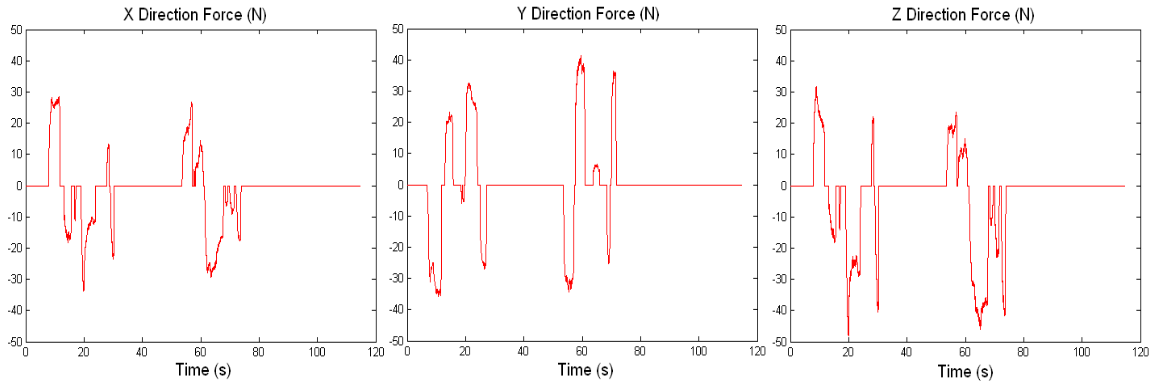


Figure 5.2 Sensed force during force sensor based compliance control.

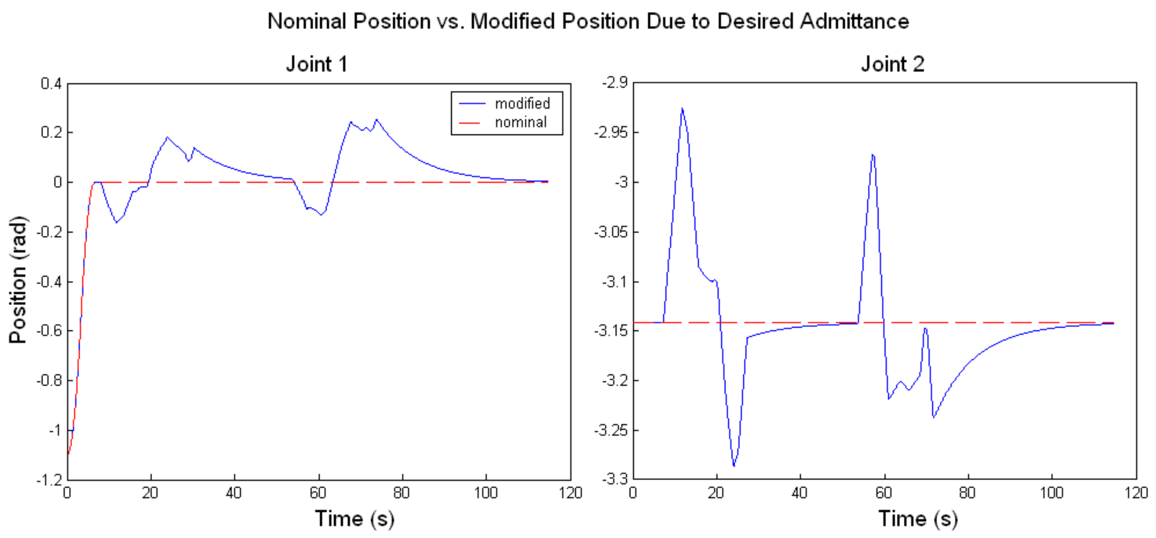


Figure 5.3 Nominal joint position versus modified joint position during force sensor based compliance control.

5.3 Force Estimation Based Compliance Control

Having demonstrated compliance control using the force/torque sensor, the case in which the force estimates are used instead is now presented. The same Figure 5.4 shows the force estimates versus actual forces during this test. The noise in the estimates is immediately evident around zero actual force. The reason for this error is that the desired velocity that is modified by the compliance controller is commanding the joints to repeatedly pass through the friction discontinuity at zero velocity. The modified position for both joints is shown in Figure 5.5 and a close-up of the modified velocity of joint 1 during the 130 to 200 second time period of the compliance testing is shown in Figure 5.6. The modified velocity during that time oscillates about a point close to zero. The noisiness in Figure 5.4 seen during that time period may stem from the fact that some of the friction torque is still not being captured by the adaptive networks. To combat this problem an additional filter was applied to each joint's estimated external torque during low velocity, defined to be an absolute value of 0.01 rad/s or less. The filter is given by

$$\bar{x}_k = \frac{1}{n}\sum_{i=k-n+1}^{k} x_i \tag{5-11}$$

where $\bar{\mathbf{x}}_{\mathbf{k}}$ is the filter's output at time stamp **k**, seen to be the average of the previous **n** values of **x**, in this case representing the filtered, thresholded estimate of the external torque, as discussed in 4.4.2. One second's worth of previous samples were chosen to be averaged - in the case of 3 kHz sampling this resulted in saving three thousand samples, though this number can be reduced if the estimated torque is saved at a lower frequency for use in the low bandwidth compliance controller. Note that this filtered force was fed back to the compliance controller rather than the forces shown in Figure 5.4.

The price for the reduced noisiness of the filtered estimates is paid in the situation where the actual force is changing while the manipulator is still moving very slowly or standing still. Eventually the moving average filter will begin to output the changes in the force and the manipulator will move in response, leaving the low velocity regime and turning the moving average filter off. The delay between the filtered output, which changes slowly because of the moving average filter's low bandwidth, and a faster changing actual force is what causes the sharp spikes in force estimation error seen in Figure 5.5.

The case of this moving average filter being applied to non-zero actual force is shown around 120 seconds with reduced noise in the error for all three directions' estimates. In Figure 5.5 both joints' velocity can be seen to be near zero at that time with the actual forces in all three directions non-zero and constant or slowly changing. Reducing the noise reveals poorer estimates in low velocity. They are made worse because the thresholding described in section 4.4.2 that was used to improve estimates made during zero actual force devalues the estimated external torque by 2 Nm in low velocity. So in solving the problem of non-zero estimates during zero actual force it exacerbates the other problem of estimating non-zero actual forces in low velocity.

In addition, though the low velocity filtering goes far in reducing the noise in the estimates the noise that remains is to blame for the slower convergence to the nominal position of the second joint in Figure 5.6, evident between 150 and 200 seconds. Though it was not implemented, a remedy would be to add additional thresholding, this time on the estimated force after it has been transformed from estimated torque. The threshold would be different than the one shown in Figure 4.12 – it would simply zero all force estimates below a certain absolute value and maintain the values of estimates above that point. The thresholding function is shown in Figure 5.8, with the proposed threshold point set at 2 N.

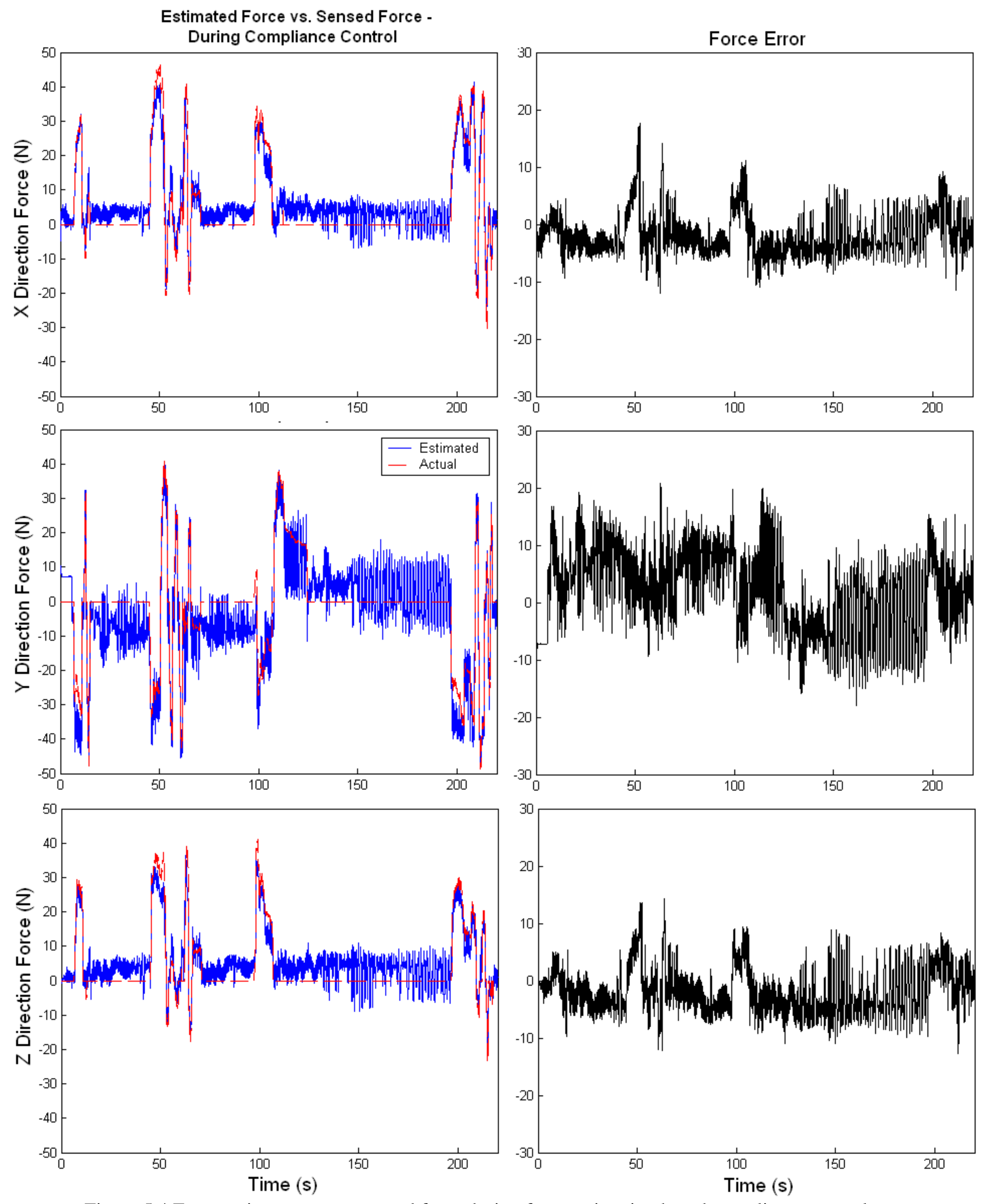


Figure 5.4 Force estimates versus sensed force during force estimation based compliance control.

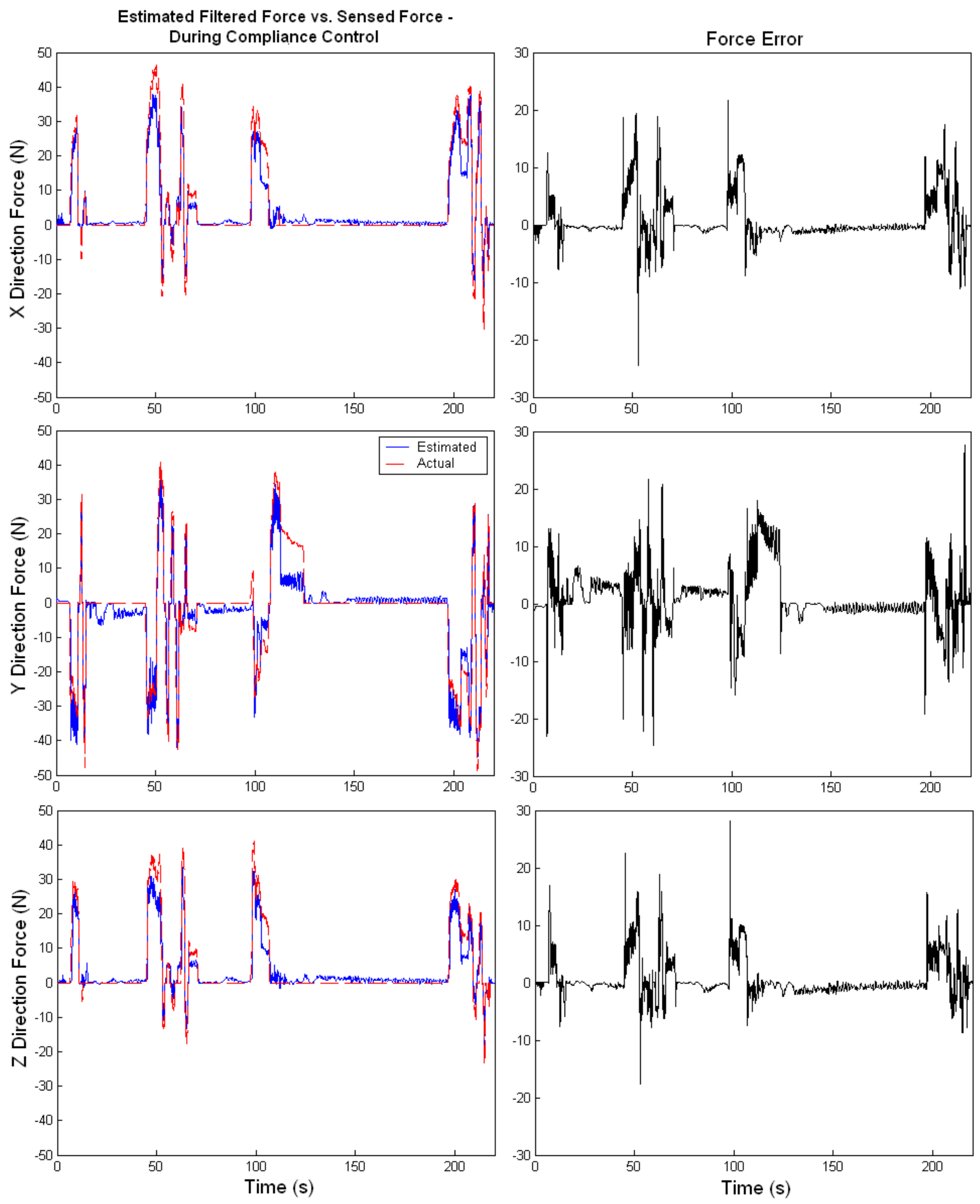


Figure 5.5 Low velocity filtered and thresholded force estimates versus sensed force during force estimation based compliance control.

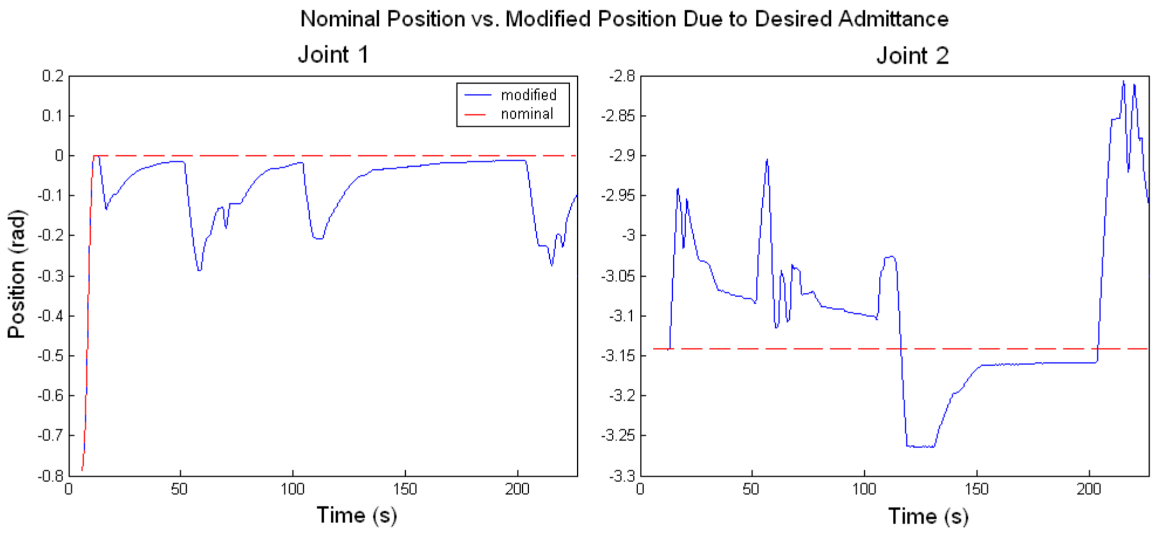


Figure 5.6 Nominal desired position vs. modified desired position during force estimation based compliance control.

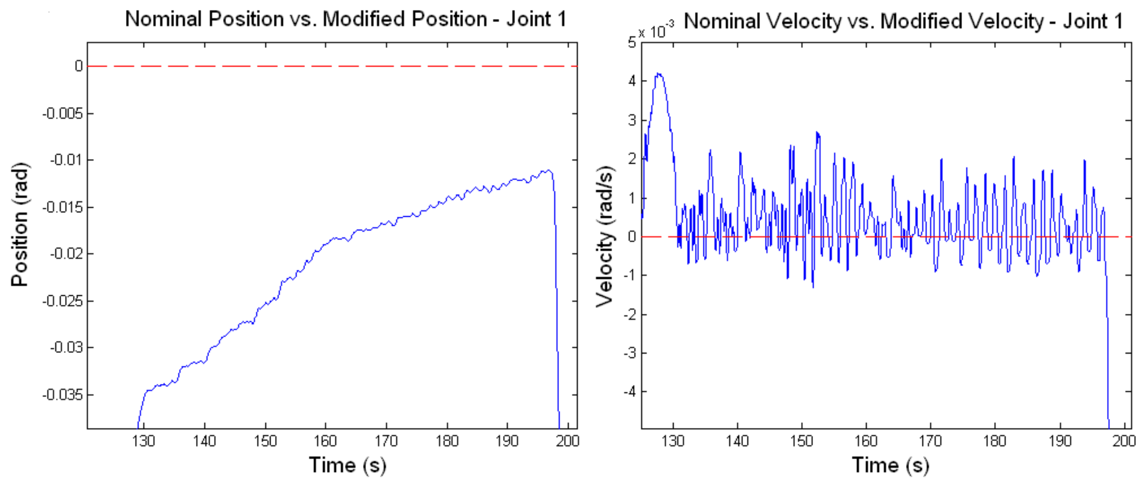


Figure 5.7 Close up of 130 sec. to 200 sec. time period during which the low velocity estimation problem occurs.

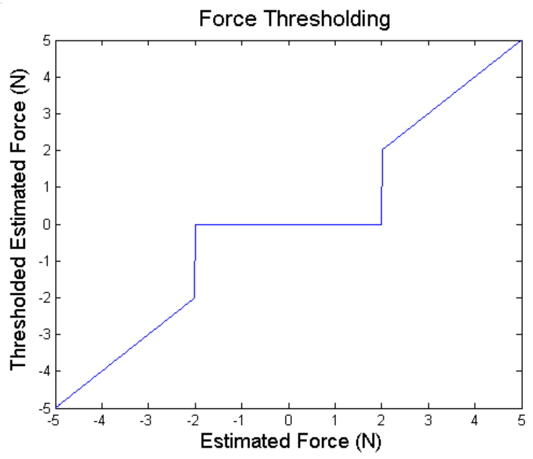


Figure 5.8 Suggested estimated force thresholding function.

Chapter 6: Conclusion and Future Work

6.1 Conclusion

This thesis demonstrated the use of an adaptive, friction-learning controller in estimating external forces exerted upon a harmonically driven manipulator. Force estimation, actual force and estimation error during both stationary and dynamic experiments were presented both in simulation and hardware experiments. Force estimation errors during the dynamic hardware test, presented in Section 4.4.2, appeared to have the same period as the trajectory that was being tracked. This may have been caused by imperfectly known constants associated with the motors and motor drivers or by the choice of training trajectory that was not sufficiently persistently exciting. Thresholding of the estimated external joint torque was used to reduce the error of the force estimates during zero actual force. A velocity-dependent threshold was used to combat the increased estimation error near zero velocity associated with stiction that was not captured by the adaptive neural networks. Outside of the near-zero velocity region the threshold attempted to eliminate unmodeled position dependence in the joint friction. The price for improved estimates of zero actual force was paid in worse estimates of non-zero actual force.

A first attempt at compliance control based on force estimation in a harmonically driven manipulator was then presented. Further problems were identified in the near zero velocity region. Filtering the estimated thresholded external joint torque in that region led to improved estimates of zero actual force at the expense of significant delay and inaccuracy in the estimates of non-zero actual force.

The results presented lead to the conclusion that further work needs to be done before force estimation is accurate enough for use in feedback controllers. The problem that needs to be addressed most urgently is the inaccuracy of estimates in the steady state. A friction model that captures more of the effects of stiction will lead to improved steady state estimation, greatly reducing or even eliminating the need for thresholding.

6.2 Future Work

As noted in the conclusion, the next step to be taken is to in better incorporating stiction into the friction model. This would greatly improve the feasibility of using estimates with compliance control. The most promising avenue is to investigate the use of a dynamic friction model, mentioned in Section 4.4.2. Such a model may only be necessary in the low velocity regime as noted in the references mentioned in that section.

Another important extension would be to model the position dependence that was identified and dealt with using thresholding. Expanding the adaptive neural network to two dimensions, making it a mapping from both position and velocity to joint torque, is a logical direction. A practical difficulty arises in finding a persistently exciting training trajectory for such a two-dimensional friction model due the need to have every position encounter the full range of input velocities. The training period necessary to accomplish this may be significantly longer than the training period used in this work depending on the position resolution chosen.

Though it was noted that the force estimation technique presented is able to retrain, that is reenter the training mode after estimation mode, this capability was not explored in the research presented. Further work examining the effect of retraining at runtime because of end effector loading and temperature variation is needed. Loading affects both the modeled parameters and the unmodeled friction while temperature has been shown to greatly affect friction. An understanding of how frequently retraining would be necessary due to these changes would be invaluable.

Appendix A

A.1 Numerical Differentiation and Integration

Numerical differentiation and integration were performed numerous times in the code given in Appendix B. The iterative version of numerical differentiation used, called Euler differentiation, is defined as

$$\dot{x}[n] = \frac{(x[n] - x[n-1])}{\Delta t} \tag{A-1}$$

where Δ**t** is the period of the system. The iterative version of numerical integration used, called the trapezoidal rule, is defined as

$$x[n] = x[n-1] + .5\Delta t(\dot{x}[n] + \dot{x}[n-1]). \tag{A-2}$$

A.2 Encoder Signal Conversion

The MORPHbots module's encoders provided Channel A/Channel B digital output signals that were fed into an LS 7184 microchip that converts the signals to the $\overline{\text{Clock}}$ / Up/$\overline{\text{Down}}$ signaling convention. The use of the LS7184 chip is necessary because the counters on the DAQ board accept the $\overline{\text{Clock}}$ / Up/$\overline{\text{Down}}$ signaling convention rather than the Channel A/Channel B convention of the encoder electronics. The timing diagram of the circuit is shown in Figure A.1.

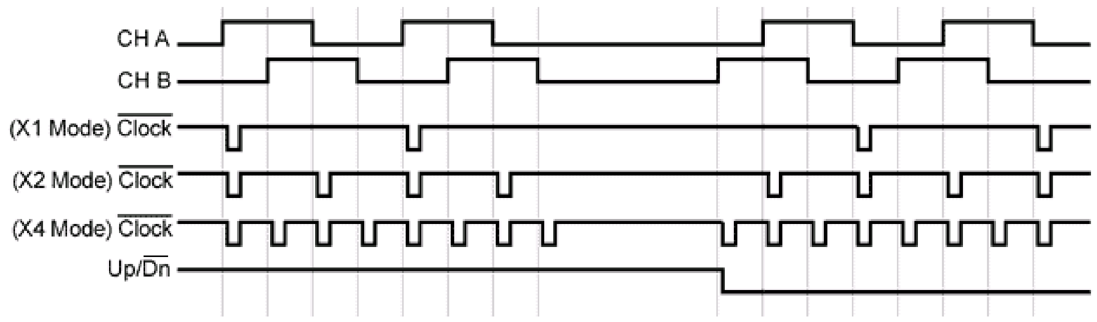

CH A
CH B
(X1 Mode) $\overline{\text{Clock}}$
(X2 Mode) $\overline{\text{Clock}}$
(X4 Mode) $\overline{\text{Clock}}$
Up/$\overline{\text{Dn}}$

Figure A.1: LS7184 Timing Diagram

As seen in Figure A.1, transitions on the Channel A or Channel B input lines are converted to inverted pulses on the $\overline{\text{Clock}}$ output lines. The chip can operate in X1, X2 or X4 mode depending on which edges of the Channel A/Channel B signals are counted as transitions. In X1 mode an inverted pulse is only generated with Channel A transitions from low to high. In X2 mode an inverted pulse is also generated when Channel A transitions from high to low. In X4 mode the transitions from X1 and X2 mode are counted as well as the transitions low to high and high to low of Channel B. The X4 mode was used because it allows for maximum resolution. When Channel A leads Channel B the Up/$\overline{\text{Down}}$ signal is high while it goes low when the direction of motion changes and Channel B leads Channel A. The two 24-bit hardware counters on the NIDAQ board increment their counts when $\overline{\text{Clock}}$ transitions from high to low and the Up/$\overline{\text{Down}}$ signal is high and decrement their counts when $\overline{\text{Clock}}$ transitions from high to low and the Up/$\overline{\text{Down}}$ signal is low.

A.3 Newton-Euler Dynamics Algorithm

The Newton-Euler recursive dynamics algorithm used to compute the dynamics presented in Appendix A.4 is detailed here. It is based on the propagation of link velocity and acceleration and the use of Newton's force equation and Euler's moment equation and the kinematic relationship between joint frames. The force and moment equations are given for the center of mass of link i ($\mathbf{C_i}$) as

$$ {}^{i}F_{i} = m_{i}\,{}^{i}\dot{v}_{C_i} \tag{A-3}$$

$$ {}^{i}N_{i} = {}^{C_i}I_{i}\,{}^{i}\dot{\omega}_{i} + {}^{i}\omega_{i} \times {}^{C_i}I_{i}\,{}^{i}\omega_{i} \tag{A-4}$$

where $\mathbf{v}$ refers to linear velocity, $\boldsymbol{\omega}$ refers to the angular velocity and **m** refers to a link's mass. The term ${}^{\mathbf{C_i}}\mathbf{I_i}$ refers to the ith link's inertia expressed in the ith link's center of mass frame. Left superscripts describe the frame of reference and the right subscripts describe which joint's term is being referred to. For example the term ${}^{\mathbf{i}}\mathbf{F_j}$ would refer to the force experienced at the center of mass of the jth link expressed in the ith link's reference frame. The algorithm works by relating (A-3) and (A-4) to $\mathbf{f_i}$ and $\mathbf{n_i}$, the forces and moments seen at the manipulator's joints through a force and torque balance. From this joint torque is taken as $\boldsymbol{\tau_i} = {}^{\mathbf{i}}\mathbf{n_i^T}\hat{\mathbf{z}}$, where $\hat{\mathbf{z}}$ refers to the unit vector in the z direction.

TABLE A.1

NEWTON-EULER RECURSIVE DYNAMICS ALGORITHM

Step 1: ***Velocity/acceleration propagation and computation of force/moment at center of mass of each link.***
(i: 0→N-1)

$$ {}^{i+1}\omega_{i+1} = {}^{i+1}_{i}R\,{}^{i}\omega_{i} + \dot{q}_{i+1}\hat{z} \tag{A-5} $$

$$ {}^{i+1}\dot{\omega}_{i+1} = {}^{i+1}_{i}R\,{}^{i}\dot{\omega}_{i} + {}^{i+1}_{i}R\,{}^{i}\omega_{i} \times \dot{q}_{i+1}\hat{z} + \ddot{q}_{i+1}\hat{z} \tag{A-6} $$

$$ {}^{i+1}\dot{v}_{i+1} = {}^{i+1}_{i}R\left({}^{i}\dot{\omega}_{i} \times {}^{i}p_{i+1} + {}^{i}\omega_{i} \times \left({}^{i}\omega_{i} \times {}^{i}p_{i+1}\right) + {}^{i}\dot{v}_{i}\right) \tag{A-7} $$

$$ {}^{i+1}\dot{v}_{C_{i+1}} = {}^{i+1}\dot{\omega}_{i+1} \times {}^{i+1}p_{C_{i+1}} + {}^{i+1}\omega_{i+1} \times \left({}^{i+1}\omega_{i+1} \times {}^{i+1}p_{C_{i+1}}\right) + {}^{i+1}\dot{v}_{i+1} \tag{A-8} $$

$$ {}^{i+1}F_{i+1} = m_{i+1}\,{}^{i+1}\dot{v}_{C_{i+1}} \tag{A-9} $$

$$ {}^{i+1}N_{i+1} = {}^{C_{i+1}}I_{i+1}\,{}^{i+1}\dot{\omega}_{i+1} + {}^{i+1}\omega_{i+1} \times {}^{C_{i+1}}I_{i+1}\,{}^{i+1}\omega_{i+1} \tag{A-10} $$

Step 2: ***Computation of force/moment and torque at joints.***
(i: N →1)

$$ {}^{i}f_{i} = {}^{i+1}_{i}R\,{}^{i+1}f_{i+1} + {}^{i}F_{i} \tag{A-11} $$

$$ {}^{i}n_{i} = {}^{i}N_{i} + {}^{i}_{i+1}R\,{}^{i+1}n_{i+1} + {}^{i}p_{C_i} \times {}^{i}F_{i} + {}^{i}p_{i+1} \times {}^{i}_{i+1}R\,{}^{i+1}f_{i+1} \tag{A-12} $$

$$ \tau_{i} = {}^{i}n_{i}^{T}\hat{z} \tag{A-13} $$

The full algorithm is given in Table A.1 for an N link manipulator with rotational joints (it can be expanded to include prismatic joints as well), taken directly from (Craig, 2005). The following terms are used in the algorithm (in addition to the terms defined in the previous paragraph): $\mathbf{{}^{i+1}_{i}R}$ is the rotation from vectors in frame i to vectors in frame i + 1, $\mathbf{p_i}$ is the position of the ith joint, $\mathbf{p_{C_i}}$ is the position of the ith link's center of mass, and **q** is the joint angle as defined in the Denavit-Hartenberg convention (sometimes called $\boldsymbol{\theta}$).

The algorithm consists of two steps: the first propagates angular velocity and acceleration of the joints as well as the linear velocity and acceleration at both the joints and each link's center of mass from the manipulator's base outwards to its last link. These velocities and accelerations are used with Newton's force equation and Euler's moment equation to generate forces and moments at each link's center of mass. The second step propagates inwards from the last link to the manipulator's base and calculates the force and moment at the joints given the force and moment at each link's center of mass. Torque at the joints is taken as the z direction of the moment since the joint frames are assumed to follow the Denavit-Hartenberg and be defined with the z direction being the axis of rotation.

After the algorithm completes both steps the manipulator's torque at each of its joints will be given by vectorizing equation (A-13). This equation, parameterized by the physical constants, is an explicit function of joint position, velocity and acceleration. The joint torque can then be placed in to the form (2-13) – the equation for manipulator dynamics having no friction or gearing.

The MATLAB code that implements this algorithm to generate the dynamical model used in this work is given in Appendix B.1.

A.4 Manipulator Case Study Dynamics

The manipulator used, described in Chapter 3, consisted of two links though they were defined differently for the kinematics and dynamics. The first link in the kinematics sense began at the first frame's origin and ended at the second frame's origin, in the middle of the hole in the bar. The second link was from the second frame's origin to the end of the bar – the point chosen to be the end effector location. The first link in the dynamics sense was from the first frame's origin to the center of mass of the second joint's motor and harmonic drive. This point was assumed to be the center of the cylinder containing the second joint's motor and harmonic drive (best seen in Figure 3.1). The mass and inertia of the first link was assumed to be zero. The second link consisted of five components whose inertias were added to make up the second link's total inertia. They were: the motor and harmonic drive combination, the plate below the force/torque sensor (the thickest plate in Figure 3.2) attaching it to the motor and harmonic drive, the force/torque sensor, the plate above the force/torque sensor attaching it to the bar and the bar itself. All parts except the bar were modeled as having the inertia of a solid cylinder. The bar was modeled as a bar of constant density and the parallel-axis theorem (Craig, 2005) was used to change the inertia's point of reference from halfway along the bar's length to the center of Frame 2. The reason the second link was made to include all of these components was because they are all rotate with that joint. The details of the inertias used for each component and the dimensions of each modeled part are in the Newton-Euler dynamics MATLAB code given in Appendix B.1. The resulting dynamics are given as

$$\tau = \begin{bmatrix} 0.055 + 0.004\cos^2(q_2) & 0.003\sin(q_2) \\ 0.003\sin(q_2) & 0.005 \end{bmatrix} \ddot{q} +$$

$$\begin{bmatrix} -0.008\cos(q_2)\sin(q_2)\dot{q}_2 & 0.003\cos(q_2)\dot{q}_2 \\ 0.004\cos(q_2)\sin(q_2)\dot{q}_1 & 0 \end{bmatrix} \dot{q} + \qquad \text{(A-14)}$$

$$\begin{bmatrix} -3.104\sin(q_1) + 0.219\cos(q_1)\cos(q_2) \\ -0.219\sin(q_1)\sin(q_2) \end{bmatrix}.$$

Note that this dynamical model is in the form (2-), lacking the inertia due to the harmonic drive's gearing and the friction term. The inertia due to the gearing is found as follows

$$\begin{aligned} G_{11}^2 I_m &= 161^2\left(3.74 * 10^{-5}\right) = .969 \\ G_{22}^2 I_m &= 160^2\left(3.74 * 10^{-5}\right) = .957 \end{aligned} \tag{A-15}$$

where the first and second joint's gearing is 161 and 160 respectively and the housed inertia of the motors used is $3.74 * 10^{-5}$ kg*m^2. The details of the mechanical aspects of the manipulator are given in section 3.2.1. The resulting full dynamical model for the manipulator, put into the form of equation (2-), is given as

$$\begin{aligned} \tau = &\begin{bmatrix} 1.012 + 0.004\cos^2(q_2) & 0.003\sin(q_2) \\ 0.003\sin(q_2) & 0.974 \end{bmatrix}\ddot{q} + \\ &\begin{bmatrix} -0.008\cos(q_2)\sin(q_2)\dot{q}_2 & 0.003\cos(q_2)\dot{q}_2 \\ 0.004\cos(q_2)\sin(q_2)\dot{q}_1 & 0 \end{bmatrix}\dot{q} + \\ &\begin{bmatrix} -3.104\sin(q_1) + 0.219\cos(q_1)\cos(q_2) \\ -0.219\sin(q_1)\sin(q_2) \end{bmatrix} + f_v(\dot{q}) \end{aligned} \tag{A-16}$$

where no assumptions are made on the form of the friction except strict dependence upon joint velocity and no coupling between joints. Equation (A-16) can be put into LIP form (see section 2.3): $\boldsymbol{\tau} = \mathbf{Y}(\ddot{\mathbf{q}}, \dot{\mathbf{q}}, \mathbf{q})\mathbf{a} + \mathbf{f}_v(\dot{\mathbf{q}})$, with **Y** and **a** defined as follows

$$Y^T(\ddot{q},\dot{q},q) = \begin{bmatrix} \ddot{q}_1 & 0 \\ \cos^2(q_2)\ddot{q}_1 & 0 \\ 0 & \ddot{q}_2 \\ \sin(q_2)\ddot{q}_2 & \sin(q_2)\ddot{q}_1 \\ \cos(q_2)\sin(q_2)\dot{q}_1\dot{q}_2 & 0 \\ \cos(q_2)\dot{q}_2^2 & 0 \\ 0 & \cos(q_2)\sin(q_2)\dot{q}_1^2 \\ \sin(q_1) & 0 \\ \cos(q_1)\cos(q_2) & -\sin(q_1)\sin(q_2) \end{bmatrix}, \quad a = \begin{bmatrix} 1.012 \\ 0.004 \\ 0.974 \\ 0.003 \\ -0.008 \\ 0.003 \\ 0.004 \\ -3.104 \\ 0.219 \end{bmatrix}. \tag{A-17}$$

Examining **a** in equation (A-17) reveals that only four parameters in this manipulator's dynamics significantly contribute to the torque seen at its joints. These parameters consist of the two parameters of the gravitational term, -3.104 and 0.219, and the two parts of the diagonal of the inertia matrix, 1.012 and 0.974, which are made significant by the gearing. Training trajectories, discussed in Section 4.1, and parameter evolution, discussed in Section 4.2, focus on accurate adaptation of these four parameters while keeping the other five parameters' values small.

A.5 Manipulator Case Study Controller

The specific form of the controller used is presented using (A-17) and the general form of the adaptive friction learning controller and adaptation laws presented in (2-48) through (2-51).

$$\tau = Y(\ddot{q}_r,\dot{q}_r,\dot{q},q)\hat{a} + \begin{bmatrix} \sum_{k=k_{MIN}}^{k=k_{MAX}} \hat{c}_{1,k} g(h\dot{q}_1 - k) \\ \sum_{k=k_{MIN}}^{k=k_{MAX}} \hat{c}_{2,k} g(h\dot{q}_2 - k) \end{bmatrix} - K_d s,$$

$$Y^T(\ddot{q}_r,\dot{q}_r,\dot{q},q)=\begin{bmatrix} \ddot{q}_{r1} & 0 \\ \cos^2(q_2)\ddot{q}_{r1} & 0 \\ 0 & \ddot{q}_{r2} \\ \sin(q_2)\ddot{q}_{r2} & \sin(q_2)\ddot{q}_{r1} \\ \cos(q_2)\sin(q_2)\dot{q}_1\dot{q}_{r2} & 0 \\ \cos(q_2)\dot{q}_2\dot{q}_{r2} & 0 \\ 0 & \cos(q_2)\sin(q_2)\dot{q}_1\dot{q}_{r1} \\ \sin(q_1) & 0 \\ \cos(q_1)\cos(q_2) & -\sin(q_1)\sin(q_2) \end{bmatrix}, \tag{A-18}$$

$$\dot{\hat{a}} = -\Gamma Y^T s_\Delta,$$

$$\dot{\hat{c}}_{i,k} = -\gamma_c g(hq_i - k)s_{\Delta i}$$

with the adaptive networks' basis function **g** defined in Section 2.3 and $\boldsymbol{\Gamma = \mathrm{diag}(\gamma_1,\gamma_2,\gamma_1,\gamma_2,\gamma_2,\gamma_2,\gamma_2,\gamma_1,\gamma_1)}$ where $\boldsymbol{\gamma_1 = 200}$ and $\boldsymbol{\gamma_2 = 2}$. In (A-18) the adaptation laws use $\mathbf{s_\Delta}$, a modified version of the tracking error **s**. The components of this modified error are defined as follows

$$s_{\Delta i} = s_i - \phi \mathrm{sat}(s_i/\phi), \quad i = 1,\ldots,N \tag{A-19}$$

where **sat( )** is the saturation function defined as follows

$$\mathrm{sat}(x) = \begin{cases} x, & |x| < 1 \\ 1, & |x| \ge 1 \end{cases}. \tag{A-20}$$

Subtracting the saturation function from **s** forms a deadzone around zero of size $\boldsymbol{\phi}$. The deadzone was used in (Liu, 1997) because it assures that the adaptation does not cause instability by trying to achieve perfect **s = 0** tracking. The value of $\boldsymbol{\phi}$ chosen was 0.001 in training.

The equation for $\dot{\hat{\mathbf{a}}}$ is broken up as follows

$$\dot{\hat{a}}_1 = -\gamma_1 \ddot{q}_{r1} s_{\Delta 1} \tag{A-21}$$

$$\dot{\hat{a}}_2 = -\gamma_2 \cos^2(q_2) \ddot{q}_{r1} s_{\Delta 1} \tag{A-22}$$

$$\dot{\hat{a}}_3 = -\gamma_1 \ddot{q}_{r2} s_{\Delta 2} \tag{A-23}$$

$$\dot{\hat{a}}_4 = -\gamma_2 \left(\sin(q_2) \ddot{q}_{r2} s_{\Delta 1} + \sin(q_2) \ddot{q}_{r1} s_{\Delta 2}\right) \tag{A-24}$$

$$\dot{\hat{a}}_5 = -\gamma_2 \cos(q_2) \sin(q_2) \dot{q}_1 \dot{q}_{r2} s_{\Delta 1} \tag{A-25}$$

$$\dot{\hat{a}}_6 = -\gamma_2 \cos(q_2) \dot{q}_2 \dot{q}_{r2} s_{\Delta 1} \tag{A-26}$$

$$\dot{\hat{a}}_7 = -\gamma_2 \cos(q_2) \sin(q_2) \dot{q}_1 \dot{q}_{r1} s_{\Delta 2} \tag{A-27}$$

$$\dot{\hat{a}}_8 = -\gamma_1 \sin(q_1) s_{\Delta 1} \tag{A-28}$$

$$\dot{\hat{a}}_9 = -\gamma_1 \left(\cos(q_1) \cos(q_2) s_{\Delta 1} - \sin(q_1) \sin(q_2) s_{\Delta 2}\right). \tag{A-29}$$

The adaptation laws of the four significant parameters (described in Appendix A.4) are given by (A-21), (A-23), (A-28), and (A-29). These laws make use of the larger $\boldsymbol{\gamma}_1$ adaptation gain while the remaining parameters use the $\boldsymbol{\gamma}_2$ gain. The reason for this distinction in the gains was that it eased the process of choosing persistently exciting training trajectories, described in section 4.1.

The adaptive networks used in the controller (A-18) are parameterized by four constants chosen by the designer: $\boldsymbol{\gamma}_\mathbf{c}$, **h**, $\mathbf{k}_{\mathbf{min}}$, and $\mathbf{k}_{\mathbf{max}}$. As discussed in Section 2.3, the center of node **k** is at **k/h**, so the constants $\mathbf{k}_{\mathbf{min}}$, $\mathbf{k}_{\mathbf{max}}$, and **h** should be chosen such that the range of joint velocity that is desired to be covered by the networks is within $[\mathbf{k}_{\mathbf{min}}/\mathbf{h},\mathbf{k}_{\mathbf{max}}/\mathbf{h}]$. In the case of the given manipulator that range of the joints' velocity was chosen to be [-1 rad/s, 1 rad/s]. Since this is symmetric about zero $\mathbf{k}_{\mathbf{max}} = -\mathbf{k}_{\mathbf{min}}$ and choosing $\mathbf{h}$ to be 20 (as in Liu (1997) and Guion (2003)) results in $\mathbf{k}_{\mathbf{max}} = \mathbf{20}$. Hence there are 41 nodes in the network since there is a node centered at zero. The learning gain $\boldsymbol{\gamma}_{\mathbf{c}}$ was chosen to be ten times larger than $\boldsymbol{\gamma}_{\mathbf{1}}$ because the friction tends to be a major term, if not the dominant term, in the dynamics of harmonically driven manipulators, so it should be learned at a higher rate if convergence is to take place quickly.

Two different sets of PD gains were used with controller. In training mode the PD gains were smaller, thereby increasing the amplitude of the error signals. The larger and therefore less noisy error signals were better suited for adaptation and learning because the adaptation laws are based on the tracking error. Their values in training mode were $\mathbf{K}_{\mathbf{p}} = \mathbf{5000I}_2$ and $\mathbf{K}_{\mathbf{p}} = \mathbf{300I}_2$ (with $\boldsymbol{\Lambda} = \mathbf{16.7I}_2$ as a result). In testing/estimation mode, the PD gains were increased significantly to reduce the effect of stiction during stationary trajectories. This turned out to be the most effective method of dealing with stiction. Other methods of reducing stiction were attempted including: dithering the commanded torque, dithering the desired trajectory, and modifying the velocity in the low velocity regime (Hauschild, 2004).

The PD gains in testing/estimation mode were

$$\mathrm{K}_{\mathrm{p}} = \begin{bmatrix} 320000 & 0 \\ 0 & 210000 \end{bmatrix} \tag{A-30}$$

$$\mathrm{K}_{\mathrm{d}} = 1700\mathrm{I}_2 \tag{A-31}$$

$$\Lambda = K_d^{-1} K_p \begin{bmatrix} 188.2 & 0 \\ 0 & 123.5 \end{bmatrix}. \tag{A-32}$$

High PD gains can be used when both good position resolution and good velocity estimates are available. The hardware, both mechanical and electrical, and software used for this manipulator made both things possible. The high gear ratios in both joints provided by the harmonic drives as well as the very accurate encoders led to the good position resolution. On the software end, the use of a real-time kernel permitted a high control frequency, which in turn led to good velocity estimates. The details of both the hardware and software are given in the sections 3.2 and 3.3.

A.6 Constants

TABLE A.2
CONSTANTS FROM HARDWARE EXPERIMENTS

Category	**Name**	**Value**	**Units**
Kinematics	L_1	0.194	m
Kinematics	L_2	0.259	m
Controller - Train	K_{d1}	300	N-m-s/rad
Controller - Train	K_{p1}	5000	N-m/rad
Controller - Train	$\Lambda_1 = K_{p1} / K_{d1}$	16.7	
Controller - Train	K_{d2}	300	N-m-s/rad
Controller - Train	K_{p2}	5000	N-m/rad
Controller - Train	$\Lambda_2 = K_{p2} / K_{d2}$	16.7	
Control Gain - Test	K_{d1}	1700	N-m-s/rad
Control Gain - Test	K_{p1}	320000	N-m/rad
Control Gain - Test	$\Lambda_1 = K_{p1} / K_{d1}$	188.2	
Control Gain - Test	K_{d2}	1700	N-m-s/rad
Control Gain - Test	K_{p2}	210000	N-m/rad
Control Gain - Test	$\Lambda_2 = K_{p2} / K_{d2}$	123.5	

Adaptive Networks	h	100	
Adaptive Networks	k_{min}	-100	
Adaptive Networks	k_{max}	100	
Adap'n Gain - Train	γ_1	200	
Adap'n Gain - Train	$\gamma_2 = \gamma_1 / 100$	2	
Adap'n Gain - Train	γ_c	2000	
Adap'n Gain - Test	γ_1	0	
Adap'n Gain - Test	$\gamma_2 = \gamma_1 / 100$	0	
Adap'n Gain - Test	γ_c	0	
Hardware – Elec'l	K_{m1}	0.0855	N-m/A
Hardware – Elec'l	Geared CPR_1	5796000	counts
Hardware – Elec'l	K_{m2}	0.0855	N-m/A
Hardware – Elec'l	Geared CPR_2	5760000	counts
Hardware – Mech'l	N_1	161	
Hardware – Mech'l	I_{m1}	$3.74*10^{-5}$	Kg-m^2
Hardware – Mech'l	N_2	160	
Hardware – Mech'l	I_{m2}	$3.74*10^{-5}$	Kg-m^2

TABLE A.3
CHANGED CONSTANTS IN SIMULATION

Category	**Name**	**Value**	**Units**
Adaptive Networks	h	10.5	
Adaptive Networks	k_{min}	-20	
Adaptive Networks	k_{max}	20	
Adap'n Gain - Train	γ_1	1000	
Adap'n Gain - Train	$\gamma_2 = \gamma_1 / 100$	10	
Adap'n Gain - Train	γ_c	3000	
Adap'n Gain - Test	γ_1	0	
Adap'n Gain - Test	$\gamma_2 = \gamma_1 / 100$	0	
Adap'n Gain - Test	γ_c	0	

Appendix B

B.1 Newton-Euler Dynamics Code – MATLAB

```
%Leon Aksman
%Space Systems Lab
%University of Maryland, College Park 20742
%copyright 2006

%Dynamics for pitch-roll 2 DOF system
%Uses Newton-Euler algorithm with MATLAB's Symbolic Math Toolbox

clear

%frame transformations, rotations, translations
syms L1 real;
syms theta_1 theta_1_dot theta_1_dot_dot real;
syms theta_2 theta_2_dot theta_2_dot_dot real;
T_0_1 = [  cos(theta_1) -sin(theta_1)   0   0; ...
                      0             0   -1  0; ...
           sin(theta_1)  cos(theta_1)   0   0; ...
                      0             0   0   1];

T_1_2 = [cos(theta_2) -sin(theta_2)     0   0; ...
                    0             0     1   L1; ...
        -sin(theta_2) -cos(theta_2)     0   0; ...
                    0             0     0   1];

R_0_1 = T_0_1(1:3, 1:3); P_0_1 = T_0_1(1:3, 4);
R_1_2 = T_1_2(1:3, 1:3); P_1_2 = T_1_2(1:3, 4);

R_0_2 = R_0_1*R_1_2;
T_0_2 = T_0_1*T_1_2


%position of the centers of mass of the links relative to link frame center
syms CM_x_2 CM_z_2 real;
P_C_1_1 = [0; 0; 0];
P_C_2_2 = [CM_x_2; 0; CM_z_2];

%joint variables
theta_dot =     [theta_1_dot; theta_2_dot];
theta_dot_dot = [theta_1_dot_dot; theta_2_dot_dot];
%d_dot =         [0; 0];    %prismatic
%d_dot_dot =     [0; 0];

Z = [0; 0; 1];

%matrix of P_i_i+1 terms
syms zero real;
P_2_3 = [zero; zero; zero];             %---- last one is always zero
P = [P_0_1 P_1_2 P_2_3];

%matrix of P_C_i+1_i+1 terms
P_C = [P_C_1_1 P_C_2_2];

%initial angular velocity and accel., linear accel.
w = [0; 0; 0];
w_dot = [ 0; 0; 0];
syms g real;
v_dot = [0; 0; g];

syms M_MB h_MB R_MB d_MB_2 real;

%inertia of MORPHbot roll motor and harmonic drive relative to frame 2 center
```

```
%syms M_MB_P h_MB_P R_MB_P real;
I_MB_1 = [(1/12)*M_MB*h_MB^2 + (1/4)*M_MB*R_MB^2    0           0; ...
                 0          (1/12)*M_MB*h_MB^2 + (1/4)*M_MB*R_MB^2       0; ...
                 0                    0                             .5*M_MB*R_MB^2];

%inertia of link 1
I_1 = I_MB_1;


%inertia of MORPHbot roll motor and harmonic drive relative to frame 2 center
I_MB_2 = [(1/12)*M_MB*h_MB^2 + (1/4)*M_MB*R_MB^2 + M_MB*d_MB_2^2   0        0; ...
          0      (1/12)*M_MB*h_MB^2 + (1/4)*M_MB*R_MB^2 + M_MB*d_MB_2^2     0; ...
          0         0                                           .5*M_MB*R_MB^2];
%inertia of bottom plate relative to frame 2 center
syms M_bp h_bp R_bp d_bp_2 real;
I_bp_2 = [(1/12)*M_bp*h_bp^2 + (1/4)*M_bp*R_bp^2 + M_bp*d_bp_2^2  0         0; ...
          0     (1/12)*M_bp*h_bp^2 + (1/4)*M_bp*R_bp^2 + M_bp*d_bp_2^2      0; ...
          0                 0                                   .5*M_bp*R_bp^2 ];
%inertia of force/torque sensor relative to frame 2 center
syms M_fts h_fts R_fts d_fts_2 real;
I_fts_2 = [(1/12)*M_fts*h_fts^2 + (1/4)*M_fts*R_fts^2 + M_fts*d_fts_2^2  0   0; ...
            0   (1/12)*M_fts*h_fts^2 + (1/4)*M_fts*R_fts^2 + M_fts*d_fts_2^2 0; ...
            0           0                                       .5*M_fts*R_fts^2 ];
%inertia of top plate relative to frame 2 center
syms M_tp h_tp R_tp d_tp_2 real;
I_tp_2 = [(1/12)*M_tp*h_tp^2 + (1/4)*M_tp*R_tp^2 + M_tp*d_tp_2^2   0      0; ...
          0     (1/12)*M_tp*h_tp^2 + (1/4)*M_tp*R_tp^2 + M_tp*d_tp_2^2     0; ...
         0          0                                       .5*M_tp*R_tp^2 ];
%inertia of bar relative to frame 2 center
syms M_bar h_bar l_bar w_bar d_bar_2 real;
I_bar_2 = [(1/12)*M_bar*(h_bar^2 + l_bar^2)      0                       0; ...
           0      (1/12)*M_bar*(w_bar^2 + h_bar^2) + M_bar * d_bar_2^2  0; ...
           0           0     (1/12)*M_bar*(l_bar^2 + w_bar^2) + M_bar * d_bar_2^2];
%inertia of link 2
I_2 = I_MB_2 + I_bp_2 + I_fts_2 + I_tp_2 + I_bar_2;

%the masses of the links
syms m1 m2 real;
m1 = M_MB;
m2 = M_MB + M_bp + M_fts + M_tp + M_bar;
m = [m1 m2];

DOFs = 2;       %number of degrees of freedom

%initialization of matices
F =     ones(3, DOFs)*zero;
N =     ones(3, DOFs)*zero;

for i = 1:DOFs
    switch i
        case 1
            R = R_0_1';
        case 2
            R = R_1_2';
    end

    switch i
        case 1
            I = I_1;
        case 2
            I = I_2;
    end

    w_prev = w;
    w = R*w_prev + theta_dot(i)*Z;

    w_dot_prev = w_dot;
    w_dot = R * w_dot_prev + cross((R*w_prev), [0; 0; theta_dot(i)]) + [0; 0;
theta_dot_dot(i)];

    v_dot_prev = v_dot;
    v_dot = R*(cross(w_dot_prev, P(:, i)) + cross(w_prev, cross(w_prev, P(:, i))) +
v_dot_prev);
            %+ 2*cross(w_dot, d_dot(i)*Z) + d_dot_dot(i)*Z;         %for prismatic
joints

    v_C_dot = cross(w_dot, P_C(:, i)) + cross(w, cross(w, P_C(:, i))) + v_dot;

    F(:, i) = m(i)*v_C_dot;
    N(:, i) = I*w_dot + cross(w, I*w);

end

F = subs(F, zero, 0);
N = subs(N, zero, 0);

%initialization of force/moment vectors - zeroed because 0 ext. f/t assumed
f =     ones(3, DOFs + 1)*zero;
n =     ones(3, DOFs + 1)*zero;
tau =   ones(DOFs, 1)*zero;

for i = DOFs:-1:1
    switch i
        case 1
            R = R_1_2;
        case 2
            R = eye(3);    %last rotation matrix does not matter if 0 ext. f/t assumed
    end

    f(:, i) = R*f(:, i+1) + F(:, i);
    n(:, i) = N(:, i) + R*n(:, i+1) + cross(P_C(:, i), F(:, i)) + cross(P(:, i+1),…
             R*f(:, i+1));

    tau(i) = n(:, i)'*Z;        %revolute joint
    %tau(i) = f(:, i)'*Z;       %prismatic joint
end

%dynamics with gravitational term
tau = simplify(subs(tau, zero, 0))

%zero gravity dynamics
tau_no_g = simplify(subs(tau, g, 0))

%inertial term
tau_I = simplify(subs(tau_no_g, theta_1_dot, 0));
tau_I = simplify(subs(tau_I, theta_2_dot, 0))

%coriolis, centripetal term
tau_C = simplify(tau_no_g - tau_I)

%gravitational term
tau_g = simplify(tau - tau_no_g)


%-----------tau with constants---------

%g
tau = subs(tau, g,        9.8);

%CM_x_2 CM_z_2 L1
tau = subs(tau, CM_x_2,    .0096); %.052
tau = subs(tau, CM_z_2,   -.058);  %-.003
tau = subs(tau, L1,        .194);

%M_MB h_MB R_MB
tau = subs(tau, M_MB,      .842);
tau = subs(tau, h_MB,      .047);
tau = subs(tau, R_MB,      .039);
tau = subs(tau, d_MB_2,   -.099);

%M_bp h_bp R_bp d_bp_2
tau = subs(tau, M_bp,      .415);
tau = subs(tau, h_bp,       .019);
tau = subs(tau, R_bp,       .006);
tau = subs(tau, d_bp_2,    -.066);

%M_fts h_fts R_fts d_fts_2
tau = subs(tau, M_fts,      .639);
tau = subs(tau, h_fts,      .042);
tau = subs(tau, R_fts,      .005);
tau = subs(tau, d_fts_2,   -.036);

%M_tp h_tp R_tp d_tp_2
tau = subs(tau, M_tp,       .229);
tau = subs(tau, h_tp,       .011);
tau = subs(tau, R_tp,       .05);
tau = subs(tau, d_tp_2,    -.009);

%M_bar h_bar l_bar w_bar d_bar_2
tau = subs(tau, M_bar,      .204);
tau = subs(tau, h_bar,      .007);
tau = subs(tau, l_bar,      .039);
tau = subs(tau, w_bar,      .298);
tau = subs(tau, d_bar_2,    .11);


tau = simplify(tau)
```

B.2 Control Code – C

START OF CONTROLLIB.C CODE

```
/*
  $Id$

(c) Copyright 1999-2006
 Space Systems Lab, University of Maryland, College Park, MD 20740

 Definitions of control law functions using feedback and model based techniques

 HISTORY

 Apr-2006     L Aksman       Created from main.c
 Jun-2006        L Aksman             Generalized to N DOF
 */

#include "main.h"
#include "ControlLib.h"

/***************************************************************************
FUNCTION DECLARATIONS
***************************************************************************/

/* return 1 and print error message if ptr equals NULL, 0 o/w
   char * parameter is name of variable - used in error message.
 */
int
equalsNULL(void * ptr, char * name);

/* saturation function */
double
sat(double x);

/***************************************************************************
FUNCTION DEFINITIONS
***************************************************************************/

/*      Function that controls gathering input, calculating desired torque and output
for current control cycle.
        Currently called by Closed_Loop_Control_N_DOF( ).
        This function also reads the force/torque sensor and generates force
estimates, saving in Force_Estimation variable.
        Generalized to N DOFs, calling dynamics and kinematics functions specified in
Kinematics_Dynamics_Functions variable.

        parameters:
        des_pos        - desired angular positions for all N DOFs in radians
        des_vel        - desired angular velocities for all N DOFs in rad/s
        des_accel      - desired angular accelerations for all N DOFs in rad/s^2
        learn_modeled_params   - controls whether adaptation/learning of modeled
                              dynamical parameters takes place during this cycle
        learn_unmodeled_params  - controls whether adaptation/learning of unmodeled
                              dynamical parameters takes place during this cycle
        kin_dyn_fns           - pointer to variable holding pointers to kinematics and
                              dynamics functions for given manipulator
        dof                   - array (of size N) of pointers to Single_DOF_Properties
                              variables holding information about each DOF
                              NOTE: DOFs must be in correct order with those closest
                              to base of the manipulator coming first.For example,
                              starting at the base, the MORPHbot module has a pitch
                              DOF followed by a roll DOF so this array would contain 2
                              elements - the Single_DOF_Properties of the pitch at
                              index 0 and the Single_DOF_Properties of the roll at
                              index 1.
        N - number of DOFs to control
*/
```

```
int
Control_Law_N_DOF(     double                                               * des_pos,
                               double
       * des_vel,
                               double
       * des_accel,
                               BOOLEAN
       learn_modeled_params,
                               BOOLEAN
       learn_unmodeled_params,
                               Kinematics_Dynamics_Functions * kin_dyn_fns,
                               Force_Estimation        * force_estimation,
                               Single_DOF_Properties                ** dof,
                               int
               N)
{
       /* static variables */
       static ftsdrv_6DOF_t          force_moment = {{0.}, {0.}};

       /* non-static variables */
       int                           volts_bits;
       int                           return_val_write;
       int                           return_val_read;
       int                           rc;
       int                           i;
       int                           j;
       int                           count_i;
       double                        act_accel_unfilt;
       double                        volts_in;
       double                        volts_out;
       double                        torque_thresh;
       double                        torque_external_est_new;
       double                        force_est[3];
       double                        force_est_thresh[3];
       struct timespec               diff;

       /* size N arrays */
       /* NOTE: assert( ) not used to check calloc( ) failure because it may end the
       program unsafely */
       double * dt =                 (double *) calloc(N, sizeof(double));
       if (equalsNULL((void *) dt, "dt")) return -1;
       double * dt_prev =            (double *) calloc(N, sizeof(double));
       if (equalsNULL((void *) dt_prev, "dt_prev")) return -1;
       double * act_pos =            (double *) calloc(N, sizeof(double));
       if (equalsNULL((void *) act_pos, "act_pos")) return -1;
       double * act_pos_prev =       (double *) calloc(N, sizeof(double));
       if (equalsNULL((void *) act_pos_prev, "act_pos_prev")) return -1;
       double * act_vel =            (double *) calloc(N, sizeof(double));
       if (equalsNULL((void *) act_vel, "act_vel")) return -1;
       double * act_vel_prev =       (double *) calloc(N, sizeof(double));
       if (equalsNULL((void *) act_vel_prev, "act_vel_prev")) return -1;
       double * act_accel =          (double *) calloc(N, sizeof(double));
       if (equalsNULL((void *) act_accel, "act_accel")) return -1;
       double * act_accel_prev =     (double *) calloc(N, sizeof(double));
       if (equalsNULL((void *) act_accel_prev, "act_accel_prev")) return -1;
       double * des_vel_r =          (double *) calloc(N, sizeof(double));
       if (equalsNULL((void *) des_vel_r, "des_vel_r")) return -1;
       double * des_accel_r =        (double *) calloc(N, sizeof(double));
       if (equalsNULL((void *) des_accel_r, "des_accel_r")) return -1;
       double * error_pos =          (double *) calloc(N, sizeof(double));
       if (equalsNULL((void *) error_pos, "error_pos")) return -1;
       double * error_vel =          (double *) calloc(N, sizeof(double));
       if (equalsNULL((void *) error_vel, "error_vel")) return -1;
       double * pseudo_vel =         (double *) calloc(N, sizeof(double));
       if (equalsNULL((void *) pseudo_vel, "pseudo_vel")) return -1;
       double * Kp =                 (double *) calloc(N, sizeof(double));
       if (equalsNULL((void *) Kp, "Kp")) return -1;
       double * Kd =                 (double *) calloc(N, sizeof(double));
       if (equalsNULL((void *) Kd, "Kd")) return -1;
       double * s =                  (double *) calloc(N, sizeof(double));
       if (equalsNULL((void *) s, "s")) return -1;
```

```
        double * s_DELTA =            (double *) calloc(N, sizeof(double));
        if (equalsNULL((void *) s_DELTA, "s_DELTA")) return -1;
        double * s_DELTA_prev =       (double *) calloc(N, sizeof(double));
        if (equalsNULL((void *) s_DELTA_prev, "s_DELTA_prev")) return -1;
        double * torque =             (double *) calloc(N, sizeof(double));
        if (equalsNULL((void *) torque, "torque")) return -1;
        double * torque_external =    (double *) calloc(N, sizeof(double));
        if (equalsNULL((void *) torque_external, "torque_external")) return -1;
        double * torque_external_est =(double *) calloc(N, sizeof(double));
        if (equalsNULL((void *) torque_external_est, "torque_external_est")) return -
1;
        double * torque_external_est_thresh = (double *) calloc(N, sizeof(double));
        if (equalsNULL((void *) torque_external_est_thresh,
        "torque_external_est_thresh")) return -1;
        double * torque_dynamic =     (double *) calloc(N, sizeof(double));
        if (equalsNULL((void *) torque_dynamic, "torque_dynamic")) return -1;
        double * torque_viscous =     (double *) calloc(N, sizeof(double));
        if (equalsNULL((void *) torque_viscous, "torque_viscous")) return -1;
        double * torque_hinges =      (double *) calloc(N, sizeof(double));

        if (equalsNULL((void *) torque_hinges, "torque_hinges")) return -1;


        j = 0;
        torque_thresh = 0.;

        /* update previous values of state */
        for (i = 0; i < N; i++)
        {
        dof[i]->control_state->counter_val_prev =dof[i]->control_state->counter_val;
        dof[i]->control_state->act_pos_prev =       dof[i]->control_state->act_pos;
        dof[i]->control_state->act_vel_prev =       dof[i]->control_state->act_vel;
        dof[i]->control_state->act_accel_prev =     dof[i]->control_state->act_accel;

                /* form previous dt from difference of previous timestamps */
                rclDiffTimespecs(&dof[i]->control_state->time_stamp_ts,
                                &dof[i]->control_state->time_stamp_prev_ts,
                                &diff);
                dt_prev[i] =   ((double) rclTimespecToMicroseconds(&diff))*1e-6;
        }

        for (i = 0; i < N; i++)
        {
                count_i = 0;
                while (count_i < 4)           /* 3 */
                {
                 dof[i]->control_state->act_vel = dof[i]->control_state->act_vel_prev;
             dof[i]->control_state->act_accel = dof[i]->control_state->act_accel_prev;

                        /* read the counters*/
                        dof[i]->control_state->counter_val =
                        ComediReadCounterWithRollover(daq_device, CTR_SUBDEVICE,
                        dof[i]->motor_constants->CTR_CHAN,
                        dof[i]->control_state->counter_val_prev);

                        if (dof[i]->control_state->counter_val == INT_MAX)
                        {
                        printf("ERROR: ComediReadCounterWithRollover() failed. \n");
                        return -1;
                        }

                        /* get current time stamp and time difference*/
                        rc = clock_gettime(CLOCK_REALTIME,
                                          &dof[i]->control_state->time_stamp_ts);
                        rclDiffTimespecs(&dof[i]->control_state->time_stamp_ts,
                                        &dof[i]->control_state->time_stamp_prev_ts,
                                        &diff);
                        dt[i] = ((double) rclTimespecToMicroseconds(&diff))*1e-6;

                        /*convert timespec time stamp to double precision time stamp */
                        rclDiffTimespecs(&dof[i]->control_state->time_stamp_ts,
                                                &initial,
                                                &diff);
                         dof[i]->control_state->time_stamp = ((double)
                        rclTimespecToMicroseconds(&diff))*1e-6;

                        /* calculate actual position from counter value */
                        dof[i]->control_state->act_pos =
                        dof[i]->control_state->counter_val*dof[i]->
                        motor_constants->CONV_COUNTS_TO_RADIANS;

                /* check current position agains soft stop position for this DOF */
                        if (fabs(dof[i]->control_state->act_pos) >=
                            dof[i]->motor_constants->SOFT_STOP_POS)
                        {
                        printf("ERROR: Current position is %f radians. Reached soft
                        stop for DOF: %s Exiting.\n",

                        dof[i]->control_state->act_pos, dof[i]->motor_name);
                                return -1;
                        }

                /* calculate actual velocity from filtered position differencing */
                        dof[i]->control_state->act_vel = .3*dof[i]->control_state-
                        >act_vel +
                        .7*(dof[i]->control_state->act_pos - dof[i]->control_state-
                        >act_pos_prev)/dt[i];

                        act_accel_unfilt = (dof[i]->control_state->act_vel -
                        dof[i]->control_state->act_vel_prev)/dt[i];

                        /* if reading is within the acceleration bounds stop reading
                        the counter for this DOF */
                        if (fabs(act_accel_unfilt) <= MAX_ACT_ACCEL)
                        {
                                break;
                        }

                        count_i++;
                }

                /* filter actual acceleration */

                        dof[i]->control_state->act_accel =
                        Filter_Digital_Signal(act_accel_unfilt,
                                              dof[i]->control_state->act_accel_unfilt,
                                              dof[i]->control_state->act_accel_filt);
                        dof[i]->control_state->time_stamp_prev_ts = dof[i]->
                                 control_state->time_stamp_ts;
        }

        /* create some useful arrays */
        for (i = 0; i < N; i++)
        {
                act_pos[i] =          dof[i]->control_state->act_pos;
                act_pos_prev[i] =     dof[i]->control_state->act_pos_prev;
                act_vel[i] =          dof[i]->control_state->act_vel;
                act_vel_prev[i] =     dof[i]->control_state->act_vel_prev;
                act_accel[i] =        dof[i]->control_state->act_accel;
                act_accel_prev[i] =   dof[i]->control_state->act_accel_prev;

                /* s_DELTA has not been updated yet so it's still the previous value */
                s_DELTA_prev[i] =     dof[i]->control_state->s_DELTA;
        }

        /* ------------------- EXTERNAL FORCE ESTIMATION ----------------------- */
    /* get dynamical torques from previous state  - adaptation_flag is always FALSE*/
    /* NOTE: in this case the third and fourth parameters in this function call
    are the previous actual velocities and accelerations rather than des_vel_r_prev,
    des_accel_r_prev because we are not controlling - we simply want the modelled dynamic
    torque from the previous state*/
        kin_dyn_fns->Dynamic_Adaptive_Torque_N_DOF(act_pos_prev, act_vel_prev,
          act_vel_prev, act_accel_prev, s_DELTA_prev, dt_prev, FALSE, torque_dynamic);

          for (i = 0; i < N; i++)
          {
                 /* read the motor current, convert to get motor torque*/
                 return_val_read = comedi_data_read(daq_device, AI_SUBDEVICE,
                                dof[i]->motor_constants->AI_CHAN, AI_RANGE_1, AREF,
                                &volts_bits);
                 volts_in = comedi_to_phys(volts_bits, input_cr, input_max_value);
                 dof[i]->control_state->torque_motor =
                        dof[i]->motor_constants->CONV_VOLTS_IN_TO_TORQUE*volts_in;


                 if (USE_VISCOUS_NN)
                 {
                        torque_viscous[i] =
                                kin_dyn_fns->Viscous_Friction_Torque(
                                               dof[i]->control_state->act_vel_prev,
                                               s_DELTA_prev[i], dt_prev[i], dof[i],
                                               FALSE);
                 }
                 if (USE_HINGES)
                 {
                        torque_hinges[i] = kin_dyn_fns->Hinges_Torque(
                                               dof[i]->control_state->act_vel_prev,
                                               s_DELTA_prev[i], dt_prev[i], dof[i],
                                               FALSE);
                 }




                 /* calculate new estimate of external torque based on subtracting motor
                 torque from previous model torque*/
                  torque_external_est_new = torque_dynamic[i] + torque_viscous[i] +
                  torque_hinges[i] - dof[i]->control_state->torque_motor;

                 /* filter the estimate by combining with previous estimate */
                 dof[i]->control_state->torque_external_est =
                 Filter_Digital_Signal(torque_external_est_new,
                                       dof[i]->control_state->torque_ext_est_unfilt,

                  dof[i]->control_state->torque_ext_est_filt);
          }

    /* form torque_external_est array by combining each dof's torque_external_est */
          for (i = 0; i < N; i++)
          {
                 torque_external_est[i] = dof[i]->control_state->torque_external_est;
          }

          /* get estimated force vector by transforming estimated joint torque - stored
          in force_est input parameter */
          kin_dyn_fns->Translational_Jacobian_Transpose_Inverse(
                                act_pos_prev, torque_external_est, force_est);

          /* save actual and estimated force in Force_Estimation type variable
          (estimated moment not implemented) */
          for (i = 0; i < 3; i++)
          {
                 force_estimation->force[i] = force_moment.force[i];
                 force_estimation->moment[i] = force_moment.moment[i];
                 force_estimation->force_est[i] = force_est[i];
          }

          /* threshold external joint torque estimate */
          for (i = 0; i < N; i++)
          {
                 #if THRESHOLD_ESTIMATES
                   /* thresholding - zeroed in low external torque, reduced in higher */
```

```
        if (fabs(des_vel[i]) < ABS_VEL_MAX_THRESHOLD)
        {
                torque_thresh = TORQUE_EST_THRESH_LOW_VEL;
        }
        else if (fabs(des_vel[i]) < (ABS_VEL_MAX_THRESHOLD +
        ABS_VEL_TRANSITION_WIDTH))
        {
                /* linear transition between TORQUE_EST_THRESH_LOW_VEL
                when |velocity| < ABS_VEL_MAX_THRESHOLD  and
                TORQUE_EST_THRESH         when |velocity| >
                ABS_VEL_MAX_THRESHOLD + ABS_VEL_TRANSITION_WIDTH */
                torque_thresh = TORQUE_EST_THRESH_LOW_VEL
                              + (TORQUE_EST_THRESH -
                              TORQUE_EST_THRESH_LOW_VEL)*((fabs(des_vel[
                              i]) -
                              ABS_VEL_MAX_THRESHOLD)/ABS_VEL_TRANSITION_
                              WIDTH);
        }
        else
        {
                torque_thresh = TORQUE_EST_THRESH;
        }

        dof[i]->control_state->torque_external_est  =
        dof[i]->control_state->torque_external_est -
        torque_thresh*sat(dof[i]->control_state->
        torque_external_est/torque_thresh);
#endif

dof[i]->control_state->torque_external_est_LV_filt =
        dof[i]->control_state->torque_external_est;

/* special low velocity estimated external torque filtering */
#if LV_FILT
        if (fabs(des_vel[i]) < ABS_VEL_MAX_THRESHOLD)
        {
                /* check if just entered low velocity regime */
                if (dof[i]->control_state->low_velocity_regime == FALSE)
                {
                dof[i]->control_state->low_velocity_regime = TRUE;
                        dof[i]->control_state->moving_average_count = 0;
                }

                if (dof[i]->control_state->moving_average_count ==
                MOVING_AVERAGE_WIDTH)
                /* samples vector full,
                        start throwing out oldest sample */
                {
                        /* shift samples vector left */
                        for (j = 0; j < (MOVING_AVERAGE_WIDTH - 1); j++)
                        {
                dof[i]->control_state->moving_average_samples[j] =
                dof[i]->control_state->moving_average_samples[j + 1];
                        }

                /* insert latest sample at the back of the vector */
                dof[i]->control_state->
                        moving_average_samples[MOVING_AVERAGE_WIDTH - 1]
                = dof[i]->control_state->torque_external_est;
                }
                else    /* haven't filled in samples vector fully yet */
                {
                /* add latest sample into current back of the vector */
                        dof[i]->control_state->
                                moving_average_samples[dof[i]->
                                                     control_state->
                                             moving_average_count] =
                                             dof[i]->control_state->
                                             torque_external_est;
                        dof[i]->control_state->moving_average_count++;
```

```
                            }

/* form low velocity (LV) filtered torque estimate from moving average samples */
                            dof[i]->control_state->torque_external_est_LV_filt = 0.;

                            for (j = 0; j < dof[i]->control_state->
                                            moving_average_count; j++)
                            {
                                   dof[i]->control_state->
                                          torque_external_est_LV_filt += dof[i]->
                                                                 control_state->
                                                          moving_average_samples[j];
                            }
                            dof[i]->control_state->
                                   torque_external_est_LV_filt /= dof[i]->
                                                                 control_state->
                                                            moving_average_count;
                     }
                     else
                     {
                            dof[i]->control_state->low_velocity_regime = FALSE;
                            dof[i]->control_state->moving_average_count = 0;
                     }
              #endif
       }

       /* form thresholded torque_external_est array by combining each dof's
       torque_external_est after thresholding */
       for (i = 0; i < N; i++)
       {
              torque_external_est_thresh[i] = dof[i]->control_state->
                                              torque_external_est_LV_filt;
       }

       /* get estimated force vector by transforming estimated joint torque - stored
       in force_est input parameter */
       kin_dyn_fns->Translational_Jacobian_Transpose_Inverse(act_pos_prev,
                                                     torque_external_est_thresh,
                                                    force_est_thresh);

       /* save thresholded estimated force in Force_Estimation type variable
       (thresholded estimated moment not implemented) */
       for (i = 0; i < 3; i++)
       {
              force_estimation->force_est_thresh[i] = force_est_thresh[i];

       }
       /* ------------------- END EXTERNAL FORCE ESTIMATION ------------------- */

       /* read FTS, store value in force_moment vector passed in as parameter */
       /* NOTE: this is done after estimated force/moment and actual force/moment are
       stored so that we are comparing both the estimated and actual values from last
       cycle. */
       rc = ftsdrvr_ReadPort6DOF(FTS_PORT_NUMBER, &force_moment);
       if (rc != FTSDRVR__ERRCODE__NO_ERROR)
       {
              printf("ERROR: Could not read force/torque sensor properly.
                     Exiting. \n");
              return -1;
       }

       /* change the FTS's left handed system readings to right handed system
       readings */
       force_moment.force[1] *=  -1.;
       force_moment.moment[1] *= -1.;

       /* compensate for end-effector plate compression */
       force_moment.force[0] -=  EE_PLATE_COMPRESSION_FX;
       force_moment.force[1] -=  EE_PLATE_COMPRESSION_FY;
       force_moment.force[2] -=  EE_PLATE_COMPRESSION_FZ;
       force_moment.moment[0] -= EE_PLATE_COMPRESSION_MX;
       force_moment.moment[1] -= EE_PLATE_COMPRESSION_MY;
       force_moment.moment[2] -= EE_PLATE_COMPRESSION_MZ;

       /* transform forces from FTS frame to world frame (same as 0), compensates for
       end effector offsets on FTS, transforms 3 axis force to N axis force if N < 3
       (currently does not do the last step to moments)*/
       kin_dyn_fns->Force_Transform(act_pos, &force_moment);

       /* transform actual force/moment into actual external joint torque via
       transpose Jacobian */
       kin_dyn_fns->Translational_Jacobian_Transpose(act_pos, force_moment.force,
                                                     torque_external);
       for (i = 0; i < N; i++)
       {
              dof[i]->control_state->torque_external = torque_external[i];
       }

       /* threshold FTS readings */
       for (i = 0; i < 3; i++)
       {
              if (fabs(force_moment.force[i]) < FTS_FORCE_THRESHOLD)
              {
                     force_moment.force[i] = 0.;
              }
       }

       for (i = 0; i < N; i++)
       {
              /*calculate position and velocity errors*/
              error_pos[i] = dof[i]->control_state->act_pos - des_pos[i];
              error_vel[i] = dof[i]->control_state->act_vel - des_vel[i];

              /*s_DELTA is used in adaptive control laws*/
              if (learn_modeled_params || learn_unmodeled_params)
              {
                     Kp[i] = dof[i]->control_gains->Kp_LEARNING;
                     Kd[i] = dof[i]->control_gains->Kd_LEARNING;
              }
              else
              {
                     Kp[i] = dof[i]->control_gains->Kp_NOT_LEARNING;
                     Kd[i] = dof[i]->control_gains->Kd_NOT_LEARNING;
              }

              s[i] = error_vel[i] + (Kp[i]/Kd[i])*error_pos[i];

              des_vel_r[i] =        des_vel[i] - (Kp[i]/Kd[i])*error_pos[i];

              des_accel_r[i] =      des_accel[i] - (Kp[i]/Kd[i])*error_vel[i];


              s_DELTA[i] = s[i] - PHI*sat(s[i]/PHI);
              dof[i]->control_state->s_DELTA = s_DELTA[i];

              /*clear out these terms before they are recomputed */
              torque_dynamic[i] =    0.;
              torque_viscous[i] =  0.;
              torque_hinges[i] =   0.;
       }

       /* --------------------- COUPLED DYNAMICS ----------------- */
       /*torque due to modelled dynamics - friction not included */
       if (USE_DYNAMIC_MODEL)
       {
              kin_dyn_fns->Dynamic_Adaptive_Torque_N_DOF(act_pos, act_vel,

          des_vel_r, des_accel_r, s_DELTA, dt, learn_modeled_params, torque_dynamic);
       }
       /* --------------------- END COUPLED DYNAMICS ------------- */

       /* ------------------------ PD LAW ------------------------*/
```

```
for (i = 0; i < N; i++)
{
        /*PD torque - based strictly on error terms */
        dof[i]->control_state->torque_PD = -Kp[i]*error_pos[i]
                                          - Kd[i]*error_vel[i];
}
/* ------------------------ END PD LAW ---------------------*/


/* -------------------  DECOUPLED DYNAMICS ---------------- */
for (i = 0; i < N; i++)
{
        if (USE_VISCOUS_NN)

        {

                /* should this be des_vel_r ? */
                torque_viscous[i] =
                        kin_dyn_fns->Viscous_Friction_Torque(
                                        dof[i]->control_state->act_vel,
                                        s_DELTA[i], dt[i], dof[i],
                                        learn_unmodeled_params);
                if (torque_viscous[i] == FLT_MAX)
                {
                        return -1;
                }
        }
        if (USE_HINGES)
        {
                torque_hinges[i] = kin_dyn_fns->Hinges_Torque(
                                                dof[i]->control_state->act_vel,
                                                s_DELTA[i], dt[i], dof[i],
                                                learn_unmodeled_params);
        }
}
/* ------------------ END DECOUPLED DYNAMICS -------------- */

/* combine coupled and decoupled terms to get desired torque, then convert
that to desired voltage to be outputted */
for (i = 0; i < N; i++)
{
        dof[i]->control_state->torque_model = torque_dynamic[i] +
                                                torque_viscous[i] +
                                                torque_hinges[i];

        /*form the total torque that will be commanded by the motor drivers*/
        torque[i] = dof[i]->control_state->torque_model +
                     dof[i]->control_state->torque_PD;

        /* voltage limitation */
        volts_out = torque[i] *
                    dof[i]->motor_constants->CONV_TORQUE_OUT_TO_VOLTS;
        if (fabs(volts_out) > dof[i]->motor_constants->MAX_VOLTS_OUT_SOFT)
        {
                if (volts_out >= 0.)
                {
                        volts_out = (double) dof[i]->motor_constants->
                                                   MAX_VOLTS_OUT_SOFT;
                }
                else
                {
                        volts_out = (double ) -dof[i]->motor_constants->
                                                    MAX_VOLTS_OUT_SOFT;
                }

                printf("WARNING: Maxing out output voltage: %.2f V on DOF:
                        %s\n", volts_out,  dof[i]->motor_name);
        }

        volts_bits = comedi_from_phys(volts_out, output_cr, output_max_value);
        return_val_write = comedi_data_write(daq_device, AO_SUBDEVICE,
                                                dof[i]->motor_constants->AO_CHAN,
                                                AO_RANGE_0, AREF, volts_bits);
                if (return_val_write==-1)
                {
                        printf("ERROR writing to DAC.\n");
                        return -1;
                }
        }

        free(act_pos);
        free(act_pos_prev);
        free(act_vel);
        free(act_vel_prev);
        free(act_accel);
        free(act_accel_prev);
        free(des_vel_r);
        free(des_accel_r);
        free(error_pos);
        free(error_vel);
        free(pseudo_vel);
        free(Kp);
        free(Kd);
        free(s);
        free(s_DELTA);
        free(s_DELTA_prev);
        free(torque);
        free(torque_external);
        free(torque_external_est);
        free(torque_external_est_thresh);
        free(torque_dynamic);
        free(torque_viscous);
        free(torque_hinges);

        return 0;
}

/* return 1 and print error message if ptr equals NULL, 0 o/w
   char * parameter is name of variable - used in error message.
 */
int
equalsNULL(void * ptr, char * name)
{
        if (ptr == NULL)
        {
                printf("ERROR: %s == NULL -> calloc( ) failed. Exiting.\n", name);
                return 1;
        }

        return 0;
}

/* filter input signal based on FILTER_NUM, FILTER_DEN constants defined in main.h .
   NOTE: filter_num and filter_den are the same (respectively) as b and a in
             MATLAB's "filter" function. Note that the first element, '1',
             in a is not actually used in calculations but should be included. */
double
Filter_Digital_Signal(double signal_unfilt_new, double * signal_unfilt,
                        double * signal_filt)
{
        const double filter_num[FILTER_NUM_LENGTH] = {FILTER_NUM};
        const double filter_den[FILTER_DEN_LENGTH] = {FILTER_DEN};

        int i;

        /*add in newest unfiltered signal and update delayed versions*/
        for (i = (FILTER_NUM_LENGTH-1); i > 0; i--)
        {
                signal_unfilt[i] = signal_unfilt[i-1];
        }
        signal_unfilt[0] = signal_unfilt_new;
        /*update delayed versions of filtered signal */
        for (i = (FILTER_DEN_LENGTH-1); i > 0; i--)
        {
                signal_filt[i] = signal_filt[i-1];
        }

        /*calculate newest filtered velocity based on stored actual velocity and
          previous filtered velocities */
        signal_filt[0] = 0;
        for (i = 0; i< FILTER_NUM_LENGTH; i++)
        {
                signal_filt[0] += filter_num[i]*signal_unfilt[i];
        }
        for (i = 1; i< FILTER_DEN_LENGTH; i++)
        {
                signal_filt[0] -= filter_den[i]*signal_filt[i];
        }

        /* return latest filtered signal */
        return signal_filt[0];
}

/*      Function that does control cycle timing and data saving.
        Currently this function also generates desired (N DOF) trajectory in
        joint space. Calls Control_Law_N_DOF() each cycle where control law for the
        cycle is calculated and outputted.

        parameters:
        kin_dyn_fns     -       pointer to variable holding pointers to kinematics and
                                dynamics functions for given manipulator
        dof             -       array (of size N) of pointers to Single_DOF_Properties
                                variables holding information about each DOF
        NOTE: DOFs must be in correct order with those closest to base of the
              manipulator coming first. For example, starting at the base, the
              MORPHbot module has a pitch DOF followed by a roll DOF so this array
              would contain 2 elements - the Single_DOF_Properties of the pitch at
              index 0 and the Single_DOF_Properties of the roll at index 1.
        N               -       number of degrees of freedom to control
*/
void
Closed_Loop_Control_N_DOF(Kinematics_Dynamics_Functions     * kin_dyn_fns,
                          Force_Estimation                  * force_estimation,
                          Single_DOF_Properties             ** dof,
                          int                               N)
{
        int                         rc;
        int                         i;
        int                         j;
        int                         num_cycles;
        int                         num_cycles_param_evolution;
        int                         cycle_count;
        int                         cycle_count_param_evolution;
        int                         param_count;
        BOOLEAN                     learning_enabled;
        BOOLEAN                     learn_modeled;
        BOOLEAN                     learn_unmodeled;
        double                      cycle_timing;
        double                      run_time;
        double                      total_moving_to_des_pos_initial_time;
        double                      total_moving_to_des_pos_final_time;
        double                      moving_to_des_pos_accel;
        double                      time_i;
        double                      time_i_minus_one;
        double                      v_initial;
        double                      two_pi_f;
        double                      * param_lengths_array;
        double                      * param_array;
        double                      * des_pos_array;
        double                      * des_vel_array;
        double                      * des_accel_array;
        double                      * des_pos_mod_array;
        double                          * des_vel_mod_array;
        double                          * des_accel_mod_array;
        double                          * act_pos_array;
        double                          * act_vel_array;
        double                          * act_accel_array;
        double                          * torque_PD_array;
        double                          * torque_motor_array;
        double                          * torque_model_array;
        double                          * torque_ext_array;
        double                          * torque_ext_est_array;
        double                          * torque_ext_est_LV_filt_array;
        double                          temp[3];
        double                          freq_mult[4] = {0.};
        double                          amp_mult[4] =  {0.};
        double                          des_pos_cart[3];
        double                          des_vel_cart[3];
        double                          des_vel_cart_prev[3];
        double                          des_accel_cart[3];
        double                          des_pos_cart_mod[3];
        double                          des_vel_cart_mod[3];
        double                          des_accel_cart_mod[3];
        RclLeonControlData              * LCD_ptr;
        struct timespec                 start;
        struct timespec                 error;
        struct timespec                 current;
        struct timespec                 prev;
        struct timespec                 diff;
        struct timespec                 just_before_sleep;
        struct timespec                 rel_sleep_time;
        double                          * act_pos_initial =
        (double *) calloc(N, sizeof(double));
        assert(act_pos_initial != NULL);
        double  * des_pos =             (double *) calloc(N, sizeof(double));
        assert(des_pos != NULL);
        double  * des_vel =             (double *) calloc(N, sizeof(double));
        assert(des_vel != NULL);
        double  * des_accel =           (double *) calloc(N, sizeof(double));
        assert(des_accel != NULL);
        double  * des_pos_mod =         (double *) calloc(N, sizeof(double));
        assert(des_pos_mod != NULL);
        double  * des_vel_mod =         (double *) calloc(N, sizeof(double));
        assert(des_vel_mod != NULL);
        double * des_vel_mod_prev =     (double *) calloc(N, sizeof(double));
        assert(des_vel_mod_prev != NULL);
        double  * des_accel_mod =       (double *) calloc(N, sizeof(double));
        assert(des_accel_mod != NULL);
        double  * moving_to_des_pos_initial_time =    (double *) calloc(N,
                                                                      sizeof(double));
        assert(moving_to_des_pos_initial_time != NULL);
        double  * moving_to_des_pos_final_time = (double *) calloc(N, sizeof(double));
        assert(moving_to_des_pos_final_time != NULL);
        double  * pos_halfway_between_initial = (double *) calloc(N, sizeof(double));
        assert(pos_halfway_between_initial != NULL);
        double  * pos_halfway_between_final = (double *) calloc(N, sizeof(double));
        assert(pos_halfway_between_final != NULL);

        if (N <= 0)
        {
               printf("ERROR: Cannot have parameter N <= 0.\n");
               exit(-1);
        }

        printf("--- Starting ---\n");
        printf("TRAIN_SECONDS:\t\t%.0f\n",    TRAIN_SECONDS);
        printf("TEST_SECONDS:\t\t%.0f\n",     TEST_SECONDS);
        printf("SAVE_CONTROL_DATA:\t%d\n",    SAVE_CONTROL_DATA);
        printf("SAVE_PARAM_EVOLUTION:\t%d\n", SAVE_PARAM_EVOLUTION);
        printf("SAVE_LEARNED_PARAMS:\t%d\n",  SAVE_LEARNED_PARAMS);
        printf("LOAD_LEARNED_PARAMS:\t%d\n",  LOAD_LEARNED_PARAMS);
        printf("ESTIMATION_IMPEDANCE:\t%d\n", ESTIMATION_IMPEDANCE);
        printf("FTS_IMPEDANCE:\t\t%d\n",      FTS_IMPEDANCE);
        printf("STOP_AFTER_TRAINING:\t%d\n",  STOP_AFTER_TRAINING);
        printf("USE_DYNAMIC_MODEL:\t%d\n",    USE_DYNAMIC_MODEL);
        printf("USE_VISCOUS_NN:\t\t%d\n",     USE_VISCOUS_NN);
        printf("USE_HINGES:\t\t%d\n",         USE_HINGES);
        printf("----------------\n");

        if (LOAD_LEARNED_PARAMS)
        {
                rc = load_learned_parameters(dof, N);
                if (rc != 0)
                {
                       printf("ERROR: Failed to load parameters from previous run.
                               Exiting.\n");
                       exit(-1);
                }
        }

        /* useful calculation for trajectory generation */
        two_pi_f = 2.*PI*FREQUENCY_TRAINING;

        /* calculate number of cycles between saving data given FREQ_SAVING */
        num_cycles = (int) (FREQ_SYSTEM/FREQ_SAVING);
        cycle_count = 0;

        /* calculate number of cycles between saving data given
           FREQ_SAVING_PARAM_EVOLUTION */
        num_cycles_param_evolution = (int) (FREQ_SYSTEM/FREQ_SAVING_PARAM_EVOLUTION);
        cycle_count_param_evolution = 0;

        /* get the actual initial position */
        for (i = 0; i < N; i++)
        {
                /* dof[i]->control_state->counter_val should be 0 before this call */
                dof[i]->control_state->counter_val =
                       ComediReadCounterWithRollover(daq_device, CTR_SUBDEVICE,
                                            dof[i]->motor_constants->CTR_CHAN,
                                             dof[i]->control_state->counter_val);
                dof[i]->control_state->counter_val_prev =
                                             dof[i]->control_state->counter_val;
                dof[i]->control_state->act_pos =
                                             dof[i]->control_state->counter_val *
                                             dof[i]->motor_constants->
                                                     CONV_COUNTS_TO_RADIANS;
                dof[i]->control_state->act_pos_prev =
                                             dof[i]->control_state->act_pos;
                act_pos_initial[i] = dof[i]->control_state->act_pos;
                des_pos[i] = act_pos_initial[i];
        }

        /* convert initial desired joint trajectory to initial desired Cartesian
           trajectory */
        kin_dyn_fns->Forward_Kinematics(des_pos, des_pos_cart);

        /* calculate time to move from actual initial position to desired initial
           position for each DOF*/
        total_moving_to_des_pos_initial_time = 0.;
        for (i = 0; i < N; i++)
        {
                pos_halfway_between_initial[i] = (dof[i]->des_pos_initial -
                                                  act_pos_initial[i])/2.;
                moving_to_des_pos_initial_time[i] = 2. *
                sqrt(2.*fabs(pos_halfway_between_initial[i])/MOVING_TO_DES_POS_ACCEL);
                total_moving_to_des_pos_initial_time +=
                               moving_to_des_pos_initial_time[i];
        }

        /* calculate time to move from desired initial position to desired final
           position for each DOF*/
        total_moving_to_des_pos_final_time = 0.;
        for (i = 0; i < N; i++)
        {
                pos_halfway_between_final[i] = (dof[i]->des_pos_final -
                                                dof[i]->des_pos_initial)/2.;
                moving_to_des_pos_final_time[i] = 2. *
                       sqrt(2.*fabs(pos_halfway_between_final[i])/
                                             MOVING_TO_DES_POS_ACCEL);
                total_moving_to_des_pos_final_time += moving_to_des_pos_final_time[i];
        }

        /* set start time */
        rc = clock_gettime(CLOCK_REALTIME, &start);
        assert(rc == 0 && "Failed clock_gettime(start)");

        prev = start;
        run_time = 0;

        /* intialize time stamps */
        for (i = 0; i < N; i++)
        {
                dof[i]->control_state->time_stamp_ts = start;
                dof[i]->control_state->time_stamp_prev_ts = start;
        }

        /* set desired start of next loop */
        start.tv_nsec += ((long) PERIOD_SYSTEM__MICROSECS) * 1000L;
        if (start.tv_nsec > NANOSECONDS_PER_SEC)
        {
                start.tv_nsec -= NANOSECONDS_PER_SEC;
                start.tv_sec += 1;
        }

        /* --- relative sleeping --- */
        rc = clock_gettime(CLOCK_REALTIME, &just_before_sleep);
        if (rc != 0)
        {
                printf("ERROR at %f s: clock_gettime() failed.\n", run_time);
        }
        rc = rclDiffTimespecs(&start, &just_before_sleep, &rel_sleep_time);
        if (rc != 0)
        {
                printf("ERROR at %f s: rclDiffTimeSpecs() failed.\n", run_time);
        }
        rc = nanosleep(&rel_sleep_time, &error);

        /* --- absolute sleeping --- */
        /*rc = clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &start, &error); */

        if (rc != 0 && !shouldQuit)
        {
                /* when we chose to quit, it throws an error. Presume that signal
                interrupts nanosleep() Dec 2005, S Roderick */
                assert(rc == 0 && "Failed sleeping.");
        }

        /* get initial time */
        rc = clock_gettime(CLOCK_REALTIME, &initial);
        assert(rc == 0 && "Failed clock_gettime(initial)");

        learning_enabled =            FALSE;
        learn_modeled =                       FALSE;
        learn_unmodeled =             FALSE;

        while (!shouldQuit)
        {
                /* get clock time coming out of sleep */
                rc = clock_gettime(CLOCK_REALTIME, &current);
                if (rc != 0)
                {
                       printf("ERROR at %f s: clock_gettime() failed.\n", run_time);
                }

                /*keep track of time relative to initial time */
```

```
rc = rclDiffTimespecs(&current, &initial, &diff);
if (rc != 0)
{
       printf("ERROR at %f s: rclDiffTimeSpecs() failed -
               (current - initial).\n", run_time);
}
run_time = ((double) rclTimespecToMicroseconds(&diff))*1e-6;

/*get time difference between current cycle start and
  previous cycle start */
rc = rclDiffTimespecs(&current, &prev, &diff);
if (rc != 0)
{
       printf("ERROR at %f s: rclDiffTimeSpecs() failed - (current -
               prev).\n", run_time);
}
prev = current;

/*leave the control loop if blown deadline*/
cycle_timing = ((double) rclTimespecToMicroseconds(&diff))*1e-6;
if ((cycle_timing > (PERIOD_SYSTEM__MICROSECS*1e-6 +
                     TIMING_ERROR__MICROSECS*1e-6)) ||
    (cycle_timing < (PERIOD_SYSTEM__MICROSECS*1e-6 -
                     TIMING_ERROR__MICROSECS*1e-6)))
{
       printf("ERROR at %f seconds: blown deadline. Cycle time was:
               %f us instead of: %f us.\n",
               run_time, (cycle_timing*1e6), PERIOD_SYSTEM__MICROSECS);
       break;
}

/* ================= generate desired trajectory ================= */
if (run_time <= total_moving_to_des_pos_initial_time)
                              /* GO TO DESIRED INITIAL POSITION */
{

       /* Disable learning of dynamics during this part of the
                                                     trajectory. */
       learning_enabled =            FALSE;
       learn_modeled =               FALSE;
       learn_unmodeled =             FALSE;

       /* set times initial to beginning of this trajectory phase */
       time_i_minus_one =    0.;
       time_i =                      0.;

       for(i = 0; i < N; i++)
       {
              if (pos_halfway_between_initial[i] < 0.)
              {
                     moving_to_des_pos_accel =
                            - MOVING_TO_DES_POS_ACCEL;
              }
              else
              {
                     moving_to_des_pos_accel =
                             MOVING_TO_DES_POS_ACCEL;

              }

              time_i_minus_one = time_i;
              time_i += moving_to_des_pos_initial_time[i];

              /* check to see whether DOF i's trajectory should be
                 changed */
              if (run_time >= time_i_minus_one && run_time < time_i)
              {
                     /*ramp up velocity, followed by ramp down - this
                     is used so that there is no jump discontinuity
                     in the velocity signal*/
                     if (run_time <= (time_i_minus_one +
                     moving_to_des_pos_initial_time[i]/2.))
                     /* ramp up part */
                     {
                             des_pos[i] =
                                     act_pos_initial[i] +
                                     0.5*moving_to_des_pos_accel*
                                             pow(run_time
                                             - time_i_minus_one, 2.);
                             des_vel[i] =
                                     moving_to_des_pos_accel*
                                     (run_time - time_i_minus_one);
                             des_accel[i] = moving_to_des_pos_accel;

                     }
                     else    /* ramp down part */
                     {
                             v_initial =
                                     moving_to_des_pos_accel*
                             (moving_to_des_pos_initial_time[i]/2.);

       des_pos[i] =    act_pos_initial[i] +
               pos_halfway_between_initial[i] +
               v_initial*(run_time - (time_i_minus_one +
               moving_to_des_pos_initial_time[i]/2.))-
              0.5*moving_to_des_pos_accel*
              pow(run_time - (time_i_minus_one +
              moving_to_des_pos_initial_time[i]/2.), 2.);
       des_vel[i] =    v_initial - moving_to_des_pos_accel*(run_time -
                     (time_i_minus_one +
                     moving_to_des_pos_initial_time[i]/2.));
       des_accel[i] = -moving_to_des_pos_accel;
                     }
              }
              else
              {
                     /* maintain fixed position */
                     des_vel[i] = 0.;
                     des_accel[i] = 0.;
              }
       }
}
else if (run_time <= (TRAIN_SECONDS +
                     total_moving_to_des_pos_initial_time))
                                     /* EXECUTE TRAINING TRAJECTORY */

{
       /* Enable the learning of dynamics during this part of the
           trajectory */
       if (learning_enabled == FALSE)
       {
              learning_enabled = TRUE;
              printf("Learning ON.\n");

       }
       learn_modeled =                 TRUE;
       learn_unmodeled =       TRUE;

       for (i = 0; i < N; i++)
       {
              if (i == 0)
              {
                     freq_mult[0] = 1.;
                     amp_mult[0] =  .8;              /* 1. */
                     freq_mult[1]=  4.;
                     amp_mult[1] =  .2;              /* .3 */
              }
              else
              {
                     freq_mult[0] = 2.5;     /* 1.5 */
                     amp_mult[0] =  .5;
                     freq_mult[1] = 0.;
                     amp_mult[1] =  0.;

              }

              /* LOW VELOCITY FRICTION LEARNING */
              if (run_time >= (SWITCH_TRAJ_PERIODS/FREQUENCY_TRAINING
                             + total_moving_to_des_pos_initial_time))
              {
                     learn_modeled =               FALSE;
                     learn_unmodeled =      TRUE;

                     freq_mult[0] = 1.;
                     amp_mult[0] =  1.;
                     if (i == 1)
                     {
                             amp_mult[0] =  .5;
                     }
                     else
                     freq_mult[1]=  0.;
                     amp_mult[1] =  0.;

              }

              des_pos[i] = dof[i]->des_pos_initial;
              des_vel[i] = 0.;
              des_accel[i] = 0.;
              for (j = 0; j < 4; j++)
              {
                     des_pos[i] += (amp_mult[j]*AMPLITUDE_TRAINING)
                             *cos(freq_mult[j]*two_pi_f*(run_time -
                             total_moving_to_des_pos_initial_time))
                             - (amp_mult[j]*AMPLITUDE_TRAINING);
                     des_vel[i] += -1*
                             (amp_mult[j]*AMPLITUDE_TRAINING)*
                             (freq_mult[j]*two_pi_f)
                             *sin(freq_mult[j]*two_pi_f*
                             (run_time -
                             total_moving_to_des_pos_initial_time));
                     des_accel[i] += -1.*
                             (amp_mult[j]*AMPLITUDE_TRAINING)*
                             (freq_mult[j]*two_pi_f)*(freq_mult[j]*two
                             _pi_f)
                             *cos(freq_mult[j]*two_pi_f*
                             (run_time -
                             total_moving_to_des_pos_initial_time));
              }
       }
}
else if (run_time <= (TRAIN_SECONDS +
         total_moving_to_des_pos_initial_time +
          total_moving_to_des_pos_final_time))
                                     /* GO TO DESIRED FINAL POSITION */
{
/* Disable learning of dynamics during this part of the trajectory. */
       if (learning_enabled == TRUE)
       {
              learning_enabled = FALSE;
              printf("Learning OFF.\n");
       }
       learn_modeled =               FALSE;
       learn_unmodeled =             FALSE;

       /* set times initial to beginning of this trajectory phase */
       time_i_minus_one =     TRAIN_SECONDS +
                     total_moving_to_des_pos_initial_time;
       time_i =               TRAIN_SECONDS +
                     total_moving_to_des_pos_initial_time;

       for(i = 0; i < N; i++)
       {
              if (pos_halfway_between_final[i] < 0.)
              {
                     moving_to_des_pos_accel = -
                                     MOVING_TO_DES_POS_ACCEL;
              }
              else
              {
                     moving_to_des_pos_accel =
                                     MOVING_TO_DES_POS_ACCEL;

              }

              time_i_minus_one = time_i;
              time_i += moving_to_des_pos_final_time[i];

              /* check to see whether DOF i's trajectory should be
                 changed */
              if (run_time >= time_i_minus_one && run_time < time_i)
              {
                     /*ramp up velocity, followed by ramp down - this
                     is used so that there is no jump discontinuity
                     in the velocity signal*/
                     if (run_time <= (time_i_minus_one +
                     moving_to_des_pos_final_time[i]/2.))
                                                     /* ramp up part */
                     {
                             des_pos[i] =   dof[i]->des_pos_initial +
                     0.5*moving_to_des_pos_accel*pow(run_time -
                                             time_i_minus_one, 2.);
                             des_vel[i] =
                             moving_to_des_pos_accel*(run_time -
                                              time_i_minus_one);
                             des_accel[i] = moving_to_des_pos_accel;

                     }
                     else    /* ramp down part */
                     {
                             v_initial =
                             moving_to_des_pos_accel*
                             (moving_to_des_pos_final_time[i]/2.);

                     des_pos[i] =   dof[i]->des_pos_initial +
                             pos_halfway_between_final[i]
                                     + v_initial*(run_time -
                             (time_i_minus_one +
                             moving_to_des_pos_final_time[i]/2.))
                             - 0.5*moving_to_des_pos_accel*
                             pow(run_time -
                             (time_i_minus_one +
                             moving_to_des_pos_final_time[i]/2.), 2.);

                     des_vel[i] =
                             v_initial
                             - moving_to_des_pos_accel*(run_time -
                             (time_i_minus_one +
                             moving_to_des_pos_final_time[i]/2.));
                     des_accel[i] = -moving_to_des_pos_accel;
                     }
              }
              else
              {
                     /* maintain fixed position */
                     des_vel[i] = 0.;
                     des_accel[i] = 0.;
              }
       }
}
else

/* EXECUTE TESTING TRAJECTORY */
{
       /* Disable learning of dynamics during this part of the
```

```
           trajectory */
        if (learning_enabled == TRUE)
        {
                learning_enabled = FALSE;
                printf("Learning OFF.\n");
        }
        learn_modeled =               FALSE;
        learn_unmodeled =             FALSE;

        for (i = 0; i < N; i++)
        {
                if (STOP_AFTER_TRAINING)
                {
                /* ---------- maintain fixed position --------- */
                        des_vel[i] =   0;
                        des_accel[i] = 0;
                }
                else
                {

                /*      -------- generate sinusoidal testing trajectory
                                   for all joints ------------ */

                        des_pos[i] =
                                AMPLITUDE_TESTING*
                                cos(2.*PI*FREQUENCY_TESTING*
                                    (run_time - (TRAIN_SECONDS +
                                    total_moving_to_des_pos_initial_time
                                    +total_moving_to_des_pos_final_time))
                                    )
                                + (dof[i]->des_pos_final -
                                                AMPLITUDE_TESTING);
                        des_vel[i] =
                                -2.*PI*FREQUENCY_TESTING*
                                    AMPLITUDE_TESTING *sin(2.*
                                PI*FREQUENCY_TESTING*(run_time -
                                (TRAIN_SECONDS +
                                total_moving_to_des_pos_initial_time +
                                 total_moving_to_des_pos_final_time)));
                        des_accel[i] = -2.*PI*FREQUENCY_TESTING*
                                2.*PI*FREQUENCY_TESTING*AMPLITUDE_TESTING
                                *cos(2.*PI*FREQUENCY_TESTING*(run_time -
                                (TRAIN_SECONDS +
                                total_moving_to_des_pos_initial_time +
                                total_moving_to_des_pos_final_time)));
                }
        }
}

/* before updating, save current desired position as previous */
for (i = 0; i < 3; i++)
{
        des_vel_cart_prev[i] = des_vel_cart[i];
}

/* convert desired joint trajectory to desired Cartesian trajectory */
kin_dyn_fns->Forward_Kinematics(des_pos, des_pos_cart);
kin_dyn_fns->Translational_Jacobian(des_pos, des_vel, des_vel_cart);
for (i = 0; i < 3; i++)
{
        des_accel_cart[i] = (des_vel_cart[i] -
                des_vel_cart_prev[i])/(PERIOD_SYSTEM__MICROSECS*1e-6);
}

/* ======== modify desired trajectory based on impedance model
========= */
if ((ESTIMATION_IMPEDANCE || FTS_IMPEDANCE) &&
        run_time > (TRAIN_SECONDS +
                total_moving_to_des_pos_initial_time +
                total_moving_to_des_pos_final_time))
{
```

```
                /* apply impedance model to modify desired Cartesian trajectory */
                        for (i = 0; i < 3; i++)
                        {
                                temp[i] = Cs*(des_vel_cart_mod[i] - des_vel_cart[i]) +
                                        Ks*(des_pos_cart_mod[i] - des_pos_cart[i]);

                                /* get modified desired acceleration based on impedance
                                rule */
                                if (ESTIMATION_IMPEDANCE)
                                {
                                        des_accel_cart_mod[i] =
                                                (Kf*force_estimation->force_est_thresh[i]
                                                - temp[i])/Ms + des_accel_cart[i];
                                }
                                else if (FTS_IMPEDANCE)
                                {
                                        des_accel_cart_mod[i] =
                                                (Kf*force_estimation->force[i]
                                                 - temp[i])/Ms + des_accel_cart[i];
                                }

                /* numerically integrate acceleration to get position, velocity */
                                des_pos_cart_mod[i] +=
                                        (PERIOD_SYSTEM__MICROSECS*1e-6)*
                                        des_vel_cart_mod[i]
                                        + .5*pow((PERIOD_SYSTEM__MICROSECS*1e-6), 2)*
                                        des_accel_cart_mod[i];
                                des_vel_cart_mod[i] += (PERIOD_SYSTEM__MICROSECS*1e-6)*
                                        des_accel_cart_mod[i];
                        }

                        /* before updating, save current modified desired joint
                           position as previous */
                        for (i = 0; i < N; i++)
                        {
                                des_vel_mod_prev[i] = des_vel_mod[i];
                        }

                        /* convert modified desired Cartesian trajectory back to joint
                                space */
                        kin_dyn_fns->Translational_Jacobian_Inverse(des_pos,
                                                des_vel_cart_mod, des_vel_mod);
                        for (i = 0; i < N; i++)
                        {
                                des_pos_mod[i] +=  (PERIOD_SYSTEM__MICROSECS*1e-6) * .5
                                                   * (des_vel_mod[i] +
                                                      des_vel_mod_prev[i]);
                                des_accel_mod[i] = (des_vel_mod[i] -
                                                      des_vel_mod_prev[i])/(
                                                      PERIOD_SYSTEM__MICROSECS*1e-6);
                        }
                }
                else
                {
                        /*desired Cartesian position and velocity not modified */
                        for (i = 0; i < N; i++)
                        {
                                des_pos_mod[i] =      des_pos[i];
                                des_vel_mod[i] =      des_vel[i];
                                des_accel_mod[i] =    des_accel[i];
                        }
                }

                /* ================= call control code for this cycle =================
*/
                rc = Control_Law_N_DOF(des_pos_mod, des_vel_mod, des_accel_mod,
                                        learn_modeled, learn_unmodeled, kin_dyn_fns,
                                        force_estimation, dof, N);
                if (rc != 0)
                {
                        printf("ERROR: Control_Cycle() failed.\n");
```

```
        break;
}

/* leave the control loop if at or past desired running time */
if (run_time >= (RUN_SECONDS + total_moving_to_des_pos_initial_time +
                 total_moving_to_des_pos_final_time))
{
        break;
}

/* count number of cycles since last time cycle_count was zeroed. see
   if statement below for zeroing */
/* making cycle_count increment after
   total_moving_to_des_pos_initial_time ensures that we don't log
   anything before that */
if (run_time > total_moving_to_des_pos_initial_time)
{
        cycle_count++;
        cycle_count_param_evolution++;
}

/* dof variable's data gets changed in Control_Law_N_DOF() - push new
   data into queue */
if (SAVE_CONTROL_DATA && run_time >total_moving_to_des_pos_initial_time
                       && cycle_count == num_cycles)
{
        /*dynamically allocate memory for saving control information in
          RclLeonControlData structure*/
        /*NOTE: array must be freed wherever RclLeonControlData pointer
          is popped off the queue */
        LCD_ptr =
        (RclLeonControlData *) malloc(sizeof(RclLeonControlData));
        assert(LCD_ptr != NULL);
        des_pos_array =(double *) malloc(N*sizeof(double));
        assert(des_pos_array != NULL);
        des_vel_array =(double *) malloc(N*sizeof(double));
        assert(des_vel_array != NULL);
        des_accel_array =(double *) malloc(N*sizeof(double));
        assert(des_accel_array != NULL);
        des_pos_mod_array =(double *) malloc(N*sizeof(double));
        assert(des_pos_mod_array != NULL);
        des_vel_mod_array =(double *) malloc(N*sizeof(double));
        assert(des_vel_mod_array != NULL);
        des_accel_mod_array =  (double *) malloc(N*sizeof(double));
        assert(des_accel_mod_array != NULL);
        act_pos_array =(double *) malloc(N*sizeof(double));
        assert(act_pos_array != NULL);
        act_vel_array =(double *) malloc(N*sizeof(double));
        assert(act_vel_array != NULL);
        act_accel_array =(double *) malloc(N*sizeof(double));
        assert(act_accel_array != NULL);
        torque_PD_array =(double *) malloc(N*sizeof(double));
        assert(torque_PD_array != NULL);
        torque_motor_array =   (double *) malloc(N*sizeof(double));
        assert(torque_motor_array != NULL);
        torque_model_array =(double *) malloc(N*sizeof(double));
        assert(torque_model_array != NULL);
        torque_ext_array =     (double *) malloc(N*sizeof(double));
        assert(torque_ext_array != NULL);
        torque_ext_est_array = (double *) malloc(N*sizeof(double));
        assert(torque_ext_est_array != NULL);
        torque_ext_est_LV_filt_array =(double *)
                              malloc(N*sizeof(double));
        assert(torque_ext_est_LV_filt_array != NULL);

        /* save control data in newly allocated arrays */
        LCD_ptr->des_pos_array = des_pos_array;
        LCD_ptr->des_vel_array = des_vel_array;
        LCD_ptr->des_accel_array = des_accel_array;
        LCD_ptr->des_pos_mod_array =  des_pos_mod_array;
        LCD_ptr->des_vel_mod_array =  des_vel_mod_array;
```

```
    LCD_ptr->des_accel_mod_array =des_accel_mod_array;
    LCD_ptr->act_pos_array = act_pos_array;
    LCD_ptr->act_vel_array = act_vel_array;
    LCD_ptr->act_accel_array = act_accel_array;
    LCD_ptr->torque_PD_array = torque_PD_array;
    LCD_ptr->torque_motor_array = torque_motor_array;
    LCD_ptr->torque_model_array = torque_model_array;
    LCD_ptr->torque_ext_array =   torque_ext_array;
    LCD_ptr->torque_ext_est_array = torque_ext_est_array;
    LCD_ptr->torque_ext_est_LV_filt_array =
                              torque_ext_est_LV_filt_array;

for (i = 0; i < N; i++)
    {
           LCD_ptr->des_pos_array[i] =          des_pos[i];
           LCD_ptr->des_vel_array[i] =          des_vel[i];
           LCD_ptr->des_accel_array[i] =        des_accel[i];
           LCD_ptr->des_pos_mod_array[i] =      des_pos_mod[i];
           LCD_ptr->des_vel_mod_array[i] =      des_vel_mod[i];
           LCD_ptr->des_accel_mod_array[i] =    des_accel_mod[i];
           LCD_ptr->act_pos_array[i] =          dof[i]->
                                                control_state->
                                                act_pos;
           LCD_ptr->act_vel_array[i] =          dof[i]->
                                                control_state->
                                                act_vel;
           LCD_ptr->act_accel_array[i] =        dof[i]->
                                                control_state->
                                                act_accel;
           LCD_ptr->torque_PD_array[i] =        dof[i]->
                                                control_state->
                                                torque_PD;
           LCD_ptr->torque_motor_array[i] =     dof[i]->
                                                control_state->
                                                torque_motor;
           LCD_ptr->torque_model_array[i] =     dof[i]->
                                                control_state->
                                                torque_model;
           LCD_ptr->torque_ext_array[i] =       dof[i]->
                                                control_state->
                                                torque_external;
           LCD_ptr->torque_ext_est_array[i] =   dof[i]->
                                                control_state->
                                               torque_external_est;
           LCD_ptr->torque_ext_est_LV_filt_array[i] =   dof[i]->
                                                control_state->
                                      torque_external_est_LV_filt;
    }
    for (i = 0; i < 3; i++)
    {
           LCD_ptr->force[i] =force_estimation->force[i];
           LCD_ptr->moment[i] =   force_estimation->moment[i];
           LCD_ptr->force_est[i] = force_estimation->force_est[i];
           LCD_ptr->force_est_thresh[i] = force_estimation->
                                         force_est_thresh[i];
    }
    LCD_ptr->time_stamp = dof[0]->control_state->time_stamp;

    /* push data onto queue*/
    if (!rclIsFullLeonControlDataPtrQueue(&g_queue))
    {
           rc = rclPushLeonControlDataPtrQueue(&g_queue, LCD_ptr);
           if (rc != 0)
           {
                  printf("ERROR pushing data to queue.\n");
           }
    }
    else
    {
           printf("ERROR pushing data to queue - queue is
                  full.\n");
```

```
        }

        cycle_count = 0;
}

/* dof variable's data gets changed in Control_Law_N_DOF() - push new
   data into queue */
if (SAVE_PARAM_EVOLUTION && run_time >
                            total_moving_to_des_pos_initial_time
         && cycle_count_param_evolution == num_cycles_param_evolution)
{
        /*dynamically allocate memory for saving control information in
          RclLeonControlData structure*/
        /*NOTE: array must be freed wherever RclLeonControlData pointer
                is popped off the queue */
        LCD_ptr = (RclLeonControlData *)
                       malloc(sizeof(RclLeonControlData));
        assert(LCD_ptr != NULL);

        /* param_lengths_array will contain: N,
                                            M,
                                            DOF 1 NUM_NODES_VISCOUS,
                                            DOF 2 NUM_NODES_VISCOUS, ...
        */
        param_lengths_array = (double *) malloc((2 + N)*
                                             sizeof(double));
        assert(param_lengths_array != NULL);
        param_lengths_array[0] =      N;
        param_lengths_array[1] =      M;
        for (i = 0; i < N; i++)
        {
                param_lengths_array[2 + i] = dof[i]->
                                 friction_parameters->NUM_NODES_VISCOUS;
        }

        /* count the number of parameters */
        param_count = M;
        for (i = 0; i < N; i++)
        {
                param_count += dof[i]->friction_parameters->
                                            NUM_NODES_VISCOUS;
        }

        /* params will contain: M modelled dynamics' adapted
           parameters, DOF 1 viscous NN parameters,
                       DOF 2 viscous NN parameters, ... */
        param_array =  (double *) malloc(param_count*sizeof(double));
        assert(param_array != NULL);

        /* save the modelled dynamics' adapted parameters */
        for (i = 0; i < M; i++)
        {
                param_array[i] = a_hat[i];
        }

        /*save viscous friction NN parameters */
        param_count = M;
        for (i = 0; i < N; i++)
        {
                for (j = 0; j < dof[i]->friction_parameters->
                                      NUM_NODES_VISCOUS; j++)
                {
                        param_array[param_count] = dof[i]->
                               friction_parameters->c_hat_VISCOUS[j];
                        param_count++;
                }
        }

        /* store arrays in RclLeonControlData type pointer */
        LCD_ptr->des_pos_array =             param_lengths_array;
        LCD_ptr->des_vel_array =             param_array;
```

```
                LCD_ptr->time_stamp =  dof[0]->control_state->time_stamp;


                /* push data onto queue*/
                if (!rclIsFullLeonControlDataPtrQueue(
                                                &g_queue_param_evolution))
                {
                        rc = rclPushLeonControlDataPtrQueue(
                                        &g_queue_param_evolution, LCD_ptr);
                        if (rc != 0)
                        {
                                printf("ERROR pushing data to parameter
                                        evolution queue.\n");
                        }
                }
                else
                {
                        printf("ERROR pushing data to parameter evolution queue
                                - queue is full.\n");
                }

                cycle_count_param_evolution = 0;
        }

        /* set desired start of next loop */
        start.tv_nsec += ((long) PERIOD_SYSTEM__MICROSECS) * 1000L;
        if (start.tv_nsec > NANOSECONDS_PER_SEC)
        {
                start.tv_nsec -= NANOSECONDS_PER_SEC;
                start.tv_sec += 1;
        }
        /* --- relative sleeping --- */
        rc = clock_gettime(CLOCK_REALTIME, &just_before_sleep);
        if (rc != 0)
        {
                printf("ERROR at %f s: clock_gettime() failed.\n", run_time);
        }
        rc = rclDiffTimespecs(&start, &just_before_sleep, &rel_sleep_time);
        if (rc != 0)
        {
                printf("ERROR at %f s: rclDiffTimeSpecs() failed - (start -
                        just_before_sleep).\n", run_time);
        }
        rc = nanosleep(&rel_sleep_time, &error);

        /* --- absolute sleeping --- */
        /*rc = clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &start, &error);
        */

        if (rc != 0 && !shouldQuit)
        {
                /* when we chose to quit, it throws an error. Presume that
                signal interrupts nanosleep() Dec 2005, S Roderick */
                assert(rc == 0 && "Failed sleeping.");
        }

}  /*************************** END WHILE *********************************/

int volts_bits = comedi_from_phys(0, output_cr, output_max_value);
comedi_data_write(daq_device, AO_SUBDEVICE, AO_CHAN_0, AO_RANGE_0, AREF,
                  volts_bits);
comedi_data_write(daq_device, AO_SUBDEVICE, AO_CHAN_1, AO_RANGE_0, AREF,
                  volts_bits);

/* save parameters of NN, gravitational and inertial terms that were learned
   online */
if (SAVE_LEARNED_PARAMS)
{
        rc = save_learned_parameters(dof, N);
        if (rc != 0)
        {
```

```
                        printf("ERROR: Failed to save parameters from previous run.
                                Exiting.\n");
                        exit(-1);
                }
        }

        free(act_pos_initial);
        free(des_pos);
        free(des_vel);
        free(des_accel);
        free(des_pos_mod);
        free(des_vel_mod);
        free(des_vel_mod_prev);
        free(des_accel_mod);
        free(moving_to_des_pos_initial_time);
        free(moving_to_des_pos_final_time);
        free(pos_halfway_between_initial);
        free(pos_halfway_between_final);
}

/*saturation function - return x if |x| < 1, otherwise return sign of x*/
double
sat(double x)
{
        if (fabs(x) < 1)
        {
                return x;
        }
        else if (x < 0)
        {
                return -1;
        }
        else return 1;
}

/*save parameters of NN, gravity and inertia that were learned online in a separate
file*/
int
save_learned_parameters(Single_DOF_Properties        ** dof,
                                             int
        N)
{
        int fclose_return_val, i, j;

        for (i = 0; i < N; i++)
        {
                FILE* out_params = fopen(dof[i]->friction_parameters->filename_params,
                                         "w");
                if (out_params == NULL)
                {
                        printf("ERROR: failed to open file %s. parameter data will not
                                be saved properly.\n",

                        dof[i]->friction_parameters->filename_params);
                        return -1;
                }
                else
                {
                        /* save the modelled dynamics' adapted parameters */
                        for (j = 0; j < M; j++)
                        {
                                fprintf(out_params, "%lf\n", a_hat[j]);
                                /* a_hat is extern - allocated in
                                   KinematicsDynamicsLib.c */
                        }

                        /* save the number of viscous NN nodes */
                        fprintf(out_params, "%d\n", dof[i]->friction_parameters->
                                                    NUM_NODES_VISCOUS);

                        /*save viscous friction NN parameters */
                        for (j = 0; j < dof[i]->friction_parameters->NUM_NODES_VISCOUS;
                             j++)
                        {
                                fprintf(out_params, "%lf\n", dof[i]->
                                        friction_parameters->c_hat_VISCOUS[j]);
                        }

                        /*save hinges parameters */
                        fprintf(out_params, "%lf\n", dof[i]->friction_parameters->
                                                                B_pos_HINGES);
                        fprintf(out_params, "%lf\n", dof[i]->friction_parameters->
                                                                B_neg_HINGES);
                }

                fclose_return_val = fclose(out_params);
                if (fclose_return_val != 0)
                {
                        printf("ERROR: failed to close parameters file %s.\n", dof[i]->
                                                  friction_parameters->filename_params);
                        return -1;
                }
        }

        return 0;
}

/*load parameters of NN and adaptive components learned from previous runs */
int
load_learned_parameters(Single_DOF_Properties       ** dof,
                                             int
        N)
{
        double  * parameters;
        int            fclose_return_val, i, j, current, num_params;

        for (i = 0; i < N; i++)
        {
                FILE* in_params = fopen(dof[i]->friction_parameters->filename_params,
                                                                        "r");
                if (in_params == NULL)
                {
                        printf("ERROR: failed to open file %s. parameter data will not
                                be loaded properly.\n",
                        dof[i]->friction_parameters->filename_params);
                        return -1;
                }

                num_params = M + 1 + dof[i]->friction_parameters->NUM_NODES_VISCOUS +
                             2;

                /*dynamically allocate parameters vector */
                parameters = (double *) calloc(num_params, sizeof(double));
                assert(parameters != NULL);

                /* read the parameters */
                for (j = 0; j < num_params; j++)
                {
                        /* loop through and store the numbers into the array */
                        fscanf(in_params, "%lf", &parameters[j]);
                }

                current = 0;

                /* load the modelled dynamics' adapted parameters */
                for (j = 0; j < M; j++)
                {
                        a_hat[j] = parameters[current];
                        current++;
                }

                /* read in number of viscous NN nodes */
                /*dof[i]->friction_parameters->NUM_NODES_VISCOUS = (int)
                  parameters[current]; */     /* assumed to be known */
                current++;

                /* read in parameters of viscous friction NN */
                for (j = 0; j < dof[i]->friction_parameters->NUM_NODES_VISCOUS; j++)
                {
                        dof[i]->friction_parameters->c_hat_VISCOUS[j] =
                                                        parameters[current];
                        current++;
                }

                /* read in parameters of hinges */
                dof[i]->friction_parameters->B_pos_HINGES = parameters[current];
                current++;
                dof[i]->friction_parameters->B_neg_HINGES = parameters[current];
                current++;

                free(parameters);

                /* close parameters file */
                fclose_return_val = fclose(in_params);
                if (fclose_return_val != 0)
                {
                        printf("ERROR: failed to close parameters file %s.\n", dof[i]->
                                               friction_parameters->filename_params);
                        return -1;
                }

                printf("DOF: %s parameters loaded successfully.\n", dof[i]->
                                                                  motor_name);
        }

        printf("\nLoaded parameter values: \n");
        for (j = 0; j < M; j++)
        {
                printf("%lf ", a_hat[j]);
        }
        printf("\n");

        return 0;
}
```

START OF KINEMATICSDYNAMICS.C CODE

```
/*
  $Id: KinematicsDynamicsLib.c 1830 2006-10-16 00:53:31Z laksman $

(c) Copyright 1999-2006
 Space Systems Lab, University of Maryland, College Park, MD 20740

 */
```

```
#include "KinematicsDynamicsLib.h"

/***************************************************************************
FUNCTION DECLARATIONS
**************************************************************************/

/* modelled dynamics adapted parameters */
double a_hat[M] = {0.};

/***************************************************************************
FUNCTION DECLARATIONS
**************************************************************************/

/*Used by Viscous_Friction_Torque()
Evaluates Kth basis function at x given parameter mesh size */
double
gk(     double  x,
        double  mesh,
        int             K);

/***************************************************************************
FUNCTION DEFINITIONS
**************************************************************************/
void
Forward_Kinematics_Pitch(     double * pos,
                              double * pos_cart)
{
        double pos_new[2] = {*pos, FIXED_ROLL_ANGLE};
        Forward_Kinematics_Pitch_Roll(pos_new, pos_cart);
}

void
Forward_Kinematics_Roll(double * pos,
                       double * pos_cart)
{
        double pos_new[2] = {FIXED_PITCH_ANGLE, *pos};
        Forward_Kinematics_Pitch_Roll(pos_new, pos_cart);
}

/*      takes position in joint space - converts to position in Cartesian space (no
orientation)
        pos -          2 x 1 vector of joint angles in radians
        pos_cart -     3 x 1 vector of end effector position in Cartesian space
*/
void
Forward_Kinematics_Pitch_Roll(       double * pos,
                                     double * pos_cart)
{
        double c1 = cos(pos[0]);
        double c2 = cos(pos[1]);
        double s1 = sin(pos[0]);
        double s2 = sin(pos[1]);

        pos_cart[0] = L2*c1*c2 - L1*s1;
        pos_cart[1] = L2*s2;
        pos_cart[2] = L2*s1*c2 + L1*c1;
}

/* performs transformation: output_vector = translational Jacobian*input_vector, given
manipulator's angular configuration
   pos -                     1 x 1 vector of joint angles in radians
   input_vector -      1 x 1 vector
   output_vector -     3 x 1 vector
*/
void
Translational_Jacobian_Pitch( double  * pos,
                              double  * input_vector,
                              double  * output_vector)
{
        double c1 = cos(pos[0]);
        double s1 = sin(pos[0]);
        /* assumes roll angle equals 180 degrees */
        output_vector[0] =   (L2*s1 - L1*c1) * input_vector[0];

        output_vector[1] =    0.;

        output_vector[2] =  (-L2*c1 - L1*s1) * input_vector[0];
}

/* performs transformation: output_vector = translational Jacobian*input_vector, given
manipulator's angular configuration
   pos -              1 x 1 vector of joint angles in radians
   input_vector -     1 x 1 vector
   output_vector -    3 x 1 vector
*/
void
Translational_Jacobian_Roll(double    * pos,
                             double  * input_vector,
                             double   * output_vector)
{
        /* assumes pitch equals 0 degrees */
        output_vector[0] = -L2*sin(pos[0]) * input_vector[0];

        output_vector[1] =  L2*cos(pos[0]) * input_vector[0];

        output_vector[2] =  0.;
}


/* performs transformation: output_vector = translational Jacobian*input_vector, given
manipulator's angular configuration
   pos -                      2 x 1 vector of joint angles in radians
   input_vector -     2 x 1 vector
   output_vector -    3 x 1 vector
*/
void
Translational_Jacobian_Pitch_Roll(   double  * pos,
                                     double  * input_vector,
                                     double  * output_vector)
{
        double c1 = cos(pos[0]);
        double c2 = cos(pos[1]);
        double s1 = sin(pos[0]);
        double s2 = sin(pos[1]);

        output_vector[0] = -(L2*s1*c2 + L1*c1)*input_vector[0] -
L2*c1*s2*input_vector[1];
        output_vector[1] =
              + L2*c2*input_vector[1];
        output_vector[2] =     (L2*c1*c2 - L1*s1)*input_vector[0] -
L2*s1*s2*input_vector[1];
}

/* performs transformation: output_vector = translational Jacobian tranpose
*input_vector, given manipulator's angular configuration
   pos -              1 x 1 vector of joint angles in radians
   input_vector -     3 x 1 vector
   output_vector -    1 x 1 vector
*/
void
Translational_Jacobian_Transpose_Pitch(      double  * pos,
                                             double  * input_vector,
                                             double  * output_vector)
{
        double c1 = cos(pos[0]);
        double s1 = sin(pos[0]);

        /* assumes roll angle equals 180 degrees */
        output_vector[0] =   (L2*s1 - L1*c1) * input_vector[0] + (-L2*c1 - L1*s1) *
                              input_vector[2];
}
```

```
/* performs transformation: output_vector = translational Jacobian*input_vector, given
manipulator's angular configuration
   pos -                     1 x 1 vector of joint angles in radians
   input_vector -      3 x 1 vector
   output_vector -     1 x 1 vector
*/
void
Translational_Jacobian_Transpose_Roll(       double  * pos,
                                             double  * input_vector,
                                             double  * output_vector)
{
       /* assumes pitch equals 0 degrees */
       output_vector[0] = -L2*sin(pos[0]) * input_vector[0] + L2*cos(pos[0]) *
                                                              input_vector[1];
}


/* performs transformation: output_vector = translational Jacobian transpose
*input_vector, given manipulator's angular configuration
   pos -              2 x 1 vector of joint angles in radians
   input_vector -      3 x 1 vector
   output_vector -     2 x 1 vector
*/
void
Translational_Jacobian_Transpose_Pitch_Roll(double   * pos,

       double  * input_vector,

       double  * output_vector)
{
       double c1 = cos(pos[0]);
       double c2 = cos(pos[1]);
       double s1 = sin(pos[0]);
       double s2 = sin(pos[1]);

       output_vector[0] = -(L2*s1*c2 + L1*c1)*input_vector[0] + (L2*c1*c2 -
                                                            L1*s1)*input_vector[2];
       output_vector[1] =     - L2*c1*s2*input_vector[0] + L2*c2*input_vector[1] -
                                                            L2*s1*s2*input_vector[2];
}

/* performs transformation: output_vector = inverse(translational
Jacobian)*input_vector, given manipulator's angular configuration
   pos  -             1 x 1 vector of joint angles in radians
   input_vector -      3 x 1 vector
   output_vector -     1 x 1 vector
*/
void
Translational_Jacobian_Inverse_Pitch( double  * pos,
                                      double  * input_vector,
                                      double  * output_vector)
{
       double c1 = cos(pos[0]);
       double s1 = sin(pos[0]);
       double L1sq_plus_L2sq = L1*L1 + L2*L2;

       /* assumes roll angle equals 180 degrees */
       output_vector[0] =   (L2*s1 - L1*c1)/L1sq_plus_L2sq * input_vector[0] +
                             (-L2*c1 - L1*s1)/L1sq_plus_L2sq * input_vector[2] ;
}

/* performs transformation: output_vector = inverse(translational
Jacobian)*input_vector, given manipulator's angular configuration
   pos  -                    1 x 1 vector of joint angles in radians
   input_vector -      3 x 1 vector
   output_vector -     1 x 1 vector
*/
void
Translational_Jacobian_Inverse_Roll(double   * pos,
                                      double  * input_vector,
                                    double  * output_vector)
{
       /* assumes pitch equals 0 degrees */
       output_vector[0] = -sin(pos[0])/L2 * input_vector[0] + cos(pos[0])/L2 *
                                                          input_vector[1];
}

/* performs transformation: output_vector = inverse(translational
Jacobian)*input_vector, given manipulator's angular configuration
   pos  -              2 x 1 vector of joint angles in radians
   input_vector -      3 x 1 vector
   output_vector -     2 x 1 vector
*/
void
Translational_Jacobian_Inverse_Pitch_Roll(   double  * pos,
                                             double   * input_vector,
                                             double  * output_vector)
{
       double c1 = cos(pos[0]);
       double c2 = cos(pos[1]);
       double s1 = sin(pos[0]);
       double s2 = sin(pos[1]);
       double t2 = tan(pos[1]);
       double L1sq_plus_L2sq = L1*L1 + L2*L2;

       output_vector[0] = -(L1*c1 + L2/c2*s1)/L1sq_plus_L2sq * input_vector[0]    -
                                  (L1*t2)/L1sq_plus_L2sq  * input_vector[1]
                         + (L2*c1/c2 - L1*s1)/L1sq_plus_L2sq * input_vector[2];

       output_vector[1] =  (-L2*c1*s2 + L1*s1*t2)/L1sq_plus_L2sq * input_vector[0] +
                          (L2*L2*c2 + L1*L1/c2)/(L2*L1sq_plus_L2sq) *input_vector[1]
                           - (L2*s1*s2 + L1*c1*t2)/L1sq_plus_L2sq * input_vector[2];
}

/* performs transformation: output_vector = inverse(transpose(translational
Jacobian))*input_vector, given manipulator's angular configuration
   pos -                     1 x 1 vector of joint angles in radians
   input_vector -      1 x 1
   output_vector -     3 x 1
*/
void
Translational_Jacobian_Transpose_Inverse_Pitch(double       * pos,
                                               double  * input_vector,
                                               double       * output_vector)
{
       double c1 = cos(pos[0]);
       double s1 = sin(pos[0]);
       double L1sq_plus_L2sq = L1*L1 + L2*L2;

       /* assumes roll angle equals 180 degrees */
       output_vector[0] =   (L2*s1 - L1*c1)/L1sq_plus_L2sq * input_vector[0];

       output_vector[1] =
                                  0.;

       output_vector[2] =    (-L2*c1 - L1*s1)/L1sq_plus_L2sq * input_vector[0];
}


/* performs transformation: output_vector = inverse(transpose(translational
Jacobian))*input_vector, given manipulator's angular configuration
   pos -                     1 x 1 vector of joint angles in radians
   input_vector -      1 x 1
   output_vector -     3 x 1
*/
void
Translational_Jacobian_Transpose_Inverse_Roll(      double  * pos,
                                                    double   * input_vector,
                                                    double  * output_vector)
{
       /* assumes pitch equals 0 degrees */
```

```
        output_vector[0] = -sin(pos[0])/L2 * input_vector[0];

        output_vector[1] =  cos(pos[0])/L2 * input_vector[0];

        output_vector[2] = 0.;
}


/* performs transformation: output_vector = inverse(transpose(translational
Jacobian))*input_vector, given manipulator's angular configuration
   pos -                    2 x 1 vector of joint angles in radians
   input_vector -      2 x 1 vector
   output_vector -     3 x 1 vector
*/
void
Translational_Jacobian_Transpose_Inverse_Pitch_Roll(double  * pos,
                                                    double  * input_vector,
                                                    double   * output_vector)
{
        double c1 = cos(pos[0]);
        double c2 = cos(pos[1]);
        double s1 = sin(pos[0]);
        double s2 = sin(pos[1]);
        double t2 = tan(pos[1]);
        double L1sq_plus_L2sq = L1*L1 + L2*L2;

        output_vector[0] = -(L1*c1 + L2/c2*s1)/L1sq_plus_L2sq * input_vector[0]
        +    (-L2*c1*s2 + L1*s1*t2)/L1sq_plus_L2sq * input_vector[1];

        output_vector[1] =           -(L1*t2)/L1sq_plus_L2sq * input_vector[0]
        +(L2*L2*c2 + L1*L1/c2)/(L2*L1sq_plus_L2sq) * input_vector[1];

        output_vector[2] =  (L2*c1/c2 - L1*s1)/L1sq_plus_L2sq * input_vector[0]
               - (L2*s1*s2 + L1*c1*t2)/L1sq_plus_L2sq * input_vector[1];
}

void
Force_Transform_Pitch( double                * pos,
                       ftsdrv_6DOF_t  * force_moment)
{
        double pos_new[2] = {*pos, FIXED_ROLL_ANGLE};
        Force_Transform_Pitch_Roll(pos_new, force_moment);
}

void
Force_Transform_Roll(  double                * pos,
                       ftsdrv_6DOF_t  * force_moment)
{
        double pos_new[2] = {FIXED_PITCH_ANGLE, *pos};
        Force_Transform_Pitch_Roll(pos_new, force_moment);
}

/* 1. transform force/moment in FTS frame to force/moment in world frame (same as 0
frame) */
/* 2. compensate end effector dynamics that cause offsets on FTS readings */
/* 3. transform compensated world frame 6 axis force/moment readings to compensated
world frame force/moment
         only along controllable DOFs
NOTE: currently only force is currently transformed in step 3. */
void
Force_Transform_Pitch_Roll(   double                * pos,
                              ftsdrv_6DOF_t  * force_moment)
{
        double c1 = cos(pos[0]);
        double c2 = cos(pos[1]);
        double s1 = sin(pos[0]);
        double s2 = sin(pos[1]);
        double force_temp[3], moment_temp[3];

        /* copy force/moment to temp */
        int i;
        for (i = 0; i < 3; i++)
        {
               force_temp[i] = force_moment->force[i];
               moment_temp[i] = force_moment->moment[i];
        }

        /* transform force/moment from FTS frame to world frame (same as frame 0) */
        force_moment->force[0] = c1*c2*force_temp[0] - c1*s2*force_temp[1] -
                                 s1*force_temp[2];
        force_moment->force[1] = s2*force_temp[0] + c2*force_temp[1];
        force_moment->force[2] = s1*c2*force_temp[0] - s1*s2*force_temp[1] +
                                 c1*force_temp[2];
        force_moment->moment[0] =-c1*s2*L_FTS*force_temp[0] -
                                  c1*c2*L_FTS*force_temp[1]
                                 + c1*c2*moment_temp[0]
                                 - c1*s2*moment_temp[1]
                                 - s1*moment_temp[2];
        force_moment->moment[1] = c2*L_FTS*force_temp[0] -s2*L_FTS*force_temp[1]
                                 + s2*moment_temp[0]     + c2*moment_temp[1];
        force_moment->moment[2] =      -s1*s2*L_FTS*force_temp[0]
                                       -s1*c2*L_FTS*force_temp[1]
                                       + s1*c2*moment_temp[0] - s1*s2*moment_temp[1]
                               + c1*moment_temp[2];

        /* copy force/moment to temp */
        for (i = 0; i < 3; i++)
        {
               force_temp[i] = force_moment->force[i];
               moment_temp[i] = force_moment->moment[i];
        }

        /* compensate end effector gravity term (other terms are negligible) */
        force_moment->force[0] =       force_temp[0];
        force_moment->force[1] =       force_temp[1];
        force_moment->force[2] =       force_temp[2] + 4.2434;
        force_moment->moment[0] =      moment_temp[0] + 0.2207*s2;
        force_moment->moment[1] =      moment_temp[1] - 0.2207*c2*c1;
        force_moment->moment[2] =      moment_temp[2];

        /* copy values to temp arrays */
        for (i = 0; i < 3; i++)
        {
               force_temp[i] =         force_moment->force[i];
        }

        /* transform 3 axis force returned by FTS into 2 axis force in
           pitch/roll direction */
        /* currently only transforms force vector */
        force_moment->force[0] =        (pow(L1, 2)*pow(c1, 2) +
                                         L2*(L2*pow(c2, 2)*pow(s1, 2) +
                                        L1*c2*sin(2*pos[0]) +
                                         L2*pow(s2, 2)))*force_temp[0]
                                        + (L2*s2*(-L2*c1*c2 + L1*s1))*force_temp[1]
                       + (-L2*c1*c2 + L1*s1)*(L1*c1 + L2*c2*s1)*force_temp[2];

        force_moment->force[1] =    (L2*s2*(-L2*c1*c2 + L1*s1))*force_temp[0]
                        + (pow(L1, 2) + pow(L2, 2)*pow(c2, 2))*force_temp[1]
                                 + (-L2*s2*(L1*c1 + L2*c2*s1))*force_temp[2];


        force_moment->force[2] = (-L2*c1*c2 + L1*s1)*(L1*c1 + L2*c2*s1)*force_temp[0]
                               + (-L2*s2*(L1*c1 + L2*c2*s1))*force_temp[1]
                               + (pow((L2*c1*c2 - L1*s1), 2)
                               + pow(L2, 2)*pow(s2, 2))*force_temp[2];

        for (i = 0; i < 3; i++)
        {
               force_moment->force[i] /= (L1*L1 + L2*L2);
        }
}
```

```
void
Dynamic_Adaptive_Torque_Pitch(       double         * act_pos,
                                     double         * act_vel,
                                     double         * des_vel_r,
                                     double         * des_accel_r,
                                     double         * s,
                                     double         * dt,
                                     BOOLEAN        adaptation_flag,
                                     double         * torque_return)
{
       static double a_sin = 0.;            /* -3.1 */
       static double old_da_sin = 0.;
       double da_sin;

       static double a_cos = 0.;            /* -.21 */
       static double old_da_cos = 0.;
       double da_cos;

       static double I = 0.;                /* 1. */
       static double old_dI = 0.;
       double dI;



       /* ------- gravitational parameters -------- */
         /*calculate the gravitational torque that will be returned */
        * torque_return = a_sin*sin(*act_pos) + a_cos*cos(*act_pos);
       /* * torque_return = a_sin * sin(*act_pos); */


       if (adaptation_flag)
       {
              /* update the current estimate of m*g*l */
              da_sin = -GAMMA_a_sin_PITCH * (*s) * sin(*act_pos);
              a_sin += .5 * (*dt) * (old_da_sin + da_sin);
              /*if (a_sin <= 0.)
              {
                     a_sin = 0.;
              }*/
              old_da_sin = da_sin;

              /* update the current estimate of a_cos */
              da_cos = -GAMMA_a_cos_PITCH * (*s) * cos(*act_pos);
              a_cos += .5 * (*dt) * (old_da_cos + da_cos);
              /*if (a_cos <= 0.)
              {
                     a_cos = 0.;
              }*/
              old_da_cos = da_cos;
        }

       /* ---------- inertia parameter --------- */
       /*calculate the inertial torque that will be returned */
       *torque_return += I * (*des_accel_r);

       if (adaptation_flag)
       {
              /* update I - the current estimate of the inertia */
              dI = -GAMMA_I_PITCH * (*des_accel_r) * (*s);
              I += (*dt) * 0.5 * (dI + old_dI);
              /*if (I <= 0.)
              {
                     I = 0.;
              }*/
              old_dI = dI;
       }

       a_hat[0] = I;
       a_hat[1] = a_sin;
       a_hat[2] = a_cos;
```

```
}

/*returns dynamic torque using model and adaptively learns parameters of model */
void
Dynamic_Adaptive_Torque_Roll( double         * act_pos,
                              double         * act_vel,
                              double         * des_vel_r,
                              double         * des_accel_r,
                              double         * s,
                              double         * dt,
                              BOOLEAN        adaptation_flag,
                              double         * torque_return)
{
        /* inertia parameter */
        static double I = 0.;

        static double old_dI = 0.;
        double dI;

        I = a_hat[0];
                                    /* --------- NEW ---------- */

        /*calculate the inertial torque that will be returned */
        *torque_return = I * (*des_accel_r);

        if (adaptation_flag)
        {
                /* update I - the current estimate of the inertia */
                dI = -GAMMA_I_ROLL * (*des_accel_r) * (*s);
                I += (*dt) * 0.5 * (dI + old_dI);
                if (I <= 0.)
                {
                        I = 0.;
                }

                old_dI = dI;

                /*printf("%f\n", I); */
        }

        a_hat[0] = I;
}

/*returns inertial torque using model and adaptively learns parameters of model */
void
Dynamic_Adaptive_Torque_Pitch_Roll(  double         * act_pos,
                                     double         * act_vel,
                                     double         * des_vel_r,
                                     double         * des_accel_r,
                                      double         * s,
                                      double         * dt,
                                      BOOLEAN        adaptation_flag,
                                      double         * torque_return)

{

        /* the current and previous versions of the derivative of a_hat */
        double da_hat[M];
        static double old_da_hat[M] = {0.};

        /* the 2 x M matrix that is independent of dynamic parameters */
        double Y[2][M];

        /* some useful variables */
        double c1 = cos(act_pos[0]);
        double c2 = cos(act_pos[1]);
        double s1 = sin(act_pos[0]);
        double s2 = sin(act_pos[1]);

        int i, j;
        /* zero Y */
        for (i = 0; i < 2; i++)
        {
                for (j = 0; j < M; j++)
                {
                        Y[i][j] = 0.;
                }
        }

        /* decoupled inertial and gravitational elements of Y - BIG TERMS */
        Y[0][0] = des_accel_r[0];
        Y[0][1] = s1;
        Y[0][2] = c1*c2;
        Y[1][2] = -s1*s2;
        Y[1][3] = des_accel_r[1];

        /* coupled inertial and Christoffel matrix elements of Y - SMALL TERMS */
        Y[0][4] = c2*c2*des_accel_r[0];
        Y[0][5] = s2*des_accel_r[1];
        Y[1][5] = s2*des_accel_r[0];
        Y[0][6] = c2*act_vel[1]*des_vel_r[1];
        Y[0][7] = c2*s2*act_vel[0]*des_vel_r[1];
        Y[1][8] = c2*s2*act_vel[0]*des_vel_r[0];

        /* calculate return torque */
        for (i = 0; i < 2; i++)
        {
                torque_return[i] = 0.;

                for (j = 0; j < M; j++)
                {
                        torque_return[i] += Y[i][j]*a_hat[j];
                }
        }

        if (adaptation_flag)
        {
                /* update a_hat - the current estimate of a */
                for (i = 0; i < M; i++)
                {
                        da_hat[i] = 0.;

                        for (j = 0; j < 2; j++)
                        {
                                if (i < 4)
                                {
                                        da_hat[i] += -GAMMA_a * Y[j][i] * s[j];

                                }
                                else    /* off-diagonal inertia and Christoffel terms */
                                {
                                        da_hat[i] += -GAMMA_a2 * Y[j][i] * s[j];
                                }
                        }

                        a_hat[i] += dt[0] * 0.5 * (da_hat[i] + old_da_hat[i]);
                        /* dt[i] changed to dt[0] */
                        old_da_hat[i] = da_hat[i];
                }
        }
}

/* get the parameters array adapted in Dynamic_Adaptive_Torque_Pitch_Roll( ) */
double * getAdaptedParams()
{
        return a_hat;
}


double
Viscous_Friction_Torque(double act_vel,
                              double                              s,
                              double                              dt,
                              Single_DOF_Properties  * dof,
                              BOOLEAN                             learning_flag)
{
        double                torque_viscous, gk1, gk2;
        int                   fres, lattice_min, lattice_max;
        Friction_Parameters   * fp;

        fp = dof->friction_parameters;

        /*determine the lattice points corresponding to the input*/
        fres = (int) floor(act_vel*fp->MESH_VISCOUS);
        lattice_min = fres - fp->MIN_NODE_VISCOUS;
        lattice_max = lattice_min + 1;

        /*perform a couple of checks to make sure the lattice points are within the
          NN's range*/
        if (lattice_min < 0)
        {
                printf("WARNING: Viscous NN lattice min = %d. Out of range of NN.
                        \n",lattice_min);
                /*printf("Input is out of range of neural network. Exiting.\n");
                return FLT_MAX; */

        }
        if (lattice_max > fp->NUM_NODES_VISCOUS)
        {
                printf("WARNING: Viscous NN lattice max = %d. Out of range of NN.
                        \n",lattice_max);
                /*printf("Input is out of range of neural network. Exiting.\n");
                return FLT_MAX; */
        }

        /* update previous lattice points if a transition has occured */
        if (learning_flag && (lattice_min != fp->old_lattice_min_VISCOUS))
        {
                fp->c_hat_VISCOUS[fp->old_lattice_min_VISCOUS] +=
                         .5 * dt * fp->old_dc_hat_VISCOUS[fp->old_lattice_min_VISCOUS];
                fp->c_hat_VISCOUS[fp->old_lattice_max_VISCOUS] +=
                         .5 * dt * fp->old_dc_hat_VISCOUS[fp->old_lattice_max_VISCOUS];
                fp->old_dc_hat_VISCOUS[lattice_min] = 0.;
                fp->old_dc_hat_VISCOUS[lattice_max] = 0.;
        }

        /*calculate the NN torque that will be returned */
        gk1 = gk(act_vel, fp->MESH_VISCOUS, lattice_min + fp->MIN_NODE_VISCOUS);
        gk2 = gk(act_vel, fp->MESH_VISCOUS, lattice_max + fp->MIN_NODE_VISCOUS);
        torque_viscous = fp->c_hat_VISCOUS[lattice_min] * gk1 +
                         fp->c_hat_VISCOUS[lattice_max] * gk2;

        if (learning_flag)
        {
                /*update the derivative of the NN's weights (dc_hat) at the current
                  lattice points*/
                fp->dc_hat_VISCOUS[lattice_min] = -dof->control_gains->GAMMA_c_VISCOUS
                                                  * s * gk1;
                fp->dc_hat_VISCOUS[lattice_max] = -dof->control_gains->GAMMA_c_VISCOUS
                                                  * s * gk2;

                /*update the NN weights at the current lattice points by performing
                  numerical integration*/
                fp->c_hat_VISCOUS[lattice_min] += .5 * dt * (fp->
                    old_dc_hat_VISCOUS[lattice_min]+fp->dc_hat_VISCOUS[lattice_min]);
                fp->c_hat_VISCOUS[lattice_max] += .5 * dt * (fp->
                    old_dc_hat_VISCOUS[lattice_max]+fp->dc_hat_VISCOUS[lattice_max]);

                /*store the current derivative of c_hat*/
                fp->old_dc_hat_VISCOUS[lattice_min] = fp->dc_hat_VISCOUS[lattice_min];
                fp->old_dc_hat_VISCOUS[lattice_max] = fp->dc_hat_VISCOUS[lattice_max];
```

```
               /*save the current lattice points so that */
               /*they can be possibly updated the next time around*/
               fp->old_lattice_min_VISCOUS = lattice_min;
               fp->old_lattice_max_VISCOUS = lattice_max;
       }

       /*return the NN torque */
       return torque_viscous;
}

/* returns torque predicted by hinge functions at low velocity (as in Peter Guion's
Master's thesis, 2003) */
double
Hinges_Torque( double                              act_vel,
               double                              s,
               double                              dt,
               Single_DOF_Properties  * dof,
               BOOLEAN                             learning_flag)
{
       double                       torque_hinges;
       double                       g_pos, g_neg;
       double                       dB_pos = 0.;
       double                       dB_neg = 0.;
       Friction_Parameters          * fp = dof->friction_parameters;

       /* evaluate hinge functions at act_vel */
       if (act_vel <= fp->ABS_VEL_MAX_HINGES && act_vel >= fp->ABS_VEL_MIN_HINGES)
       {
               g_pos = 1.;
               g_neg = 0.;
       }
       else if (act_vel >= -1*fp->ABS_VEL_MAX_HINGES &&
                 act_vel <= -1*fp->ABS_VEL_MIN_HINGES)
       {
               g_pos = 0.;
               g_neg = 1.;
       }
       else
       {
               g_pos = 0.;
               g_neg = 0.;
       }

       /*calculate torque that will be returned */
       torque_hinges = fp->B_pos_HINGES * g_pos
                      + fp->B_neg_HINGES * g_neg;
       /*torque_hinges =  fp->B_pos_HINGES * g_pos *
                                              exp(-pow(act_vel/fp->V0_HINGES, 2))
                         + fp->B_neg_HINGES * g_neg *
                                             exp(-pow(act_vel/fp->V0_HINGES, 2));  */

       /*if (torque_hinges != 0.)
       {
               printf("B+: %f        B-:%f   hinges torque: %f\n", fp->B_pos_HINGES,
                                                                      fp->B_neg_HINGES,
                                                                      torque_hinges);
       }*/

       if (learning_flag)
       {
               /*update B_pos, B_neg parameters based on adaptive learning rule */
               dB_pos =       -dof->control_gains->GAMMA_B_HINGES * s * g_pos;
               dB_neg =       -dof->control_gains->GAMMA_B_HINGES * s * g_neg;
               /*dB_pos = -dof->control_gains->GAMMA_B_HINGES * s * g_pos *
                              exp(-pow(act_vel/fp->V0_HINGES, 2));
               dB_neg = -dof->control_gains->GAMMA_B_HINGES * s * g_neg *
                              exp(-pow(act_vel/fp->V0_HINGES, 2));         */
               fp->B_pos_HINGES += .5 * dt * (fp->old_dB_pos_HINGES + dB_pos);
               fp->B_neg_HINGES += .5 * dt * (fp->old_dB_neg_HINGES + dB_neg);
               fp->old_dB_pos_HINGES = dB_pos;
               fp->old_dB_neg_HINGES = dB_neg;
```

```
        }

        return torque_hinges;
}

/*Used by friction learning functions
Evaluates Kth basis function at x given parameter mesh size */
double
gk(     double  x,
        double  mesh,
        int             K)
{
        double r = mesh*x - K;

        if ( (r<-1.) || (r>1.) )
        {
                return(0.);
        }

        if (r<0.)
        {
                return(1+r);
        }
        else
        {
                return(1-r);
        }
}
```

START OF MAIN.H CODE

```
/*
  $Id$

(c) Copyright 1999-2006
 Space Systems Lab, University of Maryland, College Park, MD 20740

 Contains the defines and extern variables used by main and control libraries

 HISTORY

Apr-2006     L Aksman        Created from main.c
 */

#ifndef        __MAIN_H
#define        __MAIN_H

#ifndef BOOLEAN
typedef char BOOLEAN;
#endif

#define _BSD_SOURCE           /* to get usleep() */

#include "ssl-os.h"
#include "rcltimes.h"
#include "comedilib.h"
#include "ComediCounter.h"
#include "rclqueue_LEON.h"
#include "ftsdrvr/h/ftsdrvr.h"
#include <math.h>
#include <string.h>
#include <stdio.h>
#include <assert.h>
#include <signal.h>
#include <limits.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>
#include <errno.h>
#include <pthread.h>
#if            !SSL_OS_IS_TIMESYS()
#include <semaphore.h>
#endif

/*****************************************************************************
DEFINES
*****************************************************************************/

/* control frequency */
#define FREQ_SYSTEM                                         3000.
                              /* Hz */

/* control periods within +/- of this will be tolerated */
#define TIMING_ERROR__MICROSECS              50.           /*microseconds */

/* priority model of Timesys 6.1, using POSIX SCHED_RR (round-robin)
or POSIX SCHED_FIFO (first in first out)
scheduler, real-time priorities from 1 to 99 (inclusive), with HIGHER
numbers being more important */
#define CONTROL_THREAD__POLICY               SCHED_FIFO
#define CONTROL_THREAD__PRIORITY             60

/* frequency at which data and parameters are saved */
#define FREQ_SAVING                          100.           /* Hz */
#define FREQ_SAVING_PARAM_EVOLUTION          (1./20.)       /* Hz */

/* running time constants */
#define TRAIN_SECONDS_DEFAULT                180
```

```
#define TEST_SECONDS_DEFAULT                 50

/* default flag values (1 - ON, 0 - OFF)
   NOTE: these defaults can be overwritten at runtime by setting them in
FILENAME_OPTIONS.
                see setOptions( ) function. */
#define ZERO_COUNTERS_DEFAULT                0 /* set the zero point of each DOF */
#define SAVE_CONTROL_DATA_DEFAULT            0 /* save control state each cycle in
                                                  FILENAME_CONTROL */
#define SAVE_PARAM_EVOLUTION_DEFAULT         0 /* save evolution of adapted parameters
                                                 in FILENAME_PARAM_EVOLUTION */
#define SAVE_LEARNED_PARAMS_DEFAULT          0 /* save adapted parameters in
                                                 FILENAME_PARAMS */
#define LOAD_LEARNED_PARAMS_DEFAULT          0 /* load parameters from
                                                 FILENAME_PARAMS */
#define ESTIMATION_IMPEDANCE_DEFAULT         0 /* impedance control based on force
                                                     estimation */
#define FTS_IMPEDANCE_DEFAULT                0 /* impedance control based on
                                               force/torque sensor (FTS) */
#define STOP_AFTER_TRAINING_DEFAULT          0 /* Maintain pos. after training or
                                                     continue training traj. */
#define USE_DYNAMIC_MODEL_DEFAULT            1 /* adapt parameters of system dynamics
                                                     */
#define USE_VISCOUS_NN_DEFAULT               1 /* viscous friction learning neural net
                                                     (NN) */
#define USE_HINGES_DEFAULT                   1 /* low velocity friction learning */

/* position defines */
#define DESIRED_INITIAL_POS_ROLL             -PI    /* radians */
#define DESIRED_INITIAL_POS_PITCH            -1.1   /* radians */
#define DESIRED_FINAL_POS_ROLL                -PI   /* radians */
#define DESIRED_FINAL_POS_PITCH              0.     /* -.6 *//* radians */
#define FIXED_ROLL_ANGLE                     -PI    /* Angle (radians) of pitch DOF
                                                    during pitch only control. */
#define FIXED_PITCH_ANGLE                    0.     /* Angle (radians) of pitch DOF
                                                    during roll only control. */
                                                    /* desired sinusoidal trajectory
                                                    defines */
#define AMPLITUDE_TRAINING                   -1.    /* Desired sinusoidal trajectory
                                                    amplitude (rad) */
#define FREQUENCY_TRAINING                   .1     /* Desired sinusoidal training
                                                    trajectory frequency (Hz) */
#define SWITCH_TRAJ_PERIODS                  100    /* Switch training trajectory
                                                    after this many periods */
#define AMPLITUDE_TESTING                    -.4    /* Desired sinusoidal testing
                                                    amplitude (rad) */
#define FREQUENCY_TESTING                    .05    /* Desired sinusoidal testing
                                                    trajectory frequency (Hz) */
#define MOVING_TO_DES_POS_ACCEL              .1     /* Accel. at which manipulator
                                                    moves to desired initial/final
                                                    pos */
#define MAX_ACT_ACCEL                        75.    /* Unfiltered accel. spikes above
                                                    this cause resampling of ctr */

/* impedance control constants (multiply identity matrix) */
#define Ms            5.     /* 100.  20. *//* Desired mass characteristic */
#define Cs            100.   /* 50.   100.*//* Desired damping characteristic */
#define Ks            10.    /* 6.   25. */ /* Desired spring characteristic */
#define Kf            .05    /* .15   .3*//* Multiplies sensed or estimated force */

/* threshold for external joint torque estimation */
#define THRESHOLD_ESTIMATES                  1      /* not 0 enables external
                                                    estimated torque thresholding */
#define ABS_VEL_MAX_THRESHOLD                .01    /* rad/s */
#define ABS_VEL_TRANSITION_WIDTH             .01    /* transition region width from
                                               low vel thresh. to normal (rad/s) */
#define TORQUE_EST_THRESH_LOW_VEL            2.     /*1.75*//* Nm */
#define TORQUE_EST_THRESH                    1.     /* Nm */
#define LV_FILT                              1      /* not 0 enables low velocity
                                                    moving average filter */
#define MOVING_AVERAGE_WIDTH                 3000   /* width of moving average filter
```

```
                                                  in cycles */

/*tuneable parameters for all DOFs */
#define PHI                          1e-3   /* used in s_DELTA calculation */
#define M                            9      /* number of physical parameters being
                                            adapted */
#define GAMMA_a                      200.   /* used in adaptation of pitch-roll
                                            dynamics.*/
#define GAMMA_a2                     (GAMMA_a/100.) /* off-diagonal inertia and
                                            Christoffel terms */

/*tuneable parameters - ROLL DOF */
#define Kp_LEARNING_ROLL             5000.          /*10000.*/

#define Kd_LEARNING_ROLL             300.           /*500.*/
#define Kp_NOT_LEARNING_ROLL         210000.
#define Kd_NOT_LEARNING_ROLL         1700.
#define GAMMA_c_VISCOUS_ROLL         2000.
#define GAMMA_B_HINGES_ROLL          2000.
#define GAMMA_I_ROLL                 100.

/*tuneable parameters  - PITCH DOF*/
#define Kp_LEARNING_PITCH            5000.          /*10000.*/

#define Kd_LEARNING_PITCH            300.           /*500.*/
#define Kp_NOT_LEARNING_PITCH        320000.
#define Kd_NOT_LEARNING_PITCH        1700.
#define GAMMA_c_VISCOUS_PITCH        2000.
#define GAMMA_B_HINGES_PITCH         2000.
#define GAMMA_I_PITCH                100.
#define GAMMA_a_sin_PITCH            100.
#define GAMMA_a_cos_PITCH            0.

/*Viscous friction NN defines*/
#define ABS_VEL_MAX_VISCOUS_ALL      2.2            /* 1.35 */
                                                    /* rad/s  (max speed at 20V
                                                    supply power is 1.3 rad/s) */
#define MESH_VISCOUS_ALL             20.            /*100.*/
                                                    /* fitting error is proportional
                                                    to 1/(MESH^2). 20 in Guion, Liu
                                                    */
#define MIN_NODE_VISCOUS_ALL  (-1.*ABS_VEL_MAX_VISCOUS_ALL*MESH_VISCOUS_ALL - 1.)
#define MAX_NODE_VISCOUS_ALL  (ABS_VEL_MAX_VISCOUS_ALL*MESH_VISCOUS_ALL + 1.)
#define NUM_NODES_VISCOUS_ALL ((-1.*MIN_NODE_VISCOUS_ALL) + MAX_NODE_VISCOUS_ALL + 1.)

/*Viscous friction hinges defines */
#define ABS_VEL_MIN_HINGES_ALL       .002           /* rad/s */
#define ABS_VEL_MAX_HINGES_ALL       .02            /* rad/s */
#define V0_HINGES_ALL                .02   /* rad/s - used for exponential basis
                                            function*/

/* useful constants */
#define NANOSECONDS_PER_SEC          (1000000000L)
#define PI                           3.14159265358979323846264338

#ifndef TRUE
#define TRUE                         (1==1)
#define FALSE                        (!TRUE)
#endif

/* physical parameters of manipulator */
#define L1                           .194           /* meters */
#define L2                           .259           /* meters */
#define L_FTS                        .180           /* meters */

/* force/torque sensor constants */
#define FTS_PORT_NUMBER              0
#define FTS_FORCE_THRESHOLD          4.             /* Newtons */
#define EE_PLATE_COMPRESSION_FX      -10.           /* Newtons */
#define EE_PLATE_COMPRESSION_FY      .9             /* Newtons */
#define EE_PLATE_COMPRESSION_FZ      -20.           /* Newtons */
```

```
#define EE_PLATE_COMPRESSION_MX       -.5            /* Nm */
#define EE_PLATE_COMPRESSION_MY       -1.5           /* Nm */
#define EE_PLATE_COMPRESSION_MZ       -.3            /* Nm */

/* COMEDI analog input constants*/
#define AI_SUBDEVICE                  0
#define AI_CHAN_0                     0              /* not used */
#define AI_CHAN_1                     1              /* roll joint motor torque */
#define AI_CHAN_2                     2              /* pitch joint motor torque */
#define AI_CHAN_3                     3              /* torque cell reading */
#define AI_RANGE_0                    0              /*   -10V to   10V */
#define AI_RANGE_1                    1              /*    -5V to    5V */
#define AI_RANGE_2                    2              /*  -0.5V to  0.5V */
#define AI_RANGE_3                    3              /* -0.05V to 0.05V */

/* COMEDI analog output constants */
#define AO_SUBDEVICE                  1
#define AO_CHAN_0                     0              /* roll joint desired torque */
#define AO_CHAN_1                     1              /* pitch joint desired torque */
#define AO_RANGE_0                    0              /*   -10V to  10V */

/* COMEDI counter constants */
#define CTR_SUBDEVICE                 4
#define CTR_CHAN_0                    0              /* roll joint counter */
#define CTR_CHAN_1                    1              /* pitch joint counter */

/* other counter constants */
#define GEAR_RATIO_ROLL               160            /*  160:1 */
#define GEAR_RATIO_PITCH              161            /*  161:1 */

#define MAX_COUNT_NO_GEARING_ROLL     36000
#define MAX_COUNT_NO_GEARING_PITCH    36000
#define MAX_COUNT_ROLL                (MAX_COUNT_NO_GEARING_ROLL*GEAR_RATIO_ROLL)
#define MAX_COUNT_PITCH               (MAX_COUNT_NO_GEARING_PITCH*GEAR_RATIO_PITCH)
#define CONV_COUNTS_TO_RADIANS_ROLL   ((2.*PI)/MAX_COUNT_ROLL)
#define CONV_COUNTS_TO_RADIANS_PITCH  ((2.*PI)/MAX_COUNT_PITCH)

/* soft stops */
#define SOFT_STOP_POS_ROLL            (5.*PI)        /*roll can make +/- 1.5
                                                     revolutions */
#define SOFT_STOP_POS_PITCH           ((75./360.)*2.*PI) /*pitch can go +/- 75 degrees
*/

/* motor names */
#define ROLL_NAME                     "ROLL"       /* MorphBots 2 DOF module's roll */
#define PITCH_NAME                    "PITCH"     /* MorphBots 2 DOF module's pitch */

/* motor control constants */
#define Kt_ROLL                       .0855          /*      Nm/amp */
#define Kt_PITCH                      .0855          /*  Nm/amp */
#define MAX_AMPS_MOTOR                5              /*      amps */
#define MAX_VOLTS_IN                  5              /*      volts */
#define MAX_VOLTS_OUT                 10             /*      volts */
#define MAX_VOLTS_OUT_SW_LIMIT        5              /*  volts */
#define AMPS_PER_VOLTS_IN_MOTOR       2              /*      amps/volt */
#define MAX_TORQUE_OUT_ROLL           (Kt_ROLL*GEAR_RATIO_ROLL*MAX_AMPS_MOTOR)
#define MAX_TORQUE_OUT_PITCH          (Kt_PITCH*GEAR_RATIO_PITCH*MAX_AMPS_MOTOR)
#define CONV_TORQUE_OUT_TO_VOLTS_ROLL   (MAX_VOLTS_OUT/MAX_TORQUE_OUT_ROLL)
#define CONV_TORQUE_OUT_TO_VOLTS_PITCH  (MAX_VOLTS_OUT/MAX_TORQUE_OUT_PITCH)
#define CONV_VOLTS_IN_TO_TORQUE_ROLL (Kt_ROLL*GEAR_RATIO_ROLL*AMPS_PER_VOLTS_IN_MOTOR)
#define CONV_VOLTS_IN_TO_TORQUE_PITCH
                                   (Kt_PITCH*GEAR_RATIO_PITCH*AMPS_PER_VOLTS_IN_MOTOR)

/*thread and file I/O defines */
#define FILENAME_OPTIONS               "/opt/wc/leon/libs/projects/leon/system_options"
#define FILENAME_CONTROL              "/tmp/datafile_control"
#define FILENAME_PARAM_EVOLUTION      "/tmp/datafile_param_evolution"
#define FILENAME_PARAMS_PITCH         "/tmp/datafile_parameters_pitch"
#define FILENAME_PARAMS_ROLL          "/tmp/datafile_parameters_roll"
#define FILENAME_PARAMS_PITCH_ROLL    "/tmp/datafile_parameters_pitch_roll"
#define FILENAME_TIMING               "/tmp/datafile_timing"
```

```
#define FILENAME_STICTION                           "/tmp/datafile_stiction"

/*COMEDI misc. defines */
#define FILENAME_COMEDI_DRIVER                      "/dev/comedi0"
#define AREF                                        AREF_GROUND

/* stiction test defines */
#define COUNT_RESOLUTION_STIC                       1000
#define END_RADIANS_STIC                            (2.*PI)
#define WAIT_TIME_STIC                              .5
#define DELTA_RADIANS_STIC
        (CONV_COUNTS_TO_RADIANS_STIC*COUNT_RESOLUTION_STIC)
#define CTR_CHAN_STIC                               CTR_CHAN_0
#define AO_CHAN_STIC                                AO_CHAN_0
#define CONV_COUNTS_TO_RADIANS_STIC                 CONV_COUNTS_TO_RADIANS_ROLL
#define CONV_TORQUE_OUT_TO_VOLTS_STIC               CONV_TORQUE_OUT_TO_VOLTS_ROLL

/* elliptic filter - 20 Hz cut-off (with 3 KHz sampling), .01 passband ripple, 40 dB
attenuation in stopband */
/* MATLAB COMMAND: [b, a] = ellip(5, .01, 40, 20/1500); */
#define FILTER_NUM_LENGTH                           6
#define FILTER_DEN_LENGTH                           6
#define FILTER_NUM
0.00151268696203, -0.00451082296087,  0.00299823910428,
0.00299823910428, -0.00451082296087,  0.00151268696203
#define FILTER_DEN
1.00000000000000, 4.88995895311333,  9.56808351940319, -9.36411094403749,
4.58381898493027, -0.89783240097174

/***************************************************************************
MODULE-LEVEL VARIABLES
***************************************************************************/
extern double                                       PERIOD_SYSTEM__MICROSECS;
extern double                                       TRAIN_SECONDS;
extern double                                       TEST_SECONDS;
extern double                                       RUN_SECONDS;

/* flags */
extern BOOLEAN                                      ZERO_COUNTERS;
extern BOOLEAN                                      SAVE_CONTROL_DATA;
extern BOOLEAN                                      SAVE_PARAM_EVOLUTION;
extern BOOLEAN                                      SAVE_LEARNED_PARAMS;
extern BOOLEAN                                      LOAD_LEARNED_PARAMS;
extern BOOLEAN                                      ESTIMATION_IMPEDANCE;
extern BOOLEAN                                      FTS_IMPEDANCE;
extern BOOLEAN                                      STOP_AFTER_TRAINING;
extern BOOLEAN                                      VELOCITY_MODIFICATION;
extern BOOLEAN                                      USE_TRAJ_DITHER;
extern BOOLEAN                                      USE_TORQUE_DITHER;
extern BOOLEAN                                      USE_DYNAMIC_MODEL;
extern BOOLEAN                                      USE_VISCOUS_NN;
extern BOOLEAN                                      USE_HINGES;

/* modelled dynamics adapted parameters */
extern double a_hat[M];

/* file I/O variables */
extern RclLeonControlDataPtrQueue            g_queue;
extern RclLeonControlDataPtrQueue            g_queue_param_evolution;

/*COMEDI (open source Linux driver project) variables */
extern comedi_t                              * daq_device;
extern comedi_range                          * output_cr, * input_cr;
extern int                                   output_max_value, input_max_value;

/* user interrupt */
extern BOOLEAN                               shouldQuit;

/* initial time used by several functions */
extern struct timespec                       initial;
```

```
/*************************************************************************
STRUCTURE DECLARATIONS
*************************************************************************/

typedef struct
{
	/* input and output */
	const int		CTR_CHAN;
	const int		AI_CHAN;
	const int		AO_CHAN;

	/* conversion constants */
	const double	CONV_COUNTS_TO_RADIANS;
	const double	CONV_VOLTS_IN_TO_TORQUE;
	const double	CONV_TORQUE_OUT_TO_VOLTS;

	/* safety */
	const double	SOFT_STOP_POS;
	const double	MAX_VOLTS_OUT_SOFT;
} Motor_Constants;

typedef struct
{
	/* PD gains */
	const double	Kp_LEARNING;


	const double	Kp_NOT_LEARNING;

	const double	Kd_LEARNING;
	const double	Kd_NOT_LEARNING;

	/* Viscous friction learning gains */


	const double	GAMMA_c_VISCOUS;


	const double	GAMMA_B_HINGES;


} Control_Gains;

typedef struct
{
	struct timespec		time_stamp_ts;
	struct timespec		time_stamp_prev_ts;
	double			time_stamp;

	/* these variables should be zeroed upon initialization */
	int				counter_val;
	int				counter_val_prev;
	double			act_pos;
	double			act_pos_prev;
	double			act_vel;
	double			act_vel_prev;


	double			des_vel_r;
	double			act_accel;


	double			act_accel_prev;
	double			* act_accel_unfilt;
	double			* act_accel_filt;
	double			des_accel_r;
	double			s_DELTA;
	double			volts_out;
	double			* torque_ext_est_unfilt;
	double			* torque_ext_est_filt;
	double			* moving_average_samples;
	int				moving_average_count;
	BOOLEAN			low_velocity_regime;
	double			torque_PD;
	double			torque_model;
	double			torque_motor;
	double			torque_external;
	double			torque_external_est;
	double			torque_external_est_LV_filt;
} Control_State;

typedef struct
{
	double			force[3];
	double			moment[3];
	double			force_est[3];
	/*double			moment_est[3]; *//* not currently being estimated */
	double			force_est_thresh[3];   /* thresholded estimated force */
	/*double			moment_est_thresh[3];*/
		/* thresholded estimated moment - not currently being estimated */
} Force_Estimation;

typedef struct
{
	/*Viscous friction NN parameters */
	double			ABS_VEL_MAX_VISCOUS;			/*const */
	double			MESH_VISCOUS;					/*const */
	int				MIN_NODE_VISCOUS;
	/*const */
	int				MAX_NODE_VISCOUS;
	/*const */
	int				NUM_NODES_VISCOUS;
	/*const */
	double			* c_hat_VISCOUS;	/* size is NUM_NODES */
	double			* dc_hat_VISCOUS;	/* size is NUM_NODES - zeroed */
	double			* old_dc_hat_VISCOUS;  /* size is NUM_NODES - zeroed */
	int				old_lattice_min_VISCOUS;/* zeroed */
	int				old_lattice_max_VISCOUS;/* zeroed */

	/*Hinges parameters */
	double			ABS_VEL_MIN_HINGES;		/*const */
	double			ABS_VEL_MAX_HINGES;					/*const */
	double			V0_HINGES;
	/*const */
	double			B_pos_HINGES;			/* zeroed */
	double			B_neg_HINGES;			/* zeroed */
	double			old_dB_pos_HINGES;		/* zeroed */
	double			old_dB_neg_HINGES;		/* zeroed */

	/* save file */
	char * filename_params;
} Friction_Parameters;

/* Contains useful constants and variables associated with one particular DOF */
typedef struct
{
	char			* motor_name;
	double			des_pos_initial;
	double			des_pos_final;
	Motor_Constants * motor_constants;
	Control_Gains  * control_gains;
	Control_State  * control_state;
	Friction_Parameters * friction_parameters;
} Single_DOF_Properties;

/* Contains useful function pointers associated with the coupling of multiple DOFs */
typedef struct
{
	/* --- Kinematics --- */
	void
	(*Forward_Kinematics)( double * pos,
						double * pos_cart);
```

```
        void
        (*Translational_Jacobian) (   double  * pos,
                                                          double  * input_vector,
                                                          double  * output_vector);

        void
        (*Translational_Jacobian_Transpose) (double  * pos,
                                            double  * input_vector,
                                            double  * output_vector);

        void
        (*Translational_Jacobian_Inverse) (   double  * pos,
                                            double  * input_vector,
                                            double  * output_vector);

        void
        (*Translational_Jacobian_Transpose_Inverse) (       double  * thetas,

                double  * input_vector,

                double  * output_vector);

        void
        (*Force_Transform) (double                   * thetas,
                                            ftsdrv_6DOF_t  * force_moment);

        /* --- Coupled Dynamics --- */
        void
        (*Dynamic_Adaptive_Torque_N_DOF)(     double         * act_pos,
                                            double         * act_vel,
                                            double         * des_vel_r,
                                            double         * des_accel_r,
                                            double         * s,
                                            double         * dt,
                                            BOOLEAN        adaptation_flag,
                                            double         * torque_return);

        /* --- Decoupled Dynamics --- */
        double
        (*Viscous_Friction_Torque)(   double         act_vel,
                                      double                              s,
                                      double                              dt,
                                      Single_DOF_Properties  * dof,
                                      BOOLEAN        learning_flag);
                                      double
        (*Hinges_Torque)(      double                              act_vel,
                               double                              s,
                               double                              dt,
                               Single_DOF_Properties                * dof,
                               BOOLEAN                             learning_flag);

} Kinematics_Dynamics_Functions;

/*************************************************************************
MAIN.c FUNCTION DECLARATIONS
*************************************************************************/

/* opens driver files and sets up I/O options, control properties */
int
controlStart(int control_mode);

/* real time thread that calls controlStart() */
void *
Control_Thread_main(void * arg);

/* created by main thread, not real time thread handles saving to file */
void *
File_Saving_Thread_main(void * arg);
```

```
/* set options to default values. open options file and set option based on whatever
options are in there.
   format of options file:
   OPTION      VALUE
   ex.
   FREQ_SYSTEM        1000
   NOTE: text should be tab delimited (as it is in this example)
*/
void setOptions(char * filename_options);

/* creates child threads which do the actual work of the program */
int
main(   int            argc,
               char    * argv[]);

/* sets parameter thread's scheduling policy and priority */
int
setScheduleParams(     pthread_t      thread,
                                      int                    sched_policy,
                                      int                    sched_priority);

#endif
```

START OF MAIN.C CODE

```
/*
  $Id$

(c) Copyright 1999-2006
 Space Systems Lab, University of Maryland, College Park, MD 20740

 Contains the Main DMU mainline

 HISTORY

 13-Dec-2005   S Roderick     Port to Timesys6
 Jan-2006             L Aksman    Began developing control code
 Jun-2006             L Aksman       Generalized control function to N DOFs
 Jul-2006             L Aksman       New method of creating multiple threads from
main thread implemented.
 */

#include "main.h"
#include "KinematicsDynamicsLib.h"
#include "ControlLib.h"
#include "FrictionTest.h"

/***************************************************************************
MODULE-LEVEL VARIABLES
***************************************************************************/
double PERIOD_SYSTEM__MICROSECS;

double TRAIN_SECONDS;                        /* specified by TRAIN_PERIODS in options
file */
double TEST_SECONDS;
double RUN_SECONDS;

/* flags */
BOOLEAN ZERO_COUNTERS;
BOOLEAN SAVE_CONTROL_DATA;
BOOLEAN SAVE_PARAM_EVOLUTION;
BOOLEAN SAVE_LEARNED_PARAMS;
BOOLEAN LOAD_LEARNED_PARAMS;
BOOLEAN ESTIMATION_IMPEDANCE;
BOOLEAN FTS_IMPEDANCE;
BOOLEAN STOP_AFTER_TRAINING;
BOOLEAN USE_DYNAMIC_MODEL;
BOOLEAN USE_VISCOUS_NN;
BOOLEAN USE_HINGES;

/*global I/O variables */
FILE *out;
FILE *out_param_evolution;
RclLeonControlDataPtrQueue g_queue;
RclLeonControlDataPtrQueue g_queue_param_evolution;
```

```
/* inter-thread variables */
BOOLEAN DONE;

/*COMEDI (open source Linux NIDAQ driver) variables */
comedi_t * daq_device;
comedi_range *output_cr, *input_cr;
int output_max_value, input_max_value;

/* number of controlled DOFs */
int N;

/* set to TRUE when want mainline to quit. This is kept outside
of d_main_t so that it always available in VxWorks once this
module has been loaded. */
BOOLEAN shouldQuit;

BOOLEAN use_NN;

/* initial time used by several functions */
struct timespec initial;

/***************************************************************************
FUNCTION DECLARATIONS
***************************************************************************/

/* instruct mainline to quit when user hits Ctrl-C */
static void
handleSigint(int in_sig);

/***************************************************************************
FUNCTION DEFINITIONS
***************************************************************************/

int
controlStart(int control_mode)
{
        int             rc;
        int             i;
        double  * a_hat;
        char    * roll_params;
        char    * pitch_params;
        char    * roll_name;
        char    * pitch_name;

        /* initialize the force/torque sensor (FTS) driver */
        rc = ftsdrvr_Initialize();
        if (rc != FTSDRVR__ERRCODE__NO_ERROR)
        {
                printf("ERROR: force/torque sensor not initialized properly. Exiting.
                        \n");
                return -1;
        }

        /* this is not done here. should be done BEFORE FTS is attached to anything */
        /* zero offset the FTS */
        /*rc = ftsdrvr_SetZeroOffset(FTS_PORT_NUMBER);
        if (rc != FTSDRVR__ERRCODE__NO_ERROR)
        {
                printf("ERROR: force/torque sensor not zeroed properly. Exiting. \n");
                return -1;
        }*/

        /*open the NIDAQ 6025e device */
        daq_device = comedi_open(FILENAME_COMEDI_DRIVER);
        if (daq_device == NULL)
        {
                printf("ERROR: COMEDI error message: %s\n",
                        comedi_strerror(comedi_errno()));
                return -1;
        }
```

```
/* comedi AO setup*/
output_cr =  comedi_get_range(daq_device, AO_SUBDEVICE, AO_CHAN_0,
                              AO_RANGE_0);
output_max_value = comedi_get_maxdata(daq_device, AO_SUBDEVICE, AO_CHAN_0);

/* comedi AI setup */
input_cr = comedi_get_range(daq_device, AI_SUBDEVICE, AI_CHAN_1, AI_RANGE_1);
input_max_value = comedi_get_maxdata(daq_device, AI_SUBDEVICE, AI_CHAN_1);


/* comedi counter setup */
if (ZERO_COUNTERS)
{
       /* Pitch DOF */
       printf("Move pitch DOF to zero position. Hit ENTER when done.\n");

       char input_char = (char) getchar();
       while (input_char != '\n')
       {
              input_char = (char) getchar();
       }
       ComediSetupCounterChannelWithZeroing(daq_device, CTR_SUBDEVICE,
                                            CTR_CHAN_1);
       printf("Pitch DOF zeroed. \n\n");

       /* Roll DOF */
       printf("Move roll DOF to zero position. Hit ENTER when done.\n");

       input_char = (char) getchar();
       while (input_char != '\n')
       {
              input_char = (char) getchar();
       }
       printf("Roll DOF zeroed. \n\n");
       ComediSetupCounterChannelWithZeroing(daq_device, CTR_SUBDEVICE,
                                            CTR_CHAN_0);
}
else
{
       ComediSetupCounterChannelWithoutZeroing(daq_device, CTR_SUBDEVICE,
                                               CTR_CHAN_0);
       ComediSetupCounterChannelWithoutZeroing(daq_device, CTR_SUBDEVICE,
                                               CTR_CHAN_1);

       /*int counter_val = 0;
       double act_pos;
       while (!shouldQuit)
       {
              counter_val = ComediReadCounterWithRollover(daq_device,
                                   CTR_SUBDEVICE, CTR_CHAN_0, counter_val);
              act_pos = counter_val * CONV_COUNTS_TO_RADIANS_ROLL;
              printf("%d    %f\n", counter_val, act_pos);
              usleep(100000);
       }*/

}

/* motor constants */
Motor_Constants motor_constants_morphBots_roll =
{      CTR_CHAN_0,
       AI_CHAN_1,
       AO_CHAN_0,
       CONV_COUNTS_TO_RADIANS_ROLL,
       CONV_VOLTS_IN_TO_TORQUE_ROLL,
       CONV_TORQUE_OUT_TO_VOLTS_ROLL,
       SOFT_STOP_POS_ROLL,
       MAX_VOLTS_OUT_SW_LIMIT
};

Motor_Constants motor_constants_morphBots_pitch =
{      CTR_CHAN_1,
                AI_CHAN_2,
                AO_CHAN_1,
                CONV_COUNTS_TO_RADIANS_PITCH,
                CONV_VOLTS_IN_TO_TORQUE_PITCH,
                CONV_TORQUE_OUT_TO_VOLTS_PITCH,
                SOFT_STOP_POS_PITCH,
                MAX_VOLTS_OUT_SW_LIMIT
        };

        /* control gains */
        Control_Gains control_gains_morphBots_roll =
        {       Kp_LEARNING_ROLL,
                Kp_NOT_LEARNING_ROLL,
                Kd_LEARNING_ROLL,
                Kd_NOT_LEARNING_ROLL,
                GAMMA_c_VISCOUS_ROLL,
                GAMMA_B_HINGES_ROLL
        };
        Control_Gains control_gains_morphBots_pitch =
        {       Kp_LEARNING_PITCH,
                Kp_NOT_LEARNING_PITCH,
                Kd_LEARNING_PITCH,
                Kd_NOT_LEARNING_PITCH,
                GAMMA_c_VISCOUS_PITCH,
                GAMMA_B_HINGES_PITCH
        };

        /* roll control state variable - calloc( ) is used because it guarantees that
all member variables will be zeroed*/
        Control_State * control_state_morphBots_roll =
        (Control_State *) calloc(1, sizeof(Control_State));
        assert(control_state_morphBots_roll != NULL);

        control_state_morphBots_roll->moving_average_samples =
                        (double *) malloc(MOVING_AVERAGE_WIDTH*sizeof(double));
        assert(control_state_morphBots_roll->moving_average_samples != NULL);

        control_state_morphBots_roll->act_accel_unfilt =
                        (double *) calloc(FILTER_NUM_LENGTH, sizeof(double));
        assert(control_state_morphBots_roll->act_accel_unfilt != NULL);
        control_state_morphBots_roll->act_accel_filt =
                        (double *) calloc(FILTER_DEN_LENGTH, sizeof(double));
        assert(control_state_morphBots_roll->act_accel_filt != NULL);
        control_state_morphBots_roll->torque_ext_est_unfilt =
                        (double *) calloc(FILTER_NUM_LENGTH, sizeof(double));
        assert(control_state_morphBots_roll->torque_ext_est_unfilt != NULL);
        control_state_morphBots_roll->torque_ext_est_filt =
                        (double *) calloc(FILTER_DEN_LENGTH, sizeof(double));
        assert(control_state_morphBots_roll->torque_ext_est_filt != NULL);


        /* pitch control state variable - calloc( ) is used because it guarantees that
          all member variables will be zeroed*/
        Control_State * control_state_morphBots_pitch =
        (Control_State *) calloc(1, sizeof(Control_State));
        assert(control_state_morphBots_pitch != NULL);

        control_state_morphBots_pitch->moving_average_samples =
                (double *) malloc(MOVING_AVERAGE_WIDTH*sizeof(double));
        assert(control_state_morphBots_pitch->moving_average_samples != NULL);
        control_state_morphBots_pitch->act_accel_unfilt =
                (double *) calloc(FILTER_NUM_LENGTH, sizeof(double));
        assert(control_state_morphBots_pitch->act_accel_unfilt != NULL);
        control_state_morphBots_pitch->act_accel_filt =
                (double *) calloc(FILTER_DEN_LENGTH, sizeof(double));
        assert(control_state_morphBots_pitch->act_accel_filt != NULL);
        control_state_morphBots_pitch->torque_ext_est_unfilt =
                (double *) calloc(FILTER_NUM_LENGTH, sizeof(double));
        assert(control_state_morphBots_pitch->torque_ext_est_unfilt != NULL);
        control_state_morphBots_pitch->torque_ext_est_filt =
                (double *) calloc(FILTER_DEN_LENGTH, sizeof(double));
```

```
assert(control_state_morphBots_pitch->torque_ext_est_filt != NULL);


/* --- ROLL DOF FRICTION PARAMETERS --- */
Friction_Parameters fp_morphBots_roll;
/*viscous */
fp_morphBots_roll.ABS_VEL_MAX_VISCOUS =      ABS_VEL_MAX_VISCOUS_ALL;
fp_morphBots_roll.MESH_VISCOUS =             MESH_VISCOUS_ALL;
fp_morphBots_roll.MIN_NODE_VISCOUS =         (int) MIN_NODE_VISCOUS_ALL;

fp_morphBots_roll.MAX_NODE_VISCOUS =         (int) MAX_NODE_VISCOUS_ALL;

fp_morphBots_roll.NUM_NODES_VISCOUS =        (int) NUM_NODES_VISCOUS_ALL;
fp_morphBots_roll.c_hat_VISCOUS =
        (double *) calloc(fp_morphBots_roll.NUM_NODES_VISCOUS, sizeof(double));
assert(fp_morphBots_roll.c_hat_VISCOUS != NULL);
fp_morphBots_roll.dc_hat_VISCOUS =
        (double *) calloc(fp_morphBots_roll.NUM_NODES_VISCOUS, sizeof(double));
assert(fp_morphBots_roll.dc_hat_VISCOUS != NULL);
fp_morphBots_roll.old_dc_hat_VISCOUS =
        (double *) calloc(fp_morphBots_roll.NUM_NODES_VISCOUS, sizeof(double));
assert(fp_morphBots_roll.old_dc_hat_VISCOUS != NULL);
fp_morphBots_roll.old_lattice_min_VISCOUS =  0;
fp_morphBots_roll.old_lattice_max_VISCOUS =  0;
/* hinges */
fp_morphBots_roll.ABS_VEL_MIN_HINGES =       ABS_VEL_MIN_HINGES_ALL;
fp_morphBots_roll.ABS_VEL_MAX_HINGES =       ABS_VEL_MAX_HINGES_ALL;
fp_morphBots_roll.V0_HINGES =                V0_HINGES_ALL;
fp_morphBots_roll.B_pos_HINGES =             0.;
fp_morphBots_roll.B_neg_HINGES =             0.;
fp_morphBots_roll.old_dB_pos_HINGES =        0.;
fp_morphBots_roll.old_dB_neg_HINGES =        0.;
/* other */
fp_morphBots_roll.filename_params =          FILENAME_PARAMS_ROLL;
/* ----------------------------------- */


/* --- PITCH DOF FRICTION PARAMETERS --- */
Friction_Parameters fp_morphBots_pitch;
/* viscous */
fp_morphBots_pitch.ABS_VEL_MAX_VISCOUS =     ABS_VEL_MAX_VISCOUS_ALL;
fp_morphBots_pitch.MESH_VISCOUS =            MESH_VISCOUS_ALL;
fp_morphBots_pitch.MIN_NODE_VISCOUS =        (int) MIN_NODE_VISCOUS_ALL;
fp_morphBots_pitch.MAX_NODE_VISCOUS =        (int) MAX_NODE_VISCOUS_ALL;
fp_morphBots_pitch.NUM_NODES_VISCOUS =       (int) NUM_NODES_VISCOUS_ALL;
fp_morphBots_pitch.c_hat_VISCOUS =
        (double *) calloc(fp_morphBots_pitch.NUM_NODES_VISCOUS,
        sizeof(double));
assert(fp_morphBots_pitch.c_hat_VISCOUS != NULL);
fp_morphBots_pitch.dc_hat_VISCOUS =
        (double *) calloc(fp_morphBots_pitch.NUM_NODES_VISCOUS,
        sizeof(double));
assert(fp_morphBots_pitch.dc_hat_VISCOUS != NULL);
fp_morphBots_pitch.old_dc_hat_VISCOUS =
        (double *) calloc(fp_morphBots_pitch.NUM_NODES_VISCOUS,
        sizeof(double));
assert(fp_morphBots_pitch.old_dc_hat_VISCOUS != NULL);
fp_morphBots_pitch.old_lattice_min_VISCOUS =  0;
fp_morphBots_pitch.old_lattice_max_VISCOUS =  0;
/* hinges */
fp_morphBots_pitch.ABS_VEL_MIN_HINGES =      ABS_VEL_MIN_HINGES_ALL;
fp_morphBots_pitch.ABS_VEL_MAX_HINGES =      ABS_VEL_MAX_HINGES_ALL;
fp_morphBots_pitch.V0_HINGES =               V0_HINGES_ALL;
fp_morphBots_pitch.B_pos_HINGES =            0.;
fp_morphBots_pitch.B_neg_HINGES =            0.;
fp_morphBots_pitch.old_dB_pos_HINGES =       0.;
fp_morphBots_pitch.old_dB_neg_HINGES =       0.;
/* other */
fp_morphBots_pitch.filename_params =         FILENAME_PARAMS_PITCH;
/* ------------------------------------ */
/* create some strings for names */
roll_params =
        (char *) malloc((strlen(FILENAME_PARAMS_ROLL) + 1) * sizeof(char));
assert(roll_params != NULL);
pitch_params =
        (char *) malloc((strlen(FILENAME_PARAMS_PITCH) + 1) * sizeof(char));
assert(pitch_params != NULL);
strcpy(roll_params,           FILENAME_PARAMS_ROLL);
strcpy(pitch_params,   FILENAME_PARAMS_PITCH);

roll_name =
                (char *) malloc((strlen(ROLL_NAME) + 1) *sizeof(char));
assert(roll_name != NULL);
pitch_name =
        (char *) malloc((strlen(PITCH_NAME) + 1)*sizeof(char));
assert(pitch_name != NULL);
strcpy(roll_name,             ROLL_NAME);
strcpy(pitch_name,            PITCH_NAME);

/* create the properties for each DOF from the above structures */
Single_DOF_Properties roll =
{       roll_name,
        DESIRED_INITIAL_POS_ROLL,
        DESIRED_FINAL_POS_ROLL,
        &motor_constants_morphBots_roll,
        &control_gains_morphBots_roll,
        control_state_morphBots_roll,
        &fp_morphBots_roll
};
Single_DOF_Properties pitch =
{       pitch_name,
        DESIRED_INITIAL_POS_PITCH,
        DESIRED_FINAL_POS_PITCH,
        &motor_constants_morphBots_pitch,
        &control_gains_morphBots_pitch,
        control_state_morphBots_pitch,
        &fp_morphBots_pitch
};

/* pass pointers to necessary kinematics and dynamics functions for pitch DOF
  of pitch-roll manipulator */
Kinematics_Dynamics_Functions pitch_kin_dyn_fns;
pitch_kin_dyn_fns.Forward_Kinematics =       &Forward_Kinematics_Pitch;
pitch_kin_dyn_fns.Translational_Jacobian =   &Translational_Jacobian_Pitch;
pitch_kin_dyn_fns.Translational_Jacobian_Transpose =
                      &Translational_Jacobian_Transpose_Pitch;
pitch_kin_dyn_fns.Translational_Jacobian_Inverse =
        &Translational_Jacobian_Inverse_Pitch;
pitch_kin_dyn_fns.Translational_Jacobian_Transpose_Inverse =
        &Translational_Jacobian_Transpose_Inverse_Pitch;
pitch_kin_dyn_fns.Force_Transform =          &Force_Transform_Pitch;
pitch_kin_dyn_fns.Dynamic_Adaptive_Torque_N_DOF =
                                             &Dynamic_Adaptive_Torque_Pitch;
pitch_kin_dyn_fns.Viscous_Friction_Torque =  &Viscous_Friction_Torque;
pitch_kin_dyn_fns.Hinges_Torque =            &Hinges_Torque;


/* pass pointers to necessary kinematics and dynamics functions for roll DOF
  of pitch-roll manipulator */
Kinematics_Dynamics_Functions roll_kin_dyn_fns;
roll_kin_dyn_fns.Forward_Kinematics =        &Forward_Kinematics_Roll;
roll_kin_dyn_fns.Translational_Jacobian =    &Translational_Jacobian_Roll;
roll_kin_dyn_fns.Translational_Jacobian_Transpose =
               &Translational_Jacobian_Transpose_Roll;
roll_kin_dyn_fns.Translational_Jacobian_Inverse =
               &Translational_Jacobian_Inverse_Roll;
roll_kin_dyn_fns.Translational_Jacobian_Transpose_Inverse =
               &Translational_Jacobian_Transpose_Inverse_Roll;
roll_kin_dyn_fns.Force_Transform =           &Force_Transform_Roll;
roll_kin_dyn_fns.Dynamic_Adaptive_Torque_N_DOF =
               &Dynamic_Adaptive_Torque_Roll;
        roll_kin_dyn_fns.Viscous_Friction_Torque = &Viscous_Friction_Torque;
        roll_kin_dyn_fns.Hinges_Torque =      &Hinges_Torque;

        /* pass pointers to necessary kinematics and dynamics functions for pitch-roll
manipulator */
        Kinematics_Dynamics_Functions pitch_roll_kin_dyn_fns;
        pitch_roll_kin_dyn_fns.Forward_Kinematics =  &Forward_Kinematics_Pitch_Roll;
        pitch_roll_kin_dyn_fns.Translational_Jacobian =
                &Translational_Jacobian_Pitch_Roll;
        pitch_roll_kin_dyn_fns.Translational_Jacobian_Transpose =
                &Translational_Jacobian_Transpose_Pitch_Roll;
        pitch_roll_kin_dyn_fns.Translational_Jacobian_Inverse =
                &Translational_Jacobian_Inverse_Pitch_Roll;
        pitch_roll_kin_dyn_fns.Translational_Jacobian_Transpose_Inverse =
                &Translational_Jacobian_Transpose_Inverse_Pitch_Roll;
        pitch_roll_kin_dyn_fns.Force_Transform =
                &Force_Transform_Pitch_Roll;
        pitch_roll_kin_dyn_fns.Dynamic_Adaptive_Torque_N_DOF =
                &Dynamic_Adaptive_Torque_Pitch_Roll;
        pitch_roll_kin_dyn_fns.Viscous_Friction_Torque = &Viscous_Friction_Torque;
        pitch_roll_kin_dyn_fns.Hinges_Torque =       &Hinges_Torque;

        Force_Estimation force_estimation;

        N = 1;
        if (control_mode == 0)                /* roll DOF control mode */
        {
                Single_DOF_Properties * roll_dof[1] = {&roll};
                Closed_Loop_Control_N_DOF(&roll_kin_dyn_fns,
                                          &force_estimation, roll_dof, N);
        }
        else if (control_mode == 1)   /* pitch DOF control mode */
        {
                Single_DOF_Properties * pitch_dof[1] = {&pitch};
                Closed_Loop_Control_N_DOF(&pitch_kin_dyn_fns,
                                         &force_estimation, pitch_dof, N);
        }
        else if (control_mode == 2)           /* two DOF control mode */
        {
                N = 2;
                /* order of DOFs is important - pitch comes first since it's closer to
                   the manipulator's base */
                Single_DOF_Properties * two_dof[2] = {&pitch, &roll};

                Closed_Loop_Control_N_DOF(&pitch_roll_kin_dyn_fns,
                                         &force_estimation, two_dof, N);
        }
        else
        {
                Stiction_Versus_Position(FILENAME_STICTION);
        }

        /* free dynamic data - roll dof */
        free(fp_morphBots_roll.c_hat_VISCOUS);
        free(fp_morphBots_roll.dc_hat_VISCOUS);
        free(fp_morphBots_roll.old_dc_hat_VISCOUS);

        free(control_state_morphBots_roll->act_accel_unfilt);
        free(control_state_morphBots_roll->act_accel_filt);
        free(control_state_morphBots_roll->torque_ext_est_unfilt);
        free(control_state_morphBots_roll->torque_ext_est_filt);
        free(control_state_morphBots_roll->moving_average_samples);
        free(control_state_morphBots_roll);
        free(roll_params);
        free(roll_name);

        /* free dynamic data - pitch dof */
        free(fp_morphBots_pitch.c_hat_VISCOUS);
        free(fp_morphBots_pitch.dc_hat_VISCOUS);
        free(fp_morphBots_pitch.old_dc_hat_VISCOUS);
        free(control_state_morphBots_pitch->act_accel_unfilt);
        free(control_state_morphBots_pitch->act_accel_filt);
        free(control_state_morphBots_pitch->torque_ext_est_unfilt);
        free(control_state_morphBots_pitch->torque_ext_est_filt);
        free(control_state_morphBots_pitch->moving_average_samples);
        free(control_state_morphBots_pitch);
        free(pitch_params);
        free(pitch_name);

        /* disable comedi device */
        int volts_bits = comedi_from_phys(0, output_cr, output_max_value);
        comedi_data_write(daq_device, AO_SUBDEVICE, AO_CHAN_0, AO_RANGE_0, AREF,
                          volts_bits);
        comedi_data_write(daq_device, AO_SUBDEVICE, AO_CHAN_1, AO_RANGE_0, AREF,
                          volts_bits);

        /* commented out because we don't want the counters losing their state between
           runs */
        /*ComediCounterDisarm(daq_device, CTR_SUBDEVICE, CTR_CHAN_0);
        ComediCounterDisarm(daq_device, CTR_SUBDEVICE, CTR_CHAN_1); */

        /* close the device drivers */
        comedi_close(daq_device);
        ftsdrvr_Shutdown();

        /* print the adapted parameters */
        a_hat = getAdaptedParams();

        printf("\nAdapted parameter values: \n");
        for (i = 0; i < M; i++)
        {
               printf("%lf ", a_hat[i]);
        }
        printf("\n");

        return 0;
}

/* real time thread that calls controlStart()
   NOTE: the returned value is not useful.
            instead, the passed in parameter arg should be checked for an error
condition */
void *
Control_Thread_main(void * arg)
{
        int                                  * rc;
        int                                  rc2;
        int                                  control_mode;
        char                         input_str[256];
        struct sigaction       sa;
        struct sigaction       saOld;
        struct sigaction       saOld2;
        sa.sa_flags     = 0;
        sa.sa_handler   = handleSigint;
        sigemptyset(&sa.sa_mask);

        /* set the input argument to zero */
        rc = (int *) arg;
        *rc = 0;

        /* set interrupt action to default for now */
        if (sigaction(SIGINT, &saOld, &saOld2) == -1)
        {
               printf("ERROR: Control_Thread_main: error setting signal handler\n");
               *rc = -1;
               return NULL;
        }

        /*------------ MODE SELECTION ------------------*/
        printf("\nHit ENTER to cycle through available control modes.\n");
        printf("Hit any key followed by ENTER when done.\n\n");
        control_mode = 0;

        printf("Control Mode: 1 DOF ROLL\n");

        /* change control modes if ENTER key hit. o/w stop waiting for input */

        input_str[0] = '\0';
        fgets(input_str, 256, stdin);

        while (strlen(input_str) == 1)        /* just the newline */
        {
                control_mode++;
                if (control_mode == 4)
                {
                        control_mode = 0;
                }

                /*print new setup based on updated control_mode */
                if (control_mode == 0)
                {
                        printf("Control Mode: 1 DOF ROLL\n");
                }
                else if (control_mode == 1)
                {
                        printf("Control Mode: 1 DOF PITCH\n");
                }
                else if (control_mode == 2)
                {
                        printf("Control Mode: 2 DOF ROLL-PITCH\n");
                }
                else
                {
                        printf("Control Mode: Stiction Test ROLL\n");
                }

                /*wait for input */
                input_str[0] = '\0';
                fgets(input_str, 256, stdin);
        }
        /*------------ END MODE SELECTION --------------*/

        /* change interrupt action */
        if (sigaction(SIGINT, &sa, &saOld) == -1)
        {
                printf("ERROR: Control_Thread_main: error setting signal handler\n");
                *rc = -1;
                return NULL;
        }

        rc2 = setScheduleParams(pthread_self(), CONTROL_THREAD__POLICY,
                                CONTROL_THREAD__PRIORITY);
        if (rc2 == 0)
        {
                *rc = controlStart(control_mode);
        }
        else
        {
                printf("\nERROR: Control_Thread_main: unable set schedule parameters
                        (rc=%d,errno=%d)\n", rc2, errno);
                printf("\n       Note that you must run as root/sudo, else get
                        (rc=1,errno=1)\n");
                return NULL;
        }

        return NULL;
}

/* created by main thread, not real time thread that handles saving to file
   NOTE: returned value is not useful. instead, parameter arg should be checked for an
error condition */
void *
```

```
File_Saving_Thread_main(void * arg)
{
        short int                     num;
        int                           i;
        int                           j;
        int                           param_count;
        int                           * rc;
        double                        * param_lengths_array;
        double                        * param_array;
        RclLeonControlData     * LCD_ptr;
        BOOLEAN                       local_DONE = FALSE;

        struct sigaction              sa;
        struct sigaction              saOld;
        sa.sa_flags     = 0;
        sa.sa_handler   = handleSigint;
        sigemptyset(&sa.sa_mask);

        /* set the input argument to zero */
        rc = (int *) arg;
        *rc = 0;

        /* change interrupt handling */
        if (sigaction(SIGINT, &sa, &saOld) == -1)
        {
               printf("ERROR: File_Saving_Thread_main: error setting signal
                      handler\n");
               *rc = -1;
               return NULL;
        }

        while(1)
        {
               /* DONE is set outside of this thread - it lets it know when to end */
               local_DONE = DONE;

               /*if there is something in the data queue, take it out regardless of
                 whether DONE is T or F*/
               num = rclNumInLeonControlDataPtrQueue(&g_queue);
               for (i = 0; i < num; i++)
               {
                      *rc = rclPopLeonControlDataPtrQueue(&g_queue, &LCD_ptr);

                      if (*rc != 0)
                      {
                             printf("ERROR popping element from queue.\n");
                             return NULL;
                      }

                      /* size N arrays */
                      for (j = 0; j < N; j++)
                      {
                             fprintf(out, "%f ",  LCD_ptr->des_pos_array[j]);

                      }
                      for (j = 0; j < N; j++)
                      {
                             fprintf(out, "%f ",  LCD_ptr->des_vel_array[j]);

                      }
                      for (j = 0; j < N; j++)
                      {
                             fprintf(out, "%f ",  LCD_ptr->des_accel_array[j]);

                      }
                      for (j = 0; j < N; j++)
                      {
                             fprintf(out, "%f ",  LCD_ptr->des_pos_mod_array[j]);

                      }
                      for (j = 0; j < N; j++)
```

```
{
        fprintf(out, "%f ",  LCD_ptr->des_vel_mod_array[j]);

}
for (j = 0; j < N; j++)
{
        fprintf(out, "%f ",  LCD_ptr->des_accel_mod_array[j]);

}
for (j = 0; j < N; j++)
{
        fprintf(out, "%f ",  LCD_ptr->act_pos_array[j]);

}
for (j = 0; j < N; j++)
{
        fprintf(out, "%f ",  LCD_ptr->act_vel_array[j]);

}
for (j = 0; j < N; j++)
{
        fprintf(out, "%f ",  LCD_ptr->act_accel_array[j]);

}
for (j = 0; j < N; j++)
{
        fprintf(out, "%f ",  LCD_ptr->torque_PD_array[j]);

}

for (j = 0; j < N; j++)
{
        fprintf(out, "%f ",  LCD_ptr->torque_model_array[j]);

}
for (j = 0; j < N; j++)
{
        fprintf(out, "%f ",  LCD_ptr->torque_motor_array[j]);

}
for (j = 0; j < N; j++)
{
        fprintf(out, "%f ",  LCD_ptr->torque_ext_array[j]);

}
for (j = 0; j < N; j++)
{
        fprintf(out, "%f ",  LCD_ptr->torque_ext_est_array[j]);

}
for (j = 0; j < N; j++)
{
        fprintf(out, "%f ",  LCD_ptr->
                              torque_ext_est_LV_filt_array[j]);
                              }
/* size 3 arrays */
for (j = 0; j < 3; j++)
{
        fprintf(out, "%f ",  LCD_ptr->force[j]);

}
for (j = 0; j < 3; j++)
{
        fprintf(out, "%f ",  LCD_ptr->moment[j]);

}
for (j = 0; j < 3; j++)
{
        fprintf(out, "%f ",  LCD_ptr->force_est[j]);
```

```
		}

		for (j = 0; j < 3; j++)
		{
			fprintf(out, "%f ",  LCD_ptr->force_est_thresh[j]);

		}
		fprintf(out, "%f\n", LCD_ptr->time_stamp);

		free(LCD_ptr->des_pos_array);
		free(LCD_ptr->des_vel_array);
		free(LCD_ptr->des_accel_array);
		free(LCD_ptr->des_pos_mod_array);
		free(LCD_ptr->des_vel_mod_array);
		free(LCD_ptr->des_accel_mod_array);
		free(LCD_ptr->act_pos_array);
		free(LCD_ptr->act_vel_array);
		free(LCD_ptr->act_accel_array);
		free(LCD_ptr->torque_PD_array);
		free(LCD_ptr->torque_motor_array);
		free(LCD_ptr->torque_model_array);
		free(LCD_ptr->torque_ext_array);
		free(LCD_ptr->torque_ext_est_array);
		free(LCD_ptr->torque_ext_est_LV_filt_array);
		free(LCD_ptr);
	}

	if (SAVE_PARAM_EVOLUTION)
	{
		/*if there is something in the param evolution queue, take it
			out regardless of whether DONE is T or F*/
		num =rclNumInLeonControlDataPtrQueue(&g_queue_param_evolution);
		for (i = 0; i < num; i++)
		{
			*rc =
			rclPopLeonControlDataPtrQueue(&g_queue_param_evolution,
			&LCD_ptr);
			if (*rc != 0)
			{
				printf("ERROR popping element from param
					evolution queue.\n");
				return NULL;
			}

			/* retrieve parameter arrays store in RclLeonControlData
			   type variable */
			param_lengths_array =  LCD_ptr->des_pos_array;
			param_array =                 LCD_ptr->des_vel_array;

			/* save param_lengths_array information */
			fprintf(out_param_evolution, "%f ",
				param_lengths_array[0]);      /* N */
			fprintf(out_param_evolution, "%f ",
				param_lengths_array[1]);      /* M */
			param_count = M;
			for (i = 0; i < N; i++)
			{
				fprintf(out_param_evolution, "%f ",
				param_lengths_array[2 + i]);
					       /* DOF i NUM_NODES_VISCOUS */
				param_count +=  param_lengths_array[2 + i];

			}

			/* save param_array information */
			for (i = 0; i < param_count; i++)
			{
			fprintf(out_param_evolution, "%f ",  param_array[i]);

			}
```

```
                                /* save time stamps */
                                fprintf(out_param_evolution,
                                        "%f\n", LCD_ptr->time_stamp);
                                free(LCD_ptr->des_pos_array);
                                free(LCD_ptr->des_vel_array);
                                free(LCD_ptr);
                        }
                }

                if (local_DONE)
                {
                        break;
                }
        }

        return NULL;
}

/* set options to default values. open options file and set option based on whatever
options are in there.
   format of options file:
   OPTION       VALUE
   ex.
   FREQ_SYSTEM          1000
   NOTE: text should be tab delimited (as it is in this example)
*/
void setOptions(char * filename_options)
{
        FILE    * options_file;
        char    str[256];
        char    * tokens[3];
        int             i;

        /* --- set default options --- */
        TRAIN_SECONDS =                         TRAIN_SECONDS_DEFAULT;

        TEST_SECONDS =                          TEST_SECONDS_DEFAULT;

        ZERO_COUNTERS =                         ZERO_COUNTERS_DEFAULT;
        SAVE_CONTROL_DATA =                     SAVE_CONTROL_DATA_DEFAULT;
        SAVE_PARAM_EVOLUTION =                  SAVE_PARAM_EVOLUTION_DEFAULT;
        SAVE_LEARNED_PARAMS =                   SAVE_LEARNED_PARAMS_DEFAULT;
        LOAD_LEARNED_PARAMS =                   LOAD_LEARNED_PARAMS_DEFAULT;
        ESTIMATION_IMPEDANCE =                  ESTIMATION_IMPEDANCE_DEFAULT;
        FTS_IMPEDANCE =                         FTS_IMPEDANCE_DEFAULT;
        STOP_AFTER_TRAINING =                   STOP_AFTER_TRAINING_DEFAULT;
        USE_DYNAMIC_MODEL =                     USE_DYNAMIC_MODEL_DEFAULT;
        USE_VISCOUS_NN =                        USE_VISCOUS_NN_DEFAULT;
        USE_HINGES =                            USE_HINGES_DEFAULT;
        /* --------------------------- */

        /* open options file */
        options_file = fopen(filename_options, "r");

        i = 1;
        while (!feof(options_file))
        {
                fgets(str, 256, options_file);

                /* get first token on line - should be the option name */
                tokens[0] = strtok(str, "\t");

                /* get second token on line - should be option value */
                tokens[1] = strtok(NULL, "\t");
                if (tokens[1] == NULL)
                {
                        printf("WARNING: Not enough words on line: %d. of options file:
                                %s. Ignoring line.\n", i, filename_options);
                        continue;
                }
```

```
		/* any other tokens on line  - error */
		tokens[2] = strtok(NULL, "\t");
		if (tokens[2] != NULL)
		{
			printf("WARNING: Extra token '%s' on line: %d. of options file:
%s. Ignoring token.\n", tokens[2], i, filename_options);
			continue;
		}

		/* determine training seconds based on number of periods of various
		   training trajectories */
		else if (!strcmp(tokens[0], "TRAIN_PERIODS"))
		{	TRAIN_SECONDS =
			atof(tokens[1])/FREQUENCY_TRAINING; }
		else if (!strcmp(tokens[0], "TEST_SECONDS"))
		{	TEST_SECONDS =		atof(tokens[1]); }
		else if (!strcmp(tokens[0], "ZERO_COUNTERS"))
		{	ZERO_COUNTERS =		(BOOLEAN) atoi(tokens[1]);
		}
		else if (!strcmp(tokens[0], "SAVE_CONTROL_DATA"))
		{	SAVE_CONTROL_DATA =	(BOOLEAN) atoi(tokens[1]); }
		else if (!strcmp(tokens[0], "SAVE_PARAM_EVOLUTION"))
		{	SAVE_PARAM_EVOLUTION = (BOOLEAN) atoi(tokens[1]); }
		else if (!strcmp(tokens[0], "SAVE_LEARNED_PARAMS"))
		{	SAVE_LEARNED_PARAMS =  (BOOLEAN) atoi(tokens[1]); }
		else if (!strcmp(tokens[0], "LOAD_LEARNED_PARAMS"))
		{	LOAD_LEARNED_PARAMS =  (BOOLEAN) atoi(tokens[1]); }
		else if (!strcmp(tokens[0], "ESTIMATION_IMPEDANCE"))
		{	ESTIMATION_IMPEDANCE = (BOOLEAN) atoi(tokens[1]); }
		else if (!strcmp(tokens[0], "FTS_IMPEDANCE"))
		{	FTS_IMPEDANCE =		(BOOLEAN) atoi(tokens[1]);
		}
		else if (!strcmp(tokens[0], "STOP_AFTER_TRAINING"))
		{	STOP_AFTER_TRAINING =  (BOOLEAN) atoi(tokens[1]); }
		else if (!strcmp(tokens[0], "USE_DYNAMIC_MODEL"))
		{	USE_DYNAMIC_MODEL =	(BOOLEAN) atoi(tokens[1]); }
		else if (!strcmp(tokens[0], "USE_VISCOUS_NN"))
		{	USE_VISCOUS_NN =	(BOOLEAN) atoi(tokens[1]); }
		else if (!strcmp(tokens[0], "USE_HINGES"))
		{	USE_HINGES =		(BOOLEAN) atoi(tokens[1]); }
		else
		{
			printf("WARNING: Token: %s on line: %d of options file: %s is
				not a valid option name. Ignoring token.\n",
				tokens[0], i, filename_options);
		}

		i++;
	}

	fclose(options_file);

	PERIOD_SYSTEM__MICROSECS =	(1000000./FREQ_SYSTEM);
	RUN_SECONDS =			TEST_SECONDS + TRAIN_SECONDS;

	if (ESTIMATION_IMPEDANCE && FTS_IMPEDANCE)
	{
		printf("ERROR: ESTIMATION_IMPEDANCE and FTS_IMPEDANCE flags both set in
			options file. Exiting.\n");
		exit(-1);
	}

	if (SAVE_PARAM_EVOLUTION && !SAVE_CONTROL_DATA)
	{
		printf("ERROR: SAVE_PARAM_EVOLUTION flag cannot be set without
			SAVE_CONTROL_DATA flag being set. Exiting.\n");
		exit(-1);
	}
}

/* creates child threads which do the actual work of the program */
```

```
int main(int argc, char       * argv[])
{
       int                   rc;
       int                   rc_control_thread;
       int                   rc_file_saving_thread;
       int                   fclose_return_val;
       pthread_t      control_thread;
       pthread_t      file_saving_thread;

       /* reset inter-thread variables */
       DONE =                        FALSE;

       struct sigaction              sa;
       struct sigaction              saOld;
       sa.sa_flags     = 0;
       sa.sa_handler   = handleSigint;
       sigemptyset(&sa.sa_mask);
       /* sa.sa_sigaction = NULL; */
                      /* _Don't_ assign this.  Is a union with sa_handler */

       rc = 0;

       /* overwrite default options with whichever are specified in options file*/
       setOptions(FILENAME_OPTIONS);

       if (sigaction(SIGINT, &sa, &saOld) == 0)
       {
              shouldQuit = FALSE;

              if (SAVE_CONTROL_DATA)
              {
                     /*open the file to be written to later*/
                     out = fopen(FILENAME_CONTROL, "w");
                     if (out == NULL)
                     {
                            printf("ERROR: failed to open file: %s. Exiting.\n",
                                   FILENAME_CONTROL);
                            exit(-1);
                     }

                     /* initialize and create the queue used for data saving*/
                     rc = rclInitLeonControlDataPtrQueue(NULL, &g_queue);
                     if (rc != 0)
                     {
                            printf("ERROR initializing queue.\n");
                     }
                     rc = rclCreateLeonControlDataPtrQueue(&g_queue);
                     if (rc != 0)
                     {
                            printf("ERROR creating queue.\n");
                     }

                     if (SAVE_PARAM_EVOLUTION)
                     {
                            /* open the second (adapted parameters) file to be
                            written to later */
                            out_param_evolution = fopen(FILENAME_PARAM_EVOLUTION,
                                                        "w");
                            if (out == NULL)
                            {
                                   printf("ERROR: failed to open file: %s.
                                          Exiting.\n", FILENAME_PARAM_EVOLUTION);
                                   exit(-1);
                            }

                            /* initialize and create the queue used for saving
                            parameter evolution */
                            rc = rclInitLeonControlDataPtrQueue(NULL,
                                                          &g_queue_param_evolution);
                            if (rc != 0)
                            {
```

```
                        printf("ERROR initializing param evolution
                                queue.\n");
                }
                rc =
                        rclCreateLeonControlDataPtrQueue(
                                        &g_queue_param_evolution);
                if (rc != 0)
                {
                        printf("ERROR creating param evolution
                                queue.\n");
                }
        }

        /* create not-real-time child thread that is in charge of
        writing data to file */
        rc = pthread_create(&file_saving_thread, NULL,
                File_Saving_Thread_main, (void *)
                &rc_file_saving_thread);
        if (rc != 0)
        {
                printf("ERROR creating file saving thread.\n");
        }

        /* sleep for a second */
        rc = usleep(500000);
        if (rc != 0)
        {
                printf("ERROR usleep() failed.\n");
        }
}

/* create real-time child thread that is in charge of control */
rc = pthread_create( &control_thread, NULL, Control_Thread_main,
                        (void *) &rc_control_thread);
if (rc != 0)
{
        printf("ERROR creating control thread.\n");
}

/* join the control thread */
rc = pthread_join(control_thread, NULL);
if (rc != 0)
{
        printf("ERROR joining control thread.\n");
}
if (rc_control_thread != 0)
{
        printf("ERROR in control thread. \n");
}

if (SAVE_CONTROL_DATA)
{
        /* let the file saving thread know it's time to stop */
        DONE = TRUE;

        /* join the file saving thread */
        rc = pthread_join(file_saving_thread, NULL);
        if (rc != 0)
        {
                printf("ERROR joining file saving thread.\n");
        }
        if (rc_file_saving_thread != 0)
        {
                printf("ERROR in file saving thread.\n");
        }

        /* destroy the queue */
        rc = rclDestroyLeonControlDataPtrQueue(&g_queue);
        if (rc != 0)
        {
                printf("ERROR destroying queue.\n");
```

```
                        }

                        /*close the file that was written to */
                        fclose_return_val = fclose(out);
                        if (fclose_return_val != 0)
                        {
                                printf("ERROR: failed to close file: %s. Exiting.\n",
                                        FILENAME_CONTROL);
                        }

                        if (SAVE_PARAM_EVOLUTION)
                        {
                                /* destroy the param evolution queue */
                                rc =
                                        rclDestroyLeonControlDataPtrQueue(
                                                &g_queue_param_evolution);
                                if (rc != 0)
                                {
                                        printf("ERROR destroying param evolution
                                                queue.\n");
                                }

                                /*close the file that was written to */
                                fclose_return_val = fclose(out_param_evolution);
                                if (fclose_return_val != 0)
                                {
                                        printf("ERROR: failed to close file: %s.
                                                Exiting.\n", FILENAME_PARAM_EVOLUTION);
                                }
                        }
                }
        }
        else
        {
                printf("\nERROR: main: error setting signal handler\n");
                return -1;
        }

        /* revert signal handler */
        if (sigaction(SIGINT, &saOld, NULL) != 0)
        {
                printf("\nERROR: main: reverting signal handler\n");
                return -1;
        }

        printf("\n--- Exiting ---\n");

        return rc;
}

/* ==================================================
Timesys FUNCTIONS
================================================== */

/* instruct mainline to quit when user hits Ctrl-C */
static void
handleSigint(int in_sig)
{
        if (in_sig == SIGINT)
        {
                shouldQuit = TRUE;
        }
}


/* returns 0 if sucessful, otherwise 1 */
int
setScheduleParams(      pthread_t       thread,
                                        int                     sched_policy,
                                        int                     sched_priority)
{
        int                                   rc2;
        struct sched_param     thread_param;

        thread_param.sched_priority = sched_priority;
        rc2 = pthread_setschedparam(thread, sched_policy, &thread_param);
        if (rc2 != 0)
        {
               printf("ERROR: main: Unable setschedparam");
        }

        return rc2;
}
```

Bibliography

Akin, D.L., "MORPHbots: Miniature self-reconfiguring modular robotics for space operations," *USU/AIAA Small Satellite Conference*, Logan, UT, August 2004.

Canudas de Wit C., Olsson H., Astrom, K.J., and Lischinsky P., "A new model for control of systems with friction," *IEEE Transactions on Automatic Control*, Vol. 40, No. 3, Mar. 1995, pp.419-425.

Canudas de Wit C. and Lischinsky P., "Adaptive friction compensation with partially known dynamic friction model," *International Journal of Adaptive Control and Signal Processing*, Vol. 11, 1997, pp.65-80.

Craig, J.J., *Introduction to Robotics*, 3rd Ed., Pearson-Prentice Hall, Upper Saddle River, NJ, 2005.

Guion, P., "Impedance control with friction adaptation for a two link robotic manipulator," *MS Thesis*, University of Maryland, College Park, 2003.

Ghandi, P.S., Ghorbel, F.H., and Dabney, J., "Modeling, identification, and compensation of friction in harmonic drives," *Proc. of the 41st IEEE Conf. on Decision and Control*, Las Vegas, Nevada, Dec. 2002, pp.160-166.

Hacksel, P. J. and Salcudean, S. J., "Estimation of environmental forces and rigid-body velocities using observers," *Proc. of IEEE Conf. on Robotics and Automation,* 1994, pp. 931-936.

Hauschild, J. P., Heppler, G., and McPhee, J., "Friction compensation of harmonic drive actuators," *6th International Conf. on Dynamics and Control of Systems and Structures in Space*, Liguria, Italy, Jul. 2004, pp.683-692.

Harmonic Drive, LLC., Peabody, MA, *http://www.harmonic -drive.net/*, 2006.

Hogan, N., "Impedance control: an approach to manipulation: parts I, II, and III, " *ASME Journal of Dynamic Systems, Measurement, and Control*, Vol. 107, No.3, 1985, pp. 1-24.

Koditschek, D., "Natural motion of robot arms," *IEEE Conference on Decision and Control*, Las Vegas, NV, 1984.

Liu, K. C., "Experimental evaluation of adaptive neurocontrollers for a prototype space robotic manipulator", *AIAA, Aerospace Sciences Meeting and Exhibit*, 35th, Reno, NV, Jan. 1997.

Maples J., and Becker, J, "Experiments in force control of robotic manipulators", *Proc. of IEEE Conf. on Robotics and Automation,* April, 1986.

Misovec, K.M, and Annaswamy, A.M., “Friction compensation using adaptive nonlinear control with persistent excitation,” *International Journal of Control*, Vol. 72, No. 5, pp.457-479.

Murakami, T., Nakamura, R., Yu, F., and Ohnishi, K., “Force sensorless impedance control by disturbance observer,” *Record of the Power Conversion Conf.*, Apr. 1993 pp.352 - 357.

Olsson, H., Aström, K.J., Canudas de Wit, C., Gäfvert, M. and Lischinsky, P., “Friction models and friction compensation,” *European Journal of Control,* No. 4, 1998, pp.176-195.

Popovic, M.R. and Goldberg A.A., “Modeling of friction using spectral analysis,” *IEEE Transactions on Robotics and Automation*, Vol. 14, No. 1, 1998, pp. 114-122.

Sabes, P. N., “Linear algebraic equations, SVD, and the pseudo-inverse,” *http://www.keck.ucsf.edu/~sabes/Docs/SVDnotes.pdf*, October 2001.

Sanner, R.M. and Slotine, J. J. E., “Gaussian networks for direct adaptive control,” *IEEE Transactions on Neural Networks*, Vol. 3, No. 6, Nov. 1992, pp. 837-863.

Sanner, R.M. and Slotine, J. J. E., “Stable adaptive control of robot manipulators using “neural” networks,” *Neural Computation*, Vol. 7, No. 4, Nov. 1995.

Salisbury, J.K., “Active stiffness control of a manipulator in Cartesian coordinates,” 19th IEEE Conference on Decision and Control, 1980, pp. 95-100.

Simpson, J. W. L., Cook, C. D., and Zheng, L. “Sensorless force estimation for robots with friction,” *Proc. of 2002 Australasian Conf. on Robotics and Automation,* Nov. 2002.

Smith, A.C. and Hashtrudi-Zaad**,** K., “Application of neural networks in inverse dynamics based contact force estimation**,”** *Proc. of IEEE Conf. on Control Applications,* Aug. 2005, pp. 1021-1026.

Slotine, J.J.E. and Li, W., "Adaptive manipulator control: a case study," *IEEE Transactions on Automatic Control*, Vol. 33, No. 11, November 1988.

Slotine, J.J.E. and Li, W., “On the adaptive control of robotic manipulators,” *International Journal of Robotics Research*, Vol. 6, No. 3, 1987.

Slotine, J.J.E and Li, W., “Theoretical issues in adaptive manipulator control,” *Proceedings of 5th Yale Workshop on Applied Adaptive Systems Theory*, New Haven, CT, 1987, pp. 252–258.

Zahn, V., Maass, R., Dapper, M., and Eckmiller, R. “Learning friction estimation for sensorless force/position control in industrial manipulator,” *Proc. of IEEE Conf. on Robotics and Automation*, pp. 2780-2785.