

2004

Ultrasonic guided wave tomography of pipes: A development of new techniques for the nondestructive evaluation of cylindrical geometries and guided wave multi-mode analysis

Kevin Raymond Leonard

College of William and Mary - Virginia Institute of Marine Science

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



Part of the [Acoustics, Dynamics, and Controls Commons](#), and the [Applied Mechanics Commons](#)

Recommended Citation

Leonard, Kevin Raymond, "Ultrasonic guided wave tomography of pipes: A development of new techniques for the nondestructive evaluation of cylindrical geometries and guided wave multi-mode analysis" (2004). *Dissertations, Theses, and Masters Projects*. Paper 1539616737.

<https://dx.doi.org/doi:10.25773/v5-b5xv-ve64>

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.

ULTRASONIC GUIDED WAVE TOMOGRAPHY OF PIPES

A Development of New Techniques for the Nondestructive Evaluation of Cylindrical Geometries and Guided Wave Multi-mode Analysis

A Dissertation

Presented to

The Faculty of the Department of Applied Science

The College of William and Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

by

Kevin Raymond Leonard

2004

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of

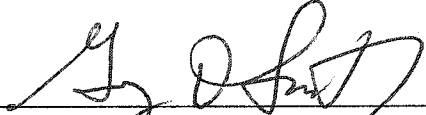
The requirements for the degree of

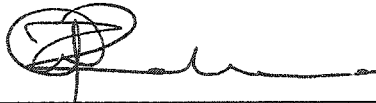
Doctor of Philosophy



Kevin Raymond Leonard

Approved by the Committee, May 2004


Mark K. Hinders, Chair


Gregory D. Smith


Zia-ur Rahman


Robert E. Welsh

To my wife and parents for all of their support, encouragement, and love

Table of Contents

Acknowledgements	vi
List of Tables.....	vii
List of Figures	viii
Abstract.....	xiii
Chapter 1 Introduction.....	2
1.1 Guided Waves.....	2
1.2 Tomography.....	6
1.3 Helical Ultrasound Tomography (HUT), Meridional Ultrasound Tomography (MUT) and Multi-Mode Analysis.....	10
Chapter 2 Fundamentals	14
2.1 Lamb Waves	14
2.2 Lamb Wave Experimental Considerations.....	21
2.2.1 <i>Coupling</i>	24
2.2.2 <i>Transduction</i>	25
2.2.3 <i>Bandwidth</i>	28
2.2.4 <i>Discussion</i>	31
2.3 NDE Applications	32
2.4 Tomography.....	34
Chapter 3 Lamb Wave Tomography (LWT).....	35
3.1 LWT Experimental Overview	37
3.2 Parallel Projection Tomography	40
3.3 Fan Beam Tomography	44
3.4 Crosshole Tomography	49
3.4.1 <i>Simultaneous Iterative Reconstruction Technique (SIRT)</i>	55
3.4.2 <i>Composite Plate Results</i>	58
3.4.3 <i>Blind Study Test</i>	64

3.5	Chapter Summary.....	73
Chapter 4	Helical and Meridional Ultrasound Tomography.....	75
4.1	Introduction.....	75
4.2	Description of Setup and Experimental Results	82
4.3	HUT Discussion.....	86
4.4	Meridional Ultrasound Tomography.....	92
4.4.1	<i>MUT Experimental Setup.....</i>	<i>98</i>
4.4.2	<i>MUT Results and Discussion.....</i>	<i>102</i>
Chapter 5	Advanced Signal Processing.....	108
5.1	Tomographic Frequency Compounding.....	109
5.2	Multi-Mode Arrival Time Extraction Algorithms	115
5.2.1	<i>Wavelet Analysis</i>	<i>121</i>
5.2.2	<i>Dynamic Wavelet Fingerprinting.....</i>	<i>123</i>
5.2.3	<i>Sorting Algorithm.....</i>	<i>125</i>
5.3	Discussion.....	156
Chapter 6	Conclusions and Future Work.....	159
Appendix A	Source Code for the Numerical Solution of the Lamb Wave Dispersion Relations.....	165
Appendix B	Matlab Code for Multi-mode Arrival Sorting.....	211
Appendix C	Source Code for the HUT Scanner.....	224
References	294
Vita	305

Acknowledgements

First, I want to thank Professor Hinders for his guidance and support throughout. Without his insight and mentorship, this work would not be possible. I am also grateful to Professors Smith, Rahman, and Welsh for their careful reading and criticism of the manuscript. Furthermore, this work would not have been possible without the assistance I've received in the lab from my colleagues – Jonathan Stevens for the construction and ingenuity behind the mechanical scanners seen in this work, Dr. Eugene Malyarenko for his mentorship and prior work on Lamb wave tomography, and Jill Bingham, Kevin Rudd, Ian Groom, and Daniel Bowring for collecting data and assisting with data analysis.

Lastly, I would like to thank my family -- my parents for all they have done and sacrificed for me, and my wife, Heather, without whose support, encouragement, and love I wouldn't have been able to finish

List of Tables

Table 1-1. Guided Wave Benefits	3
Table 5-1. Image Quality Parameters for Figure 5-1	114
Table 5-2. Image Quality Parameters for Figure 5-2	114
Table 5-3. Image Quality Parameters for the three flawed samples.	155

List of Figures

Figure 1-1. Three different Lamb wave tomography geometries.....	8
Figure 1-2. Parallel circumferential arrays of transducers can be seen to give a cross-hole tomographic geometry when the pipe is mentally “unwrapped.”.....	12
Figure 2-1. The two families of Lamb wave modes generated in a thin plate.....	15
Figure 2-2. Coordinate system for a plate of thickness $2b$	15
Figure 2-3. Phase and group velocity dispersion curves for a 3mm thick aluminum plate, where $c_L = 6.32 \text{ mm}/\mu\text{s}$ and $c_T = 3.13 \text{ mm}/\mu\text{s}$	22
Figure 2-4. Displacement curves for the two lowest order symmetric modes in an aluminum plate at two frequency-thickness products.....	23
Figure 2-5. Diagram of an variable incidence angle block transducer	27
Figure 2-6. Plots demonstrating the effects of dispersion due to an input signal’s bandwidth for a highly dispersive mode, S_0 , and a non-dispersive mode, A_0	29
Figure 3-1. Lamb wave dispersion curves showing phase velocity (a) and group velocity (b) versus frequency-thickness product for aluminum	39
Figure 3-2. The geometry for parallel-projection tomography is shown for the case of seven parallel projections at four orientations.....	41
Figure 3-3. The parallel projection scanning system is shown schematically	43
Figure 3-4. Tomographic reconstructions of a 3/32” thick aluminum plate with a 1” diameter flat bottom hole 30% thickness loss: (a) parallel projection result, (b) fanbeam result, and (c) multiple crosshole result where the scanned area is 20 cm x 20 cm.	45
Figure 3-5. Schematic of Lamb wave fanbeam tomography scanner	47
Figure 3-6. Photograph of the laboratory fanbeam scanner.....	48
Figure 3-7. Geometry for multiple crosshole Lamb wave tomography	50
Figure 3-8. Six possible crosshole projections.....	52
Figure 3-9. Lamb wave tomography scanning system with computer plug-in boards in a PC running Linux	53

Figure 3-10. Photograph of the Lamb wave tomography scanner in the laboratory.....	54
Figure 3-11. (a) Parallel projection and (b) multiple crosshole scans of a woven graphite epoxy sample with impact damage.	59
Figure 3-12. Cross-sectional slices of the images in Figure 3-11 at a horizontal mid-line through the flaw	60
Figure 3-13. (a) Parallel projection [6] and (b) multiple crosshole scans of a woven graphite epoxy sample with a 1" through-hole.....	61
Figure 3-14. Cross-sectional slices of the images in Figure 3-13 at a horizontal mid-line through the flaw	62
Figure 3-15. LWT blind study results for plates 1-4	66
Figure 3-16. LWT blind study results for plates 5-8.....	67
Figure 3-17. LWT blind study results for plates 9-12	68
Figure 3-18. LWT blind study results for plates 13-16	69
Figure 3-19. LWT blind study results for plates 17-20	70
Figure 4-1. A pipe segment is shown with a transmitting transducer at A and a receiving transducer at B.....	78
Figure 4-2. Parallel circumferential arrays of transducers can be seen to give a crosshole tomographic geometry when the pipe is mentally "unwrapped."	80
Figure 4-3. Helical ultrasound tomographic (HUT) reconstruction geometry unwrapped	81
Figure 4-4. Data acquisition block diagram for the computer-controlled HUT scanner.....	83
Figure 4-5. A thick-walled steel pipe sample is shown in the HUT scanner	85
Figure 4-6. Four reconstructions are shown for increasing flaw size in the same thick-walled steel sample shown in Figure 4-5	87
Figure 4-7. Velocity scatter plots for the 180 x 180 (32400) recorded waveforms are shown for unflawed pipe samples at (a) 0.5 MHz and (b) 1.35 MHz.....	88
Figure 4-8. Typical waveforms for (a) meridional and (b) helical guided wave modes.	90
Figure 4-9. Velocity scatter plots are shown for flawed samples from Figure 4-6 as extracted by our algorithms versus ray number.....	91

Figure 4-10. Each graph represents a different horizontal cross-section of the tomographic reconstruction shown in Figure 4-6(b)	93
Figure 4-11. A linear array of transducers along the axis of the cylinder can be seen to give a cross-hole tomographic geometry when the pipe is mentally “unwrapped”	94
Figure 4-12. Storage facility for depleted Uranium Hexafluoride (UF ₆).....	96
Figure 4-13. A photograph of a large storage tank mockup scanner in the laboratory ...	97
Figure 4-14. The linear array of transducers used in MUT is mimicked in the laboratory by two transducers attached to linear scanners	99
Figure 4-15. Crosshole geometry for meridional ultrasound tomography (MUT)	101
Figure 4-16. Simulated reconstructions for attenuation MUT data.....	103
Figure 4-17. Guided wave signals of an unflawed pipe sample for the first transmit position and three different receive positions.....	104
Figure 4-18. Amplitude data for a flawed and unflawed sections of the aluminum sample	106
Figure 4-19. MUT reconstructions of an aluminum pipe with an OD = 150 mm and a thickness of 4 mm	107
Figure 5-1. Reconstructed images for ten frequency scans of a thick steel pipe with an irregular 2” x 2” gouge on its ID.....	112
Figure 5-2. Compounded images using three methods (Mean, Π , and RMS) for the ten frequency images in Figure 5-1.	113
Figure 5-3. An example of the thresholding arrival time extraction algorithm	116
Figure 5-4. Arrival times for the first arriving mode in a clean aluminum sample with a thickness of 3.17mm.....	118
Figure 5-5. HUT arrival times from a steel simulated gun barrel sample with an OD of 175 mm and a thickness of 20mm	119
Figure 5-6. The discrete wavelet transform can be seen as the signal being split into its low frequency components – approximations – and its high frequency components – details	124
Figure 5-7. Illustration of coiflet3 mother wavelet and scaling functions.....	127
Figure 5-8. Typical Lamb wave signal demonstrating the difference between the wavelength peak enveloping and wavelet enveloping techniques.....	128

Figure 5-9. Arrival times for the first three peaks of the Lamb wave signals recorded in a tomographic scan of a clean aluminum plate.....	130
Figure 5-10. The envelopes and waveforms of the recorded Lamb wave signals in the clean aluminum plate sample for receiver positions 55-62	131
Figure 5-11. Frequency walk surface plots for the arrival times of the first transmit position in a scan of a clean aluminum plate.....	132
Figure 5-12. Arrival times for the first three peaks of a clean aluminum plate at the first 100 receiver positions	134
Figure 5-13. Arrival time plots for the first three peaks and predicted mode arrivals for a complete tomographic projection	135
Figure 5-14. Sorted arrival times for the first three modes of the clean plate data shown in Figure 5-12	136
Figure 5-15. Arrival time scatter plots for the first mode in projection #1 of a clean aluminum plate	138
Figure 5-16. Arrival time scatter plots for the second mode in projection #1 of a clean aluminum plate	139
Figure 5-17. Arrival time scatter plots for the third mode in projection #1 of a clean aluminum plate	140
Figure 5-18. Reconstructions for a flat bottomed hole in a 3.17mm thick aluminum plate.....	141
Figure 5-19. (a) The first peak's arrival times for the first 100 points of projection #1 for the flat bottomed hole sample (b) Deviations of the arrival times from the polynomial fit.....	143
Figure 5-20. (a) Frequency walk data for the first 100 arrivals of the first-arriving mode in the flat bottomed hole sample. (b) Arrival times for the "flawed" region between receiver positions #55 and #80.....	144
Figure 5-21. (a) Arrival time vs. frequency for the first three envelope peaks in signal #65 (b) Arrival time vs. frequency for signal #598 where only the first frequency missed the first arriving mode.....	146
Figure 5-22. Comparison of frequency walk data for the first arriving mode before sorting and after sorting.....	148
Figure 5-23. Tomographic reconstructions for the first three modes of a 3.17 mm thick plate sample with a 2.2" diameter flat bottomed hole	149

Figure 5-24. Velocity line plots for the first three modes of a vertical line through the center of the flat bottomed hole150

Figure 5-25. Tomographic reconstructions for the first three modes of a 3.17 mm thick plate sample with a 2.2” diameter successively dished-out flat bottomed hole.....152

Figure 5-26. Tomographic reconstructions for the first three modes of a 3.17 mm thick plate sample with a 54mm x 30mm rectangular thinned region with rounded corners.....153

Figure 5-27. Tomographic reconstructions for the first three modes of a 3.17 mm thick plate sample with a 51mm diameter circular flat bottomed hole.....154

Figure 5-28. Comparison of DWFP time extraction before sorting and after sorting....158

Abstract

This dissertation concentrates on the development of two new tomographic techniques that enable wide-area inspection of pipe-like structures. By envisioning a pipe as a plate wrapped around upon itself, the previous Lamb Wave Tomography (LWT) techniques are adapted to cylindrical structures. Helical Ultrasound Tomography (HUT) uses Lamb-like guided wave modes transmitted and received by two circumferential arrays in a single crosshole geometry. Meridional Ultrasound Tomography (MUT) creates the same crosshole geometry with a linear array of transducers along the axis of the cylinder.

However, even though these new scanning geometries are similar to plates, additional complexities arise because they are cylindrical structures. First, because it is a single crosshole geometry, the wave vector coverage is poorer than in the full LWT system. Second, since waves can travel in both directions around the circumference of the pipe, modes can also constructively and destructively interfere with each other. These complexities necessitate improved signal processing algorithms to produce accurate and unambiguous tomographic reconstructions.

Consequently, this work also describes a new algorithm for improving the extraction of multi-mode arrivals from guided wave signals. Previous work has relied solely on the first arriving mode for the time-of-flight measurements. In order to improve the LWT, HUT and MUT systems reconstructions, improved signal processing methods are needed to extract information about the arrival times of the later arriving modes. Because each mode has different through-thickness displacement values, they are sensitive to different types of flaws, and the information gained from the multi-mode analysis improves understanding of the structural integrity of the inspected material. Both tomographic frequency compounding and mode sorting algorithms are introduced. It is also shown that each of these methods improve the reconstructed images both qualitatively and quantitatively.

ULTRASONIC GUIDED WAVE TOMOGRAPHY OF PIPES

Copyright

by

Kevin R. Leonard

2004

Chapter 1

Introduction

1.1 Guided Waves

Ultrasonic guided wave inspection techniques are increasingly being used for nondestructive testing (NDT). In particular, Lamb waves have proven useful for interrogating plate-like structures. Lamb waves are ultrasonic guided waves capable of propagating relatively long distances and they provide an efficient means of detecting disbonds, corrosion and delaminations [1, 2]. In addition, as they propagate, the Lamb waves involve the entire thickness of the plate in a complex elastic deformation and recovery so that they are sensitive to flaws on both of the plate's surfaces and throughout its thickness. This "structural screening" has been a demonstrated benefit of Lamb waves for over a decade, and puts Lamb wave ultrasonography in the same league with rapid full-field techniques such as thermography and photoelasticity. Because of these features, Lamb waves can be used for rapid and cost effective inspection of large-area structures.

Furthermore, they are also able to inspect structures with limited access to their surfaces. For example, underground or insulated piping systems are often restricted in the places where they can be accessed. Lamb waves can be launched and received by

external transducer belts or by a wide variety of contact and non-contact transducer configurations. By generating the guided waves at one access point and receiving the signals at another, the entire structure in between can be inspected. This saves expensive procedures such as removing insulation or temporarily taking the pipe out of service to allow an internal pig to traverse an underground piping system.

These are just a few of the advantages gained by the use of ultrasonic guided waves for the inspection of materials. Another benefit is that Lamb waves can follow the curvature of a structure and allow detection of subsurface flaws with a single-sided pitch-catch or pulse-echo measurement. A more complete, but not exhaustive, list of advantages can be found in Table 1-1.

Traditionally, the use of bulk waves has been preferred over guided waves because their inspection and data processing schemes are well understood and can be reliably implemented by technicians in the field. Bulk wave signals are often less complicated and easier to handle than guided wave signals. Guided wave signals can consist of multiple modes, be affected by mode conversion and may also be dispersive.

Table 1-1. Guided Wave Benefits (adapted from [3])

- Inspection over long distances from a single probe position.
- Sensitivity often greater than that obtained in standard normal beam ultrasonic inspection or other NDT techniques.
- Ability to inspect structures under water, coatings, insulation, multi-layer structures or concrete with excellent sensitivity.
- Potential with multi-mode and frequency Lamb type, Surface or Horizontal Shear waves to detect, locate, classify and size defects.
- Cost effectiveness because of inspection simplicity and speed. Often less than 1/20 the cost of standard normal beam ultrasonic and other inspection techniques.

In addition, many inspection schemes depend on the accurate generation of a particular guided wave mode and this can be difficult to accomplish outside of the laboratory.

Furthermore, there are some scenarios where traditional bulk wave inspections are more advantageous. High resolution C-scans have proven extremely effective in inspecting small custom-built components with irregular geometries. Obviously, guided wave inspection of these types of materials wouldn't be practical. However, for larger structures – aircraft, piping systems, railroad rails, etc. – point-by-point bulk wave measurements can be very time consuming and expensive. Unfortunately, to reduce costs and save time, inspections of randomly chosen positions are often performed. This compromise reduces the effectiveness of the scanning process because the detection of hidden flaws may be missed. Guided ultrasonic waves are an excellent alternative because they have the ability to probe the entire thickness of a wide area from a small number of locations.

However, the physics behind guided waves are more complex than for bulk waves because of the additional boundary conditions that exist in the fundamental problem. This makes guided wave measurements extremely difficult to interpret. In the laboratory, scientists have the ability to sort out subtle variations in complex waveforms by drawing on their extensive background and intuitive understanding of the underlying physics. In the field, technicians responsible for interpreting the NDT measurements do not. Often, more than one propagating mode can exist at different frequencies, and these modes are dispersive and change velocity as material properties change. Modes also scatter and are converted to other modes at discontinuities and flaws. These are the types of complexities that make the analysis of guided waves difficult and yet at the same time so

valuable. If the information that exists in the received guided wave signals can be harnessed, extensive knowledge about a material's structural integrity can be gained.

In order to overcome this fundamental barrier to the more complex guided wave measurements, one solution is to simplify the inspection scheme so that it can be used by a technician without the in-depth knowledge of guided wave physics. Presently, this is how ultrasonic guided waves are being used within the nondestructive evaluation (NDE) community. By generating a single mode that is known to be sensitive to the particular flaw of interest, the inspection scheme can be simplified to one that is very similar to traditional bulk wave pulse-echo measurements. This has allowed guided wave inspections to become feasible for in-the-field use, but it ultimately limits the amount of information gained about the structure being inspected. By limiting the inspection scheme, the wealth of information that can be discovered from the physics of guided wave propagation is lost.

Another way to enable more widespread use of guided waves for material assessment is to develop intelligent algorithms that analyze the complete data set and render it in a way that can be used by the technician responsible for the inspection. In the past, the inability to store, process and analyze large data sets has prevented the use of this method for guided wave inspections. However, the computational capabilities that exist today are dramatically more extensive and far more inexpensive than those that existed just a decade ago. Increased memory capacity, hardware functionality, processor speed, and digital storage space have expanded the number of nondestructive evaluation techniques that are practical for real world testing scenarios because the ability to handle the complexities of guided wave ultrasonics now exists.

The demand for faster and more reliable inspections has also increased recently. This, combined with the complexities of guided wave physics, means that automatic inspection techniques are needed to filter the data and provide accurate recommendations to the technician responsible for the inspection. More advanced signal processing techniques can analyze the data in higher phase spaces and from more dimensions than a human would be able to. Also, the amount of data that can be collected, and needs to be collected, would be overwhelming for an individual to sort through. Robust artificial intelligence is the key to creating inspection algorithms that can rapidly and accurately provide quantitative information about the structural integrity of the inspected material.

1.2 Tomography

Ultrasonic guided wave tomography takes individual Lamb wave measurements and uses them to create an image of the material that unambiguously shows where flaws exist. This allows the intricacies of guided wave physics to be reduced to an easily interpretable representation of the material's structural integrity. Tomography is a method of creating a two-dimensional image of a three-dimensional object's cross-section [4]. It uses either transmission or reflection data recorded from many different orientations to map out a specific property of the object within each cross-section. For example, x-ray computerized tomography (CT) is a mature technology in medical diagnostics that provides two-dimensional cross-sections of the human body. By measuring the attenuation of x-rays at multiple orientations around the body and employing a convolution-type reconstruction technique, these cross-sections can be

stacked and rendered as three-dimensional images which are easier to interpret than typical x-ray shadowgrams.

The key to this technology is the mathematics that takes the attenuation data and transforms it into a two-dimensional image. In the case of Lamb waves, instead of measuring the attenuation due to differences in density, we can record the change in velocity as either the material's thickness or elastic properties change. Then, using the same reconstruction algorithms already refined for the medical industry, a reconstruction of material thickness and integrity can be developed. Much work has been done by Hinders *et. al.* [5-18] to develop and improve multiple ultrasonic guided wave tomography techniques for the nondestructive evaluation of various materials. These techniques are described briefly below and more in-depth in later chapters.

Parallel Projection Tomography

First-generation medical CAT scanners used the parallel projection geometry. In a parallel projection tomography (PPT) measurement, the source receiver pair is linearly scanned over the length of a projection, the source-receiver pair or sample is then rotated and the next projection is scanned. This process is repeated until a specified number of projections have been recorded (see Figure 1-1(a)). Because of the extensive use of PPT in the medical community and the maturity of this technique, the resulting reconstructions can be extremely accurate. However, the method requires either rotation of the sample or rotation of the transmitter-receiver assembly. Both are impractical for in-the-field conditions where large objects are being scanned and there are real-time requirements for the data acquisition process [8, 9, 17].

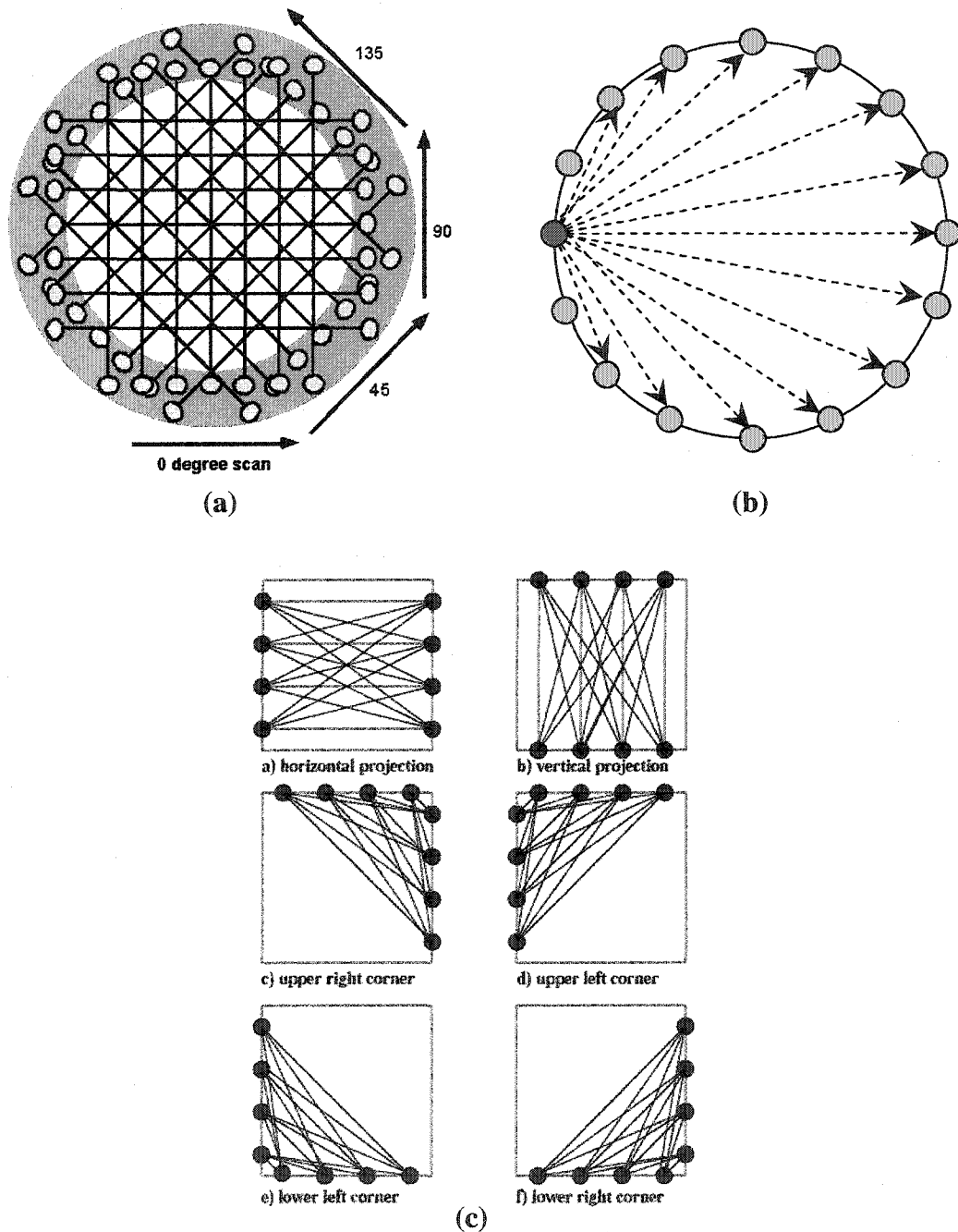


Figure 1-1. Three different Lamb wave tomography geometries. (a) Parallel Projection Tomography (PPT) geometry. Scans are recorded along parallel lines and then these lines are rotated and the scans are repeated. Four rotations at 45 degrees are shown. (b) Fan Beam Tomography geometry. One transducer is used to generate the signal and the others are used to receive the signal. This is repeated until all the transducers have been used as the transmitter. (c) Multiple Crosshole Tomography geometry. As with the fan beam geometry, each transducer pitches and catches the signal in turn. This geometry has the advantage of being able to be implemented via a flexible square perimeter array.

Fan Beam Tomography

Modern CT scanners use a fan beam tomographic geometry to overcome some of the practical problems encountered with PPT. By employing a different reconstruction algorithm, the need for a rotating line of transmitters and receivers can be replaced by a single source and an arc of receivers that rotate in tandem around the object of interest. This can also be implemented as a circular array of pitch-catch elements as shown in Figure 1-1(b). This geometry is more practical for in-the-field ultrasonic nondestructive testing because neither the sample nor the transducers need to be moved. Electronic switching between elements allows all the possible transmit and receive combinations to be realized. However, the area that the fan beam reconstructions are valid for is limited by the effective ray vector coverage. Outside the area where the individual fans overlap, the reconstructed image isn't valid because the ray vector coverage is not dense enough to provide accurate information for the tomographic algorithms. This is not a problem for the medical community because the scanning circle can be large enough that the entire cross-section of the body can fit within the valid center region. For fast and quantitative nondestructive evaluation of plate-like structures this is a serious drawback as large overlaps of many circles would have to occur among sequential scans in order to cover the entire structure. In addition, small errors and uncertainties in the recorded data causes large reconstruction artifacts in the final images [12, 17].

Cross-borehole Tomography

A different and more practical tomographic technique for NDE can be adapted from work that uses seismic waves to image subsurface structures for the detection of

mineral and oil deposits [19, 20]. In these tests it is impossible to access the region of interest from all sides. To overcome this problem, two holes are bored in to the ground and receivers and transmitters are placed in both boreholes and along the ground's surface. The reconstructions are then made using time-of-flight measurements of seismic waves. In this technique, the inverse problem is solved using an Algebraic Reconstruction Technique (ART) instead of the backprojection algorithms used in PPT and fanbeam tomography. The ART uses a crisscross pattern of rays passing through the region of interest to create the reconstruction.

Figure 1-1(c) demonstrates how this technique has been adapted to use Lamb waves for detecting flaws in aircraft structures. Specifically, the Lamb wave tomography (LWT) system uses a square perimeter array of transducers that has the added benefit of being able to access the sample from all four sides, an option not available to seismologists. It has been found that the Sequential Iterative Reconstruction Technique (SIRT), a variation of the ART, overcomes most of the barriers to nondestructive evaluation found in the PPT and fan beam geometries. With the square perimeter array, neither the sample nor the transducers need to be moved or rotated. Furthermore, the entire region within the array produces valid reconstruction data, and small experimental errors in the data do not cause severe artifacts [5-17].

1.3 Helical Ultrasound Tomography (HUT), Meridional Ultrasound Tomography (MUT) and Multi-Mode Analysis

The first part of the work described in this dissertation will concentrate on the development of two new tomographic techniques that enable wide-area inspection of

pipe-like structures. Figure 1-2 shows how the problem is similar to the LWT system. By envisioning a pipe as a plate wrapped around upon itself, we can treat cylindrical structures in the same manner as a plate. Helical Ultrasound Tomography (HUT) uses Lamb-like guided wave modes transmitted and received by two circumferential arrays in a single crosshole geometry. Meridional Ultrasound Tomography (MUT) creates the same crosshole geometry with a linear array of transducers along the axis of the cylinder.

However, even though these new scanning geometries are similar to plates, additional complexities arise because they are cylindrical structures. First, because it is a single crosshole geometry, the wave vector coverage is poorer than in the full LWT system. Second, since waves can travel in both directions around the circumference of the pipe, modes can also constructively and destructively interfere with each other. These complexities necessitate improved signal processing algorithms to produce accurate and unambiguous tomographic reconstructions.

The rest of this dissertation will describe a new algorithm for improving the extraction of multi-mode arrivals from guided wave signals. Previous work has relied solely on the first arriving mode for the time-of-flight measurements. In order to improve the LWT, HUT and MUT systems reconstructions, improved signal processing methods are needed to extract information about the arrival times of the later arriving modes. For each of these techniques, this would enable reconstructions to be made for more than one mode. Because each mode has different through-thickness displacement values, they are sensitive to different types of flaws, and the information gained from the multi-mode analysis can improve our understanding about the structural integrity of the inspected material. For both HUT and MUT, information about multiple modes also leads to the

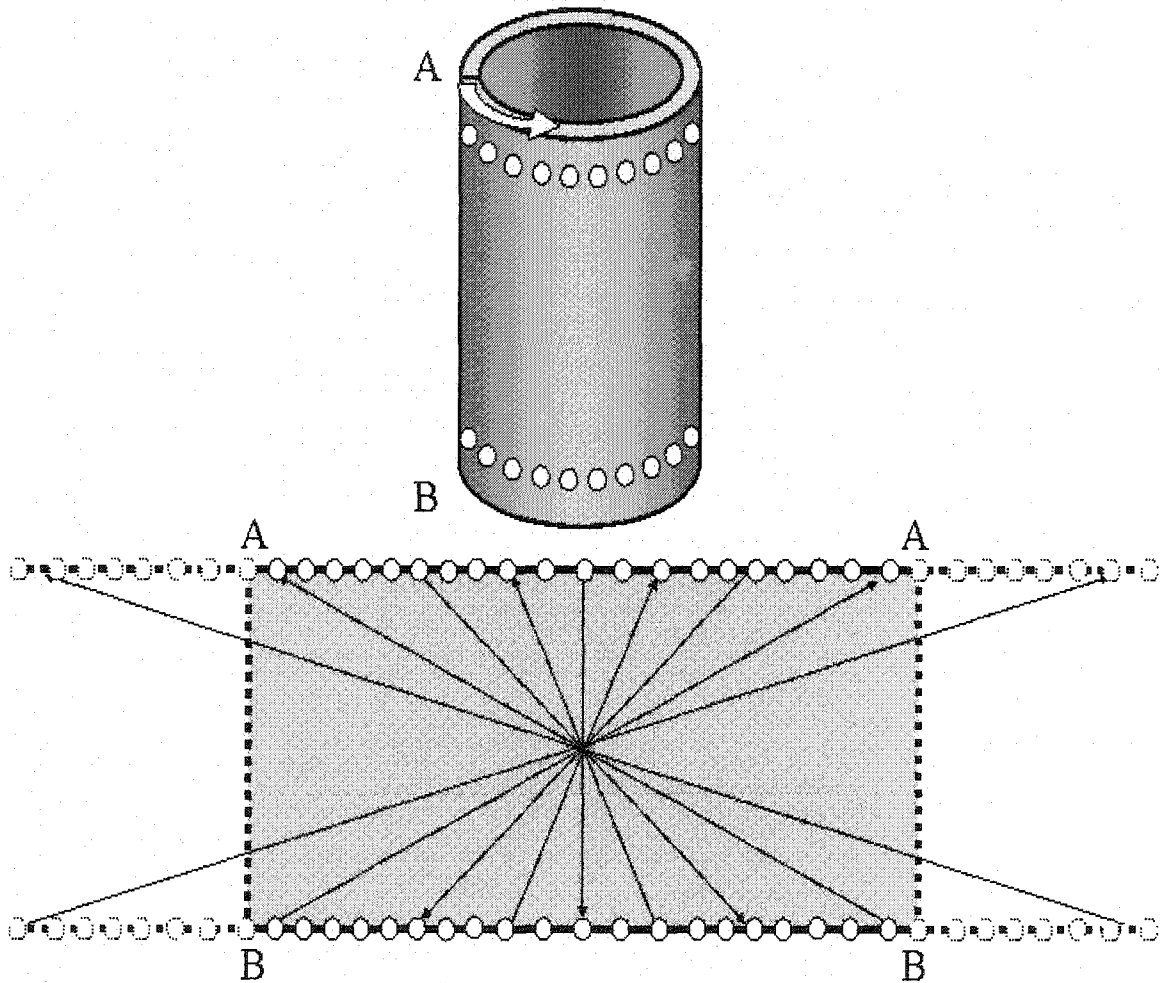


Figure 1-2. Parallel circumferential arrays of transducers can be seen to give a crosshole tomographic geometry when the pipe is mentally “unwrapped.” The two identical parallel circumferential array-belts of transducers wrap around the pipe. Each transducer in one belt transmits helical Lamb waves, which are received by all of the transducers in the other belt. Mentally break the pipe longitudinally along the line AB and then unwrap the pipe to lie flat. The circumferential belts of transducers now lie along the lines AA and BB in the “unwrapped” pipe. Note that the Lamb waves travel along the crisscross rays shown above. Because the helical waves can wrap around the pipe more than once, we can consider the “extra” regions to the left and right of AB. These longer ray paths can be used to give better tomographic reconstructions because they pass through the sample from additional angles.

improvement of the reconstructions by increasing the wave vector density of the scanning geometry. Multi-mode analysis will allow us to extract time-of-flight information about modes that travel more than once around the pipe before reaching the receiving transducer. These steeper helical paths allow the ultrasonic energy to interact with the material from different directions and angles and thus improve the reconstructed images.

This dissertation is organized as follows. Chapter 2 presents a review of guided wave theory and ultrasonic inspection techniques, particularly as they relate to guided wave tomography. In Chapter 3, guided wave tomography is introduced and the results of a blind study with the LWT system are shown in addition to reconstructions of composite samples. The development and proof-of-concept for both HUT and MUT is demonstrated in Chapter 4. Chapter 5 discusses two new techniques – tomographic frequency compounding and multi-mode sorting – for improving guided wave tomographic reconstructions. Finally, Chapter 6 summarizes the findings in this work and discusses recommendations for future research.

Chapter 2

Fundamentals

2.1 Lamb Waves

Elastic waves in extended solids are either longitudinal or transverse, characterized by compressional or shear vibrations respectively. Boundaries cause mode coupling and intermixing of these waves to the extent that it's often no longer useful to try to distinguish them. Plate-like structures have two boundaries and we refer to the intermixed propagating compressional and shear vibrations as Lamb waves. Lamb waves come in two families of modes: symmetric and antisymmetric. The lowest-order symmetric mode is a thickness stretching and contraction while the lowest-order antisymmetric mode is a constant-thickness flexing (Figure 2-1). Higher-order modes have increasingly complex through-thickness displacements. In addition, each mode has its own characteristic dispersion properties.

Many previous authors have discussed the fundamental solutions for Lamb waves in various texts [1, 2, 21-24]. The solution for ultrasonic guided waves in an isotropic plate with traction-free boundaries is presented below. The plate (Figure 2-2) is assumed to be infinite in the x and z directions and the boundaries at $y = \pm b$ are traction-free.

The fundamental equation of motion for an isotropic elastic solid in Cartesian tensor notation is:

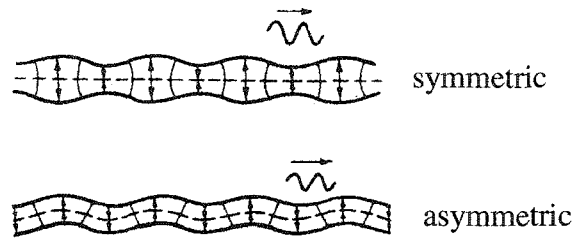


Figure 2-1. The two families of Lamb wave modes generated in a thin plate [1].

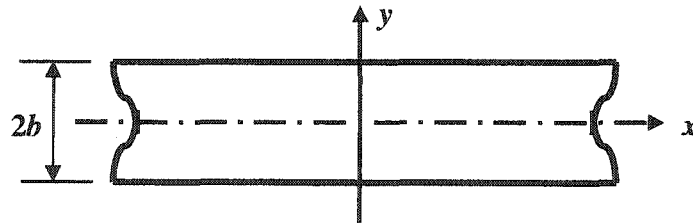


Figure 2-2. Coordinate system for a plate of thickness $2b$.

$$\rho \partial_t^2 u_i - \partial_j \sigma_{ij} = 0$$

$$\text{with, } \varepsilon_{kl} = \frac{1}{2} (\partial_k u_l + \partial_l u_k) \quad (2-1)$$

$$\text{and } \sigma_{ij} = C_{ijkl} \varepsilon_{kl}.$$

Here $u(x_1, x_2, x_3, t)$ is the displacement vector of a point within the material and σ_{ij} is the stress tensor at a point. The stress tensor is symmetric and thus $\sigma_{ij} = \sigma_{ji}$. ε_{kl} is the strain tensor at a point and is directly proportional to the stress tensor. C_{ijkl} is a fourth rank tensor of proportionality constants between the two. For an isotropic material such as aluminum, it can be defined as:

$$C_{ijkl} = \delta_{ij} \delta_{kl} \lambda + (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}) \mu. \quad (2-2)$$

The Lamé parameters, λ and μ , are the elastic constants for a given material.

The governing wave equation for elastic waves is derived by substituting the strain-displacement relation into the stress-strain relation and then substituting that result into Eqn. (2-1). The resulting wave equation is called Navier's equation and in both Cartesian tensor and vector notation it is as follows:

$$\rho \partial_t^2 u_i - \mu \partial^2 u_i - (\lambda + \mu) \partial_i \partial_j u_j = 0 \quad (2-3)$$

$$\rho \partial_t^2 \vec{u} - \mu \nabla^2 \vec{u} - (\lambda + \mu) \vec{\nabla} (\vec{\nabla} \cdot \vec{u}) = 0.$$

At this point, either a solution method based on potentials or partial waves may be used. The method of potentials is more widely used in the literature, but is limited to isotropic materials, while the partial wave method is more general. The method of potentials will be used in this work because it is conceptually simpler and can more succinctly illustrate

the characteristic properties of Lamb waves. However, both solution methods come to the same conclusions about the behavior of guided ultrasonic waves in plates.

Navier's equation, Eqn. (2-3), can be solved by applying a Helmholtz decomposition. Clebsh's theorem states that any vector field can be broken into a longitudinal part, which has no curl, and a transverse part, which has no divergence. In this case, Φ and \mathbf{H} are introduced so that:

$$\vec{u} = \vec{u}_L + \vec{u}_T = \nabla\Phi + \vec{\nabla} \times \mathbf{H} \quad \text{w here, } \nabla \cdot \mathbf{H} = 0. \quad (2-4)$$

Furthermore, by suppressing the $e^{-i\alpha t}$ harmonic time variation of Navier's equation, we find that:

$$(K^2 + \nabla^2)\vec{u} - \left(1 - \frac{K^2}{k^2}\right)\vec{\nabla}(\vec{\nabla} \cdot \vec{u}) = 0$$

$$\text{where, } K = \frac{\omega}{c_T}, \quad k = \frac{\omega}{c_L}, \quad (2-5)$$

$$c_L^2 = \frac{(\lambda + 2\mu)}{\rho}, \quad \text{and} \quad c_T^2 = \frac{\mu}{\rho}.$$

K and k are the wave numbers for the transverse and longitudinal waves respectively, and c_L and c_T are the wave speeds for the longitudinal and transverse waves for a given material. If we take the Helmholtz decomposition and substitute it into Eqn. (2.5), we get:

$$(\nabla^2 + k^2)\Phi = 0 \quad \text{and} \quad (\nabla^2 + K^2)H_p = 0, \quad (2-6)$$

where $\mathbf{p} = (x, y, z)$. Solving Eqn. (2-6) by separation of variables, the following plane wave solutions are derived:

$$\Phi = (A \cos \alpha y + B \sin \alpha y) e^{i(\xi x - \omega t)},$$

$$\begin{aligned}
H_x &= (C \cos \beta y + D \sin \beta y) e^{i(\xi x - \omega t)}, \\
H_y &= (E \cos \beta y + F \sin \beta y) e^{i(\xi x - \omega t)}, \\
H_z &= (G \cos \beta y + H \sin \beta y) e^{i(\xi x - \omega t)},
\end{aligned} \tag{2-7}$$

$$\text{where, } \alpha^2 = \frac{\omega^2}{c_L^2} - \xi^2 \quad \text{and} \quad \beta^2 = \frac{\omega^2}{c_T^2} - \xi^2.$$

In addition, by excluding variations with respect to z , due to the geometry of the system (see Figure 2-2), we can write the displacements as:

$$u_x = \frac{\partial \Phi}{\partial x} + \frac{\partial H_z}{\partial y}, \quad u_y = \frac{\partial \Phi}{\partial y} - \frac{\partial H_z}{\partial x}, \quad u_z = -\frac{\partial H_x}{\partial y} + \frac{\partial H_y}{\partial x}. \tag{2-8}$$

Using the solutions found in (2-7) we get,

$$\begin{aligned}
u_x &= [i\xi(A \cos \alpha y + B \sin \alpha y) + \beta(-G \sin \beta y + H \cos \beta y)] e^{i(\xi x - \omega t)}, \\
u_y &= [\alpha(-A \sin \alpha y + B \cos \alpha y) - i\xi(G \cos \beta y + H \sin \beta y)] e^{i(\xi x - \omega t)}, \\
u_z &= [\beta(C \sin \beta y - D \cos \beta y) + i\xi(E \cos \beta y + F \sin \beta y)] e^{i(\xi x - \omega t)}.
\end{aligned} \tag{2-9}$$

In order to solve for the 8 unknown coefficients, $A-H$, we need to apply the boundary conditions. For the plate shown in Figure 2-2, with thickness $2b$, infinite extent in the xz -plane, and traction-free boundaries, the boundary conditions of the system are:

$$\sigma_{yy} = \sigma_{xy} = \sigma_{zy} = 0 \quad \text{at } y = \pm b \tag{2-10}$$

Finally, as defined above, the relevant stress components are:

$$\sigma_{yy} = (\lambda + 2\mu) \frac{\partial u_y}{\partial y} + \lambda \frac{\partial u_x}{\partial x}, \quad \sigma_{xy} = \mu \left(\frac{\partial u_y}{\partial x} + \frac{\partial u_x}{\partial y} \right), \quad \sigma_{yz} = \mu \frac{\partial u_z}{\partial y} \tag{2-11}$$

These boundary conditions only give us six separate equations to solve for the eight unknowns above. Two additional equations can be obtained from the gauge, $\nabla \cdot \mathbf{H} = 0$, used in the Helmholtz decomposition. This gives us:

$$\frac{\partial H_x}{\partial x} + \frac{\partial H_y}{\partial y} = 0 \quad \text{at } y = \pm b. \quad (2-12)$$

The resulting system of equations is then:

$$\begin{aligned} & \{(\lambda + 2\mu)\alpha^2 + \lambda\xi^2\} \{A \cos \alpha b + B \sin \alpha b\} + 2i\mu\xi\beta \{-G \sin \beta b + H \cos \beta b\} = 0 \\ & \{(\lambda + 2\mu)\alpha^2 + \lambda\xi^2\} \{A \cos \alpha b - B \sin \alpha b\} + 2i\mu\xi\beta \{G \sin \beta b + H \cos \beta b\} = 0 \\ & \beta^2 \{C \cos \beta b + D \sin \beta b\} + i\xi\beta \{-E \sin \beta b + F \cos \beta b\} = 0 \\ & \beta^2 \{C \cos \beta b - D \sin \beta b\} + i\xi\beta \{E \sin \beta b + F \cos \beta b\} = 0 \\ & 2i\xi\alpha \{-A \sin \alpha b + B \cos \alpha b\} + (\xi^2 - \beta^2) \{G \cos \beta b + H \sin \beta b\} = 0 \\ & 2i\xi\alpha \{A \sin \alpha b + B \cos \alpha b\} + (\xi^2 - \beta^2) \{G \cos \beta b - H \sin \beta b\} = 0 \\ & \beta \{-E \sin \beta b + F \cos \beta b\} + i\xi \{C \cos \beta b + D \sin \beta b\} = 0 \\ & \beta \{E \sin \beta b + F \cos \beta b\} + i\xi \{C \cos \beta b - D \sin \beta b\} = 0 \end{aligned} \quad (2-13)$$

This is a system of eight homogeneous equations. In order to solve for the eight unknown coefficients, $A-H$, we set the determinant of the coefficient matrix equal to zero. This condition provides four separate solutions because the larger determinant can be rearranged into four separate sub-determinants that disappear independently. This yields:

- Solution I: $A, B, D, E, G, H = 0$ and $C, F \neq 0$,
- Solution II: $A, B, C, F, G, H = 0$ and $D, E \neq 0$,
- Solution III: $B, C, D, E, F, G = 0$ and $A, H \neq 0$,
- Solution IV: $A, C, D, E, F, H = 0$ and $B, G \neq 0$.

Solutions I and II result in antisymmetric and symmetric SH modes. The SH modes are shear horizontal displacements that occur in the z -direction. They are not of interest because Lamb waves are two-dimensional waves with displacements in the xy -plane.

Solutions III and IV are commonly referred to as the Rayleigh-Lamb equations. They are:

Solution III:

$$\begin{vmatrix} \{(\lambda + 2\mu)\alpha^2 + \lambda\xi^2\} \cos \alpha b & 2i\mu\xi\beta \cos \beta b \\ -2i\xi\alpha \sin \alpha b & (\xi^2 - \beta^2) \sin \beta b \end{vmatrix} = 0$$

(2-14)

$$\begin{aligned} u_x &= (i\xi A \cos \alpha y + \beta H \cos \beta y) e^{i(\xi x - \omega t)} \\ u_y &= -(\alpha A \sin \alpha y + \xi H \sin \beta y) e^{i(\xi x - \omega t)} \\ u_z &= 0 \end{aligned}$$

Solution IV:

$$\begin{vmatrix} (\xi^2 - \beta^2) \cos \beta b & 2i\xi\alpha \cos \alpha b \\ 2i\mu\xi\beta \sin \beta b & \{(\lambda + 2\mu)\alpha^2 + \lambda\xi^2\} \sin \alpha b \end{vmatrix} = 0$$

(2-15)

$$\begin{aligned} u_x &= (i\xi B \sin \alpha y - \beta G \sin \beta y) e^{i(\xi x - \omega t)} \\ u_y &= (\alpha B \cos \alpha y - i\xi G \cos \beta y) e^{i(\xi x - \omega t)} \\ u_z &= 0 \end{aligned}$$

The dispersion equations for the Lamb waves can be obtained by expanding the sub-determinants found in solutions III and IV. This yields the following relationships for the symmetric and antisymmetric modes:

Symmetric Modes:

Asymmetric Modes:

$$\frac{\tan \beta b}{\tan \alpha b} = -\frac{4\alpha\beta\xi^2}{(\xi^2 - \beta^2)^2} \qquad \frac{\tan \beta b}{\tan \alpha b} = -\frac{(\xi^2 - \beta^2)^2}{4\alpha\beta\xi^2} \qquad (2-16)$$

These two equations give rise to the velocity dispersion relationships for Lamb waves. Figure 2-3 shows the numerical solutions for the dispersion relations of an aluminum plate.

It is interesting to note from this solution that Lamb waves are generated in two different types of modes. Both the symmetric and antisymmetric families have an infinite number of possible modes. The number of propagating modes present at a given frequency-thickness product depends on both the cutoff frequencies for the individual modes and the way in which the Lamb waves are generated (see Section 2.2.2). The different modes, A_n and S_n , are named in order of their cutoff frequencies, where $n = 0, 1, 2, \dots$. Except for the fundamental modes A_0 and S_0 , all the modes have a cutoff frequency defined by:

$$\omega_c = n\pi \frac{c_t}{d}; \quad \omega_c = \pi \left(n + \frac{1}{2} \right) \frac{c_l^2}{c_t d} \quad \text{for symmetric modes} \quad (2-17)$$

$$\omega_c = \pi \left(n + \frac{1}{2} \right) \frac{c_t}{d}; \quad \omega_c = n\pi \frac{c_l^2}{c_t d} \quad \text{for antisymmetric modes.} \quad (2-18)$$

In addition to their unique dispersion relationships, each mode also has different displacement characteristics. Figure 2-4 displays both the out-of-plane and in-plane displacements for the two lowest order symmetric modes. These different displacement characteristics make individual modes sensitive to different types and locations of flaws.

2.2 Lamb Wave Experimental Considerations

There are many ways to experimentally generate Lamb waves in plate-like materials. It is advantageous to discuss some of these methods as they relate to

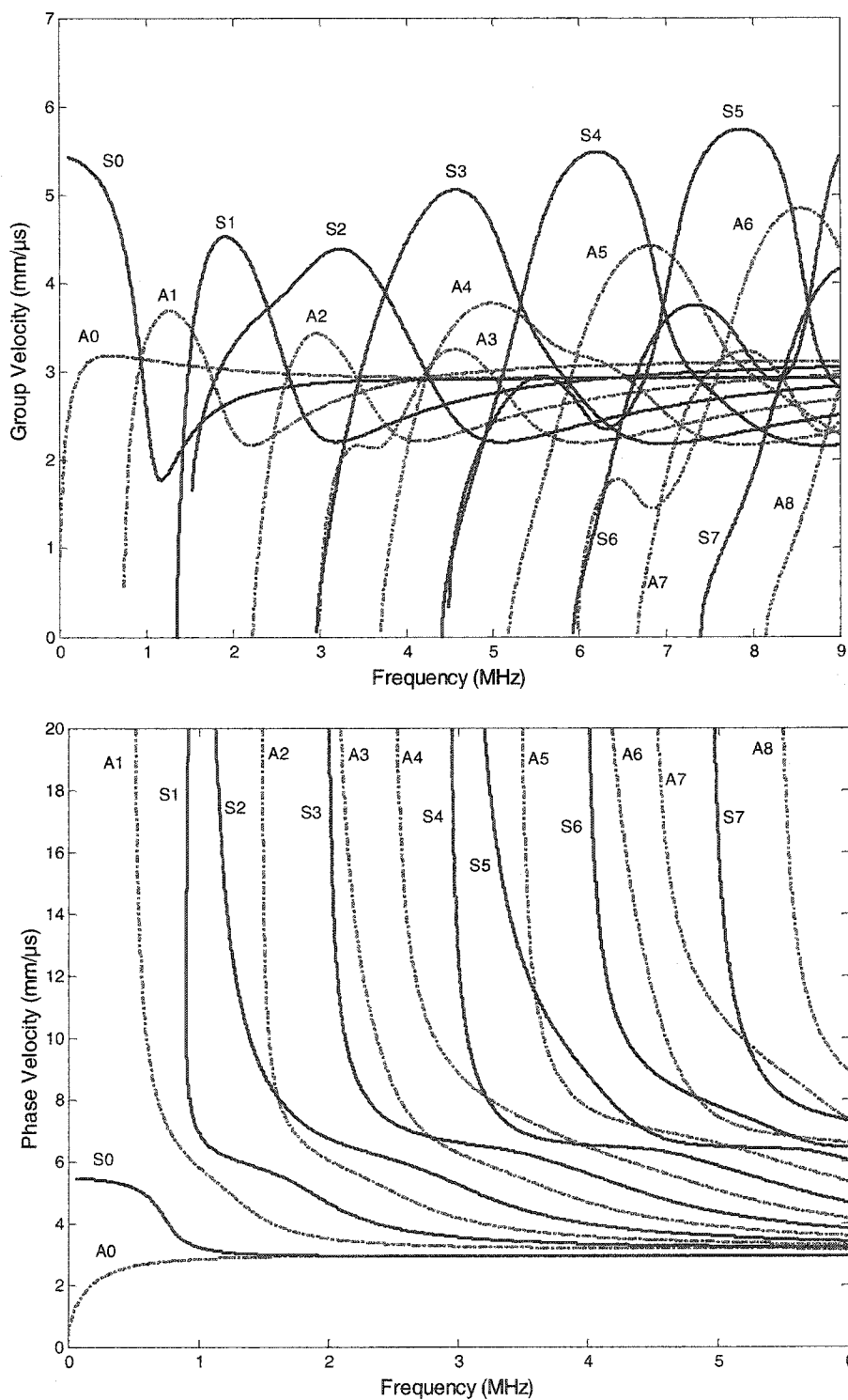


Figure 2-3. Phase and group velocity dispersion curves for a 3 mm thick aluminum plate, where $c_L = 6.32 \text{ mm}/\mu\text{s}$ and $c_T = 3.13 \text{ mm}/\mu\text{s}$. The solid lines represent the symmetric mode family and the dashed-dot lines represent the antisymmetric family. Notice that as the frequency increases, so does the number of generated Lamb wave modes. The modes are numbered by the order of their cutoff frequency.

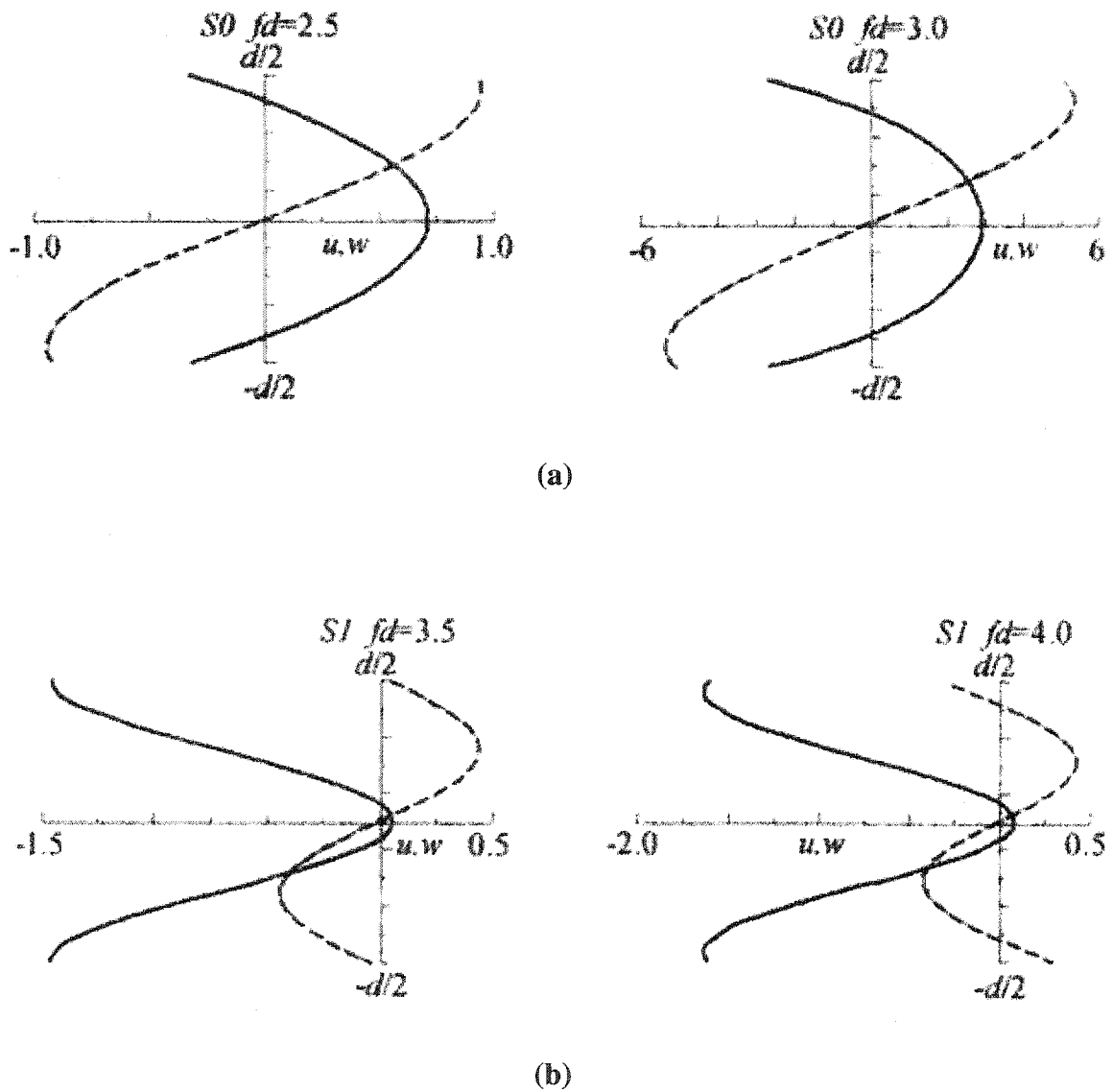


Figure 2-4. Displacement curves for the two lowest order symmetric modes in an aluminum plate at two frequency-thickness products. (a) S0 (b) S1. Dashed lines: out-of-plane displacement; Solid lines: in-plane displacement [2].

nondestructive evaluation. Any carefully designed inspection scheme needs to take into consideration the type of signal desired (broadband or narrowband, number and type of modes desired, etc.) and any physical constraints that the specimen or the inspection scheme place on the type of transducer used (i.e. – size of transducer and coupling).

2.2.1 Coupling

The effectiveness of generating ultrasonic guided waves in a material through surface perturbations is dependent upon coupling. For example, the perfect situation is where the acoustic impedances of the transducer and the material are the same. In this case, all of the ultrasonic energy will be transferred from the transducer to the material. In less than ideal cases, a medium needs to be used that reduces the impedance mismatch between the plate and transducer as much as possible. It turns out that water is a good coupling agent as it does a relatively good job of matching impedances and is extremely cost effective.

Coupling is an issue with contact transducers, but there are also non-contact methods of generating ultrasonic guided waves. Air-coupled transducers use lower frequency ultrasound that can propagate short distances in air without attenuating too appreciably. As technologies improve, these types of transducers are becoming easier to make. Because there is such a large impedance gap between gases and solids, sophisticated construction is needed to overcome the signal loss inherent in transmission and reception of the air-coupled signals. Air-coupled transducers have been used for the nondestructive testing of various structures including railroad rails and metal plates [25, 26].

Laser ultrasound is another non-contact method of generating Lamb waves [27]. By Q-switching or chopping the laser beam, the surface of the material is periodically heated. This heating and cooling of the surface produces thermal waves in the material that penetrate to a certain depth. The thermal waves in turn cause the affected region to expand and contract, thus generating elastic waves in the material. Lasers can also be used to detect the ultrasonic signal through the use of optical interferometers. Certain drawbacks of the use of laser generated ultrasound include damage to the sample through ablation and low efficiency due to how the laser's energy is transferred into the acoustic signal.

Finally, Electro-Magnetic Acoustic Transducers (EMATs) are another fully non-contact method used to generate Lamb waves [28]. EMATs can be used to generate ultrasonic guided waves in ferrous metals. They consist of a large permanent magnet and an electrical coil. The EMAT system induces eddy currents in the test material and because of the presence of the permanent magnet, a Lorentz force is created that generates acoustic waves in the conductor. By varying the orientation of the magnet and the coil, longitudinal and shear displacements can be selectively generated.

2.2.2 Transduction

The most straightforward way to generate Lamb wave signals uses either normal or shear vibrations on the surface of the plate. The number and type of modes generated depend on the material properties and the frequency spectrum of the input signal. The degree to which individual modes are excited also varies and depends on certain material and transducer properties [29, 30]. This can cause the generated signals to include strong

unwanted modes, while the modes of interest are weak. In order to generate a specific mode, specialized transducers are needed. Variable incidence and comb transducers selectively generate specific modes by controlling the phase velocity and wavelength of the input signal respectively.

Angle Block Transducers

Angle block transducers allow individual modes to be generated in the material by varying the incidence angle of the input signal. By using Snell's Law, the phase velocity of the generated Lamb wave can be controlled (Figure 2-5). Assuming an infinite plane wave source at incident angle, θ , the resulting phase velocity (V_{ph}) can be calculated using the following equation [1]:

$$V_{ph} = \frac{V_i}{\sin \theta}, \quad (2-19)$$

where V_i is the longitudinal velocity of the wedge material. By controlling V_{ph} , the frequency of the carrier signal determines which mode is generated. However, in reality, the source is finite and can not generate an infinite plane wave. This causes the phase velocity spectrum to broaden, and allows multiple angles at which a particular mode can be generated. Rajana *et. al.* have shown both theoretically and experimentally that the mode's excited amplitude is continuously dependent on the incident angle [31, 32].

Comb Transducers

Another way to generate specific Lamb wave modes is with a periodic linear array of transducers also known as a comb transducer [1, 33-35]. Comb transducers consist of a group of equally spaced elements that vibrate in phase with each other at a

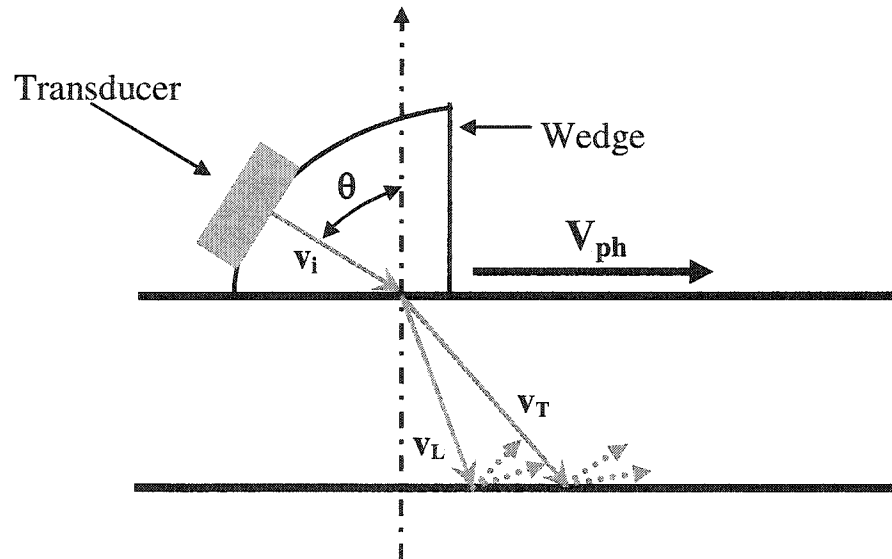


Figure 2-5. Diagram of a variable incidence angle block transducer. The incident angle, through Snell's Law, determines the resulting Lamb wave phase velocity in the given material. It also demonstrates that at the surface boundaries an incident compressional or shear wave will mode convert into both a reflected shear and compressional wave. The angle block transducer can be used to selectively generate specific Lamb wave modes.

given frequency. This generates Lamb waves with a wavelength λ equal to the spacing between the comb elements and at the given frequency f . In addition, it is possible to control the direction that the resulting wave propagates by introducing a time-delay to certain elements. By controlling the phase of the individual elements, through constructive and destructive interference, the guided wave can be steered [36, 37]. Similarly, interdigital PVDF transducers are another type of “comb” transducer. These transducers consist of a layer of piezoelectric polymer (PVDF) bonded to the bottom of a flexible PCB with comb shaped electrodes. They control the wavelength of the generated mode by specifying the spacing between electrodes instead of the individual transducers [38].

2.2.3 Bandwidth

Finally, the dispersive characteristics of Lamb waves make it important to consider the bandwidth of the input signal. For a given toneburst, as the signal propagates, the different frequencies will travel at different velocities. This causes the signal to spread over time, and in highly dispersive regions the shape of the signal will drastically change. By convolving the plane wave solution with the Fourier transform of the input signal, we can see how the dispersive nature of the individual modes spread the wave packet [39]:

$$u(x, t) = 2 \operatorname{Re} \left\{ \int_0^{\infty} U(\omega) e^{i(k(\omega)x - \omega t)} d\omega \right\} . \quad (2-20)$$

To illustrate this concept, Figure 2-6 shows how the S_0 mode envelope spreads as its propagation distance increases for a 5-cycle square-windowed toneburst at 0.7 MHz. In

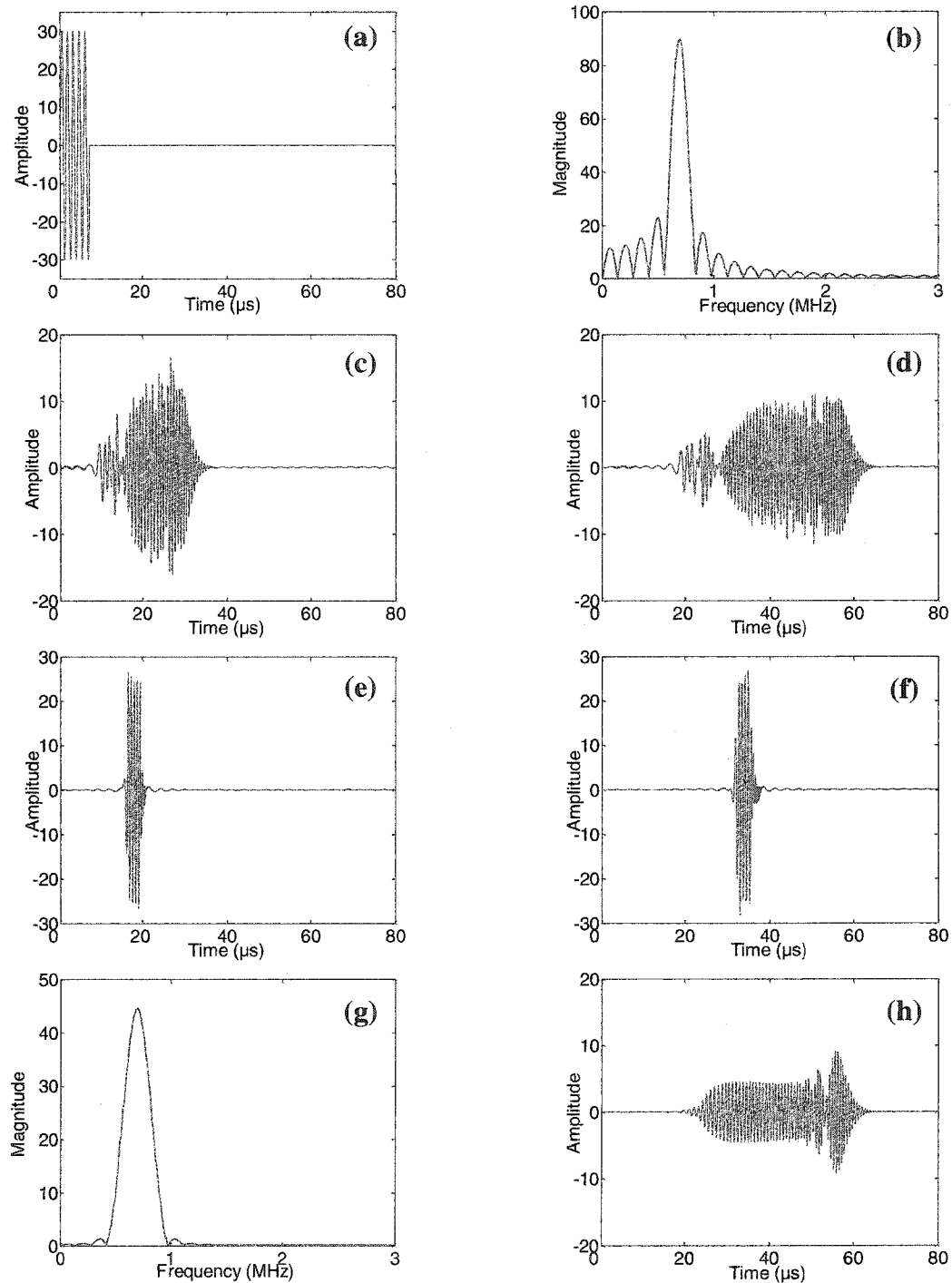


Figure 2-6. Plots demonstrating the effects of dispersion due to an input signal's bandwidth for a highly dispersive mode, S_0 , and a non-dispersive mode, A_0 (refer to Figure 2-3). (a) Simulated 0.7 MHz, 5-cycle toneburst input signal for a 3 mm thick aluminum plate. (b) FFT of simulated input signal. (c) Simulated S_0 mode after propagating 100 mm. (d) S_0 mode after traveling 200 mm. (e) Simulated A_0 mode, $d = 100$ mm. (f) A_0 mode, $d = 200$ mm. (g) FFT of input signal in (a) with Hanning window applied. (h) Simulated S_0 signal after 200 mm with Hanning windowed input signal.

contrast, we see how a non-dispersive mode, A_0 , retains its mode shape as it propagates. By applying a Hanning window to the input signal, we can reduce the effective bandwidth of the signal and reduce some of the dispersion effects.

The spread of the wave packet is important because it affects how the signals can be analyzed. There is a tradeoff between the input signal's pulse width and the bandwidth of the signal. In order to diminish the effects of packet spreading, the bandwidth of the signal can be reduced by lengthening the toneburst. However, as the toneburst is lengthened to reduce packet spreading, mode arrivals begin to overlap because of their length, thus creating the very problem that was trying to be eliminated.

When generating only a single mode, these effects can be less important. But, in the presence of defects, modes scatter and mode convert. The spreading of a dispersive mode can make it difficult to detect the presence of an additional mode or the scattered signal from a flaw. For multi-mode analysis, the problem becomes even more important as advanced artificial intelligence is needed to sort out the interfering modes from one another.

Some theoretical work and experimental verification has been done on this problem for single mode signals. By taking the received time-domain signal and assuming it is received at $x = 0$, Wilcox *et. al* [39, 40] and Sicard *et. al.* [41] have shown that by applying a backprojection method in the negative x -direction, the dispersed time-domain wave packet can be transformed to a non-dispersive signal arriving at a distance corresponding to the center group velocity of the initial toneburst:

$$h(x) = u(-x', 0) = \int G(\omega) e^{-ik(\omega)x} d\omega. \quad (2-21)$$

Both authors also showed experimental results for single mode pulse-echo signals. In addition, Wilcox *et. al.* showed experimentally how slower modes were affected by the dispersion compensation algorithm. However, these signal processing algorithms assume that only one mode is present and are not applicable to multi-mode signals in general.

Other time reversal techniques have also been applied to this problem. For example, Alleyne *et. al.* [42] have shown that if the wave packet for a recorded Lamb wave mode signal that has propagated a specified distance is reversed in the time domain, and then used as the input signal, the received signal at the original propagation distance will be the same simple shape as the initial toneburst. While this technique demonstrates the dispersive nature of Lamb wave modes, it is not useful for practical inspection of materials because it does not provide any experimental advantage for locating flaws.

2.2.4 Discussion

Guided wave tomography measurements require the signal to be omni-directional, relatively narrow-banded, capable of traveling over long distances, and robust enough to handle actual in-the-field testing environments. For these reasons, normal incidence longitudinal transducers provide the best means for generating the Lamb wave signals to be used for tomographic inspection. While angle blocks and comb transducers are able to generate specific individual modes, they are not omni-directional. Laser ultrasound is too broadbanded and PVDF transducers do not generate strong enough signals to propagate long distances and through thick materials.

2.3 NDE Applications

Lamb waves, and more generally ultrasonic guided waves, have been used to solve many NDE problems. From the inspection of aerospace structures to storage tanks and railroad rails, guided waves have unique properties that enable sophisticated inspection schemes. This section describes a few of these applications and demonstrates that guided waves can be used effectively to detect structural flaws in a variety of geometries and materials.

Corrosion detection with Lamb waves, in its simplest form, consists of monitoring the change in arrival times of the Lamb wave modes. At a given frequency-thickness product, a small change in thickness due to corrosion will cause the individual modes to speed up or slow down depending on their dispersive characteristics (Figure 2-3). Many authors, including Rose et. al. [43], Jenot et. al. [44], Sun and Johnson [45, 46], and Alleyne and Cawley [47] have performed studies using changes in arrival times to detect corrosion in plates and pipes. Another method used for detecting corrosion generates specific modes near their cutoff frequencies and looks for the mode to disappear in the presence of a defect. Silva et. al. [48] demonstrate this technique on aircraft aluminum structures using laser-generated ultrasound and wavelet transform signal analysis.

In addition to corrosion, guided waves can be used to detect discontinuities such as cracks. Much theoretical and experimental work has been done to study how Lamb waves interact with these types of flaws. Rose et. al. [49, 50] used a boundary element method (BEM) to explore the use of reflection and transmission coefficients of incident and mode converted signals to characterize and size crack and corrosion type flaws. Chang and Mal [51] used a finite element method (FEM) to analyze the effect of a crack

at the edge of a rivet hole on the scattered signal. They tested their model against experimental data and concluded that the crack's presence could be seen in the frequency spectrum of the scattered signal. Many other authors, including [52-56], have performed similar theoretical studies on different discontinuities in plates and pipes.

Lamb waves can also be used to detect flaws in composites and joined materials. Sun and Johnson [46] used time-of-flight and amplitude measurements to inspect for disbonds in doublers and lap joints with and without rivets. Hayashi et. al. [57] used a numerical technique similar to FEM and BEM to theoretically explore multiple reflections at the boundaries of a delamination in a multi-layer composite plate. Furthermore, Seale et. al. [58, 59] explored the use of Lamb waves to measure fatigue in composite samples.

Other types of ultrasonic guided waves, such as surface waves, have also been used in the nondestructive evaluation of materials. For example, Rayleigh surface waves can be used to explore railroad rails for cracks. Bray et. al. [60-62] and Grewal [63] have shown that in used railroad rail, a cold worked layer is formed that allows a higher-order Rayleigh surface wave, also known as a Sezawa wave, to propagate with its energy just under the surface. They have experimentally and theoretically shown that the Sezawa wave is slightly faster than the fundamental Rayleigh wave. The Sezawa wave, unlike the Rayleigh wave, is not affected by the rough surface properties but does interact strongly with surface breaking cracks.

These are just a few examples of how Lamb waves, and ultrasonic guided waves more generally, have been used to inspect various materials. Many others exist, including pipe inspection (discussed further in Chapter 4), finding thermal damage in

composites [64], use in acoustic emission [65], inspection of rockbolts and grouted tendons [66, 67], etc. An extensive review of Lamb waves and their applications can be found in [5]. Rose has also published a comprehensive review of ultrasonic guided wave nondestructive evaluation in [3].

2.4 Tomography

Ultrasonic guided waves have great potential for nondestructive evaluation. However, their complexities often make their use difficult in practical inspection schemes. Tomography enables these intricacies to be represented in a format that is easily interpreted by a layman. Much like CT scanners in the medical community that create high resolution images of cross-sections of the human body, tomographic algorithms can be used to create images of material thickness and integrity. Instead of measuring x-ray attenuation, slight changes in the arrival time of the Lamb wave modes can be used. As previous research has shown, various flaws can be detected by monitoring the change in velocity of the ultrasonic guided wave modes for a given path length. Coupled with tomographic reconstruction techniques, these measurements provide an excellent method for converting the complicated physics into a readily interpretable quantitative thickness map of the material. The resulting images then enable quick and accurate detection of different types of flaws in multiple plate-like structures. Ultimately, the goal is to considerably extend the usefulness of Lamb waves by adding detailed and straightforward quantitative measurement capability without sacrificing the rapid large-area capabilities inherent in the method.

Chapter 3

Lamb Wave Tomography (LWT)

Ultrasonic nondestructive evaluation of plate-like structures is of interest in many areas including the inspection of military and civilian aircraft. On April 28, 1988 an Aloha Airline's Boeing 737 experienced a structural failure and explosive decompression at 24,000 feet [68]. The aging plane lost an entire fuselage section because of cracks emanating from rivet holes in its outer skin. Amazingly, the incident only killed one person and injured sixty-five others. As the worldwide aviation fleet continues to age and planes are expected to remain in service even after their expected lifetime has expired, the Aloha Airlines accident remains as an example to the aerospace industry that improved aircraft inspection techniques are needed. Methods for accurately detecting the presence of structural flaws that compromise airworthiness also become increasingly necessary as novel material systems are incorporated into primary structural elements. At the same time, there is an increasing demand that inspections become more effective and efficient. This can only be done if the interpretation of NDE sensor data is automated to some degree.

Lamb wave tomography (LWT) is a viable solution to this problem and can be implemented in a cost efficient, real-time and highly effective manner. Tomography allows us to extend the usefulness of Lamb waves for structural inspections. In contrast

to typical point-by-point scans or pulse-echo guided wave schemes, if Lamb wave measurements are made for a number of relative pitch-catch transducer positions, also known as projections, then an image of a large region can be reconstructed tomographically. The resulting image gives an easily interpretable quantitative map of the parameter of interest. For example, one such parameter could be thickness loss due to corrosion.

Key to this work, and to any practical Lamb wave tomography system, is a suite of robust and reliable algorithms that reads in the digitized Lamb waveforms and automatically interprets them. Unlike in bulk wave ultrasonics where gating and peak-detection schemes are usually adequate, Lamb wave detection schemes must be more complicated. With guided waves, much more sophisticated signal processing is required in order to identify the various Lamb wave modes in the signals and then to extract the feature of interest needed for the reconstruction algorithm. Fairly small errors in this mode extraction step usually spoil the reconstruction completely. Also, for a practical NDE system these algorithms must run in real time on portable hardware, and must be able to deal with millions of digitized waveforms per minute. However, it is these signal processing algorithms, coupled to tomographic reconstruction, that allow this inverse problem to be solved so that Lamb wave tomography can be a viable inspection method.

Early Lamb wave tomography work by Hutchins *et al.* [25, 69-72], Achenbach [73], and Degertekin [74] used a standard parallel projection geometry with the velocity and/or attenuation of Lamb waves as input for the tomographic reconstructions. McKeon and Hinders [5, 7-11] implemented this also, and then investigated a “crosshole” tomographic scheme similar to a preliminary pipe inspection study done by Hildebrand *et*

al. [75] that has many practical advantages for aircraft NDE. Malyarenko and Hinders [12] subsequently compared fan beam and double crosshole geometries, and found the latter to be superior. A secondary conclusion of [12] is that the iterative reconstruction algorithms from the seismological literature are much better suited to Lamb wave NDE applications than are the convolution-backprojection type of algorithms from the medical imaging literature. Malyarenko *et al.* [6, 14-16] have recently implemented a curved ray tracing technique that accounts for scattering or refractive media and improves the technique's accuracy in sizing flaws. The rest of this chapter will discuss the current state of the tomographic work presented in [5-12, 14-16] and recent work performed by this author and others [13, 17] to test the robustness of Lamb wave tomography and its application to composite structures.

3.1 LWT Experimental Overview

In our laboratory, a series of ultrasonic apparatus has been assembled that perform Lamb wave scans using traditional parallel projection and fanbeam tomography as well as the various crosshole geometries. Broadbanded contact piezoelectric transducers are used to generate and receive the Lamb waves in a pitch-catch arrangement. Useful diagnostic signals can be propagated many tens of inches in plate-like structures so it is possible to rapidly interrogate large areas. However, scanning with contact transducers is slow and prone to errors from variations in coupling of the ultrasonic energy in and out of the plate. Neither of these is a concern in the laboratory, but they are a serious drawback in service. Therefore, in order for the tomographic inspection schemes to be successful, the scanning system has been designed to mimic arrays of transducers that would take

large area snapshots using Lamb waves. Such arrays of transducers can be “electronically” scanned through all the various combinations that have thus far been implemented in the lab with mechanical scanners.

The Lamb wave modes are generated by standard longitudinal contact transducers excited with a tone burst. This is in contrast to other researchers who use angle-block or comb transducers to select particular Lamb wave modes. We find that for measurements outside the laboratory the careful coupling required to select particular modes via Snell’s law with angle blocks is not practical, and the tomographic requirements for a small transducer footprint and omni-directionality rule out comb transducers. Furthermore, we typically add truncated-cone delay lines to minimize the footprint of the transducer. This increases the spatial resolution of the final tomographic reconstruction because it allows us to take smaller steps between measurements.

For corrosion detection in aluminum aircraft structures we find it convenient to monitor changes in the arrival time of the S_0 mode because of its dispersive properties. Although amplitude measurements are often most sensitive to the presence of flaws, because the received signals are strongly affected by the variations in coupling inherent in field measurements, we usually record time-of-flight information instead. Our measurements are typically performed at a frequency thickness product of $fd \approx 2$ MHz-mm where only the lowest order symmetric and antisymmetric (S_0 and A_0) modes propagate appreciably in many materials. This is done to try and control the complexity of the signals and ensure that the first arriving mode, S_0 , is highly dispersive and that higher-order modes are cut off or negligible. Figure 3-1 shows Lamb wave dispersion curves for aluminum, along with a typical Lamb waveform recorded by our system.

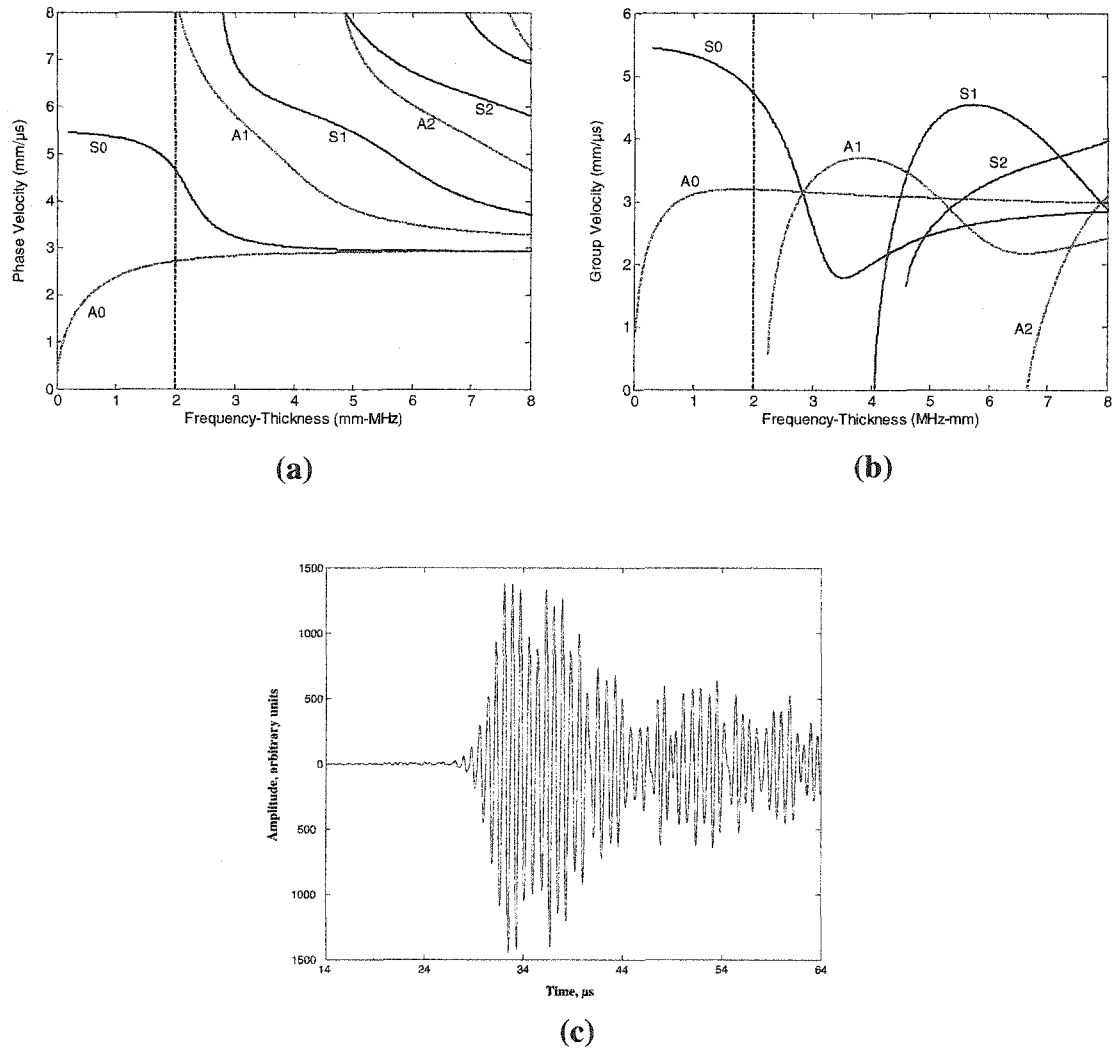


Figure 3-1. Lamb wave dispersion curves showing phase velocity (a) and group velocity (b) versus frequency-thickness product for aluminum. The antisymmetric modes are indicated by the dashed blue lines and the symmetric modes are indicated by the solid red lines. Our measurements are typically made below a frequency-thickness of 2 MHz-mm. A typical Lamb waveform recorded by our system is shown in (c).

Since tomographic reconstructions require many individual measurements to develop the projection data, we have concentrated our efforts on those measurement schemes which have the most promise for being fully automated. Initially, parallel projection and fan beam tomography were pursued because of their extensive use in the medical community. However, certain limitations in their scanning geometries and reconstruction fidelity led to the development of Lamb wave multiple crosshole tomography [15]. Ultimately, all the tomographic reconstruction algorithms run in near real time on modern PCs so there appears to be no inherent computational limits on the implementation of these techniques in the field.

3.2 Parallel Projection Tomography

Figure 3-2 shows the geometry for the parallel-projection tomography system. The transducers are scanned along parallel lines with the Lamb waves propagating between them. At each position in the scan a measurement of the Lamb wave's arrival time is recorded. The waves are assumed to travel only in straight paths (rays) as shown. Once the pitch-catch measurements for each ray in an individual orientation have been taken, the sample is rotated by a fixed amount and the measurement is repeated. Projections consisting of seven parallel rays (transducer-pair positions) for four orientations (0, 45, 90 and 135 degrees) are shown in Figure 3-2. However, in practice each projection would have about 100 rays, and projections would be taken at least every 5 degrees.

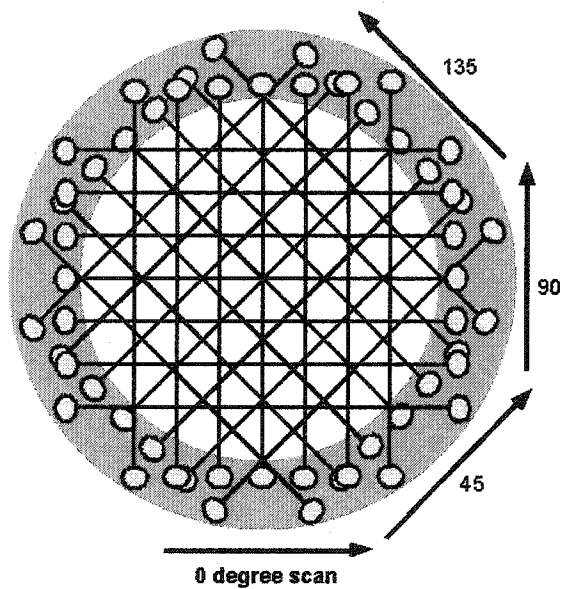


Figure 3-2. The geometry for parallel-projection tomography is shown for the case of seven parallel projections at four orientations.

For any tomographic measurement, the “ray density” is critical to the quality of the reconstruction. Ray density is a measure of the degree to which the various pitch-catch Lamb wave measurements cover the region of interest. Note from Figure 3-2 that the ray pattern within the scanning region is uniform for parallel projection tomography. This is an important difference between the parallel projection and crosshole tomography techniques. Similarly, having the rays pass through the region of interest from many orientations is also important to the quality of the reconstruction. The rays for parallel projection tomography cover all angles since projections must be evenly spaced over 180 degrees.

A schematic of the parallel projection scanner in our laboratory is shown in Figure 3-3. At each location of the pitch-catch transducer pair, the phase shift of the S_0 mode is acquired through pulsed-phase-locked-loop (P2L2) circuitry. This instrument compares the phase of its pulsed output signal, which is sent to the transmitting transducer, with that of the amplified and low-pass filtered returned signal from the receiving transducer. A frequency counter is connected to the output of the P2L2, which gives information on the phase difference of the two signals in terms of frequency. The value of this reference frequency can be used to calculate both the time-of-flight and, because the distance between two transducers is fixed, the integrated velocity of the Lamb waves. In our setup, the sample is rotated by a fixed amount between each scan by a computer-controlled rotary table in order to obtain data from the different orientations necessary for tomographic measurements. A detailed description of the fairly standard convolution-backprojection reconstruction algorithm can be found in previous work done

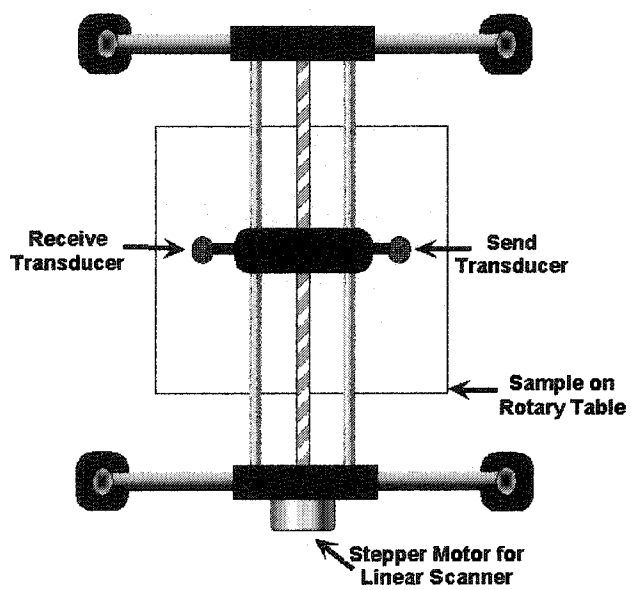


Figure 3-3. The parallel projection scanning system is shown schematically.

by McKeon and Hinders [9] or from Kak and Slaney [4]. Figure 3-4 (a) shows a parallel projection reconstruction of a flat-bottomed hole in an aluminum plate. From the reconstructed image we can see that this technique has the ability to locate and size flaws accurately. It also demonstrates how the images can be easily interpreted without any knowledge of guided wave physics.

In a parallel projection tomography measurement, the source-receiver pair has to linearly scan over the length of a projection, then rotate and scan the next projection until a specified number of projections are recorded. The method requires either rotation of the sample or rotation of the transmitter-receiver assembly. Both are slow, cumbersome, and impractical for real world testing conditions where large objects need to be scanned and the data acquisition process has to be in real-time. Transducer arrays cannot be used to solve these problems because we cannot completely exclude mechanical motion in the parallel projection tomography system. Furthermore, a fairly large ring surrounding the region of interest (shaded area in Figure 3-2) must be free of obstructions due to the mechanical motions required to record parallel line-scans at many angles. This also makes scanning over large areas cumbersome because of the overlap needed between scans to ensure that the entire region is thoroughly inspected.

3.3 Fan Beam Tomography

In order to overcome some of the mechanical limitations of parallel projection tomography, a Lamb wave fanbeam tomography scanner was built. First-generation medical CAT scanners used the parallel projection geometry, but modern CT scanners

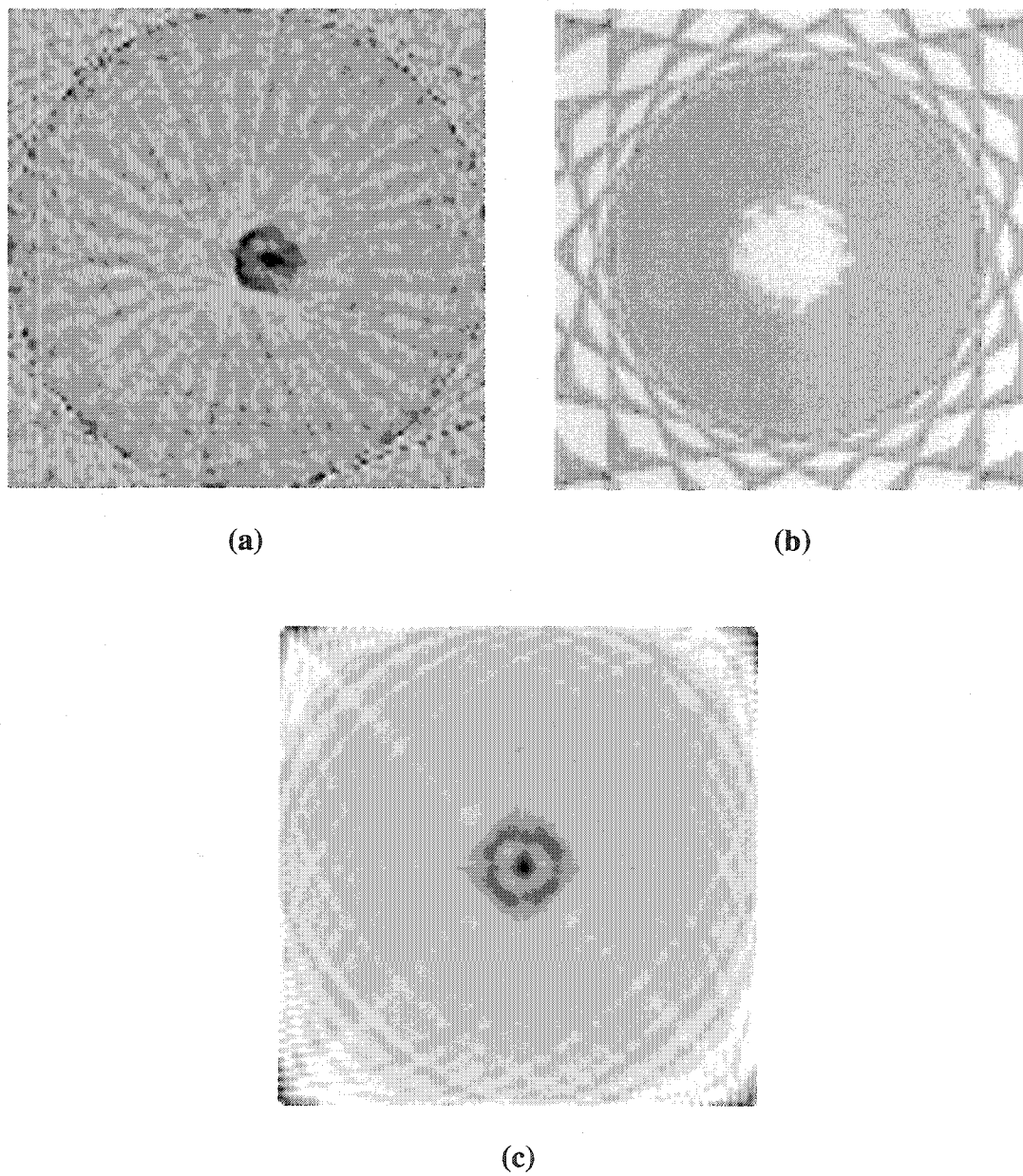


Figure 3-4. Tomographic reconstructions of a $3/32$ " thick aluminum plate with a 1" diameter flat bottom hole, 30% thickness loss: (a) parallel projection result, (b) fan beam result, and (c) multiple crosshole result where the scanned area is 20 cm x 20 cm.

use a fan beam geometry [76] which employs a stationary ring of detectors to dramatically increase the speed of data acquisition. This can also be done for Lamb wave tomography via a ring of transducers surrounding the region of interest. Ultimately, each transducer in the ring would act as both a transmitter and a receiver. Figure 3-5 and Figure 3-6 show how this was implemented for testing in the lab. A commercial system would implement this with a circular array of transducers. The reconstruction algorithms are only slightly modified from the parallel projection case, but the varying path length means that the P2L2 no longer works for extracting the Lamb wave arrival times. Moreover, the “fill factor” – defined as the area where the reconstruction is valid – associated with the fan beam geometry is quite poor. This is because the area free of reconstruction artifacts is limited to the circular area where all the fans overlap. Although this technique can be made to work [12], practical issues force one to conclude that fan beam tomography is not viable for Lamb wave NDE.

Figure 3-4 (b) shows a fan beam reconstruction for the same flat-bottomed hole sample as above. Again the flaw is accurately reconstructed in a readily interpretable thickness map, but the artifacts demonstrate that the fill factor is disappointingly small. It also should be noted that the convolution-backprojection family of algorithms is surprisingly sensitive to the types of measurement noise and imprecision inherent in any in-the-field data acquisition scenario. These issues along with the impracticality of the parallel projection geometry led to the development of the multiple crosshole technique described below [6, 14].

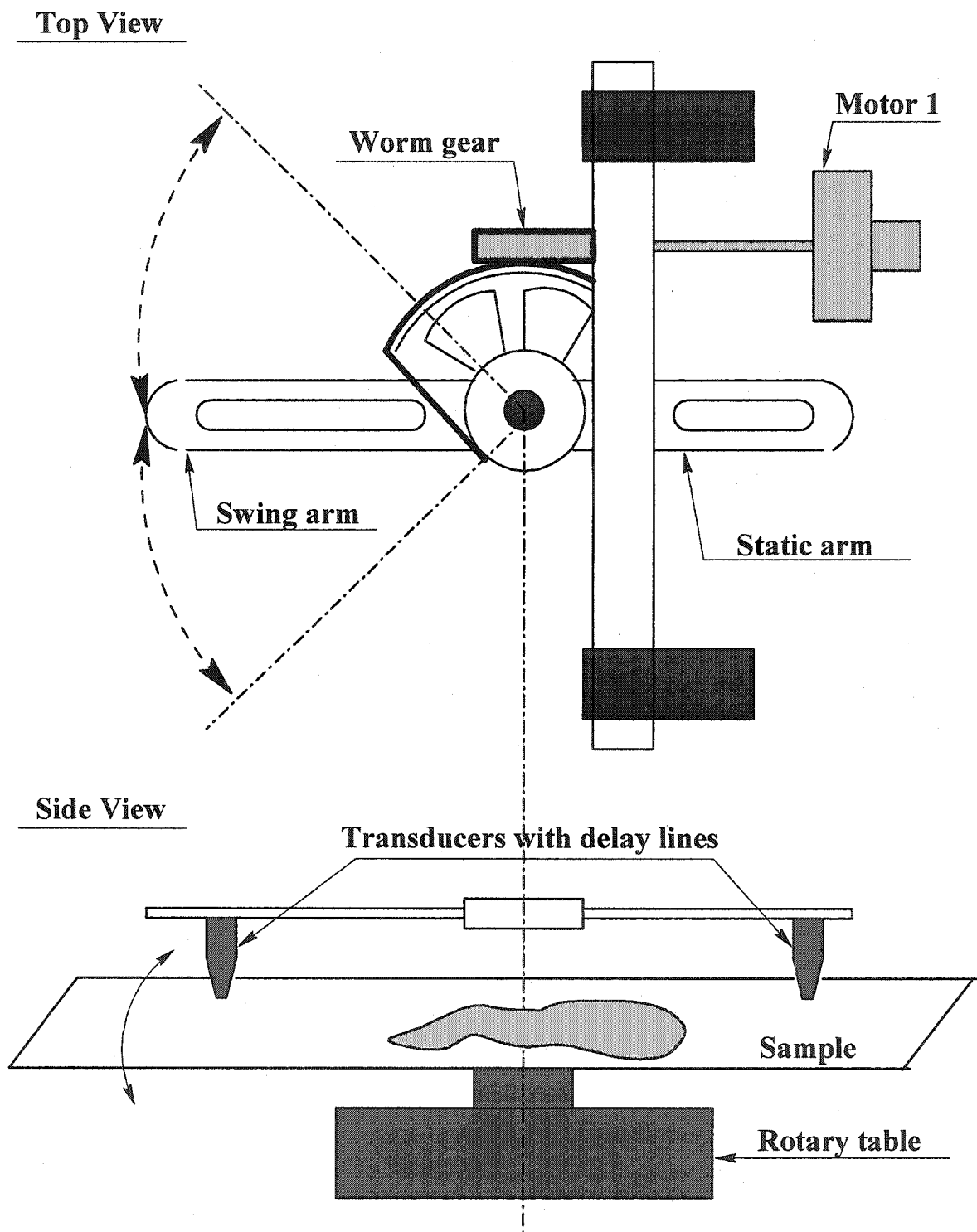


Figure 3-5. Schematic of Lamb wave fan beam tomography scanner [6].

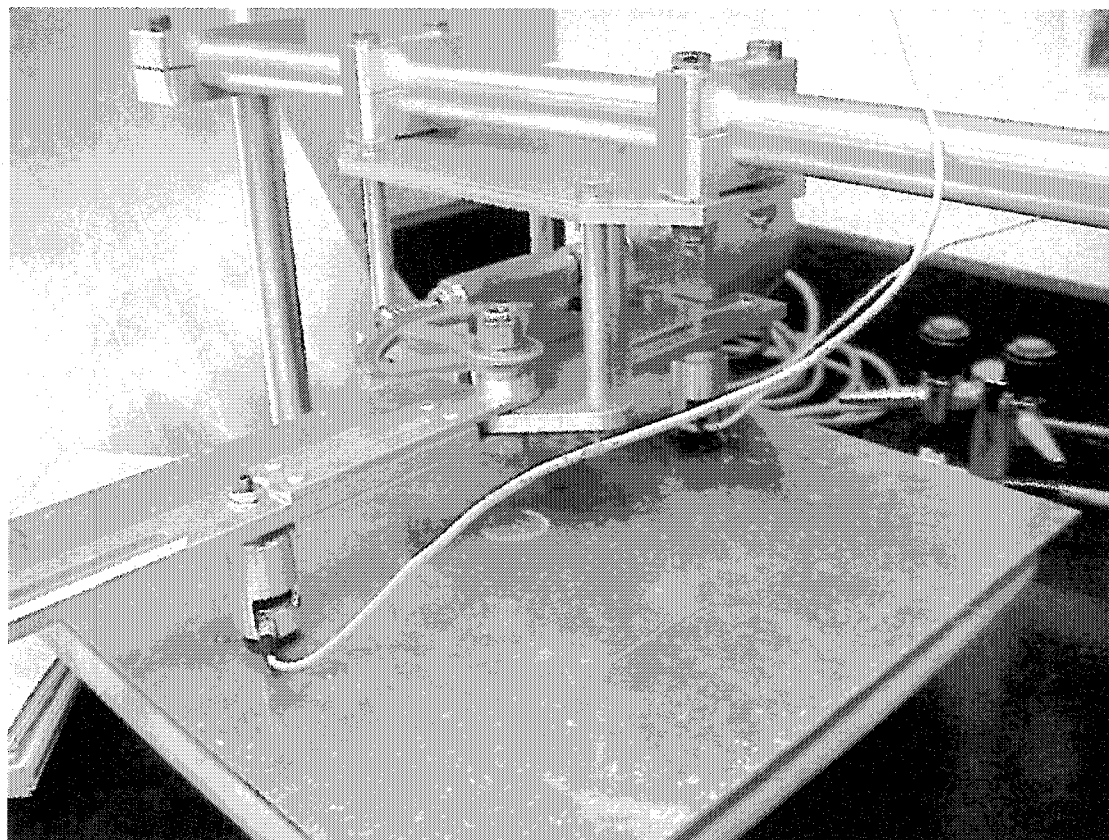


Figure 3-6. Photograph of the laboratory fan beam scanner.

3.4 Crosshole Tomography

A fast and practical alternative to the parallel projection technique can be adapted from cross-borehole tomography used by seismologists [19, 20]. In seismology, parallel boreholes are driven into the ground adjacent to the region of interest, and seismic sources and receivers are sequentially placed at many down-hole distances to record a crisscross seismic ray pattern. To improve the ray density, seismologists often place a line of receivers along the surface. In our case we are able to go a step further and use a four-legged perimeter array of transducers surrounding the region of interest. This technique is called multiple crosshole tomography.

The iterative families of algorithms developed in the seismological literature are better suited to crosshole tomography than are the convolution-backprojection algorithms developed for medical imaging and other applications [12-14, 17]. In particular, we find that the simultaneous iterative reconstruction technique (SIRT) algorithm [4] is relatively robust, computationally efficient, and insensitive to experimental noise. It also has the inherent advantage of being widely applicable to a variety of geometries and incomplete data sets. Moreover, the SIRT is able to be extended to account for material anisotropy and “ray bending” due to scattering from flaws [14].

Figure 3-7 demonstrates the geometry of our multiple crosshole setup. The circles about the perimeter of the grid represent the different transducer locations. An in-the-field system would implement this type of setup with a square-perimeter array. However, in our current laboratory system we use only one pair of independently

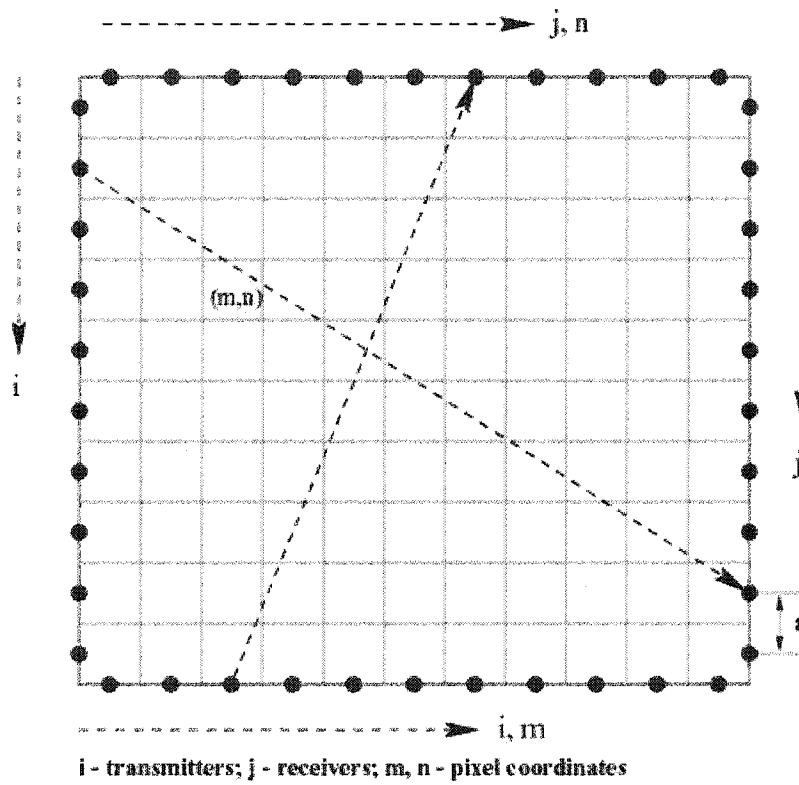


Figure 3-7. Geometry for multiple crosshole Lamb wave tomography.

scanning pitch-catch transducers. Therefore, only two positions are occupied at a time. Both transducers are attached to linear slider screws and are moved back and forth by computer controlled stepper motors. For each projection, the transmitting transducer steps along a particular edge as the receiving transducer steps through all the positions on a different edge. For example, in one of the double crosshole projections (see Figure 3-7) the transmitting transducer steps along the lower edge from left to right incrementing i from 0 to $M-1$. Meanwhile, the receiving transducer steps along the upper edge sweeping all N available j -positions for each transmitter position. At each pitch-catch position the entire waveform is digitized and recorded by stacking it into a data file.

As stated above, the quality of the resulting reconstruction is directly related to the uniformity of the ray density within the scanning region. It was also noted that the ray density is uniform for the parallel projection geometry, but not for the crosshole scheme. To best overcome this deficiency in the crosshole geometry, we use the rays that connect all the possible pairs of points in a discrete square-perimeter array. For testing in the laboratory, the square array can be split into six sets of projections as shown in Figure 3-8. The data acquisition equipment and scanning system configuration are described in Figure 3-9. In the lab, each of the different projections can be obtained by moving the pair of linear slider screws to the different positions around the perimeter of the sample (see Figure 3-10).

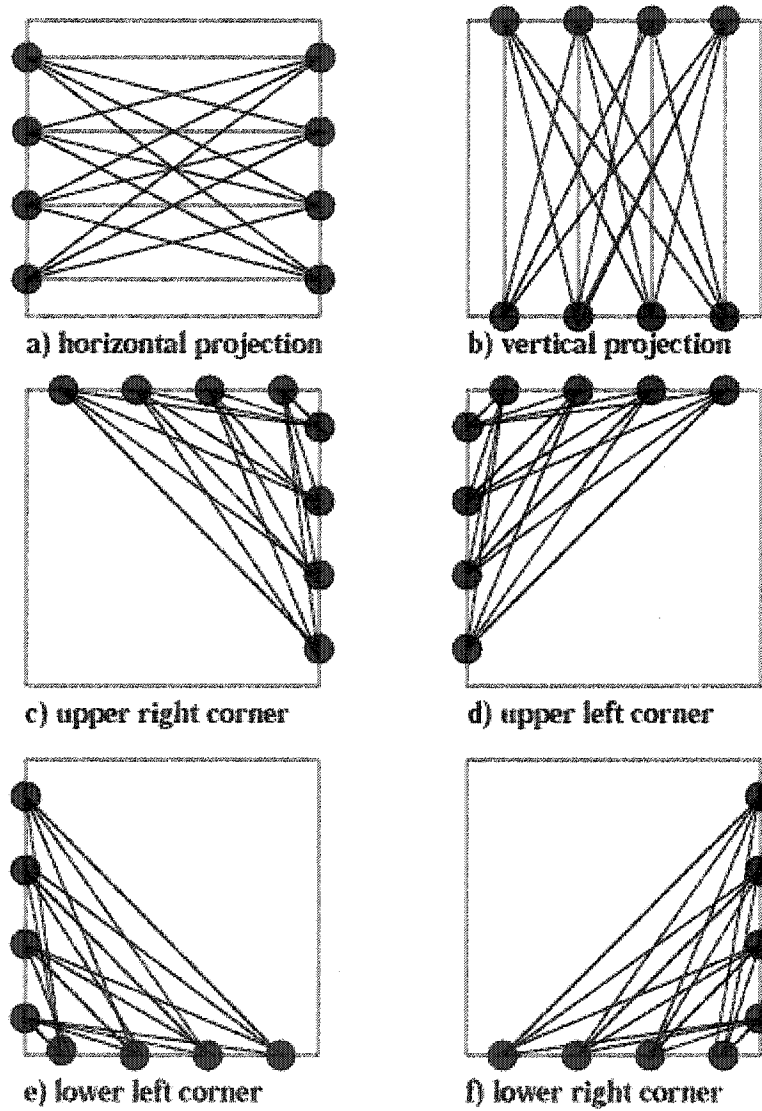


Figure 3-8. Six possible crosshole projections. Although four transducer positions are shown per side, in practice we use at least 100 positions per leg of the perimeter.

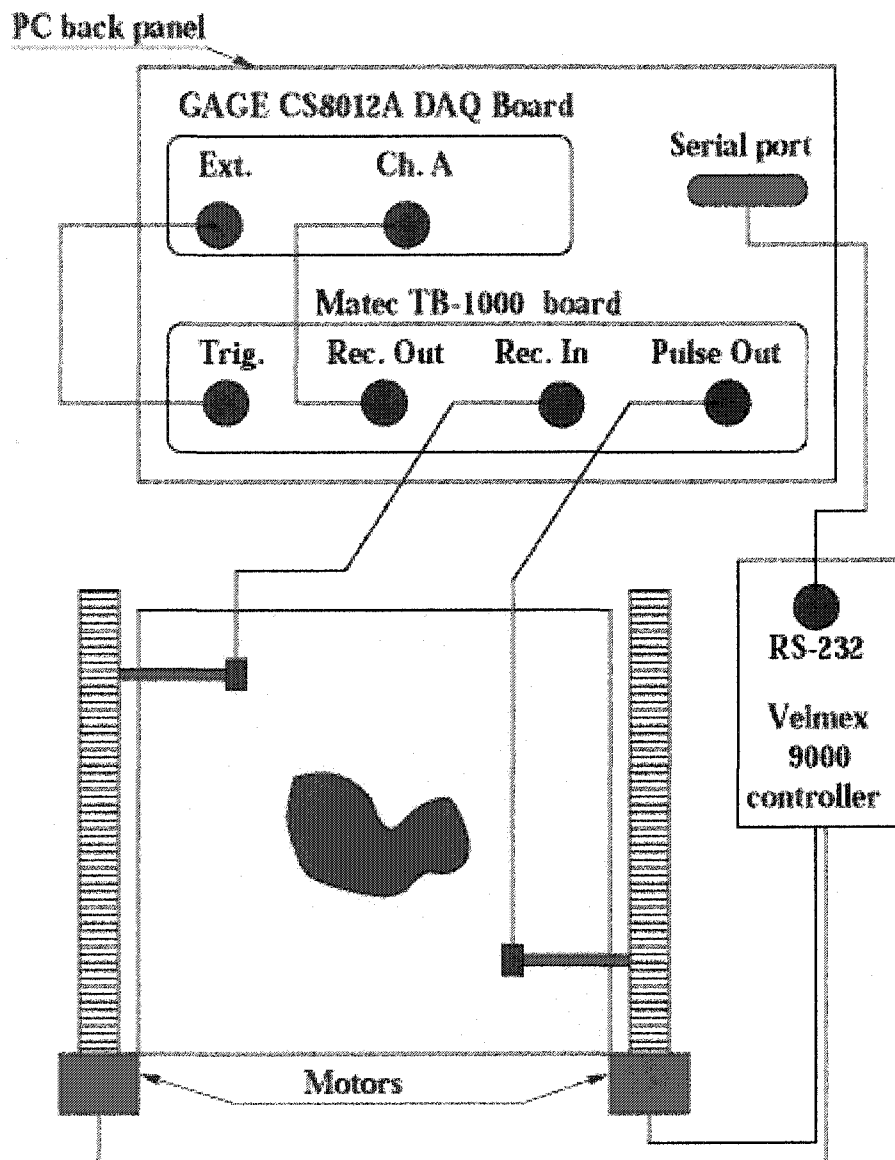


Figure 3-9. Lamb wave tomography scanning system with computer plug-in boards in a PC running Linux.

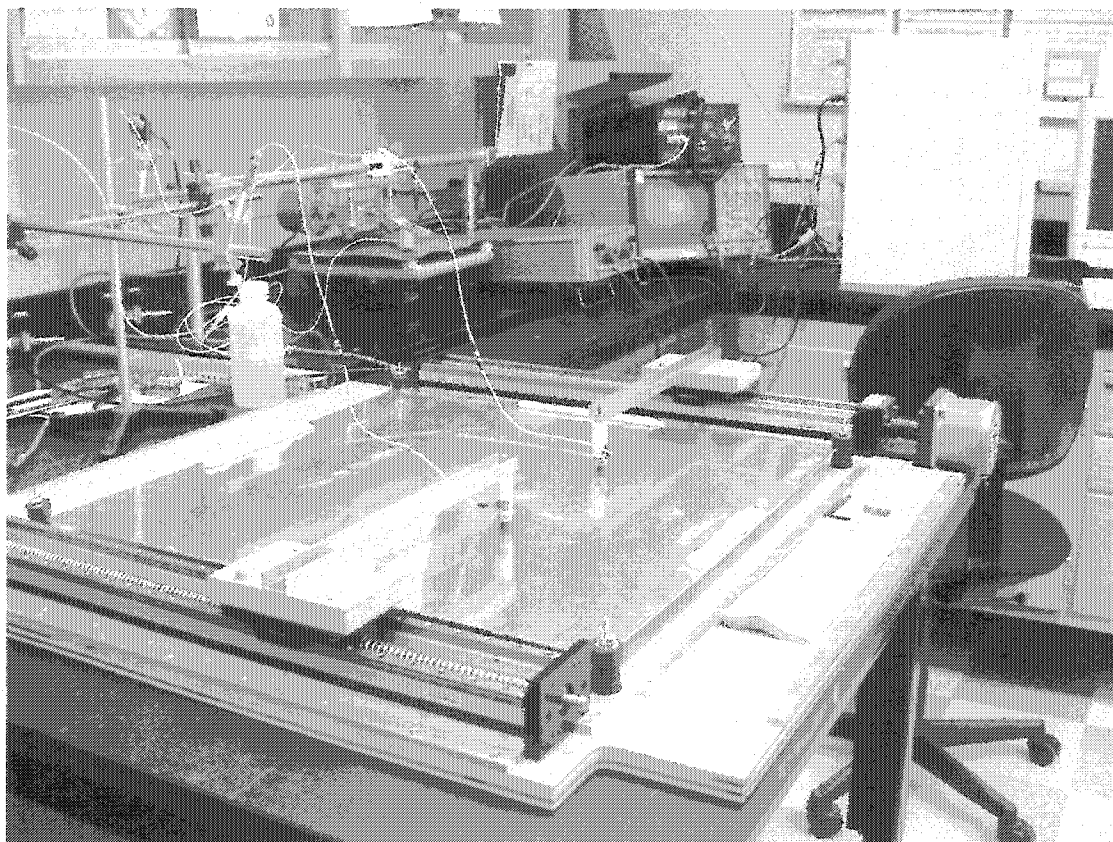


Figure 3-10. Photograph of the Lamb wave tomography scanner in the laboratory.

3.4.1 Simultaneous Iterative Reconstruction Technique (SIRT)

Crosshole tomography uses the SIRT to solve the inverse problem of recovering an object from its projections. What makes this scheme useful for Lamb wave tomography is its iterative nature and great flexibility. It allows practically any scanning geometry and the existence of incomplete data sets. In contrast, the parallel projection algorithm belongs to the convolution-backprojection family, which requires strictly determined scanning configurations and is very sensitive to incompleteness or noise in the experimental data.

For simplicity and brevity, the following discussion assumes that the waves travel in straight paths and has ignored scattering effects. The multiple crosshole tomographic reconstruction algorithm attempts to solve for the Lamb wave velocity within each grid cell, $v[m,n]$ (see Figure 3-7). In general, this can be done by solving for the slowness, $1/v[m,n]$, from the following system of linear equations:

$$T[i, j] = \sum_{m,n \in \text{ray}[i,j]} t[i, j, m, n] = \sum_{m,n \in \text{ray}[i,j]} \frac{d[i, j, m, n]}{v[m, n]}. \quad (3-1)$$

In this system of equations, $T[i,j]$ is the total time it takes the wave to travel from the transmitter to the receiver and $t[i,j,m,n]$ is the amount of time that $\text{ray}[i,j]$ travels within the cell $[m,n]$. $t[i,j,m,n]$ is equivalent to the length of $\text{ray}[i,j]$ in cell $[m,n]$ – denoted by $d[i,j,m,n]$ – divided by the cell velocity $v[m,n]$. The segment lengths, $d[i,j,m,n]$, are calculated theoretically and $T[i,j]$ is measured experimentally. The solution of this system of equations yields a velocity map over the entire region. Given the operating

frequency, we can then convert this velocity map into a thickness map used to detect flaws within the region of interest.

For our current scanning system, Eqn. (3-1) provides a system of N^2 equations, where typically $N = 100$. The limitation on the number of transducer elements (N) is a combination of the footprint of the transducer, the allowable scanning time and desired reconstruction quality. To avoid the computationally expensive inversion of such a large matrix, the SIRT algorithm is used to solve this problem. The adaptation of the SIRT algorithm to our problem has four main steps:

Step 1: First we determine the segment lengths $d[i,j,m,n]$ theoretically. Then, using an initial guess $v^0[m,n]$ for the cell velocities, we calculate the initial estimated arrival times for each ray:

$$T^0[i, j] = \sum_{m,n \in \text{ray}[i,j]} \frac{d[i, j, m, n]}{v^0[m, n]} . \quad (3-2)$$

In subsequent iterations the estimated arrival times are calculated with the updated cell velocities calculated below in Step (3):

$$T^k[i, j] = \sum_{m,n \in \text{ray}[i,j]} \frac{d[i, j, m, n]}{v^k[m, n]} , \quad (3-3)$$

where k is the iteration number.

Step 2: For every ray calculate the difference between the velocities in the current iteration from those in the previous iteration for each cell that the ray passes through:

$$\Delta \frac{1}{v_{m,n \in \text{ray}[i,j]}[m, n]} = \frac{T^k[i, j] - T^{k-1}[i, j]}{L[i, j]} , \quad (3-4)$$

where $L[i,j]$ is the length of ray $[i,j]$ and $T[i,j]$ is the experimentally measured arrival time for the ray $[i,j]$. During each iteration, cycle through each ray and record how many times each individual cell has a change in velocity and what that change is.

Step 3: Finally, update each cell's velocity by taking the average of the differences recorded for that cell in step (2) and adding it to the cell's current velocity:

$$\frac{1}{v^{k+1}[m,n]} = \frac{1}{v^k[m,n]} + \Delta_{AVG} \frac{1}{v[m,n]} . \quad (3-5)$$

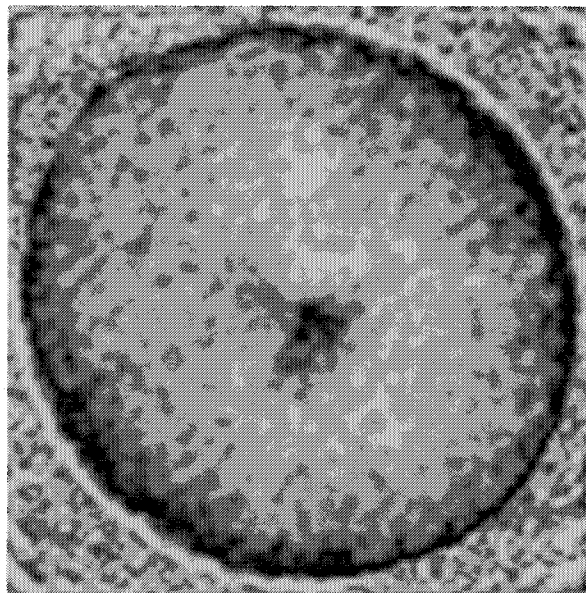
Step 4: Steps (1)-(3) are repeated until the required accuracy is reached. Typically, fifty global iterations are used.

Figure 3-4 shows a direct comparison of multiple crosshole tomography with the fanbeam and parallel projection tomography results discussed earlier. The crosshole measurements were done on a 100 x 100 square matrix with a step size between locations of 2 mm. Therefore, the total number of captured waveforms was $N_{tot} = 6 \times 10^4$. The images are reconstructions based on the extraction of the first mode's arrival time. In order to remove some of the reconstruction artifacts that were due to erroneous data points, the time-of-flight data was transformed into the velocity domain. In this domain the data are less scattered and we can identify the points with outlying velocities and truncate them using empirical rules. It has been shown previously that the extra truncation step removes artifacts in the final reconstruction and is successful where other techniques, such as smoothing or filtering time domain data, are not [6]. It can be seen

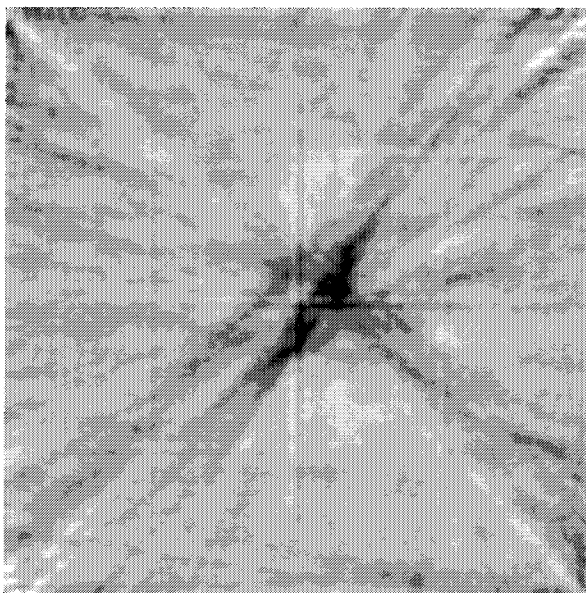
from these reconstructions that the multiple crosshole method is also effective in locating and sizing the flaw. More importantly, the advantages of this geometry over both parallel projection and fan beam tomography make it a feasible system for use in nondestructive evaluation.

3.4.2 Composite Plate Results

Composite materials introduce additional complexities due to their anisotropic nature. Consequently, the more complicated nature of these materials could adversely affect the ability of the LWT system to inspect these types of materials. Figure 3-11(a) and (b) are respectively a parallel projection and crosshole scan of an impact damaged multi-layer woven graphite epoxy plate with a Cartesian grid pattern of through-thickness Kevlar stitching. The sample was 1.75 mm thick. The impact damage was created by hitting the center of the sample repeatedly with a hammer. The actual extent of the flaw cannot be seen visually on either surface, but the reconstructions clearly show changes in the material properties at the location of the defect. Figure 3-12 is a quantitative comparison of the velocity profiles through the flaw region for the two reconstructions in Figure 3-11(a) and (b). Figure 3-13(a) and (b) are parallel projection and multiple crosshole scans of the same type of graphite epoxy sample with an approximately 25 mm diameter irregular through-hole, i.e. a circular hole cut freehand with a jig saw. These results clearly demonstrate that Lamb wave tomography is also useful for the NDE of composites.



(a)



(b)

Figure 3-11. (a) Parallel projection [6] and (b) multiple crosshole scans of a woven graphite epoxy sample with impact damage.

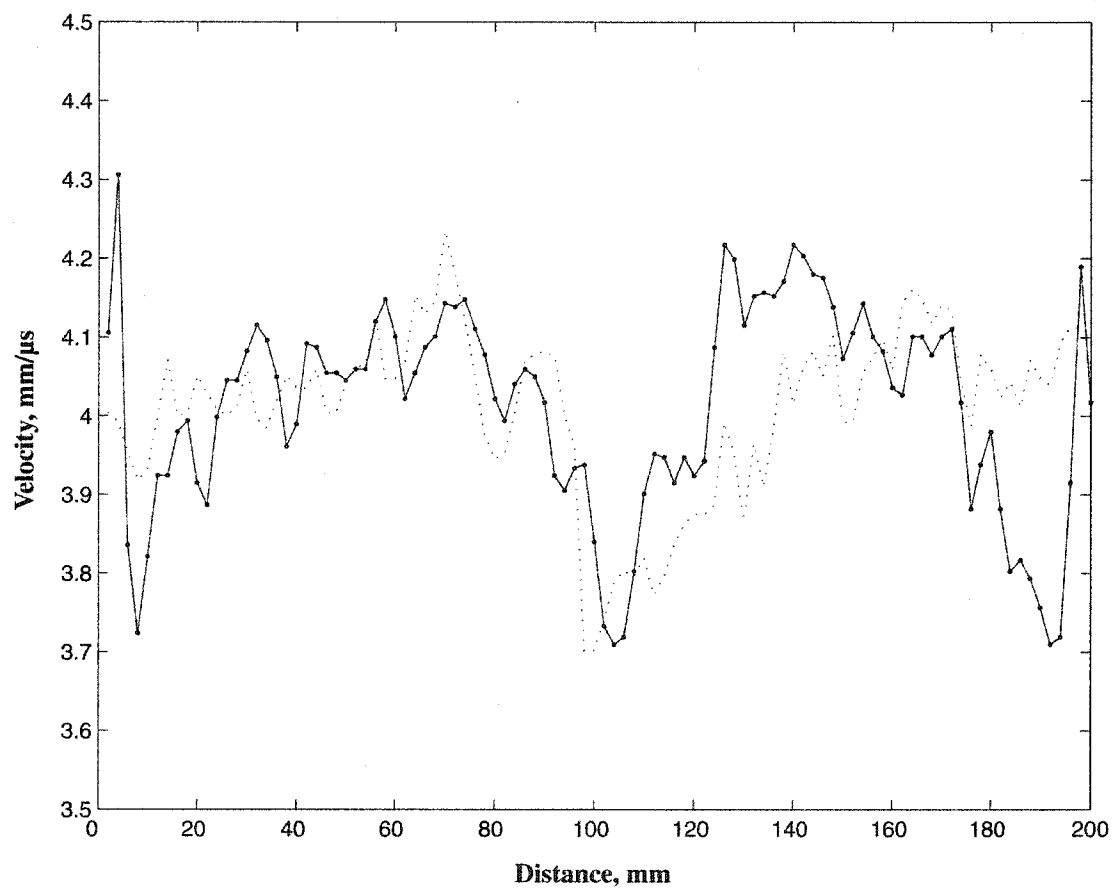
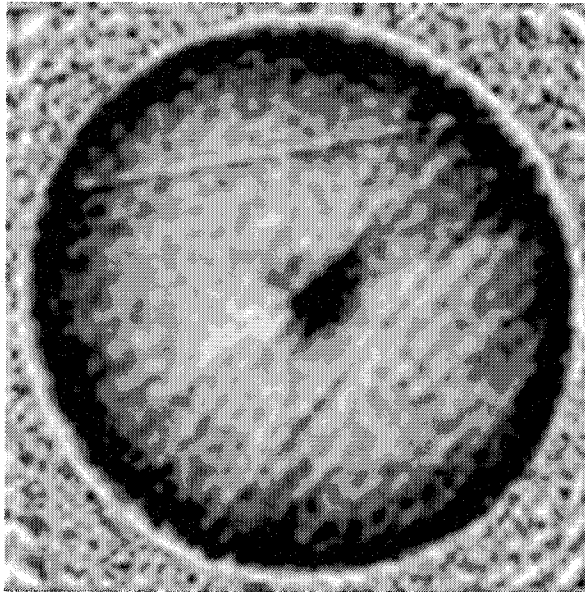
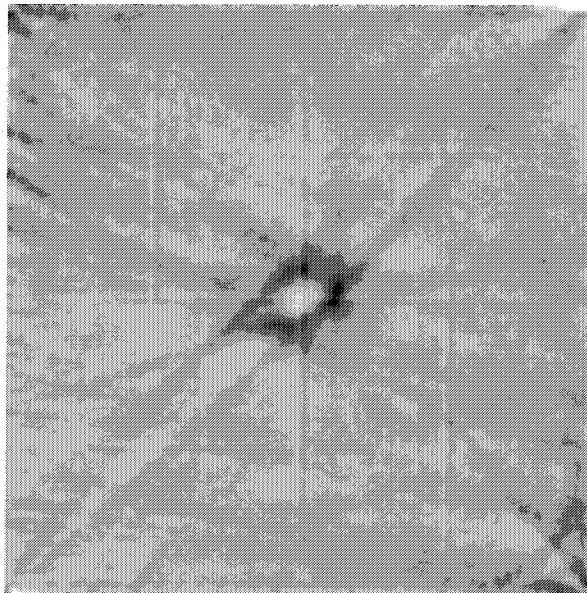


Figure 3-12. Cross-sectional slices of the images in Figure 3-11 at a horizontal mid-line through the flaw. The dotted line is for the double crosshole scan and the solid line with circles is for the parallel projection scan.



(a)



(b)

Figure 3-13. (a) Parallel projection [6] and (b) multiple crosshole scans of a woven graphite epoxy sample with a 1" through-hole

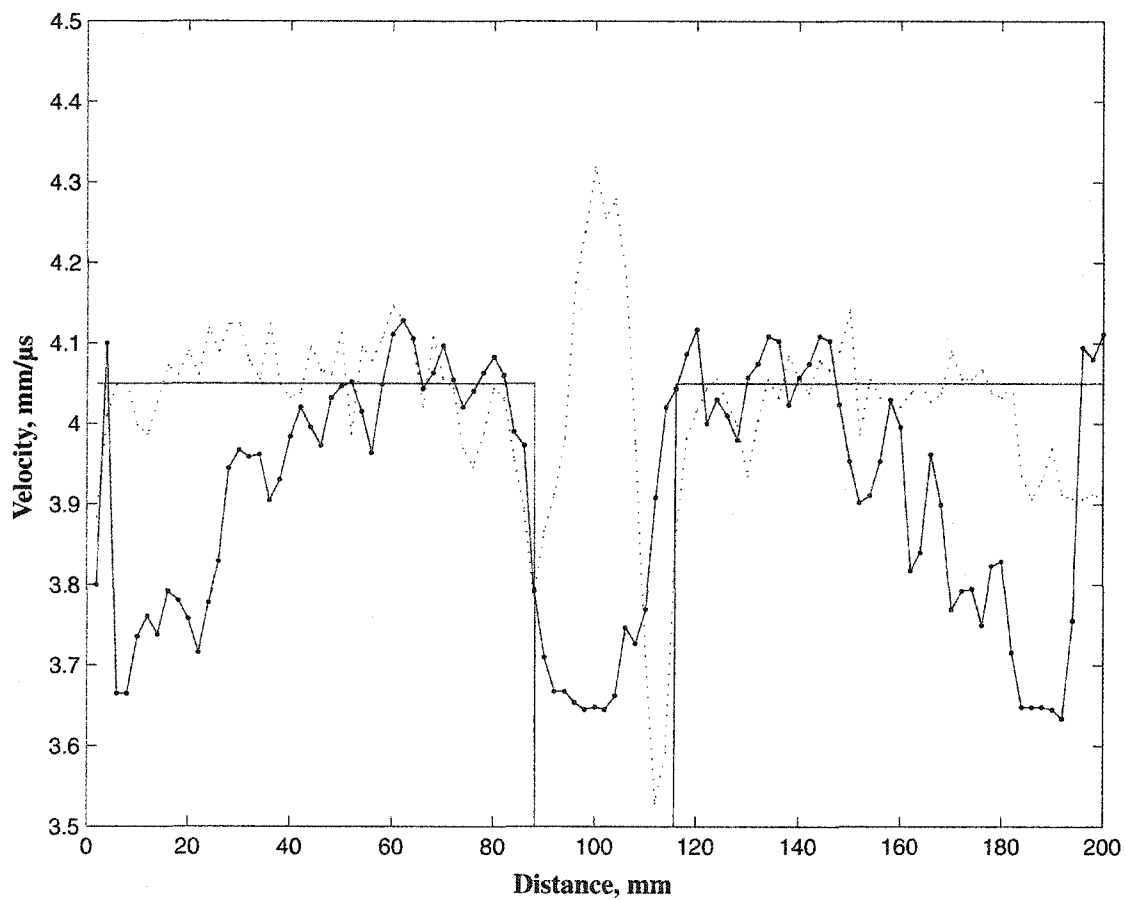


Figure 3-14. Cross-sectional slices of the images in Figure 3-13 at a horizontal mid-line through the flaw. The dotted line is for the double crosshole scan and the solid line with circles is for the parallel projection scan.

Figure 3-14 is a quantitative comparison of the velocity profiles for the two reconstructions in Figure 3-13(a) and (b). This graph plots the measured Lamb wave velocity as a function of position for a particular horizontal cross-section of the image. For this figure, a line through the center of the flaw was chosen. The solid line indicates the position and width of the flaw. It can be seen that the crosshole scan slightly oversizes the defect while the parallel projection scan more accurately sizes it. It is also apparent that the effective scanning region is greatly reduced in the parallel projection geometry. It is only over small regions (outside the flaw) that the measured velocity is at the background level of $4.05 \text{ mm}/\mu\text{s}$. This is due to the reconstruction algorithm and how it solves the inverse problem. We haven't investigated techniques to correct this for parallel projection tomography because the double crosshole scheme has so many practical advantages.

One final thing to note is the peak that exists in the center of the flaw for the multiple crosshole reconstruction. This feature is often seen in through-hole samples due to the inability for the straight ray SIRT algorithms to account for the finite Lamb wave beam. For flaws that are the same size or smaller than the diameter of the beam, some of the energy will go around the flaw and some will interact with it. This type of complicated effect is why certain artifacts, such as the null region in the middle of the flaw, are introduced into the reconstructions. It is also the reason why work is being done to adapt the diffraction tomography done in [14] to strongly anisotropic samples. Some of the distortion we see in the reconstructions for the composite samples is due to anisotropy, and we have begun to implement corrections for variation of Lamb wave velocity with direction. In cases where we can scan an "unflawed" plate with the same

layup, our tomographic measurement provides the data necessary to correct for anisotropy in the reconstructions. In cases where no pristine sample is available, an approach similar to [59] can be used for calculating the Lamb wave dispersion curves versus angle when the composite plate layup is known.

These results on composite plates are encouraging because they show the applicability of the method to more complicated structures. In addition, they demonstrate that the multiple crosshole tomographic scans can accurately image flaws despite the irregular ray density in the measurements. Coupled with the advantages over parallel projection tomography for a practical in-the-field scanner, these attributes are why we are focusing on the multiple crosshole scheme for plate inspection.

3.4.3 Blind Study Test

As a test of the multiple crosshole Lamb wave tomography (LWT) scanning apparatus and reconstruction algorithms, an experiment was constructed which is double-blind in the medical sense [13]. Since our motivation is identifying flaws in aging aircraft structures with the LWT system, we purchased a number of identical aluminum plates and introduced representative flaws into them. After scanning the samples with the LWT system, verification was independently done with traditional ultrasonic C-scans in an immersion tank. Sample construction, LWT scanning, immersion tank testing, and tomographic reconstruction were all performed by separate individuals. In addition, after construction, the samples were covered so that the personnel responsible for scanning the samples could not see or feel the different flaws. All the flaws were single-sided and scanned from the backside.

Sample reconstructions from LWT scans are shown in Figure 3-15 thru Figure 3-19. The scanned region was 20 cm x 20 cm in every case. Outside of the personnel responsible for constructing and scanning the flaws, a group of six individuals familiar with nondestructive inspection of aerospace materials answered the following four questions for each plate sample:

1. How many flaws?
2. What do you think the flaws are?
3. How severe are the flaws?
4. Describe the size, shape and orientation of the flaws?

The responses to each of the plates are indicated below, along with a description of the actual flaws that had been introduced into each plate:

- Plate #1: 2.2-inch diameter flat bottomed hole, 50% thickness loss. Everyone got this one correct, although some read a bit more into the structure of the image.
- Plate #2: 4-inch square belt-sander thinned area, ~10% thickness loss. Some thought this was a set of four perimeter troughs. The flaw was sized and localized correctly, though.
- Plate #3: Three flaws of the same diameter but different depths: 50%, 5% and ~0% thickness losses made by plunging an endmill. Since the third flaw was too minor (the end mill didn't even fully engage the material), the two-flaws, with one more severe than the other, was considered a correct call.

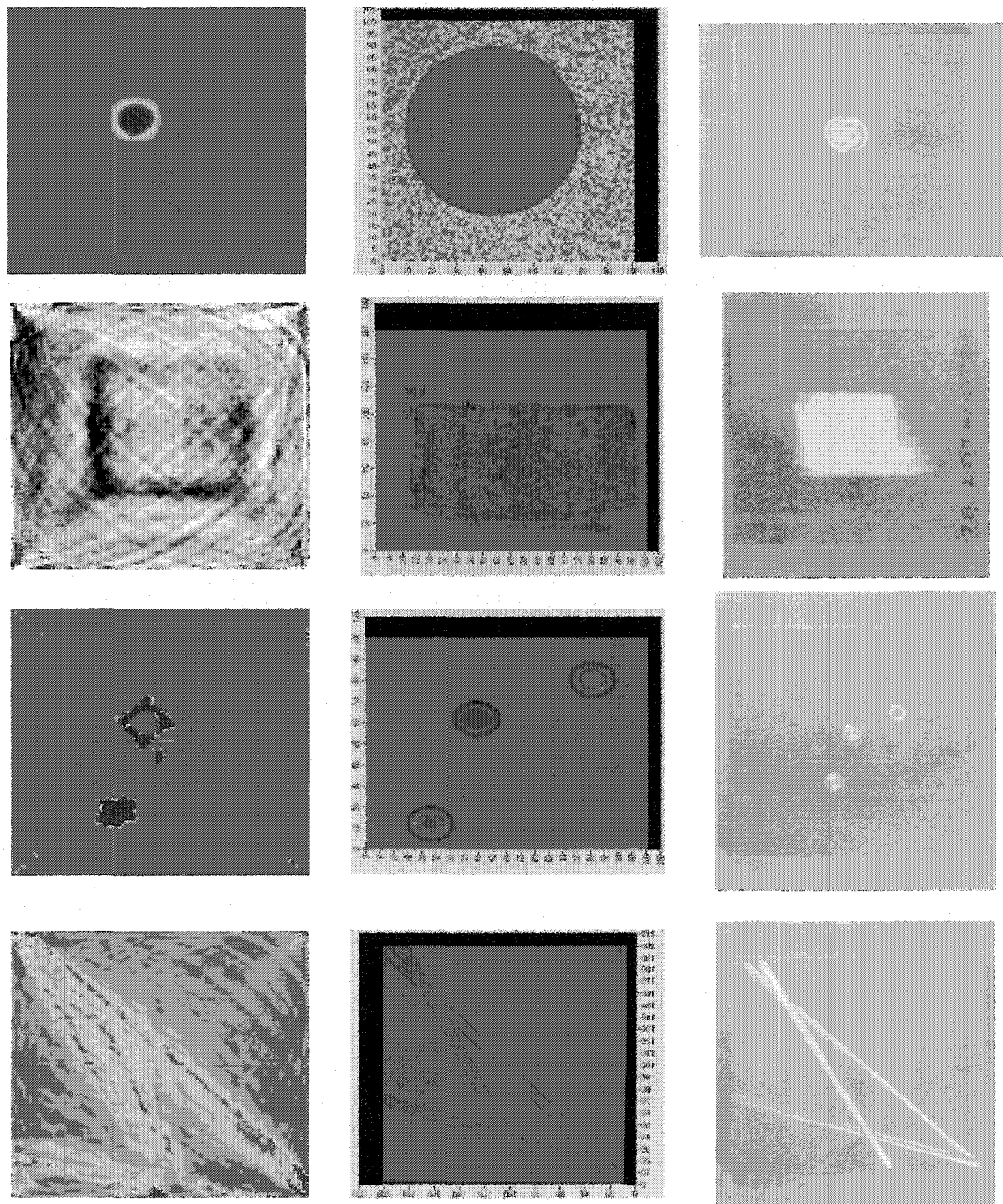


Figure 3-15. LWT blind study results for plates 1-4. The first column is the LWT tomographic reconstruction, the second column is the C-scan results and the third column is a photograph of the actual sample. Note that the scanned area is smaller than the actual plate size.

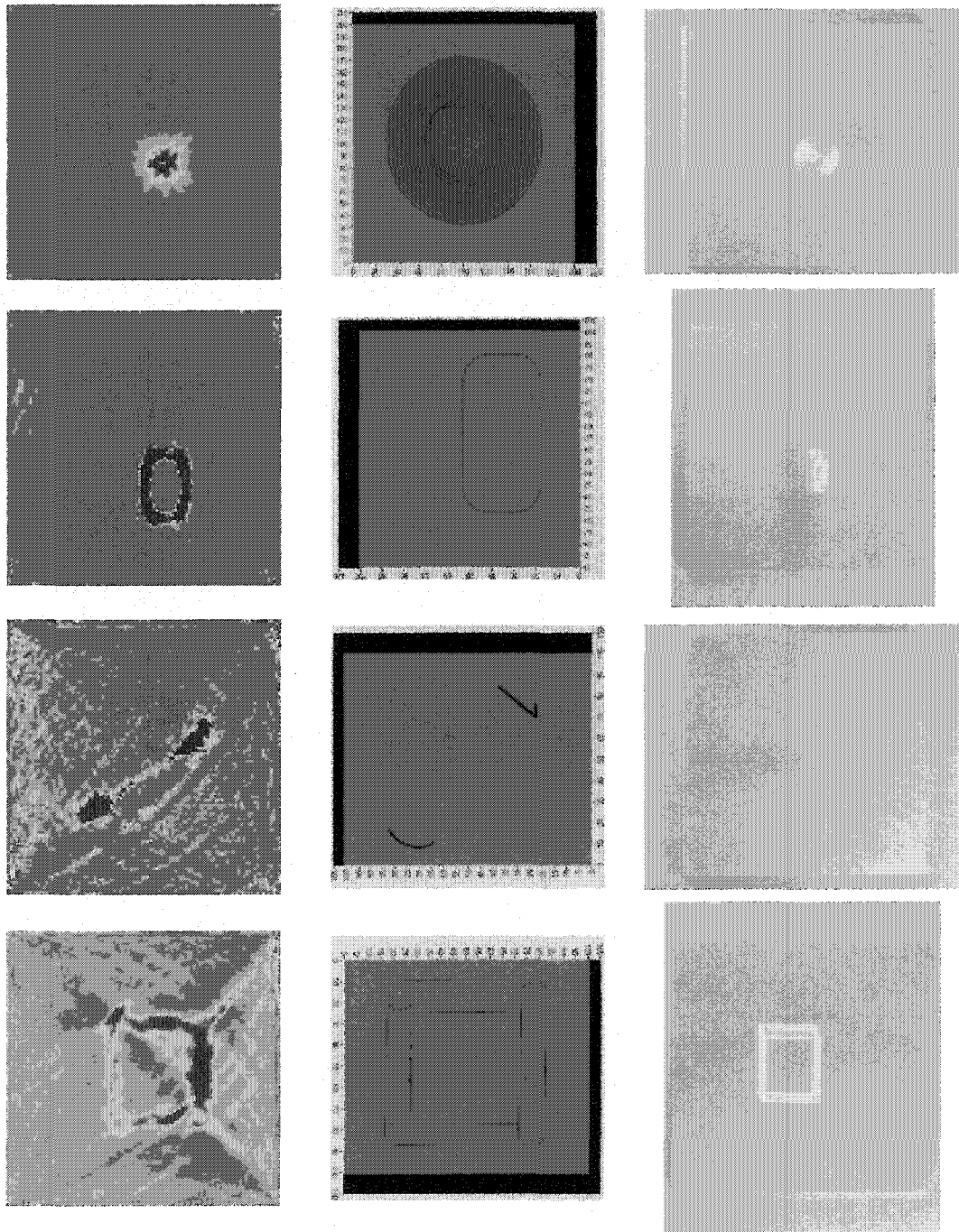


Figure 3-16. LWT blind study results for plates 5-8. The first column is the LWT tomographic reconstruction, the second column is the C-scan results and the third column is a photograph of the actual sample. Note that the scanned area is smaller than the actual plate size.

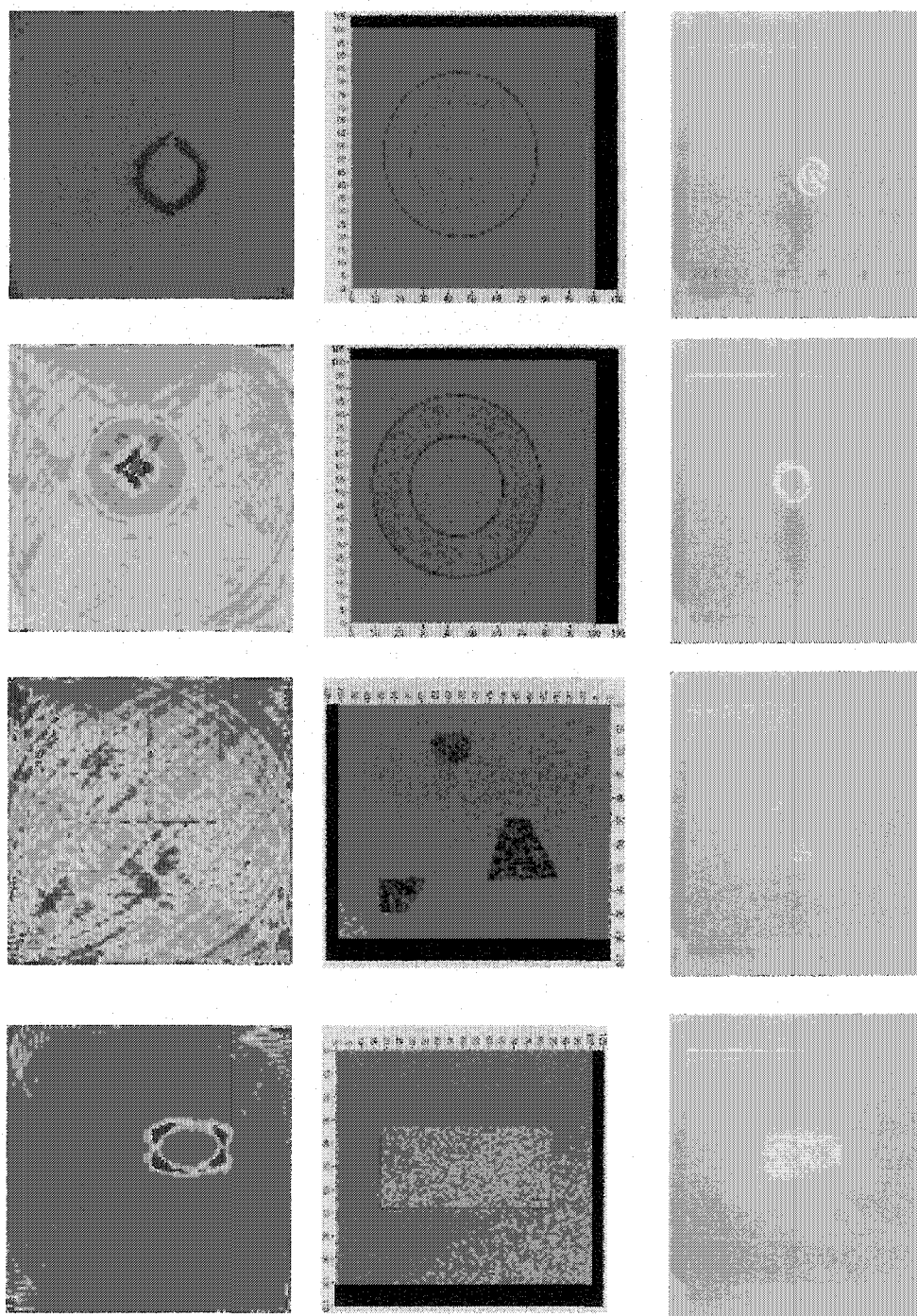


Figure 3-17. LWT blind study results for plates 9-12. The first column is the LWT tomographic reconstruction, the second column is the C-scan results and the third column is a photograph of the actual sample. Note that the scanned area is smaller than the actual plate size.

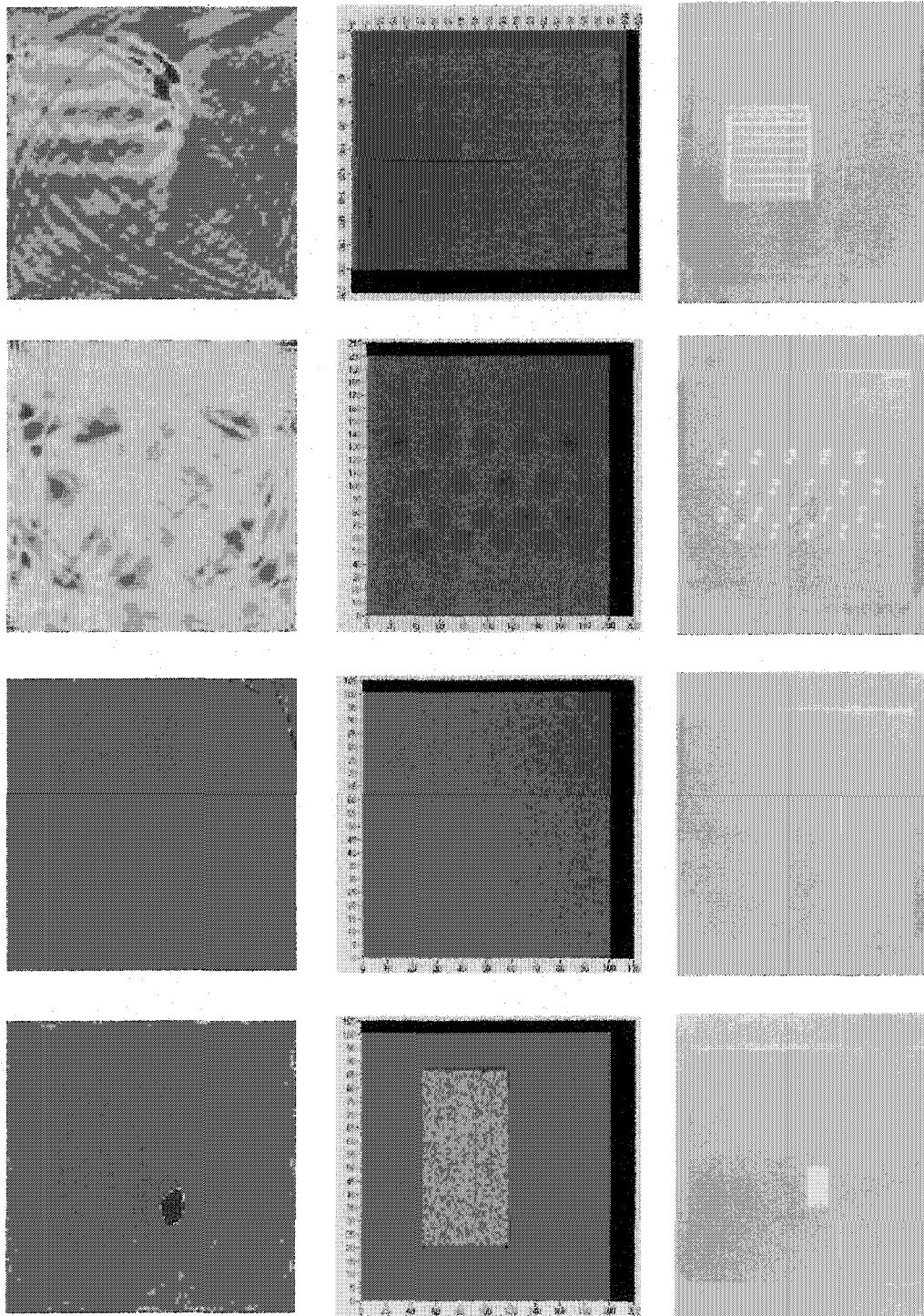


Figure 3-18. LWT blind study results for plates 13-16. The first column is the LWT tomographic reconstruction, the second column is the C-scan results and the third column is a photograph of the actual sample. Note that the scanned area is smaller than the actual plate size.

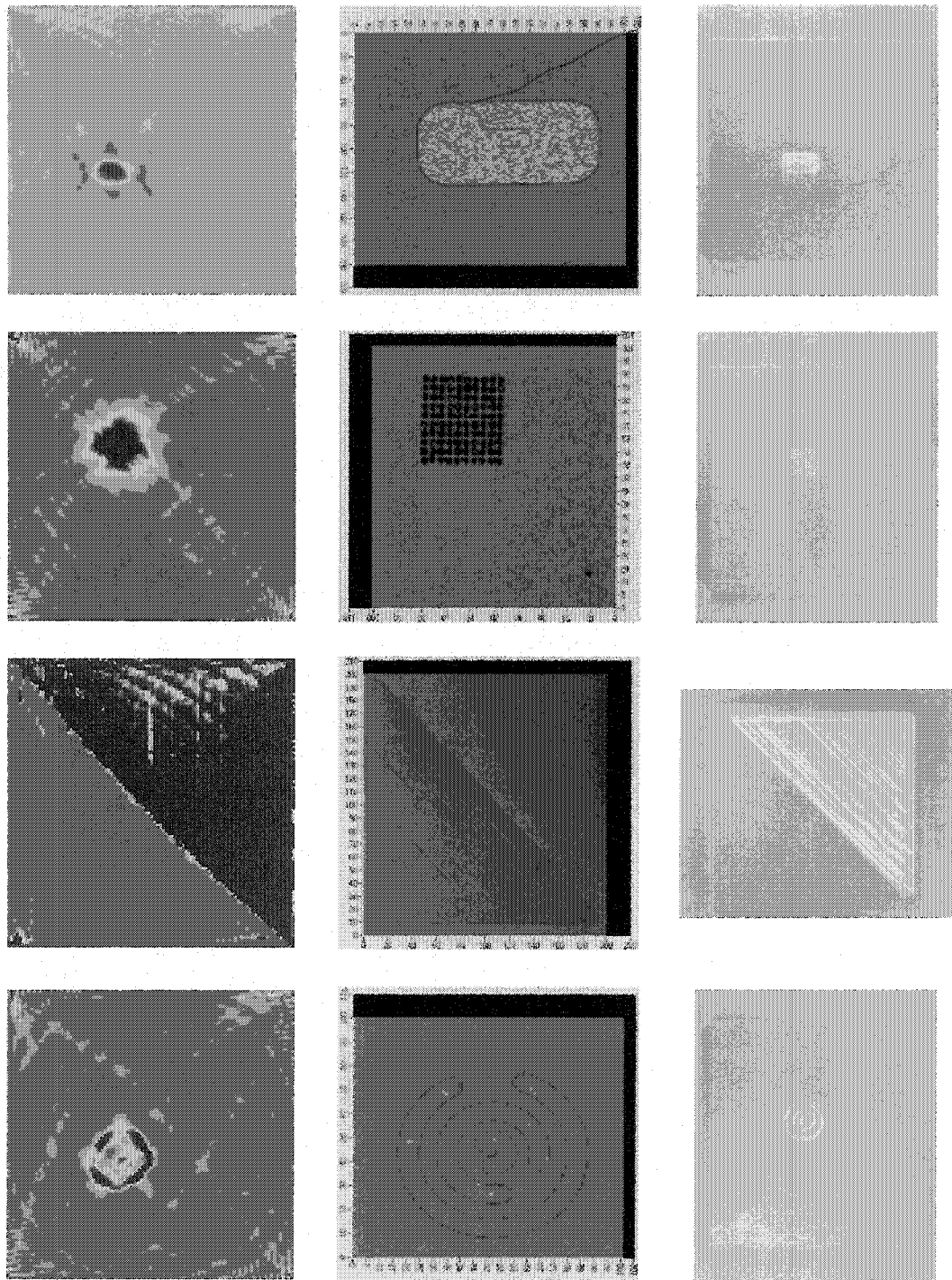


Figure 3-19. LWT blind study results for plates 17-20. The first column is the LWT tomographic reconstruction, the second column is the C-scan results and the third column is a photograph of the actual sample. Note that the scanned area is smaller than the actual plate size.

- Plate #4: Three intersecting channels at all the same depth – 20% thickness loss. All but one of the evaluators got this exactly correct.
- Plate #5: 2-inch dished-out circle, flat on the bottom with a maximum thickness loss of 60%. Although all found this, there was disagreement about the details of the flaw shape.
- Plate #6: 1”x 2” rectangular thinned region with rounded corners, 10% thickness loss. Most thought this was a racetrack groove.
- Plate #7: Two narrow slots “1” and “)”. Several thought there was a flaw connecting the two slots, but all found the two psuedo-cracks.
- Plate #8: Rectangular perimeter channel, a bit wider at top, ~2% thickness loss, and deeper on right side and bottom, 30% thickness loss. All interpreted this correctly.
- Plate #9: 2” flat-bottom circular hole similar to Plate #1, but with only 10% thickness loss. Some thought this was a perimeter groove. A few thought it was diamond shaped.
- Plate #10: Milled circular ring, 55% thickness loss. There was disagreement about the details of this flaw’s structure, but some got it right on.
- Plate #11: Three geometric shapes made by center- punching the plate numerous times. Only one person got this correct. Some guessed that there was not a flaw based on the gray scale image even though more variation was clear in some of the false-color images.
- Plate #12: Rectangular flat-bottom hole, 10% thickness loss. Most called this a channel.

- Plate #13: Square thinned region, 25% thickness loss at the top and nothing at the bottom. The left edge of the flaw is outside of the scanned region. One evaluator got this correct, but others disagreed about the flaw details even though all located and sized it correctly.
- Plate #14: Four unevenly spaced rows of five flat bottom holes of the same size. However, each hole had a different depth – ranging from 3% thickness loss to 50% thickness loss. This was tough to interpret, although most noted the deepest flaws. There was no consensus other than to go back over this region and look more closely.
- Plate #15: Plate with no flaws. This fooled two evaluators who insisted on reading something into the images. This demonstrates how our knowledge or preconceptions on what flaws exist, or should exist, can cause one to see something in the reconstructed images that isn't there. This example also shows the importance of the double-blind test.
- Plate #16: A rectangular thinned region like in Plate #12 but vertical – 45% thickness loss. Everyone called this flaw correctly.
- Plate #17: Like Plate #6, a rectangular thinned region 1”x 2” with rounded corners, but with a 45% thickness loss. Most called this flaw the same as Plate #16 but horizontal and sized it spot on.
- Plate #18: Matrix of small pits with one off all by itself. Nobody found the extra pit, but all located and sized the cluster of flaws correctly.
- Plate #19: Triangular thinned region that covers half of the plate with a 45% thickness loss. Not surprisingly, everybody assessed this plate correctly.

- Plate #20: Three interlaced ¼-inch wide C-channels. Each channel had a 15% thickness loss. Some called this a ring with a flaw in the center. One said it was a “bulls eye object.” All sized and located the flaw correctly.

If grades were to be assigned for each of the plates based on how well the group called the flaws, the results would be as follows: plates 16 and 19 spot on, **A+**; plates 1,3,4,8,17 and 18 correct, **A**; plates 2,5,6,7,9,10,12 and 15 pretty close, **B+/A-**; plates 13 and 14 only a few got right, **C**; plate 11 only one evaluator got right, **D**. Except for a couple of cases, all of the flaws were located and sized correctly. Because the interpretation was done without any training, it is not realistic to expect the structural details of the flaws to be interpreted correctly. Some rules of thumb about which indications in the reconstructions correspond to which physical flaws have been developed, but the information was deliberately withheld from those viewing the images. Nevertheless, this was a successful exercise. In NDE it's always dangerous to construct complicated data processing schemes that successfully highlight flaws which are already known to be present. It is important to evaluate the inspection schemes through tests like this double-blind study. In real testing scenarios one needs to be confident that the inspection process will reveal the flaws that are present and minimize the likelihood that flaws will be missed or called incorrectly.

3.5 Chapter Summary

This chapter has discussed the current state of the Lamb wave tomography system. It has shown that the LWT scanner successfully reconstructs various flaws in

thin plates. The method has been applied to both metallic and composite structures with positive results in each case. Furthermore, a blind study was conducted that demonstrated that the reconstructed images can be interpreted reasonably well for most types of flaws.

However, one major limitation to the current technique is that it can only scan flat plate-like materials. It is capable of scanning slightly curved structures such as aircraft, but would be ineffective for scanning pipes or other cylindrical geometries. A tomographic guided wave inspection technique similar to the LWT system that could interrogate underground and/or insulated piping systems would allow inspections of these structures to be performed quicker and with less cost.

Another drawback of the LWT system is that the time extraction algorithms used to make the reconstructed images only record information about the first arriving mode. A wealth of information still exists in the rest of the signal that is being discarded. The various Lamb wave modes interact with flaws differently because of their different displacement properties. Certain modes may not even interact with certain flaws. Furthermore, as we will see in the next chapter, a tomographic system for pipe-like geometries does not have the ability to access the sample from all sides as in the multiple crosshole system. This means that the ability to extract arrival times for multiple modes becomes even more crucial.

Chapter 4

Helical and Meridional Ultrasound Tomography

4.1 Introduction

Early work on guided waves in hollow cylinders was done by Gazis [77, 78] and others [79-82]. Many authors have recently used guided ultrasonic waves to detect flaws in pipes and tubes [33, 35, 36, 38, 55, 83-96] by generating selected modes in the pipe and using pulse-echo measurements to locate flaws. Alleyne *et. al.* [55, 88, 89] have shown that a ring of transducers around the circumference of the pipe can excite axially symmetric modes that propagate long distances down the pipe. By choosing a non-dispersive mode they are able to retain the signal's shape and amplitude as it propagates even over tens of meters. Similarly, Rose *et. al.* have explored several methods for selecting different modes for pipe inspection. These methods include using comb transducers [33, 35] to generate longitudinal guided waves and partial circumferential loading [36] to focus flexural modes to the area of interest in the pipe. Other authors have also explored the use of EMATs [91-93] and PVDF [38] transducers.

Ultimately, any real world guided wave measurement scheme is complicated by mode conversion at flaw interfaces, bends in the pipe, and loading on both the inner and outer diameters [83, 94-102]. This inherent complexity makes guided waves very informative but at the same time very difficult to utilize. As previously mentioned, an

infinite number of modes are theoretically available for use, each with characteristic dispersion and through-thickness displacement properties. At low frequencies, longitudinal, torsional and flexural pipe modes dominate, but at higher frequencies these structural modes are less important and the waves see the pipe more like a curved plate or shell. It is these helical guided wave modes that are of interest in our measurements. This is in contrast to previous work done in this area that has mostly concentrated on generating the longitudinal and flexural pipe modes. Of course, it is important to point out that there is no clear demarcation between these two perspectives once the guided waves are non-axially symmetric, as in the work by Rose [37]. There are a wide variety of models appropriate for studying this type of vibration in cylindrical shells [103-107]. Many of these studies come from the underwater sound community. One can even formally connect plate waves to the corresponding Lamb-like waves in cylindrical shells by replacing the source by a periodic array of equivalent “unwrapped” two-dimensional plates [108]. A quite useful review of guided ultrasonic waves with an extensive bibliography can be found in [3].

Because the helical modes can be considered similar to Lamb wave modes in plates, our previous work on Lamb wave tomography [9, 10, 12-14, 17] can be extended to pipe inspection systems. The Lamb-like guided waves form a series of helical crisscross paths that are a useful tomographic geometry equivalent to what seismologists use in cross-borehole tomography [4]. However, even though pipe-like objects behave similarly to curved plates, some additional complexities arise because the plate curves around upon itself. First, the pipe as a whole can exhibit three families of propagating modes: torsional, flexural and extensional. These can be distinct from, or intermingled

with Lamb wave (plate) modes in the pipe. Generally speaking, we can hope to distinguish pipe modes from plate modes in that the former are lower in frequency and longer in range, while the latter are higher frequency more localized phenomena. Of course, there is no clear demarcation between the two regimes, and in practice it's necessary to be able to sort them out or the unwanted modes will corrupt the data sets of interest.

A second complexity that arises for pipe-like objects is that more than one helical mode can travel between any two transmit and receive locations. In Figure 4-1 we show schematically a pair of transducers on a pipe and several helical rays propagating away from the transmitting transducer. One of these rays takes the most direct path part way around the pipe to the receiving transducer while others, with steeper helical paths, will travel further around the pipe circumferentially and will miss the receiver. Others will make one or more complete loops around the pipe and end up at the receiver. Although one could envision "aiming" the waves in a narrow enough beam to avoid this confusion, tomographic considerations require that both the transmitting and receiving transducers be as omni-directional as possible. This adds yet a third complexity; the helical waves are generated in both clockwise and counter-clockwise directions. Depending on the relative angular positions of the transmitting and receiving transducers, these pairs of modes may interfere either constructively or destructively.

Furthermore, these complexities are what exist in the absence of any flaws. Flaws scatter the guided waves, and even cause energy to be converted from one mode to another. These effects can be severe for strongly scattering flaws such as cracks. Thickness changes due to corrosion or gouging can cause some modes to be cut off,

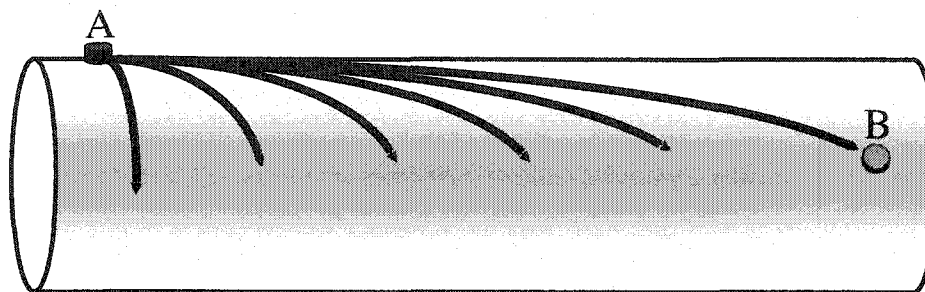


Figure 4-1. A pipe segment is shown with a transmitting transducer at **A** and a receiving transducer at **B**. Six helical Lamb wave paths are shown, with the shallowest only just beginning to wrap around the pipe before it reaches **B**. The steepest possible helical path is the circumferential guided wave that will ring around the pipe and be received at **A**. In between these two extremes, the guided wave modes are launched at **A** for all helicities (4 are shown). Some of these will travel around the pipe one, two and more times before being recorded at **B**. Note also by symmetry that all of these waves go in the opposite directions as well, and these mirror modes will, in general, overlap with the guided wave modes of interest at the receiver **B**.

resulting in a fairly dramatic reflection of those same modes or a dumping of the wave energy into other modes. In the previous Lamb wave diffraction tomography work by Malyarenko et. al. [14, 17], techniques were developed to handle much of this complexity, and we have now found that we are able to deal satisfactorily with the added difficulties of pipe-like geometries.

If we envision a series of pitch-catch helical guided wave measurements on a pipe with a large number of transducers along two separated circumferential belts, the set of helical crisscross ray paths is a single crosshole tomographic geometry. In Figure 4-2 we illustrate this by showing a pipe that is “unwrapped” and laid flat. The transducers now lie along parallel lines and form our traditional crosshole geometry. Because the helical waves can wrap around the pipe more than once, we can consider the “extra” regions and the longer ray paths to give better tomographic reconstructions because these rays pass through the region of interest from additional angles. Figure 4-3 shows the SIRT tomographic grid used to make reconstructions with the HUT system. It shows that the HUT technique is similar to the LWT technique discussed in Section 3.4. The biggest difference is that rays can wrap from the “top” of the grid to the “bottom”.

One of the fundamental limitations of HUT is that the wave vector coverage is incomplete because rays don't go through the region of interest from all angles. To improve reconstructions in cross-borehole tomography, seismologists often place a line of receivers on the surface of the ground between the boreholes. For tomographic plate inspection with Lamb waves, we mimic a four-sided square perimeter array so that the rays pass through the sample from all angles. Neither of these options are available for an inspection scheme capable of examining a wide variety of piping systems. The

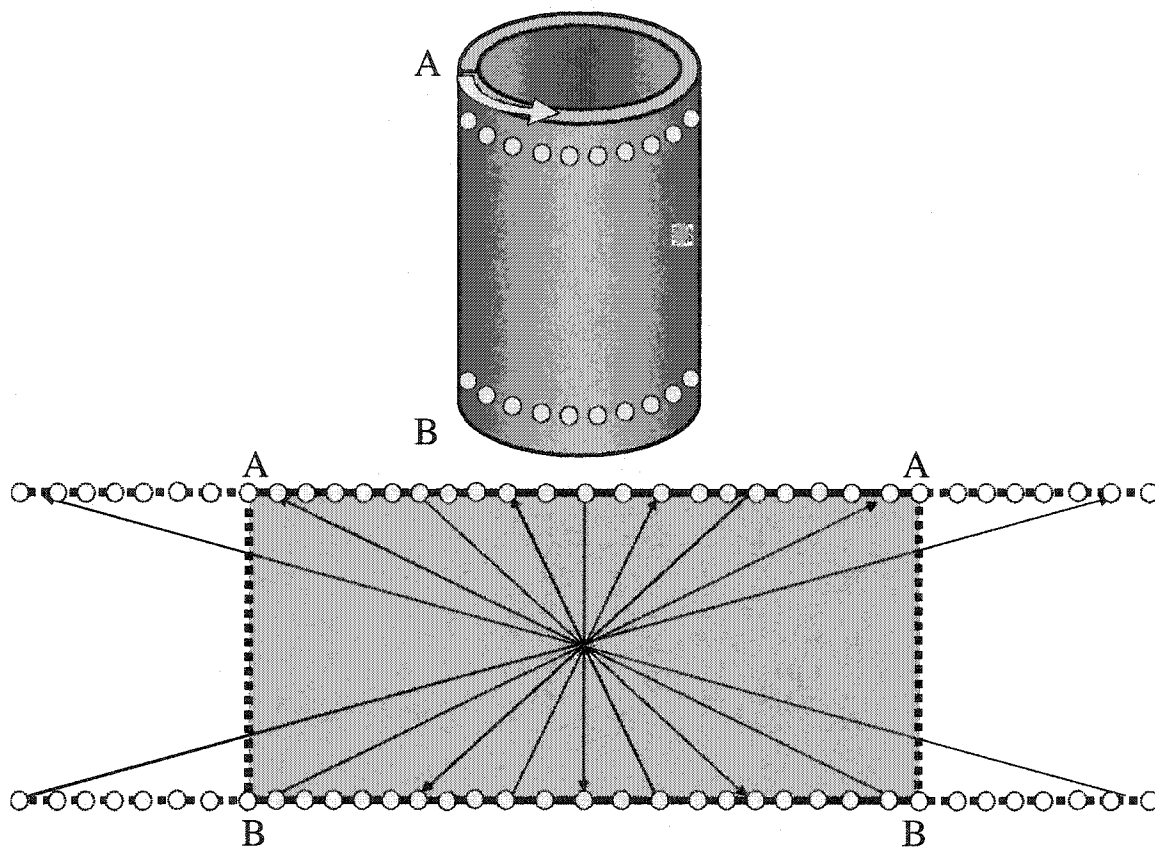


Figure 4-2. Parallel circumferential arrays of transducers can be seen to give a crosshole tomographic geometry when the pipe is mentally “unwrapped.” The two identical parallel circumferential array-belts of transducers wrap around the pipe. Each transducer in one belt transmits helical Lamb waves, which are received by all of the transducers in the other belt. Mentally break the pipe longitudinally along the line AB and then unwrap the pipe to lie flat. The circumferential belts of transducers now lie along the lines AA and BB in the “unwrapped” pipe. Note that the Lamb waves travel along the crisscross rays shown above. Because the helical waves can wrap around the pipe more than once we can consider the “extra” regions to the left and right of AB. These longer ray paths give better tomographic reconstructions because they pass through the sample from additional angles.

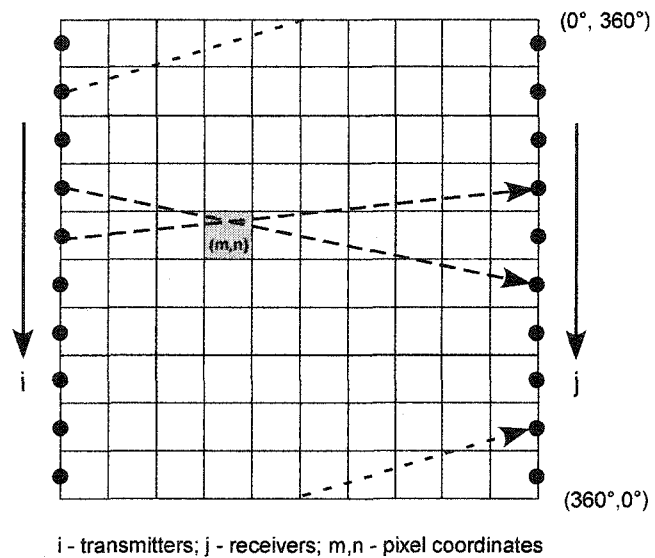


Figure 4-3. Helical ultrasound tomography (HUT) reconstruction geometry unwrapped. Transmitter and receiver locations are shown by the filled circles along the left and right sides. Three different ray paths are shown via the dashed-line arrows.

initial proof-of-concept work on HUT only employs the first arriving modes, but as this work will show, the desire for better reconstructions will require the use of the “twice around” steep helical modes as well. Without using this additional information the quality of the tomographic reconstruction is poorer because the wave vector coverage is not as universal.

4.2 Description of Setup and Experimental Results

We have constructed the apparatus shown in Figure 4-4 to mimic two circumferential belts of transducers via a pair of transducers incremented by stepper motors under computer control. For each transmitter location a guided wave is launched by exciting the contact transducer with a toneburst while the receiver is paused briefly in turn at all of the circumferential positions to catch the various helical Lamb waves. The frequency of the toneburst is chosen to select the guided wave modes of interest. Typically these are the first arriving modes in order to minimize complications from the overlapping modes. The recorded waveforms at each pitch-catch pair location are digitized and processed on the computer to extract the arrival times of the modes of interest. Additional projections are taken by stepping the transmitter in turn to all of the circumferential positions and repeating the process. The complete set of arrival times or amplitudes are then passed to the tomographic reconstruction codes.

A block diagram of the data acquisition equipment for the HUT scanner is also shown in Figure 4-4. The MatecTM TB-1000 PC ISA board is used to form a tone burst of typically 3-15 cycles. In the proof-of-concept reconstructions shown in this chapter, a 15-cycle 1.35 MHz tone burst drives the transmitting transducer, a 2.25 MHz center

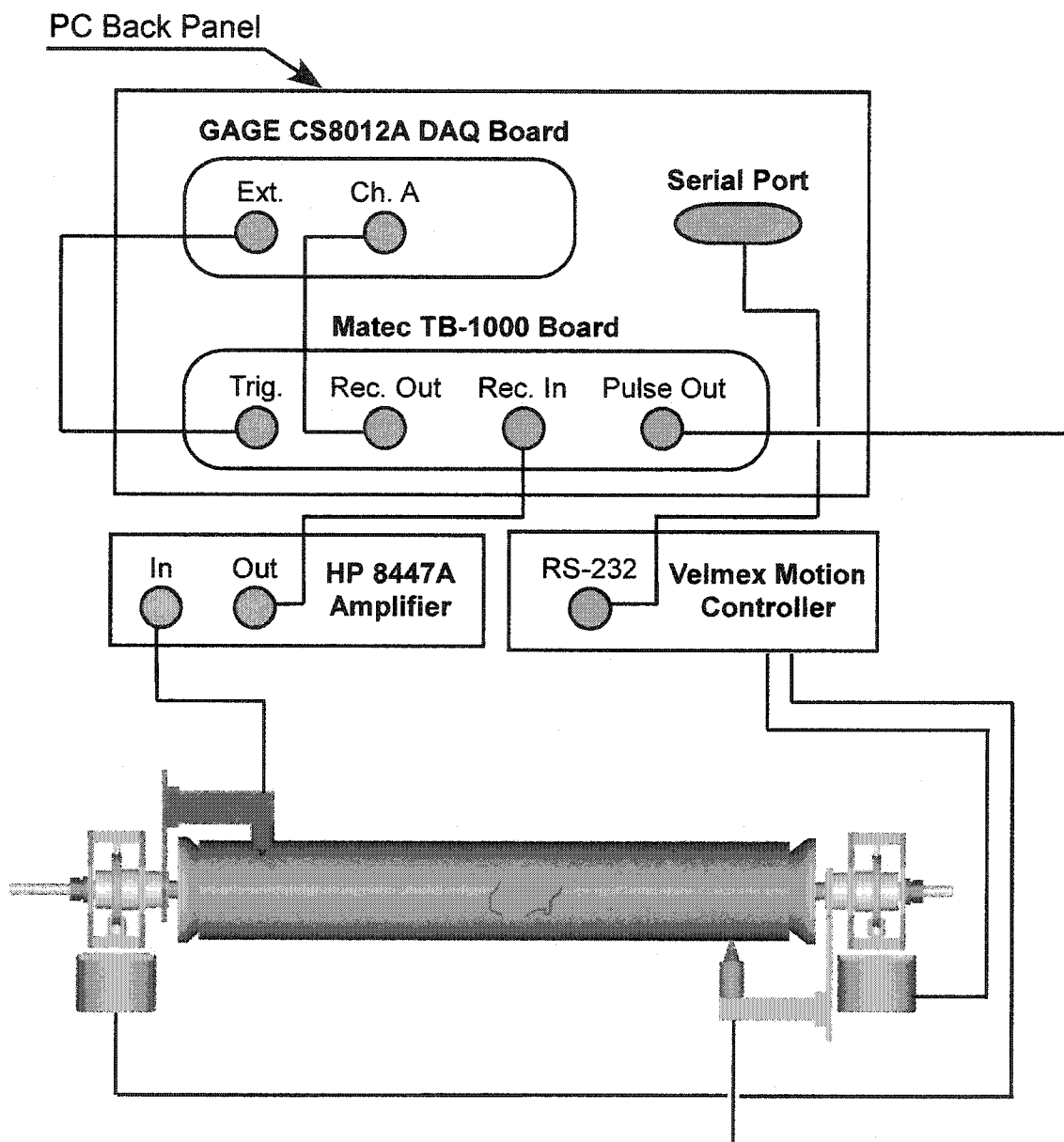


Figure 4-4. Data acquisition block diagram for the computer-controlled HUT scanner.

frequency broadband longitudinal transducer, which excites Lamb-like guided wave modes in the pipe. The received signal is amplified, filtered, and then digitized by a GAGE™ CS8012A DAQ board with 12-bit resolution and up to 100 MHz sampling rate. Initial testing was performed on a steel pipe with an ID of 75 mm and an OD of 102 mm. The distance between the transmitter and receiver at the same circumferential angle was 320 mm. Glycerin was used as a coupling agent because it provides good transmission of the signal and has a high enough viscosity that the coupling was consistent even on the bottom side of the pipe. Each recorded waveform was gated around the first arriving mode, which in our measurements was the S6 mode. For this pipe sample, the S6 mode has a group velocity of a little over 5 mm/μs and is reasonably dispersive. At this frequency-thickness product, the next fastest modes have a group velocity of less than 4 mm/μs and there are also several slower modes present in the 3 – 4 mm/μs range. Therefore, these slower modes are effectively separated out from the first arriving S6 mode. After the data was recorded, the arrival time for the fastest mode was extracted for each waveform and then sent to the tomographic algorithm described above. The resulting reconstructions are shown below. All of the signal processing algorithms are fully automated since the large number of individual pitch-catch measurements precludes any manual analysis of the waveforms. The waveform processing, mode-extraction and tomographic reconstructions take only a total of a few seconds on a modern PC running Linux.

Figure 4-5 is a photo of a thick-walled steel pipe sample with a flat rectangular thinned region. This sample, with four different lengths of flaws, was used for the proof-

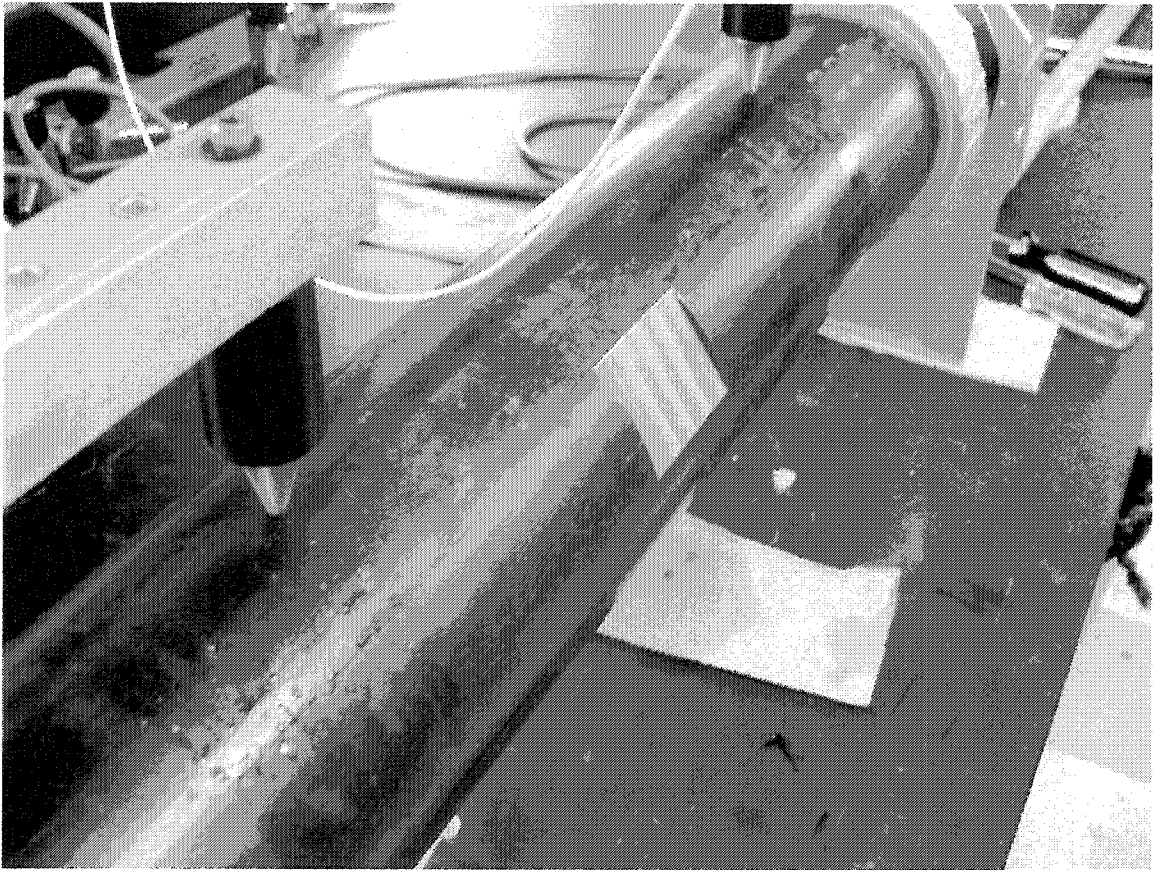


Figure 4-5. A thick-walled steel pipe sample is shown in the HUT scanner. Conical delay lines are used on the spring-loaded contact transducers in order to minimize their footprint.

of-concept tests. The thickness loss in the flaw is about 25% at the center of the rectangular flawed region, with only a surface slope change – but no thickness change – at the circumferential edges and a varying-depth step discontinuity at the axial edges. The photo also shows the conical delay lines that are used with the contact transducers in order to minimize their footprint.

Figure 4-6 shows four reconstructions for increasing length of the rectangular thinning. The red horizontal striations and the blue crisscross artifacts indicate the location of the flaws, which can be seen to increase in size as expected. Note that the scans are shown for different angular positions of the flaws, but all other scanning, reconstruction and rendering parameters are remained unchanged. No additional image processing or enhancement has been done to improve the appearance of these “raw” reconstructions.

4.3 HUT Discussion

Figure 4-7 shows two scatter plots of the velocities extracted by our algorithms versus ray number for the unflawed pipe. Figure 4-7(a) and Figure 4-7(b) were scanned at 0.5 MHz and 1.35 MHz respectively. A “perfect” result would show all of the velocities in the tight band around 2.9 mm/ μ s (Figure 4-7(a)) and 5.2 mm/ μ s (Figure 4-7(b)) – the velocities for the mode of interest at the two frequencies. Several things can be noted from these plots. First, in Figure 4-7(a) two clusters of rays gave points between 3.2 and 3.4 mm/ μ s. This most likely means that the threshold used to ignore the early pipe modes was not completely successful. Second, a weak, regular structure that has two dips is also apparent. This seems to occur when the two counter-helically

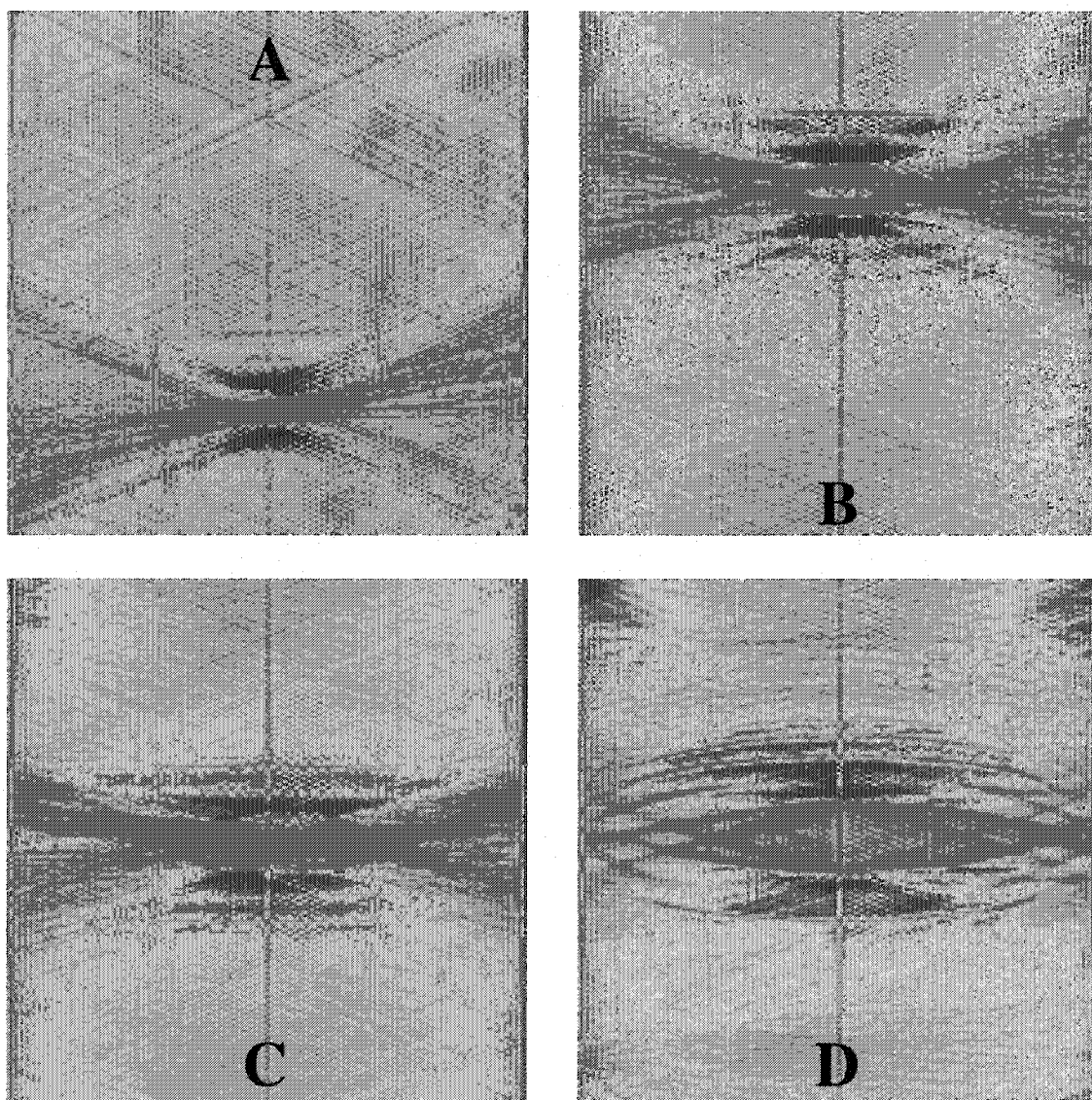


Figure 4-6. Four reconstructions are shown for increasing flaw size in the same thick-walled steel sample shown in Figure 4-5. The red horizontal striations and the blue crisscross artifacts indicate the location of the flaws, which can be seen to increase in size as expected. The vertical axis is in the circumferential direction, while the horizontal axis is the axial direction.

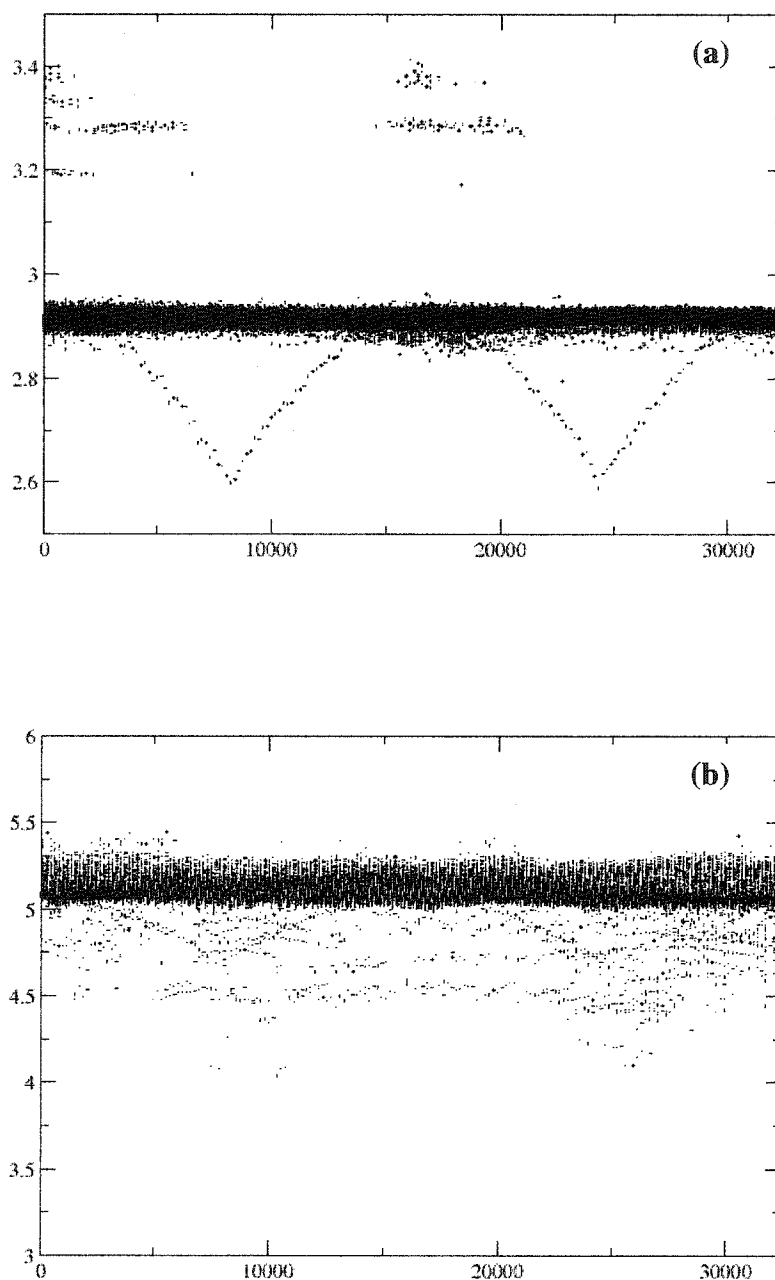


Figure 4-7. Velocity scatter plots – velocity (mm/μs) vs. waveform # – for the 180 x 180 (32400) recorded waveforms are shown for unflawed pipe samples at (a) 0.5 MHz and (b) 1.35 MHz. If our arrival-time extraction algorithms were perfect, all of the data points would be clustered about the dark horizontal bands.

propagating mode pairs destructively interfere (pitch and catch directly opposite each other) and thus the mode of interest zeros out. Consequently, our algorithms then pick out the arrival time of an unwanted mode. Finally, in Figure 4-7(b) the cluster of points seen in Figure 4-7(a) at higher velocities due to the unfiltered pipe modes are not present. This is because the second scan was taken at a higher frequency. However, we see a fair number of points where our algorithms appear to have missed the mode of interest and instead picked out later arriving modes below the dark band at about 5.2 mm/ μ s.

Figure 4-8 shows two typical waveforms recorded by our system on the unflawed pipe sample at 1.35 MHz. The first waveform is for a meridional wave where both transducers are at the same angular position. The second was recorded when the transducers were rotated 90° from one another. Both are single-shot without averaging and are shown on an arbitrary amplitude scale that is consistent between them. Note that the signals do not have distinct, isolated modes, and that the essential character of the waveforms changes dramatically throughout the measurements even in the absence of flaws. We have tended to use the subtle arrival-time changes in the first-arriving modes because coupling variations in our laboratory scanning system introduce uncertainty into amplitude measurements and because our standard mode identification and extraction algorithms are not robust enough to deal with the later-arriving mutually overlapping modes.

The four graphs in Figure 4-9 show velocity scatter plots for the flawed samples reconstructed in Figure 4-6. For those rays that do not pass through the flaw, the velocity is constant, and our algorithm reliably returns a velocity in the band just above 5 mm/ μ s –

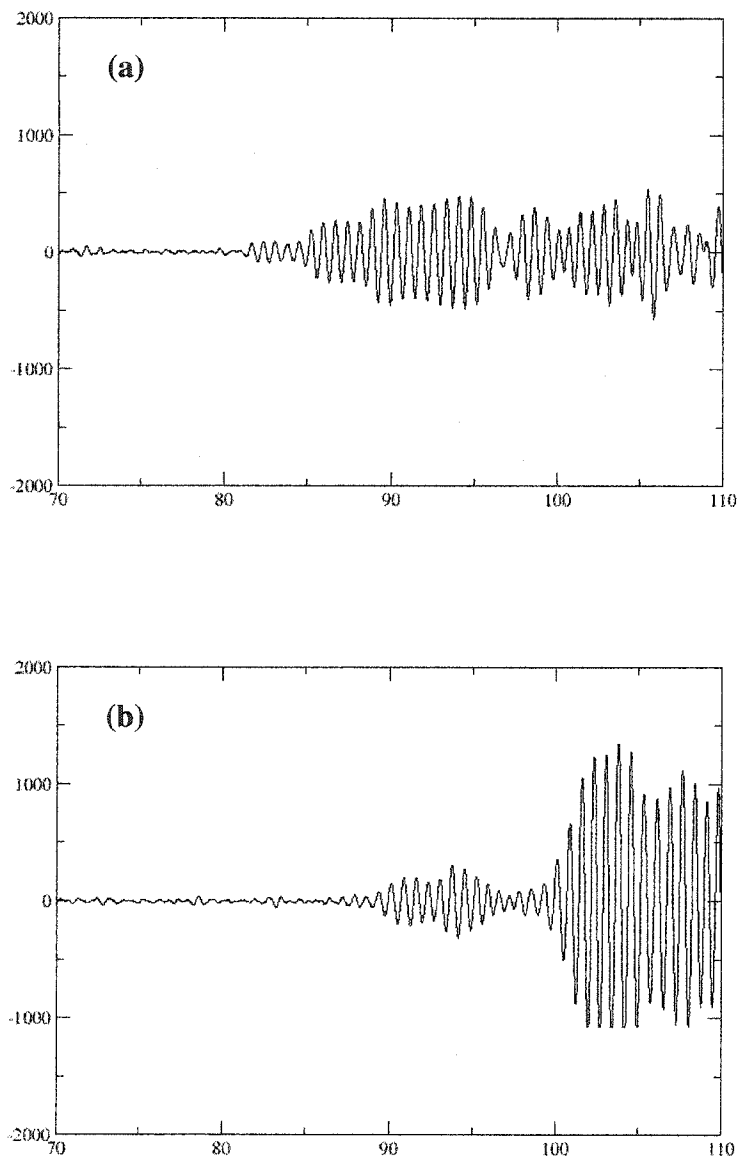


Figure 4-8. Typical waveforms for (a) meridional and (b) helical guided wave modes.

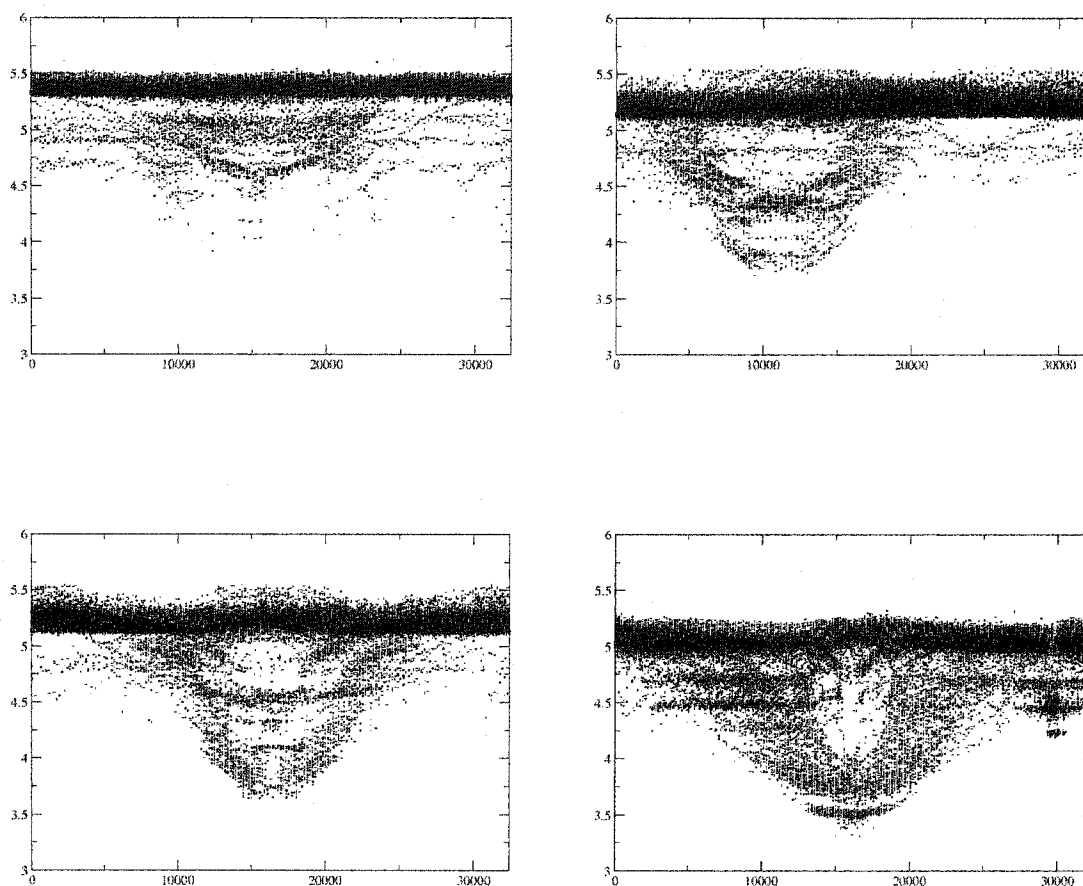


Figure 4-9. Velocity scatter plots are shown for flawed samples from Figure 4-6 as extracted by our algorithms versus ray number. The rays that pass through the flaws indicate a different velocity because the modes we select for our measurements are dispersive: *instantaneous speed varies with local pipe wall thickness*. A second effect also occurs at the strongly scattering edges, which is a diffraction of the guided wave modes. This also often shows up as a slowing, because the waves tend to skirt around the flaw and hence take an effectively longer path from transmitter to receiver.

the theoretical velocity of the S6 mode for these scans. Because the modes selected for these measurements are dispersive, their instantaneous speed varies with local pipe wall thickness and the rays that pass through the flaws indicate a different velocity.

Additionally, diffraction of the guided wave modes occurs at the strongly scattering edges of the flaws. This often shows up as a slowing of the guided wave modes, because they tend to skirt around the flaw and thus take an effectively longer path from the transmitter to the receiver. Figure 4-10 shows the tomographic velocity profiles for three different horizontal slices through Figure 4-6(b). It can be seen that outside the flaw the velocity remains close to the background level of about 5.25 mm/ μ s. The other two profiles are taken from different locations within the flaw. Figure 4-10(c) shows how the waves have an apparent decrease in velocity as they encounter the edges of the flaw and an increase in velocity at the center of the flaw where they do not interact with strongly scattering edges.

4.4 Meridional Ultrasound Tomography

In other cylindrical geometries, such as stacked storage tanks, access to the entire circumference of the structure could be impractical or even impossible. This would prevent the application of HUT to these types of structures. However, a tomographic geometry similar to the single crosshole and HUT geometries can be created by placing a line of receivers along the axis of the cylindrical structure. Figure 4-11 shows how the meridional line of transducers forms the desired crosshole geometry.

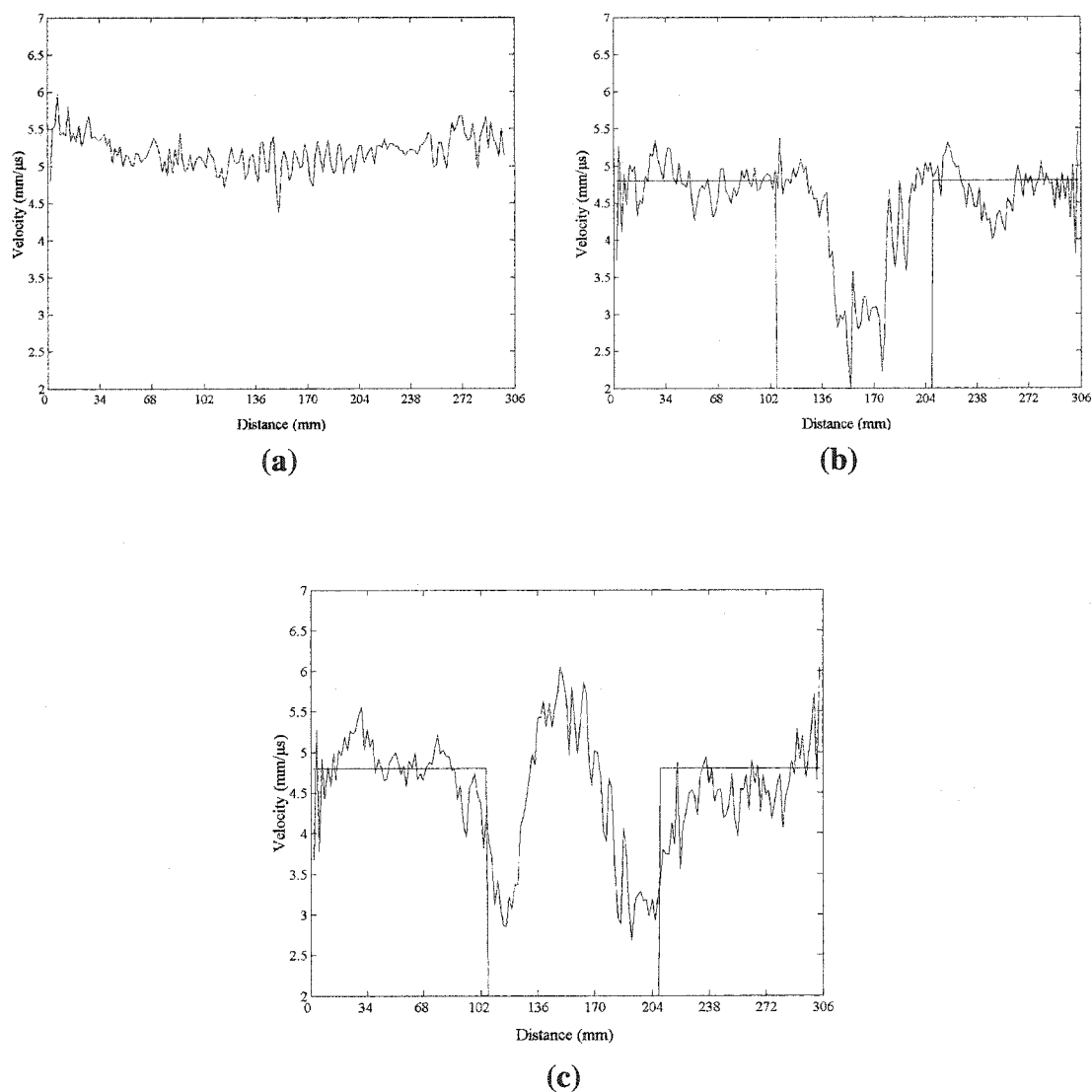


Figure 4-10. Each graph represents a different horizontal cross-section of the tomographic reconstruction shown in Figure 4-6(b). These velocity profiles can be converted to thickness profiles. (a) A horizontal cross-section outside the flawed region. The background level is around 5.25 mm/μs which is to be expected from the unflawed portion of the sample. (b) This horizontal cross-section is taken from the red region that can be seen on the top edge of the flaw in Figure 4-6(b). (c) The velocity profile for the horizontal cross-section directly through the center of the flaw.

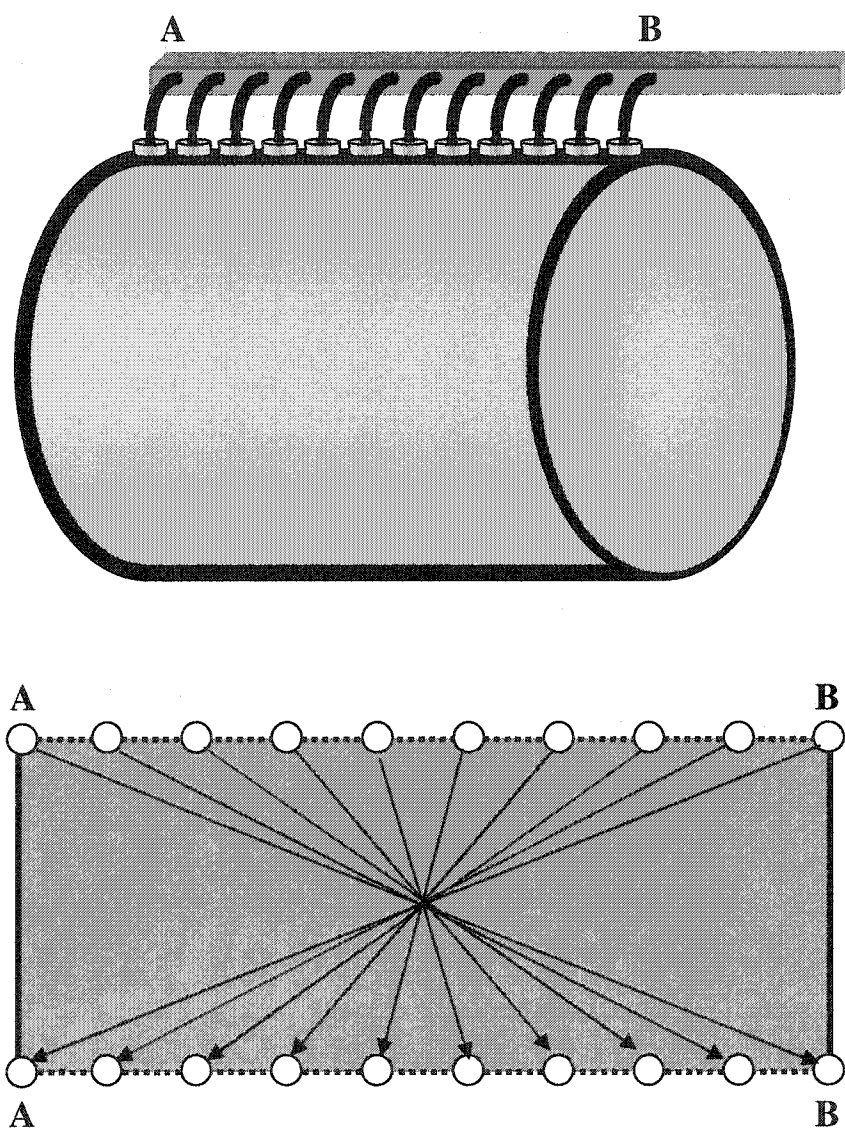


Figure 4-11. A linear array of transducers along the axis of the cylinder can be seen to give a crosshole tomographic geometry when the pipe is mentally “unwrapped”. Each transducer transmits helical Lamb waves which are received by all of the other transducers in the array. Mentally break the pipe longitudinally along the line AB and then unwrap the pipe to lie flat. The linear transducer array can now be envisioned to be on both the top and bottom of the “plate”. Note that the waves of interest are those that travel helically around the circumference of the pipe. The modes that travel directly along the axis of the pipe from the transmitter to the receiver need to be ignored in the signal processing of the recorded waveform.

This new inspection scheme provides excellent opportunities to quickly and quantitatively interrogate previously hard to inspect structures. For example, Figure 4-12 shows how depleted Uranium Hexafluoride (UF₆) tanks are stored. Approximately 50,000 storage tanks containing more than 500,000 metric tons of UF₆ are kept at facilities in Tennessee, Ohio and Kentucky. Some of the cylinders have been standing in the storage yards since 1956 and many have corroded or even failed. The largest cylinder breach resulted in a loss of approximately 110 pounds of material. Furthermore, the Department of Energy's (DOE) Office of Nuclear Energy issued a report in 1990 recommending that these structures be inspected at least twice a year [109]. The bottoms of the tanks are most vulnerable to corrosion, but this area is also inaccessible because of how the tanks are stacked. Also, because of the sheer number of these tanks, in order to adhere to the inspection requirements a tank would need to be inspected every 5 minutes, 24 hours a day, 365 days a year. This necessitates a quick, quantitative and automatic inspection scheme. Meridional Ultrasonic Tomography (MUT) would enable such a process.

Some preliminary signal analysis was done on large storage tanks using the laboratory mockup shown in Figure 4-13. The equipment that was used was unable to generate strong enough signals to travel all the way around the tank. However, the tests that were performed on the tank showed that Lamb-like waves could be generated on curved plates and that they would travel in the helical directions. These results led to the development of a smaller scale system in order to test the MUT geometry.

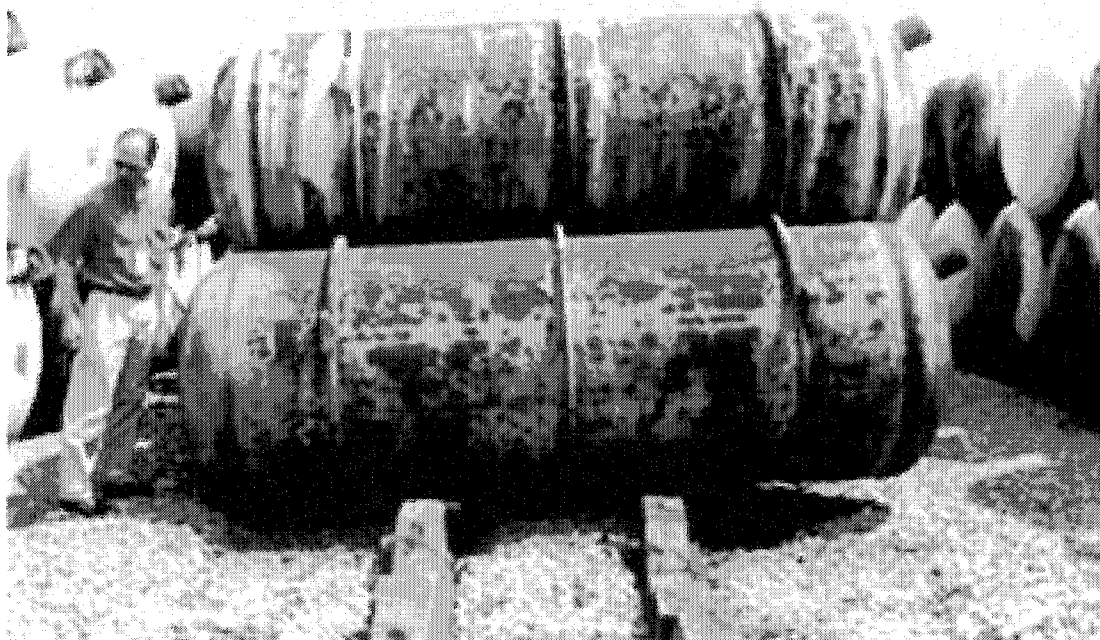


Figure 4-12. Storage facility for depleted Uranium Hexafluoride (UF₆). Tanks are stacked in such a way that HUT would not be a viable inspection technique. Therefore, meridional ultrasound tomography (MUT) is a better solution for this type of inspection. Approximately 50,000 storage tanks containing more than 500,000 metric tons of UF₆ are kept at facilities in Tennessee, Ohio and Kentucky [110].



Figure 4-13. A picture of a large storage tank mockup scanner in the laboratory. The receiving transducer was attached to a linear slider while the transmitting transducer was placed at a single position on the tank. This setup was used to perform preliminary work on guided wave signals in cylindrical structures.

Figure 4-14 shows how the MUT geometry was mimicked in the laboratory. Two linear scanners were angled so that the delay lines of the transducers were in good contact with the surface of the sample and as close together as possible. In a final system, this setup would be implemented as a line array of transmit/receive transducers that would be controlled by a multiplexer. This would allow each transducer to transmit and receive in turn so that a full crosshole geometry could be realized. In the laboratory system, one transducer transmits, while the receiving transducer steps through each of the positions along the axis. Then the transmit transducer steps once and the receive transducer again steps through each position. This is repeated until the transmit transducer has also stepped through each axial position.

The largest barrier to MUT comes in the analysis of the recorded waveforms. Because the transmitting and receiving transducers are along the same line, the portion of the signal that we are interested in is no longer the first arriving. The first arriving mode is the one that goes directly between the transducers along the axis, and not the helically generated Lamb wave that provides the needed tomographic geometry. This is another reason that more sophisticated multi-mode analysis is needed.

4.4.1 MUT Experimental Setup

For the preliminary investigations of MUT, a two foot long aluminum pipe with a thickness of 4mm and OD of 150mm was used. The transmit and receive transducers were 2.25 MHz center-frequency, broad-banded contact transducers with conical delay lines to reduce the footprint of the transducers. The pipe had an irregular flaw on its ID

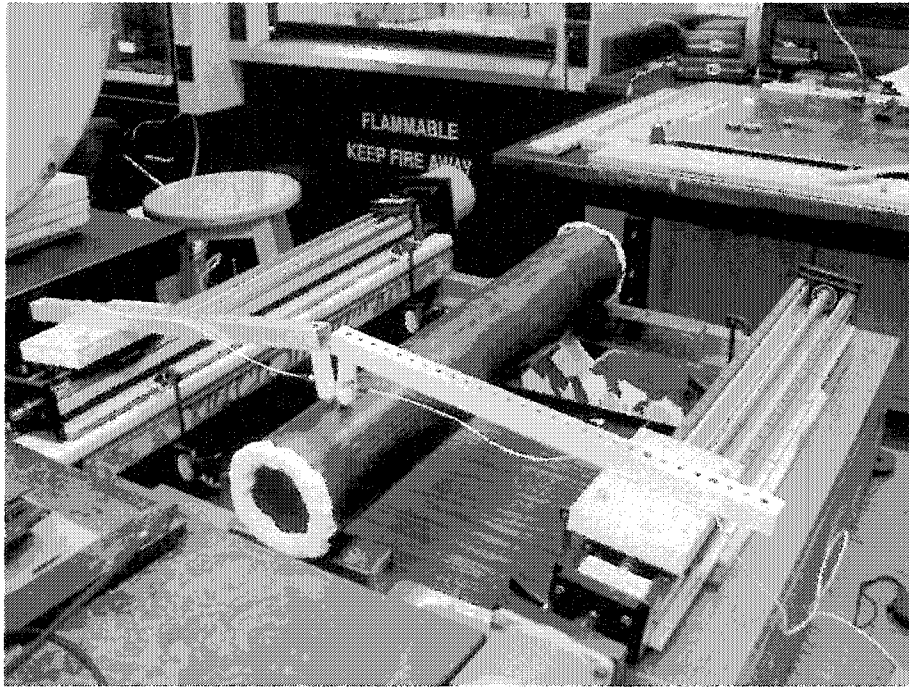


Figure 4-14. The linear array of transducers used in MUT is mimicked in the laboratory by two transducers attached to linear scanners. The scanners are tilted so that the transducers are in good contact with the surface of the pipe. For each transmit position, the transmitting transducer remains in place while the receiving transducer steps through all of the receive positions. The transmit transducers is then stepped and this process repeats until all the pitch-catch combinations have been recorded.

that was created to be representative of either gouging or corrosion. The flaw was approximately 35mm in diameter with no more than 20% thickness loss at its deepest point. The sample was placed in the lab apparatus so that the flaw was within the “sweet spot” of the tomographic geometry – where the ray density is the highest.

Because of limitations in the original arrival time extraction algorithms [6], the reconstruction algorithms were adapted for attenuation data. It was found that with fairly reliable coupling, after gating the signal to cut out the direct wave from the transmitter to the receiver, the amplitude of the helical mode of interest could be extracted. The tomographic reconstruction algorithm is again based on the SIRT and is similar to the velocity SIRT presented earlier. A simple model of attenuation is used where $A = A_0 e^{-(\mu/C)x}$. In this equation, A is the measured amplitude, A_0 is the initial amplitude, μ is the attenuation coefficient, C is a constant that relates the attenuated amplitude to the amount of attenuation, and x is the distance of propagation. For our tomographic geometry this yields:

$$A[i, j] = A_0 \sum_{m,n \in \text{ray}[i,j]} e^{-\frac{1}{C}(\mu[i,j]d[i,j,m,n])} \quad (4-1)$$

where (m,n) is the coordinates of an individual grid cell and $[i,j]$ denotes the ray defined by transmitter position i and receiver position j (see Figure 4-15). Furthermore, the cells are updated as follows (see Section 3.4.1 for full SIRT algorithm details):

$$\Delta\mu_{m,n \in \text{ray}[i,j]}[m, n] = \frac{C \{ \ln(A^{k+1}[i, j]) - \ln(A^k[i, j]) \}}{L[i, j]} \quad (4-2)$$

$$\mu^{k+1}[m, n] = \mu^k[m, n] + \Delta_{AVG}\mu[m, n].$$

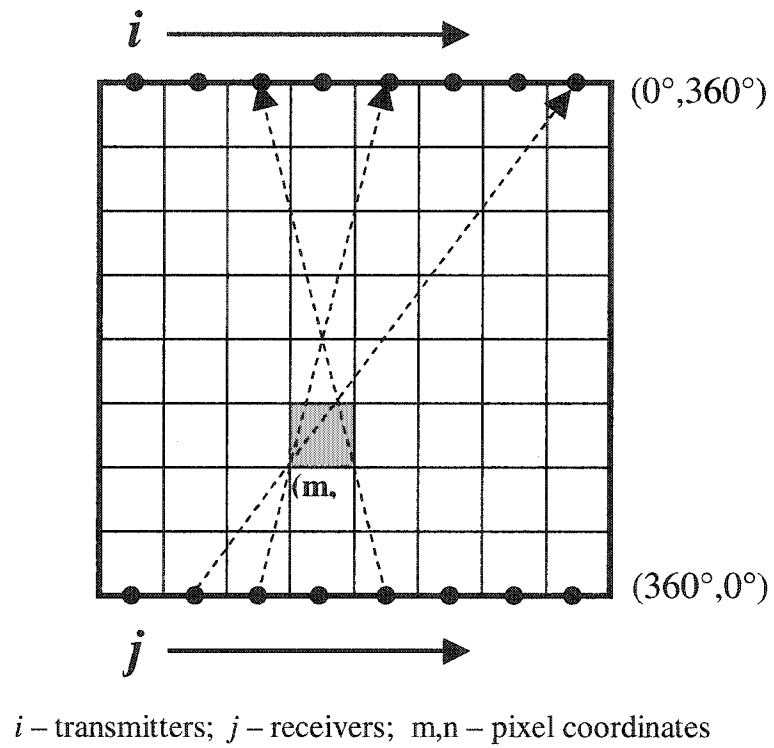


Figure 4-15. Crosshole geometry for meridional ultrasound tomography (MUT). Transmitter and receiver locations are shown by the filled circles along the top and bottom edges. Three different ray paths are shown via the dashed-line arrows.

4.4.2 MUT Results and Discussion

Figure 4-16(a) shows a simulated reconstruction of a flawless sample and Figure 4-16(b)-(d) show reconstructions for simulated square flaws of different dimensions. In these simulations, C was derived so that for $\mu = 1$ and $x = 200$ mm, $A = 0.5A_0$. These reconstructions show that the MUT geometry is able to detect flaws by measuring attenuation data. However, they also show that because the technique only allows for a single crosshole geometry, the flaw boundaries are not well defined. Because the array of transducers are only along the top and bottom of the reconstructed image, we do not get as much information from the horizontal direction. This causes the flaw boundaries to smear in the vertical direction. Ultimately, the reconstructions do show the flaws, and if the smearing effects are known *a priori*, one could effectively inspect materials with the MUT technique.

Guided wave signals from a clean portion of the 4mm-thick aluminum pipe sample are shown in Figure 4-17. Notice that in Figure 4-17(a) the mode of interest arrives around 150 μ s. The beginning of the signal is truncated to save storage space and to eliminate the first-arriving direct signal. The mode of interest for these measurements was the A0 mode (3.06 mm/ μ s) because at the chosen carrier frequency it is a relatively strong mode and since it is not as dispersive, it retains its envelope shape. It can be seen from Figure 4-17 that the mode can be tracked relatively well between scans. Because the mode of interest was not the first-arriving mode and wasn't appreciably dispersive, the signal was gated and amplitude data was used for reconstructions instead of arrival time data.

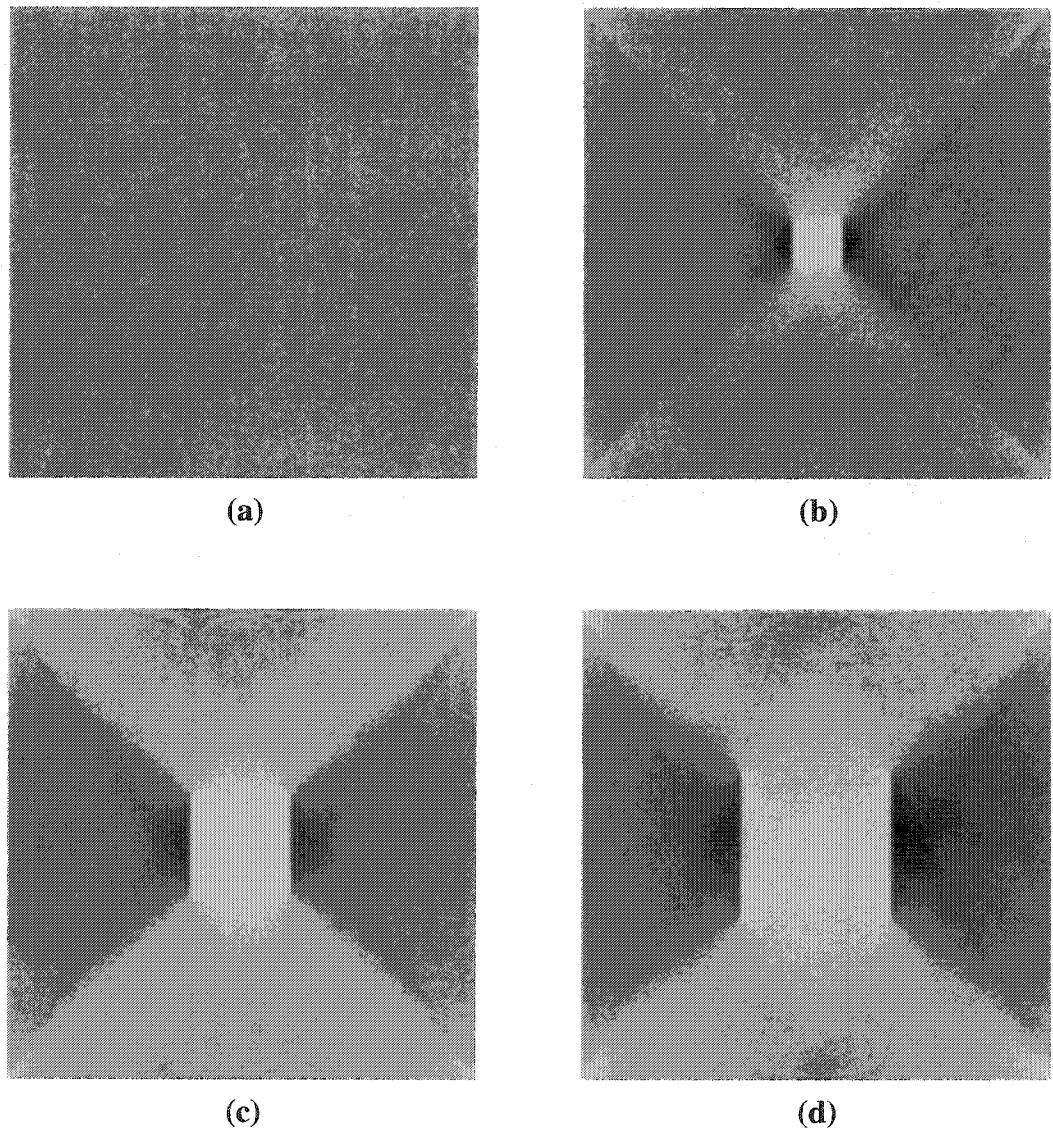


Figure 4-16. Simulated reconstructions for attenuation MUT data. Reconstructed area is 470 mm x 470 mm. The pitch and catch transducers were respectively located along the top and bottom of the reconstructed areas. (a) Unflawed sample. (b) 50 mm x 50 mm square flaw. (c) 100 mm x 100 mm square flaw. (d) 150 mm x 150 mm square flaw.

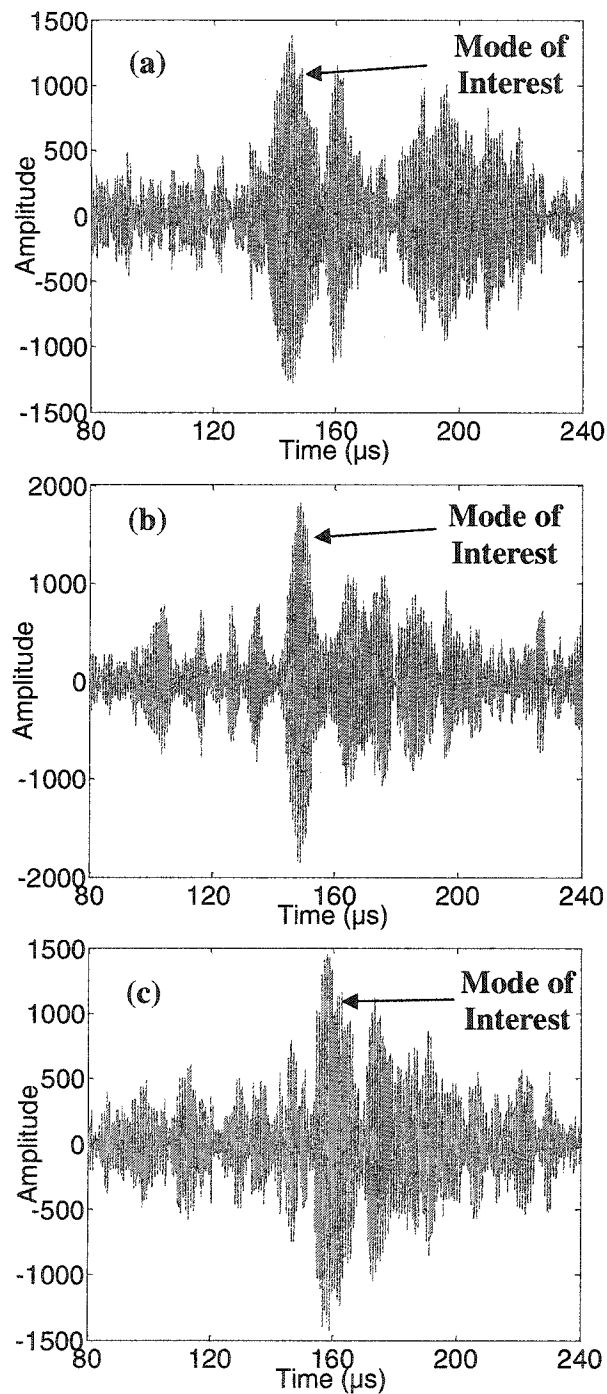


Figure 4-17. Guided wave signals of an unflawed pipe sample for the first transmit position and three different receive positions. The mode of interest is the A0 mode with a velocity of $3.06 \text{ mm}/\mu\text{s}$ (a) Receive position #1, $d = 464 \text{ mm}$. (b) Receive position #40, $d = 471 \text{ mm}$. (c) Receive position #80, $d = 491 \text{ mm}$.

Figure 4-18 shows an example of the extracted amplitude data for both the flawless and damaged portions of the aluminum pipe. Figure 4-18(a) shows the extracted amplitude data for transmitter position #45 – halfway through the scan. It can be seen that the data fluctuates a great deal due to inconsistent coupling. One way that we attempted to control this fluctuation was by measuring a ratio between the amplitude of the first arriving mode and the first-arriving helical mode. However, because of dispersion effects and the longer path length of the helical mode, this method did not provide more consistent amplitude data. Ultimately, a moving average of five points was used to smooth the extracted amplitude data. Figure 4-18(b) shows the smoothed amplitude data for the unflawed portion of the sample. Figure 4-18(c) and (d) show the raw and 5-point smoothed amplitude data respectively for transmitter position #45 – halfway through the scan and approximately at the same axial location of the flaw – for the flawed portion of the aluminum pipe. It can be seen that the amplitude data remains mostly constant for the unflawed scan and that it drops down when the receiver moves into the flawed region.

Reconstructions for the clean portion and flawed portion of the pipe sample are shown in Figure 4-19(a) and (b) respectively. It can be seen that the method clearly shows the existence of the flaw. As expected from the results of the simulated data, the flaw is larger than its actual size and some artifacts exist in both the flawed and unflawed scans due to the SIRT. It can be seen that the flaw is smeared in the circumferential direction due to the single crosshole geometry. However, the technique shows promise to be a powerful NDE technique. With improved multi-mode arrival time algorithms, the reconstructions can be improved and the simple gating can be replaced.

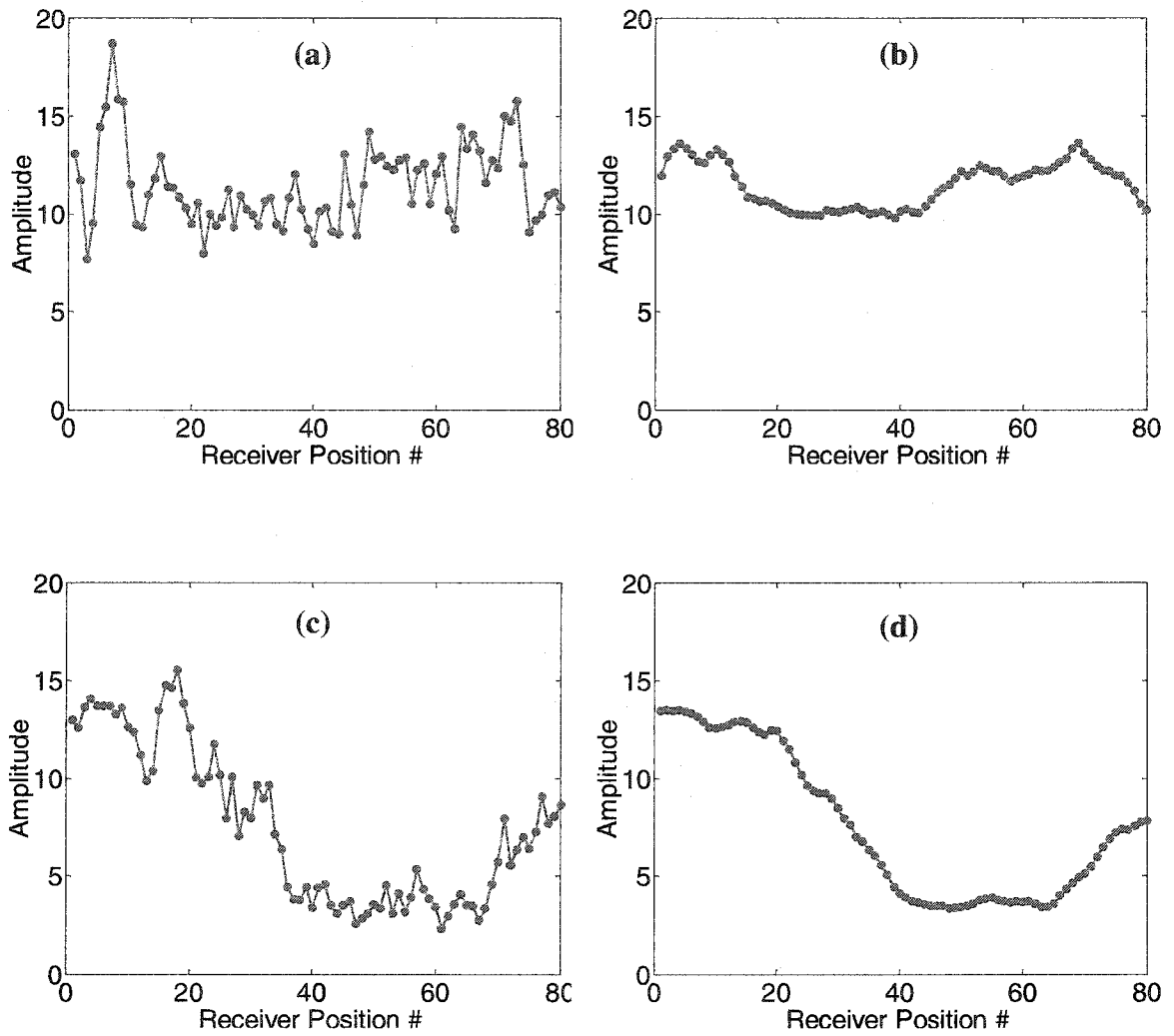


Figure 4-18. Amplitude data for the flawed and unflawed sections of the aluminum pipe sample. (a) Amplitude data for the unflawed section, transmit position #45 (halfway through scan). (b) 5-point sliding average of unflawed amplitude data in (a). (c) Amplitude data for the flawed section of the pipe, transmit position #45. Flaw was an irregularly gouged area with an approximate diameter of 35 mm. (d) 5-point sliding average of data in (c).

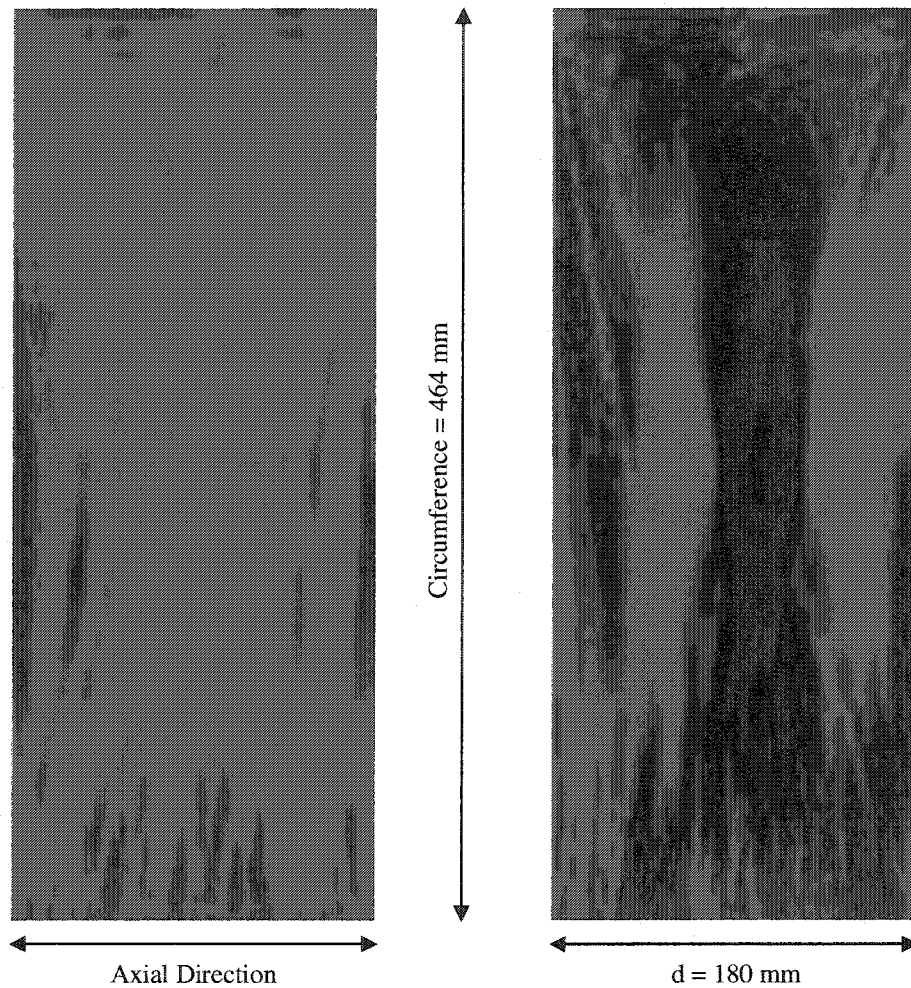


Figure 4-19. MUT reconstructions of an aluminum pipe with an OD = 150 mm and a thickness of 4 mm. The image on the left is a reconstruction of an unflawed portion of the sample. The image on the right is a reconstruction of the flawed region of the pipe. The flaw was an irregularly gouged area with an approximate diameter of 35 mm. The thickness loss within the flaw was 10-20 %.

Chapter 5

Advanced Signal Processing

As we have seen in the previous chapters, guided wave tomography has the capability to quantitatively inspect large plate-like structures quickly and robustly. However, in both the HUT and MUT scanning systems, the single crosshole geometry limits the accuracy of the resulting reconstructions. The current reconstruction algorithms only extract the time-of-flight information for the first arriving mode at a predetermined frequency. In HUT, this limits the ray vector coverage of the different projections. Also, because the pipe and plate-like guided modes are intermixed, the first arriving mode is not always the mode of interest. It can be seen that the arrival times jump from one mode to another at certain times. Both of these factors reduce the quality of the final reconstructions.

Similarly, multi-mode arrival time information could also improve the LWT scanning system. LWT uses a multiple crosshole geometry and does not suffer from the same wave vector coverage problems that HUT does. However, different modes are sensitive to different types of flaws because of their displacement properties throughout the thickness of the material. Arrival time information for later arriving modes can further improve the LWT, HUT and MUT reconstructions.

Furthermore, this work will show that image compounding techniques are also able to improve the guided wave tomographic reconstructions. Previously, only a single frequency has been used when inspecting a material with ultrasonic tomography. In order to choose an appropriate frequency, the dispersion curves for the material were calculated before testing and the received waveforms were visually inspected to ensure that a sufficient signal was generated. However, by recording data for different frequencies at each pitch-catch location in the scanning geometry, additional valuable information can be collected. The frequency “walking” described here has led to improvements in both multi-mode extraction, through a sorting algorithm described below, and the resulting reconstructions through frequency image compounding.

5.1 Tomographic Frequency Compounding

Spatial image compounding is a technique most widely used in ultrasonic B-scanning. Different methods exist that allow B-scans to be taken from slightly different spatial orientations to reduce the amount of speckle noise in the resulting images [111-113]. Speckle noise arises from the constructive interference of backscattered echoes from randomly spaced Rayleigh scatterers in the tissue [114]. By compounding the images from different spatial locations, an effective “averaging” of the individual pixels of the images, speckle noise is reduced and the contrast between the cysts/lesions and the background is increased [115]. This allows for better detection and location of the cysts and lesions. It also increases the potential for automatic detection and level of suspicion (LOS) calculations.

This same technique could be used in LWT, HUT and MUT by slightly varying the location of the transducer arrays. However, the toneburst excitation modality makes it more natural and easier to take different frequency measurements at each pitch-catch location of the scanners. By varying the toneburst frequency slightly, we can utilize what we call tomographic frequency compounding to improve the contrast-to-noise ratio in the tomographic reconstructions.

The actual compounding of the images can be done in different ways. For example, if there are three sources (A_1, A_2, A_3), the following equations are three different ways to compound their data:

Average Compounding:

$$A_{comp,mean} = (A_1 + A_2 + A_3)/3 , \quad (5-1)$$

Root-Mean-Square Compounding:

$$A_{comp,rms} = \sqrt{(A_1^2 + A_2^2 + A_3^2)/3} , \quad (5-2)$$

Geometric Mean Compounding:

$$A_{comp,\Pi} = \sqrt[3]{A_1 A_2 A_3} . \quad (5-3)$$

Tests were performed on a steel pipe with an ID of 130 mm and an OD of 175 mm. Therefore, the thickness of the sample was 20 mm (~1"). A flaw was introduced on the inside diameter approximately centered in the axial direction. The flaw is an irregularly gouged, 2" x 2" square. A 10 μ s long tone burst was used to drive the transmitting transducer at frequencies between 900 kHz and 990 kHz. In total, ten tomographic scans were taken for different frequencies in this frequency range. At each

pitch-catch position in the scan, waveforms were recorded at 10 kHz steps. The distance between the transmitter and receiver at the same circumferential angle was 320 mm. After the data was recorded, the arrival time for the first arriving mode was extracted for each waveform.

Figure 5-1 shows filled contour plots of the reconstructed flawed pipe sample for the ten frequency scans taken between 900 kHz and 990 kHz. The flaw can clearly be seen in the middle of each of the reconstructions. The lighter horizontal bands on the top and bottom of the flaw are artifacts. However, they are often found to be indicators of the size and location of the flaw. These artifacts likely arise from scattering and mode conversion of the incident Lamb waves due to the flaw. Because these were only single crosshole measurements, these types of effects are more noticeable. It is also apparent that a lot of noise is present in the reconstructed images. This occurs because of the added complexity of trying to locate a flaw on the inner diameter of a thick pipe, while the measurements were taken from the outer surface of the sample.

Frequency compounding allows us to improve the fidelity of these measurements. Figure 5-2 shows three reconstructions of the flawed sample after compounding the data from the ten different frequency scans with the three methods described above. Comparing these results with those in Figure 5-1, it can be seen that the tomographic frequency compounding enhances the contrast between the flawed and unflawed regions and reduces some of the noise in the backgrounds of the images. This demonstrates clearly that the flaw is visually more pronounced after compounding.

Furthermore, the effects of compounding on the tomographic reconstruction's quality can be quantified with the following image quality parameters – contrast-to-noise

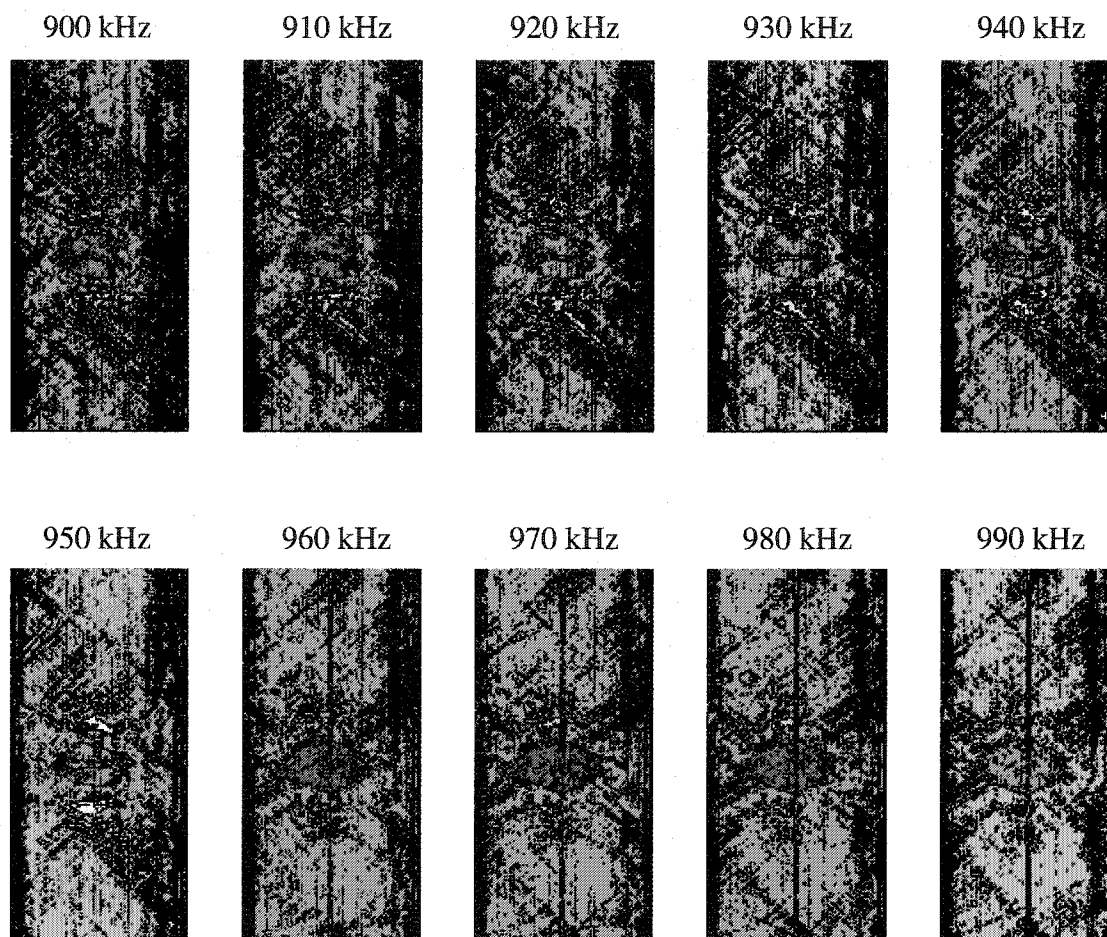


Figure 5-1. Reconstructed images for ten frequency scans of a thick steel pipe with an irregular 2'' x 2'' gouge on its ID. The flaw was approximately centered in the axial direction. Pipe thickness = 20 mm; OD = 175 mm. Horizontal axis is along the axis of the pipe ($d = 320$ mm). Vertical axis is the circumference ($d = 550$ mm).

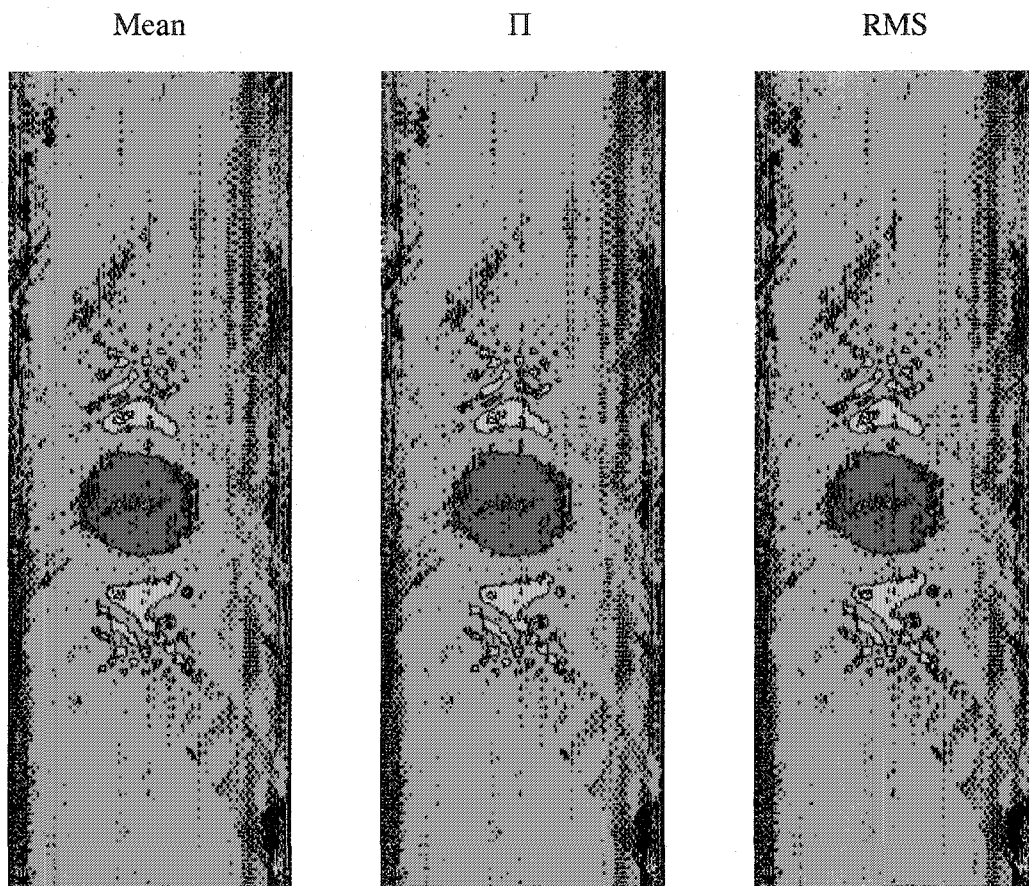


Figure 5-2. Compounded images using the 3 methods described above (Mean, Π , and RMS) for the ten frequency images in Figure 5-1.

ratio (CNR), flaw signal-to-noise ratio (FSNR), and speckle signal-to-noise ratio (SNR).

These image quality parameters are similar to those defined in [115]:

$$\begin{aligned} CNR &= (\mu_s - \mu_n) / \sigma_n \\ FSNR &= (\mu_s - \mu_n) / \sqrt{\sigma_s^2 + \sigma_n^2} \quad , \\ SNR &= \mu_n / \sigma_n \end{aligned} \quad (5-4)$$

where μ_s is the mean of the flaw data, μ_n is the mean of the noise outside of the flaw area, σ_n is the standard deviation of the noise outside of the flaw, and σ_s is the standard deviation of the flaw data. Table 5-1 and Table 5-2 show the results of these statistics for Figure 5-1 and Figure 5-2 respectively. It can clearly be seen that the tomographic frequency compounding technique significantly improves both quantitatively and qualitatively the reconstructed images.

Table 5-1. Image Quality Parameters for Figure 5-1

	Individual Frequency Images (kHz)									
	900	910	920	930	940	950	960	970	980	990
CNR	-1.31	-1.32	-1.33	-1.37	-1.33	-1.25	-1.12	-1.15	-1.17	-1.18
FSNR	-1.06	-1.06	-1.05	-1.06	-1.01	-0.94	-0.87	-0.88	-0.90	-0.91
SNR	5.95	6.11	6.44	6.86	7.12	7.34	6.04	6.28	6.50	6.62

Table 5-2. Image Quality Parameters for Figure 5-2

	Compounded Images			Average % Improvement		
	Mean	Pi	RMS	Mean	Pi	RMS
CNR	-1.52	-1.49	-1.54	17.56	16.14	18.49
FSNR	-1.20	-1.19	-1.22	19.11	17.94	19.89
SNR	7.87	7.74	7.96	17.04	15.62	18.00

5.2 Multi-Mode Arrival Time Extraction Algorithms

Even with the improvements of frequency compounding, the quality of the reconstructions that are generated by the guided wave tomographic systems are highly dependent on the accuracy of the arrival time measurements. Hardware modifications have been made to generate the cleanest signal possible (i.e. – a highly filtered signal with a narrow bandwidth), but the software time extraction algorithms still have to be robust enough to handle the noisy data that is inevitable in any real testing scenario. The method currently used to extract the arrival times of the fastest Lamb wave modes is based on enveloping the signal and is described below. It is a simple method, but it is computationally inexpensive and has been shown to perform better than other more complicated and time consuming time-frequency methods [6].

When the sample is being scanned, each projection is digitized and stored for subsequent data analysis. The waveforms are typically sampled at 50 or 25 MHz and are approximately 60 μs in duration. They are usually gated in a manner that allows us to store as little information as possible in order to save memory. For example, if a 20 cm x 20 cm area was scanned on a 3mm-thick aluminum plate at 1.2MHz, the waveforms could be recorded from the 2000th sampled point ($t = 40 \mu\text{s}$) to the 5000th point ($t = 100 \mu\text{s}$).

The time extraction algorithm begins by finding the global maximum within the recorded waveform. If at any point the waveform is truncated by the digitizer, then the first occurrence of the truncation is marked as the global maximum. The algorithm then iterates backwards and locates the local maximums for each wavelength. When it reaches the beginning of the recorded signal an array has been created that stores the

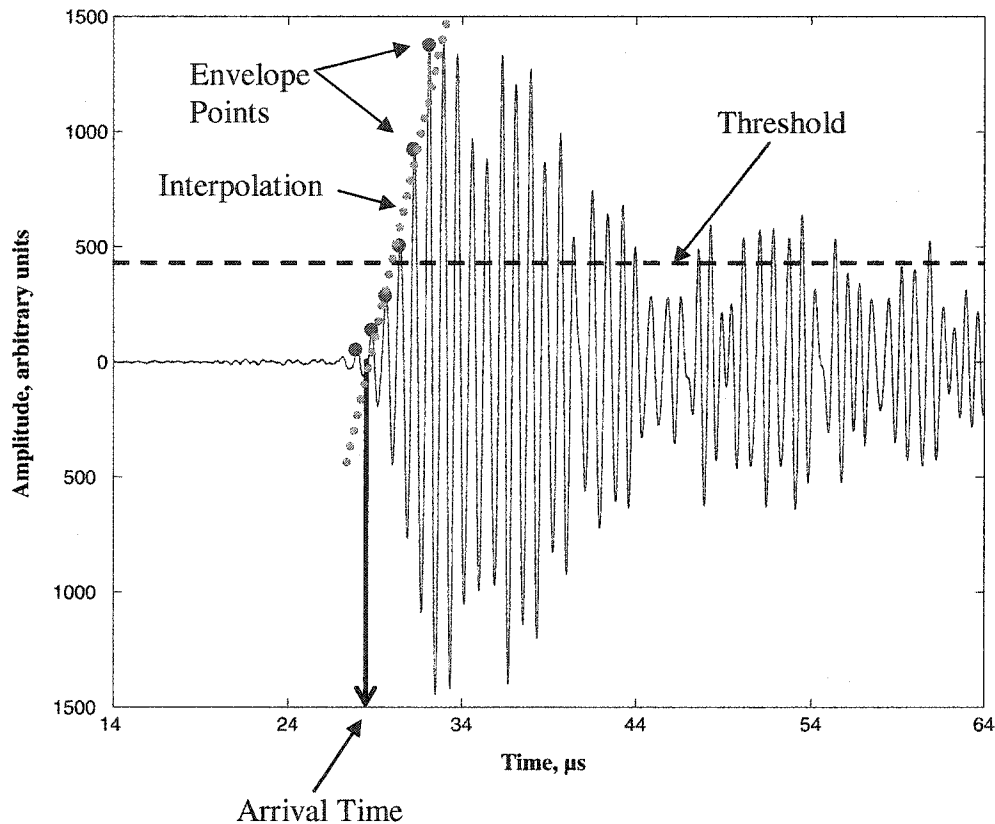


Figure 5-3. An example of the thresholding arrival time extraction algorithm. The Lamb wave signal is enveloped by selecting the peak values for each wavelength (red dots). Once enveloped, the first point that crosses the predetermined threshold value (dashed line) is marked. A linear interpolation (blue dotted line) is then computed from the adjacent envelope values to find the beginning of the envelope. This point along the time axis is marked as the arrival time for the first mode.

outline of the envelope for the signal (see Figure 5-3).

The algorithm then locates the first position where the envelope reaches a particular amplitude threshold. This value is set above the noise of the signal to locate the fastest arriving mode. It can also be set at a higher level to locate a later arriving mode as long as the amplitudes of the faster modes remain below the threshold. This latter feature has been advantageous for the initial HUT tests because it has allowed the lower energy pipe modes that arrive first to be ignored. Once the location where the envelope crosses the threshold is found, using the envelope values immediately preceding and following it, the system interpolates backwards on a straight line to locate where in time the envelope begins. The interpolation is necessary, because this front edge is often masked by the noise present in the signal.

Figure 5-4 and Figure 5-5 show arrival time scatter plots for LWT and HUT respectively. These types of plots have been found to be a very useful intermediate representation of the tomographic data. For example, Figure 5-4 shows the arrival times for a single parallel projection of a flawless aluminum plate in the LWT system. The x-axis is called the pitch-catch, or waveform, number. In the LWT scanner, the transmit transducer remains stationary while the receive transducer steps through the 100 scan positions. Once the receive transducer has stepped through all the positions, the transmit transducer steps a single position. The receive transducer then steps through its 100 positions in reverse until it reaches its original starting position. Again, the transmit position steps once and the receive transducer then steps through all of its positions. This process is repeated until the transmit transducer has stepped through all 100 positions. Therefore, the pitch-catch number – or waveform number – is: $(\text{transmit_position}-1)*100$

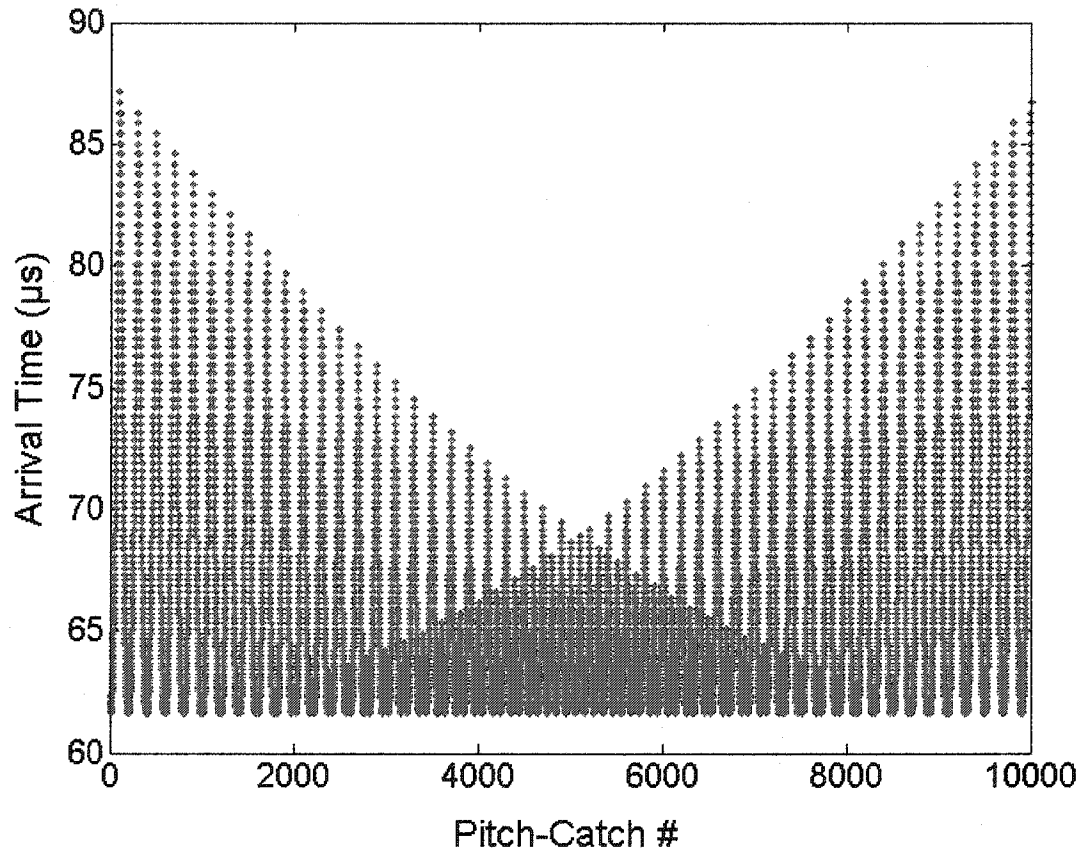


Figure 5-4. Arrival times for the first arriving mode in a clean aluminum sample with a thickness of 3.17mm. For each transmit position, there are 100 receiver positions. The pitch-catch position number is: $(\text{transmitter_position}-1)*100 + \text{receiver_position}$. The symmetrical pattern of the arrival times provide an intermediate representation to assess the validity of the recorded data.

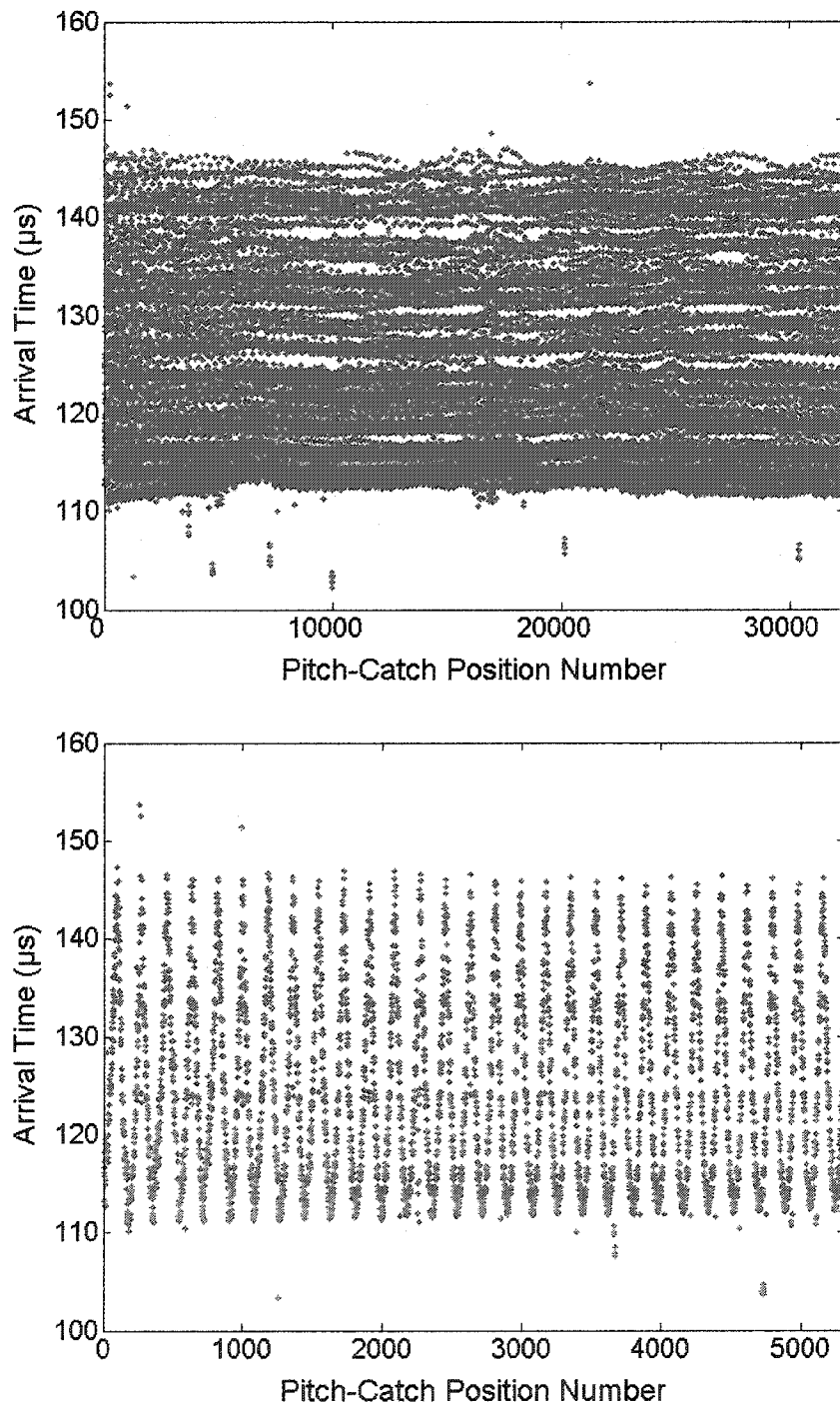


Figure 5-5. HUT arrival times from a steel simulated gun barrel sample with an OD of 175 mm and a thickness of 20 mm. The top graph is for a full scan with 180 pitch and 180 catch positions. The bottom graph is just a zoomed in portion of the complete scan. You can see the oscillatory nature of the arrival times. The pattern differs from the LWT arrival time scatter plot because the “plate” is wrapped around itself in the case of the HUT geometry.

+ receive_position. This also explains why the arrival times as a whole seem to decrease and then increase again in a symmetrical pattern. This occurs as the transmit transducer moves through all of its scanning positions. For a clean plate, this pattern should be very symmetrical as shown. For HUT, the pattern is slightly different. The transducers step through 180 positions (every 2 degrees) and since the first arriving mode only travels 180° around the pipe, the pattern is seen more as an up and down oscillation of the arrival times. As with the velocity scatter plots shown earlier, flaws appear in these representations as deviations from the symmetrical flawless patterns. This intermediate representation allows us to make sure that the data was recorded properly and it also can serve as a measure of the performance of different time extraction algorithms since errors in the arrival times show up as extreme deviations from the expected pattern.

Even though the simple thresholding arrival time extraction algorithm is effective, certain limitations still exist. For instance, if the amplitude of the mode of interest drops below the threshold level – as it often does when interacting with a flaw – then the extraction algorithm will pick up a later mode. In the case of HUT, the first arriving pipe modes, which are not of interest in our tomographic measurements, may have a larger amplitude than the true mode of interest. This also causes the current time extraction scheme to provide incorrect arrival time values. Furthermore, it is incapable of providing arrival time information for multiple modes. This limits the detection capability in the pipe geometry to waves that only travel halfway around the pipe, thus further degrading the image quality because of poorer wave vector coverage.

Many other methods have been explored by various authors to improve the accuracy of the arrival times and to extract the time-of-flight information for multiple

modes. Artificial neural networks (ANNs) were explored by this author [116] as a method to locate the arrival times, but the particular backprojection training algorithm used was unable to reliably pick out the first-arriving mode. It was found that the ANN was effective in removing some of the systematic errors present in the arrival time data, but that it was unable to recognize the arrival of the first mode. ANNs are extremely useful for pattern recognition problems, but because of the dispersive nature of Lamb waves (see Section 2.2.3), the mode shapes change and make it more difficult for them to be recognized automatically.

Malyarenko [6] explored various time-frequency methods to try and improve the arrival time extraction methods. However, Wigner-Ville distributions, short-time Fourier transforms (STFT), and positive distributions did not sufficiently improve the accuracy of the arrival times. Malyarenko suggested that wavelet time-scale analysis be explored in future work.

5.2.1 Wavelet Analysis

Due to the suggestion of Malyarenko and recent work done by Hou et. al. [117, 118], wavelet analysis was explored here as a better solution to the current arrival time extraction problem. Wavelets have become popular in many different types of signal and image processing techniques. They are particularly advantageous for both the denoising and compression of signals. For example, the FBI chose to use wavelets over STFT to compress and transmit fingerprint data. The fingerprints can be compressed with wavelets about 20-fold without any loss of information [119].

The wavelet transform is different from other time-frequency techniques because its basis functions – which are called wavelets – are local, finite signals. When using the Fourier transform, signals are broken into a sum of sines and cosines and we gain information about only the frequency content of the signal. To represent a local function – one that vanishes outside a short interval of space or time – a global basis requires extreme cancellations. Reasonable accuracy thus requires many terms in the Fourier series. One solution to this problem is to use a windowed Fourier transform, also known as the short-time Fourier transform. The STFT breaks the signal into time segments that are transformed individually. However, the time window length is fixed, and this can be a major drawback to the method. The STFT does not allow one to simultaneously look at events that happen on different time scales.

In contrast, the wavelet transform has the ability to vary both the time window and frequency, or scale, of its basis functions. Because the wavelet transform is just a correlation between the signal and the set of wavelets, it allows the signal to be broken down into a more natural representation. The variability of the wavelet basis functions allows the signal to be broken into large time windows for low frequency components and shorter time frames for higher frequencies. This ability is often referred to as a time-scale analysis, as we will see below, and is a more natural way to look at things. In a single decomposition one can look at both large and fine features.

The continuous wavelet transform is defined as:

$$W_g(a,b) = \frac{1}{\sqrt{a}} \int_{-\infty}^{\infty} x(t) g_{a,b}(t) dt$$

$$\text{where, } g_{a,b}(t) = g\left(\frac{t-b}{a}\right).$$

$W_g(a,b)$ is defined as the wavelet coefficient for the translation b and the scale a . $g_{a,b}$ is the scaling function which is calculated from the mother wavelet $g(t)$. By varying the translation and scale parameters, one can inspect the signal from different time and scale representations. This flexibility is a great advantage over the traditional Fourier analysis and STFT.

In our case, the signals being processed are discrete signals. The discrete wavelet transform (DWT) implements the CWT for discrete signals. Conceptually, the DWT can be viewed as a series of filters as in Figure 5-6. The input signal is broken down into a low frequency approximation and a high frequency detail signal. The approximation signal can then be broken down further through the same process. The result will be an approximation signal with a number of detail signals. Each of the detail signals are a set of wavelet transform coefficients at the next finer scale. Denoising and compression are accomplished by setting a threshold value for the resulting coefficients. For most signals, the majority of information is contained within only a few coefficients. Furthermore, because the details are the high frequency components, by removing these coefficients, one can also reduce the noise in the signal – which is often higher in frequency, smaller in scale. This allows one to eliminate the noise, while keeping the information that is desired.

5.2.2 Dynamic Wavelet Fingerprinting

Hou [117, 118] has shown that wavelet analysis can be used in the classification of multi-mode signals. The wavelet fingerprinting technique Hou developed transforms the one-dimensional signal identification problem into a two-dimensional image

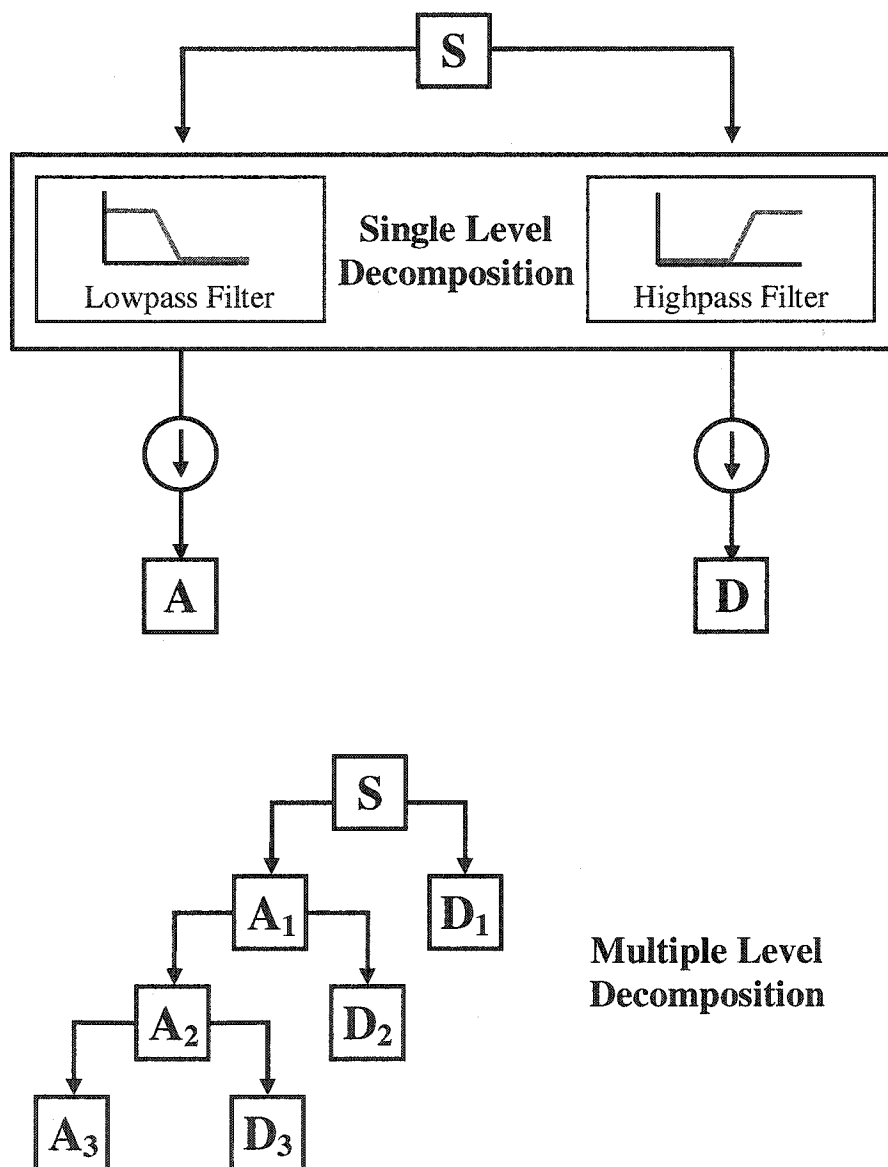


Figure 5-6. The discrete wavelet transform can be seen as the signal being split into its low frequency components – approximations – and its high frequency components – details. Subsequent decompositions can be done on the resulting approximations, and this further separates out the details of the signal from its approximation.

recognition problem. The individual Lamb wave signals are converted into unique fingerprint images through the wavelet transform. Once the wavelet coefficients have been calculated using an appropriate wavelet, a slice projection operation is used to project four equally-spaced slices of the wavelet coefficients onto the time-scale plane. These wavelet fingerprints allow image matching techniques to be used to locate the arrivals of the different modes. By identifying the feature of interest in these images that relates to the arrival of a particular Lamb wave mode, classification algorithms can be written to extract the time-of-flight data needed by the reconstruction algorithms. Dynamic wavelet fingerprints (DWFPs) have shown success in identifying and separating out multi-mode Lamb wave signals.

5.2.3 Sorting Algorithm

The work on DWFPs done by Hou was conducted in parallel to this research and is reported elsewhere [117]. Therefore, in order to generate representative data, a less sophisticated scheme was used to extract the arrival times for the multiple modes. The method presented below uses a version of the DWT to generate smooth envelopes for the Lamb wave signals. The peaks of these different envelopes are then used as the arrival times of the multiple Lamb wave modes. One of the problems with the extraction of multiple arrivals is that destructive interference between two modes may cause one of the modes to seemingly disappear in the signal. Because the DWFP algorithm assumes that the modes remain in the same arrival order throughout the scan, this causes an arrival time that belongs in the second or third mode to actually appear in the first or second mode series respectively. The goal of the sorting algorithm presented in this section is to

try and counteract this problem by using the frequency walk information to sort the arrival times into their correct mode series.

Ultimately, arrival time information is needed for the multiple guided wave modes in order to improve the accuracy of the HUT technique. However, because of the more complicated signals and geometry, preliminary work to solve this problem has been done on plates. The plate samples do not have the added complexities of the constructively and destructively interfering counter-helical modes and the existence of non-helical pipe modes.

For the multi-mode analysis of the Lamb wave signals, it is useful to use wavelets to both denoise and envelope the signal. As discussed above, the current time extraction algorithm relies on the accurate enveloping of the signal. In the past this had been done using the peak values of each wavelength. However, when trying to track more than the just the first arriving mode, the unsmooth nature of this type of envelope made it difficult to detect subsequent modes. Therefore, wavelets were used to simultaneously de-noise and envelope the signal with a pruning procedure [120] based on the discrete stationary wavelet transform [121].

MatlabTM was used to compute the wavelet transform of the Lamb wave signals using an 8 level `coiflet3` decomposition. Figure 5-7 graphically shows what the `coiflet3` wavelet looks like. The absolute value of each signal was decomposed using the stationary wavelet transform. The first five levels of coefficients were zeroed and the remaining levels were inverse transformed to provide the envelope values. The ability of the wavelet transform to separate out the high frequency noise and oscillations of the signal into the first 5 levels of detail is what makes this technique possible. Figure 5-8

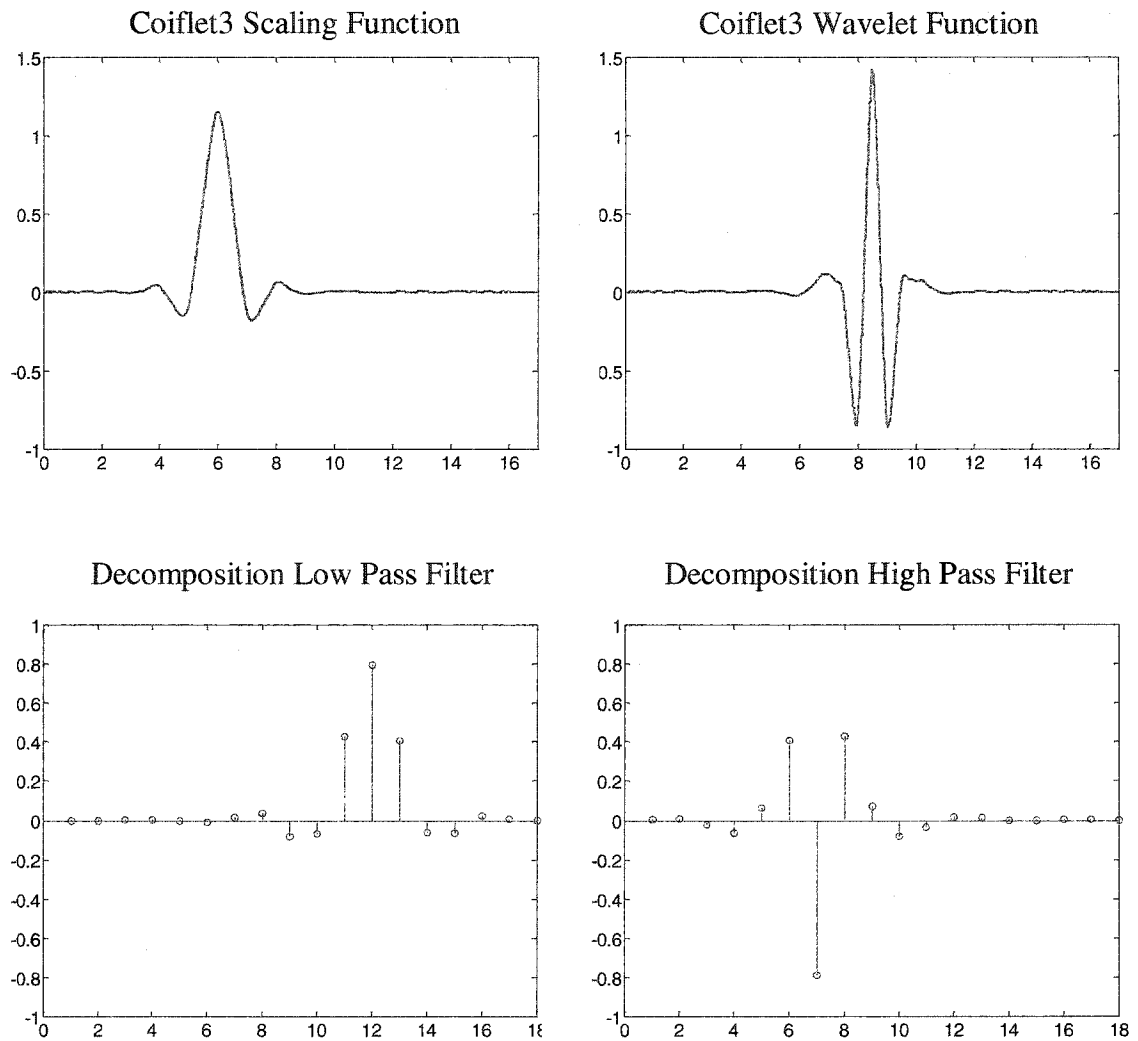


Figure 5-7. Illustration of coiflet3 wavelet and scaling functions. The low and high pass filters for the coiflet3 wavelet are also shown.

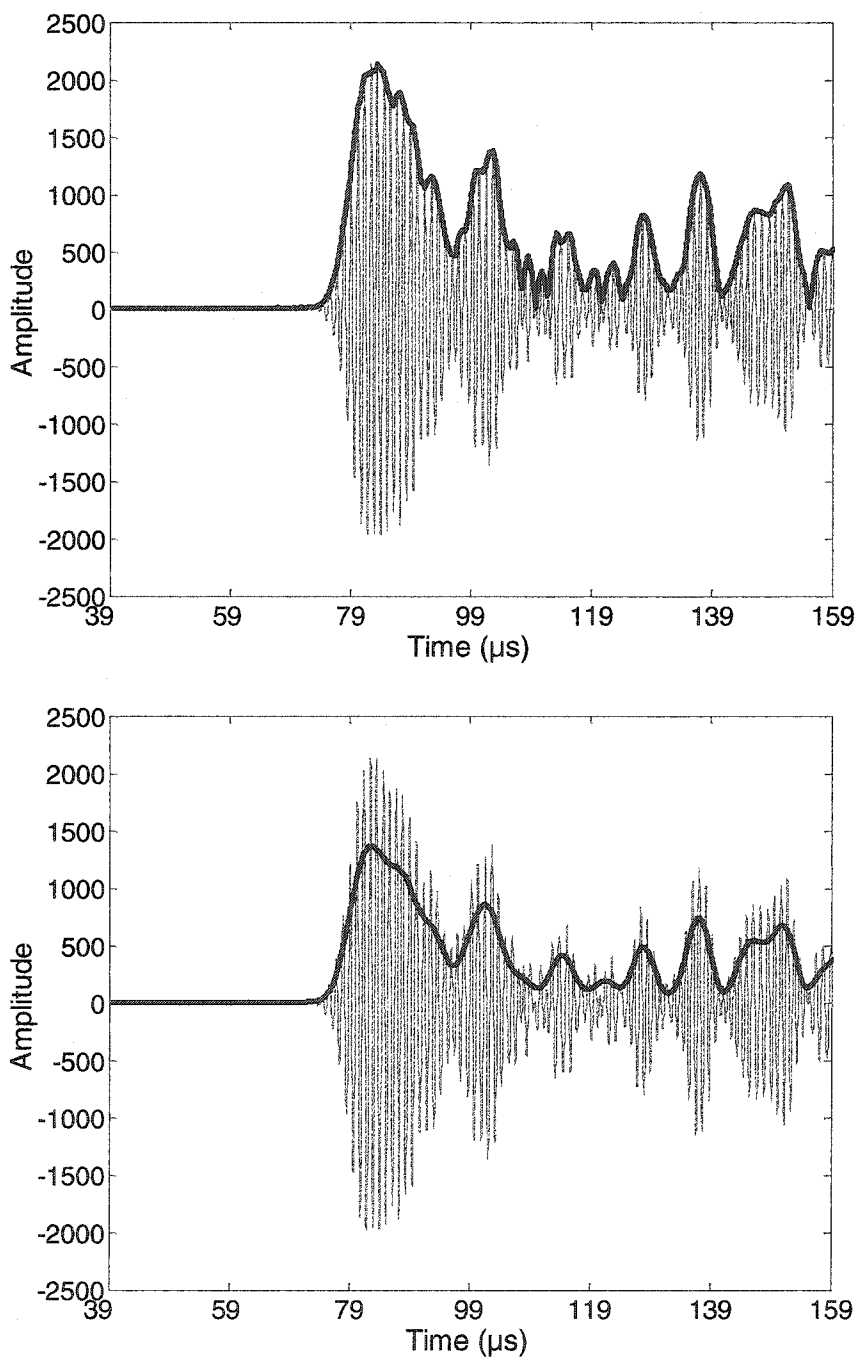


Figure 5-8. Typical Lamb wave signal demonstrating the difference between the wavelength peak enveloping (top graph) and wavelet enveloping (bottom graph) techniques. Notice that the wavelet envelopes are smoother and do not enhance some of the subtle peaks that are due to interference or signal variation.

shows a typical signal and compares the peak enveloping and wavelet enveloping techniques. It can be seen that the wavelet enveloping technique provides a smoother envelope and identifies fewer “false peaks”.

The wavelet enveloping allows us to extract the arrival times of the envelope peaks after the first arriving mode. Figure 5-9 shows the arrival times for the first three peaks for the first transmitter position in a scan of a clean aluminum plate at 920 kHz. It can be seen that the first peak’s arrival time is extracted cleanly, but that the subsequent peaks are often scattered. The uncertainty in the arrival times of the later arriving peaks arises from constructive and destructive interference between dispersive modes. As was shown earlier, the individual modes spread as they propagate and they do not retain their initial pulse shape. This is intentional because dispersion is being used to provide sensitivity to thickness changes.

In Figure 5-9 it can also be seen that at receiver position #61 the second arrival jumps to the curve where the third arrival had been appearing. Because this is a clean sample, this is not due to a flaw, but rather to the disappearance of the second peak in the enveloped signal due to interference between modes. Figure 5-10 shows the signal envelopes and waveforms from receiver positions 55 to 62. This sequence highlights how the envelope peaks combine. The disappearance of the peak is less of an issue than the intermixing of the mode series. In order to generate a tomographic image from the later arriving mode arrival times, they need to be separated from each other. However, this problem becomes nontrivial because the data series jump between modes as certain modes interfere with each other.

To gain more information, at each pitch-catch position the frequency was

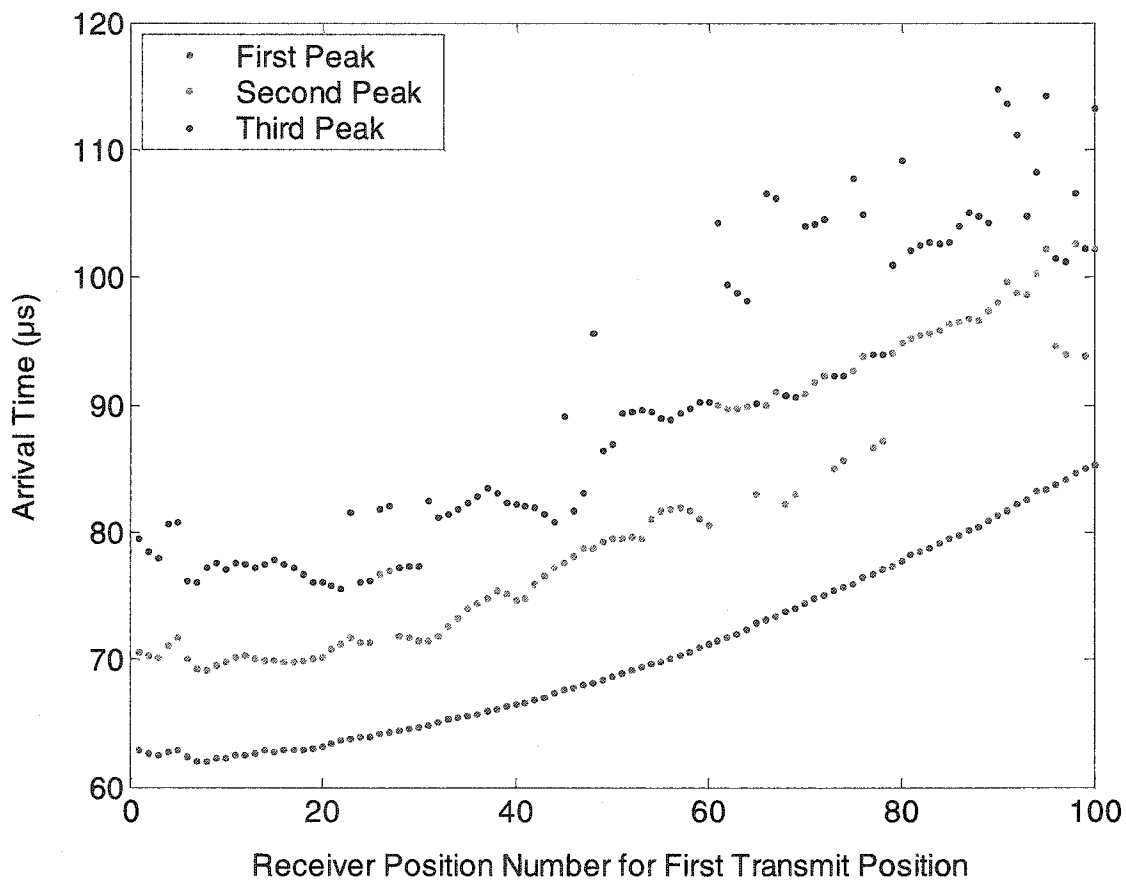


Figure 5-9. Arrival times for the first three peaks of the Lamb wave signals recorded in a tomographic scan of a clean aluminum plate. Only the arrivals for the first transmit position are shown. Notice that the first arrival is very clean, while subsequent arrivals are more scattered.

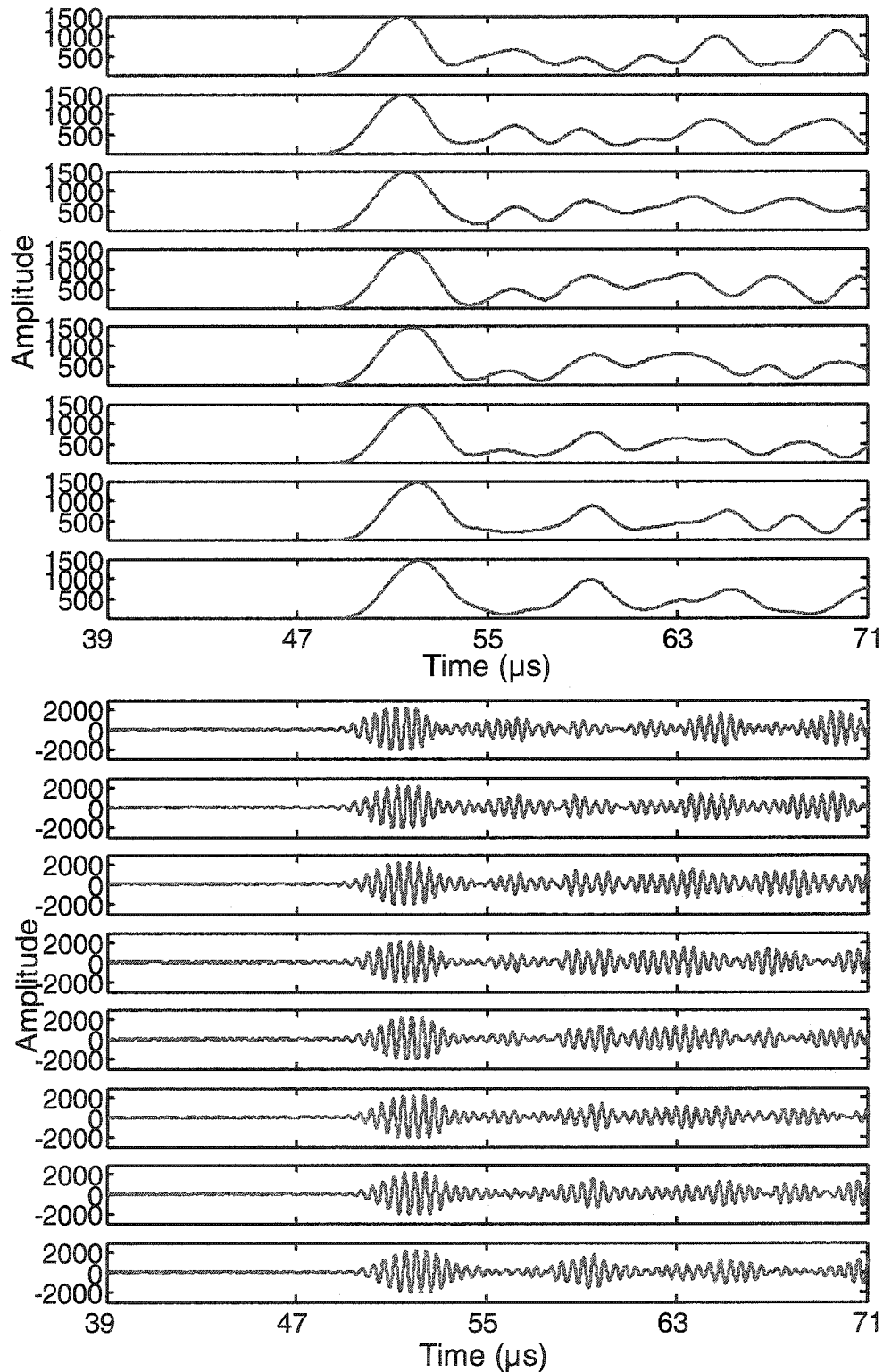


Figure 5-10. The envelopes and waveforms of the recorded Lamb wave signals in the clean aluminum plate sample for receiver positions 55-62 (top to bottom). As the second and third wave packets interfere, the second peak disappears while the third peak increases in amplitude

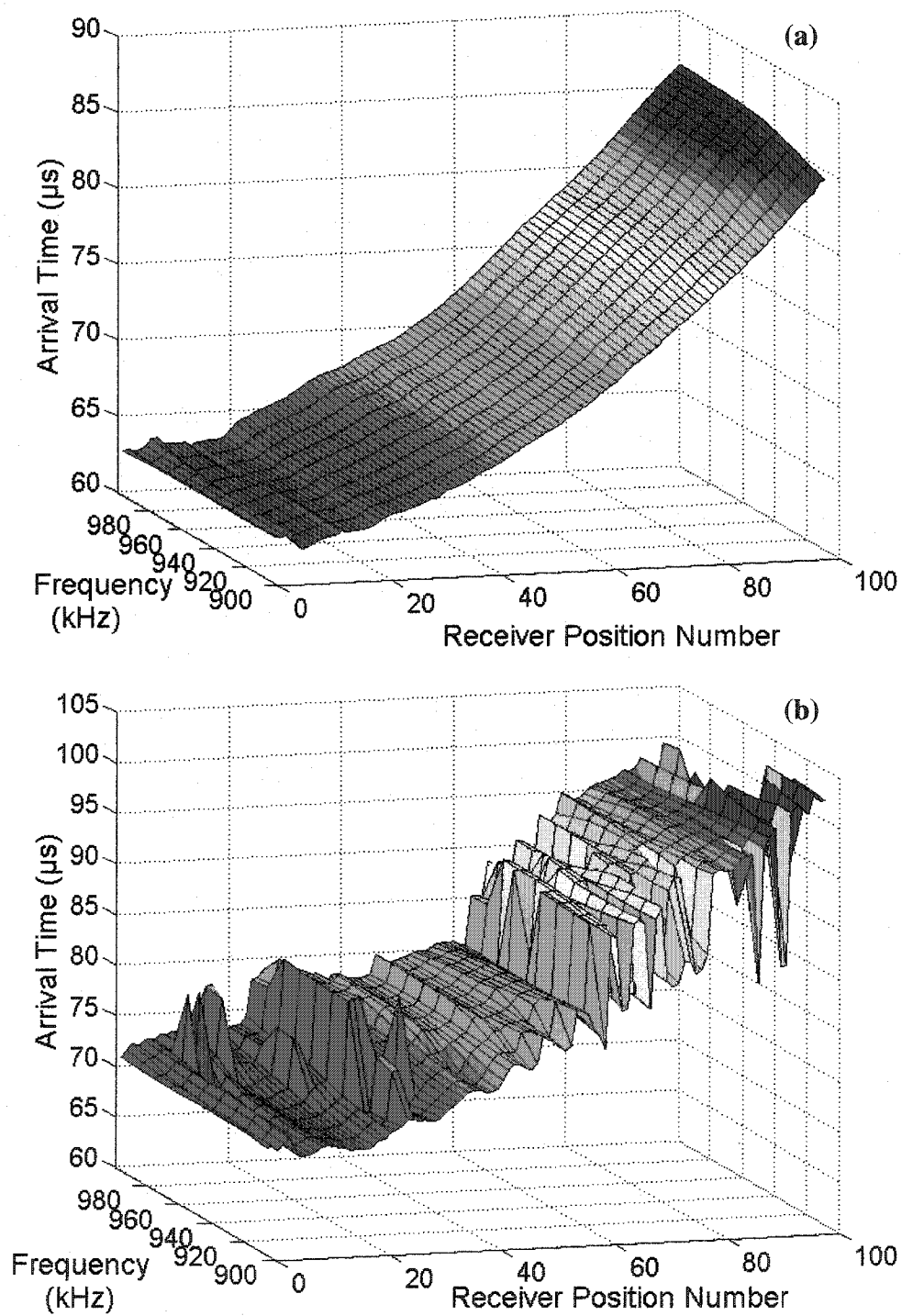


Figure 5-11. Frequency walk surface plots for the arrival times of the first transmit position in a scan of a clean aluminum plate. Times recorded for ten evenly spaced frequencies between 900 and 990 kHz. (a) Arrival times for the first peak. (b) Arrival times for the second peak.

incremented at equally spaced intervals. The frequency walking technique provides an extra dimension in order to solve the arrival time sorting problem. Figure 5-11 shows the same arrival times as in Figure 5-9, but separates them by mode and has the additional frequency walk information. The first mode is a smooth distribution between the different frequencies and the arrival times can be seen to generally get later as the frequency increases. The second peak's arrival time can again be seen to jump from a time consistent with the second mode to one consistent with the third mode at about receiver position #61. It can then be seen to oscillate between the two values for a period.

For the clean plate, the arrival times can be sorted into their correct series by fitting the individual series with a 3rd-order polynomial fit. This provides a guess for where the missing points should lie and provides a measure to determine whether a mode has been missed. Figure 5-12 shows the arrival times for the first three envelope peaks and the polynomial fit for each data series. It can be seen that the polynomial fit effectively estimates where the mode should be and also shows how at receiver position #61, the second recorded peak arrival is actually the arrival time for the third mode. Figure 5-13 shows how the polynomial fit performs for a complete projection and different zoomed-in portions of the scan.

In order to use the polynomial fit to sort the arrival times into their appropriate mode numbers, we compare the times to the fitted mode curves. For the clean plate, the arrival times are sorted into the series that corresponds to the fitted curve that they are closest to. If a particular mode is not present, we store the fitted value at that waveform number into the arrival time data. Figure 5-14 shows the same data as in Figure 5-12

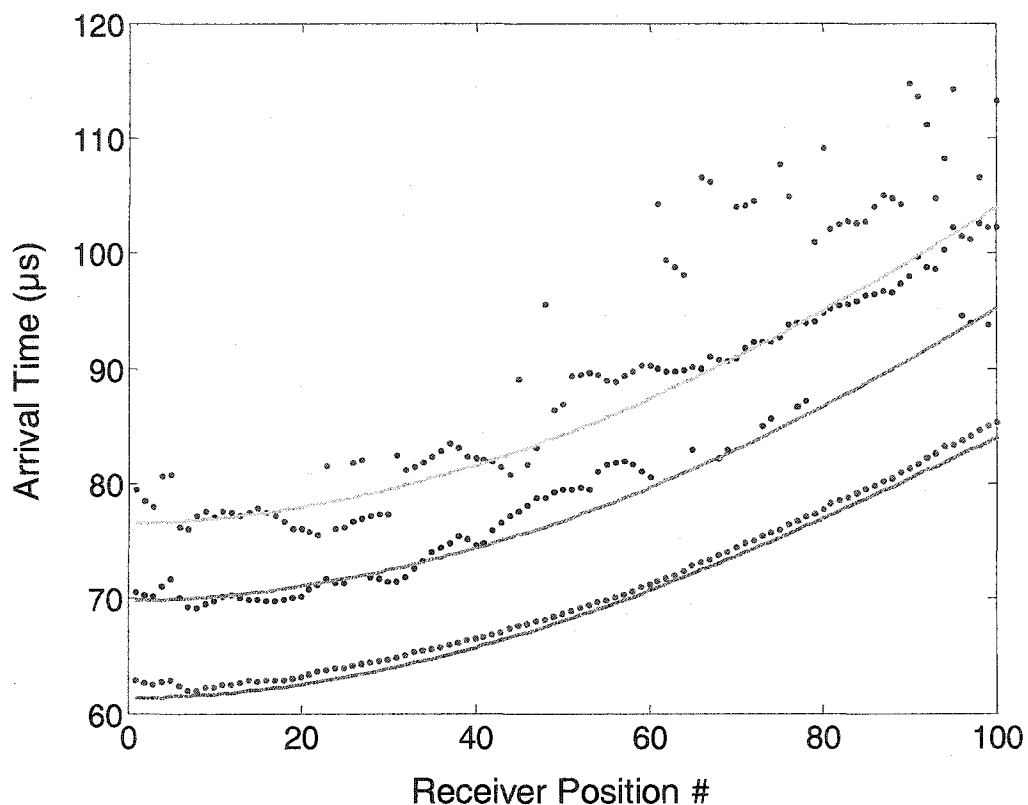


Figure 5-12. Arrival times for the first three peaks of a clean aluminum plate at the first 100 receiver positions (blue – peak 1; red – peak 2; purple – peak 3). The solid lines are the fitted mode predictions (green – mode 1; light blue – mode 2; yellow – mode 3). It can be seen that using a polynomial fit on the raw data provides an effective estimation of where the mode should arrive. It is also apparent that starting at position #61, the second peak arrives closer to the expected arrival of mode 3.

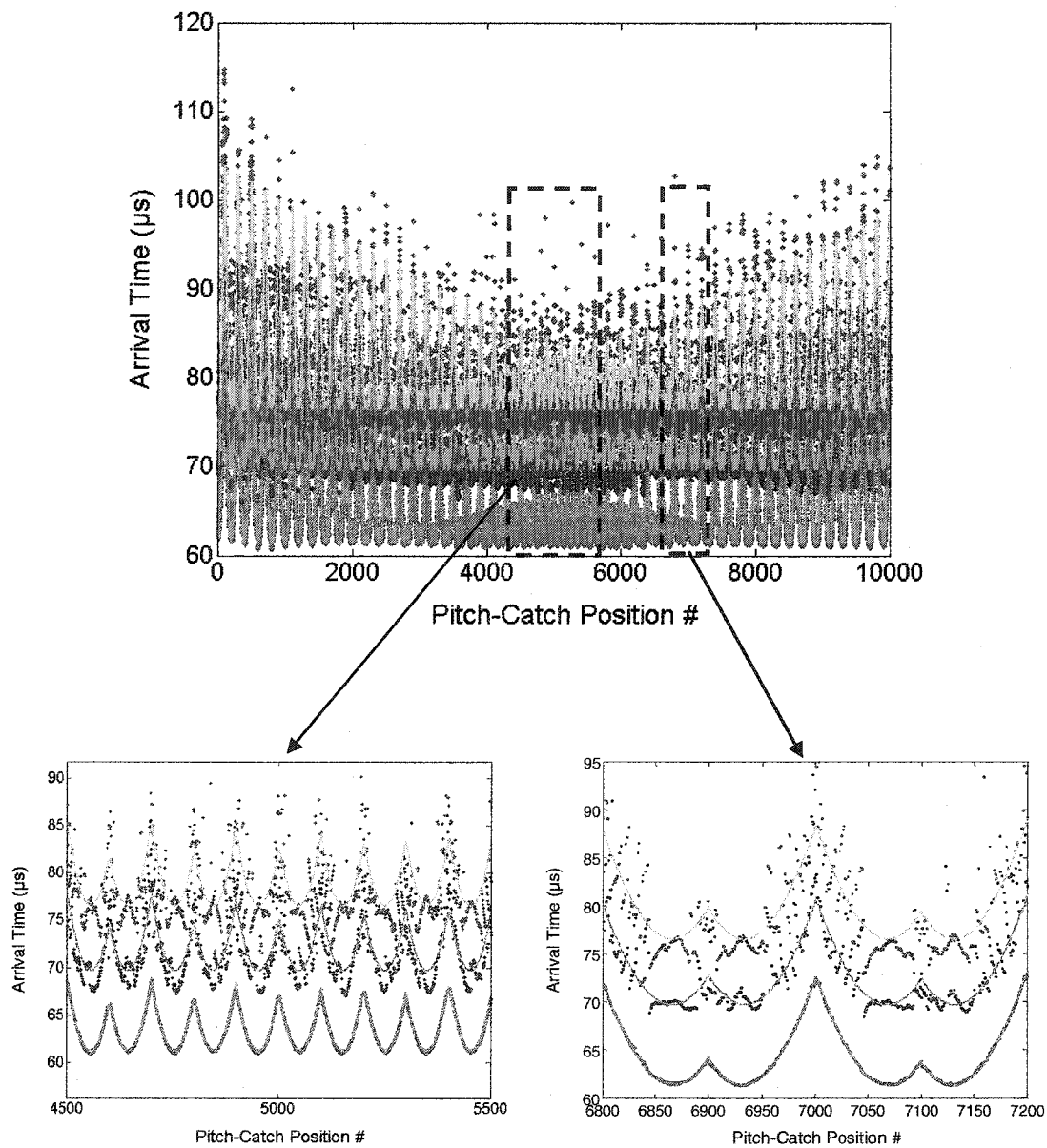


Figure 5-13. Arrival time plots for the first three peaks (dots) and predicted mode arrivals (solid lines) for a complete tomographic projection. The zoomed-in portions show how the arrival times change in a symmetrical pattern and how the fitted mode curves follow the peak arrivals for the entire scan.

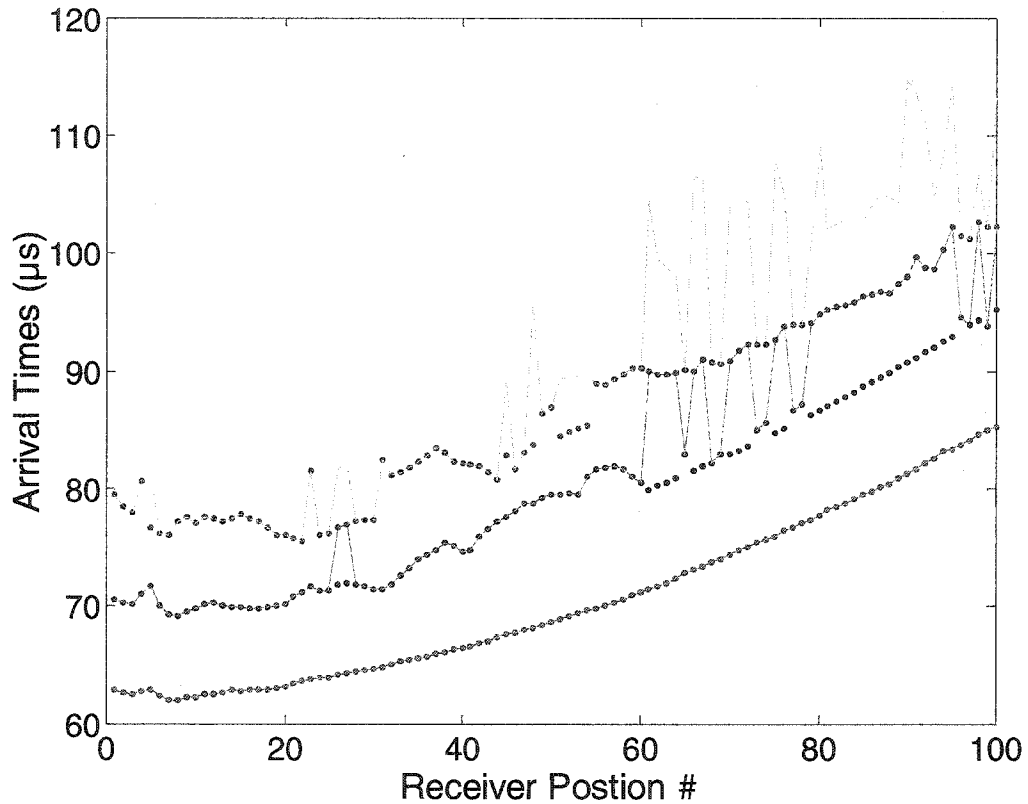


Figure 5-14. Sorted arrival times (points) for the first three modes of the clean plate data shown in Figure 5-12. The original arrival times (peak 1 – green line; peak 2 – turquoise line; peak 3 – yellow line) are shown for comparison. It can be seen that the algorithm successfully sorts the different peaks into their correct data series.

after being sorted. It can be seen that the points that jumped from mode two to mode three in the second series are now correctly sorted into the series that represent their mode number. Now that we have sorted some of the points, we can refit the mode data and sort again to further improve the arrival times for the clean sample.

The sorting algorithm compared to both the theoretical and predicted arrival times reduced the mean square error (MSE) of the measurement. The MSE for the unsorted arrival times compared to theoretical arrival times was $3.01\mu\text{s}$, $6.32\mu\text{s}$, and $8.26\mu\text{s}$ for the first three modes respectively. After sorting, the MSE was reduced to $3.01\mu\text{s}$, $5.77\mu\text{s}$, and $7.47\mu\text{s}$. Part of the large discrepancy between the theoretical and experimental arrival times of the modes is due to how the times were extracted. The arrival times are determined by the location of the envelope's peak. Dispersion of the mode shapes due to the bandwidth of the input signal can cause variations in the peak arrival. A more effective measure of arrival times has been found to be the location of the front of the envelope [6]. However, for later arriving modes, this position is hard to obtain. In contrast, the MSE between the unsorted arrival times and the predicted arrivals (the fitted data) was $0.24\mu\text{s}$, $2.24\mu\text{s}$, and $3.48\mu\text{s}$ for the first three modes and for the sorted times, the MSE was $0.24\mu\text{s}$, $1.47\mu\text{s}$, and $1.71\mu\text{s}$. Figure 5-15, Figure 5-16, and Figure 5-17 show the arrival time scatter plots before and after the sorting algorithm was applied for the first three modes. As expected, the first arrival time is fairly accurate and is not affected by the sorting algorithm. These figures show that for the first projection of the clean plate scan the second and third modes are both improved by the sorting algorithm.

The previous algorithm works well for a clean sample, but will blur actual flaws in the tomographic reconstructions because it assumes that any deviation from the

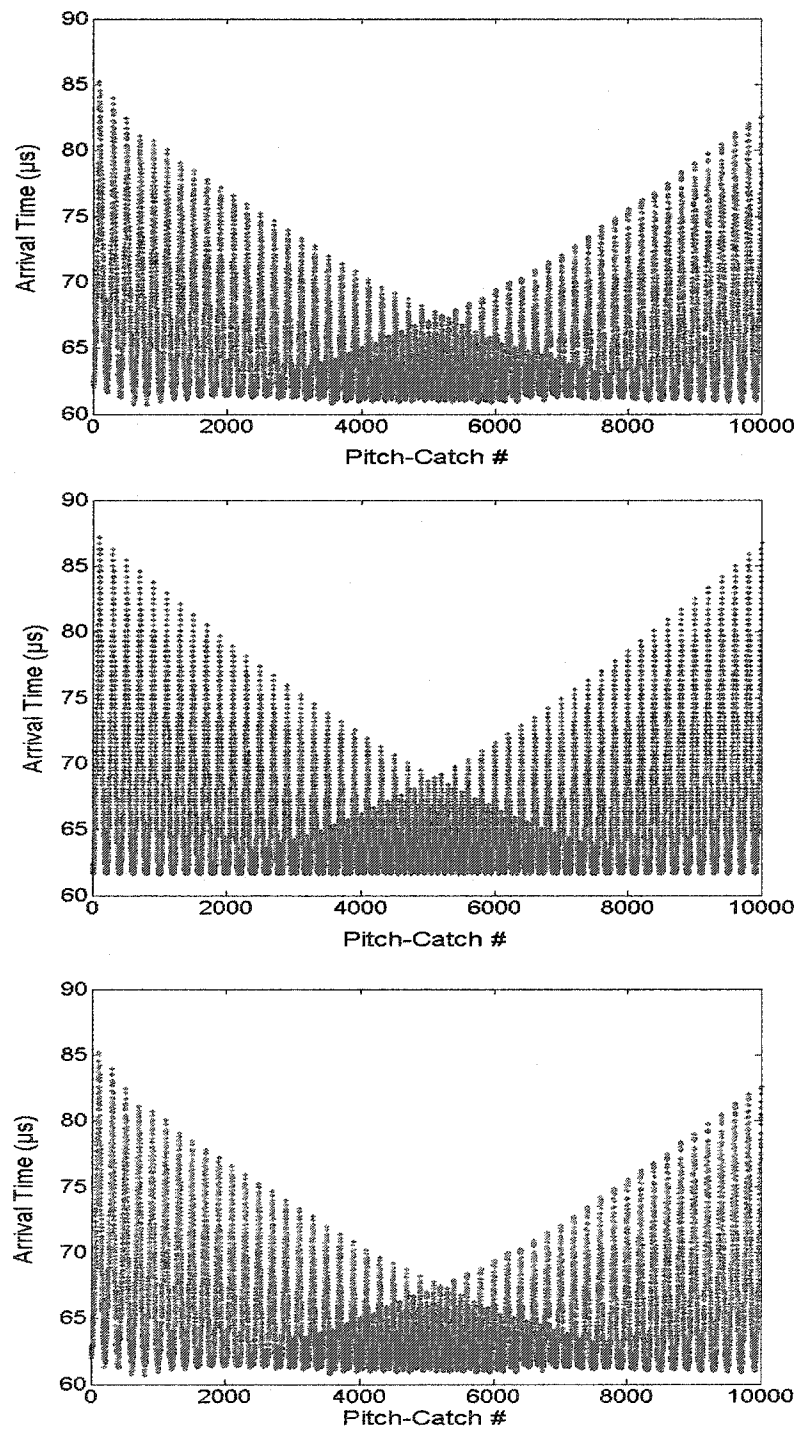


Figure 5-15. Arrival time scatter plots for the first mode in projection #1 of a clean aluminum plate. (a) Raw data of the first envelope peak arrival. (b) Theoretical data for first arriving mode $A_1 - 3.25 \text{ mm}/\mu\text{s}$. (c) Sorted data for the first arriving mode.

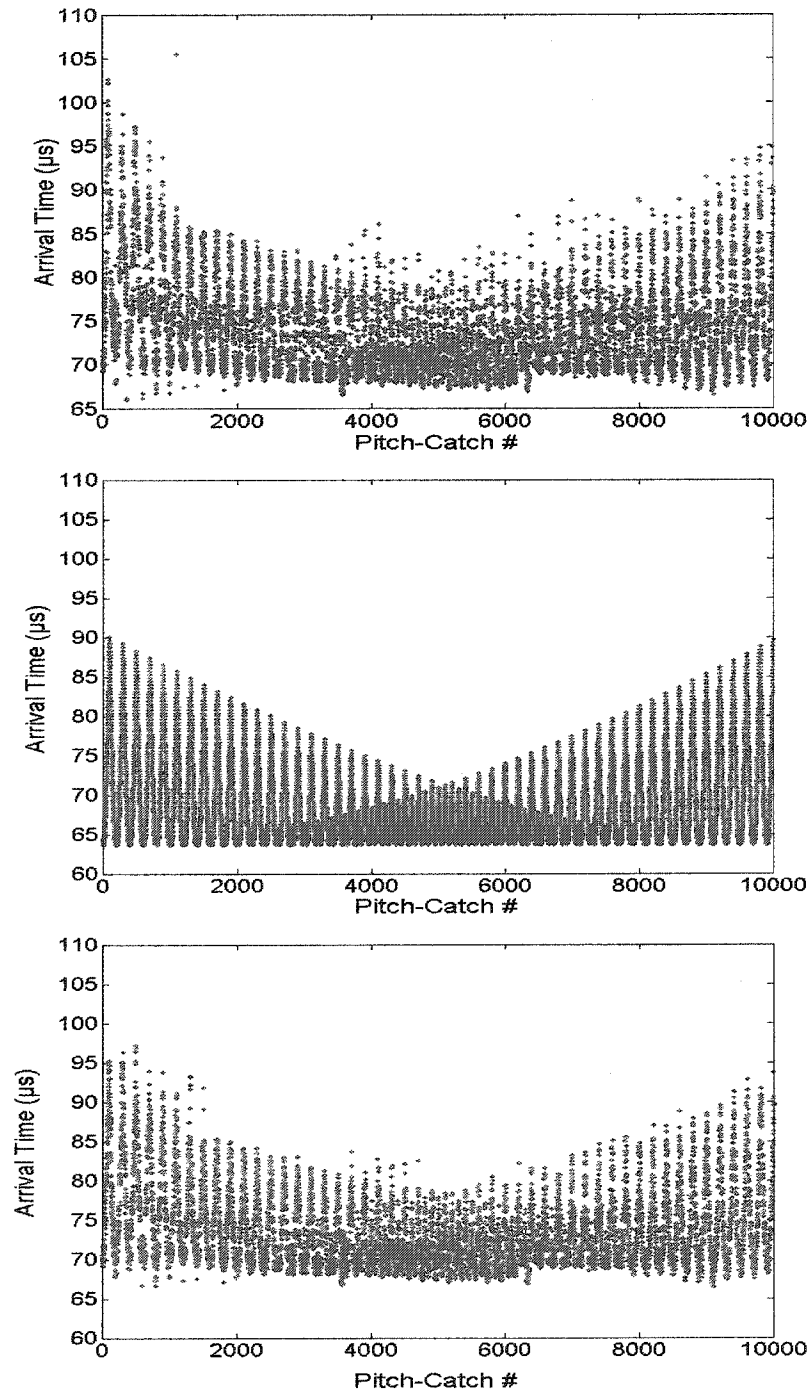


Figure 5-16. Arrival time scatter plots for the second mode in projection #1 of a clean aluminum plate. (a) Raw data of the second envelope peak arrival. (b) Theoretical data for second mode $A_0 - 3.14 \text{ mm}/\mu\text{s}$. (c) Sorted arrival times for the second mode.

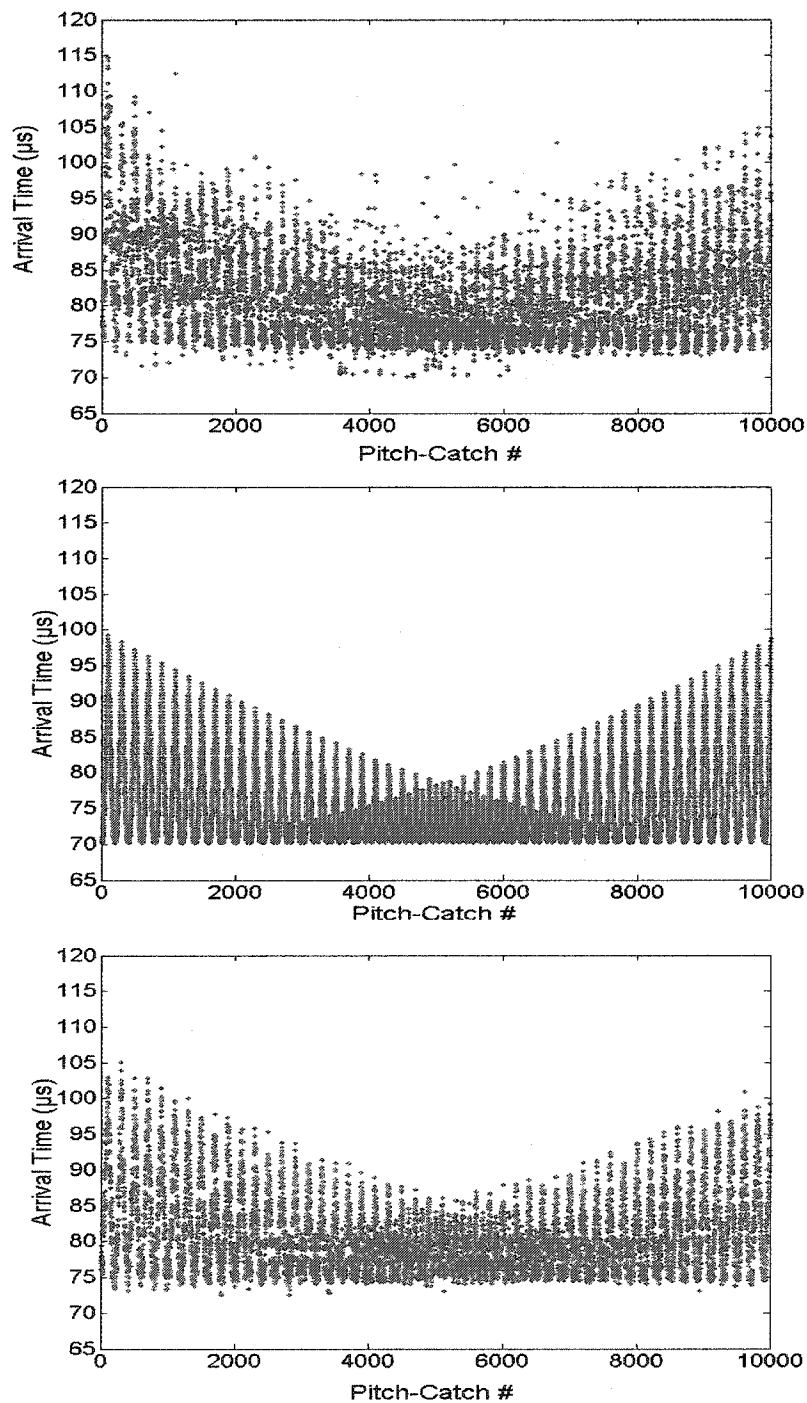


Figure 5-17. Arrival time scatter plots for the third mode in projection #1 of a clean aluminum plate. (a) Raw data of the third envelope peak arrival. (b) Theoretical data for third mode $S_0 - 2.85 \text{ mm}/\mu\text{s}$. (c) Sorted arrival times for the third mode.

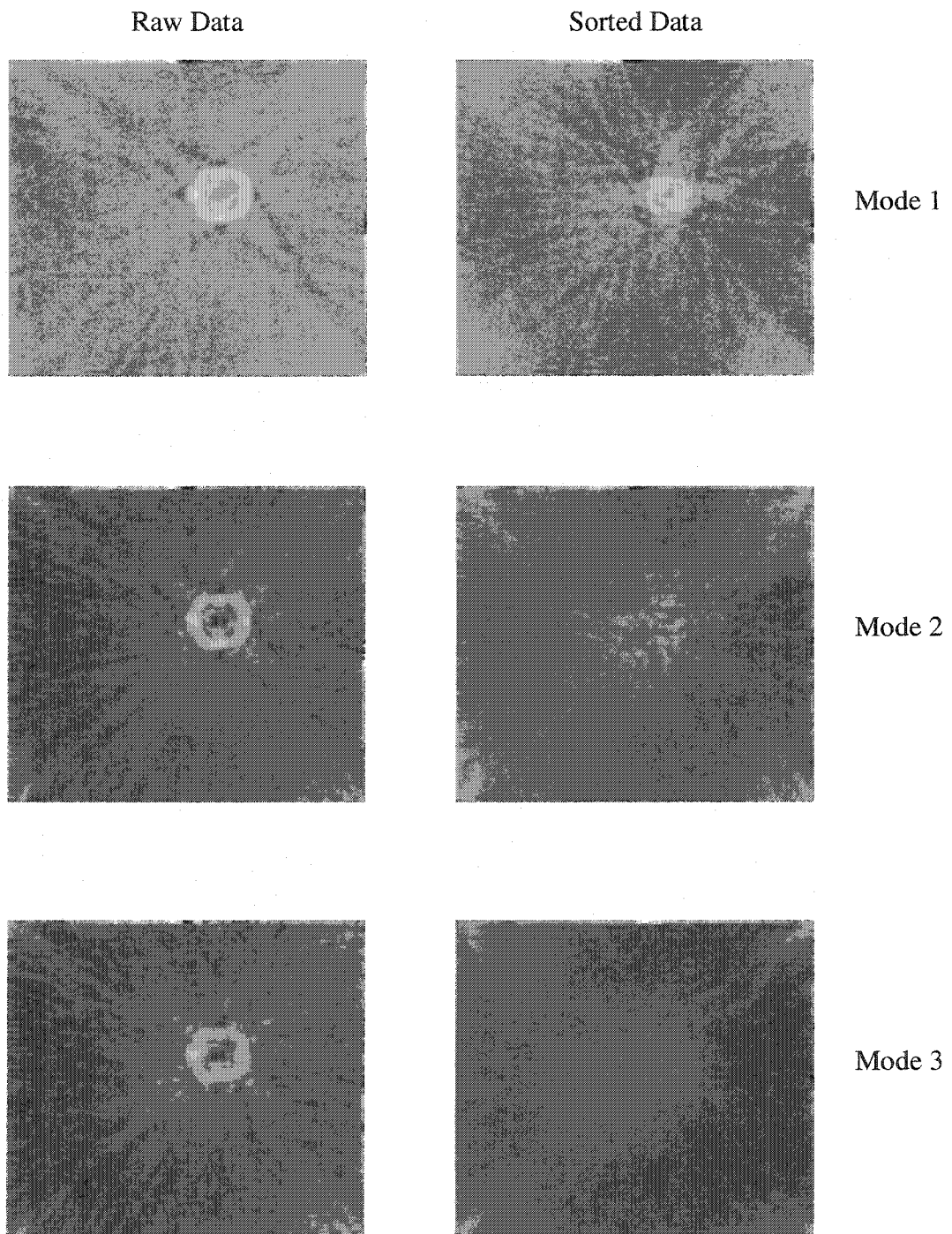


Figure 5-18. Reconstructions for a flat bottomed hole in a 3.17mm thick aluminum plate. 50% thickness loss within the flaw (Plate #1 in the blind study test). Left column is for the unsorted envelope arrival times. The right column shows the reconstructions of the sample after the mode data was sorted with only the first order comparison to the fitted mode curves.

predicted curves are due to interference. Figure 5-18 shows reconstructions for a sample with a flat bottomed hole (this was the same sample as Plate #1 from the blind study tests – see Section 3.4.3). The images show results for the first three modes both with and without applying the mode sorting algorithm. The contrast between the flawed area and the outlying regions is a lot poorer after the data has been sorted. This occurs because the flaws are seen as deviations of the arrival times from the expected arrivals and thus are sorted to a different series and replaced with the expected times. To extend the usefulness of the sorting algorithm to flawed samples, more information is needed to determine whether the individual arrival times are due to a flaw, a missed mode, or a bad data point resulting from poor coupling or digitization.

As mentioned above, the first arrival is accurately detected and is not affected by the clean plate sorting algorithm. Therefore, we can use the first arrival as a measure to see whether the subsequent arrivals can be seen as “clean” arrivals or need to be handled as “potential flaw” arrivals. Figure 5-19(a) shows the first 100 points of projection #1 for the flat bottomed hole sample. From the geometry of this sample and the scanner, the Lamb wave data should begin to interact with the flaw around receiver position #55 for the first transmit position. The arrival time plot shows how the first mode speeds up as it interacts with the flaw. It also should be noted that the second arrival also speeds up and arrives where the first mode would be located on a clean sample. This causes the sorting algorithm to choose the second arrival time as the first mode and thus discounts the actual “flaw” arrivals. In Figure 5-19(b) the deviations for the arrival times in the flawed sample are also shown. It can be seen that at position #55 – where the Lamb wave data begins to interact with the flaw – the deviation in the measurement from the fitted data is

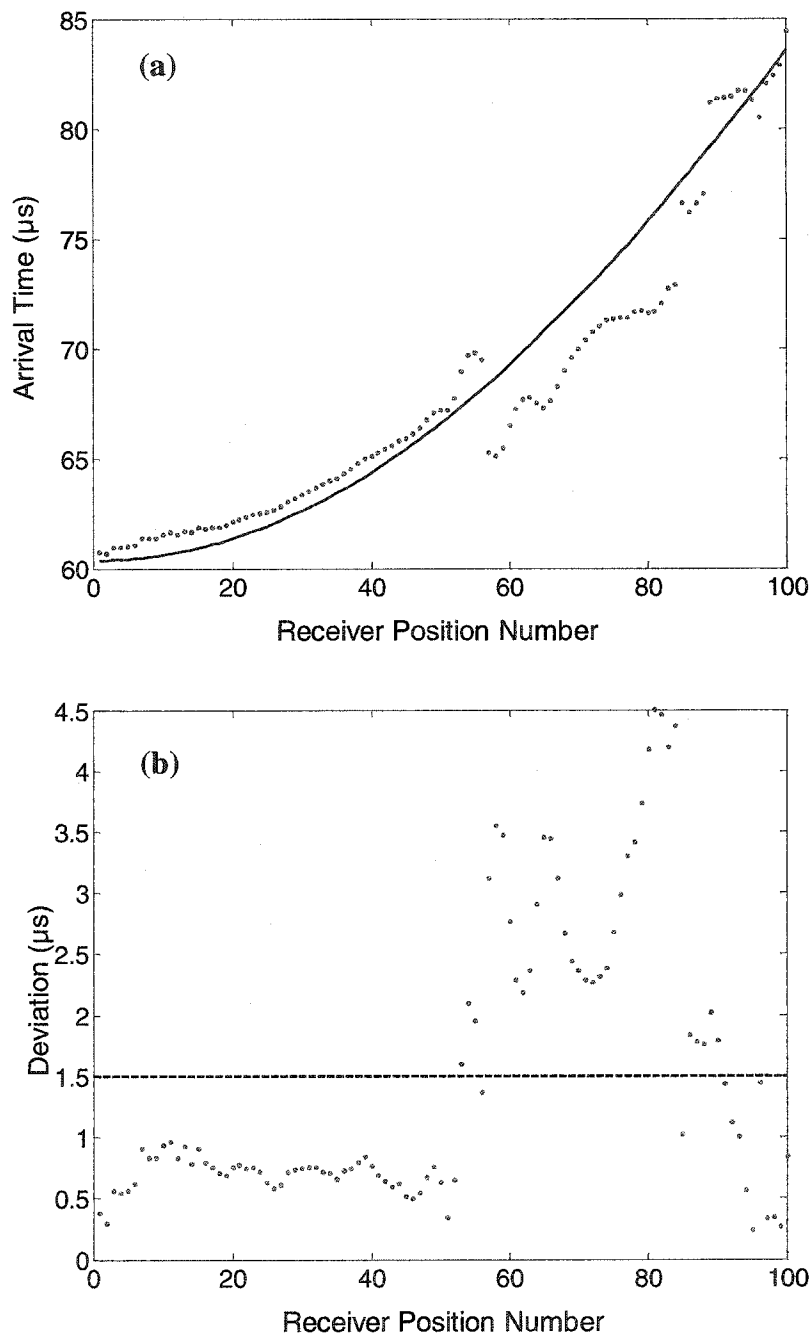


Figure 5-19. (a) The first peak's arrival times for the first 100 points of projection #1 for the flat bottomed hole sample (red line is the polynomial fit of the arrival times for the first mode). (b) Deviations of the arrival times from the polynomial fit. Notice that a deviation threshold of 1.5 μs can be used to differentiate between the flawed and unflawed regions of the plate.

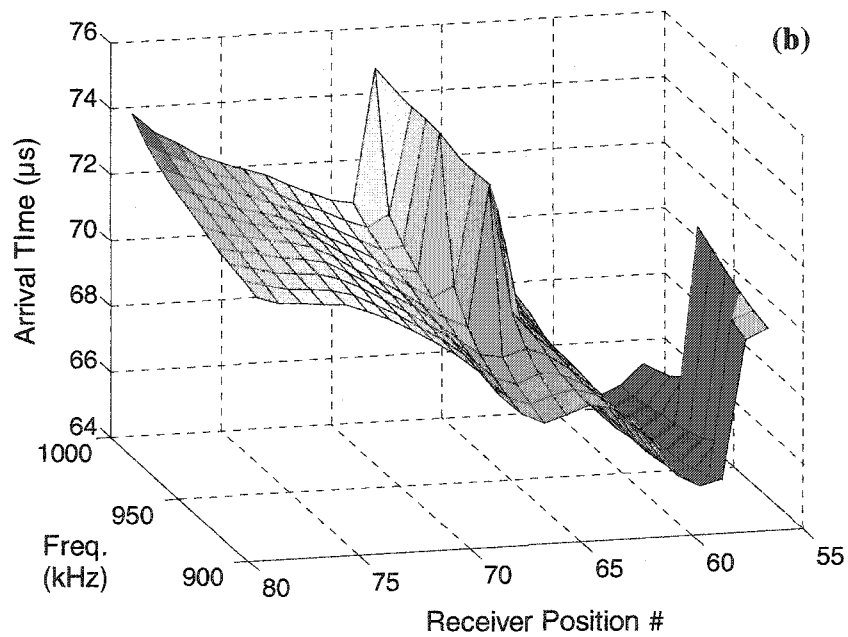
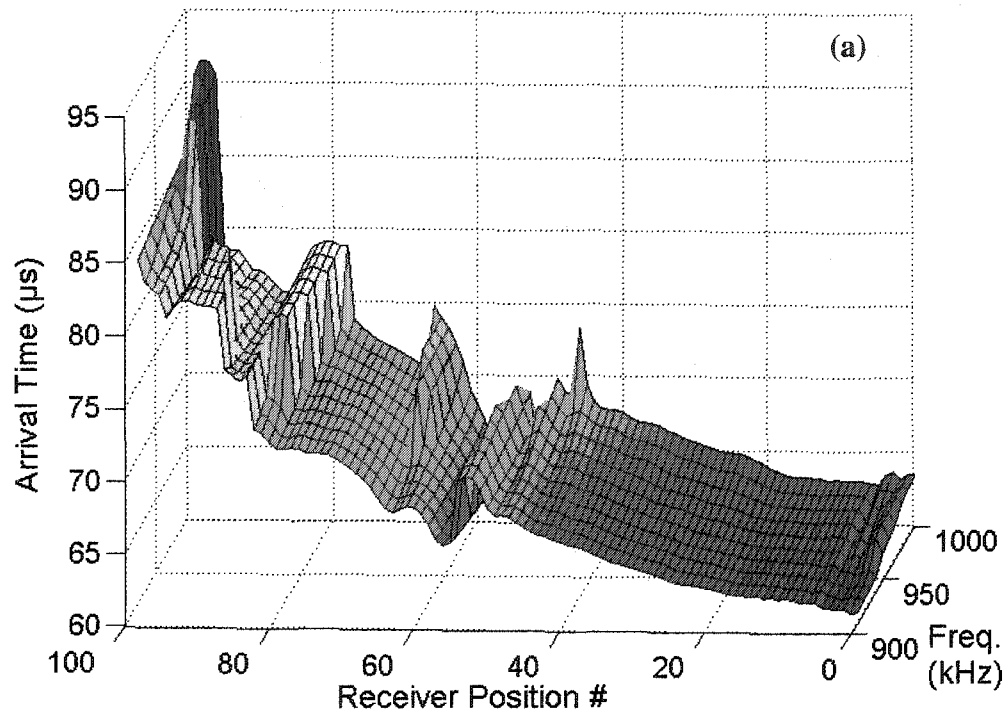


Figure 5-20. (a) Frequency walk data for the first 100 arrivals of the first arriving mode in the flat bottomed hole sample. (b) Arrival times for the "flawed" region between receiver positions #55 and #80.

greater than $1.5 \mu\text{s}$. Therefore, in order to delineate the suspected “clean” arrivals from suspected “flaw” arrivals, a deviation of $1.5 \mu\text{s}$ from the expected arrival was used as a threshold level.

Once the arrival times are able to be separated, it is still necessary to determine which mode they belong to and whether they are actual arrivals or erroneous data points. Figure 5-20(a) shows the first 100 points of the frequency walk data for the first mode’s arrival time in the same flat-bottomed hole sample as above. Some frequencies interact with the flaw before the other frequencies. Also, due to coupling, mode interference, or other factors, in the middle of the flawed area the first arriving peak in the higher frequencies can be seen to jump from the first mode to the second mode. Thus, even within the suspected flaw regions the data needs to be sorted so that the correct arrivals are included in the correct mode series.

The frequency walk data actually provides enough information to determine which mode series the arrival times should belong to. Figure 5-21(a) is a plot of the first three arrival times versus frequency number for pitch-catch position #65 (where the first peak’s arrival for the higher frequencies seems to jump to the second mode). From this graph it is seen that the first arrival for the higher frequencies, along with the second arrival for the lower frequencies, form a line. This shows that something is occurring to mask the first mode arrival and that the higher frequencies are actually locating the second mode as the first arriving peak. In the same way, Figure 5-21(b) shows a similar plot for waveform #598 (the 98th receive position for the 5th transmitter position) in the flawed sample. In this graph, it can be seen that the first arrival for 900 kHz was missed. Therefore, the first arrival will have been marked as a potential flaw area by the sorting

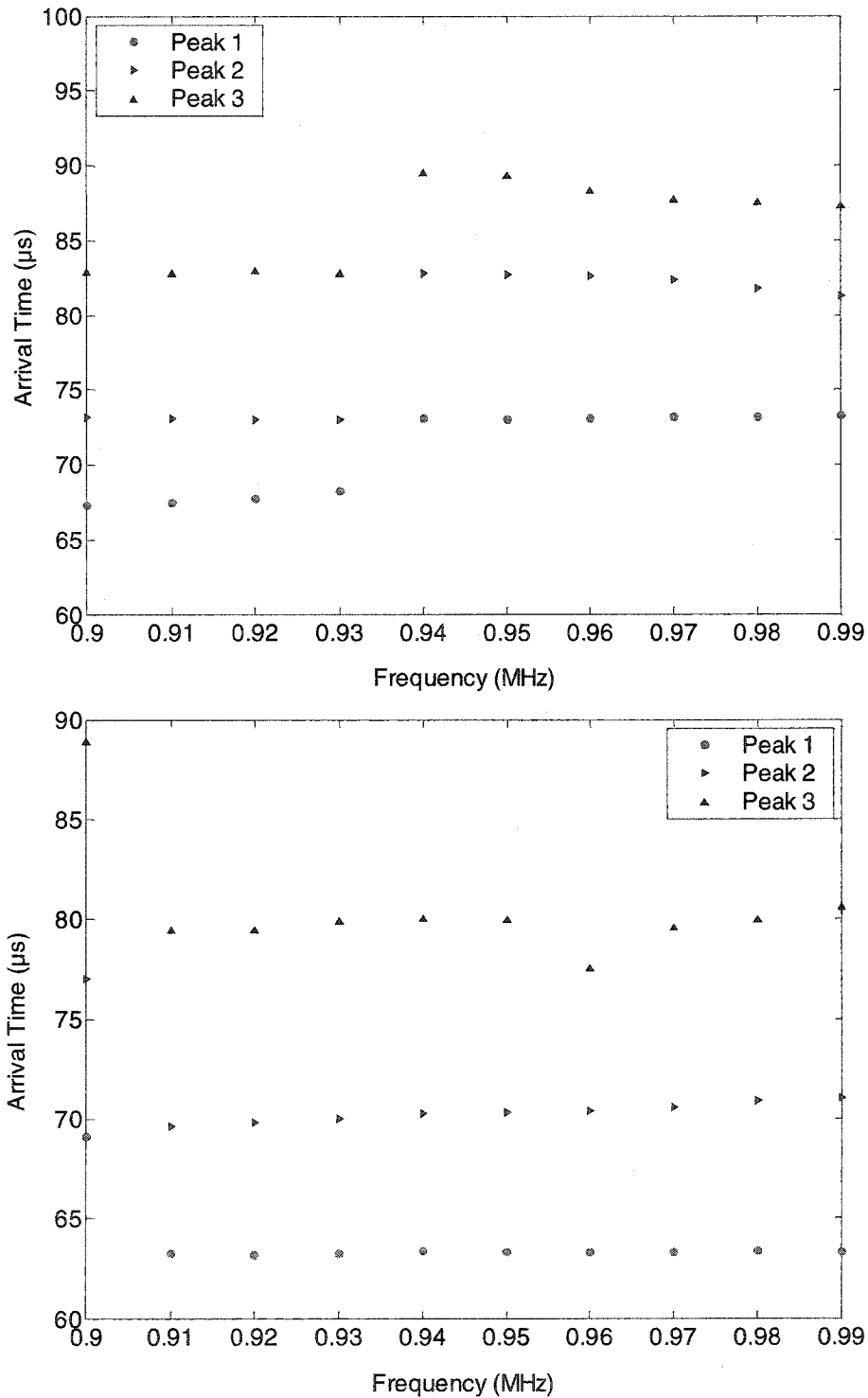


Figure 5-21. (a) Arrival time vs. frequency for the first three envelope peaks in signal #65. This waveform was chosen because it interacts with the flaw and an anomaly was seen in the frequency walk data in Figure 5-20. (b) Arrival time vs. frequency for signal #598 where only the first frequency missed the first arriving mode.

algorithm. However, the frequency data shows that the first arrival actually belongs with the second mode and can be resorted without affecting the resulting reconstruction. Therefore, using both the predicted mode curves and the frequency data, the multiple peak arrival times can be sorted into the correct mode series both inside and outside a flawed region. First, each of the original peak arrival series is fitted with a third-order polynomial. Then each arrival time is compared with these expected mode arrivals. If the deviation between the actual arrival and the expected arrival is greater than $1.5 \mu\text{s}$, then the data point is marked as a potential flaw area. After all the arrivals of each frequency scan have been marked, the predicted mode values are updated by fitting only the arrivals which were not marked as a potential flaw. The non-flaw points are then sorted by the same algorithm used above to sort the clean plate arrivals. Finally, the flaw points are sorted using the frequency walk data. By sorting the frequency data into linear series as in Figure 5-21, the arrival times within a flaw region can be sorted to the correct mode series.

Figure 5-22 shows the results of the sorting algorithm for the frequency walk data of the first 100 pitch-catch positions on the flat-bottomed hole sample. Figure 5-23 also shows the effects of the sorting algorithm on the tomographic reconstructions of the flat bottomed hole sample for the first three modes. These reconstructions are for only the two parallel crosshole projections as opposed to the full six projections in multiple crosshole tomography so there are some unavoidable artifacts in the reconstructed images. The goal of the sorting algorithm is to improve the multi-mode extraction for the single parallel crosshole pipe geometry, so we focused only on the parallel crosshole projections. It can be seen that the flaws are more accurately sized by the sorted data

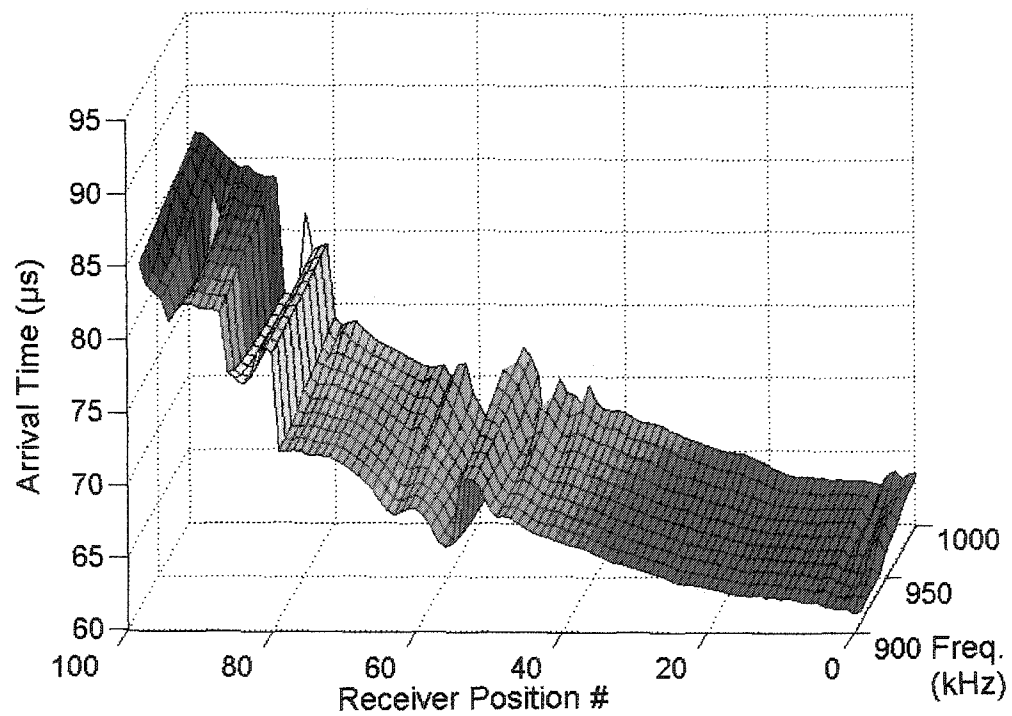
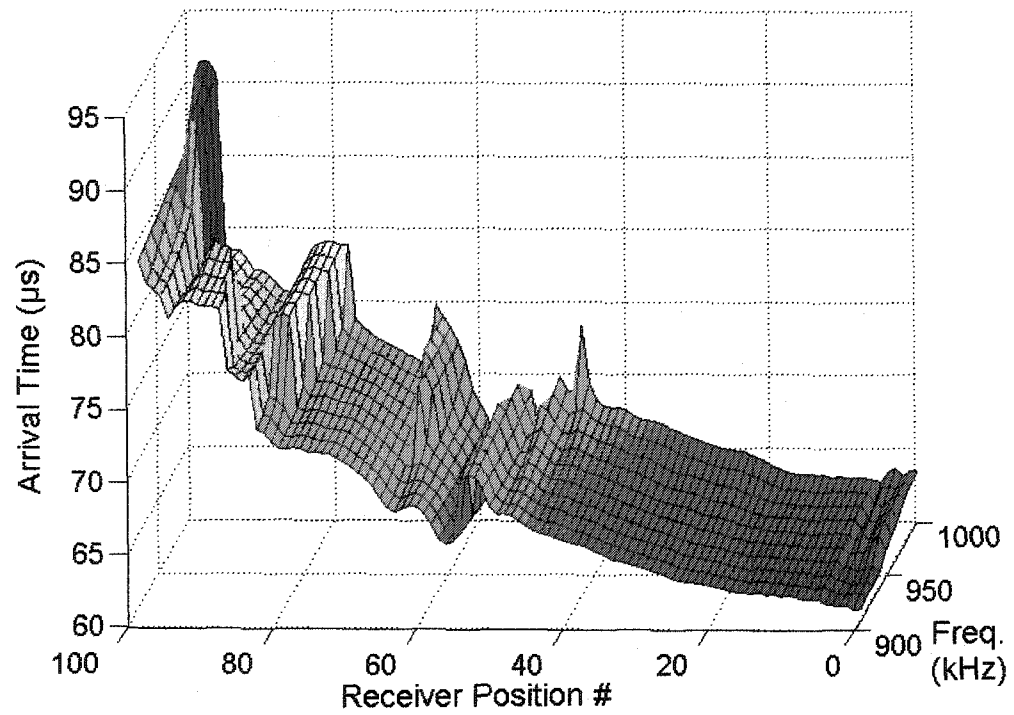


Figure 5-22. Comparison of frequency walk data for the first arriving mode before sorting (top) and after sorting (bottom).

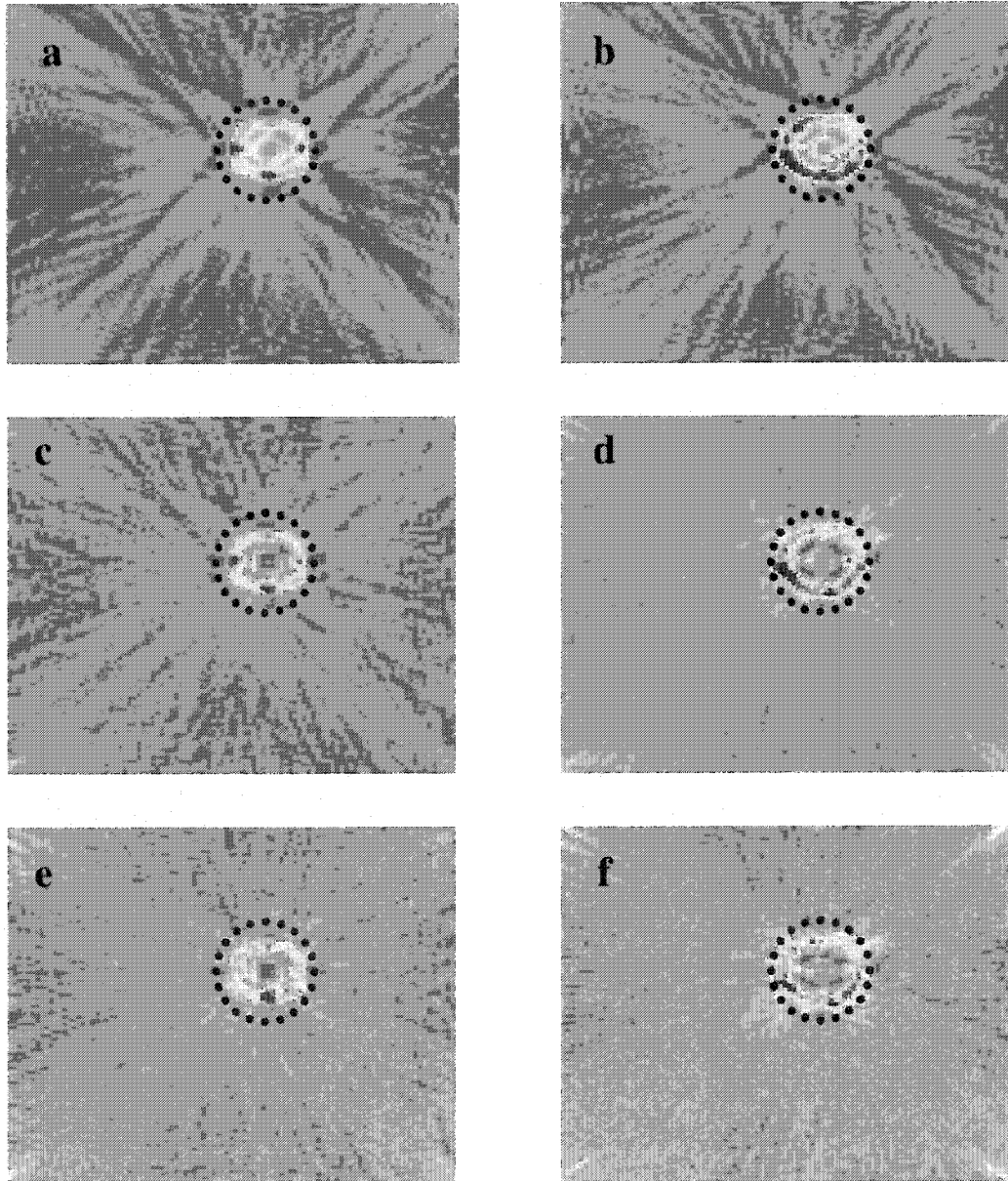


Figure 5-23. Tomographic reconstructions for the first three modes of a 3.17 mm thick plate sample with a 2.2" diameter flat bottomed hole. Thickness loss within the flaw was 50%. (a) Mode 1, unsorted. (b) Mode 1, sorted. (c) Mode 2, unsorted. (d) Mode 2, sorted. (e) Mode 3, unsorted. (f) Mode 3, sorted. The dashed line in each reconstruction represents the actual location and size of the flaw.

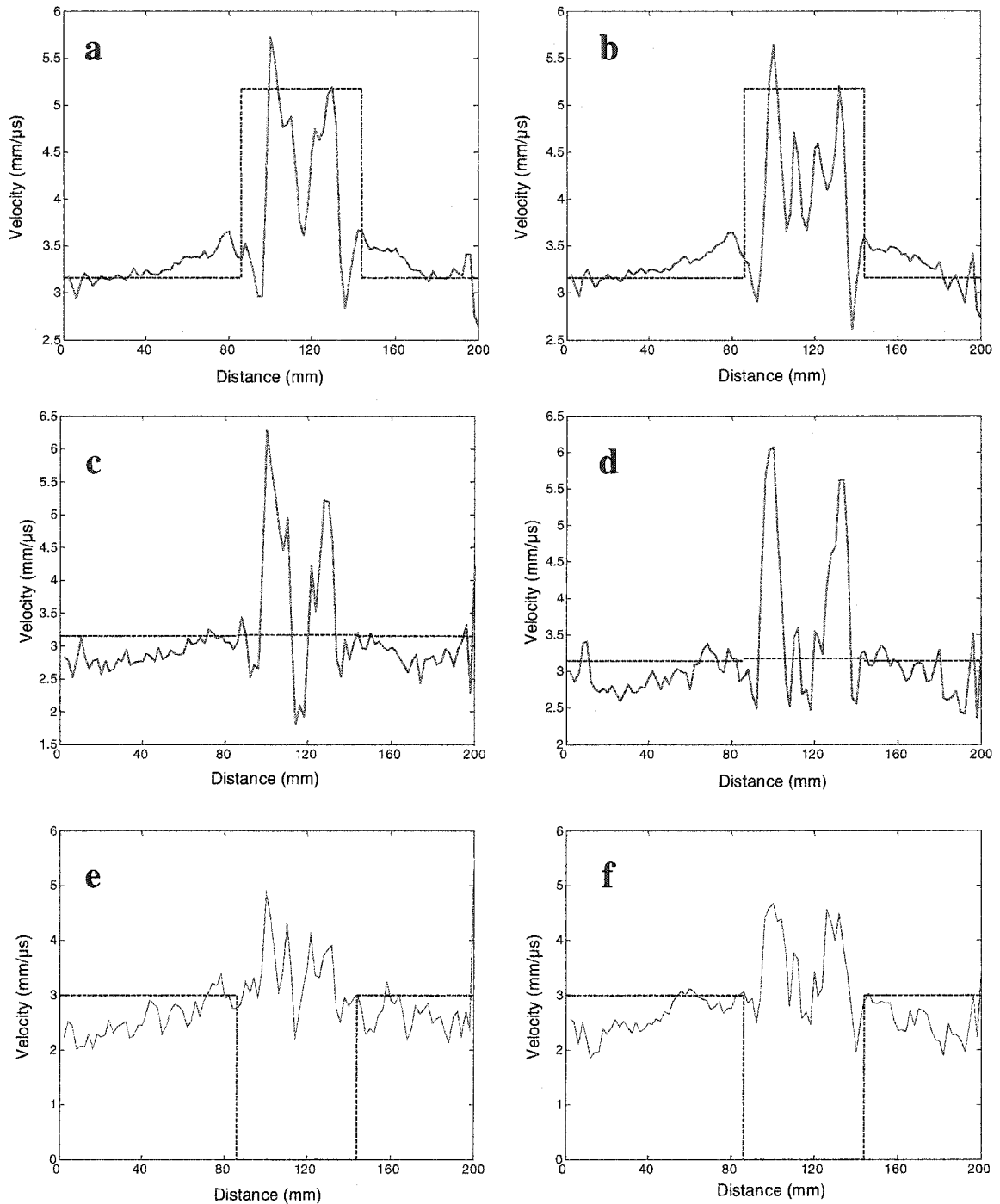


Figure 5-24. Velocity line plots for the first three modes of a vertical line through the center of the flat-bottomed hole. The dashed line represents the theoretical mode velocity for the measured frequency-thickness product. The red portion highlights the flawed region. (a) Mode 1, unsorted. (b) Mode 1, sorted. (c) Mode 2, unsorted. (d) Mode 2, sorted. (e) Mode 3, unsorted. (f) Mode 3, sorted. The second mode, A_0 , is not dispersive and the velocity does not change much in and out of the flawed region. The fd product within the flaw is also below the cutoff frequency for the third mode so its velocity in the flawed region should be zero.

than the unsorted mode data. This is also illustrated in Figure 5-24 where a velocity line plot is given for a vertical line that runs through the center of the flaw. The dashed lines in Figure 5-24 represent the theoretical velocities for the measured frequency-thickness (fd) product both within and outside of the flaw. The second arriving mode, A_0 , is not very dispersive and thus its theoretical velocity does not change much within the flaw. However, the reconstructed data shows a higher velocity within the flaw. This is most likely due to mode conversion and scattering at the flaw. Similarly, the fd product within the flaw is also below the cutoff frequency of the third mode, A_1 . Again, the velocity profile most likely shows a non-zero value because of mode conversion, scattering, and beam spreading.

Similar results have been found for other flawed samples. Figure 5-25 shows the reconstructions for the first 3 modes of a sample with a gradually dished out flat-bottomed hole the same diameter as the flat bottomed hole sample above (Plate #5 from blind study). The maximum thickness loss at the center of this flaw was 60%. The plate's thickness was again 3.17 mm. In addition, Figure 5-26 shows the same type of reconstructions for a sample with a 54 mm x 30 mm thinned rectangular region with rounded corners (Plate #6). This flaw was more difficult to detect because the thinned area was only a 10% thickness loss. Finally, Figure 5-27 shows a 2" flat-bottomed hole with a 12% thickness loss (Plate #9). It can be seen for this subtle flaw that the sorting algorithm improved the overall shape of the flaw for the first mode.

In order to quantitatively analyze the effectiveness of the sorting algorithm in improving the reconstructed images, the same image quality parameters used with frequency compounding – CNR, FSNR, and SNR – were applied to the sorted images.

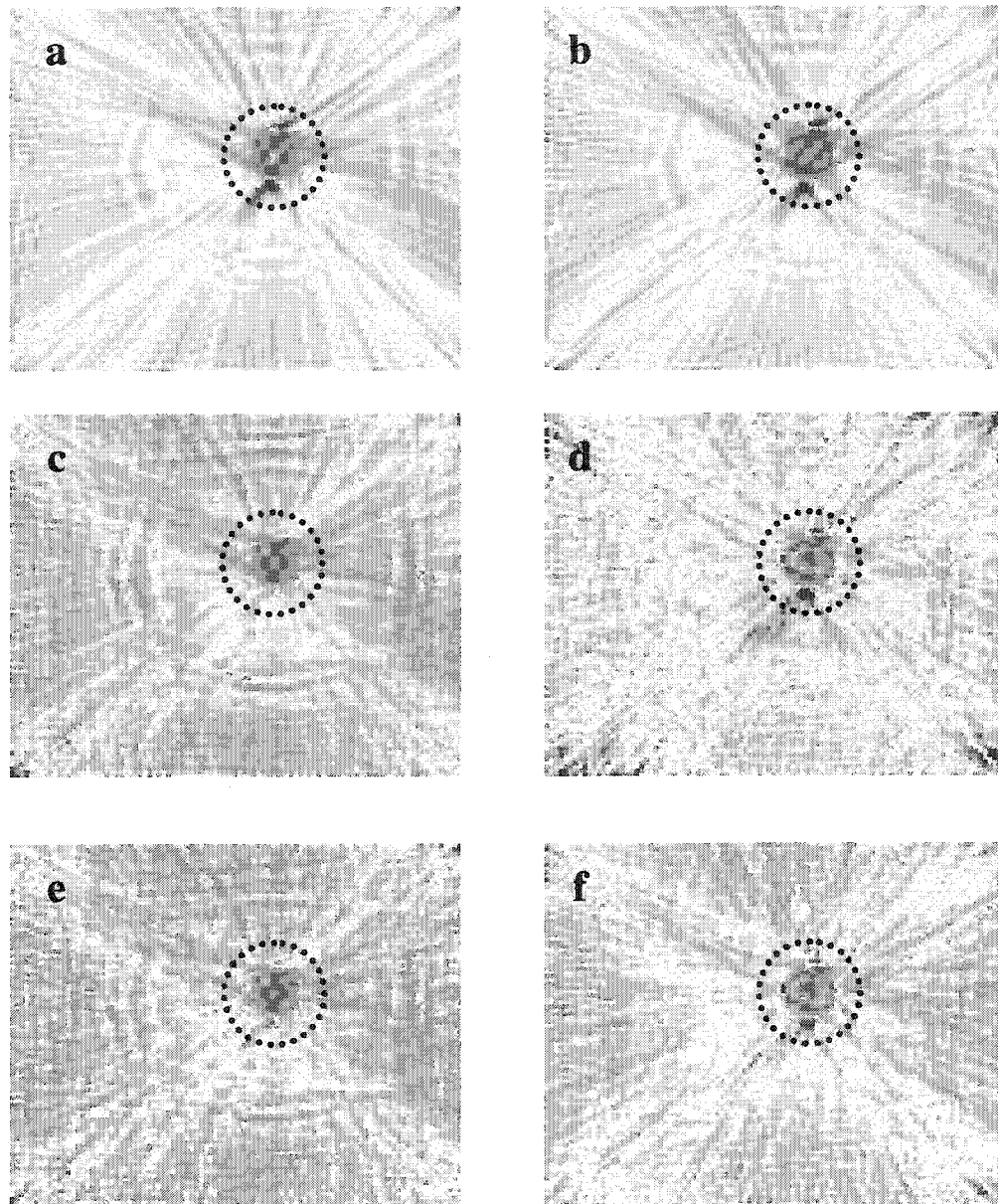


Figure 5-25. Tomographic reconstructions for the first three modes of a 3.17 mm thick plate sample with a 2.2" diameter successively dished-out flat bottomed hole. Maximum thickness loss within the center of the flaw was 60%. (a) Mode 1, unsorted. (b) Mode 1, sorted. (c) Mode 2, unsorted. (d) Mode 2, sorted. (e) Mode 3, unsorted. (f) Mode 3, sorted. The dashed line in each reconstruction represents the actual location and size of the flaw.

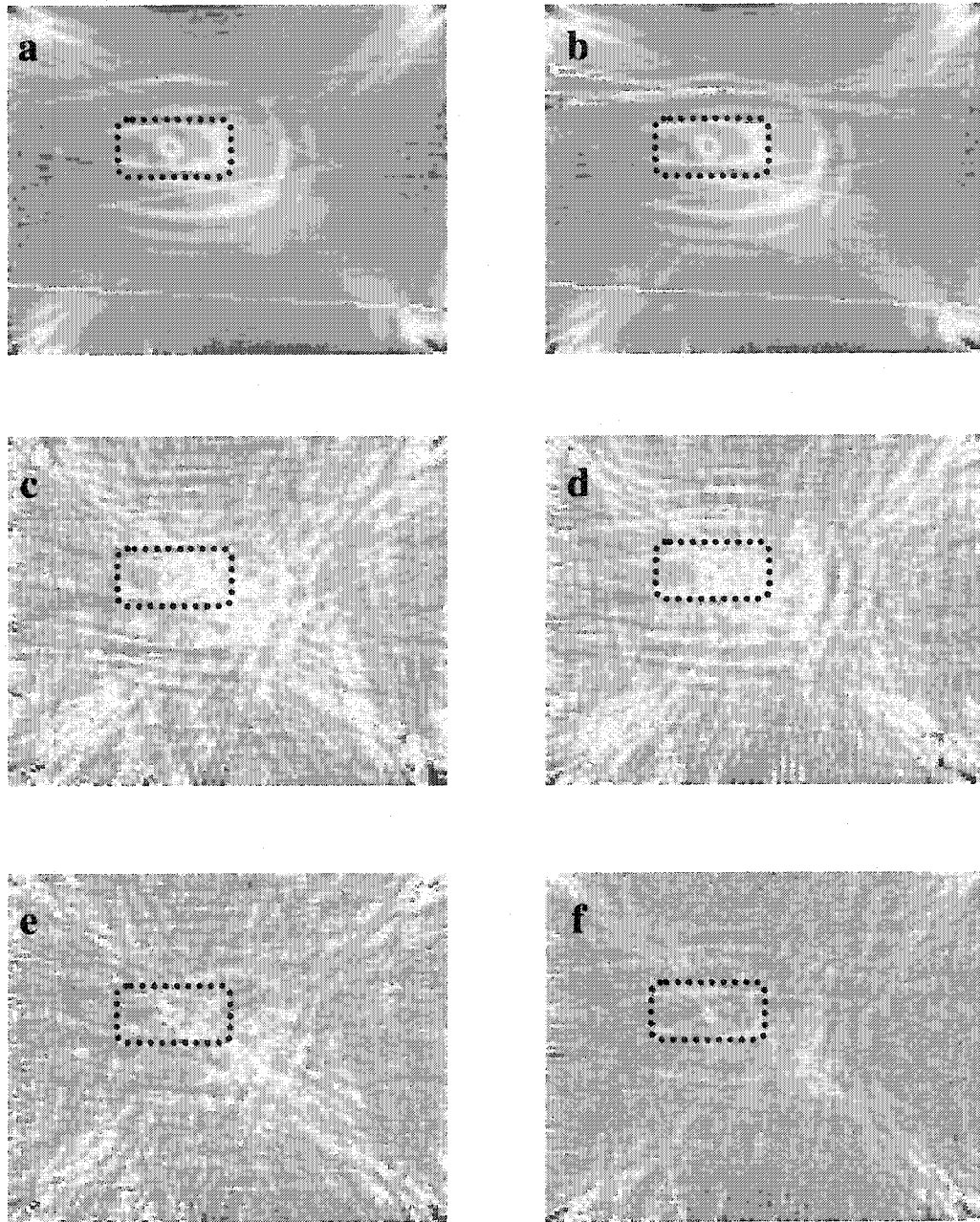


Figure 5-26. Tomographic reconstructions for the first three modes of a 3.17 mm thick plate sample with a 54 mm x 30 mm rectangular thinned region with rounded corners. Thickness loss was only 10% within the flaw. (a) Mode 1, unsorted. (b) Mode 1, sorted. (c) Mode 2, unsorted. (d) Mode 2, sorted. (e) Mode 3, unsorted. (f) Mode 3, sorted. The dashed line in each reconstruction represents the actual location and size of the flaw.

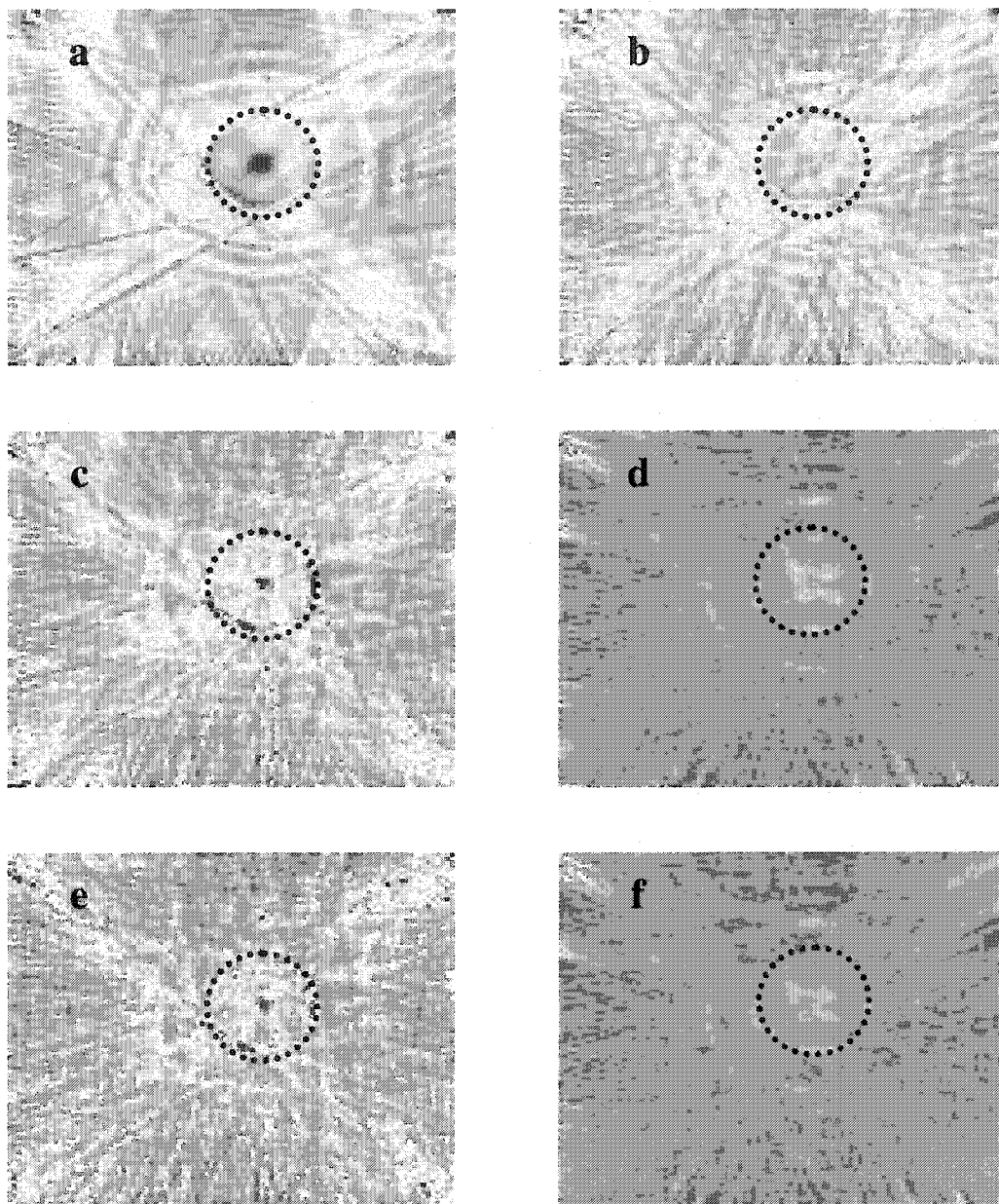


Figure 5-27. Tomographic reconstructions for the first three modes of a 3.17 mm thick plate sample with a 51 mm diameter circular flat-bottomed hole. Thickness loss was only 12% within the flaw. (a) Mode 1, unsorted. (b) Mode 1, sorted. (c) Mode 2, unsorted. (d) Mode 2, sorted. (e) Mode 3, unsorted. (f) Mode 3, sorted. The dashed line in each reconstruction represents the actual location and size of the flaw.

Table 5-3. Image Quality Parameters for the four flawed samples. (Plate 1 – Flat bottomed hole; Plate 2 – Successively dished out flat bottomed hole; Plate 3 – Rectangular thinning with rounded corners; Plate 4 – Thin flat bottomed hole, 2” diameter)

	Unsorted Plate 1 -- 900 kHz			Sorted Plate 1 - 900 kHz			% Difference		
	Mode1	Mode2	Mode3	Mode1	Mode2	Mode3	1	2	3
CNR	2.56	1.80	1.36	2.85	2.36	1.85	11	31	36
FSNR	0.76	0.70	0.65	0.89	0.80	0.82	16	14	26
SNR	14.28	7.25	6.35	15.27	8.30	7.06	7	14	11
	Unsorted Plate 2 -- 900 kHz			Sorted Plate 2 -- 900 kHz			% Difference		
	Mode1	Mode2	Mode3	Mode1	Mode2	Mode3	1	2	3
CNR	-0.60	-0.34	-0.09	-0.72	-0.42	-0.10	19	23	4
FSNR	-0.24	-0.19	-0.06	-0.28	-0.21	-0.05	18	10	-11
SNR	10.35	7.51	6.29	10.05	9.06	8.57	-3	21	36
	Unsorted Plate 3 -- 900 kHz			Sorted Plate 3 -- 900 kHz			% Difference		
	Mode1	Mode2	Mode3	Mode1	Mode2	Mode3	1	2	3
CNR	0.79	0.52	0.32	0.71	0.63	0.42	-9	22	30
FSNR	0.63	0.39	0.25	0.58	0.51	0.33	-9	29	30
SNR	20.71	9.20	7.33	20.20	10.04	8.41	-2	9	15
	Unsorted Plate 4 -- 900 kHz			Sorted Plate 4 -- 900 kHz			% Difference		
	Mode1	Mode2	Mode3	Mode1	Mode2	Mode3	1	2	3
CNR	1.44	0.84	0.57	1.20	0.84	0.77	-17	0	35
FSNR	1.13	0.64	0.44	0.92	0.65	0.59	-19	2	34
SNR	18.6	7.94	6.67	16.93	5.57	6.10	-9	-30	-9

Table 5-3 summarizes their results. It can be seen that the sorting algorithm does improve the overall image quality of the individual modes for each sample. From this data it can be seen that the sorting algorithm improves the contrast-to-noise ratio of the flaw for the second two modes the best. In terms of the contrast-to-noise ratio, only the first mode's reconstruction for the subtle rectangular thinning was not improved after the data was sorted.

5.3 Discussion

In this chapter, two new methods were presented to attempt to improve the tomographic reconstructions for LWT and HUT. Tomographic frequency compounding was shown to greatly reduce the noise in reconstructions from a thick walled pipe sample with a flaw on the ID. It also enhanced the contrast between the flawed and unflawed areas. This ultimately can enable better automatic flaw recognition. In addition, it also demonstrated that even without knowing *a priori* what frequency is best for the scanning geometry, that representative reconstructions of high quality can still be made. This is especially advantageous for the general use of this technique for rapid scanning of piping systems.

Mode sorting was introduced in order to improve the arrival time extraction of multi-mode guided wave signals. The method was demonstrated on plate samples with different flaws and was shown to improve the accuracy of the reconstructions. Because this work was done in parallel with [117], only a limited amount of data was available for comparison. However, as can be seen from Figure 5-28, the sorting algorithm improves the multi-mode arrivals and resulting reconstructions for the arrival times extracted using the dynamic wavelet fingerprinting technique.

Ultimately, it is desired to locate the modes that travel around the pipe multiple times. This work, along with [117], have laid the groundwork for this to be done. The simple peak detection algorithm used to generate the multi-mode arrivals in this chapter did not provide accurate enough arrival times from the more complicated pipe signals for the sorting algorithm to work. However, the DWFP technique has the capability to extract these more accurate arrivals and to ignore the unwanted pipe modes that exist in

the signal. Combined, the two techniques have the potential to lead to even better reconstructions that accurately locate and size flaws in both pipes and plates.

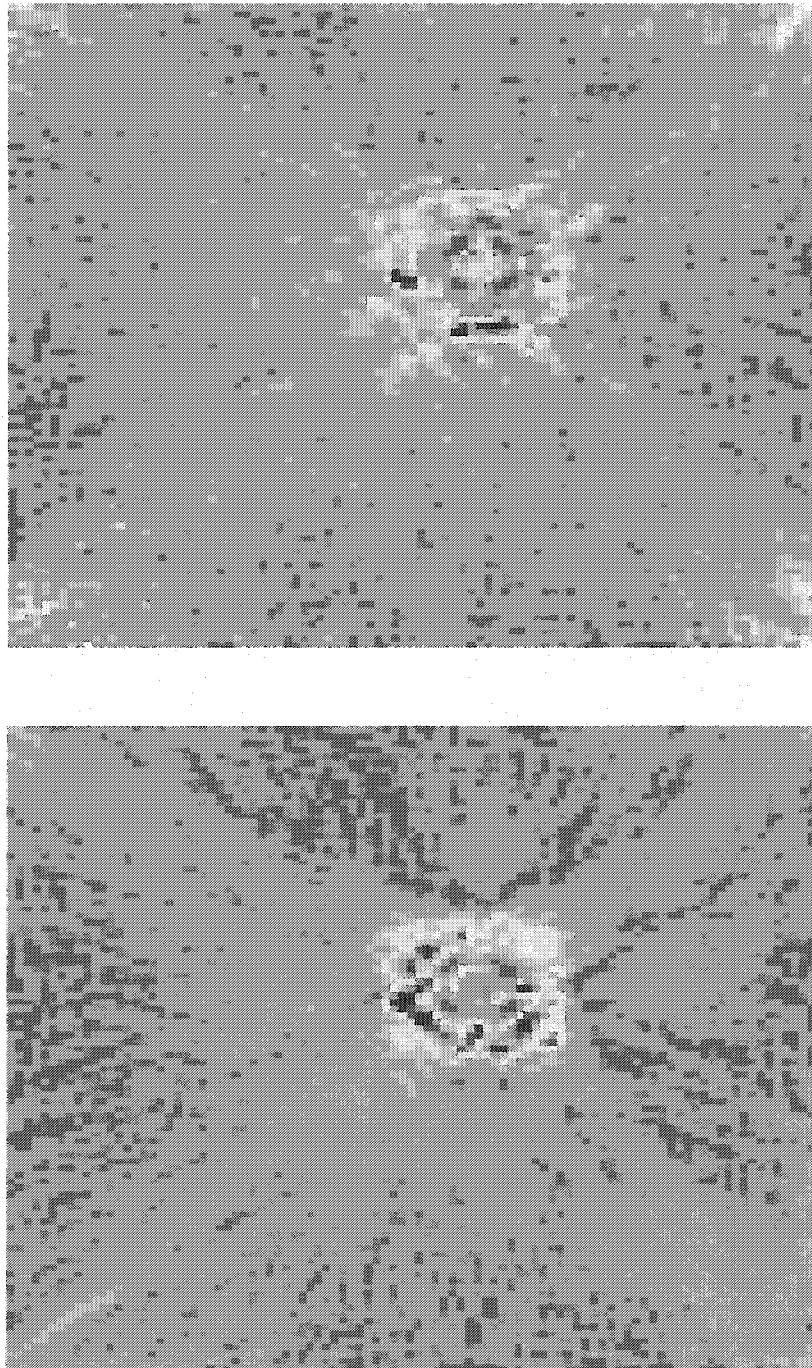


Figure 5-28. Comparison of DWFP time extraction before sorting (top) and after sorting (bottom). Reconstructions are of the flat bottomed hole sample at 990 kHz.

Chapter 6

Conclusions and Future Work

The main motivation of this work was to extend the current Lamb wave tomographic scanning system to pipe-like geometries. Ultrasonic guided waves can interrogate large areas quickly and quantitatively. Furthermore, because their propagation involves the entire thickness of the structure, they can be used to inspect hard to reach or inaccessible areas. Pipes are often placed underground or covered by insulation, but ultrasonic guided waves provide an efficient means to inspect these structures quickly and cost effectively without excavating or removing insulation. However, extracting the necessary information from the guided wave signals and rendering it in an effective manner can often be complicated.

Helical Ultrasound Tomography (HUT) was shown to be an effective solution to this problem. By adapting the Lamb wave tomography (LWT) system used to inspect plate-like structures, a scanning system was developed that was able to inspect cylindrical structures. Furthermore, the tomographic imaging techniques provide an effective way to render the ultrasonic data. The images provide a false-color image of the inspected area and flaws can readily be located and sized.

Initial tests were performed on a steel pipe segment with different sized flaws machined on the outside of the structure. As the flaw was enlarged on the sample, the resulting reconstructions effectively showed the changes in the sample. In addition, it was seen that the single crosshole geometry seemed to exaggerate the flaw size slightly because of the ineffective ray vector coverage. The reconstructions also showed artifacts around the edge of the flaws that were due to scattering. Overall the technique was proven to be useful for locating and sizing flaws in pipe-like structures.

In addition to the initial steel pipe segment with a flaw on the outside diameter, a thicker steel pipe with a flaw on the inner diameter was also scanned. This sample showed that the HUT technique is capable of detecting flaws on the inside diameter when only the outside circumference can be accessed. Furthermore, a tomographic frequency compounding technique was introduced that provides clearer reconstructions. This is an added benefit for the HUT technique because it smoothes out some of the noise and artifacts that appear in the HUT reconstructions due to the more complicated cylindrical geometry as compared to the plate geometry of LWT.

In addition to the HUT technique, another tomographic geometry capable of scanning cylindrical pipes and tanks was introduced. Instead of using two circumferential transducer array belts, Meridional Ultrasound Tomography (MUT) uses a linear array of transducers along the axis of the cylinder. This enables a different inspection geometry and is useful if the entire circumference is not accessible. One immediate application for the MUT technique is for the inspection of large, stacked depleted uranium hexafluoride storage tanks. Because the tanks are stacked on top of one another, the only accessible area is an axial line along the top of the tank.

Through tests on an aluminum pipe, it was shown that MUT can be used to image corrosion and other flaws on the inner diameter. Unlike HUT and LWT, which used arrival time data, MUT used attenuation data as an input to the tomographic reconstruction algorithms. Since the mode of interest was not the first arriving mode, the existing arrival time extraction algorithms were unable to reliably measure its time-of-flight. However, by gating the signal around the mode of interest, and smoothing the amplitude data, tomographic images were obtained that reliably showed the presence of a flaw.

These initial experiences with both HUT and MUT demonstrated the need for improved reconstruction and data analysis. In both cases the cylindrical geometry eliminates the added scanning orientations available in LWT. In other words, the technique is limited to only the single crosshole geometry. However, because the pipe can be viewed as a plate wrapped around upon itself, waves that travel multiple times around the structure can be used to provide the additional information from different orientations needed to generate more accurate reconstructions.

Furthermore, as the LWT blind test results demonstrated, certain reconstructions did not represent the flaws accurately and some flaws weren't able to be detected. Since the LWT technique only relies on the arrival time of the first mode, its detection capabilities are limited. Different modes have different displacement properties and are thus able to detect different types and locations of flaws. If the arrival times for more than just the first arriving mode can be extracted, then more information can be provided to the tomographic algorithms to improve the resulting reconstructions.

For this reason, the balance of this work focused on developing an improved time extraction algorithm capable of detecting multiple modes within the guided wave signal. Dynamic wavelet fingerprinting has shown promise for being able to separate out these different mode arrivals. Therefore, a sorting algorithm was developed that uses multiple frequency scans to better determine which mode individual arrivals belong to. Due to dispersion and interference between modes, information can be lost in the signal. The DWFP algorithm assumes that each of the modes arrives in the same order, but it was shown that this is not always the case. The mode sorting algorithm introduced in this work successfully demonstrated that the arrivals can be sorted to their correct mode series and that this improves the overall quality of the tomographic reconstructions.

The sorting algorithm was demonstrated on four of the plate samples from the blind study test. It was seen that after sorting the arrival times, the reconstructions did a better job of sizing the flaws. Another result from these tests was that after sorting, the tomographic data better represented the actual velocity data within the flawed area for the first mode. The deeper flaws showed up better in the reconstructions, but scattering from the edges of the flaw caused the reconstructed mode velocities within the flaw to be less accurate for the later modes. In contrast, the sorting algorithm improved the quantitative accuracy – both in size and velocity reconstruction – for the subtle flaws, but the actual reconstructed images for the second and third modes did not highlight the flaw as well. However, this demonstrates the validity of the statement that the individual modes will interact with the flaws differently. The second mode is non-dispersive around the frequency-thickness product that we are operating and thus we would not expect to see

the flaw in these reconstructions anyway. The fact that the flaw is seen in the reconstructed images and velocity data is because of scattering and mode conversion.

Ultimately, our goal is to extract the arrival times of the modes that travel around the pipe multiple times. Future work needs to pursue the application of the sorting and frequency compounding algorithms combined with the DWFP technique to HUT. In order for the steeper helical arrivals to be obtained, a systematic study needs to be conducted on pipe data in order to obtain the correct fingerprint patterns for the different arrivals. Furthermore, as suggested by Hou [117], different wavelets need to be studied to determine which basis is best to solve this problem.

Finally, a systematic theoretical and experimental study of a representative flaw would be extremely beneficial. Since the calculations of the theoretical arrival times of the various modes in a flawed sample are nontrivial, it is hard to provide an accurate measure of the success of the multi-mode extraction algorithms. If a systematic study was done with a simple type of flaw that was slowly enlarged, both theoretical simulations – such as FEM, BEM, FDTD, etc. – and experimental data could be collected. This would allow for a comparison of the DWFP and sorting algorithms to theoretical data and would provide a clearer understanding of how the individual guided wave modes interact with a characteristic flaw.

Helical ultrasound tomography and meridional ultrasound tomography have been shown to be effective ways to inspect cylindrical structures. Furthermore, image and signal processing techniques such as frequency compounding and mode sorting have been shown to improve the resulting tomographic images. These images demonstrate the existence of flaws in various materials and they can be used to accurately size and

determine the severity of the flaws. In the future, these techniques will also be the basis for a more sophisticated analysis of the cylindrical geometries that will allow even more accurate reconstructions to be produced.

Appendix A

Source Code for the Numerical Solution of the Lamb Wave Dispersion Relations

```
/******BEGIN LAMBLIB.H******/

#ifdef LAMBLIB

//LAMBLIB should be defined in all of the DLL's source code
//files before this header file is included.

//All functions/variables are being exported.

#else

//This header file is included by the EXE source code
//files.

//Indicate that all functions/variables are being imported.
#define LAMBLIB __declspec(dllimport)

#endif

/******
Define any data structures and symbols here.
******/
typedef struct FCOMPLEX { double r, i; } fcomplex;
typedef vector<double> doubles;

/******
Define exported variables here: (Note: Avoid exporting
variables if possible)
******/
//None

/******
End exported variables.
******/
```

```

/*****
Define exported function prototypes here:
*****/

//Definitions for file Complex.cpp
LAMBLIB fcomplex complex_sqrt(double a);
LAMBLIB fcomplex complex_sin(fcomplex a);
LAMBLIB fcomplex complex_cos(fcomplex a);
LAMBLIB fcomplex Cadd(fcomplex a, fcomplex b);
LAMBLIB fcomplex Csub(fcomplex a, fcomplex b);
LAMBLIB fcomplex Cmul(fcomplex a, fcomplex b);
LAMBLIB fcomplex Complex(double re, double im);
LAMBLIB fcomplex Conjg(fcomplex z);
LAMBLIB fcomplex Cdiv(fcomplex a, fcomplex b);
LAMBLIB double Cabs(fcomplex z);
LAMBLIB fcomplex Csqrt(fcomplex z);
LAMBLIB fcomplex RCmul(double x, fcomplex a);
//end Complex.cpp

//Definitions for file EvalLambEqn.cpppp
LAMBLIB double EvalEqn_V(double freq, double Vph, double cL, double cT,
                        double half_thick, int type);
LAMBLIB double EvalEqn(double freq, double k, double cL, double cT,
                        double half_thick, int type);
LAMBLIB double SymEqn(double freq, double k, double cL, double cT,
                       double half_thick);
LAMBLIB double AsymEqn(double freq, double k, double cL, double cT,
                        double half_thick);
//end EvalLambEqn.cpppp

//Definitions for file MinimumSweep.cpp
LAMBLIB vector<doubles> EvalSweep(double freq1, double Vph1, double freq2,
                                double Vph2, double freq_step, double cL,
                                double cT, double half_thick, int type);
LAMBLIB vector<doubles> FindSweepMins(vector<doubles> eval_vector);
LAMBLIB vector<doubles> FindMinRoots(vector<doubles> min_vector, double cL,
                                    double cT, double step_size, double
                                    tolerance, double half_thick, int type);
LAMBLIB vector<doubles> MinimumSweep(double freq1, double Vph1, double freq2,
                                    double Vph2, double freq_step, double cL,
                                    double cT, double half_thick, double
                                    type);
//end MinimumSweep.cpp

//Definitions for file Trace.cpp
LAMBLIB vector<double> GetRoot_vf(double freq, double Vph, double cL, double cT,

```

```

        double step_size, double tolerance,
        double half_thick, double type);
LAMBLIB vector<double> GetRoot_fk(double freq, double k, double cL, double cT,
        step_size, double k_step_size, double tolerance,
        double half_thick, double type, double
        percentage);
LAMBLIB vector<doubles> TraceModeBack(vector<doubles> curve, double
        k_step_size, double f_step_size, double
        cL, double cT, double half_thick, int type,
        double tolerance);
LAMBLIB vector<doubles> TraceMode(double freq1, double Vph, double k_step_size,
        f_step_size, double cL, double cT, double
        half_thick, int type, double tolerance, int
        fundamental, double max_freq);
LAMBLIB vector<double> LinearExtrapolateRoot(vector<double> point1,
        vector<double> point2,
        double k_step_size);
LAMBLIB vector<double> QuadExtrapolateRoot(vector<double> point1,
        vector<double> point2,
        vector<double> point3,
        vector<double> point4,
        double k_step_size);
LAMBLIB vector<double> GetFirstRoot(double freq, double k, double cL,
        double cT, double f_step_size, double
        tolerance, double half_thick, double type);

//end Trace.cpp

//Definitions for file Dispersion.cpp
LAMBLIB void GetDispersion(double cL, double cT, double thickness, double
        tolerance, double max_freq);

//end Dispersion.cpp

//Definitions for file group.cpp
LAMBLIB void group_velocity(vector<doubles> mode_data, FILE *group_out, double
        thickness);

//end group.cpp

/*****
End exported functions.
*****/

/*****END LAMBLIB.H*****/

```

```

/*****BEGIN GOLDEN.H*****/

/*****
This header file is taken mainly from Numerical Recipes in C.
It is the declarations for the functions used in bracketing a
minimum and then searching for the minimum in a fast and efficient
manner.
*****/

#ifndef _NR_UTILS_H_
#define _NR_UTILS_H_

#define GOLD 1.618034 //Successible ratio by which successive intervals are
                    //magnified
#define GLIMIT 100.0 //maximum magnification allowed for a parabolic-fit step
#define TINY 1.0e-20
#define SHFT(a,b,c,d) (a)=(b);(b)=(c);(c)=(d);

#define R 0.61803399
#define C (1.0-R)
#define SHFT2(a,b,c) (a)=(b);(b)=(c);

static double maxarg1, maxarg2;
#define FMAX(a,b) (maxarg1=(a),maxarg2=(b),(maxarg1) > (maxarg2) ? \
(maxarg1) : (maxarg2))

#define SIGN(a,b) ((b) >= 0.0 ? fabs(a) : -fabs(a))

double test_func(double x, double y);

double golden_search(double a, double b, double c, double x_curr, double cL,
                    double cT, double half_thick, int type,
                    double (*ftes)(double, double, double, double,
                    double, int), double tol, double *xmin);

void mnbrak(double *ax, double *bx, double *cx, double *fa, double *fb,
            double *fc, double x_curr, double cL, double cT, double half_thick, int type,
            double (*func)(double, double, double, double, double, int));

#endif

/*****END GOLDEN.H*****/

```

```

/*****BEGIN GOLDEN.CPP*****/

#include <cmath>
#include "golden.h"

/*****
Taken from Numerical Recipes in C: www.nr.com
Given a function 'func', and given distinct initial points 'ax' and 'bx', this routine
searches in the downhill direction (defined by the function as evaluated at the initial
points)
and return new points 'ax', 'bx', 'cx' that bracket a minimum of the function. Also
returned
are the function values at the three points, 'fa', 'fb', and 'fc'.
*****/

void mnbrak(double *ax, double *bx, double *cx, double *fa, double *fb, double *fc,
double x_curr, double cL, double cT, double half_thick, int type, double (*func)(double,
double, double, double, double, int)) {

double ulim, u, r, q, fu, dum;

*fa = (*func)(*ax, x_curr, cL, cT, half_thick, type);
*fb = (*func)(*bx, x_curr, cL, cT, half_thick, type);

if (*fb > *fa) {

    SHFT(dum,*ax,*bx,dum)    //Switch roles of a and b, so that we
    SHFT(dum,*fb,*fa,dum)    //can go downhill in the direction from
                            //a to b

}
*cx = (*bx) + GOLD*(*bx-*ax); //First guess for c.
*fc = (*func)(*cx, x_curr, cL, cT, half_thick, type);

while (*fb > *fc) {        //Loop until we bracket

    r = (*bx - *ax)*(*fb - *fc);
    q = (*bx - *cx)*(*fb - *fa);

    //Compute u by parabolic extrapolation from a,b,c. TINY is used to
    //prevent any possible division by zero.

    u = (*bx) - ((*bx - *cx)*q - (*bx - *ax)*r)/
        (2.0*SIGN(FMAX(fabs(q - r), TINY), (q - r)));
    ulim = (*bx) + GLIMIT*(*cx - *bx);

```

```

if ((*bx - u)*(u - *cx) > 0.0) { //Parabolic u is between b and
                                //c: try it.

    fu = (*func)(u, x_curr, cL, cT, half_thick, type);
    if (fu < *fc) { //Got a minimum between b and c.

        *ax = (*bx);
        *bx = u;
        *fa = (*fb);
        *fb = fu;
        return;

    } else if (fu > *fb) { //Got a minimum between a and u.

        *cx = u;
        *fc = fu;
        return;

    }

//Parabolic fit was no use, Use default magnification.
u = (*cx) + GOLD*( *cx - *bx);
fu = (*func)(u, x_curr, cL, cT, half_thick, type);

} else if (((*cx - u)*(u - ulim)) > 0.0) { //Parabolic fit is between c and its allowed
                                //limit.
    fu = (*func)(u, x_curr, cL, cT, half_thick, type);
    if (fu < *fc) {

        SHFT(*bx, *cx, u, *cx + GOLD*( *cx - *bx))
        SHFT(*fb, *fc, fu, (*func)(u, x_curr, cL, cT, half_thick, type))
    }

} else if (((u - ulim)*(ulim - *cx)) >= 0.0) { //Limit parabolic
                                // u to maximum allowed value.
    u = ulim;
    fu = (*func)(u, x_curr, cL, cT, half_thick, type);

} else { //Reject parabolic u, use default magnification

    u = (*cx) + GOLD*( *cx - *bx);
    fu = (*func)(u, x_curr, cL, cT, half_thick, type);
}
SHFT(*ax, *bx, *cx, u) //Eliminate oldest point and continue.
SHFT(*fa, *fb, *fc, fu)

```



```

}
}

/*****
Taken from Numerical Recipes in C: www.nr.com
This function takes as input 3 points which bracket a minimum.
It then searches for and returns the minimum.
*****/
double golden_search(double a, double b, double c, double x_curr,
                    double cL, double cT, double half_thick, int type,
                    double (*f)(double, double, double, double,
                    double, int), double tol, double *xmin) {

double f1, f2, x0, x1, x2, x3;    //At any given time we will keep
                                //track of four points,x0, x1, x2, x3.

x0 = a;
x3 = c;

//tol=.00001;

if (fabs(c - b) > fabs(b - a)) { //Make x0 to x1 the smaller segment,

x1 = b;
x2 = b + C*(c - b);           //and fill in the new point to be tried.

} else {

x2 = b;
x1 = b - C*(b - a);

}

f1 = (*f)(x1, x_curr, cL, cT, half_thick, type); //The initial function evaluations. Note
//we never need to
f2 = (*f)(x2, x_curr, cL, cT, half_thick, type); //evaluate the functions at the original
//endpoints.

while (fabs(x3 - x0) > (fabs(x1) + fabs(x2))*tol) {

if (f2 < f1) { //One possible outcome

SHFT(x0, x1, x2, R*x1 + C*x3)
SHFT2(f1, f2, (*f)(x2, x_curr, cL, cT, half_thick, type))

```

```

    } else {          //The other possible outcome

        SHFT(x3, x2, x1, R*x2 + C*x0)
        SHFT2(f2, f1, (*f)(x1, x_curr, cL, cT, half_thick, type))

    }

}          //Back to see if we are done.

if (f1 < f2) {      //We are done. Output the best of the two current values.

    *xmin = x1;
    return f1;

} else {

    *xmin = x2;
    return f2;

}

}

/*****END GOLDEN.CPP*****/

/*****BEGIN COMPLEX.CPP*****/

/*****
DLL Module: Complex.c
*****/

//Include the standard Windows and C-Runtime header files here.
#include <windows.h>
#include <cmath>
#include <vector>
#include <iterator>

using namespace std;

//This DLL source code file exports functions and variables.
#define LAMBLIB __declspec(dllexport)

//Include the exported data structures, symbols, functions, and variables.
#include "LambLib.h"

```

```
//Takes the square root of a negative number and  
//creates a complex number struct out of it.  
fcomplex complex_sqrt(double a) {
```

```
    fcomplex b;
```

```
    if (a < 0) {
```

```
        b.i = sqrt(-a);
```

```
        b.r = 0.0;
```

```
    }
```

```
    else {
```

```
        b.r = sqrt(a);
```

```
        b.i = 0.0;
```

```
    }
```

```
    return b;
```

```
}
```

```
//Evaluates sin() for a complex number struct  
fcomplex complex_sin(fcomplex a) {
```

```
    fcomplex b;
```

```
    b.r = sin(a.r)*cosh(a.i);
```

```
    b.i = cos(a.r)*sinh(a.i);
```

```
    return b;
```

```
}
```

```
//Evaluates cos() for a complex number struct  
fcomplex complex_cos(fcomplex a) {
```

```
    fcomplex b;
```

```
    b.r = cos(a.r)*cosh(a.i);
```

```
    b.i = -sin(a.r)*sinh(a.i);
```

```
    return b;
```

```
}
```

```
//Adds two complex number structs
fcomplex Cadd(fcomplex a, fcomplex b) {

    fcomplex c;
    c.r = a.r + b.r;
    c.i = a.i + b.i;
    return c;

}

//Subtracts two complex number structs
fcomplex Csub(fcomplex a, fcomplex b) {

    fcomplex c;
    c.r = a.r - b.r;
    c.i = a.i - b.i;
    return c;

}

//Multiplies two complex number structs
fcomplex Cmul(fcomplex a, fcomplex b) {

    fcomplex c;
    c.r = a.r*b.r - a.i*b.i;
    c.i = a.i*b.r + a.r*b.i;
    return c;

}

//Creates a complex number struct with the real and imaginary double
//values as input
fcomplex Complex(double re, double im) {

    fcomplex c;
    c.r = re;
    c.i = im;

    return c;

}

//Evaluates the conjugate of a complex number struct
fcomplex Conjg(fcomplex z) {
```

```

fcomplex c;
c.r = z.r;
c.i = -z.i;
return c;

```

```

}

```

```

//Divides two complex number structs
fcomplex Cdiv(fcomplex a, fcomplex b) {

```

```

    fcomplex c;
    double r, den;
    if (fabs(b.r) >= fabs(b.i) ) {
        r = b.i/b.r;
        den = b.r + r*b.i;
        c.r = (a.r + r*a.i)/den;
        c.i = (a.i - r*a.r)/den;
    }
    else {
        r = b.r/b.i;
        den = b.i + r*b.r;
        c.r = (a.r*r + a.i)/den;
        c.i = (a.i*r - a.r)/den;
    }
    return c;

```

```

}

```

```

//Evaluates the absolute value of a complex number struct
double Cabs(fcomplex z) {

```

```

    double x, y, ans, temp;
    x = fabs(z.r);
    y = fabs(z.i);

    if (x == 0.0)
        ans = y;
    else if (y == 0.0)

        ans = x;
    else if (x > y) {

        temp = y/x;
        ans = x * sqrt(1.0 + temp*temp);

```

```

} else {

    temp = x/y;
    ans = y * sqrt(1.0 + temp*temp);

}
return ans;

}

//Takes the square root of a complex number struct
fcomplex Csqrt(fcomplex z) {

    fcomplex c;
    double x, y, w, r;

    if ((z.r == 0.0) && (z.i == 0.0)) {
        c.r = 0.0;
        c.i = 0.0;
        return c;
    } else {

        x = fabs(z.r);
        y = fabs(z.i);

        if (x >= y) {
            r = y/x;
            w = sqrt(x)*sqrt(0.5*(1.0+sqrt(1.0+r*r)));
        } else {
            r = x/y;
            w = sqrt(y)*sqrt(0.5*(r+sqrt(1.0+r*r)));
        }

        if (z.r >= 0.0) {

            c.r = w;
            c.i = z.i/(2.0*w);

        } else {
            c.i = (z.i >= 0) ? w : -w;
            c.r = z.i/(2.0*c.i);
        }
        return c;
    }
}

```

```

//Multiplies a double with a complex number struct
fcomplex RCmul(double x, fcomplex a) {

    fcomplex c;
    c.r = x * a.r;
    c.i = x * a.i;
    return c;

}

/*****END COMPLEX.CPP*****/

/*****BEGIN TEST.CPP*****/

//Include the standard Windows and C-Runtime header files here.
#include <windows.h>
#include <iostream>
#include <cmath>
#include <vector>
#include <iterator>
#include <cstdio>
#include <cstdlib>

using namespace std;

//This DLL source code file exports functions and variables.
#define LAMBLIB __declspec(dllimport)

//Include the exported data structures, symbols, functions, and variables.
#include "LambLib.h"

//This executable takes 5 arguments in the command line.  Respectively
//Longitudinal Bulk Velocity, Transverse Bulk Velocity, Thickness, Tolerance,
//and Maximum Frequency.  It then computes the mode curves given this information
//and stores them in temp_curve.dat
int main(int argc, char* argv[]) {

    double cL;           //longitudinal bulk velocity
    double cT;           //transverse bulk velocity
    double thickness;    //material thickness
    double tolerance;    //solution tolerance
    double max_freq;     //maximum frequency desired

    //Get command line arguments
    cL = atof(argv[1]);
    cT = atof(argv[2]);

```

```

    thickness = atof(argv[3]);
    tolerance = atof(argv[4]);
    max_freq = atof(argv[5]);

    GetDispersion(cL, cT, thickness, tolerance, max_freq);

    return 0;
}

/*****END TEST.CPP*****/

/*****BEGIN DISPERSION.CPP*****/

//Include the standard Windows and C-Runtime header files here.
#include <windows.h>
#include <cmath>
#include <vector>
#include <iterator>

using namespace std;

//This DLL source code file exports functions and variables.
#define LAMBLIB __declspec(dllexport)

//Include the exported data structures, symbols, functions, and variables.
#include "LambLib.h"

/*****
    This function is like the main() of the .dll

    It takes the initial paramaters and calls the appropriate functions to find
    the initial roots and then calls the routines need to evaluate the symmetric and
    antisymmetric fundamental and higher-order modes. This data is all printed to the
    temp_curve.dat file. Modes are delimited by '-100'. Also, this file contains the
    f-k, Vph-f, and Vgr-f data.
*****/
void GetDispersion(double cL, double cT, double thickness, double tolerance, double
    max_freq) {

    double freq1 = .1;           //Set initial frequency to a low enough value
    double Vph1 = cL + 0.5*cT;   //Set initial Vph for sweep line to
                                //appropriate value
    double freq2 = max_freq*thickness; //Set end frequency for sweep line to
                                //maximum freq

```



```

double Vph2 = cL + cT;           //Set end Vph for sweep line to appropriate
                                //value
double half_thick = .5 /*thickness/2.0*/; //Evaluate half thickness

int fundamental, n;              //for higher-order fundamental = 0,
                                //otherwise fundamental = 1

double freq_step_size = .003;    //Set small frequency step size
double k_step_size = .02;        //likewise for wavenumber
double PI = 3.1415927;

vector<doubles> initial_sym_mins; //vector of 'double' vectors for
                                //symmetric mode minimums
vector<doubles> initial_asym_mins; //holds antisymmetric minimums
vector<doubles> fun_sym_mins;
vector<doubles> fun_asym_mins;
vector<doubles> sym_curve;        //Temporary placeholder for
                                //symmetric mode curves
vector<doubles> asym_curve;      //Temporary placeholder for
                                //antisymmetric curves

vector<doubles> fun_sym_curve;
vector<doubles> fun_asym_curve;

FILE* out;                       //output file pointer
char out_fname[100];

sprintf(out_fname,"temp_curve.dat"); //Set output file to: temp_curve.dat
out = fopen(out_fname, "wb");        //Open file

if(!out) {
    printf("Could not open file for writing\n");
    exit(-1);
}

printf("Finding Fundamental Mode Root (Antissymmetric).....");

//Set to fundamental mode evaluation
fundamental = 1;
//Find antisymmetric initial minimums for the fundamental mode
fun_asym_mins = MinimumSweep(freq1, 0.8*cT, 3, 0.8*cT, freq_step_size,
                              cL, cT, half_thick, 1);

if (fun_asym_mins.size() == 0) {
    printf("Didn't find any root for A0\n");
} else {

```

```

printf("Done\n");

printf("Tracing Assymmetric Fundamental Mode.....");
//Trace mode in positive wavenumber direction from initial minimum
//point
fun_asym_curve = TraceMode(fun_asym_mins[0][0],
                           fun_asym_mins[0][1], k_step_size, freq_step_size,
                           cL, cT, half_thick, 1, tolerance, fundamental,
                           freq2);
//Trace mode in negative wavenumber direction from initial minimum
//point
fun_asym_curve = TraceModeBack(fun_asym_curve, k_step_size,
                               freq_step_size, cL, cT, half_thick, 1, tolerance);
printf("Done\n");
}

//Do the same for the symmetric mode
printf("Finding Fundamental Mode Root (Symmetric).....");
fun_sym_mins = MinimumSweep(freq1, 1.5*cT, 3, 1.5*cT, freq_step_size,
                             cL, cT, half_thick, 0);

if (fun_sym_mins.size() == 0) {
    printf("Didn't find any root for S0\n");
} else {

    printf("Done\n");

    printf("Tracing Symmetric Fundamental Mode.....");
    fun_sym_curve = TraceMode(fun_sym_mins[0][0], fun_sym_mins[0][1],
                              k_step_size, freq_step_size, cL, cT,
                              half_thick, 0, tolerance, fundamental, freq2);

    fun_sym_curve = TraceModeBack(fun_sym_curve, k_step_size,
                                   freq_step_size, cL, cT, half_thick, 0, tolerance);
    printf("Done\n");
}

//Now move on to the higher-order modes for both symmetric and antisymmetric
fundamental = 0;
printf("Finding Higher Order Mode Roots (Antisymmetric)...");
initial_asym_mins = MinimumSweep(freq1, Vph1, freq2, Vph2, freq_step_size,
                                   cL, cT, half_thick, 1);

if (initial_asym_mins.size() == 0) {
    printf("Didn't find any root for higher-order A modes\n");
} else {

```

```

        printf("Done\n");
    }

    printf("Finding Higher Order Mode Roots (Symmetric).....");
    initial_sym_mins = MinimumSweep(freq1, Vph1, freq2, Vph2, freq_step_size,
                                    cL, cT, half_thick, 0);

    if (initial_sym_mins.size() == 0) {
        printf("Didn't find any root for higher-order S modes\n");
    } else {
        printf("Done\n");
    }

    if (fun_sym_mins.size() == 0 || fun_asym_mins.size() == 0) {
        exit(-1);
    }
    //Print number of antisymmetric modes and symmetric modes on first line of file
    //Separated by a tab.
    fprintf(out, "%d\t%d\n", initial_asym_mins.size()+1, initial_sym_mins.size()+1);

    //On each line of out file print frequency and Vph separated by a tab
    for (int k = 0; k < fun_asym_curve.size(); k++) {
        fprintf(out, "%f\t%f\n", fun_asym_curve[k][0]/thickness,
                fun_asym_curve[k][1]);
    }

    //Print '-100' to separate curves from one another
    fprintf(out, "-100\n");

    //Evaluate and print group velocities
    group_velocity(fun_asym_curve, out, thickness);

    //Evaluate and print f vs. k data
    for(n = 0; n < fun_asym_curve.size(); n++) {

        fprintf(out, "%f\t%f\n", (fun_asym_curve[n][0] / thickness) /
                fun_asym_curve[n][1], fun_asym_curve[n][0]/thickness);
    }
    fprintf(out, "-100\n");

    //Now trace and print the higher-order mode antisymmetric curves
    for (int i = 0; i < initial_asym_mins.size(); i++) {

        printf("Tracing mode A%d.....", i);
        asym_curve = TraceMode(initial_asym_mins[i][0],

```

```

        initial_asym_mins[j][1], k_step_size,
        freq_step_size, cL, cT, half_thick, 1, tolerance,
        fundamental, freq2);

    asym_curve = TraceModeBack(asym_curve, k_step_size, freq_step_size,
                               cL, cT, half_thick, 1, tolerance);

    for(int j = 0; j < asym_curve.size(); j++) {
        fprintf(out, "%ft%f\n", asym_curve[j][0]/thickness,
               asym_curve[j][1]);
    }
    fprintf(out, "-100\n");
    group_velocity(asym_curve, out, thickness);
    for(n = 0; n < asym_curve.size(); n++) {
        fprintf(out, "%ft%f\n", (asym_curve[n][0]/thickness) /
               asym_curve[n][1], asym_curve[n][0]/thickness);
    }
    fprintf(out, "-100\n");

    printf("Done\n");
}

//Print the fundamental symmetric mode to output file
for (int l = 0; l < fun_sym_curve.size(); l++) {
    fprintf(out, "%ft%f\n", fun_sym_curve[l][0]/thickness,
           fun_sym_curve[l][1]);
}
fprintf(out, "-100\n");
group_velocity(fun_sym_curve, out, thickness);
for(n = 0; n < fun_sym_curve.size(); n++) {
    fprintf(out, "%ft%f\n", (fun_sym_curve[n][0]/thickness) /
           fun_sym_curve[n][1], fun_sym_curve[n][0]/thickness);
}
fprintf(out, "-100\n");

//Now trace and print the higher-order symmetric mode curves
for ( int m = 0; m < initial_sym_mins.size(); m++ ) {

    printf("Tracing mode S%d.....", m);
    sym_curve = TraceMode(initial_sym_mins[m][0],
                          initial_sym_mins[m][1], k_step_size,
                          freq_step_size, cL, cT, half_thick, 0, tolerance,
                          fundamental, freq2);

    sym_curve = TraceModeBack(sym_curve, k_step_size, freq_step_size,
                              cL, cT, half_thick, 0, tolerance);
}

```

```

        for(n = 0; n < sym_curve.size(); n++) {
            fprintf(out, "%ft%f\n", sym_curve[n][0]/thickness,
                sym_curve[n][1]);
        }
        fprintf(out, "-100\n");
        group_velocity(sym_curve, out, thickness);
        for(n = 0; n < sym_curve.size(); n++) {
            fprintf(out, "%ft%f\n", (sym_curve[n][0]/thickness) /
                sym_curve[n][1], sym_curve[n][0]/thickness);
        }
        fprintf(out, "-100\n");

        printf("Done\n");
    }

fclose(out);          //Close output file and return

}

/*****END DISPERSION.CPP*****/

/*****BEGIN EVALLAMBEQN.CPP*****/

//Include the standard Windows and C-Runtime header files here.
#include <windows.h>
#include <cmath>
#include <vector>
#include <iterator>

using namespace std;

//This DLL source code file exports functions and variables.
#define LAMBLIB __declspec(dllexport)

//Include the exported data structures, symbols, functions, and variables.
#include "LambLib.h"

/*****
    This function, given the type and (freq, Vph) point, calls the appropriate
    characteristic equation, and transforms the Vph-f domain into the f-k domain
    *****/
double EvalEqn_V(double freq, double Vph, double cL, double cT, double half_thick,
    int type) {

```

```

double soln; //Solution to Characteristic Equation at given freq, k
double PI = 3.1415927;
if (type == 0) {          //Call Symmetric Equation
    soln = SymEqn(freq, 2*PI*freq/Vph, cL, cT, half_thick);
} else {                  //Call Antisymmetric Eqn
    soln = AsymEqn(freq, 2*PI*freq/Vph, cL, cT, half_thick);
}
return soln;
}

/*****
This function, given the type and (k, f) point, calls the appropriate
characteristic equation.
*****/
double EvalEqn(double freq, double k, double cL, double cT, double half_thick,
               int type) {
    double soln; //Solution to Characteristic Equation at given freq, k
    if (type == 0) {          //Call symmetric equation
        soln = SymEqn(freq, k, cL, cT, half_thick);
    } else {                  //Call antisymmetric eqn
        soln = AsymEqn(freq, k, cL, cT, half_thick);
    }
    return soln;
}

```

```

/*****
    Given the appropriate values (freq, wavenumber, longitudinal velocity, transverse
    velocity, and half-thickness), this function evaluates the symmetric dispersion
    relationship at the given (k,f)-point.
    *****/
double SymEqn(double freq, double k, double cL, double cT, double half_thick) {

    double omega,      //Circular frequency = 2*pi*f
           xsi;        //Wavenumber = omega/v, where v is phase velocity

    fcomplex Soln1, Soln2, den_const; //Temporary solution variables

    fcomplex c_alpha, //Separability Constant (related to cL -- longitudinal
                    // bulk velocity)
           c_beta,   //Separability Constant (related to cT -- transverse
                    // bulk velocity)
           tan_alpha, //tan(alpha*half_thick)
           tan_beta;  //tan(beta*half_thick)

    double PI = 3.1415927;

    omega = 2*PI*freq;
    xsi = k;

    //evaluate alpha and beta for current f and k
    c_alpha = complex_sqrt( (omega/cL)*(omega/cL) - xsi*xsi );
    c_beta  = complex_sqrt( (omega/cT)*(omega/cT) - xsi*xsi );

    //(sin(beta*half_thick)/cos(beta*half_thick))/beta
    tan_beta = Cdiv( complex_sin( RCmul( half_thick, c_beta ) ),
                    complex_cos( RCmul( half_thick, c_beta ) ) );
    Soln1 = Cdiv( tan_beta, c_beta );

    //(beta^2 - xsi^2)^2
    den_const = Csub( Cmul(c_beta, c_beta), Complex(xsi*xsi,0) );
    den_const = Cmul(den_const,den_const);

    //4*xsi^2*alpha*tan(alpha*half_thick)/(beta^2-xsi^2)^2
    tan_alpha = Cdiv( complex_sin( RCmul( half_thick, c_alpha ) ),
                    complex_cos( RCmul( half_thick, c_alpha ) ) );
    Soln2 = Cdiv( RCmul( 4.0*xsi*xsi, Cmul( c_alpha, tan_alpha ) ),
                    den_const );

    // return val: abs{real[tan(beta*half_thick)/beta] +
    // real[4*xsi^2*alpha*tan(alpha*half_thick)/(beta^2-xsi^2)^2]}

```

```

return fabs(Soln1.r + Soln2.r);
}

/*****
Given the appropriate values (freq, wavenumber, longitudinal velocity, transverse
velocity, and half-thickness), this function evaluates the antisymmetric dispersion
relationship at the given (k,f)-point.
*****/
double AsymEqn(double freq, double k, double cL, double cT, double half_thick) {

    double omega,      //Circular frequency = 2*pi*f
           xsi;        //Wavenumber = omega/v, where v is phase velocity

    fcomplex Soln1, Soln2, den_const; //Temporary solution variables

    fcomplex c_alpha, //Separability Constant (related to cL -- longitudinal
                    // bulk velocity)
           c_beta,   //Separability Constant (related to cT -- transverse
                    // bulk velocity)
           tan_alpha, //tan(alpha*half_thick)
           tan_beta;  //tan(beta*half_thick)

    double PI = 3.1415927;

    omega = 2*PI*freq;
    xsi = k;

    //evaluate alpha and beta for current f and k
    c_alpha = complex_sqrt( (omega/cL)*(omega/cL) - xsi*xsi );
    c_beta  = complex_sqrt( (omega/cT)*(omega/cT) - xsi*xsi );

    //(sin(beta*half_thick)/cos(beta*half_thick))/beta
    tan_beta = Cdiv( complex_sin( RCmul( half_thick, c_beta ) ),
                    complex_cos( RCmul( half_thick, c_beta ) ) );
    Soln1 = Cmul( tan_beta, c_beta );

    //(beta^2 - xsi^2)^2
    den_const = Csub( Cmul(c_beta, c_beta), Complex(xsi*xsi,0) );
    den_const = Cmul(den_const,den_const);

    //4*xsi^2*alpha*tan(alpha*half_thick)/(beta^2-xsi^2)^2
    tan_alpha = Cdiv( complex_sin( RCmul( half_thick, c_alpha ) ),
                    complex_cos( RCmul( half_thick, c_alpha ) ) );
    Soln2 = Cdiv( Cmul( den_const, tan_alpha ),
                    RCmul( 4.0*xsi*xsi, c_alpha ) );

```



```

// return val: abs{real[tan(beta*half_thick)*beta] +
// real[tan(beta*half_thick)*(beta^2-xsi^2)^2/(4*xsi^2*alpha)]}
return fabs(Soln1.r + Soln2.r);

}

/*****END EVALLAMBEQN.CPP*****/

/*****BEGIN TRACE.CPP*****/

//Include the standard Windows and C-Runtime header files here.
#include <windows.h>
#include <cmath>
#include <vector>
#include <iterator>
#include <cstdio>
#include "golden.h"

using namespace std;

//This DLL source code file exports functions and variables.
#define LAMBLIB __declspec(dllexport)

//Include the exported data structures, symbols, functions, and variables.
#include "LambLib.h"

/*****
This function takes a guess for the location of the mode's minimum (f, k) as
input. Then, using routines from Numerical Recipes for C, brackets and searches
for the minimum given the initial guess. At this point, error checking is done
to make sure that the solution is not jumping modes, that two modes crossing
doesn't mess things up, and that it is finding the right minimum. The solution to
the mode equation at the given k is returned (f, Vph).
*****/
vector<double> GetRoot_fk(double freq, //Estimated frequency to begin searching
                        double k, //Fixed wavenumber to search at
                        double cL, //Longitudinal velocity
                        double cT, //Transverse velocity
                        double f_step_size, //frequency step size
                        double k_step_size, //wavenumber step size
                        double tolerance, //tolerance for minimum solution
                        double half_thick, //half thickness of material
                        double type, //== 0 for asymmetric, == 1 for symmetric
                        double percentage) //error tolerance from initial guess
{

```

```

//Pointer to the characteristic equation solver
double (*char_eqn)(double, double, double, double, double, int);

//Used to store return value. root[0] = freq, root[1] = Vph
vector<double> root(2);

double PI = 3.1415927;
double f1 = freq, //Initialize bracket point 1 to freq guess
      f2 = freq+f_step_size, //bracket point 2 is one f step further
      f3 = 0, //bracket point 3
      min_f, //Solution (or minimum) to characteristic eqn
      golden_out = 0, //Value of characteristic eqn at min_f
      k_orig = k,
      temp_f_step; //Have to adjust f_step, so used to temporarily store
                  //original value to reset if needed.

double f_f1 = 0, //Value of character eqn. at f1
      f_f2 = 0, //Value of character eqn. at f2
      f_f3 = 0, //Value of character eqn. at f3
      check_f; //Used to store initial min_f, while we check
              //to see if there is a "better" minimum.

char_eqn = EvalEqn; //Set function pointer to EvalEqn()

////////////////////////////////////
//DEBUG CODE -- Used to stop program to allow step through
//if(k > 6.03) {
//    PI =3.1415927;
//}
////////////////////////////////////

int found = 0; // "Boolean" variable used as condition for when sol'n
              // is found.
int reduced = 0; //First time through solution error checking, used as
                // a boolean condition to reduce the f_step_size. Once
                // step size is reduced, this is set to = 1.

temp_f_step = f_step_size; //Keep "global" track of f_step_size so it can be
                           // reset to original value if needed.

//Loop until acceptable minimum is found
while (!found) {

//Using initial guess, bracket minimum and then search for it to within tolerance
//Holds k constant and searches in f-direction for minimum

```

```

mnbrak(&f1, &f2, &f3, &f_f1, &f_f2, &f_f3, k, cL, cT, half_thick, type,
      char_eqn);
golden_out = golden_search(f1, f2, f3, k, cL, cT, half_thick, type,
                          char_eqn, tolerance, &min_f);

```

```

////////////////////////////////////
//DEBUG
//printf("k: %f, freq: %f, min_f: %f, delta_f: %f\n",k, freq, min_f,
//      fabs(min_f-freq)/freq);
//printf("k: %f, freq: %f, min_f: %f, delta_Vph: %f\n", k, freq, min_f,
//      fabs((2*PI*min_f/k) - (2*PI*freq/k))/(2*PI*freq));
////////////////////////////////////

```

```

/*****
Begin Minimum Error Checking
*****/

```

```

//If first minimum isn't good, then don't need
//to waste the search time for a close nearby minimum

```

```

if (fabs(min_f - freq)/freq < percentage) {

```

```

    check_f = min_f;      //Temporary storage to keep track of
                          // current minimum while we check it is
                          //the right one

```

```

//Sometimes the guess point (k, f) is on the wrong side of the maximum hill and the
//minimum that is returned is not the correct minimum. This is seen for symmetric
//modes, and occurs on the side of the maximum away from the origin. It often occurs
//when the minimum is just a small feature on the downhill side of a larger feature. By
//walking uphill towards the origin and then bracketing the next minimum, we can see if
//this has happened. If a second minimum is found closer to the origin, then this is the
//minimum that we want.

```

```

    f1 = min_f;
    //start at current minimum
    f2 = min_f - f_step_size;
    //go 1 step closer to origin
    f_f1 = EvalEqn(f1, k, cL, cT, half_thick, type);
    f_f2 = EvalEqn(f2, k, cL, cT, half_thick, type);

```

```

    //Walk uphill until maximum is reached
    while (f_f2 > f_f1) {

```

```

        f1 = f2;
        f_f1 = f_f2;

```

```

    f2 = f2 - f_step_size;

    if (fabs(f2 - freq)/freq > (percentage + .01)) {

        f2 = -1;

    } else {

        f_f2 = EvalEqn(f2, k, cL, cT, half_thick, type);

    }

} //end while -- uphill walk to maximum

if (f2 > 0) {

//Now that we are out of the current valley, find the previous minimum to min_f by
//bracketing and then using golden search
    mnbrak(&f1, &f2, &f3, &f_f1, &f_f2, &f_f3, k, cL, cT,
        half_thick, type, char_eqn);
    golden_out = golden_search(f1, f2, f3, k, cL, cT,
        half_thick, type, char_eqn,
        tolerance, &min_f);

//If better minimum is not found, put check_f back into min_f.
//Otherwise discard and proceed with better minimum

    if (fabs(min_f - check_f)/check_f > .002) {

        min_f = check_f;

    } //end if -- better minimum check

}

}

if (fabs(min_f - freq)/freq < percentage) { //Check to make sure
// solution is not too far from
// guess freq. More than 2%
// error seems to be a good
// condition.

    root[0] = min_f;
    root[1] = 2*PI*min_f/k;
}

```

```

found = 1;    //acceptable root was found, no need to do more
              //robust/time-consuming search

} else {      //If solution is not good enough, then need to try
              //other methods to get accurate solution.

if (reduced == 0 || reduced == 1) {
              //if step size hasn't been reduced

    if (reduced == 0) {
        f_step_size = f_step_size/1000;
        //reduce by a factor of 100
        reduced = 1;
    } else {
        f_step_size = f_step_size*10;
        reduced = 2;
    }
    f1 = freq;           //reset f1, f2, f3, and min_f
    f2 = freq + f_step_size;
    f3 = 0;
    min_f = 0;

    //reduced = 1;      //set reduced = 1 to prevent from
                        //reducing further

} else {      //if f_step_size has already been reduced

    /***/
    Check for a small local minimum
    /***/
    f1 = freq;
    f2 = freq + f_step_size;
    f_f1 = EvalEqn(f1, k, cL, cT, half_thick, type);
    f_f2 = EvalEqn(f2, k, cL, cT, half_thick, type);

    if (f_f2 > f_f1) { //Downhill towards origin

        //Walk towards origin until minimum
        while (f_f1 < f_f2 && fabs(f1 - freq)/freq <
              (percentage +.01)) {

            f2 = f1;
            f_f2 = f_f1;

```

```

        f1 = f1 - f_step_size;
        f_f1 = EvalEqn(f1, k, cL, cT, half_thick,
                      type);

    } //end while -- towards origin walk

    f1 = f2;      //Set f1 to the minimum for storage
                 // when exiting if/else statement.

} else {

    //Walk away from origin until minimum
    while (f_f2 < f_f1 && fabs(f1 - freq)/freq <
          (percentage +.01)) {

        f1 = f2;
        f_f1 = f_f2;
        f2 = f2 + f_step_size;
        f_f2 = EvalEqn(f2, k, cL, cT, half_thick,
                      type);

    } //end while -- away from origin walk

/No need to switch, because minimum is already stored in f1.

} //end if-else for small minimum

/*****
End small local minimum check
*****/

    if (fabs(f1 - freq)/freq > percentage) { //Check to see if
// small local minimum check solved the incorrect minimum problem

        //Need to walk uphill in brute force manner.
        f1 = freq;
        f2 = freq + f_step_size;
        f3 = freq - f_step_size;
        f_f1 = EvalEqn(f1, k, cL, cT, half_thick, type);
        f_f2 = EvalEqn(f2, k, cL, cT, half_thick, type);
        f_f3 = EvalEqn(f3, k, cL, cT, half_thick, type);

        if(f_f1 > f_f2) { //uphill is towards origin

```

```

while (f_f1 > f_f2 && fabs(f1 - freq)/freq <
      (percentage +.01)) {

    f2 = f1;
    f_f2 = f_f1;
    f1 = f1 - f_step_size;
    f_f1 = EvalEqn(f1, k, cL, cT,
                  half_thick, type);

}

while (f_f1 < f_f2 && fabs(f1 - freq)/freq <
      (percentage +.01)) {

    f2 = f1;
    f_f2 = f_f1;
    f1 = f1 - f_step_size;
    f_f1 = EvalEqn(f1, k, cL, cT,
                  half_thick, type);

}

} else { //uphill is away from origin

while (f_f1 < f_f2 && fabs(f2 - freq)/freq <
      (percentage +.01)) {

    f1 = f2;
    f_f1 = f_f2;
    f2 = f2 + f_step_size;
    f_f2 = EvalEqn(f2, k, cL, cT,
                  half_thick, type);

}

while (f_f1 > f_f2 && fabs(f2 - freq)/freq <
      (percentage +.01)) {

    f1 = f2;
    f_f1 = f_f2;
    f2 = f2 + f_step_size;
    f_f2 = EvalEqn(f2, k, cL, cT,
                  half_thick, type);

}

```

```

    }
}

//Check to see if brute force method found an acceptable
//minimum
if (fabs(f1 - freq)/freq < percentage) {

    root[0] = f1;
    root[1] = 2*PI*f1/k;

    found = 1;

} else { //try adjusting k slightly and retrying to locate
        //an acceptable minimum

    k = k + k_step_size/2.0;
    found = 0;
    reduced = 0;
    f_step_size = temp_f_step;

    if( fabs((k_orig - k)/k_step_size) > 10 ) {

        exit(-1);

    }

}

} //end if-else -- after f_step_size has been reduced

} //end if-else -- for more intensive/brute force error checking methods

} // end while(!found)

return root; //return the root that was found

}

```



```

/*****
    This function takes the initial guess for the root of the characteristic equation.
    It does not do any error-checking on its result because the initial minimum and the
    output of this function are needed for the inputs to trace the rest of the curve.
    There is not enough information to do any acceptability checks yet.
*****/
vector<double> GetFirstRoot(double freq, //Estimated frequency to begin searching
                           double k,   //Fixed wavenumber to search at
                           double cL,   //Longitudinal velocity
                           double cT,   //Transverse velocity
                           double f_step_size, //frequency step size
                           double tolerance, //tolerance for minimum solution
                           double half_thick, //half thickness of material
                           double type) //== 0 for asymmetric, == 1 for symmetric
{
    double (*char_eqn)(double, double, double, double, double, int);

    vector<double> root(2);

    double PI = 3.1415927;

    double f1 = freq, //initial frequency for bracketing
           f2 = freq+f_step_size, //second frequency for bracketing
           f3 = 0, //third frequency for bracketing
           min_f, //Used to store frequency of function minimum
           golden_out = 0;

    double f_f1 = 0, //value of function at f1
           f_f2 = 0, //value of function at f2
           f_f3 = 0; //value of function at f3

    char_eqn = EvalEqn; //set to correct characteristic equation

    //Bracket and locate minimum by holding k constant and walking in f-direction

    mnbrak(&f1, &f2, &f3, &f_f1, &f_f2, &f_f3, k, cL, cT, half_thick, type,
           char_eqn);
    golden_out = golden_search(f1, f2, f3, k, cL, cT, half_thick, type, char_eqn,
                              tolerance, &min_f);

    root[0] = min_f;
    root[1] = 2*PI*min_f/k;

    ////////////Debug////////////////////////////////////
    //printf("First root: %f, %fn", root[0], root[1]);

```

```

////////////////////////////////////

return root;

}

/*****
This function is used to search for minimums in the v-f domain
instead of in the f-k domain. It is similar to GetRoot_fk in all
other respects. It is mainly used in the initial minimum sweep
*****/
vector<double> GetRoot_vf(double freq,
                        double Vph,
                        double cL,
                        double cT,
                        double step_size,
                        double tolerance,
                        double half_thick,
                        double type)
{

    double (*char_eqn)(double, double, double, double, double, int);

    vector<double> root(2);

    double PI = 3.1415927;

    double f1 = freq,
           f2 = freq+step_size/1000,
           f3 = 0,
           min_f,
           golden_out = 0;

    double f_f1 = 0,
           f_f2 = 0,
           f_f3 = 0;

    char_eqn = EvalEqn_V;

    mnbrak(&f1, &f2, &f3, &f_f1, &f_f2, &f_f3, Vph, cL, cT, half_thick, type,
           char_eqn);
    golden_out = golden_search(f1, f2, f3, Vph, cL, cT, half_thick, type, char_eqn,
                              tolerance, &min_f);

    root[0] = min_f;
}

```

```

root[1] = Vph;

return root;

}

/*****
This function takes two points as input [(f1,Vph1),(f2,Vph2)]
and returns a new point [(f_new,Vph_new)] based on a linear
extrapolation in the f-k domain. It converts from Vph-f domain
into f-k domain, performs extrapolation, and then converts back
to Vph-f domain.
*****/
vector<double> LinearExtrapolateRoot(vector<double> point1,
                                   vector<double> point2,
                                   double k_step_size)
{
    double slope,           //slope of line
           intercept,     //y-intercept (f value)
           f1,            //frequency point 1
           f2,            //frequency point 2
           new_f,         //frequency guess
           k1,            //wavenumber point 1
           k2,            //wavenumber point 2
           new_k;         //wavenumber guess

    double PI = 3.1415927;

    vector<double> new_point(2); //vector used to store (f,v) guess

    f1 = point1[0];
    f2 = point2[0];
    k1 = 2*PI*f1/point1[1];
    k2 = 2*PI*f2/point2[1];

    slope = (f2 - f1)/(k2 - k1); //calculate slope
    intercept = f2 - slope*k2; //calculate intercept

    new_k = k2 + k_step_size; //find new_k with step_size
    new_f = slope*new_k + intercept; //y = mx + b

    new_point[0] = new_f;
    new_point[1] = 2*PI*new_f/new_k;

    return new_point;
}

```

```

}

/*****
    This function takes 4 points as input [(f1,Vph1),(f2,Vph2), etc.]
    and returns a new point [(f_new,Vph_new)] based on a quadratic
    extrapolation in the f-k domain. It converts from Vph-f domain
    into f-k domain, performs extrapolation, and then converts back
    to Vph-f domain.
*****/
vector<double> QuadExtrapolateRoot(vector<double> point1,
                                  vector<double> point2,
                                  vector<double> point3,
                                  vector<double> point4,
                                  double k_step_size)
{
    double new_f,
          new_k;

    double PI = 3.1415927;

    vector<double> new_point(2);

    new_k = 2*PI*point4[0]/point4[1] + k_step_size;
           //Quadratic Extrapolation steps
    new_f = point1[0] - 3*point2[0] + 3*point3[0];

    new_point[0] = new_f;
    new_point[1] = 2*PI*new_f/new_k;

    return new_point;
}

/*****
    This function does the same thing as QuadExtrapolateRoot, but in the
    negative k direction. This allows us to trace the modes back to the
    origin in f-k space.
*****/
vector<doubles> TraceModeBack(vector<doubles> curve,
                              double k_step_size,
                              double f_step_size,
                              double cL,
                              double cT,
                              double half_thick,
                              int type,

```

```

double tolerance)
{
    double PI = 3.1415927;

    vector<double> mode_point(2);
    vector<doubles>::iterator p;

    double k_new = (2*PI*curve[0][0])/curve[0][1];

    while (k_new > .2) {

        p = curve.begin();
        mode_point = QuadExtrapolateRoot(curve[5], curve[3], curve[1],
                                         curve[0], -k_step_size);
                                         //Note -k_step_size
        k_new = (2*PI*mode_point[0])/mode_point[1]/* - k_step_size*/;
        mode_point = GetRoot_fk(mode_point[0], k_new, cL, cT, f_step_size, -
                                k_step_size, tolerance, half_thick, type, .02);

        curve.insert(p, 1, mode_point);
    }

    return curve;
}

/*****
This function takes the original minimum for a mode curve from the original
sweep. It then finds a second root by stepping once in the k-direction and
following the slope of the characteristic equation. Using the first 2 points, it
linearly extrapolates the third. It repeats this same step using the 2nd and 3rd
points to get the fourth and so on until 7 initial points are calculated. Then using a
quadratic extrapolation routine, the mode is traced and returned.
*****/
vector<doubles> TraceMode(double freq1,      //frequency
                        double Vph,       //phase velocity
                        double k_step_size, //wavenumber step size
                        double f_step_size, //frequency step size
                        double cL,        //longitudinal bulk velocity
                        double cT,        //transverse bulk velocity
                        double half_thick, //half thickness
                        int type,         //mode type: 0 - symmetric,
                                         //1 - antisymmetric
                        double tolerance, //tolerance for root
                        int fundamental,  //Is it a fundamental mode, 0 = no,
                                         //1 = yes

```

```

double max_freq) //max frequency desired; stop point.
{
double PI = 3.1415927;
double percentage; //value used for error checking

vector<doubles> mode_curve;
vector<double> mode_point(2);

mode_point[0] = freq1;
mode_point[1] = Vph;

mode_curve.push_back(mode_point); //enter sweep minimum into mode curve

double freq2 = freq1 + f_step_size, //step once in frequency direction
freq3 = 0, //used to store third frequency
k_new, //needed for min. bracketing
golden_out = 0, //new wavenumber value
f_f1 = 0, //value of function at f1
f_f2 = 0, //value of function at f2
f_f3 = 0, //value of function at f3
size; //total number of points in mode curve

//Get second point in mode by taking 1 small step in k and
//searching in the freq direction for minimum
k_new = (2*PI*freq1)/Vph + k_step_size;

//If fundamental mode, need to use GetFirstRoot in order to get accurate enough
//seeds for the mode tracing routines
if (fundamental) {

mode_point = GetFirstRoot(freq1, k_new, cL, cT, f_step_size, tolerance,
half_thick, type);

} else { //Higher order modes can use GetRoot_fk.

mode_point = GetRoot_fk(freq1, k_new, cL, cT, f_step_size, k_step_size,
tolerance, half_thick, type, .04);

}

mode_curve.push_back(mode_point); //Store new point in mode curve.

//Get third point in mode by linearly extrapolating from the
//first two points to get a guess for the third point. Then, using

```

```

//that guess, iterate to a minimum near that guess.
mode_point = LinearExtrapolateRoot(mode_curve[0], mode_curve[1],
                                   k_step_size);
k_new = (2*PI*mode_point[0])/mode_point[1] /*+ k_step_size*/;
mode_point = GetRoot_fk(mode_point[0], k_new, cL, cT, f_step_size,
                        k_step_size, tolerance, half_thick, type, .04);

mode_curve.push_back(mode_point);

//Get fourth point in mode by linearly extrapolating form the first
//and third points to get a guess for the fourth point. Then, using
//that guess, iterate to a minimum near that guess.
mode_point = LinearExtrapolateRoot(mode_curve[0], mode_curve[2],
                                   k_step_size);
k_new = (2*PI*mode_point[0])/mode_point[1] /*+ k_step_size*/;
mode_point = GetRoot_fk(mode_point[0], k_new, cL, cT, f_step_size,
                        k_step_size, tolerance, half_thick, type, .04);
mode_curve.push_back(mode_point);

//Get fifth point by using quadratic extrapolation with the first 4
mode_point = QuadExtrapolateRoot(mode_curve[0], mode_curve[1],
                                 mode_curve[2], mode_curve[3],
                                 k_step_size);
k_new = (2*PI*mode_point[0])/mode_point[1] /*+ k_step_size*/;
mode_point = GetRoot_fk(mode_point[0], k_new, cL, cT, f_step_size,
                        k_step_size, tolerance, half_thick, type, .04);
mode_curve.push_back(mode_point);

//Get sixth point in the same way
mode_point = QuadExtrapolateRoot(mode_curve[1], mode_curve[2],
                                 mode_curve[3], mode_curve[4],
                                 k_step_size);
k_new = (2*PI*mode_point[0])/mode_point[1] /*+ k_step_size*/;
mode_point = GetRoot_fk(mode_point[0], k_new, cL, cT, f_step_size,
                        k_step_size, tolerance, half_thick, type, .04);
mode_curve.push_back(mode_point);

//Get seventh point in the same way
mode_point = QuadExtrapolateRoot(mode_curve[2], mode_curve[3],
                                 mode_curve[4], mode_curve[5],
                                 k_step_size);
k_new = (2*PI*mode_point[0])/mode_point[1] /*+ k_step_size*/;
mode_point = GetRoot_fk(mode_point[0], k_new, cL, cT, f_step_size,
                        k_step_size, tolerance, half_thick, type, .04);
mode_curve.push_back(mode_point);

```

```

//We can now use every other point to allow our quadratic extrapolation
//to be numerically more stable. As it approaches and crosses other mode curves
//it has less tendency to jump between them.
while(mode_point[0] < max_freq) {

    size = mode_curve.size();
    mode_point = QuadExtrapolateRoot(mode_curve[size-6],
                                    mode_curve[size-4],
                                    mode_curve[size-2],
                                    mode_curve[size-1],
                                    k_step_size);
    k_new = (2*PI*mode_point[0])/mode_point[1];

    //Change percentage acceptance value based on the distance between the
    // previous 2 roots. But, don't let it be larger than .02 or 2% difference
    percentage = (4*fabs(mode_curve[size-1][0] - mode_curve[size-2][0])) /
                mode_curve[size-1][0];
    percentage = (percentage < .02) ? (percentage) : (.02);

    mode_point = GetRoot_fk(mode_point[0], k_new, cL, cT, f_step_size,
                            k_step_size, tolerance, half_thick, type,
                            percentage);
    mode_curve.push_back(mode_point);
}

return mode_curve;
}

/*****END TRACE.CPP*****/

/*****BEGIN MINIMUMSWEEP.CPP*****/

//Include the standard Windows and C-Runtime header files here.
#include <windows.h>
#include <cmath>
#include <vector>
#include <iterator>
#include <cstdio>

using namespace std;

//This DLL source code file exports functions and variables.
#define LAMBLIB __declspec(dllexport)

```



```
//Include the exported data structures, symbols, functions, and variables.
#include "LambLib.h"
```

```

/*****
Function takes an initial (f,v) point and a final (f,v) point and evaluates
the characteristic function along this line at intervals determined by
the frequency step size. Returns a vector of length, floor((freq2-freq1)/step_size,
with the values of the characteristic function stored in it. Rows = # of points,
Col1 = freq, Col2 = Vph, Col3 = eqn value @ (freq,Vph)
*****/
vector<doubles> EvalSweep( double freq1,          //Start point (freq1, Vph1) for
                        double Vph1,          //minimum sweep line
                        double freq2,        //End point (freq2, Vph2) for
                        double Vph2,        //minimum sweep line
                        double freq_step,    //Frequency step size
                        double cL,          //Longitudinal bulk velocity
                        double cT,          //Transverse bulk velocity
                        double half_thick,  //half thickness of material
                        int type)          // = 0 if symmetric; = 1 if asymmetric

{

    double sweep_flength,          //Variable for sweep line length
           sweep_slope,            //Slope of sweep line
           sweep_intercept,        //y-intercept of sweep line
           freq,                   //frequency at which function is being evaluated
           Vph;                   //velocity at which function is being evaluated

    vector<doubles> eval_vector;
    vector<double> temp_vector(3); //temporary vector to store column entries

    int i,                          //iteration counter
        num_steps;                  //Given sweep_flength and freq_step, # of
                                    //steps to be taken along sweep line

    double PI = 3.1415927;

    sweep_flength = freq2 - freq1;
    sweep_slope = (Vph2 - Vph1)/(freq2 - freq1); //Will be used to find the
    sweep_intercept = Vph1 - sweep_slope*freq1; //point along the sweep line
                                                //at the current frequency

```

```

freq_step = freq_step/3;

num_steps = (int) floor(sweep_length/freq_step);

//iterate through sweep line at fixed frequency intervals, and
//evaluate the characteristic function. Store into min_array[][]
for (i = 0; i < num_steps; i++) {

    freq = freq1 + i*freq_step;
    Vph = sweep_slope*freq + sweep_intercept;

    temp_vector[0] = freq;
    //printf("%f\n", temp_vector[0]);
    temp_vector[1] = Vph;
    //printf("%f\n", temp_vector[1]);
    temp_vector[2] = EvalEqn(freq, 2.0*PI*freq/Vph, cL, cT, half_thick,
                             type);
    //printf("%f\n", temp_vector[2]);

    eval_vector.push_back(temp_vector);
}

//return vector of function values along requested line
return eval_vector;

}

/*****
This function takes the evaluation vector of the sweep line
and extracts the minimums. Because locating each minimum is
so important at this stage, no bisection is used and a brute force
minimum find is performed. The function returns a vector of the
minimum points, with the value of the characteristic eqn at those
points. It is in the same format as the input vector.
*****/
vector<doubles> FindSweepMins(vector<doubles> eval_vector)
{
    int start_location = 0,          //Marks the initial location when
                                     // entering the while loops
    curr_location = 0,              //Keeps track of the current location
                                     // of the search iterator
    size = eval_vector.size();      //Size of the eval vector

    vector<doubles> min_vector;      //Vector to store minimum locations in.
                                     // In same format as eval_vector
}

```

```

vector<double> temp_vector(3);           //Temporary vector used for temp
of mins.                               // storage

//Continue to search for minimums until curr_location is at end of vector.
//Remember that size of vector is one off the last index number (off by one)
//We need to access the next location, so to prevent an "overrun" error, we
//need to make sure that we are "2" away from eval_vector.size()
while (curr_location + 2 < size) {

    start_location = curr_location;

    //Iterate until curr location is at a local minimum or end of vector
    while( (eval_vector[curr_location][2] > eval_vector[curr_location + 1][2])
        && curr_location + 2 < size ) {

        curr_location++;

    } //end while

    //If the current location is the same as the start location, then the
    //minimum was found at the beginning of the search and we need to iterate
    //until we get a maximum, and then iterate again until a min is found.
    if (curr_location == start_location) {

        //Iterate to maximum or end of vector
        while( (eval_vector[curr_location][2] < eval_vector[curr_location
            + 1][2] ) && curr_location + 2 < size) {

            curr_location++;

        } //end while

        //Now iterate to minimum or end of vector
        while( (eval_vector[curr_location][2] > eval_vector[curr_location
            + 1][2] ) && curr_location + 2 < size) {

            curr_location++;

        } //end while
    } //end if(start == curr)

    //If the end of the vector was not reached, at this point we have

```

```

//found a legitimate minimum and need to store it into min vector.
if (curr_location + 2 != size) {

    temp_vector[0] = eval_vector[curr_location][0];
    temp_vector[1] = eval_vector[curr_location][1];
    temp_vector[2] = eval_vector[curr_location][2];

    min_vector.push_back(temp_vector);

} else { //If end of vector is reached, return

    return min_vector;
    //have to test if min_vector is empty before using
} //end if-else

} //end main while loop

}

/*****
This function takes a vector of minimums and searches
in the phase velocity-frequency domain for the root
closest to that minimum by holding the Vph constant and
varying the frequency. It returns a vector of roots to be
the seeds for the curve tracing routines.
*****/
vector<doubles> FindMinRoots(vector<doubles> min_vector, //vector of minimums

    double cL, //longitudinal bulk vel.
    double cT, //transverse bulk vel
    double step_size, //freq step size
    double tolerance, //solution tolerance level
    double half_thick, //half thickness
    int type) //characteristic eqn
    //symmetric == 0, antisym == 1

{

    double freq,
        Vph;
    vector<double> root; //Root Locaton (freq, Vph)

    //Find root for each minimum
    for (int i = 0; i < min_vector.size(); i++) {

        freq = min_vector[i][0];
        Vph = min_vector[i][1];

```

```

////////////////////////////////////
//Debug Code
//if (Vph > 5.57) {
//    freq=freq;
//}
////////////////////////////////////

//Function returns root after searching in Vph-freq domain
root = GetRoot_vf(freq, Vph, cL, cT, step_size, tolerance, half_thick,
                 type);

//store root in minimum vector replacing previous estimation of root
min_vector[i][0] = root[0];
min_vector[i][1] = root[1];
}

return min_vector;    //return vector of initial roots for modes
}

/*****
This function takes the initial material parameters and
sweep line coordinates (freq1,Vph1) to (freq2, Vph2), and updates
min_array with the minimum points stored in it (min_array[i][0] = freq,
min_array[i][1] = Vph). It also returns the number of minimums in the
array.
*****/
vector<doubles> MinimumSweep(double freq1,      //Start point (freq1, Vph1) for
                           double Vph1,      //minimum sweep line
                           double freq2,      //End point (freq2, Vph2) for
                           double Vph2,      //minimum sweep line
                           double freq_step,  //Frequency step size
                           double cL,        //Longitudinal bulk velocity
                           double cT,        //Transverse bulk velocity
                           double half_thick, //half thickness of material
                           double type)     // = 0 if symmetric; = 1 if asymmetric
{

    vector<doubles> eval_vector;    //Stores function values along sweep line
    vector<doubles> min_vector;    //Stores the minimums along the sweep line

    //Evaluate the function along the line (freq1, Vph1) to (freq2, Vph2) at the
    //freq_step interval

    eval_vector = EvalSweep(freq1, Vph1, freq2, Vph2, freq_step,

```

```

cL, cT, half_thick, type);

if (eval_vector.size() == 0) {
    exit(-1);
}

//Find the minimums of the function evaluation done with EvalSweep()
min_vector = FindSweepMins(eval_vector);

//////////
//DEBUG CODE
//
//for (int j = 0; j < min_vector.size(); j++) {
//    printf("Sweep Minimum[%d]: %f %f\n", j, min_vector[j][0],
//          min_vector[j][1]);
//}
//////////

//Using the minimum points as a start, find the roots corresponding to those
//minimums
min_vector = FindMinRoots(min_vector, cL, cT, freq_step, .0000001, half_thick,
                          type);

//////////
//DEBUG Code
//
//for (int i = 0; i < min_vector.size(); i++) {
//    printf("Root Minimum[%d]: %f %f\n", i, min_vector[i][0],
//          min_vector[i][1]);
//}
//////////

return min_vector;                                     //return roots
}

/*****END MINIMUMSWEEP.CPP*****/

/*****BEGIN GROUP.CPP*****/

#include <cmath>
#include <cstdio>
#include <vector>
#include <iterator>

```

```

using namespace std;

//This DLL source code file exports functions and variables.
#define LAMBLIB __declspec(dllexport)

//Include the exported data structures, symbols, functions, and variables.
#include "LambLib.h"

/*****
Function takes a vector of a mode's phase velocity data and computes
the group velocity. It then outputs this to the data file, group_out, that
is passed in as one of the parameters.
*****/
void group_velocity(vector<doubles> mode_data, FILE *group_out, double thickness) {

    double x_prev, vf_prev;           // i-3 frequency and phase velocity values
    double x_cur, vf_cur;             // i-2 frequency and phase velocity values
    double x_cur_next, vf_cur_next;   // i-1 frequency and phase velocity values
    double x_next, vf_next;          // i frequency and phase velocity values
    double dx, dvf;                  // delta frequency and delta phase velocity
    double vg, vg_last;              // current and last group velocities

    //Initialize the prior values before entering loop.
    //Prior values used to reduce numerical error in the derivative.
    x_prev = mode_data[0][0]/thickness;
    vf_prev = mode_data[0][1];
    x_cur = mode_data[1][0]/thickness;
    vf_cur = mode_data[1][1];
    x_cur_next = mode_data[2][0]/thickness;
    vf_cur_next = mode_data[2][1];

    //Loop through the remaining phase velocity data points
    for (int i = 3; i < mode_data.size(); i++) {

        //Initialize current variables
        x_next = mode_data[i][0]/thickness;
        vf_next = mode_data[i][1];

        //Find deltas
        dvf = vf_next - vf_prev;
        dx = x_next - x_prev;

        //Move and update variables for next iteration
        x_prev = x_cur;
        vf_prev = vf_cur;
        x_cur = x_cur_next;
    }
}

```

```

vf_cur = vf_cur_next;
x_cur_next = x_next;
vf_cur_next = vf_next;

/*
now we'll determine the group velocity from two points
this will be the quantity  $V_g/V_{t1}$  because  $y_r = V_{ph}/V_{t1}$ 
Multiple variables are used to try and eliminate numerical error in derivative
*/
    vg = vf_next / (1.0 - x_next * dvf / vf_next / dx);

    //Also used to eliminate numerical error; rarely used
    if(dvf == 0) {
        vg = vg_last;
    } //end if

    //Print group velocity to file
    fprintf(group_out, "%f\t%f\n", x_cur+x_cur_next/2.0, vg);

    vg_last = vg;

} /* end for */
//add delimiter to mark end of mode
fprintf(group_out, "-100\n");

return;

} //end group_velocity()

/*****END GROUP.CPP*****/

```


Appendix B

Matlab Code for Multi-mode Arrival Sorting

PEAK_ARRIVAL.M

```
clear
%freq_num = 10
%for filter_num = 3:4
tic
for freq_num = 1:10

%filterfile = ['filters\filter_4_99.mat'];
%filterfile = ['filters\filter_5' int2str(freq_num - 1) '_4band.mat'];
%load(filterfile);

freq_char = '900';

dir_name = ['plate15-2_9' int2str(freq_num - 1) '0kHz\'];

if (~exist(['jill\ dir_name'],'dir'))
    mkdir('jill\ dir_name');
    mkdir(['jill\ dir_name\'],'signals\');
end

fname = ['Plate15-2_freqwlk_9-.99MHz_1500_6500_6us_bin_freq' int2str(freq_num)];
fname_out = ['Plate15-2_9' int2str(freq_num-1) '0khz_100projs'];
read_dir = ['D:\data\jill\'];
save_dir = ['jill\ dir_name'];
fname_arrplot = ['Plate15-2_9' int2str(freq_num-1) '0khz_arrivals']; %
int2str(2050+(freq_num - 1)*10) 'khz_3us_wavelet'];

pipe = 0;
step_size = 2;

if pipe
    size = 180;
else
    size = 100*100;
end

start_point = 1500;
orig_signal_length = 6500;
wavelet_level = 8;
```

```

num_zero_levels = 6; %changed from 5

%adj_signal_length = ceil(signal_length/(2^wavelet_level));
%adj_signal_length = (2^wavelet_level)*adj_signal_length;

sample_rate = 25;
delay_line = 21.1; %15 for the .5" transducers

delay_filter = 0;
%delay_filter = ceil(1229.5);

total_delay = delay_filter/sample_rate + delay_line - start_point/sample_rate;

thresh = 100;
scale = 0; %in decibles

bool_envelope = 0;
bool_signal = 0;

reproducible = 0; %boolean value
rep_projection = 1; %projection number for reproducibility test

bool_front = 0;
fid = fopen([read_dir fname], 'r');
r = zeros(size,15);
temp = 1:size;
temp = temp';

for m = 1:size

    signal_length = orig_signal_length;

    m_out = [int2str(freq_num) ':' int2str(m)]

    signal = fread(fid, signal_length, 'int16');
    signal = signal - mean(signal);

    %signal = filter(Num, Den, signal);

    %Output the signal
    if (bool_signal == 1)
        sigfile = [save_dir 'signals\' fname_out '_waveform_' int2str(m)];
        %sigout = signal';
        save(sigfile, 'signal', '-ASCII')
    end
end

```

```

adj_signal_length = ceil(signal_length/(2^wavelet_level));
adj_signal_length = (2^wavelet_level)*adj_signal_length;

signal = abs(signal);
signal = signal*10^(scale/20);

%Create envelope with wavelet transform
signal(signal_length+1:adj_signal_length)=0;
[swa,swd] = swt(signal, wavelet_level, 'coif3');
swd(1:num_zero_levels,:) = 0;
env = iswt(swa,swd,'coif3');

%Read in envelope from multiple.c
%fenv = ['al4\clean_0.9MHz_1250_2500pts_bin_envelope' int2str(m)];
%fenv = [fenv '.dat'];
%fid2 = fopen(fenv, 'r');
%env = fscanf(fid2, '%i', 2048);
%env = env';
%fclose(fid2);

%Output the envelope
if (bool_envelope == 1)
    outfile=[save_dir 'signals\' fname_out '_envelope_' int2str(m)];
    envout = env';
    save(outfile, 'envout', '-ASCII')
end

x = diff(env);
yR = [0 x];
yL = [x 0];

big = (yL >= 0);
small = (yL < 0);
yL(big) = 5;
yL(small) = 1;

big = (yR >= 0);
small = (yR < 0);
yR(big) = 1;
yR(small) = -5;

peak_index = find(yR == yL);

count = 1;

```

```

clear arrival_index;
t = 1;
while t < length(peak_index)

    if (t == 1 & env(peak_index(t)) < thresh)
        while (env(peak_index(t)) < thresh & t < length(peak_index))
            t = t+1;
        end
    end

    if (t >= length(peak_index))
        arrival_index = find( (peak_index >= 100) & (env(peak_index) >= thresh));
    end

    %measured_arrival = env(peak_index(t),1)/sample_rate - total_delay;
    %est_dist = sqrt((step_size*(m-1))^2 + sample_length^2);
    %est_arrival = est_dist/est_vel;

    %if (abs(measured_arrival - est_arrival)/est_arrival > .4)
    %
    % t = 1;
    % scale = 20*log10(1000/max(env(:,2)));
    % env(:,2) = env(:,2)*10^(scale/20);
    %
    %end

else

    arrival_index(count) = t;
    count = count + 1;
    t = t + 1;
end
end

%arrival_index = find( (peak_index >= 100) & (env(peak_index) >= thresh));

for t = 1:length(arrival_index)
    p(t) = peak_index(arrival_index(t));

    %Begin Front Edge Locator (To use, set front=1 above)
    index = p(t) - 1;
    %while (env(index) > (1/exp(1))*env(p(t)+1))
    % index = index - 1;
    %end

```

```

if ( bool_front & env(index) < env(index+1) & env(index+1) < env(index+2) &
env(index+2) < env(index+3) & env(index+3) < env(index+4) )

```

```

    index_a = index;
    index_b = index+4;
    point_a = env(index_a);
    point_b = env(index_b);

```

```

    slope = 1.0 * (point_b - point_a) / (index_b - index_a);
    b = 1.0 * (point_a*index_b - point_b*index_a) / (index_b - index_a);

```

```

    interp_val = (point_a + point_b)/2;
    p(t) = floor((interp_val - b)/slope);

```

```

    if (p(t) < 0)
        p(t) = 0;
    end

```

```

end
%End Front Edge Location

```

```

end

```

```

%if (length(arrival_index) >= 2)
%
% first_env_indices(2) = p(1);
% index_env_values(2) = env(p(1));
% first_env_indices(3) = p(2);
% index_env_values(3) = env(p(2));
%
% first_env_indices(1) = first_env_indices(2) - 1;
% i = first_env_indices(1);
% while(env(i) > index_env_values(3))
%
%     i = i - 1;
% end
% first_env_indices(1) = i;
% index_env_values(1) = env(i);
%
% energy = 0;
% for i = first_env_indices(1):first_env_indices(3)
%     energy = energy + env(i);
% end
%
% center_index = 0;
% center_energy = 0;
% while (center_energy < (energy/2))
%     center_index = center_index + 1;

```

```

% center_energy = center_energy + env(center_index);
% end

%figure
%x1 = 1:length(env);
%x2(1:3) = first_env_indices(1:3);
%x3 = center_index;
%plot(x1,env,x2,index_env_values,x3,env(center_index),'+')
% r(m,1) = center_index/sample_rate;
% r(m,2) = first_env_indices(2)/sample_rate;
%end

for i = 1:length(arrival_index)
    r(m,i) = p(i)/sample_rate - total_delay;
    %r(m,i) = (p(i) + start_point)/sample_rate - total_delay;
    %r(m,i) = p(i);
end

end

fclose(fid);

y = sqrt((step_size*((1:size)-1)).^2 + 200^2);
for i = 1:8
    vel(:,i) = y./(r(:,i));
end

rtemp = [temp(:,1) r(:,1) r(:,2) r(:,3) r(:,4) r(:,5) r(:,6) r(:,7) r(:,8) r(:,9) r(:,10) r(:,11)
r(:,12) vel(:,1) vel(:,2) vel(:,3) vel(:,4) vel(:,5)];

outfile=[save_dir fname_out '_peaks.txt'];
fid3 = fopen(outfile, 'w');
fprintf(fid3, '%d %d %d %d %d %d %d %d %d %d %d %d %d\t\t\t%d %d %d %d
%d\n', rtemp);
fclose(fid3);

figure
plot(temp,r(:,1:8),'LineStyle','none','Marker','.');
%plot(temp,r(:,1:14),'LineStyle','none','Marker','.');
%ylim([40 140]);
print('-djpeg', [save_dir fname_arrplot]);
close

%figure
%plot(temp,vel,')

```

```

%ylim([1.5 5.5]);
%print('-djpeg', [save_dir fname_arrplot '_vel.jpg']);
%close

end
toc

```

SORT_ALGORITHM.M

```

function [mode_data] = sort_algorithm(mode_data, num_modes)

tic
%Mode data's format is: mode_data(mode #, freq #, waveform #)
%           j      k      i
num_freqwalk = 10;

data_size = size(mode_data(1,1,:),3)

%generate x values for scanning geometry
x_scan_generator;

%generate predicted curves
for j = 1:num_modes+1
    for k = 1:num_freqwalk
        pred_data(j,k,:) = polyval(polyfit(x(j,k,:), mode_data(j,k,:), 4),
reshape(x(j,k,:),1,data_size));
    end
end

%earmark potential flaw areas by points that deviate more than 1.5 std deviations
for k = 1:num_freqwalk
    diff(k,:) = abs(mode_data(1,k,:) - pred_data(1,k,:));
    dev_thresh(k) = 1.5*std(diff(k,:));
end

for j = 1:num_modes+1
    for k = 1:10
        size_tempx = 1;
        for i=1:data_size
            if (abs(mode_data(1,k,i) - pred_data(1,k,i)) < dev_thresh(k))
                temp_x(size_tempx)=x(1,1,i);
                temp_data(size_tempx) = mode_data(j,k,i);
                size_tempx = size_tempx + 1;
            end
        end
    end
end

```

```

    end
end
pred_data(j,k,:) = polyval(polyfit(temp_x,temp_data,4),reshape(x(j,k,:),1,data_size));
clear temp_x;
clear temp_data;
end
end

for k = 1:num_freqwalk
    diff(k,:) = abs(mode_data(1,k,:) - pred_data(1,k,:));
    dev_thresh(k) = 1.5*std(diff(k,:));
end

['Mode2 -- Prior error: '
std(abs(mode_data(2,1,:) - pred_data(2,1,:))

for i=1:data_size
    i

    status = 0;
    for k = 1:num_freqwalk
        if (abs(mode_data(1,k,i) - pred_data(1,k,i)) > dev_thresh(k))
            status = 1;
        end
    end

    num_freq_modes = 1;
    if (status)

        %look at freqdata within flaw

        %set the first series as the first frequency's data point
        freq_modes(num_freq_modes,1) = mode_data(1,1,i);
        size_freq_modes(1) = 1;

        %loop through the remaining frequencies and try to sort them accordingly
        for j = 1:num_modes+1

            for k = 1:num_freqwalk
                if(i==53)
                    i=53;
                end
                if (j == 1 & k == 1)
                    %set the first series as the first frequency's data point

```



```

    freq_modes(num_freq_modes,1) = mode_data(1,1,i);
    size_freq_modes(1) = 1;
end

temp = 1;
%loop through the frequency data points until an appropriate series is found. If
one
%is found, set temp = -1, otherwise temp will be 1 greater than number of
frequencies.
while (temp <= num_freq_modes & temp > 0)

    %Check to see if the data point belongs in the current series
    if (abs(mode_data(j,k,i) -
freq_modes(temp,size_freq_modes(temp)))/mode_data(j,k,i) < .02)
        freq_modes(temp,k) = mode_data(j,k,i);
        size_freq_modes(temp) = k;
        temp = -1;
    else
        temp = temp + 1;
    end

end

%If a series wasn't found, create a new one
if (temp > 0)
    freq_modes(num_freq_modes+1,k) = mode_data(j,k,i);
    num_freq_modes = num_freq_modes + 1;
    size_freq_modes(num_freq_modes) = k;
end
end

end

%now we have freq data sorted into approx. linear series. we need to adjust the
mode data
%as appropriate.

max_val = max(freq_modes,[],2);

[sort_min, IX] = sort(max_val);

%Need to fit missing fwalk data after sorting into series and determining series'
order
j = 1;
found_modes = 1;
while (found_modes <= num_modes & j <= size(IX,1))
    x2 = 1:10;

```

```

temp_size = 0;
for temp_k = 1:num_freqwalk
    if (freq_modes(IX(j),temp_k) > 0)
        temp_size = temp_size + 1;
        temp_mode(temp_size,1) = temp_k;
        temp_mode(temp_size,2) = freq_modes(IX(j),temp_k);
    end
end
if (temp_size > 3)
    p_freq_mode_fit = polyfit(temp_mode(:,1),temp_mode(:,2),1);
    freq_mode_fit = polyval(p_freq_mode_fit,x2);
    for temp_k = 1:num_freqwalk
        if (freq_modes(IX(j),temp_k) > 0)
            mode_data(found_modes,temp_k,i) = freq_modes(IX(j),temp_k);
        else
            mode_data(found_modes,temp_k,i) = freq_mode_fit(temp_k);
        end
    end
    found_modes = found_modes + 1;
end
j = j + 1;
clear temp_mode
clear freq_mode_fit
end

clear temp_size
clear p_freq_mode_fit
clear max_val
clear sort_min
clear IX
clear freq_modes
clear size_freq_modes

```

else %Normal sorting outside of suspected flaw region. Should I try and sort this data with

%or without the flaw points? In other words, should I predict the data without the flaw too?

```

for k = 1:num_freqwalk
    current = 1;
    next = current + 1;
    best_fit = compare( mode_data(current,k,i), mode_data(next,k,i),
pred_data(current,k,i) );

    if ( best_fit == mode_data(next, k, i) )

```

```

    for j = current:num_modes
        mode_data(j,k,i) = mode_data(j+1,k,i);
    end

end

best_fit = compare( mode_data(current,k,i), mode_data(next,k,i), pred_data(next,k,i)
);
if ( best_fit == mode_data(current,k,i) )

    if ( abs(mode_data(current,k,i) - pred_data(next,k,i)) < abs(mode_data(current,k,i)
- pred_data(current,k,i) ) )

        for j = num_modes:-1:current
            mode_data(j+1,k,i) = mode_data(j,k,i);
        end
        mode_data(current,k,i) = pred_data(current,k,i);

    end

end

for l = 2:num_modes

    if( i == 14)
        i = 14;
    end

    current = current + 1;
    next = next + 1;

    best_fit = compare( mode_data(current,k,i), mode_data(next,k,i),
pred_data(current,k,i) );
    if ( best_fit == mode_data(next,k,i) )

        for j = current:num_modes
            mode_data(j,k,i) = mode_data(j+1,k,i);
        end

    end

    best_fit = compare( mode_data(current,k,i), mode_data(next,k,i),
pred_data(next,k,i) );
    if ( best_fit == mode_data(current,k,i) )

```

```

        if ( abs(mode_data(current, k, i) - pred_data(next,k,i)) <
abs(mode_data(current,k,i) - pred_data(current,k,i) ) )

            for j = num_modes:-1:current
                mode_data(j+1,k,i) = mode_data(j,k,i);
            end
            mode_data(current,k,i) = pred_data(current,k,i);

        end

    end

end

end

end

end

end

['Mode2 -- After error: ' ]
std(abs(mode_data(2,1,:) - pred_data(2,1,:)))

toc

function [best_fit] = compare(current, next, predicted)

    if ( abs(current - predicted) <= abs(next - predicted) )

        best_fit = current;

    else

        best_fit = next;

    end
end

```

X_SCAN_GENERATOR.M

```

clear pred
count = 1;
max = 99;
for i = 0:max
    if (mod(i,2) == 0)

        for j = 0:max
            if j >= i
                %pred(count) = polyval(p,abs(((j+1)-i)));
                temp(count) = j+1-i;
                count = count + 1;
            else
                %pred(count) = polyval(p2,abs(((j+1)-i)));
                temp(count) = (j+1)-i;
                count = count + 1;
            end
        end
    end

    else

        for j = 0:max
            if (max - j) >= i
                %pred(count) = polyval(p,abs((max-(j)-i)));
                temp(count) = max-(j)-i;
                count = count + 1;
            else
                %pred(count) = polyval(p2,abs((max-(j)-i)));
                temp(count) = (max-(j)-i);
                count = count + 1;
            end
        end
    end

end

end

clear count;

for i=1:4
    for j=1:10
        x(i,j,:) = temp;
    end
end
end

```

Appendix C

Source Code for the HUT Scanner

Note: For full reconstruction source code see [6]. Only code which was changed for the HUT geometry has been included here.

```
/******BEGIN PIPEGAGE.CC******/

/* Serial motor driver for Cross Borehole Tomography + Gage acquisition board
This is the main module of the scanning software.
It opens and initializes COM-port, then calls read and write
functions to drive motor, sets the scanning geometry

Written by Eugene Malyarenko
Adapted by Kevin Leonard
*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <errno.h>          /* to track errors while doing IO*/

#include "tb1000.h"
#include "typedefs.h"

#define FALSE 0
#define TRUE 1
#define WAIT 1

static char cmd[130],cmd1[130];
static short int *d;
static short int *avg_signal;
//float *d;
int chan3 = 0;

extern SINT32 boardNo;
```

```

/***** declarations for serial *****/
#define BAUDRATE B9600
#define MOTOR "/dev/ttyS0"      /* change for your port */
#define _POSIX_SOURCE 1        /* POSIX compliant source */

#define FALSE 0
#define TRUE 1
#define WAIT 1

static double ticktime = 5.0e-3;

#ifndef LINUX
#define LINUX    /* for waitasec */
#endif
#include <math.h> /* for waitasec */
#include <time.h> /* for waitasec */

static void waitasec(const double dt = ticktime)
{
#ifdef LINUX
    struct timespec reqt;
    reqt.tv_sec = int(dt);
    reqt.tv_nsec = long(1.0e9 * fmod(dt, 1.0));
    if(reqt.tv_nsec > 999999999.0)
        reqt.tv_nsec = 999999999;
    nanosleep(&reqt, 0);
#else
    //unixusec(int(dt * 1e6));
    printf("waitasec: please define unixusec()\n");
#endif
}

static int STOP_MOTOR1=FALSE, STOP_MOTOR2=FALSE;

#include "softscope.hh"
#include "ScopeWiggleCommand.hh"
#include "GageBank.hh"

void ScopeWiggleCommand::cbh_pipe(int first, int n_samples, char *filename,
                                   int ascii_or_bin) {

    FILE *fp;

    int i, j, k, sec, STEPS_MOTOR1, STEPS_MOTOR2, NSTEPS;

```

```

int STEP_CNT, STEP;
char fname[100];
char temp[100];

int format = ascii_or_bin; /*write data in format: binary(0) or ascii(1) */

/***** serial variables *****/

extern int errno;          /*my addition from man errno*/
int fd, readnumber;
struct termios oldtio, newtio;
char buf1[40], buf2[40];

/***** serial initialization *****/
/***** all the information in man termios *****/

fd = open(MOTOR, O_RDWR | O_NOCTTY | O_NONBLOCK );
if (fd <0) {perror(MOTOR); printf("opening"); exit(-1); }

tcgetattr(fd,&oldtio); /* save current serial port settings */
bzero(&newtio, sizeof(newtio)); /* clear struct for new port settings */

/*
  BAUDRATE: Set bps rate. You could also use cfsetispeed and cfsetospeed.
  CRTSCTS : output hardware flow control (only used if the cable has
  all necessary lines. See sect. 7 of Serial-HOWTO)
  CS8   : 8n1 (8bit,no parity,1 stopbit)
  CLOCAL : local connection, no modem contol
  CREAD  : enable receiving characters
*/
newtio.c_cflag = BAUDRATE | PARENB | CSTOPB | CS7 | CREAD;

/*
  IGNPAR : ignore bytes with parity errors
  ICRNL  : map CR to NL (otherwise a CR input on the other computer
  will not terminate input)
  otherwise make device raw (no other input processing)
*/
newtio.c_iflag = INPCK | IGNPAR;

/*
  Raw output.
*/
newtio.c_oflag = OPOST;

/*

```



```

ICANON : enable canonical input
disable all echo functionality, and don't send signals to calling program
*/
newtio.c_lflag = 0;

/*
initialize all control characters
default values can be found in /usr/include/termios.h, and are given
in the comments, but we don't need them here
*/
newtio.c_cc[VINTR] = 0; /* Ctrl-c */
newtio.c_cc[VQUIT] = 0; /* Ctrl-\ */
newtio.c_cc[VERASE] = 0; /* del */
newtio.c_cc[VKILL] = 0; /* @ */
newtio.c_cc[VEOF] = 4; /* Ctrl-d */
newtio.c_cc[VTIME] = 0; /* inter-character timer unused */
newtio.c_cc[VMIN] = 1; /* blocking read until 1 character arrives */
newtio.c_cc[VSWTC] = 0; /* \0' */
newtio.c_cc[VSTART] = 0; /* Ctrl-q */
newtio.c_cc[VSTOP] = 0; /* Ctrl-s */
newtio.c_cc[VSUSP] = 0; /* Ctrl-z */
newtio.c_cc[VEOL] = 0; /* \0' */
newtio.c_cc[VREPRINT] = 0; /* Ctrl-r */
newtio.c_cc[VDISCARD] = 0; /* Ctrl-u */
newtio.c_cc[VWERASE] = 0; /* Ctrl-w */
newtio.c_cc[VLNEXT] = 0; /* Ctrl-v */
newtio.c_cc[VEOL2] = 0; /* \0' */

/*
now clean the modem line and activate the settings for the port
*/
tcflush(fd, TCIFLUSH);
tcsetattr(fd, TCSANOW, &newtio);

strcpy(cmd, "FN"); /* remote enable (no echo) and write error detection*/
readnumber= write(fd, cmd, strlen(cmd));
if (readnumber < 0) { perror(MOTOR); printf("writing"); exit(-1); }
strcpy(cmd, "C S2M450,A2M1,S1M600,A1M2"); //6000 and 10 were default
readnumber= write(fd, cmd, strlen(cmd));
if (readnumber < 0) { perror(MOTOR); printf("writing"); exit(-1); }

printf("\nYour serial port and motor port have been initialized dude\n");
/***** end initialization *****/

/***** scanner programming block *****/

```

```

//printf("Enter the number of steps per scan:\n");
//scanf("%d", &NSTEPS);
//printf("Enter the step size (mm):\n");
//scanf("%d", &STEP_CNT);
//STEP=STEP_CNT*78;          /*1mm=78 motor steps for motors 3,4*/
NSTEPS = 180;
STEP_CNT = 0;
STEP = 40;

//printf("Enter the output filename:\n");
//scanf("%40s", fname);
//printf("Data format: binary(0) ascii(1):\n");
//scanf("%d", &format);

d = (short int *)calloc( (*wmp)(chan3,0)->samples() , sizeof(short int));
//for(j=0;j<5120;j++) d[j]=0;
//d = (float *) calloc( (*wmp)(chan3,0)->samples(), sizeof(float));

fp=fopen(filename,"wb");    /* opening file for writing */

for(STEPS_MOTOR1=0;STEPS_MOTOR1<NSTEPS;STEPS_MOTOR1++){
    printf(".....SCAN #%d\n",STEPS_MOTOR1);

    if(STEPS_MOTOR1%2==0) sprintf(cmd1,"C I1M-%d,R",STEP);/*motion
direction*/
    else sprintf(cmd1,"C I1M%d,R",STEP);

    for(STEPS_MOTOR2=0;STEPS_MOTOR2<NSTEPS;STEPS_MOTOR2++){

        write(fd,cmd1,strlen(cmd1));
        STOP_MOTOR1=FALSE;
        buf1[0]='a';

        while (STOP_MOTOR1==FALSE) {
            readnumber = read(fd,buf1,1);
            /*if (readnumber <0) {perror(MOTOR); printf("reading error/n"); }*/
            buf1[readnumber]=0;
            /* printf(":%s:%d errno=%d\n", buf1, readnumber,errno);*/
            if (buf1[0] == '^') STOP_MOTOR1=TRUE;
        }

        /****** waveform acquisition block *****/

        //app->CheckEvents();          /* IMPORTANT!!!, updates screen */

```

```

//swcmdp->realKeyIn('n', 0); /* fetches the next sample */

/***** Cool use of my_fetch *****/

while(!gsp->triggered())
    waitasec();
while(gsp->busy())
    waitasec();

gsp->my_fetch(0, d, first, n_samples); /* fetches trace into d */

gsp->startcapture(); /* without it no trigger */

/*****/

//app->AcquireControl(); // activate if we want to update screen

// for(int i = 0; i < (*wmp)(chan3,0)->samples(); i++)
//d[i] = ( (*wmp)(chan3,0)->internal())[i];

//for(j=0;j< (*wmp)(chan3,0)->samples(); j++) {

switch(format) {
case 0: /*binary format
    for(j = 0; j < n_samples; j++)
        fwrite(&d[j], sizeof(short int), 1, fp);
    break;

case 1: /*ascii format
    for(j = 0; j < n_samples; j++) {
        fprintf(fp,"%d\n",d[j]);
    }
    break;

default:
    printf("please restart and input valid data format...\n");
    exit(1);
}
} /*end loop over STEPS_MOTOR2 */

/*motor2 has completed the cycle, now moving motor1 one step*/
sprintf(cmd,"C I2M%d,R",STEP); /*command for the first motor to
move*/
write(fd,cmd,strlen(cmd));
STOP_MOTOR2=FALSE;

```

```

buf2[0]='a';

while (STOP_MOTOR2==FALSE) {
    readnumber = read(fd,buf2,1);
    buf2[readnumber]=0;
    if (buf2[0] == '^') STOP_MOTOR2=TRUE;
}
/* printf(":%s:%d \n", buf2, readnumber);*/

} /*end loop over STEPS_MOTOR1 */

fclose(fp);          /* closing data file */

/***** end data collection *****/

/*****now move motor back after pressing any key*****/

sprintf(cmd,"C S2M4000,A2M10, I2M-%d,R",STEP*NSTEPS); /*all the way
back*/

write(fd,cmd,strlen(cmd));
STOP_MOTOR2=FALSE;
buf1[0]='a';

while (STOP_MOTOR2==FALSE) {
    readnumber = read(fd,buf1,1);
    buf1[readnumber]=0;
    if (buf1[0] == '^') STOP_MOTOR2=TRUE;
}

/*****motor is again at starting position *****/

/***** closing and exiting *****/
strcpy(cmd,"Q");      /*set off-line mode*/
write(fd,cmd,strlen(cmd));
tcsetattr(fd,TCSANOW,&oldtio); /* restore the old port settings */

close(fd);

gsp->scan_info (fname); /* output scan protocol into a file */

```

```

} /* end cbh_serial */

void ScopeWiggleCommand::cbh_pipe_freq(int first, int n_samples,
                                       char *filename, int ascii_or_bin) {

    FILE *fp, *freq_out[10];
    int number_of_frequencies;          //Max of 10
    float frequency[10];

    int freq,i, j, k, sec, STEPS_MOTOR1, STEPS_MOTOR2, NSTEPS;
    int STEP_CNT, STEP;

    char temp[100];

    int format = ascii_or_bin; /*write data in format: binary(0) or ascii(1) */

    /****** serial variables *****/

    extern int errno;                  /*my addition from man errno*/
    int fd, readnumber;
    struct termios oldtio, newtio;
    char buf1[40], buf2[40];

    /****** serial initialization *****/
    /****** all the information in man termios *****/

    fd = open(MOTOR, O_RDWR | O_NOCTTY | O_NONBLOCK );
    if (fd <0) {perror(MOTOR); printf("opening"); exit(-1); }

    tcgetattr(fd,&oldtio); /* save current serial port settings */
    bzero(&newtio, sizeof(newtio)); /* clear struct for new port settings */

    /*
    BAUDRATE: Set bps rate. You could also use cfsetispeed and cfsetospeed.
    CRTSCTS : output hardware flow control (only used if the cable has
    all necessary lines. See sect. 7 of Serial-HOWTO)
    CS8 : 8n1 (8bit,no parity,1 stopbit)
    CLOCAL : local connection, no modem contol
    CREAD : enable receiving characters
    */
    newtio.c_cflag = BAUDRATE | PARENB | CSTOPB | CS7 | CREAD;

    /*
    IGNPAR : ignore bytes with parity errors

```

```

ICRNL : map CR to NL (otherwise a CR input on the other computer
will not terminate input)
otherwise make device raw (no other input processing)
*/
newtio.c_iflag = INPCK | IGNPAR;

/*
Raw output.
*/
newtio.c_oflag = OPOST;

/*
ICANON : enable canonical input
disable all echo functionality, and don't send signals to calling program
*/
newtio.c_lflag = 0;

/*
initialize all control characters
default values can be found in /usr/include/termios.h, and are given
in the comments, but we don't need them here
*/
newtio.c_cc[VINTR] = 0; /* Ctrl-c */
newtio.c_cc[VQUIT] = 0; /* Ctrl-\ */
newtio.c_cc[VERASE] = 0; /* del */
newtio.c_cc[VKILL] = 0; /* @ */
newtio.c_cc[VEOF] = 4; /* Ctrl-d */
newtio.c_cc[VTIME] = 0; /* inter-character timer unused */
newtio.c_cc[VMIN] = 1; /* blocking read until 1 character arrives */
newtio.c_cc[VSWTC] = 0; /* \0' */
newtio.c_cc[VSTART] = 0; /* Ctrl-q */
newtio.c_cc[VSTOP] = 0; /* Ctrl-s */
newtio.c_cc[VSUSP] = 0; /* Ctrl-z */
newtio.c_cc[VEOL] = 0; /* \0' */
newtio.c_cc[VREPRINT] = 0; /* Ctrl-r */
newtio.c_cc[VDISCARD] = 0; /* Ctrl-u */
newtio.c_cc[VWERASE] = 0; /* Ctrl-w */
newtio.c_cc[VLNEXT] = 0; /* Ctrl-v */
newtio.c_cc[VEOL2] = 0; /* \0' */

/*
now clean the modem line and activate the settings for the port
*/
tcflush(fd, TCIFLUSH);
tcsetattr(fd, TCSANOW, &newtio);

```

```

strcpy(cmd,"FN"); /* remote enable (no echo) and write error detection*/
readnumber= write(fd,cmd,strlen(cmd));
if (readnumber <0) {perror(MOTOR); printf("writing"); exit(-1); }
strcpy(cmd,"C S2M6000,A2M10,S1M6000,A1M10");
readnumber= write(fd,cmd,strlen(cmd));
if (readnumber <0) {perror(MOTOR); printf("writing"); exit(-1); }

printf("\nYour serial port and motor port have been initialized dude\n");
/***** end initialization *****/

/***** scanner programming block *****/

NSTEPS = 180;
STEP_CNT = 0;
STEP = 40;

d = (short int *)calloc( (*wmp)(chan3,0)->samples() , sizeof(short int));

//for(j=0;j<5120;j++) d[j]=0;
//d = (float *) calloc( (*wmp)(chan3,0)->samples(), sizeof(float));

printf("Enter the number of frequencies (max 10):\n");
scanf("%d", &number_of_frequencies);
for (j = 0; j < number_of_frequencies; j++) {
    printf("Enter frequency %d (in MHz):\n", j+1);
    scanf("%f", &frequency[j]);
    sprintf(temp, "%s_freq%d", filename, j+1);
    freq_out[j] = fopen(temp,"wb");
}

for(STEPS_MOTOR1=0;STEPS_MOTOR1 < NSTEPS;STEPS_MOTOR1++){
    printf(".....SCAN #%d\n",STEPS_MOTOR1);

    if(STEPS_MOTOR1%2==0) sprintf(cmd1,"C I1M-%d,R",STEP);/*motion
direction*/
    else sprintf(cmd1,"C I1M%d,R",STEP);

    for(STEPS_MOTOR2=0;STEPS_MOTOR2<NSTEPS;STEPS_MOTOR2++){

        write(fd,cmd1,strlen(cmd1));
        STOP_MOTOR1=FALSE;
        buf1[0]='a';

        while (STOP_MOTOR1==FALSE) {
            readnumber = read(fd,buf1,1);

```

```

    /*if (readnumber <0) {perror(MOTOR); printf("reading error/n"); }*/
    buf1[readnumber]=0;
    /* printf(":%s:%d errno=%d\n", buf1, readnumber,errno);*/
    if (buf1[0] == '^') STOP_MOTOR1=TRUE;
}

/***** waveform acquisition block *****/

//app->CheckEvents();      /* IMPORTANT!!!, updates screen */
//swcmdp-> realKeyIn('n', 0); /* fetches the next sample */

/***** Cool use of my_fetch *****/

for(freq = 0; freq < number_of_frequencies; freq++) {

    _SetFrequency( boardNo, double (frequency[freq]*1000000));
    waitasec();
    _SetFrequency( boardNo, double (frequency[freq]*1000000));

    for(k = 0; k<2; k++) {

        while(!gsp->triggered())
            waitasec();
        while(gsp->busy())
            waitasec();

        gsp->my_fetch(0, d, first, n_samples); /* fetches trace
                                                into d */

        gsp->startcapture(); /* without it no trigger */

        /*****/

        //app->AcquireControl(); // activate if we want to update screen

        switch(format) {
        case 0:          /*binary format
            if (k == 1) {

                for (j = 0; j < n_samples; j++) {
                    fwrite(&d[j], sizeof(short int), 1,
                        freq_out[freq]);
                }

            }
        }
        break;

```



```

    case 1:                //ascii format
        if(k == 1) {
            for (j = 0; j < n_samples; j++) {
                fprintf(freq_out[freq], "%d\n", d[j]);
            }
        }
        break;

    default:
        printf("please restart and input valid data format...\n");
        exit(1);
    }
} //end for k
} /*end of frequency loop*/

} /*end loop over STEPS_MOTOR2 */

/*motor2 has completed the cycle, now moving motor1 one step*/

sprintf(cmd,"C I2M%d,R",STEP); //command for the first motor to move
write(fd,cmd,strlen(cmd));
STOP_MOTOR2=FALSE;
buf2[0]='a';

while (STOP_MOTOR2==FALSE) {
    readnumber = read(fd,buf2,1);
    buf2[readnumber]=0;
    if (buf2[0] == '^') STOP_MOTOR2=TRUE;
}

/* printf(":%s:%d \n", buf2, readnumber);*/
} /*end loop over STEPS_MOTOR1 */

for(j = 0; j < number_of_frequencies; j++) {
    fclose(freq_out[j]);
}

/***** end data collection *****/

/*****now move motor back after pressing any key*****/

sprintf(cmd,"C S2M4000,A2M10, I2M-%d,R",STEP*NSTEPS); /*all the way
back*/ //CHANGE BACK TO 2M- FOR FULL SCAN!!!!

write(fd,cmd,strlen(cmd));

```

```

STOP_MOTOR2=FALSE;
buf1[0]='a';

while (STOP_MOTOR2==FALSE) {
    readnumber = read(fd,buf1,1);
    buf1[readnumber]=0;
    if (buf1[0] == '^') STOP_MOTOR2=TRUE;
}

/*****motor is again at starting position *****/

/***** closing and exiting *****/
strep(cmd,"Q");          /*set off-line mode*/
write(fd,cmd,strlen(cmd));
tcsetattr(fd,TCSANOW,&oldtio); /* restore the old port settings */

close(fd);

gsp->scan_info (filename); /* output scan protocol into a file */

} /* end cbh_serial_freq */

void ScopeWiggleCommand::cbh_pipe_freq_avg(int first, int n_samples,
                                           char *filename, int ascii_or_bin) {

    FILE *fp, *freq_out[10];
    int number_of_frequencies;          //Max of 10
    float frequency[10];

    int freq,i, j, k, sec, STEPS_MOTOR1, STEPS_MOTOR2, NSTEPS, i_avg;
    int STEP_CNT, STEP;

    char temp[100];

    int format = ascii_or_bin; /*write data in format: binary(0) or ascii(1) */

    /***** serial variables *****/

    extern int errno;          /*my addition from man errno*/
    int fd, readnumber;
    struct termios oldtio, newtio;

```

```

char buf1[40], buf2[40];

/***** serial initialization *****/
/***** all the information in man termios *****/

fd = open(MOTOR, O_RDWR | O_NOCTTY | O_NONBLOCK );
if (fd < 0) {perror(MOTOR); printf("opening"); exit(-1); }

tcgetattr(fd,&oldtio); /* save current serial port settings */
bzero(&newtio, sizeof(newtio)); /* clear struct for new port settings */

/*
BAUDRATE: Set bps rate. You could also use cfsetispeed and cfsetospeed.
CRTSCTS : output hardware flow control (only used if the cable has
all necessary lines. See sect. 7 of Serial-HOWTO)
CS8 : 8n1 (8bit,no parity,1 stopbit)
CLOCAL : local connection, no modem contol
CREAD : enable receiving characters
*/
newtio.c_cflag = BAUDRATE | PARENB | CSTOPB | CS7 | CREAD;

/*
IGNPAR : ignore bytes with parity errors
ICRNL : map CR to NL (otherwise a CR input on the other computer
will not terminate input)
otherwise make device raw (no other input processing)
*/
newtio.c_iflag = INPCK | IGNPAR;

/*
Raw output.
*/
newtio.c_oflag = OPOST;

/*
ICANON : enable canonical input
disable all echo functionality, and don't send signals to calling program
*/
newtio.c_lflag = 0;

/*
initialize all control characters
default values can be found in /usr/include/termios.h, and are given
in the comments, but we don't need them here
*/
newtio.c_cc[VINTR] = 0; /* Ctrl-c */

```

```

newtio.c_cc[VQUIT] = 0; /* Ctrl-\ */
newtio.c_cc[VERASE] = 0; /* del */
newtio.c_cc[VKILL] = 0; /* @ */
newtio.c_cc[VEOF] = 4; /* Ctrl-d */
newtio.c_cc[VTIME] = 0; /* inter-character timer unused */
newtio.c_cc[VMIN] = 1; /* blocking read until 1 character arrives */
newtio.c_cc[VSWTC] = 0; /* \0' */
newtio.c_cc[VSTART] = 0; /* Ctrl-q */
newtio.c_cc[VSTOP] = 0; /* Ctrl-s */
newtio.c_cc[VSUSP] = 0; /* Ctrl-z */
newtio.c_cc[VEOL] = 0; /* \0' */
newtio.c_cc[VREPRINT] = 0; /* Ctrl-r */
newtio.c_cc[VDISCARD] = 0; /* Ctrl-u */
newtio.c_cc[VWERASE] = 0; /* Ctrl-w */
newtio.c_cc[VLNEXT] = 0; /* Ctrl-v */
newtio.c_cc[VEOL2] = 0; /* \0' */

/*
now clean the modem line and activate the settings for the port
*/
tcflush(fd, TCIFLUSH);
tcsetattr(fd,TCSANOW,&newtio);

strcpy(cmd,"FN"); /* remote enable (no echo) and write error detection*/
readnumber= write(fd,cmd,strlen(cmd));
if (readnumber <0) { perror(MOTOR); printf("writing"); exit(-1); }
strcpy(cmd,"C S2M6000,A2M10,S1M6000,A1M10");
readnumber= write(fd,cmd,strlen(cmd));
if (readnumber <0) { perror(MOTOR); printf("writing"); exit(-1); }

printf("\nYour serial port and motor port have been initialized dude\n");
/***** end initialization *****/

/***** scanner programming block
*****/

NSTEPS = 180;
STEP_CNT = 0;
STEP = 40;

d = (short int *)calloc( (*wmp)(chan3,0)->samples() , sizeof(short int));
avg_signal = (short int *)calloc( n_samples, sizeof(short int));

for (j = 0; j < n_samples; j++) {
    avg_signal[j] = 0;
}

```

```

//for(j=0;j<5120;j++) d[j]=0;
//d = (float *) calloc( (*wmp)(chan3,0)->samples(), sizeof(float));

printf("Enter the number of frequencies (max 10):\n");
scanf("%d", &number_of_frequencies);
for (j = 0; j < number_of_frequencies; j++) {
    printf("Enter frequency %d (in MHz):\n", j+1);
    scanf("%f", &frequency[j]);
    sprintf(temp, "%s_freq%d", filename, j+1);
    freq_out[j] = fopen(temp,"wb");
}

for(STEPS_MOTOR1=0;STEPS_MOTOR1<NSTEPS;STEPS_MOTOR1++){
    printf(".....SCAN #%d\n",STEPS_MOTOR1);

    if(STEPS_MOTOR1%2==0) sprintf(cmd1,"C I1M-%d,R",STEP);/*motion
direction*/
    else sprintf(cmd1,"C I1M%d,R",STEP);

    for(STEPS_MOTOR2=0;STEPS_MOTOR2<NSTEPS;STEPS_MOTOR2++){

        write(fd,cmd1,strlen(cmd1));
        STOP_MOTOR1=FALSE;
        buf1[0]='a';

        while (STOP_MOTOR1==FALSE) {
            readnumber = read(fd,buf1,1);
            /*if (readnumber <0) {perror(MOTOR); printf("reading error/n"); }*/
            buf1[readnumber]=0;
            /* printf(":%s:%d errno=%d\n", buf1, readnumber,errno);*/
            if (buf1[0] == '^') STOP_MOTOR1=TRUE;
        }

        /****** waveform acquisition block *****/

        //app->CheckEvents();          /* IMPORTANT!!!, updates screen */
        //swcmdp->realKeyIn('n', 0); /* fetches the next sample */

        /****** Cool use of my_fetch *****/

        for(freq = 0; freq < number_of_frequencies; freq++) {

            _SetFrequency( boardNo, double (frequency[freq]*1000000));
            waitasec();
            _SetFrequency( boardNo, double (frequency[freq]*1000000));

```

```

for(k = 0; k<2; k++) {

    while(!gsp->triggered())
        waitasec();
    while(gsp->busy())
        waitasec();

    gsp->my_fetch(0, d, first, n_samples); /* fetches trace
                                           into d */

    gsp->startcapture(); /* without it no trigger */

    /*****/

    //app->AcquireControl(); // activate if we want to update screen

    switch(format) {
    case 0: /*binary format
    if (k == 1) {

        for (j = 0; j < n_samples; j++) {
            avg_signal[j] = 0;
        }
        for (i_avg = 0; i_avg < 10; i_avg++) {

            while(!gsp->triggered())
                waitasec();
            while(gsp->busy())
                waitasec();

            gsp->my_fetch(0, d, first, n_samples);
            gsp->startcapture(); /* without it no trigger */

            for (j = 0; j < n_samples; j++) {
                avg_signal[j] = avg_signal[j] + d[j];
            }
        }
        for (j = 0; j < n_samples; j++) {
            avg_signal[j] = avg_signal[j]/10.0;
            fwrite(&avg_signal[j], sizeof(short int), 1,
                freq_out[freq]);
        }

    }
    break;

```

```

case 1:                //ascii format
if(k == 1) {

    for (j = 0; j < n_samples; j++) {
        avg_signal[j] = 0;
    }
    for (i_avg = 0; i_avg < 10; i_avg++) {

        while(!gsp->triggered())
            waitasec();
        while(gsp->busy())
            waitasec();

        gsp->my_fetch(0, d, first, n_samples);
        gsp->startcapture();    /* without it no trigger */

        for (j = 0; j < n_samples; j++) {
            avg_signal[j] = avg_signal[j] + d[j];
        }
    }
    for (j = 0; j < n_samples; j++) {
        avg_signal[j] = avg_signal[j]/10.0;
        fprintf(freq_out[freq], "%d\n", avg_signal[j]);
    }

}
break;

default:
    printf("please restart and input valid data format...\n");
    exit(1);
} //end for k
} /*end of frequency loop*/

} /*end loop over STEPS_MOTOR2 */

/*motor2 has completed the cycle, now moving motor1 one step*/
sprintf(cmd,"C I2M%d,R",STEP); /*command for the first motor to move*/
write(fd,cmd,strlen(cmd));
STOP_MOTOR2=FALSE;
buf2[0]='a';

while (STOP_MOTOR2==FALSE) {
    readnumber = read(fd,buf2,1);
    buf2[readnumber]=0;
}

```

```

    if (buf2[0] == '^') STOP_MOTOR2=TRUE;
  }
  /* printf(":%s:%d \n", buf2, readnumber);*/

} /*end loop over STEPS_MOTOR1 */

for(j = 0; j < number_of_frequencies; j++) {
  fclose(freq_out[j]);
}

/***** end data collection *****/

/*****now move motor back after pressing any key*****/

sprintf(cmd,"C S2M4000,A2M10, I2M-%d,R",STEP*NSTEPS); /*all the way back*/

write(fd,cmd,strlen(cmd));
STOP_MOTOR2=FALSE;
buf1[0]='a';

while (STOP_MOTOR2==FALSE) {
  readnumber = read(fd,buf1,1);
  buf1[readnumber]=0;
  if (buf1[0] == '^') STOP_MOTOR2=TRUE;
}

/*****motor is again at starting position *****/

/***** closing and exiting *****/
strcpy(cmd,"Q"); /*set off-line mode*/
write(fd,cmd,strlen(cmd));
tcsetattr(fd,TCSANOW,&oldtio); /* restore the old port settings */

close(fd);

gsp->scan_info (filename); /* output scan protocol into a file */

} /* end cbh_serial_freq_avg */

void ScopeWiggleCommand::cbh_pipe_avg(int first, int n_samples,
char *filename, int ascii_or_bin) {

```



```

FILE *fp;

int i, j, k, sec, STEPS_MOTOR1, STEPS_MOTOR2, NSTEPS, i_avg;
int STEP_CNT, STEP;

char temp[100];

int format = ascii_or_bin; /*write data in format: binary(0) or ascii(1) */

/***** serial variables *****/

extern int errno;          /*my addition from man errno*/
int fd, readnumber;
struct termios oldtio, newtio;
char buf1[40], buf2[40];

/***** serial initialization *****/
/***** all the information in man termios *****/

fd = open(MOTOR, O_RDWR | O_NOCTTY | O_NONBLOCK );
if (fd <0) {perror(MOTOR); printf("opening"); exit(-1); }

tcgetattr(fd,&oldtio); /* save current serial port settings */
bzero(&newtio, sizeof(newtio)); /* clear struct for new port settings */

/*
  BAUDRATE: Set bps rate. You could also use cfsetispeed and cfsetospeed.
  CRTSCTS : output hardware flow control (only used if the cable has
all necessary lines. See sect. 7 of Serial-HOWTO)
  CS8    : 8n1 (8bit,no parity,1 stopbit)
  CLOCAL : local connection, no modem contol
  CREAD  : enable receiving characters
*/
newtio.c_cflag = BAUDRATE | PARENB | CSTOPB | CS7 | CREAD;

/*
  IGNPAR : ignore bytes with parity errors
  ICRNL  : map CR to NL (otherwise a CR input on the other computer
will not terminate input)
  otherwise make device raw (no other input processing)
*/
newtio.c_iflag = INPCK | IGNPAR;

/*
  Raw output.

```

```

*/
newtio.c_oflag = OPOST;

/*
  ICANON : enable canonical input
  disable all echo functionality, and don't send signals to calling program
*/
newtio.c_lflag = 0;

/*
  initialize all control characters
  default values can be found in /usr/include/termios.h, and are given
  in the comments, but we don't need them here
*/
newtio.c_cc[VINTR] = 0; /* Ctrl-c */
newtio.c_cc[VQUIT] = 0; /* Ctrl-\ */
newtio.c_cc[VERASE] = 0; /* del */
newtio.c_cc[VKILL] = 0; /* @ */
newtio.c_cc[VEOF] = 4; /* Ctrl-d */
newtio.c_cc[VTIME] = 0; /* inter-character timer unused */
newtio.c_cc[VMIN] = 1; /* blocking read until 1 character arrives */
newtio.c_cc[VSWTC] = 0; /* '\0' */
newtio.c_cc[VSTART] = 0; /* Ctrl-q */
newtio.c_cc[VSTOP] = 0; /* Ctrl-s */
newtio.c_cc[VSUSP] = 0; /* Ctrl-z */
newtio.c_cc[VEOL] = 0; /* '\0' */
newtio.c_cc[VREPRINT] = 0; /* Ctrl-r */
newtio.c_cc[VDISCARD] = 0; /* Ctrl-u */
newtio.c_cc[VWERASE] = 0; /* Ctrl-w */
newtio.c_cc[VLNEXT] = 0; /* Ctrl-v */
newtio.c_cc[VEOL2] = 0; /* '\0' */

/*
  now clean the modem line and activate the settings for the port
*/
tcflush(fd, TCIFLUSH);
tcsetattr(fd, TCSANOW, &newtio);

strcpy(cmd, "FN"); /* remote enable (no echo) and write error detection*/
readnumber= write(fd,cmd,strlen(cmd));
if (readnumber <0) { perror(MOTOR); printf("writing"); exit(-1); }
strcpy(cmd, "C S2M6000,A2M10,S1M2500,A1M2");
readnumber= write(fd,cmd,strlen(cmd));
if (readnumber <0) { perror(MOTOR); printf("writing"); exit(-1); }

printf("\nYour serial port and motor port have been initialized dude\n");

```

```

/***** end initialization *****/

/***** scanner programming block
*****/

NSTEPS = 180;
STEP_CNT = 0;
STEP = 40;

fp=fopen(filename,"wb");

d = (short int *)calloc( (*wmp)(chan3,0)->samples() , sizeof(short int));
avg_signal = (short int *)calloc( n_samples, sizeof(short int));

for (j = 0; j < n_samples; j++) {
    avg_signal[j] = 0;
}

//for(j=0;j<5120;j++) d[j]=0;
//d = (float *) calloc( (*wmp)(chan3,0)->samples(), sizeof(float));

for(STEPS_MOTOR1=0;STEPS_MOTOR1<NSTEPS;STEPS_MOTOR1++){
    printf(".....SCAN #%d\n",STEPS_MOTOR1);

    if(STEPS_MOTOR1%2==0) sprintf(cmd1,"C I1M-%d,R",STEP);/*motion
direction*/
    else sprintf(cmd1,"C I1M%d,R",STEP);

    for(STEPS_MOTOR2=0;STEPS_MOTOR2<NSTEPS;STEPS_MOTOR2++){

        write(fd,cmd1,strlen(cmd1));
        STOP_MOTOR1=FALSE;
        buf1[0]='a';

        while (STOP_MOTOR1==FALSE) {
            readnumber = read(fd,buf1,1);
            /*if (readnumber <0) {perror(MOTOR); printf("reading error/n"); }*/
            buf1[readnumber]=0;
            /* printf(":%s:%d errno=%d\n", buf1, readnumber,errno);*/
            if (buf1[0] == '^') STOP_MOTOR1=TRUE;
        }

    }

/***** waveform acquisition block *****/

//app->CheckEvents();      /* IMPORTANT!!!, updates screen */
//swcmdp-> realKeyIn('n', 0); /* fetches the next sample */

```

```

/***** Cool use of my_fetch *****/

while(!gsp->triggered())
    waitasec();
while(gsp->busy())
    waitasec();

gsp->my_fetch(0, d, first, n_samples); /* fetches trace
                                     into d */

gsp->startcapture(); /* without it no trigger */

/*****/

//app->AcquireControl(); // activate if we want to update screen

switch(format) {
case 0: /*binary format

        for (j = 0; j < n_samples; j++) {
            avg_signal[j] = 0;
        }
        for (i_avg = 0; i_avg < 10; i_avg++) {

            while(!gsp->triggered())
                waitasec();
            while(gsp->busy())
                waitasec();

            gsp->my_fetch(0, d, first, n_samples);
            gsp->startcapture(); /* without it no trigger */

            for (j = 0; j < n_samples; j++) {
                avg_signal[j] = avg_signal[j] + d[j];
            }
        }
        for (j = 0; j < n_samples; j++) {
            avg_signal[j] = avg_signal[j]/10.0;
            fwrite(&avg_signal[j], sizeof(short int), 1,
                fp);
        }

        break;

case 1: /*ascii format

```

```

for (j = 0; j < n_samples; j++) {
    avg_signal[j] = 0;
}
for (i_avg = 0; i_avg < 10; i_avg++) {

    while(!gsp->triggered())
        waitasec();
    while(gsp->busy())
        waitasec();

    gsp->my_fetch(0, d, first, n_samples);
    gsp->startcapture(); /* without it no trigger */

    for (j = 0; j < n_samples; j++) {
        avg_signal[j] = avg_signal[j] + d[j];
    }
}
for (j = 0; j < n_samples; j++) {
    avg_signal[j] = avg_signal[j]/10.0;
    fprintf(fp, "%d\n", avg_signal[j]);
}
break;

default:
    printf("please restart and input valid data format...\n");
    exit(1);
}

} /*end loop over STEPS_MOTOR2 */

/*motor2 has completed the cycle, now moving motor1 one step*/
sprintf(cmd,"C I2M%d,R",STEP); /*command for the first motor to move*/
write(fd,cmd,strlen(cmd));
STOP_MOTOR2=FALSE;
buf2[0]='a';

while (STOP_MOTOR2==FALSE) {
    readnumber = read(fd,buf2,1);
    buf2[readnumber]=0;
    if (buf2[0] == '^') STOP_MOTOR2=TRUE;
}
/* printf(":%s:%d \n", buf2, readnumber);*/

} /*end loop over STEPS_MOTOR1 */

```

```

fclose(fp);

/***** end data collection *****/

/*****now move motor back after pressing any key*****/

sprintf(cmd,"C S2M4000,A2M10, I2M-%d,R",STEP*NSTEPS); /*all the way back*/

write(fd,cmd,strlen(cmd));
STOP_MOTOR2=FALSE;
buf1[0]='a';

while (STOP_MOTOR2==FALSE) {
    readnumber = read(fd,buf1,1);
    buf1[readnumber]=0;
    if (buf1[0] == '^') STOP_MOTOR2=TRUE;
}

/*****motor is again at starting position *****/

/***** closing and exiting *****/
strcpy(cmd,"Q"); /*set off-line mode*/
write(fd,cmd,strlen(cmd));
tcsetattr(fd,TCSANOW,&oldtio); /* restore the old port settings */

close(fd);

gsp->scan_info (filename); /* output scan protocol into a file */

} /* end cbh_serial_avg */

/*****END PIPEGAGE.CC*****/

/*****BEGIN TRACER.H*****/

/*
Definitions for Lamb Wave Tomographic Reconstructions
*/

#include "graphapp.h" //for GUI

#include <stdio.h>

```

```

#include <math.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

/***** common definitions *****/

int TRANS_PER_ARRAY; /* = # of pixels in the image array */
double DIST_BET_TRANS; /* in mm (this is the cell length) */
#define ITERATIONS 1

/***** iterations at different stages *****/

#define GLOBAL_ITERATIONS 1 /* # of image tracings, used in "main" */
#define STRAIGHT_ITERATIONS 50 /* # of iterations used in straight_SIRT */
/*
#define CURVED_ITERATIONS 50 /* # of iterations used in curved_SIRT */

/***** straight-ray definitions *****/
#define SAMPLING_RATE 2.5E7 /*current SR is 100 MHz */
#define SAMPLING_INTERVAL 1.0E6/SAMPLING_RATE /* in microseconds */

int NUMBER_OF_POINTS; /* points in the trace */
int FIRST_POINT; /* first point after trigger */
int DELAY;

int FREQ;
#define DAC_RESOLUTION 4096 /* for 12-bit card */
#define EST_VELOCITY 4.8 /* in mm/us */
int NUMBER_OF_RAYS;

int WAVELENGTH;

/***** corner definitions *****/

#define CORNER_STEPS 14

int STEPS_IN_J;

int CORNER_POINTS; /* points in the trace */

int CORNER_FIRST_POINT; /* first point after trigger */

int CORNER_DELAY;

/***** other definitions *****/

```

```

int THRESH_CONTROL_POINT;

#define NUM_RAYHOLDERS 2      /* we need 2 rayholders to switch */

#define NUM_PROJECTIONS 1 /*!!!! # of projections used for
reconstruction */

#define true 1
#define false 0

#define right 1
#define left 0

#define boolean int

#define SQUARE(x) ( (x) * (x) )
#define MAX(a,b) ( ((a) > (b)) ? (a) : (b) )
#define MIN(a,b) ( ((a) < (b)) ? (a) : (b) )

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

/***** Declaration for GUI *****/

window W;
window W_truncate;

field threshold_control_point;
field truncate_high_gate, truncate_low_gate;
field frequency_field, number_of_steps_field, step_size_field;
field parallel_low_gate_limit, parallel_high_gate_limit;
field corner_low_gate_limit, corner_high_gate_limit, number_corner_steps;
radiobutton corner_scan_type, parallel_scan_type;
radiobutton binary_data_type, ascii_data_type;
radiogroup scan_type_group, data_type_group;
/*****/

typedef struct InitParams {

    double i_frequency;
    int i_numberofsteps;
    double i_stepsize;
    int i_pfirstpoint;
    int i_pnumberofpoints;

```



```

int i_cfirstpoint;
int i_cnumberofpoints;
int i_controlpoint;

} InitParams;

typedef struct Point {
    double x;
    double y;
} Point;

typedef struct vortex {
    boolean last;      /* it is true only for the last point of the ray */
    Point pt;         /* vortex point in the ray */
} vortex;

typedef struct segment_params /* the same as for the straight ray case */
{
    short int p; /*p,q are coordinates of the cell the segment belongs to */
    short int q; /* we take p=bottom and q=left sides of the cell */
    double length; /* the length of the segment */
}segment_params;

/***** External array declarations *****/

double **exp_data; /* arrival times from the data file */
double **contrast_image; /* an image array with stretched contrast */
double **Image; /* an array containing evolving image*/

//short int
cell_count[TRANS_PER_ARRAY][NUM_PROJECTIONS*TRANS_PER_ARRAY];//c
ells the ray crosses
short int **cell_count;
//double
curved_ray_length[TRANS_PER_ARRAY][NUM_PROJECTIONS*TRANS_PER_AR
RAY]; //ray lengths
double **curved_ray_length;

vortex **Ray_Holder; /* stores traced ray, we need 2 of them */

segment_params ***segment_data; /* stores curved ray data */

/***** below are function declarations and corresponding source files *****/

```

```

/***** file: gettimes.c *****/

void get_times(char filename[], int scan_type, int data_type, int control_point);
double corner_ray_length (int i, int j);

/***** file: parallel_scale.c *****/

void parallel_scale(char filename[]);
void corner_scale(char filename[]);

/***** file: recon.c *****/

void get_reconstruction(char filename[]);

/***** file: truncate.c *****/

void truncatetimes(char filename[], int scan_type, double low_gate, double high_gate);

/***** file: memory_funcs.c *****/

void init_images (int N);
void init_segment_table(int N);
void init_rayholders(int N);

void insert_ray_information
(
  int i,
  int j,
  int Q_num
);

void free_segment_pointers
(
  int start_proj,
  int end_proj,
  int N
);
void free_segment_table(int N);
double roundoff(double x);

/***** file: shooting.c *****/

int golden_shooting
(
  double **image_array,
  Point Start,
  Point End,

```

```

int j    //second ray index
);

double golden
(
double **image_array,
Point Start,
Point End,
double ax,
double bx,
double cx
);

/***** file: image_proc.c *****/
void construct_fantom(double **image_array);

void subtract_reference
(
double ** result,
double **array1,
double **array2,
int N
);

void smooth_image (double **image_array, int MASK_SIZE); /*SIZE is odd */
void Sort(double *array, int size);
void median_filter(double **input_array, int MASK_SIZE); /*SIZE is odd */

/***** file: io_funcs.c *****/

typedef struct pgm_image /* holds all necessary components of *.pgm */
{
int Nrow;
int Ncol;
int gray_levels;
unsigned char **image_data;
}PGM;

void print_rayholder(int Q_num);
int display_data (double **image_array, int flag, char *tag);
int display_raw_data (double **image_array, int flag, char *tag);
int display_inverted_data (double **image_array, int flag, char *tag);
void get_data (char filename[], double **exp_data);

```

```

void overwrite_image(double **from, double **to);
void overwrite_contrast_image(double **from, double **to);
void overwrite_inverted_contrast_image(double **from, double **to);
void read_pgm(char *fname, double **image_array);
void read_float (double **image_array, char *fname, int N);
void save_float (double **image_array, char *fname, int N);
void write_pgm (PGM *P, char *out_fname, double **array);

/***** straight_ray_sirt.c *****/
int straight_ray_sirt(void);

double hamming(int M, int m);  /* Hamming (hanning) window function */

/***** sirt.c *****/

int sirt
(
  int start_proj,
  int end_proj,
  int ray_type,
  int map
);

void estimate_times /* estimates traveltimes along curved rays */
(
  double **est_times,
  double **velocities,
  int i,
  int j
);

void update_velocities
(
  double **exp_data,
  double **est_times,
  double **delta,
  int **cnt,
  int i,
  int j
);

/***** rk_snell_tracer.c *****/

```

```

double rk4    /* sigle Runge-Kutta step for the Snell's law ODE */
(
double **image_array,
double theta,    /* value of the variable */
double dtheta_ds, /* value of the first derivative at x */
Point pt,        /* image point */
double ds        /* fixed stepsize along the ray, independent variable*/
);

typedef struct time_dist
{
double t;
double last_chunk;
} time_dist;

time_dist rk_dumb_tracer /* Fixed stepsize 4-th order Runge-Kutta tracer */
(
double **image_array,
Point Start,
Point End,
double theta_start,
int Q_num
);

double derivs /* returns deivative dtheta/ds according to the ODE */
(
double **image_array,
double theta,
double x,        /* the raypoint */
double y
);

int retrace_ray
(
int Q_num
);

int trace_line_segment
(
Point Start,
Point End,
int Q_num,        /* # of the rayholder to insert the segment */
int counter      /* current value of the counter in the rayholder Qnum*/
);

```

```

double get_last_chunk_new
(
  double **image_array,
  Point Start,
  Point End,
  int Q_num,      /* # of the rayholder to put the rest of the ray */
  int counter    /* current value of the counter in the rayholder Qnum*/
);

double get_last_chunk_time
(
  double **image_array,
  Point Start,
  Point End
);

/***** bilinear_interpolation.c *****/

double bilinear_v /* bilinear 2d interpolation of the velocity*/
(
  double **image_array,
  Point pt      /* image point */
);

double bilinear_dvdx /* bilinear 2d interpolation of the dv/dx */
(
  double **image_array,
  Point pt      /* image point */
);

double bilinear_dvdy /* bilinear 2d interpolation of the dv/dy */
(
  double **image_array,
  Point pt      /* image point */
);

int cyclic /* imposes cyclic boundary condition on a square array indices */
(
  int k,      /* the index (i or j) to be checked or modified */
  int N
);

/***** straight_tracer.c *****/

int trace_straight_ray /* traces straight ray(i,j) into rayholder[0] */
(

```

```

int i,
int j
);

/***** curved_tracer.c *****/

int trace_curved_ray /* traces curved ray(i,j) */
(
int i,
int j
);

/***** trace_ray.c *****/

#define straight 1
#define curved 2

/*
traces straight or curved rays into memory
depending on the "ray_type"
traces only projections from "start_proj" to "end_proj"
*/
int trace_rays
(
int start_proj,
int end_proj,
int ray_type
);

int trace_ray /* traces ray(i,j) depending on its type*/
(
int i,
int j,
int ray_type // can be straight or curved
);

/*****END TRACER.H*****/

/*****BEGIN XRECON.C*****/

/*****
* xrecon.c
* -----
*
* GraphApp front end GUI for Lamb Wave

```

```

*      Tomography reconstructions
*
* Author: Kevin Leonard
*
*
*****/

#include "tracer.h"

int VAR_STATUS = 0;

extern void settexfont(control, font);

void save_and_exit(button b) {

    FILE *fp;
    InitParams configuration_settings;

    configuration_settings.i_frequency = atof(gettext(frequency_field));
    configuration_settings.i_numberofsteps = atoi(gettext(number_of_steps_field));
    configuration_settings.i_stepsize = atof(gettext(step_size_field));

    configuration_settings.i_pfirstpoint = atoi(gettext(parallel_low_gate_limit));
    configuration_settings.i_pnumberofpoints = atoi(gettext(parallel_high_gate_limit));
    configuration_settings.i_cfirstpoint = atoi(gettext(corner_low_gate_limit));
    configuration_settings.i_cnumberofpoints = atoi(gettext(corner_high_gate_limit));

    configuration_settings.i_controlpoint = atoi(gettext(threshold_control_point));

    fp = fopen("xrecon.conf", "wb");

    if(fp) {
        fwrite(&configuration_settings, sizeof(InitParams), 1, fp);
        fclose(fp);
    }
    else {
        askok("Error Saving");
        return;
    }

    exitapp();

}

void update_params(void) {

```



```

FILE *fp;
InitParams configuration_settings;
char dummy_string[20];

fp = fopen("xrecon.conf","rb");
if(fp) {
    fread(&configuration_settings , sizeof(InitParams), 1, fp);
    fclose(fp);

    sprintf(dummy_string, "%f", configuration_settings.i_frequency);
    settext(frequency_field, dummy_string);

    sprintf(dummy_string, "%d", configuration_settings.i_numberofsteps);
    settext(number_of_steps_field, dummy_string);

    sprintf(dummy_string, "%f", configuration_settings.i_stepsize);
    settext(step_size_field, dummy_string);

    sprintf(dummy_string, "%d", configuration_settings.i_pfirstpoint);
    settext(parallel_low_gate_limit, dummy_string);

    sprintf(dummy_string, "%d", configuration_settings.i_pnumberofpoints);
    settext(parallel_high_gate_limit, dummy_string);

    sprintf(dummy_string, "%d", configuration_settings.i_cfirstpoint);
    settext(corner_low_gate_limit, dummy_string);

    sprintf(dummy_string, "%d", configuration_settings.i_cnumberofpoints);
    settext(corner_high_gate_limit, dummy_string);

    sprintf(dummy_string, "%d", configuration_settings.i_controlpoint);
    settext(threshold_control_point, dummy_string);

}
else {
    askok("Error Reading Configuration File");
}

}

void button_released(button b, int buttons, point p) {

    unhighlight(b);
}

```

```

void init_variables(button b) {
    int i;

    TRANS_PER_ARRAY = atoi(gettext(number_of_steps_field));

    DIST_BET_TRANS = atof(gettext(step_size_field));

    NUMBER_OF_POINTS = atoi(gettext(parallel_high_gate_limit));

    FIRST_POINT = atoi(gettext(parallel_low_gate_limit));

    DELAY = FIRST_POINT * SAMPLING_INTERVAL - 21.1; /* us, 21.1 is a
        delay-line time */

    FREQ = (int) (atof(gettext(frequency_field))*1000000); /*MHz, used only in my
        function get_window_start*/

    WAVELENGTH = floor(1.0 * SAMPLING_RATE / FREQ);

    THRESH_CONTROL_POINT = atoi(gettext(threshold_control_point));

    CORNER_POINTS = atoi(gettext(corner_high_gate_limit));

    CORNER_FIRST_POINT = atoi(gettext(corner_low_gate_limit));

    CORNER_DELAY = (CORNER_FIRST_POINT * SAMPLING_INTERVAL -
21.1);

    NUMBER_OF_RAYS = TRANS_PER_ARRAY * TRANS_PER_ARRAY;

    STEPS_IN_J = TRANS_PER_ARRAY - CORNER_STEPS;

    cell_count = (short int **)calloc(TRANS_PER_ARRAY, sizeof(short int *));
    for (i = 0; i < TRANS_PER_ARRAY; i++)
        cell_count[i] = (short int
*)calloc(NUM_PROJECTIONS*TRANS_PER_ARRAY, sizeof(short int));

    curved_ray_length = (double **)calloc(TRANS_PER_ARRAY, sizeof(double
*)));
    for (i = 0; i < TRANS_PER_ARRAY; i++)
        curved_ray_length[i] = (double
*)calloc(NUM_PROJECTIONS*TRANS_PER_ARRAY, sizeof(double));

    askok("Variables have been initialized.");
    VAR_STATUS = 1;
}

```

```

}

void save_dialog(menuitem mi) {

    char *filename = NULL;
    int result = askyesnocancel("Save Changes?");

    if (result == YES) {

        filename = askfilename("Save the file as:", "xrecon.conf");

        if (filename != NULL)
            printf("Saving %s\n", filename);
            askok("The file was succesfully saved.");
    }
}

void get_times_dialogue(button b) {

    int option, data_type, scan_type;
    char filename[100];
    char temp_name[100];

    printf("1\n");
    if (VAR_STATUS == 0) {
        askok("You must initialize variables first!");
        return;
    }
    else {
    printf("2\n");
        sprintf(filename,"%s",askfilename("Select File to Process Times:", ""));

        if (filename != NULL) {

            if ( ischecked(binary_data_type) ) {

                data_type = 0;
            }
            else {

                if ( ischecked(ascii_data_type) ) {

                    data_type = 1;
                }
                else {

```

```

                                askok("You must select either ASCII or Binary data
type!");
                                return;
                                }
                                }

                                if ( ischecked(parallel_scan_type) ) {

                                    scan_type = 0;
                                }
                                else {

                                    if ( ischecked(corner_scan_type) ) {

                                        scan_type = 1;
                                    }
                                    else {
                                        askok("You must select either Corner or Parallel Scan
Geometry");
                                        return;
                                    }
                                }

                                }

                                get_times(filename, scan_type, data_type, THRESH_CONTROL_POINT);
                                option = askyesnocancel("Would you like to scale these times?");
                                if (option == YES) {
                                    if (scan_type == 1)
                                        corner_scale(filename);
                                    else
                                        parallel_scale(filename);
                                    sprintf(temp_name,"%s has been created!",filename);
                                    askok(temp_name);
                                }

                                }
                                }

                                }

void truncate_times_window(button b) {

    int data_type, scan_type;

    if (VAR_STATUS == 0) {
        askok("You must initialize variables first!");
        return;
    }

```

```

    }

    if ( ischecked(binary_data_type) ) {

        data_type = 0;
    }
    else {

        if ( ischecked(ascii_data_type) ) {

            data_type = 1;
        }
        else {
            askok("You must select either ASCII or Binary data type!");
            return;
        }
    }

    if ( ischecked(parallel_scan_type) ) {

        scan_type = 0;
    }
    else {

        if ( ischecked(corner_scan_type) ) {

            scan_type = 1;
        }
        else {
            askok("You must select either Corner or Parallel Scan Geometry");
            return;
        }
    }

    show(W_truncate);
}

void close_truncate_window(button b) {

    show(W);
    hide(W_truncate);
}

void truncate_dialogue(button b) {

```

```

    int scan_type;
    double first, last;
    char *filename;
    char out_name[100], temp_name[100];
    int option;

    if ( ischecked(parallel_scan_type) ) {

        scan_type = 0;
    }
    else {

        scan_type = 1;
    }

    first = atof(gettext(truncate_low_gate));
    last = atof(gettext(truncate_high_gate));

    if (first == 0 || last == 0) {
        askok("You need to put in a gate!");
        return;
    }

    filename = askfilename("Select File to Truncate Times:", "");

    if(filename != NULL) {

        truncatetimes(filename, scan_type, first, last);

    }
    else {
        askok("Error opening file");
        return;
    }

    option = askyesnocancel("Would you like to scale these times?");

    sprintf(out_name, "%s_tr", filename);

    if (option == YES) {

        if (scan_type == 1)
            corner_scale(out_name);
        else
            parallel_scale(out_name);
    }

```

```

    }

    sprintf(temp_name,"%s has been created!",out_name);
    askok(temp_name);
}

void get_recon_dialogue(button b) {

    char *filename;

    if (VAR_STATUS == 0) {
        askok("You must initialize variables first!");
        return;
    }

    filename = askfilename("Select File to Reconstruct","");

    if (filename != NULL) {
        get_reconstruction(filename);
    }
    else {
        askok("Recon Cancelled (Bad Filename or User Cancel)");
    }
}

void build_time_file_dialogue(button b) {

    char temp_string[100];
    char system_string[100];
    char *filename;
    int i;

    for (i = 0; i < NUM_PROJECTIONS; i++) {

        sprintf(temp_string,"Choose projection %d time file:",i+1);
        askok(temp_string);
        filename = askfilename(temp_string,"");
        if (filename == NULL) {
            if (i > 0) {
                system("rm -f times");
            }
            return;
        }
        if (i == 0) {
            sprintf(system_string,"cp %s times",filename);
            system(system_string);
        }
    }
}

```

```

        }
        else {
            sprintf(system_string,"cat %s >> times",filename);
            system(system_string);
        }
    }
    askok("'times' has been created");
}

void scale_dialogue(button b) {

    char *filename;

    if (VAR_STATUS == 0) {
        askok("You must initialize variables first!");
        return;
    }

    filename = askfilename("Select File to Reconstruct","");

    if (filename != NULL) {
        parallel_scale(filename);
    }
    else {
        askok("Recon Cancelled (Bad Filename or User Cancel)");
    }
}

void init_interface(void)
{
    label l;
    button b_gettimes, b_init_vars, b_truncate, b_close;
    button b_build_time_file, b_truncatetimes, b_recon, b_scale;
    menu file_menu;
    menuitem quit_item;
    menuitem line_1;

    W_truncate = newwindow("Truncate Times", rect(200,200,300,150),
                           Titlebar+Resize+Minimize+Maximize);
    setbackground(W_truncate, LightBlue);

    W = newwindow("Lamb Wave Tomography Reconstruction", rect(0,0,700,375),
                  StandardWindow);
    setbackground(W, LightBlue);
}

```



```

file_menu = newmenu("Settings");
line_1 = newmenuitem("-", 0, NULL);
quit_item = newmenuitem("Exit", '0', save_and_exit);

l = newlabel("Lamb Wave Tomography Reconstruction",
            rect(0,4,700,45), Center);
settextfont(l, newfont("Times", Bold, 24));

/***** X-axis controls *****/

l = newlabel("-----",
            rect(0,28,700,15),Center);
settextfont(l,newfont("Times", Bold, 20));

l = newlabel("Frequency: ", rect(10,60,90,20), 0);
settextfont(l, newfont("Times", Bold, 18));
frequency_field = newfield(NULL, rect(110,61,60,20));
l = newlabel("(MHz)",rect(175,60,60,22), AlignLeft);
settextfont(l,newfont("Times", 0, 18));

l = newlabel("# of Steps:", rect(250,60,80,20), 0);
settextfont(l, newfont("Times", Bold, 18));
number_of_steps_field = newfield(NULL, rect(340,61,60,20));

l = newlabel("Step Size: ", rect(420,60,80,20), 0);
settextfont(l, newfont("Times", Bold, 18));
step_size_field = newfield(NULL, rect(505,61,60,20));
l = newlabel("(mm)",rect(570,60,50,22), 0);
settextfont(l,newfont("Times",0,18));

/*
scan_type_group = newradiogroup();
l = newlabel("Scan Geometry:", rect(550,60,150,20),0);
settextfont(l, newfont("Times", Bold, 18));
parallel_scan_type = newradiobutton("Parallel Scan",
rect(575,85,130,20),NULL);
corner_scan_type = newradiobutton("Corner Scan", rect(575,105,130,20),NULL);

data_type_group = newradiogroup();
l = newlabel("Data Type:", rect(550,140,150,20),0);
settextfont(l, newfont("Times", Bold, 18));
binary_data_type = newradiobutton("Binary", rect(575,165,130,20),NULL);
ascii_data_type = newradiobutton("ASCII", rect(575,185,130,20),NULL);
*/

```

```

scan_type_group = newradiogroup();
l = newlabel("Scan Geometry:", rect(370,110,150,20),0);
settextfont(l, newfont("Times", Bold, 18));
parallel_scan_type = newradiobutton("Parallel Scan",
rect(405,135,130,20),NULL);
corner_scan_type = newradiobutton("Corner Scan", rect(405,155,130,20),NULL);

data_type_group = newradiogroup();
l = newlabel("Data Type:", rect(550,110,150,20),0);
settextfont(l, newfont("Times", Bold, 18));
binary_data_type = newradiobutton("Binary", rect(585,135,130,20),NULL);
ascii_data_type = newradiobutton("ASCII", rect(585,155,130,20),NULL);

l = newlabel("Parallel Scan Parameters:", rect(10,110,300,20),0);
settextfont(l, newfont("Times", Bold, 18));

l = newlabel("First Point:", rect(10,140,90,20),0);
settextfont(l, newfont("Times", 0, 18));
parallel_low_gate_limit = newfield(NULL, rect(110,141,60,20));

l = newlabel("# of Points:", rect(186,140,90,20),0);
settextfont(l, newfont("Times", 0, 18));
parallel_high_gate_limit = newfield(NULL, rect(284,141,59,20));

l = newlabel("Corner Scan Parameters:", rect(10,190,200,20),0);
settextfont(l, newfont("Times", Bold, 18));

l = newlabel("First Point:", rect(10,220,90,20),0);
settextfont(l, newfont("Times", 0, 18));
corner_low_gate_limit = newfield(NULL, rect(110,221,60,20));

l = newlabel("# of Points:", rect(186,220,90,20),0);
settextfont(l, newfont("Times", 0, 18));
corner_high_gate_limit = newfield(NULL, rect(284,221,59,20));

l = newlabel("Threshold:", rect(370,220,90,20),0);
settextfont(l, newfont("Times", 0, 18));
threshold_control_point = newfield(NULL, rect(470,221,60,20));

b_init_vars = newbutton("Initialize", rect(585,215,90,30),
                        init_variables);
setBackground(b_init_vars, Green);

l = newlabel("-----"
            "-----",rect(0,265,700,15),Center);
settextfont(l, newfont("Times", Bold, 20));

```

```

b_gettimes = newbutton("Get Times", rect(100,300,80,30),
                        get_times_dialogue);
setBackground(b_gettimes, Green);

b_truncate = newbutton("Truncate", rect(190,300,80,30),
                       truncate_times_window);
setBackground(b_truncate, Green);

b_build_time_file = newbutton("Build Times", rect(280,300,90,30),
                              build_time_file_dialogue);
setBackground(b_build_time_file, Green);

b_recon = newbutton("Reconstruct", rect(470,300,110,30),
                   get_recon_dialogue);
setBackground(b_recon, Red);

b_scale = newbutton("Scale", rect(370,300,80,30),
                   scale_dialogue);
setBackground(b_scale, Green);

addto(W_truncate);

l = newlabel("Low Gate:", rect(80,30,80,20),0);
setTextfont(l, newfont("Times", 0, 18));
truncate_low_gate = newfield(NULL, rect(165,29,60,20));

l = newlabel("High Gate:", rect(80,60,80,20),0);
setTextfont(l, newfont("Times", 0, 18));
truncate_high_gate = newfield(NULL, rect(165,59,60,20));

b_close = newbutton("Close Window", rect(155,110,110,30),
                   close_truncate_window);
setBackground(b_close, Green);

b_truncatetimes = newbutton("Truncate Times", rect(35,110,110,30),
                           truncate_dialogue);
setBackground(b_truncatetimes, Green);

setclose(W, save_and_exit);

show(W);
}

```

```

main ()
{
    init_interface();

    update_params();

    mainloop();
}

/*****END XRECON.C*****/

/*****BEGIN GETTIMES.C*****/

/*****
!!!! reads all waveforms from the initial datafile !!!!!

!!!!MODIFIED for the corner case!!!!
All the functions now have number_of_points
parameter.

E.V. Malyarenko
Modified by Kevin Leonard

empirical function to extract S0 arrival times
from recorded Lamb waveforms.
All scan parameters should be put into the
"full_config.h" file. The behaviour can also
be controlled through several entry points scattered in the code.

IMPORTANT!!! works with highly-filtered and narrow-banded
waveforms. Other types of waves should be analysed, say,
by time-frequency or pattern-matching routines.
*****/

/*****
Pipes
*****/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include "tracer.h"

void parallel_gettimes(char *filename, int data_type, int control_point);

```

```

void corner_gettimes(char filename[], int data_type, int control_point);
void subtract_mean (short int *data, int number_of_points);
void find_maxima(int init, int final);
void rescale_waveform (short int *data, int number_of_points, int scale);
void corner_rescale_waveform (short int *data, int number_of_points, int scale);
void output_test_ray(int row, int number_of_points);
int line_int(int control_point, int number_of_points);
void find_maxima_simple(int number_of_points);
int locate_fastest_mode(int number_of_points, int control_point);
void parallel_set_times (char *in_name, char data_fname[]);
void corner_set_times (char *in_name, char data_fname[]);
//int CORNER_STEPS;

```

```

short int *orig_values;
short int *envelope_values;
int *est_arr_time;
int max1;
double *s1;
double *s2;
double *s3;

```

```

struct point_params
{
    short int value;
    int coord;
};

```

```

void get_times(char filename[], int scan_type, int data_type, int control_point) {
    if (scan_type == 0)
        parallel_gettimes(filename, data_type, control_point);
    else
        corner_gettimes(filename, data_type, control_point);
}

```

//Need to switch from NUMBER_OF_POINTS to number_of_points in common functions

```

void parallel_gettimes (char filename[], int data_type, int control_point)
{
    extern short int *orig_values;
    extern short int *envelope_values;
    extern int *est_arr_time;
    short int *data;
    FILE *in, *pos_env;

```

```

int i,j,m,k,count,max,i_max,Nrays;
int rayN, pointN;
int shift, interpolated_value;
char *times_fname = "parallel_time_simplest";
double est_time;
int number_of_points;

int i1,i2,i0,max2,max0;
extern int max1;
extern double slope,b;
int first_mode_max;

Nrays = NUMBER_OF_RAYS;

number_of_points = NUMBER_OF_POINTS;

data = (short int *)calloc(NUMBER_OF_POINTS, sizeof(short int));
orig_values = (short int *)calloc(NUMBER_OF_POINTS, sizeof(short int));
envelope_values = (short int *)calloc(NUMBER_OF_POINTS, sizeof(short int));
est_arr_time = (int *)calloc(Nrays, sizeof(int));

/***** now the regular program starts *****/

in = fopen(filename, "rb");

pos_env = fopen (times_fname,"wb");

for ( rayN = 1; rayN < (Nrays+1); ++rayN ) {

    for (pointN = 0; pointN < NUMBER_OF_POINTS; ++pointN) {
        if(data_type == 1)
            fscanf (in, "%d", &data[pointN]);
        else
            fread(&data[pointN], sizeof(short int), 1, in);
    }

    /* find and subtract mean value to make zero offset */

    subtract_mean (data, number_of_points);          /* necessary */
    //rescale_waveform (data, NUMBER_OF_POINTS, 600); /* helps a lot */

    /*we compose envelope out of both positive and negative values to increase number of
    points available for interpolation. On the other hand, this may degrade quality of the
    resulting envelope */

```

```

for (pointN=0; pointN<NUMBER_OF_POINTS; ++pointN) {
    if(data[pointN] < 0) orig_values[pointN] = 0; //-data[pointN];
    else orig_values[pointN] = data[pointN];
}
for (i = 0; i < NUMBER_OF_POINTS; ++i) envelope_values[i] = 0;

find_maxima(1, NUMBER_OF_POINTS);

first_mode_max = locate_fastest_mode(number_of_points, control_point);
if(first_mode_max == 0)
    goto failure;      /* use previous time instead */

/*****
**/
/* now we'll repeat the same with the real envelope and first_mode_max */
/*
for (pointN = 0; pointN < NUMBER_OF_POINTS; ++pointN) {
    if(data[pointN] < 0) orig_values[pointN] = 0;
    else orig_values[pointN] = data[pointN];
}
for (i = 0; i < NUMBER_OF_POINTS; ++i) envelope_values[i] = 0;

find_maxima(1,first_mode_max);

*/
/**
    now we find the abscissa of the envelope point with given value,
    interpolation is done using first envelope points. The error is
    smaller if we interpolate between existing points
***/

interpolated_value = line_int(control_point, number_of_points);

if (interpolated_value != 0.0)
    est_arr_time[rayN-1] = interpolated_value;
else {
failure:
    est_arr_time[rayN-1] = est_arr_time[rayN-2];
}

est_time=est_arr_time[rayN-1] * SAMPLING_INTERVAL + DELAY;
fprintf(pos_env,"%f\n",est_time);

printf( "%d   time= %d \n", rayN, est_arr_time[rayN-1]);

```

```

//output_test_ray(rayN); //the output function helps visualize envelopes

}/*end loop over rayN*/

fclose(pos_env);
fclose(in);

parallel_set_times (times_fname, filename); /* rearrange times */

free(data);
free(orig_values);
free(envelope_values);
free(est_arr_time);

}

void corner_gettimes (char filename[], int data_type, int control_point)
{
extern short int *orig_values;
extern short int *envelope_values;
extern int *est_arr_time;
short int *data;
FILE *in, *pos_env;
int i,j,m,k,count,max,i_max,Nrays;
int rayN, pointN;
int shift, interpolated_value;
int interp_v1, interp_v2;
char *times_fname = "corner_time_simplest";
double est_time;
int number_of_points;

int theor_value;
double ray_length;

int i1,i2,i0,max2,max0;
extern int max1;
extern double slope,b;
int first_mode_max;

//int stop_steps;
//stop_steps = TRANS_PER_ARRAY - (TRANS_PER_ARRAY*DIST_BET_TRANS-
STOP_DIST)
//
//          /DIST_BET_TRANS;
//if(stop_steps%2 == 0)
// CORNER_STEPS = stop_steps;
//else

```



```

// CORNER_STEPS = stop_steps - 1;

Nrays = NUMBER_OF_RAYS - SQUARE(CORNER_STEPS); /* corner effect */

number_of_points = CORNER_POINTS; /*number of points in the corner trace*/

data = (short int *)calloc(number_of_points, sizeof(short int));
orig_values = (short int *)calloc(number_of_points, sizeof(short int));
envelope_values = (short int *)calloc(number_of_points, sizeof(short int));
est_arr_time = (int *)calloc(Nrays, sizeof(int));

/***** now the regular program starts *****/

in = fopen(filename, "rb");

pos_env = fopen (times_fname,"wb");

for ( rayN = 1; rayN < (Nrays+1); ++rayN ) {

    for (pointN = 0; pointN < number_of_points; ++pointN) {
        if (data_type == 0)
            fread(&data[pointN],sizeof(short int),1,in);
        else
            fscanf (in, "%d", &data[pointN]);
    }

    /* find and subtract mean value to make zero offset */

    subtract_mean (data, number_of_points);
    //corner_rescale_waveform (data, number_of_points, 500);

    /*we compose envelope out of both positive and negative values to increase number of
    points available for interpolation. On the other hand, this may degrade quality of the
    resulting envelope */

    for (pointN=0; pointN<number_of_points; ++pointN) {
        if(data[pointN] < 0) orig_values[pointN] = 0; //-data[pointN];
        else orig_values[pointN] = data[pointN];
    }
    for (i = 0; i < number_of_points; ++i) envelope_values[i] = 0;

    find_maxima(1, number_of_points);

    first_mode_max = locate_fastest_mode(number_of_points, control_point);
    if(first_mode_max == 0)

```

```

goto failure;      /* use previous time instead */

/*****
**/
/* now we'll repeat the same with the real envelope and first_mode_max */

/* for (pointN = 0; pointN < number_of_points; ++pointN) {
   if(data[pointN] < 0) orig_values[pointN] = 0;
   else orig_values[pointN] = data[pointN];
}
for (i = 0; i < number_of_points; ++i) envelope_values[i] = 0;

find_maxima(1,first_mode_max);
*/

/**
    now we find the abscissa of the envelope point with given value,
    interpolation is done using first envelope points. The error is
    smaller if we interpolate between existing points
***/

interpolated_value = line_int(control_point, number_of_points);

/*lower the threshold if detects the next mode */
//if((interpolated_value - est_arr_time[rayN-2]) > 3*WAVELENGTH)
//interpolated_value = line_int(50, number_of_points);

/***** determine raylength from the filename in three raws below *****/
/* j=(rayN-1) % TRANS_PER_ARRAY;
   i=((rayN-1)-j) / TRANS_PER_ARRAY;
   if(i%2 != 0)
       j = TRANS_PER_ARRAY - 1 - j;
   ray_length = DIST_BET_TRANS * sqrt(1.0*SQUARE(TRANS_PER_ARRAY-i-0.5)
+ 1.0*SQUARE(j+0.5));

   theor_value = (ray_length / EST_VELOCITY - CORNER_DELAY) /
SAMPLING_INTERVAL;

   if((interpolated_value - theor_value) > 3*WAVELENGTH)
       interpolated_value = line_int(80, number_of_points);

   if((interpolated_value - theor_value) > 3*WAVELENGTH)
       interpolated_value = line_int(60, number_of_points);
*/

```

```

if (interpolated_value != 0.0)
    est_arr_time[rayN-1] = interpolated_value;
else {
failure:
    est_arr_time[rayN-1] = est_arr_time[rayN-2];
}

est_time = est_arr_time[rayN-1] * SAMPLING_INTERVAL + CORNER_DELAY;
fprintf(pos_env,"%f\n",est_time);

printf( "%d    time= %d \n", rayN, est_arr_time[rayN-1]);

//output_test_ray(rayN, number_of_points); //the output function helps visualize
envelopes

}/*end loop over rayN*/

fclose(pos_env);
fclose(in);

corner_set_times (times_fname, filename); /* rearrange times */

free(data);
free(orig_values);
free(envelope_values);
free(est_arr_time);
}

void subtract_mean (short int *data, int number_of_points)
{
    int i;
    double sum, aver;
    short int mean;

    sum = 0;

    for(i = 0; i < number_of_points; ++i)
        sum += 1.0 * data[i];

```

```

aver = sum / number_of_points;
mean = (short int) floor(aver);

for(i = 0; i < number_of_points; ++i)
    data[i] -= (short int) aver;

} //end subtract_mean

/* rescales positive part of the waveform
*/
void corner_rescale_waveform (short int *data, int number_of_points, int scale)
{
    int i;
    int max = 10000;
    double factor;

    for(i = 0; i < number_of_points; ++i) {
        if(data[i] > max)
            max = data[i];
    }

    factor = 1.0 * scale / max;

    for(i = 0; i < number_of_points; ++i)
        data[i] = (short int) (data[i] * factor);

} //end rescale_waveform

/**
    function that finds maxima and
    determines the envelope
    I T E R A T I V E version
    ***/

void find_maxima(int init, int final)
{
    extern short int *envelope_values;
    extern short int *orig_values;
    int i,max,i_max;

    while(init > 0) {

        max = 0;

```

```

for(i = init; i < final; ++i) { //find absolute max betw. init and final
  if(orig_values[i] > max) {
    max = orig_values[i];
    i_max = i;
    if(max >= DAC_RESOLUTION / 2) { /*it may be bigger due to mean shift*/
      //printf("First truncated value: max = %d\n",max);
      break; /* found first truncated value, no need to go further*/
    }
  }
}
envelope_values[i_max] = max;

/****
  it may happen that there is no positive maximum, (i.e. between
  different modes). In this case we
  simulate it stepping back one wavelength from the real maximum
  *****/

if(max == 0)
  i_max -= (int) WAVELENGTH; //step 1 wavelength from the preceding maximum
*/

i = i_max;

/* now we step 1.5 wavelengths back and look for the preceding maximum */
while( (orig_values[i] > 0) && (i > 0) )
  --i;

final = i;
if( final > WAVELENGTH )
  init = final - (int) WAVELENGTH;    // go back one wavelength
else
  break; /* cannot step further back */

} /*end big while */

//if(max == 0)
//printf("find_maxima: max = 0 \n");

} /* end find_maxima */

/*function that finds maxima */

```

```

void find_maxima_simple(int number_of_points) {

    extern short int *envelope_values;
    extern short int *orig_values;

    int i,max;

    max=0;
    for(i = 0; i < number_of_points; ++i) {
        if(orig_values[i]>max) {
            max=orig_values[i];
            envelope_values[i] = max;
        }
    }

}/* end find_maxima_simple */

int locate_fastest_mode(int number_of_points, int control_point) {

    extern short int *envelope_values;
    struct point_params env[800]; //temporary array to store envelope points
    int i,k;
    int max_k,start = 0,finish,max_i;

    for(i = 0; i < 800; i++) { //initializing structure each time
        env[i].value = 0;
        env[i].coord = 0;
    }

    k=0;
    for(i = 0; i < number_of_points; i++) { //fill struct with envelope points
        if(envelope_values[i] != 0) {
            env[k].value = envelope_values[i];
            env[k].coord = i;
            k++;
        }
    }
    max_k = k; //the number of points in the structure (we hope it is < 100)

    for(i = 0; i < max_k; i++) { //finding maximum of the first coming mode

        if( (env[i].value < env[i+1].value) &&

```

```

        (env[i+1].value >= control_point) && /**** !!! CONTROL POINT !!!!
                                     ***/
        (env[i+1].value < env[i+2].value) &&
        (env[i+2].value < env[i+3].value))
    {
        start = i;
        goto found;
    }
}

found:
i=start;
while( (env[i].value < env[i+1].value) )
    i++;

max_i = i; /* this should be the maximum of the first packet */

//printf("max_i=%d\n",env[max_i].coord);

return(env[max_i].coord); /* returns position of the first mode's maximum */

} /* end locate fastest mode */

**** good function for debugging (to plot rays and envelopes)****/
void output_test_ray(int row, int number_of_points) {

    extern short int *envelope_values;
    FILE *fp1;
    int i;
    char env_name[25];

    sprintf(env_name, "/TEMP_DATA/envelope%d", row);
    //printf ("%s\n", env_name);

    fp1 = fopen (env_name, "wb");
    for(i=0; i<number_of_points; i++) {
        fprintf(fp1, "%d\n", envelope_values[i]);
    }
    fclose(fp1);
}

```

```

/*straight line interpolation between envelope points
supplied is value p, output is abscissa of p */

int line_int(int control_point, int number_of_points) {

extern short int *envelope_values;
double b, slope;
int i,j,i1,i2,max1,max2;
int i_max, i_min, max, min;

int p = control_point;

/*first we have to find those two points in the envelope that we need
they either embrace p or not (short envelope).
but if there is only one point in the envelope or points belong to slower
modes, that is not this function's fault, envelope should be cleaned earlier */

max = 0; //now we'll find global max and min
min = 60000;
for(i = 0; i < number_of_points; i++) {
  if(envelope_values[i] > max) {
    i_max = i;
    max = envelope_values[i];
  }
  if( (envelope_values[i]<min) && (envelope_values[i] != 0) ) {
    i_min = i;
    min=envelope_values[i];
  }
}

/* for(i=0; i<number_of_points; i++) {
  if(envelope_values[i] = p) { return(i); }
}
*/

if( (p < max) && (p > min) ) { //if p is embraced
  i=0;
  while(envelope_values[i] < p) i++;
  i2 = i;
  max2 = envelope_values[i2];
}
}

```



```

i--;
while ( (envelope_values[i] == 0) && (i!=0) )
    i--;

i1 = i;
max1 = envelope_values[i1];
}
else if (p < min) { //if p< envelope's min use first 2 points of envelope

    i1 = i_min;
    max1 = min;

    i = i1 + 10;
    while ( (envelope_values[i] == 0) && (i != 4995))
        i++;

    i2 = i;
    max2 = envelope_values[i2];
}

else { //in this case p>=max, use 2 last points of the envelope

    i2 = i_max;
    max2 = max;

    i = i2 - 10;
    while ((envelope_values[i] == 0) && (i != 0) )
        i--;

    i1 = i;
    max1 = envelope_values[i1];
}

slope = 1.0 * (max2 - max1) / (i2 - i1);
b = 1.0 * (max1*i2 - max2*i1) / (i2-i1);

j = (int) floor( (p - b) / slope ); // we find the intersection coordinate

if(j < 0)
    j = 0;
if(j > number_of_points - 1)
    j = number_of_points - 1;

envelope_values[j] = p;

```

```

return(j);

} /* end line_int.....*/

/*
calculates length of the corner ray (i,j)
*/
double corner_ray_length (int i, int j) {

double r_l;
int N = TRANS_PER_ARRAY;

r_l = DIST_BET_TRANS * sqrt(1.0*SQUARE(N-i-0.5) + 1.0*SQUARE(j+0.5));

return r_l;
}

/* This is a function to read in the file of times found by
the time_finding routine
and reverse the order of every other set of TRANS_PER_ARRAY,
so the order will match that expected by the code
the output filename is the same as for the original projection file
plus the word "time" at the and.
This file is to be truncated and scaled and is almost ready for
the reconstruction.
*/

void corner_set_times (char *times_fname, char data_fname[])
{
int i, j, l, length;
double fvalue;//, *s1, *s2, *s3;
char *out_name;
char temp_name[100];
FILE *in, *out;
int corner_length, actual_num_of_rays;
int full_length_rays;

length = TRANS_PER_ARRAY;
corner_length = STEPS_IN_J;

```

```

actual_num_of_rays = NUMBER_OF_RAYS - SQUARE(CORNER_STEPS);
full_length_rays = (TRANS_PER_ARRAY -
CORNER_STEPS)*TRANS_PER_ARRAY;

s1 = (double *)calloc(actual_num_of_rays, sizeof(double));
s2 = (double *)calloc(NUMBER_OF_RAYS, sizeof(double));
s3 = (double *)calloc(NUMBER_OF_RAYS, sizeof(double));

sprintf(temp_name, "%s_time", data_fname);
out_name = temp_name;

if ( (in = fopen (times_fname, "r")) == NULL )
    printf ("Can't open %s for reading.\n", times_fname);
else {
    printf ("reading in the data\n");

    for ( i = 0; i < actual_num_of_rays; ++i ) {
        fscanf (in, "%lf", &fvalue);
        s1[i] = fvalue;
    }
    printf ("switching the data\n");

    /* first rearrange full rays as before, BUT j starts from the corner */

    for ( j = 0; j < (full_length_rays - 2*length + 1); j += 2*length ) {
        for ( i = j; i < (j + length); i++ )
            s2[i] = s1[j + (j+(length-1) - i)];
    }
    for ( j = length; j < (full_length_rays - length + 1) ; j += 2*length ) {
        for ( i = j; i < (j + length); i++ )
            s2[i] = s1[i];
    }

    /****** done with full-length rays, start arranging short ones *****/

    for ( j = full_length_rays; j < (actual_num_of_rays - 2*corner_length + 1); j +=
2*corner_length ) {
        for ( i = j; i < (j + corner_length); i++ )
            s2[i] = s1[j + (j+(corner_length-1) - i)];
    }
    for ( j = full_length_rays+corner_length; j < (actual_num_of_rays - corner_length + 1)
; j += 2*corner_length ) {
        for ( i = j; i < (j + corner_length); i++ )
            s2[i] = s1[i];
    }
}

```

```

/***** end rearranging real data *****/

/*****
    now fill gaps for corner rays with theoretical times
    theor_time = ray_length / ESTIMATED_VELOCITY
*****/

for(l = 0; l < full_length_rays; l++) //first copy times for usual rays
    s3[l] = s2[l];
for(l = full_length_rays; l < NUMBER_OF_RAYS; l++) {
    j = 1 % TRANS_PER_ARRAY;
    i = (l - j) / TRANS_PER_ARRAY;
    if(j < CORNER_STEPS)
        s3[l] = corner_ray_length(i,j) / EST_VELOCITY; //!!!! CONTROL !!!!
    else
        s3[l] = s2[ l - (i - STEPS_IN_J + 1)*CORNER_STEPS ];
}

if ( (out = fopen (out_name, "w")) == NULL )
    printf ("Can't open %s for writing.\n", out_name);
else
    {
        printf ("outputing the data\n");

        for ( i = 0; i < NUMBER_OF_RAYS; ++i )
            {
                fprintf (out, "%f\n", s3[i]);
            }
    }
}

strcpy(data_fname, out_name);
fclose(in);
fclose(out);

free(s1);
free(s2);
free(s3);

}

/* rescales positive part of the waveform
*/
void rescale_waveform (short int *data, int number_of_points, int scale)
{

```

```

int i;
int max = 0;
double factor;

for(i = 0; i < number_of_points; ++i) {
    if(data[i] > max)
        max = data[i];
}

factor = 1.0 * scale / max;

for(i = 0; i < number_of_points; ++i)
    data[i] = (short int) data[i] * factor;

} //end rescale_waveform

void parallel_set_times (char *times_fname, char data_fname[])
{
    int i, j, k, l, length;
    char* out_name;
    char temp_name[100];
    FILE *in, *out;

    length = TRANS_PER_ARRAY;

    s1 = (double *)calloc(NUMBER_OF_RAYS, sizeof(double)); //memory for s1,2
    s2 = (double *)calloc(NUMBER_OF_RAYS, sizeof(double));

    sprintf(temp_name, "%s_time", data_fname);
    out_name = temp_name;

    if ( (in = fopen (times_fname, "r")) == NULL )
        printf ("Can't open %s for reading.\n", times_fname);
    else {
        printf ("reading in the data\n");

        for ( i = 0; i < NUMBER_OF_RAYS; ++i ) {
            fscanf(in, "%lf", &s1[i]);
            //s1[i] = fvalue;
        }
        printf ("switching the data\n");

        for ( j = 0; j < (NUMBER_OF_RAYS - 2*length + 1); j += 2*length ) {
            for ( l = j; l < (j + length); l++) s2[l] = s1[j + (j+(length-1) - l)];
        }
    }
}

```

```

for ( k = length; k < (NUMBER_OF_RAYS - length + 1) ; k += 2*length ) {
    for ( i = k; i < (k + length); i++ ) s2[i] = s1[i];
}

if ( (out = fopen (temp_name, "w")) == NULL )
    printf ("Can't open %s for writing.\n", temp_name);
else
    {
        printf ("outputing the data\n");

        for ( i = 0; i < NUMBER_OF_RAYS; ++i )
            {
                fprintf (out, "%f\n", s2[i]);
            }
    }
}
strcpy(data_fname, out_name);
fclose(in);
fclose(out);
printf("here\n");

free(s1);
free(s2);
}

```

/******END GETTIMES.C******/

/******BEGIN TRACE_RAYS.C******/

/***** Eugene Malyarenko, June, 2000
Kevin Leonard May, 2002

Updated for projection 1 to be adapted for pipe scans
where rays wrap around the pipe.

EV COMMENTS-----

Traces both
!!!!!!!!!! straight and curved !!!!!!!!!!!
rays depending on the parameter
"ray_type".

Traces rays into rayholder for subsequent
insertion into memory. Indices (i,j) have different meanings
for rays belonging to different projections.

i is always in the range (0,N) while j increases
with projection number:

There are total 6 projections ($6*N*N$ rays). We assume rays going only in one direction. Flipping the direction physically changes nothing but the number of rays doubles and the measurement errors could be reduced. The acquisition time also doubles.

1. Parallel verticals: i and j start from upper left and right corners
2. Parallel horizontals: i and j start from left bottom and top corners
3. LOW LEFT CORNER: i starts from top left, j from bottom left
4. LOW RIGHT CORNER: i starts from bottom left, j from bottom right
5. TOP RIGHT CORNER: i starts from bottom right, j from top right
6. TOP LEFT CORNER: i starts from top right, j from top left

Actual scanning **MUST ALWAYS** be performed in the same order unless someone decides to change all the programs.

*****/

```
#include "tracer.h"
```

```
/*
traces straight or curved rays into memory
depending on the "ray_type"
traces only projections from "start_proj" to "end_proj"
*/
```

```
int trace_rays
```

```
(
int start_proj,
int end_proj,
int ray_type
)
```

```
{
int N = TRANS_PER_ARRAY;
int i, j, k;
```

```
/****** initialize external arrays to zero before tracing *****/
for(i = 0; i < N; i++) {
for(j = 0; j < NUM_PROJECTIONS*N; j++) {
cell_count[i][j] = 0; /* initialize segment counters */
curved_ray_length[i][j] = 0.0; /* initialize ray lengths */
}
}
```

```

}

/* trace rays for specified projections into memory */

for(k = start_proj; k <= end_proj; k++) {
  printf("projection #%d\n", k);
  for (i = 0; i < N; ++i)
    for (j = (k-1)*N; j < k*N; ++j)
      trace_ray (i,j, ray_type); /* trace straight or curved rays */
}

} /* end trace_rays */

/*
  Traces the ray(i,j) into rayholder[0]
  */

int trace_ray ( int i, int j, int ray_type )
{
  double a = DIST_BET_TRANS;
  int N = TRANS_PER_ARRAY;
  int counter;
  Point Start, End; /* ray endpoints*/
  int Q_num; /* rayholder number */
  int trace_type = ray_type;

  counter = 0; /* initialize counter */

  /* in the 1 projection i and j start from upper left and right corners */

  if(j < N) { /* 1 projection, 0 <= j < N */
    Start.x = 0.0;
    Start.y = a*(N-i-0.5);
    End.x = N*a;
    End.y = a*(N-j-0.5);

    // Added May, 2002 *****
    // Makes sure ray is never longer than halfway around pipe

    if ( ( j - i ) > ( N/2 ) ) {

      Start.y = Start.y - (N*a);

```



```

}
else {

    if ( (i - j) > (N/2) ) {

        End.y = End.y - (N*a);

    }

}
//*****

}

/* in the 2 projection i and j start from left bottom and top corners */

else if(j < 2*N) {          /* 2 projection, N <= j < 2N */
    Start.x = a*(i+0.5);
    Start.y = 0.0;
    End.x = a*(j+0.5-N);
    End.y = N*a;
}

/* 3. LOW LEFT CORNER: i starts from top left, j from bottom left */

else if(j < 3*N) {          /* 3 projection, 2N <= j < 3N */
    Start.x = 0.0;
    Start.y = a*(N-i-0.5);
    End.x = a*(j+0.5-2*N);
    End.y = 0.0;
    if((i >= 95) && (j < 2*N+5))
        trace_type = straight;    //in the very corner trace straight
}

/* 4. LOW RIGHT CORNER: i starts from bottom left, j from bottom right */

else if(j < 4*N) {          /* 4 projection, 3N <= j < 4N */
    Start.x = a*(i+0.5);
    Start.y = 0.0;
    End.x = N*a;
    End.y = a*(j+0.5-3*N);
    if((i >= 95) && (j < 3*N+5))
        trace_type = straight;    //in the very corner trace straight
}

```

```

/* 5. TOP RIGHT CORNER: i starts from bottom right, j from top right */

else if(j < 5*N) {          /* 5 projection, 4N <= j < 5N */
    Start.x = N*a;
    Start.y = a*(i+0.5);
    End.x = a*(5*N-j-0.5);
    End.y = N*a;
    if((i >= 95) && (j < 4*N+5))
        trace_type = straight;    //in the very corner trace straight
}

/* 6. TOP LEFT CORNER: i starts from top right, j from top left */

else if(j < 6*N) {          /* 6 projection, 5N <= j < 6N */
    Start.x = a*(N-i-0.5);
    Start.y = N*a;
    End.x = 0.0;
    End.y = a*(6*N-j-0.5);
    if((i >= 95) && (j < 5*N+5))
        trace_type = straight;    //in the very corner trace straight
}

/***** knowing start and end, trace the ray depending on its type *****/

switch(trace_type) {

case straight:
    Q_num = 0;
    counter = trace_line_segment(Start, End, Q_num, counter);
    if(counter <= 3) /* some bug when inserting short rays */
        Ray_Holder[Q_num][counter].last = true; /*mark it for subsequent search*/
    else
        Ray_Holder[Q_num][counter-1].last = true;

    //print_rayholder(Q_num);
    insert_ray_information(i,j,Q_num); /* segment pointers must be freed */
    //printf("trace_ray: ray(%d, %d) Start.y: %f End.y: %f ...done\n",i,j,Start.y, End.y);
    break;

case curved:          /*!!!! trace contrast image !!!!!*/
    Q_num = golden_shooting(contrast_image, Start, End, j);
    //print_rayholder(Q_num);
    insert_ray_information(i,j,Q_num); /* segment pointers must be freed */
    printf("trace_ray: ray(%d, %d)\n",i,j);
    break;

default:

```

```
printf("ray type should be either ``straight" or ``curved"\n");
exit(1);

} //end switch;

return (0);
}

/*****END TRACE_RAY.C*****/
```

References

- [1] Viktorov, I.A., *Rayleigh and Lamb Waves -- Physical Theory and Applications*. 1967, New York: Plenum.
- [2] Rose, J.L., *Ultrasonic Waves in Solid Media*. 1999, Cambridge: Cambridge University Press.
- [3] Rose, J.L., *A Baseline and Vision of Ultrasonic Guided Wave Inspection Potential*. *Journal of Pressure Vessel Technology*, 2002. **124**: p. 273-282.
- [4] Kak, A.C. and M. Slaney, *Principles of Computerized Tomographic Imaging*. 1988, New York: IEEE.
- [5] McKeon, J.C.P., *Tomography Applied to Lamb Wave Contact Scanning*, in *Dept. of Applied Science*. 1998, College of William and Mary: Williamsburg.
- [6] Malyarenko, E.V., *Lamb Wave Diffraction Tomography*, in *Dept. of Applied Science*. 2000, College of William and Mary: Williamsburg, VA.
- [7] Hinders, M.K. and J.C.P. McKeon. *Lamb Wave Tomography for Corrosion Mapping*. in *Proceedings of the 2nd Joint NASA/FAA/DoD Conference on Aging Aircraft*. 1999. p. 732-740
- [8] Hinders, M.K., E.V. Malyarenko, and J.C.P. McKeon, *Contact scanning Lamb wave tomography*. *J. Acoust. Soc. Am.*, 1998. **104**: p. 1790(A).
- [9] McKeon, J.C.P. and M.K. Hinders, *Parallel projection and crosshole contact scanning Lamb wave tomography*. *J. Acoust. Soc. Am.*, 1999. **106**: p. 2568-2577.
- [10] McKeon, J.C.P. and M.K. Hinders. *Lamb Wave Contact Scanning Tomography*. in *Review of Progress in QNDE*. 1999: Plenum, New York. p. 951-958
- [11] Hinders, M.K., E.V. Malyarenko, and J.C.P. McKeon. *Ultrasonic Lamb Wave Tomographic Scanning*. in *Proceedings of SPIE*. 1999. p. 279-291

- [12] Malyarenko, E.V. and M.K. Hinders, *Fan beam and double crosshole Lamb wave tomography for mapping flaws in aging aircraft structures*. J. Acoust. Soc. Am., 2000. **108**(4): p. 1631-1639.
- [13] Hinders, M.K., K.R. Leonard, and E.V. Malyarenko. *Blind test of Lamb wave diffraction tomography*. in *Review of Progress in QNDE*. 2000. Melville, NY: American Institute of Physics. p. 173-180
- [14] Malyarenko, E.V. and M.K. Hinders, *Ultrasonic Lamb wave diffraction tomography*. Ultrasonics, 2001. **39**(4): p. 269-281.
- [15] Malyarenko, E.V. and M.K. Hinders. *Comparison of Double Crosshole and Fanbeam Lamb Wave Tomography*. in *Review of Progress in QNDE*. 2001: Plenum. p. 732-739
- [16] Malyarenko, E.V., J.S. Heyman, and M.K. Hinders. *Lamb Wave Tomography for Monitoring Aircraft Structural Integrity*. in *USAF Aircraft Structural Integrity Program Conference Proceedings*. 2001. p.
- [17] Leonard, K.R., E.V. Malyarenko, and M.K. Hinders, *Ultrasonic Lamb wave tomography*. Inverse Probl., 2002. **18**: p. 1795-1808.
- [18] Leonard, K.R. and M.K. Hinders, *Guided wave helical ultrasonic tomography of pipes*. J. Acoust. Soc. Am., 2003. **114**(2): p. 767-774.
- [19] Bregman, N.D., R.C. Baily, and C.H. Chapman, *Crosshole seismic tomography*. Geophysics, 1989. **54**(2): p. 200-215.
- [20] Pratt, R.G. and N.R. Goult, *Combining wave-equation imaging with travelttime tomography to form high resolution images from crosshole data*. Geophysics, 1991. **56**(2): p. 208-224.
- [21] Graff, K., *Wave Motion in Elastic Solids*. 1975, New York: Dover Publications, Inc.
- [22] Kolsky, H., *Stress Waves in Solids*. 1963, New York: Dover Publications, Inc.

- [23] Achenbach, J.D., *Wave Propagation in Elastic Solids*. 1973, Amsterdam: North-Holland.
- [24] Auld, B.A., *Acoustic Fields and Waves in Solids*. 2nd ed. Vol. II. 1990, Florida: Krieger Publishing Company.
- [25] Wright, W., et al., *Air-coupled Lamb wave tomography*. IEEE Trans. Ultrason. Ferroelectr. Freq. Contr., 1997. **44**(1): p. 53-59.
- [26] Kenderian, S., D. B.B., and R.E. Green Jr., *Laser Based and Air Coupled Ultrasound as Noncontact and Remote Techniques for Testing of Railroad Tracks*. Materials Evaluation, 2002. **60**(1): p. 65-70.
- [27] Scruby, C.B., et al., *Quantitative studies of thermally generated elastic waves in laser-irradiated metals*. J. Appl. Phys., 1980. **51**(12): p. 6210.
- [28] Bray, D.E. and R.K. Stanley, *Nondestructive Evaluation: A Tool in Design, Manufacturing, and Service*. 1997, New York: CRC Press.
- [29] Wilcox, P.D., M.J.S. Lowe, and P. Cawley, *Mode and Transducer Selection for Long Range Lamb Wave Inspection*. J. Intellig. Mat. Sys. Struct., 2001. **12**(8).
- [30] Alleyne, D.N. and P. Cawley, *Optimization of Lamb wave inspection techniques*. NDT&E International, 1992. **25**(1): p. 11-19.
- [31] Ditri, J.J. and K.M. Rajana, *Analysis of the Wedge Method of Generating Guided Waves*. Rev. Prog. in QNDE, 1995. **14**: p. 163-170.
- [32] Rajana, K.M., D.D. Hongerholt, and J.L. Rose, *Analysis of the Generation of Guided Waves using Finite Sources: An Experimental Approach*. Rev. Prog. in QNDE, 1995. **14**: p. 171-178.
- [33] Rose, J.L., D. Jiao, and J. Spanner Jr., *Ultrasonic Guided Wave NDE for Piping*. Materials Evaluation, 1996. **54**(11): p. 1310-1313.
- [34] Hay, T.R. and J.L. Rose, *Guided Wave Testing Optimization*. Materials Evaluation, 2002. **60**(10): p. 1239-1244.

- [35] Quarry, M. and J.L. Rose, *Multimode guided wave inspection of piping using Comb transducers*. Material Evaluation, 1999. **57**(10): p. 1089-1090.
- [36] Sun, Z., et al., *Flexural Mode Tuning in Pipe Inspection*. Rev. Prog. in QNDE, 2002. **21**: p. 262-269.
- [37] Rose, J.L., et al., *Guided Wave Flexural Mode Tuning and Focusing for Pipe Testing*. Materials Evaluation, 2003. **61**(2): p. 162-167.
- [38] Wilcox, P., et al., *An Example of the Use of Interdigital PVDF Transducers to Generate and Receive a High Order Lamb Wave Mode in a Pipe*. Rev. Prog. in QNDE, 1997. **16**: p. 919-926.
- [39] Wilcox, P., M. Lowe, and P. Cawley, *The effect of dispersion on long-range inspection using ultrasonic guided waves*. NDT&E International, 2001. **34**(1): p. 1-9.
- [40] Wilcox, P.D., M.J.S. Lowe, and P. Cawley, *A Signal Processing Technique to Remove the Effect of Dispersion From Guided Wave Signals*. Rev. Prog. in QNDE, 2001. **20**: p. 555-562.
- [41] Sicard, R., J. Goyette, and D. Zellouf, *A numerical dispersion compensation technique for time recompression of Lamb wave signals*. Ultrasonics, 2002. **40**: p. 727-732.
- [42] Alleyne, D.N., T.P. Pialucha, and P. Cawley, *A signal regeneration technique for long-range propagation of dispersive Lamb waves*. Ultrasonics, 1993. **31**(3): p. 201-204.
- [43] Zhu, W., et al., *Ultrasonic guided wave NDT for hidden corrosion detection*. Res. Nondestr. Eval., 1998. **10**: p. 205-225.
- [44] Jenot, F., et al., *Corrosion thickness gauging in plates using Lamb wave group velocity measurements*. Meas. Sci. Technol., 2001. **12**(8): p. 1287-1293.
- [45] Sun, K.J. and P.H. Johnston, *Effect of Rivet Rows on Propagation of Lamb Waves in Mechanically Fastened Two-Layer Aluminum Plates*. Rev. Prog. in QNDE, 1995. **14**: p. 1569-1576.

- [46] Sun, K.J. and P.H. Johnston, *Disbond Detection in Bonded Aluminum Joints Using Lamb Wave Amplitude and Time-of-flight*. Rev. Prog. in QNDE, 1994. **13**: p. 1507-1513.
- [47] Alleyne, D. and P. Cawley, *The Long Range Detection of Corrosion in Pipes Using Lamb Waves*. Rev. Prog. in QNDE, 1995. **14**: p. 2073-2080.
- [48] Silva, M.Z., R. Gouyon, and F. Lepoutre, *Hidden corrosion detection in aircraft aluminum structures using laser ultrasonics and wavelet transform signal analysis*. Ultrasonics, 2003. **41**(4): p. 301-305.
- [49] Cho, Y., D.D. Hongerholt, and J.L. Rose, *Lamb Wave Scattering Analysis for Reflector Characterization*. IEEE Trans. Ultrason., Ferroelectr., and Freq. Contr., 1997. **44**(1): p. 44-52.
- [50] Zhao, X. and J.L. Rose, *Boundary element modeling for defect characterization potential in a wave guide*. Int. J. Solids Struct., 2003. **40**(11): p. 2645-2658.
- [51] Chang, Z. and A. Mal, *Scattering of Lamb waves from a rivet hole with edge cracks*. Mechanics of Materials, 1999. **31**(3): p. 197-204.
- [52] Verdict, G.S., P.H. Gien, and C.P. Burger, *Finite Element Study of Lamb Wave Interactions with Holes and Through Thickness Defects in Thin Metal Plates*. Rev. Prog. in QNDE, 1992. **11**: p. 97-104.
- [53] Alleyne, D. and P. Cawley, *The Interaction of Lamb Waves with Defects*. IEEE Trans. Ultrason., Ferroelectr., and Freq. Contr., 1992. **39**(3): p. 381-396.
- [54] Lowe, M., D. Alleyne, and P. Cawley, *Mode Conversion of Guided Waves by Defects in Pipes*. Rev. Prog. in QNDE, 1997. **16**: p. 1261-1268.
- [55] Lowe, M.J.S., D.N. Alleyne, and P. Cawley, *Defect detection in pipes using guided waves*. Ultrasonics, 1998. **36**(1-5): p. 147-154.
- [56] Clezio, E.L., M. Castaings, and B. Hosten, *The interaction of the S₀ Lamb mode with vertical cracks in an aluminum plate*. Ultrasonics, 2002. **40**(1-8): p. 187-192.

- [57] Hayashi, T. and K. Kawashima, *Multiple reflections of Lamb waves at a delamination*. Ultrasonics, 2002. **40**(1-8): p. 193-197.
- [58] Seale, M.D., et al., *Lamb Wave Response of Fatigued Composite Samples*. Rev. Prog. in QNDE, 1994. **13**: p. 1261-1266.
- [59] Seale, M.D., *Propagation of Guided Acoustic Waves in Composite Media*, in *Dept. Applied Science*. 1996, College of William and Mary: Williamsburg, VA.
- [60] Bray, D.E., D.M. Egle, and L. Reiter, *Rayleigh wave dispersion in the cold-worked layer of used railroad rail*. J. Acoust. Soc. Am., 1978. **64**(3): p. 845-851.
- [61] Bray, D.E. *Higher-order Mode Rayleigh Waves in the Cold-Worked Zone of Railroad Rails*. in *Proceedings Fourteenth Symposium on Nondestructive Evaluation*. 1983. Nondestructive Testing and Analysis Center (NTIAC) San Antonio, Texas. p. 520-525
- [62] Bray, D.E., *Application of the First Higher-Order (M21) Mode Rayleigh Wave to the Inspection of Stainless Steel Overlays*. Transactions of the ASME, 1990. **112**: p. 298-300.
- [63] Grewal, D.S., *Improved Ultrasonic Testing of Railroad Rail for Transverse Discontinuities in the Rail Head Using Higher Order Rayleigh (M21) Waves*. Materials Evaluation, 1996. **54**(9): p. 983-986.
- [64] Seale, M.D. and B.T. Smith. *Lamb wave propagation in thermally damaged composites*. in *Rev. Prog. in QNDE*. 1996. p. 261
- [65] Grondel, S., et al., *Fatigue crack monitoring of riveted aluminium strap joints by Lamb wave analysis and acoustic emission measurement techniques*. NDT&E International, 2002. **35**(3): p. 137-146.
- [66] Beard, M.D., M.J.S. Lowe, and P. Cawley, *Inspection of Rockbolts Using Guided Ultrasonic Waves*. Rev. Prog. in QNDE, 2001. **20**: p. 1156-1163.
- [67] Beard, M.D., M.J.S. Lowe, and P. Cawley, *Ultrasonic Guided Waves for Inspection of Grouted Tendons and Bolts*. Journal of Materials in Civil Engineering, 2003. **15**(3): p. 212-218.

- [68] National Transportation Safety Board. *Aircraft Accident Report: Aloha Airlines, Flight 243, Boeing 737-200, n73711, near Maui, Hawaii, April 28, 1988*. 1989, National Technical Information Service.
- [69] Jansen, D.P. and D.A. Hutchins, *Lamb Wave Tomography*. IEEE 1990 Ultrasonics Symposium, 1990. 2: p. 871-874.
- [70] Hutchins, D.A., D.P. Jansen, and C. Edwards, *Lamb-wave tomography using non-contact transduction*. Ultrasonics, 1993. 31(2): p. 97-103.
- [71] Jansen, D.P. and D.A. Hutchins, *Immersion tomography using Rayleigh and Lamb waves*. Ultrasonics, 1992. 30(4): p. 245-254.
- [72] Jansen, D.P., D.A. Hutchins, and J.T. Mottram, *Lamb wave tomography of advanced composite laminates containing damage*. Ultrasonics, 1994. 32(2): p. 83-89.
- [73] Nagata, Y., et al., *Lamb Wave Tomography Using Laser-Based Ultrasonics*. Rev. Prog. in QNDE, 1995. 14: p. 561-568.
- [74] Degertekin, F.L., et al., *In situ acoustic temperature tomography of semiconductor wafers*. Appl. Phys. Lett., 1994. 64(11): p. 1338-1340.
- [75] Hildebrand, B.P., et al., *Lamb Wave Tomography for Imaging Erosion/Corrosion in Piping*. Rev. Prog. in QNDE, 1999. 18: p. 967-973.
- [76] Webb, S., *The Physics of Medical Imaging*. 1988, London: IOP Publishing Ltd.
- [77] Gazis, D.C., *Three-Dimensional Investigation of the Propagation of Waves in Hollow Circular Cylinders. I. Analytical Foundation*. J. Acoust. Soc. Am., 1959. 31(5): p. 568-573.
- [78] Gazis, D.C., *Three-Dimensional Investigation of the Propagation of Waves in Hollow Circular Cylinders. II. Numerical Results*. J. Acoust. Soc. Am., 1959. 31(5): p. 573- 578.
- [79] Silk, M.G. and K.F. Bainton, *The propagation in metal tubing of ultrasonic wave modes equivalent to Lamb waves*. Ultrasonics, 1979. 17(1): p. 11-19.

- [80] Mohr, W. and P. Holler, *On Inspection of Thin-Walled Tubes for Transverse and Longitudinal Flaws by Guided Ultrasonic Waves*. IEEE Trans. Sonics and Ultrasonics., 1976. SU-23(5): p. 369-374.
- [81] Zemanek Jr., J., *An Experimental and Theoretical Investigation of Elastic Wave Propagation in a Cylinder*. J. Acoust. Soc. Am., 1972. 51(1): p. 265-283.
- [82] Mecker, T.R. and A.H. Meitzler, *Guided wave propagation in elongated cylinders and plates*. Phys. Acoust., 1964. 1A: p. 111-167.
- [83] Sun, Z., et al., *Investigation on Interaction of Lamb Waves and Circumferential Notch in Pipe by Means of Wavelet Transform*. IEEE 2000 Ultrasonics Symposium, 2000: p. 827-830.
- [84] Ditri, J.J., *Utilization of guided elastic waves for the characterization of circumferential cracks in hollow cylinders*. J. Acoust. Soc. Am., 1994. 96: p. 3769-3775.
- [85] Rose, J.L., et al., *A guided wave inspection technique for nuclear steam generator tubing*. NDT&E International, 1994. 27: p. 307-330.
- [86] Rose, J.L., K.M. Rajana, and F.T. Carr, *Ultrasonic Guided Wave Inspection Concepts for Steam Generator Tubing*. Materials Evaluation, 1994. 52(2): p. 307-311.
- [87] Alleyne, D.N. and P. Cawley, *The Excitation of Lamb Waves in Pipes Using Dry-Coupled Piezoelectric Transducers*. Journal of Nondestructive Evaluation, 1996. 15(1): p. 11-20.
- [88] Alleyne, D.N. and P. Cawley, *Long Range Propagation of Lamb Waves in Chemical Plant Pipework*. Materials Evaluation, 1997. 55(4): p. 504-508.
- [89] Alleyne, D., et al. *The Lamb wave inspection of chemical plant pipework*. in *Review of Progress in QNDE*. 1997: Plenum, New York. p. 1269-1276
- [90] Guo, D. and T. Kundu, *A new transducer holder mechanism for pipe inspection*. J. Acoust. Soc. Am., 2001. 110(1): p. 303-309.

- [91] Bottger, W., H. Schneider, and W. Weingarten, *Prototype EMAT System for Tube Inspection with Guided Ultrasonic Waves*. Nuclear Engineering and Design, 1987. **102**(3): p. 369-376.
- [92] Hirao, M. and H. Ogi, *An SH-wave EMAT technique for gase pipeline inspection*. NDT&E International, 1999. **32**(3): p. 127-132.
- [93] Alers, G.A. and J.D. McColskey, *Measurement of Residual Stress in Bent Pipelines*. Rev. Prog. in QNDE, 2002. **21**: p. 1681-1687.
- [94] Alleyne, D.N., M.J.S. Lowe, and P. Cawley, *The reflection of guided waves from circumferential notches in pipes*. J. Appl. Mech., 1998. **65**: p. 635-641.
- [95] Lowe, M.J.S., *The mode conversion of a guided wave by a part-circumferential notch in a pipe*. J. Appl. Mech., 1998. **65**: p. 649-656.
- [96] Demma, A., P. Cawley, and M.J.S. Lowe, *Guided Waves in Curved Pipes*. Rev. Prog. in QNDE, 2002. **21**: p. 157-164.
- [97] Aristegui, C., M.J.S. Lowe, and P. Cawley, *Guided waves in fluid-filled pipes surrounded by different fluids*. Ultrasonics, 2001. **39**(5): p. 367-375.
- [98] Barshinger, J.N. and J.L. Rose, *Ultrasonic Guided Wave Propagation in Pipes with Viscoelastic Coatings*. Rev. Prog. in QNDE, 2002. **21**: p. 239-246.
- [99] Long, R., et al., *The Effect of Soil Properties on Acoustic Wave Propagation in Buried Iron Water Pipes*. Rev. Prog. in QNDE, 2002. **21**: p. 1310-1317.
- [100] Shannon, K., et al., *Mode conversion and the path of acoustic energy in a partially water-filled aluminum tube*. Ultrasonics, 1999. **37**(4).
- [101] Cheeke, J.D.N., X. Li, and Z. Wang, *Observation of flexural Lamb waves (A0 mode) on water-filled cylindrical shells*. J. Acoust. Soc. Am., 1998. **104**(6): p. 3678-3680.
- [102] Aristegui, C., P. Cawley, and M. Lowe, *Reflection and Mode Conversion of Guided Waves at Bends in Pipes*. Rev. Prog. in QNDE, 2000. **19**: p. 209-216.

- [103] Junger, M.C. and D. Feit, *Sound, Structures, and Their Interaction*. 1972, Cambridge, MA: The MIT Press.
- [104] Callahan, J. and H. Baruh, *A closed-form solution procedure for circular cylindrical shell vibrations*. *Int. J. Solids Struct.*, 1999. **36**: p. 2973-3013.
- [105] Kovalev, V.A., L.Y. Kossovich, and A.V. Nikonov, *Transient waves in a cylindrical shell subjected to sudden harmonic loads*. *Mech. Solids*, 2000. **35**: p. 143-152.
- [106] Poruchikov, V.B., *Response of a cylindrical elastic shell to an applied impulse*. *Mech. Solids*, 2000. **35**: p. 147-152.
- [107] Blonigen, F.J. and P.L. Marston, *Leaky helical flexural wave scattering contributions from tilted cylindrical shells: Ray theory and wave-vector anisotropy*. *J. Acoust. Soc. Am.*, 2001. **110**: p. 1764-1760.
- [108] Pierce, A.D. and H.-G. Kil, *Elastic Wave Propagation from Point Excitations on Thin-Walled Cylindrical Shells*. *Journal of Vibration and Acoustics*, 1990. **112**: p. 399-406.
- [109] Grover, D., et al., *Integrity of Uranium Hexafluoride Cylinders: Defense Nuclear Facilities Safety Board Technical Report*. 1995.
- [110] Office of Depleted Uranium Hexafluoride, U.S.D.o.E.
- [111] Jespersen, S., J. Wilhjelm, and H. Sillesen, *Multi-angle compounding imaging*. *Ultrasonic Imaging*, 1998. **20**: p. 82-102.
- [112] Trahey, G., S. Smith, and O. van Ramm, *Speckle pattern correlation with lateral aperture translation: experimental results and implications for spatial compounding*. *IEEE Trans. Ultrason. Ferroelectr. Freq. Contr.*, 1986. **33**: p. 257-264.
- [113] Wilhjelm, J., et al. *Some imaging strategies in multi-angle spatial compounding*. in *IEEE 2000 Ultrasonics Symposium*. 2000. p. 1237-1243

- [114] Wells, P.N.T. and M. Halliwell, *Speckle in ultrasonic imaging*. Ultrasonics, 1981. **19**: p. 225-229.
- [115] Behar, V., D. Adam, and Z. Friedman, *A new method of spatial compounding imaging*. Ultrasonics, 2003. **41**(5): p. 377-384.
- [116] Leonard, K.R., *Neural Network Technology and Lamb Wave Tomography*, in *Department of Physics*. 1999, College of William and Mary: Williamsburg.
- [117] Hou, J., *Ultrasonic signal detection and recognition using dynamic wavelet fingerprints*, in *Dept. Applied Science*. 2004, College of William and Mary: Williamsburg, VA.
- [118] Hou, J. and M.K. Hinders, *Dynamic Wavelet Fingerprint Identification of Ultrasound Signals*. Materials Evaluation, 2002. **60**(9): p. 1089-1093.
- [119] Strang, G., *Wavelets*, in *American Scientist*. 1994. p. 250-255.
- [120] Abbate, A., et al., *Signal detection and noise suppression using a wavelet transform signal processor: application to ultrasonic flaw detection*. IEEE Trans. Ultrason. Ferroelectr. Freq. Contr., 1997. **44**: p. 14-26.
- [121] Coifman, R.R. and D.L. Donoho, *Translation invariant de-noising*. Lecture Notes in Statistics, 1995. **103**: p. 125-150.

Vita

Kevin Leonard was born in Rota, Spain on June 14, 1977 to Raymond and Barbara Leonard. He received a B.S. degree in Physics and Government from the College of William and Mary in 1999. During his senior year at William and Mary he also received high honors for his research on neural networks and Lamb wave tomography. In August 2000, Kevin entered the College of William and Mary as a graduate assistant in the Department of Applied Science. He received an M.S. degree in Applied Science with a concentration in Nondestructive Evaluation in 2002. Kevin successfully defended his dissertation on May 28, 2004. He currently lives in Richmond, Virginia with his wife, Heather.