Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelor Thesis

Muhammad Afaque Khan

Localization of a source in a room using Mirco-phone Arrays

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Muhammad Afaque Khan

Localization of a source in a room using Microphone Arrays

Bachelor Thesis based on the examination and study regulations for the Bachelor of Engineering degree programme
Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg

Supervising examiner: Prof. Dr. Ulrich Sauvagerd
Second examiner : Prof. Dr. Klaus Jünemann

Day of delivery July 17th 2017

Muhammad Afaque Khan

Title of the Bachelor Thesis

Localization of a speaker in a room by using microphone Arrays

Keywords

MCCC, Microphone array, Source localization, SDOA estimation

Abstract

In this thesis a Source Localization DSP program implemented on D.M.C6713 is analyzed and ported to D.M.C6657. The DSP program on D.M.C6657 is extended to take advantage of the better specifications of the new hardware.
A Source Localization progam is also developed in MATLAB that uses Multi-channel Cross Correlation Coefficient (MCCC) algorithm for SDOA estimation.

Muhammad Afaque Khan

Thema der Bachelorarbeit

Lokalisierung eines Sprechers in einem Raum mit Mikrofon Arrays

Stichworte

MCCC, Mikrofon-Array, Quelllokalisierung, SDOA-Schätzung

Kurzzusammenfassung

In dieser Arbeit wird ein auf D.M.C6713 implementiertes Source-Lokalisierungs-DSP-Programm analysiert und auf D.M.C6657 portiert. Das DSP-Programm auf D.M.C6657 wird erweitert, um die Vorteile der neuen Hardware zu nutzen.Ein Source-Lokalisierungs-Progam wird auch in MATLAB entwickelt, das den Multi-Channel Cross Correlation Coefficient (MCCC) Algorithmus für die SDOA-Schätzung verwendet.

Table of Contents

List of Figures

List of Tables

List of Codes

List of Abbreviations

SDOA	Sample Difference of Arrival
TDOA	Time Difference of Arrival
MCCC	Multiple Cross Correlation Coefficient
DOA	Direction of Arrival
McBSP	Multi-channel Buffered Serial Port

FFT	Fast Fourier Transformation
IFFT	Inverse Fast Fourier Transform
EDMA	Enhanced Direct Memory Access
DSP	Digital Signal Processing
CCS	Code Composer Studio
ADC	Analog Digital Filter
SDK	Software Development Kit
RTOS	Real-time operating system

List of Symbols

c	Speed of Sound
d_{M2M1}	Distance between two Microphone Arrays
d	Distance between two Microphones
f_s	Sampling Frequency
$fmax$	Maximum allowable frequency
N_{Block}	Length of Block
N_{FFT}	Length of FFT
p	control parameter for index of correlation for SDOA estimation
$\tilde{R}_a(p)$	Normalized Correlation matrix
$R_a(p)$	Correlation Matrix
$s(k)$	Signal of source
τ	Time Delay
$det(\cdot)$	Function for calculation of Determinant
$E[\cdot]$	Mean value operator
$Fn(\tau)$	Function for calculation of time delay to the reference Microphone
$\rho_{a,yiyj}(p)$	Correlation Coefficient of i-th und j-th Signals depending on value p
σ^2_{yn}	Variance of signal N
$Ra(p)$	Normalized Spatial Correlation matrix
$s(k)$	Source Signal
$vn(k)$	uncorrelated white noise
$xn(k)$	Input Signal at nth Microphone
$yn(k)$	Output signal at nth Microphone
αn	Damping factor
λ	Wave Length
m	meter

1. Introduction

1.1 Problem Statement

The objective of this thesis is the implementation and extension of source localization program on D.Module.C6657 processor. The program localizes a speaker in a room using two microphone arrays, each containing four microphones. The existing DSP program is analyzed and ported from D.Module.C6713 to D.Module.C6657, where the program is tested and extended.

1.2 Thesis Description

The task description of the thesis has been carefully devised to facilitate the analysis and the extension of the existing program. The thesis requires the implementation of source localization program in MATLAB. The program in MATLAB is developed with better features that are adapted to the DSP program on D.M.C6657. The DSP program is analyzed and ported from D.M.C6713 to D.M.C6657, where its features are adapted to have larger FFT buffer and higher position resolution. The source localization program is tested by test sequences created in MATLAB and the position result is displayed on the monitor.

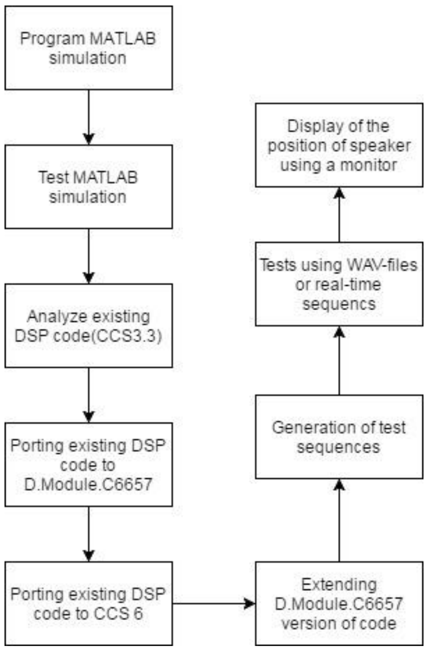

Program MATLAB simulation
Test MATLAB simulation
Analyze existing DSP code(CCS3.3)
Porting existing DSP code to D.Module.C6657
Porting existing DSP code to CCS 6
Extending D.Module.C6657 version of code
Generation of test sequences
Tests using WAV-files or real-time sequencs
Display of the position of speaker using a monitor

Figure 1: Flowchart describes the subtasks of the thesis

2.Background and Literature

2.1 Background

The nature has blessed the human species with a fascinating ability to hear. In addition to sight, the ability to hear allows the humans to respond to situations not visible to the human eye, such as noticing a fast-moving car, locating a ringing mobile phone and answering a call for help. All these tasks accomplished by humans make life easy and are necessary for survival. The one thing that aids to the process of responding to different tasks by hearing is the ability to locate the source of sound. In the field of neuroscience, this ability is called sound localization**.** If the ability to locate source of sound allows humans to accomplish various important tasks, so can this ability allow machines to accomplish similar tasks. It is therefore important to develop a mechanism which could allow machines to locate a source of sound in an area. There have been various proposals for locating sources of sound. In this thesis, a mechanism using microphone arrays for source localization will be investigated.

2.2 Human Auditory System

Source Localization is accomplished by the human auditory system with the brain's ability to interpret the information received by both ears.[1]Sound reaches both ears of a person with different phase, intensity and time. [2]Research is underway for many years to understand the mechanism behind source localization by the human auditory system. In the 1790s it was proposed, by playing flute to a group of audience, that the difference of sound amplitude between the two ears is essential for source localization. Later in 1908, it was proposed that the difference of time of arrival of sound between the two ears is used for source localization. Neurons have been found in the auditory centers of the brain that are specifically tuned to intensity and timing differences between the two ears.[3]This means that the brain is using both cues to localize source of sound. The sound coming from a source closer to the right ear will be louder and arrive faster compared to that of at left ear.

[1]http://knowingneurons.com/2013/03/15/how-does-the-brain-locate-sound-sources/
[2]http://acousticslab.org/psychoacoustics/PMFiles/Module07a.htm
[3]https://courses.washington.edu/psy333/lecture_pdfs/Week9_Day2.pdf

2.3 Source Localization

Source Localization refers to the science of using sound to determine the position of a source[4].In this thesis, two microphone arrays are used to estimate the location of a source. A single microphone array having four microphones is used to estimate the Sample Difference of Arrival (SDOA) between two adjacent microphones in the array. This technique is identical to how humans estimate the direction of sound, using the time difference of sound arrival between the two ears. There are several algorithms to estimate SDOA between two adjacent Microphones, which is essential for Source Localization. The design of the microphone array, the algorithm and the hardware specification greatly influence the performance of SDOA estimation and subsequently the accuracy of source localization.

2.4 Microphone Array

Microphone Array is a device that, instead of having only one microphone to record sound, contains more than one Microphone. Microphone Array comes in various shapes and forms, and can be designed to have as many microphones as required. The simplest microphone array has two microphones, one microphone being on right position and the other on left. These two microphones are used for binaural recording that allow sounds to be recorded from two sides in a room. The dynamic stereo recording mimics surround sound which is distinctly different when heard on a stereo headset.[5]

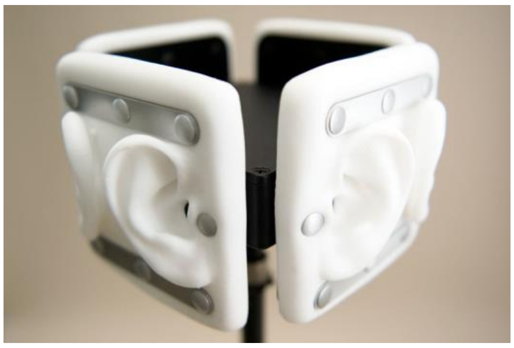


Figure 2:360-degree audio experience Microphone Array with four binaural recordings[6]

The most important characteristic of a microphone array for recording sound is matching of microphones. If the microphones are not all matched then one microphone could have a higher gain than the other. Phase Difference could also be another issue, where two microphone record with a certain delay in time. Microphones in the microphone array must record sound simultaneously. [7]In some cases, a microphone might record sound from all

[4] https://en.wikipedia.org/wiki/Acoustic_location
[5]http://www.learningaboutelectronics.com/Articles/What-is-an-array-microphone
[6]http:elevr.com/audio-for-vr-film
[7]http://www.learningaboutelectronics.com/Articles/What-is-an-array-microphone

directions and a second microphone only from a certain direction. Microphones in an array must all record sound from specific direction(s).These are unwanted problems in audio application and can have horrible consequences for their applications.

Microphone Arrays have become very popular in audio applications. There is no range to how many microphones can be used in an array. Microphone Arrays are used in hearing aids, virtual reality[8] and in military. Microphone Arrays are also popular in binaural and surround recording. The speed and the load carrying capacity of microphone arrays will increase, as more microphone arrays are designed and used in different audio applications.

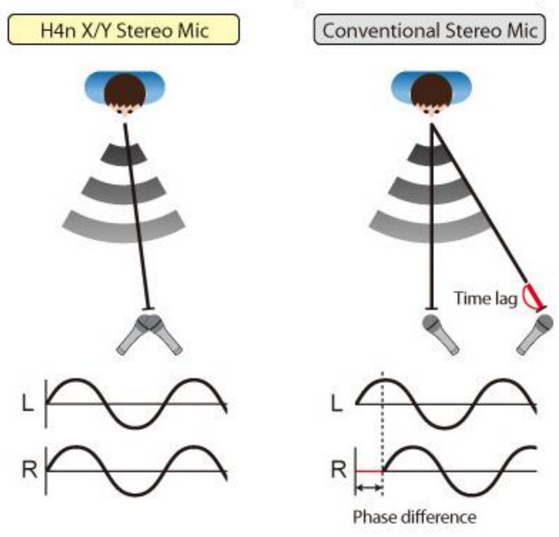


Figure 3: Stereo Recording of source of sound[9]

3. Source Localization Mechanism

3.1 Signals at microphone array

The Angle of Arrival (AOA) also known as Direction of Arrival (DOA) and Time Difference of Arrival (TDOA) are information of delay between signals of two adjacent microphones. These values are necessary for source localization. DOA is essentially the angle that the wave-front of the source makes with the normal of the microphone array. TDOA is the difference in arrival time of signals between two adjacent Microphones in an array. The following formula is used to calculate DOA and TDOA from estimated SDOA value.[10]

$$\tau_{12} = \frac{d\cos(\theta)}{c}, (TDOA)$$

[8] http://elevr.com/audio-for-vr-film/

[9] httpwww.advancedmp3players.co.ukshopimagesproductsH4NPHASE.jpg

[10] Microphone Array Signal Processing Volume 1 of the series Springer Topics chapter 9 pp1-2

$$\tau_{12} = \frac{SDOA}{fs}, (TDOA)$$

$$\varphi = arcsin\left(\frac{\tau_{12} \cdot c}{d}\right), (DOA)$$

$$\theta = arccos\left(\frac{\tau_{12} \cdot c}{d}\right), (DOA)$$

$$\theta = 90° - \varphi$$

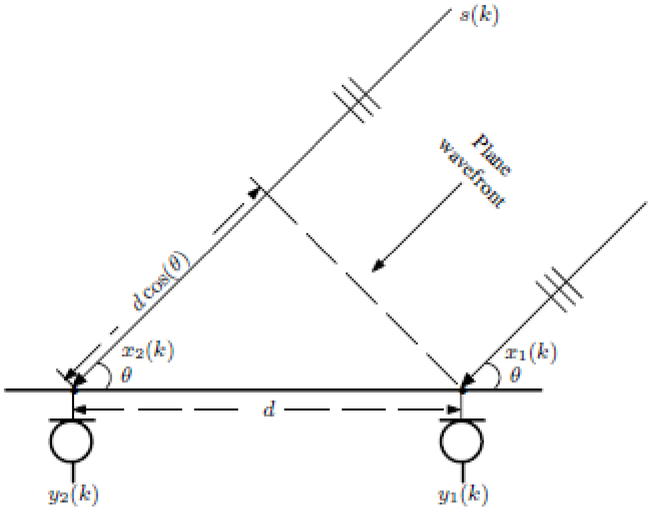


Figure 4: Microphone Array with two microphones[11]

SDOA is a unit less value which gives the difference in term of number of samples between the signal at microphone y_1(k) and at y_2(k). The TDOA represented by $\tau_{12}$is found by dividing SDOA by the sampling frequency. The additional distance travelled by signal to y_2(k) is given by multiplying TDOA with the speed of sound(c). The angle θ of wave front with the array is calculated by dividing the additional distance of the wave front by distance between adjacent microphones and then taking their inverse cosine.

3.2 Single Source Reverberant Model

A source of sound in the far field is localized under single source reverberant model. The signals received by microphones in single source reverberant model can be expressed mathematically. The signal captured by microphone 'y_n' at time 'k' can be expressed as follows:[12]

$$\begin{aligned} y_n(k) &= \alpha_n s(k - t - \tau_{n1}) + v_n(k) \\ &= \alpha_n s[k - t - F_n(\tau)] + v_n(k) \end{aligned}$$

[11] Microphone Array Signal Processing Volume 1 of the series Springer Topics chapter 9 pp 2
[12] Microphone Array Signal Processing Volume 1 of the series Springer Topics chapter 9 pp 6-7

$$= x_n(k) + v_n(k)$$

In the above equation, α_n is the attenuation factor between the range of 0 to 1 and represents the propagation effect on the signal of source *s(k)*, *t* is the propagation time from source to reference microphone, $v_n(k)$ is additive noise signal at the nth sensor and τ_{n1} is the TDOA between sensors 1 and n. The function F_n represents the TDOA between reference and n microphone. For linear and equidistance array the function is

$$F_n(\tau) = (n-1)\cdot\tau$$

and for a linear but non-equispaced array, the function is

$$F_n(\tau) = \frac{\sum_{i=1}^{n-1} d_i}{d_1}\cdot\tau$$

where d_i is the spacing between reference microphone and i_{th} microphone

3.3 Aliasing Effect

The distance between adjacent microphones in a microphone array is a physical factor that affects the performance of SDOA estimation in source localization program. The maximum distance between two microphones in the array depends on the maximum required frequency of the source. If frequencies are present in the source that the physical limitations of microphones do not allow, the program will run into spatial aliasing. Therefore the spacing between two adjacent microphones needs to be less than or equal to half the wavelength of the maximum frequency of the source.

$$\lambda = \frac{c}{f_{max}}$$

$$d \le \frac{\lambda}{2} = \frac{c}{2\cdot f_{max}}$$

3.4 Algorithm for SDOA estimation

Source Localization would not be as challenging of a task if the signal at microphones were merely shifted and attenuated versions of the original signal. However, there are various factors that make the process of SDOA estimation complicated. The sound is always immersed in noise due to the noise being present in the natural environment. In addition, due to reflection of the signals from boundaries and objects, attenuated signals at Microphones could contain multiple signals and delayed replicas of the original signal. These signals at microphones are severely deteriorated due to multipath propagation effect, which causes the signal to have echoes and spectral distortions, termed as reverberation. Furthermore, the source might move from time to time, adding to the complexity of SDOA estimation.

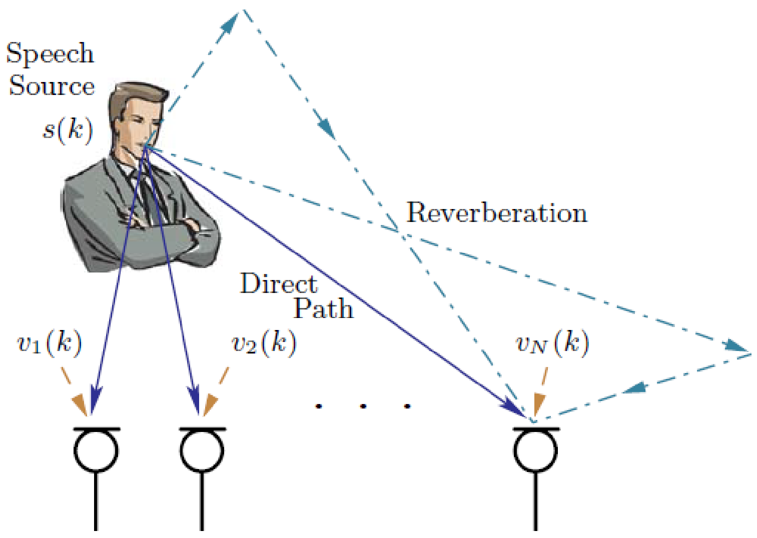


Figure 5: Reverberation Model using a microphone array[13]

In principle, there are four signal models; single-source free field model, multiple source free-field, single source reverberant and multiple source reverberant model.[14] In the free-field model the Microphone receives only the direct path signal, and in reverberant model Microphone may receive many reflected signals in addition to direct path signal. The complexity and method of SDOA estimation varies depending on the environment the source is present in.

In this section, an algorithm is discussed which is robust to noise and reverberation. The algorithm should be able to estimate SDOA in far-field reverberant environment. Cross-Correlation (CC) is the simplest method for SDOA estimation, considering that a single source is in free-field environment. The method makes use of only two microphones to estimate SDOA of the source. However, CC method performs poorly in reverberant environment and faces spatial aliasing issues. Despite these complications multiple microphones could be used to improve SDOA estimation in adverse acoustic environment. The fundamental idea is to take advantage of the redundant information provided by the multiple microphones for SDOA estimation.

Spatial linear prediction algorithm takes advantage of the SDOA redundancy among multiple microphones. The performance of SDOA estimation is expected to increase with the use of multiple microphones in a microphone Array. Spatial correlation matrix is the key element in the use of spatial linear prediction algorithm for SDOA estimation. However, the spatial correlation matrix can more naturally be used for SDOA estimation through Multichannel Cross Correlation Coefficients (MCCC). The MCCC algorithm calculates the correlations of signals at the microphones and creates matrices using correlation coefficients. The SDOA is estimated from the index of the minimum value determinants of matrices.

[13] Microphone Array Signal Processing Volume 1 of the series Springer Topics chapter 9 pp 6
[14] Microphone Array Signal Processing Volume 1 of the series Springer Topics in Signal Processing Chapter 9 pp 1-3

3.5 Multichannel Cross Correlation Coefficient Algorithm

The MCCC algorithm is a method used to estimate SDOA between adjacent microphones in a microphone array. This method takes advantage of the redundant information of multiple microphones to estimate SDOA in adverse acoustic environment. It can be deduced from the name of the method that the algorithm uses cross correlation coefficients to estimate SDOA. Cross-Correlation coefficients are a measure of the strength of the association between two signals. The Cross-Correlation coefficients are used to create spatial correlation matrices for SDOA estimation.

A signal vector is defined by

$$y_a(k,p) = \left[y_1(k)\ y_2[k+F_2(p)] \cdots y_N[k+F_N(p)]\right]^T$$

where 'k' is the time, and 'p' is the value of SDOA. The value of 'p' moves through the range of estimable SDOA. At each value of the range of SDOA, calculations of MCCC are performed.The function $F_N(p)$ represents SDOA between first and Nth Microphone.[15] The following formula gives the SDOA between reference and nth microphone, with 'p' being the SDOA between adjacent microphones.

$$F_n(p) = (n-1)\cdot p$$

The corresponding spatial correlation matrix can be written as:

$$R_a(p) = E[y_a(k,p)\ \ y_a^T(k,p)]$$

$$R_a(p) = \begin{bmatrix} \sigma_{y1}^2 & r_{a,y1y2} & \cdots & r_{a,y1yN}(p) \\ r_{a,y1yN}(p) & \sigma_{y2}^2 & \cdots & r_{a,y2yN}(p) \\ \vdots & \vdots & \ddots & \vdots \\ r_{a,yNy1}(p) & r_{a,yNy2}(p) & \cdots & \sigma_{yN}^2 \end{bmatrix}$$

The diagonal of spatial correlation matrix is filled by variance σ_{yN}^2, which are defined as the mean of the squared values of signal.

$$\sigma_{yn}^2 = E[y_n^2(k)]$$

$$\sigma_{yn}^2 = \frac{\sum_{i=0}^{N-1} y_{n,i}^2}{N-1}$$

The rest of the coordinates are filled by cross-correlation coefficients:

$$r_{a,y_iy_j} = E\{y_i[k+F_i(p)]y_j[k+F_j(p)]\}$$

[15] Microphone Array Signal Processing Volume 1 of the series Springer Topics chapter 9 pp 12-13

For spatial correlation matrices greater than 2 x 2, the value of index ‘p’ in cross-correlation coefficients depends on the maximum SDOA between signals at ith and jth microphones. For example, for matrix Ra(p) value of index of cross correlation at $r_{a,y1y2}$ will be p, but value of index will be 2p for $r_{a,y1y3}$ and 3p for $r_{a,y1y4}$ and so on. The indices of coefficients change because as the distance between ith and jth microphones increases, so does the maximum SDOA value and the index ‘p’ that represents the SDOA value in the cross correlation.

The spatial correlation matrix $R_a(p)$ can be factorized by taking out the variance present in the diagonal of matrices. The factorization is done as follow:

$$R_a(p) = \sum \overline{\mathrm{R_a}}(\mathrm{p}) \sum,$$

where

$$\sum = \begin{bmatrix} \sigma_{y1}^2 & 0 & \cdots & 0 \\ 0 & \sigma_{y2}^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_{yN}^2 \end{bmatrix}$$

is a diagonal matrix and

$$\overline{\mathrm{R_a}}(\mathrm{p}) = \begin{bmatrix} 1 & \rho_{a,y1y2} & \cdots & \rho_{a,y1yN}(p) \\ \rho_{a,y2y1}(p) & 1 & \cdots & \rho_{a,y2yN}(p) \\ \vdots & \vdots & \ddots & \vdots \\ \rho_{a,yNy1}(p) & \rho_{a,yNy2}(p) & \cdots & 1 \end{bmatrix}$$

is a symmetric matrix and

$$\rho_{a,y_iy_j}(p) = \frac{r_{a,y_iy_j}(p)}{\sigma_{y_i}\sigma_{y_j}}\ , i,j = 1,2,\dots,N$$

is the correlation coefficient between the ith and jth aligned microphone signal. The numerator is standard deviation of the signal y_i and y_j signal respectively. The standard deviation is calculated as follow:

$$\sigma_{yn} = \sqrt{\frac{\sum_{i=0}^{N-1} y_{n,i}^2}{N-1}}$$

Since the matrix $\overline{\mathrm{R_a}}(\mathrm{p})$ is the symmetric and positive semi-definite, and its diagonal elements are all equal to one, it can be shown that

$$0 \leq det\left[\hat{R}_a(p)\right] \leq 1,$$

where det(.) stands for determinant.

The MCCC has the following properties

1. $0 \leq \rho^2{}_{a,y1:yN}(p) \leq 1,$
2. If two or more signals are perfectly correlated then $\rho^2{}_{a,y1:yN}(p) = 1$
3. If all the signals are completely uncorrelated with each other, then $\rho^2{}_{a,y1:yN}(p) = 0$
4. If one of the signals is completely uncorrelated with the N-1 other signals, then the MCC will measure the correlation among those N-1 remaining signals

The determinant values of matrices are expressed logarithmically for better visualization of values in the graph by the following equation.

$$10\log(\det[R_a(p)])(dB)$$

The SDOA between two adjacent microphones is determined by the following step:

$$\hat{\tau}^{MCCC} = argmindet\left[\hat{R}_a(p)\right]$$

$$= argmindet[R_a(p)]$$

4. Hardware

The Source Localization program is implemented on DSP hardware. The hardware consists of DSP Board D. Module.C6657 from D.SignT with I/O card D.Module.PCM3003 peripheral module.

4.1. D. Module.C6657Board

D.Module.C6657 is a Texas Instruments dual-core TMS320C6657 Keystone processor. This dual-core processor is characterized by huge computational power and fast serial interfaces. It is a high-performance, stand-alone, floating-point, (DSP) computer board. The D. Module2 series represents the next generation of high-performance, stand-alone DSP boards.[16] These boards are optimized for highest I/O bandwidth to satisfy even demanding applications. The DSP can be programmed in both fixed-point and in floating-point arithmetic. The DSP has dual core architecture with a clock frequency of 1.25 GHz, the DSP 20 GFLOPS (Giga-floating-point operations per Second). The board features variety of interfaces and communication peripheries such as:

[16]http://www.gbm.de/fileadmin/produkte/Datenblatt/DM2-C6657_DS.pdf

- Large internal memories, external 512 Mbytes DDR3 memory
- Gigabit Ethernet
- UART
- SPI
- I²C
- two McBSP Synchronous Serial Ports
- Real-Time Clock

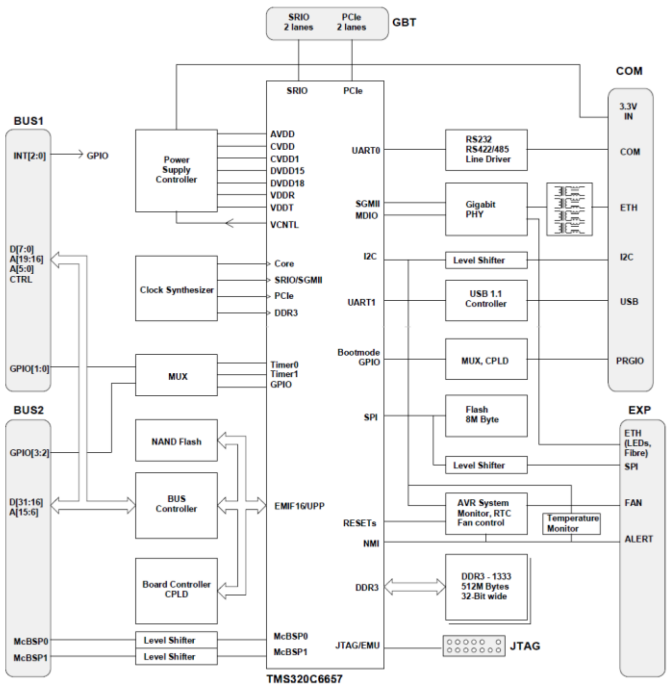


Figure 6: D. Module2.C6657 Block Diagram(from reference [9])

Applications

• Ultrasonic and eddy-current material inspection
• Machine vision
• Software defined radio (SDR)
• Radar and sonar
• Surveillance and security

4.2. D.Module.PCM3003

The D. Module.PCM3003 is a peripheral module that contains two or four PCM3003 codecs, which are connected to DSP via serial ports (McBSP, SPORT). The peripheral module interfaces to the D. Module.C6657 via one or two serial ports, The D. Module.PCM3003 is available in four channels and eight channel configurations. The version with 4-channels uses Serial Port 0 only; the 8-channel version uses both Serial Port 0 and 1. The DSP board has two McBSP, for data exchange between the PCM3003 codec. The data exchange between the McBSP and codec takes place over two data lines, one for the reception, and the other for sending. The channels from 0 to 3 are read and written via Serial Port 0 (McBSP0, SPORT0), channels from 4 to 7 are interfaced through Serial Port 1 (McBSP1, SPORT1).The hardware module is equipped with eight 16-bit resolutions AD and DA converters. The synchronizing and clock information is provided separately for receiving and sending data lines.[17]The range of sampling frequency internally generated by PCM3003 is from 8 to 48kHz.In this work, the sampling rate is set to 48 kHz.

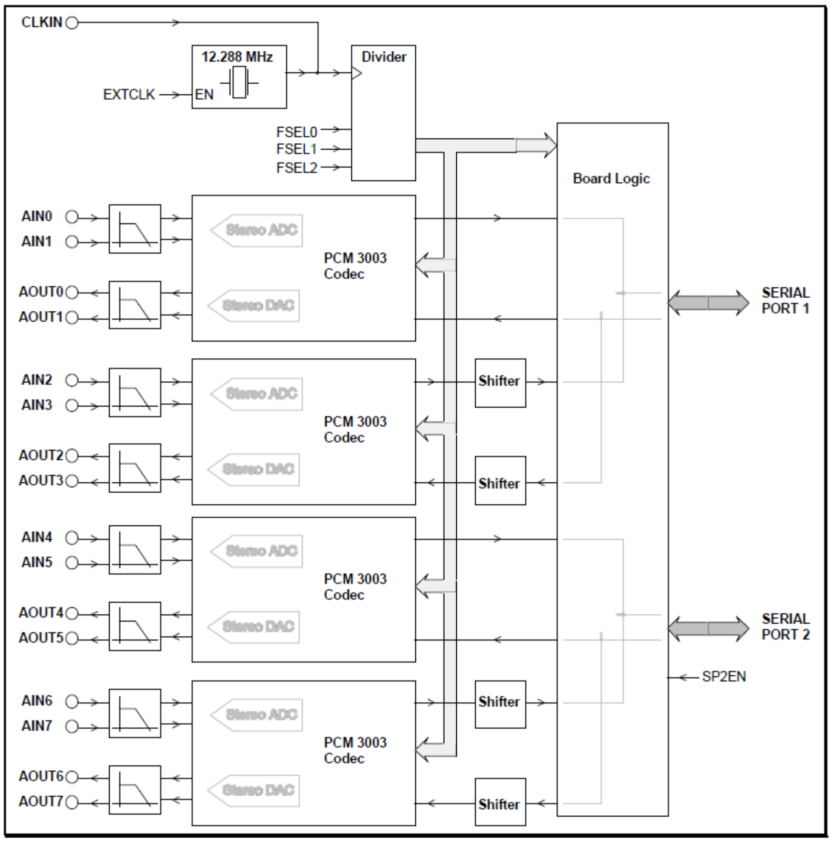


Figure 7: D. Module2.PCM3003 Block Diagram (from reference [18])

4.3. Code Composer Studio

Code Composer Studio is an integrated development environment (IDE) that supports the development of applications for Texas Instruments (TI) embedded processors.CCS offers environment for real-time DSP applications based on the C programming language. Code Composer Studio is

[17]http://www.ti.com/lit/ds/symlink/pcm3002.pdf

primarily designed as for embedded project design and low-level JTAG based debugging. [18]CCS includes an optimizing C/C++ compiler, an assembler, source code editor, project build environment, debugger, profiler, and many other features.[19]The CCS program has a better view of variables and data Graphical representations. Code Composer Studio combines the advantages of the Eclipse software framework with advanced embedded debug capabilities from TI resulting in a compelling feature-rich development environment for embedded developers.[20] Code Composer Studio version 5 or newer version is required for TMS320C6657.

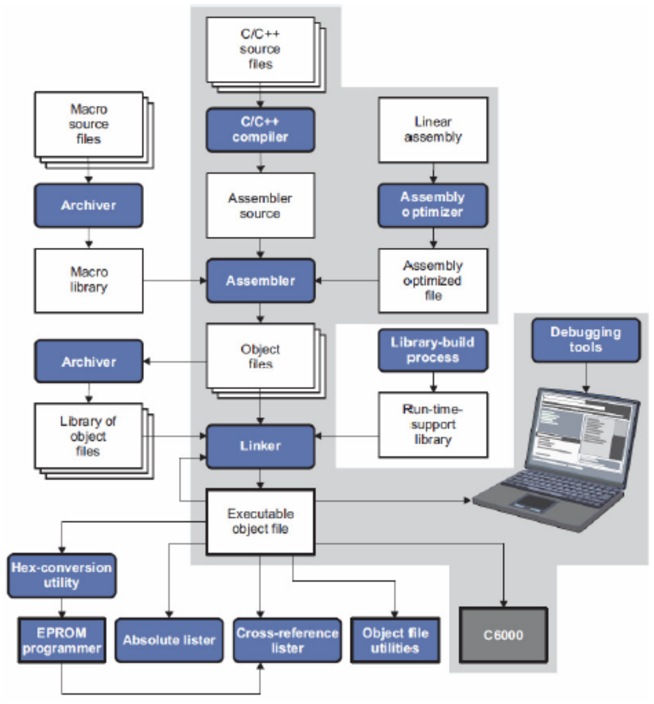


Figure 8: Code Composer Studio flow diagram (from reference [16]))

4.4. D. Module.C6657 project on CCS

Support software for D. Module2-C6657 needs to be installed to run the C6657 program on CCS. The support software contains libraries of D. Module2.C6657 program, without which the program would not run on CCS. The Support Software contains the following directories and files:

Folder D.SignT\Boards\DM2C6657\BoardSupport:

- dm2c6657.cmd - default linker command file
- dm2c6657.h - D.Module2.BIOS header file
- dm2c6657.cfg - default D. Module2 module configuration file

[18]http://softwaresofelectronics.blogspot.de/2015/07/code-composer-studio.html

[19]http://www.ti.com/tool/ccstudio

[20]http://processors.wiki.ti.com/images/tmp/f1269801485-529964023.html

- dm2c6657.gel - Code Composer Studio Emulator GEL file
- hex6x.cmd - Hex6x.exe command file to generate Intel-Hex program files to store in the board's Flash Memory(from reference [15])

Folder D.SignT\Boards\DM2C6657\Documentation

- User's Guides and BIOS Reference as .pdf files, BIOS Reference also as .chm Help File

Folder D.SignT\TI\C6657

- Chip Support Library (CSL) files

Folder D.SignT\Utilities

- currently empty, no board specific utility programs required

Folder D.SignT\USB\USBDriver\DM2C6657

- FTDI USB Virtual COM Port Driver for Windows

Folder D.SignT\Boards\DM2C6657\Projects

- A simple echo program is contained to demonstrate the usage of the D. Module2.BIOS functions and access to the on-board peripherals. Use this example as the basis for your own projects. For your program to access BoardSupport files, your project should be in this folder, two folders down from BoardSupport folder. In case of the program being outside Projects folder, the program will be unable to open files needed for DSP.

4.4.1. Building your own projects

Please refer to the instructions in the D. Module2.C6657 User's Guide. The easiest way to ensure correct project settings and configuration for your own project is to copy a suitable project from the support software:

- create your own project folder, e.g. myproject in D.SignT\Boards\DM2C6657\Projects
- copy all files from an example project to your own project folder, the subfolders must not be copied
- open the .project and .cproject files with a text editor and replace all occurrences of the original project name with your new project name (.cproject: <project id="PROJNAME.com.ti.ccstudio.... at line 81,
  .project: <name>PROJNAME</name> at line 3)
- import your project in CCS using “Project - import existing CCS eclipse project” (from reference [15])

4.4.2. CCS.V5 emulator installation

CCSV5 already installs the required emulation drivers. To create a target connection to the DM2.C6657 board:

- change to the Debug View and open the Target Configurations (if not already shown use “view – target configurations”
- right click on “User Defined” and select “New Target Configuration”
- assign a name, e.g. DM2C6657_XDS200.ccxml, a new tab will open
- in Connection select your JTAG emulator device, e.g. Texas Instruments XDS2xx USB Emulator
- In Board or Device type 6657. The following selection box will show the TMS320C6657. Activate the checkbox.
- Change to the “Advanced” tab and select the C66xx_0 core. Select the initialization script dm2_c6657.gel from the support software D.SignT\Boards\DM2C6657\BoardSupport\, click the “Save” button
- Connect the emulator to the D.Module2.C6657 and power-up the board. Click “Test Connection”(from reference [15])

5. Estimating position of source

The value of maximum SDOA estimable by the microphone arrays is significant for performance of source localization in the far-field. The value of maximum SDOA affects the resolution of angle and position of the source. The accuracy of source localization increases with the increase in resolution of angle estimation.

5.1. Determining maximum estimable SDOA

The greater the value of maximum estimation SDOA, the more accurately the source can be localized. The maximum SDOA depends on the spacing between adjacent microphones in the array and the maximum allowable frequency of the source. Assuming the spacing between adjacent microphones is 0.05(m) and the sampling frequency is 48000Hz, the maximum allowable frequency will be 3400Hz. In this case range of SDOA will be between -7 and 7. However if the spacing is 0.17(m) and sampling frequency is 48000Hz, the maximum allowable frequency will be 1000Hz. The range of SDOA will increase to -24 and 24, but maximum allowable frequency will decrease drastically.

Example of determining maximum allowable frequency:

- $Speed\ of\ sound\ , c = 340\frac{m}{s}$
- $Sampling\ Frequency, fs = 48kHz$
- $Microphone\ Spacing, d = 0.05m$

$$d = \frac{\lambda}{2}$$

$$\lambda = \frac{c}{f_{max}}$$

$$f_{max} = \frac{c}{2 \cdot \mathrm{d}}$$

$$f_{max} = \frac{340m/s}{0.1\mathrm{m}}$$

$$f_{max} = 3400Hz$$

Example of determining maximum estimable SDOA:

- $Speed\ of\ sound\ , c = 340\frac{m}{s}$
- $Sampling\ Frequency, fs = 48kHz$
- $Maximum\ allowable\ frequency, f_{max} = 3.4kHz$

$$\lambda = \frac{c}{f_{max}}$$

$$\lambda = \frac{340m/s}{3.4kHz}$$

$$\lambda = 0.1\mathrm{m}$$

$$d = \frac{\lambda}{2}$$

$$d = \frac{0.1m}{2}$$

$$d = 0.05m$$

$$Maximum\ SDOA\ = fs \times \frac{d}{c}$$

$$Maximum\ SDOA\ = 48kHz \times \frac{0.05m}{340m/s}$$

$$Maximum\ SDOA\ = 7$$

5.2. Calculation of DOA

The source is located at the intersection of the DOAs measured by the two microphone arrays. Therefore the angle of source with the microphone arrays needs to be determined to localize the source. The value of SDOA is used to determine TDOA and DOA values. The angle of source with microphone array is determined in the following example:

- SDOA = 1
- Sampling Frequency(fs) = 48000Hz
- Speed of Sound© = 340m/s
- Microphone spacing(d) = 0.05m
- D=Difference of distance travelled by signals to adjacent microphones

$$D(m) = \tau(s) \times c(m/s)$$

$$TDOA(\tau) = \frac{SDOA}{fs}$$

$$D = \frac{1}{48000Hz} \times 340m/s$$

$$D = 0.007083m$$

$$\propto = \arcsin\left(\frac{D}{\mathrm{d}}\right)$$

$$\propto = \arcsin\left(\frac{0.007083m}{0.05\mathrm{m}}\right)$$

$$\propto = 8.14°$$

$$\theta_1 = \arccos\left(\frac{0.007083m}{0.05\mathrm{m}}\right)$$

$$\theta_1 = 81.86°$$

The angle resolution is changes by changing the distance between microphones. The table below gives the resolution for two different spacing between adjacent microphones:

Sampling frequency(Hz)	Microphones Spacing(d)	Angle resolution($\Delta\theta$)
48kHz	0.05m	8.14°
48kHz	0.17m	2.38°

Table 1: Representing the resolution of angle detection depending on the spacing between microphones

5.3. Determining position of speaker

The position of the speaker is estimated after SDOA and DOA of the source have been determined. A value of DOA exists for the corresponding SDOA value, depending on the resolution of angles. The position of the source can be determined by using the DOA value at the microphone arrays and the spacing between Microphone Arrays.

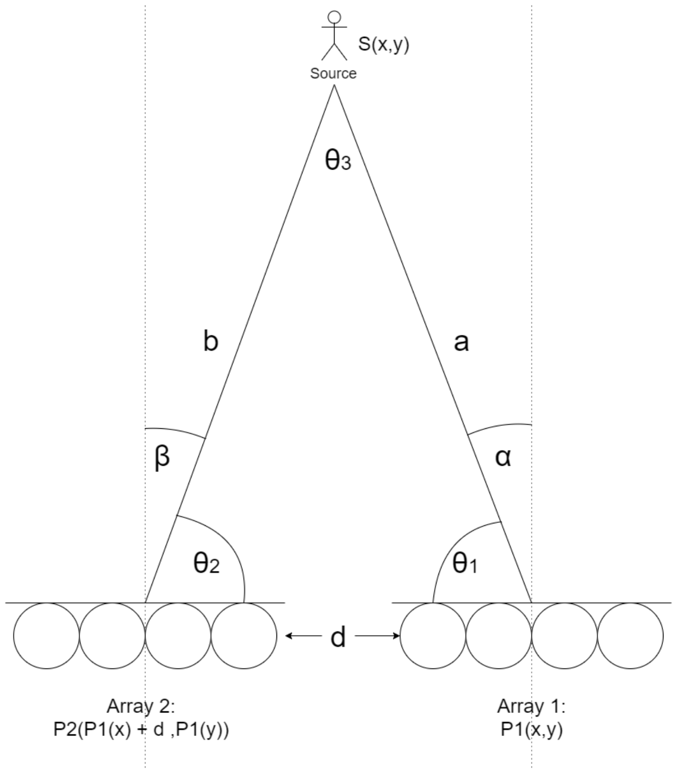


Figure 9: Source localization example with the help of two microphone arrays

$$\theta_1 = 90 + \alpha$$

$$\theta_2 = 90 - \beta$$

$$\theta_3 = 180 - \theta_1 - \theta_2$$

$$\frac{sin(\theta_2)}{a} = \frac{sin(\theta_3)}{d}$$

$$a = d\frac{sin(\theta_2)}{sin(\theta_3)}$$

$$S_x = P_{1x} + a \cdot cos(\theta_1)$$

$$S_y = P_{1y} + a \cdot sin(\theta_1)$$

The SDOA at the microphone arrays are different in the sign, since the source is on the left of array 1 but on right of array 2. The angle determined for negative SDOA will be a negative angle and positive angle for a positive SDOA. In the example above angle α is negative and angle β is positive.Therefore, adding angle α to 90° and subtracting β from 90° gives the correct angles for θ_1 and θ_2.The other angle $\theta_3$is the angle between direction of two wave fronts. Sine rule is used to determine the distance from Microphones to Source and the location of source.

5.4. Resolution of DOA at Microphone Array

The maximum estimable SDOA using Microphone Array is proportional to the sampling frequency of the hardware. Increasing the sampling frequency of Hardware increases the SDOA resolution. This in return enables the program to detect larger number of positions the source could be present at. Another possibility of increasing resolution is increasing the spacing between adjacent Microphones. However, there is a drawback to increasing the distance between two Microphones, that the maximum allowable frequency of the source will reduce and the program could run into spatial aliasing.

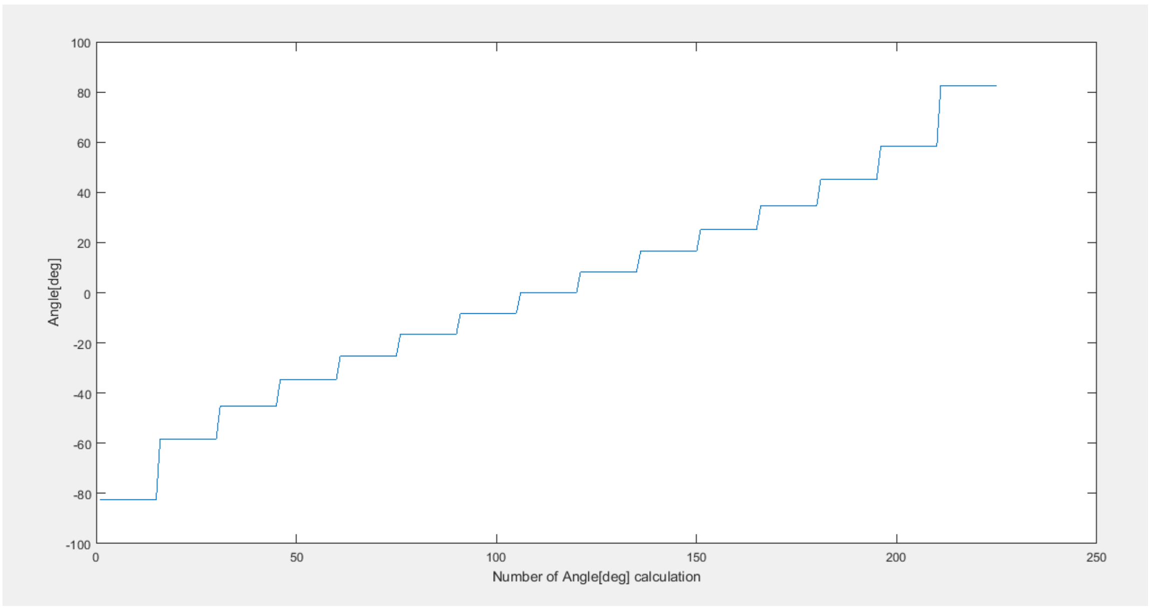


Figure 10: Graph showing the resolution of angle for SDOA range -7 to 7 with 15 steps

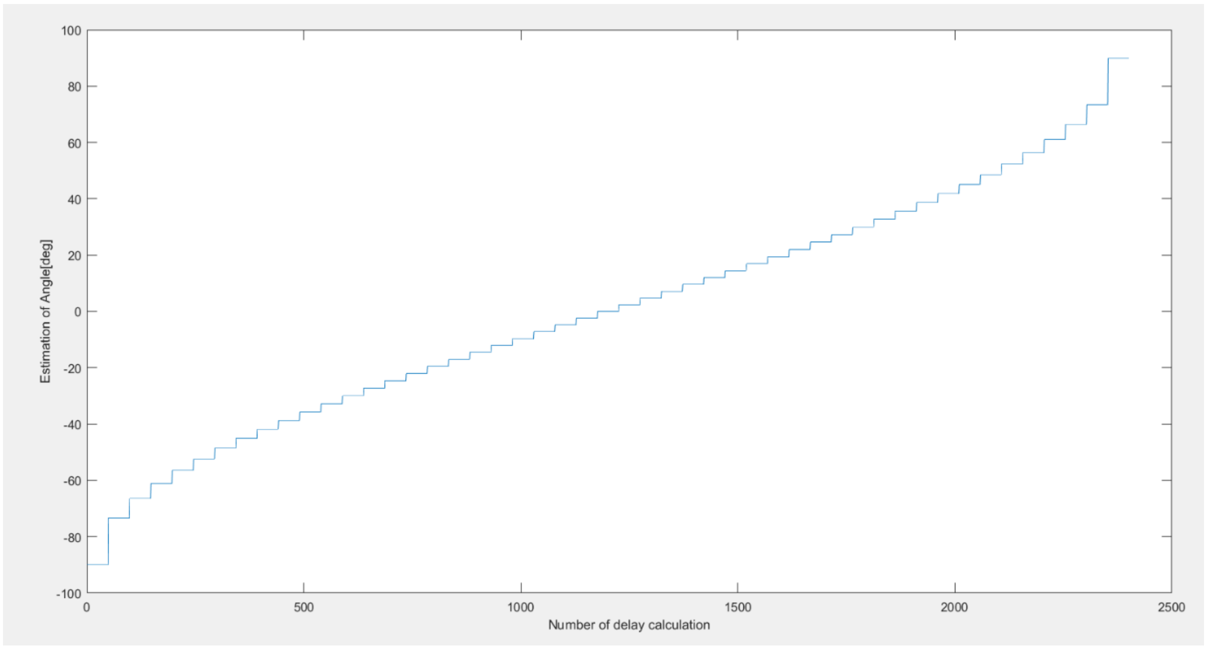


Figure 11: Graph showing the resolution of angle for SDOA range -24 to 24 with 49 steps

The graph in figure 10, illustrates the DOA resolution for SDOA range -7 to 7, whereas graph in figure 11 illustrates DOA resolution for SDOA range of -24 to 24. The resolution of DOA values in figure 11 is greater compared to figure 10. For SDOA range of -24 to 24, the location of source can be more accurately determined.

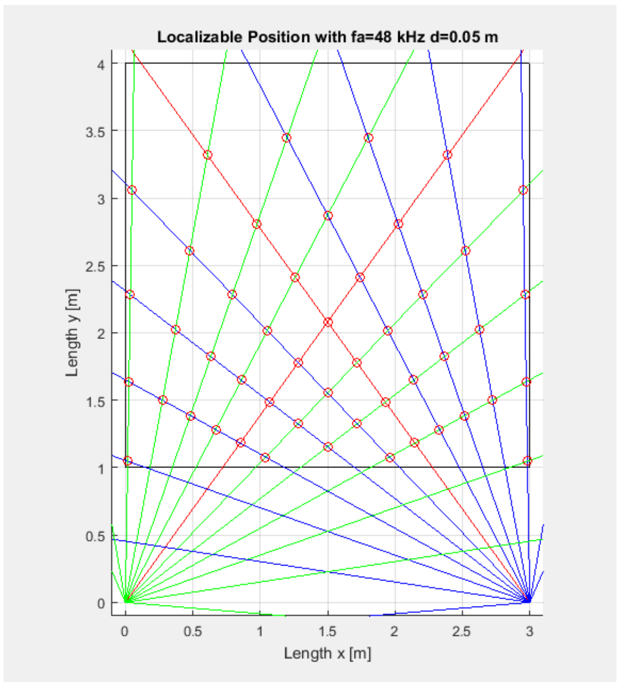


Figure 12: Detectable position of source depending on the microphone spacing of 0.05m (from reference [16])

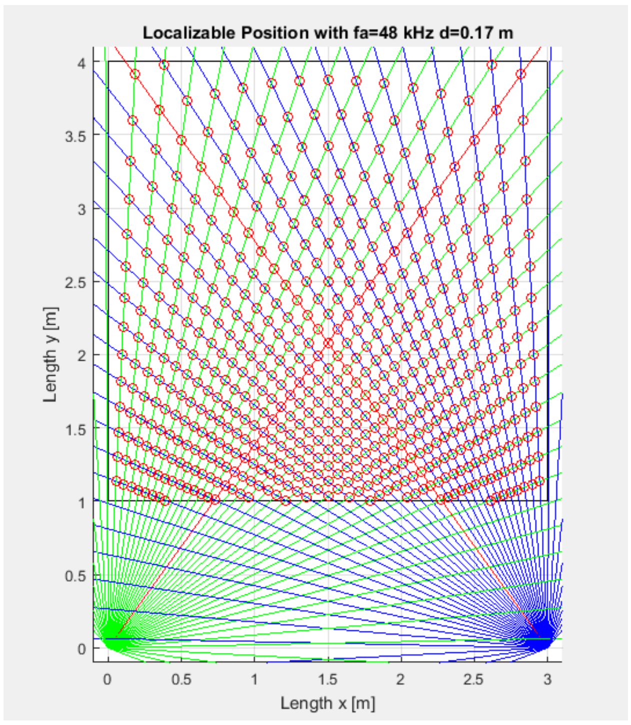


Figure 13: Detectable position of source depending on the microphone spacing of 0.17m (from reference [16])

The two diagrams in figure 12 and figure 13 complement graphs in figure 10 and 11. These two diagrams illustrate how the accuracy of source localization increases with the increase in distance between adjacent microphones in the array. The increase in distance between adjacent microphones increases the maximum estimable SDOA. This causes the resolution of DOA to increase, since there is a value of angle for each SDOA value. The increased DOA resolution increases the possible locations the source could be present at. The distance between adjacent microphones is greater in figure 13. Therefore the accuracy of position estimation is greater in figure 13 compared to accuracy of position in figure 12.

6.MATLAB Program

Source Localization program is implemented in MATLAB using MCCC algorithm. In addition to SDOA estimation simulation, a program for generating test sequences is integrated to the MATLAB program. Furthermore, a program to create a look up table of the position of source for different SDOA values of the microphone arrays is also included in MATLAB program.

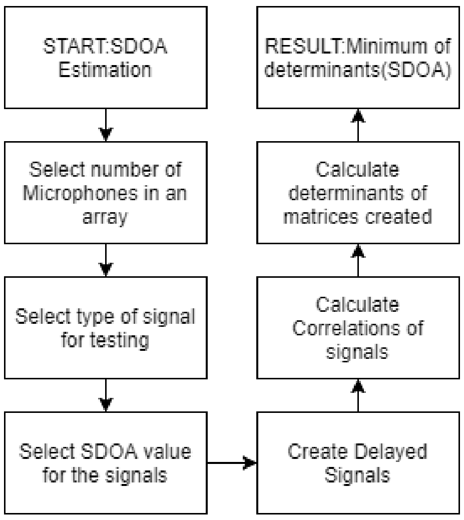

START:SDOA Estimation
Select number of Microphones in an array
Select type of signal for testing
Select SDOA value for the signals
Create Delayed Signals
Calculate Correlations of signals
Calculate determinants of matrices created
RESULT:Minimum of determinants(SDOA)

Figure 14: Diagram representing the flow of SDOA estimation program

6.1. SDOA estimation using two Microphones

Steps to develop SDOA estimation program using 2 microphones

1. An array of signal of buffer length 1024 is received at 2 microphones of the array. The signal reaches 2nd microphone with a certain delay compared to the 1st microphone or vice versa depending on the position of source.
2. The two signals are cross-correlated. The length of the cross-correlation is twice the length of buffer.
3. In the next step, 2x2 matrices are created for a range of cross-correlation coefficients
4. Determinant is calculated of each matrix
5. Finally the minimum determinant is found to be the SDOA between microphones

How are the steps implemented in MATLAB?

1. Signals for microphones are created using sinusoidal function or read from wav file
   - **a.** A signal 'x' of 1024 samples representing reference microphone is created
   - **b.** A delay to the signal at 2nd microphone is introduced by Introduction_Of_Delay function

```
win = 1024;
[x,fs] = audioFile('music.mp3');
x = x(1,1:win);
y = Introduction_Of_Delay(delay,x);
```

Code 1: The example code creates signals for SDOA estimation using 2 microphones

Example: x is original signal and y is delayed by 3 samples							
x	-0.1691	-0.1903	-0.2228	-0.2632	-0.2999	-0.3222	-0.3298
y	-0.2999	-0.3222	-0.3298	-0.1691	-0.1903	-0.2228	-0.2632

Table 2: Result part of original and delayed signal

2. Cross Correlate the signals with each other
   a. Create a cross correlation function that takes two signals and cross correlates them
   b. For N microphones there must be sum(N-1) Cross correlations
   c. Length of cross correlation must be twice the buffer length
   d. Values of cross correlation must be scaled by block length of cross-correlation

The following section includes three different sample codes that can be implemented in MATLAB for cross-correlation calculation. The function mentioned in Code 2 is an inbuilt MATLAB function that needs to be scaled to get the correct result of cross-correlation. The function in Code 3 includes self written function using Cross-Correlation formula mentioned earlier. The function in Code 4 is from Marcus Müller´s MATLAB program that uses the property of FFT multiplication to calculate cross-correlation. The three functions give the same result of cross-correlation; however the FFT method for calculating correlation is much faster.

```
Correlations = xcorr(signal_1,signal_2)/( 2*length(signal_1)-1);
```

Code 2: MATLAB function for Cross-Correlation calculation

```
function[result]= Cross_Correlation(x,y)
        win = length(x);
        K = 2*length(x)-1;
        result = zeros(1,2*win-1);
       for p=-win+1:win-1
            z = 0;
            for i=1:K-p-1
                if and(i+p<=length(x),i+p>0)
                    if i<=length(x)
                        z = z + x(i+p)*y(i);

                    end
                end
            end
                result(p+win) = z./K;
       end
end
```

Code 3: Function created for Cross-Correlation using its formula

```
function[result]=Cross_Correlation(Signal_1,Signal_2)

    %cross-correlation block size
    K = 2*length(Signal_1)-1;
    %number of cross-correlation values
    NFFT = 2*length(Signal_1)-1;
    %Fast Fourier Transform of signals
    Y_Signal_1 = fft(Signal_1,NFFT);
    Y_Signal_2 = fft(Signal_2,NFFT);
    %Element wise multiplication of fft of signals
    Y_Mul = Y_Signal_1 .* conj(Y_Signal_2);
    result = real(fftshift(ifft(Y_Mul)))./K;


end
```

Code 4: Cross-Correlation of two signals using the FFT multiplication property

Example: Result of self written cross-correlation function							
1.0e-03 *							
0.0061	0.0152	0.0276	0.0442	0.0668	0.0945	0.1260	0.1617

Table 3: Result of cross-correlation using self written function

Example: Result of inbuilt cross-correlation function(xcorr) divided by K=2*N-1							
1.0e-03 *							
0.0061	0.0061	0.0061	0.0061	0.0061	0.0061	0.0061	0.0061

Table 4: Result of cross-correlation using inbuilt function

The following graphs show the result of cross-correlation of different signal types. Cross-Correlation coefficients are used in MCC algorithm. Therefore it is important to get correct result of cross-correlation. The delay between the two correlated signals is visible in the cross-correlation which is the index of maximum value of the graph. It must be noted that depending on the type of signals that are cross-correlated, the cross-correlation graph can be very different. The type of cross-correlation graph that is created affects the performance of MCC algorithm. If the index of peak that represent the SDOA in the graph is not clearly distinguishable from other coefficients, then the algorithm can face difficulty in estimating SDOA. Therefore it is observed that the SDOA estimation program performs best when the signals are random, since the peak of the cross-correlation of random signals is clearly distinguishable.

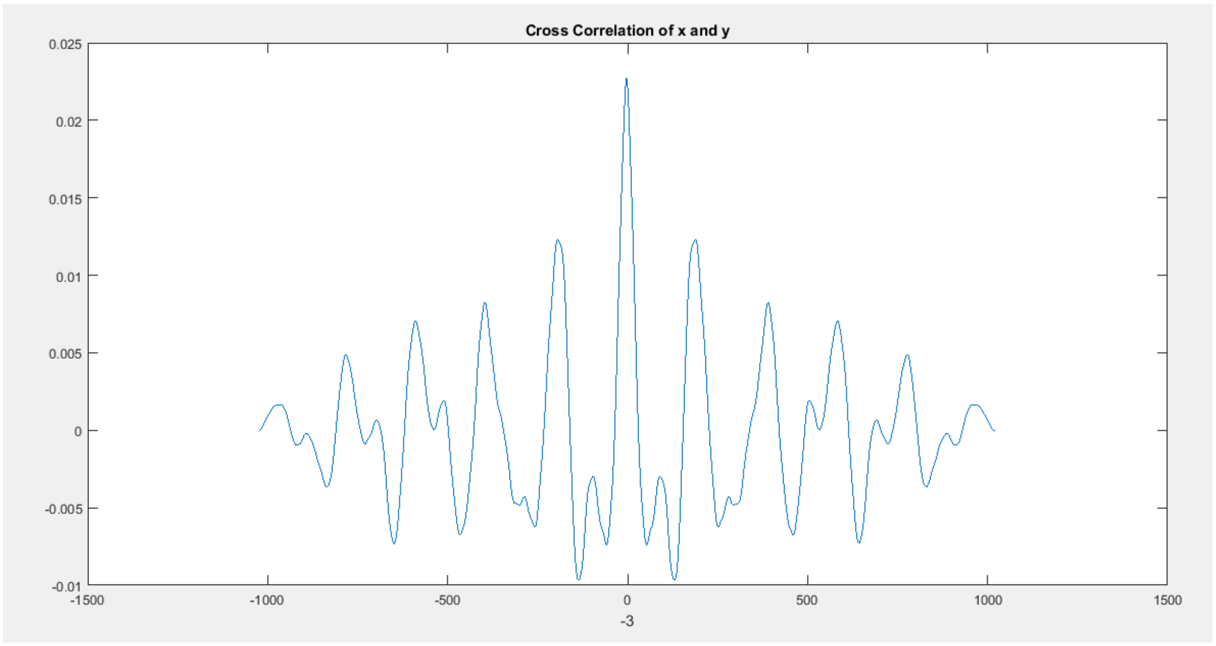


Fig15: The graph shows Cross Correlation of two speech/music signals with delay of -3 samples

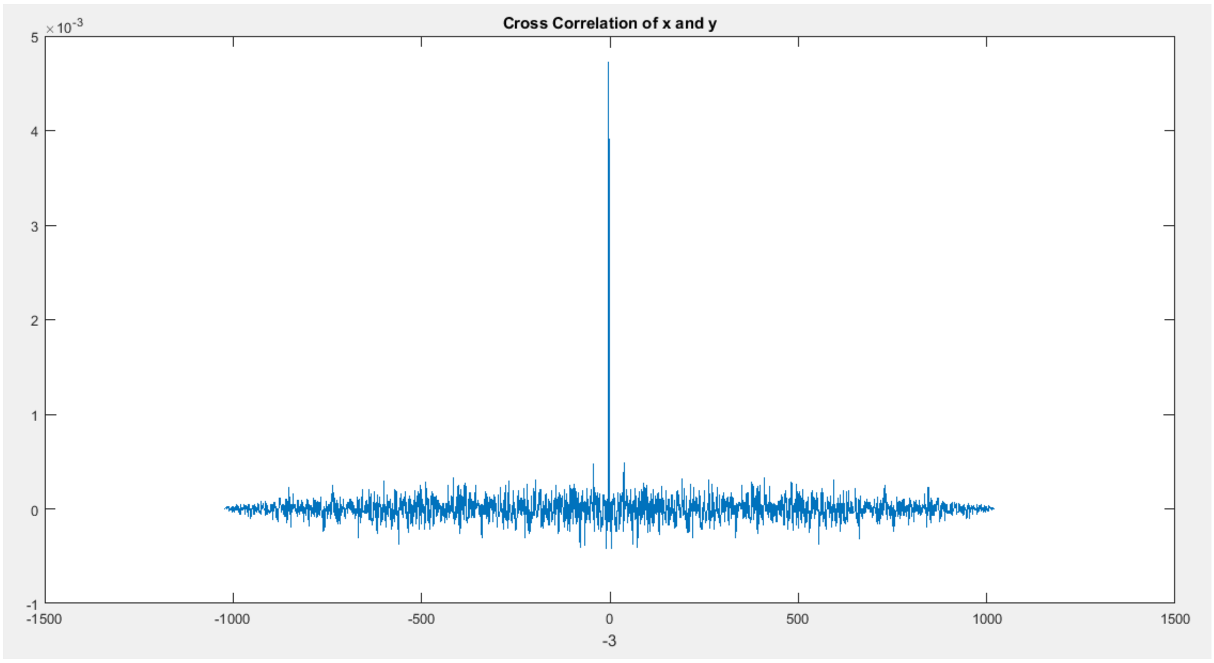


Figure 16: The graph shows Cross Correlation of two random signals with delay of -3 samples

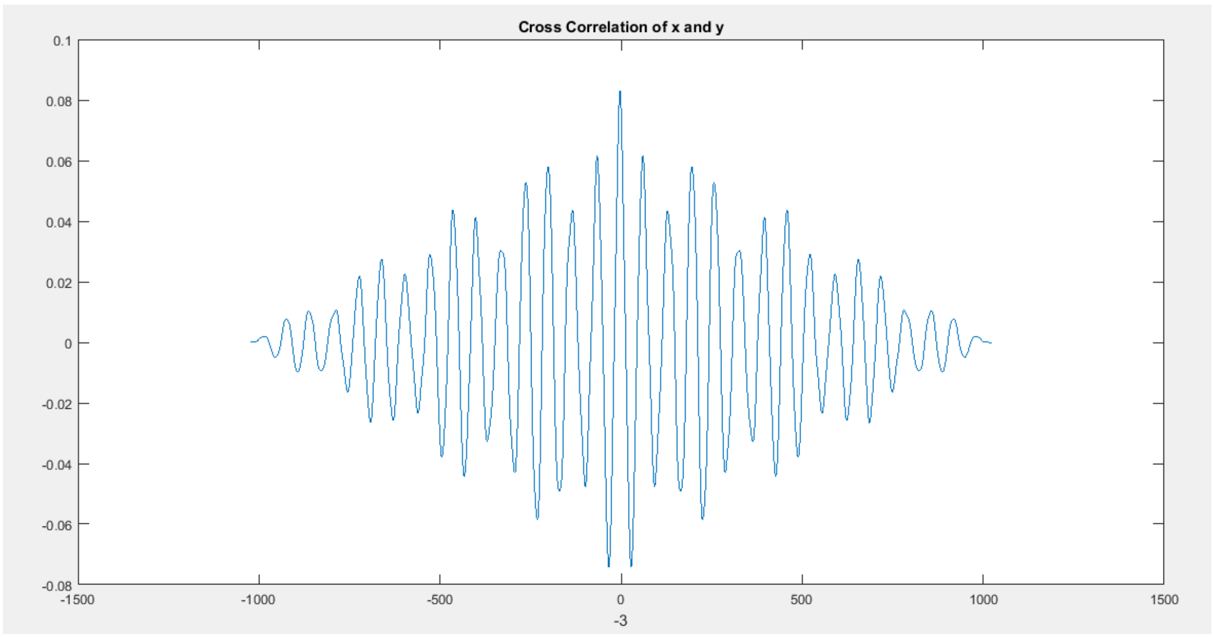


Figure 17: The graph shows Cross Correlation of multi frequency signals with delay of -3 samples

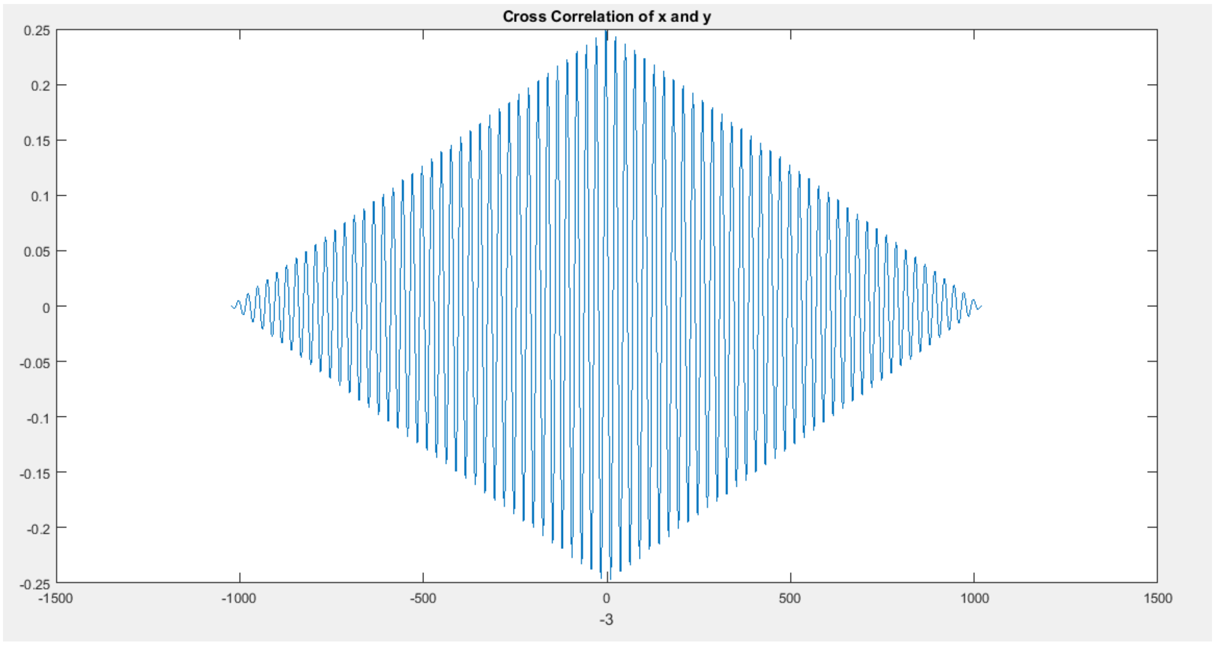


Fig 18: The graph shows Cross Correlation of single frequency signals with delay of -3 samples

3. Create 2x(maximum SDOA)+1 number of 2x2 matrices using correlation coefficients
    a. Create symmetric matrices using correlation coefficients and variance of each signal at microphone
    b. Index of P of the cross correlation is used to create matrices for SDOA estimation
    c. If resolution of SDOA is max at 7 then 2x7+1 numbers of matrices are created. Each matrix represents the SDOA value between two signals. The matrices cover coefficients maximum SDOA value left and right to the center of cross-correlation. This covers the coefficients around peak which is enough to estimate SDOA between signals

```
R = zeros(2,2);
R(1,1) = variance(x);
R(2,2) = variance(y);
R(1,2) = correlations (P);
R(2,1) = correlations(P);

OR

R(1,1) = 1;
R(2,2) = 1;
R(1,2) = correlations(P)./(standardDeviation(x)* standardDeviation(y));
R(2,1) = correlations(P)./(standardDeviation(x)* standardDeviation(y));
```

Code 5: Creation of Matrix for SDOA estimation using 2 Microphones

4. Determinant is calculated of each matrix
    a. Determinant is calculated for all of the matrices created for SDOA estimation
    b. Inbuilt function det(.) can be used for calculating determinant of the matrix
    c. Determinant of a symmetric matrix is between 0 and 1

5. Find minimum of all determinants to estimate SDOA
    a. Find the minimum of the determinants
    b. The index of the minimum determinant in the array of determinants determines the SDOA

```
function[delay]=findDelay(R)
    min = 0;
    delay = 0;
    for i=1:floor(length(R)/2)
        if R(i)<min
            min = R(i);
            delay = i;

        end
    end
    delay = ceil(length(R)/2) -delay;

end
```

Code 6: Find the index of SDOA in the graph of determinants

Testing the SDOA estimation program

The graphs of determinants for SDOA estimation for different type of signals are shown below. The delay in the signals for all test performed below is of 6 samples. The graph of determinants is different for different type of signals used to test SDOA estimation. The program is able to estimate the range of delays in the signals accurately.

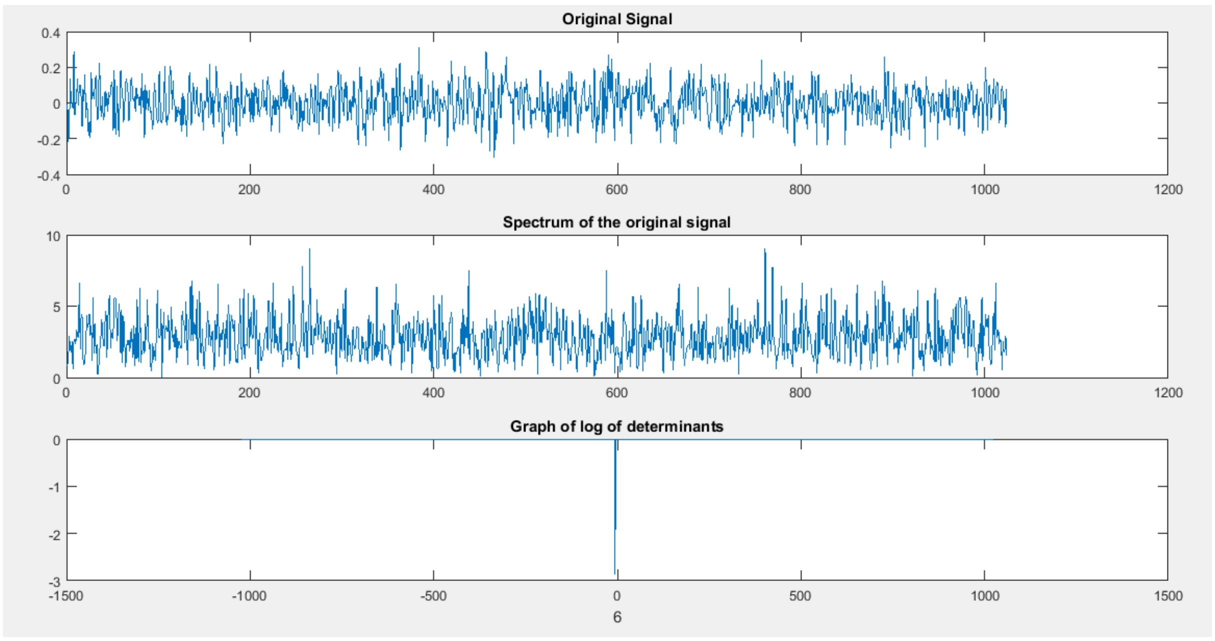


Figure 19: The graph shows the SDOA estimation of random signals with delay of 6 samples

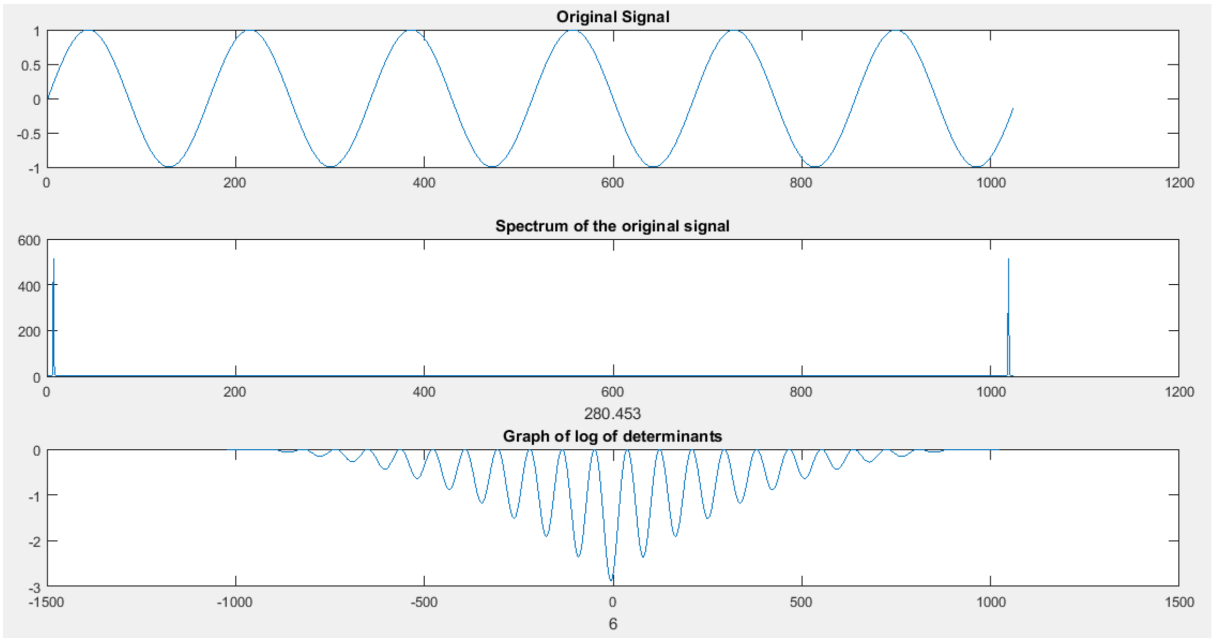


Figure 20: The graph shows the SDOA estimation of sine signals with delay of 6 samples

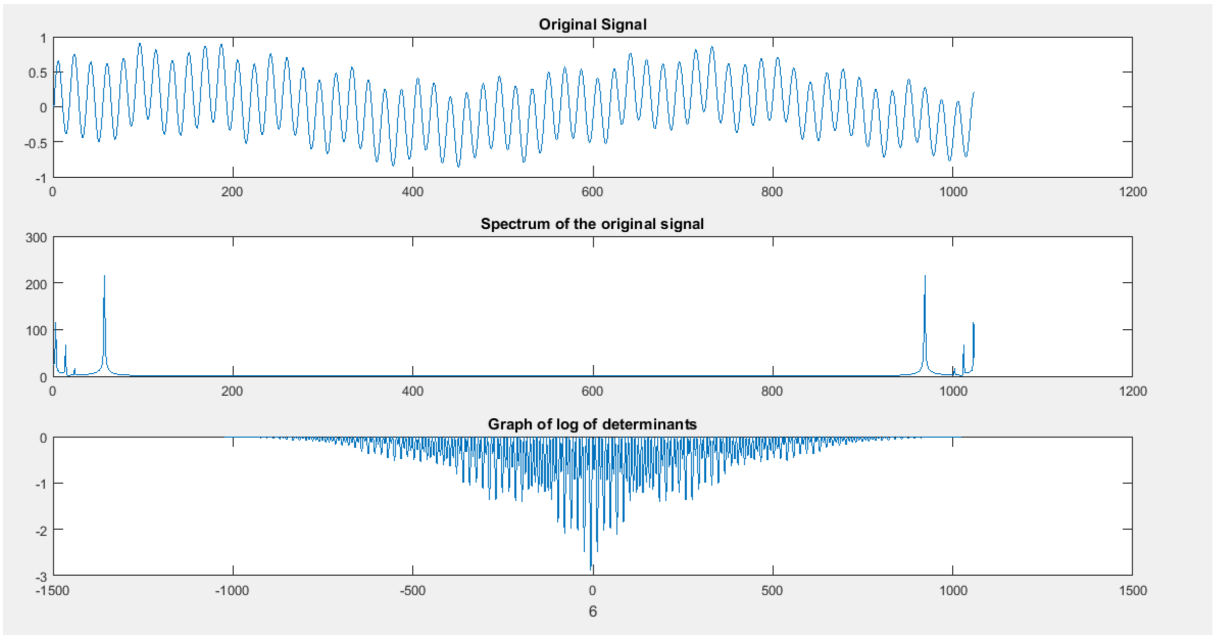


Figure 21: The graph shows the SDOA estimation of multi frequency signals with delay of 6 samples

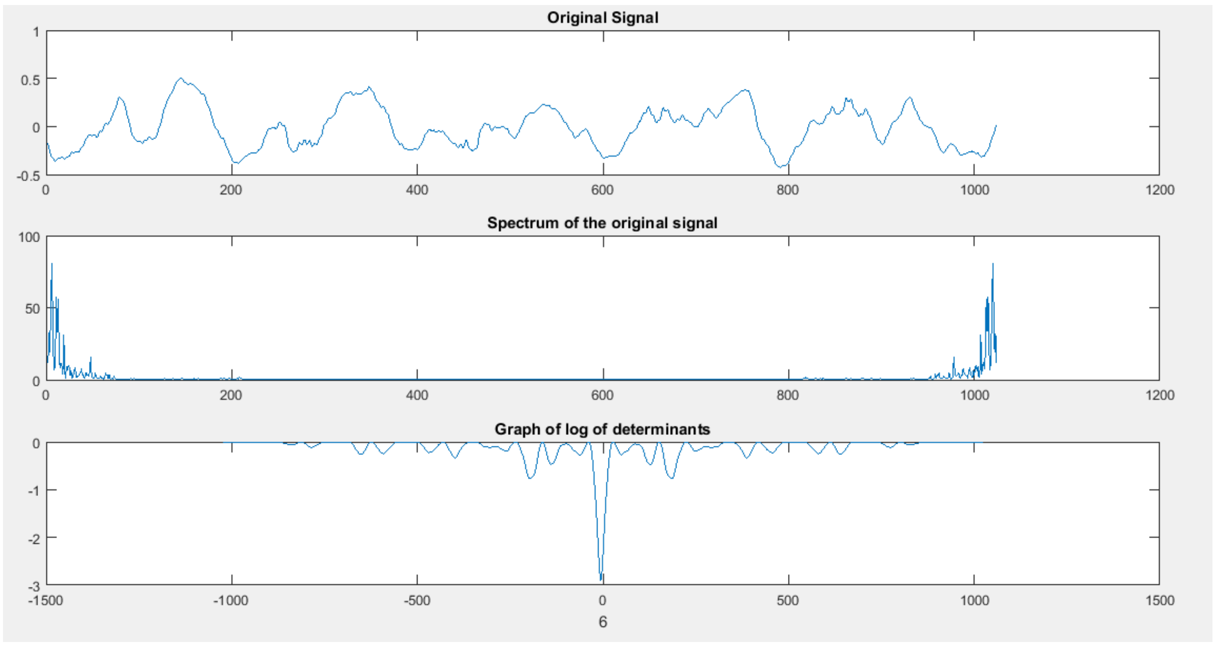


Figure 22: The graph shows the SDOA estimation of speech signals with delay of 6 samples

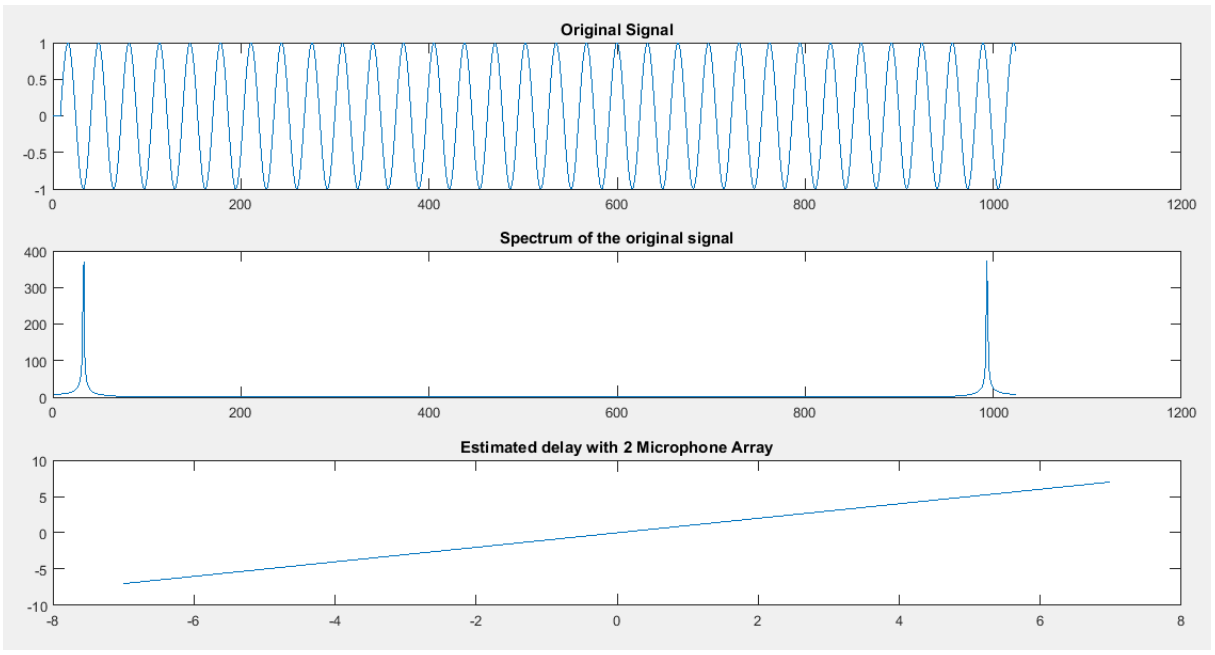


Fig 23: SDOA Estimated for single frequency signal for –SDOA max to + SDOA max

6.2. SDOA estimation with multiple microphones

The SDOA estimation with multiple microphones in an array is similar to using two microphones. However in SDOA estimation with multiple microphones more cross-correlation calculation are done and the dimension of matrix is also greater. The correlation calculations and matrix size are greater due to increase in number of signals. The determinant of the matrix with higher dimension is also more complex compared to determinant of 2 x 2 matrix.

The following code from SDOA estimation MATLAB program creates delayed signals. The number of delayed signals depends on the number of Microphones being used. The variable N represents the number of microphones being used. The original signal for the reference microphone is created outside the for-loop at signal_at_microphone(1,:) array. The for-loop uses signal of microphone at j-1 to create the delayed signal for the microphone at j. The function used for introducing delay to the signal takes in the value of delay and the signal to be delayed.

```
x = signal((i+10)*win+1:(i+11)*win)+randn(1,win)/noise_along_signal;
%n is noise in the background
n = randn(1,win)/noise_in_the_background;
%SNR of the signal to background noise
signal_to_background_noise_ratio = snr(x,n);
%signal and noise is superposed here.
%This is reference Microphone,no delay
signal_at_microphone(1,:) = x + n;
for j=2:N(m)
       signal_at_microphone(j,:) =Introduction_Of_Delay(delay,signal_at_microphone(j-1,:))+...
            Introduction_Of_Delay(j-1,n);
end
```

Code 7: Creation of delayed signals using N number of Microphones

The following code from SDOA estimation program calculates the cross-correlation of signals with each other. The number of total cross-correlation calculations increase with the use of more microphones for SDOA estimation. The number of correlation calculatuons are equal to the $\sum_{i=0}^{N-1} i$, where N is the number of microphones being used. The function Cross_correlation performs cross-correlation calculation of input signal k and l.

```
correlations = zeros(sum(1:N-1),2*win-1);
...

%the following calculates correlations of signals at microphones
        %with eachother
        index = 1;
        for k=1:N(m)-1
            for l=k+1:N(m)
                correlations(index,:) = ...
                Cross_Correlation(signal_at_microphone(k,:),signal_at_microphone(l,:));
                index = index + 1;
            end
        end
```

Code 8: Creation of Correlations for delayed signals

The main difference in SDOA estimation with 2 microphones and multiple microphones is the creation of matrix. The creation of matrix gets slightly complicated for multiple microphones. Assuming the maximum estimable SDOA between two microphones is 7. In case of four microphones, the maximum SDOA between two signals wil be thrice the actual SDOA as shown in the following figure. Therefore the result of correlation of between such signals will be different depending on the maximum SDOA between them. If the delay between adjacent microphones is 4. Then the SDOA between first and third microphone will be 8 and 12 between first and fourth. This difference in correlation of signals shoudl be reflected in matrix creation.The index of correlation coefficients in the matrix will change depending on the coordinate of the matrix that represents the two correlated signals.

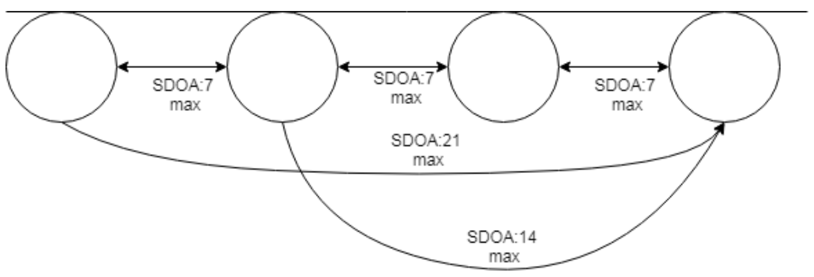


Figure 24: A Microphone Array showing maximum SDOA between Microphones for a SDOA resolution of 7

```
std = zeros(1,size(signals,1));

for i=1:size(signals,1)
    std(i) = Standard_Deviation(signals(i,:));
end

maxT = ceil(size(correlations,2)/2);
delayAt = P-maxT;

R = zeros(4,4);
R(1,1) = 1;
R(1,2) = correlations(1,P)./(std(1)*std(2));
R(1,3) = correlations(2,delayAt+P)./(std(1)*std(3));
R(1,4) = correlations(3,2*delayAt+P)./(std(1)*std(4));

R(2,1) = correlations(1,P)./(std(1)*std(2));
R(2,2) = 1;
R(2,3) = correlations(4,P)./(std(2)*std(3));
R(2,4) = correlations(5,delayAt+P)./(std(2)*std(4));

R(3,1) = correlations(2,delayAt+P)./(std(1)*std(3));
R(3,2) = correlations(4,P)./(std(2)*std(3));
R(3,3) = 1;
R(3,4) = correlations(6,P)./(std(3)*std(4));

R(4,1) = correlations(3,2*delayAt+P)./(std(1)*std(4));
R(4,2) = correlations(5,delayAt+P)./(std(2)*std(4));
R(4,3) = correlations(6,P)./(std(3)*std(4));
R(4,4) = 1;
```

Code 9: Creation of Matrix for SDOA estimation using 4 Microphones

For the example of SDOA estimation with four microphones, the spatial correlation matrix will be created as follows:

$$R_a(p) = \begin{bmatrix} \sigma_{y1}^2 & r_{a,y1y2}(p) & r_{a,y1y3}(2p) & r_{a,y1y4}(3p) \\ r_{a,y1y2}(p) & \sigma_{y2}^2 & r_{a,y2y3}(p) & r_{a,y2y4}(2p) \\ r_{a,y1y3}(2p) & r_{a,y2y3}(p) & \sigma_{y3}^2 & r_{a,y3y4}(p) \\ r_{a,y1y4}(3p) & r_{a,y2y4}(2p) & r_{a,y3y4}(p) & \sigma_{y4}^2 \end{bmatrix}$$

6.3. Performance of SDOA estimation with multiple Microphones

The graph of det(Ra(p)) for SDOA estimation with different microphones in the microphone array are given below. The MCC algorithm takes advantage of the redundancy of the microphones to increase the performance of SDOA estimation. In comparison with the graph of determinant of using 2 and 4 microphones, the graph for 8 microphones represented by yellow dots has a deeper drop and the index of drop is more distinguishable from neighboring values. The graph of determinants for 4 microphones represented by red dots gives a better estimation of SDOA than of 2 microphones represented by blue dots. Similarly graph for 8 microphones represented by yellow dots gives a better estimation of SDOA than 4 microphones. The SDOA estimation program is tested with different types of test signals such as random signal, sine wave signal and speech/music signal(seen in figure 25 to 28). In different scenerios, the SDOA estimation using 8 microphones performs better than 2 and 4 microphones.

The SDOA estimation program is tested with increasing frequency of sine wave for different number of microphones. The size of the buffer length used for SDOA estimation is 1024. The frequency of sine wave is increased from 1 to 512,for 512 SDOA estimations. For every next test, the frequency of sine wave is increased by value of 1. The graphs in figure 29 to 31 show the result of SDOA estimation with increasing frequency of sine test signal. The estimation f SDOA using 8 microphones performs the best, with the least incorrest estimations. The graph in figure 29 shows the result of SDOA estimation using 2 microphones with 66 incorrect estimations out of 512. The figure 30 and 31 for SDOA estimation using 4 and 8 microphones gives 20 and 6 incorrect estimations respectively out of 512.Therefore it is clear that the performance of SDOA estimation increases with the use of more microphones in adverse acoustic environment. It can also be observed that all of the wrong estimations happen for high frequency signals. However the maximum allowable frequency of the source for SDOA estimation is much lower.

The table 5 shows the number of incorrect SDOA estimation out of 512 for a given delay using 2, 4 and 8 microphones. The results shows that the performance of SDOA estimation imporves with more microphones in the microphone array. The table also shows better performance of MCC algorithm for smaller SDOA values. The performance of SDOA estimation is decreases for large SDOA values.

DELAY	N=2	N=4	N=8
-4	31	7	3
-7	75	21	6
2	1	0	0
4	23	7	4
7	66	20	6

Table 5: Number of incorrect estimations of MCC algorithm using N microphones for signals created with increasing frequency for every next test

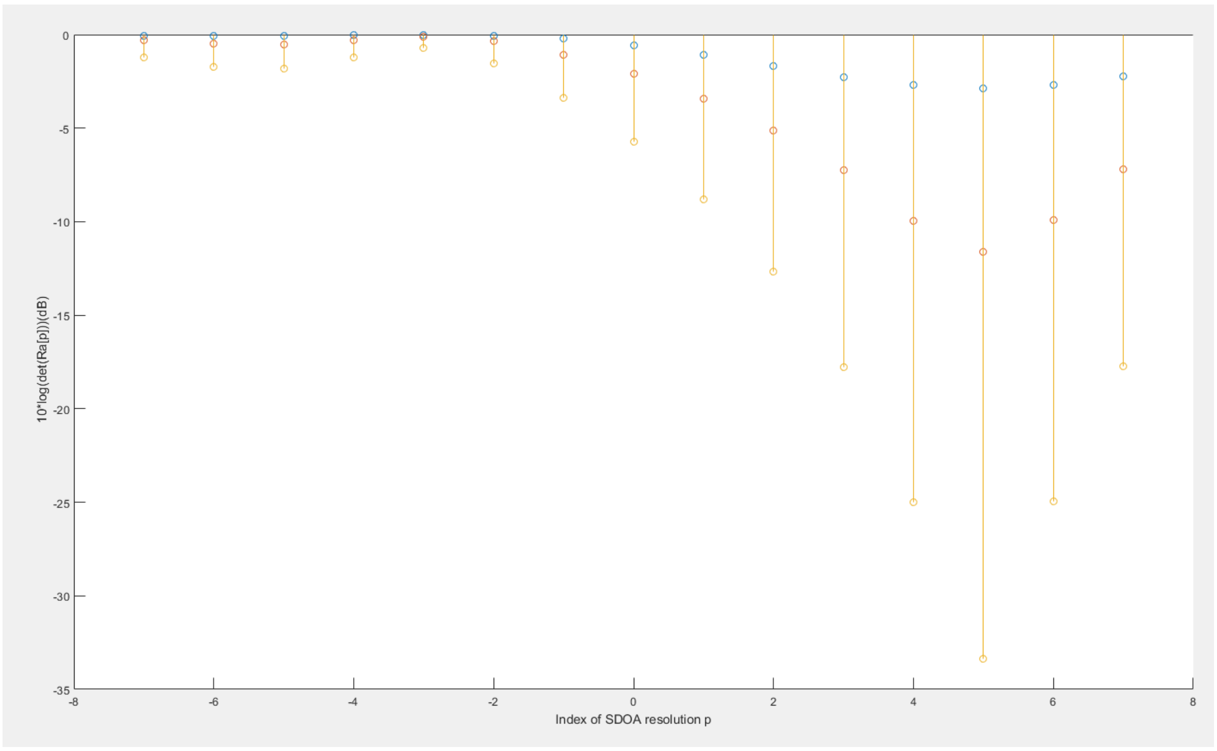


Figure 25: Graph Ra(p) for two,four and eight Microphones using random test signals

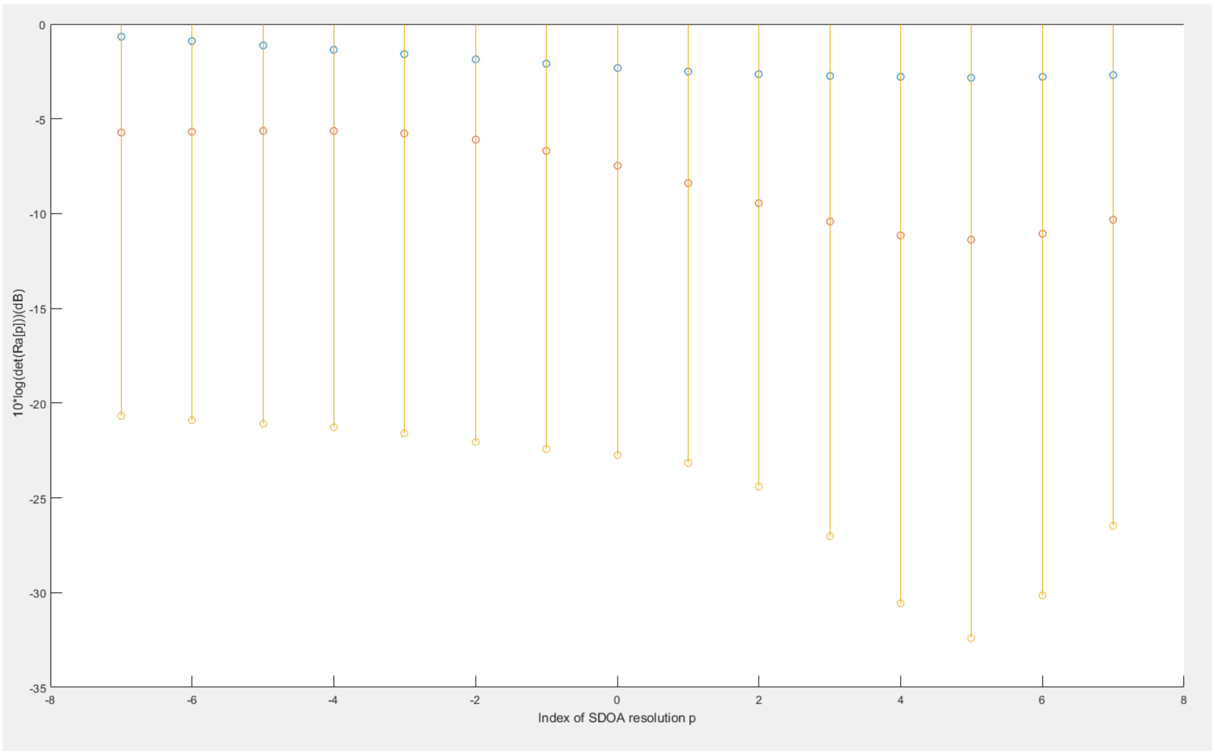


Figure 26: Graph Ra(p) for two,four and eight Microphones using sine wave test signals

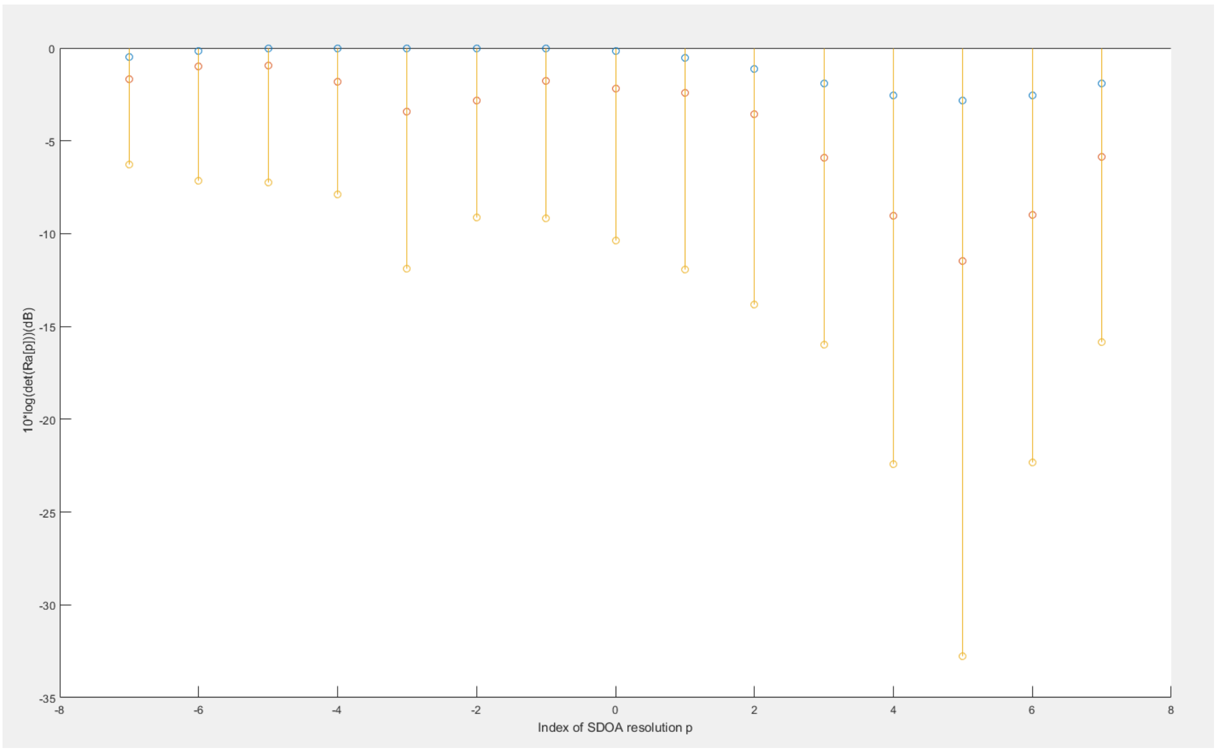


Figure 27: Graph Ra(p) for two,four and eight Microphones using multi frequency test signals

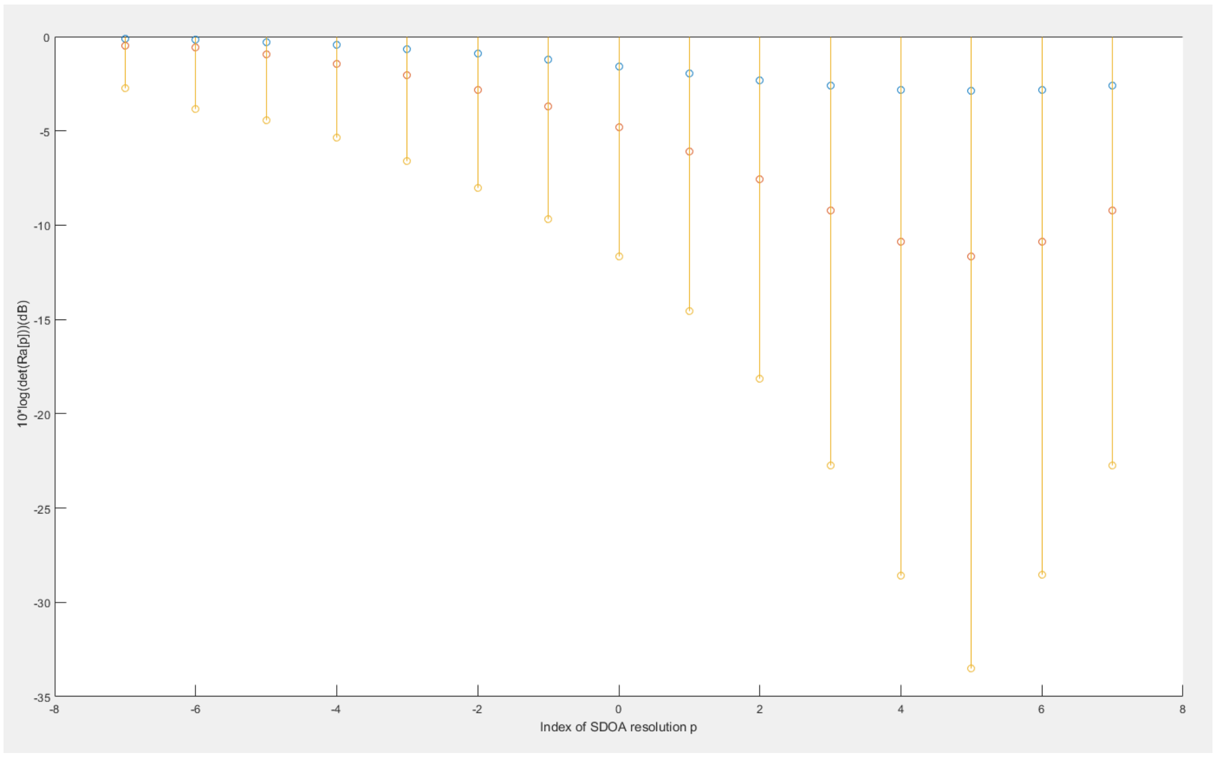


Figure 28: Graph Ra(p) for two,four and eight Microphones using speech/music test signals

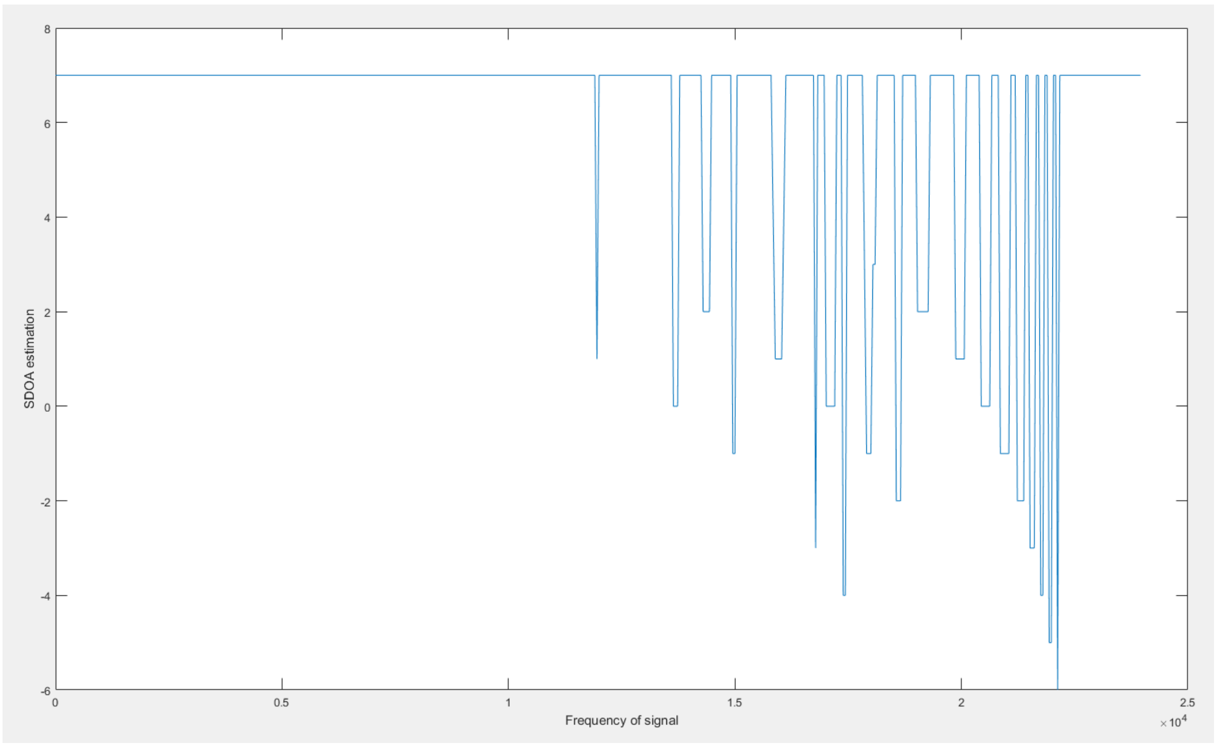


Figure 29: Graph shows 66 incorrect SDOA estimation using 2 Microphones

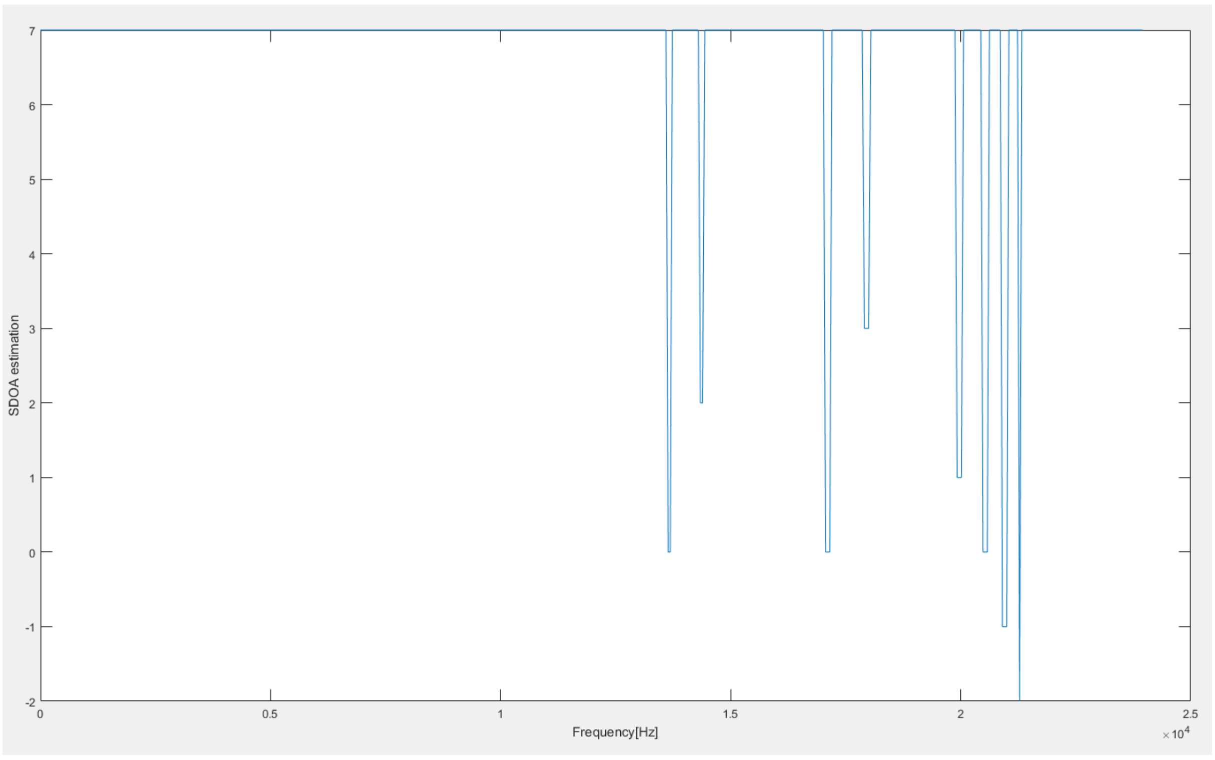


Figure 30: Graph shows 20 incorrect SDOA estimation using 4 Microphones

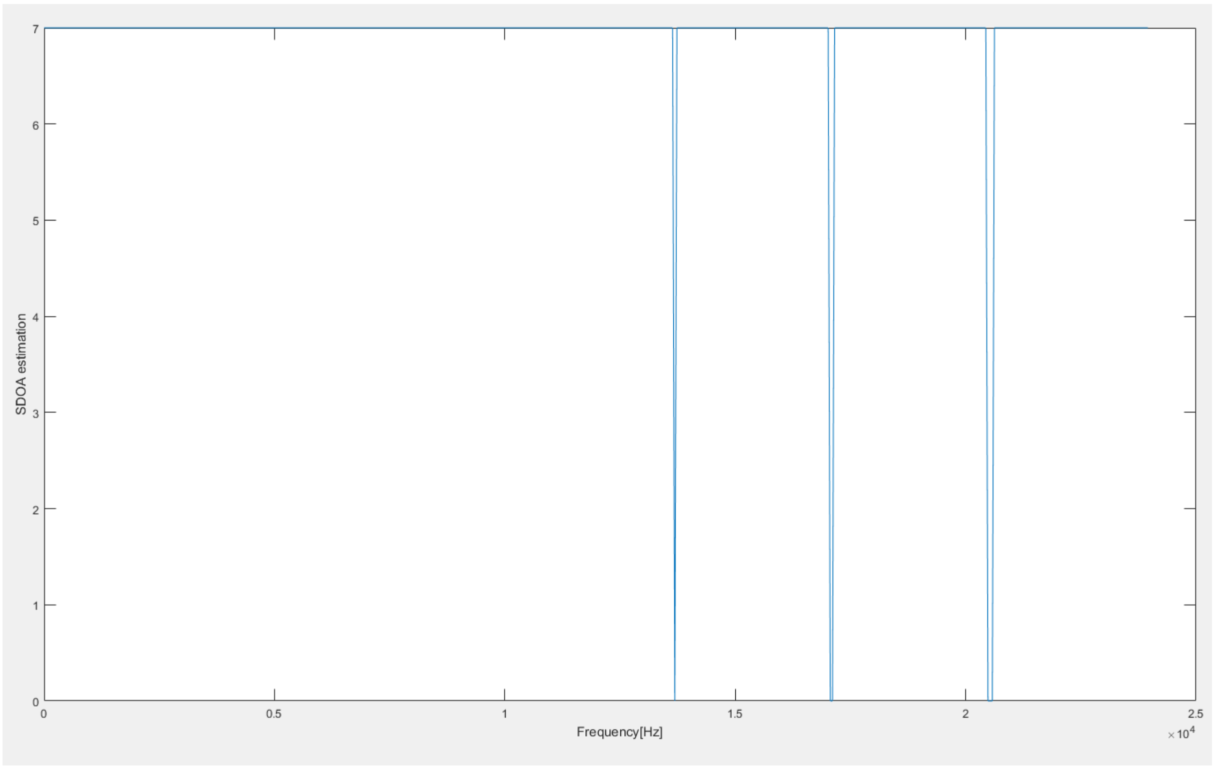


Figure 31: Graph shows 6 incorrect SDOA estimation using 8 Microphones

6.4. Source Localization Program Code

The functions and files being used in the MATLAB program are explained below:

- Select_Operation
    - This is the main file of MATLAB program that allows the user to select the operation to be implemented. This file allows the user to select from four different operations that are Source Localization, SDOA Estimation, creating test sequences and creating a look up table of position of source for different values of SDOA.
- SourceLocalization()
    - This operation estimates the position of the source using the user input delay values to create signals at microphone arrays. The MCC algorithm is used to estimate the SDOA at the microphone arrays. The values of SDOA are used to determine the angle and the position of the source.
- SDOA_Estimation()
    - This file estimates the SDOA between adjacent signals at a microphone array. The user chooses the number of microphones and the type of signal for testing the program. The delayed signals are created and their cross-correlations are calculated. A function is called that returns an array of determinants that is used to determine the SDOA between two adjacent microphones.
- Create_Test_Signals()
    - This file creates test sequences for DSP program on C compiler in text format and on CCS in header format. The user has the option to choose the type of signal and delay to be introduced to the signals. The program writes 8 delayed signal to the text file. The first four channels represent signals at array1 and other four represent signals at array 2. A file is created with the name of the signal selected and of the correct format.
- Create_File_Of_Expected_Results()
    - This file creates a look up table of angles and the position of the source for different values of SDOA at the microphone arrays.
- Determinants_Of_Matrices()
    - This function calls the function for creating matrices, calculates the determinants and returns an array of determinants for SDOA estimation.
- Create_Matrix()
    - This file is used to create spatial correlation matrices of dimension 2x2, 4x4 or 8x8. The function expects correlations, signals, index P of SDOA range and the dimension of matrix.

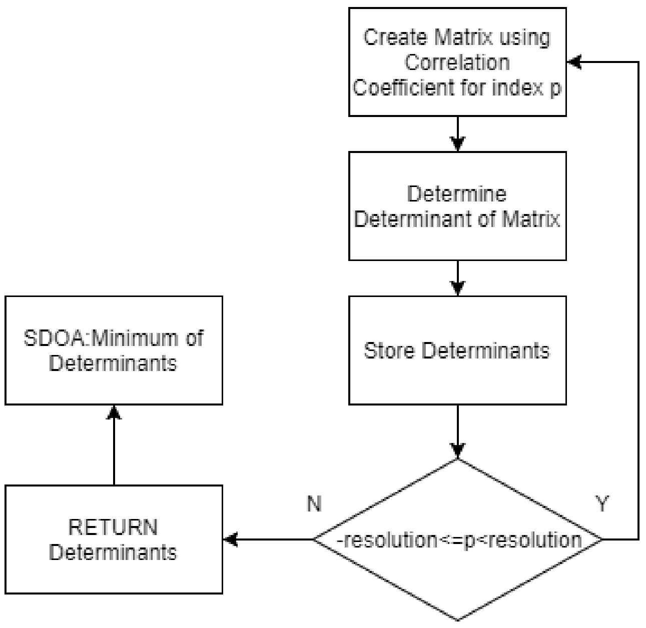

Create Matrix using Correlation Coefficient for index p
Determine Determinant of Matrix
Store Determinants
-resolution<=p<resolution
Y
N
RETURN Determinants
SDOA:Minimum of Determinants

Figure 32: Flowchart of SDOA estimation though matrix creation

- Cross_Correlation()
  - This function simply calculates the cross-correlation of two signals using the multiplication property of FFT.
- Introduction_Of_Delay()
  - This function uses the frequency domain property of signals to introduce delay to a signal.
- Standard_Deviation()
  - This function calculates standard deviation of a signal.
- Find_Index_Of_Minimum()
  - This function returns the index of the minimum determinant value.
- initializeParameters
  - The file initializes parameters such as speed of sound, maximum frequency, sampling frequency and buffer length of signal for Source Localization program. The maximum spacing between adjacent microphones and maximum estimable SDOA is also determined in this file.
- Signal_Filter
  - The function for filtering signals expects the original signal, the sampling frequency and the maximum allowable frequency in the signal. The function creates a low pass filter and filters out the original signal to get rid of frequencies higher than the maximum allowable frequency.
- SelectMicAndSignalType
  - The file allows the user to select the number of microphones to be used for Source Localization or SDOA estimation. The user can also select the type of signal to be used to test Source Localization or SDOA estimation program.
- createSignal
  - The file creates a signal for reference microphone depending on the type of the signal selected in SelectMicAndSignalType file.

- Power_Calculation
    - The function expects a signal and returns the value of power of the signal.

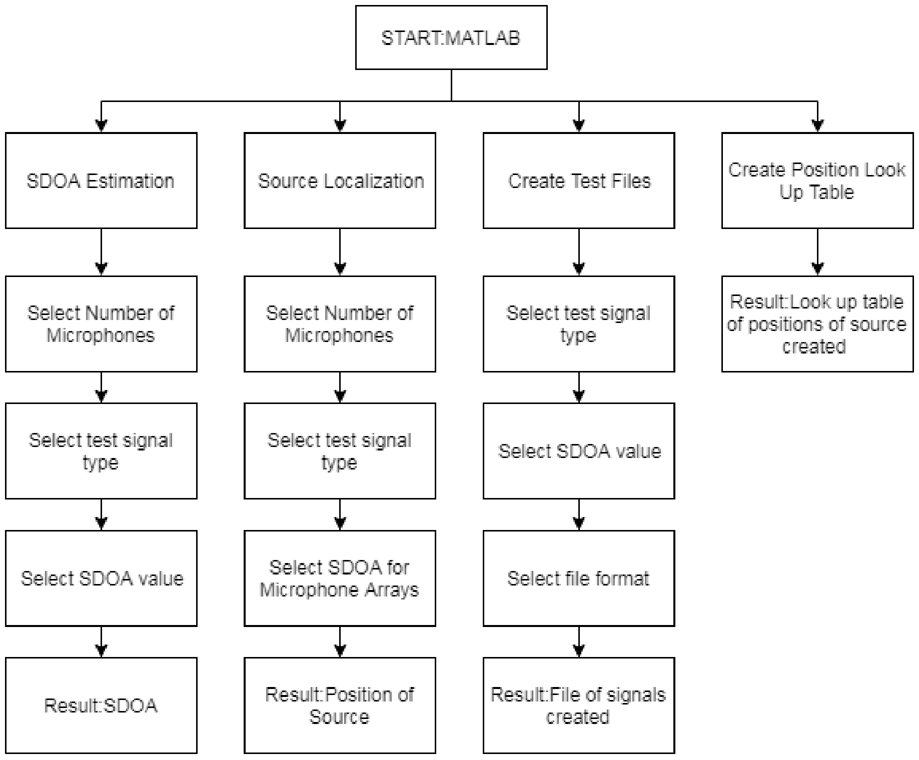

START:MATLAB
SDOA Estimation
Source Localization
Create Test Files
Create Position Look Up Table
Select Number of Microphones
Select Number of Microphones
Select test signal type
Result:Look up table of positions of source created
Select test signal type
Select test signal type
Select SDOA value
Select SDOA value
Select SDOA for Microphone Arrays
Select file format
Result:SDOA
Result:Position of Source
Result:File of signals created

Figure 33: Flow diagram of operations of MATLAB program

6.5. Test sequences and source position look up table

The position of the source for different SDOA values are shown in table 6, in addition to angle and distance information of the source to the microphone arrays. The table is created by Create_File_Of_Expected_Results() function of MATLAB program. The values of SDOA represent SDOA estimated at the two microphone arrays. The dist value in the table is the difference of distance travelled by the signals between two adjacent microphones. The values of alpha and beta are the angles between source wave front and a microphone array. The variable d gives the distance travelled by the signals of source to a microphone array. The table shows the position of the source in terms of x and y coordinates from the reference microphone.

SDOA$_1$	**SDOA$_2$**	**Dist$_1$(m)**	**Dist$_2$(m)**	**Alpha(°)**	**Beta(°)**	**d$_1$(m)**	**d$_2$(m)**	**X(m)**	**Y(m)**
-7	-7	-0.0496	-0.0496	7.402	172.6	0.0000	0.0000	0.000	0.000
-7	-6	-0.0496	-0.0425	7.402	148.21	0.3120	1.2758	1.265	0.164
-7	-5	-0.0496	-0.0354	7.402	135.10	0.2116	1.1595	1.149	0.149
-7	-4	-0.0496	-0.0283	7.402	124.52	0.1731	1.1073	1.098	0.142
-7	-3	-0.0496	-0.0213	7.402	115.15	0.1528	1.0739	1.064	0.138
-7	-2	-0.0496	-0.0142	7.402	106.46	0.1408	1.0486	1.039	0.135

Table 6: Expected angle and position of source for delays at the two Microphone Arrays

7. Implementation of source localization in C

This section discusses the process of source localization on the existing DSP program. The files present in the program are discussed and their functions are explained. The steps that are incorporated in finding the position of the speaker in DSP program will also be discussed.

7.1. Program Code

The implementation of source localization program includes various C and header files. The C files contain functions necessary for implementing SDOA estimation. There are also C files for declaration and initialization of global variables and data structures that are necessary for operations performed in the program.

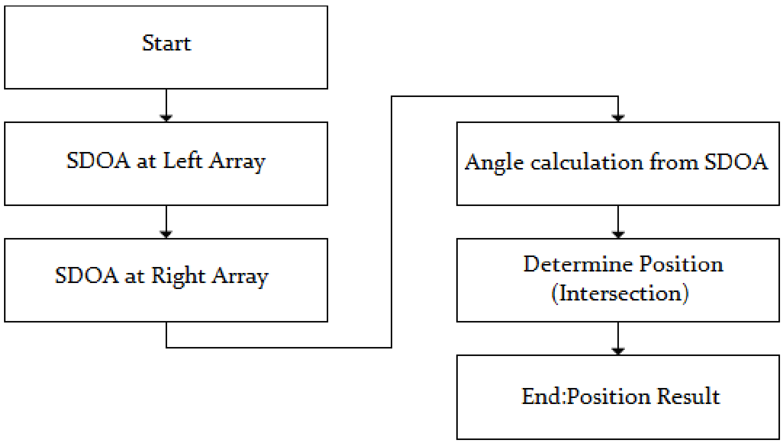


Figure 34: Flow diagram representing the steps needed to detect position of the source (from reference [16])

Program Files

- pcm3003.c
    - This is the main file of the source localization program. The main is an extension of main file of original DSP echo program. The code for setting up DSP Board is contained in this file. The file contains adc and dac interrupts for input and output of the DSP Board. In addition to ADC and DAC interrupts, data can also be read from a text file in C compiler. The function of SDOA estimation, Angle and Position calculation are called when the adcbuffers are full. A function named initGlobs() is called once in the main to initialize twiddle factors for FFT calculation. Another function tw_gen () called once in the main generates twiddle factors for the FFT from DSP library of Cx66 cores. The file also contains an unimplemented code for localization of two

sources of sound. The code for demonstration of position of speaker using hyper-terminal is also included in the file.

- CalculationOfSDOA_etc.c
  - This C file contains functions for SDOA estimation, Angle calculation and finding position. This file has been included in the main file and must not be included in the project. During the extension of the program, an FFT from DSP library of Cx66 cores is implemented in this file. The C callable FFT is also present in the file.
- functionsCalculateSDOA.c
  - This C file contains functions for SDOA estimation such as calcEnergy, copy2ComplexStruct, calcVariance, conjComplexConjMul and findIndexOfMinimum.
- functionsPositionFinding.c
  - This C file contains functions for determining position of the source, such as ULA_calcAngle, CreateLookupForAngleCalculation and positionFinding.
- globals.c
  - The global file initializes and declares various global variables. The file contains pointers and complex data structures for FFT and IFFT calculation. The file also contains function for initialization of twiddle factors and for creation of look up table for angle calculation.
- globalConstants.c
  - The globalConstants C file includes constant global variables of the program.
- CFFT_fromTI.c
  - This C file contains C callable function for calculating FFT.
- CFFT_bitreversal.c
  - This C file reverses bit of a given variable for FFT calculation.
- cfftr2_dit.sa
  - The file contains C equivalent of the assembler FFT. The routine performs the Decimation-in-Time (DIT) Radix-2 FFT of the input array x.[21]
- DSPF_sp_fftSPxSP_opt.c
  - The file contains the optimized version of FFT for C6657 core. This file is used for FFT calculation on the C6657 DSP Board.

7.2. Program functions for Source Localization

In this section the functions used for determining the position of source of sound have been discussed. The implementation and purpose of the functions have also been explained in detail.

[21]Marcus Muller Thesis

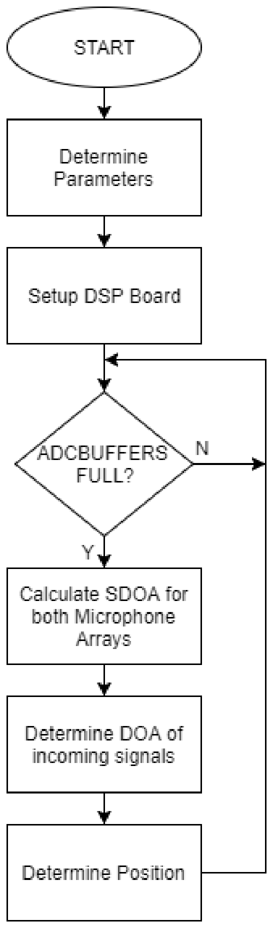


Figure 35: Flowchart of position estimation in DSP program

7.2.1. Energy calculation

Energy of signal is calculated in function calcEnergy(). The energy of signal is calculated by summing the square of scaled values of the signal. The function return -1 if signal energy is lower than energy limit defined in defines file and returns value of signal energy if the energy is greater than energy limit. The signal is voiced when the energy of signal is returned by the function, or else the signal is non-voiced. This step is important in the SDOA estimation, since not all buffers contain actual signal from the source. The non-voiced buffers which merely contain noise from the environment do not need SDOA estimation.

7.2.2. Variance Calculation

The variance of signal is calculated in function calcVariance(). This function calculates the variance over the real part of buffer g_cmpx_SigBuf. The variance of the signal is calculated by squaring the real part of the signal and summing up the values. The variance of the signal is needed for creating spatial correlation matrix for SDOA estimation.

7.2.3. Fast Fourier Transform(FFT) calculation

The FFT in the DSP program is implemented by C callable FFT and by FFT from DSPLIB library of C66x. The FFT is calculated by calling either function cfftr2_dit() or DSPF_sp_fftSPxSP_opt() for C callable FFT or DSP library FFT respectively. However, the later FFT cannot be called in C compiler, since it is specifically designed for DSP Board. The implementation of FFT is needed for FFT based cross-correlation calculation. The same FFT functions are used to perform IFFT operation on Fourier transformed signals.

7.2.4. Cross-Correlation calculation

The equations described below are used to calculate Cross Correlation of signals in time domain. The first equation below for cross correlation is a biased estimator equation whereas the second is unbiased equation.

$$\hat{r}_{y2y1}^{CC}(p) = \begin{cases} \dfrac{1}{K} \displaystyle\sum_{i=0}^{K-p-1} y_1(k+i)y_2(k+i+p) & p \geq 0 \\ \hat{r}_{y2y1}^{CC}(-p), & p < 0 \end{cases}$$

$$\hat{r}_{y2y1}^{CC}(p) = \begin{cases} \dfrac{1}{K-p} \displaystyle\sum_{i=0}^{K-p-1} y_1(k+i)y_2(k+i+p) & p \geq 0 \\ \hat{r}_{y2y1}^{CC}(-p), & p < 0 \end{cases}$$

[22]

The calculation of cross correlation in time domain is computationally expensive. For this reason, multiplication property of Fourier Transform is used to calculate cross-correlation. The multiplication property of FFT states that multiplication in Frequency Domain is convolution in time domain. The signals to be correlated will be multiplied in Frequency Domain and then Inverse Transformed to get the correlated signal in time domain.

In the program, correlation calculation requires a few pre calculation steps. The length of the resulting correlation is always double the length of the original signal. Therefore, both signals are zero-padded to twice the original length. In the next step, signals to be correlated are converted to frequency domain using FFT. Complex conjugate multiplication is performed on the Fast Fourier transformed signals, which is necessary for IFFT calculating using an FFT function. The steps for calculating Complex Conjugate multiplication of two complex signals is shown below:

[22] Microphone Array Signal Processing Volume 1 of the series Springer Topics in Signal Processing Chapter 9 pp 8

$$Input\ value: bufferA = (a + jb),$$

$$bufferB = (c + jd)$$

$$Operation: conj(\ bufferA * conj(bufferB)\ )$$

$$conj(\ (a + jb)\ * conj(c + jd)\ )\ =\ conj(\ (a + jb) * (c - jd)\ )$$

$$conj(ac - jad + jcb - j^2bd)$$

$$ac + jad - jcb + bd$$

$$Real: ac + bd$$

$$Imag: (ad - cb)$$

For the implementation of IFFT using cfftr2_dit() function, the bits of the signal after Complex conjugate multiplication are reversed using bitrev(). At this point the signal would be ready for Inverse Fourier Transform calculation. After performing IFFT operation on the signal, bits of the Inverse Fourier transformed signal are reversed once more using bitrev ().

Finally, the result of IFFT using C callable FFT or DSPLIB library FFT is scaled and shifted by half correlation buffer length to get the cross-correlation of the two original signals by convertCmpxCorrResultToFloatWithFftShift ().[23]

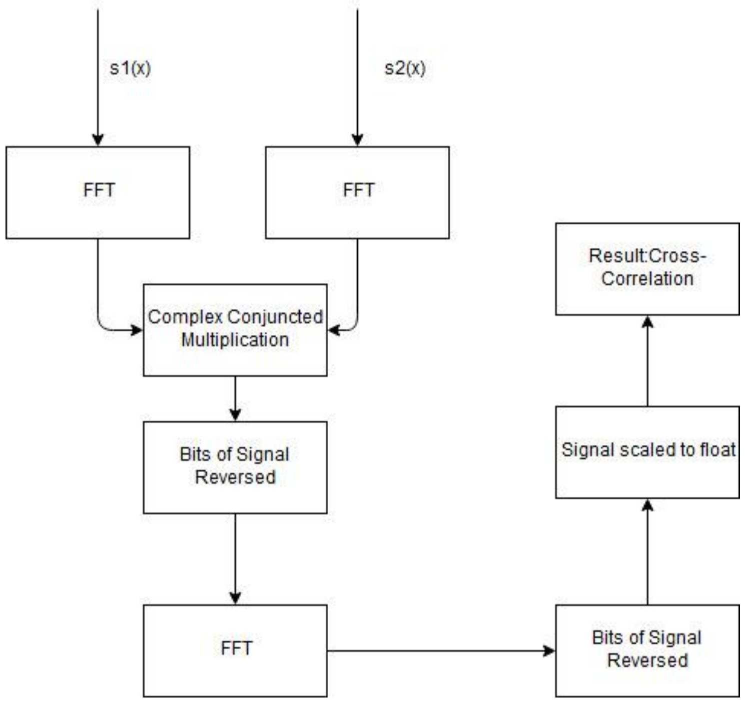


Figure 36: Flow diagram shows the steps needed to correlate two signals using C callable FFT

[23]Marcus Muller Thesis

7.2.5. Setting up of Matrix

The dimension of matrix depends on the number of microphones being used in the microphone array for SDOA estimation. The dimensions of the matrix in the DSP program will be 4x4, since there are four microphones in an array. The variance of the signals is written in the diagonals of the matrix. The Cross-Correlation coefficients are written in the rest of the matrix coordinates. The value of index of cross-correlation coefficient changes for different coordinates of the matrix. The position of coordinates in the matrix represents the position of correlated signals in the microphone array. The maximum SDOA between two signals depends on the position of their microphones. The graph in figure 37 shows correlation with a maximum at 5, whereas figure 38 shows a graph of correlation with a maximum at 15. The delay in signals of correlation of figure 38 is thrice more than the delay between signals of correlation of figure 37. This is the reason of having different index of cross-correlation coefficients in matrix at different coordinates. The matrix in the DSP program is created by function createMatrix4x4Ra ().

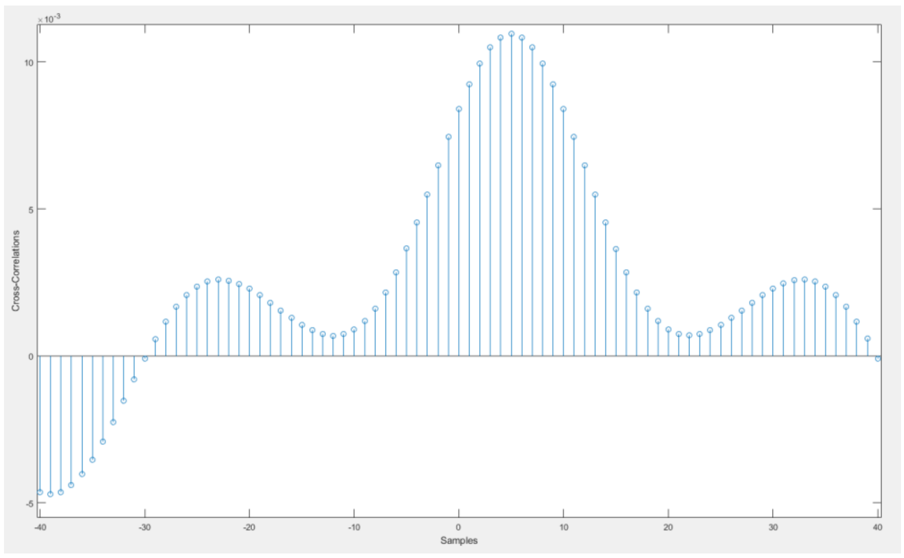


Figure 37: Correlation result of two signals in time domain with a delay of 5 samples

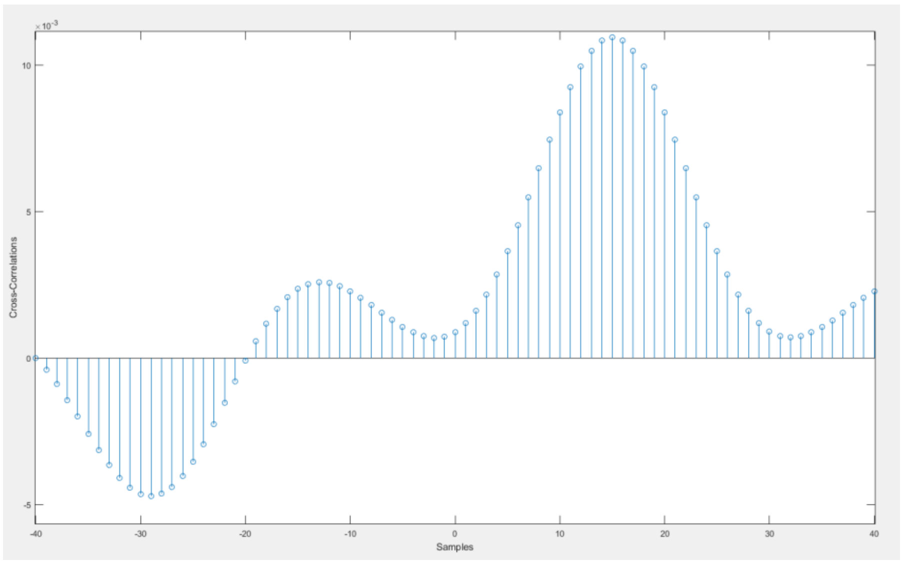


Figure 38: Correlation results of two signals in time domain with a delay of 15 samples

7.2.6. Determinant Calculation

The determinant of a matrix needs to be determined for SDOA estimation. The index of minimum determinant gives the value of SDOA. The determinant of matrix A is denoted by det(A). The function calcDeterminant4x4Matrix () is called to calculate the determinant of a 4x4 matrix. The calculation of determinant for a 2 by 2 matrix is simple, which is given by the formula:

$$\det\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = ad - bc.$$

The calculation of determinant of a 4x4 matrix explained below is slightly more complicated.

$$\mathrm{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix}$$

$$\det(A) = a_{1,1} \cdot \det\begin{pmatrix} a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix} - a_{2,1} \cdot \det\begin{pmatrix} a_{1,2} & a_{1,3} & a_{1,4} \\ a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix} + a_{3,1} \cdot \det\begin{pmatrix} a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,2} & a_{2,3} & a_{2,4} \\ a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix} - a_{4,1} \cdot \det\begin{pmatrix} a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,2} & a_{3,3} & a_{3,4} \end{pmatrix}$$

$$\det\begin{pmatrix} a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix} = a_{2,2} \cdot \left(a_{3,3}a_{4,4} - a_{3,4}a_{4,3}\right) - a_{3,2} \cdot \left(a_{2,3} \cdot a_{2,3} - a_{2,4} \cdot a_{2,4}\right) + a_{4,2} \cdot (a_{2,3} \cdot a_{2,3} - a_{2,4} \cdot a_{3,3})$$ [24]

7.2.7. Estimation of SDOA

A certain portion of cross-correlation coefficients are used to create the matrices depending on the maximum estimable SDOA. If maximum estimable SDOA is 7, then 7 coefficients to the left and right of the center of cross-correlation will be used for SDOA estimation. As a result, 15 values of determinants of matrices will be calculated .The determinants are plotted in a logarithmic graph for better scaling and readability. The logarithmic graph of determinants is used to find SDOA which is index of minimum. The SDOA is found from the det(Ra(p) by the function findIndexOfMinimum ().

7.2.8. Looking up for Angle

The calculation of angle from the SDOA is performed by the function CalculationOfAngle. A table of angles is created for the corresponding SDOA, using the function CreateLookupForAngleCalculation ().The function considers the sampling frequency and distance between microphones to determine the angle for the SDOA. The function ULA_calcAngle_Lookup() searches the array of angles for the SDOA index, and returns the angle at the index of SDOA value.

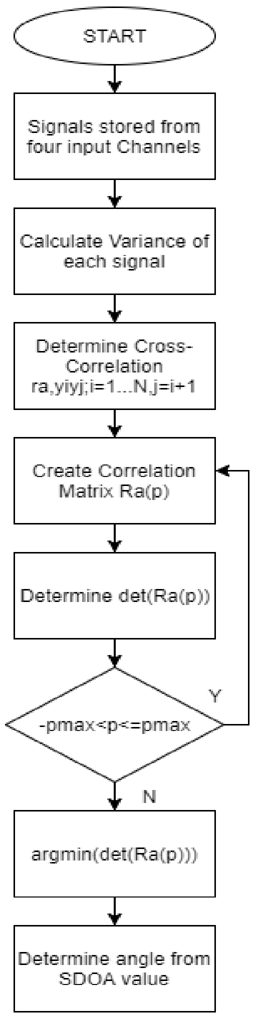
7.2.9. Calculation of Position

The position of the source is calculated using function positionFinding() which is called in the function calculationOfPosition() . The position of the source is determined using sine rule, given DOAs and distance between the Microphone Arrays.

7.3. Process of Source Localization

A ping-pong array adcbuffer of 8 channels is used for SDOA estimation in the DSP program. The channels 0-3 of adcbuffer are used for SDOA estimation of microphone array on left and channels 4-7 for SDOA estimation of microphone array on right. The energy of one of the channels of adcbuffer is calculated when the adcbuffers are full. If the signal in adcbuffer is voiced, the program proceeds to SDOA estimation using MCCC algorithm. In this way, SDOA of two microphone arrays is estimated. If the values of SDOA are valid the angle and position of the source is estimated.

[24]Marcus Muller Thesis


START
Signals stored from four input Channels
Calculate Variance of each signal
Determine Cross-Correlation ra,yiyj;i=1...N,j=i+1
Create Correlation Matrix Ra(p)
Determine det(Ra(p))
-pmax<p<=pmax
Y
N
argmin(det(Ra(p)))
Determine angle from SDOA value
Figure 39: Flowchart for SDOA and angle calculation in DSP program

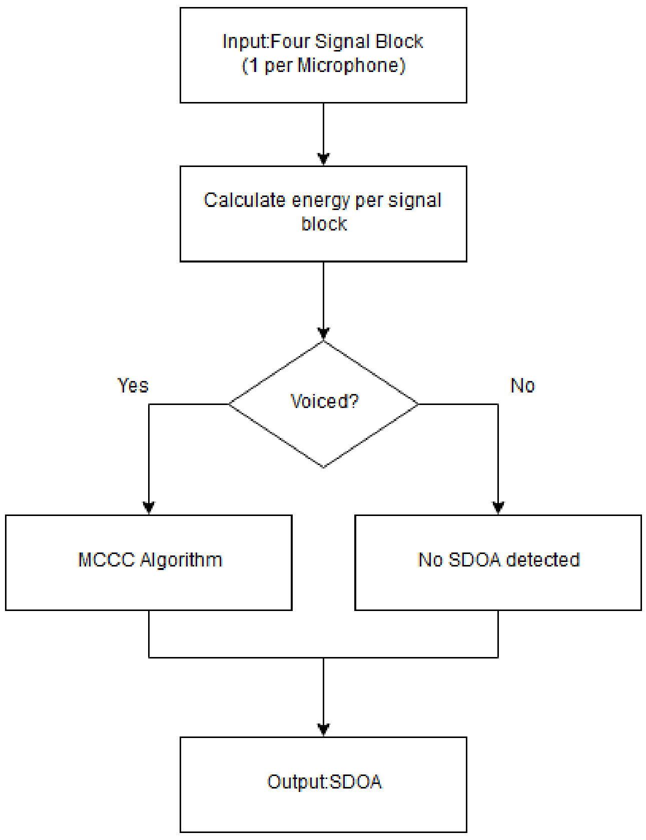


Figure 40: Flow diagram representing the steps needed to estimate SDOA (from reference [16])

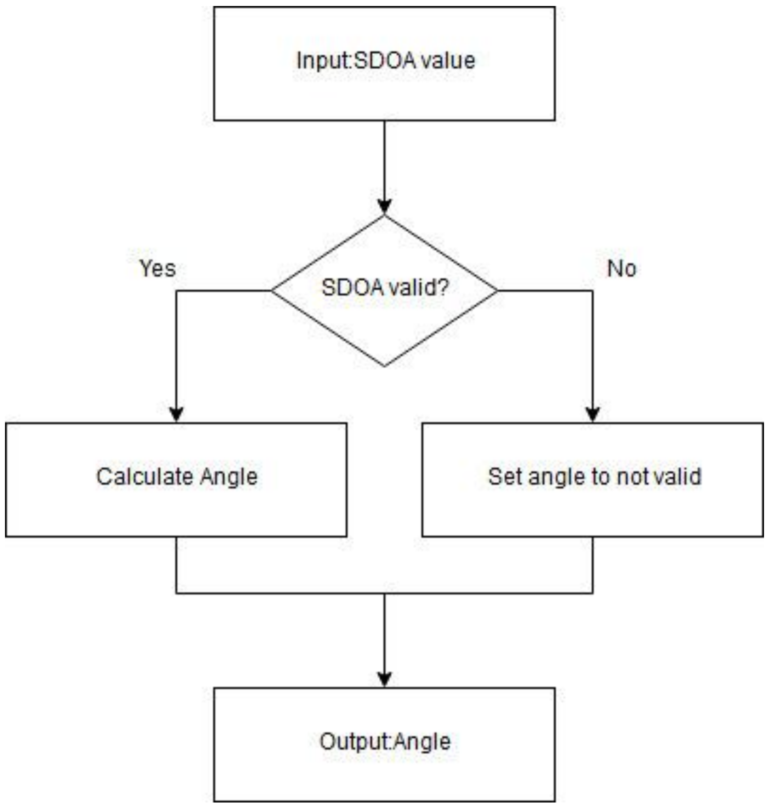


Figure 41: Flow diagram representing the steps needed to determine angle from SDOA (from reference [16])

7.4. Usage of Histogram

The source localization DSP program is relatively fast in estimating SDOA and determining the position of speaker. The program can detect sudden small movements of the source depending on the resolution of SDOA estimation. A histogram is created to avoid abrupt changes in the speaker position. For this reason the SDOA values are stored in histogram buffer and position of the source is not determined directly after estimation of a single SDOA .If the histogram has enough valid values above a certain threshold, the value of maximum occurring SDOA is determined from the histogram. The position of the source is determined, once the SDOA from histogram for both Microphone arrays is determined.

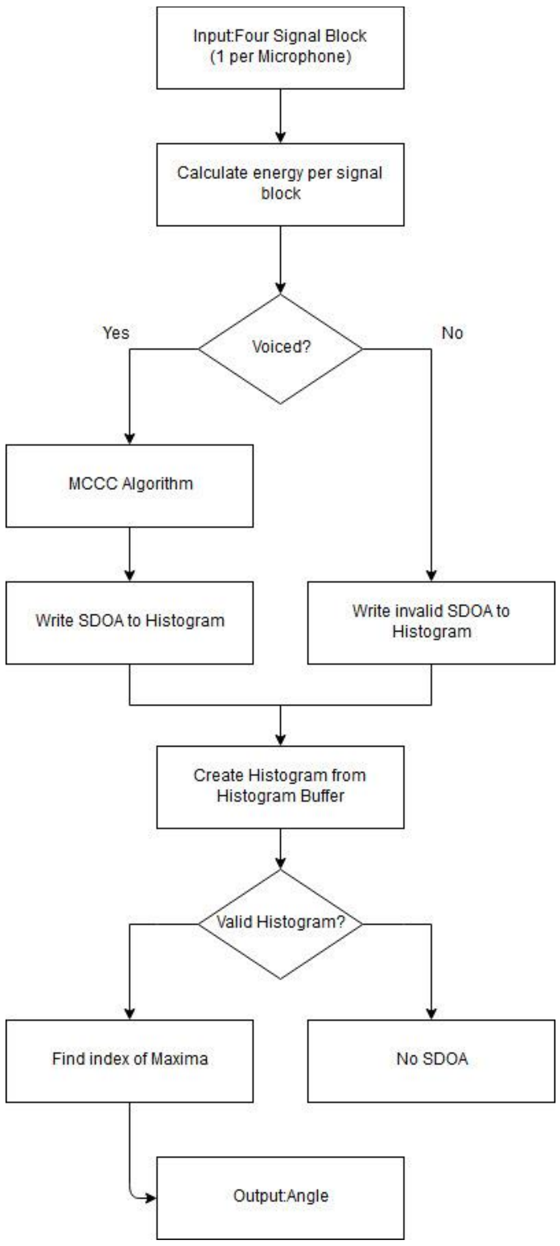

Input:Four Signal Block (1 per Microphone)
Calculate energy per signal block
Yes
Voiced?
No
MCCC Algorithm
Write SDOA to Histogram
Write invalid SDOA to Histogram
Create Histogram from Histogram Buffer
Valid Histogram?
Find index of Maxima
No SDOA
Output:Angle

Figure 42: Flow diagram representing the use of histogram buffer in source localization process (from reference [16])

8. DSPLIB library of C66x processor

The release of Processor SDK RTOS for C66x processor comes together with new optimized libraries developed for specific roles. The DSPLIB library is part of the processor SDK RTOS release and was optimized to run on different device cores. There is a specific optimized version of the functions for a specific processor and there is a standard version of the functions which can be compiled or used by any processor.

8.1. FFT function in DSPLIB

The DSPLIB contains many FFT routines. There are 16 different variants of FFT and IFFT algorithm for C66x core version, such as single-precision floating point, and double-precision floating point for FFT and IFFT cases. The floating-point function of DSPF_sp_fftSPxSP_opt, which is an optimized version of FFT, is used for calculating FFT on D.Module.C6657.

8.2. Decimation-in-Time (DIT) and Decimation-in-Frequency (DIF)

The Fast Fourier Transform (FFT) is part of family of algorithms that efficiently calculate the Discrete Fourier Transform (DFT) of a discrete signal. The classic FFT algorithm is based on breaking up a block of N number of FFT elements into two blocks (each the size of N/2).The process repeats until the number of elements in a block are two. The FFT algorithm is at its maximum efficiency when the length of the sequence is a power of 2. In case the decimation of larger blocks is applied to input vector, the algorithm is called decimation-in-time. The decimation can be applied to result vector as well, where it is called Decimation-in-Frequency (DIR).The numbers of elements that are power of 2 are supported by all FFT function in DSPLIB in DIF algorithm. In addition, numbers of sequences that are not power of 2 are also supported by FFTLIB functions.

The structure that the decimation algorithm is made of is called butterfly. The input to the butterfly is two complex numbers. A vector of complex values, also called twiddle factors are generated, that are written on top of the original complex inputs. The butterflies called Radix-4 are more efficient algorithms that have four complex input and generate four complex output. In contrary Radix 2 have two complex inputs and two complex outputs. The DSPLIB implementation of FFT uses radix-4 butterfly for number of elements of power of 4. For FFT with number of elements not power of 4, (but power of 2) the algorithm uses Radix 4 until the last phase and Radix 2 in the last phase.

The difference between DIT and DIF is that to implement the FFT the sequence must be rearranged before performing FFT in one case or rearranged after performing FFT in the other case. The rearrangement before or after FFT operation gives us the correct order of FFT values. The difference in DIF and DIT is also clearly visible in their respective flow diagrams illustrating the FFT operation.[25]

[25]http://www.tij.co.jp/jp/lit/an/sprac64/sprac64.pdf

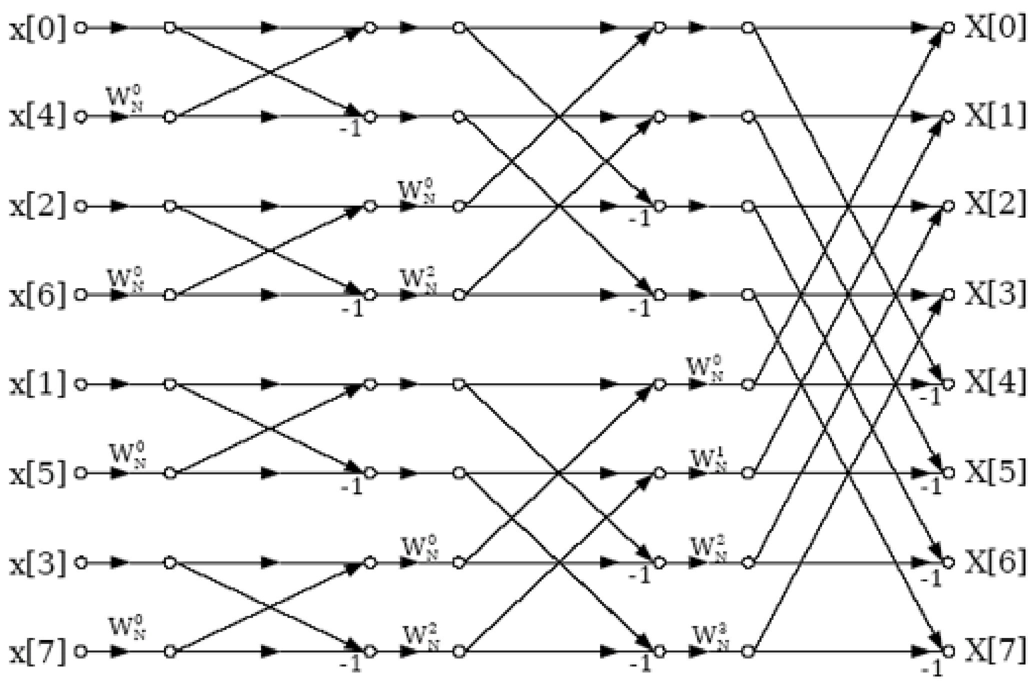


Figure 43: Schema of calculations for 8-point FFT using DIT(from reference [14])

IT can be seen, in the DIT diagram that the decimation is implemented in the time domain. That’s the reason, the time indices are in bit-reversed order. The decimation here starts with four 2-point DFTs, progress on to two 4-point DFTs and ends with a single 8-point DFT.

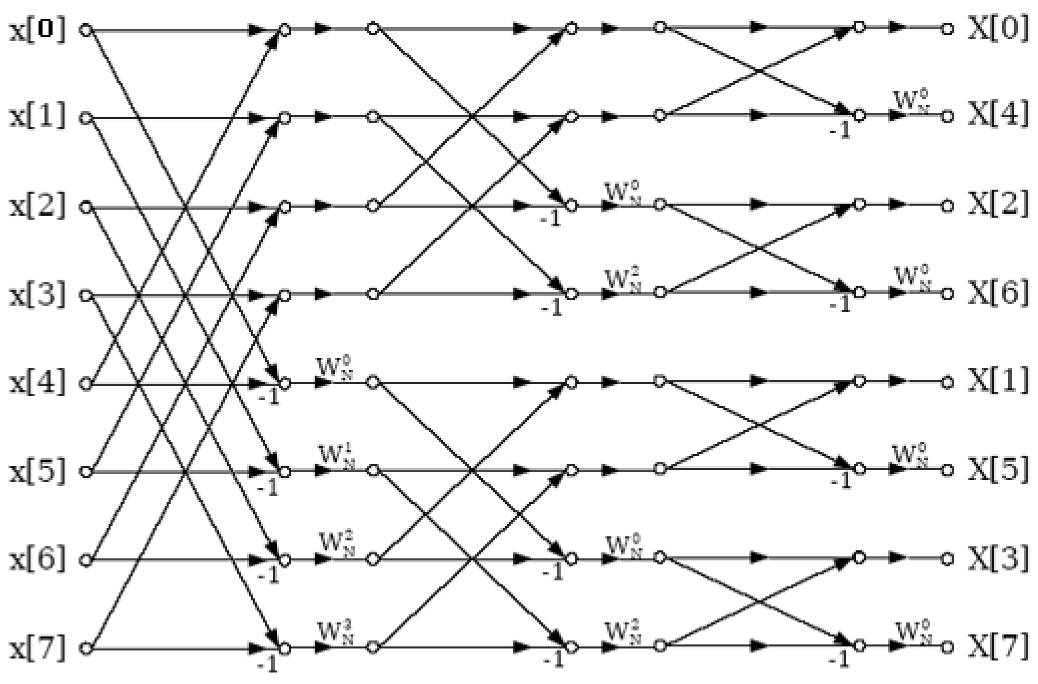


Figure 44: Schema of calculations for 8-point FFT using DIF. (from reference [14])

The bit reversed value of an index depends on the number of bits being used to represent those indices. In this case, the indices of input and output values are expressed with just 3 bits, since log2 (N) where N=8, is equal to 3. The ordered indices are 0, 1,2,3,4, 5, and using 3 bits this corresponds to 000,001,010,011,100,101 and so on. The disordered ones are 0, 4, 2, 6, 1, 5, and using 3 bits they are written as 000,100,010,110,001,101, and so on. The bit reversed values are obtained by simply flipping the bits of the ordered ones. This can be done with any to the power of 2 sized FFT.

In the DIF algorithm, the decimation is done in the frequency domain. Therefore, the frequency indices are in bit-reversed order. The FFT begins with a single 8-point DFT, progress on to two 4-point DFTs and ends with four 2-point DFTs.[26]

8.3. Bit Reversal

The function of FFT implementation in DSPLIB does not need the calculation of bit reversal. The parameter of bit reversed indices is not used by this function; however the argument is still present. The parameter may be there to provide backward compatibility with previous versions of the library or a version for another device.

[26]http://www.tij.co.jp/jp/lit/an/sprac64/sprac64.pdf

8.4. Twiddle Factors

A vector of twiddle factor is used in the implementation of FFT. The twiddle factor values contain a real part and an imaginary part, denoted by cosine and sine functions respectively. The generation of twiddle factor depends on the way the FFT algorithm uses the twiddle factors. In the first phase the N point data for the Radix 4 FFT is divided into 4 blocks of N/4 complex numbers. It requires N/4 different complex twiddle factors to combine two blocks into one block. In the second phase of Radix 4 FFT the data is divided into 16 blocks, and each block has N/16 complex numbers. It requires N/16 complex twiddle factors to combine two blocks of data. Subsequently in phase P, the data is divided into blocks of 4^P, each containing $N/4^P$complex numbers and requiring $N/4^P$complex twiddle factors to combine two blocks. This process continues until the data is divided into N number of blocks each containing one complex number.

A simple program to generate twiddle factors is given in the unit test code of any FFT function in the release. The following code snippet is an example for single precision twiddle-factor generation taken from the unit test of the function DSPF_sp_fftSPxSP(). A similar program (but not identical because of the different sign in the twiddle factor exponent) exists for IFFT.[27]

```
void tw_gen (float *w, int n)//This is for FFT, IFF will be slightly
different
{
    int i, j, k;
    const double PI = 3.141592654;
        for (j = 1, k = 0; J <= n >> 2; J = J << 2)
        {
            for (i = 0; i < n >> 2; i += j)
            {
                w[k]  = (float) sin (2 * PI * i / n);
                w[k + 1] = (float) cos (2 * PI * i / n);
                w[k + 2] = (float) sin (4 * PI * i / n);
                w[k + 3] = (float) cos (4 * PI * i / n);
                w[k + 4] = (float) sin (6 * PI * i / n);
                w[k + 5] = (float) cos (6 * PI * i / n);
                k += 6;
            }
    }
}
```

Figure 45: Function of twiddle generation for FFT from DSPLIB library(from reference [14])

[27]http://www.tij.co.jp/jp/lit/an/sprac64/sprac64.pdf

8.5. Parameters of FFT Functions

The following table lists all the parameters of FFT function and describes their role in the implementation of FFT. The parameters are of DSPF_sp_fftSPxSP_opt function which is the optimized version of FFT and is used in the DSP program for FFT and IFFT implementation.[28]

Parameter	Description
int N	Number of elements. This must be a power of 2.
float *ptr_x	A pointer to the complex input vector. The vector size is 2 × N because the real and imaginary parts are interleaves as follows: Real[0] Image[0] Real[1] Image[1] And so on.
float *ptr_w	A point to the twiddle factor array. The contains of the vector and the order of elements are as described in Section 2.3 (and in the function unit test).
float *ptr_y	A pointer to the output results. While the FFT algorithm is done in place, the last stage is written to the output vector to arrange the output in normal order and not in bit reversal order. The vector size is 2*N because the real and imaginary parts are interleaves as follows: Real[0] Image[0] Real[1] Image[1] And so on.
unsigned char *brev	The brev vector is not used in this routine. The API has backward compatibility with older functions. However, for the tools to build the code the user must supply a float pointer to some array.
int n-min	The minimum Radix size. If the number of elements is a power of 4, n_min must be 4. If it is not a power of 4, n_min must be 2.
int offset	Offset is the location of the first output in the resulted vector. It is used when multiple cores process a large FFT by dividing the FFT into smaller blocks, and then use one core to complete the last few butterflies. For a single core implementation, the offset is usually set to 0
int n_max	n_max determines how many phases will be executed. The parameter is used when multiple cores process a large FFT by dividing the FFT into smaller blocks, and then use one core to complete the last few butterflies. For a single core implementation, n_max is usually set to N.

Table 7:DSPF_sp_fftSPxSP_opt Parameters(from reference [14])

[28]http://www.tij.co.jp/jp/lit/an/sprac64/sprac64.pdf

9. Implementation of thesis task

The task description of thesis has been discussed in the introduction. In this section the implementation of tasks mentioned in the thesis description will be discussed. The implementation of each individual task, development and change in the program for that task has been explained in detail. In addition the issues faced during the implementation of subtasks are also mentioned.

9.1. Implementation of Source Localization in MATLAB

A program of source localization of a speaker is developed in MATLAB. The program in MATLAB simulates the MCC algorithm for localization of a source. The delayed signals from the source to the Microphone Arrays are delayed artificially using the time-shift property of FFT. The number of microphones in a microphone array used in MCC algorithm is selected by the user in MATLAB program. The program also allows the user to select the type of signal for testing MCC algorithm for SDOA estimation. The signal for testing the program is superposed with noise, whose factor is also selected by the user. The delay to be introduced to signals for the Microphone Arrays is also selected. The MCC algorithm is used to estimate the SDOA in the signal at Microphone Arrays. The result of SDOA values is used to estimate the position of the source.

The user can also choose to simulate SDOA estimation in the program. The performance of the multiple microphones in the MCC algorithm can be observed in the simulation of SDOA estimation. The graph of determinants of the matrices created from cross-correlation coefficients is created together for all microphones or individually for a microphone. The signal for testing, the number of microphones and delay are chosen by the user. The result of the program is the value of SDOA of signals at a Microphone Array.

The MCC algorithm uses multiple microphones to take advantage of the redundancy for SDOA estimation in adverse acoustic environment. The increase in the performance of the SDOA estimation program can be observed in the graph of determinants of the program.

The implementation of Source Localization in MATLAB is important for the understanding the DSP program in D.M.C6713. In the next step, the Source Localization program in MATLAB is adapted to the DSP program on D.M.C6657 with features such as larger sampling frequency and larger FFT buffers.

9.2. Analysis of existing DSP code

The DSP program needs to be analyzed before being ported to D.M.C6657 from D.M.C6713. The analysis and developing an understanding of the DSP program is important for porting and extension of the program. The existing source localization DSP program was developed by Marcus Müller on D.M.C6713 during his Master Thesis. Marcus Müller´s program included EDMA interrupts which were removed from the program to enable the program to run on CCSv5. Therefore the program without EDMA interrupt service routine was analyzed.

The program was developed to contain encapsulated sections of code that could be implemented on DSP board and/or C compiler. The encapsulation of code for two different platforms on the same program makes the maintenance of the program easier. The program is developed using macros to encapsulate sections of code reserved for DSP Board and for C compiler. The macro setting can be changed to switch the mode of the program from CCS to C compiler or vice-versa.

The flow of the code was followed to understand and analyze the program. The functions and files in the program were analyzed. For better understanding of a function, the result of the function was written to a file created in the program folder in C compiler. The program was able to read the file of signals whose values were displayed in the console. The functions for SDOA, Angle and Position estimation were called. However it was not obvious if any result was produced. In order to find out how far the program worked, print statements were used to keep track of the flow of the program. In the function for SDOA estimation, the results of FFT and IFFT implementation were written to the text file. The program was not able to give any result for FFT and IFFT implementation. The twiddle factor and other arrays used for FFT implementation were also empty. Therefore FFT and IFFT were implemented from the start. However it was later discovered that the reason of getting no result for FFT and IFFT was a missing function initGlobals() (seen in [A.2]) for generation of twiddle factors. After the implementation of FFT and IFFT the result of SDOA estimation function was written to the text file and was found to be correctly estimated by the program.

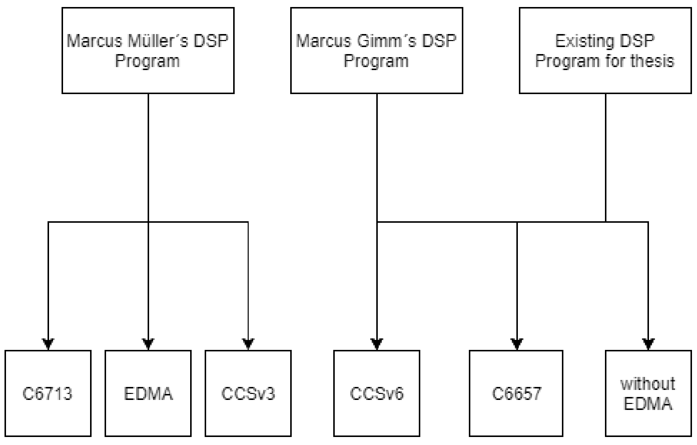

Marcus Müller´s DSP Program
Marcus Gimm´s DSP Program
Existing DSP Program for thesis
C6713
EDMA
CCSv3
CCSv6
C6657
without EDMA

Figure 46: Comparison of the three program related Source Localization programs

9.3. Porting DSP code from D.M.C6713 to D.M.C6657

The DSP program from Marcus Müller´s master thesis is ported from D.M.C6713 to D.M.C6657. The task of porting the program to D.M.C6657 involves running the program without error on the DSP hardware. The DSP program is supposed to run on CCS.v5 instead of CCS.v3, since processor C6657 is compatible for version 5 or higher of CCS. The existing DSP program on C6713 uses EDMA interrupt service routine. However the EDMA service routine is not compatible with CCSv5 and causes the program to crash. For this reason the EDMA interrupt service routine is removed from the existing DSP program. Support libraries and code for implementation of DSP interrupts are included to the program. The DSP program without EDMA interrupts is able to run without any error on D.M.C6657 in CCS.v5.

The existing DSP program on D.M.C6657 requires DSP libraries for setting up the Board and for implementing interrupts. The problem occurs when the program is unable to find the required libraries and gives an error of undefined DSP libraries. This problem can be avoided by having the program in the correct DSP project folder. The program is stored two directories down from Board Support folder of doa_esimation_C6657 in the Projects folder. Files from Board Support and Boards directories are included in the program. The program is imported to CCS from the folder named Projects. A lot of time can be saved by doing this first step of porting the program to C6657 correctly.

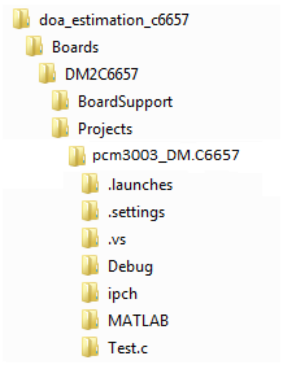

```
doa_estimation_c6657
  Boards
    DM2C6657
      BoardSupport
      Projects
        pcm3003_DM.C6657
          .launches
          .settings
          .vs
          Debug
          ipch
          MATLAB
          Test.c
```

Figure 47: Program folder including project and support libraries

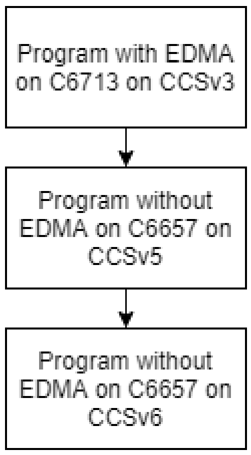

Program with EDMA on C6713 on CCSv3
Program without EDMA on C6657 on CCSv5
Program without EDMA on C6657 on CCSv6

Figure 48 Flow of program development

Modification and changes to the program

The removal of EDMA interrupts from the program meant that the adcbuffers in the program had to be filled manually; since the adcbuffers in EDMA interrupt are filled automatically by the EDMA. The program had many changes made to it to make up for the functionalities lost by removing EDMA interrupt. The following changes made to the program allow the adcbuffers to be filled in the adc interrupt and indicate when the buffers are full.

pcm3003.c

- In a program based on EDMA interrupts, the ping-pong adcbuffer array is filled automatically. However once the EDMA interrupts are removed, the adcbuffer array has to be filled manually by the program. The adcbuffers array (seen in [A.1]) is filled by even and odd channels of the DSP Board in the adc interrupt routine. In the first interrupt, idx value(seen in [A.1]) is zero and a sample value from even channels is stored in adcbuffers array. In the second interrupt, idx value (seen in [A.1]) toggles to one and a sample value from odd channels is stored in adcbuffers array.

  The value cnt (seen in [A.1]) is incremented at every second interrupt for filling the adcbuffers array. If cnt value (seen in [A.1]) is incremented at every interrupt, value of cnt will be greater than N_EDMA_LENGTH (seen in [A.1]) when only half of the buffer is full. The value of block (seen in [A.1]) represented in adcbuffers array for storing adc values gets toggled every time the adcbuffers are filled. At the same time, adcbuffers array at block 0 could be used for storing values in the interrupt, whereas adcbuffers array at block 1 could be used for FFT calculation of SDOA estimation and vice-versa.

  The bufferFlag variable (seen in [A.1]) indicates if the adcbuffers are full or not. The function for calculation of SDOA, Angle and Position of the source are called when the buffer is full, indicated by the bufferFlag being set to 1.The bufferFlag in (seen in [A.4]) is set back to default value once the SDOA function inside if-statement has been called.

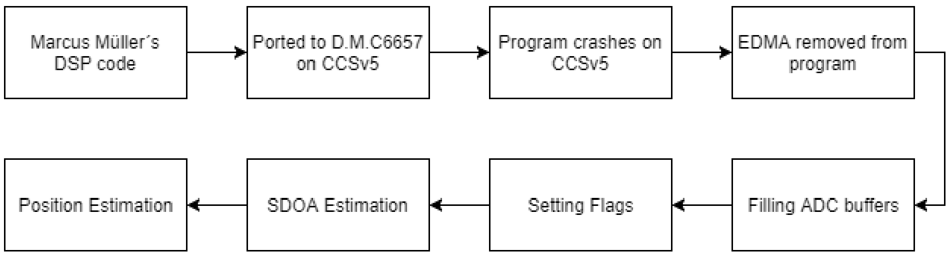


Figure 49: Process of porting program to D.M.C6657

9.4. Extending DSP program on D.Module.C6657

The extension of the DSP program after it has been ported to D.M.C6657 includes having a larger sampling frequency and larger FFT buffers. The resolution of angle and position estimation should also increase. As discussed earlier, the resolution of source localization program is dependent on the sampling frequency and distance between adjacent microphones in the microphone array. However the resolution of the program is not dependent on FFT buffer length. Increasing sampling frequency and microphone spacing increases the difference of maximum samples between signals at adjacent microphones. The drawback of increasing the spacing between adjacent microphones is that the maximum allowable frequency of the source being recorded by the microphones will decrease.

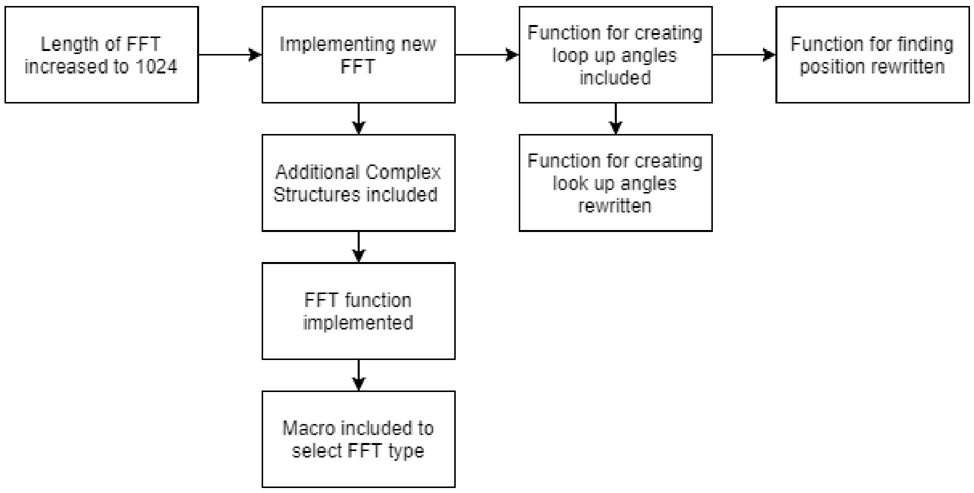

Length of FFT increased to 1024
Implementing new FFT
Function for creating loop up angles included
Function for finding position rewritten
Additional Complex Structures included
Function for creating look up angles rewritten
FFT function implemented
Macro included to select FFT type

Figure 50: Process of extending program on D.M.C6657

The DSP board has a larger memory and higher cycle speed. Therefore the length of FFT buffers in the program is increased from 256 to 1024. Increasing the FFT buffer length does not increase the resolution of angle estimation of program; it can possibly increase the performance of SDOA estimation of the program. The Source Localization program is required to run without any error or issue in real-time environment. Therefore a faster FFT, also implemented by Marcus Gimm in his program, from DSPLIB library of C66x cores is implemented on D.M.C6657.

The existing DSP program uses C callable radix-2 (DIT) FFT function included in CFFT_fromTI file. This FFT function is a C equivalent of the assembly code for FFT implementation present in cfftr2_dit.sa. The C callable FFT function takes a lot more cycles to compute and is not very efficient for SDOA estimation. The FFT present in cfftr2_dit.sa is not compatible with C6657 processor. The FFT from DSP library for C66 processor is compatible with C6657 processor. However the FFT from DSPLIB library is not compatible with CCS.v5. Hence the version of CCS being used to run the DSP program on C6657 is changed from CCSv.5 to CCSv.6. The implementation of the new FFT in the program from

DSPLIB library causes to the program code to experience changes. In addition to changes made for other function, changes due to implementation of FFT have also been discussed. These changes to the program have been correctly documented and explained, since they have the potential to affect the performance of the program.

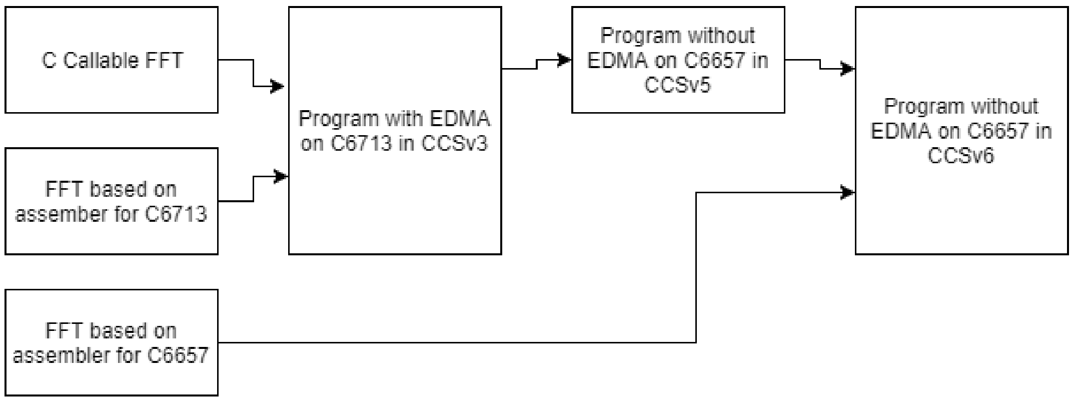

C Callable FFT
FFT based on assember for C6713
FFT based on assembler for C6657
Program with EDMA on C6713 in CCSv3
Program without EDMA on C6657 in CCSv5
Program without EDMA on C6657 in CCSv6

Figure 51: Usage and compatibility of FFTs depending on hardware and CCS version

Modification and extension to the program

defines.c

- The length of FFT and correlation signal is set in defines file (seen in [A.12]). The variable N_EDMA_LENGTH defines the length of the adcbuffer and variable N_CORR_LENGTH defines the length of the correlation. These two values need to be changed in order to change the length of the signal being used for SDOA estimation .In this thesis the length of N_EDMA_LENGTH was increased to 1024 from 256, length of variable N_CORR_LENGTH is hence doubled to 2048.

globals.c

- Complex data structures for implementing the new FFT were included in the global file (seen in [A.11]). The complex data structures were also aligned for the memory similar to other complex data structures. An example of complex data structure with aligned memory is shown:

```
st_complex IFFTSignalArray[NUM_OF_MICROPHONES][N_CORR_LENGTH];
st_complex IFFTOutArray[N_CORR_LENGTH];
#pragma DATA_ALIGN(IFFTSignalArray,sizeof(st_complex))
#pragma DATA_ALIGN(IFFTOutArray,sizeof(st_complex))
```

Code 10: Example of Complex Structures declared in defines file processor.h

- The macros included in processor file (seen in [A.13]) need to be defined to select the mode of program,select type of FFT or method of signals generation.The program can be run on CCS and C compiler. In addition, the program can choose from C callable or DSPLIB FFT. The program can also select the method of signal generation for SDOA estimation.

pcm3003.c

- The program has the option to use C callable FFT or FFT from DSPLIB of Cx66 on DSP board. The C callable FFT needs twiddle factors, that are generated by initGlobals() (seen in [A.13]) Similarly for the implementation of DSPLIB FFT, twiddle factors are generated by function tw_gen() (seen in [A.13]). The twiddle factor only needs to be generated once inside the main for FFT implementation.

CalculationOfSDOA_etc.c

- The implementation of FFT is the single most important operation in SDOA estimation. The reason for FFT implementation is calculation of Cross-Correlation of two signals. Marcus Müller´s program uses C callable function for FFT calculation. In this thesis, the FFT from DSPLIB library has also been implemented in the existing Source Localization program. The implementation of FFT from DSPLIB library has already been discussed in detail. The FFT is used twice in the program, once for converting signals in time domain to frequency domain and the later to convert them back to time domain, The function DSPF_sp_fftSPxSP_opt (seen in [A.6]) for FFT implementation takes in two pointers for reading signal for FFT conversion and for storing the output of FFT conversion. In addition the function needs the pointer of twiddle factor, the value of length of correlation and the value of radix for FFT calculation.

- The value of length of correlation is twice the length of signal buffer. The value of radix is 4 in FFT argument for signal length to the power of 4 and 2 for signal length to the power of 2(not power of 4).The argument for value of radix in the FFT function is changed from 4 to 2 for correct FFT implementation due to the change in the length of the signal.

- The C callable FFT uses a single array to read the signal and write the result of FFT conversion. On the other hand, DSPLIB FFT reads signals and writes the results of the FFT to different arrays. Therefore a different input array is used in convertCmpxCorrResultToFloatWithFftShiftUnbiased function for shifting and scaling the FFT after implementation of FFT from DSP LIB library (seen in [A.7]).

- The function CreateLookupForAngleCalculation() was added to CalculationOfAngle function in CalculationOfSDOA_etc file(seen in [A.8]). However this function is also called when the function initGlobals() for twiddle generation is called. This function added to the program creates a table of angles for the corresponding SDOA value. Therefore this function should be called only one. The function ULA_calcAngle_Lookup (seen in [A.8]) searches the table of angles for the corresponding angle for SDOA value. The angle for corresponding SDOA value cannot be determined without creating a table of angles by CreateLookupForAngleCalculation function.

functionsPositionFinding.c

- An array of angles is created by CreateLookupForAngleCalculation function in this file (seen in [A.9]). The table of angles is searched to find the angle for the corresponding SDOA value. This function fails to create a table with correct values of angle; hence this function (seen in [A.9]) is corrected. The part of the function that did not determine correct angle values has been commented out (seen in [A.9]).

- The function positonFinding() (seen in [A.10]) present in this file has also been modified to get correct position of the source. The change was made since the function did not give the correct position of the source. The part of the function that failed to give the correct position of the source has been commented out (seen in [A.10]).

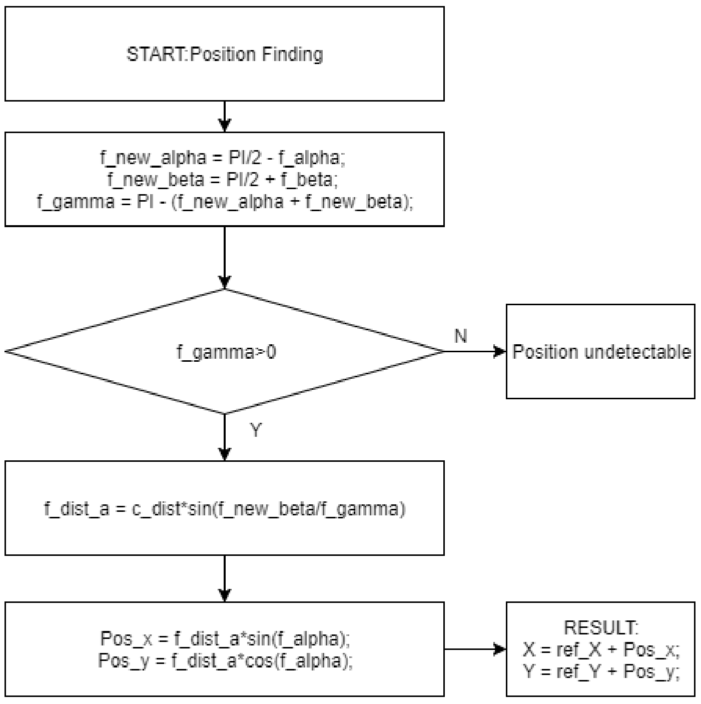

START:Position Finding
f_new_alpha = PI/2 - f_alpha;
f_new_beta = PI/2 + f_beta;
f_gamma = PI - (f_new_alpha + f_new_beta);
f_gamma>0
N
Position undetectable
Y
f_dist_a = c_dist*sin(f_new_beta/f_gamma)
Pos_x = f_dist_a*sin(f_alpha);
Pos_y = f_dist_a*cos(f_alpha);
RESULT:
X = ref_X + Pos_x;
Y = ref_Y + Pos_y;

Figure 52: Flowchart of positionFinding function for estimation of position of source

9.5. Generation of test sequences

The MATLAB program includes operation to generate test sequences for testing Source Localization program in MATLAB, in C compiler and on DSP Board. The program in MATLAB creates WAV file for testing Source Localization in MATLAB. The level of noise can be chosen for signal in WAV file. The test signals are also created internally for testing of Source Localization in MATLAB. In addition, for testing of the program in C compiler, delayed signals are written to a text file. The file is created with 8 scaled delayed signals. The file is read by the C program, where the first four channels are for left Microphone array and the remaining for right microphone array. The signals in the file are also immersed in noise to test the performance of the program. The MATLAB program also creates artificially delayed noisy signals as a header file for DSP program on CCS.

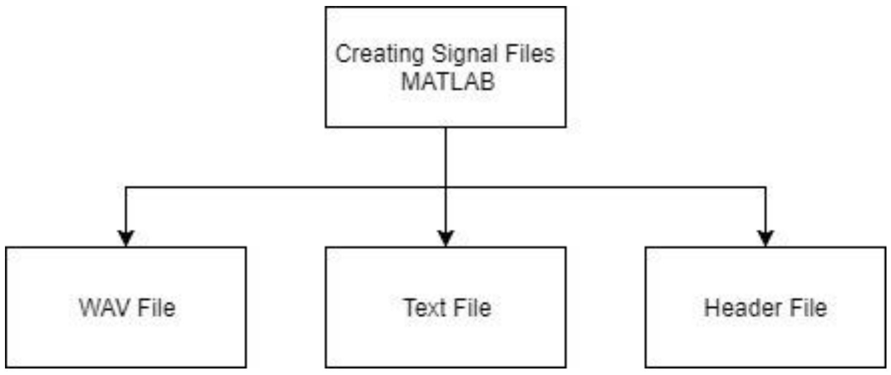


Figure 53: Process of signal generation in MATLAB

9.6. Testing of the program

The tests are carried out for Source Localization program on MATLAB, C compiler and CCS. The MATLAB simulation is tested with WAV files which are generated by the MATLAB program. The Source Localization program in C compiler is tested with file sequences created by MATLAB program and using artificially delayed signals in the program. The DSP program on CCS is tested by creating artificially delayed signals in the program. These signals are copied to the buffer arrays of the program that are used to estimate SDOA between adjacent signals. The original signal to create versions of the delayed signals is either read from the signal generator or created using a sinusoidal function of math library. The modification required for testing of the program has also been implemented in the DSP program.

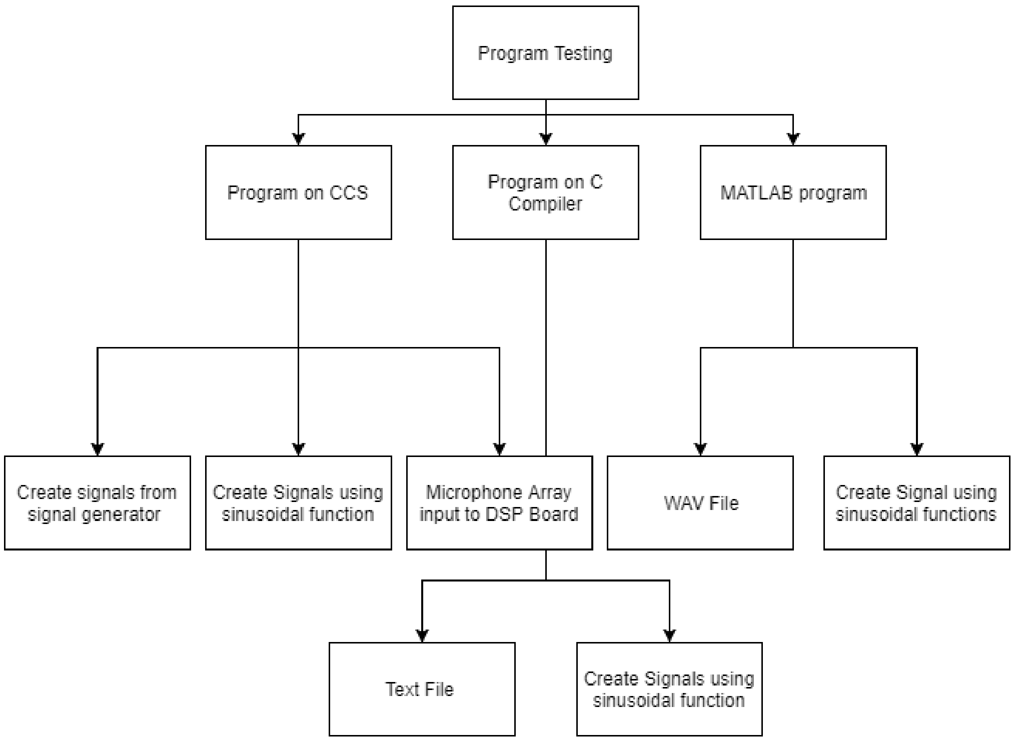

Program Testing
Program on CCS
Program on C Compiler
MATLAB program
Create signals from signal generator
Create Signals using sinusoidal function
Microphone Array input to DSP Board
WAV File
Create Signal using sinusoidal functions
Text File
Create Signals using sinusoidal function

Figure 54: Process of testing programs

Modification to the program

pcm3003.c

- The Source Localization program can be tested in reverberant environment with the use of microphone arrays. However to test the DSP program without a microphone array artificial signals are created. The signals created internally are void of noise or reverberance, hence the program is tested in free-field environment. A sinusoidal function of math library is used generate a signal that is stored in TestSignals array (seen in [A.3]). The program creates delayed versions of TestSignals array that are stored in array of signalAtMics. The signals stored in signalsAtMics represent the delayed signal for each microphone in the microphone array. This array is used by the program to test MCC algorithm for SDOA estimation.

- The program can also create delayed versions of signal stored from signal generator. The signal stored from signal generator is delayed when the adcbuffers are full. Channel zero of adcbuffer is stored in TestSignals array (seen in Code 11).The nested for-loop (seen in [A.4]) writes the delayed versions of the original signal to signalsAtMics array. The generation of artificially delayed signals using signal generator is done inside the while-loop since setting of the signal generator can be externally changed and it takes many interrupts to fill adcbuffers.

```
        //following for-loop stores adcbuffer in TestSignal array
        for(mx=0;mx<N_EDMA_LENGTH;mx++){
                TestSignals[mx] = (short) adcbuffer[0][block][mx];
                //TestSignals[mx] = (short)(32768*cos(2*PI*10.0f/(N_EDMA_LENGTH)*mx));

        }
        //following nested for-loop create signals with delays
         for(i=0;i<2;i++){
                for(k=0;k<4;k++){
                        for(mx=0;mx<N_EDMA_LENGTH-delay[i]*k;mx++){
                            signalsAtMics[k][i][mx+delay[i]*k] = TestSignals[mx];
                        }
                        for(mx=0;mx<delay[i]*k;mx++){
                            signalsAtMics[k][i][mx] =
                                      TestSignals[(N_EDMA_LENGTH - delay[i]*k)+mx];
                        }
                }
        }
```

Code 11: Creation of delayed signals in DSP program

CalculationOfSDOA_etc.c

- The program stores artificially delayed signals to signalsAtMics array for SDOA estimation. The mapping of the signalsAtMics array to pointers in implemented in the following code. The pointers (seen in [A.4]) point to signalsAtMics array, which are used in the program in the process of SDOA estimation.

```
        if (LEFT_ARRAY_M2 == g_int16_SelectedArray) {
            pp_int16_PointerOnSignals[0] = signalsAtMics[0][g_int16_SelectedArray];
            //   microphone 5 (in HW)
            pp_int16_PointerOnSignals[1] = signalsAtMics[1][g_int16_SelectedArray];
            // microphone 6 (in HW)
            pp_int16_PointerOnSignals[2] = signalsAtMics[2][g_int16_SelectedArray];
            // microphone 7 (in HW)
            pp_int16_PointerOnSignals[3] = signalsAtMics[3][g_int16_SelectedArray];
            // microphone 8 (in HW)
        }
        else if (RIGHT_ARRAY_M1 == g_int16_SelectedArray){
            pp_int16_PointerOnSignals[0] = signalsAtMics[0][g_int16_SelectedArray];
            // microphone 1 (in HW)
            pp_int16_PointerOnSignals[1] = signalsAtMics[1][g_int16_SelectedArray];
            // microphone 2 (in HW)
            pp_int16_PointerOnSignals[2] = signalsAtMics[2][g_int16_SelectedArray];
            // microphone 3 (in HW)
            pp_int16_PointerOnSignals[3] = signalsAtMics[3][g_int16_SelectedArray];
            // microphone 4 (in HW)
        }
```

Code 12: Mapping of signalsAtMics array to pointers

Issues encountered during testing

The DSP program could suddenly give an error, after the program has been tested to be free of any error. In such case, the project should be cleaned or Debug folder be deleted from the program. If the program suddenly outputs a signal that is completely different to the input signal of the program, then most probably the problem originates from the signal generator. In this situation, the signal generator needs to be turned off once before being used.

The twiddle factor and artificial signals are generated using functions from math library. These operations take a lot of cycles to computer and could cause the program to experience real-time issues if they are included in the while loop. Therefore twiddle generation and artificial signal generation should be done outside the while loop. In addition to math functions, there are parts of program which contribute to the disturbance in output of the program. The disturbance can be avoided by commenting out those parts of the program. However the SDOA calculation function causes the program output to become discontinuous. The strategies adopted to address this real-time issue of discontinuous program output will be discussed.

9.7. Displaying the position of speaker using a monitor

The result of the estimated position by the Source Localization program was not displayed on monitor since the program was not tested for real-time sequences. Another reason for not displaying the result of the position of speaker was the real-time issue with the program output. The program was tested with constant signals that were either created in the program or read from a file in the project. However the result of source localization program was viewed in the ‘Expression’ window of CCSv6. The Source Localization program tested with signals created in the program and reading the signals from a file gave correct SDOA results. The SDOA information of the two microphone arrays was used to correctly estimate the position of the source. The value of SDOA and angle are stored in g_st_MicrophonArray. The estimated position of the source is stored in g_stSourceInformation.

10. Analyzing Real-Time Issue

The real-time issue was observed while testing the DSP program on D.M.C6657. This issue is not observed in Marcus Müller´s DSP program and is not expected to occur in the existing program, due to the larger memory and higher cycle speed of the DSP hardware. The strategies adopted to solve the real-time issue of the output of the program are discussed in this section.

10.1. Strategies adopted to solve Real-Time issue

The existing program on D.M.C6657 experiences Real-Time issue until the SDOA estimation function is being called. The function for Angle and Position estimation does not cause any issues with the program output. The output of the program is a discontinuous and a distorted signal. There are a few missing blocks in the output of the program. The frequency shown on the oscilloscope of the output signal is also lower than the actual 4 kHz frequency of the input signal. The following graphs show the actual signal output of the program with different buffer lengths from the DSP Board. The actual sine signal and gap in the first graph are longer than in the second graph. The reason for a proper sine wave signal is that the dac in while-loop is being updated by adc while adcbuffers are being filled. As the adcbuffers become full, the SDOA calculation function is called and the gap in the output occurs. Therefore the signal and gap part of second graph is shorter since filling and FFT implementation of a shorter buffer happens faster. The problem appears to be insufficient cycle budget of the processor after observing the two graphs. Another possible reason could be that the memory of adc is being destroyed during SDOA estimation.

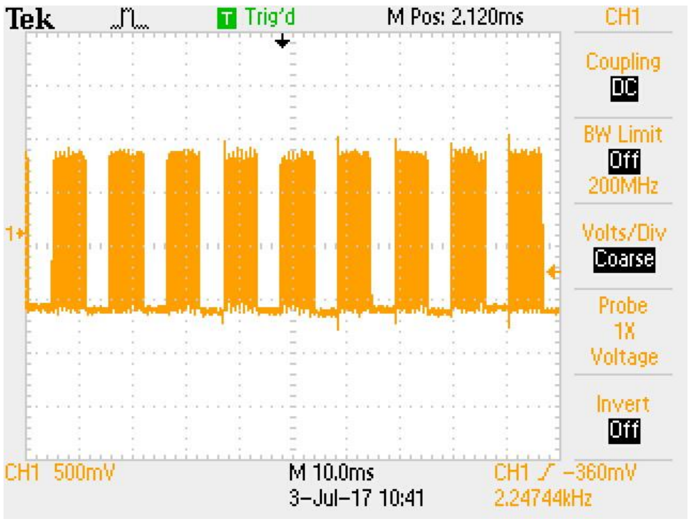

Tek
Trig'd
M Pos: 2.120ms
CH1
Coupling DC
BW Limit Off 200MHz
Volts/Div Coarse
Probe 1X Voltage
Invert Off
CH1 500mV
M 10.0ms
3-Jul-17 10:41
CH1 ⁄ −360mV
2.24744kHz

Figure 55: Screenshot of program output with 1024 buffer length without changes in optimization setting

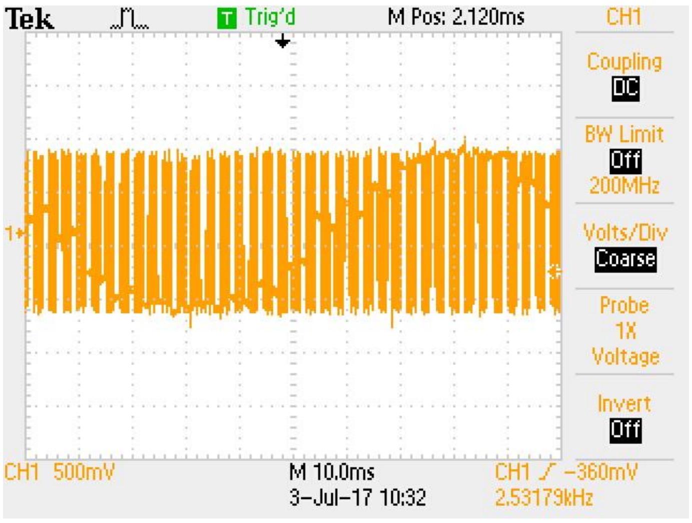

Tek Trig'd M Pos: 2.120ms CH1
Coupling DC
BW Limit Off 200MHz
Volts/Div Coarse
Probe 1X Voltage
Invert Off
CH1 500mV M 10.0ms CH1 ∕ -360mV
3-Jul-17 10:32 2.53179kHz

Figure 56: Screenshot of program output with 256 buffer length without changes in optimization setting

A mistake that was made while filling the adcbuffers was that the counter of the index of the adcbuffers was incremented at each interrupt. In the program, interrupts occur separately for even and odd channels. Even and odd channels of the DSP Board are read at different interrupts. Incrementing the counter at interrupt of both even and odd channels only allows half the buffer to be filled. Therefore the counter needs to be incremented only once, while reading even or odd channels. This has an effect on the output of the program, which is that frequency of the output signal increases. The part of the output with correct sine signal also increases. However the gap in the output is the same as earlier for both of the following graphs. The reason for improvement in the signal is due to increased time of dac updating in the while loop, since it takes longer to fill the buffers, hence the SDOA estimation function is called later. This effect also indicates that the issue the program is facing is due to insufficient cycle budget.

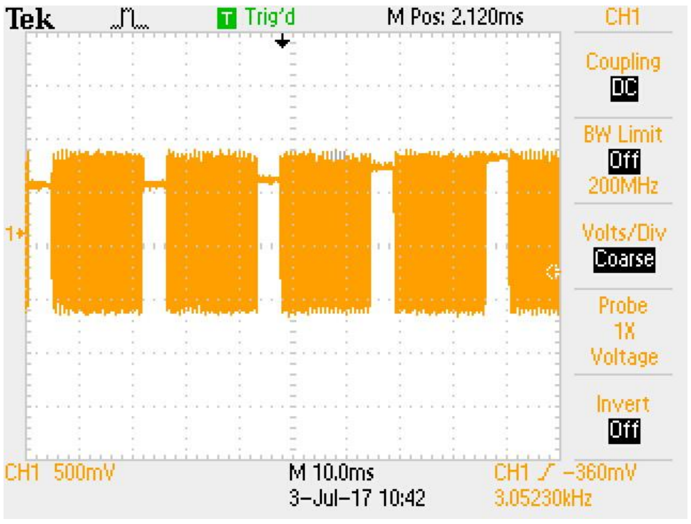

Tek Trig'd M Pos: 2.120ms CH1
Coupling DC
BW Limit Off 200MHz
Volts/Div Coarse
Probe 1X Voltage
Invert Off
CH1 500mV M 10.0ms CH1 ∕ -360mV
3-Jul-17 10:42 3.05230kHz

Figure 57: Screenshot of improved program output with 1024 buffer length without changes in optimization setting

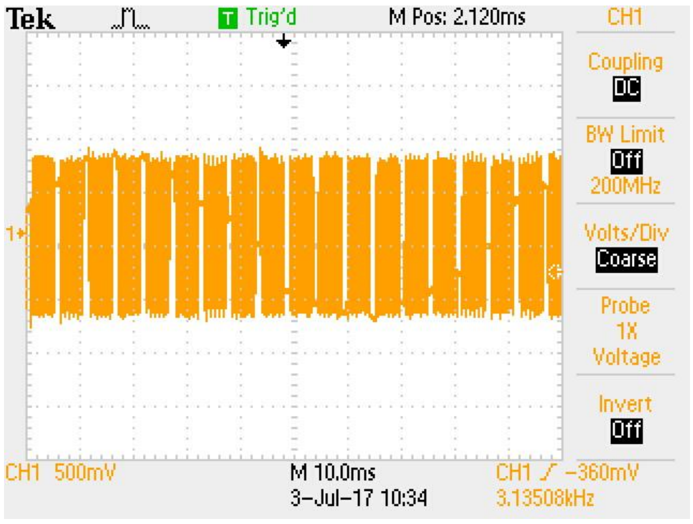


Figure 58: Screenshot of improved program output with 256 buffer length without changes in optimization setting

The complex data structures and other variables used in the program were also aligned for memory to improve their performance. A different strategy used for solving the real time issue was to change the setting of properties of the program. The files of the existing program were copied to Marcus Gimm´s program to test if change in the settings of the program solves the real time issue. The main file of the existing DSP program and the Marcus Gimm´s program were also compared line by line. The main file of the existing DSP program was made identical to Marcus Gimm´s main file in the effort to solve real-time issue. The program was tested with only relevant parts of SDOA estimation function, to help identify the reason behind the incorrect output. The program was also tested by decreasing the length of the buffer from 1024 to 256 sample values. Despite all the strategies the real-time issue observed in the discontinuous program output on the oscilloscope was not solved.

The Optimization settings in properties of the program were changed to observe the affect on the output of the program. The setting of optimize for speed option in Advanced Optimization in properties of program was changed from 0 to 1. This change shortened the discontinuity in the output, increased the length of proper sine signal and increased the frequency of the signal that can be seen in figure 59 and 60.

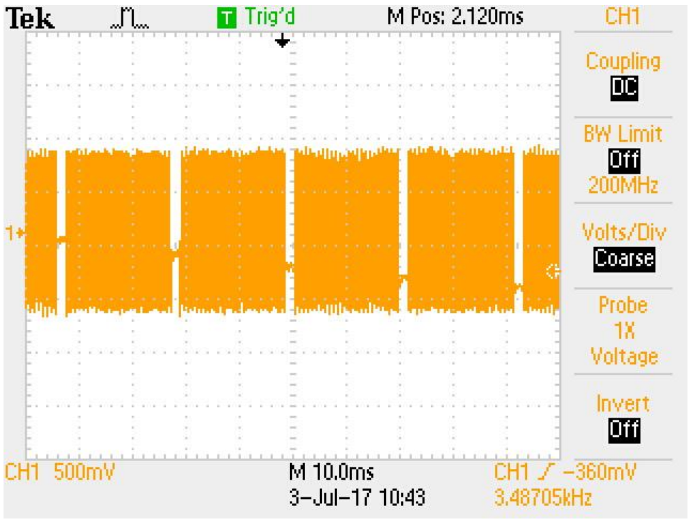


Figure 59: Screenshot of improved program output with 1024 buffer length with changes in optimization setting

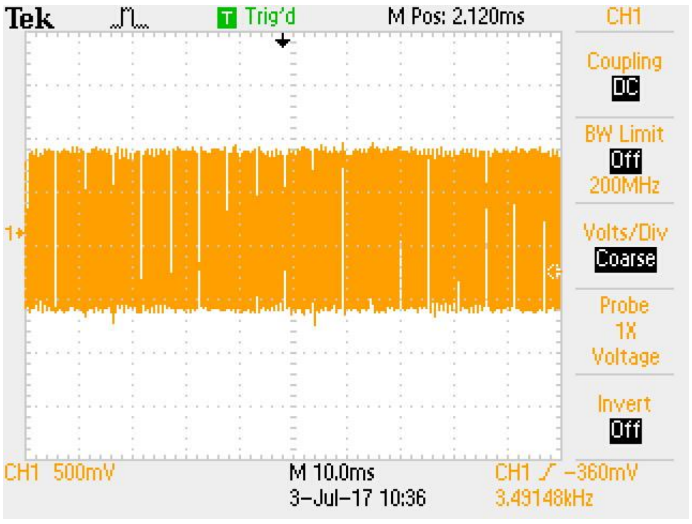


Figure 60: Screenshot of improved program output with 256 buffer length with changes in optimization setting

The main file contains for-loop that fills the tempbuffer array adcbuffers values. The main of the program also contains code used during localization of two source of sound by the program. These two sections of the program appear to be causing disturbance in the output of the program. The disturbance in the output disappears when the two parts of code of the main file are commented out. However the discontinuity in the output of the program is still the same. The two graphs with and without disturbance are shown below.

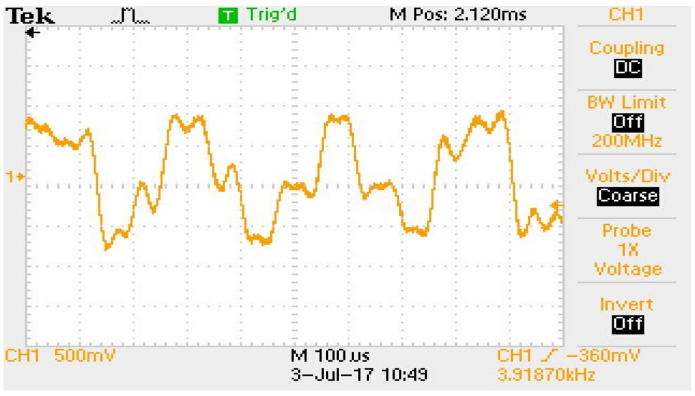


Figure 61: Disturbance in program output due to the use of tempbuffer in the main file

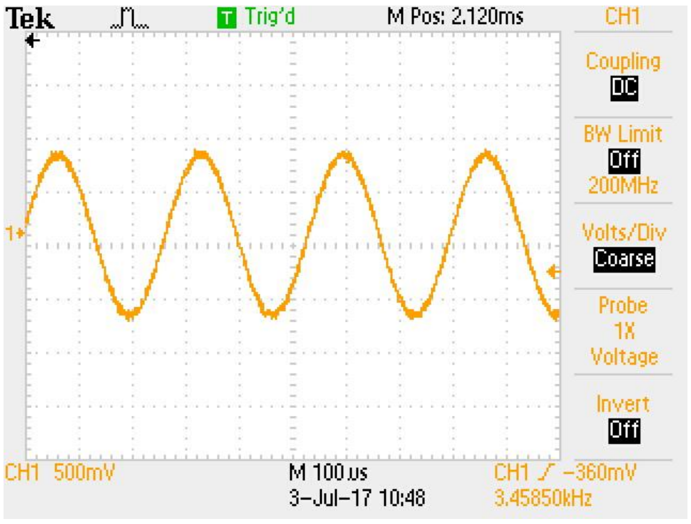


Figure 62: Program output without using tempbuffer in the main file

The program output for a low frequency input sine wave is a sine wave distorted by a rectangular signal shown in the figure 63. It is probable that the output signal is corrupted by a periodic rectangular pulse. This could explain periodic gaps in the output signal of the program for high frequency input signal. The gap might just be caused by a periodic rectangular pulse and not due to SDOA function. This would suggest that the reason for the incorrect output is not insufficient cycle budget. This assumption is strengthened when FFT function is commented out in SDOA estimation function and the same imperfect output sine wave is observed in the oscilloscope.

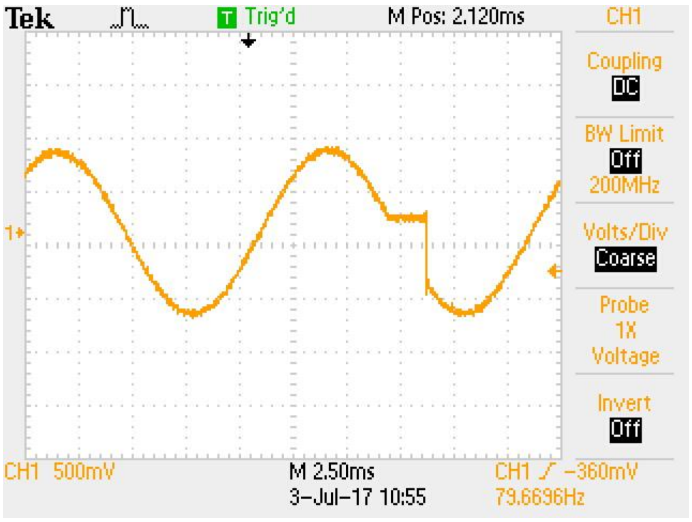


Figure 63: Screenshot of program output for low frequency input to DSP Board

After all the changes made in the software, the hardware was also changed to observe the output of the program. A different C6657 processor was used to run the program, but the real-time issue could not be solved. Finally Marcus Gimm´s DSP program was run on C6657, and the same real-time issue was observed with the program output. Since the same real-time issue was observed with Marcus Gimm´s program, any further investigation on this issue was stopped for this thesis. It should be noted that the output of the program improved considerably due to changes in the code and settings. The issue with incorrect output of the program needs to be solved irrespective of the reason behind the issue.

11. How to take the thesis further?

In the thesis, the existing DSP program is ported to D.Module.C6657 for its larger memory and higher clock speed. The better specifications of the D.M.C6657 Board are advantageous for the performance and efficiency of Source Localization program. Therefore the program on D.Module.C6657 is also extended.

11.1. Limitations of the existing program

The existing program has FFT from DSPLIB library which is faster compared to the C callable FFT used on D.Module.C6713. Despite the faster FFT, the program seems to experience real-time issues, as the output of the program is not a perfect reflection of the input signal. The resolution of angle and position estimation of the program can only be increased by increasing the length between adjacent microphones in the array. However this method limits the maximum allowable frequency present in the signal of source being recorded by the microphones. Another method to increase the resolution of angle and position estimation is by increasing the sampling frequency of the DSP Board. However the frequency of the Board cannot be further increased more than 48 kHz. The resolution of position estimation cannot be increased by increasing the size of FFT buffer. In case of larger FFT buffer, the time taken by the program for SDOA estimation is greater.

The source localization program is described to be able to estimate two source of sound in the environment. However the existing program only manages to locate one source in the field. The calculation of angle and position of the second source of sound also take place in the program, but the program is not able to find two minima in the graph of det(Ra(p)). In addition, the maximum number of microphones used in the program cannot be increased from four per microphone arrays, since there are 8 channel inputs of the DSP Board. Hence the performance of SDOA estimation of program cannot be increased.

11.2. Proposals for next Thesis

The fundamental concept of a program needs to be understood before moving forward with it. The source localization simulation in MATLAB is developed to understand MCC algorithm that is essential in the process of SDOA estimation. Therefore it is recommended to understand the existing MATLAB source localization program or develop a program from start.

The resolution of position estimation in the source localization program needs to be increased. A mechanism needs to be investigated to increase the resolution of position estimation without compromising the performance of the program. The feature for estimating two sources of sound needs to be implemented in the program. A way needs to be found to detect two sources of sound in the graph of determinants. More information can be found in Marcus Müller´s thesis about this feature.

The real-time issue of the program also needs to be fixed to get the same program output as the input of the program. Marcus Müller´s DSP program runs without any real-time issue on the C6713 processor. The echo program with EDMA interrupt was also implemented on D.M.C6657 without any problem. Therefore in the next thesis it should be considered to have EDMA interrupts in the existing DSP program, to stay as close as possible to the original Marcus Muller’s program.

11.3. Recommendations and tips

The existing program is an extension of Marcus Muller´s DSP program. However the existing DSP program has no EDMA interrupt service routine unlike Marcus Muller´s program. The existing program borrows the concept of manually filling adcbuffers from Marcus Gimm´s DSP program. In addition the FFT from DSPLIB library of C66x is also taken from Marcus

Gimm´s program. Marcus Müller´s program includes EDMA interrupts, whereas Marcus Gimm´s and the existing program are without EDMA interrupts. Hence the thesis deals with three DSP programs, which are in one way or the other related to each other. If an occurring issue needs to be solved, all three programs will have to be examined for help.

It is also important to read the datasheet of a new function before it is implemented, since it gives a clear overview of the function and saves time for implementation. A common way to analyze a program is to go with the flow of the program; however the analysis of all program files gives a much better picture of the program. This method also helps avoid overlooking important sections of the program and making mistakes during development of the program.

The real-time issue does not occur for Marcus Muller´s program. It would be better to work with both existing and Marcus Muller´s program. The modification and changes applied to the existing program should be implemented in Marcus Müller´s program with EDMA interrupt routine. The modifications implemented on Marcus Müller´s original program can help identify the reason of the real time issue in the existing program without EDMA interrupts.

However before moving to Marcus Muller´s program, a conclusion needs to be drawn about the existing program´s real-time issue. It is recommended to modify the existing program to have only three microphones in the microphone array. If the output of the program is not same as the input, then it can be concluded that the reason for real-time issue is not insufficient cycle budget. The Microphones in the array can further be decreased to 2 Microphones to strengthen the conclusion drawn earlier.

It is recommended to have a good overall understanding of the program and not to rush with programming. A lot of time is saved when most of the time is spent understanding the program instead of actually programming. Finally, small files for declaring and initializing variables should not be ignored, since many seemingly missing blocks in the program are found to be included in those files.

12. Summary and Conclusion

The Source Localization program uses MCC algorithm to take advantage of the redundancy of multiple microphones for SDOA estimation in adverse acoustic environment. The MCC algorithm uses Cross-Correlation Coefficients to create matrices, whose determinants are used to estimate the SDOA of signals of the source. The SDOA between signals at microphone arrays are used to determine the angle and the position of the source.

The MCC algorithm is implemented in Source Localization program in MATLAB and in C for SDOA estimation. This mechanism for SDOA estimation also exists in the DSP program of Marcus Müller and Marcus Gimm. The Source Localization program of Marcus Müller was implemented on D.M.C6713 that was ported D.M.C6657 for this thesis. In addition to porting the DSP program to D.M.C6657, extensions were made to the project. The DSP program was modified to get the Source Localization program give the correct position of the source. A number of steps were considered to increase the resolution of angle and position estimation. A faster FFT from DSPLIB library of C66x cores was implemented in the DSP program on D.M.C6657.

The development of the Source Localization program does not come to a halt at this stage. The resolution of the source localization needs to be increased. The program also needs to be further developed to locate two sources of sound in the field. The real-time issue of the program still remains and needs to be solved. However despite the issues, the program is able to correctly estimate the position of a single source of sound.

References

[1] Xiong, Xiaorui Ray. "How Does the Brain Locate Sound Sources?" *Knowing Neurons*. N.p., 15 Mar. 2013. Web. 10 June 2017. <http://knowingneurons.com/2013/03/15/how-does-the-brain-locate-sound-sources/>.

[2] Blauret, Jens. *Introduction to Psychoacoustics*. N.p., 2005. Web. 27 May 2017. <http://acousticslab.org/psychoacoustics/PMFiles/Module07a.htm>.

[3] "Sound Localization and the Auditory Scene." N.p., n.d. Web. 24 May 2017.

[4] "Acoustic Location." *Wikipedia*. Wikimedia Foundation, 14 July 2017. Web. 10 May 2017. <https://en.wikipedia.org/wiki/Acoustic_location>.

[5] *What Is a Microphone Array?* N.p., n.d. Web. 7 June 2017. <http://www.learningaboutelectronics.com/Articles/What-is-an-array-microphone>.

[6] Elevr. "Audio for VR Film (binaural, Ambisonic, 3d, Etc)." *Elevr*. N.p., 31 Aug. 2014. Web. 10 June 2017. <http://elevr.com/audio-for-vr-film/>.

[7] N.p., n.d. Web. 11 June 2017. <http://www.advancedmp3players.co.ukshopimagesproductsH4NPHASE.jpg>.

[8] Benesty, Jacob, J. Chen, and Yiteng Huang. "Direction-of-Arrival and Time-Difference-of-Arrival Estimation." *Microphone Array Signal Processing*. Berlin: Springer, 2010. N. pag. Print.

[9] "D.Module2.C6657 Data Sheet"n.d.Web. 7. June 2017 <http://www.gbm.de/fileadmin/produkte/Datenblatt/DM2-C6657_DS.pdf " >

[10] "16/20-BIT SINGLE-ENDED ANALOG INPUT/OUTPUT STEREO AUDIO CODECS"n.d. Web. 24 May2017< http://www.ti.com/lit/ds/symlink/pcm3002.pdf >

[11] Chandu, Manisha. "Softwares in Electronics." *CODE COMPOSER STUDIO*. N.p., 01 Jan. 1970. Web. 10 June 2017. <http://softwaresofelectronics.blogspot.de/2015/07/code-composer-studio.html>.

[12] "Code Composer Studio (CCS) Integrated Development Environment (IDE)." *CCSTUDIO Code Composer Studio (CCS) Integrated Development Environment (IDE) | TI.com*. N.p., n.d. Web. 24 May 2017. <http://www.ti.com/tool/ccstudio>.

[13] "Overview ." *GSG:CCSv4_Overview*. N.p., n.d. Web. 4 May 2017. <http://processors.wiki.ti.com/images/tmp/f1269801485-529964023.html>.

[14] *Analog, Embedded Processing, Semiconductor Company, Texas Instruments - TI.com*. N.p., Dec. 2016. Web. 4 May 2017. <http://www.ti.com/general/docs/lit/getliterature.tsp?baseLiteratureNumber=sprac64&fileType=pdf>.

[15] D.Module2.C6657 Getting Started Board Revision 1.0 Document Rev. 1.0 Nov/2014

[16] Müller, Marcus. *Sprecherlokalisierung Von Zwei Oder Mehr Sprechern Im Raum Unter Verwendung Von Mikrofonarrays:*. Thesis. HAW HAMBURG, 2012.

[17] Gimm, Marcus. *Gegenüberstellung Von Zeit- Und Unterraumbasierten Algorithmen Zur Echtzeit-Sprecherlokalisierung Mit Mikrofonarrays*. Thesis. HAW HAMBURG, 2015. N.p.: n.p., n.d. Print.

[18] Norbert Nölker und Adolf Klemenz. D.Module.PCM3003. 2011 Nov. url: http://www.dsignt.de/download/tddpcm3003.pdf -Accessed: 14.11.2016

A. Samples of source code

A.1. Filling the ADC buffers

```
  /*******************************************************************
    adc[0+idx] values are stored either in even or odd adcBuffer channels
    *******************************************************************/
    adcbuffer[0+idx][block][cnt] = adc[0+idx];
    adcbuffer[2+idx][block][cnt] = adc[0+idx];
    adcbuffer[4+idx][block][cnt] = adc[0+idx];
    adcbuffer[6+idx][block][cnt] = adc[0+idx];

   /*******************************************************************
        FULL ADCBUFFERS filled: when cnt is incremented at idx=0
   *******************************************************************/
    if(idx==0){
       cnt++;
    }
    /*******************************************************************
        FULL ADCBUFFERS: when cnt greater than N_EDMA_LENGTH
   *******************************************************************/
    if(cnt>N_EDMA_LENGTH){
        /*******************************************************************
         Fill ADCBUFFERS from start: when cnt reset to zero
        *******************************************************************/
        cnt = 0;
        /*******************************************************************
        block is toggled to make use of ping-pong buffer
        *******************************************************************/
        block ^=1;
        /*******************************************************************
        ADCBUFFERS filled & ready for SDOA Estimation, when bufferFlag=1
        *******************************************************************/
        bufferFlag = 1;
    }
```

A.2. Twiddle Factor generation

```
    /*******************************************************************
    initGlobals function generates twiddle factors for C based FFT
    *******************************************************************/
#ifdef USE_C_FFT
    initGlobals();
#endif
    /*******************************************************************
    tw_gen function generates twiddle factors for FFT from DSP library of C66 core
    *******************************************************************/
#ifdef USE_DSPLIB_FFT
    tw_gen (ptr_w, N_CORR_LENGTH);
#endif
```

A.3. Generating Artificially delayed signals

```
    /*******************************************************************
    delayed signals using cos of math library are created for SDOA calculation
    *******************************************************************/
#ifdef USE_ARTIFICIAL_SIGNALS

    for(mx=0;mx<N_EDMA_LENGTH;mx++){
        TestSignals[mx] = (short)(32768*cos(2*PI*10.0f/(N_EDMA_LENGTH)*mx));

    }

    for(i=0;i<2;i++){
        for(k=0;k<4;k++){
            for(mx=0;mx<N_EDMA_LENGTH-delay[i]*k;mx++){
                signalsAtMics[k][i][mx+delay[i]*k] = TestSignals[mx];
            }
            for(mx=0;mx<delay[i]*k;mx++){
                signalsAtMics[k][i][mx] = TestSignals[(N_EDMA_LENGTH-delay[i]*k)+mx];
            }
        }
    }
#endif
```

A.4. Generating artificially delayed signals and calling function for Source Localization

```
    /*******************************************************************
    The if statement checks if the adcbuffers have been filled.
    This statement needed only on DSP Board since signal is stored in interrupt
    *******************************************************************/
#ifdef USE_DMOD6713
        if(bufferFlag){
#endif//USE_DMOD6713

        /*******************************************************************
        Delayes signals using adcbuffer[0] for SDOA calculation are created here
        *******************************************************************/
#ifdef USE_SIGNAL_GENERATOR
        //following for-loop stores adcbuffer in TestSignal array
        for(mx=0;mx<N_EDMA_LENGTH;mx++){
                TestSignals[mx] = (short) adcbuffer[0][block][mx];
                //TestSignals[mx] = (short)(32768*cos(2*PI*10.0f/(N_EDMA_LENGTH)*mx));

        }
        //following nested for-loop create signals with delays
         for(i=0;i<2;i++){
                for(k=0;k<4;k++){
                        for(mx=0;mx<N_EDMA_LENGTH-delay[i]*k;mx++){
                            signalsAtMics[k][i][mx+delay[i]*k] = TestSignals[mx];
                        }
                        for(mx=0;mx<delay[i]*k;mx++){
                            signalsAtMics[k][i][mx] = TestSignals[(N_EDMA_LENGTH-delay[i]*k)+mx];
                        }
                }
        }

#endif
        /*******************************************************************
        the while-loop is broken in C program for artificial signals
        the SDOA results are written to the file
        the program would not stop without the following code and there is
        no need to keep estimating SDOA for the same signals
        *******************************************************************/
#ifdef USE_C_PROGRAMM
#ifdef USE_ARTIFICIAL_SIGNALS
         if(count==50){
             break;
        }
#endif//USE_ARTIFICIAL_SIGNALS
#endif//USE_C_PROGRAMM


        /*******************************************************************
            main calculation (25.06.2012, MM)
            here starts the calculation of 2 sources
        *******************************************************************/

        //-----------------------------------------------------------------
        // Calculation of SDOA at both arrays for two sources
        //-----------------------------------------------------------------

            CalculationOfSDOA();
        /*-----------------------------------------------------------------
        SDOA estimation called when adcbuffers are full
        Once the SDOA estimation function is called the bufferFlag needs to be reset to 0
        -----------------------------------------------------------------*/
#ifdef USE_DMOD6713
            bufferFlag = 0;
#endif//USE_DMOD6713

            //-----------------------------------------------------------------
            // Matching of two SDAO to a pair for localization
            //-----------------------------------------------------------------
            // SDOA Index is calculated
            matchingSDOAforSourceFinding( g_st_MicrophonArray, g_stSourceInformation);


            //-----------------------------------------------------------------
            // calculation of angles and position
            //-----------------------------------------------------------------
            CalculationOfAngle();

            CalculationOfPosition();
            /*-----------------------------------------------------------------
            count variable is only to observe that the compiler is able to reach
            this point of program when the adcbuffer gets full or not
            -----------------------------------------------------------------*/
            count++;
#ifdef USE_DMOD6713
            }
#endif//USE_DMOD6713
```

A.5. Hardware mapping of signals array

```
#ifdef USE_ARTIFICIAL_SIGNALS
        //The hardware mapping of array signalsAtMics is done here for using Artificial Signals
        // here is the correct Hardware mapping with the microfon-arrays
        if (LEFT_ARRAY_M2 == g_int16_SelectedArray) {
            pp_int16_PointerOnSignals[0] = signalsAtMics[0][g_int16_SelectedArray]; // microphone 5 (in HW)
            pp_int16_PointerOnSignals[1] = signalsAtMics[1][g_int16_SelectedArray]; // microphone 6 (in HW)
            pp_int16_PointerOnSignals[2] = signalsAtMics[2][g_int16_SelectedArray]; // microphone 7 (in HW)
            pp_int16_PointerOnSignals[3] = signalsAtMics[3][g_int16_SelectedArray]; // microphone 8 (in HW)
        }
        else if (RIGHT_ARRAY_M1 == g_int16_SelectedArray){
            pp_int16_PointerOnSignals[0] = signalsAtMics[0][g_int16_SelectedArray]; // microphone 1 (in HW)
            pp_int16_PointerOnSignals[1] = signalsAtMics[1][g_int16_SelectedArray]; // microphone 2 (in HW)
            pp_int16_PointerOnSignals[2] = signalsAtMics[2][g_int16_SelectedArray]; // microphone 3 (in HW)
            pp_int16_PointerOnSignals[3] = signalsAtMics[3][g_int16_SelectedArray]; // microphone 4 (in HW)
        }
        else {
            printf("ERROR"); // muss drinne bleiben,
sonstD:\\Daten\\TI_DSP_Software\\D_Mod_C6713_D_Mod_8xPCM\\20120813_1234_LAEUFT_CProgramm\\dmod-c6713_dmod-pcm_test_EDMA.cmd,
line 114: warning: C:/CCStudio_v3.3/C6000/cgtools/lib/rts6700.lib(.const) not found
        }
#endif
```

A.6. Implementation of FFTs

```
//Here there are no NUM_OF_MICARRAYS
            //Here we do calculation for each signal at MICROPHONE
            for (g_int16_cnt_ii=0; g_int16_cnt_ii<NUM_OF_MICROPHONES; g_int16_cnt_ii++) {

                    // calculate variances of signals
                    calcVariance( g_cmpx_SigBuf[g_int16_cnt_ii],
                                 N_CORR_LENGTH,
                                 N_MCCC_BUFLENGTH,
                                 &g_f_Variances[g_int16_cnt_ii]);
                    copy2ComplexStruct( &pp_int16_PointerOnSignals[g_int16_cnt_ii][g_int16_tmp], // startadress
                                                       N_MCCC_BUFLENGTH,                         // length of section
                                                       g_cmpx_SigBuf[g_int16_cnt_ii],            // complex buf
                                                       N_CORR_LENGTH);                           // length of complex
buf


#ifdef USE_C_FFT
                // do the fft without bit-reversal (result is scrambled)
                //-------------------------------------------------------------------------
                // der eigentliche Kern, FFT und bit-reversal (if necessary) !
                cfftr2_dit(g_cmpx_SigBuf[g_int16_cnt_ii], W, N_CORR_LENGTH ) ;          //TI floating-pt complex FFT
                //bitrev(g_cmpx_SigBuf[g_int16_cnt_ii], iData, N_CORR_LENGTH );         //freq scrambled->bit-reverse x
                //-------------------------------------------------------------------------
#endif

#ifdef USE_DMOD6713
#ifdef USE_DSPLIB_FFT
                //the following pointers are used for FFT implementation in DSPF_sp_fftSPxSP_opt()
                ptr_SignalArray[g_int16_cnt_ii] = (float *)g_cmpx_SigBuf[g_int16_cnt_ii];
                ptr_OutArray[g_int16_cnt_ii] = (float *)OutArray[g_int16_cnt_ii];
                //3rd last value is 2 for samples which are power of 2 and 4 for samples power of 4
                DSPF_sp_fftSPxSP_opt(N_CORR_LENGTH, &ptr_SignalArray[g_int16_cnt_ii][0], &ptr_w[0],
ptr_OutArray[g_int16_cnt_ii] , 0, 2, 0, N_CORR_LENGTH);

#endif//USE_DSPLIB_FFT
#endif//USE_DMOD6713
```

A.7. Function for shifting correlation signal

```
#ifdef USE_DMOD6713
#ifdef USE_DSPLIB_FFT
                    convertCmpxCorrResultToFloatWithFftShiftUnbiased(
                             IFFTOutArray,
                             N_CORR_LENGTH, // shift by int16_bufLength/2
                             NORM_FOR_IFFT_INVFFTLEN,
                             g_f_biasedFactors,
                             N_CORR_LENGTH,
                             g_f_CorrsYiYj[g_int16_cnt_ll] );
#endif//USE_DSPLIB_FFT
#endif//USE_DMOD6713
```

A.8. Creation of angle lookup table and angle calculation

```
//Modified by Muhammad Afaque Khan on 15.03.2017
#ifdef ANGLE_CALCULATION_WITH_LOOKUP
//Create loopup for the next method
            CreateLookupForAngleCalculation(g_f_Buffer_LookUpTable_MappingSDOA2Angle,
/* const short int16_lengthOfLookup*/              MCCC_LENGHT_STEERBUF,
/* const short int16_Steering_PMin,     */         MCCC_STEERING_P_MIN,
/* const float f_distanceMic,        */            DISTANCE_MICROFONE/3,
/*   const float f_acousticVelocity*/              ACOUSTIC_VELOCITY,
/*   const float f_samplingFrequency)*/            SAMFREQ);


            g_stSourceInformation[g_int16_cnt_ii].f_angle[g_int16_cnt_kk]=
                ULA_calcAngle_Lookup(// modified on 04.05.2017
                                        g_st_MicrophonArray[g_int16_cnt_kk].st_estSDOA.int16_SDOA_Index[g_int16_cnt_ii],
                                        g_st_MicrophonArray[g_int16_cnt_kk].f_angleOfAttack,
                                        g_f_Buffer_LookUpTable_MappingSDOA2Angle,
                                        MCCC_LENGHT_STEERBUF);
```

A.9. Function CreateLookupForAngleCalculation

```
void CreateLookupForAngleCalculation(float*pf_lookUpTable,
constshort int16_lengthOfLookup,
constshort int16_Steering_PMin,
constfloat f_distanceMic,
constfloat f_acousticVelocity,
constfloat f_samplingFrequency)
{
short int16_cnt_ii;// couter
float f_tmp_TDOA;// temporary helper

#ifdef USE_C_PROGRAMM
    assert((NULL!= pf_lookUpTable));// check pointer
#endif //USE_C_PROGRAMM


// Create Lookup Table
for(int16_cnt_ii =0; int16_cnt_ii<int16_lengthOfLookup; int16_cnt_ii++){

//calculate TDOA
//calculate TDOA
//f_tmp_TDOA = ((float) (int16_cnt_ii+int16_Steering_PMin)) / f_samplingFrequency;
//pf_lookUpTable[int16_cnt_ii] =  (float) acos( f_tmp_TDOA * f_acousticVelocity / f_distanceMic );

//Modified by Muhammad Afaque Khan on 17.03.2017
        f_tmp_TDOA =((float)((int16_cnt_ii+int16_Steering_PMin)*(f_acousticVelocity/f_samplingFrequency)))/f_distanceMic;
        pf_lookUpTable[int16_cnt_ii]=(float) asin(f_tmp_TDOA);

}
```

A.10. Function for finding position of source

```
void  positionFinding(  const float f_alpha,
                      const float f_betha,
                      const float f_dist_c,
                      const st_Coords *p_stCoord_ReferencePoint,
                      st_Coords *p_stCoord_Source)
{
    float f_gamma;
    float f_dist_b,f_dist_a;
    float f_new_alpha,f_new_betha;

#ifdef USE_C_PROGRAMM
    assert( (NULL != p_stCoord_Source) && (NULL != p_stCoord_ReferencePoint )); // check pointer
#endif //USE_C_PROGRAMM

    /*
    //  calc gamme
    f_gamma = PI - f_alpha - f_betha;

    // calc distance from reference to source
    f_dist_b = (float) ( f_dist_c * ( sin(f_betha)/sin(f_gamma) ));

    // calc coords
    p_stCoord_Source->f_coords[X_COORD] = (float) (p_stCoord_ReferencePoint->f_coords[X_COORD] + f_dist_b * cos(f_alpha) );
    p_stCoord_Source->f_coords[Y_COORD] = (float) (p_stCoord_ReferencePoint->f_coords[Y_COORD] + f_dist_b * sin(f_alpha) );
    */

    //  calc gamme

    //modifications by MAK on 25.03.2017
    f_new_alpha = PI/2 - f_alpha;
    f_new_betha = PI/2 + f_betha;

    f_gamma = PI - (f_new_alpha + f_new_betha);
    gamma = f_gamma;
    alpha = f_new_alpha;
    betha = f_new_betha;

    if(f_gamma>0){

        // calc distance from reference to source
        f_dist_b = (float) ( f_dist_c * ( sin(f_new_betha)/sin(f_gamma) ));

        f_dist_a = (float) ( f_dist_c * ( sin(f_new_alpha)/sin(f_gamma) ));

        dist = f_dist_b;

        X = f_dist_b * sin(f_alpha);
        Y = f_dist_b * cos(f_alpha);

        // calc coords
        p_stCoord_Source->f_coords[X_COORD] = (float) (p_stCoord_ReferencePoint->f_coords[X_COORD] + X);
        p_stCoord_Source->f_coords[Y_COORD] = (float) (p_stCoord_ReferencePoint->f_coords[Y_COORD] + Y);



        #ifdef USE_C_PROGRAMM
            //printf("Distance a is           :%f\n",f_dist_a);
            //printf("Distance b is           :%f\n",f_dist_b);
        #endif //end of USE_C_PROGRAMM
        }
    else{
        #ifdef USE_C_PROGRAMM
            //printf("Source out of reach ja\n");
        #endif //end of USE_C_PROGRAMM
    }

}
```

A.11. Memory aligned complex structures

```
#ifdef USE_DMOD6713
    st_complex g_cmpx_SigBuf[NUM_OF_MICROPHONES][N_CORR_LENGTH];    // complex struct for fft
    #pragma DATA_ALIGN(g_cmpx_SigBuf,sizeof(st_complex))            // align on boundary
    st_complex g_cmpx_tmpResult[N_CORR_LENGTH];                     // temp buffer for complex correlation result
    #pragma DATA_ALIGN(g_cmpx_tmpResult,sizeof(st_complex))         // align on boundary

    //----For new FFT implementation----(24.04.2017)

    st_complex OutArray[NUM_OF_MICROPHONES][N_CORR_LENGTH];
    #pragma DATA_ALIGN(OutArray,sizeof(st_complex))

    st_complex IFFTSignalArray[NUM_OF_MICROPHONES][N_CORR_LENGTH];
    st_complex IFFTOutArray[N_CORR_LENGTH];
    #pragma DATA_ALIGN(IFFTSignalArray,sizeof(st_complex))
    #pragma DATA_ALIGN(IFFTOutArray,sizeof(st_complex))

    float * ptr_SignalArray[NUM_OF_MICROPHONES];
    float * ptr_OutArray[NUM_OF_MICROPHONES];

    float * ptr_IFFTSignalArray[NUM_OF_MICROPHONES];
    float * ptr_IFFTOutArray[NUM_OF_MICROPHONES];


#endif // USE_DMOD6713
```

A.12. Data for Source Localization

```
     #define SAMFREQ                (48000)
     #define DISTANCE_MICROFONE     (0.15)//(0.17)
     #define ACOUSTIC_VELOCITY      (340.0) // in (m/s)

     #define X_COORD_REFERENCE      (1.0) //Positon of MicArray M2 with mics 5678
     #define Y_COORD_REFERENCE      (0.0)
     #define DISTANCE_OF_ARRAYS     (1.0)
     #define ANGLE_OF_ATTACK_DEGREE (0) // in degree (will be converted to rad: const float f_angleOfAttack = (36.0/360.0 *
     (2.0*PI));

     #define NUM_OF_MICARRAYS    ( 2)
     #define LEFT_ARRAY_M2       ( 0)
     #define RIGHT_ARRAY_M1      ( 1)

     #define NUM_OF_MICROPHONES  ( 4)
     #define NUM_OF_CORRELATIONS ( 6) // 4*(4-1)/2 cause symmetric matrix

     //#define N_EDMA_LENGTH     ( 256+2*3*12)  // attention when #ifdef USE_DMOD6713_VALIDATION
     #define N_EDMA_LENGTH       (4*256)            // attention when #ifdef USE_DMOD6713_VALIDATION
     #define N_MCCC_BUFLENGTH    (4*256)
     #define N_CORR_LENGTH       (4*512) // two times buffersize! better no calculation (2*B)

     // choosing correlation method
     #define GCC_UNBIASED
     //#define GCC_BIASED
```

A.13. Setting up macros in processor.h

```
#ifndef _PROCESSOR_H_
#define _PROCESSOR_H_

// choose program-version DSP-Board or C-Program
#define USE_DMOD6713
//#define USE_C_PROGRAMM
//#define USE_SIGNAL_FILE

// choose the version for FFT implementation
//#define  USE_C_FFT
#define USE_DSPLIB_FFT

// choose the delayed signal generation method for SDOA estiamtion
#define USE_ARTIFICIAL_SIGNALS
//#define USE_SIGNAL_GENERATOR

#endif // _PROCESSOR_H_
```

A.14. Function for shifting correlation

```
void convertCmpxCorrResultToFloatWithFftShiftUnbiased(
                                                      const st_complex* pComplex_sourcebuf,
                                                      const short int16_bufLength, // shift by int16_bufLength/2
                                                      const float f_scalingFactor,
                                                      const float *pf_biasedFactors,
                                                      const short int16_LengthbiasedFactors,
                                                      float *pf_destBuf)
{

    // example pf_biasedFactors when bufLength 8 (FFTlength 8)
    // biasedFactors = {1/4, 1/3, 1/2, 1/1, 0, 1/1, 1/2, 1/3}

    short int16_cnt_ii = 0;
    const short int16_shift  = int16_bufLength>>1;

#ifdef USE_C_PROGRAMM
    assert( ( NULL != pComplex_sourcebuf ) && ( NULL != pf_destBuf )
           && (NULL != pf_biasedFactors) && (int16_bufLength == int16_LengthbiasedFactors) );
#endif //USE_C_PROGRAMM

    // the array of pf_biasedFactors was commented out because the multitply the pComplex_sourcebuf with
    // pf_biasedFactors gives small signal values that do not give correct correlation results


    for (int16_cnt_ii=0; int16_cnt_ii< int16_shift; int16_cnt_ii++) {
        // first half: position = ii + shift
        pf_destBuf[int16_cnt_ii+int16_shift]=
            pComplex_sourcebuf[int16_cnt_ii].re * f_scalingFactor;// *pf_biasedFactors[int16_cnt_ii] ;

    }

    for (int16_cnt_ii = int16_shift; int16_cnt_ii< int16_bufLength; int16_cnt_ii++) {
        // second: position = ii - shift
        pf_destBuf[int16_cnt_ii-int16_shift]=
            pComplex_sourcebuf[int16_cnt_ii].re * f_scalingFactor;// *pf_biasedFactors[int16_cnt_ii];
    }

}
//end of functionsCalculateSDOA.c
```

B. Contents of the CD

The contents of the CD are

\00_pdf\ The Bachelor thesis report in pdf format is included here
\01_Matlab\ The MATLAB program is included here
\02_DSPProgramm\ The folder includes two versions of DSP source localization program, one with 1024 buffer length and the other with 256, for implementation on C compiler and CCS.

Declaration

I/we declare within the meaning of part 16(5) of the General Examination and Study Regulations for Bachelor and Master Study Degree Programmes at the Faculty of Engineer-ing and Computer Science and the Examination and Study Regulations of the International Degree Course Information Engineering that: this Bachelor Thesis has been completed by myself/ourselves independently without outside help and only the defined sources and study aids were used. Sections that reflect the thoughts or works of others are made known through the definition of sources.

City, Date and signature(s)