

Fast Algorithm Development for SVD: Applications in Pattern Matching and Fault Diagnosis

N.Rajasekhar

A Thesis Submitted to
Indian Institute of Technology Hyderabad
In Partial Fulfillment of the Requirements for
The Degree of Master of Technology



भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Department of Electrical Engineering

June 2018

Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.

N. Rajasekhar

(Signature)

N. RAJASEKHAR

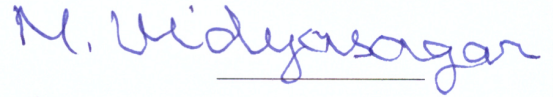
(N.Rajasekhar)

EE16MTECH11037

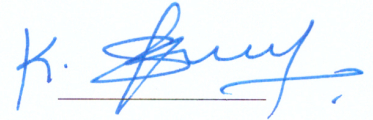
(Roll No.)

Approval Sheet

This Thesis entitled Fast Algorithm Development for SVD: Applications in Pattern Matching and Fault Diagnosis by N.Rajasekhar is approved for the degree of Master of Technology from IIT Hyderabad



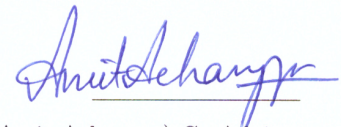
(Prof. M Vidyasagar) Examiner
Dept. of Electrical Eng
IITH



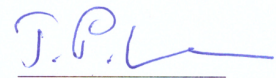
(Dr. Siva Kumar K) Examiner
Dept. Electrical Eng
IITH



(Dr. Ketan Detroja) Adviser
Dept. of Electrical Eng
IITH



(Dr. Amit Acharyya) Co-Adviser
Dept. of Electrical Eng
IITH



(Dr. Phanindra Jampana) Chairman
Dept. of Chemical Eng
IITH

Acknowledgements

I would like to thank my supervisor Dr. Ketan Detroja and co-supervisor Dr. Amit Acharyya for their continuous encouragement and constructive comments throughout my project work.

I would like to express my sincere gratitude to Ms. Rashi Dutt and other labmates of AESICD for their support in the hardware implementation of SVD.

Abstract

The project aims for fast detection and diagnosis of faults occurring in process plants by designing a low-cost FPGA module for the computation. Fast detection and diagnosis when the process is still operating in a controllable region helps avoiding the further advancement of the fault and reduce the productivity loss. Model-based methods are not popular in the domain of process control as obtaining an accurate model is expensive and requires an expertise. Data-driven methods like Principal Component Analysis(PCA) is a quite popular diagnostic method for process plants as they do not require any model. PCA is widely used tool for dimensionality reduction and thus reducing the computational effort. The trends are captured in principal components as it is difficult to have a same amount of disturbance as simulated in historical database. The historical database has multiple instances of various kinds of faults and disturbances along with normal operation. A moving window approach has been employed to detect similar instances in the historical database based on Standard PCA similarity factor. The measurements of variables of interest over a certain period of time forms the snapshot dataset, S . At each instant, a window of same size as that of snapshot dataset is picked from the historical database forms the historical window, H . The two datasets are then compared using similarity factors like Standard PCA similarity factor which signifies the angular difference between the principal components of two datasets. Since many of the operating conditions are quite similar to each other and significant number of mis-classifications have been observed, a candidate pool which orders the historical data windows on the values of similarity factor is formed. Based on the most detected operation among the top-most windows, the operating personnel takes necessary action. Tennessee Eastman Challenge process has been chosen as an initial case study for evaluating the performance. The measurements are sampled for every one minute and the fault having the smallest maximum duration is 8 hours. Hence the snapshot window size, m has been chosen to be consisting of 500 samples i.e 8.33 hours of most recent data of all the 52 variables. Ideally, the moving window should replace the oldest sample with a new one. Then it would take approximately the same number of comparisons as that of size of historical database. The size of the historical database is 4.32 million measurements(past 8years data) for each of the 52 variables. With software simulation on Matlab, this takes around 80-100 minutes to sweep through the whole 4.32 million historical database. Since most of the computation is spent in finding principal components of the two datasets using SVD, a hardware design has to be incorporated to accelerate the pattern matching approach.

The thesis is organized as follows: Chapter 1 describes the moving window approach, various similarity factors and metrics used for pattern matching. The previous work proposed by Ashish Singhal is based on skipping few samples for reducing the computational effort and also employs windows as large as 5761 which is four days of snapshot. Instead, a new method which skips the samples when the similarity factor is quite low has been proposed. A simplified form of the Standard PCA similarity has been proposed without any trade-off in accuracy. Pre-computation of historical database can also be done as the data is available aprior, but this requires a large memory requirement as most of the time is spent in read/write operations. The large memory requirement is due to the fact that every sample will give rise to 52×35 matrix assuming the top-35 PC's are sufficient enough to capture the variance of the dataset. Chapter 2 describes various popular algorithms for SVD. Algorithms apart from Jacobi methods like Golub-Kahan, Divide and conquer SVD algorithms are briefly discussed. While bi-diagonal methods are very accurate they suffer from

large latency and computationally intensive. On the other hand, Jacobi methods are computationally inexpensive and parallelizable, thus reducing the latency. We also evaluated the performance of the proposed hybrid Golub-Kahan Jacobi algorithm to our application. Chapter 3 describes the basic building block CORDIC which is used for performing rotations required for Jacobi methods or for n-D householder reflections of Golub-Kahan SVD. CORDIC is widely employed in hardware design for computing trigonometric, exponential or logarithmic functions as it makes use of simple shift and add/subtract operations. Two modes of CORDIC namely Rotation mode and Vectoring mode are discussed which are used in the derivation of Two-sided Jacobi SVD. Chapter 4 describes the Jacobi methods of SVD which are quite popular in hardware implementation as they are quite amenable to parallel computation. Two variants of Jacobi methods namely One-sided and Two-sided Jacobi methods are briefly discussed. Two-sided Jacobi making use of CORDIC has been derived. The systolic array implementation which is quite popular in hardware implementation for the past three decades has been discussed. Chapter 5 deals with the Hardware implementation of Pattern matching and reports the literature survey of various architectures developed for computing SVD. Xilinx ZC7020 has been chosen as target device for FPGA implementation as it is inexpensive device with many built-in peripherals. The latency reports with both Vivado HLS and Vivado SDSoc are also reported for the application of interest. Evaluation of other case studies and other data-driven methods similar to PCA like Correspondence Analysis(CA) and Independent Component Analysis(ICA), development of efficient hybrid method for computing SVD in hardware and highly discriminating similarity factor, extending CORDIC to n-dimensions for householder reflections have been considered for future research.

Contents

Declaration	ii
Approval Sheet	iii
Acknowledgements	iv
Abstract	v
1 Pattern Matching Approach	1
1.1 Overview of Fault Diagnosis Methods	1
1.2 Previous work	2
1.3 Pre-processing of dataset	2
1.4 Computation of PCA	2
1.4.1 Singular value Decomposition	2
1.4.2 Eigen Value Decompositon	3
1.5 Similarity factors	3
1.5.1 Standard PCA similarity factor	3
1.5.2 Modified PCA similarity factor	4
1.5.3 Distance similarity factor	5
1.5.4 Dissimilarity factor	5
1.6 Metrics for Pattern Matching	6
1.6.1 Pool Accuracy	6
1.6.2 Pattern Matching Efficiency	6
1.7 Tennessee Eastman Challenge Process	7
1.7.1 Introduction	7
1.8 Proposed Pattern matching approach	7
1.8.1 Selection of window size	8
1.8.2 Pre-computing of Historical Database	8
1.8.3 Clustering	9
1.8.4 Overview of Historical Data	9
1.9 Results for pattern matching	10
1.9.1 Methodology for Pattern Matching	10
2 Algorithms for SVD	16
2.1 Introduction	16
2.1.1 Properties of SVD	16
2.2 Golub-Kahan-Reinsch SVD:	16

2.2.1	Algorithm 1a	16
2.2.2	Householder Reflections	17
2.2.3	Givens Rotations	19
2.2.4	Algorithm 1b	22
2.2.5	Algorithms for QR decomposition	22
2.2.6	Bi-diagonalization of Triangular matrix	24
2.2.7	Diagonalization of Bi-diagonal matrix	24
2.3	Multiple Relatively Robust Representation (MRRR) algorithm	24
2.4	Divide and conquer algorithm	24
2.5	Bi-section and Inverse iteration	27
2.6	Hybrid methods for SVD	27
2.6.1	Introduction	27
2.6.2	Householder Bi-diagonalization followed by Jacobi	28
2.6.3	QR decomposition followed by Jacobi	28
2.7	Selection of an SVD algorithm for Pattern matching	28
3	CORDIC	31
3.1	Introduction	31
3.2	Modes of CORDIC	31
3.2.1	Rotation mode	32
3.2.2	Vectoring mode	33
3.3	Pipelined CORDIC	34
4	Jacobi Methods of SVD	35
4.1	Introduction	35
4.2	Classical Jacobi Algorithm	35
4.3	Two-sided Jacobi Method	35
4.3.1	Shuffling rotations	37
4.4	One-sided Jacobi Method	38
4.5	Systolic array implementation	39
4.6	Row ordering	39
4.7	Parallel ordering	40
5	Hardware implementation of Pattern Matching	42
5.1	Introduction	42
5.2	Previous work in Hardware implementations of SVD	42
5.3	Reading of historical window	42
5.4	Pre-processing	43
5.4.1	Recursive Mean and Standard deviation computation	43
5.5	Two-sided Jacobi algorithm	43
5.6	Sorting	44
5.7	Calculation of similarity factors	44
5.8	Working with Vivado HLS	44
5.8.1	HLS with precomputed dataset	45

5.9 Working with SDSoC	45
6 Future work and Conclusion	48
6.1 Conclusion	48
6.2 Future work	48
References	50

List of Figures

1.1	Fault Diagnosis Methods	1
1.2	Schematic layout of Tennessee Eastman Challenge Process	8
1.3	Computation time for various stages of pattern matching algorithm	10
2.1	Geometrical Interpretation of Householder reflection: H	18
2.2	Geometrical Interpretation of Householder reflection: H'	18
2.3	Geometrical Interpretation of Givens rotations	20
2.4	Comparison of various algorithms with snapshot as IDV1	29
2.5	Comparison of various algorithms with snapshot as IDV3	30
2.6	Worst absolute % rel. error of top 32 singular values	30
3.1	Modes of CORDIC	31
3.2	Illustration of pseudo-rotations	34
4.1	BLV array for $n=8$	39
4.2	Flow chart for Parallel ordering	41
5.1	Design flow with SDSoc	46
5.2	Software only performance for 300 windows	47
5.3	Software only performance for 400 windows	47

Chapter 1

Pattern Matching Approach

1.1 Overview of Fault Diagnosis Methods

Venkatasubramanian in [1] provided a comprehensive review of various methods of Fault diagnosis methods developed for process industries as shown in fig.1.1. From Table 1.1, it is clear that none of the fault diagnostic method possesses all the desirable characteristics. In Table 1.1, a \checkmark indicates that the property is satisfied by the method, while a \times indicates that the method doesn't satisfy the property and a ? indicates that satisfiability of the property is dependent on case study. Model-based approaches are not quite popular compared to statistical based methods in process industries because of the following reasons as stated in [2].

- While the theory of linear quantitative model-based approaches has its roots wide-spread, the design and implementation for nonlinear models is still an open issue.
- Since the models are restricted to linear domain, the advantages of a model-based approach over a simple statistical approach such as PCA might be minimal. Thus PCA-based approaches are easier for implementation than Model-based approaches.
- Most of the Model-based approaches are restricted to sensor and actuator failures.

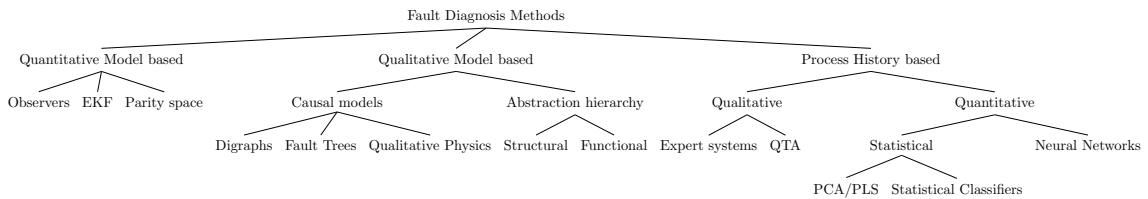


Figure 1.1: Fault Diagnosis Methods

Thus data-driven methods like PCA are much simpler in computation as they neither require an expert to build the models nor specialized tests for constructing models. This work assumes that a large history of the process variables with corresponding operation label is available for an existing process plant.

Table 1.1: Comparison of various Fault Diagnosis Methods

	Observer	Digraphs	Abstraction Hierarchy	Expert Systems	QTA	PCA	Neural Networks
Quick detection and diagnosis	✓	?	?	✓	✓	✓	✓
Isolability	✓	×	×	✓	✓	✓	✓
Robustness	✓	✓	✓	✓	✓	✓	✓
Novelty identifiability	?	✓	✓	×	?	✓	✓
Classification error	×	×	×	×	×	×	×
Adaptability	×	✓	✓	×	?	×	×
Explanation facility	×	✓	✓	✓	✓	×	×
Modelling requirement	?	✓	✓	✓	✓	✓	✓
Storage and computation	✓	?	?	✓	✓	✓	✓
Multiple fault identifiability	✓	✓	✓	×	×	×	×

1.2 Previous work

Ashish singhal et al. proposed a moving window approach for evaluating pattern matching in Tennessee Eastman process [3]. The measurements of the 52 variables over a period of time forms the snapshot dataset S . Let m be the number of samples in snapshot dataset. A moving window which is of same size as that of snapshot dataset is moved through the historical database and the two datasets are compared making use of similarity factors discussed in section 1.5. Ideally the moving window should replace the oldest sample with a new sample i.e window movement rate $w = 1$. But this increases computational effort and hence window movement rate has been chosen to be $w \simeq m/10$. Although the computational effort has been reduced, but this decreases the accuracy of the pattern matching approach.

1.3 Pre-processing of dataset

The snapshot dataset, S and the historical data window, H are pre-processed before computing singular value decomposition to find the Principal Components. The variables are scaled to zero mean and unit variance since these variables are distributed over a wide range of values.

1.4 Computation of PCA

PCA is widely used as a tool for dimensionality reduction and thus reducing the computational effort. PCA can be computed in a number of ways using SVD-Singular Value Decomposition, EVD-Eigen Value Decomposition and ALS-Alternating Least Squares algorithms. A brief overview of matrix decompositions is given below.

1.4.1 Singular value Decomposition

The Singular Value Decomposition of a matrix A is defined as factorization into a diagonal matrix of the form as shown in equation 1.1.

$$A = U\Sigma V^H \tag{1.1}$$

where Σ is $m \times n$ diagonal matrix containing singular values arranged in descending order, U is the left orthogonal matrix of order $m \times m$ and having singular vectors corresponding to $A.A^T$ and V is the right orthogonal matrix of order $n \times n$ and having singular vectors corresponding to $A^T.A$. Here V^H denotes the Hermitian transpose of the matrix V . If r is the rank of the matrix A and $r < n$

Table 1.2: Computation time for PCA using various algorithms

Algorithm	$svd(b^T.b)$	$svd(b,0)$	$svd(b)$	$pca(b)$	$eigs(b^T.b, 52, 'lm')$
Computation time in sec	7.6703×10^{-4}	0.0014	0.0063	0.0034	0.0012

then there will be only r non-zero singular values in diagonal matrix Σ . To improve execution time and for better storage, the zero singular values in the diagonal matrix Σ along with the unnecessary columns of U which multiply with these zeros are removed. This form of decomposition is called Economic SVD which can be found using $svd(A,0)$ in Matlab.

1.4.2 Eigen Value Decompositon

It is defined for only square matrices unlike SVD. Hence in the context of PCA, the EVD is performed on covariance/correlation matrix instead on dataset which is rectangular most of the time. The EVD of square matrix A of order n is defined as shown in equation 1.2.

$$A = P\Lambda P^{-1} \tag{1.2}$$

where P is the eigen vector matrix and Λ is a diagonal matrix containing eigen values of A . It is obvious that for mean centered data, eigen values(λ_i) of the covariance matrix of dataset and singular values(σ_i) of dataset, b are related as below.

$$\lambda_i = \sigma_i^2 / (m - 1) \tag{1.3}$$

In equation 1.3, m is the number of observations in the dataset b .

The eigen values corresponds to the variance explained by each Principal Component and the the eigen vector matrix or the right singular vector matrix V has the corresponding PC's arranged column wise. Table 1.2 shows the computational time (in seconds) using various approaches. After obtaining the PC's of two datasets, only k PC's which explain 95% variance in both the datasets S and H are chosen for comparison. Let k_s be the number of PC's required for explaining 95% variance in snapshot dataset, S and k_h be the number of PC's required for explaining 95% variance in historical dataset, H . Then $k = \max(k_s, k_h)$. The computation time for PCA reported in Table 1.2 is the average time taken by an approach over 1000 runs.

1.5 Similarity factors

Various similarity factors for the comparison of datasets as reported in [4] are presented here. A similarity factor should assign high value between same operating condition and should have low value for discriminating ability between two different operating conditions.

1.5.1 Standard PCA similarity factor

The geometrical interpretation for PCA similarity factor is as given in equation 1.4. It basically quantifies the angular difference between i^{th} PC of snapshot dataset and j^{th} PC of historical window. It can also be computed using the reduced sub-spaces of singular vector matrix of snapshot and

Table 1.3: Computation time for Standard PCA similarity factor

OpID	k	G.I:eq(1.4) (in sec)	Trace:eq(1.5) (in sec)	Simplified:eq(1.6) (in sec)	MAE_1	MAE_2
1	12	8.3122×10^{-4}	1.4082×10^{-5}	1.8309×10^{-5}	1.2212×10^{-15}	1.4433×10^{-15}
2	23	0.0030	3.7025×10^{-5}	3.2162×10^{-5}	2.3315×10^{-15}	2.1094×10^{-15}
3	32	0.0059	5.5679×10^{-5}	5.0411×10^{-5}	3.7748×10^{-15}	3.8858×10^{-15}

historical window as shown in equation 1.5.

$$S_{PCA} = \frac{1}{k} \sum_{i=1}^k \sum_{j=1}^k \cos^2 \theta_{ij} \quad (1.4)$$

$$S_{PCA} = \frac{\text{trace}(L^T M M^T L)}{k} \quad (1.5)$$

where L and M are the reduced sub-space containing first k principal components in snapshot window and historical window respectively. For reducing the latency in the matrix multiplications, the formula has been modified as given in equation 1.6. No significant loss of accuracy (differs in 15th or 16th decimal place) was observed with the given formula when compared with formulas quoted in equations 1.4 and 1.5.

$$S'_{PCA} = \frac{1}{k} \sum_{\text{all } i,j \text{ pairs}} (L^T M) \cdot \wedge 2 \quad (1.6)$$

where $\cdot \wedge$ represents an element-wise power operation and other symbols carry the same meaning as that of equation 1.5. Table 1.3 shows the computation time of standard PCA using trace, geometrical and simplified approach using Matlab 2017b. For comparing the performance, the snapshot with 100% fault samples from Instance-2 of testing dataset has been taken from operation ID's 1,2 and 3 and the total time taken for whole historical run is averaged out to compare per unit window computation time. The maximum absolute difference between the computed similarity factor from geometrical interpretation and the other two approaches namely using Trace and simplified formulae have been reported as MAE_1 and MAE_2 .

1.5.2 Modified PCA similarity factor

The standard PCA similarity factor doesn't give any weightage to the variance explained by each principal component and weights all the PC's equally. A modified PCA similarity factor proposed by Johannesmeyer can be more effective which weighs each PC by square-root of its eigen value. The geometrical interpretation for modified PCA similarity factor is as given in equation 1.7.

$$S_{PCA}^\lambda = \frac{\sum_{i=1}^k \sum_{j=1}^k (\lambda_i^S \lambda_j^H) \cos^2 \theta_{ij}}{\sum_{i=1}^k \lambda_i^S \lambda_i^H} \quad (1.7)$$

$$S_{PCA}^\lambda = \frac{\text{trace}(R^T T T^T R)}{\sum_{i=1}^k \lambda_i^l \lambda_i^m} \quad (1.8)$$

where

$$\begin{aligned} R &= L\Lambda_l \\ T &= M\Lambda_m \end{aligned} \quad (1.9)$$

and

$$\Lambda = \begin{bmatrix} \sqrt{\lambda_1} & 0 & 0 & 0 \\ 0 & \sqrt{\lambda_2} & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & \sqrt{\lambda_k} \end{bmatrix} \quad (1.10)$$

1.5.3 Distance similarity factor

Distance similarity factor helps in identifying the datasets that have similar principal components but the values of the process variables are quite different because of the disturbances of varying magnitudes or set point changes. Thus the datasets which have same spatial orientation but are located far apart can be identified using Distance similarity factor. The Mahalanobis distance, ϕ is given by,

$$\phi = \sqrt{(\bar{x}_H - \bar{x}_S)\Sigma_S^{*-1}(\bar{x}_H - \bar{x}_S)^T} \quad (1.11)$$

where \bar{x}_S and \bar{x}_H is the mean of the snapshot and historical windows respectively and Σ_S^{*-1} is the pseudo-inverse of the covariance matrix Σ_S of snapshot dataset. Only k ($=\max(k_s, k_h)$) singular values are used while calculating pseudo-inverse. The distance similarity factor is defined as the probability that center of the historical dataset \bar{x}_H is atleast ϕ distance from the snapshot dataset S .

$$S_{dist} \triangleq 2 \frac{1}{\sqrt{2\pi}} \int_{\phi}^{\infty} e^{-z^2/2} dz \quad (1.12)$$

1.5.4 Dissimilarity factor

Kano et al. proposed a Dissimilarity factor for comparing two datasets. Let H be the historical moving window dataset and S be the current snapshot dataset. The augmented dataset X is formed as shown below.

$$X \triangleq \begin{bmatrix} H \\ S \end{bmatrix} \quad (1.13)$$

The eigen value decomposition is then performed on the covariance matrix of this augmented dataset, X . The datasets H and S are then projected on to eigen vector matrix of covariance of X and are scaled by corresponding eigen values to produce transformed datasets \tilde{H} and \tilde{S} . Then the eigen decomposition of the covariance of transformed datasets is then performed. It has been showed that the eigen vectors of the transformed datasets are same and the corresponding eigen values are related as shown below.

$$\lambda_j^{\tilde{H}} = 1 - \lambda_j^{\tilde{S}} \quad (1.14)$$

where $\lambda_j^{\tilde{H}}$ and $\lambda_j^{\tilde{S}}$ are the eigen values of the transformed dataset. If the datasets are similar then their eigen values are close to 0.5 while if they are dissimilar then the smallest and largest eigen values will be close to 1 and 0 respectively. Finally the dissimilarity factor signifies how the eigen

values deviate from the central value of 0.5.

$$D \triangleq \frac{4}{n} \sum_{j=1}^n (\lambda_j^{\tilde{H}} - 0.5)^2 = \frac{4}{n} \sum_{j=1}^n (\lambda_j^{\tilde{S}} - 0.5)^2 \quad (1.15)$$

where n is the number of variables in each dataset. If two datasets are dissimilar then the dissimilarity factor, D will be close to 1 while if they are similar then it will be close to 0.

1.6 Metrics for Pattern Matching

Disturbances IDV(3-5) and IDV15 are quite difficult to locate in the historical database as they show high similarity factors with normal operation and also the misclassification is high among such disturbances. So to ensure proper detection, a candidate pool has been formed which is rank ordered based on the values of similarity factor. The windows that differ by m samples are deleted and only the window with highest similarity factor is retained in candidate pool to avoid repeated counting of the same instance. The windows collected in the candidate pool are called records.

N_P : Size of the candidate pool.

N_1 : Number of correctly identified records

N_2 : Number of incorrectly identified records

N_{DB} : Number of historical windows that are actually present in the database similar to snapshot window

1.6.1 Pool Accuracy

Pool accuracy gives a measure of the perfection of the candidate pool.

$$p \triangleq \frac{N_1}{N_P} \times 100\% \quad (1.16)$$

1.6.2 Pattern Matching Efficiency

Pattern Matching Efficiency η evaluates the effectiveness of a pattern matching methodology in identifying the similar instances in a historical database.

$$\eta \triangleq \frac{N_1}{N_{DB}} \times 100\% \quad (1.17)$$

Since an effective pattern matching approach should possess high values of Pool accuracy p and pattern matching efficiency η , the mean of the two quantities, ζ is used as a parameter for overall effectiveness of pattern matching.

$$\zeta = \frac{\eta + p}{2} \quad (1.18)$$

1.7 Tennessee Eastman Challenge Process

1.7.1 Introduction

Tennessee Eastman Challenge process is a challenging and popular case study in the domain of process control developed by Tennessee Eastman company. TE process is considered as a case study for fault diagnosis because of the following reasons.

- It resembles an actual chemical plant
- It is open loop unstable plant and is highly non-linear process
- The process comprises of large number of variables compared to other case studies like CSTR and fermentation process
- Highly interacting systems and popular case study

Fig. 1.2 shows the schematic layout of TE process with base control strategy. The plant produces two products G and H from four reactants namely A,C,D and E. An inert component B, is also present in the reaction mixture which enters mainly through stream C and in trace amounts from stream A. Although there are seven operating modes of the plant, only the base operating condition proposed by Mc Avoy and Ye is only considered. The plant provides a total of 41 measurements and 12 manipulated variables. The 12th manipulated variable which is agitator speed is held constant and is treated as a free variable. The historical database was generated by simulating the plant for over a period of 3000 days. This simulation resulted in over 4.32 million measurements for each of the 52 variables(41 measurements and 11 manipulated variables). The historical database contains a total of 386 instances of 20 disturbances IDV(1-20) and 80 instances of four setpoint changes SP(1-4) and normal operation. Table 1.4 shows the description of various operating IDs.

1.8 Proposed Pattern matching approach

Since to avoid computation of principal components by moving at one sample ahead when the similarity factor is low, the window movement rate can be accelerated by skipping few data points and then bring back to normal rate of $w = 1$ when the similarity factor is above certain threshold(say 0.5). From Table 1.2, it is clear that SVD of covariance matrix of dataset b i.e $\text{svd}(b^T.b)$ is quite computationally faster compared to other approaches and hence it is chosen for calculation of PCs. From Table 1.3 it is clear that trace approach outperforms the geometrical interpretation for calculation of similarity factors. We fix the number of PC's based on snapshot instead of the $k_{max} = \max(k_s, k_h)$. Although other similarity factors like Distance and Dissimilar factor can be much more efficient in pattern matching but they are computationally more expensive as compared to Standard similarity factor. No significant improvement in reducing the number of mis-classifications is observed with modified PCA similarity factor as reported in [3]. Hence Standard similarity factor has been chosen for initial study. In fig.1.3, we fed 5 operation conditions from Instance-2 of testing dataset with snapshot as fully fault samples. With change in snapshot, only the time spent in similarity factor calculation varies because of the change in the number of PC's required for capturing the 95% in the snapshot dataset. With a window size of $m=500$, on an average it takes around 0.8-1 milli-second

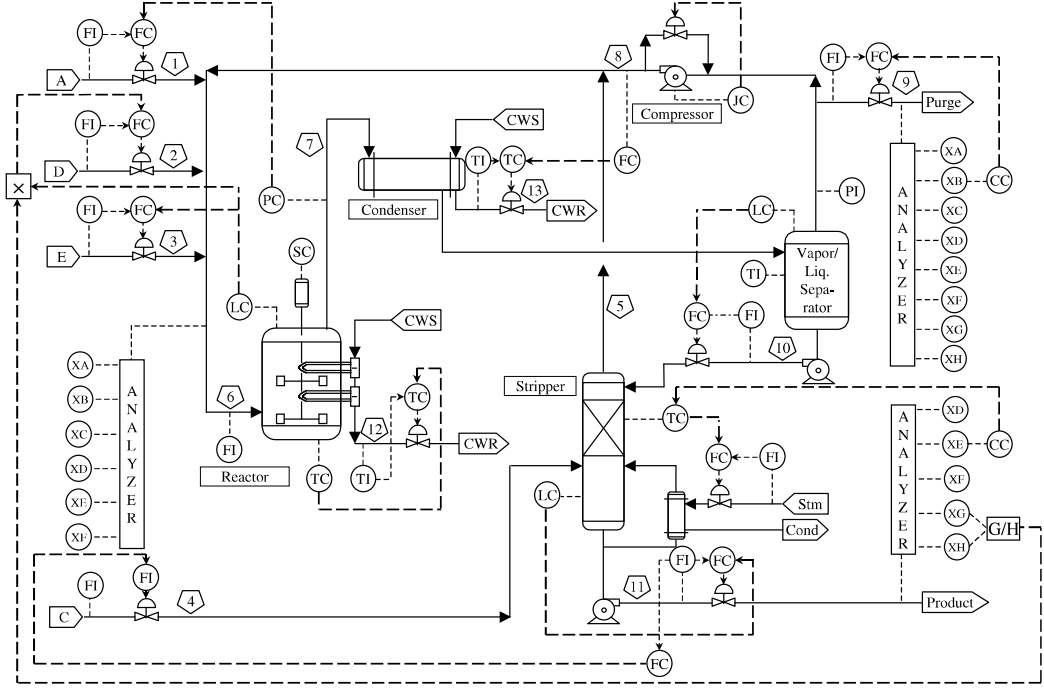


Figure 1.2: Schematic layout of Tennessee Eastman Challenge Process

(for a single window) for the 4-stages namely preprocessing, covariance $b^T \cdot b$ product calculation, svd computation and for calculation of similarity factor. It took not more than 40 minutes to sweep through the entire training database(21.60lakh) irrespective of the operating condition. From the fig.1.3 it is clear that the computation of SVD is quite dominant over other stages and hence to reduce this we go for hardware implementation of SVD. The simulation studies are performed on a system with 2.8GHz Intel i5 processor, 4GB DDR3 RAM running Matlab 2018a.

1.8.1 Selection of window size

The selected window size, m has to be properly chosen such that it captures the principal components well so as to give a high value of similarity factor between similar operating conditions. Hence, the window size has been chosen to be 500 by method of trial and error as it gave a high similarity factor even between two similar transition operating windows.

1.8.2 Pre-computing of Historical Database

Since the historical database is fixed, the whole computations like pre-processing and SVD remains the same and only the similarity factor computation has to be done with change in the snapshot. Hence the principal components of the historical windows required can be pre-computed and stored instead of storing the raw data. A major disadvantage of pre-computation of whole historic database is that it restricts the process to be Time-invariant as in real time, there will be drifts in the sensors due to aging and process dynamics also changes with time. Hence the mean and standard deviation computation along with the number of principal components chosen have to be updated as a part of model-maintenance (Adaptive PCA methods) as mentioned in [5]. Another disadvantage with pre-

Table 1.4: Operating conditions of TE process with base control strategy

OpID	Operating condition	Description
N	Normal operation	No disturbance or setpoint changes
1	IDV(1)	Step in A/C feed ration, B composition constant
2	IDV(2)	Step in B composition, A/C ratio constant
3	IDV(3)	Step in D feed temperature (stream 2)
4	IDV(4)	Step in reactor cooling water inlet temperature
5	IDV(5)	Step in condenser cooling water inlet temperature
6	IDV(6)	A feed loss(step change in stream 1). Switch pressure controller to purge stream and reduce production rate by 23.8%. Maximum disturbance duration = 72 h
7	IDV(7)	C header pressure loss-reduced availability (step change in stream 4)
8	IDV(8)	Random variation in A-C feed composition (stream 4)
9	IDV(9)	Random variation in D feed temperature (stream 2)
10	IDV(10)	Random variation in C feed temperature (stream 4)
11	IDV(11)	Random variation in reactor cooling water inlet temperature
12	IDV(12)	Random variation in condenser cooling water inlet temperature
13	IDV(13)	Slow drift in reaction kinetics. Maximum disturbance duration = 48 h
14	IDV(14)	Sticking reactor cooling water valve
15	IDV(15)	Sticking condenser cooling water valve
16	IDV(16)	Unknown disturbance. Maximum disturbance duration = 48 h
17	IDV(17)	Unknown disturbance. Maximum disturbance duration = 48 h
18	IDV(18)	Unknown disturbance. Maximum disturbance duration = 12 h
19	IDV(19)	Unknown disturbance
20	IDV(20)	Unknown disturbance. Maximum disturbance duration = 8 h
21	SP(1)	Production rate change (step down 15%)
22	SP(2)	Production mix change (50/50 to 40/60)
23	SP(3)	Reactor operating pressure change (step down by 60kPa)
24	SP(4)	Purge gas: Component B change (step up 2%)

computation of historical database is large memory requirement and frequent flushing and loading of volatile memory is dominant time consuming task as every 1lakhx52 database gives rise to 52lakhx35 database assuming we store only top-35 principal components.

1.8.3 Clustering

The historical data can be clustered into groups using the metrics like similarity factor as in [6] where the authors modified k - means clustering algorithm to cluster time series data using Mahalonobis distance and Standard PCA similarity factors. But since there are significant number of misclassification as seen in results, it leads to poor accuracy of the formed clusters. Although supervised clustering, where the windows which corresponds to same operation or has identical behavior can be grouped together but in practice the labeling of data is quite difficult.

1.8.4 Overview of Historical Data

There are a total of 55 variables being sampled for every 1minute duration. Variable 1 corresponds to the time while variables 2-53 are the variables under consideration for pattern matching. Variable 55 corresponds to the operation number being simulated at that point of time. The process data is highly ill-conditioned. Although the smaller singular values in such a kind of matrices is difficult to be determined accurately, in our pattern matching approach we deal with only those singular values which capture 95% of variance of dataset. But the singular values taken under consideration should be accurately determined for the vectors(PC's) to be determined accurately because a small perturbation in the singular values gives larger deviations in the corresponding vectors in case of

ill-conditioned matrices. Some of the variables remained constant for a period of time (due to saturation) which makes the standard deviation to become zero, hence those windows are discarded as it results in a divide by zero during preprocessing step.

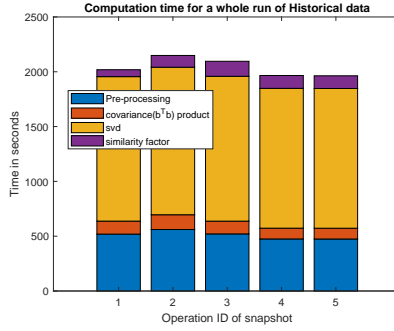


Figure 1.3: Computation time for various stages of pattern matching algorithm

1.9 Results for pattern matching

1.9.1 Methodology for Pattern Matching

The entire historical database has been divided into two equal halves (each of size 21.6 lakh) as training and testing datasets. Each operation has been simulated for atleast 8 times in both testing and training datasets. To evaluate the proposed pattern matching approach, every instance of fault operation present in testing dataset has been taken as a snapshot data. The pool size has been taken as 5 and the final decision is made on the most repeated detection out of these top-5 windows. We made use of Matlab's mode function for judging the operating condition based on the candidate pool. An operation which has been detected for more than or equal to 3 times in a candidate pool of size 5 can be diagnosed accordingly. In case of a candidate pool with top 5 records as N, F_x, F_y, N, F_z the operation will be still treated as Normal because of its clear dominance over other kinds of faults x,y and z. Hence the operator can be alerted to re-examine the process for any faults. No weightage has been given to the records of the candidate pool as the snapshot has transition data and misclassified windows are present even at top positions. The detected windows are then judged by the presence of any fault operation samples. It has been observed that with a snapshot window of only few (10% or 50%)fault samples, the detected windows may be similar to windows with 90% fault samples or windows with 100% fault samples or even identical to snapshot data (10% fault samples). Hence a window has been considered correctly classified, if the detected window possess any amount of same operation faulty samples. Since the work is concerned about Fault detection and diagnosis, we feed only the windows with 10%, 50 % and 100% fault samples windows from the testing dataset as snapshot data. A total of 156 windows of size 500 with fully normal operation are also fed to judge the performance of the proposed method. The detailed report with top-5 records for each operation from Instances 2-7 can be found in *report_fdi.pdf*.

The overall accuracy of the pattern matching for fault detection and diagnosis with faulty samples has been $\frac{113}{144} = 78.47\%$. Excluding the fault ID's 3,4,5 and 15 which are difficult to detect as mentioned in [3], the accuracy has been $\frac{107}{120} = 89.16\%$. Table 1.5,1.6 and 1.7 lists the detected operations for different kinds of faults in three scenarios 10,50 and 100% fault samples respectively.

Table 1.5: Results for snapshot as 10% fault samples for Instances I2-I7

Op ID	I2	I3	I4	I5	I6	I7	Detected
1	1	1	1	1	1	1	6/6
2	2	12	2	25	5	25	2/6
3	3	25	25	25	25	25	1/6
4	25	17	11	14	11	11	0/6
5	25	25	25	3	25	25	0/6
6	6	6	6	6	6	6	6/6
7	7	7	7	7	7	7	6/6
8	1	1	25	1	25	8	1/6
9	25	25	25	4	25	3	0/6
10	25	25	25	25	1	25	0/6
11	14	4	25	25	11	11	2/6
12	25	25	5	12	3	25	1/6
13	25	25	13	2	25	25	1/6
14	25	25	25	14	25	25	1/6
15	25	25	25	25	25	25	0/6
16	25	25	25	25	16	25	1/6
17	3	17	25	17	17	25	3/6
18	25	25	25	3	25	25	0/6
19	25	25	25	25	25	25	0/6
20	25	20	15	9	25	9	1/6
21	7	25	25	25	25	25	0/6
22	22	22	22	22	22	22	6/6
23	23	23	23	12	23	17	4/6
24	24	25	24	24	24	24	5/6

Table 1.6: Results for snapshot as 50% fault samples for Instances I2-I7

Op ID	I2	I3	I4	I5	I6	I7	Detected
1	1	1	1	1	1	1	6/6
2	2	2	2	2	2	2	6/6
3	25	25	25	25	25	25	0/6
4	11	7	4	11	4	11	2/6
5	3	25	25	25	25	25	0/6
6	6	6	6	6	6	6	6/6
7	7	7	7	7	7	7	6/6
8	8	8	8	8	8	8	6/6
9	25	25	25	25	25	25	0/6
10	10	10	10	10	10	10	6/6
11	11	11	11	11	11	11	6/6
12	12	12	12	12	12	12	6/6
13	13	13	13	13	13	13	6/6
14	14	14	14	14	24	3	4/6
15	25	25	25	15	7	25	1/6
16	16	16	16	25	16	16	5/6
17	17	17	17	17	17	17	6/6
18	18	18	18	18	18	18	6/6
19	25	25	25	7	25	25	0/6
20	20	20	20	20	20	20	6/6
21	25	21	21	21	21	21	5/6
22	22	22	22	22	22	22	6/6
23	23	23	23	23	23	23	6/6
24	24	24	24	24	24	24	6/6

Table 1.7: Results for snapshot as 100% fault samples for Instances I2-I7

Op ID	I2	I3	I4	I5	I6	I7	Detected
1	1	1	1	8	1	1	5/6
2	2	2	2	2	2	2	6/6
3	25	25	25	25	25	25	0/6
4	11	7	25	25	25	11	0/6
5	5	5	9	25	25	25	2/6
6	6	6	6	6	6	6	6/6
7	7	7	7	7	7	7	6/6
8	8	8	8	8	8	8	6/6
9	25	25	25	25	25	24	0/6
10	10	10	10	10	10	10	6/6
11	11	11	11	11	11	11	6/6
12	12	12	12	12	12	12	6/6
13	13	13	13	13	13	13	6/6
14	14	14	14	14	14	14	6/6
15	25	25	25	25	7	25	0/6
16	16	16	16	16	16	16	6/6
17	17	17	17	17	17	17	6/6
18	18	18	18	18	18	18	6/6
19	24	1	25	7	25	25	0/6
20	20	20	20	20	20	20	6/6
21	7	7	21	21	7	7	2/6
22	22	22	22	22	22	22	6/6
23	23	23	23	23	23	23	6/6
24	24	2	24	24	25	25	3/6

Table 1.8: Overall results of fault diagnosis and detection

Op ID	I2	I3	I4	I5	I6	I7	10%	50%	100%
1	10,50,100	10,50,100	10,50,100	10,50	10,50,100	10,50,100	6	6	5
2	10,50,100	50,100	10,50,100	50,100	50,100	50,100	2	6	6
3	10						1	0	0
4			50		50		0	2	0
5	100	100					0	0	2
6	10,50,100	10,50,100	10,50,100	10,50,100	10,50,100	10,50,100	6	6	6
7	10,50,100	10,50,100	10,50,100	10,50,100	10,50,100	10,50,100	6	6	6
8	50,100	50,100	50,100	50,100	50,100	10,50,100	1	6	6
9							0	0	0
10	50,100	50,100	50,100	50,100	50,100	50,100	0	6	6
11	50,100	50,100	50,100	50,100	10,50,100	10,50,100	2	6	6
12	50,100	50,100	50,100	10,50,100	50,100	50,100	1	6	6
13	50,100	50,100	10,50,100	50,100	50,100	50,100	1	6	6
14	50,100	50,100	50,100	10,50,100	100	100	1	4	6
15				50			0	1	0
16	50,100	50,100	50,100	100	10,50,100	50,100	1	5	6
17	50,100	10,50,100	50,100	10,50,100	10,50,100	50,100	3	6	6
18	50,100	50,100	50,100	50,100	50,100	50,100	0	6	6
19							0	0	0
20	50,100	10,50,100	50,100	50,100	50,100	50,100	1	6	6
21		50	50,100	50,100	50	50	0	5	0
22	10,50,100	10,50,100	10,50,100	10,50,100	10,50,100	10,50,100	6	6	6
23	10,50,100	10,50,100	10,50,100	50,100	10,50,100	50,100	4	6	6
24	10,50,100	50	10,50,100	10,50,100	10,50	10,50	5	6	3

Table 1.9: Misclassifications with snapshot as Normal data

Misclassified as	1	3	4	7	15	24
No. of misclassification (window number)	3 (22,126,147)	6 (38,60,77,81,105,138)	3 (20,53,82)	1 (139)	3 (9,48,95)	3 (45,148,154)

Table 1.8 lists the overall results of the proposed fault detection and diagnosis approach. Each cell is populated with number 10 and/or 50 and/or 100 to signify that it has been properly classified with a snapshot of 10% , 50% and 100% fault samples in each of the 6 instances I2-I7. An empty cell signifies that it has not been detected in any of the 10% , 50% and 100% fault samples snapshot. We relax the condition for fault detection and diagnosis, that a fault has to be correctly classified atleast once in any of the 10% , 50% and 100% fault samples scenarios. Because a fault once detected in 10% fault samples simulation will no longer propagate to give rise to a situation of 50% or 100% fault sample scenarios assuming the detection and diagnosis is very much faster than the process to reach those scenarios. Table 1.9 reports the mis-classifications reported with snapshot as fully normal samples.

Apart from the most difficult fault ID's (3,4,5 & 15) reported in [3], faults 9 and 19 are also difficult to detect with a window size of 500. No improvement has been seen in reducing the misclassifications in Op ID's 9 and 19 even with windows as large as 2000. The two operations showed high value of similarity with normal operation. Op ID 20, the fault with smallest maximum duration of 8 hours has been correctly classified in 50% and 100 % fault snapshots in all the instances and hence there has been no issues with the proposed method of not able to detect a fault before maximum duration leading to catastrophic damages. Op ID's 1,6,7 and 22 are well classified even with 10% fault samples in snapshot windows. Similarly the fault ID's 2,8,10-13,17,18,20-24 are well classified by the time fault has been simulated for 4 hours(50% of snapshot) while Fault ID 16 has been well classified in the snapshot with fully faulty samples. The accuracy for the proposed fault detection and diagnosis with snapshot data as a window with fully normal operation samples has been found to be $\frac{137}{156} = 87.82\%$. Excluding the operation ID's 3,4,5 and 15 in the misclassifications, the accuracy is around $\frac{149}{156} = 95.512\%$ for normal operation windows. 8 out of 24 operating conditions have similarity factor of less than 0.5 for most(> 80%) of the time and hence in those cases the window movement rate can be made to accelerate by skipping 40-50 windows until the similarity factor reaches the threshold of 0.5. Longer jumps in the window are very dangerous as it has been examined that the closely matching windows are having a higher similarity factor of around 0.9 for only 50 samples.

Chapter 2

Algorithms for SVD

2.1 Introduction

Since we require SVD for computing Principal Components of the datasets as mentioned in the previous chapter, various algorithms for SVD are briefly discussed here. The most popular SVD algorithm for hardware implementations i.e Jacobi method is discussed separately in Chapter 4.

2.1.1 Properties of SVD

- The singular values of a symmetric matrix are the absolute values of the eigen values. Further the right and left orthogonal matrices are just transpose of each other.
- Singular values are always non-negative.
- SVD is unaltered by shuffling of rows.
- Scaling the input matrix by a factor of k , will also scale the singular values by the same factor k without any change in the singular vectors.

2.2 Golub-Kahan-Reinsch SVD:

The algorithm is a two step process which first bi-diagonalizes the given matrix using orthogonal transformations like Givens rotations or Householder reflections. Then the bidiagonal matrix is further diagonalized using Givens rotations. Two variants of Golub-Kahan algorithm are here described as Algorithm 1a and 1b respectively.

2.2.1 Algorithm 1a

For a matrix A with m rows and n columns, this algorithm essentially involves two steps.

For $m \geq n$:

- Step-1: Reduce the given matrix A into bi-diagonal form using orthogonal transformations like Householder reflections.
- Step-2: Diagonalize the matrix obtained from Step-1 using Givens rotations.

For $m < n$:

- Let $A_1 = A^T$, Hence A_1 is transformed to a thin matrix and proceed as for $m \geq n$.
- Take the transpose of the obtained decomposition of matrix A_1 which equals the decomposition for A .

The matlab code for computing svd using algorithm 1a can be found in *svd_2a.m* .

2.2.2 Householder Reflections

Householder reflections are orthogonal transformations and hence they preserve the norm of the vector. The method of forming householder matrices has been taken from [7].

$$w = \frac{1}{\sqrt{2r(r+|x_k|)}} \begin{bmatrix} 0 \\ \vdots \\ 0 \\ x_k + sr \\ x_{k+1} \\ \vdots \\ x_n \end{bmatrix}, \quad s = \begin{cases} \frac{x_k}{|x_k|} & \text{if } x_k \neq 0 \\ 1 & \text{otherwise} \end{cases} \quad (2.1)$$

$$r = \sqrt{|x_k|^2 + |x_{k+1}|^2 + \dots + |x_n|^2} \quad \text{and} \quad H = I - 2ww^T \quad (2.2)$$

Geometrical Interpretation: The concept of reflection is governed by 3 equations:
The reflection plane is determined by a vector w of unit length.

$$\|w\| = 1 \quad (2.3)$$

Reflection preserves the norm of the vector.

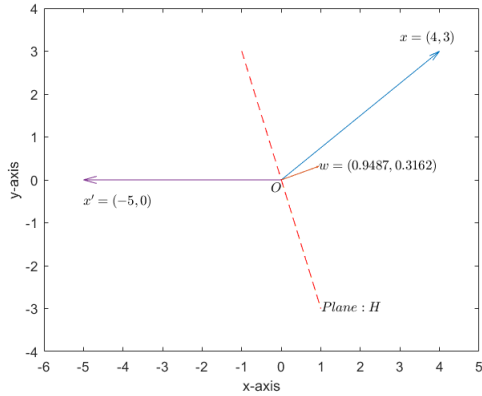
$$\|x\| = \|Hx\| \quad (2.4)$$

The difference between the vector x and its reflection x' is a scalar multiple of vector w .

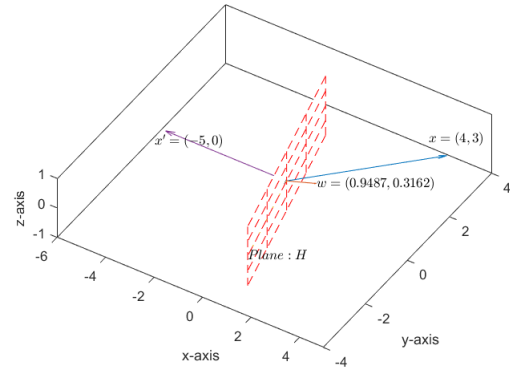
$$x - x' = fw \quad (2.5)$$

Consider a vector $x = \begin{bmatrix} 4 \\ 3 \end{bmatrix}$, we form a vector $w = \begin{bmatrix} 0.9487 \\ 0.3162 \end{bmatrix}$ of unit length as described in equation 2.1. The plane used for reflection H is formed orthogonal to the vector w as shown in fig.2.1. The plane H reflects the vector x on negative x-axis. If vector w' is chosen in such a way that it is perpendicular to computed w from equation 2.1, then there exists an another plane H' which reflects the vector x on positive x-axis as shown in fig.2.2.

For Golub-Kahan SVD, we pre-multiply a householder matrix with matrix A such that it preserves

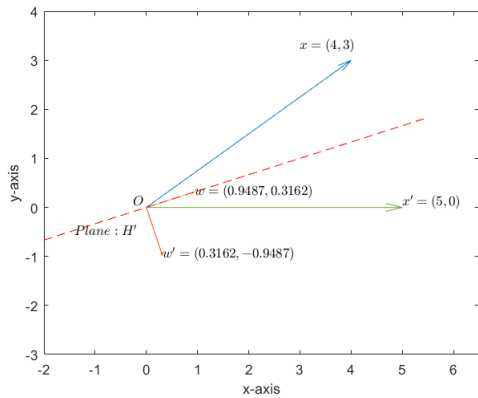


(a) 2D view

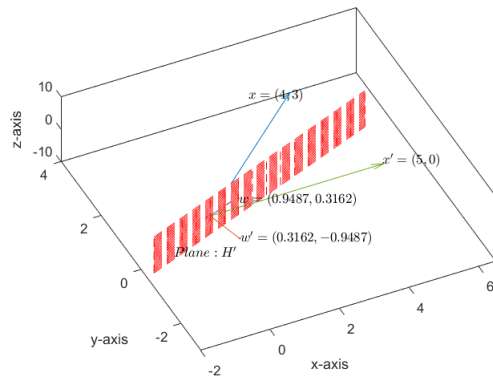


(b) 3D view

Figure 2.1: Geometrical Interpretation of Householder reflection: H



(a) 2D view



(b) 3D view

Figure 2.2: Geometrical Interpretation of Householder reflection: H'

only k components in k th column. Similarly, for each row we form a householder matrix which is used to post-multiply with A such that it retains only $k + 1$ components in k th row.

Consider the following matrix $A = \begin{bmatrix} 6 & 1 & 2 \\ 5 & -3 & 9 \\ 1 & 4 & 5 \\ -2 & 5 & 0 \end{bmatrix}$

The householder matrix to nullify all the elements of column 1 is formed as shown below.

$$x = \begin{bmatrix} 6 \\ 5 \\ 1 \\ -2 \end{bmatrix}, \quad r = \sqrt{6^2 + 5^2 + 1^2 + (-2)^2} = 8.1240, \quad w = \frac{1}{\sqrt{229.4868}} \begin{bmatrix} 6 + (1 \times 8.124) \\ 5 \\ 1 \\ -2 \end{bmatrix} = \begin{bmatrix} 0.9323 \\ 0.3301 \\ 0.0660 \\ -0.1320 \end{bmatrix}$$

$$H = I - (2ww^T) = \begin{bmatrix} -0.7385 & -0.6155 & -0.1231 & 0.2462 \\ -0.6155 & 0.7821 & -0.0436 & 0.0872 \\ -0.1231 & -0.0436 & 0.9913 & 0.0174 \\ 0.2462 & 0.0872 & 0.0174 & 0.9651 \end{bmatrix}$$

Pre-multiplying H with A results in a matrix A_1 given by

$$A_1 = H \times A = \begin{bmatrix} -8.1240 & 1.8464 & -7.6317 \\ 0.0000 & -2.7004 & 5.5903 \\ 0.0000 & 4.0599 & 4.3181 \\ -0.0000 & 4.8802 & 1.3681 \end{bmatrix}$$

Next we form householder matrix H_1 which when post-multiplied with A_1 , annihilates all the elements next to the super-diagonal entry in the first row.

$$A_2 = A_1 \times H_1 = \begin{bmatrix} -8.1240 & -7.8518 & -0.0000 \\ 0.0000 & 6.0686 & -1.3101 \\ 0.0000 & 3.2423 & 4.9615 \\ -0.0000 & 0.1781 & 5.0640 \end{bmatrix}$$

This process when repeated for all columns gives a bi-diagonal matrix B which is utilized in Step-2. The final bi-diagonal matrix B for the example considered is given by

$$B = \begin{bmatrix} -8.1240 & -7.8518 & 0.0000 \\ -0.0000 & -6.8827 & 1.3131 \\ 0.0000 & -0.0000 & 7.0889 \\ 0.0000 & -0.0000 & 0.0000 \end{bmatrix}$$

2.2.3 Givens Rotations

Let a and b be the x and y components of the given vector. The convention followed in forming the rotation matrices is as shown below.

For Row vector:

$$\begin{bmatrix} a & b \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} = \begin{bmatrix} r & 0 \end{bmatrix}$$

where $r = \sqrt{a^2 + b^2}$
 $c \leftarrow a/r$
 $s \leftarrow -b/r$

For column vector:

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

where $r = \sqrt{a^2 + b^2}$
 $c \leftarrow a/r$
 $s \leftarrow -b/r$

The Rotation matrix $R(i, j, \theta)$ is constructed in such a way that it acts only on i th and j th rows or columns of matrix A and the remaining elements are left unaltered. A typical rotation matrix to

act on i, j columns of A is as shown below.

$$R(i, j, \theta) = \begin{bmatrix} 1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \cdots & c & \cdots & -s & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & s & \cdots & c & \cdots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 0 & \cdots & 1 \end{bmatrix}$$

Geometrical Interpretation: Consider a vector $a = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$ as shown in fig.2.3a. The rotation

matrix R is found to be $R = \begin{bmatrix} 0.4472 & 0.8944 & 0 \\ -0.8944 & 0.4472 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ which annihilates the y-component and maps an equivalent vector on xz plane as shown in fig.2.3b. A rotation is equivalent to two reflections.

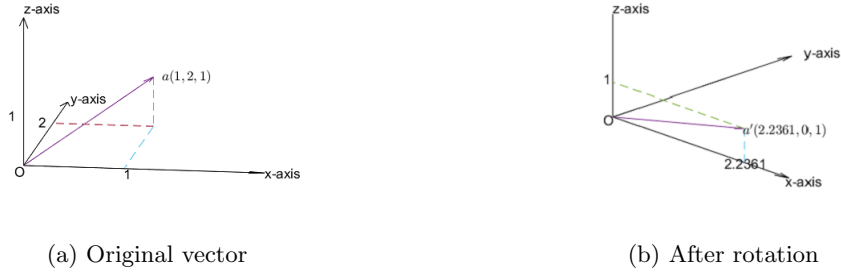


Figure 2.3: Geometrical Interpretation of Givens rotations

Consider the bi-diagonal matrix B obtained above after Step-1.

$$B = \begin{bmatrix} -8.1240 & -7.8518 & 0.0000 \\ -0.0000 & -6.8827 & 1.3131 \\ 0.0000 & -0.0000 & 7.0889 \\ 0.0000 & -0.0000 & 0.0000 \end{bmatrix}$$

We form a rotation matrix R_1 that acts on row-1 and annihilates the first element of the superdiagonal.

$$R_1 = \begin{bmatrix} -0.7190 & 0.6950 & 0 \\ -0.6950 & -0.7950 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Post-multiplying with rotation matrix R_1 results in matrix B_1 ,

$$B_1 = B \times R_1 = \begin{bmatrix} 11.2983 & 0 & 0.0000 \\ 4.7832 & 4.9490 & 1.3131 \\ -0.0000 & 0.0000 & 7.0889 \\ -0.0000 & 0.0000 & 0.0000 \end{bmatrix}$$

Since this lead to generation of a new non-zero element in column-1, we generate a rotation matrix R_2 that acts on column-1.

$$R_2 = \begin{bmatrix} 0.9209 & 0.3899 & 0 & 0 \\ -0.3899 & 0.9209 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Pre-multiplying R_2 with B_1 ,

$$B_2 = R_2 \times B_1 = \begin{bmatrix} 12.2691 & 1.9294 & 0.5119 \\ 0 & 4.5574 & 1.2092 \\ -0.0000 & 0.0000 & 7.0889 \\ -0.0000 & 0.0000 & -0.0000 \end{bmatrix}$$

Even though, this leads to new non-zero element at position (1,3), we proceed to apply rotation matrix R_3 on second row of B_2 which takes care of it.

$$R_3 = \begin{bmatrix} 1.0000 & 0 & 0 \\ 0 & 0.9666 & -0.2565 \\ 0 & 0.2565 & 0.9666 \end{bmatrix}$$

Post-multiplying with rotation matrix R_3 results in matrix B_3

$$B_3 = B_2 \times R_3 = \begin{bmatrix} 12.2691 & 1.9962 & 0.0000 \\ 0 & 4.7151 & -0.0000 \\ -0.0000 & 1.8180 & 6.8518 \\ -0.0000 & 0.0000 & 0.0000 \end{bmatrix}$$

The above process is repeated until all the entries on the super-diagonal are annihilated. The number of non-zero elements in super-diagonal elements tends to decrease as the iterations progresses. The indices of first and last non-zero element in super-diagonal are updated after every iteration to avoid applying Givens rotation to the elements which are already zero. A reduction in number of Givens rotation with index upgradation can be seen in Table 2.1. The computation of c and s required for forming rotation matrix can be found using *givens.m*. One more efficient way of diagonalization is through Fast Givens rotation which reduces the number of multiplications by half and eliminates the use of square root has been proposed in [8].

Table 2.1: Comparison of Givens rotations with and without Index upgradation

Order	Without Index Upgradation		With Index Upgradation	
	No. of Givens rotations	Overall Computation time(s)	No. of Givens rotations	Overall Computation time(s)
10	2034	0.154	1226	0.102
100	36828	7.042	36052	7.115

2.2.4 Algorithm 1b

An improvement over Golub-Kahan Algorithm is Lawson-Hanson-Chan algorithm which computes QR decomposition of matrix A before Bi-diagonalization. The Bi-diagonalization will be applied to the upper triangular matrix R which helps in reduction of number of householder reflections needed as we apply them only to rows. This method proved to be efficient for $m > \frac{5}{3}n$. An outline of algorithm 1b is presented below.

- Step-1: Perform QR decomposition of matrix A

$$A = QR \quad (2.6)$$

- Step-2: Reduce upper triangular matrix R into Bi-diagonal form using orthogonal transformations

$$R = PBQ^T \quad (2.7)$$

- Step-3: Reduce the Bi-diagonal matrix B obtained in Step-2 to diagonal form using Givens rotations

Matlab code for SVD using algorithm 1b can be found in *alg_2b.m*

2.2.5 Algorithms for QR decomposition

QR Decomposition of a matrix is decomposition of a matrix into product of an orthogonal matrix Q and an upper triangular matrix R .

$$A = QR$$

Two methods for QR decomposition namely Gram-Schmidt algorithm and using Householder reflections are discussed below.

Gram-Schmidt Algorithm

Consider the columns in the matrix A as $[a_1 \mid a_2 \mid \cdots \mid a_n]$

Then

$$\begin{aligned}
 u_1 &= a_1, & e_1 &= \frac{u_1}{\|u_1\|} \\
 u_2 &= a_2 - (a_2 \cdot e_1)e_1, & e_2 &= \frac{u_2}{\|u_2\|} \\
 u_{k+1} &= a_{k+1} - (a_{k+1} \cdot e_1) e_1 - \dots - (a_{k+1} \cdot e_k) e_k, & e_{k+1} &= \frac{u_{k+1}}{\|u_{k+1}\|}
 \end{aligned} \quad (2.8)$$

where $(a_{k+1} \cdot e_k)$ represent dot product between the vectors a_{k+1} and e_k ; $\|u_k\|$ represents Euclidian norm of vector u_k . In case if $\|u_k\|$ is zero (happens if entire k th column in matrix A is zero vector), then e_k is set to zero. Therefore matrix A can be written as

$$A = \begin{bmatrix} a_1 & | & a_2 & | & \cdots & | & a_n \end{bmatrix} = \begin{bmatrix} e_1 & | & e_2 & | & \cdots & | & e_n \end{bmatrix} \begin{bmatrix} a_1 \cdot e_1 & a_2 \cdot e_1 & \cdots & a_n \cdot e_1 \\ 0 & a_2 \cdot e_2 & \cdots & a_n \cdot e_2 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_n \cdot e_n \end{bmatrix}$$

Consider the following matrix,

$$A = \begin{bmatrix} 3 & 7 \\ 2 & 0 \\ 1 & 5 \end{bmatrix}$$

Applying Gram-Schmidt process,

$$u_1 = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}, \quad e_1 = \frac{1}{\sqrt{14}} \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 0.8018 \\ 0.5345 \\ 0.2673 \end{bmatrix}$$

$$u_2 = \begin{bmatrix} 7 \\ 0 \\ 5 \end{bmatrix} - (6.9491) \begin{bmatrix} 0.8018 \\ 0.5345 \\ 0.2673 \end{bmatrix} = \begin{bmatrix} 1.4282 \\ -3.7143 \\ 3.1425 \end{bmatrix}, \quad e_2 = \begin{bmatrix} 0.2817 \\ -0.7325 \\ 0.6197 \end{bmatrix}$$

Therefore, the matrix Q and R are given by

$$Q = \begin{bmatrix} 0.8018 & 0.2817 \\ 0.5345 & -0.7325 \\ 0.2673 & 0.6198 \end{bmatrix} \quad R = \begin{bmatrix} 3.7417 & 6.9488 \\ 0 & 5.0709 \\ 0 & 0 \end{bmatrix}$$

The matlab code for finding QR decomposition using Gram-Schmidt process can be found in *gs_qr.m*

Using Householder reflections

Householder reflectors are successively applied on each column in such a way that it retains only k elements in k th column. This method computes full QR factorization. A Matlab code for finding QR decomposition using Householder reflections can be found in *qr_hh.m*

Consider the following matrix

$$A = \begin{bmatrix} 3 & 7 \\ 2 & 0 \\ 1 & 5 \end{bmatrix}$$

Applying Householder reflections to find QR decompositon resulted in

$$Q = \begin{bmatrix} -0.8018 & 0.2817 & -0.5270 \\ -0.5345 & -0.7325 & 0.4216 \\ -0.2673 & 0.6198 & 0.7379 \end{bmatrix} \quad R = \begin{bmatrix} -3.7417 & -6.9488 \\ -0.0000 & 5.0709 \\ 0.0000 & 0.0000 \end{bmatrix}$$

Observe that using householder method we obtain matrix Q with order $m \times m$.

2.2.6 Bi-diagonalization of Triangular matrix

Using Householder reflectors

The triangular matrix R obtained in QR decomposition is bi-diagonalized by applying a series of householder reflectors on each row in such a way that it retains only k th and $k + 1$ th elements in k th row. Hence after applying reflections to $n - 2$ rows we will be landing with a bi-diagonal matrix. The matlab code for bi-diagonalization of a triangular matrix can be found in *bidig_trng.m*

2.2.7 Diagonalization of Bi-diagonal matrix

The bi-diagonal matrix B obtained in Step-2 of Algorithm 1b is diagonalized using Givens rotation as discussed in Algorithm 1a.

2.3 Multiple Relatively Robust Representation (MRRR) algorithm

This algorithm requires a tri-diagonal/bi-diagonal form and is computationally efficient if only a subset ($k < n$) of singular values is required. Given a Tri-diagonal/ bi-diagonal matrix, the algorithm computes singular values and singular values in $O(kn)$ time.

2.4 Divide and conquer algorithm

Given a bi-diagonal matrix T , the divide and conquer algorithm [9] recursively divides into two parts until it is sufficiently small that can be solved easily by other algorithms like Golub-Kahan SVD. Significant saving in flops is observed when eigen vectors are also computed. QR algorithm takes approx. $9m^3$ flops whereas divide and conquer algorithm requires only $4m^3$ flops in total. The divide and conquer algorithm developed by Ming Gu in [10] has been presented here.

Example: Let the given dense matrix be A of dimension $m \times n$ which is further reduced to bi-diagonal(lower bi-diagonal variant shown below, although same can be extended to upper diagonal form which differs in partitioning the matrix with some scalar terms along row).

$$A = \begin{bmatrix} 30 & 39 & 48 & 1 & 19 \\ 38 & 47 & 7 & 9 & 27 \\ 46 & 6 & 8 & 17 & 35 \\ 5 & 14 & 16 & 25 & 36 \\ 13 & 15 & 24 & 33 & 44 \\ 21 & 23 & 32 & 41 & 3 \\ 22 & 31 & 40 & 49 & 11 \end{bmatrix}$$

Converting into bi-diagonal form using methods like Householder reflectors as described in Algorithm 5.2.1.

$$B = \begin{bmatrix} -72.02 & 0 & 0 & 0 & 0 & 0 \\ 117.94 & -93.45 & 0 & 0 & 0 & 0 \\ 0 & -33.90 & 38.96 & 0 & 0 & 0 \\ 0 & 0 & 9.56 & -32.54 & 0 & 0 \\ 0 & 0 & 0 & 25.77 & 39.51 & 0 \\ 0 & 0 & 0 & 0 & 16.4 & 0.85 \\ 0 & 0 & 0 & 0 & 0 & -26.53 \end{bmatrix}$$

Let us assume that any dense matrix of dimension $m \times n$ be reduced to lower bi-diagonal matrix of size $(n + 1) \times n$. Then the obtained lower bi-diagonal matrix is partitioned into two sub matrices B_1 and B_2 of dimensions $k \times (k - 1)$ and $(n - k + 1) \times (n - k)$ respectively. Generally k is chosen as $\lfloor \frac{n}{2} \rfloor$.

$$B = \begin{bmatrix} B_1 & \alpha_k e_k & 0 \\ 0 & \beta_k e_1 & B_2 \end{bmatrix}$$

Hence for the example, the value of k is 3 and matrix B is partitioned into B_1 and B_2 as shown below.

$$B_1 = \begin{bmatrix} -72.02 & 0 \\ 117.94 & -93.45 \\ 0 & -33.90 \end{bmatrix} \quad B_2 = \begin{bmatrix} -32.54 & 0 & 0 \\ 25.77 & 39.51 & 0 \\ 0 & 16.4 & 0.85 \\ 0 & 0 & -26.53 \end{bmatrix}$$

Then the SVD of these sub-matrices B_1 and B_2 are computed using standard algorithms or these may be further recursively sub-divided until they are computationally less intensive compared to initial dimension.

Let the SVD of the sub-matrix B_i be given as,

$$B_i = \begin{bmatrix} Q_i & q_i \end{bmatrix} \begin{bmatrix} D_i \\ 0 \end{bmatrix} W_i^T$$

where $(Q_i \ q_i)$ and W_i are left and right orthogonal matrix respectively and D_i is the diagonal matrix containing singular values. The left orthogonal matrix U_1 is blocked into Q_1 and q_1 as shown below. Similarly the left orthogonal matrix U_2 is blocked into Q_2 and q_2 .

$$Q_1 = \begin{bmatrix} -0.37 & -0.79 \\ 0.92 & -0.25 \\ 0.12 & -0.56 \end{bmatrix}, \quad q_1 = \begin{bmatrix} -0.49 \\ -0.30 \\ 0.82 \end{bmatrix}$$

$$Q_2 = \begin{bmatrix} -0.42 & 0.84 & 0.15 \\ 0.88 & 0.30 & 0.06 \\ 0.23 & 0.41 & 0.04 \\ 0 & 0.17 & -0.99 \end{bmatrix}, \quad q_2 = \begin{bmatrix} 0.29 \\ 0.37 \\ -0.88 \\ 0.03 \end{bmatrix}$$

Let l_1^T and λ_1 be the last row and last component in Q_1 and q_1 . Similarly let f_2^t and φ_2 be the first

row and first component in Q_2 and q_2 respectively. Hence the matrix B can be rewritten as follows.

$$\begin{aligned}
B &= \begin{bmatrix} (Q_1 \ q_1) \begin{pmatrix} D_1 \\ 0 \end{pmatrix} W_1^T & \alpha_k e_k & 0 \\ 0 & \beta_k e_1 & (Q_2 \ q_2) \begin{pmatrix} D_2 \\ 0 \end{pmatrix} W_2^T \end{bmatrix} \\
&= \begin{bmatrix} Q_1 & q_1 & 0 & 0 \\ 0 & 0 & Q_2 & q_2 \end{bmatrix} \begin{bmatrix} D_1 W_1^T & \alpha_k Q_1^T e_k & 0 \\ 0 & \alpha_k q_1^T e_k & 0 \\ 0 & \beta_k Q_2^T e_1 & D_2 W_2^T \\ 0 & \beta_k q_2^T e_1 & 0 \end{bmatrix} \\
&= \begin{bmatrix} q_1 & Q_1 & 0 & 0 \\ 0 & 0 & Q_2 & q_2 \end{bmatrix} \begin{bmatrix} \alpha_k \lambda_1 & 0 & 0 \\ \alpha_k l_1 & D_1 & 0 \\ \beta_k f_2 & 0 & D_2 \\ \beta_k \varphi_2 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & W_1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & W_2 \end{bmatrix}^T
\end{aligned}$$

We apply Givens rotation to annihilate the term $\beta_k \varphi_2$. Define the parameters required for Givens rotation as,

$$r_0 = \sqrt{(\alpha_k \lambda_1)^2 + (\beta_k \varphi_2)^2}, \quad c_0 = \frac{\alpha_k \lambda_1}{r_0}, \quad s_0 = \frac{\beta_k \varphi_2}{r_0}$$

For the example the values of r_0, c_0 and s_0 are found to be 32.0977, 0.9963 and 0.0865 respectively. Pre-multiplying matrix B with the rotation matrix,

$$B = \begin{bmatrix} c_0 q_1 & Q_1 & 0 & -s_0 q_1 \\ s_0 q_2 & 0 & Q_2 & c_0 q_2 \end{bmatrix} \begin{bmatrix} r_0 & 0 & 0 \\ \alpha_k l_1 & D_1 & 0 \\ \beta_k f_2 & 0 & D_2 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & W_1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & W_2 \end{bmatrix}^T$$

The non-zero block in the middle matrix be named M , which will be used for finding the singular values.

$$M = \begin{bmatrix} r_0 & 0 & 0 \\ \alpha_k l_1 & D_1 & 0 \\ \beta_k f_2 & 0 & D_2 \end{bmatrix}$$

The matrix M has been found to be as given below.

$$M = \begin{bmatrix} 32.10 & 0 & 0 & 0 & 0 & 0 \\ 4.50 & 162.59 & 0 & 0 & 0 & 0 \\ -21.80 & 0 & 50.44 & 0 & 0 & 0 \\ -4.05 & 0 & 0 & 52.88 & 0 & 0 \\ 8.07 & 0 & 0 & 0 & 27.54 & 0 \\ 1.45 & 0 & 0 & 0 & 0 & 26.51 \end{bmatrix}$$

This matrix B possess a special structure which has non-zero entries only along the diagonal and in

the first column.

$$M = \begin{bmatrix} z_1 & & & & \\ & z_2 & d_2 & & \\ & \vdots & & \ddots & \\ & & & & d_n \\ z_n & & & & \end{bmatrix}$$

Let d be a vector with diagonal entries of matrix M with d_1 set to zero. Similarly let z be the vector containing the first column entries of matrix M . The singular values ω_i of matrix M satisfy the interlacing property and secular equation given below.

$$0 = d_1 < \omega_1 < d_2 < \dots < d_n < \omega_n$$

$$f(\omega) = 1 + \sum_{k=1}^n \frac{z_k^2}{d_k^2 - \omega^2} = 0 \quad (2.9)$$

The roots are generally found using Newton Raphson method which takes $O(m)$ operations for each roots and hence $O(m^2)$ time complexity for a $m \times m$ matrix. The singular vectors are given by,

$$u_i = \left(\frac{z_1}{d_1^2 - \omega_i^2}, \dots, \frac{z_n}{d_n^2 - \omega_i^2} \right)^T \bigg/ \sqrt{\sum_{k=1}^n \frac{z_k^2}{(d_k^2 - \omega_i^2)^2}}$$

$$v_i = \left(-1, \frac{d_2 z_2}{d_2^2 - \omega_i^2}, \dots, \frac{d_n z_n}{d_n^2 - \omega_i^2} \right)^T \bigg/ \sqrt{1 + \sum_{k=2}^n \frac{(d_k z_k)^2}{(d_k^2 - \omega_i^2)^2}}$$

We found a method of solving the roots of the secular equation by nature-inspired methods like firefly algorithm as described in [11], but since these meta-heuristic methods are time-consuming, they have been left out as they may not be apt for our time-constrained problem.

2.5 Bi-section and Inverse iteration

Bi-section method: Bi-section method makes use of binary search for finding the roots of the characteristic polynomial of the given matrix.

Inverse Iteration: Given an estimate of eigen value μ (obtained from bi-section method), inverse iteration finds the corresponding eigen vector. The algorithm starts with an initial guess b_0 and updates the eigen vector using the recurrence relation in equation 2.10.

$$b_{k+1} = \frac{(A - \mu I)^{-1} b_k}{C_k} \quad (2.10)$$

where $C_k = \left\| (A - \mu I)^{-1} b_k \right\|$.

2.6 Hybrid methods for SVD

2.6.1 Introduction

In this section, we propose a hybrid algorithm which makes use of Householder bi-diagonalization followed by the two sided Jacobi algorithm. The accuracy of the singular values are defined by a

metric, percentage of relative accuracy which is defined below whereas the accuracy of the singular vector is determined by the accuracy of the similarity factor.

$$\% \text{ rel. error} = \frac{\sigma_i - \hat{\sigma}_i}{\sigma_i} \times 100 \quad (2.11)$$

Throughout this section we make use of the historical dataset with snapshot windows (100% fault samples) from fault operations 1,2 and 3. We plot the difference between the computed similarity factor from various hybrid algorithms and the similarity factor obtained from using Matlab svd command as they differ mostly after 2 or 3 decimal places.

2.6.2 Householder Bi-diagonalization followed by Jacobi

The initial covariance (dense) matrix is first reduced to bidiagonal form using householder reflections. Then the obtained bidiagonal matrix is further reduced to diagonal matrix using Jacobi rotations. One advantage of Jacobi method is faster annihilation of off-diagonal elements. The bi-diagonal form is no longer preserved with application of Jacobi rotations after one sweep as every non-zero upper diagonal element will make the entire 2x2 block as dense matrix. N-D CORDIC can be used in place of 2-D CORDIC for reducing latency in forming the householder reflections with a space-time trade-off. J.M. Delosme et al. in [12], [13] gave a direction for multi-dimensional CORDIC algorithms and in particular 4-D CORDIC. We make use of the householder reflections described in section 2.2.1 for bi-diagonalization instead of CORDIC based methods.

2.6.3 QR decomposition followed by Jacobi

Since we are not preserving the bi-diagonal structure in the above Householder method, we also look at QR decomposition followed by Jacobi method as it is less computationally intensive compared to Householder reflection. We make use of the householder reflections described in section 2.2.1 to upper triangularize the matrix and then diagonalize this matrix using Jacobi rotations.

2.7 Selection of an SVD algorithm for Pattern matching

Fig.2.4 and fig.2.5 shows the error in the computed similarity factor with snapshot as IDV1 and IDV3 respectively. It can be seen that even 7 sweeps of Jacobi performed well in IDV1 as the number of PC's required for capturing 95% variance is just 12 whereas the number of PC's required for IDV3 are as high as 32. Simple Jacobi algorithm is only able to accurately determine the well conditioned singular values and vectors while Householder followed by 6 sweeps of Jacobi performed much better than the remaining algorithms. Since it is difficult to visualize the % relative error of top 32 singular values using various algorithms of different windows, we plot the worst(maximum) absolute % rel. error of each singular value recorded among all the windows in the training dataset as shown in fig. 2.6. A worst absolute % relative error helps in generalizing the error in a particular singular value i.e suppose using the 6 sweeps of Jacobi, the worst %rel. error of 32nd singular value has been found to be 35% which means that 32nd singular value from all the windows will have error of no more than 35%. Only top 32 singular values are considered here as it has been found that the number of PC's required for 95% variance will not exceed 32 in the entire 4.32 million historical

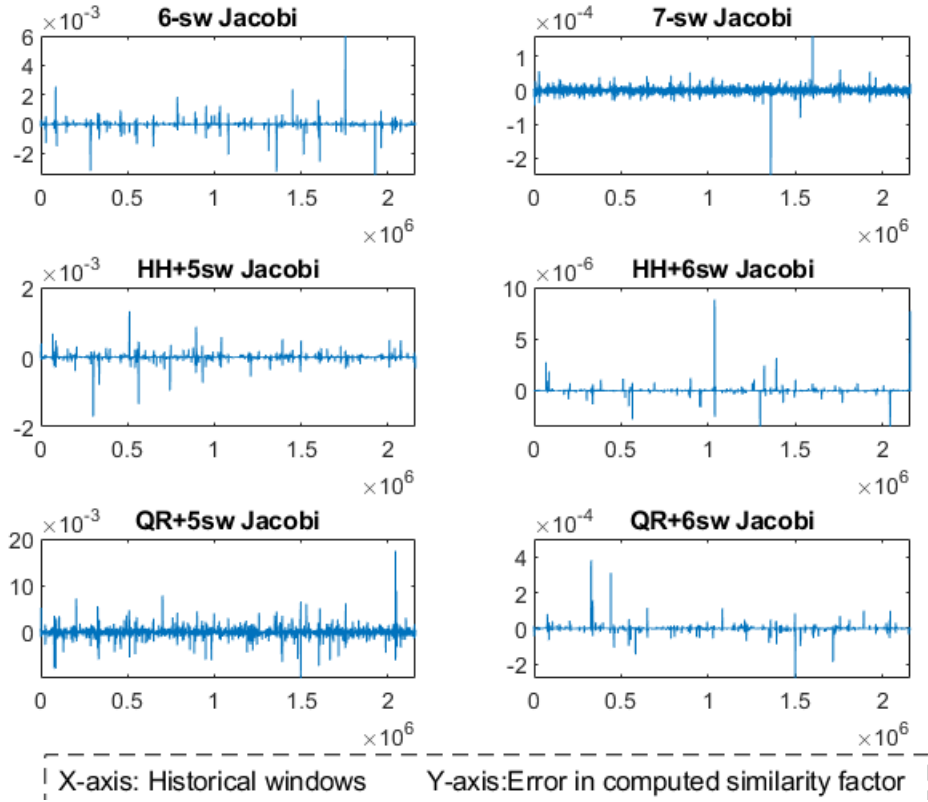


Figure 2.4: Comparison of various algorithms with snapshot as IDV1

database with a window size of 500.

The bi-diagonal methods of SVD are very accurate but suffer from large latency because of large dense matrix multiplications. Even the Matlab implementation of SVD is known to make use of divide and conquer Bi-diagonal SVD. For a general $m \times n$ matrix, it requires $2n$ matrix multiplications just for computing singular values alone. A set of n matrix multiplications for pre-multiplications each matrix of size $m \times m$ and another set of n multiplications each of size $n \times n$. If this bi-diagonalization has been marked for Hardware implementation, this demands a huge number of DSP48E multiplier blocks instead a n-D CORDIC based Bi-diagonalization has to be done to limit the number of multipliers by trading off a little accuracy. From inspection of the similarity factors computed through Matlab, it has been observed that an accuracy of upto 3 decimal places is necessary for better pattern matching to keep the mis-classifications under control. This problem can also be avoided by proposing a new similarity factor which shows better diversity among all the operating conditions, in such a case the accuracy of similarity factor can be slightly traded-off. Householder followed by 6 sweeps of Jacobi gave a better accuracy of upto 4 decimal places and hence the algorithm can be chosen when we require such close precision and accuracy i.e when the similarity factor reaches a threshold of above 0.7-0.8. In remaining cases where the similarity factor is below the threshold, the Jacobi algorithm can be chosen as base algorithm.

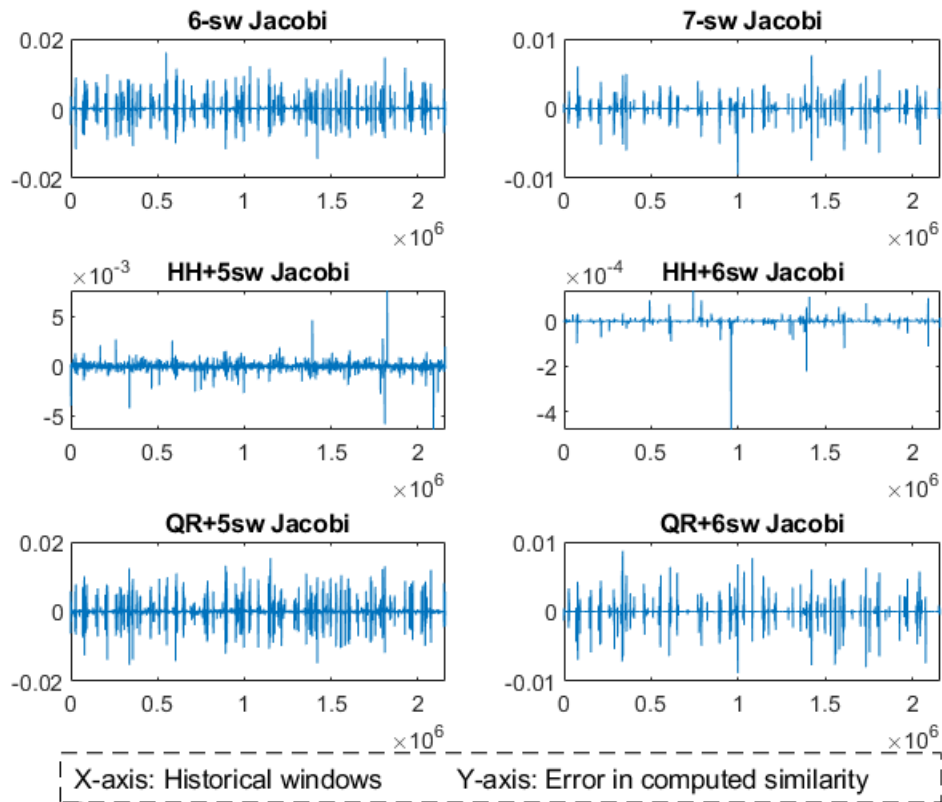


Figure 2.5: Comparison of various algorithms with snapshot as IDV3

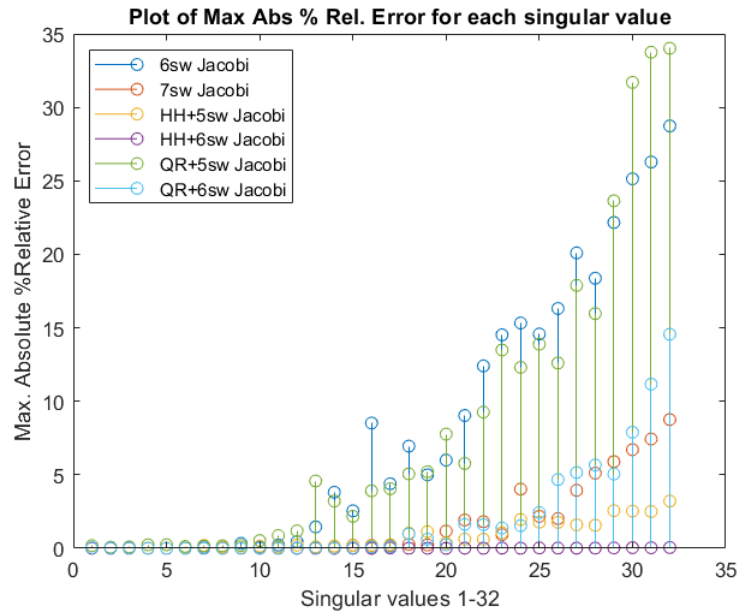


Figure 2.6: Worst absolute % rel. error of top 32 singular values

Chapter 3

CORDIC

3.1 Introduction

CORDIC(acronym for Co-Ordinate Rotational Digital Computer) is widely used for computing trigonometric functions, exponential and logarithmic functions and square root operations. As it makes use of simple shift and add/subtract operations, making the design simpler, reliable and faster without the need of multipliers and look-up table to compute trigonometric functions. It was first developed by Jack E. Volder in 1959 and thereafter many architectures have been proposed for high performance and low cost design which are briefly described in [14].

3.2 Modes of CORDIC

We restrict the discussion to circular co-ordinate system, although there are other co-ordinates in which CORDIC operates like linear and hyperbolic. CORDIC operates in two modes namely Rotation mode and Vectoring mode as shown in fig 3.1.



Figure 3.1: Modes of CORDIC

Table 3.1: Microrotations

i	0	1	2	3	4	5	6	7	8	9
θ	45°	26.565°	14.036°	7.125°	3.356°	1.789°	0.895°	0.448°	0.224°	0.112°

3.2.1 Rotation mode

Rotation mode basically rotates the initial vector (x_0, y_0) by a given angle θ and computes the final vector (x_f, y_f) . The basic rotation matrix for clockwise sense is as shown below.

$$\begin{pmatrix} x_f \\ y_f \end{pmatrix} = \begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \quad (3.1)$$

For anti-clockwise rotation, the rotation matrix can be obtained by replacing θ with $-\theta$, which just transposes the rotation matrix obtained for clock-wise sense as shown below.

$$\begin{pmatrix} x_f \\ y_f \end{pmatrix} = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \quad (3.2)$$

The rotation matrix can be decomposed to simple shift and add/subtract operations as shown below. Let R_{θ_i} be the rotation matrix at i^{th} iteration,

$$R_{\theta_i} = \begin{pmatrix} \cos \theta_i & -\sin \theta_i \\ \sin \theta_i & \cos \theta_i \end{pmatrix}$$

$$R_{\theta_i} = \cos \theta_i \begin{pmatrix} 1 & -\tan \theta_i \\ \tan \theta_i & 1 \end{pmatrix}$$

The $\tan \theta_i$ present in the above equation can be replaced with known micro-rotations with $\theta_i = \arctan \frac{1}{2^i}$ as listed in Table 3.1.

$$R_{\theta_i} = \cos \theta_i \begin{pmatrix} 1 & -\frac{1}{2^i} \\ \frac{1}{2^i} & 1 \end{pmatrix}$$

Making use of trigonometric relationship between cosine and tangent,

$$\cos \theta = \frac{1}{\sqrt{1 + \tan^2 \theta}}$$

$$R_{\theta_i} = \frac{1}{\sqrt{1 + (2^{-i})^2}} \begin{pmatrix} 1 & -\frac{1}{2^i} \\ \frac{1}{2^i} & 1 \end{pmatrix} \quad (3.3)$$

The terms $\frac{1}{2^i}$ can be easily implemented in hardware as it corresponds to Shift-right operation(\gg). The scaling factor in equation 3.3 need not be computed at each iteration as the product finally converges to 0.6037 after a finite number of iterations and hence a final scaling can be done at the end.

$$\prod_{i=0}^{\infty} \frac{1}{\sqrt{1 + (2^{-i})^2}} \simeq 0.6037 \quad (3.4)$$

Using equations 3.1 and 3.3 rotation outputs for clockwise rotation can be expressed at i^{th} instant are given by

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \begin{pmatrix} x_i + (y_i \gg i) \\ -(x_i \gg i) + y_i \end{pmatrix} \quad (3.5)$$

Table 3.2: Quadrant detector

Quadrant	MSB(x)	MSB(y)	$quad$
1 st	0	0	00
2 nd	1	0	10
3 rd	1	1	11
4 th	0	1	01

Table 3.3: Transforming inputs

Quadrant	$quad$	x_0	y_0
1 st	00	x_i	y_i
2 nd	10	$-x_i$	$-y_i$
3 rd	11	$-x_i$	$-y_i$
4 th	01	x_i	y_i

Similarly for anti-clockwise rotation,

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \begin{pmatrix} x_i - (y_i \gg i) \\ (x_i \gg i) + y_i \end{pmatrix} \quad (3.6)$$

Consider the following example. Let the vector (1,2) be rotated by an angle of 65.43°.

- Iteration-1: $i = 0 \Rightarrow$ input angle is positive hence 45° clockwise. From equation 3.5, (1,2) after 45° CW rotation gives (3,1).
Is 45° > 65.43° ? No \Rightarrow next rotation clockwise.
- Iteration-2: $i = 1 \Rightarrow$ 26.565° clockwise. From equation 3.5, (3,1) after 26.565° CW rotation gives (3.5,-0.5).
Is 45° + 26.565° = 71.565° > 65.43° ? Yes \Rightarrow next rotation anticlockwise.
- Iteration-3: $i = 2 \Rightarrow$ 14.036° anticlockwise. From equation 3.6, (3.5,-0.5) after 14.036° ACW rotation gives (3.625,0.375).
Is 45° + 26.565° - 14.036° = 57.529° > 65.43° ? No \Rightarrow next rotation clockwise.

After 11 pseudo-rotations and scaling the output vector we obtain (2.234,-0.075).

To overcome the problem of limited converge of $\pm 90^\circ$ in CORDIC, an initial 90° rotation will be performed, thus achieving a $\pm 180^\circ$ range.

3.2.2 Vectoring mode

Given a two-dimensional vector, the vectoring mode computes the angle made by the vector with respect to positive x-axis and magnitude of the vector. Thus equivalent to a cartesian to polar conversion. A quadrant detector based on the sign of x and y input gets the information of the quadrant of the vector. The input vector if lying in 2nd or 3rd maps to 4th or 1st quadrant respectively. After mapping the vector to 1st or 4th quadrant, the vector is rotated until the y -component is zero. If the y co-ordinate is positive, a clockwise rotation is performed else an anticlockwise rotation is performed. Then a final correction is made to the computed angle based on the quadrant information.

Consider the following example. Let $X_i = (-3, 4)$ be the input to the CORDIC vectoring module.

From quadrant detector shown in Table 3.2, the quadrant information will be stored precisely in a two-bit register $quad$ which is used to transform the input vector. Since $quad = 10$ for the given vector, the transformed vector is $X_t=(3,-4)$ using Table 3.3.

- Iteration-1($i=0$): Since $\text{sign}(y_0)$ is negative, an anti-clockwise rotation of 45° is made. $X_0=(3,-4)$ rotated by 45° ACW $\equiv (x_0 - [y_0 \gg 0], [x_0 \gg 0] + y_0)$ (Using equation 3.6) $= (7,-1)$
 $\theta_0 = -45^\circ$.
- Iteration-2($i=1$): Since $\text{sign}(y_1)$ is negative, an anti-clockwise rotation of 26.565° is made. $X_1=(7,-1)$ rotated by 26.565° ACW $\equiv (x_1 - [y_1 \gg 1], [x_1 \gg 1] + y_1)$ (Using equation 3.6) $= (7.5, 2.5)$ $\theta_1 = \theta_0 - 26.565^\circ = -71.565^\circ$.
- Iteration-3($i=2$): Since $\text{sign}(y_2)$ is positive, a clockwise rotation of 14.036° is made. $X_2=(7.5, 2.5)$ rotated by 14.036° ACW $\equiv (x_2 + [y_2 \gg 2], -[x_2 \gg 2] + y_2)$ (Using equation 3.5) $= (8.125, 0.625)$
 $\theta_2 = \theta_1 + 14.036^\circ = -57.529^\circ$.

After a series of 11 such rotations i.e when $i=10$, the final vector X_{10} is given by $(8.2338, 0.0016)$ and $\theta_{10}=-53.1413^\circ$. Scaling the final vector X_{10} with a factor 0.6037 as given in equation 3.4, we obtain $X_f=(5.0002, 0.0009)$ and since the angle obtained is for the transformed vector X_t the actual angle is computes as $\theta_f=180^\circ + \theta_{10}=126.858^\circ$. The pseudo-rotations made are as shown in fig 3.2.

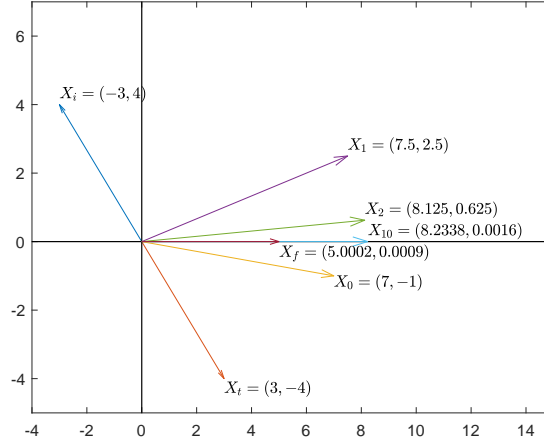


Figure 3.2: Illustration of pseudo-rotations

3.3 Pipelined CORDIC

In many applications (applies to SVD problem), the angle input for rotation module is obtained from a vectoring module. Since in vectoring mode, the required sequences of micro rotations are already computed, the same information is simultaneously passed to Rotation module. This helps in a saving of clock cycles. Thus the final vector after rotation will be obtained by a delay of just one clock cycle in rotation mode compared to it's counterpart vectoring mode.

Chapter 4

Jacobi Methods of SVD

4.1 Introduction

Jacobi methods are quite popular in hardware implementation of SVD because the computation can be grouped in parallel. Various kinds of systolic array implementations based on Jacobi methods are reported in literature for hardware acceleration of SVD. Based on the number of angles required for annihilating the off-diagonal elements, Jacobi methods are broadly classified as One-sided and Two-sided Jacobi methods.

4.2 Classical Jacobi Algorithm

Given a symmetric matrix $A \in \mathbb{R}^{n \times n}$, Jacobi method chooses an off-diagonal ($p \neq q$) index pair (p, q) and performs the rotation that diagonalizes the 2×2 subproblem. The matrix A is overwritten at each step $A \leftarrow J^T(p, q, \theta)AJ(p, q, \theta)$. Only the rows p, q and columns p, q are affected during rotation. The classical jacobi algorithm chooses an index pair (p, q) such that a_{pq}^2 is maximum of all other pairs. Let $N = \frac{n(n-1)}{2}$. A sequence of N jacobi rotations is called *sweep*. The algorithm is quite slow as it takes $O(n^2)$ operations to find the optimal (p, q) . Other implementations like parallel ordering and row ordering schedule the sequence of sub-problems to be solved in advance instead of searching for optimal index (p, q) .

4.3 Two-sided Jacobi Method

For Two-sided Jacobi method, the matrix must be square. For rectangular matrices,

- zeros are padded to make it a square matrix.
- QR decomposition can be performed to make it square but the overall assembling to final svd is quite difficult.
- Householder reflections which bidiagonalize the matrix can also be employed to solve rectangular svd problems, but generation of householder matrices is difficult and bi-diagonal structure is not exploited with regular systolic array.

Consider a non-symmetric matrix A as shown below.

$$A = \begin{bmatrix} w & x \\ y & z \end{bmatrix}$$

The jacobi rotations when applied on both sides diagonalizes the matrix A as shown below.

$$\begin{bmatrix} \cos \theta_1 & \sin \theta_1 \\ -\sin \theta_1 & \cos \theta_1 \end{bmatrix}^T \begin{bmatrix} w & x \\ y & z \end{bmatrix} \begin{bmatrix} \cos \theta_2 & \sin \theta_2 \\ -\sin \theta_2 & \cos \theta_2 \end{bmatrix} = \begin{bmatrix} d_1 & 0 \\ 0 & d_2 \end{bmatrix} \quad (4.1)$$

Solving for off-diagonal elements and equating to zero,

$$w \cdot \sin \theta_1 \cos \theta_2 + y \cdot \cos \theta_1 \cos \theta_2 - x \cdot \sin \theta_1 \sin \theta_2 + z \cdot \cos \theta_1 \sin \theta_2 = 0 \quad (4.2)$$

$$w \cdot \cos \theta_1 \sin \theta_2 - y \cdot \sin \theta_1 \sin \theta_2 + x \cdot \cos \theta_1 \cos \theta_2 - z \cdot \sin \theta_1 \cos \theta_2 = 0 \quad (4.3)$$

Adding and subtracting equations 4.2 and 4.3, gives the required jacobi rotation angles.

$$\tan(\theta_1 + \theta_2) = \frac{x + y}{z - w} \quad ; \quad \tan(\theta_1 - \theta_2) = \frac{x - y}{z + w} \quad (4.4)$$

Let $\theta_{12} = \theta_1 + \theta_2$ and $\theta_{21} = \theta_1 - \theta_2$. The angles θ_{12} and θ_{21} can be obtained from CORDIC vectoring mode which has been discussed in section 2.2.2 by feeding y input as $(x + y)$ or $(x - y)$ and x input as $(z - w)$ or $(z + w)$ respectively. The matrix A can be diagonalized in terms of θ_{12} and θ_{21} as shown below. Equation 4.1 can be re-written as follows.

$$\begin{aligned} \begin{bmatrix} d_1 & 0 \\ 0 & d_2 \end{bmatrix} &= \begin{bmatrix} \cos \theta_1 & -\sin \theta_1 \\ \sin \theta_1 & \cos \theta_1 \end{bmatrix} \begin{bmatrix} w \cdot \cos \theta_2 - x \cdot \sin \theta_2 & w \cdot \sin \theta_2 + x \cdot \cos \theta_2 \\ y \cdot \cos \theta_2 - z \cdot \sin \theta_2 & y \cdot \sin \theta_2 + z \cdot \cos \theta_2 \end{bmatrix} \\ &= \begin{bmatrix} \cos \theta_1 & -\sin \theta_1 \\ \sin \theta_1 & \cos \theta_1 \end{bmatrix} \left\{ \begin{bmatrix} w \cdot \cos \theta_2 & x \cdot \cos \theta_2 \\ y \cdot \cos \theta_2 & z \cdot \cos \theta_2 \end{bmatrix} + \begin{bmatrix} -x \cdot \sin \theta_2 & w \cdot \sin \theta_2 \\ -z \cdot \sin \theta_2 & y \cdot \sin \theta_2 \end{bmatrix} \right\} \end{aligned} \quad (4.5)$$

R.H.S of equation 4.5 can be further re-arranged as,

$$\begin{aligned} &= \begin{bmatrix} \cos \theta_1 & -\sin \theta_1 \\ \sin \theta_1 & \cos \theta_1 \end{bmatrix} \begin{bmatrix} w \cdot \cos \theta_2 & x \cdot \cos \theta_2 \\ y \cdot \cos \theta_2 & z \cdot \cos \theta_2 \end{bmatrix} + \begin{bmatrix} \cos \theta_1 & -\sin \theta_1 \\ \sin \theta_1 & \cos \theta_1 \end{bmatrix} \begin{bmatrix} -x \cdot \sin \theta_2 & w \cdot \sin \theta_2 \\ -z \cdot \sin \theta_2 & y \cdot \sin \theta_2 \end{bmatrix} \\ &= \begin{bmatrix} \cos \theta_1 \cos \theta_2 & -\sin \theta_1 \cos \theta_2 \\ \sin \theta_1 \cos \theta_2 & \cos \theta_1 \cos \theta_2 \end{bmatrix} \begin{bmatrix} w & x \\ y & z \end{bmatrix} + \begin{bmatrix} \cos \theta_1 \sin \theta_2 & -\sin \theta_1 \sin \theta_2 \\ \sin \theta_1 \sin \theta_2 & \cos \theta_1 \sin \theta_2 \end{bmatrix} \begin{bmatrix} -x & w \\ -z & y \end{bmatrix} \end{aligned} \quad (4.6)$$

Using trigonometric identities,

$$\begin{aligned} &= \frac{1}{2} \begin{bmatrix} \cos(\theta_1 + \theta_2) + \cos(\theta_1 - \theta_2) & -\{\sin(\theta_1 + \theta_2) + \sin(\theta_1 - \theta_2)\} \\ \sin(\theta_1 + \theta_2) + \sin(\theta_1 - \theta_2) & \cos(\theta_1 + \theta_2) + \cos(\theta_1 - \theta_2) \end{bmatrix} \begin{bmatrix} w & x \\ y & z \end{bmatrix} \\ &+ \frac{1}{2} \begin{bmatrix} \sin(\theta_1 + \theta_2) - \sin(\theta_1 - \theta_2) & \cos(\theta_1 + \theta_2) - \cos(\theta_1 - \theta_2) \\ -\{\cos(\theta_1 + \theta_2) - \cos(\theta_1 - \theta_2)\} & \sin(\theta_1 + \theta_2) - \sin(\theta_1 - \theta_2) \end{bmatrix} \begin{bmatrix} -x & w \\ -z & y \end{bmatrix} \end{aligned} \quad (4.7)$$

Let $rot_y^{l_1}(a, b, \theta_1)$ represents rotating a vector (a, b) by an angle θ_1 and taking the y -component of resultant vector as output. Here l_1 represents a label which groups all the rotation modules with

same angle input. Equation 4.7 can be finally expressed in terms of CORDIC rotation discussed in section 2.2.1 as follows,

$$= \frac{1}{2} \begin{bmatrix} rot_y^{l_1}(y, w, \theta_{12}) + rot_y^{l_2}(y, w, \theta_{21}) & rot_y^{l_1}(z, x, \theta_{12}) + rot_y^{l_2}(z, x, \theta_{21}) \\ rot_x^{l_1}(y, w, \theta_{12}) + rot_x^{l_2}(y, w, \theta_{21}) & rot_x^{l_1}(z, x, \theta_{12}) + rot_x^{l_2}(z, x, \theta_{21}) \end{bmatrix} \\ + \frac{1}{2} \begin{bmatrix} -rot_x^{l_1}(z, x, \theta_{12}) + rot_x^{l_2}(z, x, \theta_{21}) & rot_x^{l_1}(y, w, \theta_{12}) - rot_x^{l_2}(y, w, \theta_{21}) \\ rot_y^{l_1}(z, x, \theta_{12}) - rot_y^{l_2}(z, x, \theta_{21}) & -rot_y^{l_1}(y, w, \theta_{12}) + rot_y^{l_2}(y, w, \theta_{21}) \end{bmatrix} \quad (4.8)$$

Here only a 2×2 sub-matrix is diagonalized but in general the input matrix of any order is subdivided into 2×2 matrices as required for systolic array implementation and then the sub-matrices along the diagonal are diagonalized and then transmitting these rotations along the corresponding row and column.

With two-sided jacobi method we obtain a un-normalized SVD i.e the singular values are not arranged in descending order. So the singular values have to be sorted and the corresponding columns in singular vectors also need to be arranged accordingly.

4.3.1 Shuffling rotations

In the previous section, the elements of input matrix are shuffled according to the ordering (row/parallel) scheme in each iteration of a sweep. Also, the pivots of the rotation matrix are always on the successive pairs. Instead the rotation matrix can be pivoted according to the ordering scheme by keeping the input matrix unaltered.

$$\begin{bmatrix} c_1 & -s_1 & 0 & 0 \\ s_1 & c_1 & 0 & 0 \\ 0 & 0 & c_3 & -s_3 \\ 0 & 0 & s_3 & c_3 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} c_2 & s_2 & 0 & 0 \\ -s_2 & c_2 & 0 & 0 \\ 0 & 0 & c_4 & s_4 \\ 0 & 0 & -s_4 & c_4 \end{bmatrix} = \begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ a'_{21} & a'_{22} & a'_{23} & a'_{24} \\ a'_{31} & a'_{32} & a'_{33} & a'_{34} \\ a'_{41} & a'_{42} & a'_{43} & a'_{44} \end{bmatrix}$$

With pivoting the elements of rotation matrix,

$$\begin{bmatrix} c_1 & 0 & 0 & -s_1 \\ 0 & c_2 & -s_2 & 0 \\ 0 & s_2 & c_2 & 0 \\ s_1 & 0 & 0 & c_1 \end{bmatrix} \begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ a'_{21} & a'_{22} & a'_{23} & a'_{24} \\ a'_{31} & a'_{32} & a'_{33} & a'_{34} \\ a'_{41} & a'_{42} & a'_{43} & a'_{44} \end{bmatrix} \begin{bmatrix} c_4 & 0 & 0 & s_4 \\ 0 & c_3 & s_3 & 0 \\ 0 & -s_3 & c_3 & 0 \\ -s_4 & 0 & 0 & c_4 \end{bmatrix}$$

With input matrix elements being shuffled,

$$\begin{bmatrix} c_1 & -s_1 & 0 & 0 \\ s_1 & c_1 & 0 & 0 \\ 0 & 0 & c_2 & -s_2 \\ 0 & 0 & s_2 & c_2 \end{bmatrix} \begin{bmatrix} a'_{11} & a'_{14} & a'_{12} & a'_{13} \\ a'_{41} & a'_{44} & a'_{42} & a'_{43} \\ a'_{21} & a'_{24} & a'_{22} & a'_{23} \\ a'_{31} & a'_{34} & a'_{32} & a'_{33} \end{bmatrix} \begin{bmatrix} c_4 & s_4 & 0 & 0 \\ -s_4 & c_4 & 0 & 0 \\ 0 & 0 & c_3 & s_3 \\ 0 & 0 & -s_3 & c_3 \end{bmatrix}$$

4.4 One-sided Jacobi Method

Apart from its counterpart Two-sided Jacobi method, the One-sided Jacobi method neither requires the matrix to be square nor require special matrix properties such as symmetry. Brent and Luk developed a linear systolic array in [15] that computes in $O(mn)$ time with $O(n)$ processors and in $O(mn \log n)$ time with $O(mn)$ processors.

Hestenes Jacobi or One-sided Jacobi method finds a matrix V such that the matrix product AV has orthogonal columns. Hestenes made use of plane-rotations to generate matrix V . Let $A = [a_1^{(k)}, \dots, a_n^{(k)}]$ and $Q_k = [q_{cs}^{(k)}]$. The matrix A is updated at every iteration using the following relation.

$$A_{k+1} = A_k \cdot Q_k \quad (4.9)$$

where Q_k represents a rotation in (i, j) plane with $i < j$ i.e

$$\begin{aligned} q_{ii}^{(k)} &= \cos \theta & q_{ij}^{(k)} &= \sin \theta \\ q_{ji}^{(k)} &= -\sin \theta & q_{jj}^{(k)} &= \cos \theta \end{aligned} \quad (4.10)$$

We know that post-multiplication with Q_k will affect only i and j columns.

$$[a_i^{(k+1)}, a_j^{(k+1)}] = [a_i^{(k)}, a_j^{(k)}] \begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix}$$

The rotation angle θ is chosen such that the new columns are orthogonal.

Using the formulas of Rutishauser, we define the parameters:

$$\alpha \equiv \|a_i^{(k)}\|_2^2 \quad \beta \equiv \|a_j^{(k)}\|_2^2 \quad \gamma \equiv a_i^{(k)T} a_j^{(k)}$$

If $\gamma=0$, we set θ to zero otherwise we compute,

$$\xi = \frac{\beta - \alpha}{2\gamma}, \quad t = \frac{\text{sign}(\xi)}{|\xi| + \sqrt{1 + \xi^2}}$$

The rotation parameters are then computed as follows:

$$\cos \theta = \frac{1}{\sqrt{1 + t^2}}, \quad \sin \theta = t \cdot \cos \theta$$

The rotation angle always satisfies the condition stated below.

$$|\theta| \leq \frac{\pi}{4} \quad (4.11)$$

The cyclic-by-row ordering is chosen which processes all (i, j) pairs at least once in every sweep. Forsythe and Henrici in [16] proved that convergence is guaranteed if condition stated in equation 4.11 holds with cyclic-by-row ordering.

4.5 Systolic array implementation

Given a $A \in \mathbb{R}^{n \times n}$ matrix, the BLV array uses $n/2$ by $n/2$ processing elements. Fig. 4.1 shows a typical BLV array for matrix of dimension $n=8$. Each processors holds a 2×2 sub-matrix of A with initial elements as follows.

$$\begin{bmatrix} a_{2i-1,2j-1} & a_{2i-1,2j} \\ a_{2i,2j-1} & a_{2i,2j} \end{bmatrix} \quad (4.12)$$

The diagonal PE's compute the rotation parameters required for annihilating the off-diagonal elements while the off-diagonal PE's perform the transformations to complete the rotation. After the diagonal processor finishes the computation of rotation parameters, they are transmitted to the processing elements present in the row and column. The off-diagonal processing elements perform the rotations on the sub-matrices which they hold and after the rotations are made, the matrix elements are interchanged as per parallel ordering.

For computing singular vectors, each PE is equipped with four more memory cells and the changes being made on any sub-matrix are simultaneously made on these memory cells. When the off-diagonal elements being processed by a diagonal processor are quite small, the diagonal PE can avoid computing the rotation parameters (cos, sin) as it generates (1,0) and transmitting the same along the PE column and row. But this requires an additional logic which checks the magnitude of off-diagonal elements at the time of computation of rotation parameters.

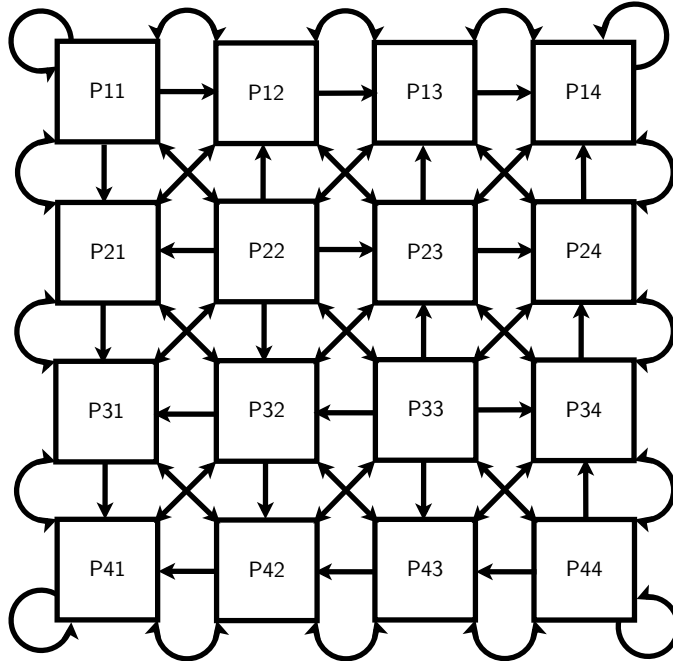


Figure 4.1: BLV array for $n=8$

4.6 Row ordering

Let (p, q) be an order pair which signifies the off-diagonal elements (pq, qp) being processed. The annihilation sequence for $n = 6$ is as shown below.

$$(p, q) = (1,2), (1,3), (1,4), (1,5), (1,6), (2,3), (2,4), (2,5), (2,6), (3,4), (3,5), (3,6), (4,5), (4,6), (5,6).$$

There are $N(= \frac{n(n-1)}{2})$ (p, q) pairs which satisfy the condition $p < q$. From the above sequence it can be understood that the all the (p, q) which satisfy the condition $p < q$ are chosen row-wise and hence the name Row-ordering. Similarly there is an another variant of sequencing the (p, q) pairs in column wise. The annihilation process in one sweep for $n=4$ with row ordering is shown below.

$$\begin{aligned} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} &\rightarrow \begin{bmatrix} a'_{11} & 0 & a'_{13} & a'_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ a'_{31} & a'_{31} & a'_{33} & a'_{34} \\ a'_{41} & a'_{42} & a'_{43} & a'_{44} \end{bmatrix} \rightarrow \begin{bmatrix} a''_{11} & \varepsilon & 0 & a''_{14} \\ \varepsilon & a''_{22} & a''_{23} & a''_{24} \\ 0 & a'_{31} & a'_{33} & a'_{34} \\ a''_{41} & a'_{42} & a'_{43} & a'_{44} \end{bmatrix} \rightarrow \begin{bmatrix} a'''_{11} & \varepsilon & \varepsilon & 0 \\ \varepsilon & a''_{22} & a''_{23} & a''_{24} \\ \varepsilon & a'_{31} & a'_{33} & a'_{34} \\ 0 & a''_{42} & a''_{43} & a''_{44} \end{bmatrix} \\ &\downarrow \\ \begin{bmatrix} a^{iv}_{11} & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & a'''_{22} & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & a'''_{33} & 0 \\ \varepsilon & \varepsilon & 0 & a'''_{44} \end{bmatrix} &\leftarrow \begin{bmatrix} a^{iv}_{11} & \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & a'''_{22} & \varepsilon & 0 \\ \varepsilon & \varepsilon & a''_{33} & a^{iv}_{34} \\ \varepsilon & 0 & a^{iv}_{43} & a^{iv}_{44} \end{bmatrix} \leftarrow \begin{bmatrix} a^{iv}_{11} & \varepsilon & \varepsilon & 0 \\ \varepsilon & a''_{22} & 0 & a'''_{24} \\ \varepsilon & 0 & a''_{33} & a'_{34} \\ 0 & a'''_{42} & a'''_{43} & a'_{44} \end{bmatrix} \end{aligned}$$

After 5-7 sweeps, the off-diagonal elements shown with ε become sufficiently small and the matrix looks diagonal.

4.7 Parallel ordering

For $n = 8$, the parallel ordering scheme for a single sweep is as shown below.

$$\begin{aligned} (p, q) = & (1,2), (3,4), (5,6), (7,8) \\ & (1,4), (2,6), (3,8), (5,7) \\ & (1,6), (4,8), (2,7), (3,5) \\ & (1,8), (6,7), (4,5), (2,3) \\ & (1,7), (8,5), (6,3), (4,2) \\ & (1,5), (7,3), (8,2), (6,4) \\ & (1,3), (5,2), (7,4), (8,6) \end{aligned}$$

Let ord be an array of index pairs at previous iteration of a sweep. Initially the ord array is initialized with numbers 1 to n . Then the ordering scheme for current iteration can be generated using the following flow chart shown in 4.2. The rotation parameters for index pairs present in each row can be calculated concurrently. The annihilation process in one iteration for $n=4$ with parallel ordering

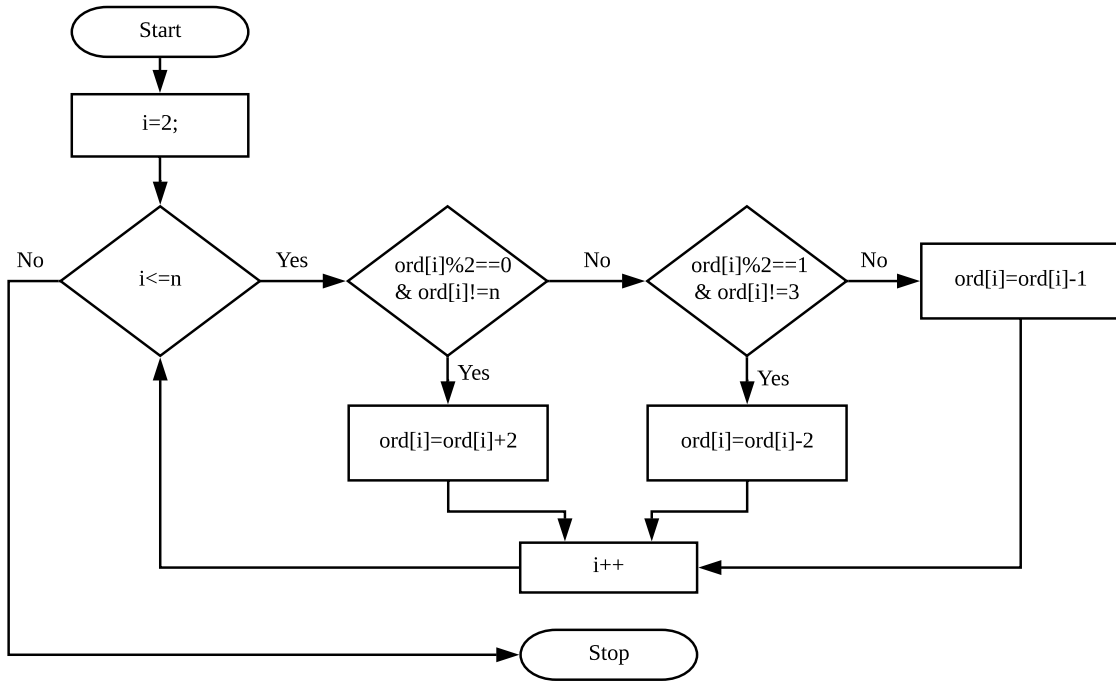


Figure 4.2: Flow chart for Parallel ordering

is shown below.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \rightarrow \begin{bmatrix} a'_{11} & 0 & a'_{13} & a'_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ a'_{31} & a'_{32} & a'_{33} & 0 \\ a'_{41} & a'_{42} & 0 & a'_{44} \end{bmatrix} \rightarrow \begin{bmatrix} a''_{11} & \varepsilon & 0 & a''_{14} \\ \varepsilon & a''_{22} & a''_{23} & 0 \\ 0 & a''_{32} & a''_{33} & \varepsilon \\ a''_{41} & 0 & \varepsilon & a''_{44} \end{bmatrix} \\
 \downarrow \\
 \begin{bmatrix} a'''_{11} & \varepsilon & \varepsilon & 0 \\ \varepsilon & a'''_{22} & 0 & \varepsilon \\ \varepsilon & 0 & a'''_{33} & \varepsilon \\ 0 & \varepsilon & \varepsilon & a'''_{44} \end{bmatrix}$$

After a series of such sweeps(generally 5-7), the off-diagonal elements shown with ε become sufficiently small making the matrix diagonal.

Chapter 5

Hardware implementation of Pattern Matching

5.1 Introduction

In this chapter the hardware implementation of pattern matching for fault diagnosis is discussed. Utmost care has to be taken while designing the basic building blocks(functions) as every clock cycle saved will result in large saving of clock cycles as the same block is instantiated all the time for millions of time but with different inputs.

5.2 Previous work in Hardware implementations of SVD

S. Majumder provided a comprehensive review of various hardware implementations for SVD in [17] which has been presented here. Initially Linear systolic arrays were employed with a time complexity of $O(mn)$ with $O(n)$ processors. Brent and Luk proposed a quadratic systolic array that computes SVD of a $n \times n$ matrix in $O(n \log n)$ with $O(n^2)$ processors. The hardware complexity was further reduced to $O(n^2/2)$ processors by Ahmedsaid and Bouridane [18]. For rectangular matrices they initially performed a QR decomposition which requires $O(m)$ computations thus a total of $O(m + n \log n)$ time for final SVD. Lahabar et al. proposed a GPU based SVD computation using CUDA programming model. They employed Golub-Reinsch algorithm and achieved a considerable speed-up of up to 60 for matrices with large dimensions. The proposed method has been proven to be efficient and faster only for matrices with leading dimension 8k or above. Luis M. Ledesma-Carrillo et al. proposed a reconfigurable FPGA based design which employed Hestenes Jacobi method. The design suffers from very high latency $O(\min(m, n)^5)$ and also the matrix dimension has been restricted to 32x127 because of the available memory in the employed FPGA devices.

5.3 Reading of historical window

We make use of the property of SVD mentioned in section 2.1.1 i.e. SVD is unaltered by shuffling rows. This gives us an advantage of overwriting the oldest sample with a new sample instead of

deleting the oldest sample and then appending the new sample at the end or reading the whole dataset again and again for each window.

5.4 Pre-processing

5.4.1 Recursive Mean and Standard deviation computation

Since we employ a moving-window approach, at each instant only the oldest sample gets replaced with the a new one. Hence, we can make use of previous computations like mean and standard deviation to update them when a sample member is replaced. Let v be the vector of dimension m with mean and standard deviation μ and σ' respectively. If the first(oldest) member of this vector v_1 is replaced with a new member v_{m+1} , then the updated mean μ' and standard deviation σ' are given by the following equations.

$$\mu' = \mu + \frac{v_{m+1} - v_1}{m} \quad (5.1)$$

$$\sigma' = \sqrt{\sigma^2 + \frac{v_{m+1}^2 - v_1^2 + m(\mu^2 - \mu'^2)}{m - 1}} \quad (5.2)$$

Care must be taken while using Recursive methods as any inaccuracies (rounding due to limited precision) present in these computations, will keep propagating to next computations leading to poor accuracy of the whole system.

The covariance matrix (i.e product $b^T b$) is symmetric with diagonal entries as $m - 1$ value and hence significant number of flops are saved by computing the (i, j) pairs which satisfy the condition $i < j$ in covariance matrix product. The diagonal entries are assigned a value equal to $m - 1$ while the remaining (i, j) pairs satisfying the condition $i > j$ are copied from the corresponding (j, i) pair. Hence the computational effort in forming the covariance matrix is only $\frac{n^2 - n}{2}$ dot products of two m -dimensional vectors excluding the assignment operations.

5.5 Two-sided Jacobi algorithm

The rotation matrices are very sparse(96.15% for 52x52 matrix), hence only the rotation parameters are only stored instead of storing them as a rotation matrices. A 16-stage CORDIC implementation of Two-sided Jacobi suffered from poor accuracy because of the inaccuracies in CORDIC combined with parallel annihilation in Jacobi method, hence the rotation parameters are computed from software *math.h* library. Other methods like angle recording and higher radix CORDIC have to be carefully examined for improving the accuracy and latency. The rotations are then performed by retrieving these parameters and multiplying them with corresponding pivots obtained from sequence generator. The sequence generator is implemented using the flow chart shown in fig. 4.2. This helps in significant saving in resources and computation time due to avoiding unnecessary multiplications with zeros. Compared to naive implementation (3 nested for-loops), we employed a two-nested for-loops. The second matrix in the matrix multiplication when transposed helps in reduced cache misses because the two dimension matrices are also arranged in row ordering fashion in memory.

5.6 Sorting

We require a sorting algorithm,

- To sort the singular values in descending order in Jacobi SVD
- To sort the array of similarity factors

We employed Bubble sort and Merge sort techniques as they are quite popular sorting algorithms. Bubble sort doesn't use a recursion and hence can be synthesized even on hardware but has worst, best and average time as $O(n)$ where as the Merge sort employs a recursion and has the worst, best and the average sorting time complexity as $O(\log n)$.

5.7 Calculation of similarity factors

For comparing two datasets, various similarity factors have been discussed in section 1.5. The geometrical interpretation of standard PCA similarity factor basically quantifies the angle between principal components of two datasets. The cosine of the angle between two principal components can be calculated as the dot product of the two vectors. Matrix multiplication has been significantly faster in Matlab because of the optimized BLAS / LAPACK routines compared to user defined nested loops. Equation 1.5 follows a matrix chain optimization but only the diagonal elements are made use in the final product. Hence this redundancy has been eliminated in the proposed simplified Similarity factor given in equation 1.6.

5.8 Working with Vivado HLS

Vivado **H**igh **L**evel **S**ynthesis(HLS) from Xilinx offers a great flexibility for specifying the design in high-level programming languages like C, C++ and System C compared to the conventional workflow with synthesis done from VHDL/Verilog programming. The tool provides the detailed analysis on timing like latency, resource utilization for each function, flexibility to export RTL in various formats. All the C-language codes are synthesized using Vivado HLS 2018.1. The tool optimally allocates the resources on its own for better performance and can be used when the whole logic has to be designed using only on hardware(PL:Programmable Logic).

We made use of Zedboard (ZC7020 Rev D) board as a platform for FPGA implementation as it is low-cost with built-in peripherals like USB OTG, Ethernet, VGA and SD card support. The final ASIC (Application Specific Integrated Chip) implementation can be further optimized in cost by selecting only the necessary peripherals. Table 5.1 summarizes the latency report with two different clock periods namely 5ns and 10ns. The detailed utilization and timing report can be found in HLS report folder. It has been found that the target device is too small for the design to be implemented as many multiplications are involved in various functions like computation of trigonometric terms, performing scaling with standard deviation and matrix multiplications even after *config_bind* command to minimize the multiplication operations.

Table 5.1: Latency report from Vivado HLS

Clock period	Latency	
	min	max
5 ns	37755687	72602195
10 ns	24254095	44164527

5.8.1 HLS with precomputed dataset

With pre-computation, the design has fit nicely into the target with utilization of DSP's, BRAM's, FF's and LUT's well under the limit so that atleast 4-5 such modules can be implemented in parallel. Since the target device has only a DDR3 RAM capacity upto 1GB which is very small when compared to our historical precomputed training dataset of size 32GB, we need special DMA transfers and memory interfacing with hard disk for scheduling the read and write operations. The latency of a single precomputed window with snapshot as IDV1 is listed with different clock cycles in Table 5.2. The latency will largely depend upon the number of PC's required for capturing the variance in the snapshot dataset.

Table 5.2: Latency report from HLS with pre-computation

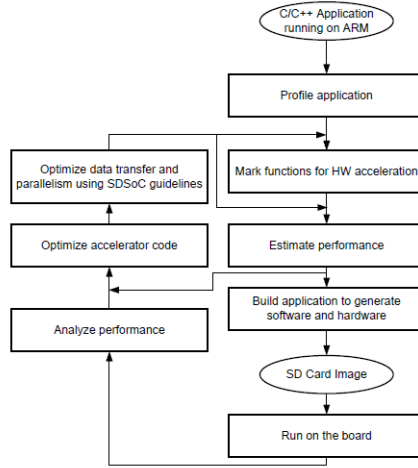
Clk period	Latency (min /max)
5 ns	4589
10 ns	4181

5.9 Working with SDSoC

Compared to Vivado HLS which is used for designing only PL, SDSoC provides better flexibility in designing a system which makes efficient use of PL and PS(Programmable System) by exploiting their advantages. The functions which are time-consuming and can be easily parallelizable can be made to run on PL by marking them for hardware whereas the functions which consumes bulk of resources or cannot be parallelizable can be run on PS. The design flow with SDSoC is as shown in the fig.5.1. It can be observed from the flowchart that the design process is cyclic until it meets the user specifications. The ARM Cortex A9 CPU can be run at maximum frequency of 666.67MHz while the PL can be made to run in any of the four available asynchronous clock frequencies namely 100MHz, 142.86MHz, 166.67MHz and 200MHz. There was large latency in preprocess and in covariance matrix formation as they deal with matrices of dimension $m \times n$ (i.e 500x52). No significant improvement has been observed with the available resources by unrolling and pipelining the loops in preprocess function.

Some of the challenges in Hardware software Co-design are listed below.

- The tool has a block RAM support of up to 16K which is small as we require 26K block RAM for our matrix dimension of 500x52. We need special interfaces like AXIMM ports for communication between hardware and software and sometimes most of the clock cycles are spent in this, thus affecting the overall design and poor performance (speed-up) even with Co-design.



source: Xilinx SDSoC Environment User Guide (UG1027)

Figure 5.1: Design flow with SDSoC

- Recursive functions other than Tail recursion are not supported for hardware implementation.
- We made use of pragma directive "data zero copy" for providing support for IN/OUT arrays and this also requires an AXIMM interface.
- Dynamic memory allocation functions like malloc, calloc and free are also prohibited in HLS.

The SDSoC provides Debug and Release configurations in addition to the customized user configuration for performance analysis. The tool also capture the baseline performance when the whole logic is made to implement only on PS(Processor System). The zynq board is provided with two serial ports and it is connected to our PC to display the output in real time as seen in the Teraterm window (COM5) in fig 5.2 and 5.3. With entire application made to run on a processor, it took on an average of 0.067 sec for processing a single window which is a promising result with a processor running at 666.67 MHz and comparable to Matlab's computation time of 1ms on a 2.8 GHz processor. The snapshot has been chosen from IDV1 for testing purpose and the similarity factors agreed to three decimal places. The performance results shown in fig.5.2 and fig.5.3 are obtained from Release mode of SDSoC 2018.1

Software only performance: Total time taken = $\frac{\text{Measuredcycles}}{\text{Clkfrequency}} = \frac{13482013984}{666.67 \times 10^6} = 20.2229 \text{sec}$
 Per unit window computation time = $\frac{20.2229}{300} = 0.067 \text{ sec.}$

Further careful designing and partitioning of hardware and software has to be made for achieving better speedup as it can be seen that the speed-up increased with increasing number of windows and with a large number(lakhs) of runs, it is expected to give satisfactory performance compared to Matlab's implementation.

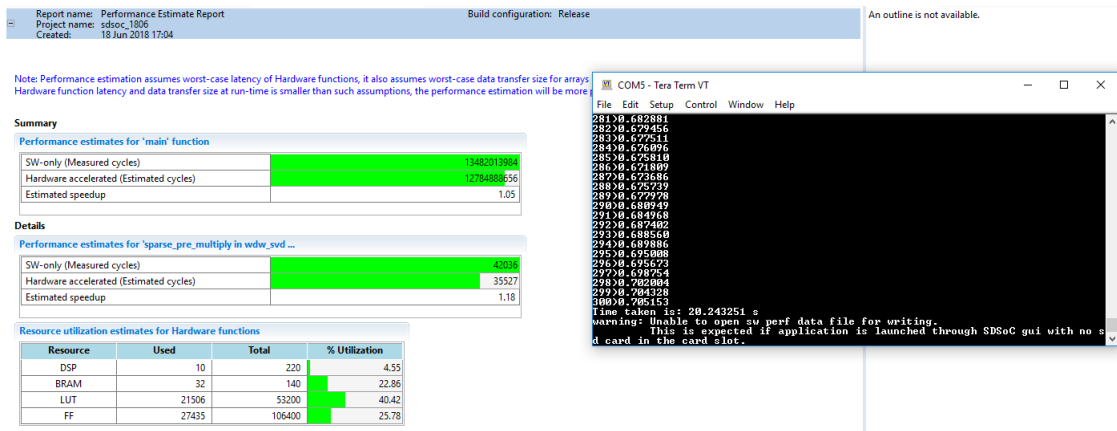


Figure 5.2: Software only performance for 300 windows

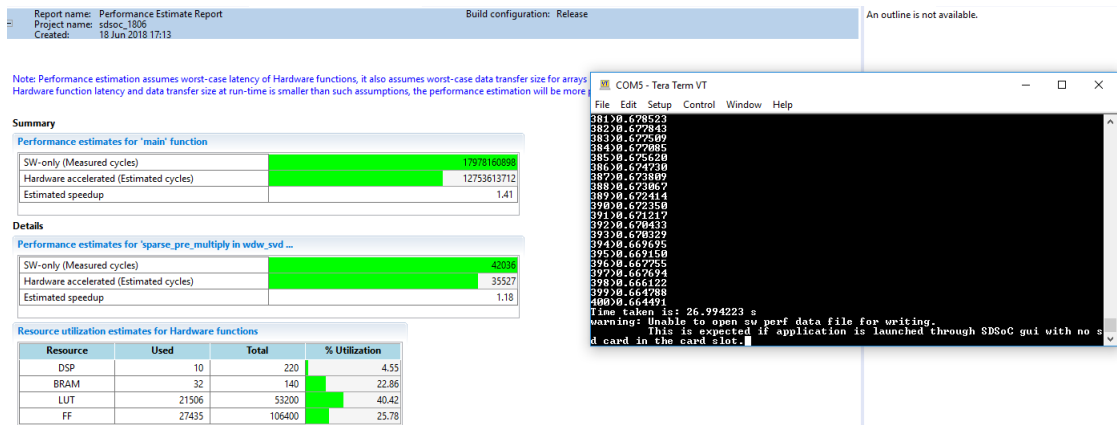


Figure 5.3: Software only performance for 400 windows

Chapter 6

Future work and Conclusion

6.1 Conclusion

The key research outcomes of this work are listed below.

- We proposed a pattern matching approach inspired from Ashish singhal et al in [3] with a smaller window size of 500 compared to the window size of 5761 as mentioned in their work. A large window size of 5761 (4days data) along with skipping of 500 windows all the time poses large difficulty in fault detection and pattern matching.
- We also imparted an accelerated pattern matching approach in case the similarity factor is less than a threshold of 0.5 and derived a simplified version of Standard similarity factor for reducing the latency.
- We proposed a hybrid Householder based Jacobi method which has better latency and accuracy compared to the existing Jacobi based methods which suffers from accuracy and Householder methods which suffers from large latency.
- A hardware implementation of the proposed pattern matching has been done which provides the flexibility to parallel processing of historical data and thereby reducing the fault detection time compared to the Software(Matlab) implementation.

6.2 Future work

- A large scope for further optimization in hardware implementation especially with SDSoC can be done by implementing the application project in Linux OS or with standalone OS with FAT file systems for reading historical data in an organized fashion.
- Optimal selection of window size and other parameters that can be manually chosen can be rounded off to the nearest powers of 2 as multiplication and division by these parameters can be done with simple shift operators instead of a complex DSP48E multiplier blocks. Loop optimization techniques like loop unrolling, pipelining, fusion and other techniques have to be manually done instead of tool for better allocation of resources.

- A new hybrid method of finding SVD for bi-diagonal or tri-diagonal matrices exploiting its sparsity has to be developed since dense matrices can be reduced to bi-diagonal or tri-diagonal form with householder reflections using simple shift operations. Since bi-diagonal methods of SVD are more accurate compared to Jacobi methods, the implementation of the former has to be explored using n-D CORDIC modules to reduce the latency in the bi-diagonalization.
- Case studies like CSTR and Fermentation process need to be reviewed and evaluated. Other data-driven methods namely Correspondence Analysis and Independent Component Analysis (ICA) all of which makes use of SVD can be applied to evaluate the performance.
- A computationally inexpensive and discriminating similarity factor which clearly distinguishes an operation from one another has to be developed.
- If the data has been labelled properly, a kind of supervised clustering can be done i.e the samples belonging to same operation can be grouped into a cluster. Since the normal operation prevails for most of the time and fault detection is more critical task than fault diagnosis, we can make use of faster detection methods like T^2 or Q - statistic tests or Machine learning techniques for detection and the moving window method may also be made to go through the cluster of normal windows. Once a fault detection has been identified with these statistical tests, the confirmation and diagnosis of the same can be obtained using the proposed moving window approach which now runs through the original unclustered historical database.

References

- [1] V. Venkatasubramanian, R. Rengaswamy, K. Yin, and S. N. Kavuri. A review of process fault detection and diagnosis: Part I: Quantitative model-based methods. *Computers & chemical engineering* 27, (2003) 293–311.
- [2] V. Venkatasubramanian, R. Rengaswamy, S. N. Kavuri, and K. Yin. A review of process fault detection and diagnosis: Part III: Process history based methods. *Computers & chemical engineering* 27, (2003) 327–346.
- [3] A. Singhal and D. E. Seborg. Evaluation of a pattern matching method for the Tennessee Eastman challenge process. *Journal of Process Control* 16, (2006) 601–613.
- [4] A. Singhal and D. E. Seborg. Pattern matching in multivariate time series databases using a moving-window approach. *Industrial & engineering chemistry research* 41, (2002) 3822–3838.
- [5] Z. Ge, Z. Song, and F. Gao. Review of recent research on data-based process monitoring. *Industrial & Engineering Chemistry Research* 52, (2013) 3543–3562.
- [6] A. Singhal and D. E. Seborg. Clustering multivariate time-series data. *Journal of chemometrics* 19, (2005) 427–438.
- [7] W. Hager. Bidiagonalization and diagonalization. *Computers and Mathematics with Applications* 14, (1987) 561 – 572.
- [8] G. H. Golub and C. F. Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.
- [9] R. Fierro. Numerical Linear Algebra (Lloyd N. Trefethen and David Bau, III). *SIAM REVIEW* 40, (1998) 735–738.
- [10] M. Gu and S. C. Eisenstat. A divide-and-conquer algorithm for the bidiagonal SVD. *SIAM Journal on Matrix Analysis and Applications* 16, (1995) 79–92.
- [11] M. Ariyaratne, T. Fernando, and S. Weerakoon. A self-tuning modified firefly algorithm to solve univariate nonlinear equations with complex roots. In *Evolutionary Computation (CEC), 2016 IEEE Congress on*. IEEE, 2016 1477–1484.
- [12] J.-M. Delosme. CORDIC algorithms: theory and extensions. In *Advanced Algorithms and Architectures for Signal Processing IV*, volume 1152. International Society for Optics and Photonics, 1989 131–146.

- [13] J.-M. Delosme and S.-F. Hsiao. CORDIC algorithms in four dimensions. In *Advanced Signal Processing Algorithms, Architectures, and Implementations*, volume 1348. International Society for Optics and Photonics, 1990 349–361.
- [14] P. K. Meher, J. Valls, T.-B. Juang, K. Sridharan, and K. Maharatna. 50 years of CORDIC: Algorithms, architectures, and applications. *IEEE Transactions on Circuits and Systems I: Regular Papers* 56, (2009) 1893–1907.
- [15] R. P. Brent and F. T. Luk. The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays. *SIAM Journal on Scientific and Statistical Computing* 6, (1985) 69–84.
- [16] G. E. Forsythe and P. Henrici. The cyclic Jacobi method for computing the principal values of a complex matrix. *Transactions of the American Mathematical Society* 94, (1960) 1–23.
- [17] S. Majumder, A. K. Shaw, and S. K. Sarkar. Hardware Implementation of Singular Value Decomposition. *Journal of The Institution of Engineers (India): Series B* 97, (2016) 227–231.
- [18] A. Ahmedsaid, A. Amira, and A. Bouridane. Improved SVD systolic array and implementation on FPGA. In *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on*. IEEE, 2003 35–42.