

UNIVERSITY OF PUERTO RICO
Mayagüez Campus

DFT Beamforming Techniques for Bioacoustics Signal Processing Applications

by

Abigail Fuentes

A thesis submitted in partial fulfillment for the degree of

Master of Science

in

Computing Engineering

Electrical and Computer Engineering Department

April 8, 2009

Approved by:

Néstor Rodríguez , Ph.D.
Member, Graduate Committee

Date

Nayda Santiago, Ph.D.
Member, Graduate Committee

Date

Domingo Rodríguez, Ph.D.
President, Graduate Committee

Date

Julio Quintana, Ph.D.
Graduate Representative, Graduate Committee

Date

Isidoro Couvertier, Ph.D.
Chairperson of the Department

Date

Declaration of Authorship

I, Abigail Fuentes, declare that this thesis titled, 'DFT Beamforming Techniques for Bioacoustics Signal Processing Applications (ESM)' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

“Faith is taking the first step, even when you don’t see the whole staircase. ”

Martin Luther King, Jr.

UNIVERSITY OF PUERTO RICO

Abstract

Electrical and Computer Engineering Department

Master of Science

by Abigail Fuentes

This work presents a computational framework for the analysis, design, and development of discrete Fourier transform (DFT) beamforming techniques for bioacoustics signal processing applications. DFT beamforming techniques are a form of hierarchical beamforming algorithm methods which deal with the processing of signals arriving at large aperture array systems. The processing of the sensed signals is conducted hierarchically in the frequency domain. The DFT beamforming algorithms consist of determining the direction of arrival (DOA) of plane waves impinging on a linear sensor array. One of the applications of the sensors is as signal instrumentation resources for monitoring and surveillance of neo-tropical anurans in the island of Puerto Rico, in a near real time manner. A plane wave complex signal model is being utilized to model the incoming signals which are spatially sampled by the array elements and temporally sampled by the sensor signal acquisition system. The DFT beamforming algorithms are being formulated using Kronecker products algebra to provide general expressions for the beamforming operations. The development effort is being conducted using the MATLAB numeric computation and software visualization package. A parallel programming modeling environment, named pMATLAB, is being utilized to study the computational performance of parallel implementation techniques. Preliminary implementation efforts have being conducted using C language on the TMS320C6713 Digital Signal Processing (DSP) unit from Texas Instruments.

A wireless sensor array processing (SAP) system has also been designed at the Automated Information Processing (AIP) Laboratory, at the University of Puerto Rico, Mayaguez Campus, with the purpose of providing further testing of the beamforming techniques developed. Such SAP testbed uses Linux-based embedded small computers, called Gumstix, as sensor signal processing nodes (SSP), with the capacity to acquire, store, and process acoustic data. A principal node, called the Master Sensor Node (MSN) receives the processed data from the SSP nodes, making the data available via wireless connection to the Internet. This type of network system will serve as an ideal tool for biologists to monitor, locate, and track species of interests in the surrounding environments, without interfering with the ecological system, and avoiding frequent field visits.

UNIVERSIDAD DE PUERTO RICO

Resumen

Departamento de Ingeniería Eléctrica y Computadoras

Maestría en Ciencias

por Abigail Fuentes

Técnicas de DFT *beamforming* o formación de haces, constituyen una forma jerárquica de los métodos de los algoritmos de beamforming, los cuales trabajan con el procesamiento de señales recibidas por sistemas de arreglo lineal de sensores, a grandes aberturas. Una de las aplicaciones de los sensores es ser usados como recursos para la instrumentación de señales con el propósito de monitorear y vigilar anuros neo-trópicos en la isla de Puerto Rico, en tiempo casi real. Un plano de onda compleja de señales ha sido utilizado para modelar las señales incidentes a un arreglo lineal de sensores, las cuales han sido muestradas espacialmente por los elementos del arreglo, y temporalmente muestradas por el sistema de adquisición de señales acústicas. Los algoritmos de DFT *beamforming* han sido formulados usando álgebra de los productos Kronecker, con el propósito de proveer expresiones generales para las operaciones de *beamforming*. El esfuerzo de desarrollo ha sido realizado usando un *software* de visualización y computación numérica, llamado MATLAB. Un ambiente de modelo en paralelo, llamado pMATLAB, ha sido utilizado para estudiar el rendimiento computacional de técnicas de implementación en paralelo. Trabajos preliminares de implementación han sido realizado, usando lenguaje C en la tarjeta de procesamiento de señales digitales (DSP) TMS320C6713 de Texas Instruments.

Un sistema de procesamiento de arreglo de sensores (SAP), inalámbrico ha sido diseñado en el laboratorio de Procesamiento de Información Automatizada (AIP), en la Universidad de Puerto Rico, Recinto de Mayaguez, con el propósito de proveer más pruebas de las técnicas desarrolladas de *beamforming*. Pequeñas computadoras, llamadas *Gumstix*, basadas en el sistema operativo de Linux, son utilizadas como nodos de sensores para el procesamiento de señales (SSP), con la capacidad de adquirir, almacenar, y procesar data acústica. Un nodo principal, llamado el *Master Sensor Node*(MSN) recibe data procesada de los nodos SSP, haciendo que la misma esté disponible mediante conexión inalámbrica al Internet. Este tipo de sistema de red servirá como una herramienta ideal para biólogos, con el propósito de monitorear, localizar, y registrar las especies de interés, en el ambiente cercano, sin tener que interferir con el sistema ecológico, y así también evitar visitas frecuentes al campo.

Acknowledgements

This work was fully supported by the NSF CISE-CNS Grant No. 0424546 under the WALSAIP project. I want to express my sincere gratitude to my mentor, Prof. Domingo Rodriguez, for all his infinite guidance and support.

I want to thanks César A. Aceros for his consistent help in revising the development of this work, and all of the team members of the WALSAIP for their endless effort in the development of the WALSAIP SAP testbed

I fully thank God, my Lord and King, for giving me the strength and health to accomplish this stage of my life.

Contents

Declaration of Authorship	ii
Abstract	iv
Resumen	v
Acknowledgements	vi
List of Figures	x
List of Tables	xiii
Abbreviations	xiv
Symbols	xv
1 Introduction	1
1.1 Problem Formulation	2
1.2 Proposed Solution	3
1.3 Thesis Research Contributions	6
1.4 Thesis Organization	6
2 Background and Related Work	7
2.1 Bioacoustics Signal Analysis	7
2.2 Acoustics Signal Analysis	7
2.3 Signal and Sensor Array Model	7
2.4 Sophisticated Algorithms	9
2.5 Structural Beamforming Mapping and Architecture	10
3 Theoretical Framework	12
3.1 Signal Characterization	12
3.2 Basic Beamforming Operation	17
3.3 DFT Multi-Beamforming Technique	19

4	DFT Kronecker Algorithms	21
4.1	Introduction to Kronecker Products and Fast Fourier Transforms	21
4.2	The Fast Fourier Transform	22
4.2.1	Example Formulation of DFT operation, for length L	24
4.3	Kronecker Product Algebra Definition	25
4.3.1	Special Properties of Kronecker Products	26
4.4	Using Kronecker Product for Parallel Operation	27
4.5	Using Kronecker Product for Vector Processing	30
4.6	Formulating FFTs Using Kronecker Product	33
4.7	Parallel Computational Architectures for DFT Multi-beamforming	36
5	Parallel Programming with pMATLAB	39
5.1	Parallel Programming Concept	39
5.1.1	Characteristics of a Parallel System Architecture	39
5.2	Parallel Speedup Process and Amdahl's Law	40
5.2.1	Amdahl's Law	42
5.3	Description	43
5.3.1	Data Mapping and Distribution	44
5.3.2	Parallel Execution in pMATLAB	47
6	Beamforming Techniques Implemented on the DSP C6713	50
6.1	TMS3020C6713 DSP Kit	50
6.2	TMS3020C6713 DSP Basic Characteristics	51
6.3	DFT and Kronecker Beamforming Algorithm Implementation Procedures on DSP C6713	51
6.3.1	Initialization and Compilation Procedures	52
6.3.2	Simulation Procedure	57
6.3.3	Program Execution Procedure on Target Architecture	68
6.3.4	Obtaining Beamforming Results using Matlab	70
6.4	A DSP Multicore Environment Architecture	71
7	Experimental Results	73
7.1	DFT and Kronecker Beamforming Algorithm Design	73
7.2	DFT Beamforming Implementation Procedure	76
7.2.1	Signal Analysis and Metrics	77
7.2.2	DFT and Kronecker Beamforming Implementation Results in MATLAB	81
7.2.3	DFT and Kronecker Beamforming Implementation Results on DSP 6713	87
7.2.4	Kronecker DFT Beamforming Implementation Results on Gumstix Verdex	88
7.2.5	DFT and Kronecker Beamforming Implementation Results in pMATLAB	90
7.2.6	DFT Beamforming Platforms Comparison	92
8	Conclusions and Future Work	94
A	MATLAB DFT Beamforming Code	96
B	MATLAB DFT Kronecker Beamforming Code	98

C	DSP DFT Beamforming Code	101
D	DSP Kronecker DFT Beamforming Code	107
E	Gumstix Verdex DFT Beamforming Code	114
F	Gumstix Verdex DFT Kronecker Beamforming Code	122
G	pMATLAB DFT Beamforming Code	132
H	pMATLAB DFT Kronecker Beamforming Code	135
I	Signal Analysis and Metrics in MATLAB	138
	Bibliography	142

List of Figures

1.1	Puerto Rican Crested Toad Bufo lemur, <i>courtesy of Gail Susana Ross</i>	2
1.2	SAP System Infrastructure	4
1.3	SAP System developed at AIP	5
2.1	Far Field Geometry and Notation	8
3.1	Linear Sensor Array Model	13
3.2	Spatial Vector Coordinates	14
3.3	Basic Linear Beamforming Operation	15
3.4	Incident Angle Relationship	15
3.5	Basic Adaptive Beamformer Processor	18
3.6	Single Beam Pattern Formation	20
4.1	Scalar Butterfly Operation	29
5.1	Single Serial Processor, <i>courtesy of Kuck, MIT Lincoln Lab.</i>	40
5.2	Serial Processors in Parallel, <i>courtesy of Kuck, MIT Lincoln Lab.</i>	40
5.3	Parallel Speedup, <i>courtesy of Kepner, MIT Lab.</i>	41
5.4	Programming Environment for pMATLAB	44
5.5	Speedup achieved with pMATLAB, <i>courtesy of Kepner, MIT Lincoln Lab.</i>	44
5.6	Block Distribution Example for First Dimension	45
5.7	Block Distribution Example for Second Dimension	46
5.8	2-D Block Distribution Example	46
5.9	Data Mapping Example in pMATLAB	47
5.10	Data Distribution Example Among Processors, <i>courtesy of Kepner, MIT Lab.</i>	47
5.11	Parallel Utility Program File	48
5.12	Parallel Utility Program File	49
5.13	DFT Beamforming Implementation	49
6.1	TMS320C6713 DSK board	50
6.2	TMS320C6713 Architectural Diagram, <i>courtesy of Chassaing</i>	51
6.3	Locating the Folder DSP_BeamformingFiles	52
6.4	Files located in DSP_BeamformingFiles	53
6.5	Input Files in 32_sensors_input_files_1	53
6.6	Input Files in 64_sensors_input_files_1	54
6.7	DFT Beamforming Function	55
6.8	DFT_M_Modules_generation()	56
6.9	Linear_Combination_DFT_Modules()	56
6.10	Locating Setup CCStudio v3.3 and CCStudio v3.3 icons	57

6.11	Path to Code Composer Studio and Setup Tools	58
6.12	C6713 Device Cycle Accurate Simulator Environment Selection	59
6.13	Creating DSP Beamforming Project	59
6.14	Locating DSP Project	60
6.15	Adding Files to DSP Beamforming Project	61
6.16	Specifying Input Files	62
6.17	Selecting Build Options	62
6.18	Building Options for Compiler→Category→Basic	63
6.19	Building Options for Compiler→Preprocessor→Pre-Define Symbol	63
6.20	Building Options for Compiler→Advanced	64
6.21	Building Options for Linker→Libraries:	65
6.22	Compiling DSP Beamforming Project in Simulation Environment	65
6.23	Successful Compilation of the DSP Beamforming Project	65
6.24	Placing Input Files in Debug Directory	66
6.25	Loading DSP Beamforming Program Application	66
6.26	Selecting and Loading Executable File onto Simulated Target	67
6.27	Executing Program Application on Simulated Target	67
6.28	TMS320C6713 Target Board Connection, <i>courtesy of Chassaing</i>	68
6.29	C6713 DSK-USB Emulator Environment Selection	69
6.30	Compiling DSP Beamforming Project in Emulation Environment	70
6.31	Matlab Code for Reading DSP Results	70
6.32	Beam Pattern Formations from DSP	71
6.33	TMS320C6474 Block Diagram, <i>courtesy of Texas Instruments, Inc.</i>	72
7.1	Beamforming Algorithms Design	74
7.2	Input Matrix Format	74
7.3	Beam Pattern Formation Example for 64 Sensors	75
7.4	Essence of Kronecker Beamforming Technique	75
7.5	Beamforming Techniques Implementation Platforms	77
7.6	SNR Analysis	78
7.7	SNR Analysis (cont.)	79
7.8	DFT Beamforming MATLAB Code	81
7.9	DFT Kronecker Beamforming MATLAB Code	82
7.10	DFT Beamforming Pattern Example	82
7.11	Analyzing relationshipsensor spacing d and wavelength λ	83
7.12	Single Beam Pattern Formations for 32 sensors	85
7.13	Single Beam Pattern Formations for 64 sensors	85
7.14	Single Beam Pattern Formations for 128 sensors	86
7.15	Single Beam Pattern Formations for 256 sensors	86
7.16	Multiple Beam Pattern Formations for 32 sensors	87
7.17	Multiple Beam Pattern Formations for 64 sensors	87
7.18	Multiple Beam Pattern Formations for 128 sensors	87
7.19	Multiple Beam Pattern Formations for 256 sensors	87
7.20	Beamforming Algorithm Implementations on the DSP	88
7.21	DSP Multiple Beam Pattern Formations for 32 sensors	89
7.22	DSP Multiple Beam Pattern Formations for 64 sensors	89
7.23	Gumstix Multiple Beam Pattern Formations for 32 sensors	89

7.24	Gumstix Multiple Beam Pattern Formations for 64 sensors	90
7.25	pMATLAB Multicore Simulation Environment, <i>courtesy of Kepner, MIT Lincoln Lab.</i>	90
7.26	DFT Beam Pattern Formation in pMATLAB for 64 sensors	91
7.27	Kronecker Beam Pattern Formation in pMATLAB for 64	91
7.28	Speedup Analysis in pMATLAB for 256 Steering Directions	92
7.29	DFT Beamforming Platforms Comparison	93

List of Tables

7.1 Signal Statistical Metrics 80

Abbreviations

AIP	A utomated I nformation P rocessing
SAP	S ensor A rray P rocessing
MSN	M aster S ensor N ode
SSP	S ensor S ignal P rocessing
DOA	D irection o f A rrival
TDE	T ime D elay E stimation
AML	A pproximate M aximum L ikelihood
DFT	D iscrete F ourier T ransform
FFT	F ast F ourier T ransform
KPA	K ronecker P roducts A lgebra
DSP	D igital S ignal P rocessor

Symbols

t	Time	seconds
f_c	Carrier Frequency	Hertz
d	Sensor Spacing	meters
λ	Wavelength	meters
θ_0	Incidence Angle	radians and degrees
β_0	Steering Direction	Numeric

Dedicated to my family.....

Chapter 1

Introduction

Monitoring anurans and their associated environment has become vital in a world of declining biodiversity and ecological uncertainty. The mystery of declining amphibian populations is particularly worrisome. Scientists agree that biodiversity in general worldwide is declining, with amphibians far more threatened than any other taxa (32.5% of amphibians are threatened as compared to 23% of mammals and 12% of birds). Alarms were first raised in 1989 at the First World Congress in Herpetology when scientists started piecing together studies of declining populations. Further studies corroborate data with land use change, overharvesting, and the introduction of exotic species most often singled out as the main issues affecting amphibian loss. However, more recently the spread of infectious disease, toxins, and climate change, have been the key discussions at scientific meetings. Amphibians are indicators of environmental health and their declining populations are indicative of unhealthy ecosystems that ultimately affect human interests. To measure changes in biodiversity and their environmental contributions we must be able to monitor these changes and their effects. Traditionally, little baseline information was available from which to determine population declines in anurans. Wildlife monitoring was challenged by issues such as access to remote sites, methods to detect difficult species, and limited human resources to deal with labor intensive tasks. Improved monitoring and data collection systems have expanded the library of data available to compare population changes and processes. Thus, concrete advances in environmental surveillance monitoring become essential in our continued understanding of these processes.

Ideal monitoring programs collect and analyze audio data, couple this information with collected environmental parameters, and transfer the complete package to the relevant decision making agencies. Monitoring a federally threatened endemic bufonid (*Peltophryne lemur*) on the island of Puerto Rico has been challenged by the occurrence of its explosive but infrequent breeding activity during extreme rain events [1](see **Figure 1.1**). Further challenging its survival is breeding competition from the infamous marine toad, *Bufo marinus*.

Wireless sensor network systems have shown to be an effective method for monitoring, collecting, and analyzing data, without interfering or causing a negative impact to the surrounding environments. Much research has been conducted in determining the ideal characteristics that such a system should possess in order to deliver a good performance.



Figure 1.1: Puerto Rican Crested Toad *Bufo lemur*, *courtesy of Gail Susana Ross*

1.1 Problem Formulation

The WALSAIP (Wide Area Large Scale Automated Information Processing) project, a program sponsored by the National Science Foundation of the United States and coordinated by the Institute for Computing and Informatics Studies of the University of Puerto Rico at Mayaguez, has been developing and field testing an array processing system (SAP) framework to continuously monitor bio-acoustic signals indicative of breeding activity of endemic and introduced anurans at key points in Puerto Rico. The SAP framework allows for the use of signal processing techniques such as acoustic beamforming for source location and adaptive direction of arrival (DOA) determination as well as novel time-frequency signal analysis techniques such as the cyclic short-time Fourier transform, the modified ambiguity function, and the modified Wigner distribution for bio-acoustics sound characterization. The SAP framework infrastructure is being developed to foster the study of advanced signal-based information processing tasks to enhance anuran bio-acoustics understanding.

There is a need to design and implement a network sensor system, which serves as a tool for biologists to locate, trace, and monitor different species, in a non-intrusive way, without changing the surrounding environment. Important issues, regarding network sensor systems,

include communication capacity, sensor spacing, number of sensors, time synchronization, and array geometry. Much research has also been conducted for determining an effective acoustics beamforming technique for obtaining accurate estimates of source location through the computation of direction of arrival (DOA).

Current acoustic algorithms are designed for single source location estimation. Such algorithms reveal degradation in robustness and performance for estimating DOA when multiple sources are present in the environment. Also, such algorithms commonly assume that the source signals are located near the sensory, and are characterized to be narrowband signals. This represents a challenge for acoustics analysis, since such signals are characterized to be wideband signals, and may originate far from the sensor array. Also important factors such as noise and reverberance can further more affect the performance of such algorithms [2].

An important consideration of this work is the mapping problem of the computational methods to distributed computational structures, such as a parallel computational structure. Previously developed work presented in [3] consider the use of Discrete Fourier Transform (DFT) computational methods that center on the use of the computational properties of such transform and some properties of the Kronecker products algebra (KPA) to formulate desired beamforming operations, when the number of sensors in a linear array is equal to the number of incidence angles that can be detected. Hence there is a further need to extend this work in order to apply DFT Kronecker Multi-beamforming in a parallel computational environment and evaluate its performance.

1.2 Proposed Solution

A sensor array processing (SAP) testbed has been developed at the AIP Laboratory that is utilized to test these computational methods in the field since bioacoustics signal analysis deals, in general, with the study of sound signals emitted by animals and humans. The framework behind the SAP testbed consists of a novel solar-powered, wireless-based, distributed signal processing system infrastructure which may be deployed in the field and accessed through cyberspace with relative ease. The system proposed is stand alone, and allows for a setup in remote settings, operation in inclement weather conditions, and data transfers over the internet, without the need for frequent field visits.

The basic SAP system infrastructure has three essential elements for determining the DOA of incoming signals (see **Figure 1.2**):

- Set of sensor signal processing (SSP) nodes

- Wireless routing mechanism
- Linux-based high-performance embedded computing unit

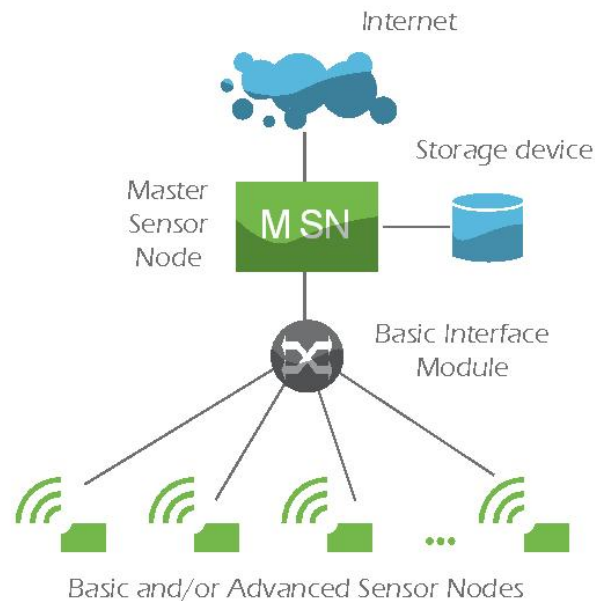


Figure 1.2: SAP System Infrastructure

Acoustic signals are collected and processed by sensor signal processing nodes (SSP). Once the acoustic data is acquired, sampled, and processed by the SSP nodes, such data and associated metadata are sent to the principal node, known as the master sensor node (MSN), via a wireless routing mechanism. The MSN stores information in a database, which can then be transferred to a server to be accessed by interested parties.

Small low-cost embedded Linux-based computers, known as Gumstix Verdex, are used as the SSP, due to the following capacities that they possess: wireless communication, data acquisition and storage, data processing, and low power consumption. The equipment specifications and operating system used for the SAP system developed at the AIP are presented as follows (see **Figure 1.3**:

- MSN Intel Core 2 Duo T7200 Merom 2.0GHz 4M shared L2 Cache Socket M 34W Dual-Core Processor
- Gumstix Verdex as the SSP nodes
 - Number of SSP nodes - 5
 - Sampling Frequency of each SSP node - 44 KHz
 - Speed 600MHz

- Memory: 128MB RAM, 32MB Flash
- Operating System - Linux Fedora 8

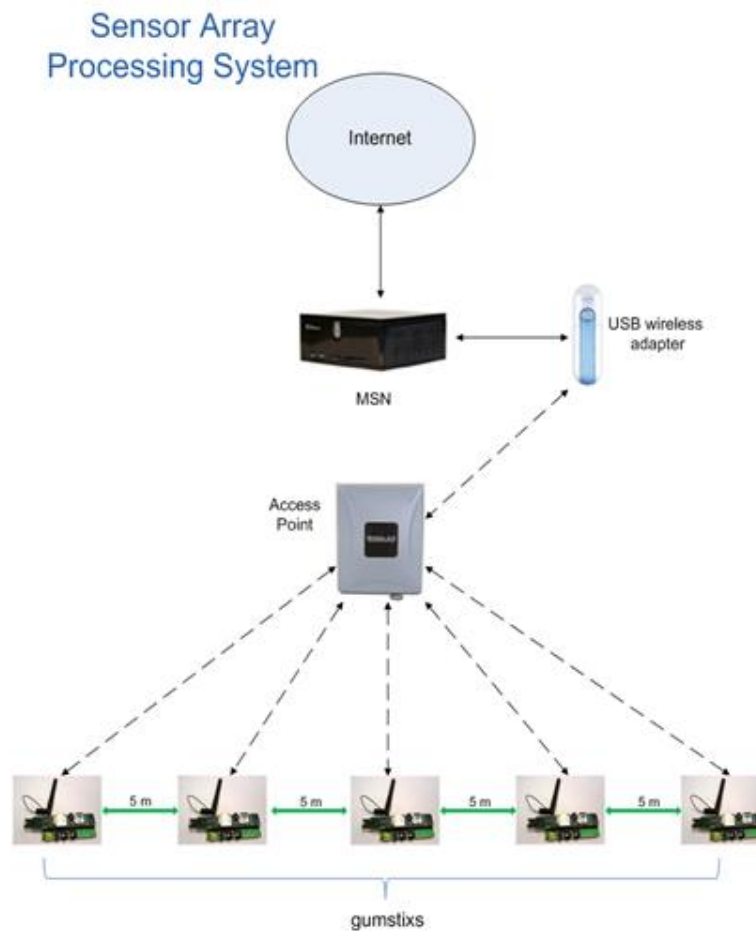


Figure 1.3: SAP System developed at AIP

DFT beamforming techniques are further extended, through the usage of Kronecker products algebra (KPA) in order to address the mapping of such techniques onto parallel structures. The DFT beamforming algorithms are initially implemented, using the MATLAB numeric computation and software visualization package, as an ideal tool for developing, testing, and optimizing signal processing applications. Once the DFT beamforming algorithms are tested and optimized using MATLAB, the DFT beamforming techniques are implemented on the high performance, floating-point TMS320C6713 digital signal processing (DSP) board, from Texas Instruments. These algorithms are also designed to be executed by a Gumstix. Parallel implementation and performance analysis is conducted, using a parallel programming modeling environment, named pMATLAB. Such tool is used to further address the mapping problem of computational methods to distributed parallel computational structures. The DFT beamforming operation performance is compared among three platforms: DSP, Gumstix, and pMATLAB.

1.3 Thesis Research Contributions

The main contributions of this research are presented as follows:

- The first and principal contribution made by this work is the development of computational methods for performing bioacoustics beamforming algorithms with the objective of estimating direction of arrival (DOA) from signals arriving at a set of acoustic sensors. The computational methods center on the use of the discrete Fourier transform and Kronecker products to formulate the desired beamforming operations on a parallel computational structure.
- A SAP testbed has been developed at the AIP Laboratory with the purpose of testing these computational methods in the field and serving as an ideal tool for biologist to monitor, collect, and analyze data in a passive way, and retrieve processed data via wireless communication, thus avoiding the necessity of frequent visits to the field.

1.4 Thesis Organization

The remaining of this work is presented as follows: Chapter 2 describes the background work and previous research conducted, related to the following: design, implementation, and ideal characteristics for distributed sensor array networks, the advantages and disadvantages of current beamforming algorithms developed for obtaining source location estimates, and the novel approach of applying DFT Kronecker beamforming operations, suitable for parallel computational structures. Chapter 3 provides a descriptive theoretical background, defining important concepts such as acoustics signal characterization, basic beamforming operation, and DFT multi-beamforming operation. Chapter 4 is completely dedicated to describing in detail the Kronecker products algebra, and how it can be integrated into the DFT beamforming techniques, in order to further reduce the computational hardware complexity involved for sensor array systems consisting of a large number of sensors. Chapter 5 presents the parallel programming modeling environment, named pMATLAB, which has been utilized to study the computational performance of parallel implementation techniques. Chapter 6 describes the implementation of the DFT beamforming algorithms on the DSP unit TMS320C6713, from Texas Instruments. Experimental results are presented and explained in details in chapter 7, culminating with conclusions and future work in chapter 8.

Chapter 2

Background and Related Work

This chapter surveys previous work related to the area of acoustic beamforming and the types of algorithms developed with the main goal of obtaining accurate source localization estimations. Beamforming by means of wireless distributed acoustic array networks is also discussed.

2.1 Bioacoustics Signal Analysis

Bioacoustics signal analysis works with extracting important information, such as source location and propagation medium, from sound signals emitted by animals, including humans. We are interested in using bioacoustics signal analysis for monitoring the anuran family. Such family includes species of frogs and toads. The study of amphibians has gained a great importance for distinct biological groups and research, where monitoring and tracking changes in population, especially endangered species, has become a top main concern.

2.2 Acoustics Signal Analysis

Acoustics can be defined as the science of sound, which studies its production, transmission, and effects. Such analysis deals with the properties of a sound waveform, such as the amplitude S_0 , fundamental frequency f_c , its duration t , and other properties of its frequency spectrum.

2.3 Signal and Sensor Array Model

A signal model has been formulated in [4]. Through this formulation, the signal detected at each sensor is modeled in terms of the *relative time delay* between the p_{th} sensor and the

array centroid r_c . In this case, the centroid of each array is used as reference point, in order to model a signal that is received at each sensor p in the array:

$$x_p[n] = \sum_{m=1}^M S^{(m)}[n - t_{cp}^{(m)}] + w_p[n], \quad (2.1)$$

where $x_p[n]$ denotes the signal detected at sensor p , $S^{(m)}$ the m^{th} source signal, $w_p[n]$ the white Gaussian noise, $t_{cp}^{(m)}$ the relative time delay between the p^{th} sensor and the array centroid r_c , and $t_c^{(m)}$, $t_p^{(m)}$ the propagation time from source $S^{(m)}$ to the centroid and sensor p respectively, at time instance n . The relative time delay is defined as:

$$t_{cp}^{(m)} = t_c^{(m)} - t_p^{(m)}. \quad (2.2)$$

This signal model assumes that each of the M sources is located in the *far-field*. In the far-field case as the distance between the source and array becomes larger, the signal detected at the sensors becomes planar and parallel (see **Figure 2.1**). Effective microphone spacing and acoustic orientation calibration must be determined in order to make the beamforming of wideband acoustic signals more robust. Using effective microphone spacing helps take into consideration unavoidable ambient factors such as noise, interference, and multipath effects.

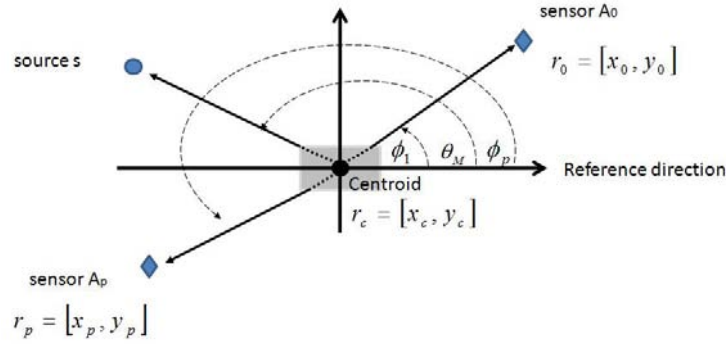


Figure 2.1: Far Field Geometry and Notation

A second model has also been proposed in [5] to include reverberant effects (signal reflections). In order to accomplish this, the impulse response function of each sensor must be blindly identified. The impulse responses model the reverberant effects. The disadvantage is that for a small number of microphones, this model can fail, due to the probability that the transfer functions can share common zeros. The number of sensors would need to be increased in order to have a better model of the reverberant effects.

For the model presented in **Figure 2.1**, microphone spacing has been seen to affect the beamforming performance. As found in literature, effective microphone spacing and acoustic orientation calibration must be determined in order to make the beamforming of wideband

acoustic signals more robust. Using effective microphone spacing helps take into consideration unavoidable ambient factors such as noise, interference, and multipath effects. It has been demonstrated that large microphone spacing results in beam patterns with side lobes. When exposed to noise, interference, and multipath effects, the side lobes can become taller than the main lobe. Hence this can cause a significant error in estimating DOAs. The microphone spacing guide line $d \leq \lambda/2$ has been shown to provide better DOA estimates.

Array shape and number of sensors in each array was one of the main features discussed, for the array model proposed [6]. By incrementing the number of sensors from one to four in each array, it was determined that arranging four square arrays, each with four sensors, provided better results in localizing multiple sources.

Traditional, centralized systems, have found to possess the disadvantage of having a small coverage of region under observation, since a small number of sensors are placed near the central processing node. It has been demonstrated that distributed local processing, instead of central processing, is more energy efficient in the sense that subarrays conduct the local processing of DOA estimation, source identification, and detection [7].

2.4 Sophisticated Algorithms

The need for source detection, location, and tracking has become a great interest for a variety of applications: military, industrial, speech analysis, bioacoustics analysis, etc. Such applications share a common objective: to design low-cost microsensor arrays which can form a distributed network, and provide the direction of arrival (DOA) estimates for source location. It has become relevant the interest of designing and implementing low-cost microsensor arrays, composed of sensor signal processing nodes (SSP) with the following capacities: wireless communication, signal acquisition, storage, and processing operations. In addition, extensive search for obtaining accurate source localization estimates has led to the development of two classes of algorithms: i) TDE (Time Delay Estimation) algorithms, which provide a closed-form solution and ii) AML (Approximate Maximum Likelihood) based method for parametric DOA estimations.

It has been proposed that AML, despite its complexity, provides better estimation accuracy [2]. In addition, when considering multiple-source wideband signals, AML has been found to be a more attractive method, by processing the data in frequency domain [8]. This is due to the fact that in the frequency domain, a wideband signal's spectrum can be taken as a series of narrowband spectrums for each frequency bin. In addition, more reliable noise model is provided in the frequency-domain than in the time domain model. The main disadvantage

of the TDE-based algorithms found in literature is that, even though they work well for both narrowband and wideband signals, these algorithms usually consider a single source case.

Algorithms have also been developed in order to estimate adaptable filter coefficients or weights of a blind beamformer. Blind beamforming operation is applied when information regarding the sensors is unknown and the steering directions and beamforming weights need to be estimated. A General Eigenvalue (GEV) based using a stochastic gradient ascent approach has been proposed in order to estimate and adapt the filter coefficients or weights of a blind beamformer [9], for each frequency bin. In order to do this, the power spectral densities of the signal detected at the microphones and the power spectral density of the noise in speech pauses are estimated. It has been found that GEV converges faster than other Stochastic Gradient (SG) algorithms and produces less variation in the SNR gain [9]. GEV converges slower than Recursive Least Squares (R-LS), focusing more computational load during the presence of desired signal, making it more robust [9].

Adaptive array processing has also been implemented with the constraint that the desired spatial output response remains approximately constant for each frequency bin [10]. All primary filters can be derived from a single set of reference coefficient filters that produce the desired frequency response obtained at a certain reference location and sampling period T [11]. There are two methods for achieving this: multirate and single rate methods. It was shown that the multirate rate, at a higher sample frequency, resulted in fewer filter coefficients [12].

In ideal condition scenario in which information regarding to the sensor array is known, it has been shown in [3], that the Discrete Fourier Transform (DFT) can be applied in the beamforming operation. Using DFT beamforming has the advantage of representing the beamforming operation as a set of M Discrete Fourier Transforms DFT modules, where N is the size of each segment and L denotes the number of sensors, and should be a power of 2:

$$L = M * N. \quad (2.3)$$

In this way, the beamforming operation can be adapted and scaled, according to the hardware architecture, by dividing the operation in M modules. Such operation is known as the DFT Kronecker products.

2.5 Structural Beamforming Mapping and Architecture

It has been seen in literature that the preferred AML algorithm for estimating source location generally maps to the blind beamforming situation. Such case represents a non-ideal condition, since information regarding to the sensor array is unknown. On the other hand, when

information of the sensor array is known, such as the microphone spacing and sensor position, the problem no longer maps to a blind beamforming operation. Instead, this situation maps to an ideal condition, where the information related to the sensor array can be used in order to obtain source location estimates. In this case, DFT beamforming algorithm can be applied.

Such hardware architectures in which the beamforming operation (AML and DFT) can be implemented include multicore. In the case of DFT, the Kronecker products can be easily used in order to divide the beamforming operation among M processors present in the multicore system. For example, it is shown how Kronecker formulations can help to formulate fast Fourier transform algorithms, and other DSP operations, providing efficient implementation on parallel and vector processing architectures [13]. In addition, it is demonstrated how the DFT computation of a L -point discrete signal $x(n)$ can be expressed in a Kronecker product form, through sparse matrices of the same dimension, when L is a composite number of the form $L = RS$, by rearranging the input discrete signal as a two-dimensional array [14].

Hence the DFT beamforming can be carried out for each module in parallel form, especially when the number of sensors is large, and Kronecker products can help provide an efficient implementation, according to the appropriate computer architecture.

Chapter 3

Theoretical Framework

3.1 Signal Characterization

A model for a plane wave signal [3] which arrives at a linear sensor array has the following form:

$$s(t, r) = s_0 e^{+j(2\pi f_c t - \langle k, r \rangle)}. \quad (3.1)$$

In the above model, $s(t, r)$ is a continuous signal, where s_0 denotes the amplitude of the signal, f_c represents the carrier frequency, t is the time instances, k denotes the wave vector, and r is known as the spatial vector. The spatial vector r is defined by the following coordinates, in a three-dimensional coordinate system:

$$r = (r_x, r_y, r_z) \quad (3.2)$$

In the same manner, the wave vector k is defined by the following coordinates, in a three-dimensional coordinate system:

$$k = (k_x, k_y, k_z) \quad (3.3)$$

The term $\langle k, r \rangle$ denotes the dot product between the wave vector k and the spatial vector r , which is defined as follows:

$$\langle k, r \rangle = k_x \cdot r_x + k_y \cdot r_y + k_z \cdot r_z. \quad (3.4)$$

Substituting the dot product expressed in the above equation, into the signal model expressed in (3.1), the following signal model, in terms of the coordinates of the spatial vector r , is

derived:

$$s(t, r_x, r_y, r_z) = s_0 e^{j(2\pi f_c t - (k_x \cdot r_x + k_y \cdot r_y + k_z \cdot r_z))}, \quad (3.5)$$

The following figure presents a linear array of N sensors, where d represents the distance between each pair of sensors, D_N is the total distance of the linear array, measuring from the first sensor denoted A_0 to the last sensor in the linear array A_{N-1} . As shown in **Figure 3.1**, r_x

Linear Sensor Array Layout

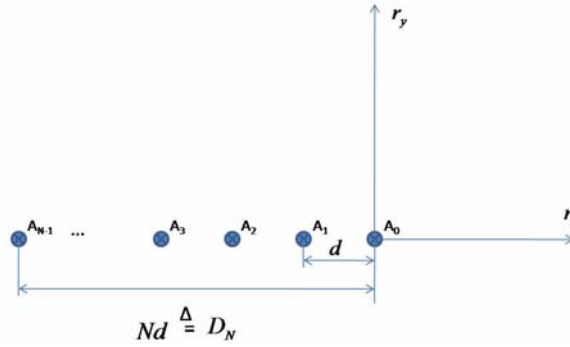


Figure 3.1: Linear Sensor Array Model

denotes the horizontal direction with respect to each sensor in the linear array, also known as x . The vertical direction, perpendicular to each sensor in the linear array, is represented by r_y , which can also be denoted as y . In a three-dimensional Euclidean space, with a right-handed coordinate system, the operation $r_x \times r_y$ denotes a cross-product operation, which defines the vector r_z , also known as z , perpendicular to the plane formed by r_x and r_y . The direction of r_z , given by the right-hand rule, corresponds to the vector pointing away from the intersection of r_x and r_y , as shown in **Figure 3.2**, as r_x closes towards the vector r_y . By observing **Figure 3.1** and **Figure 3.2**, the only spatial coordinate of the r that changes is r_x , with respect to each sensor, as the wave plane moves from one sensor to the next, maintaining r_y and r_z constant. Since the signal model (t, r) is a function of the changes in the spatial vector r , besides the time instance t , such model can be reduced to considering only the coordinate r_x of the spatial vector r . Hence the dot product $\langle k, r \rangle$ is reduced to $k_x \cdot r_x$, and the signal model may be re-expressed as follows:

$$s(t, r_x) = s_0 e^{j(2\pi f_c t - (k_x \cdot r_x))}, \quad (3.6)$$

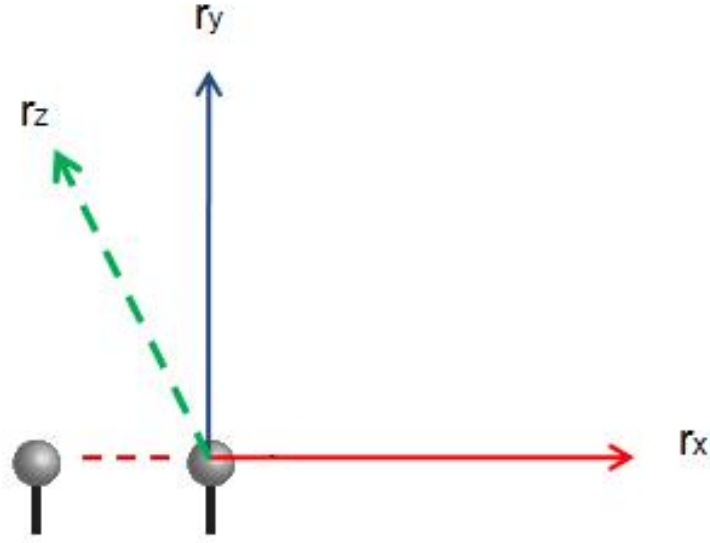


Figure 3.2: Spatial Vector Coordinates

The value $k_x \cdot r_x$ of the dot product can be simplified to $k \cdot x$, resulting in the following expression for the signal model $s(t, x)$, now in terms of the time instance t and the spatial coordinate x :

$$s(t, x) = s_0 e^{+j(2\pi f_c t - k \cdot x)}. \quad (3.7)$$

The term 2π in the signal mode $s(t, x)$ can be factored out as follows:

$$s(t, x) = s_0 e^{+j2\pi(f_c t - \frac{k \cdot x}{2\pi})}. \quad (3.8)$$

The factor $\frac{k}{2\pi}$ may also be expressed as $\frac{1}{\lambda}$, where λ corresponds to the wavelength of the incoming source signal, with the velocity of light being $c = f_c \cdot \lambda$. Therefore the signal model may be reformulated as follows:

$$s(t, x) = s_0 e^{+j2\pi(f_c t - \frac{x}{\lambda})}. \quad (3.9)$$

It is assumed that the plane wave $s(t, x)$ arrives at the linear sensor array at an incidence angle θ_0 with respect to the boresight. The boresight is defined as the vertical axis, which is normal to each sensor positioned in the linear array along the horizontal axis (see **Figure 3.3**). In order to incorporate the incidence angle θ_0 into the signal model, $s(t, x)$ needs to be reformulated to take into consideration this important factor. **Figure 3.4** presents a detailed diagram concerning how the incidence angle is integrated into the signal model. According to **Figure 3.4**, the angle of interest is θ_0 , which is measured between the boresight axis and the wave vector k originating from the plane wave. It can be shown that a relationship exists between the incidence angle θ_0 and the angle γ , presented in **Figure 3.4**. Since the boresight

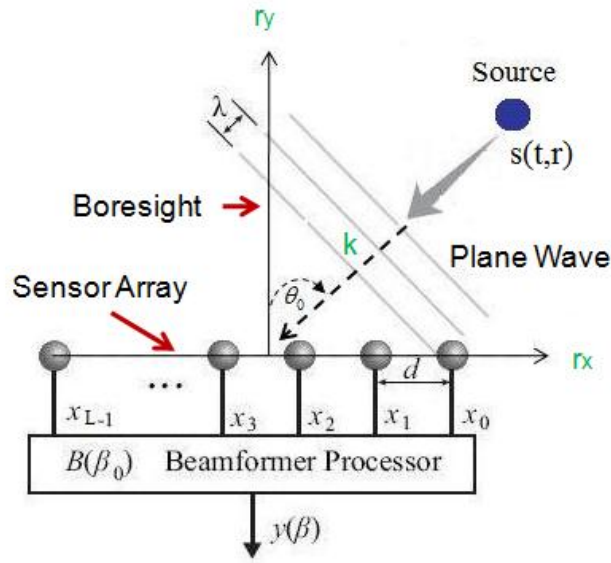


Figure 3.3: Basic Linear Beamforming Operation

Incident Angle Relationship

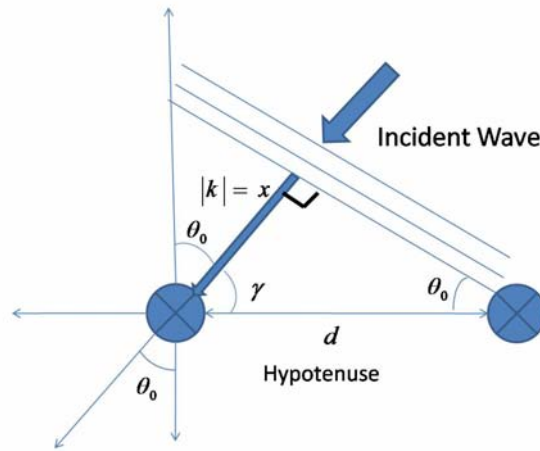


Figure 3.4: Incident Angle Relationship

axis forms a 90° angle, which corresponds to $\frac{\pi}{2}$ in radians, with respect to each sensor in the array, the angle γ may be expressed as follows:

$$\gamma = \left(\frac{\pi}{2} - \theta_0 \right) \quad (3.10)$$

By observing carefully **Figure 3.4**, a right triangle is formed between the wave vector k , the plane wave, and the distance d between a pair of sensors. The right triangle ($\frac{\pi}{2}$ in radians) is formed between the wave vector k and the plane wave. The distance $|k| = x$ denotes the

delay it takes the plane wave to reach each sensor in the linear array. Hence, it is desired to determine this value, using the relationship established in the previous expression of γ in terms of the incidence angle θ_0 . Using trigonometry properties and taking the distance d as the hypotenuse of the right triangle, the following relationship is established:

$$\cos(\gamma) = \frac{x}{d}. \quad (3.11)$$

By substituting $\gamma = (\frac{\pi}{2} - \theta_0)$ in the above expression, the following is obtained:

$$\cos\left(\frac{\pi}{2} - \theta_0\right) = \frac{x}{d}. \quad (3.12)$$

By applying trigonometry properties, the following expression is derived:

$$\cos\left(\frac{\pi}{2} - \theta_0\right) = \sin(\theta_0) = \frac{x}{d}. \quad (3.13)$$

Since it is desired to determine the distance $|k| = x$, representing the propagation delay, the above expression is solved for x , arriving at $d \cdot \sin(\theta_0)$. By substituting x in the signal model expressed in equation 3.8, the subsequent signal model formulation is obtained, where the incidence angle θ_0 is incorporated to the signal model:

$$s(t, x) = s_0 e^{+j2\pi(f_c t - \frac{d}{\lambda} \sin(\theta_0))}, \quad (3.14)$$

recalling that d represents the spacing between each pair of sensors in the linear array.

The position of a sensor A_n in the linear array may be specified in the signal model, in terms of its distance from the first sensor A_0 . This is achieved by substituting d with $-nd$, where $n = 0, 1 \dots N-1$, N being the total number of sensors in the linear array. Hence the new signal may be spatially sampled in terms of each sensor A_n in the array, where $-nd$ represents the distance of the sensor A_n with respect to the first sensor A_0 . The new signal model $s(t, x)$ is presented as follows:

$$s(t, x) = s_0 e^{+j2\pi(f_c t + \frac{nd}{\lambda} \beta_0)} = s_0 e^{+j2\pi f_c t} \cdot e^{+j2\pi(\frac{nd}{\lambda} \beta_0)}, \quad (3.15)$$

where β_0 corresponds to the steering direction of the input signal, and is defined as follows:

$$\beta_0 = \sin(\theta_0), \quad -\frac{\pi}{2} \leq \theta_0 \leq \frac{\pi}{2}. \quad (3.16)$$

The term $s_0 e^{+j2\pi f_c t}$ may be represented as a function $v_0(t) = s_0 e^{+j2\pi f_c t}$, thus $s(t, x) = v_0(t) e^{+j2\pi(\frac{nd}{\lambda} \beta_0)}$. Considering a single time instance $t = t_0$, the following formulation for the

signal model is derived:

$$s(t_0, nd) = v_0(t_0)e^{+j2\pi(\frac{nd}{\lambda}\beta_0)}|_{t=t_0} \quad (3.17)$$

The signal model $s(t_0, nd)$ now represents the signal captured at the n^{th} sensor. In addition, this new signal model now integrates the propagation delay $nds\sin(\theta_0)$, which represents the time it takes for the source signal to propagate from the first sensor A_0 , which is considered as a reference sensor or point in the linear array, to the n^{th} sensor A_n in the array. Hence, it can be observed from the previous signal model, that it is now a changing function of n , and remains constant for each time instance t . Also, the carrier frequency f_c , does not change within the same wave plane from which the signal originates. The term $e^{+j2\pi(f_c t)}$ therefore does not represent an important contribution when modeling the input signal in terms of the position of each sensor in the linear array. Eliminating this term from the signal model, we arrive at the final derivation of a discrete signal model formulation:

$$\phi_n(\beta_0) = \phi_0 \cdot e^{+j2\pi(\frac{nd}{\lambda}\beta_0)}, n \in Z_N, \quad (3.18)$$

where Z_N represents the set of natural numbers $0, 1, 2 \dots N-1$, and ϕ_0 represents the amplitude of the new discrete signal model $\phi_n(\beta_0)$. Spatial samples from all of linear array sensors can be expressed as an input column vector as follows:

$$\Phi(\beta_0) = [\phi_0(\beta_0), \phi_1(\beta_0), \dots, \phi_{N-1}(\beta_0)]^T, \quad (3.19)$$

where $\Phi(\beta_0)$ is the spatially sample arriving plane wave coming from the steering direction β_0 .

As expected for a linearly arranged sensor grid, input signals can be detected at incident angles between -90° and 90° , corresponding to $-\frac{\pi}{2} \leq \theta_0 \leq \frac{\pi}{2}$ in radians. Hence the resulting steering direction lies in range between -1 and 1.

3.2 Basic Beamforming Operation

In the time domain, a beamforming operation is performed using the time delay operation in order to obtain the coherent sum $N \cdot \phi_0$ from all sensor signals. In this work, beamforming is treated from the point of view of a linear transformation over a finite discrete input signal.

Figure 3.5 carefully illustrates with explicit details the basic components of a general, adaptive beamformer processor, as described in [15]:

As explained in the previous section, the first component $s(t, r)$ represents the incoming signal, originating from a plane wave, arriving at each sensor of the array, at an incidence angle θ_0 ,

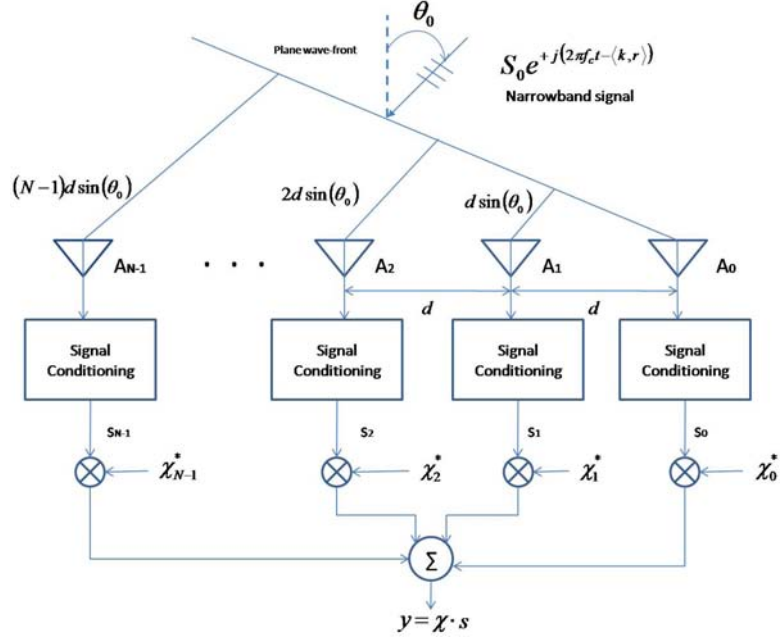


Figure 3.5: Basic Adaptive Beamformer Processor

with respect to the boresight. As it can be observed from the diagram presented in **Figure 3.5**, $nd \sin(\theta_0)$ represents, for $n = 0, 1, \dots, N-1$, the propagation delay or the time it takes for the source signal s to propagate from the first sensor A_0 to the n^{th} sensor A_n .

The second component of the beamformer processor is the linear sensor array itself, composed of N sensors, ranging from the first sensor A_0 to the last sensor A_{N-1} , where the sensors are equally spaced by a distance d . A signal conditioning module is added in order to optimize the signal received at each sensor for the beamforming operation.

The third component χ_n , for $n = 0, 1, \dots, N-1$, represents the weighting factor, associated with each receiving channel of the array, in which for this case, each receiving channel corresponds to a sensor. In this stage of the beamformer processor, the signal captured at each sensor is multiplied by the corresponding weight χ_n .

The fourth and final component of the beamformer processor is represented by the sum Σ , where the multiplied signal component obtained from each sensor are added together, resulting in the beam pattern output:

$$y = \chi \cdot s, \quad (3.20)$$

where \mathbf{s} is a column vector representing the values of the incoming source signal, and χ is a row vector that contains all of the weighting factors, as defined as follows:

$$\chi = [\chi_0, \chi_1, \dots, \chi_{N-1}] \quad (3.21)$$

In this work, a beamforming vector is defined as the following row vector:

$$B(\beta_0) = \left[1, e^{-j2\pi \frac{d}{\lambda} \beta_0}, e^{-j2\pi \frac{2d}{\lambda} \beta_0}, \dots, e^{-j2\pi \frac{(N-1)d}{\lambda} \beta_0} \right]. \quad (3.22)$$

The purpose is to obtain the coherent sum $N \cdot \phi_0$ from the following beamforming transformation operation:

$$B(\beta_0) \cdot \Phi(\beta_0) = N \cdot \phi_0. \quad (3.23)$$

In general, the beamforming vector is used to steer an input vector $\Phi(\beta_0)$ toward the steering direction β_0 , obtaining as result a beam pattern of a linear array steered to a specific direction. The product $L \cdot \phi_0$ can also be represented as:

$$B(\beta_0) \cdot \Phi(\beta_0) = \sum_{n=1}^{L-1} \phi_n \cdot e^{-j2\pi n\nu}. \quad (3.24)$$

where $\nu = (d/\lambda)\beta_0$ is called the spatial spectrum variable. Hence, when an input signal is coming from a source, at a steered direction β_0 , the beamforming transformation results in multiplying the signal amplitude ϕ_0 by the number of L sensors in the linear array.

3.3 DFT Multi-Beamforming Technique

When an incoming signal is received by a linear array of sensors, the steering direction, and thus the DOA of the signal, is unknown. This leads to considering a set of M possible steering angles from the range $-\frac{\pi}{2} \leq \theta_0 \leq \frac{\pi}{2}$ are considered, resulting in the definition of the multi-beamforming matrix as follows:

$$B = \begin{bmatrix} B(\beta_0) \\ B(\beta_1) \\ \vdots \\ B(\beta_{M-1}) \end{bmatrix} = \begin{bmatrix} 1 & e^{-j2\pi \frac{d}{\lambda} \beta_0} & \dots & e^{-j2\pi \frac{(L-1)d}{\lambda} \beta_0} \\ 1 & e^{-j2\pi \frac{d}{\lambda} \beta_1} & \dots & e^{-j2\pi \frac{(L-1)d}{\lambda} \beta_1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & e^{-j2\pi \frac{d}{\lambda} \beta_{M-1}} & \dots & e^{-j2\pi \frac{(L-1)d}{\lambda} \beta_{M-1}} \end{bmatrix}. \quad (3.25)$$

Such matrix consists of M rows, where M corresponds to the number of steering direction that the linear array is capable of detecting. Each row corresponds to a beamforming transformation operation defined for a specific steering direction, and contains L values, L being the number of sensors in the linear array. This multi-beamforming matrix is then applied to the input signal. When multiple sources are present, the multi-beamforming matrix is applied to an input matrix, where each column denotes a single spatially sampled input signal, arriving at a predetermined steering direction. An example of a single beam pattern formation obtained from a linear array of 64 sensors can be depicted in **Figure 7.3**. The main lobe of such pattern formation

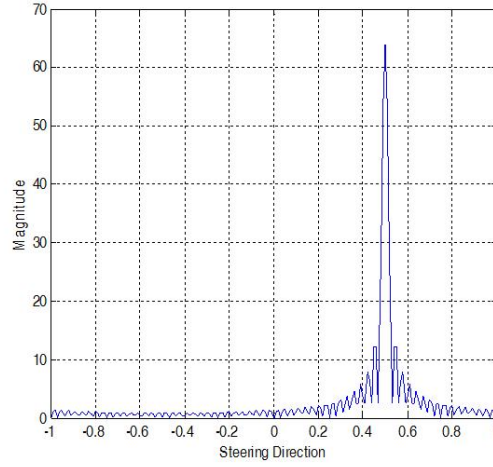


Figure 3.6: Single Beam Pattern Formation

represents the magnitude of the dominant signal, at a maximum value of 64, assuming the amplitude of the signal to be equal to one. The steering direction associated with the main lobe is used to determine the corresponding DOA or incidence angle.

The multi-beamforming matrix B becomes the DFT transform matrix when the number of steering angles M is equal to the number of sensors L in the linear array. A single beamforming transformation can be redefined as:

$$B(\beta_n) = \left[1, e^{-j2\pi\frac{n}{L}}, e^{-j2\pi\frac{2n}{L}}, \dots, e^{-j2\pi\frac{(L-1)n}{L}} \right]_{n=0,1,\dots,L-1}. \quad (3.26)$$

Thus, for the multi-beamforming operation, the matrix B can be expressed as follows:

$$B = \begin{bmatrix} B(0) \\ B(\frac{1}{L}) \\ \vdots \\ B(\frac{L-1}{L}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & W_L & \dots & W_L^{L-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & W_L^{L-1} & \dots & W_L^{(L-1)(L-1)} \end{bmatrix}. \quad (3.27)$$

where $W_L^n = e^{(-j\frac{2\pi}{L}) * n}$ is known as the twiddle factor. Hence the resulting beamforming matrix becomes the $L \times L$ DFT matrix, also known as the F_L matrix.

Chapter 4

DFT Kronecker Algorithms

4.1 Introduction to Kronecker Products and Fast Fourier Transforms

The Kronecker product is a very useful operation, based on the algebra of Kronecker products, which serves as a mathematical language for modeling, designing, analyzing, implementing, and modifying the computation of the DFT [14]. This operation can be used to formulate the fast Fourier transform (FFT) algorithms. Such algorithms have been shown to be efficient for computing the DFT. As shown previously, in chapter 3, when the number of sensors L is equal to the number of steering angles, the beamforming matrix B results to be the $L \times L$ DFT matrix, also known as the F_L matrix. The FFT, on the otherhand, represents a factorization of this DFT matrix, into M factors or modules, which can be applied to a discrete input vector, dividing it into M modules as well. For the beamforming operation, the input vector corresponds to the spatially sampled input vector, with respect to the sensors in the linear array. This will be explained in further details throughout the remaining sections of this chapter.

An important issue or concern arises as the number of points (for the beamforming operation, this would be the number of sensors) L or length defined for the DFT increases significantly, causing the computational effort and procedure to become tedious, causing a large usage of computational resources. However, this can be solved by using Kronecker products in combination with the FFT algorithms. The advantage of this is that by factorizing the DFT (beamforming) matrix into M factors or modules, each of length $\frac{L}{M}$, the computational process and effort is simplified. This in turn, simplifies in a more concise and clear manner, the analysis and implementation of the DFT matrix.

4.2 The Fast Fourier Transform

As stated in the previous section, the FFT has been shown to be an effective algorithm for computing the DFT matrix, in a factorized form [14]. Here, the factorization of the DFT matrix using FFT is carefully illustrated in detail.

For an arbitrary discrete signal $x(n)$, of finite-length L , the DFT is defined as follows:

$$X(k) = \sum_{n=0}^{L-1} x(n) e^{(-j2\pi nk/L)}; k = 0, 1, \dots, L-1; j = \sqrt{-1}. \quad (4.1)$$

In order to arrive at the definition for the DFT of an arbitrary discrete signal $x(n)$, such formulation is derived from the discrete-time Fourier transform (DTFT), or simply known as the Fourier transform of a discrete signal $x(n)$. The DTFT of a discrete signal $x(n)$ is denoted as $X(e^{j\omega})$ or $X(\omega)$, where ω represents the real frequency variable, and is defined as follows:

$$X(e^{j\omega}) = X(\omega) = \sum_{n=-\infty}^{\infty} x(n) e^{j\omega n}. \quad (4.2)$$

As demonstrated by the DTFT formulation, the discrete signal $x(n)$ is considered to be of infinite length. However, for the beamforming operation, finite-length sampled input signals of length L are considered. For this case, only L values of $X(\omega)$, which are also called frequency samples, can be used to determine or define $x(n)$ and $X(\omega)$. This is done by uniformly sampling $X(\omega)$ at L points $\omega = \omega_k, k = 0, 1, \dots, L-1$. Hence, the DFT is derived from the DTFT, as follows:

$$X(k) = X(\omega_k) \big|_{\omega_k = \frac{j2\pi k}{L}} = \sum_{n=0}^{L-1} x(n) e^{j\omega_k n} = \sum_{n=0}^{L-1} x(n) e^{(-j2\pi nk/L)}. \quad (4.3)$$

In addition to the formulation defined for the DFT of a discrete signal $x(n)$, this transform can also be expressed in matrix form. This is achieved by expanding the summation operation given in (4.1) as $X = F_L x$, where F_L is a $L \times L$ DFT matrix, defined as:

$$F_L = [W_L^{kn}] \big|_{k,n=0,1,\dots,L-1}; W_L = e^{(-j2\pi/L)}, \quad (4.4)$$

and the vectors X and x are defined as:

$$X = \begin{bmatrix} X(0) \\ X(1) \\ \vdots \\ X(L-1) \end{bmatrix}; x = \begin{bmatrix} x(0) \\ x(1) \\ \vdots \\ x(L-1) \end{bmatrix} \quad (4.5)$$

The signals $x(n)$ and $X(k)$ are written in column vector form, as x and X , respectively, where the values or entries are in natural order. The indexation for both $x(n)$ and $X(k)$ begins at the value 0 and ends at $L - 1$, for both n and k variables. Thus, the matrix F_L is formulated by defining each row in terms of k , and defining each column in terms of n .

There is a property which can be applied in the formulation for the DFT matrix, known as the *additive modulo L* $\langle p \rangle_L$, which represents the remainder of p after being divided by L :

$$\langle p \rangle_L = R\left(\frac{p}{L}\right), \quad (4.6)$$

where R denotes the remainder of the division $\frac{p}{L}$. If p is less than L , then $\langle p \rangle_L$ is equal to p . Also if p is equal to L , then $\langle p \rangle_L$ is equal to 0. For q defined as any integer in the set $Z^+ = 0, 1, 2, \dots$, the following equation can also be established:

$$\langle p + qL \rangle_L = R\left(\frac{p + qL}{L}\right). \quad (4.7)$$

The remainder R of the fraction $\frac{p+qL}{L}$ can be further expressed as:

$$R\left(\frac{p + qL}{L}\right) = R\left(\frac{p}{L}\right) + R\left(\frac{qL}{L}\right). \quad (4.8)$$

Careful observation in the above expression shows that the second term in the right-hand expression $R\left(\frac{qL}{L}\right)$ results in a remainder of 0, since the numerator qL is a multiple of the denominator L ; that is, qL is divisible by L , resulting in a remainder of 0. Hence, the following is obtained:

$$\langle p + qL \rangle_L = \langle p \rangle_L. \quad (4.9)$$

For example, $\langle -1 \rangle_7$ and $\langle 9 \rangle_4$ can be respectively expressed as follows:

$$\langle -1 \rangle_7 = \langle -1 + 7 \rangle_7 = \langle 6 \rangle_7 = 6. \quad (4.10)$$

$$\langle 9 \rangle_4 = 1. \quad (4.11)$$

The matrix F_L can be further simplified when applying the *additive modulo L* property to each element of the matrix:

$$F_L = \left[W_L^{\langle kn \rangle_L} \right]_{k,n=0,1,\dots,L-1}. \quad (4.12)$$

4.2.1 Example Formulation of DFT operation, for length L

The DFT operation for $L = 4$ can be expressed through the following summation operation, in terms of the variable n :

$$X(k) = \sum_{n=0}^3 x(n)W_4^{kn} = x(0) + x(1)W_4^k + x(2)W_4^{2k} + x(3)W_4^{3k}. \quad (4.13)$$

By further evaluating the expression defined for $X(k)$ for each value of $k = 0, 1, 2, 3$, the following formulations are derived:

$$X(0) = x(n)W_4^{kn} = x(0) + x(1) + x(2) + x(3); \quad (4.14)$$

$$X(1) = x(0) + x(1)W_4^1 + x(2)W_4^2 + x(3)W_4^3; \quad (4.15)$$

$$X(2) = x(0) + x(1)W_4^2 + x(2)W_4^4 + x(3)W_4^6; \quad (4.16)$$

$$X(3) = x(0) + x(1)W_4^3 + x(2)W_4^6 + x(3)W_4^9; \quad (4.17)$$

Hence, the DFT operation can be expressed in the matrix-vector form, $X = F_4x$, as follows:

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & W_4 & W_4^2 & W_4^3 \\ 1 & W_4^2 & W_4^4 & W_4^6 \\ 1 & W_4^3 & W_4^6 & W_4^9 \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{bmatrix} \quad (4.18)$$

where the DFT matrix F_4 is defined as:

$$F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & W_4 & W_4^2 & W_4^3 \\ 1 & W_4^2 & W_4^4 & W_4^6 \\ 1 & W_4^3 & W_4^6 & W_4^9 \end{bmatrix}. \quad (4.19)$$

By applying the *additive modulo L* property for $L = 4$, the matrix F_4 can be further simplified. For example, considering the element $W_4^{2*3} = W_4^6$ in row $k = 2$ and column $n = 3$, when applying the *additive modulo 4* property, we obtain $W_4^{(6)_4} = W_4^2$. Hence the DFT matrix F_4

may be finally expressed as:

$$F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & W_4 & W_4^2 & W_4^3 \\ 1 & W_4^2 & 1 & W_4^2 \\ 1 & W_4^3 & W_4^2 & W_4 \end{bmatrix}. \quad (4.20)$$

The DFT operation can be finally expressed in the simplified matrix-vector form, $X = F_4 x$, as follows:

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & W_4 & W_4^2 & W_4^3 \\ 1 & W_4^2 & 1 & W_4^2 \\ 1 & W_4^3 & W_4^2 & W_4 \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{bmatrix}. \quad (4.21)$$

4.3 Kronecker Product Algebra Definition

As mentioned previously, the Kronecker product is a very useful operation, and it is commonly used as a tool to model unitary and linear transforms, in particular, the DFT. In this section, the basic Kronecker product of two matrices is defined. In addition, some basic properties of the Kronecker product operation are described, such as for vector processing and parallel processing operations. The DFT matrix is further described as a composition of sparse matrices, where the Kronecker product is used to express some of them. The advantage of having these sparse matrices as factors which decompose the DFT matrix F_L , is that efficient implementations can be obtained on various computational architectures.

The Kronecker product operation uses the notation \otimes . Considering a matrix A , of size $R \times R$, and a matrix B , of size $S \times S$, the Kronecker product operation $A \otimes B$ is defined as follows:

$$C = A \otimes B = [a_{kl} B]_{k,l=0,1,\dots,R-1}, \quad (4.22)$$

producing a new matrix C of size $RS \times RS$. It is important to highlight that the definition of the Kronecker product can be generally applied to matrices of any dimensions.

Let A and B be square matrices of size 3×3 , which are defined as follows:

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}, B = \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{bmatrix}. \quad (4.23)$$

When applying the Kronecker product operation $A \otimes B$, the matrix C of size 9×9 is produced as follows:

$$C = A \otimes B = [a_{kl}B]_{k,l=0,1,\dots,R-1} = \begin{bmatrix} a_{00}B & a_{01}B & a_{02}B \\ a_{10}B & a_{11}B & a_{12}B \\ a_{20}B & a_{21}B & a_{22}B \end{bmatrix}. \quad (4.24)$$

The matrix C can also be expressed as:

$$C = A \otimes B = [a_{kl}B]_{k,l=0,1,\dots,R-1} \quad (4.25)$$

$$C = \begin{bmatrix} a_{00}b_{00} & a_{00}b_{01} & a_{00}b_{02} & a_{01}b_{00} & a_{01}b_{01} & a_{01}b_{02} & a_{02}b_{00} & a_{02}b_{01} & a_{02}b_{02} \\ a_{00}b_{10} & a_{00}b_{11} & a_{00}b_{12} & a_{01}b_{10} & a_{01}b_{11} & a_{01}b_{12} & a_{02}b_{10} & a_{02}b_{11} & a_{02}b_{12} \\ a_{00}b_{20} & a_{00}b_{21} & a_{00}b_{22} & a_{01}b_{20} & a_{01}b_{21} & a_{01}b_{22} & a_{02}b_{20} & a_{02}b_{21} & a_{02}b_{22} \\ a_{10}b_{00} & a_{10}b_{01} & a_{10}b_{02} & a_{11}b_{00} & a_{11}b_{01} & a_{11}b_{02} & a_{12}b_{00} & a_{12}b_{01} & a_{12}b_{02} \\ a_{10}b_{10} & a_{10}b_{11} & a_{10}b_{12} & a_{11}b_{10} & a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{10} & a_{12}b_{11} & a_{12}b_{12} \\ a_{10}b_{20} & a_{10}b_{21} & a_{10}b_{22} & a_{11}b_{20} & a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{20} & a_{12}b_{21} & a_{12}b_{22} \\ a_{20}b_{00} & a_{20}b_{01} & a_{20}b_{02} & a_{21}b_{00} & a_{21}b_{01} & a_{21}b_{02} & a_{22}b_{00} & a_{22}b_{01} & a_{22}b_{02} \\ a_{20}b_{10} & a_{20}b_{11} & a_{20}b_{12} & a_{21}b_{10} & a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{10} & a_{22}b_{11} & a_{22}b_{12} \\ a_{20}b_{20} & a_{20}b_{21} & a_{20}b_{22} & a_{21}b_{20} & a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{20} & a_{22}b_{21} & a_{22}b_{22} \end{bmatrix} \quad (4.26)$$

4.3.1 Special Properties of Kronecker Products

In this section a series of basic Kronecker properties that are often encountered in the formulation and implementation of a variety of operations, are defined. Let A , B , and C be defined as arbitrary matrices. First, the Kronecker product operation is associative, as demonstrated as follows:

$$A \otimes (B \otimes C) = (A \otimes B) \otimes C. \quad (4.27)$$

The Kronecker product operation is also distributive, as presented:

$$A \otimes (B \cdot C) = (A \otimes B) \cdot (A \otimes C). \quad (4.28)$$

The following special property also holds for the Kronecker product operation:

$$(A \otimes B) \cdot (C \otimes D) = AC \otimes BD. \quad (4.29)$$

A property of the Kronecker product operation, involving transposition operation denoted as T is presented as follows:

$$(A \otimes B)^T = A^T \otimes B^T. \quad (4.30)$$

Note that the Kronecker product operation is not commutative; that is, $(A \otimes B) \neq (B \otimes A)$.

4.4 Using Kronecker Product for Parallel Operation

In this section, the usage of Kronecker product formulation for implementing parallel operations is demonstrated in details. Let the matrix A be equal to the 4×4 identity matrix I_4 , which is defined as:

$$A = I_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad (4.31)$$

and let B be defined as the 2×2 DFT matrix of order 2, defined as follows:

$$B = F_2 = \begin{bmatrix} 1 & 0 \\ 0 & W_2 \end{bmatrix}. \quad (4.32)$$

The term W_2 can be expressed as:

$$W_2 = e^{\frac{-j2\pi}{2}} = \cos(\pi) - j\sin(\pi). \quad (4.33)$$

Since $\sin(\pi) = 0$, the above formulation reduces to:

$$W_2 = e^{\frac{-j2\pi}{2}} = \cos(\pi) = -1. \quad (4.34)$$

Thus the matrix $B = F_2$ may be reformulated as follows:

$$B = F_2 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}. \quad (4.35)$$

By computing the Kronecker product $C = A \otimes B = I_4 \otimes F_2$ the following 8×8 matrix is obtained:

$$C = I_4 \otimes F_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}; \quad (4.36)$$

$$C = I_4 \otimes F_2 = \begin{bmatrix} F_2 & 0 & 0 & 0 \\ 0 & F_2 & 0 & 0 \\ 0 & 0 & F_2 & 0 \\ 0 & 0 & 0 & F_2 \end{bmatrix}; \quad (4.37)$$

$$C = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix}. \quad (4.38)$$

In general, the expression $I_R \otimes F_S$ can be seen as a parallel operation since the submatrices F_S are the nonzero elements that appear along the diagonal of the resulting matrix, which is a sparse matrix. A matrix-vector multiplication operation can be defined, in terms of the sparse matrix $I_R \otimes F_S$ as follows:

$$b = (I_R \otimes F_S) a, \quad (4.39)$$

where b and a are column vectors, each of length or order $R \cdot S$. As an example, let $R = 4$ and $S = 2$. Considering a column vector b and a , each of length 8, the matrix-vector multiplication

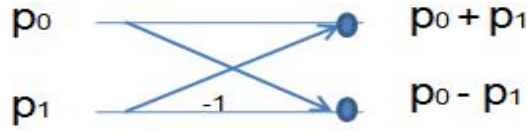


Figure 4.1: Scalar Butterfly Operation

operation $b = (I_4 \otimes F_2) a$ is formulated as follows:

$$\begin{bmatrix} b(0) \\ b(1) \\ b(2) \\ b(3) \\ b(4) \\ b(5) \\ b(6) \\ b(7) \end{bmatrix} = \begin{bmatrix} F_2 & 0 & 0 & 0 \\ 0 & F_2 & 0 & 0 \\ 0 & 0 & F_2 & 0 \\ 0 & 0 & 0 & F_2 \end{bmatrix} \begin{bmatrix} a(0) \\ a(1) \\ a(2) \\ a(3) \\ a(4) \\ a(5) \\ a(6) \\ a(7) \end{bmatrix}. \quad (4.40)$$

The column vector a can be divided into 4 sections, or subvectors, each consisting of two elements. Let a_0^* , a_1^* , a_2^* , and a_3^* be the 4 subvectors, that are defined as:

$$a_0^* = \begin{bmatrix} a(0) \\ a(1) \end{bmatrix}; a_1^* = \begin{bmatrix} a(2) \\ a(3) \end{bmatrix}; a_2^* = \begin{bmatrix} a(4) \\ a(5) \end{bmatrix}; a_3^* = \begin{bmatrix} a(6) \\ a(7) \end{bmatrix}. \quad (4.41)$$

The matrix-vector multiplication operation $b = (I_4 \otimes F_2) a$ can then be expressed as:

$$\begin{bmatrix} F_2 & 0 & 0 & 0 \\ 0 & F_2 & 0 & 0 \\ 0 & 0 & F_2 & 0 \\ 0 & 0 & 0 & F_2 \end{bmatrix} \begin{bmatrix} a_0^* \\ a_1^* \\ a_2^* \\ a_3^* \end{bmatrix} = \begin{bmatrix} F_2 \cdot a_0^* \\ F_2 \cdot a_1^* \\ F_2 \cdot a_2^* \\ F_2 \cdot a_3^* \end{bmatrix} \quad (4.42)$$

The operation $F_2 \cdot p$, where p is a vector of length 2, is known as the scalar butterfly operation, and is described as follows:

$$F_2 \cdot p = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \end{bmatrix} = \begin{bmatrix} p_0 + p_1 \\ p_0 - p_1 \end{bmatrix}. \quad (4.43)$$

Figure 4.1 presents a more conceptual view of how the scalar butterfly operation works: Hence

$b = (I_4 \otimes F_2) a$ becomes:

$$\begin{bmatrix} b(0) \\ b(1) \\ b(2) \\ b(3) \\ b(4) \\ b(5) \\ b(6) \\ b(7) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} a(0) \\ a(1) \\ a(2) \\ a(3) \\ a(4) \\ a(5) \\ a(6) \\ a(7) \end{bmatrix} = \begin{bmatrix} a(0) + a(1) \\ a(0) - a(1) \\ a(2) + a(3) \\ a(2) - a(3) \\ a(4) + a(5) \\ a(4) - a(5) \\ a(6) + a(7) \\ a(6) - a(7) \end{bmatrix}. \quad (4.44)$$

This clearly demonstrates that the operation $b = (I_4 \otimes F_2) a$ can be performed by computing 4 simultaneous FFTs, each of length 2. In general, the matrix-vector multiplication operation $b = (I_R \otimes F_S) a$ can be computed on a computer architecture with R processors, in which each processor is loaded with a subvector of the input vector a , consisting of S elements, and computes the FFT of length S .

4.5 Using Kronecker Product for Vector Processing

The Kronecker product is characterized to be a non-commutative operation; hence $I_R \otimes F_S \neq F_S \otimes I_R$. However, the Kronecker product $F_S \otimes I_R$ introduces additional special properties, in which this expression favors an architecture capable of conducting vector processing computations. Let $S = 2$ and $R = 4$. The sparse matrix $C = F_2 \otimes I_4$ can thus be defined as follows:

$$C = F_2 \otimes I_4 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}; \quad (4.45)$$

$$C = F_2 \otimes I_4 = \begin{bmatrix} I_4 & I_4 \\ I_4 & -I_4 \end{bmatrix}; \quad (4.46)$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \end{bmatrix}. \quad (4.47)$$

In this case, the operation $b = (F_S \otimes I_R) a$ can be computed at a vector level, instead of at a scalar level. The following example of the operation $b = (F_2 \otimes I_4) a$ is considered, which can be defined as follows:

$$\begin{bmatrix} b(0) \\ b(1) \\ b(2) \\ b(3) \\ b(4) \\ b(5) \\ b(6) \\ b(7) \end{bmatrix} = \begin{bmatrix} I_4 & I_4 \\ I_4 & -I_4 \end{bmatrix} \begin{bmatrix} a(0) \\ a(1) \\ a(2) \\ a(3) \\ a(4) \\ a(5) \\ a(6) \\ a(7) \end{bmatrix}. \quad (4.48)$$

The above matrix-vector multiplication operation divides the column vector a into the following two subvectors:

$$\begin{bmatrix} a(0) \\ a(1) \\ a(2) \\ a(3) \end{bmatrix}; \begin{bmatrix} a(4) \\ a(5) \\ a(6) \\ a(7) \end{bmatrix}. \quad (4.49)$$

The matrix-vector multiplication operation can be viewed as the following submatrix-vector segment multiplication:

$$\begin{bmatrix} b(0) \\ b(1) \\ b(2) \\ b(3) \\ b(4) \\ b(5) \\ b(6) \\ b(7) \end{bmatrix} = \begin{bmatrix} I_4 \\ I_4 \end{bmatrix} \begin{bmatrix} a(0) \\ a(1) \\ a(2) \\ a(3) \\ a(0) \\ a(1) \\ a(2) \\ a(3) \end{bmatrix} + \begin{bmatrix} I_4 \\ -I_4 \end{bmatrix} \begin{bmatrix} a(4) \\ a(5) \\ a(6) \\ a(7) \\ a(4) \\ a(5) \\ a(6) \\ a(7) \end{bmatrix} \quad (4.50)$$

According to the above formulation, the vector segments are multiplied by the identity matrix I_4 . This introduces the following special property regarding identity matrices: let I_N be the matrix of size $N \times N$, where the only non-zero elements equal to 1, lie along the diagonal of the matrix. Let d be a column vector, consisting of N elements. The matrix-vector multiplication operation $I_N \cdot d$ can be defined as $I_N \cdot d = d$. Letting $N = 4$, matrix-vector multiplication operation $I_4 \cdot d$ is demonstrated as follows:

$$I_4 \cdot d = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} d(0) \\ d(1) \\ d(2) \\ d(3) \end{bmatrix} = \begin{bmatrix} d(0) \\ d(1) \\ d(2) \\ d(3) \end{bmatrix}. \quad (4.51)$$

Hence, in general, $I_N \cdot d = d$, revealing the following final result with respect to the matrix-vector multiplication operation $b = (F_S \otimes I_R) a$:

$$\begin{bmatrix} b(0) \\ b(1) \\ b(2) \\ b(3) \\ b(4) \\ b(5) \\ b(6) \\ b(7) \end{bmatrix} = \begin{bmatrix} a(0) \\ a(1) \\ a(2) \\ a(3) \\ a(0) \\ a(1) \\ a(2) \\ a(3) \end{bmatrix} + \begin{bmatrix} a(4) \\ a(5) \\ a(6) \\ a(7) \\ a(4) \\ a(5) \\ a(6) \\ a(7) \end{bmatrix} - \begin{bmatrix} a(4) \\ a(5) \\ a(6) \\ a(7) \\ a(4) \\ a(5) \\ a(6) \\ a(7) \end{bmatrix}. \quad (4.52)$$

Generalizing, the matrix-vector multiplication operation $b = (F_S \otimes I_R) a$ can be computed on a computer architecture, capable of vector processing operations. The input vector a is divided into S vector segments, each consisting of R elements.

As demonstrated in this section, the sparse matrices, resulting from the expressions $T_R \otimes F_S$

and $F_S \otimes I_R$ are very common in the Kronecker product formulations of FFT algorithms. This section showcased how these expressions serve as instruments in designing and obtaining efficient FFT implementations, which will be shown in the next section.

4.6 Formulating FFTs Using Kronecker Product

In this section, the Kronecker product is used in order to obtain efficient mathematical formulations of FFT algorithms. It will be shown how the DFT matrix can actually be decomposed into a sequence of sparse matrices, all of the same dimension or order. Initially, the order of the DFT matrix was set to L . It is desirable that L be a highly composite number, and of the form $L = 2^m$ for the Kronecker formulation. A composite number is characterized to be divisible by another positive integer, other than itself or one.

Let $L = RS$, where R and S may, or may not be composite numbers. As defined previously, in order to compute the DFT of a L -point discrete signal $x(n)$, the following operation needs to be performed:

$$X(k) = \sum_{n=0}^{L-1} x(n) e^{(-j2\pi nk/L)}; k = 0, 1, \dots, L-1; j = \sqrt{-1}. \quad (4.53)$$

In addition, it was found that the DFT operation could also be expressed in the matrix-vector form $X = F_L x$. Now, the essence in deriving a Kronecker formulation for this operation is to determine how the DFT matrix F_L can be decomposed, when L is a composite number of the form RS . Furthermore, if either R or S is also a composite number, further decomposition of the DFT matrix can be achieved. As presented in [14], when L is a composite number of the form RS , and by rearranging the one-dimensional input vector $x(n)$ as a two-dimensional array, the DFT computation of a L -point discrete signal $x(n)$ can be expressed in Kronecker product form, as follows:

$$X = F_L x = (F_S \otimes I_R) T_{L,S} (I_S \otimes F_R) P_{L,S} x \quad (4.54)$$

Here, $T_{L,S}$ is a diagonal matrix of order L , called twiddle or phase factor, which is defined as follows:

$$T_{L,S} = \begin{bmatrix} I_R & 0 & 0 & 0 & 0 \\ 0 & D_{L,R} & 0 & 0 & 0 \\ 0 & 0 & D_{L,R}^2 & 0 & 0 \\ \vdots & \vdots & 0 & \vdots & \vdots \\ 0 & 0 & \dots & 0 & D_{L,R}^{S-1} \end{bmatrix}, \quad (4.55)$$

where $D_{L,R}$ is a diagonal submatrix element of $T_{L,S}$, where each of its non-zero elements is defined as follows:

$$D_{L,R} = [W_L^k]_{k=0,1,\dots,R-1}; W_L = e^{(-j\frac{2\pi}{L})}. \quad (4.56)$$

Note that the factor R should be greater than two, to form a minimum submatrix of size 2×2 . The matrix $P_{L,S}$ is a permutation matrix, also of order L , called the stride by S permutation matrix. This matrix induces a permutation operation, which is present in all Kronecker product formulation. Such matrix reorganizes the input data following a decimation procedure. For any integer L and S , the permutation matrix may be formulated as follows:

$$P_{L,S}(i,j) = \begin{cases} 1 & \text{if } i = j = L - 1 \\ 1 & \text{if } j = \langle i \times S \rangle_{L-1}; 0 \leq i < L - 1 \\ 0 & \text{otherwise} \end{cases} \quad (4.57)$$

It is important to note that the Kronecker formulation for the DFT operation can be used in an iterative manner; that is, if S or R is also a composite of the form DP , the process can be repeated, inserting the new formulation in (4.50).

To illustrate the equivalence of the DFT computation for $L = RS$, the case for $L = 4$ is considered. Let $L = 4 = 2 \cdot 2$, where $R = 2$ and $S = 2$. Then the DFT matrix expressed in Kronecker product form for $L = 4$ is defined as follows:

$$F_4 = (F_2 \otimes I_2) T_{4,2} (I_2 \otimes F_2) P_{4,2} \quad (4.58)$$

The matrix $T_{4,2}$ is expressed as follows:

$$T_{4,2} = \begin{bmatrix} I_2 & 0 \\ 0 & D_{4,2} \end{bmatrix}, \quad (4.59)$$

where the submatrix element $D_{4,2}$ can be expressed as:

$$D_{4,2} = [W_4^k]_{k=0,1} = \begin{bmatrix} 1 & 0 \\ 0 & W_4 \end{bmatrix}. \quad (4.60)$$

The permutation matrix $P_{4,2}$, derived from the formulation in (4.53), becomes:

$$P_{4,2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (4.61)$$

By formulating the sparse matrices $(F_2 \otimes I_2)$, $T_{4,2}$, and $(I_2 \otimes F_2)$, the DFT matrix is decomposed as follows:

$$F_4 = \begin{bmatrix} I_2 & I_2 \\ I_2 & -I_2 \end{bmatrix} \cdot \begin{bmatrix} I_2 & 0 \\ 0 & D_{4,2} \end{bmatrix} \cdot \begin{bmatrix} F_2 & 0 \\ 0 & F_2 \end{bmatrix} \cdot P_{4,2}. \quad (4.62)$$

When performing the matrix multiplication of the matrices $(F_2 \otimes I_2)$ and $T_{4,2}$ in (4.58), the following is obtained:

$$F_4 = \begin{bmatrix} I_2 & D_{4,2} \\ I_2 & -D_{4,2} \end{bmatrix} \cdot \begin{bmatrix} F_2 & 0 \\ 0 & F_2 \end{bmatrix} \cdot P_{4,2}. \quad (4.63)$$

Additional matrix multiplication leads to the following formulation:

$$F_4 = \begin{bmatrix} F_2 & D_{4,2}F_2 \\ F_2 & -D_{4,2}F_2 \end{bmatrix} \cdot P_{4,2}; \quad (4.64)$$

The submatrices $D_{4,2}F_2$ and $-D_{4,2}F_2$ are expressed as follows:

$$D_{4,2}F_2 = \begin{bmatrix} 1 & 0 \\ 0 & W_4 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ W_4 & -W_4 \end{bmatrix}; \quad (4.65)$$

$$-D_{4,2}F_2 = \begin{bmatrix} -1 & 0 \\ 0 & -W_4 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} -1 & -1 \\ -W_4 & W_4 \end{bmatrix}; \quad (4.66)$$

Expressing the remaining two matrices in order of 4, the following is obtained:

$$F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & W_4 & -W_4 \\ 1 & 1 & 1 & -1 \\ 1 & -1 & -W_4 & W_4 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (4.67)$$

The permutation matrix $P_{4,2}$ reorders the columns of the other matrix in the above formulation, such that the new and final matrix is formed by placing the columns 0, 2, 1, and 3 in this order. By substituting $W_4 = e^{\frac{-j2\pi}{4}} = -j$, the following matrix is obtained, resulting in the DFT matrix of order 4:

$$F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & +j \\ 1 & -1 & 1 & -1 \\ 1 & +j & -1 & -j \end{bmatrix} \quad (4.68)$$

Hence, in general form, the Kronecker formulations are shown to provide an efficient decomposition of the DFT matrix into sparse matrices, when the order L is a composite number of the form RS . In addition, the DFT matrix F_L is symmetric; that is, $F_L = F_L^T$, where T denotes the transposition operation. Hence, by applying the transposition operations on both sides of the DFT matrix F_L in equation (4.50), the formulation becomes:

$$F_L = P_{L,S}^{-1} (I_S \otimes F_R) T_{L,S} (F_S \otimes I_R). \quad (4.69)$$

4.7 Parallel Computational Architectures for DFT Multi-beamforming

There are basic computational architectures, for which the Kronecker product operations has shown to be very effective. In this subsection, a parallel computational architecture for FFT is discussed.

For a variety of applications that require data analysis, such as bioacoustics signal analysis, data is usually analyzed in the frequency domain, by applying fast Fourier transform to the data in the time-domain. Extensive computational effort and significant amount of processing time is inevitable when the amount of data to be processed is large. For a considerable large data size of L points, L be a power of two, a large DFT matrix of size $L \times L$ would be needed in order to transform the data in the frequency domain. This, in turn, leads to increase in processing time. On the other hand, by using Kronecker product, the large data may be divided into M modules, each of size N , such that the matrix F_N is applied to each module, and $L = M \times N$. The multi-beamforming matrix B becomes the DFT transform matrix F_L when the number of steering angles M is equal to the number of sensors L in the linear array. However, the DFT multi-beamforming matrix may also be formulated using Kronecker product operation. The Kronecker product operation $A \otimes B$, as defined in (4.3) considering a matrix A , of size $R \times R$, and a matrix B , of is defined as follows:

$$C = A \otimes B = [a_{kl} B]_{k,l=0,1,\dots,R-1}, \quad (4.70)$$

producing a new matrix C of size $RS \times RS$. Using the Kronecker product definition, the multi-beamforming matrix may be defined as follows, according to [1]

$$MB = (U_M^T \otimes I_N) (I_M \otimes F_N), \quad (4.71)$$

where U_M is a column vector of M ones, defined as follows:

$$U_M = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}. \quad (4.72)$$

The operation T denotes the transpose of the column vector U_M , resulting in the new row vector U_M^T :

$$U_M^T = \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix}. \quad (4.73)$$

Through this formulation the data is divided into M modules, each of size N , such that the matrix F_N is applied to each module, and $L = M \times N$. The *parallel Fourier factor* uses the Kronecker product operation in order apply the F_N matrix, to each of the M modules, as defined as follows:

$$(I_M \otimes F_N) = \begin{bmatrix} F_N & 0 & \cdots & 0 \\ 0 & F_N & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & F_N \end{bmatrix}, \quad (4.74)$$

where I_M is the $M \times M$ identity matrix. The term $(U_M^T \otimes I_N)$ can be expressed as:

$$(U_M^T \otimes I_N) = \begin{bmatrix} I_N & I_N & \cdots & I_N \end{bmatrix}. \quad (4.75)$$

This operation simply sums the M transformed modules, each of size N . In addition, if the number of sensors N in each module can be expressed as a composite number of the form $N = RS$, where R is greater than 2, then the DFT matrix F_N for each of the M modules can be expressed as ([2]):

$$F_N = (F_S \otimes I_R) T_{N,S} (I_S \otimes F_R) P_{N,S}, \quad (4.76)$$

where $T_{N,S}$ is the twiddle or phase matrix, and $P_{N,S}$ is the permutation matrix that reorders the data. Hence the multi-beamforming matrix may be reformulated, as follows:

$$MB = (U_M^T \otimes I_N) (I_M \otimes ((F_S \otimes I_R) T_{N,S} (I_S \otimes F_R) P_{N,S})). \quad (4.77)$$

Now, let matrix $A = (F_S \otimes I_R) T_{N,S}$, and matrix $B = (I_S \otimes F_R) P_{N,S}$, obtaining the following expression:

$$MB = (U_M^T \otimes I_N) (I_M \otimes (A \cdot B)). \quad (4.78)$$

By applying the distributive property to the term $(I_M \otimes (A \cdot B))$, the following formulation is derived:

$$MB = (U_M^T \otimes I_N) (I_M \otimes A) (I_M \otimes B); \quad (4.79)$$

$$MB = (U_M^T \otimes I_N) (I_M \otimes ((F_S \otimes I_R) T_{N,S})) (I_M \otimes ((I_S \otimes F_R) P_{N,S})). \quad (4.80)$$

Chapter 5

Parallel Programming with pMATLAB

In chapter 3, it was shown how DFT beamforming algorithms are developed in order to process signals arriving from sensor arrays consisting of uniform linear configurations, when the number of sensors is equal to the number of steering. It was then shown in chapter 4 how the DFT beamforming algorithms can be efficiently implemented by using kronecker product formulations, in order to provide general expressions for the beamforming operations. Here, a new parallel programming modeling environment, named pMATLAB [16], is presented, which has been utilized to study the computational performance of parallel implementation techniques, in a multi-core environment.

5.1 Parallel Programming Concept

The term codes is often used to refer to complete programs with data, that are used to implement a desired computational operation or function. The design of each code follows a certain program and data structure, which can determine the appropriate architecture that is suitable for its execution. If the program structure of a code permits it to be totally or partially divided into M modules or segments, which can be executed simultaneously and independently, then parallel programming can be achieved. Each of the M modules or segments can be assigned to a processor, thus establishing a multi-core environment, where $N_P = M$ processors can perform computational operations at the same time.

5.1.1 Characteristics of a Parallel System Architecture

The basic, common model for a serial system can be depicted in **Figure 5.1**, where P_0 represents the single serial processor with its own local memory M_0 . However, the dependency

on computation of increasing amount of data is becoming more pronounced and evident for a wide variety of applications. This has lead to extensive research on exploiting parallelism, and developing parallel architecture systems, in order to obtain more speed and simplify complex, intensive computation. A basic, general parallel system architecture is shown in **Figure 5.2**, where N_P independent processors, each with its local own memory M_0 , are within the same architecture, which can perform computations simultaneously. This type of performance is also known as **parallel computation**. Two types of acceptable parallel computation as mentioned in [17] are defined as follows: **fast-memory parallel computation** and **slow-memory parallel computation**. Fast-memory parallel computation is one, in which the memory demands of each processor are met on time, and each processor is not affected by significant degradation of its individual uniprocessor performance. Slow-memory parallel computation occurs when memory delays are present in each individual processor. However, if high parallelism is present, then acceptable, overall performance can be achieved, relative to the number of processors in the architecture.

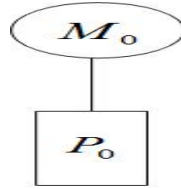


Figure 5.1: Single Serial Processor, *courtesy of Kuck, MIT Lincoln Lab.*

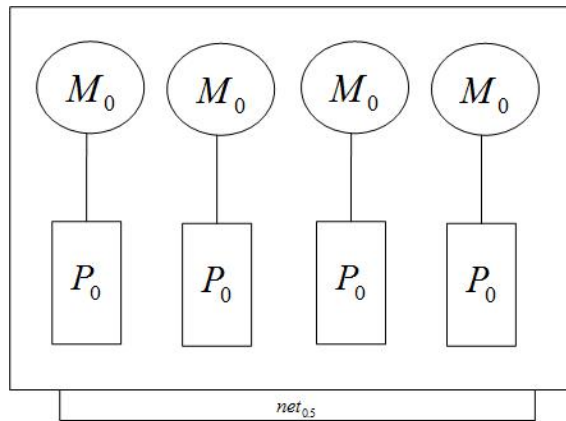


Figure 5.2: Serial Processors in Parallel, *courtesy of Kuck, MIT Lincoln Lab.*

5.2 Parallel Speedup Process and Amdahl's Law

In order to evaluate the parallel structure of a complete program, the total computational speedup serves as a parallel performance metric. Such metric is computed in a linear fashion,

in terms of the number of processors N_P :

$$S_{comp}(N_P) = \frac{T_{comp}(1)}{T_{comp}(N_P)} = N_P, \quad (5.1)$$

where $S_{comp}(N_P)$ represents the computational speed, $T_{comp}(1)$ the execution time of the program running on one processor, and $T_{comp}(N_P)$ the execution time with N_P processors. It is important to note that the number of processors is assumed to be in powers of two. As highlighted in [16], the speedup acquired by a parallel application is essential in measuring how effectively an application can take advantage of a computer system that possesses parallel capability. **Figure 5.3** presents the different types of linearity that are commonly encountered, resulting from the speedup obtained, in terms of the number of processors: As mentioned

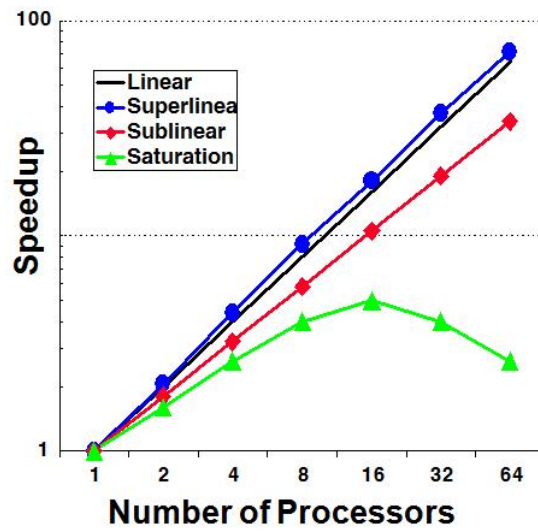


Figure 5.3: Parallel Speedup, courtesy of Kepner, MIT Lab.

in [16] and [17], the types of speedup linearity may be defined as follows: *linear*, *superlinear*, *sublinear*, and *saturated* speedup linearities. Linear speedup is the ideal type of speedup, where $S_{comp}(N_P) = N_P$. This type of speedup occurs when there is little or no communication between the processors.

The sublinear speedup shown in **Figure 5.3** is more typical than the ideal linear speedup, and is obtained when $S_{comp}(N_P) = \alpha N_P$, for $0 < \alpha < 1$. The value α represents a constant overhead due to communication delays, whose proportion remains constant as more processors are added. Thus the constant overhead is what prevents the application from achieving an ideal linear speedup. As mentioned in [16], if an application is highly parallel, then α can indicate that there may be unnecessary overhead that can be avoided.

On the other hand, saturated speedup is highly common, especially in the initial development of parallel programs. For this type of speedup curve, as shown in **Figure 5.3**, the speedup begins

to increase, as the number of processors increases, starting with a single processor. However, as the number of processors begin to increase even further, the curve begins to saturate at some low number of processors, being 16 a typical number, as shown in **Figure 5.3**. This point of the speedup curve is known as the *point of diminishing return*, which is an indicator that from this point on, the speedup will begin to degrade.

The superlinear speedup is the rarest case, where $S_{comp}(N_P) > \alpha N_P$. This type of speedup occurs in highly parallel programs, with fixed problem sizes. This means that the problem sizes do not grow, as the number of processors N_P increases. A superlinear speedup indicates that the program's performance is improving as N_P increases [16].

5.2.1 Amdahl's Law

The saturated speedup is related to Amdahl's law, which is used in parallel computing to predict the theoretical maximum speedup using multiple processors, before reaching the point of diminishing return. This law takes into consideration the time limit or communication overhead induced by the sequential or serial portion of the program; in other words, not every fraction of the parallel application or program is 100% parallel. Having this in mind, Amdahl's law is formulated as follows [16].

Assuming that the program application is not completely parallel, let the total amount of work W_{tot} that needs to be carried, be divided into two parts: one that can be done in parallel W_{par} , and a part W_{seq} that can only be accomplished by using a single, serial processor:

$$W_{tot} = W_{par} + W_{seq}; \quad (5.2)$$

The execution time $T_{comp}(N_P)$ is thus proportional to:

$$T_{comp}(N_P) \propto \frac{W_{par}}{N_P} + W_{seq} \quad (5.3)$$

Using the relationship between the execution time of a single processor $T_{comp}(1)$ and $T_{comp}(N_P)$ for multiple processors, the speedup expressed in (5.1) becomes:

$$S_{comp}(N_P) = \frac{W_{tot}}{\frac{W_{par}}{N_P} + W_{seq}}. \quad (5.4)$$

Normalizing with respect to W_{tot} , $S_{comp}(N_P)$ becomes

$$S_{comp}(N_P) = \frac{1}{\frac{w_{par}}{N_P} + w_{seq}}; w_{par} = \frac{W_{par}}{W_{tot}}; w_{seq} = \frac{W_{seq}}{W_{tot}}. \quad (5.5)$$

Now, when the number of processors N_P becomes very large, that is N_P approximates ∞ , the term $\frac{\omega_{par}}{N_P}$ approaches 0, reducing (5.5) to the following:

$$S_{comp}(N_P) = \frac{1}{\omega_{seq}} = \omega_{seq}^{-1}. \quad (5.6)$$

According to the above equation, as the number of processors N_P increases, the maximum speedup that can be achieved is ω_1^{-1} . In other words, if the fraction of the work needed to be done sequentially, that is, on a single serial processor, is made very small, then a linear speedup can be approached. Hence, this reveals the fundamental of Amdahl's Law: it is important to make very part of the code parallel. For instance, if ω_{seq} is large, indicating that more aspects of the code is sequential than parallel, then the speedup is not optimized. This also includes overheads encountered in communication, which can further degrade the speedup.

5.3 Description

Parallel programming with pMATLAB provides a series of advantages, which help overcome important issues, such as communication overhead among the processors. First, pMATLAB offers high level parallel data structures and functions as part of its tools for creating a simulated, multicore environment, in a Pure MATLAB implementation manner. **Figure 5.4** demonstrates the MATLAB environment in which the parallel program applications are developed in pMATLAB.

This permits parallel functionality to be added to serial programs already implemented in pure MATLAB, without requiring mayor modifications. Distributed matrices/vectors are created by using maps that help distribute the data among the N_P simulated processors, which is described in more details in the next section. In addition, pMATLAB uses MatlabMPI to perform message passing among the processors. MatlabMPI offers point-to-point communication, in which a I/O file is provided through the common load and save functions of MATLAB. This helps take care of complicated buffer packing/unpacking problem and communication delays caused by such. In addition, the point-to-point communication that provides pMATLAB, as shown in [18] has achieved the typical superlinear speedup on fixed problem, as shown in **Figure 5.5**, and the typical linear scenario for scaled problems.

Hence pMATLAB is a powerful tool that not only exploits the level of parallelism in programming codes, but can easily add parallel functions to existing serial programs, avoiding bottlenecks which can lead to early saturation.

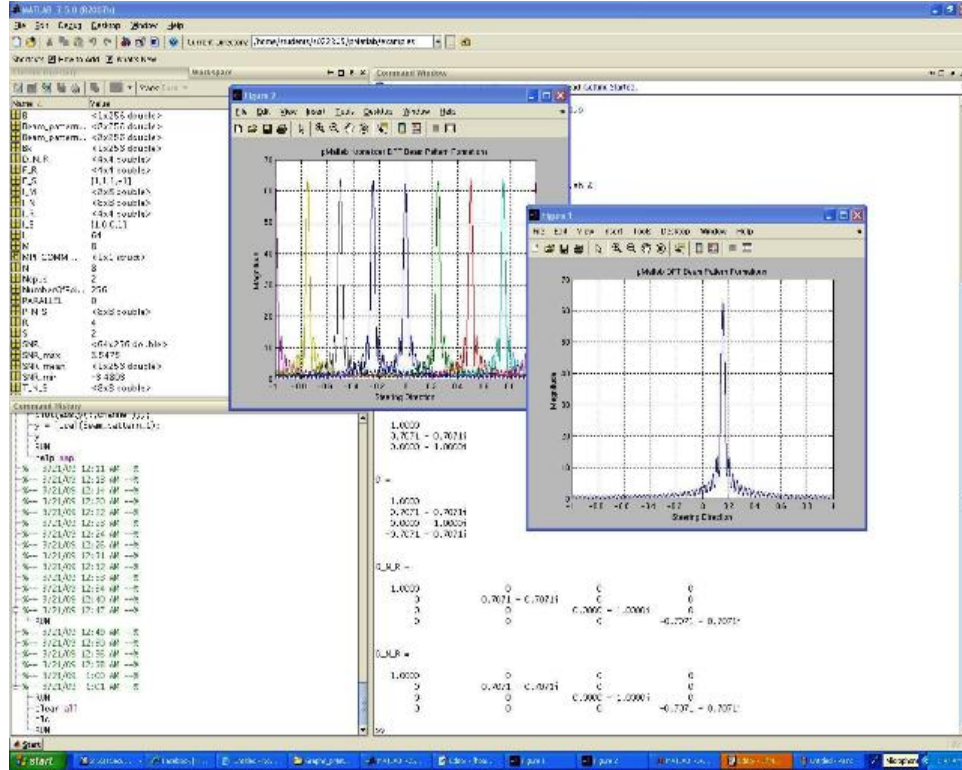


Figure 5.4: Programming Environment for pMATLAB

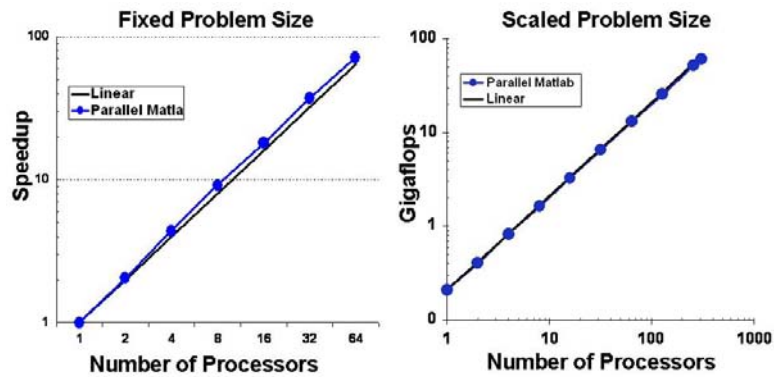


Figure 5.5: Speedup achieved with pMATLAB, courtesy of Kepner, MIT Lincoln Lab.

5.3.1 Data Mapping and Distribution

Programming in pMATLAB allows the data to be constructed as distributed arrays or matrices, called *dmat* which permits the data to be allocated among the N_P processors. This is achieved, by using the function *map* in the program application, which is defined in the following general format, as presented in [19]:

$$p = \text{map}(\text{GRID_SPEC}, \text{DIST_SPEC}, \text{PROC_LIST}). \quad (5.7)$$

This function creates a *map* object, which is used as input for the *dmat* constructor. The parameter *GRID_SPEC* specifies how the dimensions of the data will be distributed among the N_P processors. *DIST_SPEC* specifies the type of data distribution, which can be: block, cyclic, or block-cyclic. The parameter *PROC_LIST* is an array of processors, in terms of ranks, on which the data will be distributed. The processor ranks are noted as $0, 1, \dots, N_P - 1$.

Below, the following example notations or formats of the *map* are commonly used for the programs created in pMATLAB [16]. The second input parameter of the function *map* $\{\}$ indicates that the data is distributed among the N_P processors, defined in terms of ranks from the first processor 0 to the processor $N_P - 1$, in terms of blocks. The concept of block distribution among the processors is illustrated as follows.

For the first format shown below, a one-dimensional map is constructed, which maps an array or matrix along the first dimension:

$$\text{map}([N_P \ 1], \{\}, 0 : N_P - 1) \quad (5.8)$$

That is, the block distribution of the matrix consists of dividing the rows into N_P blocks. Assuming that $N_P = 4$, and the data is defined to be a 12×12 matrix, the rows of the data is distributed in 4 blocks, each block containing 3 rows. **Figure 5.6** illustrates this example.

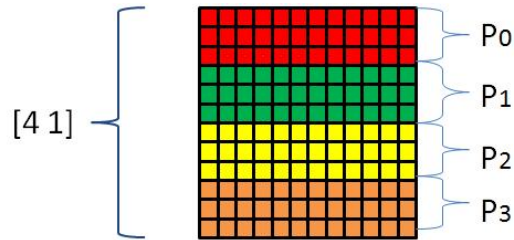


Figure 5.6: Block Distribution Example for First Dimension

The second format commonly used constructs a one-dimensional map, which maps an array or matrix along the second dimension, shown as follows:

$$\text{map}([1 \ N_P], \{\}, 0 : N_P - 1). \quad (5.9)$$

Using this format, the block distribution of the matrix consists of dividing the columns into N_P blocks. Assuming that $N_P = 4$, and the data is defined to be a 12×12 matrix, the columns of the data is distributed in 4 blocks, each block containing 3 columns. **Figure 5.7** illustrates this example.

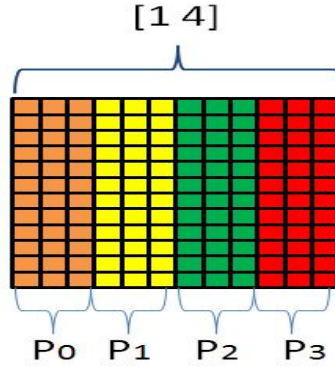


Figure 5.7: Block Distribution Example for Second Dimension

The third format commonly used constructs a two-dimensional map, which maps an array or matrix along the first and second dimensions, shown as follows:

$$\text{map}([N_1 N_2], \{\}, 0 : N_P - 1), \quad (5.10)$$

Using this format, the block distribution of the matrix consists of breaking up the first and second dimensions (rows and columns) between the processors, into N_1 blocks along the first dimension, and N_2 blocks along the second dimension. Hence, a total of $N_1 \cdot N_2$ distributed blocks are created. A total of $N_1 \cdot N_2$ processors would then be needed in order to distribute each of the blocks. Letting $N_1 = 4$ and $N_2 = 4$, the assumed 12×12 data matrix, becomes a 4×4 distributed block matrix; that is, each row and column is composed of 4 blocks, leading to a total of 16 blocks. Hence the number of processors N_P would need to be 16, in order to process each block. **Figure 5.8** illustrates this example. The form in which distributed arrays

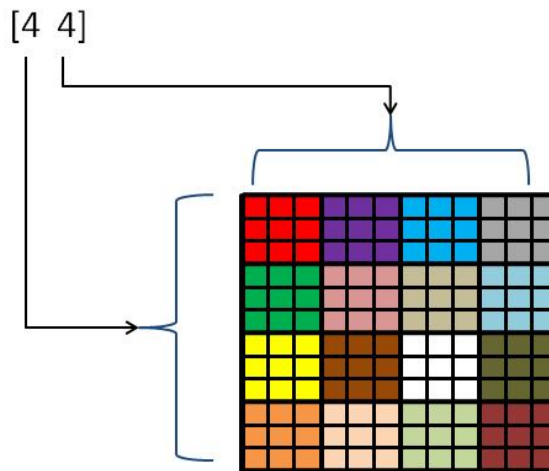


Figure 5.8: 2-D Block Distribution Example

or matrices are created in a typical pMATLAB code developed in a MATLAB environment, is shown in **Figure 5.9**. The basic steps in creating distributed arrays simply consists of first

defining the type of mapping, or the manner in which the data is to be distributed. As shown previously, this includes specifying the number of distributing blocks and to which processors the data will be distributed. In **Figure 5.9**, *mapA* and *mapB* are the two maps defined for the program application. *MapA* decomposes or divides the columns of the data in two blocks, distributing each block to the processors identified with ranks 0 and 1. *MapB* decomposes or divides the columns of the data in two blocks, distributing each block to the processors identified with ranks 2 and 3. *A* and *B* are simply the data matrices that will be processed or manipulated in the program. Matrix *A* and Matrix *B* are created, using the typical MATLAB functions *rand* and *zeros*. *rand* creates an arbitrary $m \times n$ matrix, and the latter a matrix of the same dimension whose elements are all 0.

```
mapA = map([1 2], ... % Specifies that cols be dist. over 2 procs
           {}, ...    % Specifies distribution: defaults to block
           [0:1]);    % Specifies processors for distribution
mapB = map([1 2], {}, [2:3]);

A = rand(m,n, mapA); % Create random distributed matrix
B = zeros(m,n, mapB); % Create empty distributed matrix
B(:, :) = A;         % Copy and redistribute data from A to B.
```

Figure 5.9: Data Mapping Example in pMATLAB

As observed in **Figure 5.9**, pMATLAB introduces the concept of mapping, by adding as an additional input parameter to these functions, *MapA* and *MapB* for creating the distributed matrices *A* and *B*, respectively. In other words, the contents of the matrix *A* is distributed among the processors 0 and 1, while the contents of *B* is distributed among the processors 2 and 3 (see **Figure 5.10**). The command line *B(:, :) = A* simply redistributes the matrix *A* onto the processors 2 and 3, defined in the mapping for the distributed matrix *B*.

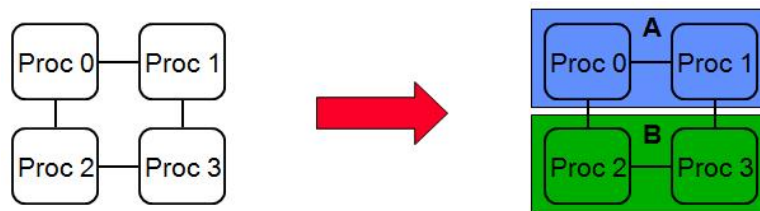
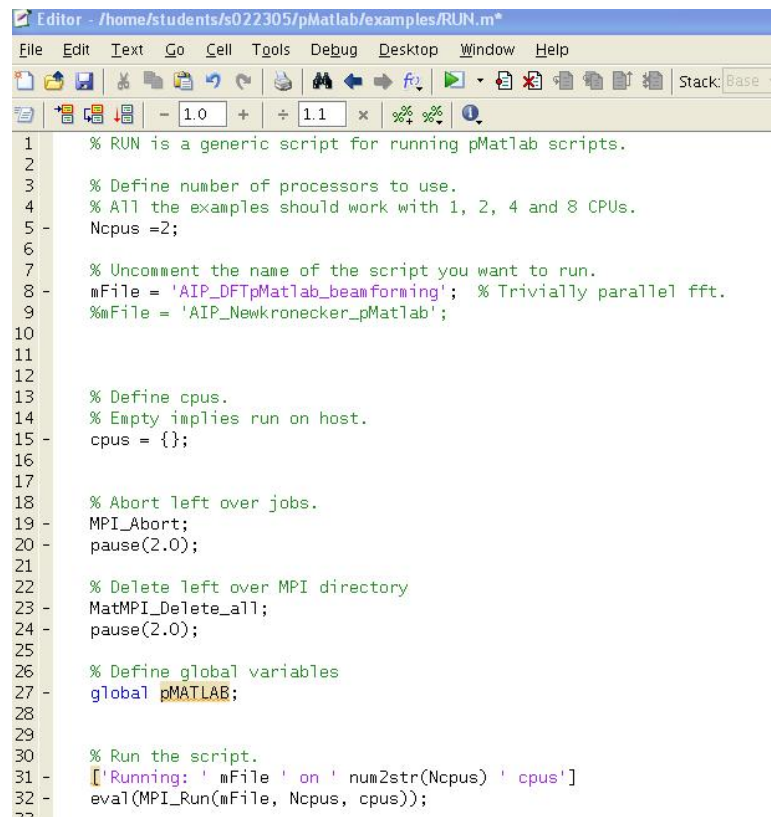


Figure 5.10: Data Distribution Example Among Processors, *courtesy of Kepner, MIT Lab.*

5.3.2 Parallel Execution in pMATLAB

In the previous section, it was shown how distributed array objects *dmats* can be created and distributed among N_P processors, by simply defining the type of mapping. Here, an example of executing the DFT Multi-beamforming algorithm implemented using pMATLAB, is demonstrated.

The pMATLAB tool provides parallel execution for the program applications by providing a utility program file, called *RUN.m*. In such file, the number of processors and the name of the program application to be executed are specified. The number of processors is specified in the parameter *Ncpus*, and the program to be executed in the parameter *mFile*, as shown in **Figure 5.11**. Once the number of processors *Ncpus* is specified in the utility program



```

1 % RUN is a generic script for running pMatlab scripts.
2
3 % Define number of processors to use.
4 % All the examples should work with 1, 2, 4 and 8 CPUs.
5 Ncpus =2;
6
7 % Uncomment the name of the script you want to run.
8 mFile = 'AIP_DFTpMatlab_beamforming'; % Trivially parallel fft.
9 %mFile = 'AIP_Newkronecker_pMatlab';
10
11
12
13 % Define cpus.
14 % Empty implies run on host.
15 cpus = {};
16
17
18 % Abort left over jobs.
19 MPI_Abort;
20 pause(2.0);
21
22 % Delete left over MPI directory
23 MatMPI_Delete_all;
24 pause(2.0);
25
26 % Define global variables
27 global pMATLAB;
28
29
30 % Run the script.
31 ['Running: ' mFile ' on ' num2str(Ncpus) ' cpus']
32 eval(MPI_Run(mFile, Ncpus, cpus));
33

```

Figure 5.11: Parallel Utility Program File

RUN.m, the program application can be developed in a MATLAB-like environment. As shown in **Figure 5.12**, the parameter *Ncpus* can be directly used to define the mapping for the distributed matrices. For the beamforming implementation, the data is decomposed along the second order, by distributing the columns among the *Ncpus* processors. The distributed matrix *sample_input_matrix_1* is allocated as a $N \times M$ matrix, where N defines the number of sensors in the linear array, and M is the number of steering angles to be detected by the array. This corresponds to the input matrix of M spatially sampled signals, indicating that each of the N sensors will receive or detect M different values or data points. The distributed matrix *Beam_pattern_1* is allocated as a $N \times M$ matrix, where N defines the number of sensors in the linear array, and M is the number of steering angles to be detected by the array; this corresponds to the output matrix, after applying the FFT to each column of the input matrix, as shown in **Figure 5.13**.


```

% Create Maps.
mapX = 1; mapY = 1;
if (PARALLEL)
    % Break up channels.
    mapX = map([1 Ncpus], {}, 0:Ncpus-1);
    mapY = map([1 Ncpus], {}, 0:Ncpus-1);
    %mapZ = map([Ncpus 1], {}, 0:Ncpus-1);

end
close all;
%B=-1:10^(-2):1;
NumberOfPoints = 256;
B=-1:2/(NumberOfPoints-1):1;

sample_input_matrix_1 = zeros(N,length(B), mapX);
Beam_pattern_1 = zeros(N,length(B),mapY);

```

Figure 5.12: Parallel Utility Program File

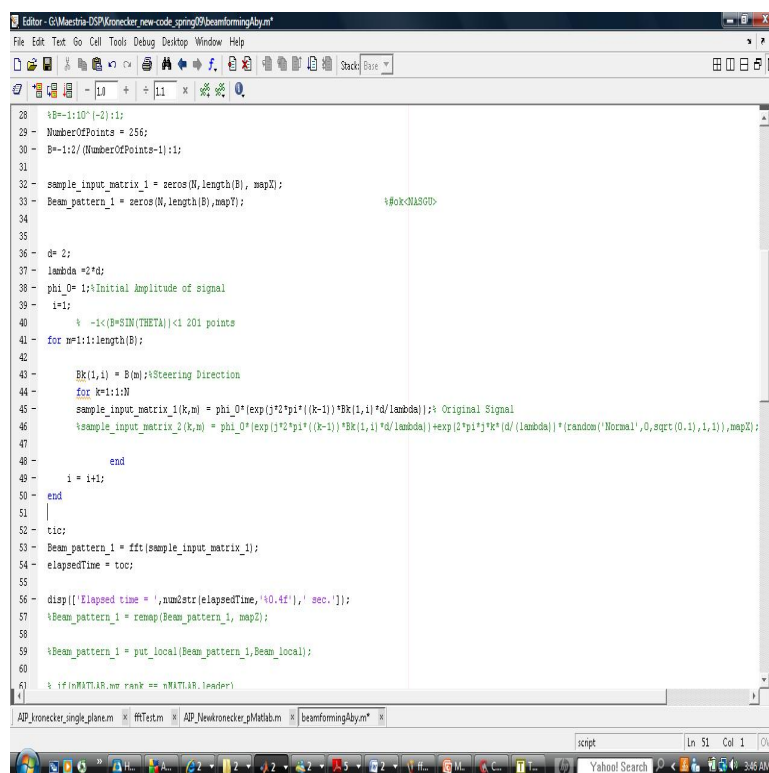


Figure 5.13: DFT Beamforming Implementation

Chapter 6

Beamforming Techniques Implemented on the DSP C6713

6.1 TMS320C6713 DSP Kit

The digital signal processors (DSP) are highly used for a variety of applications, such as image formation processing, speech recognition, communications, and much more. Advantages in using DSPs is that they are processors specially designed to efficiently implement program applications which involve working with, analysing, and processing signals in order to extract information of interest, according to the objective of the application at hand. The principal tool used in order to design program applications on the DSP is the "Digital Starter Kit" (DSK) from Texas Instruments (TI), Inc., which is composed of the TMS320C6713 (C6713) DSK board, shown in **Figure 6.1**, and the program application environment *Code Composer*

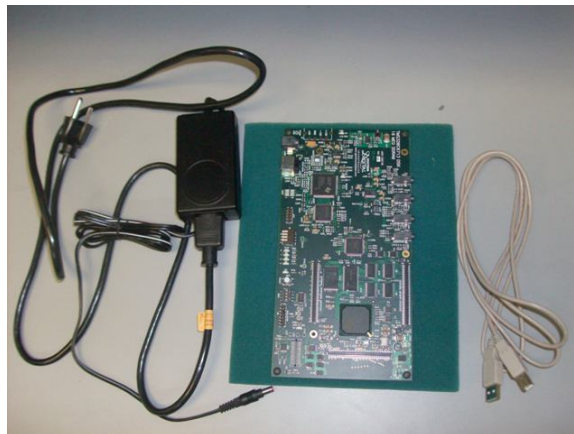


Figure 6.1: TMS320C6713 DSK board

Studio v3.3. Code Composer Studio v3.3 was used to implement the DFT and Kronecker

DFT Beamforming algorithms to be executed on the DSP board, which is further described with more details.

6.2 TMS320C6713 DSP Basic Characteristics

The TMS320C6713 DSP unit is based on the VLIW (Very Large Instruction Word) architecture, which is well-matched for intensive, computational algorithms. Such architecture permits a total of eight instructions to be fetched every cycle. This DSP unit is characterized by the following basic features, as presented in [20]:

- Clock Frequency of 225 MHz
- 32-bit Instructions
- 1.35 giga-floating operations per second (GFL0PS)
- 16 MB of Synchronous Dynamic Random Memory (SDRAM)
- 264 kB of internal memory
 - 8 kB as L1P (program) and L1D (data)Cache
 - 256 kB as L2 memory for program and data
- 256 MB Flash Memory

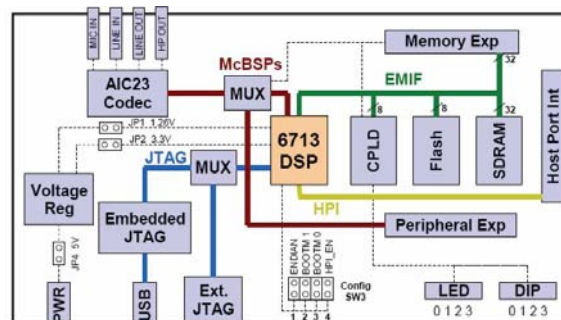


Figure 6.2: TMS320C6713 Architectural Diagram, *courtesy of Chassaing*

6.3 DFT and Kronecker Beamforming Algorithm Implementation Procedures on DSP C6713

The DFT and Kronecker beamforming techniques were implemented on the C6713, since it is considered to be one of TI's most powerful signal processor. As part of this work, a user's

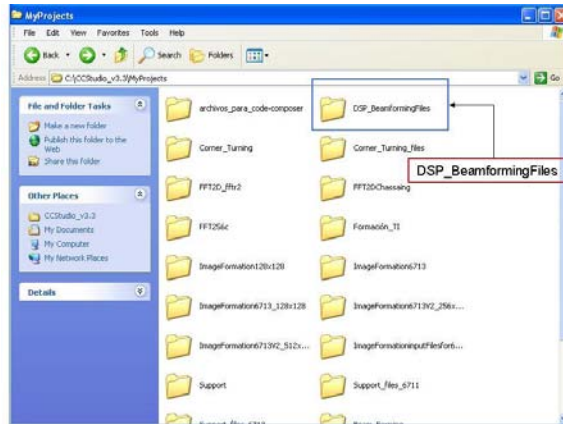


Figure 6.3: Locating the Folder DSP_BeamformingFiles

guide on implementing DFT and Kronecker Beamforming algorithms has been designed for the TMS320C6713 board. The guide follows a step by step process from creating a new project, to finally obtaining a Beamforming application developed for the DSP board, in which the user has the option of executing either DFT or Kronecker Beamforming within the same project. This guide is based on the usage of Code Composer Studio V3.3 for designing the Beam forming imaging program application. It is assumed that a series of files implemented as part of this work are located in the directory **C:\CCStudio_v3.3\MyProjects**.

In designing the user's guide, these key measurable procedures are carefully defined: **initialization and compilation**, **simulation**, and **program execution procedure** on the DSP board. In the following subsections, each of these procedures is explained in detail.

6.3.1 Initialization and Compilation Procedures

The first stage of the Beamforming algorithms implementation procedures consists of locating in the directory **C:\CCStudio_v3.3\MyProjects** the folder **DSP_BeamformingFiles**, as shown in **Figure 6.3**.

Inside the folder **DSP_BeamformingFiles**, the user will find the following subfolders and files (see **Figure 6.4**):

- **32_sensors_input_files_1** - This subfolder contains the following two sets of input files (see **Figure 6.5**):
 - *sample_input_real_without_noise.h*, *sample_input_imag_without_noise.h*
 - *sample_input_real_with_noise.h*, *sample_input_imag_with_noise.h*

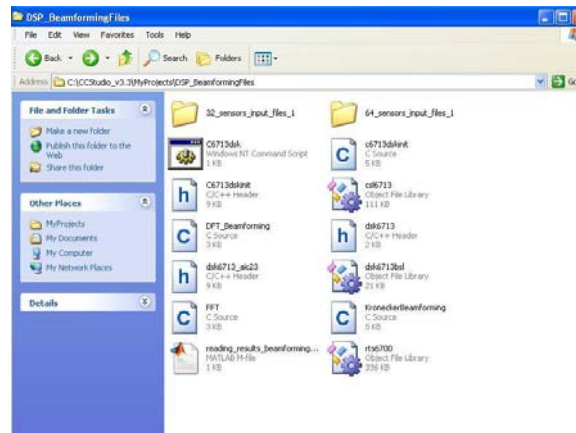


Figure 6.4: Files located in DSP_BeamformingFiles

The first set of input files **sample_input_real_without_noise.h** and **sample_input_imag_without_noise.h** correspond to the real and imaginary parts of the complex input matrix of size 32×201 ; that is, each sensor receives 201 spatially sampled values or data points. These two files assume the absence of noise in the surrounding environment (see **Figure 6.5**).

The second first set of input files **sample_input_real_with_noise.h** and **sample_input_imag_with_noise.h** correspond to the real and imaginary parts of the complex input matrix of size 32×201 ; that is, each sensor receives 201 spatially sampled values or data points. These two files assume the presence of noise in the surrounding environment (see **Figure 6.5**).

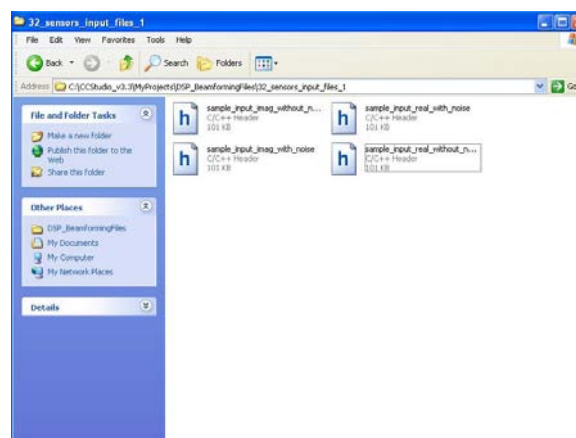


Figure 6.5: Input Files in 32_sensors_input_files_1

- **64_sensors_input_files_1** - This subfolder contains the following two sets of input files (see **Figure 6.6**):
 - *sample_input_real_without_noise.h*, *sample_input_imag_without_noise.h*

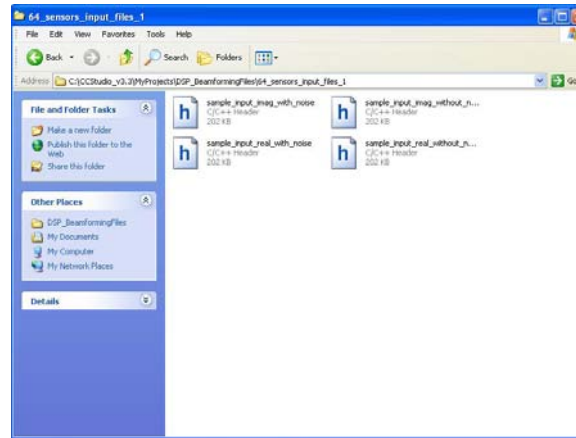


Figure 6.6: Input Files in 64_sensors_input_files_1

- *sample_input_real_with_noise.h*, *sample_input_imag_with_noise.h*

The first set of input files **sample_input_real_without_noise.h** and **sample_input_imag_without_noise.h** correspond to the real and imaginary parts of the complex input matrix of size 64×201 ; that is, each sensor receives 201 spatially sampled values or data points. These two files assume the absence of noise in the surrounding environment (see **Figure 6.6**).

The second first set of input files **sample_input_real_with_noise.h** and **sample_input_imag_with_noise.h** correspond to the real and imaginary parts of the complex input matrix of size 64×201 ; that is, each sensor receives 201 spatially sampled values or data points. These two files assume the presence of noise in the surrounding environment (see **Figure 6.6**).

In summary, for purposes of simplicity, this guide was designed for implementing DFT and Kronecker Beamforming Techniques, for two scenarios: a linear array configuration for 32 and 64 sensors, in which each sensor receive 201 data points or values. The corresponding input files were generated in Matlab. In addition, the following files are also needed:

- **Source files needed for initialization** - These source files are necessary for configuring the programming environment, based on the TMS320C6713 DSP board characteristics (see **Figure 6.4**):
 - *c6713dskinit.c*
 - *C6713dskinit.h*
 - *dsk6713.h*
 - *dsk6713_aic23.h*

- **Libraries** - The following libraries must be included in the program application project, in order to use the TMS320C6713 board features:
 - csl6713.lib
 - rts6700.lib
 - dsk6713bsl.lib
- **C6713dsk.cmd** - This is the linker command file used for the target architecture, in which the types of memory, memory length, memory address, and memory sections of the target board are declared and defined.
- **Main source files needed for program application** - The DFT and Kronecker Beamforming program applications are implemented using the following source files:
 - *DFT_Beamforming.c* - This is the application program that implements the DFT Beamforming algorithm. **Figure 6.7** shows the function: *Beam_pattern_generation()*, which performs the DFT beamforming operation by applying the FFT of N points (N being the number of sensors) to each column of the complex input matrix *input_vectors*.

```

void Beam_pattern_generation()
{
    int col_input_matrix, i, j;

    COMPLEX x_col[N];
    for(col_input_matrix = 0; col_input_matrix < B; col_input_matrix++){

        for (i=0; i<N; i++){
            x_col[i] = input_vectors[i][col_input_matrix];
        }
        FFT(x_col, N);

        for (j=0; j<N; j++){
            Beam_pattern[j][col_input_matrix].real = x_col[j].real;
            Beam_pattern[j][col_input_matrix].imag = x_col[j].imag;
        }
    }
}

```

Figure 6.7: DFT Beamforming Function

- *KroneckerBeamforming.c* - This is the application program that implements the Kronecker DFT Beamforming algorithm. **Figure 6.8** demonstrates the function *DFT_M.Modules_generation()*, which divides each column of the complex input matrix *input_vectors* into M modules, each consisting of N elements. For each module, the FFT of N points is applied. The function *Linear_Combination_DFT_Modules()* is used to add the corresponding M modules (see **Figure 6.9**).
- *FFT.c* - This is the FFT function provided from Chassaing for obtaining the Fourier transforms.

```

void DFT_M_Modules_generation()
( short i,j,k,p,b,c,n_index;

for(i=0;i<B;i++) {
    b=0; // Marks the current position in each input vector

    // Processing each input vector taken as a column of the matrix
    for(j=0;j<L;j++){
        x_L[j] = input_vectors[j][i];
    }

    // Divide each input vector into M modules
    for(k=0;k<M;k++){
        for(p=0;p<N;p++){
            x_module_N[p] = x_L[b];
            b++;
        }

        FFT(x_module_N,N);
        n_index=b-N; // To place each DFT module currently in the output matrix
        for(c=0;c<N;c++){
            DFT_modules_matrix[n_index][i] = x_module_N[c];
            n_index++;
        }
    }
}

```

Figure 6.8: DFT_M_Modules_generation()

```

void Linear_Combination_DFT_Modules()
( short i,j,k,m;

    //Linear combination of the M DFT Modules
    for(i=0;i<N;i++){
        for(j=0;j<B;j++){
            for(k=0;k<=L-N;k=k+N){
                m=i+k;
                Beam_pattern[i][j].real = Beam_pattern[i][j].real+ DFT_modules_matrix[m][j].real;
                Beam_pattern[i][j].imag = Beam_pattern[i][j].imag+ DFT_modules_matrix[m][j].imag;
            }
        }
    }
}

```

Figure 6.9: Linear_Combination_DFT_Modules()

- **reading_results_beamforming_Code_Composer.m** -This is the Matlab m.file that will be used to plot the beamforming results obtained on the DSP.

Once the user has located each of the files shown in **Figure 6.4**, he or she should proceed to find the **Setup CCStudio v3.3 icon** (see **Figure 6.10**). Here, the programming environment must be selected: simulation or emulation. Simulation implies that the application program developed can be compiled and executed, without physically connecting the target board to the computer. On the other hand, emulation implies that the target must be connected to the computer in order to compile and execute the application program. If this icon does not appear at the desktop, the user should go to **All Programs**→**Texas Instruments**→**Code Composer Studio 3.3**→**Setup CCStudio v3.3** (see **Figure 6.11**).

Next, the **CCStudio v3.3 icon** is located, which is used to launch Code Composer Studio, where the program application project for imaging formation is created (see **Figure 6.10**). Again, if this icon does not appear at the desktop, then the user should go to **All Programs**→**Texas Instruments**→**Code Composer Studio 3.3**→**Code Composer Studio** (see **Figure 6.11**).



Figure 6.10: Locating Setup CCStudio v3.3 and CCStudio v3.3 icons

6.3.2 Simulation Procedure

As mentioned previously, simulation implies that the program application developed can be compiled and executed, without physically connecting the target board to the computer. To conduct a simulation analysis (see **Figure 6.12**), the user must access the Setup Code Composer Studio v3.3 tool, and follow these subsequent steps:

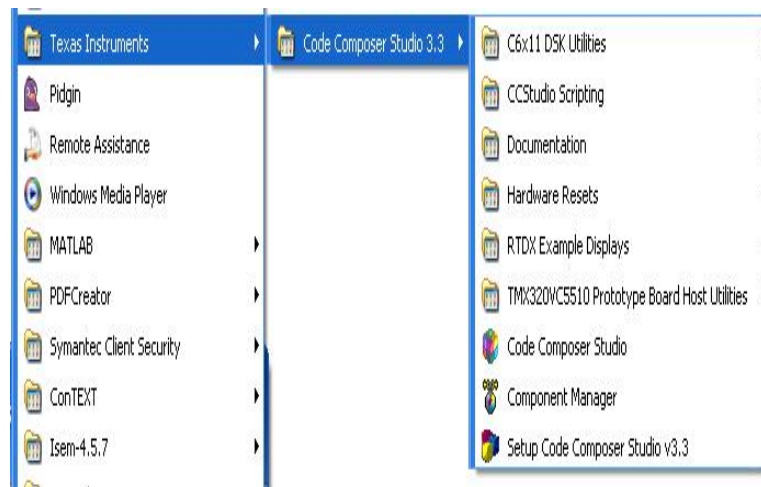


Figure 6.11: Path to Code Composer Studio and Setup Tools

- Next to **Available Factory Boards**, under **Family**, select the option **C67xx**.
- Under **Platform**, select **simulator**.
- Under **Endianness**, select **little**.
- Under **Available Factory Boards**, a list of possible simulators should appear. Here, **C6713 Device Cycle Accurate Simulator** should be selected, by a single click, then pressing the **Add button**, located at the middle bottom. The simulator can also be selected by double clicking on the simulator board.
- Next, press **Save & Quit**. Note: if there are any other boards under **System Configuration**, proceed to remove them. This is done by selecting each board and hitting the delete key. Only the **C6713 Device Cycle Accurate Simulator** must be selected.
- A prompt window will appear, asking the user if he/she wishes to save the changes made to system configuration. The button **Yes** should be selected.
- A second prompt window will appear, asking the user if Code Composer Studio should start on exit. The user should press **Yes**.

Once Code Composer is launched and opened, the user must go to **Project**, located at the upper menu, and select the option **New**. This opens the Project Creation window. Next, in **Project Name**, the user should type *DSP_Beamforming* as the name for this project. It should be verified that the location where the project will be created is:

C:\CCStudio.v3.3\MyProjects\DSP_Beamforming. Also, in textbfProject Type, the option **Executable (.out)** must be selected, and the **Target** selected should be *TMS320C67XX* (see **Figure 6.13**). The user should verify that the project **DSP_Beamforming.pjt** was successfully

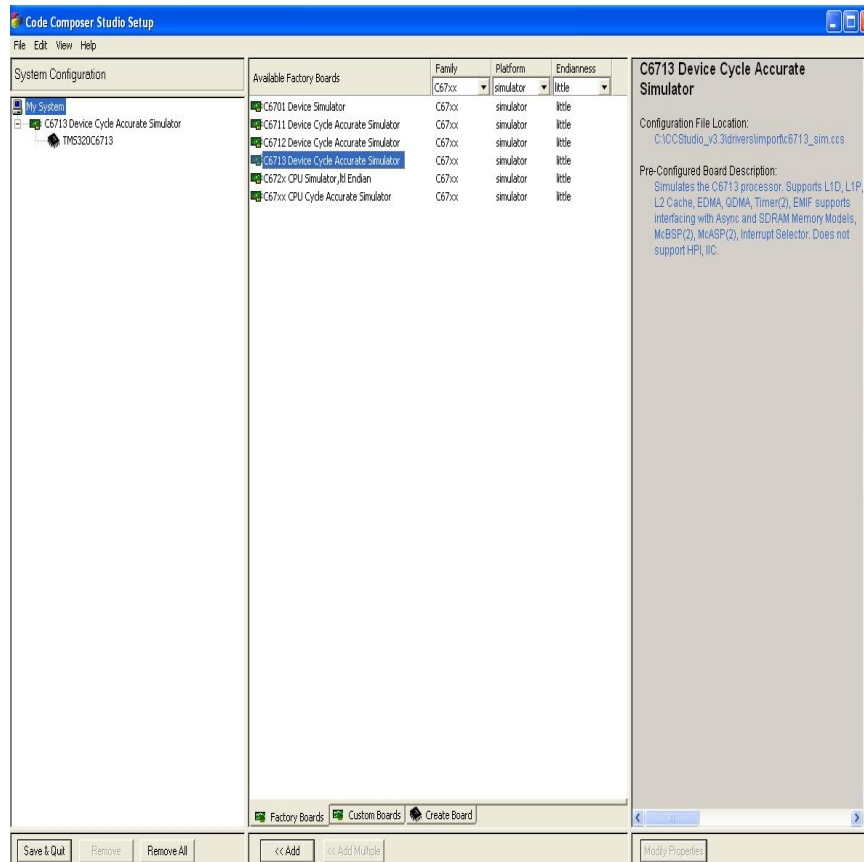


Figure 6.12: C6713 Device Cycle Accurate Simulator Environment Selection

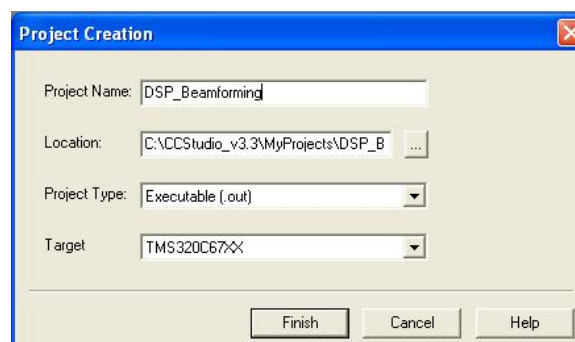


Figure 6.13: Creating DSP Beamforming Project

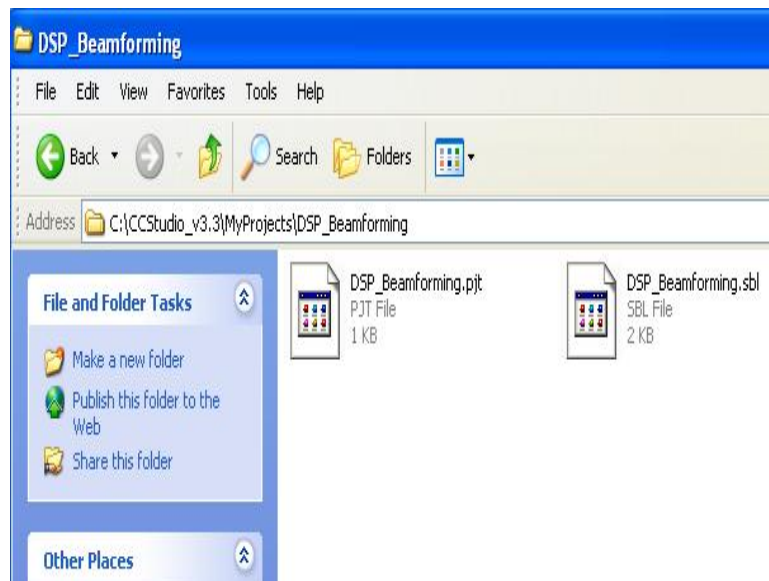


Figure 6.14: Locating DSP Project

created in the directory: **C:\CCStudio_v3.3\MyProjects\DSP_Beamforming** (see **Figure 6.14**).

All of the initialization files located in **C:\CCStudio_v3.3\MyProjects\DSP_BeamformingFiles** must be copied and placed in the same directory where the project was created:

C:\CCStudio_v3.3\MyProjects\DSP_Beamforming.

After making sure that the project was successfully created, **Project** should be once again selected, located at the upper menu. Under **Project**, the option **Add Files to Project** should be selected (see **Figure 6.15**). This procedure is used to add each of the following files located in the directory **C:\CCStudio_v3.3\MyProjects\DSP_Beamforming**:

- c6713dskinit.c
- C6713dsk - Linker Command file
 - Libraries
 - csl6713.lib
 - rts6700.lib
 - dsk6713bsl.lib
- DFT_Beamforming.c or KroneckerBeamforming.c
- FFT.c

It is important to note that either the file **DFT_Beamforming.c** or **KroneckerBeamforming.c** should be added to the project, since they are different types of beamforming techniques. If the

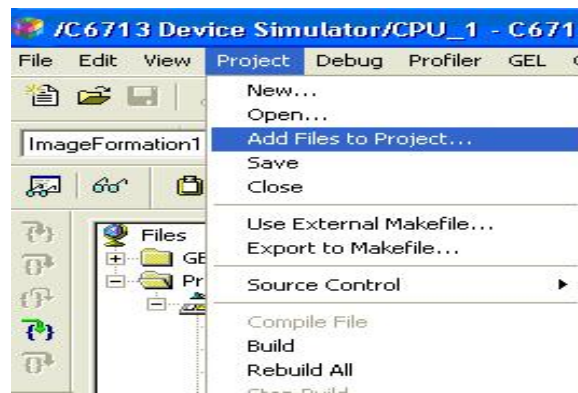


Figure 6.15: Adding Files to DSP Beamforming Project

user wishes to execute the DFT Beamforming technique, then the first file should be added. If, on the other hand, the user wishes to execute the Kronecker Beamforming technique, then the latter should be added. The user should open the file **DFT_Beamforming.c** or **KroneckerBeamforming.c** and specify in the function fopen the following possible input files (see **Figure 6.16**):

If the complex input vector is simulated without noise:

- fopen("sample_input_real_without_noise.h", "r")-real component
- fopen("sample_input_imag_without_noise.h", "r")-imaginary component

If the complex input vector is simulated with noise:

- fopen("sample_input_real_with_noise.h", "r")
- fopen("sample_input_imag_with_noise.h", "r")

Then the user should again go to **Project** and select **Build Options**, as indicated in **Figure 6.17**. This option is used to properly set up the compiler and linker, based on the characteristics of the TMS320C6713 DSP board. The following settings should to be chosen or written, and the option OK is selected after all settings are verified (see **Figure 6.18**).

- Under **Compiler**→**Category**→**Basic**
 - The target version: C671x (-mv6710) should be highlighted.
- Under **Compiler**→**Preprocessor**:

```

index=fopen("sample_input_real_without_noise.h","r");
if((index)==NULL) {
    puts("File could not be open");
    exit(-1);
}

for(m = 0; m < N; m++){
    for(k = 0; k < B; k++){
        fscanf(index, "%f",&input_vectors[m][k].real);

    }
}

fclose(index);

index=fopen("sample_input_imag_without_noise","r");
if((index)==NULL) {
    puts("File could not be open");
    exit(-1);
}

for(m = 0; m < N; m++){
    for(k = 0; k < B; k++){
        fscanf(index, "%f",&input_vectors[m][k].imag);
    }
}

```

Figure 6.16: Specifying Input Files

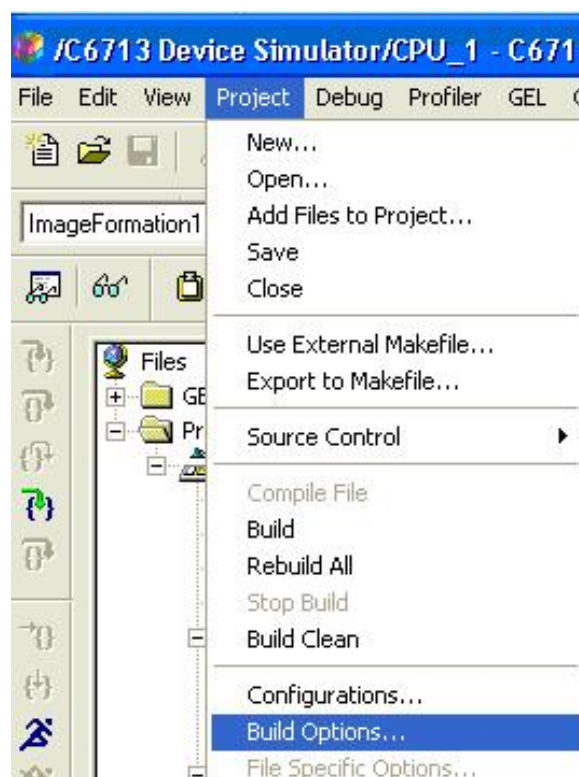


Figure 6.17: Selecting Build Options

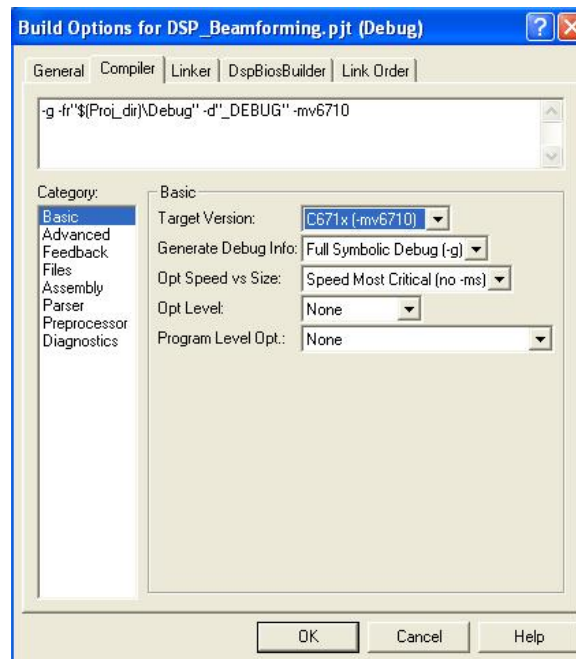


Figure 6.18: Building Options for Compiler→Category→Basic

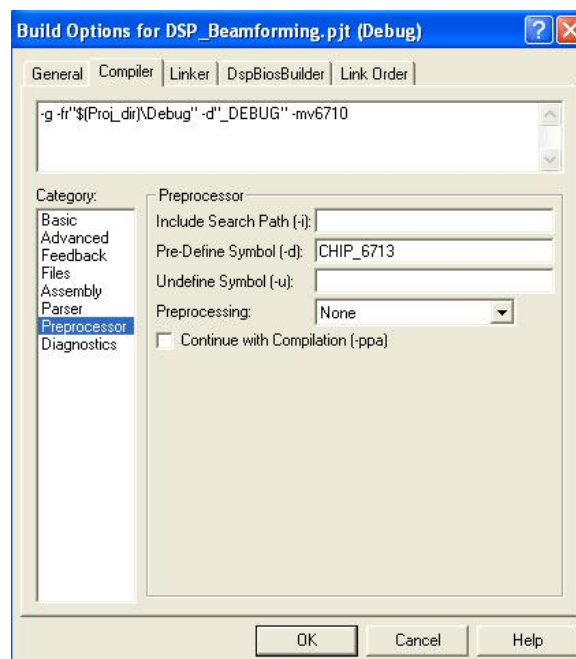


Figure 6.19: Building Options for Compiler→Preprocessor→Pre-Define Symbol

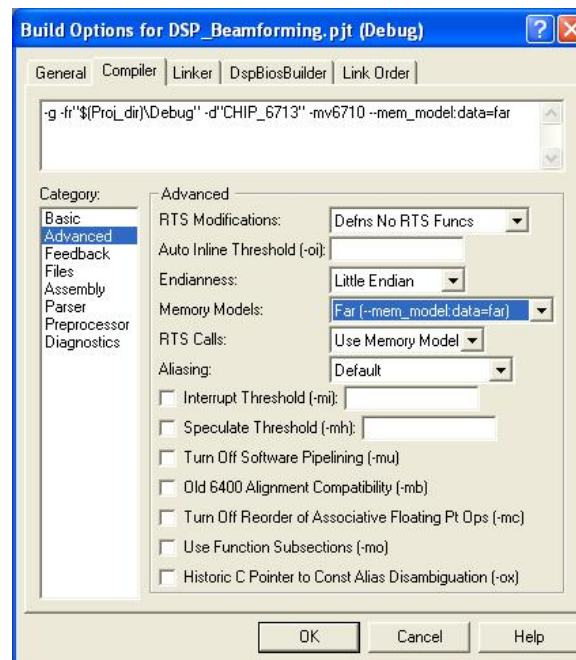


Figure 6.20: Building Options for Compiler→Advanced

- In **Pre-Define Symbol**, the following should be written: *CHIP_6713*. This specifies the DSP chip that the target board utilizes (see **Figure 6.19**).
- Under **Compiler→Advanced** (see **Figure 6.20**):
 - In **Memory Models**, *Far* (*--mem_model: data=far*) should be chosen.
 - The Endianness type should be **Little Endian**.
- Under **Linker→Libraries**:
 - In **Included Libraries (-I)**, these libraries must be specified (see **Figure 6.21**):
rts6700.lib; dsk6713bsl.lib; csl6713.lib

Now the user may click *OK* once all the previous building option settings have been established.

The next step is to compile the project **DSP_Beamforming.pjt** by selecting the option **Re-build All**, as shown in the **Figure 6.22**: If compiled correctly, there should be zero errors, at the output window, under **Build**, which is located at the bottom of the workspace (see **Figure 6.23**). Once the project has compiled correctly, the user needs to select how many sensors to use for the linear array: 32 or 64 for this example. In addition, the user has the option of selected the input matrix with or without noise, as indicated previously. The user

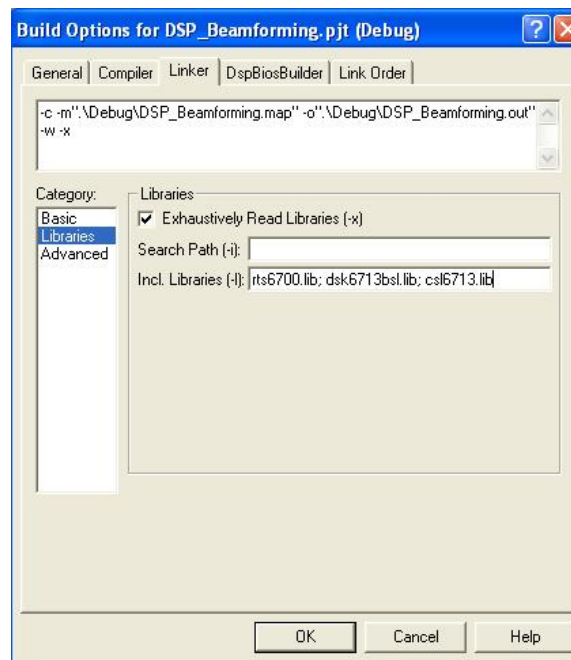


Figure 6.21: Building Options for Linker→Libraries:



Figure 6.22: Compiling DSP Beamforming Project in Simulation Environment

```
[Linking...] "C:\CCStudio_v3.3\C6000\cgtools\bin\cl6x" -@"Debug.lkf"
<Linking>
>> warning: creating .stack section with default size of 400 (hex) words.
    Use
    -stack option to change the default size.
>> warning: creating .system section with default size of 400 (hex) words.
    Use
    -heap option to change the default size.
Build Complete,
0 Errors, 3 Warnings, 0 Remarks.
```

Figure 6.23: Successful Compilation of the DSP Beamforming Project

must copy the desired files (real and imaginary files of the desired input format) from either the directory **C:\CCStudio_v3.3\MyProjects\DSP_Beamforming\32_sensors_input_files_1** or **C:\CCStudio_v3.3\MyProjects\DSP_Beamforming\64_sensors_input_files_1** and placed in the directory **C:\CCStudio_v3.3\MyProjects\DSP_Beamforming\Debug**, as shown in **Figure 6.24**. The folder Debug is created when the program compiles correctly for the first time. Once this is done, the simulation application program is ready to be loaded and executed on the simulated target, by following these subsequent steps:

- **File→Load Program** should be selected, as shown in **Figure 6.25**:

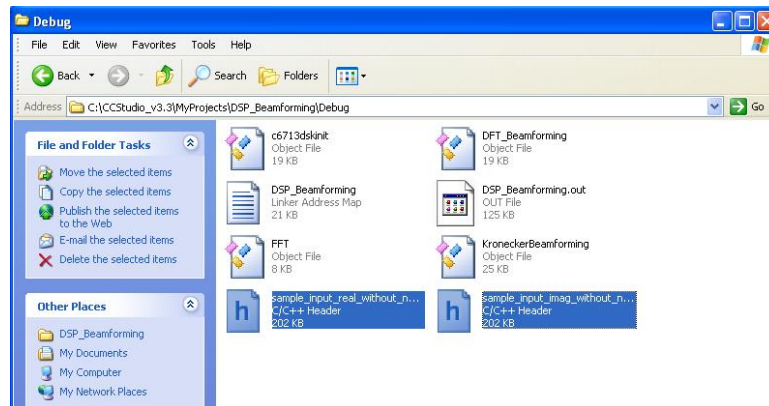


Figure 6.24: Placing Input Files in Debug Directory

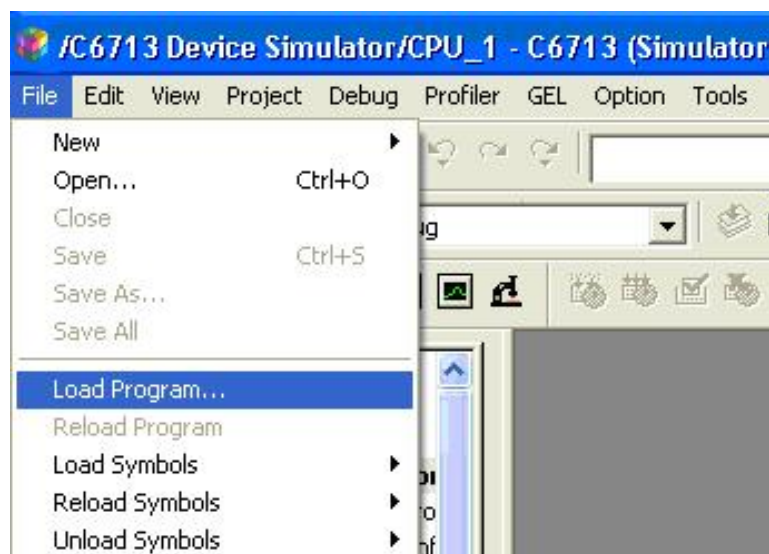


Figure 6.25: Loading DSP Beamforming Program Application

- In order to load the program application to the simulated target, which is the C6713 Device Simulator, the output file **DSP_Beamforming.out**, should be opened, as presented in **Figure 6.26**. This file is located in the following directory:
C:\CCStudio_v3.3\MyProjects\DSP_Beamforming\Debug.
- Once the executable output file is loaded to the simulated target, the application program can then be executed by clicking on the **Run** button, located on the left side of the workspace, in the project window, shown in **Figure 6.27**:

After the beamforming application program has finished execution, the following **.dat** files are created in the following directory:

C:\CCStudio_v3.3\MyProjects\DSP_Beamforming\Debug: Beam_pattern_image.dat and **Beam_pattern_real.dat**, which correspond to the real and imaginary parts of the matrix containing the beam pattern formations.

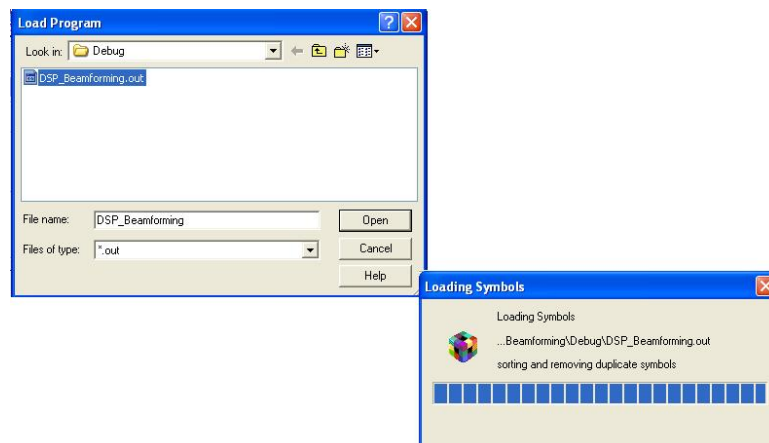


Figure 6.26: Selecting and Loading Executable File onto Simulated Target

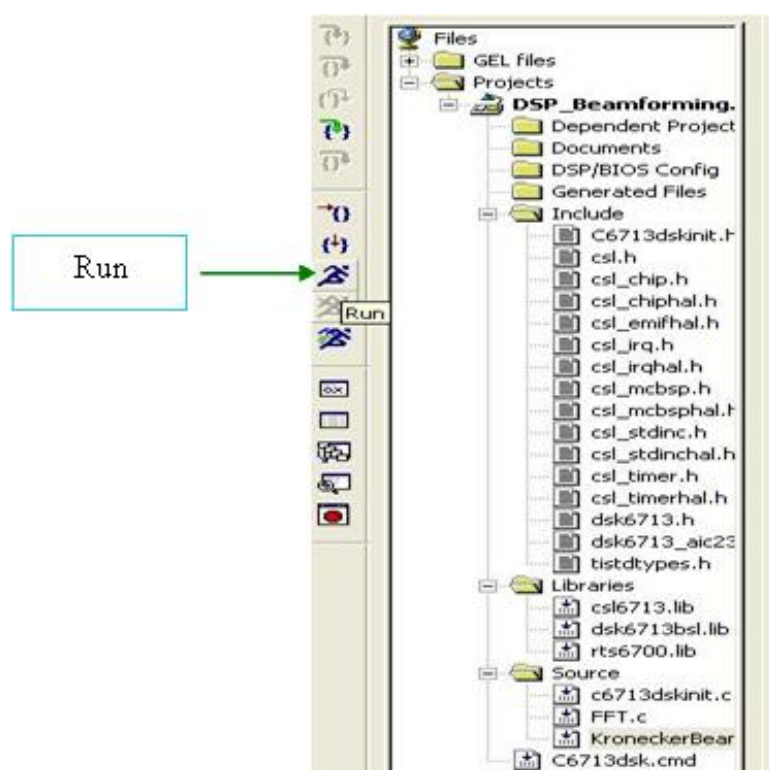


Figure 6.27: Executing Program Application on Simulated Target

6.3.3 Program Execution Procedure on Target Architecture

After completing the simulation analysis on the **C6713 Device Cycle Accurate Simulator**, the program execution procedure is then carried out on the actual target architecture; that is, the program application is compiled, loaded, and executed on the physical TMS320C6713 DSP board. In order to conduct this emulation analysis, the target board must be connected to its power supply, and a USB connection is required to communicate the board with the computer. This is shown in **Figure 6.28**



Figure 6.28: TMS320C6713 Target Board Connection, courtesy of Chassaing

To conduct the Emulation Analysis (see **Figure 6.29**), the user should close the existing Code Composer Session of the Simulation Analysis, and access once again the **Setup Code Composer Studio v3.3** tool, following these subsequent steps:

- Next to **Available Factory Boards**, under **Family**, select the option **C67xx**.
- Under **Platform**, select **dsk**.
- Under **Endianness**, select **little**.
- Under **Available Factory Boards**, the option **C6713 DSK-USB** should appear. Here, **C6713 DSK-USB** should be selected, by a single click, then pressing the **Add button**,

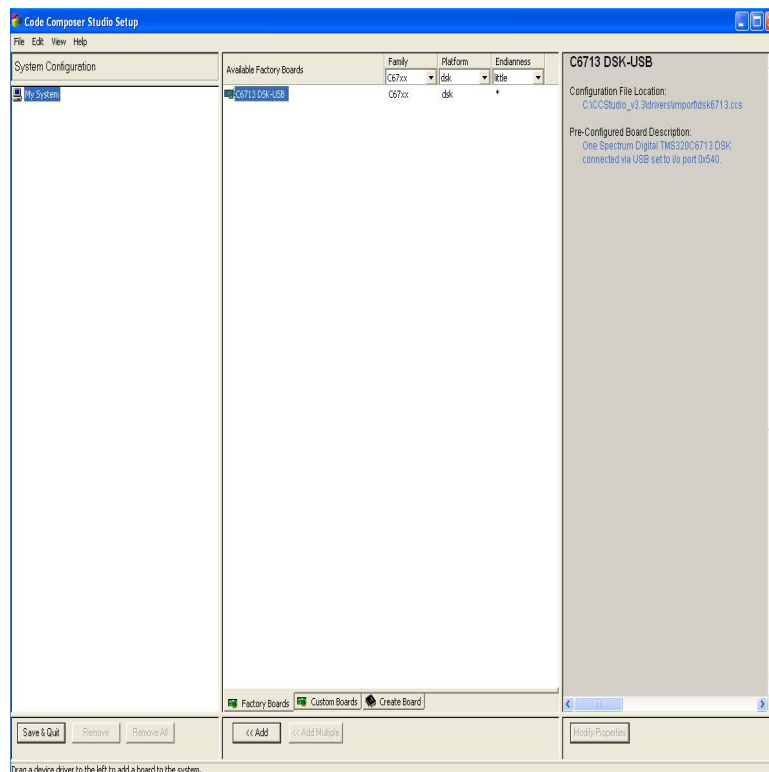


Figure 6.29: C6713 DSK-USB Emulator Environment Selection

locating at the middle bottom. The emulator can also be selected by double clicking on the emulator board.

- Next, press **Save & Quit**. Note: if there are any other boards under **System Configuration**, proceed to remove them. This is done by selecting each board and hitting the delete key. Only the **C6713 DSK-USB** must be selected.
- A prompt window will appear, asking the user if he/she wishes to save the changes made to system configuration. The button **Yes** should be selected.
- A second prompt window will appear, asking the user if Code Composer Studio should start on exit. The user should press **Yes**.

Once the target board is connected, the project **DSP_Beamforming.pjt**, created previously in the simulation analysis, should be opened, which is located in the following directory:

C:\CCStudio_v3.3\MyProjects\DSP_Beamforming. This is done by going to **Project**, and under this option, selecting **Open** and the project file **DSP_Beamforming.pjt**.

Next, the same settings in **Build Options** should be verified, just as in the simulation procedure. This is done by going to **Project** and selecting **Build Options**. The option **Finish** should not be selected until all settings have been configured and verified. For the emulation analysis



Figure 6.30: Compiling DSP Beamforming Project in Emulation Environment

procedure, the project is compiled in the same way as for the simulation procedure (see Figure 28).

6.3.4 Obtaining Beamforming Results using Matlab

The beamforming results for the simulation and emulation procedures can be viewed in Matlab, executing the Matlab file **reading_results_beamforming_Code_Composer.m**. This file should be previously placed in **C:\CCStudio_v3.3\MyProjects\DSP_Beamforming\Debug**. The user should open this file and verify that the parameters **L** (number of sensors), **B** (the number of data points that each sensor will receive), **M** (number of modules) and **N** (the number of sensors in each module) are the same as specified in the program application for the DSP (see **Figure 6.31**). To obtain the beam pattern formations, Matlab must be launched, which

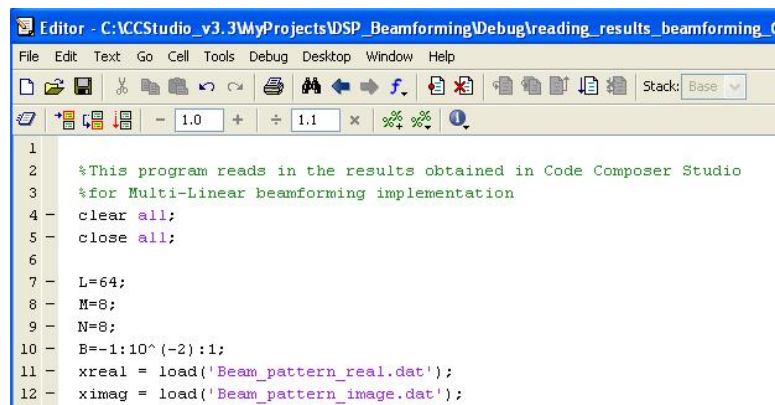


Figure 6.31: Matlab Code for Reading DSP Results

is located at **Start Menu**→**All Programs**. For the majority of cases, a desktop icon of this program exists, through which Matlab can also be opened by double clicking on this icon. In the Matlab command window, the following two lines must be typed:

- `cd C:\CCStudio_v3.3\MyProjects\DSP_Beamforming\Debug`
- `reading_results_beamforming_Code_Composer`

Figure 6.32 presents an example of multiple beam pattern formations obtained from the DSP, for 64 sensors.

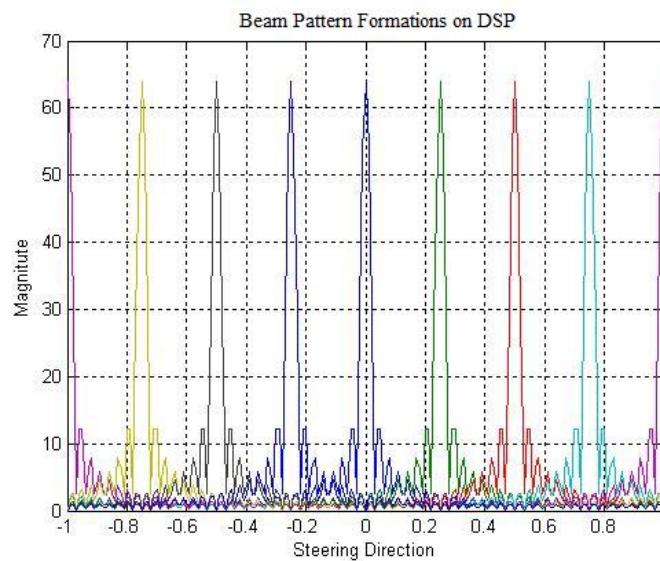


Figure 6.32: Beam Pattern Formations from DSP

6.4 A DSP Multicore Environment Architecture

In the search of an ideal hardware multicore architecture that is capable of delivering high performance, at a reasonable low power consumption, the TMS320C6474 multicore DSP from Texas Instruments showcases as an ideal hardware environment for running the DFT beamforming techniques. As presented in [21], such multicore architecture possess the following favorable features:

- 32-bit Instructions
- 3 $TMS320C64x+^{TM}$ DSP Cores, each at 1 GHz
- 32 kB L1P and L1D per core
- 3 MB of total L2 memory in two configurations

The TMS320C6474 multicore DSP presents the advantage of integrating three of TI's C64x+ cores, each core running at 1 GHz; this contributes to a delivery performance of 3 GHz. According to [21], TMS320C6474 multicore architectures are the highest-performance multicore DSP generation in the $TMS320C6000^{TM}$ DSP platform. Hence, the combination of three potential DSP cores in a multicore architecture, can be ideally used to implement the DFT beamforming techniques in parallel computational environment.

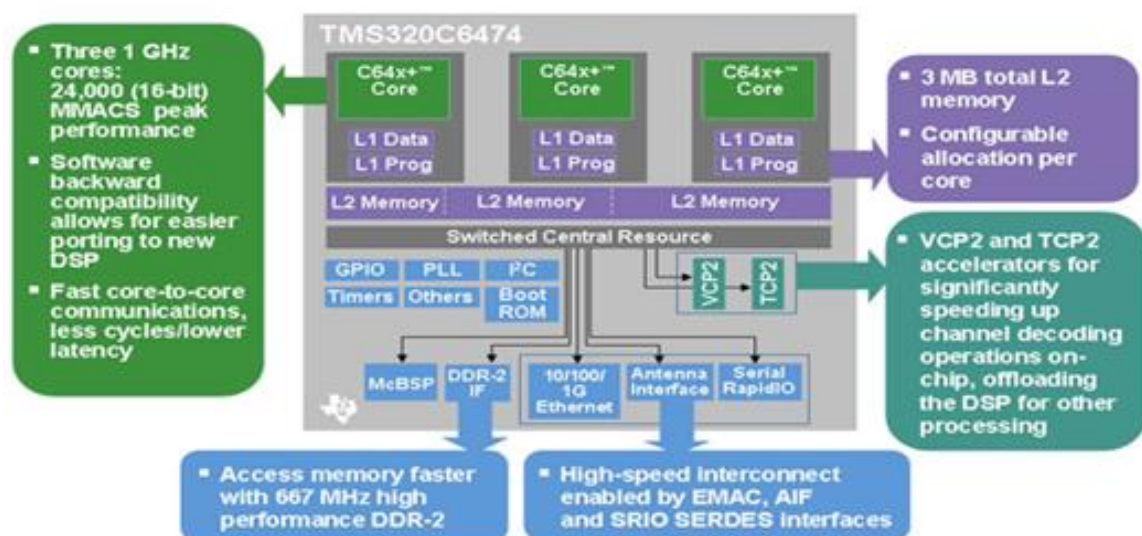


Figure 6.33: TMS320C6474 Block Diagram, courtesy of Texas Instruments, Inc.

Chapter 7

Experimental Results

In this chapter, the essence of the DFT and Kronecker Beamforming Algorithm implementations, the design-level approach used in implementing these algorithms, and the experimental results obtained are presented and discussed in details.

7.1 DFT and Kronecker Beamforming Algorithm Design

In this section, the essence of developing the DFT and Kronecker Beamforming algorithms is carefully described. **Figure 7.1** illustrates the procedure followed in developing the beamforming algorithms for the implementation platforms described in the subsequent section. First, it is assumed that an $L \times B$ input matrix is provided as input for these algorithms which is characterized as follows. B distinct steering angles defined within the interval $-\frac{\pi}{2} \leq \theta_0 \leq \frac{\pi}{2}$, are considered. Hence, B input vectors of the form $\Phi(\beta_b)$ are defined, for each steering direction $\beta_b = \sin(\theta_b)$, for $b = 0, 1, \dots, B - 1$. These input vectors constitutes the columns of the input matrix, as shown in **Figure 7.2**

According to **Figure 7.1**, the DFT beamforming technique consists of applying the FFT of length L (provided that L sensors compose the linear array) to each column of the input matrix, and replacing each column by its corresponding transform in the frequency domain. After applying the FFT to all of the columns, the resulting new matrix consists of the beam pattern formations, arranged as an $L \times B$ matrix. It is important to note that the beam pattern formations are read by each row of the final matrix. More specifically, the beam pattern contained in each row reveals the maximum value, corresponding to the dominating signal, with respect to a particular steering direction. As an example, **Figure 7.3** represents a beam pattern, for a linear array consisting of 64 sensors. The main lobe in the beam pattern shown in **Figure 7.3** indicates which of the B input signal vectors dominated, arriving at a

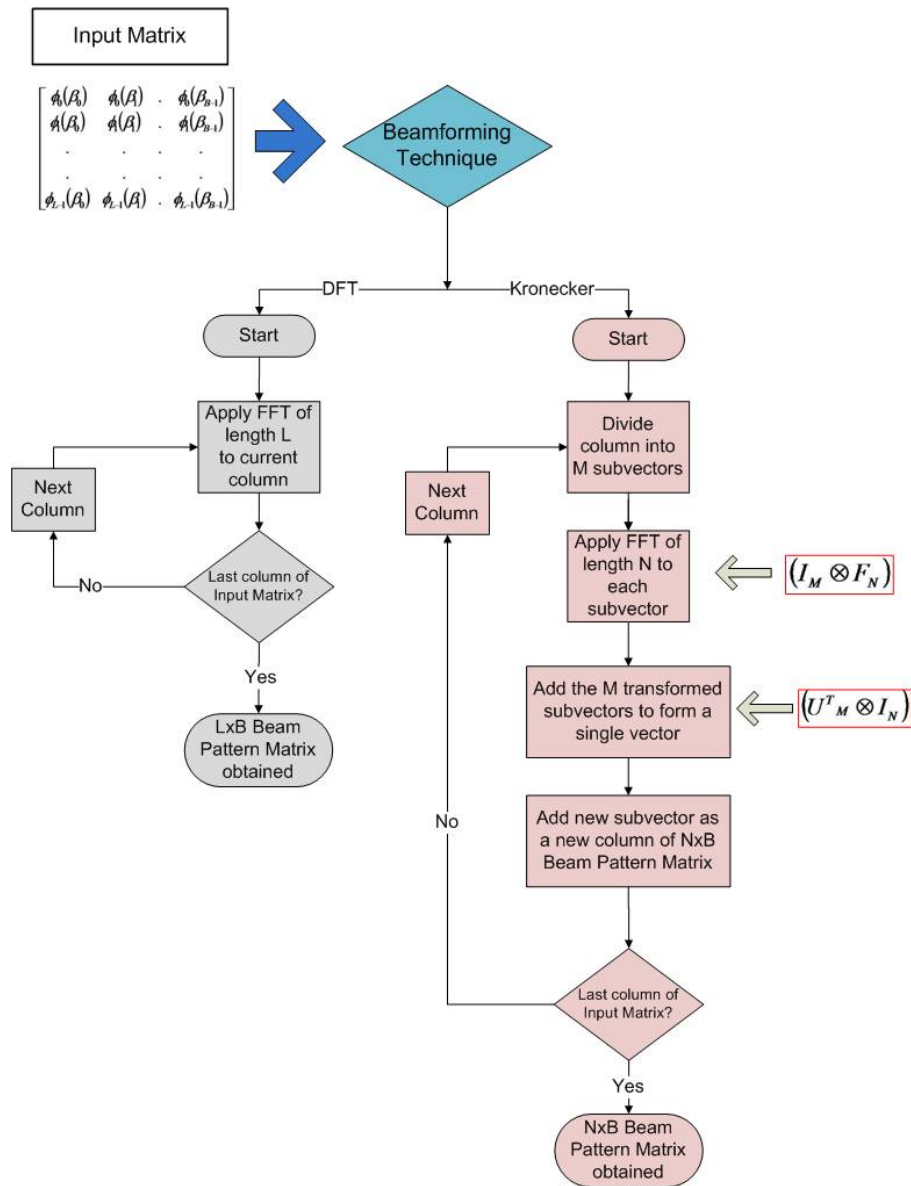


Figure 7.1: Beamforming Algorithms Design

$$\begin{bmatrix}
 \phi_0(\beta_0) & \phi_0(\beta_1) & \dots & \phi_0(\beta_{M-1}) \\
 \phi_1(\beta_0) & \phi_1(\beta_1) & \dots & \phi_1(\beta_{M-1}) \\
 \vdots & \vdots & \ddots & \vdots \\
 \phi_{L-1}(\beta_0) & \phi_{L-1}(\beta_1) & \dots & \phi_{L-1}(\beta_{M-1})
 \end{bmatrix}$$

Figure 7.2: Input Matrix Format

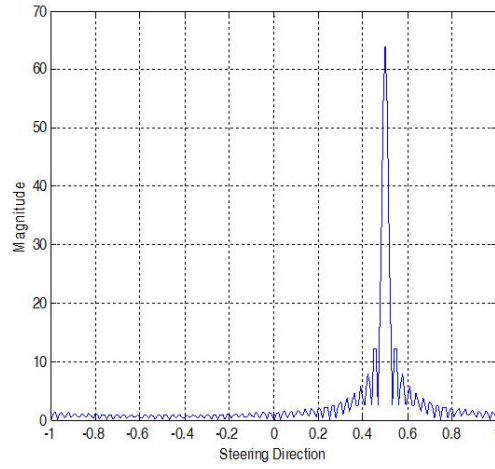


Figure 7.3: Beam Pattern Formation Example for 64 Sensors

certain steering direction $\beta_0 = \sin(\theta_0)$. For this particular beam pattern, the main lobe is detected at a steering direction of approximately $\beta = .5$, corresponding to an incidence angle of $\theta = 30^\circ$. In general, the main lobe of a particular beam pattern will indicate the steering direction or angle at which the dominant signal is detected.

For the Kronecker Beamforming technique, according to **Figure 7.1**, the concept of modularity and linear combination is applied. Having in mind that each of the B columns of the input matrix is an input signal vector ϕ of length L , where L corresponds to the total number of sensors in the linear array, the essence of this beamforming technique is described as follows (see **Figure 7.4**). Kronecker beamforming operation considers the case when L , the number of sensors can be expressed as $L = MN$. This beamforming technique consists of dividing

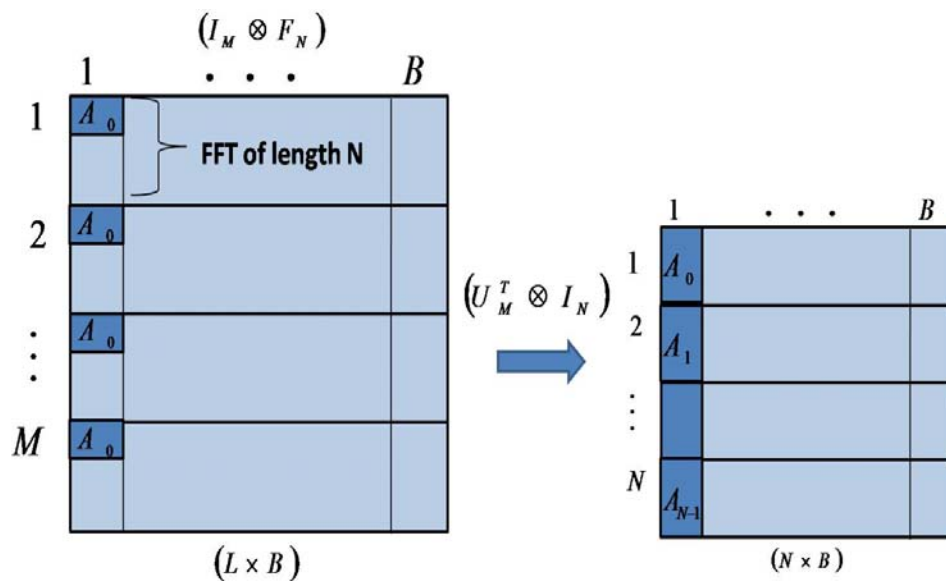


Figure 7.4: Essence of Kronecker Beamforming Technique

each input vector defined in terms of a specific steering direction (each column of the input matrix shown in **Figure 7.2**), into M subvectors, each of length N ; that is, the entire linear array of L sensors are divided into M modules, where each module has N sensors A_0, \dots, A_{N-1} (see **Figure 7.4**). For each column of the input matrix divided into M subvectors, an FFT of length N is applied to each subvector, through the kronecker operation $(I_M \otimes F_N)$. Next, the M transformed subvectors are linearly combined through addition, via the Kronecker operation $(U_M^T \otimes I_N)$, to form a new transformed column vector of size N .

This same procedure is done for each column of the input matrix, and each new transformed column vector of size N constitutes a column of the final output beam pattern matrix; in other words, through the application of modularity and linear combination of M modules, an equivalent, reduced beam pattern matrix of size $N \times B$ is obtained, in place of having a larger beam pattern matrix of size $L \times B$ (see **Figure 7.4**). This demonstrates that through modularity, a large linear array of size $L = MN$ can be reduced to an equivalent linear array of N sensors, where the N sensors of each module are linearly combined. For this technique, the final beam pattern formations are also read by rows, in which the resulting beam patterns are arranged as an $N \times B$ matrix, N being the number of sensors in each of the M modules.

7.2 DFT Beamforming Implementation Procedure

The implementations of the DFT and Kronecker Beamforming Algorithms are based on a linear array configuration, consisting of L sensors, in order to take advantage of the FFT algorithms for obtaining the beam pattern formations. Afterwards, Kronecker products formulations are integrated to the DFT Beamforming technique, with the purpose of introducing the concept of mapping the beamforming algorithm to a parallel architectural configuration through the usage of modularity and linear combination.

The approach of designing the beamforming techniques consisted in first using MATLAB as the principal tool for modeling the beamforming algorithms, in an ideal simulated environment for signal processing applications. A series of tests were conducted in order to determine how the spacing between sensors and the wavelength of the incoming signal can affect the beam pattern formations, especially in the presence of noise; these tests are discussed with further details in the next section.

Next, the DFT and Kronecker Beamforming techniques were implemented, using C language, for execution on the DSP TMS320C6713, from Texas Instruments, Inc., applying the simulation and emulation analysis procedures, defined in chapter 6. These beamforming techniques were also implemented using C language for execution on the Gumstix Verdex. Further research on using Kronecker product formulations for the mapping of the beamforming operation

onto a parallel architecture, led to using the tool pMATLAB, as an ideal parallel simulation environment, which was described in chapter 5 (see **Figure 7.5**).

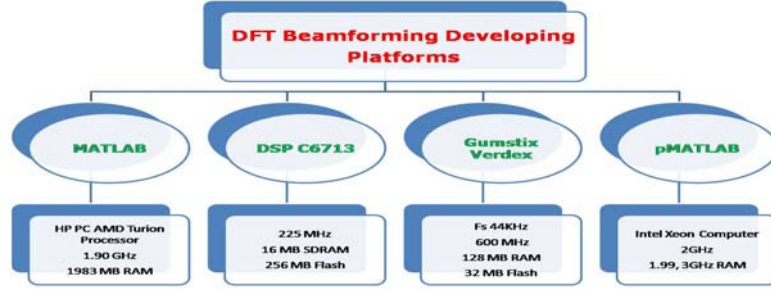


Figure 7.5: Beamforming Techniques Implementation Platforms

7.2.1 Signal Analysis and Metrics

In this section, the general model for the beamformer is provided and an analysis is performed on this model acting on an input signal when white Gaussian noise is assumed as background noise. Let the steering direction $\beta_0 = \frac{2m}{L}$, $m \in Z_L$. Then the generalized beamformer model used in this work is expressed as follows [22]:

$$x_m(t) = \sum_{n=0}^{L-1} \phi_n \left(t, \frac{m}{L} \right) e^{-j2\pi \frac{nd}{\lambda} \left(\frac{2m}{L} \right)} + n_m(t); \quad m, n \in Z_L. \quad (7.1)$$

The function $randn(N, B)$ was used to generate the background noise. This function produces pseudo-random values characterized from a normal distribution with mean zero and standard deviation σ one, of dimension $N \times B$. Such background noise was added to the original input matrix. Such noise was varied, by changing the amplitude. For example, the command in MATLAB $A * (0.5 * randn(N, B))$, varies the amplitude of noise matrix by multiplying the function $randn(N, B)$ by a factor of A , where A is assumed to be an integer. The signal-to-noise (SNR) ratio was used to compare the amount of an input signal to the amount of background noise, defined as follows in dB:

$$SNR = 10 \log_{10} \frac{E(|\phi|)^2}{E(|n|)^2} = 20 \log_{10} \frac{E(|\phi|)}{E(|n|)}; \quad (7.2)$$

where $E(|\phi|)$ and $E(|n|)$ denote the expected value or mean of an input vector and noise vector, respectively. The higher the SNR is, the less impact the noise has over the desired signal.

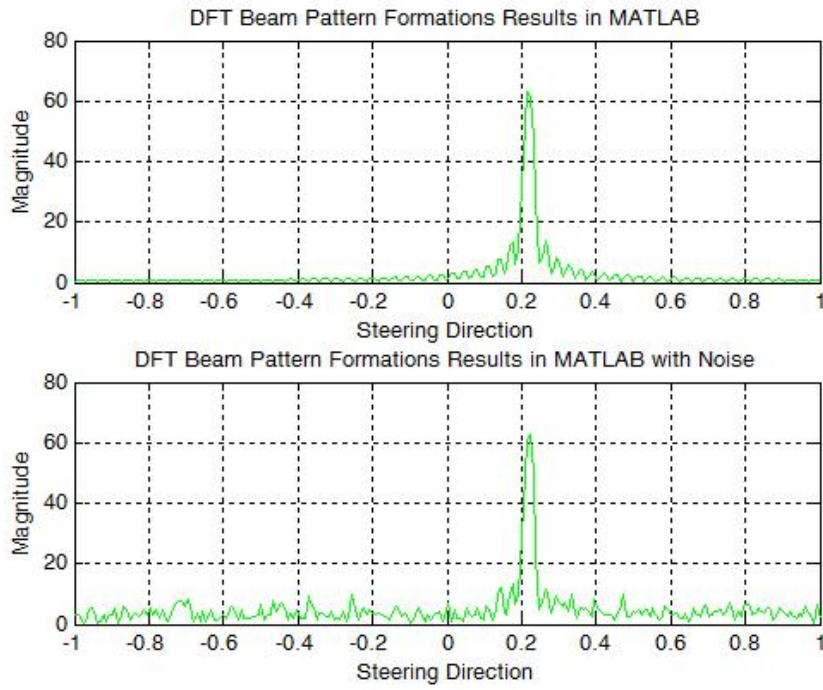


Figure 7.6: SNR Analysis

Figure 7.6 presents a beam pattern formation in MATLAB, without noise (top graph) and with noise (bottom graph), for $A = 1$ and 64 sensors. In this example a SNR between an input vector and noise vector was -8.8528 dB.

Figure 7.7 presents a beam pattern formation in MATLAB, without noise (top graph) and with noise (bottom graph), for $A = 6$ and 64 sensors. In this example a SNR between an input vector and noise vector was -35.5705 dB.

The beam pattern formations obtained in the developing platforms MATLAB, DSP 6713, Gumstix, and pMATLAB, were statistically compared, in order to determine the difference among the data. A single beam pattern formation obtained in MATLAB was defined as a row vector signal $S_{io}[n]$, for $n \in Z_B$, B being the total number of steering directions detected by the linear array. This signal is considered to be the original, ideal beam pattern formation, since MATLAB is considered as the ideal tool by excellence for algorithm development. A single beam pattern formation obtained from the DSP, Gumstix, and pMATLAB, was defined as $S_{id}[n]$, $S_{ig}[n]$, and $S_{ip}[n]$, respectively.

Afterwards, the signals $S_{id}[n]$, $S_{ig}[n]$, and $S_{ip}[n]$ were statistically compared to the ideal signal $S_{io}[n]$ obtained in MATLAB. This was done by first calculating the following differences, with

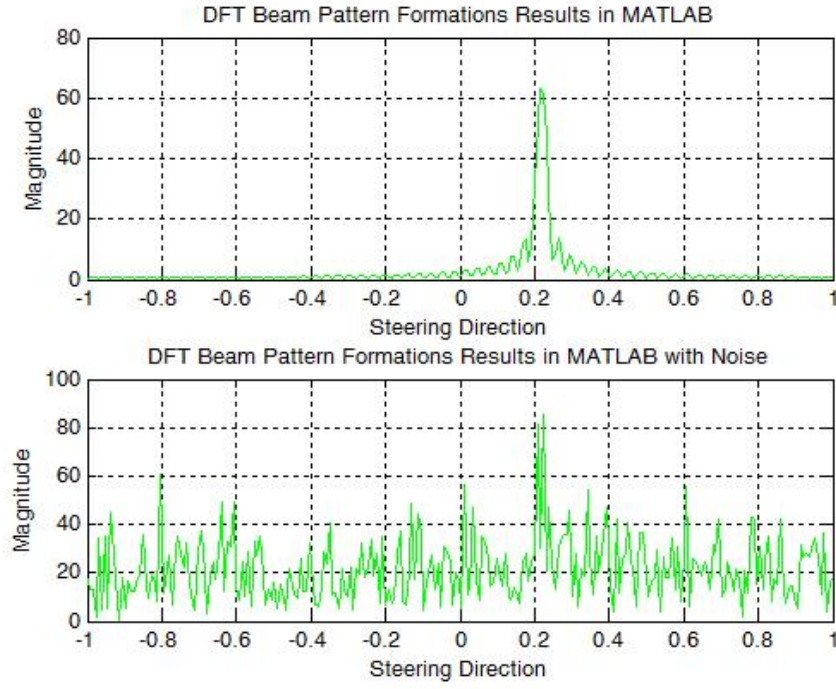


Figure 7.7: SNR Analysis (cont.)

respect to $S_{io} [n]$:

$$\begin{aligned}
 S_{io} [n] - S_{id} [n] &= x_d [n] \\
 S_{io} [n] - S_{io} [n] &= x_{iz} [n] \\
 S_{io} [n] - S_{ig} [n] &= x_g [n] \\
 S_{io} [n] - S_{ip} [n] &= x_p [n]
 \end{aligned} \tag{7.3}$$

The difference between the beam pattern formation in MATLAB $S_{io} [n]$ and the one obtained by the DSP is represented by the signal $x_d [n]$. The difference between the ideal signal $S_{io} [n]$ and itself should result in the row vector of B zeroes, denoted as $x_d [n]$. The signal $x_g [n]$ denotes the difference between the beam pattern formation obtained in MATLAB and by the Gumstix. The difference between the beam pattern formation in MATLAB $S_{io} [n]$ and the one obtained in pMATLAB is represented by the signal $x_p [n]$.

The mean, variance, standard deviation, energy, and power are the signal metrics computed for the different signals representing the beam pattern formations and the differences with respect to the beam pattern formation obtained in MATLAB [23]. The mean of a signal is defined as the signal average value of its samples, and represents the expected value:

$$\mu_x = \left(\frac{1}{B}\right) \sum_{n=0}^{B-1} x[n]. \quad (7.4)$$

The variance of a signal is defined as:

$$\sigma_x^2 = \left(\frac{1}{B}\right) \sum_{n=0}^{B-1} |x[n] - \mu_x|^2. \quad (7.5)$$

The variance is a measure of the signal value $x[n]$ and the expected value or mean μ_x . The standard deviation, denoted as σ is computed as the square root of the variance.

The total energy of a signal $x[n]$ may be defined as:

$$\epsilon_x = \sum_{n=0}^{B-1} |x[n]|^2. \quad (7.6)$$

The average power of a signal $x[n]$ is defined as the energy per sample:

$$P_x = \frac{\epsilon_x}{B} = \left(\frac{1}{B}\right) \sum_{n=0}^{B-1} |x[n]|^2. \quad (7.7)$$

Table 7.1 presents the signal metrics computed for the signals S_{io} , S_{id} , S_{ig} , S_{ip} and the signals representing the difference with respect to the signal S_{io} . For this case, the number of sensors L was 64, assuming that the linear array of sensors received a total of 256 steering directions ($B = 256$).

Table 7.1: Signal Statistical Metrics

	Mean	Std	Var	Power	Energy
x_d	$2.3216e^{-7} + 1.1591e^{-7}i$	$4.1810e^{-6}$	$1.7481e^{-11}$	$1.7480e^{-11}$	$4.4748e^{-9}$
x_g	$2.3216e^{-7} + 1.1591e^{-7}i$	$4.1810e^{-6}$	$1.7481e^{-11}$	$1.7480e^{-11}$	$4.4748e^{-9}$
x_p	0	0	0	0	0
x_{iz}	0	0	0	0	0
S_{io}	$0.9961 - 0i$	7.9375	63.0039	63.7500	$1.6320e^{+4}$
S_{id}	$0.9961 - 0i$	7.9375	63.0039	63.7500	$1.6320e^{+4}$
S_{ig}	$0.9961 - 0i$	7.9375	63.0039	63.7500	$1.6320e^{+4}$
S_{ip}	$0.9961 - 0i$	7.9375	63.0039	63.7500	$1.6320e^{+4}$

From **Table 7.1**, it can be observed that the mean, variance (*var*), and standard deviation (*std*) of the signals x_d , x_g , and x_p resulted to be nearly zero values, as well as the power and


```

for m=1:1:length(B);

    Bk(1,i) = B(m); %Steering Direction
    %Bk = B(1,2)
    for k=1:1:N
        sample_input_matrix_1(k,m) = phi_0*(exp(j*2*pi*((k-1))*Bk(1,i)*d/lambda)); % Origine
        sample_input_matrix_2(k,m) = phi_0*(exp(j*2*pi*((k-1))*Bk(1,i)*d/lambda)) +
        exp(2*pi*j*k*(d/(lambda)))*(random('Normal',0,sqrt(0.1),1,1)); %Signal with noise
    end
    i = i+1;
end

Beam_pattern_1 = fft(sample_input_matrix_1);
Beam_pattern_2 = fft(sample_input_matrix_2);

```

Figure 7.8: DFT Beamforming MATLAB Code

energy. This indicates a very small or nearly no difference among the beam pattern formation obtained from each of the developing platforms.

7.2.2 DFT and Kronecker Beamforming Implementation Results in MATLAB

Acoustic beamforming has been simulated in MATLAB. A variety of tests were conducted in order to determine how the relationship between sensor spacing d and wavelength λ , the number of sensors L , and noise can affect the beamforming operation. Based on the results obtained, it has been found that using a microphone spacing of $d = \frac{\lambda}{2}$, where λ corresponds to the wavelength of the input signal, and incrementing the number of sensors, provided a better beam pattern formation, especially in the presence of noise. Multi-beamforming was first developed in MATLAB, in the AIP laboratory at the University of Puerto Rico, Mayaguez, making use of the Discrete Fourier Transform (DFT) matrix. Afterwards, a DFT Kronecker formulation was designed for the multi-beamforming operation.

Figure 7.8 and **Figure 7.9** presents the core or essence of the beamforming techniques, implemented in MATLAB, using DFT and Kronecker product formulations, respectively. **Figure 7.10** represents a single beam pattern obtained, using DFT Beamforming technique for 32 sensors. The relationship between sensor spacing d and wavelength λ , denoted as the ratio $\frac{d}{\lambda}$ in the signal model $\phi_n(\beta_0) = \phi_0 \cdot e^{j2\pi(\frac{nd}{\lambda}\beta_0)}$ derived in chapter 3 was analyzed in more details. As part of the initial experiments conducted, it was desired to study how this relationship between the sensor spacing and wavelength of the incoming signal can actually affect the detection and quality of the beam patterns obtained from the linear array of L sensors.

Noise was simulated in MATLAB, by using the function $randn(N,M)$, as previously defined, in order to further test the beamforming operation of the linear array. In order to generate the noise signal, arbitrary scalar values were generated with the function $randn$, which were then added to the original input signals. The noise was varied by changing the amplitude of the noise signal.

```

%Permutation Matrix
P_N_S = perm_matrix(N,S);
%Phase or Twiddle Matrix
D_N_R = D_matrix(N,R);
T_N_S = blkdiag(I_R, D_N_R);
%Identity Matrices
I_R = eye(R,R);
I_S = eye(S,S);
I_N = eye(N,N);
I_M = eye(M,M);
%Fourier Transform Matrices
F_R = dftmtx(R);
F_S = dftmtx(S);
U_M = ones(M,1);
%Beam pattern formation Matrix
Beam_pattern_1 = kron(U_M', I_N) * kron(I_M, (kron(F_S, I_R) * T_N_S)) * kron(I_M, (kron(I_S, F_R) * P_N_S));
%sample_input_matrix_1; %Beam pattern of signals from the same plane without noise

```

Figure 7.9: DFT Kronecker Beamforming MATLAB Code

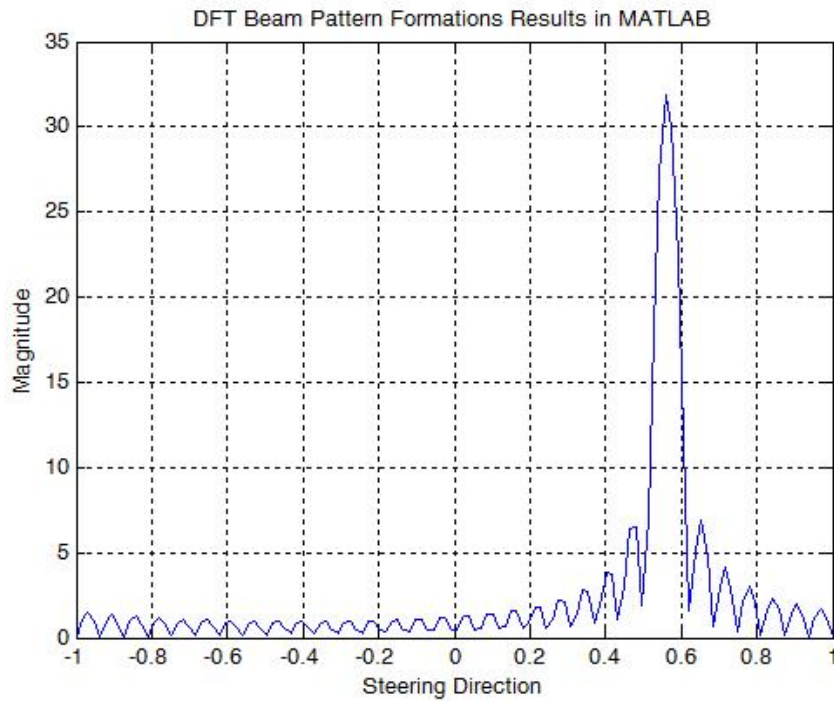
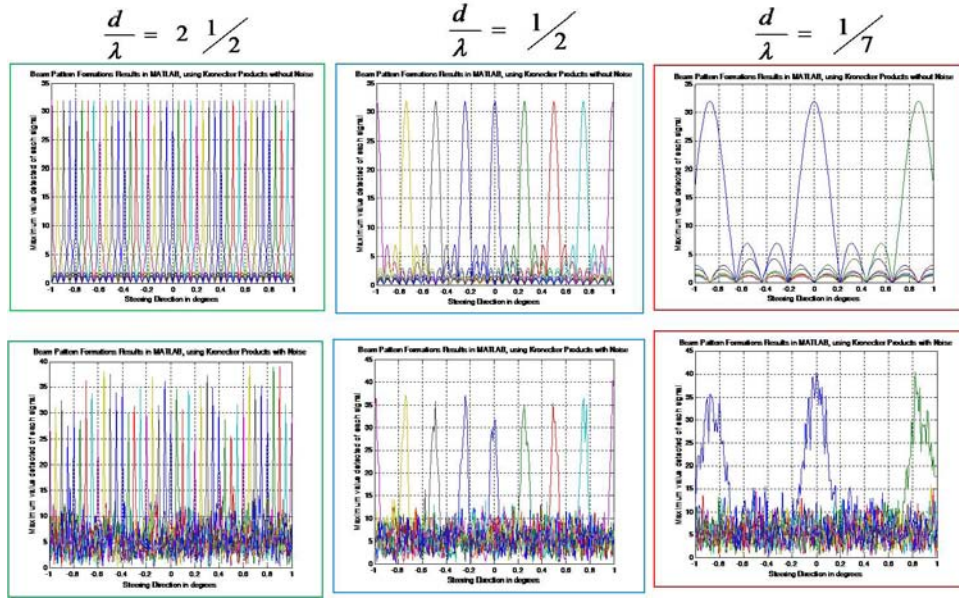


Figure 7.10: DFT Beamforming Pattern Example

In addition, multiple beam pattern formations were considered. Hence, for these initial tests, the ratio $\frac{d}{\lambda}$ was varied in the range of $\frac{d}{\lambda} \leq \frac{1}{2}$ and $\frac{d}{\lambda} \geq \frac{1}{2}$. **Figure 7.11** summarizes the results obtained, considering the case of 32 sensors and Kronecker product formulations, where the 32 sensors were divided in 4 modules, each consisting of 8 sensors, and a total of 201 steering directions were considered. Thus, for this case, as explained in section 7.1, 8 beam pattern formations were obtained of length 201 values, resulting from the linear combination of the 4 modules of 8 sensor each: As it can be shown in **Figure 7.11**, for a ratio $\frac{d}{\lambda} > \frac{1}{2}$, indicating the sensor spacing is larger than the wavelength, and taking as example $\frac{d}{\lambda} = 2\frac{1}{2}$, each of the 8 beam patterns obtained resulted to have more than one main lobe in its respective beam

Figure 7.11: Analyzing relationshipsensor spacing d and wavelength λ

pattern, making it difficult to detect the angle of incidence of the dominant incoming signal at each of the 8 linearly combined beam attens, especially in the presence of noise, as shown at the bottom of **Figure 7.11**.

On the other hand, for case of $\frac{d}{\lambda} = \frac{1}{2}$, indicating that sensor spacing d is half of the wavelength λ , each of the 8 beam patterns were clearly formed, each resulting to have a single main lobe, making it easy to identify the steering direction and hence, the angle of incidence of the dominant incoming signal, from each beam pattern obtained. Even in the presence of noise, the main lobe of each pattern formation suffered little distortion, making it possible to still identify the steering direction and incidence angle of the dominant signal.

For the case of $\frac{d}{\lambda} > \frac{1}{2}$, indicating that the wavelength λ is much larger than the sensor spacing d , and taking as an example the case for $\frac{d}{\lambda} > \frac{1}{7}$ the following was observed; not all of the beam patterns were detected, and the three main lobes that were detected resulted to be wider. Hence for the case $\frac{d}{\lambda} > \frac{1}{2}$, the main lobes of some the the beam patterns are not detected by some of the sensors, making the linear array seem blind at certain angles.

From these intial tests, it was shown that using the relationship $\frac{d}{\lambda} = \frac{1}{2}$ provided the best beam pattern formations, in the sense that the single main lobe of each pattern can easily be detected, so as to determine the direction of the dominant signal.

The next experiments conducted, having determined previously that $\frac{d}{\lambda} = \frac{1}{2}$ is the best guideline to use for the design of the linear array, consisted in determining how the number of sensors can further affect the beamforming operation, also considering the presence of noise. Subsequent tests consisted of analyzing single beam pattern, using Kronecker product formulations, for

32, 64, 128, and 256 sensors, assuming that all signals originate from the same wave plane characterized by λ . For each number of sensors, the total number of sensors was divided into M modules, such that N , the number of sensors in each module resulted to be 4, for the purpose of simplicity.

For these set of tests, the following notation is reviewed:

- L denotes the number of sensors.
- M denotes the number of modules
- N denotes the length of the FFT applied to each module ($N = 4$)
- $\frac{d}{\lambda} = \frac{1}{2}$ denotes the sensor spacing-wavelength ratio
- The range of steering angles was divided into intervals of size .01, for a total of 201 intervals between $-\frac{\pi}{2}$ and $\frac{\pi}{2}$.

The following sets of four beam pattern formations were obtained for 32, 64, 128, and 256 sensors, where the top plots represents the beam patterns without noise, and the bottom plots are the beam patterns with noise added. Based on these results, it could be observed that, as the number of sensors in the array incremented, a beam pattern with a more defined single main lobe could be perceived, even in the presence of noise, with less distortion in the lobe (see **Figures 7.12, 7.13, 7.14, and 7.15**).

After conducting the single beam pattern analysis, the number of sensors was once again varied, in powers of two, for 32, 64, 128, and 256 sensors, considering for this case, multiple beam pattern formations, as depicted in **Figures 7.16, 7.17, 7.18, and 7.19**). For these tests, Kronecker product formulations were used, such that the number of sensors N for each module was 8, resulting in 8 linearly combined beam pattern formations.

Once again, these tests showcase that as the number of sensors increments, the main lobe of each beam pattern is made more distinguished and pronounced, even when there is noise present in the input signals.

In summary, the implementations of the DFT and Kronecker beamforming algorithms initially in MATLAB, served as an ideal tool and model by excellence for analyzing and determining how the relationship between the sensor spacing and wavelength, and how the number of sensors affect the beamforming operation of a linear sensor array. As the results have shown, using a sensor spacing of $d = \frac{\lambda}{2}$ and incrementing the number of sensors help obtain better beam pattern formations, including for the situation when there is noise or interference present in the incoming signals to the linear array.

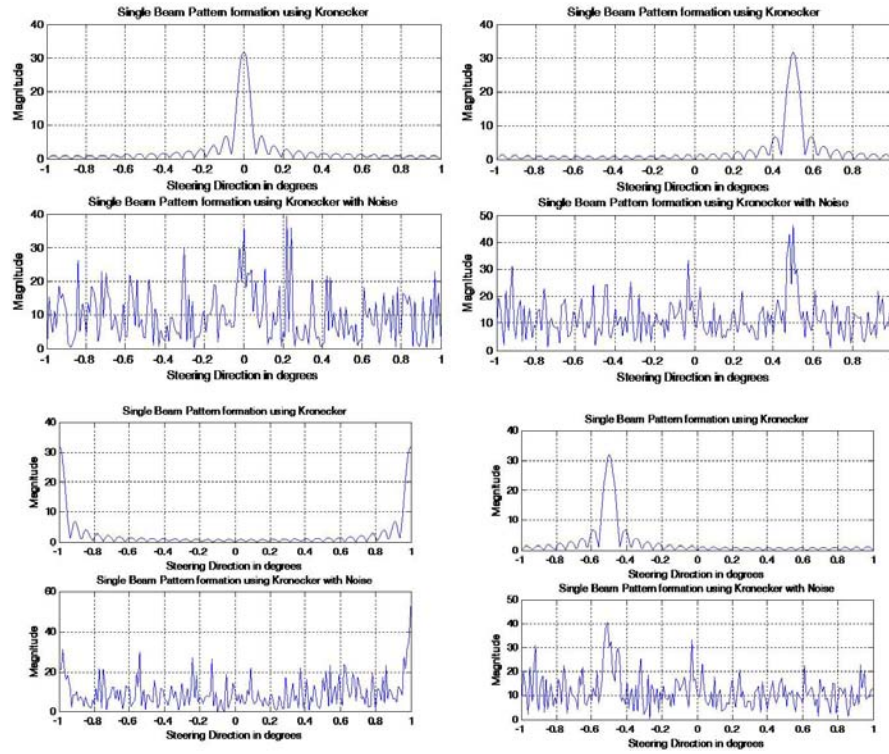


Figure 7.12: Single Beam Pattern Formations for 32 sensors

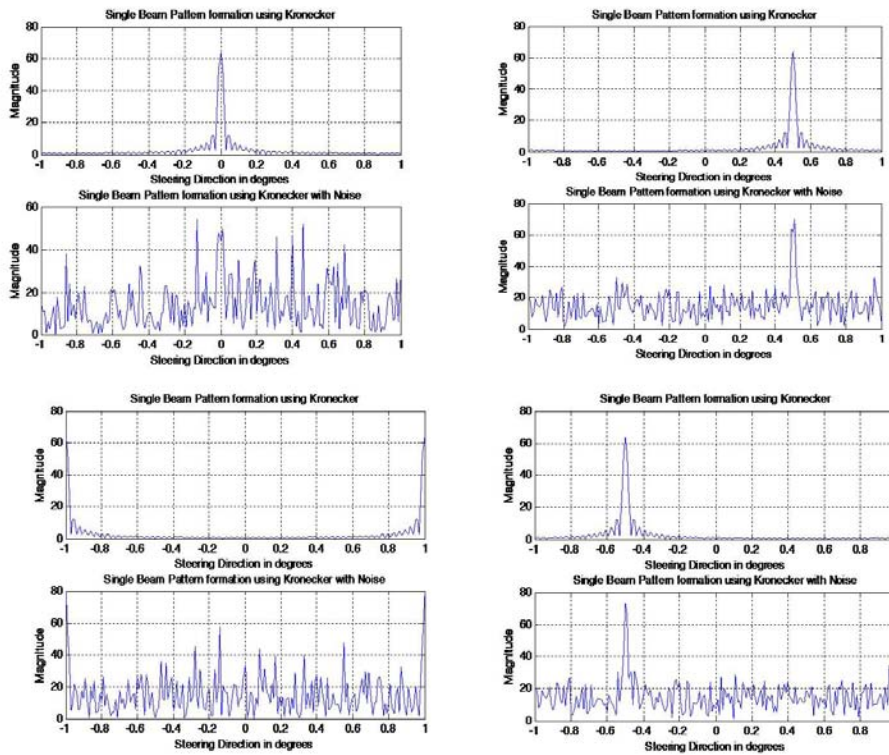


Figure 7.13: Single Beam Pattern Formations for 64 sensors

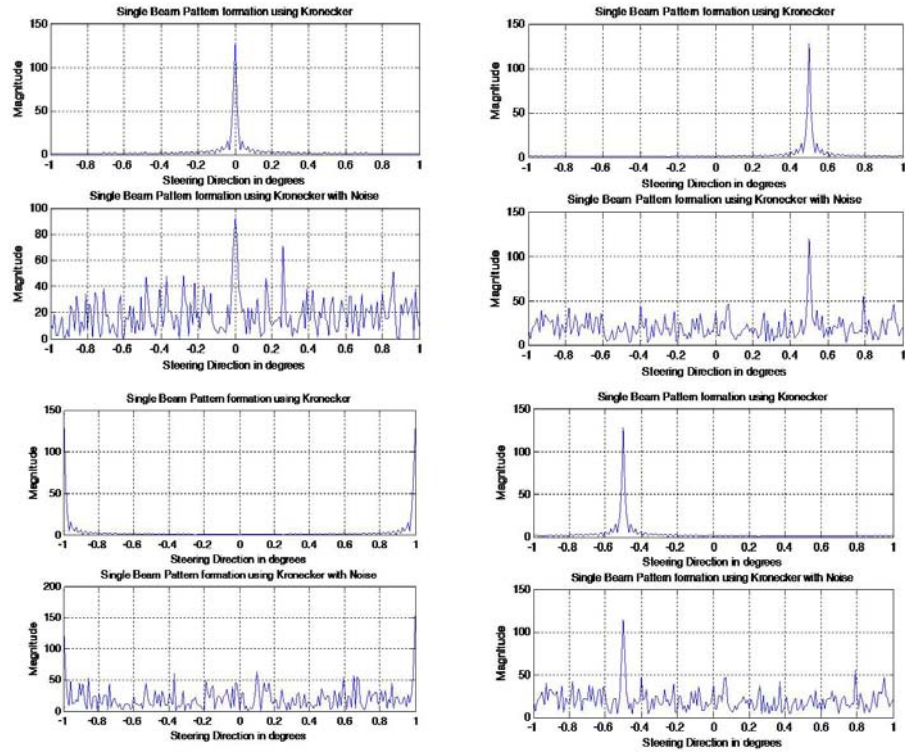


Figure 7.14: Single Beam Pattern Formations for 128 sensors

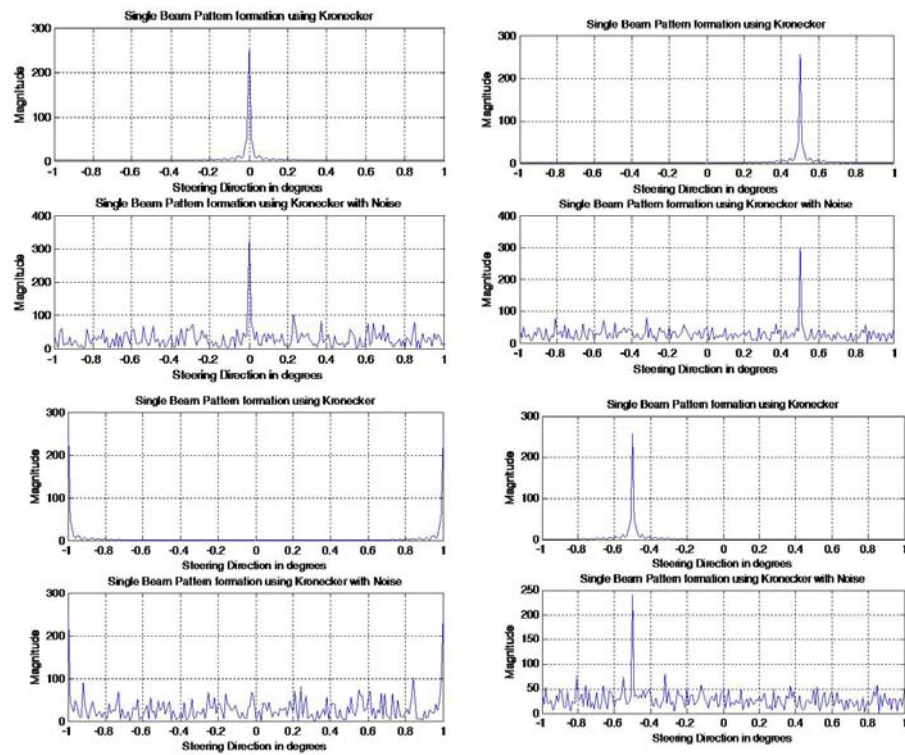


Figure 7.15: Single Beam Pattern Formations for 256 sensors

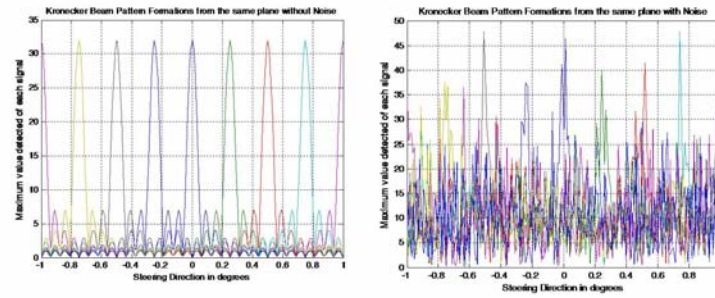


Figure 7.16: Multiple Beam Pattern Formations for 32 sensors

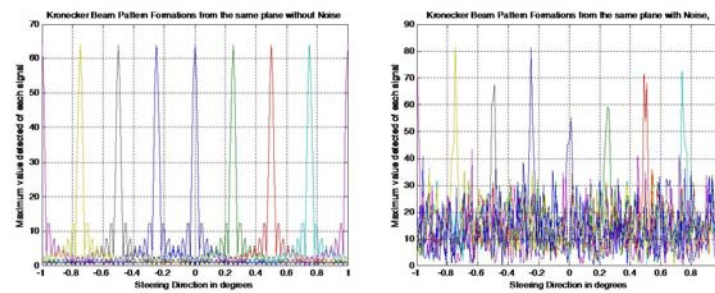


Figure 7.17: Multiple Beam Pattern Formations for 64 sensors

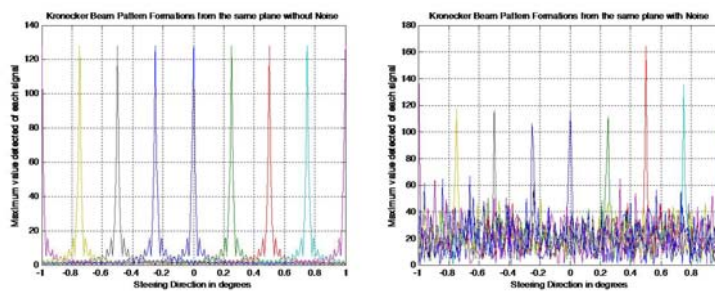


Figure 7.18: Multiple Beam Pattern Formations for 128 sensors

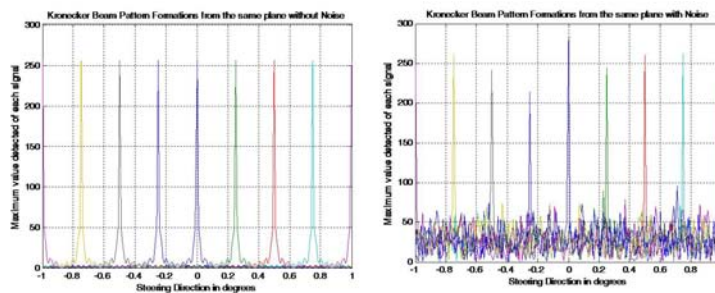


Figure 7.19: Multiple Beam Pattern Formations for 256 sensors

7.2.3 DFT and Kronecker Beamforming Implementation Results on DSP 6713

After implementing the DFT and Kronecker Beamforming algorithms in MATLAB, both beamforming operation techniques were developed in C language, for the implementation of such

algorithms for the DSP 6713. The ideal sensor spacing guideline $d = \frac{\lambda}{2}$ obtained from the series of experimental tests that were conducted in MATLAB was incorporated into implementing these algorithms on the DSP. Here the Kronecker product formulation results obtained on the DSP are presented.

The $L \times B$ complex input matrix was simulated in MATLAB, creating two **.dat** files, corresponding to the real and imaginary parts of the input matrix. Such files served as the input files for the DSP program application, developed in Code Composer Studio. **Figure 7.20** depicts the implementation procedure. The output **.dat** files of the real and imaginary parts

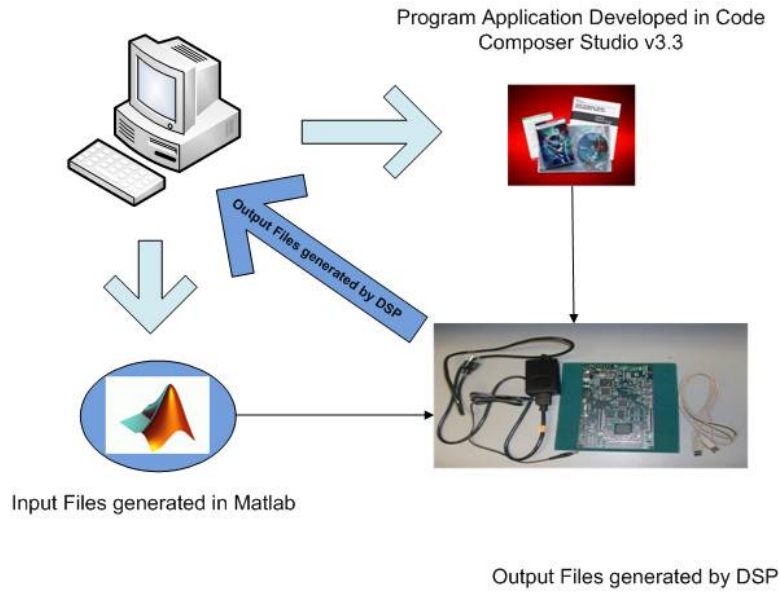


Figure 7.20: Beamforming Algorithm Implementations on the DSP

of the beam patterns generated by the DSP were read in MATLAB in order to obtain the corresponding graph of the beam patterns. **Figure 7.21** and **Figure 7.22** presents the multiple beam pattern formations for 32 and 64 sensors. As can be shown from **Figures 7.21** and **7.22**, similar results were obtained on the DSP, as in MATLAB.

7.2.4 Kronecker DFT Beamforming Implementation Results on Gumstix Verdex

After implementing the DFT and Kronecker Beamforming algorithms in MATLAB, and for the DSP 6713, DFT Kronecker beamforming operation developed in C language, was executed on the Gumstix Verdex. Here the Kronecker product formulation results obtained on the Gumstix are presented.

The $L \times B$ complex input matrix was again simulated in MATLAB, creating two **.dat** files, corresponding to the real and imaginary parts of the input matrix. Such files served as the input

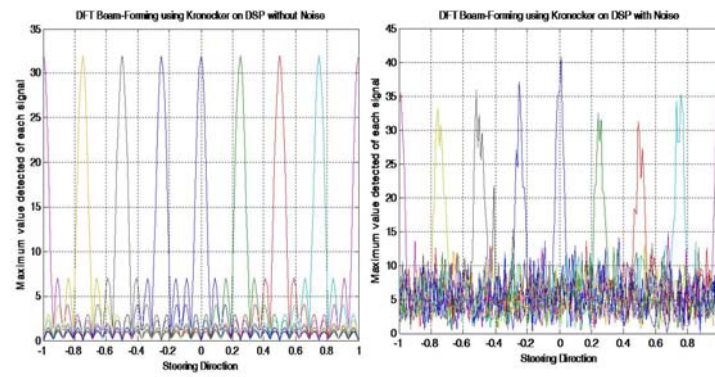


Figure 7.21: DSP Multiple Beam Pattern Formations for 32 sensors

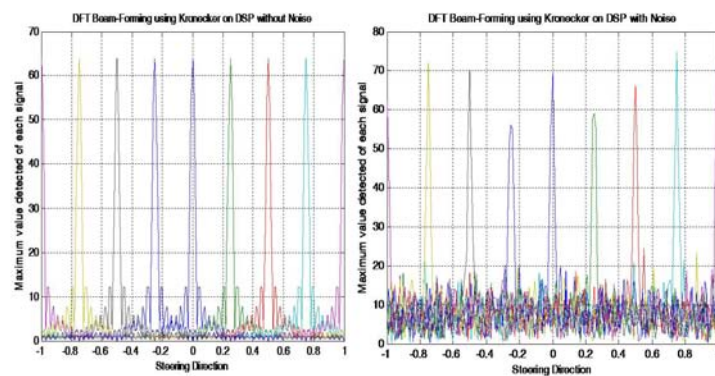


Figure 7.22: DSP Multiple Beam Pattern Formations for 64 sensors

files for the beamforming program application project, which was compiled on the MSN. Once the project was compiled, the executable output file, along with the input files, were sent to the Gumstix, via wireless communication. The tests conducted on the Gumstix also involved fixing the number of points received by each sensor, while varying the number of sensors, from 32 to 8192, as powers of two. **Figures 7.23** and **7.24** presents multiple beam pattern formations obtained by using a fixed number of 256 points received at each sensor, for 32 and 64 sensors, respectively.

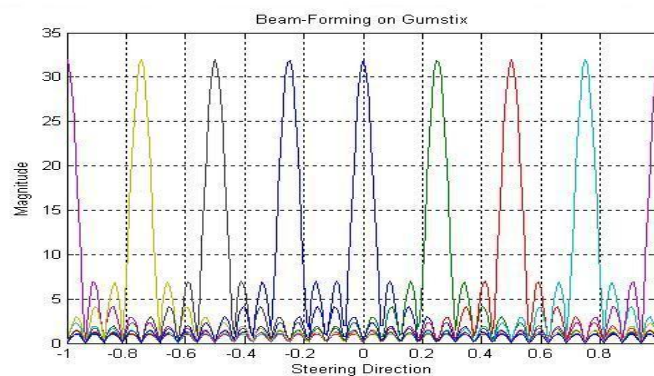


Figure 7.23: Gumstix Multiple Beam Pattern Formations for 32 sensors

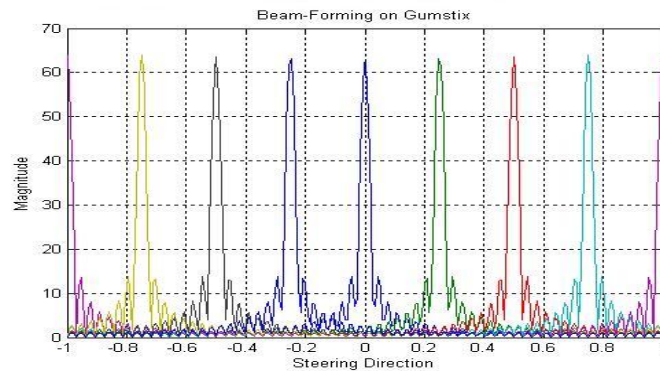


Figure 7.24: Gumstix Multiple Beam Pattern Formations for 64 sensors

7.2.5 DFT and Kronecker Beamforming Implementation Results in pMATLAB

Previous results obtained in MATLAB, on the DSP, and Gumstix have shown the effectiveness of using Kronecker product formulations as a method for mapping the beamforming operation onto a parallel computing architecture environment. In order to integrate further the concept of multicore architecture to the beamforming operation, DFT and Kronecker beamforming techniques were also implemented using the pMATLAB as an ideal tool for parallel simulation environment. **Figure 7.25** depicts the multicore simulation environment which pMATLAB provides for parallel program applications. The implementation of the DFT and Kronecker

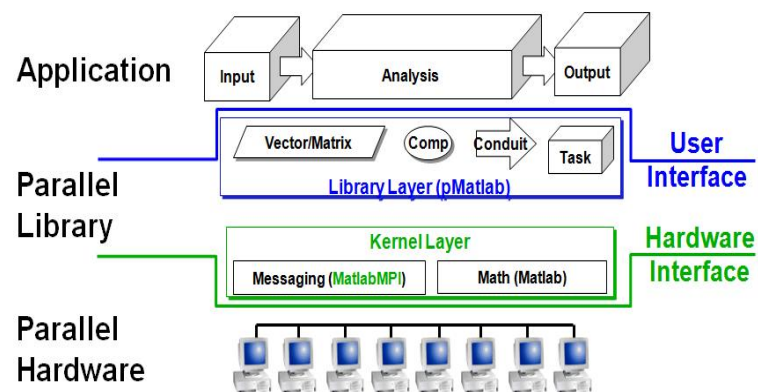


Figure 7.25: pMATLAB Multicore Simulation Environment, *courtesy of Kepner, MIT Lincoln Lab.*

beamforming algorithms in pMATLAB produced similar results, as acquired in MATLAB and from the DSP 6713. **Figure 7.26** presents a single beam pattern formation, for 64 sensors, when executing the program application in a multicore environment. **Figure 7.27** presents multiple beam pattern formations when executing the Kronecker beamforming application in pMATLAB, also for 64 sensors. For both cases, 256 steering directions were considered; that is, each sensor received 256 points of input data.

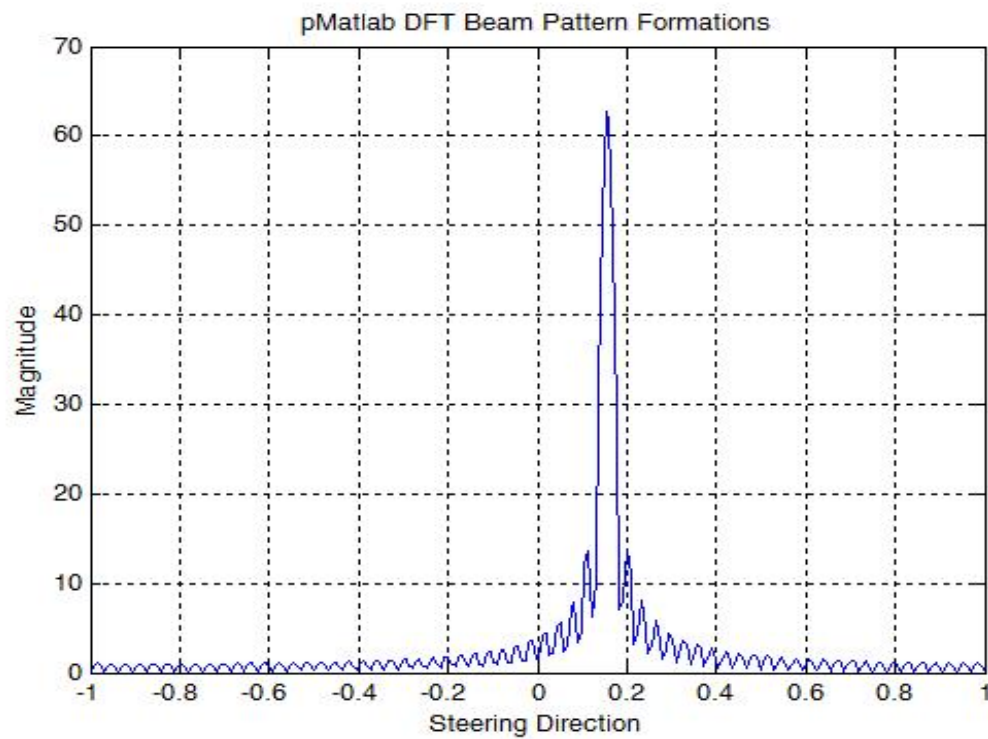


Figure 7.26: DFT Beam Pattern Formation in pMATLAB for 64 sensors

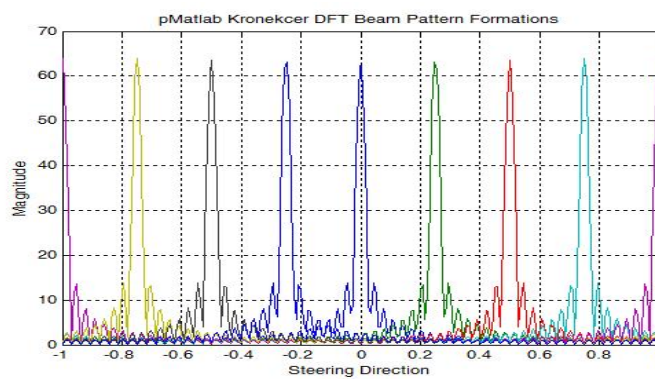


Figure 7.27: Kronecker Beam Pattern Formation in pMATLAB for 64

In addition to implementing the beamforming techniques in pMATLAB, a speedup analysis was also conducted. In general, this analysis consisted of fixing the number of steering directions, or the number of data points that each sensor in the linear array receives. The number of sensors was varied as powers of two, from 16 to 8192 sensors. For each number of sensors, the beamforming parallel application program was executed on 1,2,4,8,16,32, and 64 parallel simulated processors. **Figure 7.28** demonstrates the speedup obtained, considering the case of the linear sensor array being capable of detecting 256 steering directions.

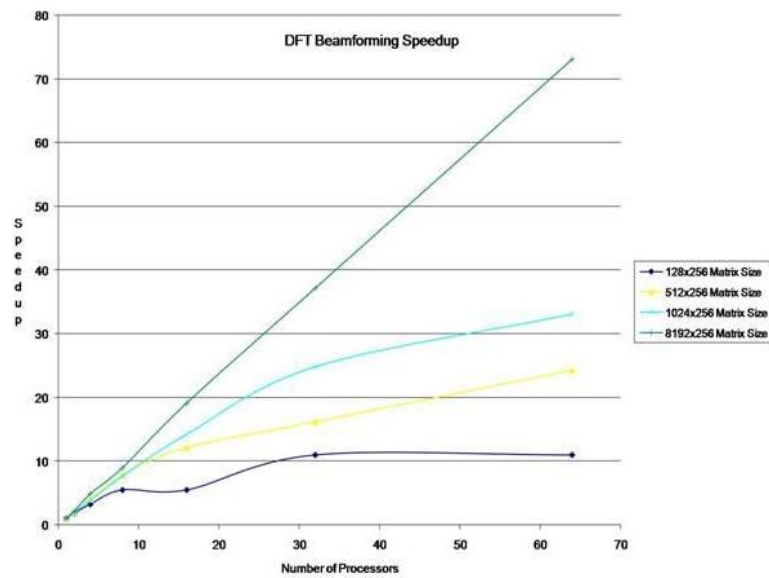


Figure 7.28: Speedup Analysis in pMATLAB for 256 Steering Directions

As can be observed from **Figure 7.28**, the beamforming program application does indeed exhibit a high parallel, linear characteristic in its structure. Also, it was observed throughout this analysis that, as the number of sensors and the number of steering directions that the linear array is capable of detecting incremented, the parallel nature of the program application was conserved for up to more than 16 processors, without suffering early degradation, as in other applications.

7.2.6 DFT Beamforming Platforms Comparison

The Kronecker DFT beamforming technique was implemented and evaluated on the following platforms:

- DSP C6713
- Gumstix Verdex

- pMATLAB

In addition, the DFT beamforming technique was also evaluated in pMATLAB. The function *kron* used in pMATLAB resulted to be implemented not so efficiently for a parallel environment, from observing the large execution times obtained, especially as the number of processors incremented. On the other hand, the DFT beamforming technique provided much better execution times, also in comparison with the Gumstix and DSP. The reason for this is that the function *fft* is implemented in pMATLAB to permit the mapping and distribution of the columns of the input matrix among the processors executing in parallel. Hence, as the number of processors increased from 1,2,4,8,16,32, up to 64, better execution times were obtained.

	Kronecker DSP C6713	Kronecker Gumstix Verdex	Kronecker pMATLAB	DFT pMATLAB
Number of sensors	Execution Time (ms)	Execution Time (ms)	Execution Time (ms)	Execution Time (ms)
32	70.4	29.76	7.18, 413.44	0.16, 0.3
64	140.45	62.75	7.6, 46.76	0.2, 0.4
128	281.36	125.59	8.72, 50.3	0.1, 1.1
256	562.45	315.06	11.54, 163	0.12, 2.3
512	1124.54	673.28	28.92, 97.48	0.2, 4.86
1024	2221.55	1436.65	88.275, 166.66	0.3, 9.94
2048	4442.815	2711	296.7, 486.3	0.5, 20.88
4096	—	5531.61	1219.44, 1469.4	0.9, 62.76
8192	—	22234.99	5138.05, 531.45	1.86, 135.98

Figure 7.29: DFT Beamforming Platforms Comparison

Chapter 8

Conclusions and Future Work

Discrete Fourier Transform (DFT) beamforming algorithms have been demonstrated to effectively obtain efficient beamforming operation results for linear sensor array configurations. The DFT beamforming algorithms formulated using Kronecker products algebra have been proven to derive expressions for the beamforming operations, for the purpose of mapping these operations onto a parallel architecture.

The development effort used the MATLAB numeric computation and software visualization package, as an ideal simulated environment for signal processing applications. By initially implementatng the DFT and Kronecker beamforming algorithms in MATLAB, this served as an ideal tool and model by excellence for studying how the sensor spacing and wavelength and the number of sensors could affect the beamforming operation of a linear sensor array. By using a sensor spacing of $d = \frac{\lambda}{2}$ and incrementing the number of sensor, optimal beam pattern formations were obtained, even in the presence of noise.

The DFT Beamforming algorithms were sucessfulling implemented on the DSP 6713 from Texas Instruments, obtaining similar results as in MATLAB. Using pMATLAB as an ideal parallel programming modeling environment, named pMATLAB, the computational performance of parallel implementation techniques for beamforming operation demonstrated the high linearity that the DFT beamforming algorithms may possess in a multicore architectural environment. It was shown that these DFT Beamforming techniques can exhibit high parallel speedup for more than 16 processors, as the number of sensors in the array, and the number of steering directions that can be detected are incremented.

As future work, the TMS320C6474 multicore DSP from Texas Instruments will be considered as the actual multicore architectural platform for implementing the DFT beamforming algorithms, due to potential characteristics that this architecture exhibits, such as high performance delivery at 3 GHz, through the integration of three DSP cores. The parallel program

pMATLAB should be continued to be utilized as the ideal parallel simulation environment for further testing and implementation work.

Appendix A

MATLAB DFT Beamforming Code

```
%Multi- Linear beamforming implementation in MATLAB, using DFT

% The number of steering angles considered is the number of Sensors in the
% system
clc;
clear all;
close all;

M=2^6;%Number of input sampled vectors and number of steering angles
N =2^6; %Number of sensors in linear array

d= 2;
lambda =2*d;
phi_0= 1;%Initial Amplitude of signal
A = 5; %Amplitude of noise signal
%B=-1:10^(-2):1;          % -1<(B=SIN(THETA))<1
NumberOfPoints = 256;
B=-1:2/(NumberOfPoints-1):1
i=1;
for m=1:1:length(B);
Bk(1,i) = B(m);%Steering Direction
    for k=1:1:N
sample_input_matrix_1(k,m) = phi_0*(exp(j*2*pi*((k-1))*Bk(1,i)*d/lambda));% Original Signal
        end
        i = i+1;
    end
end
```



```
noise_signal = A*(0.5*randn(N,NumberOfPoints));
sample_input_matrix_2 = sample_input_matrix_1+ noise_signal;%Signal with noise

Beam_pattern_1 = fft(sample_input_matrix_1);
Beam_pattern_2 = fft(sample_input_matrix_2);
    %Steering angles in radians

    %Plotting the beam pattern formed for some of the input vectors, where
    %each column of the matrix Beam_pattern is a beam pattern of the
    %corresponding input vector

channel = 8;

plot(B,abs(Beam_pattern_1(channel,:)), 'g')
    title('DFT Beam Pattern Formations Results in MATLAB');
xlabel('Steering Direction');
ylabel('Magnitude');
grid

figure

plot(B,abs(Beam_pattern_2(channel,:)), 'g')
title('DFT Beam Pattern Formations Results in MATLAB with Noise');
xlabel('Steering Direction');
ylabel('Magnitude');
grid
```

Appendix B

MATLAB DFT Kronecker Beamforming Code

```
%Multi- Linear beamforming implementation in MATLAB, based on the concepts
%using kroneckers

% The number of steering angles considered is the number of Sensors in the
% system
%clc;
clear all;
close all;

L = 64 % Number of sensors
M= 8%Number of Modules
N =8; %Number of sensors in linear array
d= 1;% distance between sensors
lambda =2*d;% wavelength
phi_0= 1;%Initial Amplitude of signal
A = 5; %Amplitude of noise signal
NumberOfPoints = 256;
B=-1:2/(NumberOfPoints-1):1
%B = -1:10^(-2):1;
i=1;
for m=1:1:length(B);
    Bk(1,i) = B(m);%Steering Direction
    for k=1:1:L
```

```

sample_input_matrix_1(k,m) = phi_0*(exp(j*2*pi*((k-1))*Bk(1,i)*d/lambda));% Original Signal
end
end
noise_signal = A*(0.5*randn(L,NumberOfPoints));
sample_input_matrix_2 = sample_input_matrix_1+ noise_signal;%Signal with noise

%Beam_pattern = (B_D0A*sample_input_matrix);
U_M = ones(M,1);
I_N = eye(N,N);
I_M = eye(M,M);

Beam_pattern_1= kron(U_M', I_N )*kron(I_M,dftmtx(N) )*sample_input_matrix_1; %Beam pattern
Beam_pattern_2= kron(U_M', I_N )*kron(I_M,dftmtx(N) )*sample_input_matrix_2; %Beam pattern

%Beam_pattern_1 = (1/L)*Beam_pattern_1;
%Beam_pattern_2 = (1/maxNoise)*Beam_pattern_2;
%Plotting the beam pattern formed for some of the input vectors, where
%each column of the matrix Beam_pattern is a beam pattern of the
%corresponding input vector
figure
channel=1:1:N
plot(B,abs(Beam_pattern_1(channel,:)))
title('Kronecker DFT Beam Pattern Formations in MATLAB');
xlabel('Steering Direction');
ylabel('Normalized Magnitude');
grid
figure;

plot(B,abs(Beam_pattern_2(channel,:)))
title('Kronecker DFT Beam Pattern Formations in MATLAB with Noise');
xlabel('Steering Direction');
ylabel('Normalized Magnitude');
grid
realIn=real(Beam_pattern_1);
imagIn=imag(Beam_pattern_1);

save Beam_pattern_real.dat realIn -ascii

```

```
save Beam_pattern_image.dat imagIn -ascii
```

Appendix C

DSP DFT Beamforming Code

```
#include "dsk6713_aic23.h"
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate

#include <stdio.h>
#include <limits.h>
#include <math.h>
#include <stdlib.h>
#include <ctype.h>

#define B 256      /*Number of points or steering directions received at each sensor*/
#define N 32      // Number of sensors
#define D 2      // Distance between sensors
#define L 2*D      //lambda
#define d_L 1/L_d
#define PI 3.14159265358979
#define DELTA (2*PI)

typedef struct {float real,imag;} COMPLEX;
void FFT(COMPLEX *Y);
//void FFT(COMPLEX *Y, int n);      //FFT prototype
COMPLEX input_vectors[N][B];
COMPLEX Beam_pattern[N][B];
```

```
COMPLEX w[N];
COMPLEX x[N];

short iTwid[N/2];

float beta,a,b;
FILE *index;

void input_vector_generation(){

short m,k;

index=fopen("sample_input_real_without_noise.h","r");

if((index)==NULL) {
puts("File could not be open");
exit(-1);
}

for(m = 0; m < N; m++){
for(k = 0; k < B; k++){
fscanf(index, "%f",&input_vectors[m][k].real);

}

}

fclose(index);

index=fopen("sample_input_imag_without_noise","r");

if((index)==NULL) {
puts("File could not be open");
exit(-1);
```

```

}

for(m = 0; m < N; m++){
for(k = 0; k < B; k++){
fscanf(index, "%f",&input_vectors[m][k].imag);

}

}

fclose(index);
//exit(-1);
}

void Beam_pattern_generation()
{ int col_input_matrix,i,j;

    COMPLEX x_col[N];
    for(col_input_matrix = 0; col_input_matrix < B; col_input_matrix++){

        for (i=0;i<N;i++){
            x_col[i] = input_vectors[i][col_input_matrix];
        }
        FFT(x_col,N);

        for (j=0;j<N;j++){
            Beam_pattern[j][col_input_matrix].real = x_col[j].real;
            Beam_pattern[j][col_input_matrix].imag = x_col[j].imag;
        }

    }

}

void main(){

    int m,k,doblepts;

```

```

//set up array of twiddle factors

input_vector_generation();
    doblepts=2*N;

    for(k = 0;k<N;k=k+1){
        w[k].real = cos((DELTA*k)/(doblepts));
        //printf("%.5f",Beam_matrix[m-1][k-1].real );
        w[k].imag =-sin((DELTA*k)/(doblepts));
        //printf("%.5f\n",Beam_matrix[m-1][k-1].imag );
    Beam_pattern_generation();
    index = fopen("Beam_pattern_real.dat","w");
        for (k = 0;k<M;k++){
            for(m = 0;m<B;m++){
fprintf(index,"% .2f\n",Beam_pattern[k][m].real);

        }
    }
fclose(index);

index = fopen("Beam_pattern_image.dat","w");

    for (k = 0;k<M;k++){
        for(m = 0;m<B;m++){

fprintf(index,"% .2f\n",Beam_pattern[k][m].imag);

        }
    }
fclose(index);
puts("done");

}

```



```
//FFT.c  C callable FFT function in C

#define PTS 8    // # of points for FFT
#define D 5
#define Lambda (2*D)
typedef struct {float real,imag;} COMPLEX;
extern COMPLEX w[PTS];          //twiddle constants stored in w

void FFT(COMPLEX *Y, int N)      //input sample array, # of points
{
    COMPLEX temp1,temp2;          //temporary storage variables
    int i,j,k;                    //loop counter variables
    int upper_leg, lower_leg;     //index of upper/lower butterfly leg
    int leg_diff;                 //difference between upper/lower leg
    int num_stages = 0;           //number of FFT stages (iterations)
    int index, step;

    int doblepts = 2*PTS;

                                //index/step through twiddle constant
    i = 1;                       //log(base2) of N points= # of stages
    do
    {
        num_stages +=1;
        i = i*2;
    }while (i!=N);
    leg_diff = N/2;              //difference between upper&lower legs

    step =doblepts/N;             //step between values in twiddle.h
    for (i = 0;i < num_stages; i++) //for N-point FFT
    {
        index = 0;
        for (j = 0; j < leg_diff; j++)
        {
            for (upper_leg = j; upper_leg < N; upper_leg += (2*leg_diff))
            {
                lower_leg = upper_leg+leg_diff;
                temp1.real = (Y[upper_leg]).real + (Y[lower_leg]).real;

```

```

        temp1.imag = (Y[upper_leg]).imag + (Y[lower_leg]).imag;
        temp2.real = (Y[upper_leg]).real - (Y[lower_leg]).real;
        temp2.imag = (Y[upper_leg]).imag - (Y[lower_leg]).imag;
        (Y[lower_leg]).real = temp2.real*(w[index]).real
-temp2.imag*(w[index]).imag;
        (Y[lower_leg]).imag = temp2.real*(w[index]).imag
+temp2.imag*(w[index]).real;
        (Y[upper_leg]).real = temp1.real;
        (Y[upper_leg]).imag = temp1.imag;
    }
    index += step;
}
leg_diff = leg_diff/2;
step *= 2;
}
j = 0;
for (i = 1; i < (N-1); i++)    //bit reversal for resequencing data
{
k = N/2;
while (k <= j)
{
    j = j - k;
    k = k/2;
}
j = j + k;
if (i<j)
{
    temp1.real = (Y[j]).real;
    temp1.imag = (Y[j]).imag;
    (Y[j]).real = (Y[i]).real;
    (Y[j]).imag = (Y[i]).imag;
    (Y[i]).real = temp1.real;
    (Y[i]).imag = temp1.imag;
}
}
return;
}
}

```

Appendix D

DSP Kronecker DFT Beamforming Code

```
#include "dsk6713_aic23.h"

Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate

#include <stdio.h>
#include <limits.h>
#include <math.h>
#include <stdlib.h>
#include <ctype.h>

#define L 64
#define M 8 //Number of FFT modules to compute
#define N 8// Length of each FFT module
#define B 201// Total number of input vectors defined for  $-1 < \sin(\theta) < 1$ 
#define PI 3.14159265358979
#define DELTA (2*PI)

typedef struct {float real,imag;} COMPLEX;
void FFT(COMPLEX* , int );
//void FFT(COMPLEX *Y, int n);      //FFT prototype
```

```

COMPLEX input_vectors[L][B];
COMPLEX DFT_modules_matrix[L][B];
COMPLEX Beam_pattern[N][B];
COMPLEX x_L[L];
COMPLEX x_module_N[N];

COMPLEX w[N];

FILE *index;

void input_vector_generation(){

int m,k;

/*for (k = 0;k<N;k++){
    for(m = 0;m<B;m++){
        input_vectors[k][m].real=input_matrix[k][m];
        input_vectors[k][m].imag=input_matrix_imag[k][m];
    }
}

*/

index=fopen("sample_input_real_without_noise.h","r");

if((index)==NULL) {
puts("File could not be open");
exit(-1);
}

for(m = 0; m < L; m++){
for(k = 0; k < B; k++){
fscanf(index, "%f",&input_vectors[m][k].real);

```

```

}

}

fclose(index);

index=fopen("sample_input_imag_without_noise.h", "r");

if((index)==NULL) {
puts("File could not be open");
exit(-1);
}

for(m = 0; m < L; m++){
for(k = 0; k < B; k++){
fscanf(index, "%f",&input_vectors[m][k].imag);

}

}

fclose(index);
//exit(-1);
}

void Twiddle_factors_generation(){
int k, doblepts;
doblepts=2*N;

for(k = 0;k<N;k=k+1){

w[k].real = cos((DELTA*k)/(doblepts));
//printf("%.5f",Beam_matrix[m-1][k-1].real );
w[k].imag =-sin((DELTA*k)/(doblepts));

```

```

        //printf("%.5f\n",Beam_matrix[m-1][k-1].imag );
    }

}

void Beam_Pattern_Init(){
    int i,j;

    for(i=0;i<N;i++){
        for(j=0;j<B;j++){
            Beam_pattern[i][j].real=0;
            Beam_pattern[i][j].imag=0;
        }
    }

}

void DFT_M_Modules_generation()
{
    short i,j,k,p,b,c,n_index;

    for(i=0;i<B;i++) {
        b=0; // Marks the current position in each input vector

        // Processing each input vector taken as a column of the matrix
        for(j=0;j<L;j++){
            x_L[j] = input_vectors[j][i];
        }

        // Divide each input vector into M modules
        for(k=0;k<M;k++){
            for(p=0;p<N;p++){
                x_module_N[p] =x_L[b];
                b++;
            }

            FFT(x_module_N,N);
            n_index=b-N; // To place each DFT module currently in the output matrix
            for(c=0;c<N;c++){
                DFT_modules_matrix[n_index][i]= x_module_N[c];
                n_index++;
            }
        }
    }
}

```

```

    }

    }

}

}

void Linear_Combination_DFT_Modules(){
    short i,j,k,m;

    //Linear combination of the M DFT Modules
    for(i=0;i<N;i++){
        for(j=0;j<B;j++){
            for(k=0;k<=L-N;k=k+N){
m =i+k;
        Beam_pattern[i][j].real = Beam_pattern[i][j].real+ DFT_modules_matrix[m][j].real;
        Beam_pattern[i][j].imag = Beam_pattern[i][j].imag+ DFT_modules_matrix[m][j].imag;
            }

        }

    }

}

void main(){

    int m,k;

    //set up array of twiddle factors

```

```
input_vector_generation();
puts("Input vector Matrix generated");
Twiddle_factors_generation();
puts("Twiddle Factors generated");
DFT_M_Modules_generation();
puts("DFT Modules generated");
Beam_Pattern_Init();
Linear_Combination_DFT_Modules();
puts("Linear Combination of DFT Modules generated");

index = fopen("Beam_pattern_real.dat", "w");

for (k = 0; k < N; k++) {
    for (m = 0; m < B; m++) {

fprintf(index, "%.5f\n", Beam_pattern[k][m].real);

    }
}
fclose(index);

index = fopen("Beam_pattern_image.dat", "w");

for (k = 0; k < N; k++) {
    for (m = 0; m < B; m++) {

fprintf(index, "%.5f\n", Beam_pattern[k][m].imag);

    }
}
fclose(index);
puts("done");}
```

}

Appendix E

Gumstix Verdex DFT Beamforming Code

```
//#include "dsk6713_aic23.h"
//Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate

#include <stdio.h>
#include <limits.h>
#include <math.h>
#include <stdlib.h>
#include <ctype.h>

#include <time.h>
#include <sys/time.h>

/*Number of points or steering directions received at each sensor*/
#define B 256
#define N 32 // Number of sensors
#define D 2 // Distance between sensors
#define L 2*D //lambda
#define d_L 1/L_d
#define PI 3.14159265358979
#define DELTA (2*PI)

/***** Constants defined for the FFT *****/
```

```

// #define PTS 64    // # of points for FFT
#define D 5
#define Lambda (2*D)

typedef struct {float real,imag;} COMPLEX;
void FFT(COMPLEX *Y);
//void FFT(COMPLEX *Y, int n);      //FFT prototype
COMPLEX input_vectors[N][B];
COMPLEX Beam_pattern[N][B];
COMPLEX w[N];
COMPLEX x[N];

short iTwid[N/2];

float beta,a,b;

//FILE *index;

/*  Function that returns "a - b" in seconds */

double timeval_diff(struct timeval *a, struct timeval *b)

{
    return
        (double)(a->tv_sec + (double)a->tv_usec/1000000) -
        (double)(b->tv_sec + (double)b->tv_usec/1000000);
}

void input_vector_generation()
{
    FILE *index;
    short m,k;

```

```

/*for (k = 0;k<N;k++)
{
    for(m = 0;m<B;m++)
    {
        input_vectors[k][m].real=input_matrix[k][m];
        input_vectors[k][m].imag=input_matrix_imag[k][m];
    }
}
*/

printf("Opening Sample input real without noise file\n");

/*****      fopen      *****/

index=fopen("sample_input_real_without_noise.h","r");

if((index)==NULL)
{
    puts("File could not be open");
    exit(-1);
}

for(m = 0; m < N; m++)
{
    for(k = 0; k < B; k++)
    {
        fscanf(index, "%f",&input_vectors[m][k].real);
    }
}

fclose(index);

printf("Closing Sample input real without noise file\n");

index=fopen("sample_input_imag_without_noise.h","r");

printf("Opening Sample input imag without noise file\n");

```

```

if((index)==NULL)
{
    puts("File could not be open");
    exit(-1);
}

for(m = 0; m < N; m++)
{
    for(k = 0; k < B; k++)
    {
        fscanf(index, "%f",&input_vectors[m][k].imag);
    }
}

fclose(index);

printf("Closing Sample input imag without noise file\n");
//exit(-1);
}

void Beam_pattern_generation()
{
    int col_input_matrix,i,j;
    COMPLEX x_col[N];
    for(col_input_matrix = 0; col_input_matrix < B; col_input_matrix++)
    {
        for (i=0;i<N;i++)
        {
            x_col[i] = input_vectors[i][col_input_matrix];
        }

        FFT(x_col);

        for (j=0;j<N;j++)
        {
            Beam_pattern[j][col_input_matrix].real = x_col[j].real;
            Beam_pattern[j][col_input_matrix].imag = x_col[j].imag;
        }
    }
}

```

```

        }
    }
}

void main()
{
    struct timeval t_ini, t_fin;
    double secs;
    int m,k,doblepts;
    FILE *index;
    //set up array of twiddle factors

    input_vector_generation();
    doblepts=2*N;

    for(k = 0;k<N;k=k+1)
    {
        w[k].real = cos((DELTA*k)/(doblepts));
        //printf("%.5f",Beam_matrix[m-1][k-1].real );
        w[k].imag =-sin((DELTA*k)/(doblepts));
        //printf("%.5f\n",Beam_matrix[m-1][k-1].imag );
    }

    gettimeofday(&t_ini, NULL);

    Beam_pattern_generation();

    gettimeofday(&t_fin, NULL);

    secs = timeval_diff(&t_fin,&t_ini);
    printf("%.16g milliseconds\n", secs *1000.0);

    index = fopen("Beam_pattern_real.dat","w");

    printf("Writing Beam pattern real\n");

    for (k = 0;k<N;k++)

```

```

    {
        for(m = 0;m<B;m++)
        {
            fprintf(index, "%.2f\n", Beam_pattern[k][m].real);
        }
    }

fclose(index);

printf("Closing Beam pattern real\n");

index = fopen("Beam_pattern_image.dat", "w");

printf("Writing Beam pattern imag\n");

for (k = 0;k<N;k++)
{
    for(m = 0;m<B;m++)
    {
        fprintf(index, "%.2f\n", Beam_pattern[k][m].imag);
    }
}
fclose(index);

printf("Closing Beam pattern imag\n");

puts("done");

}

\***** FFT *****/

//FFT.c C callable FFT function in C

//#define PTS 64 // # of points for FFT
//#define D 5
//#define Lambda (2*D)
//typedef struct {float real,imag;} COMPLEX;

```

```

//extern COMPLEX w[PTS];          //twiddle constants stored in w

void FFT(COMPLEX *Y)              //input sample array, # of points
{
    COMPLEX temp1,temp2;           //temporary storage variables
    int i,j,k;                     //loop counter variables
    int upper_leg, lower_leg;      //index of upper/lower butterfly leg
    int leg_diff;                  //difference between upper/lower leg
    int num_stages = 0;            //number of FFT stages (iterations)
    int index, step;

    int doblepts = 2*N;

                                //index/step through twiddle constant
    i = 1;                        //log(base2) of N points= # of stages
    do
    {
        num_stages +=1;
        i = i*2;
    }while (i!=N);
    leg_diff = N/2;               //difference between upper&lower legs
    step =doblepts/N;             //step between values in twiddle.h

    for (i = 0;i < num_stages; i++) //for N-point FFT
    {
        index = 0;
        for (j = 0; j < leg_diff; j++)
        {
            for (upper_leg = j; upper_leg < N; upper_leg += (2*leg_diff))
            {
                lower_leg = upper_leg+leg_diff;
                temp1.real = (Y[upper_leg]).real + (Y[lower_leg]).real;
                temp1.imag = (Y[upper_leg]).imag + (Y[lower_leg]).imag;
                temp2.real = (Y[upper_leg]).real - (Y[lower_leg]).real;
                temp2.imag = (Y[upper_leg]).imag - (Y[lower_leg]).imag;
                (Y[lower_leg]).real = temp2.real*(w[index]).real
                    -temp2.imag*(w[index]).imag;
                (Y[lower_leg]).imag = temp2.real*(w[index]).imag
                    +temp2.imag*(w[index]).real;
            }
        }
    }
}

```



```

        (Y[upper_leg]).real = temp1.real;
        (Y[upper_leg]).imag = temp1.imag;
    }
    index += step;
}
leg_diff = leg_diff/2;
step *= 2;
}
j = 0;
for (i = 1; i < (N-1); i++) //bit reversal for resequencing data
{
    k = N/2;
    while (k <= j)
    {
        j = j - k;
        k = k/2;
    }
    j = j + k;
if (i<j)
{
    temp1.real = (Y[j]).real;
    temp1.imag = (Y[j]).imag;
    (Y[j]).real = (Y[i]).real;
    (Y[j]).imag = (Y[i]).imag;
    (Y[i]).real = temp1.real;
    (Y[i]).imag = temp1.imag;
}
}
return;
}

```

Appendix F

Gumstix Verdex DFT Kronecker Beamforming Code

```
//#include "dsk6713_aic23.h"

//Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate

#include <stdio.h>
#include <limits.h>
#include <math.h>
#include <stdlib.h>
#include <ctype.h>
#include <time.h>
#include <sys/time.h>

#define L 8192 //Number of sensors
#define M 1024 //Number of FFT modules to compute
#define N 8// Length of each FFT module
#define B 256// Total number of input vectors defined for  $-1<\sin(\theta)<1$ 
#define PI 3.14159265358979
#define DELTA (2*PI)

#define PTS 8    // # of points for FFT
#define D 5
#define Lambda (2*D)
```

```

typedef struct {float real,imag;}COMPLEX ;
void FFT(COMPLEX *Y, COMPLEX *w);
//void FFT(COMPLEX *, int , COMPLEX);
//void FFT(COMPLEX* , int );
//void FFT(COMPLEX *Y, int n);      //FFT prototype
COMPLEX input_vectors[L][B];
COMPLEX DFT_modules_matrix[L][B];
COMPLEX Beam_pattern[N][B];
COMPLEX x_L[L];
COMPLEX x_module_N[N];
COMPLEX w[N];

//FILE *index;

/* retorna "a - b" en segundos */

double timeval_diff(struct timeval *a, struct timeval *b)

{
    return
        (double)(a->tv_sec + (double)a->tv_usec/1000000) -
        (double)(b->tv_sec + (double)b->tv_usec/1000000);
}

void input_vector_generation()
{
    FILE *index;
    int m,k;

    /*for (k = 0;k<N;k++)
        {
            for(m = 0;m<B;m++)
                {
                    input_vectors[k][m].real=input_matrix[k][m];
                    input_vectors[k][m].imag=input_matrix_imag[k][m];
                }
            }
        */

```

```

        }
    }
*/

printf("Opening sample input real without noise file\n");

/*****fopen*****/

index=fopen("sample_input_real_without_noise.h","r");

if((index)==NULL)
{
    puts("File could not be open");
    exit(-1);
}

for(m = 0; m < L; m++)
{
    for(k = 0; k < B; k++)
    {
        fscanf(index, "%f",&input_vectors[m][k].real);
    }
}

fclose(index);
printf("Closing sample input real without noise file\n");
/*****fopen*****/

index=fopen("sample_input_imag_without_noise.h","r");
printf("Opening sample input imag without noise file\n");

if((index)==NULL)
{
    puts("File could not be open");
    exit(-1);
}

for(m = 0; m < L; m++)

```

```

    {
        for(k = 0; k < B; k++)
        {
            fscanf(index, "%f",&input_vectors[m][k].imag);
        }
    }

fclose(index);
printf("Closing sample input imag without noise file\n");
//exit(-1);

}

void Twiddle_factors_generation()
{
    int k, doblepts;
    doblepts=2*N;

    for(k = 0;k<N;k=k+1)
    {
        w[k].real = cos((DELTA*k)/(doblepts));
        //printf("%.5f",Beam_matrix[m-1][k-1].real );
        w[k].imag =-sin((DELTA*k)/(doblepts));
        //printf("%.5f\n",Beam_matrix[m-1][k-1].imag );
    }
}

void Beam_Pattern_Init()
{
    int i,j;
    for(i=0;i<N;i++)
    {
        for(j=0;j<B;j++)
        {
            Beam_pattern[i][j].real=0;
            Beam_pattern[i][j].imag=0;
        }
    }
}

```

```

    }
}

void DFT_M_Modules_generation()    //Comienza operacion de Beamforming

{
    short i,j,k,p,b,c,n_index;
    for(i=0;i<B;i++)
    {
        b=0;    // Marks the current position in each input vector
                // Processing each input vector taken as a column of the matrix

        for(j=0;j<L;j++)
        {
            x_L[j] = input_vectors[j][i];
        }
        // Divide each input vector into M modules

        for(k=0;k<M;k++)
        {
            for(p=0;p<N;p++)
            {
                x_module_N[p] =x_L[b];
                b++;
            }
            FFT(x_module_N, w);
            n_index=b-N; // To place each DFT module currently in the output matrix
            for(c=0;c<N;c++)
            {
                DFT_modules_matrix[n_index][i]= x_module_N[c];
                n_index++;
            }
        }
    }
}

void Linear_Combination_DFT_Modules()
```

```

{
    short i,j,k,m;
    //Linear combination of the M DFT Modules
    for(i=0;i<N;i++)
    {
        for(j=0;j<B;j++)
        {
            for(k=0;k<=L-N;k=k+N)
            {
                m =i+k;
                Beam_pattern[i][j].real = Beam_pattern[i][j].real+ DFT_modules_matrix[m][j];
                Beam_pattern[i][j].imag = Beam_pattern[i][j].imag+ DFT_modules_matrix[m][j];
            }
        }
    }
}

```

```

/*****main*****/

```

```

void main()
{
    struct timeval t_ini, t_fin;
    double secs;
    int m,k;
    FILE *index;

    //set up array of twiddle factors

    input_vector_generation();
    puts("Input vector Matrix generated\n");
    Twiddle_factors_generation();
    Beam_Pattern_Init(); /*Inicializa en Beamforming*/
    puts("Twiddle Factors generated\n");

    gettimeofday(&t_ini, NULL);

```

```

DFT_M_Modules_generation();
puts("DFT Modules generated\n");

Linear_Combination_DFT_Modules();

gettimeofday(&t_fin, NULL);
puts("Linear Combination of DFT Modules generated\n");

secs = timeval_diff(&t_fin, &t_ini);
printf("%.16g milliseconds\n", secs * 1000.0);

index = fopen("Beam_pattern_real.dat", "w");
printf("Opening Beam pattern real\n");
for (k = 0; k<N; k++)
{
    for(m = 0; m<B; m++)
    {
        fprintf(index, "%.5f\n", Beam_pattern[k][m].real);
    }
}
fclose(index);
printf("Closing Beam pattern real\n");

printf("Opening Beam pattern image\n");

index = fopen("Beam_pattern_image.dat", "w");
for (k = 0; k<N; k++)
{
    for(m = 0; m<B; m++)
    {
        fprintf(index, "%.5f\n", Beam_pattern[k][m].imag);
    }
}

fclose(index);
printf("Closing Beam pattern real\n");

```



```

printf("Opening Beam pattern image\n");

index = fopen("Beam_pattern_image.dat","w");
for (k = 0;k<N;k++)
{
    for(m = 0;m<B;m++)
    {
        fprintf(index,"%0.5f\n",Beam_pattern[k][m].imag);
    }
}
fclose(index);
printf("Closing Beam pattern image\n");
puts("done");
}

/*****FFT*****/

//FFT.c  C callable FFT function in C

//#define PTS 8
//#define D 5
//#define Lambda (2*D)
//typedef struct {float real,imag;} COMPLEX;
//extern COMPLEX w[PTS];

void FFT(COMPLEX *Y, COMPLEX *w) //input sample array, # number of points

{
    COMPLEX temp1;
    COMPLEX temp2;           //temporary storage variables
    int i,j,k;               //loop counter variables
    int upper_leg, lower_leg; //index of upper/lower butterfly leg
    int leg_diff;             //difference between upper/lower leg
    int num_stages = 0;       //number of FFT stages (iterations)

```

```

int index, step;
int doblepts = 2*N;
    //index/step through twiddle constant
i = 1;    //log(base2) of N points= # of stages
do
{
    num_stages +=1;
    i = i*2;
}while (i!=N);
leg_diff = N/2; //difference between upper&lower legs

step = doblepts/N;          //step between values in twiddle.h
for (i = 0; i < num_stages; i++) //for N-point FFT
{
    index = 0;
    for (j = 0; j < leg_diff; j++)
    {
        for (upper_leg = j; upper_leg < N; upper_leg += (2*leg_diff))
        {
            lower_leg = upper_leg+leg_diff;
            temp1.real = (Y[upper_leg]).real + (Y[lower_leg]).real;
            temp1.imag = (Y[upper_leg]).imag + (Y[lower_leg]).imag;
            temp2.real = (Y[upper_leg]).real - (Y[lower_leg]).real;
            temp2.imag = (Y[upper_leg]).imag - (Y[lower_leg]).imag;
            (Y[lower_leg]).real = temp2.real*(w[index]).real-temp2.imag*(w[index]).imag;
            (Y[lower_leg]).imag = temp2.real*(w[index]).imag+temp2.imag*(w[index]).real;
            (Y[upper_leg]).real = temp1.real;
            (Y[upper_leg]).imag = temp1.imag;
        }
        index += step;
    }

    leg_diff = leg_diff/2;
    step *= 2;
}

j = 0;
for (i = 1; i < (N-1); i++) //bit reversal for resequencing data

```

```
{
    k = N/2;
    while (k <= j)
    {
        j = j - k;
        k = k/2;
    }
    j = j + k;
    if (i < j)
    {
        temp1.real = (Y[j]).real;
        temp1.imag = (Y[j]).imag;
        (Y[j]).real = (Y[i]).real;
        (Y[j]).imag = (Y[i]).imag;
        (Y[i]).real = temp1.real;
        (Y[i]).imag = temp1.imag;
    }
}
return;
}
```

Appendix G

pMATLAB DFT Beamforming Code

```
%Multi- Linear beamforming implementation in MATLAB, using DFT

% The number of steering angles considered is the number of Sensors in the
% system

N = 2^6; % NxB Matrix size.

% Turn parallelism on or off.
PARALLEL = 0; % Can be 1 or 0. OK to change.

% Initialize pMatlab.
% Initialize pMatlab.
pMatlab_Init;
Ncpus = pMATLAB.comm_size;
my_rank = pMATLAB.my_rank;

% Create Maps.
mapX = 1; mapY = 1;
if (PARALLEL)
    % Break up channels.
    mapX = map([1 Ncpus], {}, 0:Ncpus-1 );
    mapY = map([1 Ncpus], {}, 0:Ncpus-1 );
    %mapZ = map([Ncpus 1], {}, 0:Ncpus-1 );
```

```

end
close all;
%B=-1:10^(-2):1;
NumberOfPoints = 256;
B=-1:2/(NumberOfPoints-1):1;

sample_input_matrix_1 = zeros(N,length(B), mapX);
Beam_pattern_1 = zeros(N,length(B),mapY);

d= 2;
lambda =2*d;
phi_0= 1;%Initial Amplitude of signal
i=1;
    % -1<(B=SIN(THETA))<1 201 points
for m=1:1:length(B);

    Bk(1,i) = B(m);%Steering Direction
    for k=1:1:N
        % Original Signal
        sample_input_matrix_1(k,m) = phi_0*(exp(j*2*pi*((k-1))*Bk(1,i)*d/lambda));

        end
        i = i+1;
    end

tic;
Beam_pattern_1 = fft(sample_input_matrix_1);
elapsedTime = toc;

disp(['Elapsed time = ',num2str(elapsedTime,'%0.4f'),' sec.']);

%Plotting the beam pattern formed for some of the input vectors, where
%each column of the matrix Beam_pattern is a beam pattern of the
%corresponding input vector

```

```
y = local(Beam_pattern_1);

figure;
channel=6;
plot(B, abs(y(channel,:)));
title('pMatlab DFT Beam Pattern Formations');
xlabel('Steering Direction');
ylabel('Magnitude');
grid

disp('SUCCESS');
pMatlab_Finalize;

}
```

Appendix H

pMATLAB DFT Kronecker Beamforming Code

```
%Multi- Linear beamforming implementation in MATLAB, based on the concepts  
%using kroneckers
```

```
% The number of steering angles considered is the number of Sensors in the  
% system
```

```
% Turn parallelism on or off.  
PARALLEL =0; % Can be 1 or 0. OK to change.
```

```
% Initialize pMatlab.  
pMatlab_Init;  
Ncpus = pMATLAB.comm_size;  
my_rank = pMATLAB.my_rank;
```

```
% Create Maps.  
mapX = 1; mapY = 1; mapf=1;  
if (PARALLEL)  
    % Break up channels.  
    mapX = map([ 1 Ncpus], {}, 0:Ncpus-1 );  
    mapY = map([ 1 Ncpus], {}, 0:Ncpus-1 );
```

```

end
close all;
L = 64 % Number of sensors
M= 8%Number of Modules
N =8 %Number of sensors in linear array
d= 1;% distance between sensors
lambda =2*d;% wavelength
phi_0= 1;%Initial Amplitude of signal

NumberOfPoints = 256;
B=-1:2/(NumberOfPoints-1):1;

sample_input_matrix_1 = zeros(L,length(B), mapX);
Beam_pattern_1 = zeros(L,length(B),mapY);

U_M = ones(M,1);
I_N = eye(N,N);
I_M = eye(M,M);

i=1;
for m=1:1:length(B);
Bk(1,i) = B(m);%Steering Direction
    for k=1:1:L
        %Signals from the same Monochromatic plane
        sample_input_matrix_1(k,m) = phi_0*(exp(j*2*pi*((k-1))*Bk(1,i)*d/lambda));
    end
    i = i+1;
end

tic;
Beam_pattern_1= kron(U_M', I_N )*kron(I_M,dftmtx(N) )*sample_input_matrix_1; %Beam pattern
elapsedTime = toc;

```



```
disp(['Elapsed time = ',num2str(elapsedTime,'%0.4f'),' sec.']);

%Plotting the beam pattern formed for some of the input vectors, where
%each column of the matrix Beam_pattern is a beam pattern of the
%corresponding input vector

figure
channel=1:1:N;

plot(B,abs(Beam_pattern_1(channel,:)))
title('Kronecker DFT Beam Pattern Formations in pMATLAB');
xlabel('Steering Direction');
ylabel('Magnitude');
grid

pMatlab_Finalize;

realOut=real(Beam_pattern_1);
imagOut=imag(Beam_pattern_1);

save Beam_pattern_real.dat realOut -ascii
save Beam_pattern_image.dat imagOut -ascii
}
```

Appendix I

Signal Analysis and Metrics in MATLAB

```
%This program conducts a signal analysis between the beamforming data obtained in MATLAB
%The third row will be analyzed from each data
clear all;
close all;

L=64;
M=8;
N=8;
B = 256; % Number of points or plane waves received at each sensor
%Obtaining original signal Sio from MATLAB
xreal = load('Beam_pattern_realMATLAB.dat');
ximag = load('Beam_pattern_imageMATLAB.dat');
Sio = xreal+ j*ximag;
Sio = Sio(3,:);

%Obtaining original signal Sid from DSP
xreal = load('Beam_pattern_realDSP.dat');
ximag = load('Beam_pattern_imageDSP.dat');
Sid = xreal+ j*ximag;
Sid = Sid(3,:);
```

```
%Obtaining original signal Sig from Gumstix
xreal = load('Beam_pattern_realGumstix.dat');
ximag = load('Beam_pattern_imageGumstix.dat');
Sig = xreal+ j*ximag;
Sig = Sig(3,:);

%Obtaining original signal Sip from pMATLAB
xreal = load('Beam_pattern_real_pMATLAB.dat');
ximag = load('Beam_pattern_image_pMATLAB.dat');
Sip = xreal+ j*ximag;
Sip = Sip(3,:);

%Calculating difference between the original, ideal signal obtained in
%MATLAB, with the other platforms

xiz = Sio - Sio;
xd = Sio - Sid;
xg = Sio - Sig;
xp = Sio - Sip;

%Calculating the mean of each signal Sio, Sid, Sig, Sip, and xiz, xd, xg, xp

u_Sio = mean(Sio);
u_Sid = mean(Sid);
u_Sig = mean(Sig);
u_Sip = mean(Sip);

u_xiz = mean(xiz);
u_xd = mean(xd);
u_xg = mean(xg);
u_xp = mean(xp);

%Calculating the variance of each signal Sio, Sid, Sig, Sip, and xiz, xd, xg,
%xp

var_Sio = var(Sio);
var_Sid = var(Sid);
```

```
var_Sig = var(Sig);
var_Sip = var(Sip);

var_xiz = var(xiz);
var_xd = var(xd);
var_xg = var(xg);
var_xp = var(xp);

%Calculating the standard variance of each signal Sio, Sid, Sig, Sip, and xiz, xd,
%xg, xp

std_Sio = std(Sio);
std_Sid = std(Sid);
std_Sig = std(Sig);
std_Sip = std(Sip);

std_xiz = std(xiz);
std_xd = std(xd);
std_xg = std(xg);
std_xp = std(xp);

%Calculating the Energy of Sio, Sid, Sig, Sip, and xiz, xd,
%xg, xp

for b = 1:1:B

    En_Sio(1,b) = abs(Sio(1,b))^2;
    En_Sid(1,b) = abs(Sid(1,b))^2;
    En_Sig(1,b) = abs(Sig(1,b))^2;
    En_Sip(1,b) = abs(Sip(1,b))^2;

    En_xiz(1,b) = abs(xiz(1,b))^2;
    En_xd(1,b) = abs(xd(1,b))^2;
    En_xg(1,b) = abs(xg(1,b))^2;
    En_xp(1,b) = abs(xp(1,b))^2;
end

En_Sio = sum(En_Sio);
```

```
En_Sid = sum(En_Sid);
En_Sig = sum(En_Sig);
En_Sip = sum(En_Sip);

En_xiz = sum(En_xiz);
En_xd = sum(En_xd);
En_xg = sum(En_xg);
En_xp = sum(En_xp);

%Calculating the Power of Sio, Sid, Sig, Sip, and xiz, xd,
%xg, xp

P_Sio = (1/B)*En_Sio;
P_Sid = (1/B)*En_Sid;
P_Sig = (1/B)*En_Sig;
P_Sip = (1/B)*En_Sip;

P_xiz = (1/B)*En_xiz;
P_xd = (1/B)*En_xd;
P_xg = (1/B)*En_xg;
P_xp = (1/B)*En_xp;
```

Bibliography

- [1] F. Eigenbroda, S. J. Hecnarb, and L. Fahriga. The relative effects of road traffic and forest cover on anuran populations. In *Proceedings of Biological Conservation 141 year = 2008*, pages = 35–46,.
- [2] J. C. Chen, L. Yip, J. Elson, H. Wang, D. Maniezzo, R.E Hudson, K. Yao, and D. Estrin. Coherent acoustic array processing and localization on wireless sensor networks. In *Proceedings of IEEE*, number 8, pages 1154–1162, 2003.
- [3] A. Quinchanequa and D. Rodríguez. A kronecker dft multi-beamforming implementation approach. In *IEEE 3rd International Symposium on Image and Signal Processing and Analysis*, pages 1066–1071, 2003.
- [4] H. Wang, C. E. Chen, A. Ali, S. Asgari, R.E. Hudson, K. Yao, D. Estrin, and C. Taylor. Acoustic sensor networks for woodpecker localization. In *Advanced Signal Processing Algorithms, Architectures, and Implementation XV, Proceedings of SPIE*, volume 5910, pages 1–1, 2005.
- [5] Y. Huang, J. Benesty, and J. Chen. Time delay estimation and source localization. *Springer Handbook of Speech Processing*, pages 1043–1063, 2008.
- [6] H. Wang. Wireless sensor networks for acoustic monitoring. pages 1–132, 2006.
- [7] J. C. Chen, K. Yao, and R. E. Hudson. Source localization and beamforming. *Collaborative Signal and Information Processing in Microsensor Networks*, pages 30–38, 2002.
- [8] L. Yip, K. Comanor, J.C Chen, R.E. Hudson, K. Yao, and L. Vandenberghe. Array processing for target doa, localization, and classification based on aml and svm algorithms in sensor networks. pages 269–284, 2003.
- [9] E. Warsitz and R. Haeb-Umbach. Blind acoustic beamforming based on generalized eigenvalue decomposition. In *Proceedings of IEEE*, pages 1529–1539, 2007.
- [10] D. B. Ward, R. A. Kennedy, and R. C. Williamson. Fir filter design for frequency invariant beamformers. In *Proceedings of IEEE*, pages 69–71, 1996.

- [11] D. B. Ward, R. A. Kennedy, and R. C. Williamson. An adaptive algorithm for broadband frequency invariant beamforming. In *Proceedings of IEEE*, pages 3737–3740, 1997.
- [12] D. B. Ward, R. A. Kennedy, and Z. Ding. Broadband doa estimation using frequency-invariant beam-space processing. In *Proceedings of IEEE*, pages 2892–2895, 1996.
- [13] J Granata and R. Tolimieri. The tensor product: A mathematical programming language for ffts and other fast dsp operations. In *Proceedings of IEEE SP Magazine*, volume 9, pages 40–48, 1992.
- [14] Rodriguez D. Tensor product algebra as a tool for vlsi implementation of the discrete fourier transform. In *Proceedings of IEEE*, pages 1025–1028, 1992.
- [15] J.R. Guerci. *Space-Time Adaptive Processing for Radar*. Artech House Inc., MA, 2003.
- [16] J. Kepner. *Parallel Programming with pMatlab: Parts I and II*. MIT Lincoln Laboratory, Lexington, MA, 2008.
- [17] D. J. Kuck. *High Performance Computing: Challenges for Future System*. Oxford University Press, NY, 1996.
- [18] J. Kepner, A. Reuther, and H. Kim. *Parallel Programming in Matlab-Tutorial*. MIT Lincoln Laboratory, 2008.
- [19] H. Kim and N. Travinin. *pMatlab v0.7 Function Reference*. MIT Lincoln Laboratory, 2008.
- [20] R. Chassaing. *Digital Signal Processing and Applications with the C6713 and C6416 DSK*. John Wiley & Sons Inc, 2005.
- [21] Incorporated Texas Instruments. Tms320c6474 multicore digital signal processor. pages 1–215, October 2008, Revised February 2009.
- [22] S. Unnikrishna Pillai, B. Yeheskel, and F. Haber. A new approach to array geometry for improved spatial spectrum estimation. In *Proceedings of IEEE*, volume 73, pages 1522–1524, 1985.
- [23] Julius Orion Smith III Home Page. <http://ccrma.stanford.edu/~jos/>.