OKLAHOMA STATE UNIVERSITY

A METHOD FOR DETECTING SOFTWARE FAULTS DURING UDU^T COVARIANCE CALCULATIONS USED IN KALMAN FILTERING

Thesis Approved:

Gary E. Young
Thesis Adviser

[illegible]

[illegible]

Thomas C. Collins
Dean of the Graduate College

A METHOD FOR DETECTING SOFTWARE FAULTS DURING UDU^T COVARIANCE CALCULATIONS USED IN KALMAN FILTERING

By

MICHAEL R. MOAN

Bachelor of Science

Oklahoma State University

Stillwater, Oklahoma

1986

Submitted to the Faculty of the Graduate College
of the Oklahoma State University
in partial fulfillment of the requirements
for the Degree of
MASTER OF SCIENCE
July, 1993

ACKNOWLEDGMENTS

I express my sincere gratitude and appreciation to my major adviser, Dr. Gary Young, for his guidance, support, and encouragement throughout the course of this study. I also extend my appreciation to the other committee members, Dr. L. L. Hoberock and Dr. E. A. Misawa.

Special appreciation is offered to the School of Mechanical Engineering at Oklahoma State University for their assistance throughout my academic career.

Finally, many thanks and much love go to my family for their enduring support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

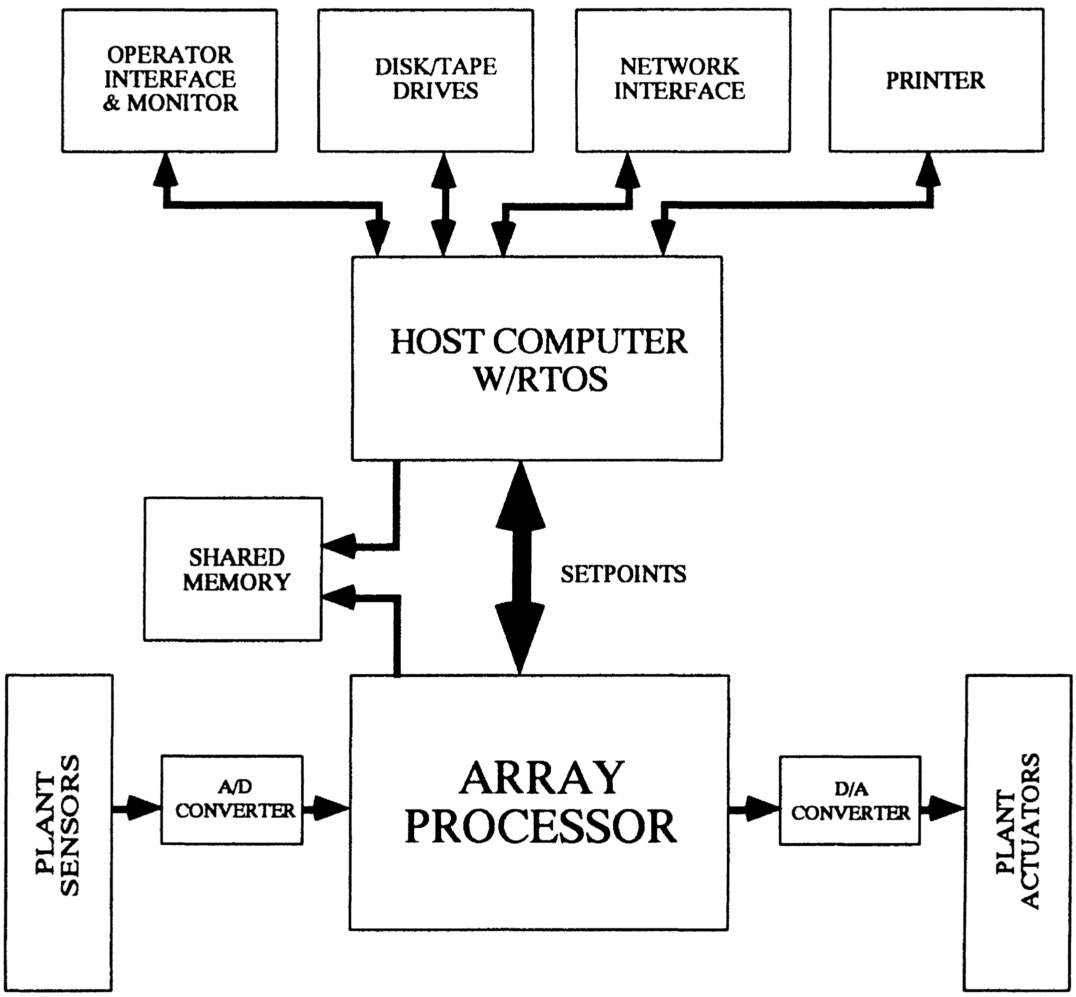
Figure Page

CHAPTER I

INTRODUCTION

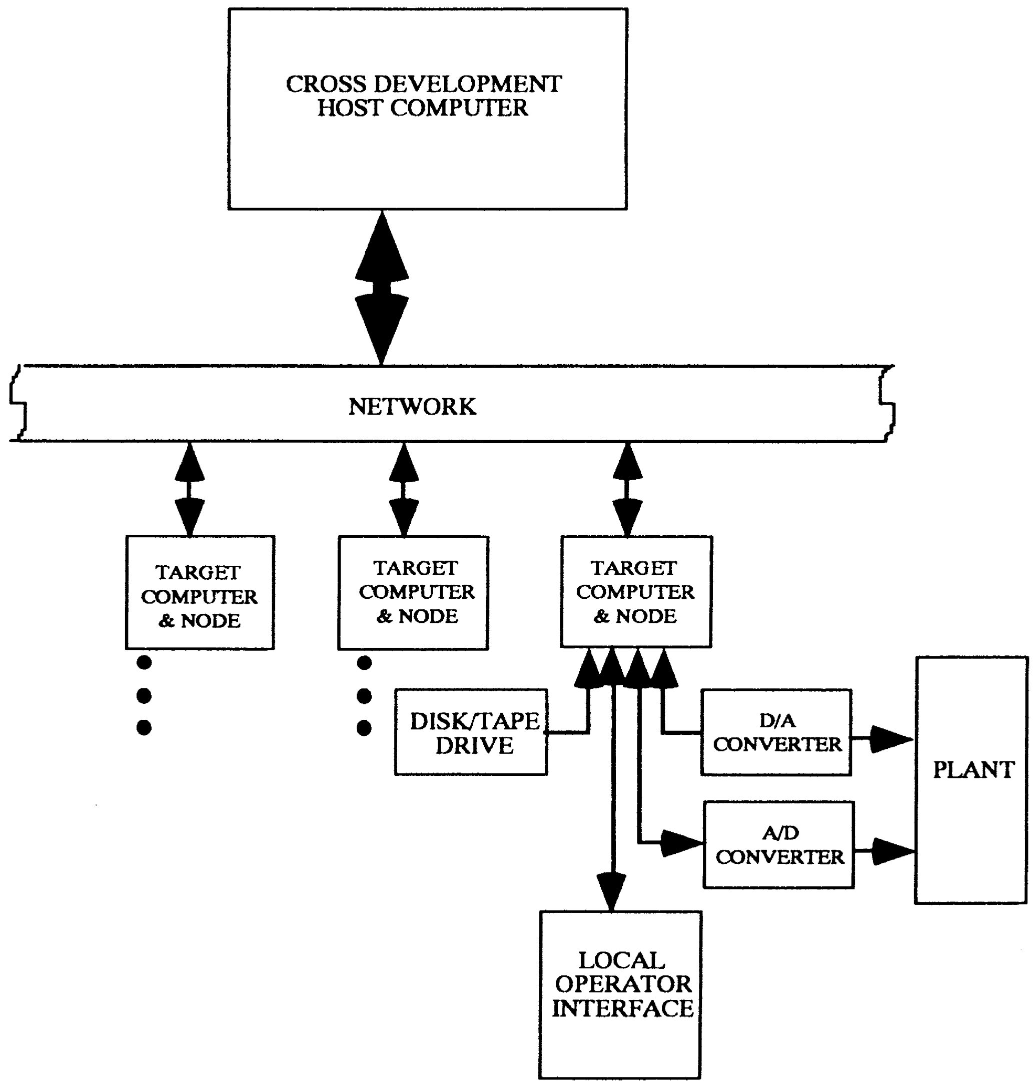
In many mechanical control applications, the proliferation of low cost general purpose microcomputers has allowed networking across large spatial distances and the development of complex distributed control systems. Many of these control systems implement algorithms with hard real-time constraints. For stability of the controlled process, it may be required that not a single control output be missed, corrupted, or delayed. Methods for implementing algorithms in fault tolerant, reliable, and numerically stable fashion are critical to meeting these demanding constraints. Because of the importance of prior research in these areas, this thesis reviews some of the existing methods for achieving fault tolerant and reliable algorithms. In addition to the review, the contribution of this thesis involves the use of a concept for encoding control algorithms so that software failures may be detected promptly before control actions are performed or sensor/actuator failure decisions are made. In this thesis, a software failure is defined to be a non-catastrophic circumstance in which the software continues to run but cannot correctly compute the intended results. A variety of computing environment faults or failures could cause a software failure of this definition, and they range from single chip MTBF failures to communication problems in multi-processing environments. In regard to the contribution, the proposed concept is applied to the Bierman algorithm for UDU^T time and measurement update of the error covariance of the Kalman filter [1, 2]. This algorithm was chosen as a representative algorithm because of its popularity in industrial applications for the sequential processing of measurement information. The methods to be discussed are intended for the application level of a software hierarchy. Many of the reviewed techniques were developed for stand-alone uniprocessors, but can or have been extended to supervisory and distributed systems for node self-diagnostics and acceptance tests whose existence is to prevent corrupt information from being passed to higher levels of authority and control.

The following typical systems, shown in Figures 1a and 1b, describe the computing node environments which are addressed. For high throughput, high bandwidth applications such as found in modern digital signal processing and control, multiprocessor architectures with systolic arrays, transputers or digital signal processors (DSPs) have been used to implement high order filters and other computationally intensive algorithms (Bromley, Kung, Swartzlander et al.,1988) [3]. Such parallelism and concurrency have been needed because uniprocessor implementations have historically been restricted by sampling rates which are dictated by the time taken for one step of recursive filters such as the Kalman Filter. For these implementations, uniprocessors with Real Time Operating Systems (RTOS) are commonly relegated to supervisory tasks such as control of data flow into and out of the array processor, network interface, graphical user interface, statistical analysis, set pointing, data storage, and control of peripherals, while number crunching is left to the array processors (Jacklin, 1988) [4]. For this case, the self-tests and audits which add fault tolerance, failure detection and stability may be an additional responsibility of the supervisory processor. However, if the processing can be distributed among several uniprocessors without the need for an array processor, tasks associated with recursive computations as well as self-test and diagnostic tasks may be implemented on each node of a distributed system of uniprocessors as depicted in Figure 1b, some taking advantage of the services and facilities of a multi-tasking RTOS. As an alternative to multiprocessing with a distributed system of uniprocessors, DSP solutions which are currently available use



a. Controller design where array processor handle high throughput computations and host computer handles supervisorory tasks and operator interface.

Figure 1. Computing Node Environments



b. Controller design where control computations are distributed among many node computers which may or may not have large spatial distances between nodes. Target processor is responsible for controlling/servicing all attached peripherals and accepting network messages and commands.

Figure 1. Continued

both multiprocessing and real time operating systems for applications with high computational loads. They are available in both single and multiple board configurations.

CHAPTER II

BACKGROUND

While much research exists for general purpose modeling and the specification of real time systems and software, published research concerning the stability and fault tolerance of RTOS software implementations of control algorithms is limited. According to Kim [5], major issues associated with designing fault tolerant capabilities into hard-real-time distributed computing systems need to be resolved in the 1990s. To help resolve the issues, research is in progress on such techniques as N-version programming, Built-In-Test software, Data Redundancy, Checksums, Distributed Recovery Blocks, Comparing Schemes, and Triple Modular Redundancy. Also, modeling methods such as Petri Nets, Data Flow Diagrams, Finite State Automata, and State Charts provide tools for analysis. However, fault tolerant software strategies which exist in literature seem to be for generic applications or processes and not specifically related to particular control algorithms. To help fill this gap, subsequent sections review techniques which are common in the control community and should be considered for use as the self tests, data validations, acceptance tests, and other components of the overall fault tolerant software solution. As an aside, in the event of a permanent, non-correctable fault or failure, it is often a requirement of the system to reconfigure to work in the presence of the fault. The redefinition of processing responsibilities among the remaining processing nodes, or in the case of a uniprocessor, the remaining operable tasks, must be coordinated. Literature available on the stability of this reconfiguration process includes that of Mariton [6] and Srichander and Walker [7].

Software Fault Tolerance

To begin, we need to review a few of the general definitions and concepts of software fault tolerance. Similar to how redundancy, built-in self-tests, and diagnostics are used to add reliability to computing hardware, software is even more flexible in regard to the addition of redundancy and self-tests given sufficient computing resources and timing constraints. Checkpointing and roll back recovery are very common techniques. According to Kim [5] (1988), checkpointing refers to saving the state of computation on a secure device at various execution points called recovery points (RP). When a fault happens, the system is able to resume computation or "roll back" to the most recent RP after any necessary reconfiguration. To determine if a fault occurs, some form of acceptance test must be performed to indicate the fault. As our concern is with the substance of the acceptance test in the context of common control algorithms, Figures 2 and 3 are two fault tolerant schemes (Kim, 1988) which use checkpointing and acceptance tests and have been adapted to illustrate a state estimation process using the Kalman filter.

Figure 2 illustrates the use of primary and backup versions of Kalman filters in a Distributed Recovery Block scheme (Kim, 1988). This scheme uses multiple processors or nodes to achieve active redundancy by concurrently executing multiple versions of a software component. The same acceptance test is used for results from different versions of software. The scheme includes a time out mechanism such as a watch dog timer. Each recovery block consists of one or more routines, called "try blocks" by Kim, which compute functionally equivalent results. In the figure, the try blocks consist of Kalman filters and suboptimal filters. The acceptance test contains the criteria used for accepting the results. By Kim's definition, a recovery block could contain two or more try blocks. If desired, the scheme could be set up as a tandem system duplicating its running process with corresponding identical processes running on the other processors. Figure 3 shows an adaptation of a conversation scheme (Kim, 1988).

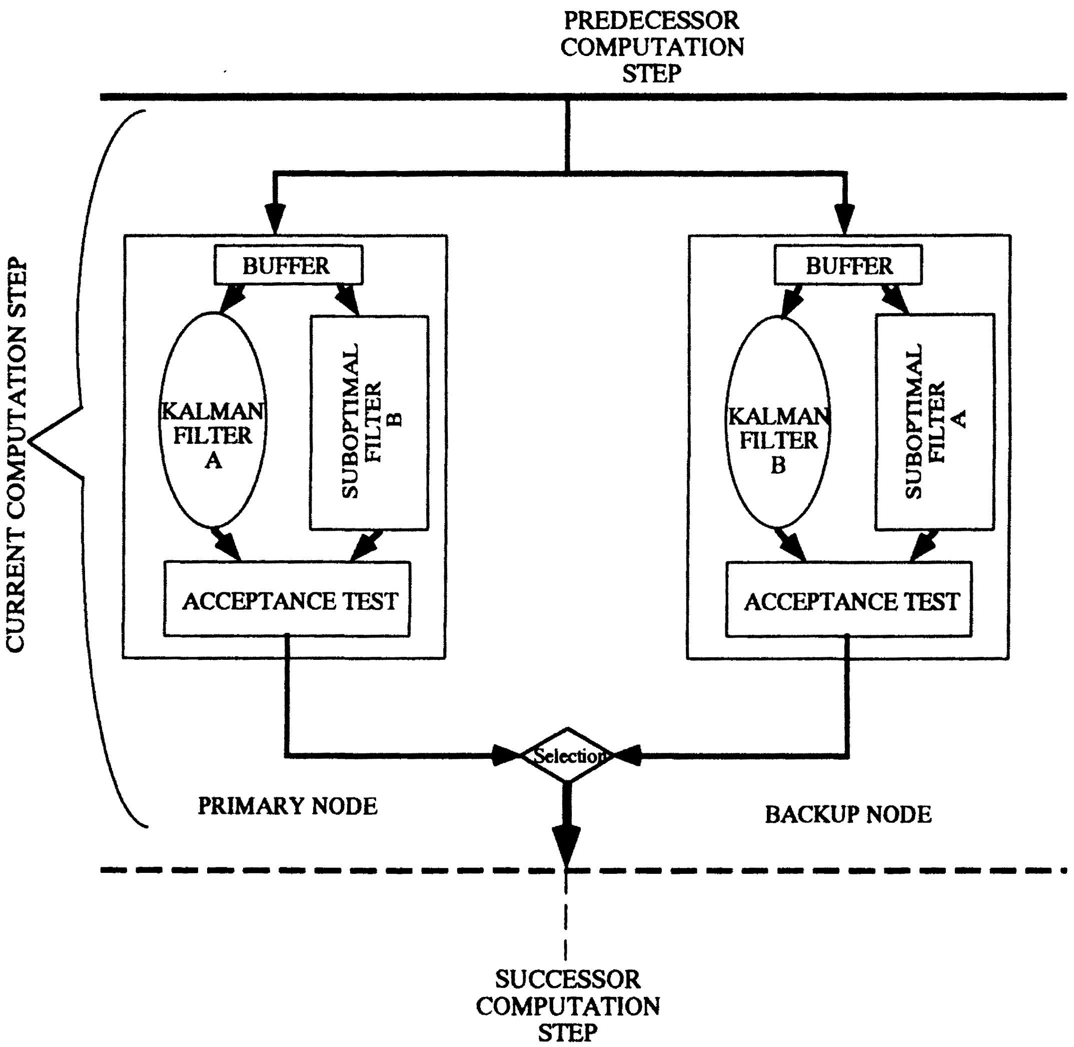


Figure 2. Fault Tolerant State Estimation Scheme Combined With a Distributed Recovery Block Structure

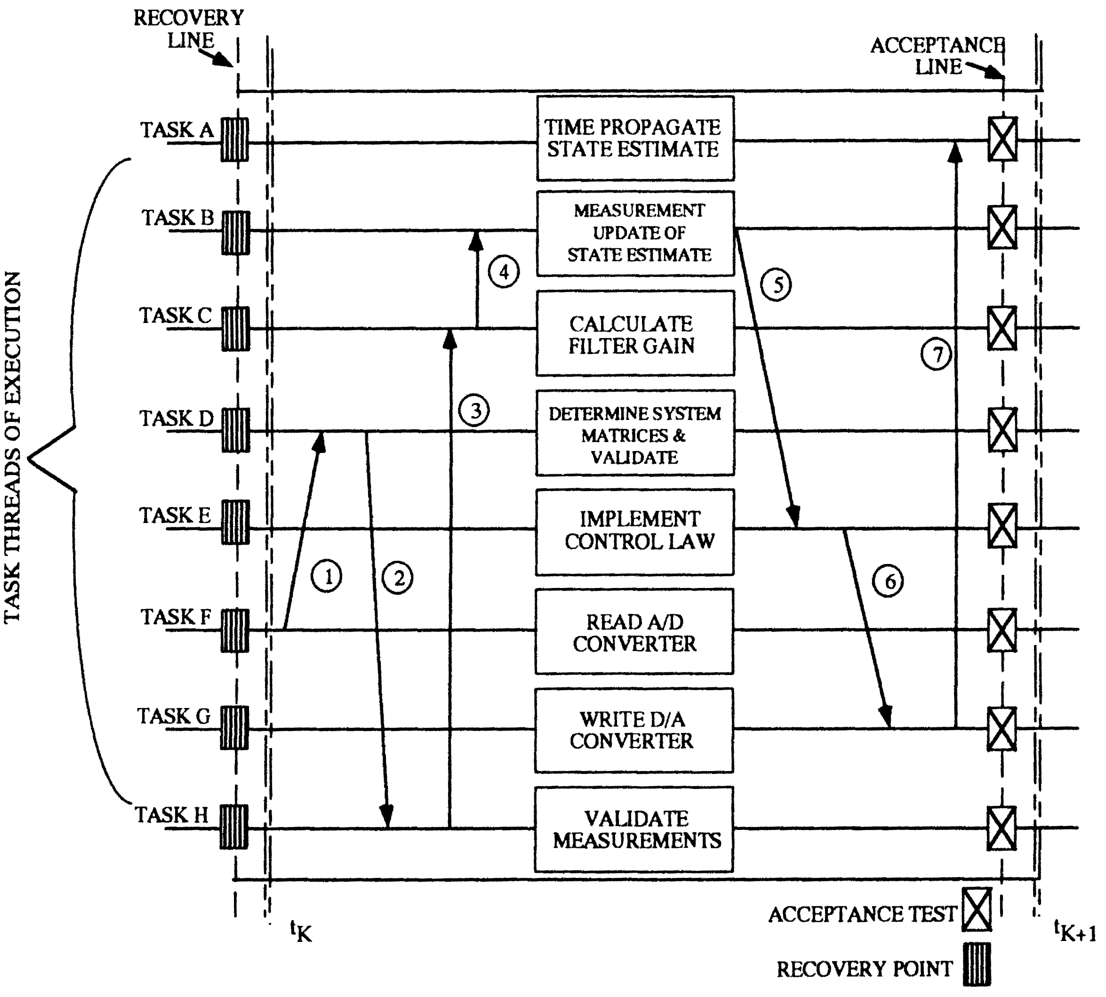


Figure 3. Schematic of Task Partitioning for Estimation and Control

This scheme illustrates how controller functions might be partitioned into a set of tasks which run concurrently, communicate between each other periodically, and deliver a result by the end of the time step regardless of missing communication or data. The tasks include state estimation functions and the control law functions. In both schemes, the most recent accepted state and covariance would be saved in a buffer at the recovery points, and upon failed acceptance of the primary results, the system would either restart from the previous recovery point, which may not be desirable when a state estimate is needed by the end of a time step, or would accept the state estimate from backup or secondary processes which might be suboptimal. Possible secondary processes might feature reduced real time computational load, assumptions of almost or completely time invariant and linear system response over short periods of operation, and the use of precomputed gains and state error covariance. Such secondary processes would be particularly applicable during instances when process noise dominates the system (Gylys, 1983) [8]. Secondary processes might also use lookup tables to determine noise levels under differing operating conditions.

Failure Modes and the Acceptance Test

Before considering what should be included in an acceptance test, it is appropriate to analyze how the implementation might be expected to fail. Since the acceptance test also represents software which could fail, the sophistication of the recovery points and acceptance tests should be balanced against the additional computational cost and complexity. Depending on the need for safety and reliability, software associated with fault tolerance should be parsimoniously applied. In regard to general failure modes, causes of software failure are language and design methodology dependent. Even with good software engineering practices, they are so varied that it is impossible to adequately test for every possibility before the software is in operational use. Once in operational use, processor failures resulting from chip MTBFs and communication failures during high speed data transfers can occur in a

MTBFs and communication failures during high speed data transfers can occur in a subtle manner and result in incorrect results. We would like to have an acceptance test which covers a large number of the potential faults. To this end, the following topics exist in literature and concern typical modes of failure for recursive control algorithms.

Numerical Stability

The acceptance tests might be expected to check for numerical stability at the end of each step of the recursion. Use of the Kalman filter to discuss numerical stability issues is appropriate because the Kalman filter is a part of the group of kernel algorithms used in a variety of applications including recursive parameter estimation and adaptive control (Astrom, p. 412, 1989) [9], (Clarke, p. 62, 1984) [10]. The following equations present typical nomenclature for a time variant, discrete, linear state space representation of the system model and measurement process, and the elements of a conventional Kalman filter.

Dynamic Model

$$x_{k+1} = \Phi_k x_k + \Gamma_k u_k + B_k \omega_k \tag{1}$$

with:

$$\begin{aligned}
&x_k = x(t_k) \in R^n, \quad \Phi_k = \Phi(t_k, t_{k+1}),\\
&\omega_k = \omega(t_k) \in R^p, \quad B_k = B(t_k, t_{k+1}),\\
&u_k = u(t_k) \in R^L, \quad \Gamma_k = \Gamma(t_k, t_{k+1}),\\
&E[\omega_k(i)] = 0, \quad E[\omega_k \omega_k^T] = Q_k \delta_{jj}
\end{aligned} \tag{2)-(8}$$

where x_k is the state vector to be estimated, $\hat{x}_k$ is the estimated state, u_k is the deterministic input vector, Φ_k and Γ_k and B_k are time variant, discrete time system matrices. The process noise vector ω_k is usually assumed to be a zero mean, gaussian sequence with constant variance, independent of and uncorrelated with the measurement noise sequence. Q_k is a positive semidefinite process noise covariance matrix.

Measurement Model

$$z_k = H_k x_k + v_k \tag{9}$$

with:

$$z_k = z(t_k) \in R^m,\ H_k = H(t_k)$$
$$v_k = v(t_k) \in R^m$$
$$E[v_k(i)] = 0,\quad E[v_k v_k^T] = R_k \delta_{jj} \qquad (10)\text{-}(14)$$

where z_k is the measurement vector to be processed, and H_k is the observation matrix. The measurement noise vector v_k is usually assumed to be a zero mean, gaussian sequence with constant variance, independent of and uncorrelated with the process noise. R_k is a positive definite measurement noise covariance matrix.

Computation of Kalman Gain

$$K_k = P_{k\,|\,k-1} H_k^T (H_k P_{k\,|\,k-1} H_k^T + R_k)^{-1} \tag{15}$$

or

$$K_k = P_{k\,|\,k} H_k^T R_k^{-1} \tag{16}$$

Conventional Measurement Update of Error Covariance

$$P_{k\,|\,k} = P_{k\,|\,k-1} - P_{k\,|\,k-1} H_k^T (H_k P_{k\,|\,k-1} H_k^T + R_k)^{-1} H_k P_{k\,|\,k-1} \tag{17}$$

Joseph's Form of Measurement Update of Error Covariance

$$P_{k\,|\,k} = (I - K_k H_k) P_{k\,|\,k-1} (I - K_k H_k)^T + K_k R_k K_k^T \tag{18}$$

State Estimate Based on Current Measurement

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k(z_k - H_k\hat{x}_{k|k-1}) \quad (19)$$

Time Propagation of State

$$\hat{x}_{k+1|k} = \Phi_k\hat{x}_{k|k} + \Gamma_k u_k \quad (20)$$

Time Propagation of Error Covariance

$$P_{k+1|k} = \Phi_k P_{k|k}\Phi_k^T + B_k Q_k B_k^T \quad (21)$$

Estimates of the initial state and error covariance, x(0) and P(0), are usually assumed to be known *a priori.*

The conventional form of the covariance measurement update for the Kalman filter can be numerically unstable when using single precision arithmetic or when the modeled process is unstable. Roundoff errors and over convergence can cause the state error covariance matrix to divergently loose symmetry and positive definiteness. Although inaccurate modeling of the system or discretization errors can also cause filter divergence (Gelb, 1974) [11], assume that accurate model structure and parameters are available at each step of the recursion in at least one of the try blocks, and momentarily that numerical issues are the primarily topic of concern for the acceptance test. Methods available for use in the acceptance tests and for correction of the conventional filter include averaging the error covariance matrix with its transpose, computing only the upper or lower part of the matrix, or adding to the diagonal elements upon detection of negative eigenvalues. Verhaegen and Van Dooren (1988) presented an analysis of error propagation due to roundoff which explains why these heuristic methods work for the conventional covariance update [12]. Having recognized that the conventional filter mechanization may be numerically unstable, the complexity of an acceptance test will depend upon whether the application uses a modified version of the conventional algorithm or an alternative, numerically stable mechanization.

The version of the error covariance measurement update, commonly called "Joseph's" form, does not require symmetry detection/correction of the error covariance matrix. Thus, when using "Joseph's" form, the acceptance test would not be required to perform such a test. Also, other numerically stable mechanizations retain symmetry by propagating a factored version of the error covariance matrix. They are often referred to as "square root" algorithms even though the methods may be based on Choleski, UDU^T or other factor types, and they may be based on sequential or simultaneous processing of the measurement vector. These filter mechanizations are equivalent to the original form in that they result in optimal rather than suboptimal estimates, and they can be more computationally expensive (Thorton and Bierman, 1981) [2] (Chin, 1983) [13] but generally give more accurate filter estimates and gains (Verhaegen and Van Dooren, 1988) [12]. Like "Joseph's" form, an acceptance test for a numerically stable filtering process would only involve ensuring that computations are performed correctly and would not involve checks for a positive definite error covariance matrix. Therefore, when the additional computational expense of an acceptance test is considered, the square root algorithm's become even more desirable, particularly when calculations are being performed with time varying system matrices and unstable process scenarios are known to exist.

Instead of waiting until the end of the time step, the acceptance test may be performed at subintervals of the main time frame. Breaking the acceptance test up into several smaller tests may be an option if sequential processing of measurement information if performed rather than simultaneous processing. When the noise covariance matrix is diagonal, the covariance measurement update can be done by sequentially processing one measurement at a time, thus allowing the computation to be tested at intermediate points after the update from each measurement. Because of the convenience of inverting scalars opposed to matrices, some of the most popular numerically stable versions for updating error covariance are based on sequential processing. Not much is lost, however, because sequential processing of the Kalman filter has long been advocated in cases of uncorrelated measurement noise as a method of avoiding the program storage and computation requirements for inverting the innovations covariance matrix $[HPH^T + R]$ (Sorenson, p. 256, 1966) (Gelb, p. 304, 1974) [14, 11]. Thus, sequential processing facilitates the detection of computation failures at an earlier time within the overall time step frame, and possible use of remaining time to perform correction or processing with secondary processes. With sequential processing, the filtering task can also be efficiently interrupted by other tasks between measurement iterations, with a reduced chance of corrupting information during context switches. The measurement updates do not have to be performed in any particular order, and individual measurements can be incorporated as they become available, without having to wait until all measurements are received and validated. Because sequential processing avoids the computational and storage requirements for an algorithm which explicitly determines the inverse of a matrix, no acceptance tests are necessary to ensure that the inversion was performed correctly. Furthermore, for less sophisticated RTOS environments, shorter and more independent calculations might prevent loss of information caused by non-maskable interrupts during longer time blocks of CPU usage required by simultaneous processing. Note that in one time step of sequential processing, the covariance matrix is not complete until the last measurement has been incorporated.

Need for Suboptimal Secondary Processes

Estimation and control systems frequently encounter missing, time delayed, and/or invalid sensor observations. The following situations can all cause invalid measurements: (1) sensor failures, (2) time skewing and improper ordering of data, (3) network induced delays, (4) intermittent loss of signals during transient periods of high noise, (5) sensor saturation, and (6) nonlinear sensor behavior. When measurement data is missing or unreliable, recursive estimation algorithms such as the Kalman filter do not give optimal estimates and would be expected to fail a fault tolerance acceptance test. In such cases, the alternative or secondary process must provide the suboptimal information necessary to continue the computation. This section discusses typical methods used to test and compensate for corrupt, missing, or time-delayed measurements.

As expected, heuristic methods for use in an acceptance test for measurement validation and inference for the Kalman filter exist throughout the literature. According to V. Gylys (1983) [8], when considering the robustness of an estimation process, distributional assumptions can be bad and/or measurements can be bad. Because bad measurements can exist, Gylys suggests that the pre-processing and screening of measurements to be used in the Kalman filter should "(1) screen against outliers, (2) detect leading and trailing edges of high amplitude noise bursts, (3) detect the onset and compensate for nonwhiteness in measurement noise, and (4) censor or bound measurements or estimates," and that "preprocessing may include conversion and prefiltering, computation of measurement residuals, and screening of residuals for rejection." For instance, electro-magnetic compatibility problems can cause severe measurement noise. Gylys also mentions that a $\pm$ scaled multiple of the innovations variance can be used as an acceptance interval to screen measurement residuals. Statistical inference based on the innovation or residual sequence $(z(k)-H(k)\hat{x}(k \mid k-1))$ is another method for validating measurements and monitoring software implementation. However, T. H. Kerr (1990, p. 944) [15], on validating linear systems software, points out that small residuals are necessary but not sufficient indicators of good filter performance and that similar statements can be made concerning statistically white residuals.

For use as a secondary process, an intuitively appealing method of Kalman filtering when a single measurement vector is missing is to simply skip the measurement update and rely on the time propagated state estimate. Guanrong Chen (1990) [16] showed that the predicted estimate of the state, $\hat{x}_{k|k-1} = \Phi_{k-1}\hat{x}_{k-1|k-1} + \Gamma_{k-1}u_{k-1}$, could replace the unknown optimal estimate, $\hat{x}_{k|k}$, and that convergence could be guaranteed for time invariant cases, for a single bit of missing data, when no other data is missed in the future. Motivation for using the estimate as a secondary process is that at the instant when the data is missing, a suboptimal estimate of the unknown state vector x_k is still needed in real time for control law or other purposes with hard time constraints. Usually, the possibility of system instability prevents waiting for late arrival of the missing data and the system must proceed to the successor time step, but Luck and Ray (1990) [17] and Zhang and Ray (1991) [18] have proposed a multi-step predictor for compensation of the effects of network induced delays.

Another alternative secondary process when data is missing is to greatly increase the assumed noise statistics associated with the invalid measurement. By increasing the assumed measurement noise for the missing information, less confidence is placed on the measurement in the computation of the optimal Kalman gain. If the diagonal elements of the measurement noise covariance matrix R associated with the invalid measurements are increased to a large number (approaching infinity), this approach is equivalent to the sequential processing technique of merely skipping the incorporation of the invalid measurement into the measurement update for state estimate $\hat{x}(k|k)$ and error covariance $P(k|k)$. However, this technique is not restricted to sequential processing of measurements. Lynch and Figueroa (1991) [19] use this method to improve the robustness of ultrasonic position estimates in the presence of missing observations resulting from both structural intermittence and stochastic intermittence.

Hardware Failure Detection and Isolation

Terminology such as "fault tolerant control" is generally associated with the detection, isolation, and reconfiguration (FDIR) of the control algorithms in the event of sensor or actuator failures. Because of the possibility of sensor and actuator failures, control system software often has the capability to gracefully degrade and avoid catastrophe immediately following the occurrence of a failure. The generation of residuals which develop a bias or exceed a threshold when a failure happens is one of the most basic concepts of these failure detection schemes. Therefore, we would expect that one or more of the redundant processes of a fault tolerant scheme would be involved in the generation of residual sequences for the purpose of testing for sensor and actuator failures. Thus, failure detection theories are available methods which would be applicable to the design of acceptance tests. There are more than five major survey papers concerning this area (Isermann, 1984) (Basseville, 1988) (Gertler, 1988) (Frank, 1990) (Willsky, 1976) (Panossian, 1988) [20-25] and several books.

General Considerations

Miscellaneous techniques which could be modified for use in an acceptance test exist throughout literature because difficult conditions, nonideal behavior, and limited resources have plagued digital controllers since the sixties. It is not the intent of this thesis to survey the general implementation issues which are covered in digital control textbooks. However, they do exist and are available to the interested reader. For instance, see Chapter 12 in Franklin, Powell, and Workman [26], Gelb [11], Chapter 11 in Astrom [9], or H. Hanselmann's survey [27] concerning implementation of digital controllers. These references cover basics of hardware speeds and architectures, fixed point and floating point arithmetic, controller structures, sensors, problems such as parameter scaling and saturation, and introductions to software design and programming issues. A good example of the amount of information available, although generally not in one place, is the fact that Hanselmann's survey cites over 200 references which in some way discuss implementation issues.

Detecting Computational Faults

Within a fault tolerant computing scheme, the potential exists to confuse software failures with sensor or actuator failures. Especially when information generated by a process is used by other tasks or nodes to make failure detection decisions, it would be desirable to have confidence that information contained in residual processes has not been corrupted by faulty software. In some applications, results of calculations may not be correct even if the processor passes its hardware self-test and is still able to communicate to the other processors in the networked/distributed environment. Therefore, the problem is to find a method of encoding algorithms with redundant information such that abnormal residual or parity relations resulting from faulty calculations can be differentiated from sensor or actuator failures. An additional benefit of such a method is that corrupt information and software processes can be identified before the information is used by other processing nodes within the system, thus preventing unwanted actions based on faulty information. An efficient technique providing such features would be of considerable value to networked or distributed computing environments consisting of multiple processing nodes among which the overall control responsibilities have been divided. The next chapter discusses methodology for one potential technique.

CHAPTER III

METHODOLOGY

Subsequent chapters of this thesis are concerned with the development of the software fault tolerance additions to the Bierman UDU^T error covariance algorithms. The Algorithm Based Fault Tolerance (ABFT) concept is used because it is a method of adding fault tolerance to matrix intensive calculations such as found in the Kalman filter. ABFT has the potential to be a method of verifying that consistent parameters have been input to the computational process and that the computation has been consistently performed for many types of control algorithms. Dormant software bugs, unexpected threads of execution, and inter-processor communication problems are a few of the situations that may be detectable. Also, the method is computationally cheaper than obtaining fault tolerance by using redundant processors and software coupled with a voting or a comparing scheme. This chapter concerns the specific methodologies used for adding software fault detection capabilities to the UDU^T error covariance algorithms.

Algorithm Based Fault Tolerance

Algorithm Based Fault Tolerance (ABFT) is a concept developed by Huang and Abraham (1984) [28]. ABFT is normally used in array processing or other instances of multiprocessing to provide uninterrupted and correct results regardless of the failure of individual processing elements. However, it also has a high probability of detecting computational failures in uniprocessor environments. Because recursive least squares parameter identification and the Kalman filter are examples of recursive algorithms which involve many matrix operations, one contribution of this thesis will be the investigation of the benefits and drawbacks of applying ABFT concepts in these algorithms.

Huang and Abraham developed the approach based on the concept of using matrix row and column checksums for detecting and correcting errors from within the confinement of the algorithms software. Background in Anfinson and Luk, 1988 [29], on the method is summarized as follows. Given nxn matrix A, nx(n+1) row checksum matrix A_r is defined as $A_r = [A \;\; Ae]$, where e is nx1 vector $e = [\,1, 1, \ldots, 1\,]^T$. By comparing $\sum_{j=1}^{n} a_{ij}$ and $(Ae)_i$ for i = 1, 2, . . ., n, an error in the ith row of A can be detected. Similarly, given nxn matrix A, (n+1)xn column checksum matrix A_c is defined as $A_c = \begin{bmatrix} A \\ e^T A \end{bmatrix}$. By comparing $\sum_{i=1}^{n} a_{ij}$ and $(e^T A)_j$ for j = 1, 2, . . ., n , an error in the jth column of A can be detected. A full (n+1)x(n+1) checksum matrix A_f is defined as $A_f = \begin{bmatrix} A & Ae \\ e^T A & e^T Ae \end{bmatrix}$. A column checksum matrix A multiplied by a row checksum matrix B is a full checksum matrix AB. Also, when checksum matrices are added and subtracted they result in checksum matrices. For error detection in full checksum matrices, the location of one error in matrix A is found by intersection of inconsistent rows and columns. The single error may be corrected using either the inconsistent row or column.

Anfinson and Luk [29] also explain how the weighted checksum approach from Jou and Abraham (1986) [30] can be used to locate more than one error. By creating d weighted checksum columns or rows, and assigning appropriate weights, a maximum of d errors can be detected and a maximum of (d/2) errors can be corrected. This was proved by Anfinson and Luk [29]. After defining unique nx1 weight vectors $w^{(i)}$, i = 1, . . ., d with unique elements $w_j^{(i)}$, j = 1, . . ., n, the nx(n+d) weighted row checksum matrix is:

$$A_{rw} = [\,A \;\; Aw^{(1)} \;\; Aw^{(2)} \;\; Aw^{(3)} \ldots Aw^{(d)}\,].$$

The (n+d)xn weighted column checksum matrix is:

$$A_{cw} = \begin{bmatrix} A \\ w(1)^T A \\ w(2)^T A \\ w(3)^T A \\ \vdots \\ w(d)^T A \end{bmatrix} .$$

An example given by Anfinson and Luk shows how the d = 2 case can be corrected using the weighted checksum approach. With the weights set as w(1) = e and w(2) = w, and assuming that an error is in element a_{pq}, then letting

$$s_1 = \sum_{i=1}^{n} a_{kq} - (Ae)_q \tag{22}$$

and

$$s_2 = \sum_{k=1}^{n} w_k a_{kq} - (Aw)_q \tag{23}$$

will allow the error to be located. The error can be located in the (p,q) position of A because $s_2/s_1 = w_q$. With the error located, a_{pq} can be corrected as $a_{pq} \leftarrow a_{pq} - s_1$. The selection of appropriate weights is an open area of research.

Using ABFT Techniques With the Kalman Filter

ABFT checksum matrix concepts can be used for a quantitative indication of consistency during each time step of the Kalman filter. The conventional form of the filter can be used to illustrate this point. Before performing the computational substeps of the filter, the consistency of time varying matrices passed to the filter can be checked by comparing the row or column checksum elements with the sum of the corresponding row or column. This check could be postponed until the end of the time steps computations if lost computing time resulting from a passed-in fault is allowable. The substeps of the filter are then performed with each substep including the amount of ABFT matrix operations and checks necessary for the desired level of fault detection. The following example Kalman filter equations have been modified for checksum matrix operations, but they do not provide the most complete fault coverage possible for each substep. However if the checksums of each system matrix are checked at the end of all substeps, a fault in any substep calculation would have a high chance of being detected. In the equations, matrix multiplications should be performed from left to right and vector e has an appropriate length for each particular multiplication.

Computation of Kalman Gain

$$\begin{bmatrix} K_k \\ --- \\ e^T K_k \end{bmatrix} = \begin{bmatrix} P_{k|k-1} \\ ----- \\ e^T P_{k|k-1} \end{bmatrix} H_k^T (H_k P_{k|k-1} H_k^T + R_k)^{-1} \qquad (24)$$

or

$$\begin{bmatrix} K_k \\ --- \\ e^T K_k \end{bmatrix} = \begin{bmatrix} P_{k|k} \\ ---- \\ e^T P_{k|k} \end{bmatrix} H_k^T R_k^{-1} \qquad (25)$$

Conventional Measurement Update of Error Covariance

$$\begin{bmatrix} P_{k|k} \\ --- \\ e^T P_{k|k} \end{bmatrix} = \begin{bmatrix} P_{k|k-1} \\ ----- \\ e^T P_{k|k-1} \end{bmatrix} - \begin{bmatrix} P_{k|k-1} \\ ----- \\ e^T P_{k|k-1} \end{bmatrix} H_k^T (H_k P_{k|k-1} H_k^T + R_k)^{-1} H_k P_{k|k-1} \qquad (26)$$

Innovation Calculation for Measurement Update of State Estimate

$$\begin{bmatrix} E_k \\ --- \\ e^T E_k \end{bmatrix} = \begin{bmatrix} z_k \\ --- \\ e^T z_k \end{bmatrix} - \begin{bmatrix} H_k \\ --- \\ e^T H_k \end{bmatrix} \hat{x}_{k|k-1} \qquad (27)$$

State Estimate Measurement Update

$$\begin{bmatrix} \hat{x}_{k|k} \\ ---- \\ e^T \hat{x}_{k|k} \end{bmatrix} = \begin{bmatrix} \hat{x}_{k|k-1} \\ ----- \\ e^T \hat{x}_{k|k-1} \end{bmatrix} + \begin{bmatrix} K_k \\ --- \\ e^T K_k \end{bmatrix} (E_k) \qquad (28)$$

Time Propagation of State

$$\begin{bmatrix} \hat{x}_{k+1|k} \\ ----- \\ e^T\hat{x}_{k+1|k} \end{bmatrix} = \begin{bmatrix} \Phi_k \\ --- \\ e^T\Phi_k \end{bmatrix} \hat{x}_{k|k} + \begin{bmatrix} \Gamma_k \\ --- \\ e^T\Gamma_k \end{bmatrix} u_k \tag{29}$$

Time Propagation of Error Covariance

$$\begin{bmatrix} P_{k+1|k} \\ ---- \\ e^TP_{k+1|k} \end{bmatrix} = \begin{bmatrix} \Phi_k \\ --- \\ e^T\Phi_k \end{bmatrix} P_{k|k}\Phi_k{}^T + \begin{bmatrix} B_k \\ --- \\ e^TB_k \end{bmatrix} Q_kB_k{}^T \tag{30}$$

As before, by comparing between $\sum_{i=1}^{n} P_{ij}$ and $(e^TP)_j$, $j = 1, 2, \ldots, n$, errors in the jth column of the error covariance matrix can be detected after both the time propagation and measurement update portions of the calculations. Similar comparisons can be made for the results of the calculation of gain, innovations, state measurement update, and state time propagation.

Additions to the Bierman UDU^T Algorithms

To further investigate the possibility of using ABFT concepts in control algorithms, software fault detection capabilities based on ABFT have been added to the Bierman UDU^T estimate error covariance factorization equations and tested using fault simulating software. The Bierman U-D covariance factorization algorithms were chosen because they are widely used in Kalman filter applications for the sequential processing of measurements, particularly when numerical precision is limited. While some users may argue that single precision processing is outdated, many applications still use single precision to obtain faster processing. Faster processing of basic algorithms allows extra CPU time for improvements such as the use of higher order models, more sensors, or failure detection and reconfiguration algorithms.

From Thornton and Bierman [2], the basic idea for obtaining numerically stable time and measurement updates of the estimate error covariance (Equations (17) and (21)) is to propagate <u>factors</u> of the estimate error covariance matrix instead of the matrix itself. The covariance matrix is factored into unit upper triangular matrix U and diagonal matrix D such that covariance matrix P is

$$P = UDU^T. \tag{31}$$

Although several different algorithms exist for the time and measurement update of U-D factors of the estimate error covariance, the following discussion concentrates on the Thornton and Bierman factorizations. Discussion on the specifics of the ABFT additions to the factorization algorithms follows the discussion of the basic algorithms.

To begin, assume that the noise and system matrices vary with time such that tabulated and/or steady state covariances cannot be precomputed and that Kalman gains must be computed in real-time. Once the initial covariance matrix is factored, the factors must be time propagated and then measurement updated at each time step. We begin with the time update algorithm and the software fault detection additions to the algorithm. In the remaining discussion, "k | k-1 or k+1 | k and k | k" subscripts are dropped in favor of "~" and "^" designations consistent with the Thorton and Bierman publication.

UDU^T Time Update of the Error Covariance

The time update algorithm time propagates the U-D factors of the error covariance and is a factorized version of the covariance calculation of Equation (21). The derivation of the algorithm begins by rewriting Equation (21) as Equation (32), with the W and DD matrices in terms of the U-D factors of the error covariance. With appropriate matrices W and DD, the matrix W is factored into upper triangular matrix $\tilde{U}$ and orthogonal row matrix $\tilde{W}$ as in Equation (33) such that $\tilde{W}$, $\tilde{W}^T$ and DD form $\tilde{D}$ as in Equation (34).

$$\tilde{P} = W(DD)W^T \tag{32}$$

$$W = \tilde{U}\tilde{W} \tag{33}$$

$$W(DD)W^T = \tilde{U}(\tilde{W}(DD)\tilde{W}^T)\tilde{U}^T = \tilde{U}(\tilde{D})\tilde{U}^T \tag{34}$$

According to Thornton and Bierman, when ϕ is large or P is ill-conditioned, the computation of Equation (21) can have serious errors, thus giving motivation for the use of their algorithm.

In terms of the $\hat{U}$ and $\hat{D}$ factors prior to update, time updating is accomplished by forming matrices DD and W such that

$$DD = \text{diag}(\hat{D}, Q) \tag{35}$$

and

$$W = \left[\Phi\hat{U} \mid B\right], \tag{36}$$

and then performing the factorization of W with a modified Gram-Schmidt orthogonalization of the rows of W. Keeping in mind the definitions that $\tilde{U}$, $\tilde{U}^T$, $\tilde{D}$, $\tilde{P}$, $\tilde{W}$, and $\tilde{W}^T$ are the matrices after time update, this orthogonalization is accomplished by starting with the last row of W, and progressively "D-orthogonalizing" the remaining rows of W by subtracting out the "D-weighted" component of each row vector which is in the direction of the row currently being used to orthogonalize remaining rows. This method creates both the updated unit upper triangular matrix $\tilde{U}$ (which is the transformation matrix) and matrix $\tilde{W}$ which satisfy Equations (32), (33) and (34).

The following definitions and equations are necessary for understanding the algorithm:

1. Since $\hat{D}$ is an nxn diagonal matrix and Q is an $n_p x n_p$ diagonal matrix, DD is an NxN diagonal matrix where $N = n + n_p$.

2. Equation (21) is rewritten as Equation (32),

$$\tilde{P} = W(DD)W^T. \tag{32}$$

3. The modified Gram-Schmidt orthogonalization of W uses a "D-weighted" inner product rather than an ordinary inner product, and a "D-weighted" inner product is defined as

$$\langle a,b\rangle_D \equiv aDb^T = \sum_{i=1}^{N} D_i a(i)b(i). \tag{37}$$

4. The modified type of orthogonalization which is performed on the rows of W is "D-orthogonalization," and two vectors are defined to be "D-orthogonal" if

$$\langle a,b\rangle_D = 0. \tag{38}$$

5. Because after the orthogonalization of W, the rows $\tilde{W}_i$ of $\tilde{W}$ are "D-orthogonal" to the columns $\tilde{W}_j^T$ of $\tilde{W}^T$ for $i \neq j$, then

$$\langle \tilde{W}_i, \tilde{W}_j\rangle_{DD} = \tilde{D}_j \delta_{ij} \tag{39}$$

where δ is the Kronecker delta.

6. As a result of Equation (39),

$$\tilde{D} = \tilde{W}(DD)\tilde{W}^T \tag{40}$$

where

$$\tilde{D} = \mathrm{diag}(\tilde{D}_1, \ldots, \tilde{D}_n) \quad \text{and} \quad \tilde{W} = \begin{bmatrix} \tilde{W}_1 \\ \vdots \\ \tilde{W}_n \end{bmatrix}.$$

From Thorton and Bierman [2], the summary of the U-D Time Update Algorithm is repeated here to facilitate the discussion of the software fault detection additions to the algorithm.

U-D Time Update Algorithm by Thorton and Bierman

For $j = n, n\text{-}1, \ldots, 2$ cycle through Equations (a) through (c).

$$\tilde{D}_j = \langle W_j^{(n-j)}, W_j^{(n-j)}\rangle_{DD} \tag{a}$$

$$\left.\begin{aligned} \tilde{U}(i,j) &= \langle W_i^{(n-j)}, W_j^{(n-j)}\rangle_{DD} / \tilde{D}_j \\ W_i^{(n-j+1)} &= W_i^{(n-j)} - \tilde{U}(i,j) W_j^{(n-j)} \end{aligned}\right\} i = 1, \ldots, j\text{-}1. \tag{b) and (c}$$

$$\tilde{D}_1 = \langle W_1^{(n-1)}, W_1^{(n-1)}\rangle_{DD} \tag{d}$$

Software Fault Detection Additions to the Time Update Algorithm

The following items describe how the UDU^T time update algorithm was modified to include the ABFT additions.

1. First, current time step system matrices and U and D factors from the measurement update of the error covariance must be input to the algorithm. Because of the ABFT modifications, the following checksum matrices were passed into the algorithm instead of matrices without checksums. For this effort, the checksums for these matrices are assumed to have been verified in the previous time steps measurement update algorithm, so no checksums were tested at the time the matrices were passed into the algorithm.

$$\begin{bmatrix} \Phi \\ \hline e^T\Phi \end{bmatrix}, \left[\begin{array}{c|c} \hat{U} & \hat{U}e \end{array}\right], \begin{bmatrix} B \\ \hline e^TB \end{bmatrix}, \left[\begin{array}{c|c} Q & 0 \\ - & - \\ 0 & e^TQe \end{array}\right], \left[\begin{array}{c|c} \hat{D} & 0 \\ - & - \\ 0 & e^T\hat{D}e \end{array}\right].$$

2. Form matrix W by calculating $\Phi\hat{U}$ and a checksum verification as follows.

a. $$\left[\begin{array}{c|c} \Phi\hat{U} & \text{Not Calc.} \\ - & - \\ e^T\Phi\hat{U} & e^T\Phi\hat{U}e \end{array}\right] = \begin{bmatrix} \Phi \\ \hline e^T\Phi \end{bmatrix} \left[\begin{array}{c|c} \hat{U} & \hat{U}e \end{array}\right].$$

b. Check $\sum_i \sum_j (\Phi\hat{U})(i,j)$ versus $e^T\Phi\hat{U}e$.

c. $$\begin{bmatrix} W \\ \hline e^TW \end{bmatrix} = \left[\begin{array}{c|c} \Phi\hat{U} & B \\ - & - \\ e^T\Phi\hat{U} & e^TB \end{array}\right]$$

3. Form $$DD = \left[\begin{array}{c|c} \begin{matrix} \hat{D} & 0 \\ 0 & Q \end{matrix} & 0 \\ - & - \\ 0 & e^TQe + e^T\hat{D}e \end{array}\right].$$

4. Factor W using the modified Gram-Schmidt orthogonalization algorithm into $\tilde{U}\tilde{W}$. Every time a vector component $\tilde{U}(i,j)W_j^{(n-j)}$ is subtracted from a row $W_i^{(n-j)}$ of W, also subtract the same amount from the column checksum row e^TW such that the column checksums are always current.

5. Reset the row checksum's for U to zero in

$$\left[\begin{array}{c|c} \tilde{U} & \tilde{U}e \end{array}\right],$$

and as each W rows $\tilde{U}$ basis transformation coefficients $\tilde{U}(i,j)$ are created, add the element to the row checksum column of $\tilde{U}$ as soon as the coefficient is calculated.

6. Reset the checksum for D and as each element $\tilde{D}_j$ is calculated, update the diagonal elements checksum $e^T\tilde{D}e$, where $e^T\tilde{D}e$ is part of the matrix

$$\left[\begin{array}{c|c} \tilde{D} & 0 \\ \hline 0 & e^T\tilde{D}e \end{array}\right].$$

7. Finally, verify that the row or column checksums are consistent with the row or column elements of each of following matrices. This is accomplished by summing row or column elements and making a direct comparison with the row or column checksum.

a. Check column checksums $(e^T\tilde{W})_j$ versus $\sum_i \tilde{W}(i,j)$.

b. Check $(\tilde{U}e)_i$ versus $\sum_j \tilde{U}(i,j)$.

c. Check $e^TQe + e^T\tilde{D}e$ versus $\sum_j DD(j)$.

d. Check $(e^T\tilde{D}e)$ versus $\sum_j \tilde{D}(j)$.

Table 1 summarizes the number of operations required before the modification and the number of operations added by the modification for a comparison of the overhead required by the ABFT additions. Table 2 contains calculations of the percentage increase in computational overhead which is caused by the ABFT addidtions for several different system order sizes. The values in Table 2 illustrate how the percentage overhead resulting from the ABFT additions decreases significantly as the number of state variables increase.

UDU^T Measurement Update of the Error Covariance

The measurement update algorithm concerns the calculations involved in updating the U-D factors of the estimate error covariance given the *a priori* state estimate

$$x_{k|k-1} = \tilde{x}, \tag{41}$$

estimate error covariance

$$P_{k|k-1} = \tilde{P} = \tilde{U}\tilde{D}\tilde{U}^T \tag{42}$$

and a scalar measurement with zero mean normally distributed noise with covariance R. It performs a factorized version of the covariance calculation of equation (17), and results in the error covariance $\hat{P}$ for the minimum variance estimate $\hat{x} = x_{k|k}$. As a byproduct, the algorithm also generates an n-state normalized Kalman gain vector. Similar to the way the time update algorithm is derived by first rewriting Equation (21) in terms of $\hat{U}$ and $\hat{D}$ factors of $\hat{P}$, the derivation of Bierman's measurement update algorithm starts by rewriting Equation (17) in terms of the factors $\tilde{U}$ and $\tilde{D}$ of $\tilde{P}$. This rewritten equation has a special structure which can be exploited to form the $\hat{U}$ and $\hat{D}$ factors of $\hat{P} = \hat{U}\hat{D}\hat{U}^T$. Using nomenclature from the Thornton and Bierman publication, it can be verified that Equation (17) can be rewritten as

$$\hat{P} = \hat{U}\hat{D}\hat{U}^T = \tilde{U}[\tilde{D} - (g\,g^T/\alpha)]\tilde{U}^T, \tag{43}$$

TABLE 1

OPERATION COUNTS FOR THE STANDARD TIME UPDATE ALGORITHM AND ABFT ADDITIONS

Algorithm	Adds	Multiplies	Divides	Logic
MWGS U-D [2]	$1.5n^3 + 0.5n^2 + n^2 n_p + (0.5n^2 - 0.5n)$*	$1.5n^3 + 2n^2 - 0.5n + (n^2 + n)n_p + (n^2 - n)$*	n-1	0
ABFT ADDITIONS TO MWGS U -D	$0.5n^3 + 3n^2 + 3.5n + n_p + 1 + n_p(0.5n^2 + 0.5n)$	$0.5n^2 + 0.5n$	0	5

*Variance matrix formed.

TABLE 2

COMPUTATIONAL COSTS FOR THE ABFT ADDITIONS IN TERMS OF A PERCENTAGE OF ORIGINAL OPERATIONS FOR THE TIME UPDATE ALGORITHM

State Variables (n)	Adds	Multiplies	Divides	Logic	Overall
10	54%	3%	0%	N/A	27%
50	37%	0.7%	0%	N/A	19%
100	35%	0.3%	0%	N/A	18%

where

$$f^T = H\tilde{U}, \tag{44}$$

$$g = \tilde{D}f \qquad (g_i = \tilde{D}_i f_i,\ i = 1, \ldots, n), \tag{45}$$

$$\alpha = R + \sum_{i=1}^{n} g_i f_i. \tag{46}$$

From Thornton and Bierman, the bracketed term in Equation (43) is positive semi-definite and can be factored as $\bar{U}\bar{D}\bar{U}^T$. Because the product of unit upper triangular matrices is unit upper triangular,

$$\hat{U} = \tilde{U}\bar{U}. \tag{47}$$

and

$$\hat{D} = \bar{D}. \tag{48}$$

As previously mentioned, it is the special structure of the bracketed term in Equation (43) which is exploited to create $\bar{U}$ and $\bar{D}$. Given that

$$\bar{U}\bar{D}\bar{U}^T = \tilde{D} - (1/\alpha)gg^T, \tag{49}$$

it can be rewritten as

$$\sum_{i=1}^{n} \bar{D}_i \bar{U}^{(i)} \bar{U}^{(i)T} = \sum_{i=1}^{n} \tilde{D}_i e_i e_i^T - (1/\alpha)gg^T, \tag{50}$$

where

$$\bar{U} = (\bar{U}^{(1)}, \ldots, \bar{U}^{(n)}), \tag{51}$$

$$\bar{D} = \mathrm{diag}(\bar{D}_1, \ldots, \bar{D}_n), \tag{52}$$

$$\bar{U}^{(i)} = (\bar{U}_1^{(i)}, \ldots, \bar{U}_{i-1}^{(i)}, 1, 0, \ldots, 0)^T, \tag{53}$$

and e_i is a null vector with the exception of a unit value for the i-th element. From this point the derivation shows that the $\bar{U}_i$ and $\bar{D}_i$ components can be determined in a backward recursive fashion for $i = n, n\text{-}1, \ldots, 1$ as depicted in the following equation,

$$\sum_{i=1}^{n} \tilde{D}_i e_i e_i^T - c_n v^{(n)} v^{(n)T} = \bar{D}_n \bar{U}_n \bar{U}_n^T + \bar{D}_{n-1} \bar{U}_{n-1} \bar{U}_{n-1}^T$$
$$+ \left(\sum_{i=1}^{n-2} \tilde{D}_i e_i e_i^T - c_{n-2} v^{(n-2)} v^{(n-2)T}\right) \tag{54}$$

where

$$\bar{D}_n = \tilde{D}_n(\alpha_{n-1}/\alpha_n) \tag{55}$$

$$\alpha_i = R + \sum_{k=1}^{i} g_k f_k, \quad i = 1, \ldots, n \tag{56}$$

$$c_{n-1} = 1/\alpha_{n-1} \tag{57}$$

$$v^{n-1} = (g_1, \ldots, g_{n-1}, 0)^T \tag{58}$$

$$\bar{U}^{(n)} = -(f_n/\alpha_{n-1}) g_i, \quad i = 1, \ldots, n-1, \tag{59}$$

but the derivations to get to the final algorithm are lengthy and the interested reader is refered to Thorton and Bierman. From Thorton and Bierman [2], the summary of the U-D measurement update algorithm is also repeated to facilitate the discussion of the software fault detection additions to the algorithm.

U-D Measurement Update Algorithm By Bierman

$$f^T := H\tilde{U} \tag{a}$$

$$g := \tilde{D}f \quad (g_i = \tilde{D}_i f_i, \; i = 1, \ldots, n) \tag{b}$$

For j = 1, . . . , n cycle through Equations (c) through (h):

$\alpha_j := \alpha_{j-1} + f_j g_j \qquad \alpha_0 = R$
(j–dimensional partial–state innovations variance) (c)

$\hat{D}_j := (\alpha_{j-1}/\alpha_j) D_j \quad (\hat{D}_j := \tilde{D}_j \;\; if \;\; \alpha_j = 0)$
(diagonal element fractional update) (d)

$$v_j := g_j \tag{e}$$

$$\lambda := -f_j/\alpha_{j-1} \quad (\lambda := 0 \; if \; \alpha_{j-1} = 0) \tag{f}^*$$

(For j = 1, (f) not included.)*

For $i = 1, \ldots, j\text{-}1$ compute recursively (g) and (h):

$\hat{U}_{ij} := \tilde{U}_{ij} + v_i\lambda$
(update of column j of the U matrix factor) (g)*

$v_i := v_i + \tilde{U}_{ij} v_j$
(j–dimensional partial state normalized gain) (h)*

(For j = 1, (g) and (h) not included.)*

Software Fault Detection Additions to the Measurement Update Algorithm

The following items describe how the UDU^T measurement update algorithm was modified to include the ABFT additions.

1. First, current time step system matrices and U and D factors from the time update of the error covariance must be input to the algorithm. Because of the ABFT modifications, the following checksum matrices were passed into the algorithm instead of matrices without checksums. For this effort, the checksums for these matrices are assumed to have been verified in the previous time steps time update algorithm, so no checksums were tested at the time the matrices were passed into the algorithm.

$$\left[\begin{array}{c|c} \tilde{U} & \tilde{U}e \end{array}\right], \left[\begin{array}{c} H \\ \hline e^T H \end{array}\right] \left[\begin{array}{c|c} \tilde{D} & 0 \\ - & - \\ 0 & e^T \tilde{D} e \end{array}\right].$$

2. In the first step of the algorithm, Equation (a), the vector f^T is formed. The ABFT addition to this step consisted of the multiplication of the column checksum matrix for H (even though H is a vector in this case) by the row checksum matrix for $\tilde{U}$ as follows,

$$\left[\begin{array}{c|c} f^T & \text{Not Calc.} \\ - & - \\ \text{Not Calc.} & e^T f^T e \end{array}\right] = \left[\begin{array}{c} H \\ \hline e^T H \end{array}\right] \left[\begin{array}{c|c} \tilde{U} & \tilde{U}e \end{array}\right]. \qquad (60)$$

Verification of the checksum was delayed until the end of the algorithm.

3. In the second step of the algorithm, Equation (b), the vector g is formed. The ABFT addition to this step consisted of the formation and update of a checksum for g as each element of g was calculated. Verification of the checksum was performed near the end of the algorithm.

4. The checksum for $\tilde{D}$ was checked following Equation (b) by comparing

$$\sum_{i} \tilde{D}_j \text{ versus } e^T\tilde{D}e\,. \qquad (61)$$

5. While cycling through Equations (c) through (h) for j = 1, . . . , n, a checksum for α was created and updated by adding each α_j as they were calculated. Verification with a duplicate copy of the current alpha is performed at the end of Equation (h) at each value of j, and a verification of the checksum for α is performed at the end of the algorithm.

6. While cycling through Equations (c) through (h) for j = 1, . . . , n, at each fractional update of D in Equation (d), the previous value of D_j was subtracted and the new value of D_j was added to the checksum for D as follows,

$$e^T\tilde{D}e = e^T\tilde{D}e - \tilde{D}_j + (\alpha_{j-1}/\alpha_j)\tilde{D}_j \qquad (62)$$

Verification of the checksum was performed at the end of the algorithm.

7. Although a software fault for λ was not simulated, the algorithm changes included keeping duplicate copies of λ in separate memory locations, and then performing a comparison when the current value would no longer need to be used at the end of Equation (h) for each value of j.

8. Similarly, although a software fault for v_j was not simulated, the algorithm changes included keeping duplicate copies of v_j in separate memory locations, and then performing a comparison when the current value would no longer need to be used. Also, a checksum for v was created and updated as each element of v was changed in Equation (h).

9. As each $v_i\lambda$ increment was added to the $\hat{U}(i,j)$ elements in equation (f), the row checksum column $\hat{U}e$ for $\hat{U}$ was updated,

$$\left[\hat{U} \;\middle|\; \hat{U}e\right].$$

Verification of the checksum was performed at the end of the algorithm.

10. Finally, the algorithm verifies that the row or column checksums are consistent with the row or column elements of each of following matrices. This is accomplished by summing row or column elements and making a direct comparison with the row or column checksum.

a. Check e^Tg versus $\sum_i g_i$.

b. Check $(\hat{U}e)_i$ versus $\sum_j \hat{U}(i,j)$.

c. Check e^Tf^T versus $\sum_i f_i^T$.

d. Check $(e^T\hat{D}e)$ versus $\sum_j \hat{D}(j)$.

e. Check α checksum.

f. Check e^Tv versus $\sum_i v_i$.

Table 3 summarizes the number of operations required before the modification and the number of operations added by the modification for a comparison of the overhead required by the ABFT additions to the measurement update algorithm. Table 4 contains calculations of the percentage increase in computational overhead which is caused by the ABFT addidtions for several different system order sizes. Again, the values in Table 4 illustrate how the percentage overhead resulting from the ABFT additions decreases significantly as the number of state variables increase.

TABLE 3

OPERATION COUNTS FOR THE STANDARD MEASUREMENT UPDATE ALGORITHM AND ABFT ADDITIONS

Algorithm	Adds	Multiplies	Divides	Logic
U-D Factorization [2]	$(1.5n^2 + 1.5n)m + (0.5n^2 - 0.5n)^*$	$(1.5n^2 + 5.5n)m + (n^2 - n)^*$	n-1	0
ABFT additions to U-D Factorization	$(1.5n^2 + 11.5n)m$	nm	0	(3n +7)m

*Variance matrix formed.

TABLE 4

COMPUTATIONAL COSTS FOR THE ABFT ADDITIONS IN TERMS OF A PERCENTAGE OF ORIGINAL OPERATIONS FOR THE MEASUREMENT UPDATE ALGORITHM

State Variables (n)	Adds	Multiplies	Divides	Logic	Overall
10	161%	5%	0%	N/A	84%
50	113%	1%	0%	N/A	58%
100	107%	0.6%	0%	N/A	54%

CHAPTER IV

SIMULATIONS

Simulation Environment

To test the software fault detection methods, an RTOS software implementation of the UDU^T Kalman filter (Thornton and Bierman (1977)) error covariance update algorithms with the ABFT modifications was developed. Rather than developing Matlab code which simulated a real-time multitasking operating system, the filtering algorithms with ABFT modifications were coded and run entirely in the VxWorks real-time operating system environment. This software implementation is included in the Appendix. If the simulations had been performed in Matlab, the introduction of faults into the calculations and the fault locations would have been pre-determined. However, by performing the simulation directly in the VxWorks environment, the faults were allowed to happen in a non-exact periodic fashion with all the timing irregularities of the pre-emptive priority based and multi-tasking operating system. Faults were simulated by creating a rogue task which periodically corrupted information in shared memory being used by the covariance update tasks. The rogue routine was created so that the user could control the periodicity, value and location of the fault. Using the algorithm modifications previously discussed, the error covariance factorization algorithms were able to independently and immediately detect simulated faults as they occurred. The simulated faults are representative of errors which might result from external environments, corrupt communication with external processors, software faults, and/or memory and logic chip MTBF failures. The software is coded in "C".

The OSU College of Engineering Interdisciplinary Real Time Distributed Systems Laboratory was used to simulate the proposed techniques. The simulation used a VME-based system with a Heurikon Motorola 68040 microprocessor-based single board computer utilizing the VxWorks real time operating system kernel, with cross development performed on a Sun Sparc workstation. Matlab compatible data files were transferred via file transfer protocol (FTP) from the Heurikon computer to a engineering college RS6000 computer. On the RS6000, a matlab script file with an embedded UNIX C-shell "sleep" command was made to periodically form the covariance matrix elements from the U-D factors stored in the data files, display plots of the Kalman gains and covariances for 100 time step frames, write the plots to a uniquely named postscript file, and then remove the used data files before arrival of the next frames data. The displays were remotely plotted on the Sun Sparc workstation being used for cross development.

Results

Figures 4 through 29 are representative plots of the effects of the simulated faults on the error covariance and Kalman gain calculations for a second-order system. Seven different cases are represented. Six cases are shown with time frames from 0 to 100 time steps and 100 to 200 time steps, and one case in Figures 12 through 13 is shown for the time frame from 0 to 100 time steps. The following second-order system parameters and noise covariances were used in the simulations.

$$\Phi = \begin{bmatrix} 1.0 & 0.02 \\ 0 & 1.00 \end{bmatrix}$$

$$R = 0.1$$

$$Q = 0.01$$

$$H = [1.0\ 0.0]$$

and

$$B = G = \begin{bmatrix} 0.0 \\ 1.0 \end{bmatrix} \tag{63}$$

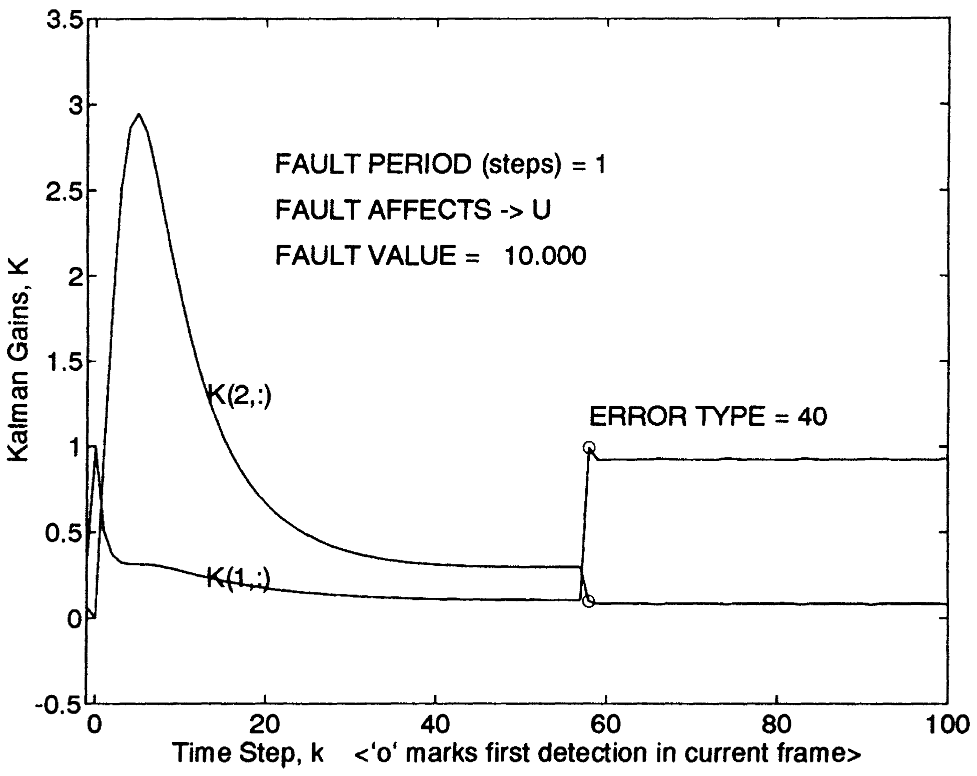


Figure 4. Software Fault Effect on Single Measurement UDU^T
Kalman Gains, Case 1, Steps 0 to 100

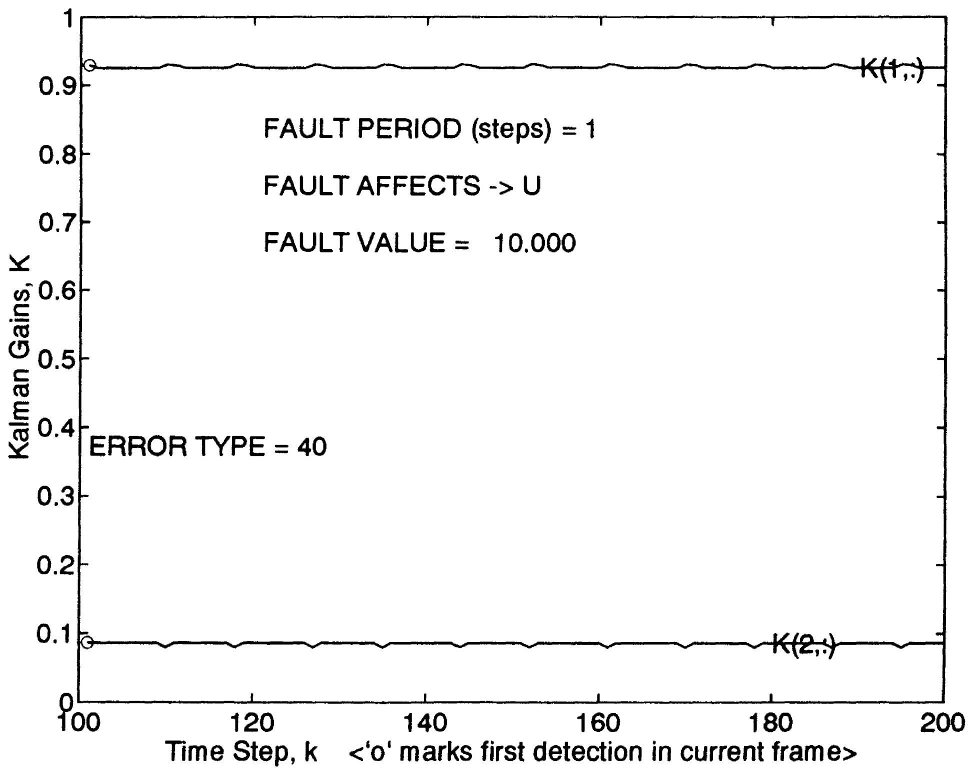


Figure 5. Software Fault Effect on Single Measurement UDU^T
Kalman Gains, Case 1, Steps 101 to 200

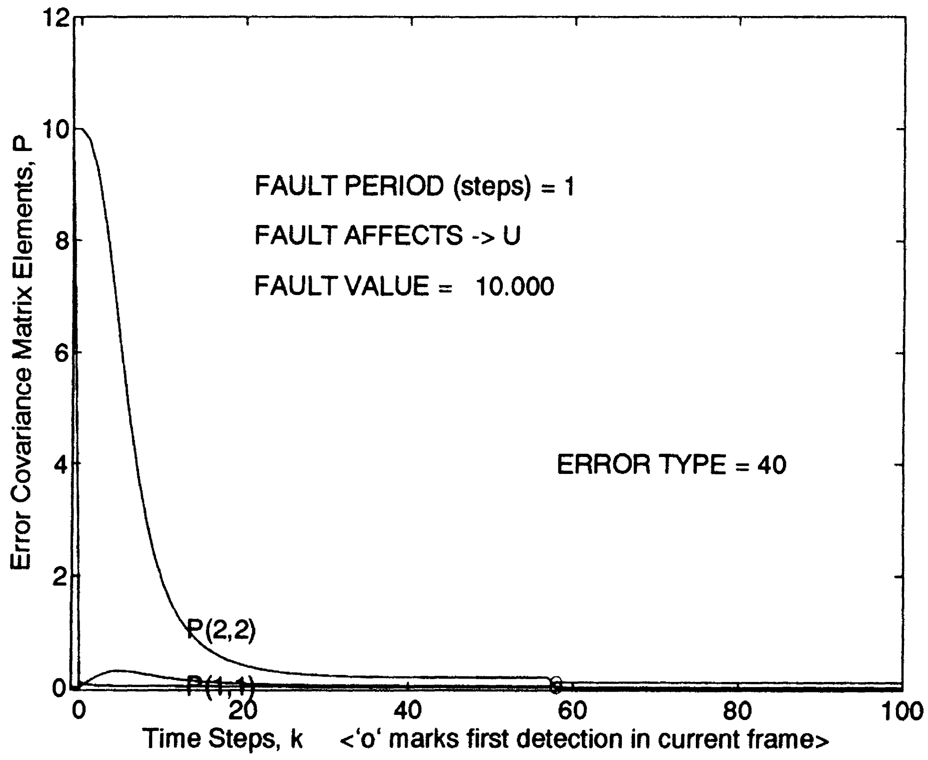


Figure 6. Software Fault Effect on Single Measurement UDU^T Error Covariance, Case 1, Steps 0 to 100

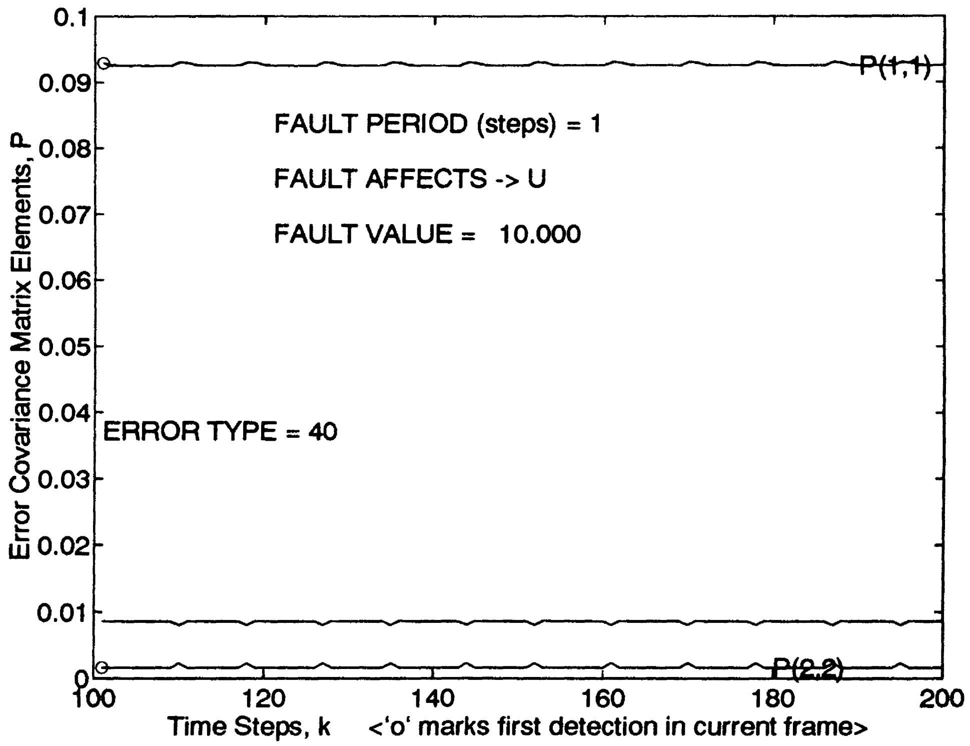


Figure 7. Software Fault Effect on Single Measurement UDU^T Error Covariance, Case 1, 101 to 200

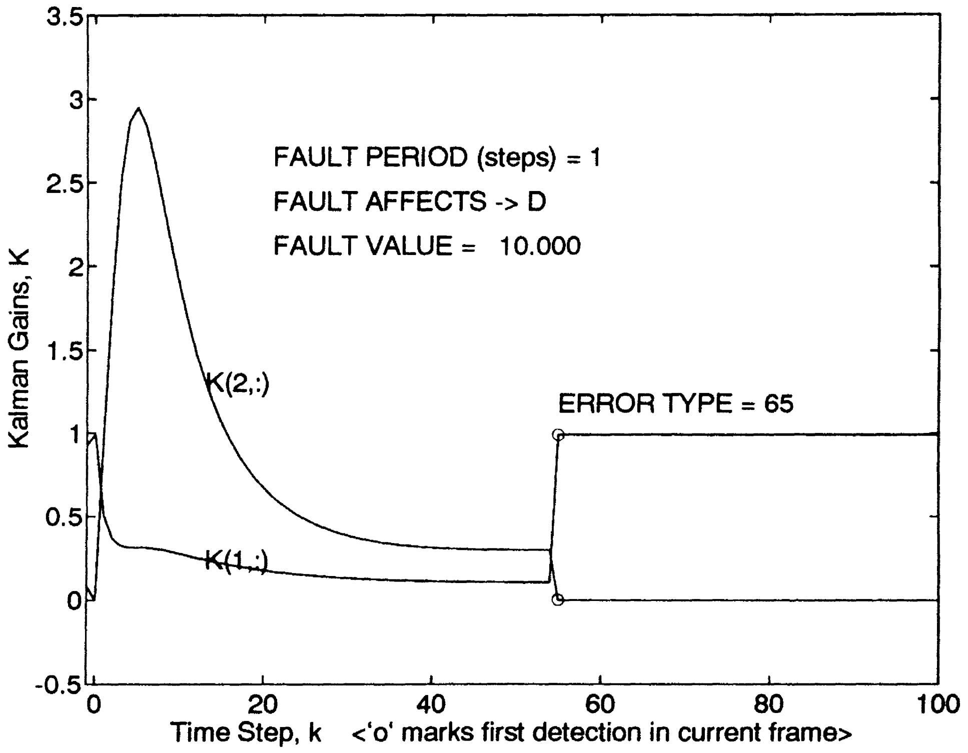


Figure 8. Software Fault Effect on Single Measurement UDU^T Kalman Gains, Case 2, Steps 0 to 100

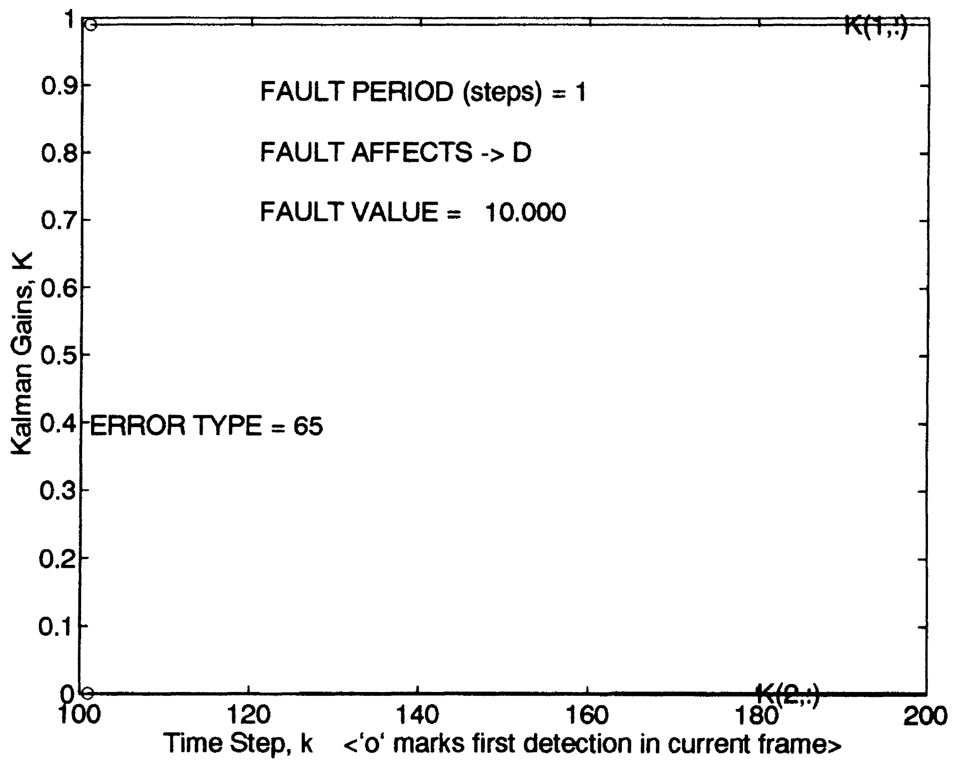


Figure 9. Software Fault Effect on Single Measurement UDU^T Kalman Gains, Case 2, Steps 101 to 200

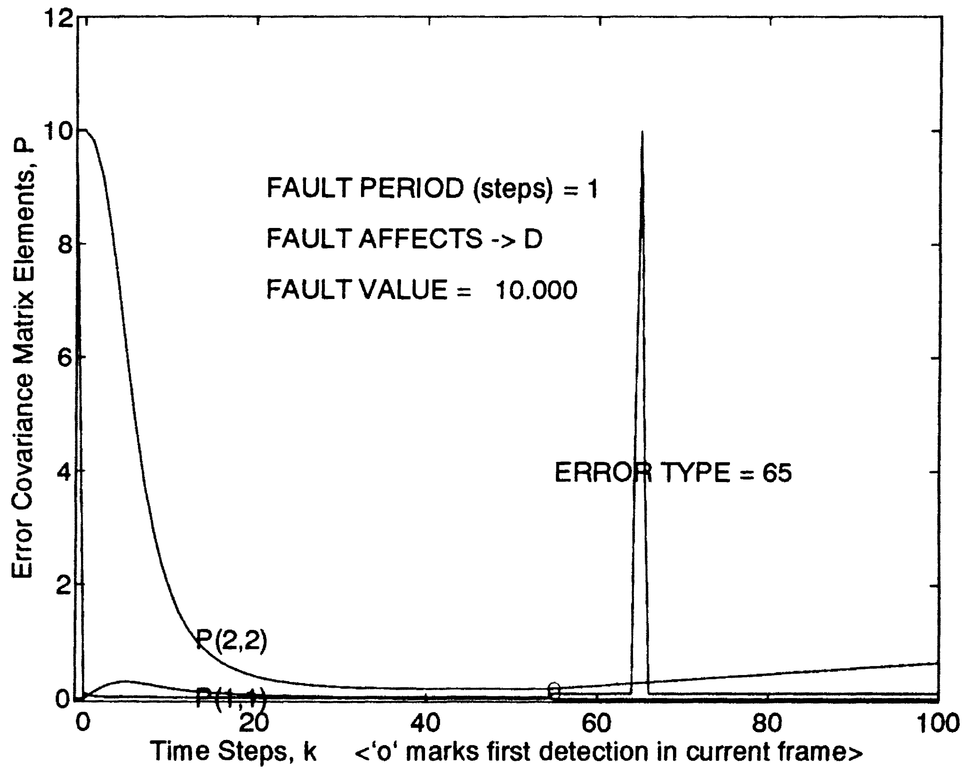


Figure 10. Software Fault Effect on Single Measurement UDU^T Error Covariance, Case 2, Steps 0 to 100

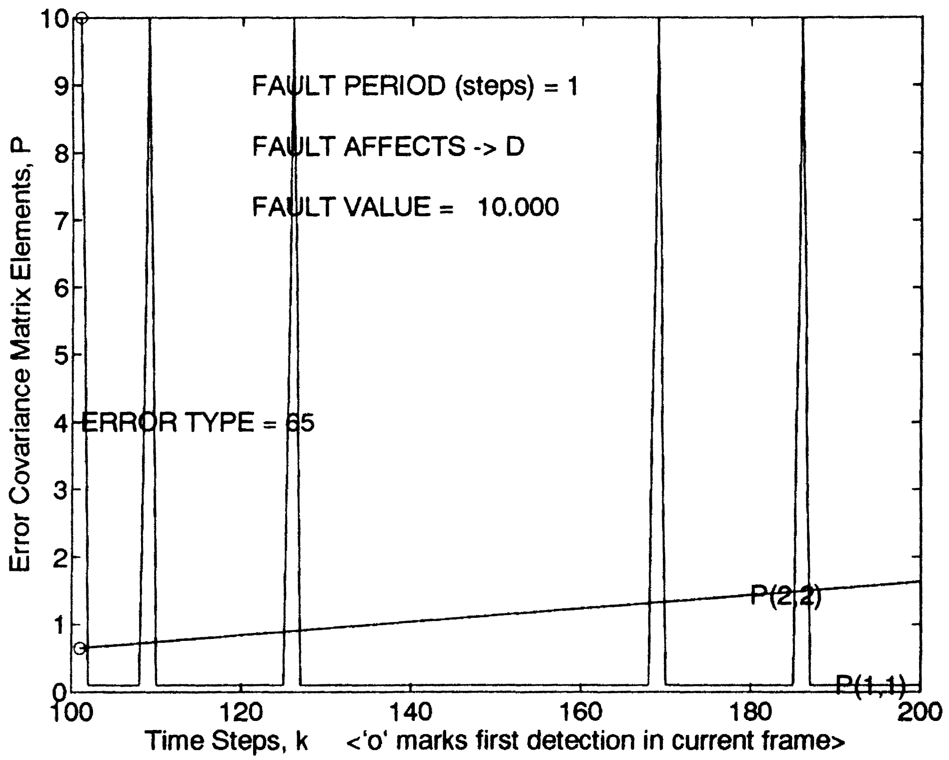


Figure 11. Software Fault Effect on Single Measurement UDU^T Error Covariance, Case 2, Steps 101 to 200

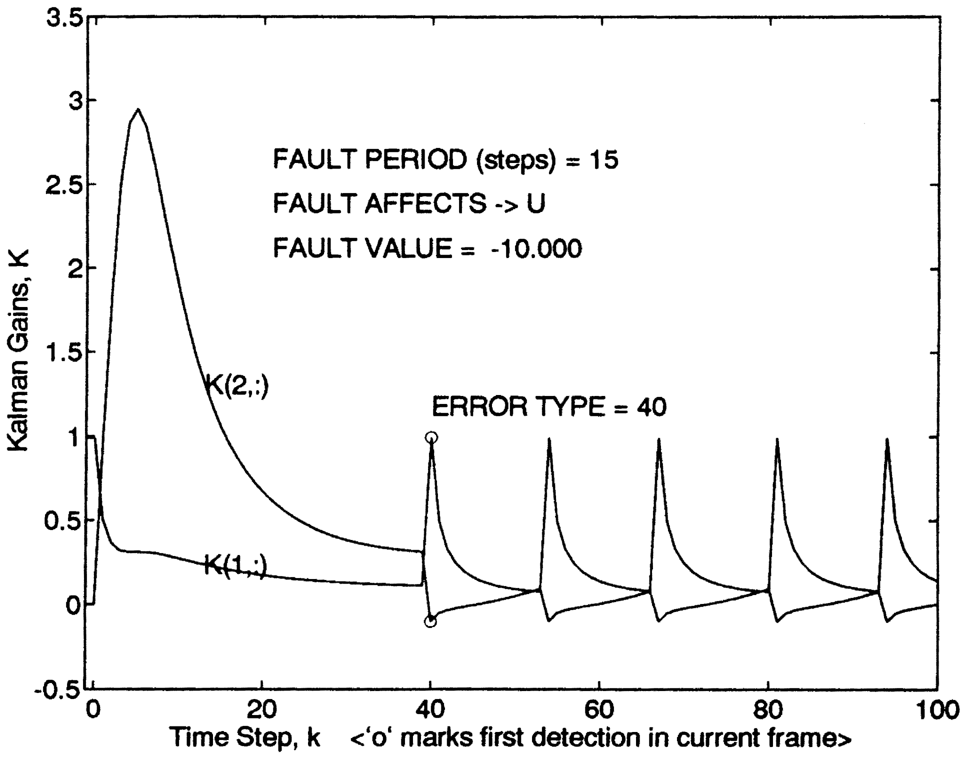


Figure 12. Software Fault Effect on Single Measurement UDU^T Kalman Gains, Case 3, Steps 0 to 100

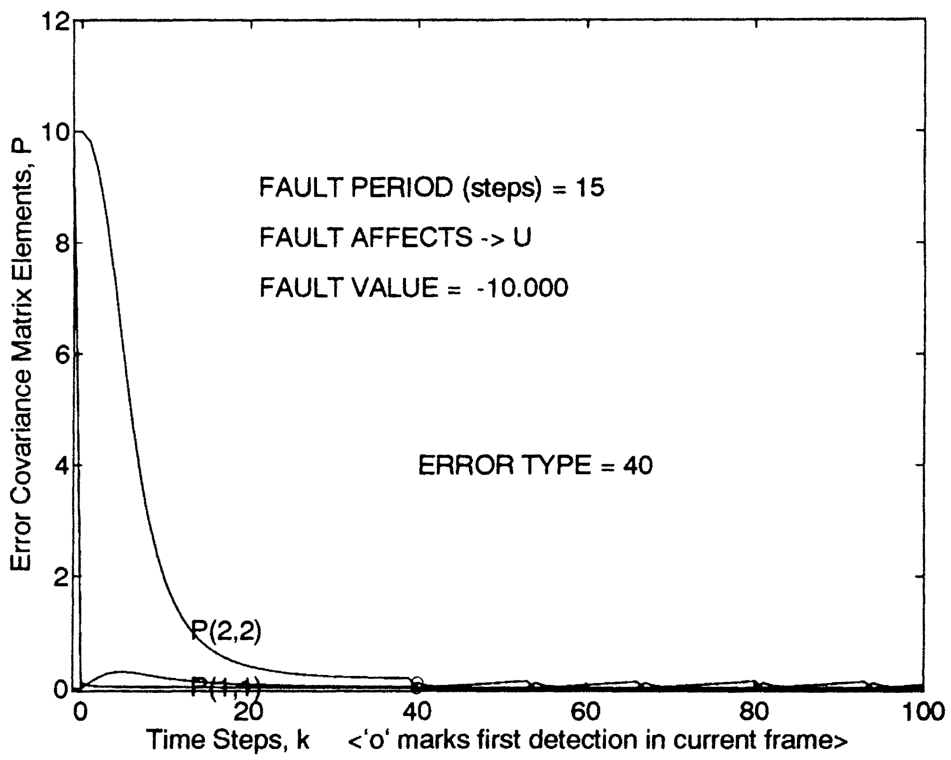


Figure 13. Software Fault Effect on Single Measurement UDU^T Error Covariance, Case 3, Steps 0 to 100

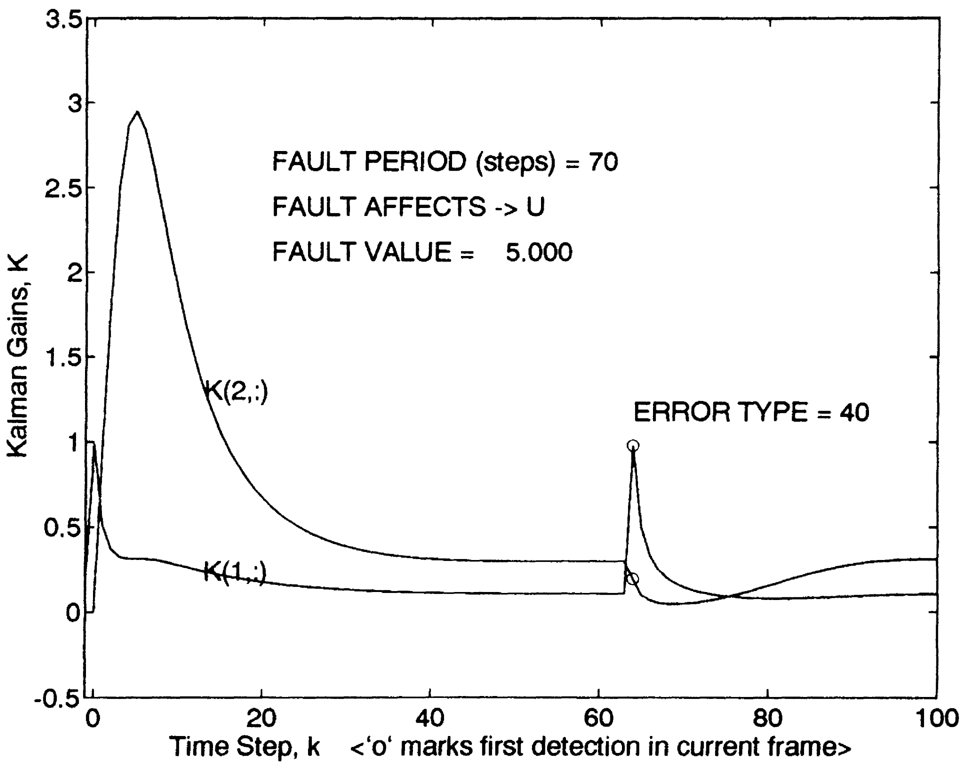


Figure 14. Software Fault Effect on Single Measurement UDU^T Kalman Gains, Case 4, Steps 0 to 100

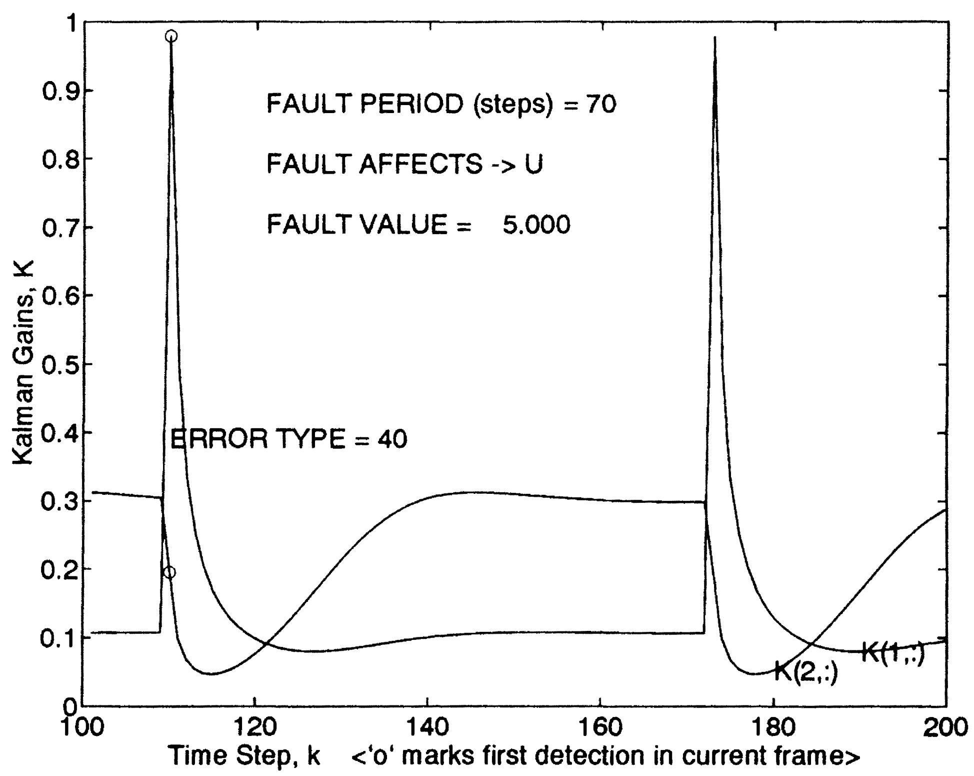


Figure 15. Software Fault Effect on Single Measurement UDU^T Kalman Gains, Case 4, Steps 101 to 200

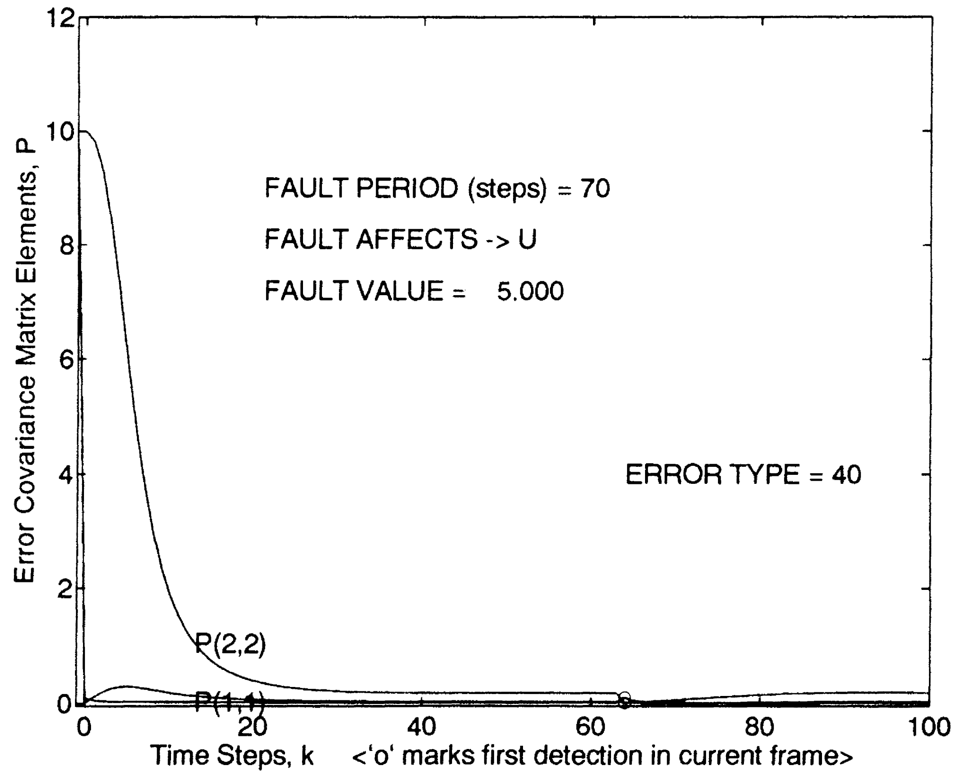


Figure 16. Software Fault Effect on Single Measurement UDU^T Error Covariance, Case 4, Steps 0 to 100

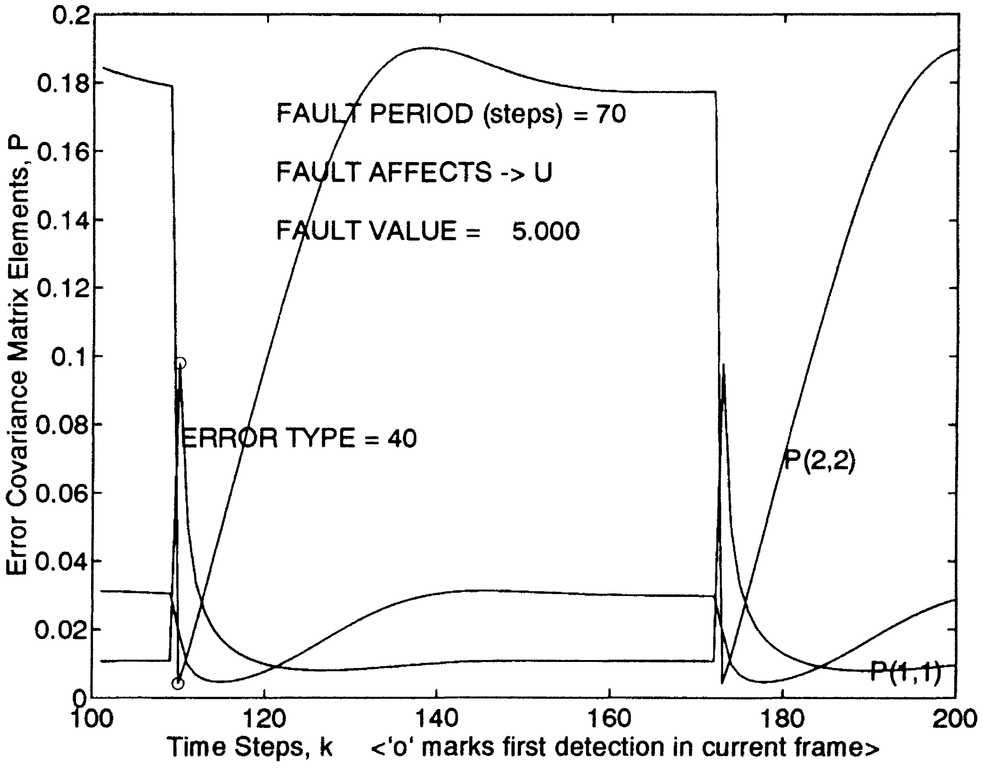


Figure 17. Software Fault Effect on Single Measurement UDU^T Error Covariance, Case 4, Steps 101 to 200

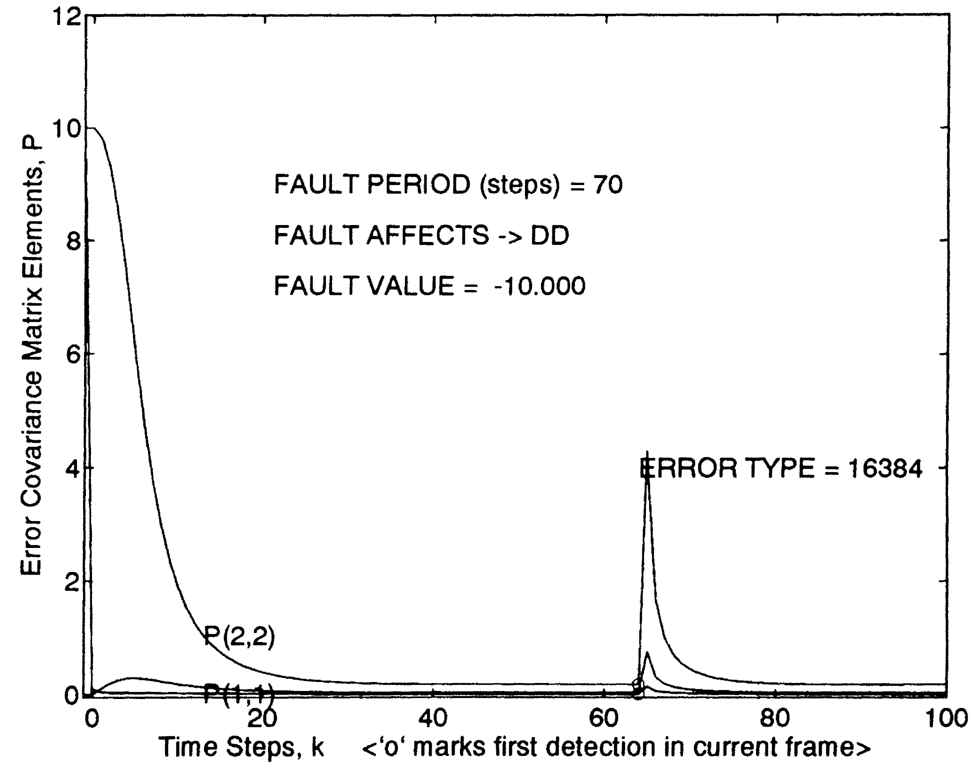


Figure 18. Software Fault Effect on Single Measurement UDU^T Error Covariance, Case 5, Steps 0 to 100

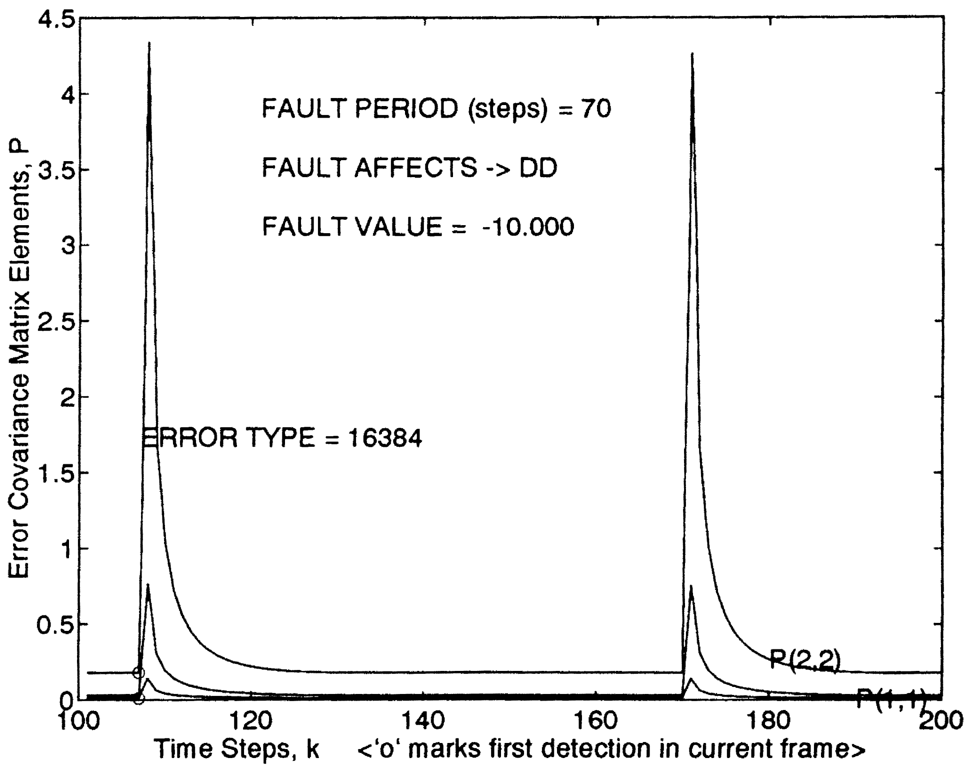


Figure 19. Software Fault Effect on Single Measurement UDU^T Error Covariance, Case 5, Steps 101 to 200

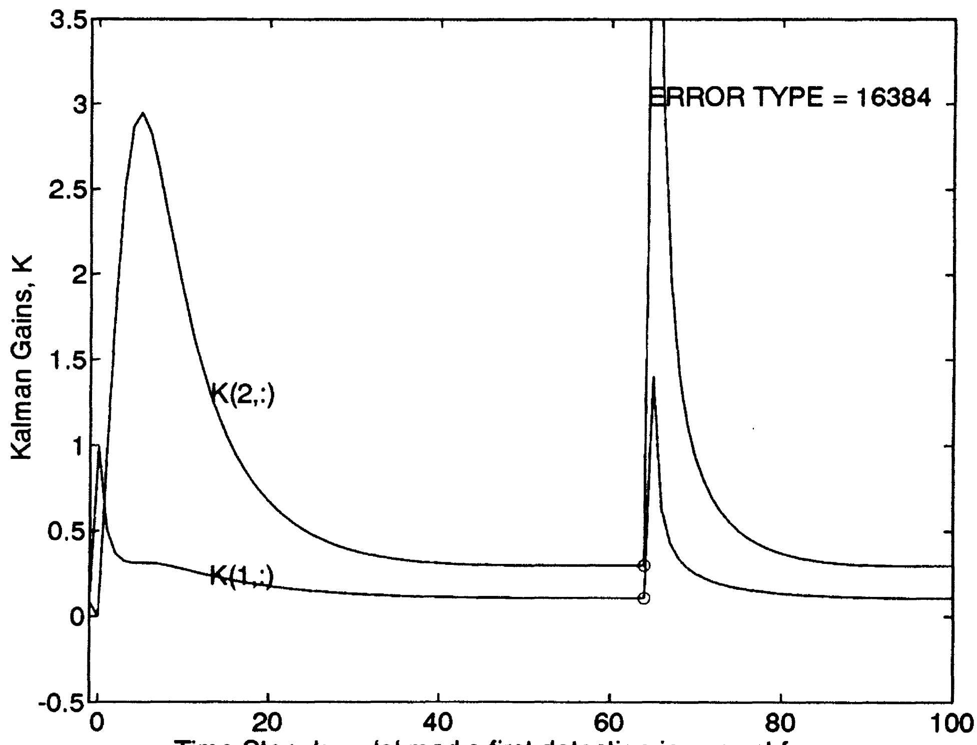


Figure 20. Software Fault Effect on Single Measurement UDU^T Kalman Gains, Case 5, Steps 0 to 100

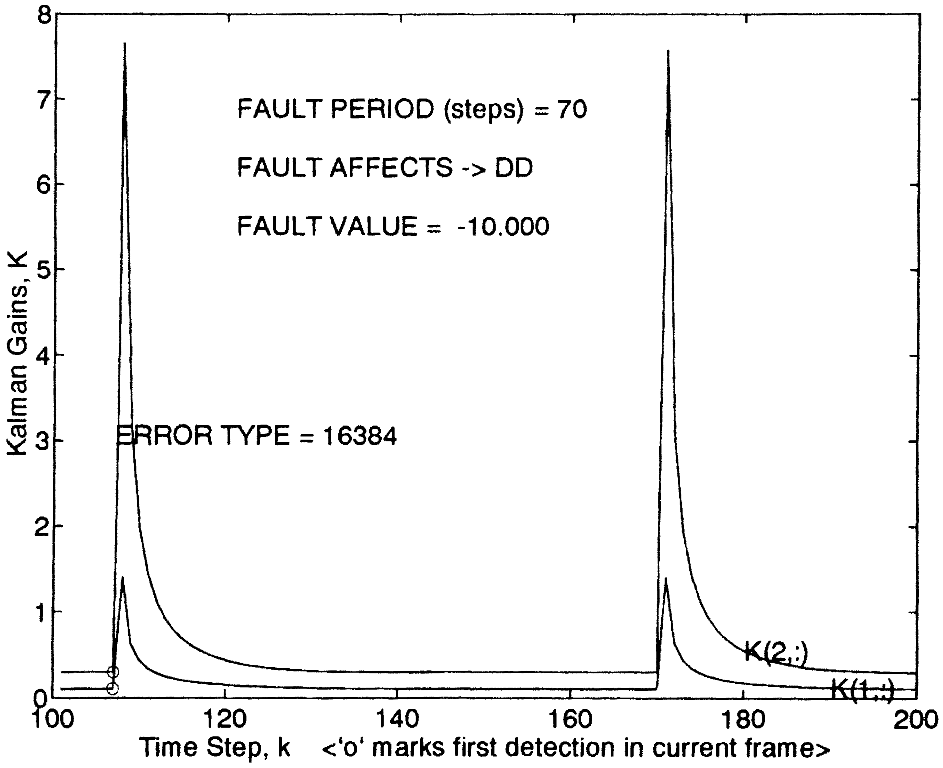


Figure 21. Software Fault Effect on Single Measurement UDU^T Kalman Gains, Case 5, Steps 101 to 200

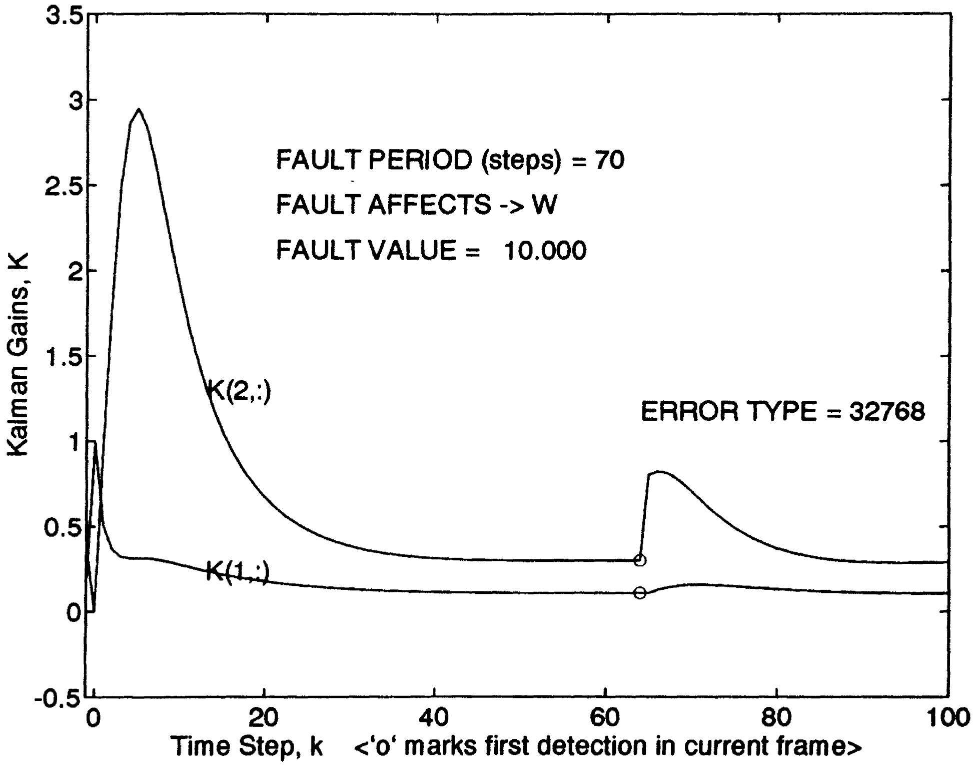


Figure 22. Software Fault Effect on Single Measurement UDU^T Kalman Gains, Case 6, Steps 0 to 100

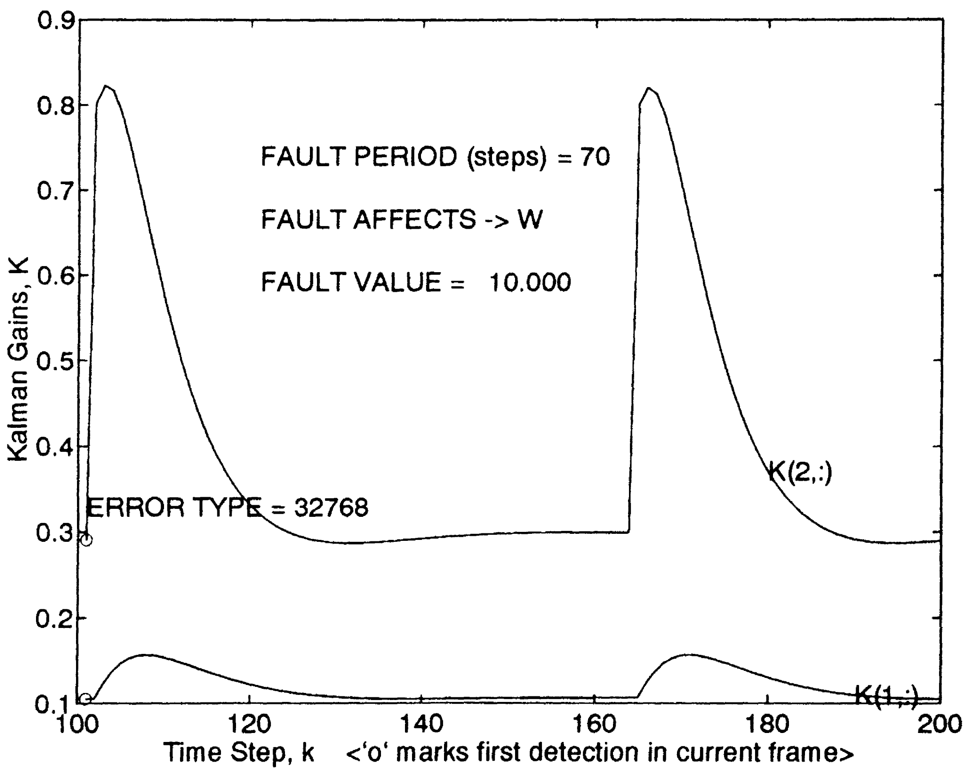


Figure 23. Software Fault Effect on Single Measurement UDU^T Kalman Gains, Case 6, Steps 101 to 200

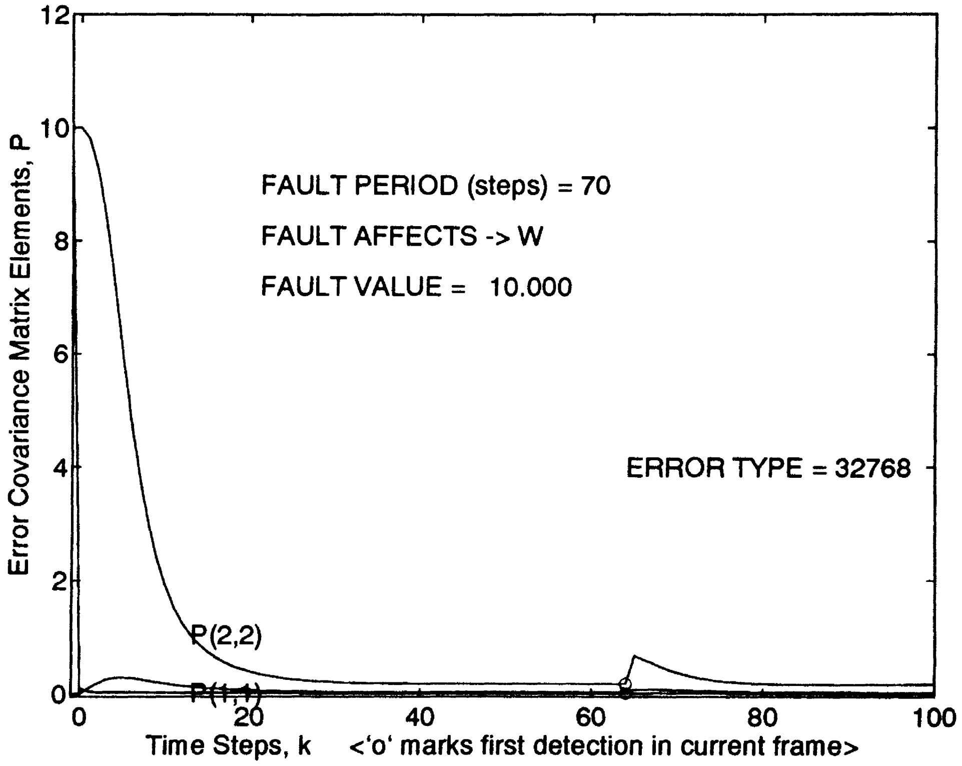


Figure 24. Software Fault Effect on Single Measurement UDU^T Error Covariance, Case 6, Steps 0 to 100

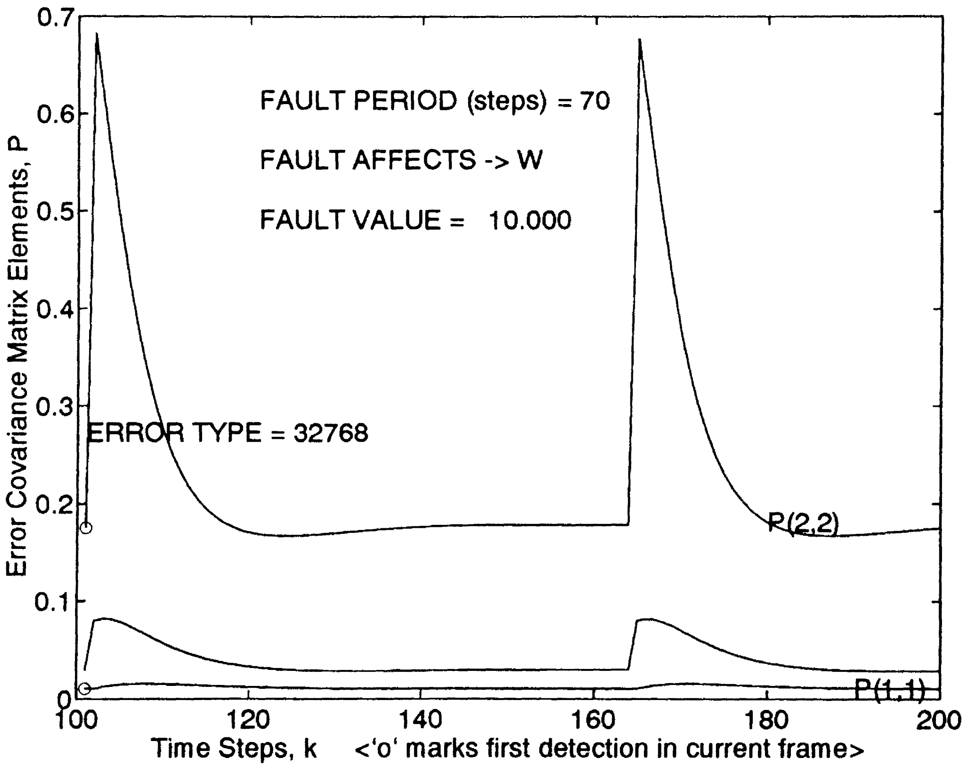


Figure 25. Software Fault Effect on Single Measurement UDU^T Error Covariance, Case 6, Steps 101 to 200

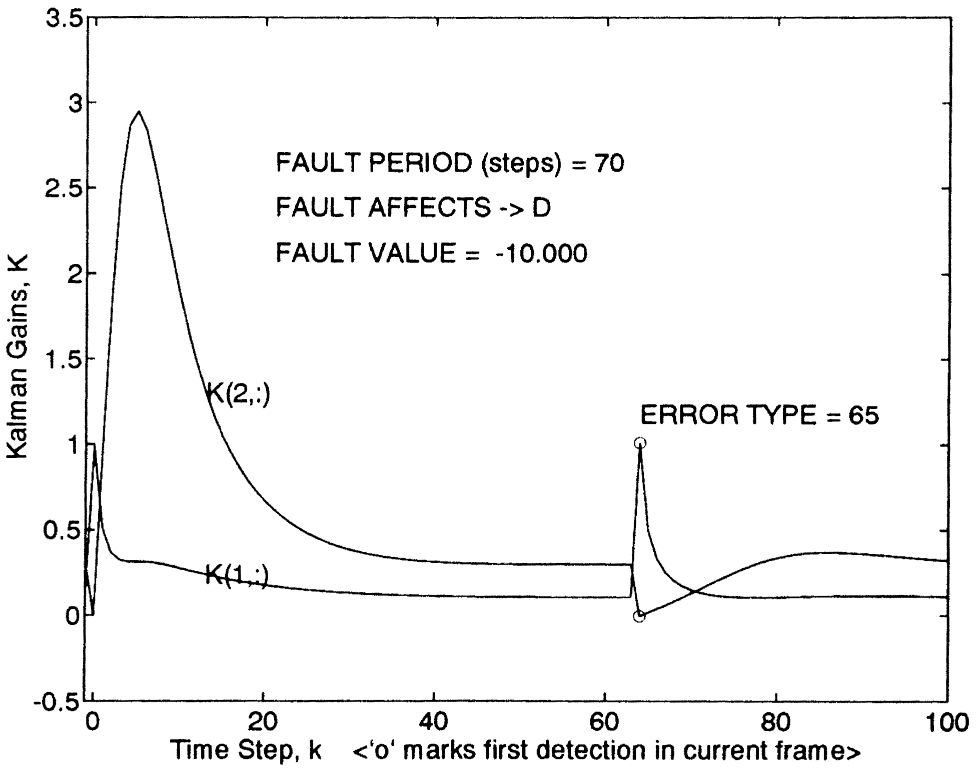


Figure 26. Software Fault Effect on Single Measurement UDU^T Kalman Gains, Case 7, Steps 0 to 100

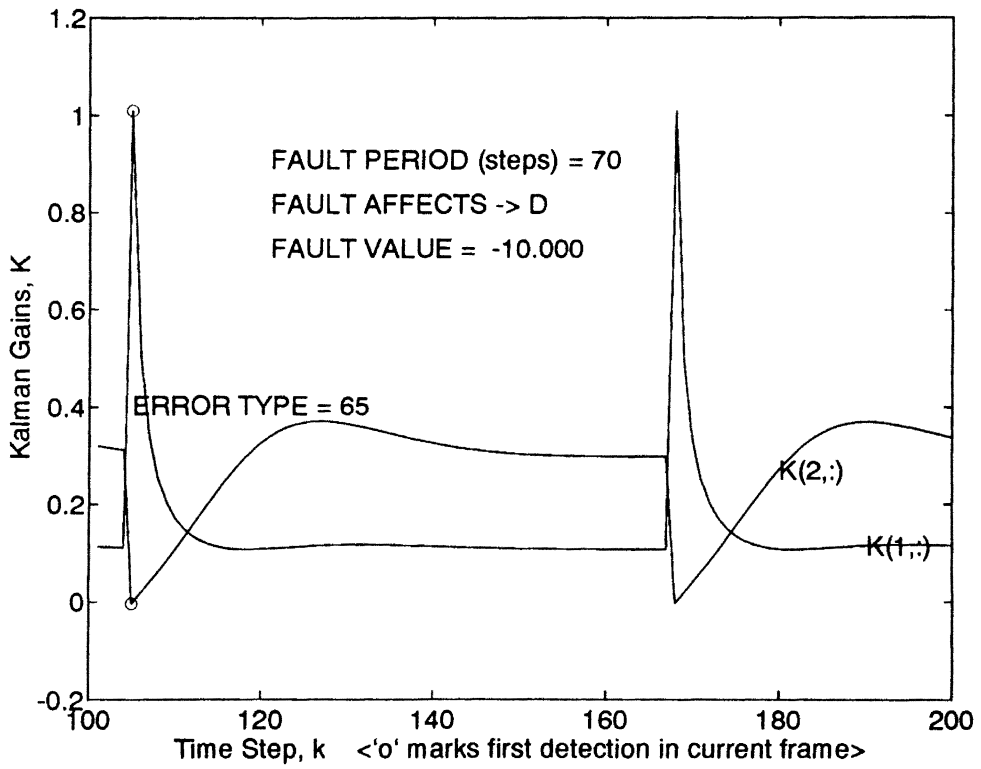


Figure 27. Software Fault Effect on Single Measurement UDU^T Kalman Gains, Case 7, Steps 101 to 200

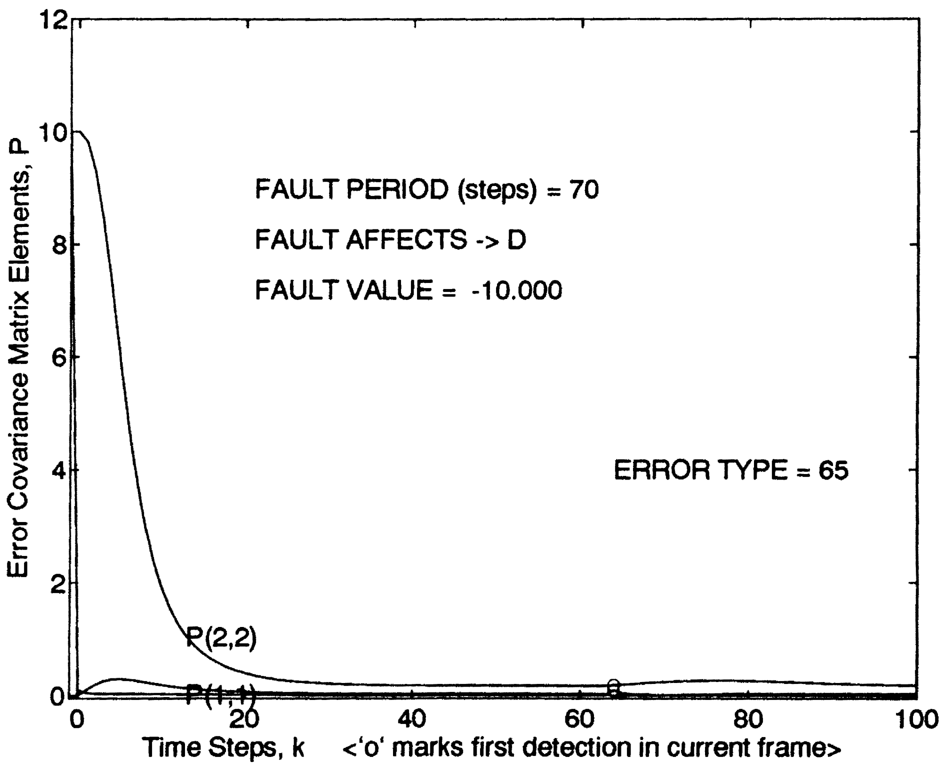


Figure 28. Software Fault Effect on Single Measurement UDU^T Error Covariance, Case 7, Steps 0 to 100

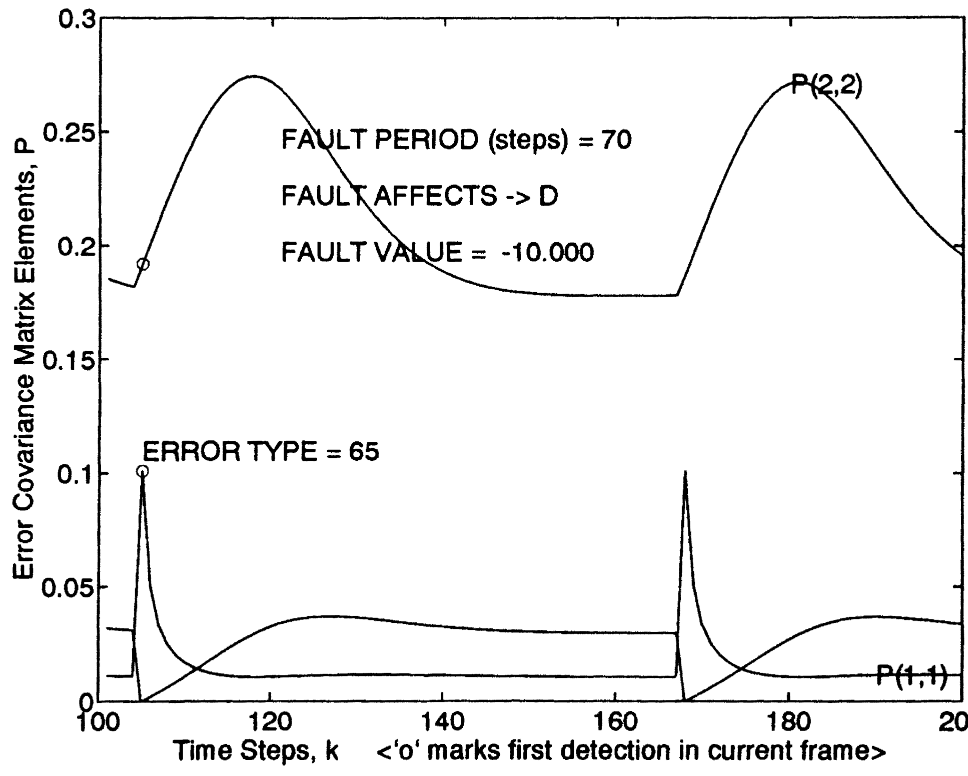


Figure 29. Software Fault Effect on Single Measurement UDU^T Error Covariance, Case 7, Steps 101 to 200

For each case, the initial starting time of the periodic faults affecting the calculations was delayed 40 to 70 steps to facilitate plotting and to avoid any confusion with the filter's startup period. As can be seen in the plots, this delay allowed the filter to reach its steady state values before being corrupted by faults. In a real-time application, a fault could occur at any time, including the startup period.

When the period of fault occurrence is very fast, such as 1 time step in the first two cases shown in Figures 4 through 7 and Figures 8 through 11, the covariance and gain calculations take on erroneous values. In the first case, which is shown in Figures 4 through 11, all erroneous values are fairly constant. In the second case, which is shown in Figures 8 through 11, the erroneous values of the Kalman gains are fairly constant, but the value for the second diagonal element of the covariance matrix is increasing with time. Also note in Figures 10 and 11 that the timing of the fault in the first element of D occasionally causes a spike in the calculation of the first diagonal element of P for some time steps. Although Figures 10 and 11 are the only figures presented which show this spike behavior when forming the covariance matrix P, the behavior was often found when simulating faults in other variables. Thus, faults can cause erroneous gains and covariances with either transient, fairly constant or increasing behaviors. Note that the "error type" shown in the figures is a number which corresponds to the locations in the software where the fault was first detected, but does not indicate whether the value is steady or transient. In regard to the remaining cases, as the period of the fault increases from once every 1 step to once every 15 steps in the case of Figures 12 and 13, and then to once every 70 steps in the other four cases in Figures 14 through 29, the algorithms react as if they have been reset with new initial conditions following the occurrence of each fault. With enough time between faults, the Kalman gains and covariance return to steady-state values. They return to steady-state values because state matrices and noise covariances were kept constant during the simulation so that effects of the fault could be illustrated separately from effects caused by varying system parameters or noise. Even though the calculations return to steady-state values, the important observation is that the erroneous Kalman gains resulting from the faulty calculations will likely cause incorrect state estimates, incorrect control actions, and unrecoverable system instabilities if corrective action is not taken upon immediate detection of a faulty calculation.

CHAPTER V

CONCLUSIONS

Future control systems need to exhibit increasingly better software fault tolerance. Systems which have human safety requirements, such as the automated highway system, are obvious examples of systems which will require software fault tolerance. With these types of systems in mind, concepts of software fault tolerance such as the Distributed Recovery Block scheme were reviewed in the context of control system applications. Several previously developed methods for identifying failures and maintaining suboptimal performance of control algorithms have been recast as candidate elements for the acceptance test in software fault tolerance schemes. In particular, the following conclusions can be made from the work of this thesis.

Algorithm Based Fault Tolerance (ABFT) techniques were shown to have the potential for use as quantitative measures for computation acceptance at the end of each time step of recursive estimation algorithms such as the Kalman Filter. To test the method, Bierman's UDU^T covariance factorization algorithms were modified to include ABFT methods and test cases were run with simulated faults. Faults causing erroneous Kalman gains and covariances with transient, fairly constant and/or increasing behaviors were immediately detected by the algorithm modifications. Operation counts for the ABFT modifications to the algorithms were tabulated versus the original operation counts. The required overhead of the modifications was tabulated as a percentage of the original algorithm operations for system orders of 10, 50, and 100 state variables. The overhead of the proposed algorithms is approximately 25% of the original operation count for the time update algorithm and 60% for the measurement update algorithm. Because the overhead is less than that required to run a duplicate process of the unmodified algorithms, the method may be particularly applicable when physically redundant processors are not desirable or available. An additional benefit of the modifications concerns the isolation of system faults to system components. The elimination of software faults (resulting from computing environment failures) as causes of large residual sequences is desirable when sensor and actuator failure detection decisions are being made. In addition, for implementations in which system matrices are passed as parameters into filter routines, checksum matrices provide an additional method of validating that the matrices are consistent and have been passed without corruption.

REFERENCES

[1] Bierman, G. J. *Factorization Methods for Discrete Sequential Estimation*. New York: Academic Press, 1977.

[2] Thornton, C. L. and Bierman, G. J. "UDU^T Covariance Factorization for Kalman Filtering." *Control and Dynamic Systems*. Vol. 16. C. T. Leodnes, Ed. New York: Academic Press, 1980, pp. 125-192.

[3] Bromley, K., Kung, S., Swartzlander, E. (Eds.). *Proceedings, International Conference on Systolic Arrays*. Washington, DC: IEEE Computer Society Press, 1988.

[4] Jacklin, S. A. "Arranging Computer Architectures to Create Higher-Performance Controllers." *Control and Dynamic Systems, Advances in Theory and Applications*, Vol. 29 (1988), pp. 67-99.

[5] Kim, K. H. "Designing Fault Tolerance Capabilities Into Real-Time Distributed Computer Systems." *Proceedings, IEEE Computer Society's Workshop on the Future Trends of Distributed Computing Systems in the 1990's*, Hong Kong, Sept. 1988, pp.318-328.

[6] Mariton, M. "Detection Delays, False Alarm Rates and the Reconfiguration of Control Systems." *International Journal of Control*, Vol. 49 (1989), pp. 981-992.

[7] Srichander, R., and Walker, B. K. "Stochastic Stability Analysis for Continuous Time Fault Tolerant Control Systems." *Proceedings of the 1991 American Control Conference*, pp. 493-501.

[8] Gylys, V. B. "Design of Real-Time Estimation Algorithms for Implementation in Microprocessor and Distributed Processor Systems." *Control and Dynamic Systems, Advances in Theory and Applications*. Vol. 19. C. T. Leodnes, Ed. New York: Academic Press, 1983, pp. 193-295.

[9] Astrom, K. J. and Wittenmark, B. *Adaptive Control*. Reading, MA: Addison-Wesley, 1989.

[10] Clarke, D. W. "Self-Tuning Controller Design and Implementation." *Real-Time Computer Control*. Bennett, S. and Linkens, D.A., Eds. London, U.K.: Peter Pergrinus, Ltd., 1984.

[11] Gelb, A. et al. *Applied Optimal Estimation*. Cambridge, MA: The MIT Press, 1974.

[12] Verhaegen, M. H., and Van Dooren, P. "New Insights in the Numerical Reliability Properties of Existing Kalman Filter Implementations." *Control and Dynamic Systems, Advances in Theory and Applications.* Vol. 29: *Advances in Algorithms and Computational Techniques in Dynamic Systems Control.* Part 2 of 3 (1988), pp. 1-45.

[13] Chin, L. "Advances in Computational Efficiencies of Linear Filtering," *Control and Dynamic Systems: Advances in Theory and Applications.* Vol. 19. C. T. Leodnes, Ed. New York: Academic Press, 1983, pp. 125-192.

[14] Sorenson, H. W. "Kalman Filtering Techniques." *Advances in Control Systems.* Vol. 3. C. T. Leodnes, Ed. New York: Academic Press, 1966.

[15] Kerr, T. "Use of Idempotent Matrices to Validate Linear Systems Software." *IEEE Transactions On Aerospace and Electronic Systems.* Vol. 26, No. 6 (1990), pp. 935-952.

[16] Chen, G. "A Simple Treatment for Suboptimal Kalman Filtering in Case of Measurement Data Missing." *IEEE Transactions on Aerospace and Electronic Systems,* Vol. 26, No. 2 (1990), pp. 413-415.

[17] Luck, R. and Ray, A. "Delay Compensation in Integrated Communication and Control Systems." *Proceedings of the 1990 American Control Conference,* pp. 2045-2055.

[18] Zhang, Z. and Ray, A. "Robust Compensation of Distributed Delays In Integrated Communication and Control Systems." *Proceedings of the 1991 American Control Conference,* pp. 2183-2184.

[19] Lynch, P. M. and Figueroa, J. F. "Position Estimation With Intermittent Measurements." *Proceeding of the 1991 American Control Conference,* pp. 2280-2285.

[20] Isermann, R. "Process Fault Detection Based on Modeling and Estimation—A Survey." *Automatica,* Vol. 20 (1984), pp. 387-404.

[21] Basseville, M. "Detecting Changes in Signals and Systems—A Survey." *Automatica,* Vol. 24 (1988), pp. 309-326.

[22] Gertler, J. J. "Survey of Model-Based Failure Detection and Isolation in Complex Plants." *IEEE Control Systems Magazine,* Vol. 8 (1988), pp. 3-11.

[23] Frank, P. M. "Fault Diagnosis in Dynamic Systems Using Analytical and Knowledge-Based Redundancy—A Survey and Some New Results." *Automatica,* Vol. 26 (1990), pp. 459-474.

[24] Willsky, A. S. "A Survey of Design Methods for Failure Detection in Dynamic Systems." *Automatica,* Vol. 12 (1976), pp. 601-611.

[25] Panossian, H. "Algorithms For System Fault Detection Through Modeling and Estimation Techniques." *Control and Dynamic Systems, Advances in Theory and Applications.* Vol. 29: *Advances in Algorithms and Computational Techniques in Dynamic Systems Control.* Part 2 of 3 (1988), pp. 47-66.

[26] Franklin, G. F., Powell, J. D., and Workman, M. L. *Digital Control Systems, 2nd ed* Reading, Mass.: Addison-Wesley, 1990.

[27] Hanselmann, H. "Implementation of Digital Controllers—A Survey." *Automatica,* Vol. 23, No. 1 (1987), pp. 7-32.

[28] Huang, K. and Abraham, J. A. "Algorithm-Based Fault Tolerance for Matrix Operations." *IEEE Transactions on Computers,* Vol. C-33 (1984), pp. 518-528.

[29] Anfinson, C. J., and Luk, F. T. "A Linear Algebraic Model of Algorithm-Based Fault Tolerance." *Proceeding of the International Conference on Systolic Arrays,* May 25-27, 1988, pp. 483-493.

[30] Jou , J. Y., and Abraham, J. A. "Fault-Tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures." *Proceedings, IEEE, Special Issue on Fault Tolerance in VLSI,* Vol. 74, No. 5 (May 1986), pp. 732-741.

General References

[31] Chen, C. H., and Cherkassky, V. "Task Reallocation for Fault Tolerance in Multiprocessor Systems." *Proceedings of IEEE National Aerospace and Electronics Conference, NAECON '90* (1990), pp. 495-500.

[32] Singer, R. A., and Sea, R. G. "Increasing the Computational Efficiency of Discrete Kalman Filters." *IEEE Transactions on Automatic Control,* 1971.

[33] Doty, K.W., McEntire, P.l., Reilly, J.G., and Sridhar, B. "Software Allocation for Distributed Signal Processors." *Real Time Signal Processing V, SPIE,* Vol. 341 (1982).

[34] Jones, R. H. "Maximum Likelihood Fitting of ARMA Models to Time Series With Missing Observations." *Technometrics,* Vol. 22, No. 3 (1980), pp. 389-394.

[35] Masreliez, C. J., and Martin, R. D. "Robust Bayesian Estimation for the Linear Model and Robustifying the Kalman Filter." *IEEE Transactions on Automatic Control.* Vol. AC-22, No. 3 (1977), pp. 361-371.

[36] Puthenpura, S. and Sinha, N. K. "Robust Bootstrap Method for Joint Estimation of States and Parameters of a Linear System." *Journal of Dynamic Systems, Measurement, and Control,* Vol. 108 (1986), pp. 255-263.

APPENDIX

SIMULATION SOFTWARE

```
/*************************************************************************/
/** File: udursf2.c rountines                                           **/
/** Purpose: UDU' Kalman filter procedures, performance analysis        **/
/** and plotting in matlab, testing ABFT tolerance techniques.          **/
/** For use on rsf.ceat.okstate.edu because of availability of matlab.  **/
/**                                                                     **/
/** Originator: MRM                                                     **/
/** Date: 5-11-93                                                       **/
/** modification history                                                **/
/** --------------------                                                **/
/** uduabrsf.c:                                                         **/
/** 02a, 12may93, mrm  added rogue routine and algorithm based fault    **/
/**                    tolerance techniques to the udufilter.c program. **/
/** 02b, 17may93, mrm  added things to make the program work on         **/
/**                    rsf.ceat.okstate.edu.                            **/
/** 02c, 19may93, mrm  changed error codes to base 2 numbers            **/
/**                    and added capability to corrupt D.               **/
/** udursf2.c:                                                          **/
/** 02d, 27may93, mrm  changed to 1 second time step, DEBUG ifdef's for **/
/**                    a lot of printf statements, added control of when **/
/**                    fault for W and DD is introduced.                **/
/*************************************************************************/
#include "vxWorks.h"
#include "stdioLib.h"
#include "ioLib.h"
#include "taskLib.h"
#include "wdLib.h"
#include "a_out.h"
#include "strLib.h"
#include "fioLib.h"
#include "vme.h"
#include "math.h"

/* macro's   */
#define mmin(a,b) (((a) < (b))? (a):(b))
#define mmax(a,b) (((a) < (b))? (b):(a))
#define abs(x)  (((x) > 0.) ? (x):-(x))


/* task oriented definitions */
#define STACKSIZE   5000

/* A/D definitions */
#define MAX_SNGL_ENDED_CHANNELS   32
#define MAX_DIFF_CHANNELS         16
#define MAX_PAST_MEASUREMENTS     50

/* ABFT USUAGE     */
#define EPSILON        0.00001

/* Definitions for Direct Digital Control code segments */
#define TICKS_PER_SEC   60
#define PI             3.14159265358979323846
#define STATES         6 /* (maximum number of states n) + 1 */
#define INPUTS         6 /* (maximum number of inputs) + 1 */
#define MEAS           6
#define PROCESS_NOISE  6
#define MEAS_NOISE     6
#define ID            10

/* Global Measurements */

double y[MAX_SNGL_ENDED_CHANNELS][MAX_PAST_MEASUREMENTS];
```

```
/* System Global Parameters     */

double phi[STATES][STATES];
double gamma[STATES][INPUTS];
double P[STATES][STATES];
double U[STATES][STATES];
double D[STATES];
double DD[STATES+PROCESS_NOISE];
double H[MEAS][STATES];
double W[STATES][STATES+PROCESS_NOISE];
double d[MEAS][INPUTS];
double x[STATES];
double K[STATES][MEAS];
double R[MEAS_NOISE];
double Q[PROCESS_NOISE];
double G[STATES][PROCESS_NOISE];

double rogue_value;

int states;
int process_noise;
int meas_noise;
int meas;
int rogue_delay;
int rogue_start;
int fault_type;

char go_on;

FILE *fpError1;
FILE *fpKgain;
FILE *fpU;
FILE *fpD;

/*************************************************************************/
/**   NOMENCLATURE:                                                     **/
/**   Time varying linear system:                                       **/
/**   x(k+1) = phi*x(k) + gamma*u + G*q                                 **/
/**   y = H*x + d*u + r                                                 **/
/**   Vectors and Matricies:                                            **/
/**   y - vector of measurements, (mx1)                                 **/
/**   u - vector of control inputs, (lx1)                               **/
/**   q - process noise, (px1)                                          **/
/**   r - measurement noise, (mx1)                                      **/
/**    r and q are mutually uncorrelated jointly Gaussian white noise   **/
/**    sequences.                                                       **/
/**   H - observation matrix (mxn)                                      **/
/**   phi - state transition matrix, (nxn)                              **/
/**   gamma - control input matrix, (nxl)                               **/
/**   G - (nxp)                                                         **/
/**   x - vector of states                                              **/
/**   K - kalman gain vector                                            **/
/**   P - original covariance matrix P                                  **/
/**   U - U factor of covariance P = UDU'                               **/
/**   D - D factor of covariance P = UDU'                               **/
/**   R - Measurement noise covariance matrix, positive definite (mxm) **/
/**       (vector of diagonal elements)                                 **/
/**   Q - Process noise covariance, positive semidefinite (pxp)         **/
/**                                                                     **/
/**   x(0) is multivariate Gaussian, with mean mx(0) and covariance     **/
/**        Px(0).  x(0) ~ N(x(0);mx(0),Px(0)),                          **/
/*************************************************************************/

SEM_ID sem_system;
SEM_ID sem_time_step_sync;
SEM_ID sem_updt;
SEM_ID sem_W_DD;

/* task identifications */
int tid1;
int tid2;
int tid3;
int tid4;
int tid5;
int tid6;

/* file descriptors  */
int fd1;
int fd2;
int fd3;
int fd4;

/*declaration of subroutines*/
void filter_init();
void sys_param_updt();
void simulator();
int meas_updt();
int ud_factor();
int ud_factor_prop();
void task_killer();
float get_time();
void printm();
void printv();
void cont();
void stop();
void start_me();
void rogue();
void abft_check();

/*********************************************************************/
/** filter_init() is the entry point for a filter spawned by       **/
/** a startup routine or function.                                   **/
/** modification history                                             **/
/** --------------------                                             **/
/** 02a, 19may93,  mrm   added fault_type to error1 output           **/
/** 02b, 27may93,  mrm   added DEBUG ifdef's for printf statements   **/
/*********************************************************************/

void filter_init()
{
    int time_step, index_by_100, k,j,l;
    int abft_error,error_flag, error_type;

    error_flag = FALSE;
    error_type = FALSE;
    abft_error = FALSE;
    index_by_100=0;
    time_step = 0;

    iam("moan","unixcshell");

    if((fpKgain=fopen("rsf:/u/moan/rs6000/Kgain.m","w"))==NULL)
        printf("\n Kgain.m fopen failed.\n");

    if((fpU=fopen("rsf:/u/moan/rs6000/UpperP.m","w"))==NULL)
        printf("\n UpperP.m fopen failed.\n");

    if((fpD=fopen("rsf:/u/moan/rs6000/DiagP.m","w"))==NULL)
        printf("\n DiagP.m fopen failed.\n");

    if((fpError1=fopen("rsf:/u/moan/rs6000/error1.m","w"))==NULL)
        printf("\n error1.m fopen failed.\n");

    taskDelay(10);

#ifdef DEBUG
    /* Print out P */
        printf("Printing P(0)\n");
        printm(P,states,states);
        printf("\n");
#endif DEBUG

    /* Initialize fault detection information to NULL state   */
        fprintf(fpError1,"\n error_type = 0;\n");
        fprintf(fpError1,"\n time_of_error = 0; \n");

   /* Initialize fault type information for plotting   */
        fprintf(fpError1,"\n fault_type = %d ;\n", fault_type);
        fprintf(fpError1,"\n rogue_start = %d ;\n", rogue_start);
        fprintf(fpError1,"\n rogue_delay = %d ;\n", rogue_delay);
        fprintf(fpError1,"\n rogue_value = %f ;\n", rogue_value);

   /* Print P0 to Upper.m  */
        fprintf(fpU,"\nP0=[ \n");
        for(k=0; k < states;k++)
            {
            for(j=0; j < states;j++)
                {
                fprintf(fpU,"%f ", P[k][j]);
                };
            fprintf(fpU,"; \n");
            };
        fprintf(fpU,"]; \n");

#ifdef DEBUG
    /* Print out phi */
        printf("Printing phi\n");
        printm(phi,states,states);
        printf("\n");
#endif DEBUG

        fprintf(fpU,"\nphi=[ \n");
        for(k=0; k < states;k++)
            {
            for(j=0; j < states;j++)
                {
                fprintf(fpU,"%f ", phi[k][j]);
                };
            fprintf(fpU,"; \n");
            };
        fprintf(fpU,"]; \n");

#ifdef DEBUG
     /* Print out R   */
        printf("Printing R\n");
        printv(R,meas_noise);
        printf("\n");
#endif DEBUG

        fprintf(fpU,"\nR=[ \n");
        for(k=0; k < meas_noise;k++)
            {
```

```
            fprintf(fpU,"%f ", R[k]);
            fprintf(fpU,"; \n");
            };
        fprintf(fpU,"]; \n");

        abft_error = ud_factor();

/* Evaluate fault status and report to error1.m if TRUE    */
        if(error_flag == FALSE)
            {
            if(abft_error != FALSE)
                {
                error_flag = TRUE;
                error_type = abft_error;
                fprintf(fpError1,"\ntime_of_error = %d ;\n", time_step);
                fprintf(fpError1,"\nerror_type = %d ;\n", error_type);
                };
            };

    #ifdef DEBUG
     /* Print out U */
        printf("Printing factored U(0)\n");
        printm(U,states,states);
        printf("\n");
    #endif DEBUG

        fprintf(fpU,"\nU=[  \n");
        for(k=0; k < states;k++)
            {
            for(j=0; j < states;j++)
                {
                if(k == j)
                 {
                 fprintf(fpU, "1.0 ");
                 }
                else
                fprintf(fpU,"%f ", U[k][j]);
                };
            };
        fprintf(fpU,"; \n");

    #ifdef DEBUG
     /* Print out D */
        printf("Printing factored D(0)\n");
        printv(D,states);
        printf("\n");
    #endif DEBUG

        fprintf(fpD,"\nD=[  \n");
        for(k=0; k < states;k++)
            {
            fprintf(fpD,"%f ", D[k]);
            };
            fprintf(fpD,"; \n");

         /* Start Printing out Kalman Gains  */
         fprintf(fpKgain, "\nK=[  \n");
         for(k=0; k < states;k++)
            {
            fprintf(fpKgain,"%f ", K[k][0]);
            };
            fprintf(fpKgain,"; \n");

         for(;;)
```

```
               {
               semTake(sem_time_step_sync);

/* loop for multiple measurements to be added here       */
/*             for(l=0; l < meas;l++)                    */
/*                 {                                     */
/*                 validate_meas();                      */

                   l=0;
                   semTake(sem_system);

                   abft_error=meas_updt(l);

                   /* Evaluate fault status and report to error1.m if TRUE   */
                   if(error_flag == FALSE)
                       {
                       if(abft_error != FALSE)
                         {
                         error_flag = TRUE;
                         error_type = abft_error;
                         fprintf(fpError1,"\ntime_of_error = %d ;\n", time_step);
                         fprintf(fpError1,"\nerror_type = %d ;\n", error_type);
                         };
                       };

    #ifdef DEBUG
                   /* Print out U */
                   printf("Printing meas_updt U\n");
                   printm(U,states,states);
                   printf("\n");
    #endif DEBUG

                   if(l == 0)
                     {
                      for(k=0; k < states;k++)
                         {
                         for(j=0; j < states;j++)
                           {
                           if(k == j)
                             {
                             fprintf(fpU, "1.0 ");
                             }
                           else
                           fprintf(fpU,"%f ",U[k][j]);
                           };
                         };
                      fprintf(fpU,"; \n");
                     };

    #ifdef DEBUG
                /* Print out D */
                   printf("Printing meas_updt D\n");
                   printv(D,states);
                   printf("\n");
    #endif DEBUG

                   if(l == 0)
                     {
                     for(k=0; k < states;k++)
                        {
                        fprintf(fpD,"%f ", D[k]);
                        };
                     fprintf(fpD,"; \n");
                     };
    #ifdef DEBUG
            /* Print out K */
               printf("Printing Kalman Gain Vector K\n");
               printm(K,states,1);
               printf("\n");
    #endif DEBUG

                 if(1 == 0)
                  {
                  for(k=0; k < states;k++)
                     {
                     fprintf(fpKgain,"%f ", K[k][0]);
                     };
                  fprintf(fpKgain,"; \n");
                  };

                  /*allow sys_param_udpt to update the system parameters */
                  /*if new values are available*/
                  semGive(sem_system);

/***********************************************************************/
/**    Comment is a stub for sequential state estimation update for each    **/
/**     measurement to be placed here.                                      **/
/***********************************************************************/

/**             }            **/

             semTake(sem_system);
             abft_error = ud_factor_prop();

            /* Evaluate fault status and report to error1.m if TRUE    */
            if(error_flag == FALSE)
                {
                if(abft_error != FALSE)
                    {
                    error_flag = TRUE;
                    error_type = abft_error;
                    fprintf(fpError1,"\ntime_of_error = %d ;\n", time_step);
                    fprintf(fpError1,"\nerror_type = %d ;\n", error_type);
                    };
                };

    #ifdef DEBUG
            /* Print out U */
            printf("Printing time propogated U\n");
            printm(U,states,states);
            printf("\n");
    #endif DEBUG

    #ifdef DEBUG
            /* Print out D */
            printf("Printing time  propogated D\n");
            printv(D,states);
            printf("\n");
    #endif DEBUG

/***********************************************************************/
/**   Comment is a stub for time propogating the state vector to the      **/
/**   next time step. (prediction of state at t(k+1)                       **/
/**   May also be good place for control law calculations                  **/
/***********************************************************************/

            semGive(sem_system);
```

```
/***********************************************************************/
/** Since global system data is protected by sem_system, this could      **/
/**  Allow sys_param_updt & control laws to proceed sending information   **/
/***********************************************************************/

             if(time_step == 100)
               {

               fprintf(fpU,"]; \n");
               fprintf(fpD,"]; \n");
               fprintf(fpKgain,"]; \n");

               fprintf(fpU,"\n index100 =  ");
               fprintf(fpU,"%d ; \n", index_by_100);

               printf("\n index100 =  ");
               printf("%d ; \n",index_by_100);

               ++index_by_100;

               time_step = 1;

               iam("moan","unixcshell");

               if((fclose(fpU)) == EOF)
               printf("\n fpU fclose failed.\n");

               if((fclose(fpD)) == EOF)
               printf("\n fpD fclose failed.\n");

               if((fclose(fpKgain)) == EOF)
               printf("\n fpKgain fclose failed.\n");

               if((fclose(fpError1)) == EOF)
               printf("\n fpError1 fclose failed.\n");

    if((fpKgain=fopen("rsf:/u/moan/rs6000/Kgain.m","w"))==NULL)
        printf("\n Kgain.m fopen failed.\n");

    if((fpU=fopen("rsf:/u/moan/rs6000/UpperP.m","w"))==NULL)
        printf("\n UpperP.m fopen failed.\n");

    if((fpD=fopen("rsf:/u/moan/rs6000/DiagP.m","w"))==NULL)
        printf("\n DiagP.m fopen failed.\n");

    if((fpError1=fopen("rsf:/u/moan/rs6000/error1.m","w"))==NULL)
        printf("\n error1.m fopen failed.\n");

               /* Initialize fault detection information to NULL state    */
               fprintf(fpError1,"\n error_type = 0;\n");
               fprintf(fpError1,"\n time_of_error = 0; \n");

               /* Initialize fault type information for plotting    */
               fprintf(fpError1,"\n fault_type = %d ;\n", fault_type);
               fprintf(fpError1,"\n rogue_start  = %d ;\n", rogue_start);
               fprintf(fpError1,"\n rogue_delay = %d ;\n", rogue_delay);
               fprintf(fpError1,"\n rogue_value = %f ;\n", rogue_value);

               error_flag = FALSE;
               error_type = FALSE;
               abft_error = FALSE;

               fprintf(fpU,"\n U=[ \n");
```

```
            fprintf(fpD,"\n D=[ \n");
            fprintf(fpKgain,"\n K=[ \n");

            }
          else
            {
             ++time_step;
            };

          };
}

/*********************************************************************/
/** sys_param_updt() is spawned to update system parameter when      **/
/** necessary.                                                        **/
/**                                                                   **/
/** modification history                                              **/
/** --------------------                                              **/
/** 01a, 12may93, mrm  added stuff for algorithm based fault          **/
/**                    tolerance.                                     **/
/**                                                                   **/
/*********************************************************************/

void sys_param_updt()
{
    int states_buf;
    int process_noise_buf;
    int meas_noise_buf;
    int meas_buf;
    int j,k;

    double phi_buf[STATES][STATES];
    double gamma_buf[STATES][INPUTS];
    double P_buf[STATES][STATES];
    double U_buf[STATES][STATES];
    double D_buf[STATES];
    double DD_buf[STATES+PROCESS_NOISE];
    double H_buf[MEAS][STATES];
    double W_buf[STATES][STATES+PROCESS_NOISE];
    double d_buf[MEAS][INPUTS];
    double x_buf[STATES];
    double K_buf[STATES][MEAS];
    double R_buf[MEAS_NOISE];
    double Q_buf[PROCESS_NOISE];
    double G_buf[STATES][PROCESS_NOISE];

        for(;;)
        {

/* This part is a stub for reading new parameters into a buffer    */
/* when a message or interrupt is sent to signal that new          */
/* parameters are available.                                       */

        semTake(sem_updt);
        states_buf=2;
        process_noise_buf=1;
        meas_noise_buf=1;
        meas_buf=1;

        phi_buf[0][0]=1.0;
        phi_buf[1][0]= 0;
        phi_buf[2][0]= 1.0;
        phi_buf[3][0]= 0;
        phi_buf[0][1]= .02;
        phi_buf[1][1]= 1.0 ;
        phi_buf[2][1]= 1.02 ;
        phi_buf[3][1]= 0 ;

        phi_buf[0][2]= 0 ;
        phi_buf[1][2]= 0 ;
        phi_buf[2][2]= 0 ;
        phi_buf[3][2]= 0 ;

        phi_buf[0][3]= 0 ;
        phi_buf[1][3]= 0 ;
        phi_buf[2][3]= 0 ;
        phi_buf[3][3]= 0 ;

        P_buf[0][0]=10;
        P_buf[0][1]= 0 ;
        P_buf[0][2]= 10 ;
        P_buf[0][3]= 0 ;

        P_buf[1][0]= 0 ;
        P_buf[1][1]= 10;
        P_buf[1][2]= 10;
        P_buf[1][3]= 0 ;

        P_buf[2][0]= 0 ;
        P_buf[2][1]= 0 ;
        P_buf[2][2]= 0 ;
        P_buf[2][3]= 0 ;

        P_buf[3][0]= 0 ;
        P_buf[3][1]= 0 ;
        P_buf[3][2]= 0 ;
        P_buf[3][3]= 0 ;

        R_buf[0]= 0.1;
        R_buf[1]= 0.1;

        Q_buf[0]=0.01;
        Q_buf[1]=0.01;
        Q_buf[2]=0;
        Q_buf[3]=0;
        Q_buf[4]=0;

        H_buf[0][0]=1.0;
        H_buf[0][1]=0.0;
        H_buf[0][2]=0.0;

        H_buf[1][0]=1.0;
        H_buf[1][1]=0.0;
        H_buf[1][2]=0.0;

        G_buf[0][0]= 0.0;
        G_buf[0][1]= 0.0;
        G_buf[0][2]= 0;
        G_buf[0][3]= 0;

        G_buf[1][0]= 1.0;
        G_buf[1][1]= 0;
        G_buf[1][2]= 0 ;
        G_buf[1][3]= 0 ;

        G_buf[2][0]= 1.0;
        G_buf[2][1]= 0 ;
```

```
         G_buf[2][2]= 0 ;
         G_buf[2][3]= 0 ;

         G_buf[3][0]= 0 ;
         G_buf[3][1]= 0 ;
         G_buf[3][2]= 0 ;
         G_buf[3][3]= 0.0;

/* End of stub                                  */

         semTake(sem_system);
         states=states_buf;
         process_noise=process_noise_buf;
         meas_noise=meas_noise_buf;
         meas=meas_buf;

/* Also reads in phi column checksums    */
         for(k=0; k < states;k++)
             {
             for(j=0; j < (states + 1);j++)
                 {
                 phi[j][k]=phi_buf[j][k];
                 };
             };

         for(k=0; k < states;k++)
             {
             for(j=0; j < states;j++)
                 {
                 P[k][j]=P_buf[k][j];
                 };
             };

/* Also reads in R diagonal checksum    */
         for(k=0; k < (meas_noise+1);k++)
             {
              R[k]=R_buf[k];
             };

/* Also reads in Q diagonal checksum    */
         for(k=0; k < (process_noise+1);k++)
             {
              Q[k]=Q_buf[k];
             };

/* Also reads in H column checksums   */
         for(k=0; k < (meas+1);k++)
             {
             for(j=0; j < states;j++)
                 {
                 H[k][j]=H_buf[k][j];
                 };
             };

/* Also reads in G column checksums    */
         for(k=0; k < (states+1);k++)
             {
             for(j=0; j < process_noise;j++)
                 {
                 G[k][j]=G_buf[k][j];
                 };
             };
         semGive(sem_system);
         };
```

```
}

/***********************************************************************/
/**  Message simulator stub, which controls how often sys_param_updt **/
/**  gets exercised.                                                  **/
/***********************************************************************/

void simulator()
{
  for(;;)
    {
    taskDelay(30000);
    semGive(sem_updt);
    };
}

/***********************************************************************/
/** U/D Measurement Update Algorithm after Bierman and Thornton.     **/
/** C language version coded by MRM from algorithm                   **/
/** given by V. Gylys on pp. 278, Control And Dynamic Systems,       **/
/** Advances in Theory and Applications, edited by C. T. Leondes     **/
/** Volume 19: Nonlinear And Kalman Filtering Techniques             **/
/**                                                                  **/
/**                                                                  **/
/** modification history                                             **/
/** --------------------                                             **/
/** 02a, 12may93, mrm   added checksum fault tolerance               **/
/** 02c, 19may93, mrm   changed error codes                          **/
/***********************************************************************/

int meas_updt(y_row)
    int y_row;
{
    double f[STATES];
    double g[STATES];
    double alpha[STATES];
    double v[STATES];
    double last_alpha,last_alpha2;
    double last_U;
    double temp;
    double lambda,lambda_check;
    double check;
    double f_sum,alpha_sum,g_sum,v_sum;

    int k,i,j;
    int check_status;

    check = 0;
    check_status = 0;

    /* I. calculate H*U    */
    for(i=0; i < states; i++)
        {
         f[i] = H[0][i];
         for(j=0; j < i; j++)
             {
             f[i] = f[i] + H[0][j]*U[j][i];
             };
        };

    /* Create f_sum = e'*H*U*e  */
    f_sum=H[meas][states-1];
    for(k=(states-2);k>=0;k--)
```

```
          {
          f_sum = f_sum + H[meas][k]*(U[k][states] + 1);
          };

     /* Calculate gi = Di*fi and checksum */
     g_sum = 0;
     for(i=0; i < states; i++)
          {
          g[i] = D[i]*f[i];
          g_sum += g[i];
          };

     /* check D diagonal checksum   */
     for(j=0; j < states; j++)
          {
          check += D[j];
          };

     if(fabs(check - D[states]) > EPSILON)
          {
          check_status += 1;
          };
     check = 0;

/* initialize v_sum, alpha_sum, lambda, lambda_check   */
     v_sum = 0;
     alpha_sum = 0;
     lambda = 0;
     lambda_check = 0;

/* Start U and D update    */
     last_alpha = R[y_row];
     last_alpha2 = R[y_row];

     for(j=0; j < states; j++)
        {
        alpha[j] = last_alpha + f[j]*g[j];
        alpha_sum += alpha[j];
        if(alpha[j] != 0)
             {
             /* fractional update D[j] and update checksum  */
             temp = D[j];
             D[j] = (last_alpha/alpha[j])*D[j];
             D[states] += D[j] - temp;
             };
        /* form v[j] and keep a checksum  */
        v[j] = g[j];
        v_sum += g[j];

        /* continue update  */
        if(j != 0)
             {
             if(last_alpha == 0)
                  {
                  lambda = 0;
                  }
             else
                  {
                  lambda = -f[j]/last_alpha;
                  };
             lambda_check = lambda;
             for(i=0; i < j;i++)
                  {
                  last_U = U[i][j];
                  temp = v[i]*lambda;
                  U[i][j] = U[i][j] + temp;
                  /* update U row checksum  */
                  U[i][states] += temp;
                  temp = last_U*v[j];
                  v[i] = v[i] + last_U*v[j];
                  /* update v_sum  */
                  v_sum += temp;
                  };
             };

        /* check lambda */
        if(fabs(lambda - lambda_check) > 0.000001)
             {
             if(check_status < 2)
                  {
                  check_status += 2;
                  };
             };

        /* check last_alpha's */
        if(fabs(last_alpha - last_alpha2) > 0.000001)
             {
             if(check_status < 4)
                  {
                  check_status += 4;
                  };
             };
        last_alpha = alpha[j];
        last_alpha2 = alpha[j];
        };

       /* Kalman gain column vector for the y_row measurement */
       /* and form column checksum                            */
         K[states][y_row] = 0;
         for(i=0; i < states; i++)
           {
           K[i][y_row] = v[i]/alpha[(states-1)];
           K[states][y_row] += K[i][y_row];
           };

/* check f' = H*U   */
    check = 0;
    for(k=0;k<states;k++)
        {
        check = check + f[k];
        };

    if(fabs(check - f_sum) > EPSILON)
        {
        check_status += 8;
        };
    check = 0;

/* check g = D*f   */
    for(j=0; j < states; j++)
        {
        check += g[j];
        };

    if(fabs(check - g_sum) > EPSILON)
        {
        check_status += 16;
        };
        check = 0;

    /* check row checksums of U  */
        for(j=0; j < states-1; j++)
            {
             for(k=states-1; k > j; k--)
                 {
                  check += U[j][k];
                 };
             if(fabs(check - U[j][states]) > EPSILON)
                 {
                 if(check_status < 32)
                     {
                     check_status += 32;
                     };
                 };
              check = 0;
            };

    /* check sums for D  */
        for(j=0; j < states; j++)
            {
            check += D[j];
            };
        if(fabs(check-D[states]) > EPSILON)
            {
            check_status += 64;
            };
        check = 0;

    /* check alpha   */
        for(j=0; j < states; j++)
            {
            check += alpha[j];
            };

        if(fabs(check - alpha_sum) > EPSILON)
            {
            check_status += 128;
            };
         check = 0;

    /* check v   */
        for(j=0; j < states; j++)
            {
            check += v[j];
            };

        if(fabs(check - v_sum) > EPSILON)
            {
            check_status += 256;
            };

        return(check_status);
    }


/******************************************************************/
/** U D Factorization Algorithm                                 **/
/** C language version coded by MRM from algorithm              **/
/** given by V. Gylys on pp. 282, Control And Dynamic Systems,  **/
/** Advances in Theory and Applications, edited by C. T. Leondes **/
/** Volume 19: Nonlinear And Kalman Filtering Techniques        **/
/**                                                             **/
/** Input: nxn symmetric matrix P, with main-diagonal and       **/
/**        upper-triangular elements stored in an n x n array P. **/
/**                                                             **/
/** Output: n x n unit-diagonal, upper-triangular matrix U, with **/
/**         its upper triangular portion stored in n x n array  **/
/**         U (which can be "equivalenced" with array           **/
/**         P so that the original P is destroyed).             **/
/**                                                             **/
/** Output: The main-diagonal elements of n x n diagonal matrix D **/
/**         stored in vector D (which optionally can be stored in **/
/**         locations of the main-diagonal elements of array P). **/
/**                                                             **/
/** Remarks: the algorithm does not explicitly generate the main- **/
/**          diagonal unit elements of U                        **/
/**                                                             **/
/** modification history                                        **/
/** --------------------                                        **/
/** 02a, 12may93, mrm   added checksum fault tolerance          **/
/** 02c, 19may93, mrm   changed error codes                     **/
/******************************************************************/

int ud_factor()
{
    double alpha;
    double beta;
    double check;
    int i,k,j;
    int check_status;

    check = 0;
    check_status = 0;
    D[states]=0;

    for(j=states-1; j > 0; j--)
        {
        D[j] = P[j][j];
        /* D[states] is the location of the checksum  */
        D[states] += D[j];
        alpha = 1.0/D[j];
        for(k=0; k < j; k++)
            {
            if(j == (states-1))
                {
                U[k][states] = 0;
                };
            beta = P[k][j];
            U[k][j] = alpha*beta;
            /* U[k][states] is the location of the rowchecksums */
            U[k][states] += U[k][j];
            for(i=0; i < k; i++)
                {
                P[i][k] -= beta*U[i][j];
                };
            };
        };
    D[0]=P[0][0];
    D[states] += D[0];

/* check sums   */
    for(j=0; j < states; j++)
        {
        check += D[j];
        };
    if(fabs(check-D[states]) > EPSILON)
```

```
        {
        check_status += 512;
        };

     check = 0;

/* check row sums   */
    for(j=0; j < states-1; j++)
        {
         for(k=states-1; k > j; k--)
             {
              check += U[j][k];
             };
         if(fabs(check - U[j][states]) > EPSILON)
             {
             if(check_status < 3)
                 {
                 check_status += 1024;
                 };
             };
         check = 0;
        };

    return(check_status);

}


/*********************************************************************/
/** U/D Factor Propagation (Time Update)                            **/
/** C language version coded by MRM from algorithm                  **/
/** given by V. Gylys on pp. 284, Control And Dynamic Systems,      **/
/** Advances in Theory and Applications, edited by C. T. Leondes    **/
/** Volume 19: Nonlinear And Kalman Filtering Techniques            **/
/**                                                                 **/
/** Input: U,D,Q,G                                                  **/
/**                                                                 **/
/** Calc: n x N symmetric matrix W, with rows w1^T,...wn^T.         **/
/**                                                                 **/
/** Calc: N x N diagonal matrix DD defined by (D 0;0 Q)             **/
/**                                                                 **/
/** Output: the upper triangular part U of propagated n x n         **/
/**         unit-diagonal, upper-triangular matrix U.               **/
/**                                                                 **/
/** Output: the main diagonal elements, stored as a vector D,       **/
/**         of n x n diagonal matrix D.                             **/
/**                                                                 **/
/** Define: wj^(0) = wj for j = 1, ..., n.                          **/
/**                                                                 **/
/**                                                                 **/
/** modification history                                            **/
/** --------------------                                            **/
/** 02a, 12may93, mrm   added checksum fault tolerance.             **/
/** 02c, 19may93, mrm   changed error codes.                        **/
/** 02d, 27may93, mrm   added sem_W_DD to control when fault occurs. **/
/*********************************************************************/

int ud_factor_prop()
{
    double DD_inner_prod;
    double check,W_sum,Temp;
    int i,j,k;
    int check_status;
```

```
    check = 0;
    check_status = 0;

/* form large DD matrix of diagonal elements  */
    for(i=0;i<states;i++)
        {
        DD[i]=D[i];
        };

    for(i=0;i<process_noise;i++)
        {
        DD[states+i]=Q[i];
        };

    /* Form diagonal checksum   */
    DD[states+process_noise] = D[states] + Q[process_noise];

/*** Put DD fault here   ***/
    if (fault_type == 3)
        {
        semGive(sem_W_DD);
        };

/***** Create W =[PHI*U | G]  ********************************/
    for(i=0;i<states;i++)
        {
          for(j=0;j<states;j++)
            {
              W[i][j]=phi[i][j];
              for(k=j-1;k>=0;k--)
                {
                W[i][j]=W[i][j]+phi[i][k]*U[k][j];
                };
            };
        };

/* Create (e'*PHI)*U  */
     for(j=0;j<states;j++)
        {
        W[states][j]=phi[states][j];
        for(k=j-1;k>=0;k--)
            {
            W[states][j]=W[states][j]+phi[states][k]*U[k][j];
            };
        };

/* Create check = e'*PHI*U*e  */
     check=phi[states][states-1];
     for(k=(states-2);k>=0;k--)
        {
        check = check + phi[states][k]*(U[k][states] + 1);
        };

/* check calculation of PHI*U   */
     W_sum = 0;
     for(j=0;j<states;j++)
        {
        for(k=0;k<states;k++)
            {
            W_sum = W_sum + W[k][j];
            };
        };

     if(fabs(check - W_sum) > EPSILON)
```

```
            {
            check_status += 2048;
            };
      check = 0;

/* also include column checksum elements for G in bottom row of W */
    for(i=0;i<states+1;i++)
        {
          for(j=0;j<process_noise;j++)
            {
             W[i][states+j]=G[i][j];
            };
        };
/***** Perform D-orthogonalization  **************/
    D[states]=0.0;

    /** reset U[i][states] = 0  ***/
    for(i=0;i < states;i++)
        {
        U[i][states]=0;
        };

    /*** Put W fault here   ***/
    if (fault_type == 4)
        {
        semGive(sem_W_DD);
        };


    /** start orthogonalization  **/
    for(j=states-1; j > 0; j--)
        {
        D[j]=0.0;
        for(i=0; i < (states+process_noise); i++)
            {
            D[j] += W[j][i]*W[j][i]*DD[i];
            };
        /*** New diagonal checksum  ***/
        D[states] += D[j];
        for(i=0; i < j; i++)
            {
            DD_inner_prod=0.0;
            for(k=0; k < (states+process_noise); k++)
                {
                DD_inner_prod += W[i][k]*W[j][k]*DD[k];
                };
            U[i][j] = DD_inner_prod/D[j];
            /** Keep track of new row checksum for U  **/
            U[i][states] += U[i][j];
            for(k=0; k < (states+process_noise); k++)
                {
                Temp = U[i][j]*W[j][k];
                W[i][k] = W[i][k] - Temp;
                /** Also subtract temp from W[states][k] **/
                W[states][k] = W[states][k] - Temp;
                };
            };
        };
    D[0]=0.0;
    for(i=0; i < (states+process_noise); i++)
        {
        D[0] += W[0][i]*W[0][i]*DD[i];
        };
```

```
/*** last update of diagonal checksum for D  ***/
D[states] += D[0];

/* check row checksums of U  */
for(j=0; j < states-1; j++)
    {
     for(k=states-1; k > j; k--)
         {
          check += U[j][k];
         };
     if(fabs(check - U[j][states]) > EPSILON)
         {
         if(check_status < 4096)
             {
             check_status += 4096;
             };
         };
     check = 0;
    };

/* check sums for D  */
for(j=0; j < states; j++)
    {
    check += D[j];
    };
if(fabs(check-D[states]) > EPSILON)
    {
    check_status += 8192;
    };
check = 0;

/* check sums for DD  */
for(j=0; j < (states + process_noise); j++)
    {
    check += DD[j];
    };
if(fabs(check-DD[states+process_noise]) > EPSILON)
    {
    check_status += 16384;
    };
check = 0;

/* check column sums of W~  */
for(j=0; j < (process_noise + states); j++)
    {
     for(k=0; k < states; k++)
         {
          check += W[k][j];
         };
     if(fabs(check - W[states][j]) > EPSILON)
         {
         if(check_status < 32768)
             {
             check_status += 32768;
             };
         };
     check = 0;
    };

/** more fault control for W and DD  **/

if (fault_type == 3)
    {
    semTake(sem_W_DD);
```

```
        };

  if (fault_type == 4)
        {
        semTake(sem_W_DD);
        };

    /** return status  **/
    return(check_status);
}


/***************************************************************/
/** Task killer is called to delete all the spawned tasks.    **/
/***************************************************************/

void task_killer(i)
     int i;
{

     if(i == 0)
        {
        taskDelete(tid1);
        taskDelete(tid2);
        taskDelete(tid3);
        taskDelete(tid4);
        taskDelete(tid5);
/*      taskDelete(tid6);          */

        fclose(fpU);
        fclose(fpD);
        fclose(fpKgain);
        fclose(fpError1);

        };
     return;
}
/*******************************************************************/
/**      get_time - returns a floating point number that contains the **/
/**      number of SECONDS on the global tick counter.            **/
/**                                                                **/
/*******************************************************************/
float get_time()
{
        ULONG ticks;
        ticks = tickGet();
        return((float) ticks/TICKS_PER_SEC);
}

/************************************************************/
/** Print out matricies                                    **/
/************************************************************/
void printm(a,row,col)
    int row,col;
    double a[][STATES];
{

    int i,j,btm,top,count;

        printf("\n");
        btm=top=0;
        while(btm<col)
            {
            top=mmin(col,(btm+8));
            printf("printing matrix columns %d to %d\n",btm,(top-1));
            for(j=0;j<row;j++)
                {
                for(i=btm;i<top;i++)
                    {
                    printf(" %e", a[j][i]);
                    };
                printf("\n");
                };
                btm+=8;
            };
        return;
}

/************************************************************/
/** Print out vector                                       **/
/************************************************************/
void printv(a,length)
    int length;
    double a[];
{

    int i,btm,top;

        printf("\n");
        btm=top=0;
        while(btm<length)
            {
            top=mmin(length,(btm+8));
            printf("printing vector entries %d to %d\n",btm,(top-1));
                for(i=btm;i<top;i++)
                    {
                    printf(" %e", a[i]);
                    };
                printf("\n");
                btm+=8;
            };
        return;
}

/************************************************************/
/** continue                                               **/
/************************************************************/
void cont()

{
    semGive(sem_time_step_sync);
    return;
}

/************************************************************/
/** Stop                                                   **/
/************************************************************/
void stop()
{
   go_on = 'n';
   return;
}

/************************************************************/
/** Rogue software to cause problems with the calculation **/
/** modification history                                  **/
/** --------------------                                  **/
/** 02c, 19may93, mrm  added D fault type                 **/
```

```
/**********************************************************/
void rogue()
{
    taskDelay(rogue_start);
    for(;;)
     {
     if (fault_type == 1)
         {
         U[0][1] = rogue_value;
         }
   else if (fault_type == 2)
         {
         D[0] = rogue_value;
         }
   else if (fault_type == 3)
         {
         semTake(sem_W_DD);
         DD[1] = rogue_value;
         semGive(sem_W_DD);
         }
   else if (fault_type == 4)
         {
         semTake(sem_W_DD);
         W[1][0] = rogue_value;
         semGive(sem_W_DD);
         };

     printf("Gotta love me step \n");
     taskDelay(rogue_delay);
     };
}

/**********************************************************/
/** Start                                                **/
/** modification history                                 **/
/** --------------------                                 **/
/** 02b, 18may93, mrm   transfer file to rsf via ftp     **/
/** 02c, 19may93, mrm   added request for fault type     **/
/**                     and changed period to make faster **/
/** 02d, 27may93, mrm   added sem_W_DD, changed to 1 sec  **/
/**                     period (defaults changed).       **/
/**********************************************************/
void start_me()
{
    sysClkRateSet(TICKS_PER_SEC);
    tickSet(0);  /*sets the time reference to zero*/

   sem_system = semCreate();
   sem_time_step_sync = semCreate();
   sem_updt = semCreate();
   sem_W_DD = semCreate();

   hostAdd("rsf","139.78.3.7");

   netDevCreate("rsf:","rsf",1);

   semGive(sem_updt);
   semGive(sem_system);

  printf("\nPlease enter the fault type: 1 (U), 2 (D), 3 (DD), 4 (W):\n");
   if((scanf("%d",&fault_type)) == NULL ||
             (fault_type != 1 && fault_type != 2 && fault_type != 3
              && fault_type != 4))
     {
         printf("\nError entering fault type! Default 1 set.\n");
         fault_type = 1;
         };

 printf("\nPlease enter the value of the rogue information,\n");
 printf("with decimal point: i.e., -10.7\n");
  if((scanf("%lf",&rogue_value)) == NULL)
         {
         printf("\nError entering value! Default -10 set.\n");
         rogue_value = -10;
         };
 printf("\nThe value entered is: %f \n", rogue_value);

  printf("\nPlease enter the delay for the rogue routine:\n");
  if((scanf("%d",&rogue_delay)) == NULL)
         {
         printf("\nError entering delay! Default 6000 set.\n");
         rogue_delay = 6000;
         };

  printf("\nPlease enter the starting delay for the rogue routine:\n");
  if((scanf("%d",&rogue_start)) == NULL)
         {
         printf("\nError entering starting delay! Default 3000 set.\n");
         rogue_delay = 3000;
         };


  tid1 = taskSpawn("param",70,0,STACKSIZE,sys_param_updt);
  taskDelay(10);
  tid2 = period(1,cont);
  taskDelay(10);
  tid3 = taskSpawn("filter_init",80,VX_STDIO,STACKSIZE,filter_init);
  taskDelay(10);
  tid4 = taskSpawn("sim_int", 60,0,STACKSIZE,simulator);
  taskDelay(10);
  tid5 = taskSpawn("rogue",55,0,STACKSIZE,rogue);
  taskDelay(10);

}
```

```
% Matlab file used to plot filter performance
% and fault detection information.
% FILE: covplotrsf.m
% 1-18-93
% modification history
% --------------------
% 02a, 17may93, mrm
%             added features to remove the files
%             Upper.m, Kgain.m, DiagP.m, error1.m from the Unix
%             directory in which they reside.  Added while loop to
%             plot five (100 time steps) postscript plots and
%             then plot indefinitely to the screen thereafter.
%             Improved information on plots.
% 03a, 19may93, mrm
%             changed while loop to look for files and all kinds of stuff
% 03b, 27may93, mrm
%             changed stuff to make the postscript files look right.
%             changed stuff to make plots less cluttered.
%             changed for a 1 second time step
% 03c, 11june93, mrm
%             made unique files for plots.
% *****************************************************************
current_time = -1;
!sleep 150
while (1),
cpu_time1 = cputime;
 Kgain
 UpperP
 DiagP
 error1
 index100
  if current_time == -1,
    time_of_error = time_of_error + 2;
    elem_locat2 = 15;
    elem_locat1 = 15;
  else,
    elem_locat2 = 80;
    elem_locat1 = 90;
  end;
  if (current_time ~= (index100*100 + 1)) & ...
     (current_time ~= -1),
    current_time = (index100*100) + 1;
  end;
 dimen=size(U)*[1 0]'
 states=size(K)*[0 1]'
 steps=[current_time:1:(current_time+dimen-1)]';
 K_alt = K;
 % *This stuff was used to crop an area around the labels*
 %blank_1 = elem_locat1;
 %blank_2 = elem_locat1+7;
 %blank_3 = elem_locat2;
 %blank_4 = elem_locat2+7;
 %K_alt(blank_1:blank_2,:) = nan*K_alt(blank_1:blank_2,:);
 %K_alt(blank_3:blank_4,2) = nan*K_alt(blank_3:blank_4,2);
 %K_alt(blank_1:blank_2,2) = K(blank_1:blank_2,2);
 axis('auto');
 figure(1)
 %subplot(2,1,1)
 plot(steps,K_alt)
  if current_time == -1,
  axis([-1 100 -.5 3.5]);
  end;
 xlabel('Time Step, k    <'o' marks first detection in current frame>')
 ylabel('Kalman Gains, K')
```

```
 for index = 1:1:states,
  if index == 1,
   text(steps(elem_locat1),K(elem_locat1,1),'K(1,:)');
   if error_type ~= 0,
    hold on
    plot(steps(time_of_error),K(time_of_error,1), 'o')
    hold off
   end;
  elseif index == 2,
   text(steps(elem_locat2),K(elem_locat2,2),'K(2,:)');
   if error_type ~= 0,
    hold on
    plot(steps(time_of_error),K(time_of_error,2), 'o')
    hold off
   end;
  elseif index == 3,
   text(steps(elem_locat1),K(elem_locat1,3),'K(3,:)');
   if error_type ~= 0,
    hold on
    plot(steps(time_of_error),K(time_of_error,3), 'o')
    hold off
   end;
  elseif index == 4,
   text(steps(elem_locat1),K(elem_locat1,4),'K(4,:)');
   if error_type ~= 0,
    hold on
    plot(steps(time_of_error),K(time_of_error,4), 'o')
    hold off
   end;
  elseif index == 5,
  text(steps(elem_locat1),K(elem_locat1,5),'K(5,:)');
   if error_type ~= 0,
    hold on
    plot(steps(time_of_error),K(time_of_error,5), 'o')
    hold off
   end;
  elseif index == 6,
  text(steps(elem_locat1),K(elem_locat1,6),'K(6,:)');
   if error_type ~= 0,
    hold on
    plot(steps(time_of_error),K(time_of_error,6), 'o')
    hold off
   end;
  elseif index == 7,
  text(steps(elem_locat1),K(elem_locat1,7),'K(7,:)');
   if error_type ~= 0,
    hold on
    plot(steps(time_of_error),K(time_of_error,7), 'o')
    hold off
   end;
  elseif index == 8,
  text(steps(elem_locat1),K(elem_locat1,8),'K(8,:)');
   if error_type ~= 0,
    hold on
    plot(steps(time_of_error),K(time_of_error,8), 'o')
    hold off
   end;
  else;
  end;
end;
s=sprintf('FAULT PERIOD (steps) = %d', round(rogue_delay/(60)));
text((max(steps)-99), (max(max(K))*0.9), s);
s=sprintf('FAULT VALUE = %8.3f', rogue_value);
text((max(steps)-99), (max(max(K))*0.72), s);
```

```
 if fault_type == 1,
  text((max(steps)-99),(max(max(K))*0.81), 'FAULT AFFECTS -> U');
 end;
 if fault_type == 2,
   text((max(steps)-99),(max(max(K))*0.81), 'FAULT AFFECTS -> D');
 end;
 if fault_type == 3,
  text((max(steps)-99),(max(max(K))*0.81), 'FAULT AFFECTS -> DD');
 end;
 if fault_type == 4,
   text((max(steps)-99),(max(max(K))*0.81), 'FAULT AFFECTS -> W');
 end;
%text((max(steps)-20), (max(max(K))*0.9), date);
if error_type ~= 0,
%text(steps(time_of_error), (max(max(K))*.5), ...
%'o => SOFTWARE FAULT DETECTED');
s=sprintf('ERROR TYPE = %d', error_type);
text(steps(time_of_error), (max(max(K))*.4), s);
end;
title('SOFTWARE FAULT EFFECT ON SINGLE MEASUREMENT UDU' KALMAN GAINS')
if current_time <= 401,
 print -dps -append gain_cov.ps;
end;
% Error Covariance
figure(2);
%subplot(2,1,2)
for n = 1:1:dimen,
utemp = (reshape(U(n,:),states,states))';
ptemp=utemp*diag(D(n,:)',0)*utemp';
P(n,:)=reshape(ptemp',1,(states*states));
end;
P_alt = P;
%for index = 0:1:(states-1),
%P_alt(blank_3:blank_4,(1+index*(states+1))) = ...
%nan*P_alt(blank_3:blank_4,(1+index*(states+1)));
%end;
%P_alt(blank_3:blank_4,1) = P(blank_3:blank_4,1);
%P_alt(blank_1:blank_2,1) = nan*P_alt(blank_1:blank_2,1);
plot(steps,P_alt)
if current_time == -1,
axis([-1 100 -.05 12])
end;
xlabel('Time Steps, k      <'o' marks first detection in current frame>')
ylabel('Error Covariance Matrix Elements, P')
for index = 0:1:(states-1),
if index == 0,
text(steps(elem_locat1),P(elem_locat1,(1+index*(states+1))),'P(1,1)');
if error_type ~= 0,
hold on
plot(steps(time_of_error), P(time_of_error,(1+index*(states+1))), 'o')
hold off
end;
elseif index == 1,
text(steps(elem_locat2),P(elem_locat2,(1+index*(states+1))),'P(2,2)');
if error_type ~= 0,
hold on
plot(steps(time_of_error), P(time_of_error,(1+index*(states+1))), 'o')
hold off
end;
elseif index == 2,
text(steps(elem_locat2),P(elem_locat2,(1+index*(states+1))),'P(3,3)');
if error_type ~= 0,
hold on
plot(steps(time_of_error), P(time_of_error,(1+index*(states+1))), 'o')
```

```
hold off
end;
elseif index == 3,
text(steps(elem_locat2),P(elem_locat2,(1+index*(states+1))),'P(4,4)');
if error_type ~= 0,
hold on
plot(steps(time_of_error), P(time_of_error,(1+index*(states+1))), 'o')
hold off
end;
elseif index == 4,
text(steps(elem_locat2),P(elem_locat2,(1+index*(states+1))),'P(5,5)');
if error_type ~= 0,
hold on
plot(steps(time_of_error), P(time_of_error,(1+index*(states+1))), 'o')
hold off
end;
elseif index == 5,
text(steps(elem_locat2),P(elem_locat2,(1+index*(states+1))),'P(6,6)');
if error_type ~= 0,
hold on
plot(steps(time_of_error), P(time_of_error,(1+index*(states+1))), 'o')
hold off
end;
elseif index == 6,
text(steps(elem_locat2),P(elem_locat2,(1+index*(states+1))),'P(7,7)');
if error_type ~= 0,
hold on
plot(steps(time_of_error), P(time_of_error,(1+index*(states+1))), 'o')
hold off
end;
elseif index == 7,
text(steps(elem_locat2),P(elem_locat2,(1+index*(states+1))),'P(8,8)');
if error_type ~= 0,
hold on
plot(steps(time_of_error), P(time_of_error,(1+index*(states+1))), 'o')
hold off
end;
else;
end;
end;
s=sprintf('FAULT PERIOD (steps) = %d', round(rogue_delay/(60)));
text((max(steps)-99), (max(max(P))*0.9), s);
s=sprintf('FAULT VALUE = %8.3f', rogue_value);
text((max(steps)-99), (max(max(P))*0.72), s);
 if fault_type == 1,
  text((max(steps)-99),(max(max(P))*0.81), 'FAULT AFFECTS -> U');
 end;
 if fault_type == 2,
   text((max(steps)-99),(max(max(P))*0.81), 'FAULT AFFECTS -> D');
 end;
 if fault_type == 3,
  text((max(steps)-99),(max(max(P))*0.81), 'FAULT AFFECTS -> DD');
 end;
 if fault_type == 4,
   text((max(steps)-99),(max(max(P))*0.81), 'FAULT AFFECTS -> W');
 end;
%text((max(steps)-20),(max(max(P))*0.9), date);
if error_type ~= 0,
%text(steps(time_of_error), (max(max(P))*0.5), ...
%'o => SOFTWARE FAULT DETECTED');
s=sprintf('ERROR TYPE = %d', error_type);
text(steps(time_of_error), (max(max(P))*0.4), s);
end;
title('SOFTWARE FAULT EFFECT ON SINGLE MEAS. UDU' ERROR COVARIANCE');
```

```
if current_time <= 401,
 print -dps -append gain_cov.ps;
end;
if current_time == 401,
!cp gain_cov.ps thesis$$.ps
!rm gain_cov.ps
end;
current_time = current_time + 100;
 if current_time == 99,
   current_time = 101;
 end;
track_cputime = cputime - cpu_time1
whos
%
!rm UpperP.m
!rm DiagP.m
!rm Kgain.m
!rm error1.m
cpu_time2 = cputime
file_sum=0;
time_before_plot = 90;
while(file_sum < 4),
test_for_files = ls;
files_size = size(test_for_files)*[0 1]';
file_sum = 0;
for n1 = 1:1:(files_size-8),
  file_sum = file_sum + strcmp(test_for_files(1,n1:(n1+7)),'error1.m');
end;
 for n2 = 1:1:(files_size-8),
  file_sum = file_sum + strcmp(test_for_files(1,n2:(n2+7)),'UpperP.m');
end;
for n3 = 1:1:(files_size-7),
  file_sum = file_sum + strcmp(test_for_files(1,n3:(n3+6)),'DiagP.m');
end;
for n4 = 1:1:(files_size-7),
  file_sum = file_sum + strcmp(test_for_files(1,n4:(n4+6)),'Kgain.m');
end;
file_sum
time_before_plot = time_before_plot - (cputime - cpu_time2)
!sleep 20
end;
clear P
%return to matlab
end;
```

VITA

Michael R. Moan

Candidate for the Degree of

Master of Science

Thesis: A METHOD FOR DETECTING SOFTWARE FAULTS DURING UDU^T COVARIANCE CALCULATIONS USED IN KALMAN FILTERING

Major Field: Mechanical Engineering

Biographical:

Personal Data: Born in Tulsa, Oklahoma, August 9, 1963, the son of Ray O. and Kathleen L. Moan.

Education: Graduated from Booker T. Washington High School, Tulsa, Oklahoma, in 1981; received the Bachelor of Science degree in Mechanical Engineering from Oklahoma State University in May, 1986; completed requirements for the Master of Science degree in July, 1993.

Professional Experience: Mechanical Engineer, Frontier Engineering, Stillwater, Oklahoma, August 1985, to December 1990; Research Assistant, School of Mechanical Engineering, Oklahoma State University, January 1991, to July 1993; member of Tau Beta Pi and Pi Tau Sigma.