Kalman Filter
Estimation of angular velocity and acceleration
On-line implementation

Pierre Canet

TR-CIM-94-15 November 22, 1994

Department of Electrical Engineering
McGill Centre for Intelligent Machines
McGill University
Montréal, Québec, Canada

Project report submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Engineering

Postal Address: 3480 University Street, Montreal, Quebec, Canada, H3A 2A7
Telephone: (514) 398-7357 Fax: (514) 398-7348
E-mail: canet@mcrcim.mcgill.edu

Contents

List of Figures

List of Tables

Abstract

This report deals with the on-line implementation of a *low velocity modified Kalman filter algorithm.* This algorithm consists of the classical time-varying Kalman filter at high velocity and in the well-known constant time algorithm when the velocity is not high enough. This algorithm is based on a simplified model of a DC motor consisting of a 3^{rd} order integrator.

The experimental results show that the estimation is both effective and efficient. Furthermore, a comparison between the *traditionnal algorithms* and the algorithm implemented in this project has been done and the results reveal the power of the latter method compared to the other ones.

Besides, it was observed from simulations that an appropriate choice of plant noise covariance in the model can provide a considerable improvement in the estimates of angular velocities and accelerations.

It was also observed that plant noise covariance was correlated to the angular velocity and a relation has been determined between these two data.

Finally, a comparison between the low velocity modified algorithm with a constant plant noise covariance and the same algorithm with an adaptive covariance provides interesting results and some ideas to improve estimates.

This opens the way for high performance robot arm control without increasing the cost of the final product : without adding expensive devices (tachometer, accelerometer), velocity and acceleration information are now available (at least good estimation) and could be used to compensate for nonlinear dynamics.

Résumé

Ce rapport traite de l'implementation en ligne de l'algorithme de filtrage de Kalman modifié à basse vitesse. Cet algorithme peut se décomposer en deux parties : à grande vitesse, l'algorithme à période d'échantillonage variable est utilisé tandis qu'à basse vitesse, c'est l'algorithme à période d'échantillonage constante qui est considéré. Cet algorithme modifié a été utilisé sur le modèle simplifié (intégrateur de troisième ordre) d'un moteur à courant continu.

Les résultats expérimentaux montrent que les estimations sont de bonne qualité. De plus, une comparaison entre les algorithmes traditionnels et l'algorithme modifié a été faite et les résultats montrent la puissance de cette dernière méthode comparée aux autres.

Par ailleurs, les simulations permettent d'observer qu'un choix approprié de la covariance du bruit de modèlisation peut conduire à une grande amélioration des valeurs estimées de la vitesse angulaire et de l'accélération.

De ces mêmes simulations, il a été constaté que la valeur de la covariance du bruit de modèlisation était liée à la vitesse angulaire et une relation entre ces deux données a été déterminée.

Enfin, une comparaison entre l'algorithme modifié utilisé avec une covariance de bruit de modèlisation constante et le même algorithme utilisé avec une valeur de covariance variant avec la vitesse angulaire donne d'intéressants résultats et quelques idées pour améliorer les estimations.

Ceci ouvre la voie du contrôle de haute performance en robotique sans augmenter le prix du produit fini : sans ajouter de capteurs onéreux (tachomètre, accéléromètre), la vitesse et l'accélération (au moins une bonne estimation) sont maintenant disponibles et peuvent être utilisées pour compenser certaines non-linéarités.

Acknowledgements

I would like to express my deepest gratitude and appreciation to my research supervisor, Prof Pierre R. Belanger. His excellent guidance, his broad knowledge of the subject and his help were essential for the completion of my research project.

I would also like to thank Paul Dobrovolny, Hamid Tadghirad and Ahmed Helmy for their help during my research project.

Finally, much gratitude goes to my wife, my parents and my friends for providing various forms of support over the past sixteen months while doing a full-time Master of Electrical Engineering program.

Chapter 1

Introduction

In commercial robots, the common device used to measure the position is an optical shaft encoder which is a disk with two sets of regularly-spaced slots set along concentric circles. The basic principle of such a device is that a pulse is produced each time a slot passes in front of a light beam. The angular displacement is determined by counting the net number of pulses in a given direction and multiplying this count by the (constant) angle between two slots. As the two sets are in quadrature, it is possible to determine direction of motion by knowing which pulse train leads the other.

High performance robot control systems usually call for velocity and/or acceleration information of the joints. Tachometers are not always used in robotics fields for economic considerations; the angular velocity is often obtained from the joint angle measurements. As is the case with velocities, acceleration is also deduced from angular position.

The probem of velocity estimation could be solved with two approaches. The first one could be divided into two categories:

- the **finite difference method** where the net number of pulses during a fixed interval of time is counted, then multiplied by the (constant) angle between two successive slots and finally divided by the duration of time. This method is fully described in references [2][3][4],

- the **inverse-time method** where velocity is inferred as the interpulse angle divided by the time between successive pulses [5][6].

The other group of solutions consists in posing the problem in terms of Kalman filtering. Two different solutions have been proposed in [1]:

- a **steady-state Kalman filter**; the estimates are computed each n milliseconds where n is a fixed duration; this solution may be seen as an heir to finite-difference methods,
- a **time-varying Kalman filter**; the estimates are updated each time an encoder pulse is received; this solution may look close to the interpulse-time method.

In [1], robot dynamics are not used and the joint motions are modeled as the outputs of independent linear filters driven by white noise. This presents the advantage of simplicity of adaptation.

The method presented in this report uses the same manipulator model as in [1] but the solution proposed here consists in a fusion of the two solutions mentionned above. At high velocity the algorithm considered in this project is the time-varying Kalman filter algorithm but at low velocity, a modification is introduced. At low velocity, the interpulse time can be relatively long and as there is information in the fact that a pulse was not received during a given time interval. Therefore, the estimator is triggered by both encoder pulse arrivals and by the clock.

The report proceeds as follows. The system model, the *classical* Kalman filter algorithms (steady-state and time-varying) and the low velocity modified algorithm are described in Section 2. The details of implementation are given in Section 3 and Section 4 deals with the experimentations. Conclusion is given in Section 5.

Chapter 2

Description of the method

2.1 System model

As mentionned in the introduction, the system model used here is simply an integrator model. The justification of a multiple-integrator signal model is given in [1]. In brief, the results come from the fact that modern robots operate at high sampling rates and as the sampling time tends to zero, the asymptotic analysis shows that the full manipulator model acts as a multiple-integrator system. The system is described by the following set of equations:

$$\dot{X}(t) = AX(t) + \Gamma\omega(t) \tag{2.1}$$

$$y(t) = \Theta(t) = CX(t) + e(t) \tag{2.2}$$

$$A = \begin{bmatrix} 0 & 1 & 0 & \dots & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 \\ \dots & \dots & \dots & & & \\ 0 & 0 & 0 & \dots & 0 & 1 \\ 0 & 0 & 0 & \dots & 0 & 0 \end{bmatrix}$$

$$C = [1 \quad 0 \quad 0 \quad \dots \quad 0]$$

$$\Gamma = [0 \quad 0 \quad 0 \quad \dots \quad 1]^T$$

The components $x_1, x_2, x_3, \dots$ of X vector which represents the state of the system, are respectively the angular position, angular velocity, angular acceleration, etc. The system outputs y represent the available information on the system, namely the measurements. Here, only the angular position $\Theta(t)$ is measured so that y is a scalar. The system model error $\omega(t)$ is assumed to be white Gaussian noise, with zero mean and q covariance. The nature of the measurement error $e(t)$ and its covariance r depend on the algorithms. Section 2.4 deals with the description of the measurement error.

As both velocity and acceleration are to be estimated, the system order must be at least 3 (using a second order model, the acceleration would not be available). The description of the triple and quadruple integrator model follows.

Triple integrator model:

$$X(kT+T) = \begin{bmatrix} 1 & T & \frac{1}{2}T^2 \\ 0 & 1 & T \\ 0 & 0 & 1 \end{bmatrix} X(kT) + W(kT)$$

$$y(kT) = [1 \quad 0 \quad 0]\, X(kT) + e(kT) \tag{2.3}$$

with

$$Q = E\left[W(kT)W^T(kT)\right] = q \begin{bmatrix} \frac{1}{20}T^5 & \frac{1}{8}T^4 & \frac{1}{6}T^3 \\ \frac{1}{8}T^4 & \frac{1}{3}T^3 & \frac{1}{2}T^2 \\ \frac{1}{6}T^3 & \frac{1}{2}T^2 & T \end{bmatrix} \tag{2.4}$$

Quadruple integrator model:

$$X(kT+T) = \begin{bmatrix} 1 & T & \frac{1}{2}T^2 & \frac{1}{6}T^3 \\ 0 & 1 & T & \frac{1}{2}T^2 \\ 0 & 0 & 1 & T \\ 0 & 0 & 0 & 1 \end{bmatrix} X(kT) + W(kT)$$

$$y(kT) = [1 \quad 0 \quad 0 \quad 0]\, X(kT) + e(kT) \tag{2.5}$$

with

$$Q = E\left[W(kT)W^T(kT)\right] = q \begin{bmatrix} \frac{1}{252}T^7 & \frac{1}{72}T^6 & \frac{1}{30}T^5 & \frac{1}{24}T^4 \\ \frac{1}{72}T^6 & \frac{1}{20}T^5 & \frac{1}{8}T^4 & \frac{1}{6}T^3 \\ \frac{1}{30}T^5 & \frac{1}{8}T^4 & \frac{1}{3}T^3 & \frac{1}{2}T^2 \\ \frac{1}{24}T^4 & \frac{1}{6}T^3 & \frac{1}{2}T^2 & T \end{bmatrix} \tag{2.6}$$

2.2 Classical methods

As mentioned in the introduction the method used here comes from the mergence of two classical methods. This explains why the description of these algorithms is given before the description of the low velocity modified algorithm.

All algorithms, the low velocity modified method included, are composed of two steps that are performed for each recursion, namely the estimate (observation) update and time propagation between measurements. The first stage determines the current estimate $\hat{x}(k|k)$ in terms of the current measurement $y(k)$ and an *a priori* estimate $\hat{x}(k|k-1)$ [1] . This step corresponds to the Ricatti equation propagation. The second step consists in computing the so-called *a posteriori* time update $\hat{x}(k+1|k)$ in terms of $y(k)$.

The main difference between the algorithms described below is the computation time.

2.2.1 Constant time Kalman filter

Using a constant time Kalman filter, a sampling period T is chosen and maintained constant during the study period. As **A** and **Q** matrices are only expressed in terms of T, these two matrices remain constant during the study period. A new encoder output data is read and a new estimation is computed at each sampling instant. The algorithm follows:

<u>First Step: Ricatti equation propagation:</u>

- Kalman gain computation:

$$K(k) = P(k|k-1)C^T[CP(k|k-1)C^T + R(k)]^{-1} \tag{2.7}$$

[1]The notation $\hat{x}(n|p)$ means the estimate of x is computed at the n instants according to information available at the instant p.

where $R(k)$ is the covariance matrix of the measurement noise.

- $\hat{x}(k|k)$ computation:

$$\hat{x}(k|k) = \hat{x}(k|k-1) + K(k)[y(k) - C\hat{x}(k|k-1)] \tag{2.8}$$

- Estimation error covariance matrix computation:

$$P(k|k) = P(k|k-1) - K(k)CP(k|k-1) \tag{2.9}$$

<u>Second Step: Time propagation:</u>

- One-step ahead state estimate prediction:

$$\hat{x}(k+1|k) = A\hat{x}(k|k) + Bu(k) \tag{2.10}$$

- One-step ahead state estimate covariance prediction:

$$P(k+1|k) = AP(k|k)A^T + Q(k) \tag{2.11}$$

2.2.2 Time-varying Kalman filter

This filter could be considered as a pulse-driven Kalman filter since computations are performed each time a pulse arrives. So now, the time between two successive updates is no longer constant. As a result of this, A and Q matrices are no longer constant. Nevertheless, the problem formulation is the same as in the previous subsection with T replaced by the time interval between the two last successive pulses.

2.3 Low velocity modified Kalman filter

As mentioned above, both traditional algorithms are not optimal: constant time Kalman filter is not as powerful as it should be at high velocity because updates are not computed as often as desired (during a sampling period several pulses may arrive) and at low velocity, time-varying Kalman filter could lose information since at such velocities time between two pulses may be relatively long.

To avoid these disadvantages, an estimator triggered by both encoder pulse arrivals and by the clock is considered. This estimator computes updates either when a pulse arrives or when the time elapsed since the last pulse arrived is greater than a chosen time T. This could be seen as a velocity-driven *adaptive* estimator: at low velocity, this estimator performs like constant-time Kalman filter and on the other extreme, at high velocity, it acts like a time-varying Kalman filter.

2.4 Measurement error

The measurement error value depends on the algorithms. As far as the constant time Kalman filter is concerned, the scalar output $y(t)$ is equal to the latest encoder count times the resolution Θ_m. So the measurement error $e(t)$ is the quantization error, assumed white, zero mean and uniformly-distributed between $\pm\Theta_m$. The variance R of such a signal is $\frac{1}{3}{\Theta_m}^2$.

With the time-varying Kalman filter algorithm, the measurement noise is now the error between the ideal and the actual quantization levels, essentially due to the imperfect alignment of the slots. An experiment done in [9] shows that the misalignment errors are Gaussian, independent random variables. The error standard deviation is $\sigma_\epsilon = 0.00645\,\mathrm{deg}$.

Then, for the modified time-varying Kalman filter, the measurement error variance is either equal to $\frac{1}{3}{\Theta_m}^2$ when the computations are performed at the clock signal or to ${\sigma_\epsilon}^2$ when done at a pulse occurrence.

Chapter 3

Implementation

The aim of this chapter is to provide some details about the implementation. The C code written during this project could be divided into two parts:

- a part concerning the implementation of the Kalman filter computation loop (called UD.c);
- a part dealing with the real time aspect of the project (called kalman2.c).

The main programs are given in Appendices A, B, C. For purposes of comparison, some modifications have been done to provide either a constant time or a time-varying Kalman filter. The listing of these modifications are not given here.

3.1 U-D factorization

Theoretically speaking, the first part comprises no difficulties. However, even though the Kalman sequential filter is simple, numerical experience has proven that the

Kalman filter algorithm is sensitive to computer roundoff and that numerical accuracy sometimes degrades to the point where the results cease to be meaningful.

Therefore, using the Kalman filter formulation as described in section 2.2.1 is not accurate, particularly for the covariance matrix computation (the required nonnegativity of this matrix may be altered by computer roundoff).

Bierman in [7] proposed a modified algorithm based on the U-D factorization of the covariance matrix. The complete theory is given in [7] but formulas are rewritten for completeness.

The system is described by equations (2.3) and (2.4). The first step is to perform an upper triangular UDU^T factorization of the *a priori* covariance matrix $P(k|k-1)$ where U is an upper triangular matrix with unit diagonal elements and D a diagonal matrix. This is performed by the following algorithm:

for $j = n, n-1, \dots, 2$

$$\begin{cases} d_j(k|k-1) &= P_{jj}(k|k-1) \\ U_{jj}(k|k-1) &= 1 \\ U_{kj}(k|k-1) &= P_{kj}(k|k-1)/d_j, \text{ k = 1, \dots, j-1} \\ P_{ik}(k|k-1) &= P_{ik}(k|k-1) - U_{ij}(k|k-1)U_{kj}(k|k-1)d_j(k|k-1), \begin{cases} k &= 1,\dots,j-1 \\ i &= 1,\dots,k \end{cases} \end{cases}$$

and then

$$U_{11}(k|k-1) = 1 \text{ and } d_1(k|k-1) = P_{11}(k|k-1)$$

where d_j stands for the j^{th} diagonal element of D and U_{kj} (resp. P_{kj}) for the j^{th} element of the k^{th} row of U (resp. P).

This decomposition is always possible because of the nature of P which is a definite positive matrix.

The next step is to propagate the Ricatti equation. The Kalman gain K and the updated covariance factors U and D can be obtained as follows:

$$\begin{cases} f & = U(k|k-1)^T C \\ v & = D(k|k-1) f \end{cases}$$

$d_1(k|k) = d_1(k|k-1) r/\alpha_1$, with $\alpha_1 = r + v_1 f_1$, $K_2{}^T = (\, v_1 \quad 0 \quad 0 \,)$

where r represents the measurement noise covariance.

For $j = 2, \ldots, n$ recursively cycle through the following equations:

$$\begin{cases} \alpha_j & = \alpha_{j-1} + v_j f_j \\ d_j(k|k) & = d_j(k|k-1)\alpha_{j-1}/\alpha_j \\ u_j(k|k) & = u_j(k|k-1) + \lambda_j K_j \text{ where } \lambda_j = -f_j/\alpha_{j-1} \\ K_{j+1} & = K_j + v_j u_j(k|k-1) \end{cases}$$

where

$$U(k|k-1) = [\, u_1(k|k-1) \ldots u_n(k|k-1)\,],$$
$$U(k|k) = [\, u_1(k|k) \ldots u_n(k|k)\,]$$

and the Kalman gain is given by

$$K = \frac{K_{n+1}}{\alpha_n}.$$

Then, the computations of $\hat{x}(k|k)$, $\hat{x}(k+1|k)$, $P(k+1|k)$ (or $U(k+1|k)$ and $D(k+1|k)$) are performed respectively with the equations (2.8), (2.10) and (2.11).

3.2 Real time loops

Depending on the algorithm used (constant time, time-varying or modified Kalman filter), the real time aspects may change. In this section real time aspects of the modified algorithm are developed because it is the more complex situation.

These aspects have been treated as follows. A main program manages three subroutines respectively called *c-int-04, c-int-09, c-int-10.* The two first subroutines are used to perform the so-called Kalman filter computations while the third one just reads periodically (period T1, timer 1) the accelerometer information (details about the reasons of this will be given in Section 4.1.3).

Nevertheless, as traditional methods are a subset of the algorithm developed here, the parts concerning these methods are given in the diagram at the end of this section.

Further more, two routines for Kalman filter computations are required because two kinds of events -either an encoder pulse or a clock top (period T0, timer 0)- may occur. As these two kinds of events induce quite similar effects, the implementation choice is as follows.

One procedure, *c-int-09*, is always called (at least each T0 seconds). It performs the Kalman filter computations, either with default values of measurement noise standard deviation ($\frac{1}{3}\Theta_m^2$) and position (the previous one) if no pulse arrives or with changed values (σ_ϵ^2 for standard deviation and updated encoder information for position) if a pulse arrives. These changes are made by the *c-int-04* procedure.

The program can be easily described by a Petri net (See Figure 3.1). In this kind of net, circles stand for a state of the system and arrows for an event (an event is required to switch from one state to another).

Notation:

- T_i means that a clock top of timer i arrives and $\bar{T_i}$ the opposite,
- P means that a pulse occurs and $\bar{P}$ the opposite,
- End-*name of procedure* means *name of procedure* ends.

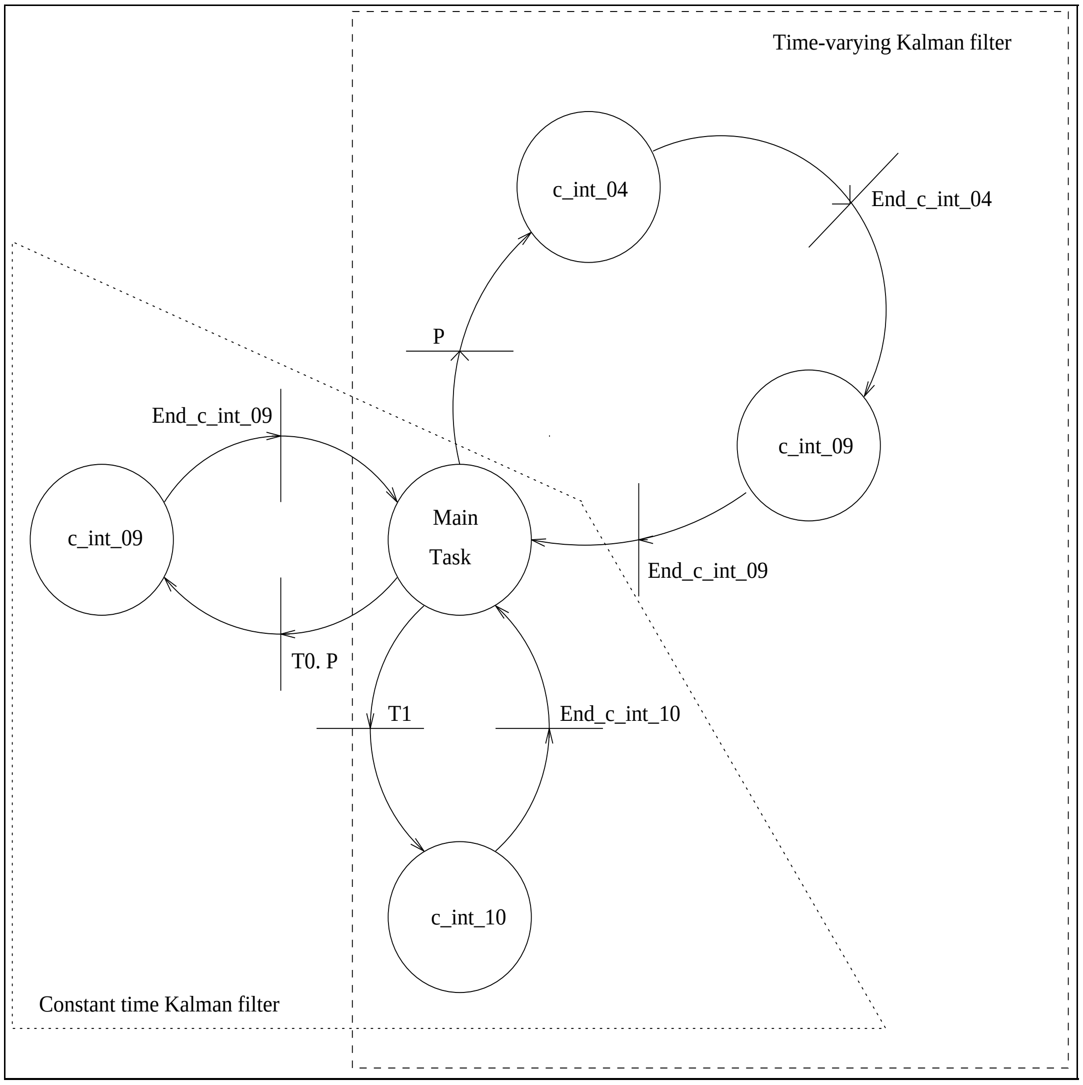


Figure 3.1: Petri net.

Chapter 4

Experimentation

4.1 Setup

The hardware used in the experimentation is now described. The robot manipulator joint consists of a velocity-regulated actuator, a speed-reducing gearbox and an inertial load. The sensors provided are a shaft encoder, a tachometer and an accelerometer. The first sensor is used for Kalman filter computation while the two last sensors (tachometer and accelerometer) are only used for verification of the estimation methods.

4.1.1 Shaft encoder interface

The incremental-type shaft encoder used contains an optical disk having regularly-spaced slots through which a light beam passes. It is an *Optical Hollow Shaft Encoder* (model R80) from RENCO with a resolution of 1000 pulses/rev. It has two pulse train channels A and B having a variable frequency that is linearly proportional to the shaft velocity. The phase between these two channels is $\pm 90°$ (the polarity of the phase shift depends on the angular direction). One index pulse occurs once per revolution.

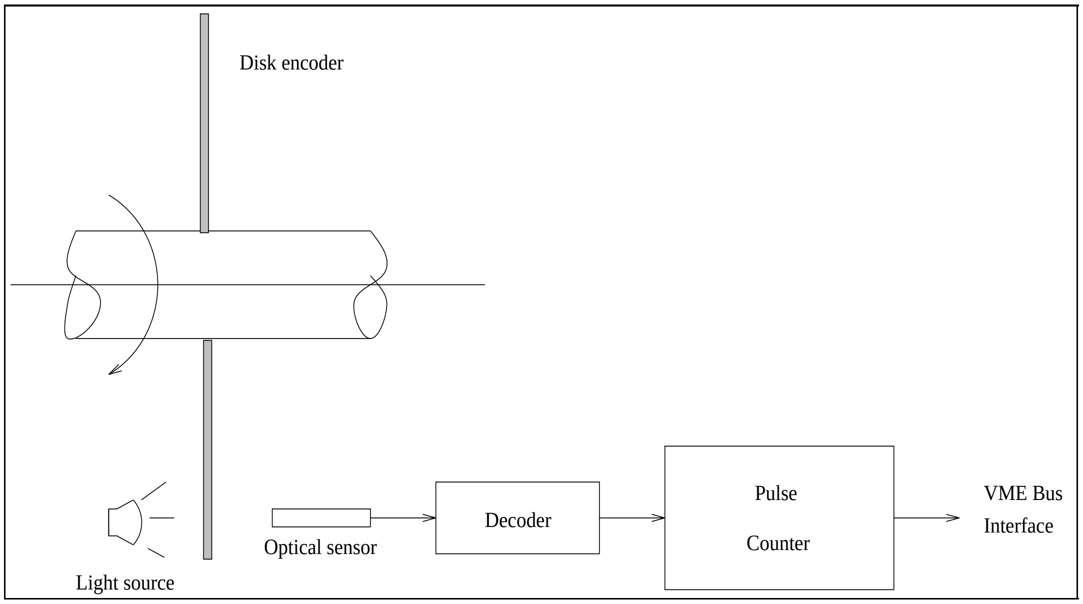


Figure 4.1: The encoder system.

As mentioned in Figure 4.1, a decoder used these two channels to provide information for the pulse counter. This decoder generates pulses at the leading edge and trailing edge of the signals from both channels, as shown in Figure 4.2. So, the pulse counter picks up 4 times the pulses emitted by either channel of the encoder during one sampling period.

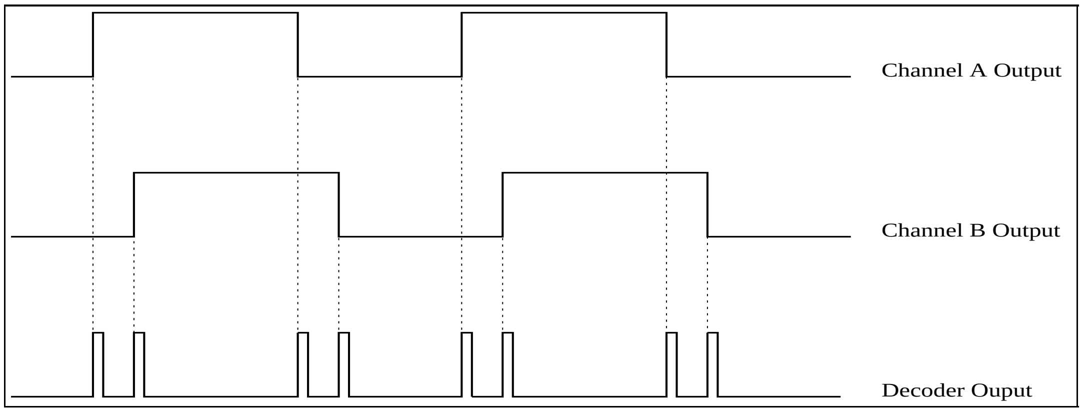


Figure 4.2: Pulse trains from encoder and decoder.

4.1.2 Tachometer

The tachometer used is an electromagnetic device composed of a winding that is integral to the bifilar-wound rotor of the brush-type, permanent DC motor. Voltage-divider resistors and provisions for a passive RC filter are located below the servo controller to reduce the potentially high tachometer output voltage.

4.1.3 Accelerometer

The accelerometer used is comprised of a force-balance servo amplifier and an output stage. It is made by *Lucas Schaevitz Inc., Pennsauken, NJ.* The output stage is an active RC single order, low-pass filter having an 18 Hz cut-off frequency and a unity dc gain so as to avoid saturation. Subsequent filtering of the accelerometer output signal has been numerically implemented using a third order low-pass Butterworth filter having a cut-off frequency of 10 Hz. This is why a constant sampling of the accelerometer output is required (done by the *c-int-10* routine).

4.2 Noise Problem

The main observation concerning the initial results is that the command signal provided by the DAC is noisy. However, in order to analyze the results and the effects of the Kalman filter, a command signal as pure as possible is required (a noisy command may corrupt the velocity and acceleration estimates). This noise is shown on Figure 4.3.

This figure shows two parts of a square signal corresponding to two different amplitudes. The noise magnitude is constant with respect to the command amplitude. As a result of this, if the noise could be ignored at high velocity, it becomes predominent when velocity becomes lower and lower. Further more this noise is coloured with two major frequencies at 60Hz and 180Hz, as shown in Figure 4.4

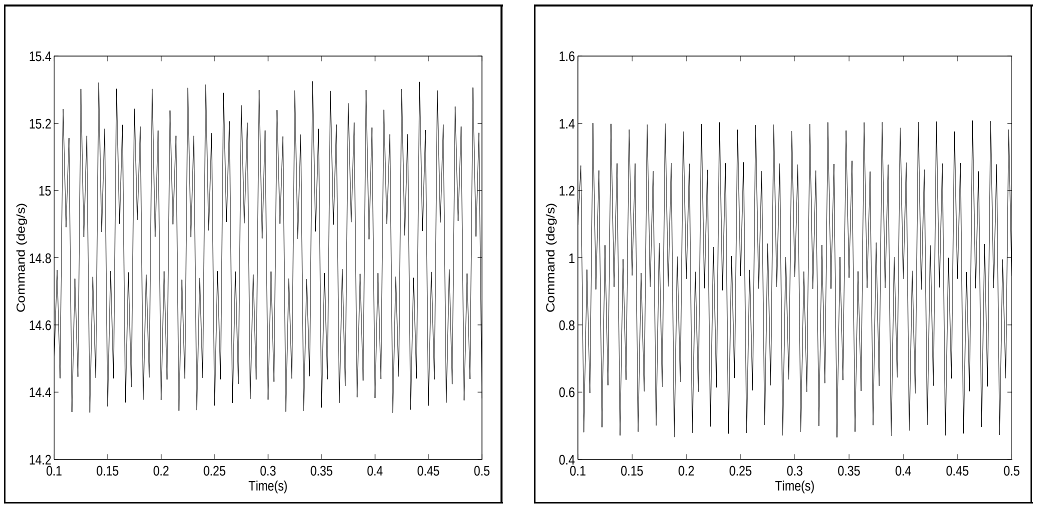


Figure 4.3: Command noise provided by the DAC.

In order to reduce the noise effects a first order filter has been added to the hardware. The choice of the cutoff frequency is not free. The signal used to drive the DC motor is a 1Hz signal. So the cutoff frequency should be between 1Hz and 60Hz, far enough from conboth boundaries to attenuate the perturbations without affecting the main signal. After different trials, the different components of the filter have been chosen as follows :

$$R = 3.3\ k\Omega;$$
$$C = 1\mu F;$$

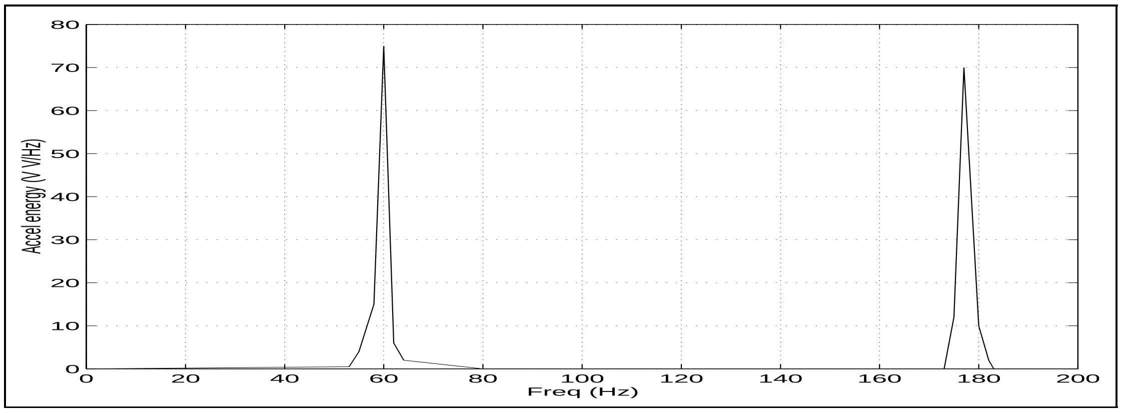


Figure 4.4: Fast Fourier Transform of the DAC command signal

The Figure 4.5 provides the filtered DAC output.

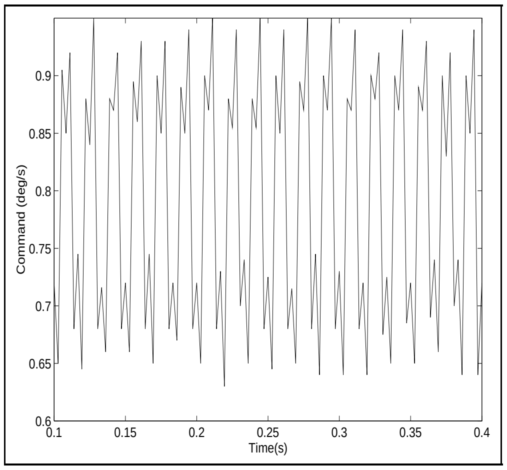


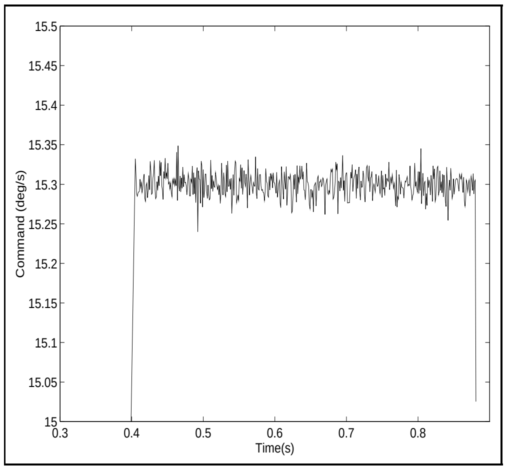
Figure 4.5: Filtered command noise.

The amplitude of the command signal is the same as in Figure 4.3(b). As far as the major signal is concerned, a little attenuation appears and the noise magnitude is divided by 3.

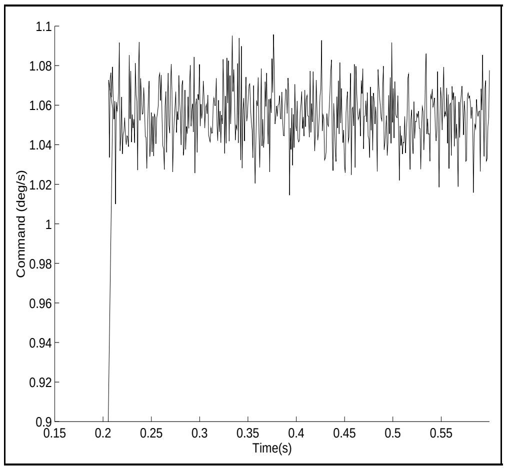
Even if the quality of the command signal is better, the results could be still improved. The second step of the experimentation has consisted in replacing the DAC by a frequency generator. The Figure 4.6 shows the noise provided by this latter power supply for different values of command signal magnitude (these magnitudes are about the same as in the Figure 4.3).

The noise magnitude is smaller now than with the DAC (divided by 30 with respect to non-filtered noise magnitude and by 10 with respect to filtered noise magnitude). Due to the very small noise magnitude value, no first order filter has a significant effect on the noise.

Consistently, the frequency generator has been used to provide the command signal.



(a)



(b)

Figure 4.6: Command noise provided by the frequency generator.

4.3 Experimental results

The experimentation comprises four parts:

1. a comparison of the modified algorithm to the classical methods,
2. the effects of q parameter on the results,
3. the identification of a characteristic curve $q = f(\text{angular velocity})$,
4. a comparison of the modified algorithm using a constant q to the same algorithm using an adaptive q depending on the estimated velocity.

All experiments have been done with the third order integrator model.

Furthermore, the maximum time interval T between two successive estimate computations is 1 millisecond.

4.3.1 Comparison of low modified algorithm to classical methods

An expected result was that increased information would lead to better estimations. Accordingly, constant time Kalman filter is expected to be more powerful than time-varying Kalman filter at low velocity since this latter method does not perform calculations as often at such velocities. On the other hand, the time-varying method is more appropriate at high velocity since it allows to update the Kalman gain as often as a pulse arrives (at such velocities, several pulses can occur during the sample period used in the constant time Kalman filter).

As the modified algorithm acts like the constant time Kalman filter at low velocity and like the time-varying Kalman filter at high velocity, it is expected to be more powerful than the time-varying Kalman filter at low velocity and than the constant time Kalman filter at high velocity.

Figures 4.7 and 4.8 show the comparison between the estimates obtained with the modified algorithm and the estimates obtained with the time-varying Kalman filter. Velocity and acceleration errors are given on the same Figures. The command signal used is a sine wave. Its amplitude is about 300 deg / sec and its offset is 150 deg / sec. The chosen value for q is 1000.

The improvement due to the modified algorithm is especially obvious on the acceleration curves. On Figure 4.7, the estimated acceleration follows the actual one without any distorsion. On the other hand, the estimated acceleration on Figure 4.8 presents a distorsion compared to the actual acceleration when the acceleration increases. This part of the curve (when the acceleration increases) corresponds to the lower part of the velocity curve (low velocity). This shows the estimates with the time-varying Kalman filter become poorer and poorer when the velocity decreases whereas estimates obtained with the modified algorithm are always as good.

On the other hand, when the acceleration decreases (high velocity) estimates obtained with the time-varying Kalman filter are as good as the estimates obtained with the modified algorithm.

It is clear on Figure 4.8 that the velocity and acceleration errors increases with time due to the estimates distortion at low velocity.

Figure 4.9 represents the estimates obtained with the constant time Kalman filter, the velocity and acceleration errors. Compared to results obtained with the modified algorithm, this experiment shows the estimates are not so accurate as far as both velocity and acceleration are concerned.

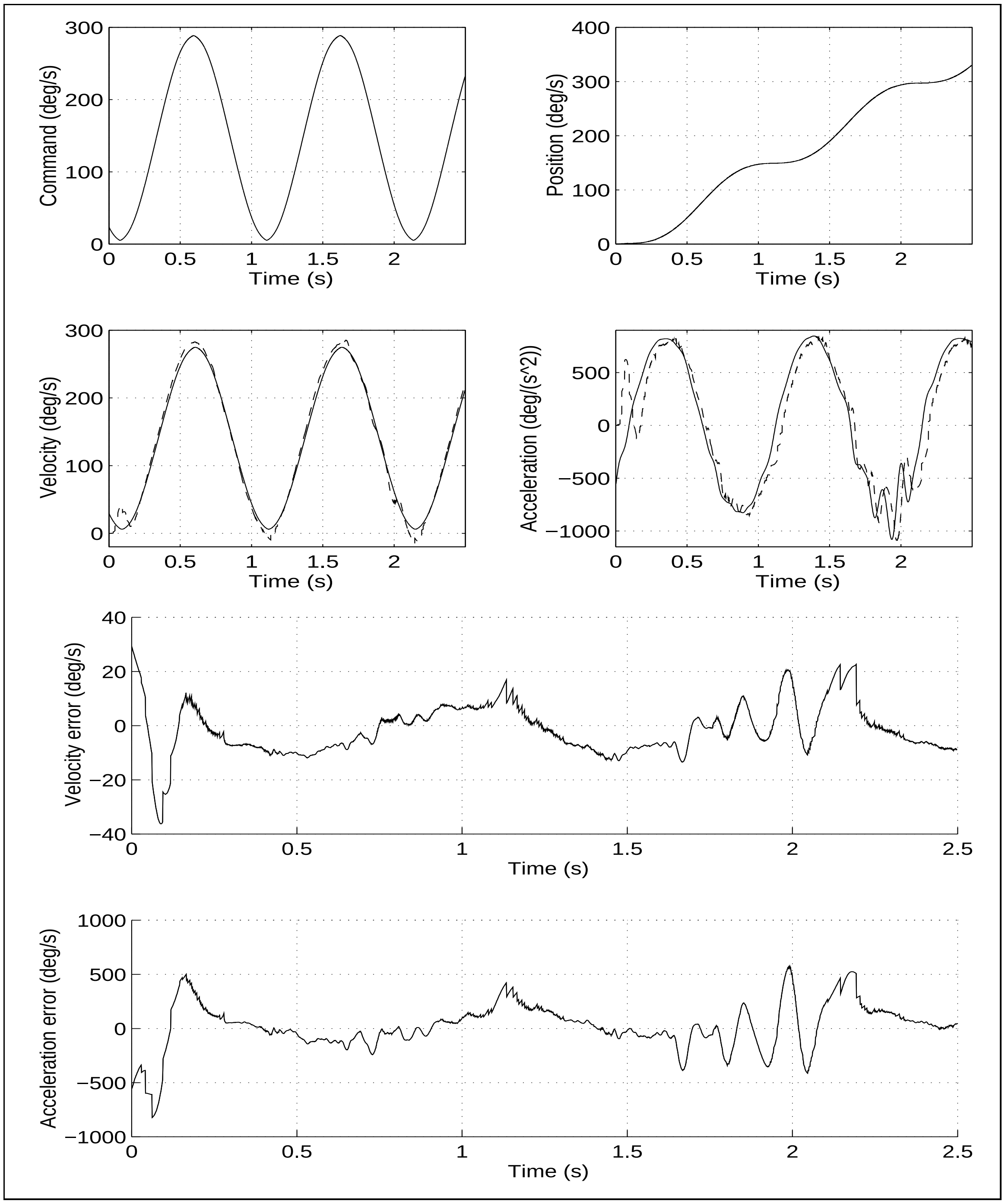


Figure 4.7: Estimates and errors obtained with the modified algorithm (q = 1000 - amplitude = 300 deg / sec - offset = 150 deg / sec)

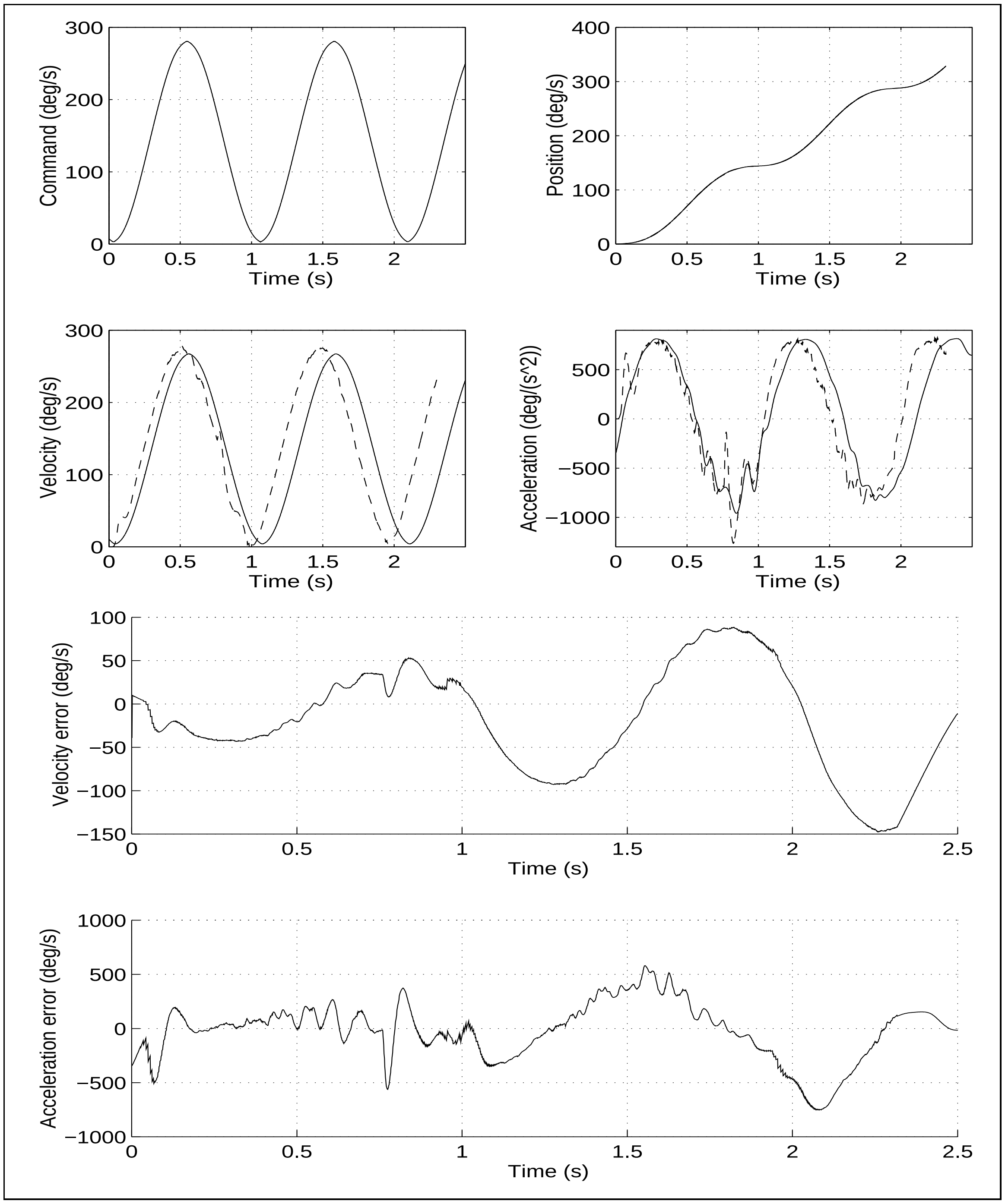


Figure 4.8: Estimates and errors obtained with the time-varying Kalman filter (q = 1000 - amplitude = 300 deg / sec - offset = 150 deg / sec)

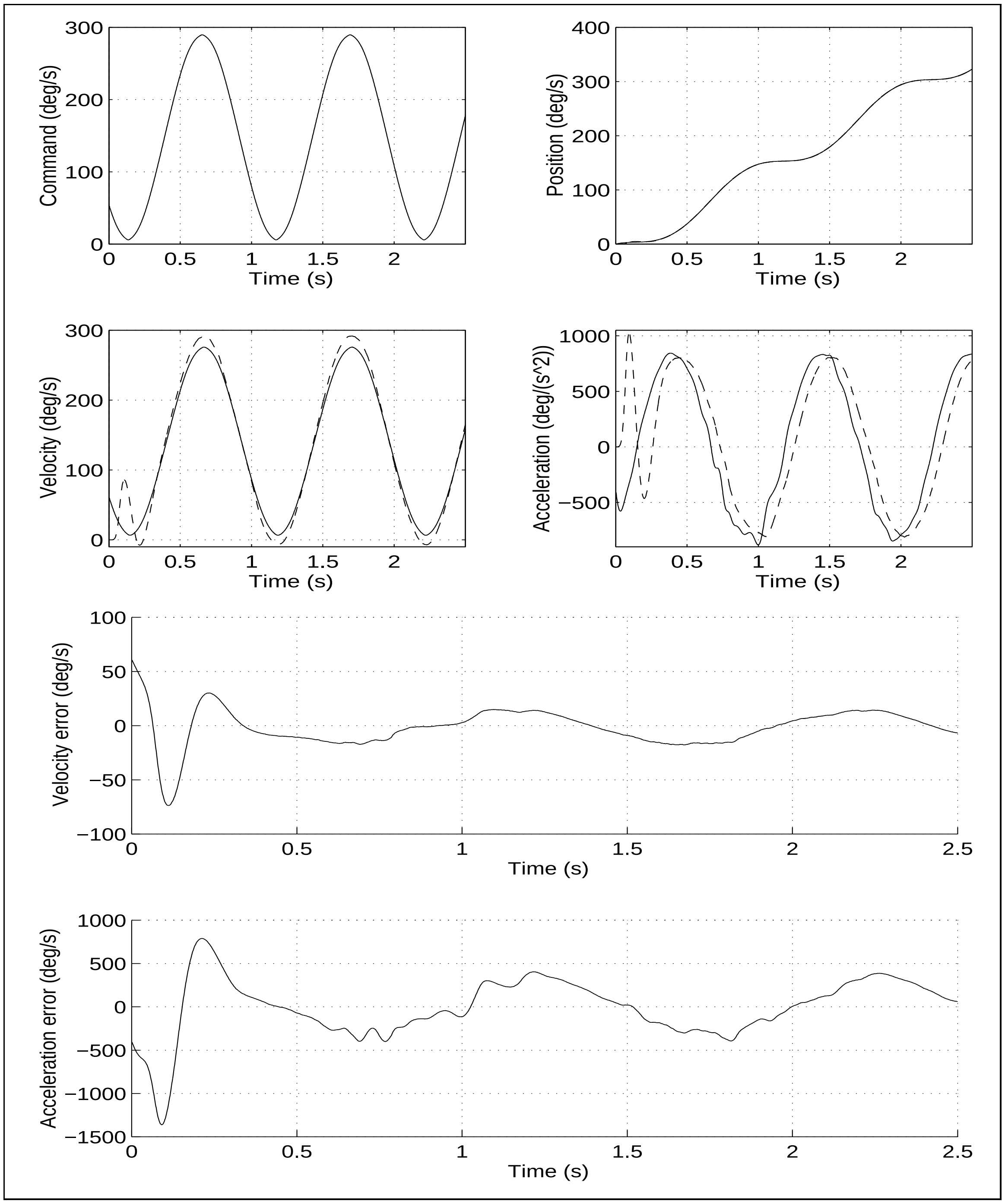


Figure 4.9: Estimates and errors obtained with the constant time Kalman filter (q = 1000 - amplitude = 300 deg / sec - offset = 150 deg / sec)

4.3.2 Q effects

In the model given in section 2.1, the q parameter represents the covariance of the plant noise (white, Gaussian). It can be seen as a parameter to tune since, by definition, the plant noise is mainly unknown.

For the first experiments (See Figures 4.10 and 4.11), the command signal is a sine wave (offet = 15 deg / sec and magnitude = 30 deg / sec). The modified algorithm has been used since the experiments described in the previous section have proven the power of this algorithm compared to the traditional methods.

Figure 4.10 and Figure 4.11 represent the estimates, the velocity and acceleration errors respectively obtained for q = 1000 and q = 100. These two figures show that getting the optimal q is a trade-off between a noisy estimated signal but in phase with the actual one (great q) and a smooth but lagged estimated signal (small q).

The second part of the experimentation (See Figures 4.12 and 4.13) shows a fact developed in detail in the following section. The experimental conditions are the same except that the offset of the command signal is about 150 deg / sec and its amplitude is 300 deg / sec.

Figure 4.12 and Figure 4.13 represent the estimates, the velocity and acceleration errors respectively obtained for q = 1000 and for q = 100. But now at high velocity, the optimal q has not the same value as at low velocity. At high velocity, the optimal q is about 1000 while at low velocity it is about 100. So the optimal q and the velocity seem to be well correlated.

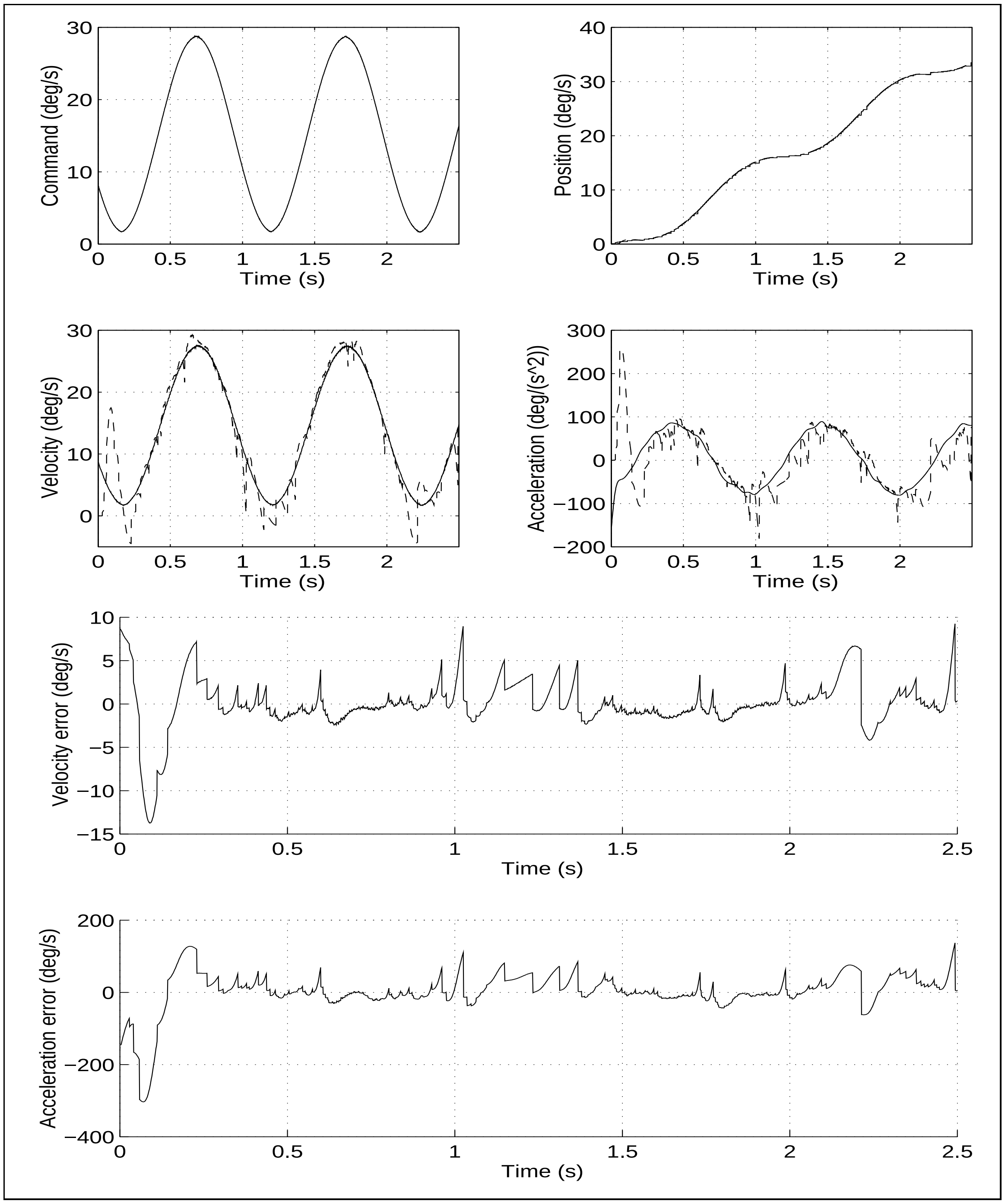


Figure 4.10: Estimates and errors obtained with the modified algorithm (q = 1000 - amplitude = 30 deg / sec - offset = 15 deg / sec)

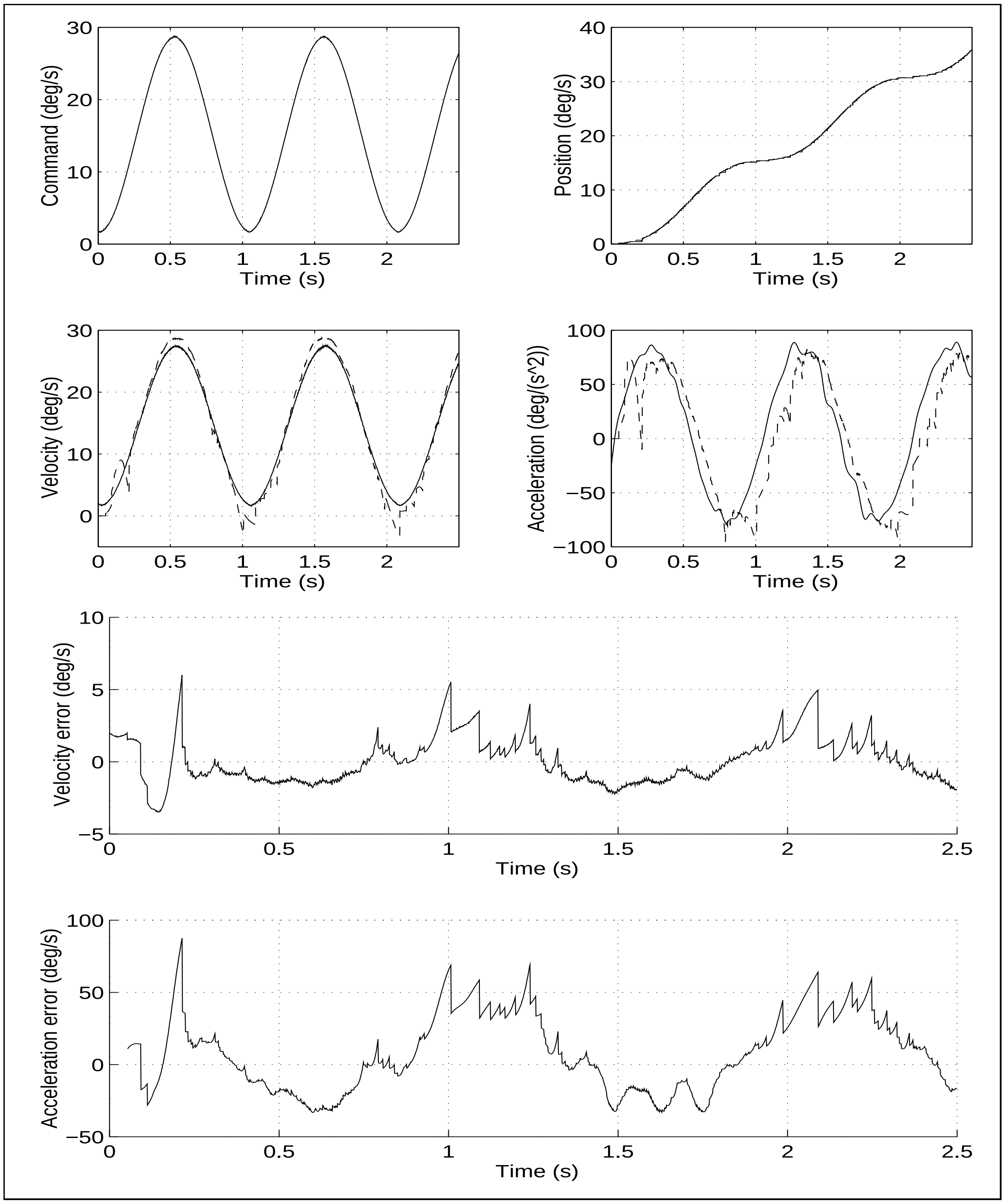


Figure 4.11: Estimates and errors obtained with the modified algorithm (q = 100 - amplitude = 30 deg / sec - offset = 15 deg / sec)

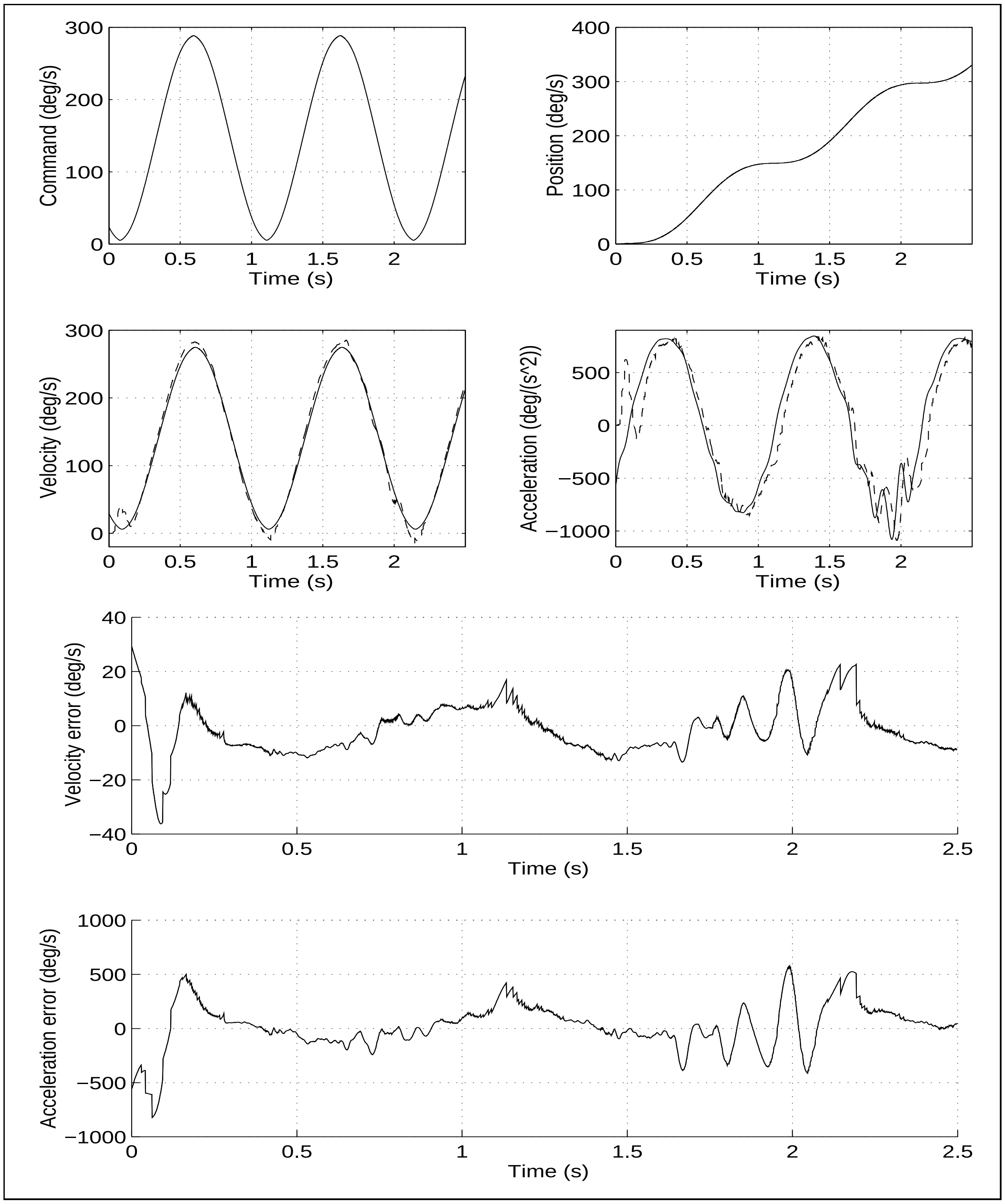


Figure 4.12: Estimates and errors obtained with the modified algorithm (q = 1000 - amplitude = 300 deg / sec - offset = 150 deg / sec)

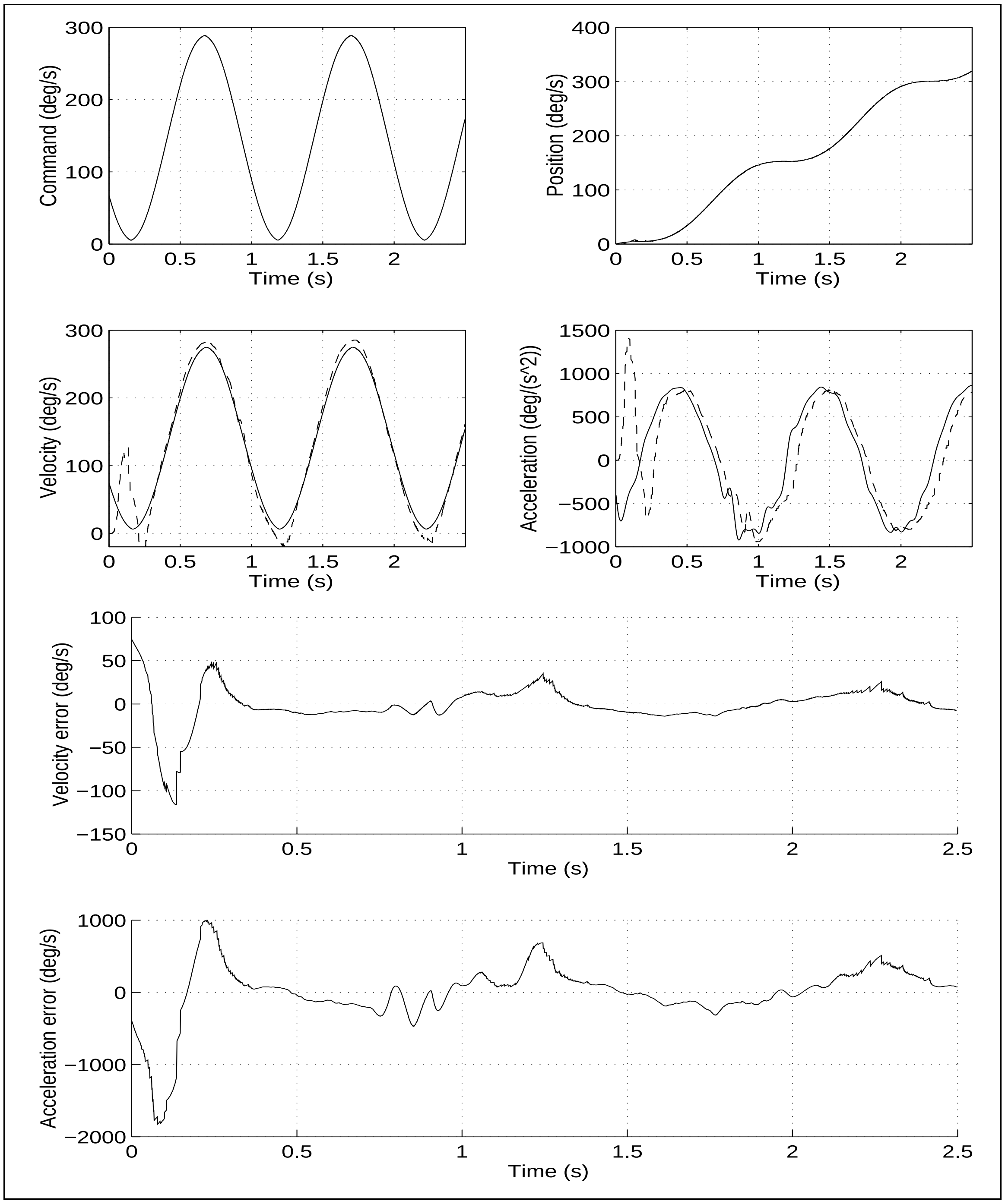


Figure 4.13: Estimates and errors obtained with the modified algorithm (q = 100 - amplitude = 300 deg / sec - offset = 150 deg / sec)

4.3.3 Relationship between q and the angular velocity

As mentioned above, the covariance q of the white Gaussian plant noise can be viewed as a filter parameter to be adjusted. In [8] and [9] some experiments show that the optimal value of q depends on the order of the model (the higher the order, the larger the value), and the value of angular velocity. However, no quantitative study have been done to determine a relationship between q and one of the influent parameters. This section provides some quantitative results on the variations of q with respect to the angular velocity.

The command signals are sine waves. Their amplitude is constant, about 50 deg / sec but their offset is varying between 275 deg / sec and 95 deg / sec. Then a complementary study has been done at low velocity with sine command signals with a 20 deg / sec constant amplitude and an offset varying between 65 deg / sec and 27 deg / sec. These experimental conditions have been chosen so that the variation of the velocity amplitude is almost insignificant with respect to the offset value.

The results are summed up in Table 4.1. Since for each velocity there is a certain range of q (fairly large, depending on the velocity) within which the Kalman filter gives stable and similar estimation results, three values of q are provided:

- *q min* which is the lower limit of the optimal range,
- *q max* which is the upper limit of the optimal range,
- *q optimal* which is the value chosen as optimal to determine the characteristic function $q = f(\text{angular velocity})$.

Further more, the range width (*q max* - *q min*) is mentioned.

Table 4.1 shows that the range width increases with the velocity offsets. The other fact to point out is that a linear relation exists between the optimal q and the angular velocity. Figure 4.14 represents the minimum q curve, the maximum q curve and the optimal q curve.

Simulation	*Offset* deg / sec	*Magnitude* deg / sec	*q min*	*q max*	*width range*	*q optimal*
1	275	50	750	1000	250	900
2	215	50	500	700	200	600
3	155	50	250	350	100	300
4	125	50	175	225	50	200
5	95	50	80	120	40	100
6	65	20	60	90	30	75
7	38	20	30	50	20	40
8	27	20	20	30	10	25

Table 4.1: Summary of experimentations

According to this net of experimental curves, different trials have been done to approximate as well as possible the optimal q curve with a simple function. On Figure 4.14, two functions have been drawn:

- $q = f_1(offset) = 0.0125 offset^2$ (dashed curve),
- $q = f_2(offset) = 16.3944 + offset^2$ (dashed dotted curve).

A brief statistic study has provided the results given in Table 4.2.

error	*mean*	*standard deviation*
q optimal - $f_1(offset)$	3.5578	23.4733
q optimal - $f_2(offset)$	2.6442	17.4055

Table 4.2: Statistical comparison between the different functions

According to this part, a good approximation to the variation of q with respect to the angular velocity amplitude is a second order polynomial of the form $16.3944 + 0.0118 velocity^2$.

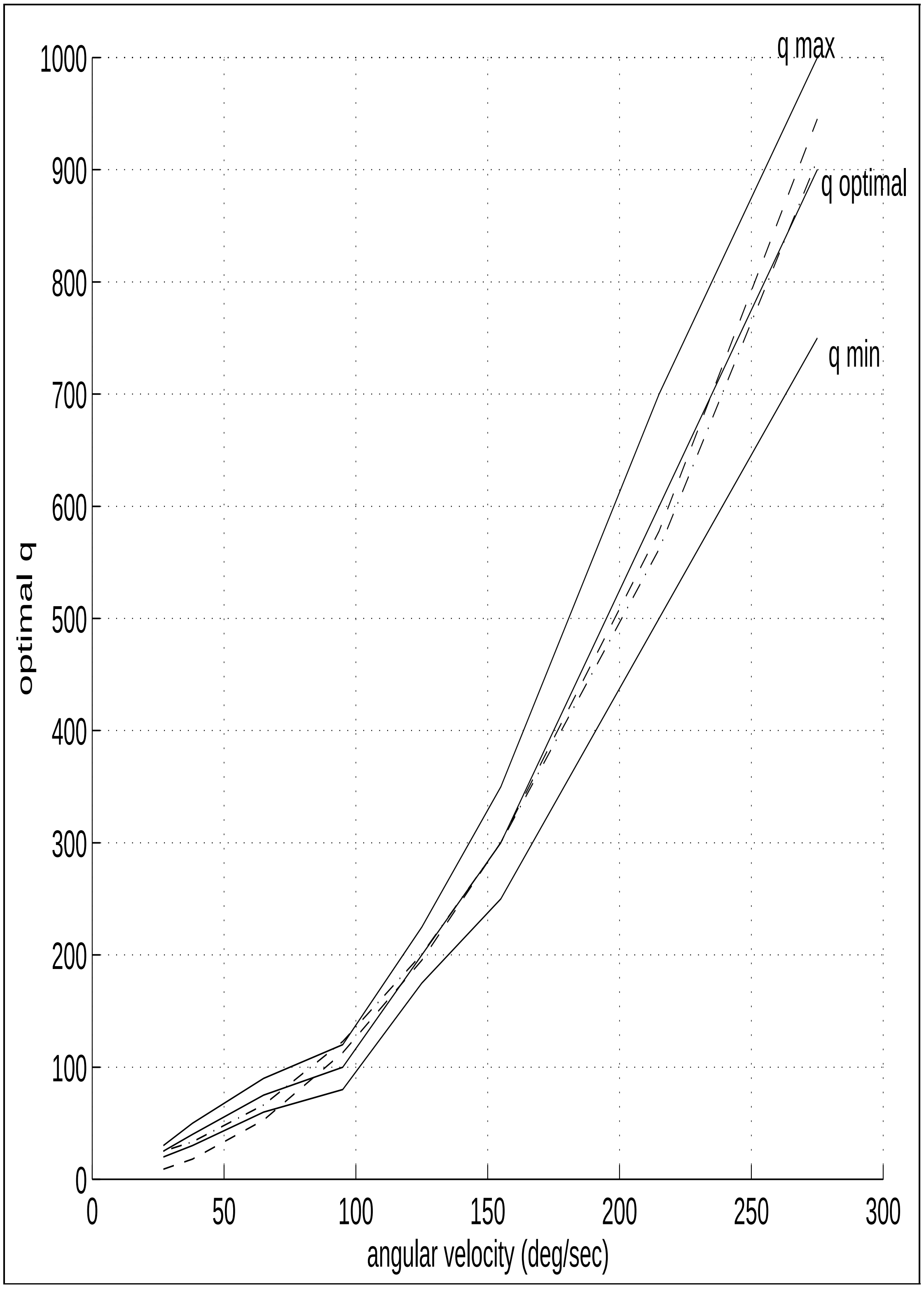


Figure 4.14: Characteristic curve $q = f(angular\ velocity)$ dash line: $q = 0.0125velocity^2$ dash-dot line: $q = 16.3944 + 0.0118velocity^2$

4.3.4 Comparison of the modified algorithm using a constant q to the same algorithm using an adaptive q depending on the estimated velocity

This section uses the previous results and compares estimates obtained with a constant q (here q = 1000) in the modified algorithm to the estimates obtained with an adaptive one. In this latter experiment, the q parameter varies according to the relationship found in the previous section ($q = 16.3944 + velocity^2$).

The command signal used is a square wave (offset = 205 deg / sec and amplitude = 190 deg / sec). This kind of signal has been chosen to show the improvement due to the use of an adaptive q when large changes in the velocity magnitude occur. Figure 4.15 represents the estimates obtained with a constant q and Figure 4.17 the estimates obtained with an adaptive q. Velocity and acceleration errors for each experiments are respectively given in Figure 4.16 and Figure 4.18

The statistical study on the velocity error has provided the results summed up in Table 4.3.

method	*velocity error*		*acceleration error*	
	mean	*standard deviation*	*mean*	*standard deviation*
constant q	4	18.6	60	661.5
adaptive q	3.6	19.1	55.2	661

Table 4.3: Statistical comparison between the constant q method and the adaptive q algorithm

Table 4.3 shows the improvements on velocity estimates due to the adaptive q. The careful comparison of acceleration curves shows smaller overshoots when the adaptive q is used. This fact is obvious by the inspection of acceleration error, too. These remarks lead to conclude that the modified algorithm used with an adaptive q is the most powerful method among all the methods treated in this project.

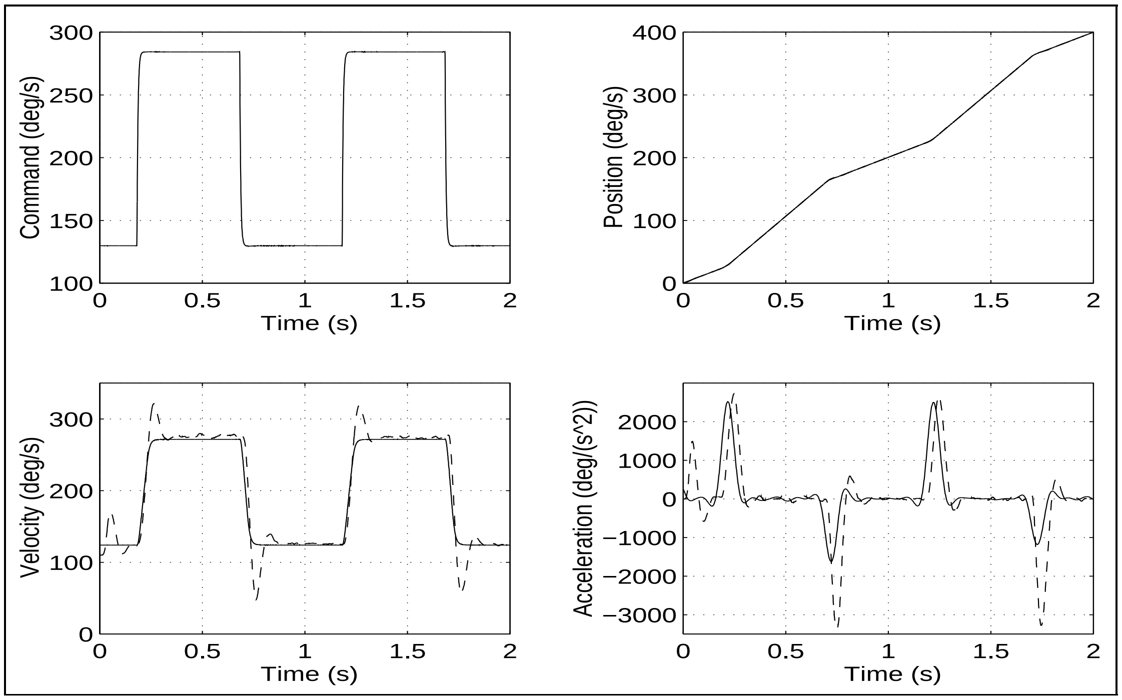


Figure 4.15: Estimates obtained with the modified algorithm (q = cste = 1000)

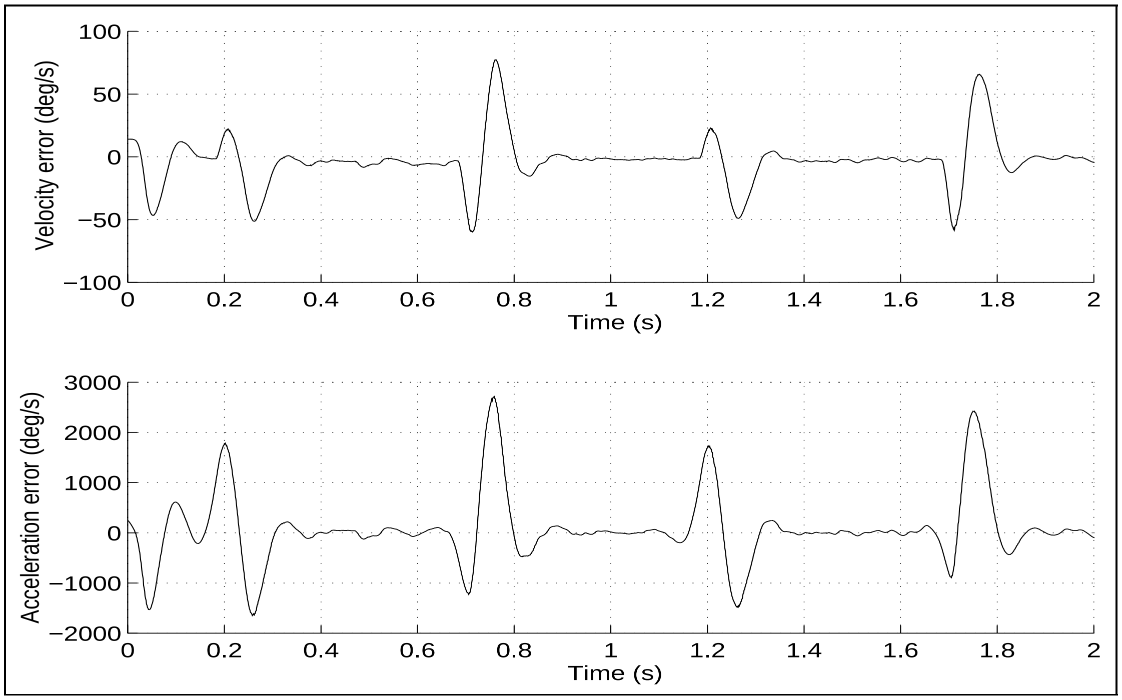


Figure 4.16: Velocity and acceleration errors (q = cste = 1000)

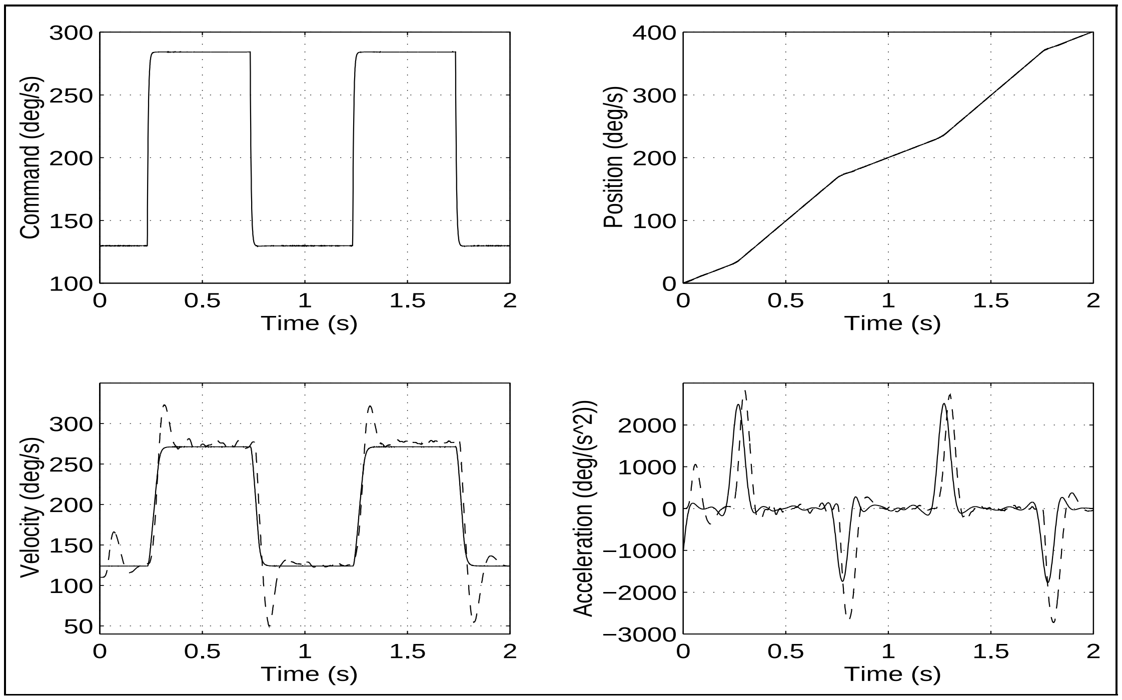


Figure 4.17: Estimates obtained with the modified algorithm (adaptive q)

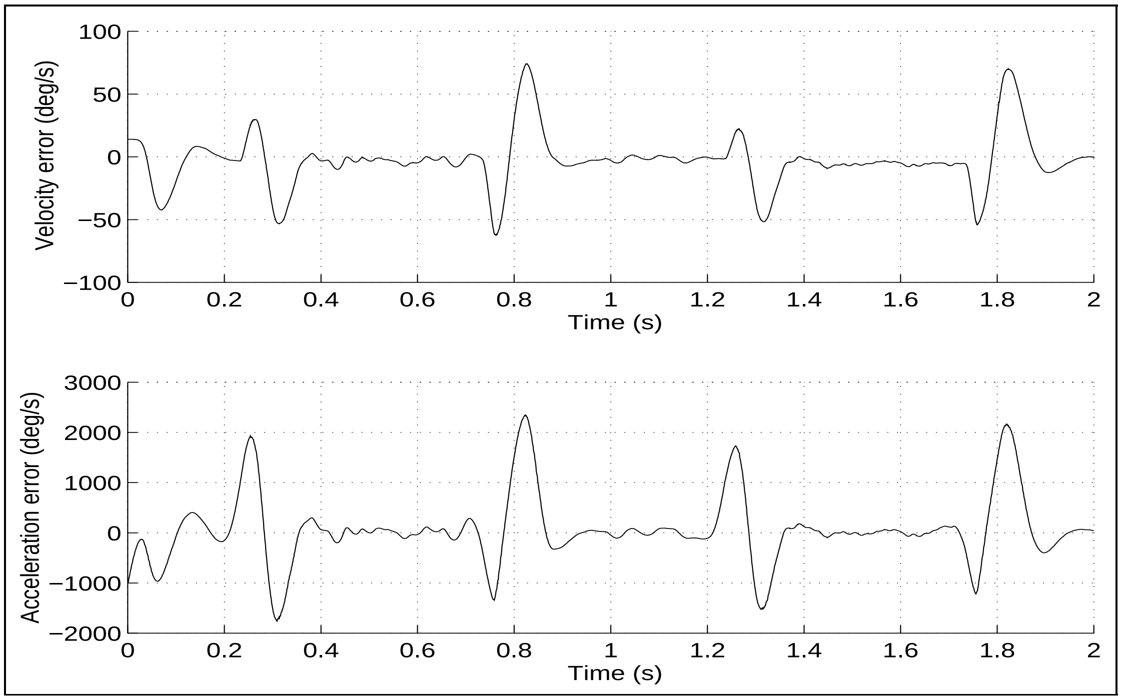


Figure 4.18: Velocity and acceleration errors (adaptive q)

Chapter 5

Conclusion

In this project, a low velocity modified Kalman filter algorithm has been studied, implemented on-line and tested in detail with experimental data. The results obtained show the advantages of using this algorithm rather than the classical ones (constant time and time-varying Kalman filter).

Furthermore, based on the fact that the choice of q has an effect on the performance of the Kalman filter, a relationship has been found between q and the angular velocity. This allowed to provide a new version of the low velocity modified Kalman algorithm where the filter parameter q is not manually adjusted by the programmer but is precisely adjusted by the motor itself. This allows a faster response of the estimates in case of large and fast changes in the angular velocity.

Lastly, even if the way for high performance robot and control without increasing the cost of the final product is opened, some improvements could be still done, like determining a more accurate relationship between q and the angular velocity, using tachometer output, or taking into account backlash and low-speed inaccuracies.

Bibliography

[1] Belanger, P. R., *Estimation of Angular Velocity and Acceleration from Shaft encoder Measurements*, McRCIM report TR-CIM-91-1, McGill University, May 1991.

[2] Sinha, N. K., B. Szavados and D. C. di Cenzo, *New high precision digital tachometer*, Electron. Lett., vol 7, pp 174-176, (1971).

[3] Hoffman de Visme, G., *Digital processing unit for evaluating angular acceleration*, Electron. Eng., vol 40, pp 183-188, (1968).

[4] Dunworth, A., *Digital intrumentation for angular velocity and acceleration*, IEEE Trans. Instrum. Meas., vol IM-18, pp 132-138, (1969).

[5] Habibullah, B., H. Singh, K. L. Soo and L. C. Ong, *A new digital speed transducer*, IEEE Trans. Ind. Electron. Contr. Instr., vol IECI-25, pp 339-343, (1978).

[6] Wallingford, E. E. and J. D. Wilson, *High resolution shaft speed measurements using a microcomputer*, IEEE Trans. Instr. and Meas., vol IM-26, pp 113-116, (1977).

[7] Bierman, G. J., *Factorization methods for discrete sequential method*, Academic Press, New York, San Francisco, London, 1977.

[8] Xiangkai, M., *Angular velocity and acceleration estimation from position measurements*, Master thesis, McGill University, August 1992.

[9] Zhang, J., *Angular Velocity and Acceleration Estimation from Shaft Encoder -Its Experimental Aspects-*, Master thesis, McGill University, April 1993.

Appendix A

UD.c

```
/*-----------------------------------------------------------------------------
 *   FILE: UD.c
 *
 *   Calling Sequence: library of functions used for UD factorization
 *
 *-----------------------------------------------------------------------------*/

#include <stdlib.h>

extern int order;

/*-------------------------------------------------------------------------
 *  Name: A_update
 *
 *  Function: This function computes the new matrix A for the new T value
 *
 *-------------------------------------------------------------------------*/
void A_update(A_current, T)
float  A_current[3][3];
float  T;

{
/* variables declaration */
A_current[0][0] = 1.0;
A_current[0][1] = T;
A_current[0][2] = T*T/2;
A_current[1][0] = 0.0;
A_current[1][1] = 1.0;
A_current[1][2] = T;
A_current[2][0] = 0.0;
A_current[2][1] = 0.0;
A_current[2][2] = 1.0;

}

/*------------------------------------------------------------------------
 *  Name: Q_update
 *
 *  Function: This function computes the new matrix Q for the new T value
 *
 *
 *----------------------------------------------------------------------*/
void Q_update(Q_current, T, q)
float   Q_current[3][3];
float   T;
float   q;

{
/* variables declaration */
Q_current[0][0] = q*T*T*T*T*T/20;
Q_current[0][1] = q*T*T*T*T/8;
Q_current[0][2] = q*T*T*T/6;
Q_current[1][0] = Q_current[0][1];
Q_current[1][1] = 2*Q_current[0][2];
Q_current[1][2] = q*T*T/2;
Q_current[2][0] = Q_current[0][2];
Q_current[2][1] = Q_current[1][2];
Q_current[2][2] = q*T;
}

/*----------------------------------------------------------------------------
 *   NAME: UD_factorization
 *
 *   INPUT: P a define positive matrix
 *
 *   OUTPUT: U an upper unit triangular matrix with D(i) stored on the
 *           diagonals
 *
 *   Function: This function computes U and D such that:
 *                       P = U * diag(D(1),...,D(n)) * U'
 *
 *------------------------------------------------------------------------*/
void UD_factorization(P, U)
float    P[3][3];
float    U[3][3];

{
/* variables declaration */
extern int  order;
int         i,j,k;
float       alpha, beta;


for(i=order-1; i>0; --i)
  {
    U[i][i] = P[i][i]; /* D[i] = P[i][i] */
    alpha = 1/U[i][i];
    for(j=0; j<i; j++)
      {
        beta = P[j][i];
        U[j][i] = alpha * beta; /* U[j][i] = P[j][i]/D[i] */
        for(k=0;k<j+1;k++)
          /* P[k][j] = P[k][j] - U[k][i]*U[j][i]*D[i] */
          P[k][j] -= beta * U[k][i];
      }
  }
U[0][0] = P[0][0];
}


/*-----------------------------------------------------------------------
 *   NAME: UD_update
 *
 *   CAUTION: C is implicitly [1 0 0]
 *
 *   INPUTS: X_a_priori A priori state estimate X(k|k-1)
 *           U_a_priori  Upper unit triangular matrix with D(i) stored on
 *            the diagonals; this corresponds to the a priori covariance
 *            P(k|k-1)
 *           y the observation  y(k) (the new position measurement)
 *           r the observation error variance
 *
 *   OUTPUTS: X_update Updated state estimate X(k|k)
 *            U_update Updated upper unit triangular matrix with the updated
 *             D(i) stored on the diagonals; this corresponds to the updated
 *             covariance P(k|k)
 *            K Kalman Gain K(k)
 *
 *   Function: This function computes:
 *                  - X(k|k),
 *                  - U(k|k) and D(k|k),
 *                  - K(k).
 *
 *------------------------------------------------------------------------*/
void UD_update(X_a_priori, U_a_priori, y, r, X_update, U_update, K)
float   X_a_priori[3];
float   U_a_priori[3][3];
float   y;
float   r;
float   X_update[3];
float   U_update[3][3];
float   K[3];

{
/* variables declarations */
extern int     order;
float          B[3];
int            i, j, k;
float          alpha, beta, lambda, gamma;

y -= X_a_priori[0]; /* y = y - C * X_a_priori */

/* B = D * U' * C */
for(i=order-1;i>0;--i)
    B[i] = U_a_priori[i][i] * U_a_priori[0][i];

B[0] = U_a_priori[0][0];
alpha = r + B[0];
gamma = 1/alpha;
U_update[0][0] = r * gamma * U_a_priori[0][0];
for(i=1; i<order; i++)
  {
    beta = alpha;
    alpha += B[i] * U_a_priori[0][i];
    lambda = -U_a_priori[0][i] * gamma;
    gamma = 1/alpha;
    U_update[i][i] = beta * gamma * U_a_priori[i][i];
    for(j=0; j<i; j++)
      {
        beta = U_a_priori[j][i];
        U_update[j][i] = beta + B[j] * lambda;
        B[j] += B[i] * beta;
      }
  }

y *= gamma;

/* X_update = X_a_priori + B/alpha * (y - C * x) */
for(i=0; i<order; i++)
  X_update[i] = X_a_priori[i] + B[i] * y;

/* K = B/alpha */
for(i=0; i<order; i++)
  K[i] = B[i]/alpha;
}



/*---------------------------------------------------------------------------
 *   NAME: UD_propagation
 *
 *   CAUTION: C is implicitly [1 0 0]
 *
 *   INPUTS: X_update    State estimate X(k|k)
 *           U_update    Upper unit triangular matrix with D(i) stored on
 *            the diagonals; this corresponds to the update covariance
 *            P(k|k)
 *           A State transition matrix
 *           Q Covariance process noise matrix
 *
 *   OUTPUTS: X_a_priori Time updated state estimate X(k|k+1)
 *            U_a_priori Time updated upper unit triangular matrix with
 *             D(i) stored on the diagonals; this corresponds to the updated
 *             covariance P(k|k+1)
 *
 *   Function: This function computes:
 *                  - X(k|k+1),
 *                  - U(k|k+1) and D(k|k+1).
 *
 *-----------------------------------------------------------------------*/
void UD_propagation(X_update, U_update, A, Q, X_a_priori, U_a_priori)
float  X_update[3];
float  U_update[3][3];
float  A[3][3];
float  Q[3][3];
float  X_a_priori[3];
float  U_a_priori[3][3];

{
/* variables declaration */
extern int      order;
float           U[3][3], UD[3][3], P_a_priori[3][3];
int             i, j, k;

/* X propagation X_a_priori = A * X_update */
for(i=0;i<order;i++)
  {
    X_a_priori[i] = X_update[i];
    for(j=i+1;j<order;j++)
      X_a_priori[i] += A[i][j] * X_update[j];
  }

/* U = A * U_update */
for(i=0;i<order;i++)
    for(j=i+1;j<order;j++)
      {
        U[i][j] = U_update[i][j] + A[i][j];
        for(k=1;k<j-i;k++)
          U[i][j] += A[i][k] * U_update[k][j];
      }
```

```
/* UD = U * D */
for(i=0;i<order;i++)
  {
    for(j=i+1;j<order;j++)
      UD[i][j] = U[i][j] * U_update[j][j];
    UD[i][i] = U_update[i][i];
  }

/* P_a_priori = UD * U + Q = A * U_update * D * U_update' * A' + Q */
for(i=0;i<order;i++)
  for(j=0;j<order;j++)
    {
      P_a_priori[i][j] = UD[i][j];
      for(k=j+1;k<order;k++)
        P_a_priori[i][j] += UD[i][k] * U[j][k];
      P_a_priori[i][j] += Q[i][j];
    }
UD_factorization(P_a_priori, U_a_priori);
}

/*----------------------------------------------------------------------------
 *   NAME: UD_loop_pulse_computation
 *
 *   INPUTS: X_update    State estimate X(k|k)
 *           U_update    Upper unit triangular matrix with D(i) stored on
 *            the diagonals; this corresponds to the update covariance
 *            P(k|k)
 *           A State transition matrix
 *           Q Covariance process noise matrix
 *           X_a_priori Time updated state estimate X(k|k+1)
 *           U_a_priori Time updated upper unit triangular matrix with
 *            D(i) stored on the diagonals; this corresponds to the updated
 *            covariance P(k|k+1)
 *
 *   OUTPUTS: The same ones.
 *
 *   Function: This function computes all the matrices.
 *
 *--------------------------------------------------------------------------*/
void UD_loop_pulse_computation(X_a_priori, U_a_priori, y, r, X_update, U_update, Gai
float   X_a_priori[3];
float   U_a_priori[3][3];
float   y;
float   r;
float   X_update[3];
float   U_update[3][3];
float   Gain[3];
float   T;
float   q;
float   A[3][3];
float   Q[3][3];

{
UD_update(X_a_priori, U_a_priori, y, r, X_update, U_update, Gain);
Q_update(Q, T, q);
A_update(A, T);
UD_propagation(X_update, U_update, A, Q, X_a_priori, U_a_priori);
}
```

Appendix B

kalman2.c

```
/**************************************************************************
 * FILE: kalman2.c
 *
 * Function: Node 2 parallel port is used for interrupt-driven asynchronous
 *     sampling of the pulse count from the quadrature decoder, and encoder
 *     interpulse arrival times when the velocity is high enough.
 *       Node 2 timer0 is used for interrupt-driven synchronous sampling of
 *     the pulse count from the quadrature decoder, and decoder interpulse
 *     arrival times at low velocity.
 *       Node 2 timer1 is used for synchronous sampling of acceleration
 *     input.
 *
 *************************************************************************/
#include <math.h>
#include "DT140X.h"        /* Analog-to-digital converter definitions */
#include "node_map.h"      /* C30 node address definitions */
#include "stypes.h"        /* Boolean type definition */
#include "conversion.h"    /* Parameter conversion definitions */
#include "xvme085.h"           /* Encoder interface protoboard defs */

#define NRAM_2_GRAM   16
#define VME_2_NRAM    33
#define NRAM_2_VME    18
```

```
#define GRAM_2_NRAM   21

#define thetam          0.09
#define r_synchronous  thetam * thetam / 3
#define r_asynchronous 0.00645 * 0.00645

#define FIELD_QTY      5   /* Number of fields per sample */
#define NUM_A2D_CHAN   DT1401_A2D_CHANNELS
#define C30_PPINT_REG   0x804402       /* Parallel port interrupt reg. */


short   *mbox = (short *) 0xe0;   /* Mail boxes */
float   *parm_ptr = (float *)  0xc000c0;  /* Parameter passing mailboxes */
float   *acc_ptr = (float *)  0xc03000; /* Acceleration data storage */
float   *est_ptr = (float *) 0xc06000; /* Estimated data storage */
float   *actual_ptr = (float *) 0xc09000; /* Actual data storage */

extern float N2V_linear_conv();   /* Speed (RPM) to voltage conversion */
   float Namplitude = 25.0, Vamplitude;
   float Nbias = 25.0, Vbias;

float theta = 0, freq = 1.0;  /* Angle and angular frequency of function */

extern int sine_gen();
extern int square_gen();
   float rise = 25.0;     /* Generates triangle wave form */
   float fall = 25.0;
void c_int04();
void c_int09();
void c_int10();

DT140XChan
  str_ch = DT140X_CH3,
  end_ch = DT140X_CH6;

int   numchannels = FIELD_QTY - 1;       /* number of a2d channels */
int   a2d_samplecount = 0, actual_samplecount = 0;
float sample_period = 0.001;
int   order = 3;
float q = 1000.0;

float  X_a_priori[3];
float  X_update[3];
float  Gain[3];
float  U_a_priori[3][3];
float  U_update[3][3];
float  A[3][3];
float  Q[3][3];

Bool   sync_flag = False;
float  a2dsamples[NUM_A2D_CHAN];
float  y,  y_old = 0, measure, offset = 0;
short  pcsample;
short  pcsample_old = 0;

/* ****************************************************************** */
void main()
{
Bool           Samples_Ready = True;
int            pause,delay, numsamples;

/* matrices initialization */
A_update(A, (double) sample_period);
Q_update(Q, (double) sample_period, q);
UD_factorization(Q, U_a_priori);
Q_update(Q, (double) sample_period, q);
X_a_priori[0] = 0.0;
X_a_priori[1] = 0.0;
X_a_priori[2] = 0.0;

copy(parm_ptr+3, &numsamples, 1, 0x0000, GRAM_2_NRAM);  /* GRAM -> NRAM */

/* Initialize the hardware */
xvme085_init();  /* Prototype board encoder interface initialized */
init_c30v_bus();        /* C30 node and buses initialized */
enable_ppint(1);        /* C30 parallel port interrupts enabled */
c30_led_on();           /* Port configuration reset, mode 0 */
load_vector(c_int04, C30_INT3_VECLOC);  /* CPU interrupt level 0x04 vector */
enable_ie_int3();       /* CPU external interrupt #3 enabled */

t0_schedule_intr(c_int09, sample_period); /* CPU interrupt level 0x9 vector */

t1_schedule_intr(c_int10, sample_period); /* CPU interrupt level 0xa vector */
```

```
DT140XInit(str_ch, end_ch); /* setup a2d converter; sample rate & mapping *

/* Wait for parallel port interrupts, etc. */
for(pause=0; pause<0xfff; pause++);   /* Allow for mech. time constant */

t0_start();    /* Start the sample period */

t1_start();    /* Start the sample period */

enable_intr();
while(a2d_samplecount <  numsamples)
  {
  }
disable_intr();

copy(&Samples_Ready, (parm_ptr + 1), 1, 0x0000, NRAM_2_GRAM);  /* NRAM->GRAM */
copy(&actual_samplecount, (parm_ptr + 6),1,0x0000,NRAM_2_GRAM);/* NRAM->GRAM */

c30_led_sflash();  /* Slow flashing LED annunciates program end */
}


/* External INT3 handler for parallel port interrupts **********************/
void c_int04()
{
int    *t0_counter_reg = (int *) T0_CNTR_REG;
int    timecnt, i, channel;
double T;
float  r = r_asynchronous;

/* if a pulse arrives */
if ( sync_flag == False )
  {
    /* read the interpulse time */
    t0_stop();
    timecnt = *t0_counter_reg;
    t0_start();

    /* Time counter conversion */
    T = ((double) (timecnt * 256.0 * 16777215.0 * 120.0E-9 / (unsigned long)0xffffff

    /* Read the analog command and angular velocity */
    DT1401Read(str_ch, str_ch+1, a2dsamples);

    /*  Read prototype board pulse counter latch contents from XVME-085 and
        transfer to Node RAM  */
    copy(PB_LAT_RD_ADDR, &pcsample, 1, 0xca00, VME_2_NRAM);
  }

/* if no pulse arrives */
else
  {
    T = (double) (sample_period);
    /* position does not change */
    pcsample = pcsample_old;
    r = r_synchronous;
    sync_flag = False;   /* reset the flag */
  }

/* Process and store the pulse count data */
pcsample &= 0x0000ffff;     /* Avoid hi/lo duplication of short */
y = ((float) (pcsample * 65535.0 )/(float) 0xffff);
y = y * thetam;

/* Avoid encoder reset effects */
if ( ((y_old > 2) && (y < 0.27)) || ((y_old < -300) && (y > -0.27)))
  offset = y_old;
if ((y == 0) && (offset != 0))
  y = offset + 0.09;
else
  y += offset;

/* Store the actual position */
copy(&y, actual_ptr++, 1, 0x0000, NRAM_2_GRAM);  /* NRAM->GRAM */

/* Store the interpulse time data */
copy(&T, actual_ptr++, 1, 0x0000, NRAM_2_GRAM);    /* NRAM->GRAM */

/* Perform the Kalman gain computation loop */
UD_loop_pulse_computation(X_a_priori, U_a_priori, y, r, X_update, U_update, Gain, T,

/* Store the a2d data in GRAM */
for(channel = str_ch; channel < str_ch + 2; channel++)
  copy(&a2dsamples[channel], actual_ptr++, 1, 0x0000, NRAM_2_GRAM);
```

```
/* Store the estimated values in est_ptr */
for(i=0; i<order; i++)
  {
    *est_ptr = X_update[i];
    est_ptr++;
  }

pcsample_old = pcsample;
y_old = y;

enable_ppint(1);       /* Re-enable the external interrupts */
clear_if_int3();       /* Reset the external interrupt  #3 flag bit */

actual_samplecount++;
enable_intr();

} /* c_int04() */


/* Timer 0 ETINT0 handler **************************************************/
void c_int09()
{
int    timecnt, i, channel;
double T;
int    *c30_ppint_reg = (int *) C30_PPINT_REG;


DT1401Read(str_ch, str_ch+1, a2dsamples);

set_if_int3();
sync_flag = True;

enable_intr();
}

/* Timer 1 ETINT1 handler **************************************************/
void c_int10()
{
float a2dacceleration[NUM_A2D_CHAN];

/* Read channels 5 and 6, although only need 5 */
```

```
DT1401Read(str_ch+2, str_ch+2, a2dacceleration);
copy(&a2dacceleration[str_ch+2], acc_ptr++, 1, 0x0000,16);

a2d_samplecount++;

clear_if_int3();       /* Reset the external interrupt  #3 flag bit */

enable_intr();
}
```

Appendix C

load-dac-kalman2.c

```
/*--------------------------------------------------------------------------
 *   FILE: load_dac_kalman2.c
 *
 *   Calling Sequence: Main program.
 *
 *   Function: This is a loader program that loads both nodes with programs.
 *             Node 1 issues the speed reference to the dac.
 *             Node 2 and the parallel port of Node 2 are used for
 *             interrupt-driven acquisition.
 *
 *
 *
 *   Implementation Details:   IEEE/TI floating-point conversion is made
 *             using host-based functions from ti2ie3cnv.a, according to
 *             the code of static int parse_func_arguments() from kali_unix.c
 *             located in fantasio/kalix/dsp/src/target.
 *               Data is transferred to/from the C30 nodes via global memory.
 *
  *-------------------------------------------------------------------------*/
#include <sys/errno.h>
#include <stdio.h>   /* contains declaration of scanf()  */
#include <stdlib.h>  /* declaration of atof(); "extern double atof() */
#include <malloc.h>
```

```
#include "skyc30v.h"
#include "stypes.h"
#include "ti2ie3cnv.h"

#define NODE1       1                         /* for readability */
#define NODE2       2                         /* for readability */
#define CARD        0                         /* the Challenger card number */
#define GLOBAL      0                         /* for GRAM */

#define NODE1FILE        "dac.out"               /* C30 program:  speed ref */
#define NODE2FILE        "kalman2.out"           /* C30 program: no speed ref */
#define GMEM_BASE        0xc00000             /* Global memory */
#define GMEM_OFFSET_PP   0xc0                 /* C30 Parameter Passing space  */
#define GMEM_OFFSET_accSD   0x03000           /* C30 Sample Data space        */
#define GMEM_OFFSET_estSD   0x06000           /* C30 Sample Data space        */
#define GMEM_OFFSET_actSD   0x09000           /* C30 Sample Data space        */

/* #define NUMSAMPLES  4000 */                /* Number of sample data */
#define FIELD_QTY   5                         /* Number of fields per sample */
/* #define ARRAY_SIZE  ( (NUMSAMPLES+1) * FIELD_QTY ) */

/* long samples[ARRAY_SIZE]; */
long *acc_samples, *est_samples, *act_samples;

extern TiFloat ie3_to_ti();
extern float ti_to_ie3();
SKYC30_device  *sky_open_card();
char *sky_filename();
/* ------------------------------------------------------------------------ */
main()
{
    unsigned long  i, timecnt, numsamples, wavetype, actual_samplecount;
    unsigned long  time,pulse;
    int            dummy, delay, status, channel, numchannels = 4;
    BIT_32         *addr;           /* C30 base address */

    char           fname1[80];
    char           inputvar1[80];
    FILE           *out_file1, *out_file2, *out_file3;
    short          a2dvalue;
    float          Nref, a2d_dec;
    Bool           Samples_Ready = False;
    TiFloat         tmpf_Ti, Nref_Ti;
    char            *src, *dst;
    SKYC30_device   *device;  /* pointer to the hardware */


/*  Acquire the card and make sure that the card is not doing anything.   */
    out_file1 = fopen("acceleration.dat","w");
    out_file2 = fopen("actual.dat","w");
    out_file3 = fopen("estimated.dat","w");
    device = sky_open_card(0);      /* assign the card */
    if ( device == NULL )
      {
        perror("sky_open_card: ");
        exit(-1);
      }
    sky_nodes_off(0);               /* turn it off */

/* Load the C30 test program into node 1.  */
    /* printf(" Filename to load in node 1  :  ");
    scanf("%s",fname1); */
    /* status = sky_ld_file (0, NODE1, fname1); */
    status = sky_ld_file (0, NODE1, NODE1FILE);
    if (status != 0)
        {
        printf ("Node 1, sky_ld_file() for file '%s' failed; status = %d\n",
                        fname1, status);
        exit (-1);
        }

/*  Load the C30 test program into node 2.    */
    status = sky_ld_file (0, NODE2, NODE2FILE);
    if (status != 0)
        {
        printf ("Node 2, sky_ld_file() for file '%s' failed; status = %d\n",
                        NODE2FILE, status);
        exit (-1);
        }

/*  Parameter passing via C30 global memory.  */
    addr = device->mem + GMEM_OFFSET_PP;
    printf("      Enter speed reference bias Nbias (RPM):  ");
    scanf("%s",inputvar1);
    Nref = atof(inputvar1);
    tmpf_Ti = ie3_to_ti( &Nref );
          src = (char *) &tmpf_Ti;
          dst = (char *) &Nref_Ti;   /* Byte-wise copy of the bit pattern */
      *dst++ = *src++; *dst++ = *src++; *dst++ = *src++; *dst++ = *src++;
    status = sky_wr_mem (CARD, GLOBAL, addr, 1, &Nref_Ti);
   if (status != 0)
        {
        printf ("sky_wr_mem() failed; status = %d\n", status);
        exit (-1);
        }

/*  Parameter passing via C30 global memory.  */
    addr = device->mem + GMEM_OFFSET_PP + 1;
       status = sky_wr_mem (CARD, GLOBAL, addr, 1, &Samples_Ready);
    if (status != 0)
        {
        printf ("sky_wr_mem() failed; status = %d\n", status);
        exit (-1);
        }

/*  Parameter passing via C30 global memory.  */
    addr = device->mem + GMEM_OFFSET_PP + 3;
    printf("      Enter the number of samples:  ");
    scanf("%s",inputvar1);
    numsamples = atoi(inputvar1);
    status = sky_wr_mem (CARD, GLOBAL, addr, 1, &numsamples);
   if (status != 0)
        {
        printf ("sky_wr_mem() failed; status = %d\n", status);
        exit (-1);
        }

/*  Parameter passing via C30 global memory.  */
    addr = device->mem + GMEM_OFFSET_PP + 4;
    printf("      Enter speed reference amplitude Namplitude (RPM):  ");
    scanf("%s",inputvar1);
    Nref = atof(inputvar1);
    tmpf_Ti = ie3_to_ti( &Nref );
          src = (char *) &tmpf_Ti;
          dst = (char *) &Nref_Ti;
      *dst++ = *src++; *dst++ = *src++; *dst++ = *src++; *dst++ = *src++;
    status = sky_wr_mem (CARD, GLOBAL, addr, 1, &Nref_Ti);
   if (status != 0)
        {
        printf ("sky_wr_mem() failed; status = %d\n", status);
        exit (-1);
        }

/*  Parameter passing via C30 global memory. */
    addr = device->mem + GMEM_OFFSET_PP + 5;
    printf("     Enter the waveform - sine(1), square(2), triangle(3):  ");
    scanf("%s",inputvar1);
    wavetype = atof(inputvar1);
    status = sky_wr_mem (CARD, GLOBAL, addr, 1, &wavetype);
   if (status != 0)
        {
        printf ("sky_wr_mem() failed; status = %d\n", status);
        exit (-1);
        }

/*  Parameter passing via C30 global memory.  */
    addr = device->mem + GMEM_OFFSET_PP + 6;
    status = sky_wr_mem(CARD, GLOBAL, addr, 1, &actual_samplecount);
   if (status != 0)
        {
        printf ("sky_wr_mem() failed; status = %d\n", status);
        exit (-1);
        }

/* Run the C30 programs.  */
    sky_reset_go(0);
    addr = device->mem + GMEM_OFFSET_PP + 1;
      while( Samples_Ready == False ){     /* Samples ready ???? */
        printf ("    Waiting for Samples_Ready flag...    ");
        for (delay=0; delay< (numsamples * 800) ; delay++);
    status = sky_rd_mem(CARD,GLOBAL,addr,1,&Samples_Ready);
     if (status != 0)
       {
         printf("sky_rd_mem failed \n");
         exit (-1);
       }
    }
     printf("   Sample data upload to host in progress...   ");
```

```
/*  Parameter fetching from C30 global memory. */
    addr = device->mem + GMEM_OFFSET_PP + 6;
    status = sky_rd_mem(CARD,GLOBAL,addr,1,&actual_samplecount);
     if (status != 0)
       {
         printf("sky_rd_mem failed \n");
         exit (-1);
       }
    printf("actual_samplecount = %d\n",actual_samplecount);
    act_samples = (long *) malloc(  (4 * actual_samplecount) * sizeof(long) );
    est_samples = (long *) malloc(  (3 * actual_samplecount) * sizeof(long) );
    acc_samples = (long *) malloc(  numsamples * sizeof(long) );

/*  Sample data fetching from C30 global memory. */
    addr = device->mem + GMEM_OFFSET_accSD;
    status = sky_rd_mem(CARD,GLOBAL,addr,numsamples,acc_samples);
     if (status != 0)
       {
         printf("sky_rd_mem of acc_samples failed \n");
         exit (-1);
       }

/*  Sample data fetching from C30 global memory. */
    addr = device->mem + GMEM_OFFSET_actSD;
    status = sky_rd_mem(CARD,GLOBAL,addr,(4*actual_samplecount),act_samples);
    if (status != 0)
       {
         printf("sky_rd_mem of act_samples failed \n");
         exit (-1);
       }

/*  Sample data fetching from C30 global memory. */
    addr = device->mem + GMEM_OFFSET_estSD;
    status = sky_rd_mem(CARD,GLOBAL,addr,(3*actual_samplecount),est_samples);
    if (status != 0)
       {
         printf("sky_rd_mem of est_samples failed \n");
         exit (-1);
       }

/* Fetch the C30 sample data and write to output */
```

```
   for (i=0; i< numsamples; i++){
       fprintf(out_file1,"%12.8f\n", ti_to_ie3( acc_samples++ )  );
     }

   for (i=0; i< actual_samplecount; i++) {
       /* pulse = (float) *act_samples; act_samples++; */
       fprintf(out_file2,"%12.4f\n", ti_to_ie3( act_samples++ ));
       /* time = (float) *act_samples; act_samples++;  */
       fprintf(out_file2,"%12.8f\n", ti_to_ie3( act_samples++ ));
       for(channel=0;channel<2;channel++) {
         fprintf(out_file2,"%12.8f\n", ti_to_ie3(act_samples));
         act_samples++;
       }
}

   for (i=0; i< actual_samplecount; i++) {
       for(channel=0;channel<3;channel++) {
         fprintf(out_file3,"%12.8f\n", ti_to_ie3(est_samples));
         est_samples++;
       }
       fprintf(out_file3,"\n");
}

    fclose(out_file1);
    fclose(out_file2);
    fclose(out_file3);
}

/* --------------------- End of Module ------------------------------- */
```