

2007

## Parallel three-dimensional acoustic and elastic wave simulation methods with applications in nondestructive evaluation

Kevin Edward Rudd

*College of William & Mary - Arts & Sciences*

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Acoustics, Dynamics, and Controls Commons](#), and the [Computer Sciences Commons](#)

---

### Recommended Citation

Rudd, Kevin Edward, "Parallel three-dimensional acoustic and elastic wave simulation methods with applications in nondestructive evaluation" (2007). *Dissertations, Theses, and Masters Projects*. Paper 1539623332.

<https://dx.doi.org/doi:10.21220/s2-4ez7-qc09>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact [scholarworks@wm.edu](mailto:scholarworks@wm.edu).

**Parallel 3D Acoustic and Elastic Wave Simulation Methods with Applications  
in Nondestructive Evaluation**

**Kevin Edward Rudd**

**Mechanicsville, Virginia**

**B.S. Physics, Virginia Commonwealth University, 2003  
B.S. Computer Science, Virginia Commonwealth University, 2003**

**A Dissertation presented to the Graduate Faculty  
of the College of William and Mary in Candidacy for the Degree of  
Doctor of Philosophy**


**The Applied Science Department**

**The College of William and Mary  
August, 2007**

## APPROVAL PAGE

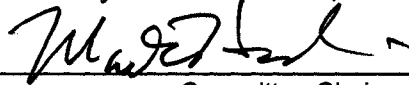
This Dissertation is submitted in partial fulfillment of  
the requirements for the degree of

Doctor of Philosophy



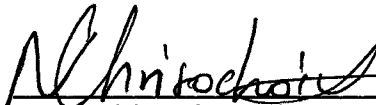
Kevin Edward Rudd

Approved by the Committee, June, 2007




Committee Chair

Professor Dr. Mark Hinders, The Applied Science Department  
The College of William and Mary



Associate Professor Dr. Nikos Chrisochoides, The Computer Science Department  
The College of William and Mary



Assistant Professor Dr. Christopher Del Negro, The Applied Science Department  
The College of William and Mary



Assistant Professor Dr. Leah Shaw, The Applied Science Department  
The College of William and Mary

---

---

## ABSTRACT PAGE

In this dissertation, we present two parallelized 3D simulation techniques for three-dimensional acoustic and elastic wave propagation based on the finite integration technique. We demonstrate their usefulness in solving real-world problems with examples in the three very different areas of nondestructive evaluation, medical imaging, and security screening. More precisely, these include concealed weapons detection, periodontal ultrasography, and guided wave inspection of complex piping systems. We have employed these simulation methods to study complex wave phenomena and to develop and test a variety of signal processing and hardware configurations. Simulation results are compared to experimental measurements to confirm the accuracy of the parallel simulation methods.

## Table of Contents

<b>1. Introduction</b>	<b>1</b>
1.1 – Introduction	1
1.2 - Acoustic and Elastic Wave Simulation Methods	3
1.2.1 - The Finite Integration Technique	3
1.2.2 - Finite Difference Method	5
1.2.3 - Boundary Element Method	6
1.3 - 2D and 2.5D Simulation Examples	8
1.3.1 - 2D Elastic Block	8
1.3.2 - 2D Lamb Waves	12
1.3.3 - Axial-Symmetric (2.5D) Cylindrical Acoustic Simulations	15
1.4 - Need for 3D Computer Simulation Methods	18
1.5 – References	19
 <b>2. Parallel Three-Dimensional Acoustic Finite Integration Technique (3DPAFIT)</b>	 <b>21</b>
2.1 - Three-Dimensional Acoustic Finite Integration Technique	22
2.1.1 - Derivation of the Discrete Simulation Equations	22
2.1.2 - Derivation of the Discrete Continuity Equation	23
2.1.3 - Derivation of the Discrete Equation of Motion	26
2.1.4 - Stability Criteria	28
2.2 - Simple Scattering Examples and 3D Visualization	29
2.2.1 - Three-Dimensional Visualization	29
2.2.2 Acoustic Scattering from Simple-Shape Objects	30
2.3 Parallel Acoustic Finite Integration Technique	33
2.4 Inhomogeneous Simulation Spaces	36
2.4.1 Discretization of Material Parameters	36
2.4.2 Sample Acoustic Interaction with Objects of Different Materials	37
2.4.3 Simulations including Material Layers	39
2.5 3DPAFIT Conclusions	41
2.6 References	42
 <b>3. Experimental Verification of 3DPAFIT</b>	 <b>43</b>
3.1 - Experimental Apparatus	43
3.2 - Acoustic Back Scatter Experiments	46
3.2.1 - Scattering from two 18” Cylinders	47
3.3 - Signal Processing	50
3.3.1 - The Short Time Fourier Transform (Spectrogram)	50
3.3.2 - Feature Extraction	53
3.3.3 - Feature Extraction using Pulse Compression	58
3.4 - Comparison to 3DPAFIT Simulations	63
3.5 - Conclusions	67
3.6 - References	68

<b>4. Applied 3DPAFIT Simulations:</b>	
<b>Nonlinear Acoustic Concealed Weapons Detection</b>	69
4.1 - Nonlinear Concealed Weapons Detection	69
4.2 - KZK Nonlinear Sound Beam Simulations	71
4.2.1 - Linear vs. Nonlinear Parametric Simulations	72
4.2.2 - Comparison of Simulation and Experimental Results	77
4.2.3 - Comparison of Confocal and Parametric Transducer Configurations	79
4.2.3.1 - Parametric Transducer Array Configuration	80
4.2.3.2 - Confocal Transducer Configuration	83
2.2.3.3 - Comparison of the Two Transducer Configurations	88
4.2.3.4 - Amplitude Modulated Beat Frequency – Vibro-Acoustography	94
4.2.3.5 - Confocal vs. Parametric Array Conclusion	95
4.2.4 - Effects of the Initial Sound Pressure Intensity on the Resulting Sound Beam	95
4.2.5 - Extended Distance Simulations	100
4.2.6 - Nonlinear Sound Beam Discussion	103
4.3 - 3DPAFIT Simulations with Incident Nonlinear Sound Beam	105
4.3.1 - Nonlinear Sound Beam Input	105
4.3.2 - Inserting and Scattering from a Human Model	105
4.3.3 - Inserting CAD Models	110
4.3.4 - Back Scattered Acoustic Energy as a Function of Angle	112
4.4 - Conclusion	116
4.5 – References	117
 <b>5. Applied 3DPAFIT Simulations: Ultrasonic Periodontal Probe</b>	118
5.1 – Introduction	118
5.1.1 - The Ultrasonic Periodontal Probe	119
5.2 - Acoustic Simulations of the Ultrasonic Periodontal Probe	123
5.3 - Two-Dimensional Periodontal Tissue and Tip Geometry	125
5.3.1 - Adaptable 2D Periodontal Tissue Model	127
5.3.2 - Adding the Periodontal Pocket	128
5.3.3 - Two Dimensional Tip Construction and Placement	129
5.4 - Three Dimensional Periodontal Pocket and Ultrasonic Tip Geometry	131
5.5 - Example Simulation Output and Visualization	135
5.6 - Rigid Simulation and Experimental Results	139
5.7 - Periodontal Tissue Material Parameters	142
5.8 - 10 MHz Ultrasound Beam Study	143
5.8 - Conclusions	149
5.9 - References	149

<b>6. 3D Parallel Cylindrical Elastic Finite Integration Technique (3DPCEFIT).</b>	<b>153</b>
6.1 - Introduction .....	153
6.2 - 3D Cylindrical Elastic Finite Integration Technique (3DCEFIT) .....	154
6.3 - Finite Integration Procedure .....	154
6.4 - Modifications for Pipe Bends .....	163
6.5 - Stability Criteria .....	166
6.6 - Boundary Conditions .....	168
6.7 - Absorbing Boundary Layers .....	169
6.8 - Parallel Implementation .....	170
6.9 - Conclusions .....	172
6.10 - References .....	173
 <b>7. Applied 3DPCEFIT Simulations: Guided Waves in Complex Piping Geometries</b>	
7.1 - Introduction .....	174
7.2 - Comparison with Experimental Data .....	176
7.3 - Guided Wave Scattering From Flaws .....	178
7.4 - Focusing Techniques .....	180
7.4.1 - Focusing with Hardware: Phased Array Transducer Belts .....	180
7.4.2 - Focusing in Software: The Synthetic Aperture Focusing Technique (SAFT) .....	183
7.5 - Pipe Bend Simulations .....	186
7.6 - 3DCEFIT Conclusions .....	190
7.7 - References .....	191
 <b>8. Conclusions</b> .....	<b>192</b>
 <b>A1. Appendix 1 - Determining Parameters of the Nonlinear KZK Simulation Code</b> .....	<b>196</b>
 <b>A2. Appendix 2 – Simulation Source Code</b> .....	<b>208</b>

## **Acknowledgements**

I would like to thank the many people who helped make this research possible and successful. This includes Kevin Leonard, Jill Bingham, Jonathan Stevens, Wen Gao, Crystal Bertoncini, Brian Walsh, and Alison Pouch for all there assistance and helpful discussion. I would also like to thank Tom Crockett and Chris Bording for all their technical help regarding the SciClone.

Most importantly, I would like to sincerely thank my advisor Dr. Hinders for his all of his support. I am grateful for all the knowledge I have gained from his teachings, advice, and example.

## **Dedication**

To my wife Meghan for all her love and support.



# Chapter I

## Introduction

### 1.1 Introduction

This dissertation focuses on three-dimensional parallel computer simulation methods for simulating acoustic and elastic waves in realistic geometries. We present two three-dimensional parallel simulation methods and demonstrate their usefulness with three very different applications in nondestructive evaluation. These applications range from locating corrosive damage in complex piping systems to detecting concealed weapons using nonlinear acoustics. We utilize the 3D parallel simulation methods to develop and test experimental hardware configurations and signal processing algorithms.

Computer or numerical simulations are based on mathematical models of physical laws. They are used to perform complex numerical experiments to solve problems that are difficult or impossible to solve using traditional mathematical methods. We will focus on the finite integration technique (FIT) for simulating acoustic and elastic waves, but we will also use a finite difference method for simulating nonlinear acoustic waves in Chapter 4. The different techniques for simulating acoustic and elastic waves will be presented in the next section. At the end of this introduction, we will present examples of 2D acoustic and elastic wave simulations and then discuss both their usefulness and limitations.

Today's desktop computers do not have the computational resources to run full 3D realistic simulations. The simulation methods presented in this dissertation have been designed to distribute the computational demands across many computers networked together to form a supercomputer. Each computer works on its own part of the simulation space and stays in constant communications with the other computers to perform large 3D simulations. Each computer works in parallel with the others (hence the term parallel in the names of the simulation method).

In Chapter 2, we present the 3D parallel acoustic finite integration technique (3DPAFIT) for simulating acoustic waves with large and realistic geometries. We validate this simulation method with a novel experiment that is presented in Chapter 3. In this experiment, we use an air-coupled nonlinear sound beam to study the acoustic back scatter from multiple objects. We then demonstrate the usefulness of this simulation method with two very-different applied examples. These are the development of a nonlinear acoustic concealed weapons detector in Chapter 4 and an ultrasonic periodontal probe in Chapter 5.

A 3D parallel cylindrical finite integration technique (3DPCEFIT) for simulating elastic waves in complex piping geometries is presented in Chapter 6. This simulation method introduces a coordinate transform to handle elastic waves in pipe bends. In Chapter 7, the 3DPCEFIT method is validated by direct comparisons to experimental data and simulation results from a commercial finite-element package. Several simulation scenarios are presented including guided elastic wave interaction with corrosive-like flaws in pipes and hardware and software guided wave focusing techniques.

## **1.2. Acoustic and Elastic Wave Simulation Methods**

There are many simulation methods for modeling acoustic and elastic waves. All of them begin with a set of partial differential equations that govern the wave mechanics. Some simulation methods begin with a wave equation while others begin with conservation laws. The specific details of the discretization and solution methods are where the simulation methods differ.

### **1.2.1 The Finite Integration Technique**

Since this dissertation mainly focuses on the finite integration technique, we will review its theory and the relevant work of other researchers in detail. The finite integration technique has been successfully used to model acoustic, elastic, and electromagnetic waves [1]. The finite-integration method is similar to the finite-difference method but has some important distinctions. The finite-difference method directly approximates the derivatives of a differential equation, where as the finite-integration technique first integrates the differential equation over a control volume and then approximates the integrals. In simple cases, both methods can yield the same discrete versions of a given differential equation. The advantage of the finite integration method is that it leads directly to a staggered grid formulation that is more stable and accurate [2, 3]. It also provides a simpler method of handling boundary conditions as well as the discretization of inhomogeneous and anisotropic materials [3].

The finite integration technique has been used successfully to simulate acoustic and elastic waves for a variety of applications. Fellingner et al. first introduced the Elastodynamic Finite Integration Technique (EFIT) for simulating elastic waves in 2D

and 3D Cartesian coordinates [2]. They start with Cauchy's equation of motion and the deformation rate equation and integrate them over a cube (or square for 2D). In 3D, the discretization leads to a staggered grid of 12 variables: the three components of the velocity vector, the six components of the stress tensor, and the three material components (density and the two Lamé constants). Fellingner also includes a very detailed derivation of the stability criteria that is valid for the finite integration techniques introduced in this dissertation. At the end of this chapter, we show two example simulations using the 2DEFIT method. The first simulates an impulse on the top of a steel block, and the second simulates Lamb waves in an aluminum plate.

Peiffer et. al. developed the Cylindrical Acoustic Finite Integration Method CAFIT for simulating acoustic waves in axial symmetric cylindrical coordinates [4]. This allowed for the simulation of limited 3D scenarios using a 2D simulation space. They begin with the linear equation of continuity and of motion. These equations are integrated over a 3D "piece-of-cake" control volume. It is assumed that the acoustic waves are axial symmetric, so this integration reduces to two dimensions: axial ( $z$ ) and radius ( $r$ ). The discretization leads to a staggered grid of only three variables, which are the two velocity components and the acoustic pressure. They validate this simulation method against two well known test cases. The first is a spherical expanding pressure wave and the second is a piston in an infinite baffle. For both cases, the CAFIT simulation was very accurate. At the end of this chapter we will show sample CAFIT simulations that were used to study ultrasound scattering from air and fat bubbles in the blood stream.

F. Schubert et al. developed the cylindrical elastodynamics finite integration technique for simulating elastic waves in axial-symmetric cylindrical coordinates (CEFIT) [3]. Schubert starts with Hooke's law and the equation of motion and discretizes these equations in a similar manner to Fellingner's and Peiffer's methods. Schubert also includes a detailed derivation on handling inhomogeneous materials and various boundary conditions including a plane-wave boundary condition. In a later paper, Schubert demonstrates how the EFIT technique can be used to simulate a variety of applications including inspection of concrete and acoustic emission problems [5]. Also, in this same paper, he uses the EFIT technique to simulate a one-dimensional nonlinear elastic wave. The EFIT technique has also been shown to model dissipative and anisotropic materials [6], air-coupled ultrasound [7], and applied to various NDE techniques [8].

### **1.2.2 Finite Difference Method**

The finite difference method is one of the most common and widely used simulation methods. It has been used to simulate and model a wide variety of processes in acoustics, elastodynamics, thermodynamics, electromagnetics, and even used in fields such as financial modeling. The finite difference method is a relatively simple method. The derivatives of a differential equation are directly approximated with a finite difference. This transforms a differential equation into an algebraic "difference" equation. Given both the initial and boundary conditions, the difference equation is then solved to yield an approximate solution to the original differential equation. The most notable cases that are related to the research in this dissertation will be mentioned here.

A majority of the advances in elastic wave propagation methods utilizing the finite-difference method have come from the geophysics community. Kelly et al. developed a 2D finite difference simulation method based on second-order elastic equations [9, 10]. J. Vireux introduced a finite difference method for simulating shear horizontal (SH) waves [11] and pressure and shear vertical (P-SV) waves [12]. He begins with the basic elastic equations and uses a staggered grid similar to the one used in the finite integration method. Since then, many researchers have used the finite-difference method to study elastic waves including sonic logging [13, 14]. Bohlen introduced a 3D parallel finite difference method that uses a domain decomposition scheme similar to the one used in this dissertation [15].

### **1.2.3 Boundary Element Method**

The Boundary Element Method (BEM) is another popular technique for studying acoustic problems. It was first introduced by Chen and Schweikert [16] in 1963. The boundary element method directly solves the Helmholtz equation in either a bounded interior domain or an unbounded exterior domain [17]. One of the main advantages of the boundary element method is that only the boundary of the domain and interfaces need to be discretized and not the entire simulation space. This is important when realistic 3D volumes were impossible to simulate with even the largest computers. In many cases, this reduces the complexity and computational time of setting up a simulation. The BEM may not be the best simulation method when the geometry of the problem is complicated or when there are many material boundaries.

The boundary element method is typically used to study steady state problems such as radiation from an arbitrary vibrating source or scattering from an object with a continuous source. Early research has been carried out for time-dependent problems [18, 19]. In addition, the boundary element has also been successfully used to study elastodynamics problems [20]. For an overview of the Time-Domain Boundary Element Method, refer to the chapter 8 of reference [17].

### **1.3 - 2D and 2.5D Simulation Examples**

Simulations are very useful tools for exploring and visualizing elastic and acoustic wave propagation and interaction. Two-dimensional simulations are now relatively easy to program in a high level numerical software package such as MATLAB. We here present some examples of two-dimensional simulations and show they are useful for visualizing and learning about the different types of waves, their propagation, interaction, and complex mode conversions. First we present a 2D elastic simulation method and demonstrate its usefulness for exploring and learning about elastic wave propagation. Then we present an applied example of how we can use the finite integration technique to characterize air-bubbles in the blood stream.

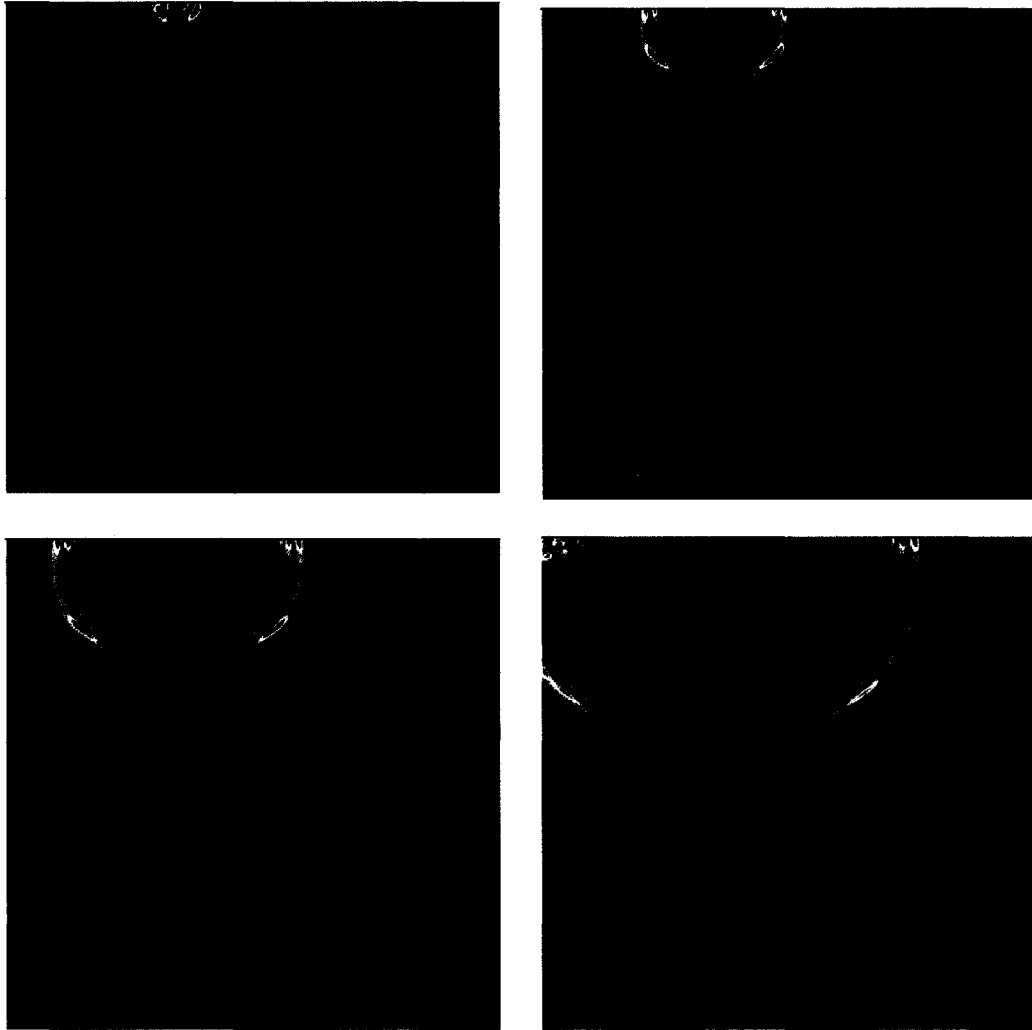
#### **1.3.1 - 2D Elastic Block**

Elastic waves are much more complicated phenomena than acoustic waves. There are three types of waves that can co-exist in a solid: longitudinal bulk (compression), transverse bulk (shear), and surface-guided waves. When these waves interact with an interface, material variation, edge, inclusion, or flaw, a portion of the wave energy will mode-convert to another type of wave. For example, a longitudinal wave obliquely incident on a boundary between two solid materials will result in at least four outgoing waves: a transmitted and reflected longitudinal wave and a transmitted and reflected transverse wave.

Figure 1.1 shows four time snapshots of a 2D elastic wave simulation. An impulse (a half-cycle 800 kHz wave) excites the top of a 7 cm square steel block. The colors of these plots are proportional to the absolute velocity of the material. Radiating



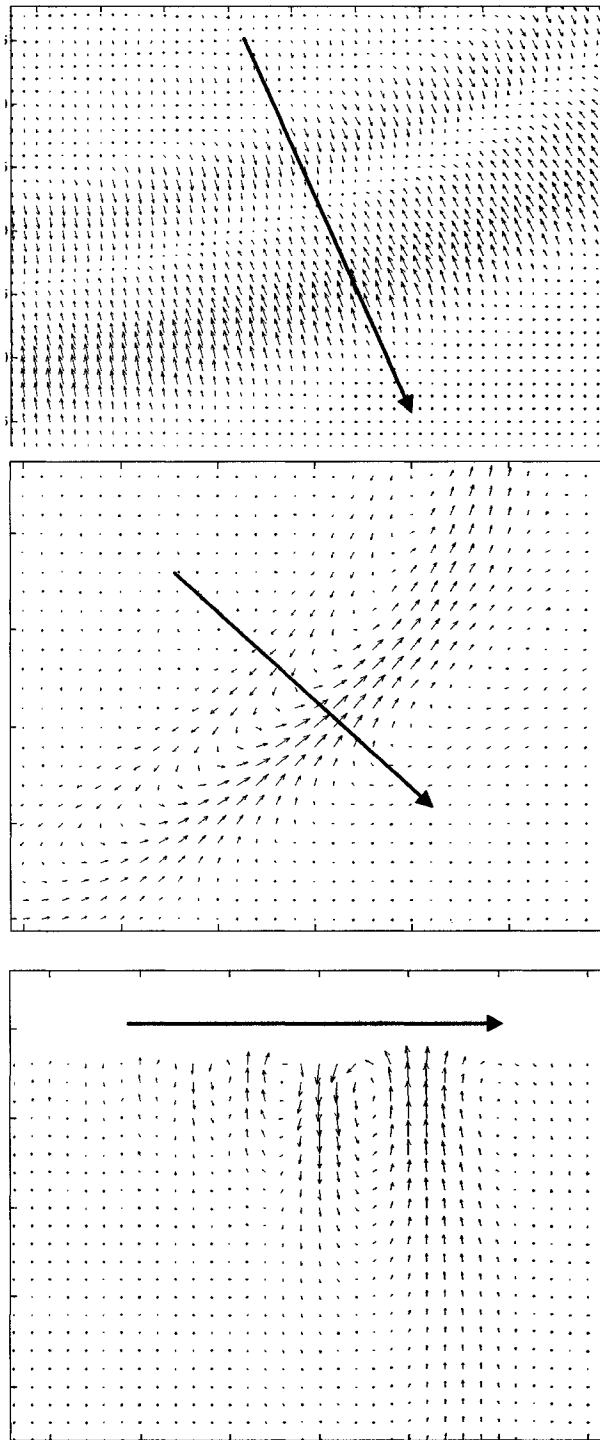
from the point of impact are the three different elastic wave types. Just as expected, the longitudinal wave is the fastest followed by the shear wave and then the surface wave. As the longitudinal wave reflects from the edges, some of the energy is converted into a transverse wave.



**Figure 1.1.** 2D elastic wave simulation. An impulse excites the top of a 7-cm square block. In the bottom left snapshot, the labeled waves are longitudinal wave (a), transverse wave (b), surface (Rayleigh) wave (c), and a reflected longitudinal wave (d). The shear wave (e) is the result of a mode-conversion from the reflected longitudinal wave on the left edge.

Looking at figure 1.1, it is difficult to distinguish between the different wave types. One of the advantages of simulations is the ability to stop time and to examine complex phenomena more carefully. Another advantage is the ability to look inside a material that is often impossible to do experimentally. Instead of a false-color plot of the absolute velocity, figure 1.2 shows three quiver plots of three areas inside the steel block. A quiver plot is a set of arrow or vectors indicating the discrete velocity of the material.

The velocity profile of the longitudinal, transverse, and surface wave are all shown in Figure 1.2. The large black arrow indicates the direction the wave is propagating. In these plots it is easy to see the difference in the longitudinal and transverse wave. With a longitudinal or compression wave, the local material velocity components are parallel to the direction of the propagating wave. With a transverse or shear wave, the local material velocity component is perpendicular to the direction of the propagating wave. The surface or Rayleigh wave has an elliptical velocity profile.

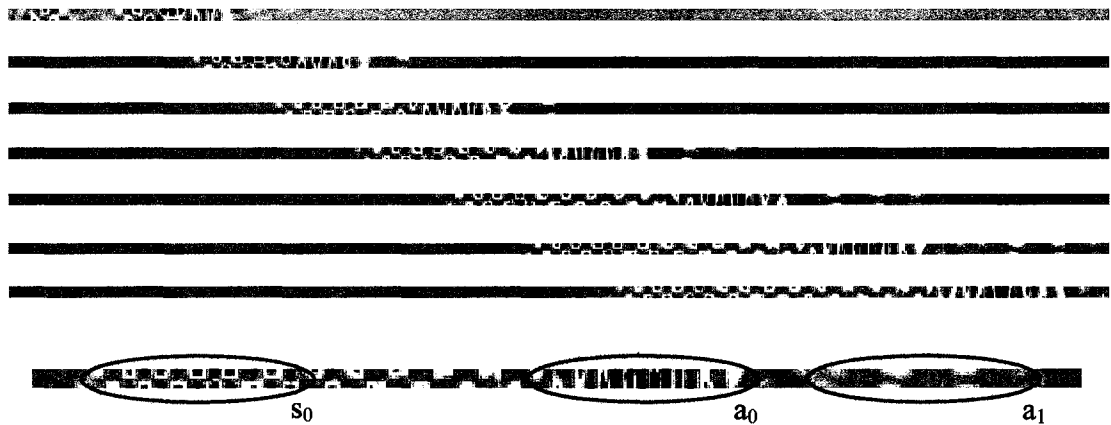


**Figure 1.2.** Velocity profiles of a longitudinal (top), shear (middle), and surface (bottom) waves. The large black arrow indicates the direction of the wave propagation.

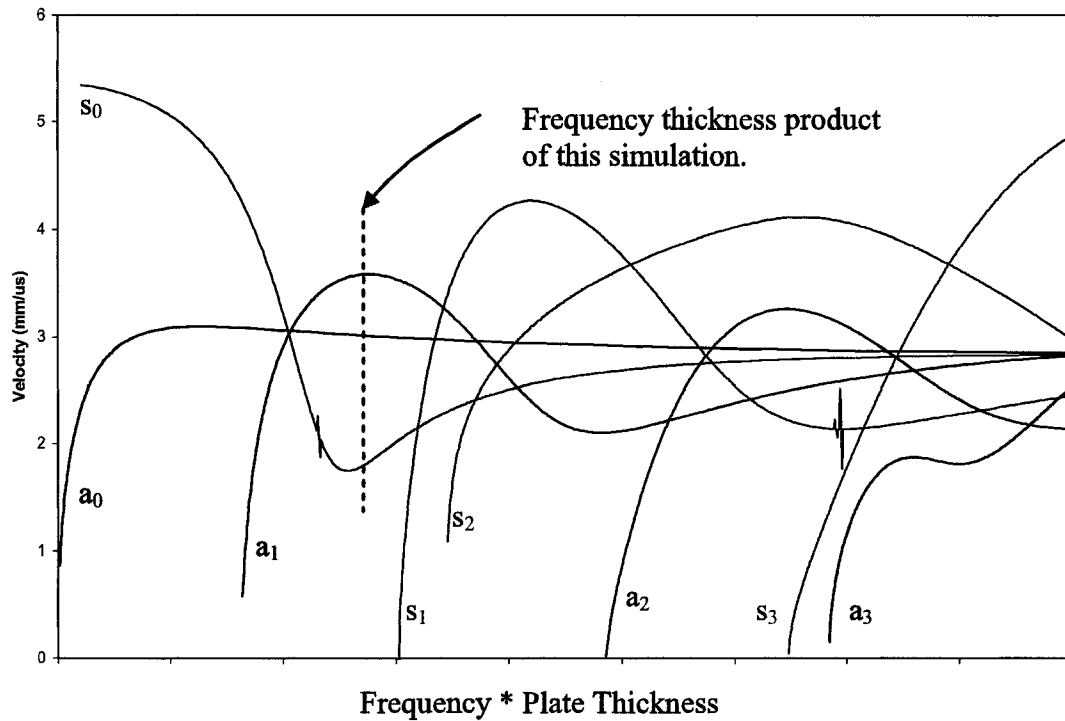
### 1.3.2 - 2D Lamb Waves

Lamb waves are routinely used to nondestructively evaluate thin solid materials such as sheet metal, piping, and composites. In chapter 6, we present a 3D cylindrical simulation method for modeling Lamb waves in complex piping systems. Lamb waves are guided elastic wave modes that form after multiple reflections and mode conversions from the top and bottom surface of a thin plate. There are two types of Lamb wave modes: symmetric and anti-symmetric. The quantity and group velocity of these modes is dependent on the thickness of the plate and frequency of the initial excitation.

Figure 1.3 shows snapshots of a 2D elastic wave simulation of a 3.2 mm thick aluminum plate. The color of these snapshots is proportional to the velocity of the plate in the  $x_1$ -direction (out of plane). A 1mm transducer excites the top of the plate with a five cycle 680 kHz tone burst. At first, the lamb wave modes overlap. As they propagate down the plate, the three Lamb wave modes separate because they have different group velocities. The bottom snapshot of figure 1.3 identifies the three modes.



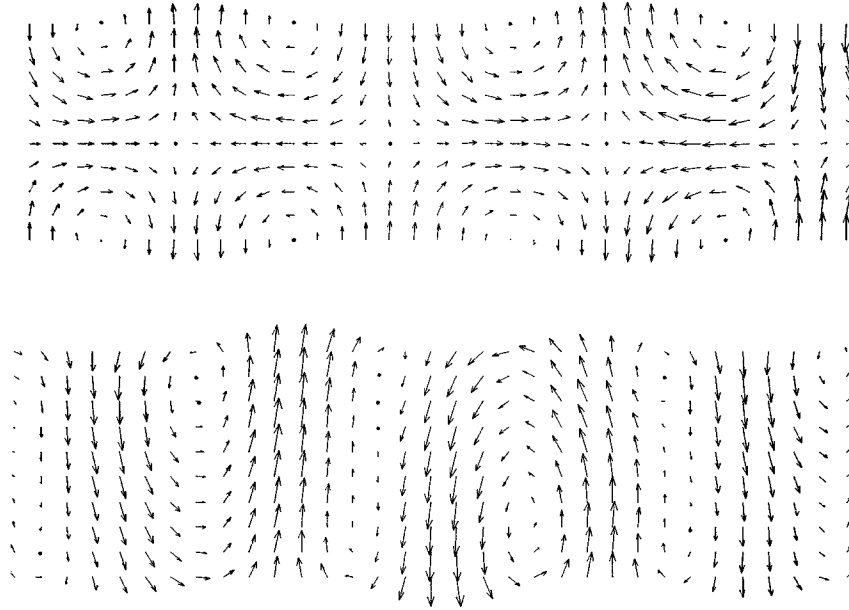
**Figure 1.3.** 2D elastic wave simulation of Lamb waves in a 3.2 mm thick aluminum plate. At 680 kHz, three Lamb wave modes form and propagate at different velocities.



**Figure 1.4.** A dispersion curve showing the group velocities of the lamb waves as a function of the frequency plate thickness product. The dotted line indicates the Lamb wave velocities for the simulation shown in figure X-3.

Figure 1.4 shows the Lamb wave dispersion curve for an aluminum plate. A dispersion curve provides the group velocity of the Lamb wave modes as a function of the frequency and plate-thickness product. The dotted line indicates the location along the dispersion curves where this simulation is carried out. It clearly indicates that there should be three modes and in order of velocity they are  $a_1$ ,  $a_0$ , and  $s_0$ . This is exact order we see the lamb waves in the simulation snapshot.

The snapshots in figure 1.3 are very informative, but we can extract more detail from the 2D simulations. Figure 1.5 shows the velocity profile of the symmetric  $s_0$  and anti-symmetric  $a_0$  modes. From these quiver plots, it is easy to see the differences between the two mode types.



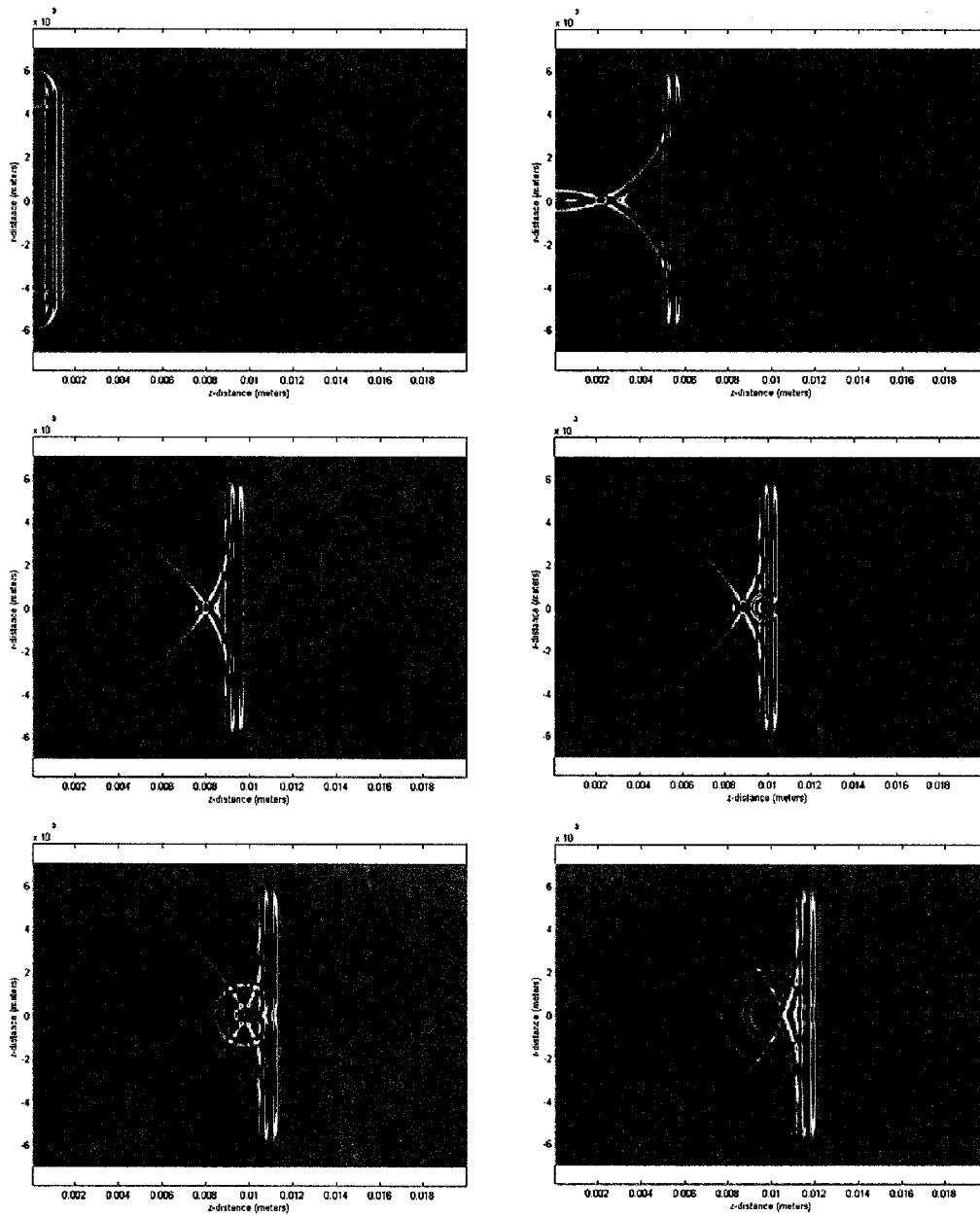
**Figure 1.5.** Velocity profiles of a symmetric  $s_0$  (top) and anti-symmetric  $a_0$  (bottom) Lamb wave modes.

These two-dimensional simulations are a valuable tool for learning about complex elastic wave propagation. It is simple to “experiment” with the different simulation parameters and to observe how they affect the wave propagation. The simulation can be stopped to closely study the fine details of the wave propagation, interaction, and mode conversions.

### **1.3.3. Axial-Symmetric (2.5D) Cylindrical Acoustic Simulations**

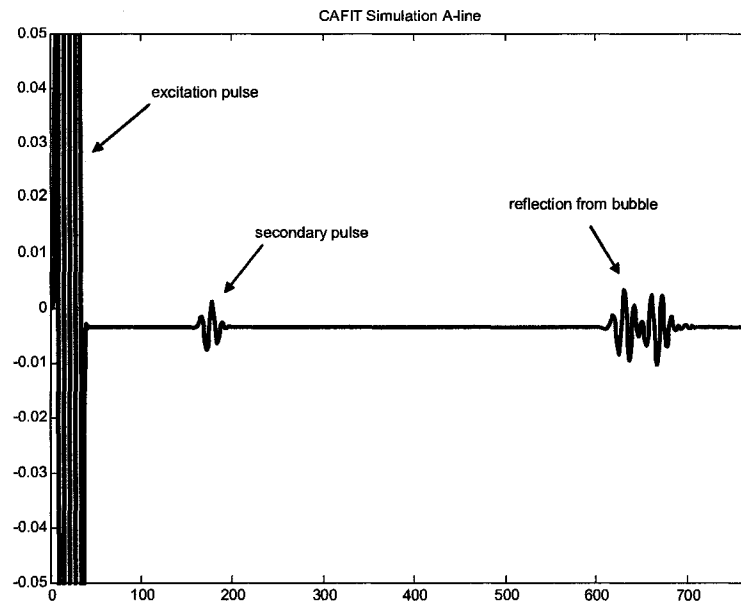
Now we present an applied example of a cylindrical acoustic finite integration simulation method. In this study, the frequency-domain and time-domain analysis of ultrasound scattering by fluid spheres is applied to emboli classification. An embolus (pl. emboli) refers to a microbubble, generally of gas or lipid composition, that flows through the bloodstream. Presenting a significant health hazard, these emboli may occlude blood vessels and thereby prevent the flow of blood to surrounding tissue and vital organs [21]. The cylindrical acoustic finite integration technique was used to model the ultrasound scattering in order to evaluate the inverse problem of determining the size and composition of individual spherical scatterers.

Figure 1.6 shows several snapshots of an acoustic wave interacting with a small air bubble in blood. A typical pulse-echo A-line is shown in Figure 1.7. The back scattered acoustic wave from the simulation is compared directly to a theoretical back scatter model [22]. Figure 1.8 shows the two signals plotted together showing good agreement. This example shows that 2D and 2.5D simulations can be useful for solving problems of simple geometries.

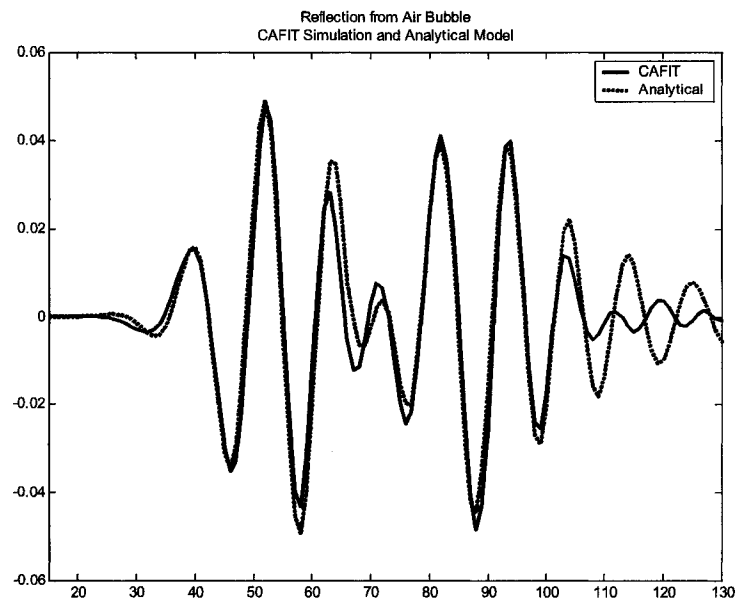


**Figure 1.6.** Snapshots from a CAFIT simulation showing the ultrasound scattering from air bubble in blood.





**Figure 1.7.** An A-line produced by the CAFIT simulation for a 300 $\mu\text{m}$  air bubble in blood.



**Figure 1.8.** Time-domain reflection from air in blood. The CAFIT curve is shown in blue, and the analytical comparison is presented in black.

#### **1.4. Need for 3D Computer Simulation Methods**

In the previous section, we demonstrated the pedagogical value of 2D simulations and also presented a case where they can solve simple real-world problems. In most situations, the geometries are too complex to generalize into a 2D problem. The ability to simulate full 3D wave propagation and interaction in realistic geometries is the goal of this dissertation. The major limitation to solving full 3D simulations is the computational resources required. For example, to simulate a 330 kHz acoustic wave in a two-dimensional one meter box will require roughly 32 megabytes of computer memory. To simulate the same wave in a three-dimensional one meter cube of air will require roughly 40 gigabytes of computer memory. This amount of memory is not available on today's desktop PCs.

To solve this problem, we have developed the simulation software to run on a parallel super computer. In Chapter 2 and Chapter 6, we present two full 3D parallel simulation methods for simulating acoustic and elastic wave propagation. We present three applications in nondestructive evaluation with complex geometries that can only be solved with 3D simulations. We employ these simulation methods to help understand the complex physics of full 3D wave propagation and scattering. These simulation methods are also very useful at designing and virtual prototyping hardware configurations and signal processing algorithms.

## 1.5 References

1. Marklein, R., *Chapter 11*, in *Review of Radio Science*, W.R. Stone, Editor. 2002, Press and John Wiley and Sons.
2. P. Fellingner, R.M., K.J. Langenberg, and S. Klaholz, *Numerical modeling of elastic wave propagation and scattering with EFIT - elastodynamic finite integration technique*. *Wave Motion*, 1995. **21**: p. 47-66.
3. F. Schubert, A.P., and B. Kohler, *The elastodynamic finite integration technique for waves of cylindrical geometries*. *Journal of the Acoustical Society of America*, 1998. **104**(5): p. 2604-2614.
4. A. Peiffer, B.K., and S. Petzold, *The acoustic finite integration technique for waves of cylindrical symmetry (CAFIT)*. *Journal of the Acoustical Society of America*, 1997. **102**(2): p. 697-706.
5. Schubert, F., *Numerical time-domain modeling of linear and nonlinear ultrasonic wave propagation using finite integration techniques--theory and applications*. *Ultrasonics*, 2004. **42**(42): p. 221-229.
6. R. Marklein, R.B., K.J. Langenberg, *The Ultrasonic Modeling Code EFIT as Applied to Inhomogeneous Dissipative Isotropic and Anisotropic Media*. *Review of Progress in Quantitative Nondestructive Evaluation*, 1995. **14**: p. 251-258.
7. M. Rudolph, P.F., K.J. Langenberg, D.E. Chimenti, *Numerical Modeling of Air-Coupled Ultrasound with EFIT*. *Review of Progress in Quantitative Nondestructive Evaluation*, 1995. **14**: p. 1053-1060.
8. R. Marklein, K.J.L., S. Klaholz, J. Kostka, *Ultrasonic Modeling Real-Life NDT Situations: Applications and Further Developments*. *Review of Progress in Quantitative Nondestructive Evaluation*, 1996. **15**(57-64).
9. K.R. Kelly, R.W.W., S. Treitel, R.M. Alford, *Synthetic Seismograms: a finite difference approach*. *Geophysics*, 1976. **41**(1): p. 2-27.
10. Kelly, K.R., *Numerical study of Love wave propagation*. *Geophysics*, 1983. **48**(7): p. 833-853.
11. Vireux, J., *P-SV wave propagation in heterogenous media: Velocity stress finite-difference method*. *Geophysics*, 1986. **51**: p. 889-901.
12. Vireux, J., *SH-wave propagation in heterogenous media: Velocity stress finite-difference method*. *Geophysics*, 1984. **49**(1933-1957).
13. Q.H. Liu, E.S., F. Daube, C. Randall, H.L. Liu, and P. Lee, *Large-scale 3D finite-difference simulation of elastic wave propagation in borehole environments*. *Journal of the Acoustical Society of America*, 1994. **94**: p. 3337.
14. Q.H. Liu, E.S., F. Daube, C. Randall, H. L. Liu, and P. Lee, *A three-dimensional finite difference simulation of sonic logging*. *Journal of the Acoustical Society of America*, 1996. **100**: p. 72-79.
15. H. Dong, A.M.K., C. Madshus, and J. M. Hovem, *Sound propagation over layered poro-elastic ground using a finite-difference model*. *Journal of the Acoustical Society of America*, 2000. **108**: p. 494-502.
16. L.H. Chen, D.G.S., *Sound Radiation from an arbitrary body*. *Journal of the Acoustical Society of America*, 1963. **35**: p. 1626-1632.
17. Wu, T.W., *Boundary Element Acoustics: Fundamentals and Computer Codes*. 2000: WIT Press.

18. W.J. Mansur, C.A.B., *Formulation of the boundary element method for transient problems governed by the scalar wave equation*. **Appl. Math. Modelling**, 1982. **6**: p. 307-311.
19. W.J. Mansur, C.A.B., *Numerical Implementation of the boundary element method for two-dimensional transient scalar wave propagation problems*. **Appl. Math. Modelling**, 1982. **6**: p. 299-306.
20. Mansur, W.J., *A Time Stepping Technique to Solve Wave Propagation Problems Using the Boundary Element Method*. 1987, Southhamton University: Southhampton, England.
21. Lynch, *FILL IN TED's EMBOLI SIZING PAPER*.
22. V.C. Anderson, *Sound Scattering from a Fluid Sphere*. **Journal of Acoustical Society of America**, 1950. **22**: p. 426-431.

## Chapter II

### Parallel Three-Dimensional Acoustic Finite Integration Technique

Simulating three-dimensional acoustic waves is an important and very challenging task. Our world is full of complicated shaped objects that are impossible to describe using two-dimensional models. For example, modeling sonar scattering from a submarine, medical ultrasound imaging of a fetus, or acoustic emissions from an automobile engine all require a three-dimensional acoustic computer simulation. In this chapter, we will describe a three-dimensional parallel acoustic simulation method (3DPAFIT) based on the finite integration technique. The derivations of the difference equations will be presented along with the stability criteria.

At the writing this dissertation, the computer power is not yet available to run useful three-dimensional simulations on a single desktop computer. So, we will present a parallel version of the three-dimensional acoustic simulation technique. The simulation method is validated by comparing scattering results to experimental measurements. A method of visualizing the 3D acoustic waves will also be presented. This simulation method is validated by comparing directly to experimental results in Chapter 3. We then use the 3DPAFIT simulations to assist in the development of Nonlinear Acoustic Concealed Weapons Detector and an Ultrasonic Periodontal Probe in chapters 4 and 5.

## 2.1 Three-Dimensional Acoustic Finite Integration Technique

The finite integration technique has been used successfully to simulate acoustic and elastic waves in varying coordinate systems [1-3]. We have used the finite integration method to solve the basic acoustic equations in the three-dimensional Cartesian coordinate system.

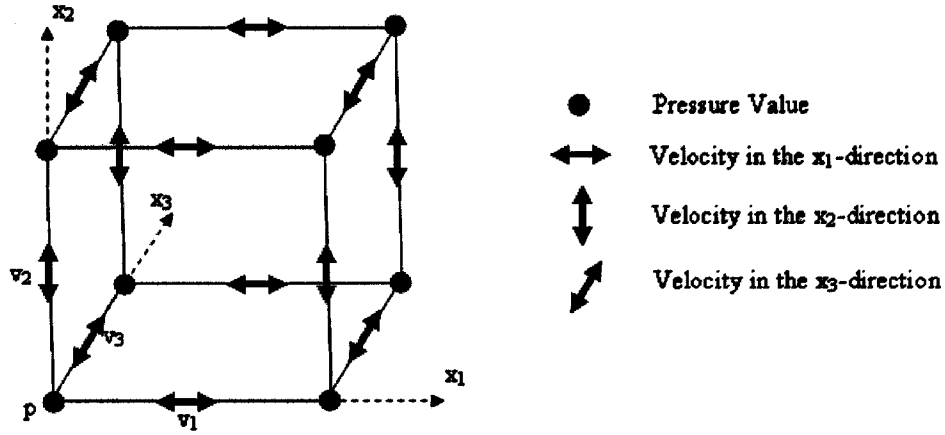
### 2.1.1 Derivation of the Discrete Simulation Equations

Here, we will describe how to derive the difference equations that are necessary to simulate three-dimensional acoustic waves. First, we begin with the equation of continuity (2.1) and the equation of motion (2.2) [4].

$$\frac{\partial p}{\partial t} + \rho_0 c_0^2 \nabla \cdot \vec{v} = M \quad (2.1)$$

$$\rho_0 \frac{\partial \vec{v}}{\partial t} + \nabla p = \vec{F} \quad (2.2)$$

The unknown variables are pressure  $p$  and the three components of the velocity vector  $\vec{v}$ . The remaining known variables are time  $t$ , density  $\rho_0$ , speed of sound  $c_0$ , and the pressure and velocity source functions  $M$  and  $\vec{F}$ . To discretize these equations, we use the finite-integration technique (FIT). If we were to use the finite-difference technique, we would approximate the derivatives in (2.1) and (2.2) directly. But, with the finite integration technique, we integrate the partial differential equations over a control volume and then approximate the integrals. This leads to a staggered grid in space and time that is more stable and accurate than a straight forward finite-difference method.



**Figure 2.1.** Staggered grid used in the 3D Acoustic Finite Integration Technique

Additional steps would be required to introduce this staggered grid in a finite difference method [2]. Figure 2.1 shows the staggered grid used in the 3D acoustic finite integration technique.

### 2.1.2 Derivation of the Discrete Continuity Equation

We begin by integrating both sides of continuity equation (2.1) over a cube control volume  $V$ .

$$\iiint_V \dot{p} dV = \iiint_V (-\rho_0 c_0^2 \nabla \cdot \vec{v} + M) dV \quad (2.3)$$

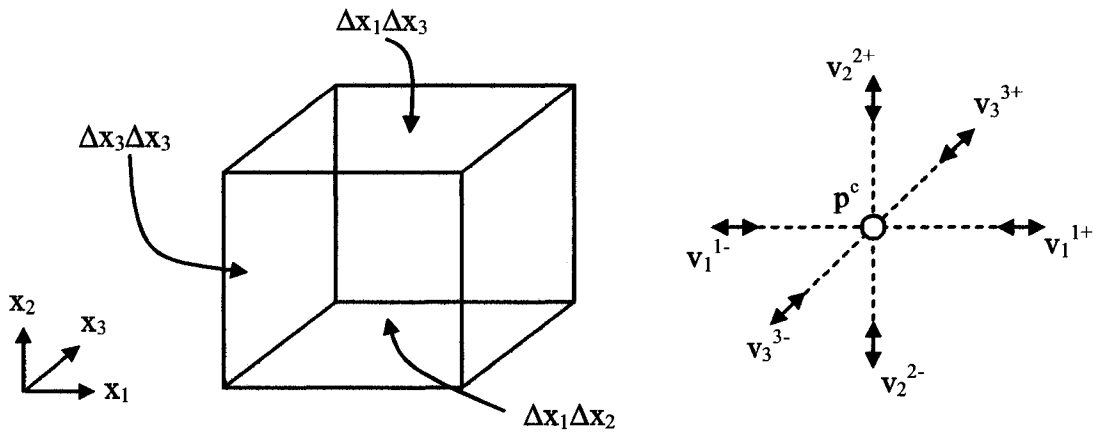
Now, we use the Divergence Theorem (2.4), also called the Gauss's Theorem, which states that in the absence of the sources, the density within a region of space can only change when mass flows into or out of the region through its boundary [5].

$$\iiint_V \nabla \cdot \vec{v} dV = \oint_{\partial V} \vec{v} \cdot d\vec{a} \quad (2.4)$$

Here  $a$  is the surface of volume  $V$ . We use the divergence theorem to replace the volume integrals of the velocity components on the right hand side with surface flux integrals.

$$\iiint_V \dot{p} dV = -\rho_0 c_0^2 \oint_{\partial V} \vec{v} \cdot d\vec{a} + \iiint_V M dV \quad (E2.5)$$

Now we approximate all the integrals by multiplying the integrands by the volume or surface of the integrals. This is easier done by visualizing the control volume with the variable distribution. The cube control volume and the variable distribution are shown in figure 2.2. We place the pressure value at the center of the control volume and the velocity components on the center of the faces. We introduce superscripts that indicate the position of a given variable relative to the center of the cube. For example, the superscript (2-) indicates that the variable is in the negative  $x_2$  direction from the center



**Figure 2.2.** Control volume and the variable distribution for the computation of the Discrete Continuity Equation.



of the cube.

When approximating the surface-flux integral only the velocity components that are perpendicular to a given surface contribute to that term. So, we approximate the surface-flux integral in the following way.

$$\oint_{\partial V} \vec{v} \cdot d\vec{a} \approx (v_1^{1+} - v_1^{1-})\Delta x_2 \Delta x_3 + (v_2^{2+} - v_2^{2-})\Delta x_1 \Delta x_3 + (v_3^{3+} - v_3^{3-})\Delta x_1 \Delta x_2 \quad (2.6)$$

We simplify our cube by setting all three sides equal ( $\Delta x = \Delta x_1 = \Delta x_2 = \Delta x_3$ ). The volume integrals are more straight-forward to approximate leading us to the following equation.

$$\dot{p} \Delta x^3 = -\rho_0 c_0^2 [(v_1^{1+} - v_1^{1-})\Delta x^2 + (v_2^{2+} - v_2^{2-})\Delta x^2 + (v_3^{3+} - v_3^{3-})\Delta x^2] + M \Delta x^3 \quad (2.7)$$

$$\dot{p} = -\frac{\rho_0 c_0^2}{\Delta x} [(v_1^{1+} - v_1^{1-}) + (v_2^{2+} - v_2^{2-}) + (v_3^{3+} - v_3^{3-})] + M \quad (2.8)$$

Now we use a standard central difference to replace the time derivative on the left hand side. The standard central-difference is given by the following equation.

$$\dot{p} \approx \frac{(p^{t+\Delta t/2} - p^{t-\Delta t/2})}{\Delta t} \quad (2.9)$$

Where  $\Delta t$  is the time step that will be defined in the following section. Using this, we arrive at our final discrete equation.

$$p^{t+\Delta t/2} = p^{t-\Delta t/2} - \frac{\rho_0 c_0^2 \Delta t}{\Delta x} [(v_1^{1+} - v_1^{1-}) + (v_2^{2+} - v_2^{2-}) + (v_3^{3+} - v_3^{3-})] + M \Delta t \quad (2.10)$$

This equation tells us what the pressure value at time  $(t + \Delta t / 2)$  is based on the surrounding velocity values and the pressure value at the past time step.

### 2.1.3 Derivation of the Discrete Equation of Motion

The Equation of Motion will tell us how to update our velocity values as our simulation marches in time. The equation of motion can be split to individually look at the three components of the velocity vector. Here, we will find the discrete equation of motion in the  $x_1$  direction.

$$\rho_0 \frac{\partial v_1}{\partial t} + \frac{dp}{dx_1} = F_1 \quad (2.11)$$

We integrate both sides over the cube control volume.

$$\iiint_V \rho_0 \dot{v}_1 dV = \iiint_V \left( -\frac{dp}{dx_1} + F_1 \right) dV \quad (2.12)$$

Again, we use the Divergence Theorem to transform the volume integral of the pressure value on the right hand side to a surface-flux integral.

$$\iiint_V \rho_0 \dot{v}_1 dV = - \oint_{\partial V} p \cdot da + \iiint_V F_1 dV \quad (2.13)$$

Next, we approximate the integrals and let  $\Delta x = \Delta x_1 = \Delta x_2 = \Delta x_3$ .

$$\rho_0 \dot{v}_1 \Delta x^3 = -((p^{1+} - p^{1-}) \Delta x_2 \Delta x_3) + F_1 \Delta x^3 \quad (2.14)$$

$$\dot{v}_1 = -\frac{1}{\rho_0 \Delta x} (p^{1+} - p^{1-}) + \frac{F_1}{\rho_0} \quad (2.15)$$

Then we approximate the time derivative using a central time difference.

$$v_1^t = v_1^{t-\Delta t} - \frac{\Delta t}{\rho_0 \Delta x} (p^{1+} - p^{1-}) + \frac{\Delta t F_1}{\rho_0} \quad (2.16)$$

The same method is used to arrive at the difference equations for the velocity components in the  $x_2$  and  $x_3$  directions.

$$v_2^t = v_2^{t-\Delta t} - \frac{\Delta t}{\rho_0 \Delta x} (p^{2+} - p^{2-}) + \frac{\Delta t F_2}{\rho_0} \quad (2.17)$$

$$v_3^t = v_3^{t-\Delta t} - \frac{\Delta t}{\rho_0 \Delta x} (p^{3+} - p^{3-}) + \frac{\Delta t F_3}{\rho_0} \quad (2.18)$$

Now we have four equations (2.10) and (2.16) to (2.18) that instruct us how to update our simulation grid based on the surrounding values. The time steps are also staggered in

time. First the pressure values are updated. Then a half time step later, the velocity values are updated.

#### 2.1.4 Stability Criteria

Stability conditions are necessary to achieve stable and accurate results. We begin by defining our spatial step size  $\Delta x$ . This value is determined by assigning at least 8 grid points to the shortest wavelength  $\lambda_{\min}$  in the simulation [2]. Peifer et al. choose to assign 15 grid points to the shortest wavelength in axial-symmetric acoustic version of the finite integration technique [1]. This was done to ensure accuracy but is not necessary. For our simulations, we assign 10 grid points to the shortest wavelength.

$$\Delta x \approx \frac{\lambda_{\min}}{10} \quad (2.19)$$

This stability criterion can also be found using the maximum speed of sound in the simulation space  $c_{\max}$  and the maximum frequency  $f_{\max}$  of the acoustic waves present in the simulation.

$$\Delta x \approx \frac{c_{\max}}{10 f_{\max}} \quad (2.20)$$

The temporal time step  $\Delta t$  is found using the standard Courant condition [2]. In three dimensions, this is given by the following equation.

$$\Delta t \leq \frac{\Delta x}{c_{\max} \sqrt{3}} \quad (2.21)$$

## 2.2 Simple Scattering Examples and 3D Visualization

Results from three different simulations are presented here to illustrate scattering from three-dimensional objects and the method used to visualize the three-dimensional acoustic waves.

### 2.2.1 Three-Dimensional Visualization

One of the major complications of three-dimensional simulations is visualizing the wave propagation and interaction. Visualization is important because it provides insight into the complex physics of acoustic interactions. The visualizations also provide a way to represent a large amount of data in a format that is easily understandable by a large audience: pictures and movies.

We use the MATLAB programming environment to display 2D and 3D images of acoustic wave propagation. Two-Dimensional images can be made by taking slices through the 3D simulation space and plotting those values in a 2D color plot. An example of these 2D plots can be seen in figure 2.7.

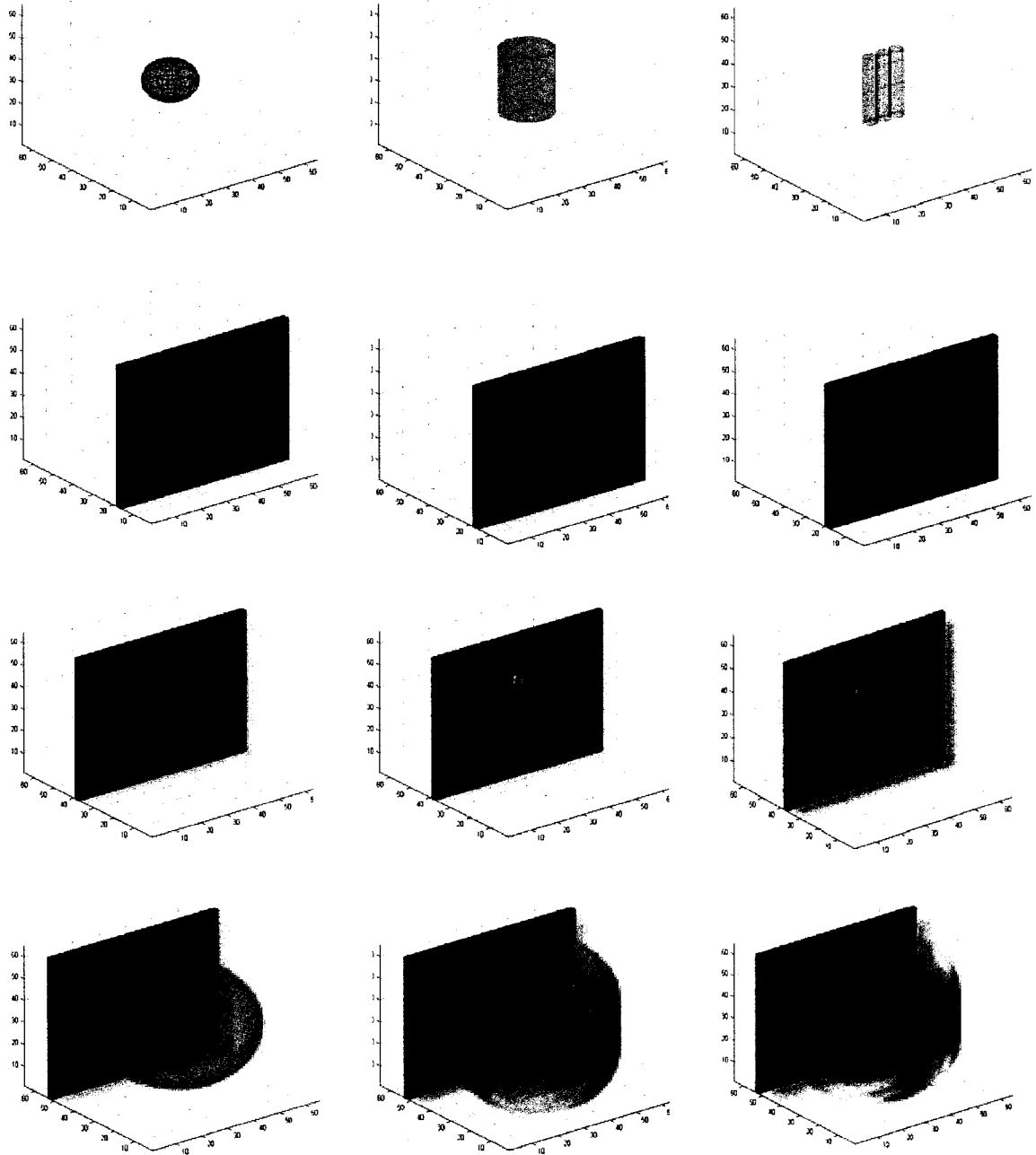
3D images can be made by taking many 2D slices through the simulation space. These 2D slices are taken in all three planes in equal increments. The color and transparency of these 2D slices are proportional to the pressure values. This creates a cloud-like 3D image of acoustic waves. An example of these 3D images can be found in figure 2.3.

### 2.2.2 Acoustic Scattering from Simple-Shape Objects

Three sample simulations are presented here to demonstrate the capabilities of the simulation method and provide examples of the 3D visualization. The first two simulations consist of a short 10 kHz acoustic wave incident upon a single 2" diameter sphere and a single 2" diameter by 4" tall cylinder, respectively. We have also included results from the same 10 kHz wave incident upon three 1" diameter by 6" tall cylinders. Figure 2.3 shows the geometry of the objects along with the three-dimensional pressure fields at three time instances. It's important to note that three-dimensional pressure fields are very computationally demanding to simulate, but the 3DAFIT technique on a large parallel computer allows us to handle arbitrary geometries and perform systematic parameter variations.

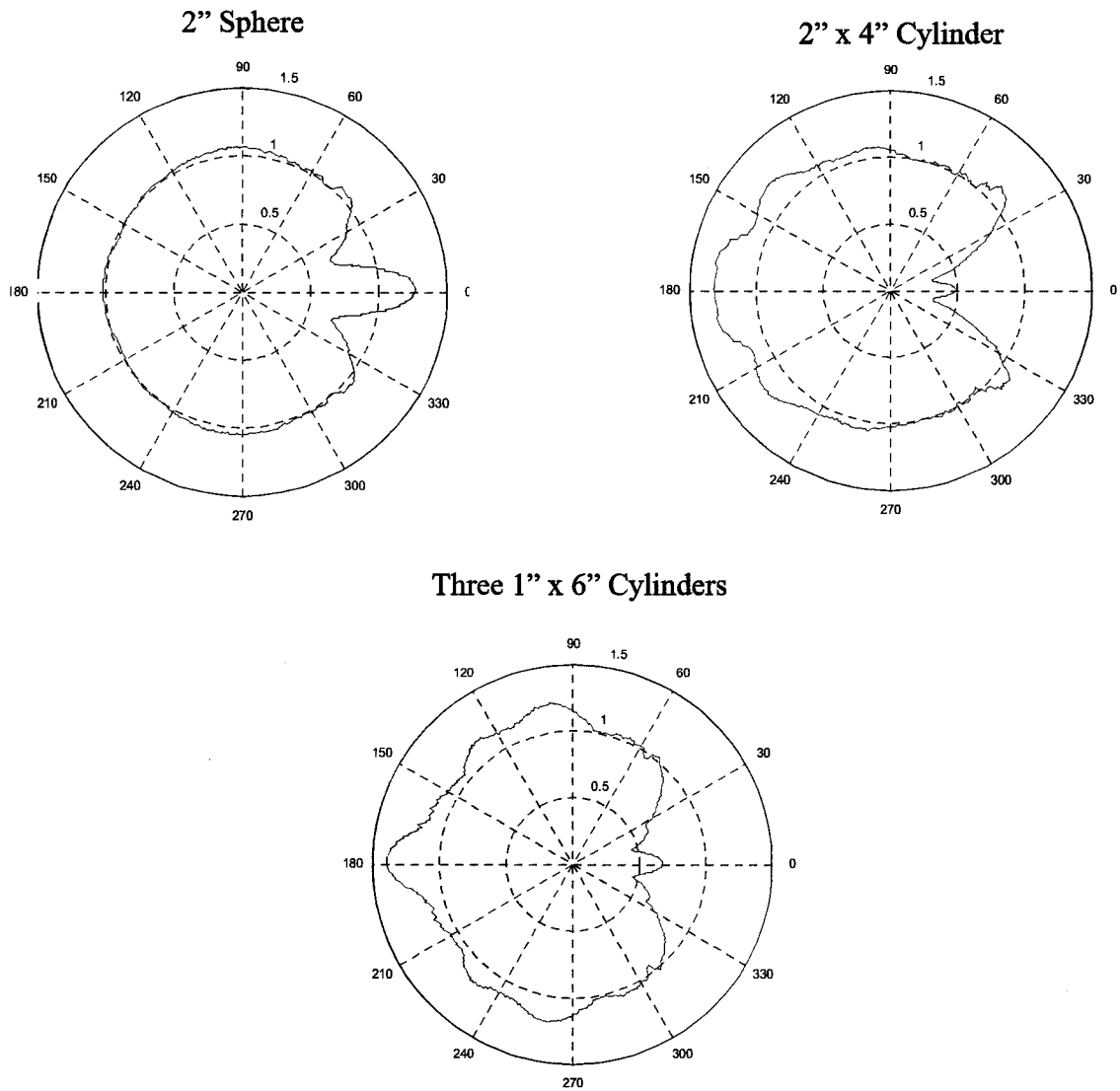
The most important use of the simulations is to provide insight to complicated acoustic problems. In scattering simulations, it is useful to find the scattered acoustic field. Figure 2.4 shows the scattered acoustic energy as a function of angle for the three simulations. These values were recorded in the horizontal plane intersecting the center of the scattering object. The value at  $180^\circ$  corresponds to the acoustic energy reflected back to the source, where the signal would be recorded in a monostatic measurement scheme.

## Visualization of Three-Dimensional Acoustic Scattering



**Figure 2.3.** Three-dimensional acoustic fields are shown at three time instances in the three rows above. The top row is just before the plane wave interacts with the target. The scatterers are a 2" diameter sphere (left column), a 2" by 4" cylinder (middle column), and three 1" by 6" cylinders (right column). The incident wave is a single cycle 10 kHz wave.

## Acoustic Scattering Polar Plots



**Figure 2.4.** Acoustic polar scattering plots for the three example simulations.

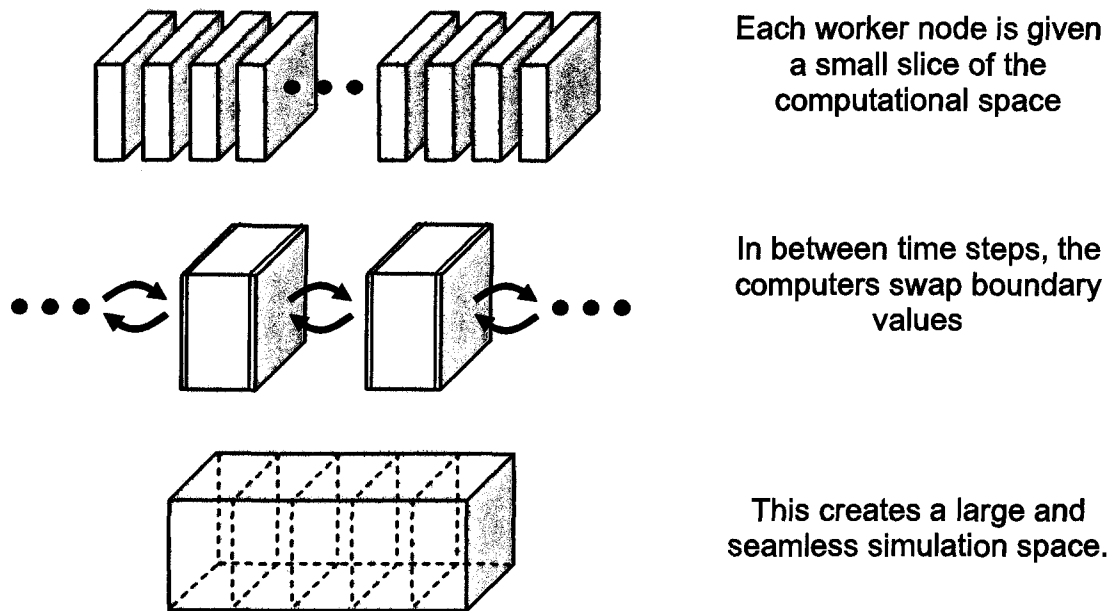


### **2.3 Parallel Acoustic Finite Integration Technique**

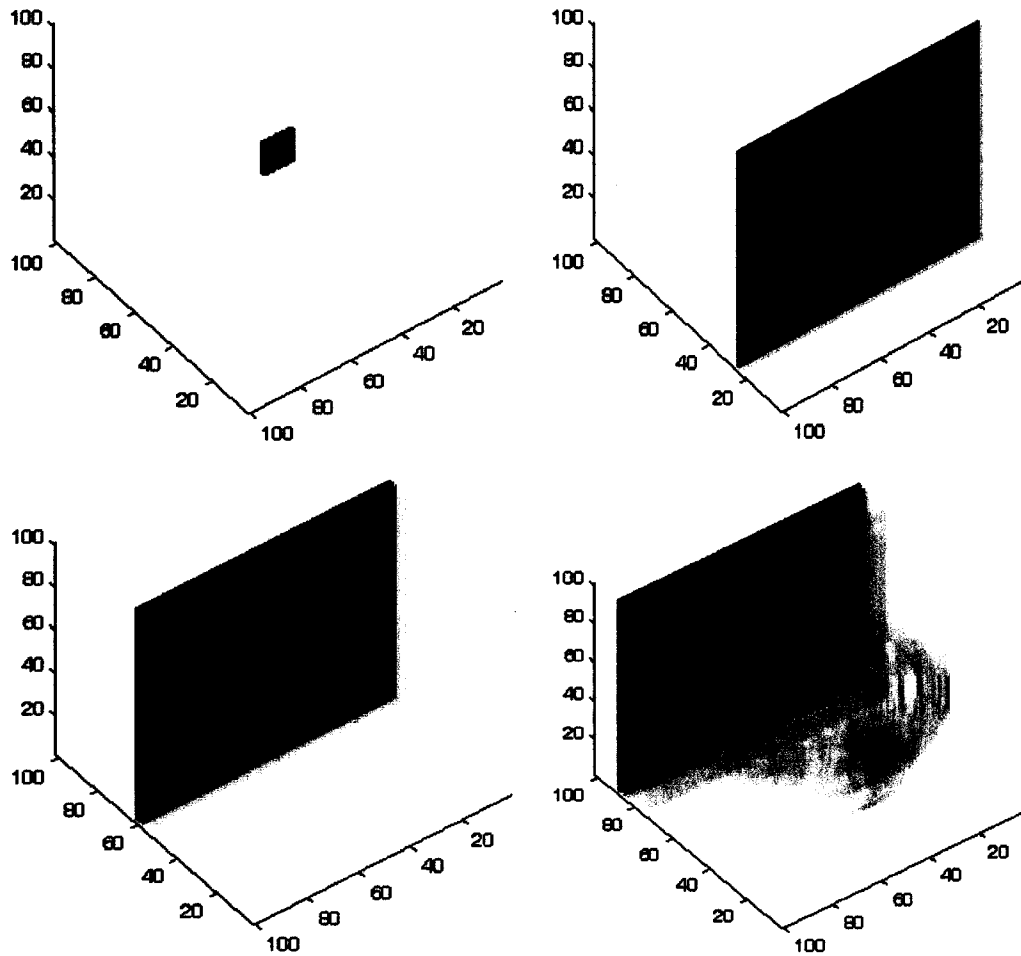
To run realistic 3D scenarios, our acoustic simulation code had to be parallelized. This was accomplished by dividing the simulation space into many 3D slices and distributing them across many computers. Each computer then treats an individual slice as a separate simulation space. After each time step, neighboring computers trade boundary values to create a large and seamless simulation space. Figure 2.5 illustrates the simulation space decomposition. Similar decomposition methods can be found in the following references [6, 7].

Figure 2.6 shows several snapshots from a large simulation that was performed using the parallelized acoustic simulation code running on the SciClone. The simulation space is a cube with each edge measuring a meter long. Five 1" by 6" cylindrical objects are located at the center of the simulation space. As a 10 kHz wave passes through the cube it scatters from the five cylinders. During this simulation, each of the 256 million unknown variables were computed at each of the 700 time steps. This required 2.6 gigabytes of computer memory and produced 1.3 terabytes of data over the entire simulation.

This same scenario was simulated using a much larger simulation space measuring 2m on each edge. This simulation had 2 billion unknown variables that required 16 gigabytes of computer memory. The simulation produced 22 terabytes of data over 1400 time steps. It is not obvious how to best visualize datasets this large. Simulations this large (and larger) were performed in Chapter 5 for the ultrasonic periodontal probe. In that application, we only visualize two-dimensional pressure slices because of the large memory requirements of 3D visualizations.



**Figure 2.5.** This figure demonstrates how the computational simulation space is divided across many computers and recombined to form a large and seamless computational space.



**Figure 2.6.** Snapshots from a simulation showing a 10 kHz wave scattering off five 1" by 6" cylinders. These simulation results were obtained with the new parallelized acoustic simulation code running on the SciClone.

## 2.4 Inhomogeneous Simulation Spaces

The 3D Parallel Acoustic Finite Integration Technique as described above did not allow for objects of different material compositions. In the derivation of the difference equations, the material parameters (speed of sound  $c_0$  and density  $\rho_0$ ) were treated as continuous variables. Objects placed into the simulation space were considered rigid such that no acoustic energy penetrated them.

### 2.4.1 Discretization of Material Parameters

To include objects of different materials, we re-derived the difference equations with the material parameters also discretized in space. We began with the equation of continuity (2.22) and the equation of motion (2.23) where the material parameters have not been factored out.

$$\frac{\partial p}{\partial t} = -\rho_0 c_0^2 \nabla \cdot \bar{v} + M \quad (2.22)$$

$$\frac{\partial \bar{v}}{\partial t} = -\left(\frac{1}{\rho_0}\right) \nabla p + \bar{F} \quad (2.23)$$

We then transform these partial difference equations using the Finite Integration Technique just as before to reveal the following difference equations.

$$p = p - \frac{\rho_0 c_0^2 \Delta t}{\Delta x} \left( (v_1^{1+} - v_1^{1-}) + (v_2^{2+} - v_2^{2-}) + (v_3^{3+} - v_3^{3-}) \right) + M \Delta t \quad (2.24)$$

$$v_1 = v_1 - \frac{2\Delta t}{(\rho_0^{1+} + \rho_0^{1-})\Delta x} (p^{1+} - p^{1-}) + \frac{F_1\Delta t}{\rho_0} \quad (2.25)$$

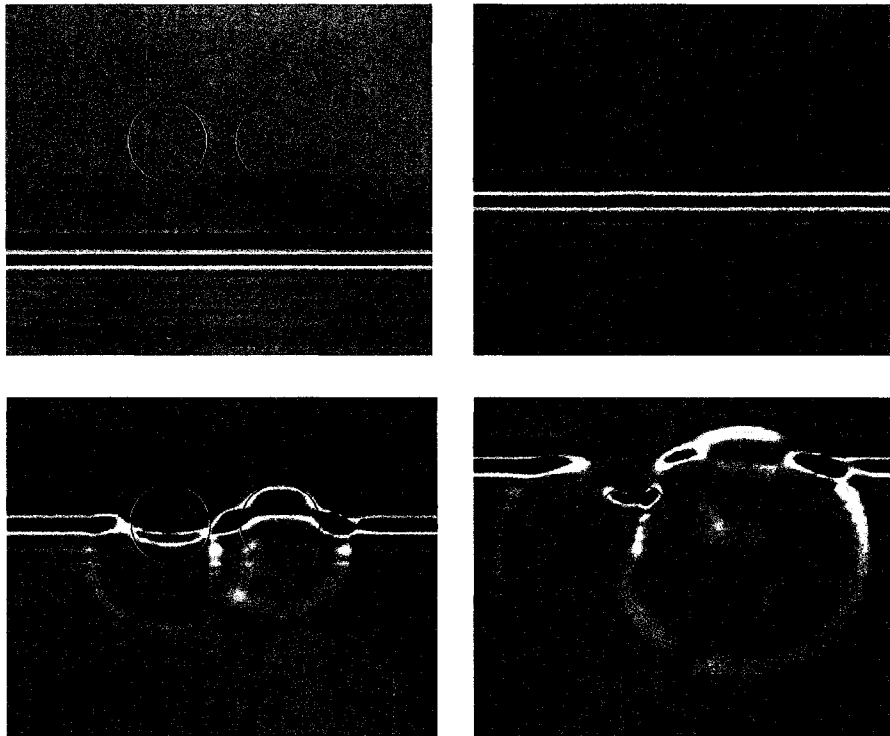
$$v_2 = v_2 - \frac{2\Delta t}{(\rho_0^{2+} + \rho_0^{2-})\Delta x} (p^{2+} - p^{2-}) + \frac{F_2\Delta t}{\rho_0} \quad (2.26)$$

$$v_3 = v_3 - \frac{2\Delta t}{(\rho_0^{3+} + \rho_0^{3-})\Delta x} (p^{3+} - p^{3-}) + \frac{F_3\Delta t}{\rho_0} \quad (2.27)$$

The spatial distribution of the simulation variables is done just as before with pressure  $p$  and velocity  $\bar{v}$  distributed on a staggered grid. The material parameters  $c_0$  and  $\rho_0$  are now discretized and spatially aligned with the pressure values. This places the velocity values on the boundary between any two materials. The new difference equations enforce that the normal of material displacement be continuous across any material boundary.

#### 2.4.2 Sample Acoustic Interaction with Objects of Different Materials

With these new difference equations and the appropriate boundary and stability criteria, we can now simulate acoustic interactions with objects and collections of objects of any material type. Figure 2.7 shows 2D pressure slices through a 3D acoustic simulation. There are two cylinders placed in the middle of the simulation space of different material types. The cylinder on the left is less dense and has a smaller acoustic wave velocity than the rest of the propagation medium. The cylinder on the right is denser and has a larger acoustic wave velocity than the rest of the propagation medium. The difference in wave speed is clearly seen as a 10 kHz acoustic wave passes through the two cylinders.

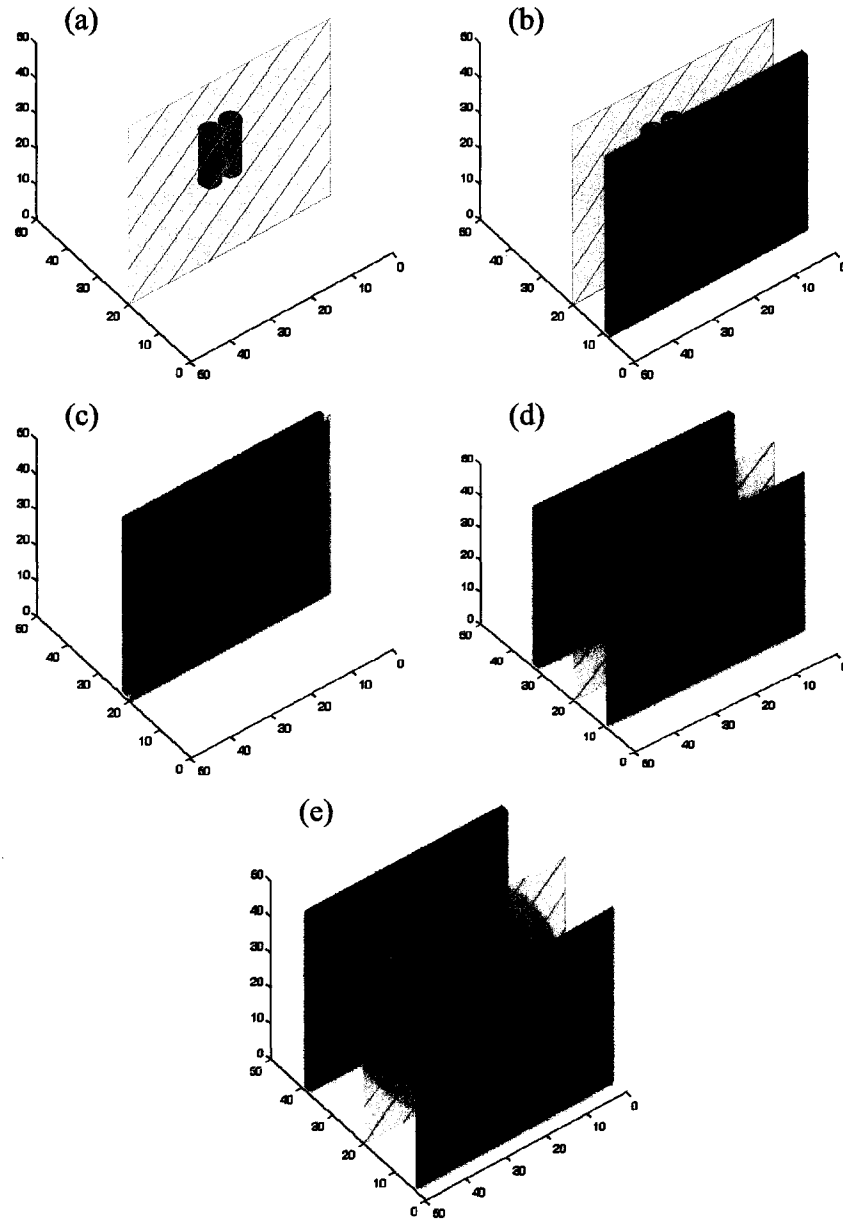


**Figure 2.7.** 2D pressure snapshots from a 3D acoustic simulation. Two cylinders of different material compositions are placed in the middle of the simulation space. A 10 kHz passes through the two cylinders differently because of the difference in material composition.

### **2.4.3 Simulations including Material Layers**

There are many physical scenarios where acoustic waves interact with objects and thin material layers. For example, in medical ultrasound, acoustic waves may penetrate many layers of tissue before reaching the desired location [8]. In other scenarios, the thin layers themselves are being inspected by an acoustic method [9]. To study these types of problems, we next introduce thin material layers into our simulation space.

Material layers are added to the simulation space just as other objects are placed into the simulation space. This is done by adjusting the discrete material parameters in the simulation space. Figure 2.8 shows pressure snapshots from a 3D acoustic simulation of two rigid cylinders behind a thin layer. The thin layer reflects about 20% of the incoming acoustic energy. The acoustic-back-scattered energy from the cylinders and the thin material layer now overlap making the interpretation of the back-scattered data more difficult.



**Figure 2.8.** Pressure snapshots from a 3D acoustic simulation. Two rigid cylinders are placed behind a thin layer. The thin layer allows 80% of the acoustic energy to penetrate. The acoustic backscatter from the cylinders is now overlaps with the backscattered energy from the thin layer which complicates the interpretation of the backscattered data.



## **2.5 3DPAFIT Conclusions**

We have completed a computational framework to systematically study acoustic wave interactions with complex shaped objects and layers. In this chapter we have derived and presented the equations necessary to simulate 3D acoustic waves. Simulating 3D acoustic waves is a very computationally demanding task, so we have also developed a parallel algorithm that allows for very large simulation spaces. A very successful and fairly simple method for visualizing 3D acoustic wave interactions has also been presented.

In addition to studying acoustic wave interactions with rigid objects, we have also developed the necessary simulation equations to create an inhomogeneous simulation space. This allows for the study acoustic wave interactions with objects of varying material parameters and with material layers. Overall, the parallel 3DPAFIT simulation method is a stable, accurate, and cost effective way to study 3D acoustic problems. In the following chapter we validate this simulation method by compare simulation results directly to experimental results.

## 2.6 References

1. A. Peiffer, B.K., and S. Petzold, *The acoustic finite integration technique for waves of cylindrical symmetry (CAFIT)*. Journal of the Acoustical Society of America, 1997. **102**(2): p. 697-706.
2. F. Schubert, A.P., and B. Kohler, *The elastodynamic finite integration technique for waves of cylindrical geometries*. Journal of the Acoustical Society of America, 1998. **104**(5): p. 2604-2614.
3. P. Fellingner, R.M., K.J. Langenberg, and S. Klaholz, *Numerical modeling of elastic wave propagation and scattering with EFIT - elastodynamic finite integration technique*. Wave Motion, 1995. **21**: p. 47-66.
4. Pierce, A.D., *Acoustics: An introduction to its physical principles and application*. 1989: Acoustical Society of America.
5. Kreysig, E., *Advanced Engineering Mathematics*. 1999: John Wiley & Sons.
6. Bohlen, T., *Parallel 3-D viscoelastic finite difference seismic modeling*. Computers & Geosciences, 2002. **28**: p. 887-899.
7. Marklein, R., *Numerical Simulation of Fields and Waves in Nondestructive Testing*. 9th European Conference on Non-Destructive Testing, Berlin, 2006.
8. J. Bushberg, J.S., E. Leidholdt, Jr., J. Boone, *The Essential Physics of Medical Imaging*. 1994: Williams and Wilkins.
9. Krautkrämer, J.K.a.H., *Ultrasonic Testing of Materials*. 4th Edition ed. 1990: Springer-Verlag.

## Chapter III

### Experimental Verification of 3DPAFIT

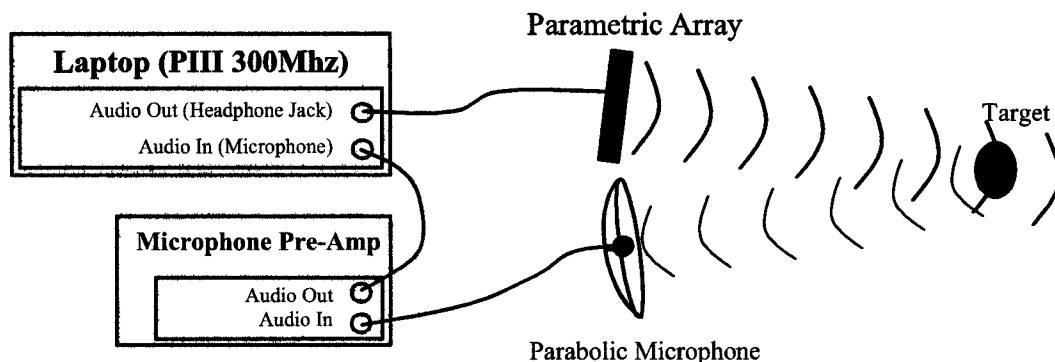
In this chapter, we present an experimental study of acoustic scattering from cylindrical targets. We then use the results of this experimental study to validate the 3D Parallel Acoustic Finite Integration (3DPAFIT) technique described in Chapter 2. Several aspects make this experiment are unique. It includes the use of a parametric array to create a narrow but low-frequency sound beam, the use of a frequency sweep (chirp) excitation signal, and the use of robust signal processing techniques to locate and analyze acoustic backscatter signatures.

#### 3.1. Experimental Apparatus

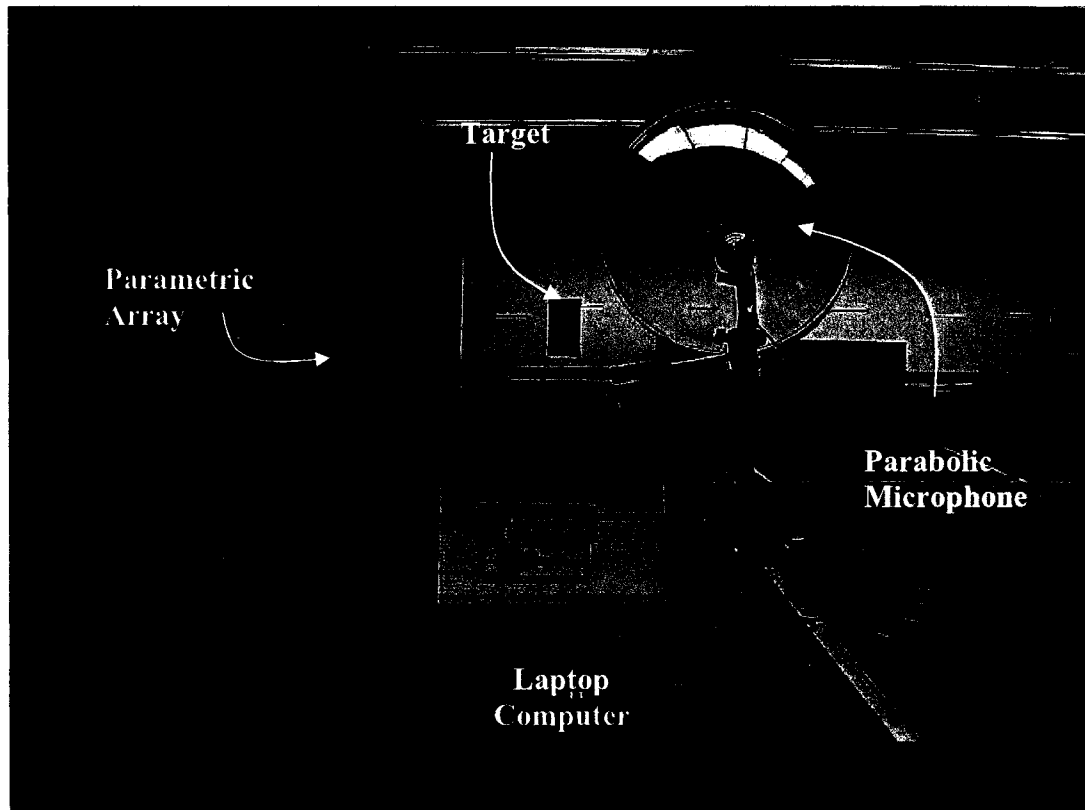
The experimental apparatus is composed of three main components: a parametric array (Sennheiser AudioBeam), a parabolic microphone, and a laptop computer running a simple MATLAB program to control the entire experimental set-up. The experimental apparatus is composed of off-the-shelf components and except for the parametric array all the components are inexpensive. The cost of the parametric array is associated with the complex internal signal conditioning algorithms that are necessary to accurately reproduce music and commentary for a wide variety of audio applications. The transducer array itself is made from very inexpensive elements. A more detailed discussion of parametric arrays can be found in the following chapter.

First, a digital waveform is created in MATLAB and played through the audio out of the laptop computer. This waveform is sent to the parametric array where its internal signal conditioning transforms the signal into a high intensity ultrasound signal. This ultrasound signal is then played through an array of roughly 150 half-inch ultrasound transducers. This creates a high intensity ultrasound waveform that propagates through the air in a very narrow beam. As the waves propagate, they undergo distortions due to the nonlinearity of air. This creates an audio signal that is nearly identical to the original waveform created in the MATLAB environment. This audio signal is confined to a much narrower beam than if it was created with a traditional loudspeaker because the beam width is determined by the effective aperture of the array relative to the wavelength at the 50 kHz ultrasound frequencies.

The resulting sound waves interact with any objects in their path and some of the acoustic energy will scatter back to the experimental apparatus. The returning waves are collected using a parabolic microphone and this signal is passed through a microphone preamplifier. The amplified signal is feed into the laptop via the microphone port and then digitized and stored for analysis. A diagram of the experimental apparatus is shown in figure 3.1. A picture of the experimental apparatus is shown in figure 3.2.



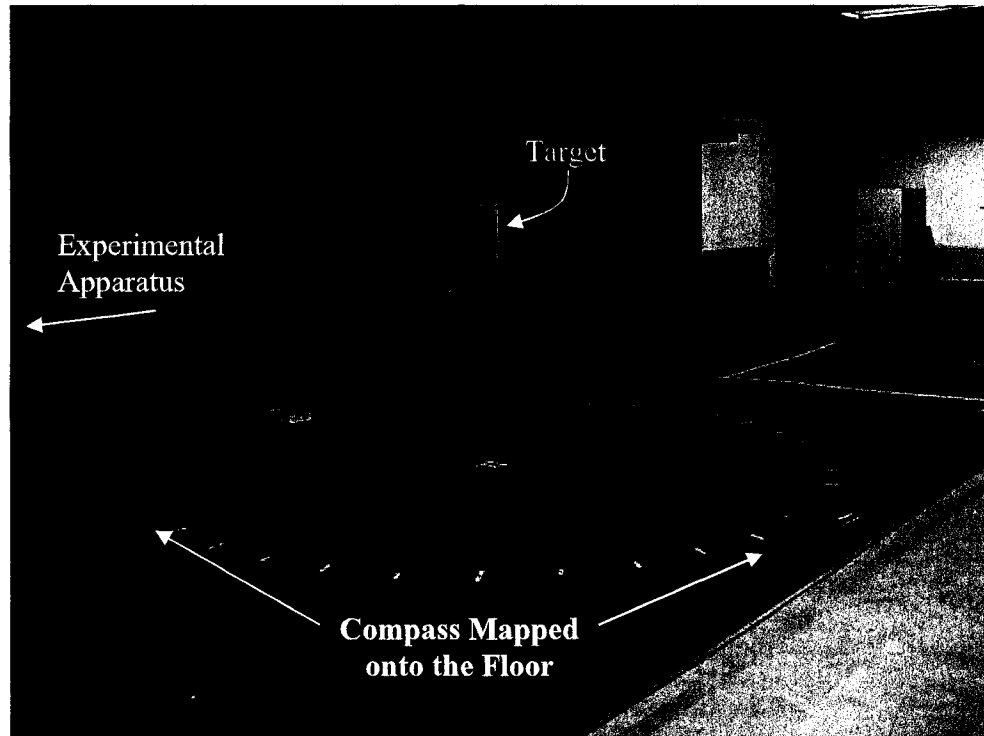
**Figure 3.1.** Diagram showing the experimental apparatus. The main components are the parametric array, parabolic microphone, and the laptop computer.



**Figure 3.2.** Picture of the experimental apparatus. The main components are the parametric array, parabolic microphone, and the laptop computer. A target consisting of five closely packed one foot long pipes with 1" inner diameters are shown in the distance.

### **3.2. Acoustic Back Scatter Experiments**

A scattering experiment was designed to evaluate the capabilities of the experimental system, to explore signal processing techniques, and to validate 3D scattering simulations. Objects were suspended 15 feet in front of the experimental apparatus using thin wire. The parametric array was then excited using various waveforms while the parabolic microphone recorded any returning sounds. The microphone also records any extraneous environmental noises (such as people talking), but the experiments were performed in a large empty room with minimal background noise. The target is then rotated to study how the backscattered acoustic energy is affected as a function of angle of the incident beam. A large compass was mapped out on the floor to provide an accurate indication of angle of the target. Figure [3] shows a picture of a target consisting of five one inch PVC tubes suspended above a large compass.



**Figure 3.3.** Picture of a target suspended above a large compass mapped out onto the floor. The compass is used to accurately measure the angle between the target and the incoming sound beam.

### 3.2.1 Scattering from two 18" Cylinders

In this section, we describe an experiment where the target is composed of two 18 inch PVC pipe (2" inner diameter and 2.375" outer diameter) separated by two inches.

The initial waveform used is a linear frequency-modulated chirp. The chirp waveform  $S$  is created using the following equation.

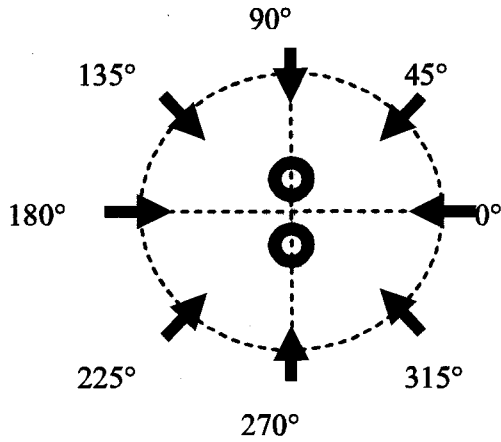
$$S = \sin\left(\left(f_1 + \frac{(f_2 - f_1)t}{2D}\right)2\pi t\right) \quad (3.1)$$

Where the variables are pulse duration  $D$ , time  $t$  ranging from zero to  $D$ , the starting frequency  $f_1$ , and the ending frequency  $f_2$ . In this particular experiment the chirp waveform started at 500Hz and ended at 9400 Hz and was 0.4 seconds long. In air, the acoustic wavelengths of this chirp range from 66cm at 500Hz to 3.5 cm at 9400 Hz. This broad range of acoustic wavelengths is intended to provide a lot of useful information about the scatter since the scattering behavior is strongly frequency dependent. A 0.4 second long sound pulse in air is roughly 132 meters (433 feet) long. This makes resolving individual echoes very difficult but we will show how it can be done later with signal processing.

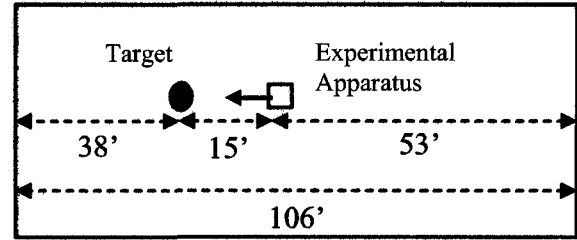
Figure 3.4 shows a top down view of the two PVC pipes with the arrows indicating the direction of the incoming sound beam as the cylinders are rotated. Figure 3.5 shows a rough diagram of the dimensions of the large empty room where the experiments were conducted. The blue arrow in this figure indicates the direction of the initial sound beam. Experimental data was collected from angles ranging from  $-90^\circ$  (or  $270^\circ$ ) to  $90^\circ$  degrees in  $5^\circ$  increments. Additional waveforms were collected when there was no target present.

Figure 3.6 shows the two raw waveforms as collected with no target present (top) and with the two 18 inch cylinders at 0 degrees. Although they are clearly different, it is difficult to identify characteristic differences in the two waveforms in this raw form. The distance between the target and the back wall of the room where the measurements were taken is 38 feet. This means that roughly 83% of the backscatter waveforms from the target and the back wall overlap in time. It is clear that a time-domain only signal processing approach will not be successful in identifying useful features from these signals.

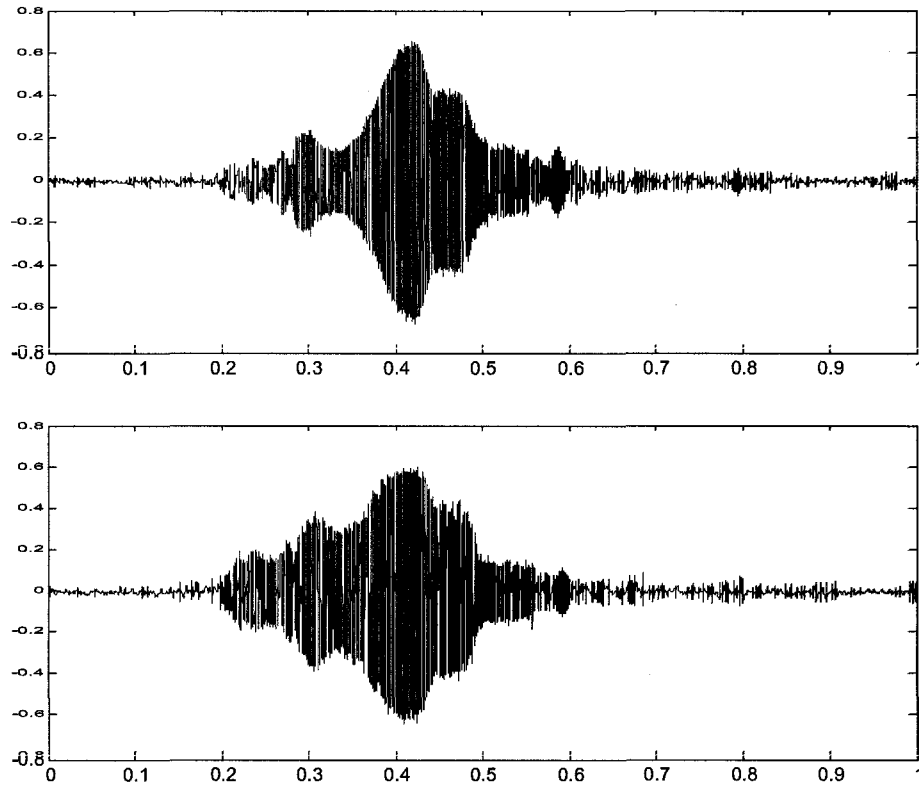




**Figure 3.4.** Top-down view of the two PVC pipes with the angles of the incoming sound beam shown.



**Figure 3.5.** Top-down diagram of the room used to take the experimental measurements. The blue arrow indicates the initial direction of the sound beam. The room dimensions are important because multiple reflections from the walls are present in the data.



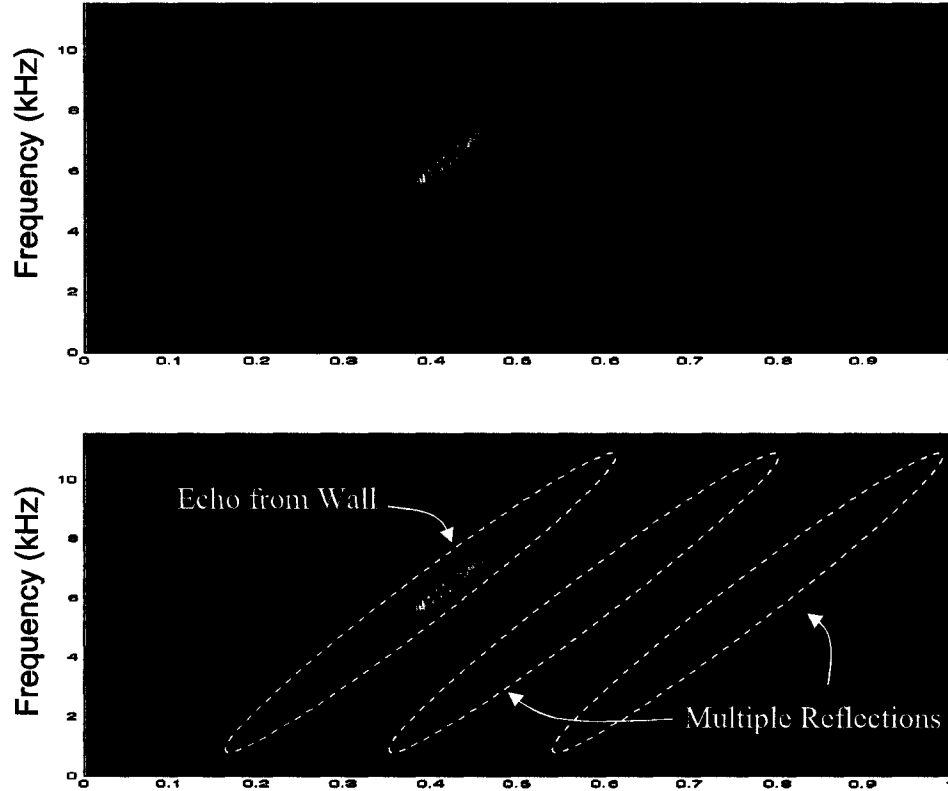
**Figure 3.6.** Experimental waveforms collected with no target present (top) and with a target present (bottom). In the bottom case, the target is two 18 inch PVC pipes with inner diameter of two inches. The pipes are separated by two inches and the incoming sound beam is at 0 degrees.

### **3.3. Signal Processing**

We use advanced signal processing techniques to extract more information from the acoustic signals to ultimately obtain more information about the scattering objects. The original excitation signal is a chirp containing frequencies over roughly 9000 Hz. The obvious technique to use is a short-time Fourier transform (also referred to as a spectrogram).

#### **3.3.1 The Short Time Fourier Transform (Spectrogram)**

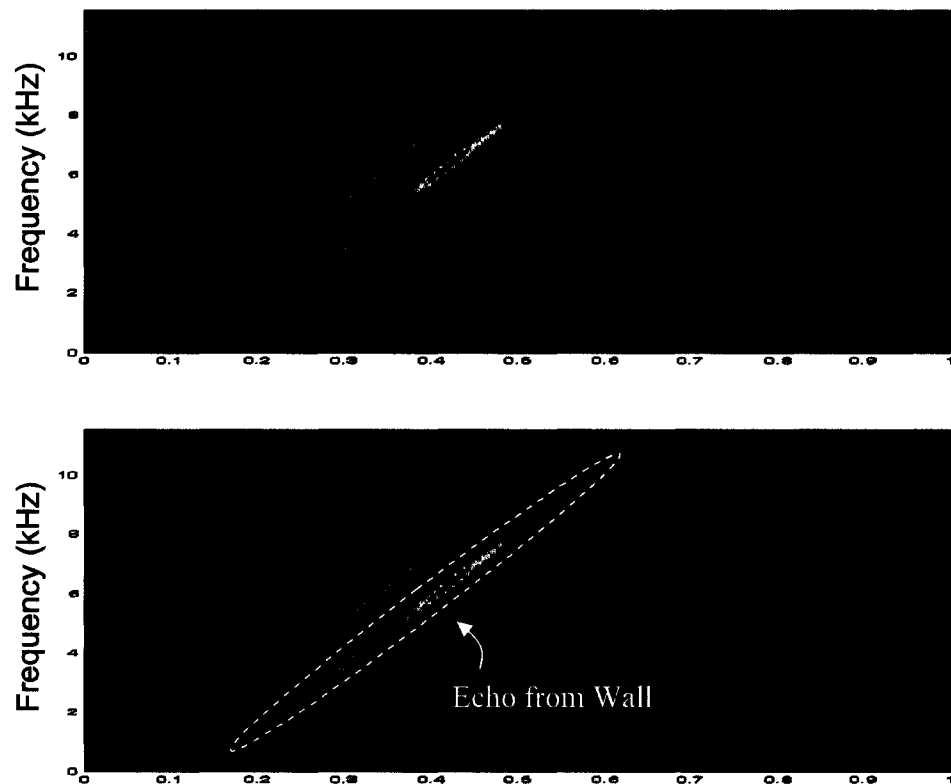
This technique transforms the one-dimensional time-domain signal into a two-dimensional representation. This representation preserves temporal information (along the x-axis) but also reveals frequency information about the signal (along the y-axis). Figure 3.7 shows a sample spectrogram of the signal collected when there is no target present (the same signal at the top of figure 3.6). The echo from the back wall is clearly visible in the spectrogram as shown in the bottom of figure 3.7. This upward slant in the time-frequency representation is exactly as expected because the original excitation was a linear frequency-modulated chirp. There are also several other faint echoes present in the signal that are the result of multiple reflections in the large room where the experiment took place. Even the echoes from the two end walls of the room, which are 106 feet apart overlap in time by 50 percent.



**Figure 3.7.** Spectrogram of the signal recorded with no target present. The identical spectrogram on the bottom is used to show the features that correspond to the backscatter echo from the wall and the multiple room reflections.

Figure 3.8 shows a sample spectrogram of the signal collected when a target was present (the same signal shown in the bottom of figure 3.6). The target is two 18 inch long PVC pipes with a two inch inner diameter and separated by two inches. The incoming beam was at 0 degrees to the pipes as shown in figure 3.4. The strong echo from the wall and the multiple reflections are still visible in this spectrogram. However, this spectrogram has an additional feature that corresponds to the back scattered echo from the target.

In this two-dimensional representation, it is easy to visually separate the features that represent the back-scattered echoes from the target, the back wall of the room, and the multiple room reflections. Now that we can visually identify the separate signals, we need to extract useful information so that we can say something about the scattering objects.

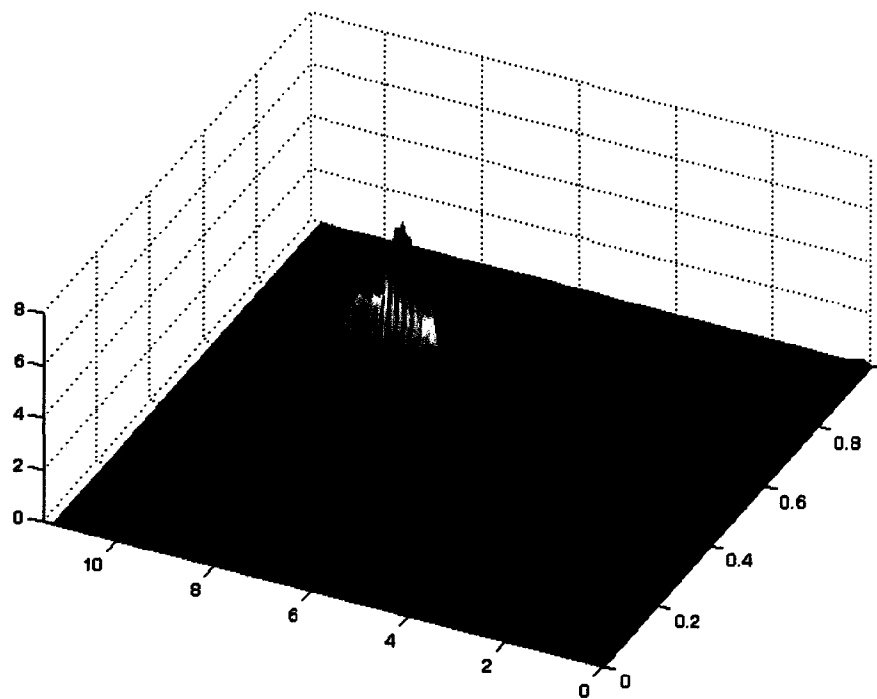


**Figure 3.8.** Spectrogram of the signal recorded with a target present. The identical spectrogram on the bottom is used to show the features that correspond to the backscatter echo from the target and the wall.

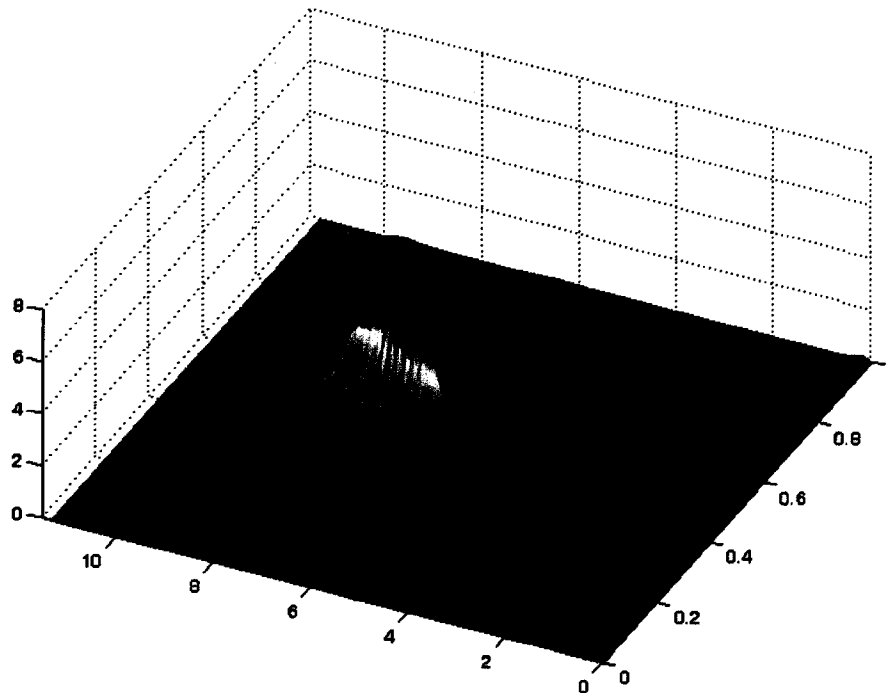
### 3.3.2 Feature Extraction

We use the term feature extraction to describe the act of retrieving information from such a signal. This information is then used to describe the physical situation. In this case, we want to extract information from the back-scattered echo to identify and describe the object that caused the acoustic backscatter. We will begin by extracting the amplitude of the backscattered echo as a function of frequency. Visually, this can be accomplished by plotting the spectrogram as a 3D surface plot instead of a 2D color intensity plot. Figure 3.9 shows a 3D spectrogram of the signal collected when no target was present. Now, instead of a slanted line feature of the 2D plot, we have a narrow mountain-like feature in the 3D representation. We can see a tall mountain that represents the backscatter from the wall and smaller mountain-like features that represent the multiple echoes from the room.

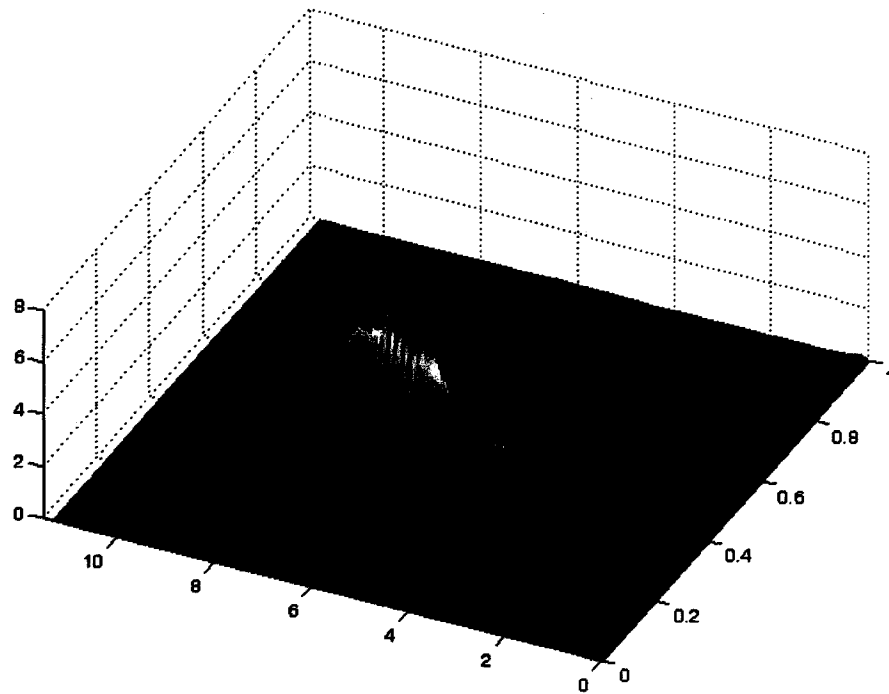
Figure 3.10 shows the 3D spectrogram of a signal with the target present. In this plot, we can see two prominent mountain-like features. The smallest one in the front represents the back-scattered echo from the target and the larger one represents the backscattered echo from the wall. Figure 3.11 shows the 3D spectrogram from the same target, but when the incoming sound beam was at 90 degrees. Notice that the mountain-like feature that represents the back scatter from the wall is nearly identical in shape in all three 3D spectrograms.



**Figure 3.9.** 3D Spectrogram of the signal recorded with no target present. The prominent mountain-like feature corresponds to the acoustic back-scatter from the wall.

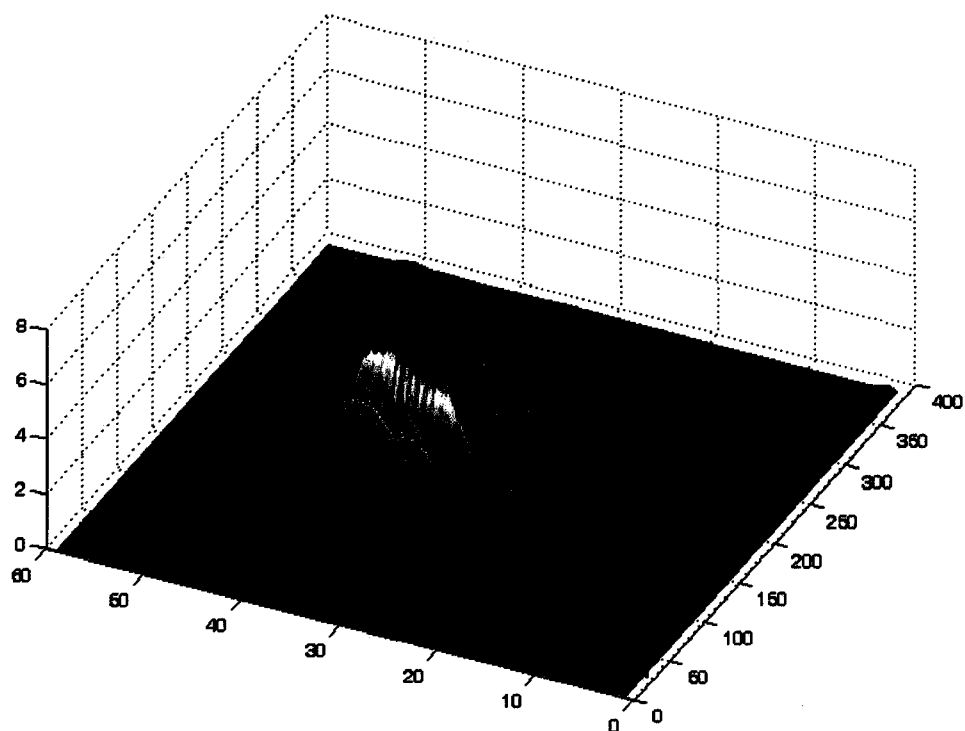


**Figure 3.10.** 3D Spectrogram of the signal recorded with the target present and at 0 degrees. The additional mountain-like feature in front corresponds to the acoustic backscatter from the target.



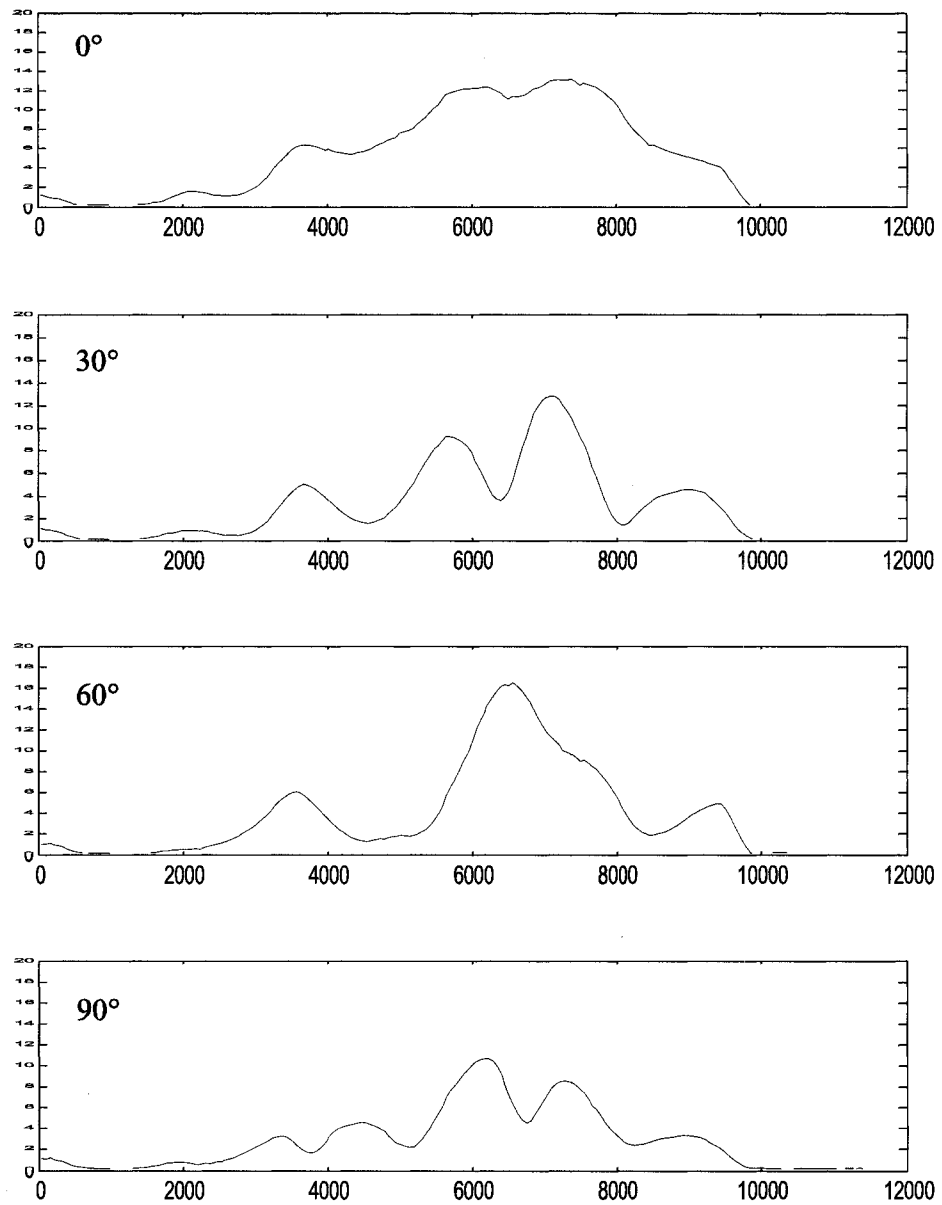
**Figure 3.11.** 3D Spectrogram of the signal recorded with the target present and at 90 degrees. The additional mountain-like feature in front corresponds to the acoustic backscatter from the target.

Looking at the 3D spectrograms in Figure 3.10 and 3.11, we can see that the shape mountain-like feature changes as the target is rotated. To describe the shape of the mountain-like features, we trace out the height along the ridge line of the feature of interest. A sample 3D trace is shown in Figure 3.12 for the target at 0 degrees. This trace gives us a function of the amplitude of the acoustic backscatter as a function of frequency. Examples of these traces are shown for 0°, 30°, 60°, and 90° in figure 3.13.



**Figure 3.12.** A 3D spectrogram with a trace of the ridge line of the mountain-like feature shown by the black line.





**Figure 3.13.** Several examples of the amplitude of the back scattered acoustic energy as a function of frequency for the same target but at 0°, 30°, 60°, and 90°.

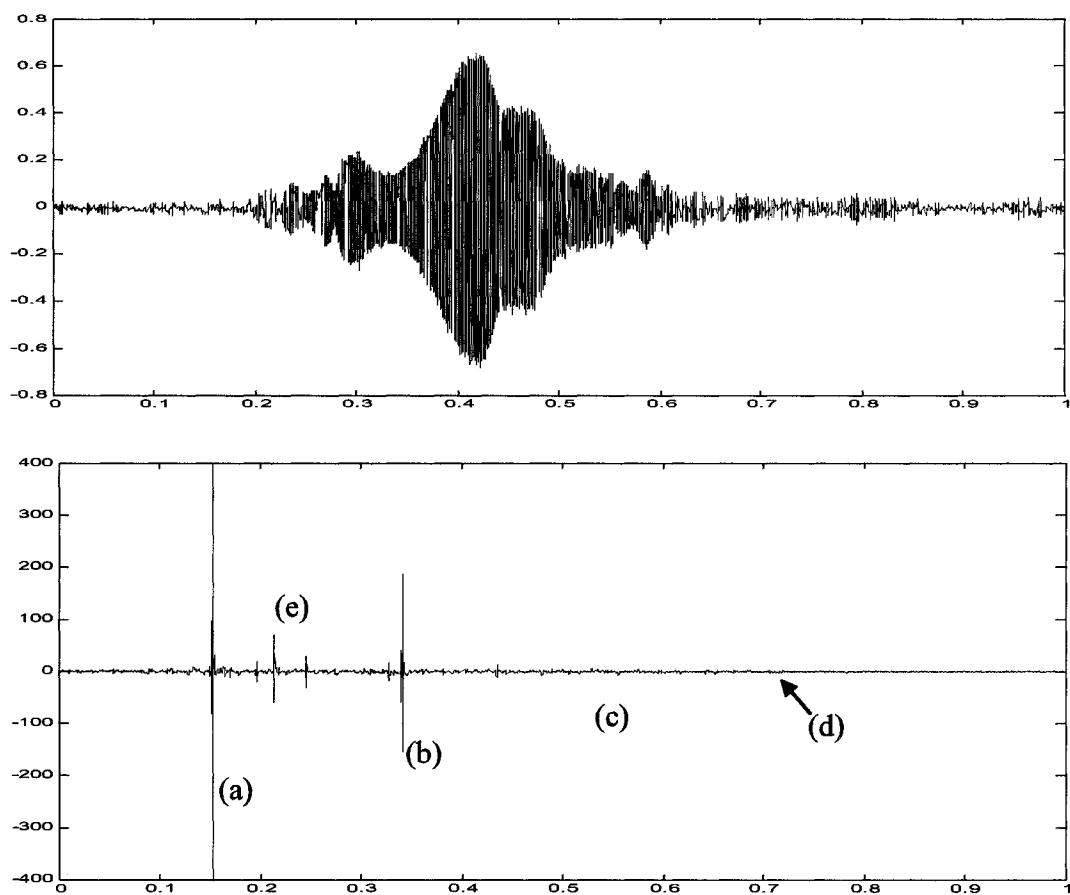
### 3.3.3 Feature Extraction using Pulse Compression

This method for feature extraction in the previous section works reasonably well but is not robust enough for an automatic feature extraction system. To quickly and accurately extract information from the signal, we turn to a signal processing method called pulse compression [1]. Here we use the fact that we know the structure of the initial excitation and we assume that any backscattered signals will share some of that structure. In this case, our initial excitation  $g(t)$  is a 0.4 second long frequency chirp ranging from 500Hz to 9500Hz. We “compress” our incoming signal by finding the cross-correlation between our initial excitation  $g(t)$  and our recorded signal  $f(t)$ . The cross-correlation function of two signals is found using equation (3.2).

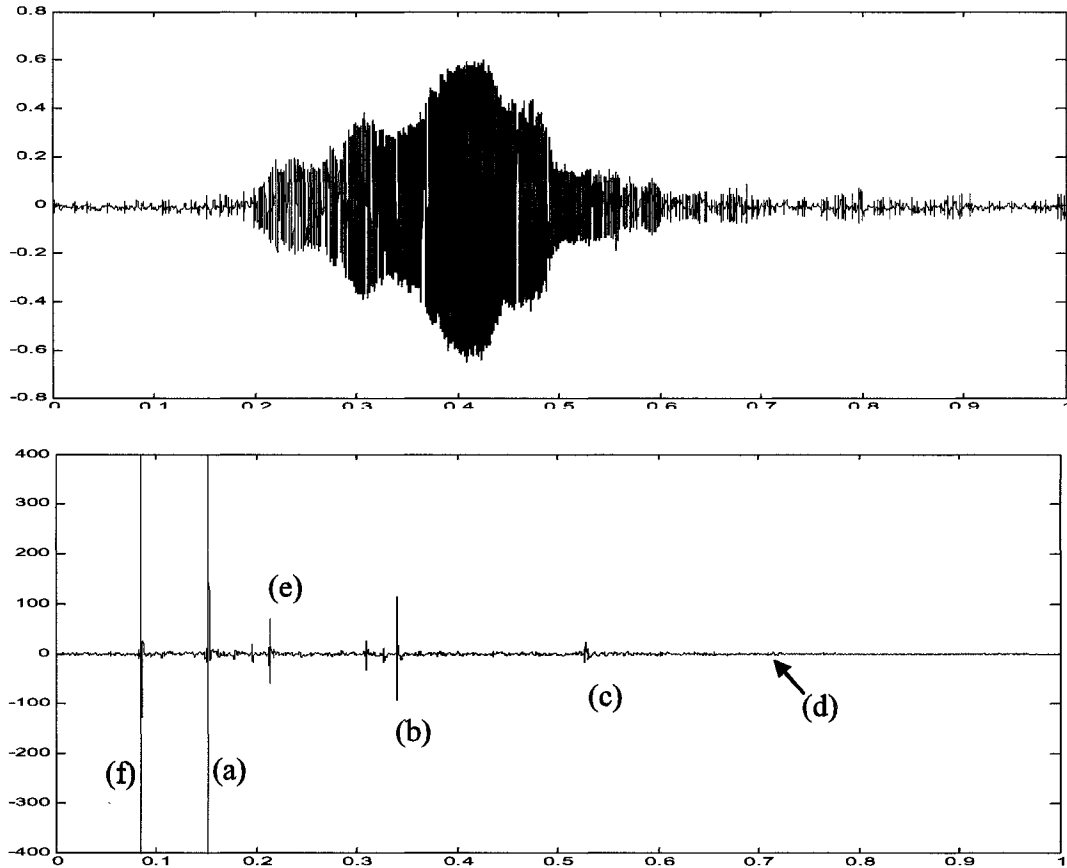
$$(f * g)(x) = \int f(t)^* g(t+x) dt \quad (3.2)$$

Note that the cross-covariance function is very similar to the convolution function. The difference is the cross-covariance function takes the complex conjugate of  $f(t)$  (that has no effect on our real signals) and has an addition of  $x$  instead of a subtraction of  $x$  in the  $g(t+x)$  term.

Figure 3.14 shows the raw recorded waveform when there is no target present. It is impossible to distinguish individual echoes in this signal. The bottom plot in Figure 3.14 shows the pulse-compressed signal. Now, the individual echoes are distinguishable and separable. Figure 3.15 shows a raw and pulsed-compressed signal when the target is present. The pulsed compressed signal looks identical to the one with no target except for the large echo that is a result of the acoustic interaction with the target.



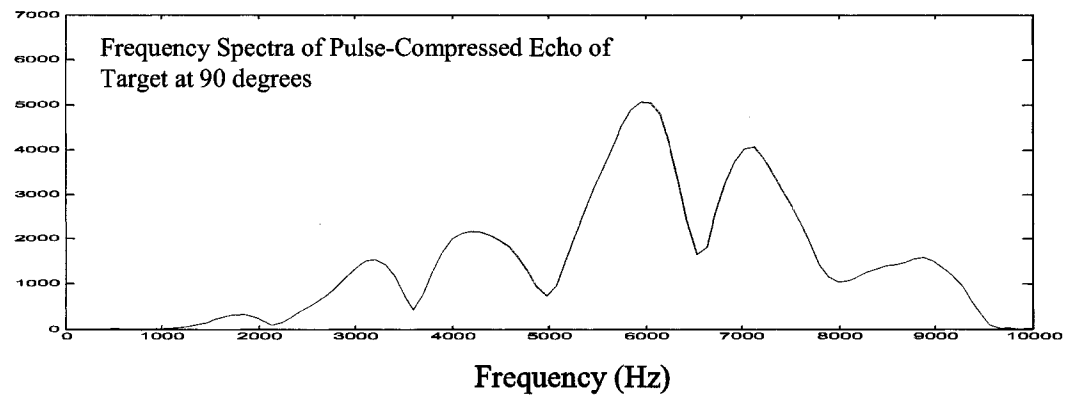
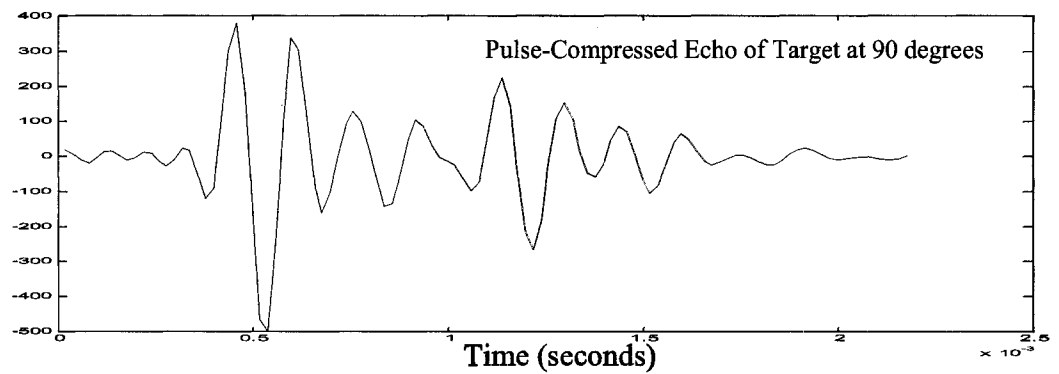
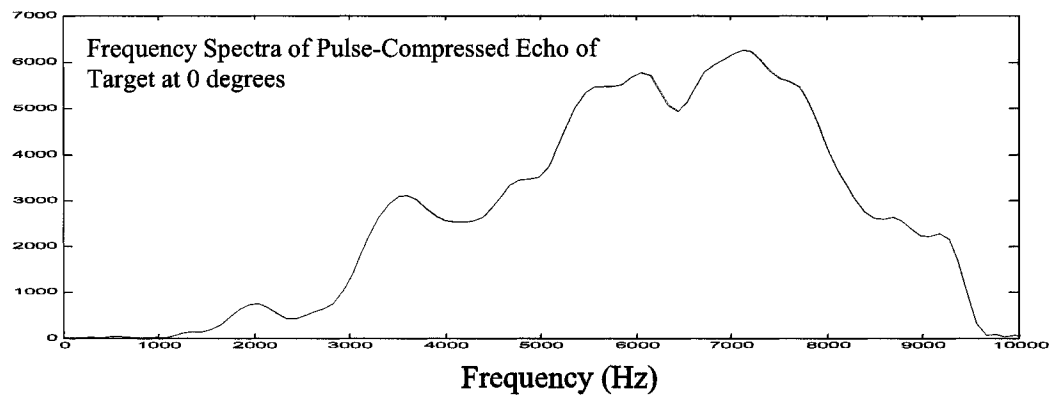
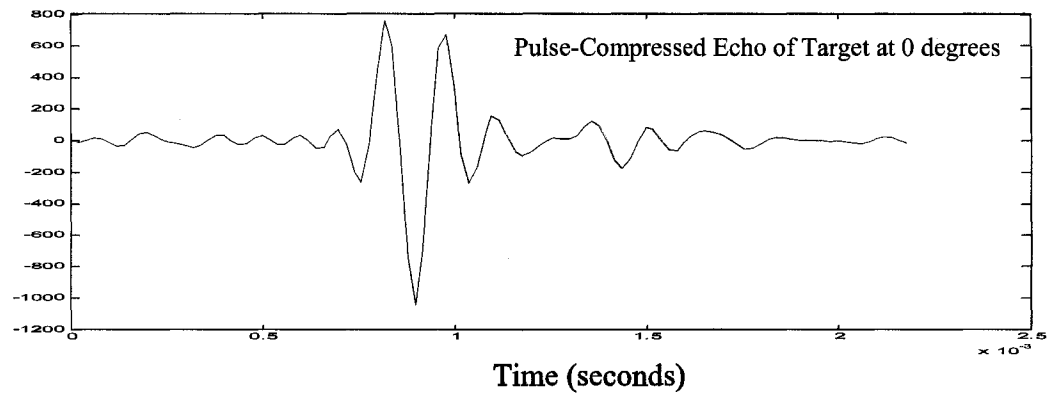
**Figure 3.14.** The top plot is the raw recorded waveform with no target present but with several overlapping echoes resulting from the room where the measurements were taken. The bottom plot shows the pulsed compression of the same signal. Several echoes are clearly distinguishable. They are the backscatter from the wall (a) and the several multiple room echoes (b-d). The signal also includes several echoes from other objects about the room (e).



**Figure 3.15.** The top plot is the raw recorded waveform with the target present which includes many overlapping echoes. The bottom plot shows the pulsed compression of the same signal. The same echoes are present (a-e) in this signal as there was in the signal with no target. There is an additional strong echo (f) that is the result of the acoustic interaction with the target.

The original 0.4 second long frequency chirp was approximately 132 meters long in air. After pulse-compression, the resulting echoes are approximately .0004 seconds long that is approximately 13 centimeters in air. The pulse compression improves the spatial resolution of the signal by a factor 1000. Another interesting ability of the pulse compression is its ability to pick out very faint echoes. For example, we can just pick out the echo resulting in 3 round trips in the large room where the measurements were taken. This sound wave traveled roughly 742 feet (or 226 meters).

Now we can clearly separate the echoes, but we need to extract the frequency backscattered content. We do this by simply windowing around the desired echo and taking the Fourier transform. Figure 3.16 shows the windowed echoes from the target at 0 degrees and 90 degrees along with their Fourier transforms. These frequency results are identical to the ones we extracted using the previous image processing technique. The pulse compression technique is fairly straight forward, easy to implement, and very robust.

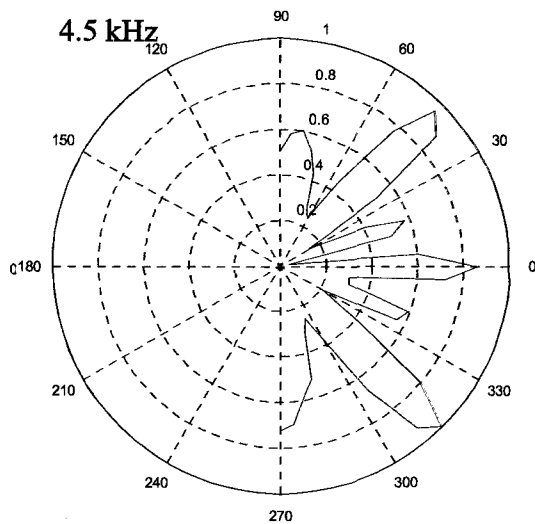
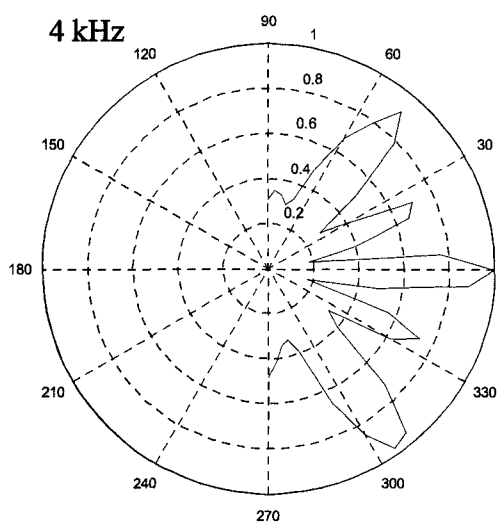
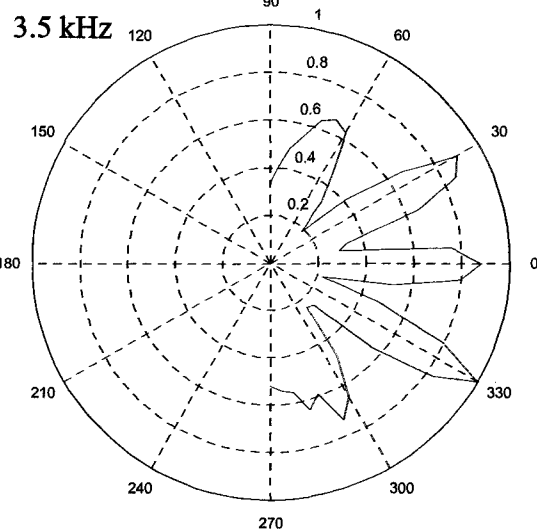
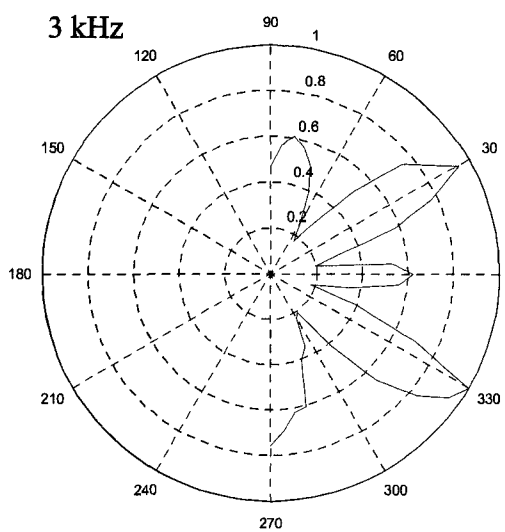
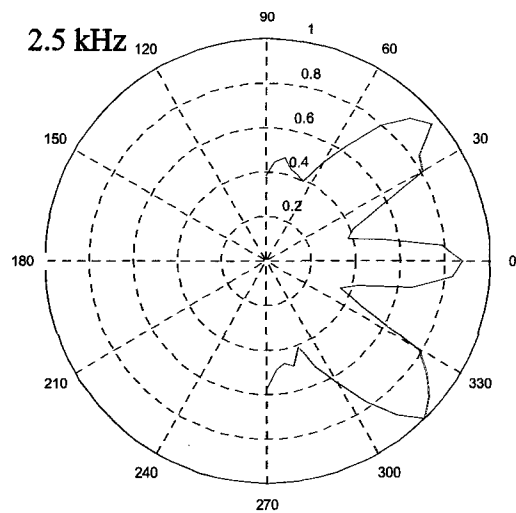
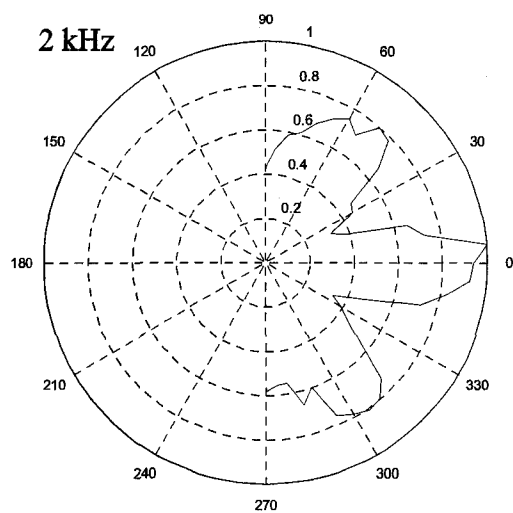


**Figure 3.16.** Windowed echoes from the target at 0 degrees and 90 degrees along with their Fourier transforms

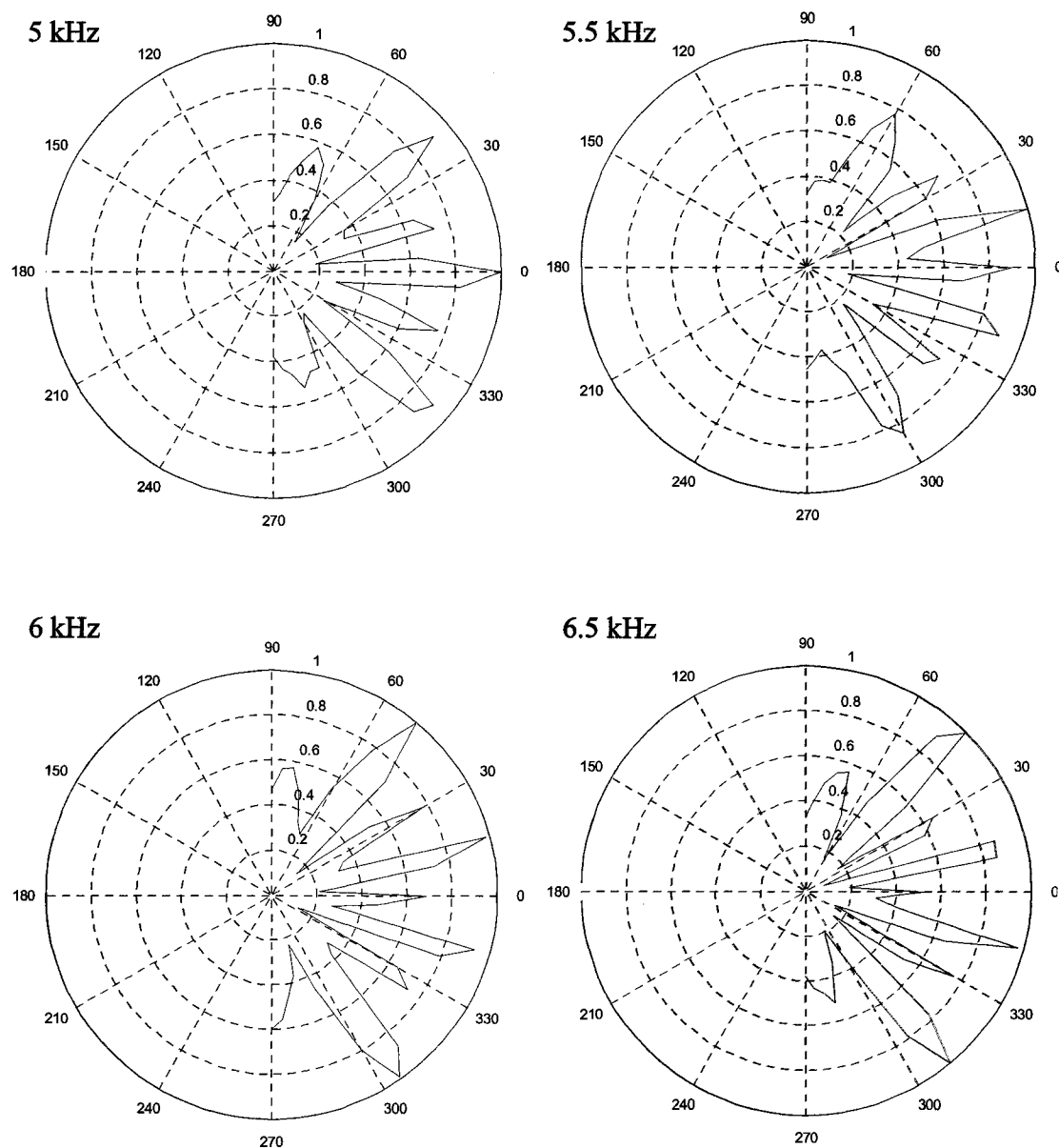
### **3.4. Comparison to 3DPAFIT Simulations**

The main objective of the parametric experiments was to validate the nonlinear acoustic propagation and 3D scattering simulations. The entire process of nonlinear beam forming and acoustic scattering from complex shaped objects is an extremely complicated physical process. To validate the simulations we will directly compare to measurements of backscattered acoustic energy from the PVC cylinders as a function of frequency and incident angle.

Experimental acoustic backscatter data was recorded as the target was rotated 180 degrees in Figure 3.4. The target was the two 18 inch long PVC pipes as described in the previous sections. The backscattered acoustic energy as a function of frequency was extracted at each angular position. Picking a certain frequency, a polar plot is then produced to show the backscattered acoustic energy as a function of incident angle. Examples of these polar plots at different frequencies are shown in figure 3.17.



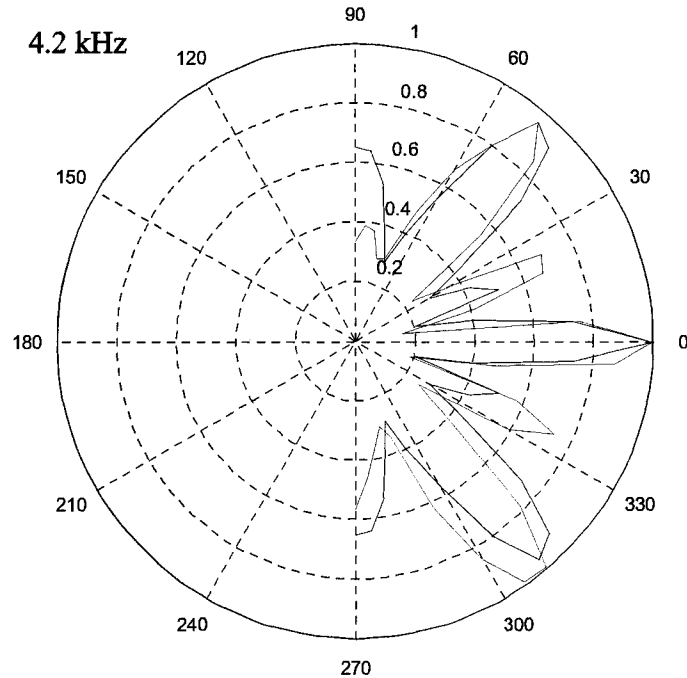




**Figure 3.17.** Polar plots showing the acoustic backscattered energy as a function angle at different frequencies.

The symmetric multiple lobe structure of the polar plots are expected; as the frequency is increased, the number of lobes also increases. Even with a relatively simple target, the nature of the scattering is, of course, very complicated. The backscattered acoustic wave is not only a sum of the scattering waves from the individual cylinders, but also the multiple scattered waves that reflect from one cylinder than the other.

The same scenario was simulated to compare to the experimental measurements. The KZK nonlinear simulations (which are described in Chapter 4) were used to propagate the acoustic waves from the parametric array to the target. The waves then entered the 3D acoustic simulation space where they reflected from the two cylinders. The cylinders were rotated just as they were in the physical experiment. The only difference between the two scenarios was that a single frequency tone burst was used in the simulations instead of a frequency chirp. The acoustic backscatter was recorded at each angle to produce a polar plot. The comparison between simulation and experiment results at 4.2 kHz are shown in figure 3.18. There is very good agreement between the two datasets.



**Figure 3.18.** Polar plot showing the acoustic backscattered energy as a function angle at 4.2kHz. Blue represents the simulation results and green represents the experimental results.

### 3.5 Conclusions

The acoustic scattering experiments using the parametric array and parabolic microphone were very successful. We demonstrated that this system along with advanced signal processing can be used to locate and describe targets. Two methods for extracting the backscattered frequency content of the targets were successfully presented. The pulse-compression method was superior in that it improved spatial resolution by at least 1000 percent over the original signal and it provided a straight forward and robust means to extract the frequency content of the individual echoes. This experimental

setup and the pulse-compression signal processing technique was used successfully validate the acoustic scattering simulations described in chapter 2.

### **3.6 References**

1. Szilard, F.L.a.J., *Pulse compression techniques in ultrasonic non-destructive testing*. Ultrasonics, 1976. **14**(3): p. 111-114.

## Chapter IV

### Applied 3DPAFIT Simulations:

#### Nonlinear Acoustic Concealed Weapons Detection

In this chapter, we present examples of acoustic simulations and how they can be applied to design hardware configurations and to study complicated acoustic interactions. The first part of this chapter is devoted to a nonlinear finite-difference simulation method used to simulate nonlinear sound beams. Many systematic simulations were run with this code to provide hardware design guidelines for the development of a prototype Nonlinear Acoustic Concealed Weapons Detector. Then this simulation method is coupled with the 3D parallel acoustic simulation method (3DPAFIT) to study acoustic wave interactions with humans, clothing layers, and weapons.

#### 4.1 Nonlinear Concealed Weapons Detection

Detecting concealed weapons and explosives on persons is an important and challenging problem. Current weapon detection technologies are often inadequate because they only detect metal objects at short distances. Our post 9/11 studies in acoustic concealed weapons detection are showing great potential in their ability to detect concealed objects at significant stand-off distances. The continued improvement of acoustic weapons detection technology will require exhaustive experiments, development

of nonlinear acoustic computer simulations, and sophisticated signal processing algorithms.

An acoustic weapons detection system consists of an ultrasonic transducer that emits a short inaudible acoustic beam into the air. This sound propagates away from the transducer until it comes in contact with the target person. The sound beam interacts with the person's clothes, body, and any other objects. Some of the energy in the sound burst reflects back to the transducer where it can be recorded on a computer as a waveform. The structure of the waveform depends on the details of the interaction between the sound waves and the person. A waveform from a person with a hidden weapon will differ from a waveform from a person without a hidden weapon.

Narrow-frequency band transducers were originally used to create the initial sound burst in acoustic concealed weapons detection [1]. A nonlinear acoustic concealed weapons detector uses sound waves to interrogate a person at large stand-off distances via a sound beam created using a parametric array and directed onto a person [2-4]. Using parametric arrays to create the initial sound burst may have many advantages over using typical transducers. Parametric arrays work by emitting high powered ultrasound waves that transition to lower frequencies because of nonlinear and absorption effects as the acoustic waves propagate. These nonlinear effects allow one to create a very narrow sound beam that can deliver the acoustic energy over large distances. Traditional air-coupled transducers have very narrow frequency bands that only allow single frequency tone-bursts. In contrast, parametric arrays have a broad frequency band that allows the initial waveform to contain a range of frequencies. In addition, the lower frequencies

produced by the nonlinear propagations will penetrate layers of clothing more effectively than the higher ultrasound frequencies.

#### 4.2 KZK Nonlinear Sound Beam Simulations

To model nonlinear acoustic beams, we turn to numerical solutions of the Khokhlov-Zabolotskaya-Kuznetsov (KZK) equation. The KZK equation is a nonlinear parabolic wave equation that accounts for the combined effects of diffraction, absorption, and nonlinearity in finite amplitude acoustic beams. In its derivation, the sound waves are assumed to form a directive beam, which permits a parabolic approximation to be made in the terms that account for diffraction. The parabolic approximation introduces errors at more than 20° off the beam axis, and at locations within several source radii to the source [5].

The KZK equation for an axi-symmetric sound beam that propagates in the positive  $z$  direction can be written in terms of the acoustical pressure  $p$  as follows.

$$\frac{\partial^2 p}{\partial z \partial t'} = \frac{c_0}{2} \left( \frac{\partial^2 p}{\partial r^2} + \frac{1}{r} \frac{\partial p}{\partial r} \right) + \frac{D}{2c_0^3} \frac{\partial^3 p}{\partial t'^3} + \frac{\beta}{2\rho_0 c_0^3} \frac{\partial^2 p^2}{\partial t'^2} \quad (4.1)$$

The first term on the right-hand side accounts for diffraction, the second term accounts for absorption, and the third accounts for nonlinearity [5-7]. Lee and Hamilton developed a finite difference method for simulating nonlinear sound beams based on the KZK Equation [5, 8]. We developed a simulation code to model parametric arrays for use in concealed weapons detection based on Lee and Hamilton's method.

With the KZK nonlinear acoustic simulations, we can test different parametric array specifications to judge their effectiveness for concealed weapons detection. For example, we can change the parametric array size, curvature (for focusing), frequency, and initial intensity. We can also test environmental effects such as air temperature and humidity level and how these variables affect the acoustic wave propagation. We have successfully used these KZK nonlinear simulations to provide design guidelines for building custom parametric arrays and accurately predict the pressure waveforms before they interact with the person.

#### **4.2.1 Linear vs. Nonlinear Parametric Simulations**

Parametric arrays create acoustic beams in air that are much narrower than traditional speakers of the same size because the beam properties are determined by the effective aperture relative to the ultrasound wavelengths. They work by emitting two high powered ultrasound frequencies  $f_1$  and  $f_2$  that undergo nonlinear distortion in the air to create a low acoustic difference frequency  $f_1 - f_2$ . This process is referred to as nonlinear-demodulation and is explained more in reference [9]. This difference frequency will be confined to a narrow beam and will propagate much further than the ultrasound frequencies. Using the KZK simulation code, we are able to perform many simulations of different transducer configurations with varying the degrees of nonlinearity and absorption. This allows us to systematically explore the nonlinear effects of acoustic propagation.

As an example, we will present the results of two simulations of the same parametric configuration with and without nonlinear effects. The first simulation does

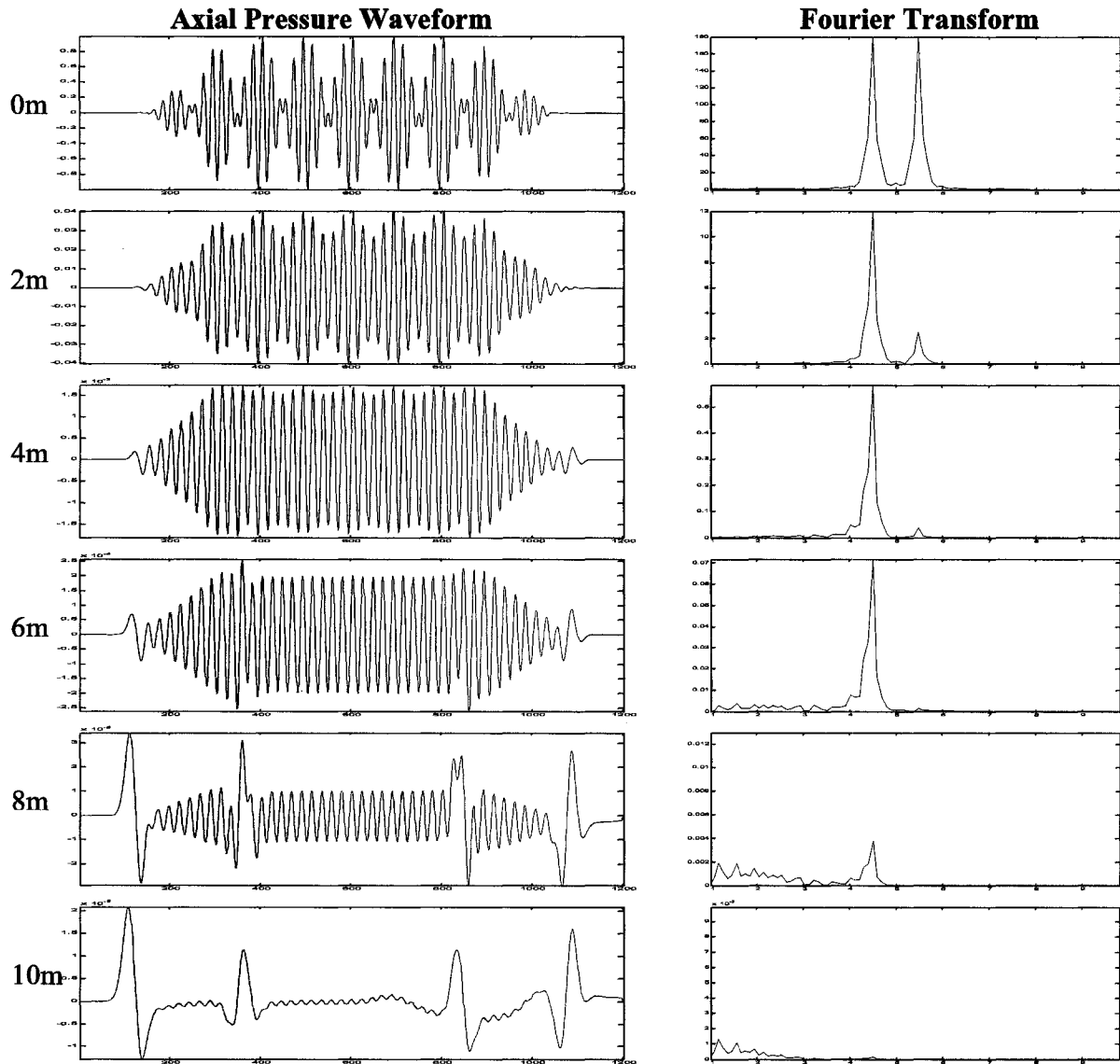


not include any nonlinear effects and the second uses the appropriate coefficient of nonlinearity for air ( $\beta = 1.2$ ). A 2 ft diameter transducer with a geometrical focus of 8m is excited with a short pulse that contains two frequencies: 45 kHz and 55 kHz. The initial sound pressure is 120 decibels.

Figure 4.1 shows the axial pressure waveforms from the linear simulation. Each waveform is recorded at 2m intervals starting at the face of the transducer and extending to 10m. At 0m, both ultrasound frequencies are of course present. As the wave propagates away from the transducer, the ultrasound is quickly absorbed due to the viscosity of the air. The 55 kHz component is absorbed much faster than the 45 kHz component because absorption is frequency dependent. After 6m, most of ultrasonic energy has been absorbed.

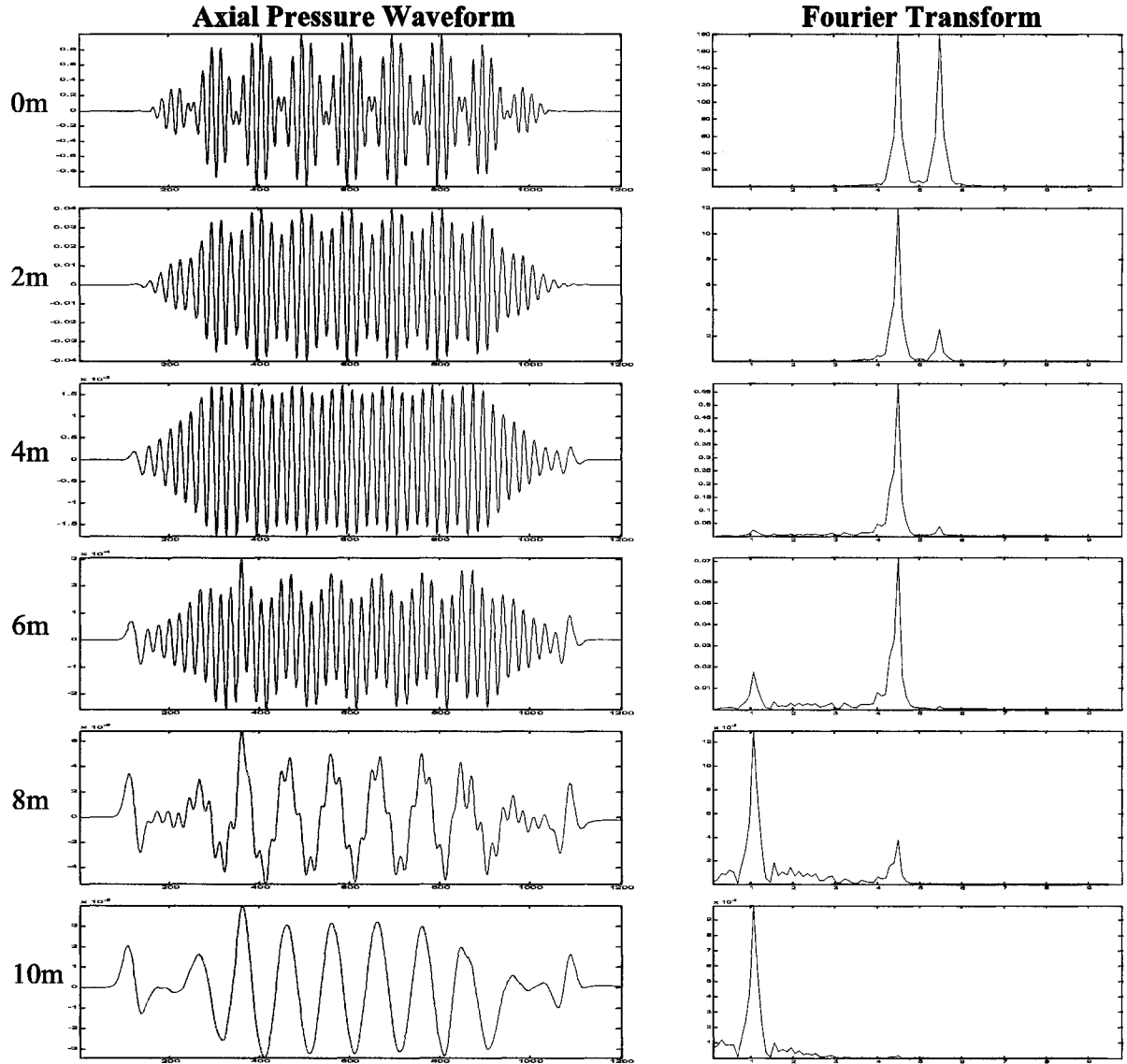
Figure 4.2 shows the axial pressure waveforms from the nonlinear simulation. The pressure wave is absorbed in the same manner as in the linear simulation. Now the difference frequency of 10 kHz is present and builds in amplitude as the waves propagate further away from the parametric array. As the higher frequencies are absorbed, the difference frequency becomes the dominant frequency in the wave. Figure 4.3 compares the full pressure fields from the nonlinear and linear simulations. The two simulations are almost identical until up the ultrasound frequencies are attenuated and the difference frequency becomes the dominant component of the nonlinear simulation. This clearly shows that the creation of the difference frequency is a result of the nonlinearity of air.

## Linear Axial Waveforms and their Fourier Transforms



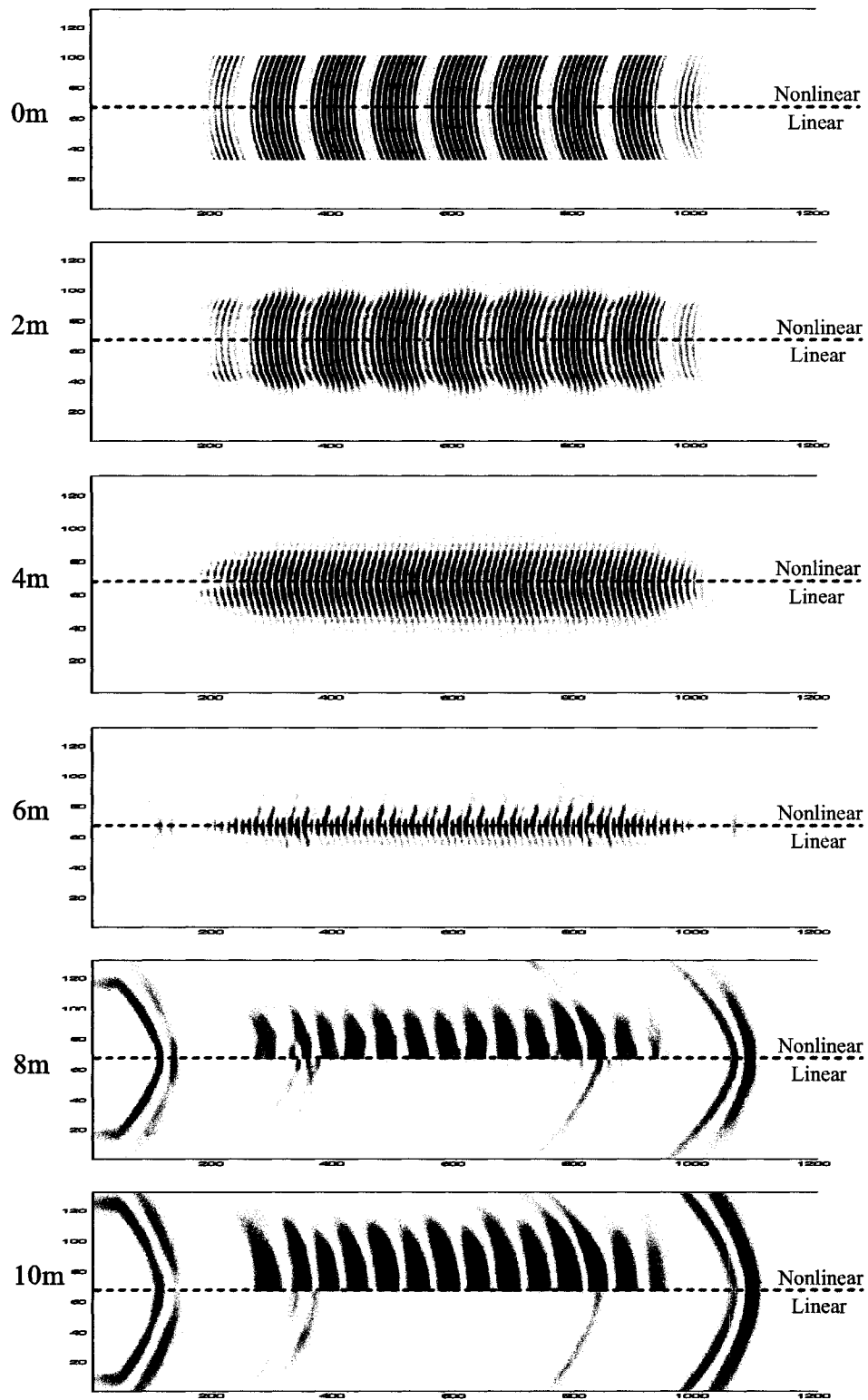
**Figure 4.1.** The left plots show the axial pressure waveforms from a focused parametric transducer. The right plots show their corresponding frequency spectra. No nonlinear effects are included in this simulation.

## Nonlinear Axial Waveforms and their Fourier Transforms



**Figure 4.2.** The left plots show the axial pressure waveforms from a focused parametric transducer. The right plots show their corresponding frequency spectra. Nonlinear effects are included in this simulation.

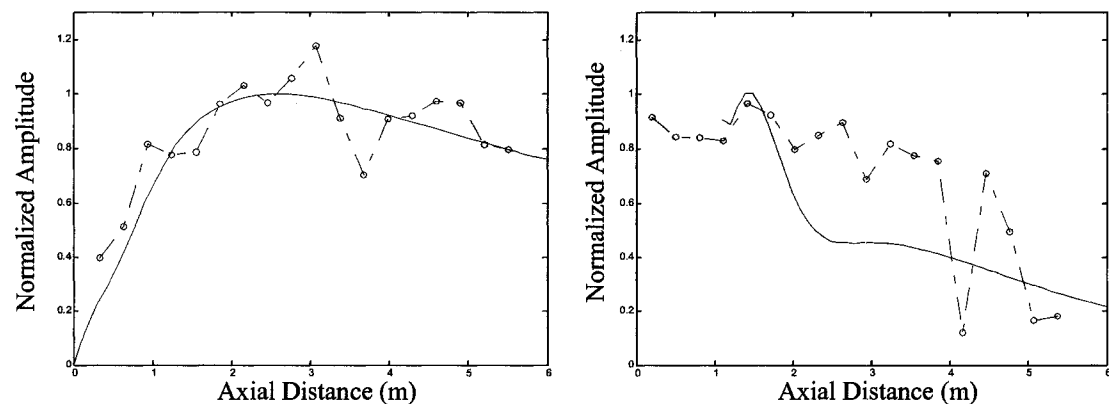
### Parametric Pulse Propagation – Nonlinear Vs. Linear



**Figure 4.3.** Shown is the full pressure field as the wave propagates away from the transducer. Each figure is split with the nonlinear simulation on the top and the linear simulation on the bottom. The x-axis represents time and the y axis represents radial direction (2ft from axis to edge).

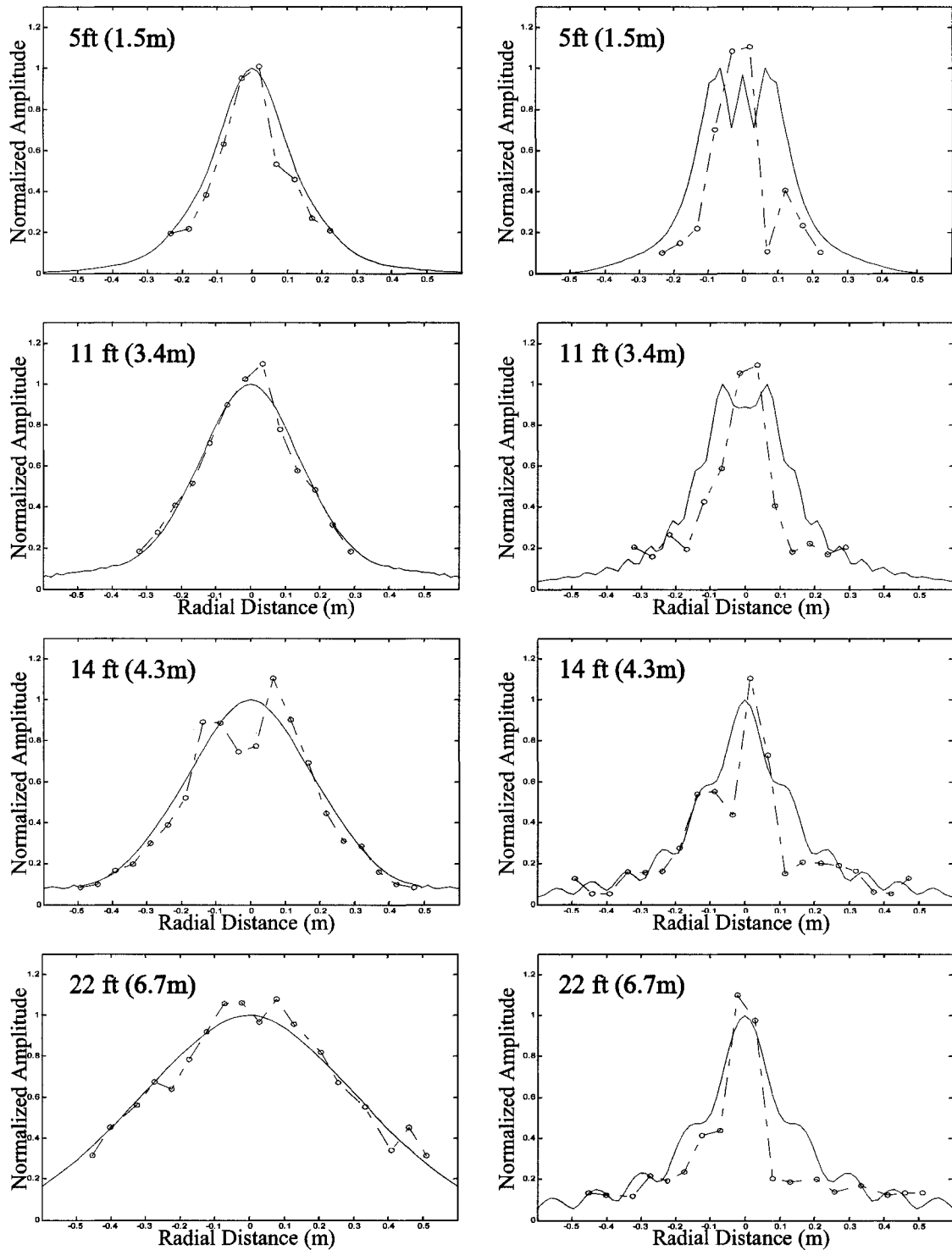
### 4.2.2 Comparison of Simulation and Experimental Results

Simulations have been performed to validate the final version of the nonlinear acoustic simulation code. This was done to confirm both the experimental measurements [2-4] and results from the new nonlinear acoustic simulation code. A 1ft diameter transducer was excited with two frequencies: 53620 Hz and 45620 Hz. The amplitude of the ultrasound frequencies and the difference frequencies were recorded along the beam axis at several distances. Figure 4.4 shows the audio and ultrasound absorption data from the transducer out to 6 meters. The experimental and simulated absorption data agree overall. Figure 4.5 shows the beam width profiles for the simulated and experimental data recorded at various distances from the parametric transducer. The simulated and experimental beam width profiles match very well.



**Figure 4.4.** Axial absorption plots for the audio (left) and ultrasound (right) components of the sound beam. The solid line represents the simulated data and the dotted line represents the experimental data.

## Simulated and Experimental Beam Width Profile Comparison

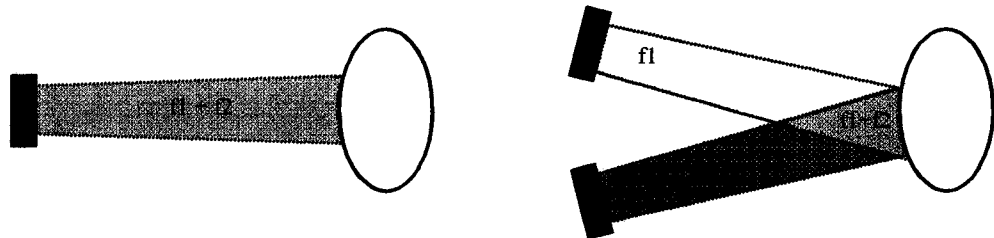


**Figure 4.5.** Beam width profiles for the audio (left) and ultrasound (right) components of the sound beam measured at various distances from the parametric transducer. The solid line represents the simulated data and the dotted line represents the experimental data.

### 4.2.3 Comparison of Confocal and Parametric Transducer Configurations

There are many hardware configurations that can deliver acoustic energy to the target at large stand-off distances. In some cases, more than one device can be used to deliver energy to the target. Many variables such as aperture size, curvature, and position can influence the resulting sound beams that can influence the performance of the nonlinear concealed weapons detector. A primary objective of this portion of the work was to study and compare the acoustic beams created by parametric array and cross beam transducer configurations. By studying the resulting sound beams, we can help to determine which configuration is best suited for detecting concealed weapons. In both configurations, two ultrasonic frequencies are emitted. In a parametric array configuration, the two ultrasonic frequencies are emitted from the same device that is pointed at the target. In a cross beam configuration, two ultrasound frequencies are emitted from two separate devices pointed at the target. Figure 4.6 shows a diagram of the two transducer configurations.

The main difference between the two configurations is the location and size of the interaction region of the two ultrasonic frequencies. In a cross beam configuration, the two frequency components only interact where the beams intersect. In a parametric



**Figure 4.6:** Diagram of parametric (left) and cross beam (right) transducer configurations. The blocks represent the transducers and the oval represents the target. The green regions indicate the interaction region between the two ultrasound frequencies.

configuration, the two frequencies interact over the entire length of the beam.

Simulations, using the nonlinear KZK code have been conducted to explore and compare the resulting sound beams of both configurations.

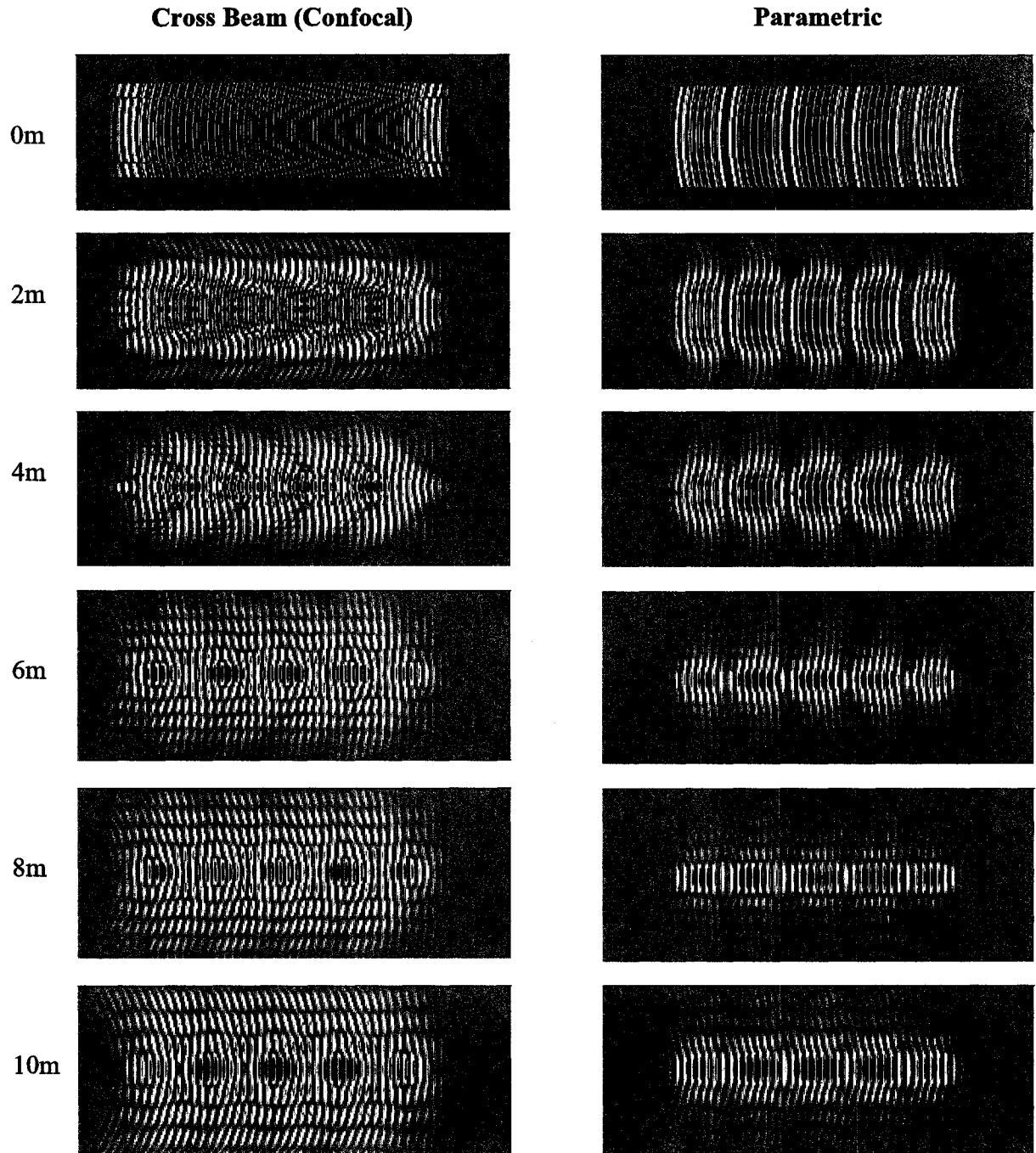
#### **4.2.3.1. Parametric Transducer Array Configuration**

A parametric array is a large transducer or an array of transducers that simultaneously emit two high-powered high-frequency ultrasound waves. The superposition of two frequencies causes the waveform to be amplitude modulated at the difference of the two original frequencies ( $f_1 - f_2$ ). As the waves propagate away from the parametric array, they begin to transform because of the nonlinearity of the air. When two high-powered ultrasound frequencies are present, a nonlinear phenomenon called demodulation occurs [9]. This creates a third frequency component at the difference frequency of the two original frequencies. The difference frequency will stay confined to the narrow beam creating a highly directional low frequency device.

Figure 4.7 shows pressure waveform snapshots for both a parametric and crossbeam configuration from 0 to 10 meters in two meter increments. In the parametric simulation, the transducer has a diameter of 0.6m (~2ft) with a geometrical focus of 8 meters. The parametric array emits a short pulse that contains two frequencies: 47 kHz and 53 kHz. Figure 3 shows the beam profiles for the two initial frequency components and the 6 kHz difference frequency component. The last plot of figure 4.8 shows the interaction region between the two main frequency components.

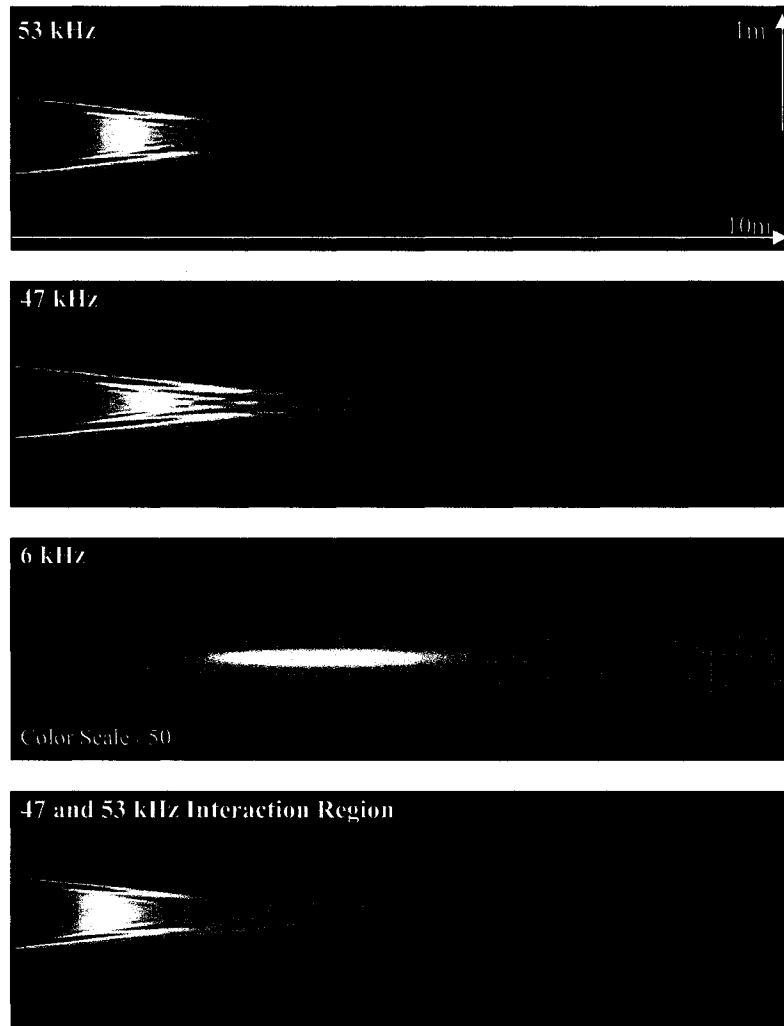


## Pressure Waveform Snapshots at Two Meter Increments



**Figure 4.7:** Pressure waveform snapshots at two meter increments for crossbeam (left) and parametric (right) transducer configurations. The crossbeam configuration is simulated using a confocal geometry. In both simulations the entire transducer has a diameter of 2ft and a geometrical focus of 8m.

### Parametric Beam Profile (in Pascals)



**Figure 4.8:** Beam profiles for the parametric array configuration. The top two plots show the beam profile for the ultrasound frequencies: 53 kHz and 47 kHz. The third plot shows the 6 kHz difference frequency component. The last plot shows the interaction region of both ultrasound frequencies which results in a 6 kHz amplitude modulated beat frequency

#### 4.2.3.2 Confocal Transducer Configuration

A confocal geometry was used to simulate the crossbeam transducer configuration. A confocal transducer consists of two transducer elements: a disk transducer surrounded by a ring transducer. The two transducer elements are excited at separate ultrasound frequencies. The faces of the two transducer elements are curved to cross the two sound beams at the focal point.

A number of confocal simulations were performed to compare the resulting sound beams to those of the parametric array configuration. Figure 4.9 shows typical pressure waveform snapshots from a confocal transducer configuration starting at the face of the transducer to 10 meters in 2 meter increments. In this simulation, the combined transducer diameter is 0.6m (~2ft) with a geometrical focal distance of 8 meters. The two transducer elements have the same surface area. The inner transducer emits a 53 kHz pulse at the same time the outer transducer emits a 47 kHz pulse. The two beams overlap as the two waves propagate towards the focal point. This creates both a beat frequency due to the linear superposition of the beams and a difference frequency component due to the nonlinear mixing. Figure 4.10 shows the beam profiles for the two main frequency components, the nonlinear generated difference frequency component, and the interaction region of the two ultrasound components.

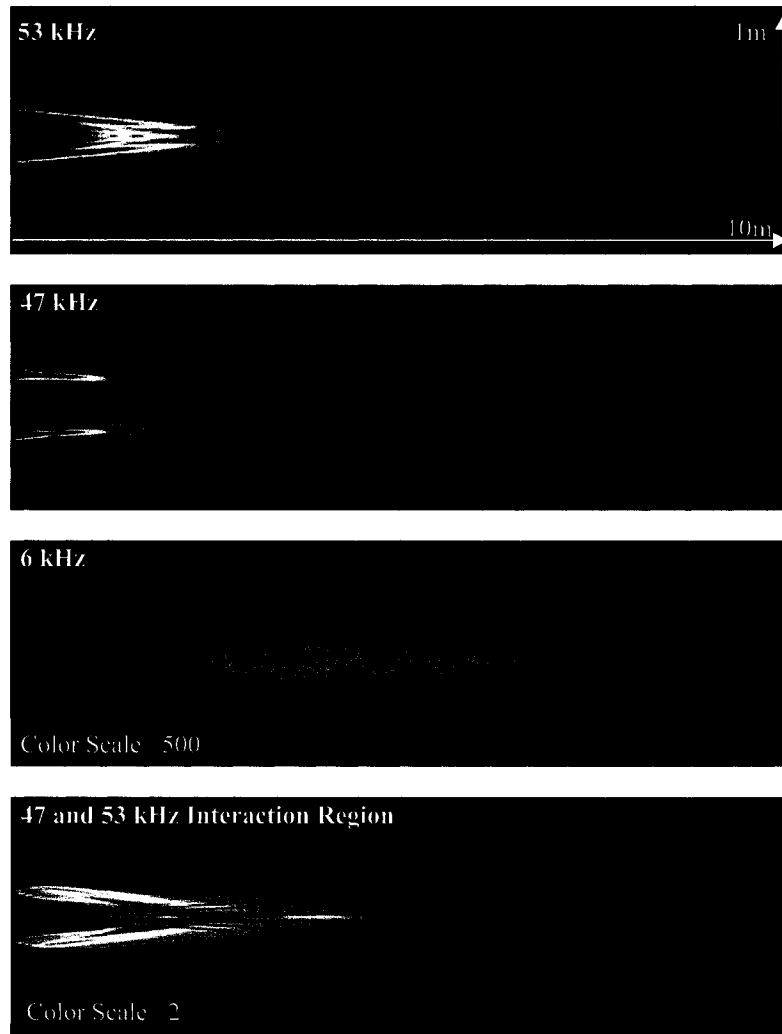
Additional simulations were performed to explore the effect of increasing the angle between the two sound beams. Figure 4.11 shows pressure waveform snapshots from one of these simulations. In this simulation, the two transducer elements are separated by about one meter. Figure 4.12 shows the different components of the resulting sound beams. In the wide angle cross beam simulations, the nonlinear

generated difference frequency is much smaller than in the narrow angle cross beam configuration. This smaller amplitude difference frequency is expected because the interaction region of the two sound beams is much smaller than the interaction region of in the small angle configuration.

It should be noted here that the wide angle cross beam configuration results show a very narrow sound beam. The purpose of these simulations is to show that increasing the angle of the two sound beams decreases the size of the interaction region. The confocal transducer used to create the results shown in figure 4.11 and 4.12 has a diameter of two meters. The large transducer size is the reason for the narrow sound beam. In a traditional cross-beam configuration (such as the one pictured in Figure 4.6), increasing the angle between the two transducers will not drastically affect the beam width at the target.

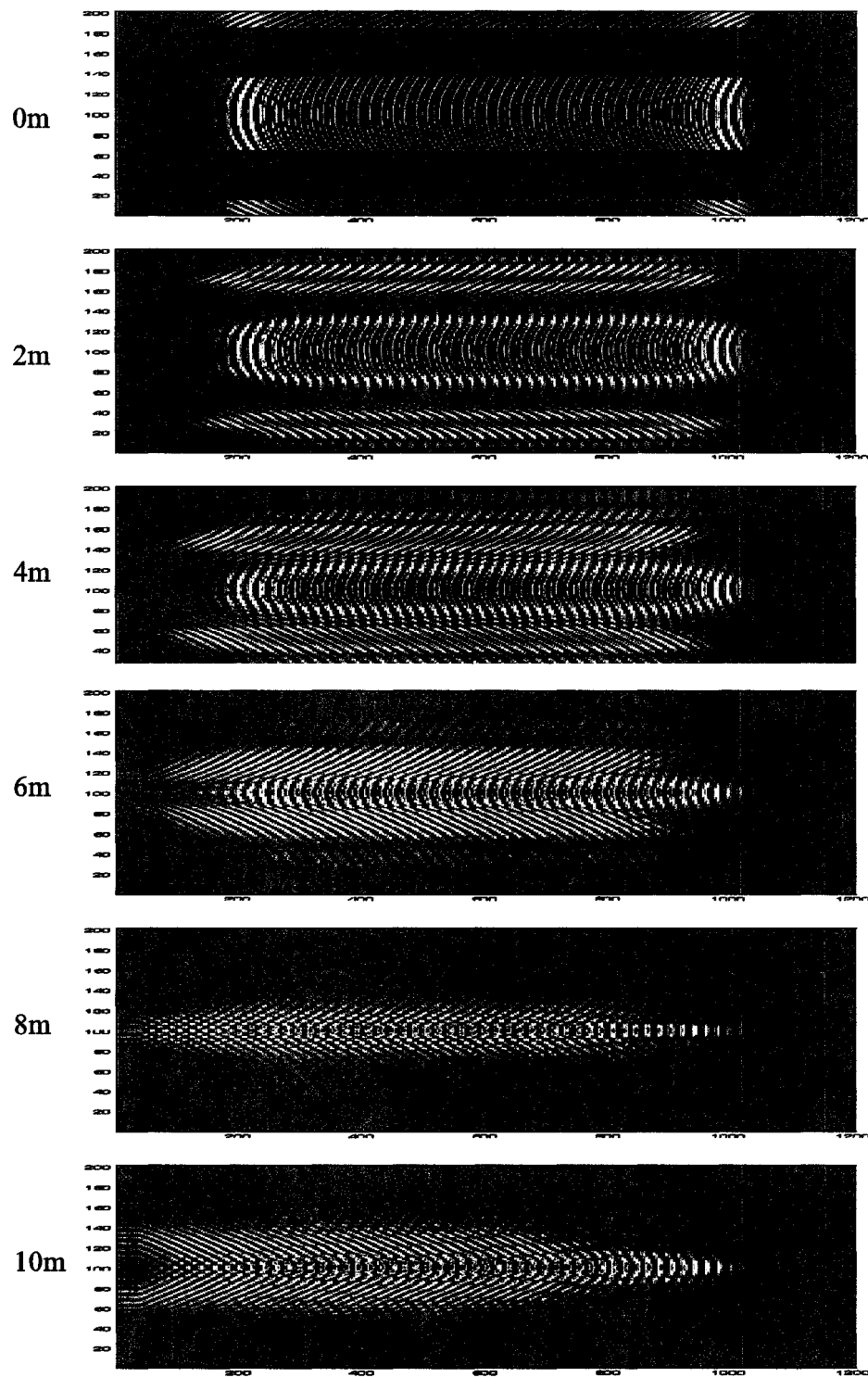
There may be a pure geometrical advantage of a cross beam configuration. By using two sound beams at separate angles, one should be able to retrieve more information about the target and the presence of any concealed weapons. Basically, the two sound beams have two separate “views” of the target. This is a purely geometric advantage and should not be considered in comparing the details of the two transducer configurations. This same effect could be achieved with a parametric array configuration by using two parametric arrays, moving the single parametric array to capture both “views”, or having the target simply rotate.

### Confocal Beam Profile (in Pascals)



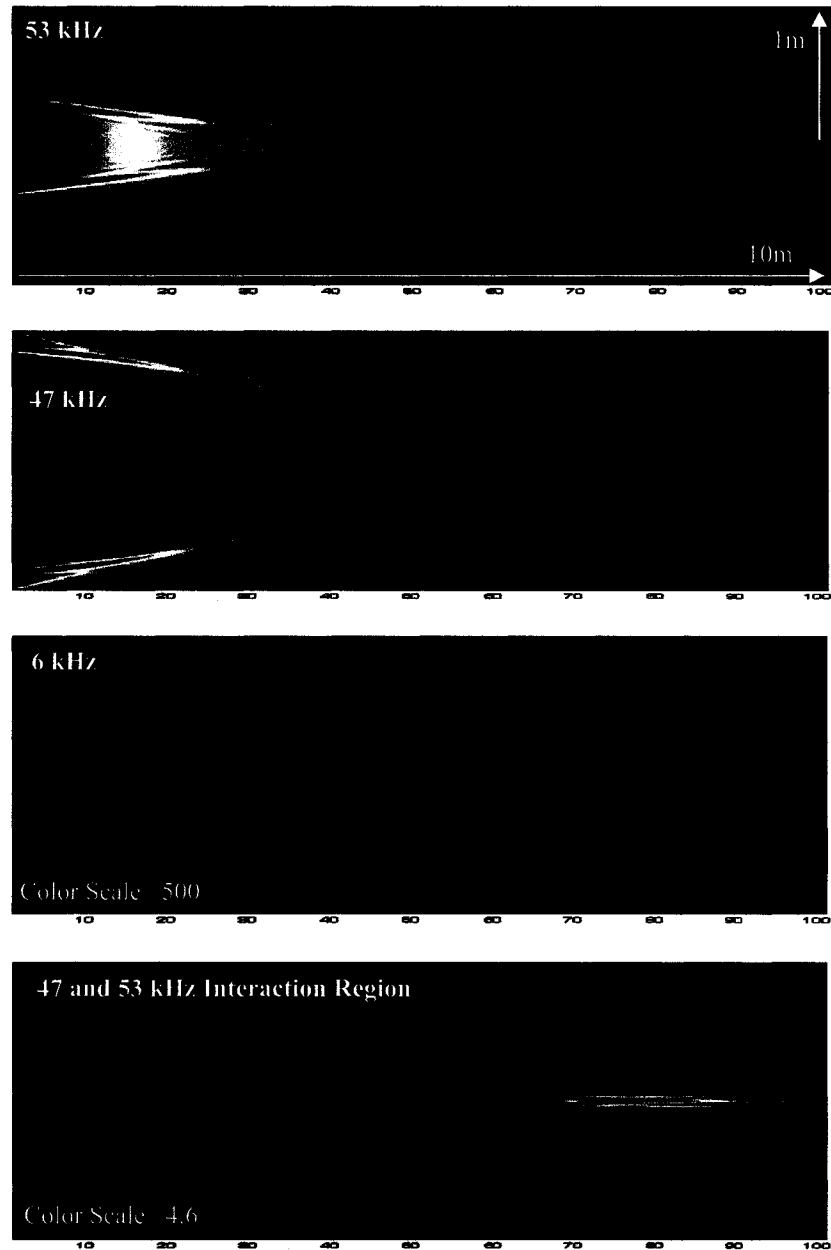
**Figure 4.9:** Beam profiles for the cross-beam array configuration. The top two plots show the beam profile for the ultrasound frequencies: 53 kHz and 47 kHz. The third plot shows the difference frequency 6 kHz component. The last plot shows the interaction region of both ultrasound frequencies which results in a 6 kHz amplitude modulated beat frequency.

### Pressure Snapshots for a Wide Angle Confocal Geometry



**Figure 4.10.** Pressure snapshots from a wide angle confocal simulation.

### Wide Angle Confocal Beam Profile (in Pascals)



**Figure 4.11:** Beam profiles for a wide angle cross-beam array configuration. The top two plots show the beam profile for the ultrasound frequencies: 53 kHz and 47 kHz. The third plot shows the difference frequency 6 kHz component. The last plot shows the interaction region of both ultrasound frequencies which results in a 6 kHz amplitude modulated beat frequency.

### **2.2.3.3. Comparison of the Two Transducer Configurations**

There are some obvious similarities and some noticeable differences in the sound beams created by parametric and the cross beam transducer configuration. While discussing the two transducer configurations, we emphasize on results that affect the interaction of the sound waves with a person's clothing and/or a concealed weapon.

#### ***Acoustic Beam Widths***

Knowing the physical dimensions of the acoustic beam of any transducer configurations is very important. The width of an acoustic beam quantitatively describes where the acoustic energy is located in the beam. The beam width is defined as the full-width at the half-maximum pressure intensity. Figure 4.12 shows the beam width profiles for the individual frequency components of the parametric and confocal geometries at 3, 6, and 9 meters. Table 4.1 provides the beam width values at these distances.

Another aspect that must be considered is the acoustic energy distribution at the target for each transducer configuration. In a parametric configuration, the energy distribution can be described by the beam width of the different frequency components. For a cross-beam configuration, the two beams cross at an angle. This creates a complicated interference pattern where the two sound waves constructively and deconstructively interfere with one another. This effect is easily seen for the confocal pressure waveform snapshots in figure 4.10. For a non-confocal cross beam transducer configuration, the interference pattern is highly dependent on the orientation and angle of the two sound beams. A small change in orientation of one of the sound beams will



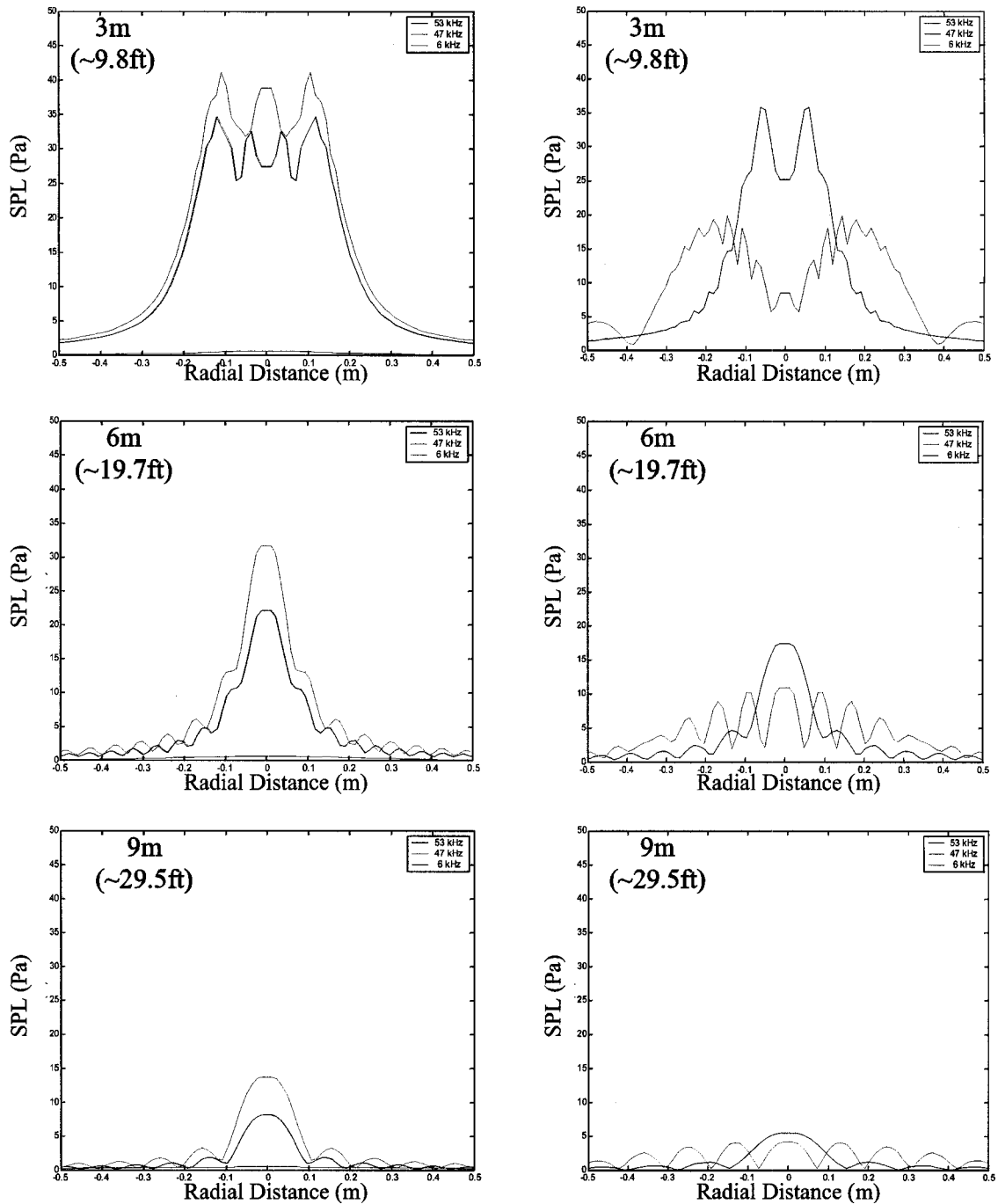
change the resulting energy distribution at the target. An experimental device that requires the two or more devices that are precisely positioned and aimed towards an uncooperative target is impractical.

**Full-Width at Half-Maximum Beam Widths**

	Parametric			Confocal		
	53 kHz	47kHz	6kHz	53 kHz	47kHz	6kHz
<b>3m</b>	0.38m	0.39m	0.27m	--	--	--
<b>6m</b>	0.12m	0.12m	0.54m	0.13m	--	0.70m
<b>9m</b>	0.14m	0.15m	0.89m	0.19m	--	0.95m

**Table 1:** Beam width for the different frequency components for the parametric and confocal geometry. For the confocal geometry, no beam widths are recorded at 3m because the beams have not yet fully overlapped. The interference pattern in the interaction region makes defining a beam width for the 47 kHz component difficult.

# **Beam Width Profiles at 3, 6, and 9 meters (in Pascals)** **Parametric                      Confocal**



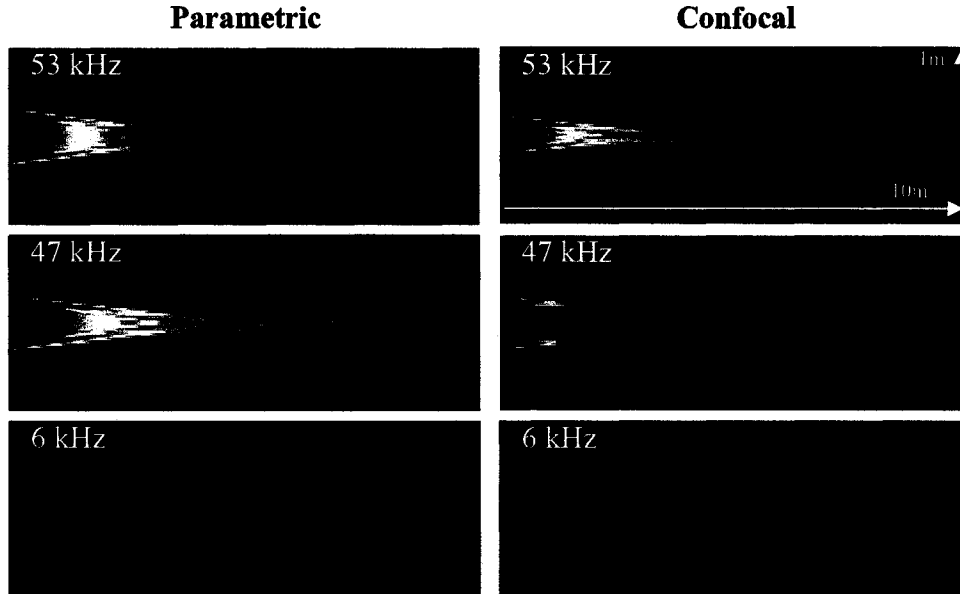
**Figure 4.12:** Beam width profiles for the different frequency components of the parametric (left) and confocal (right) transducer configurations. These beam width profiles are shown for the following taken at

### ***Magnitude of the Nonlinear Generated Difference Frequency***

There are several reasons the nonlinear generated difference frequency is useful for acoustic concealed weapons detection. First, this lower acoustic frequency is less affected by absorption. The absorption of a sound wave in air is proportional to the frequency of the sound wave squared  $f^2$  [10]. A 50 kHz acoustic wave will lose about 10 percent of its energy per meter where as a 5 kHz acoustic wave will only lose about 1 percent of its energy per meter. Therefore, creating a strong nonlinear generated difference frequency should result in much longer standoff distances for concealed weapons detection.

Another advantage of lower frequency sound waves is their ability to efficiently penetrate clothing. It has long been known in architectural acoustics, that lower frequency sound can penetrate thin structures much more efficiently than higher frequency sound. When a thin interface is smaller than the wavelength of the sound waves, the amount of energy that is transmitted across the interface is proportional to  $1/f^2$  [11]. Therefore, a 5 kHz sound wave is 100 times more efficient in penetrating a thin interface than a 50 kHz sound wave. As a result, lower frequency sound waves will also penetrate clothing much more efficiently than higher ultrasound frequencies sound waves.

### Linear Beam Profiles for Parametric and Confocal Configurations

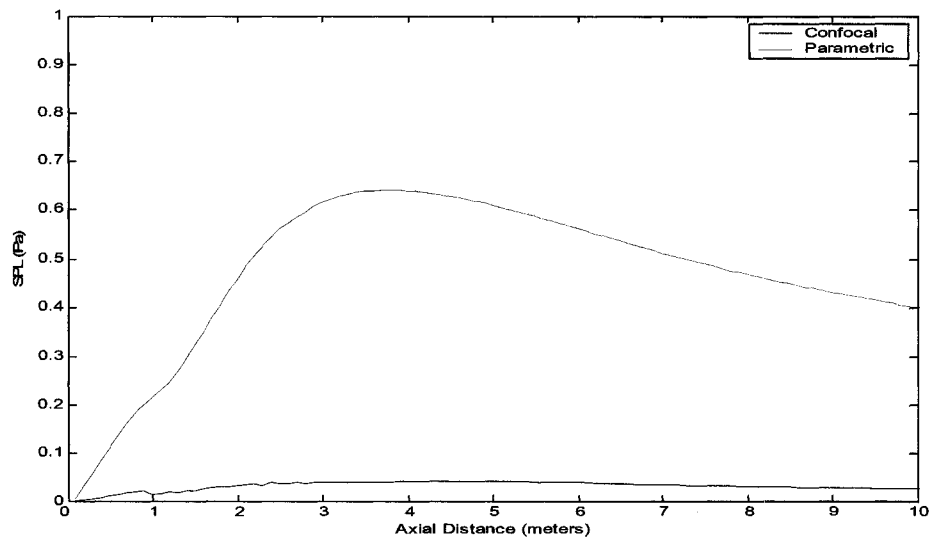


**Figure 4.13:** Linear beam profiles for parametric (left) and confocal (right) transducer geometries. These simulations are identical to the ones of figures 3 and 4, except the nonlinearity is turned off. This shows that the 6 kHz frequency component is a product of the nonlinearity of the air.

We have shown that the difference frequency observed in the simulations is coming from the nonlinearity in air. We proved this by performing the exact same simulations but with the nonlinearity turned off (Figure 4.13). With the nonlinearity turned off, there is no 6 kHz difference frequency component. The most noticeable difference between the two transducer configurations is the magnitude of the nonlinear difference frequency. The magnitude of the nonlinear difference frequency component is dependent on both the intensity of the two ultrasonic frequencies and the size of the interaction region. For the parametric configuration, the two ultrasonic frequencies overlap over the entire sound beam. This allows the nonlinear difference frequency

component to be generated over the entire length of the beam. The parametric configuration also overlaps the two frequencies near the face of the transducer where the ultrasonic frequencies are the strongest. Figure 4.14 shows the axial intensity of the nonlinear difference frequency for both the parametric and confocal geometry.

For the confocal transducer configuration, the intensity of the nonlinearly generated frequency component is significantly smaller than the intensity found with the parametric configuration. As shown in figure 4.14, the nonlinear difference frequency of the confocal configuration is at least an order of magnitude smaller than found for the parametric configuration. This lower intensity difference frequency is caused by two factors. First the interaction region of the two beams is much smaller. Also, the two beams overlap away from the face of the transducers where the intensities of the ultrasound frequencies have already diminished because of absorption effects. As the angle between the two transducers becomes larger, the interaction region becomes smaller. This further reduces the magnitude of the nonlinearly generated difference



**Figure 4.14:** Axial intensity values of the nonlinear difference frequency for the confocal (blue) and parametric (green) configurations.

frequency.

Using a cross beam configuration will significantly reduce the amplitude of the nonlinear generated difference frequency. This conclusion is supported by an analytical study of Hamilton [12], where he shows that the nonlinear generated difference frequency component is much smaller for noncollinear (small angle cross beam) transducer geometries. Hamilton concludes that the reduction in the size of the interaction region drastically reduces the ability for the cross beam configuration to generate a difference frequency. We thus conclude that the parametric configuration is the most efficient configuration for generating a nonlinear difference frequency.

#### **4.2.3.4. Amplitude Modulated Beat Frequency - Vibro-Acoustography**

Creating a nonlinear difference frequency is not the only proposed method for delivering energy to the target. Vibro-acoustography is an imaging method that uses an amplitude modulated ultrasound wave to create a localized dynamic radiation force on an object [13, 14]. This dynamic radiation force causes the object to vibrate and emit an acoustic signal. This acoustic emission signal is dependent on the object's geometry and material parameters.

Radiation pressure is defined as the time-averaged force exerted by an acoustic field on an object [14]. An amplitude modulated ultrasound wave incident on an object's surface creates a dynamic radiation force at the same frequency of the amplitude modulation. Unfortunately, this amplitude modulated signal will also undergo nonlinear demodulation which generates a difference frequency component. When recording a low frequency signal coming from the object, it is very difficult to determine which

phenomenon caused the low frequency signal. It could be from the nonlinear difference frequency component scattering off the object, the acoustic emissions created by the radiation pressure effect, or some combination of the two.

Both, the parametric and confocal geometry produce an amplitude modulated acoustic wave that could produce a radiation pressure effect. The parametric configuration produces a stronger and more uniform amplitude modulated signal than a confocal geometry of the same overall transducer size. The parametric array emits both frequencies over the entire face of the transducer. The confocal transducer only emits one frequency per transducer element. Two transducers of the same surface area of the parametric array would be needed to achieve the equivalent acoustic intensities.

For the Vibro-Acoustography method to be successful, the ultrasound frequencies must reach the concealed weapon. We have already shown that the higher frequency ultrasound waves are less efficient in penetrating clothing. Dense clothing materials, such as leather, may reflect too much of the ultrasound energy for this method to be viable. Additional studies will be needed to test the acoustic transmission of various clothing types.

#### **4.2.3.5. Confocal vs. Parametric Array Conclusion**

We have simulated both parametric and cross beam transducer configurations to compare the resulting sound beams. The parametric array produces a narrow and uniform sound beam with a strong nonlinear difference frequency. The crossbeam geometry has a reduced interaction region that reduces the intensity of the nonlinear difference frequency. The cross beam geometry also produces a complicated interference

pattern in the interaction region of the two sound beams. This interference pattern may make characterizing and repeating experimental measurements difficult, especially for “uncooperative” targets under field conditions.

#### **4.2.4. Effects of the Initial Sound Pressure Intensity on the Resulting Sound Beam**

Several additional simulations have been performed to compare the sound beams of different initial sound pressure intensities. A 2ft diameter transducer with a geometrical focus of 8m was used in all of the following simulations. The transducer emits a short dual frequency tone burst at 47 kHz and 53 kHz. Six initial sound pressure intensities were tested: 120dB, 125dB, 130dB, 135dB, 140dB, and 145dB.

Table 4.2 gives the sound pressure intensities and the beam widths of the three frequency components of the sound beam recorded at 5.8 meters. Figure 4.15 shows the beam width profiles for the three frequency components of the sound beam measured at 5.8 meters. The shape and width of the beam are not affected by changing the initial sound pressure intensity.

Nonlinear effects are amplified by increasing the amplitude of the initial sound wave. As the initial sound intensity is increased, more energy is sent to the nonlinear generated frequency components ( $nf_1 \pm mf_2$ ). This pattern is shown Table 4.2. As the initial sound pressure is increased in 5dB increments, the intensity of ultrasound frequencies recorded at 5.8m increases in increments less than 5dB. This loss is attributed to more energy being sent to the nonlinearly generated frequencies. The nonlinearly generated difference frequency (6 kHz) increases in increments of about 8-10 db each time the initial sound pressure is increased by 5dB. Figure 4.16 shows the axial

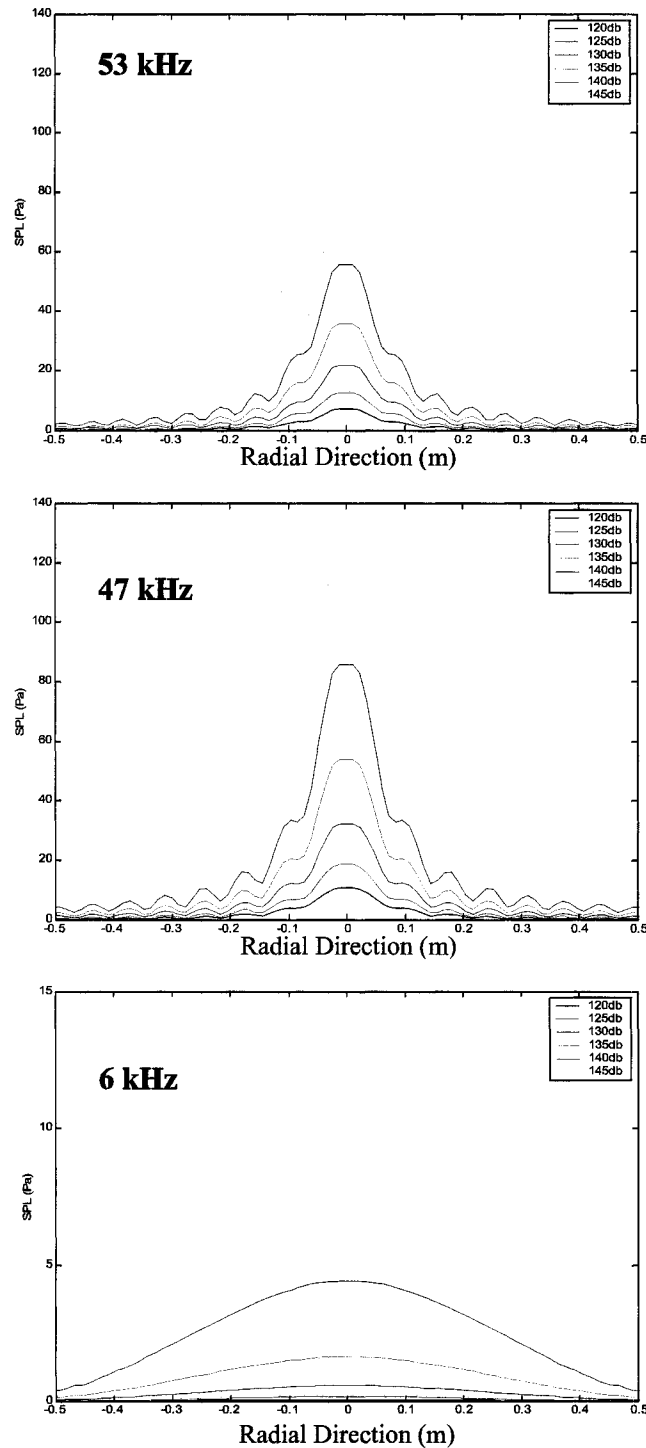


sound pressure intensities of the three frequency components for the different initial sound pressure intensities.

**Table 4.2 Beam Width and Intensities recorded at 5.8 meters for different initial sound pressure intensities.**

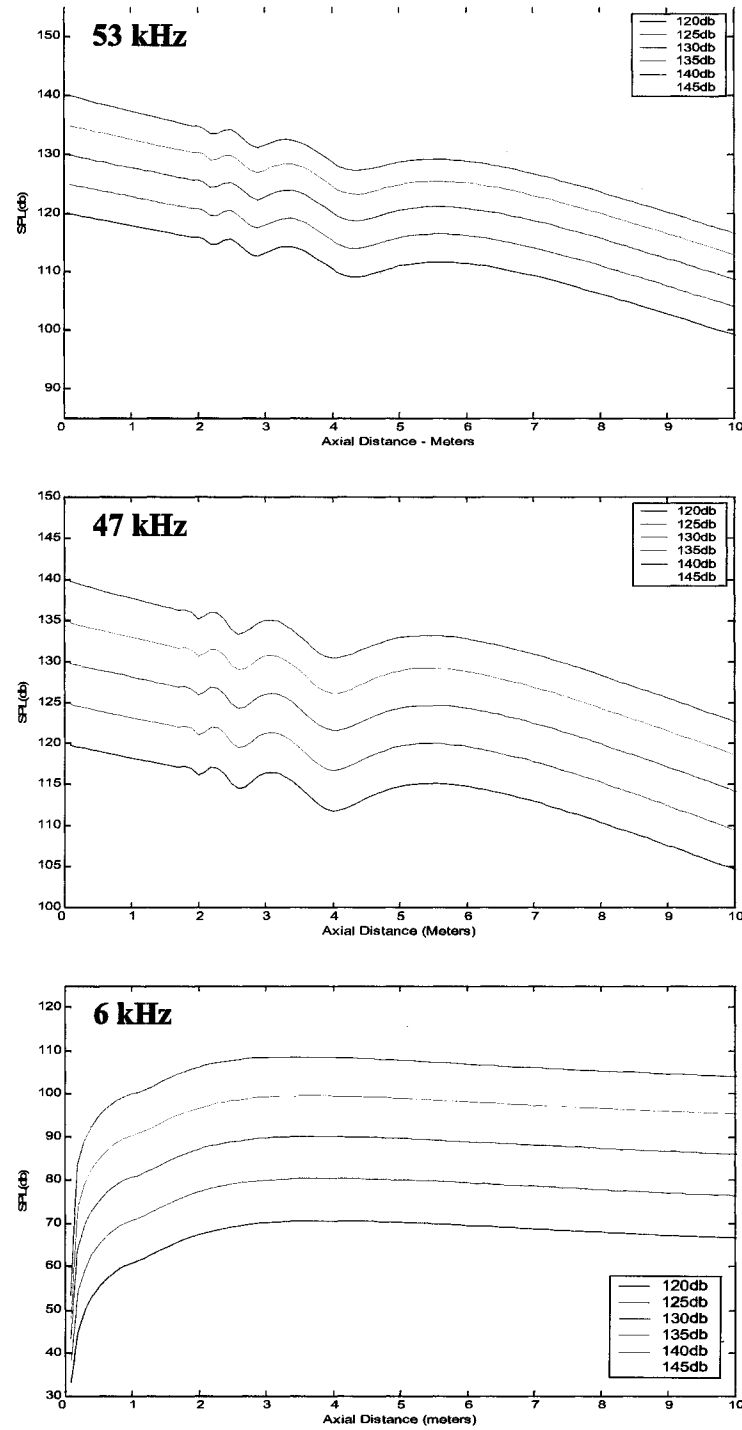
<b>Initial Sound Pressure</b>	<b>53 kHz Component</b>		<b>47 kHz Component</b>		<b>6 kHz Component</b>	
	<b>Intensity</b>	<b>B Width</b>	<b>Intensity</b>	<b>B Width</b>	<b>Intensity</b>	<b>B Width</b>
120 db	111.6dB	12 cm	115.1dB	12cm	69.9dB	59cm
125 db	116.5dB	12 cm	119.9dB	12cm	79.7dB	59cm
130 db	121.2dB	12 cm	124.6dB	12cm	89.2dB	59cm
135 db	125.5dB	12 cm	129.1dB	12cm	98.6dB	59cm
140 db	129.3dB	12 cm	133.1dB	12cm	107.3dB	59cm
145 db	132.1dB	12 cm	136.5dB	12cm	115.2dB	59cm

### Beam Width Profiles for the Different Initial Sound Pressure Intensities Measured at 5.8m



**Figure 4.15.** Beam width profiles recorded at 5.8m for the different initial sound pressure intensities. Changing the initial intensity does not affect the shape or width of the resulting sound beam.

## Axial Absorption Plots for the Different Initial Sound Pressure Intensities



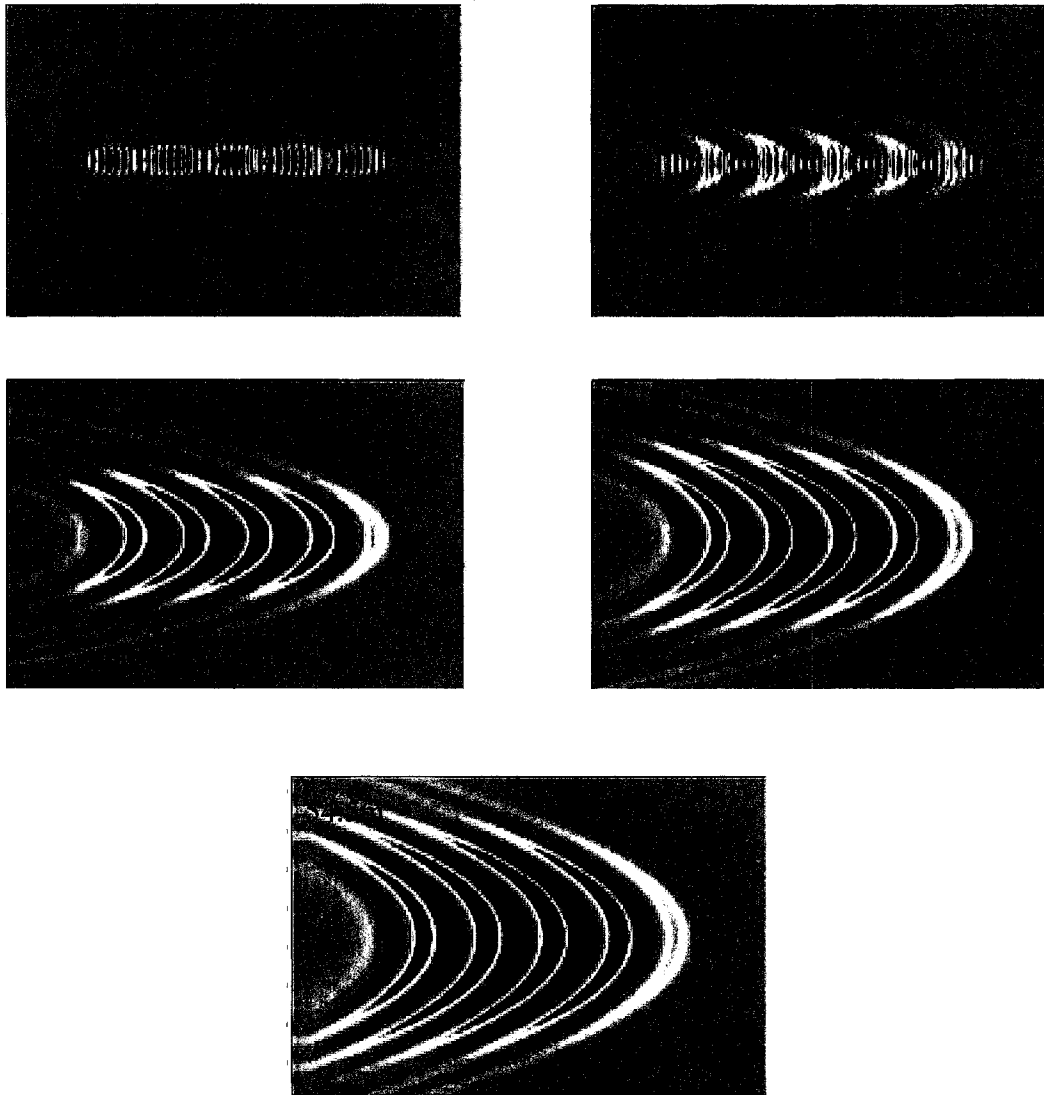
**Figure 4.16.** Axial absorption plots for the different initial sound pressure intensities. As the initial sound intensity is increased, more energy is passed to the nonlinear generated frequencies.

#### **4.2.5 Extended Distance Simulations**

Extended simulations have been performed to test the KZK acoustic code to larger distances such as 55 meters. Here we present an example of an extended-distance simulation using a 2ft diameter flat parametric transducer. The parametric transducer emits a short 140db dual frequency tone burst at 47 kHz and 53 kHz. Figure 4.17 shows the pressure waveform snapshots recorded at various distances starting at 3.2 meters and ending at 54.5 meters. Figure 4.18 shows the beam width profiles at these same distances.

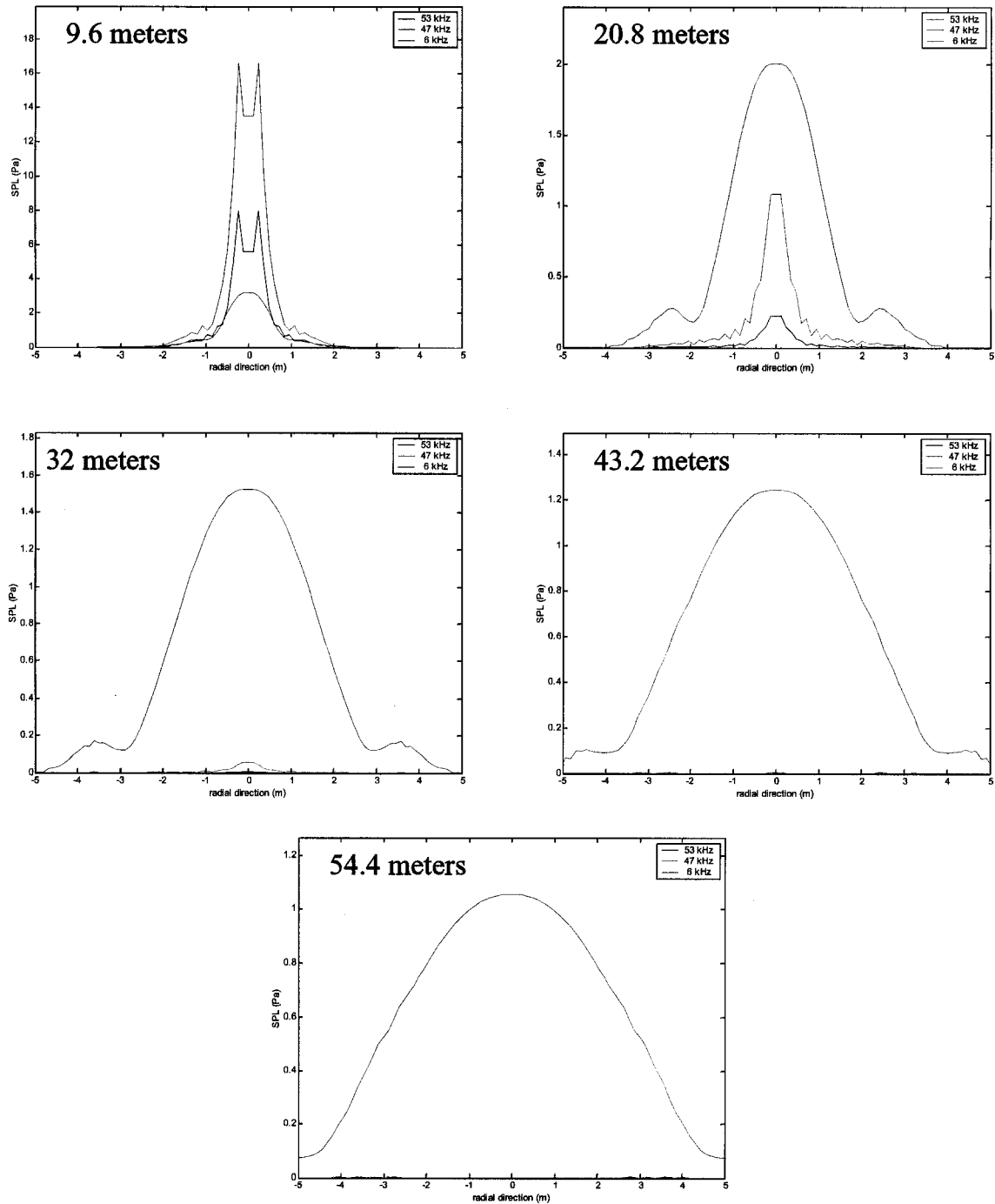
Figure 4.19 shows the absorption plots for the three frequency components of the sound beam. At 15-20 meters, the nonlinear generated difference frequency component becomes stronger than the ultrasound frequencies. At 50 meters, only the audio component is present. The beam width of the 6 kHz component of the sound beam is about 5 meters at 50 meters. The beam width can be reduced by using a larger transducer and/or employing focusing techniques.

### Pressure Waveform Snapshots from an Extended Distance Simulation



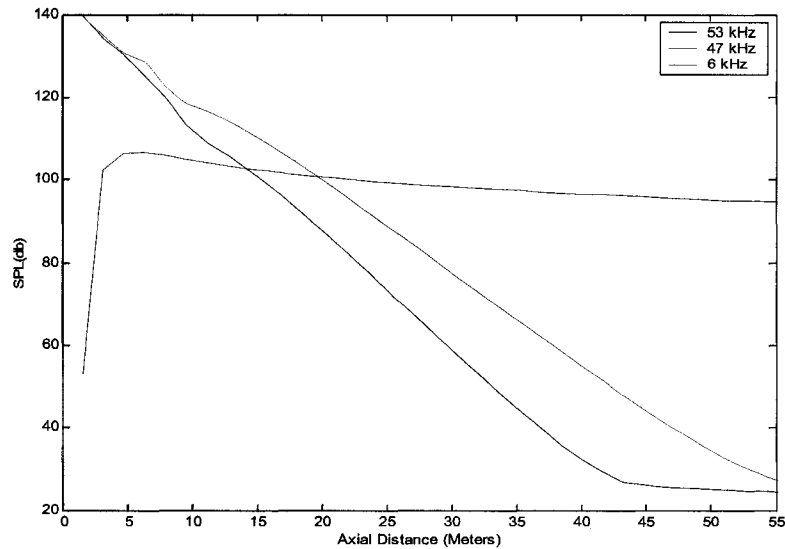
**Figure 4.17.** Pressure waveform snapshots recorded at various distances from an extended distance simulation.

## Beam Width Profiles at Various Distances for an Extended Distance Simulation



**Figure 4.18.** Beam width profiles at various distances for an extended distance simulation. The three different components of the sound beam are shown: 53 kHz (blue), 47 kHz (green), and 6 kHz (red).

### Absorption Plots for an Extended Distance Simulation



**Figure 4.19.** Absorption plots for an extended distance simulation to a distance of 55 meters. The three different components of the sound beam are shown: 53 kHz (blue), 47 kHz (green), and 6 kHz (red).

#### 4.2.5 Nonlinear Sound Beam Discussion

We have developed a useful and efficient tool for characterizing nonlinear acoustic beams created by various devices. With the new nonlinear KZK simulation code, many different transducer configurations can be quickly and systematically simulated. Simulations results have been shown to match well with experimental results.

We have simulated both parametric and cross beam transducer configurations to compare the resulting sound beams. The parametric array produces a narrow and uniform sound beam with a strong nonlinear difference frequency. The crossbeam geometry has a reduced interaction region that reduces the intensity of the nonlinear difference frequency. The cross beam geometry also produces a complicated interference

pattern in the interaction region of the two sound beams. This interference pattern may make characterizing and repeating experimental measurements difficult.

We now have a good understanding of the acoustic beams and their frequency components just before the waveforms interact with the target. Additional experimental, analytical, and computational studies should be performed to characterize the acoustic interaction with various sized and shaped objects. The interaction of the acoustic waves with an object depends strongly on the object's material composition, shape, size, and orientation. Understanding these interactions will be vital to developing a successful acoustic concealed weapons detector.



### **4.3 3DPAFIT Simulations with Incident Nonlinear Sound Beam**

#### **4.3.1 Nonlinear Sound Beam Input**

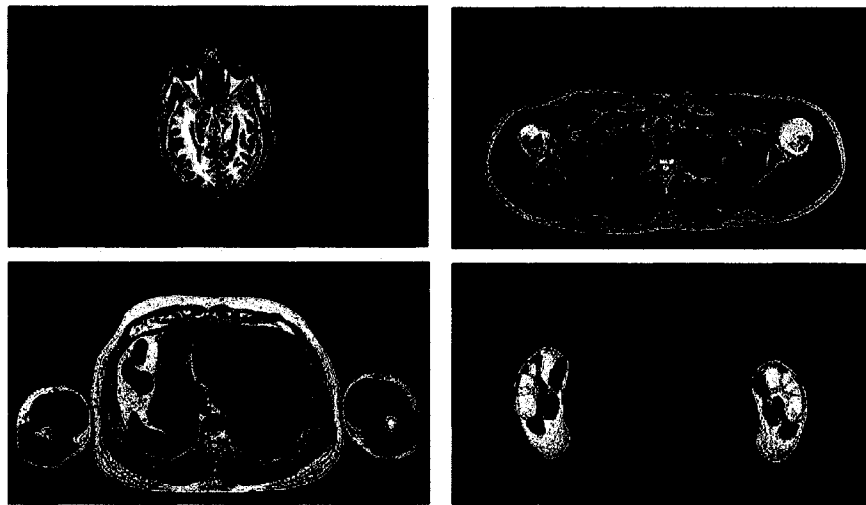
We have also merged the KZK simulation code and our 3D acoustic finite integration (3DPAFIT) code. The KZK simulation code is used to propagate the acoustic wave from the parametric array to the target taking into account the effects of nonlinearity and absorption. At a certain distance, beyond which the nonlinear conversion is complete, the pressure waves flow from the KZK simulation space into the 3DPAFIT simulation space. We can thus place any combination of objects and layers into the 3DPAFIT simulation space to study the acoustic interaction.

#### **4.3.2 Inserting and Scattering from a Human Model**

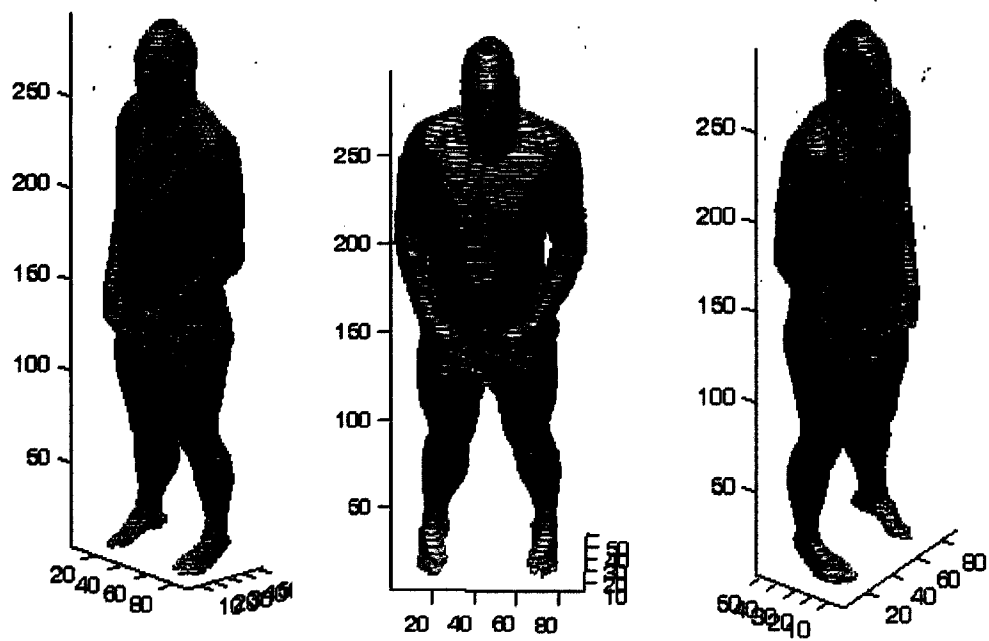
For concealed weapons and explosives detection, it is important to understand how acoustic waves interact with the human body with and without objects concealed under clothing layers. This is a very complicated scattering problem that is practically impossible to study theoretically. Our 3D parallel acoustic simulation technique (3DPAFIT) provides an accurate way to model these scenarios and study the acoustic interactions.

To create realistic scenarios, we have included a 3D human model into our acoustic scattering simulations. The 3D human model was created from low resolution anatomical cross-sections taken from a male cadaver obtained in the Visible Human Project [15]. Figure 4.20 shows four sample anatomical cross-sections from this dataset. Figure 4.21 shows three different views of our 3D human male model.

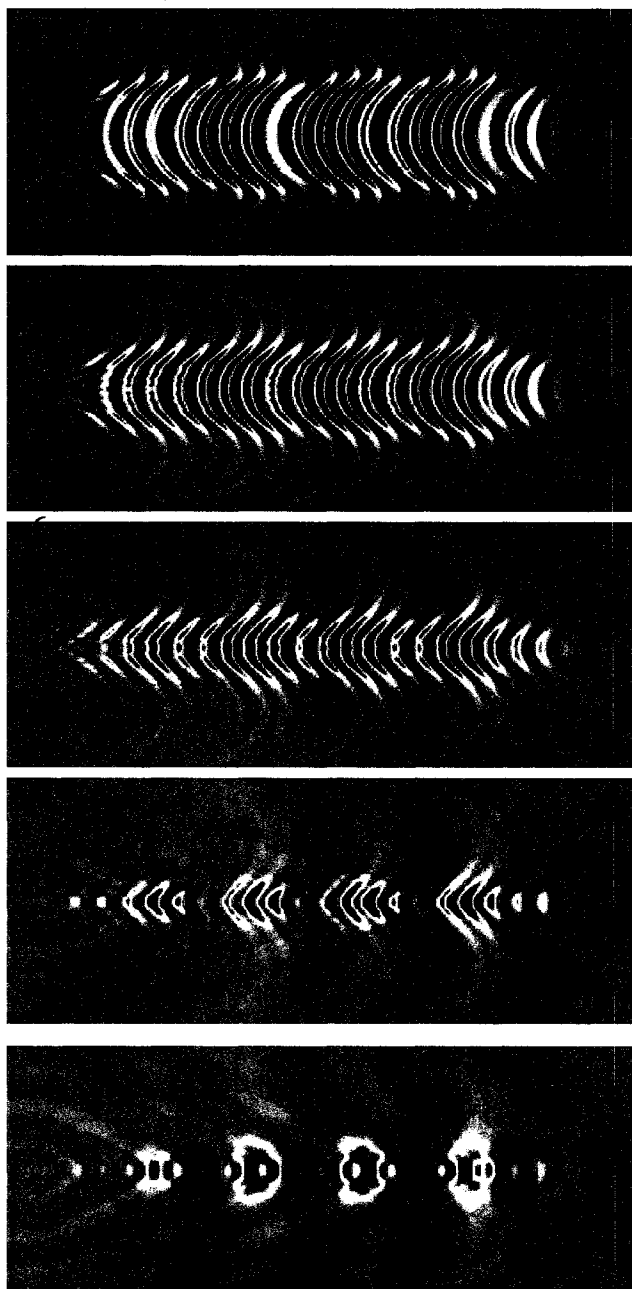
Figure 4.22 shows pressure waveform snapshots created with the KZK simulation code. A one meter diameter focused parametric array emits a short dual-tone pulse (45 kHz and 55 kHz). As the pulse propagates away from the parametric array, a 10 kHz component is created due to the nonlinearity of the air. At 10 meters, the 10 kHz component becomes the dominant frequency of the pulse because the higher ultrasound frequencies are absorbed much more quickly. At 10 meters, this waveform propagates into the 3DAFIT simulation space where it scatters from the 3D human model as shown in figure 4.23.



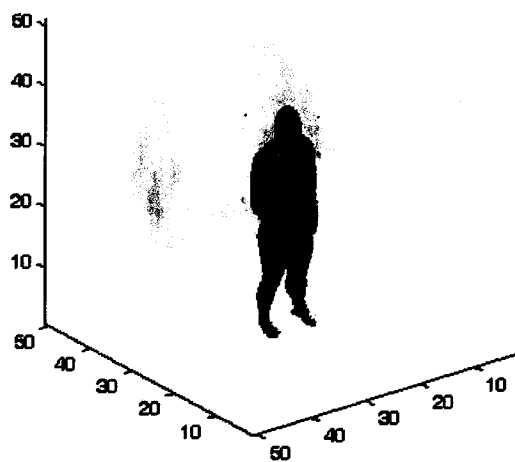
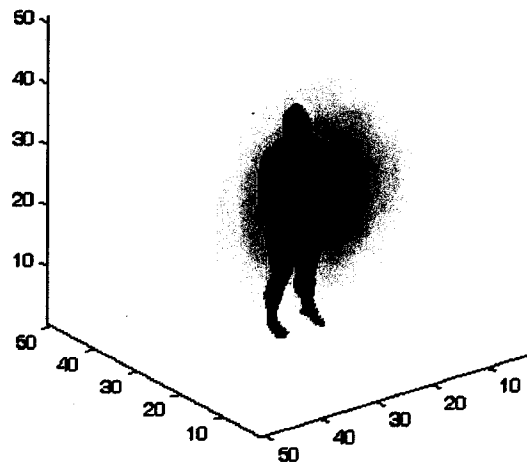
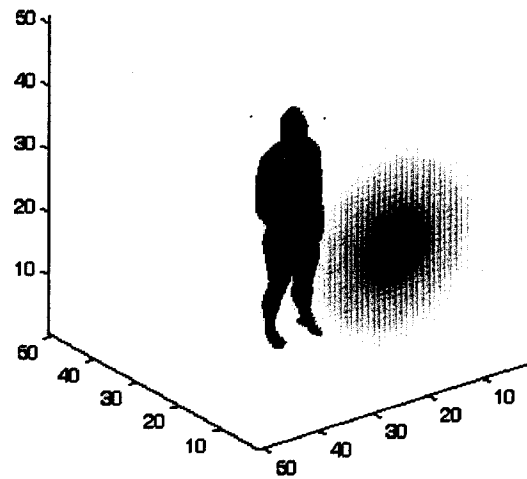
**Figure 4.20.** Anatomical cross-sections taken from a male cadaver obtained by the Visible Human Project.



**Figure 4.21.** Three different views of the 3D human male model created from the anatomical cross-sections.



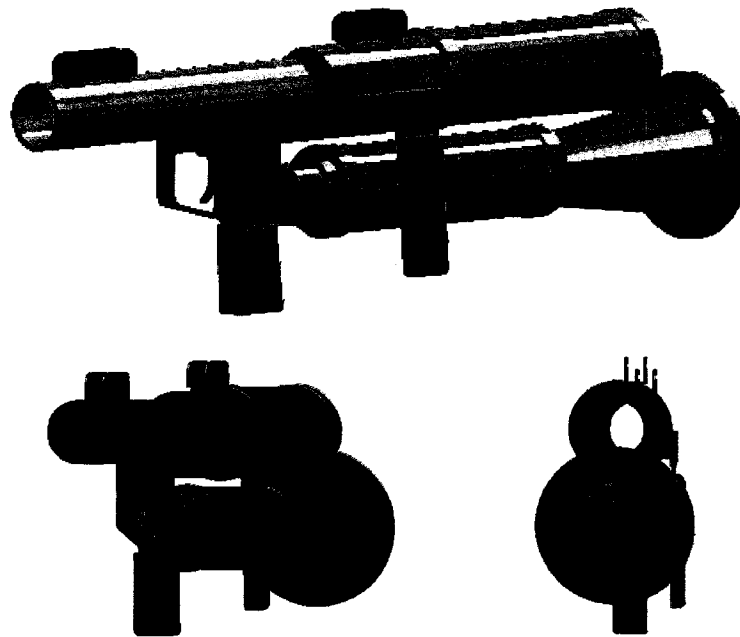
**Figure 4.22.** Pressure waveform snapshots created with the KZK simulation code. A one meter diameter focused parametric array emits a short 140db dual-tone pulse (45 kHz and 55 kHz). As the pulse propagates away from the parametric array, a 10 kHz component is created due to the nonlinearity of the air. The higher ultrasound frequencies are absorbed much quicker so that at 10 meters, only the 10 kHz component remains.



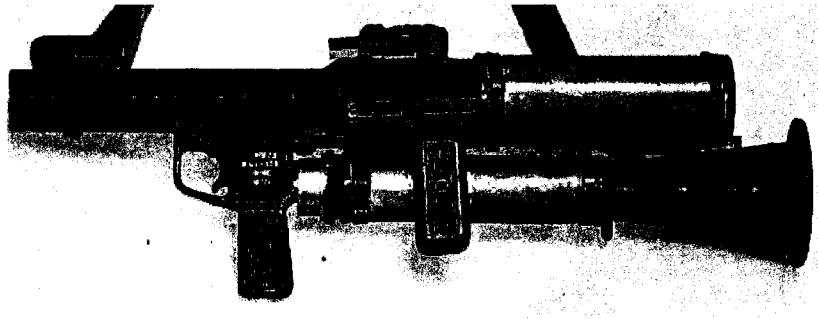
**Figure 4.23.** An acoustic pulse from the KZK simulation shown in figure 3 scatters from the 3D human model.

### 4.3.3 Inserting CAD Models

We have also created a 3D model of the RPG-7V1 Rocket Propelled Grenade Launcher. The 3D model is shown at several angles in Figure 4.24. The dimensions of each of the major geometrical features of the RPG were extrapolated from the image shown in Figure 4.25. The 3D model was created in a commercial CAD package and imported and rendered in the MATLAB environment. Once in the MATLAB environment, we can position it into our simulation space at any orientation to study how acoustic waves interact with it.



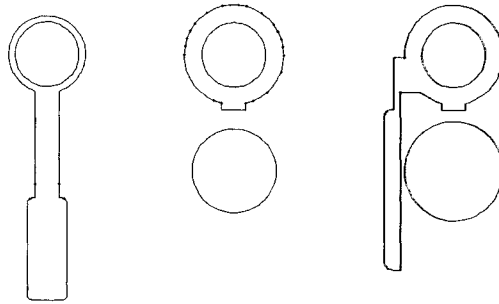
**Figure 14.24.** A 3D model of the RPG-7V1 Rocket Propelled Grenade Launcher is shown at three angles in the carrying position.



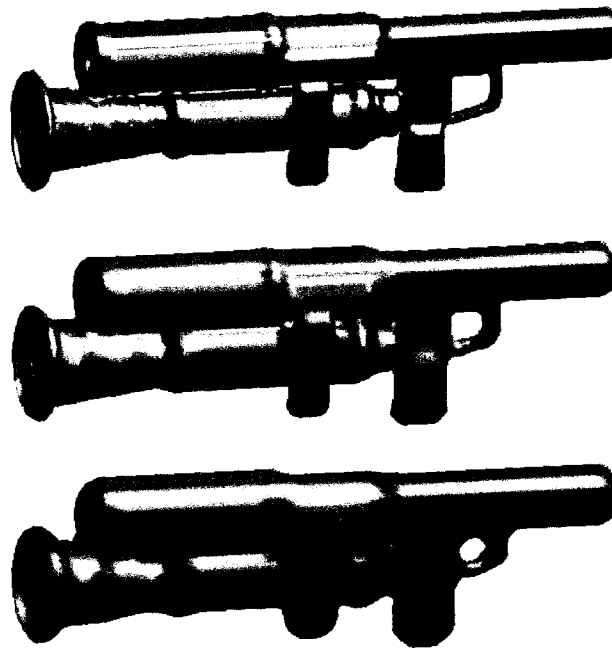
**Figure 4.25.** RPG-7V1 Rocket Propelled Grenade Launcher in carrying position.

Before the CAD model can be placed into the simulations space, it must be converted into the correct data format. To do this, we need cross sectional images similar to the anatomical cross sectional data from the human model. CAD models are composed of many triangles that fit together to create the surface of the 3D model. An algorithm has been developed to slice the 3D CAD model to create cross sectional images.

Sample cross sectional images of the RPG-7V1 are shown in figure 4.26. Many of these images are combined into the correct format so that the object can be inserted into the simulation space. This algorithm automatically adjusts the resolution of the object to fit the resolution of the simulation. Figure 4.27 shows the RPG-7V1 at three resolutions: 3mm, 6mm, and 9mm. It is easy to see that decreasing the resolution also decreases the level of detail of the RPG model. In most all cases, the resolution of the acoustic simulations will be 3mm or finer.



**Figure 4.26.** Cross-sectional images of the RPG-7V1 Rocket Propelled Grenade Launcher model. They are taken at the front handle (left), back handle (right) and in between the two (middle).



**Figure 4.27.** The RPG-7V1 Rocket Propelled Grenade Launcher model shown at three different resolutions: 3mm (top), 6mm (middle), and 9mm (bottom). The resolution of the 3D model must match the resolution of the simulation.



#### **4.3.4 Back Scattered Acoustic Energy as a Function of Angle**

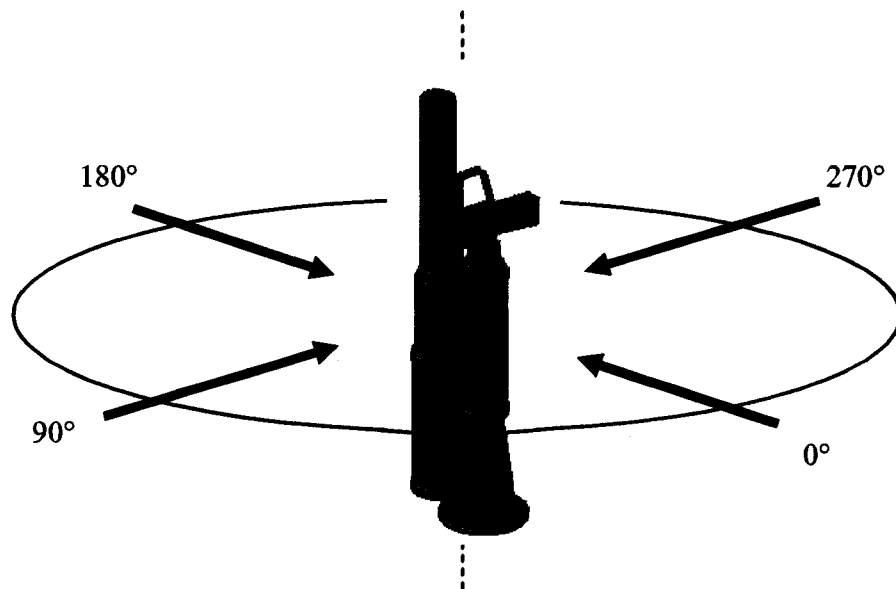
With the ability to add 3D CAD models into the computational space, we can systematically explore different scattering scenarios. These simulations will provide valuable information about how acoustic energy scatters with various objects and clothing layers. This information will be key in the development of signal processing algorithms and next generation hardware.

As an example, we placed the RPG model into the simulation and systematically rotated it to determine the back-scattered energy as a function of incident beam angle. The incoming sound wave is a 6 kHz sound beam that is the result of a nonlinear KZK simulation. In this simulation, a 1 meter focused parametric array emits a dual tone burst (50 kHz and 56 kHz). The waves are focused as they propagate and a 6 kHz difference frequency wave becomes the dominate component in the wave. At 10 meters, the sound waves flow from the nonlinear acoustic simulation into the 3D acoustic scattering simulation. The waves then scatter from the RPG and the backscattered energy is recorded.

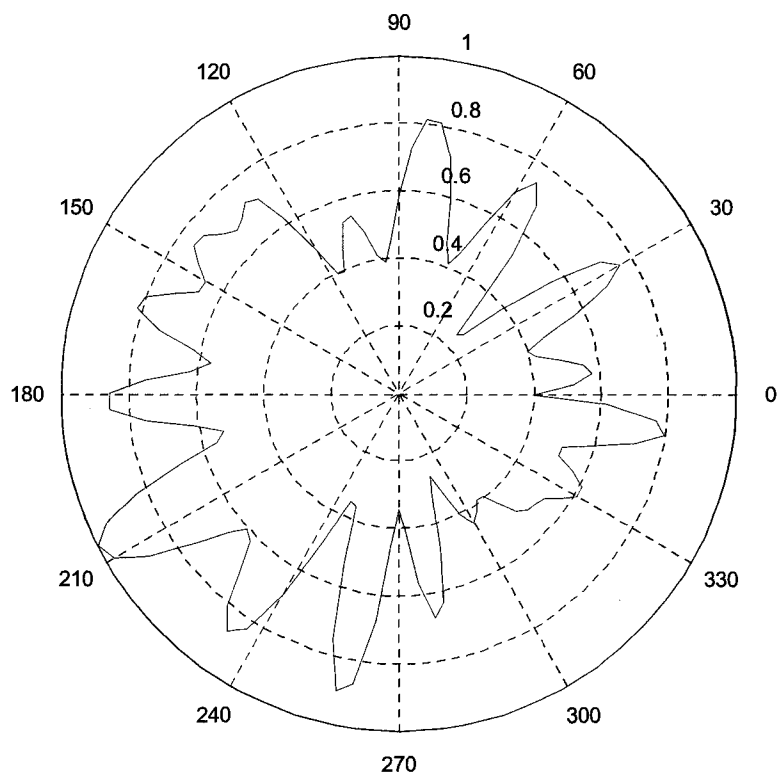
This process was carried out for 40 different orientations of the RPG model. Figure 4.28 shows the orientation of the RPG along with the incident wave angles. The RPG was rotated 9 degrees between each simulation. This process provides the backscattered acoustic energy as a function of angle for the RPG model at 6 kHz. This distribution can be found in the polar plot shown in Figure 4.29. The strongest back scattered energy is found between 120° and 240°. At these angles, both handles of the

RPG are facing the incoming sound beam which strongly reflects the incoming acoustic energy.

The peak and valley structure of the backscattered energy is as expected and is perhaps the most interesting part of the data. In some cases, a small change in orientation (as small as  $10^\circ$ ) can lead to a 60% drop in the backscattered energy. This plot only shows the backscattered energy at 6 kHz, but a sequence of these backscattered-energy-vs.-angle plots at different frequencies can provide a template of what to look for in experimental measurements. Knowing how acoustic waves reflect from complicated shaped weapons is necessary to develop a robust signal processing algorithms to automatically detect them.



**Figure 4.28.** The RPG-7V1 model is shown with angles of the incident sound beam.



**Figure 4.29.** Polar plot shows the backscattered energy of the RPG-7V1 model as a function of incident wave angle.

#### **4.4 Conclusion**

With the KZK nonlinear sound beam simulations, we can explore how variables such as parametric array size, curvature, initial waveform, and intensity level affect the resulting sound beams. We can also use these simulations to predict how parametric arrays will perform under different environmental factors such as air temperature and humidity level. With our 3D acoustic simulations (3DPAFIT), we can study how acoustic waves interact with complex shaped objects such as the human figure, clothing layers, and concealed weapons. We can also explore how the material composition of objects affects the scattered acoustic energy.

We have coupled our KZK nonlinear acoustic simulations with our 3D acoustic scattering simulations. Together we can simulate the entire process of nonlinear sound propagation and acoustic scattering from the target. We have the full capabilities to systematically simulate acoustic interactions with complex shaped objects including humans, clothing layers, and weapons. These simulations provide a very powerful tool to assist in the development of hardware and software systems for the next generation acoustic concealed weapons detector.

#### 4.5 References

1. Inc, J., *Handheld Remote Concealed-Weapons Detector, Final Technical Report*. National Institute of Justice, 1999. #J200-99-0032/3031.
2. A. Achanta, M.M., S. Guy, E. Malyarenko, J. Lynch, J. Heyman, K. Rudd, M. Hinders, *Nonlinear Acoustic Concealed Weapons Detection*. Materials Evaluation, 2005. **63**(12): p. 1195-1202.
3. A. Achanta, J.H., K. Rudd, M. Hinders, P. Costaines, *Non-linear Acoustic Concealed Weapons Detection System*. Automatic Target Recognition XV, Proc. SPIE, 2005. **5807**: p. 162-169.
4. A. Achanta, J.H., M. McKenna, K. Rudd, M. Hinders, *Nonlinear Acoustic Concealed Weapons Detection*. Proceedings of the 34th Applied Imagery Recognition Workshop IEEE, 2005.
5. Lee, Y., *Numerical Solution of the KZK Equation for Pulsed Finite Sound Beams in Thermoviscous Fluids*. 1993, The University of Texas Austin.
6. Kuznetsov, V.P., *Equations of Nonlinear Acoustics*. Sov. Phys. Acoust, 1971. **24**: p. 310-313.
7. Zabolotskaya, E.A.a.K., R.V., *Quasi-plane waves in the nonlinear acoustics of confined beams*. Sov. Phys. Acoust, 1969. **15**: p. 35-40.
8. Y. Lee, M.F.H., *Time-domain Modeling of pulsed finite amplitude sound beams*. The Journal of the Acoustical Society of America, 1995. **97**(2): p. 906-917.
9. M.F. Hamilton, D.T.B., *Nonlinear Acoustics*. 1998: Academic Press.
10. H.E. Bass, L.C.S., A.J. Zuckerwar, D.T. Blackstock, and D.M. Hester, *Atmospheric absorption of sound: Further developments*. Journal of Acoustical Society of America, 1995. **97**(1): p. 680-683.
11. L. Kinsler, A.F., A. Coopens, and J. Sanders, *Fundamentals of Acoustics*. 4th ed. 2000: John Wiley & Sons, Inc. 162-63.
12. Hamilton, M.F., *Effects of noncollinear interaction on parametric acoustic arrays in dispersive fluids*. Journal of Acoustical Society of America, 1984. **76**(5): p. 1493-1504.
13. M. Fatemi, A.A., J.F. Greenleaf, *Characteristics of the audio sound generated by ultrasound imaging systems*. Journal of Acoustical Society of America, 2005. **117**(3): p. 1448-1455.
14. M. Fatemi, J.F.G., *Vibro-acoustography: An imaging based on ultrasound-stimulated acoustic emission*. The National Academy of Science USA, 1999. **96**: p. 6603-6608.
15. Project®, V.H., [http://www.nlm.nih.gov/research/visible/visible\\_human.html](http://www.nlm.nih.gov/research/visible/visible_human.html).

## Chapter V

### Applied 3DPAFIT Simulations: Ultrasonic Periodontal Probe

#### 5.1 Introduction

Periodontal disease refers to the inflammatory process of the tissues surrounding the teeth due to bacterial accumulations. If untreated, periodontal disease can lead to a progressive loss of tissue attachment to the tooth and underlying alveolar (jaw) bone and ultimately lead to tooth loss [1]. The periodontal pocket is the crevice that forms in between the tooth and the supporting tissue. Depending on depth of the pocket, which can extend from 2 to 12 mm, it can harbor as much as  $10^7$  to  $10^9$  bacterial cells [1, 2]

Half of the adult population in the United States has mild inflammation (gingivitis), and about 30% of the population has periodontal disease that is defined by having three or more periodontal pockets with depths of 4mm or more [3-7]. Between 5% and 15% of adults with periodontal disease have advanced forms with pocket depths measuring 6mm or more [8]. Periodontal disease has also been associated with diabetes, stroke, and adverse pregnancy conditions [9-11]. At least 14 of 17 studies have provided statistically significant data to associate periodontal disease with cardiovascular disease [12-14]. This suggest the possibility of periodontal disease as a risk factor for cardiovascular disease [1].

Today, periodontal disease is typically diagnosed with manual probing [15]. A clinician determines the depth of the periodontal pocket by inserting a thin metal probe

directly into the pocket. Ridges or markings on the probe indicate for depth of the pocket. Manual probing has been shown to be unreliable [16-22] and it can be invasive and painful. Studies in automatically-controlled force probes have shown success in reducing operator-related error and subjectivity inherent in manual probing [23-26], but these probes do not account for anatomic and inflammatory factors that can affect on measurement accuracy [27, 28]. New techniques and technologies based on ultrasound may be able to diagnose periodontal disease more reliably than manual techniques while being less invasive and painful to the patient.

#### **5.1.1 The Ultrasonic Periodontal Probe**

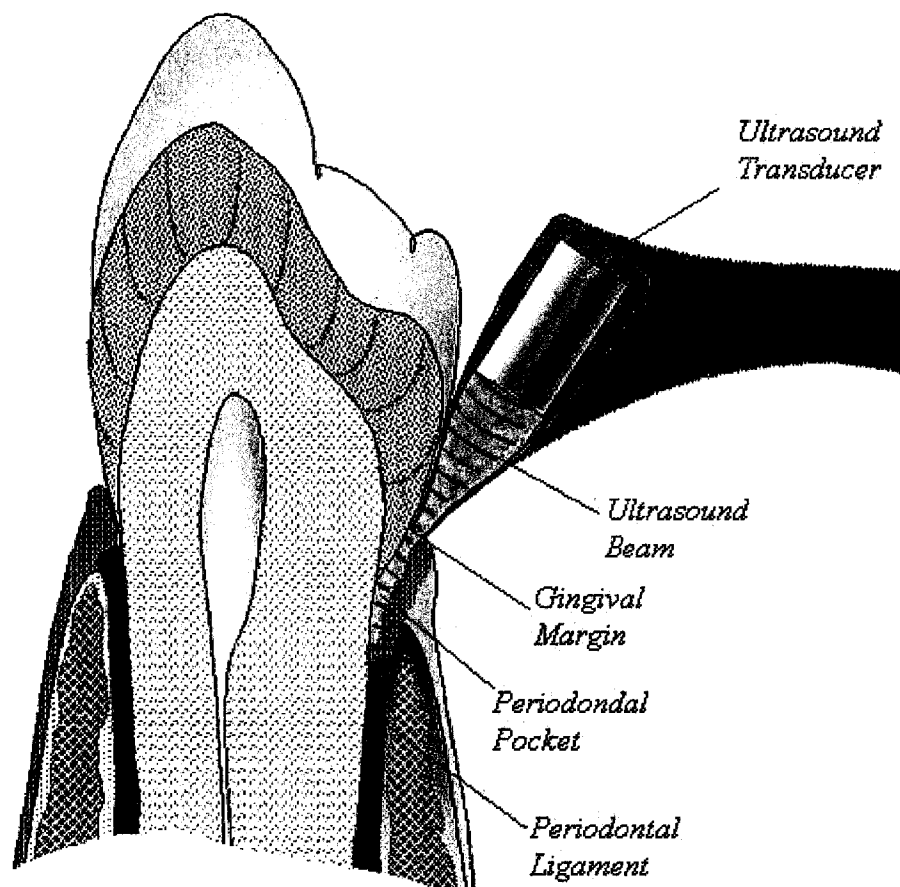
Over the past four decades, many researchers have explored the use of ultrasound to image the periodontal region [29-38]. In 1998, Loker and Hagenbuch developed a prototype of an ultrasonic device to measure the depth of the periodontal pocket [39]. Their device used a solid taper-delay line to couple the ultrasound into the tissue at approximately the same location and orientation as manual probing [40]. Results from a pilot clinical trial showed that correlation between measurements taken by manual probing and with their ultrasonic device were “not particularly good”.

Also in 1998, Companion and Hinders [41, 42] first reported results of an ultrasonic periodontal probe that had been under development at NASA Langley for several years [43, 44]. Hinders et. al reported on various aspects of this work [45-49] as it developed over the next several years. This ultrasonic periodontal probe, which is the basis of the simulations in this chapter, uses a hollow tapered tip that is filled with water for coupling of the ultrasonic beam into the tissues. A diagram of this ultrasonic probe

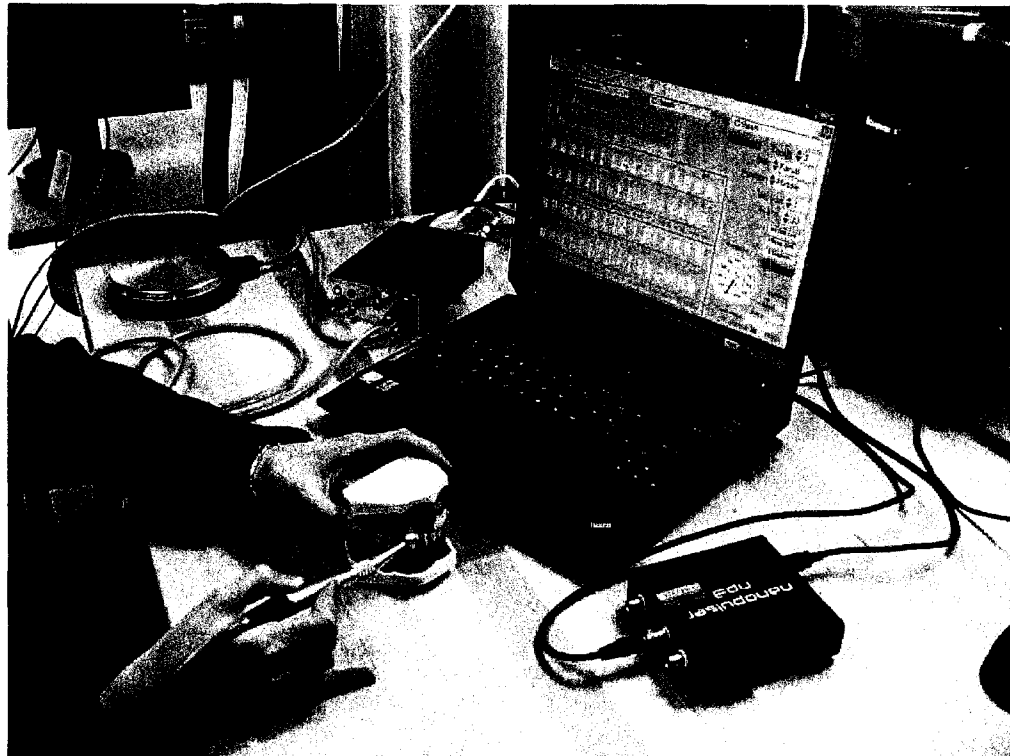
and the geometry of the periodontal tissues are shown in Figure 5.1. The internal shape of the hollow probe tip was optimized via a combination of computer simulations and systematic experiments, and a sequence of increasingly more practical clinical prototypes were developed and used in several pilot studies comparing ultrasonic to calibrated-manual and controlled-force probing. Pictures of the latest generation of the ultrasonic periodontal probe including the water flow system, ultrasonic probe, and the data acquisition system is shown in Figure 5.2 and Figure 5.3. A critical development was the recognition of the need for artificial intelligence algorithms to automatically identify the very subtle echo-waveform features corresponding to the anatomy of interest. The wavelet fingerprint technique of Hou and Hinders [50-55] was adapted for this purpose and shows promise.

In this chapter, we use the three-dimensional parallel acoustic finite integration technique (3DPAFIT) to simulate the ultrasound propagation in the tip and the intricate geometries periodontal tissues. These simulations provide valuable insight into the complex underlying physics of the ultrasound propagation and interaction in the soft-tissues. A sophisticated software package was developed to automatically define the 2D and 3D geometry of the tip and the periodontal tissue structures and allow for easy modifications of these geometries. Many simulations were completed to provide systematic data sets to assist in the development of automated software algorithms for determining the periodontal pocket depth under a variety of conditions.

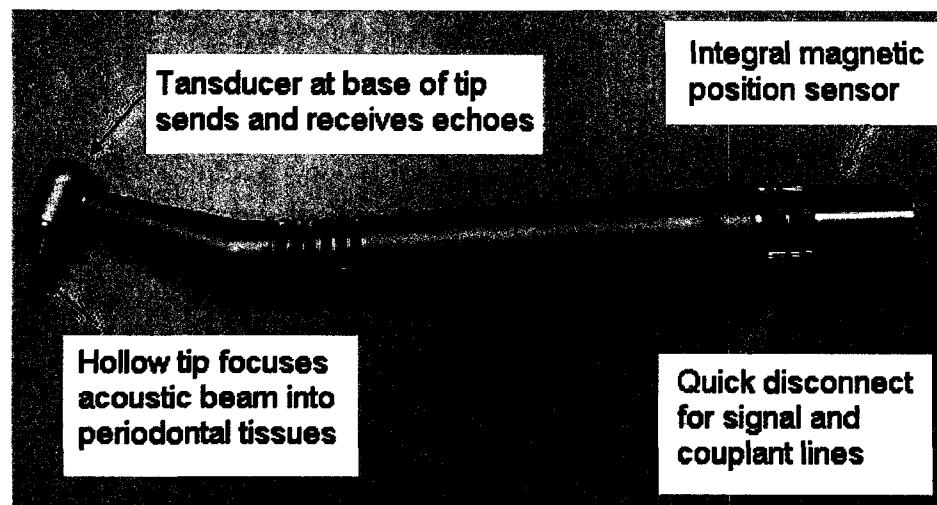




**Figure 5.1.** Diagram of the ultrasonic periodontal probe and the major tissue structures of the periodontal region.



**Figure 5.2.** Picture of the latest generation of the Ultrasonic Periodontal Probe System.



**Figure 5.3.** Close-up picture of the Ultrasonic Periodontal Probe hand piece.

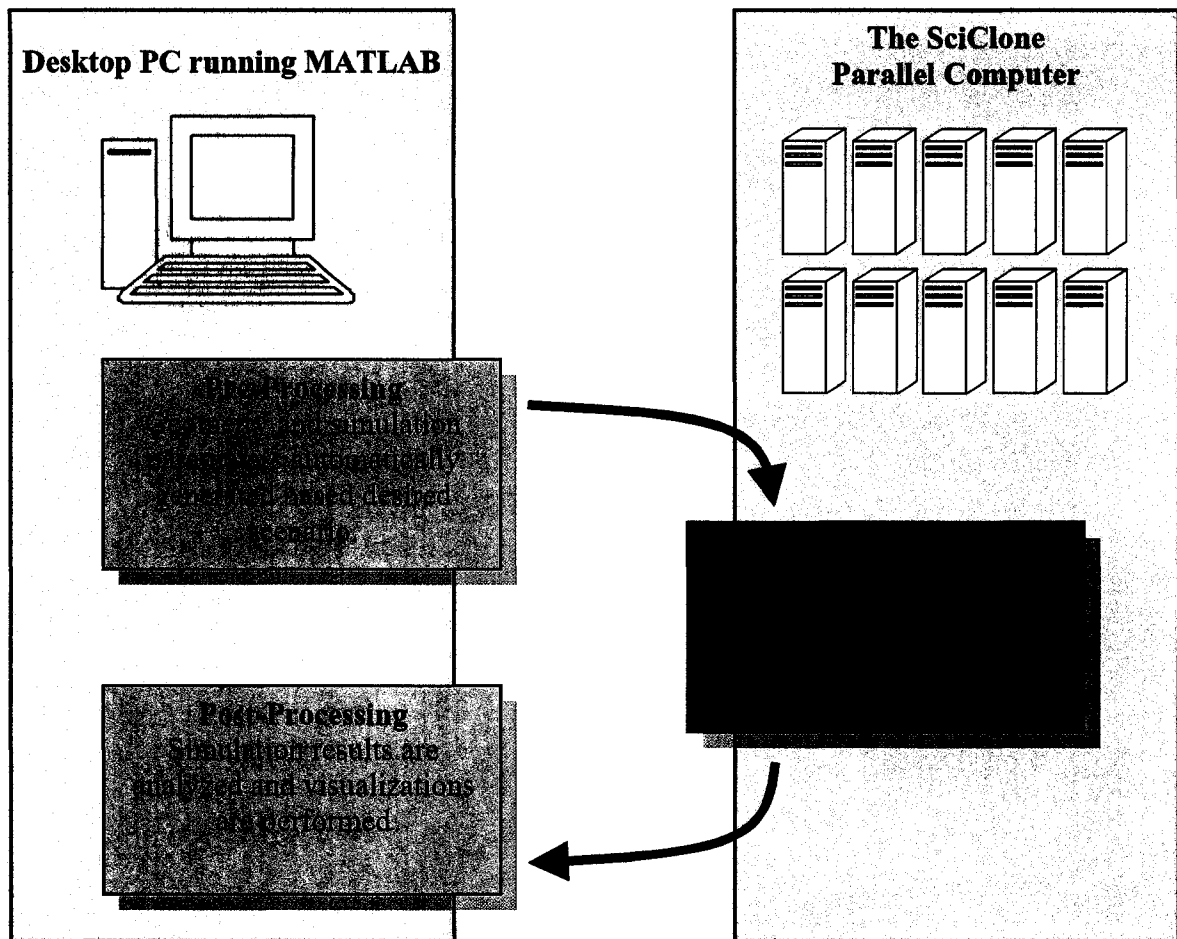
## **5.2 Acoustic Simulations of the Ultrasonic Periodontal Probe**

A sophisticated set of software tools were developed to simulate the 3D acoustic propagation and interaction in the tip and the periodontal tissues. These software components automatically create the 3D geometries of interest, perform actual acoustic simulations on a parallel super-computer, and process and visualize the results. The pre-processing and post-processing, including visualizations, are performed on a single desktop computer using the MATLAB programming environment. The actual simulations are performed on William and Mary's high performance computational cluster (The SciClone). A flow chart of the entire process is shown in Figure 5.4.

First, a software system was developed to automatically create the 2D periodontal geometry using a small number of parameters which define the scenario the user wants to simulate. Most of the important features of the model are parameterized so that they can be changed without having to modify the simulation software. These include the geometry of the tissue structures, material parameters, the depth of the periodontal pocket, and the curvature of the tooth and tissue structure. The ultrasonic tip is also parameterized so that its shape, the angle in which it sits on top of the periodontal pocket, and the size and frequency of the transducer can also be changed. Once these parameters are set, the MATLAB software automatically creates the appropriate 2D models which ultimately define the 3D geometry. Then, a set of input files are created that define all the simulation parameters and geometries for the 3D acoustic simulation code.

These input files are then moved to the SciClone where the 3D parallel acoustic simulations are performed. As the simulation runs, the simulation software computes and records a variety of simulation values such as acoustic pressure inside the tissue. The

output files include sets of 2D pressure slices which show the acoustic wave propagation and typical A-line data that is recorded across the front of the transducer face. More details of the software components and the periodontal and tip geometries will be discussed in the following sections.



**Figure 5.4.** Chart showing the flow of data from the major software components for simulating 3D acoustic waves for the ultrasonic periodontal probe.

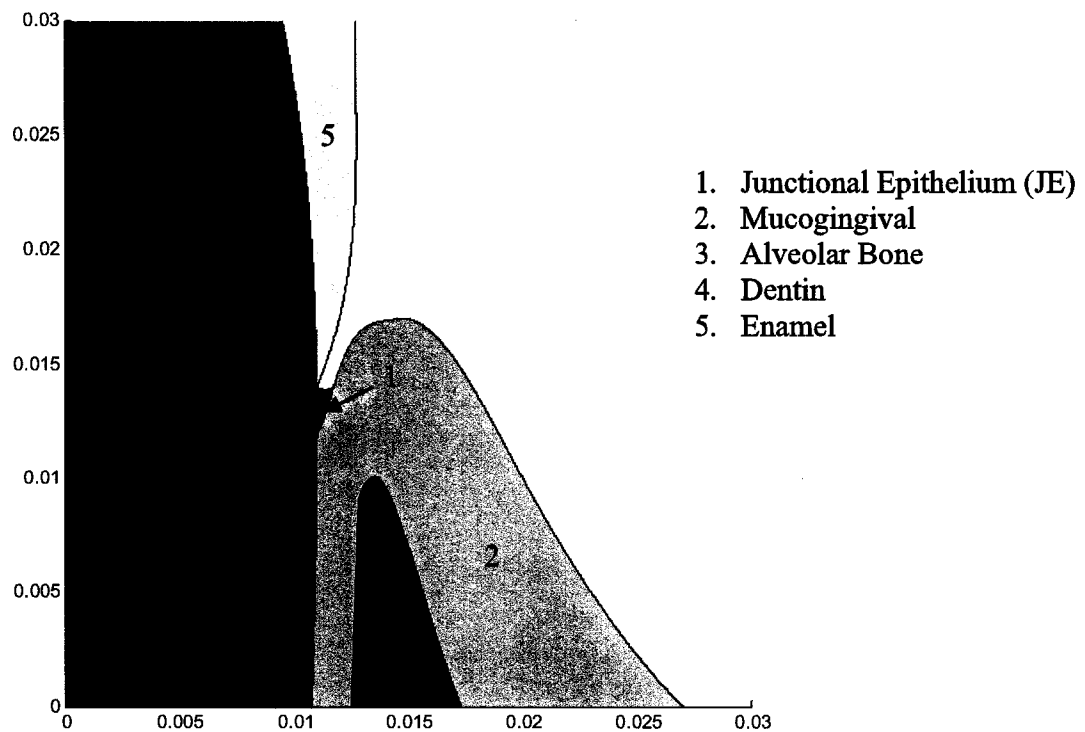
### 5.3 Two-Dimensional Periodontal Tissue and Tip Geometry

The geometry of the periodontal pocket and surrounding tissues is very complex. The shape and material properties of the tooth and tissues can vary from tooth to tooth and from patient to patient. We have created a 2D geometry that is based on several sources including real anatomical cross-sections and diagrams from leading periodontal disease textbooks [ref]. Samples of an anatomical cross-sections and a diagram of the anatomical area of interest are shown in figure 5.5.

A 2D model of the periodontal pocket and surrounding tissue is shown in figure 5.6. It includes the major anatomical features that are important to periodontal disease development and the ultrasound propagation. The model includes the hard tissues of the dentin, tooth enamel, and alveolar bone. It also includes the soft tissues of the mucogingival which makes up most of the major soft tissue at the base of the tooth. The junctional epithelium (JE) is located at the base of the periodontal pocket.



**Figure 5.5.** Anatomical cross-section and a diagram of the periodontal pocket and surrounding tissues that were used to create the 2D periodontal model.

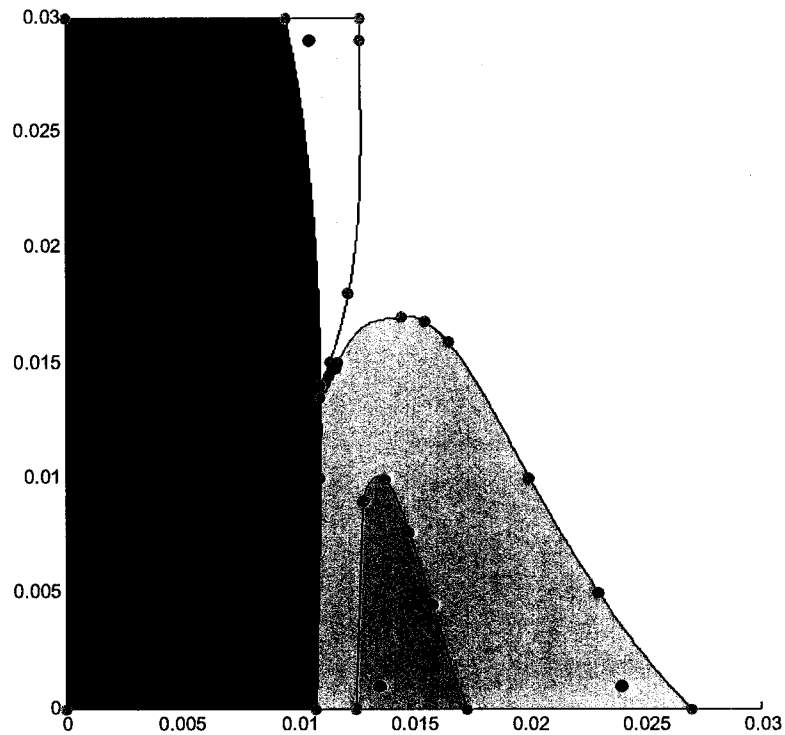


**Figure 5.6.** 2D slice of the 3D periodontal model.

The 2D model maps the geometry of the different tissue structures. From the beginning of development, it was assumed that the 2D model will have to be adaptive instead of a static 2D image. With the advice from clinicians and experts in the field of periodontics, the model can be continually improved in accuracy. We can also purposely change the model to test a variety of geometries and pocket depths. In addition, the 2D model needs to be scalable depending on the resolution of the simulation. For example, a 5 MHz simulation needs a model that is 5 times higher in resolution than a 1 MHz simulation. To perform these changes on a static image would be difficult to accomplish in a timely manner. The details of the adaptable 2D model of the periodontal tissue structures are found in the following section.

### 5.3.1 Adaptable 2D Periodontal Tissue Model

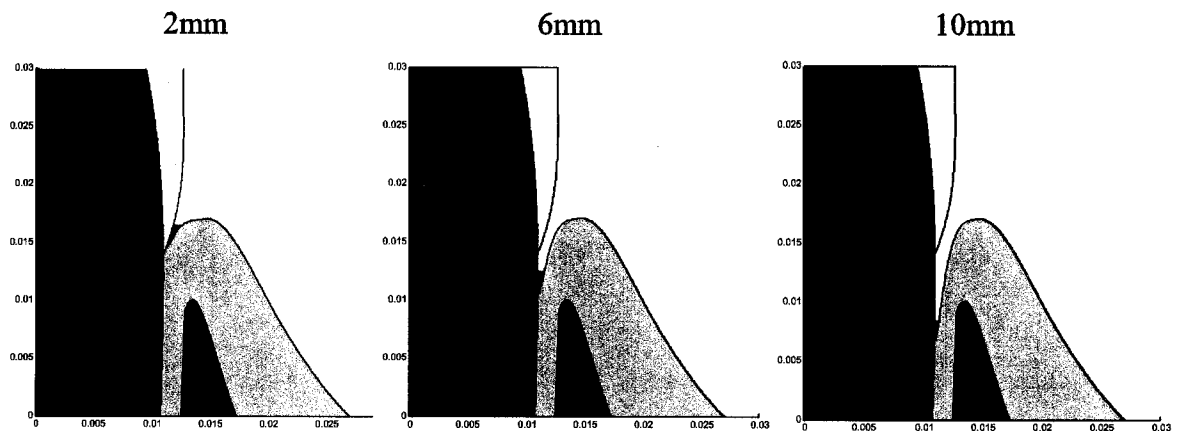
A software system was developed so that the entire 2D model is defined by a small set of points that indicate the boundaries between the different tissue structures. Figure 5.7 shows the 2D model with a set of green points on the tissue boundaries. Once the resolution of the simulation space is determined, the rest of the boundaries are found using a cubic-spline interpolation between the major points. For example, the position, size, and shape of the alveolar bone are determined by only 5 points. The entire geometry of the model can be modified by simply moving the locations of these points or adding new ones. Once the boundaries are defined, the material parameters are literally filled in starting at the location of the red/blue dots.



**Figure 5.7.** The 2D geometry of the periodontal tissue structures is defined by small set of points (shown in green), which indicate the boundaries between the different tissues. The entire geometry can be changed by moving these points or adding new ones.

### 5.3.2 Adding the Periodontal Pocket

Once the major geometry is established, the periodontal pocket is created automatically by specifying the depth of the pocket. For simplicity, the depth of the periodontal pocket is defined as the vertical distance from the top of the gum to the location of the junction between the gum and the tooth. The junctional epithelium (JE) is then positioned in the bottom of the pocket with a vertical height of 1.5mm (this can be changed). Then the pocket is completely filled with water (which is not shown in most of the figures). Figure 5.8 shows the 2D model with the periodontal pocket depth of 2mm, 6mm, and 10mm.



**Figure 5.8.** The 2D periodontal model is shown for three different pocket depths.



### 5.3.3 Two Dimensional Tip Construction and Placement

Like the periodontal pocket, the tip of the ultrasonic probe is modeled using a 2D cross section. Previous work suggested that a tip with linear sloped walls would be the most effective shape to deliver the ultrasound energy into the periodontal pocket [48]. A 2D model of a linear tip is shown in figure 5.9. It is parameterized such that one can easily change the dimensions of the tip. These dimensions include base radius, transducer radius, wall thickness, length, and tip radius. Additional tip shapes can be modeled by specifying their cross sectional shape.

Once the shape of the tip is defined, it is placed into position at the top of the periodontal pocket. The angle of the tip is also parameterized so that it can be easily changed. The placement of the tip is completely automated. Figure 5.10 shows the tip of the periodontal ultrasound probe at 65°, 50°, and 30°.

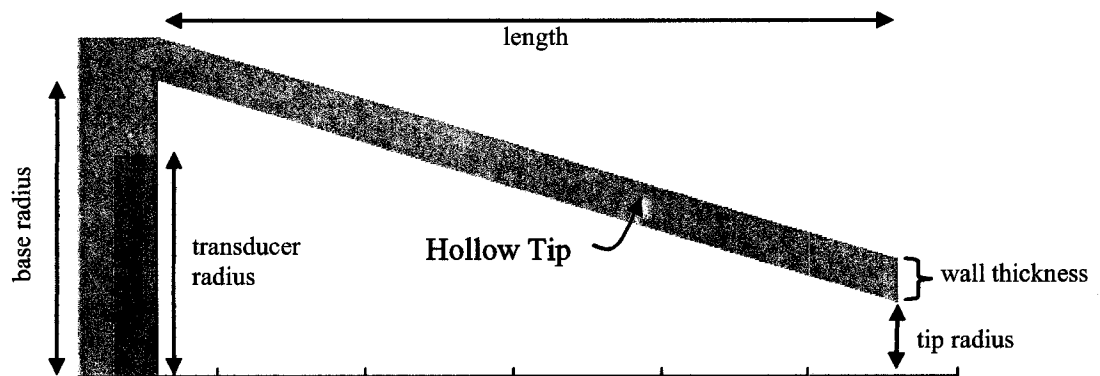
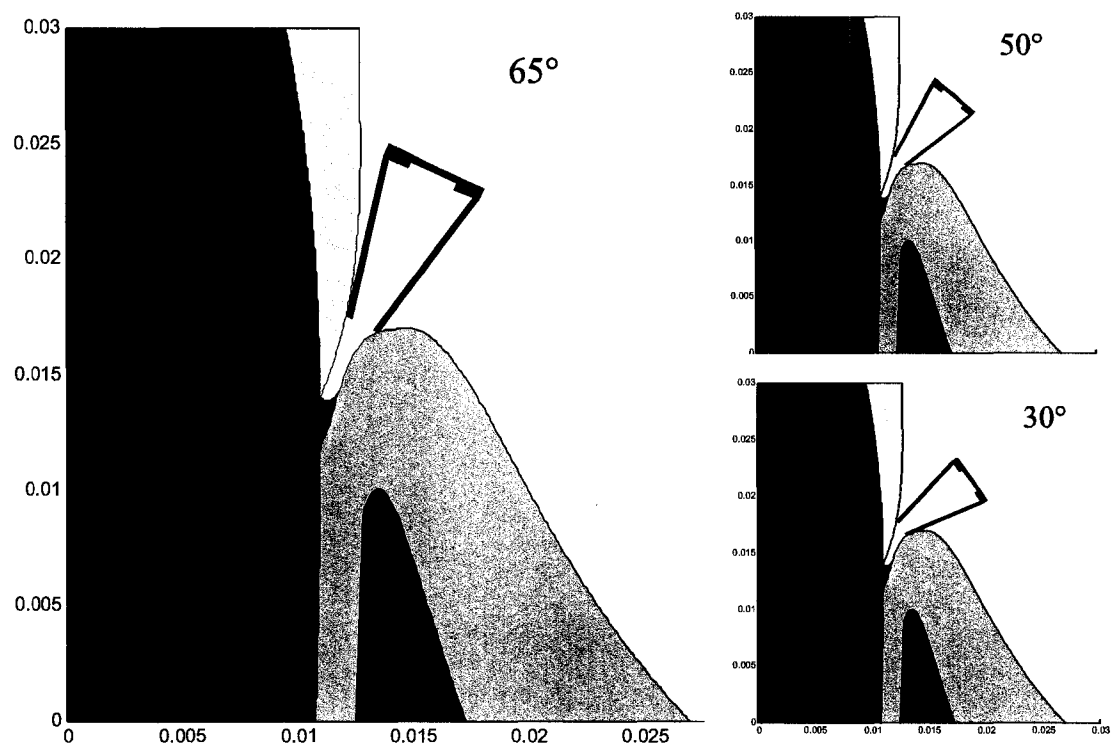


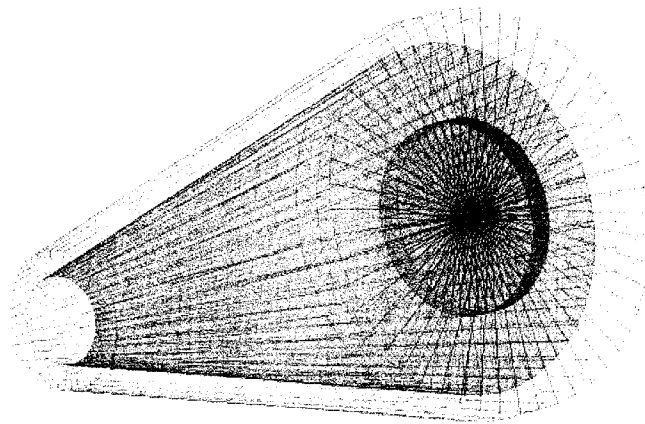
Figure 5.9. A linear tip and its parameterized dimensions.



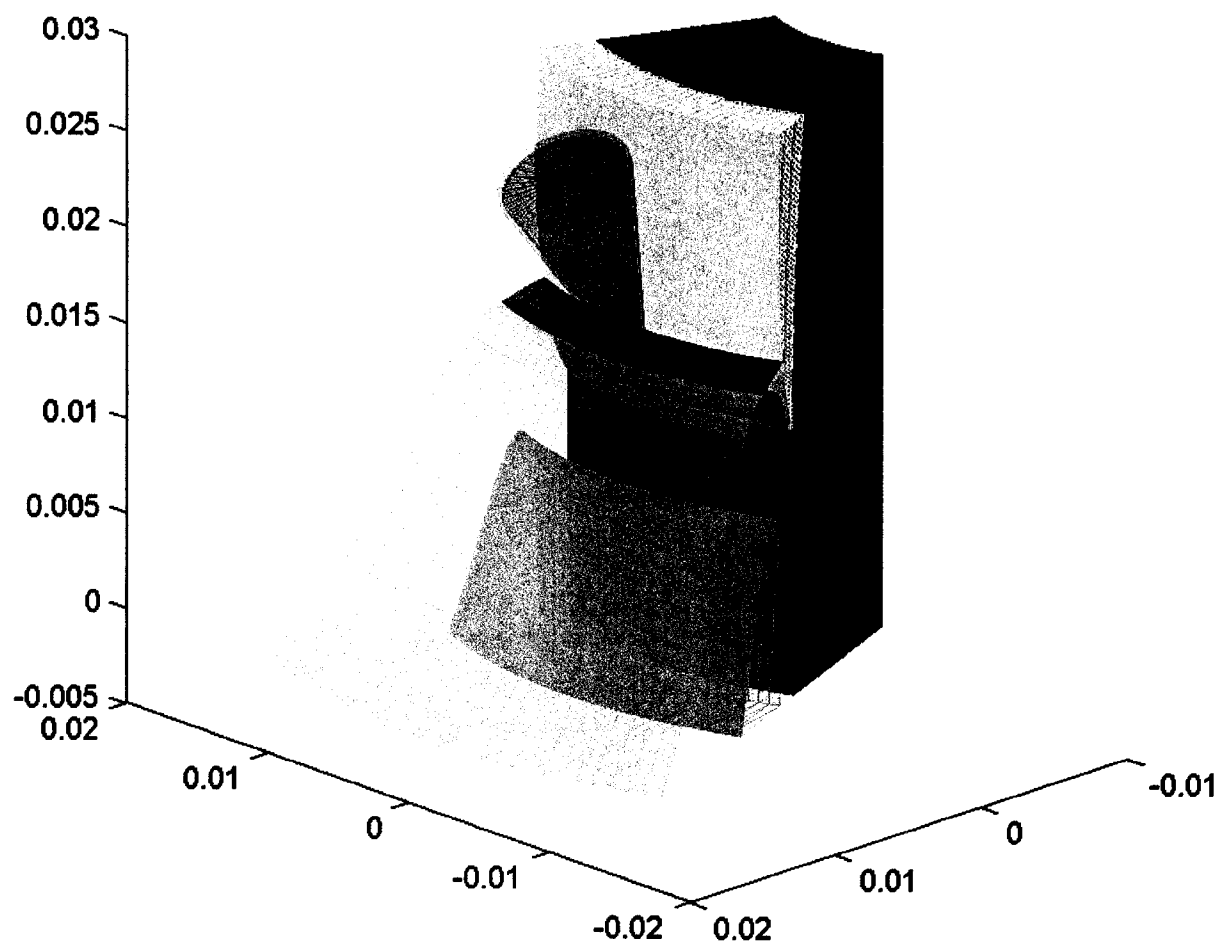
**Figure 5.10.** The 2D periodontal model with the linear tip placed at three different angles. Once the periodontal and tip geometries are defined, the placement of the tip is done automatically.

#### 5.4 Three Dimensional Periodontal Pocket and Ultrasonic Tip Geometry

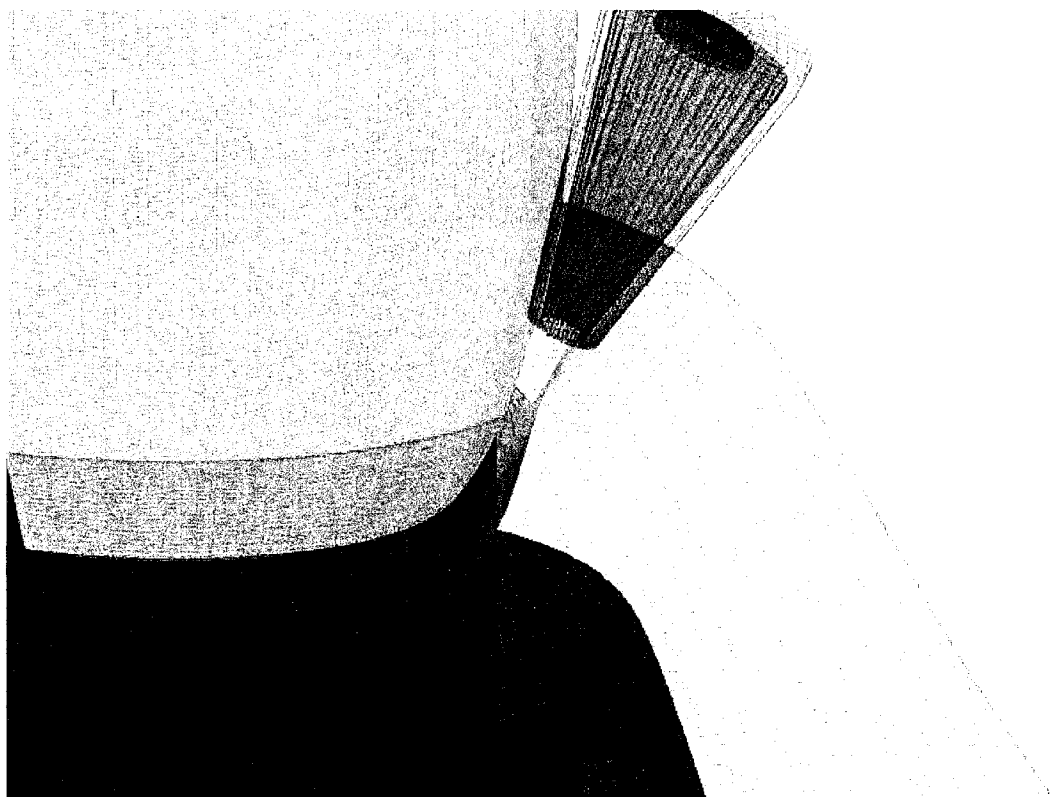
Once the 2D geometries are defined, the 3D geometries are created by rotating or sweeping the 2D models. The 3D tip is created by rotating the 2D tip geometry 360° about its central axis. Figure 5.11 shows an example 3D tip with the transducer shown in green. The 3D periodontal geometry is created by sweeping the 2D periodontal pocket model to create a curved tooth and tissue structure. An example 3D geometry with the 3D tip in place is shown in figure 5.12. A close-up of the tip on top of the periodontal pocket is shown in Figure 5.13 (the water is not shown). The radius of curvature is parameterized so that tooth structures of different curvatures can be simulated. Figure 5.14 shows three different tooth curvatures. The blue boxes in these figures are the boundaries of the simulation space.



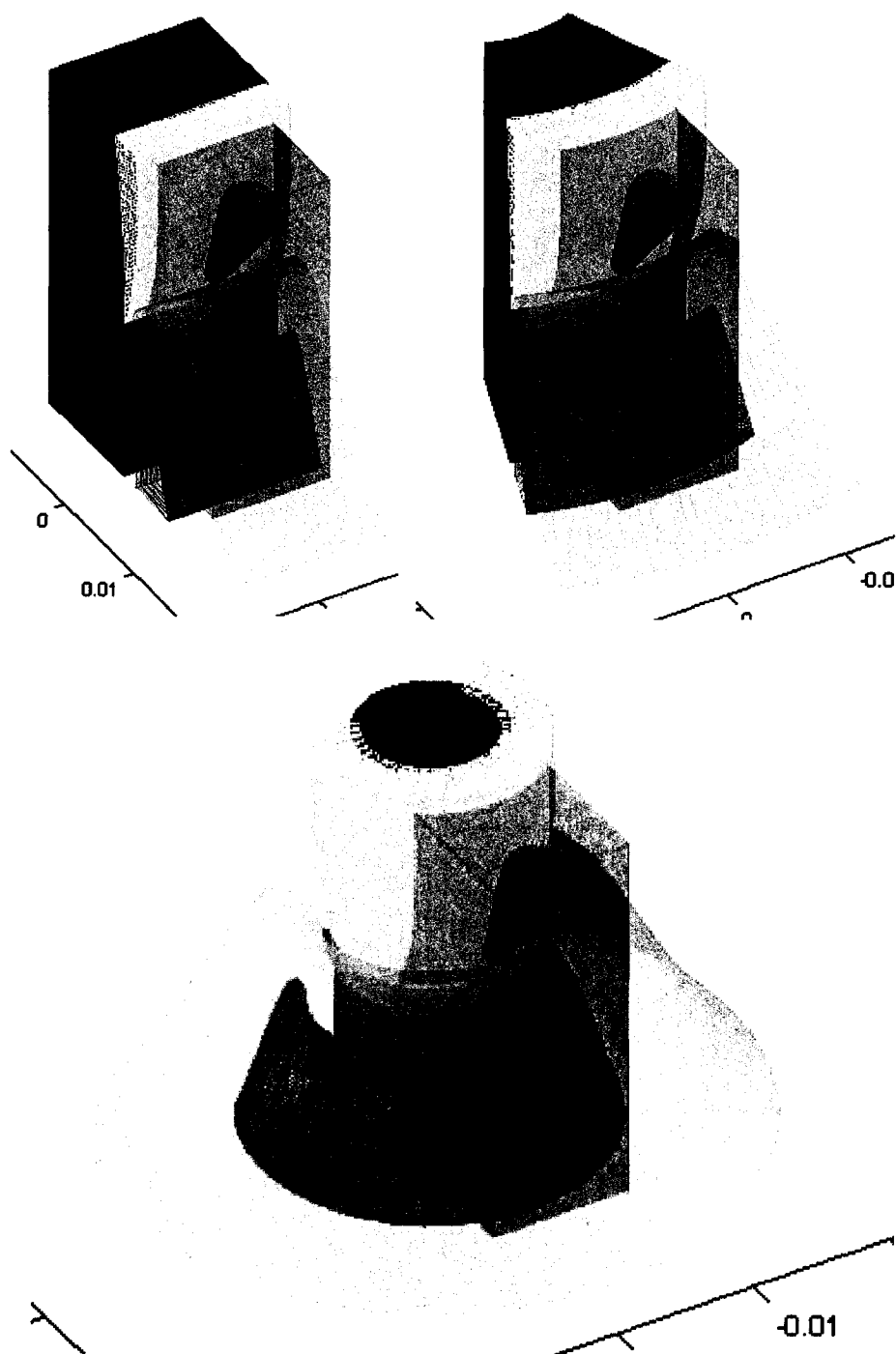
**Figure 5.11.** 3D linear tip with the transducer in the base of the tip shown in green.



**Figure 5.12.** 3D periodontal simulation geometry



**Figure 5.13.** The 3D periodontal model is shown with the 3D tip placed at the top of the pocket. The 3D model is created by sweeping the 2D model with a fixed radius of curvature.



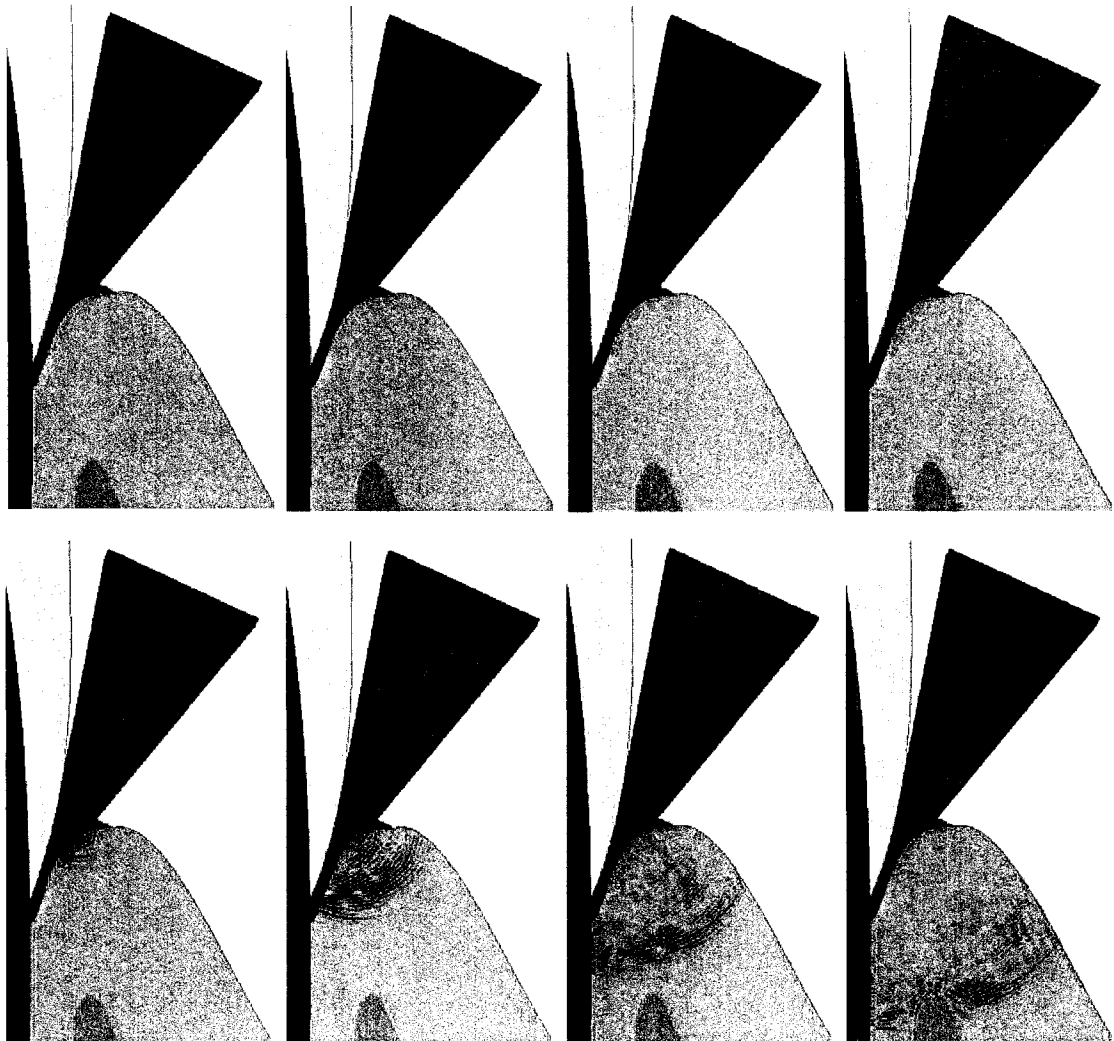
**Figure 5.14.** Three 3D models with different curvatures of the tooth anatomy ranging from completely straight (top left) to very curvy (bottom).

## **5.5 Example Simulation Output and Visualization**

Once the 3D geometries and simulation parameters are determined, a set of input files are created and passed to the 3DPAFIT code which runs on a parallel super computer (The SciClone). The details of the 3DPAFIT simulation can be found in chapter 2. Currently, the simulation software creates three different types of output or simulation results for the ultrasonic periodontal probe simulations.

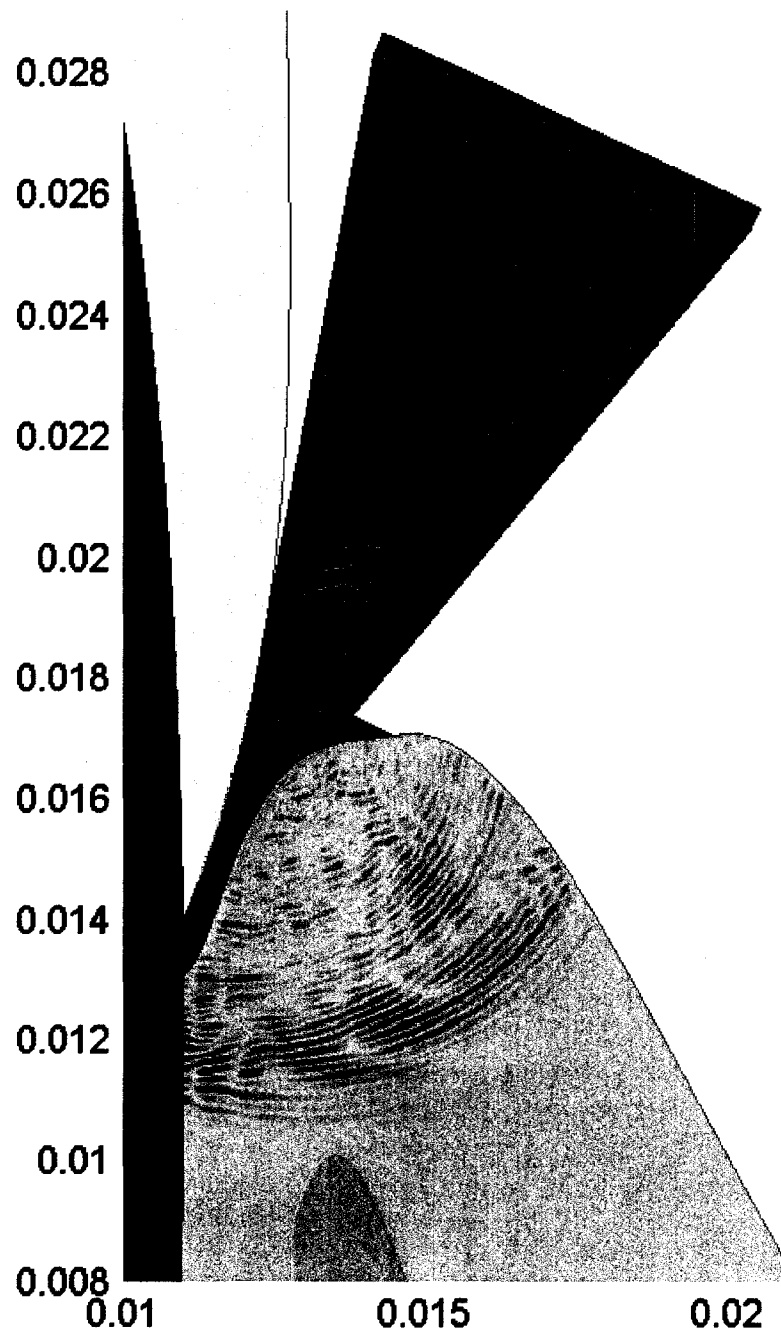
The first is a series of vertical 2D slices through the center of the simulation space. These slices show the propagation of the acoustic waves starting from the transducer and propagating into the periodontal pocket and surrounding tissues. Figure 5.15 shows a series of these snapshots for a simulation where the periodontal pocket depth is approximately 2.5mm. Figure 5.16 shows a close up of one of these snapshots.

In addition to these 2D slices, the value of the pressure waves over the entire 3D simulation space is accumulated. This 3D volume can be sliced to show the acoustic energy distribution inside the tissue. Figure 5.17 shows several horizontal slices through the volume to at three different depths below the tip. The brightness and color of these plots shows the horizontal acoustic energy distribution. Finally, the pressure across the face of the transducer is recorded to create the typical A-line. Figure 5.18 shows one of these A-lines indicating the initial burst and two reflections from the internal tip reflections.

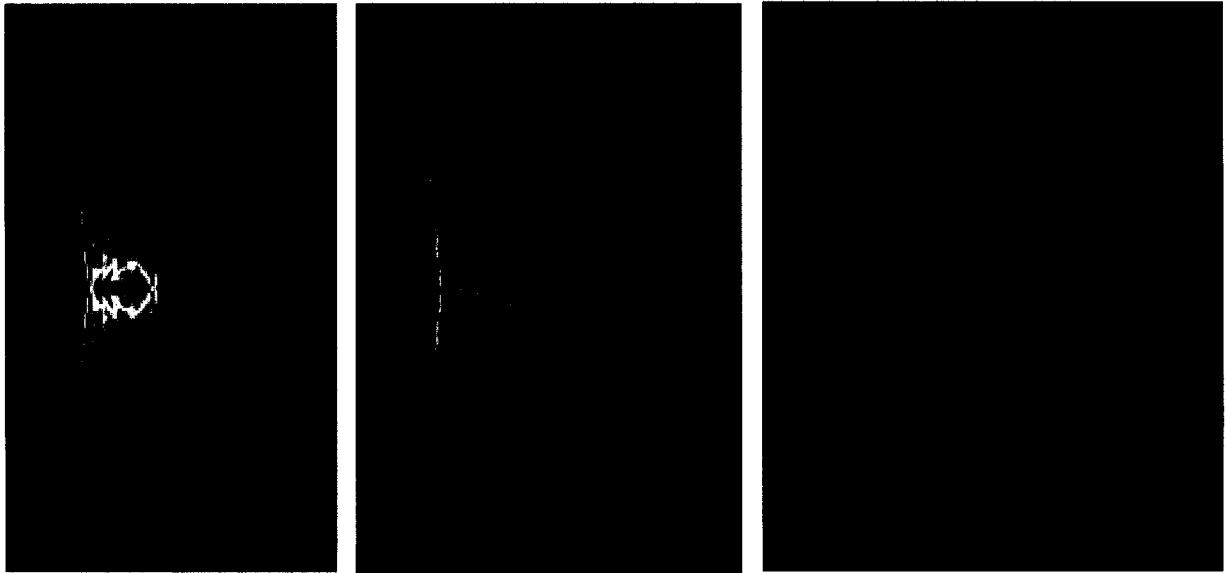


**Figure 5.15.** Several 2D snapshots from the 3D simulation showing the acoustic waves created by the transducer traveling down the tip and into the periodontal pocket and surrounding tissue. The depth of the periodontal pocket is approximately 2.5mm.

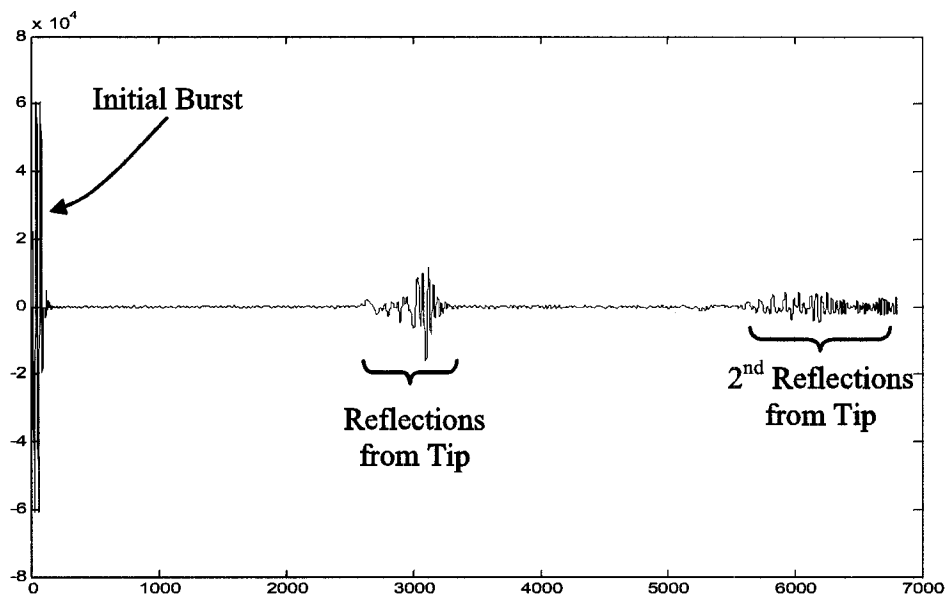




**Figure 5.16.** A close up of the pressure waves in the tip and the pocket.



**Figure 5.17.** Horizontal slices showing the acoustic energy distribution at three different depths.

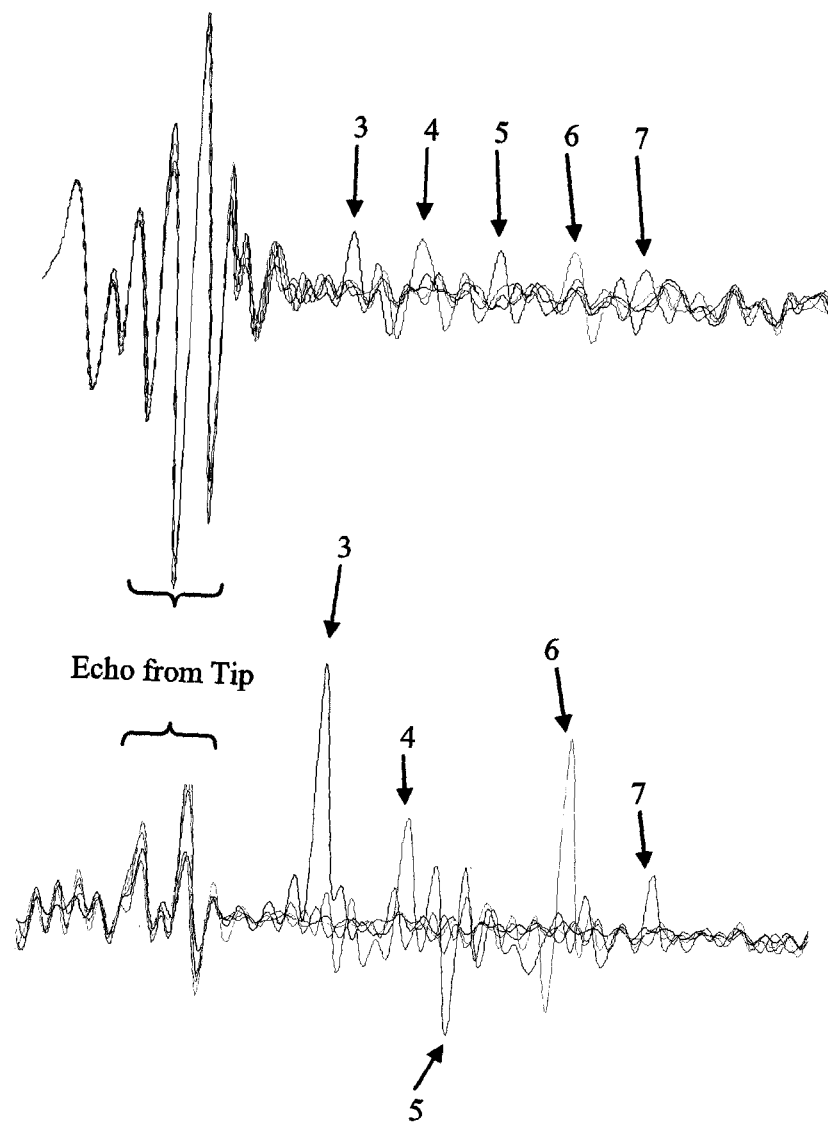


**Figure 5.18.** A typical A-line (at 5 MHz) showing the initial burst and two reflections from the tip.

## **5.6 Rigid Simulation and Experimental Results**

Systematic rigid simulations were performed to provide an indication of where the echoes from the bottom of the pocket should appear in the A-lines for the more realistic simulations. Rigid simulations are performed with the geometry of the periodontal tissues described in the previous sections except that the soft tissues of the gingival and junctional epithelium (JE) are made rigid. This guarantees that all the acoustic energy stays in the water inside the tip and the periodontal pocket and ensures that all the acoustic energy that reaches the bottom of the pocket will be reflected. A total of 40 rigid simulations were performed at 5 MHz with the pocket depth ranging from 0.5mm to 10.5mm in 0.25mm increments. Each of these simulations required about 8GB of computer memory and took approximately one day to complete when running on 16 computers (at 750Mhz each).

Experimentally, data is collected with an aluminum block with a sequence of holes drilled at different depths. The tip and the holes are filled with water just as the periodontal pocket is in the simulations. The large acoustic impedance mismatch between aluminum and the water keeps a large majority of the acoustic energy in the water. Figure 5.19 shows A-lines from simulation of the rigid periodontal region and experimental A-lines from the aluminum block at five different depths. (3mm to 7mm in 1mm increments). In both cases, the A-lines have been low-passed filtered and the amplitude of the experimental A-line at 3mm was reduced to make the plot more consistent (this echo had a very large amplitude). In both the simulated and experimental data peaks stand out and shift to the right (further in time) as the depth of the pocket/hole gets deeper.



**Figure 5.19.** A-lines are plotted for five simulations (top) and five experiments (bottom).

The differences in amplitudes can be attributed to several experimental and geometrical differences. The experimental data was collected at 10 MHz and includes hardware amplification. Also, the crevice (or slit like) geometry of the 3D periodontal pocket confines the acoustic energy to two-dimensions while the hole of the experiment confines the energy to one dimension. This would cause the echoes from the experiments to be larger than seen in the simulations.

The velocity of sound in water is approximately 1500 meters per second. In the simulated data, the average time between each peak is 1.3337 microseconds. This corresponds to a distance of 0.00200055 meters or indicating the average depth change of the pocket is 1.00027 mm. This distance is indeed the change in pocket depth in between each of the simulations. In the experimental data, the average time between the peaks is 1.5625 microseconds which corresponds to an average change in the depth of the hole of 1.171 mm. This example illustrates that the acoustic simulations of the 3D periodontal pocket can accurately create A-line data with echoes from the bottom of the periodontal pocket. The echoes accurately represent the depth of the periodontal pocket. In the more realistic simulations and experimental data, a more sophisticated signal processing technique will be needed to detect the faint echoes from the bottom of the periodontal pocket.

## 5.7 Periodontal Tissue Material Parameters

Accurate material parameters are necessary to simulate realistic acoustic propagation in the soft tissues surrounding the periodontal pocket. For the 3D acoustic simulations, the material parameters that are needed are the density  $\rho$  and the acoustic wave velocity  $c$ . Unfortunately, we have been unable to find any references that quote these material parameters directly. In previous work, 2D simulations were performed of the tip and the region surrounding the periodontal pocket [48]. In these 2D simulations, the junctional epithelium and gingiva are modeled as skin and muscle, respectively. We will take the same approach in our 3D simulations.

There are several references for the acoustic material properties of muscle and skin. From [56, 57], we find the density of skin and muscle as 1020 kg/m<sup>3</sup> and 1080 kg/m<sup>3</sup>, respectively. In addition, Culjat et. al. cite the density and acoustic wave velocity of soft tissue as 1540 m/s and 1060 kg/m<sup>3</sup> [58] and Duck cites the acoustic velocity and density of muscle as 1550m/s and 1060 kg/m<sup>3</sup> [59]. The table below indicates the material parameters used in the 3D simulations. For reference, the acoustic wave velocity and density of water is also presented [60]. We suspect that these values are close but not exact. The tissue surrounding the periodontal pocket contains many muscle-like fibers that run perpendicular to acoustic wave propagation direction. This could raise the acoustic impedance mismatch between the water and tissue. In most simulations, the acoustic impedance mismatch was increased to account for this fact.

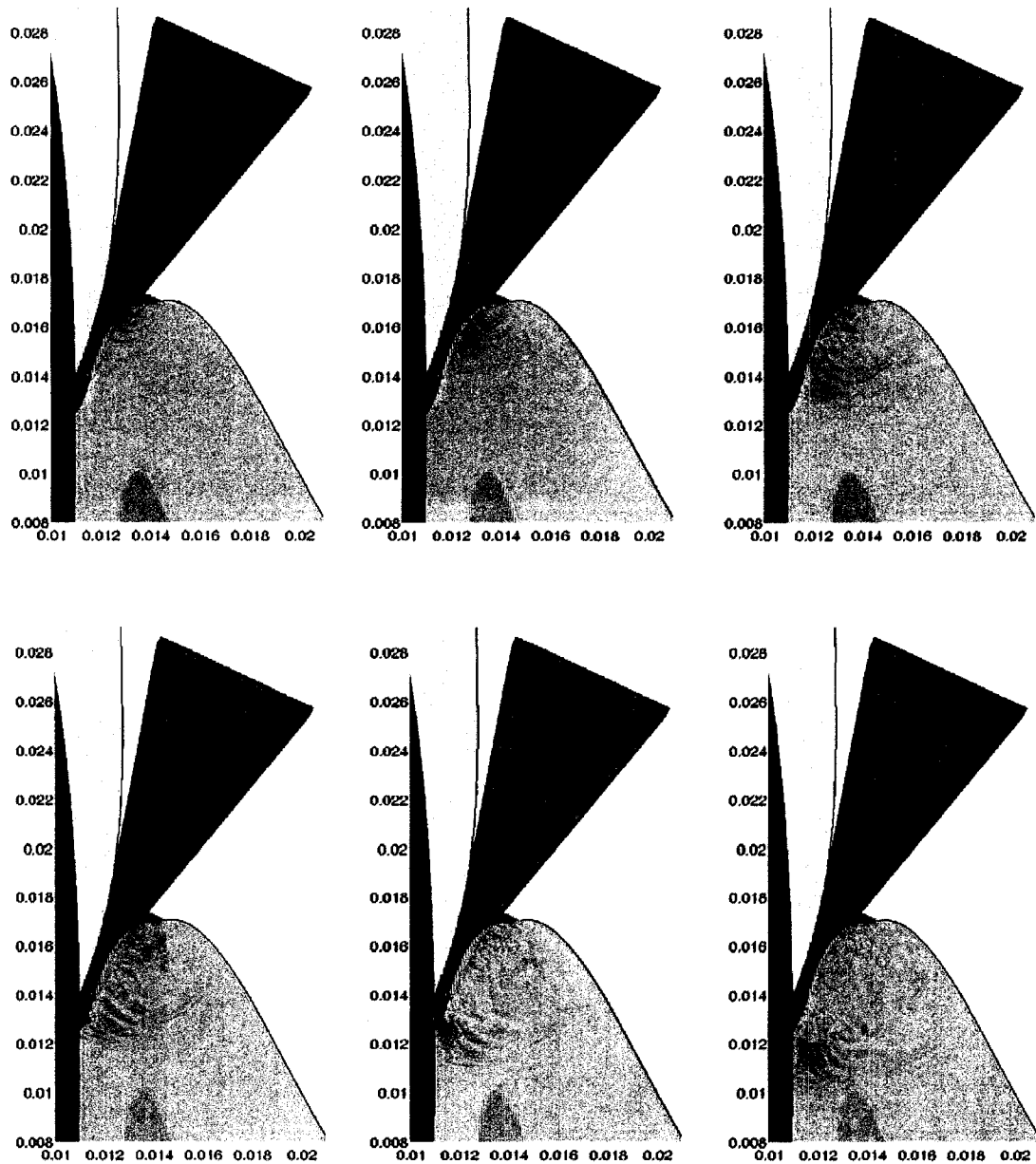
Material	Density (kg/m <sup>3</sup> )	Acoustic Wave Velocity (m/s)
Skin (JE)	1020	1540
Muscle (Gingival)	1080	1550
Water	998.2	1482.1

## 5.8 10 MHz Ultrasound Beam Study

Describing the 3D ultrasound beam inside the probe tip and the complex tissue structures requires a very sophisticated 3D model such as the one presented in this chapter. The ultrasound beam is defined by the spatial distribution and intensity of the acoustic energy inside the tissues. The ultrasound beam is very difficult to characterize because the beam interacts with several tissue layers in an inherently 3D and non-symmetric geometry. Understanding the shape of the ultrasound beam inside the tissues can assist in the interpretation of the experimental A-line measurements.

We use several visualization methods to describe the ultrasound beam inside the tissues. The first is a series of 2D vertical pressure snapshots showing the wave propagation through the probe tip and into the periodontal tissues. Figure 5.20 shows several snapshots from a 10 MHz simulation. These snapshots show that a large portion of the acoustic wave energy is channeled down the water filling the pocket. After the interaction with the bottom of the pocket, the waves scatter off the dentin below the crest and the beam begins to slowly diverge away from the tooth.

Figure 5.21 shows a vertical pressure accumulated snapshot where the dark blue color indicates the spatial distribution of the acoustic energy. This plot also shows a large amount of the acoustic energy is confined to the pocket and scatters off the dentin below the crest. This degree of the scattering off the dentin depends on the angle of the probe tip and the geometry of the tooth structure.



**Figure 5.20.** A series of vertical pressure snapshots showing the acoustic wave progression. In the top center snapshot, the 10 MHz acoustic waves are mostly confined to the pocket or near the interface between the water in the pocket and the gingival tissue.

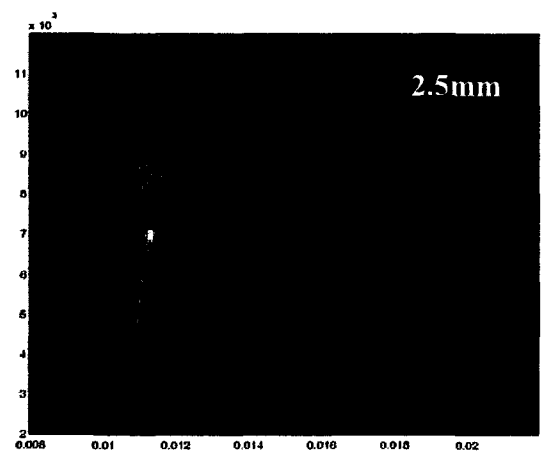
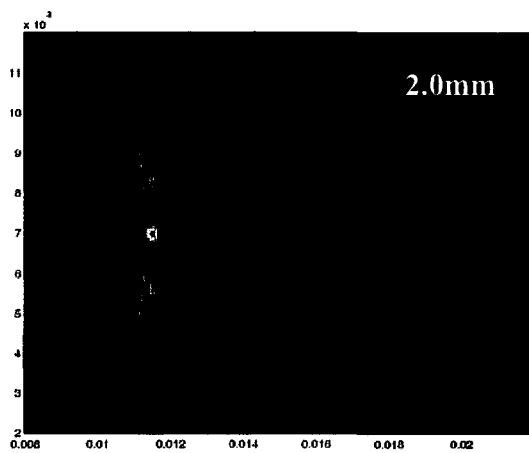
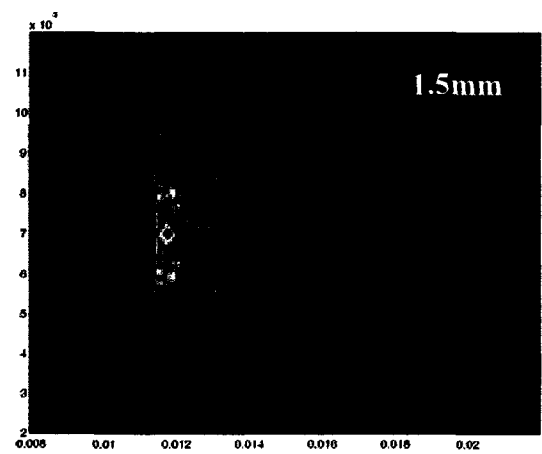
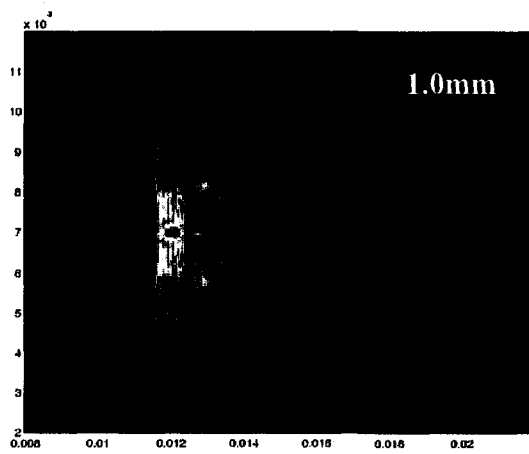
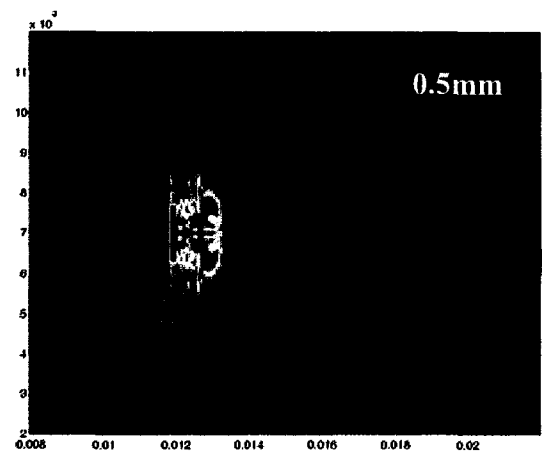
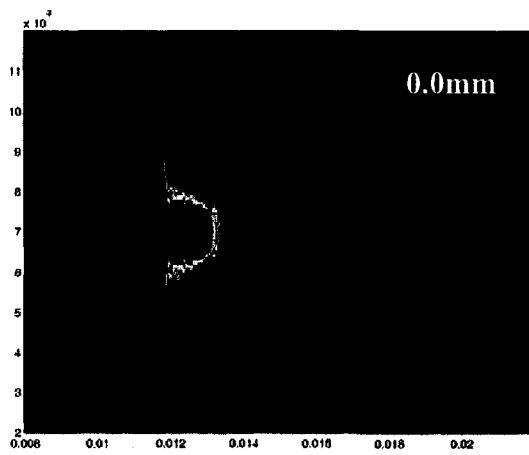


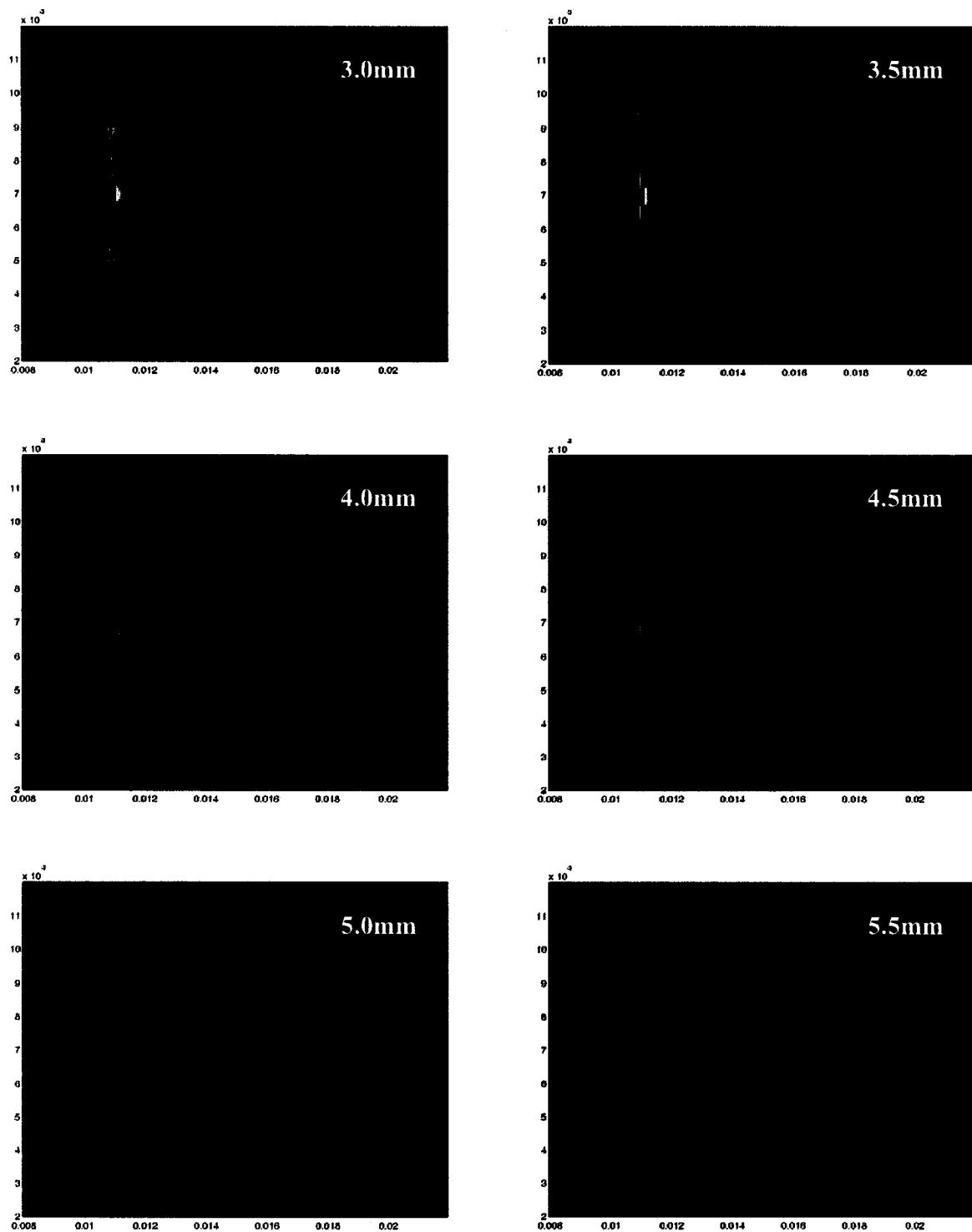


**Figure 5.21.** Vertical beam profile from a 10 Mhz simulation showing the spatial distribution of the acoustic energy. The darkness of the blue is proportional to the acoustic energy intensity.

In addition to vertical profile of the ultrasound beam, we also examine horizontal profile slices showing the lateral spread of the beam as it propagates down through the tissue. Figure 5.22 (spanning two pages) contains 12 horizontal beam profile slices located in 0.5 mm increments below the bottom edge of the tip. The colors of the images are proportional to the spatial energy intensity with red being more intense than blue. The right side of these images would correspond to the facial side of the tissue geometry.

In this simulation, the depth of the pocket is 3mm. In between 0.5mm and 2.5mm, you can make out the boundary between the water in the pocket and the soft gingival tissue. As the energy propagates through the tissue, several side lobes form inside the gingival tissue and inside the pocket. The side lobes inside the gingival tissue quickly lose their strength as compared to the main beam inside the pocket. Two side lobes form on either side of the main beam inside the pocket. It is not suspected that any significant returns would come from these side lobes since these lobes diverge away from the main lobe as the waves propagate. When the acoustic waves interact with the bottom of the pocket (3mm), the beam is very narrow and close to the tooth surface. From approximately 4 – 6mm, the beam reflects off the dentin and begins to slowly spread out but is still relatively narrow.





**Figure 5.22.** A series of horizontal beam profiles showing the lateral resolution of the ultrasound beam at 10 MHz.

## 5.9 Conclusions

In this chapter, we have shown that the three-dimensional parallel acoustic finite integration technique can accurately simulate acoustic wave propagation in the complex geometry of the periodontal region. A sophisticated set of software tools were created to automatically create the 3D geometry of the periodontal region and run systematic simulations. Several techniques are used to visualize the ultrasound waves inside the tip and in the periodontal pocket and surrounding tissues.

Rigid simulations were performed to show that the simulation software can produce realistic data with echoes corresponding to the depth of the periodontal pocket. A set of 10 MHz simulations were completed to describe the ultrasound beam inside the tissue. In addition many systematic simulations were performed to create a large data set to assist in the development of the signal processing algorithms to automatically detect the pocket depth.

## 5.10 References

1. Walter J. Loesche, a.N.S.G., *Periodontal Disease as a Specific, albeit Chronic, Infection: Diagnosis and Treatment*. Clin Microbiol Rev, 2001. 14(4): p. 727–752.
2. Socransky S S, H.A.D., Smith C, Dibart S., *Relation of counts of microbial species to clinical status at the sampled site*. Journal of Clinical Periodontol, 1991. 17: p. 788–792.
3. C.A. Evans, D.V.L., W.R. Maas, et al, *Oral Health in America: A Report to the Surgeon General*. 2000, National Institute of Dental and Craniofacial Research: Bethesda.
4. Nevins, E.P.B.a.M., *Diagnosing periodontal diseases*. J. Am. Dental Assoc., 1990. 121: p. 460–464.
5. G.C. Armitage, e.a., *Diagnosis of Periodontal Diseases, Academy Report Position Paper*. J. Periodontol, 2003. 74: p. 1237–1247.

6. Albandar J M, B.J.A., Kingman A., *Destructive periodontal disease in adults 30 years of age and older in the United States, 1988–1994*. J Periodontol, 1999. **70**: p. 13-29.
7. Oliver R C, B.L.J., Loe H., *Periodontal diseases in the United States population*. J Periodontol., 1998. **69**: p. 269-278.
8. N., P.P., *Periodontal diseases: epidemiology*. Ann Periodontol, 1996. **1**: p. 1–36.
9. Fowler, E.B., *Periodontal disease and its association with systemic disease*. Military Medicine, 2001. **166**: p. 85-89.
10. Paquette, D.W., *The periodontal infection-systemic disease link: a review of the truth or myth*. J. Int. Acad. of Periodontology, 2002. **4**: p. 101-109.
11. Slots, J., *Update on general health risk of periodontal disease*. International Dental Journal, 2003. **53**: p. 200-207.
12. Beck J D, P.J., Tyroler H A, Offenbacher S., *Dental infections and atherosclerosis*. Am Heart J., 1999. **138**: p. S528–S533.
13. Loesche W J, S.A., Terpenning M S, Chen Y M, Dominguez B L, Grossman N., *Assessing the relationship between dental disease and coronary heart disease in elderly U.S. veterans*. J Am Dent Assoc, 1998. **129**: p. 301-311.
14. Mattila K J, V.V.V., Nieminen M S, Asikainen S, *Role of infection as a risk factor for atherosclerosis, myocardial infarction, and stroke*. Clin Infect Dis, 1998. **26**: p. 719–734.
15. Trovato, J.P., *role of the general dentist in periodontal care*. General Dentistry, 2003. **March-April**: p. 176-181.
16. A.D. Haffajee, e.a., *Clinical parameters as predictors of destructive periodontal disease activity*. J. Clinical Periodontology, 1982. **10**: p. 257.
17. Slots, T.E.R.a.J., *Comparison of two pressure-sensitive periodontal probes and a manual probe in shallow and deep pockets*. Int. J. of Periodontics & Restorative Dentistry, 1993. **13**: p. 521-529.
18. Corbet, N.P.L.a.E.F., *Diagnostic procedures in daily practice*. Int. Dental J., 1995. **45**: p. 5-15.
19. Lamster, G.G.a.I., *Understanding diagnostic testing for periodontal diseases*. J. Periodontology, 1995. **66**: p. 659-666.
20. Listgarten, M.A., *Periodontal probing: What does it mean?* J. Clin Periodontology., 1980. **7**: p. 165.
21. Hunter, F., *Periodontal probes and probing*. Int. Dental J., 1994. **44**: p. 557-583.
22. L. Mayfield, G.B.a.R.A., *Periodontal probe precision using 4 different periodontal probes*. J. Clin. Periodontology, 1996. **23**: p. 76-82.
23. L. Tupta-Veselicky, e.a., *A clinical study of an electronic constant force periodontal probe*. J. Periodontology, 1994. **65**: p. 616-622.
24. Cattabriga, M., *Future diagnostic possibilities in periodontology*. Int. Dental J., 1993. **43**: p. 109-115.
25. M.C.K. Yang, e.a., *Reproducibility of an electronic probe in relative attachment level measurements*. J. Clin. Periodontology, 1992. **19**: p. 306-311.
26. N. Ahmed, T.L.P.W.a.R.F.W., *An investigation of the validity of attachment level measurements with an automated periodontal probe*. J. Clin. Periodontology, 1996. **23**: p. 452-455.

27. A. Agüero, e.a., *Histological location of a standardized periodontal probe in man*. J. Periodontology, 1995. **66**: p. 184-190.
28. J.G. Keagle, e.a., *Effect of gingival wall on resistance to probing forces*. J. Clin. Periodontology, 1995. **22**: p. 953-957.
29. Kydd WL, D.C., Wheeler JB 3rd., *The thickness measurement of masticatory mucosa in vivo*. Int Dent J., 1971. **21**(4): p. 430-441.
30. H., S., *Ultra-sonic diagnosis of marginal periodontal diseases*. 1971, Int Dent J. **21**(4): p. 442-455.
31. Muraoka Y, S.T., Kinoshita S., *Examination of periodontal tissue with an ultrasonic apparatus. Measurements of the thickness of gingiva, alveolar mucosa and alveolar bone [Article in Japanese]*. Nippon Shishubyo Gakkai Kaishi, 1982. **24**(4): p. 601-606.
32. Sawada K, F.T., Sunada I., *Ultrasonography of the periodontal tissue [Article in Japanese]*. Nippon Shishubyo Gakkai Kaishi, 1984. **26**(1): p. 88-93.
33. Lost C, I.K., Nussle W., *Periodontal ultrasonic diagnosis: experiments on thin bony platelets and on a simulated periodontal ligament space*. J Periodontal Res, 1988. **41**(9): p. 347-351.
34. Palou ME, M.M., Rossmann JA., *The use of ultrasound for the determination of periodontal bone morphology*. J Periodontol, 1987. **58**(4): p. 262-265.
35. Walmsley AD, L.W.a.L.P., *Ultrasound in dentistry. Part 2 - periodontology and endodontics*. J. Dent, 1991. **19**: p. 11-17.
36. K., O., *Application of ultrasonography to periodontal diagnosis. [Article in Japanese]*. Nippon Shishubyo Gakkai Kaishi, 1989. **1**: p. 235-40.
37. T. Eger, H.-P.M.a.A.H., *Ultrasonic determination of gingival thickness*. Journal of Clinical Periodontology, 1996. **23**(9): p. 839.
38. Tsiolis FI, N.I., Griffiths GS, *Periodontal ultrasonography*. J. Clin. Periodontology, 2003. **30**: p. 849-854.
39. K, L.D.R.H., *Ultrasonic periodontal diagnostic instrumentation system with clinical results*. Measurement, 1998. **23**(3): p. 125-129.
40. Demyun, S.M.H., Keith M., *Ultrasonic method and apparatus for measuring the periodontal pocket*, Periosonics, Inc: United States.
41. Mark Hinders, J.C., *ULTRASONIC PERIODONTAL PROBE*. 25th Review of Progress in Quantitative Nondestructive Evaluation, 1998.
42. Mark Hinders, A.G., John Companion, *Ultrasonic periodontal probe*. The Journal of the Acoustical Society of America, 1998. **104**(3): p. 1844.
43. *Goodbye Gingivitis*. Virginia Business, 1997: p. 9.
44. Companion, J.A., *Differential measurement periodontal structures mapping system*, The United States of America as represented by the Administrator NASA: United States.
45. Farr, C., *Ultrasonic Probing: The Wave of the Future in Dentistry*. Dentistry Today, 2000.
46. Mark K. Hinders, J.E.L.a.G.B.M., *CLINICAL TESTS OF AN ULTRASONIC PERIODONTAL PROBE*. 28th Review of Progress in Quantitative Nondestructive Evaluation, 2001: p. 1880-1890.
47. M. Hinders, J.L.a.G.M. *Disease Diagnosis using an UltraSonographic Probe*. in *Advancements in Ultrasonics Symposium*. 2001.

48. Lynch, T., *Ultrasonographic Measurement of Periodontal Attachment Levels*, in *Department of Applied Science*. 2001, William and Mary.
49. JE Lynch, M.H., *Ultrasonic device for measuring periodontal attachment levels*. *Review of Scientific Instruments*, 2002. **73**(7): p. 2686-2693.
50. Hou, J., *Ultrasonic Signal Detection and Characterization Using Dynamic Wavelet Fingerprints*, in *Department of Applied Science*. 2004, William and Mary.
51. Mark K. Hinders, J.H., *ULTRASONIC PERIODONTAL PROBING BASED ON THE DYNAMIC WAVELET FINGERPRINT*. 31st Review of Progress in Quantitative Nondestructive Evaluation. See: AIP Conference, 2004. **760**(2): p. 1549-1556.
52. Hou, J.R., ST; Hinders, MK, *Ultrasonic Periodontal Probing Based on the Dynamic Wavelet Fingerprint*. *Eurasip Journal on Applied Signal Processing*, 2005. **7**: p. 1137-1146.
53. Lynch, J.H., MK; McCombs, *Clinical comparison of an ultrasonographic periodontal probe to manual and controlled-force probing*. *Measurement*, 2006. **39**(5): p. 429-439.
54. Hinders, G.B.M.a.M., *The Potential of the Ultrasonic Probe*. *Dimensions of Dental Hygiene*, 2006. **4**(4): p. 16-18.
55. M. Hinders, J.H., *Dynamic Wavelet Fingerprint Identification of Ultrasound Signals*. *Materials Evaluation*, 2002. **60**(9): p. 1089-1093.
56. EL Madsen, J.S., JA Zagzebski, *Ultrasonic shear wave properties of soft tissues and tissuelike materials*. *Journal of Acoustical Society of America*. **74**: p. 1346-55.
57. SA Goss, R.J., F Dunn, *Comprehensive compilation of empirical ultrasonic properties of mammalian tissues*. *Journal of Acoustical Society of America*, 1978. **64**: p. 423-56.
58. Culjat MO, S.R., Brown ER, Neurgaonkar RR, Yoon DC and White SN, *Ultrasound crack detection in a simulated tooth*. *Dentomaxillofacial Radiology*, 2005. **34**: p. 80-85.
59. Duck, F.A., *Physical properties of tissue*. 1990, London: Academic Press.
60. L. Kinsler, A.F., A. Coopens, and J. Sanders, *Fundamentals of Acoustics*. 4th ed. 2000: John Wiley & Sons, Inc. 162-63.



## Chapter VI

# 3D Parallel Cylindrical Elastic Finite Integration Technique (3DPCEFIT)

### 6.1 Introduction

We present here a simulation method based on the elastodynamic finite integration technique (EFIT) that can model guided elastic wave propagation in pipe-like structures including 3D pipe bends. Several simulation techniques exist for modeling elastic waves in pipe-like structures. Gsell *et al.* developed a finite-difference technique for modeling elastic waves in straight pipe-like structures based on the displacement-equations of motion [1]. One advantage of this technique is its ability to model elastic wave interaction with subtle flaws due to its fine grid spacing. Leutenegger *et al.* showed how this method could be used to assist in locating defects in cylindrical structures [2]. Hayashi *et al.* developed a semi-analytical finite-element (SAFE) technique for modeling elastic wave propagation in pipe-like structures including pipe bends [3, 4]. Their technique uses a relatively large spatial discretization which leads to fast computational times and allows for long pipe sections with multiple bends to be modeled. Unfortunately, this technique as described can not be used to model guided wave interactions with subtle flaws. The finite integration technique is a powerful, accurate, and stable time-domain method for numerically solving partial differential equations. It

has been used to model 2D, axially-symmetric (2.5D) and full 3D elastic waves in the Cartesian and cylindrical coordinate systems[5-7].

We present a finite integration method for modeling elastic waves in pipe-like structures and pipe bends. We then show that this method compares well to experimental results. We also show how the fine spatial discretization allows guided elastic wave scattering from subtle flaws to be modeled. This simulation method can be used to design complicated hardware devices such as phased array transducer belts to focus the elastic wave energy on straight pipe sections, as well as beyond pipe bends, and to generate systematic data to test signal processing algorithms.

## 6.2 3D Cylindrical Elastic Finite Integration Technique (3DCEFIT)

We first describe how the finite integration technique is used to simulate 3D elastic waves in complex pipe-like structures. We present the equations necessary to simulate elastic waves in a pipe-bend and show how they can easily be adapted to model straight pipe sections.

## 6.3 Finite Integration Procedure

We begin with the nine equations for elastic wave propagation in solids using the cylindrical coordinate system [8].

$$\rho \dot{v}_r = \frac{\partial T_{rr}}{\partial r} + \frac{1}{r} \frac{\partial T_{r\phi}}{\partial \phi} + \frac{\partial T_{rz}}{\partial z} + \frac{T_{rr} - T_{\phi\phi}}{r} + f_r \quad (6.1)$$

$$\rho \dot{v}_z = \frac{\partial T_{rz}}{\partial r} + \frac{1}{r} \frac{\partial T_{\varphi z}}{\partial \varphi} + \frac{\partial T_{zz}}{\partial z} + \frac{1}{r} T_{rz} + f_z \quad (6.2)$$

$$\rho \dot{v}_\phi = \frac{\partial T_{r\phi}}{\partial r} + \frac{1}{r} \frac{\partial T_{\phi\phi}}{\partial \varphi} + \frac{\partial T_{\phi z}}{\partial z} + \frac{2}{r} T_{r\phi} + f_\phi \quad (6.3)$$

$$\dot{T}_{rr} = (\lambda + 2\mu) \frac{\partial v_r}{\partial r} + \lambda \left( \frac{v_r}{r} + \frac{1}{r} \frac{\partial v_\phi}{\partial \varphi} + \frac{\partial v_z}{\partial z} \right) \quad (6.4)$$

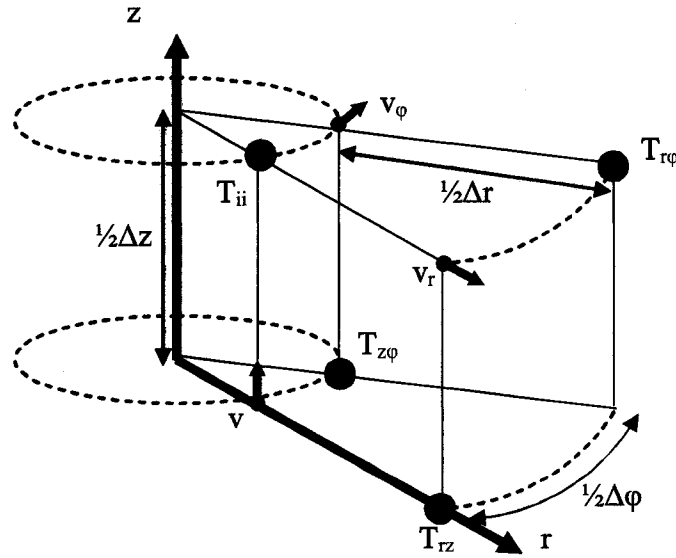
$$\dot{T}_{\phi\phi} = (\lambda + 2\mu) \left( \frac{v_r}{r} + \frac{1}{r} \frac{\partial v_\phi}{\partial \varphi} \right) + \lambda \left( \frac{\partial v_r}{\partial r} + \frac{\partial v_z}{\partial z} \right) \quad (6.5)$$

$$\dot{T}_{zz} = (\lambda + 2\mu) \frac{\partial v_z}{\partial z} + \lambda \left( \frac{v_r}{r} + \frac{1}{r} \frac{\partial v_\phi}{\partial \varphi} + \frac{\partial v_r}{\partial r} \right) \quad (6.6)$$

$$\dot{T}_{r\phi} = \mu \left( \frac{1}{r} \frac{\partial v_r}{\partial \varphi} + \frac{\partial v_\phi}{\partial r} - \frac{v_\phi}{r} \right) \quad (6.7)$$

$$\dot{T}_{rz} = \mu \left( \frac{\partial v_z}{\partial r} + \frac{\partial v_r}{\partial z} \right) \quad (6.8)$$

$$\dot{T}_{z\phi} = \mu \left( \frac{\partial v_\phi}{\partial z} + \frac{1}{r} \frac{\partial v_z}{\partial \varphi} \right) \quad (6.9)$$



**Figure 6.1.** Staggered distribution of the nine unknown variables on a single 3D cylindrical computational cell.

The variables  $v$  and  $T$  are the components of the velocity vector and stress tensor, respectively. The material parameters are density  $\rho$  and the Lamé constants  $\lambda$  and  $\mu$ . The velocity source function components are represented by  $f_r$ ,  $f_z$ , and  $f_\phi$ . For the finite integration method, the 9 unknown variables are placed on a staggered grid. A single control volume with the variable distribution for the cylindrical coordinate system is shown in figure 6.1. Many millions of these grid cells are stacked together to create a computational space that accurately represents a pipe-like structure. Now we will describe the derivation of the simulation equations.

### ***Simulation Equation for $v_r$***

First we integrate both sides of equation 6.1 over a cylindrical control volume centered on the radial velocity component  $v_r$ .

$$\iiint \rho \dot{v}_r r \, dr dz d\varphi = \iiint \left( \frac{\partial T_{rr}}{\partial r} + \frac{1}{r} \frac{\partial T_{r\varphi}}{\partial \varphi} + \frac{\partial T_{rz}}{\partial z} + \frac{T_{rr} - T_{\varphi\varphi}}{r} + f_r \right) r \, dr dz d\varphi \quad (6.10)$$

Now we invoke the divergence theorem which replaces part of the volume integrals on the right hand side with a surface integral.

$$\iiint \rho \dot{v}_r r \, dr dz d\varphi = \oint_{\partial V} (r T_{rr} dz d\varphi + T_{r\varphi} dr dz + r T_{rz} dr d\varphi) + \iiint \left( \frac{T_{rr} - T_{\varphi\varphi}}{r} + f_r \right) r \, dr dz d\varphi \quad (6.11)$$

Next we approximate the integrals by multiplying the integrand by the volume or surface of each integral. Unlike the cartesian method (EFIT), the inner and outer surfaces of the control volume have different surface areas. These surfaces areas are defined by  $r \Delta z \Delta \varphi$ , where the distance  $r$  is different for the inner and outer surface of the control volume. Thus we denote  $r^{(i)}$  and  $r^{(o)}$  as the radial distance to the inner and outer surfaces, respectively. We also introduce superscripts  $+$ ,  $-$ ,  $t$ , and  $b$  to indicate the direction of the variables relative to the unknown variable  $v_r$ . The superscripts  $+$  and  $-$  represent the direction of the variables in the positive and negative  $\varphi$  direction and superscripts  $t$  and  $b$  represent the direction of the variables in the positive and negative  $z$  direction.

$$\begin{aligned} \rho \dot{v}_r r \Delta r \Delta z \Delta \varphi &= (r^{(o)} T_{rr}^{(o)} - r^{(i)} T_{rr}^{(i)}) \Delta z \Delta \varphi + (T_{r\varphi}^{(+)} - T_{r\varphi}^{(-)}) \Delta r \Delta z + (T_{rz}^{(i)} - T_{rz}^{(b)}) r \Delta r \Delta \varphi + \\ &\left( \frac{T_{rr}^{(o)} + T_{rr}^{(i)} - T_{\varphi\varphi}^{(o)} - T_{\varphi\varphi}^{(i)}}{2r} + f_r \right) r \Delta r \Delta z \Delta \varphi \end{aligned} \quad (6.12)$$

Next, we divide both sides by  $r \Delta r \Delta z \Delta \varphi$ .

$$\begin{aligned} \rho \dot{v}_r &= \frac{(r^{(o)} T_{rr}^{(o)} - r^{(i)} T_{rr}^{(i)})}{r \Delta r} + \frac{(T_{r\varphi}^{(+)} - T_{r\varphi}^{(-)})}{r \Delta \varphi} + \frac{(T_{rz}^{(i)} - T_{rz}^{(b)})}{\Delta z} \\ &+ \frac{T_{rr}^{(o)} + T_{rr}^{(i)} - T_{\varphi\varphi}^{(o)} - T_{\varphi\varphi}^{(i)}}{2r} + f_r \end{aligned} \quad (6.13)$$

We have now spatially discretized our equation. Later we will use a central time difference to replace the time derivative on the left hand side to reveal our final equation.

We follow these same four steps to find the other 8 equations.

#### ***Simulation Equation for $v_z$ (Equation 6.2)***

$$\begin{aligned} 1) \quad \iiint \rho \dot{v}_z r \, dr \, dz \, d\varphi &= \iiint \left( \frac{\partial T_{rz}}{\partial r} + \frac{1}{r} \frac{\partial T_{\varphi z}}{\partial \varphi} + \frac{\partial T_{zz}}{\partial z} + \frac{1}{r} T_{rz} + f_z \right) r \, dr \, dz \, d\varphi \\ 2) &= \oint_{\partial V} (r T_{rz} \, dz \, d\varphi + T_{\varphi z} \, dr \, dz + r T_{zz} \, dr \, d\varphi) + \iiint \left( \frac{1}{r} T_{rz} + f_z \right) r \, dr \, dz \, d\varphi \\ 3) \quad \rho \dot{v}_z r \Delta r \Delta z \Delta \varphi &= (r^{(o)} T_{rz}^{(o)} - r^{(i)} T_{rz}^{(i)}) \Delta z \Delta \varphi + (T_{\varphi z}^{(+)} - T_{\varphi z}^{(-)}) \Delta r \Delta z \\ &+ (T_{zz}^{(i)} - T_{zz}^{(b)}) r \Delta r \Delta \varphi + \left( \frac{T_{rz}^{(o)} + T_{rz}^{(i)}}{2r} + f_z \right) r \Delta r \Delta z \Delta \varphi \\ 4) \quad \rho \dot{v}_z &= \frac{(r^{(o)} T_{rz}^{(o)} - r^{(i)} T_{rz}^{(i)})}{r \Delta r} + \frac{(T_{\varphi z}^{(+)} - T_{\varphi z}^{(-)})}{r \Delta \varphi} + \frac{(T_{zz}^{(i)} - T_{zz}^{(b)})}{\Delta z} \\ &+ \frac{(T_{rz}^{(o)} + T_{rz}^{(i)})}{2r} + f_z \end{aligned}$$

**Simulation Equation for  $v_\phi$  (Equation 6.3)**

$$\begin{aligned}
 1) \quad \iiint \rho \dot{v}_\phi r \, dr dz d\phi &= \iiint \left( \frac{\partial T_{r\phi}}{\partial r} + \frac{1}{r} \frac{\partial T_{\phi\phi}}{\partial \phi} + \frac{\partial T_{\phi z}}{\partial z} + \frac{2}{r} T_{r\phi} + f_\phi \right) r \, dr dz d\phi \\
 2) \quad &= \oint_{\partial V} (r T_{r\phi} dz d\phi + T_{\phi\phi} dr dz + T_{z\phi} r dr d\phi) \\
 &\quad + \iiint \left( \frac{2}{r} T_{r\phi} + f_\phi \right) r \, dr dz d\phi \\
 3) \quad \rho \dot{v}_\phi r \Delta r \Delta z \Delta \phi &= (r^{(o)} T_{r\phi}^{(o)} - r^{(i)} T_{r\phi}^{(i)}) \Delta z \Delta \phi + (T_{\phi\phi}^{(+)} - T_{\phi\phi}^{(-)}) \Delta r \Delta z \\
 &\quad + (T_{z\phi}^{(t)} - T_{z\phi}^{(b)}) r \Delta r \Delta \phi + \left( \frac{2}{r} T_{r\phi}^{(c)} + f_\phi \right) r \Delta r \Delta z \Delta \phi \\
 4) \quad \rho \dot{v}_\phi &= \frac{(r^{(o)} T_{r\phi}^{(o)} - r^{(i)} T_{r\phi}^{(i)})}{r \Delta r} + \frac{(T_{\phi\phi}^{(+)} - T_{\phi\phi}^{(-)})}{r \Delta \phi} + \frac{(T_{z\phi}^{(t)} - T_{z\phi}^{(b)})}{\Delta z} \\
 &\quad + \frac{(T_{r\phi}^{(o)} + T_{r\phi}^{(i)})}{r} + f_\phi
 \end{aligned}$$

**Simulation Equation for  $T_{rr}$  (Equation 6.4)**

$$\begin{aligned}
 1) \quad \iiint \dot{T}_{rr} r \, dr dz d\phi &= \iiint \left( (\lambda + 2\mu) \frac{\partial v_r}{\partial r} + \lambda \left( \frac{v_r}{r} + \frac{1}{r} \frac{\partial v_\phi}{\partial \phi} + \frac{\partial v_z}{\partial z} \right) \right) r \, dr dz d\phi \\
 2) \quad &= \oint_{\partial V} r \left( (\lambda + 2\mu) v_r dz d\phi + \frac{\lambda}{r} v_\phi dr dz + \lambda v_z dr d\phi \right) \\
 &\quad + \iiint \left( \lambda \frac{v_r}{r} \right) r \, dr dz d\phi \\
 3) \quad \dot{T}_{rr} r \Delta r \Delta z \Delta \phi &= (\lambda + 2\mu) (r^{(o)} v_r^{(o)} - r^{(i)} v_r^{(i)}) \Delta z \Delta \phi + \lambda (v_\phi^{(+)} - v_\phi^{(-)}) \Delta r \Delta z \\
 &\quad + \lambda (v_z^{(t)} - v_z^{(b)}) r \Delta r \Delta \phi + \left( \lambda \frac{v_r^{(o)} + v_r^{(i)}}{2r} \right) r \Delta r \Delta z \Delta \phi \\
 4) \quad \dot{T}_{rr} &= \frac{(\lambda + 2\mu)}{r \Delta r} (r^{(o)} v_r^{(o)} - r^{(i)} v_r^{(i)}) + \frac{\lambda}{r \Delta \phi} (v_\phi^{(+)} - v_\phi^{(-)}) \\
 &\quad + \frac{\lambda}{\Delta z} (v_z^{(t)} - v_z^{(b)}) + \frac{\lambda}{2r} (v_r^{(o)} + v_r^{(i)})
 \end{aligned}$$

**Simulation Equation for  $T_{\phi\phi}$  (Equation 6.5)**

$$\begin{aligned}
 1) \quad \iiint \dot{T}_{\phi\phi} r dr dz d\phi &= \iiint \left( (\lambda + 2\mu) \left( \frac{v_r}{r} + \frac{1}{r} \frac{\partial v_\phi}{\partial \phi} \right) + \lambda \left( \frac{\partial v_r}{\partial r} + \frac{\partial v_z}{\partial z} \right) \right) r dr dz d\phi \\
 2) \quad &= \oint_{bV} \left( (\lambda + 2\mu) v_\phi dr dz + \lambda v_r r dz d\phi + \lambda v_z r dr d\phi \right) \\
 &+ \iiint \left( (\lambda + 2\mu) \frac{v_r}{r} \right) r dr dz d\phi \\
 3) \quad \dot{T}_{\phi\phi} r \Delta r \Delta z \Delta \phi &= (\lambda + 2\mu) (v_\phi^{(+)} - v_\phi^{(-)}) \Delta r \Delta z + \lambda (r^{(o)} v_r^{(o)} - r^{(i)} v_r^{(i)}) \Delta z \Delta \phi \\
 &+ \lambda (v_z^{(i)} - v_z^{(b)}) r \Delta r \Delta \phi + \left( (\lambda + 2\mu) \frac{(v_r^{(o)} + v_r^{(i)})}{2r} \right) r \Delta r \Delta z \Delta \phi \\
 4) \quad \dot{T}_{\phi\phi} &= \frac{(\lambda + 2\mu)}{r \Delta \phi} (v_\phi^{(+)} - v_\phi^{(-)}) + \frac{\lambda}{r \Delta r} (r^{(o)} v_r^{(o)} - r^{(i)} v_r^{(i)}) + \frac{\lambda}{\Delta z} (v_z^{(i)} - v_z^{(b)}) \\
 &+ \frac{(\lambda + 2\mu)}{2r} (v_r^{(o)} + v_r^{(i)})
 \end{aligned}$$

**Simulation Equation for  $T_{zz}$  (Equation 6.6)**

$$\begin{aligned}
 1) \quad \iiint \dot{T}_{zz} r dr dz d\phi &= \iiint \left( (\lambda + 2\mu) \frac{\partial v_z}{\partial z} + \lambda \left( \frac{v_r}{r} + \frac{1}{r} \frac{\partial v_\phi}{\partial \phi} + \frac{\partial v_r}{\partial r} \right) \right) r dr dz d\phi \\
 2) \quad &= \oint_{bV} \left( r (\lambda + 2\mu) v_z dr d\phi + \lambda v_\phi dr dz + r \lambda v_r dz d\phi \right) \\
 &+ \iiint \left( \lambda \frac{v_r}{r} \right) r dr dz d\phi \\
 3) \quad \dot{T}_{zz} r \Delta r \Delta z \Delta \phi &= (\lambda + 2\mu) (v_z^{(i)} - v_z^{(b)}) r \Delta r \Delta \phi + \lambda (v_\phi^{(+)} - v_\phi^{(-)}) \Delta r \Delta z \\
 &+ \lambda (r^{(o)} v_r^{(o)} - r^{(i)} v_r^{(i)}) \Delta z \Delta \phi + \left( \lambda \frac{(v_r^{(o)} + v_r^{(i)})}{2r} \right) r \Delta r \Delta z \Delta \phi \\
 4) \quad \dot{T}_{zz} &= \frac{(\lambda + 2\mu)}{\Delta z} (v_z^{(i)} - v_z^{(b)}) + \frac{\lambda}{r \Delta \phi} (v_\phi^{(+)} - v_\phi^{(-)}) + \frac{\lambda}{r \Delta r} (r^{(o)} v_r^{(o)} - r^{(i)} v_r^{(i)}) \\
 &+ \frac{\lambda}{2r} (v_r^{(o)} + v_r^{(i)})
 \end{aligned}$$



**Simulation Equation for  $T_{r\phi}$  (Equation 6.7)**

$$\begin{aligned}
 1) \quad \iiint \dot{T}_{r\phi} r dr dz d\phi &= \iiint \mu \left( \frac{1}{r} \frac{\partial v_r}{\partial \phi} + \frac{\partial v_\phi}{\partial r} - \frac{v_\phi}{r} \right) r dr dz d\phi \\
 2) &= \oint_{\partial V} (\mu v_r dr dz + r \mu v_\phi dz d\phi) - \iiint \left( \mu \frac{v_\phi}{r} \right) r dr dz d\phi \\
 3) \quad \dot{T}_{r\phi} r \Delta r \Delta z \Delta \phi &= \mu (v_r^{(+)} - v_r^{(-)}) \Delta r \Delta z + \mu (r^{(o)} v_\phi^{(o)} - r^{(i)} v_\phi^{(i)}) \Delta z \Delta \phi \\
 &\quad - \frac{\mu}{2r} (v_\phi^{(o)} + v_\phi^{(i)}) r \Delta r \Delta z \Delta \phi \\
 4) \quad \dot{T}_{r\phi} &= \frac{\mu}{r \Delta \phi} (v_r^{(+)} - v_r^{(-)}) + \frac{\mu}{r \Delta r} (r^{(o)} v_\phi^{(o)} - r^{(i)} v_\phi^{(i)}) - \frac{\mu}{2r} (v_\phi^{(o)} + v_\phi^{(i)})
 \end{aligned}$$

**Simulation Equation for  $T_{rz}$  (Equation 6.8)**

$$\begin{aligned}
 1) \quad \iiint \dot{T}_{rz} r dr dz d\phi &= \iiint \mu \left( \frac{\partial v_z}{\partial r} + \frac{\partial v_r}{\partial z} \right) r dr dz d\phi \\
 2) &= \oint_{\partial V} (\mu v_z r dz d\phi + \mu v_r r dr d\phi) \\
 3) \quad \dot{T}_{rz} r \Delta r \Delta z \Delta \phi &= \mu (r^{(o)} v_z^{(o)} - r^{(i)} v_z^{(i)}) \Delta z \Delta \phi + \mu (v_r^{(i)} - v_r^{(b)}) r \Delta r \Delta \phi \\
 4) \quad \dot{T}_{rz} &= \frac{\mu}{r \Delta r} (r^{(o)} v_z^{(o)} - r^{(i)} v_z^{(i)}) + \frac{\mu}{\Delta z} (v_r^{(i)} - v_r^{(b)})
 \end{aligned}$$

**Simulation Equation for  $T_{z\phi}$  (Equation 6.9)**

$$\begin{aligned}
 1) \quad \iiint \dot{T}_{z\phi} r dr dz d\phi &= \iiint \mu \left( \frac{\partial v_\phi}{\partial z} + \frac{1}{r} \frac{\partial v_z}{\partial \phi} \right) r dr dz d\phi \\
 2) &= \oint_{\partial V} (\mu v_\phi r dr d\phi + \mu v_z dr dz) \\
 3) \quad \dot{T}_{z\phi} r \Delta r \Delta z \Delta \phi &= \mu (v_\phi^{(i)} - v_\phi^{(b)}) r \Delta r \Delta \phi + \mu (v_z^{(+)} - v_z^{(-)}) \Delta r \Delta z \\
 4) \quad \dot{T}_{z\phi} &= \frac{\mu}{\Delta z} (v_\phi^{(i)} - v_\phi^{(b)}) + \frac{\mu}{r \Delta \phi} (v_z^{(+)} - v_z^{(-)})
 \end{aligned}$$

To recap, our final nine simulation equations are giving below

$$\rho \dot{v}_r = \frac{(r^{(o)} T_{rr}^{(o)} - r^{(i)} T_{rr}^{(i)})}{r \Delta r} + \frac{(T_{r\phi}^{(+)} - T_{r\phi}^{(-)})}{r \Delta \phi} + \frac{(T_{rz}^{(t)} - T_{rz}^{(b)})}{\Delta z} + \frac{T_{rr}^{(o)} + T_{rr}^{(i)} - T_{\phi\phi}^{(o)} - T_{\phi\phi}^{(i)}}{2r} + f_r \quad (6.14)$$

$$\rho \dot{v}_z = \frac{(r^{(o)} T_{rz}^{(o)} - r^{(i)} T_{rz}^{(i)})}{r \Delta r} + \frac{(T_{\phi z}^{(+)} - T_{\phi z}^{(-)})}{r \Delta \phi} + \frac{(T_{zz}^{(t)} - T_{zz}^{(b)})}{\Delta z} + \frac{(T_{rz}^{(o)} + T_{rz}^{(i)})}{2r} + f_z \quad (6.15)$$

$$\rho \dot{v}_\phi = \frac{(r^{(o)} T_{r\phi}^{(o)} - r^{(i)} T_{r\phi}^{(i)})}{r \Delta r} + \frac{(T_{\phi\phi}^{(+)} - T_{\phi\phi}^{(-)})}{r \Delta \phi} + \frac{(T_{z\phi}^{(t)} - T_{z\phi}^{(b)})}{\Delta z} + \frac{(T_{r\phi}^{(o)} + T_{r\phi}^{(i)})}{r} + f_\phi \quad (6.16)$$

$$\dot{T}_{rr} = \frac{(\lambda + 2\mu)}{r \Delta r} (r^{(o)} v_r^{(o)} - r^{(i)} v_r^{(i)}) + \frac{\lambda}{r \Delta \phi} (v_\phi^{(+)} - v_\phi^{(-)}) + \frac{\lambda}{\Delta z} (v_z^{(t)} - v_z^{(b)}) + \frac{\lambda}{2r} (v_r^{(o)} + v_r^{(i)}) \quad (6.17)$$

$$\dot{T}_{\phi\phi} = \frac{(\lambda + 2\mu)}{r \Delta \phi} (v_\phi^{(+)} - v_\phi^{(-)}) + \frac{\lambda}{r \Delta r} (r^{(o)} v_r^{(o)} - r^{(i)} v_r^{(i)}) + \frac{\lambda}{\Delta z} (v_z^{(t)} - v_z^{(b)}) + \frac{(\lambda + 2\mu)}{2r} (v_r^{(o)} + v_r^{(i)}) \quad (6.18)$$

$$\dot{T}_{zz} = \frac{(\lambda + 2\mu)}{\Delta z} (v_z^{(t)} - v_z^{(b)}) + \frac{\lambda}{r \Delta \phi} (v_\phi^{(+)} - v_\phi^{(-)}) + \frac{\lambda}{r \Delta r} (r^{(o)} v_r^{(o)} - r^{(i)} v_r^{(i)}) + \frac{\lambda}{2r} (v_r^{(o)} + v_r^{(i)}) \quad (6.19)$$

$$\dot{T}_{r\phi} = \frac{\mu}{r \Delta \phi} (v_r^{(+)} - v_r^{(-)}) + \frac{\mu}{r \Delta r} (r^{(o)} v_\phi^{(o)} - r^{(i)} v_\phi^{(i)}) - \frac{\mu}{2r} (v_\phi^{(o)} + v_\phi^{(i)}) \quad (6.20)$$

$$\dot{T}_{rz} = \frac{\mu}{r \Delta r} (r^{(o)} v_z^{(o)} - r^{(i)} v_z^{(i)}) + \frac{\mu}{\Delta z} (v_r^{(t)} - v_r^{(b)}) \quad (6.21)$$

$$\dot{T}_{z\phi} = \frac{\mu}{\Delta z} (v_\phi^{(t)} - v_\phi^{(b)}) + \frac{\mu}{r \Delta \phi} (v_z^{(+)} - v_z^{(-)}) \quad (6.22)$$

After having discretized the equations in space using the finite integration

technique, we approximate the time derivatives using the standard central difference

$$v_i^{(n)} = v_i^{(n-1)} - \dot{v}_i^{(n-1/2)} \Delta t \quad (6.23)$$

$$T_{ij}^{(n+1/2)} = T_{ij}^{(n-1/2)} - \dot{T}_{ij}^{(n-1/2)} \Delta t \quad (6.24)$$

Here the superscript  $n$  represents the time step. This leads to a temporal discretization that is staggered in time. With these equations, we can update each value in our simulation space based on the neighboring values.

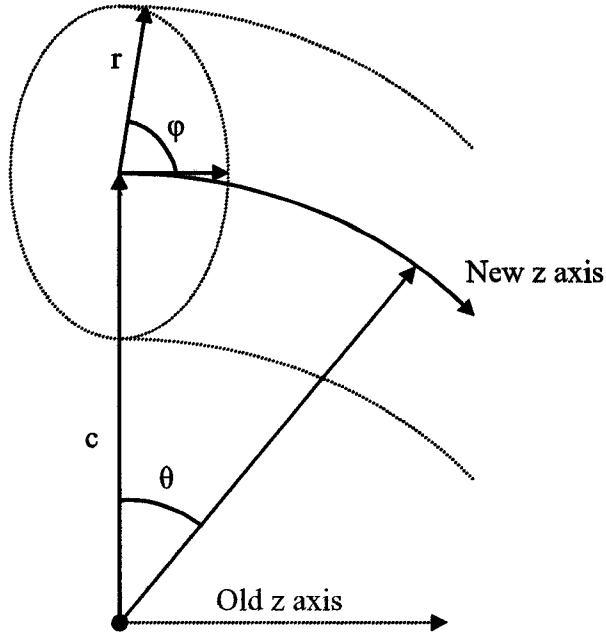
#### 6.4 Modifications for Pipe Bends

To describe elastic waves in pipe bends, we use a modified cylindrical coordinate system where the  $z$ -axis is shifted and then curved to follow the center of a pipe bend as shown in figure 6.2. The only effect this ultimately has on equations (6.1 to 6.9) is that  $dz$  is now replaced with the following:

$$dz = r \sin(\varphi - 0.5\pi + \varphi_{curvature}) + r_{curvature} \quad (6.25)$$

Here  $r_{curvature}$  is the radius of curvature of the pipe bend and  $\varphi_{curvature}$  is the angle that points towards the inside of the pipe bend.

After making this substitution, we again we use the finite integration technique to find our new simulation equations. Our control volume is similar to the one pictured in figure 6.2, except that the  $z$ -axis is now bent. Now, the size of the volume and its faces change depending where on the grid it is located. This requires us to keep track of our position on the grid so that we can properly account for the different size volumes. We rederive our simulation equations after making the substitution in (6.25) to find the following new simulation equations.



**Figure 6.2.** Curved cylindrical coordinate system. The z-axis is shifted and curved to follow the middle of a pipe bend.

$$p\dot{v}_r = \frac{(r^{(o)}c^{(o)}T_{rr}^{(o)} - r^{(i)}c^{(i)}T_{rr}^{(i)})}{r^{(c)}c^{(c)}\Delta r} + \frac{(T_{r\phi}^{(+)} - T_{r\phi}^{(-)})}{r^{(c)}\Delta\phi} + \frac{(T_{rz}^{(t)} - T_{rz}^{(b)})}{c^{(c)}\Delta\theta} + \frac{T_{rr}^{(o)} + T_{rr}^{(i)} - T_{\phi\phi}^{(o)} - T_{\phi\phi}^{(i)}}{2r^{(c)}} + f_r \quad (6.26)$$

$$p\dot{v}_z = \frac{(r^{(o)}c^{(o)}T_{rz}^{(o)} - r^{(i)}c^{(i)}T_{rz}^{(i)})}{r^{(c)}c^{(c)}\Delta r} + \frac{(T_{\phi z}^{(+)} - T_{\phi z}^{(-)})}{r^{(c)}\Delta\phi} + \frac{(T_{zz}^{(t)} - T_{zz}^{(b)})}{c^{(c)}\Delta\theta} + \frac{(T_{rz}^{(o)} + T_{rz}^{(i)})}{2r^{(c)}} + f_z \quad (6.27)$$

$$p\dot{v}_\phi = \frac{(r^{(o)}c^{(o)}T_{r\phi}^{(o)} - r^{(i)}c^{(i)}T_{r\phi}^{(i)})}{r^{(c)}c^{(c)}\Delta r} + \frac{(T_{\phi\phi}^{(+)} - T_{\phi\phi}^{(-)})}{r^{(c)}\Delta\phi} + \frac{(T_{z\phi}^{(t)} - T_{z\phi}^{(b)})}{c^{(c)}\Delta\theta} + \frac{(T_{r\phi}^{(o)} + T_{r\phi}^{(i)})}{r^{(c)}} + f_\phi \quad (6.28)$$

$$\dot{T}_{rr} = \frac{(\lambda + 2\mu)(r^{(o)}c^{(o)}v_r^{(o)} - r^{(i)}c^{(i)}v_r^{(i)})}{r^{(c)}c^{(c)}\Delta r} + \frac{\lambda(v_\phi^{(+)} - v_\phi^{(-)})}{r^{(c)}\Delta\phi} + \frac{\lambda(v_z^{(t)} - v_z^{(b)})}{c^{(c)}\Delta\theta} + \frac{\lambda(v_r^{(o)} + v_r^{(i)})}{2r^{(c)}} \quad (6.29)$$

$$\dot{T}_{zz}^{(n)} = \frac{(\lambda + 2\mu)(v_z^{(t)} - v_z^{(b)})}{c^{(c)} \Delta \theta} + \frac{\lambda(v_\phi^{(+)} - v_\phi^{(-)})}{r^{(c)} \Delta \phi} + \frac{\lambda(r^{(o)} c^{(o)} v_r^{(o)} - r^{(i)} c^{(i)} v_r^{(i)})}{r^{(c)} c^{(c)} \Delta r} + \frac{\lambda(v_r^{(o)} + v_r^{(i)})}{2r^{(c)}} \quad (6.30)$$

$$\dot{T}_{\phi\phi} = \frac{(\lambda + 2\mu)(v_\phi^{(+)} - v_\phi^{(-)})}{r^{(c)} \Delta \phi} + \frac{\lambda(c^{(o)} r^{(o)} v_r^{(o)} - c^{(i)} r^{(i)} v_r^{(i)})}{r^{(c)} c^{(c)} \Delta r} + \frac{\lambda(v_z^{(t)} - v_z^{(b)})}{c^{(c)} \Delta \theta} + \frac{(\lambda + 2\mu)(v_r^{(o)} + v_r^{(i)})}{2r^{(c)}} \quad (6.31)$$

$$\dot{T}_{rz} = \frac{\mu(r^{(o)} c^{(o)} v_z^{(o)} - r^{(i)} c^{(i)} v_z^{(i)})}{r^{(c)} c^{(c)} \Delta r} + \frac{\mu(v_r^{(t)} - v_r^{(b)})}{c^{(c)} \Delta \theta} \quad (6.32)$$

$$\dot{T}_{r\phi} = \frac{\mu(v_r^{(+)} - v_r^{(-)})}{r^{(c)} \Delta \phi} + \frac{\mu(r^{(o)} c^{(o)} v_\phi^{(o)} - r^{(i)} c^{(i)} v_\phi^{(i)})}{r^{(c)} c^{(c)} \Delta r} - \frac{\mu(v_\phi^{(o)} + v_\phi^{(i)})}{2r^{(c)}} \quad (6.33)$$

$$\dot{T}_{z\phi} = \frac{\mu(v_\phi^{(t)} - v_\phi^{(b)})}{c^{(c)} \Delta \theta} + \frac{\mu(v_z^{(+)} - v_z^{(-)})}{r^{(c)} \Delta \phi} \quad (6.34)$$

Here  $r^{(i)}$ ,  $r^{(c)}$ , and  $r^{(o)}$  are the radial distances measured from the curved z-axis to the inside, center, and outside of the control volume. The variables  $c^{(i)}$ ,  $c^{(c)}$ , and  $c^{(o)}$  are the distances measured from the axis of curvature to the inside, center, and outside of the control volume. These values are a function of the current radial position, the radius of curvature  $r_{curvature}$ , and the angle pointing towards the inside of the bend  $\phi_{curvature}$ .

$$c^{(i)} = (r^{(i)} \sin(\phi - 0.5\pi + \phi_{curvature}) + r_{curvature}) \quad (6.35)$$

$$c^{(c)} = (r^{(c)} \sin(\phi - 0.5\pi + \phi_{curvature}) + r_{curvature}) \quad (6.36)$$

$$c^{(o)} = (r^{(o)} \sin(\phi - 0.5\pi + \phi_{curvature}) + r_{curvature}) \quad (6.37)$$

To simulate a straight pipe section with these equations, we set  $c^{(i)} = c^{(c)} = c^{(o)} = 1$  and  $\Delta\theta = \Delta z$ . With these equations, it is straightforward to simulate elastic waves in pipe segments containing combinations of straight sections and bends in any direction, including multiple out-of-plane bends.

## 6.5 Stability Criteria

The spatial discretization in the radial direction is identical to the one derived by Schubert in his axially-symmetric cylindrical finite integration technique [5]. We assign 8 grid points to the smallest wavelength present in the simulation. In elastodynamics, this is typically the Rayleigh wave, which is at most 13% slower than the shear wave, so we assign 10 grid points to the shear wavelength.

$$\Delta r \approx \left( \frac{1}{10} \right) \frac{c_s}{f_{\max}} \quad (6.38)$$

Here,  $c_s$  is the shear wave speed and  $f_{\max}$  is the maximum frequency present in the simulation. The exact radial step size is adjusted as needed to simulate the correct pipe wall thickness. Next, we choose  $\Delta\varphi$  so that the grid spacing on the outer circumference of the pipe ( $r_{\text{outer}}\Delta\varphi$ ) is equal or smaller than  $\Delta r$  and that the total number of grid points in the  $\varphi$  direction sweeps an angle of exactly  $2\pi$ . First we find the number of grid points in the  $\varphi$ -direction  $N_\varphi$ .

$$N_\varphi = \left\lceil \frac{2\pi r_{\text{outer}}}{\Delta r} \right\rceil \quad (6.39)$$

Now we find the exact discretization angle  $\Delta\varphi$ .

$$\Delta\varphi = \frac{2\pi}{N_\varphi} \quad (6.40)$$

Again, we choose  $\Delta\theta$  so that the grid spacing on the outside of a bend is equal or less than  $\Delta r$ . First we find the number of grid points in the  $z$ -direction through the curve that sweeps an angle of  $S$ .

$$N_\theta = \left\lceil \frac{S(r_{outer} + r_{curve})}{\Delta r} \right\rceil \quad (6.41)$$

Now we find  $\Delta\theta$  as

$$\Delta\theta = \frac{S}{N_\theta} \quad (6.42)$$

If a straight section is being simulated, than we set  $\Delta\theta = \Delta z \leq \Delta r$ . The temporal discretization is found using the fastest wave speed and the smallest spatial grid sizes. In the  $\varphi$ -direction, this is found on the inside circumference of the pipe. In the  $\theta$ -direction, this is found on the outer edge of the pipe on the inside of the pipe-bend. We use the grid spacing at these locations to compute our time step  $\Delta t$ .

$$\Delta t \leq \frac{1}{c_l \sqrt{\frac{1}{\Delta r^2} + \frac{1}{((r_{curve} - r_{outer})\Delta\theta)^2} + \frac{1}{(r_{inner}\Delta\varphi)^2}}} \quad (6.42)$$

If a simulation models a pipe segment containing multiple bends of different curvatures, then the smallest curvature must be used to determine the temporal step size.

## 6.6 Boundary Conditions

Here we derive the stress-free boundary conditions at the surfaces of the pipe. Even with added complexity of the full 3D space and the curved z-axis, we arrive at the same boundary conditions as Shubert[5] in the axially symmetric cylindrical case. We begin by enforcing that the velocity components be placed on the surface of the pipe. At the inner and outer surface of the pipe, we want the stress components  $T_{rr}$ ,  $T_{rz}$ , and  $T_{r\phi}$  to be zero. Since the  $T_{rz}$ , and  $T_{r\phi}$  stress components are on the surface, we simply set them to zero. To enforce that  $T_{rr}$  is zero at the surface, we set  $T_{rr}^{(o)} = -T_{rr}^{(i)}$  at the outer surface of the pipe and  $T_{rr}^{(i)} = -T_{rr}^{(o)}$  at the inner surface of the pipe. We then extrapolate to find the  $T_{\phi\phi}$  term that is outside the boundary of the pipe. This leads to the following algorithm for finding  $v_r$ .

*At the outer surface of the pipe ( $r = r_{maz}$ ).*

We set  $T_{rr}^{(o)} = -T_{rr}^{(i)}$ ,  $T_{\phi\phi}^{(o)} = 2T_{\phi\phi}^{(i)} - T_{\phi\phi}^{(ii)}$ ,  $T_{rr} = T_{rz} = T_{r\phi} = 0$ .

$$p\dot{v}_r = -\frac{2T_{rr}^{(i)}}{\Delta r} - \frac{3T_{\phi\phi}^{(i)} - T_{\phi\phi}^{(ii)}}{2r^{(c)}} + f_r \quad (6.43)$$

*At the inner surface of the pipe ( $r = r_{min}$ ).*

We set  $T_{rr}^{(i)} = -T_{rr}^{(o)}$ ,  $T_{\phi\phi}^{(i)} = 2T_{\phi\phi}^{(o)} - T_{\phi\phi}^{(oo)}$ ,  $T_{rr} = T_{rz} = T_{r\phi} = 0$ .

$$p\dot{v}_r = \frac{2T_{rr}^{(o)}}{\Delta r} - \frac{3T_{\phi\phi}^{(o)} - T_{\phi\phi}^{(oo)}}{2r^{(c)}} + f_r \quad (6.44)$$



The same procedure is carried for the z-velocity components  $v_z$  equations on the pipe ends.

*At the ends of the pipe ( $z = z_{min}$ ).*

We set  $T_{zz}^{(b)} = -T_{zz}^{(t)}$ ,  $T_{zz} = T_{rz} = T_{z\varphi} = 0$ .

$$p\dot{v}_z = -\frac{2T_{zz}^{(t)}}{c^{(c)}\Delta\theta} + f_z \quad (6.45)$$

*At the ends of the pipe ( $z = z_{max}$ ).*

We set  $T_{zz}^{(t)} = -T_{zz}^{(b)}$ ,  $T_{zz} = T_{rz} = T_{z\varphi} = 0$ .

$$p\dot{v}_z = \frac{2T_{zz}^{(b)}}{c^{(c)}\Delta\theta} + f_z \quad (6.46)$$

## 6.7 Absorbing Boundary Layers

In most simulations, it is important to eliminate or significantly reduce wave reflections from the pipe ends, i.e. when the actual pipe is longer than the simulation space allows. This is accomplished by adding absorbing boundary regions to the end(s) of the simulated pipe. To do this, the velocities in the absorbing region are computed as usual but with a small damping factor. The damping factor is a function of the distance from the inside position in the absorbing layer  $a$  and the percentage  $p$  in which each layer removes from the velocity values. We replace (6.23) with the following equation.

$$v_i^{(n)} = v_i^{(n-1)} - D\dot{v}_i^{(n-1/2)}\Delta t \quad (6.47)$$

Where the damping factor  $D$  is given by

$$D = (1 - p * a) \quad (6.48)$$

The damping factor is zero at the inside of the region and steadily increases as one moves to the outside of the absorbing region as shown in figure 6.3. This technique can be used to significantly reduce reflections on all acoustic and elastic finite integration methods. For the finite integration technique described in this paper, we found that an absorbing region 40-70 nodes thick with a damping percentage of 0.2% ( $p=0.002$ ) worked well for reducing reflections from the artificial pipe ends.

## 6.8 Parallel Implementation

While some results can be obtained using the 3DCEFIT technique on a standard desktop computer, substantial improvements in computational time and model complexity are achieved with a parallel implementation. A parallel version of the 3DCEFIT has been implemented on William and Mary's high performance computational cluster, the SciClone. At the time of this work, the SciClone was composed of 311 computer processors with 236 GB of physical memory and 15.1 TB of disk capacity, and with a peak performance of 362 billion floating point operations per second.

The parallel algorithm uses a straight forward domain decomposition approach to divide the simulation space across many computers. Similar decomposition methods can be found in [9]. After every half time step of the simulation, each computer swaps the appropriate boundary values with neighboring computers to create a large and seamless simulation space. Most, if not all message passing interfaces allow blocking and non-blocking routines to send and receive data. We use a combination of these routines to achieve the most optimized parallel algorithm. The parallel algorithm is given below.

#### Parallel Algorithm

- 1) Compute boundary velocity values
- 2) Send the new boundary values to appropriate neighboring computers (using a MPI non-blocking send)
- 3) Compute the rest of the velocity values
- 4) Receive the boundary velocity values from neighbors (using a MPI blocking receive)
- 5) Repeat these 4 steps with the stress values.

Simulations that take many hours to complete on a single high-end desktop PC take just minutes using this parallel implementation.

## **6.9 Conclusions**

In this chapter, we have presented all the necessary equations and stability and boundary conditions to simulate elastic waves in pipes. In the following chapter, we will validate this simulation method by comparing simulation results directly to experimental data and to results from a commercial finite-element simulation package. We will also present several applied examples of how this simulation technique can be used to solve real-world problems in hardware and signal processing design.

## 6.10 References

1. D. Gsell, T.L., and J. Dual, *Modeling three-dimensional elastic wave propagation in circular cylindrical structures using a finite-difference approach*. Journal of Acoustical Society of America, 2004. **116**(6): p. 3284-3293.
2. T. Leutenegger, J.D., *Detection of defects in cylindrical structures using a time reverse method and a finite-difference approach*. Ultrasonics, 2002. **40**: p. 721–725.
3. T. Hayashi, K.K., Z. Q. Sun, and J. L. Rose, *Guided wave propagation mechanics across a pipe elbow*. Journal of Pressure Vessel Technology-Transactions of the Asme, 2005. **137**(3): p. 322-327.
4. T. Hayashi, K.K., Z. Q. Sun, and J. L. Rose, *Analysis of flexural mode focusing by a semianalytical finite element method*. Journal of Acoustical Society of America, 2003. **113**(3): p. 1241-1248.
5. F. Schubert, A.P., and B. Kohler, *The elastodynamic finite integration technique for waves of cylindrical geometries*. Journal of the Acoustical Society of America, 1998. **104**(5): p. 2604-2614.
6. P. Fellingner, R.M., K.J. Langenberg, and S. Klaholz, *Numerical modeling of elastic wave propagation and scattering with EFIT - elastodynamic finite integration technique*. Wave Motion, 1995. **21**: p. 47-66.
7. Schubert, F., *Numerical time-domain modeling of linear and nonlinear ultrasonic wave propagation using finite integration techniques--theory and applications*. Ultrasonics, 2004. **42**(42): p. 221-229.
8. Graff, K.F., *Wave Motion In Elastic Solids*. 1991, New York: Dover Publications.
9. Marklein, R., *Numerical Simulation of Fields and Waves in Nondestructive Testing*. 9th European Conference on Non-Destructive Testing, Berlin, 2006.
10. Harker, A.H., *Elastic Waves in Solids - With Applications to Nondestructive Testing of Pipelines*. 1988: British Gas.
11. J. L. Rose, D.J., and J. Spanner Jr, *Ultrasonic Guided Wave NDE for Piping*. Materials Evaluation, 1996. **54**(11): p. 1310-1313.
12. Alleyne, D.N.a.P.C., *Long Range Propagation of Lamb Waves in Chemical Plant Pipework*. Materials Evaluation, 1997. **55**(4): p. 504-508.
13. M.J.S. Lowe, D.N.A., and P. Cawley, *Defect detection in pipes using guided waves*. Ultrasonics, 1998. **36**: p. 147-154.
14. D. Alleyne, e.a., *The Lamb wave inspection of chemical plant pipework*. Review of Progress in QNDE, 1997: p. 1269-1276.

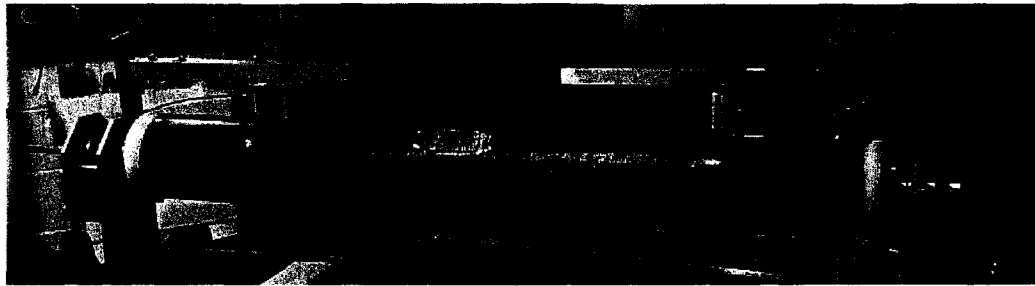
## **Chapter VII**

### **Applied Simulations: Guided Waves in Complex Piping Geometries**

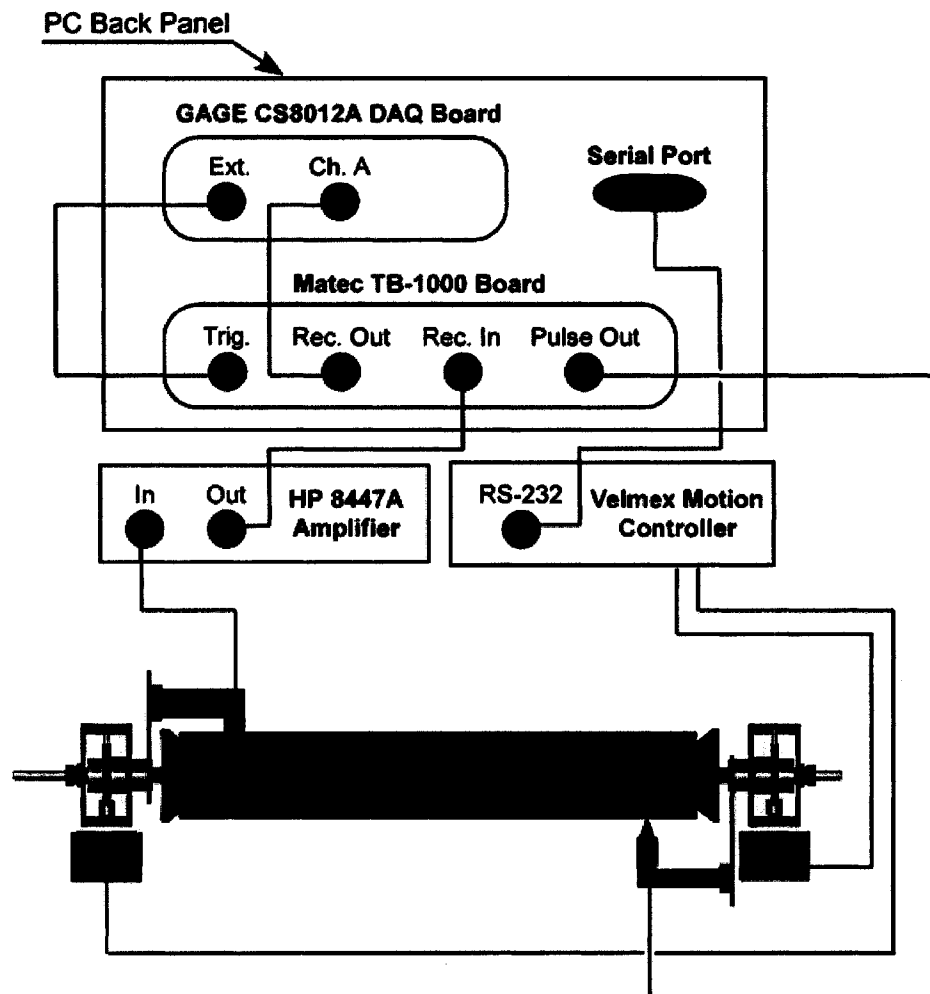
#### **7.1 Introduction**

Ultrasonic guided waves have been used successfully for nondestructive evaluation of a wide variety of structures [1-3]. Using ultrasonic guided wave methods to inspect piping systems hold great promise [4-8] but to successfully develop an ultrasonic guided wave system for remotely inspecting beyond pipe elbows it is first necessary to understand in detail how elastic waves propagate through and beyond pipe bends. With sufficient computational resources, this can be accomplished efficiently using 3D numerical simulations.

The 3DPCEFIT method allows us to accurately and systematically simulate the interaction of guided elastic waves with arbitrary flaws in complex piping structures in order to optimize ultrasonic guided wave pipe inspection protocols. In this Chapter we validate the 3DPCEFIT technique by directly comparing simulation and experimental results for a straight pipe. We then present several techniques for focusing guided waves on a pipe. For pipe bends, we compare simulation results directly to results obtained using a commercial finite element software package.



**Figure 7.1.** A photograph of the experimental apparatus. The steel pipe segment is 3 feet long, has an inner diameter of 4 inches, and a wall thickness of 0.25 inches.

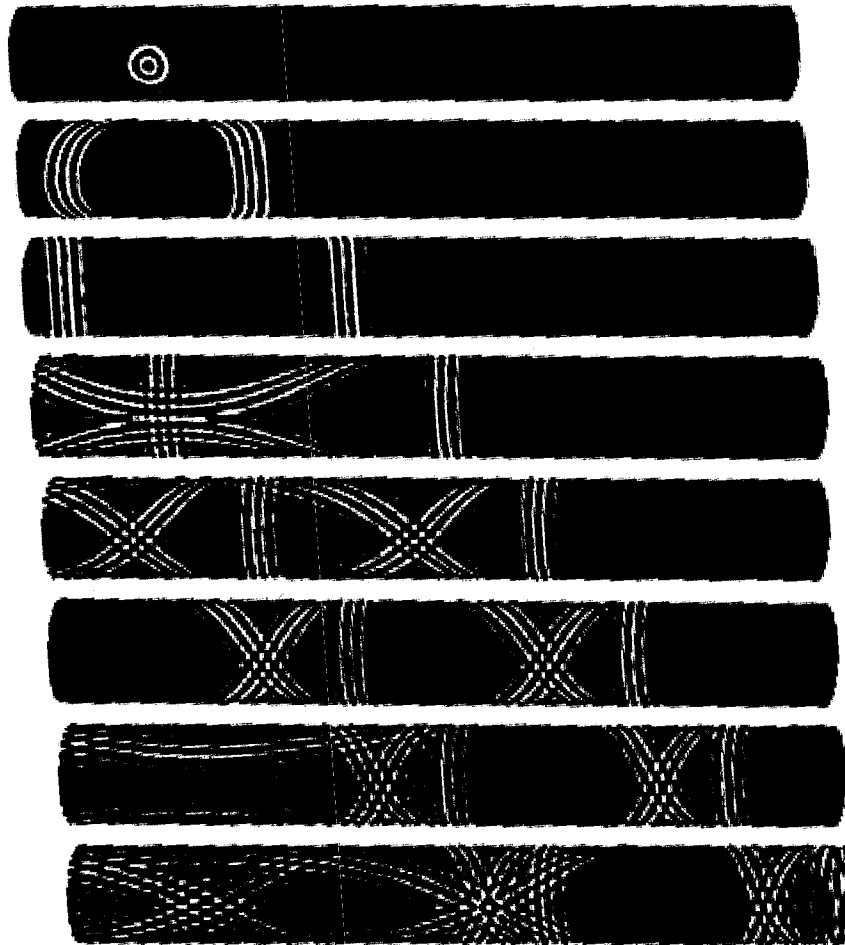


**Figure 7.2.** Line diagram showing the major components and connections of the experimental apparatus.

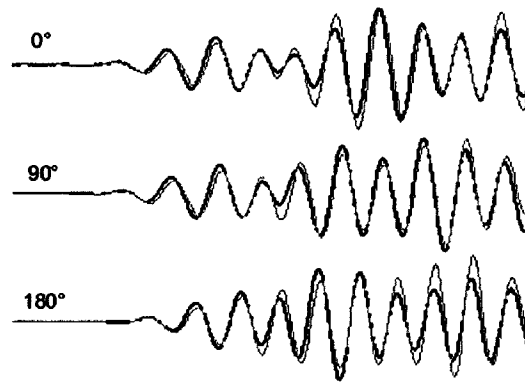
## **7.2. Comparison with Experimental Data**

To validate this simulation method, we compared simulated and experimental ultrasonic guided waveforms. This was done by performing pitch-catch measurements and corresponding simulations on a straight pipe section mounted in a laboratory scanner [9-12] as shown in figure 7.1 and figure 7.2. A-line waveforms were directly compared for three different catch transducer locations located at  $0^\circ$ ,  $90^\circ$ , and  $180^\circ$  and at a separation distance of 60cm from the pitch transducers. The longitudinal contact transducers are 3cm in diameter and the pitch transducer is driven with a short 200 kHz toneburst. Figure 7.3 shows several snapshots in time from this simulation. The gray-scale intensity of these plots is proportional to the radial displacement on the outer surface of the pipe. Absorbing boundary conditions were not included in this simulation in order to closely match the experimental set-up, i.e. a short segment of pipe. It can be seen from this figure how the presence of multiple modes, end reflections, and wrap around waves can make signal interpretation quite difficult. Figure 7.4 shows normalized A-line waveforms from the simulation and experiment plotted together. The A-lines waveforms match well at all three locations which gives us confidence that our simulation is accurately describing guided wave propagation in pipe-like structures.





**Figure 7.3.** Snapshots of a three dimensional 200 kHz pipe simulation. A single 3cm transducer is driven with a short 200kHz tone-burst. The gray scale color intensity is proportional to the radial displacement on the outer surface of the pipe.

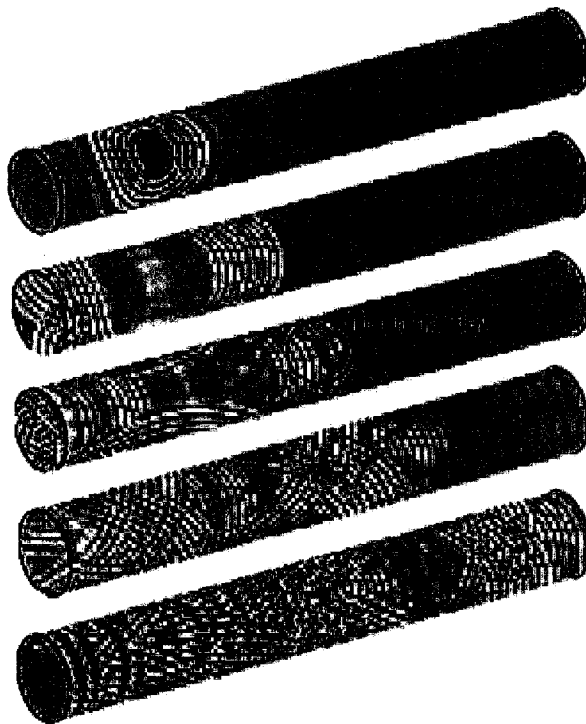


**Figure 7.4.** Comparison between the simulated (gray) and the experimental (black) A-line data recorded at three different locations on the pipe showing very good agreement.

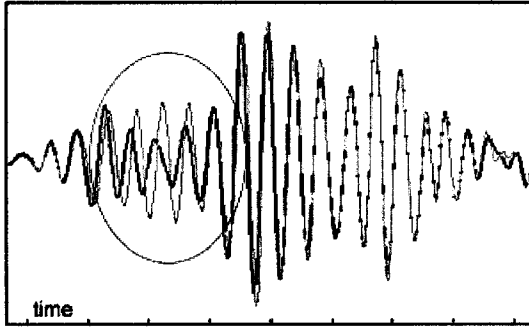
### 7.3 Guided Wave Scattering From Flaws

We next present simulation results from a pipe segment with a small thinning area located in the center of the pipe. The pipe dimensions are identical to the pipe simulated in the previous section. Figure 7.5 shows several snapshots from the simulation, while figure 7.6 shows two recorded A-lines taken from a clean and flawed pipe segment, respectively. The differences in the two A-lines are apparent at the beginning of the signals where one of the guided wave modes has shifted in time and changed amplitude. A systematic study of the interaction of guided waves with flaws and the resulting changes in the recorded A-line measurements would greatly benefit the development of automatic flaw detection algorithms.

Figure 7.6 demonstrates how small the differences are in the recorded waveforms between a corrosive-type flaw and unflawed pipe sample. Corrosive flaws are gradual thinnings that do not reflect strongly compared to other types of flaws such as saw cuts or flat-bottomed holes. The problem is compounded if the flaw is small and located in a hard to reach area such as underground or beyond pipe bends. For these reasons, it is advantageous to be able to focus guided wave energy at long distances and beyond pipe bends. The focal spot can then be walked about the circumference of the pipe and down the axis to inspect the structure completely for flaws.



**Figure 7.5.** Snapshots of a three dimensional pipe simulation with a small thinning flaw located in the center of the pipe. The color is proportional to the radial displacement on the outer surface of the pipe.



**Figure 7.6.** Comparison of A-line data from a clean (black) and flawed (gray) pipe segment. The circled region shows where one of the guided wave modes has shifted and changed in amplitude because of the interaction with the flaw.

## 7.4 Focusing Techniques

### 7.4.1. Focusing with Hardware: Phased Array Transducer Belts

Guided wave focusing in pipes is typically done with phased array transducer belts [13]. The timing and amplitude of each excitation waveform are adjusted such that the desired guided-wave mode from each transducer arrives at the focal spot at the same time. The delay of each transducer  $d$  can be found given the velocity  $v$  of the desired guided wave mode and the shortest distance  $s$  between the given transducer and focal point. The delay  $d$  is given by the following equation

$$d = \frac{s}{v} \quad (7.1)$$

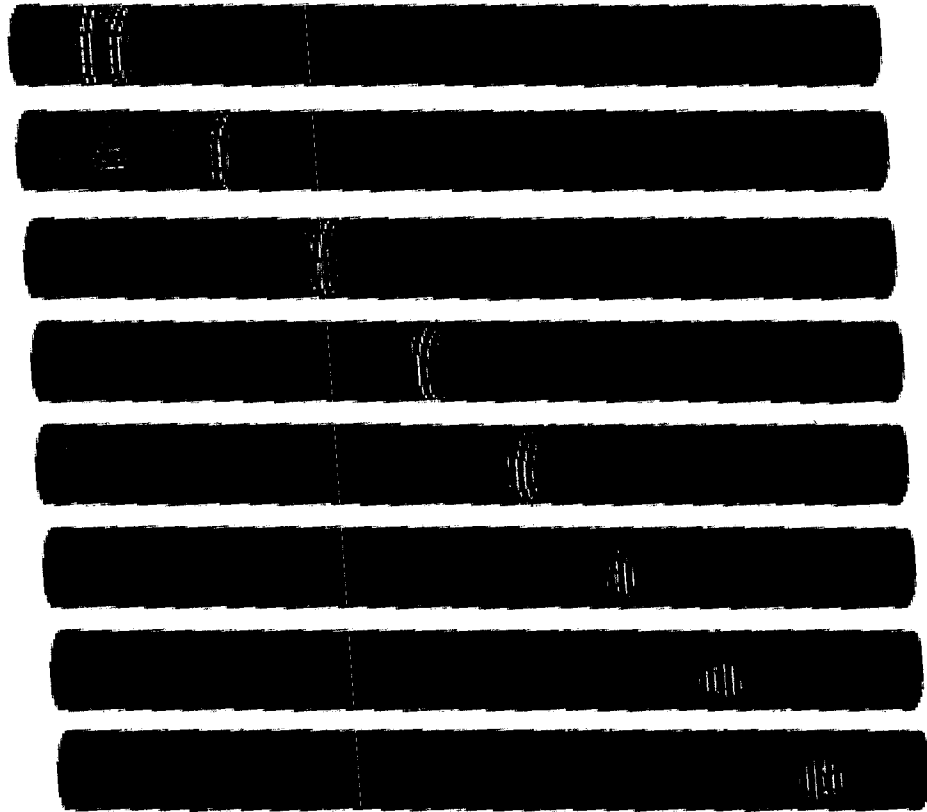
For a straight pipe section, the shortest distance between the transducer and the focal point is given by

$$s = \sqrt{f_z^2 + \gamma r^2} \quad (7.2)$$

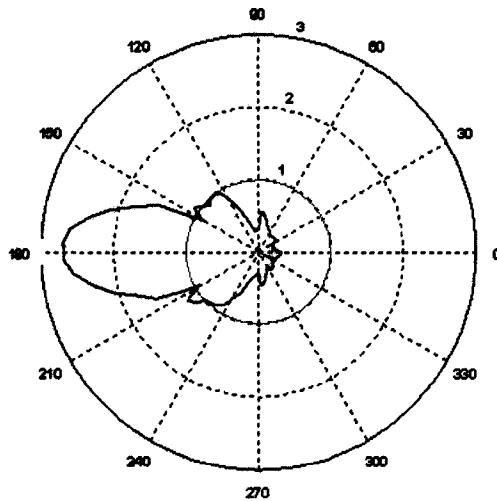
where  $f_z$  is the axial distance between the transducer and the focal point,  $r$  is the radius of the pipe, and  $\gamma$  is the smallest angle between the transducer and focal point. This angle  $\gamma$  is given by the following equation

$$\gamma = \min \begin{cases} (a_\phi - f_\phi + 2\pi)^2 \\ (a_\phi - f_\phi)^2 \\ (a_\phi - f_\phi - 2\pi)^2 \end{cases} \quad (7.3)$$

Here,  $a_\phi$  and  $f_\phi$  are the angles in radians of the transducer and the focal point, respectively. The following simulation demonstrates phased array focusing on a straight pipe section. A phased array of thirty-two 1cm diameter normal-incidence contact transducers evenly spaced along the circumference of a 4" inner diameter steel pipe with a wall thickness of 0.25" is simulated. All the transducers are driven with the same 200kHz tone-burst excitation except they are delayed according to equation (7.1) such that the desired guided wave mode arrives at the focal point, 1 meter from the transducer belt, at the same time. Figure 7.7 shows several snapshots from this simulation. Figure 7.8 compares the energy distribution on the circumference of the pipe at one meter for the focused simulation (black) and an axially-symmetric wave (gray). The energy of the axially-symmetric wave is evenly distributed across the circumference while in the focused simulation, the energy is concentrated at 180°.



**Figure 7.7.** Snapshots of a three dimensional 200 kHz pipe simulation. A transducer belt containing 32 omni-directional transducers. The transducers fire out of phase such that the desired guided wave becomes focused one meter from the transducer belt.



**Figure 7.8.** Polar plot comparing the energy distribution on the circumference of the pipe at one meter for an axial symmetric wave (gray circle) and the focused wave (black). For an axial symmetric wave, the energy is uniformly distributed around the pipe. Using a phased array transducer belt and the focusing algorithm, the energy becomes concentrated at 180 degrees.

#### 7.4.2. Focusing in Software: The Synthetic Aperture Focusing Technique (SAFT)

The Synthetic Aperture Focusing Technique (SAFT) is a numerical method for focusing wave fields. SAFT was originally developed for radar applications and has been adapted by the NDE community for improving lateral resolution and imaging quality [14]. The SAFT technique has been used successfully with Lamb waves for locating and identifying flaws in plate-like structures [15]. Here we implement a time-domain SAFT technique for focusing Lamb waves in pipe structures.

One of the advantages of this technique is that no complicated phased array hardware is required. Instead, each transducer in the array fires individually while A-lines are recorded at each of the catch transducer locations. These A-lines are stored and then later combined using the SAFT algorithm to synthetically focus the ultrasound waves onto any location on the pipe. The A-line waveforms are shifted in time and summed up such that the guided waves arrived at the focal point at the same time. The time shift of each waveform is identical to the time shift we previously used for phased

array focusing. The following equation produces a new A-line  $A_c$  at a given transducer location  $c$  by combining all the A-lines recorded at that location.

$$SA_c(t) = \sum_{n=1}^N A_{n,c}(t + d) \quad (7.4)$$

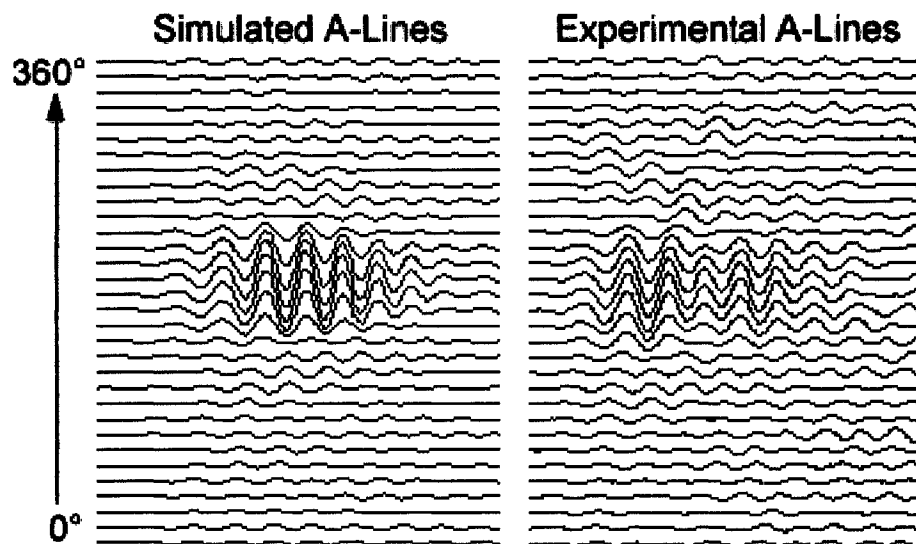
Here  $N$  is the number of pitch transducers,  $SA_c(t)$  is the new SAFT constructed A-line at catch transducer location  $c$ ,  $A_{n,c}(t)$  is the recorded A-line from catch transducer  $c$  when the pitch transducer  $n$  fired, and  $d$  is the same delay we computed in equation (7.1). It should be noted that there are frequency domain versions of the SAFT algorithm that are more computationally efficient [14]. A simulation was performed to validate the SAFT algorithm, using a one meter pipe with 32 pitch and 32 catch transducers separated by 60 centimeters was simulated. Each pitch transducer fires individually while all 32 catch transducers record the radial pipe displacements over time (A-lines). This creates 32 A-lines recorded at each catch transducer location with 1024 A-lines in total. Figure 7.9 shows SAFT A-lines where the focal point was chosen to be at the location of one of the catch transducers. The focused guided wave mode is clearly visible in the new A-line data.

The SAFT technique also works well with experimental data. The same set-up as described above was performed experimentally. The same pitch and catch transducer locations were recorded experimentally so that the results could be compared to the simulation results. Figure 7.9 shows the SAFT results with the experimental A-lines. Figure 7.10 shows a polar plot comparing the energy distribution on the pipe for the simulated and experimental data. In these examples, the focal point was chosen to be

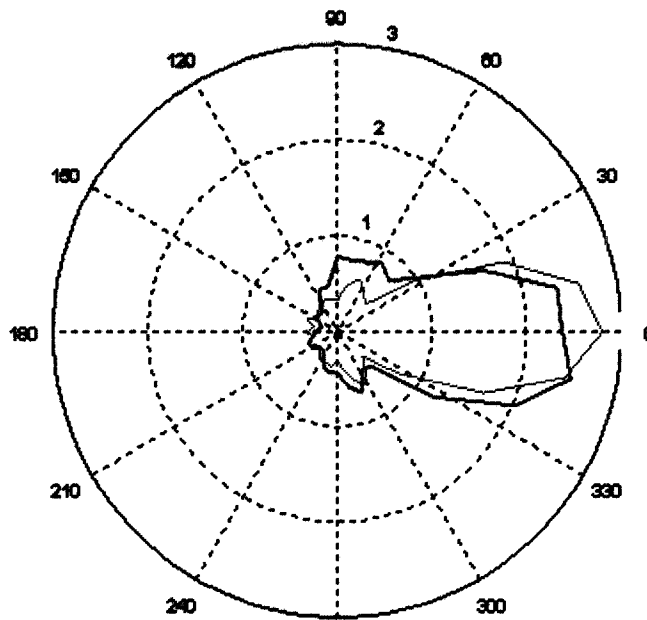


located along the catch transducer ring. In practice, the focal point would be swept across the entire surface of the pipe or sections of interest to image the pipe for flaws.

There is good agreement between the experimental and simulation results, and thus gives an example of how simulations of this type can be used to develop signal processing techniques that may be difficult to refine using experimental data alone. With the SAFT algorithm, the focal point can be swept along the pipe in software to inspect it for flaws, which can substantially reduce the cost and complexity of the experimental apparatus.



**Figure 7.9.** A-lines from produced by SAFT with the focal point at the catch transducer located at 180 degrees. Results are shown for simulated data (left) and experimental data (right). In both cases, the  $S_0$  mode is focused at 180.



**Figure 7.10.** Polar plot showing the energy distribution on the circumference of the pipe at the focal point using the SAFT algorithm. There is very good agreement between the experimental (black) and simulated (gray) results.

### 7.5 Pipe Bend Simulations

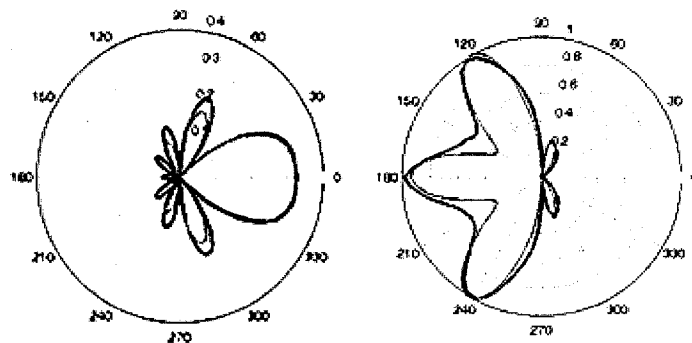
Most piping systems contain bends which make inspecting them problematic. The 3DCEFIT technique as described above can simulate elastic guided waves propagation in piping systems that contain combinations of bends of various curvature and in any direction. Figure 7.11 shows snapshots from an initially axial-symmetric 50 kHz guided wave propagating through a 90 degree pipe bend with radius of curvature of 6 inches. The axi-symmetric wave was created with a transducer belt consisting of 32 omni-directional 1cm diameter transducers. As expected, the guided waves naturally focus at the back of the pipe bend.

To validate the simulation method for pipes with bends, we compare our simulation results directly to results obtained with the commercial finite element software package COMSOL. The pipe is symmetrically excited with 5 cycle 100kHz shear transducer on the outer radius of the pipe. The pipe segment contains a 90 degree bend of the same dimensions as the previous example, located 1.5 meters from the transducer belt. The shear displacement on the outer surface of the pipe on the end of the bend opposite the transducer belt was recorded over time. The shear displacements obtained using COMSOL and 3DCEFIT at two time instances are shown in figure 7.12 and reveal very good agreement. For comparison, this simulation took roughly 18 hours to run using COMSOL on a high end desktop computer. The 3DCEFIT simulation took 7 minutes on a 64 node parallel computer (650 MHz processor per node). Similarities have also been observed between simulation and experimental results. The pipe segment used to obtain experimental results contained multiple welds which complicated the comparison so they will not be presented here.

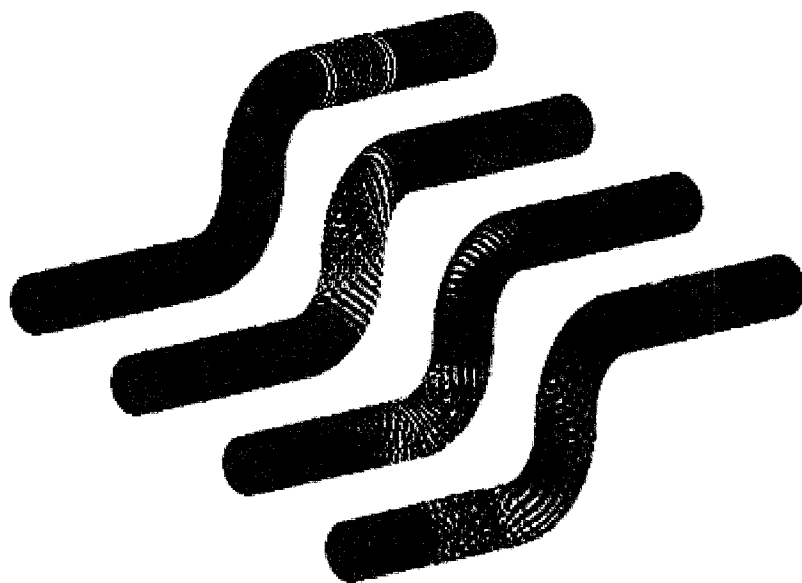
Figure 7.13 shows a 100 kHz guided wave propagating through an S-bend and figure 7.14 shows a 100 kHz guided wave propagating through a series of 3D pipe bends. These figures demonstrate the ability to simulate complex piping systems containing multiple bends. This simulation method can also be used to simulate guided waves in pipe coils that are routinely found in heating and cooling systems and power plants.



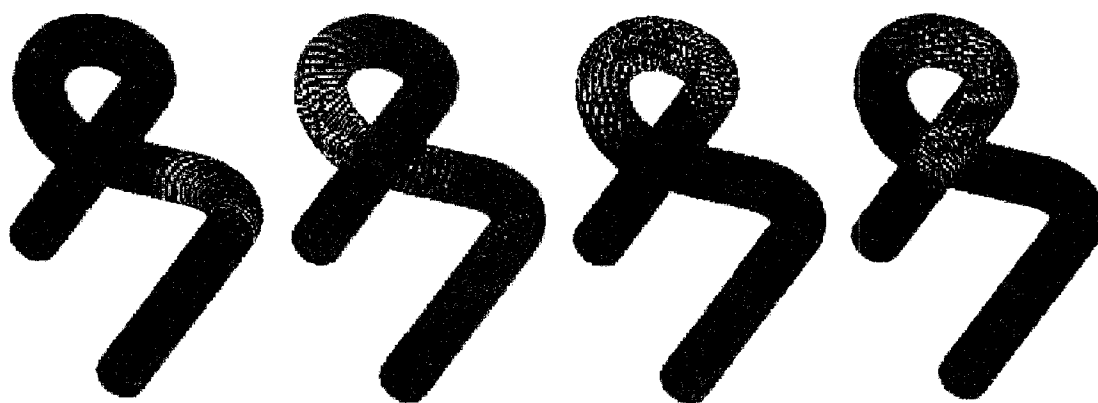
**Figure 7.11.** A 50 kHz guided elastic wave propagates through a 90 degree pipe bend. The guided waves naturally focus at the back of the bend.



**Figure 7.12.** Polar plots of the shear displacements obtained from a COMSOL finite element simulation (black) and a 3DCEFIT simulation (gray). The shear displacements were recorded at 361 $\mu$ S (left) and 390 $\mu$ S (right) from the initial transducer excitation.



**Figure 7.13.** A 100 kHz guided elastic wave propagates through a pipe S-bend.



**Figure 7.14.** A 100 kHz guided elastic wave propagates through many 3D pipe bends. The simulations provide a way to predict the path of the guided waves in complicated piping systems.

### **7.6 3DCEFIT Conclusions**

We have developed and implemented a 3D simulation method based on the finite integration technique for modeling guided elastic waves in pipe-like structures including pipe bends. Comparisons show agreement between simulated and experimental data, and we have shown that the finite integration technique is well suited for modeling elastic wave propagation and interactions with flaws. This simulation method can be used as a design tool for developing complicated inspection hardware and signal processing algorithms. Future work will focus on systematically studying guided wave interaction with varying flaw types and sizes. Discretizing the material parameters will also allow one to model piping systems which contain coatings and welds. Overall, the 3DCEFIT technique provides an accurate method for simulating guided elastic waves in complex piping systems.

## 7.7 References

1. Rose, J.L., *Ultrasonic Waves in Solid Media*. 1999: Cambridge University Press.
2. Rose, J.L., *A baseline and vision of ultrasonic guided wave inspection potential*. *Journal of Pressure Vessel Technology-Transactions of the Asme*, 2002. **124**(3): p. 273-282.
3. Rose, J.L., *Standing on the shoulders of giants: An example of guided wave inspection*. *Materials Evaluation*, 2002. **60**(1): p. 53-59.
4. Harker, A.H., *Elastic Waves in Solids - With Applications to Nondestructive Testing of Pipelines*. 1988: British Gas.
5. J. L. Rose, D.J., and J. Spanner Jr, *Ultrasonic Guided Wave NDE for Piping*. *Materials Evaluation*, 1996. **54**(11): p. 1310-1313.
6. Alleyne, D.N.a.P.C., *Long Range Propagation of Lamb Waves in Chemical Plant Pipework*. *Materials Evaluation*, 1997. **55**(4): p. 504-508.
7. M.J.S. Lowe, D.N.A., and P. Cawley, *Defect detection in pipes using guided waves*. *Ultrasonics*, 1998. **36**: p. 147-154.
8. D. Alleyne, e.a., *The Lamb wave inspection of chemical plant pipework*. *Review of Progress in QNDE*, 1997: p. 1269-1276.
9. K.R. Leonard, M.K.H., *Guided Wave Helical Ultrasonic Tomography of Pipes*. *Journal of Acoustical Society of America*, 2003. **767-774**(114): p. 2.
10. K.R. Leonard, M.K.H., *Lamb wave tomography of pipe-like structures*. *Ultrasonics*, 2005. **44**(7): p. 574-583.
11. K.R. Leonard, M.K.H., *Lamb Wave Helical Ultrasonic Tomography*. *Review of Progress in QNDE*, 2004. **23**: p. 173-180.
12. M.K. Hinders, K.R.L., *Lamb Wave Tomography of Pipes and Tanks using Frequency Compounding*. *Review of Progress in QNDE*, 2005. **24**: p. 867-874.
13. T. Hayashi, K.K., Z. Sun, J. Rose, *Guided Wave Focusing Mechanics in Pipe*. *Journal of Pressure Vessel Technology-Transactions of the Asme*, 2005. **127**: p. 317-321.
14. V. Schmitz, S.C.a.W.M., *Experiences with synthetic aperture focusing technique in the field*. *Ultrasonics*, 2000. **38**: p. 731-738.
15. R. Sicard, J.G.a.D.Z., *A SAFT algorithm for Lamb wave imaging of isotropic plate-like structures*. *Ultrasonics*, 2002. **39**: p. 487-494.

## Chapter VIII

### Conclusions

In this dissertation, we have presented two parallelized simulation techniques for three-dimensional acoustic and elastic wave propagation. We have demonstrated their usefulness in solving real-world problems with examples in the three very different areas of nondestructive evaluation, medical imaging, and security screening. More precisely, these include concealed weapons detection, periodontal ultrasography, and guided wave inspection of complex piping systems. In addition to this, we have also presented a novel experimental study of air-coupled nonlinear sound beam scattering from complex targets (Chapter 3) with very interesting and promising results.

The 3D parallel acoustic finite integration technique (3DPAFIT) can be used to study acoustic interactions with objects and layers in large and realistic geometries. We employed this technique along with a nonlinear finite-difference method to study nonlinear acoustic beams and their interaction with complex shaped objects. These objects included material layers, a human model, and an RPG model to assist in the development of nonlinear acoustic concealed weapons detector. We also performed a novel experiment to study air-coupled nonlinear sound beam scattering from objects. This study included the design of a robust signal processing technique to extract useful information about an object from backscattered acoustic energy. We used this experiment to validate the 3DPAFIT simulation technique.



The 3DPAFIT technique was also used to support the development of an ultrasonographic periodontal probe. A sophisticated software system was created to automatically define the intricate three-dimensional geometry of tissues in the periodontal region at the base to the tooth. Systematic simulations were performed to provide a large dataset to assist in the development of signal processing techniques to automatically determine the depth of the periodontal pocket from ultrasonic pulse-echo measurements.

We also presented a three-dimensional parallel cylindrical finite integration technique (3DPCFIT). This simulation method is ideal for modeling elastic waves in piping systems. In the derivation of the 3DPCFIT method, we introduce a coordinate transform to allow for the simulation of piping systems which include bends and twists. We validated this simulation technique by comparing simulation results directly to experimental measurements and to results from a commercial finite-element simulation package. This simulation method was employed to study guided elastic wave inspection of complex piping geometries and assisted in the development of both hardware configurations and signal processing algorithms.

### **8.1 Suggestions for Future Work**

Both simulation methods presented in this dissertation use a structured grid of Cartesian (3DPAFIT) or curved cylindrical (3DPCEFIT) grid cells. One possible improvement would be the introduction of a non-structured grid that is not confined to regular grid shapes. This could possibly improve the accuracy of the simulations when complex geometries are being modeled. One downfall to this adaptation will be the increases the computation resources needed to model complex acoustic and elastic wave

interactions. The volume and surface areas of the cells will vary over the computational space and thus require more memory to store these variables and require more computations to update the individual simulation values.

There are several directions that can be pursued to further develop the Nonlinear Acoustic Concealed Weapons Detector (NACWD). With the simulation code fully functional, the first objective may be to systematically explore acoustic wave interactions from people, clothing layers, and weapons. This would provide a large dataset of simulated measurements to further refine the signal processing algorithms to automatically identify concealed weapons. The pulse-compression technique described in chapter 3 may be able to identify subtle features in the frequency backscatter measurements to identify concealed weapons. This technique may also have applications in other fields of nondestructive evaluation and robotics.

For the Ultrasonic Periodontal Probe, the geometry of the periodontal region can be adapted to represent different tooth structures which vary from patient to patient and at each probing site. Additional anatomical features can be placed into the model such as cementum pearls, which form under the gum line and could possibly influence the ultrasound measurements. Further systematic simulations will account for a broad range of tooth and tissue geometries. This large data set could further enhance the signal processing algorithms responsible for the automatic determination of the depth of the periodontal pocket. Additional simulation studies can be performed to further optimize the design of the tip geometry and to study elastic waves in the hard tooth tissue for the detection of cavities and micro-cracks.

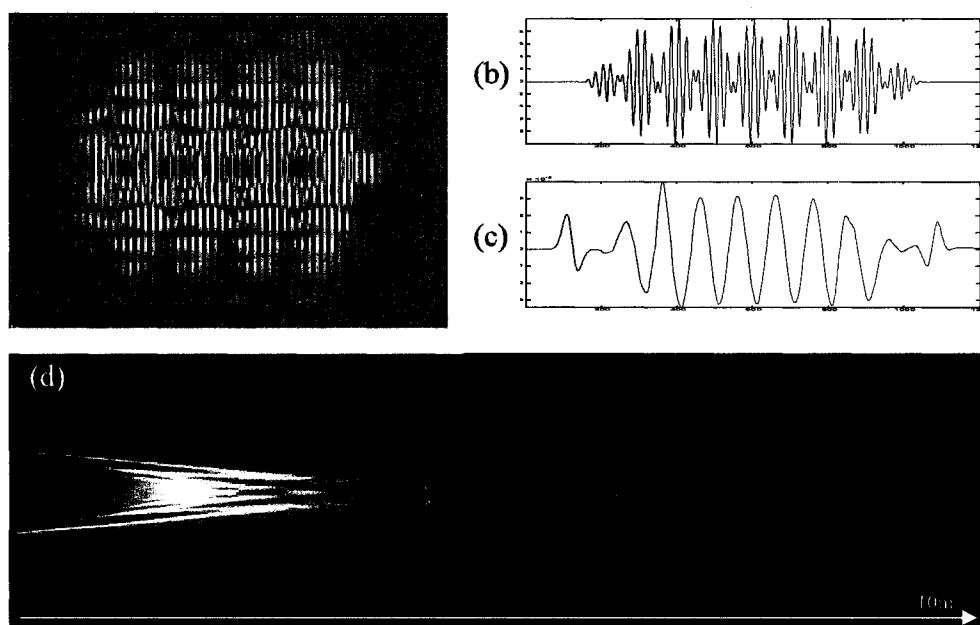
The 3DCEFIT simulations presented in this dissertation can model homogenous and isotropic materials such as solid pipes made of one material. In some situations, pipe systems can contain coatings or be composed of anisotropic materials such as composites. The material parameters in the simulations software can be discretized to allow for the simulation of piping systems composed of more complicated materials. This modification would allow the simulations of

In this dissertation, we have shown that parallel simulation methods for acoustic and elastic waves have a wide application range. We presented several applied examples where we used 3D parallel simulations to explore very difficult problems. There is potential to apply the simulation methods presented in this dissertation to new application areas. These application areas can include medical ultrasound, underwater acoustics, geoacoustics (seismic modeling), and additional areas of nondestructive evaluation.

## Appendix A1

### Determining Parameters of the Nonlinear KZK Simulation Code

A finite difference method used in Chapter 3 is presented here for simulating nonlinear acoustic beams. This code package was written at the College of William and Mary and is based on an existing and freely available algorithm and code package developed at the University of Texas Austin [1-3]. The code presented here includes several improvements over the Texas KZK code that improves computational efficiency and usability.



Example results from the KZK nonlinear computer simulations. Shown are a confocal pressure waveform snapshot (a), waveforms from a parametric source (initial waveform (b) and the resulting waveform after propagating over some distance (c)), and an energy distribution plot (d) for a 2ft focused transducer emitting a 50kHz pulse into air.

### A1.1. Overview of the KZK Simulation Algorithm

The Khokhlov-Zabolotskaya-Kuznetsov (KZK) equation is a nonlinear parabolic wave equation that accounts for the combined effects of diffraction, absorption, and nonlinearity in finite amplitude sound beams.

$$\frac{\partial^2 p}{\partial z \partial t'} = \frac{c_0}{2} \left( \frac{\partial^2 p}{\partial r^2} + \frac{1}{r} \frac{\partial p}{\partial r} \right) + \frac{D}{2c_0^3} \frac{\partial^3 p}{\partial t'^3} + \frac{\beta}{2\rho_0 c_0^3} \frac{\partial^2 p^2}{\partial t'^2} \quad (1)$$

Approximations are made in the term that account for diffraction because the sound is assumed to be confined to a narrow beam. This approximation introduces small errors at locations far from the beam axis (more than 20°) and close to the source (within several source radii of a piston).

Lee and Hamilton developed a finite difference technique to simulate nonlinear acoustic propagation for axial symmetric sources in the time domain (often referred to as the Texas KZK code) [1 - 3]. The core of the KZK simulation code written at the College of William and Mary is based entirely on Lee and Hamilton's algorithm.

The initial pressure field is specified by the user at the face of the transducer. Then using a finite difference method, the waveform propagates away from the source over small spatial steps. The algorithm first uses an Implicit Backward Finite Difference (IBFD) method near the source of the transducer to minimize numerical error. Then the algorithm switches to a Crank-Nicolson Finite Difference (CNFD) method which allows

for a larger spatial step size to improve computational efficiency. Details of the finite difference derivations can be found in [1 – 2].

As the pressure waves propagate away from the transducer, a waveform at any spatial location can be recorded. A series of individual waveforms will reveal the evolution of the waveform as it propagates away from the transducer. Additional processing techniques can be employed to quantitatively study and describe acoustic emissions from axial symmetric transducers and transducer arrays.

#### **A1.1.1. Improvements made the Texas KZK Code.**

The core of the KZK simulation code written at the College of William and Mary is based entirely on the Texas KZK Code. Several improvements were included in the new KZK simulation code. The new KZK code was completely re-written in the freely available Java programming language with an end goal of improving the computational efficiency of the algorithm and usability of the code.

The largest improvement was the addition of absorbing boundary conditions. With the Texas KZK code, the simulation space had to be made much larger than needed to avoid non-physical reflections from the boundaries interfering with the acoustic beam. By adding absorbing boundary conditions, the simulation space can be made at least four times smaller while achieving the same results (if not better) than with the Texas KZK code. This allows the simulations to run much faster and requires far less computer memory. Several other smaller improvements were made to improve the computational efficiency of the algorithm. For example, the same data structures are used in computing

the solutions in the absorption and diffraction steps. This lowers the memory requirements of the algorithm.

Another goal of rewriting the KZK simulation code was to improve usability. MATLAB scripts were created to set-up, execute, and display results from the KZK simulations from within the MATLAB environment. This allows simulation space parameters such as transducer dimensions and waveform frequency of the initial pressure waveform to be easily changed. A MATLAB script was also created to read in the results of the simulation directly into the MATLAB environment for further analysis.

## **A1.2. KZK Simulation Files. Installation and Execution**

All the files needed to execute, record, and view simulation results are included in the *wmkzk.zip* file. The contents of the *wmkzk.zip* file are described below.

<i>kzk.java</i>	- Java source code for the KZK simulation.
<i>kzk.class</i>	- Compiled KZK simulation code (Windows OS).
<i>runkzkparametric.m</i>	- Example MATLAB interface file to set-up and run a KZK simulation.
<i>ReadWaves.m</i>	- Example MATLAB file to read in the waveforms from a KZK simulation.
<i>KZKSimulationDoc_Ver1.doc</i>	- This document.

### **A1.2.1 Installation**

The contents of the *wmkzk.zip* should be placed into a new directory where simulations will be performed. The KZK simulations have been developed and tested on the Sun Microsystems's Java 1.4 platform. This can easily and freely be downloaded from Sun's website at <http://java.sun.com/j2se/1.4.2/ja/download.html>. This java

platform will need to be installed before continuing with the remaining of the installation process.

After java has been installed, the appropriate CLASSPATH environmental variable must be set. This variable will direct the java environment to the location of the KZK simulation code. This can be done in Windows XP by first clicking on the Start Menu and then the Control Panel icon. Once in the Control Panel, make sure you are in classic view by clicking the top left link which says “switch to classic view”. Now double click on the System icon, click the Advanced tab, and then the Environmental Variables button to open a new window. In this window, add a new system variable called CLASSPATH. Assign it a value of the path of the directory that holds the kzk.class file. Now exit this window and the control panel. Instructions for setting the CLASSPATH variable in other operating systems can be found on Sun’s Java Website (<http://java.sun.com>).

MATLAB will be required to use the interface files provided with the KZK simulation code. MATLAB Version 6.1 was used to execute and test the KZK simulation code. MATLAB is not required to execute the KZK simulations but is highly recommended for execution and analysis of the results.

#### **A1.2.2. Executing a KZK Simulation from MATLAB Example**

The file *runkzkparametric.m* is a MATLAB file that computes all the appropriate parameters, initial pressure field, creates an input file, and then executes the KZK simulation. It simulates a 2 ft diameter parametric array that emits a short burst that



contains two frequencies: 45 kHz and 55 kHz. The waveforms are recorded every 0.1 m from the face of the transducer to 10 m.

Once the `runkzkparametric` command is run from MATLAB, a file selection window will prompt the user where to save the input file to be passed to the KZK simulation. As the simulation runs, it will output the waveforms into an ASCII file named *waves.txt*. This file will be placed in the same directory that the input file was saved. The MATLAB function *ReadWaves.m* can then be used to import the waveforms into the MATLAB environment. The MATLAB command sequence to execute this example simulation and plot the waveform from the beam axis 5m from the transducer is shown below.

```
>> runkzkparametric
== Starting KZK simulation == Version 1.0

    number outputs in r-direction = 7.0
    number outputs in z-direction = 101.0
    number of t points = 1203

== Starting with IBFD Method == ds is now: 0.0010
current step: 50/675
current step: 100/675
== Switching to CNFD Method == ds is now: 0.0020
current step: 150/675
current step: 200/675
current step: 250/675
current step: 300/675
current step: 350/675
current step: 400/675
current step: 450/675
current step: 500/675
current step: 550/675
current step: 600/675
current step: 650/675
== Done with KZK simulation ==
>> w = ReadWaves('waves.txt');
>> plot(reshape(w(1,50,:),1,1203));
```

### **A1.3. Description of the KZK Simulation Parameters**

The KZK simulation code requires many simulation parameters to be specified that describe the transducer configuration, propagation medium, and details about the simulation space. These parameters are passed to the KZK simulation with an input file. The MATLAB file called *runkzkparametric.m* shows how to create this input file and execute a KZK simulation. This section will give the specifics for finding some of simulation parameters.

### A1.3.1. Unit-less Simulation Parameters

The KZK simulation requires three unit-less simulation parameters which describe the degree of nonlinearity, absorption, and diffraction used in the simulation.

#### A1.3.1.1 Nonlinearity Coefficient

The unit-less nonlinear coefficient is found by the following equation.

$$N = \frac{2\pi d \beta f_0 p_0}{\rho_0 c_0^3} \quad (2)$$

Where  $d$  is the geometrical focal distance of the transducer in meters,  $\beta$  is the coefficient of nonlinearity of propagation medium,  $f_0$  is the center frequency of the initial waveform in Hertz,  $p_0$  is the initial sound pressure amplitude in Pascals (Pa),  $\rho_0$  is the density of the propagation medium (kg/m<sup>3</sup>), and  $c_0$  is the small amplitude speed of sound in the propagation medium (m/s). For air, the coefficient of nonlinearity  $\beta$  is 1.2 [4], the density  $\rho_0$  is 1.15 kg/m<sup>3</sup>, and the speed of sound is 330 m/s<sup>2</sup>.

### A1.3.1.2 Absorption Coefficient

The unit-less Absorption coefficient is found by equation 3.

$$A=2\pi \alpha_0 d \quad (3)$$

Where  $d$  is the geometrical focal distance of the transducer and  $\alpha_0$  is the absorption parameter in units of nepers/meter. For air, the absorption parameter is dependent on the temperature of the air, the relative humidity of the air, and the frequency of the pressure wave. The absorption parameter of air  $\alpha_0$  and related quantities are given in [Ref 5-6] and can be found in equations 4-8.

$$\alpha_0 = \omega_0^2 \left\{ 1.84 \times 10^{-11} \left( \frac{T}{T_0} \right)^{1/2} + \left( \frac{T}{T_0} \right)^{-5/2} \left[ 0.01275 \frac{e^{-2239.1/T}}{F_{ro} + \omega_0^2 / F_{ro}} + 0.1068 \frac{e^{-3352/T}}{F_{rn} + \omega_0^2 / F_{rn}} \right] \right\} \quad (4)$$

Where  $\omega_0$  is the frequency of the wave,  $T_0$  is the reference atmospheric temperature of air (293.15 K),  $T$  is the temperature of the air in Kelvins, and  $F_{ro}$  and  $F_{rn}$  are the relaxation frequencies of oxygen and nitrogen, respectively. These relaxation frequencies are dependent on the air temperature  $T$  and the absolute value of humidity of the air  $h$ , and are given in equations 5 and 6.

$$F_{ro} = \left( 24 + 4.04 \times 10^4 h \frac{0.02 + h}{0.391 + h} \right) \quad (5)$$

$$F_{rn} = \left( \frac{T}{T_0} \right)^{1/2} \left( 9 + 280h \times \exp \left\{ -4.17 \left[ \left( \frac{T}{T_0} \right)^{1/3} - 1 \right] \right\} \right) \quad (6)$$

The absolute humidity  $h$  can be found from the relative humidity  $h_r$  by equation 7.

$$h = h_r p_{sat} \quad (7)$$

Where  $p_{sat}$  is the saturation vapor pressure which is given by equation 8.

$$p_{sat} = 10^{\left( -6.8346 \left( \frac{T_{01}}{T} \right)^{1.261} + 4.6151 \right)} \quad (8)$$

Where  $T_{01} = 273.16\text{K}$  is the triple-point isotherm temperature. These equations are valid when the ambient pressure level is 1 atm. Refer to [Ref 5] for finding absorption values when the ambient pressure level is not 1 atm.

#### A1.3.1.3 Diffraction/Gain Coefficient

The unit-less Diffraction/Gain coefficient is found by equation 9.

$$G = \frac{2\pi f_0 a^2}{2c_0 d} \quad (9)$$

Where  $f_0$  is the center frequency of the waveform,  $a$  is the radius of the transducer in meters,  $c_0$  is the small amplitude speed of sound in the propagation medium (m/s), and  $d$  is the geometrical focus distance of the transducer in meters.

### **A1.3.2. Other Simulation Parameters**

The remaining simulation parameters describe the simulation space, transducer configuration, initial pressure field, and output parameters. The *runkzkparametric.m* MATLAB file shows how to set these parameters and execute a KZK simulation.

Following is a brief description of the parameters of the KZK simulation using the same variable names as used in the *runkzksimulation.m* file.

#### **Simulation Control Parameters**

- isNon* – Boolean Variable to instruct the KZK simulations to include the effects of Nonlinearity. (1 = include, 0 = do not include)
- isDon* – Boolean Variable to instruct the KZK simulations to include the effects of Diffraction. (1 = include, 0 = do not include)
- isAon* – Boolean Variable to instruct the KZK simulations to include the effects of Absorption. (1 = include, 0 = do not include)
- isABCon* – Boolean Variable to instruct the KZK simulations to include the Absorbing Boundary Conditions. (1 = include, 0 = do not include)

#### **Unit less Simulation Parameters (see section 3.1)**

- N* – Unit-less nonlinear coefficient of the simulation.
- G* – Unit-less diffraction/gain coefficient of the simulation.
- A* – Unit-less absorption coefficient of the simulation.

#### **Transducer Parameters**

- radius* – Radius of transducer in meters.
- focald* – Transducer geometrical focal distance in meters.

#### **Initial Waveform Parameters**

- wo* – Center frequency of the simulation. The initial waveform is normalized to this center frequency.
- F1* – First frequency component of the initial waveform.
- F2* – Second frequency component of the initial waveform.

*taumin* – Minimum of the time range.  
*taumax* – Maximum of the time range.  
*numtaupercycle* – Number of time diversions per time cycle.  
*Zpadby* – Amount of zero padding to include on the edges of the initial waveform  
(in percent: 0 – 1)  
*tukeya* – Alpha of the Tukey window used to envelope the initial waveform  
(Range from 0 – 1).

#### **Simulation Space Parameters**

*maxr* – Maximum number of steps in the radial direction.  
*ntrans* – Number of radial points across the transducer.  
*IBFDds* – Sigma (z) step size of IB finite difference method.  
*CNFDds* – Sigma (z) step size of CN finite difference method.  
*rabc* – Thickness of the absorbing boundary layer in the radial direction.  
*tabc* – Thickness of the absorbing boundary layer in the time direction.

#### **Output Parameters (in meters)**

*outstartz* – Output waveforms starting at this z.  
*outbyz* – Output waveforms at every interval of this z step.  
*outendz* – Output waveforms ending at this z.  
*outstartr* – Output waveforms starting at this r.  
*outbyr* – Output waveforms at every interval of this r step.  
*outendr* – Output waveforms ending at this r.

### **A1.4. References**

- [1] Y. Lee, "Numerical Solution of the KZK Equation for Pulsed Finite Sound Beams in Thermoviscous Fluids." Ph.D. Dissertation. The University of Texas Austin, 1993.
- [2] Y. Lee and M.F. Hamilton, "Time-domain Modeling of pulsed finite amplitude sound beams," The Journal of the Acoustical Society of America 97(2), 906-917, 1995
- [3] Time-domain computer code developed at The University of Texas Austin – <http://people.bu.edu/robinc/kzk/>
- [4] M.F. Hamilton and D.T. Blackstock, "Nonlinear Acoustics," Academic Press, 1998
- [5] H.E. Bass, L.C. Sutherland, A.J. Zuckerwar, D.T. Blackstock, and D.M. Hester, "Atmospheric absorption of sound: Further developments," Journal of the Acoustic Society of America 97(1), 680-683, January 1995
- [6] H.E. Bass, L.C. Sutherland, A.J. Zuckerwar, D.T. Blackstock, and D.M. Hester, "Erratum: Atmospheric absorption of sound: Further developments," Journal of the Acoustic Society of America 99(2), 1259, February 1996

## Appendix A2

### Source Code for Simulation Techniques

#### A2.1 - 3D Parallel Acoustic Finite Integration Technique

##### A2.1.1 - Main Parallel Simulation Code

The following code initializes the simulation space and distributes the simulation parameters. It uses the Message Passing Interface (MPI) to communicate between the nodes.

```
#include <mpi.h>
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include "acousticrect.h"
#include "time.h"

using namespace std;

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

int maxt, outputevery, totalz, m2m3;
int rank, numworkers;

int whohasaline = 0;
int recordalineat = 2;

void master();
void inputnode();
void slave();
void DistributeSimulationParameters();
void dumpP(acousticrect &ar, int t);
void dumpTopPlate(int t);
void addArbReflector(acousticrect ar, double filename, double s1, double s2, int s3, double dd, double rc);

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numworkers); /* get number of nodes */

    numworkers = numworkers-2; //numworkers--; changed for inputnode

    if (rank == 0)
        master();
    else if (rank == numworkers+1)
        inputnode();
    else
        slave();

    MPI_Finalize();

    return 0;
}

// This Runs on Master Node
void master()
{
    MPI_Status status;
    time_t start,end;
    time_t(&start);

    cout << "master node is online! \n";

    DistributeSimulationParameters();

    double al=0;
    ofstream outFile("alineout.ascii", ios::out);

    for (int t=0; t<maxt; t++)
    {
        MPI_Recv(&al, 1, MPI_DOUBLE, whohasaline, 858, MPI_COMM_WORLD, &status);
        outFile << al << " ";

        if (t%outputevery == 0 )
        {
            dumpTopPlate(t);
            cout << "Collecting Slices at time: " << t << "\n";
        }
    }
}
```



```

outFile.close();
time (&end);
printf ("Total Run Time: %.2lf seconds\n", difftime (end,start) );
return;
}

// This runs on all the slave nodes
void slave()
{
    // -----
    // -- Receive Initial Data From Master
    MPI_Status status; MPI_Request request[2];
    double simparams[11];
    MPI_Recv(&simparams, 11, MPI_DOUBLE, 0, 201, MPI_COMM_WORLD, &status);

    acousticrect ar;
    ar.num1 = simparams[0]+2;
    ar.num2 = simparams[1];
    ar.num3 = simparams[2];
    ar.ds = simparams[3];
    ar.dt = simparams[4];
    ar.den = simparams[5];
    ar.cc = simparams[9];
    maxt = simparams[6];
    outputevery = simparams[7];
    ar.zbeg = simparams[8];
    ar.totalz = simparams[10];
    m2m3 = ar.num2*ar.num3;

    if (rank == 1) ar.type = 1;
    else if (rank == numworkers) ar.type = 3;
    else ar.type = 2;

    ar.Init();

    // -----
    // -- Receive Drive Function
    if (rank==1)
    {
        double *drive = new double[maxt];
        MPI_Recv(&drive[0], maxt, MPI_DOUBLE, 0, 202, MPI_COMM_WORLD, &status);
        ar.df = drive;
    }

    // -----
    // -- Receive Reflectors
    int nr; double *rpars = new double[8];
    MPI_Recv(&nr, maxt, MPI_INT, 0, 203, MPI_COMM_WORLD, &status);
    for (int i = 0; i < nr; i++)
    {
        MPI_Recv(&rpars[0], 8, MPI_DOUBLE, 0, 204, MPI_COMM_WORLD, &status);
        if (rpars[0] == 101)
            addArbReflector(ar, rpars[5], rpars[1], rpars[2], rpars[3], rpars[6], rpars[7]);
        else
            ar.addReflector(rpars[0], rpars[1], rpars[2], rpars[3], rpars[4], rpars[5], rpars[6], rpars[7]);
    }
    //cout << " 6";
    }

    // -----
    // -- Run Simulation
    //double *tosend = new double[m2m3*ar.num1];;
    double al;
    for (int t = 0; t < maxt; t++)
    {
        if (rank == 1) cout << " time: " << t << " " << ar.num1<< " " << ar.num2<< " " << ar.num3 << endl;

        if ((recordalineat >= ar.zbeg) && (recordalineat < (ar.zbeg+ar.num1-1)))
        {
            al = ar.pp.val(recordalineat-ar.zbeg,100,100);
            MPI_Isend(&al, 1, MPI_DOUBLE, 0, 858, MPI_COMM_WORLD, request);
        }

        if (t%outputevery == 0)
        {
            //tosend = ar.pp;
            int len = ar.pp.GetEvenVolLen(ar.zbeg);
            double *x = new double[len];
            x = ar.pp.GetEvenVol(ar.zbeg);

            MPI_Isend(&len, 1, MPI_INT, 0, 1101, MPI_COMM_WORLD, request);
            MPI_Isend(&x[0], len, MPI_DOUBLE, 0, 1102, MPI_COMM_WORLD, request);
        }

        ar.time = t;

        ar.UpdatePs(1,1);
        if (rank > 1) MPI_Isend(&ar.pp.a[m2m3], m2m3, MPI_DOUBLE, (rank-1), 301, MPI_COMM_WORLD, request);
        ar.UpdatePs(2,ar.num1-2);
        if (rank < numworkers) MPI_Recv(&ar.pp.a[(ar.num1-1)*m2m3], m2m3, MPI_DOUBLE, (rank+1), 301,
MPI_COMM_WORLD, &status);

        //if (rank>1) ar.doBackABCs(totalz);
    }
}

```

```

        // if rank == 1 receive input from input node
        if (rank == 1) MPI_Recv(&ar.pp.a[m2m3], m2m3, MPI_DOUBLE, (numworkers+1), 303, MPI_COMM_WORLD, &status);

        ar.UpdateVs(ar.num1-2, ar.num1-2);
        if (rank < numworkers) MPI_Isend(&ar.v1.a[(ar.num1-2)*m2m3], m2m3, MPI_DOUBLE, (rank+1), 302, MPI_COMM_WORLD,
request);
        ar.UpdateVs(1, ar.num1-3);
        if (rank > 1) MPI_Recv(&ar.v1.a[0], m2m3, MPI_DOUBLE, (rank-1), 302, MPI_COMM_WORLD, &status);

        ar.doDriveFunction();
    }
}

void DistributeSimulationParameters()
{
    char inputFilename[] = "in.file";
    ifstream inFile;
    inFile.open("in.file", ios::in);

    if (!inFile) {
        cerr << "Can't open input file " << inputFilename << endl;
        exit(1);
    }

    double *simpars = new double[11];

    inFile >> simpars[0]; //max1
    inFile >> simpars[1]; //max2
    inFile >> simpars[2]; //max3
    inFile >> simpars[3]; //ds
    inFile >> simpars[4]; //dt
    inFile >> simpars[5]; //default den
    inFile >> simpars[9]; //default speed of sound
    inFile >> simpars[6]; //maxt
    inFile >> simpars[7]; //outevery

    maxt = simpars[6];
    outevery = simpars[7];
    m2m3 = simpars[1]*simpars[2];
    totalz = simpars[0];
    simpars[10] = totalz;
    // send initial data to each node
    // -----
    int div, divaccum = 0;
    for (int n = 1; n <= numworkers; n++)
    {
        /* divide space along x1 direction */
        div = (totalz/(numworkers)); if ((n-1)<= (totalz%(numworkers))) div++;
        simpars[0] = div;
        simpars[8] = divaccum; // tells the worker where its starting z location is
        MPI_Send(&simpars[0], 1, MPI_DOUBLE, n, 201, MPI_COMM_WORLD);
        divaccum = divaccum+div;
        if ((whohasaline==0)&&(divaccum>=recordalineat)) whohasaline=n;
    }

    // send simpars to input node
    MPI_Send(&simpars[0], 10, MPI_DOUBLE, numworkers+1, 201, MPI_COMM_WORLD);

    cout << "whohasaline = " << whohasaline << "\n";
    // read in DF and send to worker number 1
    // -----
    double *drive = new double[maxt];
    for (int i = 0; i<maxt; i++)
        inFile >> drive[i];
    MPI_Send(&drive[0], maxt, MPI_DOUBLE, 1, 202, MPI_COMM_WORLD);

    // read in reflectors and distribute to all workers
    // -----
    int numref; inFile >> numref;
    double *rpars = new double[8];
    cout << " Number of reflectors: " << numref << endl;

    for (int n = 1; n <= numworkers; n++)
        MPI_Send(&numref, 1, MPI_INT, n, 203, MPI_COMM_WORLD);

    for (int i = 0; i < numref; i++)
    {
        inFile >> rpars[0]; // reflector type
        inFile >> rpars[1]; // reflector position in x1
        inFile >> rpars[2]; // reflector position in x2
        inFile >> rpars[3]; // reflector position in x3 - (start for cylinder)
        inFile >> rpars[4]; // reflector position in x3 - (end for cylinder)
        inFile >> rpars[5]; // reflector radius
        inFile >> rpars[6]; // reflector density
        inFile >> rpars[7]; // reflector speed of sound
        for (int n = 1; n <= numworkers; n++)
            MPI_Send(&rpars[0], 8, MPI_DOUBLE, n, 204, MPI_COMM_WORLD);
        cout << " " << rpars[6] << " " << rpars[7] << "\n";
    }

    inFile.close();
    return;
}

```

```

}

void dumpSlice(int t)
{
    MPI_Status status;
    double *topplate;
    int len;

    stringstream strm; strm << t;
    string fname = "toplate_at_t" + strm.str() + ".ascii";
    ofstream outFile(fname.c_str(), ios::out);

    for (int n = 1; n <= numworkers; n++)
    {
        MPI_Recv(&len, 1, MPI_INT, n, 1101, MPI_COMM_WORLD, &status);
        if (n==1) topplate = new double[len];

        MPI_Recv(&topplate[0], len, MPI_DOUBLE, n, 1102, MPI_COMM_WORLD, &status);

        for (int i = 0; i < len; i++)
            outFile << topplate[i] << " ";

    }

    delete topplate;
    outFile.close();
    return;
}

// dump 3D Pressure Values
void dumpP(acousticrect &ar, int t)
{
    stringstream strm; strm << t;
    string fname = "Pat" + strm.str() + ".ascii";
    ofstream outFile(fname.c_str(), ios::out);

    outFile << ar.num1-2 << " " << ar.num2 << " " << ar.num3 << " ";
    for (int i3=0; i3 < ar.num3; i3++)
        for (int i2=0; i2 < ar.num2; i2++)
            for (int i1=1; i1 < ar.num1-1; i1++)
                outFile << ar.pp.val(i1,i2,i3) << " ";

    outFile.close();
    return;
}

// Do the input!!
void inputnode()
{
    printf("InputNODE! ");

    int nr, nt, c, x2, x3, rf, rc;
    double r, dir, dit;

    MPI_Status status; MPI_Request request[2];
    double simparams[10];
    MPI_Recv(&simparams, 10, MPI_DOUBLE, 0, 201, MPI_COMM_WORLD, &status);

    acousticrect ar; // just used to hold simulation parameters
    ar.num1 = 1;
    ar.num2 = simparams[1];
    ar.num3 = simparams[2];
    ar.ds = simparams[3];
    ar.dt = simparams[4];
    ar.den = simparams[5];
    maxt = simparams[6];
    int m2m3 = ar.num2*ar.num3;

    //int m2m3 = max2*max3; // size of 2D p-array
    int c2 = (ar.num2/2); // find center value in x2 direction
    int c3 = (ar.num3/2); // find center value in x3 direction

    double *p = new double[m2m3]; // pressure matrix
    for (int ii = 0; ii<m2m3; ii++) p[ii]=0;

    // open file
    ifstream inFile;
    inFile.open("KZKwaveform.in", ios::in);
    if (!inFile) {
        cerr << "Can't open KZK input file " << endl;
        exit(1);
    }

    inFile >> nr; // read number of r steps
    inFile >> nt; // read number of t steps
    inFile >> dir; // read size of r step
    inFile >> dit; // read size of t step (should be same as simulation dt)

    double *pin = new double[nr]; // allocate array to hold p values along r at each time step
    for (int tt=1; tt<=maxt; ++tt)
    {
        if (tt < nt)

```

```

        for (int i=0; i < nr; ++i)
            inFile >> pin[i]; // read in p values for this time step

    for (x2 = 1; x2<ar.num2-1; ++x2)
        for (x3 = 1; x3<ar.num3-1; ++x3) // loop over 2d plane interpolating p-wave onto it.
            {
                r = fabs(sqrt( (double) ((x2-c2)*(x2-c2) + (x3-c3)*(x3-c3)) ))*(ar.ds/dir);

                rf = floor(r); // compute floor of r
                rc = ceil(r); // compute ceiling of r

                c=(x2*ar.num3)+x3;

                if (rc < nr)
                    p[c] = pin[rf]+(r-rf)*(pin[rc]-pin[rf]);
                // interpolate pressure wave onto cartesian grid
                //p[c] = pin[0];
            }

        MPI_Send(&p[0], (m2m3), MPI_DOUBLE, 1, 303, MPI_COMM_WORLD); //Send ps to node 2
    }

    inFile.close();

    return;
}

void addArbReflector(acousticrect ar, double filenameumber, double s1, double s2, int s3, double dd, double rc)
{
    //cout << "arb!";

    stringstream strm; strm << filenameumber;
    string fname = strm.str()+ ".ArbSimulationObject";

    ifstream inFile;
    inFile.open(fname.c_str(), ios::in);

    if (!inFile) {
        cerr << "Can't open input file " << fname.c_str() << endl;
        exit(1);
    }

    int n1, n2, n3, vv;

    inFile >> n1; // read number of steps in x1
    inFile >> n2; // read number of steps in x2
    inFile >> n3; // read number of steps in x3

    for (int i1 = 0; i1<n1; i1++)
        for (int i2 = 0; i2<n2; i2++)
            for (int i3 = 0; i3<n3; i3++)
                {
                    inFile >> vv; // read in volume value (1 or 0)

                    if ( (vv == 1) && ((s1+i1)>=ar.zbeg) && ((s1+i1)<(ar.num1+ar.zbeg-1)) )// && ((s2+i2)<(ar.num2)) &&
                        ((s3+i3)<(ar.num3)) )
                        {
                            if ((rc == -1) && (dd == -1))
                                {
                                    ar.B.set(s1+i1-ar.zbeg+1,s2+i2,s3+i3,2);
                                }
                            else
                                {
                                    ar.c.set(s1+i1-ar.zbeg+1,s2+i2,s3+i3,rc);
                                    ar.d.set(s1+i1-ar.zbeg+1,s2+i2,s3+i3,dd);
                                }
                        }
                }

    inFile.close();
}

```

### A2.1.2 – Acoustic Simulation Code (AcousticRect.h)

This code is where the actual simulation is performed. Each slave node gets a slice of the continous simulation space and the individual pressure and velocity values are updated [here](#).

```

#include <iostream>
#include "array3D.h"
#include "array3D_int.h"
// #include "transducer.h"

#define min(a,b) (((a)<(b))?(a):(b))
#define max(a,b) (((a)>(b))?(a):(b))

class acousticrect

```

```

{
public:
    acousticrect() { }
    ~acousticrect() { }

    int num1;        // number of grid points in r direction
    int num2;        // number of grid points in z direction
    int num3;        // number of grid points in p direction
    int totalz;      //

    int abc;         // number of abc points on each end

    double ds;       // spatial step size in r and z direction (meters)
    double dt;       // time step size (seconds)

    double den;      // density (kg/m^3)
    double cc;       // default speed of sound

    int zbeg;        // z start position (meters)

    int type;        // type -> 1 = left end , 2 = middle, 3 = right end

    array3D v1;      // 1 - velocities
    array3D v2;      // 2 - velocities
    array3D v3;      // 3 - velocities
    array3D pp;      // pressures
    array3D c;       // speed of sound
    array3D d;       // density

    array3D_int B;   // Boundary Array

    int time;

    double *df;

    //transducer *trans;
    //int numtrans;

    //int numreflectors;
    //int *rftype;    // reflector type 0 = sphere, 1 = cylinder
    //double *rx1;
    //double *rx2;
    //int *rx3start;
    //int *rx3end;
    //double *rrad;

private:

    double dtods;
    //double dtods;

    int i1,i2,i3;

public:

    void Init()
    {
        v1.Init(num1,num2,num3);
        v2.Init(num1,num2,num3);
        v3.Init(num1,num2,num3);
        pp.Init(num1,num2,num3);
        c.Init(num1,num2,num3,cc);
        d.Init(num1,num2,num3,den);
        B.Init(num1,num2,num3, type);

        dtods = dt/ds;

        time = 0;
        abc = 40;
    }

    void UpdatePs(int zs, int zend)
    {
        //for (i1 = 1; i1 < num1-1; i1++) // changed num1 -> num1-1
        for (i1 = zs; i1 <= zend; i1++) // changed num1 -> num1-1
            for (i2 = 1; i2 < num2-1; i2++)
                {
                    pp.setindx(i1,i2,1); v1.setindx(i1,i2,1); v2.setindx(i1,i2,1);
                    v3.setindx(i1,i2,1); d.setindx(i1,i2,1); c.setindx(i1,i2,1);
                    for (i3 = 1; i3 < num3-1; i3++)
                        {
                            pp.sv( pp.v()-dtods*d.v()*c.v()*c.v()*((v1.v()-v1.v1m())+(v2.v()-v2.v2m())+(v3.v()-v3.v3m())) );
                            pp.incindx(); v1.incindx(); v2.incindx(); v3.incindx(); d.incindx(); c.incindx();
                        }
                }

        //plane - bcs

        //for (i1 = 1; i1 < num1; i1++)
        for (i1 = zs; i1 <= zend; i1++)
            {

```

```

        pp.set(i1,0,0, pp.val(i1,0,0)-dtods*d.val(i1,0,0)*c.val(i1,0,0)*c.val(i1,0,0)*(v1.val(i1,0,0)-v1.val(i1-1,0,0)));
        pp.set(i1,0,num3-1, pp.val(i1,0,num3-1)-dtods*d.val(i1,0,num3-1)*c.val(i1,0,num3-1)*c.val(i1,0,num3-1)*(v1.val(i1,0,num3-1)-v1.val(i1-1,0,num3-1)));
        pp.set(i1,num2-1,num3-1, pp.val(i1,num2-1,num3-1)-dtods*d.val(i1,num2-1,num3-1)*c.val(i1,num2-1,num3-1)*(v1.val(i1,num2-1,num3-1)-v1.val(i1-1,num2-1,num3-1)));
        pp.set(i1,num2-1,0, pp.val(i1,num2-1,0)-dtods*d.val(i1,num2-1,0)*c.val(i1,num2-1,0)*c.val(i1,num2-1,0)*(v1.val(i1,num2-1,0)-v1.val(i1-1,num2-1,0)));

        for (i2 = 1; i2 < num2-1; i2++)
        {
            pp.set(i1,i2,0, pp.val(i1,i2,0)-dtods*d.val(i1,i2,0)*c.val(i1,i2,0)*c.val(i1,i2,0)*(v1.val(i1,i2,0)-v1.val(i1-1,i2,0))+v2.val(i1,i2,0)-v2.val(i1,i2-1,0)));
            pp.set(i1,i2,num3-1, pp.val(i1,i2,num3-1)-dtods*d.val(i1,i2,num3-1)*c.val(i1,i2,num3-1)*c.val(i1,i2,num3-1)*(v1.val(i1,i2,num3-1)-v1.val(i1-1,i2,num3-1))+v2.val(i1,i2,num3-1)-v2.val(i1,i2-1,num3-1)));
        }
        for (i3 = 1; i3 < num3-1; i3++)
        {
            pp.set(i1,0,i3, pp.val(i1,0,i3)-dtods*d.val(i1,0,i3)*c.val(i1,0,i3)*c.val(i1,0,i3)*(v1.val(i1,0,i3)-v1.val(i1-1,0,i3))+v3.val(i1,0,i3)-v3.val(i1,0,i3-1)));
            pp.set(i1,num2-1,i3, pp.val(i1,num2-1,i3)-dtods*d.val(i1,num2-1,i3)*c.val(i1,num2-1,i3)*c.val(i1,num2-1,i3)*(v1.val(i1,num2-1,i3)-v1.val(i1-1,num2-1,i3))+v3.val(i1,num2-1,i3)-v3.val(i1,num2-1,i3-1)));
        }
    }

    void UpdateVs(int zs, int zend)
    {
        //for (i1 = 1; i1 < num1-2; i1++)
        for (i1 = zs; i1 <= zend; i1++)
        for (i2 = 0; i2 < num2-1; i2++)
        {
            pp.setindx(i1,i2,0); v1.setindx(i1,i2,0); v2.setindx(i1,i2,0); v3.setindx(i1,i2,0); d.setindx(i1,i2,1);
            for (i3 = 0; i3 < num3-1; i3++)
            {
                v1.sv( v1.v() - 2*dtods/(d.v()+d.v1p())*(pp.v1p()-pp.v()) );
                v2.sv( v2.v() - 2*dtods/(d.v()+d.v2p())*(pp.v2p()-pp.v()) );
                v3.sv( v3.v() - 2*dtods/(d.v()+d.v3p())*(pp.v3p()-pp.v()) );
                pp.incindx(); v1.incindx(); v2.incindx(); v3.incindx(); d.incindx();
            }
        }

        //for (i1 = 1; i1 < num1-2; i1++)
        for (i1 = zs; i1 <= zend; i1++)
        {
            for (i2 = 0; i2 < num2; i2++) // changed num2-1 -> num2
            {
                v1.set(i1,i2,num3-1, v1.val(i1,i2,num3-1) - 2*dtods/(d.val(i1+1,i2,num3-1)+d.val(i1,i2,num3-1))*(pp.val(i1+1,i2,num3-1)-pp.val(i1,i2,num3-1)));
                for (i3 = 0; i3 < num3; i3++)
                {
                    v1.set(i1,num2-1,i3, v1.val(i1,num2-1,i3) - 2*dtods/(d.val(i1+1,num2-1,i3)+d.val(i1,num2-1,i3))*(pp.val(i1+1,num2-1,i3)-pp.val(i1,num2-1,i3)));
                }
            }

            // Rigid Reflectors!

            //for (i1 = 1; i1 < num1-2; i1++)
            for (i1 = zs; i1 <= zend; i1++)
            for (i2 = 0; i2 < num2-1; i2++)
            {
                B.setindx(i1,i2,0); v1.setindx(i1,i2,0); v2.setindx(i1,i2,0); v3.setindx(i1,i2,0);
                for (i3 = 0; i3 < num3-1; i3++)
                {
                    if (B.v() == 2)
                    {
                        if (B.v1p() == 2) v1.sv(0);
                        if (B.v2p() == 2) v2.sv(0);
                        if (B.v3p() == 2) v3.sv(0);
                    }

                    B.incindx(); v1.incindx(); v2.incindx(); v3.incindx();
                }
            }
            doABCs();
            doBackABCs(totalz);
        }

        void doABCs()
        {
            int aabc = 25;
            double per;

            for (i1 = 2; i1 < num1-2; i1++)
            for (i2 = 0; i2 < num2; i2++)
            for (i3 = 1; i3 < aabc; i3++)
            {
                per = (1-.002*(aabc-i3));
                v1.setindx(i1,i2,i3); v2.setindx(i1,i2,i3); v3.setindx(i1,i2,i3);
                v1.sv(v1.v()*per);v2.sv(v2.v()*per);v3.sv(v3.v()*per);

                v1.setindx(i1,i2,num3-i3-1); v2.setindx(i1,i2,num3-i3-1);
                v3.setindx(i1,i2,num3-i3-1);
                v1.sv(v1.v()*per);v2.sv(v2.v()*per);v3.sv(v3.v()*per);
            }
        }
    }

```

```

        for (i1 = 2; i1 < num1-2; i1++)
            for (i2 = 1; i2 < aabc; i2++)
                for (i3 = aabc; i3 < num3-aabc; i3++)
                {
                    per = (1-.002*(aabc-i2));
                    v1.setindx(i1,i2,i3); v2.setindx(i1,i2,i3); v3.setindx(i1,i2,i3);
                    v1.sv(v1.v()*per);v2.sv(v2.v()*per);v3.sv(v3.v()*per);

                    v1.setindx(i1,num2-i2-1,i3); v1.setindx(i1,num2-i2-1,i3); v1.setindx(i1,num2-
i2-1,i3);
                    v1.sv(v1.v()*per);v2.sv(v2.v()*per);v3.sv(v3.v()*per);
                }
    }

void doBackABCs(int TotalZ) //ABC on the backside of the space (max1)
{
    int aabc = 25;
    double per;

    for (i1 = max(TotalZ-aabc-1,zbeg); (i1 >= zbeg) & (i1 < (zbeg+num1-1)); i1++)
    {
        v1.setindx(i1-zbeg,0,0); v2.setindx(i1-zbeg,0,0); v3.setindx(i1-zbeg,0,0);
        per = (1+.002*(-i1+(TotalZ-aabc-1)));
        //std::cout << " " << i1 << " " << per << '\n';
        //if (pipetype==3) std::cout << i1 << " " << i1-zbeg << " " << per << '\n';
        for (i2 = 1; i2 < num2; i2++)
            for (i3 = 1; i3 < num3; i3++)
            {
                v1.setindx(i1-zbeg,i2,i3); v2.setindx(i1-zbeg,i2,i3); v3.setindx(i1-zbeg,i2,i3);
                v1.sv(v1.v()*per);v2.sv(v2.v()*per);v3.sv(v3.v()*per);
                //v1.incindx(); v2.incindx(); v3.incindx();
            }
    }

void doDriveFunction()
{
    if (type == 1)
    {
        v1.setindx(0,0,0);
        for (i2 = 0; i2 < num2; i2++)
            for (i3 = 0; i3 < num3; i3++)
            {
                v1.sv( v1.v() -
2*dtods/(d.val(1,i2,i3)+d.val(0,i2,i3))* (pp.val(1,i2,i3)-pp.val(0,i2,i3)+df[time]) );
                v1.incindx();
            }
    }

void addReflector(double typ, double p1, double p2, int start3, int end3, double rad, double dd, double rc)
{
    if (typ == 0) //sphere
    {
        for (i1 = 0; i1 < num1; i1++)
            for (i2 = 0; i2 < num2; i2++)
                for (i3 = 0; i3 < num3; i3++)
                    if (((i1+zbeg-1-p1)*(i1+zbeg-1-p1) + (i2-p2)*(i2-p2) + (i3-start3)*(i3-
start3)) < rad*rad)
                    {
                        if ((rc == -1) && (dd == -1))
                        {
                            B.set(i1,i2,i3,2);
                        }
                        else
                        {
                            c.set(i1,i2,i3,rc);
                            d.set(i1,i2,i3,dd);
                        }
                    }
    }
    else if (typ == 1) //cylinder
    {
        for (i1 = 0; i1 < num1; i1++)
            for (i2 = 0; i2 < num2; i2++)
                if (((i1+zbeg-1-p1)*(i1+zbeg-1-p1) + (i2-p2)*(i2-p2)) < rad*rad)
                    for (i3 = start3; i3 <= end3; i3++)
                        if ((rc == -1) && (dd == -1))
                        {
                            B.set(i1,i2,i3,2);
                        }
                        else
                        {
                            c.set(i1,i2,i3,rc);
                            d.set(i1,i2,i3,dd);
                        }
    }
    else if (typ == 2) //rectangle
    {
        for (i1 = 0; i1 < num1; i1++)

```

```

        for (i2 = 0; i2 < num2; i2++)
            for (i3 = 0; i3 < num3; i3++)
                if ((i1+zbeg-1 >= start3) && (i1+zbeg-1 <= end3))
                    if ((rc == -1) && (dd == -1))
                    {
                        B.set(i1,i2,i3,2);
                    }
                    else
                    {
                        c.set(i1,i2,i3,rc);
                        d.set(i1,i2,i3,dd);
                    }
            }
    }
};

```

## A2.1.3 - 3D Array

Object that mimics 3D array of doubles with fast access methods.

```

#include <iostream>

class array3D
{
private:
    int    ci;    // current index
    int    ci3;   // current i3 index (used for continous Boundary)
    int    L2L3;  // max2*max3

public:
    double *a;
    int    len1;   // number of grid points in r direction
    int    len2;   // number of grid points in z direction
    int    len3;   // number of grid points in p direction

    // =====
    // Blank Constructor
    array3D() {}

    // =====
    // Blank Deconstructor
    ~array3D() {}

    // =====
    // Init - defines the array and its dimensions - MUST BE CALLED BEFORE USING
    void Init(int m1, int m2, int m3)
    { Init(m1,m2,m3,0); }
    void Init(int m1, int m2, int m3, double def)
    {
        len1 = m1; len2 = m2; len3 = m3;
        L2L3 = m2*m3;
        a = new double[m1*m2*m3];
        clear(def);
        return;
    }

    // =====
    // Return value at i1, i2, i3
    double val(int i1, int i2, int i3)
    {
        return a[(i1*L2L3)+(i2*len3)+i3];
    }

    // =====
    // Set value at i1, i2, i3
    void set(int i1, int i2, int i3, double val)
    {
        a[(i1*L2L3)+(i2*len3)+i3] = val;
        return;
    }

    // =====
    // quick access methods
    void setindx(int i1, int i2, int i3) { ci = (i1*L2L3)+(i2*len3)+i3; ci3 = i3; }
    void incindx()
    {
        ci = ci+1; ci3 = ci3+1;
        if (ci3==len3) ci3=0;
    }
    void sv(double x)
    {
        double v() { return a[ci]; } // equiv of a[i1][i2][i3]
        double vlp() { return a[ci+L2L3]; } // equiv of a[i1-1][i2][i3]
        a[ci] = x; // set value at ci
    }
};

```



```

double v1m() { return a[ci-L2L3]; } // equiv of a[i1+1][i2][i3]
double v2p() { return a[ci+len3]; } // equiv of a[i1][i2+1][i3]
double v2p2() { return a[ci+2*len3]; } // equiv of a[i1][i2+2][i3]
double v2m() { return a[ci-len3]; } // equiv of a[i1][i2-1][i3]
double v3p() { // return a[ci+1]; } // equiv of a[i1][i2][i3+1]
               if (ci3 == len3-1)
                   return a[ci-len3+1];
               else
                   return a[ci+1];
}
double v3m() { //return a[ci-1]; } // equiv of a[i1][i2][i3-1]
               if (ci3 == 0)
                   return a[ci+len3-1];
               else
                   return a[ci-1];
}

// =====
// clear - sets all values = 0;
void clear(double def)
{
    for (int i = 0; i < L2L3*len1; i++)
        a[i] = def;
}

// =====
// returns 2D slice through 3D array at fixed index 2
double* slice_fix2(int i2)
{
    double *x = new double[(len1-2)*len3];
    x[0] = (len1-2)*len3;

    int c = 0;
    for (int i1 = 1; i1 < len1-1; i1++) // does not return ends
        for (int i3 = 0; i3 < len3; i3++)
        {
            x[c] = val(i1, i2, i3);
            c++;
        }

    return x;
}
int slice_fix2_count() { return (len1-2)*len3; }

// =====
// returns 3D volume returning only the even indexes
double* GetEvenVol(int start)
{
    int len = GetEvenVolLen(start);
    double *x = new double[len];

    int c = 0;
    for (int i1 = 1+(start%2); i1 < len1-1; i1=i1+2) // does not return ends
        for (int i2 = 0; i2 < len2; i2=i2+2)
            for (int i3 = 0; i3 < len2; i3=i3+2)
            {
                x[c] = (val(i1-1, i2, i3) + val(i1, i2, i3))/2;
                c++;
            }

    //std::cout << len << " " << c-1 << "\n";

    return x;
}

int GetEvenVolLen(int start)
{
    int len;
    if (start%2 == 0)
        len = (len1-1)/2*(len2/2)*(len3/2);
    else
        len = (len1-2)/2*(len2/2)*(len3/2);
    return len;
}
};

```

### A2.1.3 - 3D Array of integers

Object that mimics 3D array of Integers with fast access methods.

```

class array3D_int
{
private:
    int    *a;
    int    ci; // current index
    int    L2L3; // max2*max3

```

```

    int    endtype;

public:

    int len1;          // number of grid points in z direction
    int len2;          // number of grid points in r direction
    int len3;          // number of grid points in p direction

    // Blank Constructor
    array3D_int() {}

    // Deconstructor
    ~array3D_int() {}

    // Init - defines the array and its dimensions - MUST BE CALLED BEFORE USING
    void Init(int m1, int m2, int m3, int type)
    {
        len1 = m1; len2 = m2; len3 = m3;
        L2L3 = m2*m3;
        a = new int[m1*m2*m3];
        endtype = type;
        clear();
        return;
    }

    // Return value at i1, i2, i3
    int val(int i1, int i2, int i3)
    {
        return a[(i1*L2L3)+(i2*len3)+i3];
    }

    // Set value at i1, i2, i3
    void set(int i1, int i2, int i3, int val)
    {
        a[(i1*L2L3)+(i2*len3)+i3] = val;
        return;
    }

    // quick access methods
    void setindx(int i1, int i2, int i3) { ci = (i1*L2L3)+(i2*len3)+i3; }
    void incindx() { ci = ci+1; }
    int v() { return a[ci]; } // equiv of a[i1][i2][i3]
    int v1p() { return a[ci+L2L3]; } // equiv of a[i1+1][i2][i3]
    int v1m() { return a[ci-L2L3]; } // equiv of a[i1-1][i2][i3]
    int v2p() { return a[ci+len3]; } // equiv of a[i1][i2+1][i3]
    int v2m() { return a[ci-len3]; } // equiv of a[i1][i2-1][i3]
    int v3p() { return a[ci+1]; } // equiv of a[i1][i2][i3+1]
    int v3m() { return a[ci-1]; } // equiv of a[i1][i2][i3-1]

    // clear - sets all values = 0;
    void clear()
    {
        //std::cout << "type " << endtype << "\n";
        for (int i = 0; i < L2L3*len1; i++)
            a[i] = 0;
    }
};

```

## A2.2 KZK Nonlinear Sound Beam Simulations

### A2.2.1 Java KZK source code (*kzk.java*)

```
// =====
// kzk.java - March 2005                               Kevin Rudd -- kerudd@wm.edu
//
// This code solves the KZK equation for focused axial symmetric sources. The core algorithm
// was developed at the University of Texas at Austin. It includes the effects of nonlinearity,
// absorption, and diffraction. More details of the code can be found in the following references.
//
// [1] Yang Sub-Lee, "Numerical solution of the KZK equation for pulsed finite-
// amplitude sound beams in thermoviscous fluids", Ph D dissertation,
// The University of Texas at Austin, December 1993.
//
// [2] Yang Sub-Lee and Mark F. Hamilton, "Time-domain modeling of finite-
// amplitude beams", J. Acoust. Soc. Am. 97, 906-917 (1995).
//
// This java version was written to be easily interfaced with MATLAB. It also includes absorbing
// boundary conditions which allow the simulations to run much quicker because the simulation
// space can be reduced. Any questions and comments about this code can be
// directed to ...
//
// Kevin Rudd
// The Nondestructive Evaluation Laboratory
// The Applied Science Department
// The College of William and Mary
// kerudd@wm.edu
// =====

import java.util.*;
import java.io.*;
import java.lang.Math.*;

public class kzk
{
    public static double G = 0;           // Gain/Diffraction (to be read in)
    public static double A = 0;           // Absorption (to be read in)
    public static double N = 0;           // Nonlinearity (to be read in)

    public static int isNon = 0;           // is Nonlinearity on? (to be read in)
    public static int isAon = 0;           // is Absorption on? (to be read in)
    public static int isDon = 0;           // is Diffraction on? (to be read in)
    public static int isABCon = 0;         // is Absorbing Boundary on? (to be read in)

    public static int maxr = 0;           // number of points in r-direction (to be read in)
    public static int IBFDzsteps = 100;   // number of steps in z-direction for IBFD
    public static int totalzsteps = 0;     // total number of steps in z-direction (to be read in)

    public static int maxt = 0;           // number of time steps (to be read in)
    public static int rabc = 50;          // thickness of r absorbing condition (to be read in)
    public static int tabc = 50;          // thickness of t ac (both ends) (to be read in)

    public static double dt = 0;          // time step size (to be read in)
    public static double tstart = 0;      // start time (to be read in)
    public static double dr = 0;          // radial step size (to be read in)
    public static double ds = 0;          // axial (z) step size (to be read in)
    public static double IBFDds = 0;      // IBFD ds (to be read in)
    public static double CNFDds = 0;      // CNFD ds (to be read in)

    public static int t = 1;              // integer time step
    public static double sigma = 0;       // current sigma

    public static double[][] p;           // pressure matrix

    public static double[][] LHSDiff;     // LHS of the tridiagonal system for Diffraction
    public static double[][] LHSAbsor;    // LHS of the tridiagonal system for Absorption

    public static int ors = 0; public static int orb = 0; // which waveforms to output
    public static int ore = 0; public static double oze = 0; // s = start, b = skip by
    public static double ozb = 0; public static double oze = 0; // e = end (to be read in)

    public static String workdir = "";     // working directory

    //-----
    //--- temp vars - I know this is bad programming, but it speeds up computations ---
    public static int i; public static int ii; public static int j; public static int jj;
    public static double[] sumP;
    public static double[] rhs;
    public static double[] beta;
    public static double[] gamma;
    public static double[] sol;
    public static double R; public static double Ro2; public static double Ro4; //R for Diff
    public static double Ro8; public static double S; public static double So2; //S for Absorb
    public static double dDisto; public static double dDeltaPmax; //for Nonlinear
    public static double dDeltaPdt; public static int k;
    public static double[] TauDisto; public static double[] pold;
    //-----

    public static void main(String[] args) throws IOException
    {
        initVars(args);           // Initialize Variables from input file
        ds = IBFDds;              // Set the z-step size to the CNFD ds
        initLHS_IBFD();           // Initialize LHS matrices for the IBFD method

        // Set up outputfile
        BufferedWriter outfile=new BufferedWriter (new FileWriter(workdir + "waves.txt"));

        System.out.println("== Starting KZK simulation == Version 1.0");
        System.out.println("");
        System.out.println("number outputs in r-direction = " + (Math.floor((ore-ors)/orb)+1) + " ");
        System.out.println("number outputs in z-direction = " + (Math.floor((oze-ozs)/ozb)+1) + " ");
        System.out.println("number of t points = " + (maxt+1) + " ");
        System.out.println("");
    }
}
```

```

OutputWaveforms(outfile); // Output initial waveform?

System.out.println("== Starting with IBFD Method == ds is now: " + IBFDds );
for (t=1;t<=IBFDzsteps;t++)
{
    sigma = t*ds; // set current sigma

    if (isDon==1) IBFDDiffraction(); // do the IBFD Diffraction Step
    if (isAon==1) IBFDAbsorption(); // do the IBFD Absorption Step
    if (isNon==1) Nonlinear(); // do the Nonlinear Step
    if (isABCon==1) DoABCs(); // do the Absorbing Boundaries Step
    OutputWaveforms(outfile); // Output waveform?

    if (t%(50) == 0) System.out.println(" current step: "+t+"/"+totalzsteps);
}

System.out.println("== Switching to CNFD Method == ds is now: " + CNFDds );
ds = CNFDds; // Set the z-step size to the CNFD ds
initLHS_CNFD(); // Initialize LHS matrices for the CNFD method

for (t=t;t<=totalzsteps;t++)
{
    sigma =(t-IBFDzsteps)*CNFDds+(IBFDzsteps)*IBFDds; // set current sigma

    if (isDon==1) CNFDDiffraction(); // do the CNFD Diffraction Step
    if (isAon==1) CNFDAbsorption(); // do the CNFD Absorption Step
    if (isNon==1) Nonlinear(); // do the Nonlinear Step
    if (isABCon==1) DoABCs(); // do the Absorbing Boundaries Step
    OutputWaveforms(outfile); // Output waveform?

    if (t%(50) == 0) System.out.println(" current step: "+t+"/"+totalzsteps);
}

outfile.close();

System.out.println("== Done with KZK simulation ==");
}

// -----
// IBFDDiffraction - Implicit Backward Finite Difference Method for the diffraction term in the KZK
// -----
public static void IBFDDiffraction()
{
    for (j=1; j<=maxr-1; j++)
    {
        sumP[j] = 0; sol[j] = 0;
    }

    for (i=1; i<=maxt-1; i++)
    {
        for (j=0; j<=maxr-1; j++)
            sumP[j] = sumP[j] + p[i-1][j];
        rhs[0] = p[i][0] + R*(sumP[1]-sumP[0]);
        for (j=1; j<=maxr-2; j++)
            rhs[j] = p[i][j]+(1-1/(2*j))*(Ro4)*sumP[j-1]-(Ro2)*sumP[j]+(1+1/(2*j))*(Ro4)*sumP[j+1];
        rhs[maxr-1] = p[i][maxr-1] + (1-1/(2*(maxr-1)))*(Ro4)*sumP[maxr-2]-(Ro2)*sumP[maxr-1];

        tridiagDiff(LHSDiff, rhs, 0, maxr-1);
        for (j=0; j<=maxr-1; j++)
            p[i][j] = sol[j];
    }
}

// -----
// CNFDDiffraction - Crank-Nicolson Finite Difference Method for the diffraction term in the KZK
// -----
public static void CNFDDiffraction()
{
    for (j=0; j<=maxr-1; j++)
    {
        sumP[j] = 0; sol[j] = 0;
    }

    for (i=1; i<=maxt-1; i++)
    {
        for (j=0; j<=maxr-1; j++)
            sumP[j] = sumP[j] + sol[j];
        rhs[0] = 2*p[i][0] + Ro2*(sumP[1]-sumP[0]);
        for (j=1; j<=maxr-2; j++)
            rhs[j] = 2*p[i][j]+(1-1/(2*j))*(Ro8)*sumP[j-1]-(Ro4)*sumP[j]+(1+1/(2*j))*(Ro8)*sumP[j+1];
        rhs[maxr-1] = 2*p[i][maxr-1] + (1-1/(2*(maxr-1)))*(Ro8)*sumP[maxr-2]-(Ro4)*sumP[maxr-1];

        tridiagDiff(LHSDiff, rhs, 0, maxr-1);
        for (j=0; j<=maxr-1; j++)
            p[i][j] = sol[j]-p[i][j];
    }
}

// -----
// IBFDAbsorption - Implicit Backward Finite Difference Method for the absorption term in the KZK
// -----
public static void IBFDAbsorption()
{
    for (j=0; j<=maxr-1; j++)
    {
        for (i=1; i<=maxt-1; i++)
        {
            rhs[i]=p[i][j];
            tridiagDiff(LHSAbso, rhs, 1, maxt-1);
            for (i=1; i<=maxt-1; i++)
                p[i][j]=sol[i];
        }
    }
}

// -----
// CNFDAbsorption - Crank-Nicolson Finite Difference Method for the absorption term in the KZK
// -----
public static void CNFDAbsorption()
{
}

```

```

        for (j=0; j<maxr-1; j++)
        {
            for (i=1; i<maxt-1; i++)
            {
                rhs[i]=(So2)*p[i-1][j]+(1-S)*p[i][j]+(So2)*p[i+1][j];
                tridiagDiff(LHSAbso, rhs, 1, maxt-1);
                for (i=1; i<maxt-1; i++)
                {
                    p[i][j]=sol[i];
                }
            }
        }

// -----
// Nonlinear - nonlinear term in the KZK
// -----
public static void Nonlinear() throws IOException
{
    for (j=0; j<maxr; j++)
    {
        pold[0] = 0; pold[maxt] = 0;
        for (i=1; i<maxt-1; i++)
        {
            if (p[i][j] >= 0)
            {
                pold[i] = p[i][j]/(1-N*(p[i+1][j]-p[i][j])*ds);
            }
            else
            {
                pold[i] = p[i][j]/(1-N*(p[i][j]-p[i-1][j])*ds);
            }
        }
        for (i=0; i<maxt; i++)
        {
            p[i][j] = pold[i];
        }
    }
}

// -----
// DoABCs - Attenuates the pressure values at the boundaries to reduce reflections
// -----
public static void DoABCs()
{
    for (i = 1; i <= rabc; i++)
    {
        for (j = 1; j <= maxt; j++)
        {
            p[j][maxr-rabc+i] = (1-.005*i)*p[j][maxr-rabc+i];
        }
        for (i = 1; i <= maxr; i++)
        {
            for (j = 1; j <= tabc; j++)
            {
                p[maxt-tabc+j][i] = (1-.005*j)*p[maxt-tabc][i];
                p[tabc-j][i] = (1-.005*j)*p[tabc][i];
            }
        }
    }
}

// -----
// tridiagDiff - Solves the tridiagonal system for the <LHS> coefficients for a given <rhs>.
// The solution is left in variable sol. This is the Thomas algorithm.
// -----
public static void tridiagDiff(double[][] LHS, double[] rhs, int start, int end)
{
    beta[start] = LHS[1][start];
    gamma[start] = rhs[start] / beta[start];
    for (ii=start+1; ii<=end; ii++)
    {
        beta[ii] = LHS[1][ii] - LHS[0][ii]*LHS[2][ii-1]/beta[ii-1];
        gamma[ii] = (rhs[ii] - LHS[0][ii]*gamma[ii-1])/beta[ii];
    }
    sol[end] = gamma[end];
    for (ii=end-1; ii>=start; ii--)
    {
        sol[ii] = gamma[ii] - LHS[2][ii]*sol[ii+1]/beta[ii];
    }
}

// -----
// initLHS_IBFD - Initialize LHS matrix Coefficients for the IBFD method
// -----
public static void initLHS_IBFD()
{
    //== Initialize left hand side diffraction Coefficients ==
    LHSDiff = new double[3][maxr+1];
    R=dt*ds/(G*dr*dr); Ro2 = R/2; Ro4 = R/4; Ro8 = R/8;

    LHSDiff[0][0] = 0; LHSDiff[0][maxr-1] = (-1+1/(2*(maxr-1)))*Ro8;
    LHSDiff[1][0] = 1+Ro2; LHSDiff[1][maxr-1] = 1+Ro4;
    LHSDiff[2][0] = -Ro2; LHSDiff[2][maxr-1] = 0;
    for (j=1; j<=maxr-2; j++)
    {
        LHSDiff[0][j] = (-1+1/(2*j))*Ro8;
        LHSDiff[1][j] = 1+Ro4;
        LHSDiff[2][j] = -(1+1/(2*j))*Ro8;
    }

    //== Initialize left hand side absorption Coefficients ==
    LHSAbso = new double[3][maxt+1];
    S=A*ds/(dt*dt); So2 = S/2;

    LHSAbso[0][1] = 0; LHSAbso[0][maxt-1] = -S;
    LHSAbso[1][1] = 1+S; LHSAbso[1][maxt-1] = 1+2*S;
    LHSAbso[2][1] = -S; LHSAbso[2][maxt-1] = 0;
    for (j=1; j<=maxt-1; j++)
    {
        LHSAbso[0][j] = -S;
        LHSAbso[1][j] = (1+2*S);
        LHSAbso[2][j] = -S;
    }
}

// -----
// initLHS_CNFD - Initialize LHS matrix Coefficients for the CNFD method
// -----
public static void initLHS_CNFD()
{
    //== Initialize left hand side diffraction Coefficients ==
    R=dt*ds/(G*dr*dr); Ro2 = R/2; Ro4 = R/4; Ro8 = R/8;

```

```

LHSDiff[0][0] = 0;      LHSDiff[0][maxr-1] = (-1+1/(2*(maxr-1)))*R/16;
LHSDiff[1][0] = 1+Ro4; LHSDiff[1][maxr-1] = 1+Ro8;
LHSDiff[2][0] = -Ro4;  LHSDiff[2][maxr-1] = 0;
for(j=1; j<=maxr-2; j++)
{
    LHSDiff[0][j] = (-1+1/(2*j))*R/16;
    LHSDiff[1][j] = 1+Ro8;
    LHSDiff[2][j] = -(1+1/(2*j))*R/16;
}

//== Initialize left hand side absorption Coefficients ==
S=A*ds/(dt*dt); So2 = S/2;

LHSAbso[0][1] = 0;      LHSAbso[0][maxt-1] = -So2;
LHSAbso[1][1] = 1+S;    LHSAbso[1][maxt-1] = 1+S;
LHSAbso[2][1] = -So2;   LHSAbso[2][maxt-1] = 0;
for(j=1; j<=maxt-1; j++)
{
    LHSAbso[0][j] = -So2;
    LHSAbso[1][j] = (1+S);
    LHSAbso[2][j] = -So2;
}
}

// -----
// initP - Reads in initial configuration from file
// -----
public static void initVars(String[] args) throws IOException
{
    if (args.length != 1)
        System.out.println(" You must specify initial waveform file! ");

    BufferedReader infile=new BufferedReader (new FileReader(args[0])); // open waveform file
    StringTokenizer st = new StringTokenizer(infile.readLine()); // read it all in as a String

    infile.close();

    N = Double.parseDouble(st.nextToken()); // N
    G = Double.parseDouble(st.nextToken()); // G
    A = Double.parseDouble(st.nextToken()); // A

    maxr = Integer.parseInt(st.nextToken()); // maxr
    IBFDds = Double.parseDouble(st.nextToken()); // IBFD spatial step
    CNFDds = Double.parseDouble(st.nextToken()); // CNFD spatial step
    rabc = Integer.parseInt(st.nextToken()); // # r abc
    tabc = Integer.parseInt(st.nextToken()); // # t abc
    isNon = Integer.parseInt(st.nextToken()); // is N on?
    isDon = Integer.parseInt(st.nextToken()); // is D on?
    isAon = Integer.parseInt(st.nextToken()); // is A on?
    isABCon = Integer.parseInt(st.nextToken()); // is ABC on?

    int nrpiston = Integer.parseInt(st.nextToken()); // number of elements across piston
    maxt = Integer.parseInt(st.nextToken()+1); // number of t steps
    dt = Double.parseDouble(st.nextToken()); // time step size
    tstart = Double.parseDouble(st.nextToken()); // start time
    dr = Double.parseDouble(st.nextToken()); // r step size

    p = new double[maxt+1][maxr+1]; // initialize p matrix to correct size

    for (int r = 0; r<nrpiston; r++) // read in the waveforms
        for (int t = 0; t<maxt; t++)
            p[t][r] = Double.parseDouble(st.nextToken());

    for (int r = nrpiston; r<maxr; r++) // set the rest of the values to zero
        for (int t = 0; t<maxt; t++)
            p[t][r] = 0;

    // output stuff
    totalzsteps = Integer.parseInt(st.nextToken()); // total z steps
    ors = Integer.parseInt(st.nextToken()); // out wave position - start rho
    orb = Integer.parseInt(st.nextToken()); // out wave position - skip rho
    ore = Integer.parseInt(st.nextToken()); // out wave position - end rho
    ozs = Double.parseDouble(st.nextToken()); // out wave position - start sigma
    ozb = Double.parseDouble(st.nextToken()); // out wave position - skip sigma
    oze = Double.parseDouble(st.nextToken()); // out wave position - end sigma

    workdir = st.nextToken();

    int s = Math.max(maxt,maxr);
    // init some other vars while we are at it
    rhs = new double[s+1];
    beta = new double[s+1];
    gamma = new double[s+1];
    sol = new double[s+1];
    TauDisto = new double[s+1];
    pold = new double[s+1];
    sumP = new double[s+1];
}

// -----
// OutputWaveforms -
// -----
public static void OutputWaveforms(BufferedWriter outfile) throws IOException
{
    if (ozs <= sigma)
    {
        for (int r = ors; r<= ore; r+=orb)
            writeWave(outfile, r);
        ozs += ozb;
        if (ozs > oze) ors = 999999;
    }
}

// -----
// writeWave - Writes current waveform at r to outfile
// -----

```

```

public static void writeWave(BufferedWriter outfile, int r) throws IOException
{
    //System.out.println("Writing wave to file at sigma: " + sigma+" r:" +r + "    step: "+t);
    for (int t=1; t<= maxt; t++)
        outfile.write( p[t][r]+" ");
    outfile.newLine();
}

// -----
// writeP - Writes current pressure field to outfile
// -----
public static void writeP(BufferedWriter outfile) throws IOException
{
    for (int r=0; r<=199; r++)
    {
        for (int t=1; t<= maxt; t++)
            outfile.write( p[t][r]+" ");
        outfile.newLine();
    }
}
}

```

## A2.2.2 MATLAB KZK setup and execution script (*runkzkparametric.m*)

```

function [ output_args ] = runkzkparametric( input_args )
%KZKPARAMETRIC -- Runs KZK simulation for a parametric source

% =====
% KZK Simulation Settings
% =====

% ===== % Effects to include in Simulation (1 = on, 0 = off)
isNon  = 1;      % Nonlinearity
isDon  = 1;      % Diffraction
isAcon = 1;      % Absorption
isABCon = 1;     % Absorbing Boundary Conditions

% ===== % Main Coefficients of Simulation
N      = 0.230;   % Nonlinear Coefficient
G      = 0.094;   % Diffraction/Gain Coefficient
A      = 0.860;   % Absorption Coefficient

% ===== % Transducer Configuration
radius = 0.3;     % Transducer Radius in meters
focald = 8;       % Focal length in meters

% ===== % Waveform Parameters
wo      = 50000;   % Center Frequency (normalized to this one)
F1      = 55000/wo; % First frequency component
F2      = 45000/wo; % Second frequency component
taumin  = -30;     % time range - min
taumax  = 30;      % time range - max
numtaupercycle = 20; % number of time points per wo cycle
Zpadby  = 1;       % Zero pad percent
tukeya  = 0.5;     % Tukey Window Alpha (0-1)

% ===== % Simulation Space Parameters
maxr    = 300;     % Maximum number of steps in the radial direction
ntrans  = 100;     % Number of points across the transducer
IBFDds  = 0.001;   % Step Size of IB finite difference
CNFDds  = 0.002;   % Step Size of CN finite difference
rabc    = 40;      % Number of absorbing boundary layers in the radial direction
tabc    = 40;      % Number of absorbing boundary layers in the time direction

% ===== % Where to record the Waveforms (in Meters)
outstartz = 0;     % Output to start at outstartz \
outbyz    = 0.1;   % Output by every outbyz      > In the z-Direction
outendz    = 10;   % Output to end at outendz    /
outstartxr = 0;    % Output to start at outstartxr \
outbyxr    = 0.1;  % Output by every outbyxr      > In the r-Direction
outendxr   = 0.66; % Output to end at outendxr   /

% =====
% Create initial waveform - change this is you want another type of initial waveform
% =====
Ntau=ceil((taumax-taumin)*numtaupercycle);
taumin=taumin;
taumax=taumax;
Dtau=(taumax-taumin)/Ntau;
tau=[taumin:Dtau:taumax];
Mintau = min(tau);
Ntau = length(tau);
Drho=1/(ntrans-1);
rho=[0:Drho:1];

middle = ceil((1-2*Zpadby)*Ntau)+1;
midstart = floor(Zpadby*Ntau);
zeropad(1:midstart) = 0;

for jj=1:ntrans
    dum=tau+G*(jj-1).^2*Drho.^2;
    dumm = dum(midstart:midstart+middle-1);
    p = tukeywin(middle,tukeya).*(sin(dumm.*F1*2*pi)+sin(dumm.*F2*2*pi));
    p = [zeropad p zeropad];
    Ps(jj,:)=p;
end

Ps = Ps ./max(max(Ps));

% =====
% Write Inputfile for KZK simulation - do not change the order of this!
% =====

```

```

[pname,pname] = uiputfile('user.cfg', 'Save Configuration');
fp=fopen([pname '\ ' fname], 'w');

fprintf(fp, ' %15.6f ', N); % Nonlinear Coefficient
fprintf(fp, ' %15.6f ', G); % Diffraction/Gain Coefficient
fprintf(fp, ' %15.6f ', A); % Absorption
fprintf(fp, ' %15.0f ', maxr); % maxr
fprintf(fp, ' %15.6f ', IBFDds); % IBFD ds
fprintf(fp, ' %15.6f ', CNFDds); % CNFD ds
fprintf(fp, ' %15.0f ', rabc); % # r abc
fprintf(fp, ' %15.0f ', tabc); % # t abc
fprintf(fp, ' %15.0f ', isNon); % is N on?
fprintf(fp, ' %15.0f ', isDon); % is D on?
fprintf(fp, ' %15.0f ', isAon); % is A on?
fprintf(fp, ' %15.0f ', isABCon); % is Absorbing Boundary on?
fprintf(fp, ' %8.0f ', ntrans); % number of points across transducer
fprintf(fp, ' %8.0f ', Ntau); % total number of time points
fprintf(fp, ' %15.8f ', Dtau); % time step size
fprintf(fp, ' %15.8f ', Mtau); % minimum time
fprintf(fp, ' %15.8f ', Drho); % rho step size
fprintf(fp, ' %15.8f ', Ps); % initial pressure field
fprintf(fp, ' %15.0f ', ceil((outendz-IBFDds*focald*100)/(CNFDds*focald))+100); % total z steps
fprintf(fp, ' %15.0f ', outstartr/radius*ntrans); % out waves start r
fprintf(fp, ' %15.0f ', outbyr/radius*ntrans); % out waves by r
fprintf(fp, ' %15.0f ', outendr/radius*ntrans); % out waves end r
fprintf(fp, ' %15.4f ', outstartz/focald); % out waves start z
fprintf(fp, ' %15.4f ', outbyz/focald); % out waves by z
fprintf(fp, ' %15.4f ', outendz/focald); % out waves end z

fprintf(fp, ' %s ', [ pname ]); % working directory
fclose(fp);

%=====
% Run the KZK simulation
%=====
dos(['java -Xmx500m kzk ' pname '\ ' fname]);

```

### A2.2.3 MATLAB read data output script (*ReadWaves.m*)

The following code is used to read in the KZK results and plot the wave field as the sound propagates away from the source.

```

function [ w ] = ReadWaves( fn )

in = textread(fn);

nz = 101;
nr = length(in(:,1))/nz;
nt = length(in(1,:));

w(1:nr,1:nz,1:nt) = 0;

c = 1;

for r = 1:nr
    for z = 1:nz
        for t = 1:nt
            w(r,z,t) = in((r-1)*nr+z,t);
            c = c+1;
        end
    end
end
end

```

### A2.2.4 MATLAB GUI to setup a nonlinear KZK simulation

This is a graphical user interface to automatically determine simulation parameters, save the configuration files, and run the KZK nonlinear acoustic simulations.

#### *GUI Screenshot*



Setup KZK

Medium Properties (Air)	
Density (kg/m <sup>3</sup> )	1.15
Speed of Sound (m/s)	330
Parameter of Nonlinearity (B/A)	.4
Relative Humidity (%)	20

Source Properties	
Radius (m)	.10
Center Frequency (Hz)	50000
Initial Sound Intensity (db)	120
Norm Distance	8

Output Parameters	
Total Axial Distance (m)	10
<input checked="" type="checkbox"/> Output Waveforms	
start r	0 by r .1 end r 0
start z	0 by z .1 end z 10
<input checked="" type="checkbox"/> Output Total Pressure Intensity Prof...	
Geometrical Focal Distance (m)	8

Initial Waveform	
Frequency 1 (Hz)	45000
Frequency 2 (Hz)	45000
Frequency 3 (Hz)	55000
Frequency 4 (Hz)	0
tau min	-15
tau max	15
#tau per wo cycle	100
Zero Pad Length (%)	10
Tukey Window Alpha	.5

Simulation Space Parameters	
Cells Across Source (r-dir)	100
Max cells (r-dir)	220
IBFD Sigma Step Size	0.001
CNFD Sigma Step Size	0.002
Cells for r-ABC	40
Cells for t-ABC	40

**Include These Effects**

☒ Nonlinear ☒ Diffraction ☒ Absorption

☒ Absorbing Boundary Condio...

Show Waveform Show Wavefield

Show Details

Save Configuration Load Configuration

Make KZK File Run KZK Simulation

## GUI Sourcecode

```
function varargout = setupKZK(varargin)
% SETUPKZK Application M-file for setupKZK.fig
% FIG = SETUPKZK launch setupKZK GUI.
% SETUPKZK('callback_name', ...) invoke the named callback.

% Last Modified by GUIDE v2.0 05-Aug-2005 11:51:23

if nargin == 0 % LAUNCH GUI

    fig = openfig(mfilename,'reuse');

    % Generate a structure of handles to pass to callbacks, and store it.
    handles = guihandles(fig);
    guidata(fig, handles);

    if nargin > 0
        varargout{1} = fig;
    end

elseif ischar(varargin{1}) % INVOKE NAMED SUBFUNCTION OR CALLBACK
    try
        if (nargout)
            [varargout{1:nargout}] = feval(varargin{:}); % FEVAL switchyard
        else
            feval(varargin{:}); % FEVAL switchyard
        end
    catch
        disp(lasterr);
    end
end

% -----
% This function automatically determines the simulation parameters
function [A,N,G,shock,raydist,a] = getANG(handles)

    den = str2num(get(handles.editDensity, 'string')); % density
    c = str2num(get(handles.editSos, 'string')); % speed of sound
    B = str2num(get(handles.editBoA, 'string')); % B/A - nonlinearity
    hr = str2num(get(handles.editRH, 'string')); % relative humidity
    r = str2num(get(handles.editRadius, 'string')); % radius of source
    wo = str2num(get(handles.editFreq, 'string')); % center frequency
    P = str2num(get(handles.editIntensity, 'string')); % max pressure at source
    d = str2num(get(handles.editFocal, 'string')); % focal length

    B = (1+B/2);

    % == find shock formation distance
    Ppa = 0.00002*10^(P/20); % convert db to pascals
    shock = (den*c^3)/(B*wo^2*pi*Ppa);
```

```

% == find rayleigh distance
raydist = (2*pi*wo*r^2)/(2*c);

% == find ao - absorbtion coefficient

To = 293.15; % kelvin      - Reference (room temp)
T = 293.15; % Temperature - room temp

psat = 10^(-6.8346*(273.16/T)^(1.261)+4.6151);
h = hr*(psat);

Fro = 24+4.04*10^4*h*((0.02+h)/(0.391+h));
Frn = sqrt(To/T)*(9+280*h*exp(-4.17*((To/T)^(1/3)-1)));

a = (wo^2)*(1.84*10^(-11)*sqrt(T/To)+(T/To)^(-5/2)*(0.01275*exp(-2239.1/T)/(Fro+wo^2/Fro)+0.1068*exp(-3352/T)/(Frn+wo^2/Frn)));

% == find NAG

A = a*d*(2*pi);
N = d/shock;
G = raydist/d;

% =====
% Creates the initial waveform
function [Ps, Nrho, Ntau, Dtau, Mintau, Drho] = getInitialWaveforms(handles, G)

% new! - re-normalize initial waveform

rf = str2num(get(handles.editRealFocus, 'string')); % real focal length
d = str2num(get(handles.editFocal, 'string')); % old focal length (normalized)
G = G*d;
G = G/rf;

% =====

wo = str2num(get(handles.editFreq, 'string')); % center frequency
F1 = str2num(get(handles.editF1, 'string'))/wo; % first frequency component
F2 = str2num(get(handles.editF2, 'string'))/wo; % second frequency component
F3 = str2num(get(handles.editF3, 'string'))/wo; % third frequency component
F4 = str2num(get(handles.editF4, 'string'))/wo; % forth frequency component

taumin = str2num(get(handles.edittauamin, 'string')); % min tau
taumax = str2num(get(handles.edittauamax, 'string')); % max tau
numtaupercycle = str2num(get(handles.editNumtaupercycle, 'string')); % num tau per center frequency cycle
Zpadby = str2num(get(handles.editZeropadpercent, 'string'))/100; % Zero pad percent
tukeya = str2num(get(handles.edittukeyalpha, 'string')); % Tukey Window Alpha

Ntau=ceil((taumax-taumin)*numtaupercycle);
taumin=taumin*2*pi;
taumax=taumax*2*pi;
Dtau=(taumax-taumin)/Ntau
tau=[taumin:Dtau:taumax];
Mintau = min(tau);
Ntau = length(tau);

Nrho=str2num(get(handles.edithumsource, 'string')); % Number of cells across source (in r-direction)
Drho=1/(Nrho-1);
rho={0:Drho:1};

middle = ceil((1-2*Zpadby)*Ntau)+1;
midstart = floor(Zpadby*Ntau);
zeropad(1:midstart) = 0;

%if (G == 0) Nrho = 1; end
for jj=1:Nrho
    dum=tau+(G)*(jj-1).^2*Drho.^2;
    dumm = dum(midstart:midstart+middle-1);
    p = tukeywin(middle,tukeya).^.*(sin(dumm.*F1)+sin(dumm.*F2)+sin(dumm.*F3)+sin(dumm.*F4));
    p = [zeropad p zeropad];
    Ps(jj,:)=p;
end

Ps = Ps./max(max(Ps));

% =====
function varargout = buttonShowDetails_Callback(h, eventdata, handles, varargin)

[A,N,G,shock,raydist,a] = getANG(handles);
ibfdds = str2num(get(handles.editIBFDDs, 'string'))*str2num(get(handles.editFocal, 'string'));
cnfdds = str2num(get(handles.editCNFDDs, 'string'))*str2num(get(handles.editFocal, 'string'));
totalsteps = ceil((str2num(get(handles.editEndDistance, 'string'))-ibfdds*100)/cnfdds)+100;
dr = str2num(get(handles.editRadius, 'string'))/str2num(get(handles.edithumsource, 'string'));

m = ['Detail Variables for Simulation'];
m = strvcacat(m, ['-']);
m = strvcacat(m, ['Shockwave Formation Distance: ', num2str(shock), 'm ']);
m = strvcacat(m, ['Rayleigh Distance: ', num2str(raydist), 'm']);
m = strvcacat(m, ['Absortion Coef: ', num2str(a), 'Napier/(m*Hz^2) ']);
m = strvcacat(m, ['-']);
m = strvcacat(m, ['Unitless Variables for Simulation ']);
m = strvcacat(m, ['-']);
m = strvcacat(m, ['Gain (G): ', num2str(G), '']);

```

```

m = strvcat(m, ['Absorption (A): ', num2str(A), ']);
m = strvcat(m, ['Nonlinearity (N): ', num2str(N), ']);
m = strvcat(m, ['-']);
m = strvcat(m, ['Real spatial step sizes ']);
m = strvcat(m, ['-']);
m = strvcat(m, ['z (IBFD): ', num2str(ibfdds), ' m']);
m = strvcat(m, ['z (CNFD): ', num2str(cnfdds), ' m']);
m = strvcat(m, ['r : ', num2str(dr), ' m']);
m = strvcat(m, ['Total simulation Steps : ', num2str(totalsteps)]);

msgbox(m, 'Details')

% -----
function varargout = buttonShowWaveform_Callback(h, eventdata, handles, varargin)
% Plot the Initial Waveform
[Ps] = getInitialWaveforms(handles, 0);
figure; plot(Ps(1,:)); axis tight;

% -----
function varargout = buttonShowWaveField_Callback(h, eventdata, handles, varargin)
[A,N,G] = getANG(handles);
[Ps] = getInitialWaveforms(handles, G);
figure; h = pcolor(Ps); set(h,'linestyle','none');

% -----

% -----
function varargout = buttonLoadConfiguration_Callback(h, eventdata, handles, varargin)

[fname,pname] = uigetfile('*.kzcfg', 'Open Configuration');
n=load([pname '\ ' fname]);

set(handles.editDensity, 'string', num2str(n(1))); % density
set(handles.editSos, 'string', num2str(n(2))); % speed of sound
set(handles.editBoA, 'string', num2str(n(3))); % B/A - nonlinearity
set(handles.editRH, 'string', num2str(n(4))); % relative humidity
set(handles.editRadius, 'string', num2str(n(5))); % radius of source
set(handles.editFreq, 'string', num2str(n(6))); % center frequency
set(handles.editIntensity, 'string', num2str(n(7))); % max pressure at source
set(handles.editFocal, 'string', num2str(n(8))); % focal length
set(handles.editFreq, 'string', num2str(n(9))); % center frequency
set(handles.editF1, 'string', num2str(n(10))); % first frequency component
set(handles.editF2, 'string', num2str(n(11))); % second frequency component
set(handles.editF3, 'string', num2str(n(12))); % third frequency component
set(handles.editF4, 'string', num2str(n(13))); % forth frequency component
set(handles.edittau_min, 'string', num2str(n(14))); % min tau
set(handles.edittau_max, 'string', num2str(n(15))); % max tau
set(handles.editNumtaupercycle, 'string', num2str(n(16))); % num tau per center frequency cycle
set(handles.editZeropadpercent, 'string', num2str(n(17))); % Zero pad percent
set(handles.edittukeyalpha, 'string', num2str(n(18))); % Tukey Window Alpha
set(handles.edittumsource, 'string', num2str(n(19))); % Number Cells across Source (r-dir)
set(handles.edittmaxr, 'string', num2str(n(20))); % maxr
set(handles.editIBFDds, 'string', num2str(n(21))); % IBFD sigma step size
set(handles.editCNFDds, 'string', num2str(n(22))); % CNFD sigma step size
set(handles.editrabc, 'string', num2str(n(23))); % # r abc
set(handles.edittabc, 'string', num2str(n(24))); % # t abc
set(handles.radiobuttonN, 'value', (n(25))); % is N on?
set(handles.radiobuttonD, 'value', (n(26))); % is D on?
set(handles.radiobuttonA, 'value', (n(27))); % is A on?
set(handles.radiobuttonABC, 'value', (n(28))); % is Absorbing Boundary on?
set(handles.editEndDistance, 'string', (n(29))); % end z distance
set(handles.checkboxOutWaves, 'value', (n(30))); % output Waves?
set(handles.edittowstartx, 'string', num2str(n(31))); % out waves start x
set(handles.edittowbyr, 'string', num2str(n(32))); % out waves by r
set(handles.edittowendr, 'string', num2str(n(33))); % out waves end r
set(handles.edittowstartz, 'string', num2str(n(34))); % out waves start z
set(handles.edittowbyz, 'string', num2str(n(35))); % out waves by z
set(handles.edittowendz, 'string', num2str(n(36))); % out waves end z
set(handles.checkboxOutTotalP, 'value', (n(37))); % output totalP

% -----
function varargout = buttonSaveConfiguration_Callback(h, eventdata, handles, varargin)

[fname,pname] = uiputfile('test.kzcfg', 'Save Configuration');
fp=fopen([pname '\ ' fname],'w');

fprintf(fp, '%15.4f ', str2num(get(handles.editDensity, 'string'))); % density
fprintf(fp, '%15.4f ', str2num(get(handles.editSos, 'string'))); % speed of sound
fprintf(fp, '%15.4f ', str2num(get(handles.editBoA, 'string'))); % B/A - nonlinearity
fprintf(fp, '%15.4f ', str2num(get(handles.editRH, 'string'))); % relative humidity
fprintf(fp, '%15.4f ', str2num(get(handles.editRadius, 'string'))); % radius of source
fprintf(fp, '%15.4f ', str2num(get(handles.editFreq, 'string'))); % center frequency
fprintf(fp, '%15.4f ', str2num(get(handles.editIntensity, 'string'))); % max pressure at source
fprintf(fp, '%15.4f ', str2num(get(handles.editFocal, 'string'))); % focal length
fprintf(fp, '%15.4f ', str2num(get(handles.editFreq, 'string'))); % center frequency
fprintf(fp, '%15.4f ', str2num(get(handles.editF1, 'string'))); % first frequency component
fprintf(fp, '%15.4f ', str2num(get(handles.editF2, 'string'))); % second frequency component
fprintf(fp, '%15.4f ', str2num(get(handles.editF3, 'string'))); % third frequency component
fprintf(fp, '%15.4f ', str2num(get(handles.editF4, 'string'))); % forth frequency component
fprintf(fp, '%15.4f ', str2num(get(handles.edittau_min, 'string'))); % min tau
fprintf(fp, '%15.4f ', str2num(get(handles.edittau_max, 'string'))); % max tau
fprintf(fp, '%15.4f ', str2num(get(handles.editNumtaupercycle, 'string'))); % num tau per center frequency cycle
fprintf(fp, '%15.4f ', str2num(get(handles.editZeropadpercent, 'string'))); % Zero pad percent
fprintf(fp, '%15.4f ', str2num(get(handles.edittukeyalpha, 'string'))); % Tukey Window Alpha
fprintf(fp, '%15.4f ', str2num(get(handles.edittumsource, 'string'))); % Number Cells across Source (r-dir)
fprintf(fp, '%15.4f ', str2num(get(handles.edittmaxr, 'string'))); % maxr
fprintf(fp, '%15.6f ', str2num(get(handles.editIBFDds, 'string'))); % IBFD sigma step size

```

```

fprintf(fp, ' %15.6f ', str2num(get(handles.editCNFDds, 'string'))); % CNFD sigma step size
fprintf(fp, ' %15.4f ', str2num(get(handles.editrabc, 'string'))); % # r abc
fprintf(fp, ' %15.4f ', str2num(get(handles.edittabc, 'string'))); % # t abc
fprintf(fp, ' %15.4f ', (get(handles.radioButtonN, 'value'))); % is N on?
fprintf(fp, ' %15.4f ', (get(handles.radioButtonD, 'value'))); % is D on?
fprintf(fp, ' %15.4f ', (get(handles.radioButtonA, 'value'))); % is A on?
fprintf(fp, ' %15.4f ', (get(handles.radioButtonABC, 'value'))); % is Absorbing Boundary on?
fprintf(fp, ' %15.4f ', str2num(get(handles.editEndDistance, 'string'))); % end z distance
fprintf(fp, ' %15.4f ', (get(handles.checkboxOutWaves, 'value'))); % output Waves?
fprintf(fp, ' %15.4f ', str2num(get(handles.editowstartz, 'string'))); % out waves start z
fprintf(fp, ' %15.4f ', str2num(get(handles.editowbyr, 'string'))); % out waves by r
fprintf(fp, ' %15.4f ', str2num(get(handles.editowendr, 'string'))); % out waves end r
fprintf(fp, ' %15.4f ', str2num(get(handles.editowstartz, 'string'))); % out waves start z
fprintf(fp, ' %15.4f ', str2num(get(handles.editowbyz, 'string'))); % out waves by z
fprintf(fp, ' %15.4f ', str2num(get(handles.editowendz, 'string'))); % out waves end z
fprintf(fp, ' %15.4f ', (get(handles.checkboxOutTotalP, 'value'))); % output totalP?
fclose(fp);

% -----
% -----
function varargout = buttonMakeKZKFile_Callback(h, eventdata, handles, varargin)

[A,N,G] = getANG(handles);
[Ps, Nrho, Ntau, Dtau, Mintau, Drho] = getInitialWaveforms(handles, 0); % WAG G

% -----
% -----

[fname,pname] = uiputfile('user.cfg', 'Save Configuration');
fp=fopen([pname '\ ' fname], 'w');

fprintf(fp, ' %15.6f ', N); % Nonlinear Coefficient
fprintf(fp, ' %15.6f ', G); % Diffraction/Gain Coefficient
fprintf(fp, ' %15.6f ', A); % Absorption

fprintf(fp, ' %15.0f ', str2num(get(handles.editmaxr, 'string'))); % maxr
fprintf(fp, ' %15.6f ', str2num(get(handles.editIBFDds, 'string'))); % IBFD ds
fprintf(fp, ' %15.6f ', str2num(get(handles.editCNFDds, 'string'))); % CNFD ds
fprintf(fp, ' %15.0f ', str2num(get(handles.editrabc, 'string'))); % # r abc
fprintf(fp, ' %15.0f ', str2num(get(handles.edittabc, 'string'))); % # t abc
fprintf(fp, ' %15.0f ', (get(handles.radioButtonN, 'value'))); % is N on?
fprintf(fp, ' %15.0f ', (get(handles.radioButtonD, 'value'))); % is D on?
fprintf(fp, ' %15.0f ', (get(handles.radioButtonA, 'value'))); % is A on?
fprintf(fp, ' %15.0f ', (get(handles.radioButtonABC, 'value'))); % is Absorbing Boundary on?

% Waveform and Time parameters
fprintf(fp, ' %8.0f ', Nrho);
fprintf(fp, ' %8.0f ', Ntau);
fprintf(fp, ' %15.8f ', Dtau);
fprintf(fp, ' %15.8f ', Mintau);
fprintf(fp, ' %15.8f ', Drho);
fprintf(fp, ' %15.8f ', Ps);

% output stuff
d = str2num(get(handles.editFocal, 'string'));
ibfdds = str2num(get(handles.editIBFDds, 'string'))*d;
cnfdds = str2num(get(handles.editCNFDds, 'string'))*d;

ed = str2num(get(handles.editEndDistance, 'string'));
totalsteps = ceil((ed-ibfdds*100)/cnfdds)+100;

a = str2num(get(handles.editRadius, 'string'));
ns = str2num(get(handles.editnumrsources, 'string'));

fprintf(fp, ' %15.0f ', totalsteps); % total z steps
o = round(str2num(get(handles.editowbyr, 'string'))/a*ns);
fprintf(fp, ' %15.0f ', round(str2num(get(handles.editowstartz, 'string'))/a*ns)); % out waves start z
fprintf(fp, ' %15.0f ', round(str2num(get(handles.editowbyr, 'string'))/a*ns)); % out waves by r
fprintf(fp, ' %15.0f ', round(str2num(get(handles.editowendr, 'string'))/a*ns)); % out waves end r
fprintf(fp, ' %15.4f ', str2num(get(handles.editowstartz, 'string'))/d); % out waves start z
fprintf(fp, ' %15.4f ', str2num(get(handles.editowbyz, 'string'))/d); % out waves by z
fprintf(fp, ' %15.4f ', str2num(get(handles.editowendz, 'string'))/d); % out waves end z

fprintf(fp, ' %s ', [ pname ]); % working directory

fclose(fp);

% -----
% -----
function varargout = ButtonRunKZK(h, eventdata, handles, varargin)

[fname,pname] = uigetfile('*.cfg', 'Open Configuration');
% n=load([pname '\ ' fname]);
c = ['java -Xmx500m kzk ' pname '\ ' fname];
dos(['java -Xmx500m kzk ' pname '\ ' fname]);

```

## A2.3 Periodontal Acoustic Simulation Code

### A2.3.1 Main Structure of Parallel Simulation Code

This code reads in the input files that define the simulation space geometry and distributes these values to all the nodes.

```
#include <mpi.h>
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <cmath>
#include "acousticrect.h"
#include "time.h"

using namespace std;

int maxt, outpotevery, totalz, m2m3;
int rank, numworkers;

double origin1; //origin of simulation
double origin3; //

void master();
void slave();
void syncnodes_master();
void DistributeSimulationParameters();
void dumpP(acousticrect &ar, int t);
void createCurvedDomain(acousticrect ar);
void drop_pAcum(acousticrect ar, int time);
void sendslicefix3_slave(acousticrect ar, int fix3);
void collectsllices_master(int t);
acousticrect addTip(acousticrect ar);

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numworkers); /* get number of nodes */
    numworkers--;

    if (rank == 0)
        master();
    else
        slave();

    MPI_Finalize();

    return 0;
}
```

```

void master()
{
    MPI_Status status2;
    time_t start,end;
    time (&start);

    cout << "master node is online! \n";

    DistributeSimulationParameters();

    syncnodes_master();

    for (int t=0; t<maxt; t++)//maxt
    {
        if (t%outputevery == 0 )
        {
            collectsllices_master(t);
            cout << "Collecting Slices at time: " << t << "\n";
        }

    }

    // receive aline

    double al_, tal;
    ofstream outFile("aline.ascii", ios::out);

    for (int t=0; t<maxt; t++)
    {
        al_ = 0;
        for (int n = 1; n <= numworkers; n++)
        {
            MPI_Recv(&tal_, 1, MPI_DOUBLE, n, 858, MPI_COMM_WORLD, &status2);
            al_ = al_ + tal_;
        }

        //cout << al << " ";
        outFile << al_ << " ";
    }

    outFile.close();
    time (&end);
    printf ("Total Run Time: %.2lf seconds\n", difftime (end,start) );
    return;
}

void slave()
{
    // -----
    // -- Receive Initial Data From Master
    MPI_Status status; MPI_Request request[2];
    MPI_Request request2[2];
    double simparams[10];
    MPI_Recv(&simparams, 10, MPI_DOUBLE, 0, 201, MPI_COMM_WORLD, &status);

    acousticrect ar;
    ar.num1 = simparams[0]+2;
    ar.num2 = simparams[1];
    ar.num3 = simparams[2];
    ar.ds = simparams[3];
    ar.dt = simparams[4];
    ar.den = 1000; // Default=water simparams[5];
    ar.cc = 1490; // simparams[9];
    maxt = simparams[5];
    outputevery = simparams[6];

    ar.zbeg = simparams[7];
    ar.totalz = simparams[8];

    m2m3 = ar.num2*ar.num3;

    if (rank == 1) ar.type = 1;
    else if (rank == numworkers) ar.type = 3;
    else ar.type = 2;

    ar.Init();

    // Get Drive Function from master
    MPI_Recv(&ar.drivelen, 1, MPI_INT, 0, 208, MPI_COMM_WORLD, &status);
    double *drive = new double[ar.drivelen];
    MPI_Recv(&drive[0], ar.drivelen, MPI_DOUBLE, 0, 209, MPI_COMM_WORLD, &status);
    ar.df = drive;

    // Create Curved Perio Geom
    createCurvedDomain(ar);

    // Now add the Tip and Transducer
    ar = addTip(ar);
    ar.preFindBoundaries();

    // SYNC nodes!

```

```

        int temps = rank;
        MPI_Send(&temps, 1, MPI_INT, 0, 744, MPI_COMM_WORLD);
        MPI_Recv(&temps, 1, MPI_INT, 0, 745, MPI_COMM_WORLD, &status);

        double *a__line = new double[maxt];

        // Run Simulation!
        for (int t = 0; t < maxt; t++)//maxt
        {

            // Send Aline
            a__line[t]=ar.getAline();

            // Send Slice
            if (t%outputevery == 0 )
                sendslice3_slave(ar, (int) (ar.num3/2));

            if (rank == 1) cout << " time: " << t << "/" << maxt << "      " << ar.num1 << ", " << ar.num2 << ", " << ar.num3 <<
endl;

            ar.time = t;

            ar.UpdatePs(1,1);
            if (rank > 1) MPI_Isend(&ar.pp.a[m2m3], m2m3, MPI_DOUBLE, (rank-1), 301, MPI_COMM_WORLD, request);
            ar.UpdatePs(2,ar.num1-2);
            if (rank < numworkers) MPI_Recv(&ar.pp.a[(ar.num1-1)*m2m3], m2m3, MPI_DOUBLE, (rank+1), 301,
MPI_COMM_WORLD, &status);
            ar.doDriveFunction();

            ar.UpdateVs(ar.num1-2,ar.num1-2);
            if (rank < numworkers) MPI_Isend(&ar.v1.a[(ar.num1-2)*m2m3], m2m3, MPI_DOUBLE, (rank+1), 302, MPI_COMM_WORLD,
request);
            ar.UpdateVs(1,ar.num1-3);
            if (rank > 1) MPI_Recv(&ar.v1.a[0], m2m3, MPI_DOUBLE, (rank-1), 302, MPI_COMM_WORLD, &status);

            //if (t==2500) drop_pAccum(ar,t);
            //if (t==3000) drop_pAccum(ar,t);

        }

        for (int t=0; t<maxt; t++)
            MPI_Send(&a__line[t], 1, MPI_DOUBLE, 0, 858, MPI_COMM_WORLD);

    }

void synchnodes_master()
{
    int len;MPI_Status status;
    cout << " sync nodes - ";
    for (int n = 1; n <= numworkers; n++)
    {
        MPI_Recv(&len, 1, MPI_INT, n, 744, MPI_COMM_WORLD, &status);
        cout << n << " ";
    }
    for (int n = 1; n <= numworkers; n++)
        MPI_Send(&len, 1, MPI_INT, n, 745, MPI_COMM_WORLD);
}

void DistributeSimulationParameters()
{
    char inputFilename[] = "perioin.ascii";
    ifstream inFile;
    //inFile.open("perioin.ascii", ios::in);
    inFile.open(inputFilename, ios::in);

    if (!inFile) {
        cerr << "Can't open input file " << inputFilename << endl;
        exit(1);
    }

    double *simparams = new double[10];

    inFile >> simparams[1]; //max2 Switched these to divide along longest direction
    inFile >> simparams[0]; //max1
    inFile >> simparams[2]; //max3
    inFile >> simparams[3]; //ds
    inFile >> simparams[4]; //dt
    inFile >> simparams[5]; //maxt
    inFile >> simparams[6]; //outevery

    maxt = simparams[5];
    outputevery = simparams[6];
    m2m3 = simparams[1]*simparams[2];
    totalz = simparams[0];
    simparams[8]= totalz;

    // send initial data to each node
    // -----
    int div, divaccum = 0;
    for (int n = 1; n <= numworkers; n++)
    {

```

```

        div = (totalz/(numworkers)); if ((n-1)<= (totalz%(numworkers))) div++; /* divide space along x1
direction */
        simparams[0] = div;
        simparams[7] = divaccum; // tells the worker where its starting z location is
        MPI_Send(&simparams[0], 10, MPI_DOUBLE, n, 201, MPI_COMM_WORLD);
        divaccum = divaccum+div;
    }

    // read in drive function and send to all nodes
    int dlen;
    inFile >> dlen;
    double *drivefun = new double[dlen];
    //cout << " " << dlen << " ";
    for (int i = 0; i <= dlen; i++)
        inFile >> drivefun[i];

    for (int n = 1; n <= numworkers; n++)
    {
        MPI_Send(&dlen, 1, MPI_INT, n, 208, MPI_COMM_WORLD);
        MPI_Send(&drivefun[0], dlen, MPI_DOUBLE, n, 209, MPI_COMM_WORLD);
    }

    inFile.close();
    return;
}

// dump toplate
void dumpP(acousticrect &ar, int t)
{
    stringstream strm; strm << t;
    string fname = "Pat" +strm.str()+ ".ascii";
    ofstream outFile(fname.c_str(), ios::out);

    outFile << ar.num1-2 << " " << ar.num2 << " " << ar.num3 << " ";
    for (int i3=0; i3 < ar.num3; i3++)
        for (int i2=0; i2 < ar.num2; i2++)
            for (int i1=1; i1 < ar.num1-1; i1++)
                outFile << ar.pp.val(i1,i2,i3) << " ";

    outFile.close();
    return;
}

void createCurvedDomain(acousticrect ar)
{
    double curve;
    double num2d1;
    double num2d3;
    int numzones;

    // read in 2D domain
    char inputFilename[] = "2DPerioGeom.ascii";
    ifstream inFile;
    inFile.open(inputFilename, ios::in);
    inFile >> curve; //if (rank==1) cout <<curve << " ";
    inFile >> origin3; //if (rank==1) cout <<origin3/ar.ds << " "; GLOBAL
    inFile >> origin1; //if (rank==1) cout <<origin1/ar.ds << " "; GLOBAL
    inFile >> num2d1; //if (rank==1) cout <<num2d1 << " ";
    inFile >> num2d3; //if (rank==1) cout <<num2d3 << " ";
    inFile >> numzones;

    // read in zone info
    double *zonedensities = new double[numzones];
    double *zonespeedofsounds = new double[numzones];
    for (int i1=0; i1<numzones; i1++)
    {
        inFile >> zonedensities[i1];
        inFile >> zonespeedofsounds[i1];
        //if (rank==1) cout <<zonedensities[i1] << " " << zonespeedofsounds[i1] <<" / ";
    }

    // read in 2d data
    int len = num2d1*num2d3;
    double *y = new double[len];
    for (int i1 = 0; i1<num2d1;i1++)
        for (int i2 = 0; i2<num2d3; i2++)
            inFile >> y[(i1)*(int)num2d3+i2];

    // now sweep the 2D domain to create a 3D one
    double mid3 = ar.num3/2-.5;
    double d;
    int zonenumber;
    int tmpvar;

    for (int i2=0; i2<ar.num2;i2++)
    {
        for (int i3=0; i3<ar.num3; i3++)
        {
            d = sqrt((i2+curve/ar.ds)*(i2+curve/ar.ds)+(i3-mid3)*(i3-mid3)); //sqrt this!
            d = d-curve/ar.ds;

```



```

        for (int i1 = 0; i1<ar.num1;i1++)
        {
            tmpvar = (int) (ar.zbeg+(i1-0)+origin1/ar.ds);
            zonenumber = y[tmpvar*(int) num2d3+(int) (d+origin3/ar.ds)];

            if (zonenumber == 0)
            {
                //just keep as default
                ar.B.set(i1,i2,i3,2);
            }
            else if (zonespeedofsounds[zonenumber-1] == -1) //rigid zone
            {
                ar.B.set(i1,i2,i3,2);
            }
            else
            {
                ar.c.set(i1,i2,i3,zonespeedofsounds[zonenumber-1]);
                ar.d.set(i1,i2,i3,zonedensities[zonenumber-1]);
                ar.B.set(i1,i2,i3,0);
            }
        }
    }
    return;
}

void sendslicefix3_slave(acousticrect ar, int fix3)
{
    MPI_Status status; MPI_Request request[2];
    int len = (ar.num1-2)*ar.num2;
    double *x = new double[len];
    MPI_Isend(&len, 1, MPI_INT, 0, 1151, MPI_COMM_WORLD, request);

    for (int i1=1; i1<(ar.num1-1);i1++)
        for (int i2=0; i2<ar.num2; i2++)
        {
            x[(i1-1)*ar.num2+i2]=ar.pp.val(i1,i2,fix3);
            //x[(i1-1)*ar.num2+i2]=ar.B.val(i1,i2,fix3);
        }

    MPI_Isend(&x[0], len, MPI_DOUBLE, 0, 1152, MPI_COMM_WORLD,request);

    //delete(x); <--- BIG NoNo
    return;
}

void collectsllices_master(int t)
{
    MPI_Status status;
    double *topplate;
    int len;

    stringstream strm; strm << t;
    string fname = "Slice_" +strm.str()+ ".ascii";
    ofstream outFile(fname.c_str(), ios::out);

    for (int n = 1; n <= numworkers; n++)
    {
        MPI_Recv(&len, 1, MPI_INT, n, 1151, MPI_COMM_WORLD, &status);
        if (n==1) topplate = new double[len];

        MPI_Recv(&topplate[0], len, MPI_DOUBLE, n, 1152, MPI_COMM_WORLD, &status);

        for (int i = 0; i < len; i++)
        {
            outFile << topplate[i] << " ";
        }
    }

    delete topplate;
    outFile.close();
    return;
}

void drop_pAcum(acousticrect ar, int time)
{
    stringstream strm; strm << rank;
    stringstream strmt; strmt << time;
    string fname = "PAccum_" +strm.str()+ "t"+strmt.str()+".ascii";
    ofstream outFile(fname.c_str(), ios::out);

    outFile << ar.num1 << " " << ar.num2 << " " << ar.num3 << " ";

    for (int n1 = 1; n1 < ar.num1; n1++)
        for (int n2 = 0; n2 < ar.num2; n2++)
            for (int n3 = 0; n3 < ar.num3; n3++)
                outFile << ar.pAcum.val(n1,n2,n3) << " ";

    outFile.close();
}

```

```

        return;
    }

    acousticrect addTip(acousticrect ar)
    {
        string fname = "2DTipGeom.ascii";
        ifstream inFile;
        inFile.open(fname.c_str(), ios::in);

        if (!inFile) {
            cerr << "Can't open input file " << fname.c_str() << endl;
            exit(1);
        }

        double o1, o2, o3, d1, d2, d3, scalef; // o=origin, d=vect pointing down tip, u = vect pointing up
        double num2d1, num2d2;

        inFile >> o1; // tip origin
        inFile >> o2; //
        inFile >> o3; //

        inFile >> d1; // orientation
        inFile >> d2; // vector pointing down the tip!
        inFile >> d3; //

        inFile >> scalef;

        inFile >> num2d1; // image dimensions
        inFile >> num2d2; //

        int len = num2d1*num2d2; //
        double *y = new double[len]; //
        for (int i1 = 0; i1<num2d1;i1++) // Read in tip image
            for (int i2 = 0; i2<num2d2; i2++) //
                inFile >> y[(i1)*(int)num2d2+i2]; //

        double mid3 = ar.num3/2-.5; // find mid point in 3
        int i1, i2;
        double PV1, PV2, PV3, PVmag, A;

        for (int n2=0; n2<ar.num2;n2++)
            for (int n3=0; n3<ar.num3; n3++)
                for (int n1 = 0; n1<ar.num1;n1++)
                {
                    PV1 = n1 + ar.zbeg - (o2/ar.ds - origin1/ar.ds); // find vector pointing from tip
                    PV2 = n2 - (o1/ar.ds - origin3/ar.ds); // origin to point of interest
                    PV3 = n3 - (o3/ar.ds + mid3);

                    PVmag = sqrt(PV1*PV1+PV2*PV2+PV3*PV3); // Mag of pointing vector
                    PV1 = PV1/PVmag; PV2 = PV2/PVmag; PV3 = PV3/PVmag; // normalize pointing vector

                    A = acos(PV1*d2+PV2*d1+PV3*d3); // angle between pointing vector, and down vector

                    i2 = (int) (cos(A)*PVmag*scalef+.5);
                    i1 = (int) (sin(A)*PVmag*scalef+.5);

                    if ((i1<num2d1) && (i1>=0) && (i2<num2d2) && (i2>=0))
                    {
                        //cout << " " << (int)(j1-ar.zbeg+1) << " ";

                        if (y[(i1)*(int)num2d2+i2] == 1) // tip
                        {
                            ar.B.set( n1,n2,n3,2);
                        }
                        else if (y[(i1)*(int)num2d2+i2] == 2) // transducer
                        {
                            ar.addTpoint(n1,n2,n3);
                            ar.B.set( n1,n2,n3,0);
                            //ar.B.set( n1,n2,n3,3);
                        }
                        else if (y[(i1)*(int)num2d2+i2] == 3) // water
                        {
                            ar.d.set(n1,n2,n3,998); // Default=water
                            ar.c.set(n1,n2,n3,1482); //
                            ar.B.set(n1,n2,n3,0);
                        }
                    }
                }

        inFile.close();
        return ar;
    }
}

```

### A3 - 3DPCEFIT – Cylindrical Elastic Wave Source Code

```
#include <mpi.h>
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <time.h>
#include "spipe.h"

using namespace std;
const int syncevery = 100;

void master();
void slave();
int* DistributeSimulationParameters(); // sends out simulation params to workers
void DistributeTransducers(int *zpos); // distributes transducers to the appropriate workers
void dumpTopPlate(int t);
void collectAlines();
void SyncNodes();

int rank, numworkers;

int maxt, maxz, m2m3; // max number of time steps
int outpuevery; // output every
int numtransducers; //
int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numworkers); /* get number of nodes */
    numworkers--;

    if (rank == 0)
        master();
    else
        slave();

    MPI_Finalize();
    return 0;
}

// =====
// master node! -- distribures simulation space and receives data for output
// =====
void master()
{
    time_t start,end;
    time (&start);

    int rank, div, n, i, maxz;
    int *zstartpos = new int[numworkers];

    MPI_Status status;
    cout << "master node is online! \n";
```

```

//-----
/* Send out initialization messages to each node */

zstartpos = DistributeSimulationParameters();
DistributeTransducers(zstartpos);

for (int t=0; t<maxt; t++)
{
    if (t%outputevery == 0 )
    {
        dumpTopPlate(t);
        cout << "Collecting Slices at time: " << t << "\n";
    }
    //if (t%syncevery == 0)
    //    SyncNodes();
}

cout << "Collecting A-lines! \n";
collectAlines();

time {end};
printf ("Total Run Time: %.2lf seconds\n", difftime (end,start) );
return;
}

// =====
// slave node! -- Does the grunt work
// =====
void slave()
{
    // -----
    // receive simulation parameters from master and initialize pipe section
    MPI_Status status; MPI_Request request[2];
    double simparams[15];
    MPI_Recv(&simparams, 15, MPI_DOUBLE, 0, 201, MPI_COMM_WORLD, &status);

    spipe pipe;
    pipe.numr = simparams[0]; // number of nodes in r direction
    pipe.numz = simparams[1]+2; // number of nodes in z direction
    pipe.nump = simparams[2]; // number of nodes in p direction
    pipe.ds = simparams[3]; // spatial step size in r and z (meters)
    pipe.dp = simparams[4]; // spatial step size in phi (radians)
    pipe.dt = simparams[5]; // time step size (seconds)
    pipe.den = simparams[6]; // density
    pipe.lm = simparams[7]; // Lamé constant - lambda
    pipe.mu = simparams[8]; // Lamé constant - mu
    pipe.rbeg = simparams[10]; // pipe inner radius (in ds units)
    pipe.zbeg = simparams[11]; // pipe starting z position
    maxt = simparams[12]; // number of time steps
    outputevery = simparams[13]; // output every time steps
    maxz = simparams[14]; // total number of z across entire simulation
    n2m3 = simparams[0]*simparams[2];

    if (rank == 1) pipe.pipetype = 1;
    else if (rank == numworkers) pipe.pipetype = 3;
    else pipe.pipetype = 2;

    // receive curve data
    int numz = pipe.numz-2;
    double *curveparams1 = new double[numz];
    double *curveparams2 = new double[numz];
    double *curveparams3 = new double[numz];
    MPI_Recv(&curveparams1[0], numz, MPI_DOUBLE, 0, 231, MPI_COMM_WORLD, &status);
    pipe.curvem = curveparams1;
    if (rank==numworkers) cout<<pipe.curvem[0]<< " " << pipe.curvem[numz-1] <<"\n";
    //pipe.curvem = new double[numz];
    MPI_Recv(&curveparams2[0], numz, MPI_DOUBLE, 0, 232, MPI_COMM_WORLD, &status);
    pipe.dtheta = curveparams2;
    MPI_Recv(&curveparams3[0], numz, MPI_DOUBLE, 0, 233, MPI_COMM_WORLD, &status);
    pipe.anglem = curveparams3;

    //cout << pipe.numr << " " << pipe.numz << " " << pipe.nump << " " << pipe.curvem[1] <<" 99 \n";
    pipe.Init();

    // -----
    // receive transducer parameters from master and add them to the pipe
    double tparams[6];
    bool done = false;
    while (done == false)
    {
        MPI_Recv(&tparams, 6, MPI_DOUBLE, 0, 211, MPI_COMM_WORLD, &status);
        //cout << " " << tparams[5] << " " << rank << " \n";
        if (tparams[0] == -1) done = true;
        else
        {
            transducer t(tparams[0],tparams[1],tparams[2],tparams[3],tparams[5],maxt);
            //transducer t(tparams[0],5,tparams[2],tparams[3],(int)tparams[5], maxt);
            if (tparams[4] > 0)
            {
                double *drive = new double[tparams[4]];
                MPI_Recv(&drive[0], tparams[4], MPI_DOUBLE, 0, 212, MPI_COMM_WORLD, &status);
                t.setDriveFunction(tparams[4],drive);
            }
            pipe.addTransducer(t);
        }
    }
}

```

```

    }
}

//stream outFile;      if (pipe.zbeg <= 471 && pipe.zbeg+pipe.numz >= 471)
//{
//      ofstream outFile("aalines", ios::out);
//      cout<<"I got it: "<<rank<<" "<< pipe.zbeg <<"\n";
//}

// -----
// perform simulation
double *toplate; int len;
for (int t = 0; t<maxt; t++)
{
    //if (pipe.zbeg <= 471 && pipe.zbeg+pipe.numz >= 471)
    //{
    //
    // stringstream strm; strm << t;
    // string fname = "alines_at_t" +strm.str() + ".ascii";
    // ofstream outFile(fname.c_str(), ios::out);

    //for (int y=0; y<pipe.nump; y++)
    // outFile << pipe.vr.val(471-pipe.zbeg+1,pipe.numr-1,y) << " ";

    //outFile.close();
    //}

    if (rank == 1 && t%10==0) cout << "timestep: " << t << " " << pipe.numz << " " << pipe.numr << " " << pipe.nump
    <<"\n";

    pipe.time=t;
    pipe.UpdateTransducers(t);

    // ----- Send Output to Master -----
    if (t%outputevery == 0)
    {
        len = pipe.vr.slice_fix2_count();
        toplate = new double[len];
        toplate = pipe.vr.slice_fix2(pipe.numr-1);
        MPI_Send(&len, 1, MPI_INT, 0, 401, MPI_COMM_WORLD);
        MPI_Send(&toplate[0], len, MPI_DOUBLE, 0, 402, MPI_COMM_WORLD);
        delete toplate;
    }

    // ----- Update V's -----
    pipe.UpdateVs(1,1); // Update left boundary
    pipe.UpdateVs(pipe.numz-2,pipe.numz-2); // Update right boundary

    if (rank>1) // send vz left
        MPI_Isend(&pipe.vz.a[m2m3], m2m3, MPI_DOUBLE, (rank-1), 301, MPI_COMM_WORLD, request);
    if (rank<numworkers) // send vr, vp right

    {
        MPI_Isend(&pipe.vr.a[(pipe.numz-2)*m2m3], m2m3, MPI_DOUBLE, (rank+1), 302,
MPI_COMM_WORLD, request);
        MPI_Isend(&pipe.vp.a[(pipe.numz-2)*m2m3], m2m3, MPI_DOUBLE, (rank+1), 303, MPI_COMM_WORLD,
request);
    }

    pipe.UpdateVs(2,pipe.numz-3); // update inner nodes

    if (rank<numworkers) // receive vz from right
    {
        MPI_Recv(&pipe.vz.a[(pipe.numz-1)*m2m3], m2m3, MPI_DOUBLE, (rank+1), 301, MPI_COMM_WORLD,
&status);
    }

    if (rank>1) // receive vr, vp from left
    {
        MPI_Recv(&pipe.vr.a[0], m2m3, MPI_DOUBLE, (rank-1), 302, MPI_COMM_WORLD, &status);
        MPI_Recv(&pipe.vp.a[0], m2m3, MPI_DOUBLE, (rank-1), 303, MPI_COMM_WORLD, &status);
    }

    pipe.doABCs(maxz);

    // ----- Update T's -----
    pipe.UpdateTs(1,1); // Update left boundary
    pipe.UpdateTs(pipe.numz-2,pipe.numz-2); // Update right boundary

    if (rank>1) // send Trz, Tzp left
    {
        MPI_Isend(&pipe.Trz.a[m2m3], m2m3, MPI_DOUBLE, (rank-1), 311, MPI_COMM_WORLD, request);
        MPI_Isend(&pipe.Tzp.a[m2m3], m2m3, MPI_DOUBLE, (rank-1), 312, MPI_COMM_WORLD, request);
    }

    if (rank<numworkers) // send Tzz, Trp right
    {
        MPI_Isend(&pipe.Tzz.a[(pipe.numz-2)*m2m3], m2m3, MPI_DOUBLE, (rank+1), 313,
MPI_COMM_WORLD, request);
        MPI_Isend(&pipe.Trp.a[(pipe.numz-2)*m2m3], m2m3, MPI_DOUBLE, (rank+1), 314, MPI_COMM_WORLD,
request);
    }

    pipe.UpdateTs(2,pipe.numz-3); // update inner nodes

```

```

        if (rank < numworkers) // receive Trz, Tzp from right
        {
            MPI_Recv(&pipe.Trz.a[(pipe.numz-1)*m2m3], m2m3, MPI_DOUBLE, (rank+1), 311, MPI_COMM_WORLD, &status);
            MPI_Recv(&pipe.Tzp.a[(pipe.numz-1)*m2m3], m2m3, MPI_DOUBLE, (rank+1), 312, MPI_COMM_WORLD, &status);
        }
        if (rank > 1) // receive Tzz, Trp from left
        {
            MPI_Recv(&pipe.Tzz.a[0], m2m3, MPI_DOUBLE, (rank-1), 313, MPI_COMM_WORLD, &status);
            MPI_Recv(&pipe.Trp.a[0], m2m3, MPI_DOUBLE, (rank-1), 314, MPI_COMM_WORLD, &status);
        }
    }

    // Send A-line Data to master
    MPI_Send(&pipe.numtrans, 1, MPI_INT, 0, 501, MPI_COMM_WORLD);
    for (int tr = 0; tr < pipe.numtrans; tr++)
    {
        tparams[0] = pipe.trans[tr].posi1*pipe.ds; // transducer z position (meters)
        tparams[1] = pipe.trans[tr].posi2*pipe.ds; // transducer r position (meters)
        tparams[2] = pipe.trans[tr].posi3*pipe.dp; // transducer p position (radians)
        tparams[3] = pipe.trans[tr].radius*pipe.ds; // transducer radius (meters)

        MPI_Send(&pipe.trans[tr].transID, 1, MPI_INT, 0, 502, MPI_COMM_WORLD);
        MPI_Send(&tparams[0], 4, MPI_DOUBLE, 0, 503, MPI_COMM_WORLD);
        MPI_Send(&pipe.trans[tr].record[0], maxt, MPI_DOUBLE, 0, 504, MPI_COMM_WORLD);
    }

    return;
}

// =====
// Reads in parameter file and distributes parameters to all workers. This is also
// where the simulation space is divided up.
// =====
int* DistributeSimulationParameters()
{
    char inputFilename[] = "in.file";
    ifstream inFile;
    inFile.open("in.file", ios::in);

    if (!inFile) {
        cerr << "Can't open input file " << inputFilename << endl;
        exit(1);
    }

    double *simparams = new double[15];

    inFile >> simparams[0]; //pipe.numr; // number of nodes in r direction
    inFile >> simparams[1]; //pipe.numz; // number of nodes in z direction
    inFile >> simparams[2]; //pipe.numphi; // number of nodes in phi direction
    inFile >> simparams[3]; //pipe.ds; // spatial step size in r and z (meters)
    inFile >> simparams[4]; //pipe.dp; // spatial step size in phi (radians)
    inFile >> simparams[5]; //pipe.dt; // time step size (seconds)
    inFile >> simparams[6]; //pipe.den; // density
    inFile >> simparams[7]; //pipe.lm; // Lamé constant - lambda
    inFile >> simparams[8]; //pipe.mu; // Lamé constant - mu
    inFile >> simparams[10]; //pipe.rbeg; // pipe inner radius (in ds units)
    inFile >> maxt; // number of time steps
    inFile >> outputevery; // number of nodes in x3 direction
    simparams[12] = maxt; simparams[13] = outputevery;
    simparams[14] = simparams[1];
    m2m3 = simparams[0]*simparams[2];

    maxz = simparams[1];
    double *cur = new double[maxz]; for(int i = 0; i < maxz; i++) inFile >> cur[i];
    double *dth = new double[maxz]; for(int i = 0; i < maxz; i++) inFile >> dth[i];
    double *ang = new double[maxz]; for(int i = 0; i < maxz; i++) inFile >> ang[i];

    // send initial data to each node
    int div, divaccum = 0;
    int* zpos = new int[numworkers];
    for (int n = 1; n <= numworkers; n++)
    {
        div = (maxz/(numworkers)); if ((n-1) < (maxz%(numworkers))) div++; // divide space along x1 direction */
        simparams[11] = div;
        MPI_Send(&simparams[0], 15, MPI_DOUBLE, n, 201, MPI_COMM_WORLD);

        MPI_Send(&cur[divaccum], div, MPI_DOUBLE, n, 231, MPI_COMM_WORLD);
        MPI_Send(&dth[divaccum], div, MPI_DOUBLE, n, 232, MPI_COMM_WORLD);
        MPI_Send(&ang[divaccum], div, MPI_DOUBLE, n, 233, MPI_COMM_WORLD);

        zpos[n-1] = simparams[11]; divaccum = divaccum+div;
    }

    inFile.close();
    return zpos;
}

// =====
// Reads in transducer file and distributes transducers to the correct workers.

```

```

// =====
void DistributeTransducers(int *zpos)
{
    double *drive;
    double tparams[6];
    int drivelen, numtrans, worker;

    char inputFilename[] = "trans.file";
    ifstream inFile;
    inFile.open("trans.file", ios::in);

    if (!inFile) {
        cerr << "Can't open input file " << inputFilename << endl;
        exit(1);
    }

    inFile >> numtrans;
    cout << " number of transducers: " << numtrans << endl;
    numtransducers = numtrans;

    for (int tr = 0; tr < numtrans; tr++)
    {
        inFile >> tparams[0]; // tposz; // transducer z location
        inFile >> tparams[1]; // tposr; // transducer r location
        inFile >> tparams[2]; // tposp; // transducer p location
        inFile >> tparams[3]; // trad; // transducer radius
        inFile >> tparams[4]; // drivelen; // len of drive function
        tparams[5] = tr;

        if (tparams[4] > 0)
        {
            drive = new double[tparams[4]];
            for (int i = 0; i < tparams[4]; i++)
                inFile >> drive[i];
        }

        // find which worker gets the transducer
        worker = 0;
        for (int tosend = 1; tosend < numworkers; tosend++)
            if (tparams[0] >= zpos[tosend-1] && tparams[0] < zpos[tosend]) worker = tosend;
            if (tparams[0] >= zpos[numworkers-1] && tparams[0] < maxz) worker = numworkers;
        else if (worker == 0) cout << "error: transducer postion not found: zpos - " << tparams[0] << ", " <<
            zpos[numworkers-1] << ", " << maxz << endl;

        // send the transducer info to worker
        if (worker > 1)
            if ((tparams[0] - tparams[3]) <= zpos[worker])
            {
                MPI_Send(&tparams[0], 6, MPI_DOUBLE, worker-1, 211, MPI_COMM_WORLD);
                if (tparams[4] > 0) MPI_Send(&drive[0], tparams[4], MPI_DOUBLE, worker-1, 212, MPI_COMM_WORLD);
            }

            MPI_Send(&tparams[0], 6, MPI_DOUBLE, worker, 211, MPI_COMM_WORLD);
            if (tparams[4] > 0) MPI_Send(&drive[0], tparams[4], MPI_DOUBLE, worker, 212, MPI_COMM_WORLD);

            if (worker < numworkers)
                if ((tparams[0] + tparams[3]) >= zpos[worker+1])
                {
                    MPI_Send(&tparams[0], 6, MPI_DOUBLE, worker+1, 211, MPI_COMM_WORLD);
                    if (tparams[4] > 0) MPI_Send(&drive[0], tparams[4], MPI_DOUBLE, worker+1, 212, MPI_COMM_WORLD);
                }
        }

        // send all workers a message letting them know we are done distributing transducers
        tparams[0] = -1; tparams[1] = -1; tparams[2] = -1; tparams[3] = -1; tparams[4] = -1; tparams[5] = -1;
        for (int n = 1; n <= numworkers; n++)
            MPI_Send(&tparams[0], 5, MPI_DOUBLE, n, 211, MPI_COMM_WORLD);

        inFile.close();
        delete drive;

        return;
    }
}

void SyncNodes()
{
    int s; MPI_Status status;
    for (int n = 1; n <= numworkers; n++)
        MPI_Recv(&s, 1, MPI_INT, n, 721, MPI_COMM_WORLD, &status);
    for (int n = 1; n <= numworkers; n++)
        MPI_Send(&n, 1, MPI_INT, n, 722, MPI_COMM_WORLD);
    cout << " nodes synced \n";
}

// dump toplate
void dumpTopPlate(int t)
{
    MPI_Status status;
    double *topplate;
    int len;

    stringstream strm; strm << t;
}

```

```

string fname = "toplate_at_t" + strm.str() + ".ascii";
ofstream outFile(fname.c_str(), ios::out);

for (int n = 1; n <= numworkers; n++)
{
    MPI_Recv(&len, 1, MPI_INT, n, 401, MPI_COMM_WORLD, &status);
    toplate = new double[len];
    MPI_Recv(&topplate[0], len, MPI_DOUBLE, n, 402, MPI_COMM_WORLD, &status);

    //cout << " << " << h[0] << "\n";
    for (int i = 1; i <= len; i++)
        outFile << toplate[i] << " ";

    outFile.close();
    delete toplate;
    return;
}

void collectAlines()
{
    MPI_Status status;

    array3D alines;
    alines.Init(1,numtransducers,maxt+4);

    int numts, ct;
    double tparams[4];
    double *rec = new double[maxt];

    for (int n = 1; n <= numworkers; n++)
    {
        MPI_Recv(&numts, 1, MPI_INT, n, 501, MPI_COMM_WORLD, &status);

        for (int i = 0; i < numts; i++)
        {
            MPI_Recv(&ct, 1, MPI_INT, n, 502, MPI_COMM_WORLD, &status);
            //cout << n << " 1\n";
            MPI_Recv(&tparams, 4, MPI_DOUBLE, n, 503, MPI_COMM_WORLD, &status);
            //cout << n << " 2\n";
            MPI_Recv(&rec[0], maxt, MPI_DOUBLE, n, 504, MPI_COMM_WORLD, &status);
            //cout << n << " 3\n";

            alines.set(0,ct,0,tparams[0]); // trans xpos
            alines.set(0,ct,1,tparams[1]); // trans rpos
            alines.set(0,ct,2,tparams[2]); // trans ppos
            alines.set(0,ct,3,tparams[3]); // trans radius

            for (int t = 0; t < maxt; t++)
                alines.set(0,ct,4+t,(alines.val(0,ct,4+t) + rec[t]));
        }

        string fname = "alines.ascii";
        ofstream outFile(fname.c_str(), ios::out);

        for (int n = 0; n < numtransducers; n++)
        {
            for (int i = 0; i < (maxt+4); i++)
                outFile << alines.val(0,n,i) << " ";
            outFile << "\n";
        }

        outFile.close();
        return;
    }
}

#include <iostream>
#include "array3D.h"
#include "array3D_int.h"
#include "transducer.h"

#define min(a,b) (((a)<(b))?(a):(b))
#define max(a,b) (((a)>(b))?(a):(b))

class spipe
{
public:
    spipe() { }
    ~spipe() { }

    int numr; // number of grid points in r direction
    int numz; // number of grid points in z direction
    int nump; // number of grid points in p direction

    int abc; // number of abc points on each end

    double ds; // spatial step size in r and z direction (meters)
    double dp; // angular step size in p direction (radians)
    double dt; // time step size (seconds)

    double den; // density (kg/m^3)
    double lm; // lame constant - lamda

```



```

double mu;          // lame constant - mu

double *curvem;
double *anglem;
double *dtheta;

double zbeg;        // z start position (meters)
double rbeg;        // r start position (meters) !inner pipe radius!

int pipetype;       // pipe type 1 = left end , 2 = middle, 3 = right end

array3D vr;         // r - velocities
array3D vz;         // z - velocities
array3D vp;         // p - velocities
array3D Trr;        // rr - normal stress
array3D Tzz;        // zz - normal stress
array3D Tpp;        // pp - normal stress
array3D Trz;        // rz - shear stress
array3D Trp;        // rp - shear stress
array3D Tzp;        // zp - shear stress

array3D_int B;      // Boundary Array

int time;

transducer *trans;
int numtrans;

private:

double dtodsp;
double lmdtods;
double l2mdtods;
double mdtods;

int r,z,p,ppl,pml;
double ro,ri,rr,co,ci,cc;

double PIo2;

public:

void Init()
{
vr.Init(numz,numr,nump);
vz.Init(numz,numr,nump);
vp.Init(numz,numr,nump);
Trr.Init(numz,numr,nump);
Tzz.Init(numz,numr,nump);
Tpp.Init(numz,numr,nump);
Trz.Init(numz,numr,nump);
Trp.Init(numz,numr,nump);
Tzp.Init(numz,numr,nump);
B.Init(numz+2,numr+2,nump+2, pipetype);

dtodsp = (dt)/(den*ds);
lmdtods = (lm*dt)/ds;
l2mdtods = ((lm+2*mu)*dt)/ds;
mdtods = (mu*dt)/ds;

PIo2 = 3.14159265358979/2;

numtrans=0;
time = 0;
abc = 80;
}

void UpdateVs(int zs, int zend)
{
for (z = zs; z <= zend; z++)
{
vr.setindx(z,0,0); vz.setindx(z,0,0); vp.setindx(z,0,0);
Trr.setindx(z,0,0); Tzz.setindx(z,0,0); Tpp.setindx(z,0,0);
Trz.setindx(z,0,0); Trp.setindx(z,0,0); Tzp.setindx(z,0,0);

for (r = 0; r < numr; r++)
{
B.setindx(z+1,r+1,1);

for (p = 0; p < nump; p++)
{
rr = r+rbeg;
ro = (rr+0.5);
ri = (rr-0.5);
if (curvem[z-1]>0)
{
ci = (ri*ds*sin(p*dp-PIo2-anglem[z-1])+curvem[z-1]);
cc = (rr*ds*sin(p*dp-PIo2-anglem[z-1])+curvem[z-1]);

```

```

        co = (ro*ds*sin(p*dp-Pio2-anglem[z-1])+curvem[z-1]);
    }
    else { ci = 1; cc = 1; co = 1; }

    ri = ri*ci;

    ro = ro*co;
    //if (z==numz-2 && r==0 && p==10 && pipetype==2) std::cout << Tpp.v() << " " << Tpp.v2m() << "\n";

    if (B.v()==0)
        vr.sv( vr.v() + dtodsp*((1/(rr*cc))*(ro*Trr.v2p()-
ri*Trr.v()+1/(rr*dp))*(Trp.v()-Trp.v3m())+(ds/(cc*dtheta[z-1]))*(Trz.vlp()-Trz.v()+1/(2*rr))*(Trr.v2p()+Trr.v()-
Tpp.v2p()-Tpp.v())) );
    else if (B.v()==2 | B.vlp()==2 | B.v3m()==2) {}
    //else if (B.v2p()==2 & B.v()>=1000) vr.sv( vr.v() - dtodsp*(trans[B.v()-1000].drivef(time)+2*Trr.v()
+((1/(2*rr))*(3*Tpp.v()-Tpp.v2m())) ) );
    //vr.sv( vr.v() + dtodsp*(trans[B.v()-
1000].drivef(time)+1/(rr)*((1/(2*rr))*(3*Tpp.v()-Tpp.v2m())) ) );
    else if (B.v2p()==2)
        vr.sv( vr.v() - dtodsp*(2*Trr.v() + (1/(2*rr))*(3*Tpp.v() -
Tpp.v2m())) );
    else if (B.v2m()==2)
        vr.sv( vr.v() + dtodsp*(2*Trr.v2p() + (1/(2*rr))*(3*Tpp.v2p()-
Tpp.v2p2())) );
    else
        vr.sv( vr.v() + dtodsp*((1/(rr*cc))*(ro*Trr.v2p()-
ri*Trr.v()+1/(rr*dp))*(Trp.v()-Trp.v3m())+(ds/(cc*dtheta[z-1]))*(Trz.vlp()-Trz.v()+1/(2*rr))*(Trr.v2p()+Trr.v()-
Tpp.v2p()-Tpp.v())) );

    rr = r+rbeg-0.5;
    ro = (rr+0.5);
    ri = (rr-0.5);
    if (curvem[z-1]>0)
    {
        ci = (ri*ds*sin(p*dp-Pio2-anglem[z-1])+curvem[z-1]);
        cc = (rr*ds*sin(p*dp-Pio2-anglem[z-1])+curvem[z-1]);
        co = (ro*ds*sin(p*dp-Pio2-anglem[z-1])+curvem[z-1]);
    }
    else { ci = 1; cc = ci; co = ci; }
    ri = ri*ci;
    ro = ro*co;

    // vz -----
    if (B.v()==0)
        vz.sv( vz.v() + dtodsp*((1/(rr*cc))*(ro*Trz.v()-
ri*Trz.v2m()+1/(rr*dp))*(Tzp.v()-Tzp.v3m())+(ds/(cc*dtheta[z-1]))*(Tzz.vlp()-Tzz.vlm()+1/(2*rr))*(Trz.v()+Trz.v2m()))
);
    else if (B.v()==2 | B.v2m()==2 | B.v3m()==2) {}
    else if (B.v2p()==2 & B.v()>=1000) vz.sv( vz.v() - dtodsp*(trans[B.v()-
1000].drivef(time)+2*Tzz.vlm()) );
    else if (B.vlp()==2)
        vz.sv( vz.v() - dtodsp*(2*Tzz.vlm()) );
    else if (B.vlm()==2)
        vz.sv( vz.v() + dtodsp*(2*Tzz.v()) );
    else
        vz.sv( vz.v() + dtodsp*((1/(rr*cc))*(ro*Trz.v()-
ri*Trz.v2m()+1/(rr*dp))*(Tzp.v()-Tzp.v3m())+(ds/(cc*dtheta[z-1]))*(Tzz.vlp()-Tzz.vlm()+1/(2*rr))*(Trz.v()+Trz.v2m()))
);

    // vp -----
    if (B.v()==0)
        vp.sv( vp.v() + dtodsp*((1/(rr*cc))*(ro*Trp.v()-
ri*Trp.v2m()+1/(rr*dp))*(Tpp.v3p()-Tpp.v()+1/(rr*cc*dtheta[z-1]))*(Tzp.vlp()-Tzp.v()+1/(rr)*(Trp.v()+Trp.v2m())) );
    else if (B.v()==2 | B.v2m()==2 | B.vlp()==2) {}
    else if (B.v3p()==2)
        vp.sv( vp.v() - dtodsp*(2*Tpp.v()) );
    else if (B.v3m()==2)
        vp.sv( vp.v() + dtodsp*(2*Tpp.v3p()) );
    else
        vp.sv( vp.v() + dtodsp*((1/(rr*cc))*(ro*Trp.v()-
ri*Trp.v2m()+1/(rr*dp))*(Tpp.v3p()-Tpp.v()+1/(rr*cc*dtheta[z-1]))*(Tzp.vlp()-Tzp.v()+1/(rr)*(Trp.v()+Trp.v2m())) );

    B.incindx(); vr.incindx(); vz.incindx(); vp.incindx();
    Trr.incindx(); Tzz.incindx(); Tpp.incindx(); Trz.incindx(); Trp.incindx(); Tzp.incindx();
}
}
}

void UpdateTs(int zs, int zend)
{
    for (z = zs; z <= zend; z++)
    {
        vr.setindx(z,0,0); vz.setindx(z,0,0); vp.setindx(z,0,0);
        Trr.setindx(z,0,0); Tzz.setindx(z,0,0); Tpp.setindx(z,0,0);
        Trz.setindx(z,0,0); Trp.setindx(z,0,0); Tzp.setindx(z,0,0);

        for (r = 0; r < numr; r++)
        {
            B.setindx(z+1,r+1,1);

            for (p = 0; p < nump; p++)
            {
                //rr = r+rbeg-0.5;
                rr = r+rbeg;
                ro = (rr+0.5);
                ri = (rr-0.5);

                if (curvem[z-1]>0)
                {
                    ci = (ri*ds*sin(p*dp-Pio2-anglem[z-1])+curvem[z-1]);
                    cc = (rr*ds*sin(p*dp-Pio2-anglem[z-1])+curvem[z-1]);
                    co = (ro*ds*sin(p*dp-Pio2-anglem[z-1])+curvem[z-1]);
                }
                else { ci = 1; cc = 1; co = 1; }
                ri = ri*ci;
            }
        }
    }
}

```

```

    ro = ro*co;

    // Tii
    if (B.v()==2 | B.v2m()==2 | B.v3m()==2 | B.vlp()==2) {}
    else
    {
        Trr.sv( Trr.v() + (12mdtods/(rr*cc))*(ro*vr.v()-ri*vr.v2m()) + lmdtods*((1/(rr*dp))*(vp.v()-
        vp.v3m())+(ds/(cc*dtheta[z-1]))*(vz.vlp()-vz.v())+(1/(2*rr))*(vr.v()+vr.v2m())) );
        Tpp.sv( Tpp.v() + 12mdtods*((1/(rr*dp))*(vp.v()-
        vp.v3m())+(1/(2*rr))*(vr.v()+vr.v2m()))+lmdtods*((1/(rr*cc))*(ro*vr.v()-ri*vr.v2m())+(ds/(cc*dtheta[z-1]))*(vz.vlp()-
        vz.v())) );
        Tzz.sv( Tzz.v() + 12mdtods*(ds/(cc*dtheta[z-1]))*(vz.vlp()-
        vz.v())+lmdtods*((1/(rr*dp))*(vp.v()-vp.v3m())+(1/(cc*rr))*(ro*vr.v()-ri*vr.v2m())+(1/(2*rr))*(vr.v()+vr.v2m())) );
    }

    // Tzp
    if (B.v()==2 | B.vlp()==2 | B.vlm()==2 | B.v3p()==2 | B.v3m()==2 | B.v2m()==2) {}
    else Tzp.sv( Tzp.v() + mdtods*( ds/(cc*dtheta[z-1]))*(vp.v()-vp.vlm())+1/(rr*dp)*(vz.v3p()-vz.v()) );
    //z[p] = Tzp[p] + mdtods* ( ds/(cc*dtheta[z])*(vp[r][z][p]-vp[r][z-
    1][p])+1/(rr*dp)*(vz[r][z][p]-vz[r][z-1][p])) );

    rr = r+rbeg;
    ro = (rr+0.5);
    ri = (rr-0.5);

    if (curvem[z-1]>0)
    {
        ci = (ri*ds*sin(p*dp-Pto2-anglem[z-1])+curvem[z-1]);
        cc = (rr*ds*sin(p*dp-Pto2-anglem[z-1])+curvem[z-1]);
        co = (ro*ds*sin(p*dp-Pto2-anglem[z-1])+curvem[z-1]);
    }
    else { ci = 1; cc = ci; co = ci; }
    ri = ri*ci;
    ro = ro*co;

    // Trp
    if (B.v()==2 | B.v2p()==2 | B.v2m()==2 | B.v3p()==2 | B.v3m()==2 | B.vlp()==2) {}
    else Trp.sv( Trp.v() + (mdtods/rr)*((1/dp)*(vr.v3p()-vr.v())+(1/cc)*(ro*vp.v2p()-ri*vp.v())-
    (1/2)*(vp.v2p()+vp.v())) );

    // Trz
    if (B.v()==2 | B.v2p()==2 | B.v2m()==2 | B.vlp()==2 | B.vlm()==2 | B.v3m()==2) {}
    else Trz.sv( Trz.v() + mdtods*( (1/(cc*rr))*(ro*vz.v2p()-ri*vz.v())+(ds/(cc*dtheta[z-1]))*(vr.v()-
    vr.vlm())) );

    B.incindx(); vr.incindx(); vz.incindx(); vp.incindx();
    Trr.incindx(); Tzz.incindx(); Tpp.incindx(); Trz.incindx(); Trp.incindx(); Tzp.incindx();

}
}
}

void doABCs(int TotalZ)
{
    double per;
    for (int i1 = min(abc+1,zbeg+numz-1); (i1>=zbeg); i1--)
    {
        per = (1-.0015*(abc-i1));
        vr.setindx(i1-zbeg,0,0); vz.setindx(i1-zbeg,0,0); vp.setindx(i1-zbeg,0,0);
        for (int i2 = 0; i2 < numr; i2++)
            for (int i3 = 0; i3 < nump; i3++)
            {
                vz.sv(vz.v()*per);vr.sv(vr.v()*per);vp.sv(vp.v()*per);
                vr.incindx(); vz.incindx(); vp.incindx();
            }

        for (int i1 = max(TotalZ-abc-1,zbeg); (i1 >= zbeg) & (i1<(zbeg+numz-1)); i1++)
        {
            vr.setindx(i1-zbeg,0,0); vz.setindx(i1-zbeg,0,0); vp.setindx(i1-zbeg,0,0);
            per = (1+.0015*(-i1+(TotalZ-abc-1)));
            //std::cout << time << ", " << i1 << ", " << per<<'\n';
            //if (pipetype==3) std::cout << i1 << ", " << i1-zbeg << ", " << per<<'\n';
            for (int i2 = 0; i2 < numr; i2++)
                for (int i3 = 0; i3 < nump; i3++)
                {
                    vz.sv(vz.v()*per);vr.sv(vr.v()*per);vp.sv(vp.v()*per);
                    vr.incindx(); vz.incindx(); vp.incindx();
                }
        }
    }

    void UpdateTransducers(int t)
    {
        int tr;

        for (int i1 = 1; i1<numz-1; i1++)
            for (int i3 = 0; i3<nump; i3++)
                if (B.val(i1+1,numr,i3+1) >= 1000)
                {

```

```

        tr = B.val(il+1,numr,i3+1)-1000;
        trans[tr].record[t] = trans[tr].record[t] + vz.val(il,numr-1,i3);
    }

    void addTransducer(transducer t)
    {
        numtrans = numtrans+1;
        transducer *temp = new transducer(numtrans);
        for (int i = 0; i<numtrans-1; i++)
            temp[i] = trans[i];
        temp[numtrans-1] = t;
        trans = temp;
        int nnodes = 0;
        //if (trans[numtrans-1].driven)
        for (int il = 1; il<numz-1; il++)
            for (int i3 = 0; i3<numr; i3++)
                if (((trans[numtrans-1].posil-zbeg-il)*(trans[numtrans-1].posil-zbeg-il)+(trans[numtrans-1].posi3-i3)*(t.posi3-i3))<=(trans[numtrans-1].radius*trans[numtrans-1].radius))
                {
                    B.set(il+1,t.posi2+1,i3+1,1000+numtrans-1);
                    nnodes++;
                }
        //if (t.posil-zbeg+1 >0) B.set(t.posil-zbeg+1,t.posi2+1,t.posi3+1,1000+numtrans-1);
        temp[numtrans-1].numnodes = nnodes;
    }
};

class transducer
{
private:
    double *drive; // array that holds drive function

    int dflen; // length of drivefunc

public:
    double posil; // transducer center (r-direction) - meters
    double posi2; // transducer center (z-direction) - meters
    double posi3; // transducer center (p-direction) - angle

    double radius; // transducer radius - meters
    bool driven; // driven = true - active (pitch or pitch/catch)
                // = false - passive (catch)
    int transID;
    int numnodes; // number of nodes in simulation space

    double *record; // array that holds recorded value

    // Blank Constructor
    transducer() {driven = false;}

    //
    transducer(double x1, double x2, double x3, double rad, int tID, int maxt)
    {
        posil = x1;
        posi2 = x2;
        posi3 = x3;
        radius = rad;
        transID = tID;
        driven = false;

        dflen=0;

        record = new double[maxt];
        for (int i = 0; i< maxt; i++) record[i] = 0;
    }

    // Blank Deconstructor
    ~transducer() {}

    // Init - defines the array and its dimensions - MUST BE CALLED BEFORE USING
    void setDriveFunction(int len, double df[])
    {
        //drive = new double[len];
        drive = df;
        dflen = len;
        driven = true;
        return;
    }

    double drivef(int t)
    {
        if (t<dflen)
            return drive[t];
        else
            return 0;
    }
};

```