Advanced Prognosis and Health Management of Aircraft and Spacecraft Subsystems

by

Heemin Yi Yang

S.B.E.E.C.S., Massachusetts Institute of Technology (1998)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2000

Author .....
Department of Electrical Engineering and Computer Science
February 3, 2000

Certified by .....
David H. Staelin
Professor of Electrical Engineering
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Advanced Prognosis and Health Management of Aircraft and Spacecraft Subsystems

by

Heemin Yi Yang

Submitted to the Department of Electrical Engineering and Computer Science
on February 3, 2000, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Beacon Exception Analysis for Maintenance (BEAM) has the potential to be an efficient and effective model in detection and diagnosis of nominal and anomalous activity in both spacecraft and aircraft systems. The main goals of BEAM are to classify events from abstract metrics, reduce the telemetry requirements during normal and abnormal flight operations, and to detect and diagnose major system-wide changes. This thesis explores the mathematical foundations behind the BEAM process and analyzes its performance on an experimental dataset. Furthermore, BEAM's performance is compared to analysis done with principal component transforms. Metrics are established where accurate reduction of observable telemetry and detection of system-wide activities are stressed. Experiments show that BEAM is able to detect critical and yet subtle changes in system performance while principal component analysis proves to lack the sensitivity and at the same time requires more computation and subjective user inputs. More importantly, BEAM can be implemented as a real-time process in a more efficient manner.

Thesis Supervisor: David H. Staelin
Title: Professor of Electrical Engineering

Contents

List of Figures

List of Tables

Chapter 1

Introduction

When ground station personnel of either aircraft or spacecraft are presented with hundreds or even thousands of channels of information ranging from oil pressure to vibration measurements, they are often overwhelmed with too much information. Therefore, it is impossible to track every channel of information. Instead, they frequently follow channels that are considered critical. In effect, the majority of the channels are neglected and underutilized until a critical failure occurs, upon which the ground crew will exhaustively search all channels of information for signs of failure. The UltraComputing Technologies research group (UCT) at NASA's Jet Propulsion Laboratory (JPL) in Pasadena, California believed that a process could be developed which could bypass these problems.

1.1 A Brief History of the Beacon Exception Analysis Method

The Beacon Exception Analysis Method (BEAM) is currently being developed under the supervision of Ryan Mackey in the UCT. They have been developing cutting-edge diagnostics and prognostics utilities for both aircraft and spacecraft systems since 1995 during the final phase of Project Cassini, JPL's mission to Saturn. The Attitude and Articulation Control Subsystem (AACS) of the Cassini spacecraft contains over

2000 channels of information, ranging from temperature sensors to Boolean variables. The UCT saw an opportunity to develop a diagnostics tool that could simplify a complex system for analysis and at the same time improve both the detection time and false alarm rate. The UCT research group used the AACS as a performance benchmark for the development and validation of several diagnostic and prognostic utilities, one of which later developed into BEAM during the summer of 1998. BEAM

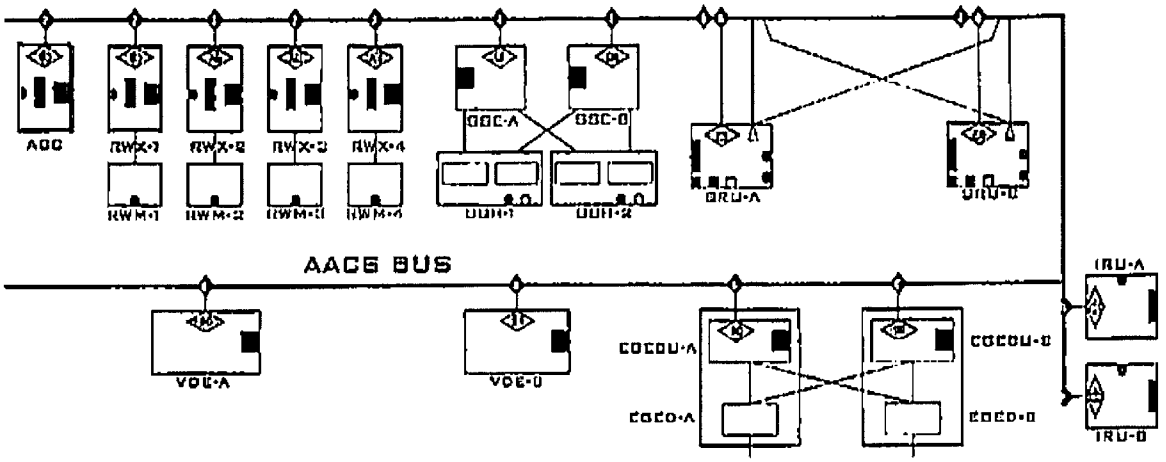

Figure 1-1: Block diagram of Cassini's Attitude and Articulation Control Subsystem

has successfully been demonstrated as a valid diagnostics tool on projects ranging from Cassini to NASA's X33 Reusable Launch Vehicle program. The UCT plans to implement BEAM as a critical element of prognostics and health maintenance (PHM) packages for future generation spacecraft and aircraft. Consequently, BEAM was developed to achieve the following goals:

1. Create an abstract measurement tool to track the entire system.

2. Reduce the telemetry set to simplify tracking of the system.

3. Maximize error detection while minimizing the false alarm rate.

4. Simplify diagnostics of system-wide faults.

BEAM attempts to achieve these goals in part by incorporating all channels of information. It takes advantage of the fact that many channels develop correlations over

time and that system-wide changes can be captured by detecting changes in these relationships.

1.2 Structure of Thesis

While principal component transforms is not an element of BEAM, it was, however, used as an alternative tool in analyzing the experimental dataset. In order to measure both the effectiveness and efficiency of BEAM as a real-time process, we required another process that could deliver analogous results on the dataset. When we explore the idea of reducing the number of significant variables in a system, principal component analysis becomes an obvious candidate in performing such a task. However, PCA has no method to automatically track system-wide events. Chapter 2 will present the mathematical foundations behind principal component analysis so that the reader will better understand the motivations behind the performance metrics of chapter 4. We will stress the aspects of PCA that are critical when applied to a real-life dataset.

Chapter 3 will present both the derivation and mathematical properties of the Beacon Exception Analysis Method. Furthermore, it will discuss the importance of coherence visualization and its implications.

Chapter 4 will analyze and discuss experimental results from both the BEAM and PCA processes. Their results will also be compared to test request documentation that describes in detail the different stages of the experimental dataset. In addition, the derivations of the performance metrics will be presented.

Finally, chapter 5 will summarize the work and discuss other components of BEAM that could prove to be a practical and efficient diagnostics vehicle.

1.3 Definition of Terms

Before we proceed, let us define what we mean by system-wide events. Obviously, the definition will differ among various types of datasets. However, we must note that in many of our experiments, we were blind to the functionality of the sensors

involved. In other words, we treated all channels of information as abstract variables whose values had no positive or negative implications on the system. Therefore, a system-wide event could be defined as either an expected change in the system or as a major or minor failure.

Chapter 2

Principal Component Analysis

2.1 Introduction

Principal Component Analysis (PCA) is a conventional and popular method for reducing the dimensionality of a large dataset while retaining as much information about the data as possible[1]. In the case of PCA, the variance and relationship among the different channels of data can be seen as the "information" within a dataset. PCA attempts to reduce the dimension of the dataset by projecting the data onto a new set of orthogonal variables, the principal components (PC)[2]. These new variables are ordered in decreasing variance. The idea is to capture most of the variance of the dataset within the first few PC's, thereby significantly reducing the dimension of the dataset that we need to observe. Section 2.2 will discuss the definition and derivation of the discrete-time Karhunen-Loève (KL) expansion that lies as the fundamental theory behind PCA. Then section 2.3 will explore a few but important mathematical properties of PCA. Finally, section 2.4 will investigate the implications of using a dataset with a finite collection of data points.

[1]PCA does not actually reduce the dimension of the dataset unless there are perfectly linear relationships among some variables. See section 2.3

[2]The PC's are orthogonal in the sense that the correlation matrix of the PC's is diagonal.

2.2 Discrete Time Karhunen-Loève Expansion

2.2.1 Definition

Let us define x_n to be a real-valued, zero-mean[3], discrete random process[4]. In addition, let Λ_x be the covariance matrix of x_n such that

$$\Lambda_x[i,j] = E[x_i x_j]. \tag{2.1}$$

Then the discrete-time KL expansion of x_n is defined as a linear combination of weighted basis functions in the form of

$$x_n = \sum_{i=1}^{N} y_i \phi_n[i] \tag{2.2}$$

such that the weights, y_i, are orthogonal random variables. The basis functions, $\phi_n[i]$, that produce uncorrelated weights are the eigenvectors of the covariance matrix, Λ_x, and the variance of y_i is the eigenvalue associated with eigenvector $\phi_n[i]$. If x_n is finite in length and contains N elements, then the total number of eigenvectors and eigenvalues will be less than or equal to N, i.e., the upper limit of the summation in equation 2.2.

2.2.2 Derivation

Let x_n be a real-valued, zero-mean, discrete random process with a covariance function, $K_{xx}[n,m]$, defined as

$$K_{xx}[n,m] = E[x_n x_m]. \tag{2.3}$$

We would like to find a set of deterministic, real-valued, orthonormal basis functions, $\phi_i[n]$, such that x_n can be represented in the form of equation 2.2 with y_i being

[3]If x_n has non-zero mean, we can define a new random process $\tilde{x}_n = x_n - E[x_n]$.

[4]While a random process is often denoted as x[n], we will denote x_n as a random process in order to remain consistent throughout the thesis.

zero-mean, uncorrelated random variables,i.e.,

$$E[y_i] = 0 \tag{2.4}$$

$$E[y_i y_j] = \begin{cases} \lambda_i, & i = j \\ 0, & i \neq j \end{cases} \tag{2.5}$$

Multiplying both sides of equation 2.2 by $\phi_n[j]$ and summing over all n will produce

$$\sum_{n=1}^{N} \phi_n[j] x_n = \sum_{n=1}^{N} \phi_n[j] \sum_{i=1}^{N} y_i \phi_n[i]. \tag{2.6}$$

Since y_i is independent of n and $\phi_n[j]$ is independent of i, we can rearrange the summations in order to exploit the orthonormal characteristics of ϕ_n. Equation 2.6 can be rewritten as

$$\sum_{n=1}^{N} \phi_n[j] x_n = \sum_{i=1}^{N} y_i \sum_{n=1}^{N} \phi_n[j] \phi_n[i], \tag{2.7}$$

and since we know that the projection of ϕ_i onto ϕ_j is non-zero and unity only when $i = j$, we can express the new set of random variables as

$$y_i = \sum_{n=1}^{N} x_n \phi_n[i]. \tag{2.8}$$

It is also straight forward to show that

$$E[y_i] = E[x_n] = 0 \tag{2.9}$$

and that

$$E[y_i y_j] = \begin{cases} \lambda_i, & i = j \\ 0, & i \neq j \end{cases} \tag{2.10}$$

for some value λ_i.

The orthonormal functions that produce uncorrelated coefficients in y_i can be obtained by taking advantage of the constraint imposed by the diagonal covariance matrix of y_i [5, p. 12]

For the remainder of the section, let us represent elements in their vector form in

order to simplify calculations through linear algebra. Let $\mathbf{y}$ be an array of the random variable coefficients defined as

$$\mathbf{y} \equiv \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \tag{2.11}$$

with the covariance matrix for $\mathbf{y}$ being

$$E[\mathbf{y}\mathbf{y}^T] = \begin{bmatrix} \lambda_1 & 0 & \cdots & \cdots \\ 0 & \lambda_2 & \cdots & \cdots \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \cdots & \lambda_N \end{bmatrix}. \tag{2.12}$$

Let $\mathbf{\Phi}$ represent the vector form of the orthonormal functions such that the i^{th} column of $\mathbf{\Phi}$ represents the i^{th} orthonormal function, $\phi_n[i]$:

$$\mathbf{\Phi} \equiv \begin{bmatrix} \phi_1[1] & \phi_1[2] & \cdots & \phi_1[N] \\ \phi_2[1] & \phi_2[2] & \cdots & \phi_2[N] \\ \vdots & \vdots & \ddots & \vdots \\ \phi_N[1] & \phi_N[2] & \cdots & \phi_N[N] \end{bmatrix}. \tag{2.13}$$

Then if we let

$$\mathbf{x} \equiv \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \tag{2.14}$$

we can rewrite equation 2.8 in vector form as

$$\mathbf{y} = \begin{bmatrix} \sum_{n=1}^{N} x_n \phi_n[1] \\ \sum_{n=1}^{N} x_n \phi_n[2] \\ \vdots \\ \sum_{n=1}^{N} x_n \phi_n[N] \end{bmatrix} \equiv \mathbf{\Phi}^T \mathbf{x} \tag{2.15}$$

Consequently, the covariance of $\mathbf{y}$ can also be written as

$$E[\mathbf{y}\mathbf{y}^T] = \mathbf{\Phi}^T \Lambda_x \mathbf{\Phi} \tag{2.16}$$

where

$$\Lambda_x = \begin{bmatrix} K_{xx}[1,1] & K_{xx}[1,2] & \cdots & \cdots \\ K_{xx}[2,1] & K_{xx}[2,2] & \cdots & \cdots \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \cdots & K_{xx}[N,N] \end{bmatrix}. \tag{2.17}$$

By utilizing the orthonormal property of $\mathbf{\Phi}$[5], we can multiply both sides of equation 2.16 by $\mathbf{\Phi}$ and extract the equation

$$\lambda_i \phi_n[i] = \Lambda_x \phi_n[i]. \tag{2.18}$$

Equation 2.18 reveals that the real-valued, orthonormal, deterministic set of functions that produce uncorrelated coefficients in y_i are the eigenvectors of the covariance matrix, Λ_x. In component form, equation 2.18 becomes

$$\sum_m K_{xx}[n,m]\phi_m[i] = \lambda_i \phi_n[i]. \tag{2.19}$$

2.2.3 KL Expansion and PCA

Before we proceed any further, it is important to address a few issues regarding PCA. While principal component analysis and discrete time KL expansion can be used interchangeably, it is necessary to make a few notes about the subtle differences in notations and terminologies between PCA and KL expansion. As described in the previous section, KL expansion allows us to represent a random process as a linear combination of weighted basis functions where the weights represent a new set of orthogonal random variables. PCA is the analysis of these new random variables,

[5] A matrix of orthonormal functions, Φ, has the convenient property that $\mathbf{\Phi}^{-1} = \mathbf{\Phi}^T$. In other words, $\Phi\Phi^T = \Phi\Phi^{-1} = \mathrm{II}$, the identity matrix [5, p. 13].

the principal components. Specifically, y_i in equation 2.2 can be regarded as the i^{th} principal component. Then to produce the i^{th} PC, we simply project the dataset onto the i^{th} eigenvector as seen in equation 2.8.

2.3 Mathematical Properties of Principal Components

Property 1. *The principal component y_i has variance λ_i, the eigenvalue associated with the i^{th} eigenvector[2, p. 9].*

This property was implicitly proven in section 2.2.2. The latter leads us to property 2, a key element of PCA.

Property 2. *For any integer q, $1 \leq q \leq p$, consider the orthonormal linear transformation*

$$\mathbf{y} = \mathbf{B}^T\mathbf{x}, \tag{2.20}$$

where $\mathbf{y}$ is a q-element vector and $\mathbf{B}^T$ is a (q x p) matrix, and let $\mathbf{\Lambda_y} = \mathbf{B}^T\mathbf{\Lambda_x}\mathbf{B}$ be the variance-covariance matrix for $\mathbf{y}$. Then the trace of Λ_y is maximized by taking $\mathbf{B} = \mathbf{\Phi_q}$, where $\mathbf{\Phi_q}$ consists of the first q columns of $\mathbf{\Phi}$[2, p. 9].

In the introduction to this chapter, we described how PCA can reduce the number of variables we need to observe. However, when we refer to equation 2.18, we can see that the total number of principal components will equal the total number of random variables produced by the random process, x_n[6]. Nevertheless, we should remember that the goal of PCA is to observe the first few PC's that contain the highest variances of the system. Property 2 reveals that in order to observe the first q PC's with the highest variances, we simply project the data onto the first q eigenvectors. While some data will be lost, we can often capture the majority of the variance in a system by carefully choosing the value of q and then observing the q PC's.

[6]The number of PC's will be smaller than the number of random variables if and only if there exists repetitive eigenvalues due to perfect linear relationships among some variables.

For example, let us examine a simple dataset that consists of eight variables. The following is a log-scale plot of the eigenvalues associated with the 8x8 covariance matrix. If we limit ourselves to observing only the first 4 principal components, we have captured over 79% of the variance in the system.

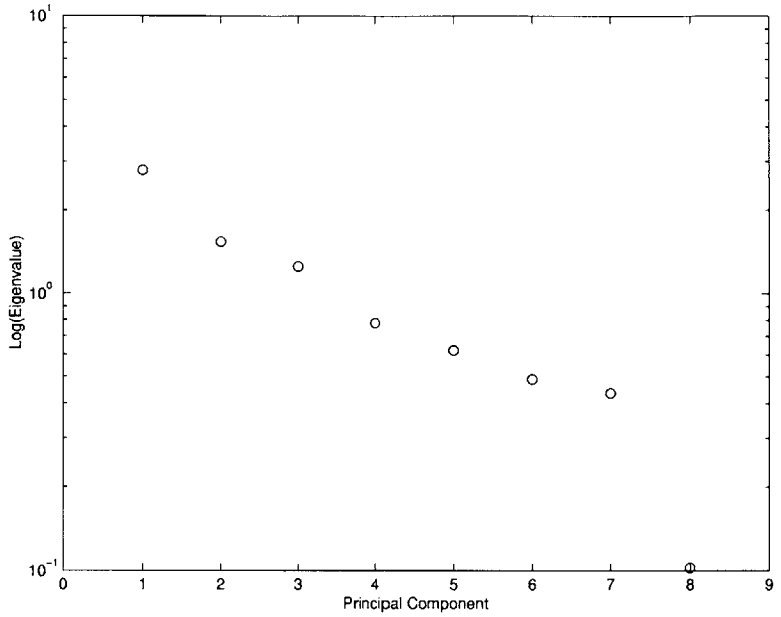

Figure 2-1: Eigenvalues of an 8-channel dataset

2.4 Applying Principal Component Analysis

2.4.1 Application to an N-channel System

In section 2.2, we explored a KL expansion on a random process, x_n. Since a random process is not constrained to a finite length, the number of eigenvectors, eigenvalues, and coefficients in equation 2.18 can be infinite as well. For all practical purposes, let us re-define x_n to be an Nx1 array of random variables. Then the NxN covariance matrix, Λ_x, will produce at most N eigenvalues with N unique eigenvectors.

2.4.2 Application on a Finite Dataset

In the previous sections, we assumed that each random variable was ideal in the sense that their variances and means were well established and known. However, only a finite collection of samples for these random variables will be available. Nonetheless, we can clearly see that using sample covariance/correlation structures does not affect either the properties or definition of PCA. We can simply replace any occurrences of covariance/correlation matrices with their respective sample correlation/covariance matrices[2, p. 24].

2.4.3 Normalizing the Data

In the derivation and discussion of PCA above, we utilized the eigenvalues and eigenvectors based on a covariance matrix. However, we will see that using a correlation matrix is much more practical. Using a correlation matrix is analogous to normalizing the dataset. For instance, let us examine the following equation,

$$\mathbf{z} = \mathbf{\Phi}^T \mathbf{x}^* \tag{2.21}$$

where $x_i^* = x_i/\sigma_{ii}$ and σ_{ii}^2 is the variance of x_i. Then the covariance matrix of $\mathbf{x}^*$ is equivalent to the correlation matrix for $\mathbf{x}$. The following example will attempt to demonstrate the importance of using a correlation matrix over a covariance matrix in calculating the principal components.

Let us briefly examine the Rocketdyne engine dataset. The data contains 323 channels of information with each channel containing over 3200 data points. If we examine table 2.1, we can see that the variances span a wide range. This is a clear indication that the principal components under a covariance matrix will be significantly different from the principal components under a correlation matrix. In fact, in figure 2-2, we can see that the first eigenvalue is clearly orders of magnitude greater than the next eigenvalue. Such a distorted distribution of eigenvalues will produce principal components in which the first PC will contain the majority of the variance of the system. In our case, the first PC contains over 99% of the variance. Obviously,

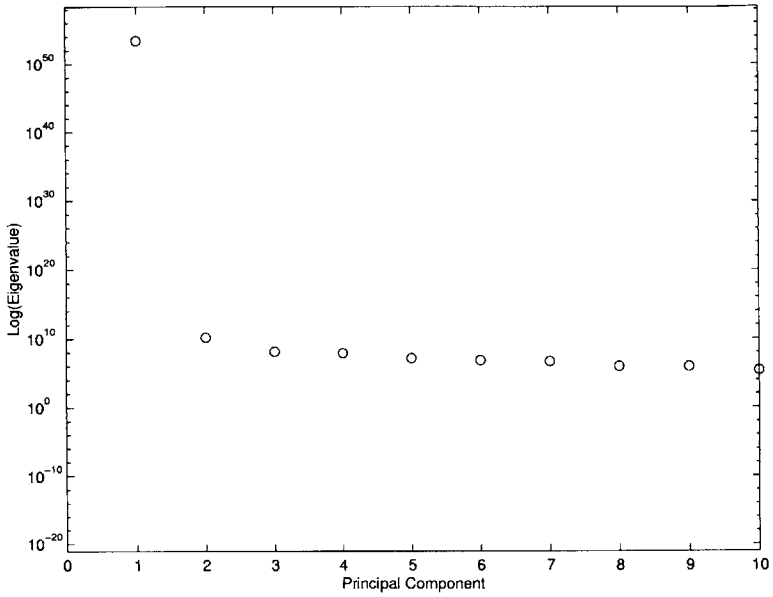

Figure 2-2: Subset of eigenvalues of covariance matrix for engine data (log scale)

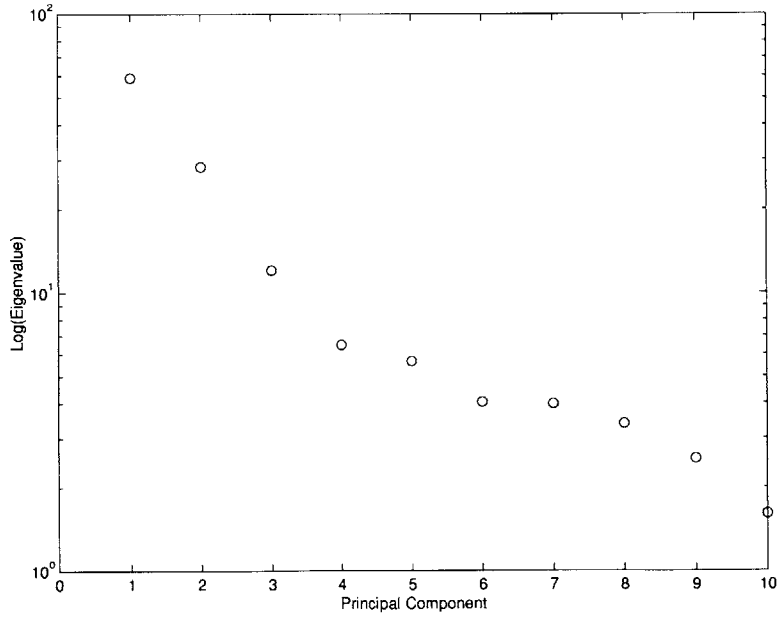

Figure 2-3: Subset of eigenvalues of correlation matrix for engine data (log scale)

such a transformation of variables is not very useful. If we limit our observations to this single PC, we will be tracking the lone variable in table 2.1 with a standard deviation of $5.046 * 10^{26}$. In other words, the PC's will essentially be the original variables placed in a descending order according to their variances. On the other hand, if we use a correlation matrix, the eigenvalues will be distributed in a more reasonable and useful manner as illustrated in figure 2-3. If we examine table 2.3, it is clear through the principal component coefficients and the eigenvalues that information is distributed in a more effective and useful manner. The latter example is an illustration of the dangers in using a covariance matrix for principal component transforms when dealing with channels that have widely varying variances.

The main argument for using a correlation matrix for PCA is based on the preposition that most datasets contain variables that are measured in different units. Such was the case with the Rocketdyne engine dataset. When we are exploring the relationship between the channels, we must normalize the dataset in order to extract principal components that capture these relationships accurately.

Table 2.1: Covariance and standard deviations for subset of Rocketdyne engine data

Channel Name	Time	Timetag	72B	399B	24BA
Time (secs)	$2.546 * 10^{53}$				
Timetag (msecs)	$1.677 * 10^{29}$	$1.737 * 10^{7}$			
72B (Rankine)	$6.922 * 10^{26}$	$9.305 * 10^{4}$	$5.136 * 10^{2}$		
399B (Rankine)	$3.855 * 10^{27}$	$3.320 * 10^{5}$	$1.718 * 10^{3}$	$1.107 * 10^{4}$	
24BA (PSIA)	$1.692 * 10^{27}$	$4.572 * 10^{5}$	$2.750 * 10^{3}$	$5.887 * 10^{3}$	$4.250 * 10^{4}$
Standard deviation	$5.046 * 10^{26}$	$4.168 * 10^{3}$	22.662	105.213	206.15

Table 2.2: Eigenvectors based on covariance matrix for engine data

	Principal Component Number			
Channel Name	1	2	3	4
Time (secs)	1.00	0.00	0.00	0.00
Timetag (msecs)	0.00	0.00	0.302	0.93
72B (Rankine)	0.00	0.00	0.00	0.00
399B (Rankine)	0.00	0.71	0.00	0.00
24BA (PSIA)	0.00	0.00	0.01	0.01
Percentage of total variance	99	0	0	0

Table 2.3: Eigenvectors based on correlation matrix for engine data

	Principal Component Number			
Channel Name	1	2	3	4
Time (secs)	0.01	0.02	0.02	0.03
Timetag (msecs)	0.10	0.09	0.08	0.08
72B (Rankine)	0.10	0.07	0.08	0.12
399B (Rankine)	0.05	0.12	0.15	0.12
24BA (PSIA)	0.12	0.07	0.04	0.03
Percentage of total variance	42.1	20.2	8.6	4.6

Chapter 3

Beacon Exception Analysis Method

3.1 General Outline of BEAM Algorithm

The following will provide an abstract outline of the BEAM algorithm. The outline describes the basic structure of the process while section 3.2 will provide the reader with a detailed derivation of BEAM.

1. Obtain new observation.
2. Update covariance matrix.
3. Subtract the previous covariance matrix from the current covariance matrix to obtain the difference covariance matrix.
4. Compute scalar value from delta covariance matrix.
5. If scalar value is greater than threshold, go to step 6, else go to step 1.
6. Signal system-wide change, dump prior data and go to step 1.

With the general structure of the algorithm in place, let us now explore the in detail the mathematical foundations that underlie the process.

3.2 Mathematical Foundations

3.2.1 Real-time Correlation Matrix

Consider the observation matrix, $\mathbf{X}$, that is defined as

$$\mathbf{X} \equiv \begin{bmatrix} x_1[1] & x_2[1] & \cdots & x_N[1] \\ x_1[2] & x_2[2] & \cdots & x_N[2] \\ \vdots & \vdots & \ddots & \vdots \\ x_1[t] & x_2[t] & \cdots & x_N[t] \end{bmatrix} \tag{3.1}$$

where there are t observations of N variables. Let m_x be an array of averages of the channels defined as

$$\mathbf{m_x} \equiv \begin{bmatrix} m_{x_1} \\ m_{x_2} \\ \vdots \\ m_{x_N} \end{bmatrix}. \tag{3.2}$$

If we let $\mathbf{R}$ be the NxN covariance matrix, the covariance between variables x_i and x_j after t observations can be defined as

$$R_{ij} = \frac{\sum_{k=1}^{t} \sum_{l=1}^{t} (x_i[k] - m_{x_i}) * (x_j[l] - m_{x_j})}{t-1}.^{1} \tag{3.3}$$

The $[i, j]$ element of $\mathbf{C}_t$, the correlation matrix after t samples, can then be defined as

$$C_{ij} = \frac{|R_{ij}|}{\sqrt{R_{ii} * R_{jj}}}.^{2} \tag{3.4}$$

The correlation matrix is updated after each new observation. However, to avoid any unnecessary computations, we can utilize previous values of the cross products. Specifically, if we define $\mathbf{K}_t$, the cross product matrix after t observations, to be as

[1]The t-1 divisor is used for an unbiased covariance structure

[2]The absolute value is used since we only care about the absolute relationship between two variables.

follows

$$\mathbf{K}_t = \mathbf{X}^T\mathbf{X}, \tag{3.5}$$

then the covariance matrix can be defined as

$$\mathbf{R} = \frac{\mathbf{K}_t}{t-1} - \mathbf{m}_x\mathbf{m}_x^T * \left(\frac{t}{t-1}\right). \tag{3.6}$$

After the $t+1$ sample is observed for all N variables, the cross product matrix can be updated via the equation

$$\mathbf{K}_{t+1} = \mathbf{K}_t + \mathbf{X}_{t+1}^T\mathbf{X}_{t+1} \tag{3.7}$$

where $\mathbf{X}_{t+1}$ is the $t+1$ row of observation matrix $\mathbf{X}$, or specifically

$$\mathbf{X}_{t+1} \equiv \left[\begin{array}{cccc} x_1[t+1] & x_2[t+1] & \cdots & x_N[t+1] \end{array}\right]. \tag{3.8}$$

Notice that using equation 3.7 eliminates the need to recalculate the cross product of the entire observation matrix, $\mathbf{X}$, after every new observation. Instead, we can obtain the current cross product matrix, $\mathbf{K}_{t+1}$, by updating the earlier cross product matrix, $\mathbf{K}_t$, with a cross product of a 1xN vector, $\mathbf{X}_{t+1}$. Then by using equations 3.6 and 3.4, we can obtain a real-time correlation matrix that is updated after every observation with minimal computation.

3.2.2 Difference Matrix and Scalar Measurement

We are presented with a new correlation matrix after each new observation. BEAM tracks the changes in the correlation matrix after each sample through a difference matrix and signals a system-wide change if there is an instantaneous change of sufficient magnitude in the difference matrix. We can capture a change in the correlation matrix through the difference matrix, D, that is defined as

$$\mathbf{D} = \mathrm{C}_{t+1} - \mathbf{C}_t \tag{3.9}$$

where $\mathbf{C}_t$ is the correlation matrix as described in equation 3.4 after t observations and $\mathbf{C}_{t+1}$ is the correlation matrix after $t+1$ observations. Attempting to observe the difference matrix for system-wide changes presents us with the original problem of tracking too much information. However, BEAM quantifies the difference matrix to a single scalar, γ, through

$$\gamma = \frac{\sum_{i=1}^{N} \sum_{j=1}^{N} |D_{ij}|}{N^2} \tag{3.10}$$

where D_{ij} is the $[i, j]$ element of the difference matrix. One should note that BEAM tracks the absolute value of D_{ij} since we care only about the absolute change in the system. Also notice that the scalar value is normalized by N^2 so as to measure the change per channel. This single scalar value, as opposed to an N-dimensional system, can now be used as a tool to track system-wide changes.

3.2.3 Dynamic Threshold

While the formulation of the scalar measurement is critical to the BEAM process, the derivation of the threshold to which γ is measured against is just as important. BEAM uses a dynamic threshold that adapts to the trends of γ. A system-wide change is defined to occur at time t when γ is greater than the "3σ" value. After t observations, let γ be stored in a time-indexed vector as

$$\gamma \equiv \begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \vdots \\ \gamma_{t-1} \end{bmatrix}. \tag{3.11}$$

The characteristics of this time-indexed array can now be used to create σ, a threshold computed via

$$\sigma = \frac{2}{M} * \sum_{i=\lfloor t-\frac{M}{2}+1 \rfloor}^{t-1} \gamma_i \tag{3.12}$$

where M is the number of samples observed since the last system-wide change. Equation 3.12 shows that σ is a running average of the γ's taken over an expanding time window. As more samples are introduced (as both t and M increase), the window expands linearly at the rate of $M/2$. If γ exceeds 3σ, BEAM signals a system-wide change and reinitializes to its original state as described in section 3.2.4.

3.2.4 Effects of System-wide Changes on BEAM

When an aircraft or spacecraft enters different stages of its flight, the inherent relationship among channels of information often enter new modes themselves. For instance, the characteristics of an aircraft change significantly under take-off, cruise, and landing modes. Therefore, it is practical to treat system-wide changes as points of initialization so as to capture the new relationships among the channels.

While certain channels of information might be uncorrelated prior to a system-wide change, these very same channels might be highly correlated after entering a new mode. Thus, previous data might in fact corrupt current calculations. For example, let us assume that channels A and B have little or no correlation before the 1000th sample and are highly correlated after the 1001st sample. A problem arises when a system-wide change occurs at the 1001st sample. Then due to both the initial lack of data after the change and the dominance of the 1000 previous samples in the calculations, channels A and B will remain uncorrelated for a significant number of samples. BEAM, however, bypasses this problem by returning to its initial state after a system-wide change is detected. Specifically, given that a change occurs at observation t, all the previous data before observation t is ignored, in effect dumping prior values of the correlation matrix and difference matrix. As a result, a new correlation matrix, difference matrix, and γ are rebuilt starting at observation $t + 1$.

3.2.5 Channel Contribution and Coherence Visualization

Channel Contribution: After BEAM signals that a system-wide change occurred, it is often useful to understand which channels contributed most to this event. The

BEAM algorithm maintains an N-dimensional array, p, that contains, in percentage terms, the contributions of the N channels. If an event occurs at time t, we can compute the contribution of the i^{th} channel via

$$p(i) = \frac{\left|\sum_{j=1}^{N} D_{ij}\right|}{\sum_{j=1}^{N}\sum_{k=1}^{N} |D_{jk}|} \tag{3.13}$$

where $\mathbf{D}$ is the difference matrix at time t. Since the difference matrix is diagonally symmetric, $p(i)$ will never be greater than 50%. For example, if channel i changes such that the difference matrix only has non-zero entries at row i and column i, then $p(i)$ will be exactly 50%. However, we care only about the relative contribution of the channel, i.e., p(i) vs. p(j), rather than it's absolute value. Therefore, normalizing the contributions to 100% by doubling p(i) does not change our analysis.

Coherence Visualization: While the latter formulation is quite useful when utilizing numerical analysis, it is also useful to provide a graphical representation of both system-wide changes and channel contribution. A natural way to offer such a visualization is to provide a three-dimensional plot of the difference matrix. The plot's third dimension, magnitude, could consist of either varying colors or heights. Figures 3-1 and 3-2 illustrate a sample difference matrix taken from a subset of Cassini's AACS dataset. Figure 3-1 utilizes a varying color scheme as the third dimension, where lighter colors reflect larger values, and figure 3-2 is a 3-dimensional representation of the same data. We can see that the latter figures represent a change due to a single channel. Figures 3-3 and 3-4 represent an event where changes in correlations among multiple channels contributed to the event.

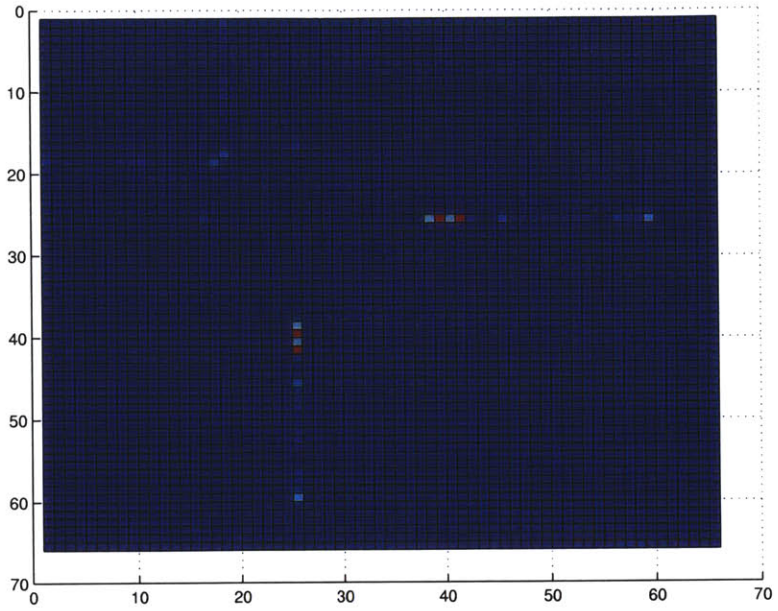

Figure 3-1: Surface plot of a difference matrix just after an event in the AACS.

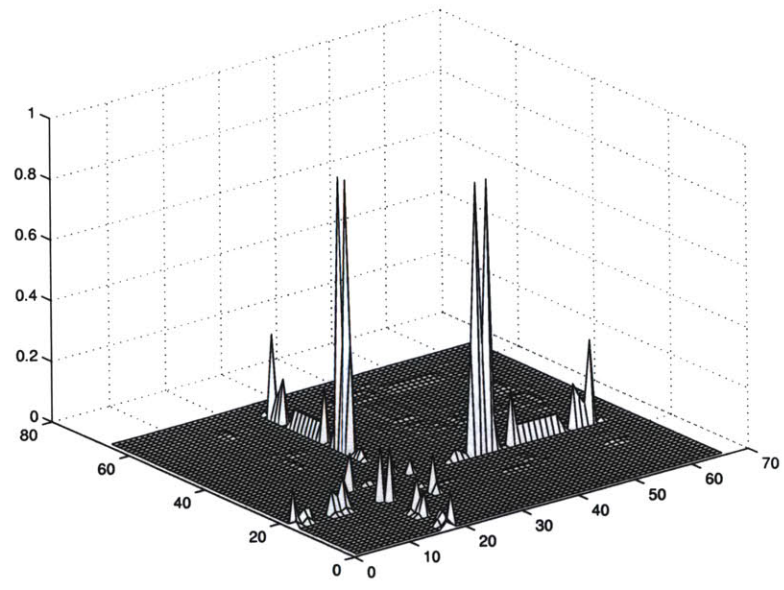

Figure 3-2: 3-Dimensional surface plot of figure 3-1

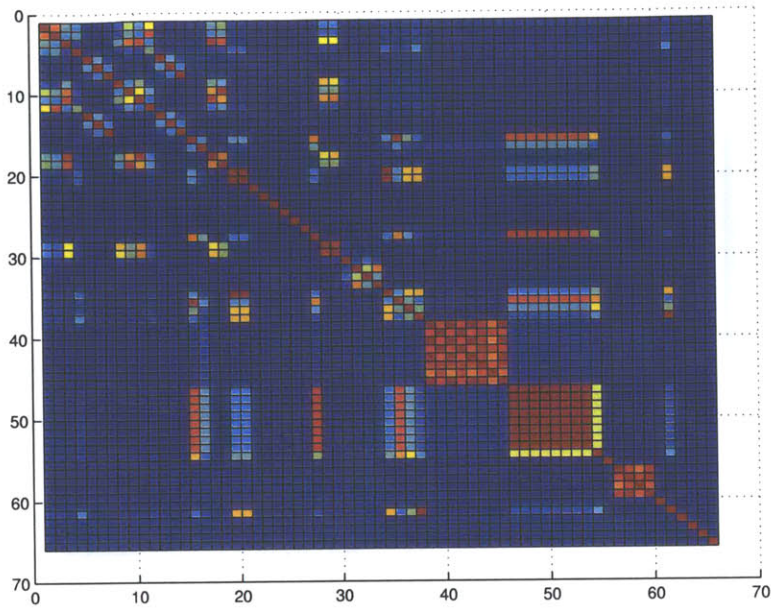

Figure 3-3: Surface plot of a difference matrix with multi-channel contribution.

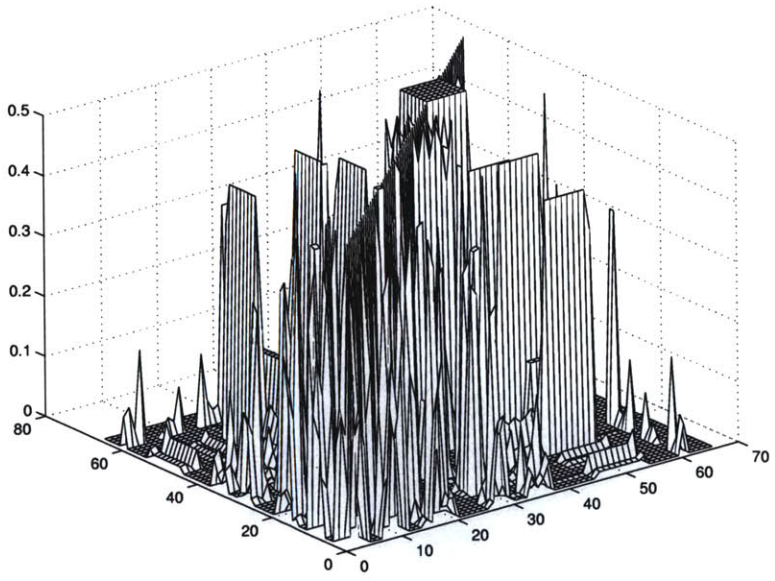

Figure 3-4: 3-Dimensional surface plot of figure 3-3

3.2.6 Real-Time Implementation and its Consequences

BEAM was designed to be implemented as a real-time measurement vehicle. Issues such as causality and minimum observation requirements play a key role in determining the effectiveness of BEAM as a real-time process. The following will describe in detail how the process is effected by these issues.

Determining the value of buffer: The minimum number of samples required to produce a valid correlation matrix is determined by the number of channels, i.e., an N channel system would require approximately N or more samples. Any correlation matrix produced with a smaller sample size is often considered to be inaccurate. On the other hand, if we limit our initial sample space to be at least the dimension of our channels, then we will lose the ability to track any system-wide changes during this period. Therefore, we are presented with the problem of balancing precision and sensitivity. While this problem might at first appear to only affect the process in the beginning when observations are first being incorporated, section 3.2.4 clearly indicates that this is an issue after *every* system-wide change. BEAM attempts to resolve this issue by setting the *buffer* [3] as a relatively small constant and compensating for inaccurate correlation values with a higher threshold in σ.

In figure 3-5 and in its accompanying algorithm in figure 3-6, *buffer* is set at a fixed value of 3. The function *reset(x)* simply clears the memory buffer associated with variable x, and the function *signalevent()* records that an event, or system-wide change, has occurred. At first, such a small value for *buffer* could be viewed as producing unreliable correlation matrices, and rightfully so. However, analyses in chapter 4 will reveal that for the Rocketdyne engine data, such a small buffer does not produce any false alarms. Particularly, the scalar measurement tool, γ, is a function of the "derivative" of the correlation matrix. In turn, large fluctuations in γ will produce a higher threshold, σ. Therefore, the system is able to compensate for the large variance in γ during the early stages of data acquisition. Later, as more samples

[3] The *buffer* is defined as the minimum number of samples BEAM must observe before any decision is made.

are introduced, the correlation matrix becomes more stable, i.e., the changes in $\mathbf{C}$ from sample to sample become smaller. The difference matrix, and thus γ, approach smaller values. Consequently, the system compensates for the smaller variance in γ with a smaller threshold in σ.

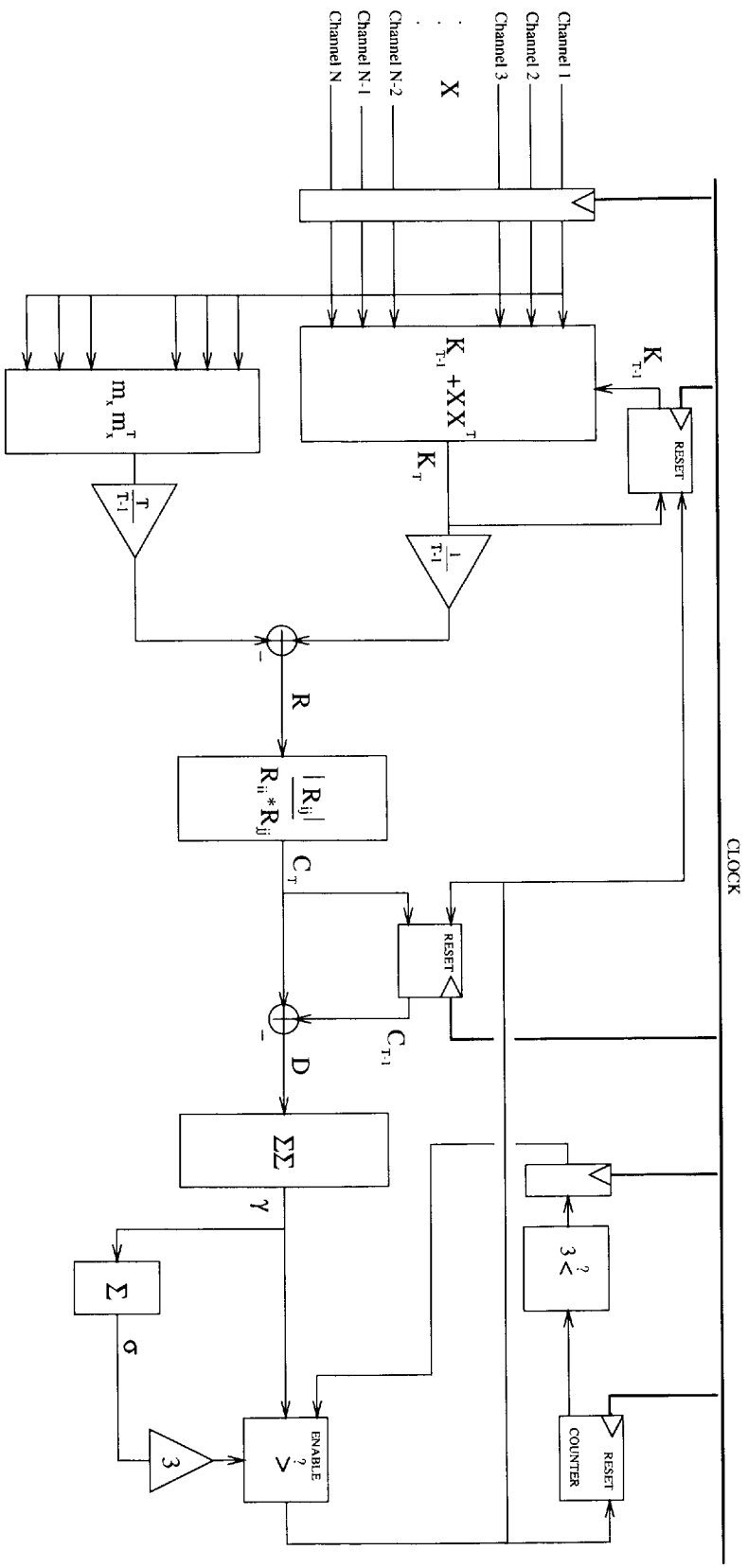

Figure 3-5: A flow chart of the BEAM algorithm

```
T = 0;
buffer = 3;
for i = 1 : m
    T = T + 1;
    for j = 1 : n
        m_x(j) = ((T-1) * m_x(j) + X(i,j))/T;
    end
    for j = 1 : n
        for k = 1 : n
            K(j,k) = K(j,k) + X(i,j) * X(i,k);
        end
    end
    for j = 1 : n
        for k = 1 : n
            R(j,k) = K(j,k)/(T-1) - m_x(j) * m_x(k) * T/(T-1)
        end
    end
    for j = 1 : n
        for k = 1 : n
            C(j,k) = abs(R(j,k))/(R(j,j)*R(k,k));
        end
    end
    D = abs(C_T − C_{T−1});
    for j = 1 : n
        temp = 0;
        for k = 1 : n
            temp = temp + D(j,k);
        end
        Γ(i) = Γ(i)+abs(temp);
    end
    if T > buffer
        for j = ⌊ i - T/2 + 1 ⌋ : i-1
            σ = σ + Γ(j)/T;
        end
        if Γ(i) > 3 ∗ σ
            T = 0;
            signalevent();
            reset(m_x);
            reset(K);
        end
    end
    C_{T−1} = C_T
end
```

Figure 3-6: BEAM algorithm

Chapter 4

Analysis of the BEAM System

The following chapter will discuss the effectiveness of the BEAM algorithm. We utilized an experimental dataset to illustrate the capabilities of BEAM as compared to those of Principal Component Analysis. Furthermore, given that test request documentation was provided with the experimental dataset, an explicit benchmark existed for comparing the accuracy of both the BEAM and PCA algorithms.

4.1 Sample Dataset

The sample dataset was provided by Rocketdyne as part of an agreement with JPL in the development of the Linear Aerospike Engine for NASA's X-33 Reusable Launch Vehicle (RLV) program. The dataset consists of 323 channels of information from the Power Pack Assembly (PPA) of the Linear Aerospike Engine. The goal of test number A1X003 was to demonstrate the PPA performance and operation stability at 80 percent equivalent engine power level. While the engine firing lasted for 36 seconds, the dataset at hand contains over 5634 seconds of information with each channel sampling at 100 Hz. That equates to approximately 563,400 data points per channel. However, since the majority of that information contains little or no information due to engine inactivity, the data that was actually used for the performance benchmark experiment contained 47.01 seconds of information, or 4701 data points per channel. The data begins at "startup", or the beginning of the engine firing, and ends at

"postshutdown", approximately 11 seconds past engine shutdown[4].

The A1X003 dataset provides a large and diverse set of sensor data. For example, the type of channels of information range from an oxygen pump shaft speed sensor to a hydrogen inlet turbine temperature meter. Such a diverse and large dataset is an excellent test bench for evaluating the effectiveness of BEAM. As mentioned in chapter 3, some of the main goals of BEAM are to simplify diagnostics of system-wide faults and to create an abstract measurement tool to track a complex system. For all practical purposes, all references to data and experimental results refer to the A1X003 dataset for the remainder of the chapter.

4.2 Rules of Measurement

Before we proceed, we must clearly define the criteria for measuring BEAM's performance. While we can investigate many areas of performance, the following section will investigate the two most critical rules of measurement: capturing system-wide events and ranking the channel contribution surrounding those events.

4.2.1 System-wide Events

Computing system-wide events under PCA: The most obvious and practical method for detecting system-wide events under PCA is to track the n most significant principal components and their derivatives. If one of the derivatives exceeds some threshold at time t, then we can signal that an event has occurred at time t. This threshold is preset by the user and customized for each principal component so as to minimize false alarm rate while maintaining a minimum level of sensitivity. Figure 4-1 illustrates the first two principal components, their absolute differences, and their respective thresholds that were used to detect events.

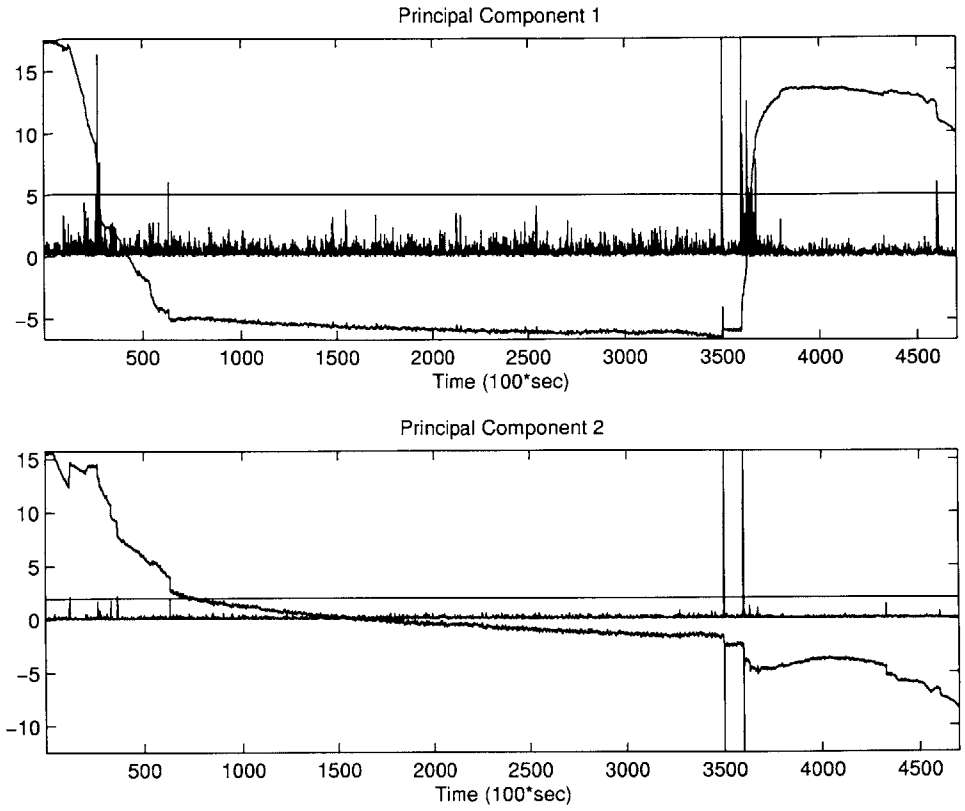

Figure 4-1: Plot of the first 2 PC's, their differences, and thresholds

The n significant number of principal components can be computed by observing the log-scale eigenvalue plot as described in section 2.3. If we examine figure 4-2, we can see that analyzing the first seven principal components will suffice since there is a sharp drop off in the eighth eigenvalue. In order to utilize all seven principal components in detecting events, a simple process called *pcaevents()* is used [see Appendix B, Section II]. The function computes system-wide events for each principal component and then compares the timing of these events across all seven principal components. For example, if the first principal component detects an event at t milliseconds and the second principal component detects an event at $t+5$ milliseconds, then the process recognizes that only a single event occurred at time t. The incremental time that must be exceeded to record a new event is an arbitrary value preset by the user. In the A1X003 dataset, it is fixed at 50 samples, or 500 milliseconds.

Computing system-wide events under BEAM: Section 3.2 outlined the process by

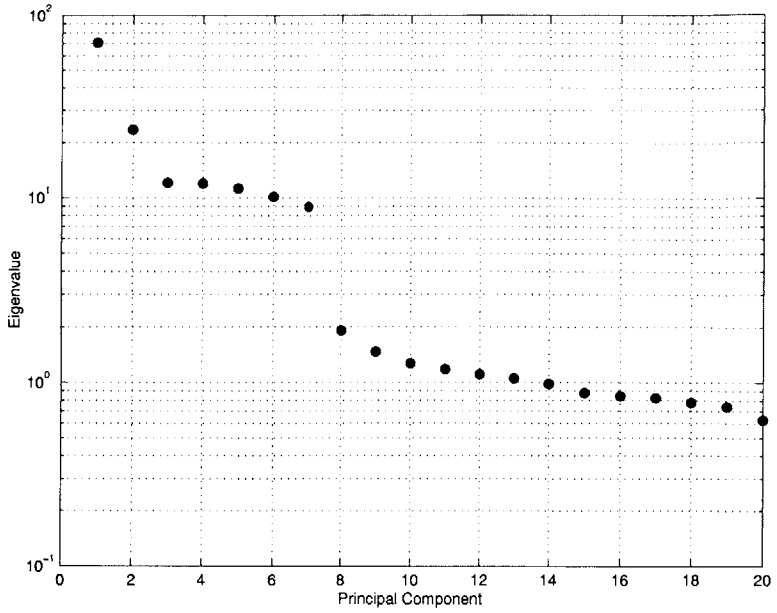

Figure 4-2: Eigenvalues of A1X003 in Log-scale

which BEAM detects a system-wide event. Figure 4-3 illustrates the progressive visual output of BEAM when analyzing the A1X003 dataset. Note that the middle figure lacks a threshold "bar" since the process is within the *buffer*. Furthermore, note that the abnormally large "spike" in the third plot of figure 4-3 reflects a re-initialization where the correlation matrix is computed anew. After a system-wide event occurs and BEAM re-initializes it's variables, the first difference matrix is equivalent to the first correlation matrix since a previous correlation matrix does not exist. Figure 4-4 is the final visual output of γ.

Comparing the two results: A clear way to compare the two processes' abilities to track system-wide events is to examine which events are common to both BEAM and PCA and which events are unique to BEAM and PCA. A visual representation, such as the one in figure 4-5, provides an excellent means by which we can compare the outcomes of the two processes. Markers along the 45 degree line denote an event common to both BEAM and PCA. Markers on the x-axis denote events unique to BEAM, and markers on the y-axis denote events unique to PCA. As we can see, events under PCA is a subset of the events under BEAM. The events at 2050 and 38,010 milliseconds were detected by BEAM but not by the PCA method. If we attempt

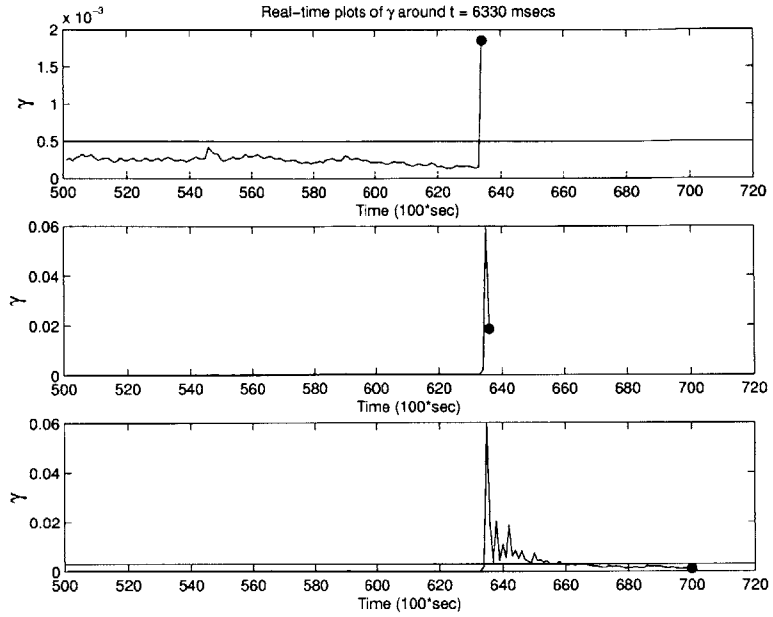

Figure 4-3: Real-time plots of γ for the A1X003 dataset

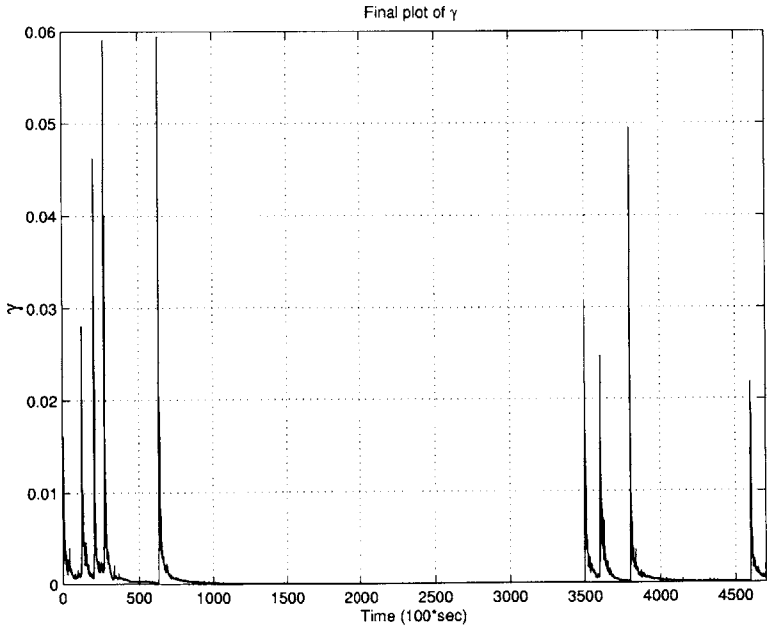

Figure 4-4: Final plot of γ for the A1X003 dataset

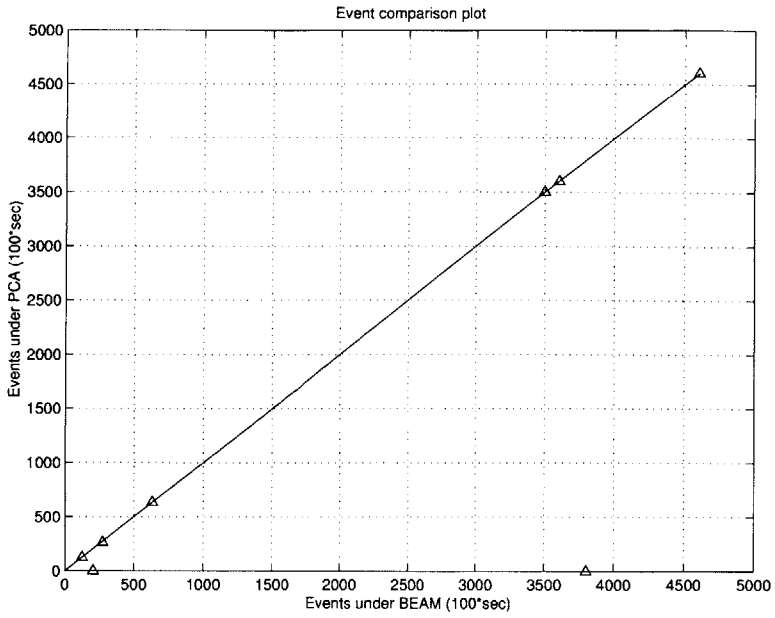

Figure 4-5: BEAM vs. PCA in detecting events for the A1X003 dataset

to capture these events under PCA by lowering the thresholds, we begin to detect events that cannot be verified through the test request documentation. For example, if we lower the threshold for the first principal component, the first new event that is captured occurs at 25,440 milliseconds. In addition, if we lower the threshold in the second principal component, the first new event occurs at 43,280 milliseconds. The test request documentation cannot account for either of these events. Therefore, we can state that there is no benefit to using PCA over BEAM in detecting system-wide events that are accountable through the test request documentation. After observing figure 4-5, we can see that the superset of system-wide events occurred at 1250, 2050, 2730, 6330, 34,980, 36,010, 38,010, and 46,010 milliseconds. Section 4.3 will explore the accuracy of these detections when compared with the test request documentation.

4.2.2 Channel Localization and Contribution

Upon detecting a system-wide event, we must localize the sources of these changes. In a 323 channel system, as in the case of the A1X003 dataset, localizing the possible sources to 10 channels is a fairly efficient reduction in the observable telemetry. The latter equates to a 96.9% reduction in the number of channels one needs to observe

surrounding a specific system-wide event. The following will discuss the methods that were used to localize the sources of events for both PCA and BEAM. Furthermore, in section 4.3, we will compare how effective these localizations were when compared to the information in the test request documentation. But before we proceed, it is important to note that we used the BEAM events as the global list of events for computing channel contribution under both BEAM and PCA. A concern arises when we compute channel contribution around an event unique to BEAM; the calculations under PCA might be unreliable due to the fact that PCA had not signaled an event at that time. We will illustrate this further through an example in section 4.3. Note, however, that the opposite scenario does not exist due to the fact that BEAM events are a superset of PCA events. Unless otherwise noted, we will examine in detail the channel contributions around the system-wide event at 6330 milliseconds for the remainder of this section[1].

Computing channel contribution under PCA: We developed three methods for calculating channel contribution under PCA. The first method is to simply use the coefficients in the eigenvector of the correlation matrix. For instance, if an event is captured by the n^{th} principal component, then we simply sort the elements of $|\phi_n|$ where ϕ_n is the n^{th} eigenvector. A serious problem arises when we utilize this method. Since we are often limited to observing the first few significant principal components, we limit our number of possible unique channel contributions. For example, if the first principal component captures three different system-wide events, then all three events will list the same channel contributions because we would use the first eigenvector's coefficients as the channel contributions for all three events.

The second method, which we will refer to as the 1^{st} *order* method, simply uses the derivative of the first order principal component. If we let $\mathbf{x}_n$ be the n^{th} channel, i.e., the n^{th} column in matrix $\mathbf{X}$ of equation 3.1, we can define the difference of $\mathbf{x}_n$ as

$$x'_n[t] = x_n[t+1] - x_n[t]. \tag{4.1}$$

[1]This event is common to both PCA and BEAM, i.e. it lies along the 45 degree line in figure 4-5

Then due to linearity, we can compute the difference of the n^{th} principal component via

$$\mathbf{y'}_n = {\phi_n}^T * \mathbf{x'}_n \tag{4.2}$$

where ϕ_n is the n^{th} eigenvector of the correlation matrix for $\mathbf{x}$. If, for example, the first principal component captured an event at time t, i.e., a sharp impulse above threshold in the derivative of the first principal component, we can calculate how much each channel contributed to that impulse. Let us define $\mathbf{c}$ as

$$\mathbf{c} = \left[\begin{array}{cccc} {x'}_1[t] * \phi_1[1] & {x'}_2[t] * \phi_1[2] & \cdots & {x'}_N[t] * \phi_1[N] \end{array} \right], \tag{4.3}$$

where c[n] is channel n's contribution to the impulse at time t.[2] In other words,

$$y'_1[t] = \sum_{n=1}^{N} c[n]. \tag{4.4}$$

The latter equation simply states that if we sum vector $\mathbf{c}$, we obtain the value of the difference of the first principal component at time t. We can rank the elements of $\mathbf{c}$ after taking the sign of the impulse into account. If the impulse is negative, we would place the greatest significance on the most negative element of $\mathbf{c}$.

The third and final method, which we will refer to as the 2^{nd} *order window* method, uses windowed principal components to perform the 1^{st} *order* method. Given that we have already computed the principal components on the entire dataset, we would use a subset of these principal components to compute a new set of principal components. Let 2Δ be the length of the window and $\mathbf{y}_n$ be the n^{th} principal component. In addition, let us say that we want to calculate the channel contribution for an event

[2] In equation 4.3, we need to use the principal component that is responsible for capturing the event. For example, if the event under investigation was captured by the third principal component, we would replace $\phi_1[i]$ with $\phi_3[i]$.

at time t. Then if we define $\mathbf{W}$ as

$$\equiv \begin{bmatrix} w_1[1] & w_2[1] & \cdots & w_N[1] \\ w_1[2] & w_2[2] & \cdots & w_N[2] \\ \vdots & \vdots & \ddots & \vdots \\ w_1[2\Delta+1] & w_2[2\Delta+1] & \cdots & w_N[2\Delta+1] \end{bmatrix} \tag{4.5}$$

$$= \begin{bmatrix} y_1[t-\Delta] & y_2[t-\Delta] & y_3[t-\Delta] & \cdots & y_N[t-\Delta] \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ y_1[t] & y_2[t] & y_3[t] & \ddots & y_N[t] \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ y_1[t+\Delta] & y_2[t+\Delta] & y_3[t+\Delta] & \cdots & y_N[t+\Delta] \end{bmatrix}, \tag{4.6}$$

then we would simply replace x_n with w_n in equations 4.1 through 4.3 and compute channel contributions as outlined in the 1^{st} *order* method. There are two opposing factors involved when setting the value of 2Δ. First, we must maintain a reasonable window size in order to provide enough sample points for computing correlation matrices. Second, we want to avoid or minimize the number of peripheral events occurring within the window so as to maximize the influence of the centered event when computing the 2^{nd} order principal components. For the A1X003 dataset, the minimum separation between PCA events was 100 samples. However, convention states that we must choose at least 323 samples to compute a correlation matrix for 323 channels. On the other hand, only about 100 channels are active at a given time when we limit our observations to a window of that magnitude. Therefore, we set $2\Delta = 100$ samples. Figure 4-6 is the histogram of the ten most significant channels surrounding the event at 6330 milliseconds when using the 2^{nd} *order window* method.

Computing channel contribution under BEAM: Upon detecting a system-wide event, BEAM computes in percentage terms how much each channel contributed to that event. Section 3.2.5 outlined in detail the precise method for computing channel contribution. Figure 4-7 illustrates the channel contributions around the event at 6330 milliseconds under BEAM.

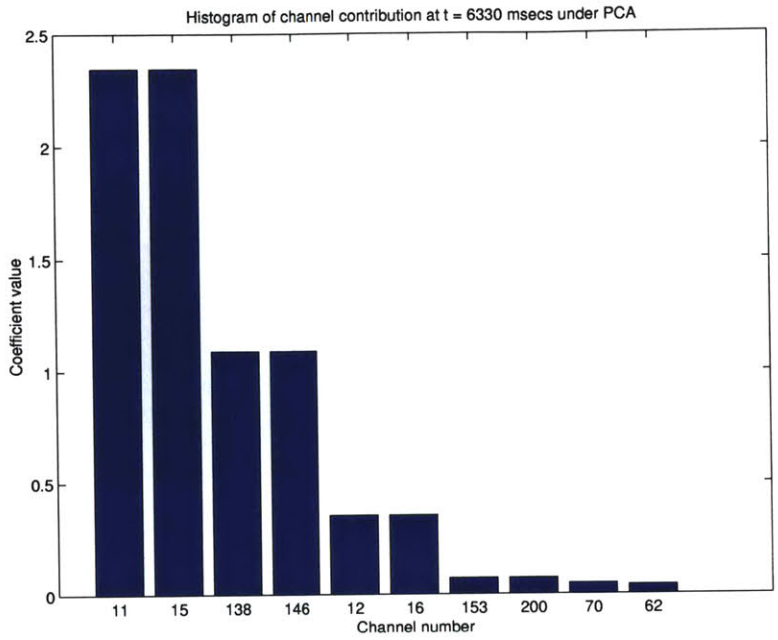

Figure 4-6: Channel contributions under PCA

Comparing the two results: In order to compare the channel contribution calculations from BEAM and PCA, we must look at their relative contributions rather than their absolute contributions. The outputs of BEAM's channel contribution method and PCA's 2^{nd} *order* method are separated by an order of magnitude, i.e., we cannot compare the absolute values in figures 4-6 and 4-7. Therefore, we must compare the relative channel contributions. If we are observing the channel contributions around the event at 6330 milliseconds, how do the ten most important channels under the PCA's 2^{nd} *order* method compare to the ten most important channels under BEAM? Figure 4-8 provides a visual method to compare the relative rankings. The top plot in figure 4-8 is just a normalized superposition of the plots in figures 4-6 and 4-7. In the bottom plot, we are comparing the relative rankings from BEAM and PCA. For instance, the channel that contributed the most to the event at 6330 milliseconds under BEAM is the seventh most important contributor under PCA. If all of the rankings from the two processes are identical, all points would lie along the 45 degree line. While the example in figure 4-8 did not produce a perfect match, the importance lies in the fact that the eight most significant channels are common among BEAM and PCA. Further examination will show that all eights channels have identical charac-

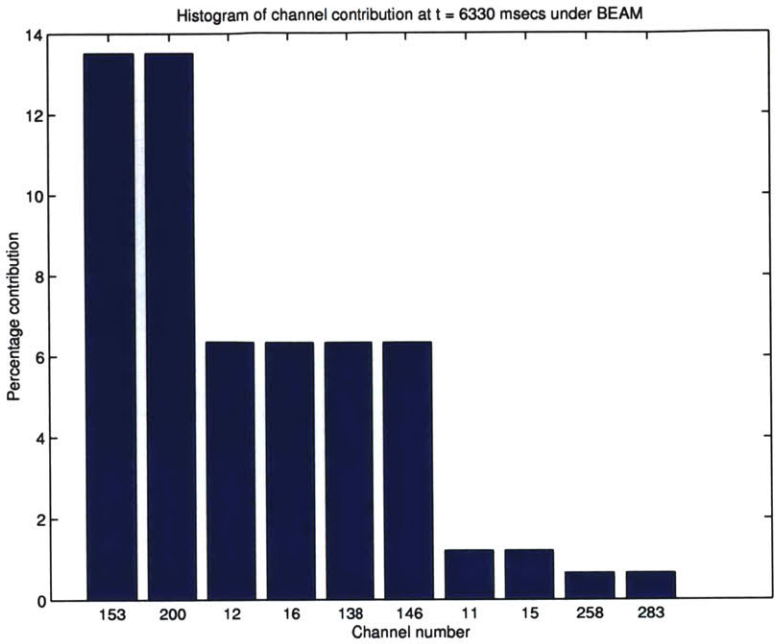

Figure 4-7: Channel contributions under BEAM

teristics around the event at 6330 milliseconds. Figure 4-9 contains the plots of three of the eight channels involved. We can clearly see that all three channels behave identically at 6330 milliseconds. Therefore, while an identical match along the 45 degree line would be ideal, a closer examination of the actual channel data reveals that localizing our attention to a small group of channels will suffice. We only care that both BEAM and PCA were able localize our attention to the *same* eight channels. An exhaustive search among the remaining 315 channels revealed that other channels were barely, if at all, involved in this event. We obtained similar results when we calculated, ranked, and compared channel contributions around the remaining seven events [see Appendix A].

4.3 Test Request Benchmark

Let us now turn to the test request documentation to verify the results from analyses performed on the A1X003 dataset. We pointed out earlier that system-wide events under BEAM were a superset of events under PCA. Therefore, we can come to one of three possible conclusions. First, both BEAM and PCA lacked the sensitivity to

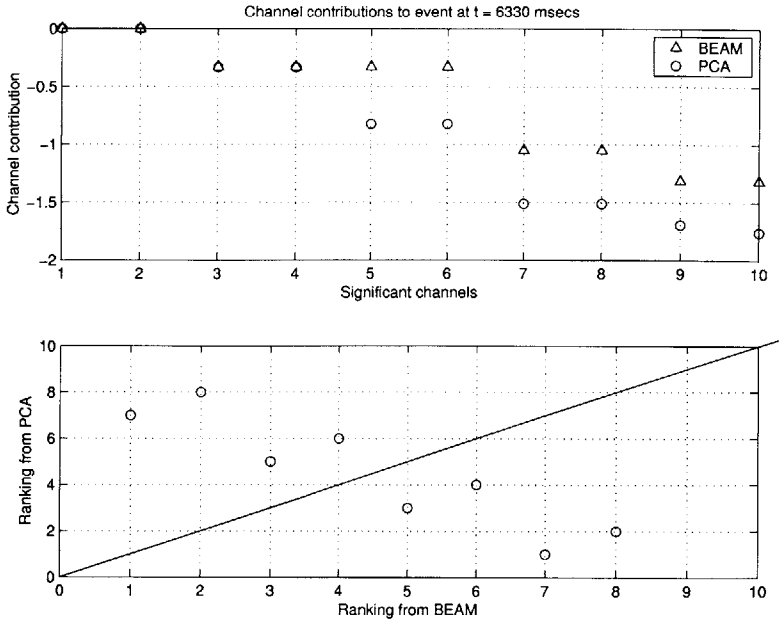

Figure 4-8: Comparison of the 10 most significant channels under BEAM and PCA.

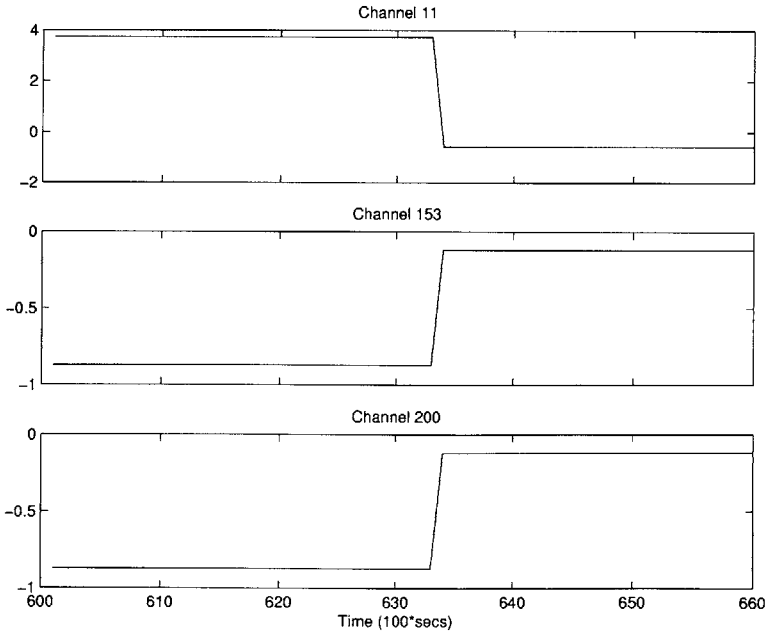

Figure 4-9: Plots of three channels surrounding the event at 6330 msecs.

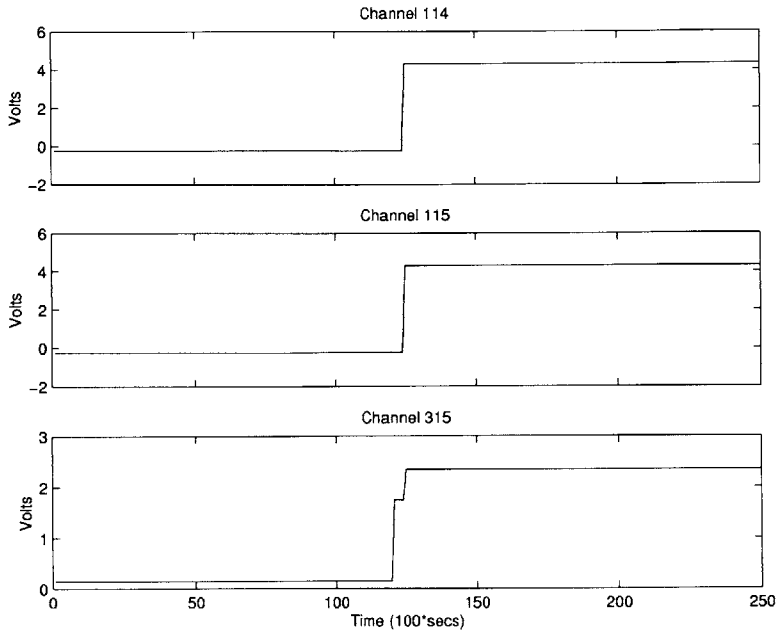

Figure 4-10: Channel outputs around event at 1250 msecs

detect all significant system-wide events. Second, BEAM detected the majority of events with a zero false alarm rate while PCA failed to detect a few critical events. Third, the BEAM process was over-sensitive and declared events during quiet modes, i.e., the events unique to BEAM in figure 4-5 were in fact false alarms. Figures 4-10, 4-11, and 4-12 illustrate some of the significant channels participating in three additional events detected by BEAM. The three channels in figure 4-10 were all among the ten most significant contributors identified by BEAM; the same event was also detected by PCA. Figures 4-11 and 4-12 present dominant channels for events at 1250 and 2050 msecs, both of which were detected by BEAM, but only the second was detected by PCA, perhaps because the ramp illustrated in figure 4-11 was less prominent.

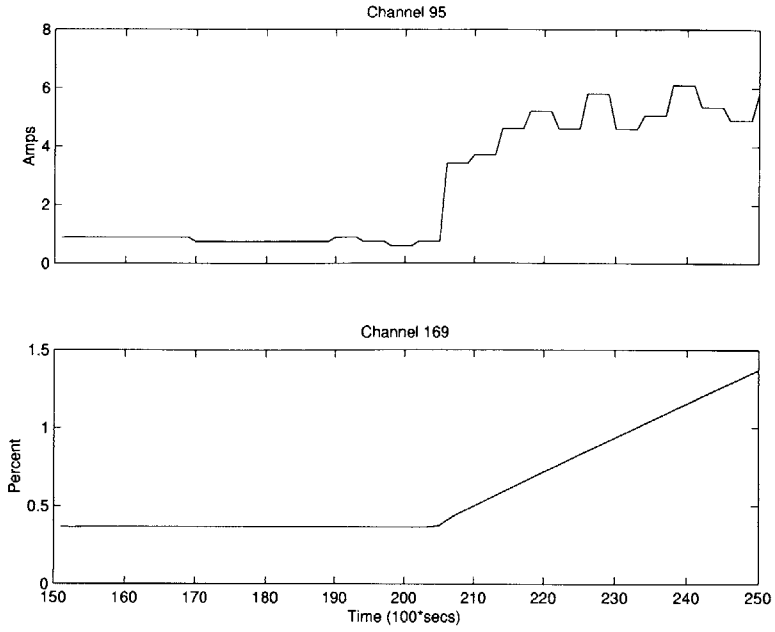

Figure 4-11: Channel outputs around event at 2050 msecs

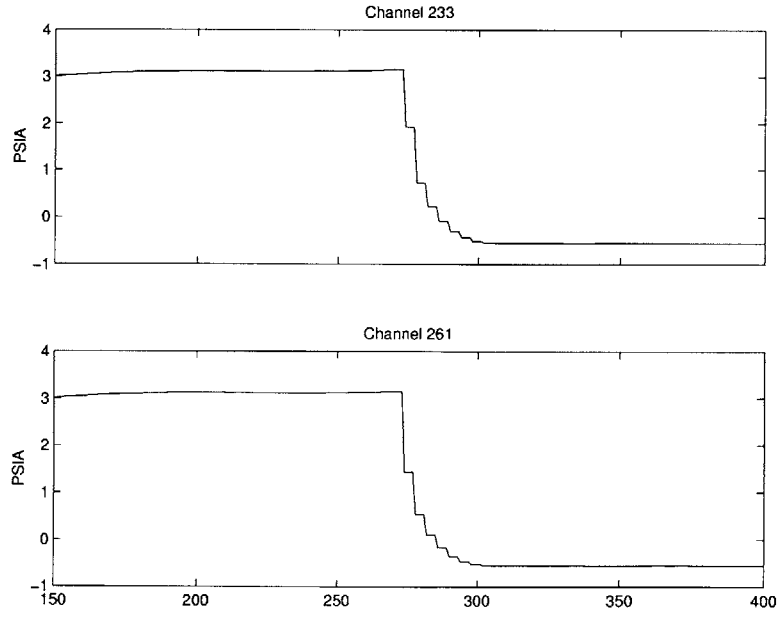

Figure 4-12: Channel outputs around event at 2730 msecs

Chapter 5

Conclusion

The Beacon Exception Analysis Method has proven to be an efficient and effective vehicle for detecting and isolating system-wide changes. However, much of the theoretical work is still under development, and we need to address some of the weaker aspects of the BEAM process. For example, a proper method has yet to be established for calculating a dynamic threshold for γ. Furthermore, a more robust method must be developed for setting *buffer*. One possible method is to implement a dynamic *buffer* that varies according to the activity of γ.

Nevertheless, it was shown that BEAM was superior to Principal Component Analysis in detecting system-wide events and isolating the channels associated with these events. But a more important aspect of BEAM is the ease of real-time implementation. Methods exist for real-time principal component transforms [1]. However, BEAM was designed as a real-time process that requires little incremental computation. The analysis performed with principal component transforms required hours of computation. While we did not explicitly explore an analogous real-time implementation of principal component transforms, it is clear that BEAM has many benefits as a real-time process. On the other hand, recently developed techniques based on principal component transforms, such as the Iterative Order and Noise (ION) estimation algorithm, could prove to be an alternative tool for detecting system-wide changes. The ION algorithm was proven to perform exceptionally well under similar conditions as the BEAM process [3].

In order to develop BEAM into a comprehensive prognostics and diagnostics process, members of the UltraComputing Technologies group at JPL are currently attempting to incorporate wavelet transforms in the prognostics aspect of the PHM. The wavelet transform possesses the quality of presenting the data in a time-frequency domain. If a proper wavelet is chosen, the wavelet transform can capture sharp transitions through lower level details while gradual transitions are best captured within the higher level approximations. Therefore, the wavelet transform has the potential to detect low-level transients that the current BEAM process lacks. But more importantly, information from a wavelet transform analysis, carefully combined with the current implementation of BEAM, can prove to be a powerful tool in predicting system-wide failures. Information from a complete BEAM package that include wavelet transform analysis will be multidimensional, i.e., each event possesses unique characteristics such as decomposition level, approximation or detail, time of event, amplitude of γ, wavelet width, and channel rankings[1]. Therefore, we can utilize multidimensional aspects of each event to predict future events. For example, we could run hours of experiments on simulation data so as to create a comprehensive database of event classifications. We can create a look-up table that points to a library of such events, each categorized according to their multidimensional characteristics. Thus, if certain aspects of the system at some given time produces a "hit" in the look-up table, we signal that a specific event is to occur within some level of probability. If the UCT succeeds in developing such a process, it will prove to be a superior PHM package for *any* system composed of an intricate array of sensors and channels of information.

[1]Approximation and detail refer to the two types of decomposition outputs from a wavelet transform.

Appendix A

Channel Contributions: Experimental Results

The channel coefficients and ranking comparisons are shown for the eight events in the Rocketdyne engine dataset, A1X003. The upper figures are normalized and superimposed graphs of the ten most significant channels' coefficients under BEAM and PCA. The lower figures are one to one comparisons of the channel rankings under BEAM and PCA.

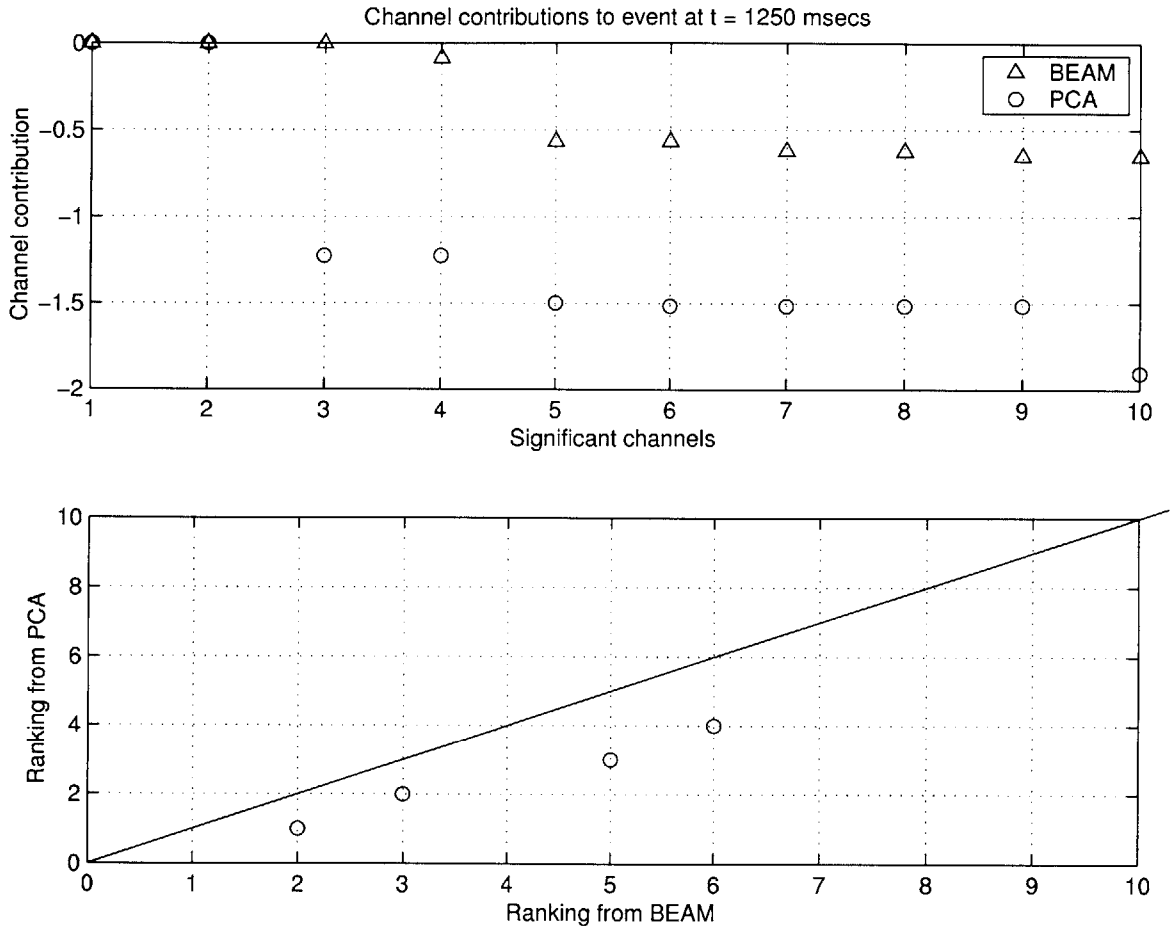

Figure A-1: Channel coefficient and ranking comparisons around event at 1250 msecs.

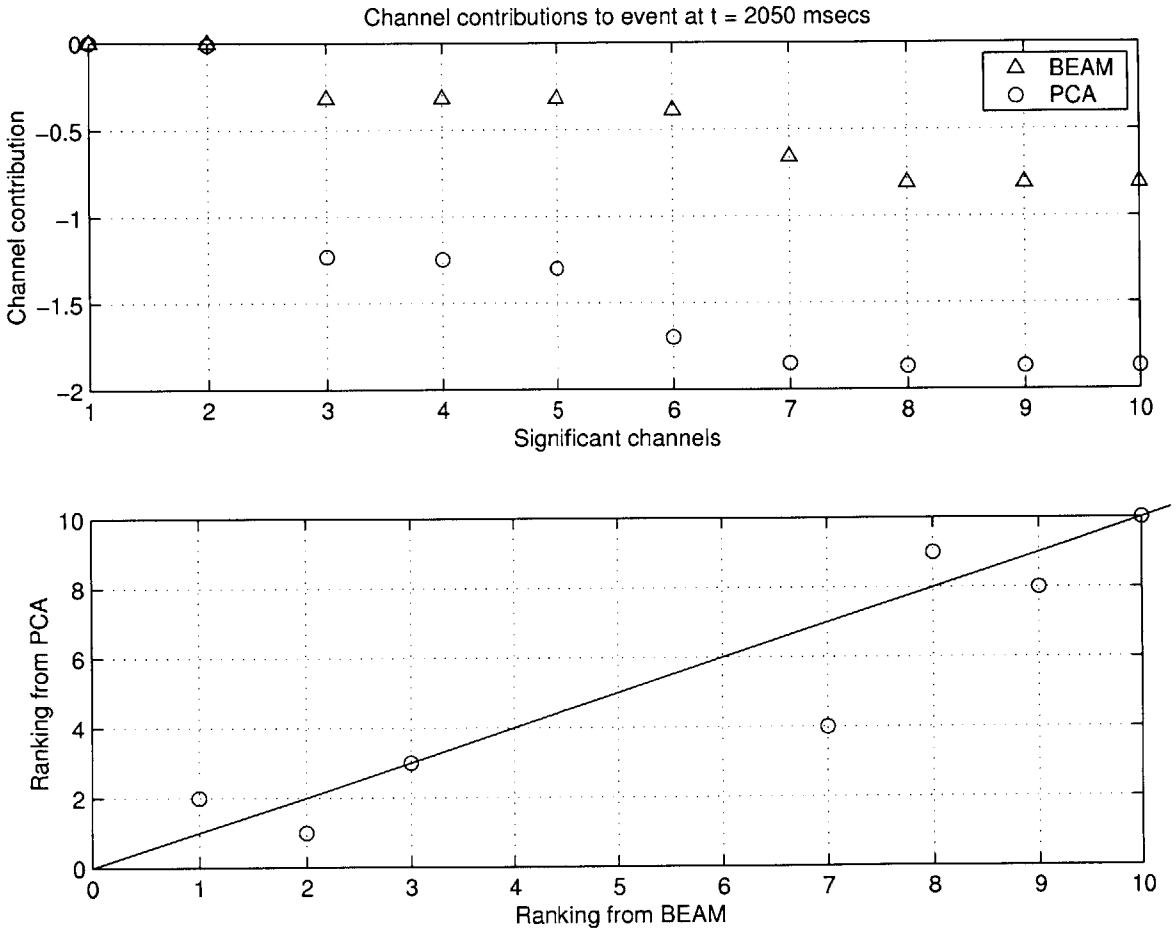

Figure A-2: Channel coefficient and ranking comparisons around event at 2050 msecs.

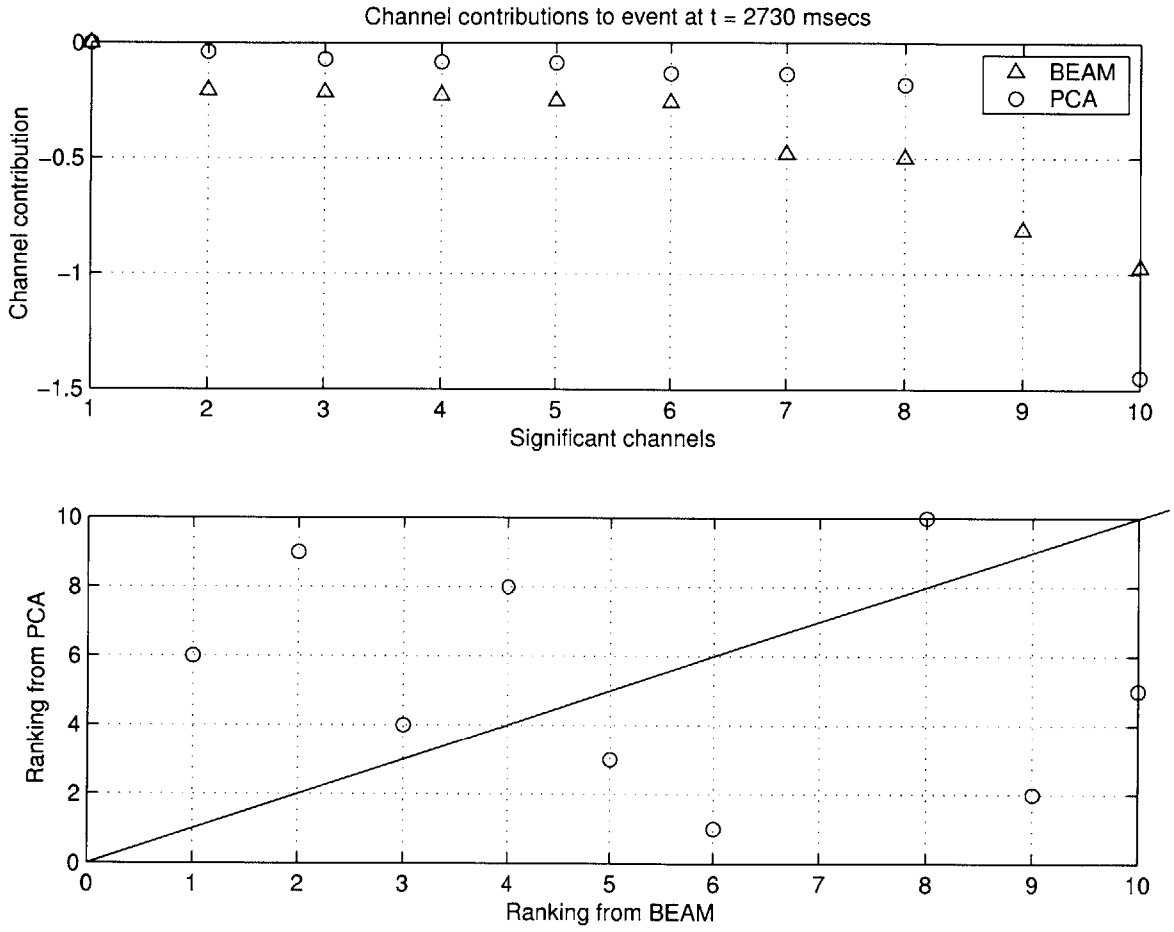

Figure A-3: Channel coefficient and ranking comparisons around event at 2730 msecs.

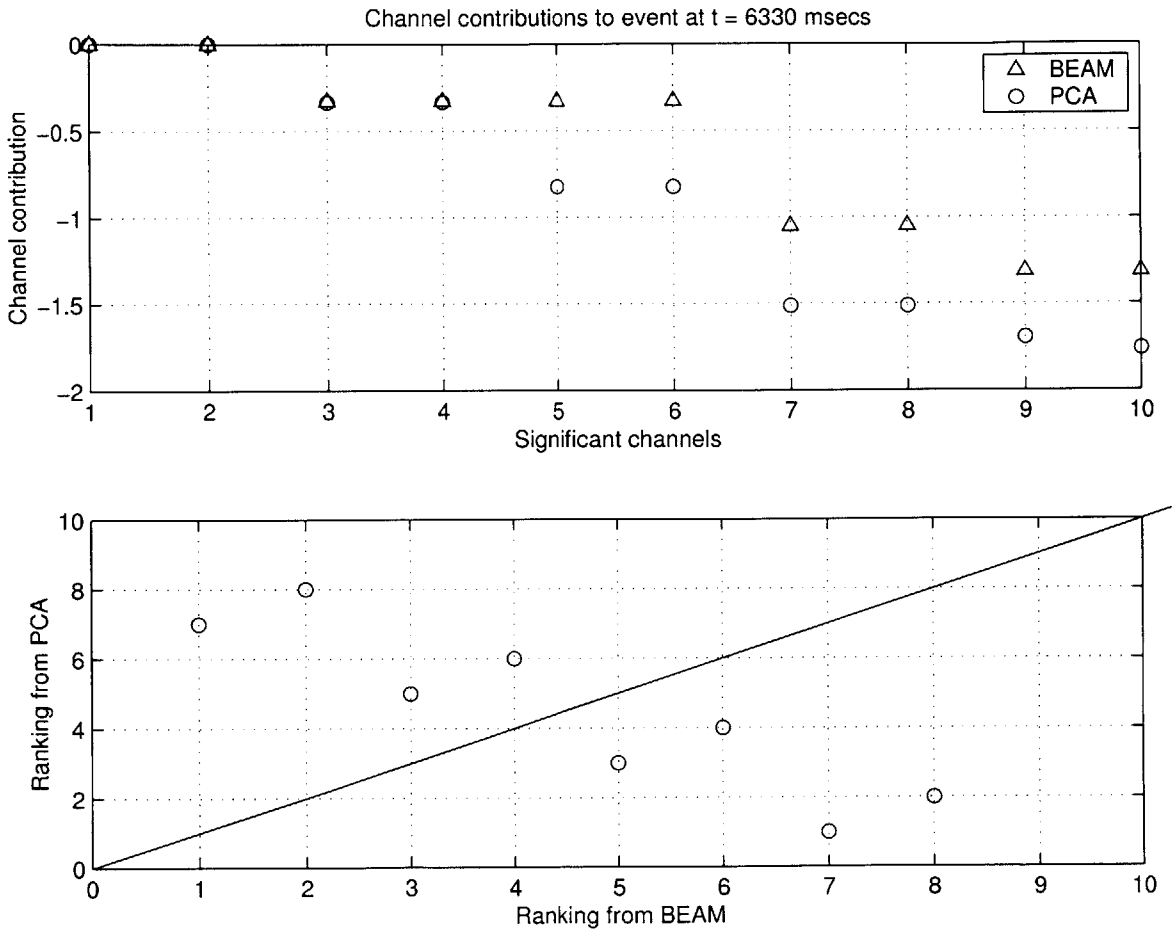

Figure A-4: Channel coefficient and ranking comparisons around event at 6330 msecs.

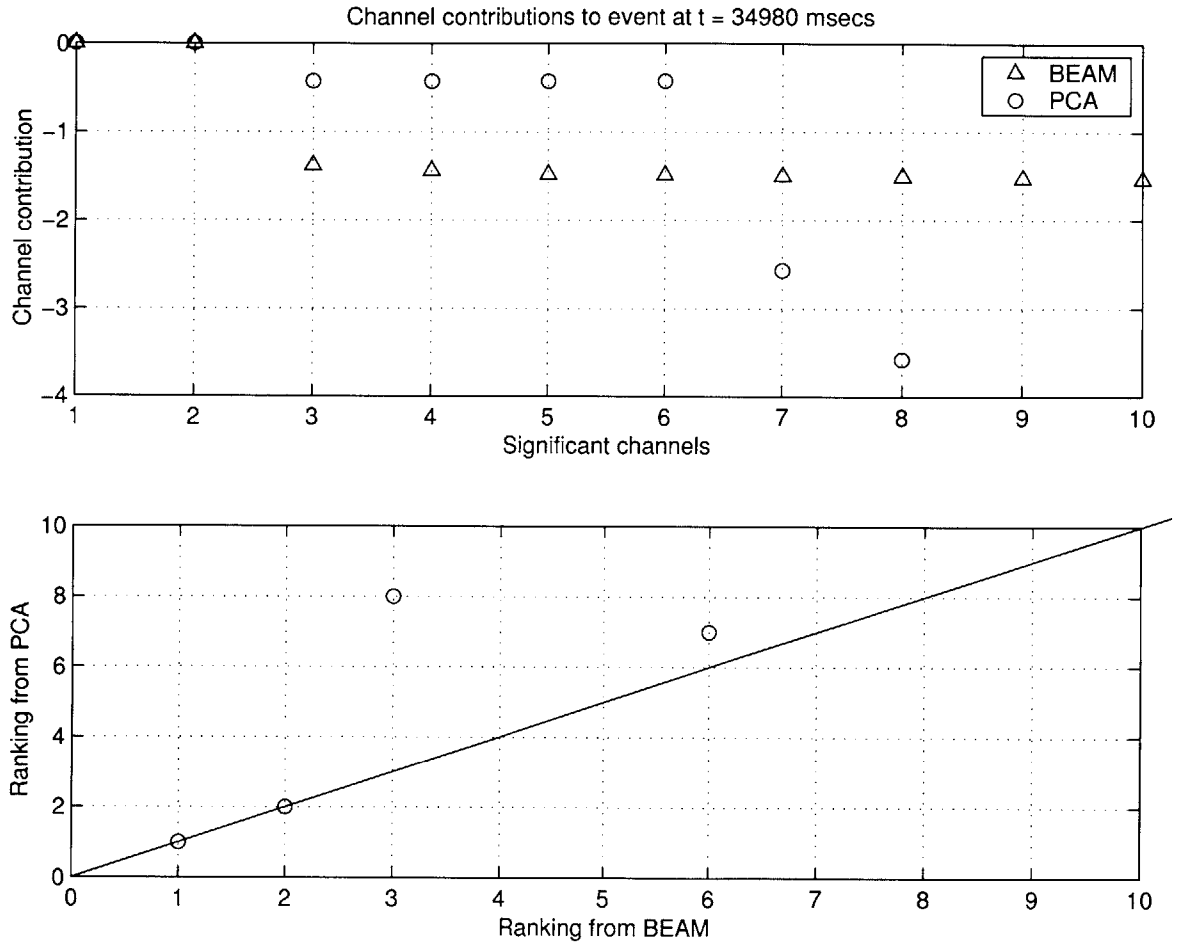

Figure A-5: Channel coefficient and ranking comparisons around event at 34,980 msecs.

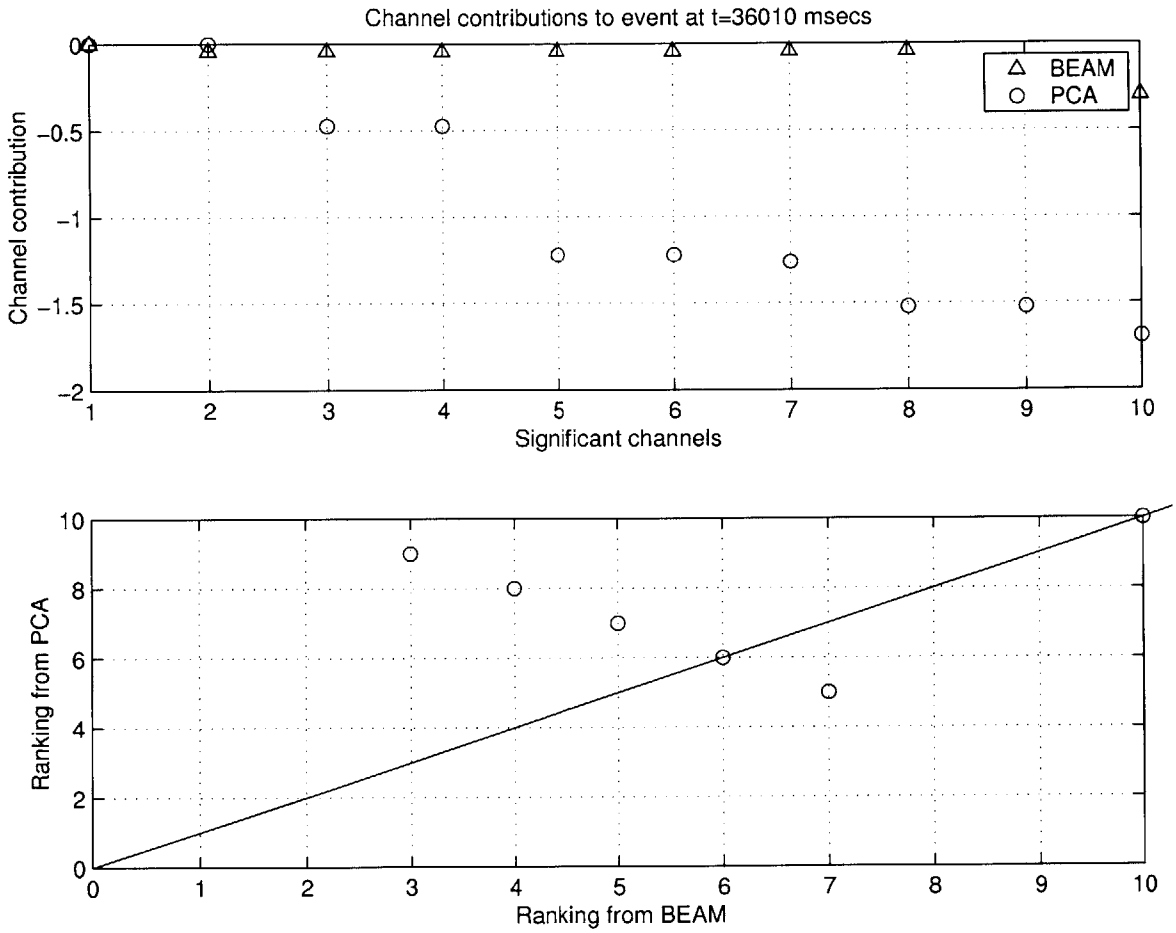

Figure A-6: Channel coefficient and ranking comparisons around event at 36,010 msecs.

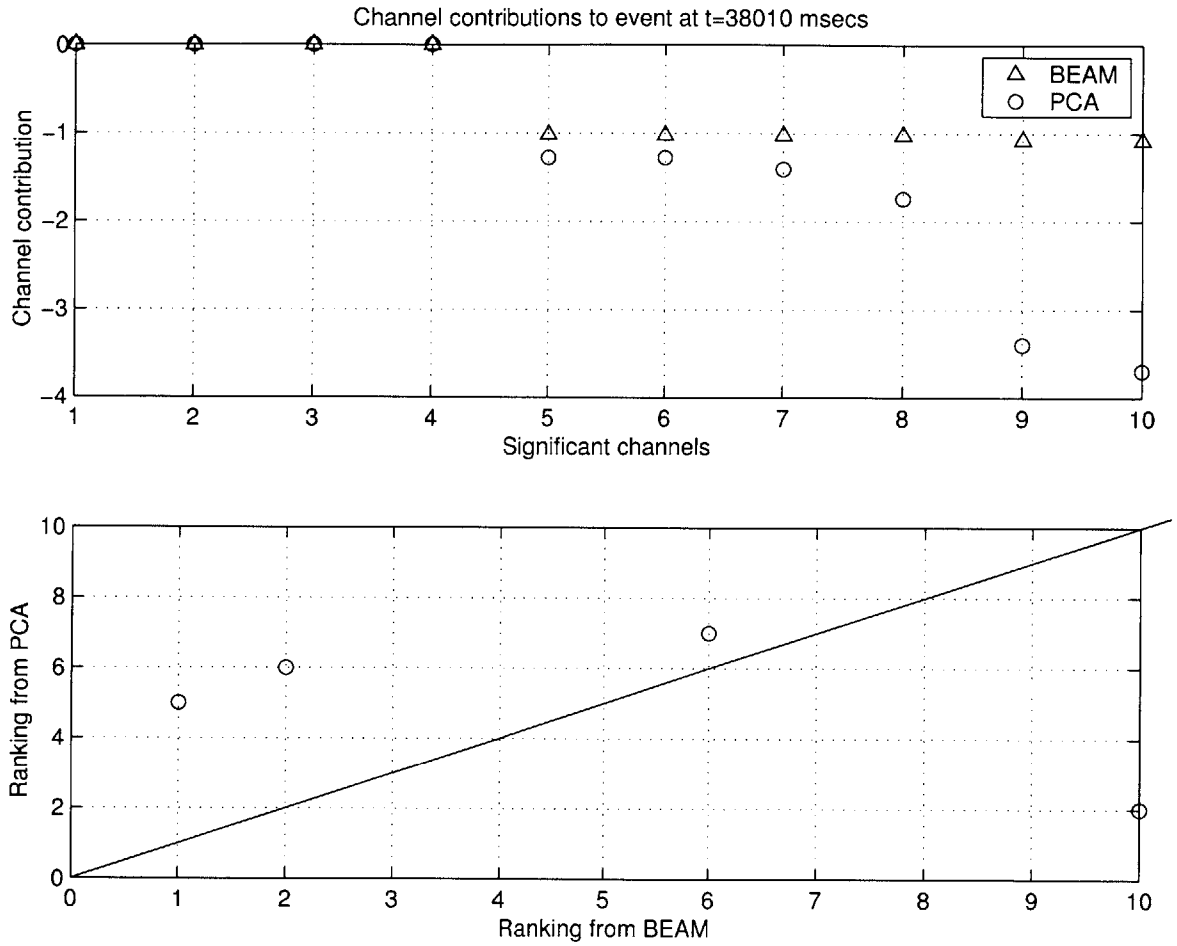

Figure A-7: Channel coefficient and ranking comparisons around event at 38,010 msecs.

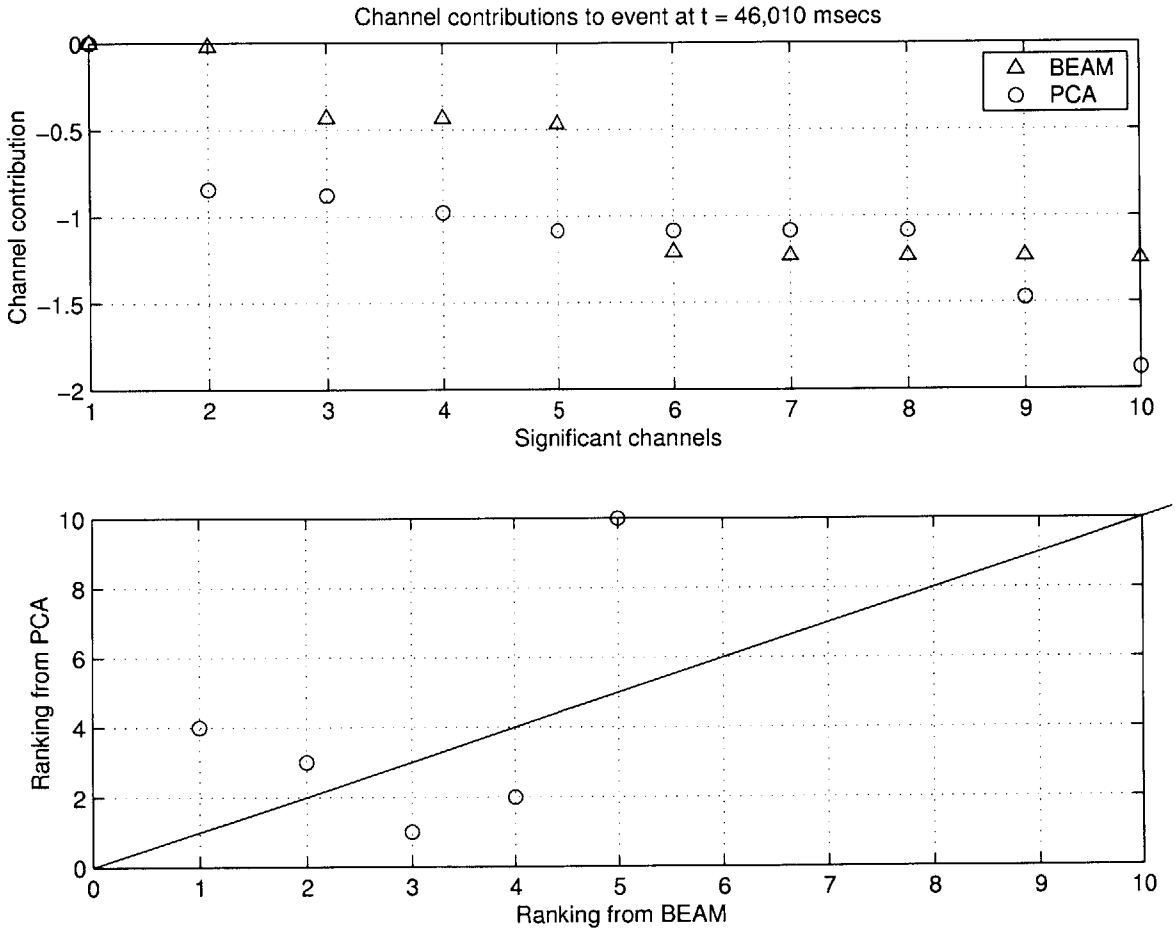

Figure A-8: Channel coefficient and ranking comparisons around event at 46,010 msecs.

Appendix B

Matlab Source Code

The following Matlab source codes were developed to evaluate both the performance of BEAM and to perform principal component analysis. In order to better understand the arguments and outputs of the scripts, we will define the structures of the most common variables that are used throughout the appendix.

data:

$$data \equiv \begin{bmatrix} channel_1[1] & channel_2[1] & \cdots & channel_n[1] \\ channel_1[2] & channel_2[2] & \cdots & channel_n[2] \\ \vdots & \vdots & \ddots & \vdots \\ channel_1[m] & channel_2[m] & \cdots & channel_n[m] \end{bmatrix} \tag{B.1}$$

score:

$$score \equiv \begin{bmatrix} score_1[1] & score_2[1] & \cdots & score_n[1] \\ score_1[2] & score_2[2] & \cdots & score_n[2] \\ \vdots & \vdots & \ddots & \vdots \\ score_1[m] & score_2[m] & \cdots & score_n[m] \end{bmatrix} \tag{B.2}$$

where $score_i$ refers to the i^{th} principal component.

chanBEAM and **chanPCA**:

$$chanX \equiv \begin{bmatrix} chanX_1[1] & chanX_1[2] & \cdots & chanX_1[t] \\ chanX_2[1] & chanX_2[2] & \cdots & chanX_2[t] \\ \vdots & \vdots & \ddots & \vdots \\ chanX_n[1] & chanX_n[2] & \cdots & chanX_n[t] \end{bmatrix} \tag{B.3}$$

where $chanX_i[j]$ refers to i^{th} channelś contribution to event number j under method X and there are a total of t events.

posBEAM and **posPCA**:

$$posX \equiv \begin{bmatrix} posX_1[1] & posX_1[2] & \cdots & posX_1[t] \\ posX_2[1] & posX_2[2] & \cdots & posX_2[t] \\ \vdots & \vdots & \ddots & \vdots \\ posX_n[1] & posX_n[2] & \cdots & posX_n[t] \end{bmatrix} \tag{B.4}$$

where $posX_i[j]$ refers to the i^{th} channelś ranking (significance) for event number j under method X and there are a total of t events.

rankBEAM and **rankPCA**:

$$rankX \equiv \begin{bmatrix} rankX_1[1] & rankX_1[2] & \cdots & rankX_1[t] \\ rankX_2[1] & rankX_2[2] & \cdots & rankX_2[t] \\ \vdots & \vdots & \ddots & \vdots \\ rankX_n[1] & rankX_n[2] & \cdots & rankX_n[t] \end{bmatrix} \tag{B.5}$$

where $rankX_i[j]$ refers to the i^{th} most significant channel number for event number j under method X and there are a total of t events.

Section I: The following functions were used to compute principal component transforms, eigenvectors, and eigenvalues.

```
function [pc, score, latent, pn] = mypca(data,type,debug);

% This function takes an m x n matrix, where n is the number of channels and m
% is the number of data points, and normalizes the data through normalize.m
% It produces the eigenvalues of its covariance matrix in 'latent', the eigenvectors
% in 'pc', and the principal components in 'score'.
% The outputs are rearranged to place the most significant principal component in the first
% column of 'score' and the largest eigenvalue in the first row of 'latent'.
%
% ARGUMENTS                                                                    10
%   data        - raw dataset matrix
%   type        - (0) normalizes the data with standard deviation
%                 (1) uses peak to peak value for normalization
%                 (-1) forces no normalization
%   debug       - nonzero value displays debugging messages
%
% OUTPUTS
%   pc          - eigenvectos
%   score       - principal components
%   latent      - eigenvalues                                                    20
%   pn          - normalized dataset

[m n] = size(data);
if type ~= -1
  disp('Normalizing data');
  pn = normalize(data', type)';
end
if(debug)
  disp('Creating covariance matrix');
end                                                                            30
covmat = cov(pn);
if(debug)
  disp('Done finding eigenvalues and eigenvectors');
```

```
end
[pc, latent]  = eig(covmat);
latent = diag(latent);
[latent, poslatent] = sort(-latent);
latent = -latent;
pc = pc(:,poslatent);
if(debug)                                                          40
 disp('Computing principal components');
end
score = pn*pc;
return
```

```
function [pn,meanp,divisor] = normalize(data, type)

% The function preprocesses the dataset by normalizing the inputs according to the
% 'type' argument.
%
% ARGUMENTS
%    data        - raw dataset to be normalized
%    type        - (0) normalize using standard deviations so that they have
%                    means of zero and standard deviations of 1 -> pn = (data-meanp)/stdp
%                  (1) normalizes by subtracting the mean and dividing by          10
%                    the peak to peak value -> pn = (data-meanp)/pp
%
% OUTPUTS
%    pn          - normalized dataset
%    meanp       - mean vector for dataset
%    divisor     - either standard deviation or peak to peak amplitude

if nargin > 2
 error('Wrong number of arguments.');
end                                                                20
if nargin == 1
 type = 0;
end
```

```
[R,Q]=size(p);
oneQ = ones(1,Q);
meanp = mean(data')';
if(type == 0)
  divisor = std(data')';
  disp('Using standard deviation to normalize');
else
  disp('Using peak to peak to normalize');
  maxi = max(data')';
  mini = min(data')';
  divisor = maxi-mini;
end
equal = divisor == 0;
nequal = ~equal;
if sum(equal) ~= 0
  divisor0 = divisor.*nequal + 1*equal;
else
  divisor0 = divisor;
end
pn = (data-meanp*oneQ)./(divisor0*oneQ);
return
```

Section II: The following functions were used to capture system-wide events under principal component analysis.

```
function [events_PCA] = pcaevents(score, thresh, inc)

% The function takes the principal components, 'score', and a threshold vector, 'thresh',
% and finds the location of events among the significant principal components.
% Each principal component and its respective threshold is passed to findpcevents.m
% which in turn returns the location of events in that particular PC.
% The outputs of the 'length(thresh)' number of PC's is then merged by examining the
% locations of these events and combining any repetitive events to create a master list that
% contains all the events seen by all the significant PC's.
%
% ARGUMENTS
```

```
% score      - all significant principal components to be used to detec events
% thresh     - array of thresholds to be used for each principal component
% inc        - maximum number of samples that two events can be separated by
%              before a new and unique event is declared
%
% OUTPUTS
% events_PCA - master array of events under PCA

if nargin < 3
  inc = 50;
end
depth = length(thresh);
fig = 1;
start = 1;
fig_vector = [fig, start, depth, depth];
d_pc = myplot(score, -2, fig_vector, 1, 2);
hold on;
events_PCA = [];
for i = 1:depth
  t = thresh(i);
  [tmp cur_pos] = findpcaevents(d_pc(:,i), t, inc);
  events_PCA = [events_PCA cur_pos'];
end
events_PCA = sort(events_PCA);
d_pos = diff(events_PCA);
ln = length(d_pos);
j = 1;
pos(1) = events_PCA(1);
for i = 1:ln
  if d_pos(i) >= inc
    j = j+1;
    pos(j) = events_PCA(i+1);
  end
end
ln = length(pos);
events_PCA = pos';
```

```
figure(start);
for i = 1:depth
  subplot(depth, 1, i);
  hold on;
  thresh_bar = thresh(i)*ones(length(score),1);
  plot(thresh_bar, '-');
  xlabel('Time (100*sec)');
  hold off;
end
return
```

```
function [maxi, pos] = findpcaevents(d_score, thresh, inc)

% The function takes a vector, 'd_score', a threshold, 'thresh', and combines all the elements
% of the vector that exceed the threshold and lie within 'inc' samples of each other.
% This is used to eliminate any redundant capturing of events surrounding a transition
% in the principal component, i.e., there usually exists multiple tall spikes in the
% difference around a transition.
%
% ARGUMENTS
%   d_score     - vector of difference principal component
%   thresh      - threshold that determines if an event occurs
%   inc         - the maximum number of samples two events can lie within
%                 before function declares another event
%
% OUTPUTS
%   maxi        - value of difference score where it exceeded threshold
%   pos         - position of difference score where it exceeded threshold

if nargin == 2
  inc = 50;
end
tmp_pos = find(d_score >= thresh);
d_pos = diff(tmp_pos);
ln = length(d_pos);
```

```
j = 1;
pos(1) = tmp_pos(1);
for i = 1:ln
  if d_pos(i) >= inc
    j = j+1;
    pos(j) = tmp_pos(i+1);                                          30
  end
end
pos = pos';
maxi = d_score(pos);
return
```

Section III: The following functions were used to read in pre-processed channel contribution information from BEAM as well as compute the channel contributions under PCA.

```
function [out1, out2] = readchan(filename, posPCA, eventnumber)

% The function reads the channel contribution data from the inputfile.
% The inputfile is a column of percentages ordered by the channel number.
% A -1 within the inputfile means a new event has occured.
% The function reads the channel data from filename and returns the contribution
% and channel rankings in descending order.
% If 'posPCA' and 'eventnumber' are supplied, the function also takes the
% position matrix from the outcome of findchan and plots the comparison
% in a scatter plot while returning the rankings in out1 and out2.        10
%
% ARGUMENTS
%   filename     - input file name that contains the channel contributions from BEAM
%   posPCA       - position vector determined by findchan.m
%   eventnumber  - which event to compare rankings
%
% OUTPUTS
%   nargin == 1
%   out1         - chanBEAM -> channel contributions under BEAM
%   out2         - posBEAM  -> channel positions under BEAM                20
```

```
%   nargin ~= 1
%   out1          - rankBEAM -> channel rankings under BEAM
%   out2          - rankPCA  -> channel rankings under PCA

if nargin == 1
 posPCA = 0;
end
fid = fopen(filename, 'r');
stop = 0;
tmp = 0;                                                          30
i = 1;
while ~stop
  j = 1;
  while tmp ~=-1 & ~stop
    tmp = fscanf (fid, '%lf',1);
    if isempty(tmp)
        stop = 1;
    else if tmp ~= -1
        chanmat(j,i) = tmp;
        j = j+1;                                                  40
        end
    end
  end
  i = i+1;
  tmp = 0;
end
if nargin < 3
 eventnumber = i-2;
end
% negate the positive vector to sort in descending order          50
chanmat = -chanmat;
[cont,pos] = sort(chanmat);
% now return the original values
cont = -cont;
[misc, rank] = sort(pos);
if posPCA == 0
```

```
  out1 = cont;
  out2 = pos;
  return
end                                                                    60
out1 = rank;
[misc, out2] = sort(posPCA);
if nargin == 3
  rankplot(out2, out1, eventnumber);
else
  rankplot(out2, out1);
end
return
```

```
function [out1,out2,out3] = findchan(data, pc_num, eventBEAM, sigchan, wv, oflag, opt)

% The function computes the contribution from 'sigchan' number of variables and sorts
% them in descending order.
%
% ARGUMENTS
%     data        - either normalized dataset or principal components (score)
%     pc_num      - vector of principal component numbers to be used for events
%                   i.e. [1 3 1 1] translates to using pc 1 for event 1, pc 3 for event2, etc...
%                   if scalar value, 'pc_num' is used for all events                    10
%     eventBEAM   - array of time markers to system-wide events
%     sigchan     - number of significant channels to be computed
%     wv          - array of sizes of windows to be used around each event
%                   i.e. [10 20] translates to using 10 sample window for event 1
%                   and 20 sample window for event 2
%     oflag       - if nonzero, findchan returns the [pc, score, latent]
%                   around event# 'oflag' -> 'pc_num' nor 'sigchan' is used
%     opt         - (1) uses the coefficients in the eigenvector to sort
%                   channel contribution
%                                                                          20
% OUTPUTS
%     oflag ~=0
```

```
%    out1        - pc -> eigenvectors of windowed PCA around event 'oflag'
%    out2        - score -> principal components
%    out3        - latent -> eigenvalues
%
%    oflag ==0
%    out1        - chanPCA -> channel coefficients
%    out2        - posPCA -> rankings of each channel
                                                                        30

if nargin < 6
  oflag = 0;
end
if nargin < 7
  opt = 0;
end
if nargin < 5
  error('FINDCHAN must have at least 5 arguments');
end                                                                     40
% check if 'pc_num' array is correct in length
% if scalar, just create identical array
points = max(size(eventBEAM));
pointspca = max(size(pc_num));
if pointspca == 1
  pc_num = ones(1,points).*pc_num;
end
if max(size(pc_num)) ~= points
  error('pc_num vector incorrect in size');
end                                                                     50
[m n] = size(data);
% start loop at oflag so as to avoid unneccesary computations
start = max([oflag 1]);
if start > points
  error('oflag is too large:  EXITING');
end
% make elements of 'wv' even
wv= 2*floor(wv/2);
```

```
% see if 'wv' was passed as vector argument
if length(wv) == 1                                                    60
  wv = ones(1,points)*wv;
end
for i = start:points
 index = eventBEAM(i);
 pc_index = pc_num(i);
 window = wv(i);
 if index+window/2 > m
  en = m;
  st = m - window;
  ei = window - m + index;                                            70
 else if index-window/2 <= 0
      st = 1;
      en = 1 + window;
      ei = index;
    else
        st = index - window/2;
        en = index + window/2;
        ei = window/2+1;
    end
 end                                                                  80
 if oflag > 0
  disp(sprintf('Event %d occurs @ %d', i, ei));
 end
 windata = data(st:en, : );
 windatad = diff(windata);
 % compute pca of windowed data
 % -1 argument to mypca is no normalization
 [pc2, score, latent] = mypca(windata,-1, 0);
 % find the channels significant to transition at window/2
 % note that the new windata has the event occur at index=window/2     90
 if i == oflag
  disp(sprintf('Returning windowed PCA output of event %d', i));
  out1 = pc2;
  out2 = score;
```

```
  out3 = latent;
  return;
 else
  disp(sprintf('Using PC %d for analysis on Event %d', pc_index, i));
  if opt == 0
    % take the slice of the data set at the index given in 'ei'
    cut = windatad(ei-1, : );
    % compute the coefficients in the PC summation and figure out the sign
    % of the derivative of the PC at the point 'index' using principal
    % component pc_index
    coeff = cut.*pc2(:,pc_index)';
    sgn = cut*pc2(:,pc_index);
    % find level number of signficant channels
    for j = 1:sigchan
      % if derivative of PC at index is < 0, then find min
      % where if derivative of PC is > 0, find max contributor
      if sgn > 0
        [chanPCA,posPCA] = max(coeff);
        coeff(posPCA) = min(coeff)-99999;
      else
        [chanPCA,posPCA] = min(coeff);
        coeff(posPCA) = max(coeff)+99999;
      end
      % save channel positions and coefficient values
      out2(j,i) = posPCA;
      out1(j,i) = chanPCA;
    end
   else if opt == 1
            [out1,out2] = sort(-abs(pc2(:,pc_index)));
          end
   end
  end
end
return
```

```
function chanplot(chanPCA, chanBEAM, eventnumber, level, subp)

% The function plots the channel contribution from both the BEAM computation and
% PC analysis in log scale.
%
% ARGUMENTS
%     chanPCA      - channel contributions under PCA
%     chanBEAM     - channel contributions under BEAM
%     eventnumber  - which event to examine
%                    default is to examine all events                    10
%     level        - number of significant channels to compute
%                    default is 30 channels
%     subp         - (0) stand-alone computation
%                    (1) chanplot called by larger function

if nargin <= 3
  level = 30;
end
if nargin == 2
  [misc, eventnumber] = size(chanPCA);                                   20
end
if nargin < 5
  subp = 0;
end
if nargin >= 3
  start = eventnumber;
else
  start = 1;
end
for i = start:eventnumber                                                30
  chanBEAM(:,i) = chanBEAM(:,i)/chanBEAM(1,i);
  chanPCA(:,i) = chanPCA(:,i)/chanPCA(1,i);
  if ~subp
    figure(i-start+1);
  end
  tmpBEAM = real(log10(chanBEAM(1:level, i)));
```

```
tmpPCA = real(log10(chanPCA(1:level, i)));
hold off;
plot(tmpBEAM, '^');
hold on;                                                          40
grid on;
ylabel('Channel contribution');
xlabel('Significant channels');
plot(tmpPCA, 'o');
legend('BEAM', 'PCA');
title(sprintf('Channel contribution to event %d', i));
hold off;
end
```

Section IV: The following functions were used to compute the channel rankings under BEAM and PCA.

```
function [rankBEAM, rankPCA] = rankchan(posBEAM, posPCA)

% The function takes the position matrix of both PCA and BEAM and computes the rankings.
% The input arguments were computed via findchan.m
%
% ARGUMENTS
%   posBEAM   - position vector for BEAM
%   posPCA    - position vector for PCA
%
% OUTPUTS                                                          10
%   rankBEAM  - ranking vector for BEAM
%   rankPCA   - ranking vector for PCA

[tmp, rankBEAM] = sort(posBEAM);
[tmp, rankPCA] = sort(posPCA);
return
```

```
function rankplot(rankPCA, rankBEAM, eventnumber, level, subp)
```

```
% The function plots the channel rankings from both the event horizon computation and
% PC analysis.
%
% ARGUMENTS
%   eventnumber – determines the event to look at
%                 if not supplied by the user, the default is to plot all of scatter plots
%   subp        – (0) rankplot was called from terminal
%                 (1) rankplot was called by a larger plotting routine                    10

if nargin > 2
  start = eventnumber;
else
  [misc, eventnumber] = size(rankPCA);
  start = 1;
end
if nargin < 4
  subp = 0;
end                                                                                      20
for i = start:eventnumber
  if ~subp
    figure(i-start+1);
  end
  plot(rankBEAM(:,i), rankPCA(:,i), 'o');
  hold on;
  plot([0 max(rankPCA(:,1))], [0 max(rankPCA(:,1))], 'r');
  xlabel('Ranking from BEAM');
  ylabel('Ranking from PCA');
  title(sprintf('Channel Comparison around Event %d', i));                               30
  axis([0 level 0 level]);
  hold off;
  grid on;
end
```

Section V: The following functions were used to numerically and visually compare the events and channel contributions under BEAM and PCA.

```
function [posx, posy] = cmpevents(eventBEAM, eventPCA, inc);

% The function compares the events posted by BEAM  and PCA.
% It eliminates any redundant set of points that fall within the 'inc' data points
% and arranges the 'posx' and 'posy' vectors to plot along a 45 degree line.
% If an event occurs in both BEAM and PCA, then it falls on the 45 degree line.
%
% ARGUMENTS
%   eventBEAM   - events detected by BEAM
%   eventPCA    - events detected by PCA                                    10
%   inc         - the maximum difference between two events before
%                 cmpevents declares a new event
%
% OUTPUTS
%   posx        - xaxis coordinates on the event comparison plot
%   posy        - yaxis coordinates on the event comparison plot

[m n] = size(eventPCA);
if m>n
  eventPCA = eventPCA';                                                     20
end
[m n] = size(eventBEAM);
if m>n
  eventBEAM = eventBEAM';
end
mpos = [eventPCA eventBEAM];
mpos = sort(mpos);
if nargin == 2
  inc = 50;
end                                                                         30
d_pos = diff(mpos);
ln = length(d_pos);
j = 1;
for i = 1:ln
  if d_pos(i) >= inc
    j = j+1;
```

```
  end
end
posx = zeros(j, 1);
posy = posx;
if mpos(1) == eventPCA(1)
  posy(1) = mpos(1);
  posx(1) = 0;
else
  posy(1) = 0;
  posx(1) = mpos(1);
end
j = 1;
k = 1;
for i = 1:ln
  a = find( eventPCA == mpos(i+1) );
  b = find( eventBEAM == mpos(i+1) );
  if d_pos(i) >= inc
    j = j+1;
  end
  if ~isempty(a)
    posy(j) = mpos(i+1);
  end
  if ~isempty(b)
    posx(j) = mpos(i+1);
  end
end
plot(posx, posy, 'b^');
hold on;
plot([0 max(mpos)], [0 max(mpos)], 'r');
grid on;
title('Event comparison plot');
ylabel('Events under PCA (100*sec)');
xlabel('Events under BEAM (100*sec)');
hold off;
return
```

```
function quickchan(chanPCA, chanBEAM, rankPCA, rankBEAM, opt, level, figvector,
                   data, eventBEAM, window)

% The function plots the channel contribution and its respective ranking comparisons on a
% single figure per event. The upper figure is a superposition of the channel coefficients
% from BEAM and PCA. The lower figure is a comparison of the rankings.
%
% ARGUMENTS
%    chanPCA    - coefficients determined by PCA
%    chanBEAM   - coefficients determined by BEAM                              10
%    rankPCA    - ranking of channel contributions by PCA
%    rankBEAM   - ranking of channel contributions by BEAM
%    opt        - (0) plot only channel histogram and rankings
%                 (1) calls findchan s.t. the eigenvalues
%                 and the pc's can be plotted
%    level      - number of channels to look at
%    figvector  = [fig, start, en]
%        fig    - first figure number
%        start  - first event to be plotted
%        en     - last event to be plotted                                     20
%    data       - only used if opt = 1
%    eventBEAM  - "
%    window     - "

if nargin < 4
 error('QUICKCHAN requires at least four inputs');
end
if nargin <= 6
 [misc, en] = size(chanPCA);
 start = 1;                                                                    30
 fig = 1;
else
 fig = figvector(1);
 start = figvector(2);
 en = figvector(3);
end
```

```
if nargin <= 5
  level = 15;
end
if nargin == 4                                                    40
  opt = 0;
end
if opt == 1
  if nargin ~= 10
    error('QUICKCHAN requires nine arguments with option = 1');
  end
end
eventcount = min(size(chanBEAM));
if en > eventcount
 error('EN exceeds the number of events');                        50
end
if opt == 0
 for i = start:en
  figure(i-start+fig);
  subplot(2,1,1);
  hold off;
  chanplot(chanPCA, chanBEAM, i, level, 1);
  hold off;
  subplot(2,1,2);
  hold off;                                                       60
  rankplot(rankPCA, rankBEAM, i, level, 1);
  hold off;
  axis([0 level 0 level]);
 end
else
 for i = start:en
  figure(i-start+fig);
  subplot(3,2,1);
  hold off;
  chanplot(chanPCA, chanBEAM, i, level, 1);                       70
  hold off;
  subplot(3,2,3);
```

```
hold off;
rankplot(rankPCA, rankBEAM, i, level, 1);
hold off;
[pc, score, latent] = findchan(data, 1, eventBEAM, max(size(rankPCA)), window, i);
scored = diff(score);
subplot(3,2,5);
hold off;
semilogy(latent(1:level));
grid on;
title('Eigenvalue plot');
hold off;
for j = 1:3
  subplot(3,2,2*j);
  hold off;
  plot(score(:,j));
  hold on
  plot(scored(:,j), 'r');
  title(sprintf('PC %d', j));
  grid on;
  hold off;
 end
end
end
```

Section VI: The following function is a custom plotting function utilized throughout the analysis.

```
function [d_score] = myplot(score, eventBEAM, figvector, range, options)

% The function plots each vector in the matrix 'score' indexed from start to 'en'
% and 'inc' many per plotting window.
%
% ARGUMENTS
%   score      - matrix of data, channel by column
%   eventBEAM  - the event vector determined by BEAM
%   figvector  = [fig, start, en, inc]
```

```
%       fig       - first figure number                                    10
%       start     - first channel to be plotted
%       en        - last channel to be plotted
%       inc       - subplots per figure
%   range         - range of the time axis, [xmin xmax]
%                 - (1) range is length(score)
%   options       - (1) semilogy is used
%                 - (2) plots both 'score', it's derivative, and 'eventBEAM'
%                   returns derivative to 'd_score'
%                   set 'eventBEAM'=0 to plot only score
%                                                                          20
% OUTPUTS
%   d_score       - matrix of data, channel by column

if nargin < 4
  error('MYPLOT requires at least four arguments');
end
if nargin == 4
  options = 0;
end
if options == 2                                                            30
  d_score = diff(score);
end
fig = figvector(1);
start = figvector(2);
en = figvector(3);
inc = figvector(4);
if length(range) == 2
  xmin = range(1);
  xmax = range(2);
else                                                                       40
  xmin = 1;
  xmax = length(score(:,1));
end
j = fig-1;
for i = start:en
```

```
if mod(i-start, inc) == 0
  j = j+1;
  figure(j);
  subplot(inc, 1, 1);
else
  subplot(inc, 1, (i-(j-fig)*inc));
end
if options == 1
  semilogy(score(:,i));
else
  plot(score(:,i), 'b');
end
hold on;
title(sprintf('Principal Component %d', i));
subscore(:,i) = score(xmin:xmax,i);
if options == 2
  subd(:,i) = d_score(xmin:xmax-1, i);
end
ymin =  min(subscore(:,i));
if ymin > 0
  ymin = 0;
end
bound = [xmin xmax ymin max(abs(subscore(:,i))) ];
axis(bound);
if options == 2
  subd(:,i) = (abs(subd(:,i)));
  d_score(:,i) = abs(d_score(:,i)).*(max(abs(subscore(:,i))) / max(subd(:,i)));
  plot(d_score(:,i), 'r');
end
if length(eventBEAM) > 1
  subevent = eventBEAM(xmin:xmax);
  eventBEAM = eventBEAM.*(max(abs(subscore(:,i)))/max(subevent));
  plot(eventBEAM, 'w');
end
hold off;
end
```

```
if eventBEAM < 0
  return
end
```

Bibliography

[1] Lao Fa-Long and Li Yan-Da. Real-time computation of the eigenvector corresponding to the smallest eigenvalue of a positive definite matrix. *IEEE Transactions on Circuits and Systems-I: Fundamental Theory and Applications*, 41(8):550–553, August 1994.

[2] I.T. Jolliffe. *Principal Component Analysis.* Springer Series in Statistics. Springer-Verlag, New York, New York, 1986.

[3] Junehee Lee. *Blind Noise Estimation and Compensation for Improved Characterization of Multivariate Processes.* PhD dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, January 2000.

[4] J. Smith. Test request, xrs-2200 powerpack assembly, test a1x003: Revision b. Test request documentation for the a1x003 dataset, October 1998.

[5] Alan S. Willsky, Gregory W. Wornell, and Jeffrey H. Shapiro. *Course Notes: 6.432 Stochastic Processes, Detection, and Estimation*, chapter 5. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1994.