HIERARCHICAL APPROACH TO GLOBAL MODELING OF ACTIVE ANTENNA ARRAYS

by

USMAN AZEEZ MUGHAL

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

ELECTRICAL ENGINEERING

Raleigh

1999

APPROVED BY:

Abstract

MUGHAL, USMAN AZEEZ. Hierarchical Approach to Global Modeling of Active Antenna Arrays. (Under the direction of Michael B. Steer.)

This thesis presents an hierarchical modeling approach to electromagnetic modeling of electrically large planar structures. The computer aided design tool, *ArraySim*, is ideally suited for modeling active antenna arrays. The structure is sketched in a graphical layout tool which produces a layout file in a CIF format. The CIF file is parsed, circuit ports identified and basis cells assigned. A search algorithm is used to minimize repetitive calculations of elements thus increasing speed and efficiency. An independent MoM analysis tool is called to fill an evolving impedance/admittance matrix, element by element. *ArraySim* is a complete modeling scheme for spatial power combining arrays targeted at producing tens and hundreds of watts of power at microwave and millimeter wave frequencies. While *ArraySim* is targeted at spatial power combining arrays, it can be used to model any tri-layer structure if appropriate MoM routines are used. Using *ArraySim*, we study the behavior of double-slot stripline coupled (SSS) and folded-slot structures. Unit cell as well as arrays are simulated and results are compared with published results.

Dedication

This thesis is dedicated to my newly wedded wife Zahra whose persisted patience and encouragement helped me finish my Masters. I would also wish to dedicate this thesis to my parents, Azeez and Gulnaz, my brother Adnan and my sister Mona whose understanding and support gave me motivation for my studies.

Biographical Summary

Usman Azeez Mughal was born in Hawalli, Kuwait on December 27, 1973. He finished his high school from Government College Lahore, Pakistan with a 3rd position in the B.I.S.E. pre-engineering group. He received his B.S. degree in Computer Engineering (Summa Cum Laude) from North Carolina State University. While pursuing his B.S. degree he worked as a co-op for BNR, RTP NC in the Spring of 1996 and NORTEL, RTP, NC in the Summer of 1997. He was admitted into the Masters Program at North Carolina State University in the Fall of 1997. While working toward his M.S. degree he held a research assistantship with the Electronics Research Laboratory in the Dept. of Electrical and Computer Engineering. In the summer of 1998, he worked as an intern at Cadence Design System, Inc. Cary, NC. His research interests include VLSI design, RF circuits, digital electronics, computer aided simulation tool development and international finance. He is an elected member of Phi Kappa Phi and a member of the Institute of Electrical and Electronic Engineers and the Microwave Theory and Techniques Society.

Acknowledgments

I wish to express my gratitude to my advisor Dr. Michael Steer for his support and guidance during this lengthy work. It has been an honor working with him in his quasi-optical group and I extend good luck to him on his new career at the University of Leeds, U.K. Surely, this group and N.C. State University will miss his valuable engineering experience and humble presence. I would also like to thank Dr. Amir Mortazawi and Dr. James Mink for serving on my committee.

As a computer engineering background, I struggled learning about quasi-optical systems, but with the help of Mostafa Abdullah, I was able to understand this exciting new research field and learn more about EM and circuits. My thanks go to everyone in Room 410 and 412 EGRC. First to Satoshi Nakazawa for helping trace some of my code segmentation faults, Chris Hicks for giving me advises as an elder brother, Carlos Christoferson for his help in Fortran integration problem, Hector Gutierrez for his help in analog circuits concepts, Bari Biswas for his digital electronics help and all other colleagues; Mete Ozkar, Ahmed Khalil and Rizwan Bashirullah for their help in numerous ways.

Finally, I would like to express my thanks to my past and present instructors especially my high school professor, Dr. "Sir Arshad" who built my foundation in electrical engineering.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Quasi-optical power combiners using active antenna arrays are becoming an important and efficient way of combining power in free space. Optimum design requires modeling and understanding of coupling effects among unit cells in arrays.

Costly and tedious experiments to study the characteristics of these array antennas can be replaced by computer aided tools to simulate array structures. Changing dimensions of array unit cell and adjusting spacing between unit cells in an array can change the resonating frequency and coupling effects among array unit cells and thus increase system efficiency. These changes in behavior of a system can be viewed much quicker using computer aided tools rather than by conducting individual experiments.

A computer aided engineering tool (CAE) has been developed that is based on Method of Moments (MoM) analysis to simulate two antenna structures; Coplanar Waveguide (CPW) Slot Antenna and Slot Strip-line Slot (SSS) Antenna. The CAE tool developed here, called *ArraySim*, is based on the $C++$ programming language with MoM analysis implementation in Fortran. The tool is designed so that new antenna structures with their MoM functions can be added with no difficulty. This approach is similar to adding a circuit element into SPICE. *ArraySim* is developed to simulate and study the behavior of CPW and SSS arrays in free space.

1.2 Background

Computer aided design and modeling provides efficient ways of evaluating performance of a system. Several CAD tools are now available for simulating circuits at low and high frequencies. Quasi-optical systems involve several components such as; hard horns for plane wave excitation, optical lenses for beam focusing and active antenna arrays for power combining. The need for CAD tools to model and describe such systems is overwhelming. The basic approach here is to model each part of the quasi-optical system individually and then cascade the individual information to analyze the complete system. This approach is sometimes referred as an hierarchical approach to modeling structures.

Several approaches to simulating an antenna system include:

- Methods of Moments (MoM) and spectral domain analysis
- Finite Time Domain Analysis (FTDA)

- Transmission Line Method (TCM), time domain simulation using diakoptics
- Modal expansion and multi-port network development technique

In this thesis, a CAD tool for analyzing an antenna array in free space, a block of Quasi-Optical Systems, is described. This tool is based on Method of Moment analysis.

1.3 Quasi-Optical System Description

Circuit power combining schemes are associated with high losses in conductors. These high losses can be avoided by using antenna arrays for combining power. Such antenna arrays form part of a system referred to as a "Quasi-optical power combining system" as shown in Figure 1.1. The system involves integration of many solid state devices into a quasi-optical component like antenna array and thus achieve a high power source. The basic idea is to replace corporate combining with its associated conductor losses with lossless or low-loss combining in air dielectric. Thus combining in air results in efficient power combining. Numerous solid-state millimeter-wave sources are attached to radiating antenna arrays so that power from these radiating elements is combined in free space and focused down into a single mode. Mink [1] developed the first theoretical basis of power combining using source array in a quasi-optical resonator. Presently, active quasi-optics has become one of the

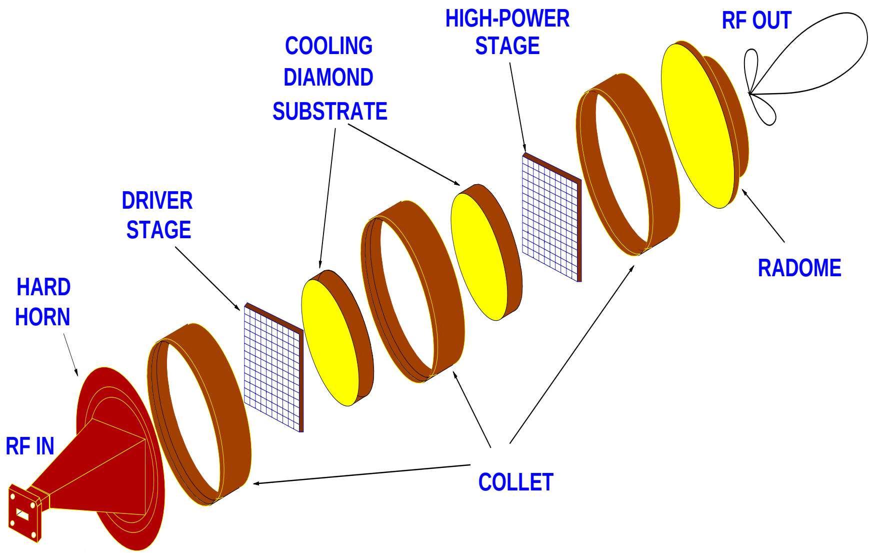


Figure 1.1: A Quasi-Optical Power Combining Model

most exciting research areas in the microwave power combining community. As a result of high complexity and challenge in understanding the behavior of quasi-optical systems, it has become imperative to create computer aided design tools to study the characteristics of quasi-optical components to achieve higher efficiency.

The most important achievements discussed in this thesis are:

- Developing a tool for full wave analysis of antenna arrays in free space. *ArraySim* allows to investigate the behavior of various structures up to three layers of depth.
- Creating a general algorithm to develop an efficient CAD tool to simulate antenna array structures using MoM.

- Reducing computation time by using smart search routines to find symmetries within the antenna array.
- Integrating the CAD tool with the non-linear circuit simulator Transim for designing amplifiers using network parameters.

1.4 Thesis Overview

Chapter 2 covers the implementation of MoM analysis and structure of *ArraySim* in detail. Chapter 3 describes simulation results of Slot stripline Slot (SSS) unit cells and arrays and compares them with measurements. Chapter 4 describes simulation results of folded-slot unit cells and arrays with connecting CPW lines and compares them with measurements. Chapter 5 summarizes the thesis with conclusions and suggestions for future work on this topic.

1.5 Publications

Following publications resulted from this work:

- Mostafa N. Abdullah, Usman A. Mughal, Huan-Shang Tsai, Michael B. Steer and Robert A. York, "A Full-Wave System Simulation of a Folded-Slot Spatial Power Combining Amplifier Array," *accepted for publication in IEEE MTT-S Int. Microwave Symp.*, June 1999.
- Mostafa N. Abdullah, Usman A. Mughal and Michael B. Steer, "Network Characterization for a Finite Array of Folded-Slot Antennas for Spatial Power Combining Application," *accepted for publication in IEEE AP-S*, July 1999.
- M. B. Steer, J. Harvey, J. W. Mink, M. N. Abdullah, C. E. Christoffersen, H. Gutierrez, C. W. Hicks, A. I. Khalil, U. A. Mughal, S. Nakazawa, T. W. Nuteson, J. Patwardhan, M. A. Summers, A. B. Yakovlev, "Global Modeling of Spatially Distributed Microwave and Millimeter-Wave Systems," *accepted for publication in IEEE MTT-S*, July 1999.

Chapter 2

Structure of MoM Based ArraySim

In this chapter we describe in detail the structure of *ArraySim*, a CAD tool based on MoM to simulate antenna arrays in quasi-optical systems. *ArraySim* allows users to draw array structures using a commercial layout editor like Cadence Virtuoso and then use the CIF description produced along with other inputs parameters (frequency sweep, dielectric properties, incident fields, antenna type, near and far field distances etc.) describing the structure to run a full-wave analysis of the given problem. *ArraySim* is essentially composed of two parts; MoM based passive analysis and active analysis followed by a non-linear circuit simulator interface. Before describing parts of *ArraySim*, it is important to understand the basics of Method of Moments as it forms the basis of *ArraySim*.

2.1 Background of Method of Moments

Method of Moments is a numerical technique to solve complex integral equations generated by electro-magnetic description of a certain structure. In electro-magnetics, we usually end up with integral equations that cannot be solved analytically and thus have to be solved using some kind of numerical technique. Method of Moments (MoM) offers accurate numerical solutions that transform integral equations into linear equations which are easier to solve using a computer program. MoM was first introduced by Harrington [2]. A general electro magnetic integral equation is;

$$f(u) = \int_0^1 G(u, u')I(u')du' \tag{2.1}$$

where $G(u, u')$ is a known kernel, Greens function, $f(u)$ is a known function and $I(u')$ is the unknown function to be determined. The unknown function, $I(u')$ can be expressed as a Fourier series sine and cosine functions, unit height-pulse functions or overlapping functions. Figure 2.1 and Figure 2.2 represent a sinusoidal and triangular basis function that are used to describe the unknown. Usually, the unknown, $I(u')$ is a current function that is approximated as a summation of a certain type of basis function. $I(u')$ is given as;

$$I(u') = \sum_{n=1}^{N} I_n \Phi_n(u') \tag{2.2}$$

where I_n are unknown amplitude constants to be determined.

If triangular basis functions are used as approximations to I_n, we get a linear interpolation of $I(u)$ with sample points represented by coefficients I_n. Triangular and sinusoidal basis

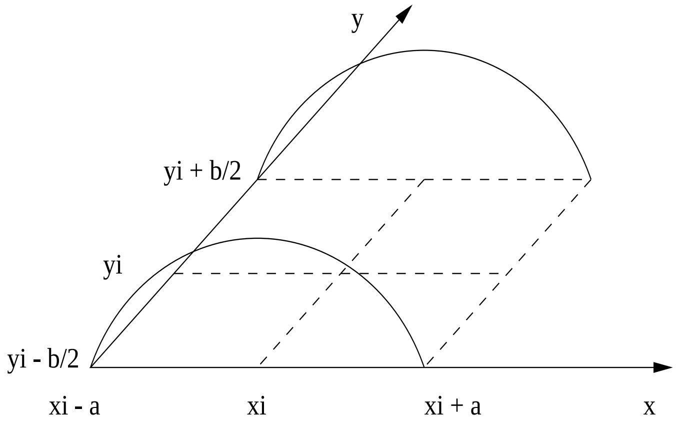


Figure 2.1: An x-directed sinusoidal basis function

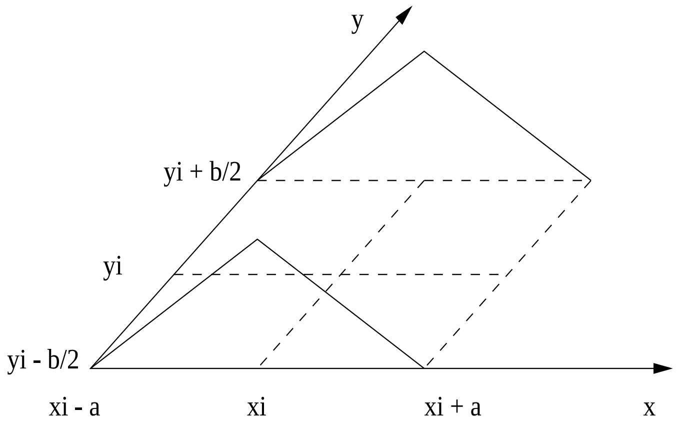


Figure 2.2: An x-directed triangle basis function

functions tend to give a smother approximation with better accuracy than rectangular basis functions [3]. Another reason to use triangular basis cells is their inherent property of numerical convergence.

When the expansion of Equation 2.2 is substituted into the integral Equation 2.1 we obtain

$$f(u) = \sum_{n=1}^{N} I_n \int_0^1 G(u,u')\Phi_n(u')du' = \sum_{n=1}^{N} I_n G_n(u) \tag{2.3}$$

where

$$G_n(u) = \int_0^1 G(u,u')\Phi_n(u')du' \tag{2.4}$$

The above equation can be considered as a moment of $G(u,u')$ with respect to ϕ_n. So, the objective is to choose coefficients I_n such that Equation 2.3 is satisfied as closely as possible.

Equation 2.4 forms the basis of point matching system. In this system, the test cell is a delta function while the source is the basis cell.

There are N unknown constants I_n, so a system of N equations must be obtained that will allow these unknown constants to be determined. One procedure is to equate both sides of Equation 2.3 at N different values of u, which are equally spaced at increments of

$$h = \frac{1}{N-1} \tag{2.5}$$

This point matching system gives a system of equations as

$$f(u) = \sum_{n=1}^{N} G_{mn} I_n = \sum_{n=1}^{N} I_n G_n(mh) = f(mh) = f_m \tag{2.6}$$

where

$$G_{mn} = G_n(mh) \tag{2.7}$$

So, in matrix form:

$$[G_{mn}]\,[I_n] = [f_m] \tag{2.8}$$

Galerkins method, on the other hand, is a more complex method because the test and source cells are actual basis cells and so reactions computed (G_{mn}) are over the area of the test and source basis cells and not the source basis cell center. The trade off between Galerkins method over the point matching method is speed versus accuracy. Galerkins method ends up with large integral equations while point matching method has simple integral equations. The spatial-domain dyadic Greens function are calculated using a numerical evaluation of Sommerfeld integration [4]. However, Galerkins method is more accurate than point matching method. In *ArraySim*, the MoM technique used is based on Galerkins method.

In the above Equation 2.8, I_n is solved by taking the inverse of G_{mn}. One of the major problems of MoM is deriving the Greens function of a given structure. Once the **G** matrix is computed, the remaining problem becomes a matter of matrix solving.

In the next section we discuss the structure of *ArraySim* that is based on MoM. Smart algorithms are used to fill the **G** matrix in order to reduce the integral evaluation of complex Greens functions.

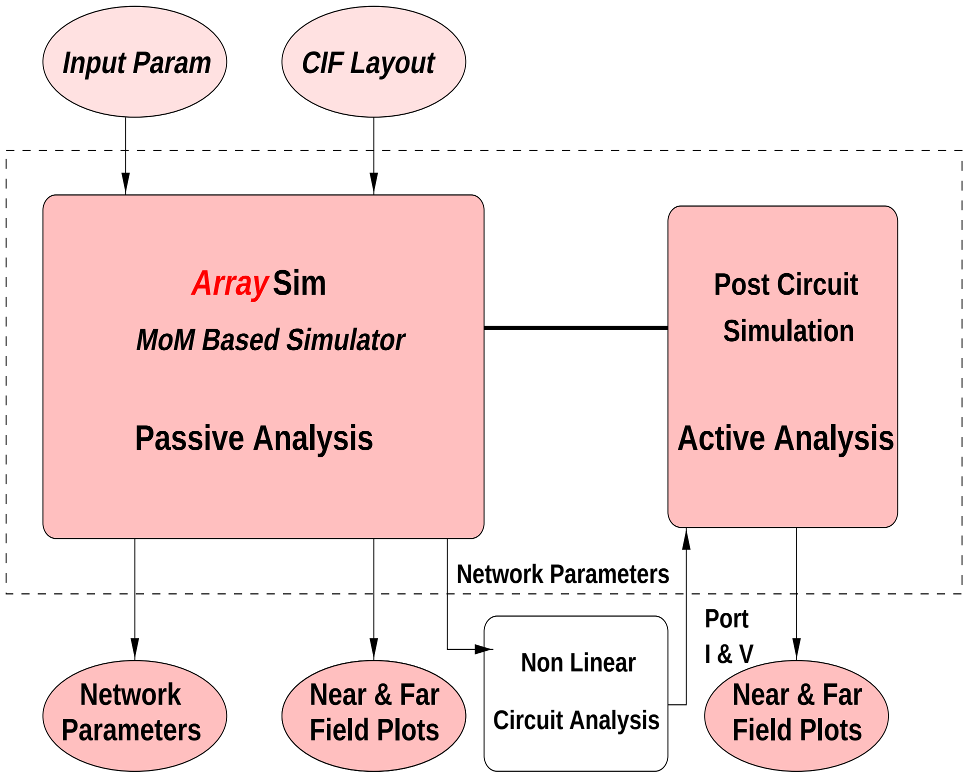


Figure 2.3: Top Level ArraySim Simulation Environment for Passive and Active Analysis of Array Structures

2.2 ArraySim

ArraySim is CAD tool based on MoM analysis. It is a complete simulation environment for full-wave analysis of antenna array structures. As discussed in the previous section, the values of the **G** elements in Equation 2.7 are computed for a given structure. Depending on boundary conditions, each structure has its own specific Greens function. These integral equations of Greens function are implemented in Fortran due to Fortran's fast math functions. In crude form we can say that a Greens function is the behavior of an antenna structure just like a circuit element model in a SPICE like circuit simulator. So, *ArraySim* treats a Greens function of a certain structure as an element model. Presently, *ArraySim* can handle Slot-Stripline-Slot structure and a folded-slot structure. As Greens functions of other structures are derived, their MoM interface can be included in *ArraySim* with few modifications.

ArraySim allows users to study the behavior of a given structure, laid out using a commercial layout editor. Figure 2.3 shows a top level flow of *ArraySim.* After drawing the structure, a CIF file is extracted describing the layout. This CIF file is used as input to *ArraySim* which parses the layout and extracts the basis cells. Other input parameters describing the structure material are listed in an input parameter file. *ArraySim* will then use these two input files and perform passive analysis of the array. In the next set of subsections, each component of *ArraySim* is described in detail. Each component is an object defined in $C++$ with a certain set of instance variables and member functions describing the behavior of the object [6].

The simulation environment of *ArraySim* is divided into two categories:

- Passive analysis

- Active analysis

2.3 Passive Analysis

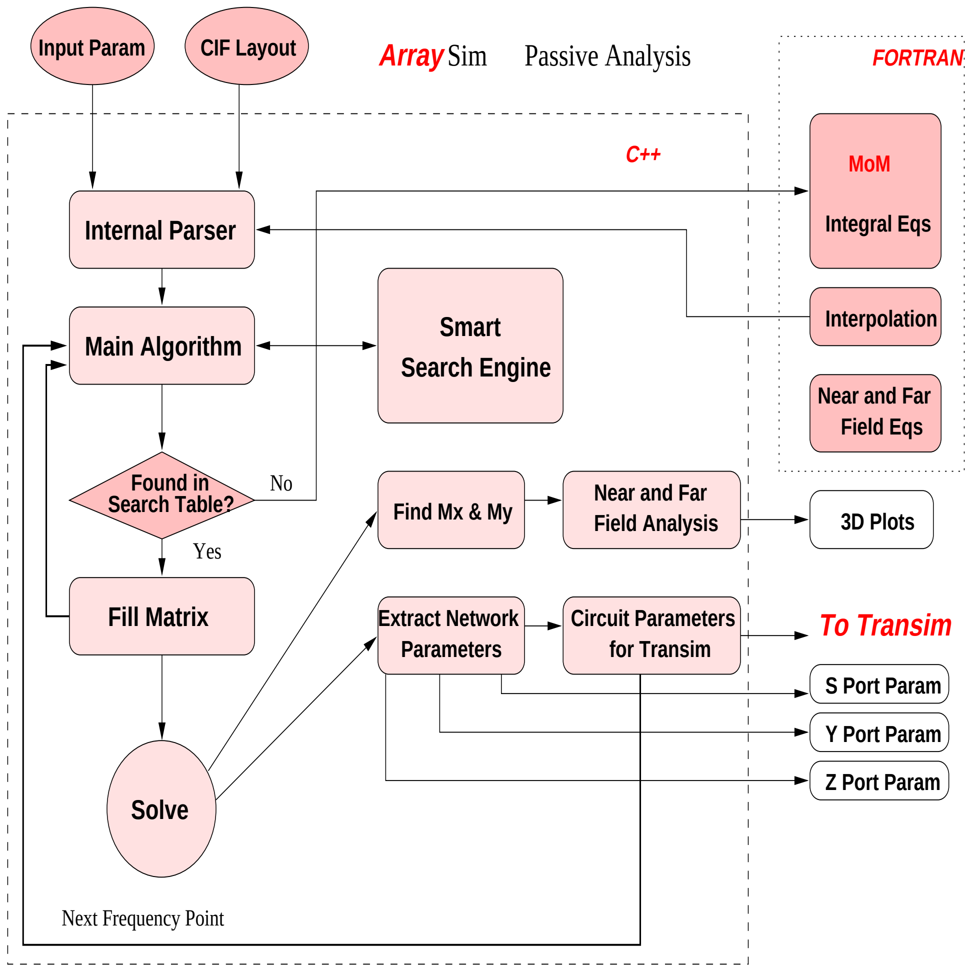


Figure 2.4: Flow for Passive Analysis

Passive Analysis makes up the first part of *ArraySim*. This process consists of parsing a CIF file [11] of an antenna array layout and reading the input parameters describing the structure. The input parameter file contains information about the type of incident fields on the array and other parameters like frequency sweep and print options. Figure 2.4 shows the flow of passive analysis procedure in *ArraySim*.

After parsing the input files, many data structures are allocated memory. These data structures keep track of basis cells, ports, layer information etc. The main algorithm is then invoked that begins to calculate the G_{mn} values. The $\mathbf{G}$ matrix is a complex matrix that includes many sub-matrices. Each sub-matrix further contains a set of four matrices. These matrices are filled in an intelligent way. After filling, excitation vectors ($f(m)$) are set up and the matrix is numerically solved. The solution of the matrix is used to extract network parameters at the ports. Finally port currents or voltages along with their network parameters are given to a circuit simulator for active analysis. Near and far field plots are also generated to study the passive behavior of the antenna array.

The next subsections describe the individual blocks of *ArraySim* used in passive analysis.

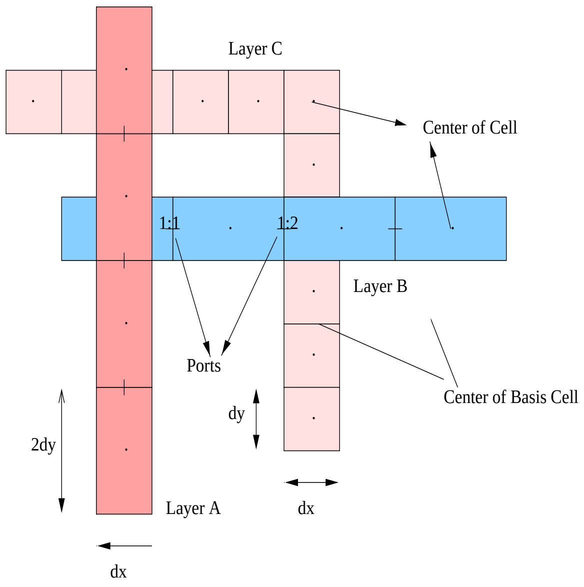


Figure 2.5: A Sample Three Layer Layout

2.3.1 Layout Design Rules

ArraySim allows users to draw their antenna structure using a layout editor. Presently, up to three layers of material can be used in laying out an array. In our examples, we used Cadence's Virtuoso as a layout tool. For proper parsing of the layout CIF file, a few design rules must be observed. These design rules include;

- Cell Size: Presently, all cell sizes of a particular layer must have the same dimensions. In short, a layer 'A' cell will have the same dx and dy as any other cell of layer 'A'. However, Layer 'B' cells may be different in size from layer 'A' cells but a layer 'B' cell will also have the same dimensions as any other layer 'B' cell. Same rule holds for layer 'C' cells.

- Cell Shape: Presently, the *ArraySim* parser can only handle rectangular or square cells 2.6. These polygons are called boxes in CIF terminology.

- Cell Overlap: Same layer cells are not allowed to overlap one another. However, cells of another layer can overlap cells of a different layer i.e. layer 'A' cells can overlap layer 'B' and 'C' cells but not layer 'A' cells 2.7.

- Cell layout: Cells of a certain layer must be directed either in the x-axis or 1 y-axis direction i.e. they cannot have an inclination angle associated with them as shown in Figure 2.8.

It is important to recognize the difference in meanings of the term basis cell and cell. A cell like a basis cell has x and y dimensions and a center described in x and y coordinates. Basis cell, however, is formed by two abut cells and is described by the same dimensions as that for a cell but its center is offset by half the x-dimension, dx, or y-dimension, dy, depending

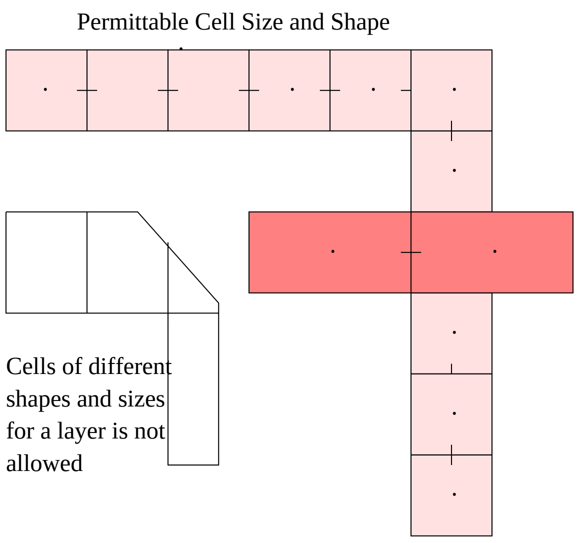

Permittable Cell Size and Shape
Cells of different shapes and sizes for a layer is not allowed

Figure 2.6: Allowed Cell Shapes

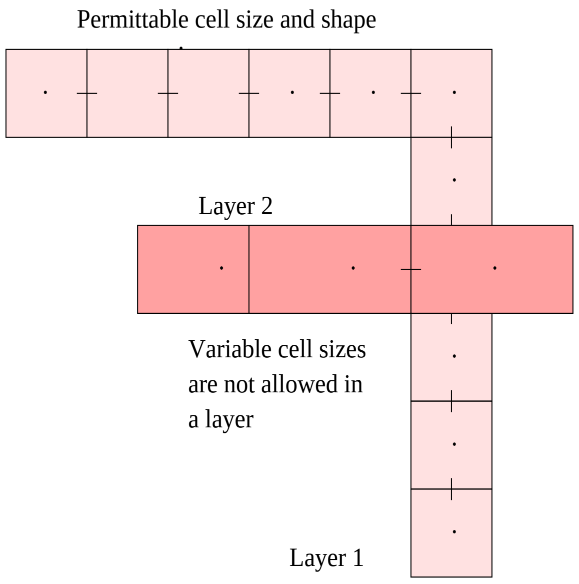

Permittable cell size and shape
Layer 2
Variable cell sizes are not allowed in a layer
Layer 1

Figure 2.7: Cell cannot overlap one another in the same layer

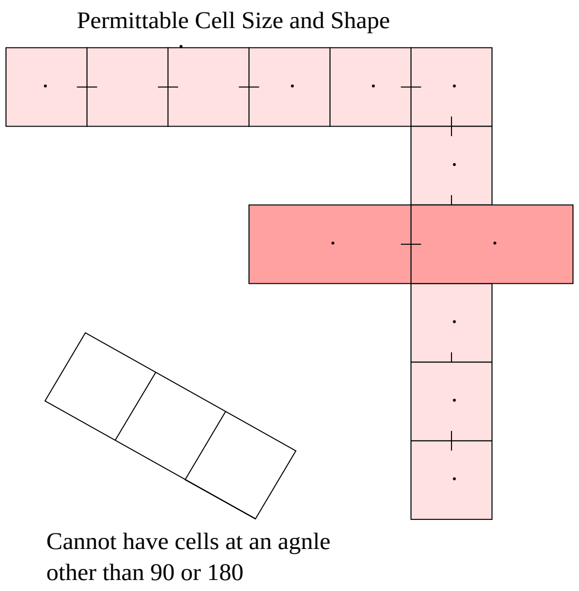

Permittable Cell Size and Shape
Cannot have cells at an agnle
other than 90 or 180

Figure 2.8: Inclination of cells is not allowed

if it is an x-directed basis cell or y-directed basis cell. Figure 2.5 shows a sample layout of different layer cells.

Ports in a layout are described using a *Portnum*:*Portgroup* format. In Virtuoso, a layout editor from Cadence Design Systems Inc., a port is added as a label at the center of a basis cell. It is not allowed to place a port at the center of a cell. Figure 2.5 shows two ports on layer B. The CIF file is later parsed in another format which is more readable. The later CIF conversion of the CIF file is created by a CIF parser. A sample CIF file containing cell information is shown below.

```
(CIF file written on 15-Sep-1998  22:37:02 by CADENCE);
DS 1 1 1;
9 3L_UNIT_10GHZ_13_7.5_3;
L CMF;B 3000 4500 9500,-67500;B 3000 4500 27500,-67500;
B 3000 4500 6500,-67500;B 3000 4500 24500,-67500;
B 3000 4500 21500,-67500;B 3000 4500 18500,-67500;
B 3000 4500 15500,-67500;B 3000 4500 12500,-67500;
B 3000 4500 3500,-67500;94 1:2 23000,-67500;
94 1:1 20000,-67500;B 3000 4500 39500,-67500;
B 3000 4500 36500,-67500;B 3000 4500 33500,-67500;
B 3000 4500 30500,-67500;L CVA;B 1500 1000 33500,-69000;
B 1500 1000 33500,-66000;B 1500 1000 33500,-67000;
B 1500 1000 33500,-65000;B 1500 1000 33500,-75000;
B 1500 1000 33500,-70000;B 1500 1000 33500,-63000;
B 1500 1000 33500,-71000;B 1500 1000 33500,-74000;
B 1500 1000 33500,-72000;B 1500 1000 33500,-73000;
B 1500 1000 33500,-68000;B 1500 1000 33500,-61000;
B 1500 1000 33500,-60000;B 1500 1000 33500,-64000;
B 1500 1000 33500,-62000;L CPG;B 1500 1000 9500,-71000;
B 1500 1000 9500,-70000;B 1500 1000 9500,-74000;
B 1500 1000 9500,-72000;B 1500 1000 9500,-73000;
B 1500 1000 9500,-75000;B 1500 1000 9500,-63000;
```

```
B 1500 1000 9500,-62000;B 1500 1000 9500,-64000;
B 1500 1000 9500,-60000;B 1500 1000 9500,-61000;
B 1500 1000 9500,-68000;B 1500 1000 9500,-67000;
B 1500 1000 9500,-69000;B 1500 1000 9500,-66000;
B 1500 1000 9500,-65000;
DF;
C 1;
E
```

In this CIF file, L is a layer symbol that follows with layer information. CMF, CVA and CPA are three different layer types in order of first, second and third layer respectively. Ports are identified by the colon symbol in the CIF file. B stands for box described by its horizontal and vertical dimensions and center coordinates.

2.3.2 CIF Parser and Modified CIF File

The raw CIF file shown in previous section is re-ordered in a certain fashion that allows to read each layer information, step by step. The new CIF file is arranged so that top most layer is on the top in the modified CIF file followed by the middle layer and then the lowest layer in the layout. This order is achieved by selecting poly, via and active for layer 1, layer 2 and layer 3 respectively. Ports are labeled with lowest layer material i.e. active material for a three layer structure. If above convention is followed, port information is ordered at the end of the CIF file. For a single layer structure, both layer and port label are made from the same material. For a two layer structure, upper layer is made from poly, while lower layer is made from metal. Ports in this case are labeled using metal layer. In any case, ports are always labeled using the lowest layer material. A modified CIF file is shown below:

```
(Plain CIF file);
DS 1 1 1;
9 Expanded_Array;
L CPG;
B 1500 1000 9500 -65000;
B 1500 1000 9500 -66000;
B 1500 1000 9500 -69000;
B 1500 1000 9500 -67000;
B 1500 1000 9500 -68000;
B 1500 1000 9500 -61000;
B 1500 1000 9500 -60000;
B 1500 1000 9500 -64000;
B 1500 1000 9500 -62000;
B 1500 1000 9500 -63000;
B 1500 1000 9500 -75000;
B 1500 1000 9500 -73000;
B 1500 1000 9500 -72000;
B 1500 1000 9500 -74000;
B 1500 1000 9500 -70000;
B 1500 1000 9500 -71000;
L CVA;
B 1500 1000 33500 -62000;
B 1500 1000 33500 -64000;
```

```
B 1500 1000 33500 -60000;
B 1500 1000 33500 -61000;
B 1500 1000 33500 -68000;
B 1500 1000 33500 -73000;
B 1500 1000 33500 -72000;
B 1500 1000 33500 -74000;
B 1500 1000 33500 -71000;
B 1500 1000 33500 -63000;
B 1500 1000 33500 -70000;
B 1500 1000 33500 -75000;
B 1500 1000 33500 -65000;
B 1500 1000 33500 -67000;
B 1500 1000 33500 -66000;
B 1500 1000 33500 -69000;
L CMF;
B 3000 4500 30500 -67500;
B 3000 4500 33500 -67500;
B 3000 4500 36500 -67500;
B 3000 4500 39500 -67500;
B 3000 4500 3500 -67500;
B 3000 4500 12500 -67500;
B 3000 4500 15500 -67500;
B 3000 4500 18500 -67500;
B 3000 4500 21500 -67500;
B 3000 4500 24500 -67500;
B 3000 4500 6500 -67500;
B 3000 4500 27500 -67500;
B 3000 4500 9500 -67500;
94 1:1 20000 -67500;
94 1:2 23000 -67500;
DF;
E
```

where L stands for layer, B for box and ":" for port information. All dimensions of boxes drawn in layout tool must be extracted in microns μm. Micron is the normal way of expressing dimensions in layout editors as they are used for transistor level layout. *ArraySim* converts these microns into meters internally.

2.3.3 Input Parameters

The second input to *ArraySim* is a file containing a list of parameters. This file includes antenna type, frequency sweep, print options, incident field type, near and far field analysis information and layout material information like; substrate height and dielectric material. This file is parsed by *ArraySim* parser. The user is warned to follow the convention of units described in the input parameter file. A sample input parameter file is given below

```
           Enter data in following format.
value   Description
2       1  antenna type: 1:sss 2:cpw 0:not defined
2.0     2  frequency in GHz
```

```
0.813  3  substrate height in mm
2.2    4  epsilon_r
1      5  Search Table on/off.  1:on 0:off
1e-6   6  Search Table Tolerance Level (Default: 1e-6)
0      7  Print Individual Matrices 1:print 0:do not print
1      8  Print Inverted Matrix 1:print 0:do not print
2.0    9  START frequency (GHz). Should be same as in line 2
6.0    10 END frequency (GHz)
0.1    11 Step size for frequency range
180    12 Characteristic Imp of cpw in Ohms.
0      13 new port def flag. If 1 use new method for Y_nn
1      14 stub len factor. if 1:1 cell away 2:2 cell away..
*************************** INCIDENT WAVE INFO **************
1      15 E_o antenna excitation in V/m
0      16 Ein_x incident in V/m
1      17 Ein_y incident in V/m
*************************** FAR FIELD INFO ********
10     18 Far Field Distance in meters
*************************** NEAR FIELD INFO *******
0.078  19 near field window length (m) i.e l_x
0.09   20 near field window width (m)  i.e l_y
75     21 no. of columns (for resolution)
75     22 no. of rows (for resolution)
0.0008 23 distance away from array in meters
0      24 by pass field analysis if 1:bypass 0:perform
```

The above file is parsed by the *InternalParser* $C++$ object. The first column in the input file consists of values and flags (0 or 1), the second column are line numbers for Table 2.3.3 and third column are comments about the values. Flags consist of *SearchTable_flag*, *PrintMatrices_flag*, *PrintInvertedMat_flag*, *new_port_def_flag* and *bypass*. If flags are set to 1 then they are active. Active flags mean; search engine is enabled, print all sub-matrices, print inverted **G** matrix, use new port definition which is described for SSS type antenna structure and skip near and far field analysis. On the other hand if flags are set to 0, then they are disabled.

2.3.4 Internal Parser

The parser forms the first block of *ArraySim*. It parses the input file and the modified CIF file. It is a $C++$ object called *InternalParser*. It allocates data structures for basis cells, cells and ports for each layer. Figure 2.9 shows the object structure of *InternalParser*.

After reading the number of cells per layer, *InternalParser* dynamically allocates memory for cells of that layer. In this specific case, an array of *Cell* object is created. *Cell* is a $C++$ class that describes the attributes and behavior of a cell in a given structure. So, a *Cell* knows the layer it belongs to, its center, its x and y dimensions, and has some standard member functions. If the structure has three layers then three similar data structures are made. All Cell arrays are ordered with respect to their x axis position using selective sort routines [7]. Cells are ordered to facilitate the creation of basis cell arrays. The next step is to read the port information. Ports are detected by the ":" sign in the modified CIF file. *Port* is a $C++$ object that knows its location, the layer it belongs to, its future index in the

Table 2.1: InternalParser class member variables for storing above input data parameters

Line Number	Variable Name
1	antenna
2	frequency
3	height
4	epsilon_r
5	SearchTable_flag
6	max_tol
7	PrintMatrices_flag
8	PrintInvertedMat_flag
9	Start_freq
10	End_freq
11	Step_freq
12	Zc
13	new_port_def_flag
14	stub_len_factor
15	E0
16	Ein_x
17	Ein_y
18	farFieldDistance
19	window_length_x
20	window_length_y
21	numOfCols
22	numOfRows
23	nearFieldDistance
24	bypass

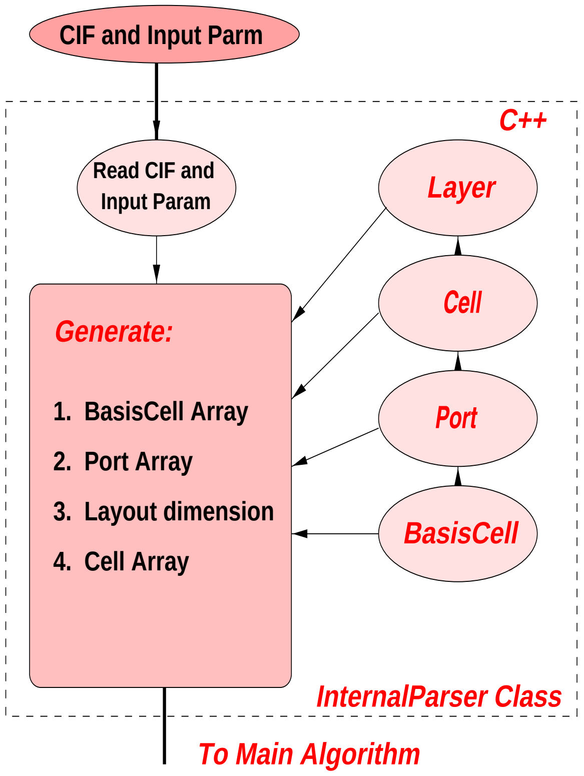


Figure 2.9: Object Structure of InternalParser Class

filled and inverted matrix and some standard $C++$ member functions. An array of ports is dynamically created depending upon the number of ports. The ports are then ordered with respect to their x-axis using selective sort routines.

After reading the input parameter file and the CIF layout, the *InternalParser* object begins to create basis cells. As defined before, a basis cell is composed of two cells and has the same dimension as a cell. However, unlike a cell, a basis cell is associated with a direction. A layer may have both x-directed and y-directed basis cells. So, to keep track of all basis cells in each layer, six basis cell arrays are created. Each layer will consist of two basis cell arrays; x basis cell array and y basis cell array. The nomenclature used in defining basis cell arrays is;

```
L1_BasisCell_x   //contains x-directed basis cells in layer 1
L1_BasisCell_y
L2_BasisCell_x
L2_BasisCell_y
L3_BasisCell_x
L3_BasisCell_y
```

Since all cells in a layer have the same size, a basis cell is formed if two cells are lying next to each other and the intersection plane of the two cells is equal to half the x-dimension of the cell, see Figure 2.10 (for an x-directed basis cell), or half the y-dimension of the cell, see Figure 2.11 (for a y-directed basis cell). i.e. for x-directed basis cell the center is computed as;

$$center_x = \frac{1}{2}d_x + x_{cell} \tag{2.9}$$

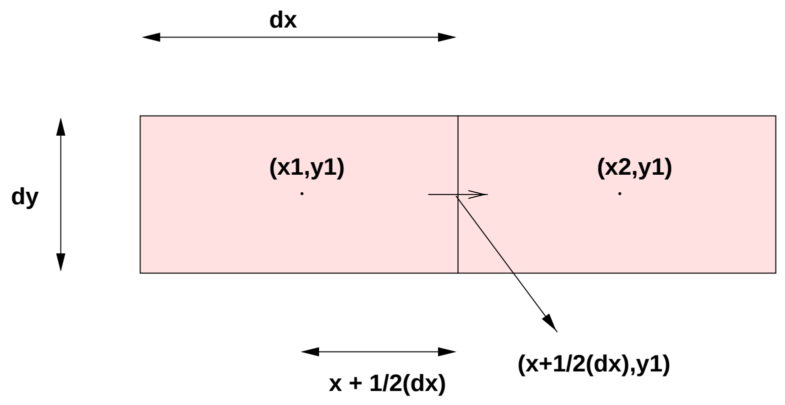


Figure 2.10: x-directed basis cell

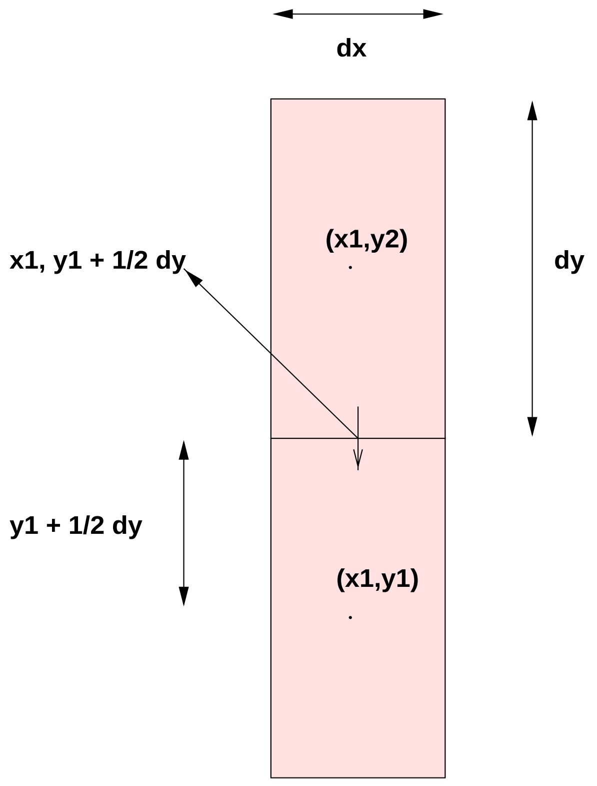


Figure 2.11: y-directed basis cell

for y-directed basis cell the center is computed as;

$$center_y = \frac{1}{2}d_y + y_{cell} \tag{2.10}$$

After locating all basis cells, the parser determines the total x and y dimensions of the layout. This is calculated by finding the minimum and maximum x and y centers of each layer cells. Then the maximum and minimum among them is chosen to determine the x and y dimension of the layout. This dimension is used by the interpolation routine to speed the calculation of the Greens function integral equations. So, the *InternalParser* object is responsible for

- Reading the input parameter file
- Reading the modified CIF layout file
- Indentifing all circuit ports
- Indentifing all basis cells in each layer
- Computing the total x and y dimension of the layout
- Creating data structures like; *BasisCell* Arrays, *Cell* Arrays *Port* Arrays and ordering them with respect to their x location in the layout

In the next subsection we define the *Port* Class that is used to describe the behavior and attribute of an electrical port in a structure.

2.3.5 Circuit Ports

Ports are used as amplifier terminals. For passive analysis, we do not connect an amplifier to the ports. However, port network parameters are calculated to study passive effects of an array structure. So, to make the existing code more modular, a *Port* class is created that defines the behavior and attributes of an electrical port in a structure. As mentioned earlier, ports are labels in the layout with the nomenclature *Port_num:Port_group*. *Port_num* may or may not be correctly labeled in the layout, but *ArraySim* internally keeps track of port numbers. However, it is important to give the port a *Port_group* number. Figure 2.12 gives an example of ports in a layout.

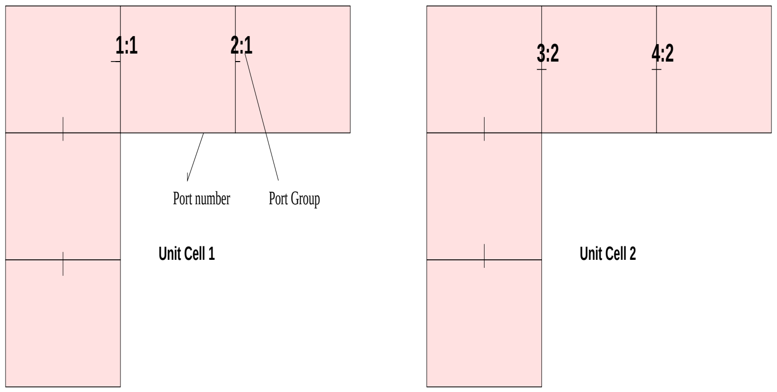


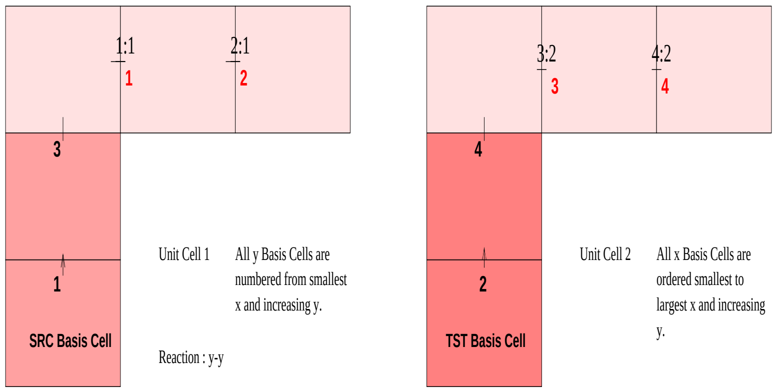
Figure 2.12: Port nomenclature

The ports with the same port group number belong to the same local reference group (LRG) and must share a common cell which becomes the local reference node (LRN). *Port_group* number defines the group to which the port belongs. So, a unit cell may have two ports and there may be several unit cells in an array e.g. in a 3x3 array there are nine unit cells which implies eighteen ports. Each unit cell is considered a group. A port in the first unit cell has a *Port_group* number 1. Similarly a port in the second unit cell has a group number 2. This nomenclature is used as an interface with the circuit simulator and also creates order among the ports. Thus a port is associated with a particular basis cell.

As the modified CIF file is being parsed and basis cells assigned, each basis cell is checked for its association with a port. If the basis cell is a port, then a port is identified and so the port object's *type* member variable is asserted.

Each port has an index number which is the position of the port in the filled or inverted matrix. This index is used to locate the ports from the large filled **G** matrix when port network parameters are being extracted. All ports are stored in an array of *Port* objects allocated by *InternalParser*.

2.3.6 MoM Based Analysis and Matrix Formulation



1:1
1
2:1
2
3
1
SRC Basis Cell
Unit Cell 1
All y Basis Cells are numbered from smallest x and increasing y.
Reaction : y-y
3:2
3
4:2
4
4
2
TST Basis Cell
Unit Cell 2
All x Basis Cells are ordered smallest to largest x and increasing y.

Figure 2.13: YY reaction of a source and test basis cell

Given two basis cells, their reaction with one another is found using the MoM routines. The result of this reaction is then stored in the **G** matrix depending upon the basis cell number of the source cell and then test cell. Both source and test cells are basis cells that can belong to the same layer or different layers. Figure 2.13 shows a reaction of a source and test basis cell such that both are in the y-direction. Such a reaction is referred as a yy reaction and the value obtained from this reaction is stored in the yy *Quad* of the sub-matrix (more on quad and sub-matrices in the next sections). The information needed to calculate the reaction are;

- dx and dy of both the source (src) and test (tst) basis cell
- center coordinates of both the source (src) and test (tst) basis cell
- direction of src and tst basis cell which is usually defined as flags i.e. flag 1: both src and tst are in x-direction flag 2: src in x-direction and tst in y-direction flag 3: both src and tst are in y-direction

Of course, depending upon antenna type, the corresponding antenna's MoM functions are called. All reactions are computed by the *MainAlg* object.

2.3.7 Hierarchical Matrix Composition

In order to fill the main matrix, the **G** matrix, with elements computed by MoM, a hierarchy of matrices is used. So, the main matrix is subdivided into sub-matrices depending upon the number of layers in the layout. The sub-matrices are further divided into quads which are 4 small matrices. *MainMatrix*, *SubMatrix* and *QuadMatrix* are $C++$ classes that are used to define the hierarchy of matrices. All matrices are single dimensional (1D) and their size are dynamically allocated by the *MainAlg* object discussed in the previous sub-section. As matrices are 1D, they are defined with an offset that is used to locate the correct element address in the **G** matrix. All elements computed by MoM are complex numbers. So, a *Complex* class is written with normal complex functions to define each element.

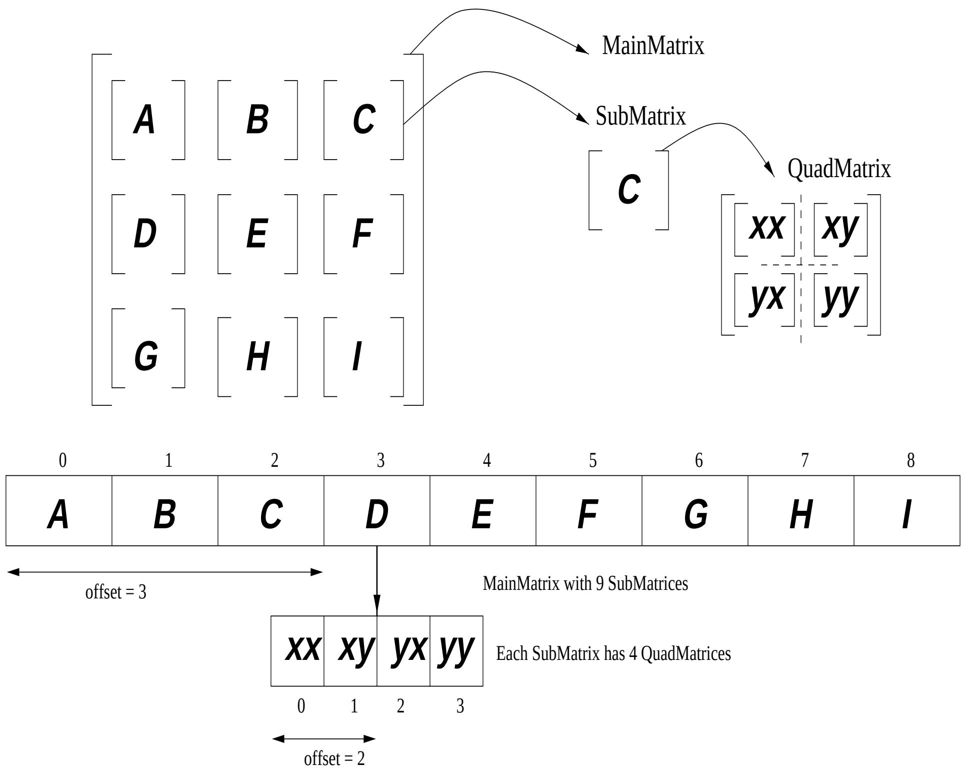


Figure 2.14: Matrix configuration in ArraySim

$$submatrices = n^2 \tag{2.11}$$

where n is the number of layers in the layout. So, for a 3 layer layout, the number of sub-matrices is 9.

The *QuadMatrix* class consists of four small matrices. The small matrices are named xx, xy, yx and yy which correspond to the src and tst basis cell direction. Usually, the yx matrix is simply the transpose of the xy matrix.

In order to form the *MainMatrix* object which consists of a number of *SubMatrix* objects (depending upon Equation 2.11), all *SubMatrix* objects have to be merged together to form one large *MainMatrix*. Before merging the *SubMatrix* objects, all *QuadMatrix* objects in a *SubMatrix* are merged together.

Figure 2.14 shows the construction of the *MainMatrix* from *SubMatrix* and *QuadMatrix* objects.

2.3.8 Matrix Filling

Matrix filling is done by the *MainAlg* object. After allocating size of *QuadMatrix*, *SubMatrix* and *MainMatrix* objects, *MainAlg* begins to compute the reaction of basis cells with one another. Depending upon the antenna type, corresponding MoM functions are used to calculate the reaction of two elements (src and tst basis cells). In essence, the *QuadMatrix* objects are filled first with respect to the direction of src and tst basis cells. Once all the *QuadMatrix* objects are filled up, they are merged together to form the respective *SubMatrix* object. After filling all *QuadMatrix* and thus *SubMatrix* objects, all *SubMatrix* objects are mergered together to form the *MainMatrix* object. This *MainMatrix* object is then solved using LU decomposition technique for further analysis.

Matrix filling is directly coupled with a Search Engine. In the next section we describe the working of Search Engine, a $C++$ object called *SearchTable*.

2.3.9 Search Engine for Symmetries

MainMatrix grows much faster than the increase in number of cells in a layout. Simulating large array structures especially three layered structures explodes the *MainMatrix* size. Consequently, the computation time is increased as more elements are computed for a given layout. Element computation time is a big blow to the overall analysis. So, in order to reduce computation time and use the CPU more efficiently, a search engine is introduced. This search engine is a $C++$ class called *SearchTable*. It is a collection of STL link lists [8] that hold elements that have already been computed. Figure 2.15 shows the search engine algorithm. For each *QuadMatrix* object, there is a corresponding link list in the search table. For example the XX *QuadMatrix* object has a link list to store XX reaction elements, XY *QuadMatrix* object has its own link list to store XY reaction elements and so on. The basic idea is that once an element has been computed for a certain src and tst cell then their is no need to re-compute a similar reaction between the src and tst cell. The criteria of deciding not to re-compute a reaction between a src and tst basis cell depends on the direction of the src and tst cells, the layer the src and tst cell belong to, the absolute x and y distance between the src and tst cell and the antenna type. *ArraySim* shows the number of elements that were not computed because of the search engine. The output of *ArraySim* for a sample run looks like;

```
A ONE LAYER STRUCTURE ONLY
CALLING L1_XX
 elm found: 24105 elm not found: 231**
CALLING L1_XY
 elm found: 1094 elm not found: 154**
TRANSPOSING QUAD1 to get QUAD 2 for L1_YX
CALLING L1YY
 elm found: 54 elm not found: 10**
```

The above example is a part of a one layer antenna MoM simulation. It clearly shows the benefit of using a search table. In the first part, the search engine link list size for an XX QuadMatrix object is only 231. This means out of a total of 24336 elements in the XX *QuadMatrix* object, only 231 had to be computed while 24105 elements were just fetched from the search table. This reduces the computation time by more than 70%. Presently, linear sequential search [7] is being used to search an element in the link list. Other search routines may be used to enhance searching speed. Let us consider an example of a layout

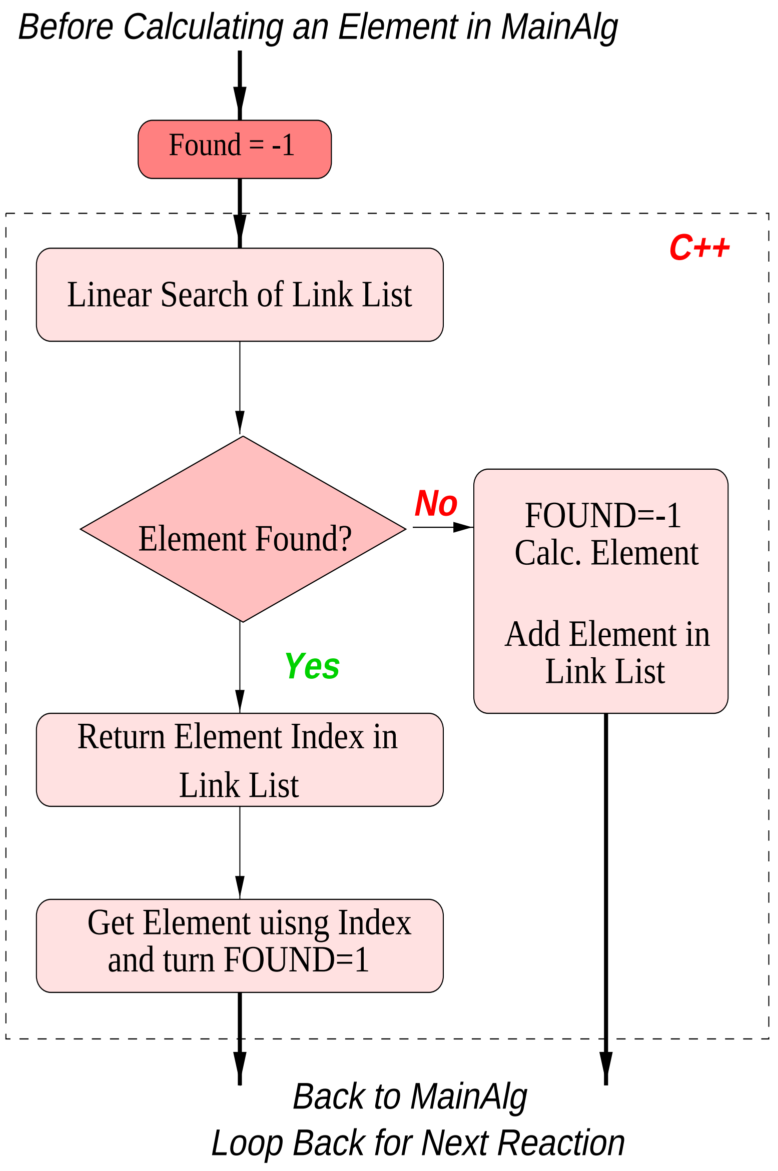

Before Calculating an Element in MainAlg
Found = -1
C++
Linear Search of Link List
Element Found?
No
FOUND=-1
Calc. Element
Add Element in Link List
Yes
Return Element Index in Link List
Get Element uisng Index and turn FOUND=1
Back to MainAlg
Loop Back for Next Reaction

Figure 2.15: Search Engine

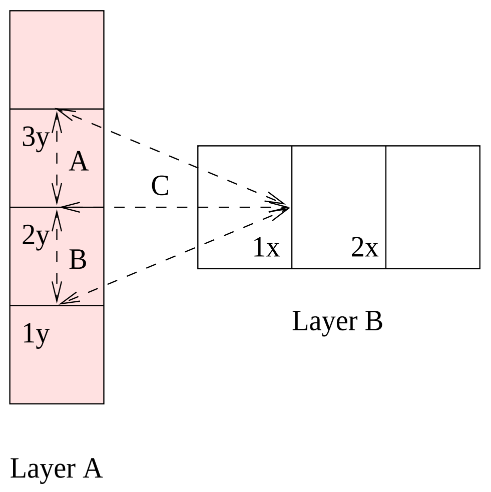


Figure 2.16: Symmetrical reaction between test and source basis cells of different layers is identified by search engine

with five cells consisting of three y-direction basis cells in layer A and two x-direction basis cells in layer B as shown in Figure 2.16.

Reaction between $1y$ and $1x$ is identical to the reaction of $3y$ and $1x$ because distances A, B and C are same in absolute terms. In such a case, link list XY, in the *SearchTable* object, contains reaction $1x$ and $1y$, and reaction $2y$ and $1x$ only because reaction $3y$ and $1x$ is identical to reaction $1y$ and $1x$ So, before computing a reaction between a src and tst basis cell, corresponding *SearchTable* link lists are traversed and checked for any similar reaction depending on absolute distances like A, B and C, if found then the reaction value is pulled out from the link list and MoM integral equations are avoided from being computed.

Depending upon the structure and its Greens function formulation, conditions to find identical reactions are different. For SSS structure, which is a three layer structure, we check for absolute x and y distances, and layers of tst and src basis cells i.e.

$$x_{ABSdistance} = |x_src - x_tst| \tag{2.12}$$

$$y_{ABSdistance} = |y_src - y_tst| \tag{2.13}$$

If $x_{ABSdistance}$ and $y_{ABSdistance}$, for a pair of src and tst basis cells in the *SearchTable* object link lists, is same as absolute x and y distances of the given src and tst basis cells then a symmetrical match is found and MoM based integral equation computation is by-passed. When checking for Equations 2.12 and 2.13 it is important to keep track of layer information of current src and tst basis cells and src and tst basis cells in search table link lists. Similar condition is checked for folded-slot single layer structures. However, in XY reactions (where tst is x-directed and src is y-directed), depending upon x and y distances (absolute and normal), placing a negative sign to element values may by-pass MoM computation. Figure 2.17 shows a reaction that results in same absolute values but with different signs.

$$sign_{current} = x_{distance}y_{distance} \tag{2.14}$$

$$sign_{ST} = x_{STdistance}y_{STdistance} \tag{2.15}$$

where Equation 2.14 gives the sign for current tst and src basis cell distances while Equation 2.15 gives the sign of tst and src basis cells in search engine link lists. If Equations 2.14 and

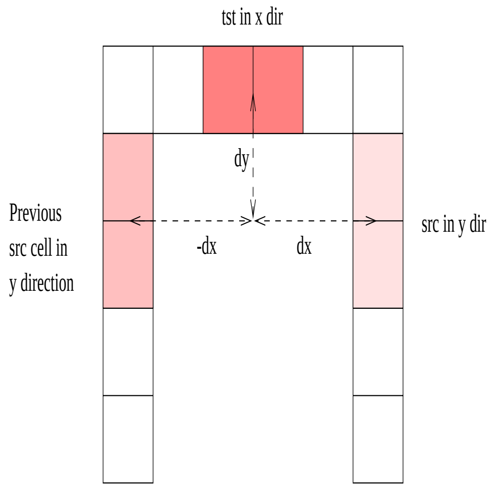


Figure 2.17: Reaction between highlighted test and source basis cells is equal in magnitude but opposite in signs

2.15 have different signs then the element in the search table link list with same absolute x and y distances is fetched and placed in the **G** matrix with a negative sign.

By using search routines to look for symmetries and avoid expensive and time consuming MoM calculations, we speed up overall performance of *ArraySim*. Chapter 4. tabulates some simulations for folded-slot structures showing effects of search tables on computation time.

2.3.10 Solving Matrix using LU Decomposition

Once the matrix is filled with complex elements, it is inverted in order to solve the system of equations for the unknowns. LU decomposition coupled with back substitution is used to invert the matrix [12]. All matrix manipulation is performed by the *MatrixManipulation* class. This class allows to add, multiply and solve complex matrices such that the matrices are 1D matrices. After inversion, the elements are ordered from [1..n] instead of [0..n-1]. This new notation allows us to use published algorithms about matrix manipulation. The inverted matrix is printed to a file called out_Inverse.txt . This file is read by the active analysis part of *ArraySim* in order to find near and far field plots.

2.3.11 Port Based Reduced Matrix

The next step is to extract only those reaction elements that are also ports. These elements are obtained when a reaction of a basis cell with itself is computed if that basis cell is labeled a port. So, for a 2 port network, the reduced matrix will be a 2×2 matrix. Similarly, for a 4 port network, the reduced matrix will be a 4×4.

As elements are being computed and filled in the matrix, a check is made for port basis cells. If both the src and the tst basis cell are same and have their instance variable, port, asserted then the port index number is set to new element position in the main matrix. This way all ports are tracked and their corresponding instance variable, *index*, is set.

Port based reduced matrix is formed by a function call in main_new.C file that contains the main of the program. These ports may be Y-ports or Z-ports depending up on the antenna structure. For folded-slot structures, port based reduced matrix is a Z matrix while for the SSS structure, the reduced matrix is a Y matrix. All port based matrices are converted into Y-port matrix for Transim input. Figure 2.12 has a 4 port network. The main matrix for this structure is a 8×8 matrix. However, the ports are all located in the xx quad matrix as shown in matrix below. The following matrix equation is an example of inverted **G** matrix from Figure 2.12.

$$\mathbf{Z} = \begin{bmatrix} \mathbf{z_{1,1}} & \mathbf{z_{1,2}} & \mathbf{z_{1,3}} & \mathbf{z_{1,4}} & z_{1,5} & \cdots & z_{1,8} \\ \mathbf{z_{2,1}} & \mathbf{z_{2,2}} & \mathbf{z_{2,3}} & \mathbf{z_{2,4}} & z_{2,5} & \cdots & z_{2,8} \\ \mathbf{z_{3,1}} & \mathbf{z_{3,2}} & \mathbf{z_{3,3}} & \mathbf{z_{3,4}} & z_{3,5} & \cdots & z_{3,8} \\ \mathbf{z_{4,1}} & \mathbf{z_{4,2}} & \mathbf{z_{4,3}} & \mathbf{z_{4,4}} & z_{4,5} & \cdots & z_{4,8} \\ \vdots & & & & & & \\ z_{8,1} & z_{8,2} & z_{8,3} & z_{8,4} & z_{8,5} & \cdots & z_{8,8} \end{bmatrix} \tag{2.16}$$

where all the bold elements make up the port based reduced matrix and **Z** is a *SubMatrix* object containing four sub-matrices (quads). This reduced matrix is then converted into S-parameters to study the behavior of the passive structure.

2.3.12 Network Parameters: Y,Z and S

After reducing the large main matrix into port based reduced matrix, we compute some common network parameters that are used to study the array structure behavior. Equations 2.17 and 2.18 are used to convert Y and Z port parameters into S port parameters [9]. Any plotting tool like xess, gnuplot etc. may be used to view the S-parameters. S-Parameters are written in *out_S_Port.txt*.

$$[S] = [U + Y]^{-1} [U - Y] \tag{2.17}$$

$$[S] = [Z + U]^{-1} [Z - U] \tag{2.18}$$

2.3.13 Formation of Excitation Vectors

Excitation vector depends upon the incident field on the array. In the input parameter file, the user is given the option of setting the intensity of incident electric field. Presently, incident fields are set either in the x or y-direction. If Ein_x is 0 and Ein_y is 1, then it means that the incident electric field is only in the y-direction (with Hx component) and has a magnitude of 1 V/m. The excitation vector is defined in A/m as;

$$[Hin_x] = \frac{2d_x}{Z_0} [Ein_y] \tag{2.19}$$

$$[Hin_y] = \frac{2d_y}{Z_0} [Ein_x] \tag{2.20}$$

where Z_0 is the free space impedance (377 Ω), d_x is the upper layer's basis cell x dimension, d_y is the upper layer's basis cell y dimension. For active analysis the excitation vector is adjusted with amplifier currents and voltages obtained from non-linear circuit simulation discussed in the next section.

2.3.14 Interface with Non-Linear Circuit Simulator

Active analysis requires linear and non-linear circuit simulation of amplifiers being used in array structures. For this purpose, an interface [15] is developed between *ArraySim* and

Transim, a non-linear circuit simulator. After extracting port based reduced matrix from main matrix, we extract port based currents/voltage sources. If the reduced matrix is an impedance matrix, then it is inverted into admittance matrix for Transim interface. A sample Transim input file is given below;

```
File # 1:  out_TransimPortInfo.txt
# port:group
1:1
# GHZ Y RI R 50
2
0.0233506 ,0.00380929
File # 2:  out_TransimExitationInfo.txt
2.0000000000e+09
7.6385605435e-01 ,-1.3684498083e+00
```

File 2. contains either short circuit current sources or open circuit voltage sources that are used as initial conditions for non-linear circuit simulation. The first parameter in that file is frequency in GHz and then the real and imaginary current/voltage value. Figure 2.18 shows an n port matrix with short circuit current and open circuit voltage sources respectively. These figures form the input to the circuit simulator that computes the active currents and voltages at the ports.

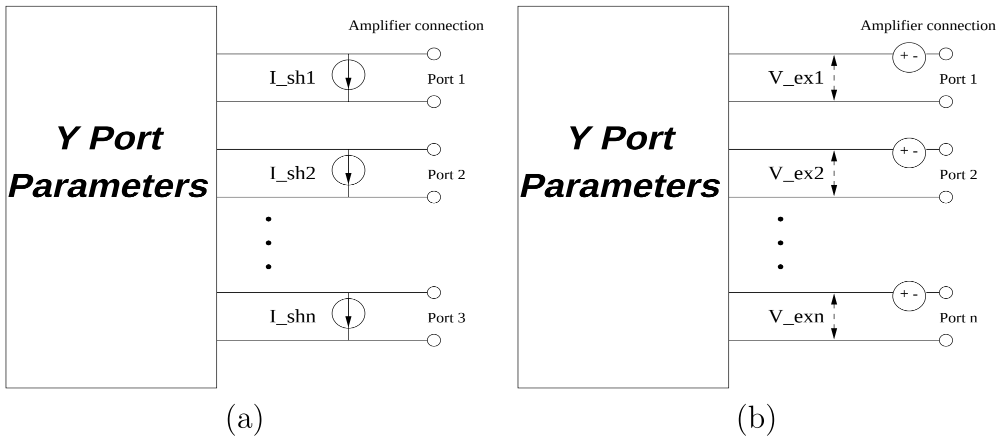


Figure 2.18: Y Port model for Transim: (a) used by folded-slot based CPW Structure (b) used by SSS Structure

2.3.15 Near and Far Field Analysis

Field profile is the last part of passive analysis. It gives an overall behavior of the array. Far field analysis considers the array as a point charge and calculates the field due to the point charge. Far field analysis thus gives the direction of the output field due to the incident field using polar plots.

Near field analysis, on the other hand, gives the field distribution on the array. The user is allowed to view the near field plots for different distances away from the array. In the input file, the user sets the display window size as well as the resolution of the plot. On completion of passive analysis for a frequency point, the user can study the near field analysis for different inputs and then allow *ArraySim* to run for next frequency. Figure 2.19 is a sample output of near and far field analysis which is displayed using matlab [10]. Magnetic currents that are used as input for both near and far field analysis are computed as;

$$[M_x] = \left[\frac{V_x}{d_y}\right] \tag{2.21}$$

$$[M_y] = \left[\frac{V_y}{d_x}\right] \tag{2.22}$$

where d_x and d_y are x and y dimensions of layer 1 basis cells and V_x and V_y are the unknowns computed by solving the **G** matrix. The object responsible for near and far field analysis is

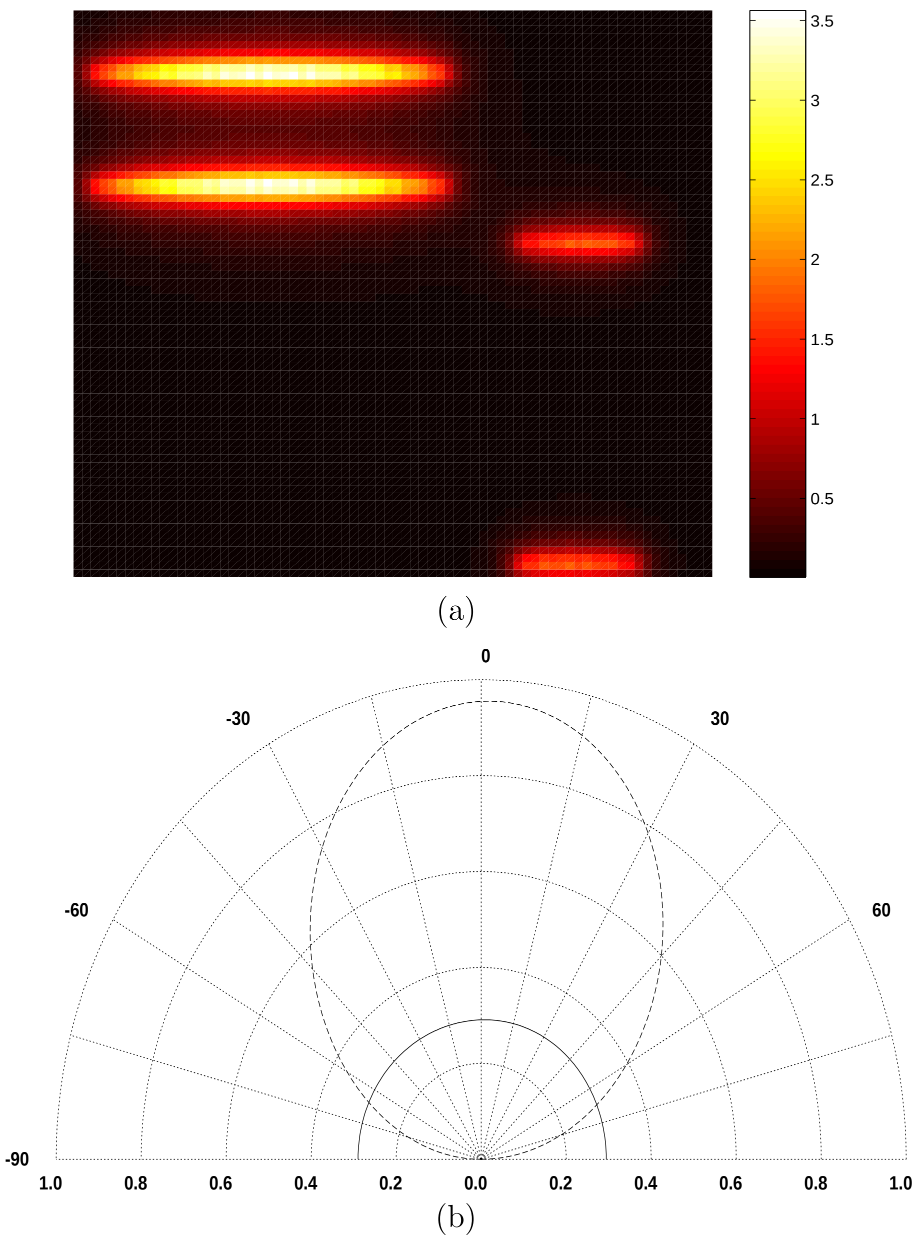


Figure 2.19: Field Profile: (a) Near Field Plot of a folded-slot (FS) at a distance of 5 mm at 2GHz (b) Far field plot of a FS at a distance of 10 m at 2GHz

FieldAnalysis. Depending on the kind of antenna, the respective near and far field functions are called.

2.4 Hierarchical Modeling Philosophy

The previous section on passive analysis forms the basis of hierarchical modeling approach. MoM analysis is not an iterative process like $FTDA$. In MoM, once the **G** matrix is filled, it can be used for both active and passive analysis. In active analysis, amplifier currents and voltages are computed that are added to the existing excitation vector to form a new excitation vector. This new excitation vector when multiplied with the original inverted matrix, gives new values for currents. The basic idea is the computation of **G** matrix after that, remaining problem becomes more or less a circuit problem.

ArraySim provides a perfect environment for hierarchical modeling of array structures. Its object oriented design methodology allows to include MoM functions of other structures into *ArraySim* with minor adjustments. The basic algorithm of parsing, assigning basis cells identifying circuit ports, searching for symmetries when filling the **G** matrix, and extracting network parameters remains same for all structures.

2.5 Active Analysis

After solving for non-linear currents and voltages using Transim, active analysis is performed. Active analysis involves, recomputing the excitation vector with new currents or voltages and then solving for output magnetic currents, M_x and M_y. These currents are then used for near and far field analysis. Figure 2.20 shows a step-by-step process of active analysis. This analysis is much faster than passive analysis as the **G** matrix is not computed.

2.5.1 Inputs from Passive Analysis

Most of the inputs to active analysis procedure are obtained from passive analysis of the structure. These inputs include, old excitation vector that was formed using incident E field, inverted **G** matrix and port row numbers in the inverted matrix. Port row numbers are used to fill the new excitation vector, E2 (see Figure 2.20), correctly with Transim port currents or voltages.

2.5.2 Inputs from Circuit Simulator

Presently, *ArraySim* is interfaced with Transim, a non linear Spice like circuit simulator. Transim, takes the Y-port parameters along with current or voltage sources and performs harmonic balance analysis for a non-linear amplifier. It then outputs currents and voltages at the amplifier ports. These currents and voltages are used as input for active analysis by *ArraySim* to form the new excitation vector.

2.5.3 Formation of Excitation Vector

Using port row information, a new excitation vector is formed that is filled with current or voltage values from Transim in port row number locations. The remaining elements of this excitation vector, E2, are filled with zeros. This vector is then added with E1, excitation

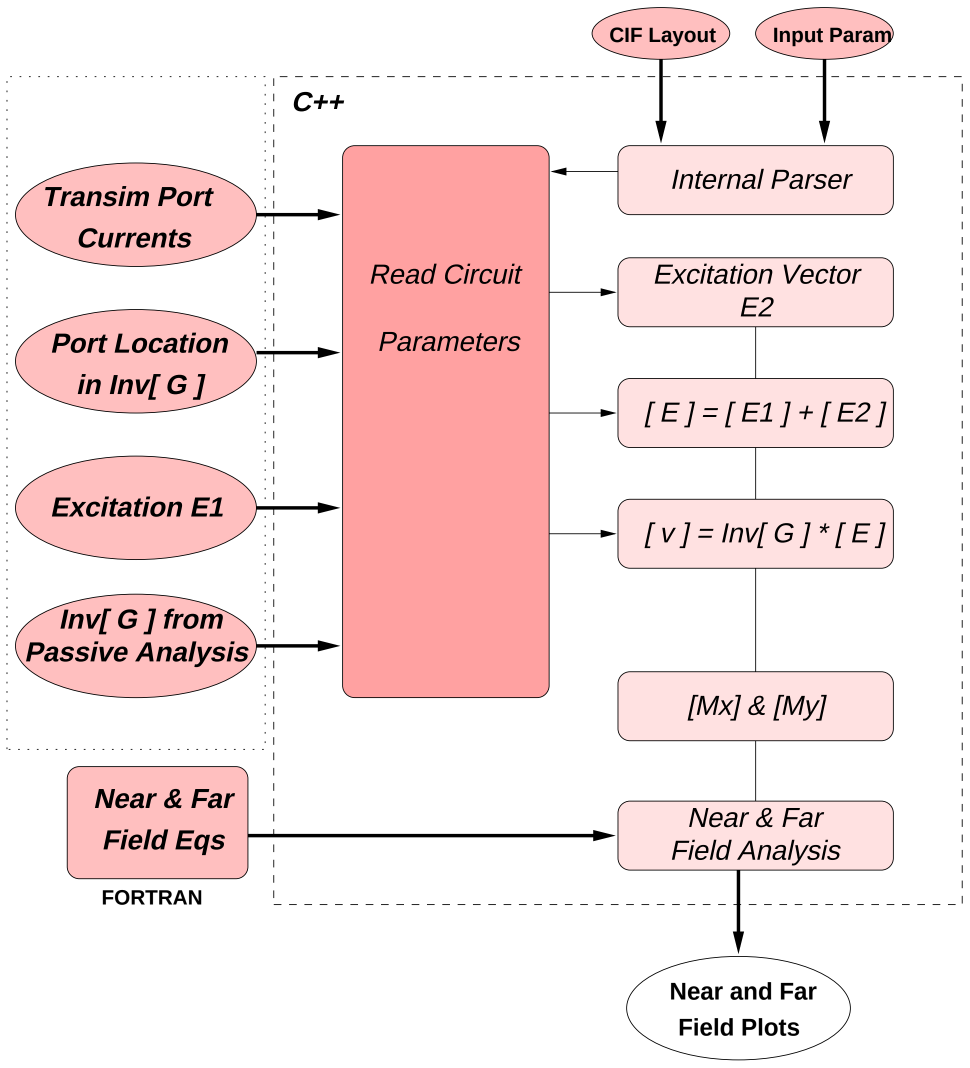


Figure 2.20: Active Analysis Flow

vector used in passive analysis, to form a new excitation vector for active analysis. M_x and M_y vectors are then computed with this new excitation vector. These M_x and M_y are used for near and far field analysis in the same way as described in passive analysis procedure.

$$\mathbf{E_{new}} = \begin{bmatrix} E1_1 \\ \mathbf{E1_2} \\ E1_3 \\ \mathbf{E1_4} \\ E1_5 \\ \vdots \\ E1_8 \end{bmatrix} + \begin{bmatrix} 0 \\ \mathbf{E2_2} \\ 0 \\ \mathbf{E2_4} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{2.23}$$

In Equation 2.23, the right hand side vector is created using port row information and Transim currents or voltages. The left hand matrix, on the other hand, is the old excitation vector. The bold elements in these vectors refer to excitation at the ports. E_{new} is then used to find new M_x and M_y vectors that are used for near and far field analysis as shown in 2.24.

$$[f_m] = [G]^{-1} [E_{new}] \tag{2.24}$$

where $\mathbf{f_m}$ holds M_x and M_y values.

2.6 Conclusion

ArraySim provides an environment for hierarchical modeling of antenna arrays used in spatial power combiners. It performs active and passive analysis of any structure as long as their MoM based functions are provided. In this chapter we discussed various modules of *ArraySim*. These modules are C++ objects that talk with each other in a defined way. Using object oriented methodology, hierarchical modeling of antenna arrays has become suitable. The analysis is divided into three parts, passive, circuit and active. An interface with a circuit simulator is established to solve for port currents and voltages due to amplifiers. These new currents and voltages are then used to create a new excitation vector that computes new output magnetic currents. In the next chapter we look at some simulation and experimental results for SSS and folded-slot array structures using *ArraySim*.

Chapter 3

Investigating SSS Structure

ArraySim is used for behavior modeling of structures. It allows users to study the behavior of a given structure provided its MoM functions are interfaced correctly in the *MainAlg* object. In this chapter we use MoM functions developed by [14] for slot-stripline-slot (SSS) structure. This three layer antenna has a complicated behavior due to coupling between layers. The MoM matrix for such structures are usually large and take much time to solve. However, if the lower and upper slots are symmetric, *ArraySim* uses this symmetry to speed up computation time. Various simulations are done on SSS structures which include unit cells and arrays. Since this particular three layer structure as shown in Figure 3.1 has not been experimented before, we use a two layer slot-stripline structure and compare simulation results with published results. Simulation results show close agreement with published results and thus verifies *ArraySim*. Effect of search table is also discussed by comparing simulation time with and without search tables.

3.1 Simulation of SSS Array Structure

Slot-Stripline-Slot antenna consists of three layers. The upper layer and the lower layers have slots cut into the metal while the middle layer is a stripline sandwiched between the upper and lower slots. Figure 3.1 shows a unit cell description of a SSS antenna. Active device is connected in the middle layer. The basic idea is that incident electric field on the upper slot induces current on the stripline. This current is then amplified by the active device and radiated out though the lower layer slot. Before simulating an array structure, it is worth analyzing the behavior of a unit cell as array usually behaves similar to a unit cell. For symmetry purpose, upper and lower layer are made identical with respect to the stripline, the middle layer. Replicating unit cells side by side results into an array of unit cells. Figure 3.7 shows a 3×3 array of SSS type. In spatial power combiners, arrays are made with active devices that are used to gain high output power.

3.1.1 Matrix Configuration

SSS is a three layer structure and its **G** matrix consists of nine *SubMatrix* objects. Each *SubMatrix* object further includes a *QuadMatrix* object consisting of four small matrices as discussed in the previous chapter. Figure 3.1 is based on equivalence principle and therefore decomposes the structure into three different regions. MoM matrix for this three layer

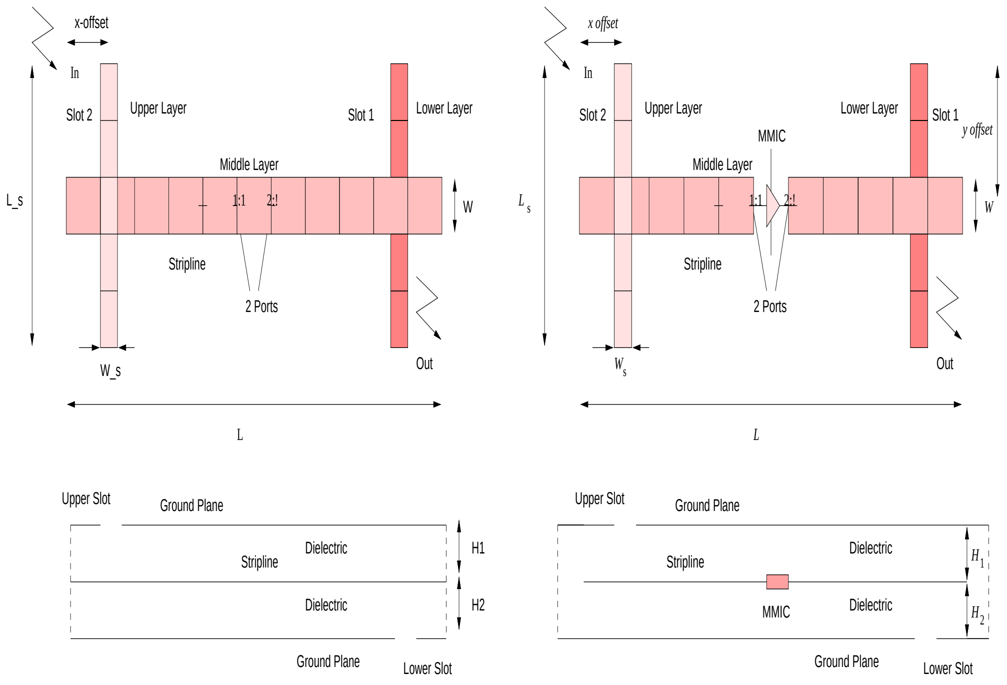


Figure 3.1: 3 Layer SSS Unit Cell: (a) Passive (b) Active

structure is given by Equation 3.1 [14];

$$\begin{bmatrix} \mathbf{H_{upper}} \\ \mathbf{V_{strip}} \\ \mathbf{H_{lower}} \end{bmatrix} = \begin{bmatrix} \mathbf{Y_{11}} & \mathbf{U_{12}} & \mathbf{Y_{13}} \\ \mathbf{W_{21}} & \mathbf{Z_{22}} & \mathbf{W_{23}} \\ \mathbf{Y_{31}} & \mathbf{U_{32}} & \mathbf{Y_{33}} \end{bmatrix} \times \begin{bmatrix} \mathbf{M_{upper}} \\ \mathbf{I_{strip}} \\ \mathbf{M_{lower}} \end{bmatrix} \tag{3.1}$$

where $[Y]$,$[U]$,$[W]$ and $[Z]$ are mutual coupling integrals between slots and the stripline. $[M_{upper}]$,$[I_{stripline}]$ and $[M_{lower}]$ are unknown coefficients of the basis function on receiving slot, stripline and transmitting slot respectively. $[H_{upper}]$ is the excitation vector on the upper slot due to incident fields (assuming excitation is unidirectional and not from lower slots).

If lower and upper slots are made identical with respect to each other and stripline, then *ArraySim* recognizes this symmetry and only computes matrices, $[Y_{11}]$, $[W_{21}]$, $[Z_{22}]$ and $[Y_{13}]$. All other sub-matrices are filled by symmetry. Individual sub-matrices are also filled using search routines that further reduces computation time. The only drawback in simulating this kind of structure is memory. Large memory is needed to store all nine sub-matrices and then solve the whole matrix.

For a two layer structure shown in Figure 3.2a, the MoM based matrix reduces to four *SubMatrix* objects. Equation 3.2 described in [13] shows the matrix configuration of a 2 layer SS type structure. *ArraySim* uses a search engine to find symmetries when filling and thus reduce computation time.

$$\begin{bmatrix} \mathbf{H_{in}} \\ \mathbf{V_{strip}} \end{bmatrix} = \begin{bmatrix} \mathbf{Y_{11}} & \mathbf{U_{12}} \\ \mathbf{W_{21}} & \mathbf{Z_{22}} \end{bmatrix} \times \begin{bmatrix} \mathbf{M_{slot}} \\ \mathbf{I_{strip}} \end{bmatrix} \tag{3.2}$$

In the next subsection we use *ArraySim* as a tool to study a two layer slot-stripline (SS) published structure and compare simulations with published results. We then investigate the behavior of simulated results by changing stripline basis cell numbers.

3.1.2 2 Layer Unit Cell Simulation

A two layer unit cell is shown in Figure 3.2. It is similar to unit cell in Figure 3.1, but with one slot only. The dimensions of this unit cell are;

```
W = 2.6 mm  of stripline
L = 30  mm  of stripline
H1=H2=1.57 mm or (62mil) thickness of substrate
W_s = 2  mm  of slot
L_s = 30 mm  of slot
x_offset = 10mm
y_offset = 0
```

$\epsilon_r = 2.2$

The stripline width is selected as 2.6 mm as this width gives an impedance of 50 ohms to the stripline. Equation 3.3 (for $W \geq$ 2H) and 3.4 (for $W \leq$ 0.4H) from [5] were used to compute the characteristic impedance, Z_c of the stripline.

$$Z_c = \frac{\pi Z_0}{8\sqrt{\epsilon_r}(\ln 2 + \pi W/(4H)} \tag{3.3}$$

For a very narrow strip,

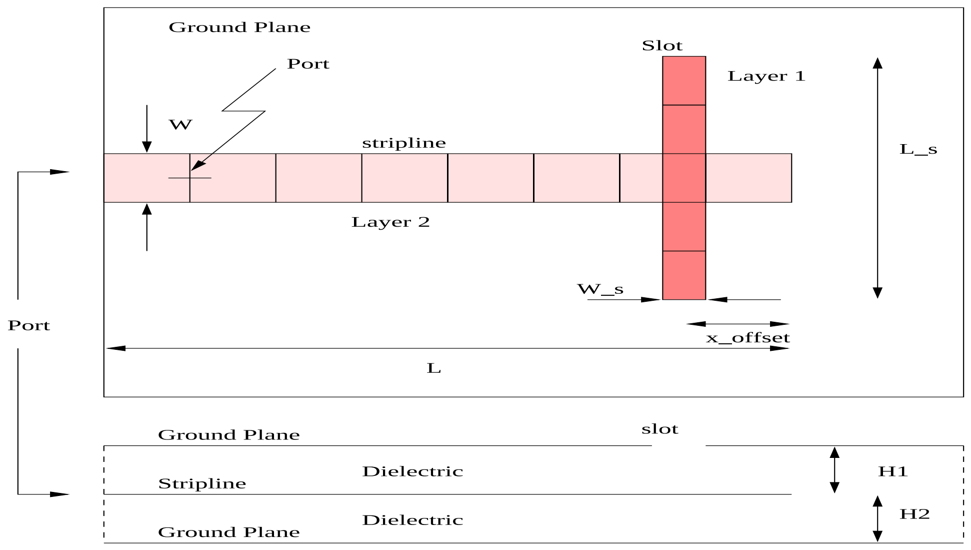
$$Z_c = \frac{Z_0}{2\pi\sqrt{\epsilon_r}} \ln \frac{16H}{\pi W} \tag{3.4}$$

ArraySim was used to simulate this simple 2 layer structure with the given dimensions. S-parameters were extracted for this structure and compared with published results [13]. A close agreement is seen in simulated and published results which shows the correct working of *ArraySim*. The simulated S_{11} plot is slightly shifted to the right of the published S_{11} plot. This shift is due to the number of stripline basis cells. Increasing stripline basis cell number shifts the curve to the left. More on effects of basis cell numbers in the next subsection.

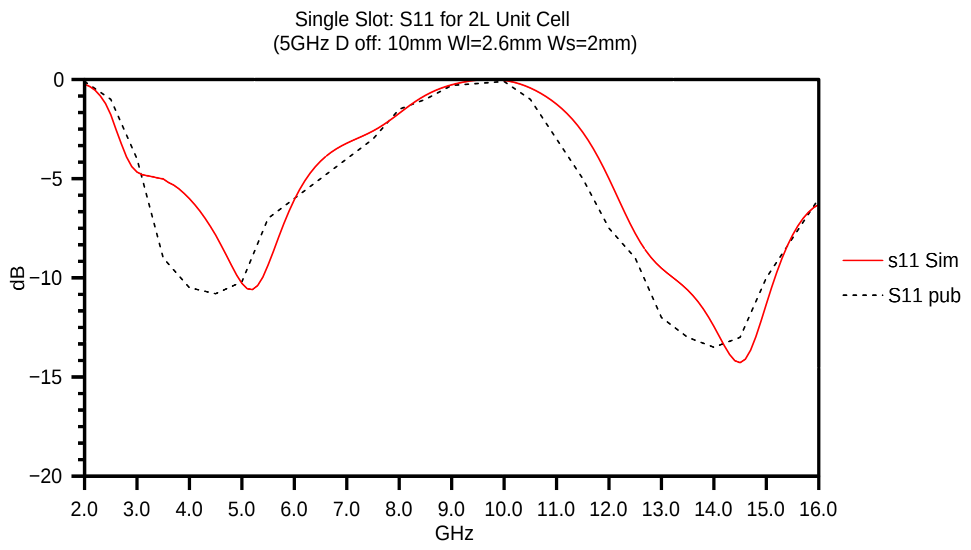
3.1.3 Effect of Basis Cells Numbers on Simulations

A test simulation was performed to check the behavior of a unit cell when number of basis cells were changed. Figure 3.3 shows simulation result of a series of simulations with different number of cells for a 3 layer SSS unit cell shown in 3.1. Dimensions for this unit cell were;

```
Stripline:
W = 4.5 mm
L = 38.75 mm
Slots:
W_s = 1.5 mm
L_s = 16 mm
x_offset = 7.5 mm
Dielectric:
Epsilon_r = 2.2
H1=H2=1.57 mm
x offset of unit cell = 20 mm
y offset of unit cell = 20 mm
x_dist = 156.25 mm
y_dist = 88 mm
```



(a)



(b)

Figure 3.2: 2 Layer Slot Stripline Unit Cell: (a) Structure layout (b) Comparing simulations and published results

$\epsilon_r = 2.2$ It is observed that as cell number increases from 13 to 55 for the same length of stripline, the curve tends to shift to the upper left side. This behavior is attributed to MoM dependence on number of integration points. Figure 3.3 plots $S_{11}^2 + S_{12}^2$ in magnitude.

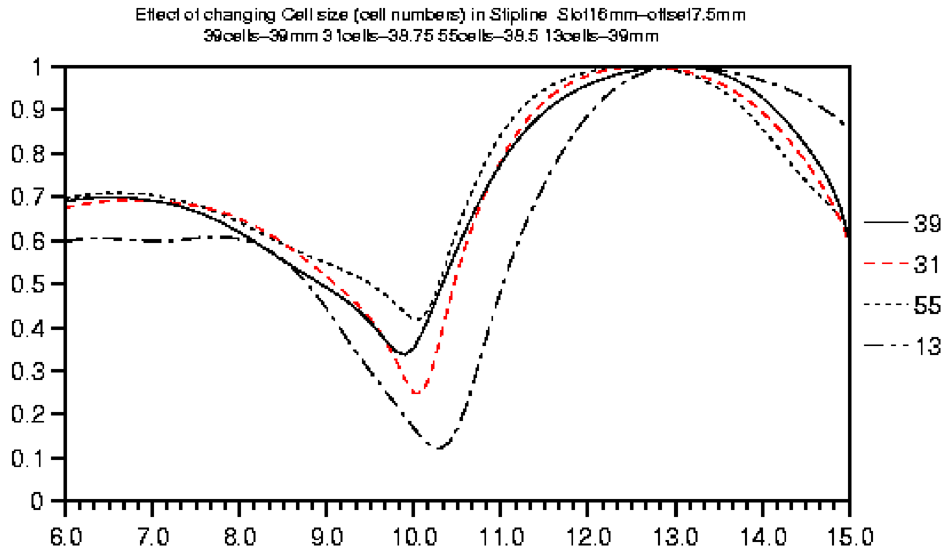


Figure 3.3: Effect of number of cells on simulation result

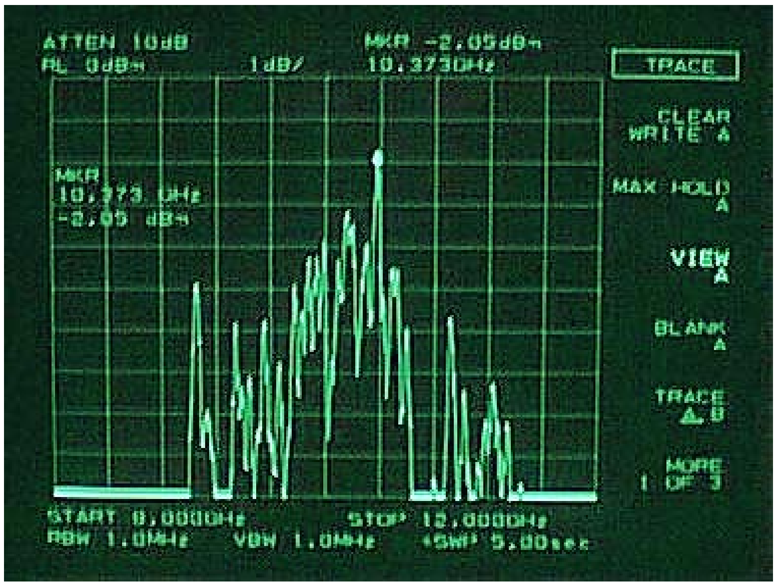


Figure 3.4: Measurement showing minimum insertion loss around 10.3 GHz as expected by simulation results

Since, SSS is a lossy structure, we observe that around 10 GHz, most of the input power is absorbed i.e. minimum reflection. This structure was fabricated and some measurements were made. Measurements show a minimum insertion loss of 2 dB near 10.3 GHz which agrees with simulated behavior as seen in Figure 3.4. An HP Spectrum analyzer was used to measure insertion loss. Scattering parameters were not measured because it was not possible to drill into the stripline. Drilling would have caused to change the boundary conditions used in formulation of Greens functions.

3.1.4 Differential Versus Normal Ports

In SSS, the middle layer is a stripline with an active device. So, the stripline is sandwiched between two layers separated with a dielectric medium that makes it impossible to refer the amplifier ports to ground. The Y-port parameters obtained by MoM in this case are differential ports rather than normal ports referring to ground. So, to get the correct Y-port parameters, the MoM based parameters are adjusted for an open circuit stub that is matched with the input impedance. Equation 3.5 is used to calculate Y-port parameters in this case.

$$Y_{11_{new}} = \frac{1}{\frac{1}{Y11_{MoM}} + jZ_cCot(\beta a)} \tag{3.5}$$

This adjustment is made in structures similar to SSS structure. However, in folded-slots

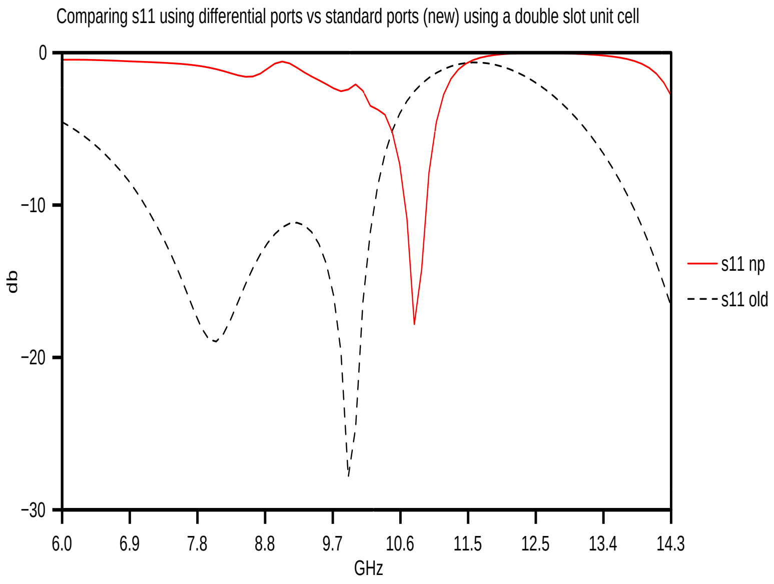


Figure 3.5: S_{11} of a unit SSS Cell using differential and normal ports

structures discussed in the next chapter, such modifications are not needed. Figure 3.5 shows S_{11} for a unit SSS cell using both differential port and normal port definition. Once the differential ports from MoM are adjusted by stub matching, we get less oscillations in the S_{11} plot.

3.1.5 Simulation of 3x3 SSS Array

A 3×3 SSS structure with dimensions shown below is simulated using *ArraySim*. Figure 3.6 shows the physical sense of the simulation setup. The array is centered between two hard horns, transmitting an receiving. A curve showing output radiation for a certain frequency range is plotted. This curve can be interpreted as S-parameters as it shows the frequency points at which the array resonates. The curves are formed by plotting the following equations.

$$S'_{11} = S^2_{11} + S^2_{12} + S^2_{13} + ... + S^2_{19} \tag{3.6}$$

Similarly,

$$S'_{99} = S^2_{91} + S^2_{92} + S^2_{93} + ... + S^2_{99} \tag{3.7}$$

The dimensions of this array are;

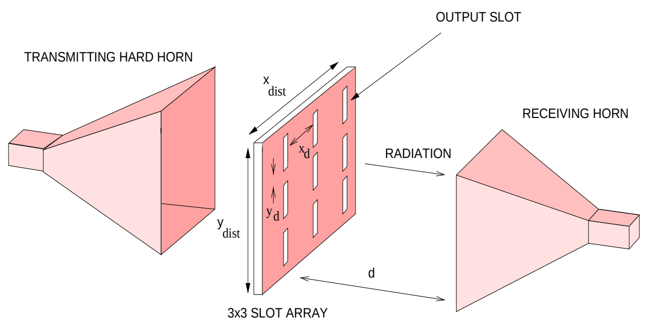


Figure 3.6: A 3×3 SSS array is excited with a hard horn and near field plots are made at various distances, d, from array

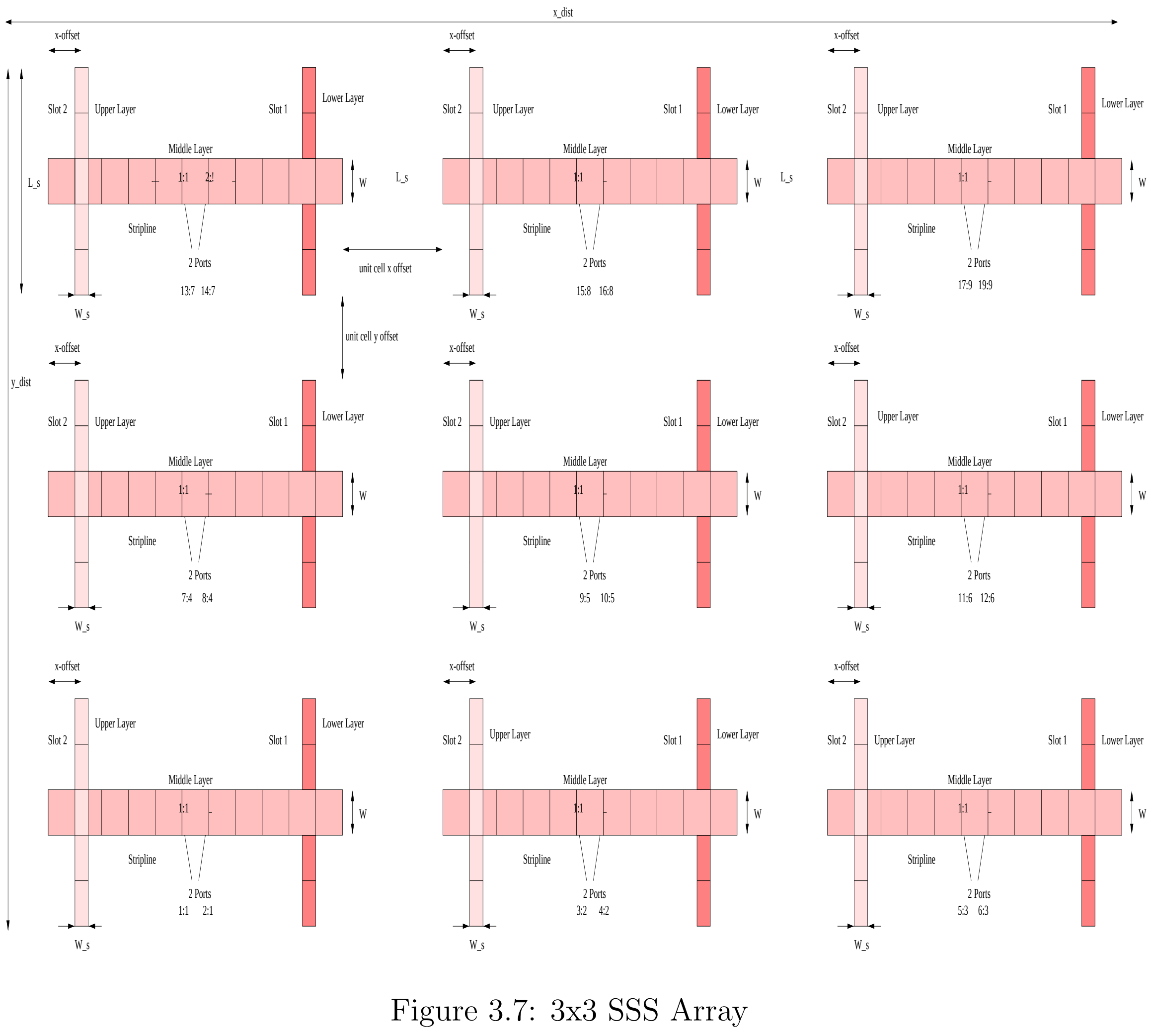


Figure 3.7: 3x3 SSS Array

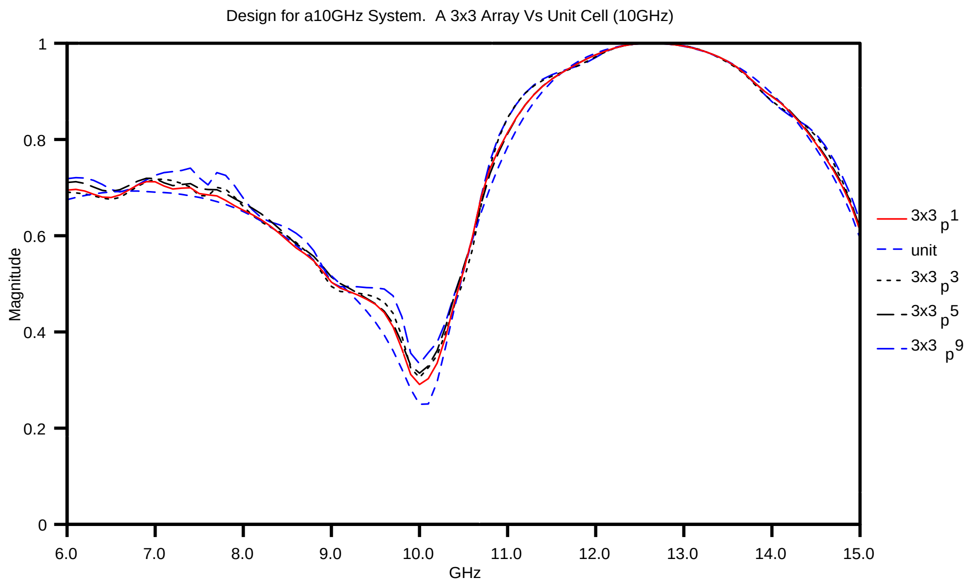


Figure 3.8: Simulation of a 3x3 SSS array shows minimum reflection of input power around 10 GHz

```
Stripline:
W = 4.5 mm
L = 38.75 mm
Slots:
W_s = 1.5 mm
L_s = 16 mm
x_offset = 7.5 mm
Dielectric:
Epsilon_r = 2.2
H1=H2=1.57 mm
x offset of unit cell = 20 mm
y offset of unit cell = 20 mm
x_dist = 156.25 mm
y_dist = 88 mm
```

As expected, the array behaves similar to a unit cell. The array shows minimum insertion loss around 10 GHz.

3.1.6 Field Plots

Near and far field plots were made for the 3×3 array at 10 GHz. Figure 3.9 shows near field plots at a distance of 0.8 mm, 10 mm, and 1 m. Note, at large distances, we see a beam as an output. Far field plot in y-direction is shown in Figure 3.10.

Near field plots in Figure 3.9 describe coupling effects in great detail. If all unit cells are isolated, then near field plots of lower slots (transmitting slots) show almost equal output field intensity as shown in Figure 3.9(a), (d) and (f) which is an ideal case. However, if coupling effects are taken into account, as shown in Figure 3.9(b), (c) and (e), then the left lower layer slots show more output field intensity than the remaining lower layer slots. This behavior is attributed to the cancelation of input E fields in the middle and the right lower

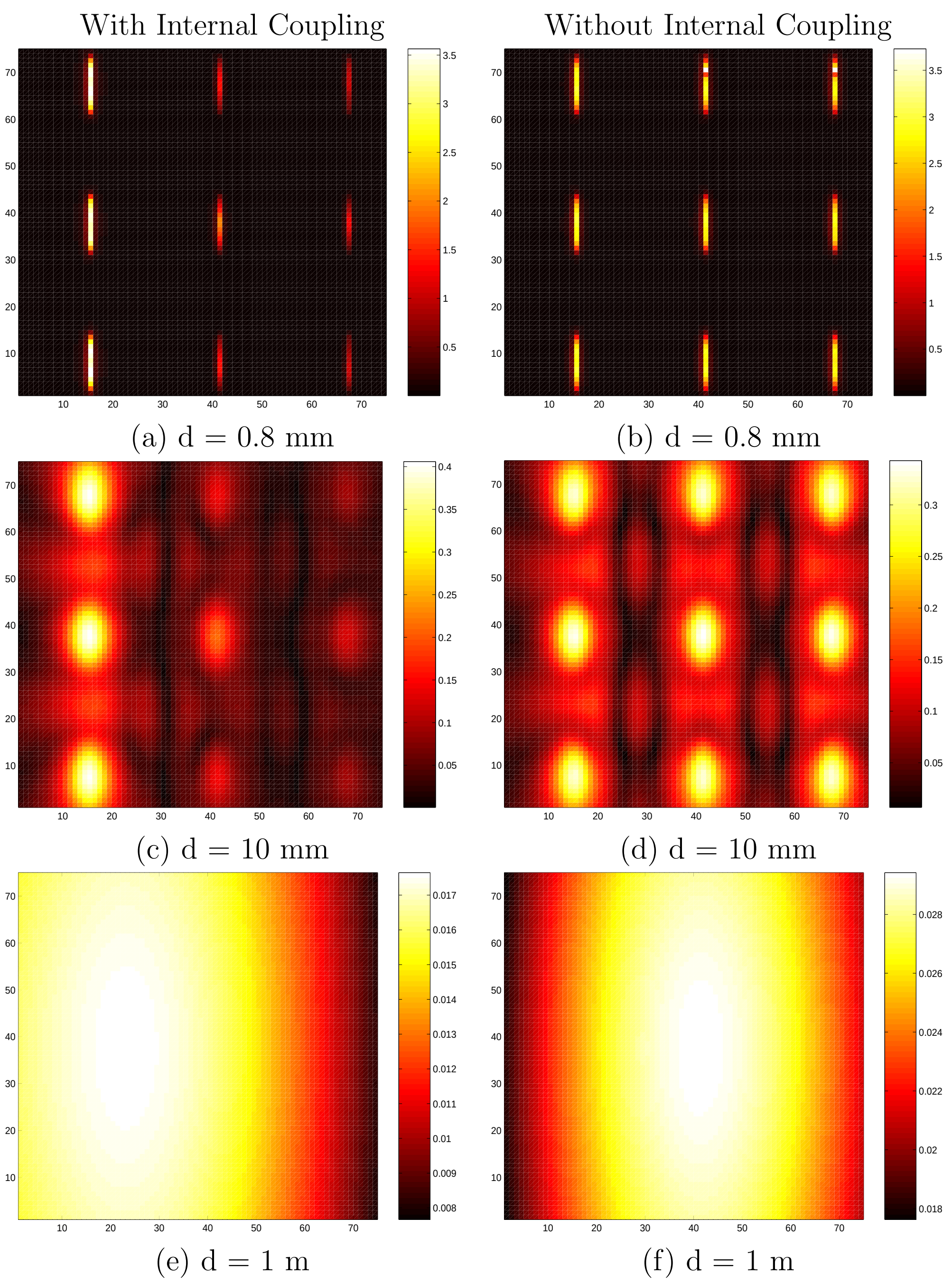


Figure 3.9: Near field plots for 3×3 array at 10GHz

layer slots. However, the left lower slots are more isolated than the middle and the right lower slots and thus show more output E field intensity.

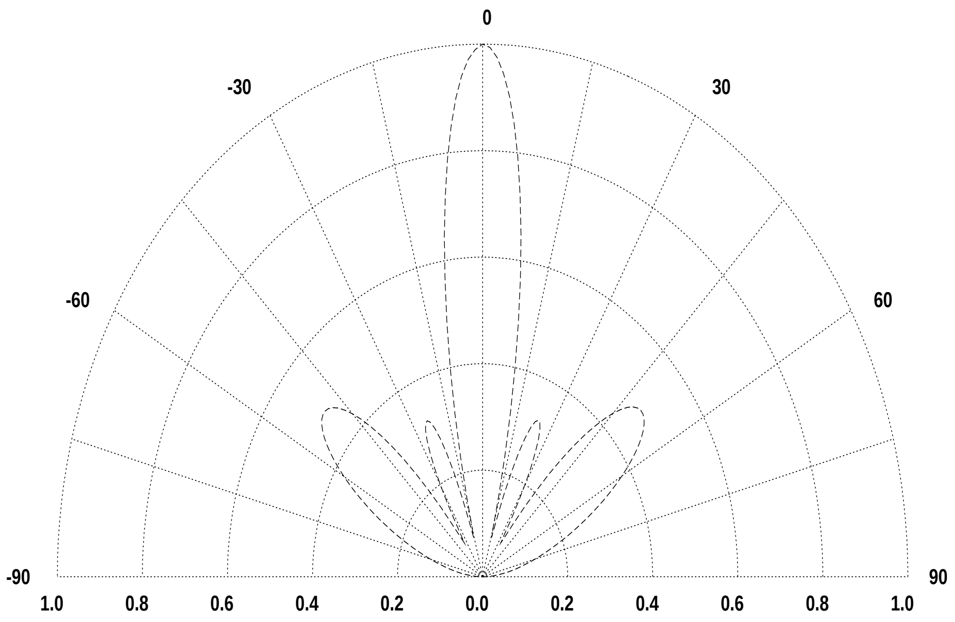


Figure 3.10: Far Field radiation pattern in the y-direction for 3×3 array at 10 GHz and a distance of 10 m away from array

3.2 Conclusion

In this chapter we take a sample three layer structure, SSS, and study its characteristics using *ArraySim*. Matrix configuration for the three and two layer structures of type SSS is discussed. If upper and lower slots shown in Figure 3.1 are made symmetric, *ArraySim* uses this symmetry to compute only 4 out of 9 *SubMatrix* objects. A two layer slot-stripline structure is simulated and compared with published results and a close agreement seen between results. A unit SSS structure is simulated and effects of stripline basis cell number on passive analysis is shown. We observe that as stripline cells in layout are increased, the plot in Figure 3.3 moves up and left. From simulating this unit cell we expect to see maximum power absorption (by plotting $S_{11}^2 + S_{12}^2$) around 10 GHz. This simulation is verified by observing a minimum insertion loss (of 2 dB) around 10.3 GHz on a spectrum analyzer for a fabricated SSS structure.

A 3×3 SSS array is also simulated and results plotted. We studied the coupling effects as observed from near field analysis of the 3×3 SSS array. Two cases are simulated and their near field plots are shown. In the first case, all unit cells (which are 9 for the 3×3 array) are simulated as if they are isolated from one another while in the second case, unit cells are allowed to couple with other unit cells within the array. A shift in the output beam is seen for the later case while a centered output beam is observed for the isolated case. The shift in the beam is due to cancelation of input E_y field that falls on the upper slots.

Coupling effects among unit cells may result in low efficiency of the system where they are used and so unit cells should be isolated by some technique during fabrication of such arrays to produce uniform output field patterns. Coupling effects may also reduce by rearranging unit cells in a fashion that gives minimum cancelation of input E_y thus enhancing overall system performance.

ArraySim can be further enhanced by interfacing more MoM functions describing other structures thus enabling to study more structures. Search tables are used to find symmetries and reduce computation time. In the next chapter we use *ArraySim* to simulate folded-slot structures given MoM functions describing single layer folded-slots.

Chapter 4

Investigating Folded-Slot Structures

In this chapter we show how *ArraySim* is used to study the behavior of folded slot structures that are based on a single layer of material. Various folded-slot antennas coupled with CPW lines are simulated. For simplification and speed purposes, CPW lines are neglected during passive analysis. The term folded-slot and CPW slot is interchangeably used in this thesis. Simulation results are then compared with published measurements. Coupling effects between array unit cells to overall performance is shown. Scaling of port input impedance in folded-slots is discussed. We will see from simulations that Z_{in} seen by the port is inversely promotional to the number of slots. This simple, but intuitive observation allows us to match the folded-slot antenna with transmission lines of various characteristic impedance. We see from simulation results that as unit cell distances within an array is increased, coupling effect decreases. So, at large distances of unit cells, arrays can be modeled as a unit cell which requires less simulation time. Effect of search table is also discussed by comparing simulation time with and without search tables.

4.1 Simulation of Folded Slot Antenna

Folded-slot antenna with coplanar waveguides, CPW, in an orthognal layout behaves as an effective receiver and transmitter antenna. The two orthognal folded-slots are connected to one another via a 90 degrees bent coplanar transmission line. Figure 4.1 shows a CPW unit cell which consists of two folded-slots on each CPW line (see Figure 4.23 for layout). Input impedance seen from the port into the slot depends upon the number of slots. We will see from a simple simulation that as the number of slots is increased, input impedance seen by the port reduces. So, an active device's input impedance can be matched with the folded-slot antenna by changing the number of slots.

Folded-slot unit cells are simulated using *ArraySim* and results compared with published measured results. A close agreement is seen in simulated and measured results. These results verify the correct working of *ArraySim* environment and so folded-slot arrays like 2×2, and 4×4 are simulated and their behavior investigated. Matrix configuration is simple for a single layer structure. In the next subsection, we describe the matrix configuration of the MoM based **G** matrix for a folded-slot antenna structure.

4.1.1 Matrix Configuration

Folded slot structures are single layer structures. So, the **G** matrix filled by *ArraySim* using MoM based functions, is a simple admittance matrix as described in [20]. The **G**

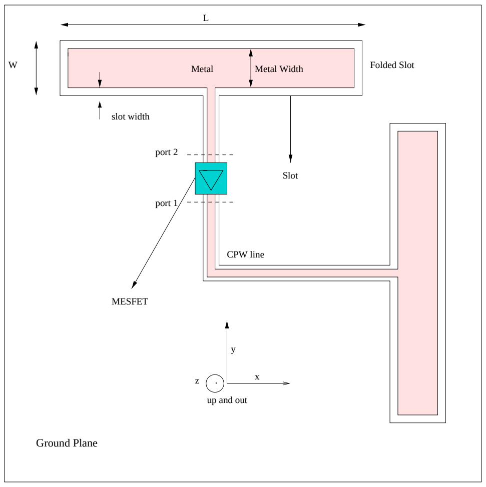


Figure 4.1: CPW based folded-slot unit cell

matrix (*MainMatrix* object) consists of a single *SubMatrix* object which in tern includes a *QuadMatrix* object as discussed in chapter 2. The matrix configuration of a this single layer folded-slot antenna is given in equation below.

$$\begin{bmatrix} \mathbf{H_x} \\ \mathbf{H_y} \end{bmatrix} = \begin{bmatrix} \mathbf{Y_{xx}} & \mathbf{Y_{xy}} \\ \mathbf{Y_{yx}} & \mathbf{Y_{yy}} \end{bmatrix} \times \begin{bmatrix} \mathbf{V_x} \\ \mathbf{V_y} \end{bmatrix} \tag{4.1}$$

where $[Y_{xx}]$, $[Y_{xy}]$, $[Y_y x]$ and $[Y_{yy}]$ are MoM admittance matrices. These four matrices are part of a *QuadMatrix* object as mentioned in chapter 2. $[H_x]$ and $[H_y]$ make up the excitation vector depending upon incident electric fields. For example an incident uniform field in the y-direction, E_y, only results in zero values for $[H_y]$ vector and finite values for $[H_x]$ vector. *ArraySim* uses the search engine described in chapter 2. to look for possible symmetries and reduce computation time. In the next subsection we discuss the effect of number of slots on input impedance of folded-slots.

4.1.2 Controlling Input Impedance

ArraySim allows to study the behavior of a given structure in detail. One of the characteristics of folded-slot antennas is their relation with input port impedance to number of slots. In this section we describe a rule of thumb that can be used to scale input port impedance with varying number of slots. Input impedance of a dipole in free space at first resonance is around 70Ω [17]. If this wire is folded, a folded dipole is created as two equal parallel currents with same direction flow on the wires. The input impedance of this folded dipole is on the order of 300Ω. The input impedance of folded dipole is around 4 times that of the input impedance of a single dipole. This results in a simple rule of thumb give by Equation 4.2;

$$Z_{in} = N^2 Z_{dipole} \tag{4.2}$$

where Z_{dipole} is the input impedance of a single dipole (approximately 70Ω).

A similar relation holds for slot antennas. As described in [17] and [18] impedance of a half-wave slot is given as;

$$Z_{slot} = \frac{(377)^2}{4Z_{dipole}} \tag{4.3}$$

So, the input impedance of N slots is given as;

$$Z_{slot} = \frac{Z_{slot}}{N^2} \tag{4.4}$$

Equation 4.4 shows that as the number of slots is increased, the input impedance lowers by a factor of the number of slots squared. So, adding additional slots allows the designer to vary the slot input impedance over a wide range. This simple design rule is useful for circuit designs that require impedance matching with slots.

ArraySim was used to simulate three structures with same dimensions and material but different number of slots. Figure 4.2 shows the three types of slot antenna structures. We plot input impedance seen at the ports and compare them with the rule of thumb described by Equation 4.4. Figure 4.3 shows simulated input impedances for single, double and quadruple slots. From simulations we see that input impedance does reduce with increasing number of slots by N^a where a is between 1 and 2. We also see a frequency shift as number of slots is increased. The dimensions for Figure 4.2 a, b and c were; L=18 mm, slot gap=1 mm, metal width=1mm, side width=1mm, ϵ_r=10.8, substrate thickness=0.635 mm. In Figure 4.2a, W=1mm, in Figure 4.2b W=7mm and in Figure 4.2c W=19mm. The simulated structures are simple folded-slots with no CPW line as shown in Figure 4.2a, b and c.

4.1.3 Unit Cell Behavior

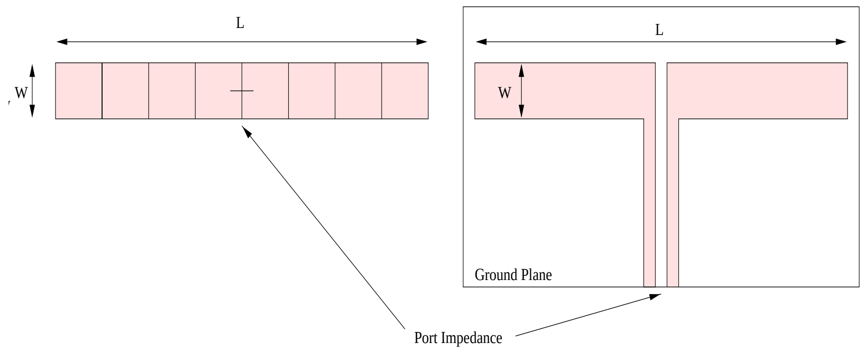
To study the behavior of CPW based folded-slot arrays of various sizes, it is meaning full to simulate a unit cell first as unit cells tend to give an insight into array characteristics. In this section simulation results from *ArraySim* for a folded slot antenna are compared with measured results. Figure 4.4 shows a folded two slot antenna layout. This simple folded slot structure is simulated with dimensions L= 78mm, W= 6mm, slot width= 2mm, metal width= 2mm, layout cell size= 2×2 mm. ϵ_r= 2.2 and substrate thickness= 0.813mm. Figure 4.5 shows simulated and measured S_{11} [18] at the port. A good agreement is seen between simulation and measured results [19]. These results indicate the correct working of *ArraySim* and show how it can be used to study basic structures as well as complicated arrays. Note: unit cell only consists of horizontal double slots, othognal slots are not part of the above simulation result.

4.1.4 Near and Far Field Analysis

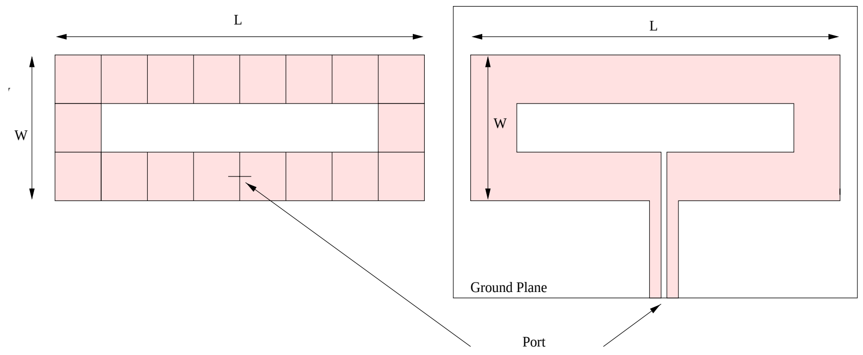
Near field analysis is done on folded-slot unit cell and arrays. These cells are excited with a uniform electric field, E_y, with a magnitude of 1 V/m. The horizontal slots are receivers while vertical slots are transmitters. Initially, during passive analysis we expect vertical slots to transmit lesser field than that transmitted/scattered by the horizontal slots. This is because the horizontal and vertical slots are not connected with one another.

So, incident field on horizontal slots are scattered more than being transmitted to vertical slots through mutual coupling. This means that vertical slots show weaker field intensity during passive analysis than field intensity on horizontal slots. However, during active analysis, vertical slots are excited with amplified currents and voltages that result in stronger E field.

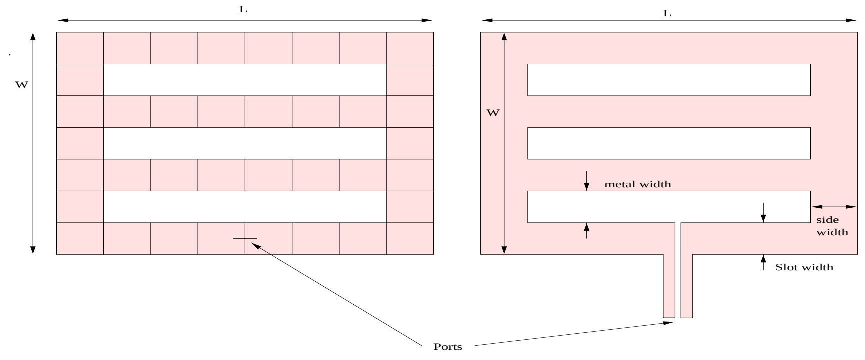
This implies that in active analysis (depending on the direction of active device), transmitting slots will show stronger radiation than receiving slots. This is the basics of power combining. Figure 4.6 shows an othognal folded-slot cell without a CPW transmission line. The dimensions for this structure are; L= 18 mm, W= 7 mm, metal and slot width= 1 mm,



(a)

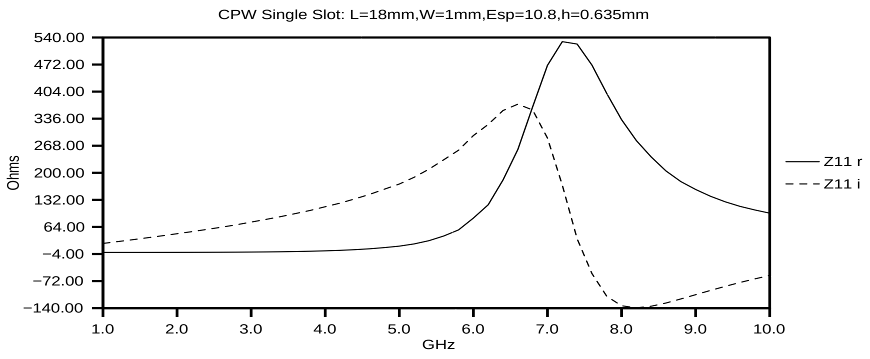


(b)

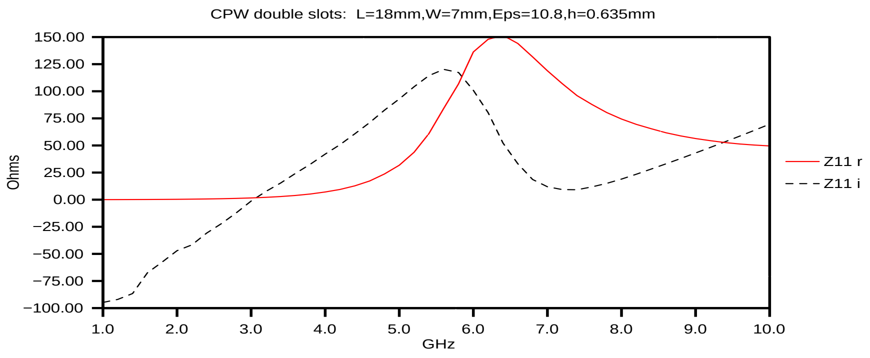


(c)

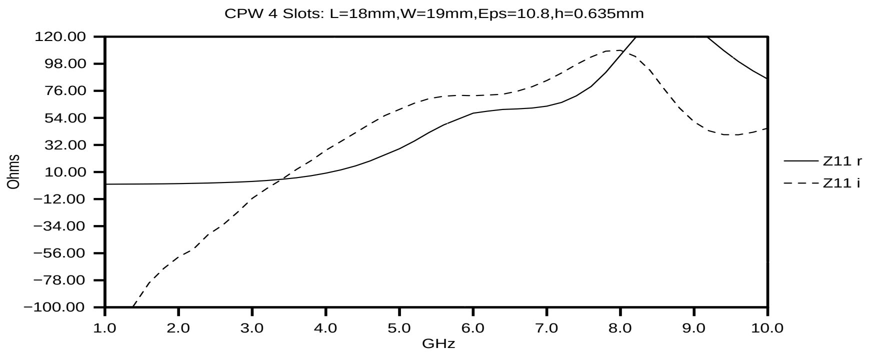
Figure 4.2: Layout (left) and real (right) structure (a) single slot (b) double slot (c) triple slot. Simulations are performed on the layouts using ArraySim (MoM) and Z_{in} of (a), (b) and (c) are compared



(a)



(b)



(c)

Figure 4.3: Z_{in} reduces with number of slots as seen in (a) single slot (b) double slot (c) triple slot

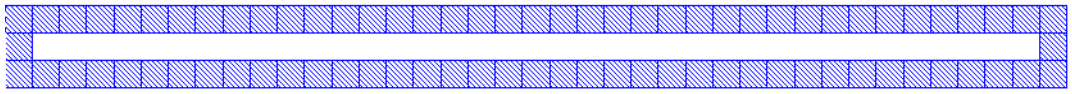


Figure 4.4: Layout of a 78 mm long double folded slot cell with 2×2 mm cell size drawn in Virtuoso. Port label is assigned at the center of the length on the lower slot.

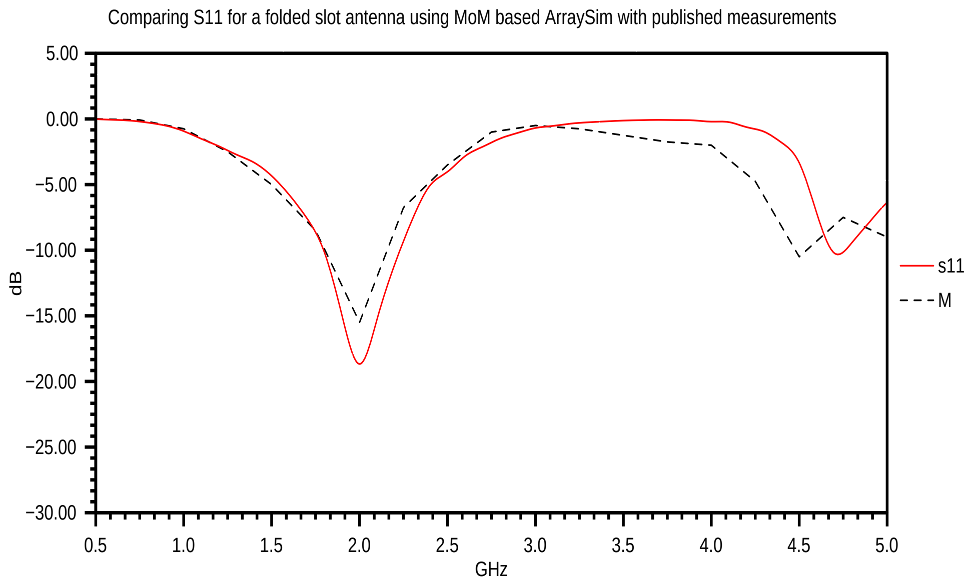


Figure 4.5: Comparison between measurement and MoM based ArraySim simulation of a folded-slot antenna

layout cell size = 1×1 mm ϵ_r= 10.8 and substrate thickness=0.635 mm. Near field analysis is performed at distance of 0.8 mm away from the slots as shown in Figure 4.8. As expected, for passive analysis, the horizontal slots show higher field intensity than vertical slots because their is no transmission line and active device involved to amplify transmitted fields. Figure 4.9 shows a far field plot at a distance of 10 m away for fields in the x direction.

4.1.5 Simulation of 2×2 Folded-Slot Array

A 2×2 CPW array is simulated using *ArraySim*. Network parameters are extracted and analyzed. Figure 4.10 shows a 2×2 array. In this figure, $D_x = D_y = 6$ mm. between unit cells. Other dimensions include, L=78mm, W=6mm, W_{slot}=2mm, ϵ_r=2.2 and h=0.813mm (thickness of substrate). Figure 4.11a,b,c and d show S and Z parameters for this 2×2 array. Figure 4.11a compares S_{11} for a horizontal folded-slot, shown in Figure 4.5, with S_{11} obtained for the 2×2 array and S_{11} for a unit othognal folded-slot.

Figure 4.11b shows S_{11}, S_{12}, S_{13}, S_{14} S_{15} S_{16}, $S_{17}$and S_{18} for the 2×2 array while Figures 4.12 and 4.13 show real and imaginary parts of input port impedances at ports 1,3,5 and 7. Coupling effects due to unit cells next to one another is responsible for changes in impedance and S-parameters. Note, for this simulation, unit cells were separated by 6 mm from one another. This distance is fairly small ,however, the 2×2 array behaves similar to the unit othognal cell.

Figure 4.14 shows near field plots in mesh view for a plane incident E field in the y-direction. Since, folded-slots are not connected, we expect horizontal slots in the x-direction to show higher field intensity than field intensities on vertical y-direction folded slots. Near field plots show results as expected. These plots are created at a distance of 0.8 mm from the array, in the z-direction.

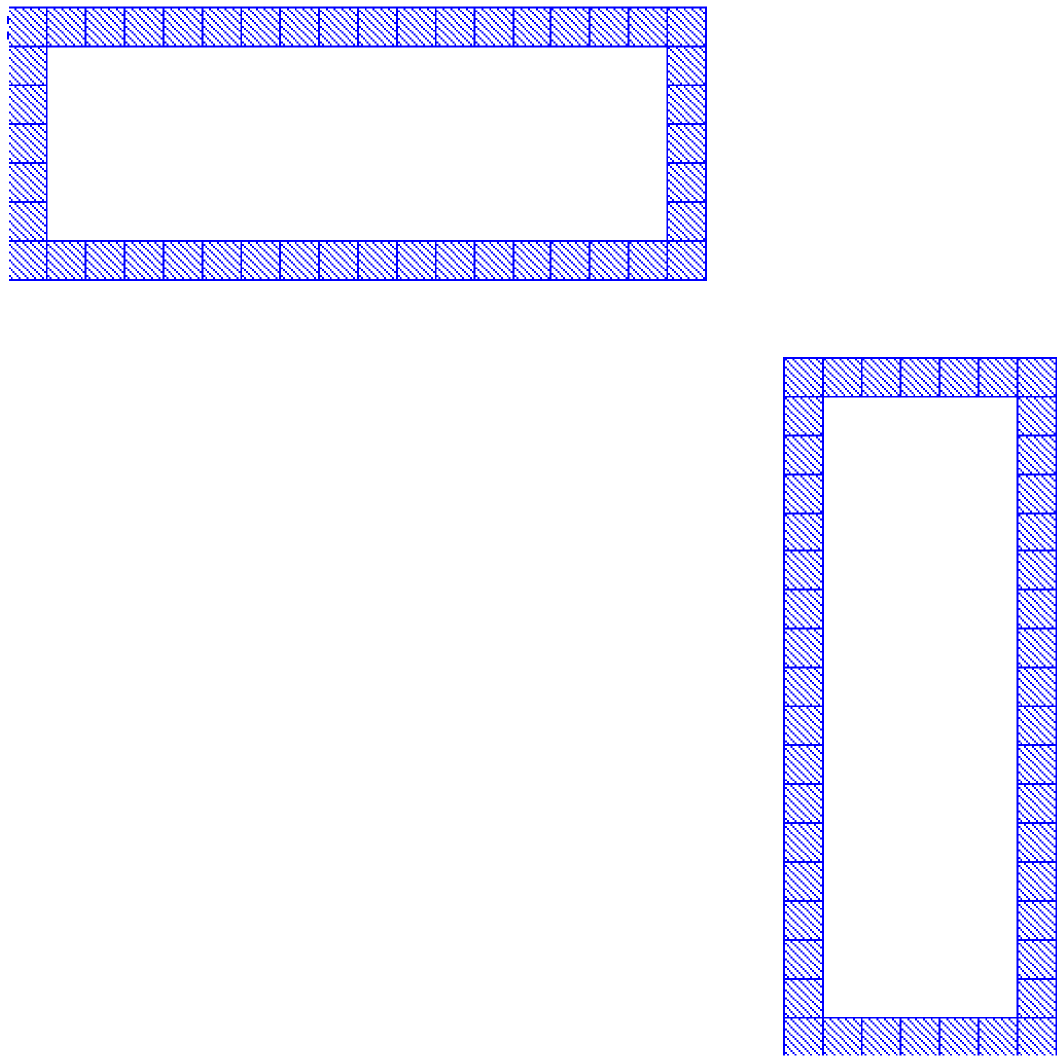
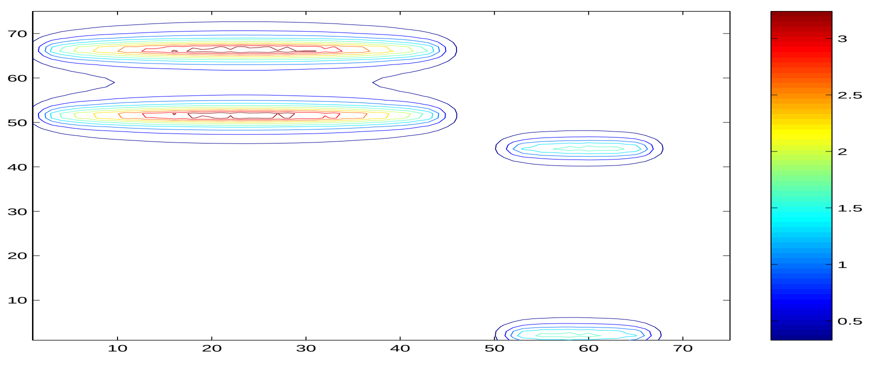
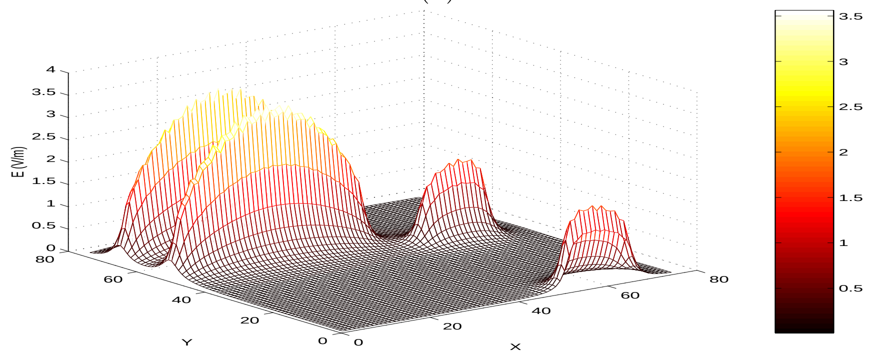


Figure 4.6: Layout of an 18 mm long double folded slot cell with 1×1 mm cell size drawn in Virtuoso. Near and far field analysis is done on this slot with an uniform incident field E_y= 1 V/m


(a)



(b)

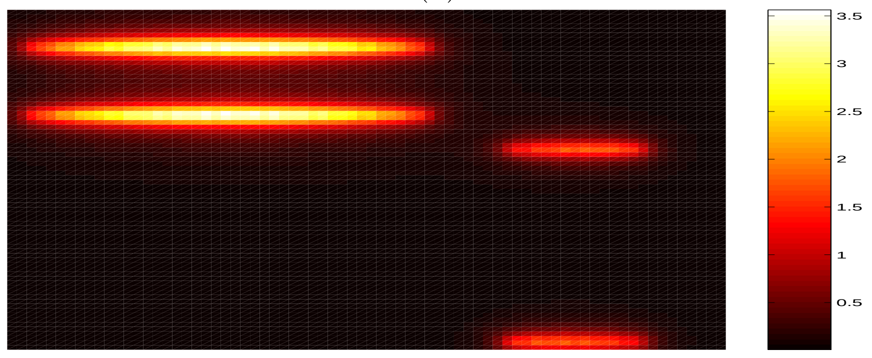
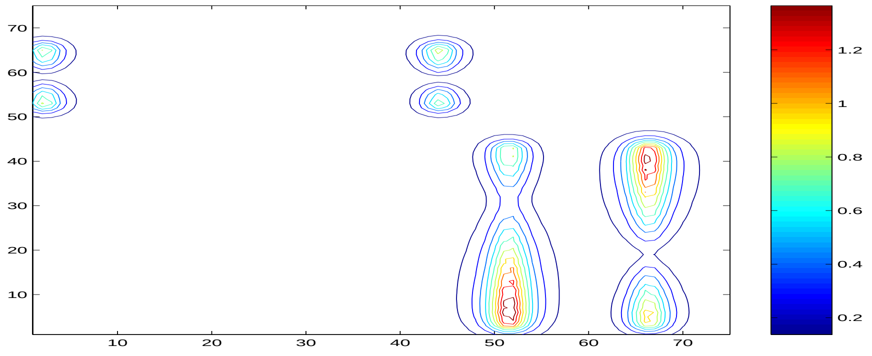
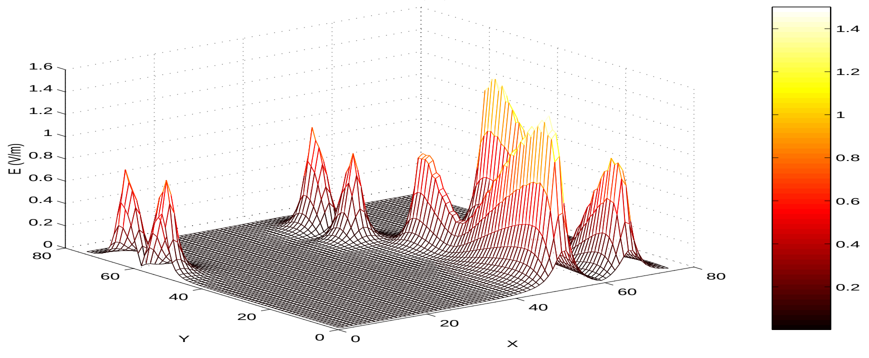

(c)

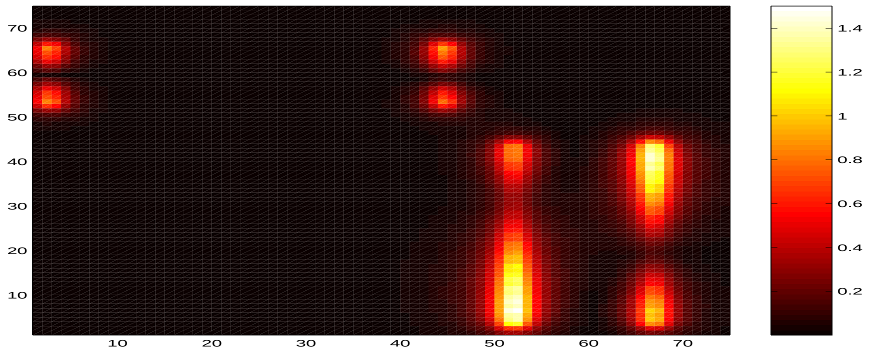
Figure 4.7: Near field plots at 0.8mm for a unit folded-slot cell with plane E_y of 1V/m. All plots show E_x on the horizontal transmitting slots



(a)

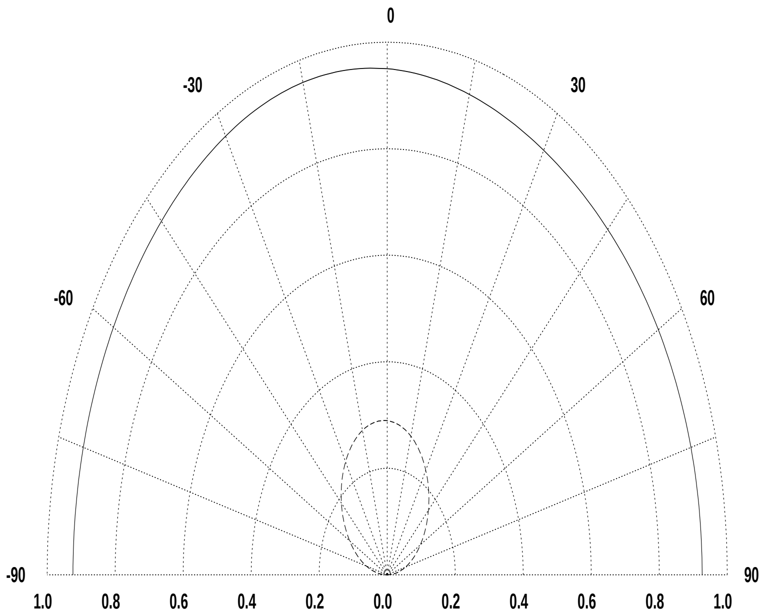


(b)

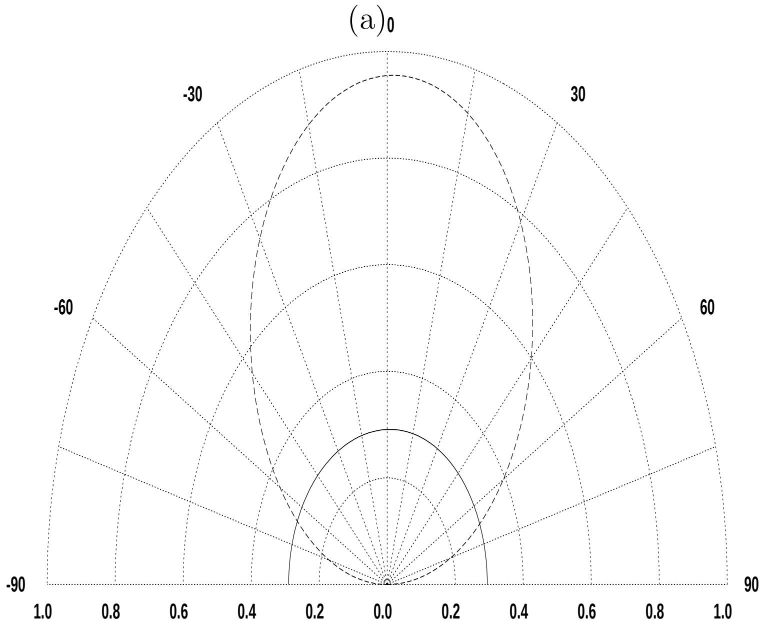


(c)

Figure 4.8: Passive near field plots at 0.8mm for a unit cpw cell with incident plane E_y of 1V/m. All plots show E_y on the vertical transmitting slots



(a)



(b)

Figure 4.9: Passive radiation pattern at 2 GHz at 10m distance for a unit folded-slot cell with incident E_y fields of 1V/m. (a) in y direction (b) in x direction

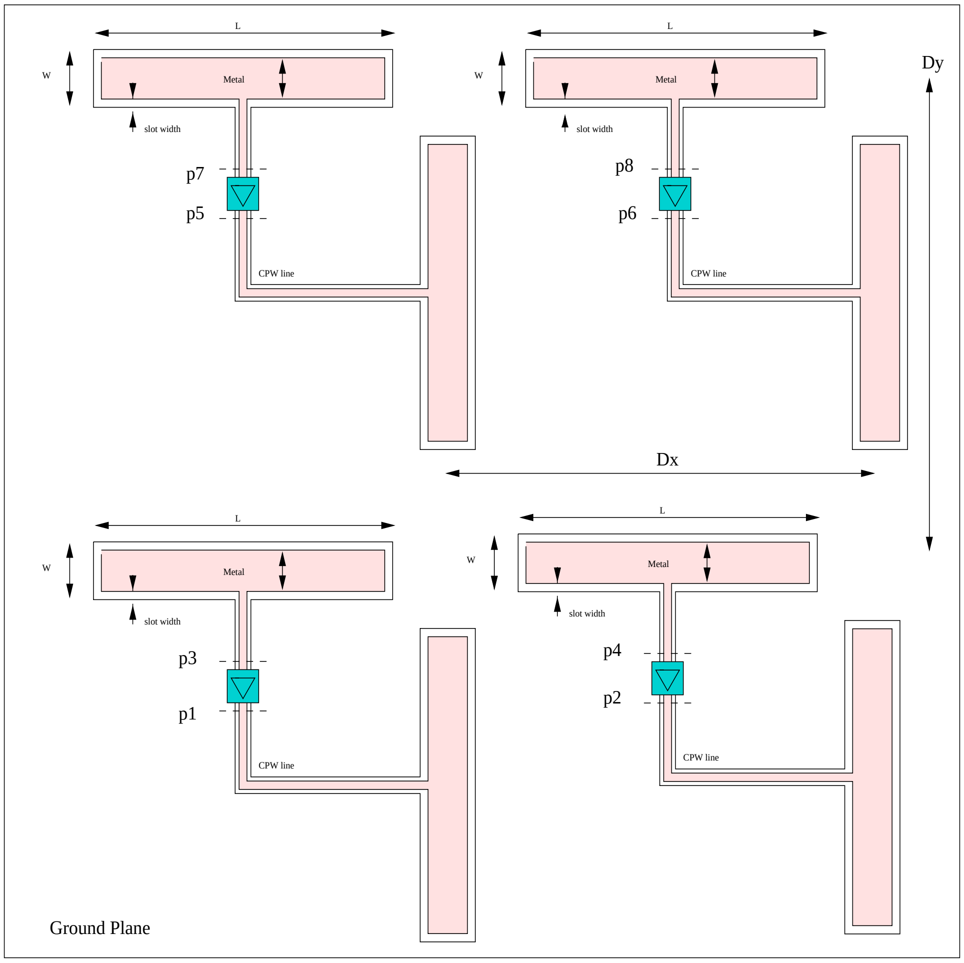
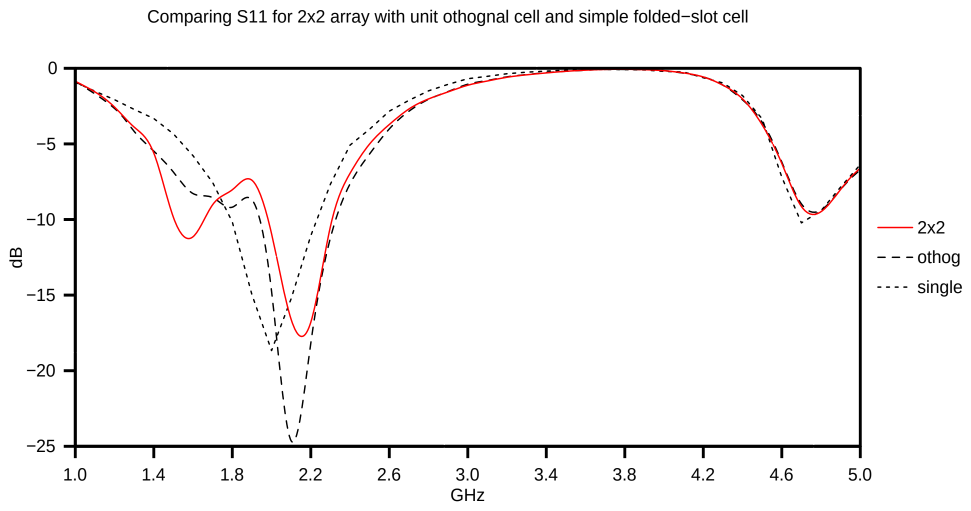
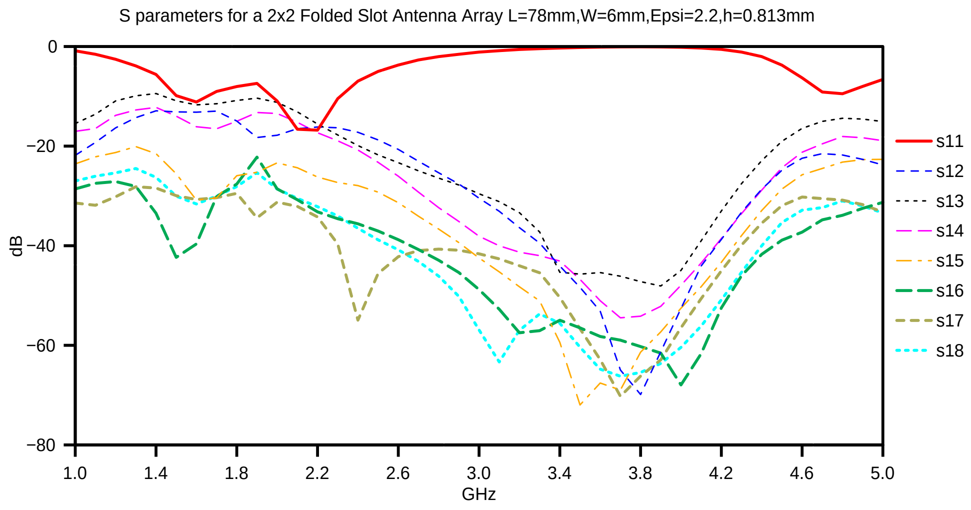


Figure 4.10: CPW 2x2 Array

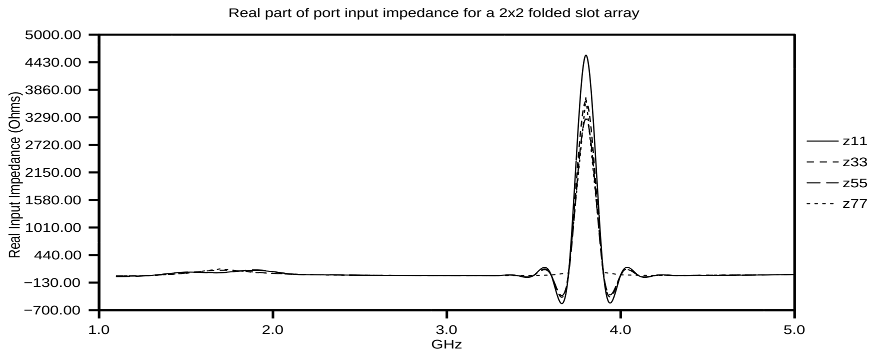


(a)

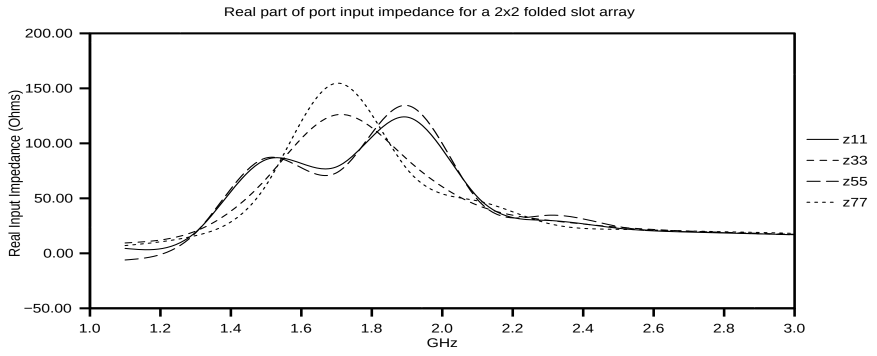


(b)

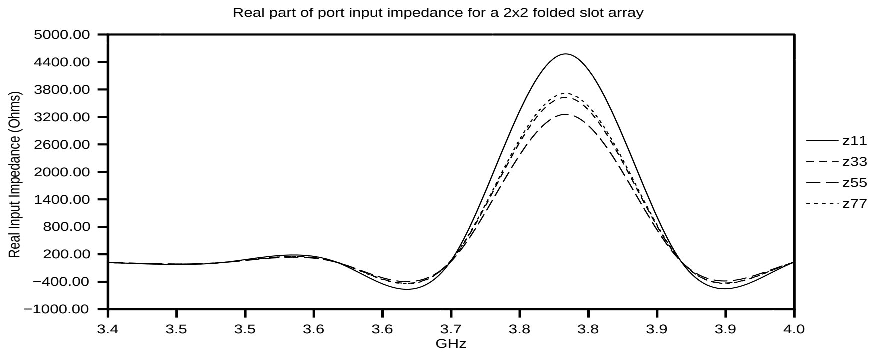
Figure 4.11: Passive simulation of folded-slot 2×2 array (unit cells separated by 6mm): (a) S_{11} of othognal (othog), horizontal double slots (single) and 2×2 array (b) S parameters of 2×2 array



(a)

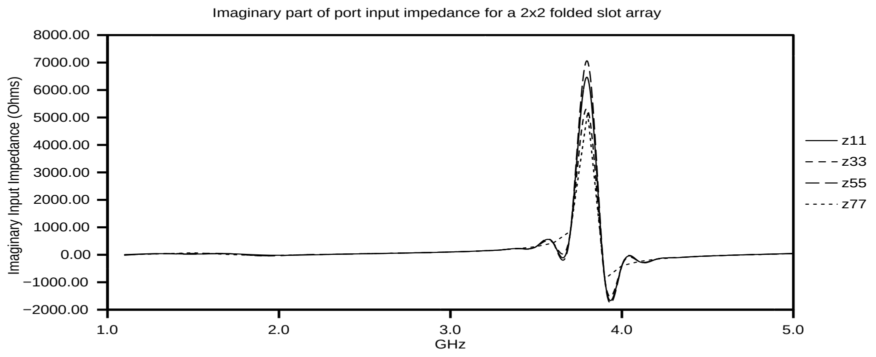


(b)

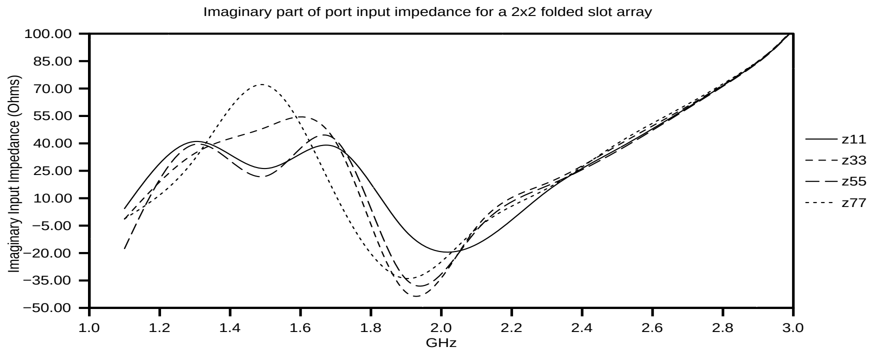


(c)

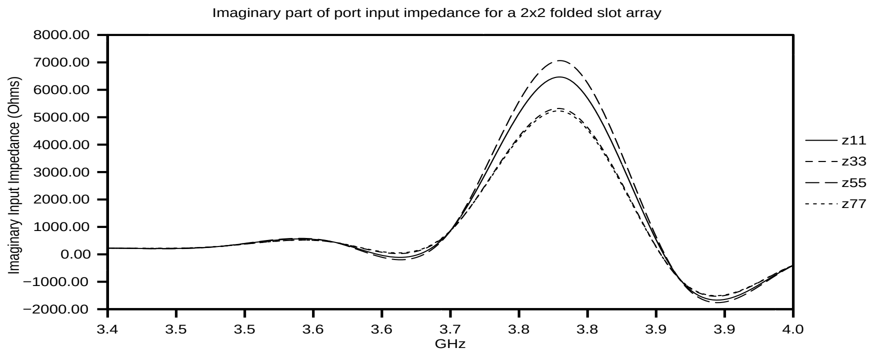
Figure 4.12: Folded-slot 2×2 array (unit cells separated by 6mm): (a) Real Z_{in} of 2×2 array (b) and (c) Zoomed real Z_{in} of 2×2 array



(a)

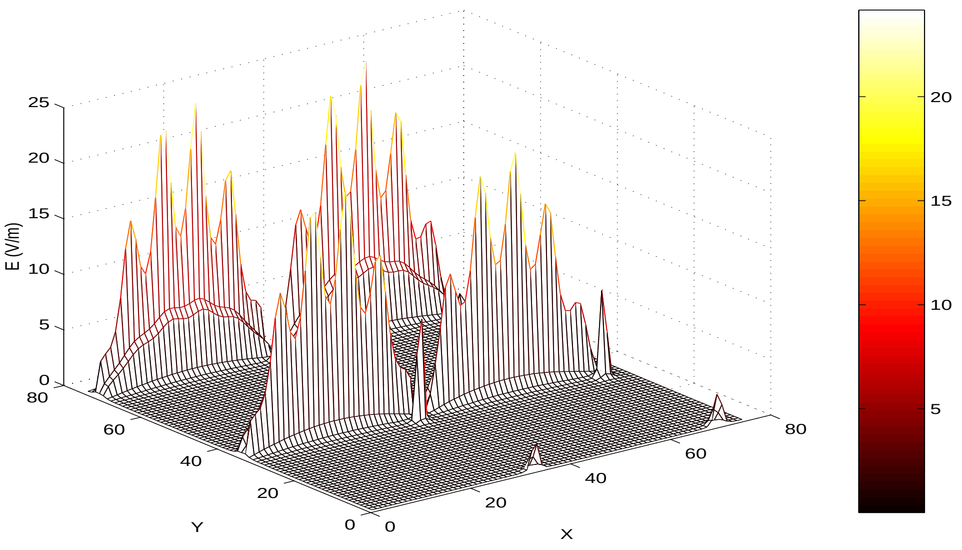


(b)

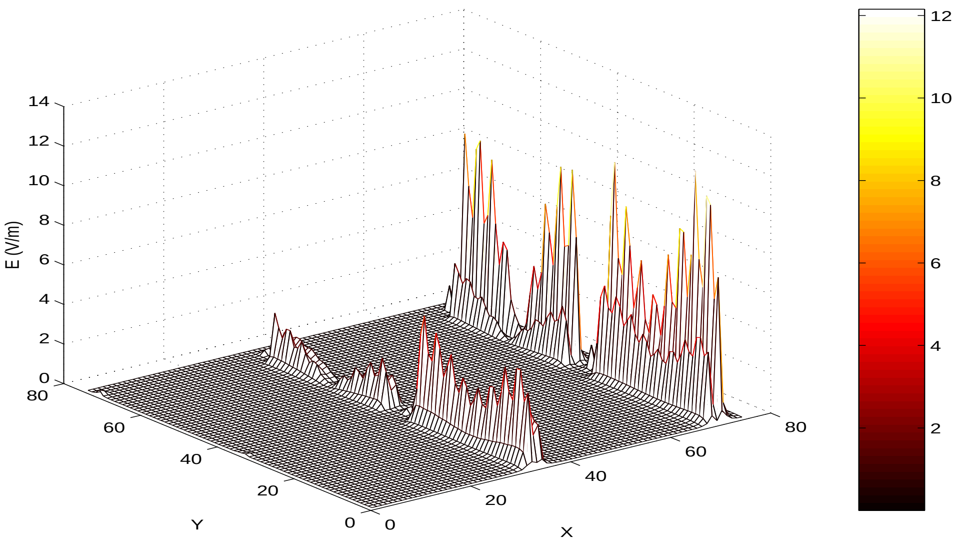


(c)

Figure 4.13: Folded-slot 2×2 array (unit cells separated by 6mm): (a) Imaginary Z_{in}(b) and (c) Zoomed imaginary Z_{in}



(a)



(b)

Figure 4.14: Passive near field plots of folded-slot 2×2 array at 0.8mm: (a) Mesh view of x direction slots (b) Mesh view of y direction slots

4.1.6 Simulation of 4×4 Folded-Slot Array

A 4×4 folded-slot array is simulated using *ArraySim*. Figure 4.15 shows a 4×4 array of othognal folded-slots. However, the simulated layout of the folded-slot array does not include CPW transmission lines. Neglecting, CPW lines is basically done to ignore transmission line coupling effects. Input impedance parameters for this array are plotted in Figures 4.16a and b with real and imaginary parts compared with unit cell Z_{11} to analyze coupling effects.

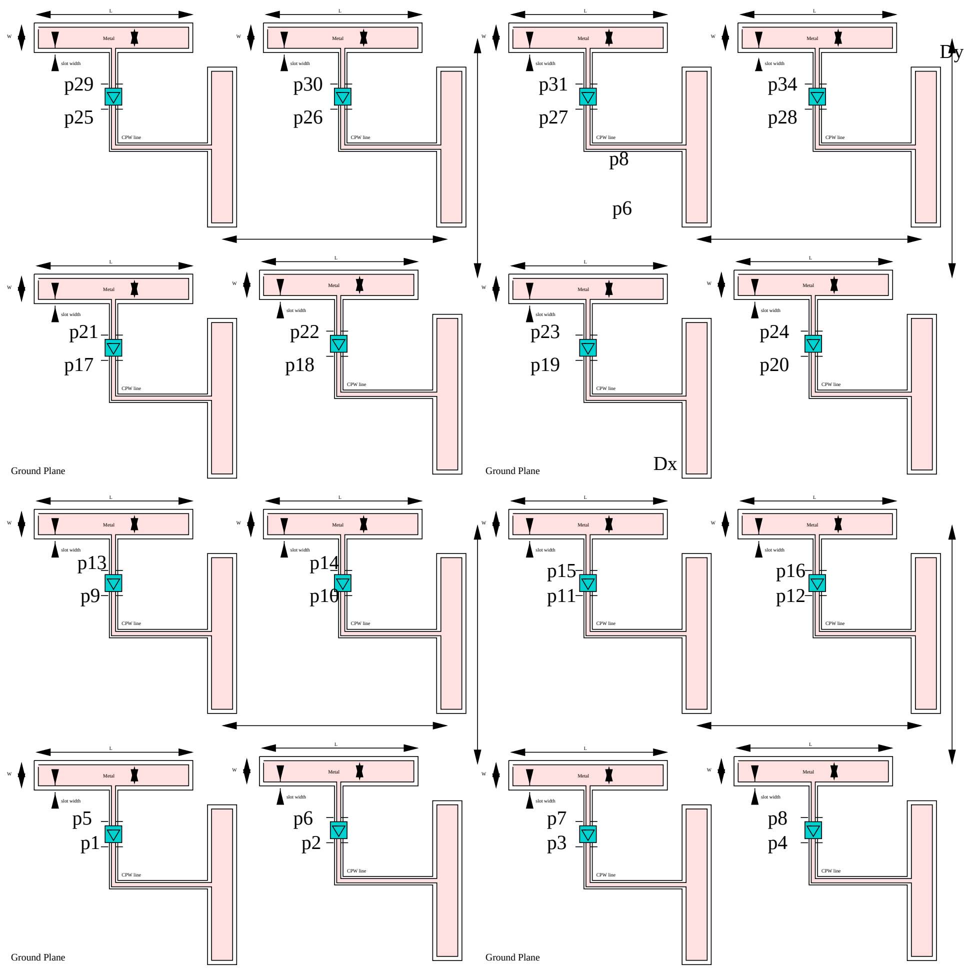
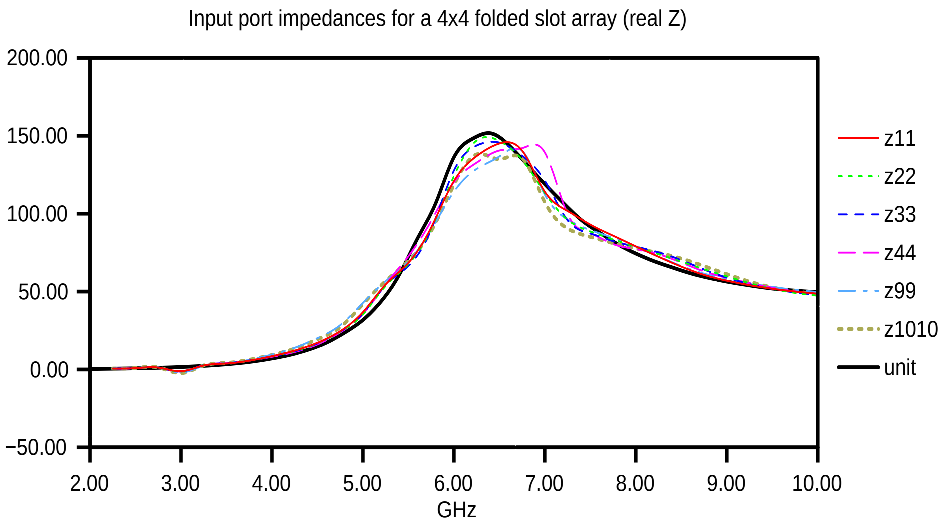


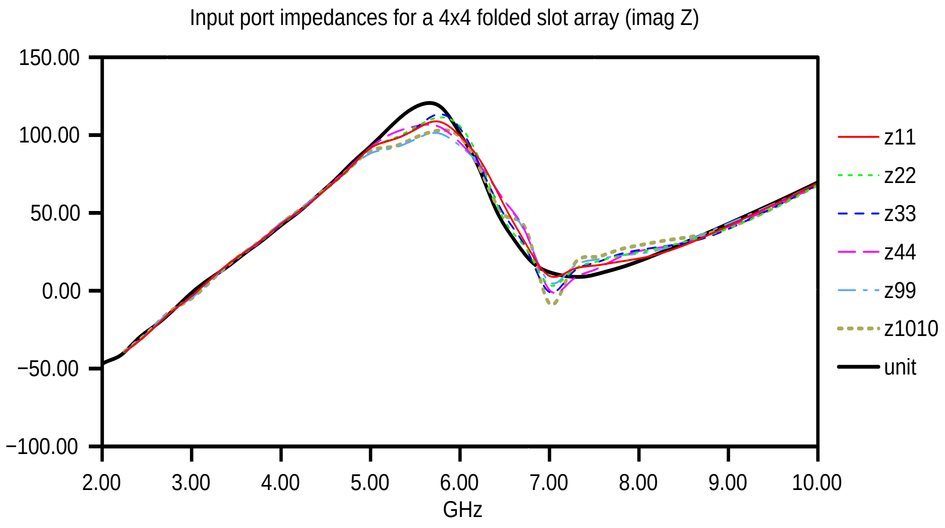
Figure 4.15: CPW 4x4 Array

4.1.7 Designing Folded-Slot Arrays

As observed in chapter 3, coupling effects among unit cells in an array may degrade the performance of the overall system. In this section we look at three configurations of 2×2 folded-slot arrays as shown in Figure 4.17 and plot their near field plots at 0.8 mm, 10 mm and 1 m away from the array to show which one of the three configuration may be a designer's choice. For all designs the length of the slots is 18 mm while the width is 7 mm (same dimensions as described for the 4×4 array). From near field plots shown in Figure 4.18 we observe that design 3 is the most symmetric design and probabally is a better choice than design 2 and design 1 because of its symmetric coupling cancelation around 10 mm distance. Since quasi-optical systems tend to be smaller in size at higher frequencies, near field distances like 10 mm make more sense than 1 m near field distances in order to

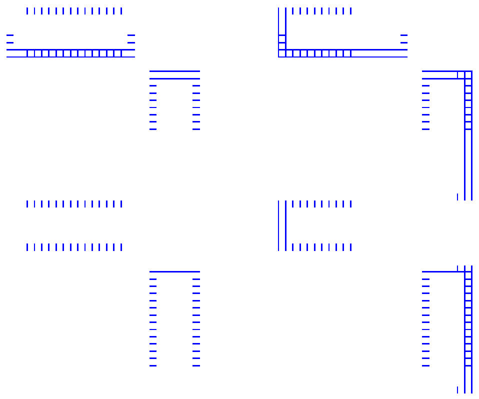


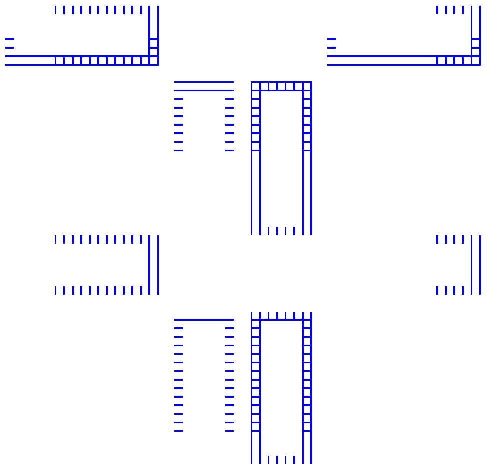
(a)



(b)

Figure 4.16: Z_{in} of folded-slot array compared with unit double folded-slot cell (a) real (b) imaginary


(a) Design 1: Array made from replicating unit cells


(b) Design 2: Array with its 2nd column a mirror image of its 1st column

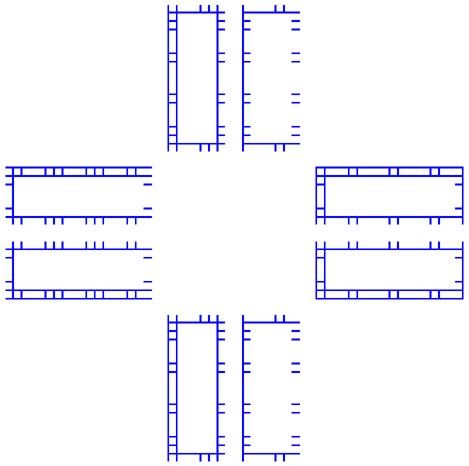

(c) Design 3: Symmetric array

Figure 4.17: Three designs for 2×2 folded-slot arrays with vertical slots as transmitting slots and horizontal slots as receiving slots

understand the system behavior. Once again we have shown the versatility of *ArraySim* is designing array structures.

4.1.8 Active Analysis and EIPG

Active analysis involves interfacing *ArraySim* passive analysis with a circuit simulator. As discussed in chapter 2. , an interface with a circuit simulator, Transim, is developed. In order to perform active analysis for folded-slot based CPW structure, we first perform passive analysis and find the frequency for a certain input impedance. A CPW transmission line is then designed for simulated input impedance. Figure 4.19 shows a CPW line whose characteristic impedance depends upon W and S, where W is the width of the CPW slot and S is the width of the CPW line (metal). Z_0 of the CPW line is calculated using the equations from [22] which are;

$$k_1 = \frac{S}{S+2W} \tag{4.5}$$

$$\epsilon_{re} = \frac{\epsilon_r + 1}{2} \tag{4.6}$$

$$Z_0 = \frac{30\pi}{\sqrt{(\epsilon_r)}} \frac{K'(k_1)}{K(k_1)} \tag{4.7}$$

where $\frac{K'(k_1)}{K(k_1)}$ depends on the value of k_1 (see [22] for $\frac{K'(k_1)}{K(k_1)}$ expressions).

After designing the CPW line for a certain impedance, EIPG (Isotropic Power Gain) is computed, for an operating frequency range, with no active device. The EIPG just computed indirectly gives the gain of the passive antenna system. For the folded-slot based CPW unit cell discussed in [18] ,and earlier in the chapter, with dimensions; L=18 mm and W=7 mm, the CPW line is designed for Z_0=80 Ω. For that purpose Equation 4.5 is recomputed to Equation 4.8;

$$S = 0.3694011276W \tag{4.8}$$

It is not possible to layout a folded-slot CPW unit cell with cell size of 0.33×0.33 because of large number of cells needed to draw the layout. Greater number of cells implies higher simulation time and possibly memory problems. So, as a compromise, we use layout cell size of 0.5×0.5 which gives us Z_0 (CPW) of 82 Ω. Figure 4.21 shows the EIPG in dB with the vertical slots as the transmitting slots while Figure 4.20 shows the actual layout drawn in Virtuoso. The width of the CPW line i.e. S= 1 mm while the width of the CPW slot is 3 mm.

An active device (like MESFET or BJT) is then connected on the transmission line and currents and voltages for this amplifier are computed using Transim. A simple small signal BJT/MESFET model is used for AC analysis with Z_{in}=Z_{out}=125 Ω and g_m=33 mS. Output currents and voltages are then calculated. These currents or voltages are used to form a new excitation vector and overall gain (EIPG) is re-computed. Similar procedure is used for array active analysis. Figure 4.24 shows a connection of a MESFET with a two port based admittance matrix. Care should be taken in signs of currents or voltages that are added to form the new excitation vector. Figure 4.22 shows the EIPG for when an active device is connected (see Appendix A for EIPG). In this particular simulation, a simple transmission line is used to model the CPW line. However, using the CPW line modeled earlier in this subsection and shown in Figure 4.23, we expect to see the simulated EIPG to be closer to the published measurement (measurement data is obtained from [18]).

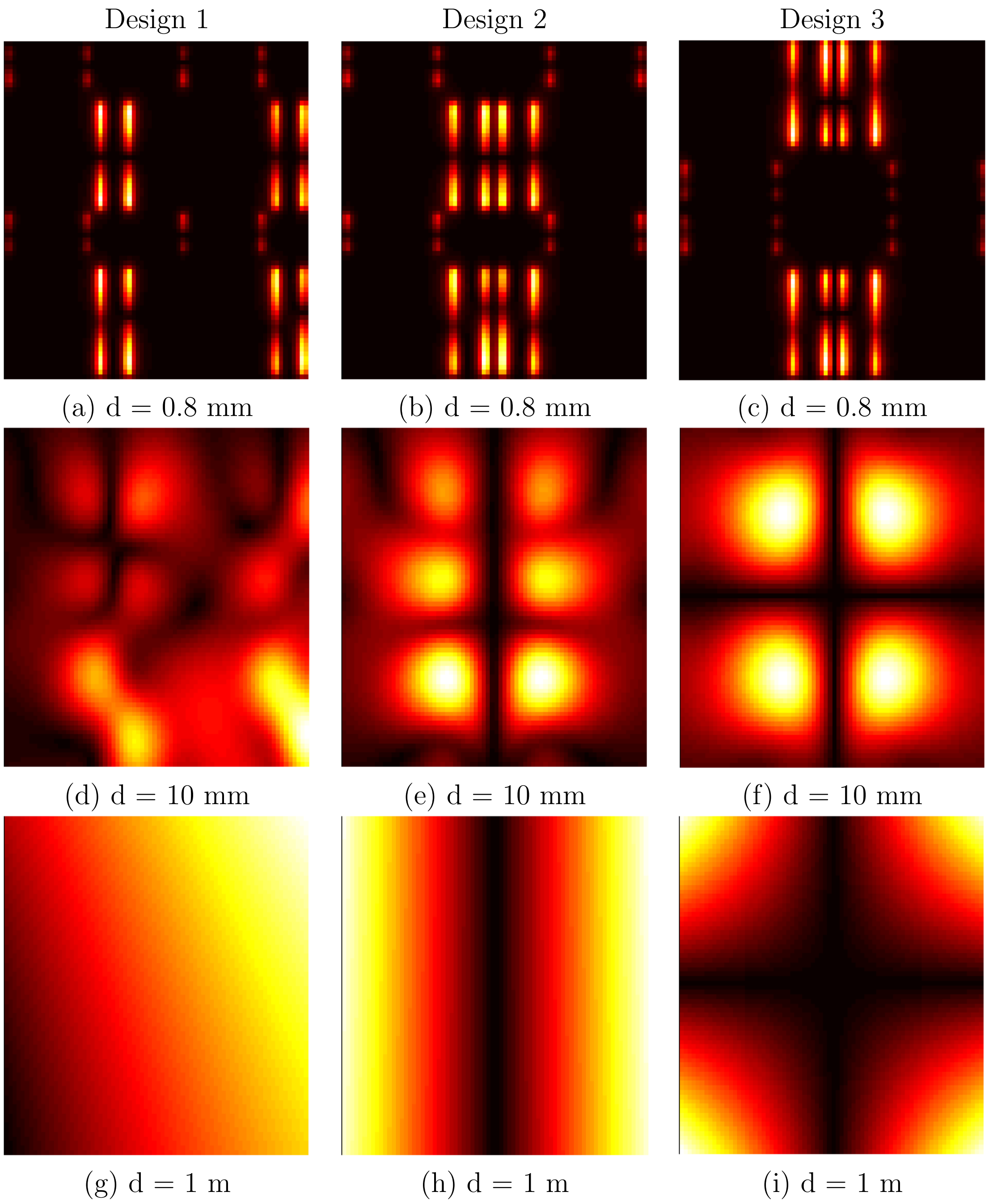


Figure 4.18: Near field plots for 2×2 array at 9GHz

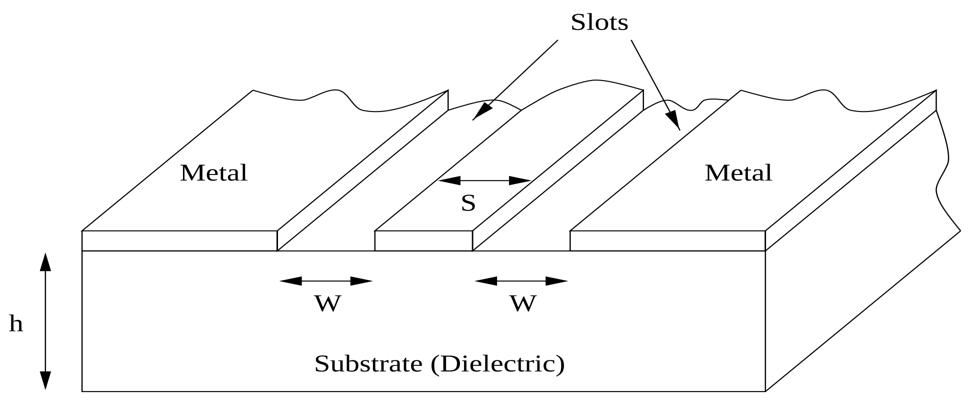


Figure 4.19: Coplanar waveguide (CPW) geometry

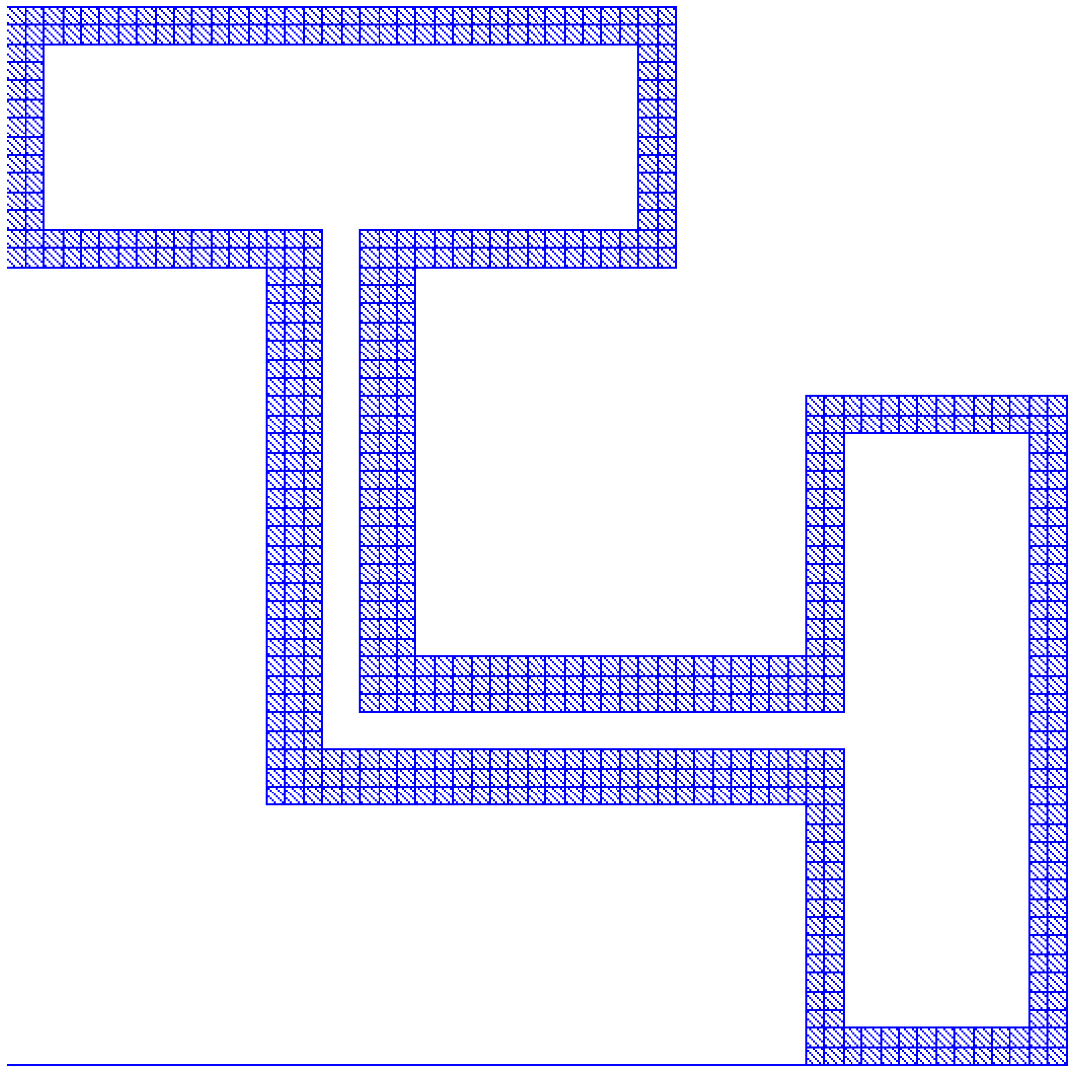


Figure 4.20: Layout for an othognal folded-slot based CPW cell which is used to compute the EIPG without active device

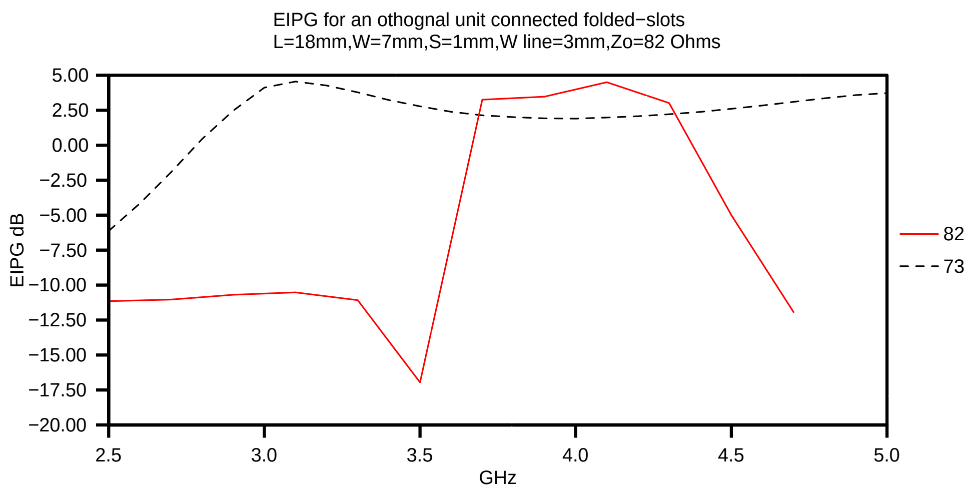


Figure 4.21: EIPG for an othognal CPW folded-slot unit cell with a CPW designed for 82 Ω (solid). Frequency range between 3.6 GHz and 4.4 GHz is the operating range of the structure. A 73 Ω CPW line (dotted) is compared which shows a slightly different behavior due to mismatching of CPW line impedance, Z_0 and folded-slot Z_{in}.

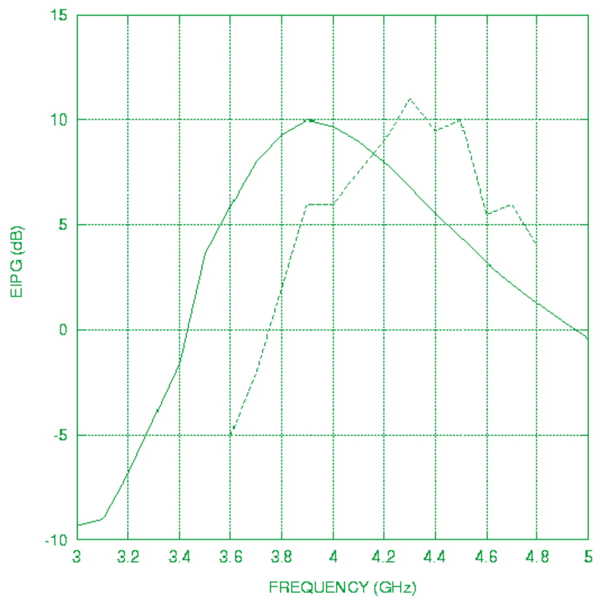


Figure 4.22: EIPG for an active othognal folded-slot unit cell with a simple transmission line modeled as a CPW line.

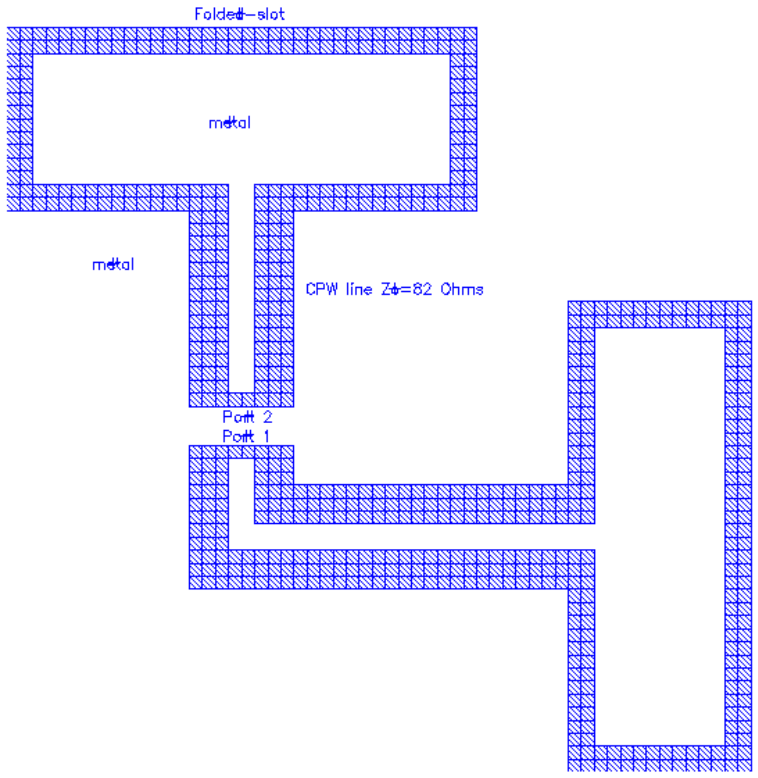


Figure 4.23: Layout of folded-slot unit cell with 82 Ohm CPW line used for active analysis

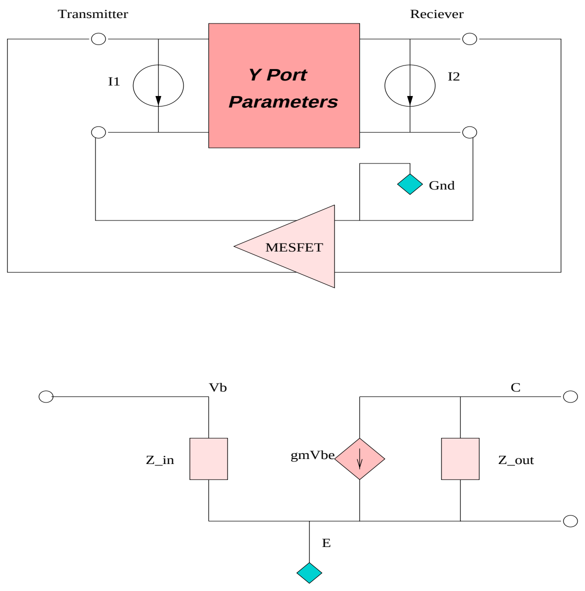


Figure 4.24: Circuit Analysis of an amplifier given a Y port based matrix

4.1.9 Search Engine Efficiency

ArraySim uses symmetries in the drawn layout to reduce computation time for calculating and filling MoM based **G** matrix. Folded-slot (with and without CPW lines) are single layer structures and size of **G** matrix (depending on number of cells in layout) is generally smaller in size than the size of **G** matrix for three layer structures like SSS. However, the search engine increases efficiency by more than 90%. Table 4.1.9 shows simulation time of relatively small unit cells with and without search table. Table 4.1.9 lists the number of elements that were computed by solving MoM integral equations when search engine option was turned off and on.   where ST stands for search table and Śim. Time‘ includes time for MoM

Table 4.1: Comparing simulation times with and without search engine for small single layer structures with varying number of cells in layout.

Num. of Layers	Figure	Num. of Cells	Sim. Time ST OFF secs	Sim. Time ST ON secs	Time Improvement %
1	4.1	17424	12539	388	96.9
1	4.3*a*	289	606	33	94.5
1	4.3*b*	2116	2639	63	97.6

Table 4.2: Comparing number of elements computed by solving MoM integral equations with and without search engine.

Num. of Layers	Figure	Num. of Cells	Elements Computed ST OFF	Elements Computed ST ON
1	4.1	17424	13068	1094
1	4.3*a*	289	289	17
1	4.3*b*	2116	1705	148

based **G** matrix filling, solving of matrix, reducing matrix to port based matrix, calculation of network parameters, and far field analysis (near field analysis was turned off). Clearly, having a search engine in *ArraySim* enhances performance. All simulations were made on a Sun Ultra 1 workstation.

4.2 Discussion

In this chapter we depict *ArraySim* as a CAE tool to study behaviors of single layer structures. Folded-slot antenna structure is used as a sample single layer structure and simulations are performed using *ArraySim* to study basic characteristics of folded-slots. Simulation results are compared with published measurements that show close agreement. This very notion verifies *ArraySim* and shows how this tool can be used to study large structures like those used in quasi-optical systems. Coupling effects among folded-slot unit cells is studied using near field plots. Generally, replicating unit cells one after the other to form arrays may not give an optimum performance. So, rearranging unit cells to get more symmetric arrays may give a better performance as seen from near field plots in Figure 4.18 . At the end we tabulate effect of search engine on simulation time and prove the efficiency of *ArraySim*. In the next chapter we conclude this thesis and list possible future research that may enhance performance and useablity of *ArraySim*.

Chapter 5

Conclusions and Future Research

5.1 Conclusions

A hierarchical modeling approach was presented for simulating electrically large structures used in spatial power combiners. A computer aided engineering tool, *ArraySim*, is developed to study behavior of various structures as long as their appropriate MoM functions are interfaced. This CAE tool allows users to draw their structure on a commercial graphical layout editor that produces a layout CIF description.

This CIF file is parsed to extract circuit port information and rectangular based basis cells. These basis cells are reacted with one another to form an impedance or admittance based MoM **G** matrix. Smart search routines are used to constantly check for symmetries in the drawn structure thus reducing computation time. The MoM **G** matrix is then solved and reduced to get port based impedance/admittance matrix. An interface with a circuit simulator is defined that uses these port based admittance matrices along with excitation currents/voltages at the ports to compute currents and voltages of an active device. These new currents and voltages are used to form an updated excitation vector to find near and far fields due to inclusion of active device.

Sample simulations are done on a three layer SSS structure, two layer slot-stripline structure and a single layer folded-slot structure. Unit cells as well as arrays are simulated and discussed. Simulation results are compared with published measurements for specific dimensions. A close agreement is seen in simulated and measured results that verify the correct working of *ArraySim*.

ArraySim allows the user to model structures like antennas and grids provided their correct MoM routines are used. From near field analysis of various array configurations of single layer folded-slot structures and triple layer SSS structure, we observe coupling effects at various distances from the arrays (on the output side). From these coupling effects we conclude that arrays should be designed carefully so as to reduce coupling effects among unit cells. In the next section we discuss various options that can be included in *ArraySim* to make it more efficient.

5.2 Future Research

Simulating components of quasi-optical systems has become unavoidable. As this new research area is being explored, new and better techniques of simulating array structures are evolving. Basic problems to simulating these electrically large structures include; formulation of MoM greens functions, computation time, matrix solution and memory. These problems are listed below with suggestions for improvement;

- *ArraySim* reduces computation time by using smart search routines that look for symmetries in the given structure and thus avoids re-computing symmetric reactions. Currently, sequential search routines are being used for searching purposes. These routines can be replaced by more efficient binary search routines to further speed up simulation time by several folds.

- Simulating large structures like a 10$\times$10 CPW array, fixed size cells to mesh the layout generates an extremely large MoM **G** matrix and thus increasing simulating time by thousands of folds. A practical approach to this problem is to use adaptive cell size. For e.g. the CPW line can be made from a few long rectangular cells rather than small cells used to layout the folded-slots.

- The size of MoM **G** matrix is directly proportional to number of cells used in the layout as well as number of layers. For large arrays, like, 5$\times$5 arrays, matrix size could be easily as large as 6 million elements. It is not efficient to store such matrices in data structures at all times. Each sub-matrix should be computed and stored in a file and should only be read into data structures just before solving.

- Presently, matrix solving, that includes inversion and multiplication, takes up a chunk of total simulation time. This is because we use LU decomposition based matrix inversion. Since a new matrix is computed for every frequency point, other matrix inversion algorithms can be used that avoid the initial expensive step of LU decomposition.

- Currently, *ArraySim* can be invoked in stand-alone mode or in a graphical user interface (GUI) as discussed in the appendix. This interface can be made more automated with the circuit simulator, Transim, and active analysis procedure. Presently, once the port based admittance matrix with port currents/voltages are computed, Transim (circuit simulator) is run that uses Y-port information to compute amplifier currents and voltages. This step can be reduced into a single process by invoking *ArraySim* inside Transim's netlist and avoid unnecessary hand moving of *ArraySim* files thus enhancing overall efficiency of the system.

ArraySim can be made more effective by introducing more MoM based routines describing variety of structures. This is analogous to adding more circuit models in Spice. Currently, SSS and folded-slot structure based MoM routines are interfaced as shown in Figure 5.1. By expanding the ability of *ArraySim*, we can model and study behaviors of various spatial power combiners.

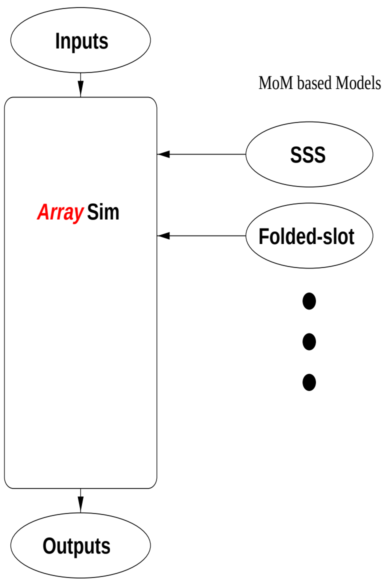


Figure 5.1: More MoM based routines can be interfaced to handle more structures

Bibliography

[1] J. W. Mink, “Quasi-optical Power Combining of Solid-State Millimeter Wave Sources,” *IEEE Microwave and Guided wave letters*, Vol. MTT-34, Feb 1986, pp. 273-279.

[2] R. F. Harrington, “Field Computation by Moment Methods,” *New York, Macmillan*, 1968.

[3] R. Collin, “Antennas and Radiowave Propagation,” *McGraw-Hill, Inc.*,1985.

[4] T. Itoh, “Numerical Techniques for Microwave and Millimeter-Wave Passive Structures,” *John Wiely and Sons, Inc.*,1989.

[5] R. E. Collin, “Foundations for Microwave Engineering,” *McGraw-Hill, Inc.* ,1992.

[6] J. E. Perry and H. D. Levin, “An Introduction to Object-Oriented Design in C++,” *Addison-Wesley Pub. Cp.*,1996.

[7] R. Sedgewick, “Algorithms in C++,”*Addison-Wesley Pub. Cp.*,1992.

[8] “http://www.sgi.com/Technology/STL/stl_index_cat.html”

[9] N. Balabanian, T. A. Bickart “Electrical Network Theory,” *John Wiely and Sons, Inc.*,1969.

[10] “Student Edition of Matlab,”*Prentice Hall.*,1995.

[11] S. M. Rubin, “Computer Aids for VLSI Design,” *Appendix: Caltech Intermediate Format, Addison-Wesley Pub. Cp.*,1987.

[12] W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling “ Numerical Recipes in C The Art of Scientific Computing,” *Cambridge University Press.*,1988.

[13] C. Chen and N. G. Alexopoulos, “Radiation by Aperture Antennas of Arbitrary Shape Fed by a Covered Microstrip Line,” *IEEE nnas and Propagation Society International Symposium, 1995*, Vol. 4, June 1995, pp. 2066-2069.

[14] Mostafa N. Abdulla and Michael B. Steer, “A Finite Double-Layer Slot Array Structure for Spatial Power Combining Amplifiers,”*IEEE AP-S.*, June 1998.

[15] M. B. Steer, M. N. Abdulla, C. E. Christoffersen, M. Summers, S. Nakazawa, A. Khalil and J. Harvey, “Integrated Electromagnetic and Circuit Modeling of Large Microwave and Millimeter-Wave Structures,”*IEEE AP-S.*, June 1998.

[16] R. A. York and Z. B. Popovic, “Active and Quasi Optical Arrays for Solid-State Power Combining,” *John Wiely and Sons, Inc.*,1997.

[17] C. A. Balanis, “Antenna Theory: Analysis and Design,” *Wiley*, New York, 1982.

[18] H. Tsai, “Quasi Optical Amplifier Arrays and FDTD Analysis of Planar Antennas,” *Phd Thesis, Dept. of ECE UCSB*,1995.

[19] Mostafa N. Abdulla, Usman A. Mughal, Huan-Shang Tsai, Michael B. Steer and Robert A. York, “A Full-Wave System Simulation of a Folded-Slot Spatial Power Combining Amplifier Array,” *accepted for publication in IEEE MTT-S Int. Microwave Symp.*, June 1999.

[20] Mostafa N. Abdulla, Usman A. Mughal and Michael B. Steer, “Network Characterization for a Finite Array of Folded-Slot Antennas for Spatial Power Combining Application,” *submitted to IEEE AP-S*, July 1999.

[21] H. Tsai, M. J. W. Rodwell and R. A. York, “Planar Amplifier Array with Improved Bandwidth using Folded-slots,” *IEEE Microwave and Guided wave letters*, Vol. 4, No. 4, April 1994, pp. 112-114.

[22] K. C. Kupta, R. Garg, I. Bahl, P. Bhartia “Microstrip Lines and Slotlines,” *Arttech House, Inc.*, Norwood, MA, 1996.

Appendix A

Effective Isotropic Power Gain

Effective Isotropic Power Gain, EIPG, refers to the actual measured gain of the quasi-optical system which includes a transmitting horn, antenna array with active devices and polarizers.

The EIPG measurement setup is shown in Figure A.1. The signal is transmitted by a standard gain horn with vertical polarization. The signal is then amplified by the active antenna array and then received by another standard gain horn with horizontal polarization. If P_{tran} and P_{meas} are the transmitted and received power, then according to [18]:

$$P_{meas} = P_{meas}\frac{\lambda_o}{4\Pi R}^{4} G^2{}_{horn}G^2{}_{array}G_{amp} \tag{A.1}$$

where λ_o is the wavelength in free space, G_{horn} and G_{array} are the gain of the horn and

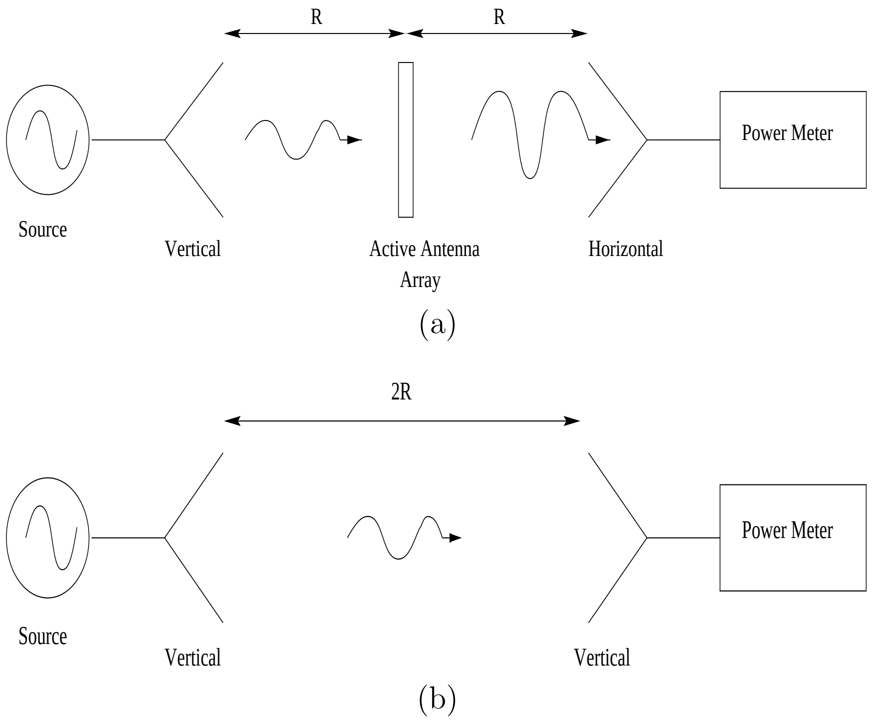


Figure A.1: Measurement setup for computing EIPG (a) Measurement (b) Calibration

antenna array while G_{amp} is the gain of the amplifier connected in the array.

As discussed in [18], measurement apparatus dependence is eliminated by the calibration procedure shown in Figure A.1b. In this case the array has been removed and the receiving and the transmitting horn have the same vertical polarization. The power received in the this case, P_{cali}, is expressed as;

$$P_{cali} = P_{trans} \frac{\lambda_o}{8\Pi R}^2 G^2{}_{horn} \tag{A.2}$$

Using Equations A.1 and A.2, a term EIPG is defined as

$$EIPG = G^2{}_{array} G_{amp} = \frac{P_{meas}}{4P_{cali}} \frac{4\Pi R}{\lambda_o}^2 \tag{A.3}$$

ArraySim uses the definition on the left hand side of Equation A.3 to compute the EIPG. Note, EIPG is not the amplifier gain, G_{amp}

Appendix B

Usage of ArraySim

Two steps are involved to run *ArraySim*:

- Converting layout CIF into modified CIF file
- Running ArraySim with input parameter file and modified CIF file

For passive analysis the usage is:

```
cifParser filename.cif
mom input_param.txt filename.cif
```

cifParser converts *filename.cif* into modified CIF file keeping the same name, *filename.cif*. *ArraySim* executable is called *mom* that takes two arguments; *input_param.txt* and *filename.cif*.

For active analysis, the usage is:

```
PostTransim input_param.txt filename.cif
```

where *input_param.txt* and *filename.cif* are the same files used for passive analysis discussed earlier. An *Outputs* and *Inputs* directory has to be created to contain all output files.

B.1 Input Files

```
           Enter data in following format.
value   Description

2       antenna type: 1:sss 2:cpw 0:not defined
2.0     frequency in GHz
0.635   substrate height in mm
10.8    epsilon_r
1       Search Table on/off.  on == 1 off == 0
1e-6    Search Table Tolerance Level (Default: 1e-6)
0       Print Individual Matrices and Complete Matrix
0       Print Inverted Matrix (Default on)
2.0     START freq (GHz). Should be same as one in line 2
4.2     END frequency (GHz)
0.1     Step size for frequency range
```

```
180     Characteristic Imp of cpw in Ohms.
0       new port definition flag. If 1 use new method for Y_nn
1       stub length factor. if 1 then port is 1 cell away if 2.
************************** INCIDENT WAVE INFO ***************
1       E_o to generate input for antenna  in V/m
0       Ein_x incident in V/m
1       Ein_y incident in V/m
************************** FAR FIELD INFO ******************
10      Far Field Distance in meters
************************** NEAR FIELD INFO *****************
0.030    near field window length (m) i.e l_x
0.030    near field window width (m)  i.e l_y
75       no. of columns (for resolution)
75       no. of rows (for resolution)
0.0008   distance away from array in meters
1        by pass field analysis
```

B.2 Output Files

Output files include network parameter files containing network parameters (S,Z and Y) for each frequency point, set in *input_param.txt*, matrix files that include **G** matrix as a whole, individual sub-matrices, and inverted **G** matrix, output magnetic currents in x and y direction on the slots, all currents i.e. on all layers, data for Transim (circuit analysis), data for active analysis, near and far field data.

B.3 Matlab File

```
%*********************************************
%
% File: nearplot.m
%    This file plots the Ex and Ey fields
%    obtained from near field analysis
%
% History:
%   Mostafa       Created
%   Usman Mughal   Edited
%*********************************************
clear;
fid=fopen('ex');
%fid=fopen('ex_phase');
%fid=fopen('ey');
%fid=fopen('ey_phase');
a=fscanf(fid,'%g ',[1 inf]);
a=a';
fclose(fid);
n=length(a);
for i=1:n;
```

Table B.1: ArraySim Output Files

Type	File Name	Description
Matrix files	$out_IncidentVector.txt$	contains all sub-matrices
	$out_Complete.txt$	Full MoM **G** matrix
	$out_Inverse.txt$	Inverted MoM **G** matrix
Network Files	$out_Reduced_Mat.txt$	contains port based reduce
		matrix
	$out_SPort.txt$	port S parameters
Current Files	$out_AllCurrents.txt$	current on all layers
	$out_Current_AllFreq_x.txt$	magnetic current in
		x direction for all freq
	$out_Current_AllFreq_y.txt$	magnetic current in
		y direction for all freq
	$out_Current_x.txt$	magnetic current
		in x dir for last freq pt.
	$out_Current_y.txt$	magnetic current
		in y dir for last freq pt.
Transim Files	$out_TransimExitationInfo.txt$	I or V sources
		at the ports used to
		excite the active device
	$out_TransimPortInfo.txt$	contains port group
		num. and port
		num. with frequency info.
	$out_PostTransimInput.txt$	contains row num. of **G**
		where ports are located.
		This info. is used to create
		a new excitation vector.
Near Field	ex	E in the x direction
	ey	E field in the y direction
	ex_phase	Phase of E in the x dir
	ey_phase	Phase of E in the y dir
Far Field	EIPG.dat	Isotropic power gain in dB
		in the y-dir
	epp0.dat	radiation pattern phi=0
	etp0.dat	radiation pattern theta=0
	epp90.dat	radiation pattern phi=90
	etp90.dat	radiation pattern theta=90

```
  z(i)=a(i);
 end
n=sqrt(n);
for i=1:n
    for j=1:n
    Z(i,j)=z((i*n-n)+j);
     end
end
Z=Z';  x=1:n;  y=x;
figure
colormap(gray);
grid;   contour(x,y,Z,10);  colorbar;
figure
surf(x,y,Z)  mesh(Z)  hidden on
colormap(hsv); colorbar;
zlabel('E (V/m)')  xlabel('X')  ylabel('Y')
colormap(hot) colorbar
figure
contour(x,y,Z,10); colorbar
figure
pcolor(x,y,Z)  axis([1 n 1 n]) shading flat
colormap(hot) colorbar
```

B.4 Makefile

```
# Makefile for MOM
#
# Auther: Usman Mughal
#
# Date: 11/10/97
#
# Use: - the  xlC   compiler on IBM's RS6000,
#      - the  g++   compiler on SUN workstations.
# Remove the comment flag '#' from the
# respective "CPP" definition.
# Compile your project with:
# "make"
#
# "make clean" will delete all extra files
# and executables (useful when switching systems).
#
CXX=g++
#CXX=xlC
#
FC=g77
#
#STL=-lstdc++
#
#CXXFLAGS=
CDFLAGS= -L/ncsu/gnu/lib -L/afs/eos.ncsu.edu/ \
          contrib/gcc272/lib -lf2c
#
XLIB=-lX11
#
# Set the DEBUG definition to be empty if you do not
# want debug.
#
#DEBUG=-DDEBUG -g
#
#use this for Active Analysis
#CXXSRC = PostTransim.C
#use this for Passive Analysis
CXXSRC = main_new.C
FSRCS =  interpolation.f ginterpolated.f math.f \
         matrix_element.f \
 param_S.f scalargreen.f spatial_integrand.f \
 spatial_mat_elem.f \
 spectral_integrand.f zelement.f nearfield.f \
 zinput.f  matrix_inversion.f cpw_farfield.f\
 cpw_interpolation.f cpw_yel.f param_Z_to_S_.f
OBJS = $(FSRCS:.f=.o) $(CXXSRC:.C=.o)
#use this for Active Analysis
#all: PostTransim
#
#mom: $(OBJS)
# $(CXX) -o PostTransim $(DEBUG) $(OBJS) $(CDFLAGS)
#$
all: mom
#
mom: $(OBJS)
$(CXX) -o mom $(DEBUG) $(OBJS) $(CDFLAGS)clean:
-rm -f $(OBJS)
#use this for Avtive Analysis
#depend:
# g++ -M PostTransim.C >> Makefile
depend:
g++ -M main_new.C >> Makefile
```

B.5 Sample Run

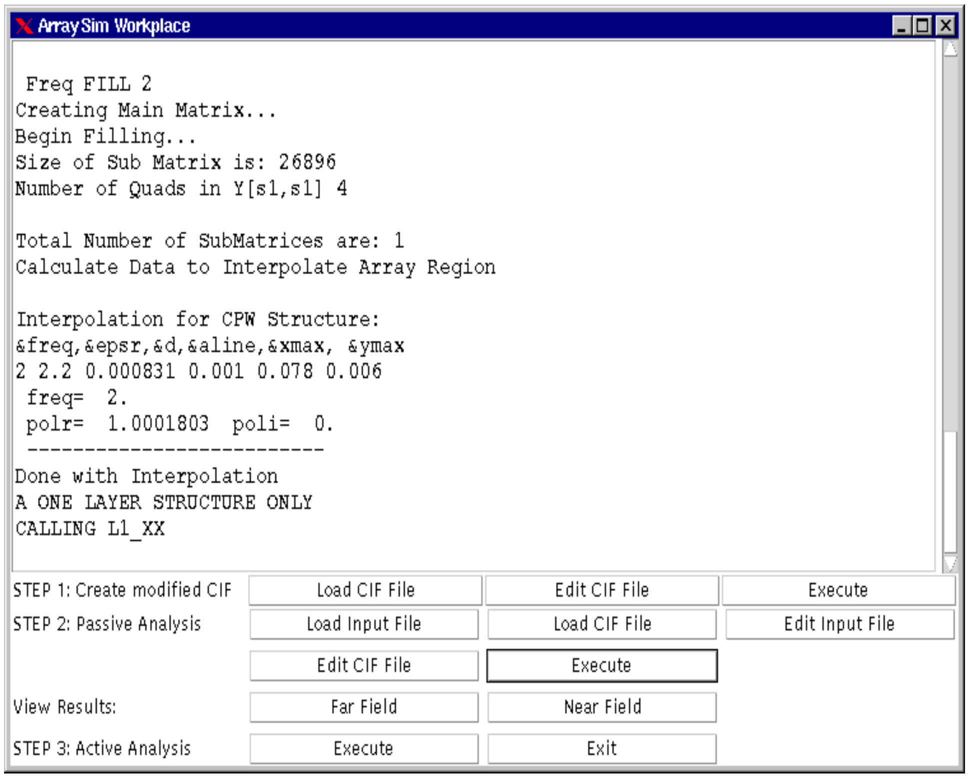
```
%
*********************************************************
*                                                       *
*          ArraySim for Quasi Optical Systems           *
*                                                       *
* Simulation Begins                                       *
*                                                         *
* Version 3.0    01/25/1999                               *
* ERL Group                                               *
* North Carolina State University                         *
***********************************************************
Parsing CIF FILE...
Num of Ports 1
ORDER OF PORTS IN PORTARRAY[]
9000,   -12500  Port Group: 1
Num of Layers: 1
Layer 1 cells: 264
Layer 2 cells: 0
Layer 3 cells: 0
L1_BasisCell_x[ ] size is: 201

L2_BasisCell_x[ ] size is: 0

L3_BasisCell_x[ ] size is: 0

L1_BasisCell_y[ ] size is: 201

L2_BasisCell_y[ ] size is: 0

L3_BasisCell_y[ ] size is: 0
Reading INPUT FILE...

Antenna Structure: CPW
Frequency (GHz):4    Substrate Height (mm):  0.635
Epsilon_r:      10.8 ST_on/off:      1   maxtol: 1e-06
Print Matrices: 0    Print Inverted Mat: 0
Start Freq:     4    End Freq:       5.2 Step Freq: 0.1
Charateristic Impedance of Stripline, Zc :  180
USE new port definition:  0

Incident Field Const: 1
Incident E-Field x:   0
Incident E-Field y:   1
Far Field Distance:   10
window_length_x 0.03
window_length_y 0.03
numOfCols       75
numOfRows       75
nearFieldDistance     0.0008
By Pass Field Analysis: 1
xmin :500 xmax: 26500
xMax_distance 27000
ymin :-19500 ymax: 6500
yMax_distance 27000
```

```
Frequency: 4
Done with Internal Parsing!
FREQUENCY : 4

 Freq FILL 4
Creating Main Matrix...
Begin Filling...
Size of Sub Matrix is: 161604
Number of Quads in Y[s1,s1] 4

Total Number of SubMatrices are: 1
Calculate Data to Interpolate Array Region

Interpolation for CPW Structure:
&freq,&epsr,&d,&aline,&xmax, &ymax
4 10.8 0.000635 0.001 0.027 0.027
 freq=  4.
 polr=  1.00118569  poli=  0.
 ---------------------------
Done with Interpolation
A ONE LAYER STRUCTURE ONLY
CALLING L1_XX
 elm found: 39865 elm not found: 536**
CALLING L1_XY
 elm found: 39749 elm not found: 652**
TRANSPOSING QUAD1 to get QUAD 2 for L1_YX
CALLING L1YY
 elm found: 39865 elm not found: 536**
Merging all 4 Quads

Merging all submatrices to form one Matrix...
End of Filling...

LU Factorizing....
Calling Matrix_inverse

Inverting Matrix....

Reducing Matrix....

Calculating Current Vector...

Exiting Structure with incident Field....

CPW ANTENNA
Plane Perpendicular Incident Field with Hy zero
Printing Incident Vector to file....
Multiply Inverted Matrix with Incident Vector
LHS VECTOR
```

```
Transim Port Parameters....

Converting Z port params to Y port parameters...

1.6816649013e-06   ,  1.4958530703e-06
Calculating S Parameters...

Converting Z to S param

Printing Reduced Port Matrix...

CPW Array FAR FIELD ANALYSIS...
freq:=  4.
nx:= 201
ny:= 201
ax:=  0.001
ay:=  0.001
Phi = 0 degrees
 |E| theta maximum: 0.000743245921 V/m
 |E| phi maximum:   0.000417748433 V/m
Phi = 90 degrees
 |E| theta maximum: 0.000419009956 V/m
 |E| phi maximum:   0.000743258345 V/m
 EIPG :   1.54894234
 EIPG (dB):   1.9003525
 EIPR (mW):   0.000290037328
 EIRP (dBm):  -35.375461
DONE CPW Array FAR FIELD ANALYSIS...

next freq....
```

B.6 GUI Version

A simple GUI (graphical user interface) is created to ease the analysis. Figure B.1 shows the GUI which is written in *Java*. The graphical user interface allows to run ArraySim by clicking respective buttons. The analysis is broken into three steps. In the first step the raw CIF file is converted into modified CIF file. In the second step, the modified CIf and the input parameter file is loaded in and then *Execute* is clicked to run the passive simulation. At the end of the passive simulation, step 3 is performed which executes post transim simulation.



```
 Freq FILL 2
Creating Main Matrix...
Begin Filling...
Size of Sub Matrix is: 26896
Number of Quads in Y[s1,s1] 4

Total Number of SubMatrices are: 1
Calculate Data to Interpolate Array Region

Interpolation for CPW Structure:
&freq,&epsr,&d,&aline,&xmax, &ymax
2 2.2 0.000831 0.001 0.078 0.006
 freq=  2.
 polr=  1.0001803  poli=  0.
 --------------------------
Done with Interpolation
A ONE LAYER STRUCTURE ONLY
CALLING L1_XX
```

Figure B.1: A Graphical User Interface of ArraySim

Appendix C

C++ Class Headers

This chapter includes alll C++ class headers that are used in *ArraySim*. Some of the member functions have lengthy implementation and thus their definition is not listed. However, for complete source code and header information visit:

`http://ncsu/erl/mbs_group/work/www/ArraySim.html.`

C.1 Complex.h

```
/////////////////////////////////////////////////////////////
// Name: Usman A Mughal
// File: Complex.h
//
// Description:
//      This class contains the definition of complex numbers
//      with their regular functions.  An Element in the G
//      matrix is of type Complex.
/////////////////////////////////////////////////////////////
class Complex{

   public:
      double real,
             imag;

Complex(){real =-1; imag=-1;}
Complex( double r, double i){ real = r; imag = i;}
void initialize(double r, double i);
friend ostream& operator<<( ostream& out,
    const Complex& c);
int operator==(const Complex& c);
Complex& operator=(const Complex& rhs);
///////////// adding a real to a Complex number ///////////
friend Complex operator+( const Complex& lhs, float rhs);
friend Complex operator+(float lhs, const Complex& rhs);
///////////// adding Complex with Complex //////////////////
friend Complex operator+( const Complex& lhs,
  const Complex& rhs);
friend Complex operator-(const Complex& lhs, float rhs);
// the following friend function definition uses operator
// overloading "-" .
// It subtracts  a  Complex number from a real
friend Complex operator-(float lhs, const Complex& rhs);
// the following friend function definition uses operator
// overloading "-" . It subtracts two Complex numbers
friend Complex operator-(const Complex& lhs,
 const Complex& rhs);
friend Complex operator*(const Complex lhs,
const Complex rhs);
// the following friend function definition uses operator
// overloading "*" . It multipies a Complex number to a real
friend Complex operator*(const Complex& lhs, float rhs);
// the following friend function definition uses operator
// overloading "*" . It multipies a real to a Complex number
friend Complex operator*(float lhs, const Complex& rhs);
friend Complex operator/(const Complex& lhs ,
 const Complex& rhs);
// the following friend function definition uses operator
// overloading "/" . It divides a real by a Complex number
friend Complex operator/(float lhs , const Complex& rhs);
// the following friend function definition uses operator
// overloading "/" . It divides a Complex number by a real
friend Complex operator/(const Complex& lhs , float rhs);
double mag();
double phase();
 }; //end of class Complex
```

C.2 Port.h

```
/////////////////////////////////////////////////////////////
//  Name:  Port.h
//  Auther:
//         Usman Mughal
//  Description:
//   This class contains neccessay information about a port
//   All ports are contained in a PortArray which is declared
//   InternalParser.  Ports in PortArray[] are ordered w.r.t
//   port coordinates.
/////////////////////////////////////////////////////////////

class Port{
 public:
  double x;  // x and y coordinates of the port position
  double y;
  int indx_x, indx_y;
  // row and col indeces of port in the
  // inverted Matrix (Matrix Manipulation)
  int index;
  ndex number of the port in
  // single dimmension C_Invert_Matrix[].  It is
  // used to get the row and col indeces
  // which are
  // then assigned to indx_x and indx_y.
  int portGroup;
  // An amplifier is two port device
  // such that the two ports belong to a
  // single group.  In an array there are
  // several amplifiers and each amp's
  // port belong to a certain group
  // e.g in layout 1:2 means port 1 beloging
  // to group two.
////////////////// Constructors ///////////////////////
Port();
Port(double x1, double y1);
////////////////////// Overload Operators /////////////////
friend ostream& operator<< (ostream& out, const Port& p);
int operator< ( const Port& c);
Port& operator= ( const Port& p );
}; // end of class Port
```

C.3 Layer.h

```
////////////////////////////////////////////////////////
// Name:  Usman A. Mughal
// File:
//    Layer.h
//
// Description:
//    Contains attributes of a Layer.
////////////////////////////////////////////////////////
class Layer {

   public:

   int     type;             // layer type e.g 1, 2, 3
   double  dielectric;       // dielectric of layer
   double  conductivity;     // zero for air
////////////////// Constructors //////////////////
Layer();
Layer( int t, double d, double c) ;
/////////////////// Overload Operators ///////////
Layer& operator= ( const Layer& rhs);
int operator== (const Layer& L);
friend ostream& operator<<(ostream& out, const Layer& L);
};  // end of class Layer
```

C.4 Cell.h

```
////////////////////////////////////////////////////////////
// Name of File: Cell.h
//
// Authur:
//      Usman A Mughal
//
// Description:
//      Class declaration of a unit Cell.  The input
//      CIF file has Cell information which are then
//      combined to form BasisCell
////////////////////////////////////////////////////////////
#define false 0
#define true  1
///////////// Class Declaration for Cell ////////
class Cell{

   public:
   double x, y;   // Position of Unit cell in the layer
                  // both x and y denote the
  // center of the unit cell
   double dx, dy; // Dimmensions of a unit cell
   Layer L;
////////////////////////// constructors ////////
Cell();
Cell ( double     x1,
       double     y1,
       double     dx1,
       double     dy1
       );
///////////////// Overloading operators ////////
Cell& operator= ( const Cell& c );
int operator< ( const Cell& c);
friend ostream& operator<<( ostream& out,
const Cell& c);
int operator== (const Cell& c);
void absDistance (double &xd, double &yd,
  const Cell& c );
///////////////////////// getters ///////////////
double getX(){ return x; }
double getY(){ return y; }
///////////////////////// setters ///////////////
void setX(double X){ x = X;}
void setY(double Y){ y = Y; }
}; //end of class Cell
```

C.5 BasisCell.h

```
////////////////////////////////////////////////////
//  Authur:
//       Usman A Mughal
//  File Name:
//       BasisCell.h
//  Description:
//   This file contains the definition of BasisCell.h.
//   BasisCell in this case is a triangular cell with
//   a rectangular base.  A BasisCell is formed by the
//   combination of two Cells.  The center of the
//   BasisCell is thus the center point of the interface
//   of the two Cells forming the BasisCell.
//   The dimensions of the BasisCell is the same as
//   that of a unit Cell i.e. same dx and dy.
/////////////////////////////////////////////////////

class BasisCell{

   public:
   //member variables
   double x, y;    // Center of BasisCell
   double dx, dy;  // Dimension of BasisCell
   int direction;
   // direction of BasisCell, i.e. x or y where
   // x ==1 amd y == 2
   int port;
   // if port then its value is '1' else '-1'
   int type;       // sourse or test ?
   Layer L;
   //member functions
   //default constructor
BasisCell() { x = -1; y = -1; dx = -1; dy =-1;
     direction = -1; type = -1;
     port = -1; }
//constructor
BasisCell( double x_b,
   double y_b,
   double dx_b,
         double dy_b,
         int d,
         Layer t );
//copy constructor
BasisCell (const BasisCell& b);
//overload identity operator
int operator== (const BasisCell& b);
//overload less than operator
 int operator< ( const BasisCell& c);
//overload assignment operator
 BasisCell operator= ( const BasisCell& rhs);
//overload output operator
friend ostream& operator<< (ostream& out,
    const BasisCell& b);
}; // end of class BasisCell
```

C.6 Element.h

```
/////////////////////////////////////////////////////
//  Authur:
//        Usman A Mughal
//  File Name:
//        Element.h
//  Description:
//        Definition of Element class.
//        Element is created by the reaction
//        of two BasisCells.
//
/////////////////////////////////////////////////////
class Element{

   public:

    Complex value; // value of the element calculated
                   // using Fortran prog
    int type;      // xx, xy, yx, yy
    BasisCell parent_src,
              parent_tst;
    double delta_X, delta_Y, dist;
//////////////////// Constructors /////////////
Element(){ type = -1;}
Element(Complex v, int t, BasisCell& src,
BasisCell& tst);
//Returns the distance between a src and tst cell that
//create an element
double distance();
double deltaX(){
   return (parent_src.x - parent_tst.x) ;
}
double deltaY(){
   return (parent_src.y - parent_tst.y) ;
}
//////////////////////// Overload Operators /////////
int operator== ( Element& e);
int operator<(Element &e);
/////////////////// Member Functions /////////////
int lessthan(Element &e1);
double absDeltaX();
double absDeltaY();
void setElement(Complex v, int t, BasisCell& src,
BasisCell& tst);
friend ostream& operator<<(ostream& out,
 const Element& e);
Element& operator= ( const Element& rhs);
void negateValue();
};
```

C.7 InternalParser.h

```
/////////////////////////////////////////////////////////
// Authur:
//      Usman A Mughal
// Name of File:
//      InternalParser.h
// Description:
//  This class reads the cif file for the layout
//  and then orders the cells of each layer in
//  separate arrays. These ordered cells and then
//  used to create BasisCells for each layer.
//  Note: Each layer has two BasisCell Associated
//  to it. i.e. BasisCell_x and BasisCell_y.
//  The interface function of this class with the
//  main.C is through the method parse().
//////////////////////////////////////////////////////////

class InternalParser{

   public:

   int numOfLayers, numOfPorts,
       L1_numOfCells,    L1_numOfCells_x, L1_numOfCells_y,
       L2_numOfCells,    L2_numOfCells_x, L2_numOfCells_y,
       L3_numOfCells,    L3_numOfCells_x, L3_numOfCells_y,
       L1_numOfBCells_x, L1_numOfBCells_y,
       L2_numOfBCells_x, L2_numOfBCells_y,
       L3_numOfBCells_x, L3_numOfBCells_y;
   int SearchTable_flag;  //if 1 then on if 0 then off
   int PrintMatrices_flag; //if 1 then on if 0 then off
   int PrintInvertedMat_flag; //if 1 then on if 0 then off
   int new_port_def_flag;
                       //if 1 then use the new port def for
                       //Y port parameters
   int antenna;
   // if 1 then slot-stripline-slot (3 layers)
   // if 2 then cpw  (1 layer)
   double E0; // const for
              // incident input field. Read from input file
   double Ein_x, Ein_y;
   int TypeOfField_flag; //1: plane perpendicular
                         //2: plane field with Q=Q'
   double farFieldDistance;
   double window_length_x;
   double window_length_y;
   int    numOfCols;
   int    numOfRows;
   double nearFieldDistance;
   double xoff,yoff;
   double xMax_distance, yMax_distance;
   double freq, height, epsilon_r, max_tol;
   double Start_freq, End_freq, Step_freq,
          Zc, stub_len_factor;
   int bypass;  //if one then bypass field analysis
                // till last freq point
   Port *PortArray;
   Cell *Layer1Cells,
        *Layer2Cells,
*Layer3Cells;
   BasisCell *L1_BasisCell_x, *L1_BasisCell_y,
     *L2_BasisCell_x, *L2_BasisCell_y,
     *L3_BasisCell_x, *L3_BasisCell_y;

InternalParser();
~InternalParser();
int L1_L2_sym();
void getPorts(istream& ins)
void countCells( istream& ins){
void createCellArrays(istream& ins){
// This function is called after createLayerCellArrays()
// This function will fill the Layer#Cells[] from the .cif
// file.  The cells in the arrays are then later ordered
// w.r.t. to their coordinates by orderLayerCells()
void swap(Cell a[], int min, int i);
void swap(Port a[], int min, int i);
void swap(BasisCell a[], int min, int i);
void selectionSort(Cell a[], int N);
void selectionSort(BasisCell a[], int N);
void selectionSort(Port a[], int N);
void orderPorts()
void orderLayerCells();
// This function will order the cells in each
// array w.r.t their a 7 y coordinates
void createBasisCellArrays();
// This function will use the cell arrays
//orderd by orderLayerCells()
// and create two BasisCellArrays per Layer.
//i.e BasisCell_x & BasisCell_y
void cal_xMax_yMax_distance();
void Find_Max_x_Max_y(double& Min_x, double& Min_y,
      double& Max_x, double& Max_y,
               Cell a[], int n);
void readInputFile( istream &ins );
void parse( istream& ins, istream& ins2);
};
```

C.8 QuadMatrix.h

```
//////////////////////////////////////////////////////
//  Name:
//    Usman A Mughal
//  File:
//    QuadMatrix.h
//
//  Description:
//  This class contains the attibutes of a QuadMatrix.
//  QuadMatrix is a part of SubMatrix object.
//  It contains Elements.
//
///////////////////////////////////////////////////////
class QuadMatrix {

 public:
  int quadSize_x, //number of rows in QuadMatrix
      quadSize_y, //number of rows in QuadMatrix
      size_t,  //total Elements in a QuadMatrix
      offset,  //Equal to number of columns
      empty;   //asserted if QuadMatrix is empty
  Element* elmArray; // a quad contains an array of
                     // Element types
QuadMatrix() { offset = 0;
       quadSize_x = 0;
          quadSize_y = 0;
          size_t = 0;
          empty = 0;
          elmArray = NULL;}
void QuadMat(int size_x, int size_y, int os);
~QuadMatrix(){
   delete [] elmArray;
}
//////////////////////// Transpose a Matrix //////////
//
//This function takes a QuadMatrix as a parameter and
//transposes it.  The current instance thus becomes the
//transposed QuadMatrix.
void transpose( QuadMatrix &q);
/////////////////// Add Element to Matrix ////////////
void addElement( Complex& v, int type, BasisCell& s,
                 BasisCell& t,
 int row, int col);
/////////////////// Fill Zeros //////////////////////
void fillZero();
/////////////////// Overload Operators ///////////////
friend ostream& operator<<(ostream& out,
                           const QuadMatrix& q);
QuadMatrix& operator=(const QuadMatrix &rhs);
}; // end of class QuadMatrix
```

C.9 SubMatrix.h

```
///////////////////////////////////////////////////////
// Name:
//    Usman A Mughal
// File:
//    SubMatrix.h
//
// Description:
//  This class defines the SubMatrix which is inside the
//  MainMatix.  The number of SubMatrices depends upon the num
//  of layers.  i.e number of sumatrices is equal to
//  (num of layers * num of layers).
//  Each submatrix is made up of max. four quad matrices.
//  Each quad has reaction information e.g quad[0] has xx
//  reaction info, quad[1] has xy etc.
//  The user can do the following with a sub matrix matrix:
//  - negate and transpose a matrix,
//  - print a submatrix to a file
//  - assign a submatrix to another
//  - merge the quad matrices inside the submatrix to form one
//    solid submatrix.
//////////////////////////////////////////////////////////////
class SubMatrix {

   public:
     int subSize_x;
     int subSize_y;
     int offset;
     int size_t;
     int numOfQuads;
     int maxNumOfQuads;
     Element* elmSubArray;
     QuadMatrix* quads;   // Max 4 in number

SubMatrix();
void setSize(int num);
~SubMatrix();
//////////////////////// Merge //////////////////////
void merge();
///////////////////// Negate and Transpose //////////
void negateAndTranspose(SubMatrix &s);
//////////////////////// Overload ///////////////////
friend ostream& operator<<(ostream& out,
const SubMatrix& s);
SubMatrix& operator= (const SubMatrix& rhs);
void negReverse(const SubMatrix& s);
};
```

C.10 MatrixManipulation.h

```
////////////////////////////////////////////////////////
// Name:  Matrix_Manipulation.h
//
// Auther:  Usman A Mughal
// Description:
//   This class contains varous funtions for solving
//   a complex matrix.  It will Inverse the Main_Matrix
//   using LU decomposition method.
//   Reference:
//      Numerical Recipies in C  pp43. ed 1988
//////////////////////////////////////////////////////////
extern "C"
{
void  matinv_();
}
class Matrix_Manipulation{

    public:

    Complex *C_Inverse_Matrix;
    Complex *Reduced_Matrix;
    float d;
    int N;
    int offset;  // for LU_dcmp_Matrix
    int size_c;

////////////// default constructor ////////////////////
Matrix_Manipulation()
```

```
///////////// constructor used for ArraySim ///////////
Matrix_Manipulation(MainMatrix& m);
///////////////// destructor ////////////////////////
~Matrix_Manipulation();
/////// call this funtiion to begin matrix manip ///////
void Begin_Matrix_Manip(MainMatrix &m ,
InternalParser& p);
////////////////////// get value ////////////////////
// it was with Complex instaed of Element return
// a[((row-1)*offset + col)];
// and the complex vector was from 1 to n.
inline Complex getValue_mm(int row,int col,
 int offset, Element a[])
 {return a[((row)*offset + col)].value;}
inline Complex getValue_mm(int row,int col,
int offset, Complex a[])
{return a[((row-1)*offset + col)];}
/////////////// set value same as a[i][j] = value /////
//  was a[( (row-1)*offset + col ) ] = rhs;
inline void setValue_mm(int row, int col, int offset,
Element a[], Complex& rhs)
      {a[( (row)*offset + col ) ].value = rhs;}
inline void setValue_mm(int row, int col, int offset,
Complex a[], Complex& rhs)
      {a[( (row -1)*offset + col ) ] = rhs;}
//The following get function is for arrays
//starting from 1 to n
inline Complex getValue_1_to_n(int row,int col,
 int offset, Complex a[])
{return a[((row-1)*offset + col)];}
////////////// set value same as a[i][j] = value //////
inline void setValue_1_to_n(int row, int col,
   int offset,
                            Complex a[], Complex& rhs)
a[( (row-1)*offset + col ) ] = rhs;}
void convert_IndxTo_Row_and_Col(int &row, int &col,
int Ind, int offset);
//////////////// Fill Redeuced Matrix ////////////////
void Reduced_Port_Matrix(InternalParser& p);
/////// Make Copy of elmMainArray in the form [1..n]///
void makeCopy(MainMatrix& m, InternalParser& p);
///////////////////////// LU Decompose //////////////
void LU_dcmp(Element a[], int n, int indx[],
               float &d, int offset);
///////////////////////// Overload output operator ////
friend ostream& operator<<(ostream& out,
                       const Matrix_Manipulation& m);
///////////'////////// Matrix inverse ////////////////
void Matrix_Inverse_Fortran( Element a[], int size_a,
     int n, Complex aInv[]);
///////////////// forTran  Complex Input //////////////////
void Matrix_Inverse_Fortran( Complex a[], int size_a,
     int n, Complex aInv[])
//////////////////////////////////////////////////////////
void Matrix_Inverse( Element a[], int n, int offset,
     Complex aInv[], float d);
  // first call the LU decomposition method on matrix
  // LU_dcmp_Matrix and then inverse this matrix
  // This function call LU_dcmp() will initialize the
  // vector indx[] and re-fill LU_dcmp_Matrix[].
 /*
 a[]:  It is a Complex vector that is LU decomposed
 n:  Is the order of the input matrix (not size of)
         vector a[]
 indx:   is an output vector which records the row
       permutation effected by the partial pivoting
 d:  Is output as +-1 depending on whether the number
         of row
       interchanges was even or odd
 offset: Offset of a[].  This way we know when the
         next row begins.
 col[]:  a vector (complex) of zeros and ones used by
         LU_bksb
 aInv:   A Complex vector which is the final result.
         It is the inverse of a[].
 */
//////////////// Backsubstitution //////////////////////
void LU_bksb(Element a[], int n, int indx[],
              int offset,Complex cb[]);
//////////////////////// for complex inputs /////////////
void Matrix_Inverse( Complex a[], int n, int offset,
                     Complex aInv[], float d);
 // first call the LU decomposition method on matrix
 // LU_dcmp_Matrix and then inverse this matrix
 // This function call LU_dcmp() will initialize the
 // vector indx[] and re-fill LU_dcmp_Matrix[].
 /*
 a[]:  It is a Complex vector that is LU decomposed
 n:  Is the order of the input matrix (not size of)
         vector a[]
 indx:   is an output vector which records the row
       permutation effected by the partial pivoting
 d:  Is output as +-1 depending on whether the number
         of row
       interchanges was even or odd
 offset: Offset of a[].  This way we know when the
         next row begins.
 col[]:  a vector (complex) of zeros and ones used by
```

```
         LU_bksb
 aInv:   A Complex vector which is the final result.
         It is the inverse of a[].
 */
///////////////// Backsubstitution for complex ///////////
void LU_bksb(Complex a[], int n, int indx[],
             int offset,Complex cb[]);
//////////////// LU Decompose for complex ////////////////
void LU_dcmp(Complex a[], int n, int indx[], float &d,
             int offset);
///////// Matrix Multiplication with a real vector ///////
void multiply( Complex a[], int a_offset,
               float  b[], Complex ans[]);
   // the two matrices are multiplied as
   // ans = a*b;
/////////// multiply a matrix with a complex vector //////
void multiply( Complex a[], int a_offset,
               Complex  b[], Complex ans[]);
  // the two matrices are multiplied as
  // ans = a*b;
///// add a complex matrix with a complex vector ////////
//This fucntion adds two same sized matrices
//ans=a+b;
//for a vector with one coloumn, a_offset=1;
void add( Complex a[], int a_offset, Complex  b[],
          Complex ans[], int size);
   // the two matrices are multiplied as
   // ans = a+b;

}; // end of class definition
```

C.11 MainMatrix.h

```
//////////////////////////////////////////////////////////
// Name:
//    Usman A Mughal
// File:
//     MainMatrix.h
//
// Description:
//     This class contains the attributes of MainMatrix.
//     The is the large matrix containing SubMatrix objects
//     SubMatrix objects are merged row at a time in order
//     to merge all SubMatrix objects to form one large
//     MainMatrix.
///////////////////////////////////////////////////////////
class MainMatrix {
  public:

    int mainSize_x;  //number of columns
    int mainSize_y;  //num of rows
    int size_t;      //total elements in MainMatrix
    int offset;      //number of columns
    int numOfSubsPerRow;
    int sizePerRow ;
    int antenna;  //if 1 then sss; 2: cpw; else undefined
    SubMatrix *subMat;
    Element *elmMainArray;

MainMatrix();
MainMatrix(InternalParser &p);
/////////////////////// Compute Sixe of MainMatrix ////
void setSize( int index, InternalParser& p);
~MainMatrix();
////////////////// Access a Row and Col of Main Mat ////
 Complex getValue(int row,int col);
///////////////////// Overload output operator /////////
friend ostream& operator<<(ostream& out,
  const MainMatrix& m);
/////////////////// Merge A Row //////////////////////
int mergeRow(int start, int &elmIndex){
};
```

C.12 SearchTable

```
///////////////////////////////////////////////////////
// Name:
//      Usman A Mughal
// File:
//      SearchTable.h
// Description:
//  At the present time the search table uses
//  linear search to find an element.In later versions,
//  binary search should be used
///////////////////////////////////////////////////////
class SearchTable {

   public:
   //STL vectors. Elements can be added from both
   //sides of a vector.  STL vectors have their
   //own destructors.
   vector<Element> ElemArray_ST_xx;  // for xx quad
   vector<Element> ElemArray_ST_yy;  // for yy quad
   vector<Element> ElemArray_ST_xy;  // for xy quad.
                   // yx = [xy]T for Y
                   // therefore no ST needed for yx.
   // used to define the size of the searchtable
   int start, numOfCells;
   int index_xx;
   int index_yy;
   int index_xy;
   int entries_xx; // enteries in the search table
   int entries_yy;
   int entries_xy;
   int antenna;    //antenna type (1 for SSS, 2 for CPW)
   double tol;     //tolarance level
//////////////////// Constructors //////////////////////
SearchTable(int nOfCells, InternalParser& p);
/////////////////////// add Item ///////////////////////
void addItem ( Element &e);
///////////////////////// find /////////////////////////
// This function returns the index in the search
// table for the
// element which has been found.  This index is then
// used by
// getElement() to extract the respective Element value
int find ( BasisCell& s, BasisCell& t, double &flag_sign);
/////////////////////// getElement /////////////////////
Element& getElement(BasisCell& s, BasisCell& t, int ii)
void swap(Element a[], int i, int j);
void selectionSort(Element a[], int N);
};
```

C.13 MainAlg.h

```
////////////////////////////////////////////////////////
//
// Name:
//    Usman A Mughal
// File Name:
//   MainAlg.h
//
// Description:
//  This is the core of the program.
//  This class acts like a main
//  First we setup the sizes of all the quad matrices
//  in all the 9 SubMatrices.  Then we call
//  Reaction() to find the reaction between two
//  BasisCell arrays. Each time Reaction is called,
//  one of the quad is filled up.
//
/////////////////////////////////////////////////////////
//just for fortran
```

```
extern "C"
{
void interpolation_( );
void cpwinterpolation_( );
void matelem_( );
void cpwyelem_( );
}
struct element_value{
   double r;
   double i;
};
class MainAlg{

   public:

   int index;
   //gives the total number of Submatrices
   int position;
   //actual index position of the submatrix
   int positionTranspose;
   //The index of Submatrix to be transposed
   int numOfLayers;
   int result;
   int dummy;
   double freq, epsr, d, lline, wline,
          lslot, wslot, xoff, yoff,
  xmax, ymax, aline, bslot;
   element_value z, z1, z2;
//////////// Constructor called in main() ///////
/* This constructor allocates memory for QuadMatrices
 depending upon the number of layers.  Depending upon
 the antenna type, it calls the respective interpolation
 functions.  Then it calls the functions to fill the
 individual SubMatrices which intern call the respective
 reaction functions.  After filling all SubMatrices,
 subMatrices are merged to form one large MainMatrix.
 The MainMatrix and the individual matrices are then
 printed
 */
MainAlg(InternalParser& p, MainMatrix& m);
void OutputMatrices(InternalParser& p, MainMatrix& m );
//////////////////////// Fill Y[s1,s1] //////////////
void FillYs1s1( InternalParser& p, MainMatrix& m,
int pos);
/////////////////////////Fill W[ln,s1]//////////////
void FillWlns1( InternalParser& p, MainMatrix& m,
int pos);
//////////////////////// Fill Z[ln,ln] //////////////
void FillZlnln( InternalParser& p, MainMatrix& m,
int pos);
/////////////////////////Fill U[s1,ln] /////////////
void FillUs1ln( InternalParser& p, MainMatrix& m,
int pos, int posTran);
/////////////////////////Fill U[s2,ln] /////////////
void FillUs2ln( InternalParser& p, MainMatrix& m,
int pos, int posTran);
/////////////////////////Fill W[ln,s2] /////////////
void FillWlns2( InternalParser& p, MainMatrix& m,
int pos);
/////////////////////////Fill Y[s2,s2] /////////////
void FillYs2s2( InternalParser& p, MainMatrix& m,
int pos);
/////////////////////////Fill Y[s2,s1] /////////////
void FillYs2s1( InternalParser& p, MainMatrix& m,
int pos);
//////////////////////// Fill Y[s1,s2] //////////////
void FillYs1s2( InternalParser& p, MainMatrix& m,
int pos);
//////////////////////// setupQuadMatrix ////////////
void setupQuadMatrices( InternalParser& p,
 MainMatrix& m);
 //setting up sizes of all the quad matrices.
 //This is an initialization process.

 // The number of quads depends on the structure
 // It is max 4 in value and is set to set the
 // size of the submatrix. Each submatrix knows
 // the number of quads in it.
/////////////////////////Set up Main Matrix /////////
void setupMainMatrix(MainMatrix& m,
     InternalParser& p);

///////////////////////// Reaction ////////////////
void Reaction( BasisCell tst[], int tstCells,
              BasisCell src[], int srcCells,
      MainMatrix& m, InternalParser &p,
      int pos);
};
```

C.14 FieldAnalysis.h

```
/////////////////////////////////////////////////////
//  Name:  Usman A. Mughal
//  File:
//     FieldAnalysis.h
//
//  Description:
```

```
//   This class calls the respective near and far
//   field fortran fuctions.  This class is used in
//   passive and active analysis
////////////////////////////////////////////////////////
// The following lines are addedto avoid warning messages
// due to Fortran function call.
extern "C"
{
void  cpwfarfield_();
void  nearfield_();
}
class FieldAnalysis{
 public:
    int typeOfAnalysis;  //near = 1; far = 2;

FieldAnalysis(){typeOfAnalysis = 1;};
void nearField(InternalParser &p );
void farField( InternalParser &p);
};  // end of class FieldAnalysis
```

Appendix D

C++ Source Files

D.1 main_new.C

```
///////////////////////////////////////////////////////
//
// Name: main_new.C
//
// Auther: Usman A Mughal
//
// Description:
//   This is the main program. It creates instances
//   of InternalParser, MainMatrix, Matrix_Manipulation
//   All results are output to files.
//
///////////////////////////////////////////////////////
//just for fortran
extern "C"
{
void  paramz2s_();
void  params_();
}
void banner_end();
void banner_begin();
void Fill_and_Solve_Matrix(
     istream& ins, istream& ins2, InternalParser &p,
     double freq_point,
     ofstream &outTime,
     ofstream &outfileInverse,
     ofstream &outfileRM,
     ofstream &outfileTransimPortInfo,
     ofstream &outfileTransimExitationInfo,
     ofstream &outfileAllCurrents,
     ofstream &outfilePortCurrents,
     ofstream &outfileCurrentsForFeildAnalysis_x,
     ofstream &outfileCurrentsForFeildAnalysis_y,
     ofstream &outfileS);
```

```
void nearField_forOtherParameters(InternalParser &p);
void Calculate_S_Parameters( InternalParser& p,
                             Matrix_Manipulation& mp,
     ofstream &outfileS);
void OutputFor_Transim_Input (
     Complex aZ[],
     InternalParser& p,
     Matrix_Manipulation& mp,
     double freq_point,
     ofstream& outfileTransimPortInfo,
     ofstream& outfileTransimExitationInfo,
     Complex lhs[], Complex Ish[]);
void Convert_Z_to_Y(Complex aZ[], Complex aY[],
                   InternalParser& p,
    Matrix_Manipulation& mp
        );
void Convert_Y_to_S( InternalParser& p,
                    Matrix_Manipulation& mp,
     double S_mag[],
     double S_phase[]
     );
void Convert_Z_to_S( InternalParser& p,
                    Matrix_Manipulation& mp,
     double S_mag[],
     double S_phase[]
     );
void printS_parameters(double S_mag[],
                      double S_phase[],
       InternalParser &p,
       ofstream &outfileS);
////////////////// Set up rhs Vector //////////////////
// This vector could be either
//1.  plane incident field with Q=90 deg
//2.  plane incident field at an angle Q
//Depending upon the parameter set in the input file
//the rhs current vector will be filled with certain
//values describing the incident field.
//The rhsVector[1...n] and TypeOfField_flag is read
//from input file.
void setup_rhs_Vector(Matrix_Manipulation& mp,
                    MainMatrix& mm,
     InternalParser &p,
     Complex rhsVector[]
     );
void printIncidentVector(int size,
                 Complex rhsVector[]);
void fill_rhsVector_Plane_Perpendicular(
     Matrix_Manipulation& mp,
     MainMatrix& mm,
     InternalParser &p,
     Complex rhsVector[]
     );
/*
   We fill excitation vector in this function
   In this function we fill the rhsVector according
   to the user given input factor, E0.  For a
   cpw antenna we fill
   Hx while Hy is all zero.  For sss antenna
   (case 1) we fill Hin while Vex and Ho is all
   set to zero.
*/
void fill_rhsVector_Plane_inclined(Complex rhsVector[]);
void FindCurrent(Matrix_Manipulation& mp,
                 InternalParser& p,
 MainMatrix& mm,
 Complex rhsVector[],
 Complex lhsVector[]);
/*Ax=B;
  x = [B]*[A]^-1
  where x is lhsVector and B is the rhsVector
*/
void printInvertedMat_to_file(
     InternalParser& p,
     Matrix_Manipulation &mp,
     ofstream& outfileInverse);
void printTransimPortInfo_to_file(
     InternalParser &p,
     double freq_point,
     ofstream& outfileTransimPortInfo,
     Complex in[]
     );
void printTransimExitationInfo_to_file(
     InternalParser &p,
      Matrix_Manipulation &mp,
      double freq_point,
      ofstream& outfileTransimExitationInfo,
      Complex lhsVector[], //Ax=b x is lhsvector
      Complex Ish[],
      Complex aY[]  //[Z]*[V] = [I] so A=Z & x=V
     );
void printReducedMat_to_file(Matrix_Manipulation &mp,
     InternalParser &p,
                             ofstream& outfileRM);
void printAllCurrents_to_file(
          MainMatrix &mm, InternalParser &p,
          Matrix_Manipulation &mp,
  ofstream&  outfileAllCurrents,
  ofstream&  outfilePortCurrents,
  ofstream&  outfileCurrentsForFeildAnalysis_x,
  ofstream&  outfileCurrentsForFeildAnalysis_y,
  Complex rhsVector[], Complex lhsVector[]
  );
void Calculate_Zstub(InternalParser &p, Complex &Zstub);

 double beta,  //beta is a the propagation constant
  a,   //a is the dx of a basis of the stripline cell
  Zc,  //characteristic impedance of the stipline given
       //by the user.  It is calculated using
       //the Zc formula in Collins p171.
  c,   //speed of light.
  f,   //freq in Hz
  Pi,
  delta,  //extra stub len
  h,   // height of substrate in meters
  stub_len_factor;
  f    = p.freq*1000000000;
  Pi   = 3.141592654;
  c    = 2.998e8; //2.998 x 10^8 m/s
  Zc   = p.Zc;
  stub_len_factor = p.stub_len_factor;
  //is one if new_port_def_flag is 1 and
  //the port is only at dist of 1 cell
  //else it is a higher factor of 2
  //depending upon the number of cells
  //before the port.
double Calculate_delta_Stub_len(double beta,
        double w, double h);
//In this function the reduced matrix that contains the
//Y port param directly from the inverted matrix is edited
//using open cct. stub matching.  So, Zstub is
//first calculated
//then added to the original y11 to find the new y11 which is
//stored in the Y_vec.  This Y_vec is then converted into
//S params
void Y_param_array_to_Y_param_vec(Matrix_Manipulation &mp,
                                  InternalParser &p,
                                  double Y_vec[]);
int main(int argc, char *argv[]);
/////////////////END OF FILE /////////////////////////////
```

D.2 PostTransim.C

```
#include "headersForMain.h"
#include <set.h>

struct ltint
```

```
{
  bool operator()(const int s1, const int s2) const
  {
    return (s1 < s2);
  }
};
void readInvertedMatrix(Complex invMat[],
int sizeofinvMat);
void readTransimCurrents(Complex I_Transim[],
 int sizeofI_Transim);
void readTransimVoltages(Complex V_Transim[],
 int sizeofV_Transim);
void readExitationVector(Complex Exitation_old[],
 int sizeofI_old);
void readPortRowNum( set<int,ltint> &A);
void postTransimSimulation(InternalParser &p);
void formNewCurrentVector(set<int,ltint> &PortRowInfo,
                          int numOfPorts,
                          Complex I_Transim[],
  Complex I_new[]
  );
void printCurrents( InternalParser &p,
                    Matrix_Manipulation &mp,
    Complex lhsVector[]
    );
int main(int argc, char *argv[]);
```

Appendix E

Java Source Files

```
// ---------------------------------------------
// ArraySim shell
//     by Usman Mughal
// ---------------------------------------------
import java.awt.*;
import java.awt.event.*;
import java.io.*;
public class ArraySim {

  static ArraySim tr_shell = null;
  TextArea ta = new TextArea("",20, 80,
              TextArea.SCROLLBARS_VERTICAL_ONLY);
  GUI gui = null;
  String proj_name = "inputfile.txt";
  String proj_cif  = "filename.cif";
  // ---------------------
  // Some default strings
  // ---------------------
  String editor = "/usr/local/bin/nedit";
  //String mat_lab = "/ncsu/matlab/bin/matlab";
  String gnu_plot = "/ncsu/gnu/bin/gnuplot";
  String xtermWindow = "/usr/bin/X11/xterm";
  String cifparser = "/ncsu/erl/mbs_group/work/
                      uamughal/CIF_Parser/cifParser";
  String netscape = "/usr/local/bin/Netscape4";
  String helpfile = "file:/ncsu/erl/mbs_group/work/
                     radanchi/transim/docs/transim_docs/
     help_start_here.html";
  String arraysim = "/ncsu/erl/mbs_group/work/uamughal/
  ArraySim_01_13_99/mom";
  String postsim =
  "/ncsu/erl/mbs_group/work/uamughal/
  ArraySim_01_13_99/PostTransim/PostTransim";
  // ---------------------
static public void main(String args[]);
ArraySim();
```

```
static public ArraySim get();
public void open();
public void open_cif();
public void edit();
public void edit_cif();
public void near();
public void far();
public void help();
public void text_read(String text);
public void run_passive();
public void run_parser();
public void run_post();
public void view();
public void exit();

class GUI {

  Frame f = null;
  Panel cPanel = null;  //upper frame
  Panel nPanel = null;
  Panel sPanel = null;
  TextField near_text;
  public GUI(TextArea ta);
  Frame getFrame();

class near_text implements ItemListener {

 ArraySim parent;
 near_text(ArraySim outer)
 {
 parent = outer;
 }
 public void itemStateChanged(ItemEvent evt) {

   String s =  parent.gui.near_text.getText();
   parent.text_read(s);
      }
}
class OpenButton extends Button
        implements ActionListener
{
  private ArraySim app = ArraySim.get();
  public OpenButton() {
    this.app = app;
    this.addActionListener(this);
    this.setLabel("Load Input File");
  }
  public void actionPerformed (ActionEvent e) {
    app.open();
  }
}
class farButton extends Button
implements ActionListener
{
  private ArraySim app = ArraySim.get();
  public farButton() {
    this.app = app;
    this.addActionListener(this);
    this.setLabel("Far Field");
  }
  public void actionPerformed (ActionEvent e) {
    app.far();
  }
}
class nearButton extends Button
implements ActionListener
{
  private ArraySim app = ArraySim.get();
  public nearButton() {
    this.app = app;
    this.addActionListener(this);
    this.setLabel("Near Field");
  }
  public void actionPerformed (ActionEvent e) {
    app.near();
  }
}
class OpenCIFButton extends Button
implements ActionListener
{
  private ArraySim app = ArraySim.get();
  public OpenCIFButton() {
    this.app = app;
    this.addActionListener(this);
    this.setLabel("Load CIF File");
  }
  public void actionPerformed (ActionEvent e) {
    app.open_cif();
  }

}
class EditButton extends Button
implements ActionListener
{
  private ArraySim app = ArraySim.get();
  public EditButton() {
    this.app = app;
    this.addActionListener(this);
    this.setLabel("Edit Input File");
  }
  public void actionPerformed (ActionEvent e) {
    app.edit();
  }
}
class EditCIFButton extends Button
implements ActionListener
{
  private ArraySim app = ArraySim.get();
  public EditCIFButton() {
    this.app = app;
    this.addActionListener(this);
    this.setLabel("Edit CIF File");
  }
  public void actionPerformed (ActionEvent e) {
    app.edit_cif();
  }

}
class HelpButton extends Button
implements ActionListener
{
  private ArraySim app = ArraySim.get();
  public HelpButton() {
    this.app = app;
    this.addActionListener(this);
    this.setLabel("Help");
  }
  public void actionPerformed (ActionEvent e) {
    app.help();
  }
}
class Run_passive_Button extends Button
implements ActionListener
{
  private ArraySim app = ArraySim.get();
  public Run_passive_Button() {
    this.app = app;
    this.addActionListener(this);
    this.setLabel("Execute");
  }
  public void actionPerformed (ActionEvent e) {
    app.run_passive();
  }
}
class Run_parser_Button extends Button
implements ActionListener
{
  private ArraySim app = ArraySim.get();
  public Run_parser_Button() {
    this.app = app;
    this.addActionListener(this);
    this.setLabel("Execute");
  }
  public void actionPerformed (ActionEvent e) {
    app.run_parser();
  }
}
class Run_post_Button extends Button
implements ActionListener
{
  private ArraySim app = ArraySim.get();
  public Run_post_Button() {
    this.app = app;
    this.addActionListener(this);
    this.setLabel("Execute");
  }
  public void actionPerformed (ActionEvent e) {
    app.run_post();
  }
}
class ViewButton extends Button
implements ActionListener
{
  private ArraySim app = ArraySim.get();
  public ViewButton() {
    this.app = app;
    this.addActionListener(this);
    this.setLabel("View Output");
  }
  public void actionPerformed (ActionEvent e) {
    app.view();
  }
}
class ExitButton extends Button
implements ActionListener
{
  private ArraySim app = ArraySim.get();
  public ExitButton() {
    this.app = app;
    this.addActionListener(this);
    this.setLabel("Exit");
  }
  public void actionPerformed (ActionEvent e) {
    System.exit(0);
  }
}
```