Phased Array Processing: Direction of Arrival Estimation on Reconfigurable Hardware

Master's Thesis
by

Jasper D. Vrielink

Committee:
prof. dr. ir. Gerard J.M. Smit
dr. ir. André B.J. Kokkeler
ir. Marcel D. van de Burgwal
ir. Kenneth C. Rovers

University of Twente, Enschede, The Netherlands
January 16, 2009

Abstract

A beamforming system consists of three different parts, the beamformer, the beamsteering, and the parameter estimation. In this thesis the parameter estimation is described, the Direction Of Arrival (DOA) estimation in particular.

Two popular DOA estimation algorithms are described. The first algorithm is MUltiple SIgnal Classification (MUSIC), and the second algorithm is Estimation of Signal Parameters by Rotational Invariance Techniques (ESPRIT). Both algorithm can be used to estimate the DOAs of multiple signals. Covariance Matrix Differencing (CMD) is an extension to MUSIC to improve the performance of the MUSIC algorithm. This CMD extension is also described in this thesis.

A model of MUSIC and a model of ESPRIT are made in Matlab to analyse the performance, and the effects of different test scenarios on the DOA estimation. Both algorithms are compared by means of these test scenarios. The performance of the CMD extension is also analyzed by means of a set of test scenarios. Based on the superior performance of the MUSIC algorithm when the Signal to Noise Ratio (SNR) is low, the MUSIC algorithm is chosen to be implemented on the reconfigurable architecture.

The MUSIC algorithm is implemented on the Montium2 architecture. The implementation is described by means of pseudo code. Implementation aspects such as, accuracy, computational load, and scalability are analyzed. The complete implementation requires 1.5 million clock cycles on the Montium2. This number of clock cycles results in an execution time of 7.5ms. A practical example of a beamforming system used as a satellite television receiver, mounted on the roof of a car, shows that this is an acceptable execution time in this particular situation.

Acknowledgement

The author hereby wants to thank the members of the Computer Architecture for Embedded Systems group at the Computer Science department of the University of Twente, the graduation committee in particular, for their support and advice during this master assignment. The author also wants to thank Recore Systems for their support with the implementation part of this assignment.

Contents

List of Acronyms

CMD	Covariance Matrix Differencing
CORDIC	COordinate Rotation DIgital Computer
DOA	Direction Of Arrival
dword	double word
DSP	Digital Signal Processing
ESPRIT	Estimation of Signal Parameters by Rotational Invariance Techniques
FPGA	Field Programmable Gate Array
LSB	Least Significant Bit
MSB	Most Significant Bit
MUSIC	MUltiple SIgnal Classification
SNR	Signal to Noise Ratio
SRAM	Static Random Acces Memory
ULA	Uniform Linear Array

Chapter 1

Introduction

A phased array antenna is a direction sensitive antenna, constructed out of a number of smaller antennas. The signals received by the smaller antennas are combined, to increase the SNR of the output signal. Phased array antennas are used in communication systems, sonar and radar applications, space exploration, and many other applications.

The signals received by a phased array antenna are processed in a beamforming system. A digital beamforming system consist of three different parts, a beamformer, a beamsteerer, and parameter estimator. The contents of a beamformer is described in the master's thesis of Rik Portengen [13]. In this master's thesis the contents of a parameter estimator is described. The analysis of the contents of the beamsteerer is the subject of a successive master assignment.

A electronically adaptable beamformers can be used in a mobile, non-stationary environment, such as a satellite television receiver, mounted on the roof of a car. In this case mechanically adaptable beamformers are to slow to keep the phased array antenna focused on the satellite when the car drives into a hard turn. Electronically adaptable beamformers are expected to be fast enough to keep the phased array antenna focused on the satellite.

In chapter 2, the three different parts of a beamforming system are briefly explained. In this thesis the focus is on the parameter estimation part, the DOA estimation in particular. A data model is defined, which is used throughout the thesis. The problem statement is also defined in chapter 2. Two popular DOA estimation algorithms, MUSIC and ESPRIT, are described in chapter 3. In chapter 4, the effects of different test scenarios on the DOA estimation are analyzed by means of a model of MUSIC and a model of ESPRIT. In chapter 5, the implementation of the MUSIC algorithm on the Montium2 architecture is described. Chapter 6 is the last chapter of this thesis. In this chapter conclusions are drawn, and recommendations are made to optimize the implementation.

Chapter 2

Phased array processing

A phased array antenna is an antenna which consists of a number of smaller antennas. These antennas are generally mounted on a flat surface and consequently they are separated by a certain distance. A signal from a certain direction arrives at the antenna array with a certain time shift between the antennas. Signals from different directions arrive at the individual antennas with different time shifts. The largest distance between two elements of a phased array antenna can vary from a few centimeters to several kilometers, depending on the application. The received signals at the array are combined in a beamforming system into a signal which can be used for further processing. Combining the received signals increases the SNR. A phased array antenna can be used in communication systems, sonar and radar applications, space exploration, etc.

2.1 System model

A schematic representation of the digital processing part of a generic beamforming system is shown in figure 2.1 [5]. This system is preceded by a frontend, which transforms the signal received by the different antennas in the antenna array into a snapshot[1] containing complex data samples. (The snapshots do not necessarily need to be complex.) The algorithms treated in this thesis require complex snapshots. The output of this beamforming system is a signal, which for example can be fed into a decoder. The system is separated into three different blocks. Each block has its own functionality and in the next sections that functionality is briefly explained.

Beamformer

In this part of the beamforming system, the actual beam is composed. This means that a snapshot from the antenna array is transformed into a signal. A snapshot containing complex samples is received from the frontend. Each element of this snapshot is delayed in case of wideband signals, or the delay is approximated by a multiplication with a complex weight factor in case of narrowband signals. The time delay or complex multiplication is done to cor-

[1] A snapshot is a vector containing samples of each antenna in the antenna array at one instance of time.

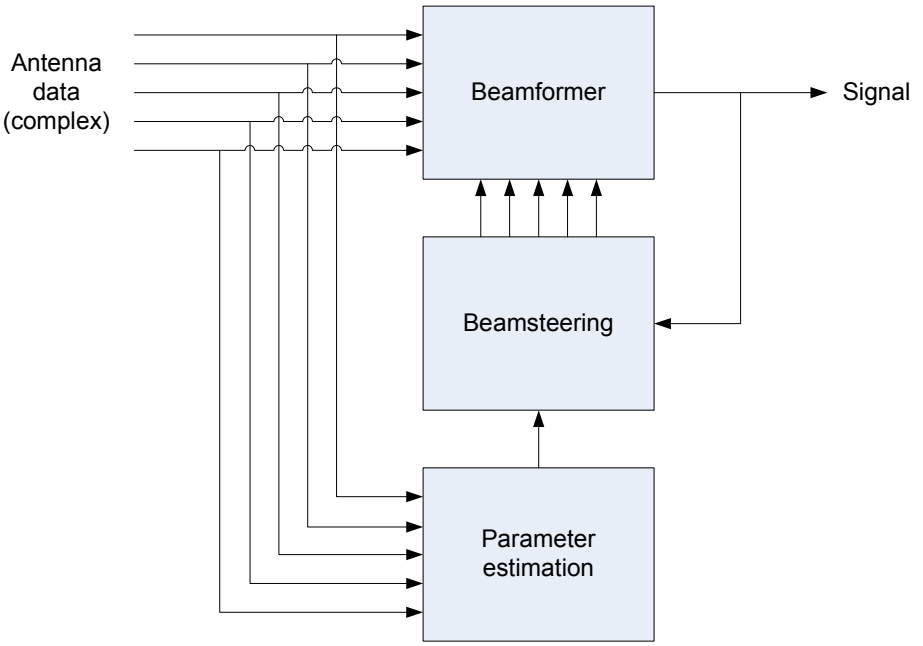


Figure 2.1: Schematic representation of a beamforming system.

rect the time shift of the signal between the antennas, and thus between the different elements of the snapshot. The time delay or weight factors determine in which direction the beam is aimed. After the time delay or complex multiplication all values of the snapshot are summed to form the resulting signal. That signal can be fed into a subsequent system for further processing.

Figure 2.2 shows the sensitivity pattern of a 16 element Uniform Linear Array (ULA), the main beam is directed to 0° with respect to the broadside of the array. Signals impinging from directions other than direction of the main beam are received attenuated. A treatment of beamforming algorithms in more detail is beyond the scope of this thesis, and the reader is therefore referred to [5, 13].

Beamsteering

The beamsteering part of the beamforming system calculates the time delay or weight factor for each antenna, based on a DOA estimation of the parameter estimation part described below. The result of the beamsteering algorithm is used in de beamformer described above. If the DOA of the signal received by the antennas is stationary, the beamsteering algorithm is executed only once. If the DOA varies in time, the beamsteering algorithm is executed for every update of the time delays or weight factors. The rate of change of the DOA determines the update frequency. In case of rapid changing sources, the time delays or weight factors need to be updated more often.

The feedback of the output signal of the beamformer is used when the beamsteering algorithm can track the signal of a source in a non stationary environment. In this case the beamsteering part contains an algorithm to

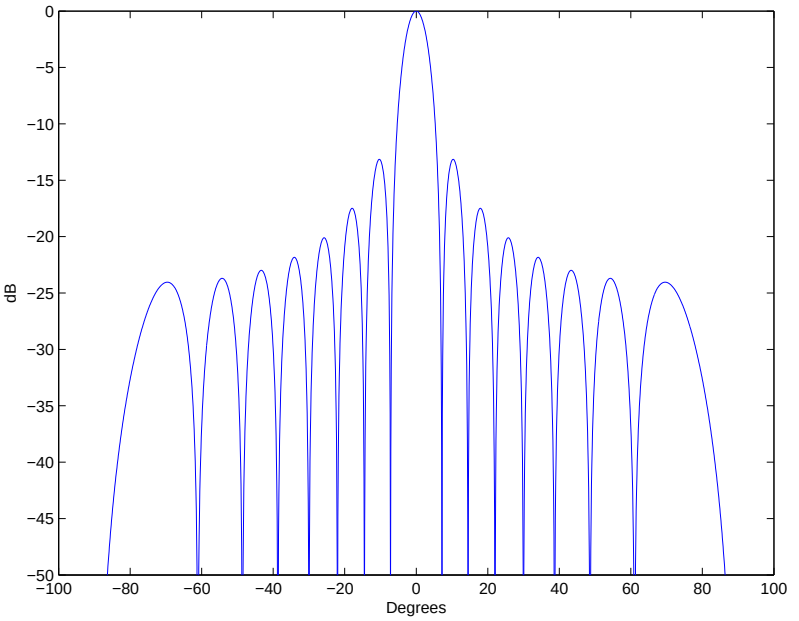


Figure 2.2: Sensitivity pattern of a 16 element ULA.

adapt the time delays or weight factors used in the beamformer. These tracking algorithms often need an initial estimation of the DOA of the signal.

Parameter estimation

In this section of the beamforming system, parameters of the signal are estimated. A number of subsequent snapshots are used to estimate for example the number of sources, the DOAs, the strengths and cross correlations among the sources, polarizations, strength of the noise or the signal frequency. In most cases these parameters can not be computed simultaneously and different algorithms are needed. In this thesis the focus is on DOA estimation. Most DOA estimation algorithms can estimate the DOA of multiple signals simultaneously. In case of multiple signals, one parameter estimation algorithm can provide DOAs to several beamsteering algorithms. Each signal needs its own beamsteering algorithm and probable its own beamsteering algorithm.

2.2 Data model

A data model is a mathematical representation of the data received by the antenna array. The data model is based on a ULA, however, other shapes are also possible. A ULA is an antenna array with identical antennas, arranged in a straight line. The distance between the antennas is equal and at most $\lambda/2$, where λ is the wavelength of the center frequency of the signals. The relation between the distance d and λ is described by $d = \lambda/2$.

A schematic representation of a signal impinging at the antenna array is shown in figure 2.3. To keep the figure simple only one signal is represented.

The angle θ is the angle of the source with respect to a vector which is orthogonal to the array. The distance between two antennas is d, and λ is the wavelength of the signal. ϕ is the phase shift of the signal between two antennas. This phase shift is calculated by:

$$\phi = 2\pi(d/\lambda)\sin\theta \tag{2.1}$$

Consider a n-element array and k sources. The data model is defined by:

$$\begin{pmatrix} x_1(t) \\ x_2(t) \\ \vdots \\ x_{n-1}(t) \\ x_n(t) \end{pmatrix} = \begin{pmatrix} & & & \\ a(\theta_1) & a(\theta_2) & \cdots & a(\theta_k) \\ & & & \end{pmatrix} \begin{pmatrix} s_1(t) \\ s_2(t) \\ \vdots \\ s_k(t) \end{pmatrix} + \begin{pmatrix} n_1(t) \\ n_2(t) \\ \vdots \\ n_{n-1}(t) \\ n_n(t) \end{pmatrix} \tag{2.2}$$

where x_n is the signal received at element n of the antenna array, $a(\theta_k)$ is a steering vector of source k, s_k the signal of source k and n_n the additional noise introduced at element n. The steering vectors are defined by:

$$a(\theta_k) = \begin{pmatrix} 1 \\ e^{j2\pi(d/\lambda)\sin\theta_k} \\ \vdots \\ e^{j2\pi(d/\lambda)(n-2)\sin\theta_k} \\ e^{j2\pi(d/\lambda)(n-1)\sin\theta_k} \end{pmatrix} \tag{2.3}$$

Equation 2.3 describes the phase shift of a signal of one source between the antennas of the antenna array. Equation 2.2 can also be written as:

$$X(t) = AS(t) + N(t) \tag{2.4}$$

where X is the n-element snapshot, A is the set of steering vectors of the k sources, S are the signals of the k sources, and N is the additional noise [16], [20]. This noise is collected by the signal or generated internally in the instrumentation.

To validate the data model of equation 2.2, some assumptions need to be made.

1. The center frequency of the signals or the distance between the antennas is known. When the center frequency of the signals is known, the distance between the different antennas of the antenna array can be determined. When the distance between the antennas is known, the center frequency of the signals can be determined.

2. The signals are narrow band. Signals are narrow band if the energy of the signal is located at, or close to the center frequency of the signal. If the energy of the signal is located at another frequency, an error arises in the DOA estimation. If the deviation of the energy of the signal with respect to the center frequency increases, the error in the DOA estimation increases.

3. The number of sources is known. The number of sources has to be known to determine the size of matrix A, in combination with the number of antennas.

4. The sources are located in the far field of the array, such that the signals are impinging at the array as plane waves. If the narrow band signals are not impinging as plane waves, the shifted reception of the signals at the antennas is not corrected completely. This results in a smaller SNR of the output signal of the beamformer.

5. During the execution of the algorithm, the DOAs of the sources are stationary (wide-sense stationarity), or at least do not change more than the spatial resolution of the DOA algorithm.

6. The noise is white, and uncorrelated between the antennas [12, 15].

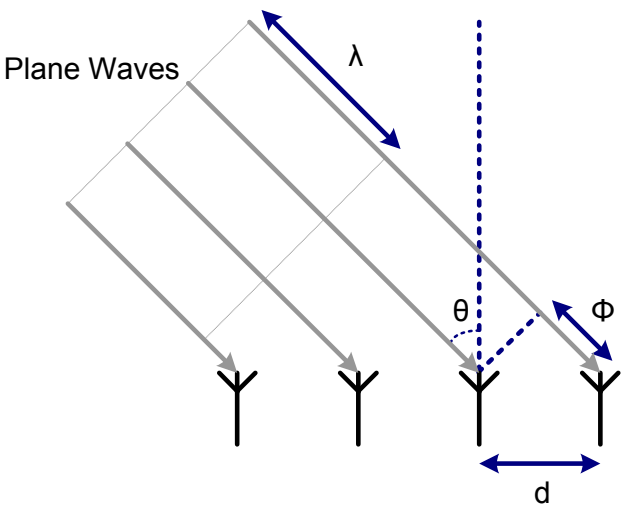


Figure 2.3: Phased array antenna model.

2.3 Processing architecture

A multiprocessor architecture can be used to executed the different sections of the beamforming system of figure 2.1. In the CRISP project [18] such a reconfigurable architecture is developed. This architecture can be used for a wide range of (streaming) applications; from low-cost consumer applications to very demanding high performance applications.

One or more processors can be assigned to each part of the beamforming system. The beamformer has to process each snapshot to produce a continuing stream of samples of the output signal. Therefore, in most cases one or more processors are dedicated to perform this task.

The other two parts of the beamforming system can be executed less often, since the update frequency of the time delays or weight factors is determined by the rate of change of the DOA of the sources. In case of a tracking algorithm is used in the beamsteering part, it is even possible that the DOA estimation algorithm is executed only once.

Because of the lower execution rate, it is expensive to permanently reserve a processor for the beamsteering or DOA estimation algorithm. Reconfigurable

hardware can reduce this cost by offering the possibility of an interleaved execution with an other part of the beamforming system.

2.4 Problem statement

The main assignment of this thesis is to search and investigate the implementation aspects on reconfigurable hardware of current, commonly used algorithms for DOA estimation in beamforming systems. These algorithms are used for the parameter estimation part of the beamforming system in figure 2.1. An implementation of one algorithm has to be made on reconfigurable hardware to investigate performance and scalability.

If the DOAs of the signals with respect to the antenna array are unknown, the contents of matrix A in equation 2.4 is unknown. The objective of the DOA estimation algorithms investigated in this thesis, is to retrieve the contents of matrix A.

A set of subsequent snapshots is used as an input for the DOA estimation algorithms. The algorithms perform a number of operations on these snapshots to estimate the different angles of the sources relative to the antenna array. The kernels of these operations have to be identified.

It is expected that the DOA estimation algorithm is executed only in the startup phase of the beamforming system, or with long intervals between two successive executions. The execution time of the implementation of the chosen DOA estimation algorithm should allow an interleaved execution with an other part of the beamforming system.

2.5 Related work

A few implementations of DOA estimation algorithms are listed below.

FPGA based MUSIC estimator

In [6], the MUSIC algorithm is implemented on 1 Xilinx Virtex-II Pro. MUSIC is a DOA estimation algorithm (which will be explained later). The implementation is based on a 4 element ULA. A Field Programmable Gate Array (FPGA) is chosen to reduce the execution time of the algorithm. The total execution time of the MUSIC algorithm on the FPGA is 30μs.

FPGA based unitary MUSIC estimator

In [10], the unitary MUSIC algorithm is implemented on 2 FPGAs (EP20K600, Altera). Unitary MUSIC is a variant of the MUSIC algorithm mentioned above. This implementation is also based on a 4 element ULA. The execution time of the complete algorithm is 28μs.

DSP based DOA estimator

In [29], a DOA estimation algorithm is implemented on four Digital Signal Processings (DSPs). The implementation is based on a 8 element circular array. The four DSPs execute the complete algorithm in 187μs.

Chapter 3

Methods for DOA estimation

In this chapter, two popular methods are described which can be used to estimate the DOAs of multiple signals. A comparison between several beamforming algorithms has been made in [14,16]. The conclusions showed that MUSIC and ESPRIT are useful beamforming algorithms. The first method described below is MUSIC, and the second method is ESPRIT.

3.1 MUSIC

MUSIC [16] is one of the first DOA algorithms which can be used to estimate the DOA with a high spatial resolution. The basic idea of MUSIC is that the eigenvalues and eigenvectors of a signal covariance matrix are used to estimate the DOAs of multiple signals received by the antenna array.

3.1.1 Basic algorithm

In figure 3.1 the different steps of the algorithm are shown. The algorithm starts with the composition of a covariance matrix. A covariance matrix is calculated by multiplying a snapshot and the Hermitian adjoint[1] of that snapshot. To reduce the disturbing effect of the noise, the expected value is taken over a number of multiplied snapshots. A covariance matrix is calculated by:

$$R_{xx} = \frac{1}{m} \sum_{i=1}^{m} X_i X_i^H \tag{3.1}$$

where R_{xx} is the n by n covariance matrix, X_i is the ith snapshot containing n elements, and m is the number of snapshots.

The next step is the eigendecomposition of the covariance matrix. An eigendecomposition is used to determine the eigenvalues and eigenvectors of the covariance matrix. See section 3.3.1 for a detailed description of an eigendecomposition algorithm. The k (number of sources) largest eigenvalues and their corresponding eigenvectors are assigned to the sources and the $n-k$ other (smallest) eigenvalues and their corresponding eigenvectors are assigned to the noise (n denotes the number of antennas). The eigenvectors assigned to the

[1] A Hermitian adjoint of a vector or matrix is the complex conjugate transpose of that vector or matrix. The Hermitian adjoint is denoted by $()^H$

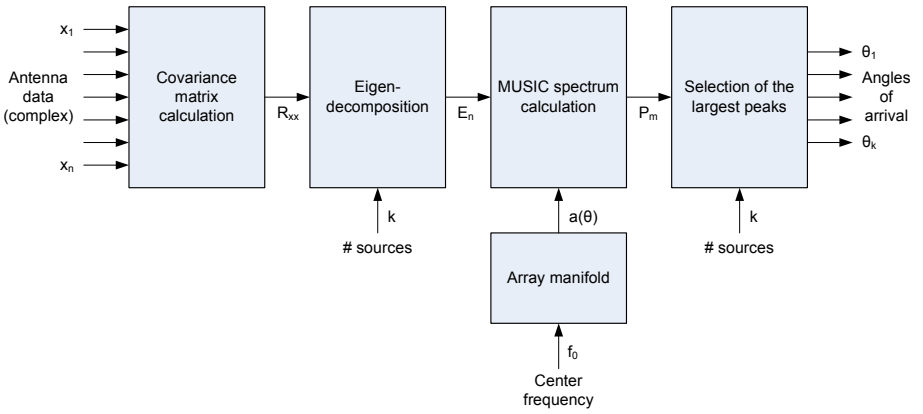


Figure 3.1: Schematic representation of the MUSIC algorithm.

noise are combined in matrix E_n (the noise subspace). Each eigenvector is a column in En.

The number of snapshots used to calculate the covariance matrix determines the accuracy of the eigenvalues and eigenvectors. If an infinite number of snapshots is used, the eigenvalues of the noise converge to the same value, namely the variance of the noise.

If the number of sources is unknown, a threshold value can be used to determine the number of sources. The SNR determines the level of the threshold value. The number of eigenvalues greater than the threshold value determine the number of sources.

After the eigendecomposition, the MUSIC spectrum is calculated. This is done by:

$$P_m(\theta) = \frac{1}{a^H(\theta) E_n E_n^H a(\theta)} \tag{3.2}$$

where $P_m(\theta)$ is the measure for the MUSIC spectrum, E_n is the noise subspace and $a(\theta)$ is a steering vector of the array manifold. The array manifold is a collection of predefined steering vectors. Every steering vector represents a angle in the MUSIC spectrum. The steering vectors of the array manifold are defined the same as a steering vector of a source (see equation 2.3). $P_m(\theta)$ is calculated for every vector in the array manifold. In figure 3.2 an example of a MUSIC spectrum is shown. The maximal spatial resolution of the MUSIC algorithm is determined by the angle between two adjacent steering vectors in the array manifold. If the area of interest is smaller than the whole spectrum, all vectors other than the area of interest can be removed from the array manifold, and therefore reduce the number of computations.

A source is located at the angle where the steering vector of the array manifold is (almost) orthogonal to the noise subspace. The denominator of equation 3.2 is an inner product. When a steering vector is orthogonal to the noise subspace, the result of the inner product is zero. Therefore, the locations of the sources appear as peaks in the MUSIC spectrum. So the last step in the MUSIC algorithm is the selection of the k largest peaks in the MUSIC spectrum. Figure 3.2 shows a MUSIC spectrum with five sources located at -30, -8, 0, 3 and 60 degrees.

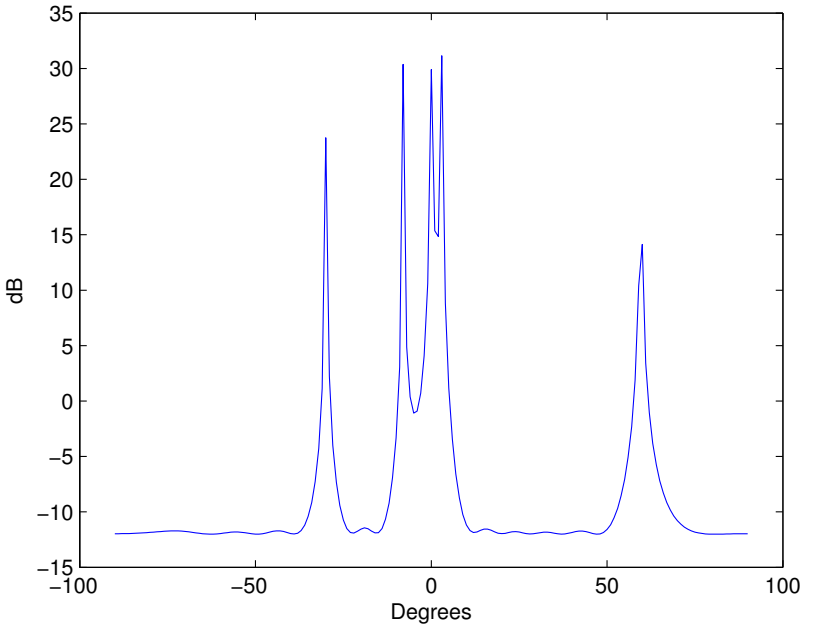


Figure 3.2: The MUSIC spectrum.

3.1.2 CMD extension

Coherent or fully correlated signals have a destructive effect on the result of the MUSIC algorithm. If, for example, two received signals are coherent sinusoids, the frequency of both signals is equal. The signals received by one antenna of the antenna array can be written as:

$$x_n = a_1 \sin(2\pi f_c + \theta_1) + a_2 \sin(2\pi f_c + \theta_2) = a_3 \sin(2\pi f_c + \theta_3) \tag{3.3}$$

where x_n is the signal at the nth element of the array, a is the amplitude of a signal, f_c the center frequency of a signal, and θ the phase shift of a signal. The two signals merge into one signal, and as a consequence, the DOA of the resulting signal is estimated instead of the DOAs of the two original signals.

CMD [20] is an extension to MUSIC to eliminate the received noise, and to allow MUSIC to work on coherent signals at the cost of extra elements in the antenna array. The CMD extension replaces the covariance matrix calculation of the standard MUSIC algorithm. If CMD is used, the number of snapshots and the SNR can be reduced and still achieve the same result as MUSIC without the CMD extension. Figure 3.3 shows that the covariance matrix calculation of the MUSIC algorithm is replaced bij the CMD extension. In figure 3.4 a schematic representation of the CMD extension is shown.

CMD uses L maximally overlapping subarrays of n elements, where n is equal to the number of elements used in the original version of the MUSIC algorithm. Two antenna arrays maximally overlap if they make use of the same elements except for the first element of the first array and the last element of

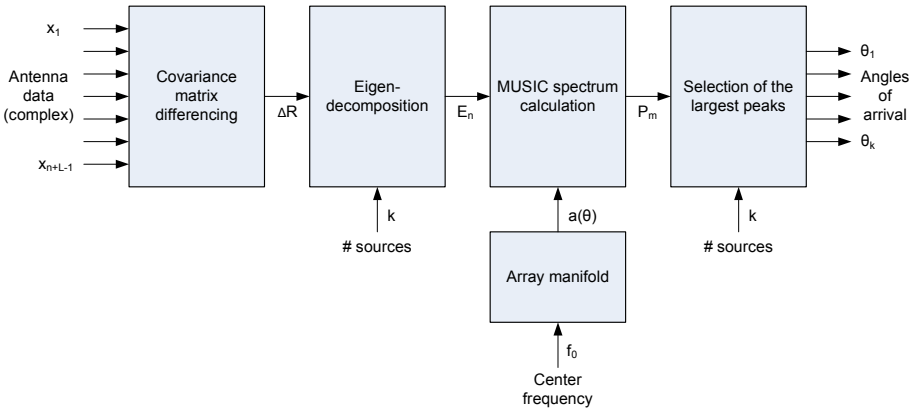


Figure 3.3: Schematic representation of MUSIC including the CMD extension.

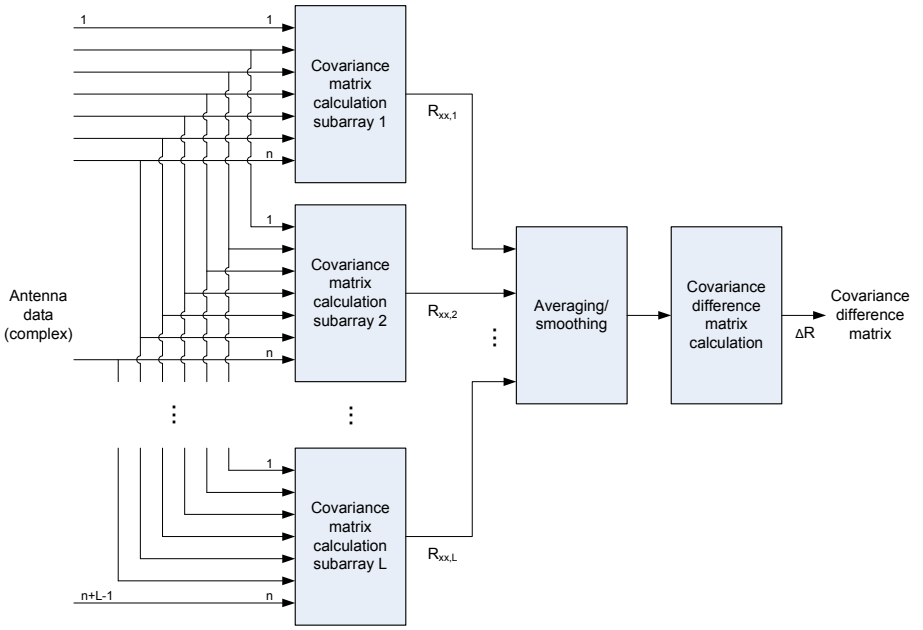


Figure 3.4: Schematic representation of the CMD extension.

the second array (see figure 3.5). Due to the maximal overlap, the L subarrays form one ULA of $n + L - 1$ elements.

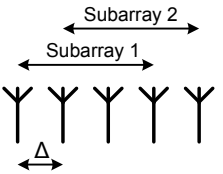


Figure 3.5: Two maximally overlapping antenna arrays.

A covariance matrix is calculated over each of the L subarrays using equation 3.1. Due to the maximal overlap of the subarrays, $n - 1$ elements are shared by two adjacent subarrays. Therefore $(n-1)^2 = n^2 - 2n + 1$ elements are shared by both covariance matrices of two adjacent subarrays. So, for every additional subarray, $n^2 - (n^2 - 2n + 1) = 2n - 1$ new elements of the additional covariance matrix have to be calculated.

An average of the L covariance matrices is calculated by:

$$R = \frac{1}{L} \sum_{l=1}^{L} R_{xx,l} \tag{3.4}$$

The extra covariance matrices introduce additional information about the DOA of the signals. This information is used to retrieve the DOAs.

After the averaging, the covariance difference matrix is calculated. The covariance difference matrix is used in the MUSIC algorithm instead of the covariance matrix. The calculation of the covariance difference matrix (ΔR) is not discussed here, because a detailed explanation of this calculation is beyond the scope of this thesis. The CMD extension is only mentioned as a way to improve the result of the MUSIC algortihm. See [20] for a complete explanation of the CMD algorithm.

3.2 ESPRIT

The second DOA algorithm described in this thesis is ESPRIT [14, 15]. It is also a high resolution DOA algorithm. In contrast to MUSIC, ESPRIT does not need an array manifold. It uses the property of a time shifted reception of a signal by two identical subarrays. The two subarrays are separated by a known distance (displacement vector Δ), and therefore the DOA of that signal can be determined. The displacement vector Δ can have arbitrary length.

If the different elements of the two subarrays are all completely identical, and the distance between two elements is equal, the two subarrays can maximally overlap to reduce the number of antennas (see figure 3.5). In case of two maximally overlapping arrays, the displacement vector Δ is equal to the distance between two adjacent elements.

In figure 3.6 the different steps of the algorithm are shown. The algorithm starts with the composition of a autocovariance matrix and a crosscovariance matrix. The calculation of the autocovariance matrix is equal to equation 3.1,

this equation is repeated below.

$$R_{xx} = \frac{1}{m}\sum_{i=1}^{m} X_i X_i^H = AR_sA^H + \sigma^2 I \tag{3.5}$$

R_{xx} is the n by n autocovariance matrix, X_i is the ith snapshot of the first subarray containing n elements, and m is the number of snapshots. The last part of the equation is the model of the autocovariance matrix used by ESPRIT. A are the steering vectors of the k sources (so the size of A is n by k), R_s is the k by k signal covariance matrix, σ^2 is the additional noise, and I is an n by n identity matrix. R_s is a diagonal matrix if the signals are uncorrelated.

The crosscovariance matrix is calculated by:

$$R_{xy} = \frac{1}{m}\sum_{i=1}^{m} X_i Y_i^H = AR_s\Phi^H A^H + \sigma^2 Z \tag{3.6}$$

where R_{xy} is the n by n crosscovariance matrix, X_i is the ith snapshot of the first subarray (n elements), Y_i is the ith snapshot of the second subarray (also n elements), and m is the number of snapshots. The last part of the equation is the model of the crosscovariance matrix used by ESPRIT. A are the steering vectors of the k sources (A is a n by k matrix), R_s is the n by n signal covariance matrix, Φ is a diagonal k by k matrix containing the phase shifts of the k signals between the two subarrays, σ^2 is the additional noise, and Z is a n by n matrix with ones on the first subdiagonal and zeros elsewhere (a delay operator). A, R_s, Φ, and σ in equations 3.5 and 3.6 are assumed to be unknown. The goal of ESPRIT is to retrieve the contents of Φ.

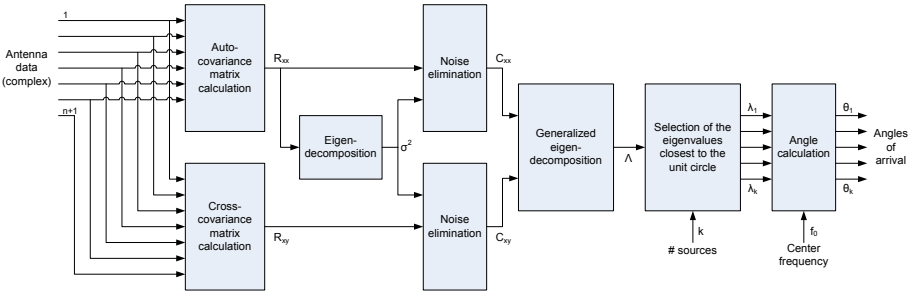


Figure 3.6: Schematic representation of the ESPRIT algorithm.

The next step is the eigendecomposition of the autocovariance matrix. The smallest eigenvalue (λ_{min}) of the resulting eigenvalues of the eigendecomposition is assumed to be de variance of the noise. $\lambda_{min} = \sigma^2$ in equations 3.5 and 3.6. Because the smallest eigenvalue is assigned to the noise, the maximum number of sources is $n - 1$. See section 3.3.1 for a detailed description of an eigendecomposition algorithm.

After the eigendecomposition, σ^2 is used to eliminate the noise out of the autocovariance matrix and the crosscovariance matrix. The noise is eliminated by:

$$C_{xx} = R_{xx} - \sigma^2 I = AR_sA^H \tag{3.7}$$

in case of the autocovariance matrix and

$$C_{xy} = R_{xy} - \sigma^2 Z = AR_s\Phi^H A^H \tag{3.8}$$

in case of the crosscovariance matrix.

The next step is the generalized eigendecomposition of the autocovariance and the crosscovariance matrices. The result of the generalized eigendecomposition is a vector Λ, containing n generalized eigenvalues. The contents of Λ are estimates of the phase shifts of the signals between two subarrays, and they are located on or close to the unit circle. See section 3.3.2 for an explanation of the generalized eigendecomposition.

After the generalized eigendecomposition, the k generalized eigenvalues which are closest to the unit circle are selected. The angles of these k generalized eigenvalues determine the phase shifts of matrix Φ.

When matrix Φ and the wavelength at the center frequency of the signals are known, the DOAs of the signals can be calculated by:

$$\theta_k = \sin^{-1}\left(\frac{\phi_k \lambda}{2\pi\Delta}\right) \tag{3.9}$$

where θ_k is the DOA of the kth signal, ϕ_k is the kth element of the diagonal of Φ, λ is the wavelength of the signal, and Δ is the distance between the two subarrays. With this last step the algorithm has finished.

3.3 Eigenproblems

The DOA estimation in MUSIC and ESPRIT is based on an accurate estimation of the eigenvalues and eigenvectors of a covariance matrix. In MUSIC the eigendecomposition of the covariance matrix is calculated. In ESPRIT the eigendecomposition of the autocovariance matrix and a generalized eigendecomposition of the autocovariance matrix in combination with the crosscovariance matrix are calculated.

3.3.1 Eigendecomposition

An eigendecomposition is used to calculate the eigenvalues and eigenvectors of a square matrix. The relation between a square matrix A, an eigenvalue λ and its corresponding eigenvector v is, by definition [9, 22], described by:

$$Av = \lambda v \tag{3.10}$$

The analytical method for calculating the eigenvalues [9] is by solving the characteristic equation:

$$det\,(A - \lambda I) = 0 \tag{3.11}$$

where A is a square matrix, λ are the eigenvalues of A and I is the identity matrix. The problem with this analytical method is that there is no formula or finite algorithm to solve the characteristic equation of a n by n matrix in case $n \geq 5$ [9]. A numerical method can be used to overcome this problem.

A few examples of numerical algorithms are:

- Power Method [9].

- Jacobi Method [17, 24].
- QR algorithm [11].

The Power Method will find only one eigenvalue and its corresponding eigenvector, the eigenvalue with the greatest absolute value. The Jacobi Method and QR algorithm are both algorithms which calculate all eigenvalues and eigenvectors simultaneously.

The Power Method is not useful as an eigendecomposition algorithm for MUSIC or ESPRIT, since it finds only one eigenvalue and eigenvector. The Jacobi Method and QR algorithm are similar algorithms, and both useful as an eigendecomposition algorithm for MUSIC and ESPRIT. In [1, 17, 27] is stated that the QR algorithm is the standard method for computing the eigenvalues of a general dense matrix, because of the faster convergence of the algorithm. Therefore the QR algorithm is chosen as the algorithm to compute the eigenvalues and eigenvectors.

QR algorithm

The QR algorithm starts with the decomposition of matrix A into a Q and a R matrix.

$$A = A_0 = Q_0 R_0 \tag{3.12}$$

where Q is a orthogonal matrix[2] and R is an upper triangular matrix. The next step is to calculate a new A matrix by multiplying the Q_0 and the R_0 matrices in the reverse order.

$$A_1 = R_0 Q_0 \tag{3.13}$$

During the repetition of these two steps, the values on the diagonal of matrix A converge. The converged values on the diagonal of matrix A represent the eigenvalues of matrix A.

The calculation of the eigenvectors of matrix A is done by multiplying all Q matrices calculated during the calculation of the eigenvalues.

$$S_0 = Q_0 \tag{3.14}$$

$$S_1 = S_0 Q_1 \tag{3.15}$$

The last multiplication is repeated for every Q matrix. After multiplication of all Q matrices, the columns of matrix S represent the eigenvectors of matrix A.

The complete algorithm can be written as:

$$\begin{gathered} A = Q_0 R_0, \quad S_0 = Q_0 \\ A_{k+1} = R_k Q_k = Q_{k+1} R_{k+1}, \quad S_k = S_{k-1} Q_k, \quad k = 0, 1, 2, \ldots \end{gathered} \tag{3.16}$$

The QR algorithm in its original form will converge to the correct eigenvalues only if the eigenvalues of matrix A are real. The covariance matrices used in MUSIC and ESPRIT are Hermitian[3]. One of the properties of a hermitian

[2] A square matrix Q is called an orthogonal matrix if it satisfies $Q^T Q = I$.

[3] In a Hermitian matrix the element on the ith row and jth column is the complex conjugate of the element on the jth row and ith column. Therefore $X = X^H$.

matrix is that it has real eigenvalues, so the QR algorithm can be used without any extensions.

The columns of matrix S converge to the eigenvectors only if matrix A is hermitian and positive definite (i.e. matrix A has positive eigenvalues). If matrix A is not hermitian or not positive definite, the Inverse Power Method [9] can be used to calculate the eigenvectors out of the found eigenvalues.

QR decomposition

A QR decomposition based on complex Givens rotations is used to calculate the Q and the R matrices. Givens rotations can rotate a vector by a certain angle and are the core operations in the QR decomposition discussed here. Givens rotations can introduce zeros in a vector or a matrix. This property is used in the QR decomposition. Parts of the text and equations written below are comparable to [7].

A real Givens rotation is given by:

$$\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix} \tag{3.17}$$

where $\theta = \arctan\left(\frac{b}{a}\right)$, a and b are real values and $r = \sqrt{a^2 + b^2}$. The first matrix in equation 3.17 is the rotation matrix.

A complex Givens rotation is given by:

$$\begin{pmatrix} \cos\theta_1 & (\sin\theta_1)e^{i\theta_2} \\ (-\sin\theta_1)e^{-i\theta_2} & \cos\theta_1 \end{pmatrix} \begin{pmatrix} a_r + ia_i \\ b_r + ib_i \end{pmatrix} = \begin{pmatrix} r_r + ir_i \\ 0 \end{pmatrix} \tag{3.18}$$

where a_r and a_i together form a complex value, this also holds for b_r, b_i and r_r, r_i. The complex rotation matrix is derived from the real rotation matrix by applying a unitary transformation:

$$U = \begin{pmatrix} e^{-i\theta_a} & 0 \\ 0 & e^{-i\theta_b} \end{pmatrix} \tag{3.19}$$

To reduce the number of complex multiplications, the complex conjugate of the unitary transformation above is applied to the left side of the real Givens rotation matrix. As a result, the diagonal of the rotation matrix becomes real. The resulting complex Givens rotation matrix is given by:

$$\begin{pmatrix} e^{i\theta_a} & 0 \\ 0 & e^{i\theta_b} \end{pmatrix} \begin{pmatrix} \mathrm{c}\theta_1 & \mathrm{s}\theta_1 \\ -\mathrm{s}\theta_1 & \mathrm{c}\theta_1 \end{pmatrix} \begin{pmatrix} e^{-i\theta_a} & 0 \\ 0 & e^{-i\theta_b} \end{pmatrix} = \begin{pmatrix} \mathrm{c}\theta_1 & (\mathrm{s}\theta_1)e^{i\theta_2} \\ (-\mathrm{s}\theta_1)e^{-i\theta_2} & \mathrm{c}\theta_1 \end{pmatrix} \tag{3.20}$$

where $\mathrm{c}\theta_1 = \cos\theta_1$ and $\mathrm{s}\theta_1 = \sin\theta_1$.
θ_1 is calculated by:

$$\theta_1 = \arctan\left(\frac{|b|}{|a|}\right) \tag{3.21}$$

θ_2 is calculated by:

$$\theta_2 = \theta_a - \theta_b \tag{3.22}$$

$|a|$ and θ_a are caculated by:

$$\theta_a = \arctan\left(\frac{a_i}{a_r}\right), \quad |a| = \sqrt{a_r^2 + a_i^2} \tag{3.23}$$

and $|b|$ and θ_b are caculated by:

$$\theta_b = \arctan\left(\frac{b_i}{b_r}\right), \quad |b| = \sqrt{b_r^2 + b_i^2} \tag{3.24}$$

The decomposition of a matrix A in a Q and a R matrix [26] is explained by an example of a 3 by 3 matrix. Let matrix A be defined by:

$$A = \begin{pmatrix} |r|e^{i\theta_r} & |s|e^{i\theta_s} & |t|e^{i\theta_t} \\ |u|e^{i\theta_u} & |v|e^{i\theta_v} & |w|e^{i\theta_w} \\ |x|e^{i\theta_x} & |y|e^{i\theta_y} & |z|e^{i\theta_z} \end{pmatrix} \tag{3.25}$$

The first step of the decomposition is zeroing element $|x|e^{i\theta_x}$. Rotation matrix $Q_{3,1}$ is composed to rotate this element.

$$Q_{3,1} = \begin{pmatrix} \cos\theta_1^{3,1} & 0 & (\sin\theta_1^{3,1})e^{i\theta_2^{3,1}} \\ 0 & 1 & 0 \\ (-\sin\theta_1^{3,1})e^{-i\theta_2^{3,1}} & 0 & \cos\theta_1^{3,1} \end{pmatrix} \tag{3.26}$$

where $\theta_1^{3,1} = \arctan\left(\frac{|x|}{|r|}\right)$, and $\theta_2^{3,1} = \theta_r - \theta_x$. Matrices $Q_{3,1}$ and A are multiplied to calculate matrix A'.

$$A' = Q_{3,1}A = \begin{pmatrix} |r'|e^{i\theta_r}{}' & |s'|e^{i\theta_s}{}' & |t'|e^{i\theta_t}{}' \\ |u|e^{i\theta_u} & |v|e^{i\theta_v} & |w|e^{i\theta_w} \\ 0 & |y'|e^{i\theta_y}{}' & |z'|e^{i\theta_z}{}' \end{pmatrix} \tag{3.27}$$

The second step of the decomposition is zeroing element $|u|e^{i\theta_u}$. A second rotation matrix ($Q_{2,1}$) is composed to rotate this element.

$$Q_{2,1} = \begin{pmatrix} \cos\theta_1^{2,1} & (\sin\theta_1^{2,1})e^{i\theta_2^{2,1}} & 0 \\ (-\sin\theta_1^{2,1})e^{-i\theta_2^{2,1}} & \cos\theta_1^{2,1} & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.28}$$

where $\theta_1^{2,1} = \arctan\left(\frac{|u|}{|r'|}\right)$, and $\theta_2^{2,1} = \theta_r' - \theta_u$. Matrices $Q_{2,1}$ and A' are multiplied to calculate matrix A''.

$$A'' = Q_{2,1}A' = \begin{pmatrix} |r''|e^{i\theta_r}{}'' & |s''|e^{i\theta_s}{}'' & |t''|e^{i\theta_t}{}'' \\ 0 & |v'|e^{i\theta_v}{}' & |w'|e^{i\theta_w}{}' \\ 0 & |y'|e^{i\theta_y}{}' & |z'|e^{i\theta_z}{}' \end{pmatrix} \tag{3.29}$$

The third step of the decomposition is zeroing element $|y'|e^{i\theta_y}{}'$. A third rotation matrix ($Q_{3,2}$) is composed to rotate this element.

$$Q_{3,2} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_1^{3,2} & (\sin\theta_1^{3,2})e^{i\theta_2^{3,2}} \\ 0 & (-\sin\theta_1^{3,2})e^{-i\theta_2^{3,2}} & \cos\theta_1^{3,2} \end{pmatrix} \tag{3.30}$$

where $\theta_1^{3,2} = \arctan\left(\frac{|y'|}{|v'|}\right)$, and $\theta_2^{3,2} = \theta_v' - \theta_y'$. Matrices $Q_{3,2}$ and A'' are multiplied to calculate matrix A'''.

$$A''' = Q_{3,2}A'' = \begin{pmatrix} |r''|e^{i\theta_r}{}'' & |s''|e^{i\theta_s}{}'' & |t''|e^{i\theta_t}{}'' \\ 0 & |v''|e^{i\theta_v}{}'' & |w''|e^{i\theta_w}{}'' \\ 0 & 0 & |z''|e^{i\theta_z}{}'' \end{pmatrix} \tag{3.31}$$

The lower left triangle of matrix A''' is completely zeroed, therefore:

$$R = A''' = Q_{3,2}Q_{2,1}Q_{3,1}A \tag{3.32}$$

The fourth and last step of the QR decomposition is the calculation of the final Q matrix. Matrix Q is calculated by:

$$Q = Q_{3,1}^H Q_{2,1}^H Q_{3,2}^H \tag{3.33}$$

After the calculation of matrix Q, the QR decomposition has finished.

To decompose n by n matrices, the rotation matrix is extrapolated to:

$$Q_{i,j} = \begin{pmatrix} 1 & & & & & & & & & & \\ & \ddots & & & & & & & & & \\ & & 1 & & & & & & & & \\ & & & \cos\theta_1^{i,j} & & & & (\sin\theta_1^{i,j})e^{i\theta_2^{i,j}} & & & \\ & & & & 1 & & & & & & \\ & & & & & \ddots & & & & & \\ & & & & & & 1 & & & & \\ & & & (-\sin\theta_1^{i,j})e^{-i\theta_2^{i,j}} & & & & \cos\theta_1^{i,j} & & & \\ & & & & & & & & 1 & & \\ & & & & & & & & & \ddots & \\ & & & & & & & & & & 1 \end{pmatrix} \tag{3.34}$$

The subscript i and j after Q are the row and column index of the element in matrix A which will be zeroed. The generalized equation to calculate the n by n upper triangular matrix R is given by:

$$R = Q_{n,n-1}Q_{n-1,n-2}Q_{n,n-2}\dots Q_{n-2,1}Q_{n-1,1}Q_{n,1}A = \left(\prod_{j=-n+1}^{-1}\prod_{i=-j+1}^{n} Q_{i,-j}\right)A \tag{3.35}$$

The generalized equation to calculate the n by n final orthogonal matrix Q is given by:

$$Q = Q_{n,1}^H Q_{n-1,1}^H Q_{n-2,1}^H \dots Q_{n,n-2}^H Q_{n-1,n-2}^H Q_{n,n-1}^H = \prod_{j=1}^{n-1}\prod_{i=-n}^{-j-1} Q_{-i,j}^H \tag{3.36}$$

3.3.2 Generalized eigendecomposition

The generalized eigendecomposition is used to calculate the generalized eigenvalues and eigenvectors of two square matrices. The relation between two square matrices A and B, an eigenvalue λ and its eigenvector v is, by definition [22], described by:

$$Av = \lambda Bv \tag{3.37}$$

If B is invertible, the relation can be rewritten into:

$$B^{-1}Av = \lambda v \tag{3.38}$$

which is the relation of the normal eigendecompostion. In most situations it is preferable not to perform the inversion of B, but to calculate the generalized eigendecomposition. The analytical method for calculating the generalized eigenvalues [22] is by solving the characteristic equation:

$$det\,(A - \lambda B) = 0 \tag{3.39}$$

where A and B are n by n matrices, and λ are the generalized eigenvalues of the matrices A and B. With this analytical method for the generalized eigendecomposition the same problem arises as with the eigendecomposition, there is no formula or finite algorithm to solve the characteristic equation of n by n matrices where $n \geq 5$ [9]. The solution to this problem is also to use a numerical method. The QZ algorithm [4, 8] is an example of such a numerical method which is commonly used. In the next chapter it will become clear why the QZ algorithm is not treated in detail.

3.4 Comparison of MUSIC and ESPRIT

Both algorithms make use of an eigendecompostion. One iteration of the eigendecomposition has a computational complexity of $O(n^3)$ [28], where n is the number of antennas, this step has greatest computational complexity in both algorithms. The number of iterations depend on the accuracy of the eigenvalues. It is not unlikely that the number of iterations can grow to n or more, and therefore increase the order of complexity of the eigendecomposition algorithm.

In this thesis a 1-dimensional ULA is used for both algorithms. If the dimensions of the antenna array expand to 2 or 3 dimensions, the computational load of both algorithms grows linear with the dimensions, since every dimension can be seen as a 1-dimensional array. In every dimension k angles of interest are selected, therefore the angles of interest grow exponentially with the dimensions. To find the DOAs of the k sources, all angles of interest have to be examined.

The model used in ESPRIT is based on a situation where the signals of the sources are uncorrelated. In this case the signal covariance matrix R_s is equal to the identity matrix I. In most practical cases there is at least some correlation between the sources, so R_s is not strictly equal to I. Therefore, the noise is not completely eliminated out of the autocovariance and crosscovariance matrices. Due to the difference between the noise model of ESPRIT and the practical situation, the DOA estimations become less accurate. In contrast to ESPRIT, MUSIC needs a certain level of noise, because MUSIC has to assign a noise subspace.

The last difference between MUSIC and ESPRIT mentioned in the thesis is the memory used by MUSIC to store the array manifold. The size of this memory depends on the resolution of the search on 1 dimension of the array. ESPRIT does not have to store any information about the antenna array. On the other hand, ESPRIT requires two identical subarrays.

In the next chapter some practical situations are simulated in MUSIC and ESPRIT. The results of the simulations are used to compare the accuracy of both algorithms.

Chapter 4

Modeling and simulations

A model of MUSIC and a model of ESPRIT are made in Matlab to analyse the performance, and the effects of different test scenarios on the DOA estimation. MUSIC and ESPRIT are tested in the first set of simulations. MUSIC with the CMD extension is tested in the second set of simulations.

4.1 MUSIC and ESPRIT simulations

To analyse the performance of MUSIC and ESPRIT a test case is defined.

- The antennas are arranged in a ULA with a distance of $\lambda/2$ of the reference frequency between the antennas. The distance $\lambda/2$ is a critical element distance [25]. A smaller distance increases the width of the main beam. A larger distance introduces additional main beams.
- The signals of 5 sinusoid sources are impinging at the antenna array. The number of sources is chosen in combination with the locations of the sources to create a specific sitation.
- The sources are located at angles of respectively -30, -8, 0, 3 and 60 degrees with respect to a vector orthogonal to the antenna array. The ability of the algorithms to detect the sources at -8, 0, and 3 degrees is interesting, because the spatial difference between these sources is small. The sources at -30 and 60 degrees are used to check if the estimation error changes if the angle becomes larger.
- The different sources have a small difference in center frequency. This small difference is needed to prevent full correlation of the signals. The differences in center frequency are by steps of -0.1% of the reference frequency per signal. This step size causes sufficient decorrelation, and the DOA estimation error can be ignored.

The parameters changed between the different simulations are:

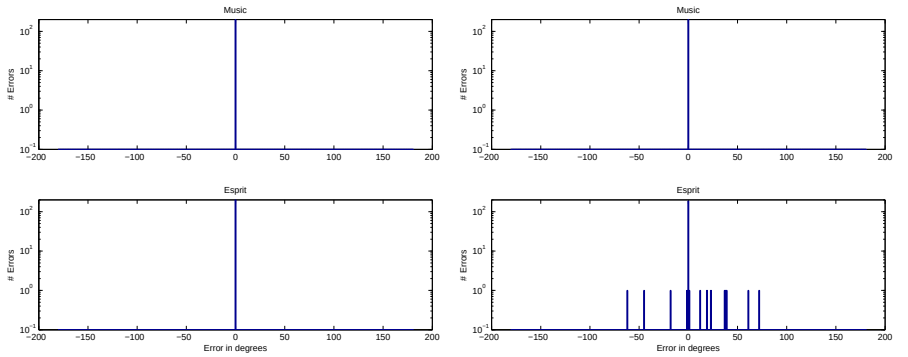
- the SNR (60dB, 40dB and 20dB).
- the number of snapshots (2048, 1024 and 512 snapshots).
- the number of antennas (32, 16 and 8 antennas).

The values of these parameters are chosen after a parameter sweep on the test case. These parameters showed interesting observations. For every configuration of simulation parameters, 40 datasets are determined. Each dataset consists of the signals of the 5 sources added with white noise per signal. The variance of the noise is the same in all 40 datasets to achieve the correct SNR. Eventually, for every configuration of simulation parameters the DOA of 200 sources are estimated.

In ESPRIT two subarrays are used. The size of each subarray is equal to the number of antennas parameter. In this way the sizes of the correlation matrices used in MUSIC and ESPRIT are equal. As a consequence, the total number of antennas used in ESPRIT is one more.

Reducing the SNR

The best simulation results for all algorithms are obtained if the SNR is 60dB (largest value used in the simulations). In figure 4.1a the result of a simulation is shown. In this example the number of antennas used in the array is 16, the number of snapshots is 2048 and the SNR is 60dB. As can been seen in figure 4.1a, all 200 DOA estimations per algorithm have an error of 0 degrees, so in this case all DOAs are correctly estimated. In MUSIC the spectrum is calculated using a resolution of 1 degree. The estimation results of ESPRIT are rounded to the closest integer value, so all estimation errors are always integer values. In section A.1 of the appendix, all results of the simulations are shown.



(a) 60dB (b) 20dB

Figure 4.1: Two simulation results with different SNRs. (2048 snapshots, 16 antennas)

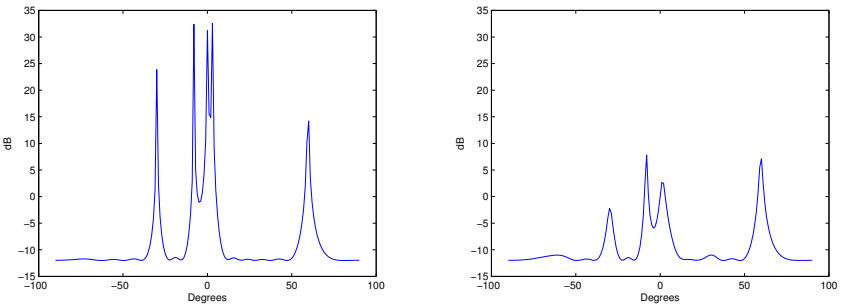
If the SNR is reduced, the difference between the eigenvalues of the signals and the eigenvalues of the noise become smaller. In case of the MUSIC algorithm, the peaks in the MUSIC spectrum (see figure 3.2) become smaller with respect to the noise level. High peaks are desired to distinguish the sources more easily. In case of the ESPRIT algorithm, more (generalized) eigenvalues are situated close to the unit circle. A clear distinction between the eigenvalues of the sources close to the unit circle and the other eigenvalues is desired to

select the eigenvalues of the sources more easily. The variance of the errors of MUSIC due to smaller SNR is less than the variance of the errors of ESPRIT.

Reducing the number of snapshots

The number of snapshots used to estimate the covariance matrix determines the variance of the off-diagonal elements of R_s (equations 3.5, and 3.6), and therefore the estimated correlation between the signals. If less snapshots are used, the signals seem to be more correlated. In MUSIC, the level of correlation determines the size of all the peaks and the distinction of closely situated peaks in the MUSIC spectrum. A low level of correlation between the signals results in high peaks in the spectrum. A high level of correlation result in small peaks in the spectrum. If the signals are fully correlated, no peaks appear in the spectrum, and therefore, no DOA estimation can be done.

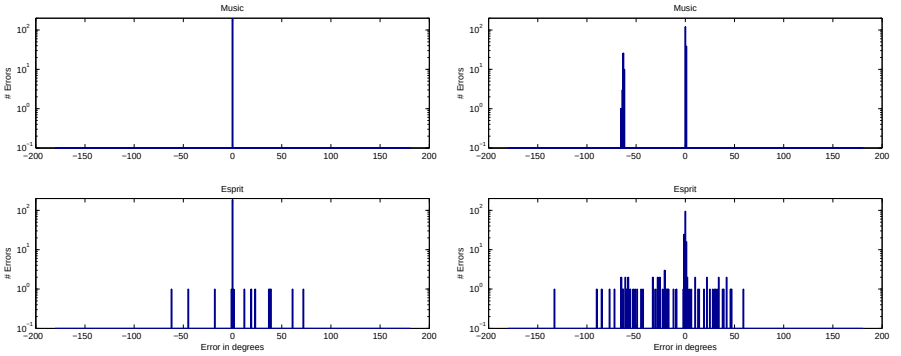
In figure 4.2a the sizes of the peaks have relatively large differences, despite of an equal SNR. Probably this is a consequence of the small differences in center frequencies of the sources.



(a) 2048 snapshots (b) 1024 snapshots

Figure 4.2: The effect of different numbers of snapshots on the MUSIC spectrum. (16 antennas, 20dB SNR)

In ESPRIT, the level of correlation does not affect the detection of closely situated sources, because the DOA of the signals is determined by the angle of the generalized eigenvalues (the angle represents the phase shift of a signal between two antennas, and therefore the DOA). A small difference of the angle of two closely situated generalized eigenvalues is detectable in most cases. The covariance matrices are estimated using a finite number of snapshots, therefore the eigenvalues of these matrices contain an error. The variance of this error decreases linearly with the number of snapshots [2]. Since the DOA are derived out of the eigenvalues, reducing the number of snapshots in ESPRIT does affect the variance of the DOA estimation error. If less snapshots are used, the variance of the DOA estimation error increases, see figure 4.3. The effect of reducing the number of snapshots is less than the effect of reducing the SNR on the variance of the DOA estimation error.



(a) 2048 snapshots (b) 1024 snapshots

Figure 4.3: The result of two simulations with a different number of snapshots. (16 antennas, 20dB SNR)

Reducing the number of antennas

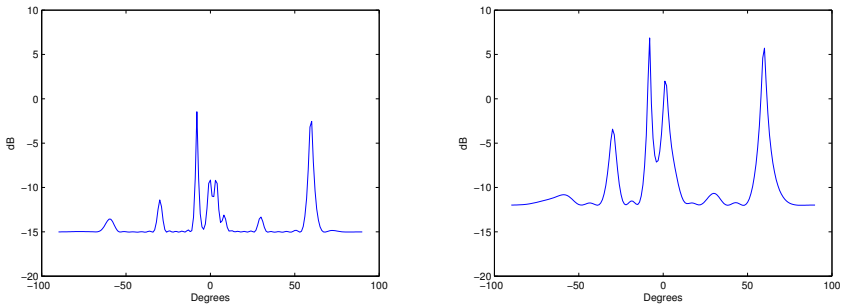
The number of antennas in the antenna array affects the spatial resolution of both DOA algorithms. More antennas result in a higher spatial resolution, and less antennas reduce the spatial resolution. In MUSIC the peaks in the MUSIC spectrum become narrower when more antennas are used, and therefore two adjacent sources may be located with a smaller mutual distance. Figure 4.4 shows the effect of different numbers of antennas on the MUSIC spectrum. In figure 4.4a the small peaks in the noise level located at the mirrored angle of a DOA of a signal exist due to the fact that the implementation of the Hilbert transform in Matlab suffers from the effect of a turn-on and turn-off transient. If more snapshots are used, the peaks become smaller and will disappear eventually. If two peaks merge into one peak, the MUSIC algorithm still has to find 5 peaks. Therefore, one of the small peaks mentioned above is chosen. This results in a wrong estimation of the DOA of source. In figure 4.5b several times the same wrong peak is chosen, which results in the peak at -63 degrees.

In ESPRIT, more antennas also result in a higher spatial resolution. However, the effect of a higher resolution due to more antennas is less noticeable, because two closely situated sources are already detectable, and therefore it only affects the accuracy of the angles of the generalized eigenvalues.

4.2 CMD MUSIC simulations

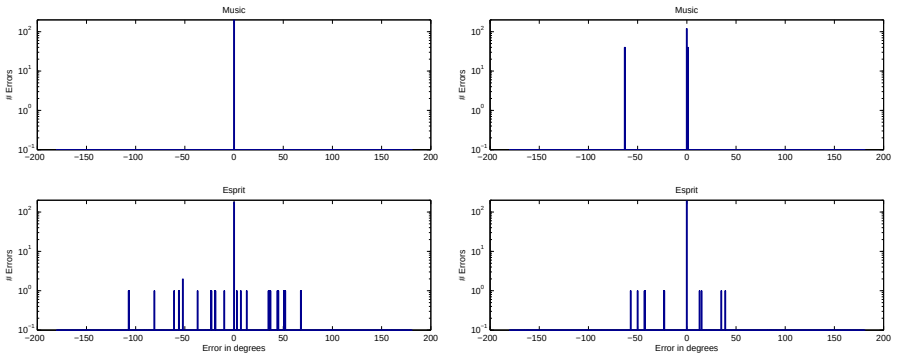
To analyse the performance of MUSIC in combination with the CMD extension, the test case of the previous section is used. The only difference with respect to the test case of the previous section is the center frequency of the sources. In the test case used in this section, all sources have the same center frequency. Due to the equal center frequency, the signals of the sources are fully correlated, because all signals are sinusoids. The parameters changed between the different simulations are:

- the SNR. (20dB, 10dB, 5dB, 0dB and -5dB)



(a) 32 antennas (b) 16 antennas

Figure 4.4: The effect of different numbers of antennas on the MUSIC spectrum. (1024 snapshots, 40dB SNR)



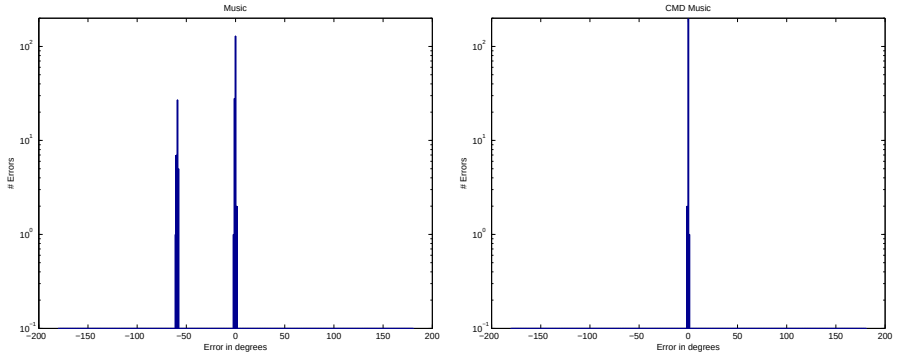
(a) 32 antennas (b) 16 antennas

Figure 4.5: The result of two simulations with a different number of antennas. (1024 snapshots, 40dB SNR)

- the number of snapshots. (512, 256, 128, 64 and 32 snapshots)

These parameters are different from the parameters used in the previous section, because the previous parameters showed almost no differences in the results of the simulations of MUSIC including the CMD extension. The new parameters are chosen after a parameter sweep on the test case using the CMD extension. The number of antennas does not change between the different simulations and is fixed at 16 antennas. This fixed number is chosen to reduce the total number of simulations, and the previous simulations of the MUSIC algorithm showed that 16 antennas is a good choice. Per simulation 200 DOA estimations are done (40 datasets with 5 signals each).

In figure 4.6 the superior performance of MUSIC including the CMD extension over the original version of MUSIC is shown. In figure 4.6a the small peak at an error of -59 degrees arise from the the fact that two closely situated peaks in the MUSIC spectrum merge into one peak if the signals are highly correlated. The MUSIC algorithm used in the simulations must always find the



(a) MUSIC (b) CMD MUSIC

Figure 4.6: The result of two simulations. (512 snapshots and 20dB SNR)

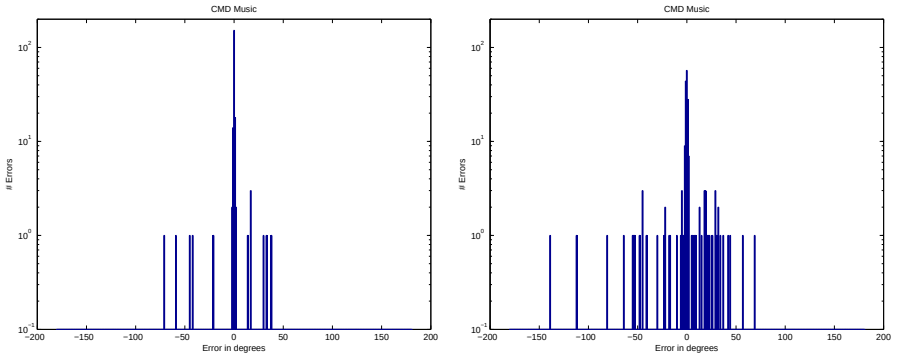
5 largest peaks. As a consequence a smaller peak in the noise level is wrongly assumed to be a DOA of a source. In this example always the same peak with an error of about -59 degrees with respect to the correct peak is chosen, due to the turn-on and turn-off transient of the implementation of the Hilbert filter in Matlab, explained earlier. In figure 4.6b almost all DOA estimations are correct. In section A.2 of the appendix, all results of the simulations are shown.

Reducing the SNR

If the SNR is reduced, the number of incorrect DOA estimations increases. Figure 4.7 shows the effect of reducing the SNR of the signals. In this example 64 snapshots are used to calculate the correlation matrix. The DOA estimation errors are spread more randomly over the different angles. In figure 4.7, two cases of CMD MUSIC are considered. A simulation of the basic algorithm of MUSIC using equal parameters (10dB, 0dB SNR and 64 snapshots) results in arbitrary DOA estimations. Therefore, it is not interesting to compare a simulation of basic MUSIC and CMD MUSIC.

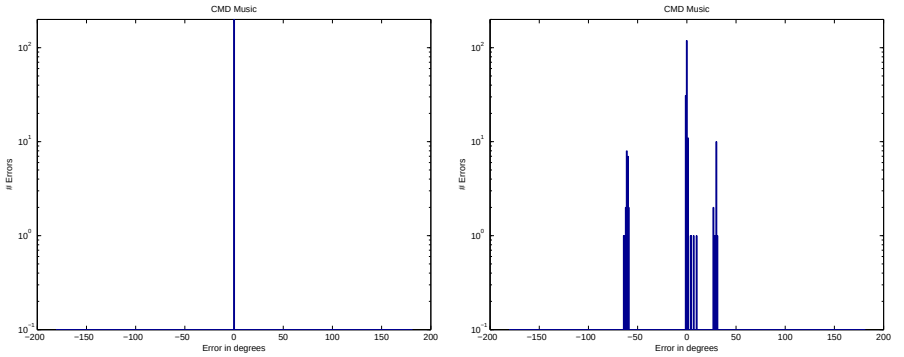
Reducing the number of snapshots

Reducing the number of snapshots in CMD MUSIC has the same effect as reducing the number of snapshots in the original version of MUSIC. Two closely situated peaks in the spectrum merge into one peak, and the sizes of the peaks decrease, when the number of snapshots reduces. In figure 4.8b the small peaks at -61 and 30 degrees are incorrect DOA estimations due to the fact that two peaks are merged and a peak in the noise level is chosen as a DOA. These two small peaks are again a consequence of the imperfect implementation of the Hilbert transform in Matlab.



(a) 10dB SNR (b) 0dB SNR

Figure 4.7: The result of two simulations. (64 snapshots)



(a) 64 snapshots (b) 32 snapshots

Figure 4.8: The result of two simulations. (20dB SNR)

4.3 Conclusion

A model of MUSIC and a model of ESPRIT are simulated in Matlab. A test case as defined in section 4.1 is used to compare both algorithms. For the comparison of both algorithms, only the results of the simulations are considered. The simulations show that MUSIC has a better performance compared to ESPRIT when the SNR is low. Figure 4.9 shows the number of errors of all simulations with 20dB SNR (the lowest SNR used in the simulations). Table 4.1 shows the number of antennas and the number of snapshots used in these simulations.

When less snapshots are used to calculate the covariance matrix, the performance of ESPRIT is superior to the performance of MUSIC (in case of high SNR). Changing the number of antennas does not result in a significant difference in performance between both algorithms.

The simulations as described in section 4.2 show that the performance of MUSIC increases considerably if the CMD extension is used. The impinging signals may be fully correlated. To achieve the same performance, the number

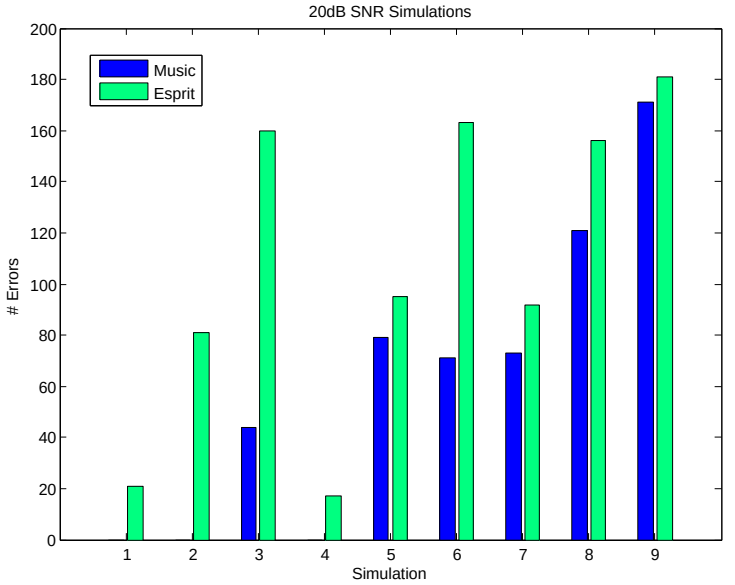


Figure 4.9: Comparison of all simulations with 20dB SNR.

Simulation	**Antennas**	**Snapshots**
1	32	2048
2	32	1024
3	32	512
4	16	2048
5	16	1024
6	16	512
7	8	2048
8	8	1024
9	8	512

Table 4.1: Number of antennas and snapshots used in the simulation.

of snapshots reduces significantly.

Both algorithms, MUSIC and ESPRIT, have comparable computational complexity. The largest complexity in both algorithms is at least $O(n^3)$ in the eigendecomposition. Because the performance of MUSIC is better than the performance of ESPRIT when the SNR is low, the MUSIC algorithm is chosen to be implemented on reconfigurable hardware. Initially the basic algorithm is implemented.

Chapter 5

Algorithm implementation

The MUSIC algorithm is implemented on the Montium2 [19]. The MUSIC algorithm is chosen, because the performance of MUSIC is better compared to ESPRIT if the SNR is low. The Montium2 architecture is chosen, because it is designed for the CRISP project, to process high performance streaming applications such as beamforming. The Montium2 is one of the tiles in the multiprocessor reconfigurable architecture.

5.1 Montium2

The Montium2 architecture consist of 16 functional units, 5 Static Random Acces Memory (SRAM) units (1024 32-bit words per unit), and a number of shared register files. The interconnection between the different parts of the architecture is reconfigurable. Because of the reconfigurable interconnection, the interconnection between all 16 functional units can be determined at design time by the application engineer. It is also possible to use a subset of all functional units in the datapath, depending of the application. See figure 5.1 for a simplified schematic representation of the Montium2 architecture.

The 16 functional units of the Montium2 architecture are divided into seven different categories. The Montium2 has 4 S units (S0 to S3), 2 P units (P0 and P1), 2 M units (M0 and M1), 5 A units (A0 to A4), 1 LC unit (LC), 1 EI unit (EI) and 1 EO unit (EO). The S and P units can perform general DSP operations (e.g. addition, subtraction, etc). The S units can, additionally, perform shift operations. The P units can, additionally, perform compare and pack operations. The M units can perform multiply operations. One M unit contains two multipliers, and is therefore also called a dual multiplier. The A units can perform address and memory load/store operations. The LC unit can perform loop counter operations. The EI unit can perform external input operations. The EO unit can perform external output operations. Each unit can perform one operation per clock cycle and all operations take a single clock cycle. The Montium2 can calculate one complex multiplication per clock cycle. The maximum clock frequency is targeted at 200 MHz.

The Montium2 targets the 16-bit DSP algorithm domain. Therefore, the word size of the Montium2 is 16-bit. In most DSP algorithms complex data samples are used, consisting of one 16-bit word for the real part of the sample and one 16-bit word for the imaginary part of the sample. In the Montium2 the

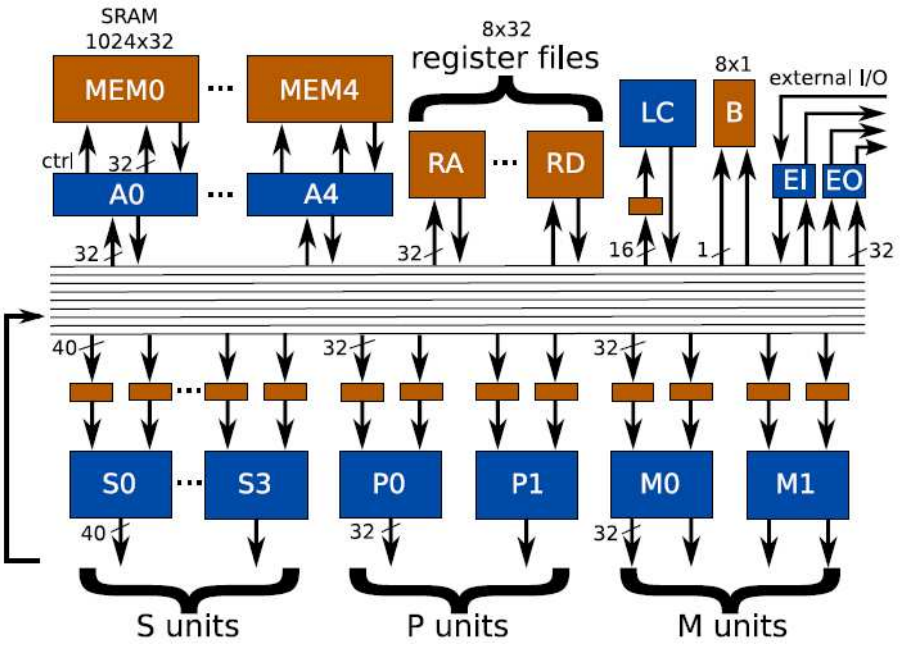


Figure 5.1: Simplified schematic representation of the Montium2 architecture.

width of the datapath is 32-bit, therefore the 16-bit real and 16-bit imaginary part of a complex number can be transported simultaneously. The datapath of the S units is locally expanded to 40-bit. The additional 8 bits are accumulator bits. A 40-bit word cannot be stored in memory, it has to be transformed into a 32-bit word by a shift operation, because the width of the SRAM units are 32-bit.

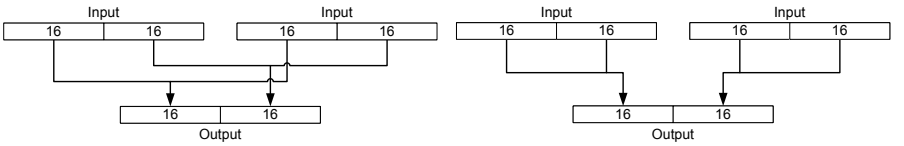
The S, P, and M units can operate in 3 different modes:

- *32-bit double word (dword) mode*
- *vertical vector mode*
- *horizontal vector mode*

In *32-bit dword mode* the S and P units interpret the 32-bit dword as one word. The M units use only the 16 Least Significant Bits (LSBs) of the 32-bit dword, because the multipliers inside the M units are 16-bit. The output of a multiplier is 32-bit. In *32-bit dword mode* only one multiplier of an M unit is used, and the output of an M unit is one 32-bit dword.

In *vertical vector mode* and *horizontal vector mode* a 32-bit dword is interpreted as a vector containing two 16-bit words. In figure 5.2 the relations between the input and the output of the two vector modes are shown. In *vertical vector mode* the upper 16 bits of the output dword are calculated out of the upper 16 bits of the two input dwords. The lower 16 bits of the output dword are calculated out of the lower 16 bits of the two input dwords. In *horizontal vector mode* the upper 16 bits of the output dword are calculated out of the first input argument. The lower 16 bits of the output dword are calculated

out of the second input argument. The two multipliers of an M unit are used simultaneously in both *vector modes*, and the output of an M unit are two 32-bit dwords.



(a) Vertical mode. (b) Horizontal mode.

Figure 5.2: Vector modes of the Montium2.

Additionally, the S units can operate in *40-bit accumulator mode*. In this mode an S unit operates as an 40-bit accumulator. Because of the 8 additional bits more dwords can be accumulated before a shift operation transforms the 40-bits into a 32-bit dword. The 32-bit result is calculated using less round off actions, and therefore the result is calculated more accurately. All other functional units always operate in *32-bit dword mode*. In all modes the unit operands are interpreted, either as signed integer numbers, or as signed fixed point numbers.

The architecture and instruction set of the Montium2, as well as the name itself, are subject to changes, because the Montium2 is still in its development phase. The implementation described in the next sections is based on the architecture and instruction set as known at 17 september 2008. Because the Montium2 architecture and instruction set are subject to changes, the Montium2 compiler is not available. The programs written for the Montium2 can be compiled to a General Purpose Processor, to verify the functionality of the programs. A prediction of the computational load can only be done by a manual estimation of the mapping of the instructions onto the Montium2 architecture.

5.2 Music algorithm

The implementation of the MUSIC algorithm is based on a 16 element ULA. 1024 snapshots are used to calculate the covariance matrix, and the DOAs of 5 sources have to be estimated. A 16 element ULA is chosen because a 16 by 16 covariance matrix contains 256 elements, and this is a useful size because of the memory structure of the Montium2. 1024 snapshots are chosen because it is a power of 2, and therefore in an average calculation the division reduces to a shift operation. Estimation of 5 sources is chosen because of the same reason as explained in section 4.1. These parameters are also used in simulations described in section 4.1, and these simulations showed that it are practical parameters as well. This implementation of the MUSIC algorithm can be executed on one Montium2, and it is assumed that the complete algorithm will fit into the program memory of the Montium2. The size of the program memory is still unknown, therefore it cannot be said if it is a reasonable assumption. The pseudo code listed in the next sections is a mix of human language and a simplified version of the Montium2C code.

5.2.1 Covariance matrix

In this section the implementation of the calculation of the covariance matrix is discussed. The calculation of the covariance matrix is the first step of the MUSIC algorithm. This implementation of the MUSIC algorithm presumes the snapshots to be stored in some external memory.

Implementation

Because of the limited memory space in the Montium2 the snapshots are loaded in blocks of 64 snapshots. The 64 snapshots are stored into MEM0 and the conjugated values of those 64 snapshots are stored into MEM1 (see figure 5.4). The conjugated snapshots represent the Hermitian adjoint of the snapshots. The conjugate operation is done by the Montium2. A complex value a is conjugated by *vertical vector mode* operation:

```
a_conj = addsub_v(0, a);
```

The value 0 and the real part of a are summed, and the imaginary part of a is subtracted from 0. The `addsub_v` instruction can be executed on S and P units.

A covariance matrix is calculated out of the 64 snapshots using equation:

$$R_{xx} = \frac{1}{m} \sum_{i=1}^{m} X_i X_i^H \tag{5.1}$$

This equation is equal to equation 3.1. This covariance matrix calculation is implemented by:

```
void avg_covmatr_64snapsh(*memory_real, *memory_imag)
{
 for (i = 0; i < 16; i++)
 {
  for (j = 0; j < 16; j++)
  {
   acc_real = 0;
   acc_imag = 0;
   for (k = 0; k < 64; k++)
   {
    c = compl_mul(snapshot64[i+k*16], snapshot64_conj[j+k*16]);
    acc_real = add_a(acc_real, c.real);
    acc_imag = add_a(acc_imag, c.imag);
   }
   memory_real[j*16+i] = asr_a(acc_real, 6);
   memory_imag[j*16+i] = asr_a(acc_imag, 6);
  }
 }
}
```

The pointers `*memory_real` and `*memory_imag` point to a block of 256 dwords in respectively MEM2 and MEM3, where the real and imaginary part of R_{xx} are stored. The array `snapshot64` contains the values of 64 snapshots. The

array `snapshot64_conj` contains the conjugated values of the same 64 snapshots. The variable c is a structure containing two 32-bit dwords, representing the real and imaginary part of a complex value. An element of R_{xx} is calculated by accumulating (`add_a`) all results of 64 complex multiplications. The accumulated value is shifted to the right (`asr_a`) by 6 positions, which is equal to a division by 64. The instructions `add_a` and `asr_a` are operations of an S unit in *accumulator mode*. Three successive blocks of 64 snapshots are processed in an identical way to fill MEM2 and MEM3 with the real and imaginary values of 4 covariance matrices. The function `compl_mul` is an implementation of the complex multiplication. A complex multiplication consists of four multiplications and an addition and subtraction ($(a_r+ia_i)(b_r+ib_i) = a_rb_r-a_ib_i+i(a_rb_i+a_ib_r)$). These operations can be mapped on the Montium2 efficiently in a pipelined structure.

```
struct complex_ri compl_mul(a, b)
{
 mul_v(a, b, PSL1S, &abh, &abl);
 mul_v(a, b, SWAP_B | PSL1S, &absh, &absl);
 c.real = sub(abh, abl);
 c.imag = add(absh, absl);
 return c;
}
```

In the first clock cycle the two M units calculate the four multiplications using the `mul_v` instruction. This instruction multiplies values a and b in *vertical vector mode*. The operands of the M units are represented as 16-bit signed fixed point numbers in the Q(0,15) format (1 sign bit (not denoted), 0 integer bits, and 15 fractional bits). The result of the multiplication is represented in Q(1,14) format. The option `PSL1S` of the `mul_v` instruction shifts the result of the multiplication to the left by 1 position, to correct the representation of the result into the Q(0,15) format. The option `SWAP_B` interchanges the two elements of the second input operand, to calculate the values of a_rb_i and a_ib_r. All instruction options do not require additional clock cycles. The results of the multiplications are four intermediate 32-bit dwords.

In the second clock cycle, two dwords are input for a subtraction in an S unit. The other two dwords are input for an addition in an other S unit. The subtraction and addition are both executed in *32-bit dword mode*. The result of the subtraction represents the real part of a complex value in 32-bit. The result of the addition represents the imaginary part of the same complex value in 32-bit. The function `compl_mul` returns a structure containing these two dwords.

The function `compl_mul_to_dword()` combines the result of function `compl_mul()` in a vector representation of one dword.

```
void compl_mul_to_dword(a, b, *c)
{
 c_temp = compl_mul(a, b);
 *c = packhh(c_temp.real, c_temp.imag);
}
```

The instruction `packhh` selects the 16 Most Significant Bits (MSBs) of both dwords, and stores them into the 16 MSBs and 16 LSBs of one dword. The

`packhh` instruction is executed in the third clock cycle in a P unit. Figure 5.3 shows the pipeline of a complex multiplication on the Montium2.

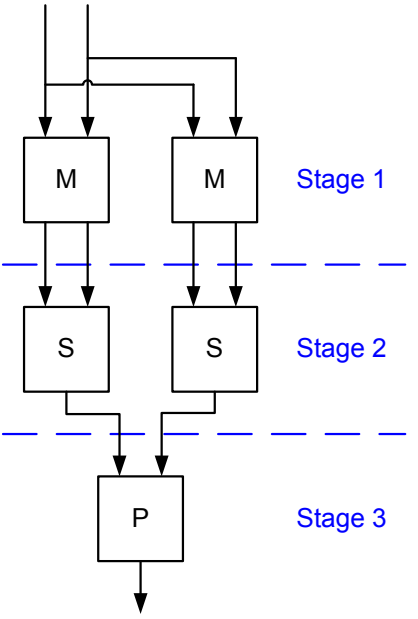


Figure 5.3: Three stage pipeline of the complex multiplication

As a consequence of the pipelined execution, the complex multiplication produces a result every clock cycle. The latency of one complex multiplication depends on the representation of the input of the subsequent instructions. If the next instructions need a separate real and imaginary part the latency is 2 clock cycles. If the next instructions need a combined representation in one dword, the latency is 3 clock cycles. For now the complex multiplication is implemented in a function, in subsequent versions of the Montium2C compiler instructions will be introduced to support the pipelined execution. Therefore, a pipelined execution of the complex multiplication is assumed.

An average of the 4 covariance matrices is calculated and the real and imaginary parts are combined into 32-bit dwords.

```
void avg_4matrices(*input_mem_real, *input_mem_imag, *output_mem)
{
 for (i = 0; i < 256; i++)
 {
  acc_real = 0;
  acc_imag = 0;
  for (j = 0; j < 4; j++)
  {
   acc_real = add_a(acc_real, input_mem_real[j*256+i]);
   acc_imag = add_a(acc_imag, input_mem_imag[j*256+i]);
  }
  real = asr_a(acc_real, 2);
  imag = asr_a(acc_imag, 2);
```

```
  output_mem[i] = packhh(real, imag);
 }
}
```

Pointer `*input_mem_real` points to MEM2 and pointer `*input_mem_imag` points to MEM3. Pointer `*output_mem` points to a block of 256 dwords in MEM4, where the average covariance matrix is stored. The real and imaginary parts of the ith elements of the four matrices stored in MEM2 and MEM3 are accumulated and shifted to the right by 2 positions (division by 4) to calculate element i of the average covariance matrix. The instruction `packhh` selects the 16 MSBs of both dwords, and stores them into the 16 MSBs and 16 LSBs of one dword. The 12 remaining blocks of 64 snapshots are processed as described above to fill MEM4 with 4 covariance matrices.

The last step of the calculation of the covariance matrix is the calculation of the average covariance matrix of the 4 matrices stored in MEM4. The result is the final covariance matrix, containing information of 1024 snapshots.

```
for (i = 0; i < 256; i++)
{
 a = add_v(MEM4[i], MEM4[i+256], PSR1);
 b = add_v(MEM4[i+2*256], MEM4[i+3*256], PSR1);
 c = add_v(a, b, PSR1);
 compl_mul_to_dword(c, div_val, &MEM0[i]);
}
```

The values of the 4 matrices are combined, using an addertree structure. Instruction `add_v` is an add instruction in *vertical vector mode*. The option `PSR1` of the `add_v` instruction is used to shift the result of the addition to the right by 1 position (a division by 2). The advantage of using the `add_v` instruction including the `PSR1` option is that an average calculation of two values van be executed in 1 clock cycle. The resulting values are divided by 16, to keep the eigenvalues smaller than 1. The size of the eigenvalues scales linear with the size of the values of the matrix, without influencing the direction of the eigenvectors [9]. The value 16 is a result of an empirical observation. The division is implemented by a multiplication with 1/16 (*div_val*). The funtion `compl_mul_to_dword()` is an implementation of the 3 staged complex multiplication. The final matrix is stored in the first 256 dwords of MEM0.

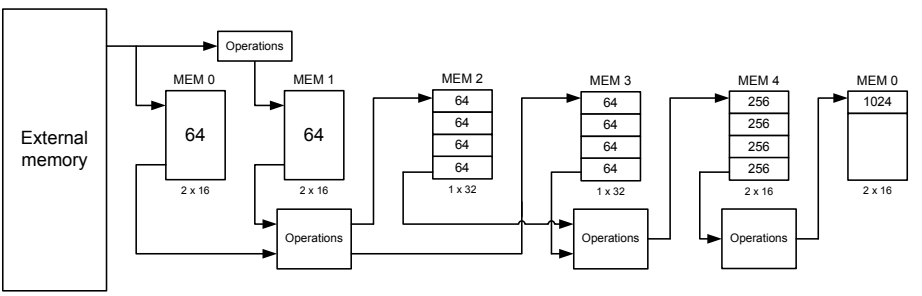


Figure 5.4: Memory usage during the covariance matrix calculation

The numbers inside the memory blocks of figure 5.4 describe de number of snapshots used to calculate the data, stored in that block. The numbers under the memory blocks describe the representation of the data, 2×16 for *vector mode*, and 1×32 for *dword mode*.

Accuracy

In the implementation both 16-bit and 32-bit values are used. The LSB values in this section are defined by the LSB of a 16-bit value.

The worst-case quantization error of the elements in a snapshot is $\frac{1}{2}$LSB. A fixed point multiplication of two values containing an error of $\frac{1}{2}$LSB, results in a value containing a worst-case error of 1LSB. A fixed point addition or subtraction increases the worst-case error by 1LSB. In the first stage of the complex multiplication 4 multiplications are executed in parallel, so the 4 results contain a worst-case error of 1LSB each. Keep in mind that the input operands of the multiplication are always ≤ 1 in this implementation, therefore no extra integer bits are needed in the result. The `PSL1S` option of the `mul_v` instruction corrects the representation of the result into Q(0,15) representation. In the second stage of the complex multiplication an addition and subtraction are executed in parallel. Both results contain a worst-case error of 2LSBs, which is therefore the worst-case error of the 2-staged complex multiplication.

An element of the resulting matrix of multiplication of a snapshot to its Hermitian adjoint is calculated by one 2-staged complex multiplication. In the calculation of an average covariance matrix of 64 snapshots, the 64 complex values are accumulated in two accumulators (1 real part accumulator, and 1 imaginary part accumulator) to prevent saturation. The 63 additions in the accumulator increase the worst-case error by 63LSBs to 65LSBs. The accumulated values are divided by 64 (right shift by 6 positions), so the worst-case error in a covariance matrix containing the average of 64 snapshots is $\lceil 65/64 \rceil$ = 2LSBs per element (a ceiling function is needed because of the truncation of the shift action).

The next averaging step, the averaging of 4 matrices, the worst-case error is increased to $\lceil (4+2)/4 \rceil$ = 2LSBs.The matrices stored in MEM4 are averaged in two steps. The worst-case error of the calculation of the final covariance matrix is $\lceil (1 + \lceil (1+2)/2) \rceil / 2 \rceil$ = 2LSBs.The final covariance matrix is scaled by means of a complex multiplication. Therefore, the final worst-case error of an element of the covariance matrix is 4LSBs.

Computational load

For the estimation of the computational load, it is assumed that a dword can be fetched from the external memory every clock cycle. To store a block of 64 snapshots into MEM0 and MEM1, $16 \times 64 = 1024$ clock cycles are needed. During the calculation of the covariance matrix 16 blocks are stored into the memory of the Montium2. A total of $1024 \times 16 = 16384$ clock cycles are needed to store all snapshot into the memory of the Montium2.

The calculation of the covariance matrix of 1 snapshot requires $16 \times 16 = 256$ multiplications. The calculation of the covariance matrix of 1 block of 64 snapshots requires $256 \times 64 = 16384$ multiplications. The calculation of the covariance matrices of the 16 blocks requires $16384 \times 16 = 262144$ multiplications.

The calculation of the average of the 4 matrices stored in MEM2 and MEM3 requires $4 \times 256 = 1024$ averaging operations. The averaging of these matrices is executed four times during the covariance matrix calculation.

The final covariance matrix is calculated out of the 4 matrices stored in MEM4 by a two staged adder tree. The result is calculated in $3 \times 256 = 768$ averaging operations. See table 5.2 for an overview of the computational load. The loop and pipeline overhead is hard to estimate, because the Montium2 is not implemented yet. Therefore, the overhead is not mentioned.

Operation	**Cycles**	**# Executions**	**Total**
load 64 snapshots	1024	16	16348
average 64 snapshots	16384	16	262144
average 4 matrices (MEM2/3)	1024	4	4096
average 4 matrices (MEM4)	768	1	768
total clock cycles			283356

Table 5.1: Clock cycles of the covariance matrix.

Scalability

This implementation of the calculation of a 16 by 16 covariance matrix can be executed on one Montium2. It is estimated that an implementation based on a 32 by 32 matrix is the largest covariance matrix calculation that can be executed on one Montium2. A 32 by 32 matrix fits exact in 1 memory block of 1024 dwords. An accumulator contains 8 extra bits, so 256 values can be accumulated before it has to be scaled and stored. Four average matrices, based on 256 values each, can be stored in 4 memory blocks. The average matrix of these 4 matrices, which is the covariance matrix based on 1024 snapshots, can be stored in the 5th memory block. This 32 by 32 covariance matrix calculation can be executed on one Montium2, at a costs of more load and store operations to external memory.

The calculation of a larger covariance matrix has to be distributed over multiple Montium2s. The covariance matrix can be separated into smaller pieces, and each piece can be assigned to a Montium2. An example of a separation is given by:

$$\left(\begin{array}{c|c} M2_1 & M2_2 \\ \hline M2_3 & M2_4 \end{array}\right)$$

where $M2$ denotes a Montium2. This division of the covariance matrix can be done at the cost of tripling the communication overhead.

5.2.2 Eigendecomposition

In this section the implementation of the eigendecomposition of the covariance matrix is discussed. The calculation of the eigendecomposition is the second

step of the MUSIC algorithm. The eigendecomposition is based on the QR algorithm. In section 3.3.1 the theory of the QR algorithm is described.

Implementation

The explanation of the implementation of the eigendecomposition is subdivided into three paragraphs. Due to the subdivision, the implementation is explained in a top-down approach.

QR algorithm The QR algorithm is explained by means of equation 3.16. This equation is repeated below.

$$\begin{aligned} A = Q_0 R_0, \quad S_0 = Q_0 \\ A_{k+1} = R_k Q_k = Q_{k+1} R_{k+1}, \quad S_k = S_{k-1} Q_k, \quad k = 0, 1, 2, \ldots \end{aligned} \tag{5.2}$$

The QR algorithm starts with the QR decomposition of the covariance matrix stored in the first 256 dwords of MEM0. The function `qr_decomp()` is an implementation of the QR decomposition. The implementation of this function is explained in the next paragraph. The similarity between the code described below and equation 5.2 is demonstrated by: A = `covariance_matrix0[]`, Q_0 = `Q_matr[]`, R_0 = `R_matr[]`, and S_0 = `S_matr[]`. `Q_matr[]` is stored in MEM2, `R_matr[]` is stored in the first 256 dwords of MEM3, and `S_matr[]` is stored in the second 256 dwords of MEM3.

Equations $A_{k+1} = R_k Q_k = Q_{k+1} R_{k+1}$ and $S_k = S_{k-1} Q_k$ of the QR algorithm are implemented in a loop.

```
qr_decomp(covariance_matrix0, Q_matr, R_matr);
for (idx i = 0; i < 256; i++)
{
 S_matr[i] = Q_matr[i];
}
while (cmpgt(eigval_max, 0.001))
{
 matr_mult(R_matr, Q_matr, RQ_matr);
 qr_decomp(RQ_matr, Q_matr, R_matr);
 matr_mult2(S_matr, Q_matr, S_matr);
 eigval_max = 0;
 for (idx i = 0; i < 16; i++)
 {
  eigval_temp = R_matr[i*16+i];
  eigval_diff = sub_v(eigvals[i], eigval_temp);
  eigvals[i] = eigval_temp;
  eigval_max = max(packhh(abs_hl(eigval_diff), 0), eigval_max);
 }
}
```

The function `matr_mult()` is an implementation of a matrix multiplication. The function `matr_mult2()` is an implementation of a matrix multiplication, where the result of the multiplication is stored on the location of one of the multiplied matrices. The values on the diagonal of R are assumed to be converged to the eigenvalues of the covariance matrix, if the maximal difference

between the diagonal of R_{k-1} and R_k is smaller than 0.001. The choice of 0.001 is explained later. `eigvals[]` is an array containing values of the diagonal of the previous R_{k-1}. `eigval_diff` is the difference between the previous and the current value of an element of the diagonal of R. `eigval_max` is the maximum difference of all differences between the two diagonals. In the instruction sequence `packhh(abs_hl(eigval_diff), 0)` provides `abs_hl` an elementwise absolute value (i.e. the sign bits of both vector elements are cleared). An elementwise operation, since `eigval_diff` is a complex value. The `packhh` instruction in combination with the value 0, cleares the imaginary part of `eigval_diff`. In case of a Hermitian covariance matrix the eigenvalues are always real, therefore the real parts of the eigenvalues are compared. To ensure that the imaginary part has no influence on the comparison, the imaginary part is cleared.

The function `matr_mult2()` is implemented by:

```
void matr_mult2(*input_matr1, *input_matr2, *output_matr)
{
 matr_mult(input_matr1, input_matr2, temp_matrix);
 for (i = 0; i < 256; i++)
 {
  output_matr[i] = temp_matrix[i];
 }
}
```

The matrix multiplication is calculated by the function `matr_mult()`, the result is stored in temporary matrix `temp_matrix[]`. After the multiplication the values of `temp_matrix[]` are copied into `output_matr[]`, which is one of the multiplied matrices.

The function `matr_mult()` is implemented by:

```
void matr_mult(*input_matr1, *input_matr2, *output_matr)
{
 for (i = 0; i < 16; i++)
 {
  for (j = 0; j < 16; j++)
  {
   acc_real = 0;
   acc_imag = 0;
   for (k = 0; k < 16; k++)
   {
    c = compl_mul(input_matr1[k*16+j], input_matr2[i*16+k]);
    acc_real = add_a(acc_real, c.real);
    acc_imag = add_a(acc_imag, c.imag);
   }
   output_matr[i*16+j] = packhh(acc_real, acc_imag);
  }
 }
}
```

A row of `input_matr1[]` is elementwise multiplied with a column of `input_matr2[]`, and the real and imaginary results are separately accumulated. The real and imaginary parts are combined and stored in `output_matr[]`.

QR decomposition The QR decomposition is explained by means of equations 3.35 and 3.36. These equations are repeated below. The row and column indices of the matrices are replaced by the values used in this implementation.

$$R = Q_{16,15}Q_{15,14}Q_{16,14}\dots Q_{14,1}Q_{15,1}Q_{16,1}A \quad (5.3)$$

$$Q = Q^H_{16,1}Q^H_{15,1}Q^H_{14,1}\dots Q^H_{16,14}Q^H_{15,14}Q^H_{16,15} \quad (5.4)$$

The QR decomposition is implemented as the funtion `qr_decomp()`. The structure of this function is given by:

```
void qr_decomp(*input_matrix, *Q_matrix, *R_matrix)
{
 separate input_matrix into A_real and A_imag.
 initialize Q_real as I and Q_imag as 0.

 for (col = 0; col < 15; col++)
 {
  for (row = 15; row > col; row--)
  {
   calculate theta1 by CORDIC.
   calculate theta2 by CORDIC.
   calculate theta2neg.

   calculate Q_11_22.
   calculate Q_12.
   calculate Q_21.

   calculate A_real, A_imag by optimized multiplication.

   calculate Qh_11_22.
   calculate Qh_12.
   calculate Qh_21.

   calculate Q_real, Q_imag by optimized multiplication.
  }
 }
 combine A_real, A_imag into R_matrix.
 combine Q_real, Q_imag into Q_matrix.
}
```

The elements of `input_matrix[]` are separated into a real and imaginary part. The real part is copied into `A_real`, and the imaginary part is copied into `A_imag`. This separation is needed to achieve an acceptable accuracy of the QR decomposition, because the intermediate real and imaginary values can now be stored in 32-bit precision. This holds also for matrices `Q_real` and `Q_imag`. Matrix `Q_real` is initialized as the identity matrix, and matrix `Q_imag` is completely initialized to 0.

To calculate the complete QR decomposition, 120 $Q_{i,j}$ (see equation 3.34) matrices have to be calculated. `theta1` $= \theta_1$, `theta2` $= \theta_2$, and `theta2neg` $= -\theta_2$. These variables have to be recalculated for every $Q_{i,j}$ matrix. The calculation of these values by COordinate Rotation DIgital Computer (CORDIC)

is explained later. The values of variables `Q_11_22`, `Q_12`, and `Q_21` are calculated out of `theta1`, `theta2`,and `theta2neg`. Equation 5.5 shows the similarity between elements of the rotation matrix and the variables `Q_11_22`, `Q_12`, and `Q_21`.

$$\begin{pmatrix} \cos\theta_1 & (\sin\theta_1)e^{i\theta_2} \\ (-\sin\theta_1)e^{-i\theta_2} & \cos\theta_1 \end{pmatrix} = \begin{pmatrix} Q_{11\_22} & Q_{12} \\ Q_{21} & Q_{11\_22} \end{pmatrix} \tag{5.5}$$

The multiplication of matrices $Q_{i,j}$ and A (see equation 5.3) can be executed using an optimized matrix multiplication. This multiplication is explained later. The values of variables `Qh_11_22`, `Qh_12`, and `Qh_21` are calculated out of `Q_11_22`, `Q_12`, and `Q_21` by conjugation. The multiplication of two $Q_{i,j}^H$ matrices (see equation 5.4) is also implemented by an optimized matrix multiplication.

When all 120 $Q_{i,j}$ matrices are calculated and multiplied to A, the real part of the result is stored in `A_real` and the imaginary part is stored in `A_imag`. The elements of `A_real` and `A_imag` are combined and copied into `R_matrix`. When all 120 $Q_{i,j}^H$ matrices are calculated and multiplied to each other, the real part of the result is stored in `Q_real` and the imaginary part is stored in `Q_imag`. The elements of `Q_real` and `Q_imag` are combined and copied into `Q_matrix`.

The optimized matrix multiplication mentioned before, is explained by means of an example of a multiplication of two 3 by 3 matrices. Let matrix Q_{ex} be an example of a 3 by 3 version of matrix $Q_{i,j}$. Matrix Q_{ex} can be separated into the identity matrix and a zero matrix except for four values.

$$Q_{ex} = \begin{pmatrix} a & 0 & b \\ 0 & 1 & 0 \\ c & 0 & a \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + \begin{pmatrix} a-1 & 0 & b \\ 0 & 0 & 0 \\ c & 0 & a-1 \end{pmatrix} \tag{5.6}$$

Let general non-sparse 3 by 3 matrix A_{ex} be defined by:

$$A_{ex} = \begin{pmatrix} r & s & t \\ u & v & w \\ x & y & z \end{pmatrix} \tag{5.7}$$

Matrices Q_{ex} and A_{ex} are multiplied.

$$\begin{aligned} Q_{ex}A_{ex} &= \begin{pmatrix} a & 0 & b \\ 0 & 1 & 0 \\ c & 0 & a \end{pmatrix} \begin{pmatrix} r & s & t \\ u & v & w \\ x & y & z \end{pmatrix} \\ &= \left(\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + \begin{pmatrix} a-1 & 0 & b \\ 0 & 0 & 0 \\ c & 0 & a-1 \end{pmatrix} \right) \begin{pmatrix} r & s & t \\ u & v & w \\ x & y & z \end{pmatrix} \\ &= \begin{pmatrix} r & s & t \\ u & v & w \\ x & y & z \end{pmatrix} + \begin{pmatrix} ar-r+bx & as-s+by & at-t+bz \\ 0 & 0 & 0 \\ cr+ax-x & cs+ay-y & ct+az-z \end{pmatrix} \\ &= \begin{pmatrix} 0 & 0 & 0 \\ u & v & w \\ 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} ar+bx & as+by & at+bz \\ 0 & 0 & 0 \\ cr+ax & cs+ay & ct+az \end{pmatrix} \\ &= \begin{pmatrix} ar+bx & as+by & at+bz \\ u & v & w \\ cr+ax & cs+ay & ct+az \end{pmatrix} \end{aligned} \tag{5.8}$$

The multiplication of matrices Q_{ex} and A_{ex} only affects the top and bottom rows of A_{ex}, as can be seen in equation 5.8. Due to the specific structure of Q_{ex}, the calculations in the multiplication of Q_{ex} and A_{ex} can be reduced to:

$$\begin{pmatrix} a & b \\ c & a \end{pmatrix} \begin{pmatrix} r & s & t \\ x & y & z \end{pmatrix} = \begin{pmatrix} ar+bx & as+by & at+bz \\ cr+ax & cs+ay & ct+az \end{pmatrix} \tag{5.9}$$

To produce the complete result of the multiplication, the top row of matrix A_{ex} is replaced by the top row of the result of equation 5.9, and the bottom row of matrix A_{ex} is replaced by the bottom row of the result of equation 5.9. If both rows of the second matrix of equation 5.9 are expanded to 16 elements, this optimized matrix multiplication also works on 16 by 16 matrices. The first matrix of equation 5.9 does not change. All 120 $Q_{i,j}$ matrices can be multiplied by this optimized multiplication. The rows affected by this multiplication depend on the position of the values $\cos\theta_1$, $(\sin\theta_1)e^{i\theta_2}$, and $(-\sin\theta_1)e^{-i\theta_2}$ in matrix $Q_{i,j}$. The calculation of the values of `Q_11_22`, `Q_12`, and `Q_21` and the optimized matrix multiplication are implemented by:

```
Q_11_22.real = theta1.real;
Q_11_22.imag = 0;
temp_val.real = theta1.imag;
temp_val.imag = 0;
Q_12 = compl_mul32(temp_val, theta2);
temp_val.real = neg(theta1.imag);
temp_val.imag = 0;
Q_21 = compl_mul32(temp_val, theta2neg);

for (i = 0; i < 16-col; i++)
{
 temp_upper.real = A_real[col+16*col+16*i];
 temp_upper.imag = A_imag[col+16*col+16*i];
 temp_lower.real = A_real[row+16*col+16*i];
 temp_lower.imag = A_imag[row+16*col+16*i];
 temp1 = compl_mul32(temp_upper, Q_11_22);
 temp2 = compl_mul32(temp_lower, Q_12);
 A_real[col+16*col+16*i] = add(temp1.real, temp2.real);
 A_imag[col+16*col+16*i] = add(temp1.imag, temp2.imag);
 temp1 = compl_mul32(temp_upper, Q_21);
 temp2 = compl_mul32(temp_lower, Q_11_22);
 A_real[row+16*col+16*i] = add(temp1.real, temp2.real);
 A_imag[row+16*col+16*i] = add(temp1.imag, temp2.imag);
}
```

If the angles θ_1, θ_2, and $-\theta_2$ are each represented by a complex value (with an absolute value of 1) in Cartesian representation. The cosine of such an angle is represented by the real part of the complex value, and the sine is represented by the imaginary part of the complex value (Euler's formula: $e^{i\theta} = \cos\theta + i\sin\theta$). `theta1`, `theta2`, and `theta2neg` are all structure variables containing two dwords. One dword for the real part of a complex value, and one dword for the imaginary part of a complex value.

The values of variables `Q_11_22`, `Q_12`, and `Q_21` are assigned according to equation 5.5. To calculate the values of these variables, function `compl_mul32()`

is used. `compl_mul32()` is an implementation of a 32-bit complex multiplication. This function is not yet implemented in Montium2C code, but as a C function. For now, the Montium2C compiler and simulator are under construction and accept and execute C code, as long as the syntax is correct. To execute this implementation of the MUSIC algorithm on the Montium2, the function `compl_mul32` has to be implemented using Montium2C code. The need for an 32-bit complex multiplication is explained later. The optimized matrix multiplication is implemented in the loop, in a configuration equal to equation 5.9.

The relation between `Qh_11_22`, `Qh_12`, `Qh_21` and `Q_11_22`, `Q_12`, `Q_21` is shown by equation 5.10 The values of `Qh_12` and `Qh_21` are calculated out of `Q_12` and `Q_21` using conjugate a operation.

$$\begin{pmatrix} Q_{11\_22} & Q_{12} \\ Q_{21} & Q_{11\_22} \end{pmatrix}^{H} = \begin{pmatrix} Qh_{11\_22} & Qh_{12} \\ Qh_{21} & Qh_{11\_22} \end{pmatrix} \tag{5.10}$$

The implementation of the optimized matrix multiplication used for the multiplications in equation 5.4, has a small difference with respect to equation 5.9. In equation 5.11 the order of the multiplied matrices is changed. The reduced matrix of A_{ex} is now in front of the reduced matrix of Q_{ex}, and two columns of A_{ex} are affected instead of two rows.

$$\begin{pmatrix} r & t \\ u & w \\ x & z \end{pmatrix} \begin{pmatrix} a & b \\ c & a \end{pmatrix} = \begin{pmatrix} ra+tc & rb+ta \\ ua+wc & ub+wa \\ xa+zc & xb+za \end{pmatrix} \tag{5.11}$$

The calculation of the values of `Qh_11_22`, `Qh_12`, and `Qh_21` and the second optimized matrix multiplication are implemented by:

```
Qh_11_22 = Q_11_22;
Qh_21.real = Q_12.real;
Qh_21.imag = neg(Q_12.imag);
Qh_12.real = Q_21.real;
Qh_12.imag = neg(Q_21.imag);

for (i = 0; i < 16; i++)
{
 templ.real = Q_real[i+col*16];
 templ.imag = Q_imag[i+col*16];
 tempr.real = Q_real[i+row*16];
 tempr.imag = Q_imag[i+row*16];
 temp1 = compl_mul32(templ, Qh_11_22);
 temp2 = compl_mul32(tempr, Qh_21);
 Q_real[i+col*16] = add(temp1.real, temp2.real);
 Q_imag[i+col*16] = add(temp1.imag, temp2.imag);
 temp1 = compl_mul32(templ, Qh_12);
 temp2 = compl_mul32(tempr, Qh_11_22);
 Q_real[i+row*16] = add(temp1.real, temp2.real);
 Q_imag[i+row*16] = add(temp1.imag, temp2.imag);
}
```

CORDIC A CORDIC algorithm [3, 21] is an algorithm to calculate hyperbolic and trigonometric functions. The algorithm requires addition, subtraction, bitshift and table lookup operations. The algorithm is originally designed for computer architectures where no hardware multiplier is available. The CORDIC algorithm is iterative. After every iteration the precision of the result is increased by 1 bit. In this implementation the CORDIC algorithm is used to transform a complex number from polar form to cartesian form and the other way around. Although the Montium2 contains multipliers the CORDIC is still useful, because no table lookup operations are needed if the number of iterations is fixed. In this implementation the number of iterations is fixed to 32, since the complex number transformations are calculated in 32-bit precision.

The CORDIC algorithm can be used in two modes:

- *rotation mode*
- *vectoring mode*

The *rotation mode* is used to rotate a vector over a known angle. In figure 5.5 three iterations of the *rotation mode* are shown. Vector v_0 is rotated over angle β in three steps. The rotation angles mentioned below, are different from the rotation angles of the implementation, but they clarify the basic idea of the CORDIC algorithm. In the first iteration v_0 is rotated over 45° to v_1. In the second iteration, this new angle (45°) is compared to β to determine the sign of the next rotation. In this example the angle of v_1 is smaller than β. The second rotation is over $45°/2 = 22.5°$ to v_2 (67.5°). In the third iteration, the angle of v_2 is compared to β, and rotated over $-(22.5°/2) = -11.25°$ to v_3 (56.25°). If the difference between the angle of the rotated vector and β is smaller than a predefined value the algorithm is stopped.

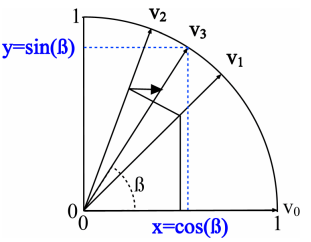


Figure 5.5: Three iterations of the CORDIC algorithm

The *vectoring mode* is used to determine the length of a vector and the angle of a vector with respect to a vector (1,0). If, for example, v_3 in figure 5.5 is a vector of unknown length and angle, these values can be determined using the *vectoring mode*. In this example the angle of v_3 is reduced to 0 in three iterations ($56.25° - 45° - 22.5° + 11.25° = 0°$). The length of the vector is the value of the non zero element in the vector, in this example (1,0). The CORDIC algorithm is explained by means of a citation out of [3].

> The CORDIC algorithm is derived from the general (Givens) rotation transform:
>
> $$\begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= y\cos\theta + x\sin\theta \end{aligned} \qquad (5.12)$$

which rotates a vector in a cartesian plane by angle θ. These can be rearranged so that:

$$\begin{aligned} x' &= \cos\theta(x - y\tan\theta) \\ y' &= \cos\theta(y + x\tan\theta) \end{aligned} \tag{5.13}$$

So far, nothing is simplified. However, if the rotation angles are restricted so that $\tan\theta = \pm 2^{-i}$, the multiplication by the tangent term is reduced to a simple shift operation. Arbitrary angles of rotation are obtainable by performing a series of successively smaller elementary rotations. If the decision at each iteration, i, is which direction to rotate rather than whether or not to rotate, then the $\cos\delta_i$ term becomes a constant (because $\cos\delta_i = \cos -\delta_i$). The iterative rotation can now be expressed as:

$$\begin{aligned} x_{i+1} &= K_i(x_i - y_i d_i 2^{-i}) \\ y_{i+1} &= K_i(y_i + x_i d_i 2^{-i}) \end{aligned} \tag{5.14}$$

where:

$$K_i = \cos(\arctan 2^{-i}) = 1/\sqrt{1 + 2^{-2i}}, \quad d_i = \pm 1 \tag{5.15}$$

Removing the scale constant from the iterative equations yields a shift-add algorithm for vector rotation. The product of the K_i's can be applied elsewhere in the system or treated as part of a system processing gain. That product approaches 0.6073 as the number of iterations goes to infinity. Therefore, the rotation algorithm has a gain, A_n, of approximately 1.647. The exact gain depends on the number of iterations, and obeys the relation

$$A_n = \prod_n \sqrt{1 + 2^{-2i}} \tag{5.16}$$

The angle of a composite rotation is uniquely defined by the sequence of the directions of the elementary rotations. That sequence can be represented by a decision vector.

In *rotation mode* the decision vector is known, while in *vectoring mode* the decision vector is determined. Since the number of iterations is fixed at 32, the value of $K(32) = \prod_{i=0}^{31} K_i$ can be precalculated.

Before an angle of a vector, or in this example a complex, value is determined by the *vectoring mode* of the CORDIC algorithm, the complex value is scaled. The scaling increases the accuracy of the CORDIC algorithm.

```
a.real = A_real[col+16*col];
a.imag = A_imag[col+16*col];
exp_a.real = exp(a.real);
exp_a.imag = exp(a.imag);
exp_shift_a = min(exp_a.real, exp_a.imag);
a.real = asl(a.real, exp_shift_a);
a.imag = asl(a.imag, exp_shift_a);
absa = cordic_vec(a, rhoa);
absa = asr(absa, exp_shift_a);
```

In the example above, a complex value a is scaled. The instruction `exp` returns the number of sign bits−1. This number is equal to the number of positions value a_{real} can be shifted to the left before it saturates. The minimal shift value `exp_a` is determined. Value a is shifted to the left. Function `cordic_vec()` is an implementation of the CORDIC algorithm in *vectoring mode*. `cordic_vec()` returns the absolute value of a (*absa*) and the decision vector of a (array `rhoa[]`). The absolute value of a is scaled back to the original length.

The function `cordic_vec()` is implemented by:

```
cordic_vec(compl_val, *rho)
{
 K_val.real = 0.6073;
 K_val.imag = 0;
 compl_val = compl_mul32(compl_val, K_val);
 I = compl_val.real;
 Q = compl_val.imag;

 for (i = 0; i < 32; i++)
 {
  I_shift = asr(I, i);
  Q_shift = asr(Q, i);
  if (bittest(xor_d(I, Q), 31))
  {
   rho[i+1] = 1;
   I = sub(I, Q_shift);
   Q = add(Q, I_shift);
  }
  else
  {
   rho[i+1] = -1;
   I = add(I, Q_shift);
   Q = sub(Q, I_shift);
  }
 }
 return(I);
}
```

The complex value is corrected for the processing gain of the CORDIC algorithm. The implementation of the loop is similar to equation 5.14.

The implementation of `cordic_vec()` described above works only if the complex value is in the first or fourth quadrant. Therefore, some extra code has to be added to accept all four quadrants.

```
if (cmplt(I, 0))
{
 if (cmpgt(Q, 0))
 {
  I_new = Q;
  Q = neg(I);
  I = I_new;
```

```
  rho[0] = -1;
 }
 else
 {
  I_new = neg(Q);
  Q = I;
  I = I_new;
  rho[0] = 1;
 }
}
else
{
 rho[0] = 0;
}
```

This code rotates the complex value by $\pm 90°$, without affecting the size of the complex value.

The structure of the calculation of θ_1 and θ_2, using the CORDIC algorithm in *vectoring mode* is shown in figure 5.6. The angles θ_1 and θ_2, both represented by a decision vector, are used in the QR decomposition. The subtraction $\theta_a - \theta_b$ is implemented by een elementwise subtraction of the decision vectors of a and b.

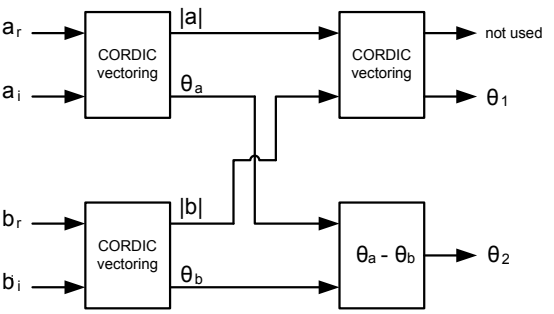


Figure 5.6: Calculation of θ_1 and θ_2.

The *rotation mode* of the CORDIC algorithm is used to rotate a vector, or in this case a complex value. Because of the different ranges of θ_1 and θ_2, two implementations of the *rotation mode* are made. The range of θ_1 is $0°$ to $90°$, because $|a|$ and $|b|$ are always nonnegative. The range of θ_2 is $-360°$ to $360°$, because the ranges of θ_a and θ_b are both $-180°$ to $180°$. A complex value is rotated by θ_1 using function `cordic_rot`. The function `cordic_rot` is implemented by:

```
cordic_rot(compl_val, *rho)
{
 K_val.real = 0.6073;
 K_val.imag = 0;
 compl_val = compl_mul32(compl_val, K_val);
 I = compl_val.real;
 Q = compl_val.imag;
```

```
 for (i = 0; i < 32; i++)
 {
  Q_shift = asr(Q, i);
  I_shift = asr(I, i);
  if (bittest(rho[i+1], 31))
  {
   I = sub(I, Q_shift);
   Q = add(Q, I_shift);
  }
  else
  {
   I = add(I, Q_shift);
   Q = sub(Q, I_shift);
  }
 }
 compl_val.real = I;
 compl_val.imag = Q;
 return(compl_val);
}
```

The complex value is corrected for the processing gain of the CORDIC algorithm. The implementation of the loop is similar to equation 5.14.

A complex value is rotated by θ_2 using function `cordic_rot2`. The first element of the decision vector of θ_2, which determines the starting quadrant, can contain the values 0, ± 1, and ± 2. In case of 0, the rotation starts in the current quadrant. In case of ± 1 the complex value is rotated by $\pm 90°$. In case of ± 2 the complex value is rotated by $\pm 180°$. All other elements can only contain the values 0 and ± 2. Because θ_2 is a difference angle, the rotations have to be executed twice in every iteration. Because of the double rotations the processing gain has to be corrected by the value $K(32)^2$. When the decision vector contains a zero, the complex value is rotated once and then rotated back to achieve the correct processing gain, because it is not known in advance where a zero is located. The function `cordic_rot2` is implemented by:

```
cordic_rot2(compl_val, *rho)
{
 K_val.real = 0.3688;
 K_val.imag = 0;
 compl_val = compl_mul32(compl_val, K_val);
 I = compl_val.real;
 Q = compl_val.imag;
 if (!(cmpeq(rho[0], 0)))
 {
  if (cmpeq(rho[0], 1))
  {
   I_new = Q;
   Q = neg(I);
   I = I_new;
  }
  if (cmpeq(rho[0], -1))
  {
```

```
   I_new = neg(Q);
   Q = I;
   I = I_new;
  }
  if (cmpeq(rho[0], -2)|cmpeq(rho[0], 2))
  {
   I = neg(I);
   Q = neg(Q);
  }
 }
 for (i = 0; i < 32; i++)
 {
  Q_shift = asr(Q, i);
  I_shift = asr(I, i);
  if (cmpeq(rho[i+1], 0))
  {
   I = add(I, Q_shift);
   Q = sub(Q, I_shift);
   Q_shift = asr(Q, i);
   I_shift = asr(I, i);
   I = sub(I, Q_shift);
   Q = add(Q, I_shift);
  }
  else
  {
   if (bittest(rho[i+1], 31))
   {
    I = sub(I, Q_shift);
    Q = add(Q, I_shift);
    Q_shift = asr(Q, i);
    I_shift = asr(I, i);
    I = sub(I, Q_shift);
    Q = add(Q, I_shift);
   }
   else
   {
    I = add(I, Q_shift);
    Q = sub(Q, I_shift);
    Q_shift = asr(Q, i);
    I_shift = asr(I, i);
    I = add(I, Q_shift);
    Q = sub(Q, I_shift);
   }
  }
 }
 compl_val.real = I;
 compl_val.imag = Q;
 return(compl_val);
}
```

To be able to use the values of θ_1 and θ_2 in the QR decomposition, the representation of these values have changed from a decision vector into a complex value. This can be achieved by rotating the value 1 ($= 1 + i0$), see figure 5.7.

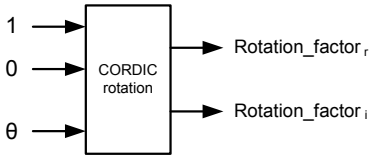


Figure 5.7: CORDIC in *rotation mode*

Accuracy

The LSB values in the paragraphs about the CORDIC algorithm and the QR decomposition are defined by the LSB of a 32-bit value, since all operations are executed on 32-bit dwords. In the paragraph about the QR algorithm the value of the LSB is defined by the LSB of a 16-bit value.

CORDIC The gain correction in `cordic_vec()` introduces a worst-case error of 2LSBs. The rotation to the first or fourth quadrant increases the worst-case error to 3LSBs. The number of iterations of the loop is 32. Therefore, the result of the loop increases the worst-case error by 1LSB (see [3]) to a final worst-case error of 4LSBs.

In function `cordic_rot()`, the gain correction introduces a worst-case error of 2LSBs. The loop increases the worst-case to 3LSBs.

The gain correction in function `cordic_rot2()` introduces a worst-case error of 2LSBs. The rotation to the start quadrant of the cordic rotation increases the worst-case error to 3LSBs. The loop increases the worst-case error to 4LSBs.

QR decomposition The values of θ_1, θ_2, and $-\theta_2$ are calculated by a CORDIC algorithm containing a worst-case error of 4LSBs. The two off-diagonal nonzero elements of matrix Qi, j are calculated by a complex multiplication. This complex multiplication increases the worst-case error of the values in $Q_{i,j}$ to 6LSBs.

The optimized matrix multiplication contains 2 complex multiplications and 1 addition per element of the result. Therefore, the optimized matrix multiplication increases worst-case error by 3LSBs. To calculate the complete QR decomposition 120 optimized multiplications have to be performed, which results in an increase of the worst-case error by 360LSBs to 366LSBs (8.5 bits). The elements of the resulting Q and R matrices are stored in *vector mode* in the memory of the Montium2. The 32-bit elements are truncated to 16-bit elements (the 16MSBs are selected), which result in a worst-case error of 1LSB of a 16-bit value.

QR algorithm As of this paragraph the LSB is again defined by the LSB of a 16-bit value. The matrix multiplication is calculated by 16 multiplications

and 15 additions. An element of the result of this matrix multiplication is calculated containing a worst-case error of $2 + 15 = 17$LSBs.

An iteration of the loop increases the worst-case error by $1 + 17 = 18$LSBs. The number of iterations of the loop is 8. This is a result of an empirical observation, with the value 0.001 used as the maximum difference between the eigenvalues of two iteration. The value 0.001 resulted in a stable number of iterations, whereas a smaller value increases the number of iterations and increases the variance of the number of iterations, without increasing the accuracy of the eigenvalues. A larger value decreases the accuracy of the eigenvalues.

The total worst-case error of the QR algorithm is $8 \times 18 = 144$LSBs, which is just over 7 bits. This result justifies the choice for a 32-bit multiplication in the QR decomposition, since a 16-bit multiplication results in a worst-case error of $8(366 + 17) = 3064$LSBs, which is more than 11,5 bits.

Computational load

The estimation of the computational load of the CORDIC algorithm, the QR decomposition, and the QR algorithm are considered separately.

CORDIC The scaling operations before and after the function `cordic_vec()` require 5 clock cycles. The function `cordic_vec()` start with a 32-bit complex multiplication. The computational load of this multiplication is not known, because it is not implemented in Montium2C code. Therefore, the computational load of the 32-bit complex multiplication is denoted by x. The assign operations outside the loop require 2 clock cycles. The rotation of the complex value from the second or third quadrant to the first of fourth quadrant, requires 5 clock cycles. However, if the complex value is already in the first or fourth quadrant, a compare and assign operation still have to be executed, requiring 2 clock cycles. It is assumed that the complex values are randomly distributed over all four quadrants. Therefore, the expected number of clock cycles is $(5 + 2)/2 = 3.5$, which is rounded to 4 clock cycles. Every iteration of the loop requires 4 clock cycles. The number of iterations is 32, so the all iterations require $32 \times 4 = 128$ clock cycles. The overall computational load of the function `cordic_vec` is $5 + x + 2 + 4 + 128 = 139 + x$ clock cycles.

In function `cordic_rot()` all assign operations outside the loop require 3 clock cycles in total. The 32-bit complex multiplication requires x clock cycles. Every iteration of the loop requires 3 clock cycles. The number of iterations is 32, so the all iterations require $32 \times 3 = 96$ clock cycles. The overall computational load of the function `cordic_rot` is $3 + x + 96 = 99 + x$ clock cycles.

In function `cordic_rot2()` all assign operations outside the conditional statements and loop require 3 clock cycles. The 32-bit complex multiplication requires x clock cycles. The expected number of clock cycles of the complete conditional statement outside the loop is rounded to 5 ($\approx (1 + 5 + 6 + 5 + 6)/5, which is the average of all possible situations$) in case of randomly distributed choices. The expected number of clock cycles of the complete conditional statement inside the loop is rounded to 5 ($\approx (4+5+5)/3$), if the choices are assumed to be randomly distributed. Every iteration of the loop requires $5 + 1 = 6$ clock cycles. The number of iterations is 32, so the all iterations re-

quire $32 \times 6 = 192$ clock cycles. The overall computational load of the function `cordic_rot2` is $3 + x + 192 = 195 + x$ clock cycles.

QR decomposition The calculation of θ_1 and θ_2 by means of the structure shown in figure 5.6 requires $3 \times (139 + x) + 33 = 450 + 3x$ clock cycles ($\theta_a - \theta_b$ requires 33 cycles). The calculation of rotation factor `theta1` requires $99 + x$ clock cycles, and calculation of rotation factor `theta2` requires $195 + x$ clock cycles. Rotation factor `theta2neg` is calculated out of `theta2` by conjugation, which requires 1 clock cycle. The calculation of matrix $Q_{i,j}$ requires $450+3x+99+x+195+x+1 = 745+5x$ clock cycles. To decompose 1 matrix, 120 $Q_{i,j}$ matrices have to be calculated, which requires $120(745 + 5x) = 89400 + 600x$ clock cycles.

One iteration of the loop of the optimized matrix multiplication requires 4 32-bit complex multiplication, 2 parallel assign operations, and 2 parallel additions. The number of iterations is 16, therefore the optimizes matrix multiplication requires $16(4 + 4x) = 64 + 64x$ clock cycles. The calculation of equation 5.3 requires 120 of these optimized matrix multiplications, and equation 5.4 requires 119 optimized matrix multiplications. The calculation of all matrix multiplication in both equations requires $(120+119)(64+64x) = 15296+15296x$ clock cycles. The complete QR decomposition requires $89400 + 600x + 15296 + 15296x = 104696 + 15896x$ clock cycles.

QR algorithm The function `matr_mult()` contains three nested loops of 16 iterations each. The complete function requires $16 \times 16 \times (2 + 16 \times 1) = 4608$ clock cycles. The function `matr_mult2()` requires 256 additional clock cycles, and therefore $4608 + 256 = 4864$ clock cycles in total. The comparison of the eigenvalues requires $16 \times 6 = 96$ clock cycles. Every iteration of the loop requires $4608 + 104696 + 15896x + 4864 + 96 = 114264 + 15896x$ clock cycles. The number of iterations of the loop is 8.

The complete QR algorithm requires $104696 + 15896x + 256 + 8(114264 + 15896x) = 1019064 + 143064x$ clock cycles.

Scalability

This implementation of the QR algorithm is based on a 16 by 16 matrix, and it fits in the memory space of one Montium2. It is estimated that a QR algorithm based on a 22 by 22 matrix is the largest implementation of the QR algorithm that fits in the memory space of one Montium2, because two 22 by 22 matrices can be stored in one memory block of 1024 dwords, and therefore 10 matrices can be stored simultaneously. The maximum number of matrices that is stored simultaneously during the execution of the QR algorithm is 9. A QR algorithm based on larger matrices can be distributed over multiple Montium2s. Different parts of the matrices are distributed over different Montium2s. The QR decomposition is calculated at a cost of continuously distributing the rotation factors. The two matrix multiplications at the end of an iteration of the QR algorithm cause an exponential increase of the communication, because parts of the matrices are continuously transferred between the Montium2s.

5.2.3 MUSIC spectrum

In this section the implementation of the calculation of the MUSIC spectrum is discussed. The calculation of the MUSIC spectrum is the third step of the MUSIC algorithm. This implementation of the MUSIC algorithm presumes the array manifold to be stored in some external memory, because the covariance matrix calculation makes use of all memories during its computations. The array manifold is determined once, and never changed.

Implementation

The MUSIC spectrum is calculated using formula 3.2. In the next step the locations of the peaks of the MUSIC spectrum are selected. Therefore, if the inverse of the MUSIC spectrum is calculated, the location of the lows have to be selected to achieve the same result. To reduce the computational load, the division is not implemented. The inverse of the MUSIC spectrum is calculated by:

$$P_m(\theta)^{-1} = a^H(\theta) E_n E_n^H a(\theta) \tag{5.17}$$

The first step is the multiplication of the noise subspace matrix (E_n) and its hermitian adjoint (E_n^H).

$$F = E_n E_n^H \tag{5.18}$$

In this implementation 5 impinging sources are assumed, so the size of matrix E_n is 16 by 11 and the size of matrix E_n^H is 11 by 16. The size of the resulting matrix F is 16 by 16. Matrix F is calculated by:

```
for (i = 0; i < 16; i++)
{
 for (j = 0; j < 16; j++)
 {
  real = 0;
  imag = 0;
  for (k = 0; k < 11; k++)
  {
   c = compl_mul(En[k*16+j],En_hermitian[k+i*11]);
   real = add(real, c.real);
   imag = add(imag, c.imag);
  }
  F_matrix[j+i*16] = packhh(real,imag);
 }
}
```

Matrix E_n is stored in MEM2, matrix E_n^H is stored in MEM4, and the resulting matrix F is stored in MEM0.

As a consequence of the scaling operation in the covariance matrix calculation, the accumulated value in matrix multiplication described above does not become larger than 1. Therefore, the values are accumulated in *32-bit dword mode*. Matrix F is calculated only once.

The final implemented equation to calculate the MUSIC spectrum is:

$$P_m(\theta)^{-1} = a^H(\theta) F a(\theta) \tag{5.19}$$

This equation is implemented by:

```
for (i = 0; i < 181; i++)
{
 load_array_man_vector(i);
 for (j = 0; j < 16; j++)
 {
  acc_real = 0;
  acc_imag = 0;
  for (k = 0; k < 16; k++)
  {
   c = compl_mul(array_man_hermitian[k],F_matrix[k+j*16]);
   acc_real = add_a(acc_real, c.real);
   acc_imag = add_a(acc_imag, c.imag);
  }
  real = asr_a(acc_real, 4);
  imag = asr_a(acc_imag, 4);
  temp_result[j] = packhh(real,imag);
 }
 acc_real = 0;
 for (idx j = 0; j < 16; j++)
 {
  c = compl_mul(temp_result[j],array_man[j]);
  acc_real = add_a(acc_real, c.real);
 }
 real = asr_a(acc_real, 4);
 P_music[i] = real;
}
```

The function `load_array_man_vector()` loads a vector of the array manifold ($a(\theta)$), and stores it in the first 16 dwords of MEM1. The function also conjugates the values, to calculate $a^H(\theta)$, and stores them in the first 16 dwords of MEM4. Vector `array_man_hermitian[]`, representing $a^H(\theta)$, is multiplied by matrix F. The values are scaled to keep the values between -1 and 1, before they are combined in one dword and stored in the resulting 16 element vector `temp_result[]` (MEM2). The vector `temp_result[]` and vector `array_man[]` (representing $a(\theta)$) are multiplied. The resulting value is scaled and stored in `P_music[]`, the MUSIC spectrum. Only the real part is stored, because the imaginary part is always zero. The MUSIC spectrum is stored in MEM3. All scaling values in this section are a result of an empirical observation.

Accuracy

To calculate an element of matrix F, 11 complex multiplications and 10 addition are performed. A complex multiplication introduces a worst-case error of 2LSBs. The 10 additions in the accumulator increases the worst-case error by 10LSBs. An element of matrix F is calculated with a worst-case error of 12LSBs.

The calculation of an element of the MUSIC spectrum, is performed by multiplying vector $a^H(\theta)$ and matrix F. The resulting vector is multiplied to vector $a(\theta)$. The elements of $a(\theta)$ and $a^H(\theta)$ contain a worst-case error of $\frac{1}{2}$LSB. The elements of F contain a worst-case error of 12LSBs. To calculate

one element of the result of the $a^H(\theta)F$ multiplication, 16 complex multiplications and 15 additions are performed. A complex multiplication of an element of $a^H(\theta)$ and an element of F result in a worst-case error of 14LSBs. The 15 additions increases the worst-case error to $14+15 = 29$LSBs. The elements of the resulting vector of the $a^H(\theta)F$ multiplication are divided by 16, which result in worst-case error of $\lceil 29/16 \rceil = 2$LSBs. This resulting vector is multiplied by $a(\theta)$ using 16 complex multiplications and 15 additions. The complex multiplication and additions increase the worst-case error to $2+2+15 = 19$LSBs. The last scaling operation reduces the worst-case error to $\lceil 19/16 \rceil = 2$LSBs. Therefore, an element of the MUSIC spectrum is calculated containing a calculation error of 2LSBs.

Computational load

In this implementation the DOA of 5 sources have to be estimated. Therefore, E_n is a 16 by 11 matrix. The calculation of matrix F requires $16\times11\times16 = 2816$ mac operations.

Equation 5.19 is a vector-matrix-vector multiplication, and is calculated for every angle of the spectrum. The calculation of one point in the spectrum requires $1 \times 16 \times 16 + 16 = 272$ multiplications.

The spatial resolution of the MUSIC algorithm is 1 degree when a 16 element ULA and 1024 snapshots are used (see the simulation results in appendix A.1). The complete spectrum consists of all angles from -90 degrees to 90 degrees in steps of 1 degree. A total of 181 points have to be calculated. The calculation of the spectrum requires $181 \times 272 = 49232$ multiplications. The complete calculation of the MUSIC spectrum requires $2816 + 49232 = 52048$ clock cycles, excluding overhead.

Scalability

The calculation of the MUSIC spectrum could be distributed over multiple Montium2s, at the cost of distributing the complete matrix F over these Montium2s. Each Montium2 can calculate a different part of the spectrum using equation 5.19. This option is usefull if the spectrum contains many angles.

If matrix F does not fit into the memory space of one Montium2, the communication between the Montium2s increases exponentially, because the calculation of equation 5.19 is divided over multiple Montiums.

5.2.4 Peak selection

In this section the implementation of the selection of the peaks in the MUSIC spectrum is discussed. The selection of the peaks is the fourth step of the MUSIC algorithm.

Implementation

In the previous part of the MUSIC algorithm the inverted spectrum is calculated. Therefore, instead of the peaks, the lows have to be selected.

The first step is de detection of all local minimums in the spectrum. All spectrum values of the angles in the spectrum are compared with their neighbours. A local minimum is detected if the spectrum value of the previous angle

and the next angle are greater than the spectrum value of the current angle. All local minimums are stored in an array. The detection of the local minimums starts with a corner case.

```
peak_number = 0;
if (!(cmplt(P_music[1],P_music[0])))
{
 peak_angle[peak_number] = 0;
 peak_size[peak_number] = P_music[0];
 peak_number = add(peak_number, 1);
}
```

The first element of the MUSIC spectrum is compared to the second element of the MUSIC spectrum. If the second element is not less than the first element, the location and the size of the first element are stored in respectively, `peak_angle[]` (MEM2) and `peak_size[]` (MEM4). The variable `peak_number` stores the location where the next local minimum can be stored in `peak_angle[]` and `peak_size[]`.

```
for (i = 1; i < 180; i++)
{
 if (cmpgt(P_music[i-1],P_music[i]))
 {
  if (!(cmplt(P_music[i+1],P_music[i])))
  {
   peak_angle[peak_number] = i;
   peak_size[peak_number] = P_music[i];
   peak_number = add(peak_number, 1);
  }
 }
}
```

The general case of the local minimum detection is described above. All elements of the MUSIC spectrum, except for the first and last element, are compared to their neighbours. If an element is smaller than both neighbours, the location and size is stored in `peak_angle[]` and `peak_size[]`.

The second corner case is the comparison of the second last and the last element of the MUSIC spectrum.

```
if (cmpgt(P_music[179],P_music[180]))
{
 peak_angle[peak_number] = 180;
 peak_size[peak_number] = P_music[180];
}
```

If the second last element is greater than the last element, the location and the size is stored in `peak_angle[]` and `peak_size[]`.

This implementation has to detect a predefined number of sources, in this case 5 sources. The second step is to select the 5 smallest minimums. All local minimums are compared to each other, and the 5 smallest are stored in an array. These 5 lows are the DOAs of the 5 sources.

```
for (i = 0; i < 5; i++)
{
 source_size[i] = 0x7FFFFFFF;
}
```

`source_size[]` is a 5 element array, and is initialized with the largest value possible in signed fixed point Q(0,15) format.

```
for (i = 0; i < peak_number; i++)
{
 temp_size = peak_size[i];
 temp_angle = peak_angle[i];
 for (j = 0; j < 5; j++)
 {
  if (cmplt(temp_size, source_size[j]))
  {
   swap_size = source_size[j];
   swap_angle = source_angle[j];
   source_size[j] = temp_size;
   source_angle[j] = temp_angle;
   temp_size = swap_size;
   temp_angle = swap_angle;
  }
 }
}
```

The size of the 5 smallest local minimums are stored in `source_size[]`. The locations of the 5 smallest local minimums are stored in `source_angle[]`. The 5 smallest minimums are selected by comparing the sizes of the minimums and storing them using an insertion sort algorithm.

The last step of the peak selection, and therefore of the MUSIC algorithm, is the calculation of the DOAs of the sources. These directions are calculated by subtracting the value 90 from the locations of the sources.

```
for (i = 0; i < 5; i++)
{
 source_angle[i] = sub(source_angle[i], 90);
}
```

The estimated DOAs in degrees are stored in `source_angle[]`.

Computational load

The detection of the local minimums in the spectrum requires $2 \times 179 + 2 = 360$ compare operations (the two outer values of the spectrum are compared to one neighbour). An empirical observation stated that on average 11 local minimums are detected. The selection of the 5 smallest minimums requires $5 \times 11 = 55$ compare operations. The selection of all 5 DOAs is requires on average 415 clock cycles without overhead.

Scalability

The peak selection part of the MUSIC algorithm scales linear if it is distributed over multiple Montium2s. A different part of the spectrum is assigned to each Montium2. All Montium2s select the 5 peaks of their part of the spectrum. One Montium2 has to select the largest peaks of the results of all Montium2s.

5.3 Conclusion

The implementation of the MUSIC algorithm on the Montium2 is described in this chapter. An estimation is made of the execution time in clock cycles of each part of the MUSIC algorithm. In table 5.2 the clock cycles of the different parts are summed to estimate the computational load of the complete MUSIC algorithm. In the eigendecomposition a 32-bit complex multiplier is used. Since the 32-bit complex multiplication is not implemented in Montium2C code, the value of x is not known. The architecture of the Montium2 is still under construction. Therefore, if it is assumed that the four 16-bit multipliers (2 M units) of the Montium2 are extended to 32-bit multipliers, the value of $x = 1$ (i.e. every clock cycle a result of a complex multiplication is calculated). In this case the minimal number of clock cycles needed, to execute the implementation described above, can be calculated.

Algorithm	**Clock cycles**	**%**
covariance matrix	283.356	18.92
eigendecomposition	1.162.128	77.58
music spectrum	52.048	3.47
peak selection	415	0.03
total	1.497.947	100

Table 5.2: Clock cycles of the MUSIC algorithm.

The complete algorithm is executed in 1.497.947 clock cycles. The clock frequency of the Montium2 is targeted at 200MHz. The execution time of the complete algorithm is 1.497.947/200.000.000 = 7.5ms. To achieve this result, it is assumed that the complete algorithm will fit into the program memory of the Montium2. The size of the program memory is still unknown, therefore it cannot be said if it is a reasonable assumption. Although the implementation contains many loops, which reduces the number of instructions, and therefore the size of the program memory needed to store the instructions.

The results described above do not contain any (loop, pipeline, etc) overhead. Because the Montium2 is still under construction, it is hard to estimate the total amount of overhead.

In section 2.5 the shortest execution time of the described implementations is 28μs. This implementation is based on a 4 element ULA and 2 FPGAs. To be able to compare this implementation with the implementation described in this chapter, the execution time of the implementation in this chapter is roughly estimated for a 4 element ULA. A 4 element ULA results in 4 by 4 matrices during all calculations. A 4 by 4 matrix is 16 times smaller than a 16 by 16 matrix. In the QR decomposition of a 16 by 16 matrix, 120 $Q_{i,j}$ matrices are calculated and multiplied. This number reduces in the calculation

of a QR decomposition of a 4 by 4 matrix to 6 $Q_{i,j}$ matrix calculations and multiplication. Because of the optimized matrix multiplications, a reduction of the size of the matrix scales linear with the number of computations, a factor 4 in this case. The execution time of the eigendecomposition is determined almost only by the execution time of the QR decomposition. So if the execution time of the QR decomposition is $120/6 = 20$ times shorter, the execution time of the eigendecomposition is 20 times shorter. The eigendecomposition consumes 77.6% of the execution time. Therefore it reduces the total execution time to roughly $0.224 \times 7500/16 + (0.776 \times 7500/4/20) = 178\mu$s. This execution time is about 6 times longer than the execution time of the implementation on 2 FPGAs.

The advantage of the Montium2 with respect to an FPGA is the faster reconfiguration of the application. Another advantage of the Montium2 is the estimated energy consumption. Because the size of the Montium2 is limited to 2mm^2, it is estimated that the energy figures of the Montium2 architecture are more than 6 times lower than the energy figures of 2 FPGAs (EP20K600, Altera).

Practical example

Assume that a beamforming system for satellite television reception, containing a 16 element ULA, is installed on the roof of a car. In this beamforming system one Montium2 is reserved to execute the implementation of the MUSIC algorithm described in this thesis. One iteration of the complete implementation of the MUSIC algorithm on the Montium2 takes 7.5ms. To be able to track the sources a maximum rotation of the car of 1° is allowed in these 7.5ms. This maximal rotation is the spatial resolution of the MUSIC algorithm based on a 16 element ULA and 1024 snapshots. The maximum rotation of the car is limited at 133° per second, which is enough for normal use of the car.

Chapter 6

Conclusion and Recommendations

6.1 Conclusion

Two popular DOA estimation algorithms, MUSIC and ESPRIT, are analyzed. Both algorithms contain an eigendecomposition. Since the eigendecomposition is the part containing the largest order of complexity in both algorithms, the order of complexity of both algorithms is the same.

MUSIC and ESPRIT are compared by means of a test case. The results of the comparison showed no significant differences in the number of correct estimations when the number of antennas are changed. In case of low SNR, the MUSIC algorithm showed less estimation errors in the result of the simulations, with respect to the ESPRIT algorithm. When the signals are more correlated, ESPRIT showed less estimation errors, except for situations where the SNR is low, because influence of the SNR becomes more dominant.

The MUSIC algorithm is implemented on the Montium2 architecture. MUSIC contains many vector and matrix multiplications. The QR decomposition is based on many iterations of the CORDIC algorithm. Therefore the kernels of the MUSIC algorithm are the CORDIC algorithm and matrix multiplications.

The execution time of the implementation described in chapter 5 is 7.5ms on one Montium2. The practical example showed that it is acceptable execution time when the Montium2 is completely dedicated to execute the MUSIC algorithm. Therefore, there is no time left for an interleaved execution with other parts of the beamforming system. To create the possibility of an interleaved execution, and therefore employ the purpose of a reconfigurable architecture, the execution time has to be reduced.

6.2 Recommendations

Optimizing the eigendecomposition

The eigendecomposition is responsible for 78% of the execution time of the MUSIC algorithm. Optimizations in the eigendecomposition should reduce the amount of contribution. The CORDIC algorithm consumes a large portion of the execution time of the eigendecomposition. Therefore the possibilities of

an other algorithm for conversion from polar form to Cartesian form and vice versa should be investigated.

The QR decompostion can also be optimized by first transforming the covariance matrix into an upper Hessenberg matrix [23]. An upper Hessenberg matrix is almost an upper triangular matrix. The calculation of such a matrix is often less computational expensive.

The covariance matrix contains complex values. By applying a unitary transform [12], the covariance matrix can be transformed into a real symmetric matrix, which at least halves the number of multiplications in the eigendecomposition. If all 4 multipliers in the Montium2 could be used to calculate 4 real multiplications simultaneously, the number of multiplications is divided by 4. This is not a straight forward solution because of the grouping of the multipliers in the Montium2.

If the QR decomposition is calculated in a multi processor architecture, the Jacobi Method could increase the performance of a parallel implementation.

To increase the accuracy of the eigendecomposition, without increasing the computational load, the 16-bit multipliers have to be extended to 32-bit multipliers in the Montium2.

Optimizing the covariance matrix calculation

The implementation of the CMD extension could reduce the computational load of the covariance matrix calculation and increase the number of correct DOA estimations of MUSIC when the sources are correlated.

Appendix A

Simulation results

A.1 MUSIC and ESPRIT

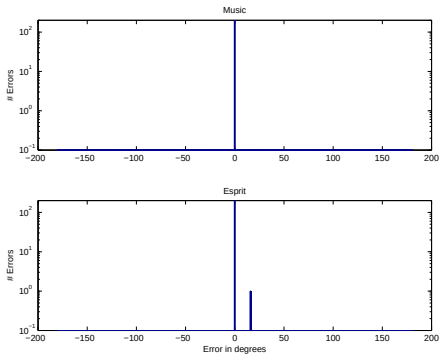


Figure A.1: 32 antennas, 2048 snapshots, 60dB SNR.

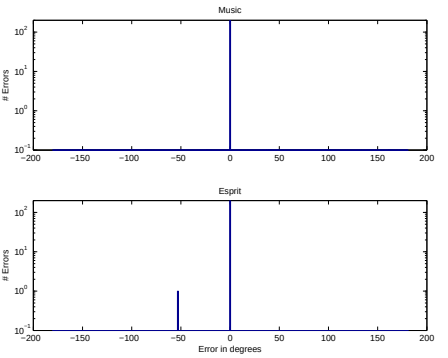


Figure A.2: 32 antennas, 2048 snapshots, 40dB SNR.

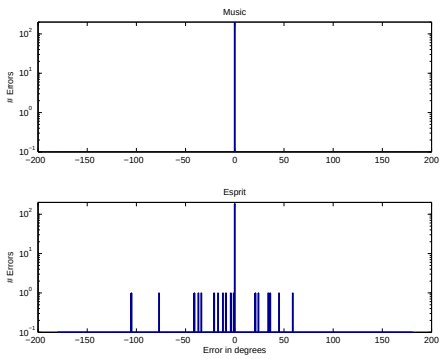


Figure A.3: 32 antennas, 2048 snapshots, 20dB SNR.

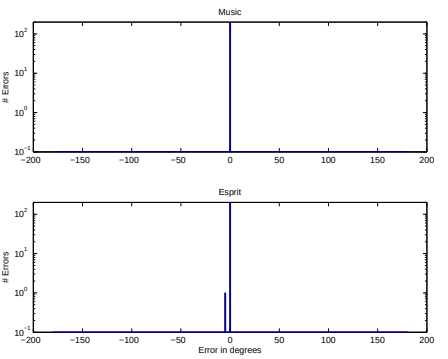


Figure A.4: 32 antennas, 1024 snapshots, 60dB SNR.

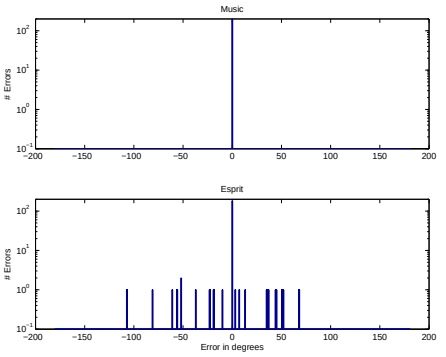


Figure A.5: 32 antennas, 1024 snapshots, 40dB SNR.

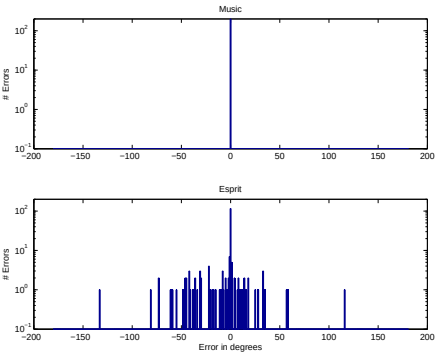


Figure A.6: 32 antennas, 1024 snapshots, 20dB SNR.

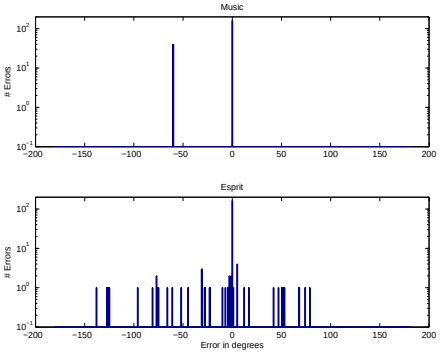


Figure A.7: 32 antennas, 512 snapshots, 60dB SNR.

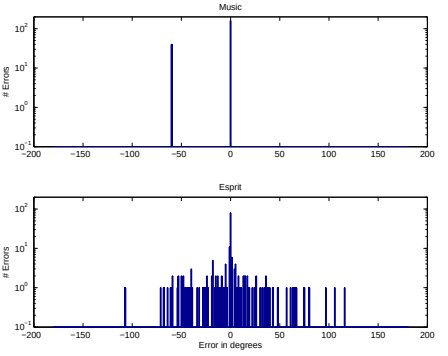


Figure A.8: 32 antennas, 512 snapshots, 40dB SNR.

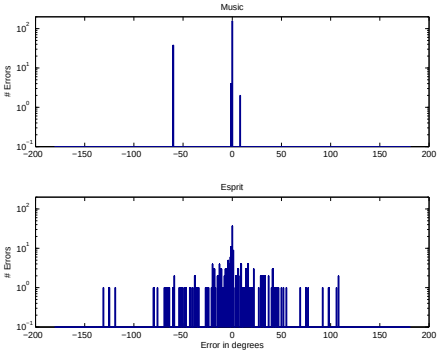


Figure A.9: 32 antennas, 512 snapshots, 20dB SNR.

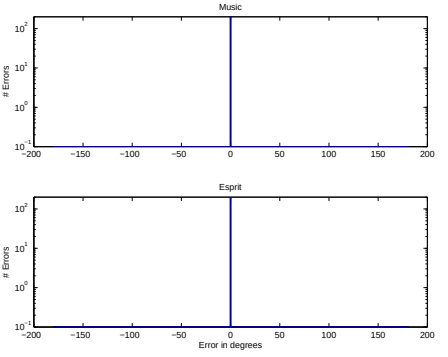


Figure A.10: 16 antennas, 2048 snapshots, 60dB SNR.

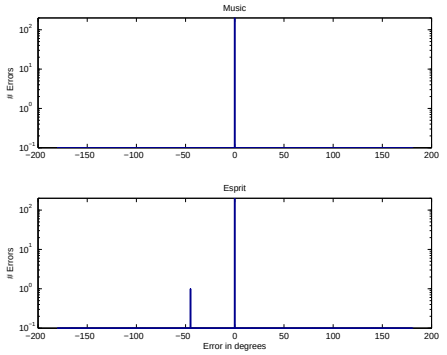


Figure A.11: 16 antennas, 2048 snapshots, 40dB SNR.

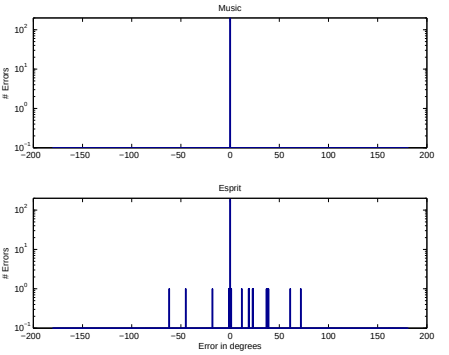


Figure A.12: 16 antennas, 2048 snapshots, 20dB SNR.

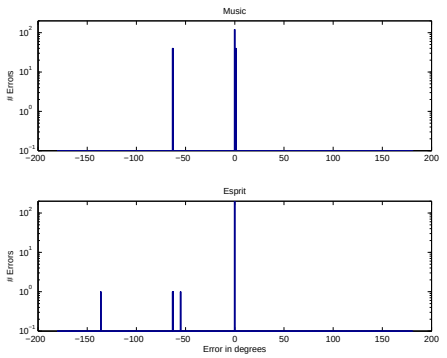


Figure A.13: 16 antennas, 1024 snapshots, 60dB SNR.

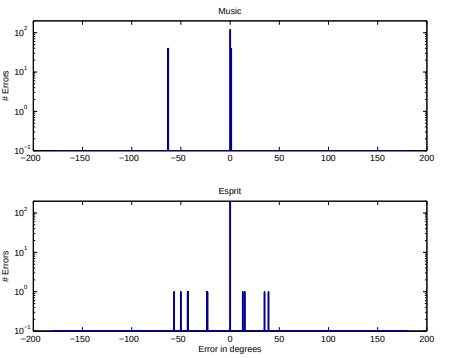


Figure A.14: 16 antennas, 1024 snapshots, 40dB SNR.

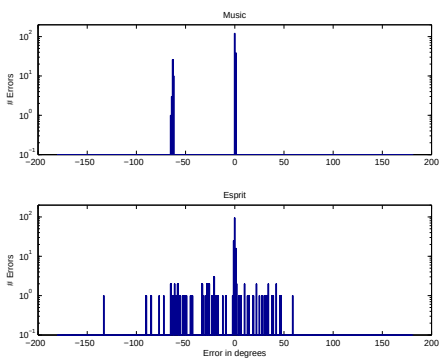


Figure A.15: 16 antennas, 1024 snapshots, 20dB SNR.

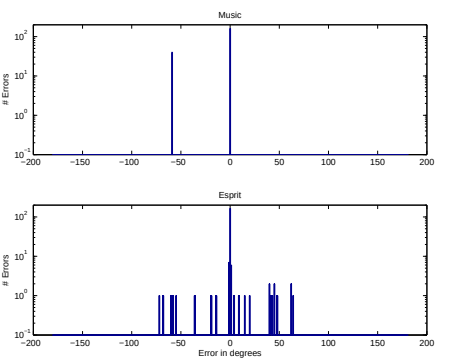


Figure A.16: 16 antennas, 512 snapshots, 60dB SNR.

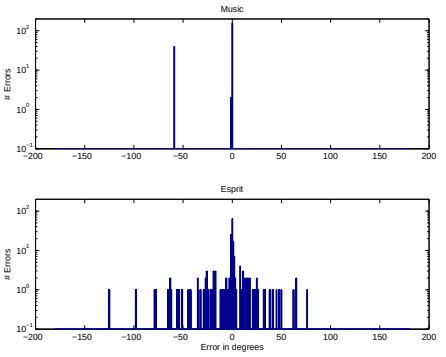


Figure A.17: 16 antennas, 512 snapshots, 40dB SNR.

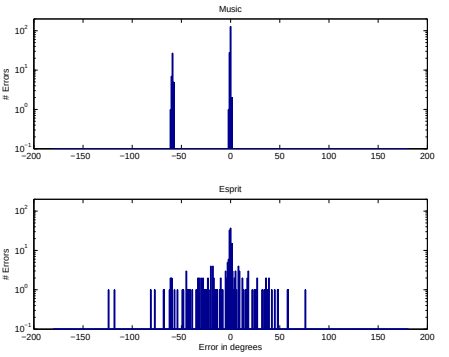


Figure A.18: 16 antennas, 512 snapshots, 20dB SNR.

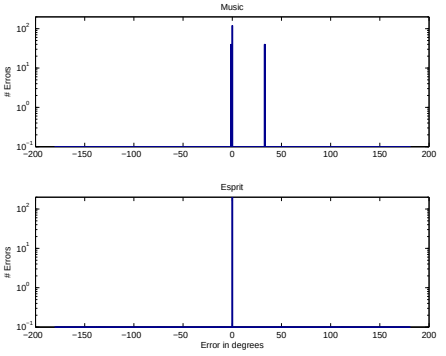


Figure A.19: 8 antennas, 2048 snapshots, 60dB SNR.

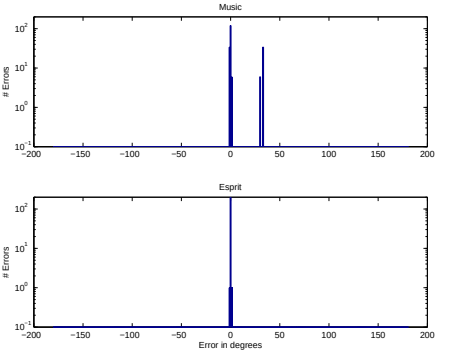


Figure A.20: 8 antennas, 2048 snapshots, 40dB SNR.

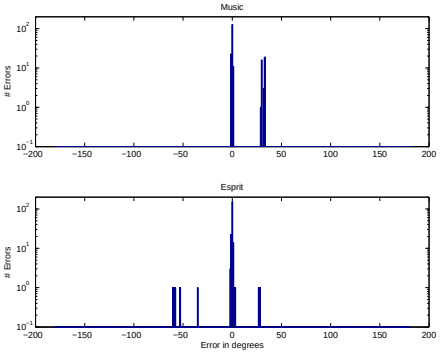


Figure A.21: 8 antennas, 2048 snapshots, 20dB SNR.

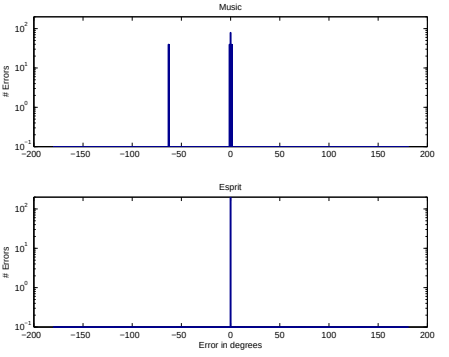


Figure A.22: 8 antennas, 1024 snapshots, 60dB SNR.

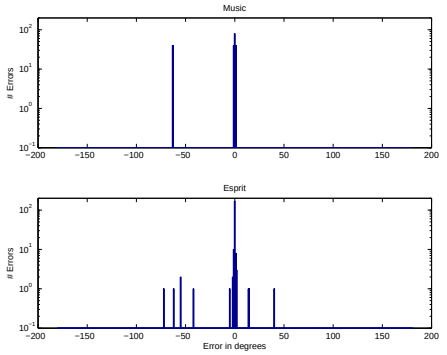


Figure A.23: 8 antennas, 1024 snapshots, 40dB SNR.

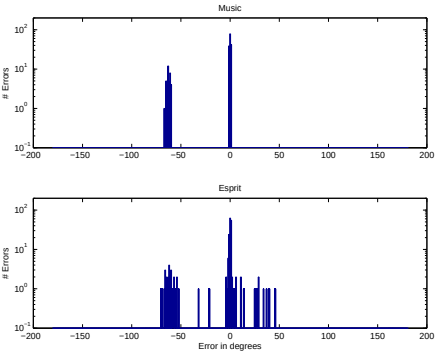


Figure A.24: 8 antennas, 1024 snapshots, 20dB SNR.

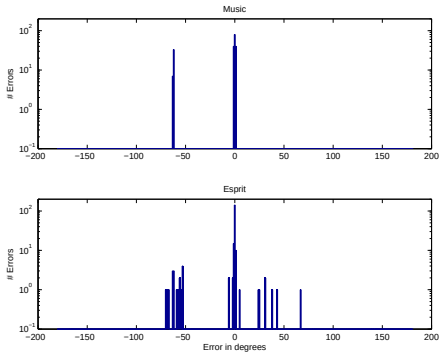


Figure A.25: 8 antennas, 512 snapshots, 60dB SNR.

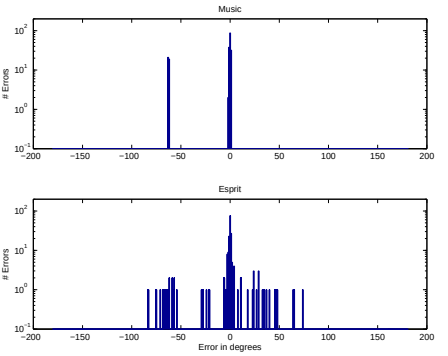


Figure A.26: 8 antennas, 512 snapshots, 40dB SNR.

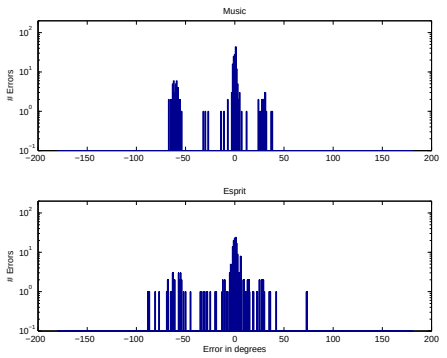


Figure A.27: 8 antennas, 512 snapshots, 20dB SNR.

A.2 CMD MUSIC

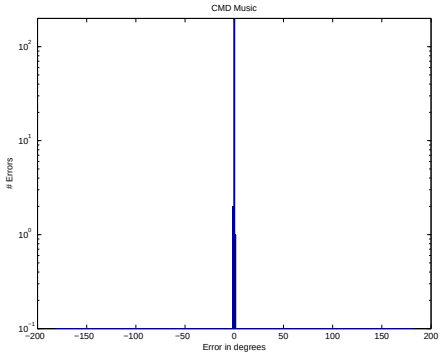


Figure A.28: 512 snapshots, 20dB SNR.

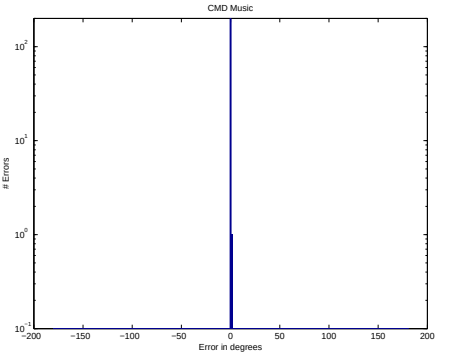


Figure A.29: 256 snapshots, 20dB SNR.

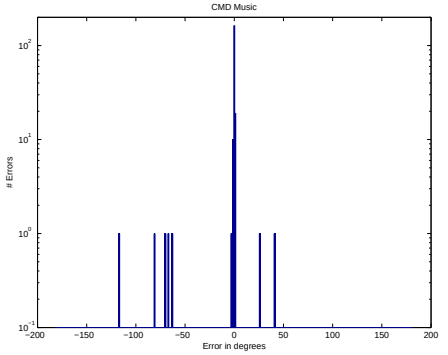


Figure A.30: 256 snapshots, 10dB SNR.

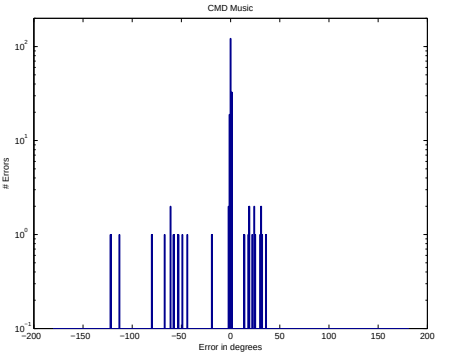


Figure A.31: 256 snapshots, 5dB SNR.

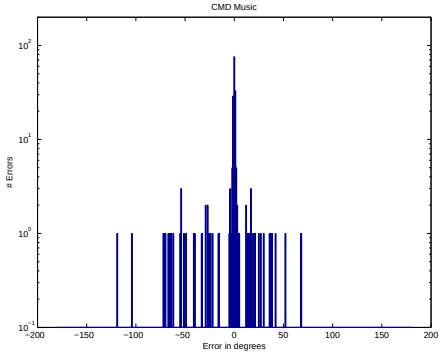


Figure A.32: 256 snapshots, 0dB SNR.

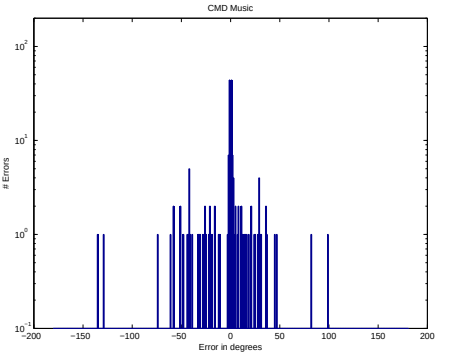


Figure A.33: 256 snapshots, -5dB SNR.

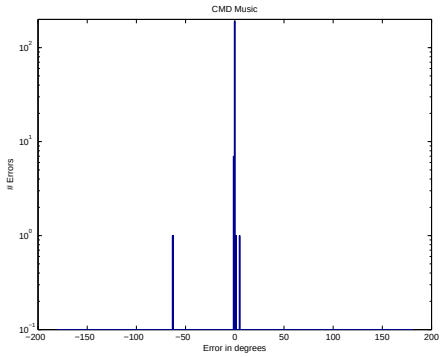


Figure A.34: 128 snapshots, 20dB SNR.

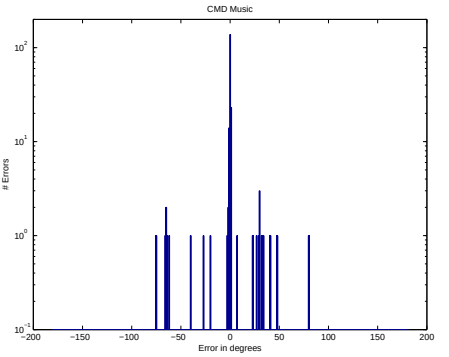


Figure A.35: 128 snapshots, 10dB SNR.

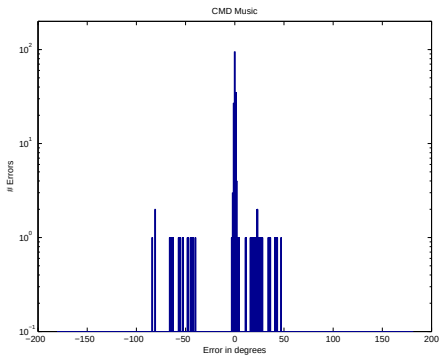


Figure A.36: 128 snapshots, 5dB SNR.

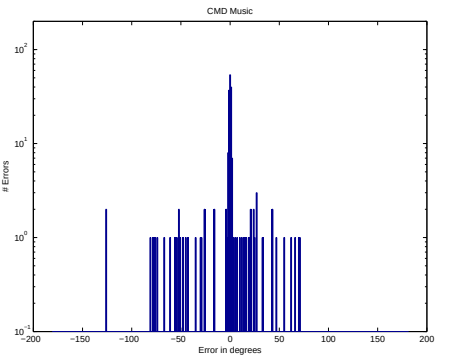


Figure A.37: 128 snapshots, 0dB SNR.

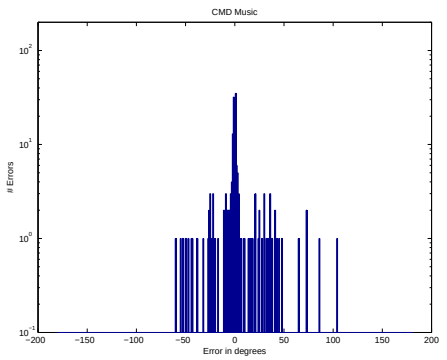


Figure A.38: 128 snapshots, -5dB SNR.

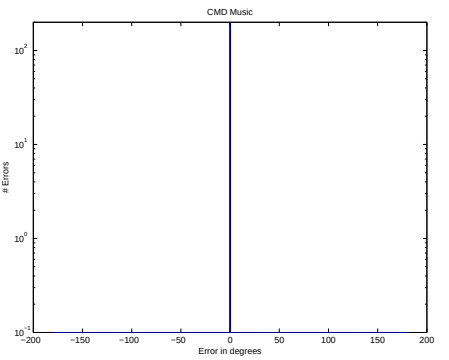


Figure A.39: 64 snapshots, 20dB SNR.

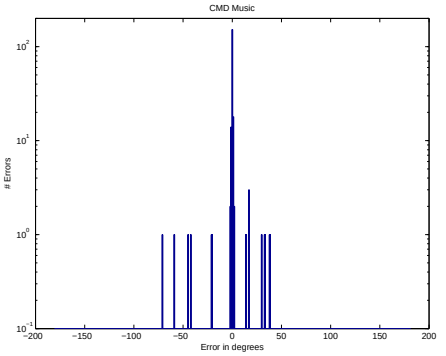


Figure A.40: 64 snapshots, 10dB SNR.

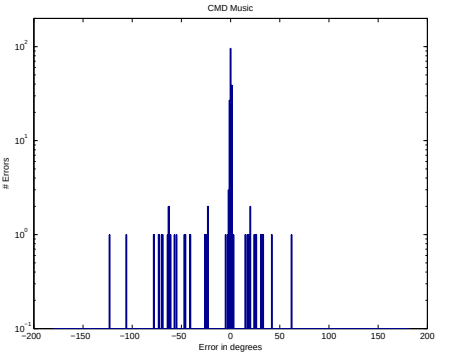


Figure A.41: 64 snapshots, 5dB SNR.

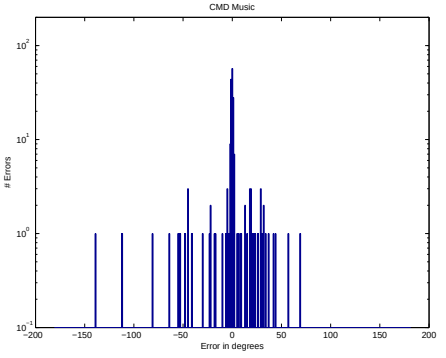


Figure A.42: 64 snapshots, 0dB SNR.

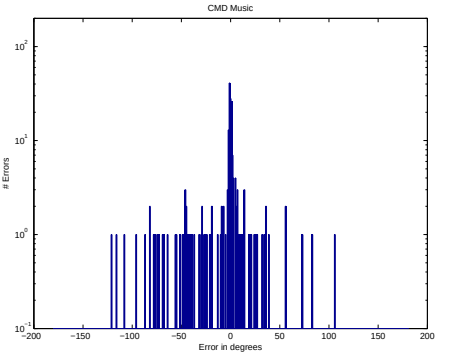


Figure A.43: 64 snapshots, -5dB SNR.

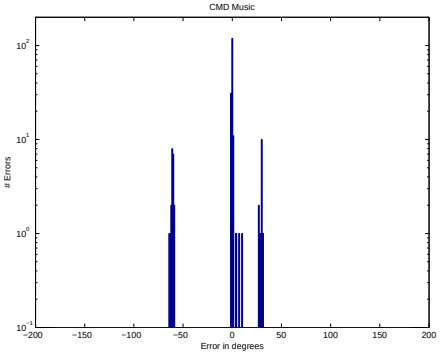


Figure A.44: 32 snapshots, 20dB SNR.

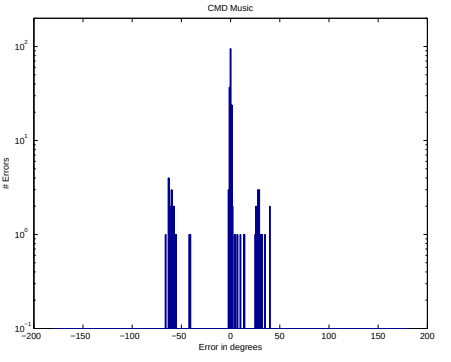


Figure A.45: 32 snapshots, 10dB SNR.

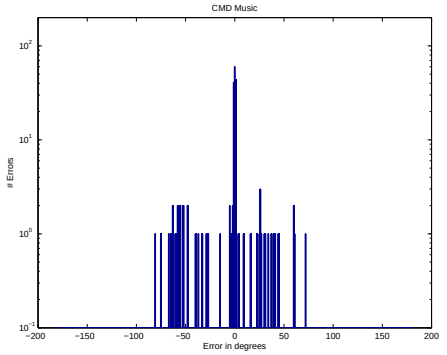


Figure A.46: 32 snapshots, 5dB SNR.

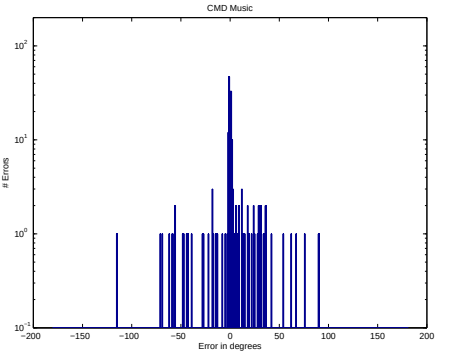


Figure A.47: 32 snapshots, 0dB SNR.

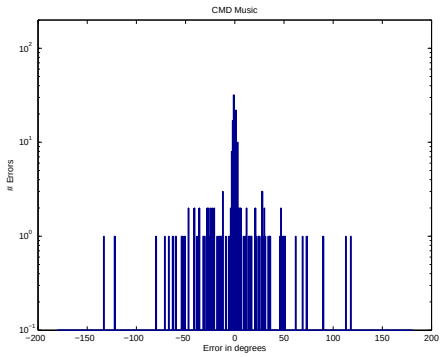


Figure A.48: 32 snapshots, -5dB SNR.

Bibliography

[1] Ziad Al-Qadi and Musbah Aqel. Performance analysis of parallel matrix multiplication algorithms used in image processing. In *World Applied Sciences Journal*, volume 6, pages 45–52. IDOSI Publications, 2009.

[2] M.S. Oude Alink. Increasing the spurious-free dynamic range of an integrated spectrum analyzer. Master's thesis, University of Twente, Enschede, The Netherlands, November 2008.

[3] Ray Andraka. A survey of cordic algorithms for fpga based computers. In *Proceedings of the 1998 ACM/SIGMA sixth international symposium on Field Programmable Gate Arrays*, pages 191–200, February 1998.

[4] Burton S. Garbow. Algorithm 535: The qz algorithm to solve the generalized eigenvalue problem for complex matrices [f2]. *ACM Trans. Math. Softw.*, 4(4):404–410, 1978.

[5] Lal Chand Godora. *Smart Antennas*. CRC Press LLC, January 2004.

[6] M. Glesner H. Wang. Hardware implementation of smart antenna systems. In *Advances in Radio Science*, volume 4, pages 185–188, September 2006.

[7] Nariankadu D. Hemkumar. A systolic vlsi architecture for complex svd. Master's thesis, Rice University, Houston, Texas, May 1991.

[8] Linda Kaufman. Some thoughts on the qz algorithm for solving the generalized eigenvalue problem. *ACM Trans. Math. Softw.*, 3(1):65–75, 1977.

[9] David C. Lay. *Linear Algebra and Its Applications*. Addison-Wesley Publishing Company, second edition, April 2000.

[10] K. Arai H Minseok, K. Ichige. Implementation of fpga based fast doa estimator using unitary music algorithm. In *Vehicular Technology Conference*, volume 1, pages 213–217, October 2003.

[11] Peter J. Olver. Orthogonal bases and the qr algorithm. Technical report, 2006.

[12] M. Pesavento, A.B. Gershman, and M. Haardt. Unitary root-music with a real-valued eigendecomposition: a theoretical and experimental performance study. 48(5):1306–1314, 2000.

[13] Rik Portengen. Phased array antenna processing on reconfigurable hardware. Master's thesis, University of Twente, Enschede, The Netherlands, December 2007.

[14] R Roy and T Kailath. Esprit - estimation of signal parameters via rotational invariance techniques. In *IEEE Transactions on Acoustics, Speech, and Signal Processing*, volume 37, pages 984–995, July 1989.

[15] R Roy, A Paulraj, and T Kailath. Esprit - a subspace rotation approach to estimation of parameters of cisoids in noise. In *IEEE Transactions on Acoustics, Speech, and Signal Processing*, volume ASSP-34, pages 1340–1342, October 1986.

[16] R. Schmidt. Multiple emitter location and signal parameter estimation. In *IEEE Transactions on Antennas and Propagation*, volume AP-34, pages 276–280, March 1986.

[17] Gautam M. Shroff. A parallel algorithm for the eigenvalues and eigenvectors of a general complex matrix. In *Numer. Math.*, volume 58, pages 779–805, 1991.

[18] Recore Systems. Crisp project. `http://www.crisp-project.eu/`, Last checked: 1-15-2009.

[19] Recore Systems. *Montium User Guide*. 2008. Version 2.

[20] Nizar Tayem, Hyuck M. Kwon, Seunghyun Min, and Dong Hee Kang. Covariance matrix differencing for coherent source doa estimation under unknown noise field. In *Proc. VTC-2006 Fall Vehicular Technology Conference 2006 IEEE 64th*, pages 1–5, 2006.

[21] Various. Coordinate rotation digital computer. `http://en.wikipedia.org/wiki/CORDIC`, Last checked: 1-13-2009.

[22] Various. Eigendecomposition of a matrix. `http://en.wikipedia.org/wiki/Eigendecomposition_of_a_matrix`, Last checked: 1-13-2009.

[23] Various. Hessenberg matrix. `http://en.wikipedia.org/wiki/Hessenberg_matrix`, Last checked: 1-13-2009.

[24] Various. Jacobi eigenvalue algorithm. `http://en.wikipedia.org/wiki/Jacobi_eigenvalue_algorithm`, Last checked: 1-13-2009.

[25] Hubregt J. Visser. *Array and Phased Array Antenna Basics*. John Wiley and Sons, 2005.

[26] Shaoyun Wang and Jr. Swartzlander, E.E. The critically damped cordic algorithm for qr decomposition. In *Conference Record of the Thirtieth Asilomar Conference on Signals, Systems and Computers*, pages 908–911 vol.2, November 1996.

[27] David S. Watkins. Understanding the qr algorithm. In *SIAM Review*, volume 24, pages 427–440. Society for Industrial and Applied Mathematics, October 1982.

[28] David S. Watkins. *Fundamentals of Matrix Computations*. John Wiley and Sons, 2002.

[29] J. Wu, W.-X. Sheng, K.-P. Chan, W.-K. Chung, K.-K.M. Cheng, and K.-L. Wu. Smart antenna system implementation based on digital beam-forming and software radio technologies. In *Proc. IEEE MTT-S International Microwave Symposium Digest*, volume 1, pages 323–326, 2002.