NOTE TO USERS

Experimental Results for Output Feedback Adaptive Robot Control

by

John M. Daly, B.Eng.

A thesis submitted to
The Faculty of Graduate Studies and Research
in partial fulfilment
of the requirements for the degree of
Master of Applied Science in Electrical Engineering

Ottawa-Carleton Institute of Electrical and Computer Engineering
Department of Systems and Computer Engineering

Carleton University
Ottawa, Ontario
March 9, 2005

Your file Votre référence
ISBN: 0-494-06788-8
Our file Notre référence
ISBN: 0-494-06788-8

Abstract

This work examines three methods of adaptive output feedback control for robotic manipulators. Implementing output feedback for control, instead of full-state feedback, allows use of only the position information. The position can be measured quite accurately, while velocity and acceleration measurements tend to get corrupted by noise. As well, having only a position sensor reduces costs in producing the robot. The three methods examined each use some form of state estimation. The methods examined are: a method proposed by Lee and Khalil using a high-gain observer, Craig, Hsu, and Sastry's method of adaptive robot control with the addition of a linear observer that we propose herein, and a method proposed by Gourdeau and Schwartz using an Extended Kalman Filter (EKF). The methods are all implemented in simulation for both noise-free and noise-contaminated cases, and experimentally on the Carleton University Direct-Drive Robot.

Acknowledgements

I would first like to thank my thesis supervisor, Professor Howard M. Schwartz, for being an excellent supervisor who was readily available, full of inspiration and guidance, and very kind. I was consistently impressed by Professor Schwartz's vision for this research, and he was a major contributor to the very positive experience I enjoyed pursuing this work.

Another major thanks goes to Professor Victor C. Aitken who was always a very willing source of help to me. Professor Aitken often helped me both to understand the theoretical side of this work, and to understand the intricacies of performing experiments of this nature on a real platform. Whenever I ran into him and had an issue, he was most willing to talk with me about it and offered very thoughtful and useful suggestions.

I would also like to thank Ian Showalter for all of his help with the Carleton University Direct-Drive Robot. Whenever I experienced difficulty using the robot, Ian was always very willing to come in and resolve the issues with me. This is especially meaningful, as Ian graduated in 2003 and has no responsibility to be involved in this work. However, I am very grateful that he is so willing and generous with his time.

Contents

List of Figures

Chapter 1

Introduction

1.1 Robot Adaptive Control

The field of Control Systems Engineering is one that encompasses a variety of control problems. From space vehicles, to power grids, to the stock market, control systems technology can be used to improve the performance of dynamic systems. This work focuses on specific control strategies for direct-drive robot manipulators.

Robot manipulators may be found in a variety of industries and can be used to grasp and manoeuvre a variety of objects. When a manipulator picks up some object, the addition of that payload mass to the robot may have an effect on its dynamics. Often in controlling such robot manipulators, the controllers implemented are based on a knowledge of the dynamics of the robot. However, a robot grasping an object may have altered the parameters of its dynamics, affecting the accuracy of the controller. This situation provides motivation for adaptive control algorithms for the robot manipulator. It is desirable to have an algorithm that will, in the presence of a change in the dynamic parameters, learn the new robot parameters.

These learned parameters may then adapt the controller to ensure that it remains an accurate representation of the dynamics of the robot. Adaptive controllers are also advantageous in that one does not need to have precise quantitative knowledge of the robot parameter values. Instead, an adaptive algorithm can be used to estimate the parameters.

This work focuses on implementation and comparison of a special class of robot adaptive control algorithms. Many adaptive robot control algorithms (e.g. [1] and [2]) require position, velocity, and sometimes acceleration measurements in order to drive the control and adaptation laws. However, while the position of a robot link can be measured accurately, measurement of velocity and acceleration tends to result in noisy signals [3]. In extreme cases, these signals could be so noisy that their use in the control or adaptation would no longer be feasible. In order to overcome the problem of noisy velocity and acceleration measurements, an observer can be used to estimate these values based on position measurements only. Not only can such a method help to yield velocity and acceleration estimates with less noise than their measured values, but robot manipulator setups using this sort of approach need only to be equipped with position sensors. This can help to decrease the costs of production. This class of algorithms is known as output feedback control since only certain system outputs are used for feedback. These methods are in contrast to full-state feedback control algorithms, in which the full system state is available for feedback.

Three methods of output feedback adaptive robot control are implemented in this work. The algorithms are initially implemented in simulation in environments free of noise as well as environments containing measurement noise. They are then implemented experimentally on the Carleton University Direct-Drive Robot. A major contribution of this work is in the experimental implementation and comparison of

these algorithms. It will be seen that an algorithm, when implemented in simulation, may perform very differently than when it is implemented experimentally. The presence of unmodeled dynamics in the robot can, in some cases, lead to difficulty of implementation.

The first method examined in this work is an approach proposed by Gourdeau and Schwartz [4] which involves the use of an Extended Kalman Filter (EKF) to perform both robot parameter estimation and estimation of position and velocity states. This results in a consolidated algorithm where a separate adaptation law is not required. However, the use of the EKF can be quite computationally expensive. A strength of this approach is in the noise rejection properties of the EKF. The noise on the position measurements is not amplified and passed through to the controller as much as in other approaches examined.

A method of output feedback adaptive control proposed by Lee and Khalil [5] is next examined. This method makes use of a high-gain observer to estimate position error and velocity error, and these estimated values are used in the control and adaptation laws. The advantage of such a high-gain observer is that the error in the observed signals tends toward zero quite rapidly, and the system is quick to recover performance similar to that achieved under full-state feedback control. As well, the high observer gains help to make it robust to unmodeled dynamics, but will increase the observer's sensitivity to measurement noise [6].

The final algorithm that will be examined in this work involves the full-state feedback adaptive algorithm proposed by Craig, Hsu, and Sastry [1]. We propose the addition of a simple linear observer to estimate position, velocity, and acceleration states from position measurements. When the Computed Torque Method (CTM) is used to control a nonlinear robot manipulator, assuming that perfect linearization

occurs, each link of the closed-loop system can be modeled as a double integrator. A simple second-order linear observer is then constructed based on the dynamics of the double integrator to estimate position, velocity, and acceleration from the position measurements.

A comparison of each of the three algorithms will be performed. One of the primary concerns when comparing these algorithms is the position tracking error. It is important to know both the peak tracking error in the transient period as well as the error at steady-state. Peak error is of importance as a large deviation from the desired trajectory may represent an attempt to move the manipulator beyond its physical limits. Steady-state error is important since it is a benchmark of the performance of the algorithm after the initial parameter tuning and transient period has taken place. Through the simulations and experiments other benchmarks will also be examined, including inertial parameter estimates, estimation error on the position and velocity states, and presence of noise in the control signal. All of these criteria will be considered to determine the relative performance of each of the algorithms.

1.2 Outline of the Thesis

The remainder of the body of this document contains four chapters divided up as follows. **Chapter 2** contains an introduction to the theory of robot adaptive control. The mathematics of two full-state feedback adaptive control algorithms are presented initially, followed by the mathematics of the three output feedback algorithms examined in this work.

Chapter 3 contains a description of the simulations performed, including the dynamics of the simulated robot. It also contains the simulation results for each of

the three algorithms. The chapter concludes with a comparison of the algorithms based on their simulation results.

Chapter 4 contains information about the experimental platform used, including physical setup and its dynamics. As well, all of the experimental results for each of the algorithms are presented in this chapter. A comparison of each of the algorithms based on experimental results is given.

Chapter 5 contains a final discussion of the results of this work for each of the three algorithms studied. Following that, some proposals for future work related to this are given.

Following the main body of the text, several appendices are included. These appendices contain code used in this work to perform simulations and experiments. This information is included with the hope that it will be of use to people working in this area in the future.

Chapter 2

Robot Adaptive Control Theory

2.1 Dynamic Model of Robot Manipulators

In this work the algorithms studied depend heavily on having an accurate dynamic model of the plant being controlled. As such, it is important to understand the dynamic models used for robot manipulators. The controllers used are known as Computed Torque Method (CTM) controllers [1]. These controllers are based on the inverse dynamic equation for the robot manipulator, and work to compute the required input torques in order to have the manipulator follow a desired trajectory.

For the algorithms considered herein, a nonlinear dynamic model of the robot is used. This model assumes that the robot links are rigid, and is formulated using Lagrangian dynamics [4]. The equation of dynamics is given as:

$$T = M(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) \tag{2.1}$$

for an n degree of freedom manipulator. Here, T is a $n \times 1$ vector of torques applied to the joints, $M(q)$ is the $n \times n$ mass (or inertia) matrix, q is the $n \times 1$ vector of

joint positions, $C(q,\dot{q})\dot{q}$ is the $n \times 1$ vector of centrifugal and Coriolis terms, and $G(q)$ is the $n \times 1$ gravity vector. The effects of friction are neglected in this model. The dynamics of (2.1) may also be expressed in linear regression form as,

$$T = Y(q,\dot{q},\ddot{q})\theta$$

where $Y(q,\dot{q},\ddot{q})$ is an $n \times p$ matrix of known functions, and θ is a $p \times 1$ vector of robot inertial parameters. When expressed in this form, it is noted that the dynamics are linear in the inertial parameters [7]. This form of the dynamics is useful in certain adaptation laws which will be presented later.

It is possible also to take into account frictional forces in the dynamics of the robot. To do that, the model may be extended as,

$$T = M(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) + V(\dot{q})$$

where the additional term, $V(\dot{q})$, represents the $n \times 1$ vector of frictional forces.

When working in simulation, it is necessary to integrate the dynamics of the robot in order to determine its position. To do that, the dynamics given by (2.1) can be reformulated as,

$$\ddot{q} = M^{-1}(q)(T - C(q,\dot{q})\dot{q} - G(q)) \tag{2.2}$$

This is a vector of n differential equations to solve for acceleration, $\ddot{q}$. Integrating these once[1] results in an $n \times 1$ vector of joint velocities, and integrating a second time will yield an $n \times 1$ vector of joint positions.

[1] A numerical algorithm such as a 4^{th} order Runge-Kutta method will work to numerically integrate these dynamics.

2.2 Full-State Feedback Adaptive Control

Early work in robot adaptive control (see [1] and [2]) yielded algorithms that depended on feeding back all of the states of the robot to the controller. These algorithms require position, velocity, and in some cases acceleration measurements.

Prior to use of nonlinear models for control of robot manipulators, many researchers focused on linear control techniques. Such techniques involve using second-order linear models to represent the nonlinear robot plant [1]. However, as manipulators move rapidly, the robot parameters may vary at such a rate that a linear model will not be adequate to represent the rapid changes in dynamics [8]. In order to compensate for this, adaptive techniques based on nonlinear models of dynamics for robot manipulators were developed.

Two early full-state feedback nonlinear adaptive algorithms will be examined. These algorithms are useful in understanding the theory involved in the use of adaptive controllers. First, the adaptive control algorithm proposed by Craig, Hsu, and Sastry in [1] will be examined, followed by the algorithm proposed by Slotine and Li in [2].

Both of the algorithms examined here are direct adaptive controllers. With this approach the adaptation is driven by the tracking error. In contrast, indirect adaptive control involves use of the difference between measured torques and predicted torques to drive the parameter adaptation algorithm [9].

2.2.1 Craig, Hsu, and Sastry's Algorithm

Craig, Hsu, and Sastry propose a method of direct adaptive control which requires measurement of position, velocity, and acceleration signals from the robot. It is

pointed out in [1] that after sufficient parameter adaptation, this algorithm decouples and linearizes the manipulator. The result of this is that each joint behaves as an independent second-order system having fixed dynamics. The control law is given by,

$$T = \hat{M}(q)\ddot{q}^* + \hat{C}(q,\dot{q})\dot{q} + \hat{G}(q) \tag{2.3}$$

where $\hat{M}(\cdot)$, $\hat{C}(\cdot)$, and $\hat{G}(\cdot)$ represent estimates of the mass matrix, coriolis matrix, and vector of gravity terms. The term $\ddot{q}^*$ is defined as,

$$\ddot{q}^* = \ddot{q}_d + K_v\dot{e} + K_p e \tag{2.4}$$

The servo error, e, is an $n \times 1$ vector that is defined as,

$$e = q_d - q$$

where q_d is the desired robot position, and q is the actual robot position. The matrices K_v and K_p are constant diagonal $n \times n$ gain matrices [1], which determine the location of the poles of the error dynamics.

For the full derivation of the adaptation law, the reader is referred to [1] and [9]. The adaptation law is given as

$$\dot{\hat{\theta}} = \Gamma Y(q,\dot{q},\ddot{q})^T \hat{M}^{-1}(q) E_1 \tag{2.5}$$

where Γ represents the adaptation gain. When choosing the adaptation gain, Γ, it is important to choose a value that will be a compromise between quick adaptation of the parameters and sensitivity to noise. This value must be selected by trial and error, as there is no theoretical approach for determining a suitable Γ. While any value for Γ which is positive definite will not affect system stability, in practice the

levels of noise must be taken into account in selecting the adaptation gain [8]. It is noted in [9] that for this algorithm low values of Γ must be used to prevent divergence, unless explicit bounds for allowable ranges are set on the parameter estimates, see (2.6). E_1 is the filtered servo error, and is given by,

$$E_1 = \dot{e} + \Psi e$$

Here Ψ is an $n \times n$ diagonal matrix, with $\psi_i > 0$ as entries along the diagonal. The ψ_i are chosen to make the transfer function

$$\frac{s + \psi_i}{s^2 + k_{vi}s + k_{pi}}$$

strictly positive real (SPR).

Additionally, the adaptation law includes reset conditions on the parameters to ensure they lie within the bounds. The reset conditions are given as,

$$\begin{aligned} \hat{\theta}_i(t^+) &= a_i, \quad \text{if } \hat{\theta}_i(t) \leq a_i - \delta, \\ \hat{\theta}_i(t^+) &= b_i, \quad \text{if } \hat{\theta}_i(t) \geq b_i + \delta \end{aligned} \tag{2.6}$$

It is clear from the adaptation law (2.5) that availability of position, velocity, and acceleration measurements are necessary for adaptation. This is an important consideration for implementation of this algorithm on an actual robot manipulator.

Using a Computed Torque Method (CTM) such as this results in error dynamics which are linear. Substituting for the torque with the robot dynamics (2.1) and the proposed controller (2.3), gives

$$\hat{M}(q)(\ddot{q}_d + K_p e + K_v \dot{e}) + \hat{C}(q, \dot{q})\dot{q} + \hat{G}(q) = M(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q)$$

From there, $\hat{M}(q)\ddot{q}$ can be subtracted from both sides. This gives,

$$\hat{M}(q)(\ddot{q}_d - \ddot{q} + K_p e + K_v \dot{e}) + \hat{C}(q,\dot{q})\dot{q} + \hat{G}(q) = M(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) - \hat{M}(q)\ddot{q}$$

Noting that $\ddot{e} = \ddot{q}_d - \ddot{q}$, and defining $\tilde{M}(q) = M(q) - \hat{M}(q)$, $\tilde{C}(q,\dot{q}) = C(q,\dot{q}) - \hat{C}(q,\dot{q})$, and $\tilde{G}(q) = G(q) - \hat{G}(q)$, this yields,

$$\hat{M}(q)(\ddot{e} + K_v \dot{e} + K_p e) = \tilde{M}(q)\ddot{q} + \tilde{C}(q,\dot{q})\dot{q} + \tilde{G}(q)$$

Finally, rearranging the equation and exploiting the fact that the robot dynamics may be written as linear in the parameters, the closed-loop error dynamics become,

$$\ddot{e} + K_v \dot{e} + K_p e = \hat{M}^{-1}(q)(Y(q,\dot{q},\ddot{q})\tilde{\theta}) \tag{2.7}$$

where $\tilde{\theta} = \theta - \hat{\theta}$. It is clear that the dynamics are driven by the parameter estimation error, $\tilde{\theta}$. As well, the error dynamics are constant over the entire workspace of the manipulator [9]. In an ideal situation where there was exact knowledge of robot parameters, the right hand side of (2.7) would be zero, and the tracking error would reach a value of zero at some point in time, depending on the placement of the poles for the error dynamics.

2.2.2 Slotine and Li's Algorithm

Slotine and Li propose a method of direct adaptive control that requires only position and velocity measurements, and does not require inversion of the inertia matrix in the adaptation law. This method also uses a Computed Torque controller that is adapted by the adaptation law. The controller contains a full dynamics feedforward part as well as a PD feedback part [2].

With this algorithm, in order to ensure that the steady-state position error is zero, a sliding surface is defined on which the position errors will be restricted to lie,

$$\dot{e} + \Lambda e = 0$$

where, in this case, the error is defined as $e = q - q_d$. Note that this definition of tracking error is opposite to the definition of tracking error used in Craig, Hsu, and Sastry's algorithm. The matrix Λ is a constant matrix having eigenvalues strictly in the right-half complex plane [2]. As well, a "virtual reference trajectory" is defined as,

$$q_r = q_d - \Lambda \int_0^t e(\tau)\, d\tau \tag{2.8}$$

Taking derivatives of (2.8), one will get

$$\dot{q}_r = \dot{q}_d - \Lambda e \tag{2.9}$$

$$\ddot{q}_r = \ddot{q}_d - \Lambda \dot{e} \tag{2.10}$$

With those definitions another term, s, can be defined as,

$$s = \dot{e}_r = \dot{q} - \dot{q}_r = \dot{e} + \Lambda e \tag{2.11}$$

The control law is then defined as,

$$T = \hat{M}(q)\ddot{q}_r + \hat{C}(q, \dot{q})\dot{q}_r + \hat{G}(q) - K_v s \tag{2.12}$$

Taking advantage of the fact that the robot manipulator dynamics are linear in the inertial parameters, this control law can be expressed equivalently as,

$$T = Y_f(q, \dot{q}, \dot{q}_r, \ddot{q}_r)\hat{\theta} - K_v s$$

where $Y_f(q, \dot{q}, \dot{q}_r, \ddot{q}_r)$ is a regression matrix similar in form to $Y(q, \dot{q}, \ddot{q})$ but using $\dot{q}_r$ to multiply with the coriolis matrix and $\ddot{q}_r$ to multiply with the mass matrix, instead of $\dot{q}$ and $\ddot{q}$ respectively.

This control law is similar to that proposed by Craig, Hsu, and Sastry, except for a few changes. Here, the term $\ddot{q}_r$ replaces the term $\ddot{q}^*$ of (2.4). As well, in this control law the coriolis matrix is multiplied by $\dot{q}_r$ instead of $\dot{q}$ as in Craig, Hsu, and Sastry's algorithm. Finally, the position and velocity error terms are grouped in s and included as a final term in the control law, whereas Craig, Hsu, and Sastry include the error feedback in the term $\ddot{q}^*$.

The adaptation law is now presented. The reader is referred to [2] for its derivation. After derivation, the adaptation law is given as,

$$\dot{\hat{\theta}} = -\Gamma^{-1} Y_f(q, \dot{q}, \dot{q}_r, \ddot{q}_r)^T s \tag{2.13}$$

where Γ is a positive definite symmetric matrix that is usually diagonal. It is noted that the adaptation law contains the matrix $Y_f(q, \dot{q}, \dot{q}_r, \ddot{q}_r)$, which makes use of the reference signal $\ddot{q}_r$ instead of the measured acceleration $\ddot{q}$. This removes the necessity of having a measured acceleration signal. The proof of stability [2] shows that the output error will converge to zero as time approaches infinity. This is shown by demonstrating that the output error converges to the sliding surface $s = 0$ given in (2.11).

In order to determine the closed-loop error dynamics, a substitution is made with the torques from the equation of dynamics (2.1) and the proposed controller (2.12). This yields,

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) = \hat{M}(q)\ddot{q}_r + \hat{C}(q,\dot{q})\dot{q}_r + \hat{G}(q) - K_v s$$

From there, substitute for $s = \dot{e} + \Lambda e$, $\ddot{q}_r = \ddot{q}_d - \Lambda\dot{e}$, and $\dot{q}_r = \dot{q}_d - \Lambda e$. This gives,

$$M(q)\ddot{q}+C(q,\dot{q})\dot{q}+G(q) = \hat{M}(q)\ddot{q}_d-\hat{M}(q)\Lambda\dot{e}+\hat{C}(q,\dot{q})\dot{q}_d-\hat{C}(q,\dot{q})\Lambda e+\hat{G}(q)-K_v\dot{e}-K_v\Lambda e$$

Next, substitute for $\ddot{q}_d = \ddot{q} - \ddot{e}$ and $\dot{q}_d = \dot{q} - \dot{e}$. As well, define $\tilde{M}(q) = \hat{M}(q) - M(q)$, $\tilde{C}(q,\dot{q}) = \hat{C}(q,\dot{q}) - C(q,\dot{q})$, and $\tilde{G}(q) = \hat{G}(q) - G(q)$. After substitution and rearranging of terms, the error dynamics become,

$$\begin{aligned} \ddot{e} \;+\; & (\Lambda + \hat{M}^{-1}(q)(\hat{C}(q,\dot{q}) + K_v))\dot{e} + \hat{M}^{-1}(q)(\hat{C}(q,\dot{q}) + K_v)\Lambda e \\ = \; & \hat{M}^{-1}(q)(\tilde{M}(q)\ddot{q} + \tilde{C}(q,\dot{q})\dot{q} + \tilde{G}(q)) \\ = \; & \hat{M}^{-1}(q)(Y(q,\dot{q},\ddot{q})\tilde{\theta}) \end{aligned} \tag{2.14}$$

In the case of this algorithm, it is clear that the error dynamics are not just governed by the feedback gains, as was the case in Craig, Hsu, and Sastry's algorithm in (2.7). In this case, the error dynamics are a function of trajectory [9]. As such, the poles of the error dynamics cannot be placed arbitrarily.

2.3 Methods of Output Feedback Adaptive Control

Following the development of full-state feedback adaptive control algorithms, a class of algorithms requiring only measurement of robot position was developed. These algorithms are termed output feedback, since the full state is not required.

In some cases, the output feedback algorithms require measurement of position, and estimation of the state through an observer. The estimated state is then used in control and adaptation (see [5], [10], [3]). Other approaches to output feedback adaptive control do not require measurement *or* estimation of states of higher order than position. See [11] for one such approach. A third approach to this problem is to combine the parameter adaptation and state estimation into one set of filter equations, instead of having a separate adaptation law and observer. The reader is referred to [12] and [4] for an example of this approach.

This class of algorithms is quite useful in that the burden of measuring several states is removed. By providing the algorithms with only a position measurement, a complete position tracking adaptive controller can be implemented.

As well, a difficulty that arises in working with the robot manipulator is in the measurement of velocity and acceleration. It is possible to obtain quite accurate measurements of position, but measurement of velocity and acceleration can lead to very noisy signals [3]. Noise on the measured signals will be fed back into the controller, and this could result in a noisy control signal. It is possible that noise on the velocity and acceleration measurements could lead to instability, due to parameter drift [8]. As a result, approaches that seek to estimate velocity and acceleration, based on a position measurement that contains tolerable amounts of noise, are very interesting to investigate.

This work examines three algorithms for output feedback adaptive control. In Section 2.3.1 an approach making use of an Extended Kalman Filter (EKF) for estimation of robot parameters *and* robot states is examined. Section 2.3.2 examines an approach based on a high-gain observer to estimate the robot states, and an adaptation law to learn the robot inertial parameters. Section 2.3.3 details an approach

based on the algorithm proposed by Craig, Hsu, and Sastry described in Section 2.2.1 with the addition of a linear observer.

2.3.1 Extended Kalman Filter Approach

Gourdeau and Schwartz [4] propose a method of nonlinear output feedback adaptive control based on an Extended Kalman Filter to provide inertial parameter estimates, and position and velocity estimates. A Computed Torque Method (CTM) controller makes use of the estimated parameters in determining the input torques [8].

An advantage of implementing a Kalman Filter approach is that the theory of Kalman filtering provides guidelines for the selection of filter parameters based on the statistics of the noise processes [8]. In other adaptive algorithms, there are no clear rules for selection of adaptation gains. These values must be found by trial and error.

Implementation of the Extended Kalman Filter

When working with this algorithm, the robot dynamic model is expressed as,

$$T = M(q,\theta)\ddot{q} + h(q,\dot{q},\theta) + G(q,\theta) \tag{2.15}$$

where $h(q,\dot{q})$ here is the same term as $C(q,\dot{q})\dot{q}$ of (2.1) except it is expressed here as an $n \times 1$ vector, instead of an $n \times n$ matrix times an $n \times 1$ vector. This notation simplifies calculation of the derivatives of the coriolis terms when deriving the perturbation model. As well, it is expressed like this to show the dependence of the model on the $p \times 1$ parameter vector θ. Re-writing the dynamics in terms of the joint acceleration and defining the input $u(t) = T$, the dynamic equation is expressed as,

$$\ddot{q} = M^{-1}(q,\theta)(u(t) - h(q,\dot{q},\theta) - G(q,\theta)) \tag{2.16}$$

In order to implement the EKF, a state space model of the system is defined. The state vector is defined as,

$$x = \begin{bmatrix} q \\ \dot{q} \\ \theta \end{bmatrix} \tag{2.17}$$

With the state vector, the nonlinear state space model of the system is given by,

$$\dot{x} = f(x,u) \tag{2.18}$$

where

$$f(x,u) = \begin{bmatrix} \dot{q} \\ \ddot{q} \\ 0 \end{bmatrix} = \begin{bmatrix} \dot{q} \\ M^{-1}(q,\theta)(u(t) - h(q,\dot{q},\theta) - G(q,\theta)) \\ 0 \end{bmatrix} \tag{2.19}$$

and $\ddot{q}$ is as given in (2.16). Note that the derivative of θ is zero, since θ is a vector of constant robot parameters. This set of equations represents a system free of noise that is deterministic [8]. However, in reality there may be sources of noise, including noise on the input torques, and noise causing parameter variation. In order to take these factors into account, the model may be extended as follows,

$$\dot{x} = f(x,u) + G(x)w \tag{2.20}$$

$$G(x) = \begin{bmatrix} 0 & 0 \\ M^{-1}(q,\theta) & 0 \\ 0 & I \end{bmatrix} \tag{2.21}$$

$$w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \tag{2.22}$$

Here, w_1 represents input disturbances and is mapped through to the input yielding $u + w_1$, instead of u in the deterministic case. The term w_2 represents parameter variations, and replaces the derivative of the inertial parameters, $\dot{\theta} = 0$ by $\dot{\theta} = w_2$. Both elements of w are random variables [8]. The matrix $G(x)$ maps the disturbances into the state space of the system.

In this work, a continuous-time EKF is implemented. Based on a general nonlinear dynamic model,

$$\dot{x}(t) = f(x(t), u(t)) + G(x(t))w(t)$$

with the measurement model,

$$z(t) = Hx(t) + v(t)$$

where $v(t)$ is a Gaussian independent random process representing measurement noise, the dynamic EKF equations can be given. (See [13] and [8] for more information on the Extended Kalman Filter.) Note that the matrix H is a constant matrix, and the measurements are a linear combination of the states [8]. The matrix H is given as,

$$H = \begin{bmatrix} I_{n\times n} & 0_{n\times n} & 0_{n\times p} \end{bmatrix}$$

A more general case would be a time varying matrix, $H(t)$, however that is not treated here. In order to implement an EKF, the nonlinear system model is linearized about the best estimate of the states at each instant in time using a first order Taylor series expansion [13]. Ignoring terms of order higher than one in the power series, this yields,

$$f(x(t), u(t)) \approx f(\hat{x}(t), u(t)) + \frac{\partial f}{\partial \hat{x}}(x(t) - \hat{x}(t))$$

where $\hat{x}(t)$ represents an estimate of $x(t)$. With this model, the EKF equations can be outlined. The state estimate equation is given by,

$$\dot{\hat{x}}(t) = f(\hat{x}(t), u(t)) + P(t)H^T R^{-1}(t)(z(t) - H\hat{x}(t)) \tag{2.23}$$

and the error covariance equation is given by,

$$\dot{P}(t) = \frac{\partial f}{\partial \hat{x}}P(t) + P(t)\frac{\partial f^T}{\partial \hat{x}} + G(x(t))Q(t)G^T(x(t)) - P(t)H^T R^{-1}(t)HP(t) \tag{2.24}$$

The matrices $Q(t)$ and $R(t)$ are known as *spectral density matrices* [13]. The relationship between the spectral density matrix and the covariance matrix is established through multiplication of the spectral density matrix by the Dirac delta function [13]. For $Q(t)$ and $R(t)$, this is expressed mathematically as,

$$\text{cov}\{w(t), w(\tau)\} = Q(t)\delta_D(t - \tau) \tag{2.25}$$

$$\text{cov}\{v(t), v(\tau)\} = R(t)\delta_D(t - \tau) \tag{2.26}$$

However, it is common to refer to the matrices $Q(t)$ and $R(t)$ as covariance matrices. That convention will be adopted in this work.

In order to determine how to correctly set the initial filter conditions, an understanding of the role of each of them is important. For the initial estimate of the state and the initial error covariance matrix, the equations are given as,

$$\begin{aligned} \hat{x}(t_o) &= \mathrm{E}[x(t_o)] \\ P(t_o) &= \mathrm{var}[\tilde{x}(t_o)] \end{aligned}$$

where $\tilde{x}(t_o)$ represents this initial error on the estimate of the state. The operator $\mathrm{E}[\cdot]$ represents the expectation of a variable and $\mathrm{var}[\cdot]$ represents the variance. When considering the states of the robot manipulator, as given in (2.17), a suitable initial estimate of the state is,

$$\hat{x}(t_o) = \begin{bmatrix} z(t_o) \\ 0 \\ \theta_o \end{bmatrix}$$

This assumes that the robot is starting at rest. The first measured position is used to initialize the position estimate, and the best available estimate of robot parameters is used to initialize θ_o, the initial vector of inertial robot parameters.

The initial error covariance matrix is given by,

$$P(t_o) = \begin{bmatrix} P_q & 0 & 0 \\ 0 & P_{\dot{q}} & 0 \\ 0 & 0 & P_\theta \end{bmatrix}$$

where P_q is a diagonal matrix representing the estimate of the variance of the initial position measurement. The matrix $P_{\dot{q}}$ is diagonal and represents an estimate of the variance of the confidence in the fact that the robot actually does begin at rest. Finally, P_θ is a diagonal matrix representing an estimate of the covariance of the error on the initial robot parameter estimate [8].

With respect to the matrices $Q(t)$ and $R(t)$, they can be considered intuitively as follows. The matrix $R(t)$ is a diagonal matrix representing the variance on the system measurements. The matrix $Q(t)$ may be broken into two diagonal sub-matrices as follows,

$$Q(t) = \begin{bmatrix} Q_1(t) & 0 \\ 0 & Q_2(t) \end{bmatrix}$$

where $Q_1(t)$ represents the magnitude of the disturbances caused by unmodeled dynamics. This is effectively related to the confidence in the dynamic model available. The matrix $Q_2(t)$ represents the rate at which the vector of inertial robot parameters is estimated to vary. It affects the rate at which the robot inertial parameters are learned. As a result, it would affect the amount of time for the EKF to respond to a change in inertial parameters, such as an increased payload. As the elements of $Q_2(t)$ are increased, the EKF estimates the robot parameters at a greater rate [8].

In equation (2.24) the derivative of $f(\hat{x}, u)$ with respect to the estimate of the state, $\hat{x}$, is required. It is worth outlining the procedure used to obtain that derivative. The nonlinear dynamics written in terms of the estimated states takes the form,

$$f(\hat{x},u) = \begin{bmatrix} \dot{\hat{q}} \\ \ddot{\hat{q}} \\ 0 \end{bmatrix} = \begin{bmatrix} \dot{\hat{q}} \\ M^{-1}(\hat{q},\hat{\theta})(u(t) - h(\hat{q},\dot{\hat{q}},\hat{\theta}) - G(\hat{q},\hat{\theta})) \\ 0 \end{bmatrix} \tag{2.27}$$

The derivate of one $n \times 1$ vector with respect to another $n \times 1$ vector is an $n \times n$ matrix. For the case of $f(x(t),u(t))$, the derivative takes the general form,

$$F(x(t),u(t)) = \frac{\partial f(x(t),u(t))}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}$$

For the robot dynamics given as $f(\hat{x},u)$ the derivative becomes

$$\frac{\partial f}{\partial \hat{x}} = \begin{bmatrix} 0 & I & 0 \\ F_{21}(t) & F_{22}(t) & F_{23}(t) \\ 0 & 0 & 0 \end{bmatrix} \tag{2.28}$$

$$F_{21}(t) = \frac{\partial f_2}{\partial \hat{x}_1} = -M^{-1}\left(\frac{\partial M}{\partial \hat{q}}\ddot{\hat{q}} + \frac{\partial h}{\partial \hat{q}} + \frac{\partial G}{\partial \hat{q}}\right) \tag{2.29}$$

$$F_{22}(t) = \frac{\partial f_2}{\partial \hat{x}_2} = -M^{-1}\frac{\partial h}{\partial \dot{\hat{q}}} \tag{2.30}$$

$$F_{23}(t) = \frac{\partial f_2}{\partial \hat{x}_3} = -M^{-1}\left(\frac{\partial M}{\partial \hat{\theta}}\ddot{\hat{q}} + \frac{\partial h}{\partial \hat{\theta}} + \frac{\partial G}{\partial \hat{\theta}}\right) \tag{2.31}$$

In order to compute the partial derivative of $M(\hat{q},\hat{\theta})$ with respect to either $\hat{q}$ or $\hat{\theta}$, as required in the above equations, one ends up with a three dimensional array [8]. However, as this term is multiplied by the $n \times 1$ vector $\ddot{\hat{q}}$, the end result is an $n \times n$ matrix $\frac{\partial M}{\partial \hat{q}}\ddot{\hat{q}}$ (or $\frac{\partial M}{\partial \hat{\theta}}\ddot{\hat{q}}$ when the partial derivative is taken with respect to $\hat{\theta}$). The ij^{th} element of this matrix may be computed as,

$$[\cdot]_{ij} = \left(\sum_{k=1}^{n} \frac{\partial M_{ik}}{\partial \hat{q}_j} \ddot{\hat{q}}_k \right), \ddot{\hat{q}} \in \Re^n, M \in \Re^{n \times n}$$

when the partial derivative of $M(\hat{q}, \hat{\theta})$ is taken with respect to $\hat{q}$. This result comes from [6]. The result takes the same form when the partial derivative is taken with respect to $\hat{\theta}$.

Again when taking these derivatives, another important identity is used in order to put the terms in the form above. That identity is given as,

$$\frac{\partial}{\partial(\cdot)} M^{-1} = -M^{-1} \frac{\partial M}{\partial(\cdot)} M^{-1}$$

With an understanding of the EKF used to estimate the robot states and inertial parameters, attention is now turned to the control law used to drive the robot. A Computed Torque Method controller relying on position, velocity, and robot inertial parameter estimates is used. This controller is given by,

$$T = \hat{M}(\hat{q})(\ddot{q}_d + K_v \dot{\hat{e}} + K_p \hat{e}) + \hat{C}(\hat{q}, \dot{\hat{q}})\dot{\hat{q}} + \hat{G}(\hat{q}) \tag{2.32}$$

where $\hat{e} = q_d - \hat{q}$ and $\dot{\hat{e}} = \dot{q}_d - \dot{\hat{q}}$ are $n \times 1$ error vectors. Note that $\hat{C}(\hat{q}, \dot{\hat{q}})\dot{\hat{q}} = \hat{h}(\hat{q}, \dot{\hat{q}})$ as defined above. The matrices K_v and K_p are constant diagonal $n \times n$ gain matrices.

The Extended Kalman Filter given by equations (2.23) and (2.24) along with the CTM controller given in (2.32) make up this output feedback adaptive control algorithm.

2.3.2 High-Gain Observer Approach

Lee and Khalil [5] propose a nonlinear output feedback adaptive control algorithm that makes use of a high-gain nonlinear observer. This observer is used to estimate

the position and velocity error of the robot in tracking a desired trajectory. An important advantage that comes with a high-gain observer is that the error dynamics of the observer are very fast. This results in very rapid convergence of estimated state values to their true values. The result is that the system is quick to recover performance similar to that achieved under full-state feedback control.

However, use of such a high-gain observer may result in peaking in its transient behaviour [5]. This peaking could be transmitted to the manipulator, and could lead to instabilities. In order to overcome this issue with peaking, Lee and Khalil propose a modification to the control which saturates the control inputs above some pre-defined level.

A potential drawback of this method is that, due to the high gain of the observer, the presence of noise in the measurements (such as quantization error on digitized position measurements) may make estimation of the position and velocity errors prohibitively inaccurate.

In their work [5], Lee and Khalil begin by developing a globally bounded state feedback controller and then introduce the observer to estimate position and velocity error. However, in this presentation of the material we begin with the output feedback controller including the observer.

Based on the dynamic model (2.1), Lee and Khalil propose the control law,

$$T = \hat{M}(\hat{q})\ddot{q}_d + \hat{C}(\hat{q}, \hat{\dot{q}}_r)\dot{q}_d + \hat{G}(\hat{q}) - K_d\hat{\dot{e}} - K_p\hat{e} \tag{2.33}$$

The desired trajectory is given by q_d, an $n \times 1$ vector. Here, $\hat{e} = \hat{q} - q_d$, $\hat{\dot{q}}_r = \hat{\dot{q}} - \lambda\hat{e}$, with $\lambda = \lambda_0/(1 + ||\hat{e}||)$, $\lambda_0 > 0$. The notation $||\hat{e}||$ is defined as $||\hat{e}|| = (\hat{e}^T\hat{e})^{1/2}$. $\hat{M}(\cdot)$, $\hat{C}(\cdot)$, and $\hat{G}(\cdot)$ represent estimates (based on robot parameter estimates) of each of the actual matrices. K_d and K_p are positive definite symmetric constant gain

matrices. Their values will determine the response of the error dynamics.

To augment the control law (2.33) in order to take into account saturation, the saturated control input is given by,

$$T_i^s = S_i \,\mathrm{sat}\left(\frac{T_i}{S_i}\right), 1 \leq i \leq n \tag{2.34}$$

where $\mathrm{sat}(\cdot)$ is defined as,

$$\mathrm{sat}(\xi) = \begin{cases} -1, & \text{for } \xi < -1 \\ \xi, & \text{for } |\xi| \leq 1 \\ 1, & \text{for } \xi > 1 \end{cases}$$

Implementing this saturated control input will ensure that, for the i^{th} control input, $|T_i^s| \leq S_i$ for all $t \geq 0$. This saturation ensures that the peaking resulting from the high-gain observer is not transmitted to the plant.

The derivation of the adaptation law is based on Lyapunov stability and can be found in [5]. This adaptation law requires *a priori* bounds on the parameter estimates. These bounds are represented by the set,

$$\Theta = \{\theta | a_i \leq \theta_i \leq b_i, 1 \leq i \leq p\}$$

Now let,

$$\Theta_\delta = \{\theta | a_i - \delta \leq \theta_i \leq b_i + \delta, 1 \leq i \leq p\}$$

where $\delta > 0$. The adaptation rule then contains a parameter projection feature to ensure that the parameter estimates remain in Θ_δ whenever the initial parameter estimates are in Θ. The adaptation rule is given as,

$$\dot{\hat{\theta}}_i = \begin{cases} \gamma_{ii}\phi_i & \text{if } a_i < \hat{\theta}_i < b_i \text{ or} \\ & \text{if } \hat{\theta}_i \geq b_i \text{ and } \phi_i \leq 0 \text{ or} \\ & \text{if } \hat{\theta}_i \leq a_i \text{ and } \phi_i \geq 0 \\ \gamma_{ii}(1 + \dfrac{b_i - \hat{\theta}_i}{\delta})\phi_i & \text{if } \hat{\theta}_i \geq b_i \text{ and } \phi_i \geq 0 \\ \gamma_{ii}(1 + \dfrac{\hat{\theta}_i - a_i}{\delta})\phi_i & \text{if } \hat{\theta}_i \leq a_i \text{ and } \phi_i \leq 0 \end{cases} \tag{2.35}$$

where ϕ_i is the ith element of $\phi = -Y_r^T(\hat{q}, \dot{\hat{q}}_r, \dot{q}_d, \ddot{q}_d)s$. Y_r is the same function as Y given in the linear regression form of the dynamics, but it has different arguments. See [5] for a more complete treatment. The vector s is defined as $s = \dot{\hat{e}} + \lambda\hat{e}$. The scalar γ_{ii} is the ith diagonal of a positive diagonal matrix Γ which represents the adaptation gain. The $n \times 1$ vector $\hat{\theta}$ is the estimate of the parameter vector θ.

In order to determine the matrix $Y_r^T(\hat{q}, \dot{\hat{q}}_r, \dot{q}_d, \ddot{q}_d)$ from robot dynamics expressed in the form of (2.1), one can begin with Lee and Khalil's control law (2.33), neglecting the position error and velocity error feedback terms. By making the appropriate substitutions into each term (i.e. making use of $\dot{\hat{q}}_r$, $\dot{q}_d$, and $\ddot{q}_d$) one can combine all terms into an $n \times 1$ vector. One can then write that equation as linear in the parameters and end up with,

$$T = Y_r(\hat{q}, \dot{\hat{q}}_r, \dot{q}_d, \ddot{q}_d)\hat{\theta}$$

The matrix Y_r may then be used in the adaptation law.

The high-gain observer used to estimate the error dynamics will now be given. First, define $x_1 = e$ and $x_2 = \dot{e}$. The equations for the observer are given as,

$$\dot{\hat{x}}_1 = \hat{x}_2 + \frac{1}{\epsilon} L_1(x_1 - \hat{x}_1) \tag{2.36}$$

$$\begin{aligned}\dot{\hat{x}}_2 = {} & \frac{1}{\epsilon^2} L_2(x_1 - \hat{x}_1) - \ddot{q}_d - \\ & \hat{M}^{-1}(\hat{x}_1, q_d)[\hat{C}(\hat{x}, q_d, \dot{q}_d)(\hat{x}_2 + \dot{q}_d) \\ & + \hat{G}(\hat{x}_1, q_d)] + \\ & \hat{M}^{-1}(\hat{x}_1, q_d) T^s(\hat{x}, q_d, \dot{q}_d, \ddot{q}_d, \hat{\theta})\end{aligned} \tag{2.37}$$

where $L_1 = \text{diag}\{\alpha_{1i}\}$ and $L_2 = \text{diag}\{\alpha_{2i}\}$, $i = 1, \ldots, n$. The high-gain of the observer comes from ϵ, which is a small positive parameter. The values for matrices L_1 and L_2 are chosen such that,

$$\bar{A} = \begin{bmatrix} -L_1 & I \\ -L_2 & 0_{n \times n} \end{bmatrix}$$

is Hurwitz.[2] T^s represents the saturated input torques, which are saturated at some pre-determined threshold.

In (2.36) and (2.37) the parameter ϵ exists. This parameter is directly related to the observer gain, and it is its setting that makes this observer high-gain. Lee and Khalil use two values for ϵ in [5]: $\epsilon = 0.001$ and $\epsilon = 0.01$. One can see from the observer equations that these values result in a large observer gain.

The control law given by (2.33) together with the adaptation law of (2.35) and the observer given by (2.36)-(2.37) serve to make up this output feedback adaptive control algorithm.

[2] A matrix is said to be Hurwitz if the real parts of its eigenvalues lie in the left half-plane.

2.3.3 Addition of a Linear Observer to Craig's Algorithm

In Section 2.2.1 a method of full-state feedback adaptive controlled proposed by Craig, Hsu, and Sastry [1] is examined. Motivated by the potential of using only a position measurement to perform adaptive control, a modification to this algorithm is proposed in order to use output feedback techniques. The algorithm proposed by Craig, Hsu, and Sastry requires knowledge of position, velocity, and acceleration. Many robots do not come equipped with velocity and acceleration sensors [5], and specifically the Carleton University Direct-Drive Robot, the robot used for experiments in this work, does not have an acceleration sensor.

In order to overcome these issues, we have proposed the use of a second-order linear observer to estimate position, velocity, and acceleration from the position measurements. This observer has its poles placed so that they are much faster than the error dynamics of the closed-loop system, but small enough that the response to noise in the system will remain reasonable.

With the exception that an observer is used to estimate position, velocity, and acceleration, the control and adaptation laws are the same as those given in Section 2.2.1. The system equations are given here to reflect the added observer. The control law is given by,

$$T = \hat{M}(\hat{q})\hat{\ddot{q}}^* + \hat{C}(\hat{q}, \hat{\dot{q}})\hat{\dot{q}} + \hat{G}(\hat{q}) \tag{2.38}$$

The term $\hat{\ddot{q}}^*$ is defined as,

$$\hat{\ddot{q}}^* = \ddot{q}_d + K_v\hat{\dot{e}} + K_p\hat{e} \tag{2.39}$$

The estimated servo error, $\hat{e}$, is an $n \times 1$ vector defined as,

$$\hat{e} = q_d - \hat{q}$$

where q_d is the desired robot position, and $\hat{q}$ is the estimated actual robot position. The adaptation law is given as,

$$\dot{\hat{\theta}} = \Gamma Y(\hat{q}, \dot{\hat{q}}, \ddot{\hat{q}})^T \hat{M}^{-1}(\hat{q}) E_1 \tag{2.40}$$

where E_1 is the filtered servo error, based on estimates of position and velocity, and is given by,

$$E_1 = \dot{\hat{e}} + \Psi \hat{e}$$

Note that all parameters have the same meanings as those given in Section 2.2.1, and the reader is directed there for further explanation. The adaptation law given in (2.40) contains reset conditions on the parameters that are identical to those given in (2.6).

After feedback linearization, using a computed torque method such as the one given, each link of the robot manipulator can be thought of as a double integrator [11]. Based on that, we propose using a second-order linear observer for each joint. From this observer, position, velocity, and acceleration will be estimated. In state-space form, the equation of the observer for the ith link can be expressed as,

$$\begin{bmatrix} \dot{\hat{q}}_i \\ \ddot{\hat{q}}_i \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \hat{q}_i \\ \dot{\hat{q}}_i \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} v + \begin{bmatrix} K_1 \\ K_2 \end{bmatrix} (q_i - \hat{q}_i) \tag{2.41}$$

The values K_1 and K_2 represent the observer gains. For the input, v, to the observer, the same input as we give to the control law is used, that is $v = \ddot{q}^*$. By integrating the left hand side of (2.41), the estimates of position and velocity are

obtained. In order to obtain the acceleration estimate, a copy of the $\hat{\dot{q}}_i$ term is kept. The control and adaptation laws can then be implemented with the estimates obtained.

This modification to Craig, Hsu, and Sastry's algorithm allows its use on platforms without velocity and acceleration sensors, and also eliminates dependence on the potentially noisy velocity and acceleration signals. The control law given by (2.38) together with the adaptation law given by (2.40), the parameter reset conditions of (2.6), and the linear observer given by (2.41) make up this adaptive output feedback control algorithm.

Chapter 3

Simulation Results

3.1 Methodology Used in Simulations

This chapter outlines the implementation of the three output feedback adaptive control algorithms given in Chapter 2 with the dynamics of a 2 degree of freedom robot manipulator in simulation. The simulations were undertaken to examine the performance of the adaptive control algorithms while maintaining the ability to regulate the environment. When working in simulation, one can ensure that the dynamic model used in the algorithms is completely accurate and that there are no unmodeled dynamics. As well, the amount of noise present in the simulated system can be completely specified. This allows the testing of the algorithms in completely noise free environments, as well as in very noisy ones.

It can be advantageous to examine the performance of an algorithm in a situation without measurement and process noise to ensure that it functions correctly as a theoretical tool. Once this is determined, addition of noise to the system can aid in determining the effectiveness of the algorithm in situations that are closer to true

environments. For these reasons, simulation was used as an approach in this work to verify the adaptive control algorithms given in Chapter 2. Cases free of noise were run, as well as cases having uniformly distributed random noise to simulate the quantization error of a 12-bit position resolver.

In order to implement the algorithms in simulation, the continuous time equations of dynamics were used. At each time step in the simulation, a 4^{th} order Runge-Kutta algorithm is used to numerically integrate the continuous time dynamics. The error associated with numerical integration becomes smaller as the step size decreases, so a small sample period was used in all simulations. The sample period T_s was set to $T_s = 0.001$ seconds, corresponding to a sampling frequency of 1000 Hz. This sample frequency is well above the bandwidth of the closed-loop system.

The simulations were all implemented in C++. This approach allows reasonably fast simulations, as compared to an interpreted scripting environment. As well, since the eventual goal was to implement the algorithms experimentally on the Carleton University Direct-Drive Robot, a solution that would facilitate implementation in both simulation and on the real platform was sought. C++ is a reasonable language to use when interfacing at the device level, and as a result it was chosen for both the simulations and the experiments. The result is that minimal re-writing of code was necessary when implementing the control algorithms experimentally.

3.2 Dynamic Model of the Simulated Robot

For the simulations in this work, the dynamics of a $n = 2$ degree of freedom serial link manipulator are used. The equation of dynamics for this manipulator comes from [11] and is repeated here for convenience. The dynamics take the form of (2.1)

but the robot operates in the horizontal plane, and as such $G(q)$ is zero. There are $p = 2$ parameters to be estimated for the robot, they are,

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} = \begin{bmatrix} m_1 l^2 \\ m_2 l^2 \end{bmatrix}$$

where the true values of these parameters are specified as $\theta_1 = 1.0$ and $\theta_2 = 2.0$.

The mass and coriolis matrices are given as,

$$M(q) = \begin{bmatrix} \theta_1 + 2\theta_2 + 2\theta_2 \cos q_2 & \theta_2 + \theta_2 \cos q_2 \\ \theta_2 + \theta_2 \cos q_2 & \theta_2 \end{bmatrix} \tag{3.1}$$

$$C(q, \dot{q}) = \begin{bmatrix} -2\theta_2 \dot{q}_2 \sin q_2 & -\theta_2 \dot{q}_2 \sin q_2 \\ \theta_2 \dot{q}_1 \sin q_2 & 0 \end{bmatrix} \tag{3.2}$$

Fig. 3.1 shows a diagram of the robot manipulator represented by the given dynamics. This diagram is based on a figure found in [11]. Since the robot operates in the horizontal plane, one must interpret this diagram as a top view of the manipulator.

3.3 Description of Simulation Runs

In order to perform comparisons of the algorithms in a consistent manner, it was necessary to ensure that as many parameters as possible be kept the same among the algorithms. Such parameters include controller feedback gains, trajectory pre-filter, and desired trajectory.

To generate the trajectory for the robot to follow, a command signal consisting of a square wave with a period of 20 seconds was used. The square wave has peak values of ± 1 radians. This signal was pre-filtered using a critically damped second-

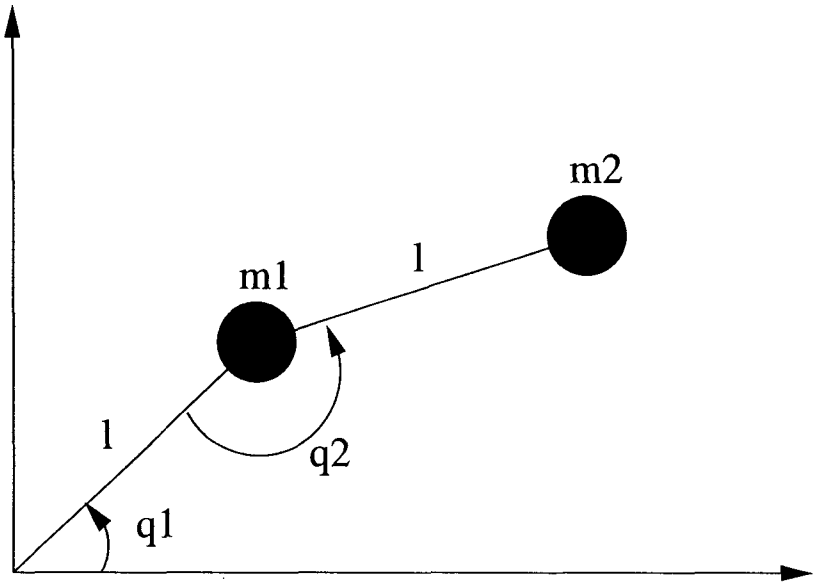

Figure 3.1: Diagram of the robot manipulator used in simulation.

order linear filter with a bandwidth of $\omega_n = 2.0$ rad/s. The transfer function for this filter is given as,

$$G(s) = \frac{4}{s^2 + 4s + 4} \tag{3.3}$$

Filtering of the square wave resulted in a smooth trajectory for the robot to follow. Fig. 3.2 shows the original square wave as well as the filtered trajectory signal.

In all simulations, the controller dynamics were set to have a bandwidth ω_n of 2.0 rad/sec, and a damping ration ζ of 1, matching the bandwidth of the trajectory pre-filter (3.3). This resulted in controller feedback gains of $K_p = 4I_{2\times2}$ and $K_d = 4I_{2\times2}$. The sample period for all of the simulations was set at $T_s = 0.001$, corresponding to a sampling frequency of 1000 Hz, which is well above the bandwidth of the closed-loop system and is faster than the observer time constants. In all cases the robot parameter estimates were initialized to $\hat{\theta}_1 = 1.5$ and $\hat{\theta}_2 = 3$, representing 1.5 times

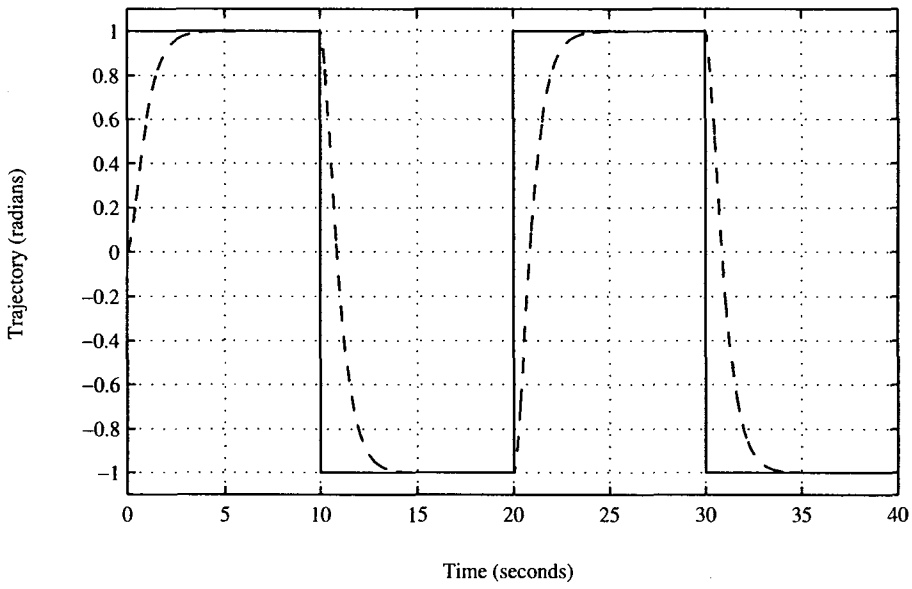

Figure 3.2: Original signal (solid) and pre-filtered trajectory (dashed) use in the simulations. The robot is required to track the dashed line.

the true parameter values.

3.4 Results of Simulations

Simulation is a very powerful tool in that it is possible to get results at a much faster rate than is possible with experiments on a real platform. A true robot must be operated in real time such that in order to examine the performance of an algorithm for some period of time, the robot would need to be operated for that whole period of time. For example, if a person wished to run a controller on a robot for a 5 minute experiment, the person would need to wait 5 minutes for the experiment to finish. Simulation can speed this process up and allow a person to run tests on the algorithms at a much faster rate. As a result, it is a very convenient approach in order

to establish an initial comparison of the three algorithms examined in this work.

Each of the algorithms performed differently with respect to each other in simulation. This section will present an in-depth look at the performance of each algorithm in a noise-free environment, as well as a look at the tracking performance and control signals in an environment with simulated measurement noise.

3.4.1 Extended Kalman Filter Approach

This algorithm implements both state estimation and inertial parameter estimation in the same Extended Kalman Filter. This is convenient in implementation as one only needs to keep track of one set of state variables in order to perform state estimation and parameter adaptation. However, it will be seen that the use of an EKF can be quite computationally intensive.

In implementing the EKF it is important to properly set the filter parameters $Q(t)$ and $R(t)$ in order to run the filter. Section 2.3.1 outlines the role of these matrices. Matrix $Q(t)$ represents the covariance of the process noise, while matrix $R(t)$ represents the covariance of the noise on the measurements. As well, it is important to initialize the error covariance matrix $P(t)$ to suitable values.

The simulations with this algorithm will examine its performance at steady-state as well as the peak trajectory errors in the initial transient portion of the response. By adjusting the measurement noise covariance matrix $R(t)$ it will be seen that this has a significant effect on peak tracking error. Simulations will be performed in cases free of noise as well as in cases with noise added to simulate 12-bit quantization.

Initial Simulation without Noise

To simulate the EKF initially, the filter parameters were set as $R(t) = 0.0001I$, $Q_1(t) = 0.01I$, $Q_2(t) = 0.0025I$, and $P(t_o) = \text{diag}\{0.0001, 0.0001, 0.0001, 0.0001, 0.01, 0.01\}$. These parameters used for simulation were chosen based on the values used in [4], except that the initial error covariance for the parameter estimates is set larger in this case. This is due to greater initial error on the parameter estimates in this simulation than in [4]. In simulation, the initial positions of the robot links were set to zero radians, and the initial velocities were set to zero rad/sec. The position and velocity state estimates of the EKF were also initialized to zero.

Initially, the simulation was performed without any noise in the system. Fig. 3.3 shows a plot of the desired and actual trajectories for link 1, and Fig. 3.4 shows a plot of desired and actually trajectories for link 2. Overall, the results for the simulation were good, with the exception of the peak tracking error in the transient period. Results for the tracking error are illustrated in Fig. 3.5. The maximum tracking error in this case is roughly 0.4 radians for link 1, and 0.1 radians for link 2. Fig. 3.6 shows the tracking error after 180 seconds. At this point the error is very small.

The maximum tracking error on link 1 is fairly significant in this case, and is due to large error on the observed velocity signals early in the transient period. Fig. 3.7 shows the error on the estimated position signals, and Fig. 3.8 shows the error on the estimated velocity signals. The error on the velocity estimates is clearly much larger than that of the position estimates. Since the controller makes use of the velocity error in generating the torque inputs to the robot, large error on the velocity estimate could affect the torques computed by the controller. However, the tracking error is quick to decrease and enter the range of error of the other algorithms. As well, the parameters are quick to converge using this approach. Fig. 3.9 shows the parameter

estimates over time.

Initial Simulation with Quantization Noise

When noise was added to the position measurements to simulate 12-bit quantization error, the results were similar to the noise-free case. Fig. 3.10 demonstrates the tracking error of the robot in this case. The maximum tracking error was roughly 0.4 radians for link 1, and 0.15 radians for link 2. An important note here is that when noise was introduced, the control signal that was generated remained fairly free of noise. The torques computed can be seen in Fig. 3.11. It will be seen that position measurement noise can lead to noisy control signals in other algorithms. This is an important fact for implementation, since a control signal as free of noise as possible is desired to drive the motors, so as not to excite high frequency and unmodeled dynamics.

Simulation Using Smaller $R(t)$, without Noise

The initial experiment yielded large peak tracking errors in both the noise-free and noisy simulations. This can be a hindrance in real implementation of an algorithm, as large peak tracking error could result in attempts to drive the manipulator to positions beyond its physical limits. As a result, a method of decreasing this peak tracking error was sought. One cause of this is the initially inaccurate estimates of the inertial robot parameters. However, a solution not based on more precise parameter knowledge was desired. Various simulations were performed in order to determine the most effective means of decreasing the peak error. It was found that decreasing the values of the diagonals in the measurement noise covariance matrix $R(t)$ was quite effective. The value of $R(t)$ was decreased from $1 \times 10^{-4} I$ in the initial experiment to

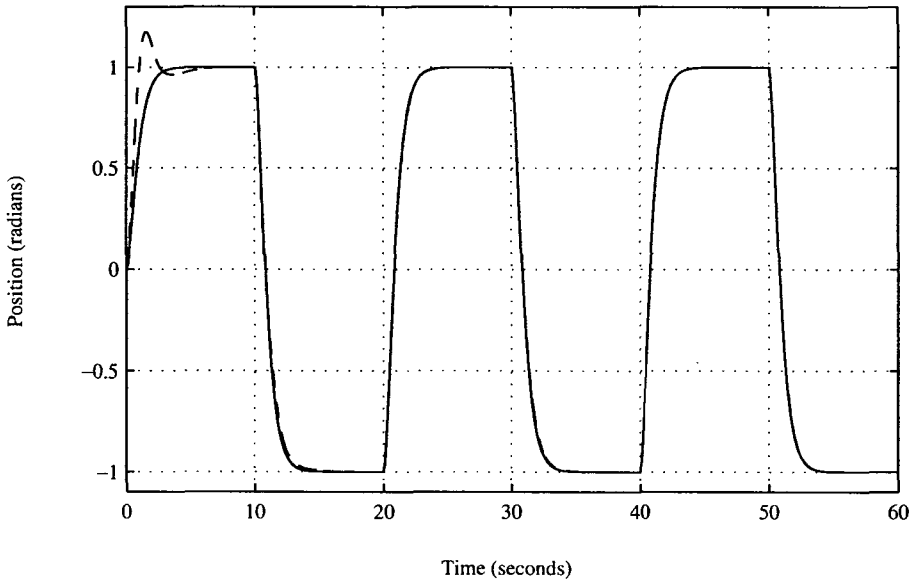

Figure 3.3: Desired (solid) and actual (dashed) trajectories for link 1 of the simulated robot using Gourdeau and Schwartz's EKF with no noise, initial experiment.

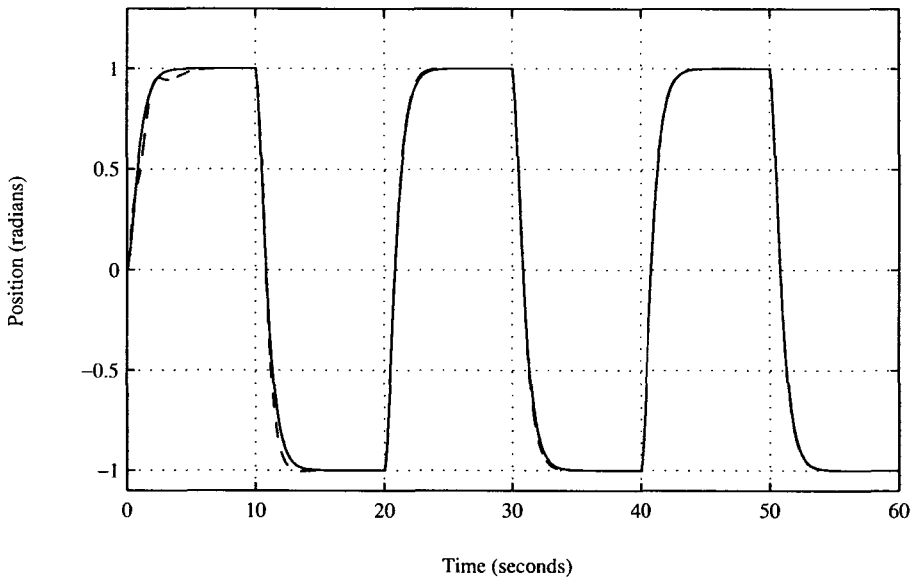

Figure 3.4: Desired (solid) and actual (dashed) trajectories for link 2 of the simulated robot using Gourdeau and Schwartz's EKF with no noise, initial experiment.

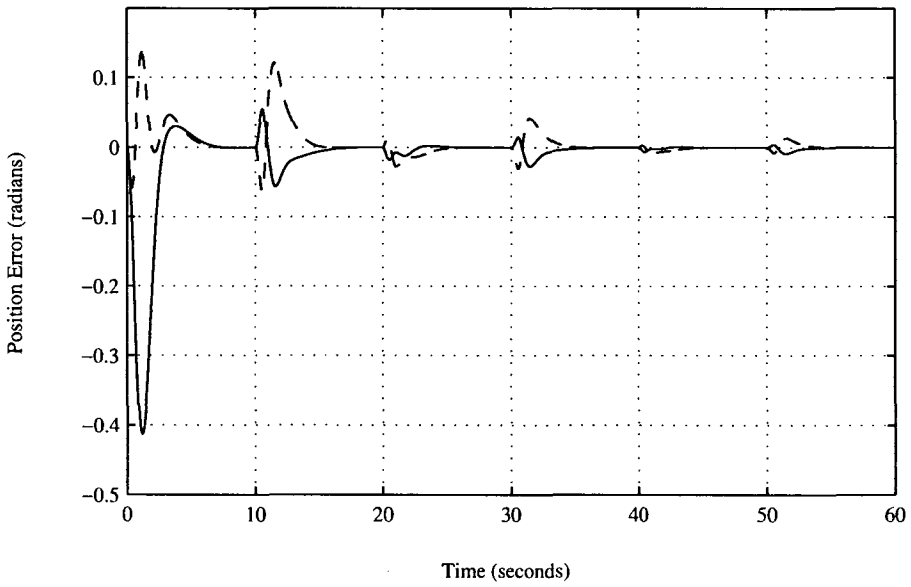

Figure 3.5: Position error for link 1 (solid) and link 2 (dashed) using Gourdeau and Schwartz's EKF in simulation with no noise, initial experiment.

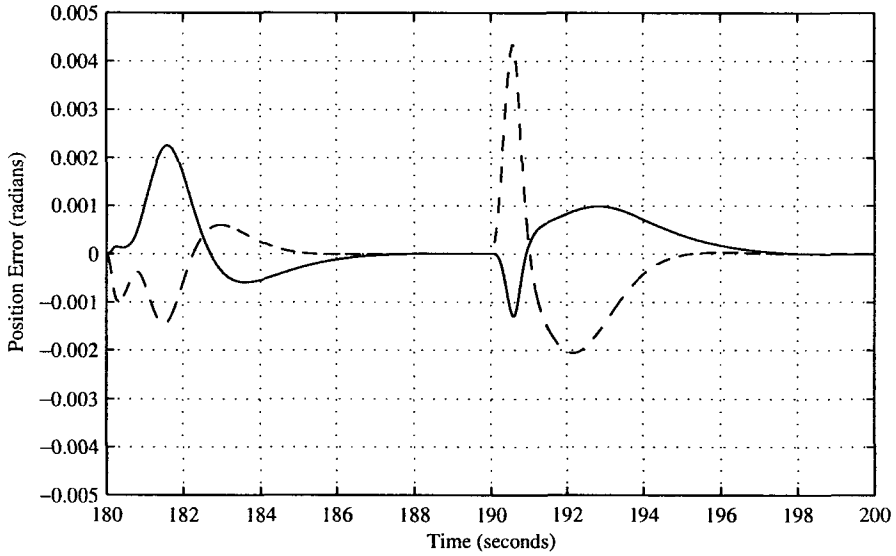

Figure 3.6: Position error for link 1 (solid) and link 2 (dashed) after 180 seconds using Gourdeau and Schwartz's EKF in simulation with no noise, initial experiment.

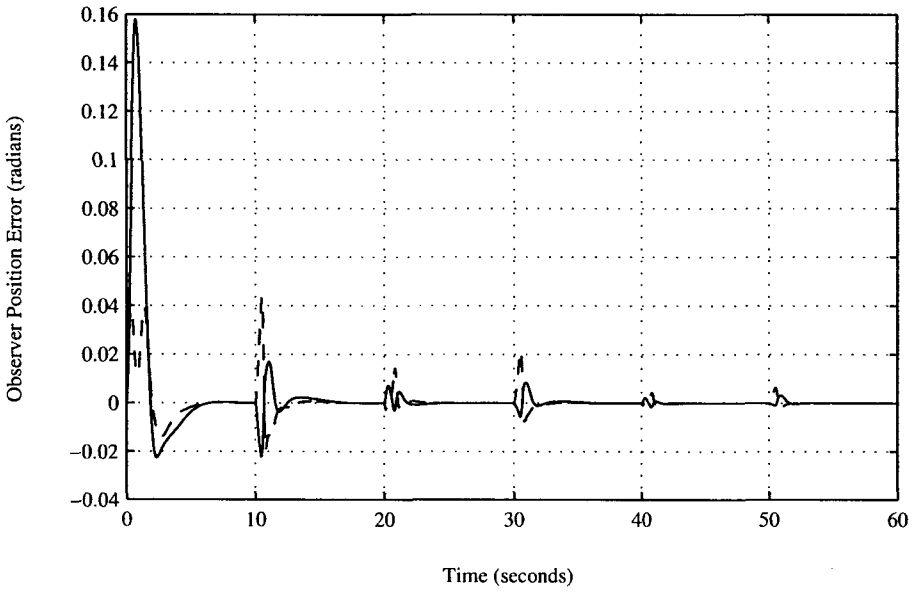

Figure 3.7: Error on the positions estimated using Gourdeau and Schwartz's EKF in simulation with no noise for link 1 (solid) and link 2 (dashed), initial experiment.

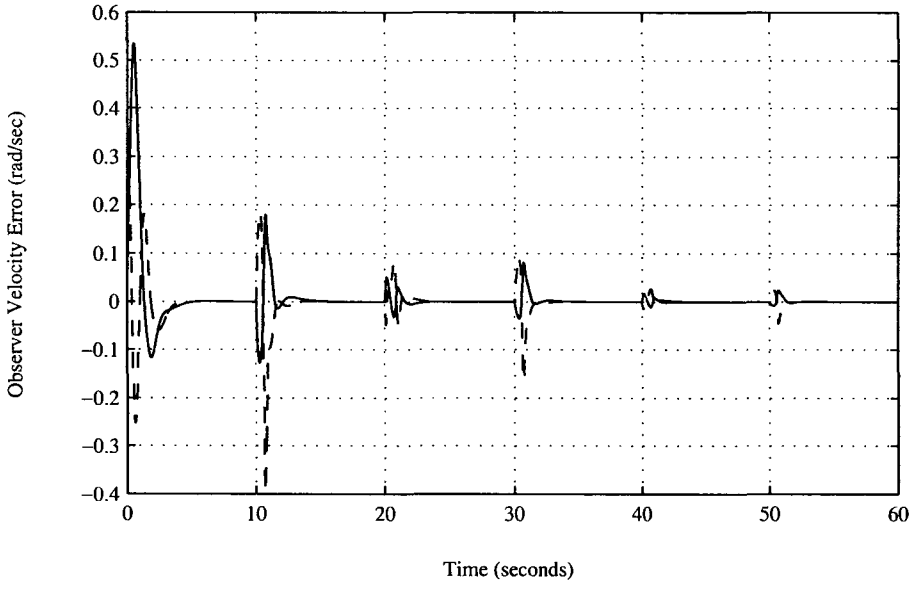

Figure 3.8: Error on the velocities estimated using Gourdeau and Schwartz's EKF in simulation with no noise for link 1 (solid) and link 2 (dashed), initial experiment.

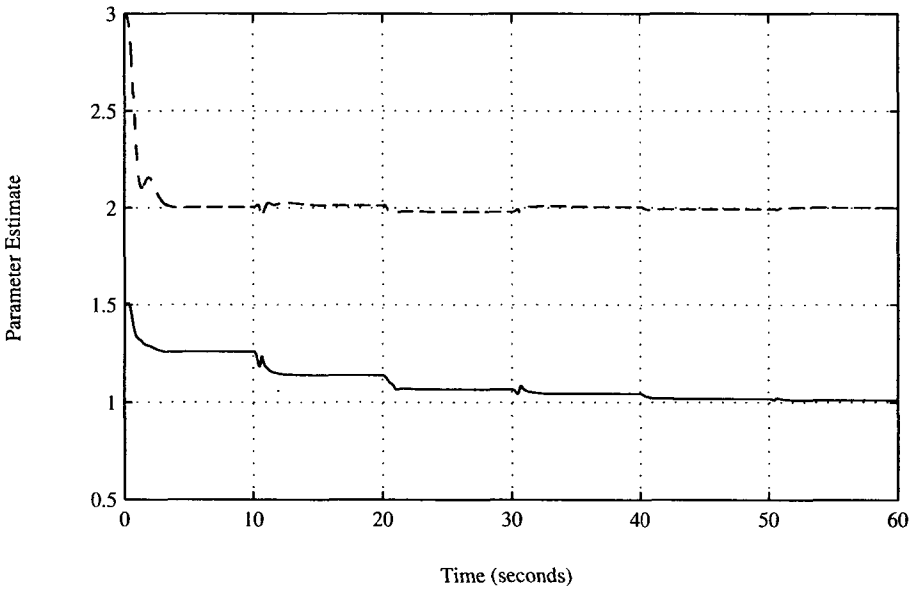

Figure 3.9: Parameter estimates over time for θ_1 (solid), and θ_2 (dashed) using Gourdeau and Schwartz's EKF in simulation with no noise, initial experiment.

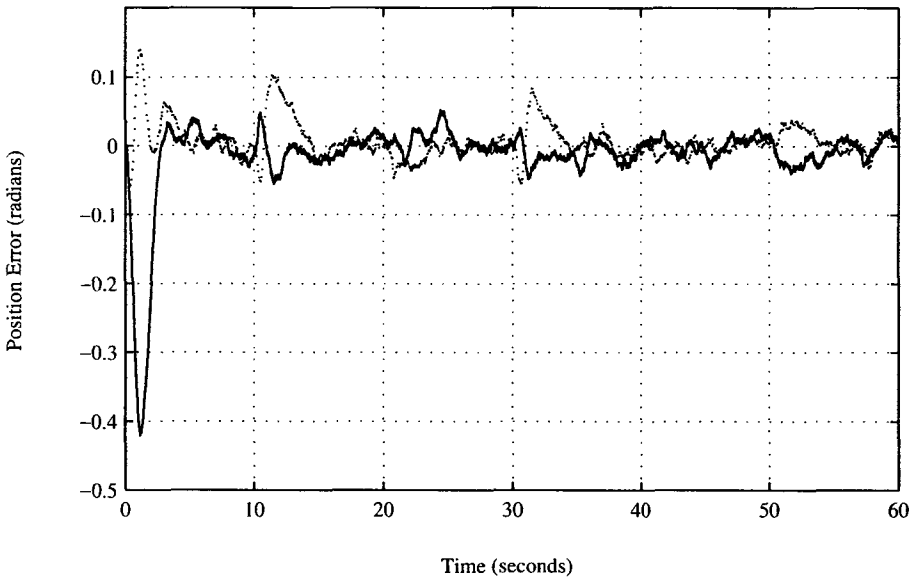

Figure 3.10: Position error for link 1 (solid) and link 2 (dashed) using Gourdeau and Schwartz's EKF in simulation with quantization noise, initial experiment.

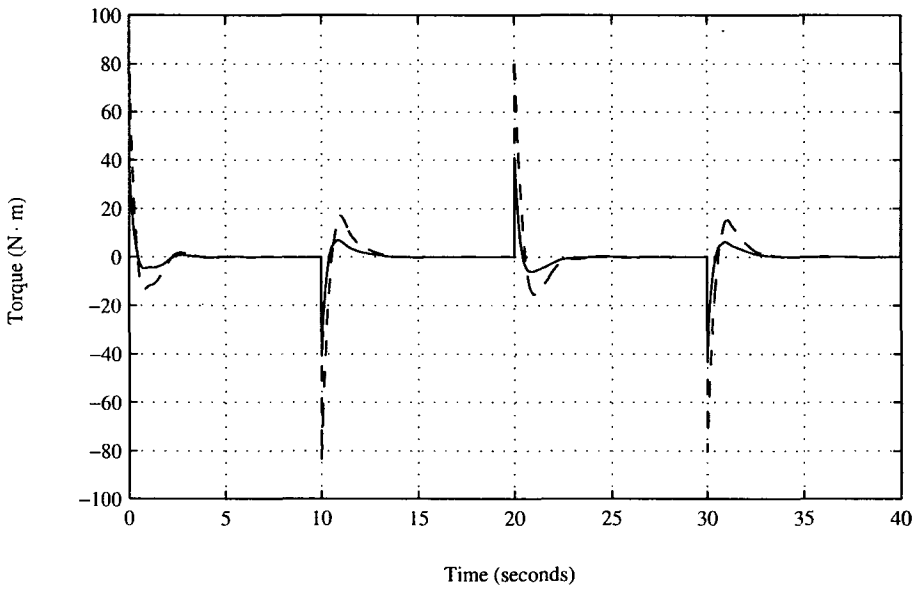

Figure 3.11: Computed torques used to drive the simulated robot for link 1 (dashed) and link 2 (solid) using Gourdeau and Schwartz's control algorithm with quantization noise.

$1 \times 10^{-7} I$. This has the effect of increasing the contribution of the measurements in the state estimate equation (2.23). Decreasing $R(t)$ suggests that there is less noise on the measurements, causing them to have a larger impact on the state estimate. If the measurements were quite noisy and $R(t)$ were large, the state estimate update would rely more on the propagation of the state estimate based on the assumed dynamic model than on the measurements.

Fig. 3.12 shows the tracking error for the simulation with no noise and $R(t) = 1 \times 10^{-7} I$. The tracking error is much smaller here than in the case of Fig. 3.5, which had a larger $R(t)$. The larger gain on the measurement term that results from a smaller $R(t)$ helps to speed up the convergence of the state estimates produced by the EKF. Fig. 3.13 shows the error on the observed velocity signals for the current simulation. When compared with Fig. 3.8 it can be seen that the error on the velocity estimate is much smaller when the smaller values for $R(t)$ are used.

Simulation Using Smaller $R(t)$, with Quantization Noise

Another simulation was performed with the same parameters, however measurement noise was introduced on the position measurements to simulate the effect of a 12-bit quantizer. While the tracking error in the transient response did not peak to a large value, the ability of the robot to track the trajectory with this setup is fairly poor. Fig. 3.14 shows the tracking performance of this setup for the first 60 seconds of the simulation. There are peaks in the tracking error throughout that time which are undesirable. In this case $R(t)$ is set too small to represent the noise on the measurements. As a result, the filter depends too much on the measurements for updating the state. It is interesting to examine the actual and estimated velocity signals. Fig. 3.15 shows a plot of the actual and estimated velocity signals for link

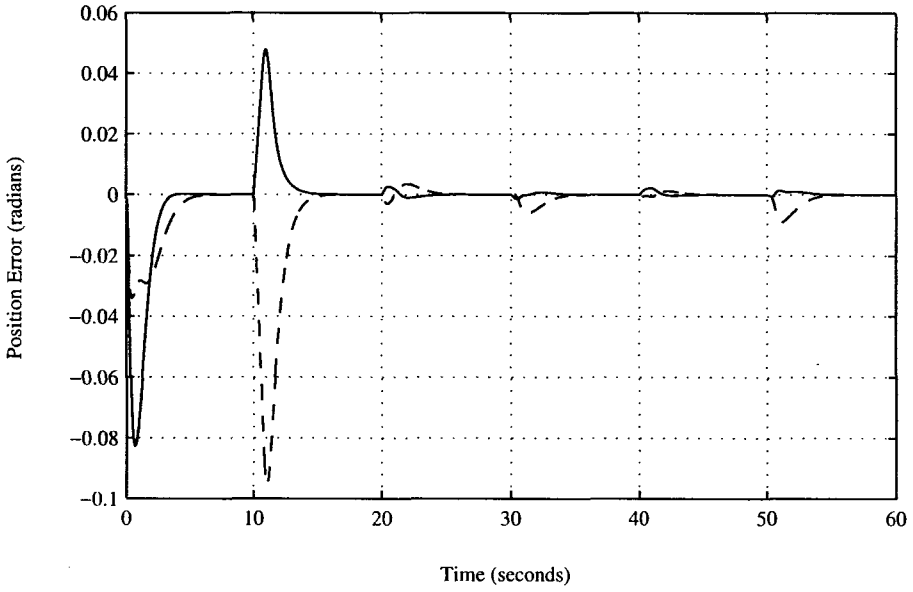

Figure 3.12: Position error for link 1 (solid) and link 2 (dashed) using Gourdeau and Schwartz's EKF in simulation with no noise, smaller $R(t)$.

1. It appears from this plot that there is in fact more noise on the velocity estimate than on the measured velocity. This is due to the low value of $R(t)$, which does not accurately represent the covariance of the measurement noise.

These results demonstrate that, while a low value of $R(t)$ may improve filter performance, one has to be careful in setting the measurement noise covariance to reflect accurately the level of noise in the system. If improperly set, as in this case, more emphasis may be placed on a noisy measurement, leading to overly noisy state estimates.

Results of the Simulations

Several simulations were performed using Gourdeau and Schwartz's algorithm to investigate the tracking performance. Cases free of noise as well as cases containing

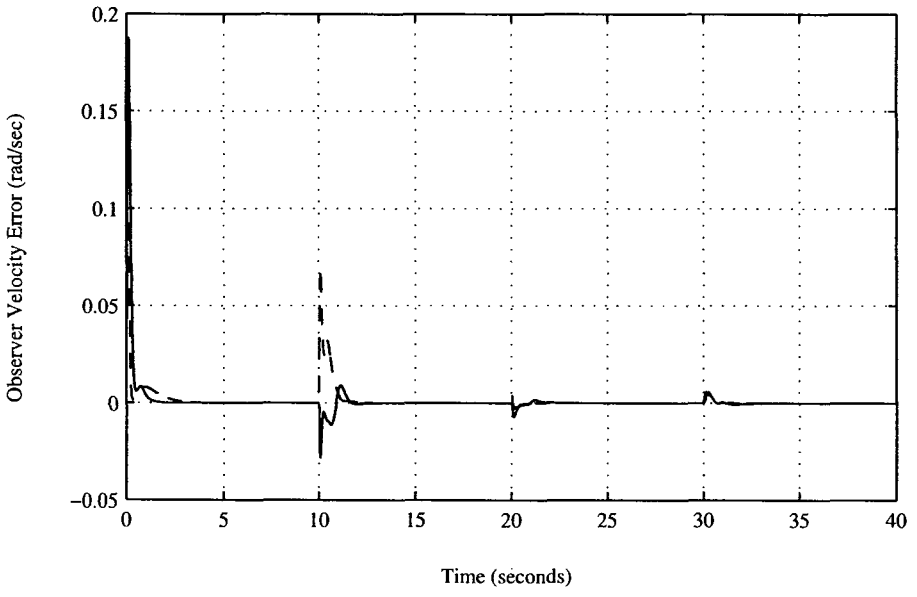

Figure 3.13: Error on the velocities estimated using Gourdeau and Schwartz's EKF in simulation with no noise for link 1 (solid) and link 2 (dashed), smaller $R(t)$.

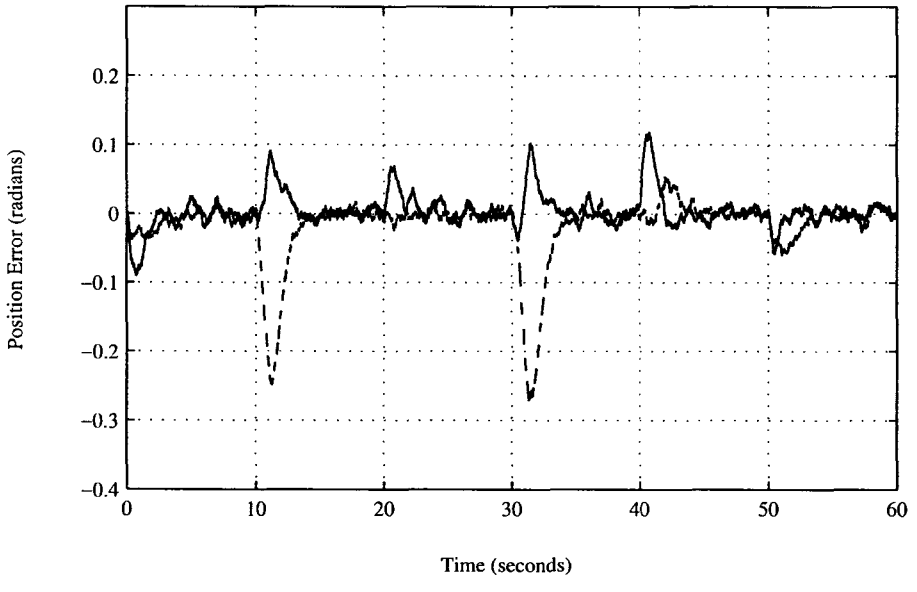

Figure 3.14: Position error for link 1 (solid) and link 2 (dashed) using Gourdeau and Schwartz's EKF in simulation with quantization noise, smaller $R(t)$.

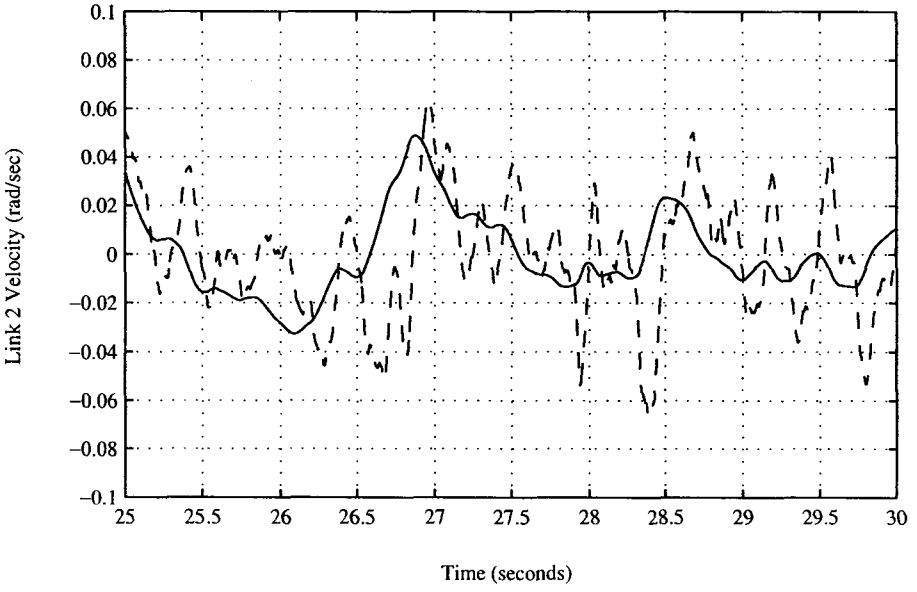

Figure 3.15: Actual (solid) and estimated (dashed) velocity signal for link 1 using Gourdeau and Schwartz's EKF in simulation with quantization noise, smaller $R(t)$.

quantization noise were used. It was seen with the initial simulation that peak tracking error was quite large, on the order of 0.4 radians for link 1. When the errors on the state estimates were examined, it was seen that the peak error on the velocity estimates was quite large. This large estimation error implies initially inaccurate estimates of the states, which could lead to computing of inaccurate control torques by the control law. These control torques would then contribute to larger tracking error. This behaviour was seen in both the case free of noise and the case with quantization noise.

In order to decrease the peak tracking error, simulations were performed using smaller values for $R(t)$. This has the effect of increasing the contribution of the measurements in the state update equation. The diagonal entries of $R(t)$ were decreased by 3 orders of magnitude. In the simulation without noise, this change had the de-

sired effect. The peak tracking error decreased significantly and remained below 0.1 radians. As well, the peak error on the velocity estimates was much smaller in this simulation. However, when the simulation was repeated with quantization noise on the position measurements, the results were not as desirable. The plot of tracking performance showed periods of peaking in the tracking error throughout the simulation. It was determined that such a low value for $R(t)$ was not representative of the actual noise in the system when quantization noise was added, and too much emphasis was being placed on the noisy measurements. The actual and estimated velocities were compared and it was found that the estimated velocity actually contained more noise than the true velocity. This result demonstrates the importance of correctly determining the noise processes in the dynamic system and setting the EKF parameters appropriately. The initial simulation, with larger values for $R(t)$, yielded better steady-state tracking error in the presence of noise. However, the peak tracking error in the case with smaller values for $R(t)$ was less than the peak error with larger values for $R(t)$.

3.4.2 High-Gain Observer Approach

The algorithm proposed by Lee and Khalil [5] was successfully implemented in simulation. Several simulation parameters must be determined before the simulations are run. The observer parameters L_1 and L_2 were both set equal to $I_{2\times 2}$ to ensure that the matrix $\bar{A}$ is Hurwitz. The parameter ϵ which is directly related to observer gain was set to $\epsilon = 0.001$, as per one of the simulations in [5]. Initial position and velocity errors were both set to zero, meaning that the robot starts at rest in its zero position.

The simulations performed with this algorithm will examine its performance with different values of the adaptation gain, Γ. Simulations will be run in cases free of

noise and in cases with noise to simulate the effects of 12-bit quantization.

Initial Simulation without Noise

In the first simulation, the adaptation gain Γ was set to $I_{p\times p}$ for unity adaptation gain. This simulation was performed without noise. Plots of the desired and actual trajectories can be seen in Fig. 3.16 and Fig. 3.17 for links 1 and 2 respectively. Fig. 3.18 illustrates the tracking error of the algorithm without any noise during the first 60 seconds. It reaches a peak error of roughly 0.26 radians on link 1, and 0.1 radians on link 2. A plot of the tracking error at steady-state is given in Fig. 3.19. It is seen that the tracking errors converge to very small values at steady state. Compared to the tracking performance of Gourdeau and Schwartz's algorithm (seen in Fig. 3.5 and Fig. 3.6) it can be seen that the ability to track the trajectory at steady state is similar. Both algorithms exhibit large peak error, although it is larger in the case of Gourdeau and Schwartz.

The large peak error in this algorithm can be attributed to the relatively inaccurate initial estimates of the robot parameters, and a fairly small adaptation gain, which slows parameter convergence. The inertial parameter estimates over time can be seen in Fig. 3.20. It is seen here that the parameters are relatively slow to converge. A small adaptation gain was chosen in order to compare this algorithm with simulations performed using Craig's algorithm with our observer, presented in Section 3.4.3, which requires a small adaptation gain to prevent the simulation from diverging.

Plots of the observer error for this simulation are given in Fig. 3.21 and Fig. 3.22 for position error and velocity error, respectively. The benefit of the high-gain observer is seen here. The observer error remains quite small and rapidly reaches a steady-state value. When one compares this observer error to Fig. 3.7, Fig. 3.8, and

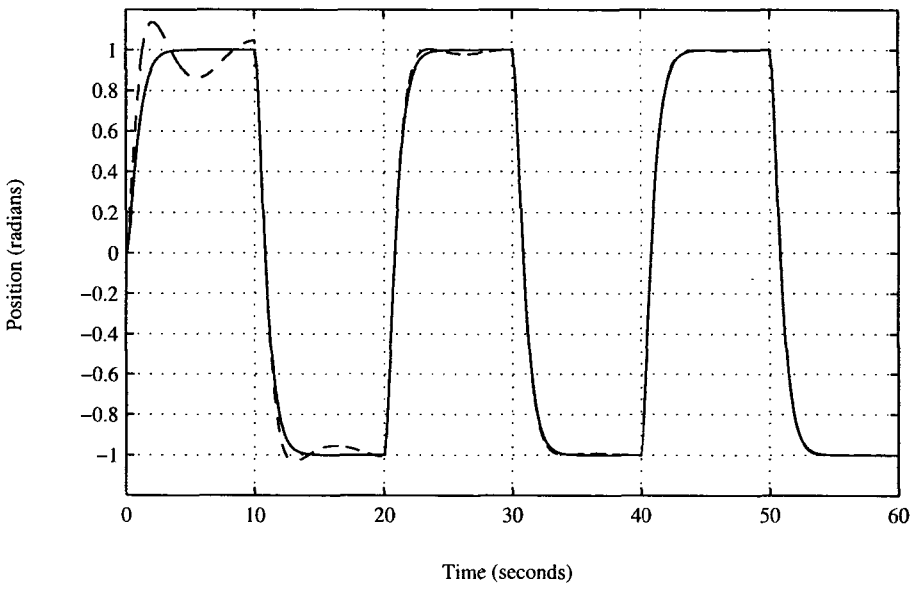

Figure 3.16: Desired (solid) and actual (dashed) trajectories for link 1 of the simulated robot using Lee and Khalil's algorithm with no noise, $\Gamma = I$.

Fig. 3.13, which show the performance of Gourdeau and Schwartz's EKF, it can be seen that the high-gain observer performs much better in terms of observer error. This has the benefit that the system with a high-gain observer will rapidly approach the performance of a full-state feedback controller.

Initial Simulation, with Quantization Noise

Adding noise to the position measurements to simulate a 12-bit resolver resulted in tracking performance comparable to the simulation without noise, as seen in Fig. 3.23. Here the maximum tracking error was roughly 0.3 radians on link 1, and 0.14 radians on link 2. However, due to the high-gain observer, velocity estimates were quite noisy, and this resulted in a very noisy control signal that could make implementation on a real platform difficult. Fig. 3.24 shows a plot of the torques computed for this

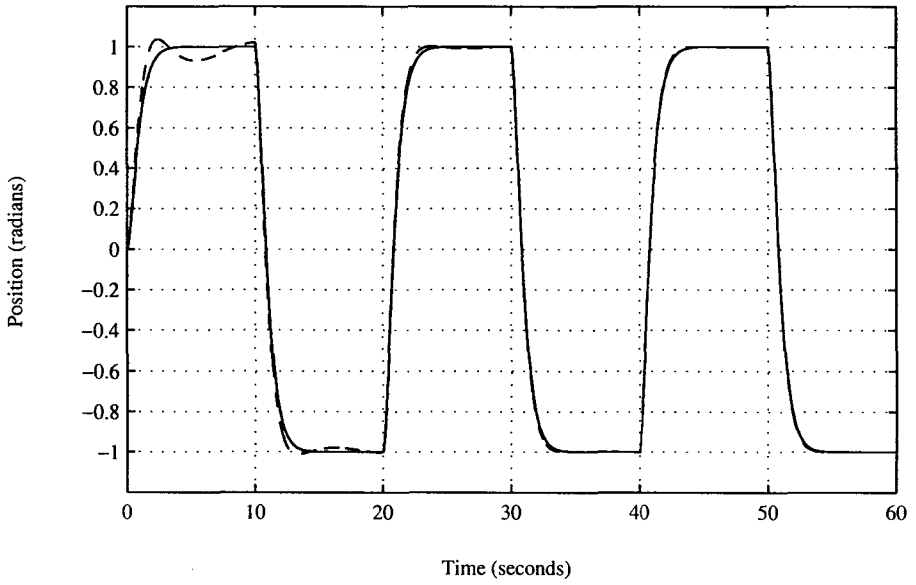

Figure 3.17: Desired (solid) and actual (dashed) trajectories for link 2 of the simulated robot using Lee and Khalil's algorithm with no noise, $\Gamma = I$.

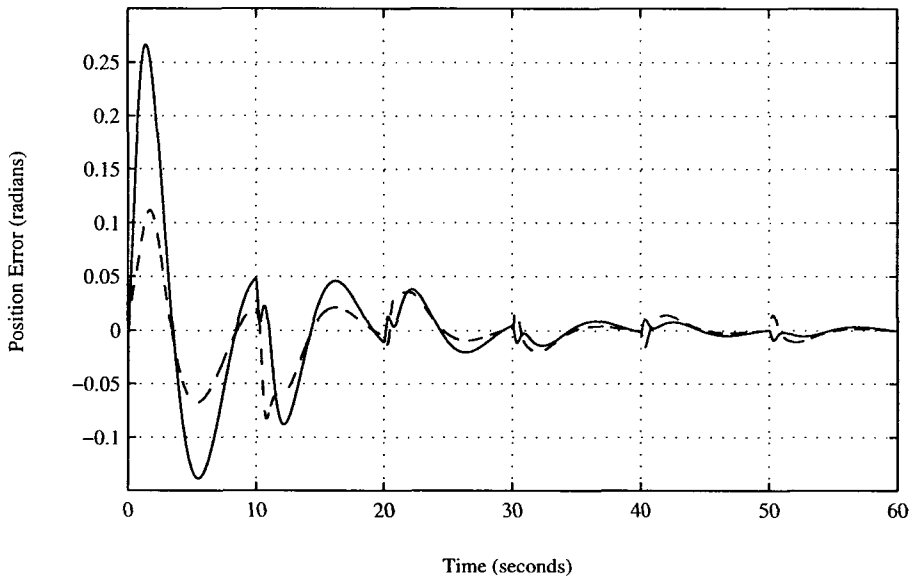

Figure 3.18: Position error for link 1 (solid) and link 2 (dashed) using Lee and Khalil's algorithm in simulation with no noise, $\Gamma = I$.

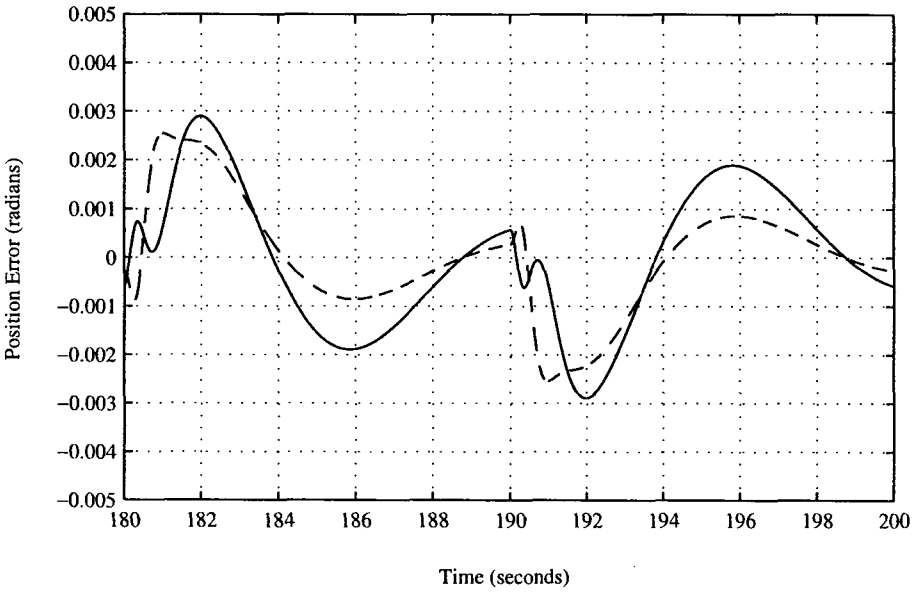

Figure 3.19: Position error for link 1 (solid) and link 2 (dashed) after 180 seconds using Lee and Khalil's algorithm in simulation with no noise, $\Gamma = I$.

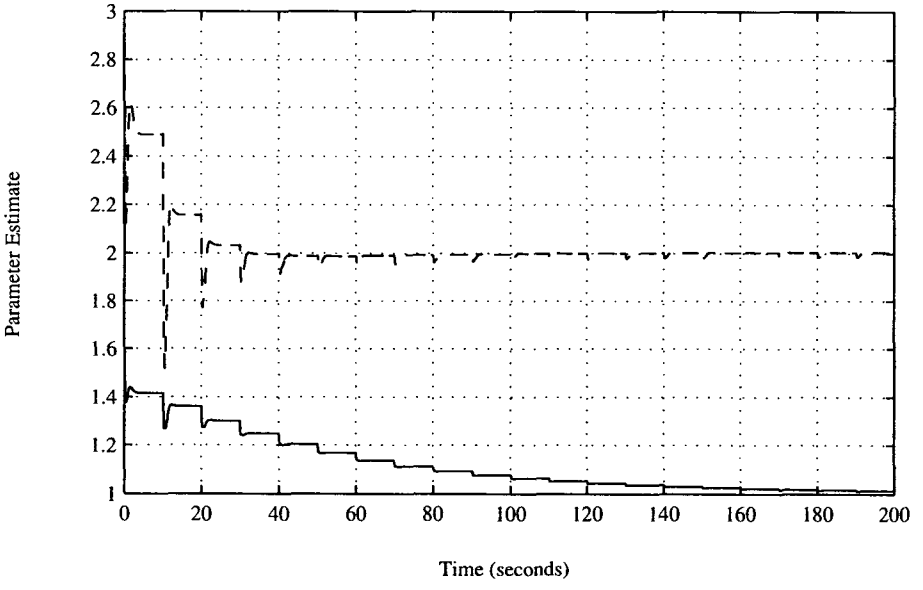

Figure 3.20: Parameter estimates over time for θ_1 (solid), and θ_2 (dashed) using Lee and Khalil's algorithm in simulation with no noise, $\Gamma = I$.

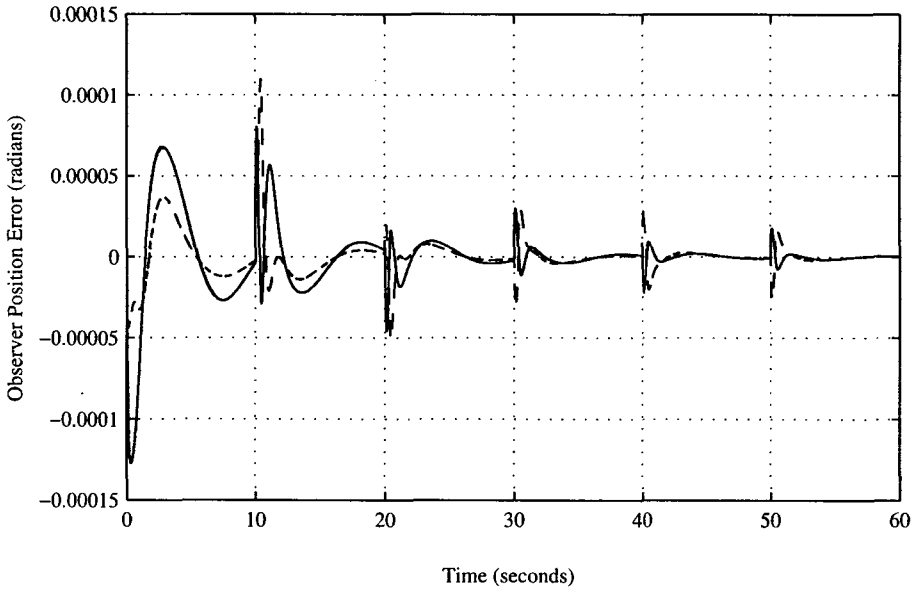

Figure 3.21: Error on the estimate of tracking error e using Lee and Khalil's algorithm in simulation with no noise for link 1 (solid) and link 2 (dashed), $\Gamma = I$.

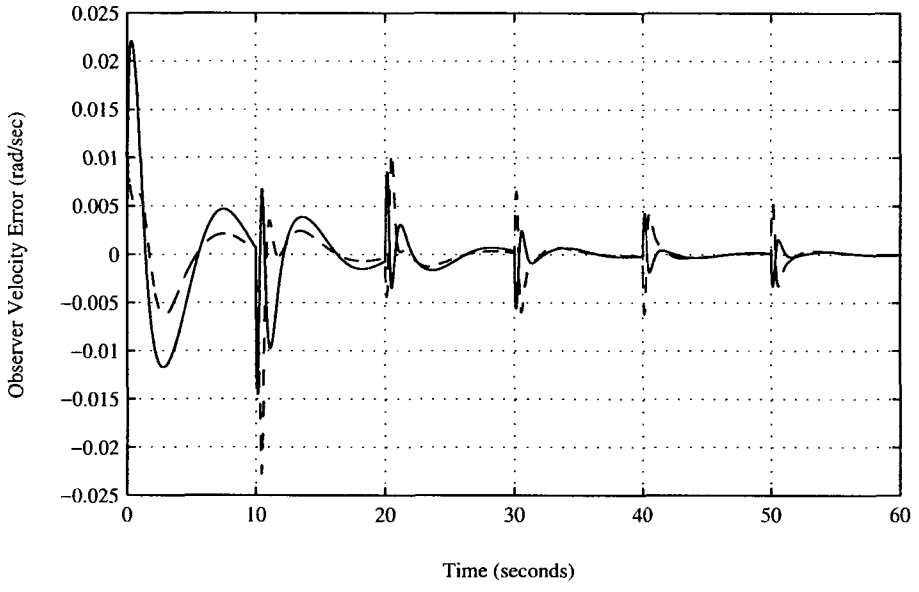

Figure 3.22: Error on the estimates of velocity error $\dot{e}$ using Lee and Khalil's algorithm in simulation with no noise for link 1 (solid) and link 2 (dashed), $\Gamma = I$.

simulation with measurement noise. One can see that these torques, when compared to those computed in Gourdeau and Schwartz's algorithm in Fig. 3.11, contain more noise. This is due to the high gain of the observer. The noise on the position measurements is amplified significantly, leading to very noisy state estimates. The noisy state estimates are used in the control law and produce torque signals containing high amounts of noise.

Simulation Using Larger Adaptation Gain, without Noise

In order to better gauge the performance of the algorithm by itself, the adaptation gain was increased to $\Gamma = 50I$, and the results for tracking error are quite good. Fig. 3.25 shows the tracking error for the first 60 seconds. When compared with Fig. 3.18, it can be seen that the tracking error in this case peaks at a significantly smaller value. This indicates better tracking performance. As well, the tracking error at steady-state was found to be lower with a larger adaptation gain. The much quicker adaptation of parameters resulted in this improved tracking performance. A plot of parameter estimates over time for this case is given in Fig. 3.26. It is apparent that the parameters reach a steady-state much more rapidly, which is desirable behaviour. However, when 12-bit resolver noise was added to the position measurements, with this value for Γ the simulation diverged. This shows excellent theoretical performance, but performance that may not be achievable in practice.

Results of the Simulations

Through simulation, Lee and Khalil's algorithm was examined to determine its tracking performance. Initially, a simulation was performed without any added noise for $\Gamma = I$. The tracking error at steady state was quite small, although the parameter

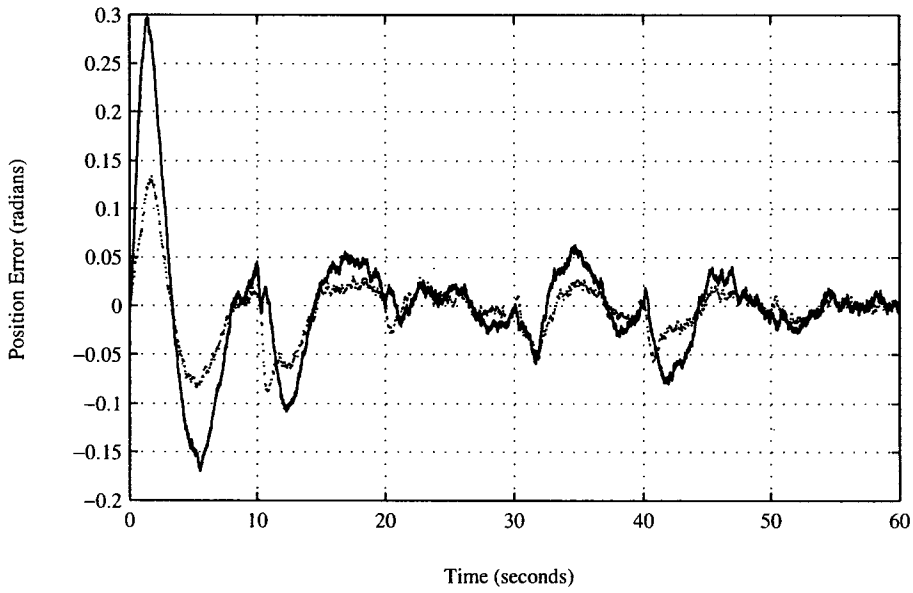

Figure 3.23: Position error for link 1 (solid) and link 2 (dotted) using Lee and Khalil's algorithm in simulation with quantization noise, $\Gamma = I$.

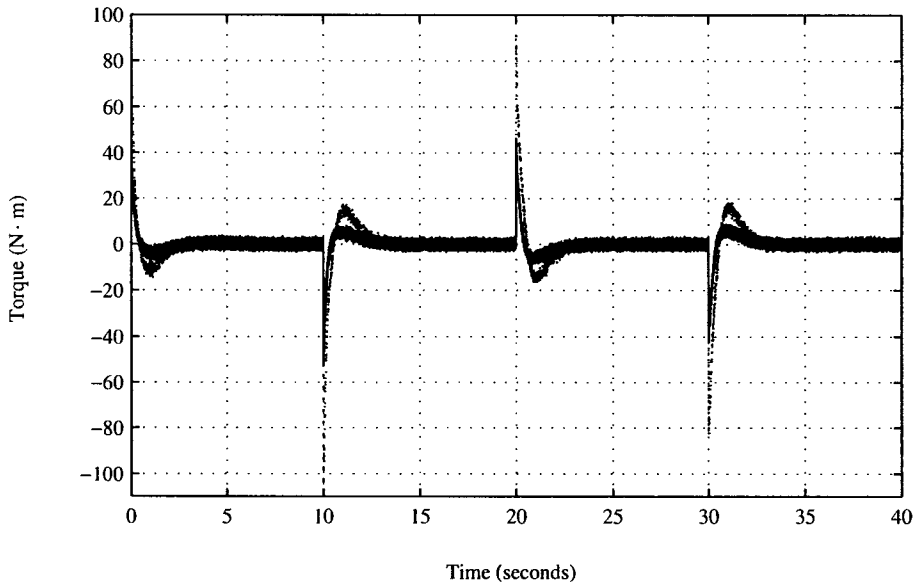

Figure 3.24: Computed torques used to drive the simulated robot for link 1 (dashed) and link 2 (solid) using Lee and Kahlil's algorithm with quantization noise.

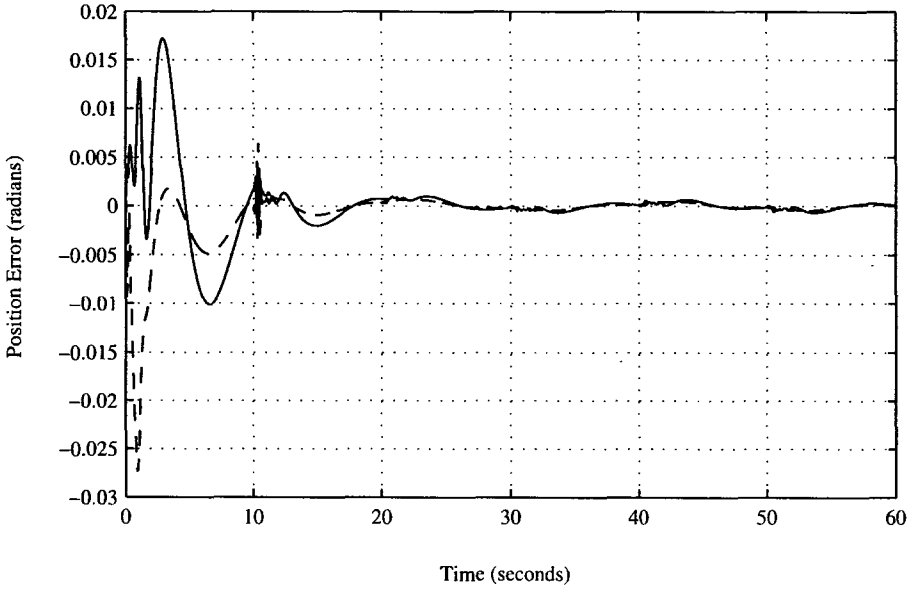

Figure 3.25: Position error for link 1 (solid) and link 2 (dashed) using Lee and Khalil's algorithm in simulation with no noise, $\Gamma = 50I$.

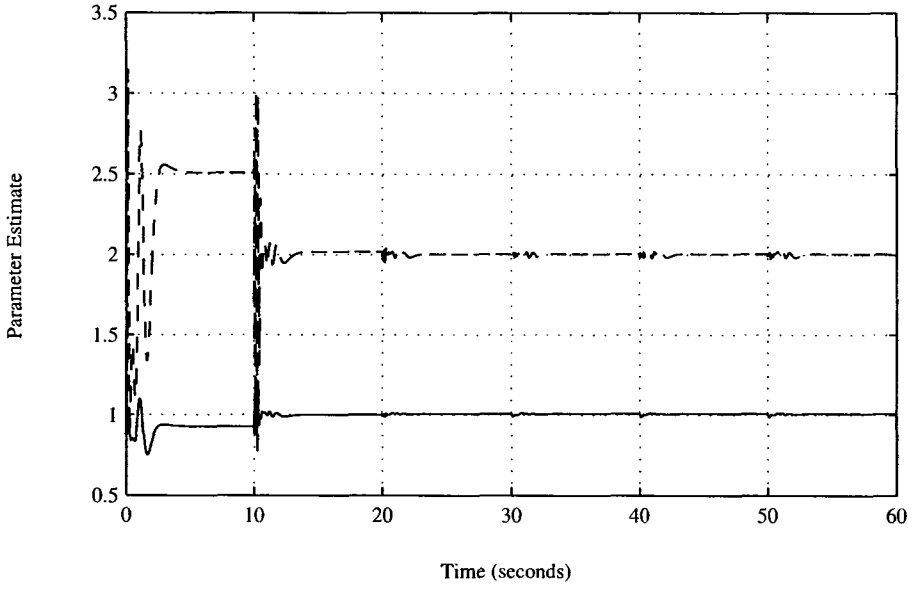

Figure 3.26: Parameter estimates over time for θ_1 (solid), and θ_2 (dashed) using Lee and Khalil's algorithm in simulation with no noise, $\Gamma = 50I$.

estimates took nearly 180 seconds to reach a steady state. This is due to the low value used for the adaptation gain. The observer error was very small for both the position and velocity states. This is due to the high observer gains which cause rapid convergence of the observer error.

The same simulation was repeated with quantization noise on the position measurements. It was seen that addition of noise has an effect on tracking performance, however the position error remains quite small. When examining the torque signals that were produced by the controller, it was seen that significant noise is transmitted through the system by the high-gain observer. This could lead to difficulty in implementation, as a control signal with significant high frequency content could excite the high frequency dynamics of a system.

Another simulation was run with a much larger adaptation gain of $\Gamma = 50I$. A noticeable increase in tracking performance was observed. The peak tracking errors were much smaller when using the larger adaptation gain, as well as the steady-state error. The inertial parameter estimates had reached a steady state after roughly 10 seconds. However, when the simulation was repeated for the case with quantization noise, the algorithm diverged. This fact and the observation that the control torques are quite noisy with the addition of quantization noise to the position measurement suggest that while this algorithm shows excellent theoretical performance, it may be difficult to implement in practice.

3.4.3 Addition of a Linear Observer to Craig's Algorithm

Craig, Hsu, and Sastry's algorithm with the addition of a linear observer was implemented in simulation for comparison with the other algorithms studied. This method was very straightforward to implement, owing to the use of a linear observer based

on the dynamics of a double integrator.

With this algorithm there are no clear guidelines for setting of the observer gains. One must ensure that the observer poles are fast enough to not have an effect on the convergence of the system. It was determined through simulation that if the observer gains are not set above some threshold, the observer error will remain large at steady state and the algorithm will exhibit very poor tracking performance.

The simulations with this algorithm will examine the effect of varying the observer gains on tracking performance. Simulations will be performed in cases free of noise as well as cases with quantization noise on the position measurements.

Initial Simulation, without Noise

In the initial simulation, the poles of the observer were placed at $s = -10$. These poles are critically damped, and are five times faster than the poles of the linearized closed-loop system. The closed-loop system has feedback gains of $K_p = K_v = 4I$, placing both of its poles at $s = -2$. In this simulation the adaptation gain was set to $\Gamma = I$. Through simulation it was determined that values of the adaptation gain larger than $\Gamma = I$, while not causing divergence, yielded unacceptable performance of the robot with respect to tracking error and parameter estimates.

It was found that this configuration of the observer poles yielded large trajectory tracking errors. Fig. 3.27 shows the tracking error for links 1 and 2. The tracking error remains large at steady-state, with peak values of roughly 0.4 radians for link 1, and 0.6 radians for link 2. When one examines the parameter estimates given in Fig. 3.28, it can be seen that there is significant oscillation in the parameter estimates, and they converge to values that are not near the true values.

The poor tracking and the incorrect parameter estimates are likely due to the poor

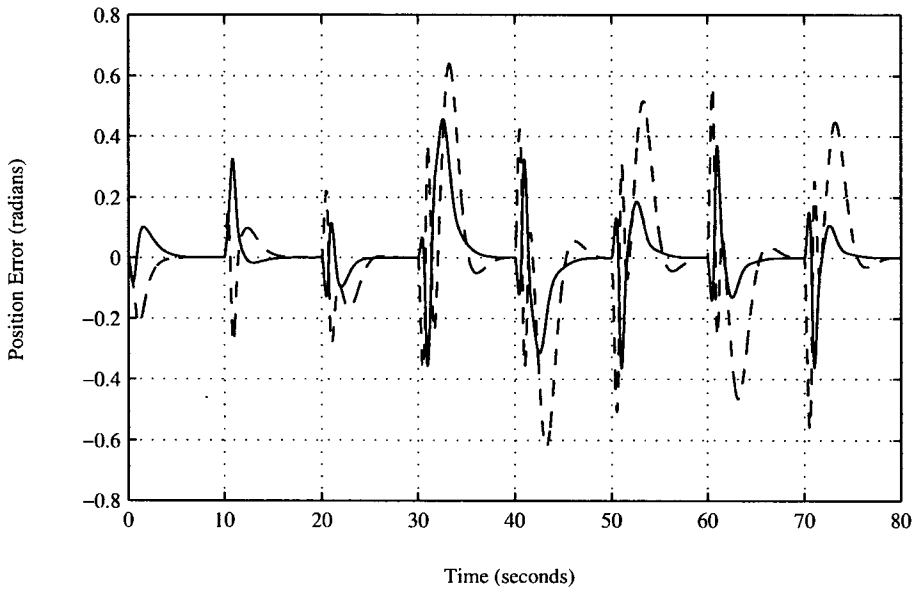

Figure 3.27: Position error for link 1 (solid) and link 2 (dashed) using Craig's algorithm with a linear observer with no noise, $K_1 = 20$, $K_2 = 100$.

performance of the observer in this case. Fig. 3.29 shows the observer error for the position states after 140 seconds. The error on the link 1 position estimate reaches values of roughly 0.05 radians and the error on the link 2 position estimate reaches values of roughly 0.1 radians, both at steady-state. The error on the velocity estimates is also quite large. The velocity estimate error for link 1 reaches roughly 1 rad/sec at steady-state, while the velocity estimate error for link 2 reaches roughly 2 rad/sec. It is likely that this large observer error is contributing to the large tracking error. The control law will receive position and velocity estimates that contain significant error with respect to the true values of those states. As a result, the torques computed by the control law will not be correct in order to have the robot track the desired trajectory. The incorrect torque signals, then, will lead to large tracking error.

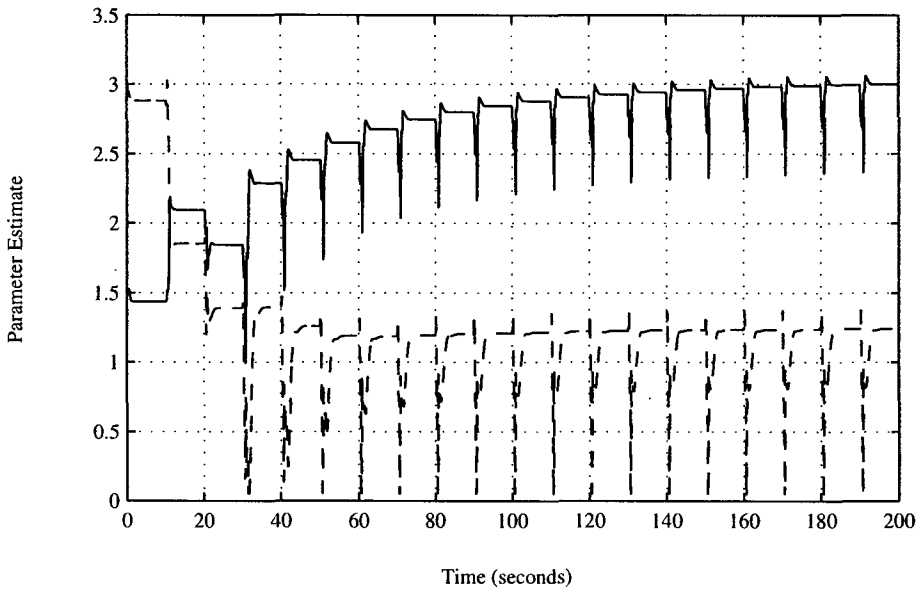

Figure 3.28: Parameter estimates over time for θ_1 (solid), and θ_2 (dashed) using Craig's algorithm with a linear observer with no noise, $K_1 = 20$, $K_2 = 100$.

Simulation Using Larger Observer Gains, without Noise

A second simulation was performed with all parameters except the observer gains set the same as the previous simulation. In this case the observer gains were set to $K_1 = 20$, and $K_2 = 500$. This has the effect of moving the two observer poles off of the real axis in the s-plane and making them underdamped. This set of gains was chosen to examine the effect of underdamped observer poles on the system. With these gains, the observer poles are placed at $s = -10 \pm 20j$. This yields an undamped natural frequency of $\omega_n = 22.36$ rad/sec and a damping ratio of $\zeta = 0.45$, versus $\omega_n = 10$ rad/sec and $\zeta = 1$ in the previous simulation.

When this simulation was run, the performance of the algorithm was significantly improved. Fig. 3.30 shows the tracking error for the first 60 seconds of this simulation. In this case, the tracking error peaks at roughly 0.17 radians for link 1, and 0.15

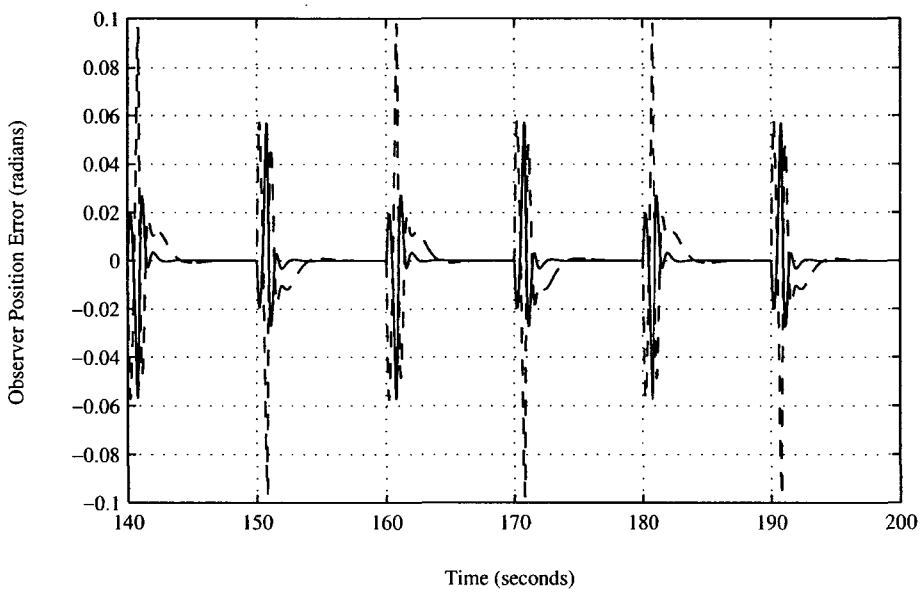

Figure 3.29: Error on the position estimates after 140 seconds using Craig's algorithm with an observer with no noise for link 1 (solid) and link 2 (dashed), $K_1 = 20$, $K_2 = 100$.

radians for link 2. This is significantly less than the tracking error of the previous simulation. The tracking error at steady-state is shown in Fig. 3.31. Here, the tracking error is on the order of 1×10^{-3} radians. The inertial parameter estimates converge fairly rapidly to values very close to the true values of the parameters. Fig. 3.32 shows the inertial parameter estimates over time. After roughly 30 seconds the parameter estimates reached a steady state.

In order to examine the effect of increasing the observer gain, plots of the observer error are given in Fig. 3.33 and Fig. 3.34, for position and velocity respectively. When comparing the error on the position estimates with the previous simulation (seen in Fig. 3.29), it can be seen that the peak error is much smaller with the increased observer gain. As well, the observer errors converge to very small values at steady state. The fact that the estimated positions and velocities are quick to approach the true values of the states helps to increase the tracking performance of the algorithm. The control law receives more accurate state estimates and as a result computes the torques more accurately.

This simulation shows that it is important to have an observer that will be able to accurately estimate the system states, while ensuring that the error on the state estimates converges to a small value with reasonable speed.

Simulation Using Larger Observer Gains, with Quantization Noise

The previous simulation with underdamped observer poles was repeated, this time with noise on the position measurements to simulate 12-bit quantization. The tracking error that resulted from this noisy simulation did not change significantly from the case free of noise. Fig. 3.35 shows the tracking error for the first 80 seconds of the simulation. In this case, the maximum tracking error was roughly 0.17 radians for

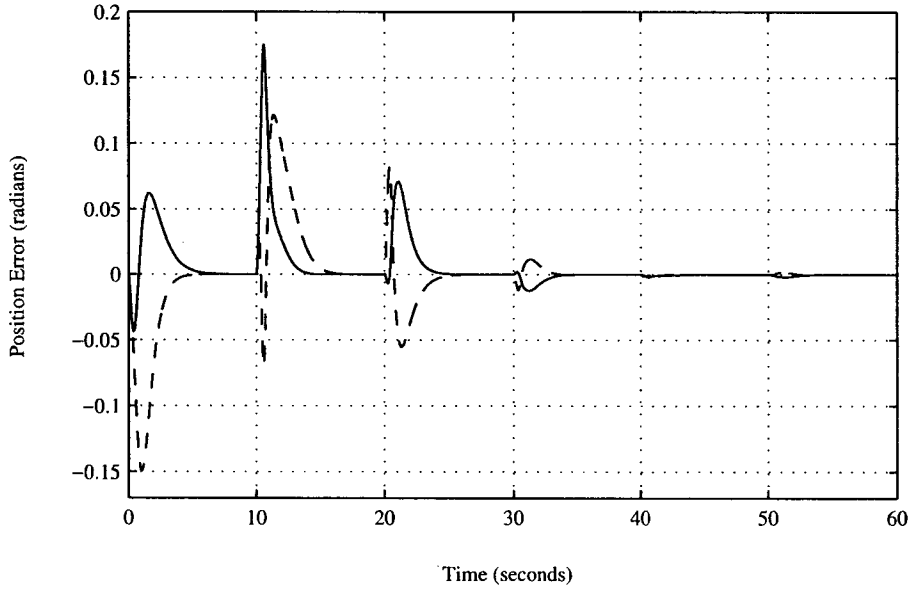

Figure 3.30: Position error for link 1 (solid) and link 2 (dashed) using Craig's algorithm with a linear observer with no noise, $K_1 = 20$, $K_2 = 500$.

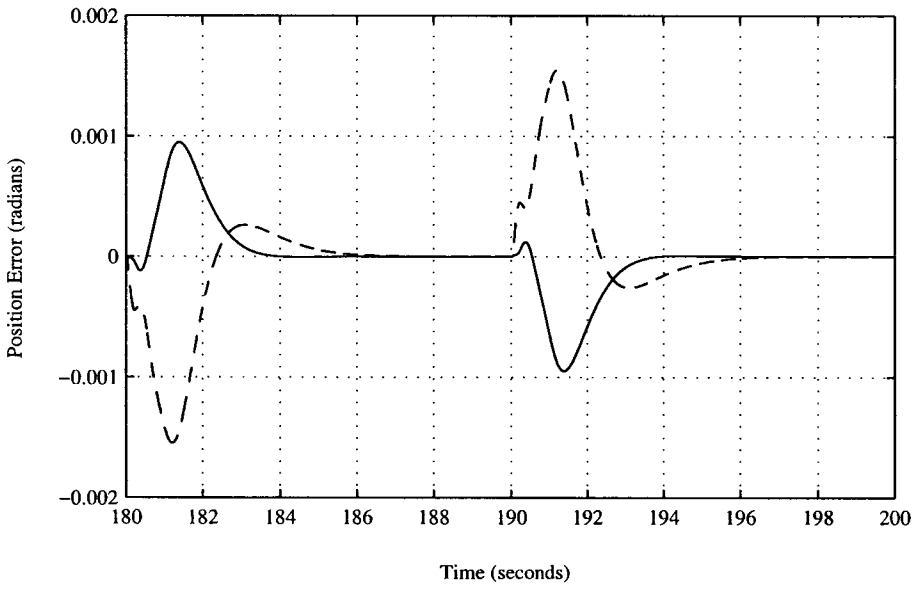

Figure 3.31: Position error for link 1 (solid) and link 2 (dashed) after 180 seconds using Craig's algorithm with a linear observer with no noise, $K_1 = 20$, $K_2 = 500$.

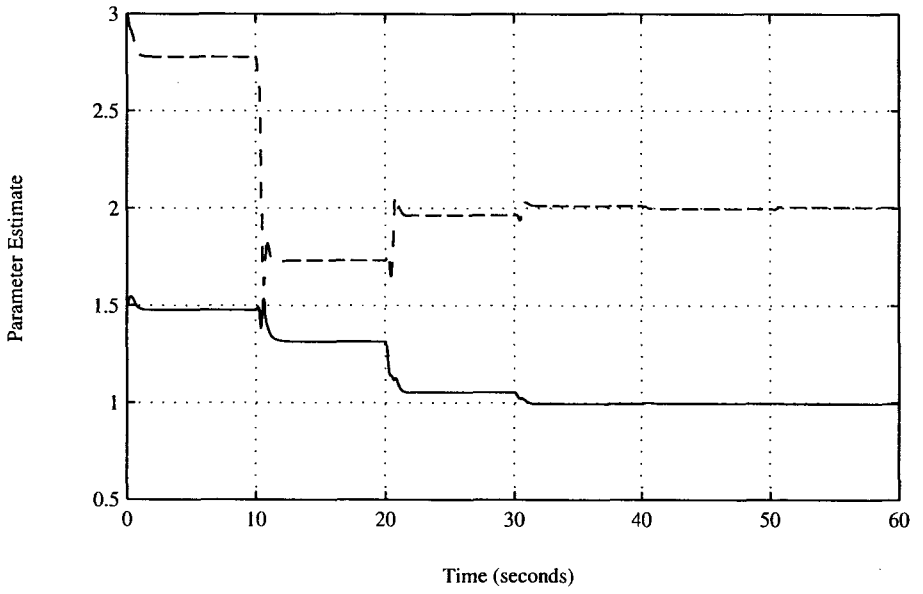

Figure 3.32: Parameter estimates over time for θ_1 (solid), and θ_2 (dashed) using Craig's algorithm with a linear observer with no noise, $K_1 = 20$, $K_2 = 500$.

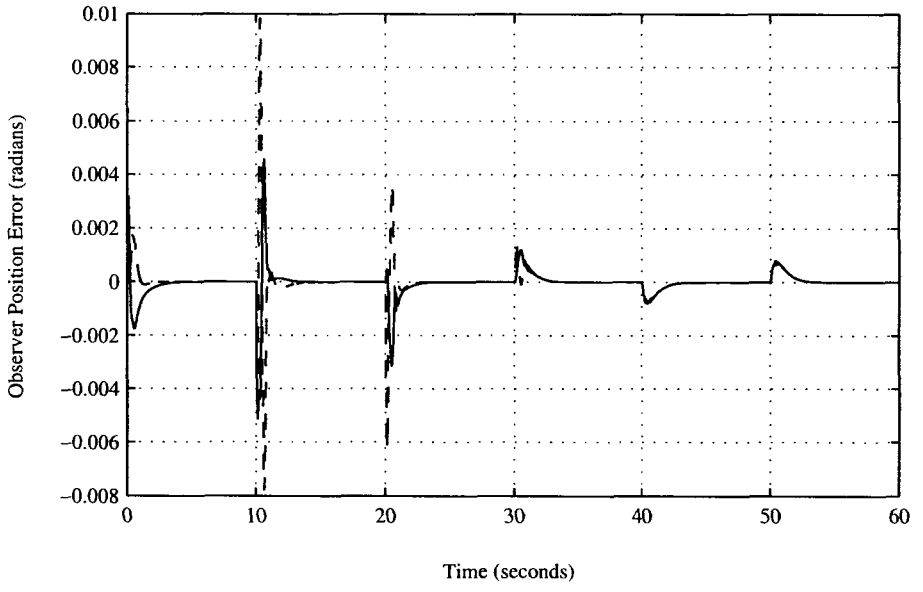

Figure 3.33: Error on the position estimates using Craig's algorithm with an observer with no noise for link 1 (solid) and link 2 (dashed), $K_1 = 20$, $K_2 = 500$.

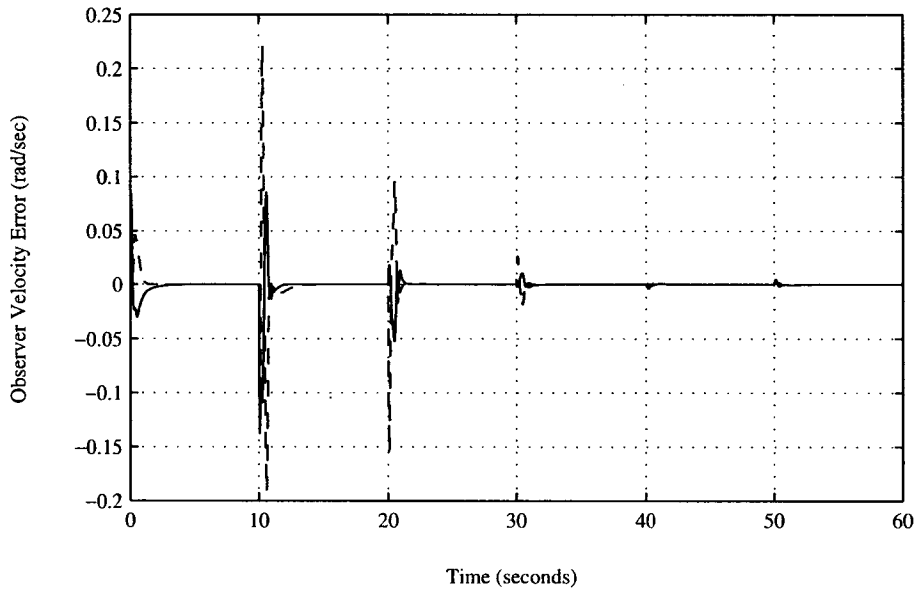

Figure 3.34: Error on the velocity estimates using Craig's algorithm with an observer with no noise for link 1 (solid) and link 2 (dashed), $K_1 = 20$, $K_2 = 500$.

link 1, and 0.16 radians for link 2. The error reached a reasonably low steady-state value, despite containing a noticeable amount of noise.

Examining the inertial parameter estimates in Fig. 3.36 it is interesting to note that the parameters estimates are near the true values. However, they do not ever reach a steady state. This is due to the random noise on the position measurements which ends up in the error signal. This noise then drives the parameter adaptation law, and prevents the parameters from reaching a steady state. As the amount of noise on the position measurements is decreased, the parameter estimates become smoother.

The torques computed from this simulation for the first 40 seconds are given in Fig. 3.37. One can see that there is some noise that ends up in the control signal. However, when compared to the torques produced using Lee and Khalil's algorithm

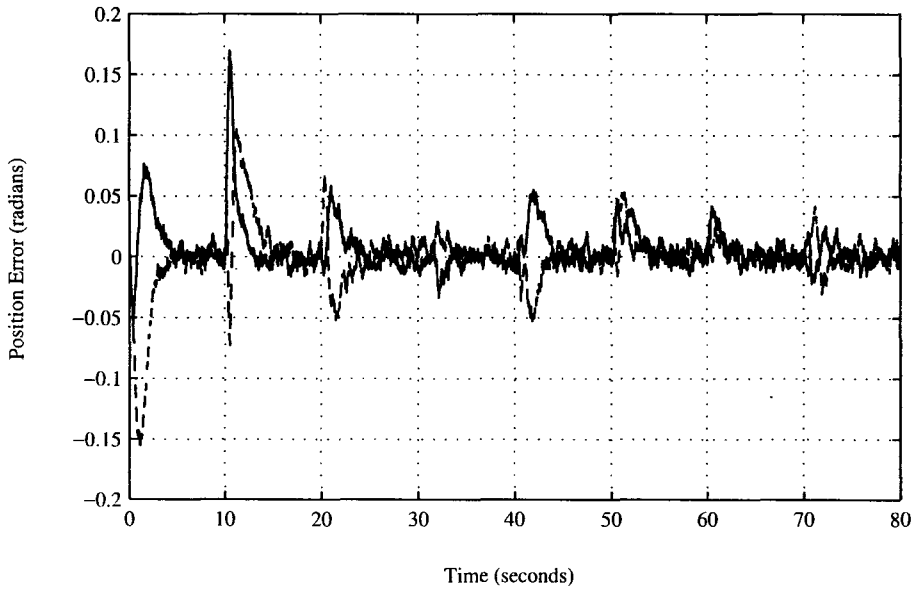

Figure 3.35: Position error for link 1 (solid) and link 2 (dashed) using Craig's algorithm with a linear observer with quantization noise, $K_1 = 20$, $K_2 = 500$.

(seen in Fig. 3.24) it can be seen that Craig's algorithm with our observer produces cleaner torque signals. A control signal containing less noise is more desirable in implementation as it is less likely to excite high frequency dynamics in the robot.

Results of Simulations

Simulations were performed using Craig's adaptive control algorithm with the addition of a linear observer. These simulations investigated the effect of pole placement for the observer poles. The initial simulation set the observer poles to be critically damped and placed at $s = -10$ in the s-plane. The poles for the closed-loop system are placed at $s = -2$. So, the observer dynamics were set to be five times faster than the closed-loop dynamics. However, it was found that these observer poles were not suitable for estimating the states. The error on the state estimates was large, as

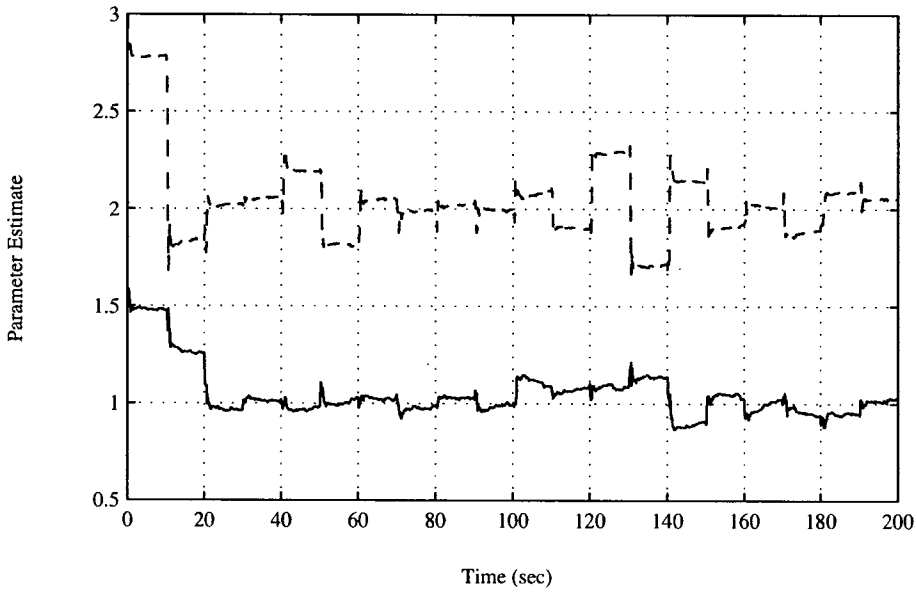

Figure 3.36: Parameter estimates over time for θ_1 (solid), and θ_2 (dashed) using Craig's algorithm with a linear observer with quantization noise, $K_1 = 20$, $K_2 = 500$.

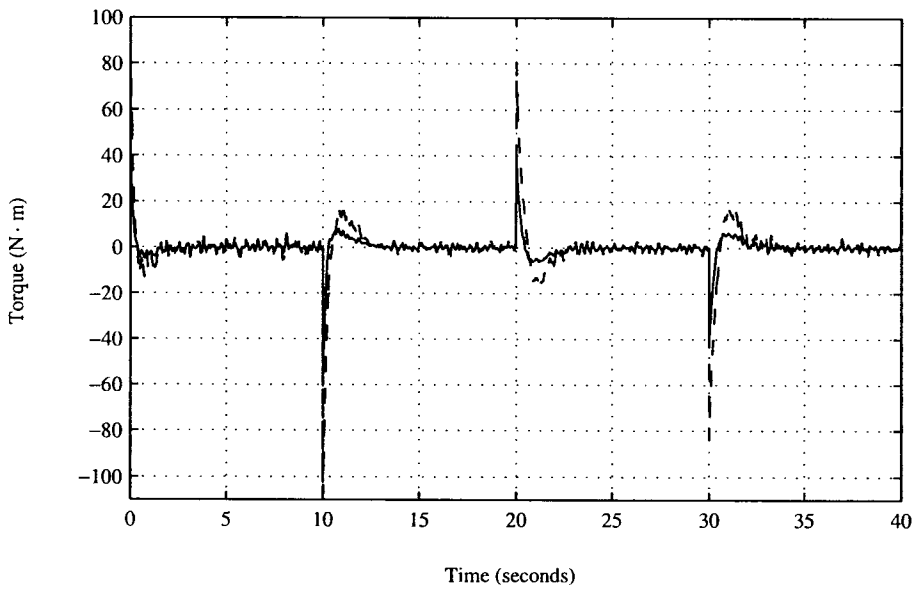

Figure 3.37: Computed torques used to drive the simulated robot for link 1 (dashed) and link 2 (solid) using Lee and Kahlil's algorithm with quantization noise.

shown in Fig. 3.29.

In order to overcome the poor performance, the observer gain K_2 was increased from 100 to 500. This resulted in observer poles placed at $s = -10 \pm 20j$. These observer poles were underdamped, with a damping ratio of $\zeta = 0.45$. It was found that the performance of the observer with this set of gains was much better. This led to better tracking performance for the algorithm, and more accurate inertial parameter estimates.

To test this setup in the presence of noise, the simulation with larger observer gains was repeated. This time noise was added to the position measurements to simulate the effect of a 12-bit quantizer. The tracking performance was quite comparable in this case, although it was evident that the position error signal contained noise. When the computed torque signals were examined, it was seen that they did contain some noise. However, the amount of noise was noticeably less than the simulations performed using Lee and Khalil's algorithm with noise on the position measurements.

This set of simulations with this algorithm shows the importance of properly setting the observer gains. It is important that the gains be large enough to ensure accurate trajectory tracking. However, as the observer gains are increased the noise on position measurements is amplified by the observer, increasing the sensitivity of the algorithm to noise.

3.5 Discussion of Simulation Results

This chapter has presented simulation results for each of the algorithms studied. All of the algorithms were examined in simulations free of noise as well as simulations with noise on the position measurements. This noise was used to simulate the effect

of a 12-bit quantizer. Every simulation was performed with the same simulated robot dynamics, the same trajectory and pre-filter, and the same set of controller feedback gains. This was done to ensure consistency among the simulations. In the previous section, each algorithm was compared against itself for various changes in the parameters. This section aims to compare all of the algorithms against each other.

In many simulations it was found that tracking performance in cases free of noise was fairly similar to cases with measurement noise. This is encouraging, as implementations on a real experimental platform are subject to quantization noise. When examining the performance of such a control algorithm, one is often very concerned with the ability of the algorithm to track a desired trajectory. As a result, plots of the trajectory error for each of the algorithms are given. Fig. 3.38 shows the tracking error for link 1 at steady-state using each of the three algorithms. Fig. 3.39 shows the same information, except it is the tracking error for link 2. From these graphs it can be seen that Lee and Khalil's algorithm yields the lowest trajectory tracking error at steady-state. Craig's algorithm with our observer is next in terms of tracking performance, followed by Gourdeau and Schwartz's algorithm. It should be noted that all algorithms perform quite well nonetheless.

Gourdeau and Schwartz's algorithm performed well in the simulations. However, in the transient error response, this algorithm showed the largest peak error values. It was found that decreasing the values of the diagonals of matrix $R(t)$ improved performance in terms of the peak error. The error on the state estimates was also much smaller with a smaller $R(t)$, and this contributed to decreasing peak tracking error. The importance of correctly setting $R(t)$ to reflect the actual level of noise in the measurements was demonstrated as well. Introduction of position noise to simulate a 12-bit quantizer, while using the low value for $R(t)$, resulted in noisy state

estimates and poor tracking. However, introduction of position measurement noise with a larger $R(t)$ did not result in such poor tracking. An obvious strength of this algorithm is the noise rejection properties of the EKF [8]. With 12-bit quantization noise on the position measurements, this algorithm produced the cleanest control signals with which to drive the robot.

Lee and Khalil's algorithm yielded very satisfactory trajectory tracking results. Simulations were performed with two different values of the adaptation gain to examine the effect. The trajectory tracking at steady-state was significantly better with a larger adaptation gain. However, when a large adaptation gain, $\Gamma = 50I$, was used, while the peak error was smaller and the inertial parameter estimates were quick to converge, introduction of noise on the position measurements caused divergence of the algorithm. At a lower value for the adaptation gain, $\Gamma = I$, noise was introduced on the position measurements. While tracking performance remained in the same range as the noise free case, the torque signals computed by the controller contained relatively large amounts of noise, as compared to Gourdeau and Schwartz's algorithm with the same amounts of noise. The observer in this algorithm performed quite well, yielding lower error on the state estimates than the state estimators of the other algorithms.

Craig, Hsu, and Sastry's algorithm with our observer was tested in simulation for different sets of observer gains. It was found that with small observer gains the performance of the algorithm was not acceptable from the perspective of tracking performance. As well, the inertial parameter estimates converged to values far from the true values. When the observer gain K_2 was increased, resulting in underdamped observer poles, the performance of the algorithm became much better. The tracking performance was improved and the inertial parameter estimates converged to values

very close to their true values. When noise was added to position measurements to simulate 12-bit quantization error, the algorithm continued to perform well. When the control signals were examined, there appeared to be some noise on the signal, although it was less noise than that produced using Lee and Khalil's control algorithm. With large enough observer gains the linear observer performed quite well with respect to estimating the position and velocity of the robot. One limitation of this algorithm is that the adaptation gain Γ must be kept below some threshold. In the simulations it was found that values of Γ larger than $\Gamma = I$ resulted in unacceptable performance of the manipulator.

While performing simulations with each of the algorithms, a program known as a *code profiler* was run to determine the relative computation time required for each of the algorithms. It is important to know the computational requirements for an algorithm in order to determine how best to implement it in real-time. The GNU profiler (gprof) was used to perform profiling. It was determined that Gourdeau and Schwartz's algorithm, as implemented in this work, required the most amount of processing time per time step. Lee and Khalil's algorithm required less processing time than Gourdeau and Schwartz's, while Craig's algorithm with the addition of the observer required the least amount of processing time per time step. In a real-time implementation it is important to ensure that the algorithm being implemented can be supported by the computer platform used for implementation. Code profiling can be a very useful tool for determining such processing requirements.

Having examined these algorithms in simulation, a good idea of their relative performance has been obtained. Different algorithms show different strengths in simulation, with respect to tracking performance, parameter estimates, and amount of noise in the control signal. With knowledge of the performance of the algorithms

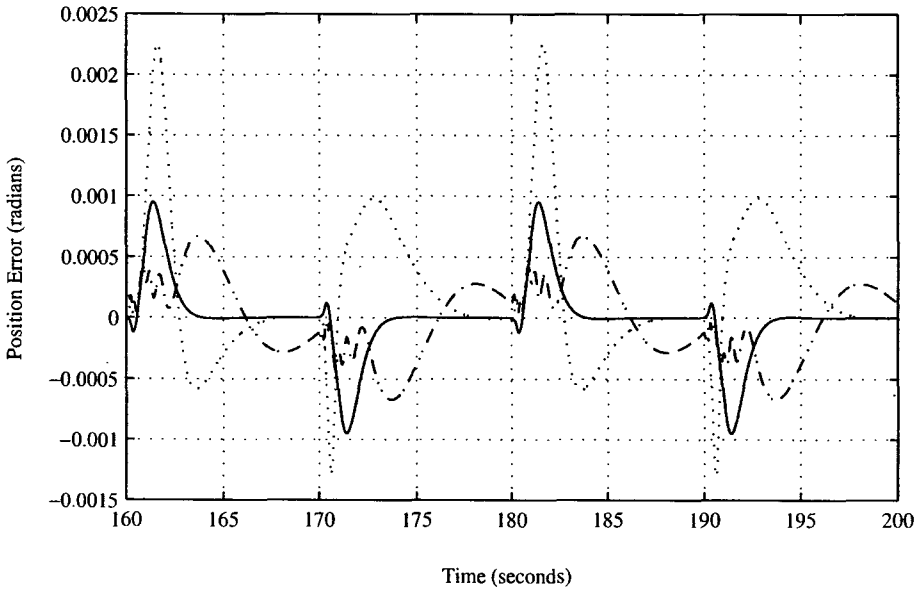

Figure 3.38: Link 1 position error for Lee and Khalil's algorithm (dash-dotted), Craig's algorithm with observer (solid), Gourdeau and Schwartz's algorithm (dotted).

in simulation, it is important to determine the effectiveness of each of them on an experimental platform. Chapter 4 will demonstrate performance of the algorithms through experiments on the Carleton University Direct-Drive Robot.

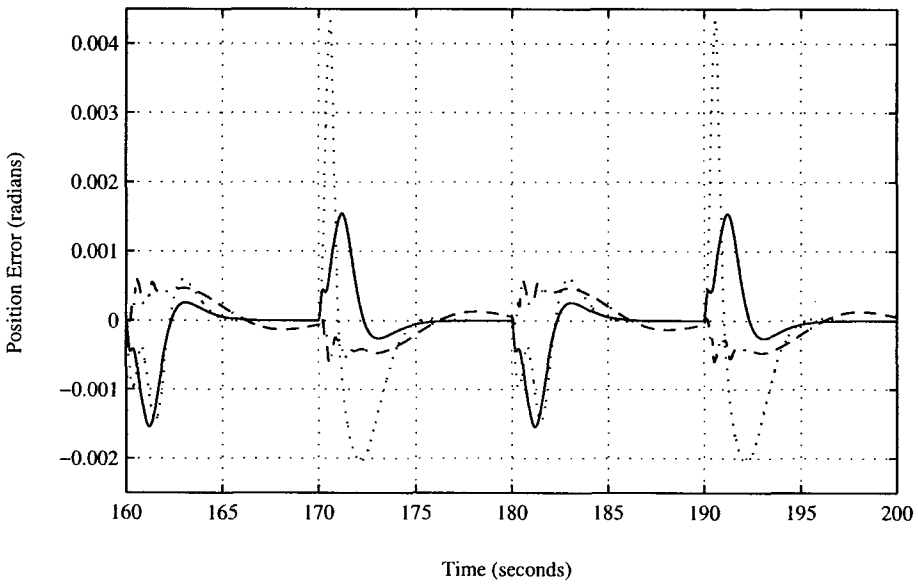

Figure 3.39: Link 2 position error for Lee and Khalil's algorithm (dash-dotted), Craig's algorithm with observer (solid), Gourdeau and Schwartz's algorithm (dotted).

Chapter 4

Experiments on the Direct-Drive Robot

4.1 Robot Hardware

In order to verify experimentally the adaptive control algorithms examined in this work, each of them was implemented on the Carleton University Direct-Drive Robot. This robot is found in the Robotics Lab of the Department of Systems and Computer Engineering. It is a two degree of freedom direct-drive manipulator that operates in the horizontal plane. It consists of four links in a parallelogram linkage. Being of the direct-drive type, the motors are attached directly to the links without a transmission or any other gearing in between. This configuration has the advantage that undesirable effects such as backlash and friction through the gears are eliminated [14]. See Fig. 4.1 for a photograph of the manipulator.

The links of the robot are driven by two brushless DC servo motors, built by Motion Control Systems (MCS) Inc. Each motor is supplied with current by a MCS

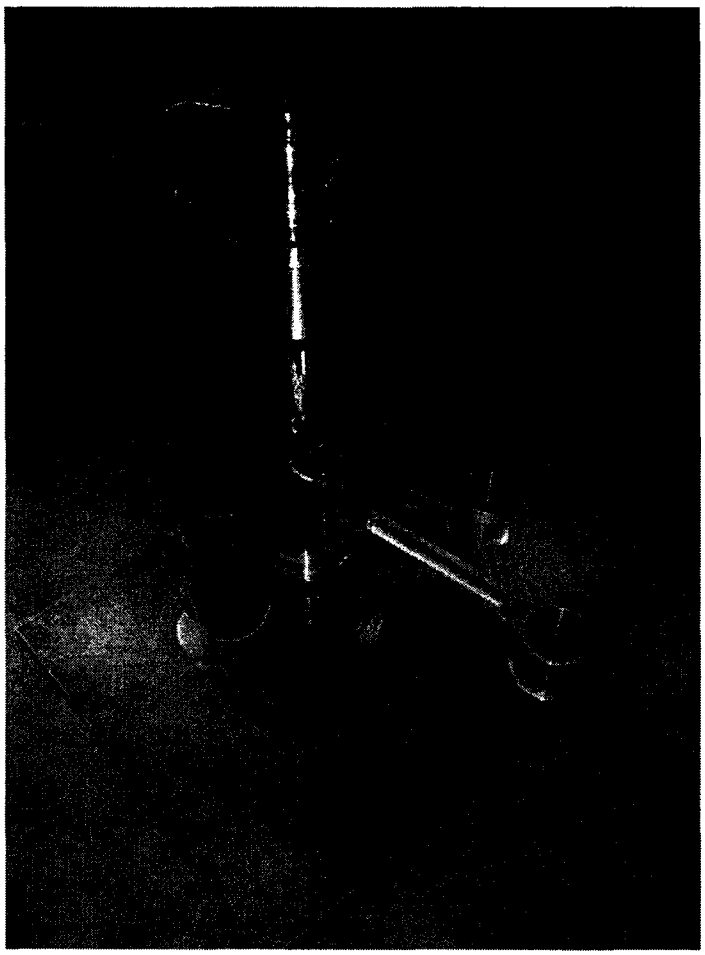

Figure 4.1: Photograph of the Carleton University Direct-Drive Robot used for all experiments in this work.

servo amplifier, which is powered by a MCS high voltage power supply [15]. Each motor has a resolver for measuring position, and a tachometer for measuring velocity. The resolver signals are converted by the amplifiers into 12-bit digital words. These 12-bit words are then sent to the computer through digital input/output ports of a Data Translation Inc. DT-2817 digital interface card. The analog tachometer signals are sent to a DT-2811 analog/digital card inside the computer, which samples and

digitizes the velocity measurements using 12 bits of resolution.

To drive the motors, the torques computed by the controller are converted to a voltage value through a torque constant. In previous research by Gabriel Warshaw the torque constant for each motor was measured at 1.7 N-m/V, giving a torque saturation value of 8.5 N-m [16] (based on a ± 5 V maximum control signal). The resulting values are scaled to 12-bit numbers, and are then output to the DT-2811 digital/analog converter where they are converted to a ± 5 V signal that is sent to the amplifiers. The amplifiers take these signals and convert them to 20 KHz Pulse Width Modulated (PWM) signals, which drive the motors.

The computer used to control the robot has an Intel Pentium-II CPU running at 300 MHz, with 128 MBytes of RAM. This computer provided ample computing power to run each of the algorithms at a sample frequency of 500 Hz.

4.1.1 Dynamic Model of the Direct-Drive Robot

In order to implement the control algorithms of Chapter 2, one must have knowledge of the actual dynamics of the robot being controlled. The dynamics of the Carleton University Direct-Drive Robot are derived in [12] and are now presented. This robot is a $n = 2$ degree of freedom robot, and the dynamics take the form of (2.1), but since the robot operates in the horizontal plane $G(q)$ is zero. In this case there are $p = 3$ robot parameters to be estimated, they are,

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix} = \begin{bmatrix} I_0 + I_2 + m_2 l_4^2 + m_3 l_2^2 \\ I_1 + I_3 + m_2 l_1^2 + m_3 l_3^2 \\ m_2 l_1 l_4 + m_3 l_2 l_3 \end{bmatrix}$$

where I_0 through I_3 represent inertia terms, l_1 through l_4 are the lengths of each of

the links, and m_2 and m_3 are the masses of links 2 and 3, respectively.

The mass and coriolis matrices are given as,

$$M(q) = \begin{bmatrix} \theta_1 & \theta_3 \cos(q_1 - q_2) \\ \theta_3 \cos(q_1 - q_2) & \theta_2 \end{bmatrix} \tag{4.1}$$

$$C(q, \dot{q}) = \begin{bmatrix} 0 & \theta_3 \dot{q}_2 \sin(q_1 - q_2) \\ -\theta_3 \dot{q}_1 \sin(q_1 - q_2) & 0 \end{bmatrix} \tag{4.2}$$

In this model of the robot dynamics, the viscous friction terms, $V(\dot{q})$, are neglected.

A very simple linear model is used for the robot motors. For motor i, the model equation is given as,

$$T_i = b_i v_i$$

where T_i represents the torque of motor i, b_i represents the torque constant, and v_i represents the voltage applied to the motor. The torque constant is given above as $b_i = 1.7$ N-m/V.

4.2 Control Software

In order to run the algorithms of interest on the Direct-Drive Robot, it was necessary to write software in order to both interface with the robot and implement the dynamic equations representing the control and adaptation laws. All of the software was implemented in Linux, a 32-bit Unix Operating System.

The possible choices of operating systems on which to implement the controllers were MS-DOS, MS Windows 95, or Linux. MS-DOS is a 16-bit environment which is limited in the amount of system memory it is able to address. However, writing

interrupt handlers is very straightforward under DOS. Windows 95 and Linux are both 32-bit operating systems and are capable of addressing more memory than DOS. With a fast sample rate of 500 Hz, it was important to be able to have access to enough memory to store the relevant data, and as a result the decision to use a 32-bit operating system was made. The author of this work has a stronger background in Linux, so that was chosen as an appropriate environment.

The control software can be divided into two components. The first component is called the *device driver* and is directly responsible for all communication to and from the robot – sending out torque signals, and reading position and velocity signals.[1] The device driver was written entirely in C, and provides a standardized software interface that the rest of the control software can interface with. This code effectively becomes a part of the operating system's kernel, and passes data through to the user-space control algorithm software using read and write buffers. Appendix A contains the source code listing for the device driver.

The second component of the control software is the implementation of the control algorithms themselves. All code for this section was written using C++.[2] Three separate pieces of code were written, one for each of the control algorithms:

1. Gourdeau and Schwartz's method using an Extended Kalman Filter
2. Lee and Khalil's method using a high-gain observer
3. Craig, Hsu, and Sastry's method with the addition of a linear observer, as proposed herein.

[1]While the software was written to enable reading of velocity data, that functionality is not used in this work since we are concerned with methods of output feedback adaptive control.

[2]Note that, while C++ was the language used, concepts of object oriented programming were not used in these programs.

The structure of each of the control programs is identical, each of them implement the equations outlined in Chapter 2 that make up the particular control algorithm. The source code for the implementation of Craig, Hsu, and Sastry's method with the addition of a linear observer is given for reference in Appendix B.

In order to interface with the robot, the device driver makes use of the ability of the DT-2811 analog/digital card to generate interrupts driven by a timer. The device driver software sets the DT-2811 to generate interrupts at a specified frequency related to the sample frequency. When an interrupt is generated, the interrupt handling portion of the device driver is invoked, and at that point it reads position data from the DT-2817 and outputs control torque data to the DT-2811. It is important to note that a system split into components like this can be advantageous. Having a device driver/control program pair removes any of the hardware interfacing and synchronization details that are necessary in interrupt driven I/O from the control program. It could be argued that this modular design makes the control program more straightforward for a person wishing to understand the control algorithms, without needing to complicate issues by introducing hardware interfacing.

4.3 Description of Experimental Runs

When comparing each of the three adaptive control algorithms, it was important to ensure consistencies such as use of the same trajectory function and trajectory pre-filter.

When running the experiments, a trajectory consisting of a combination of sines and cosines at different frequencies is selected. It is based on a trajectory used in [15] and is given as

$$y_1(t) = \cos(2\omega t) - \cos(4\omega t) \tag{4.3}$$

$$y_2(t) = (\pi/2) - 1.9 + \sin(\omega t) + \sin(2\omega t) \tag{4.4}$$

where ω was chosen to be 1.257 rad/s.

This signal was pre-filtered using a critically damped second-order linear filter with a bandwidth of $\omega_n = 2.0$ rad/s. A filter with this bandwidth was selected to ensure that the torques required to track the trajectory were kept within the torque saturation value of 8.5 N-m. The transfer function for this filter is given as,

$$G(s) = \frac{4}{s^2 + 4s + 4} \tag{4.5}$$

After filtering the given signals the following desired trajectories were obtained (at steady-state),

$$y_{1d}(t) = 0.3876\cos(2\omega t - 103^o) - 0.1366\cos(4\omega t - 137^o) \tag{4.6}$$

$$y_{2d}(t) = (\pi/2) - 1.9 + 0.7168\sin(\omega t - 64^o) + 0.3876\sin(2\omega t - 103^o) \tag{4.7}$$

It is these filtered trajectories that the robot was required to track. A plot of the desired trajectory for each link can be seen in Fig. 4.2.

Filtering the original trajectory provided access to the desired velocity and desired acceleration signals corresponding to the filtered trajectory. As well, the range of the filtered positions matched well with the achievable range of motion for each link. This resulted in a path that the robot was able to follow without encountering points of singularity or obstacles such as its own aluminum structure.

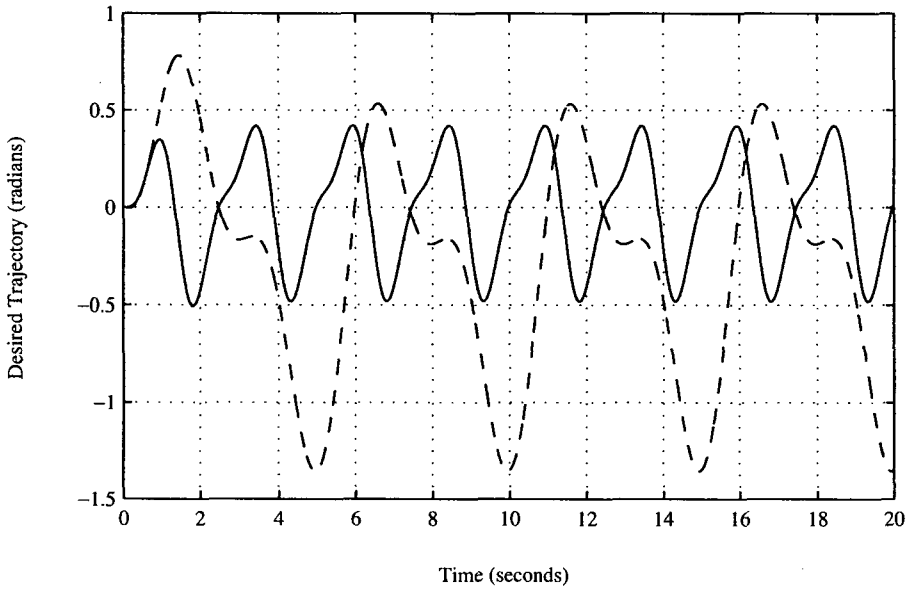

Figure 4.2: Pre-filtered trajectories used with experiments on the Direct-Drive Robot for link 1 (solid) and link 2 (dashed).

In each of the experiments, the controller gains were chosen to be $K_p = 36I_{2\times 2}$ and $K_d = 12I_{2\times 2}$. If one assumes perfect linearization of the closed-loop system, this results in closed-loop dynamics that are critically damped and have a bandwidth $\omega_n = 6.0$ rad/s, or $f_n = \omega_n/2\pi \approx 0.9549$ Hz. This is based on the idea that each joint, in combination with a linearizing controller, can be treated as a double integrator [11]. The feedback gains then specify the dynamics of the linearized second-order closed-loop system.

For all experiments, the robot parameter estimates were initialized to $\hat{\theta}_1 = 0.2$, $\hat{\theta}_2 = 0.2$, and $\hat{\theta}_3 = 0.01$.

Each of the control programs begins with a Proportional-Derivative (PD) controller to move the robot links to the origin, that is zero radians. The reason for doing this is to minimize the initial position error, since the desired position trajec-

tory begins at zero radians. Gains for the PD controller were chosen experimentally.

The sample frequency of 500 Hz was chosen to be well above the bandwidth of the closed-loop system. As well, in the case of Lee and Khalil's high-gain observer [5], the modes of that dynamic system are quite fast. It is important to have a sample frequency large enough to capture the effect of the fast modes. Another consideration in choosing the sample frequency is that all of the state estimators and adaptation laws in this work were implemented in their continuous time differential equation form. As such, it was important to sample at a high rate in order to get a very good approximation of the true continuous time system.

4.4 Results of Experiments

A variety of experiments were run on the robot for each of the algorithms studied. Each of the algorithms performed differently with respect to ability to track the desired trajectory. This section will examine those results in depth.

It was expected that each of the algorithms would perform quite comparably given that the control laws are similar, and the initial position was set the same for each experiment. As well, initial robot parameter estimates were all initialized to the same values in each experiment. While each of the algorithms converged to very similar parameter estimates, the ability to track the trajectory varied.

4.4.1 Extended Kalman Filter Approach

Implementation of Gourdeau and Schwartz's Extended Kalman Filter based algorithm on the Direct-Drive robot proved to be a challenging task. The determination of the derivative of $f(x, u)$ with respect to x requires very careful differentiation by

hand of several matrices. However, this needs only to be performed once for initial implementation of the filter.

The experiments performed in this section will deal with setting of the filter parameters to achieve the tracking performance yielding the lowest error. By varying the values of the covariance matrices, different results with respect to accuracy of tracking and speed of convergence will be achieved.

With the EKF implemented in control software, together with the proposed controller, it was necessary to determine the filter parameters, $Q(t)$ and $R(t)$, in order to run the filter. The theory of Kalman filtering provides guidelines for the initial selection of these matrices, depending on the noise levels in the system. Section 2.3.1 outlines the role of each of these matrices in the EKF. For convenience, their role is briefly restated here. $R(t)$ is a matrix representing the variance on the position measurements. The position measurements are taken using a position resolver, so one should look to the technical specifications of the resolver for measurement accuracy. According to [8], the resolver accuracy is given as $\pm 3 \times 10^{-3}$ rad, which defines the vector v of measurement noise. It should be noted that there is also additional noise due to the quantization effects of the data acquisition hardware. The A/D card has a resolution of 12 bits, yielding $2^{12} = 4096$ levels of quantization. The effect of this quantization of the measurements is then seen by dividing the range of possible values by the number of quantization levels, giving an error due to the quantizer of,

$$\text{quantization error} = \frac{\pm\pi \,\text{rad}}{4096} \approx \pm 7.7 \times 10^{-4} \,\text{rad}$$

which is less than the stated accuracy of the resolver, so it is not taken into account.

Given that the noise processes in the system are assumed uncorrelated, the diagonals of the covariance matrices $R(t)$ and $Q(t)$ represent the variance of the random

noise processes. Knowing that the resolver accuracy is stated as $\pm 3 \times 10^{-3}$ rad, and assuming the resolvers are identical, one can write the measurement noise covariance matrix $R(t)$ as,

$$R(t) = \begin{bmatrix} 9 \times 10^{-6}\text{rad}^2 & 0 \\ 0 & 9 \times 10^{-6}\text{rad}^2 \end{bmatrix}$$

In order to set $Q_1(t)$, Gourdeau [8] assumes that there is input noise of 1% in the system. The full range of input voltage (to the amplifiers), according to Gourdeau, is ± 10V. This is a 20 V range, yielding a noise of $w_1 = 0.2$V.[3] Squaring that term to obtain the variance, one arrives at $Q_1(t) = 0.04I\text{V}^2$, representing the first two diagonal elements of the matrix $Q(t)$. The diagonal elements of $Q_2(t)$ are chosen to affect the rate that the EKF learns the inertial robot parameters. Gourdeau initially chose $Q_2(t) = 9 \times 10^{-4}I$ for slow tracking of the inertial parameter estimates. This yields a process noise covariance matrix $Q(t)$ given as,

$$Q(t) = \begin{bmatrix} 0.04\text{V}^2 & 0 & 0 & 0 & 0 \\ 0 & 0.04\text{V}^2 & 0 & 0 & 0 \\ 0 & 0 & 9 \times 10^{-4} & 0 & 0 \\ 0 & 0 & 0 & 9 \times 10^{-4} & 0 \\ 0 & 0 & 0 & 0 & 9 \times 10^{-4} \end{bmatrix}$$

The initial value for the error covariance matrix $P(t)$ was set to $P(t_o) = 0.1I_{7\times 7}$, as in [8].

An experiment was performed with the values for $R(t)$, $Q(t)$, and $P(t_o)$ given

[3] According to the technical specifications for the A/D Interface Card given in Appendix E of [15], the A/D card actually has an output range of ± 5V. This would make the full range of input voltage to the amplifiers ± 5V, and not ± 10V.

above. However, with this setup on the robot, the EKF diverged. This was repeated several times with the same result each time. Examining the diagonals of the error covariance matrix $P(t)$, it was found that the diagonal elements corresponding to the error covariance of the velocity estimates become negative when $R(t)$ is set to the value given above. According to [17], if a diagonal element in a covariance matrix is less than zero, the matrix is not positive definite. As a result, the error covariance matrix was not remaining positive definite and was causing divergence of the algorithm. Grewal and Andrews [17] discuss occurrence of such undesirable behaviour and mention numerical instability resulting from the finite precision computer implementation as a cause. It is suggested that numerical errors can result in the error covariance matrix becoming indefinite, leading to the destabilization of the filter estimation error [17]. By employing a different form of the error covariance update equation, one could make the EKF more robust to such numerical issues. For an in-depth discussion of numerical issues and remedies in Kalman filtering, the reader is referred to [17].

It was determined experimentally that if the value of matrix $R(t)$ was increased, the experiment would run without diverging. The smallest value for $R(t)$ that was successfully used without causing divergence of the EKF is $R(t) = 2 \times 10^{-4} I_{2\times 2}$. This is much larger than the value used by Gourdeau in [8] and [12], which could suggest that there is more noise on the position measurements than that stated in the technical specifications. This is not an unrealistic suggestion, as the robot was constructed originally in 1989, 15 years prior to this work. It is quite possible that over time and with wear on the mechanical components of the resolver that more noise has been introduced into the system. However, one can not neglect the fact that real implementations of Kalman filtering algorithms can be subject to numerical

instability, which can lead to divergence ([13], [17]). It should also be noted that use of smaller values for $R(t)$ yielded better tracking performance than use of larger values in the experiments performed here. The issue of numerical stability of the EKF algorithm may be dealt with by implementing a square-root formulation of the filter. In such an approach, a matrix $W(t)$ is calculated instead of the error covariance matrix $P(t)$. The matrix $W(t)$ is defined such that $P(t) = W(t)W(t)^T$. This approach ensures that $P(t)$ remains positive definite. The reader is referred to [13] for a more complete discussion of square-root formulations of the Kalman Filter. It is noted in [13] that the "square-root formulation gives the same accuracy in single precision as does the conventional formulation in double precision." This suggests that such an approach is capable of providing much more accurate results.

Initial Experiment

An experiment was run with $R(t) = 2 \times 10^{-4} I_{2\times 2}$, the diagonals of $Q_1(t)$ set to 0.04, the diagonals of $Q_2(t)$ set to 1×10^{-3}, and $P(t_o) = 0.1I$. The error on the estimates of the states produced by the EKF converged to a steady state, as did the inertial parameter estimates. However, the position tracking error in this experiment remained quite large. Fig. 4.3 shows a plot of the desired and actual trajectories for link 1, and Fig. 4.4 shows a plot of the desired and actual trajectories for link 2. Fig. 4.5 shows the tracking error for this experiment. From this figure it can be seen that the tracking error is quite large on each of the links. The tracking error for link 2 is much larger than that of link 1. For this experiment, the estimates of robot inertial parameters can be seen in Fig. 4.6. After an initial tuning period of roughly 30 seconds, the parameters reach a steady state.

The poor tracking performance of this setup can likely be attributed to the per-

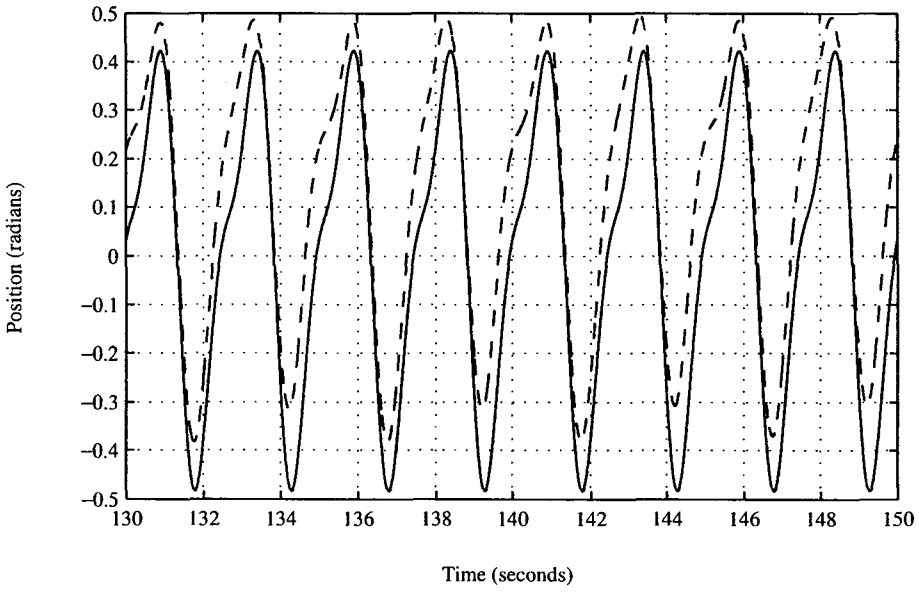

Figure 4.3: Desired (solid) and actual (dashed) trajectories for link 1 of the Direct-Drive Robot using Gourdeau and Schwartz's EKF, initial experiment.

formance of the EKF for estimating the robot position and velocity. Fig. 4.7 demonstrates the error on the estimates of the position states after 100 seconds of the experiment. For link 1, the error in the observed position has a magnitude of roughly 0.1 radians, and for link 2 the error has a magnitude of roughly 0.2 radians. This demonstrates that the EKF is not estimating the states as well as we would like.

This relatively large estimation error suggests that there may be a problem with the accuracy of the robot model being used [13]. The same model is used in [8] and [12] with much better performance of the EKF. However, the robot and associated components have aged since that time, and the robot has been partly rebuilt since then. Should modeling error be the case, a remedy suggested in [13] is to add fictitious process noise to the system. Initially, the process noise was thought to have a variance $0.04V^2$. By performing repeated experiments, the most suitable value for

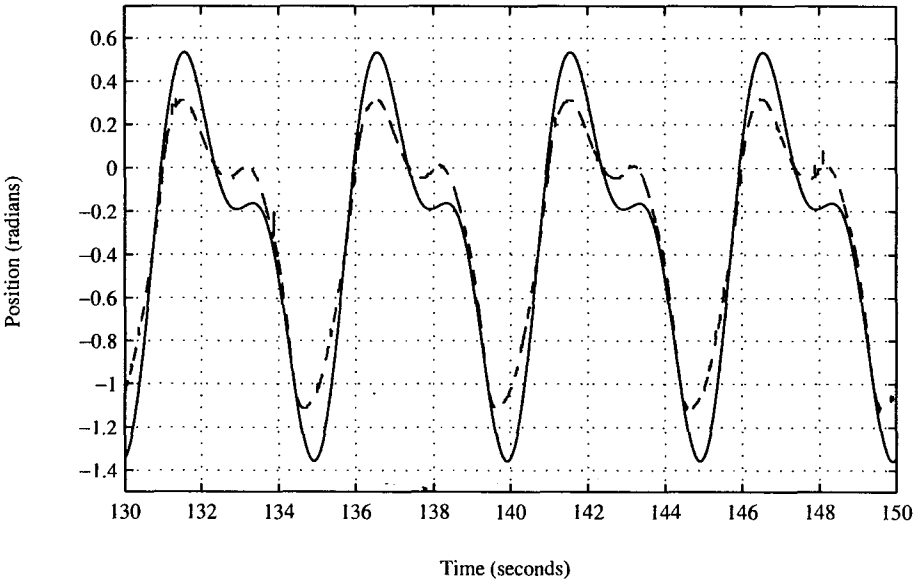

Figure 4.4: Desired (solid) and actual (dashed) trajectories for link 2 of the Direct-Drive Robot using Gourdeau and Schwartz's EKF, initial experiment.

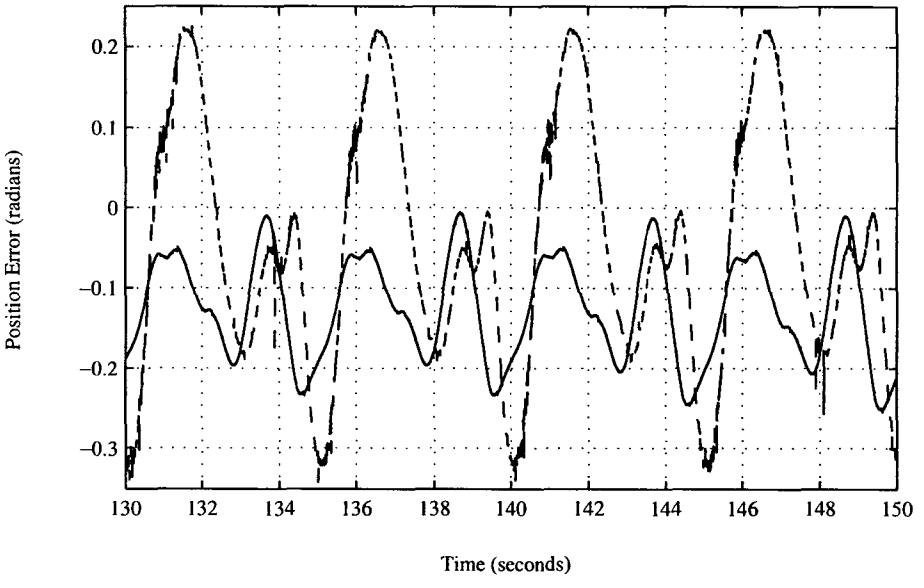

Figure 4.5: Position error for link 1 (solid) and link 2 (dashed) after 130 seconds using Gourdeau and Schwartz's EKF, initial experiment.

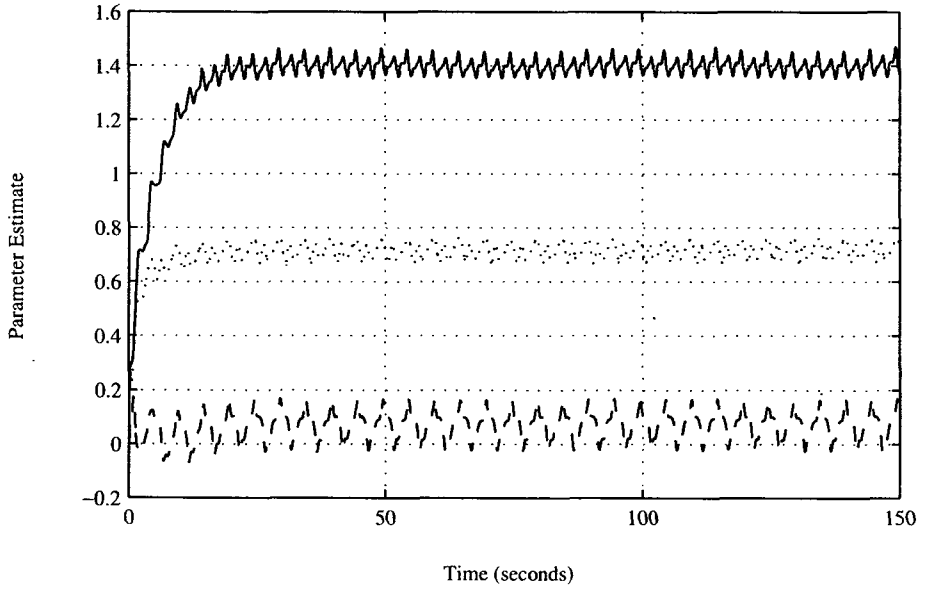

Figure 4.6: Parameter estimates over time for θ_1 (solid), θ_2 (dotted), and θ_3 (dashed) using Gourdeau and Schwartz's EKF, initial experiment.

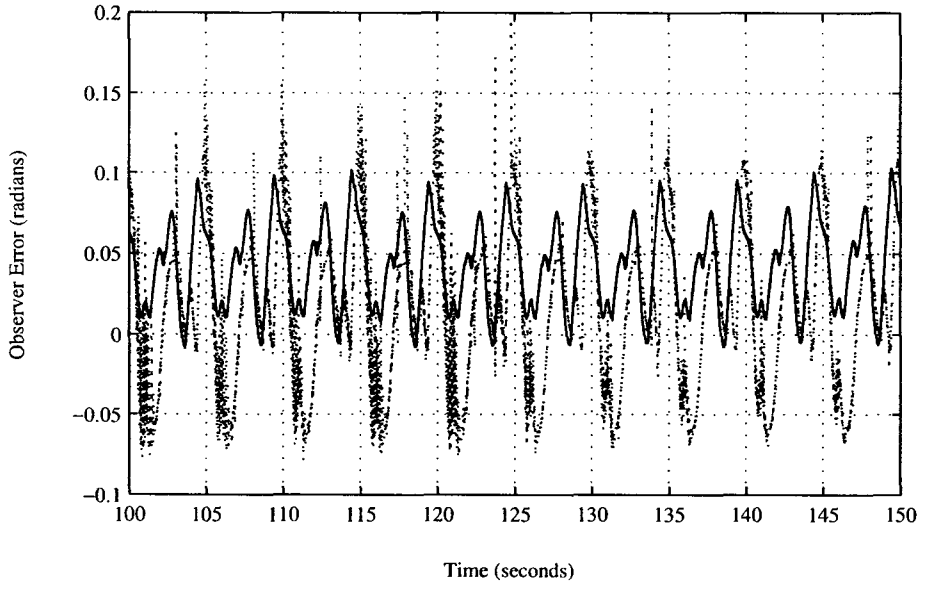

Figure 4.7: Error on the positions estimated using Gourdeau and Schwartz's EKF, on the Direct-Drive Robot for link 1 (solid) and link 2 (dotted), initial experiment.

the diagonals of $Q_1(t)$ was found to be 0.8V^2, a much higher process noise variance. In Section 2.3.1 it was mentioned that the matrix $Q_1(t)$ can be thought of intuitively as the confidence in the dynamic model available. So from an intuitive perspective it seems reasonable to increase the value of $Q_1(t)$ if the performance is poor, and a less than ideal model is thought to be the cause.

Experiment with larger $Q_1(t)$

An experiment was performed with this change made. The filter values used in this experiment were $R(t) = 2 \times 10^{-4} I_{2\times2}$, $Q_1(t) = 0.8 I_{2\times2}$, $Q_2(t) = 1 \times 10^{-3} I_{3\times3}$, and $P(t_o) = 0.1I$. The only difference between the setup for this experiment and the initial experiment is the increase in the values of $Q_1(t)$. The result was much improved position tracking performance. Fig. 4.8 shows a plot of the desired and actual trajectories for link 1, and Fig. 4.9 shows a plot of the desired and actual trajectories for link 2. The tracking error for each of the links is shown in Fig. 4.10. From this graph it is apparent that the tracking error, while initially quite large, decreases and reaches a steady state value much smaller than the position error of the previous experiment, as seen in Fig. 4.5.

For this experiment, the estimates of robot inertial parameters can be seen in Fig. 4.11. The convergence of the inertial parameter estimates takes longer than in the initial experiment, although it should be noted that the steady state parameter estimates oscillate less than those of the initial experiment, as seen in Fig. 4.6. As well, the parameter estimates of the two experiments are close in steady state value, but differ by a slight amount.

The improved tracking performance is likely due to the increased accuracy of the state estimates from the EKF. Fig. 4.12 shows the error on the estimated position

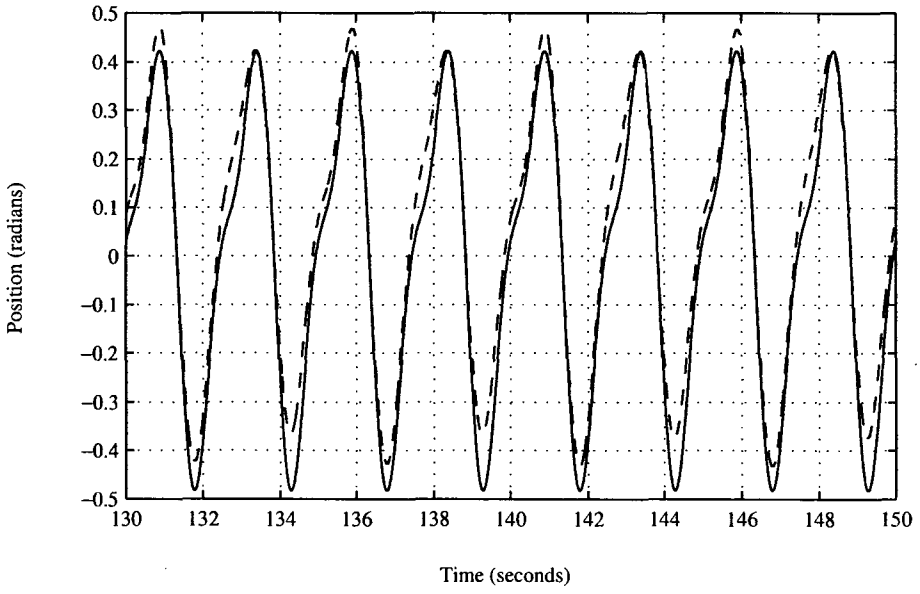

Figure 4.8: Desired (solid) and actual (dashed) trajectories for link 1 of the Direct-Drive Robot using Gourdeau and Schwartz's EKF, larger $Q_1(t)$.

values after 100 seconds of the experiment. At this point the observer error has reached a steady state. It can be seen, when compared to Fig. 4.7, that increasing $Q_1(t)$ has a very positive effect on the performance of the EKF and as a result on tracking error.

A very desirable feature of this algorithm can be seen when one examines the torques produced by the algorithm. Fig. 4.13 shows the torques computed for this experiment. It is important to note the lack of noise in the control torques. When the other algorithms will be examined, the presence of noise on the control signals will be a potentially limiting result.

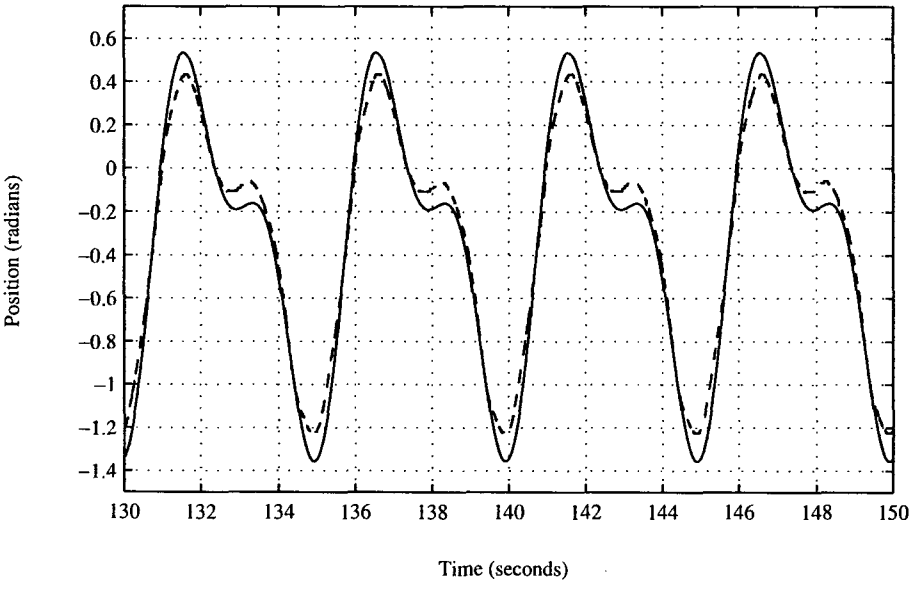

Figure 4.9: Desired (solid) and actual (dashed) trajectories for link 2 of the Direct-Drive Robot using Gourdeau and Schwartz's EKF, larger $Q_1(t)$.

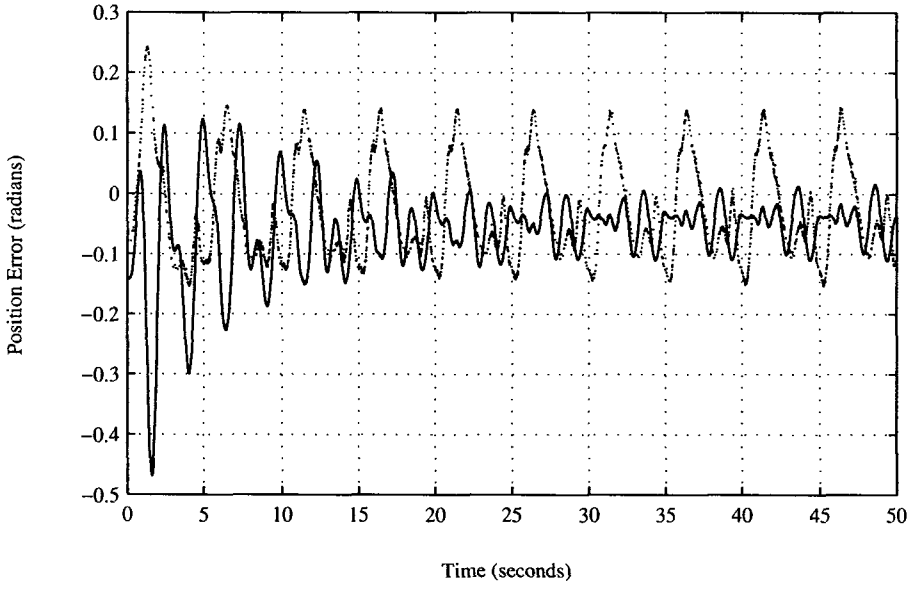

Figure 4.10: Position error for link 1 (solid) and link 2 (dotted) seconds using Gourdeau and Schwartz's EKF, larger $Q_1(t)$.

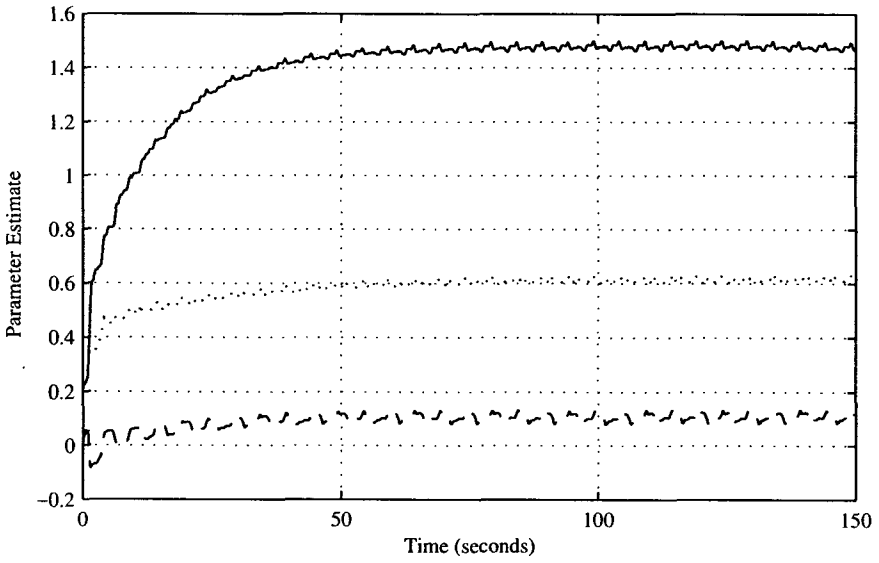

Figure 4.11: Parameter estimates over time for θ_1 (solid), θ_2 (dotted), and θ_3 (dashed) using Gourdeau and Schwartz's EKF on the Direct-Drive Robot, larger $Q_1(t)$.

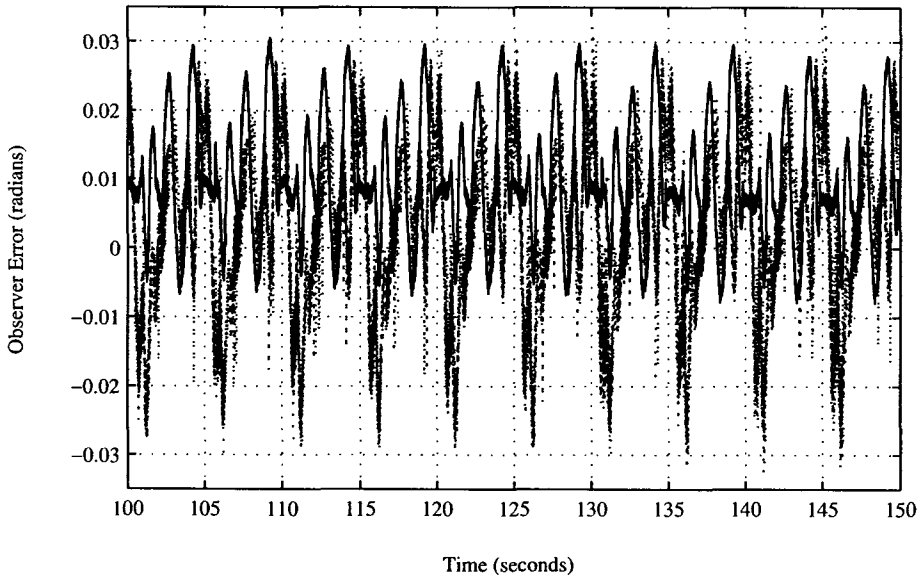

Figure 4.12: Error for the positions estimated using Gourdeau and Schwartz's EKF on the Direct-Drive Robot for link 1 (solid) and link 2 (dotted), larger $Q_1(t)$.

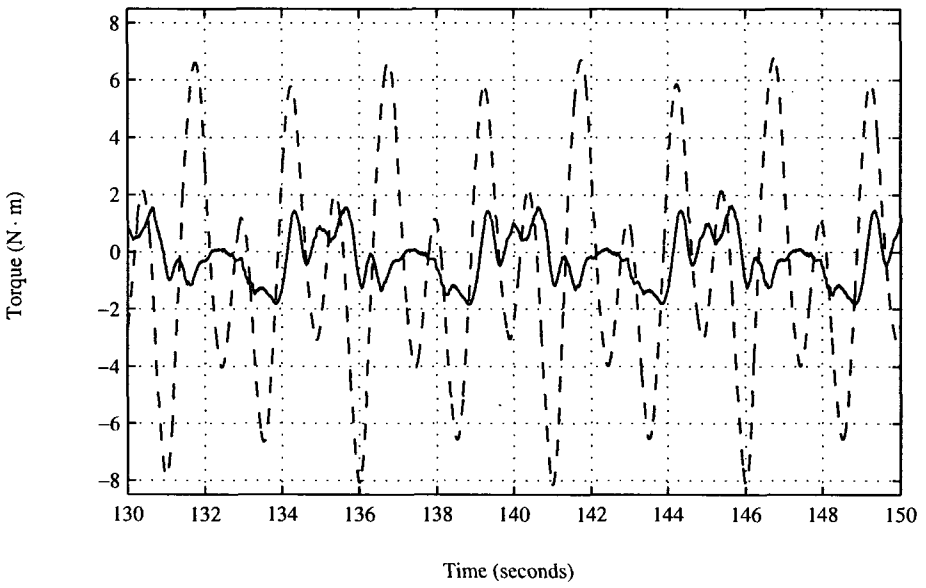

Figure 4.13: Computed torques used to drive the robot links for link 1 (dashed) and link 2 (solid) using Gourdeau and Schwartz's control algorithm, larger $Q_1(t)$.

Experiment with larger $Q_2(t)$

Following this experiment, an attempt was made to increase the rate of convergence of the inertial parameter estimates. The matrix $Q_2(t)$ governs this rate, so its values were increased by an order of magnitude to $Q_2(t) = 1 \times 10^{-2}$. The result of this change was decreased time for parameter convergence, however the parameter estimates contained larger oscillations at steady state than in the previous case with the smaller $Q_2(t)$. Fig. 4.14 shows the parameter estimates that resulted from this new experiment. Comparing that with Fig. 4.11, one can see a faster convergence with more oscillations in the current experiment. A plot of the position tracking error is seen in Fig. 4.15. When one compares this to the tracking error of the previous experiment, shown in Fig. 4.10, it is clear that at steady state there is no appreciable decrease in tracking error that results from faster parameter convergence. As

expected, though, the transient period for the tracking error is shorter when a larger value of $Q_2(t)$ is used.

Results of the experiments

Three sets of experiments were performed on the Direct-Drive Robot using Gourdeau and Schwartz's algorithm. An initial experiment was performed which yielded relatively poor tracking performance. In order to improve the tracking performance, the values of the diagonal elements of matrix $Q_1(t)$ were increased. A plot of the tracking error for link 1 comparing the performance of the experiment with $Q_1(t) = 0.04I$ and that of $Q_1(t) = 0.8I$ is given in Fig. 4.16. From this plot, one can see that the tracking error improves significantly by increasing the values of $Q_1(t)$.

After performing the experiment with increased $Q_1(t)$ an attempt was made to further increase performance of the algorithm by increasing the rate of convergence for inertial parameter estimates. The third experiment was performed, having $Q_1(t) = 0.8I$ as in the second experiment, but increasing $Q_2(t)$ to $Q_2(t) = 1 \times 10^{-2}$, an order of magnitude larger than the value used in the second experiment. A comparison of link 1 tracking performance for each of these latter two experiments is given in Fig. 4.17. One can see here that the steady-state tracking performance is roughly the same, however the tracking error is faster to decrease in the experiment with a larger $Q_2(t)$. The trade-off to make in choosing a larger $Q_2(t)$ is faster tracking error convergence at the expense of increased oscillation in the inertial parameter estimates at steady-state.

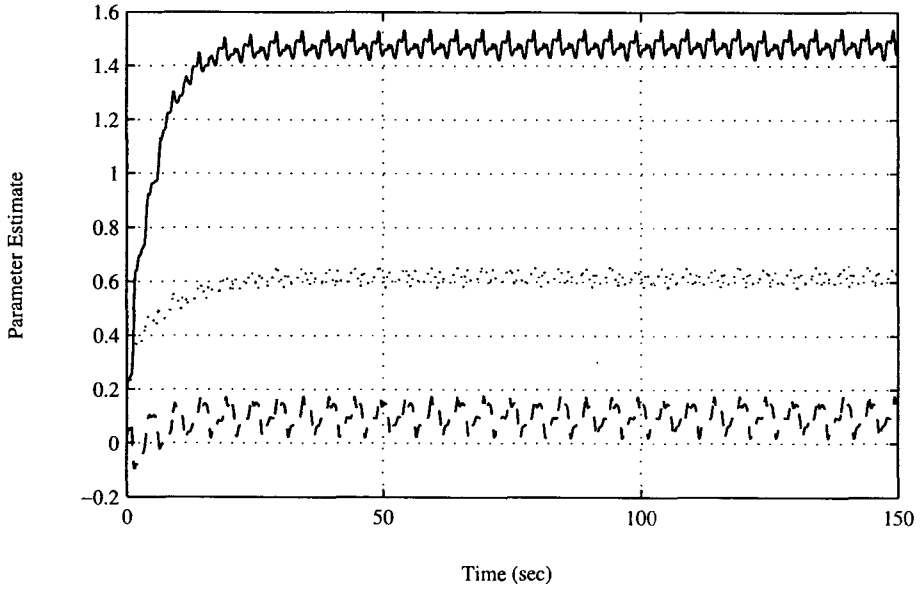

Figure 4.14: Parameter estimates over time for θ_1 (solid), θ_2 (dotted), and θ_3 (dashed) using Gourdeau and Schwartz's EKF, $Q_2(t) = 0.01I$.

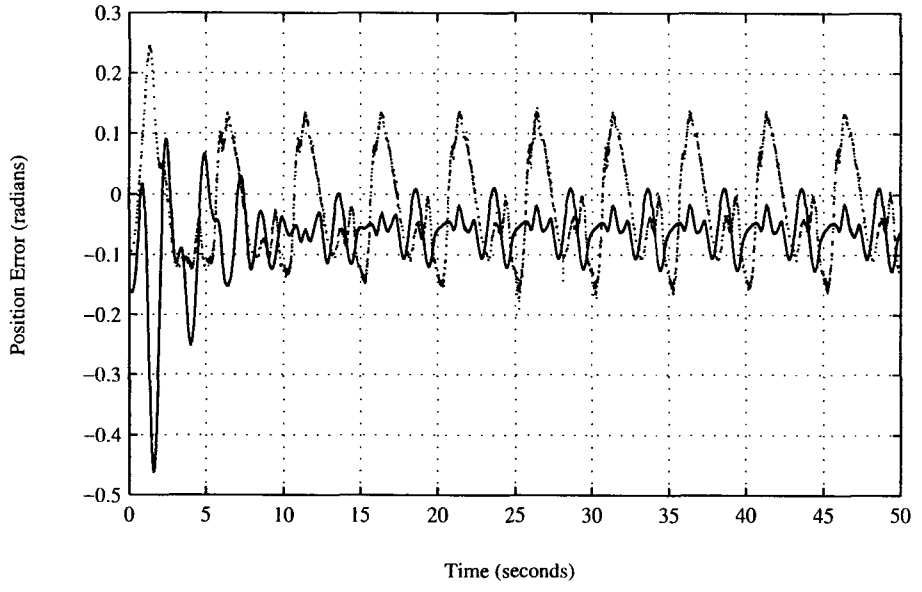

Figure 4.15: Position error for link 1 (solid) and link 2 (dotted) seconds using Gourdeau and Schwartz's EKF, $Q_2(t) = 0.01I$.

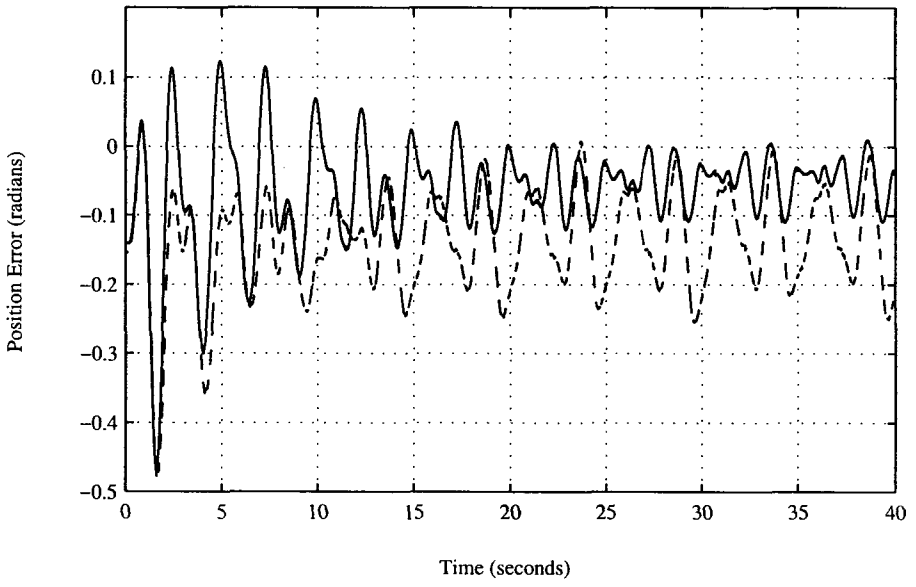

Figure 4.16: Position error for link 1 comparing two values for $Q_1(t)$. Experiments performed for $Q_1(t) = 0.8I$ (solid) and $Q_1(t) = 0.04I$ (dashed).

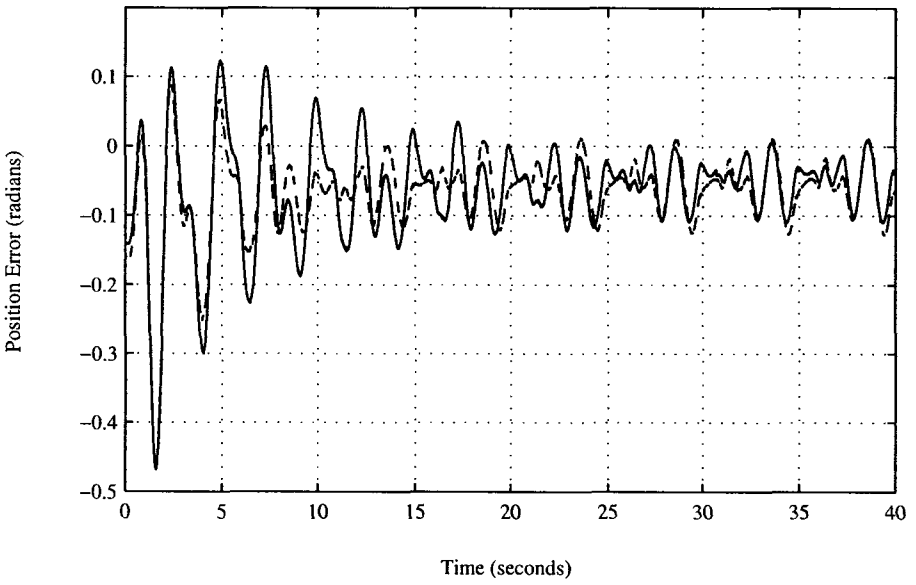

Figure 4.17: Position error for link 1 comparing two values for $Q_2(t)$. Experiments performed for $Q_2(t) = 1 \times 10^{-3} I$ (solid) and $Q_2(t) = 1 \times 10^{-2} I$ (dashed).

4.4.2 High-Gain Observer Approach

Lee and Khalil's adaptive control algorithm [5] was implemented on the Direct-Drive Robot. This algorithm has a very interesting theoretical result owing to the use of a high-gain observer. The high-gain observer results in very fast observer error dynamics, meaning that the performance of the output feedback controller recovers the performance of a full-state feedback controller quite rapidly.

The experiments performed in this section deal heavily with exciting unmodeled high frequency dynamics in the robot. The tracking performance using this approach is consistently very good. However, it will be shown that a difficulty arises as the control torque signals saturate and cause the robot to undergo much vibration. The value used for the adaptation gain Γ will be varied to determine if it has an effect on the amount of vibration occurring in the robot links.

Several choices were made for the setting of the parameters of the algorithm when conducting the experiments. Given that a real platform is used, instead of simulated dynamics, control is lost over the noise processes in the system. As a result, some care had to be taken when parameters were selected.

As in simulation, the observer parameters L_1 and L_2 were both set equal to $I_{2\times 2}$ to ensure the matrix $\bar{A}$ is Hurwitz. The parameter ϵ, related to the observer gain, was set to $\epsilon = 0.01$. This value is an order of magnitude larger than the value used for the simulations, however it is still one of the values used in the simulations presented in [5]. Such a value will also result in a high-gain observer.

To initialize the observer, its state vector was set as follows,

$$\hat{x} = \begin{bmatrix} \hat{e}_1 \\ \hat{e}_2 \\ \dot{\hat{e}}_1 \\ \dot{\hat{e}}_2 \end{bmatrix} = \begin{bmatrix} e_1(t_o) \\ e_2(t_o) \\ 0 \\ 0 \end{bmatrix}$$

where $e_1(t_o)$ and $e_2(t_o)$ represent the initial measured position error for links 1 and 2 respectively. A Proportional-Derivative (PD) controller was initially used to move the robot close to a start position of zero radians for each link. The PD controller moved each link close to zero radians, but there remained some small position error. This position error is used to initialize the observer above.

Initial Experiment

The initial experiment performed using this algorithm involved setting the adaptation gain to $\Gamma = 0.3I$. Fig. 4.18 illustrates the tracking performance of this controller for $\Gamma = 0.3I$ after 80 seconds. The tracking error here is quite small, which is a desirable result. Plots of the desired and actual trajectories for each of the links may be seen in Fig. 4.19 for link 1 and Fig. 4.20 for link 2. The inertial parameter estimates reach a steady state after roughly 20 seconds, however they do oscillate somewhat in the steady state. Fig. 4.21 shows convergence of the estimated parameters over time. Convergence happens rapidly, which helps to reduce tracking error more quickly. The high gain of the observer allows observer error to converge to a very small value quite rapidly. This is demonstrated in Fig. 4.22, which shows the speed with which the estimated position errors converge toward the true position errors. The observer error at steady state is very small, as compared to the other approaches (For example, see Fig. 4.12). The small error in observed signals leads to small tracking error. Note

that the occurrence of noise in this signal is due to the noise found on the position measurements.

While this performance appears to be quite desirable, a problem is sometimes encountered in implementation. It was found that this algorithm excites unmodeled high-frequency dynamics in the robot. While the robot is running, significant vibrations are observed in its links. At times, the whole platform of the robot shakes. This can be seen graphically by examining the control torques produced by this experiment in Fig. 4.23. At the points in the graph where the link 2 control torque (solid) saturates, the robot experiences much shaking. One can observe the high frequency content of the link 2 control torque. Fig. 4.24 shows the tracking error for links 1 and 2 during the same time interval as the torques given in Fig. 4.23. Comparing the two plots, one can see that when the link 2 torque signal saturates, the tracking error for link 2 contains high frequencies and becomes larger in magnitude. This shows the negative effect of the saturated torque signal on tracking performance. The high-gain observer of this algorithm, which works to minimize observer error, is also responsible for amplifying the system noise. This contributes to the high frequency saturating control signal that excites the robot's high frequency dynamics and causes a deterioration in tracking.

Experiment with Smaller Adaptation Gain

Experiments were performed with various values of Γ to determine if the adaptation gain has an effect on the high frequency excitations. Fig. 4.25 shows the torques computed for an experiment with $\Gamma = 0.05$, much smaller than before. When compared with the torques computed in other algorithms, such as Fig. 4.39 for Craig's algorithm with our observer and Fig. 4.13 for Gourdeau and Schwartz's algorithm, it

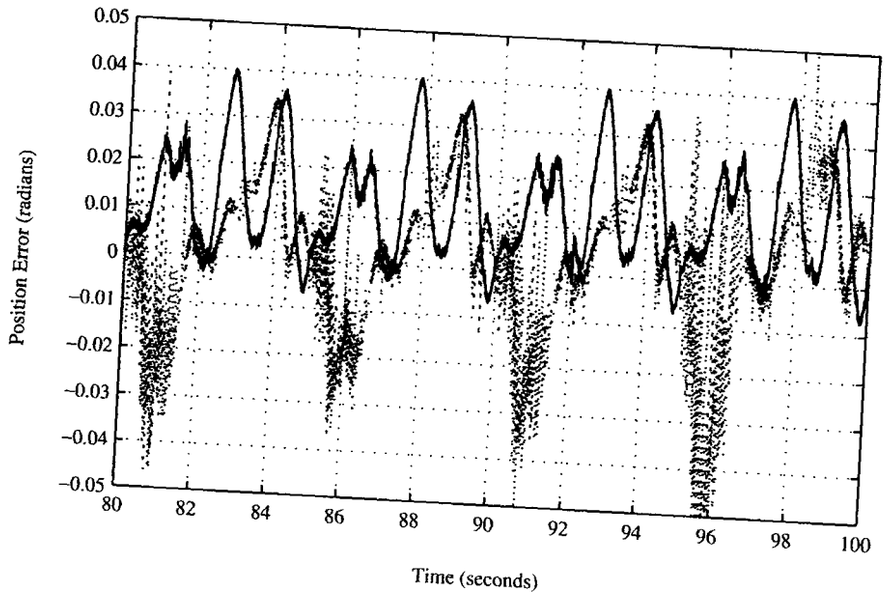

Figure 4.18: Position Error for link 1 (solid) and link 2 (dotted) after 80 seconds using Lee and Khalil's Adaptation Law with $\Gamma = 0.3I$.

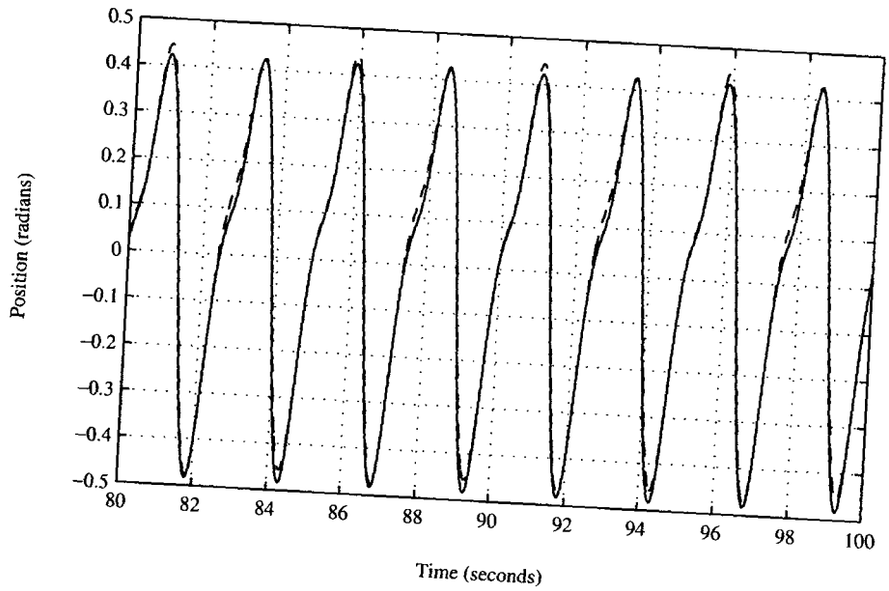

Figure 4.19: Desired (solid) and actual (dashed) trajectories for link 1 of the Direct-Drive Robot, Lee and Khalil's algorithm with $\Gamma = 0.3I$.

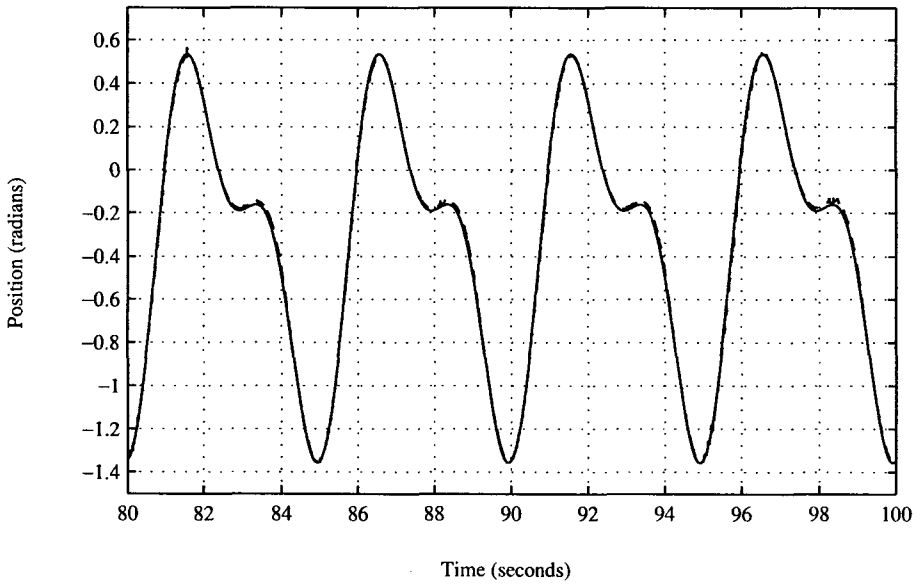

Figure 4.20: Desired (solid) and actual (dashed) trajectories for link 2 of the Direct-Drive Robot, Lee and Khalil's algorithm with $\Gamma = 0.3I$.

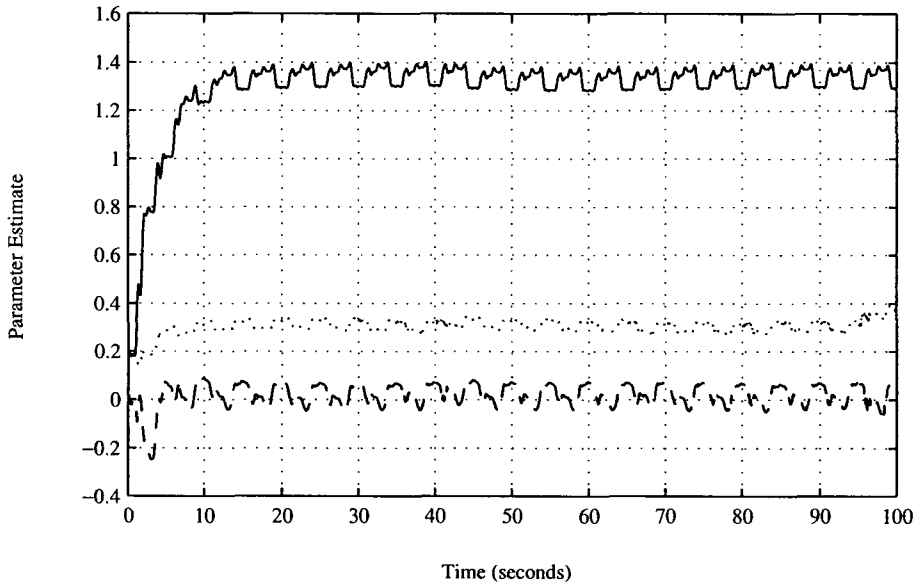

Figure 4.21: Parameter estimates over time for θ_1 (solid), θ_2 (dotted), and θ_3 (dashed) using Lee and Khalil's Adaptation Law with $\Gamma = 0.3I$.

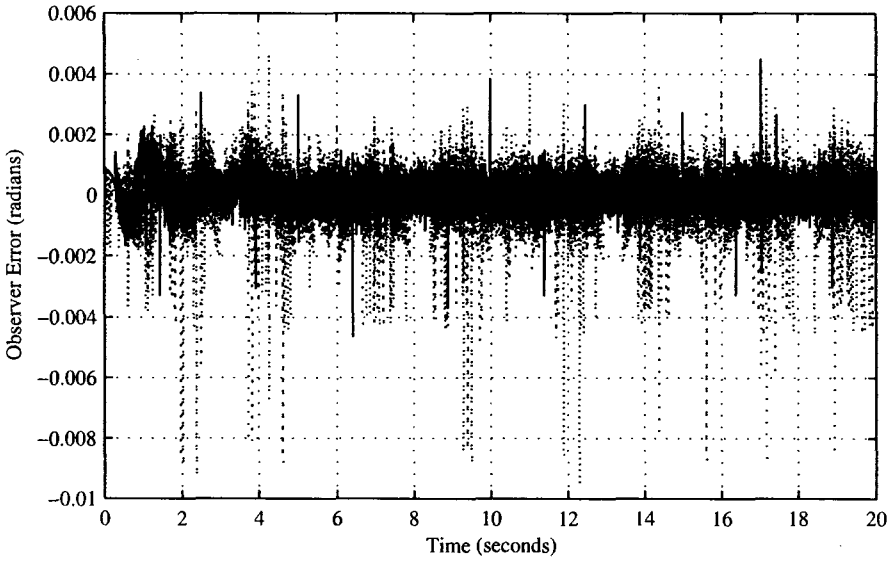

Figure 4.22: Error on the position estimates using Lee and Khalil's high-gain observer on the Direct-Drive Robot for link 1 (solid) and link 2 (dotted), $\Gamma = 0.3I$.

should be noted that the torques computed for link 2 are much smaller in those cases, and do not approach the saturation point. Again, though, in this case the torque for link 2 saturates and contains high frequency components. During this experiment the robot shook significantly, although tracking error remained quite good. Fig. 4.26 shows the tracking error for the first 80 seconds of the experiment. It takes roughly a minute for the tracking error to reach a steady state. The small adaptation gain slows parameter adaptation, which in turn slows the convergence of tracking error.

The plot of inertial parameter estimates over time is given in Fig. 4.28. The effect of the lower adaptation gain is seen in that the oscillation of the parameter estimates at steady state is much less significant than with a larger adaptation gain, as in Fig. 4.21. One can see that at roughly 160 seconds, two of the parameters begin to adapt significantly. This is due to an increased tracking error at this time.

Fig. 4.27 shows that between 160 and 180 seconds link 2 experiences position errors that are larger than the steady state values reached in Fig. 4.26. The cause of this appears to be increased vibration on link 2 between 160 and 180 seconds. Examining the computed torques given in Fig. 4.25 over the same time interval, it is clear that the increased position errors correspond with the saturated torque signals. Through observation of the experiment, it appears that the saturated torques are responsible for this vibration. The high-gain observer amplifies noise on the measurements and causes it to be passed through to the adaptation and control laws, producing a noisy control signal. This contributes to the observed vibration and the adaptation of the parameters despite the fact that there is no change in loading of the robot.

The cause of this high frequency excitation appears to be due initially to a difference between measured and estimated positions. It is this error term that drives the observer, and is scaled by the high observer gains. Such disturbances as noise on the position measurements could be responsible for this behaviour. It is interesting to note that these vibrations occur primarily with link 2. The position resolver for the motor attached to link 2 of the Direct-Drive Robot appears to produce slightly noisier position measurements than that of link 1. However, the other algorithms tested do not respond to the increased noise in this fashion. This demonstrates this algorithm's intolerance to noise on the measurements.

Experiment with Larger Adaptation Gain

Another point of interest is that the occurrence of these high frequency excitations is not predictable. The same experiment was run several times over, and in one instance the robot shook considerably, but in another the robot only vibrated slightly. The lack of predictability of these vibrations further complicates the implementation of

this algorithm in practice. In one experiment, Γ was set to $\Gamma = 2$ and left to run for a period of time. Beginning at 50 seconds, the computed torque for link 2 started saturating and the robot vibrated significantly. At 90 seconds into the experiment, the algorithm diverged. It is not sufficient for the robot to track the position well if it vibrates so significantly as to cause divergence of the experiment.

In another instance, Γ was set to $\Gamma = 2$ again, and the experiment was run for 150 seconds. Fig. 4.29 shows the tracking error for this run after 130 seconds. At a steady state, the tracking error is quite small. As a result of using the larger adaptation gain, the tracking error is quick to converge to a steady state as well. Fig. 4.30 shows the tracking error during the first 20 seconds of tracking. Note that there is an outlier in the link 2 position measurement at roughly 17 seconds. The inertial parameter estimates are shown in Fig. 4.31. A steady state in the parameter estimates is also reached in the first 20 seconds, however the large adaptation gain has the effect of creating large oscillations in the parameter estimates.

Examining Fig. 4.32, it is apparent that, while the torques contain noise, there is little saturation observed. This translates to minimal vibration in the robot links as the experiment is run.

Results of the Experiments

The experiments performed using Lee and Khalil's algorithm examined the effect of the high-gain observer on the tracking performance of the robot. It has been shown that tracking performance is consistently good with this approach, due to the rapid convergence of the observer error dynamics. However, the observer amplifies noise which gets passed through to the control signal. This causes excitation of high frequency and unmodeled dynamics.

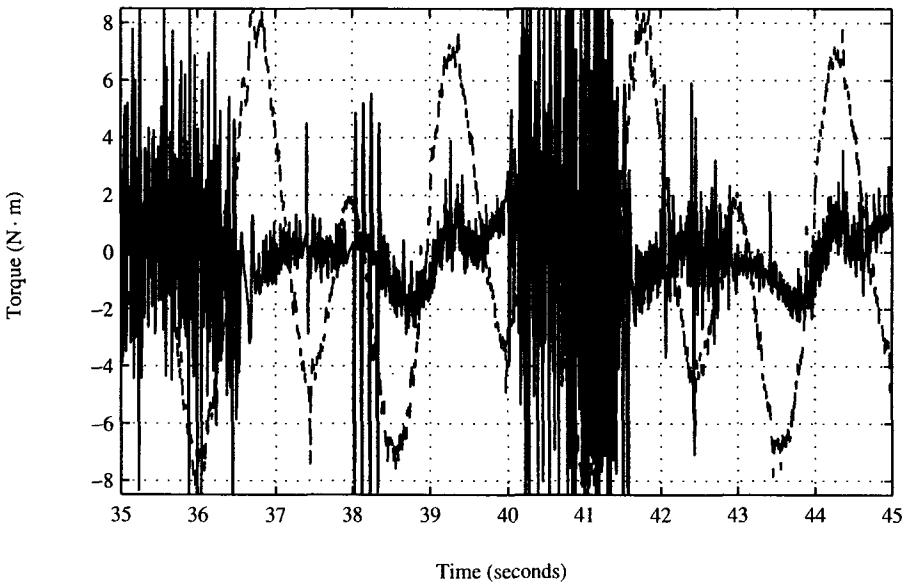

Figure 4.23: Computed torques used to drive the robot links for link 1 (dashed) and link 2 (solid) computed using Lee and Khalil's control algorithm for $\Gamma = 0.3I$.

The experiments varied the adaptation gain in order to determine if the rate of convergence of the parameter estimates has an impact on the high frequency excitations. It was found that these vibrations occurred in cases with a small adaptation gain as well as in cases with a larger adaptation gain. Further, it was found that the same experiment could be run with the same parameters repeatedly with different outcomes. In one outcome, the robot could track the desired trajectory very smoothly and not experience vibration. In another outcome, the robot could vibrate so significantly that the algorithm would diverge. This result suggests that the occurrence of the vibrations is not predictable. Based on the outcomes of these experiments, it is apparent that there are serious practical limitations for the implementation of this algorithm.

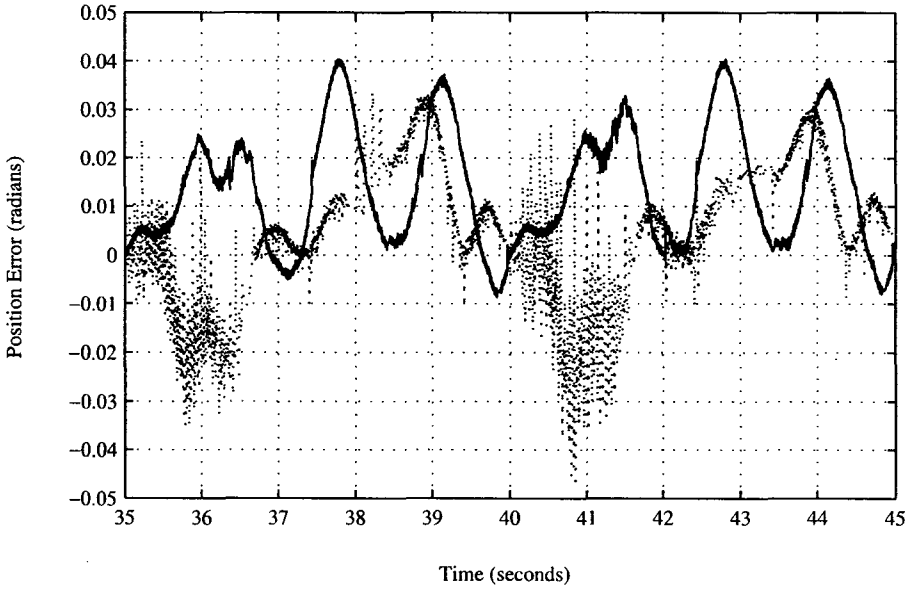

Figure 4.24: Position Error for link 1 (solid) and link 2 (dotted) between 35 and 45 seconds using Lee and Khalil's Adaptation Law with $\Gamma = 0.3I$.

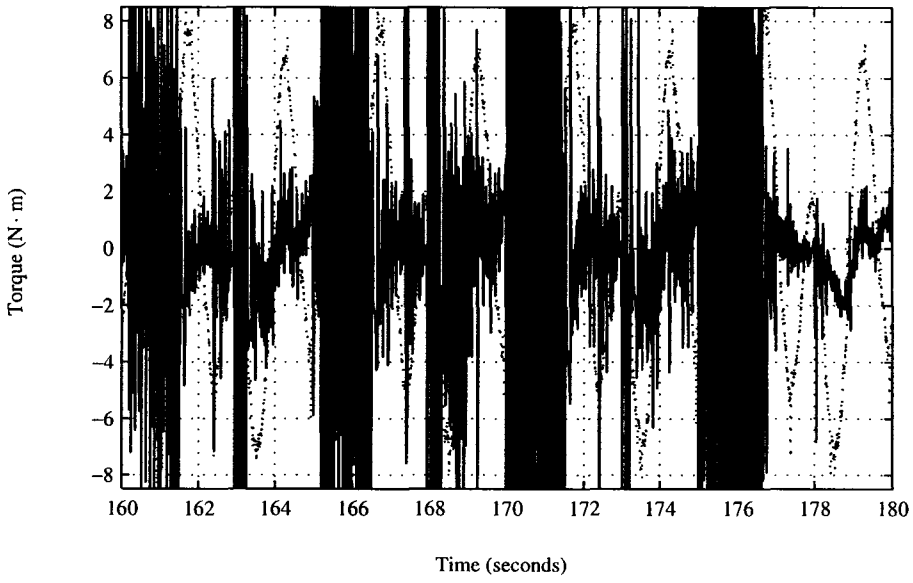

Figure 4.25: Computed torques used to drive the robot links for link 1 (dashed) and link 2 (solid) computed using Lee and Khalil's control algorithm for $\Gamma = 0.05I$.

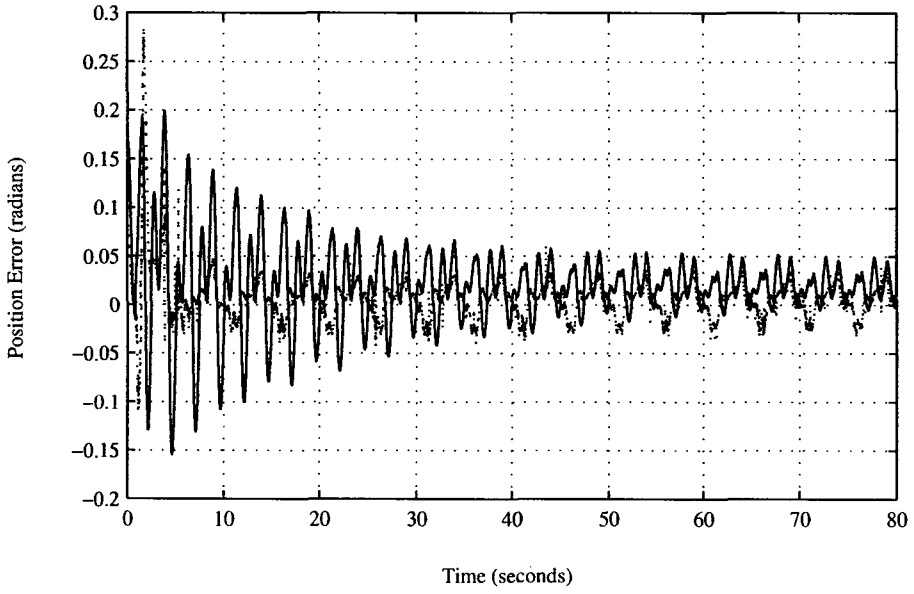

Figure 4.26: Position Error for link 1 (solid) and link 2 (dotted) for the first 80 seconds using Lee and Khalil's Adaptation Law with $\Gamma = 0.05I$.

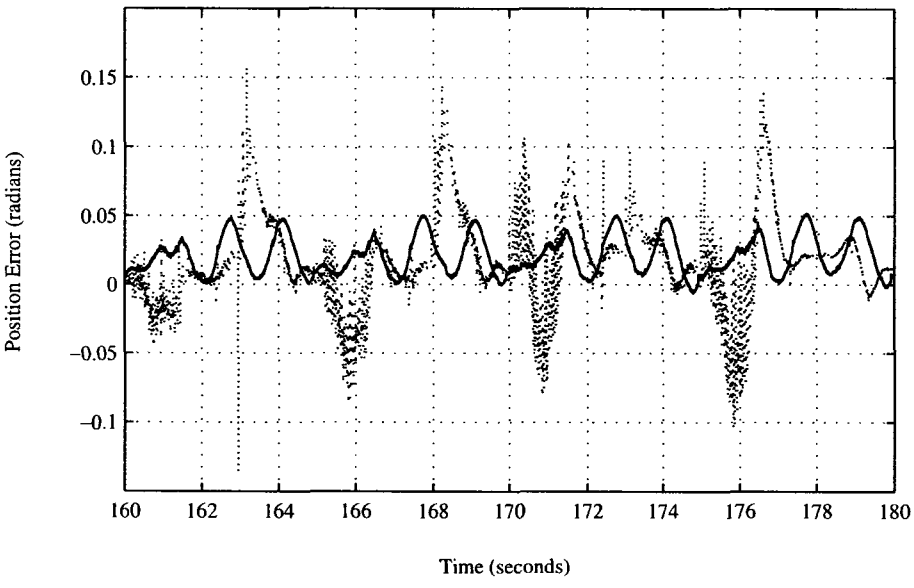

Figure 4.27: Position Error for link 1 (solid) and link 2 (dotted) after 160 seconds using Lee and Khalil's Adaptation Law with $\Gamma = 0.05I$.

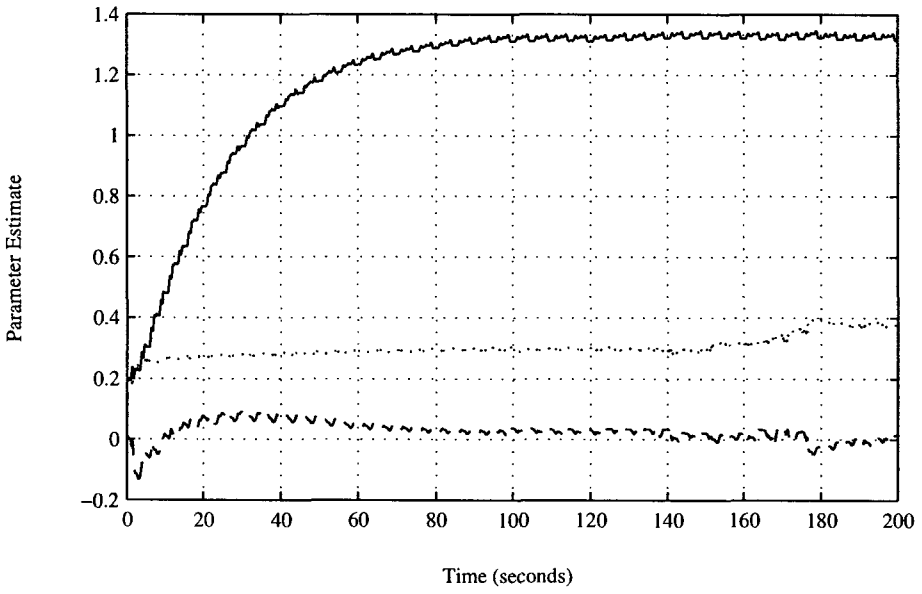

Figure 4.28: Parameter estimates over time for θ_1 (solid), θ_2 (dotted), and θ_3 (dashed) using Lee and Khalil's Adaptation Law with $\Gamma = 0.05I$.

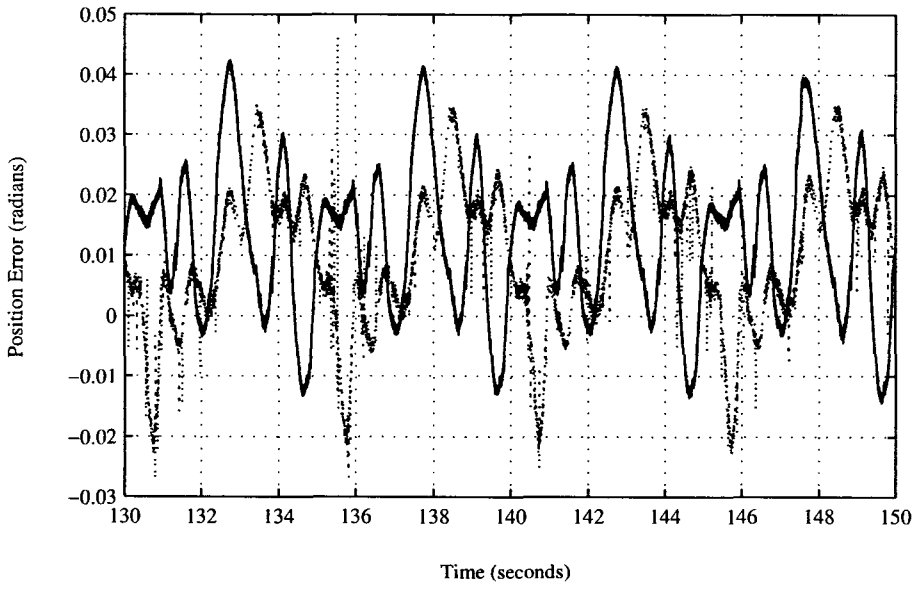

Figure 4.29: Position Error for link 1 (solid) and link 2 (dotted) after 130 seconds using Lee and Khalil's Adaptation Law with $\Gamma = 2I$.

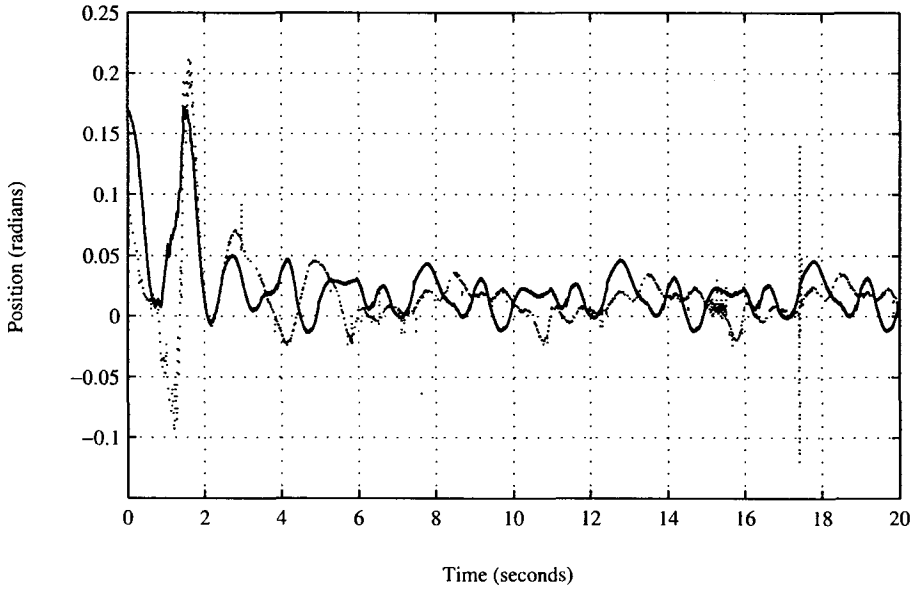

Figure 4.30: Position Error for link 1 (solid) and link 2 (dotted) for the first 20 seconds using Lee and Khalil's Adaptation Law with $\Gamma = 2I$.

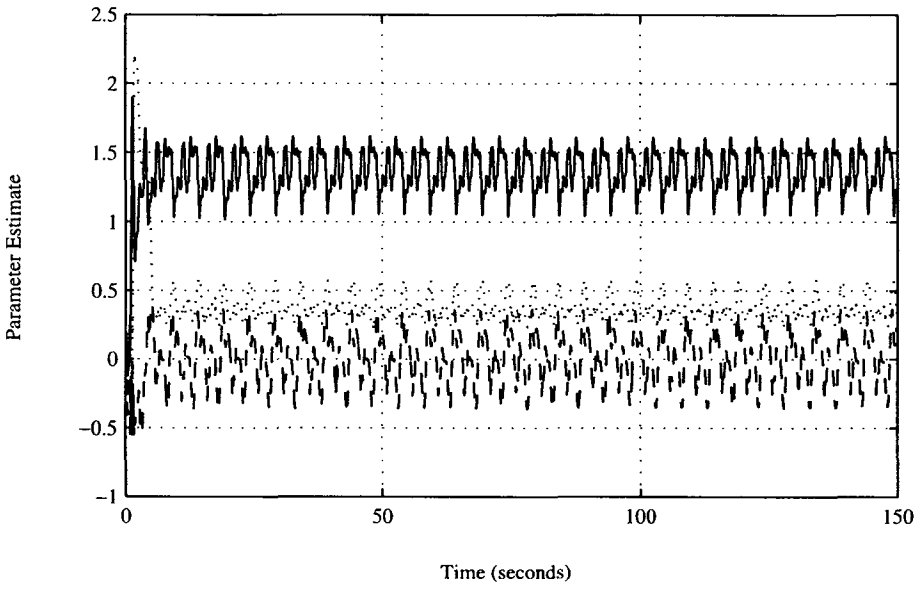

Figure 4.31: Parameter estimates over time for θ_1 (solid), θ_2 (dotted), and θ_3 (dashed) using Lee and Khalil's Adaptation Law with $\Gamma = 2I$.

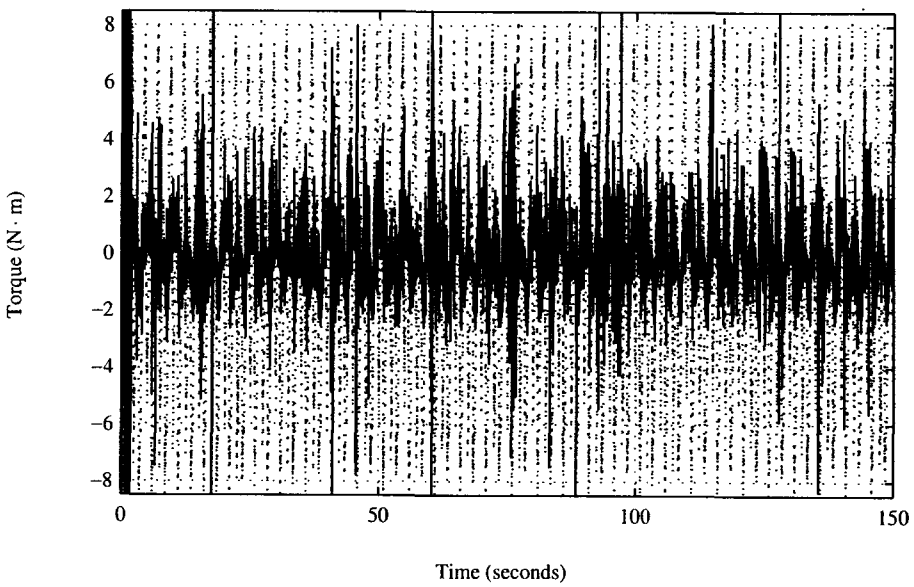

Figure 4.32: Computed torques used to drive the robot links for link 1 (dashed) and link 2 (solid) computed using Lee and Khalil's control algorithm for $\Gamma = 2I$.

4.4.3 Addition of a Linear Observer to Craig's Algorithm

In the experiments with this algorithm on the Direct-Drive robot, the parameters were chosen to allow a good comparison with the other algorithms. The controller feedback gains were set as outlined in Section 4.3. For the initial experiments, the observer gains were set to $K_1 = 20$ and $K_2 = 500$, placing the observer poles at $s = -10 \pm 20j$. These poles are faster than the dynamics of the closed-loop system.[4] It was determined experimentally that using underdamped observer poles yielded performance comparable to a set of critically damped observer poles placed farther into the negative real portion of the s-plane, while allowing use of smaller observer gains. An experiment was performed with observer poles at $s = -25$, giving observer gains of $K_1 = 50$ and $K_2 = 625$, and the observer error was not significantly different than with the observer poles placed at $s = -10 \pm 20j$. For all experiments, the value of Ψ was set to $\Psi = I$. Several values were used for the adaptation gain Γ to compare performance.

To initialize the observer for each link, the state vector was set as follows,

$$\hat{x}_i = \begin{bmatrix} \hat{q}_i \\ \dot{\hat{q}}_i \end{bmatrix} = \begin{bmatrix} q_i(t_o) \\ 0 \end{bmatrix}$$

where $q_i(t_o)$ represents the initial position measurement for link i.

Initial Experiment

An initial experiment was run with $\Gamma = 0.1I$. The tracking performance with this setup is good. Plots of actual and desired trajectories are given in Fig. 4.33 for link

[4]The settling time of a system is a function of its time constant, defined as $T = 1/\zeta\omega_n$. A smaller time constant indicates a faster settling time and is associated with faster poles.

1, and Fig. 4.34 for link 2. Fig. 4.35 shows the tracking error for this experiment at a steady state, while Fig. 4.36 shows the tracking error during the first 80 seconds of the experiment. The error takes roughly 60 seconds to reach a steady state. This is a long time when compared with other algorithms. This is due to the fact that the adaptation gain Γ is kept relatively low. The reason for this is that when run experimentally, values for Γ any larger than this result in unacceptable performance of the robot during the experiment. The parameter estimates converged to steady state values as seen in Fig. 4.37. However, due to the low adaptation gain it took close to 150 seconds for the parameters to reach the steady state.

The linear observer in this algorithm is successful in achieving a low observer error. Fig. 4.38 is a plot of the error in the observed position signals over time. Within 20 seconds the error on the observed signals had converged to values on the order of 10^{-2}. Note that there is more noise in the observer error for link 2 position. This is likely due to the fact that the position resolver on motor 2 produces slightly noisier signals.

It is interesting to note the quality of the torque signals used to command the robot. From Fig. 4.39 it is apparent that the computed torques do not contain large quantities of noise. It is this characteristic that allows smooth tracking of the links of the robot, without causing vibrations as it travels.

Experiment with Smaller Adaptation Gain

While the performance of this experiment is quite good, the choice of Γ is close to the largest possible value before unacceptable performance is observed. As a result, another experiment was performed with an adaptation gain of $\Gamma = 0.05I$, half of the previous value. The tracking error for this experiment at a steady state is shown in

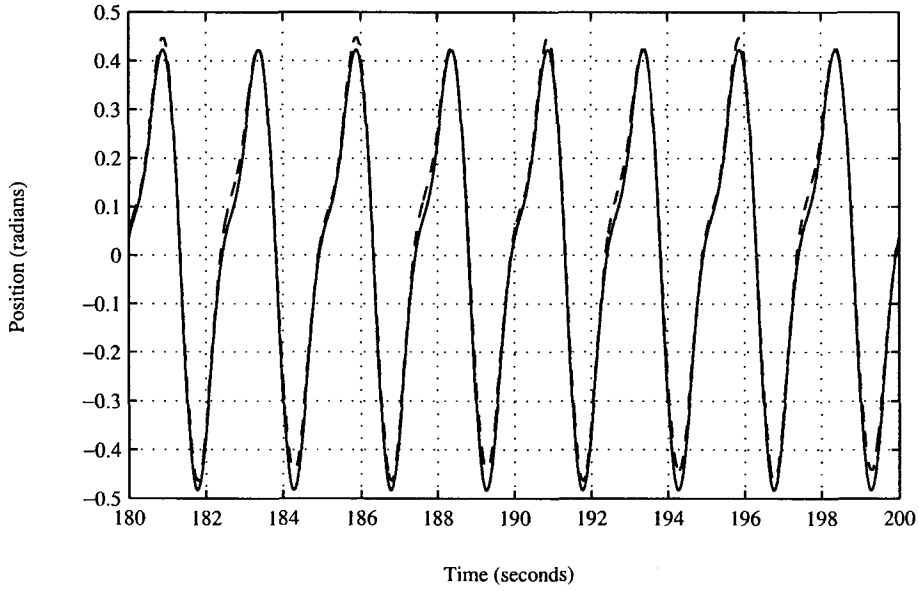

Figure 4.33: Desired (solid) and actual (dashed) trajectories for link 1 of the Direct-Drive Robot, Craig's algorithm with a linear observer, $\Gamma = 0.1I$.

Fig. 4.40. A plot of the tracking error during the first 100 seconds is given in Fig. 4.41. It can be seen here that it requires close to 80 seconds for the tracking error to reach a steady state. However, the low tracking error achieved at steady state is a positive result, since performance similar to the larger value of the adaptation gain is achieved over time with a lower adaptation gain. Using this lower adaptation gain keeps the system much further from the point of divergence of the experiment. A plot of the observer error on the position estimate for link 1 is given in Fig. 4.42.

For the case where $\Gamma = 0.05I$, a plot of the estimated inertial parameters over time is given in Fig. 4.43. After 200 seconds of running, the parameters have not all reached a steady state. However, this algorithm is limited in practice to slower adaptation, as values for Γ above $0.1I$ were not successfully implemented on the robot due to the level of noise in the system.

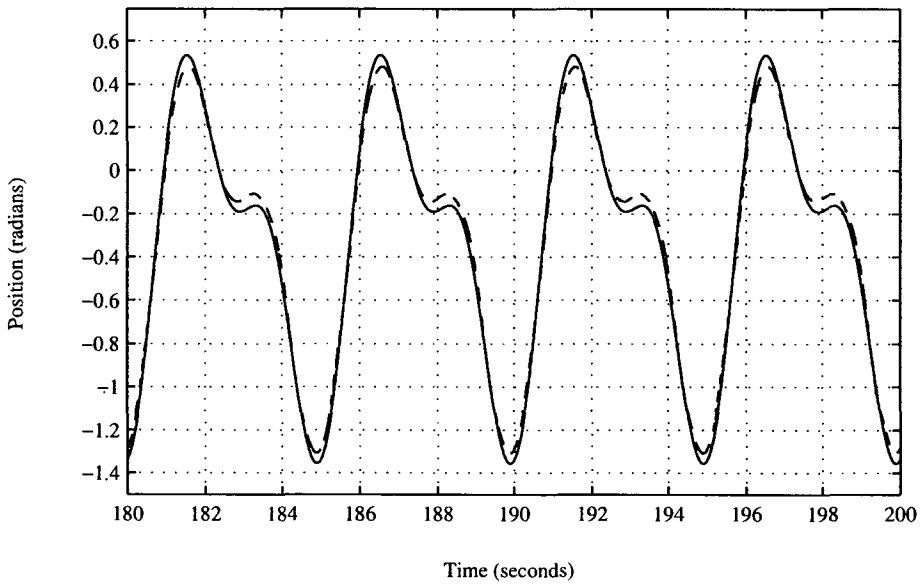

Figure 4.34: Desired (solid) and actual (dashed) trajectories for link 2 of the Direct-Drive Robot, Craig's algorithm with a linear observer, $\Gamma = 0.1I$.

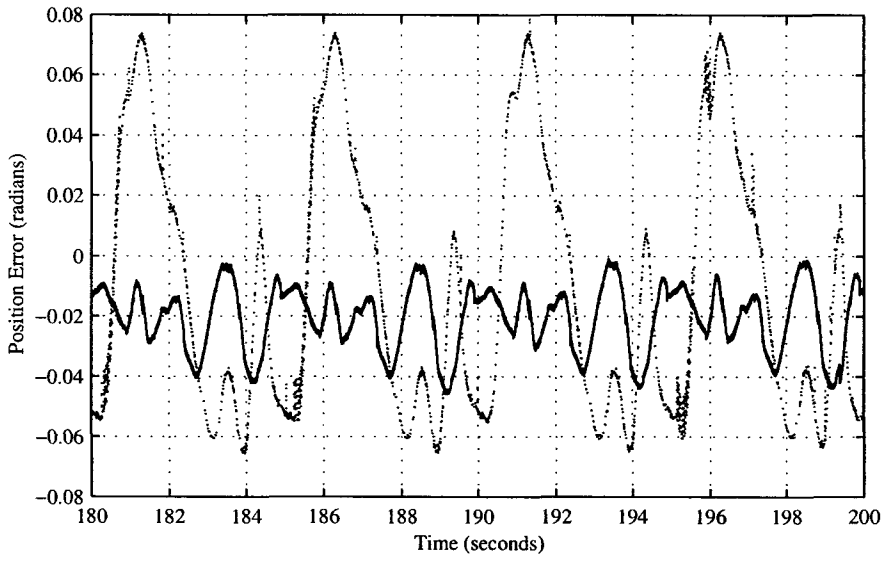

Figure 4.35: Position Error for link 1 (solid) and link 2 (dotted) after 180 seconds using Craig's method with a linear observer, $\Gamma = 0.1I$.

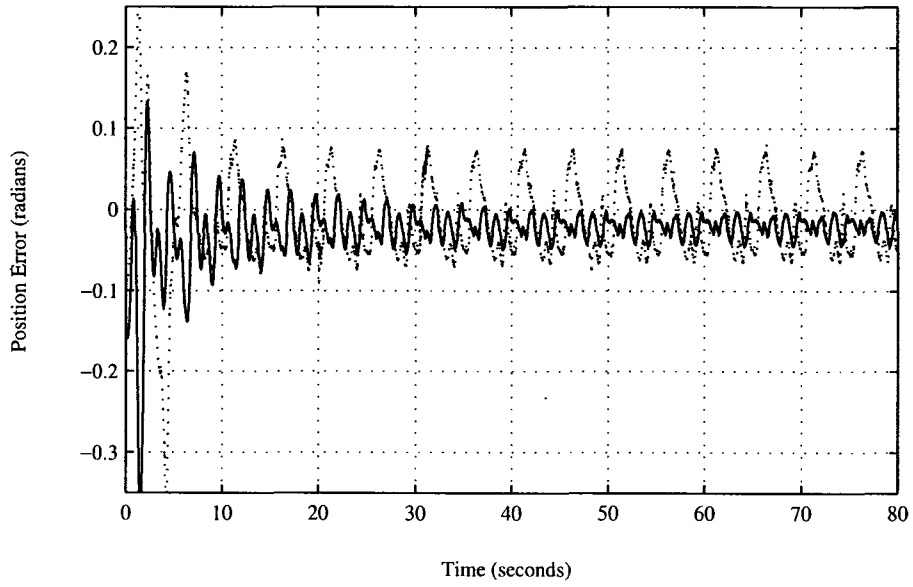

Figure 4.36: Position Error for link 1 (solid) and link 2 (dotted) during first 80 seconds using Craig's method with a linear observer, $\Gamma = 0.1I$.

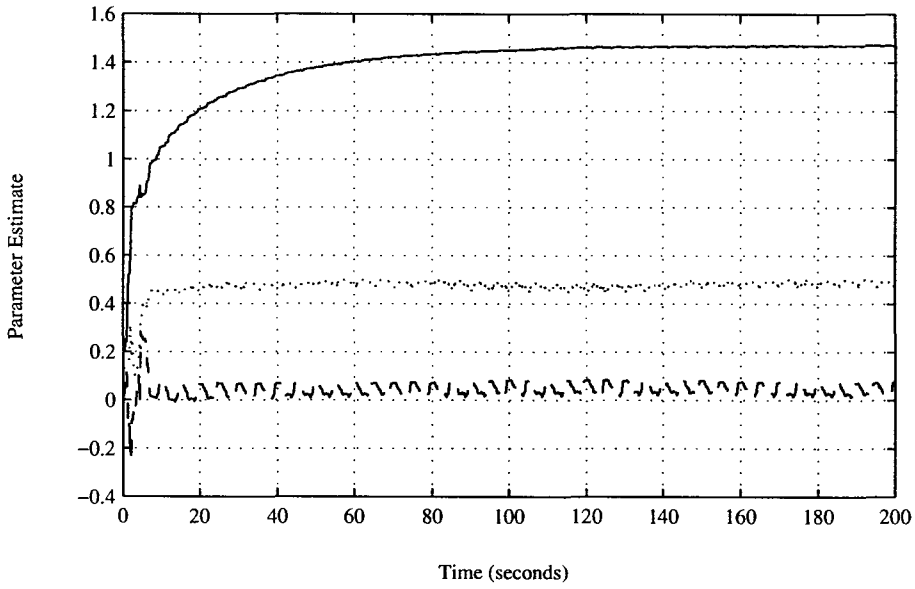

Figure 4.37: Parameter estimates over time for θ_1 (solid), θ_2 (dotted), and θ_3 (dashed) using Craig's method with a linear observer, $\Gamma = 0.1I$.

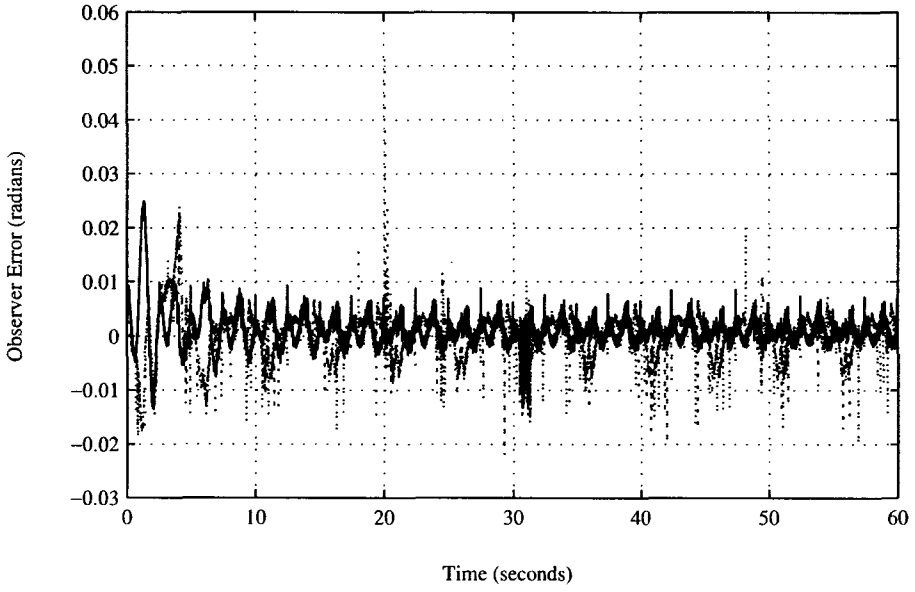

Figure 4.38: Error on the position estimates using Craig's algorithm with a linear observer on the Direct-Drive Robot for link 1 (solid) and link 2 (dotted), $\Gamma = 0.1I$.

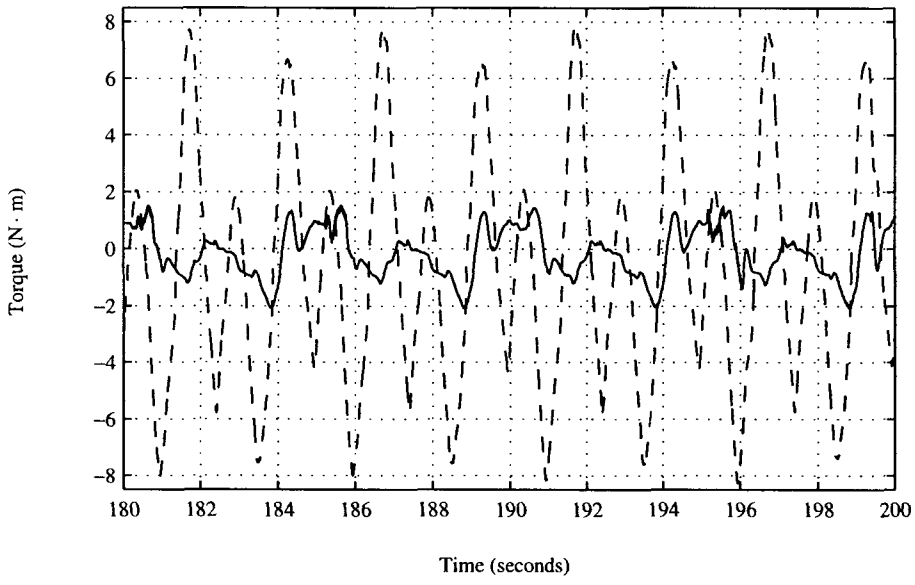

Figure 4.39: Computed torques used to drive the robot links for link 1 (dashed) and link 2 (solid) computed using Craig's method with a linear observer, $\Gamma = 0.1I$.

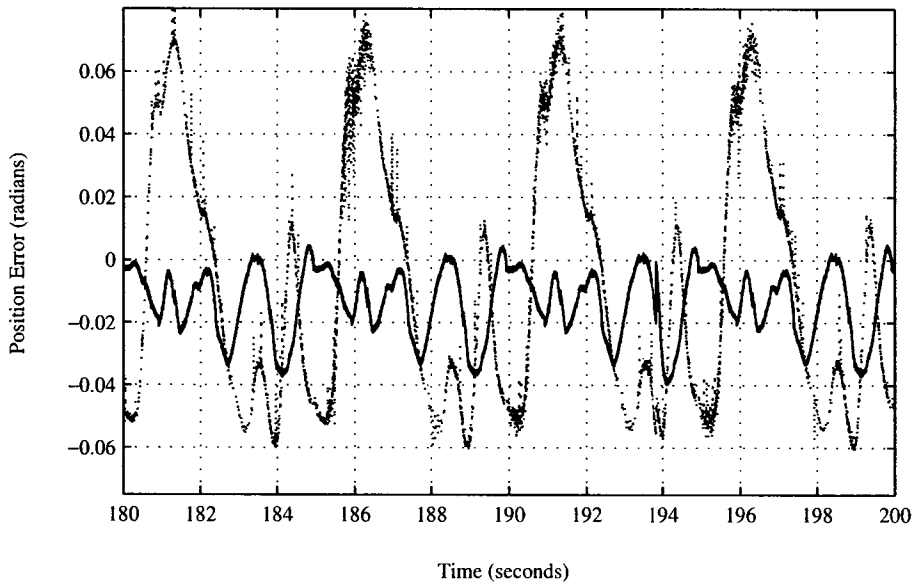

Figure 4.40: Position Error for link 1 (solid) and link 2 (dotted) after 180 seconds using Craig's method with a linear observer, $\Gamma = 0.05I$.

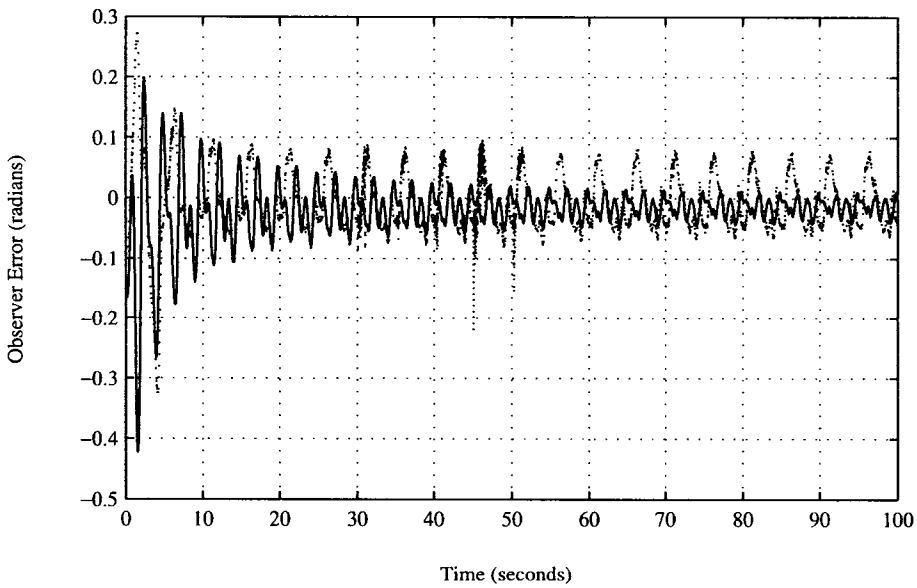

Figure 4.41: Position Error for link 1 (solid) and link 2 (dotted), first 100 seconds using Craig's method with a linear observer, $\Gamma = 0.05I$.

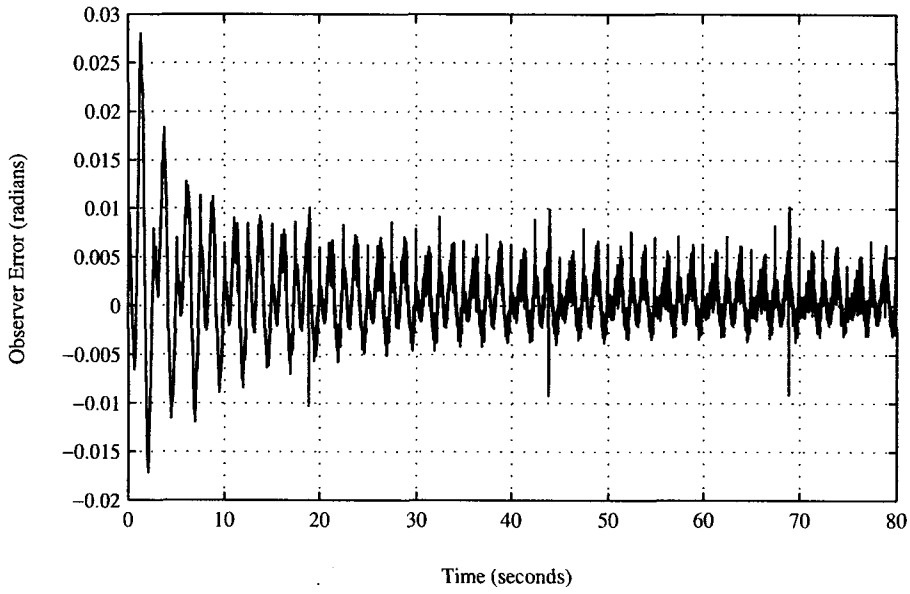

Figure 4.42: Error on the position estimate of link 1 using Craig's algorithm with a linear observer, $K_1 = 20$, $K_2 = 500$.

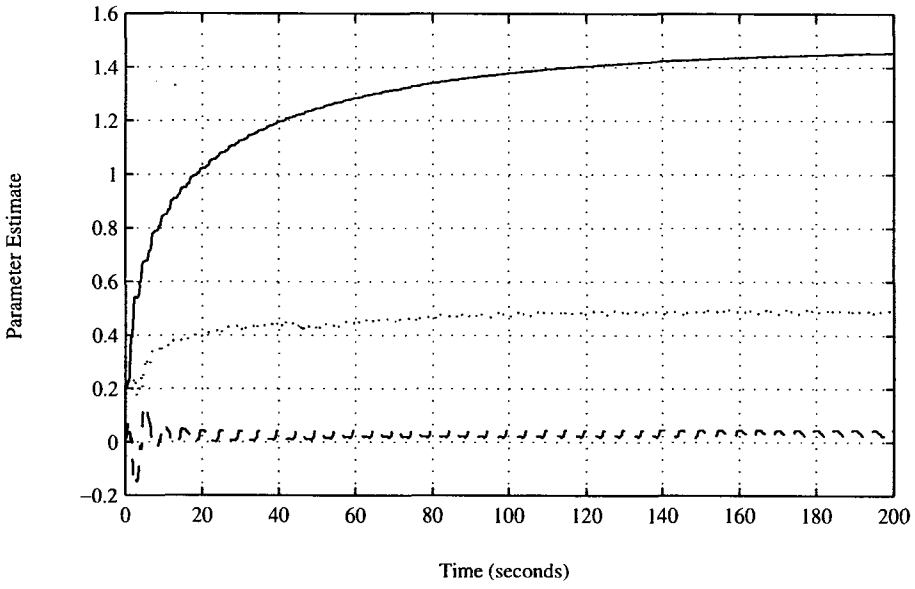

Figure 4.43: Parameter estimates over time for θ_1 (solid), θ_2 (dotted), and θ_3 (dashed) using Craig's method with a linear observer, $\Gamma = 0.05$.

Experiment with Larger Observer Gains

In order to examine the effect of different placement of the observer poles, an experiment was run with observer gains set to $K_1 = 50$ and $K_2 = 625$. These gains place the two observer poles at $s = -25$, which results in critically damped observer error dynamics. The adaptation gain was set to $\Gamma = 0.05I$. The tracking error at steady state for this experiment is given in Fig. 4.44. One can see that it is very similar to the tracking error of Fig. 4.40, an experiment with underdamped observer error dynamics and smaller observer gain. The observer error for the link 1 position signal is given in Fig. 4.45. It can be seen here that the observer error is very close in magnitude to that of Fig. 4.42. However, in this case there is less peaking in the error due to the critically damped observer poles.

Results of the Experiments

The experiments using Craig, Hsu, and Sastry's algorithm with the addition of a linear observer examined the effect of varying both adaptation gain and observer gain. It was found that the adaptation gain was limited to an upper bound of $\Gamma = 0.1I$. Values larger than that led to unacceptable performance of the robot. An experiment was performed with this upper bound as the adaptation gain, using observer gains that yield underdamped observer dynamics. The tracking performance of this setup is good. However, since the value used for adaptation gain is close to the point where the algorithm diverges, a second experiment was performed with a smaller adaptation gain, $\Gamma = 0.05I$. At a steady state this experiment performed as well as the previous one with respect to position tracking. However, it took much longer for the tracking error to converge to a steady state. As well, after 200 seconds the inertial parameter estimates had not yet reached a steady state.

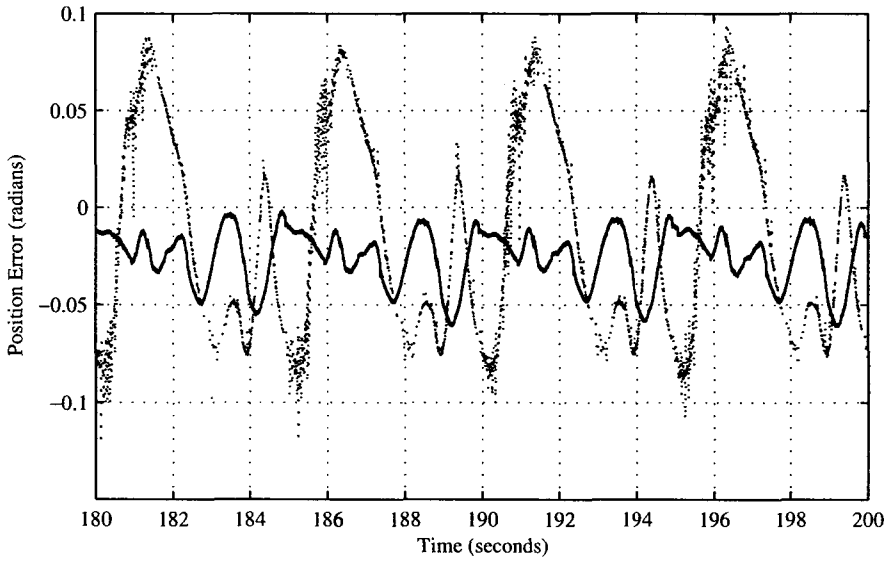

Figure 4.44: Position Error for link 1 (solid) and link 2 (dotted), after 180 seconds using Craig's method with a linear observer, $K_1 = 50$, $K_2 = 625$.

A third experiment was conducted to examine the effects of the observer gains. The gains were changed from $K_1 = 20$, $K_2 = 500$, to $K_1 = 50$, $K_2 = 625$. This moves the observer error poles from underdamped at $s = -10 \pm 20j$ to critically damped at $s = -25$. It was found that the position error at steady state was comparable. As well, the time required for the error on the position estimates to converge to a steady state was also comparable. As expected, there was slightly more overshoot on the observer error associated with the underdamped poles. This suggests that it is possible to use smaller observer gains with underdamped poles to achieve performance similar to larger observer gains and critically damped poles with this algorithm. It could be desirable to use lower observer gains so that noise on the measured states is not amplified as significantly and propagated through the system.

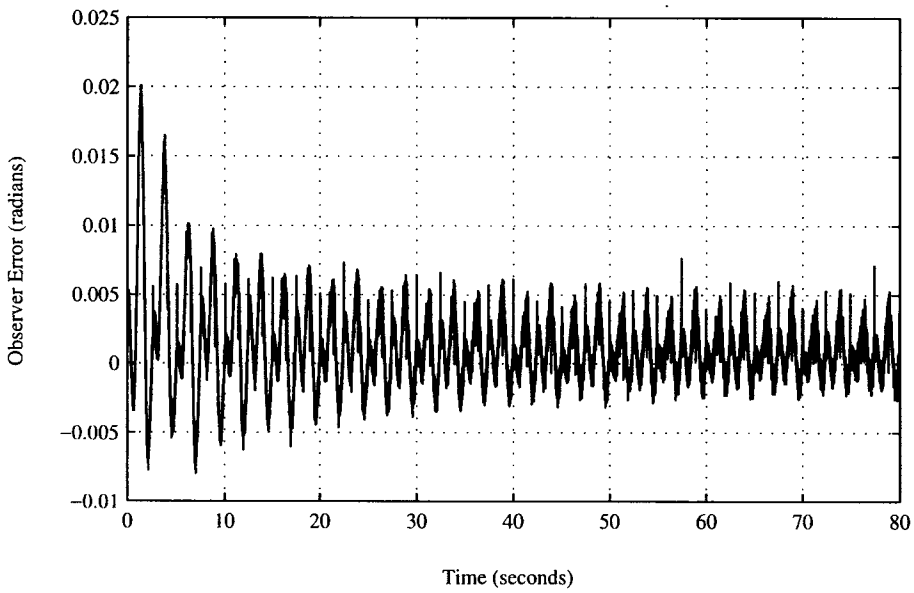

Figure 4.45: Error on the position estimate of link 1 using Craig's algorithm with a linear observer, $K_1 = 50$, $K_2 = 625$.

4.5 Discussion of Experimental Results

Experimental results for the three algorithms examined were presented in this chapter. All experiments were performed on the same robot platform – the Carleton University Direct-Drive Robot. With these types of algorithms there are many parameters which can be set, so significant effort was made in order to be clear about which parameters were changed from experiment to experiment. In the experimental results section for each of the algorithms, the focus was to compare the algorithm against itself. In this section the goal is to compare the algorithms against each other.

Each of the algorithms examined had specific strengths and weaknesses. Fig. 4.46 shows the position error on link 1 of the Direct-Drive Robot over time for each of the three algorithms studied. Fig. 4.47 shows the position error on link 2 of the Direct-Drive Robot over time for the same algorithms. From these graphs it is clear that Lee and Khalil's algorithm consistently demonstrates the lowest tracking error. Craig's algorithm with our observer follows Lee and Khalil's algorithm in tracking performance, with slightly greater tracking error. Gourdeau and Schwartz's method shows the largest tracking error in both of the robot links.

Gourdeau and Schwartz's algorithm yielded convergence of the inertial parameter estimates that was quite rapid as well as being relatively free of oscillations. This combination of rapid parameter convergence that does not oscillate significantly is very desirable when performing system identification. This is also useful in helping to bring the tracking error to its steady-state value fairly quickly. The state estimates produced by the EKF were also relatively free of noise. This is an important factor in ensuring that the control signals computed are also relatively free of noise. However, the position tracking performance of this algorithm was noticeably larger

than either of the other two. This could be due to inaccuracy of the robot model with respect to the real world, and the sensitivity of the Extended Kalman Filter to that discrepancy [13]. Another factor contributing to this could also be that the matrix $R(t)$ could only take on values as low as 2×10^{-4} in this implementation before the algorithm was destabilized. Perhaps a more numerically stable form of the filter would allow smaller values for $R(t)$.

The algorithm proposed by Lee and Khalil has the advantage of the smallest trajectory tracking error. The high-gain observer used with this method allows fast convergence of the observer error, and it is quick to approach the performance of full-state feedback control [5]. This accuracy in state estimates helps in reducing tracking error since state estimates very close to the true states are passed to the control law to compute the torques. However, use of the high-gain observer makes this system quite sensitive to noise. As measurement noise is amplified by the observer, this contributes to a very noisy control signal that causes vibrations in the robot links. The high frequencies in the control signal excite unmodeled system dynamics. While tracking performance is very good in the experiments, the problem of vibrating links limits the practical usefulness of this algorithm.

The adaptive control algorithm proposed by Craig, Hsu, and Sastry, with the addition of a linear observer that we propose yielded positive results in the experiments. The tracking error was reasonably small in magnitude, although not as small as Lee and Khalil's algorithm. As well, this method produces quite clean torque signals with which to drive the robot. This is due to the fact that the observer gains are much smaller than those used in Lee and Khalil's method. One issue with this algorithm is the limitation on the adaptation gain used in practice. Values above $\Gamma = 0.1I$ resulted in divergence of the algorithm. While tracking error is not affected by this

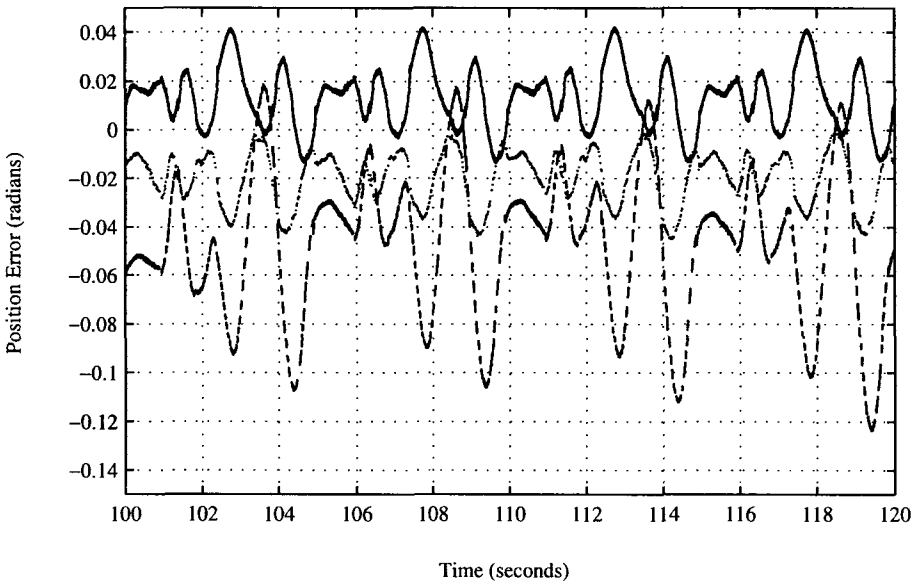

Figure 4.46: Link 1 position error for Lee and Khalil's algorithm (solid line), Craig's algorithm with observer (dotted line), Gourdeau and Schwartz's algorithm (dashed line).

restriction at steady state, the transient performance of the tracking error is poor. As well, in a situation where the inertial parameters of the robot varied over time, this algorithm could result in slow adaptation to the change in parameters. The result of this would be increased tracking error until the inertial parameter estimates have reached a steady state.

When one considers all of the issues with each of the algorithms, the best results in this case were obtained using Craig, Hsu, and Sastry's algorithm with the addition of a linear observer to allow only output feedback. This approach yields low tracking error and reasonable parameter convergence. As well, it produces consistent results and does not excite high frequency or unmodeled dynamics of the robot.

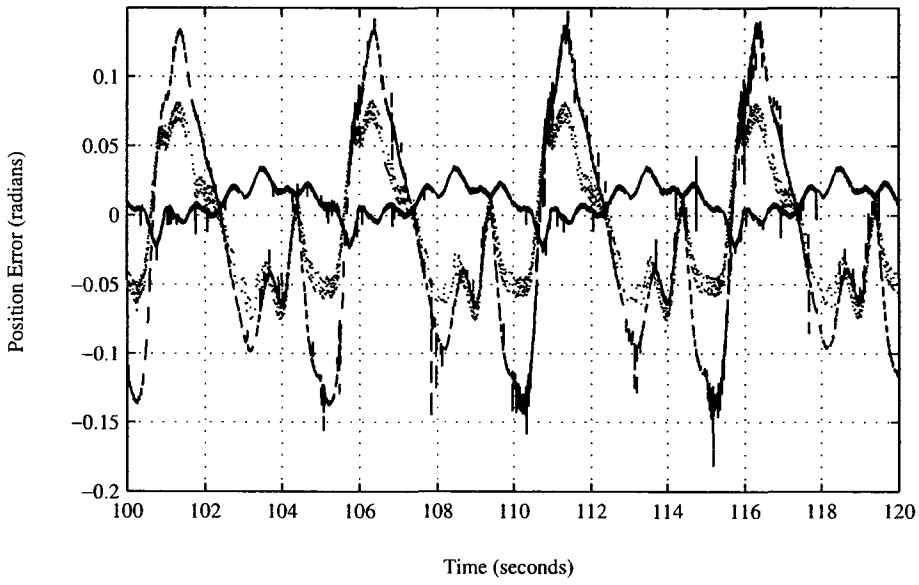

Figure 4.47: Link 2 position error for Lee and Khalil's algorithm (solid line), Craig's algorithm with observer (dotted line), Gourdeau and Schwartz's algorithm (dashed line).

Chapter 5

Conclusions

5.1 Discussion of Results

The research performed in this work has involved comparing the performance of three nonlinear output feedback adaptive control algorithms in simulation and through experiments. It was shown that the three algorithms can exhibit very different performance with respect to trajectory tracking and parameter estimation on the same robot platform. In particular, an important contribution made in this work includes results which demonstrate that, while an algorithm may perform well in simulation, experimental implementation can lead to unacceptable performance. This unacceptable performance can be caused by the presence of unmodeled dynamics in a real robot. A dynamic model is not always a perfect representation of the true system, and discrepancies can lead to implementation problems.

Chapter 2 was a theoretical review of nonlinear adaptive control algorithms for robot manipulators. Two earlier full-state feedback algorithms were discussed, followed by the three output feedback algorithms examined in detail in this work. Fol-

lowing a theoretical discussion, simulation results for the three output feedback algorithms were presented in Chapter 3. Chapter 4 presented experimental results for each of the algorithms when run on the Carleton University Direct-Drive Robot. This work has focused on comparing each of the algorithms presented in order to better gauge their relative performances. The effect of varying several parameters in each of the algorithms was studied by varying those parameters and performing repeated simulations and experiments. This has allowed a more in-depth understanding of the dynamic interactions of the various components (controller, observer, and adaptation law) in each of the algorithms.

Gourdeau and Schwartz's algorithm, based on an Extended Kalman Filter, performed well in simulation with respect to both tracking performance and inertial parameter estimation. It exhibited a larger peak tracking error in the transient period than the other approaches. This is likely due to initially large error on the estimates of the robot states. However, at steady-state in simulation the tracking performance of this algorithm was within the same order of magnitude as the other approaches. Introduction of noise on the position measurements in the simulated system yielded a control signal that remained relatively free of noise. As compared to the other algorithms, the control signals generated using this approach contained the least amount of noise. This is due to the noise rejection properties of the EKF.

Implementation of Gourdeau and Schwartz's algorithm on the experimental platform yielded tracking error at steady-state that was more than double that of the experimental results of Craig's algorithm with a linear observer. Based on simulation results, this behaviour was not expected. However, this result could be due to inaccuracies in the dynamic model of the experimental platform leading to biases in the state estimates produced by the EKF. As well, the experimental performance

of this algorithm was tuned by systematically increasing the values of the process noise covariance matrix $Q(t)$ to achieve improved performance. This suggests that the values chosen for the noise covariance matrices in the experiments may not be representative of the actual noise in the system, as the covariance matrices were not set using an analytic approach.

The algorithm proposed by Lee and Khalil yielded excellent tracking performance in simulation in cases free of noise as well as cases with measurement noise. However, when this algorithm was implemented experimentally different results were observed. It was seen that the high-gain observer of this algorithm amplified noise in the system and contributed to a noisy control signal. This control signal excited high frequency unmodeled dynamics in the experimental robot platform. This caused the robot to undergo much vibration throughout some of the experiments. In one case this vibration led to divergence of the experiment. The adaptation gain Γ was varied to study its effect on the vibrations. Through these experiments it was found that the occurrence of the vibrations in the robot is not predictable and is not a function of adaptation gain. In practical implementations of such an algorithm this can be a serious limitation. High frequency vibration of the robot manipulator can negatively affect the performance of the system.

The algorithm proposed by Craig with the addition of a linear observer proposed herein performed well in simulation as well as through experimentation. A unique feature of this approach, as compared to the other two methods, is the use of a simple linear system model in the observer. The other approaches use the full nonlinear dynamic model of the robot for state estimation. However, with this approach using a linear observer each link in the linearized system is treated as a double integrator. The dynamics of the observer reflect that, and as a result the sensitivity of the observer to

unmodeled dynamics is decreased. This assumes that the computed torque controller is in fact linearizing the nonlinear system.

Aitken [6] discusses the issues of robustness to unmodeled dynamics and the presence of noise on the system measurements with respect to observer design. It is desirable to design observers that are both robust to modeling error and insensitive to noise on the system measurements. However, such goals are in conflict with one another [6]. Use of high observer gains can produce robustness to unmodeled dynamics, but in order to reduce the observer's sensitivity to measurement noise, low observer gains must be used. This insight is very useful in explaining the observations made about the effectiveness of the algorithms in this work. The high gain of Lee and Khalil's observer ensures that it is robust to unmodeled dynamics, while its sensitivity to measurement noise is apparent in the experimental data. The Extended Kalman Filter of Gourdeau and Schwartz's work is not a high-gain observer, and it appears to be more sensitive to modeling errors. However, it rejects measurement noise much better than Lee and Khalil's algorithm. Craig's algorithm with the addition of a linear observer is not a high gain observer and, being based on a simple linear system model, is not as sensitive to error in the nonlinear dynamic model. The success of this approach is encouraging in that it shows less sensitivity to measurement noise as well as unmodeled dynamics, as compared to the other approaches.

5.2 Future Work

An important issue for future work relating to this research is the development of a proof of stability for Craig's algorithm with the addition of a linear observer. The promising simulation and experimental results justify further research in that area.

Based on the experimental results presented in this work, it is recommended that model verification be performed using the known dynamic model for the Carleton University Direct-Drive Robot to ensure that it remains an accurate model.

Another interesting avenue for future work would be to attempt to implement Lee and Khalil's algorithm in simulation and recreate the vibrations observed in the experiments. This would likely involve use of simulated robot dynamics that contain high frequency and other dynamics which do not appear in the robot model used in the algorithm. Such a setup would allow for more comprehensive testing of the vibrating phenomenon observed in the experiments in this work.

References

[1] J. J. Craig, P. Hsu, and S. S. Sastry, "Adaptive control of mechanical manipulators," *The International Journal of Robotics Research*, vol. 6, no. 2, pp. 16–27, 1987.

[2] J.-J. E. Slotine and W. Li, "On the adaptive control of robot manipulators," *The International Journal of Robotics Research*, vol. 6, no. 3, pp. 49–59, 1987.

[3] H. Hajjir and H. M. Schwartz, "An adaptive nonlinear output feedback controller for robot manipulators," in *Proceedings of the American Control Conference*, June 1999.

[4] R. Gourdeau and H. M. Schwartz, "Adaptive control of robotic manipulators using an extended kalman filter," *Journal of Dynamic Systems, Measurement, and Control*, vol. 115, pp. 203–208, Mar. 1993.

[5] K. W. Lee and H. K. Khalil, "Adaptive output feedback control of robot manipulators using high-gain observer," *International Journal of Control*, vol. 6, pp. 869–886, 1997.

[6] V. C. Aitken, "Sliding mode state estimation for nonlinear discrete time systems: Applications in image sequence analysis," PhD Dissertation, Carleton University, Ottawa, Ontario, Apr. 1995.

[7] C. H. An, C. G. Atkeson, and J. M. Hollerbach, "Model-based control of a direct drive arm, part i: Building models," in *IEEE International Conference on Robotics and Automation*, Apr. 1988.

[8] R. Gourdeau, "Adaptive control of robotic manipulators," PhD Dissertation, Carleton University, Ottawa, Ontario, Apr. 1991.

[9] H. M. Schwartz, G. Warshaw, and T. Janabi, "Issues in robot adaptive control," in *Proceedings of the American Control Conference*, 1990.

[10] P. R. Pagilla and M. Tomizuka, "An adaptive output feedback controller for robot arms: stability and experiments," *Automatica*, vol. 37, pp. 983–995, July 2001.

[11] H. M. Schwartz, "Model reference adaptive control for robotic manipulators without velocity measurements," *International Journal of Adaptive Control and Signal Processing*, vol. 8, pp. 279–285, 1994.

[12] R. Gourdeau and H. M. Schwartz, "Adaptive control of robotic manipulators: Experimental results," in *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*, Apr. 1991, pp. 8–15.

[13] A. Gelb, Ed., *Applied Optimal Estimation.* Massachusetts: The M.I.T. Press, 1974.

[14] S. Rajguru, "Analysis and design of a direct drive robot," Master's Thesis, Carleton University, Ottawa, Ontario, Dec. 1989.

[15] G. D. Warshaw, "Investigations of adaptive control for a direct drive robotic manipulator," Master's Thesis, Carleton University, Ottawa, Ontario, Jan. 1990.

[16] G. Warshaw, "Sampled-data robot adaptive control," PhD Dissertation, Carleton University, Ottawa, Ontario, Jan. 1994.

[17] M. S. Grewal and A. P. Andrews, *Kalman Filtering: Theory and Practice.* New Jersey: Prentice Hall Inc., 1993.

Appendix A

Linux Device Driver Code Listing

```
/*
 * This is the device driver for Bobo, the Carleton University
 * Direct Drive Robot.
 */

// Required for building a module:
#include <linux/module.h>
// To allocate and free io ports:
#include <linux/ioport.h>
// Needed to register the character driver:
#include <linux/fs.h>
// For the ioctl numbers to control the DT28xx cards:
#include <linux/ioctl.h>
// the capabilities, ie CAP_DAC_OVERRIDE:
#include <linux/capability.h>
// To call the capable function (also for delaying):
#include <linux/sched.h>
// current structure comes from here, I believe:
#include <linux/tty.h>
// To implement the spinlocks:
#include <linux/spinlock.h>
// This defines HZ for timeouts:
#include <linux/param.h>
// for kmalloc and kfree:
#include <linux/slab.h>
// to get the system time, for debugging purposes:
#include <linux/time.h>
// for the waitqueue functions:
#include <linux/wait.h>
// For the in and out functions to access the device:
```

```
#include <asm/io.h>
// for the rmb, wmb functions:
#include <asm/system.h>
// for the atomic functions.:
#include <asm/atomic.h>
// for the functions to copy to and from the user:
#include <asm/uaccess.h>

/* Some macros to split minors in two parts */
#define TYPE(dev)   (MINOR(dev) >> 4)  /* high nibble */
#define NUM(dev)    (MINOR(dev) & 0xf) /* low  nibble */


MODULE_AUTHOR("John M. Daly");
MODULE_DESCRIPTION("This module is used to interface with the
  Carleton University Direct Drive Robot.");
MODULE_SUPPORTED_DEVICE("This module interfaces directly with the
  DT2811 AD/DA and DT2817 Digital Interface Cards");

/* Here we define port offsets to the DT2811 and DT2817: */
// First the DT2811 (ADDA):

// offset for AD Gain Control Register:
#define ADGCR_OFFSET 0x1
// offset for dac0 data low byte/ad data low byte:
#define DADAT0_LOW_OFFSET 0x2
// offset for dac0 data high byte/ad data high byte:
#define DADAT0_HIGH_OFFSET 0x3
// offset for dac1 data low byte:
#define DADAT1_LOW_OFFSET 0x4
// offset for dac1 data high byte:
#define DADAT1_HIGH_OFFSET 0x5
// offset for timer control register:
#define TMRCTR_OFFSET 0x7

// Now the DT2817 (DD):

// offset for motor 0 position low byte:
#define DD_PORT0_OFFSET 0x1
// offset for motor 0 position high byte:
#define DD_PORT1_OFFSET 0x2
// offset for motor 1 position low byte:
#define DD_PORT2_OFFSET 0x3
// offset for motor 1 position high byte:
#define DD_PORT3_OFFSET 0x4

// Some constants we send to the DT2811 for initializing it:

// clears the DT2811 error flag bit:
```

```
#define CLEAR_ADDA_ERROR 0x10
// enable interrupts on the DT2811:
#define SET_ADDA_INT 0x4
// Mode 1 for DT2811 to interrupt on internal clock:
#define MODE 0x1

/* We define the default base ports and ranges here */

// Data Translation DT2811 AD/DA Card:
static int adda_base_address = 0x248;
static int adda_range = 0x008;

// The MODULE_PARM makes this variable settable at module load
// time.
MODULE_PARM (adda_base_address, "i");
MODULE_PARM_DESC (adda_base_address, "Base I/O port for the DT2811
  AD/DA Card (Default: 0x248)");

// Data Translation DT2817 Digital Interface Card:
static int dd_base_address = 0x308;
static int dd_range = 0x005;
MODULE_PARM (dd_base_address, "i");
MODULE_PARM_DESC (dd_base_address, "Base I/O port for the DT2817
  Digital Interface Card (Default: 0x308)");

// The interrupt line used by the DT2811 to generate interrupts:
static int bobo_irq = 7;
MODULE_PARM (bobo_irq, "i");
MODULE_PARM_DESC (bobo_irq, "The interrupt line that the DT2811
  uses for interrupts (Default: 7)");

// The major number we want for our device:
static int bobo_major = 60;
MODULE_PARM (bobo_major, "i");
MODULE_PARM_DESC (bobo_major, "Major number requested for the Bobo
  device (Default: 60)");

// The timer frequency we want the DT2811 to interrupt at:
// (see p. 5-16 of DT2811 manual for setting this value.)
// Note: The timer frequency should be twice the desired sampling
// frequency for the system, since it only gets info from 1 motor
// at each interrupt. (So, set to 200 Hz, data would be read from
// each of the motors, alternating between interrupts, giving a net
// frequency of 100 Hz for everything.)
static int timer_frequency = 0x1B;
MODULE_PARM (timer_frequency, "i");
MODULE_PARM_DESC (timer_frequency, "The frequency to interrupt at.
  (Should be twice desired sampling frequency). See p. 5-16 of
  DT2811 manual for setting this. (Default: 0x1B = 200 Hz)");
```

```
// We have these variables around so the ISR can get them,
// and the read functions can make use of them.
unsigned int position0, position1;
unsigned int velocity0, velocity1;
unsigned int torque0, torque1;

// The ISR sets these so the read method will know when data
// for both channels is ready. We implement these as atomic
// operations that protect access to the status variables defined
// below:
atomic_t channel0_ready;
atomic_t channel1_ready;

// The write methods sets this so the ISR knows when
// data (torque signals) is available to write to the
// device:
atomic_t torque_ready;

// We define a wait queue for the read process to sleep while
// it waits for data to become available through the interrupt:
//DECLARE_WAIT_QUEUE_HEAD (bobo_rq);

// This is the channel to perform data conversion on. The ISR uses
// this to set itself up for conversions on different channels.
unsigned int bobo_channel = 0;

// Debug stuff:
//unsigned int num_reads; // Tells us how many times the read
// method is called

// We now need the methods to perform various operations on the
// device, such as reading and writing.

// Here we have the interrupt handler:
void bobo_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
   unsigned char posLowByte;
   unsigned char posHighByte;

   unsigned char velLowByte;
   unsigned char velHighByte;

   unsigned char t0LowByte;
   unsigned char t0HighByte;

   unsigned char t1LowByte;
   unsigned char t1HighByte;
```

```
// We use these to ensure the value of the DT2817 registers
// are not changing as we read them:
unsigned int w1, w2;

/* We get the data that has been collected from the Analog to
   Digital Conversion. */
if (bobo_channel == 0) { // we are working on channel 0:

   // get the velocity data from the DT2811
   velLowByte = inb(adda_base_address + DADAT0_LOW_OFFSET);
   rmb();
   velHighByte = inb(adda_base_address + DADAT0_HIGH_OFFSET);

   // And read position from the DT2817
   posLowByte = inb(dd_base_address + DD_PORT0_OFFSET);
   rmb(); // We need to ensure that things happen in this order
   posHighByte = inb(dd_base_address + DD_PORT1_OFFSET);

   w1 = (((posHighByte << 8) + posLowByte) & 0xFFF);

   posLowByte = inb(dd_base_address + DD_PORT0_OFFSET);
   rmb(); // We need to ensure that things happen in this order
   posHighByte = inb(dd_base_address + DD_PORT1_OFFSET);

   w2 = (((posHighByte << 8) + posLowByte) & 0xFFF);

   // This loop ensure the values weren't changing as we read
   // them:
   while (w1 != w2) {

      w2 = w1;

      posLowByte = inb(dd_base_address + DD_PORT0_OFFSET);
      // We need to ensure that things happen in this order
      rmb();
      posHighByte = inb(dd_base_address + DD_PORT1_OFFSET);

      w1 = (((posHighByte << 8) + posLowByte) & 0xFFF);

   } // end of while

   // Now, we only want to modify these values if the read()
   // function is done with them:
   if (atomic_read(&channel0_ready) == 0) {
      // convert the position to one unsigned int:
      position0 = (((posHighByte << 8) + posLowByte) & 0xFFF);

      // same for velocity:
      velocity0 = (((velHighByte << 8) + velLowByte) & 0xFFF);
```

```
            // Now we atomically set the value of channel0_ready:
            atomic_set(&channel0_ready, 1);

            // Now wake up the potentially sleeping read process:
            //wake_up_interruptible(&bobo_rq);

         }

      } else if (bobo_channel == 1) { // we are working on channel 1

         // get the velocity data from the DT2811
         velLowByte = inb(adda_base_address + DADAT0_LOW_OFFSET);
         rmb();
         velHighByte = inb(adda_base_address + DADAT0_HIGH_OFFSET);


         // And read position from the DT2817
         posLowByte = inb(dd_base_address + DD_PORT2_OFFSET);
         rmb(); // We need to ensure that things happen in this order
         posHighByte = inb(dd_base_address + DD_PORT3_OFFSET);

         w1 = (((posHighByte << 8) + posLowByte) & 0xFFF);

         posLowByte = inb(dd_base_address + DD_PORT2_OFFSET);
         rmb(); // We need to ensure that things happen in this order
         posHighByte = inb(dd_base_address + DD_PORT3_OFFSET);

         w2 = (((posHighByte << 8) + posLowByte) & 0xFFF);

         // This loop ensure the values weren't changing as we read
         // them:
         while (w1 != w2) {

            w2 = w1;

            posLowByte = inb(dd_base_address + DD_PORT2_OFFSET);
            // We need to ensure that things happen in this order
            rmb();
            posHighByte = inb(dd_base_address + DD_PORT3_OFFSET);

            w1 = (((posHighByte << 8) + posLowByte) & 0xFFF);

         } // end of while

         // Now, we only want to modify these values if the read()
         // function is done with them:
         if (atomic_read(&channel1_ready) == 0) {
```

```
        // convert the position to one unsigned int:
        position1 = (((posHighByte << 8) + posLowByte) & 0xFFF);

        // same for velocity:
        velocity1 = (((velHighByte << 8) + velLowByte) & 0xFFF);

        // Now we atomically set the value of channel1_ready:
        atomic_set(&channel1_ready, 1);

        // Now wake up the potentially sleeping read process:
        //wake_up_interruptible(&bobo_rq);

    }

    // Next, we write out torques to the robot if they
    // have been made available by the write method:

    if (atomic_read(&torque_ready) == 1) {
        // then there are torques ready.

        // Write them out here:
        // Low byte of torque 0:
        t0LowByte = (unsigned char)(torque0 & 0xFF);
        // High byte of torque 0
        t0HighByte = (unsigned char)((torque0 >> 8) & 0xF);

        // Low byte of torque 0
        t1LowByte = (unsigned char)(torque1 & 0xFF);
        // High byte of torque 0
        t1HighByte = (unsigned char)((torque1 >> 8) & 0xF);

        // Now we write this data out to the robot:
        outb(t0LowByte, adda_base_address + DADAT0_LOW_OFFSET);
        // Want to preserver the order of these instructions, so
        // use a memory barrier.
        wmb();
        outb(t0HighByte, adda_base_address + DADAT0_HIGH_OFFSET);

        outb(t1LowByte, adda_base_address + DADAT1_LOW_OFFSET);
        wmb();
        outb(t1HighByte, adda_base_address + DADAT1_HIGH_OFFSET);

        // Debug stuff:
        /* printk("<1>bobo: Received - Torque0: %u Torque1: %u\n",
           torque0, torque1); */

        // Now we clear that torque ready flag:
        atomic_set(&torque_ready, 0);
```

```
      } // end of if torque ready


   } // end of if else for channels

   // Now we change the channel:
   bobo_channel = (bobo_channel + 1) % 2;

   // set up the DT2811 to convert AD on the new current channel:
   outb(0x3*bobo_channel, adda_base_address + ADGCR_OFFSET);

   //printk("<1>bobo: Received an interrupt!\n");
   //printk("<1>bobo: Position0: %i\n", position0);
   //printk("<1>bobo: Position1: %i\n", position1);


} // end of bobo_interrupt

/* This function is called when the device is opened by a program.
   It increments the usage count... */

/* One thing we want to ensure here is that only one user can open
   the device at a time. The user can run multiple processes on it,
   but just one user. This way, some other user won't be able to
   send some crazy control signal to the robot, while the current
   user is controlling it. */

int bobo_u_count = 0; // Number of users accessing it
uid_t bobo_u_owner = 0;
spinlock_t bobo_u_lock;

int bobo_open(struct inode *inode, struct file *filp)
{
   int result; // to store the result of our irq request in
   int num = NUM(inode->i_rdev);

   if (!filp->private_data && num > 0)
      return -ENODEV; /* not devfs: allow 1 device only */
   spin_lock(&bobo_u_lock); /* Entering a Critical Section! Need to
   lock */
   if (bobo_u_count &&
      (bobo_u_owner != current->uid) &&  /* allow user */
      (bobo_u_owner != current->euid) && /* allow whoever did su */
         !capable(CAP_DAC_OVERRIDE)) { /* still allow root */
      spin_unlock(&bobo_u_lock);
      return -EBUSY;
   }

   if (bobo_u_count == 0)
```

```
      bobo_u_owner = current->uid; /* grab it */

   bobo_u_count++;
   spin_unlock(&bobo_u_lock); /* End of Critical Section! */

   /* Here, we register our interrupt handler: (We do
    * this at open, so the interrupt line is only occupied
    * when there's an actual process using the device. This way
    * the IRQ can be free when the device isn't in use. ) */

   /* We only want to do this if the module is being opened for the
    * first time. */
   if (!MOD_IN_USE) {

      if (bobo_irq >= 0) {
         result = request_irq(bobo_irq, bobo_interrupt,
         SA_INTERRUPT, "bobo", NULL);

         if (result) {
            printk(KERN_INFO "bobo: can't get assigned irq %i\n",
              bobo_irq);
            bobo_irq = -1;
         }
         else { /* Actually enable interrupting on the device. */
            // (From Ian Showalter's code)
            // Send the DT2811 the clear error, enable interrupts,
            // and mode 1 mdessage:
            outb(CLEAR_ADDA_ERROR + SET_ADDA_INT + MODE,
              adda_base_address);
            // Set the timer on the DT2811 to interrupt at the
            // frequency related to timer_frequency
            outb(timer_frequency, adda_base_address +
              TMRCTR_OFFSET);

            /* We now start the interrupts by loading the A/D
               Gain/Channel
               Register with selected gain and channel number. */
            outb(0, adda_base_address + ADGCR_OFFSET);

            // We want to initialize our atomic variables so data
            // is not ready intially, until an interrupt occurs and
            // changes that:
            atomic_set(&channel0_ready, 0);
            atomic_set(&channel1_ready, 0);

            // Likewise for the torque variable, we don't want data
            // to appear ready until the write method provides
            // some:
            atomic_set(&torque_ready, 0);
```

```
         }
      } // end of if bobo_irq >= 0

   } // end of if !MOD_IN_USE

   printk("<1>bobo: Calling Bobo's OPEN function\n");

// Debug:
   //num_reads = 0;
// End Debug

   MOD_INC_USE_COUNT;
   return 0; /* success */

} // End of function bobo_open

/* This function releases the device, and decrements its usage
   count */
int bobo_release(struct inode *inode, struct file *filp)
{

   bobo_u_count--;
   MOD_DEC_USE_COUNT;

   if (!MOD_IN_USE) { // so if the module is now not used:
      // First thing to do is to shutdown the motors by outputting
      // zero from the DA card:
      outb(0, adda_base_address + DADAT0_LOW_OFFSET);
      outb(0x8, adda_base_address + DADAT0_HIGH_OFFSET);
      outb(0, adda_base_address + DADAT1_LOW_OFFSET);
      outb(0x8, adda_base_address + DADAT1_HIGH_OFFSET);

      // Now disable interrupts: (same as what we did to enable,
      // except we clear bit 2 instead of setting it.)
      outb(CLEAR_ADDA_ERROR + MODE, adda_base_address);

      /* Here, we free the interrupt line for the device: */
      if (bobo_irq >= 0) {
         printk("<1>bobo: Freeing Interrupt %i\n", bobo_irq);
         free_irq(bobo_irq, NULL);
      }
   } // end of if !MOD_IN_USE

   printk("<1>bobo: Calling Bobo's RELEASE function\n");
   //printk("<1>bobo: Read was called %u times\n", num_reads);
   return 0;
}
```

```
/* This function exists to allow the driver to seek
   through the device file. In our case, the concept
   of seeking doesn't make sense because we're just receiving
   data on the interrupts, and there is nothing to seek through. */
loff_t bobo_llseek(struct file *filp, loff_t off, int whence)
{
   return -ESPIPE; /* unseekable */
}

/* The function to read from the device */
ssize_t bobo_read(struct file *filp, char *buf, size_t count,
                  loff_t *f_pos)
{
   // We want to check the time, to ensure interrupts are happening
   // when we expect them to. (for debugging)
   struct timeval thetime;

   // We need to allocate some space in kernal-space to put the
   // data into:
   unsigned char *kbuf=kmalloc(count, GFP_KERNEL), *ptr;

   int retval = count; // This is what we will return

   if (!kbuf) return -ENOMEM;
   ptr=kbuf; // the pointer to the beginning of allocated memory.


   // Wait for the data to become available through the interrupt:
   // We will be using a wait queue to sleep with:
   while (atomic_read(&channel0_ready) == 0 || atomic_read(&
     channel1_ready) == 0) {
      //interruptible_sleep_on(&bobo_rq);
      do_gettimeofday(&thetime); // this is a debugging thing

/* This seems to affect the rate at which interrupts are received.
Not good!
      // Tell the kernel we're going to sleep:
      set_current_state(TASK_INTERRUPTIBLE);
      // sleep for this amount of time, in jiffies:
      schedule_timeout(0.0001*HZ);
*/

   }

   // Ok, now we want to put this data in the buffer we have
   // allocated using kmalloc, and then copy it to the user. There
   // are 4 variables of interest here, each using 2 bytes. So, we
   // have 8 bytes to deal with, and we're going to do it all
   // seperately.
```

```
   // The low byte of position 0:
   *(ptr++) = (unsigned char)(position0 & 0xFF);
   // The high byte of position 0:
   *(ptr++) = (unsigned char)((position0 >> 8) & 0xFF);
   // The low byte of position 1:
   *(ptr++) = (unsigned char)(position1 & 0xFF);
   // The high byte of position 1:
   *(ptr++) = (unsigned char)((position1 >> 8) & 0xFF);
   // The low byte of velocity 0:
   *(ptr++) = (unsigned char)(velocity0 & 0xFF);
   // The high byte of velocity 0:
   *(ptr++) = (unsigned char)((velocity0 >> 8) & 0xFF);
   // The low byte of velocity 1:
   *(ptr++) = (unsigned char)(velocity1 & 0xFF);
   // The high byte of velocity 1:
   *(ptr++) = (unsigned char)((velocity1 >> 8) & 0xFF);

   // Now the data is here:
/*   printk("<1>bobo: We are READING DATA\n");
     printk("<1>bobo: Position0: %u Position1: %u Velocity0: %u
       Veloctity1: %u\n", position0, position1, velocity0,
       velocity1);
*/

   // We now must reset the status variables:
   atomic_set(&channel0_ready, 0);
   atomic_set(&channel1_ready, 0);

   // Here we use copy_to_user to get the data over to user space
   if ( (retval > 0) && copy_to_user(buf, kbuf, retval))
      retval = -EFAULT;

// Debug stuff:
   //num_reads++;
// End Debug

   // need to free the memory we allocated with kmalloc:
   kfree(kbuf);

   return retval;
}

/* This function writes to the device (DT2811) */
ssize_t bobo_write(struct file *filp, const char *buf, size_t
  count, loff_t *f_pos)
{

   // We need to allocate some space in kernal-space to put the
```

```
   // data into:
   unsigned char *kbuf=kmalloc(count, GFP_KERNEL), *ptr;

   int retval = count; // This is what we will return

   if (!kbuf) return -ENOMEM;
   // Now we copy the data from the user:
   if (copy_from_user(kbuf, buf, count))
      return -EFAULT;

   ptr=kbuf; // the pointer to the beginning of allocated memory.

   // Now, we only proceed if the ISR is not currently using
   // the toruqe global variables:
   if (atomic_read(&torque_ready) == 0) {
      // Then the torque variables are free

      // We next put the data aquired from user space into our
      // torque variables:
      torque0 = *(ptr++);
      torque0 = torque0 + (*(ptr++) << 8);
      torque1 = *(ptr++);
      torque1 = torque1 + (*(ptr++) << 8);

      if (torque0 > 0xFFF) {
         torque0 = 0xFFF;
      } else if (torque0 < 0) {
         torque0 = 0;
      }
      if (torque1 > 0xFFF) {
         torque1 = 0xFFF;
      } else if (torque1 < 0) {
         torque1 = 0;
      }

      // That's it for the data, tell the ISR it's ready:
      atomic_set(&torque_ready, 1);

   } // end of if torque_ready == 0

   // Debug stuff:
   //printk("<1>bobo: calling WRITE function\n");

   // need to free the memory we allocated with kmalloc:
   kfree(kbuf);

   return retval;

} // end of write method
```

```
/* This method performs IO to set the device up */
int bobo_ioctl(struct inode *inode, struct file *filp,
                 unsigned int cmd, unsigned long arg)
{
   return -ENOTTY;
} // end of bobo_ioctl

// We need to define the file operations structure, so the
// kernel knows what functions to call to access the device:

struct file_operations bobo_fops = {
   llseek:      bobo_llseek,
   read:       bobo_read,
   write:       bobo_write,
   ioctl:       bobo_ioctl,
   open:       bobo_open,
   release:    bobo_release,
};

// This performs all the initialization, such as ensuring the
// motors are receiving a zero output, and setting up the DT2817.
int bobo_init_module(void)
{
   int err; // will use this to see if io ports are available
   int result; // will hold the result of major number request
   unsigned char dummy; // for our dummy input conversions

   // These values are sent to the DT2811 to make it output zero:
   char M0High = 0x8;
   char M1High = 0x8;
   char is = 0;

   // Need to initialize our spin lock for the open fucntion:
   // (Part of the code that only allows one user to use the
   // robot at a time.)
   spin_lock_init(&bobo_u_lock);

   /* Register our major number, based on what is give in the
    * variable bobo_major. */
   result = register_chrdev(bobo_major, "bobo", &bobo_fops);
   if (result < 0) {
      printk(KERN_WARNING "bobo: can't get major %d\n",bobo_major);
      return result;
   }

   /* Request the necessary io ports: */
   // First for the DT2811:
   if ((err = check_region(adda_base_address, adda_range)) < 0) {
```

```
      return err; // Couldn't get the range
   }
   request_region(adda_base_address,adda_range,"bobo");

   // Now for the DT2817:
   if ((err = check_region(dd_base_address, dd_range)) < 0) {
      return err; // Couldn't get the range
   }
   request_region(dd_base_address,dd_range,"bobo");

   /* Initialize the Hardware Here */
   /* The first thing to do is to ensure that the output of the
    * DT2811 card is set to 0 volts. It defaults to -5 V, which
    * will send a large control signal to the robot and have it
    * slam violently.
    * (Ian Showalter wrote a DOS program, DAZERO.EXE, which does
    * that. The functionality is taken from that program and
    * duplicated here.) */
   outb(0, adda_base_address + 0);
   outb(is, adda_base_address + DADAT0_LOW_OFFSET);
   outb(M0High, adda_base_address + DADAT0_HIGH_OFFSET);
   outb(is, adda_base_address + DADAT1_LOW_OFFSET);
   outb(M1High, adda_base_address + DADAT1_HIGH_OFFSET);

   /* Next we initialize the DT2817. We set each of the four
    * ports to be input ports by writing 0 to each of the bits
    * in the control register. (The control register is at the base
    * address.) */
   outb(0, dd_base_address + 0);

   /* Now wait for the hardware to get those instructions */
   /* The DT2811 manual tells us that after initialization we need
    * to wait at least 100 microseconds. We will wait more than
    * enough! */
   /* (This timeout will put the driver to sleep, so other
    * processes can access the processor. This is good!) */
   set_current_state(TASK_INTERRUPTIBLE);
   schedule_timeout(HZ);

   // In Ian Showalter's PID code, after initializing the
   // DT2811, he gets "dummy input conversions from the DT2811".
   // For consistency, we do that here:
   dummy = inb(adda_base_address + DADAT0_LOW_OFFSET);
   dummy = inb(adda_base_address + DADAT0_HIGH_OFFSET);

   printk("<1>bobo: Bobo Module now loaded!\n");
   printk("<1>bobo: It is now SAFE to enable the amplifiers!\n");
   return 0;
}
```

```
void bobo_cleanup_module(void)
{
   // Unregister the major number:
   unregister_chrdev(bobo_major, "bobo");

   // Free up the requested io regions:

   // For the DT2811:
   release_region(adda_base_address, adda_range);

   // For the DT2817:
   release_region(dd_base_address, dd_range);

   printk("<1>bobo: Unloading Module Bobo!\n");
}

// These lines tell the kernel which functions
// to call for initialization and cleanup.
module_init(bobo_init_module);
module_exit(bobo_cleanup_module);
```

Appendix B

Experimental Software Code Listing

This appendix contains the code listing to run Craig, Hsu, and Sastry's algorithm with our linear observer on the Carleton University Direct-Drive Robot. It interfaces with the device driver of Appendix A in order to communicate with the robot.

The experimental code was written based on the simulation code. Additions were made to interface with the robot. As a result, portions of the code in this listing are not necessary for the experimental work (e.g. code to add of noise to the position measurements) but are left in for completeness, while ensuring that their inclusion had no effect on the experimental results.

```
/* This program simulates a non-linear robot manipulator under
 * adaptive control using only output feedback, according to the
 * algorithm given in Craig's paper, with Schwartz's linear
 observer.
 * */

#include <iostream.h>
#include <stdlib.h>
#include <fstream.h>
#include <math.h>
```

```
// The following includes are to interface with the device driver:
// *** Experimental Section ***
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h> // for the open function
#include <unistd.h> // the close function is here
#include <stdlib.h> // for malloc and free
// *** End Experimental Section ***


// Put function prototypes here:
void rk_traj(double y[], double u1, double u2);
void cf_traj(double y[], double f[], double u1, double u2);

void rk_adapt(double y[], double u[]);
void cf_adapt(double y[], double f[], double u[]);

void rk_robot(double y[], double u[]);
void cf_robot(double y[], double f[], double u[]);

void rk_observer(double y[], double u[]);
void cf_observer(double y[], double f[], double u[]);

// Define our constants:
// *** Experimental Section ***
// The number of bytes of memory to allocate for reading
#define NUM_R_BYTES 8
// The number of bytes of memory to allocate for writing
#define NUM_W_BYTES 4

// Conversion constants for Tach 0 in (rad/sec)/V
#define KTACH0 5.7247
// Conversion constants for Tach 1 in (rad/sec)/V
#define KTACH1 4.2586

#define MOTOR0_CONST 1.7 // Conversion contstant for motor in Nm/V
#define MOTOR1_CONST 1.7 // Conversion contstant for motor in Nm/V

// Determines the maximum outlier difference for position
// measurements
#define MAX_OUTLIER 0.3

// *** End Experimental Section ***

#define pi 3.141592653589793238

const double Ts = 0.002; // Sample Period
const double ftime = 200; // How long to run the simulation for.
```

```
// This will be the length of simulation
const int length_t = int(ftime / Ts);

// Specify the input:
double u_link1[length_t]; // Link 1 input
double u_link2[length_t]; // Link 2 input

// And some storage space:
// We need three rows for the desired trajectory (position,
// velocity, and acceleration)
double traj_store1[3][length_t]; // Link 1 trajectory
double traj_store2[3][length_t]; // Link 2 trajectory
// And we need 4 rows for the robot states (2 link robot,
// so 2 position and 2 velocity rows)
double y_store[4][length_t];

// We also need 2 rows for the computed torques:
double T_store[2][length_t];
// Save the tracking error over time:
double e_store[2][length_t];
double e_dot_store[2][length_t];

// We need storage for the observer states:
// It has 6 rows since we're also storing the
// estimated accelerations
double x_store[6][length_t];

// Need some storage for parameter estimates:
double P_store[3][length_t];

// Now the main program:
int main()
{

int i = 0; // One of the inevitable for loop counters
int j = 0; // Another inevitable for loop counter

// *** Experimental Section ***
// Need som variables to interface with the robot:
int fd; // the file descriptor of the open file (/dev/bobo0)
int result; // result of close operation
int read_result; // result of read operations
int write_result; // result of write operations

double position0, position1, velocity0, velocity1;
double torque0, torque1;
unsigned int temp0, temp1; // to hold the values cast to an integer
```

```
// allocate buffer space:
void *r_buf=malloc(NUM_R_BYTES); // Read buffer
unsigned char *r_ptr;

void *w_buf=malloc(NUM_W_BYTES); // Write buffer
unsigned char *w_ptr;
// *** End Experimental Section ***

// For passing arguments to the runge-kutta functions, I'm going to
// define some small one dimensional arrays that will hold the info
// to be passed in. C++ doesn't deal as nicely with matrices as
// MATLAB, // so I have to make some changes.
double y_traj[6]; // This contains the i-1 trajectory

// This contains the i-1 parameter estimates
double y_adapt[3];
// This contains the arguments that need to go
// in to compute the adaptaion law:
double u_adapt[15];

// Containts the i-1 position and accleration of
// both robot links
double y_robot[4];
// Contains all the arguments needed to integrate the robot
// dynamics
double u_robot[10];

// Contains the i-1 observer estimates
double y_observer[6];
// Conatins all the input arguments for
// observer dynamics
double u_observer[6];

// Define the robot parameters:
double theta[3]; // These are the parameters in the regression form
theta[0] = 0.5773;
theta[1] = 0.3364;
theta[2] = 0.0260;

// We want to inject some noise into the position measurements.
// noise is due to quanitization error. So in the variable noise,
// put the number of bits of the quantizer.
double noise = 10;
// Gives us quanitzation noise in degrees:
noise = 360/(pow(2,noise));
double noise_rad = (pi/180)*noise; // Noise in radians
// Seed the random number generator:
srand(5961500);
// This is a boolean to tell us whether or not
```

```
// to add noise to the system.
int add_noise = 0;

// Define controller parameters:
double kp = 36; // Position Error
double kd = 2*sqrt(kp); // Velocity Error

double Kd[2][2];
Kd[0][0] = kd;
Kd[0][1] = 0;
Kd[1][0] = 0;
Kd[1][1] = kd;

double Kp[2][2];
Kp[0][0] = kp;
Kp[0][1] = 0;
Kp[1][0] = 0;
Kp[1][1] = kp;


// Define observer parameters:

// Observer gains:
double L1 = 20;
double L2 = 500;

// Define adaptation law parameters:
double gamma11 = 0.1;
double gamma22 = 0.1;
double gamma33 = 0.1;
double delta = 0.05;

double psi11 = 1;
double psi22 = 1;

// We need the bounds on the convex hypercube for the parameter
// update law:
double a1 = 0.1;
double b1 = 10;
double a2 = 0.1;
double b2 = 10;
double a3 = -0.5;
double b3 = 10;

// We need some space to store the error:
double e1;
double e2;
double e1_dot;
double e2_dot;
```

```
// And the estimated error:
double e1_hat;
double e2_hat;
double e1_dot_hat;
double e2_dot_hat;

double qd_dot[2];
double qd_ddot[2];

// Our adaptation law here requires use of filtered error, which we
// define:
double E1[2];

// We also need the term q_ddot_star:
double q_ddot_star[2];

// In the Craig/Schwartz method we don't estimate the error
// but we estimate the position, velocity, and acceleration. So
// we need storage for that:
double q1_hat,q2_hat;
double q1_dot_hat,q2_dot_hat;
double q1_ddot_hat,q2_ddot_hat;


// I use the following variables as elements in matrices during the
// simulation:
double a,b,c,d;
double e,f,g,h,i1,j1;
double a_hat,b_hat,c_hat,d_hat;
double e_hat,f_hat,g_hat,h_hat;

// We need to store the torques:
double T1,T2;

// And we need some storage for our noise variables:
double n1,n2;

// We define our observer inputs here:
double v1,v2;

// We define a variable to store the current parameter estimates
double P_hat[3];

/* *** For writing data to the file: */
const char out_file[] = {"data/data.dat"};
const char out_file2[] = {"data/params.m"};

// *** Experimental Section ***
```

```
// We set the desired started position here: (For the PD
// controller)
for (i = 0; i < length_t; i++) {
      u_link1[i] = 0;
      u_link2[i] = 0;

}

// Open the bobo0 device:
fd = open("/dev/bobo0", O_RDWR);

if (fd < 0) { // there was an error
        return -1;
}

double kp0_pd = 8.24;
double kd0_pd = 0.98;
double ki0_pd = 25;
double kp1_pd = 5.24;
double kd1_pd = 0.48;
double ki1_pd = 1;

// We want to have a PD controller move the robot
// to the desired start position. Here is the code for that:
cout << "Moving robot to start position!\n";

// Get the starting positions of the robot:
// Read the data from the robot:
read_result = read(fd, r_buf, NUM_R_BYTES);

// A pointer to the beginning of the buffer
r_ptr = (unsigned char*)r_buf;

// We have our data, lets extract it:
position0 = *(r_ptr++);
position0 = position0 + (*(r_ptr++) << 8);
position1 = *(r_ptr++);
position1 = position1 + (*(r_ptr++) << 8);

velocity0 = *(r_ptr++);
velocity0 = velocity0 + (*(r_ptr++) << 8);
velocity1 = *(r_ptr++);
velocity1 = velocity1 + (*(r_ptr++) << 8);

// Now we convert position to radians
position0 = (position0 - 2048)*pi/2048.0;
position1 = -(position1 - 2048)*pi/2048.0;

// Initialize the trajectory:
```

```
traj_store1[0][0] = 0;
traj_store1[1][0] = 0;
traj_store1[2][0] = 0;

traj_store2[0][0] = 0;
traj_store2[1][0] = 0;
traj_store2[2][0] = 0;

// Initial robot state
y_store[0][0] = position0;
y_store[1][0] = position1;
y_store[2][0] = 0;
y_store[3][0] = 0;

T_store[0][0] = 0;
T_store[1][0] = 0;

e_store[0][0] = 0;
e_store[1][0] = 0;

e_dot_store[0][0] = 0;
e_dot_store[1][0] = 0;

// This is the PD controller loop:
for (i = 1; i < int(10/Ts); i++) {

    // Run the Runge-Kutta method for this time step:
    // We want to comput the trajectory here:
    // First, we fit what we need into a 1D array to pass to the
    // Runge-Kutta Function.
    y_traj[0] = traj_store1[0][i-1];
    y_traj[1] = traj_store1[1][i-1];
    y_traj[2] = traj_store1[2][i-1];

    y_traj[3] = traj_store2[0][i-1];
    y_traj[4] = traj_store2[1][i-1];
    y_traj[5] = traj_store2[2][i-1];

    // We now call the Runge-Kutta function for the trajectory and
    // pass this info into it:
    rk_traj(y_traj,u_link1[i],u_link2[i]);

    // With this function executed, y_traj will contain the
    // trajectory for the next time step. We then fit that into our
    // array:
    traj_store1[0][i] = y_traj[0];
    traj_store1[1][i] = y_traj[1];
    traj_store1[2][i] = y_traj[2];
```

```
traj_store2[0][i] = y_traj[3];
traj_store2[1][i] = y_traj[4];
traj_store2[2][i] = y_traj[5];


// traj_store is set up like:
// traj_store[0] = qd
// traj_store[1] = qd_dot
// traj_store[2] = qd_ddot

// For the control law, we need e = q_d - q and the derivatives
e1 = -(y_store[0][i-1] - traj_store1[0][i-1]);
e2 = -(y_store[1][i-1] - traj_store2[0][i-1]);
e1_dot = -(y_store[2][i-1] - traj_store1[1][i-1]);
e2_dot = -(y_store[3][i-1] - traj_store2[1][i-1]);

e_store[0][i] = e1;
e_store[1][i] = e2;
e_dot_store[0][i] = e1_dot;
e_dot_store[1][i] = e2_dot;

// *** Controller Section ***
// We compute the torques using 2 PID controllers:
T1 = kp0_pd*e1 + kd0_pd*e1_dot + ki0_pd*(e_store[0][i-1] +
e_store[0][i])*Ts;
T2 = kp1_pd*e2 + kd1_pd*e2_dot + ki1_pd*(e_store[1][i-1] +
e_store[1][i])*Ts;

T_store[0][i] = T1;
T_store[1][i] = T2;

// *** Experimental Section ***
// Now that the torques are ready, we will write them out to
// the device.
// Now we convert the torques to a range appropriate for
// the DA card:
torque0 = T_store[0][i];
torque1 = T_store[1][i];

temp0 = (unsigned int)((torque0/MOTOR0_CONST)*2048/5.0 + 2048);
temp1 = (unsigned int)((torque1/MOTOR1_CONST)*2048/5.0 + 2048);

// Now make sure the torques are within range:
if (temp0 > 0xFFF) {
    temp0 = 0xFFF;
} else if (temp0 < 0) {
    temp0 = 0;
}
if (temp1 > 0xFFF) {
```

```
    temp1 = 0xFFF;
} else if (temp1 < 0) {
    temp1 = 0;
}

// Send torque data out:
// A pointer to the beginning of the buffer:
w_ptr = (unsigned char*)w_buf;
// the low byte of torque 0:
*(w_ptr++) = (unsigned char)(temp0 & 0xFF);
// the high byte of torque 0:
*(w_ptr++) = (unsigned char)((temp0 >> 8) & 0xF);
// the low byte of torque 1:
*(w_ptr++) = (unsigned char)(temp1 & 0xFF);
// the high byte of torque 1:
*(w_ptr++) = (unsigned char)((temp1 >> 8) & 0xF);

// Call the write method. This, in term, calls the device
// driver's write method.
write_result = write(fd, w_buf, NUM_W_BYTES);

// The next step is to get the data from the robot:
// Read the data from the robot:
read_result = read(fd, r_buf, NUM_R_BYTES);

// A pointer to the beginning of the buffer
r_ptr = (unsigned char*)r_buf;
// We have our data, lets extract it:
position0 = *(r_ptr++);
position0 = position0 + (*(r_ptr++) << 8);
position1 = *(r_ptr++);
position1 = position1 + (*(r_ptr++) << 8);

velocity0 = *(r_ptr++);
velocity0 = velocity0 + (*(r_ptr++) << 8);
velocity1 = *(r_ptr++);
velocity1 = velocity1 + (*(r_ptr++) << 8);

// Now we convert all this to radians and radians/sec
position0 = (position0 - 2048)*pi/2048.0;
position1 = -(position1 - 2048)*pi/2048.0;

velocity0 = (velocity0*10/4096.0 - 5.0)*KTACH0;
velocity1 = (velocity1*10/4096.0 - 5.0)*KTACH1;

// Now we store these values in the appropriate array:
y_store[0][i] = position0;
y_store[1][i] = position1;
y_store[2][i] = velocity0;
```

```
    y_store[3][i] = velocity1;

} // end of PD for loop

cout << "Robot is in start position. Beginning Experiment!\n";

// *** End Experimental Section ***

// We initialize the input signal here
for (i = 0; i < length_t; i++) {
    if (i % int(20/Ts) < 10/Ts) {
        u_link1[i] = -1;
        u_link2[i] = -1;
    } else {
        u_link1[i] = 1;
        u_link2[i] = 1;
    } // This will give a square wave of period 20 s

// For the experiments, we use this trajectory
    // Loopy Path: (Identifiable)
    u_link1[i] = 1.0*(0 + cos(2*1.257*(i*Ts)) -
    cos(4*1.257*(i*Ts)));
    u_link2[i] = -0.4 + 1.0*(pi/2 - 1.5 + sin(1.257*(i*Ts)) + sin(2
    *1.257*(i*Ts)));

}

// *** Experimental Section ***
// Get the starting positions of the robot:
// Read the data from the robot:
read_result = read(fd, r_buf, NUM_R_BYTES);

// A pointer to the beginning of the buffer
r_ptr = (unsigned char*)r_buf;

// We have our data, lets extract it:
position0 = *(r_ptr++);
position0 = position0 + (*(r_ptr++) << 8);
position1 = *(r_ptr++);
position1 = position1 + (*(r_ptr++) << 8);

velocity0 = *(r_ptr++);
velocity0 = velocity0 + (*(r_ptr++) << 8);
velocity1 = *(r_ptr++);
velocity1 = velocity1 + (*(r_ptr++) << 8);

// Now we convert position to radians
position0 = (position0 - 2048)*pi/2048.0;
position1 = -(position1 - 2048)*pi/2048.0;
```

```
// *** End Experimental Section ***


// Initialize the trajectory:
traj_store1[0][0] = 0;
traj_store1[1][0] = 0;
traj_store1[2][0] = 0;

traj_store2[0][0] = 0;
traj_store2[1][0] = 0;
traj_store2[2][0] = 0;


y_store[0][0] = position0;
y_store[1][0] = position1;
y_store[2][0] = 0;
y_store[3][0] = 0;

T_store[0][0] = 0;
T_store[1][0] = 0;

e_store[0][0] = 0;
e_store[1][0] = 0;

e_dot_store[0][0] = 0;
e_dot_store[1][0] = 0;


x_store[0][0] = y_store[0][0];
x_store[1][0] = y_store[1][0];
x_store[2][0] = 0;
x_store[3][0] = 0;
x_store[4][0] = 0;
x_store[5][0] = 0;

// Initialize the parameter estimates:
P_store[0][0] = 0.2;
P_store[1][0] = 0.2;
P_store[2][0] = 0.01;


// Now we simulate the system:
for (i = 1; i < length_t; i++) {

    // Run the Runge-Kutta method for this time step:
    // We want to comput the trajectory here:
    // First, we fit what we need into a 1D array to pass to the
    // Runge-Kutta Function.
    y_traj[0] = traj_store1[0][i-1];
```

```
y_traj[1] = traj_store1[1][i-1];
y_traj[2] = traj_store1[2][i-1];

y_traj[3] = traj_store2[0][i-1];
y_traj[4] = traj_store2[1][i-1];
y_traj[5] = traj_store2[2][i-1];

// We now call the Runge-Kutta function for the trajectory and
// pass this info into it:
rk_traj(y_traj,u_link1[i],u_link2[i]);

// With this function executed, y_traj will contain the
// trajectory for the next time step. We then fit that into our
// array:
traj_store1[0][i] = y_traj[0];
traj_store1[1][i] = y_traj[1];
traj_store1[2][i] = y_traj[2];

traj_store2[0][i] = y_traj[3];
traj_store2[1][i] = y_traj[4];
traj_store2[2][i] = y_traj[5];


// traj_store is set up like:
// traj_store[0] = qd
// traj_store[1] = qd_dot
// traj_store[2] = qd_ddot


// For the control law, we need e = q_d - q and the derivatives
e1 = -(y_store[0][i-1] - traj_store1[0][i-1]);
e2 = -(y_store[1][i-1] - traj_store2[0][i-1]);
e1_dot = -(y_store[2][i-1] - traj_store1[1][i-1]);
e2_dot = -(y_store[3][i-1] - traj_store2[1][i-1]);

e_store[0][i] = e1;
e_store[1][i] = e2;
e_dot_store[0][i] = e1_dot;
e_dot_store[1][i] = e2_dot;


// We're going to use the estimated error for the controller
// instead of the actual error, since in practice we don't have
// access to the velocity error (we're not measuring velocity)
// but we do have the estimated velocity error.
e1_hat = -(x_store[0][i-1] - traj_store1[0][i-1]);
e2_hat = -(x_store[1][i-1] - traj_store2[0][i-1]);
e1_dot_hat = -(x_store[2][i-1] - traj_store1[1][i-1]);
e2_dot_hat = -(x_store[3][i-1] - traj_store2[1][i-1]);
```

```
// *** The parameter adaptation is here ***

// We need Y(q,q_dot,q_ddot)' = [a b c;d e f] (or rather, the
// estimates of those values) for Craig's adaptation law:
q1_hat = x_store[0][i-1];
q2_hat = x_store[1][i-1];
q1_dot_hat = x_store[2][i-1];
q2_dot_hat = x_store[3][i-1];
q1_ddot_hat = x_store[4][i-1];
q2_ddot_hat = x_store[5][i-1];

a = q1_ddot_hat;
b = 0;
c = 0;
d = q2_ddot_hat;
e = q2_ddot_hat*cos(q1_hat - q2_hat) + q2_dot_hat*q2_dot_hat*
sin(q1_hat - q2_hat);
f = q1_ddot_hat*cos(q1_hat - q2_hat) - q1_dot_hat*q1_dot_hat*
sin(q1_hat - q2_hat);


// Determine M_hat for the adaptation law:
a_hat = P_store[0][i-1];
b_hat = P_store[2][i-1]*cos(q1_hat - q2_hat);
c_hat = P_store[2][i-1]*cos(q1_hat - q2_hat);
d_hat = P_store[1][i-1];


// Now, define the elements of M_hat_inv as [e f;g h];
g = d_hat/(a_hat*d_hat - b_hat*c_hat);
h = -b_hat/(a_hat*d_hat - b_hat*c_hat);
i1 = -c_hat/(a_hat*d_hat - b_hat*c_hat);
j1 = a_hat/(a_hat*d_hat - b_hat*c_hat);

// Now we need the filtered error
E1[0] = e1_dot_hat + psi11*e1_hat;
E1[1] = e2_dot_hat + psi11*e2_hat;


// Now, to evaluate the dynamics we need to pass everything
// over to the Runge-Kutta function, which will call the
// cf_adapt function to calculate the right hand side of the
// dynamic equation.
u_adapt[0] = a;
u_adapt[1] = b;
u_adapt[2] = c;
u_adapt[3] = d;
```

```
u_adapt[4] = e;
u_adapt[5] = f;
u_adapt[6] = g;
u_adapt[7] = h;
u_adapt[8] = i1;
u_adapt[9] = j1;
u_adapt[10] = gamma11;
u_adapt[11] = gamma22;
u_adapt[12] = gamma33;
u_adapt[13] = E1[0];
u_adapt[14] = E1[1];

// Now we fill up the y array that will contain the old P
// values
y_adapt[0] = P_store[0][i-1];
y_adapt[1] = P_store[1][i-1];
y_adapt[2] = P_store[2][i-1];

// We now call the Runge-Kutta function to integrate the
// adaptation dynamics for this time step. (Note that this
// function actually modifies the contents of the array y_adapt
// to contain the result of the integration for the next
// time-step, since it is passed by reference.)
rk_adapt(y_adapt,u_adapt);

// Store the updated parameters here:
P_store[0][i] = y_adapt[0];
P_store[1][i] = y_adapt[1];
P_store[2][i] = y_adapt[2];

// We need to implement parameter resetting as given in Craig's
// paper in eqn 24:

if (P_store[0][i] <= a1 - delta) {
    P_store[0][i] = a1;
} else if (P_store[0][i] >= b1 + delta) {
    P_store[0][i] = b1;
}

if (P_store[1][i] <= a2 - delta) {
    P_store[1][i] = a2;
} else if (P_store[1][i] >= b2 + delta) {
    P_store[1][i] = b2;
}

if (P_store[2][i] <= a3 - delta) {
    P_store[2][i] = a3;
} else if (P_store[2][i] >= b3 + delta) {
    P_store[2][i] = b3;
```

```
}


P_hat[0] = y_adapt[0];
P_hat[1] = y_adapt[1];
P_hat[2] = y_adapt[2];

// *** Controller Section ***

// With the trajectory, we need to compute the torques to give
// the robot:

// Determine M_hat:
a_hat = P_hat[0];
b_hat = P_hat[2]*cos(q1_hat - q2_hat);
c_hat = P_hat[2]*cos(q1_hat - q2_hat);
d_hat = P_hat[1];

// For Craig's paper, C(q,q_dot):
e_hat = 0;
f_hat = P_hat[2]*q2_dot_hat*sin(q1_hat - q2_hat);
g_hat = -P_hat[2]*q1_dot_hat*sin(q1_hat - q2_hat);
h_hat = 0;

// Now Craig's control law:
q_ddot_star[0] = traj_store1[2][i] + Kd[0][0]*e1_dot_hat +
Kp[0][0]*e1_hat;
q_ddot_star[1] = traj_store2[2][i] + Kd[1][1]*e2_dot_hat +
Kp[1][1]*e2_hat;

T1 = a_hat*q_ddot_star[0] + b_hat*q_ddot_star[1] + e_hat*
q1_dot_hat + f_hat*q2_dot_hat;
T2 = c_hat*q_ddot_star[0] + d_hat*q_ddot_star[1] + g_hat*
q1_dot_hat + h_hat*q2_dot_hat;



T_store[0][i] = T1;
T_store[1][i] = T2;

// *** Experimental Section ***
// Now that the torques are ready, we will write them out to
// the device.
// Now we convert the torques to a range appropriate for
// the DA card:
torque0 = T_store[0][i];
torque1 = T_store[1][i];

temp0 = (unsigned int)((torque0/MOTOR0_CONST)*2048/5.0 + 2048);
```

```
    temp1 = (unsigned int)((torque1/MOTOR1_CONST)*2048/5.0 + 2048);

    // Now make sure the torques are within range:
    if (temp0 > 0xFFF) {
        temp0 = 0xFFF;
    } else if (temp0 < 0) {
        temp0 = 0;
    }
    if (temp1 > 0xFFF) {
        temp1 = 0xFFF;
    } else if (temp1 < 0) {
        temp1 = 0;
    }

    // Send torque data out:
    // A pointer to the beginning of the buffer:
    w_ptr = (unsigned char*)w_buf;
    // the low byte of torque 0:
    *(w_ptr++) = (unsigned char)(temp0 & 0xFF);
    // the high byte of torque 0:
    *(w_ptr++) = (unsigned char)((temp0 >> 8) & 0xF);
    // the low byte of torque 1:
    *(w_ptr++) = (unsigned char)(temp1 & 0xFF);
    // the high byte of torque 1:
    *(w_ptr++) = (unsigned char)((temp1 >> 8) & 0xF);

    // Call the write method. This, in term, calls the device
    // driver's write method.
    write_result = write(fd, w_buf, NUM_W_BYTES);
    // *** End Experimental Section


    // *** Robot Dynamics ***

    // We now have to use these to put into our robot dynamics as
    // follows:
    // These terms are for the mass matrix:
    // y_store contains position and velocity states as follows:
    // y_store(0,:) = q1
    // y_store(1,:) = q2
    // y_store(2,:) = q1_dot
    // y_store(3,:) = q2_dot

/*
    // *** Simulation Section ***
    // M, the Mass Matrix:
    a = theta[0];
    b = theta[2]*cos(y_store[0][i-1] - y_store[1][i-1]);
    c = theta[2]*cos(y_store[0][i-1] - y_store[1][i-1]);
```

```
    d = theta[1];


    // These terms are for the corriolis matrix:
    e = 0;
    f = theta[2]*y_store[3][i-1]*sin(y_store[0][i-1] -
    y_store[1][i-1]);
    g = -theta[2]*y_store[2][i-1]*sin(y_store[0][i-1] -
    y_store[1][i-1]);
    h = 0;


    // Now, to evaluate the dynamic equation at this point for the
    // robot, we need to pass a,b,c,d,e,f,g,h, and the torques to
    // the Runge-Kutta function, which in turn passes these
    // parameters to cf_robot, which computes the robot's dynamics.
    u_robot[0] = a;
    u_robot[1] = b;
    u_robot[2] = c;
    u_robot[3] = d;
    u_robot[4] = e;
    u_robot[5] = f;
    u_robot[6] = g;
    u_robot[7] = h;
    u_robot[8] = T1s;
    u_robot[9] = T2s;

    // Now we fill up the y array to contain the previous position
    // and velocity values:
    y_robot[0] = y_store[0][i-1];
    y_robot[1] = y_store[1][i-1];
    y_robot[2] = y_store[2][i-1];
    y_robot[3] = y_store[3][i-1];


    // We now call the Runge-Kutta function to integrate the robot
    // dynamics for this time step.
    rk_robot(y_robot,u_robot);

    y_store[0][i] = y_robot[0];
    y_store[1][i] = y_robot[1];
    y_store[2][i] = y_robot[2];
    y_store[3][i] = y_robot[3];
    // *** End Simulation Section ***
*/

    // *** Experimental Section ***
    // The next step is to get the data from the robot:
    // Read the data from the robot:
```

```
read_result = read(fd, r_buf, NUM_R_BYTES);

// A pointer to the beginning of the buffer
r_ptr = (unsigned char*)r_buf;
// We have our data, lets extract it:
position0 = *(r_ptr++);
position0 = position0 + (*(r_ptr++) << 8);
position1 = *(r_ptr++);
position1 = position1 + (*(r_ptr++) << 8);

velocity0 = *(r_ptr++);
velocity0 = velocity0 + (*(r_ptr++) << 8);
velocity1 = *(r_ptr++);
velocity1 = velocity1 + (*(r_ptr++) << 8);

// Now we convert all this to radians and radians/sec
position0 = (position0 - 2048)*pi/2048.0;
position1 = -(position1 - 2048)*pi/2048.0;

velocity0 = (velocity0*10/4096.0 - 5.0)*KTACH0;
velocity1 = (velocity1*10/4096.0 - 5.0)*KTACH1;

// Now we store these values in the appropriate array:
y_store[0][i] = position0;
y_store[1][i] = position1;
y_store[2][i] = velocity0;
y_store[3][i] = velocity1;

// We occasionally have outliers that mess with the control,
// here we remove the most significant of them:
if (fabs(y_store[0][i] - y_store[0][i-1]) > MAX_OUTLIER) { //
then it must be an outlier
    y_store[0][i] = y_store[0][i-1];
}

if (fabs(y_store[1][i] - y_store[1][i-1]) > MAX_OUTLIER) { //
then it must be an outlier
    y_store[1][i] = y_store[1][i-1];
}


// *** End Experimental Section

// We need to contaminate the position measurements
// with uniformly distributed random noise between -noise/2 and
// noise/2 degrees.

if (add_noise == 1) { // then we want noise
    n1 = noise_rad*(1.0*rand()/RAND_MAX) - noise_rad/2;
```

```
    n2 = noise_rad*(1.0*rand()/RAND_MAX) - noise_rad/2;

    y_store[0][i] = y_store[0][i] + n1;
    y_store[1][i] = y_store[1][i] + n2;
}


// *** Observer Section ***

// Now we're using Schwartz's observer,  This observer
// estimates the position, velocity,
// and we're going to pass back the acceleration as well.

// Here, we build the observer dynamics.
// The vector x_store looks like:
// x_store(0,:) = q1_hat
// x_store(1,:) = q2_hat
// x_store(2,:) = q1_dot_hat
// x_store(3,:) = q2_dot_hat
// x_store(4,:) = q1_ddot_hat
// x_store(5,:) = q2_ddot_hat

// Here we need to define v1 and v2, the closed loop inputs to
the linearized system.
// v = q_ddot_com - K*x_hat
v1 = q_ddot_star[0];
v2 = q_ddot_star[1];

// Now, to evaluate the dynamics we need to pass everything
// over to the Runge-Kutta function, which will call the
// cf_observer function to calculate the right hand side of the
// dynamic equation.
u_observer[0] = y_store[0][i];
u_observer[1] = y_store[1][i];
u_observer[2] = L1;
u_observer[3] = L2;
u_observer[4] = v1;
u_observer[5] = v2;

// And we send last time step's estimates back in for
// integration:
y_observer[0] = x_store[0][i-1];
y_observer[1] = x_store[1][i-1];
y_observer[2] = x_store[2][i-1];
y_observer[3] = x_store[3][i-1];
y_observer[4] = 0;
y_observer[5] = 0;

// We now call the Runge-Kutta function to integrate the
```

```
    // observer dynamics for this time step.
    rk_observer(y_observer,u_observer);

    x_store[0][i] = y_observer[0];
    x_store[1][i] = y_observer[1];
    x_store[2][i] = y_observer[2];
    x_store[3][i] = y_observer[3];
    x_store[4][i] = y_observer[4];
    x_store[5][i] = y_observer[5];


} // end of simulation for loop

// *** Experimental Section ***
// close bobo
result = close(fd);

// free the allocated memory:
free(r_buf);
free(w_buf);
// *** End Experimental Section ***

cout << "Done Simulation! Writing Data to Disk..." << endl;

// Write the data to the file:
ofstream fout2;
fout2.open(out_file2);

// Some variables we like:
fout2 << "kp = " << kp << ";" << endl;
fout2 << "kd = " << kd << ";" << endl;
fout2 << "L1 = " << L1 << ";" << endl;
fout2 << "L2 = " << L2 << ";" << endl;
fout2 << "gamma11 = " << gamma11 << ";" << endl;
fout2 << "gamma22 = " << gamma22 << ";" << endl;
fout2 << "gamma33 = " << gamma22 << ";" << endl;

fout2 << "theta1 = " << theta[0] << ";" << endl;
fout2 << "theta2 = " << theta[1] << ";" << endl;
fout2 << "theta3 = " << theta[2] << ";" << endl;

// Close this out file:
fout2.close();

ofstream fout;
fout.open(out_file);

// Store the data we want:
for (i = 0; i < length_t; i++) {
```

```
        // This is a time vector
        fout << i*Ts << "\t";

        // The trajectory desired:
        fout << traj_store1[0][i] << "\t";
        fout << traj_store1[1][i] << "\t";
        fout << traj_store1[2][i] << "\t";

        fout << traj_store2[0][i] << "\t";
        fout << traj_store2[1][i] << "\t";
        fout << traj_store2[2][i] << "\t";

        // And the actual robot trajectory:

        fout << y_store[0][i] << "\t";
        fout << y_store[1][i] << "\t";
        fout << y_store[2][i] << "\t";
        fout << y_store[3][i] << "\t";

        // Now the torques
        fout << T_store[0][i] << "\t";
        fout << T_store[1][i] << "\t";

        // And the parameter adaptation:
        fout << P_store[0][i] << "\t";
        fout << P_store[1][i] << "\t";
        fout << P_store[2][i] << "\t";

        // And the observed states
        fout << x_store[0][i] << "\t";
        fout << x_store[1][i] << "\t";
        fout << x_store[2][i] << "\t";
        fout << x_store[3][i] << "\t";

        // Now the position error:
        fout << e_store[0][i] << "\t";
        fout << e_store[1][i] << "\t";

        // The velocity error:
        fout << e_dot_store[0][i] << "\t";
        fout << e_dot_store[1][i] << "\t";

        fout << endl;

}

// Close our output file:
fout.close();
```

```
return 0;

} // end of int main()



// This procedure computes the derivative vector
//   f(y(t)) = F(t,y(t)) + b(t,y(t))*u(t).  This
//   procedure is called four times by the procedure
//   runge_kutta.

// So, really we're specifying the right hand side
// of our dynamic system here. i.e. Ax + Bu for
// a linear system. This will return x_dot.
// The runge-kutta function then integrates x_dot
// numerically to get x.

void cf_traj(double y[], double f[], double u1, double u2)
{

// These are the parameters for our 2nd order prefilter
double wn = 2;
double zeta = 1;

// First link trajectory:
f[0] = y[1];
f[1] = -(pow(wn,2))*y[0] - (2*zeta*wn)*y[1] + pow(wn,2)*u1;
f[2] = f[1]; // In here we return the acceleration signal

// Second link trajectory:
f[3] = y[4];
f[4] = -(pow(wn,2))*y[3] - (2*zeta*wn)*y[4] + pow(wn,2)*u2;
f[5] = f[4]; // In here we return the acceleration signal


} // end of function cf_traj


/*% This function performs the Runge-Kutta calculation.
% y is the function to be integrated
% u is the input function. It's passed right through to compute_f,
% which determines the right hand side of the dynamic system
% dt is the step size (sample period)
% n is the number of states in the system. */
void rk_traj(double y[], double u1, double u2)
{
```

```
/*% This is the numerical solution to the nonlinear dynamics
%   of the robot manipulator.  This procedure calls another
%   procedure, which is compute_f. Compute_f computes the
%   derivatives of the states in vector y. These derivatives
%   are stored in vector f(y(t)).  The state of the
%   manipulator is calulated based on the following recursion;

%      y(t + dt) = y(t) + 1/6(k0 + 2k1 + 2k2 + k3)

%   where
%         k0 = f(y(t))*dt
%         k1 = f(y(t) + 1/2k0)*dt
%         k2 = f(y(t) + 1/2k1)*dt
%         k3 = f(y(t) + k2)*dt

%   From the above formula it can be seen that to calculate the
%   new state at the next time step, the function f(y(t)) has to be
%   calculated four times. *)*/

double f[6];
double k[4][6]; // k0,k1,k2,k3 for all states
double y1[6]; // Present system state

double dt = Ts;
double n = 6; // The number of states to integrate
int i = 0; // A counter
int j = 0; // A counter

// And we store our acceleration in here:
double accel1;
double accel2;

for (i = 0; i < 4; i++) {

    // Compute the derivative vector f(j)
    cf_traj(y,f,u1,u2);

    // We want to return the acceleration.
    // Since we don't want to integrate this, we'll grab it on the
    // first time out, and make sure it doesn't get integrated:
    if (i == 0) {
        accel1 = f[2];
        accel2 = f[5];
    }

    // If this is the first time through the loop we will
    //     store the present state of the manipulator
    //    and compute k0 and y(t) + (1/2)k0 which will be used
    //    as the argument for f(y(t) +(1/2)k0)
```

```
      if (i == 0) {
          for (j = 0; j < n; j++) {
                // store present robot state
                y1[j]  = y[j];
                // compute k0 = f(y(t))*dt
                k[i][j] = f[j]*dt;
                // compute y(t) + (1/2)k0
                y[j]  = y1[j] + 0.5*k[i][j];
          }
      }

      if (i == 1) {
          for (j = 0; j < n; j++) {
                // compute k1 = f(x(t))*dt
                k[i][j] = f[j]*dt;
                // compute y(t) + (1/2)k1
                y[j]   = y1[j] + 0.5*k[i][j];
          }
      }

      if (i == 2) {
          for (j = 0; j < n; j++) {
                // compute k2 = f(x(t))*dt
                k[i][j] = f[j]*dt;
                // compute y(t) + k2
                y[j]   = y1[j] + k[i][j];
          }
      }

      if (i == 3) {
          for (j = 0; j < n; j++) {
                //  compute k3 = f(x(t))*dt
                k[i][j] = f[j]*dt;

                // compute updated state of the robot

                y[j] = y1[j] + (1.0/6.0)*(k[0][j] + 2.0*k[1][j] +
                2.0*k[2][j] + k[3][j]);

                // The robot state has been updated

          }          // end of loop for j=1,4
      }              // end of loop for the if i=4 decision
}                    // end of the for loop i=1,4

// Now we throw accel into the returned variable. We need this for
// our acceleration signal.
y[2] = accel1; // y[0] and y[1] contain position and velocity
```

```
y[5] = accel2;

} // End of rk_traj function



// This procedure computes the derivative vector
//   f(y(t)) = F(t,y(t)) + b(t,y(t))*u(t).
// This function currently determines the right hand side of the
// dynamics of the adaptation law.

void cf_adapt(double y[], double f[], double u[])
{

double a1 = u[0];
double b1 = u[1];
double c1 = u[2];
double d1 = u[3];
double e1 = u[4];
double f1 = u[5];
double g1 = u[6];
double h1 = u[7];
double i1 = u[8];
double j1 = u[9];
double gamma11 = u[10];
double gamma22 = u[11];
double gamma33 = u[12];
double E11 = u[13];
double E12 = u[14];

f[0] = E11*gamma11*(a1*g1 + b1*i1) + E12*gamma11*(a1*h1 + b1*j1);
f[1] = E11*gamma22*(c1*g1 + d1*i1) + E12*gamma22*(c1*h1 + d1*j1);
f[2] = E11*gamma33*(e1*g1 + f1*i1) + E12*gamma33*(e1*h1 + f1*j1);

} // End of function cf_adapt



/* Perform the Runge-Kutta calculation to integrate the adaptation
   law dynamics: */
void rk_adapt(double y[], double u[])
{

const int num_states = 3;
// This has the parameter updates as well as phi, and phi doesn't
// get integrated.
double f[num_states];
double k[4][num_states]; // k0,k1,k2,k3 for all states
double y1[num_states]; // Present system state
```

```
double dt = Ts;
double n = 3; // The number of states to integrate
int i = 0; // A counter
int j = 0; // A counter

for (i = 0; i < 4; i++) {

    // Compute the derivative vector f(j)
    cf_adapt(y,f,u);

    // If this is the first time through the loop we will
    //     store the present state of the manipulator
    //    and compute k0 and y(t) + (1/2)k0 which will be used
    //    as the argument for f(y(t) +(1/2)k0)

      if (i == 0) {
          for (j = 0; j < n; j++) {
                y1[j]  = y[j];
                k[i][j] = f[j]*dt;
                y[j]  = y1[j] + 0.5*k[i][j];
          }
      }

      if (i == 1) {
          for (j = 0; j < n; j++) {
                k[i][j] = f[j]*dt;
                y[j]   = y1[j] + 0.5*k[i][j];
          }
      }

      if (i == 2) {
          for (j = 0; j < n; j++) {
                k[i][j] = f[j]*dt;
                y[j]   = y1[j] + k[i][j];
          }
      }

      if (i == 3) {
          for (j = 0; j < n; j++) {
                k[i][j] = f[j]*dt;

                // compute updated state of the robot

                y[j] = y1[j] + (1.0/6.0)*(k[0][j] + 2.0*k[1][j] +
                2.0*k[2][j] + k[3][j]);

                // The robot state has been updated
```

```
          }          // end of loop for j=1,4
      }              // end of loop for the if i=4 decision
}                    // end of the for loop i=1,4

} // End of rk_adapt function



// This procedure computes the derivative vector
//   f(y(t)) = F(t,y(t)) + b(t,y(t))*u(t).

// a1,b1,c1,d1 describe the Mass matrix, e1, f1, g1, and h1
// describe the corriolis matrix, T1 and T2 are the input torques.

// This function currently determines the dynamics of the robot.
// It is only used in simulation to give simulated data. With the
// experiments, this step is replaced by real calls to the actual
// robot.

void cf_robot(double y[], double f[], double u[])
{

// Now, u is a vector containing a1,b1,c1,d1,e1,f1,T1,T2
double a1 = u[0];
double b1 = u[1];
double c1 = u[2];
double d1 = u[3];
double e1 = u[4];
double f1 = u[5];
double g1 = u[6];
double h1 = u[7];
double T1 = u[8];
double T2 = u[9];

f[0] = y[2]; // y[2] is q1_dot
f[1] = y[3]; // y[3] is q2_dot
f[2] = (d1*(T1 - e1*y[2] - f1*y[3]) - b1*(T2 - g1*y[2] - h1*y[3]))/
(a1*d1 - b1*c1);
f[3] = (-c1*(T1 - e1*y[2] - f1*y[3]) + a1*(T2 - g1*y[2] - h1*y[3]))
/(a1*d1 - b1*c1);

} // end of cf_robot


/* Perform the Runge-Kutta calculation to integrate the simulated
   robot dynamics: */
void rk_robot(double y[], double u[])
{
```

```
// Number of states: This could include ones that don't get
// integrated and do get passed back.
const int num_states = 4;
// This has the parameter updates as well as phi, and phi doesn't
// get integrated.
double f[num_states];
double k[4][num_states]; // k0,k1,k2,k3 for all states
double y1[num_states]; // Present system state

double dt = Ts;
double n = 4; // The number of states to integrate
int i = 0; // A counter
int j = 0; // A counter

for (i = 0; i < 4; i++) {

    // Compute the derivative vector f(j)
    cf_robot(y,f,u);

    // If this is the first time through the loop we will
    //     store the present state of the manipulator
    //    and compute k0 and y(t) + (1/2)k0 which will be used
    //    as the argument for f(y(t) +(1/2)k0)

      if (i == 0) {
          for (j = 0; j < n; j++) {
                y1[j]  = y[j];
                k[i][j] = f[j]*dt;
                y[j]  = y1[j] + 0.5*k[i][j];
          }
      }

      if (i == 1) {
          for (j = 0; j < n; j++) {
                k[i][j] = f[j]*dt;
                y[j]   = y1[j] + 0.5*k[i][j];
          }
      }

      if (i == 2) {
          for (j = 0; j < n; j++) {
                k[i][j] = f[j]*dt;
                y[j]   = y1[j] + k[i][j];
          }
      }

      if (i == 3) {
          for (j = 0; j < n; j++) {
                k[i][j] = f[j]*dt;
```

```
                // compute updated state of the robot

                y[j] = y1[j] + (1.0/6.0)*(k[0][j] + 2.0*k[1][j] +
                2.0*k[2][j] + k[3][j]);

                // The robot state has been updated

            }           // end of loop for j=1,4
        }               // end of loop for the if i=4 decision
}                       // end of the for loop i=1,4

} // End of rk_robot function



// This procedure computes the derivative vector
//   f(y(t)) = F(t,y(t)) + b(t,y(t))*u(t).
// This function currently determines the dynamics of the observer.

void cf_observer(double y[], double f[], double u[])
{

double q1 = u[0];
double q2 = u[1];
double L1 = u[2];
double L2 = u[3];
double v1 = u[4]; // These are the control inputs to the robot
double v2 = u[5];

// f is of length 6 because we need to return the accelerations in
// this vector as well, but we don't want to integrate them.
// y(0) = q1_hat
// y(1) = q2_hat
// y(2) = q1_hat_dot
// y(3) = q2_hat_dot
// y(4) = q1_hat_ddot
// y(5) = q2_hat_ddot - We don't actually use these accelerations
// in the observer calculations, we just need to keep them
f[0] = y[2] + L1*(q1 - y[0]);
f[1] = y[3] + L1*(q2 - y[1]);
f[2] = v1 + L2*(q1 - y[0]);
f[3] = v2 + L2*(q2 - y[1]);

// Also want to pass through the accelerations, un-integrated:
f[4] = f[2];
f[5] = f[3];

} // end of cf_observer
```

```
/* Perform the Runge-Kutta calculation to integrate the observer
   dynamics: */
void rk_observer(double y[], double u[])
{

const int num_states = 6;
// This has the parameter updates as well as
// q_ddot_hat, which doesn't get integrated.
double f[num_states];
double k[4][num_states]; // k0,k1,k2,k3 for all states
double y1[num_states]; // Present system state

double dt = Ts;
double n = 4; // The number of states to integrate
int i = 0; // A counter
int j = 0; // A counter

// And we store our acceleration estimate in here:
double q1_ddot_hat;
double q2_ddot_hat;

for (i = 0; i < 4; i++) {

    // Compute the derivative vector f(j)
    cf_observer(y,f,u);

    // We want to return the unintegrated phi for later.
    // Since we don't want to integrate this, we'll grab it on the
    // first time out, and make sure it doesn't get integrated:
    if (i == 0) {
        q1_ddot_hat = f[4];
        q2_ddot_hat = f[5];
    }

    // If this is the first time through the loop we will
    //     store the present state of the manipulator
    //    and compute k0 and y(t) + (1/2)k0 which will be used
    //    as the argument for f(y(t) +(1/2)k0)

      if (i == 0) {
          for (j = 0; j < n; j++) {
                y1[j]  = y[j];
                k[i][j] = f[j]*dt;
                y[j]  = y1[j] + 0.5*k[i][j];
          }
      }
```

```
      if (i == 1) {
          for (j = 0; j < n; j++) {
                k[i][j] = f[j]*dt;
                y[j]    = y1[j] + 0.5*k[i][j];
          }
      }

      if (i == 2) {
          for (j = 0; j < n; j++) {
                k[i][j] = f[j]*dt;
                y[j]    = y1[j] + k[i][j];
          }
      }

      if (i == 3) {
          for (j = 0; j < n; j++) {
                k[i][j] = f[j]*dt;

                // compute updated state of the robot

                y[j] = y1[j] + (1.0/6.0)*(k[0][j] + 2.0*k[1][j] +
                2.0*k[2][j] + k[3][j]);

                // The robot state has been updated

          }          // end of loop for j=1,4
      }              // end of loop for the if i=4 decision
}                    // end of the for loop i=1,4

// Now we throw accel into the returned variable. We need this for
// our estimated acceleration signal.
y[4] = q1_ddot_hat;
y[5] = q2_ddot_hat;

} // End of rk_observer function
```