

IMPLEMENTATION OF A STATE SPACE KALMAN
FILTER ON A DIGITAL SIGNAL PROCESSING
MICROPROCESSOR

A. THESIS
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL
AND ELECTRONICS ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND
SCIENCES OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
M. Khaleedul Islam
August 1990

IMPLEMENTATION OF A STATE-SPACE KALMAN FILTER ON A DIGITAL SIGNAL PROCESSING MICROPROCESSOR

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL AND
ELECTRONICS ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCES
OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF
MASTER OF SCIENCE

M. Khaledul Islam
তারিখ: ১৫/০৮/৯০

By

M. Khaledul Islam

August 1990

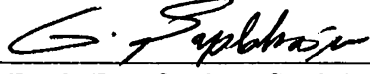
TK
7872
F5
182
1990

B. 1992

To my mom who taught me

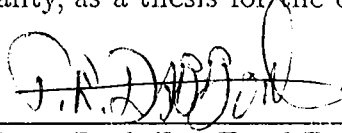
अ, आ and A,B,C

I certify that I have read this thesis and that in my opinion it is fully adequate,
in scope and in quality, as a thesis for the degree of Master of Science.



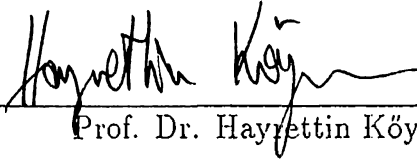
Asst. Prof. Dr. Gürhan Şaplan (Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate,
in scope and in quality, as a thesis for the degree of Master of Science.



Asst. Prof. Dr. Tayel E. Dabbous

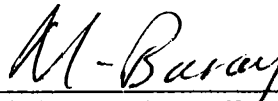
I certify that I have read this thesis and that in my opinion it is fully adequate,
in scope and in quality, as a thesis for the degree of Master of Science.



Prof. Dr. Hayrettin Köymen

Bilkent Univ.
Library

Approved for the Institute of Engineering and Sciences:



Prof. Dr. Mehmet Baray
Director of Institute of Engineering and Sciences

ABSTRACT

IMPLEMENTATION OF A STATE-SPACE KALMAN FILTER ON A DIGITAL SIGNAL PROCESSING MICROPROCESSOR

M. Khaledul Islam

M.S. in Electrical and Electronics Engineering

Supervisor: Asst. Prof. Dr. Gürhan Şaplakoğlu

August 1990

A general software written in assembly language for the real-time implementation of state-space Kalman filter on Texas Instruments TMS320C25 fixed-point digital signal processor is given. The software can accomodate dynamic system having up to 14 state variables. As a specific application, the Kalman filter is used to restore the sound of a flute embedded in white noise.

ÖZET

DURUM UZAYI KALMAN FİLTRESİNİN BİR SAYISAL İŞARET İŞLEYİCİSİNDE GERÇEKLEŞTİRİLMESİ

M. Khaledul Islam

Elektrik ve Elektronik Mühendisliği Bölümü Yüksek Lisans

Tez Yöneticisi: Yrd. Doç. Dr. Gürhan Şaplakoğlu

Ağustos 1990

Durum uzayı Kalman filtresinin gerçek zamanda, Texas Instruments'in TMS320-C25 sabit noktalı sayısal mikroişlemcisi ile gerçekleştirilebilmesi için simgesel dilde yazılmış genel bir yazılım verilmiştir. Yazılım durum değişkeni sayısı en çok 14 olan dinamik sistemlere uygulanabilir. Kalman filtre, özel bir uygulama olarak beyaz gürültüyle bozulmuş flüt sesinin yeniden elde edilmesi için kullanılmıştır.

ACKNOWLEDGMENT

I would like to express deep gratitude to my thesis supervisor Dr. Gürhan Şaplakoğlu for his invaluable guidance, stimulating inspirations and continuing help. I am also indebted to him for introducing me to the fascinating world of real-time signal processing. A special thank you goes to Mrs. Umay Şaplakoğlu who by playing flute played a vital role in our thesis progress. And last but not the least, I wish to acknowledge the assistance of Ergin Atalar who literally rescued me when I got lost in the schematic diagrams of SWDS board while making interfacing between 8088 and TMS320C25.

Contents

1	INTRODUCTION	1
1.1	The State-Space Kalman Filter	1
1.2	TMS 320 Digital Signal Processor Family	2
1.3	Why Fixed-Point Processor ?	3
1.4	Real-time Kalman Filter on a Fixed-point Digital Signal Processor	4
2	A REVIEW OF KALMAN FILTER ALGORITHMS	6
2.1	The Kalman Filter	6
2.1.1	Notational Convention	6
2.1.2	The Algorithms	7
2.2	Practical problems in real-time Kalman filtering	10
2.2.1	Roundoff Errors	10
2.2.2	Model Mismatch	11
2.2.3	Observability Problems	11
2.3	Tuning of Kalman Filter	12
3	A GENERAL SOFTWARE FOR KALMAN FILTER	13
3.1	A Software for Kalman Filter : An Introduction	13
3.2	The Storage Strategy Used in the Software	14

3.3	The Macro Library	15
3.3.1	General Matrix Macros	15
3.3.2	Some Special Macros	20
3.4	Building up the Kalman Filter Algorithms	21
3.4.1	Scalar Observation	23
3.4.2	Vector Observation	27
3.5	Performance Evaluation of Different Implementations	31
3.5.1	Scalar Observation	31
3.5.2	Vector Observation	34
4	RESTORATION OF FLUTE IN WHITE NOISE	38
4.1	Flute from an Engineering Perspective	38
4.2	State-space Modeling of Flute	39
4.3	Using Kalman Filter to Recover the Sound of Flute	40
4.4	Event Detection	43
4.4.1	Windowed-normalized-residue(WNR)-based detection	43
4.4.2	Windowed-fundamental-state(WFS)-based detection	46
4.5	Physical Insights about the Operation of the Kalman filter	49
5	CONCLUSION	55
A	Derivation of Fading Memory Filter	57
B	Fundamental Frequencies of Flute Notes	59
C	State-space model of a sinusoid	61

<i>CONTENTS</i>	viii
D Interfacing Between 8088 and SWDS board	63
E Role of Filter Gain on Bandwidth	65
F The Macro Library	67
F.1 General Macros . .	67
F.2 Special Macros	88

List of Figures

1.1	Kalman Filter	2
3.1	Compact Storage Scheme of Example 1	14
3.2	Storage Scheme for LTA and MPY pair option	24
3.3	Storage Scheme for MAC option	26
3.4	Storage Scheme for Sequential Processing, $N = 8$	27
3.5	Storage Scheme for Batch Processing, $N = 8$	29
3.6	Program Memory Words Vs. Dimension of State Vector in Scalar Observation	33
3.7	Max. Sampling Frequency Vs. Dimension of State Vec- tor in Scalar Observation	33
3.8	Program Memory Words Vs. Dimension of Observation Vector in Vector Observation	35
3.9	Max. Sampling Frequency Vs. Dimension of Observa- tion Vector in Vector Observation	35
3.10	Kalman Gain Computation Time Vs. Dimension of Ob- servation Vector	36
3.11	Comparison of LU and Choleski Decomposition	37
4.1	The Overall Setup for Kalman Filter Implementation . .	41
4.2	Flow-chart of 8088 and TMS Programs Running Simul- taneously	42

4.3	Filter Performance in Real-time Operation : the top-most waveform is observation followed by filtered output and actual note	44
4.4	Filter Performance in Real-time Operation : the top-most waveform is observation followed by filtered output and actual note	45
4.5	Performance of WNR and WFS methods (High SNR case: $Q = 0.002, R = 0.16, \epsilon = 1.02$)	47
4.6	Performance of WNR and WFS methods (Low SNR case: $Q = 0.002, R = 0.16, \epsilon = 1.02$)	48
4.7	Mesh Diagram of bandwidth vs. κ_1 and κ_2 , (for $-0.4 \leq \kappa_1, \kappa_2 \leq 0.4$ and $\alpha = 0.809, \beta = 0.588$)	50
4.8	Effect of \hat{Q} , \hat{R} and ϵ on \hat{P} and \bar{k} . a) Effect of SNR for fixed \hat{Q} , \hat{R} , ϵ , b) Effect of increasing \hat{Q} , c) Effect of increasing ϵ	52
4.9	Effect of \hat{R} on the Transient Response of the Filter Output	53
4.10	Effect of \hat{Q} on the Transient Response of the Filter Output	54
4.11	Effect of ϵ on the Transient Response of the Filter Output	54

List of Tables

3.1	LTA and MPY pair option	32
3.2	MAC option	32
3.3	Sequential Processing for $N = 8, 2 \leq M \leq 8$	34
3.4	Batch Processing for $N = 8, 2 \leq M \leq 8$	34
4.1	Number of Harmonics of Flute Notes	39
D.1	Addresses of Shared Memory as Seen by TMS and 8088	63

Chapter 1

INTRODUCTION

A brief introduction to the state-space Kalman filter and the digital signal processors particularly Texas Instruments TMS320C25 is given in the first two sections of this chapter. Section three discusses the choice of fixed-point arithmetic in real-time signal processing. The motivations behind choosing a digital signal processor for the implementation of real-time fixed-point Kalman filter are described in section four.

1.1 The State-Space Kalman Filter

A discrete time Kalman filter is a recursive algorithm that calculates the linear, unbiased, minimum mean squared estimate of the state of a dynamic system from noise-corrupted observation data. The algorithm also allows random perturbations in the state evolution of the system. If the state perturbation noise and the measurement noise are uncorrelated and Gaussian, then the filter provides the best performance among all the estimators in mean squared sense. A very simple pictorial representation of the filter is given in Figure 1.1. Although the name *filter* sounds like a misnomer, it is the universally accepted term to describe the recursive algorithm that R. E. Kalman proposed back in 1960 [1]. The Kalman filter can be applied to any system that has a dynamic state-space representation. Apart from the state estimation of the system, it can also be used for system identification and deconvolution [2]. The wide spectrum of applications span as diverse fields as the space-craft orbit determination [3] and the demographic of cattle production [4]. Comprehensive treatment of the Kalman filter and its applications can be found in [5], [6], [7], [8] .

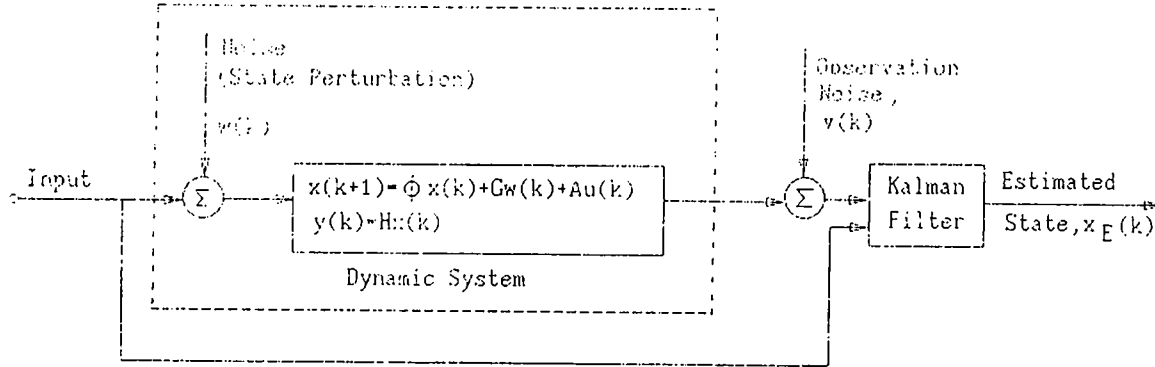


Figure 1.1: Kalman Filter

1.2 TMS 320 Digital Signal Processor Family

Since Intel introduced the 2920 in 1979, the first microprocessor specifically tailored for digital signal processing applications, various VLSI digital signal processor chips have been launched by Texas Instruments, NEC, AT & T, Motorola and many others [9]. Among these, Texas Instruments TMS320 family is one of the most widely used digital signal processors due to its relatively low cost, powerful instruction set, inherent flexibility and comprehensive hardware support. The family consists of three generations of fixed and floating point digital signal processors. The second generation, which is considered in next chapters, comprises of five 16-bit microprocessors - namely TMS32020, TMS320C25, TMS320C25-50, TMS320E25 and TMS320C26.

Compared with conventional microprocessors, the most striking feature in TMS320 family is the use of parallelism and pipelining to enhance execution speed. It uses Harvard-type architecture which separates program and data memory spaces, eliminating the throughput bottleneck associated with the shared-bus structure of general-purpose microprocessors. This structure enables data fetching concurrent with the fetching of next program instruction making the program execution faster. Another important difference is the fast fixed-point multiplication unit available in TMS320 family. As a matter of fact, the multiplier unit occupies most of the space (up to 40% in TMS32020) on the chip. Some of the key features of the TMS320C25, the most widely used member of the second generation which is considered in the subsequent chapters, can be summarized as [10] :

- 16 bit fixed-point operation with some provision for floating-point operation.
- 100 nanoseconds instruction cycle time which makes the microprocessor capable of executing 10 million instructions per second(MIPS).
- single cycle multiply/accumulate instruction with data move option that makes the digital filter realization very efficient.
- 544 words of on-chip RAM.
- 4K words of ROM that makes it a true single chip microprocessor.
- total 64K words of program memory space and 64K words of data memory space.
- 32 bit dedicated Central Logic Unit(CALU)
- 8 auxiliary registers with an Auxiliary Register Arithmetic Unit(ARAU) that operates in parallel with CALU
- sixteen input and sixteen output ports

1.3 Why Fixed-Point Processor ?

All the members of the second generation TMS320 family perform 16-bit fixed-point arithmetic. The fixed-point arithmetic is based on the assumption that the location of the binary point is fixed. The Q notation is commonly used to specify the location of the binary point. A binary number in Qn format is defined as having n bits to the right of binary point. For example in sign-extension mode, the maximum and minimum numerical values represented by $Q15$ format are hexadecimal 8000 and 7FFF or equivalently decimal -1 and $(1 - 2^{-15})$ respectively. To get the Qn representation of a fractional number, the first step is to multiply it by 2^n and round the result to an integer. The 2's complement hexadecimal representation of the integer is the Qn equivalent of the corresponding fractional number.

Despite the fact that the fixed-point arithmetic has a limited dynamic range as compared to floating-point arithmetic, it does have some appreciable advantages in real-time applications. First of all, the fixed-point digital signal processors are much faster and cheaper than their floating-point counterparts. In floating-point arithmetic, errors due to arithmetic roundoff are introduced both

in addition and multiplication whereas in fixed-point arithmetic such errors occur only in multiplication. The major disadvantage of a fixed-point processor is the possibility of overflow. However, this problem can be overcome by using appropriate scaling. To be more specific, the program can be written in such a way that the occurrence of overflow will depend solely on input samples. The necessary scaling can be determined by simulating the system on a floating-point computer before real-time operation. Although the scaling may cause some loss in numerical accuracy, it is not usually significant. Another important advantage of fixed-point arithmetic is the compatibility of the numerical representation used in the fixed-point digital signal processors and the analog interfacing devices. Most of the A/D and D/A converters available in the market use either 2's complement or offset binary format to represent numerical values. This representation is very convenient when fixed-point arithmetic is used in $Q15$ format, because the input or output samples can be interpreted as $Q15$ representations of voltages normalized to the peak magnitude of converters. The fact that the TMS320C5X, the 5th generation of the TMS320 family to be launched in late 1990, will be again a fixed-point processor reflects the preference of fixed-point arithmetic to floating-point arithmetic in real-time signal processing applications [11].

1.4 Real-time Kalman Filter on a Fixed-point Digital Signal Processor

Implementation of a Kalman filter involves heavy computational complexities. As far as conventional computers are concerned, the largest amount of program execution time is taken by multiplications. And to make things worse, the number of multiplications in Kalman filtering is proportional to the third power of the state size of the system [6]. Hence running a state-space Kalman filter on-line was not a realistic possibility until recently. Most of the real-time applications reported so far are applied to the systems that do not require fast execution times like navigation. The introduction of digital signal processors, which have very fast multiply/accumulate instructions, has opened a new era of real-time Kalman filter applications.

The TMS320 family is an ideal cost-effective choice for implementing real-time Kalman filters. Implementation of a simple two-state tracking Kalman filter on TMS32010, the first generation of TMS320 family, was reported [12]. Recently, implementation of a narrow-band Kalman filter on AT & T DSP32

floating point processor has been published [13].

In this thesis, a general software written in TI assembly language version 5.0, is introduced to implement real-time Kalman filter on TMS320C25. The software consists of a collection of matrix manipulation macros, consequently can be modified to accommodate different Kalman filter algorithms. Efforts have been made to make an optimal trade-off between user-friendliness and execution speed of the software. As a matter of fact, our software is much more efficient in terms of speed as compared to [13] although DSP32 runs at a higher clock rate. Use of the software is thoroughly explained in chapter 3 along with illustrative examples. This chapter is preceded by chapter 2 which contains a review of Kalman filter algorithms and a discussion of the problems encountered in the implementation of the filter. As a specific application, real-time implementation of state-space Kalman filter to recover the sound of a flute embedded in white noise is described in chapter 4. Based on simulation results, relationship between various filter parameters and their effects on filter gain and bandwidth are investigated.

Chapter 2

A REVIEW OF THE KALMAN FILTER ALGORITHMS

In this chapter, different Kalman filter algorithms are reviewed. A variety of problems arise in real-time implementation of Kalman filter. Some of the most likely problems are discussed in section two . Several criteria which can be used as tests for performance evaluation of the filter are described in section three.

2.1 The Kalman Filter

Since R. E. Kalman published his landmark paper in 1960, it arouse great interest among researchers. The material is now well covered in literature [5],[6],[7],[8],[14],[15],[16]. A number of algorithms have been proposed as an alternative to the original Kalman filter algorithm. These algorithms make a trade-off between the numerical stability and the computational requirements. In the following sub-sections, some of the widely used algorithms are reviewed.

2.1.1 Notational Convention

The notational conventions used in the subsequent sections as well as in the following chapters are summarized below :

- vectors are lower-case letters with bars.
- matrices are upper-case letters with hats.
- $[\cdot]^T$ denotes transposition.

- $[\cdot]^{-1}$ denotes inversion.
- $\bar{x}(j \mid k)$ is the estimate of $\bar{x}(j)$ given the observation sequence $\{\bar{y}(n) : n = 0, 1 \dots k\}$.
- $E\{\cdot\}$ is the expectation operator.

2.1.2 The Algorithms

The Kalman filter requires that the relationship between the process to be estimated and the observation must be of the following state-space form :

$$\bar{x}(k+1) = \hat{\Phi}(k)\bar{x}(k) + \hat{G}(k)\bar{w}(k), \quad (2.1)$$

$$\bar{y}(k) = \hat{H}(k)\bar{x}(k) + \bar{v}(k), \quad (2.2)$$

where

$\bar{x}(k) = N \times 1$ state vector at time t_k ,

$\hat{\Phi}(k) = N \times N$ state-transition matrix at time t_k ,

$\hat{G}(k) = N \times L$ process noise matrix at time t_k ,

$\bar{w}(k) = L \times 1$ process noise vector at time t_k with known covariance matrix $\hat{Q}(k)$,

$\bar{y}(k) = M \times 1$ observation vector at time t_k ,

$\hat{H}(k) = (M \times N)$ observation matrix at time t_k ,

$\bar{v}(k) = M \times 1$ observation noise vector at time t_k with known covariance matrix $\hat{R}(k)$.

Furthermore, the noise vectors are assumed to be white and uncorrelated with each other i. e. ,

$$E\{\bar{w}(i)\bar{w}^T(j)\} = \begin{cases} \hat{Q}_i & \text{if, } i = j \\ 0 & \text{otherwise} \end{cases} \quad (2.3)$$

$$E\{\bar{v}(i)\bar{v}^T(j)\} = \begin{cases} \hat{R}_i & \text{if, } i = j \\ 0 & \text{otherwise} \end{cases} \quad (2.4)$$

$E\{\bar{w}(i)\bar{v}^T(j)\} = 0$ for $\forall i$ and j , $E\{\bar{x}(0 \mid 0)\bar{w}^T(i)\} = 0$ for $\forall i$, $E\{\bar{x}(0 \mid 0)\bar{v}^T(i)\} = 0$ for $\forall i$.

With these assumptions and notational conventions, some of the widely used Kalman filter algorithms are described in the subsequent subsections.

Standard Kalman Filter

This is probably the most widely used form of the Kalman filter algorithm and is essentially the one that Kalman derived in 1960. It can be described in the prediction-correction form as :

- Step 1 : Initialization - $k = 0$; Input $\bar{x}(0 | 0), \hat{P}(0 | 0)$

- Step 2 : Prediction -

$$\text{state prediction : } \bar{x}(k+1 | k) = \hat{\Phi}(k)\bar{x}(k | k) \quad (2.5)$$

covariance prediction :

$$\hat{P}(k+1 | k) = \hat{\Phi}(k)\hat{P}(k | k)\hat{\Phi}^T(k) + \hat{G}(k)\hat{Q}(k)\hat{G}^T(k) \quad (2.6)$$

- Step 3 : Measurement - read $\bar{y}(k+1)$

- Step 4 : Innovation -
innovation sequence :

$$\bar{v}(k+1) = \bar{y}(k+1) - \hat{H}(k+1)\bar{x}(k+1 | k) \quad (2.7)$$

innovation covariance :

$$\hat{C}(k+1) = \hat{H}(k+1)\hat{P}(k+1 | k)\hat{H}^T(k+1) + \hat{R}(k+1) \quad (2.8)$$

- Step 5 : Computation of Kalman gain -

$$\text{Kalman gain : } \hat{K}(k+1) = \hat{P}(k+1 | k)\hat{H}^T(k+1)\hat{C}^{-1}(k+1) \quad (2.9)$$

- Step 6 : Correction -
state correction :

$$\begin{aligned} \bar{x}(k+1 | k+1) &= [I - \hat{K}(k+1)\hat{H}(k+1)]\bar{x}(k+1 | k) + \\ &\quad \hat{K}(k+1)\bar{y}(k+1) \end{aligned} \quad (2.10)$$

covariance correction :

$$\hat{P}(k+1 | k+1) = [I - \hat{K}(k+1)\hat{H}(k+1)]\hat{P}(k+1 | k) \quad (2.11)$$

- Step 7 : Continuation - $k = k+1$, go to Step 2

A major drawback of this algorithm is the fact that the covariance matrix $\hat{P}(k+1 | k+1)$ is prone to numerical roundoff errors and may lose positive definitiveness resulting in serious errors. Even the symmetry of \hat{P} can be lost in a few iterations as a direct result of fixed-length numerical computations [18].

Stabilized Kalman Filter

This method is superior to the standard Kalman filter since it is less sensitive to numerical roundoff. As expected, the price paid for it is more computational burden. The steps of the algorithm are the same as the standard one except the covariance correction in step 6. In this case, the filtered estimate of the error covariance matrix is computed as,

$$\begin{aligned} \hat{P}(k+1 | k+1) &= \hat{K}(k+1)\hat{R}(k+1)\hat{K}(k+1)^T + \\ &[I - \hat{K}(k+1)\hat{H}(k+1)]\hat{P}(k+1 | k)[I - \hat{K}(k+1)\hat{H}(k+1)]^T \end{aligned} \quad (2.12)$$

This expression of P in the form of sum of two symmetric matrices is called “Joseph form”. Numerical computations based on this form are better conditioned and symmetry as well as positive definitiveness of matrix \hat{P} are preserved [17].

Sequential Kalman Filter

The sequential algorithm refers to the technique of processing the measurement vector $\bar{y}(k)$ one component at a time as opposed to the batch processing where all the elements of the observation vector are treated at the same time. The beauty of this algorithm is that the direct computation of inverse of the innovation covariance matrix \hat{C} is avoided. This results in appreciable amount of computational savings. The sequential algorithm can be applied when the covariance matrix of observation noise, $\hat{R}(k)$ is diagonal. However, this is not a big constraint, since the observation $\bar{y}(k)$ can always be transformed to an uncorrelated process say $\tilde{y}(k)$ with a diagonal covariance matrix $\tilde{R}(k)$ by a nonsingular transformation $\hat{D}(k)$. In doing so, the observation equation ,

$$\bar{y}(k) = \hat{H}\bar{x}(k) + \bar{v}(k) \quad (2.13)$$

becomes,

$$\tilde{y}(k) = \tilde{H}\tilde{x}(k) + \tilde{v}(k) \quad (2.14)$$

where, $\tilde{y}(k) = \hat{D}(k)\bar{y}(k)$, $\tilde{H}(k) = \hat{D}(k)\hat{H}(k)$, $\tilde{v}(k) = \hat{D}(k)\bar{v}(k)$ such that $\mathcal{E}\{\tilde{v}(i)\tilde{v}^T(j)\} = \hat{D}(k)\hat{R}(k)\hat{D}^T(k)$. Details of the algorithm can be found in [14].

Square Root Algorithm

As compared to the previous algorithms, the square root filter is better from an accuracy point of view. The reason is that square-rooting a small number yields a large number and vice versa, thus computations are carried out more precisely. The core of this algorithm is the Choleski decomposition which decomposes a non-negative definite symmetric matrix \hat{A} into the product of a lower triangular matrix \hat{A}^c and its transpose such that,

$$\hat{A} = \hat{A}^c(\hat{A}^c)^T \quad (2.15)$$

If all the components of the measurement vector are treated at the same time as in the standard and the stabilized filter, the inversion of a lower triangular matrix is needed. On the other hand, the sequential processing can be incorporated into the square root algorithm thus making a good compromise between the computational requirements and the numerical accuracy[14].

A comparison between these algorithms in terms of memory requirements and execution speed can be found in [18]. However, the comparison is based on the assumption that the multiplication takes thrice as much time as addition which is the case with general-purpose computers.

2.2 Practical problems in real-time Kalman filtering

Some of the usual problems encountered in real-time Kalman filtering are described in the following subsections.

2.2.1 Roundoff Errors

As with all digital hardware of finite word-length, fixed-point implementation of Kalman filter is not immune from unavoidable roundoff errors. Due to recursive nature of the algorithm, these errors may be quite significant. One of its worst consequences reported by many researchers, is that the covariance matrix \hat{P} may become negative definite due to the accumulation of roundoff errors [16]. This leads to instability of the filter. One solution is to add some process noise i. e. to perturb the evolution of the state by a random disturbance even if the system is known to be deterministic. In doing so, the positive definite \hat{Q} matrix prevents the error covariance matrix \hat{P} from going

negative as evidenced from the prediction step of the algorithm. This imposed uncertainty leads to a degree of suboptimality, but it is better than having the filter diverge. Another solution is to symmetrize $\hat{P}(k+1 | k)$ and $\hat{P}(k+1 | k+1)$ matrices at each iteration step. Since covariance matrix must be symmetric, any form of asymmetry can be attributed to roundoff errors. The symmetry problem is automatically solved if, in implementation symmetry of covariance matrix is assumed and only the lower (or the upper) triangular part is used and updated in all operations. Another way is the square root algorithm which propagates the square root of \hat{P} rather than the \hat{P} itself. Various other methods have been proposed to encounter this problem [19], [20].

2.2.2 Model Mismatch

Inaccuracy in system-modeling can severely deteriorate the filter performance. The errors associated with model mismatch can be minimized to certain extent by adding fictitious process noise. A more effective way to accomplish this is to use fading memory filter which takes into account the gradual change in system parameters by exponentially diminishing the effects of older data on recent calculations [17]. Basically this means that as one moves along time, the strength of corrupting noises in prior iterations are artificially increased before their influence is brought into current estimation of the states. For example, if the conventional Kalman filter provides state estimates at time t_i based on observation noise sequence $R(t_1), R(t_2) \dots R(t_i)$ then the fading memory filter would do so using the exponentially decaying sequence $e^{\sigma_1} R(t_1), e^{\sigma_2} R(t_2) \dots e^{\sigma_i} R(t_i)$. The term e^{σ_i} is called the “forgetting factor” since it determines how heavily the recent data are overweighed. It can be shown that the fading memory has the same steps as conventional Kalman filter with a minor change in the prediction of error covariance matrix $\hat{P}(k+1 | k)$ which in this case is given by,

$$\hat{P}(k+1 | k) = \hat{\Phi}(k) \hat{P}(k | k) \hat{\Phi}^T(k) e^{\sigma_k} + \hat{G}(k) \hat{Q}(k) \hat{G}^T(k). \quad (2.16)$$

For completeness, the derivation of the discrete time fading memory is given in Appendix A following [21].

2.2.3 Observability Problems

This problem arises when the system is not observable i. e. one or more of the state variables (or their linear combinations) are hidden from the view of

the observer [16]. As a result, if the unobserved states are unstable, the corresponding estimations will be unstable. The occurrence of such a problem can be evidenced by the unbounded growing of one or more diagonal entries of error covariance matrix.

2.3 Tuning of Kalman Filter

The estimates of states of a system as provided by Kalman filter is optimal only if the filter is properly “tuned”. A couple of parameters can be checked to ensure that the filter is operating properly. A necessary and sufficient condition for the Kalman filter to be working properly is that the innovation sequence must be zero-mean and white [8]. This can be easily checked in practice. The square roots of the diagonal entries of the covariance matrix \hat{P} represent the root-mean-square(RMS) error associated with the states [2]. If the filter is properly tuned, the diagonals should reach steady-state values irrespective of initial assumption $\hat{P}(0 | 0)$. This arises from the fact that as long as the filter is stable and the state-space model is completely controllable and observable, then \hat{P} attains a steady state value [8]. It should be noted that initial guesses on the state vector $\bar{x}(0 | 0)$ and covariance matrix $\hat{P}(0 | 0)$ are not important in tuning the filter since their effects are reduced as more observation data are processed.

Chapter 3

A GENERAL SOFTWARE FOR FIXED-POINT IMPLEMENTATION OF REAL-TIME KALMAN FILTER

In this chapter, a general software written in TI assembly language for the implementation of fixed-point Kalman filter on TMS320C25 digital signal processor is introduced. Use of macro library for various implementations of the filter algorithms are described and illustrated with examples. A comparison between these approaches is discussed in the last section.

3.1 A Software for Kalman Filter : An Introduction

A very general user-friendly software in the form of macros is written in Texas Instruments(TI) assembly language version 5.0 for TMS320C25 digital signal processor in order to implement real-time fixed-point Kalman filter having as many as 14 states. Any of the filter algorithms discussed in chapter two can be implemented using these macros. As a matter fact, the extensive macro library provides the user with a wide choice implementations. To incorporate the macros in the main program, the user does not have to be an expert of TI assembly language. As illustrated with examples in the next sections, only a little knowledge of TMS assembler directives and its memory configuration is enough to efficiently use the macros in filter algorithm. Since one of the prime concerns in the implementation of real-time Kalman filter is execution speed, macros are preferred to subroutines at the expense of considerable demand on program memory. The macros are written in such a way that they fully exploit the unique architecture of the TMS microprocessor. Some of the steps in the

filter realization are rearranged to make them more suitable to TMS structure. The macros are written with the assumption that all numerical computations are carried out using fixed-point Q15 format.

3.2 The Storage Strategy Used in the Software

A general storage strategy is used for storing scalars, vectors, matrices in all the macros. It is assumed that maximum allowable size of a matrix is 14×14 . For matrices, the entries of a row are stored in consecutive places whereas those of a column are stored hexadecimal 10 (decimal 16) places apart in memory. The only exception to this rule is the storage of diagonal matrices. In this case, the diagonal entries are stored in successive memory locations. When we say that a matrix \hat{A} is stored at "Loc_of_A" in data memory, we simply mean that $A(1,1)$ is stored right at "Loc_of_A" and the remaining entries are stored relative to this location. For example, storing \hat{A} at hexadecimal 211h (denoted as 211h) means that $A(1,1)$ is stored at 211h, $A(1,2)$ at 212h, $A(2,1)$ at 221h and so on. The same storage strategy holds for vectors i. e. elements of a row vector are stored in consecutive places whereas those of a column vector are placed hexadecimal 10 places apart. This storage strategy facilitates efficient use of memory and at the same time enables the user to visualize everything in terms of the usual indexing used in vectors and matrices. Example 1 illustrates how this can be used to store different matrices and vectors in compact form in the memory.

Example 1

Let us suppose that \hat{A} is a 6×8 matrix, \hat{B} is a 7×7 matrix, \hat{D} is a 8×8 diagonal matrix, \bar{e} is a 1×8 row vector and \bar{f} is a 8×1 column vector. Now if the available memory starts from 300h, then all these parameters can be stored in a compact form as illustrated by Figure 3.1.

(300h)=A(1,1) ... (307h)=A(1,8)	(308h)=B(1,1) ... (30Eh)=B(1,7)	(30Fh)=f(1)
(310h)=A(2,1) ... (317h)=A(2,8)	(318h)=B(2,1) ... (31Eh)=B(2,7)	(31Fh)=f(2)
(350h)=A(6,1) ... (357h)=A(6,8)		
(360h)=D(1,1) ... (367h)=D(8,8)	(368h)=B(7,1) ... (36Eh)=B(7,7)	(36Fh)=f(7)
(370h)=e(1) (377h)=e(7)	(378h)=e(8) unused 6 locations	(37Fh)=f(8)

Figure 3.1: Compact Storage Scheme of Example 1

3.3 The Macro Library

The extensive macro library provides a variety of implementation of the Kalman filter. Appendix F contains all the macro codes which are written with extensive comments.

3.3.1 General Matrix Macros

A brief summary of the general macros is given below :

1. Make necessary Initialization

- macro : MakeInit
- operation : Make general initializations that are used by all macros
- description : make necessary initialization for fixed-point Q15-based numerical computation

2. Scalar Addition or Subtraction

- macro : ScalAorS Location, OPTION
- operation :
 - (a) $[ACCH] + [Location] \longrightarrow [ACCH]$, if $OPTION = 0$
 - (b) $[ACCH] - [Location] \longrightarrow [ACCH]$, if $OPTION = 1$
- description : add (or subtract) a scalar value stored at “Location” to (or from) high accumulator(ACCH)

3. Vector Addition or Subtraction

- macro : VectAorS M, Loc_of_a, Loc_of_b, OPTION
- operation :
 - (a) $\bar{a} + \bar{b} \longrightarrow \bar{a}$, if $OPTION = 0$
 - (b) $\bar{a} - \bar{b} \longrightarrow \bar{a}$, if $OPTION = 1$
- description : add (or subtract) an $M \times 1$ column vector \bar{b} stored at “Loc_of_b” in data memory to (or from) another $M \times 1$ column vector \bar{a} stored at “Location_of_a” in data memory and store the resulting vector in \bar{a} ’s place

4. Vector Multiplication or Division by scalar

- macro : VectMorD M , Loc_of_a, Loc_of_c, OPTION
- operation :
 - (a) $\bar{a} \times [\text{ACCH}] \longrightarrow \bar{c}$, if OPTION = 0
 - (b) $\bar{a} \div [\text{ACCH}] \longrightarrow \bar{c}$, if OPTION = 1
- description : multiply (or divide) an $M \times 1$ column vector \bar{a} stored at “Loc_of_a” in data memory by a scalar stored in upper-half of accumulator and store the resulting vector \bar{c} at “Loc_of_c” in data memory

5. Vector Vector Multiplication

- macro : VecVecMl M , Loc_of_a, Loc_of_b, Loc_of_C, OPTION
- operation :
 - (a) $\bar{a}(\text{row vector in data memory}) \times \bar{b}(\text{column vector in data memory}) \longrightarrow [\text{ACCH}]$, if OPTION = 0
 - (b) $\bar{a}(\text{column vector in data memory}) \times [\bar{b}(\text{column vector in data memory})]^T \longrightarrow \hat{C}(\text{only the lower-half})$, if OPTION = 1
 - (c) $\bar{a}(\text{row vector in program memory}) \times \bar{b}(\text{column vector in data memory}) \longrightarrow [\text{ACCH}]$, if OPTION = 2
- description : find inner-product (or outer-product) of two vectors \bar{a} and \bar{b} stored at “Loc_of_a” and “Loc_of_b” respectively and store the inner-product in high accumulator (or the lower-half of the outer-product at “Loc_of_C” in data memory)

6. Matrix Addition or Subtraction

- macro : Mat_AorS M , Loc_of_A, Loc_of_B, OPTION
- operation :
 - (a) $\hat{A}(\text{lower-triangular matrix}) + \hat{B}(\text{lower-triangular matrix}) \longrightarrow \hat{A}$, if OPTION = 0
 - (b) $\hat{A}(\text{lower-triangular matrix}) - \hat{B}(\text{lower-triangular matrix}) \longrightarrow \hat{A}$, if OPTION = 1
 - (c) $\hat{A}(\text{lower-triangular matrix}) + \hat{B}(\text{diagonal matrix}) \longrightarrow \hat{A}$, if OPTION = 2
- description : add (or subtract) an $M \times M$ lower-triangular matrix or diagonal matrix \hat{B} located at “Loc_of_B” in data memory to (or from) an $M \times M$ lower-triangular matrix \hat{A} located at “Loc_of_A” in data memory and store the resulting lower-triangular matrix in \hat{A} ’s place

7. Matrix Matrix Multiplication between program memory and data memory

- macro : MtMtMlpd $M, N, P, \text{Loc_of_A}, \text{Loc_of_B}, \text{Loc_of_C}, \text{OPTION_1}, \text{OPTION_2}$
- operation :
 - (a) $\hat{A}(\text{in program memory}) \times \hat{B}(\text{in data memory}) \longrightarrow \hat{C}$,
if $\text{OPTION_1} = 0$ and $\text{OPTION_2} = 0$
 - (b) $[\hat{A}(\text{in program memory}) \times \hat{B}(\text{in data memory})]^T \longrightarrow \hat{C}$,
if $\text{OPTION_1} = 1$ and $\text{OPTION_2} = 0$
 - (c) $\hat{A}(\text{in program memory}) \times \hat{B}(\text{in data memory}) \longrightarrow \hat{C}$ (only the lower-half), if $\text{OPTION_1} = x$ (don't care) and $\text{OPTION_2} = 1$
- description : multiply an $M \times N$ matrix \hat{A} stored at "Loc_of_A" in program memory by an $N \times L$ matrix \hat{B} stored at "Loc_of_B" in data memory and store the resulting matrix or its transpose or the only the lower-half of resulting matrix (if it is known to be symmetric beforehand) at "Loc_of_C" in data memory

8. Matrix Matrix Multiplication between data memory and data memory

- macro : MtMtMldd $M, N, S, T, \text{Loc_of_A}, \text{Loc_of_B}, \text{Loc_of_C}, \text{OPTION_1}, \text{OPTION_2}$
- operation :
 - (a) $\hat{A}(\text{in data memory}) \times \hat{B}(\text{in data memory}) \longrightarrow \hat{C}$,
if $\text{OPTION_1} = 0$ and $\text{OPTION_2} = 0$
 - (b) $[\hat{A}(\text{in data memory}) \times \hat{B}(\text{in data memory})]^T \longrightarrow \hat{C}$,
if $\text{OPTION_1} = 1$ and $\text{OPTION_2} = 0$
 - (c) $\hat{A}(\text{in data memory}) \times \hat{B}(\text{in data memory}) \longrightarrow \hat{C}$ (only the lower-half), if $\text{OPTION_1} = 0$ and $\text{OPTION_2} = 1$
 - (d) $\hat{A}^T(\text{in data memory}) \times \hat{B}(\text{in data memory}) \longrightarrow \hat{C}$,
if $\text{OPTION_1} = 1$ and $\text{OPTION_2} = 1$
- description : multiply an $M \times N$ matrix \hat{A} (or its transpose) stored at "Loc_of_A" in data memory by an $S \times T$ matrix \hat{B} stored at "Loc_of_B" in data memory and store the resulting matrix or its transpose or the only the lower-half of resulting matrix (if it is known to be symmetric beforehand) at "Loc_of_C" in data memory. Note that for the first three options, $N = S$, else $M = S$

9. Fill upper-half of a Matrix from its lower-half

- macro : Fill_Mat M, Loc_of_A
- operation : \hat{A} (lower-triangular matrix) \longrightarrow \hat{A} (symmetric matrix)
- description : fill the upper-half of a $M \times M$ lower-triangular matrix \hat{A} stored at “Loc_of_A” in data memory with its lower-half and thereby symmetrize the matrix

10. Matrix Copying

- macro Mat_Copy M, N, SOURCE, DEST
- operation : \hat{A} (at SOURCE in data memory) \longrightarrow \hat{A} (at DEST in data memory)
- description : copy an $M \times N$ matrix \hat{A} stored at “SOURCE” in data memory to “DEST” in data memory

11. Move from Program memory to Data memory

- macro : Move_P_D M, N, SOURCE_P, DEST_D
- operation : \hat{A} (at SOURCE_P in program memory) \longrightarrow \hat{A} (at DEST_D in data memory)
- description : move an $M \times N$ matrix \hat{A} stored at “SOURCE_P” in program memory to “DEST_D” in data memory

12. Q15 Division

- macro : Q15_Div
- operation : $[ACCH] \div [065h] \longrightarrow [ACCL]$
- description : divide a Q15 number stored in high accumulator(ACCH) by another Q15 number stored at 065h of data memory and store the result in low accumulator(ACCL)

13. LU-Factorization of a symmetric matrix

- macro : LU_Fact M, Loc_of_A, Loc_of_C
- operation : \hat{A} (lower-triangular matrix) $\longrightarrow \hat{C} = \hat{L} \times \hat{U} = \hat{A}$
- description : From the knowledge of lower-half of a symmetric matrix \hat{A} stored at “Loc_of_A” in data memory, perform LU decomposition such that $\hat{C} = \hat{L} \times \hat{U} = \hat{A}$, and store it in compact form at “Loc_of_C” in data memory, where \hat{L} is the lower-triangular half and \hat{U} is the upper-triangular half of \hat{C} . The diagonal entries in \hat{C} correspond to those of \hat{L} , whereas diagonals of \hat{U} are not stored since they are known to be 1's

14. solve a system of equation by **ForWard** substitution

- macro : For_Ward M, N, Loc_of_L, Loc_of_B, Loc_of_Y
- operation : solve \hat{Y} from $\hat{L}\hat{Y} = \hat{B}^T$
- description : using forward substitution, solve \hat{Y} from $\hat{L}\hat{Y} = \hat{B}^T$, where \hat{L} is an $M \times M$ lower-triangular matrix stored at “Loc_of_L” in data memory and \hat{B} is an $N \times M$ matrix stored at “Loc_of_B” in data memory. \hat{Y} is stored at “Loc_of_Y” in data memory

15. solve a system of equation by **BackWard** substitution

- macro : Bck_Ward M, N, Loc_of_U, Loc_of_Y
- operation : solve \hat{X} from $\hat{U}\hat{X} = \hat{Y}$
- description : using backward substitution, solve \hat{X} from $\hat{U}\hat{X} = \hat{Y}$ where \hat{U} is an $M \times M$ upper-triangular matrix with diagonals as 1's stored at “Loc_of_U” in data memory and \hat{Y} is an $M \times N$ matrix stored at “Loc_of_Y” in data memory, and store \hat{X} in \hat{Y} 's place

16. find **Square-Root** of a Q15 number

- macro : Sqr_Root
- operation : $\sqrt{[ACCH]} \rightarrow [ACCL]$
- description : find the square-root of a Q15 number stored in high accumulator(ACCH) by Newton-Raphson method and store it in low accumulator(ACCL)

17. **Choleski** factorization of a symmetric matrix

- macro : Choleski M, Loc_of_A
- operation : $\hat{A}(\text{lower-triangular matrix}) \rightarrow \hat{C} = \hat{A}^c(\hat{A}^c)^T$
- description : From the knowledge of lower-half of a symmetric matrix \hat{A} stored at “Loc_of_A” in data memory, perform Choleski-decomposition such that $\hat{A} = \hat{A}^c(\hat{A}^c)^T$ and store the lower-triangular matrix \hat{A}^c in \hat{A} 's place

18. perform **Sequential Processing**

- macro : Seq_Proc M, N, Loc_of_H, Loc_of_P, Loc_of_R, Loc_of_y, Loc_of_x, Loc_of_k, temp_1, temp_2
- operation : find the filtered estimates by sequential processing

- description : for vector observation case, find filtered estimates of the state vector $\bar{x}(k+1 | k+1)$ and the error covariance matrix $\hat{P}(k+1 | k+1)$ by sequential processing as discussed in last chapter, where $M \times N$ matrix \hat{H} , $N \times N$ matrix \hat{P} , $M \times M$ diagonal matrix \hat{R} , $M \times 1$ column vector \bar{y} and $N \times 1$ column vector \bar{x} are stored at "Loc_of_H", "Loc_of_P", "Loc_of_R", "Loc_of_y" and "Loc_of_x" in data memory respectively. The intermediate result \bar{k} is stored at "Loc_of_k" whereas "temp_1" and "temp_2" are $N \times N$ and $N \times 1$ storage locations used for temporary storage.

3.3.2 Some Special Macros

To get rid of unnecessary computations like multiplication with ones and zeros or addition with zeros, some special macros are written when state transition matrix $\hat{\Phi}$ have the following block diagonal structure :

$$\hat{\Phi} = \begin{bmatrix} \hat{\Phi}_1 & | & 0 & | & & | & 0 \\ \hline & & & & & & \\ 0 & | & \hat{\Phi}_2 & | & & | & \vdots \\ \hline & & & & & & \\ & & & & & & 0 \\ \hline & & & & & & \\ 0 & | & & | & 0 & | & \hat{\Phi}_i \end{bmatrix}, \hat{\Phi}_j = \begin{bmatrix} \alpha_j & \beta_j \\ -\beta_j & \alpha_j \end{bmatrix} \quad (3.1)$$

and observation vector \bar{h} is of the form :

$$\bar{h} = \begin{bmatrix} \bar{h}_1 & | & \bar{h}_2 & | & \dots & | & \bar{h}_i \end{bmatrix}, \bar{h}_i = \begin{bmatrix} 1 & 0 \end{bmatrix}, \quad (3.2)$$

where $1 \leq i, j \leq 4$. In these macros, the observation vector \bar{h} does not have to be stored. Another important feature is that state vector \bar{x} is stored as a row vector. The upper-half of error covariance matrix \hat{P} is never calculated and in the computations, \hat{P} is assumed to be symmetric. This results in considerable savings when Kalman filter having the above mentioned state-space structure is implemented using these special macros.

1. Find **PRED**icted **EST**imates of state vector and error covariance matrix

- macro : PRED_EST N, Loc_of_Phi, Loc_of_P, Loc_of_x, Loc_of_Q
- operation : $\hat{\Phi}\bar{x} \longrightarrow \bar{x}$ and $\hat{\Phi}\hat{P}\hat{\Phi}^T + \hat{Q} \longrightarrow \hat{P}$

- description : find predicted estimates of the state vector $\bar{x}(k+1 | k)$ and the error covariance matrix $\hat{P}(k+1 | k)$ where $N \times N$ block diagonal matrix $\hat{\Phi}$, $N \times N$ lower-triangular (symmetric) matrix \hat{P} , $1 \times N$ vector \bar{x} and $N \times N$ diagonal matrix \hat{Q} are stored at “Loc_of_Phi”, “Loc_of_P”, “Loc_of_x” and “Loc_of_Q” in data memory respectively. The macro uses three other macros namely MAT221, MAT222 and PHLX for operations on 2×2 block matrices. The operations performed by these three macros are :
 - MAT221 : $\hat{A}\hat{B}(\text{lower-triangular})\hat{A}^T + \hat{D}(\text{diagonal}) \longrightarrow \hat{B}$
 - MAT222 : $\hat{A}\hat{B}\hat{C}^T \longrightarrow \hat{C}$
 - PHLX : $\hat{\Phi}\bar{x} \longrightarrow \bar{x}$

2. Find **FILTERed ESTimates** of state vector and error covariance matrix

- macro : `FILT_EST N, Loc_of_P, temp_1, Loc_of_r, Loc_of_k, Loc_of_x, Loc_of_y`
- operation : $\hat{P}\bar{h}^T/[\bar{h}\hat{P}\bar{h}^T + R] \longrightarrow \bar{k}$, $\hat{P} - \hat{P}\bar{h}^T\bar{k}^T \longrightarrow \hat{P}$
and $\bar{x} - \bar{k}[\bar{h}\bar{x} - y] \longrightarrow \bar{x}$
- description : find filtered estimates of state vector $\bar{x}(k+1 | k+1)$ and error covariance matrix $\hat{P}(k+1 | k+1)$ where $N \times N$ lower-triangular matrix \hat{P} , observation variance R , $1 \times N$ vector \bar{x} and observation y are stored at “Loc_of_P”, “Loc_of_r”, “Loc_of_x” and “Loc_of_y” in data memory respectively. It should be noted that this is a special macro which serves some other specific purposes. To communicate with the outside world, it resets external flag(XF) to get data. The new data is written into “Loc_of_y” by the external processor whereas previously processed data which is the sum of every other entries of filtered state vector is written into next location. At the end of the macro, the external flag is set. As far as I/O operations are concerned, the macro converts offset binary(OB) format into 2’s complement format and vice versa.

3.4 Building up the Kalman Filter Algorithms

The wide choice of macros provides a variety of standard and user-defined implementations of the Kalman filter. While using the macros in the filter algorithm, the architecture of the TMS processor has to be exploited to get the best throughput. For example, the multiplication of two operands (vectors

or matrices) can be very efficiently performed at minimum cost of program memory by putting one operand in program memory and the other in data memory. The TMS320C25 digital signal processor has 544 words of on-chip data RAM of which 256 words can be configured either as program memory or data memory by a software command (CNFP/CNFD instruction). Apart from this, all the system boards that are developed around the TMS320C25 chip have no-wait-state external RAM which can be as large as 123K. As far as execution speed is concerned, the most efficient implementation requires that both the data and the program space should reside in on-chip RAM. However, because of macro-based straight-line code implementation, the program occupies considerable space. To make an optimal trade-off between speed and memory space, the macros are written mostly from the first four classes of instruction set for which running the program from no-wait-state external RAM with data in on-chip RAM is as fast as executing from on-chip RAM. According to the number of cycles required for execution, all the instructions are grouped into fifteen classes. When both the program and the data are in on-chip RAM, the first four classes take one instruction cycle. On the other hand, for external program RAM and on-chip data RAM each instruction takes $(p + 1)$ cycles where p is the wait state of external program memory. Hence it is recommended to store the data and the intermediate results in on-chip data RAM and run the program from no-wait-state external program RAM for which $p=0$. One of the most-likely problems encountered in implementation of the Kalman filter is the divergence of error covariance matrix \hat{P} as discussed in chapter two. This problem is automatically solved as the macros concerning the predicted and the filtered estimates of \hat{P} calculate only its lower-half and then fills the upper-half making sure that the symmetry is preserved. In the correction part of error covariance matrix \hat{P} , the filter step terms are rearranged to make it more suitable for TMS structure. Since the error covariance $\hat{P}(k + 1 | k + 1)$ given by,

$$\hat{P}(k + 1 | k + 1) = [I - \hat{K}(k + 1)\hat{H}]\hat{P}(k + 1 | k) \quad (3.3)$$

is symmetric, it can be written as,

$$\begin{aligned} \hat{P}(k + 1 | k + 1) &= \hat{P}^T(k + 1 | k + 1) \\ &= [\hat{P}(k + 1 | k) - \hat{K}(k + 1)\hat{H}\hat{P}(k + 1 | k)]^T \\ &= \hat{P}(k + 1 | k) - \hat{P}(k + 1 | k)\hat{H}^T\hat{K}^T(k + 1) \end{aligned} \quad (3.4)$$

The term $\hat{P}(k + 1 | k)\hat{H}^T$ needs to be calculated for the computation of Kalman gain in the preceding step. Hence, it can be stored in a temporary storage location and readily used in error covariance correction. For TMS320C25, this

form of realization of $\hat{P}(k+1 | k+1)$ is computationally faster than its standard form.

The macros can be used to define a new user-defined macro. While doing so, two important points should be taken into account - the auxiliary register AR0 should not be changed and the data memory locations 65-68h of on-chip block B0 should not be used.

The implementation of the Kalman filter using the macro library is discussed in the following subsections with examples.

3.4.1 Scalar Observation

For scalar observation, two different approaches are considered. The first approach is called “LTA and MPY pair option” because of frequent use of these two instructions in multiplication. In this case, the data and the intermediate results are stored in data memory. In the second implementation, they reside either in program memory or data memory to exploit the fast MAC(Multiply and ACcumulate) instruction and hence is called “MAC option”.

LTA and MPY Pair Option

For a dynamic system up to ten states i. e. up to $N = 10$, all permanent data and intermediate results can be stored in on-chip data RAM. The scalars, vectors, matrices and intermediate results can be efficiently accommodated in on-chip RAM as shown in Figure 3.2. Incorporation of macros in Kalman filter algorithm is illustrated in Example 3.2. Note that the initialization part and the data acquisition parts are not included in this example as well as the subsequent examples.

Example 3.2

Consider the case when the dynamic system has N states where $2 \leq N \leq 10$. Let us suppose that the $N \times N$ state transition matrix $\hat{\Phi}$ is stored at “Loc.of_phi”, $N \times N$ error covariance matrix \hat{P} at “Loc.of_P”, $N \times N$ measurement noise covariance matrix \hat{Q} at “Loc.of_Q”, $N \times 1$ state vector \bar{x} at “Loc.of_x”, the process noise variance R at “Loc.of_r” and the observation data y is stored at “Loc.of_y”. For storage of intermediate results, the locations “temp_1” and “temp_2” as indicated by Figure 3.2 are used. The filter steps can be realized as :

$\left. \begin{array}{l} 200h - 209h \\ 210h - 219h \end{array} \right\} \phi$ $\left. \begin{array}{l} 290h - 299h \end{array} \right\}$ * Loc_of_phi=200h	$\left. \begin{array}{l} 29Ah - 2A3h \\ 2AAh - 2B3h \end{array} \right\} P$ $\left. \begin{array}{l} 32Ah - 333h \end{array} \right\}$ * Loc_of_P=29Ah	$\left. \begin{array}{l} 334h - 33Dh \\ 344h - 34Dh \end{array} \right\}$ $\left. \begin{array}{l} 3C4h - 3CDh \end{array} \right\}$ temp_1: for storage of intermediate results ($\Phi h'k'$ and $P\phi'$) * temp_1=334h
$\left. \begin{array}{l} 33Eh \\ 34Eh \end{array} \right\} x$ $\left. \begin{array}{l} 3CEh \end{array} \right\}$ * Loc_of_x=33Eh	$\left. \begin{array}{l} 33Fh \\ 34Fh \end{array} \right\}$ temp_2: for storage of intermediate results ($\phi x, \Phi h', k(hx-y)$) $\left. \begin{array}{l} 3C0h \end{array} \right\}$ * temp_2=33Fh	$\left. \begin{array}{l} 340h \\ 350h \end{array} \right\} k$ $\left. \begin{array}{l} 3D0h \end{array} \right\}$ * Loc_of_k=340h
3E0h - 3E9h : h ; * Loc_of_h=3E0h		
3F0h - 3F9h : diagonals of Q ; * Loc_of_Q=3F0h		
3FAh : r ; * Loc_of_r=3FAh		
3FBh : y : * Loc_of_y=3FBh		

Figure 3.2: Storage Scheme for LTA and MPY pair option

- find $\bar{x}(k+1 | k) = \hat{\Phi} \bar{x}(k | k)$
MtMtMldd $N, N, N, 1, \text{Loc_of_phi}, \text{Loc_of_x}, \text{temp_2}, 0, 0$
; temp_2 $\leftarrow \hat{\Phi} \bar{x}$
Mat_Copy $N, 1, \text{temp_2}, \text{Loc_of_x}$; $\bar{x} \leftarrow \text{temp_2}$
- find $\hat{P}(k+1 | k) = \hat{\Phi} \hat{P}(k | k) \hat{\Phi}^T + \hat{Q}$
MtMtMldd $N, N, N, N, \text{Loc_of_phi}, \text{Loc_of_P}, \text{temp_1}, 1, 0$
; temp_1 $\leftarrow [\hat{\Phi} \hat{P}]^T = \hat{P} \hat{\Phi}^T$
MtMtMldd $N, N, N, N, \text{Loc_of_phi}, \text{temp_1}, \text{Loc_of_P}, 0, 1$
; $\hat{P} \leftarrow \hat{\Phi} \hat{P} \hat{\Phi}^T$ (only lower-half)
Mat_AorS $N, \text{Loc_of_P}, \text{Loc_of_Q}, 2$; $\hat{P} \leftarrow \hat{\Phi} \hat{P} \hat{\Phi}^T + \hat{Q}$
Fill_Mat $N, \text{Loc_of_P}$; fill the upper of \hat{P}
- find $\bar{k}(k+1) = \hat{P}(k+1 | k) \bar{h}^T / (\bar{h} \hat{P}(k+1 | k) \bar{h}^T + R)$
MtMtMldd $1, N, N, N, \text{Loc_of_h}, \text{Loc_of_P}, \text{temp_2}, 1, 0$
; temp_2 $\leftarrow [\bar{h} \hat{P}]^T = \hat{P} \bar{h}^T$
VecVecMl $N, \text{Loc_of_h}, \text{temp_2}, 0, 0$; (ACCH) $\leftarrow \bar{h} \hat{P} \bar{h}^T$
ScalAorS Loc_of_r ; (ACCH) $\leftarrow (\text{ACCH}) + R$
VectMorD $N, \text{temp_2}, \text{Loc_of_k}, 1$; $\bar{k} \leftarrow \hat{P} \bar{h}^T / (\bar{h} \hat{P} \bar{h}^T + R)$
- find $\hat{P}(k+1 | k+1) = \hat{P}(k+1 | k) - \hat{P}(k+1 | k) \bar{h}^T \bar{k}^T (k+1)$
VecVecMl $N, \text{temp_2}, \text{Loc_of_k}, \text{temp_1}, 1$
; temp_1 $\leftarrow \hat{P} \bar{h}^T \bar{k}^T$ (only lower-half)

Mat_AorS N , Loc_of_P, temp_1, 1 ; $\hat{P} \leftarrow \hat{P} \bar{h}^T \bar{k}^T$
 Fill_Mat N , Loc_of_P ; fill the upper-half of \hat{P}

- find $\bar{x}(k+1 | k+1) = \bar{x}(k+1 | k) - \bar{k}(k+1)[\bar{h}\bar{x}(k+1 | k) - y]$
 VecVecMl N , Loc_of_h, Loc_of_x, 0, 0 ; (ACCH) $\leftarrow \bar{h}\bar{x} - y$
 ScalAorS Loc_of_y, 1 ; (ACCH) \leftarrow (ACCH) $- y$
 VectMorD N , Loc_of_k, temp_2, 0 ; temp_2 $\leftarrow \bar{k}(\bar{h}\bar{x} - y)$
 VectAorS N , Loc_of_x, temp_2, 1 ; $\bar{x} \leftarrow \bar{x} - \bar{k}(\bar{h}\bar{x} - y)$

MAC Option

In this case, the fast multiply and accumulate instruction of TMS320C25 is exploited. To achieve this operands (i. e. vectors and matrices) are stored either in program memory or data memory. For a filter that involves up to 8 state variables all the permanent data and intermediate results can be accommodated in on-chip RAM as shown in Figure 3.3. Note that CNFD/CNFP instruction moves the on-chip block B0 back and forth between 200h of data memory and 0FF00h of program memory. The incorporation of macros into the standard Kalman filter is described in Example 3.3.

Example 3.3

The Example 3.2 is reconsidered again with the difference that in this case, $2 \leq N \leq 8$. For this case, the filter steps can be summarized as :

- cnfp ; configure block B0 as program memory
- find $\bar{x}(k+1 | k) = \hat{\Phi}\bar{x}(k | k)$
 MtMtMlPd $N, N, 1, 0FF00h-200h+Loc_of_phi, Loc_of_x, temp_2, 0, 0$
 ; temp_2 $\leftarrow \hat{\Phi}\bar{x}$
 Mat_Copy $N, 1, temp_2, Loc_of_x$; $\bar{x} \leftarrow temp_2$
- find $\hat{P}(k+1 | k) = \hat{\Phi}\hat{P}(k | k)\hat{\Phi}^T + \hat{Q}$
 MtMtMlPd $N, N, N, 0FF00h-200h+Loc_of_phi, Loc_of_P, temp_1, 1, 0$
 ; temp_1 $\leftarrow [\hat{\Phi}\hat{P}]^T = \hat{P}\hat{\Phi}^T$
 MtMtMlPd $N, N, N, 0FF00h-200h+Loc_of_phi, temp_1, Loc_of_P, 1, 1$
 ; $\hat{P} \leftarrow \hat{\Phi}\hat{P}\hat{\Phi}^T$ (only lower-half)
 Mat_AorS N , Loc_of_P, Loc_of_Q, 2 ; $\hat{P} \leftarrow \hat{\Phi}\hat{P}\hat{\Phi}^T + \hat{Q}$
 Fill_Mat N , Loc_of_P ; fill the upper of \hat{P}
- find $\bar{k}(k+1) = \hat{P}(k+1 | k)\bar{h}^T / (\bar{h}\hat{P}(k+1 | k)\bar{h}^T + R)$
 MtMtMlPd 1, $N, N, 0FF00h - 200h + Loc_of_h, Loc_of_P, temp_2, 1, 0$
 ; temp_2 $\leftarrow [\bar{h}\hat{P}]^T = \hat{P}\bar{h}^T$

- VecVecMl N , $0FF00h - 200h + \text{Loc_of_h}$, temp_2, 0, 2
 ; (ACCH) $\leftarrow \bar{h}\hat{P}\bar{h}^T$
 ScalAorS Loc_of_r ; (ACCH) $\leftarrow (\text{ACCH}) + R$
 VectMorD N , temp_2, Loc_of_k, 1 ; $\bar{k} \leftarrow \hat{P}\bar{h}^T / (\bar{h}\hat{P}\bar{h}^T + R)$
- find $\hat{P}(k+1 | k+1) = \hat{P}(k+1 | k) - \hat{P}(k+1 | k)\bar{h}^T\bar{k}^T(k+1)$
 VecVecMl N , temp_2, Loc_of_k, temp_1, 1
 ; temp_1 $\leftarrow \hat{P}\bar{h}^T\bar{k}^T$ (only the lower-half)
 MatAorS N , Loc_of_P, temp_1, 1 ; $\hat{P} \leftarrow \hat{P} - \hat{P}\bar{h}^T\bar{k}^T$
 Fill_Mat N , Loc_of_P ; fill the upper-half of \hat{P}
 - find $\bar{x}(k+1 | k+1) = \bar{x}(k+1 | k) - \bar{k}(k+1)[\bar{h}\bar{x}(k+1 | k) - y]$
 VecVecMl N , $0FF00h - 200h + \text{Loc_of_h}$, Loc_of_x, 0, 2
 ; (ACCH) $\leftarrow \bar{h}\bar{x} - y$
 ScalAorS Loc_of_y, 1 ; (ACCH) $\leftarrow (\text{ACCH}) - y$
 VectMorD N , Loc_of_k, temp_2, 0 ; temp_2 $\leftarrow \bar{k}(\bar{h}\bar{x} - y)$
 VectAorS N , Loc_of_x, temp_2, 1 ; $\bar{x} \leftarrow \bar{x} - \bar{k}(\bar{h}\bar{x} - y)$

For $9 \leq n \leq 14$, the on-chip RAM can not accommodate all permanent and temporary data. In this case, some of the data have to reside in external RAM. To keep the execution speed as fast as possible, only those data that are less involved in computations are stored in external memory.

200h - 207h 210h - 217h 270h - 277h * Loc_of_phi=200h	300h - 307h 310h - 317h 370h - 377h * Loc_of_P=300h	308h - 30Fh 318h - 31Fh 378h - 37Fh * temp_1=308h	temp_1 : for storage of intermediate results ($\Phi'k', P\Phi'$)
380h 390h 3F0h * Loc_of_x=380h	381h 391h 3F1h * temp_2=381h	382h 392h 3F2h * Loc_of_k=382h	temp_2: for storage of intermediate results ($\Phi x, \Phi'k', k(hx-y)$)
2F0h - 2F7h : h ; * Loc_of_h=2F0h			
3F4h - 3FBh : diagonals of Q ; * Loc_of_Q=3F4h			
3FCh : r ; * Loc_of_r=3FCh			
3FDh : y ; * Loc_of_y=3FDh			

Figure 3.3: Storage Scheme for MAC option

3.4.2 Vector Observation

When the observation is a vector quantity, the term $(\hat{H}\hat{P}\hat{H}^T + \hat{R})$ is no longer a scalar but a symmetric matrix. Hence inversion of an $(M \times M)$ matrix is required if the standard Kalman filter is to be implemented. This problem can be overcome by sequential processing of observation vector. Another way to circumvent inversion is to solve Kalman gain \hat{K} as a system of linear equations using either LU or Choleski factorization. In both the cases, matrix and vector multiplications are accomplished by “MAC option” i. e. we assume that the operands reside either in data memory or in program memory to maximize execution speed.

Sequential Processing :

For a system having up to 8 states and observation vector having up to 8 components i. e. up to $N = 8, M = 8$, all the permanent and intermediate results can be put into on-chip RAM. The storage scheme for this case is illustrated in Figure 3.4. The sequential processing can be very easily realized

$\left. \begin{array}{l} 200h - 207h \\ 210h - 217h \end{array} \right\} - \phi$		$\left. \begin{array}{l} 208h - 20Fh \\ 218h - 21Fh \end{array} \right\} - H$		
270h - 277h		278h - 27Fh		
* Loc_of_phi=200h		* Loc_of_H=208h		
$\left. \begin{array}{l} 300h - 307h \\ 310h - 317h \end{array} \right\} - P$		$\left. \begin{array}{l} 308h - 30Fh \\ 318h - 31Fh \end{array} \right\}$		
370h - 37Fh		378h - 37Fh		
* Loc_of_P=300h		* temp_1=308h		
		temp_1 : for storage of intermediate results (PH'K' , P ϕ ')		
$\left. \begin{array}{l} 380h \\ 390h \end{array} \right\} - x$	$\left. \begin{array}{l} 381h \\ 391h \end{array} \right\}$	temp_2: for storage of inter- mediate results ($\phi x, Ph'$, F(hx-y))	$\left. \begin{array}{l} 382h \\ 392h \end{array} \right\} - f$	$\left. \begin{array}{l} 383h \\ 393h \end{array} \right\} - y$
3F0h	3F1h	3F2h	3F3h	
*Loc_of_x=380h	*temp_2=381h	*Loc_of_f=382h	*Loc_of_y=383h	
385h - 38Ch : diagonals of R ; * Loc_of_R=385h				
395h - 39Ch : diagonals of Q ; * Loc_of_Q=395h				

Figure 3.4: Storage Scheme for Sequential Processing, $N = 8$

using the special macro “Seq_Proc” which finds the filtered estimates of state vector $\bar{x}(k+1 | k+1)$ and error covariance matrix $\hat{P}(k+1 | k+1)$. The use

of macros in sequential processing is described in Example 3.4.

Example 3.4

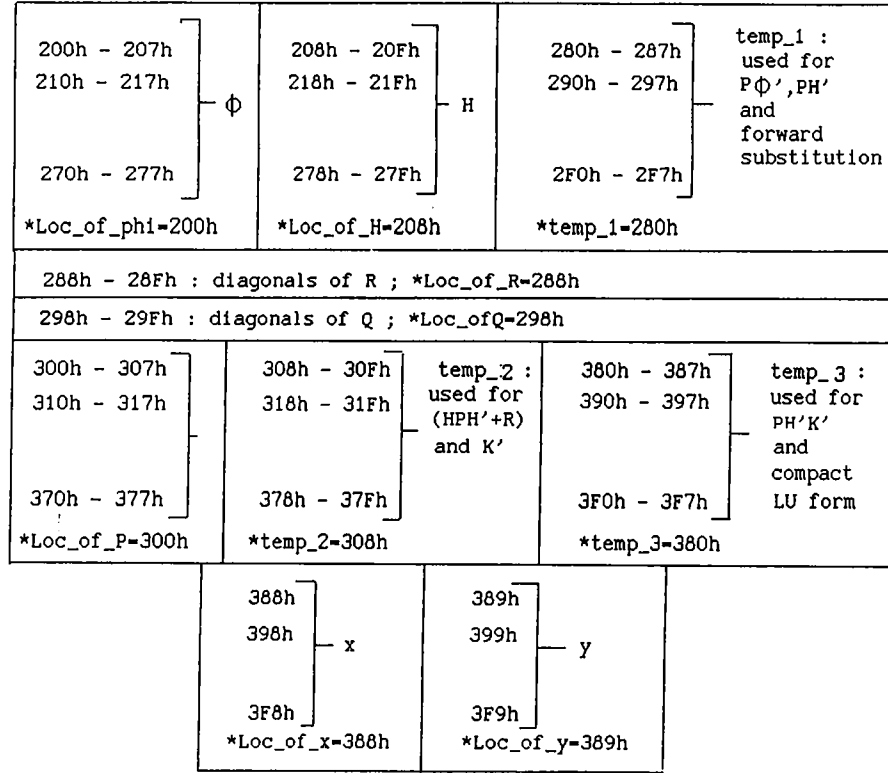
We shall consider the case when the system has N states and the measurement vector has M components, where $2 \leq M, N \leq 8$. The temporary storage and permanent data locations are the same as in Figure 3.4. The incorporation of macros into sequential processing is described as follows :

- cnfp ; configure block B0 as program memory
- find $\bar{x}(k+1 | k) = \hat{\Phi}\bar{x}(k | k)$.
 MtMtMlpd $N, N, 1, 0FF00h-200h+Loc_of_phi, Loc_of_x, temp_2, 0, 0$
 ; temp_2 $\leftarrow \hat{\Phi}\bar{x}$
 Mat_Copy $N, 1, temp_2, Loc_of_x$; $\bar{x} \leftarrow \hat{\Phi}\bar{x}$
- find $\hat{P}(k+1 | k) = \hat{\Phi}\hat{P}(k | k)\hat{\Phi}^T + \hat{Q}$
 MtMtMlpd $N, N, N, 0FF00h-200h+Loc_of_phi, Loc_of_P, temp_1, 1, 0$
 ; temp_1 $\leftarrow (\hat{\Phi}\hat{P})^T = \hat{P}\hat{\Phi}^T$
 MtMtMlpd $N, N, N, 0FF00h-200h+Loc_of_phi, temp_1, Loc_of_P, 1, 1$
 ; $\hat{P} \leftarrow \hat{\Phi}\hat{P}\hat{\Phi}^T$ (lower-half only)
 Mat_AorS $N, Loc_of_P, Loc_of_Q, 2$; $\hat{P} \leftarrow \hat{\Phi}\hat{P}\hat{\Phi}^T + \hat{Q}$
 Fill_Mat N, Loc_of_P ; fill the upper-half of \hat{P}
- find filtered estimate of state vector $\bar{x}(k+1 | k+1)$ and error covariance matrix $\hat{P}(k+1 | k+1)$.
 Seq_Proc $M, N, Loc_of_H, Loc_of_P, Loc_of_R, Loc_of_y, Loc_of_x, Loc_of_K,$
 temp_1, temp_2

Batch Processing

When all the elements of observation vector \bar{y} are treated at the same time, the Kalman gain matrix \hat{K} can be solved either by LU or Choleski decomposition. We shall consider only the case when LU decomposition is applied since there is not much difference in the way LU or Choleski decomposition is used to solve a system of linear equations. To avoid matrix inversion, the following steps are performed to compute Kalman gain \hat{K} :

- find $\hat{B} = \hat{P}\hat{H}^T$
- find $\hat{A} = \hat{H}\hat{P}\hat{H}^T + \hat{R}$
 $\Rightarrow \hat{K} = \hat{B}\hat{A}^{-1}$

Figure 3.5: Storage Scheme for Batch Processing, $N = 8$

$$\Rightarrow \hat{K}^T = \hat{A}^{-1} \hat{B}^T \text{ (since } \hat{A} \text{ is symmetric)}$$

$$\Rightarrow \hat{A} \hat{K}^T = \hat{B}^T$$

- perform LU decomposition of \hat{A} such that $\hat{A} = \hat{L} \hat{U}$, where \hat{L} is a lower-triangular matrix and \hat{U} is an upper-triangular matrix with 1's in its diagonals.

$$\Rightarrow \hat{L} \hat{U} \hat{K}^T = \hat{B}^T$$

$$\Rightarrow \hat{L} Y = \hat{B}^T, \text{ where } \hat{U} \hat{K}^T = Y$$

- solve \hat{Y} from $\hat{L} \hat{Y} = \hat{B}^T$ by forward substitution.
- solve \hat{K}^T from $\hat{U} \hat{K}^T = \hat{Y}$ by backward substitution.

For a system having up to 8 states and measurement vector having 8 elements all the permanent and intermediate results can be stored in on-chip RAM as shown in Figure 3.5. Incorporation of macros for batch processing is described in example 3.5.

Example 3.5

We consider the same system as in Example 3.4. Instead of sequential processing of observation data, the filter implementation is based on batch processing.

- cnfp ; configure block B0 as program memory
- find $\bar{x}(k+1 | k) = \hat{\Phi}\bar{x}(k | k)$
MtMtMlpd $N, N, 1, 0FF00h-200h+Loc_of_phi, Loc_of_x, temp_2, 0, 0$
; temp_2 $\leftarrow \hat{\Phi}\bar{x}$
Mat_Copy $N, 1, temp_2, Loc_of_x$; $\bar{x} \leftarrow \hat{\Phi}\bar{x}$
- find $\hat{P}(k+1 | k) = \hat{\Phi}\hat{P}(k | k)\hat{\Phi}^T$
MtMtMlpd $N, N, N, 0FF00h-200h+Loc_of_phi, Loc_of_P, temp_2, 1, 0$
; temp_2 $\leftarrow [\hat{\Phi}\hat{P}]^T = \hat{P}\hat{\Phi}^T$
MtMtMlpd $N, N, N, 0FF00h-200h+Loc_of_phi, temp_2, Loc_of_P, 1, 1$
; $\hat{P} \leftarrow \hat{\Phi}\hat{P}\hat{\Phi}^T$ (lower-half only)
cnfd ; configure block B0 as data memory
Mat_AorS $N, Loc_of_P, Loc_of_Q, 2$; $\hat{P} \leftarrow \hat{\Phi}\hat{P}\hat{\Phi}^T + \hat{Q}$
Fill_Mat N, Loc_of_P ; fill upper-half of \hat{P}
- find $\hat{K}(k+1)$
cnfp ; configure block B0 as program memory
MtMtMlpd $M, N, N, 0FF00h-200h+Loc_of_H, Loc_of_P, temp_2, 1, 0$
; temp_2 $\leftarrow [\hat{H}\hat{P}]^T = \hat{P}\hat{H}^T = \hat{B}$
MtMtMlpd $M, N, M, 0FF00h-200h+Loc_of_H, temp_2, temp_3, 1, 1$
; temp_3 $\leftarrow \hat{H}\hat{P}\hat{H}^T$ (only lower-half)
cnfd ; configure block B0 as data memory
Mat_AorS $M, temp_3, Loc_of_R, 2$; temp_3 $\leftarrow \hat{H}\hat{P}\hat{H}^T + \hat{R} = \hat{A}$
LU_Fact $M, temp_3, temp_1$; temp_1 $\leftarrow \hat{L}\hat{U} = \hat{A}$
For_Ward $M, N, temp_1, temp_2, temp_3$
; solve \hat{Y} from $\hat{L}\hat{Y} = \hat{B}^T$ and store it in temp_3
Bck_Ward $M, N, temp_1, temp_3$
; solve \hat{K}^T from $\hat{U}\hat{K}^T = \hat{Y}$ and store it in temp_3
- find $\hat{P}(k+1 | k+1) = \hat{P}(k+1 | k) + \hat{P}(k+1 | k)\hat{H}^T\hat{K}^T$
MtMtMldd $N, M, M, N, temp_2, temp_3, temp_1, 0, 1$
; temp_1 $\leftarrow \hat{P}\hat{H}^T\hat{K}^T$ (only the lower-half)
Mat_AorS $N, Loc_of_P, temp_1, 1$; $\hat{P} \leftarrow \hat{P} - \hat{P}\hat{H}^T\hat{K}^T$
Fill_Mat N, Loc_of_P ; fill the upper-half of \hat{P}
- find $\bar{x}(k+1 | k+1) = \bar{x}(k+1 | k) - \hat{K}(k+1)[\hat{H}\bar{x}(k+1 | k) - \bar{y}(k+1)]$
cnfp ; configure block B0 as program memory
MtMtMlpd $M, N, 1, 0FF00h-200h+Loc_of_H, Loc_of_x, temp_2, 0, 0$
; temp_2 $\leftarrow \hat{H}\bar{x}$
VectAorS $M, temp_2, Loc_of_y, 1$; temp_2 $\leftarrow \hat{H}\bar{x} - \bar{y}$
MtMtMldd $M, N, M, 1, temp_3, temp_2, temp_1, 1, 1$

```

; temp-1  $\leftarrow \hat{K}[\hat{H}\bar{x} - \bar{y}]$ 
VectAorS N, Loc.of-x, temp-1,1 ;  $\bar{x} \leftarrow \bar{x} - \hat{K}[\hat{H}\bar{x} - \bar{y}]$ 

```

3.5 Performance Evaluation of Different Implementations

Based on the usual criteria of any real-time implementation like program execution speed, consumption of memory space etc., the performance of different approaches is considered in this section. It should be noted that this performance evaluation is valid only when the Kalman filter is implemented on TMS320C25 digital signal processor. As a matter of fact, good performance of a particular implementation does not necessarily mean that the same scheme of implementation on a different digital signal processor or a general-purpose microprocessor will be as good as its TMS320C25 counterpart. As performance evaluation indicators, we consider total program memory space including initialization, data memory space needed to accommodate both permanent and intermediate results, total instruction cycles for a single iteration of filter algorithm and finally maximum sampling frequency that can be achieved assuming that TMS320C25 runs with 10MHz instruction cycle.

3.5.1 Scalar Observation

As discussed in the previous section, there are two ways to implement the Kalman filter when the observation is a scalar quantity. For “LTA and MPY pair option”, the required memory space as a function of state dimension is shown in a tabular form in Table 3.1 .

In this table, we consider the implementation of filter having as many as 10 states for which on-chip RAM can accommodate all pertinent data. It is assumed that I/O operation takes no more than 15 instruction cycles. For a system having N states, “LTA and MPY pair option” requires $(3N^2 + 5N + 2)$ words of data memory, $(82N^2 - 290N + 940)$ words of program memory and $(73N^2 - 193N + 520)$ instruction cycles per iteration.

The performance indices for the “MAC option” is considered for the filter implementation that involves up to 14 states. As discussed in previous section, external RAM is needed for $N \geq 9$. All the performance parameters for this

N	<i>program memory words</i>	<i>data memory words</i>	<i>instruction cycles</i>	<i>maximum sampling frequency in KHz</i>
2	475	24	390	25.64
3	739	44	638	15.67
4	1101	70	978	10.22
5	1567	102	1428	6.98
6	2167	140	2006	4.99
7	2915	184	2730	3.66
8	3829	234	3618	2.76
9	4927	290	4688	2.13
10	6227	352	5960	1.68

Table 3.1: LTA and MPY pair option

implementation are shown in Table 3.2. The memory space needed is $(N^2 + N)$ for data memory, $(277N - 211)$ for program memory and the number of instruction cycles per iteration is approximately $(43N^2 + 230)$ for $2 \leq N \leq 8$.

N	<i>program memory words</i>	<i>data memory words</i>	<i>instruction cycles</i>	<i>maximum sampling frequency in KHz</i>
2	432	18	395	25.32
3	623	32	630	15.87
4	848	50	934	10.71
5	1107	72	1316	7.67
6	1400	98	1785	5.6
7	1727	128	2350	4.25
8	2088	162	3020	3.31
9	2488	191	4539	2.2
10	2927	232	5663	1.77
11	3391	277	6345	1.58
12	3889	326	8401	1.19
13	4421	379	10073	0.993
14	4987	436	11937	0.837

Table 3.2: MAC option

When these two options are considered, it turns out that “MAC option” is far better than the “LTA and MPY pair option” as far as memory space is concerned. For a particular value of N , the maximum possible sampling frequency is essentially the same in both the cases. However, the “MAC option” requires much less memory space. These two facts are graphically illustrated in Figure 3.6 and Figure 3.7 respectively.

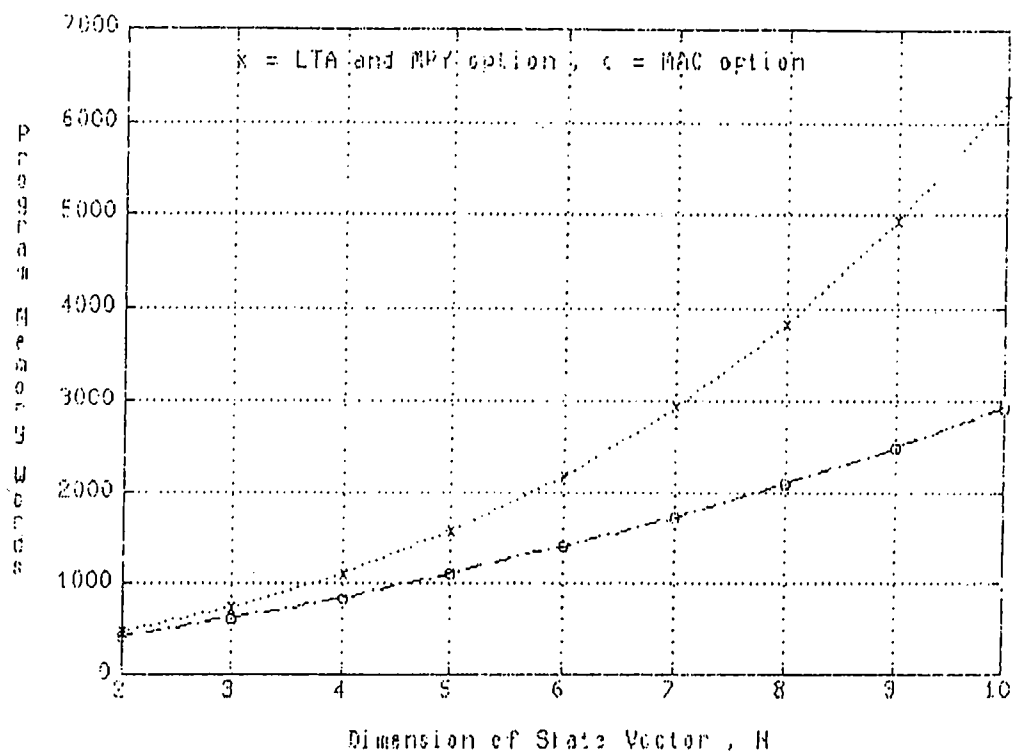


Figure 3.6: Program Memory Words Vs. Dimension of State Vector in Scalar Observation

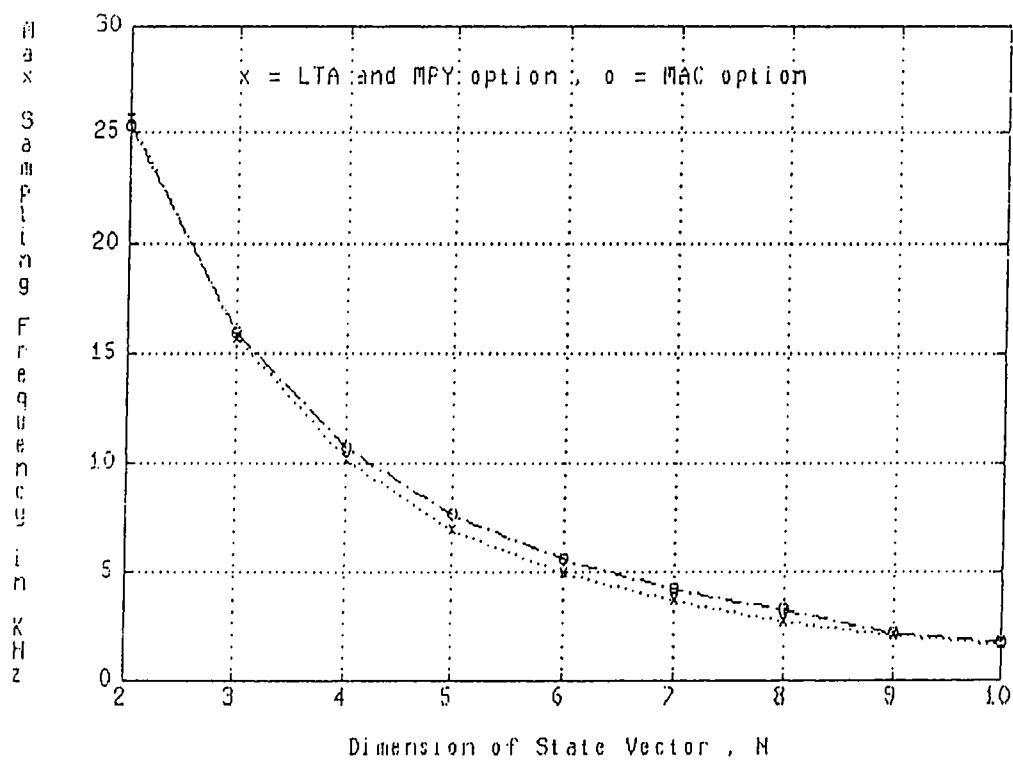


Figure 3.7: Max. Sampling Frequency Vs. Dimension of State Vector in Scalar Observation

3.5.2 Vector Observation

For vector observation case, there are two ways to implement the Kalman filter - sequential processing and batch processing. In both cases the performance is evaluated as a function of the dimension of observation vector M ($2 \leq M \leq 8$) for a fixed value of state dimension N (which is 8 in this case). It is assumed that I/O consumes $(2M + 15)$ program memory words and the same number of instruction cycles. The performance parameters for sequential processing is given in Table 3.3. The data memory needed in this case is $(2N^2 + 3N + 2M)$ words. However, it is difficult to give symbolic expression for program memory

M ($N = 8$)	program memory words	data memory words	instruction cycles	maximum sampling frequency in KHz
2	2771	156	3847	2.6
3	3455	158	4672	2.14
4	4137	160	5496	1.82
5	4819	162	6320	1.58
6	5501	164	7144	1.4
7	6179	166	7966	1.25
8	6861	168	8792	1.14

Table 3.3: Sequential Processing for $N = 8$, $2 \leq M \leq 8$

space and instruction cycle since they are functions of both N and M . As for batch processing, we consider the case when the Kalman gain matrix \hat{K} is solved by LU-decomposition. The performance indicators for this case are summarized in Table 3.4.

M ($N = 8$)	program memory words	data memory words	instruction cycles	maximum sampling frequency in KHz
2	3478	276	4537	2.204
3	4377	278	5578	1.79
4	5178	280	6535	1.53
5	6173	282	7701	1.3
6	7270	284	8981	1.11
7	8473	286	10382	0.963
8	9786	288	11907	0.84

Table 3.4: Batch Processing for $N = 8$, $2 \leq M \leq 8$

When these two processing schemes are compared, it is evident the sequential processing is more efficient both in terms of execution speed and cost of memory space. This is illustrated in Figure 3.8 and Figure 3.9 respectively.

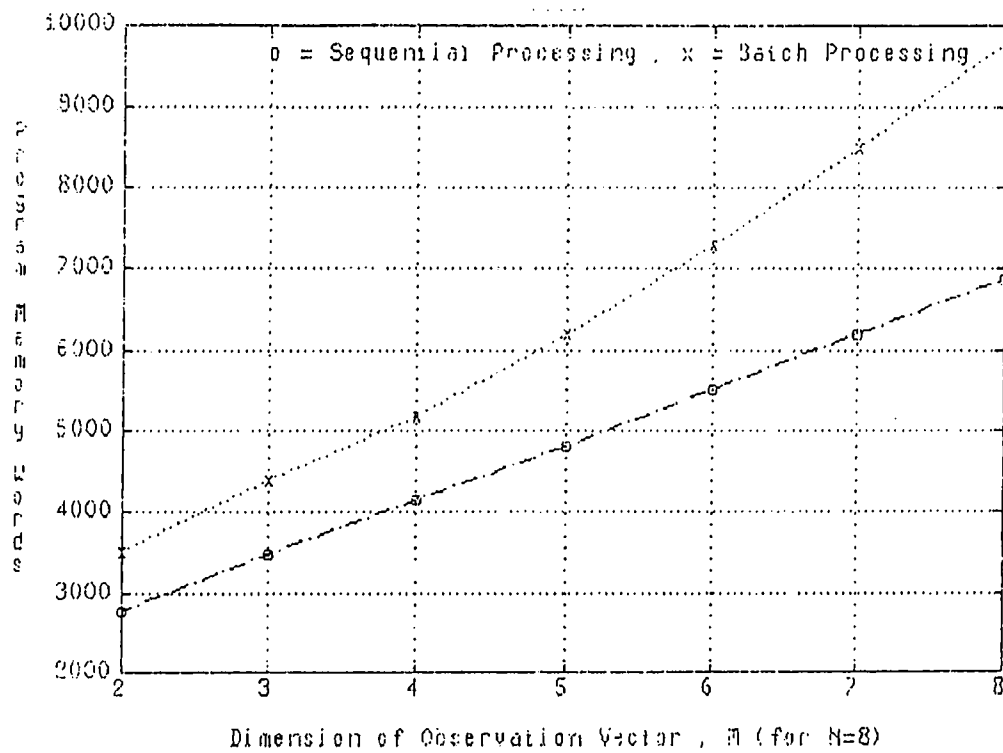


Figure 3.8: Program Memory Words Vs. Dimension of Observation Vector in Vector Observation

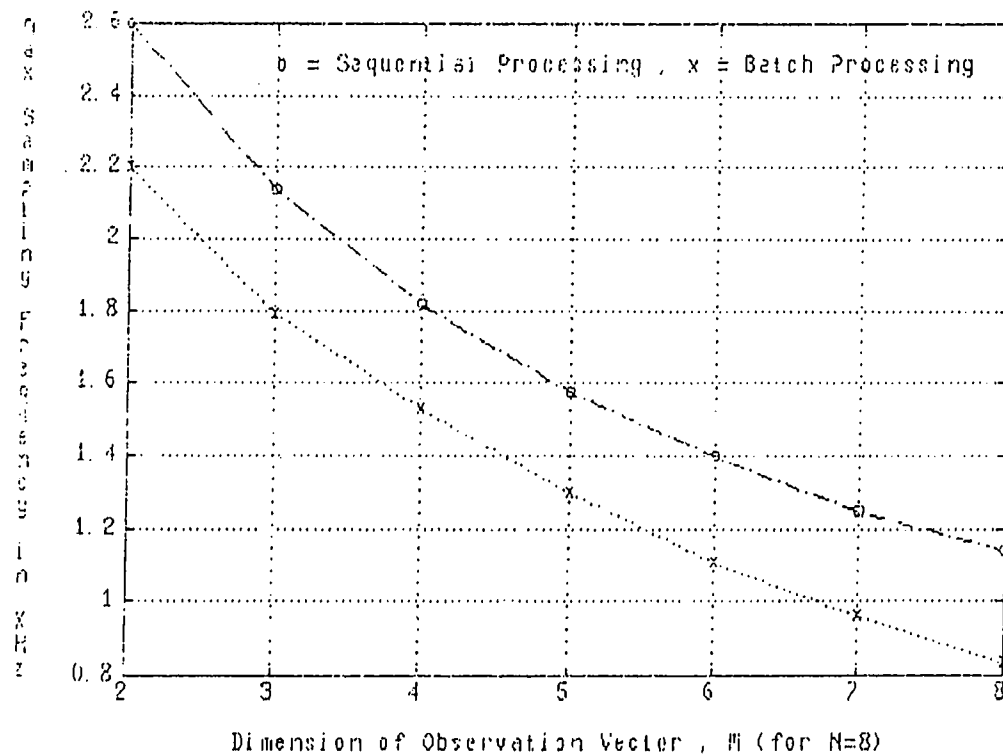


Figure 3.9: Max. Sampling Frequency Vs. Dimension of Observation Vector in Vector Observation

In fact the demand on memory space in batch processing increases at a faster rate than that of the sequential one as M increases. The apparent inefficiency of batch processing arises largely from the computation of Kalman gain matrix. To calculate Kalman gain matrix \hat{K} , we need to perform LU-factorization such that $\hat{H}\hat{P}\hat{H}^T = \hat{L}\hat{U}$ and solve \hat{K}^T from $\hat{L}\hat{U}\hat{K}^T = \hat{P}\hat{H}^T$ in two steps namely forward substitution and backward substitution. Apart from the backward substitution part, a huge number of divisions $((MN + (M - 1)^2)/2)$ has to be performed in these intermediate steps. However, the TI assembly language does not have any fast division instruction. Even the fastest possible implementation like ours requires 24 instruction cycles. Consequently, the percentage of total iteration time spent on LU-factorization and forward substitution increases substantially as M i. e. the dimension of observation vector increases. As shown in Figure 3.10, it consumes about 47% of total iteration time when $M = 8$ and $N = 8$. Nevertheless, this worse performance is better

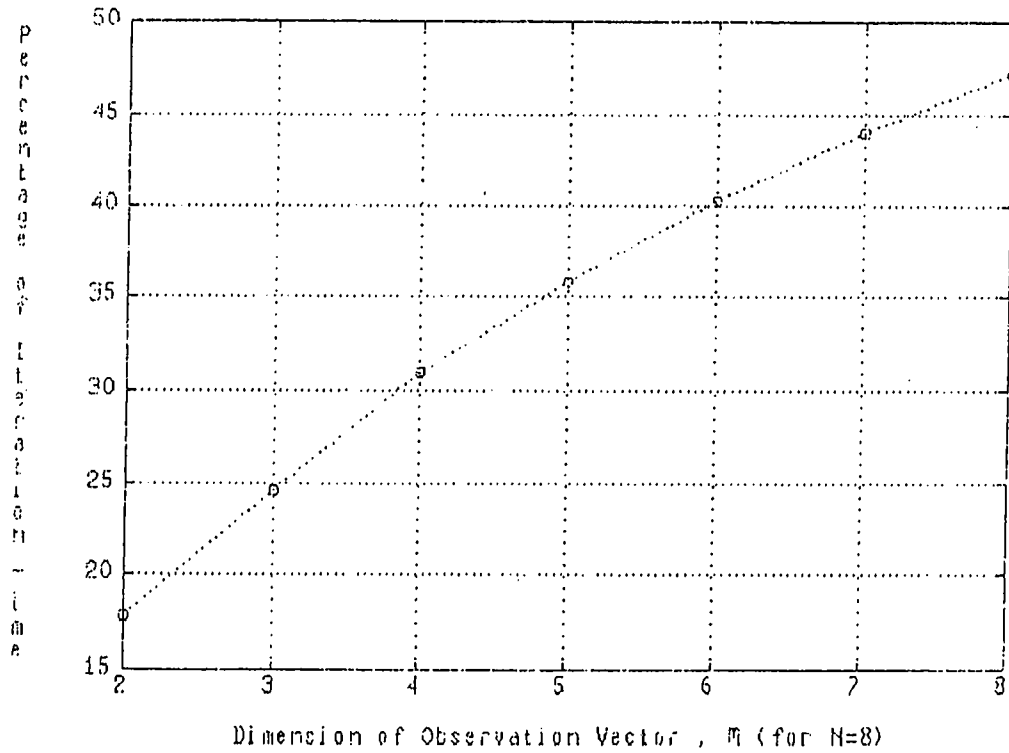


Figure 3.10: Kalman Gain Computation Time Vs. Dimension of Observation Vector

than the Choleski-decomposition-based Kalman gain computation. Although, the Choleski decomposition is numerically better conditioned, it takes twice the instruction cycle of LU-decomposition. This increase can be attributed to the fact that in Choleski decomposition involves more divisions plus computation of square-roots which is pretty demanding as far as execution speed is concerned. In LU-decomposition, backward substitution does not require any

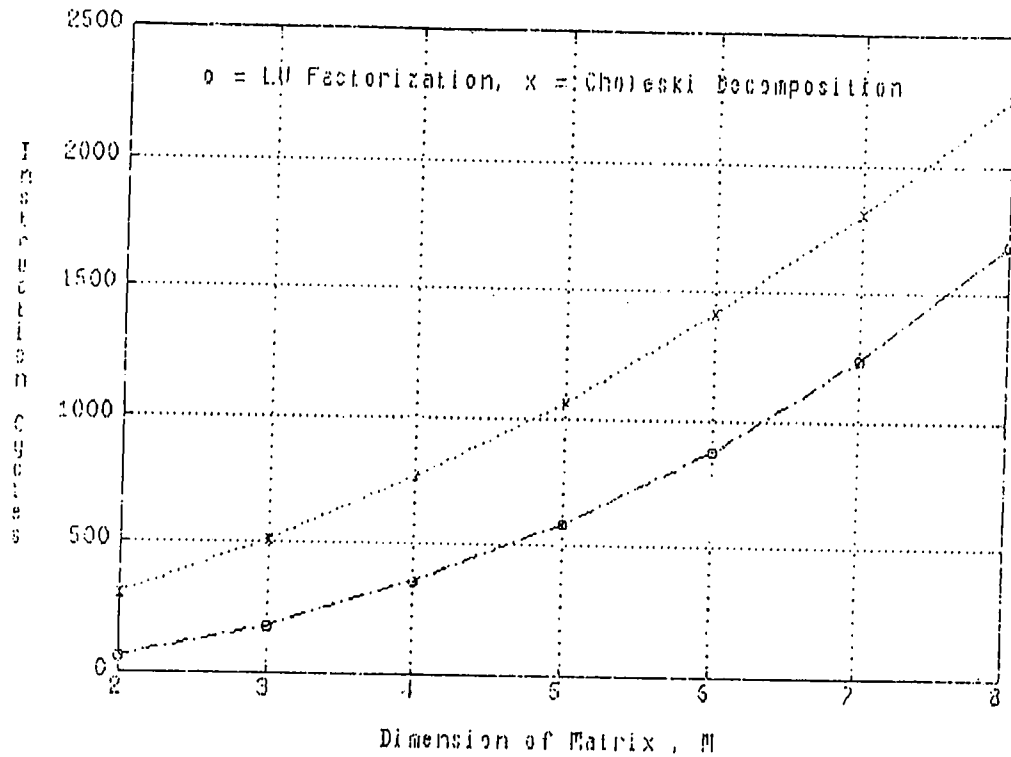


Figure 3.11: Comparison of LU and Choleski Decomposition

division since the diagonals of the upper-triangular matrix \hat{U} are all 1's. But this is not the case in Choleski-decomposition as the matrix $\hat{A} = \hat{H}\hat{P}\hat{H}^T$ is decomposed into product of \hat{A}^T and $(\hat{A}^c)^T$ where diagonals of \hat{A}^c are arbitrary. A comparison between LU and Choleski decomposition in terms instruction cycle vs. matrix size is illustrated in Figure 3.11.

Chapter 4

RESTORATION OF THE SOUND OF FLUTE EMBEDDED IN WHITE NOISE

In this chapter, a real-time application of Kalman filter namely the restoration of the sound of flute embedded in white noise is considered. Following the derivation of the state-space models of flute notes, the implementation of the filter is described. Methods are proposed on how to detect the change in model parameters. Relationships between various filter parameters and their effects on the filter gain and bandwidth are investigated.

4.1 Flute from an Engineering Perspective

From an engineering point of view, the sound of a musical instrument is nothing but a linear combination of weighted sinusoidal waveforms. The woodwind family of musical instruments which consists of flute, oboe, clarinet, piccolo, bassoon etc. produces musical sounds through open or closed cylindrical tubes with holes. The holes on the tube modify the fundamental vibrational mode of the tube resulting in different musical sounds.

The flute is an open-ended cylindrical tube with six finger holes. When all the holes are covered, the full length of the tube is utilized and the air column vibrates with its fundamental frequency. If the lowest hole is uncovered, the effective length of the tube gets shortened and the air column resonates with a higher frequency. The frequency as well as the pitch increases as the upper holes are successively uncovered. When all the holes are uncovered, the generated frequency is one note from the complete in the equal temperament scale. In the next octave, the holes are covered again and altering the blowing technique

(usually blowing harder), the second harmonic of the tube, an octave above the fundamental is generated. The process goes on in a similar manner.

4.2 State-space Modeling of Flute

About 36 notes can be generated by the flute spanning a frequency range from 260 Hz to 3.5 KHz. As with any other musical instruments, the notes produced by the flute is composed of a fundamental plus weighted sum of its harmonics. The fundamental frequencies of the notes generated by the flute are given in Appendix B.

To find the discrete-time state space-modeling of the flute , the single sinusoid case is considered first. It is shown in the Appendix C that a sinusoid of frequency ω rad/sec can be represented by the following state-space representation ,

$$\begin{bmatrix} x_1(k+1) \\ x_2(k+1) \end{bmatrix} = \begin{bmatrix} \alpha & \beta \\ -\beta & \alpha \end{bmatrix} \begin{bmatrix} x_1(k) \\ x_2(k) \end{bmatrix} \equiv \hat{\Phi} \bar{x}(k) \quad (4.1)$$

with observation equation,

$$y(k) = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1(k) \\ x_2(k) \end{bmatrix} \quad (4.2)$$

where $\alpha = \cos \omega T$ and $\beta = \sin \omega T$, T being the sampling period. This result can be extended to represent any particular note of the flute . From our experimental results, it is found that the whole spectrum of the flute notes can be represented by only three harmonics for practical purposes. Table 4.1 shows the number of harmonics for all the notes of flute.

Notes	Number of of Notes	Number of Harmonics
C_4 to F_5	19	3
G_5 to C_6	7	2
D_6 to A_6	9	1
B_6	1	0

Table 4.1: Number of Harmonics of Flute Notes

If a note has a fundamental frequency ω , then it can be represented in time domain as,

$$s(t) = a_1 \cos \omega t + a_2 \cos 2\omega t + a_3 \cos 3\omega t + a_4 \cos 4\omega t \quad (4.3)$$

where one or more terms of the last three components of the above expression can be zero. Let the state-transition matrix of the fundamental component be $\hat{\Phi}_1$ and those of its harmonics be $\hat{\Phi}_j$'s where $2 \leq j \leq 4$. The overall note can be represented as ,

$$\bar{x}(k+1) = \begin{bmatrix} \hat{\Phi}_1 & | & 0 & | & & | & 0 \\ \hline & & & & & & \\ 0 & | & \hat{\Phi}_2 & | & & | & \\ \hline & & & & & & \\ & | & & | & & | & 0 \\ \hline & & & & & & \\ 0 & | & & | & 0 & | & \hat{\Phi}_i \end{bmatrix} \bar{x}(k) \equiv \hat{\Phi} \bar{x}_k, \quad (4.4)$$

with the observation equation ,

$$\eta(k) = \begin{bmatrix} \bar{h}_1 & | & .. & | & \bar{h}_i \end{bmatrix} \bar{x}(k), \text{ where } \bar{h}_i = \begin{bmatrix} 1 & 0 \end{bmatrix} \quad (4.5)$$

for $1 \leq i \leq 4$. In reality, the fundamental frequencies of the notes played by different players vary slightly around those of Appendix B. Hence, some fictitious process noise is added which takes into account the little difference in system modeling. With this addition, the state-space model of a particular note embedded in noise can be described by,

$$\bar{x}(k+1) = \hat{\Phi} \bar{x}(k) + \bar{w}(k) \quad (4.6)$$

with noise-corrupted observation,

$$y(k) = \bar{h} \bar{x}(k) + R. \quad (4.7)$$

4.3 Using Kalman Filter to Recover the Sound of Flute

To recover sound of flute embedded in white noise, a state-space Kalman filter is implemented on TI SoftWare Development System(SWDS) board. The SWDS hardware is a PC-resident 6-layer printed circuit board that contains TI second generation digital signal processor (in this case, TMS320C25). Since the SWDS board does not have any analog interface facilities, PCL 718 data acquisition card is used for A/D and D/A conversion. Intel 8088 processor of the host computer serves as a communication link between SWDS board and data acquisition board for I/O operations. The SWDS board has 24K words of no-wait-state RAM which is accessible to the host processor via memory-mapped host port. This makes SWDS appear as a segment of 64K bytes of

memory to the host computer. In our case, the SWDS board is installed in IBM XT compatible and the RAM is divided as 16K words of program memory and 8K words of data memory. The actual interfacing between the two processor is given in more detail in Appendix D. From 8088's side, accessing TMS RAM requires that TMS320C25 must be in hold state. When TMS320C25 wants new observation data, it sends a message by resetting the external flag(XF). Prior to that, 8088 starts checking on XF bit. As soon as it senses this, it holds TMS320C25, writes new observation data, reads previously processed output data and unholds TMS320C25. TMS320C25 acknowledges new data acquisition by setting XF flag. After getting the acknowledgment signal from TMS, 8088 performs D/A and A/D conversion through PCL 718 board and keeps on waiting until XF flag is reset again. This form of polling is unidirectional since it is 8088 that always waits.

Because of the unique structure of the state-transition matrix $\hat{\Phi}$, the special macros discussed in chapter three are used to implement the Kalman filter. For an 8×8 state-transition matrix, the maximum sampling frequency of the filter can be 10.51KHz. However, 8088 puts the TMS320C25 in hold state for 6.8 microseconds resulting in an effective sampling frequency of 9.8 KHz. This sampling frequency is more than enough to avoid aliasing since the maximum

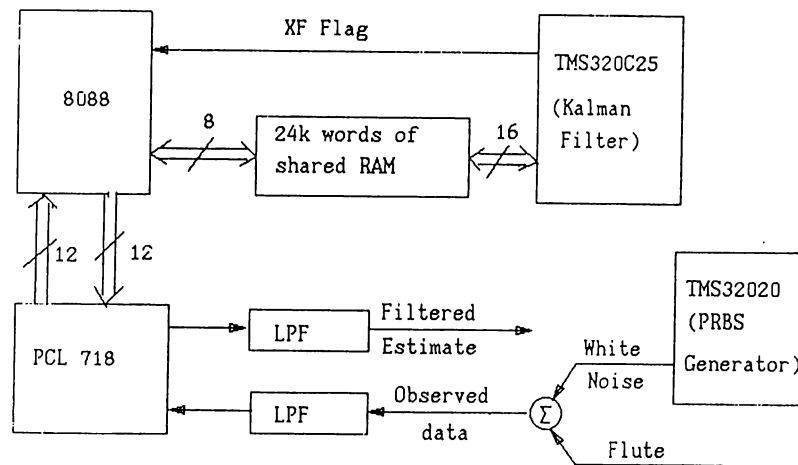


Figure 4.1: The Overall Setup for Kalman Filter Implementation

frequency of any of the flute notes does not go beyond 3.5 KHz. For generating white noise, pseudo-random-binary-sequence (PRBS) is produced from a TMS32020 board. In fact the generated PRBS is a band-limited white noise whose spectral main lobe has zero crossing at 42 KHz. The overall set-up is

shown in Figure 4.1 . It should be pointed out here that two separate programs written in two different assembly languages run simultaneously. First of all, the Kalman filter program is loaded into TMS RAM while in SWDS environment. Next step is to quit SWDS and run the 8088 program. This program enables 8088 to perform A/D or D/A operation through PCL 718 board as well as communication with TMS320C25. The schematic diagram of the inter-relationship between the two programs is given in Figure 4.2 .

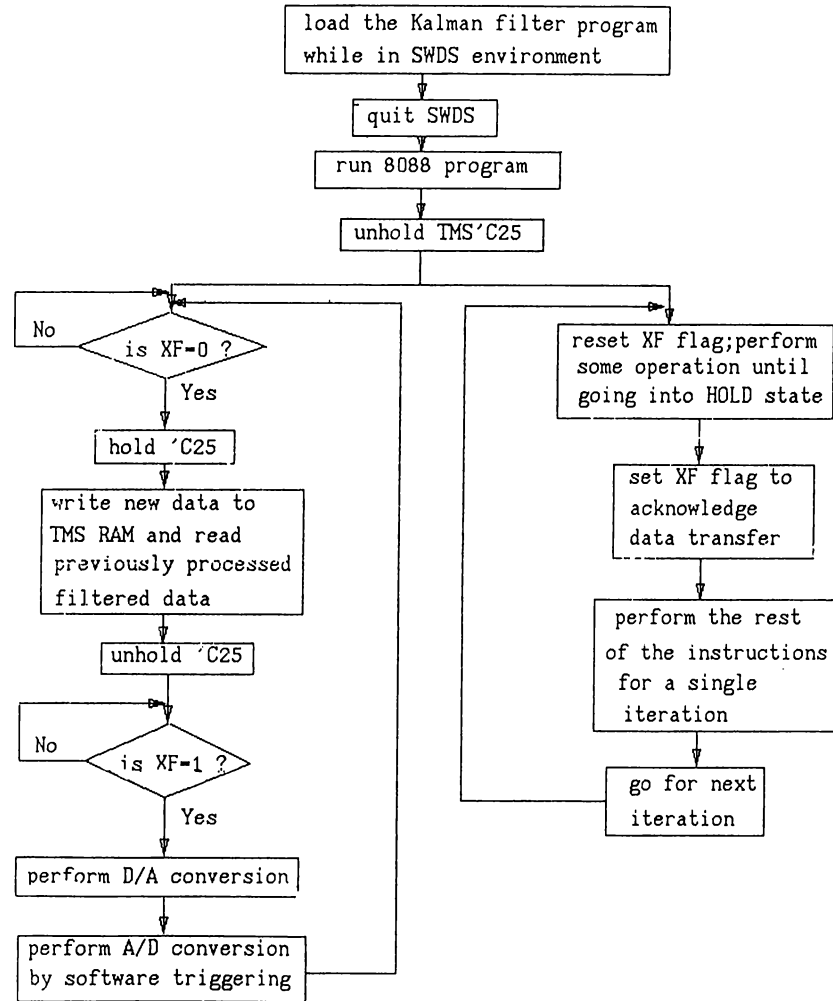


Figure 4.2: Flow-chart of 8088 and TMS Programs Running Simultaneously

Results From Real-time Operation of the Filter

The filter is run in real-time with $R = 1.25$ and $\text{diag}\{\hat{Q}\} = 0.0015I$. The model of the first note *do* (C_4 , fundamental frequency 261.63 Hz) is used as

the entries of state-transition matrix $\hat{\Phi}$. As input to the filter, four consecutive notes namely *do*, *do#*, *re* and *re#* are used. The performance of the filter is shown in Figure 4.3 and Figure 4.4. The topmost waveform shows the observed signal after the analog low-pass filter followed by filtered estimate of flute note and the actual flute note respectively. From the figure, it is evident that the Kalman filter appreciably recovers the neighboring notes of the model when some process noise is added.

4.4 Event Detection

A musical piece played on flute or any other musical instruments is composed of different notes which have different state-space models. As a result, the model parameters of the system changes in accordance with the notes being played. Addition of fictitious process noise or introduction of a forgetting factor takes this change into account and the Kalman filter works suboptimally for a few neighboring notes of the actual model. However, a single model is not obviously adequate to cover the wide span of thirty-six notes. This brings in the necessity of using atleast six to ten models if not thirty-six. Before making a decision on which model to use, the first thing to be done is to detect that a model change have taken place.

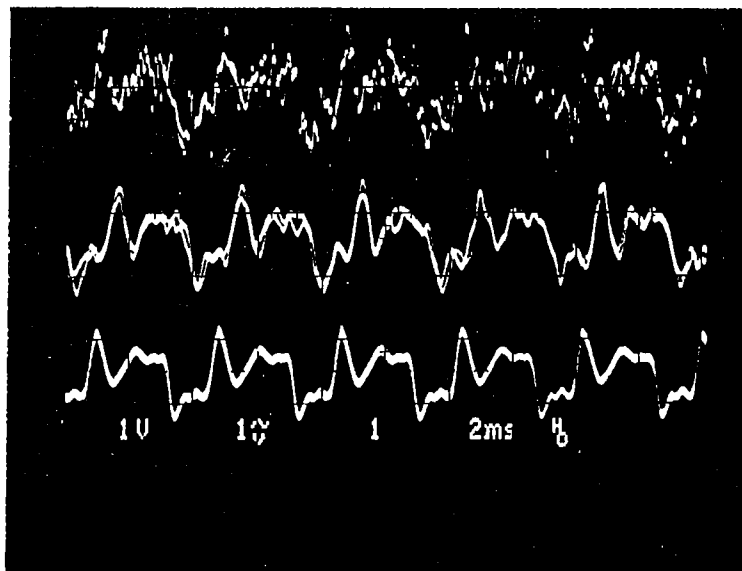
Two methods can be used to track a change in model - a situation that will be called event detection.

4.4.1 Windowed-normalized-residue(WNR)-based detection

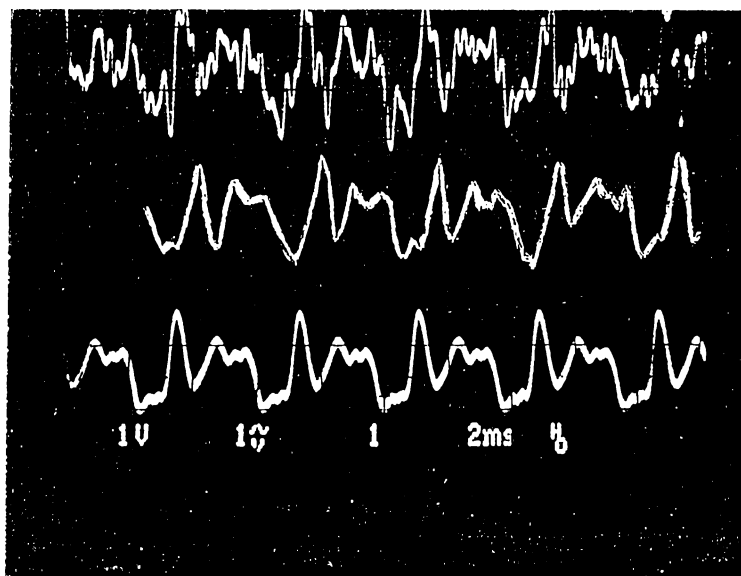
The fact that the innovation or residual sequence $\nu(k) = y(k) - \bar{h}\bar{x}(k)$ is zero mean and white with covariance $\bar{h}\hat{P}(k | k-1)\bar{h}^T + R$ can be used for event detection. The normalized-residue(NR) is defined as,

$$NR(i) = \frac{[y(i) - \bar{h}\bar{x}(i | i-1)]^2}{\bar{h}\hat{P}(i | i-1)\bar{h}^T + R} \quad (4.8)$$

An abrupt change in NR sequence indicates abnormality. To make sure that this sudden spike in NR has not come from a single erratic data, previous $(N-1)$ elements of NR are aggregated over a window of length N resulting in

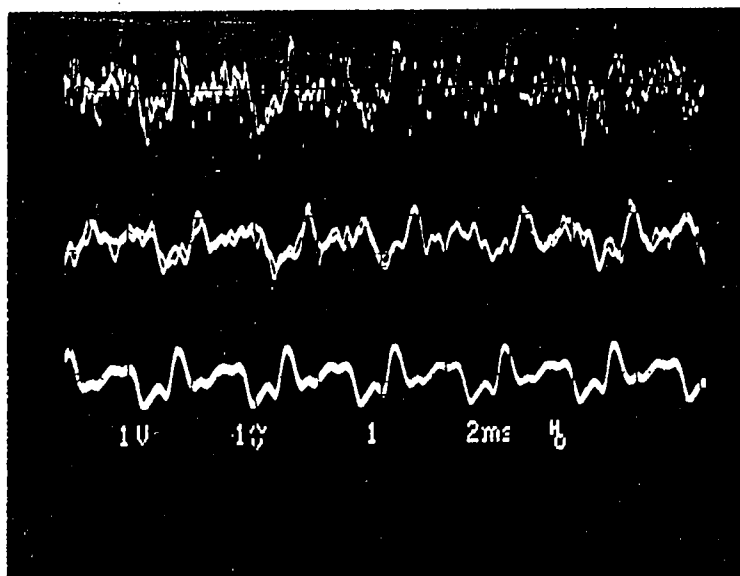


a) actual input : do(fundamental frequency=261.63 Hz)
 Model used : do(fundamental frequency=261.63 Hz)

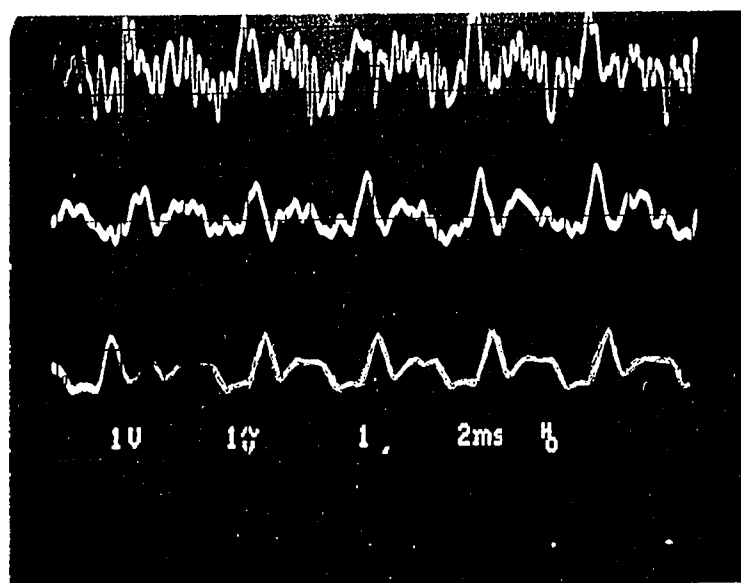


b) actual input : do#(fundamental frequency=277.18 Hz)
 Model used : do(fundamental frequency=261.63 Hz)

Figure 4.3: **Filter Performance in Real-time Operation** : the topmost waveform is observation followed by filtered output and actual note



c) actual input : Re(fundamental frequency=293.66 Hz)
 Model used : do(fundamental frequency=261.63 Hz)



d) actual input : re#(fundamental frequency=311.13 Hz)
 Model used : do(fundamental frequency=261.63 Hz)

Figure 4.4: Filter Performance in Real-time Operation : the topmost waveform is observation followed by filtered output and actual note

the windowed-normalized-residue(WNR) sequence as,

$$\text{WNR}(k) = \sum_{i=k-N+1}^k \text{NR}(i)\omega(i) \quad (4.9)$$

where $\omega(i)$'s are the window-coefficients. If WNR exceeds certain pre-defined threshold level consistently, then we can assume that a model-change has occurred.

4.4.2 Windowed-fundamental-state(WFS)-based detection

The fundamental frequencies of the notes are distinctly different from one another and their presence are reflected by the 1st(or 2nd) element in the state vector \bar{x} . Hence when an input stream of different notes are fed into the Kalman filter, x_1 will be attenuated in all cases except when the particular frequency that it represents is present in the observation. Same is true for other elements of the state vector. However, attenuation is more clearly observed in x_1 or x_2 than other entries of \bar{x} because fundamental frequencies play the most dominant roles.

The windowed-fundamental-state(WFS) sequence is defined as,

$$\text{WFS}(k) = \sum_{i=k-N+1}^k |(x_1(i | i) | \omega(i) \quad (4.10)$$

The reason for using a window is the same as WNR method. However, in this case, if WFS goes below a threshold, an event change is assumed to have taken place.

Performance Evaluation of WNR and WFS

Extensive simulations are performed to check and compare how these methods work for different notes at various noise levels. In the simulations, a particular model is chosen with 5% error and an input stream of 704 samples is generated such a way that the first 64 samples and the last 128 samples come from different notes and the middle 512 samples belong to the actual model. As for the choice of window, a triangular window is used which turns out to give the best performance among all windows. Some of the simulation results are shown in Figure 4.5 and Figure 4.6. As observed from the Figure 4.6, WNR-based detection does not work at all when signal-to-noise(SNR) ratio is low.

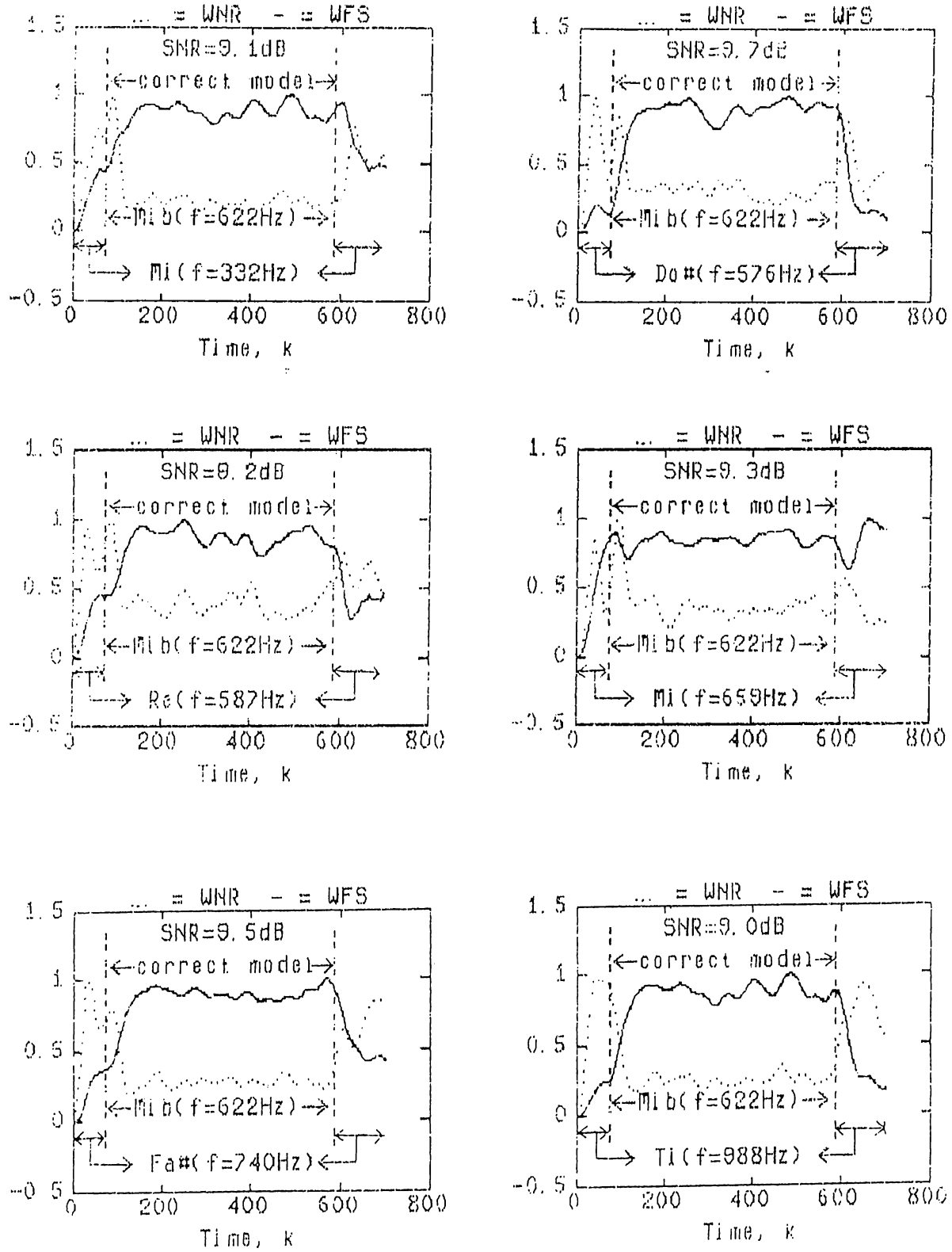


Figure 4.5: Performance of WNR and WFS methods (High SNR case: $Q = 0.002, R = 0.16, \epsilon = 1.02$)

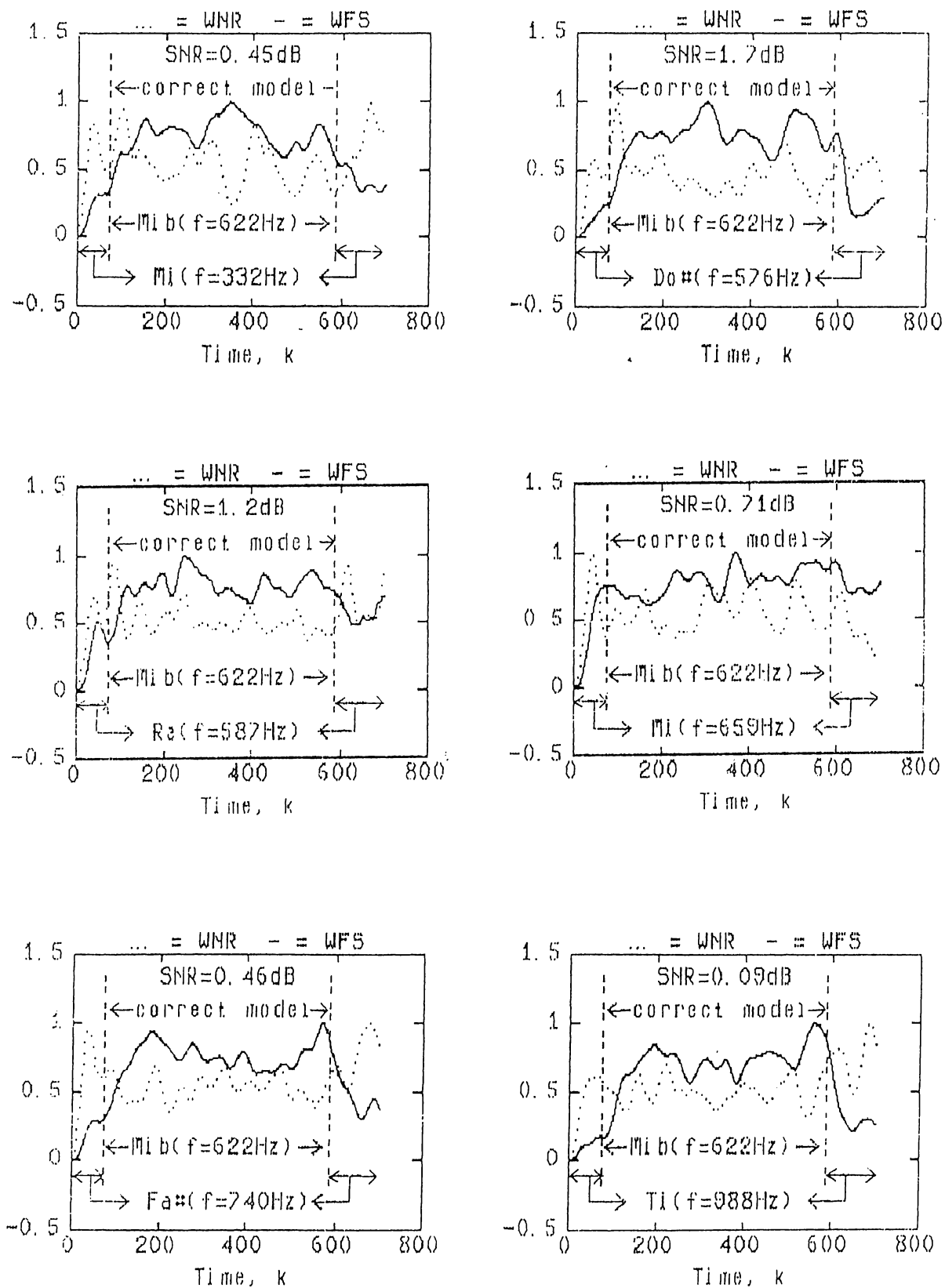


Figure 4.6: Performance of WNR and WFS methods (Low SNR case: $Q = 0.002, R = 0.16, \epsilon = 1.02$)

At high SNR, it works well only if the fundamental frequencies of the notes played consecutively are far apart. When the fundamental frequencies are closer than a certain limit, the test fails irrespective of \hat{Q} , \hat{R} and ϵ . In general WFS approach works better than the WNR method. When the fundamentals of the notes are significantly different, this method works well even at low SNR. However, when the notes have very close fundamental frequencies with comparable amplitude (e. g. two consecutive notes in equal temperament scale), this test fails as expected. This should not be considered as a drawback of WFS method because in any event, incorporating all the thirty-six models in real-time is just too luxurious to afford. A more realistic approach would be to split the notes into six to ten groups and assign a single model to each group such a way that the fundamental frequencies are sufficiently far enough. As shown in the figure, irrespective of SNR a threshold level of 0.5 can be used for WFS-based detection for this particular case.

4.5 Physical Insights about the Operation of the Kalman filter

Although the Kalman algorithm has a mathematically compact formulation, the physical insights provided by the equations are rather limited. In this section, we will try to establish an intuitive feel towards the operation of the filter.

To this end, the Kalman filter can be considered as an LTI system with time varying frequency response [13]. Although the above statement sounds contradictory (i. e. assuming time invariance and then saying that the frequency response is time dependent) what is meant is that the rate of variation of the frequency response is much slower than the signals present in the system. Once we have this conceptual model in mind, we can investigate the effects of the parameters like \hat{Q} , \hat{R} and ϵ on the frequency response of the filter. At this point, we will assume that the filter is of a band-pass or low-pass type for which we can talk about a bandwidth.

Various authors state without proof that the filter bandwidth is proportional to the Kalman gain. This statement seems to be true for a variety of practical problems although a general statement to this effect has not appeared in the literature. As a specific case, we will consider the situation where the Kalman filter is used to restore a single sinusoid from noise corrupted observation. It is shown in Appendix E that the transfer function of the filter from

the observation input $y(k)$ to the filter output $x_1(k)$ is given by,

$$H(z) = \frac{\kappa_1 + (\beta\kappa_2 - \alpha\kappa_1)z^{-1}}{1 - (2\alpha - \alpha\kappa_1 - \beta\kappa_2)z^{-1} + (1 - \kappa_1)z^{-2}} \quad (4.11)$$

where κ_1, κ_2 are the components of the steady-state Kalman gain vector \bar{k} and α, β are the elements of state-transition matrix $\hat{\Phi}$ as given by equation 4.1 with observation vector $[1 \ 0]$. To get a feel of what equation 4.11 means a mesh diagram of bandwidth versus κ_1 and κ_2 is shown in Figure 4.7. Assuming a

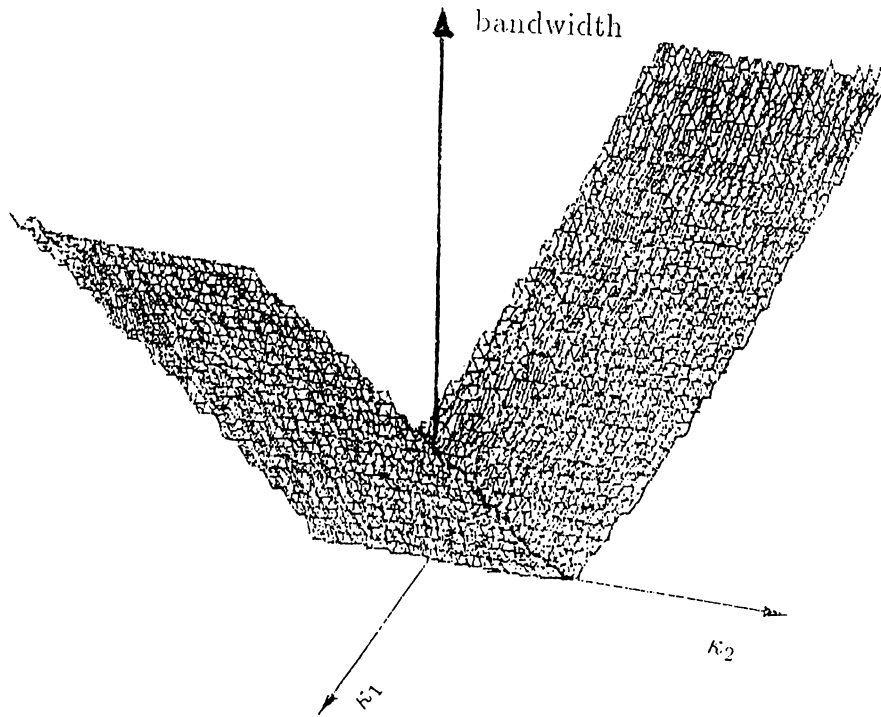


Figure 4.7: Mesh Diagram of bandwidth vs. κ_1 and κ_2 , (for $-0.4 \leq \kappa_1, \kappa_2 \leq 0.4$ and $\alpha = 0.809, \beta = 0.588$)

sampling frequency of 10KHz, the state-space model of a 1 KHz sinusoid is used for which $\alpha = 0.809$ and $\beta = 0.588$. As seen from the figure, the bandwidth of the filter increases as the magnitude of κ_1 gets larger. However, it remains virtually independent of κ_2 . This is something expected since the observer is measuring only $x_1(k)$ for which the corresponding component of the gain vector is effective.

Having shown that the Kalman gain is indeed proportional to the filter bandwidth, we will investigate how other parameters namely \hat{Q} , \hat{R} and ϵ affects

\bar{k} . Using the matrix inversion lemma, the Kalman gain can be alternatively expressed as [2] ,

$$\hat{K}(k+1) = \hat{P}(k+1 | k+1) \hat{H}_T^T R^{-1}(k+1) \quad (4.12)$$

As seen from equation 4.12, the Kalman gain is inversely proportional to the covariance matrix of observation noise. This makes sense since if the observation noise is increased the filter should decrease its bandwidth to minimize the effects of unreliable observation data.

On the other hand, the covariance matrix \hat{Q} of the process noise is directly proportional to \hat{K} . To see this, note that at steady-state the predicted and filtered estimates of \hat{P} attain almost identical values. Hence as observed from equation 4.12, increase in \hat{Q} effectively increases the Kalman gain. Furthermore from the real-time implementation of the Kalman filter to restore the sound of flute embedded in noise, it is observed that the notes lying in a small neighborhood of the actual model can be significantly recovered when some fictitious process noise is introduced. The span of the neighborhood expands as the process noise covariance \hat{Q} gets larger. Hence intuitively it is seen that increase in process noise covariance results in an equivalent increase in gain which in turn makes the filter bandwidth larger. As a result, transition time decreases, but more ripple is observed in time domain.

When using a fading memory filter, we can again heuristically conclude that increasing ϵ increases the bandwidth. This can be justified by the fact that the filter disregards most of the previous data with increasing ϵ thereby responds to the current observation more quickly i. e. transients are shortened indicating an increase in the bandwidth. In time domain, this corresponds to more ripple in transition period but less lock-in time. The lock-in time is the time required by the filter to produce steady-state output.

To check the validity of our intuitive arguments, extensive simulations are carried out for different values of \hat{Q} , \hat{R} and ϵ at various noise levels. The same 2-note input pattern discussed in the event detection section is used for this purpose.

Simulation Results

Effects of \hat{Q} , \hat{R} and ϵ on \hat{P} and \bar{k} are shown in Figure 4.8. Only the first diagonal of \hat{P} and the first component of \bar{k} are shown since all other entries of \hat{P} and \bar{k} show similar trends. It is observed that in the scalar case, \hat{P} and

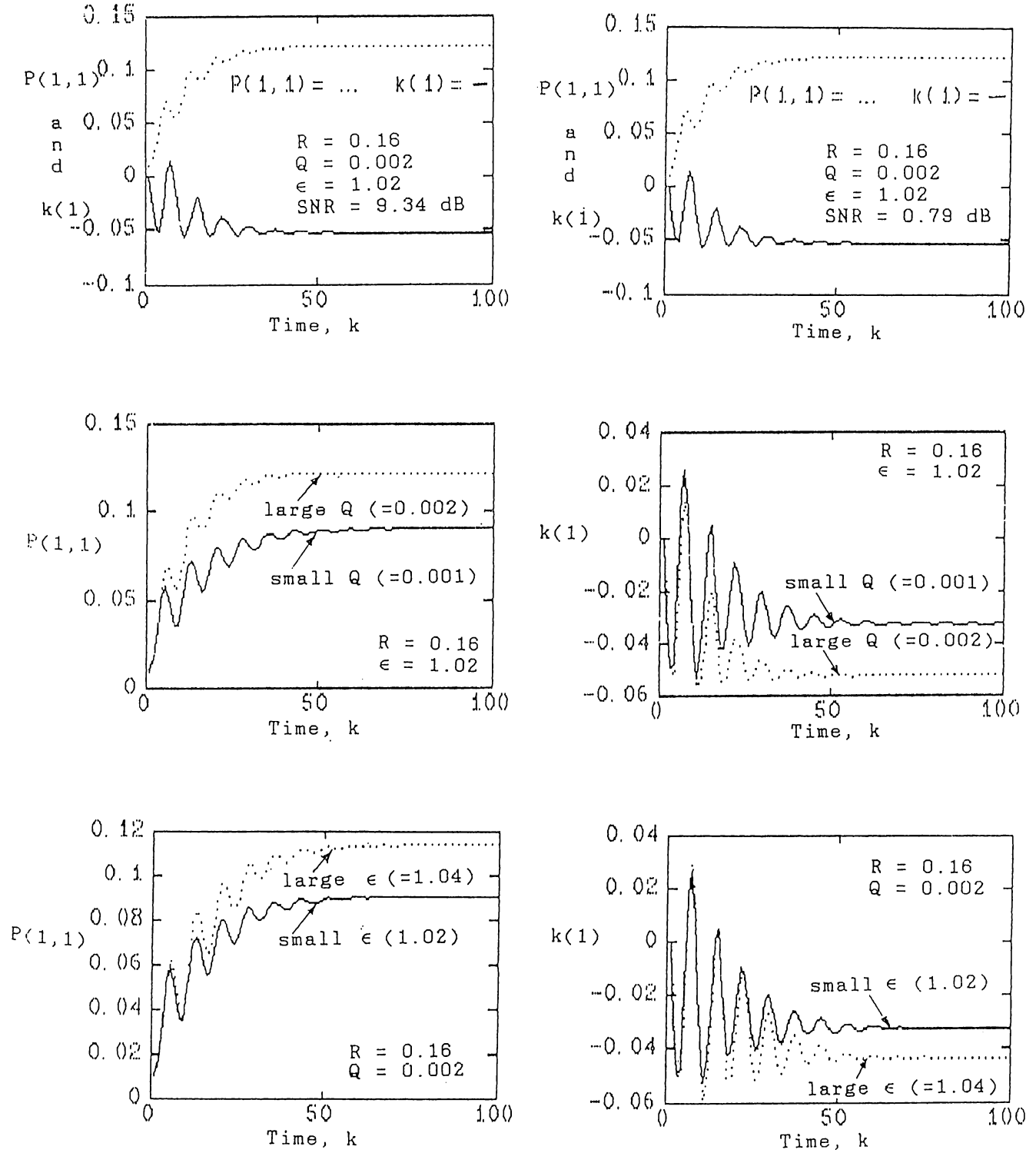


Figure 4.8: Effect of \hat{Q} , \hat{R} and ϵ on \hat{P} and \bar{k} . a) Effect of SNR for fixed \hat{Q} , \hat{R} , ϵ , b) Effect of increasing \hat{Q} , c) Effect of increasing ϵ

\bar{k} remain insensitive to SNR as long as Q/R is kept constant. For fixed R , increase in either Q or ϵ result in larger magnitude of steady-state \hat{P} and \bar{k} .

The time-domain behavior of the filter is illustrated in Figure 4.9, Figure 4.10 and Figure 4.11. In this case we investigate how R , Q and ϵ affect the transient response of the filter output when a model-change is detected and the correct model is put into the filter. As observed from Figure 4.9, increase in R results in a decrease in ripple in transition period. This is expected since increase in R is accompanied by an equivalent decrease in Kalman gain \bar{k} which makes the filter bandwidth narrower. Increase in either Q or ϵ makes the transition smaller but introduces more ripple. Another important observation from Figure 4.11 is that increase in epsilon reduces the lock-in time. All these results comply with our earlier arguments on the effects of \hat{Q} , \hat{R} and ϵ on filter gain and bandwidth.

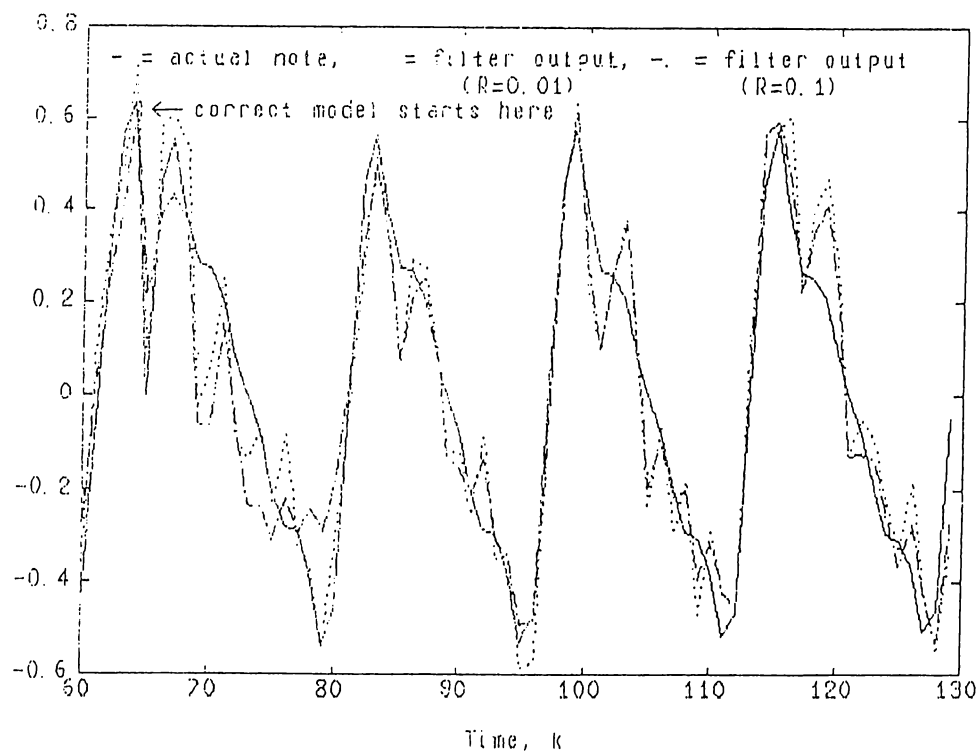
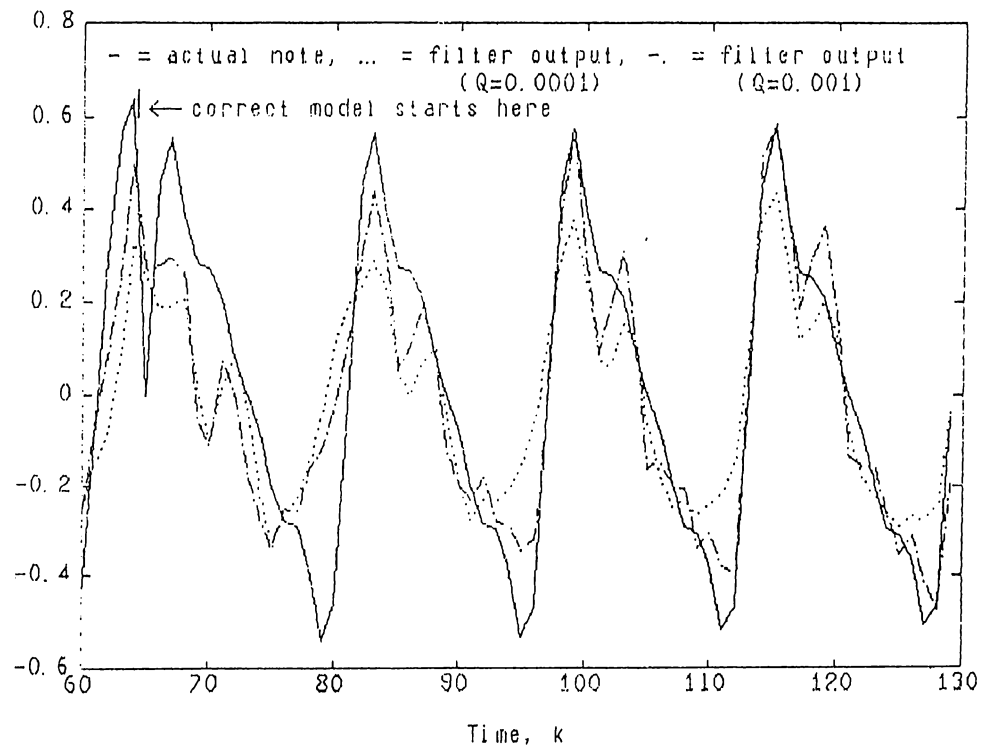
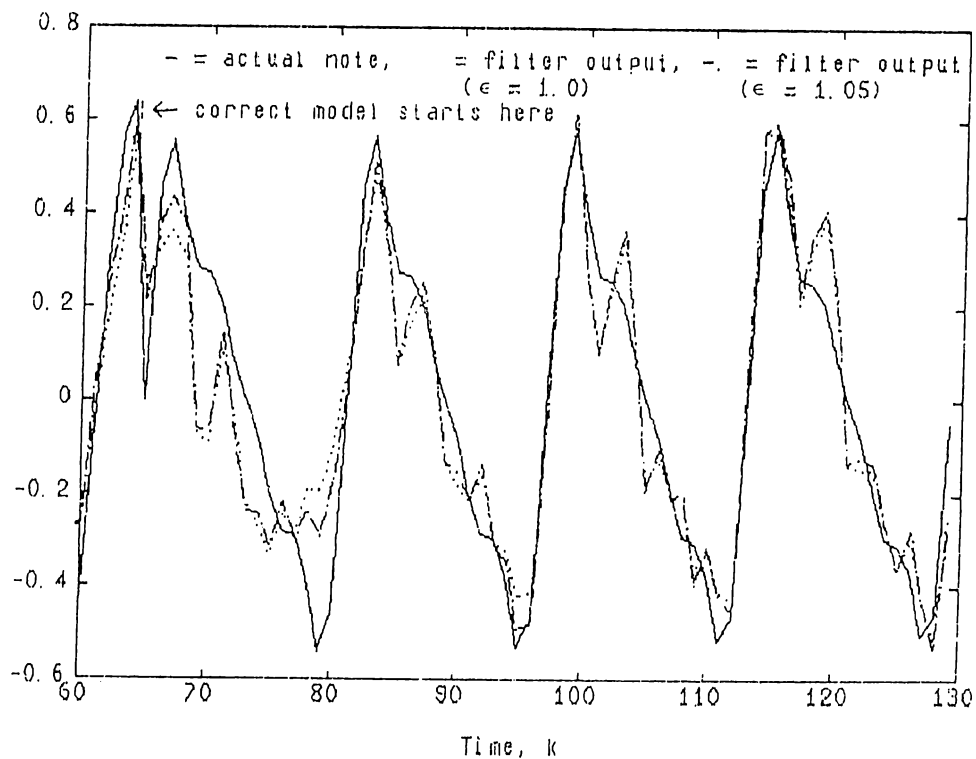


Figure 4.9: Effect of \hat{R} on the Transient Response of the Filter Output

Figure 4.10: Effect of \hat{Q} on the Transient Response of the Filter OutputFigure 4.11: Effect of ϵ on the Transient Response of the Filter Output

Chapter 5

CONCLUSION

Despite the fact that the Kalman filter has been widely used in various control-oriented problems, it has found very limited applications in real-time digital signal processing. The apparent reasons can be blamed on the large computational burden associated with the real-time implementation of the filter as well as some sort of unfamiliarity of digital signal processing people with the filter itself. Even the introduction of fast digital signal processors does not seem to have changed the situation much. In this thesis, a very general user-friendly software written in TI assembly language is introduced for the implementation of Kalman filter on TI TMS320C25 digital signal processor. It has been shown that virtually all the Kalman filter algorithms can be easily and efficiently realized using our macro-based software library. As compared to the only digital signal processor-based implementation reported so far [13] our implementation is much more faster and user-friendly.

Until now all the studies carried out on the relationship between the state size and other parameters such as memory requirements, execution speed etc. are based on the assumption that the multiplication operation takes as much as thrice the time taken by addition. However, these expressions do not hold when the filter is implemented on a digital signal processor. Unlike the general-purpose microcomputers, these processors have single cycle multiplication instruction which takes exactly the same time as addition. Our studies give symbolic relationship between various parameters and state size for TMS320C25 implementation which in general holds for other digital signal processing chips as well.

As a specific application, restoration of the sound of flute embedded in white noise is considered. The state-space models for flute notes are derived and the real-time Kalman filter is implemented using state-space approach. To

make the filter somewhat adaptive to change in model parameters, a forgetting factor is introduced which emphasizes more recent data. Apart from running the filter real-time, extensive simulations are carried out to find the effects of \hat{Q} and \hat{R} on filter gain \hat{K} and the filter bandwidth. Illustrative examples are supported with proofs which show that addition of fictitious process noise or introduction of a forgetting factor simply increases the band-width of the filter - a fact that is not obvious at all from time-domain expressions. While running the filter real-time, it is observed that the addition of some process noise enables the filter not only recover the note for which the model is used but some other neighboring notes as well. Hence it can be assumed that all the thirty-six notes generated by the flute can be recovered from noise using only six to ten models. As for the detection of a change in model, a method is proposed which works remarkably well for this specific case.

Various interesting works can be performed as a continuation of this thesis. In our case, white noise is considered as the corrupting agent. Any other disturbance such as background music can be considered as an extension of our case. However, the model of the noise irrespective of its origin needs to be known. Based on this rudimentary work, further serious works like the extraction of flute from background music (such as other instruments in an orchestra) can be carried out. Another extension of our studies would be model identification. We have proposed a very effective method for the detection of model change. Further researches can be done on how to make a decision on which model to use when an event change is detected.

Appendix A

Derivation of Fading Memory Filter

In the fading memory filter, it is assumed that the states of the system can not be accurately represented by the usual state-space model over an infinite interval of time. It is more appropriate to express the states at time N with the following state-space representation,

$$\left. \begin{aligned} \bar{x}_N(k+1) &= \hat{\Phi} \bar{x}_N(k) + \bar{w}(k) \\ \bar{y}(k) &= \hat{H} \bar{x}_N(k) + \bar{v}(k) \end{aligned} \right\} (A.1)$$

To put more emphasis on the recent data and state estimates, the initial states and the noise covariances are assumed to be,

$$\left. \begin{aligned} E\{\bar{w}_N(i)\bar{w}_N^T(j)\} &= \hat{Q}_N(i)\delta_{ij} = \hat{Q}(i)e^{\sum_{k=i+1}^N \sigma_k} \delta_{ij} \\ E\{\bar{v}_N(i)\bar{v}_N^T(j)\} &= \hat{R}_N(i)\delta_{ij} = \hat{R}(i)e^{\sum_{k=i}^N \sigma_k} \delta_{ij} \\ E\{\delta \bar{x}_N(0)\delta \bar{x}_N^T(0)\} &= \hat{P}_N(0|-1) = \hat{K}_{xx}(0)e^{\sum_{k=0}^N \sigma_k} \delta_{ij} \end{aligned} \right\} (A.2)$$

Let us define $\hat{P}_N(k+1|k)$ and $\hat{P}_N(k+1|k+1)$ by the following transformations,

$$\left. \begin{aligned} \hat{P}_N(k+1|k) &= \hat{P}(k+1|k)e^{\sum_{i=k+1}^N \sigma_i} \\ \hat{P}_N(k+1|k+1) &= \hat{P}(k+1|k)e^{\sum_{i=k+1}^N \sigma_i} \end{aligned} \right\} (A.3)$$

Since σ_i 's are known, the transformation is invertible. The standard Kalman filter algorithm for the state-space representation of (A.1) can be expressed as,

$$\begin{aligned} \bar{x}_N(k+1|k) &= \hat{\Phi} \bar{x}_N(k|k) \\ \hat{P}_N(k+1|k) &= \hat{P}_N(k|k)\hat{\Phi} \hat{P}_N(k|k)^T + \hat{Q}_N(k) \\ \hat{K}_N(k+1) &= \hat{P}_N(k+1|k)\hat{H}^T[\hat{H} \hat{P}_N(k+1|k)\hat{H}^T + \hat{R}_N(k+1)]^{-1} \\ \bar{x}_N(k+1|k+1) &= \bar{x}_N(k+1|k) - \hat{K}_N(k+1)[\hat{H} \bar{x}_N(k+1|k) - \bar{y}(k+1)] \\ \hat{P}_N(k+1|k+1) &= [I - \hat{K}_N(k+1)\hat{H}]\hat{P}_N(k+1|k) \end{aligned}$$

Substituting (A.2) and (A.3) into these equations and canceling the common terms we get,

$$\begin{aligned}
\bar{x}_N(k+1 | k) &= \hat{\Phi} \bar{x}_N(k | k) \\
\hat{P}(k+1 | k) &= \hat{P}(k | k) \hat{\Phi} \hat{P}(k | k)^T e^{\sigma_k} + \hat{Q}(k+1) \\
\hat{K}_N(k+1) &= \hat{P}(k+1 | k) \hat{H}^T [\hat{H} \hat{P}_N(k+1 | k) \hat{H}^T + \hat{R}_N(k+1)] \\
\bar{x}_N(k+1 | k+1) &= \bar{x}_N(k+1 | k) - \hat{K}_N(k+1) [\hat{H} \bar{x}_N(k+1 | k) - \bar{y}(k+1)] \\
\hat{P}(k+1 | k+1) &= [I - \hat{K}_N(k+1) \hat{H}] \hat{P}(k+1 | k)
\end{aligned}$$

Note that all steps except the prediction part of error covariance matrix are the same as the usual Kalman filter. The term e^{σ_k} can be represented by any number $\epsilon > 1$.

Appendix B

Fundamental Frequencies of Flute Notes

In the following table, the fundamental frequencies of all the thirty-six notes generated by flute is given in equal temperament scale [22].

<i>Major Scale</i>	<i>Note</i>	<i>Frequency in Hz</i>
C_4	<i>do</i>	261.63
	<i>do#</i>	277.18
D_4	<i>re</i>	293.66
	<i>re#</i>	311.13
E_4	<i>mi</i>	329.63
F_4	<i>fa</i>	349.23
	<i>fa#</i>	369.99
G_4	<i>sol</i>	392.00
	<i>sol#</i>	415.3
A_4	<i>la</i>	440.0
	<i>la#</i>	466.16
B_4	<i>ti</i>	493.88
C_5	<i>do</i>	523.25
	<i>do#</i>	554.37
D_5	<i>re</i>	587.33
	<i>re#</i>	622.25
E_5	<i>mi</i>	659.26
F_5	<i>fa</i>	698.46
	<i>fa#</i>	739.99
G_5	<i>sol</i>	783.99
	<i>sol#</i>	830.61
A_5	<i>la</i>	880.00
	<i>la#</i>	923.33
B_5	<i>ti</i>	987.77

APPENDIX B. FUNDAMENTAL FREQUENCIES OF FLUTE NOTES 60

<i>Major Scale</i>	<i>Note</i>	<i>Frequency in Hz</i>
C_6	<i>do</i>	1046.5
	<i>do#</i>	1108.7
D_6	<i>re</i>	1174.7
	<i>re#</i>	1244.5
E_6	<i>mi</i>	1318.5
F_6	<i>fa</i>	1396.9
	<i>fa#</i>	1480.0
G_6	<i>sol</i>	1568.0
	<i>sol#</i>	1661.2
A_6	<i>la</i>	1760.0
	<i>la#</i>	1864.7
B_6	<i>ti</i>	1975.5

Appendix C

State-space model of a sinusoid

A process having harmonic oscillation with frequency ω_0 rad/sec satisfies the differential equation,

$$\ddot{y}(t) + \omega_0^2 y(t) = 0 \quad (\text{C.1})$$

whose solution is given by,

$$y(t) = y(0) \cos \omega_0 t + (\dot{y}(0)/\omega_0) \sin \omega_0 t \quad (\text{C.2})$$

In order to get the state-space representation, we define two state variables $x_1(t)$ and $x_2(t)$ such that $x_1(t) = y(t)$ and $x_2 = \dot{y}(t)$. With these variables, the state-space form in time domain is represented as,

$$\begin{aligned} \begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \end{bmatrix} &= \begin{bmatrix} 0 & 1 \\ -\omega_0^2 & 0 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} \\ &= \hat{A} \bar{x}(t) \end{aligned} \quad (\text{C.3})$$

with observation equation given by,

$$y(t) = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} \quad (\text{C.4})$$

The solution of the state-space representation can be written as,

$$\bar{x}(t) = \hat{\Phi}(t, \tau) \bar{x}(\tau) \quad (\text{C.5})$$

where the state transition matrix $\hat{\Phi}(t, \tau)$ is given by the expression $\hat{\Phi}(t, \tau) = e^{\hat{A}(t-\tau)}$, $0 \leq \tau \leq t$. Since,

$$e^{\hat{A}t} = \mathcal{L}^{-1}\{[SI - \hat{A}]^{-1}\} \quad (\text{C.6})$$

$$\begin{aligned}
\Rightarrow e^{\hat{A}t} &= \mathcal{L}^{-1} \left\{ \left[\begin{array}{cc} s & -1 \\ \omega_0^2 & s \end{array} \right]^{-1} \right\} \\
&= \begin{bmatrix} \cos \omega_0 t & (1/\omega_0) \sin \omega_0 t \\ -\omega_0 \sin \omega_0 t & \cos \omega_0 t \end{bmatrix}
\end{aligned} \tag{C.7}$$

$$\Rightarrow \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} \cos \omega_0(t - \tau) & (1/\omega_0) \sin \omega_0(t - \tau) \\ -\omega_0 \sin \omega_0(t - \tau) & \cos \omega_0(t - \tau) \end{bmatrix} \begin{bmatrix} x_1(\tau) \\ x_2(\tau) \end{bmatrix} \tag{C.8}$$

for $0 \leq \tau \leq t$. For normalization purpose, we define two new variable $\tilde{x}_1(t)$ and $\tilde{x}_2(t)$ such that $\tilde{x}_1(t) = x_1(t)$ and $\tilde{x}_2(t) = (1/\omega_0)x_2(t)$. In doing so, we obtain,

$$\begin{bmatrix} \tilde{x}_1(t) \\ \tilde{x}_2(t) \end{bmatrix} = \begin{bmatrix} \cos \omega_0(t - \tau) & \sin \omega_0(t - \tau) \\ -\sin \omega_0(t - \tau) & \cos \omega_0(t - \tau) \end{bmatrix} \begin{bmatrix} \tilde{x}_1(\tau) \\ \tilde{x}_2(\tau) \end{bmatrix} \tag{C.9}$$

To convert from continuous domain to discrete domain, a sampling frequency $f_s = 1/T$ is assumed. By setting $t = (k+1)T$ and $\tau = kT$, we get the following state-space representation in discrete domain,

$$\begin{bmatrix} \tilde{x}_1(k+1) \\ \tilde{x}_2(k+1) \end{bmatrix} = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} \tilde{x}_1(k) \\ \tilde{x}_2(k) \end{bmatrix} \tag{C.10}$$

where $\theta = \omega_0 T = 2\pi f_0/f_s$. Since we are interested in the variable $x_1(k)$ only and $\tilde{x}_1(k) = x_1(k)$, the state-space representation for a single sinusoid can be written as,

$$\begin{bmatrix} x_1(k+1) \\ \tilde{x}_2(k+1) \end{bmatrix} = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x_1(k) \\ \tilde{x}_2(k) \end{bmatrix} = \hat{\phi} \bar{x}(k) \tag{C.11}$$

with observation equation,

$$y(k) = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1(k) \\ \tilde{x}_2(k) \end{bmatrix} \tag{C.12}$$

Appendix D

Interfacing Between 8088 and SWDS board

The SWDS board is installed in IBM XT compatible and E segment is chosen as the segment (64K bytes) of PC memory that SWDS occupies. The 24K no-wait state RAM of SWDS board is divided as 16K of program memory and 8K of data memory. In the 8088 two 16-bit words are used to represent the 20-bit-long address. The first word is called segment address which has an implied zero tacked to its end. The second word called offset address gives the relative part of the effective 20-bit address. The correspondence between the 16-bit address of TMS and the 20-bit effective address of 8088 is illustrated in Table D.1. It should be noted here that in this table, the offset address is added

<i>Memory</i>	<i>16-bit TMS address</i>	<i>20-bit 8088 address</i>
16K program memory	0	E2000h
	2FFFh	EBFFFh
8K data memory	0	EA000h
	1FFFh	EDFFFh

Table D.1: Addresses of Shared Memory as Seen by TMS and 8088

to the segment address to denote the effective address of 8088. For example, as far as 8088 is concerned, a 16-bit data written in 400h (as seen from SWDS side) of SWDS data RAM appears in the locations EA800h (E000h:A800h) and EA801h (E000h:A801h) as low and high bytes respectively. If the RAM is divided in some other way, the starting locations of offset addresses as seen from 8088 would be different.

The memory locations EE000h and EE001h are the control registers to

APPENDIX D. INTERFACING BETWEEN 8088 AND SWDS BOARD 64

accomplish communication between 8088 and TMS320C25. When a read operation is performed, EE000h acts as a status register whose contents are :

7	6	5	4	3	2	1	0
$\overline{\text{HOLDA}}$	$\overline{\text{ERS}}$	$\overline{\text{BP}}$	XF	RBIO	$\overline{\text{INT2}}$	$\overline{\text{INT1}}$	$\overline{\text{INT0}}$

The same register acts as a control register with the following bits when a write operation is carried out.

7	6	5	4	3	2	1	0
$\overline{\text{CHOLD}}$	$\overline{\text{SWRST}}$	$\overline{\text{CTEST}}$	$\overline{\text{BPACT}}$	$\overline{\text{CLRBP}}$	$\overline{\text{CLKSEL}}$	MC1	MC2

More information about these register and its contents can be found in the chapter 5 of [23].

In order to know whether XF flag is set or reset, 8088 reads the EE000h location and checks the XF bit. To write data into the RAM or to be able to read data from RAM, TMS320C25 must be put into hold state. This is accomplished by writing the byte 6Ah to EE000h. To unhold the TMS320C25, 8088 writes the byte EAh again to EE000h. It should be noted that choice of MC1=1 and MC2=1 conforms to the choice of RAM division (16K program and 8K data RAM). In this implementation, no interrupts whatsoever is used. Use of interrupts which is done by writing necessary byte to EE000h is not recommended since SWDS software takes interrupts as breakpoint event and causes a lot of problems.

Appendix E

Role of Filter Gain on Bandwidth

The prediction and correction steps of the system states are given by,

$$\bar{x}(k+1 | k) = \hat{\Phi} \bar{x}(k | k) \quad (\text{E.1})$$

and

$$\bar{x}(k+1 | k+1) = \bar{x}(k+1 | k) + \bar{k}[y(k+1) - \bar{h}\bar{x}(k+1 | k)] \quad (\text{E.2})$$

where \bar{k} is the steady-state Kalman gain.

$$\begin{aligned} \Rightarrow \bar{x}(k+1 | k+1) &= \hat{\Phi} \bar{x}(k+1 | k) + \bar{k}[y(k+1) - \bar{h}\bar{x}(k+1 | k)] \\ &= [I - \bar{k}\bar{h}] \hat{\Phi} \bar{x}(k | k) + \bar{k}y(k+1) \\ \Rightarrow \bar{x}(k | k) &= [I - \bar{k}\bar{h}] \hat{\Phi} \bar{x}(k-1 | k-1) + \bar{k}y(k) \end{aligned} \quad (\text{E.3})$$

Taking Z-transform on both sides of the equation E.3 yields,

$$\bar{X}(z) = [I - \bar{k}\bar{h}] \hat{\Phi} z^{-1} \bar{X}(z) + \bar{k}Y(z) \quad (\text{E.4})$$

where $\bar{X}(z)$ and $Y(z)$ are the Z-transform of $\bar{x}(k)$ and $y(k)$ respectively.

$$\begin{aligned} \Rightarrow \bar{X}(z) &= [I - (I - \bar{k}\bar{h}) \hat{\Phi} z^{-1}]^{-1} \bar{k}Y(z) \\ &= \hat{\Phi}^* \bar{k}Y(z) \end{aligned} \quad (\text{E.5})$$

where $\hat{\Phi}^* = [I - (I - \bar{k}\bar{h}) \hat{\Phi} z^{-1}]^{-1}$. Since we are interested only in $x_1(k)$ and $x_1(k) = [1 \ 0] \bar{x}(k)$

$$\begin{aligned} \Rightarrow X_1(z) &= [1 \ 0] \bar{X}(z) \\ &= [1 \ 0] \hat{\Phi}^* \bar{k}Y(z) \end{aligned} \quad (\text{E.6})$$

Hence the transfer function from observation input $y(k)$ to filter output $x_1(k)$ is given by,

$$H(z) = X_1(z)/Y(z) = [1 \ 0]\hat{\Phi}^*\bar{k} \quad (\text{E.7})$$

Impulse response is found by setting $y(k) = \delta(k)$ i. e. $Y(z) = 1$. In doing so, the Z -transform of the impulse response can be expressed as,

$$H(z) = [1 \ 0]\hat{\Phi}^*\bar{k} \quad (\text{E.8})$$

Let us consider the situation when a single sinusoid of frequency f is concerned. If a sampling frequency of f_s is assumed then, the state-transition matrix $\hat{\Phi}$ and observation vector \bar{h} are given by,

$$\hat{\Phi} = \begin{bmatrix} \alpha & \beta \\ -\beta & \alpha \end{bmatrix} \quad (\text{E.9})$$

and

$$\bar{h} = [1 \ 0]. \quad (\text{E.10})$$

where $\alpha = \cos(2\pi f/f_s)$, and $\beta = \sin(2\pi f/f_s)$.

$$\begin{aligned} \Rightarrow \hat{\Phi}^* &= [I - (I - \bar{k}\bar{h})\hat{\Phi}z^{-1}]^{-1} \\ &= \left[\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} \kappa_1 \\ \kappa_2 \end{bmatrix} [1 \ 0] \right) \begin{bmatrix} \alpha & \beta \\ -\beta & \alpha \end{bmatrix} z^{-1} \right]^{-1} \\ &= \begin{bmatrix} 1 - \alpha(1 - \kappa_1)z^{-1} & -\beta(1 - \kappa_1)z^{-1} \\ (\alpha\kappa_2 + \beta)z^{-1} & 1 - (\alpha - \beta\kappa_2)z^{-1} \end{bmatrix}^{-1} \end{aligned} \quad (\text{E.11})$$

Substituting $\hat{\Phi}^*$ in equation E.8 gives,

$$H(z) = [1 \ 0] \begin{bmatrix} 1 - \alpha(1 - \kappa_1)z^{-1} & -\beta(1 - \kappa_1)z^{-1} \\ (\alpha\kappa_2 + \beta)z^{-1} & 1 - (\alpha - \beta\kappa_2)z^{-1} \end{bmatrix}^{-1} \begin{bmatrix} \kappa_1 \\ \kappa_2 \end{bmatrix} \quad (\text{E.12})$$

After some tedious steps, the equation E.12 simplifies to a

$$H(z) = \frac{\kappa_1 + (\beta\kappa_2 - \alpha\kappa_1)z^{-1}}{1 - (2\alpha - \alpha\kappa_1 - \beta\kappa_2)z^{-1} + (1 - \kappa_1)z^{-2}} \quad (\text{E.13})$$

Appendix F

The Macro Library

F.1 General Macros

```
*****
**                               MACRO : MakeInit                               **
*****
***** MAKE NECESSARY INITIALIZATIONS FOR THE MACROS *****
*****
* This macro named " MakeInit " makes proper initialization required *
* for using the macros in Q15-based numerical computations.This must *
* be the first macro in the main program. The macro initializes the *
* following parameters :                                           *
*   Data Pointer(DP) = 0 , Sign-Extension Mode(SXM) = 1          *
*   Overflow Mode(OVM) = 1 , Product Mode(PM) = 1                *
*   Auxiliary Register 0(ARO) = 10h                               *
* The above parameters must be restored if the user changes any of *
* these in between two macros.                                     *
*****
*
MakeInit $MACRO
LDPK 0 ;point to page 0 to use it as scratch pad for other macros.
SSXM ;set sign-extension mode.
SOVM ;set overflow mode.
SPM 1 ;set PM = 1 for Q15 operation.
LARK ARO,10h ;make content of ARO = 10h for indexing of matrices.
$ENDM ;end of the macro.

*****
**                               MACRO : ScalaorS                               **
*****
*** ADD TO OR SUBTRACT FROM ACCH ANOTHER # FROM A SPECIFIED LOCATION ***
*****
* Depending on the macro parameter OPTION, this macro named " ScalaorS *
* Location,OPTION " adds to (or subtract from) the upper-half of ACC *
* another number located at "Location" of data memory.           *
*                                                                    *
```

```

*          OPTION                      OPERATION                      *
*          -----                      -----                      *
*          0                      (ACCH) = (ACCH) + scalar          *
*          1                      (ACCH) = (ACCH) - scalar          *
*
* The macro is called as ScalAorS Location,OPTION e.g. ScalAorS 311h,0 *
*****
*
ScalAorS $MACRO Location,OPTION
  LARP 1
  LRLK AR1,:Location:      ;point AR1 to Location.
  $IF :OPTION.V: = 0        ;is option = 0 ?
  ADDH *                    ;if so,add the # to ACCH.
  $ELSE                     ;else, subtract the
  SUBH *                    ;number from ACCH.
  $ENDIF
$ENDM                      ;end of macro

*****
**                          MACRO : VectAorS                          **
*****
***** PERFORM ADDITION OR SUBTRACTION BETWEEN TWO COLUMN VECTORS *****
*****
* Depending on the macro parameter OPTION,this macro named " VectAorS *
* M,Loc_of_a,Loc_of_b,OPTION " adds (or subtracts) a column vector b *
* ( M by 1 , stored at Loc_of_b in data memory ) to (or from) another *
* column vector a ( M by 1,stored Loc_of_a in data memory) and stores *
* the resulting column vector c in a's place.                          *
*
*
*          OPTION                      OPERATION                      *
*          -----                      -----                      *
*          0                      c = a + b                          *
*          1                      c = a - b                          *
*
* The macro is called as VectAorS M,Loc_of_a,Loc_of_b,OPTION *
* e.g. VectAorS 3,0311h,0411h,0.                                  *
*****
*
VectAorS $MACRO M,Loc_of_a,Loc_of_b,OPTION
  LRLK AR1,:Loc_of_a:      ;point AR1 to location of a.
  LRLK AR2,:Loc_of_b:      ;point AR2 to location of b.
  LARP 1
  $LOOP :M.V:              ;loop for # of entries.
  LAC *,0,2                ;load ACCUMULATOR with an element of a.
  $IF :OPTION.V:=0         ;are we doing addition ?
  ADD *0+,0,1              ;if so,add to it the corresponding element of b.
  $ELSE                    ;else
  SUB *0+,0,1              ;subtract the corresponding element of b.
  $ENDIF
  SACL *0+                 ;store it in a's place.
  $ENDLOOP
$ENDM                      ;end of macro.

```

```

*****
**                                MACRO : VectMorD                                **
*****
***** MULTIPLY OR DIVIDE A COLUMN VECTOR BY A SCALAR *****
*****
* Depending on the macro parameter OPTION, this macro named *
* " VectMorD M,Loc_of_a,Loc_of_c,OPTION " multiplies (or divides) *
* a column vector a( M by 1,stored at Loc_of_a in data memory ) by *
* a scalar(in Q15 format, stored in upper half of ACCUMULATOR) and *
* stores the resulting column vector c at Loc_of_c in data memory. *
*
*
*          OPTION          OPERATION
*          - - - - -      - - - - -
*          0              c = a x (scalar)
*          1              c = a v (scalar)
*
* The macro is called as VectMorD M,Loc_of_a,Loc_of_c,OPTION *
* e.g. VectMorS 4,311h 411h,0. NOTE THAT the macro uses another *
* macro Q15_Div for division and memory 065h as temporary *
* storage location.
*****
*
VectMorD $MACRO M,Loc_of_a,Loc_of_c,OPTION
    SACH 065h                ;store the scalar at 065h.
    LRLK AR2,:Loc_of_a:      ;point AR2 to location of original vector.
    LRLK AR3,:Loc_of_c:      ;point AR3 to location of scaled vector.
    LARP 2

    $IF :OPTION.V:=0         ;is option = 0 ?if so perform division.
    LT 065h                  ;load TREG with the scalar.

    $LOOP :M.V:              ;loop for # of entries.
    MPY *0+,3                ;multiply a single element by scalar.
    SPH *0+,2                ;store scaled vector element.
    $ENDLOOP

    $ELSE                    ;else perform division.

    $LOOP :M.V:              ;loop for # of entries.
    ZALH *0+,3               ;load ACCH with a single element of original vector.
    Q15_Div                  ;call the macro Q15_Div which performs (ACCH)/(065h).
    SACL *0+,0,2             ;store scaled vector element.
    $ENDLOOP

    $ENDIF
$ENDM

```

```

*****
**                                MACRO : VecVecML                                **
*****
***** CALCULATE INNER-PRODUCT OR OUTER-PRODUCT OF TWO VECTORS *****
*****
* Depending on the macro parameter OPTION, this macro named "VecVecM1 *
* M,Loc_of_A,Loc_of_B,OPTION,Loc_of_C " finds inner-product or outer- *
* product of two vectors a(stored at Loc_of_A in data/program memory) *
* and b (stored at Loc_of_B in data memory).If inner-product option *
* is used then ACCH contains the inner-product. If outer-product *
* option is used, then lower-half of outer-product is calculated and *
* stored at Loc_of_C of data memory.
*
*
*   OPTION          STATUS OF VECTORS          OPERATION
*   -----          -
*   0               a is row vector in data memory      (ACCH) = a*b
*                  b is column vector in data memory
*   1               a is column vector in data memory    C = a*b'(only
*                  b is column vector in data memory      lower-half)
*   2               a is row vector in program memory    (ACCH) = a*b
*                  b is column vector in data memory
*
*
* The macro is called as VecVecM1 M,Loc_of_A,Loc_of_B,OPTION,Loc_of_B *
* e.g. VecVecM1 5,311h,411h,0
*****
*
VecVecM1 $MACRO M,Loc_of_A,Loc_of_B,Loc_of_C,OPTION

$IF :OPTION.V:=2 ;if OPTION = 2, then a is in program memory and
                  ;inner product is to be calculated.
LRLK AR1,:Loc_of_B: ;point AR1 to location of b.
LARP 1
MPYK 0 ;make PREG = 0.
ZAC ;zero ACC.
RPTK :M.V:-1
MAC :Loc_of_A:,*0+ ;perform a*b.
APAC ;ACCH contains inner-product.

$ELSE

$IF :OPTION.V:=0 ;if OPTION = 0, then a is in data memory.
LRLK AR1,:Loc_of_A: ;point AR1 to location of A.
LRLK AR2,:Loc_of_B: ;point AR2 to location of B.
LARP 1
MPYK 0 ;make PREG = 0.
LTP *+,2 ;load TREG with 1st entry of a.
MPY *0+,1 ;multiply it with 1st entry of b.
$LOOP :M.V:-1
LT *+,2 ;perform a*b.
MPYA *0+,1
$ENDLOOP
APAC ;ACCH contains inner-product.

```

```

$ELSE                ;OPTION = 1, so outer-product is to be calculated.
$VAR N                ;define a var named N.
$ASG M.V TO N.V      ;assign value of M to N.
LRLK AR1,:Loc_of_A:+(:M.V:-1)*16    ;point AR1 to entry of a.
LRLK AR2,:Loc_of_B:      ;point AR2 to first entry of b.
LRLK AR3,:Loc_of_C:+(:M.V:-1)*16    ;point AR3 to last row of ab'.

$LOOP :M.V:          ;loop for rows of ab'.
LARP 1
LT *0-,2

$LOOP :N.V:          ;loop for entries in a row.
MPY *0+,3
SPH *+,2              ;store the element of ab'.
$ENDLOOP

SBRK :N.V:*16         ;point to 1st entry of b.
LARP 3
SBRK :N.V:+16         ;point to next upper row of ab'.
$ASG N.V-1 TO N.V     ;modify inner loop counter.
$ENDLOOP              ;done with outer-product

$ENDIF
$ENDIF
$ENDM                  ;end of macro.

```

```

*****
**                                MACRO : Mat_AorS                                **
*****
* PERFORM ADDITION OR SUBTRACTION BETWEEN TWO LOWER-TRIANGULAR MATRICES
*****
* Depending on the macro parameter OPTION, this macro named "Mat_AorS"
* M, Loc_of_A, Loc_of_B, OPTION " adds (or subtracts) a lower-triangular
* matrix B (M by M, stored at Loc_of_B in data memory) to (or from)
* another lower-triangular matrix A (M by M, stored at Loc_of_A in data
* memory) and stores the resulting matrix in A's place.
*
*
*          OPTION          OPERATION
*          - - - - -      - - - - -
*          0              A = A + B
*          1              A = A - B
*          2              A = A + B(diagonal matrix)
*
* The macro is called as Mat_AorS M, Loc_of_A, Loc_of_B, OPTION e.g.
* Mat_AorS 5, 311h, 411h, 1
*****
*
Mat_AorS $MACRO M, Loc_of_A, Loc_of_B, OPTION
$VAR N
$ASG M.V TO N.V

```

```

$IF :OPTION.V:=2          ;if OPTION = 2 ,then B is diagonal.

LRLK AR1,:Loc_of_A:      ;point AR1 to 1st diagonal of A.
LRLK AR2,:Loc_of_B:      ;point AR2 to 1st diagonal of B.
LARP 1
LARK ARO,11H             ;load ARO with 11h for diagonal indexing.
$LOOP :M.V:              ;# of loop.
LAC *,0,2                ;load ACCL with a diagonal entry of A.
ADD *+,0,1               ;add the corresponding diagonal of B.
SACL *0+                 ;store it in proper place.
$ENDLOOP
LARK ARO,10H             ;restore content of ARO.

$ELSE                    ;both A and B are lower-triangular matrices.

LRLK AR1,:Loc_of_A:+(:M.V:-1)*16 ;point AR1 to last row of A.
LRLK AR2,:Loc_of_B:+(:M.V:-1)*16 ;point AR2 to last row of B.

$LOOP :M.V:              ;loop for # of rows.
LARP 1

$LOOP :N.V:              ;loop for # of entries in a row.
LAC *,0,2                ;load ACC with an element of A.
$IF :OPTION.V: = 0       ;are doing addition ?
ADD *+,0,1               ;if so,add the corresponding entry of B .
$ELSE                    ;else
SUB *+,0,1               ;subtract corresponding entry of B from it.
$ENDIF
SACL *+,0,1              ;store it in A's place.
$ENDLOOP                 ;done with a row.

SBRK :N.V:+16            ;point to next upper row of B.
LARP 2
SBRK :N.V:+16            ;point to next upper row of A.
$ASG N.V-1 TO N.V        ;modify counter for # of entries in a row.
$ENDLOOP

$ENDIF
$ENDM                    ;end of macro.

```

```

*****
**                                MACRO : MtMtMlpd                                **
*****
**** MULTIPLY A MATRIX (STORED IN PROGRAM MEMORY) BY ANOTHER MATRIX ****
**** (STORED IN DATA MEMORY) AND STORE THE RESULT (IN DATA MEMORY) ****
*****
* This macro named " MtMtMlpd M,N,P,Loc_of_A,Loc_of_B,Loc_of_C " *
* multiplies a matrix A (M by N,stored at Loc_of_A in program memory) *
* by another matrix B(N by P,stored at Loc_of_B in data memory) and *
* depending on the macro parameters OPTION_1 and OPTION_2 stores the *.

```

```

* resulting matrix C at Loc_of_C in data memory.
*
*   OPTION_1  OPTION_2      OPERATION
*   -----  -
*   0         0      C = A*B, all entries of C are stored
*   1         0      C = (A*B)', all entries of C are stored
*   x         1      C = A*B = (A*B)', lower-half of C is stored
*
* The macro is called as MtMtMlpd M,N,P,Loc_of_A,Loc_of_B,Loc_of_C,
* e.g. MtMtMlpd 4,5,3,311h,411h,511h,1,0.
*****
MtMtMlpd $MACRO M,N,P,Loc_of_A,Loc_of_B,Loc_of_C,OPTION_1,OPTION_2

$VAR NUM                      ;define a var named NUM.
$ASG 0 TO NUM.V               ;assign 0 to NUM.

$IF :OPTION_2.V:=0            ;is OPTION_2 = 0 ? if so, calculate all entries.
                              ;of AB or (AB)'.
    LRLK AR3,:Loc_of_C:      ;point AR3 to location of C.
    $LOOP :M.V:              ;loop for # of column of (AB) or (AB)'.
    LARP 1
    LRLK AR1,:Loc_of_B:      ;point AR1 to location of B.

    $LOOP :P.V:              ;loop for # of rows of (AB) or (AB)'.
    MPYK 0
    ZAC
    RPTK :N.V:-1
    MAC :Loc_of_A+:NUM.V:*16,*0+ ;multiply a row of A by a column of B.
    APAC
    LARP 3
    $IF :OPTION_1.V:=0        ;is OPTION_1 = 0 ? if so AB is calculated.
    SACH *+,0,1              ;store the corresponding entry of AB.
    $ELSE                    ;else, (AB)' is being calculated.
    SACH *0+,0,1             ;store the corresponding element of(AB)'.
    $ENDIF
    SBRK :N.V:*16-1          ;point to next column of B.
    $ENDLOOP

    LARP 3
    $IF :OPTION_1.V:=0        ;if OPTION_1 = 0, point to next row of
    ADRK 10H-:P.V:           ;AB.
    $ELSE                    ;else,
    SBRK :P.V:*16-1          ;point to next column of (AB)'.
    $ENDIF
    $ASG NUM.V+1 TO NUM.V    ;increment variable NUM.
    $ENDLOOP

$ELSE                        ;OPTION_2 = 1, hence half of AB = (AB)' is
                              ;to be calculated.
    LRLK AR3,:Loc_of_C:+(M.V:-1)*16 ;point AR2 to last row of AB.
    $LOOP :M.V:              ;loop for # of rows of AB.

```

```
LRLK AR1,:Loc_of_B: ;point AR1 to 1st column of B.
```

```
LARP 1
```

```
$LOOP :P.V:-:NUM.V: ;loop for # of entries in a row of AB.
```

```
MPYK 0
```

```
ZAC
```

```
RPTK :N.V:-1
```

```
MAC :Loc_of_A:+(:M.V:-:NUM.V:-1)*16,*0+ ;multiply a row of A by  
APAC ;a column of B.
```

```
LARP 3
```

```
SACH *+,0,1 ;store the resulting entry of AB.
```

```
SBRK :N.V:*16-1 ;point to next column of B.
```

```
$ENDLOOP ;done with a row of AB.
```

```
LARP 3
```

```
SBRK (:P.V:-NUM.V)+16 ;point to next upper row of AB.
```

```
$ASG NUM.V+1 TO NUM.V ;increment var NUM.
```

```
$ENDLOOP
```

```
$ENDIF
```

```
$ENDM ;end of macro.
```

```
*****  
**                               MACRO : MtMtMldd                               **  
*****  
***** MULTIPLY A MATRIX (STORED IN DATA MEMORY) BY ANOTHER MATRIX *****  
***** (STORED IN DATA MEMORY) AND STORE THE RESULT (IN DATA MEMORY) ***  
*****  
* This macro named " MtMtMldd  M,N,P,Loc_of_A,Loc_of_B,Loc_of_C " *  
* multiplies a matrix A( M by N, stored Loc_of_A in data memory ) *  
* or its transpose by another matrix B(P by Q,stored at Loc_of_B in *  
* data memory) and depending on the macro parameters OPTION_1 and *  
* OPTION_2 stores the resulting matrix C at Loc_of_C in data memory. *  
* *  
*   OPTION_1  OPTION_2      OPERATION *  
*   - - - - -  - - - - -  - - - - - *  
*       0       0      C = A*B, all entries of C are stored *  
*       1       0      C = (A*B)', all entries of C are stored *  
*       0       1      C = A*B = (A*B)', lower-half of C is stored *  
*       1       1      C = A'*B ,all entries of C are stored *  
* *  
* The macro is called as MtMtMldd  M, N, P, Q,Loc_of_A,Loc_of_B, *  
* Loc_of_C, e.g. MtMtMldd 4,5,5,3,311h,411h,511h,1,0. *  
*****  
*  
MtMtMldd $MACRO M,N,P,Q,Loc_of_A,Loc_of_B,Loc_of_C,OPTION_1,OPTION_2  
*  
$IF :OPTION_2.V:= 0 ;if OPTION_2 = 0, then calculate all entries  
;of AB or (AB)'.  
LRLK AR1,:Loc_of_A: ;point AR1 to location of A.
```



```

LRLK AR3,:Loc_of_C: ;point AR3to location of C

$LOOP :M.V: ;loop counter for # of rows/columns of AB/(AB)'.
LRLK AR2,:Loc_of_B: ;point AR2 to location of B.
LARP 1

$LOOP :Q.V: ;loop counter for # of columns/rows of AB/(AB)'.
MPYK 0
LTP *+,2 ;load TREG with the 1st entry of a row of A.
MPY *0+,1 ;multiply it with 1st entry of a column of B.

$LOOP :N.V:-1 ;multiply a row of A by a column
LTA *+,2 ;of B.
MPY *0+,1
$ENDLOOP ;done with an entry of AB/(AB)'.

APAC
SBRK :N.V: ;point to 1st entry of the same row of A.
LARP 2
SBRK :N.V:*10H-1 ;point to next column of B.
LARP 3
$IF :OPTION_1.V:=0 ;if OPTION_1 = 0, then store it
SACH *+,0,1 ;in proper place of AB.
$ELSE ;else, store it in proper
SACH *0+,0,1 ;place of (AB)'.
$ENDIF
$ENDLOOP ;done with a row/column of AB/(AB)'.

MAR *0+,3
$IF :OPTION_1.V:=0 ;if OPTION_1 = 0, then point to
ADRK 10H-:Q.V: ;next row of AB.
$ELSE ;else point to next column of
SBRK :Q.V:*10H-1 ;(AB)'.
$ENDIF
$ENDLOOP ;done with all entries of AB or (AB)'.

$ELSE ;if OPTION_2 = 1, then calculate either lower
;half of AB = (AB)' or A'B.
$IF :OPTION_1.V:=0 ;if OPTION_1 = 0, then calculate lower
;half of (AB)'.
$VAR NUM ;define a variable name NUM.
$ASG 0 TO NUM.V ;assign 0 to NUM.
LRLK AR1,:Loc_of_A:+(M.V-1)*10H ;point AR1 to last row of A.
LRLK AR3,:Loc_of_C:+(M.V-1)*10H ;point AR2 to last row of AB.

$LOOP :M.V: ;loop counter for # of rows of AB.
LRLK AR2,:Loc_of_B: ;point AR2 to location of B.
LARP 1

$LOOP :Q.V:-:NUM.V: ;loop counter for # of entries in a row of AB.
MPYK 0
LTP *+,2 ;load TREG with 1st entry of a row of A.
MPY *0+,1 ;multiply it with 1st entry of a column of B.

```

```

$LOOP :N.V:-1
LTA *+,2
MPY *0+,1
$ENDLOOP                                ;done with an entry of a row of AB.

APAC
SBRK :N.V:                             ;point to 1st entry of a row of A.
LARP 2
SBRK :N.V:*10H-1                       ;point to next column of B.
LARP 3
SACH *+,0,1                            ;store ACCH in proper place of AB.
$ENDLOOP                                ;done with a row of AB.

MAR *0-,3                               ;point to next upper row of A.
SBRK (:Q.V:-:NUM.V:)+10H               ;point to next upper row of AB.
$ASG NUM.V+1 TO NUM.V                  ;increment variable NUM.
$ENDLOOP                                ;done with lower half of AB.

$ELSE                                   ;else, OPTION_1=1 , so A'B
                                       ;will be calculated.
LRLK AR1,:Loc_of_A:                   ;point AR1 to location of A.
LRLK AR2,:Loc_of_B:                   ;point AR2 to location of B.
LRLK AR3,:Loc_of_C:                   ;point AR3 to location of C.
$LOOP :N.V:                            ;loop counter for # of rows of A'B.

    $LOOP :Q.V:                        ;loop counter for # of entries in a row of A'B.
    LARP 1
    MPYK 0                             ;zero PREG.
    LTP *0+,2
    MPY *0+,1
    $LOOP :M.V:-1                      ;multiply a column of A by
    LTA *0+,2                          ;a column of A.
    MPY *0+,1
    $ENDLOOP
    APAC                               ;one entry of a row of A'B
    LARP 3                             ;is calculated.
    SACH *+,0,2                        ;store it in proper place.
    SBRK :P.V:*10H-1                  ;point to next column of B.
    LARP 1
    SBRK 10H*:P.V:                    ;point to the 1st entry of the same column of A
    $ENDLOOP                          ;done with all the entries of a row of A'B.

    ADRK 1                             ;point to next column of A.
    LRLK AR2,:Loc_of_B:                ;point to the 1st column of B.
    LARP 3                             ;point to the next row of A'B.
    ADRK 10h-:Q.V:
$ENDLOOP

$ENDIF                                  ;end of IF statement for OPTION_1.
$ENDIF                                  ;end of IF statement fro OPTION_2.
$ENDM

```

```

*****
**                               MACRO : Fill_Mat                               **
*****
**** FILL UP THE UPPER HALF OF A SYMMETRIC MATRIX FROM LOWER HALF ****
*****
* This macro named " Fill_Mat M,Loc_of_A " fills up the upper half of *
* a lower-triangular matrix A (M BY M stored at ALOC in data memory) *
* from the knowledge of its lower half. The macro is called as *
* Fill_Mat M, M,Loc_of_A e.g. Fill_Mat 5,0311h,                               *
*****
*
Fill_Mat $MACRO M,Loc_of_A
$VAR N                               ;define a variable named N.
$ASG M.V TO N.V                     ;assign value of M to N.
LRLK AR1,:Loc_of_A:+1               ;point AR1 to 2nd column of A.
LARP 1
$LOOP :M.V:-1
RPTK :M.V:-:N.V:
BLKD :Loc_of_A:+16+(:M.V:-:N.V:)*16 ,*0+ ;move half-row to half-column.
SBRK (:M.V:-:N.V:+1)*16 -1           ;modify AR1 for next transfer.
$ASG N.V-1 TO N.V                   ;decrement var NUM.
$ENDLOOP
$ENDM

*****
**                               MACRO : Mat_Copy                               **
*****
***** MAKE A COPY OF A MATRIX TO A SPECIFIED LOCATION *****
*****
* This macro named " Mat_Copy M,N,SOURCE,DEST " makes a copy of matrix *
* A ( M by N, stored at SOURCE in data memory ) to location DEST in *
* data memory . The macro is called as Mat_Copy M,N,SOURCE,DEST e.g. *
* Mat_Copy 5,5,311h,411h.                               *
*****
*
Mat_Copy $MACRO M,N,SOURCE,DEST
LRLK AR1,:DEST:                     ;point AR1 to DEST.
LRLK AR2,:SOURCE:                   ;point AR2 to SOURCE
$LOOP :N.V:
LARP 2
  $LOOP :M.V:                       ;loop for # of entries.
  LAC *0+,0,1                       ;load ACC with entry of SOURCE.
  SACL *0+,0,2                      ;store it in DEST.
  $ENDLOOP
SBRK :M.V:*10H-1                    ;modify AR2 for next column.
LARP 1
ADRK :M.V:*10H-1                    ;modify AR1 for next column.
$ENDLOOP
$ENDM                               ;end of macro.

```

```

*****
**                               MACRO : Move_P_D                               **
*****
***** MOVE A MATRIX FROM PROGRAM MEMORY TO DATA MEMORY *****
*****
* This macro named " Move_P_D M,N,SOURCE_P,DEST_D " makes a copy of *
* matrix A (M by N), which is stored at SOURCE_P in program memory, *
* to DEST_D in data memory. The macro is called as Move_P_D M,N, *
* SOURCE_P,DEST_D e.g. Move_P_D 2,5,411h,411h *
*****

```

```

Move_P_D $MACRO M,N,SOURCE_P,DEST_D
  LARP 1
  LRLK AR1,:DEST_D: ;point AR1 to DEST_D in data memory.
  LALK :SOURCE_P: ;ACC contains address of SOURCE_P.
  $LOOP :M.V: ;copy row by row.
  RPTK :N:-1
  TBLR *+ ;copy one row from SOURCE_P to DEST_D.
  ADDK 16 ;point to next row at SOURCE_P.
  ADRK 16-:N: ;modify AR1 s.t. it points to next row at DEST_D.
  $ENDLOOP
$ENDM

```

```

*****
**                               MACRO : Q15_Div                               **
*****
***** DIVIDE A Q15 NUMBER BY ANOTHER Q15 NUMBER *****
*****
* This macro named Q15_Div divides a Q15 number stored in the *
* upper-half of ACCUMULATOR by another Q15 number stored in *
* location 065h of data memory. The resulting Q15 quotient is *
* stored back in the lower-half of ACCUMULATOR The macro is called *
* with no parameter e.g. Q15_Div .But make sure that before the *
* macro is called, ACCH contains numerator and data memory 065h *
* contains denominator. The macro uses 065h and 066h of data memory.*
*****

```

```

Q15_Div $MACRO
DENOM: .equ 065h ;contains denominator beforehand.
NUMER: .equ 066h ;location of numerator.
  SACH NUMER ;store numerator from ACCH to NUMER.
  LT NUMER ;load TREG with numerator.
  MPY DENOM ;multiply numerator by denominator.
  PAC
  BGEZ DIV1? ;check for sign of quotient.if +ve go to DIV1?
  LAC DENOM ;else continue.
  ABS ;get absolute value of denominator.
  SACL DENOM ;store it in DENOM.
  ZALH NUMER ;load ACCH with numerator.
  ABS ;get absolute value of numerator.
  RPTK 14
  SUBC DENOM ;perform division.

```

```

NEG                ;negate quotient.
B DIV2?            ;go to DIV2?
DIV1? LAC DENOM    ;quotient is +ve.
ABS                ;take absolute value of denominator.
SACL DENOM         ;store it in DENOM.
ZALH NUMER         ;load ACCH with numerator.
ABS                ;take absolute value of numerator.
RPTK 14
SUBC DENOM         ;perform division.
DIV2? NOP          ;we're done.
$ENDM

```

```

*****
**                                MACRO : LU_Fact                                **
*****
***** PERFORM LU FACTORIZATION OF A SYMMETRIC MATRIX *****
*****
* From the knowledge of lower-half of a symmetric matrix A (M by M, *
* stored at Loc_of_A in data memory), this macro named " LU_Fact M, *
* Loc_of_A ,Loc_of_C " performs its LU decomposition i.e. it *
* factorizes matrix A as A =LU ,where L is a lower-triangular matrix *
* and U is a upper-triangular matrix with diagonal entries as 1's. *
* The diagonals of U matrix are not stored since they are known to *
* be 1's. The macro stores all the entries of L and U at Loc_of_C *
* in data memory in a compact matrix form. The algorithm for LU *
* decomposition is :
*
*
*      1) L(i,1) = A(i,1),
*              for 1 ≤ i ≤ M
*
*
*      2) U(1,j) = A(1,j)/A(1,1) ,
*              for 2 ≤ j ≤ M
*
*
*              j-1
*      3) L(i,j) = A(i,j) - Σ L(i,k)U(k,j),
*              k=1
*              for 2 ≤ i ≤ M , 2 ≤ j ≤ i
*
*
*              i-1
*      U(i,j) = [ A(j,i) - Σ L(i,k)U(k,j) ]/L(i,i),
*              m=1
*              for 2 ≤ i ≤ j, ≤ j ≤ M
*
*
* The macro uses two other macros named Q15_Div and Mat_Copy. The *
* macro Q15_Div uses 065h and 066h of data memory. The macro is *
* called as LU_Fact M,Loc_of_A,Loc_of_C e.g. LU_Fact 5,311h,411h. *
*****
*
LU_Fact $MACRO M,Loc_of_A,Loc_of_C
$VAR NUM
$ASG 0 TO NUM.V

```

```

* find the 1st row of U s.t.  $U(1,j) = A(1,j)/A(1,1)$ .
  LRLK AR1,:Loc_of_A:           ;point AR1 to location of A.
  LRLK AR3,:Loc_of_C:+1
  LARP 1
  LAC *0+                       ;store A(1,1) at 065h for division.
  SACL 65H
  $LOOP :M.V:-1
  ZALH *0+,3
  Q15_Div                       ;( $ACCL$ ) =  $A(1,j)/A(1,1)$ .
  SACL *+,0,1                   ;store it in proper place.
$ENDLOOP
*
* move first column of A so that  $L(i,1) = A(i,1)$ .
  Mat_Copy :M.V:,1,:Loc_of_A:,:Loc_of_C:
*
* starting from 2nd row, find L-entries first followed by U-entries.
$VAR U_SUM,L_SUM                ;U_SUM and L_SUM are d counters.
$ASG 1 TO U_SUM                  ;initialize U_SUM=1 for 2nd row.
$LOOP :M.V:-1
  LRLK AR1,:Loc_of_A:+17+:NUM.V:*16 ;point AR1 to  $A(i,2)$ ,  $2 \leq i \leq M$ .
  LRLK AR2,:Loc_of_C:+16+:NUM.V:*16 ;point AR2 to  $L(i,1)$ ,  $2 \leq i \leq M$ .
  LRLK AR3,:Loc_of_C:+1             ;point AR3 to  $U(1,2)$ .
  $ASG 1 TO L_SUM                  ;L_SUM = 1 for 2nd column.
  LARP 3
*
  $LOOP :NUM.V:+1                 ;loop for entries of L-row.
  LRLK AR2,:Loc_of_C:+16+:NUM.V:*16
  ZAC
  MPYK 0
  $LOOP :L_SUM.V:                 ;counter for k.
  LTS *0+,2
  MPY *+,3                       ;( $PREG$ ) =  $L(i,k)*U(k,j)$ .
  $ENDLOOP                       ;
  SPAC                           ;( $ACCH$ ) =  $-\sum_{k=1}^{j-1} L(i,k)U(k,j)$ .
  LARP 1                          ;
  ADDH *+,2                       ;( $ACCH$ ) = ( $ACCH$ ) +  $A(i,j)$ .
  SACH *,0,3                      ; $L(i,j) = (ACCH)$ .
  SBRK :L_SUM.V:*16-1             ;modify AR3 for next L-entry.
  $ASG L_SUM.V+1 TO L_SUM.V       ;increase L_SUM for next entry.
$ENDLOOP
* done with L-entries in a row.
*
* calculate U-entries in a row.
$IF NUM.V < (:M.V:-2)            ;if all rows of LU-compact-form not
  LARP 2                          ;calculated, then continue for U-entries.
  LAC *,0,1                       ;store  $L(i,i)$  at 065h
  SACL 65H                        ;for division.
  MAR *0+
  MAR *-,2
  $LOOP :M.V:-2-:NUM.V:           ;loop for entries of U-row.
  LRLK AR2,:Loc_of_C:+16+:NUM.V:*16
  ZAC

```

```

MPYK 0
$LOOP :U_SUM.V:           ;counter for m.
LTS *+,3
MPY *0+,2                 ;(PREG) = L(i,m)*U(m,j).
$ENDLOOP                  ;
SPAC                       ;(ACCH) = -  $\sum_{m=1}^{i-1} L(i,m)U(m,j)$ .
LARP 1                    ;
ADDH *0+,3                ;(ACCH) = (ACCH) + A(j,i).
Q15_Div                   ;(ACCL) = (ACCH) ÷ L(i,i).
SACL *                    ;U(i,j) = (ACCL).
SBRK (:NUM.V:+1)*16-1     ;modify AR3 for next U-entry.
LARP 2
$ENDLOOP
*done with entries of a row of U.
*
$ELSE                      ;if all the rows of U
NOP                        ;not calculated yet
$ENDIF                    ;then go for next row.
$ASG NUM.V+1 TO NUM.V
$ASG U_SUM.V+1 TO U_SUM.V ;increase U_SUM counter of U.
$ENDLOOP
$ENDM                      ;end of macro.

```

```

*****
**                               MACRO : For_Ward                               **
***** SOLVE LY=B' (L IS LOWER-TRIANGULAR) BY FORWARD SUBSTITUTION *****
*****
* This macro named " For_Ward M,N,Loc_of_L,Loc_of_B,Loc_of_Y " solves *
* Y from LY=B' using forward substitution , where L is a lower *
* triangular matrix.The matrices L (M by M) and B (N by M) are stored *
* at Loc_of_L and Loc_of_B in data memory respectively. Note that it *
* is matrix B , not B' that is stored at Loc_of_B. The entries of Y *
* matrix are stored starting at YLOC in data memory. *
*
* The compact algorithm for this method is, *
*
*                               j-1 *
*       Y(i,j) = [ B(j,i) -  $\sum_{k=1}^{j-1} L(i,k)Y(k,j)$  ]/L(i,i) *
*                               k=1 *
*       for 1 ≤ i ≤ M and 1 ≤ j ≤ M *
*
* The macro is called as For_Ward M,N,Loc_of_L,Loc_of_B,Loc_of_Y e.g. *
* For_Ward 3,4,311h,411h,511h. NOTE THAT this macro uses another *
* macro named Q15_Div which uses 065h and 066h of data memory. *
*****
*
For_Ward $MACRO M,N,Loc_of_L,Loc_of_B,Loc_of_Y
$VAR NUM
$ASG 0 to NUM.V
LARP 1
LRLK AR1,:Loc_of_L:      ;point AR1 to L(1,1).

```

```

LAC *
SACL 65H                      ;(065H) = L(1,1).
$LOOP :M.V:                   ;loop for i (i.e. rows of Y).
  LRLK AR2,:Loc_of_B+:+NUM.V:
  LRLK AR3,:Loc_of_Y:
  $LOOP :N.V:                 ;loop for j (i.e. columns of Y).
    LRLK AR1,:Loc_of_L+:+NUM.V:*16
    ZAC
    MPYK 0
    $LOOP :NUM.V:             ;loop for k.
      LTS *+,3
      MPY *0+,1
    $ENDLOOP
    SPAC                      ;          l-1
                                ;(ACCH) = -  $\sum_{k=1}^{l-1} L(i,k)*Y(k,j)$ .
    LARP 2                    ;          k=1
    ADDH *0+,3                ;(ACCH) = B(j,i) + (ACCH).
    Q15_Div                   ;(ACCL) = (ACCH) ÷ L(1,1).
    SACL *                    ;Y(i,j)=(ACCL).
    $IF NUM.V = 0
      ADRK 1
    $ELSE
      SBRK :NUM.V:*16-1
    $ENDIF
    LARP 1
  $ENDLOOP                    ;end of loop of j.
$IF NUM.V < (:M.V:-1)
  MAR *0+
  MAR *+
  LAC *
  SACL 65H                    ;store L(i+1,i+1) for next row.
$ELSE
  NOP
$ENDIF
$ASG NUM.V+1 TO NUM.V
$ENDLOOP                      ;end of loop for i.
$ENDM

```

```

*****
**                               MACRO : Bck_Ward                               **
***** SOLVE UX=Y (U IS UPPER-TRIANGULAR) USING BACK-SUBSTITUTION *****
*****
* Using back-substitution,this macro named " Bck_Ward M,N,Loc_of_U, *
* Loc_of_Y " solves X from UX = Y, where U is an upper-triangular *
* matrix with diagonals as 1's. The matrices U (M by M) and Y(M by N) *
* are stored at Loc_of_U and Loc_of_Y of data memory respectively.The *
* entries of matrix X are stored in Y's place. Since the last row of *
* X is same as Y, this macro calculates entries of X starting from *
* 2nd-last-row up to the 1st row. The algorithm for this is shown *

```



```

* below :
*
*          1)  $X(M,i) = Y(M,i)$ , for  $1 \leq i \leq M$ 
*
*          2)  $X(i,j) = Y(i,j) - \sum_{k=i+1}^M U(i,k)X(k,j)$ ,
*             for  $2 \leq i \leq M$  and  $1 \leq j \leq M$ 
*
* The macro is called as Bck_Ward M,N,Loc_of_U,Loc_of_Y e.g.
* Bck_Ward 3,4,311,411h.
*****
Bck_Ward $MACRO M,N,Loc_of_U,Loc_of_Y
$VAR NUM
$ASG 0 to NUM.V
BKINCO: .equ :M.V:-1
BKINC1: .equ BKINCO*16
BKINC2: .equ BKINCO*17-16
*
*calculate rows of X starting from 2nd-last-row.
*
  LARP 1
  $LOOP :M.V:-1 ;loop for rows of X.
  LRLK AR1,:Loc_of_Y.V:+BKINC1 ;AR1 points to location of Y.
  LRLK AR2,:Loc_of_U.V:+BKINC2-:NUM.V:*16 ;AR2 points to location of U.
*
  $LOOP :N.V: ;loop for entries in a row of X.
  ZAC
  MPYK 0
  $LOOP :NUM.V:+1 ;counter for k.
  LTS *0-,2
  MPY *-,1 ;(PREG) = - U(i,k)*X(k,j).
  $ENDLOOP
  SPAC ;(ACCH) = -  $\sum_{k=i+1}^M U(i,k)X(k,j)$ .
  ADDH *
  SACH * ;X(i,j) = Y(i,j) + (ACCH).
  LRLK AR2,:Loc_of_U.V:+BKINC2-:NUM.V:*16 ;modify AR2 for next entry.
  ADRK (:NUM.V:+1)*16+1 ;modify AR1 for next entry.
  $ENDLOOP
*done with one row of X.
*
  $ASG NUM.V+1 TO NUM.V ;go for next upper row.
$ENDLOOP
$ENDM

*****
**          MACRO : Sqr_Root          **
*****
***** FIND SQUARE-ROOT OF A Q15 NUMBER STORED IN ACCH *****
*****
* This macro named "Sqr_Root" finds square-root of a Q15 number *

```

```
* stored in upper half of ACCUMULATOR using Newton_Raphson method.*
* The result is stored back in lower half of ACCUMULATOR.The *
* macro uses data memory locations 65h to 68h. It is called as *
* Sqr_Root with no parameter. However, make sure that the Q15 *
* number ,whose square-root is to be calculated, resides in upper *
* half of ACCUMULATOR before the macro is called. *
```

```
*****
*
```

```
Sqr_Root $MACRO
```

```
SQ1 .equ 65H
```

```
SQ2 .equ 66H
```

```
SQ3 .equ 67H
```

```
SQ4 .equ 68H
```

```
SFR
```

```
SACH SQ1 ; (SQ1) = a/2.
```

```
LAC SQ1
```

```
ADLK 4000H
```

```
SACL SQ4
```

```
SQR? LAC SQ4 ; (SQ4) = xold.
```

```
SACL SQ2 ; (SQ2) = xold.
```

```
SFR
```

```
SACL SQ3 ; (SQ3) = xold/2.
```

```
ZALH SQ1
```

```
RPTK 14
```

```
SUBC SQ2 ; (ACCL) = a/(2*xold).
```

```
ADD SQ3
```

```
SACL SQ4 ; (SQ4)=xnew = xold/2 + a/(2*xold).
```

```
LAC SQ2
```

```
SUBS SQ4
```

```
SBLK 2 ; is (xold-xnew) > 0002h ?
```

```
BGZ SQR? ; if so,continue iteration.
```

```
LAC SQ4 ; (ACCL)=1/a.
```

```
$ENDM
```

```
*****
** MACRO : Choleski **
```

```
*****
```

```
***** PERFORM CHOLESKY DECOMPOSITION OF A SYMMETRIC MATRIX *****
```

```
*****
```

```
* Using Choleski's method, this macro named " Choleski M,Loc_of_A " *
```

```
* decomposes a symmetric matrix A ( N by N,just lower-half is stored *
```

```
* at Loc_of_A in data memory ) into a lower triangular matrix M and *
```

```
* an upper triangular matrix M' such that A=M*M'. The entries are *
```

```
* calculated as : *
```

```
* *
```

```
* 
$$M(k,k) = \sqrt{A(k,k) - \sum_{j=1}^{k-1} [M(k,j)]^2}$$
 *
```

```
* 
$$M(i,k) = [ A(i,k) - \sum_{j=1}^{k-1} M(i,j)M(k,j) ] / M(k,k)$$
 *
```

```
* *
```

```
* *
```

```
* *
```

```
* *
```

```

*                                     j=1                                     *
*                                     for 1 ≤ k ≤ N and (k+1) ≤ i ≤ N         *
*                                     *                                       *
* The lower triangular matrix M is stored in A's place. The macro uses *
* two other macros named Q15_Div and Sqr_Root which use data memory *
* locations 65h to 68h. The macro is called as Choleski M, Loc_of_A e.g. *
* Choleski 5, 311h. *
*****
*
Choleski $MACRO M, Loc_of_A
$VAR NUM
$ASG 0 TO NUM.V
* find 1st column of M.
LRLK AR2, :Loc_of_A: ;point AR2 to MLOC.
LARP 2
ZALH *
Sqr_Root
SACL *0+ ;M(1,1)=√A(1,1).
SACL 65h ;(65h)=M(1,1).
$LOOP :M.V:-1 ;loop for off-diagonals of 1st column of M.
ZALH *
Q15_Div
SACL *0+ ;M(i,1)=A(i,1)/M(1,1).
$ENDLOOP
*
* find all other columns of M except last column.
$LOOP :M.V:-2 ;loop for (N-2) columns.
* first find the diagonal entry of the particular column.
LRLK AR2, :Loc_of_A:+10H+:NUM.V:*16
LRLK AR3, :Loc_of_A:+20H+:NUM.V:*16
ZAC
MPYK 0
$LOOP :NUM.V:+1 ;loop for j.
SQRS *+
$ENDLOOP ; k-1
SPAC ;(ACCH) = - Σ [M(k,j)²]
ADDH * ; j=1
Sqr_Root
SACL * ;M(k,k) = √{A(k,k) + (ACCH)}
SACL 65H ;(65h) = M(k,k).
* done with diagonal entry of a column.
* find off-diagonal entries of the column.
$LOOP :M.V:-2-:NUM.V:
LRLK AR2, :Loc_of_A:+10H+:NUM.V:*16
ZAC
MPYK 0
$LOOP :NUM.V:+1
LT *+, 3
MPYS *+, 2
$ENDLOOP ; k-1
SPAC ;(ACCH) = - Σ M(i,p)*M(k,j)
LARP 3 ; j=1

```

```

      ADDH *                ;(ACCH) = A(i,k) + (ACCH).
      Q15_Div
      SACL *                ;M(i,k) = (ACCH)/M(k,k).
      ADRK 15-:NUM.V:      ;modify AR3 for next entry of the column.
      LARP 2
      $ENDLOOP
      $ASG NUM.V+1 TO NUM.V ;go for next column.
    $ENDLOOP
* done with first (N-1) columns.
*
* find the single entry of last column of M.
LRLK AR2,:Loc_of_A:+10H+:NUM.V:*16
ZAC
MPYK 0
  $LOOP :NUM.V:+1
  SQRS *+
  $ENDLOOP
SPAC                ;          N-1
                  ;(ACCH) =  $\sum_{j=1}^{N-1} [M(N,j)]^2$ 
ADDH *              ;
Sqr_Root
SACL *              ;M(N,N) =  $\sqrt{A(N,N)-(ACCH)}$ .
$ENDM

```

```

*****
**                               MACRO : Seq_Proc                               **
*****
**** FOR VECTOR OBSERVATION ,FIND FILTERED ESTIMATE OF STATE VECTOR ****
***** & ERROR COVARIANCE MATRIX BY SEQUENTIAL PROCESSING *****
*****
* This macro named " Seq_Proc M, N, Loc_of_H, Loc_of_P, Loc_of_R, *
* Loc_of_y, Loc_of_x, Loc_of_k, temp_1, temp_2 " calculates filtered *
* estimate of state vector x(n|n) and error covariance matrix P(n|n) *
* in STANDARD KALMAN FILTER ALGORITHM using sequential processing *
* of data vector y(n) when measurement noise v(n) is uncorrelated i.e. *
*  $E\{v(n)v(k)\} = R\delta(n,k)$ . The MxN observation matrix H(n), the NxN *
* predicted error covariance matrix P(n+1|n), the Nx1 predicted *
* state estimate x(n+1|n),the Mx1 measurement vector y(n) and the MxM *
* diagonal matrix R(n) are stored at HLoc in program memory,PLoc *
* in data memory, XLoc in data memory, YLoc in data memory and RLoc *
* in data memory respectively. The intermediate steps use two NxN *
* data memory pads namely whose starting addresses are given by temp_1 *
* and temp_2.The steps for the sequential lgorithm are given below :
*
*
*      P(0)=P(n+1|n) ; x(0)=x(n+1|n)
*      For i=1 to M do
*          1) k(i) =  $P(i-1)*h'(i) / [h(i)*P(i-1)*h'(i)+r(i,i)]$ 
*              where h(i) is ith row of H(n).
*          2) P(i) =  $P(i-1)-k(i)*h(i)*P(i-1)$ 
*          3) x(i) =  $x(i-1)-[y(i+1)-h(i)*x(i-1)]*k(i)$ 
*      P(n+1|n+1) = P(M)
*      x(n+1|n+1) = x(M)
*

```

```

* The macro is called as Seq_Proc M,N,HLoc,PLoc,RLoc,YLoc,XLoc,KLoc, *
* temp_1,temp_2 e.g. Seq_Proc 4,5,0FF00h,300h,388h,384hh,380h,383h, *
* 382h,308h. *
*****
*
Seq_Proc $MACRO M,N,HLoc,PLoc,RLoc,YLoc,XLoc,KLoc,temp_1,temp_2
$VAR R_C ;define a variable named R_C.
$ASG 0 TO R_C.V ;assign 0 to R_C.

$LOOP :M.V: ;execute the loop M times

* find k(i)
MtMtMlpd 1,:N,:N,:HLoc+:R_C.V:*10h,:PLoc:,:temp_1:,1,0 ;find P*h'.
VecVecM1 :N,:HLoc+:R_C.V:*10H,:temp_1:,0,2 ;(ACCH)=h(i)*P*h'(i).
ScalAorS :RLoc+:R_C.V:,0 ;(ACCH)=(ACCH)+R(i,i).
VectMorD :N,:temp_1:,:KLoc:,1 ;k(i)=P*h'(i)/(ACCH).

* find P(i)
VecVecM1 :N,:temp_1:,:KLoc:,:temp_2:,1 ;(temp_2)=P*h'(i)*k'(i).
Mat_AorS :N,:PLoc:,:temp_2:,1 ;P=P-(temp_2).
Fill_Mat :N,:PLoc: ;P=P-P*h'(i)*k'(i).

• * find x(i)
VecVecM1 :N,:HLoc+:R_C.V:*10h,:XLoc:,0,2 ;(ACCH)=h(i)*x.
ScalAorS :YLoc+:R_C.V:*10H,1 ;(ACCH)=(ACCH)-y(i+1).
VectMorD :N,:KLoc:,:temp_1:,0 ;(temp_1)=k(i)*(ACCH).
VectAorS :N,:XLoc:,:temp_1:,1 ;x=x-(temp_1).
$ASG R_C.V+1 TO R_C.V

$ENDLOOP
$ENDM

```

F.2 Special Macros

```

*****
**                                MACRO : PRED_EST                                **
*****
**** FIND THE ESTIMATES OF PREDICTED COVARIANCE AND STATE VECTOR ****
***** WHEN STATE-TRANSITION MATRIX  $\Phi$  HAS A SPECIAL STRUCTURE *****
*****
* This macro named " PRED_EST N,Loc_of_Phi, Loc_of_P,Loc_of_x, *
* Loc_of_Q " calculates the lower-half of predicted estimates of *
* error covariance matrix  $P(k+1|k)$  and state vector  $x(k+1|k)$  using *
* the equations :
*
*
*           $P(k+1|k) = \Phi P(k|k) \Phi' + Q$ 
*          and       $x(k+1|k) = \Phi x(k|k)$ 
*
* and stores them in  $P(k|k)$ 's and  $x(k|k)$ 's places. The matrix  $\Phi$  has *
* a special structure as shown below :
*
*           $\Phi = \begin{bmatrix} \theta_1 & 0 & 0 & 0 \\ 0 & \theta_2 & 0 & 0 \\ 0 & 0 & \theta_3 & 0 \\ 0 & 0 & 0 & \theta_4 \end{bmatrix}$ ,  $\theta_i = \begin{bmatrix} a_i & b_i \\ -b_i & a_i \end{bmatrix}$ 
*
* The  $\Phi$  matrix may or may not have all the three 2 x 2 matrices  $\theta_2$ , *
*  $\theta_3$  and  $\theta_4$  i.e.  $\Phi$  is a 2x2 or 4x4 or 6x6 or 8x8 matrix. The N x N *
* matrices  $P(k|k)$ ,  $\Phi$  and  $Q$  are stored at Loc_of_P, Loc_of_Phi and *
* Loc_of_Q in data memory respectively. The Nx1 column vector  $x(k|k)$  *
* is stored at Loc_of_x in data memory. To find  $P(k+1|k)$ , the macro *
* uses two other macros namely MAT221 and MAT222 which operate on *
* 2x2 matrices as,
*
*          MAT221 : A * B (symmetric, lower-half is known) * A' -> B
*          MAT222 : A * B * C' + D (diagonal) -> B
*
* To calculate  $x(k+1)$ , it uses the macro PHI_X. The macro is called *
* as PRED_EST N,Loc_of_Phi,Loc_of_P,Loc_of_x,Loc_of_Q e.g. PRED_EST *
* 8,311h,411h,511h,611h. NOTE that the macro uses locations 60h-63h *
* of data memory.
*****
*
PRED_EST $MACRO N,Loc_of_Phi,Loc_of_P,Loc_of_x,Loc_of_Q
  LARP 1 ;set ARP=1.
  LRLK AR1,:Loc_of_Phi: ;point AR1 to location of  $\Phi(1,1)$ .
  LRLK AR2,:Loc_of_P: ;point AR2 to location of  $P(1,1)$ .
  LRLK AR3,:Loc_of_Q: ;point AR3 to location of  $Q(1,1)$ .
  $IF N.V = 2 ;is N = 2 ?
  MAT221 1 ;if so, perform  $P = \Phi * P * \Phi' + Q$ .
  PHI_X 1,:Loc_of_Phi:,:Loc_of_x: ;and find  $x = \Phi * x$ .
$ENDIF

```

```

$IF N.V = 4                                ;if not, is N =4 ?
MAT221 1                                  ; if so, find
MAT221 2                                  ; P =  $\Phi * P * \Phi' + Q$ .
MAT222 :Loc_of_Phi:+22H,:Loc_of_P:+20H,:Loc_of_Phi:
PHI_X 2,:Loc_of_Phi:,:Loc_of_x: ;and find x =  $\Phi * x$ .
$ENDIF
$IF N.V = 6                                ;if not, is N = 6 ?
MAT221 1                                  ;if so,
MAT221 2                                  ;find
MAT221 3                                  ; P =  $\Phi * P * \Phi' + Q$ .
MAT222 :Loc_of_Phi:+22H,:Loc_of_P:+20H,:Loc_of_Phi:
MAT222 :Loc_of_Phi:+44H,:Loc_of_P:+40H,:Loc_of_Phi:
MAT222 :Loc_of_Phi:+44H,:Loc_of_P:+42H,:Loc_of_Phi:+22H
PHI_X 3,:Loc_of_Phi:,:Loc_of_x: ;and find x =  $\Phi * x$ .
$ENDIF
$IF N.V = 8                                ;if not, N is 8.
MAT221 1                                  ;find
MAT221 2                                  ; P =  $\Phi * P * \Phi' + Q$ .
MAT221 3
MAT221 4
MAT222 :Loc_of_Phi:+22H,:Loc_of_P:+20H,:Loc_of_Phi:
MAT222 :Loc_of_Phi:+44H,:Loc_of_P:+40H,:Loc_of_Phi:
MAT222 :Loc_of_Phi:+66H,:Loc_of_P:+60H,:Loc_of_Phi:
MAT222 :Loc_of_Phi:+44H,:Loc_of_P:+42H,:Loc_of_Phi:+22H
MAT222 :Loc_of_Phi:+66H,:Loc_of_P:+62H,:Loc_of_Phi:+22H
MAT222 :Loc_of_Phi:+66H,:Loc_of_P:+64H,:Loc_of_Phi:+44H
PHI_X 4,:Loc_of_Phi:,:Loc_of_x: ;and find x =  $\Phi * x$ .
$ENDIF
$ENDM

```

***** This macro is used by PRED_EST *****

MAT221 \$MACRO ALOC,BLOC

T_221_1: .set 60H

T_221_2: .set 61H

T_221_3: .set 62H

T_221_4: .set 63H

*

***** find D = A*B.

*

* find D(1,1) = A(1,1)*B(1,1)+ A(1,2)*B(2,1).

LT *+,2 ;(TREG) = A(1,1).

MPY *0+,1 ;(PREG) = A(1,1)*B(1,1).

LTP *- ,2

MPY *+ ;(PREG) = A(1,2)*B(2,1).

MPYA *- ,1 ;(PREG) = A(1,2)*B(2,2).

SACH T_221_1 ;(T_222_1) = D(1,1) = A(1,1)*B(1,1)+A(1,2)*B(2,1).

*

* find D(1,2) = A(1,1)*B(2,1)+A(1,2)*B(2,2).

LTP *0+,2

MPY *0-,1 ;(PREG) = A(1,1)*B(2,1).

LTA *+,2 ;(TREG) = A(2,1).

SACH T_221_2 ;(T_222_2) = D(1,2) = A(1,1)*B(2,1)+A(1,2)*B(2,2).

```

*
* find  $D(2,1) = A(2,1)*B(1,1)+A(2,2)*B(2,1)$ .
  MPY *0+,1          ;(PREG) =  $A(2,1)*B(1,1)$ .
  LTP *-,2
  MPY *+              ;(PREG) =  $A(2,2)*B(2,1)$ .
  MPYA *-,1           ;(PREG) =  $A(2,2)*B(2,2)$ .
  SACH T_221_3        ;(T_222_3) =  $D(2,1) = A(2,1)*B(1,1)+A(2,2)*B(2,1)$ .
*
* find  $D(2,2) = A(2,1)*B(2,1)+A(2,2)*B(2,2)$ .
  LTP *0-,2
  MPY *0-,1           ;(PREG) =  $A(2,1)*B(2,1)$ .
  LTA T_221_1         ;(TREG) =  $D(1,1)$ .
  SACH T_221_4        ;(T_222_4) =  $D(2,2) = A(2,1)*B(2,1)+A(2,2)*B(2,2)$ .
*
***** find  $E=D*A'$ .
* find  $E(1,1) = D(1,1)*A(1,1)+D(1,2)*A(1,2)$ .
  MPY *+              ;(PREG)= $D(1,1)*A(1,2)$ .
  LTP T_221_2         ;(TREG) =  $D(1,2)$  ; (ACCH) =  $D(1,1)*A(1,2)$ .
  MPY *,3             ;(PREG) = $D(1,2)*A(1,2)$ .
  ADDH *+,1           ;(ACCH) = $D(1,2)*A(1,2)+ D(1,1)*A(1,2) + G(1,1)$ .
  LT T_221_4          ;(TREG) =  $D(2,2)$ .
  MPYA *-,2           ;(PREG) =  $D(2,2)*A(1,2)$ .
  SACH *0+,0,1        ; $E(1,1) = D(1,1)*A(1,1)+D(1,2)*A(1,2)+G(1,1)$ .
*
* find  $E(2,1) = D(2,1)*A(1,1)+D(2,2)*A(1,2)$ .
  LTP T_221_3         ;(TREG) =  $D(2,1)$  ; (ACCH)=  $D(2,2)*A(1,2)$ .
  MPY *0+             ;(PREG) =  $D(2,1)*A(1,1)$ .
  MPYA *+,2           ;(PREG) =  $D(2,1)*A(2,1)$ .
  SACH *+,0,1         ; $E(2,1) = D(2,1)*A(1,1)+D(2,2)*A(1,2)$ .
*
* find  $E(2,2) = D(2,1)*A(2,1)+D(2,2)*A(2,2)$ .
  LTP T_221_4         ;(TREG) =  $D(2,2)$  ; (ACCH)=  $D(2,1)*A(2,1)$ .
  MPY *0+             ;(PREG) =  $D(2,2)*A(2,2)$ .
  MAR *+,3            ;point to next diagonal block of  $\Phi$ .
  ADDH *+,2           ;(ACCH) =  $D(2,1)*A(2,1) + G(2,2)$ .
  APAC                ;(ACCH) =  $D(2,1)*A(2,1) + G(2,2) + D(2,2)*A(2,2)$ .
  SACH *0+            ; $E(2,2) = D(2,1)*A(2,1)+D(2,2)*A(2,2)+G(2,2)$ .
  MAR *+,1            ;modify AR2 to point to next diagonal block of P.
$ENDM

***** This macro is used by PRED_EST *****
MAT222 $MACRO ALOC,BLOC,CLOC
T_222_1: .set 60H
T_222_2: .set 61H
T_222_3: .set 62H
T_222_4: .set 63H
  LRLK AR1,:ALOC:      ;point AR1 to location of A.
  LRLK AR2,:BLOC:      ;point AR2 to location of B.
  LRLK AR3,:CLOC:      ;point AR3 to loaction of C.
*
***** find  $D=A*B$ .
*

```



```

* find D(1,1) = A(1,1)*B(1,1)+ A(1,2)*B(2,1).
  LT *+,2          ;(TREG) = A(1,1)
  MPY *0+,1        ;(PREG) = A(1,1)*B(1,1).
  LTP *- ,2
  MPY *+           ;(PREG) = A(1,2)*B(2,1).
  MPYA *0-,1       ;(PREG) = A(1,2)*B(2,2) .
  SACH T_222_1     ;(T_222_1) = D(1,1)=A(1,1)*B(1,1)+A(1,2)*B(2,1).
*
* find D(1,2) = A(1,1)*B(1,2)+A(1,2)*B(2,2).
  LTP *0+,2
  MPY *- ,1        ;(PREG) = A(1,1)*B(1,2) ;i=1,2.
  LTA *+,2         ;(TREG) = A(2,1).
  SACH T_222_2     ;(T_222_2) = D(1,2)=A(1,1)*B(1,2)+A(1,2)*B(2,2).
*
* find D(2,1) = A(2,1)*B(1,1)+A(2,2)*B(2,1).
  MPY *0+,1        ;(PREG) = A(2,1)*B(1,1).
  LTP *- ,2
  MPY *+           ;(PREG) = A(2,2)*B(2,1).
  MPYA *0-,1       ;(PREG) = A(2,2)*B(2,2) .
  SACH T_222_3     ;(T_222_3) = D(2,1) = A(2,1)*B(1,1)+A(2,2)*B(2,1).
*
* find D(2,2) = A(2,1)*B(1,2)+A(2,2)*B(2,2).
  LTP *0+,2
  MPY *- ,3        ;(PREG) = A(2,1)*B(1,2).
  LTA T_222_1      ;(TREG) = D(1,1).
  SACH T_222_4     ;(T_222_4) = D(2,2) = A(2,1)*B(1,2)+A(2,2)*B(2,2).
*
**** find E = D*C'.
*
* find E(1,1) = D(1,1)*C(1,1)+D(1,2)*C(1,2).
  MPY *+           ;(PREG) = D(1,1)*C(1,1)
  LTP T_222_2
  MPY *0+,3        ;(PREG) = D(1,2)*C(1,2)
  MPYA *- ,2       ;(PREG) = D(1,2)*C(2,2)
  SACH *+,0,3      ;E(1,1)= D(1,1)*C(1,1)+D(1,2)*C(1,2).
*
* find E(1,2) = D(1,1)*C(2,1)+D(1,2)*C(2,2).
  LTP T_222_1
  MPY *+,2         ;(PREG) = D(1,1)*C(2,1)
  LTA T_222_4      ;(TREG) = D(2,2).
  SACH *0+,0,3     ;E(1,2) = D(1,1)+C(2,1)+D(1,2)*C(2,2).
*
* find E(2,2) = D(2,1)*C(2,1)+D(2,2)*C(2,2).
  MPY *-           ;(PREG) = D(2,2)*C(2,2)
  LTP T_222_3
  MPY *0-         ;(PREG) = D(2,1)*C(2,1)
  MPYA *+,2       ;(PREG) = D(2,1)*C(1,1)
  SACH *- ,0,3    ;E(2,1) = D(2,1)*C(2,1)+D(2,2)*C(2,2).
*
* find E(2,1) = D(2,1)*C(1,1)+D(2,2)*C(1,2).
  LTP T_222_4
  MPY *- ,2        ;(PREG) = D(2,2)*C(1,2)

```

```

APAC
SACH *O+,0,1      ;E(2,1) = D(2,1)*C(2,1)+D(2,2)*C(2,2).
*
$ENDM

```

***** This macro is used by PRED_EST *****

```

PHI_X $MACRO M,Loc_of_Phi,Loc_of_x
p_x_1 .set 60H
$VAR P          ;P variable is used as offset to address of
$ASG 0 TO P.V   ;vector x while multiplying blocks of  $\Phi$ .
LARK ARO,21H     ;ARO is used to point to successive blocks of  $\Phi$ .
LRLK AR1,:Loc_of_Phi: ;AR1 points to location of  $\Phi(1,1)$ .
$LOOP :M.V:      ;loop counter contains # of 2 x 2 blocks of  $\Phi$ .
LT *+           ;(TREG) =  $\Phi(2*i-1,2*i-1)$  ; i=1..M.
MPY :Loc_of_x+:P.V: ;(PREG) =  $\Phi(2*i-1,2*i-1)*x(2*i-1)$ .
SPH p_x_1        ;(p_x_1) =  $\Phi(2*i-1,2*i-1)*x(2*i-1)$ .
MPY :Loc_of_x+:P.V:+1 ;(PREG) =  $\Phi(2*i-1,2*i-1)*x(2*i)$ .
LTP *O+          ;(TREG)= $\Phi(2*i-1,2*i)$ ;(ACCH)= $\Phi(2*i-1,2*i-1)*x(2*i)$ .
MPY :Loc_of_x+:P.V: ;(PREG) =  $\Phi(2*i-1,2*i)*x(2*i-1)$ .
MPYS :Loc_of_x+:P.V:+1 ;(PREG)= $\Phi(2*i-1,2*i)*x(2*i)$ ;(ACCH)=(ACCH)-(PRI

SACH :Loc_of_x+:P.V:+1 ;x(2*i)=(ACCH).
PAC          ;(ACCH) =  $\Phi(2*i-1,2*i)*x(2*i)$ .
ADDH p_x_1    ;(ACCH)= $\Phi(2*i-1,2*i)*x(2*i)+\Phi(2*i-1,2*i-1)*x(2*i-1)$ 
SACH :Loc_of_x+:P.V: ;x(2*i-1)=(ACCH).
$ASG P.V+2 TO P.V ;increase variable by 2.
$ENDLOOP
$ENDM

```

```

*****
**                               MACRO : FILT_EST                               **
*****
**** FIND THE ESTIMATES OF FILTERED COVARIANCE AND STATE VECTOR ****
***** WHEN STATE-TRANSITION MATRIX  $\Phi$  HAS A SPECIAL STRUCTURE *****
*****
* This macro named " FILT_EST N, Loc_of_P, temp_1, Loc_of_R, *
* Loc_of_k, Loc_of_x, Loc_of_y " calculates the lower-half of filter- *
* ed estimates of error covariance matrix  $P(k+1|k+1)$  and state *
* vector  $x(k+1|k+1)$  using the equations : *
* *
*  $k(k+1) = P(k+1|k)h' / [hP(k+1|k)h' + r]$  , *
*  $P(k+1|k+1) = P(k+1|k) - P(k+1|k)H'k'(k+1)$  *
* and  $x(k+1|k+1) = x(k+1|k) - k(k+1)[hx(k+1|k) - y]$  *
* *
* and stores them in  $P(k+1|k)$ 's and  $x(k+1|k)$ 's places. The matrix  $\Phi$  *
* and vector h have the following special structure : *
* *
* 
$$\Phi = \begin{bmatrix} \theta_1 & 0 & 0 & 0 \\ 0 & \theta_2 & 0 & 0 \\ 0 & 0 & \theta_3 & 0 \\ 0 & 0 & 0 & \theta_4 \end{bmatrix}, \theta_i = \begin{bmatrix} a_i & b_i \\ -b_i & a_i \end{bmatrix}$$

* *
* and  $h = [h_1 \mid h_2 \mid h_3 \mid h_4]$ ;  $h_i = [1 \ 0]$  *
* *
* The  $\Phi$  matrix may or may not have all the three 2 x 2 matrices  $\theta_2$ , *
*  $\theta_3$  and  $\theta_4$  i.e.  $\Phi$  is a 2x2 or 4x4 or 6x6 or 8x8 matrix. Same is *
* true for vector h which is not stored because of known 1's and *
* 0's. The N x N matrices  $P(k+1|k)$ , Nx1 column vector  $x(k+1|k)$ , scalar *
* r are stored at Loc_of_P, Loc_of_x and Loc_of_R in data memory. The *
* observation input is stored at Loc_of_y in external data memory *
* and the filtered output  $[x_1(k+1|k+1) + x_3(k+1|k+1) + x_5(k+1|k+1) +$  *
*  $x_7(k+1|k+1)]$  is stored at Loc_of_y+1. The NxN scratch pad temp_1 *
* of data memory is used for temporary storage. The kalman gain k is *
* stored at Loc_of_k. To calculate  $P(k+1|k+1)$ , it uses the macro *
* Div_Vect. This is a special macro which uses XF flag to communi- *
* cate with I/O devices. The macro is called Filt_Est N, Loc_of_P, *
* temp_1, Loc_of_R, Loc_of_k, Loc_of_x, Loc_of_y. *
*****
Filt_Est $MACRO N, Loc_of_P, temp_1, Loc_of_R, Loc_of_k, Loc_of_x, Loc_of_y
$VAR M, P, Q ; M, P and Q are dummy variables.
$IF N.V = 2
$ASG 1 TO M.V ; if P is 2 X 2, then M = 1.
$ENDIF
$IF N.V = 4
$ASG 2 TO M.V ; if P is 4 X 4, then M = 2.
$ENDIF
$IF N.V = 6

```

```

$ASG 3 TO M.V          ;if P is 6 X 6, then M = 3.
$ENDIF
$IF N.V = 8
$ASG 4 TO M.V          ;if P is 8 X 8, then M = 4.
$ENDIF
LRLK AR2,:Loc_of_P     ;point AR2 to location of matrix P.
*
*find z=Ph'.
LARP 2
*get new data
RXF                    ;reset XF flag to inform that 'C25 is ready
*(1)                   ;for data.
LARK ARO,20H
LAC *0+
RPTK :M.V:-3
ADD *0+
ADD *0+
SACL :temp_1:          ;z(1) = (ACCL) =  $\sum_{i=1}^{N/2} P(2i-1,1)$ 
*
*(2)
SBRK :M.V:*20H-10H
LAC *
ADRK 11H
RPTK :M.V:-3
ADD *0+
ADD *0+
SACL :temp_1:+1        ;z(2)=(ACCL) =  $[ P(2,1) + \sum_{i=2}^{N/2} P(2i-1,2) ]$ .
*
*(3)
$IF M.V >= 2           ;is P 's order >= 4 ?
SBRK (:M.V:-2)*20H+21H
LAC *
ADRK 2
RPTK :M.V:-3
ADD *0+
ADD *0+
SACL :temp_1:+2        ;z(3)=(ACCL) =  $[ P(3,1) + \sum_{i=2}^{N/2} P(2i-1,3) ]$ .
*
*(4)
SBRK (:M.V:-2)*20H+12H
LAC *+
MAR *+
ADD *
ADRK 11H
RPTK :M.V:-4
ADD *0+
ADD *0+
SACL :temp_1:+3        ;z(4)=(ACCL)=  $[P(4,1)+P(4,3)+ \sum_{i=3}^{N/2} P(2i-1,4)]$ .
$ENDIF
*
*(5)
$IF M.V >= 3           ;is P's order >= 6 ?

```

```

LARK AR0,2H
SBRK (:M.V:-3)*20H+23H
LAC *0+
ADD *0+
ADD *
$IF M.V = 4
ADRK 20H
ADD *
$ENDIF
SACL :temp_1:+4 ; z(5)=(ACCL)=[  $\sum_{i=1}^{N/2-1} P(5,2i-1)+\{P(7,5), \text{ if } N=8\}$  ].
* ; i=1
*(6)
LRLK AR2,:Loc_of_P:+50H
LAC *0+
ADD *0+
ADD *
$IF M.V = 4
ADRK 11H
ADD *
$ENDIF
SACL :temp_1:+5 ; z(6)=(ACCL) = [  $\sum_{i=1}^{N/2-1} P(6,2i-1)+\{P(7,6), \text{ if } N=8\}$  ].
$ENDIF ; i=1
*
*(7)
$IF M.V >= 4 ; is P's order = 8 ?
SBRK 5H
LAC *0+
ADD *0+
ADD *0+
ADD *
SACL :temp_1:+6 ; z(7)=(ACCL) =  $\sum_{i=1}^{N/2-1} P(7,2i-1)$ .
* ; i=1
*(8)
ADRK 0AH
LAC *0+
ADD *0+
ADD *0+
ADD *
SACL :temp_1:+7 ; z(8)=(ACCL) =  $\sum_{i=1}^{N/2-1} P(8,2i-1)$ .
$ENDIF ; i=1
*done with Ph'.
*
*find hPh'
$ASG 2 TO P.V
LAC :temp_1:
$LOOP :M.V:-1
ADD :temp_1:+:P.V:
$ASG P.V+2 TO P.V
$ENDLOOP ; (ACCL) = h*P*h'.
ADD :Loc_of_R ; (ACCL) = h*P*h'+r.
*
*find k = Ph'/(hPh'+r).

```

```

SACL 060h                ;store (hPh'+r) in 060h for Div_Vect macro call.
Div_Vect :N.V:,:temp_1:,:Loc_of_k: ;find k and store it at Loc_of_k.
*
*find lower half of P = P - Ph'k'.
$ASG N.V TO P.V          ;assign N to variable P.
LRLK AR2,:Loc_of_P:+(N.V:-1)*10H ;point AR1 to last row of P.
$LOOP :N.V:              ;loop for # of rows of P.
$ASG 0 TO Q.V
LT :temp_1+:P.V:-1
$LOOP :P.V:
ZALR *                   ;(ACCH)=P(i,j),i=1..N,j<=i.
MPY :Loc_of_k+:Q.V:      ;(PREG)=D(i,j) ;D=P*h'*k'.
SPAC                     ;do P(i,j)-D(i,j) ;i=1..N,j<=i.
SACH *+                 ;store in the corresponding place.
$ASG Q.V+1 TO Q.V
$ENDLOOP
SBRK :P.V:+10H           ;point AR3 to next upper row of P.
$ASG P.V-1 TO P.V
$ENDLOOP
*
*
*find  $x(n|n) = x(n|n-1) - k(n) * [h(n) * x(n|n-1) - y(n)]$ 
*
* Read input
* new data is written at Loc_of_y.
SXF                     ;set XF flag to acknowledge data acquisition.
LRLK AR2,:Loc_of_y:
LAC *+
XORK 8000H
SACL 60h                ;store 2's comp. version of y at 60h.
$ASG 2 TO Q.V
LAC :Loc_of_x:
$LOOP :M.V:-1
ADD :Loc_of_x+:Q.V:
$ASG Q.V+2 TO Q.V
$ENDLOOP                ;(ACCL) = h*x.
SUB 60H                 ;(ACCL) = h*x - y.
SACL 60H                ;(060h) = h*x-y.
*
LT 060h                 ;load TREG with (h*x-y).
$ASG 0 TO Q.V
$LOOP :N.V:             ;loop for # of entries.
ZALR :Loc_of_x+:Q.V:    ;(ACCH) = x(i), i=1..N.
MPY :Loc_of_k+:Q.V:     ;(PREG) = f(i);f=k*(h*x-y),i=1..N.
SPAC                   ;(ACCH) = x(i)-f(i);f=k*(h*x-y),i=1..N.
SACH :Loc_of_x+:Q.V:    ;store the entry in vector x's place.
$ASG Q.V+1 TO Q.V
$ENDLOOP
*
$ASG 2 TO Q.V
LAC :Loc_of_x:          ;(ACCL) = x(1).
$LOOP :M.V:-1
ADD :Loc_of_x+:Q.V:
$ASG Q.V+2 TO Q.V
$ENDLOOP

```

```

XORK 8000H
*write output at Loc_of_y+1.
  SACL *                ;(Loc_of_y)=(ACCL)=  $\sum x(i+1)$ .
  LARK ARO,10H          ;restore ARO for normal addressing.
* $ENDM

***** This macro is used by FILT_EST *****
Div_Vect $MACRO N,A_LOC,B_LOC
  $VAR M
  $ASG 0 TO M.V
DENOM:      .set 060h    ;contains denominator beforehand.
  LT DENOM          ;(TREG) = DENOM.
  LAC DENOM          ;(ACCL) = DENOM.
  ABS                ;get absolute value of denominator.
  SACL DENOM          ;store it in DENOM.
  $LOOP :N.V:        ;loop-counter contains # of entries of vector.
DIV3:M.V: MPY :A_LOC+:M.V:      ;(PREG) = Vect(i)*DENOM ,i=1..N.
  PAC                ;(ACC) = (PREG).
  BGZ DIV1:M.V:      ;check for sign of quotient.if +ve go to DIV1?
  ZALH :A_LOC+:M.V:    ;load ACCH with Vect(i) i.e. numerator.
  ABS                ;get absolute value of numerator.
  RPTK 14
  SUBC DENOM          ;perform division.
  NEG                ;negate quotient.
  B DIV2:M.V:        ;go to DIV2?
DIV1:M.V: ZALH :A_LOC+:M.V:      ;load ACCH with Vect(i) i.e. numerator.
  ABS                ;take absolute value of numerator.
  RPTK 14
  SUBC DENOM          ;perform division.
DIV2:M.V: SACL :B_LOC+:M.V:      ;store the quotient in its place..
  $ASG M.V+1 TO M.V
$ENDLOOP
$ENDM

```

References

- [1] R. E. Kalman, "A New Approach to Linear Filtering and Prediction Problems," *Trans. of the ASME J. Basic Eng.* , 82, pp. 35-45, Mar. 1960.
- [2] J. V. Candy, *Signal Processing : the Model-based Approach*. McGraw-Hill Inc. 1986.
- [3] J. K. Campbell, S. P. Synnot and G. J. Bierman, "Voyager Orbit Determination at Jupiter," *IEEE Trans. Automat. Contr.* , vol. AC-28, pp. 256-268, Mar. 1983.
- [4] B. G. Leibundgut, A. Rault and F. Gendreau, "Applications of Kalman Filtering to Demographic Model," *IEEE Trans. Automat. Contr.* , vol. AC-28, pp. 427-434, Mar. 1983.
- [5] Kalman Filtering : Theory and Applications, edited by H. W. Sorenson, IEEE Press, 1985.
- [6] A. Gelb, *Applied Optimal Estimation*. Cambridge, MA: MIT Press, 1974.
- [7] B. D. O. Anderson and J. B. Moore, *Optimal Filtering*. Englewood Cliffs, NJ:Prentice-Hall, 1979.
- [8] A. Jazwinski, *Stochastic Processes and Filtering Theory*. New York:Academic Press, 1970.
- [9] D. J. Quarmby, *Signal Processor Chips*. Granada Publishing Ltd. 1984.
- [10] Texas Instruments Inc. , Second-Generation TMS320 User's Guide, 1987.
- [11] Texas Instruments Inc. , TMS320C50 Digital Signal Processor Preview Bulletin, 1989.
- [12] J. Tan and N. Kyriakopoulos, "Implementation of a Tracking Filter on a digital signal processor," *IEEE Trans. Ind. Electron.* vol. ID-35, pp. 126-134, Feb. 1988.

- [13] Hen-geul Yel, "Real-time Implementation of a Narrow-band Kalman Filter with a Floating-point Processor DSP32," *IEEE Trans. Ind. Electron.* vol. ID-37, pp. 13 -17, Feb.1990.
- [14] C. K. Chui and G. Chan, *Kalman Filtering*. Springer Verlag, 1987.
- [15] S. M. Bozic, *Digital and Kalman Filtering*. London:Edward Arnold, 1986.
- [16] R. G. Brown, *Introduction to Random Signal Analysis and Kalman Filtering*. New York:Wiley, 1983.
- [17] P. S. Maybeck, *Stochastic Models ,Estimation, and Control Volume 1*. New York :Academic Press, 1979.
- [18] I. A. Gura and A. B. Bierman, "On Computational Efficiency of Linear Filtering Algorithms," *Automatica*, vol. 7, pp. 299-314, 1971.
- [19] G. Amit and U. Shaked, "Minimization of Roundoff Errors in Digital Realization of Kalman Filters," *IEEE Trans. Acoust. Speech,Signal Processing*, vol. ASSP-37, pp. 1980-1982, Dec.1989.
- [20] D. Williamson, "Finite Wordlength Design of Digital Kalman Filters for State Estimation," *IEEE Trans. Automat. Contr.* , vol. AC-30, Oct. , 1985.
- [21] H. W. Sorenson and J. E. Sacks, "Recursive Fading Memory Filtering ," *IEEE Trans. Automat. Contr.* , vol. AC-28, pp. 185-194, Mar. 1983.
- [22] J. H. Rigden, *Physics and the Sound of Music*. New York:Wiley, 1977.
- [23] Texas Instruments Inc. , TMS320C2X Software Development System, 1988.