AFRL-SN-WP-TR-2007-1125

APERTURE AND RECEIVER TECHNOLOGY

Delivery Order 0002: Bandwidth Invariant Spatial Processing

Volume 2 - Digital Signal Processor (DSP) Based Implementation of Direction of Arrival (DOA) for Wideband Sources

Abdel Affo

The University of Toledo
Department of Electrical Engineering and Computer Science
Toledo, OH 43606

MAY 2007

Final Report for 21 December 2005 – 31 May 2007

Approved for public release; distribution is limited.

STINFO COPY

SENSORS DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AFB, OH 45433-7320

NOTICE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Air Force Research Laboratory Wright Site (AFRL/WS) Public Affairs Office (PAO) and is releasable to the National Technical Information Service (NTIS). It will be available to the general public, including foreign nationals.

THIS REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

*//signature//	//signature//
CHRISTOPHER R. REHM, Capt, USAF	JILL E. JOHNSON

This report is published in the interest of scientific and technical information exchange and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show "//signature//" stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS**.

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
May 2007	Final	12/21/2005 – 05/31/2007

4. TITLE AND SUBTITLE
APERTURE AND RECEIVER TECHNOLOGY
Delivery Order 0002: Bandwidth Invariant Spatial Processing
Volume 2 - Digital Signal Processor (DSP) Based Implementation of Direction of Arrival (DOA) for Wideband Sources

5a. CONTRACT NUMBER FA8650-05-D-1848-0002
5b. GRANT NUMBER
5c. PROGRAM ELEMENT NUMBER 62204F

6. AUTHOR(S)
Abdel Affo

5d. PROJECT NUMBER 7622
5e. TASK NUMBER 11
5f. WORK UNIT NUMBER 7622110L

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
The University of Toledo
Department of Electrical Engineering and Computer Science
Toledo, OH 43606

8. PERFORMING ORGANIZATION REPORT NUMBER
DSPH-11

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Sensors Directorate
Air Force Research Laboratory
Air Force Materiel Command
Wright-Patterson AFB, OH 45433-7320

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL-SN-WP

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-SN-WP-TR-2007-1125

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
Report contains color. PAO Case Number: AFRL/WS-07-1443, 18 June 2007. Report contains color.

14. ABSTRACT
Sensor arrays are used in many digital signal processing applications due to their ability to locate signal sources. For most of these applications, it is necessary to estimate the Direction of Arrival (DOA) of the sources. Numerous algorithms have been developed, but most of them focus on narrowband signals where the time delay can be approximated as a phase shift. This thesis focuses on the coherent signal subspace method for DOA estimation of wideband sources. However, the coherent signal subspace method has high computational requirements that prevented its use in many real-time applications. With the growth in technology, it's now possible to implement these algorithms in real-time embedded systems. These array signal processing algorithms can be implemented in hardware using Digital Signal Processors (DSP), Field Programmable Gate Arrays (FPGA), and Application-Specific Integrated Circuits (ASIC). DSPs offer flexibility and also the best development time and cost with proper use of high-level programming. In this thesis, we propose a DSP based architecture for detecting and estimating the DOA of wideband sources. Also, a parallelized algorithm is presented and its performance is evaluated with respect to the original architecture.

15. SUBJECT TERMS
Wide-band, Direction of Arrival, Array Processing, DSP Processor

16. SECURITY CLASSIFICATION OF: a. REPORT	b. ABSTRACT	c. THIS PAGE	17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
Unclassified	Unclassified	Unclassified	SAR	110	Capt. Christopher Rehm **19b. TELEPHONE NUMBER** *(Include Area Code)* (937) 255-5579, ext. 3728

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

TABLE OF CONTENTS

Acknowledgments……………………………………………………………………vii

List of Figures………………………………………………………………..….........iv

List of Tables...v

I INTRODUCTION...1

1.1 NARROWBAND SOURCES 3

1.1.1 Signal model 3

1.1.2 Multiple signal classification algorithm:...4

1.1.3 Estimating number of sources: 8

1.2 WIDEBAND SOURCES..9

1.2.1 Signal model 9

1.2.2 Coherent signal subspace method for wideband sources 10

1.3 THESIS ORGANIZATION 12

II EIGENDECOMPOSITION USING HOUSEHOLDER MATRICES 13

2.1 HOUSEHOLDER ALGORITHM 13

2.2 QR FACTORIZATION...18

III DSP IMPLEMENTATION OF CSS ALGORITHM 24

3.1 DIGITAL SIGNAL PROCESSOR 24

3.1.1 Microcontroller 24

3.1.2 Software..27

3.1.3 Evaluation board 27

3.2 SIMULATION 28

3.2.1 Sampling..29

3.2.2 Data generation 30

3.2.3 Covariance matrix computation..32

3.2.4 Eigendecomposition of covariance matrix 34

3.2.5 Power spectrum..36

3.2.6 Coherent signal subspace processing 37

3.3 SIMULATION RESULTS 37

IV PARALLEL ARCHITECTURE FOR COHERENT SIGNAL SUBSPACE ALGORITHM 41

4.1 COVARIANCE MATRIX COMPUTATION 41

4.2 EIGENVALUES AND EIGENVECTORS COMPUTATION 44

4.2.1 Householder Method 46

4.2.2 QR method..48

4.2.3 Power Spectrum 50

4.3 SIMULATIONS 50

V CONCLUSION 53

5.1 SUMMARY 53

5.2 FUTURE WORK..54

VI REFERENCES..55

VII APPENDIX 57

LIST OF TABLES

Table 4.1: Performance results for single DSP and parallel architecture 51

LIST OF FIGURES

Figure 2.1: Flowchart of function House(x, k, n, w) 19

Figure 2.2: Flowchart of function Sym (A) 20

Figure 2.3: Flowchart of function QR (A, Q, R) 22

Figure 2.4: Flowchart of function QRShift (A, n) 23

Figure 3.1 mAgic DSP Block Diagram 26

Figure 3.2 JTST Layout 29

Figure 3.3: Power spectrum for initial DOA estimate of angles $\theta_1 = 50^\circ$ and $\theta_2 = 20^\circ$ 38

Figure 3.4: Power spectrum for final DOA estimate of angles $\theta_1 = 50^\circ$ and $\theta_2 = 20^\circ$ 38

Figure 3.5: Power spectrum for initial DOA estimate of angles $\theta_1 = 30^\circ$ and $\theta_2 = 40^\circ$ 39

Figure 3.6: Power spectrum for final DOA estimate of angles $\theta_1 = 30^\circ$ and $\theta_2 = 40^\circ$ 39

Figure 4.1: Parallel architecture for coherent signal subspace algorithm 43

Figure 4.2 Flowchart of parallel Householder method 47

Figure 4.3: Flowchart of parallel QR method 49

Figure 4.4: Power spectrum for final DOA estimate of angles $\theta_1 = 20^\circ$ and $\theta_2 = 60^\circ$ using single and parallel DSP approach 51

ACKNOWLEDGEMENTS

I would like to express my deep sense of gratitude to my advisor Dr. Mohsin Jamali for his encouragement, guidance and support during this research. I would like to thank Dr. Junghwan Kim and Dr. Mohamed El Bialy for serving in my committee.

I would also like to express my sincere thanks to Atmel Corporation for being a valuable source of help during this research.

Finally, I would like to express my sincere gratitude to my parents for their support and understanding.

Chapter 1

Introduction

Array processing has been an important part of signal processing in the past few years. It can be defined as the use of an array of sensors to receive and sample signals. It then processes the received data in order to obtain some information about the sampled signals. Some of that information includes the number of sources, the locations of those sources and the type of signals being emitted. The array consists of sensors located at different points in space with respect to a reference point. There are two types of sensor array systems: active and passive. In the active case, a known waveform is generated and reflected by the target. In the passive case, the signal received by the array is generated by the target.

Direction of Arrival (DOA) denotes the direction from which the wave fields arrive at the sensor array. The goal in DOA detection and estimation is to accurately determine the number of sources producing waveforms and the locations of those sources. Some of its applications include cellular communications, air traffic control, seismology, sonar and bioengineering.

Many algorithms have been proposed to solve the DOA estimation problem. The Beamforming method [1] is one of the simplest techniques for DOA estimation. This method consists in computing the output signal using a weighted sum of the sensor

outputs. The DOA is determined by locating the peaks in the power spectrum of the array output. This technique has poor resolution and requires a great number of sensors for a good estimation of the DOA.

The Maximum Likelihood approach [2] was proposed by Bohme in 1984. Using this method, the signals are modeled as unknown deterministic elements, rather than random signals. The Maximum Likelihood Estimator of the DOA is obtained by searching over the array manifold for the D steering vectors that form a signal subspace that is closest to the array data input matrix. Closeness is measured by the modulus of the orthogonal projection of the input vectors onto that subspace. This technique has high resolution, but is very computationally intensive.

The Multiple Signal Classification algorithm (MUSIC) [3] was proposed by Schmidt in 1987. It is one of the most popular techniques for DOA estimation because it has a high resolution, but is also less computationally intensive than the Maximum Likelihood technique. MUSIC makes use of the eigenstructure of the input covariance matrix by separating the eigenvectors into orthogonal subspaces. The MUSIC approach was first proposed for the narrowband case [4-6]. It can also be used to determine the DOA of wideband signals after separation of those signals into narrowband components [7-9].

In this thesis, a parallel architecture for DOA estimation of wideband sources is devised. The algorithm used is called the Coherent Signal-Subspace (CSS) method and was proposed by Wang and Kaveh [10]. This technique separates the wide frequency band into narrowband components. The covariance matrices for all the frequency components are estimated and combined to form a single focused covariance matrix. The

narrowband MUSIC algorithm can then be applied to the resulting focused covariance matrix.

In this chapter, the narrowband signal model is given and the MUSIC algorithm is presented. The wideband data model is also presented along with the Coherent Signal Subspace method for wideband sources.

1.1 Narrowband sources

1.1.1 Signal model

Let us consider an array of *M* identical sensors. Assuming that *D* narrowband sources (*D* < *M*) with identical bandwidth impinge on the array from directions $\theta_1, \theta_2, \ldots, \theta_D$, the received signal at the ith sensor can be expressed as:

$$\mathbf{x}_i(t) = \sum_{k=1}^{D} \mathbf{a}_{ik} \mathbf{s}_k(t - \tau_{ik}) + \mathbf{n}_i(t) \quad (1.1)$$

where $s_k(t)$ is the signal at a reference point, τ_{ik} is the propagation delay between the reference point and the ith sensor for the kth source, a_{ik} is the impulse response of the ith sensor to the kth source, and $n_i(t)$ is the additive noise at the ith sensor. Since $s_k(t)$ is a narrowband process, we can approximate the time delay by a phase shift [11]. Equation (1.1) can be rewritten as:

$$\mathbf{x}_i(t) = \sum_{k=1}^{D} \mathbf{a}_{ik} e^{-jw_0 \tau_{ik}} \mathbf{s}_k(t) + \mathbf{n}_i(t) \quad (1.2)$$

$$\tau_{ik} = \Delta \frac{\sin\theta}{c} \quad (1.3)$$

where Δ is the distance between the reference point and the sensor, θ is the direction of arrival and c is the velocity of the wave. The model used to represent the output of the M sensors is:

$$\mathbf{X}(t) = \mathbf{A}(\theta)\mathbf{S}(t) + \mathbf{N}(t) \tag{1.4}$$

$$\mathbf{A}(\theta) = \left[\mathbf{a}(\theta_1), \quad \mathbf{a}(\theta_2), \quad \cdots \quad \mathbf{a}(\theta_D)\right] \tag{1.5}$$

$$\mathbf{a}(\theta_K) = \begin{bmatrix} a_1(\theta_k)e^{-jw_0\tau_1(\theta_k)} \\ \vdots \\ a_M(\theta_k)e^{-jw_0\tau_M(\theta_k)} \end{bmatrix} \tag{1.6}$$

Matrix $\mathbf{A}$ is called the direction matrix and each of the column vectors are the direction vectors of the sources. Assuming that the noise is independent of the signal with zero mean, the covariance matrix can be expressed as:

$$\mathbf{R}_{xx} = E\left[\mathbf{X}(t)\mathbf{X}^H(t)\right] \tag{1.7}$$

$$\mathbf{R}_{xx} = \mathbf{A}(\theta)E\left[\mathbf{S}(t)\mathbf{S}^H(t)\right]\mathbf{A}^H(\theta) + E\left[\mathbf{N}(t)\mathbf{N}^H(t)\right] \tag{1.8}$$

If the output array vector $\mathbf{X}(t)$ is observed over K subintervals of duration ΔT seconds each, the covariance matrix [10] can be expressed as the snapshot averaged cross-product of $\mathbf{x}_k(t)$:

$$\mathbf{R}_{xx} = \frac{1}{K}\sum_{k=1}^{K}\mathbf{x}_k(t)\mathbf{x}_k^H(t) \tag{1.9}$$

1.1.2 Multiple signal classification algorithm:

The MUSIC (Multiple Signal Classification) algorithm is a high resolution technique based on some properties of the input covariance matrix. Following the signal

model discussed, the received data vector can be expressed as a linear combination of the incident waveforms and noise [12]:

$$\mathbf{X}(t) = \mathbf{A}(\theta)\mathbf{S}(t) + \mathbf{N}(t) \tag{1.10}$$

or

$$\mathbf{X} = \mathbf{A}\mathbf{S} + \mathbf{N} \tag{1.11}$$

where $\mathbf{S} = [\mathbf{s}_1(t), \mathbf{s}_2(t), \ldots, \mathbf{s}_D(t)]^T$ is the incident vector and $\mathbf{N} = [\mathbf{n}_1(t), \mathbf{n}_2(t), \ldots, \mathbf{n}_D(t)]^T$ is the noise vector.

The input covariance matrix can be expressed as:

$$\mathbf{R}_{xx} = E\left[\mathbf{X}\mathbf{X}^H\right] \tag{1.12}$$

$$\mathbf{R}_{xx} = \mathbf{A}E\left[\mathbf{S}\mathbf{S}^H\right]A^H + E\left[\mathbf{N}\mathbf{N}^H\right] \tag{1.13}$$

$$\mathbf{R}_{xx} = \mathbf{A}\mathbf{R}_{ss}\mathbf{A}_H + \sigma_n^2\mathbf{I} \tag{1.14}$$

where $\mathbf{R}_{ss}$ is the signal correlation matrix $E\left[\mathbf{S}\mathbf{S}^H\right]$ and σ_n^2 is the noise variance. The eigenvalues of the input covariance matrix $\mathbf{R}_{xx}$ are the values $\lambda_1, \lambda_2, \ldots, \lambda_M$ that satisfy the following equation:

$$\left|\mathbf{R}_{xx} - \lambda_i\mathbf{I}\right| = 0 \tag{1.15}$$

We can rewrite the previous equation as

$$\left|\mathbf{A}\mathbf{R}_{ss}\mathbf{A}^H + \sigma_n^2\mathbf{I} - \lambda_i\mathbf{I}\right| = 0 \tag{1.16}$$

$$\left|\mathbf{A}\mathbf{R}_{ss}\mathbf{A}^H - \left(\lambda_i - \sigma_n^2\right)\mathbf{I}\right| = 0 \tag{1.17}$$

Therefore, the eigenvalues of $\mathbf{A}\mathbf{R}_{ss}\mathbf{A}^H$ are

$$v_i = \lambda_i - \sigma_n^2 \tag{1.18}$$

Matrix $\mathbf{A}$ is composed of linearly independent direction vectors and therefore has full column rank D. The signal correlation matrix is non singular as long as the incoming signals are not highly correlated. As a result, if the number of signals D is less than the number of array sensors M, the M x M matrix $\mathbf{A}\mathbf{R}_{ss}\mathbf{A}^H$ is positive semi definite with rank D. This implies that the matrix $\mathbf{A}\mathbf{R}_{ss}\mathbf{A}^H$ had D non-zero eigenvalues. From (1.18), we see that M - D eigenvalues of $\mathbf{R}_{xx}$ are equal to the noise variance σ_n^2 . After sorting the eigenvalues of $\mathbf{R}_{xx}$ so that λ_1 is the largest value and λ_M the smallest, we have

$$\lambda_{D+1},\ldots,\lambda_M = \sigma_n^2 \tag{1.19}$$

The eigenvector associated with the eigenvalue λ_i is the vector $\mathbf{q}_i$ such that

$$\left(\mathbf{R}_{xx} - \lambda_i \mathbf{I}\right)\mathbf{q}_i = 0 \tag{1.20}$$

For the eigenvectors associated with the M - D eigenvalues, we have

$$\left(\mathbf{R}_{xx} - \lambda_i \mathbf{I}\right)\mathbf{q}_i = 0 \tag{1.21}$$

$$\left(\mathbf{R}_{xx} - \sigma_n^2 \mathbf{I}\right)\mathbf{q}_i = 0 \tag{1.22}$$

$$\left(\mathbf{A}\mathbf{R}_{ss}\mathbf{A}_H + \sigma_n^2 \mathbf{I} - \sigma_n^2 \mathbf{I}\right)\mathbf{q}_i = 0 \tag{1.23}$$

$$\mathbf{A}\mathbf{R}_{ss}\mathbf{A}^H\mathbf{q}_i = 0 \tag{1.24}$$

Since $\mathbf{A}$ has full rank and $\mathbf{R}_{ss}$ is non singular, we have

$$\mathbf{A}^H\mathbf{q}_i = 0 \tag{1.25}$$

$$\begin{bmatrix} \mathbf{a}^H(\theta_1)\mathbf{q}_i \\ \vdots \\ \mathbf{a}^H(\theta_D)\mathbf{q}_i \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix} \tag{1.26}$$

This means that the eigenvectors associated with the M - D smallest eigenvalues are orthogonal to the direction vectors making up $\mathbf{A}$. These observations form the basis of the MUSIC algorithm. The analysis shows that the eigenvectors of the covariance matrix belong to two orthogonal subspaces called the signal and noise subspaces. We can estimate the direction of arrival by finding direction vectors which lie in the signal subspace. These vectors are the direction vectors that are orthogonal to the noise subspace. To search the noise subspace, we form a matrix containing the noise eigenvectors:

$$\mathbf{V}_n = \begin{bmatrix} \mathbf{q}_{D+1} & \cdots & \mathbf{q}_D \end{bmatrix} \tag{1.27}$$

Since the direction vectors of the incoming signals are orthogonal to the noise subspace eigenvectors, we can say:

$$\mathbf{a}^H(\theta)\mathbf{V}_n\mathbf{V}_n^H\mathbf{a}(\theta) = 0 \tag{1.28}$$

where θ is the direction of arrival of a signal component.

The direction of arrival can then be estimated by finding the peaks of the MUSIC spectrum given by:

$$p(\theta) = \frac{1}{\mathbf{a}^H(\theta)\mathbf{V}_n\mathbf{V}_n^H\mathbf{a}(\theta)}, \theta \in [0, 2\pi] \tag{1.29}$$

The D largest peaks in the spectrum will correspond to the directions of arrival of the signals impinging on the sensor array.

The MUSIC algorithm can be summarized as follow:

1. Collect input samples $\mathbf{x}_k(t)$
2. Estimate the covariance matrix given by:

$$\mathbf{R}_{xx} = \frac{1}{K}\sum_{k=1}^{K}\mathbf{x}_k(t)\mathbf{x}_k^H(t)$$

3. Compute eigenvalues and eigenvectors of the covariance matrix.
4. Find number of sources *D*.
5. Find the DOA estimates by finding the *D* largest peaks of the MUSIC spectrum given by:

$$p(\theta) = \frac{1}{\mathbf{a}^H(\theta)\mathbf{V}_n\mathbf{V}_n^H\mathbf{a}(\theta)}$$

1.1.3 Estimating number of sources:

The MUSIC algorithm depends on the parameter *D*, the numbers of wave fronts impinging on the sensor array, for estimating the DOA of multiple targets. The Minimum Description Length (MDL) was used in order to achieve this objective [13]. The MDL is specified by:

$$\text{MDL}(D) = -2\log\left(\frac{\prod_{k=D+1}^{M}\lambda_k^{\frac{1}{(M-D)}}}{\frac{1}{M-D}\sum_{k=D+1}^{M}\lambda_k}\right) + \frac{1}{2}(2M-D)\log(N) \qquad (1.30)$$

where N is the number of samples, λ represents the eigenvalues of the covariance matrix and M is the number of sensors. The number of sources is determined by finding a value of D which minimizes the MDL criterion. The maximum number of sources that can be estimated is M - 1.

1.2 Wideband sources

1.2.1 Signal model

Let us consider as before an array of M sensors. Assuming that D wideband sources with identical bandwidth B impinge on the array from directions $\theta_1, \theta_2, \ldots, \theta_D$, we can use Equation (1.1) to represent the signal received at the ith sensor. However, the time delay cannot be approximated as a phase shift in the wideband case. Assuming that the signals are observed over a finite interval T, we can represent the signal $\mathbf{x}_i$ by a Fourier series [11]:

$$\mathbf{x}_i(t) = \sum_{n=l}^{l+m} X_i(w_n) e^{jw_n t} \tag{1.31}$$

where $X_i(w_n)$ are the Fourier coefficients given by:

$$X_i(w_n) = \frac{1}{T^{1/2}} \int_{-T/2}^{T/2} x_i(t) e^{jw_n t} dt \tag{1.32}$$

$$w_n = \frac{2\pi}{T} n, \quad n = l, \ldots, l+m \tag{1.33}$$

where w_l is the lowest frequency and w_{l+m} is the highest frequency included in the bandwidth B. Assuming that the observation time T is much greater than the propagation delay across elements of the array, we can use a phase shift as an approximation of the time delay in the Fourier domain.

$$\mathbf{x}_i(w_n) = \sum_{k=1}^{D} \mathbf{a}_{ik} e^{-jw_n \tau_{ik}} \mathbf{s}_k(w_n) + \mathbf{n}_i(w_n) \tag{1.34}$$

The model used to represent the output vector is:

$$\mathbf{X}(w_n) = \mathbf{A}(w_n) \mathbf{S}(w_n) + \mathbf{N}(w_n) \tag{1.35}$$

$$\mathbf{A}(w_n) = \left[\mathbf{a}_{\theta 1}(w_n), \quad \mathbf{a}_{\theta 2}(w_n), \quad \cdots \quad \mathbf{a}_{\theta D}(w_n)\right] \tag{1.36}$$

$$\mathbf{a}_{\theta k}(w_n) = \begin{bmatrix} a_1(\theta_k)e^{-jw\tau_1(\theta_k)} \\ \vdots \\ a_M(\theta_k)e^{-jw\tau_M(\theta_k)} \end{bmatrix} \tag{1.37}$$

As a result of the Fourier transform applied over a time segment ΔT, the array output vector is decomposed into non-overlapping narrowband components. The covariance matrix for component w_n can be expressed as:

$$\mathbf{R}_{xx}(w_n) = E\left[\mathbf{x}(w_n)\mathbf{x}^H(w_n)\right] \tag{1.38}$$

$$\mathbf{R}_{xx}(w_n) = \mathbf{A}(w_n)E\left[\mathbf{S}(w_n)\mathbf{S}^H(w_n)\right]\mathbf{A}^H(w_n) + E\left[\mathbf{N}(w_n)\mathbf{N}^H(w_n)\right] \tag{1.39}$$

This covariance matrix can also be expressed as:

$$\mathbf{R}_{xx}(w_n) = \frac{1}{K}\sum_{k=1}^{K}\mathbf{x}_k(w_n)\mathbf{x}_k^H(w_n) \tag{1.40}$$

1.2.2 Coherent signal subspace method for wideband sources

It is possible to combine the signal subspaces of different frequencies in order to generate a single subspace that will allow us to determine the correct number of sources and directions of arrival [10]. The matrix $\mathbf{R}$ can be used to find the final DOA estimates. This matrix can be formed by:

$$\mathbf{R} = \sum_{j=1}^{J} \mathbf{T}(w_j)\mathbf{R}_{xx}(w_j)\mathbf{T}^H(w_j) \tag{1.41}$$

where J is the number of narrowband components. The matrices $\mathbf{T}$ are called transformation matrices and can be expressed as

$$\mathbf{T}(w_j) = \begin{bmatrix} a_{1\beta}(w_o)/a_{1\beta}(w_j) & & & \\ & a_{2\beta}(w_o)/a_{2\beta}(w_j) & & \\ & & \ddots & \\ & & & a_{M\beta}(w_o)/a_{M\beta}(w_j) \end{bmatrix} \tag{1.42}$$

where w_o is the central frequency of bandwidth B, β is the initial DOA value, and $a_{i\beta}(w_j)$ is the ith element of the direction vector $\mathbf{a}_\beta(w_j)$.

The coherent signal subspace method for computation of DOA (wideband sources) as proposed by Wang & Kaveh [10] can be summarized as follow:

1. Collect data samples and convert the samples into frequency domain using FFT.
2. Estimate the covariance matrix for each frequency component given by:

$$\mathbf{R}_{xx}(w_n) = \frac{1}{K}\sum_{k=1}^{K} \mathbf{x}_k(w_n)\mathbf{x}_k^H(w_n)$$

3. Compute eigenvalues and eigenvectors of the covariance matrix.
4. Find initial estimates of direction of arrival by computing the MUSIC spectrum given by:

$$p(\phi) = \frac{1}{\mathbf{a}^H(\theta)\mathbf{V}_n\mathbf{V}_n\mathbf{a}(\theta)}$$

5. Compute transformation matrix focusing on central frequency.

6. Compute eigenvalues and eigenvectors of the focus matrix.
7. Find number of sources *D*.
8. Find the final DOA estimates by finding the *D* largest peaks of the MUSIC spectrum.

1.3 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, we present some of the methods that can be used to compute the eigenvalues and eigenvectors of the covariance matrix. Chapter 3 gives the simulations details for a single DSP implementation of the Coherent Signal Subspace method and their results. In Chapter 4, a parallel Coherent Signal Subspace algorithm is presented and its performance is measured. Chapter 5 presents the conclusion and future research topics.

Chapter 2

Eigendecomposition using Householder Matrices

The eigendecomposition problem is a very important part of the DOA estimation algorithm. As explained in the previous chapter, finding the eigenvalues and eigenvectors of the covariance matrix is needed to construct the signal and noise subspaces that the MUSIC algorithm will use. The Householder and QR algorithms [14-15] can be used to compute the eigenvalues and eigenvectors of the symmetric covariance matrix. The Householder algorithm is used to reduce the bandwidth of the covariance matrix by transforming it into tridiagonal form. The eigenvalues and eigenvectors can then be computed using the QR algorithm. The computational cost needed for computing the eigenvalues and eigenvectors of a tridiagonal matrix will be much smaller than that of the original symmetric matrix. This chapter provides a brief narrative outlining the mechanics of the Householder and QR algorithms.

2.1 Householder algorithm

The Householder algorithm [14-15] reduces an $n \times n$ symmetric matrix $\mathbf{A}$ to tridiagonal form by n-2 orthogonal transformations. Each of the transformations is used to eliminate part of a column and its corresponding row. By similarity, the tridiagonal matrix **B** and the original symmetric matrix **A** have the same eigenvalues. Furthermore, it

can be proven that if $(\mathbf{x}, \lambda)$ is an eigenpair of $\mathbf{A}$ and $\mathbf{N}$ is an orthogonal matrix, then $(\mathbf{N}^H \mathbf{x}, \lambda)$ is an eigenpair of $\mathbf{N}^H \mathbf{R} \mathbf{N}$. As a result, it is possible to find the eigenvectors of matrix $\mathbf{A}$ using orthogonal transformations on the eigenvectors of $\mathbf{B}$.

An n x n matrix $\mathbf{H}$ is called a Householder matrix if

$$\mathbf{H} = \mathbf{I} - 2\mathbf{w}\mathbf{w}^T \tag{2.1}$$

This equation can be rewritten as

$$\mathbf{H} = \mathbf{I} - \frac{\mathbf{u} \cdot \mathbf{u}^T}{\|\mathbf{u}\|_2} \tag{2.2}$$

where $\mathbf{w} = \dfrac{\mathbf{u}}{\|\mathbf{u}\|_2}$ and $\|\mathbf{u}\|_2 = \mathbf{u}^2$.

Note that $\mathbf{H}$ is symmetric and orthogonal.

Let us consider $\mathbf{H}$ such that

$$\mathbf{H}\mathbf{x} = \alpha \mathbf{e}_1 \tag{2.3}$$

where α is a constant, $\mathbf{e}_1$ is the unit vector $[1,0,\ldots,0]$ and $\mathbf{x} = [x_1, \ldots, x_n]^T$

It is possible to set $\mathbf{u} = \mathbf{x} + \beta \mathbf{e}_1$ and determine β such that $\mathbf{H}\mathbf{x}$ doesn't have any component in the direction of $\mathbf{x}$, as specified in (2.3).

$$\mathbf{w} = \frac{\mathbf{u}}{\|\mathbf{u}\|_2} = \frac{\mathbf{x} + \beta \mathbf{e}_1}{\|\mathbf{x} + \beta \mathbf{e}_1\|_2} \tag{2.4}$$

$$\mathbf{H}\mathbf{x} = \mathbf{x} - 2\mathbf{w}\mathbf{w}^T \mathbf{x} \tag{2.5}$$

$$\mathbf{H}\mathbf{x} = \mathbf{x} - 2\left(\frac{\mathbf{x} + \beta \mathbf{e}_1}{\|\mathbf{x} + \beta \mathbf{e}_1\|_2}\right) \cdot \frac{(\mathbf{x} + \beta \mathbf{e}_1)^T}{\|\mathbf{x} + \beta \mathbf{e}_1\|_2} \cdot \mathbf{x} \tag{2.6}$$

$$\mathbf{Hx} = \mathbf{x} - 2\frac{\left(\mathbf{x} + \beta\mathbf{e}_1\right)}{\left\|\mathbf{x} + \beta\mathbf{e}_1\right\|_2} \cdot \frac{\left(\mathbf{x}^T\mathbf{x} + \beta\mathbf{e}_1^T\mathbf{x}\right)}{\left\|\mathbf{x} + \beta\mathbf{e}_1\right\|_2} \tag{2.7}$$

$$\mathbf{Hx} = \mathbf{x} - 2\frac{\mathbf{x}\left(\mathbf{x}^T\mathbf{x} + \beta\mathbf{e}_1^T\mathbf{x}\right) + \beta\mathbf{e}_1\left(\mathbf{x}^T\mathbf{x} + \beta\mathbf{e}_1^T\mathbf{x}\right)}{\left\|\mathbf{x} + \beta\mathbf{e}_1\right\|_2^2} \tag{2.8}$$

Since $\beta\mathbf{e}_1^T\mathbf{x} = \beta x_1$

$$\mathbf{Hx} = \frac{\mathbf{x}\left\|\mathbf{x} + \beta\mathbf{e}_1\right\|_2^2 - 2\mathbf{x}\left(\left\|\mathbf{x}\right\|_2^2 + \beta x_1\right) - 2\beta\mathbf{e}_1\left(\left\|\mathbf{x}\right\|_2^2 + \beta x_1\right)}{\left\|\mathbf{x} + \beta\mathbf{e}_1\right\|_2^2} \tag{2.9}$$

$$\mathbf{Hx} = \frac{\mathbf{x}\left(\left\|\mathbf{x}\right\|_2^2 + 2\beta x_1 + \beta^2 - 2\left\|\mathbf{x}\right\|_2^2 - 2\beta x_1\right) - 2\beta\mathbf{e}_1\left(\left\|\mathbf{x}\right\|_2^2 + \beta x_1\right)}{\left\|\mathbf{x} + \beta\mathbf{e}_1\right\|_2^2} \tag{2.10}$$

$$\mathbf{Hx} = \frac{\mathbf{x}\left(\beta^2 - \left\|\mathbf{x}\right\|_2^2\right) - 2\beta\mathbf{e}_1\left(\left\|\mathbf{x}\right\|_2^2 + \beta x_1\right)}{\left\|\mathbf{x} + \beta\mathbf{e}_1\right\|_2^2} \tag{2.11}$$

For the assumption in (2.3) to be valid, we need

$$\beta^2 = \left\|\mathbf{x}\right\|_2^2 \tag{2.12}$$

$$\beta = \pm\left\|\mathbf{x}\right\|_2 \tag{2.13}$$

$$\mathbf{u} = \begin{bmatrix} x_1 \pm \left\|\mathbf{x}\right\|_2 & x_2 \cdots & x_n \end{bmatrix}^T \tag{2.14}$$

The sign of β will be chosen to be the same as the sign of x_1.

As a result, the vector $\mathbf{w}$ can be expressed as:

$$\mathbf{w} = \frac{1}{\sqrt{2\|\mathbf{x}\|_2 (\|\mathbf{x}\|_2 + |x_1|)}} \begin{bmatrix} x_1 + sign(x_1)\|\mathbf{x}\|_2 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{2.15}$$

From (2.11), we have

$$\mathbf{Hx} = -sign(x_1)\|\mathbf{x}\|_2 \, \mathbf{e}_1 \tag{2.16}$$

and

$$\alpha = -sign(x_1)\|\mathbf{x}\|_2 \tag{2.17}$$

With the appropriate choice of β , the Householder matrix **H** is used to zero out part of a vector. For example, if

$$\mathbf{u} = \begin{bmatrix} 0 & 0 & \cdots & x_k + sign(x_k)\|\mathbf{x}\|_2 & x_{k+1} & \cdots & x_n \end{bmatrix}^T \tag{2.18}$$

then

$$\mathbf{Hx} = \begin{bmatrix} x_1 & x_2 & \cdots & x_{k-1} & -sign(x_k)\|\mathbf{x}\|_2 & 0 & 0 \end{bmatrix}^T \tag{2.19}$$

H has the form:

$$\begin{bmatrix} \dfrac{[\mathbf{I}_{k-1}][0]}{[0]\left[\hat{\mathbf{H}}\right]} \end{bmatrix}$$

where $\hat{\mathbf{H}}$ is the Householder matrix such that

$$\hat{\mathbf{H}}\begin{bmatrix} x_k \\ \vdots \\ x_n \end{bmatrix} = \alpha\mathbf{e}_1 \tag{2.20}$$

where α is a constant.

The tridiagonalization process begins by computing the 1st Householder matrix, where $\mathbf{x}$ is chosen to be the lower n-1 elements of the first column of $\mathbf{A}$. As a result, the lower n-2 elements are eliminated.

$$\mathbf{H}_1\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & & & & \\ 0 & & \hat{\mathbf{H}} & & \\ 0 & & & & \\ 0 & & & & \end{bmatrix}\begin{bmatrix} a_{11} & a_{21} & \cdots & \cdots & a_{n1} \\ a_{21} & & & & \\ \vdots & & & & \\ \vdots & & & & \\ a_{n1} & & & & \end{bmatrix} \tag{2.21}$$

$$\mathbf{H}_1\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ k & & & \\ 0 & & & \\ 0 & & & \\ \vdots & & & \end{bmatrix} \tag{2.22}$$

The first orthogonal transformation is:

$$\mathbf{A}_1 = \mathbf{HAH} = \begin{bmatrix} a_{11} & k & 0 & 0 \\ k & & & \\ 0 & & & \\ 0 & & & \\ 0 & & & \end{bmatrix} \tag{2.23}$$

The next step involves the computation of the second Householder matrix using the lower n-2 elements of the second column.

$$\mathbf{H}_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & & \\ 0 & 0 & & \hat{\mathbf{H}}_2 \\ 0 & 0 & & \end{bmatrix} \tag{2.24}$$

This process is repeated n-2 times in order to obtain the tridiagonal matrix.

The flowcharts in Figures 2.1 and 2.2 illustrate and summarize the tridiagonalization process.

2.2 QR factorization

The basic idea behind the QR algorithm [14-15] states that any real matrix **A** can be decomposed in the form

$$\mathbf{A} = \mathbf{QR} \tag{2.25}$$

where **Q** is an orthogonal matrix and **R** is an upper triangular matrix.

As for the tridiagonalization process, it is possible to use Householder transformations to eliminate the elements under the main diagonal.

Then

$$\mathbf{H}_{n-1}\mathbf{H}_{n-2}\cdots\mathbf{H}_2\mathbf{H}_1\mathbf{A} = \mathbf{R} \tag{2.26}$$

Using the fact that the Householder matrices are symmetric, it follows that

$$\mathbf{A} = \mathbf{H}_1\mathbf{H}_2\cdots\mathbf{H}_{n-1}\mathbf{R} \tag{2.27}$$

From (2.27), we get

$$\mathbf{Q} = \mathbf{H}_1\mathbf{H}_2\cdots\mathbf{H}_{n-1} \tag{2.28}$$

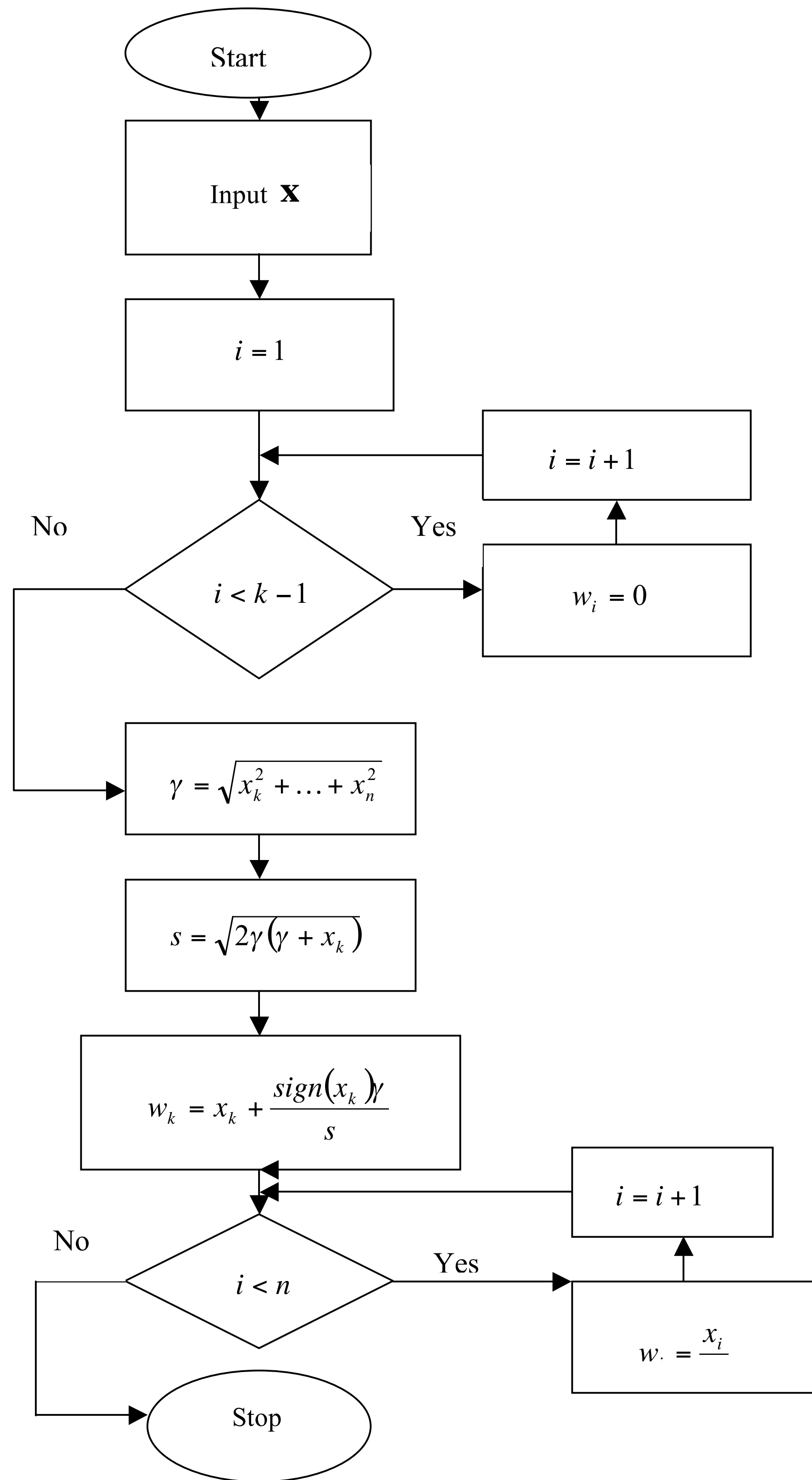


Figure 2.1: Flowchart of function House(**x,**k,n,**w**) which computes the householder vector **w** used to eliminate the lowest k components of a vector **x** of length n

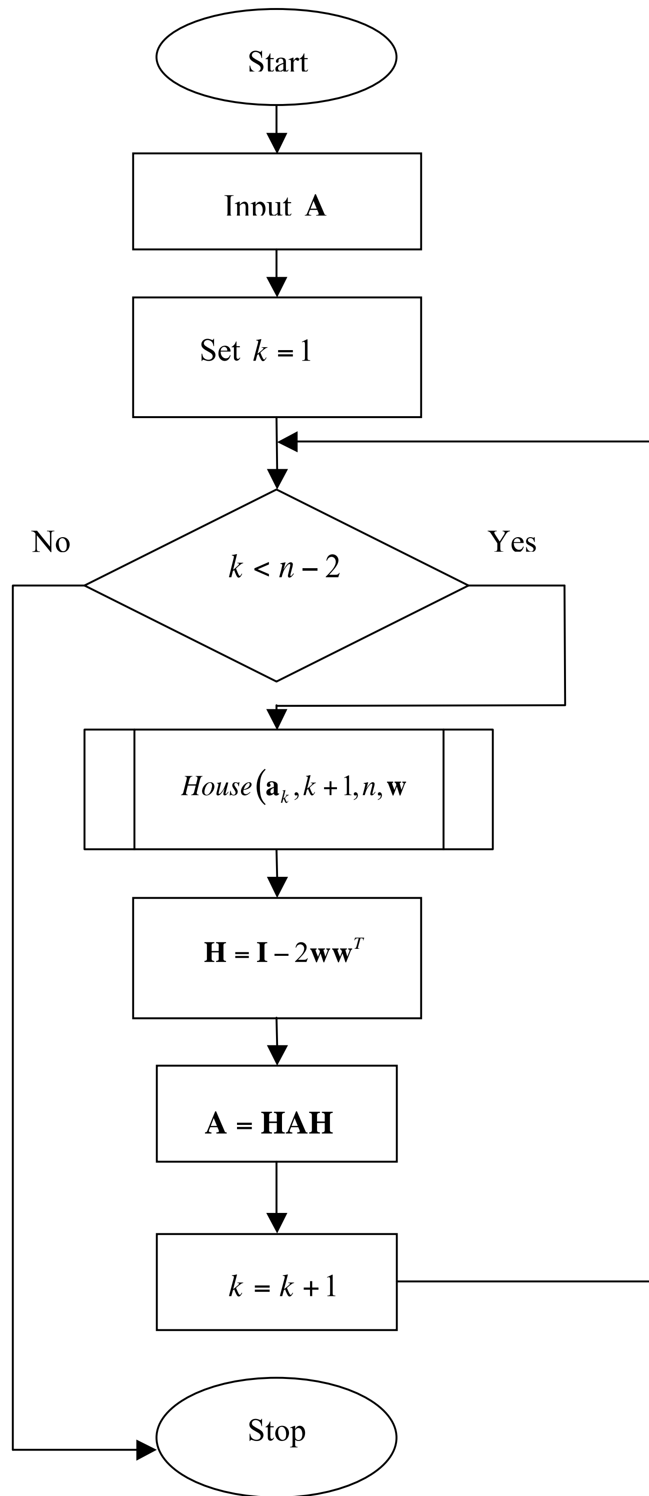


Figure 2.2: Flowchart of function Sym(**A**) which transforms a symmetric matrix **A** into tridiagonal form

The QR method can be used to find the eigenvalues of a tridiagonal matrix A using the following procedure:

$$\mathbf{A}_k = \mathbf{Q}_k \mathbf{R}_k \tag{2.29}$$

$$\mathbf{A}_{k+1} = \mathbf{R}_k \mathbf{Q}_k \tag{2.30}$$

Since $\mathbf{Q}_k$ is orthogonal, (2.30) can be rewritten as

$$\mathbf{R}_k = [\mathbf{Q}_k]^T \mathbf{A}_k \tag{2.31}$$

and

$$\mathbf{A}_{k+1} = [\mathbf{Q}_k]^T \mathbf{A}_k \mathbf{Q}_k \tag{2.32}$$

The matrix $\mathbf{A}$ will converge to a diagonal matrix that contains the eigenvalues of the tridiagonal matrix because of the properties of similar matrices. After m iterations, the product of the orthogonal transformations $\mathbf{Q} = \mathbf{Q}_1 \mathbf{Q}_2 \cdots \mathbf{Q}_m$ contains the eigenvectors of the tridiagonal matrix. The eigenvectors of the original matrix can be computed using the following orthogonal transformations:

$$\mathbf{X} = \mathbf{H}_1 \mathbf{H}_2 \cdots \mathbf{H}_{n-2} \mathbf{Q} \tag{2.33}$$

where $\mathbf{X}$ is the matrix that contains the eigenvalues of the original symmetric matrix, $\mathbf{Q}$ is the matrix that contains the eigenvalues of the tridiagonal matrix, and $\mathbf{H}_1, \mathbf{H}_2, \cdots, \mathbf{H}_{n-2}$ are the Householder matrices computed in the tridiagonal process. The flowcharts in Figures 2.3 and 2.4 show the implementation of the QR algorithm.

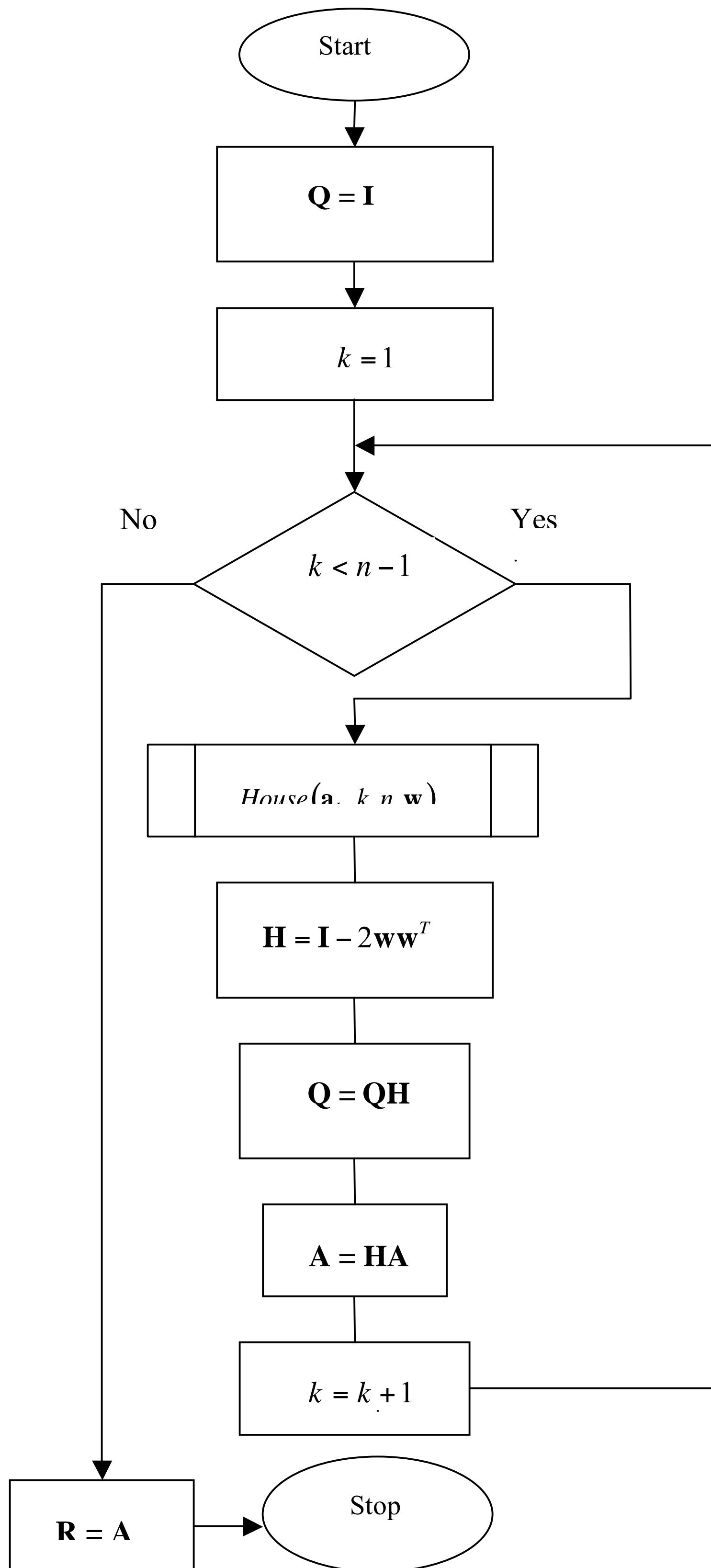


Figure 2.3: Flowchart of function QR (**A,Q,R**) which computes the QR factorization of a matrix **A**

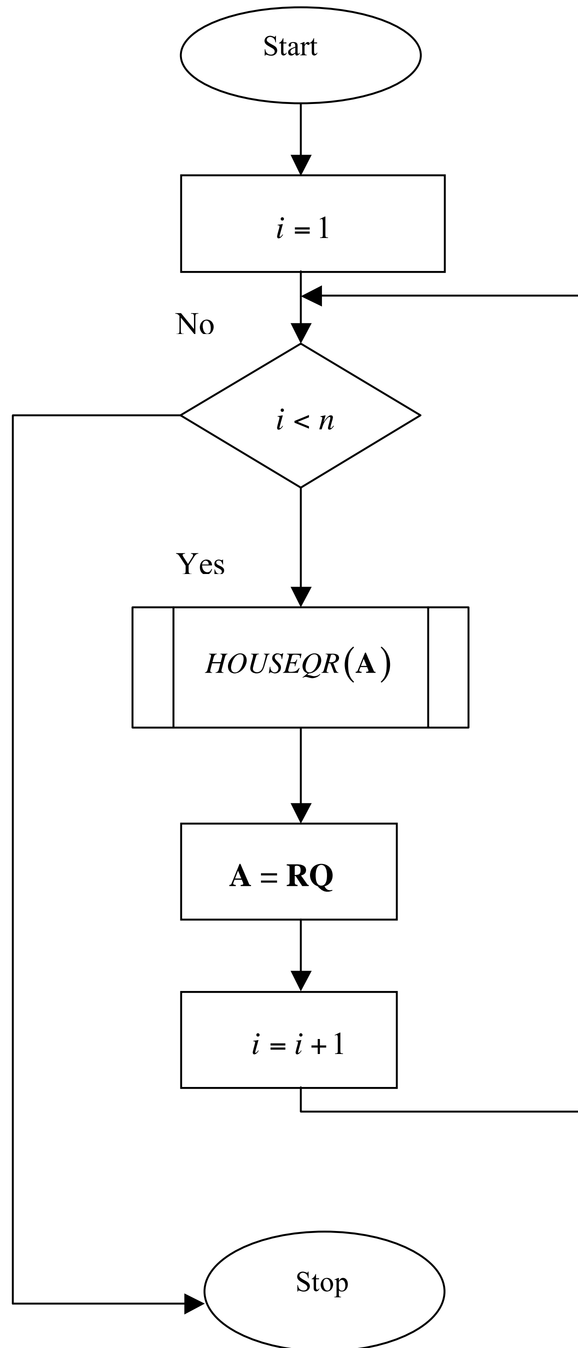


Figure 2.4: Flowchart of function QRShift (**A**,n) which computes the eigenvalues of tridiagonal matrix **A** after n iterations

Chapter 3

DSP Implementation of CSS algorithm

The CSS algorithm is based on matrix computations and orthogonal transformations, which are computationally intensive. To implement this algorithm in real-time, hardware capable of executing millions of operations per second is required. Some of the architectures available for that purpose are DSPs, FPGAs and ASICs. A general purpose DSP was selected as an appropriate platform for implementation because of its ease of programming. Also, the DSP is the best suited for matrix and floating points computations. This chapter provides some information about the DSP used and gives the simulations details. Experimental results are also presented.

3.1 Digital Signal Processor

3.1.1 Microcontroller

The DSP used in our implementation is a DIOPSIS™ 740 by Atmel. DIOPSIS™ 740 (D740) is a high performance dual-core processing platform for audio, communication and beam-forming applications, integrating a mAgic DSP and an ARM7TDMI RISC MCU[16]. The D740 is optimally suited for floating point applications complex domain computations.

The ARM7TDMI embedded microcontroller core is a member of the Advanced RISC Machines (ARM) family of general-purpose 32-bit microprocessors. The ARM architecture is based on Reduced Instruction Set Computer (RISC) principles. It is equipped with peripherals such as serial interfaces (SPI, USART), timers, watchdog audio A/D and D/A converters. The ARM7 is equipped with 32 Kbytes of on-chip memory and runs at half the clock speed of mAgic.

The mAgic DSP is the VLIW numeric processor of the D740. It operates on IEEE 754 40-bit extended precision floating-point and 32-bit integer numeric format. mAgic is capable of delivering 1 Giga Floating-Point Operations Per Second (GFLOPS) at a clock rate of 100 MHz. The main components of the DSP subsystem are the core processor, the on-chip memories and the interfaces to and from the ARM subsystem. The operators block, the register file, the address generation unit and the program decoding and sequencing unit compose the core processor. Figure 3.1 shows the architecture of the mAgic DSP.

The mAgic DSP has four on-chip memory blocks: the program memory, the data memory, the data buffer, and the dual ported memory shared with the ARM processor. An external memory interface multiplexes the data accesses and the program accesses to and from the external memory. The program memory stores the Very Long Instruction Word (VLIW) program to be executed by mAgic. It is 8 KWords x 128-bit single port memory. The mAgic internal data memory is made of three memory pages that are

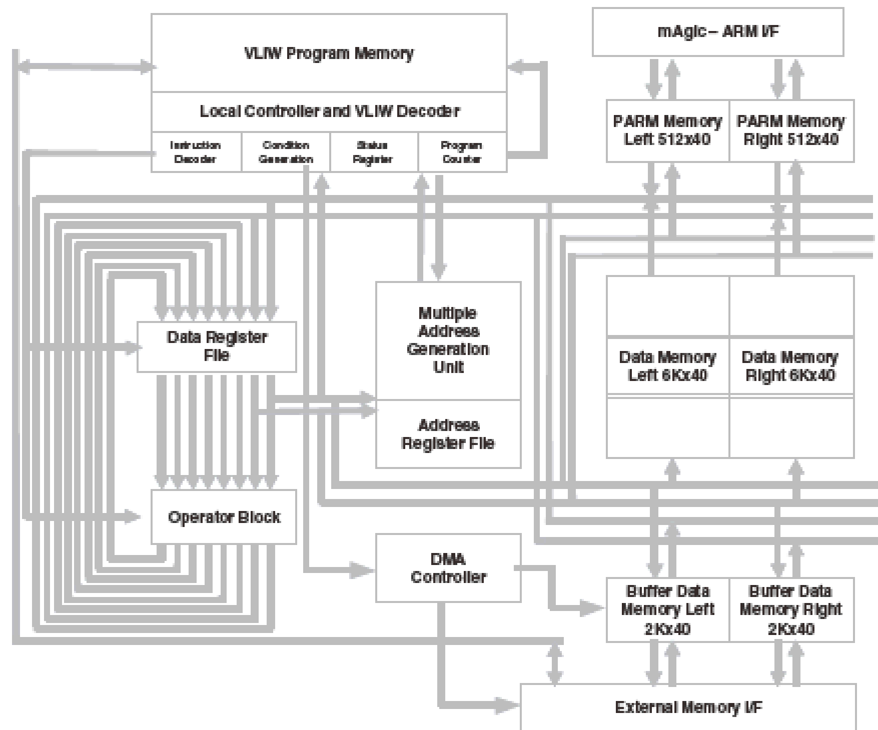


Figure 3.1 mAgic DSP Block Diagram (Courtesy of Atmel Corporation) [16]

partitioned into left and right memory banks. Each bank contains 2 KWords x 40-bit, thus a total of 12 KWords. Each data memory bank is a dual port memory that allows four simultaneous accesses, two read and two write. The buffer memory is a dual port memory, and contains 2 KWords x 40-bits for each bank. The buffer memory is a dual port memory. One port is connected to the core processor and the second port is connected to the external memory interface. The maximum external memory size of mAgic is 16 MWords for each bank (equivalent to 32 Mword or 160 Mbytes; 24-bit address bus). A Direct Memory Access (DMA) controller manages the data transfer between the external memory and the buffer memory. The last memory block in the address space of the mAgic DSP is the shared memory (PARM) between mAgic and the ARM processor. It is a dual port memory containing 512 Words by 40- bit for both the

left and the right bank (total 1K by 40-bit). This memory can be used to efficiently transfer data between the two processors.

3.1.2 Software

Multicore Application Development Environment (MADE) is an Integrated Development Environment (IDE) that can be used to develop D740 applications [16]. It includes the C compilers for both ARM and mAgic DSP based on GNU compiling tools named as GCC. It has a high-level mAgic DSP macro-assembler/optimizer, an editor tool with syntax highlighting. It also includes a unified debugging environment that can be interfaced with a cycle accurate simulator, or with a development board through the GNU debugger tool called GDB. The magic C compiler contains a DSP library composed of over 220 functions such as Fast Fourier Transform and IIR and FIR filter creation.

MADE can run exclusively in two modes:

1. Cycle Accurate Simulator mode, i.e. attached to the cycle accurate simulator.
2. Diopsis target mode, i.e. connected to a board through a serial link.

3.1.3 Evaluation board

The JTST board [16] is low-cost, stand-alone, general-purpose module that provides the appropriate resources in order to evaluate D740 DSP performances in a wide range of applications. The JTST board provides the following resources to D740:

- Memories: mAgic SSRAM, ARM FLASH and ARM SRAM

- Stereo Audio CODECs (4 in + 4 out)
- Serial I/O:
- 1 USB 2.0 Full (12 Mbps)
- 2 RS232/LVTTL asynchronous/synchronous serial I/O lines
- 2 SPI serial I/O lines
- Reset Logic (Power ON, Push Button, WDG)
- PLL - Clock Logic
- Configuration DIP SWITCH & Status 7-segment Display
- Voltage Regulators 5V/3.3V & 5V/1.8V
- Connectors (USART, SPI, USB, PIO, AUDIO, JTAG-ICE, EXT CLK, PSU)

Figure 3.2 shows the layout of the JTST board.

3.2 Simulation

In order to demonstrate the performance of the DOA algorithm for wideband signals, a linear array of sixteen equally spaced omni directional sensors was used. The spacing between sensors is $\frac{c}{2f_o}$, where c is the velocity of propagation and f_o is the central frequency. We will attempt to estimate DOA θ_1 and θ_2 of two source signals. The signals are stationary zero mean band pass white Gaussian processes with central frequency $f_o = 100\text{Hz}$ and bandwidth $B = 40\text{Hz}$. The array noise is also stationary zero mean band pass with the same pass band as the signal with a SNR of 10dB at each sensor.

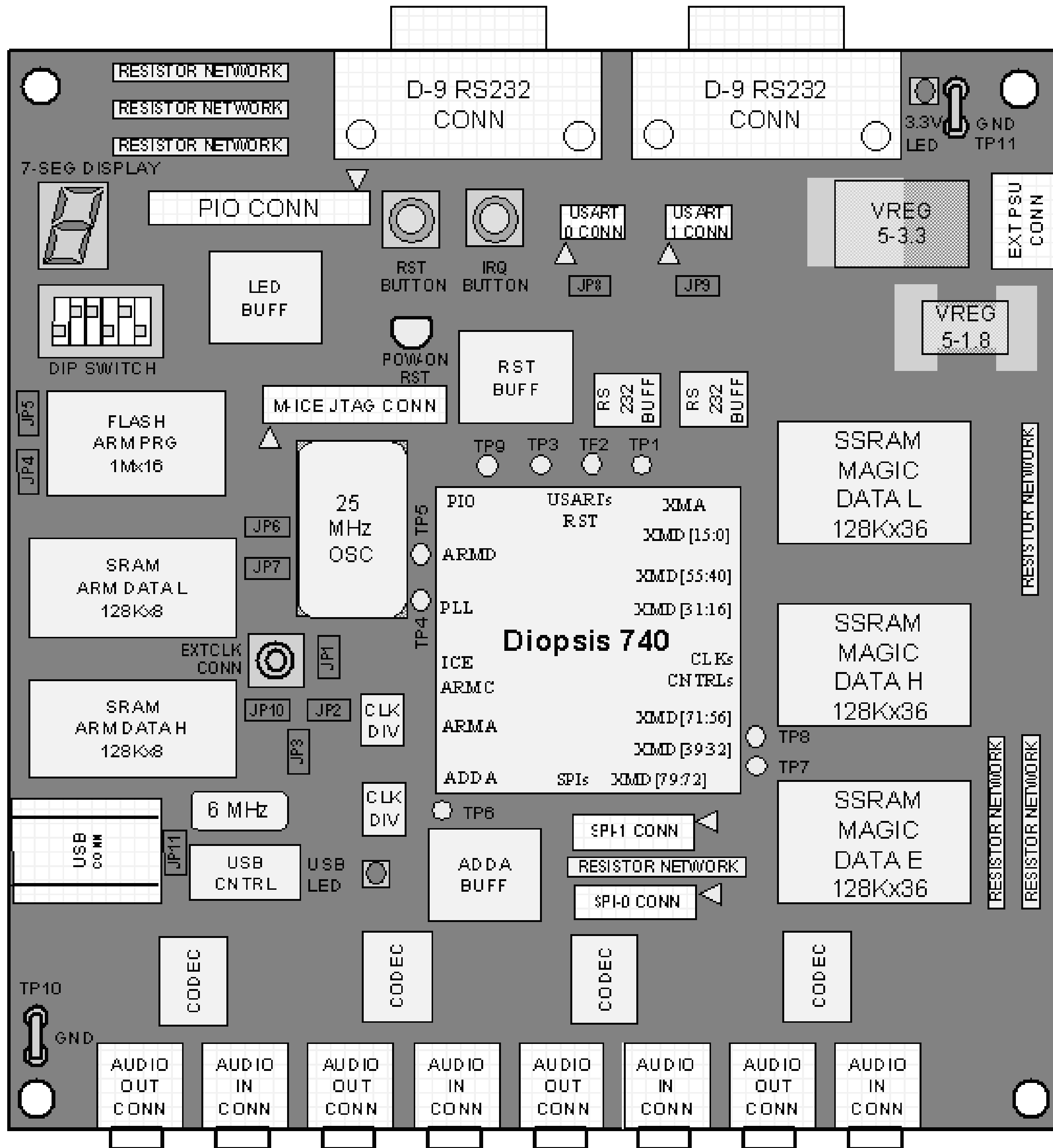


Figure 3.2 JTST Layout [16]

3.2.1 Sampling

As mentioned above, the sources signals and the noise are random processes with a bandwidth of 40 Hz. To avoid aliasing, the sampling frequency should be higher than the Nyquist frequency, which is twice the bandwidth. The sampling frequency is chosen

to be 300 Hz. The signal will be observed over a period of T_o seconds and T_o will be divided into $k = 64$ segments. On each of those segments, the array output along with corresponding noise will be decomposed into narrowband components using a fast Fourier transform. We will choose a number of 64 sample points per segment in order to use a 64 point FFT. The total number of samples taken by each sensor will be 4096. Simulation data is similar to the method described by Wang & Kaveh [10].

3.2.2 Data generation

Both the signal and noise are white Gaussian processes. They can then be generated by passing a set of random numbers through a band pass filter. Random numbers can be generated using a linear congruence algorithm, which is explained in [17]. The band pass filter is generated by using the function IIR1 in the DSP library and specifying the band pass frequencies. The signal can be represented as

$$\mathbf{s}(t) = [s_1(t_o)\ldots s_1(t_{63})\ldots s_n(t_o)\ldots s_n(t_{63})\ldots s_{64}(t_o)\ldots s_{64}(t_{63})]^T \tag{3.1}$$

where each sample is a random number passed through a band pass filter.

The array output can then be generated by using the wideband signal model in (1.35). The time domain samples will be transformed into frequency domain by applying a 64 point FFT to each of the 64 segments. The contents of both the time sample vectors and the frequency sample vectors after FFT are shown below.

$$\begin{bmatrix} s_n(t_o) \\ s_n(t_1) \\ \vdots \\ s_n(t_{63}) \end{bmatrix} \underset{--->}{FFT} \begin{bmatrix} s_n(w_o) \\ s_n(w_1) \\ \vdots \\ s_n(w_{63}) \end{bmatrix} \tag{3.2}$$

where $s_n(t_{k-1})$ is the kth sample of the nth segment and $s_n(w_n)$ represents the component of the nth segment at frequency w_n.

The value of the sensor impulse response a is 1 for omni directional vectors. The signal received at each array is the frequency domain data scaled by a factor of $e^{jw_n\tau}$, which is the phase shift at that particular sensor, and then increased by a value $n(w_n)$, which is the noise value. The following pseudo-code shows the generation for two signals arriving at an array composed of sixteen sensors:

```
For i = 1 to 16
        For i = 1 to 64
                For i = 1 to 64
                    Generate random signal values s1 and s2
                    Filter values of s1 and s2
                    Generate random noise values n1 and filter values
                    Compute time delay for each frequency
                    Apply FFT to s1 and s2
                    Multiply s1 and s2 by time delay
                    Add n1 to s1 and n1 to s2
                    Add s1 and s2 to obtain x
                    Send resulting matrix to external memory
                End
        End
End
```

The output of the sensor array is a 4096x16 matrix expressed as

$$\mathbf{X} = \begin{bmatrix} x_1^1(0) & \cdots & x_1^n(0) & \cdots & x_1^{16}(0) \\ \vdots & & \vdots & & \vdots \\ x_1^1(63) & \cdots & x_1^n(63) & \cdots & x_1^{16}(63) \\ \vdots & & \vdots & & \vdots \\ \vdots & & \vdots & & \vdots \\ \vdots & & \vdots & & \vdots \\ x_{64}^1(0) & & x_{64}^n(0) & & x_{64}^{16}(0) \\ \vdots & & \vdots & & \vdots \\ x_{64}^1(63) & \cdots & x_{64}^n(63) & \cdots & x_{64}^{16}(63) \end{bmatrix} \tag{3.3}$$

where $x_n^m(k)$ is the m_{th} output sample of frequency w_k for the n_{th} segment.

Due to the memory limitation of the DSP, the output data will be saved in the external memory as sixty four 64x16 matrices, where each matrix represents the output for each of the 64 segments. This is done using a memory transfer from local to external memory.

3.2.3 Covariance matrix computation

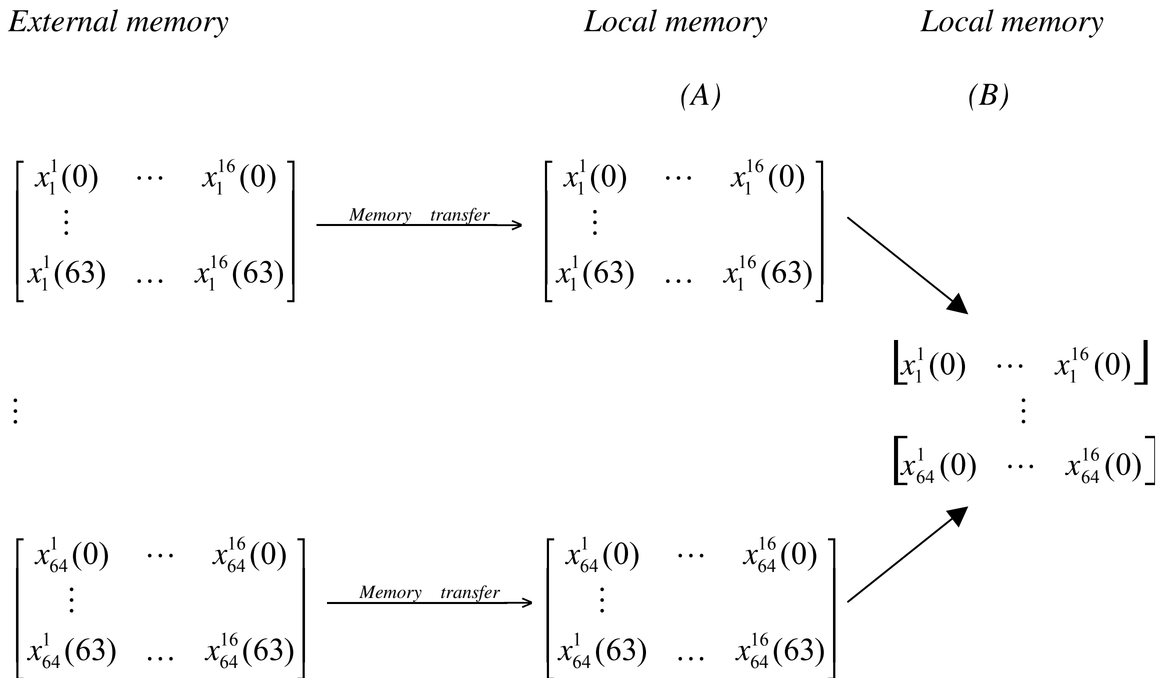
The covariance matrix for each frequency component can be computed using Equation (1.40). The main issue resides in forming the matrix **X** containing all samples of one frequency components. The contents of the external memory are the following:

$$\begin{bmatrix} x_1^1(0) & \cdots & x_1^{16}(0) \\ \vdots & & \\ x_1^1(63) & \ldots & x_1^{16}(63) \end{bmatrix}$$

$$\vdots \qquad\qquad \vdots$$

$$\begin{bmatrix} x_{64}^1(0) & \cdots & x_{64}^{16}(0) \\ \vdots & & \\ x_{64}^1(63) & \ldots & x_{64}^{16}(63) \end{bmatrix}$$

As we can see, the samples related to each frequency are spread across the 64 matrices and need to be put in the same matrix. This can be done by using the memory transfer function to transfer each of the data matrices and then store the row corresponding to the frequency needed. The following diagram shows that computation process for the frequency component w_o.



Using the periodicity and symmetric properties of the FFT, it is possible to have all the information needed by selecting frequencies w_o to w_{32}. The process above is repeated for each of the 33 covariance matrices. The following pseudo-code shows the covariance matrices computation process:

```
For i = 0 to 32
    For j = 0 to 63
        Transfer j^th data matrix to local memory matrix A
        Set j^th row of matrix B equal to i^th of matrix A
        Compute covariance matrix using matrix B
        Send covariance matrix to external memory
    End
End
```

After execution of this code, all the covariance matrices are stored in external memory.

3.2.4 Eigendecomposition of covariance matrix

The eigenvalues and eigenvectors of a symmetric matrix can be computed using the Householder and *QR* algorithms, as indicated in the previous chapter. However, these algorithms cannot be directly applied to the covariance matrix as it is a Hermitian matrix, i.e. the complex analog of a symmetric matrix. Solving this problem requires the conversion of the Hermitian matrix into a real symmetric matrix. It should be noted that Hermitian matrices have real eigenvalues.

Let $\mathbf{C} = \mathbf{A} + i\mathbf{B}$ where $\mathbf{C}$ is an n x n Hermitian matrix, $\mathbf{A}$ and $\mathbf{B}$ are n x n real matrices. The eigenvalue problem for a complex matrix is

$$(\mathbf{A} + i\mathbf{B})(\mathbf{u} + i\mathbf{v}) = \lambda(\mathbf{u} + i\mathbf{v}) \tag{3.4}$$

where λ is the real eigenvalue of matrix **C**, and $\mathbf{u} + i\mathbf{v}$ is the complex eigenvector of matrix **C**. This *n* x *n* complex eigenvalue problem is the equivalent to the 2*n* x 2*n* real eigenvalue problem.

$$\begin{bmatrix} \mathbf{A} & -\mathbf{B} \\ \mathbf{B} & \mathbf{A} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix} = \lambda \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix} \tag{3.5}$$

The 2*n* x 2*n* matrix is only symmetric if $\mathbf{C} = \mathbf{A} + i\mathbf{B}$ is Hermitian. Solving for the eigenvalues and eigenvectors of the 2*n* x 2*n* matrix using the Householder and *QR* methods will yield two 2*n* x 2*n* matrices containing its eigenvalues and eigenvectors. There are 2*n* eigenvalues in the form $\lambda_1, \lambda_1, \ldots, \lambda_n, \lambda_n$ where $\lambda_1, \ldots, \lambda_n$ are the eigenvalues of the original matrix **C**. The eigenvectors matrix contains the eigenvectors in the form

$$\begin{bmatrix} \mathbf{u_1} & \mathbf{-v_1} & \cdots & \cdots & \mathbf{u_n} & \mathbf{-v_n} \\ \mathbf{v_1} & \mathbf{u_1} & \cdots & \cdots & \mathbf{v_n} & \mathbf{u_n} \end{bmatrix} \tag{3.6}$$

Corresponding to an eigenvalue λ_n, the vectors $\begin{bmatrix} \mathbf{u}_n \\ \mathbf{v}_n \end{bmatrix}$ and $\begin{bmatrix} -\mathbf{v}_n \\ \mathbf{u}_n \end{bmatrix}$ are both eigenvectors. Thus, the eigenvectors pair for the Hermitian matrix can be $(\mathbf{u}_n + i\mathbf{v}_n)$ or $(-\mathbf{v}_n + i\mathbf{u}_n)$. The following pseudo-code shows the computation process for the eigenvalues of eigenvectors of the covariance matrices:

for $i = 0$ to 32:

> Transfer i^{th} covariance matrix to local memory matrix A
>
> Convert A into real symmetrical matrix B
>
> Find eigenvalues and eigenvectors of matrix B

```
        Form complex eigenvectors of matrix A
        Sort eigenvalues and corresponding eigenvectors from largest to smallest
        Transfer eigenvalues and eigenvectors to external memory
End
```

3.2.5 Power spectrum

The power spectrum is computed using Equation (1.29). As explained in Section 1.1.2, the column vectors forming the noise matrix are the eigenvectors associated with the *M*-*D* lowest eigenvalues, where *M* is the number of sensors and *D* is the number of sources. The power spectrum needs to be computed to obtain an initial estimate for the DOA. The number of sources has been determined using the MDL algorithm. The following pseudo-code shows the computation process for the power spectrum:

```
Input value d for the number of sources
Create matrix v of size 16 x d
for i = 0 to 90, form v using m-d eigenvectors
    Form direction vector A using angle i
    Compute power spectrum
    Store value of power spectrum in array Pm
End
Find d peaks in array Pm
```

The value of D used is 1 for the initial DOA estimates.

3.2.6 Coherent signal subspace processing

As explained in Section 1.2.2, the matrix pencil is formed using the covariance matrices computed in Section 3.2.3. The covariance matrices must be transferred back to local memory for the computation of matrix $\mathbf{R}$. The process is included in the following pseudo-code.

Set $R = 0$

Set $R_n = 0$

For i = 0 to 32

Transfer covariance matrix $Cov(w_i)$ to local memory

Compute transformation matrix T using central frequency and initial DOA estimate

Compute $T \cdot Cov(w_i) \cdot T^H$

Add result to R

End

The eigenvalues and eigenvectors of the matrix pencil will be computed using the Householder and *QR* algorithms. Thereafter, they are used to determine the final DOA estimates.

3.3 Simulation results

The Figures 3.3 to 3.6 show the results of simulations for estimation of DOA of two sources located in the interval [0, 90]. For each case, the initial and final DOA estimates are shown.

Case 1: $\theta_1 = 50^{\circ}$ and $\theta_2 = 20^{\circ}$

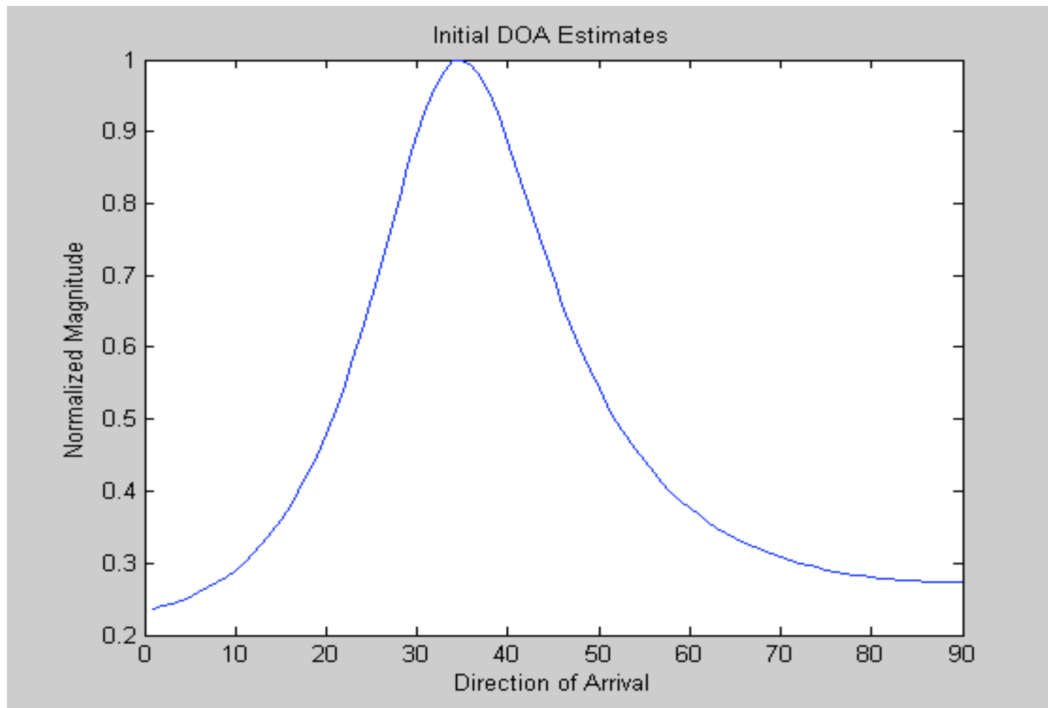


Figure 3.3: Plot of power spectrum for initial DOA estimates of angles $\theta_1 = 50^{\circ}$ and $\theta_2 = 20^{\circ}$

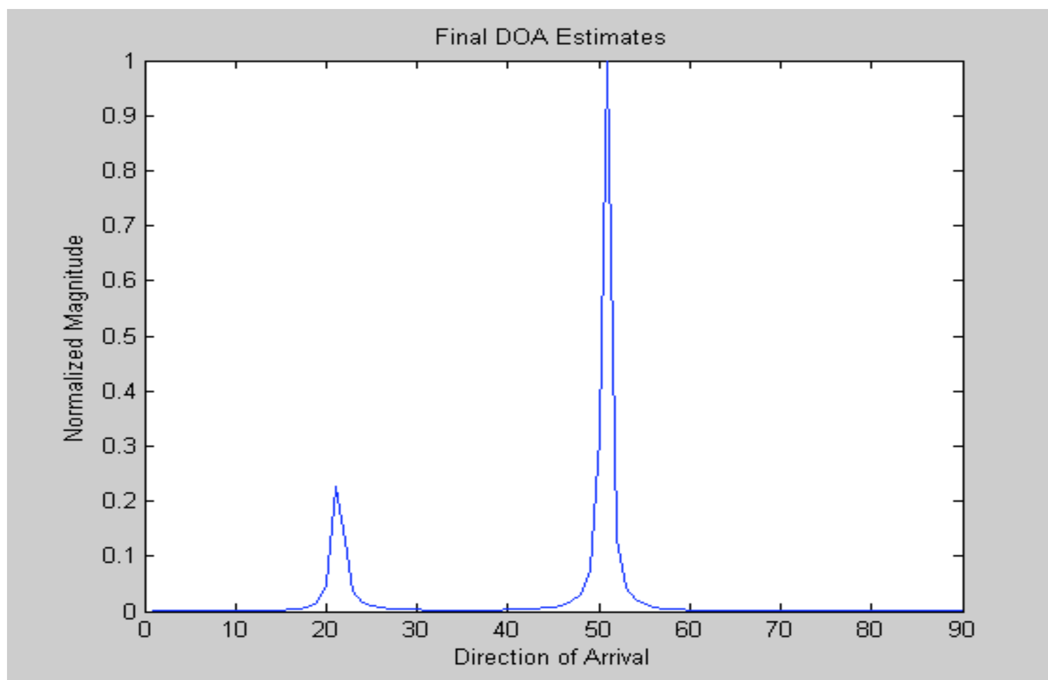


Figure 3.4: Plot of power spectrum for final DOA estimates of angles $\theta_1 = 50^{\circ}$ and $\theta_2 = 20^{\circ}$

Case 2: $\theta_1 = 30^{\circ}$ and $\theta_2 = 40^{\circ}$

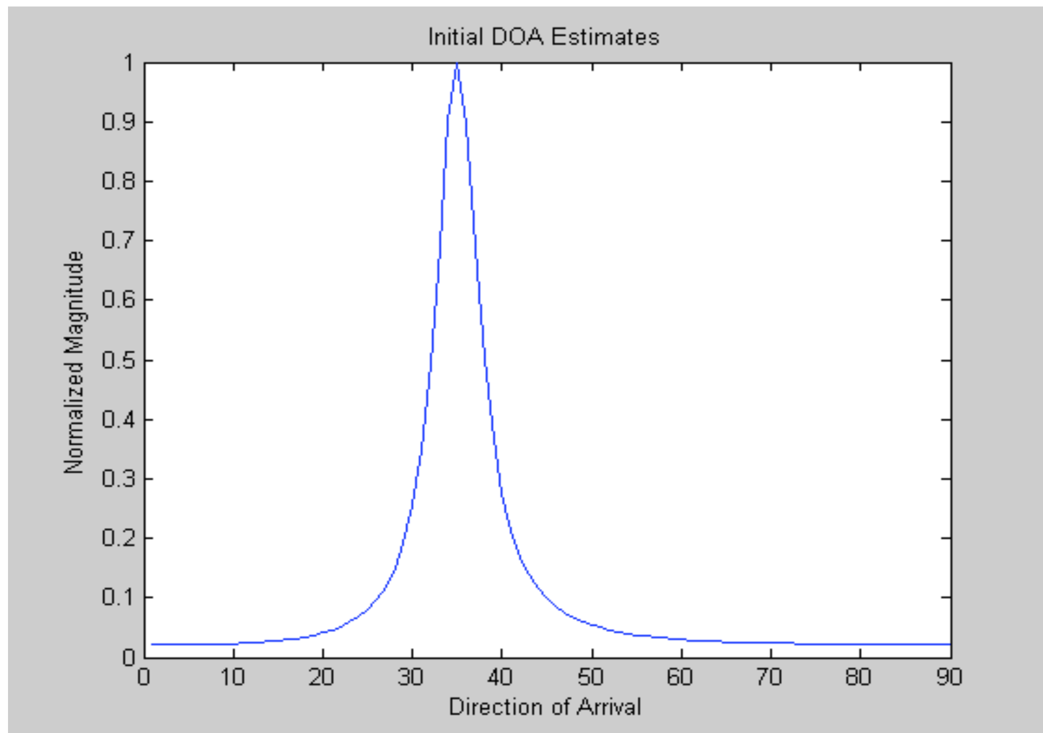


Figure 3.5: Plot of power spectrum for initial DOA estimates of angles $\theta_1 = 30^{\circ}$ and $\theta_2 = 40^{\circ}$

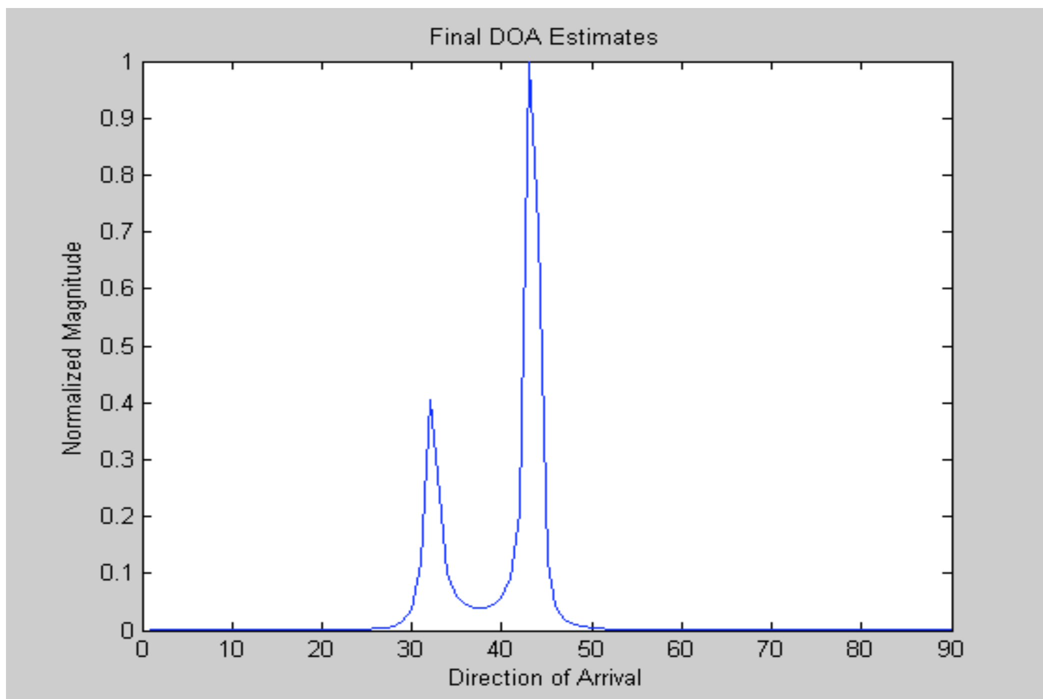


Figure 3.6: Plot of power spectrum for final DOA estimates of angles $\theta_1 = 30^{\circ}$ and $\theta_2 = 40^{\circ}$

The results indicate that DOA detection and estimation was successful. However, a high number of iterations (40) were required by the QR algorithm for a good approximation of the eigenvalues and eigenvectors.

Chapter 4

Parallel Architecture for Coherent Signal Subspace Algorithm

The coherent signal subspace method for wideband DOA algorithm uses the MUSIC algorithm which is a high resolution method capable of yielding very accurate DOA estimates. However, this algorithm is computationally very intensive. The overall computational time can be significantly decreased through the use of parallel systems. The following approach is used in the implementation process:

- The areas that can be improved are identified.
- The algorithms used are converted into computationally efficient modules.
- The optimized modules are divided into parallel processes.

In this chapter, the coherent signal subspace algorithm is implemented on parallel processors to improve its performance. Sections of the algorithm that do not lend themselves to parallelization will be implemented on a single DSP.

4.1 Covariance matrix computation

In Section 3.2.3, a method for computing the covariance matrices using a single DSP was presented. This method can be improved upon by using one DSP to compute

each of the 33 covariance matrices. All 33 DSPs share the same external memory. Figure 4.1 shows its parallel architecture.

The computation process for the covariance is described below.

After the data generation step, the contents of the external memory are the following:

$$\begin{bmatrix} x_1^1(0) & \cdots & x_1^{16}(0) \\ \vdots & & \\ x_1^1(63) & \ldots & x_1^{16}(63) \end{bmatrix}$$
$$\begin{matrix} \vdots & & \vdots \\ \vdots & & \vdots \\ \vdots & & \vdots \end{matrix}$$
$$\begin{bmatrix} x_{64}^1(0) & \cdots & x_{64}^{16}(0) \\ \vdots & & \\ x_{64}^1(63) & \ldots & x_{64}^{16}(63) \end{bmatrix}$$

Processors can initiate a DMA transfer from the external memory to their local memory. The address used during the transfer is the address of the row containing the samples at a particular frequency and the number of elements transferred is the number of elements contained in the row. The DMA transfer for the first matrix is shown below:

$$\begin{matrix} \textit{External memory} & & \textit{Local Memory} & \\ \begin{bmatrix} x_1^1(0) & \cdots & x_1^{16}(0) \\ \vdots & & \vdots \\ x_1^1(32) & \cdots & x_1^{16}(32) \end{bmatrix} & \begin{matrix} \rightarrow \\ \\ \rightarrow \end{matrix} & \begin{matrix} \begin{bmatrix} x_1^1(0) & \cdots & x_1^{16}(0) \end{bmatrix} \\ \\ \begin{bmatrix} x_1^1(32) & & x_1^{16}(32) \end{bmatrix} \end{matrix} & \begin{matrix} DSP\ 0 \\ \\ DSP\ 32 \end{matrix} \end{matrix}$$

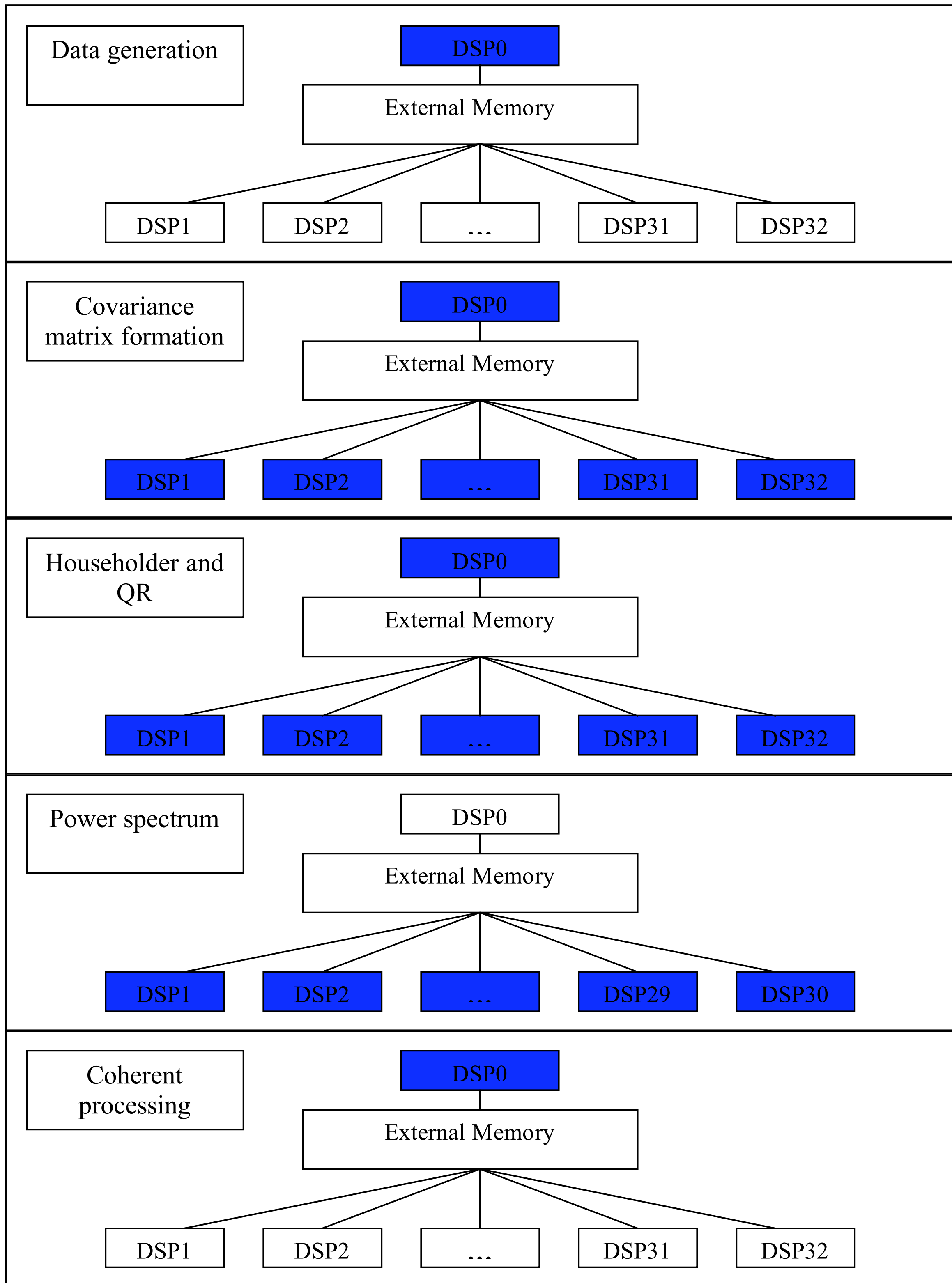


Figure 4.1: Parallel architecture for coherent signal subspace algorithm

After a DMA transfer is complete, another transfer will be initiated for a different segment matrix. This process is repeated for all 64 matrices (segments) in external memory. The pseudo-code executed by the ith processor is shown as:

For m = 0 to 63

Go to address i of mth matrix in external memory and transfer 16 elements to vector v in local memory

Transfer vector v to m^{th} row of vector A

End

Compute covariance matrix using matrix A

Send covariance matrix to external memory

The execution time will be the same for each processor. This execution time can be further reduced by computing only the lower triangular part of the covariance matrix. Since the covariance matrix is Hermitian, the information contained in the lower triangular part is sufficient to form the entire matrix. Using a single DSP approach, a total of 64 x 33 DMA transfers of 256 elements each were required. The parallelized process only requires 64 transfers of 16 elements each, which should result in increased performance.

4.2 Eigenvalues and Eigenvectors computation

The eigenvalues and eigenvectors computation process for the covariance matrix is based on two algorithms, the Householder and the *QR* methods, which require the use

of Householder matrices. Using certain properties of the Householder matrices, it is possible to parallelize the process. Let **A** be an *n* x *n* matrix and **H** be an *n* x *n* Householder matrix.

If **B** = **HA** then

$$\mathbf{B} = (\mathbf{I} - 2\mathbf{w}\mathbf{w}^T)\mathbf{A} \tag{4.1}$$

$$\mathbf{B} = \mathbf{A} - 2\mathbf{w}\mathbf{w}^T\mathbf{A} \tag{4.2}$$

If we let $\mathbf{b}_j$ be the jth column vector of **B** and $\mathbf{a}_j$ be the jth column vector of **A**, then

$$\mathbf{b}_j = (\mathbf{ÇÁ})_j \tag{4.3}$$

$$\mathbf{b}_j = \mathbf{H}\mathbf{a}_j \tag{4.4}$$

$$\mathbf{b}_j = (\mathbf{I} - 2\mathbf{w}\mathbf{w}^T)\mathbf{a}_j \tag{4.5}$$

$$\mathbf{b}_j = \mathbf{a}_j - 2\mathbf{w}\mathbf{w}^T\mathbf{a}_j \tag{4.6}$$

since $\mathbf{w}$ is a n x 1 vector, the product $\mathbf{w}^T\mathbf{a}_j$ will result in a real value that can be called α. Equation (4.6) can then be expressed as

$$\mathbf{b}_j = \mathbf{a}_j - 2\alpha\mathbf{w} \tag{4.7}$$

It is thus possible to compute each column of the product **HA** separately. For a matrix **A** of size *n* x *n, n* processors can be used in parallel to perform this computation. This observation will allow us to create parallel architectures for both the Householder and *QR* methods.

4.2.1 Householder Method

The Householder method consists of transforming a symmetric matrix into tridiagonal form using the following transformations

$$\mathbf{B} = \mathbf{H}_{n-2} \ldots \mathbf{H}_2 \mathbf{H}_1 \mathbf{A} \mathbf{H}_1 \mathbf{H}_2 \cdots \mathbf{H}_{n-2} \qquad (4.8)$$

The orthogonal transformation will be accumulated in a matrix **Q** in order to recover the eigenvectors of the original matrix **A**. Equation (4.8) can be rewritten as

$$\mathbf{B} = \mathbf{QAQ} \qquad (4.9)$$

It was shown that the product **HA** could be computed using n parallel processors if **H** was a Householder matrix. As a result, $\mathbf{H}_2\mathbf{H}_1\mathbf{A}$ and $\mathbf{H}_{n-2} \ldots \mathbf{H}_2\mathbf{H}_1\mathbf{A}$ can also be computed with n processors. If we let $\mathbf{C} = \mathbf{QA}$, then **C** can also be expressed as

$$\mathbf{C} = \mathbf{H}_{n-2} \ldots \mathbf{H}_2 \mathbf{H}_1 \mathbf{A} \qquad (4.10)$$

Matrix **Q** is expressed as

$$\mathbf{Q} = \mathbf{H}_{n-2} \ldots \mathbf{H}_2 \mathbf{H}_1 \mathbf{I} \qquad (4.11)$$

where **I** is the identity matrix. This implies that **C** and **Q** can be computed using the parallel approach explained above. The tridiagonal matrix will be the result of the product $\mathbf{CQ}$. Figure 4.1 shows a flowchart of the parallel Householder method.

After execution of the Householder algorithm, matrix **A** is the tridiagonal matrix and matrix **Q** contains the product of all the orthogonal transformations.

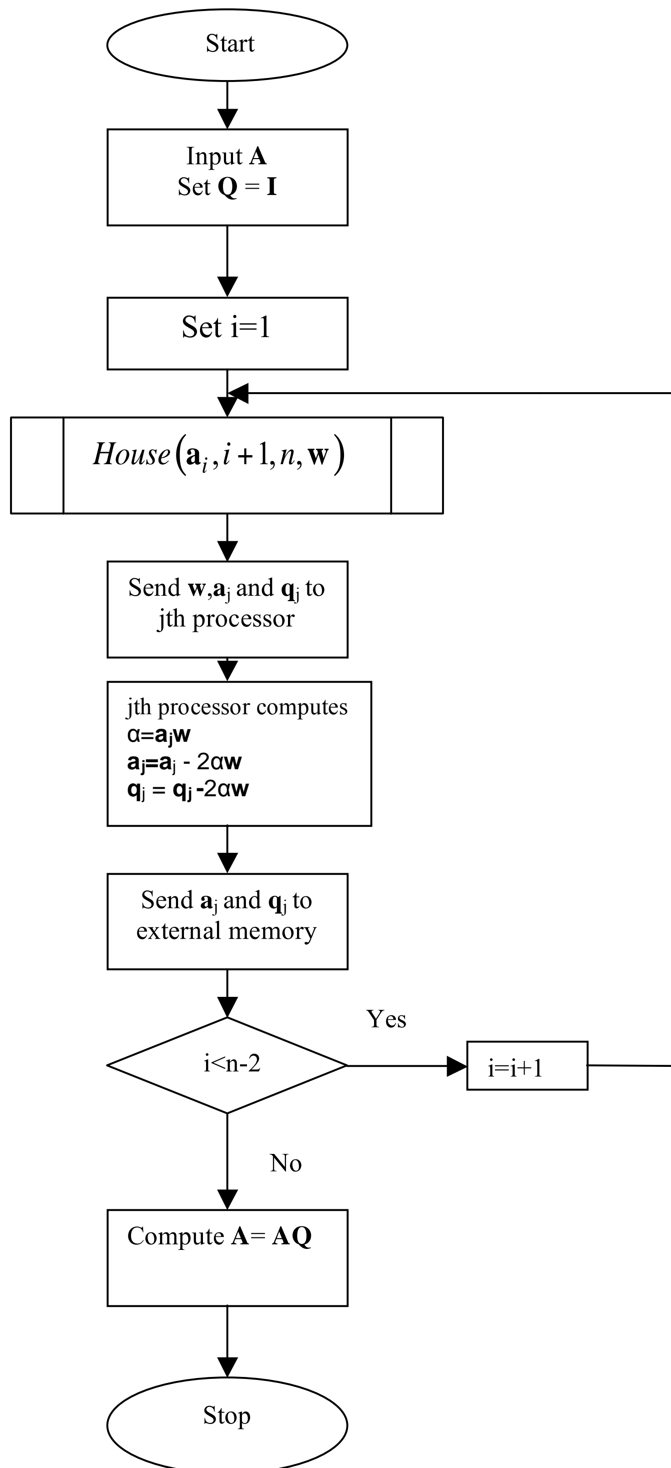


Figure 4.1: Flowchart of parallel Householder method

4.2.2 QR method

The QR method is based of the use of the following orthogonal transformations

$$\mathbf{R} = \mathbf{H}_{n-1} \dots \mathbf{H}_2 \mathbf{H}_1 \mathbf{A} \tag{4.12}$$

and

$$\mathbf{Q} = \mathbf{H}_{n-1} \dots \mathbf{H}_2 \mathbf{H}_1 \tag{4.13}$$

Using the scheme explained in the previous section, we can compute **R** and **Q** using n parallel processors. The next step involves the computation of the matrix **A** = **RQ** and the computation of a matrix **X** containing the product of the orthogonal transformations $\mathbf{Q}_1\mathbf{Q}_2 \cdots \mathbf{Q}_m$, where m is the number of iterations. These computations can be done using a single DSP. This process will start over until the number of iterations required for a good approximation has been reached. The Figure 4.2 shows a flowchart for the QR method.

After execution of the QR algorithm, the matrix **A** contains the eigenvalues of the original matrix along its diagonal and matrix **X** contains the eigenvectors of the tridiagonal matrix. The eigenvectors of the original matrix can be obtained be multiplying the matrix **Q** obtained in the Householder process and matrix **X**.

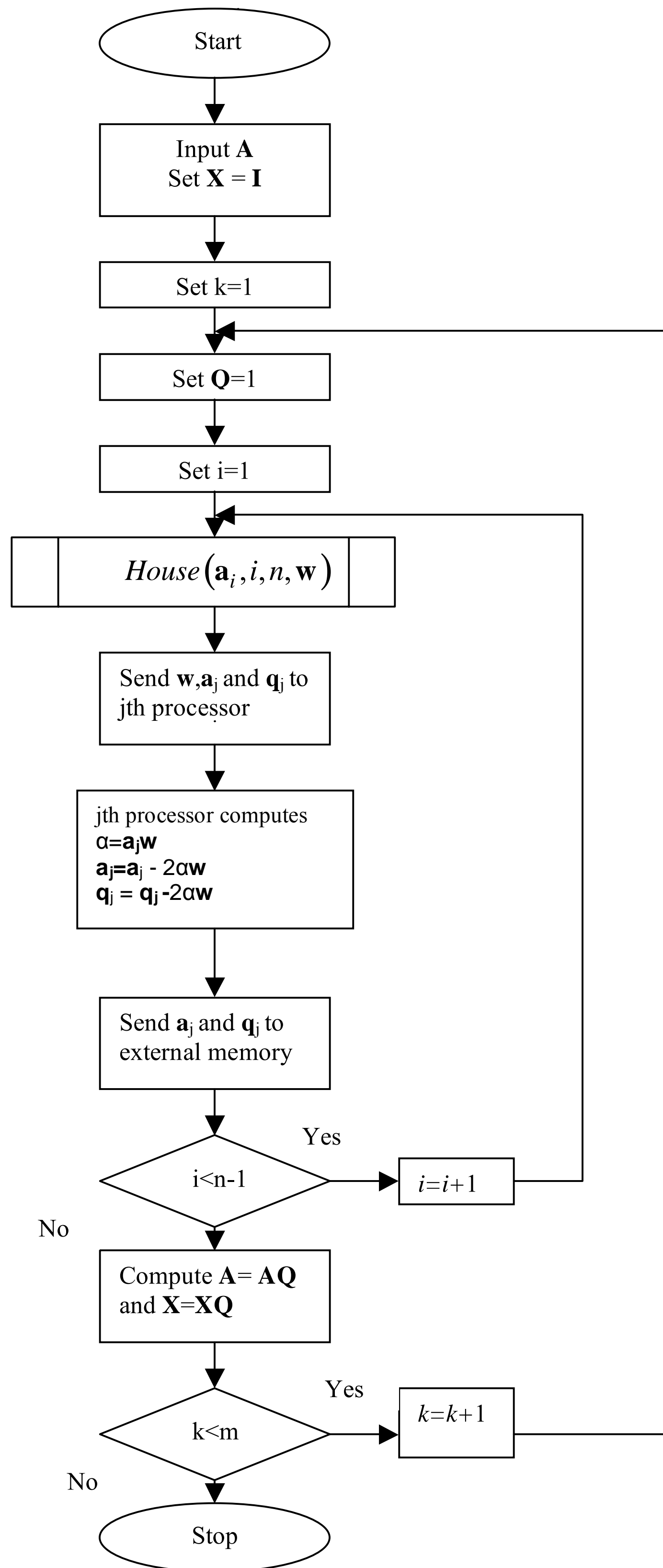


Figure 4.2: Flowchart of parallel QR method

4.2.3 Power Spectrum

The power spectrum needs to be computed for every angle between 0 and 90 degrees. Each spectrum value can be computed by sending the matrix containing the eigenvectors, the number of sources and the angle of arrival to a specific processor. The DSP can then compute the power spectrum and send the results back to external memory. By sending 3 angles values to 30 of the 33 available DSPs, it is possible to reduce the time needed to compute the power spectrum by a factor of 30.

4.3 Simulations

The previous simulation created for single DSP was used again for the case of the parallel DSP architecture. This was done to measure DOA and compare their performances. The parallel architecture was simulated using a single DSP by executing the parallel processes sequentially. However, the performance would be measured using the longest running process in that sequence in terms of number of clock cycles. The purpose of the simulations was to detect and estimate the DOA of two sources located at 20° and 60° using 40 iterations for the QR algorithm. Figure 4.3 shows the simulations results .It can be seen that both techniques yielded the same results.

The performance analysis showed that the most computationally intensive tasks were the QR algorithm, the Householder algorithm and the power spectrum computation. The following table shows the comparison between the performance of the single DSP and the performance of the parallel architecture.

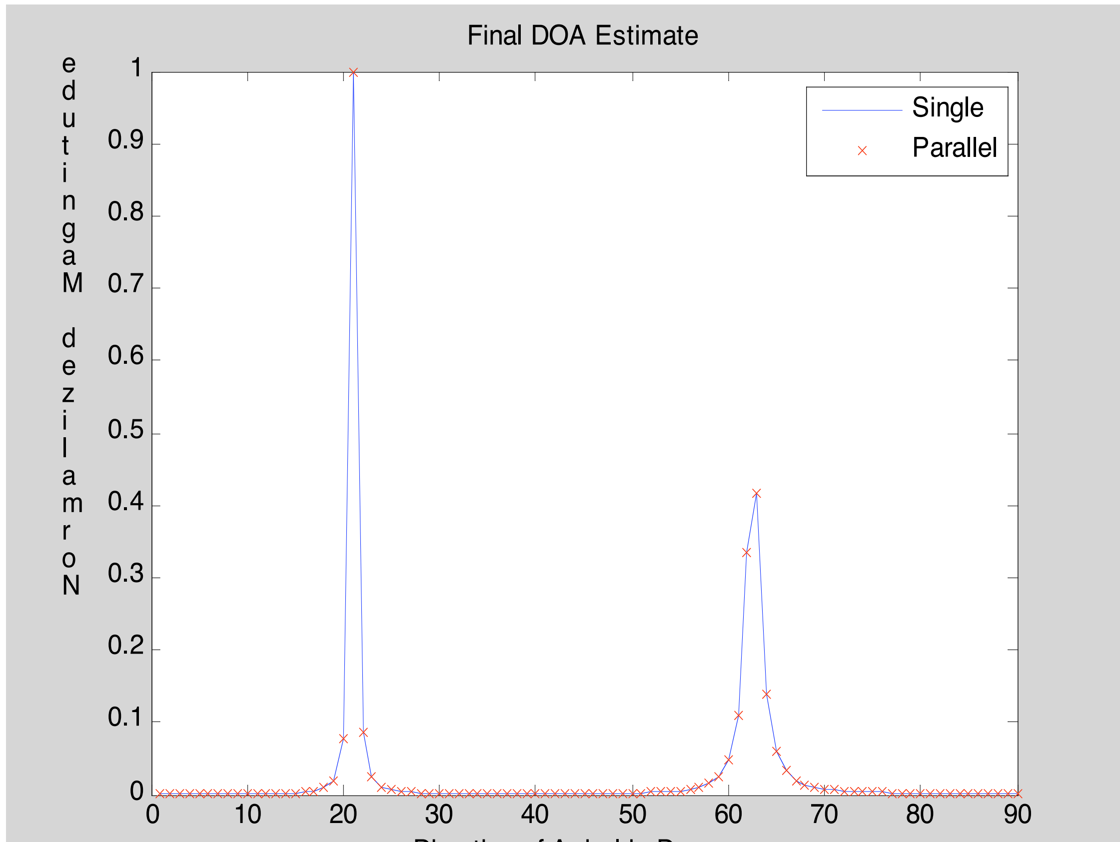


Figure 4.3: Plot of power spectrum for final DOA estimate of angles $\theta_1 = 20^\circ$ and $\theta_2 = 60^\circ$ using single and parallel DSP approach

Task	Single DSP		Parallel architecture	
	Cycles	Seconds	Cycles	Seconds
Covariance matrix	1400000	0.014	42000	0.00042
Householder	2600000	0.026	30000	0.0003
QR	100000000	1	2100000	0.021
Power spectrum	1400000	0.014	47000	0.00047
Total	210000000	2.1	5300000	0.053

Table 4.1 : Performance results for single DSP and parallel architecture

The total number of cycles for both the single DSP and the parallel architecture includes tasks that could not be parallelized and are not listed in the table. The total execution time for the parallel architecture is 0.05 seconds, compared to 2.1 seconds for the single DSP. This is due to the fact that the execution time for QR algorithm, which accounts for approximately 95% of the whole process, was significantly reduced using the parallel architecture.

Chapter 5

Conclusion

5.1 Summary

Direction of Arrival detection and estimation is a critical part of array processing. This thesis focused on developing architecture for the DOA detection and estimation for wideband sources. The coherent signal subspace algorithm was chosen for the implementation because of its high resolution; the platform selected was an Atmel Diopsis740 DSP. We first explained the difference between the narrowband and wideband MUSIC algorithms. We then showed that the coherent signal subspace algorithm was mainly based on orthogonal transformations and matrix computations, which had high computational requirements. A parallel architecture capable of improving the performance of the coherent signal subspace algorithm was proposed. The performance of the proposed architecture was measured and compared with the single DSP implementation. The results showed that the parallel architecture yielded the same results, while providing superior performance.

5.2 Future work

One of the limitations of the proposed architecture was the use of static matrices. This implied that the number of sensors and sources was known in advance. It also implied that the system would not be able to detect a greater number of sources without major modifications in the source code. Using dynamic matrices would allow the system to easily adapt to the number of sources to be detected.

A Digital Signal Processor was chosen for the implementation because of its flexibility and ease of programming. Other platforms such as ASICs and FPGAs could be used. The same algorithm could be implemented on those platforms so that the best system for DOA estimation can be determined. Some of the criteria for comparison could be performance, cost, size and energy consumption.

Chapter 6

References

1. J. Bohme, "Estimation of Source Parameters by Maximum Likelihood and Nonlinear Regression", Proc. IEEE International Conference on Acoustic, Speech and Signal Processing, pp.7.3.1-7.3.4, 1984.

2. B. D. Van Veen and K. M. Buckley, "Beamforming: a versatile approachto spatial filtering," IEEE ASSP Mag., vol. 5, pp. 4–24, Apr.1988

3. R.Schmidt, "A Signal Subspace Approach to Multiple Emitter Location and Spectral Estimation", PhD thesis, Sanford University.

4. B. Porat and B. Friedlander, "Estimation of spatial and spectral parameters of multiple sources." IEEE Transactions on Information Theory, vol. IT-29, pp. 412-425, May 1983.

5. A. Nehorai, G. Su, and M. Morf, "Estimation of time differences of arrival by pole decomposition," IEEE Transactions on Acoustic, Speech and Signal Processing, vol. ASSP-31, pp. 1478-1491, December 1983.

6. R. Schmidt, "Multiple emitter location and signal parameter estimation,"in Proc. RADC Spectrum Estimation Workshop, ct. 1979.

7. M. Coker and E. Ferrara, "A new method for multiple source location,"in Proc. IEEE International Conference on Acoustic, Speech and Signal Processing,Paris, France, Apr. 1982.

8. G. Bienvenu, "Eigensystem properties of the sampled space correlation matrix," in.Proc. IEEE International Conference on Acoustic, Speech and Signal Processing, April. 1983.

9. G. Su and M. Morf, “Signal subspace approach for multiple wideband emitter location,” IEEE Transactions on Acoustic, Speech and Signal Processing, vol. ASSP-31, pp. 1502-1522, December 1983.

10. H. Wang, and M. Kaveh, “Coherent Signal-Subspace Processing for Multiple Wideband Sources” in Proc. IEEE International Conference on Acoustic, Speech and Signal Processing, vol. 33, No. 4, pp. 823-831, August 1985

11. M. Wax, T. San and T. Kailath, “Spatio-Temporal spectral analysis by eigenstructure methods”, IEEE International Conference on Acoustic, Speech and Signal Processing, vol 32, No.4 , pp 817-827.

12. J. Liberti and T. Rappaport, “Smart antennas for wireless communications”, Prentice Hall, 1999

13. M. Wax and T. Kailath, “Determining the number of signals by information theoretical criteria,” in Proc. ASSP Spectrum Estimation Workshop 11, Tampa, Florida, pp. 192-196, 1983

14. N.S. Asaithambi, “Numerical Analysis”,Saunders College Publishing, 1995.

15. W. Vetterling and B. Flannery, “Numerical Recipes in C++”, Cambridge University Press, 2002

16. “Diopsis 740 Dual Core user manual”, Atmel, http://www.atmel.com

17. D. Grover and J. Deller, “Digital Signal Processing and the Microcontroller”, Prentice Hall, 1999.

Chapter 7

Appendix

System Software

The source files are provided on a compact disk

All files are listed below along with a brief description:

- single.c - This file contains the single DSP source code.
- parallel.c - This file contains the parallel architecture source code.
- test.c- This file contains the source code needed by the ARM processor

single.c - This file contains the single DSP source code.

```
#include "math.h"
#include "magic.h"
#include "DSPlib.h"
#define Pi 3.1415
#define nss 2
#define velocity 300000000
#define omega 1885
#define dist 500000
#define ncc 16
float Pm[91]={23};
float eigenvalues[16]={333};
float eigvects[32][16]={14};
float cova[32][32]={55};
__complex__ float Part[15] ={5};
__complex__ float eigvects2[16][16] ={5};
__complex__ float En[16][15] ={15};
__complex__ float nos[64][16] BUFF={5};
__complex__ float time_delay[16][2] BUFF ={2,2,4,5,3,6,9,8,7,7,4,1,1,1,1,5};
float buff21[32][32] ={45} ;
float buff5[32][32] ={23} ;
float w[32] ;
float rand();
        //random number generator
int main() __attribute__((halt,naked));
int random_seed =0 ;
int siggen();
        //data generation function
__complex__ float conj(__complex__ float a);
void cov1();
        // Covariance matrix computation part1
void cov2();
        //Covariance matrix computation part2
void fourier();
void house(float x[32],int k);                    //Householder vector function
void eig1(int f);
void QR();
        //QR decomposition
void sym();
        //Householder method
void QRshift(int f);                                          // QR
method
int count4;
void eigfunc();
void bubbleSort(float vals[], int array_size);
```

```
void music2();                                                                  //
Power spectrum
float Q[32][32]={55};
float R[32][32] ={55} ;
 float H[32][32]={55};
__complex__ float W[32] ={

#include "coeff.dat"

};
__complex__ float xnt1[64][16] XM;
__complex__ float xnt2[64][16] XM;
__complex__ float xnt3[64][16] XM;
__complex__ float xnt4[64][16] XM;
__complex__ float xnt5[64][16] XM;
__complex__ float xnt6[64][16] XM;
__complex__ float xnt7[64][16] XM;
__complex__ float xnt8[64][16] XM;
__complex__ float xnt9[64][16] XM;
__complex__ float xnt10[64][16] XM;
__complex__ float xnt11[64][16] XM;
__complex__ float xnt12[64][16] XM;
__complex__ float xnt13[64][16] XM;
__complex__ float xnt14[64][16] XM;
__complex__ float xnt15[64][16] XM;
__complex__ float xnt16[64][16] XM;
__complex__ float xnt17[64][16] XM;
__complex__ float xnt18[64][16] XM;
__complex__ float xnt19[64][16] XM;
__complex__ float xnt20[64][16] XM;
__complex__ float xnt21[64][16] XM;
__complex__ float xnt22[64][16] XM;
__complex__ float xnt23[64][16] XM;
__complex__ float xnt24[64][16] XM;
__complex__ float xnt25[64][16] XM;
__complex__ float xnt26[64][16] XM;
__complex__ float xnt27[64][16] XM;
__complex__ float xnt28[64][16] XM;
__complex__ float xnt29[64][16] XM;
__complex__ float xnt30[64][16] XM;
__complex__ float xnt31[64][16] XM;
__complex__ float xnt32[64][16] XM;
__complex__ float xnt33[64][16] XM;
__complex__ float xnt34[64][16] XM;
__complex__ float xnt35[64][16] XM;
__complex__ float xnt36[64][16] XM;
```

```
__complex__ float xnt37[64][16] XM;
__complex__ float xnt38[64][16] XM;
__complex__ float xnt39[64][16] XM;
__complex__ float xnt40[64][16] XM;
__complex__ float xnt41[64][16] XM;
__complex__ float xnt42[64][16] XM;
__complex__ float xnt43[64][16] XM;
__complex__ float xnt44[64][16] XM;
__complex__ float xnt45[64][16] XM;
__complex__ float xnt46[64][16] XM;
__complex__ float xnt47[64][16] XM;
__complex__ float xnt48[64][16] XM;
__complex__ float xnt49[64][16] XM;
__complex__ float xnt50[64][16] XM;
__complex__ float xnt51[64][16] XM;
__complex__ float xnt52[64][16] XM;
__complex__ float xnt53[64][16] XM;
__complex__ float xnt54[64][16] XM;
__complex__ float xnt55[64][16] XM;
__complex__ float xnt56[64][16] XM;
__complex__ float xnt57[64][16] XM;
__complex__ float xnt58[64][16] XM;
__complex__ float xnt59[64][16] XM;
__complex__ float xnt60[64][16] XM;
__complex__ float xnt61[64][16] XM;
__complex__ float xnt62[64][16] XM;
__complex__ float xnt63[64][16] XM;
__complex__ float xnt64[64][16] XM;
__complex__ float bcov1[16][16] XM;
__complex__ float bcov2[16][16] XM;
__complex__ float bcov3[16][16] XM;
__complex__ float bcov4[16][16] XM;
__complex__ float bcov5[16][16] XM;
__complex__ float bcov6[16][16] XM;
__complex__ float bcov7[16][16] XM;
__complex__ float bcov8[16][16] XM;
__complex__ float bcov9[16][16] XM;
__complex__ float bcov10[16][16] XM;
__complex__ float bcov11[16][16] XM;
__complex__ float bcov12[16][16] XM;
__complex__ float bcov13[16][16] XM;
__complex__ float bcov14[16][16] XM;
__complex__ float bcov15[16][16] XM;
__complex__ float bcov16[16][16] XM;
__complex__ float bcov17[16][16] XM;
__complex__ float bcov18[16][16] XM;
```

```
__complex__ float bcov19[16][16] XM;
__complex__ float bcov20[16][16] XM;
__complex__ float bcov21[16][16] XM;
__complex__ float bcov22[16][16] XM;
__complex__ float bcov23[16][16] XM;
__complex__ float bcov24[16][16] XM;
__complex__ float bcov25[16][16] XM;
__complex__ float bcov26[16][16] XM;
__complex__ float bcov27[16][16] XM;
__complex__ float bcov28[16][16] XM;
__complex__ float bcov29[16][16] XM;
__complex__ float bcov30[16][16] XM;
__complex__ float bcov31[16][16] XM;
__complex__ float bcov32[16][16] XM;
__complex__ float bcov33[16][16] XM;
int main()
{
            siggen();
          fourier();
 count4 =0;
          cov1();
           cov2();
           eig1(0);
           sym();
           QRshift(30);
          eigfunc();
          bubbleSort(eigenvalues, 32);
          music2();
          focus();
          eig2();
          sym();
           QRshift(30);

          eigfunc();
          bubbleSort(eigenvalues, 32);
          music2();
          return 0;
}
__complex__ float conj(__complex__ float a){
          __complex__ float b;
          __real__ b=__real__ a;
          __imag__ b = -1*__imag__ a;
          return b;}
float rand()
{
          float test;
```

```
  random_seed = random_seed * 25173 +13849;
 // return (unsigned int)(random_seed );
        random_seed=(random_seed  % 1024);
        test = ((float)(random_seed))/1024;
  return (test);
}
int siggen(){
        float azi[2]={9,50};
        float sig_pow[2]={1,1};
        float nos_pow=0.1;
        int count,count2,count3;
        float temp;
        int index,index2,index3;
        __complex__ float sig_temp;
        __complex__ float sig[64][2];
        nos_pow=sqrt(nos_pow);
        for (count =0;count<nss;count++){
                azi[count]=azi[count]*Pi/180;
        }
        nos_pow = sqrt(nos_pow);
        for (count2 =0;count2<ncc;count2++){
                for (count=0;count<nss;count++){
                        temp = sin(azi[count])*omega*dist*count2/velocity;
                        //temp = temp*omega*count2*dist;
                __real__ time_delay[count2][count]=  cos(temp);
                __imag__ time_delay[count2][count]=  sin(temp)*-1;
        //
                }

        }
//      index2=index*64;
//      index3= index2+64;
        for (count = 0;count<64;count++){
                for (count2 = 0;count2<16;count2++){
                        nos[count][count2]=rand();
                nos[count][count2]=nos[count][count2]*nos_pow;
                }
        }
        for (count = 0;count<64;count++){
                for (count2 = 0;count2<2;count2++){
                        sig[count][count2]=rand();
                        sig[count][count2]=sig[count][count2]*sig_pow[count2];
                }
        }
        for (count = 0;count<64;count++){
                for (count2 = 0;count2<16;count2++){
```

```
				sig_temp =0;
				for (count3 = 0;count3<2;count3++){
					sig_temp
=sig_temp+sig[count][count3]*time_delay[count2][count3];
				}
				nos[count][count2] =nos[count][count2] +sig_temp;
			}
		}

			P32XM_B(nos,xnt1,1024);
		return 0;
}
void fourier(){
		int count1,count2,count3;
		__complex__ float buff1[64];
		__complex__ float buff2[64];
		__complex__ float temp1[64];
XM2P3_B(nos,xnt1,1024);
		for (count1 =0;count1<16;count1++){

			for (count2 =0;count2<64;count2++){
			buff1[count2]=nos[count2][count1];
		//~ //	test1[count2]   =buff1[count2];
			}
			fft64(&W[0], &buff1[0], &temp1[0], &buff2[0]);
			for (count2 =0;count2<64;count2++){
			nos[count2][count1]   =buff2[count2];
			}

		}
P32XM_B(nos,xnt1,1024);
}
void cov1(){
		__complex__ float temp[64][16];
		int count,count2,count3;
		for (count=0;count<64;count++){
XM2P3_B(nos,xnt1,1024);
		for (count3 =0;count3<16;count3++){
			temp[count][count3]=nos[count4][count3];
		}
		}
		for (count2 =0;count2<64;count2++){
			for (count =0;count<16;count++){
				nos[count2][count]=temp[count2][count];
			}
		}
```

```
}
void cov2(){
        int count,count2,count3;
        __complex__ float bcov[16][16];
        for (count =0;count<16;count++){
                for(count2=0;count2<16;count2++){
                        bcov[count][count2]=0;
                }
        }
        for (count=0;count<64;count++){
                for(count2=0;count2<16;count2++){
                        for(count3=0;count3<16;count3++){
        bcov[count2][count3]=bcov[count2][count3]+(nos[count][count2]*conj(nos[coun
t][count3]));
                        }
                }
        }
        for (count =0;count<16;count++){
                for(count2=0;count2<16;count2++){
                        bcov[count][count2]=bcov[count][count2]/64;
                nos[count][count2]=bcov[count][count2];
                }
        }
P32XM_B(nos,bcov1,64);
}
void eig1(int f){
        int x,y;
        switch(f) {
        case 0: XM2P3_B(nos,bcov1,64);
                        break;
        case 1: XM2P3_B(nos,bcov2,64);
                        break;
        case 2: XM2P3_B(nos,bcov3,64);
                        break;
        case 3: XM2P3_B(nos,bcov4,64);
                        break;
        case 4: XM2P3_B(nos,bcov5,64);
                        break;
        case 5: XM2P3_B(nos,bcov6,64);
                        break;
        case 6: XM2P3_B(nos,bcov7,64);
                        break;
        case 7: XM2P3_B(nos,bcov8,64);
                        break;
        case 8: XM2P3_B(nos,bcov9,64);
                        break;
```

```
case 9: XM2P3_B(nos,bcov10,64);
                break;
case 10:        XM2P3_B(nos,bcov11,64);
                break;
case 11:        XM2P3_B(nos,bcov12,64);
                break;
case 12:        XM2P3_B(nos,bcov13,64);
                break;
case 13:        XM2P3_B(nos,bcov14,64);
                break;
case 14:        XM2P3_B(nos,bcov15,64);
                break;
case 15:        XM2P3_B(nos,bcov16,64);
                break;
case 16:        XM2P3_B(nos,bcov17,64);
                break;
case 17:        XM2P3_B(nos,bcov18,64);
                break;
case 18:        XM2P3_B(nos,bcov19,64);
                break;
case 19:        XM2P3_B(nos,bcov20,64);
                break;
case 20:        XM2P3_B(nos,bcov21,64);
                break;
case 21:        XM2P3_B(nos,bcov22,64);
                break;
case 22:        XM2P3_B(nos,bcov23,64);
                break;
case 23:        XM2P3_B(nos,bcov24,64);
                break;
case 24:        XM2P3_B(nos,bcov25,64);
                break;
case 25:        XM2P3_B(nos,bcov26,64);
                break;
case 26:        XM2P3_B(nos,bcov27,64);
                break;
case 27:        XM2P3_B(nos,bcov28,64);
                break;
case 28:        XM2P3_B(nos,bcov29,64);
                break;
case 29:        XM2P3_B(nos,bcov30,64);
                break;
case 30:        XM2P3_B(nos,bcov31,64);
                break;
case 31:        XM2P3_B(nos,bcov32,64);
                break;
```

```
	case 32:	XM2P3_B(nos,bcov33,64);
			break;
	}
	//~ //XM2P3_B(m,bcov1,16);
	for (x =0;x<16;x++){
		for (y=0;y<16;y++){
			cova[x][y] =__real__ nos[x][y];
		}
	}
	for (x =16;x<32;x++){
		for (y=0;y<16;y++){
			cova[x][y] =(__imag__ nos[x-8][y])*(-1);
		}
	}
	for (x =0;x<16;x++){
		for (y=16;y<32;y++){
			cova[x][y] =(__imag__ nos[x][y-8]);
		}
	}
	for (x =16;x<32;x++){
		for (y=16;y<32;y++){
			cova[x][y] =__real__ nos[x-8][y-8];
		}
	}
}
void house(float x[],int k){
	float s,gamma;
	int i;
	gamma = 0;
	for (i = 0;i<(k-1);i++){
	w[i] = 0;
	}
	for (i =(k-1);i<32;i++){
	gamma += x[i]*x[i];
	}
	gamma = sqrt(gamma);
	s = x[k-1]*x[k-1];
	s = sqrt(s);
	s = gamma+s;
	s = 2*gamma*s;
	s = sqrt(s);
	if (x[k-1]>= 0){
	w[k-1] = x[k-1]+gamma;
	w[k-1] = w[k-1]/s;
	}
	else {
```

```
		w[k-1] = x[k-1]-gamma;
		w[k-1] = w[k-1]/s;
		}
		for (i = k;i<32;i++){
		w[i] = x[i]/s;
		}
}
void QR(){
float buff2[32];
__complex__ float buff[32][32];
//float buff3[32][32];
		int i,j,x,y,z;
		for (i = 0;i<32;i++){
			for (j = 0;j<32;j++){
				if (i ==j) __imag__ buff[i][j] =1;
				else __imag__ buff[i][j] = 0;
			Q[i][j] =__imag__ buff[i][j];
			}
		}
		for (i = 0;i<31;i++){
			for (j =0;j<32;j++){
			buff2[j] = cova[j][i];

			}
			house(buff2,(i+1));
			//gamma = 0;
			for ( x =0;x<32;x++){
				for(y = 0;y<32;y++){
				H[x][y] = w[x]*w[y];
				H[x][y] = 2*H[x][y];
				H[x][y] = __imag__ buff[x][y] - H[x][y];
				__real__ buff[x][y] =0;
				}
			}
			for ( x =0;x<32;x++){
				for (  y =0; y<32; y++)		{
					for ( z =0; z<32; z++) {
					__real__ buff[x][y] += Q[x][z]*H[z][y];
					}
				}
			}
			for ( x =0;x<32;x++){
				for(y = 0;y<32;y++){
				Q[x][y] =__real__ buff[x][y];
				__real__ buff[x][y] = 0;
				}
```

```
                }
                for ( x =0;x<32;x++){
                        for (  y =0; y<32; y++)          {
                                for (  z =0; z<32; z++){
                                __real__ buff[x][y] += H[x][z]*cova[z][y];
                                }
                        }
                }
                for (  x =0;x<32;x++){
                                for( y = 0;y<32;y++){
                                        cova[x][y] =__real__ buff[x][y];
                                        R[x][y] =__real__ buff[x][y];
                                }
                        }
                        }
        }
void sym(){
__complex__ float buff[32][32];
float buff2[32];
float buff3[32][32];
//~ float buff4[n2][n2];
//~ float buff6[n2][n2];
//~ float buff7[n2][n2];
        float y[32];
        float z[32];
        float delta;
        int x,s,i,j,v;
        for ( x =0;x<32;x++){
                for ( s =0;s<32;s++){
                        if (x ==s) __real__ buff[x][s] =1;
                        else __real__ buff[x][s] =0;
                        buff5[x][s] = __real__ buff[x][s];
                }
        }
        for ( i =0;i<14;i++){
                for ( j =0;j<32;j++){
                        buff2[j] = cova[j][i];
                        z[j] = 0;
                        y[j] = 0;
                }
        delta =0;
        house(buff2,(i+2));
        for ( x =0;x<32;x++){
                for ( s =0;s<32;s++){
                                if (x ==s) __real__ buff[x][s] =1;
                        else __real__ buff[x][s] =0;
```

```
				__imag__ buff[x][s] = w[x]*w[s];
				__imag__ buff[x][s] = 2*__imag__ buff[x][s];
				__imag__ buff[x][s] = __real__ buff[x][s] -__imag__ buff[x][s];
			}
		}
		for (x = 0;x<32;x++){
				for ( s =0;s<32;s++){
				for (v =0; v<32; v++){
					buff3[x][s]+= __imag__buff[x][v]*cova[v][s];
				}
				}
			}
			for (x = 0;x<32;x++){
				for ( s =0;s<32;s++){
				for (v =0; v<32; v++){
					cova[x][s]+= buff3[x][v]* __imag__buff[v][s];
				}
				}
			}
		}
}
void QRshift(int f){
__complex__ float  buff[32][32];
		float  buff7[32][32];
float buff2[32];
int x,y,i,z;
		for (x =0;x<32;x++){
			for (y =0;y<32;y++){
				__imag__ buff[x][y] =0;
				if (x ==y) __real__ buff[x][y] =1;
				else __real__ buff[x][y] =0;
			}
		}
		for (i=0;i<f;i++){
		QR();
		for (x =0;x<32;x++){
				for (y =0; y<32; y++)  {
				buff7[x][y] =0;
					__imag__ buff[x][y] = 0;
					for (z =0; z<32; z++)   {
						buff7[x][y] += R[x][z]*Q[z][y];
				__imag__ buff[x][y]+= __real__ buff[x][z]*Q[z][y];
					}
				//		if (fabs(buff[x][y])<1e-6) buff[x][y]=0;
				}
			}
```

```
for (  x =0;x<32;x++){
                for (y =0; y<32; y++)  {
                        cova[x][y] =buff7[x][y];
                        __real__ buff[x][y]=__imag__ buff[x][y];
        //              buff6[x][y]=__imag__ buff[x][y];
                        buff21[x][y]=__imag__ buff[x][y];
                }
            }
        }
int j;
for ( x =0;x<32;x+=2){
j=x/2;
eigenvalues[j]=buff7[x][x];
}
}
void eigfunc(){
int x,s,j,y,z;
        float test2;
        float buff23[32][32];
for ( x =0;x<32;x++){
        for (  s =0; s<32; s++) {
                buff23[x][s] = 0;
                for ( z =0; z<32; z++) {
                        buff23[x][s]+= buff5[x][z]*buff21[z][s];
                }
//              float test2 =fabs(buff23[x][s]);
//              if (fabs(test2)<1e-4){
//                      buff23[x][s] =0;}
        }
}
for (  x =0;x<32;x++){

                        for( y = 0;y<32;y+=2){
                        j=y/2;
                        eigvects[x][j]=buff23[x][y];
                        }
                }
}
void bubbleSort(float vals[], int array_size)
{
  int i, j, k;
float temp1;
float temp2[32];
  for (i = (array_size - 1); i >= 0; i--)
  {
    for (j = 1; j <= i; j++)
```

```
    {
     if (vals[j-1] < vals[j])
     {
       temp1 = vals[j-1];
       vals[j-1] = vals[j];
       vals[j] = temp1;
                    for(k =0;k<32;k++){
                    temp2[k]=eigvects[k][j-1];
                    }
                    for(k =0;k<32;k++){
                    eigvects[k][j-1]=eigvects[k][j];
                    }
                    for(k =0;k<32;k++){
                    eigvects[k][j]=temp2[k];
                    }
     }
    }
  }
  for (i=0;i<16;i++){
            for (j=0;j<16;j++){
                    __real__ eigvects2[i][j]=  eigvects[i][j];
                    __imag__ eigvects2[i][j]= eigvects[i+8][j];
            }
  }
}
void music2(){
            int k,count;
            int l,v;
            float temp,temp2;
            int kk;
            __complex__ float sr;
__complex__ float array[16]={6};
__complex__ float array2[16]={7};
//__complex__ float Part[15] ={5};
__complex__ float Part2[15] ={5};
float xxx[91]= {5};
__complex__ float En2[15][16] ={5};
            __vector__ int error;
            for (l = 0;l<16;l++){
                    for (v =0;v<7;v++){
                            En[l][v]=eigvects2[l][v+1];
                            En2[v][l]=eigvects2[l][v+1];
                    }
            }
            for (k=0;k<=90;k++){
                    xxx[k]=k*Pi;
```

```
		xxx[k] =xxx[k]/180;
		temp = sin(xxx[k]);
		temp = temp*Pi;
		for (count = 0;count<16;count++){
			temp2 = temp*count;
			__real__ array[count]=  cos(temp2);
			__real__ array2[count]=  cos(temp2);
			__imag__ array[count]=  sin(temp2)*-1;
			__imag__ array2[count]=  sin(temp2)*-1;
		}
		int s,j;
	for (s=0;s<7;s++){
			Part2[s] =0;
			for (kk=0;kk<16;kk++){
				Part2[s]+=conj(En2[s][kk])*array[kk];
			}
	}
		for (j=0;j<7;j++){
			Part[j] =0;
			for (kk=0;kk<16;kk++){
				Part[j]+=conj(array2[kk])*En[kk][j];
			}
		}
		Pm[k] =0;
		sr =0;
			for (kk=0;kk<7;kk++){
				sr+=Part[kk]*Part2[kk];
			}
	Pm[k] = (__real__ sr * __real__ sr)+(__imag__ sr*__imag__ sr);
	Pm[k]=sqrt(Pm[k]);
	}
}
void focus(){
	float f0;
	float temp,temp2,temp3;
	int t,i,j,i2,j2,k2,l2,m2;
	//b1=10.5;
	for (i =0;i<nfib;i++){
	fbin[i] =(i+1)*300/16;
	}
	for (i =0;i<ncc;i++){
		for (j=0;j<ncc;j++){
			rn[i][j] =0;
			signal[i][j] =0;
		}
	}
```

```
t = (nfib+1)/2;
f0 =fbin[t-1];

for(i=0;i<nfib;i++){
	for(i2=0;i2<nfib;i2++){
		for(j2=0;j2<nfib;j2++){
			t1[i2][j2] =0;
			tp1[i2][j2] =0;
			tc1[i2][j2] =0;
			tc2[i2][j2] =0;
			tt1[i2][j2] =0;
		}
	}
	omg =2*Pi*(f0-fbin[i]);
	b1=20;
	b1 =b1*Pi;
	b1=b1/180;
	temp =sin(b1);
	temp =sin(b1)/300000000;
	temp =temp*500000;
	temp =temp*omg;
	//
	for(k2=0;k2<ncc;k2++){
		temp2= temp*k2;
		__real__ t1[k2][k2]=  cos(temp2);
		__imag__ t1[k2][k2]=  sin(temp2)*-1;
		}
switch (i){
	case 0:
		XM2P3_B(tpp,bcov1,256);
		break;
	case 1:
		XM2P3_B(tpp,bcov2,256);
		break;
	case 2:
		XM2P3_B(tpp,bcov3,256);
		break;
	case 3:
		XM2P3_B(tpp,bcov4,256);
		break;
	case 4:
		XM2P3_B(tpp,bcov5,256);
		break;
	case 5:
		XM2P3_B(tpp,bcov6,256);
```

```
		break;
	case 6:
		XM2P3_B(tpp,bcov7,256);
		break;
	case 7:
		XM2P3_B(tpp,bcov8,256);
		break;
	case 8:
		XM2P3_B(tpp,bcov9,256);
		break;
	case 9:
		XM2P3_B(tpp,bcov10,256);
		break;
	case 10:
		XM2P3_B(tpp,bcov11,256);
		break;
	case 11:
		XM2P3_B(tpp,bcov12,256);
		break;
	case 12:
		XM2P3_B(tpp,bcov13,256);
		break;
	case 13:
		XM2P3_B(tpp,bcov14,256);
		break;
	case 14:
		XM2P3_B(tpp,bcov15,256);
		break;
	case 15:
		XM2P3_B(tpp,bcov16,256);
		break;
	case 16:
		XM2P3_B(tpp,bcov17,256);
		break;
	case 17:
		XM2P3_B(tpp,bcov18,256);
		break;
	case 18:
		XM2P3_B(tpp,bcov19,256);
		break;
	case 19:
		XM2P3_B(tpp,bcov20,256);
		break;
	case 20:
		XM2P3_B(tpp,bcov21,256);
		break;
```

```
case 21:
        XM2P3_B(tpp,bcov22,256);
        break;
case 22:
        XM2P3_B(tpp,bcov23,256);
        break;
        case 23:
        XM2P3_B(tpp,bcov24,256);
        break;
        case 24:
        XM2P3_B(tpp,bcov25,256);
        break;
        case 25:
        XM2P3_B(tpp,bcov26,256);
        break;
        case 26:
        XM2P3_B(tpp,bcov27,256);
        break;
        case 27:
        XM2P3_B(tpp,bcov28,256);
        break;
        case 28:
        XM2P3_B(tpp,bcov29,256);
        break;
        case 29:
        XM2P3_B(tpp,bcov30,256);
        break;
        case 30:
        XM2P3_B(tpp,bcov31,256);
        break;
        case 31:
        XM2P3_B(tpp,bcov32,256);
        break;
        case 32:
        XM2P3_B(tpp,bcov33,256);
        break;
}
        for(i2=0;i2<ncc;i2++){
                tp1[i2][i2] =conj(t1[i2][i2]);
                tt1[i2][i2]=t1[i2][i2]*tp1[i2][i2];
        }
        for (  i2 =0;i2<ncc;i2++){
for (  j2 =0; j2<ncc; j2++)       {
        tc1[i2][j2] = 0;
        for (k2 =0; k2<ncc; k2++){
                tc1[i2][j2]+= t1[i2][k2]*tpp[k2][j2];
```

```
                    }
                }
            }
                for (  i2 =0;i2<ncc;i2++){
                for (  j2 =0; j2<ncc; j2++)        {
                    tc2[i2][j2] = 0;
                    for (k2 =0; k2<ncc; k2++){
                        tc2[i2][j2]+= tc1[i2][k2]*tp1[k2][j2];
                    }
                    signal[i2][j2] = signal[i2][j2]+tc2[i2][j2];
                }
            }
            }
            for (  i2 =0;i2<ncc;i2++){
            signal[i2][i2]=__real__ (signal[i2][i2]);
            }
        }
        void eig2(){
            int i2,j2;
                for (i2 =0;i2<16;i2++){
                for (j2=0;j2<16;j2++){
                    A[i2][j2] =__real__ signal[i2][j2]*1e8;
                }
            }
            for (i2 =16;i2<32;i2++){
                for (j2=0;j2<16;j2++){
                    A[i2][j2] =__imag__ signal[i2-16][j2]*1e8;
                }
            }
            for (i2 =0;i2<16;i2++){
                for (j2=16;j2<32;j2++){
                    A[i2][j2] =(__imag__ signal[i2][j2-16])*(-1)*1e8;
                }
            }
            for (i2 =16;i2<32;i2++){
                for (j2=16;j2<32;j2++){
                A[i2][j2] =__real__ signal[i2-16][j2-16]*1e8;
                }
            }
        }
```

parallel.c - This file contains the parallel architecture source code.

```
#include "math.h"
#include "magic.h"
#include "DSPlib.h"
#define Pi 3.1415
#define nss 2
#define velocity 300000000
#define omega 1885
#define dist 500000
#define ncc 16
float Pm[91]={23};
float eigenvalues[16]={333};
float eigvects[32][16]={14};
float cova[32][32]={55};
__complex__ float Part[15] ={5};
__complex__ float eigvects2[16][16] ={5};
__complex__ float En[16][15] ={15};
__complex__ float nos[64][16] BUFF={5};
__complex__ float time_delay[16][2] BUFF ={2,2,4,5,3,6,9,8,7,7,4,1,1,1,1,5};
float buff21[32][32] ={45} ;
float buff5[32][32] ={23} ;
float w[32] ;
float rand();
    //random number generator
int main() __attribute__((halt,naked));
int random_seed =0 ;
int siggen();
    //data generation function
__complex__ float conj(__complex__ float a);
void cov1();
    // Covariance matrix computation part1
void cov2();
    //Covariance matrix computation part2
void fourier();
void house(float x[32],int k);                              //Householder vector
function
void eig1(int f);
void QR();
    //QR decomposition
void sym();
    //Householder method
void QRshift(int f);
    // QR method
int count4;
void eigfunc();
```

```
void bubbleSort(float vals[], int array_size);
void music2();
float Q[32][32]={55};
float R[32][32] ={55} ;
 float H[32][32]={55};
__complex__ float W[32] ={

#include "coeff.dat"

};
__complex__ float xnt1[64][16] XM;
__complex__ float xnt2[64][16] XM;
__complex__ float xnt3[64][16] XM;
__complex__ float xnt4[64][16] XM;
__complex__ float xnt5[64][16] XM;
__complex__ float xnt6[64][16] XM;
__complex__ float xnt7[64][16] XM;
__complex__ float xnt8[64][16] XM;
__complex__ float xnt9[64][16] XM;
__complex__ float xnt10[64][16] XM;
__complex__ float xnt11[64][16] XM;
__complex__ float xnt12[64][16] XM;
__complex__ float xnt13[64][16] XM;
__complex__ float xnt14[64][16] XM;
__complex__ float xnt15[64][16] XM;
__complex__ float xnt16[64][16] XM;
__complex__ float xnt17[64][16] XM;
__complex__ float xnt18[64][16] XM;
__complex__ float xnt19[64][16] XM;
__complex__ float xnt20[64][16] XM;
__complex__ float xnt21[64][16] XM;
__complex__ float xnt22[64][16] XM;
__complex__ float xnt23[64][16] XM;
__complex__ float xnt24[64][16] XM;
__complex__ float xnt25[64][16] XM;
__complex__ float xnt26[64][16] XM;
__complex__ float xnt27[64][16] XM;
__complex__ float xnt28[64][16] XM;
__complex__ float xnt29[64][16] XM;
__complex__ float xnt30[64][16] XM;
__complex__ float xnt31[64][16] XM;
__complex__ float xnt32[64][16] XM;
__complex__ float xnt33[64][16] XM;
__complex__ float xnt34[64][16] XM;
__complex__ float xnt35[64][16] XM;
__complex__ float xnt36[64][16] XM;
```

```
__complex__ float xnt37[64][16] XM;
__complex__ float xnt38[64][16] XM;
__complex__ float xnt39[64][16] XM;
__complex__ float xnt40[64][16] XM;
__complex__ float xnt41[64][16] XM;
__complex__ float xnt42[64][16] XM;
__complex__ float xnt43[64][16] XM;
__complex__ float xnt44[64][16] XM;
__complex__ float xnt45[64][16] XM;
__complex__ float xnt46[64][16] XM;
__complex__ float xnt47[64][16] XM;
__complex__ float xnt48[64][16] XM;
__complex__ float xnt49[64][16] XM;
__complex__ float xnt50[64][16] XM;
__complex__ float xnt51[64][16] XM;
__complex__ float xnt52[64][16] XM;
__complex__ float xnt53[64][16] XM;
__complex__ float xnt54[64][16] XM;
__complex__ float xnt55[64][16] XM;
__complex__ float xnt56[64][16] XM;
__complex__ float xnt57[64][16] XM;
__complex__ float xnt58[64][16] XM;
__complex__ float xnt59[64][16] XM;
__complex__ float xnt60[64][16] XM;
__complex__ float xnt61[64][16] XM;
__complex__ float xnt62[64][16] XM;
__complex__ float xnt63[64][16] XM;
__complex__ float xnt64[64][16] XM;
__complex__ float bcov1[16][16] XM;
__complex__ float bcov2[16][16] XM;
__complex__ float bcov3[16][16] XM;
__complex__ float bcov4[16][16] XM;
__complex__ float bcov5[16][16] XM;
__complex__ float bcov6[16][16] XM;
__complex__ float bcov7[16][16] XM;
__complex__ float bcov8[16][16] XM;
__complex__ float bcov9[16][16] XM;
__complex__ float bcov10[16][16] XM;
__complex__ float bcov11[16][16] XM;
__complex__ float bcov12[16][16] XM;
__complex__ float bcov13[16][16] XM;
__complex__ float bcov14[16][16] XM;
__complex__ float bcov15[16][16] XM;
__complex__ float bcov16[16][16] XM;
__complex__ float bcov17[16][16] XM;
__complex__ float bcov18[16][16] XM;
```

```
__complex__ float bcov19[16][16] XM;
__complex__ float bcov20[16][16] XM;
__complex__ float bcov21[16][16] XM;
__complex__ float bcov22[16][16] XM;
__complex__ float bcov23[16][16] XM;
__complex__ float bcov24[16][16] XM;
__complex__ float bcov25[16][16] XM;
__complex__ float bcov26[16][16] XM;
__complex__ float bcov27[16][16] XM;
__complex__ float bcov28[16][16] XM;
__complex__ float bcov29[16][16] XM;
__complex__ float bcov30[16][16] XM;
__complex__ float bcov31[16][16] XM;
__complex__ float bcov32[16][16] XM;
__complex__ float bcov33[16][16] XM;
int main()
{
     siggen();
    fourier();
 count4 =0;
    cov1();
     cov2();
     eig1(0);
     sym();
     QRshift(30);

    eigfunc();
    bubbleSort(eigenvalues, 32);
    music2();
    focus();
    eig2();
    sym();
     QRshift(30);

    eigfunc();
    bubbleSort(eigenvalues, 32);
    music2();
    return 0;
}
__complex__ float conj(__complex__ float a){
    __complex__ float b;
    __real__ b=__real__ a;
    __imag__ b = -1*__imag__ a;

    return b;}
float rand()
```

```
{
     float test;
  random_seed = random_seed * 25173 +13849;
 // return (unsigned int)(random_seed );
     random_seed=(random_seed  % 1024);
     test = ((float)(random_seed))/1024;
  return (test);
}
int siggen(){
     float azi[2]={9,50};
     float sig_pow[2]={1,1};
     float nos_pow=0.1;
     int count,count2,count3;
     float temp;
     int index,index2,index3;

     __complex__ float sig_temp;
     __complex__ float sig[64][2];
     nos_pow=sqrt(nos_pow);

     for (count =0;count<nss;count++){
             azi[count]=azi[count]*Pi/180;

     }
     nos_pow = sqrt(nos_pow);
     for (count2 =0;count2<ncc;count2++){
             for (count=0;count<nss;count++){
                     temp = sin(azi[count])*omega*dist*count2/velocity;
                     //temp = temp*omega*count2*dist;

             __real__ time_delay[count2][count]=  cos(temp);
             __imag__ time_delay[count2][count]=  sin(temp)*-1;
     //
             }


     }

//   index2=index*64;
//   index3= index2+64;


     for (count = 0;count<64;count++){
             for (count2 = 0;count2<16;count2++){
                     nos[count][count2]=rand();
             nos[count][count2]=nos[count][count2]*nos_pow;
```

```
		}
	}
	for (count = 0;count<64;count++){
		for (count2 = 0;count2<2;count2++){
			sig[count][count2]=rand();
			sig[count][count2]=sig[count][count2]*sig_pow[count2];
		}
	}
	for (count = 0;count<64;count++){
		for (count2 = 0;count2<16;count2++){
			sig_temp =0;
			for (count3 = 0;count3<2;count3++){
				sig_temp
=sig_temp+sig[count][count3]*time_delay[count2][count3];
			}
			nos[count][count2] =nos[count][count2] +sig_temp;
		}
	}

		P32XM_B(nos,xnt1,1024);

	return 0;

}

void fourier(){

	int count1,count2,count3;
	__complex__ float buff1[64];
	__complex__ float buff2[64];
	__complex__ float temp1[64];
XM2P3_B(nos,xnt1,1024);

	for (count1 =0;count1<16;count1++){

		for (count2 =0;count2<64;count2++){
		buff1[count2]=nos[count2][count1];
	//~ //	test1[count2]	=buff1[count2];
		}
		fft64(&W[0], &buff1[0], &temp1[0], &buff2[0]);
		for (count2 =0;count2<64;count2++){
		nos[count2][count1]	=buff2[count2];

		}
	}
```

```
P32XM_B(nos,xnt1,1024);

}
void cov1(){
     __complex__ float temp[64][16];
     int count,count2,count3;
     for (count=0;count<64;count++){

XM2P3_B(nos,xnt1,1024);


     for (count3 =0;count3<16;count3++){
               temp[count][count3]=nos[count4][count3];
     }



     }

     for (count2 =0;count2<64;count2++){
               for (count =0;count<16;count++){
                         nos[count2][count]=temp[count2][count];

               }

     }

}
void cov2(){
     int count,count2,count3;
     __complex__ float bcov[16][16];
     for (count =0;count<16;count++){
               for(count2=0;count2<16;count2++){
                         bcov[count][count2]=0;
               }

     }

     for (count=0;count<64;count++){
               for(count2=0;count2<16;count2++){
                         for(count3=0;count3<16;count3++){

     bcov[count2][count3]=bcov[count2][count3]+(nos[count][count2]*conj(nos[coun
t][count3]));

                         }
               }
```

```
	}
	for (count =0;count<16;count++){
		for(count2=0;count2<16;count2++){
			bcov[count][count2]=bcov[count][count2]/64;
		nos[count][count2]=bcov[count][count2];
		}

	}

P32XM_B(nos,bcov1,64);
}
void eig1(int f){
	int x,y;

	switch(f) {
	case 0: XM2P3_B(nos,bcov1,64);
			break;
	case 1: XM2P3_B(nos,bcov2,64);
			break;
	case 2: XM2P3_B(nos,bcov3,64);
			break;
	case 3: XM2P3_B(nos,bcov4,64);
			break;
	case 4: XM2P3_B(nos,bcov5,64);
			break;
	case 5: XM2P3_B(nos,bcov6,64);
			break;
	case 6: XM2P3_B(nos,bcov7,64);
			break;
	case 7: XM2P3_B(nos,bcov8,64);
			break;
	case 8: XM2P3_B(nos,bcov9,64);
			break;
	case 9: XM2P3_B(nos,bcov10,64);
			break;
	case 10:		XM2P3_B(nos,bcov11,64);
			break;
	case 11:		XM2P3_B(nos,bcov12,64);
			break;
	case 12:		XM2P3_B(nos,bcov13,64);
			break;
	case 13:		XM2P3_B(nos,bcov14,64);
			break;
	case 14:		XM2P3_B(nos,bcov15,64);
			break;
	case 15:		XM2P3_B(nos,bcov16,64);
```

```
                break;
case 16:        XM2P3_B(nos,bcov17,64);
                break;
case 17:        XM2P3_B(nos,bcov18,64);
                break;
case 18:        XM2P3_B(nos,bcov19,64);
                break;
case 19:        XM2P3_B(nos,bcov20,64);
                break;
case 20:        XM2P3_B(nos,bcov21,64);
                break;
case 21:        XM2P3_B(nos,bcov22,64);
                break;
case 22:        XM2P3_B(nos,bcov23,64);
                break;
case 23:        XM2P3_B(nos,bcov24,64);
                break;
case 24:        XM2P3_B(nos,bcov25,64);
                break;
case 25:        XM2P3_B(nos,bcov26,64);
                break;
case 26:        XM2P3_B(nos,bcov27,64);
                break;
case 27:        XM2P3_B(nos,bcov28,64);
                break;
case 28:        XM2P3_B(nos,bcov29,64);
                break;
case 29:        XM2P3_B(nos,bcov30,64);
                break;
case 30:        XM2P3_B(nos,bcov31,64);
                break;
case 31:        XM2P3_B(nos,bcov32,64);
                break;
case 32:        XM2P3_B(nos,bcov33,64);
                break;
}
//~ //XM2P3_B(m,bcov1,16);
for (x =0;x<16;x++){

        for (y=0;y<16;y++){

                cova[x][y] =__real__ nos[x][y];

        }
}
for (x =16;x<32;x++){
```

```
		for (y=0;y<16;y++){
			cova[x][y] =(__imag__ nos[x-8][y])*(-1);
		}
	}
	for (x =0;x<16;x++){

		for (y=16;y<32;y++){

			cova[x][y] =(__imag__ nos[x][y-8]);

		}

	}
	for (x =16;x<32;x++){

		for (y=16;y<32;y++){

			cova[x][y] =__real__ nos[x-8][y-8];

		}

	}

}
void house(float x[],int k){
	float s,gamma;
	int i;
	gamma = 0;
	for (i = 0;i<(k-1);i++){
	w[i] = 0;
	}
	for (i =(k-1);i<32;i++){
	gamma += x[i]*x[i];
	}
	gamma = sqrt(gamma);
	s = x[k-1]*x[k-1];
	s = sqrt(s);
	s = gamma+s;
	s = 2*gamma*s;
	s = sqrt(s);
	if (x[k-1]>= 0){

	w[k-1] = x[k-1]+gamma;
	w[k-1] = w[k-1]/s;
	}
```

```
    else {
    w[k-1] = x[k-1]-gamma;
    w[k-1] = w[k-1]/s;
    }
    for (i = k;i<32;i++){
    w[i] = x[i]/s;
    }
}
void QR(){
float buff2[32];
__complex__ float buff[32][32];
//float buff3[32][32];
    int i,j,x,y,z;
    for (i = 0;i<32;i++){
            for (j = 0;j<32;j++){
                    if (i ==j) __imag__ buff[i][j] =1;
                    else __imag__ buff[i][j] = 0;
            Q[i][j] =__imag__ buff[i][j];

            }
    }
    for (i = 0;i<31;i++){
            for (j =0;j<32;j++){
            buff2[j] = cova[j][i];

            }

            house(buff2,(i+1));
            //gamma = 0;
            gamma = 0;
            delta =0;
for(x=0;x<n2;x++){
delta+=buff2[x]*w[x];
    }               for(x=0;x<n2;x++){
buff2[x]=buff2[x]*delta;
    }
     for ( x =0;x<32;x++){
             for ( y =0;y<32;y++){

                     A[x][y]=A[x][y]-buff2[y];
                     Q[x][y]=Q[x][y]-buff2[y];
             }
     }
                    }
    }
void sym(){
```

```
__complex__ float buff[32][32];
float buff2[32];
float buff8[32][32];
//~ float buff4[n2][n2];
//~ float buff6[n2][n2];
//~ float buff7[n2][n2];
     float y[32];
     float z[32];
     float delta;
     int x,s,i,j,v;
     for ( x =0;x<32;x++){
               for ( s =0;s<32;s++){
                         if (x ==s) __real__ buff[x][s] =1;
                         else __real__ buff[x][s] =0;
                         buff5[x][s] = __real__ buff[x][s];
               }
     }
     for ( i =0;i<30;i++){
               for ( j =0;j<32;j++){
                         buff2[j] = cova[j][i];
                         z[j] = 0;
                         y[j] = 0;
               }
     delta =0;
     house(buff2,(i+2));
               for(x=0;x<32;x++){
delta+=buff2[x]*w[x];
     }
     for(x=0;x<n2;x++){
buff2[x]=buff2[x]*delta;
     }
      for ( x =0;x<32;x++){
                for ( s =0;s<32;s++){
                          cova[x][s]=cova[x][s]-buff2[x];
                          buff5[x][s]=buff5[x][s]-buff2[x];
                }
      }
}
                         for (x = 0;x<32;x++){
                         for ( s =0;s<32;s++){
                         for (v =0; v<32; v++){
                                   buff8[x][s]+= cova[x][v]*buff5[v][s];
                         }
                         }
               }
     for (x = 0;x<n2;x++){
```

```
                    for ( s =0;s<n2;s++){
                            A[x][s]= cova[x][s];
                    }
            }
                    }
            }
    }
}
void QRshift(int f){
__complex__ float  buff[32][32];
    float  buff7[32][32];
float buff2[32];
int x,y,i,z;
    for (x =0;x<32;x++){
            for (y =0;y<32;y++){
                    __imag__ buff[x][y] =0;
                    if (x ==y) __real__ buff[x][y] =1;
                    else __real__ buff[x][y] =0;
            }
    }
    for (i=0;i<f;i++){
    QR();
    for (x =0;x<32;x++){
                    for (y =0; y<32; y++)  {
                    buff7[x][y] =0;
                            __imag__ buff[x][y] = 0;
                            for (z =0; z<32; z++)   {
                                    buff7[x][y] += R[x][z]*Q[z][y];
                    __imag__ buff[x][y]+= __real__ buff[x][z]*Q[z][y];
                            }
                    //      if (fabs(buff[x][y])<1e-6) buff[x][y]=0;
                    }
            }
for (  x =0;x<32;x++){
                    for (y =0; y<32; y++)  {
                            cova[x][y] =buff7[x][y];
                            __real__ buff[x][y]=__imag__ buff[x][y];
    //                      buff6[x][y]=__imag__ buff[x][y];
                            buff21[x][y]=__imag__ buff[x][y];
                    }
            }
    }
int j;
for ( x =0;x<32;x+=2){
j=x/2;
eigenvalues[j]=buff7[x][x];
```

```
}
}
void eigfunc(){
int x,s,j,y,z;
     float test2;
     float buff23[32][32];
for ( x =0;x<32;x++){
     for (  s =0; s<32; s++) {
               buff23[x][s] = 0;
               for ( z =0; z<32; z++) {
                         buff23[x][s]+= buff5[x][z]*buff21[z][s];
               }
//             float test2 =fabs(buff23[x][s]);
//             if (fabs(test2)<1e-4){
//                       buff23[x][s] =0;}
     }
}
for (  x =0;x<32;x++){
                         for( y = 0;y<32;y+=2){
                         j=y/2;
                         eigvects[x][j]=buff23[x][y];
                         }
               }
}
void bubbleSort(float vals[], int array_size)
{
  int i, j, k;
float temp1;
float temp2[32];
  for (i = (array_size - 1); i >= 0; i--)
  {
    for (j = 1; j <= i; j++)
    {
     if (vals[j-1] < vals[j])
     {
       temp1 = vals[j-1];
       vals[j-1] = vals[j];
       vals[j] = temp1;
               for(k =0;k<32;k++){
               temp2[k]=eigvects[k][j-1];
               }
               for(k =0;k<32;k++){
               eigvects[k][j-1]=eigvects[k][j];
               }
               for(k =0;k<32;k++){
               eigvects[k][j]=temp2[k];
```

```
            }
      }
    }
  }
  for (i=0;i<16;i++){
        for (j=0;j<16;j++){
              __real__ eigvects2[i][j]=  eigvects[i][j];
              __imag__ eigvects2[i][j]= eigvects[i+8][j];
        }
  }
}
void music2(){
      int k,count;
      int l,v;
      float temp,temp2;
      int kk;
      __complex__ float sr;
__complex__ float array[16]={6};
__complex__ float array2[16]={7};
//__complex__ float Part[15] ={5};
__complex__ float Part2[15] ={5};
float xxx[91]= {5};
__complex__ float En2[15][16] ={5};
      __vector__ int error;
      for (l = 0;l<16;l++){
            for (v =0;v<7;v++){
                  En[l][v]=eigvects2[l][v+1];
                  En2[v][l]=eigvects2[l][v+1];
            }
      }
      for (k=0;k<=90;k++){
            xxx[k]=k*Pi;
            xxx[k] =xxx[k]/180;
            temp = sin(xxx[k]);
            temp = temp*Pi;
            for (count = 0;count<16;count++){
                  temp2 = temp*count;
                  __real__ array[count]=  cos(temp2);
                  __real__ array2[count]=  cos(temp2);
                  __imag__ array[count]=  sin(temp2)*-1;
                  __imag__ array2[count]=  sin(temp2)*-1;
            }
            int s,j;
      for (s=0;s<7;s++){
                  Part2[s] =0;
                  for (kk=0;kk<16;kk++){
```

```
                Part2[s]+=conj(En2[s][kk])*array[kk];
            }
    }
        for (j=0;j<7;j++){
            Part[j] =0;
            for (kk=0;kk<16;kk++){
                Part[j]+=conj(array2[kk])*En[kk][j];
            }
        }
        Pm[k] =0;
        sr =0;
            for (kk=0;kk<7;kk++){
                sr+=Part[kk]*Part2[kk];
            }
    Pm[k] = (__real__ sr * __real__ sr)+(__imag__ sr*__imag__ sr);
    Pm[k]=sqrt(Pm[k]);
    }
}
void focus(){
    float f0;
    float temp,temp2,temp3;
    int t,i,j,i2,j2,k2,l2,m2;
    //b1=10.5;
    for (i =0;i<nfib;i++){
    fbin[i] =(i+1)*300/16;
    }
    for (i =0;i<ncc;i++){
        for (j=0;j<ncc;j++){
            rn[i][j] =0;
            signal[i][j] =0;
        }
    }
    t = (nfib+1)/2;
    f0 =fbin[t-1];
    for(i=0;i<nfib;i++){
        for(i2=0;i2<nfib;i2++){
            for(j2=0;j2<nfib;j2++){
                t1[i2][j2] =0;
                tp1[i2][j2] =0;
                tc1[i2][j2] =0;
                tc2[i2][j2] =0;
                tt1[i2][j2] =0;
            }
        }
        omg =2*Pi*(f0-fbin[i]);
        b1=20;
```

```
        b1 =b1*Pi;
        b1=b1/180;
        temp =sin(b1);
        temp =sin(b1)/300000000;
        temp =temp*500000;
        temp =temp*omg;
        //
        for(k2=0;k2<ncc;k2++){
                temp2= temp*k2;
                __real__ t1[k2][k2]=  cos(temp2);
                __imag__ t1[k2][k2]=  sin(temp2)*-1;
                }
switch (i){
        case 0:
                XM2P3_B(tpp,bcov1,256);
                break;
        case 1:
                XM2P3_B(tpp,bcov2,256);
                break;
        case 2:
                XM2P3_B(tpp,bcov3,256);
                break;
        case 3:
                XM2P3_B(tpp,bcov4,256);
                break;
        case 4:
                XM2P3_B(tpp,bcov5,256);
                break;
        case 5:
                XM2P3_B(tpp,bcov6,256);
                break;
        case 6:
                XM2P3_B(tpp,bcov7,256);
                break;
        case 7:
                XM2P3_B(tpp,bcov8,256);
                break;
        case 8:
                XM2P3_B(tpp,bcov9,256);
                break;
        case 9:
                XM2P3_B(tpp,bcov10,256);
                break;
        case 10:
                XM2P3_B(tpp,bcov11,256);
                break;
```

```
case 11:
        XM2P3_B(tpp,bcov12,256);
        break;
case 12:
        XM2P3_B(tpp,bcov13,256);
        break;
case 13:
        XM2P3_B(tpp,bcov14,256);
        break;
case 14:
        XM2P3_B(tpp,bcov15,256);
        break;
case 15:
        XM2P3_B(tpp,bcov16,256);
        break;
case 16:
        XM2P3_B(tpp,bcov17,256);
        break;
case 17:
        XM2P3_B(tpp,bcov18,256);
        break;
case 18:
        XM2P3_B(tpp,bcov19,256);
        break;
case 19:
        XM2P3_B(tpp,bcov20,256);
        break;
case 20:
        XM2P3_B(tpp,bcov21,256);
        break;
case 21:
        XM2P3_B(tpp,bcov22,256);
        break;
case 22:
        XM2P3_B(tpp,bcov23,256);
        break;
        case 23:
        XM2P3_B(tpp,bcov24,256);
        break;
        case 24:
        XM2P3_B(tpp,bcov25,256);
        break;
        case 25:
        XM2P3_B(tpp,bcov26,256);
        break;
        case 26:
```

```
                    XM2P3_B(tpp,bcov27,256);
                    break;
                    case 27:
                    XM2P3_B(tpp,bcov28,256);
                    break;
                    case 28:
                    XM2P3_B(tpp,bcov29,256);
                    break;
                    case 29:
                    XM2P3_B(tpp,bcov30,256);
                    break;
                    case 30:
                    XM2P3_B(tpp,bcov31,256);
                    break;
                    case 31:
                    XM2P3_B(tpp,bcov32,256);
                    break;
                    case 32:
                    XM2P3_B(tpp,bcov33,256);
                    break;
            }
                    for(i2=0;i2<ncc;i2++){

                            tp1[i2][i2] =conj(t1[i2][i2]);

                            tt1[i2][i2]=t1[i2][i2]*tp1[i2][i2];
                    }
                    for (  i2 =0;i2<ncc;i2++){
            for (  j2 =0; j2<ncc; j2++)       {
                    tc1[i2][j2] = 0;
                    for (k2 =0; k2<ncc; k2++){
                            tc1[i2][j2]+= t1[i2][k2]*tpp[k2][j2];
                    }
            }
    }
            for (  i2 =0;i2<ncc;i2++){
            for (  j2 =0; j2<ncc; j2++)       {
                    tc2[i2][j2] = 0;
                    for (k2 =0; k2<ncc; k2++){
                            tc2[i2][j2]+= tc1[i2][k2]*tp1[k2][j2];
                    }
                    signal[i2][j2] = signal[i2][j2]+tc2[i2][j2];
            }
    }

    }
```