University of New Haven
Digital Commons @ New Haven

Master's Theses Student Works

8-2019

# Application of Elastodynamic Finite Integration Technique (EFIT) to Three-Dimensional Wave Propagation and Scattering in Arbitrary Geometries

Sean M. Raley
*University of New Haven*

Follow this and additional works at: https://digitalcommons.newhaven.edu/masterstheses

Part of the Mechanical Engineering Commons

## Recommended Citation

Raley, Sean M., "Application of Elastodynamic Finite Integration Technique (EFIT) to Three-Dimensional Wave Propagation and Scattering in Arbitrary Geometries" (2019). *Master's Theses*. 149.
https://digitalcommons.newhaven.edu/masterstheses/149

THE UNIVERSITY OF NEW HAVEN

# APPLICATION OF ELASTODYNAMIC FINITE INTEGRATION TECHNIQUE (EFIT) TO THREE-DIMENSIONAL WAVE PROPAGATION AND SCATTERING IN ARBITRARY GEOMETRIES

A THESIS

submitted in partial fulfillment

of the requirements for the degree of

MASTER OF SCIENCE IN MECHANICAL ENGINEERING

BY

Sean M. Raley

University of New Haven
West Haven, Connecticut
August, 2019

## ACKNOWLEDGMENTS

Sincerest thanks to Professor Eric A. Dieckman, Ph.D. for his patience and support in teaching me the many, many things I needed to know to complete this thesis.

Special thanks also to Meredith C. Powell, Ph.D. and John E. Leonard; and appreciation to UNH.

ABSTRACT

Over several decades, railroad Ultrasonic Testing (UT) industry techniques have primarily been developed through simple analytical modelling and experimental approaches. However, with present-day computational capabilities, we can use numerical techniques like the Elastodynamic Finite Integration Technique (EFIT) to fine-tune systems for complex applications before the fabrication process begins. EFIT is well-established as a useful method in numerical analysis of ultrasonic wave propagation with distinct advantages over the Finite Difference Time Domain method. Several software packages exist that use EFIT as the primary method for simulating the behavior of ultrasonic waves over time in 2 or 3 dimensions, but none of them are well-suited for railroad UT research and development. This thesis explores the complete development of a custom tool for this purpose which was designed to: (1) allow for the input of various profile geometries, boundary conditions, and material inclusion geometries (such as a bolt hole in a railroad track); and (2) allow for the input of specific ultrasonic impulses from varying emitter designs. The custom software produced results that closely matched expected wave propagation behavior. The results were processed into useful visual representations of that behavior.

TABLE OF CONTENTS

ABSTRACT ... iv

LIST OF FIGURES ... vi

CHAPTER I: Ultrasonic Testing in Rail Applications and Relevant Numerical Solutions ... 1

CHAPTER II: AFIT Implementation and Results ... 11

CHAPTER III: EFIT Implementation and Results ... 16

CHAPTER IV: Conclusions and Future Work ... 30

APPENDIX ... 32

MATLAB Code: Input File Generation ... 33

MATLAB Code: Arbitrary Geometry Importing and Processing ... 37

C++ Code: ‘efit.cpp’ ... 41

C++ Code: ‘space.h’ ... 53

C++ Code: ‘transducer.h’ ... 68

SELECTED REFERENCES ... 71

LIST OF FIGURES

Figure 1: UT wheel probe layout. 3

Figure 2: Staggered grid architecture. 6

Figure 3: This flowchart outlines the general architecture for this FIT implementation. 11

Figure 4: Example of PML's implemented in 2-D space. 13

Figure 5: Half-space 3-D view of the rigid sphere AFIT simulation. 14

Figure 6: AFIT results: rigid sphere. 15

Figure 7: AFIT PML illustration. 15

Figure 8: The Cauchy stress tensor 16

Figure 9: Calibration rail modeling. 18

Figure 10: Calibration rail meshing. 19

Figure 11: EFIT Case 1 – simple block with void. 20

Figure 12: EFIT Case 2 – simple block comparison (no void). 21

Figure 13: Cutting planes for 2-D slices of 3-D rail space. 22

Figure 14: EFIT Case 3 – beam in rail, 0º, 2-D slices. 23

Figure 15: EFIT Case 3 – beam in rail, 0º, 2-D slice (longitudinal). 24

Figure 16: EFIT Case 3 – beam in rail, 0º, 3-D view. 24

Figure 17: Snell's Law for acoustics in two solids. 26

Figure 18: EFIT Case 4 – beam in rail, 56.4º, 3-D view. 26

Figure 19: EFIT Case 4 – beam in rail, 56.4º, 2-D slice. 27

Figure 20: EFIT Case 5 – beam in rail, 37.5º, 2-D slice (longitudinal). 28

Figure 21: EFIT Case 5 – beam in rail, 37.5º, 3-D view. 28

Figure 22: EFIT Case 6 – beam in rail, 70º, 2-D slice (longitudinal). .......................................... 29

Figure 23: EFIT Case 6 – beam in rail, 70º, 3-D view. ................................................................. 29

# CHAPTER I: Ultrasonic Testing in Rail Applications and Relevant Numerical Solutions

How do we predict an imminent train derailment on railroad track that has been in service for 80 years or more? How do we determine that a brand new gas cylinder is likely to explode if it is put into service? In cases like these and many others, the answer is Ultrasonic Testing (UT), a subset of nondestructive evaluation (NDE). UT is, at its core, concerned with inducing ultrasonic waves into a material and making inferences about the material's internal structure based on the exiting ultrasonic waves. Feasible candidates for UT include product forms that are wrought, cast, welded, composite, or other materials as long as they have: (a) surface and internal geometries suitable for the application of UT; and (b) relevant defect definitions that are of shapes and orientations that conducive to ultrasonic inspection [1]. UT is used to ensure the safety of many types of system components and structures, including train wheels, pressurized tanks, railroad tracks, tubes, pipes, ammunition, airplane parts, and structural beams. Many of these components are inspected using UT techniques both at the time of manufacture and while in service.

UT is a well-established application of acoustic science. The basic principle is the same as SONAR: a sound wave is emitted, it reflects off of some feature of interest, and the reflection is detected and interpreted. The departure from large scale sensing techniques is driven by the frequencies used. SONAR mainly uses frequencies in the human audible range (20-20,000 Hz), while ultrasonic waves are by definition above 20 kHz. Attenuation is proportional to the square of frequency, so detection range decreases at higher frequencies, but resolution and sensitivity increase. An important note is that at ultrasonic frequencies, sound cannot propagate any useful distance in air. UT is typically conducted in the MHz range (0.5-25 MHz) [1].

Ultrasonic waves are typically generated by piezoelectric transducers, which exploit the piezoelectric effect, in which structural deformation is induced by applied voltage and vice versa. This characteristic allows them to both emit ultrasonic waves and convert incoming ultrasonic signals into electrical signals, which can then be recorded and interpreted by the rest of the system. Transducer configurations include pulse-echo (the emitting transducer is also the detecting transducer), pitch-catch (a second transducer acts as the detector for reflections of an angled ultrasonic beam), and through-transmission (similar to pitch-catch, but with the transducers located on opposite sides of the target material).

Since railroad track inspection is one of the primary motivators of this project, a more thorough background of that specific UT application is appropriate. While the details of the application are complex, the general concept is relatively simple: a fluid-filled wheel probe with several transducers held at fixed angles inside it rolls down the track, continuously collecting measurements (Figure 1). The ultrasonic beam propagates from each transducer, through the wheel fluid, through the membranous tire, through a film of liquid couplant sprayed on the rail to eliminate air gaps, and into the railroad track. From there, if the beam strikes a discontinuity roughly orthogonal to its propagation direction, it will reflect back along the same path up into the wheel probe and be received by the same transducer that emitted the pulse. There are a number of different defect types which occur at various positions and angles within railroad tracks, so an array of transducers at various positions and angles must be used to find them.

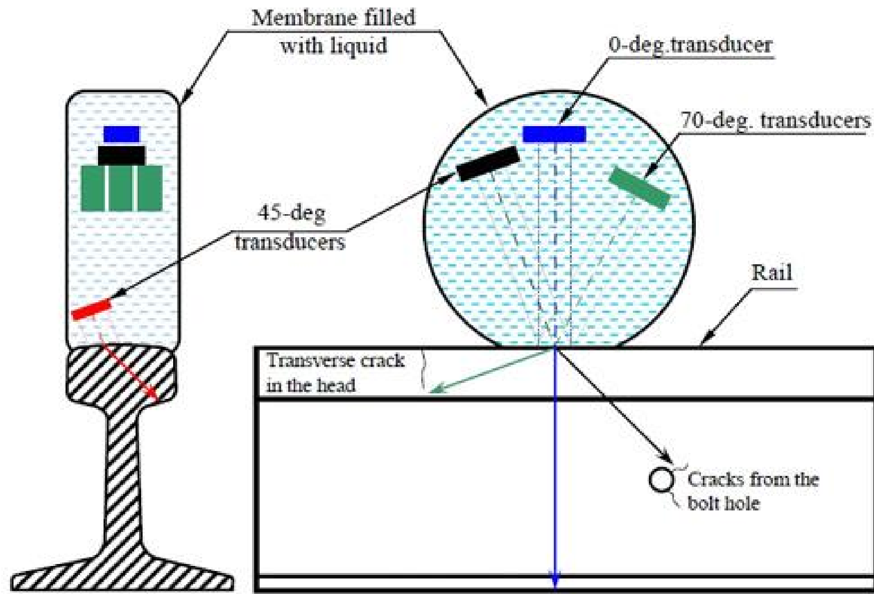


*Figure 1: UT wheel probe layout. Wheel probes are rolled along in-service railroad tracks to detect many different types of flaws. The wheel probe shown here contains a typical array of transducers. Adapted from* [2].

Railroad tracks experience a multitude of cyclic and transient loading conditions, including vertical loading from trains, lateral loading from trains, lateral creep forces at the rail/wheel interface, thermal stresses, and residual stresses from manufacturing or welding processes [3]. Due to these complex loading patterns which span decades or longer, there are many different types of defects which may develop in the rail and eventually cause failure. One of the most common defect types is the transverse fissure, which is a crosswise fracture originating from a nucleus inside the head and spreading outward orthogonally to the length of the rail. They are impossible to detect visually until the rail has already broken, but they can be found with a 70º angled transducer in a wheel probe. Another common defect type is the vertical split head, a progressive longitudinal fracture near the middle of the head along the length of the rail. They are typically not visible from the surface until they have grown several feet long. Vertical split heads are particularly prone to causing derailments after complete mechanical failure. Luckily, they can be found using 0º or 45º angled transducers.

Many discontinuities appropriate for UT in general fall under the categories of cracks, gaps, or other inclusions that behave approximately like rigid reflectors. Scattering of acoustic waves has well-established solutions for very simple geometries – for example, scattering of a propagating plane wave from a rigid sphere has been thoroughly explored [4]–[6]. However, as complexity is added to the reflector geometry, the incident waveform, or the boundary conditions, analytical solutions quickly become very cumbersome or impossible. At this point, more advanced numerical tools and techniques are required. As applicable to UT, these include semi-analytical techniques like the geometric theory of diffraction and the boundary element method (BEM), and numerical techniques operating directly on the fundamental equations of motion, among them finite difference time domain methods (FDTD), and finite element methods (FEM) [7].

This paper will focus on the Finite Integration Technique (FIT), an explicit hyperbolic time-domain solver, as applied to the acoustic and elastodynamic cases. In the case of the Acoustic Finite Integration Technique (AFIT), the model directly develops the fundamental governing equations into a parallelized solver for simulation of acoustic wave propagation in fluids in 3 dimensions. The Elastodynamic Finite Integration Technique (EFIT) is an analogous model for ultrasonic wave propagation in isotropic homogeneous solid media.

To better understand the general FIT, it is useful to first compare it to FDTD. Both methods generally use a cubic hexahedral or cylindrical grid, though advanced meshing techniques such as subgridding can be applied [8]. They both use Yee's staggered grid structure (primary grid and offset secondary grid), first presented in 1966 for electromagnetic simulation (Figure 2) [9]. FIT and FDTD both use a marching-in-time leapfrog scheme, meaning that the pressure updates alternate with the velocity updates a half timestep apart. By using this explicit

time integration scheme, no iteration is involved in the solution procedure. This also means that the maximum possible time step is determined by Courant-Friedrich-Lewy-criterion,

$$\Delta t \leq \Delta t_{max} = \frac{1}{\sqrt{n}} \frac{\Delta x}{c_{max}} \tag{1}$$

$$\Delta x \approx \frac{\lambda_{\min}}{10} \tag{2}$$

where $n$ is the number of spatial dimensions and $c_{max}$ is the highest wave propagation speed in the simulation. The minimum wavelength $\lambda_{\min}$ drives the maximum spatial discretization size $\Delta x$. Due to the inverse relationship between frequency and wavelength, high-frequency simulations are forced to finer resolutions, increasing computational requirements. The maximum temporal discretization size $\Delta t_{max}$ is directly proportional to the chosen $\Delta x$, so higher frequencies also force a finer time resolution.

In both models, the use of a fixed mesh causes spatial discretization error. A material property discretization error also occurs [10]. Additionally, FDTD and FIT are full-wave propagation methods as opposed to spectral methods, which are numerical techniques that operate in the frequency domain.

The key difference between FIT and FDTD is form of the governing field equations on which they are based. The FDTD approach uses the differential form of these equations, while the FIT is based on the integral forms of the field equations. This means that for a fixed mesh (which includes an inherent spatial discretization error), no additional equation discretization error is introduced when passing from the continuous to the discrete form [11]. In other words, FIT contains less inherent error than FDTD.

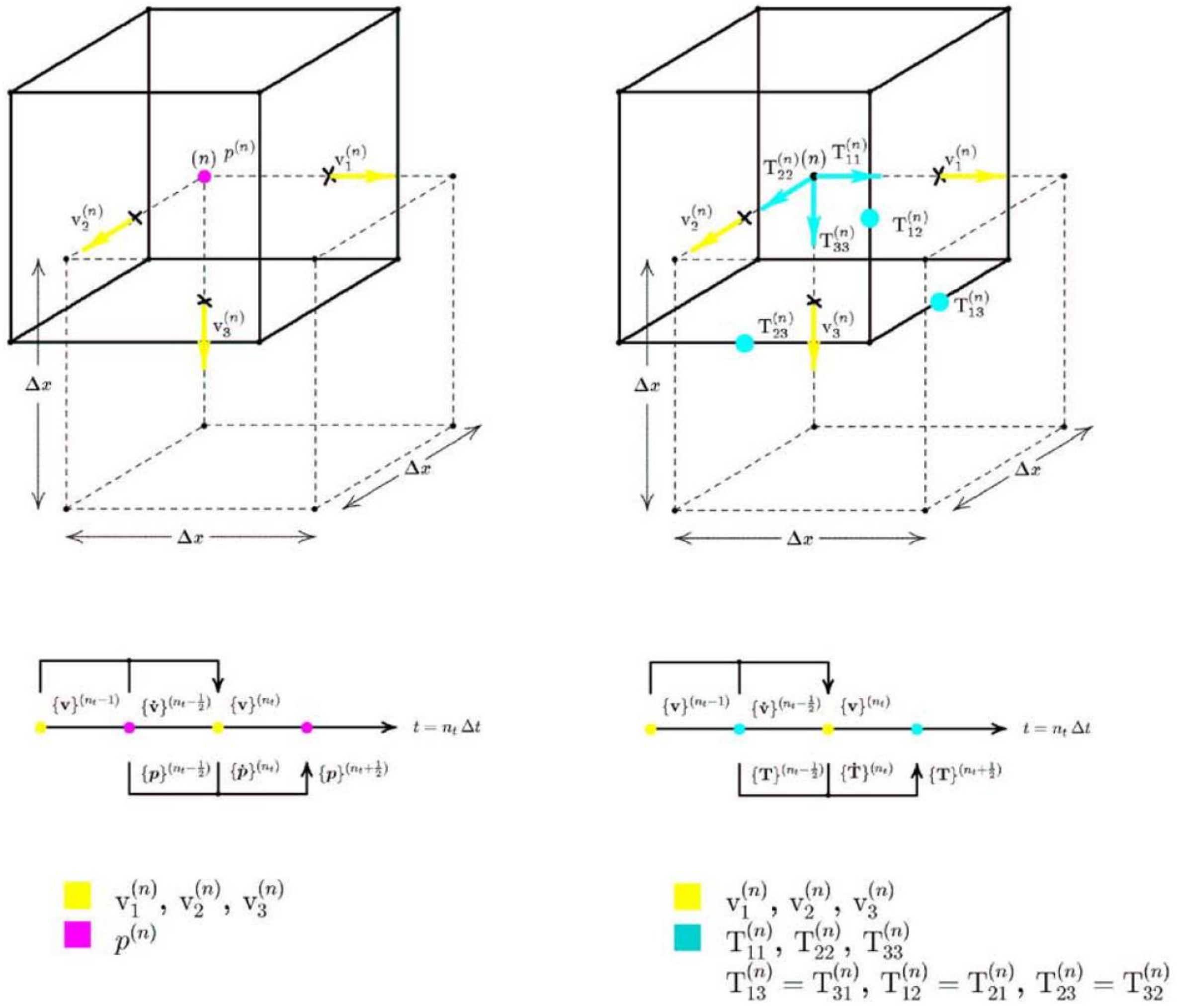


*Figure 2: Staggered grid architecture. Yee's staggered grid method can be applied to FIT for the simplified model of acoustic wave propagation in fluids (left) or the full elastodynamic model in solids (right). The corresponding leapfrog update schemes are shown as well (below). The Cartesian velocity vector* v *is positioned at the center of each face of the primary grid and the center of each edge of the secondary grid. The scalar pressure value for fluids* p *is shown in the center of the primary grid and at the corner of the secondary grid. The normal components of the Cauchy stress tensor* T *are positioned at the center of the primary grid, while the shear components are on the center of each edge of the primary grid. Adapted from* [12].

FIT and FDTD can be compared with FEM as a single grouping. As discussed in [13], FIT and FDTD are both more widely implemented than FEM mainly due to the relative simplicity of their programming. FEM is formulated using unstructured grids, which give it greater versatility for complex geometries but require advanced knowledge of mesh generation in order to properly implement. FEM requires far more computational resources than FDTD and

FIT for relatively simple geometric cases; however, the typical strategy for handling complex geometry in FDTD and FIT is to use a finer mesh, either globally or locally. Due to this limitation, FEM can be the more computationally efficient method for certain complex geometries. FEM is also generally superior to FDTD when material interfaces are involved, but FIT can easily account for continuity conditions at these interfaces.

For Acoustic FIT, propagation is only simulated through fluids, so no shear stresses can be carried in the material. The governing field equations are derived from conservation of mass, conservation of momentum, and a thermodynamic equation of state [14]. This leads to equations for the pressure $p$ and velocity vector $\underline{\mathbf{v}}$ in real cartesian space $\underline{\mathbf{R}}$ and time $t$,

$$\frac{\delta}{\delta t}\underline{\mathbf{j}}(\underline{\mathbf{R}},t) = -\nabla p(\underline{\mathbf{R}},t) + \underline{\mathbf{f}}(\underline{\mathbf{R}},t) \tag{3}$$

$$\frac{\delta}{\delta t}S(\underline{\mathbf{R}},t) = \nabla \cdot \underline{\mathbf{v}}(\underline{\mathbf{R}},t) + h(\underline{\mathbf{R}},t) \tag{4}$$

where $\underline{\mathbf{j}}$ is the momentum density vector, $\underline{\mathbf{f}}$ is the volume force density vector, $S$ is the scalar deformation, and $h$ is the injected deformation rate. This is the differential form used in FDTD.

To create the integral form of these equations for AFIT, Equations 3 and 4 are first combined with the constitutive material equations,

$$\underline{\mathbf{j}}(\underline{\mathbf{R}},t) = \rho_{a0}(\underline{\mathbf{R}})\underline{\mathbf{v}}(\underline{\mathbf{R}},t) \tag{5}$$

$$S(\underline{\mathbf{R}},t) = -\kappa(\underline{\mathbf{R}})p(\underline{\mathbf{R}},t) \tag{6}$$

where $\rho_{a0}$ is the acoustic mass density at rest and κ is the compressibility. The constitutive equations limit the scope of this model to linear, inhomogeneous, anisotropic, instantaneously and locally reacting media. Gauss' and Stokes' Theorems are then applied [12], [15] to arrive at the integral forms of the governing field equations,

$$\iiint_V \rho_{a0}(\underline{\mathbf{R}})\dot{\underline{\mathbf{v}}}(\underline{\mathbf{R}},t)dV = -\oiint_{S=\delta V} p(\underline{\mathbf{R}},t)\underline{\mathbf{dS}} + \iiint_V \underline{\mathbf{f}}(\underline{\mathbf{R}},t)dV \tag{7}$$

$$\underline{\mathbf{v}}(\underline{\mathbf{R}},t) = \underline{\mathbf{v}}(\underline{\mathbf{R}},t_0) + \int_{t_0}^{t} \dot{\underline{\mathbf{v}}}(\underline{\mathbf{R}},t')dt' \tag{8}$$

$$\iiint_V \kappa(\underline{\mathbf{R}})\dot{p}(\underline{\mathbf{R}},t)dV = -\oiint_{S=\delta V} \underline{\mathbf{v}}(\underline{\mathbf{R}},t)\cdot\underline{\mathbf{dS}} - \iiint_V h(\underline{\mathbf{R}},t)dV \tag{9}$$

$$p(\underline{\mathbf{R}},t) = p(\underline{\mathbf{R}},t_0) + \int_{t_0}^{t} \dot{p}(\underline{\mathbf{R}},t')dt'. \tag{10}$$

The integral forms are then directly converted into the numerical matrix update equations for AFIT,

$$\{\dot{\mathbf{v}}\}^{(n_t-1/2)} = [\tilde{\boldsymbol{\rho}}_{a0}]^{-1}\,[\tilde{\mathbf{R}}]^{-1}[\widetilde{\mathbf{grad}}]\{\mathbf{p}\}^{(n_t-1/2)} + [\tilde{\boldsymbol{\rho}}_{a0}]^{-1}\{\mathbf{f}\}^{(n_t-1/2)} \tag{11}$$

$$\{\mathbf{v}\}^{(n_t)} = \{\mathbf{v}\}^{(n_t-1)} + \Delta t\{\dot{\mathbf{v}}\}^{(n_t-1/2)} \tag{12}$$

$$\{\dot{\mathbf{p}}\}^{(n_t)} = -[\boldsymbol{\kappa}]^{-1}[\mathbf{div}][\mathbf{R}]^{-1}\{\mathbf{v}\}^{(n_t)} - [\boldsymbol{\kappa}]^{-1}\{\mathbf{h}\}^{(n_t)} \tag{13}$$

$$\{\mathbf{p}\}^{(n_t+1/2)} = \{\mathbf{p}\}^{(n_t-1/2)} + \Delta t\{\dot{\mathbf{p}}\}^{(n_t)} \tag{14}$$

where $n_t$ is an integer time step counter such that $t = \Delta t\, n_t$ and the ~ operator indicates that a matrix is defined on the secondary grid. This direct, one-to-one mapping of the field equations is the reason FIT does not have the equation discretization error found in FDTD and other numerical techniques.

For EFIT, the full elastodynamic equations must be used. The governing field equations for viscid fluids and solids are the linear vectorial Cauchy-Newton’s law of motion and the tensorial law of deformation rate,

$$\frac{\delta}{\delta t}\underline{\mathbf{j}}(\underline{\mathbf{R}},t) = \nabla\cdot\underline{\underline{\mathbf{T}}}(\underline{\mathbf{R}},t) + \underline{\mathbf{f}}(\underline{\mathbf{R}},t) \tag{15}$$

$$\frac{\delta}{\delta t}\underline{\underline{\mathbf{S}}}(\underline{\mathbf{R}},t) = sym\{\nabla\cdot\underline{\mathbf{v}}(\underline{\mathbf{R}},t)\} + \underline{\underline{\mathbf{h}}}(\underline{\mathbf{R}},t) \tag{16}$$

where $\underline{\underline{\mathbf{T}}}$ is Cauchy's stress tensor of 2nd rank, $\underline{\underline{\mathbf{S}}}$ is the deformation tensor of 2nd rank, $\underline{\underline{\mathbf{h}}}$ is the injected deformation rate tensor of 2nd rank, and "sym" is a symmetric gradient operator. This is the differential form used in FDTD.

To create the integral form of these equations for EFIT, Equations 17 and 18 are first combined with the constitutive material equations,

$$\underline{\mathbf{j}}(\underline{\mathbf{R}}, t) = \rho_{e0}(\underline{\mathbf{R}})\underline{\mathbf{v}}(\underline{\mathbf{R}}, t) \tag{17}$$

$$\underline{\underline{\mathbf{S}}}(\underline{\mathbf{R}}, t) = \underline{\underline{\underline{\underline{\mathbf{s}}}}}(\underline{\mathbf{R}}): \underline{\underline{\mathbf{T}}}(\underline{\mathbf{R}}, t) \tag{18}$$

where $\rho_{e0}$ is the elastodynamic mass density at rest, $\underline{\underline{\underline{\underline{\mathbf{s}}}}}$ is the compliance tensor of 4th rank, and the colon ":" is an operator denoting a double-scalar product. The constitutive equations limit the scope of this model to linear, inhomogeneous, anisotropic, instantaneously and locally reacting media. Gauss' and Stokes' Theorems are then applied [12], [15] to arrive at the integral forms of the governing field equations,

$$\iiint_V \rho_{e0}(\underline{\mathbf{R}})\dot{\underline{\mathbf{v}}}(\underline{\mathbf{R}}, t)dV = \oiint_{S=\delta V} \underline{\underline{\mathbf{T}}}(\underline{\mathbf{R}}, t) \cdot \underline{\mathbf{dS}} + \iiint_V \underline{\mathbf{f}}(\underline{\mathbf{R}}, t)dV \tag{19}$$

$$\underline{\mathbf{v}}(\underline{\mathbf{R}}, t) = \underline{\mathbf{v}}(\underline{\mathbf{R}}, t_0) + \int_{t_0}^{t} \dot{\underline{\mathbf{v}}}(\underline{\mathbf{R}}, t')dt' \tag{20}$$

$$\iiint_V \underline{\underline{\underline{\underline{\mathbf{s}}}}}(\underline{\mathbf{R}}): \dot{\underline{\underline{\mathbf{T}}}}(\underline{\mathbf{R}}, t)dV = \oiint_{S=\delta V} sym\{\underline{\mathbf{n}}\,\underline{\mathbf{v}}(\underline{\mathbf{R}}, t)\}dS + \iiint_V \underline{\underline{\mathbf{h}}}(\underline{\mathbf{R}}, t)dV \tag{21}$$

$$\underline{\underline{\mathbf{T}}}(\underline{\mathbf{R}}, t) = \underline{\underline{\mathbf{T}}}(\underline{\mathbf{R}}, t_0) + \int_{t_0}^{t} \dot{\underline{\underline{\mathbf{T}}}}(\underline{\mathbf{R}}, t')dt' \tag{22}$$

where $\underline{\mathbf{n}}$ is the unit normal at an interface.

The integral forms are then directly converted into the numerical matrix update equations for EFIT,

$$\{\dot{\mathbf{v}}\}^{(n_t-1/2)} = [\widetilde{\boldsymbol{\rho}}_{e0}]^{-1}\,[\widetilde{\mathbf{DIV}}][\widetilde{\mathbf{R}}_{\dot{V}}^{T}]^{-1}[\mathbf{A}_{\dot{V}}^{T}]\{\mathbf{T}\}^{(n_t-1/2)} + [\widetilde{\boldsymbol{\rho}}_{e0}]^{-1}\{\mathbf{f}\}^{(n_t-1/2)} \tag{23}$$

$$\{\mathbf{v}\}^{(n_t)} = \{\mathbf{v}\}^{(n_t-1)} + \Delta t\{\dot{\mathbf{v}}\}^{(n_t-1/2)} \tag{24}$$

$$\{\dot{\mathbf{T}}\}^{(n_t)} = [\mathbf{c}][\mathbf{R}_{\dot{T}}^{V}]^{-1}[\mathbf{GRAD}][\mathbf{A}_{\dot{T}}^{V}]\{\mathbf{v}\}^{(n_t)} + \{\mathbf{g}\}^{(n_t)} \tag{25}$$

$$\{\mathbf{T}\}^{(n_t+1/2)} = \{\mathbf{T}\}^{(n_t-1/2)} + \Delta t\{\dot{\mathbf{T}}\}^{(n_t)} \tag{26}$$

where $[\mathbf{A}_{\dot{V}}^{T}]$ and $[\mathbf{A}_{\dot{T}}^{V}]$ are averaging matrices and $\mathbf{g}$ is the interface source density of injected deformation rate.

Now that the update equations have been developed for AFIT and EFIT, the coding and implementation can begin. We apply the numerical technique to a number of simple test cases, then expand the capabilities beyond well-established basic FIT implementations to accommodate arbitrary geometries – in this case, the simulation of ultrasound in railroad track.

## CHAPTER II: AFIT Implementation and Results

In this chapter, the programming and technical implementations will be discussed for AFIT only, but much of the information here will form a basis for the EFIT discussion in the next chapter. The general simulation architecture is to 1) write input files using MATLAB, 2) perform the marching-in-time algorithm using a parallelized C++ executable program written specifically for this project, and 3) perform postprocessing and create graphics using VisIt, an open-source visualization tool developed by the U.S. Department of Energy [16]. Each of the three main simulation steps can be performed sequentially on a computing cluster. The system used for these simulation runs had a 16-core 2.10 GHz CPU and 500 GB of RAM.

**Create input files in MATLAB**

- Space/time parameters
- Material properties
- Scatterers

**Parallelized C++ program (OpenMPI)**

- Reads input files and distributes them to any number of processors
- Sets up arrays, then does the math
- PMLs to minimize space requirements
- Periodically extracts output values

**Postprocessing in VisIt (visualization and graphical analysis tool)**

*Figure 3: This flowchart outlines the general architecture for this FIT implementation.*

The MATLAB script begins by defining various material, spatial, and temporal properties. For AFIT, these include material density, sound speed, maximum input frequency, number of time steps, physical size of the simulation space, and the sizes and locations of any scatterers. Several other parameters are then calculated; for example, the maximum input wavelength, spatial discretization size, and temporal discretization size are all interrelated and

dependent on the sound speed and maximum frequency. A "drive function" is defined based on either an analytical equation (e.g. a sine wave) or an input data file. The drive function is simply an array of pressure values for each time step which will later define a plane wave input at one end of the simulation space. Finally, a text file is written to pass all the necessary parameters (including the full drive function) to the C++ executable.

The executable simulation program uses Open MPI to pass information between processors "nodes" [17]. One node is defined as the master, which is responsible for reading and preprocessing the simulation parameters from the input file, distributing the parameters to all the other nodes, and recording output data periodically. The rest of the nodes are defined as slaves or workers, which each receive an evenly divided slice of the simulation space from the master node and are responsible for performing the actual calculations for the marching-in-time algorithm. All nodes are synced to the current time step as the simulation proceeds. The parallelization slices are orthogonal to the propagation direction of the plane wave.

All the calculations and array bookkeeping are defined by a ~500-line header file that is instantiated separately by each worker node. For each time step, every worker performs the same procedure independently. First, the drive function is applied as an input plane wave on only a single worker node. Then the pressures are updated for the entire space. Each worker node has one layer of overlap with its neighboring node(s), so these values must be passed and shared using MPI. Next, the velocities are updated for the entire space, with similar value sharing between nodes. This alternating update pattern is the leapfrog scheme in practice. At periodic intervals, the output data is extracted and sent to the master node for recording. Once all of a worker's values are updated for a given time step, absorbing boundary conditions are applied to the outer faces of the overall simulation space. The boundary condition used is a Perfectly

Matched Layer (PML), which is a technique that applies an incrementally increasing reduction factor to each element of the layer moving towards the outer faces of the simulation space [18]. These layers are part of the computational simulation space but are considered sacrificial and cannot produce valid physical results (Figure 4).

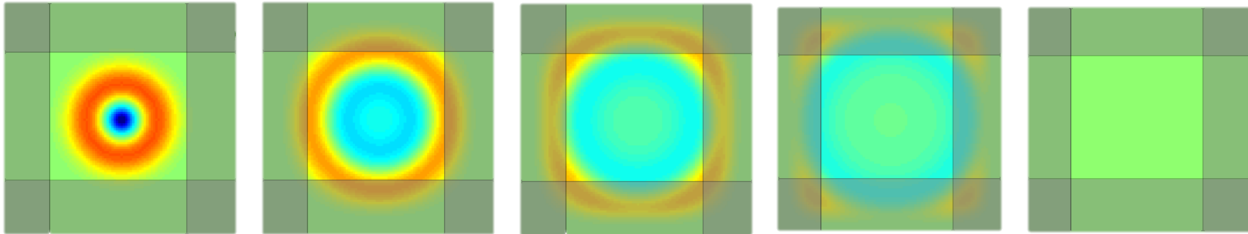

*Figure 4: Example of PML's implemented in 2-D space. The method avoids reflections at the inner edge of a PML by matching the impedance of the PML to the impedance of the problem space. Adapted from [19].*

Once the simulation is complete, a series of binary files containing the output data has been produced. To help VisIt interpret these binary files, a simple bash script is used to write a series of "brick of values" (.bov) files, which define data type and size, byte order, and other parameters for the corresponding data file. From there, it is relatively simple to create a 3-D plot of the data (Figure 5), which can be manipulated in a wide variety of ways, including video generation.

The primary test case for the AFIT implementation was a rigid sphere in a fluid, which is a well-studied acoustics problem. The input was a standing 2.5 kHz plane wave. The pressure output was recorded for only the even element positions in each principal direction, resulting in an 87.5% reduction of overall data file size. The expected interference patterns due to reflections from the sphere can be seen in Figure 6. The expected diffraction patterns behind the sphere were also observed. The simulation used PMLs on 5 faces of the computational space (Figure 7). In this implementation, the source face does not have a PML, so the sphere was spaced far

enough from the source face that reflections from it were not a concern for the simulation time duration.

There are far more interesting problems which can be solved using this implementation of AFIT, but for this thesis, AFIT is primarily used as a stepping stone on the way to the motivating problem of EFIT in arbitrary geometry.

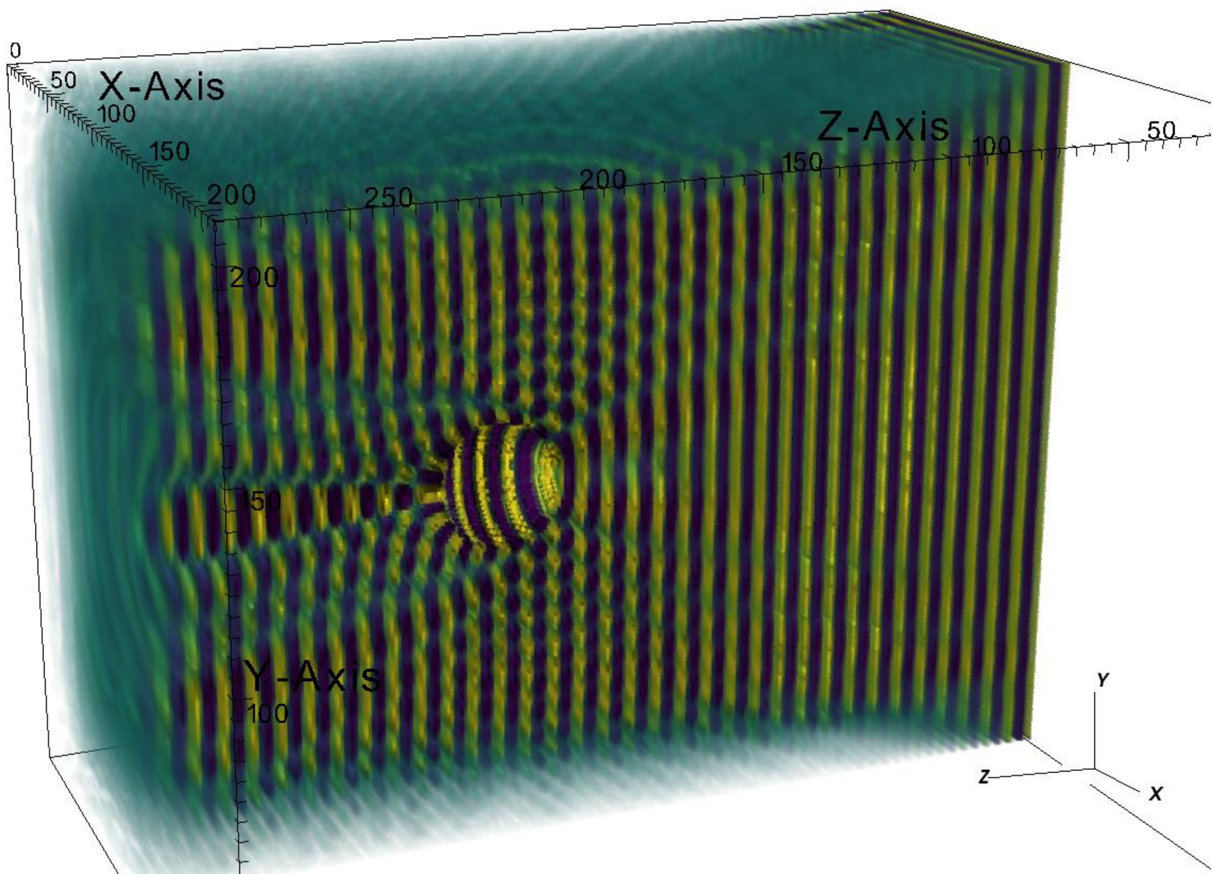


*Figure 5: Half-space 3-D view of the rigid sphere AFIT simulation. Note the clearly visible attenuation at the outer faces caused by the PML's.*

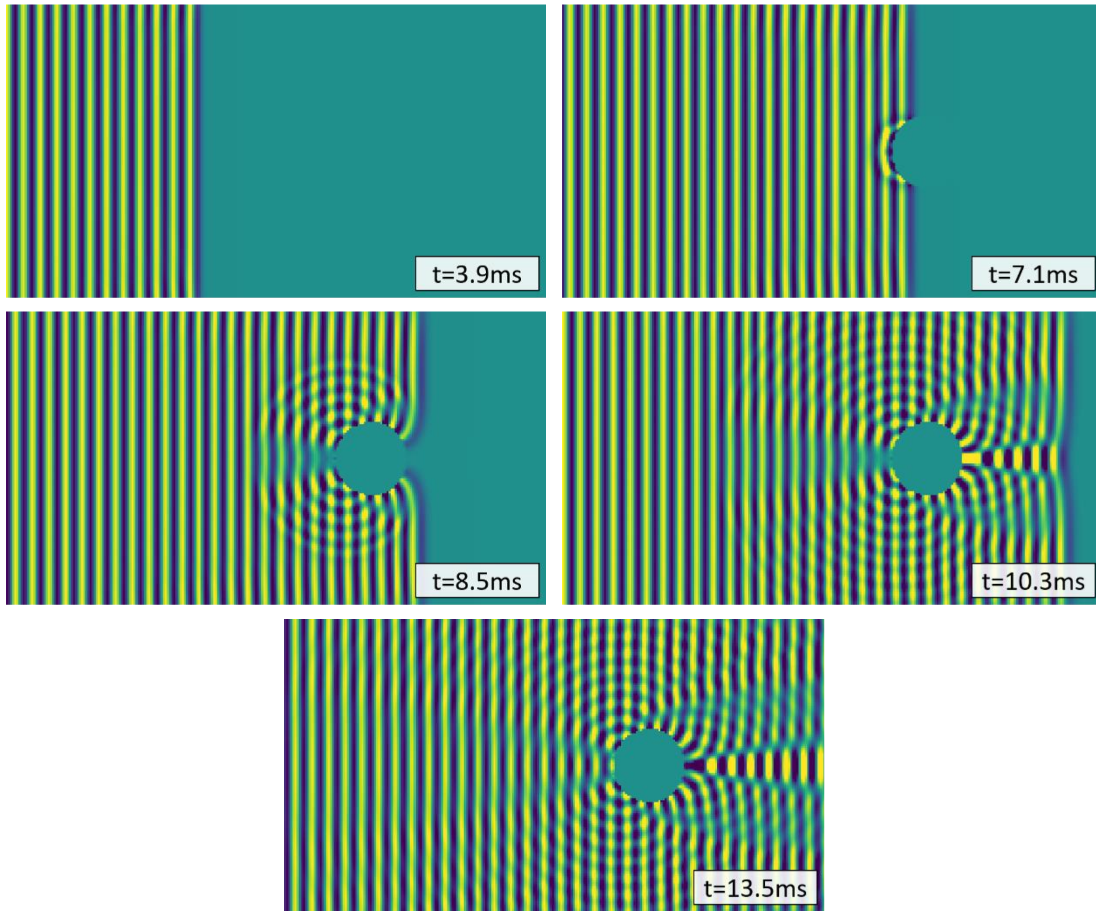


*Figure 6: AFIT results: rigid sphere. 2-D slice of a 3-D simulation of a 2.5 kHz plane wave reflecting from a rigid sphere. Results are shown at simulation time steps 345, 620, 740, 900, and 1180, with 1 time step = 11.43μs.*

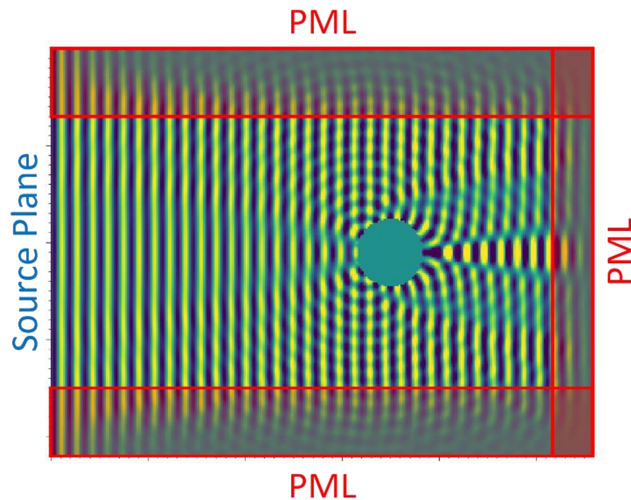


*Figure 7: AFIT PML illustration. The PML's in this simulation space (boxed in red) are not valid physical results, so they are generally omitted from final figures. Only 3 PMLs are shown here because the figure is a 2-D slice of a 3-D simulation. Result is shown at simulation time t=13.5 ms.*

# CHAPTER III: EFIT Implementation and Results

When transitioning from AFIT to EFIT, the general simulation structure remains more or less the same. There are two key differences that add a nontrivial complexity to the implementation of EFIT. First, instead of pressure, all six components of the Cauchy stress tensor must be used since solid materials can support shear stresses (Figure 8). This complicates bookkeeping and calculations. The second difference is specific to the intended application of this work: UT does not typically involve full-space plane waves, so a localized transducer definition must be developed to include the drive function in a localized "drive region."

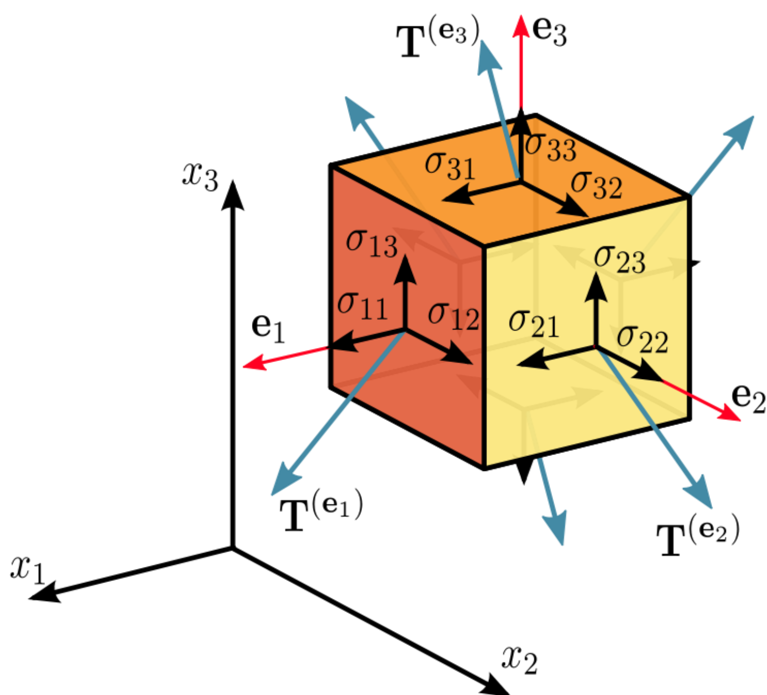


*Figure 8: The Cauchy stress tensor is composed of six unique stress vectors. Image from* [20].

In order to simplify the implementation of a discretized transducer definition, several assumptions are made. One of the largest simplifications is the limitation of transducer geometry to circular drive regions. There are many situations in which this leads to a slight departure from real-world physics, such as an ultrasonic beam striking the inspected material at an angle (which would create an oblong drive region) or a rectangular transducer being used (which creates a

drive region that is not quite circular, but not quite round). However, for most situations, a circular drive region can be assumed to introduce an acceptably low error into the simulation. Another simplification is the limitation of velocity input values to the direction orthogonal to the drive region, which can only be placed within the top plane of the simulation space. This is accurate for transducer beams propagating in the same direction (0º transducers in rail testing), but again, it introduces some error for angle beams, which realistically strike the test material with a total velocity parallel to the beam path.

Angled transducer orientations are crucial in many UT applications (with 37.5° and 45° being two of the most commonly used beam angles), so the addition of this feature was highly desirable. Angled propagation paths in the inspected material were generated using 2-D array beam steering [21].The drive function time delay for each spatial element is calculated from Equation 27 using the center of the transducer as a reference point, then offset so that all elements have a nonnegative time delay value. This delay time $\tau_{ij}$ is defined as

$$\tau_{ij} = \frac{z_i}{c} \sin\theta\cos\phi + \frac{y_j}{c} \sin\theta\sin\phi \tag{27}$$

where θ is the angle from incident normal (the 0º direction) towards positive *z*, ϕ is the clockwise angle from the positive *z* direction, and *c* is the longitudinal speed of sound. Once this delay array is set, the same drive function is used throughout the drive region with some elements simply calling a drive function value from an earlier timestep.

Another feature included in this EFIT implementation is the ability to import arbitrary geometry as the simulation space. UT is rarely performed on rectangular prisms, so defining other geometries is essential to creating a useful simulation tool. This is accomplished using stereolithography (STL) files as inputs. STL is a common filetype that represents 3-D geometry using interconnected triangular faces (Figure 9). In the initial MATLAB setup script, the STL

file is processed into a logical cubic mesh defining whether material exists at any given Cartesian coordinate within the simulation space (Figure 10). The coarseness of this mesh will determine the order of the spatial discretization error. A finer mesh is always more accurate, but computational requirements increase rapidly as discretization length decreases. Anywhere material does not exist will be treated as vacuum in the simulation, so the stresses and velocities in those elements will always be zero. This assumption is minor due to the extreme impedance differential between air and most solids relevant to UT.

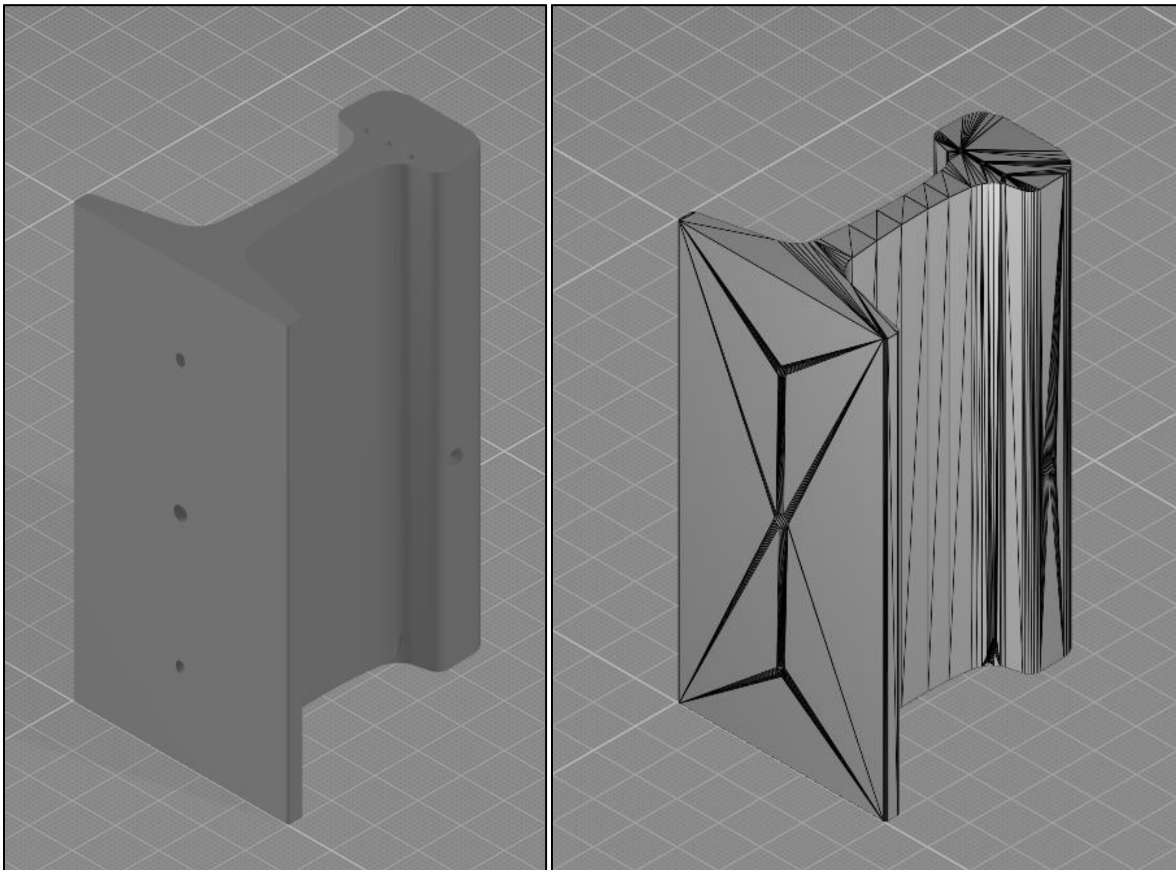

*Figure 9: Calibration rail modeling. A 3-D CAD model (left) of a calibration rail used in the UT industry. In the STL version of the same part (right), a high density of triangular faces can be seen intersecting at the hole locations.*

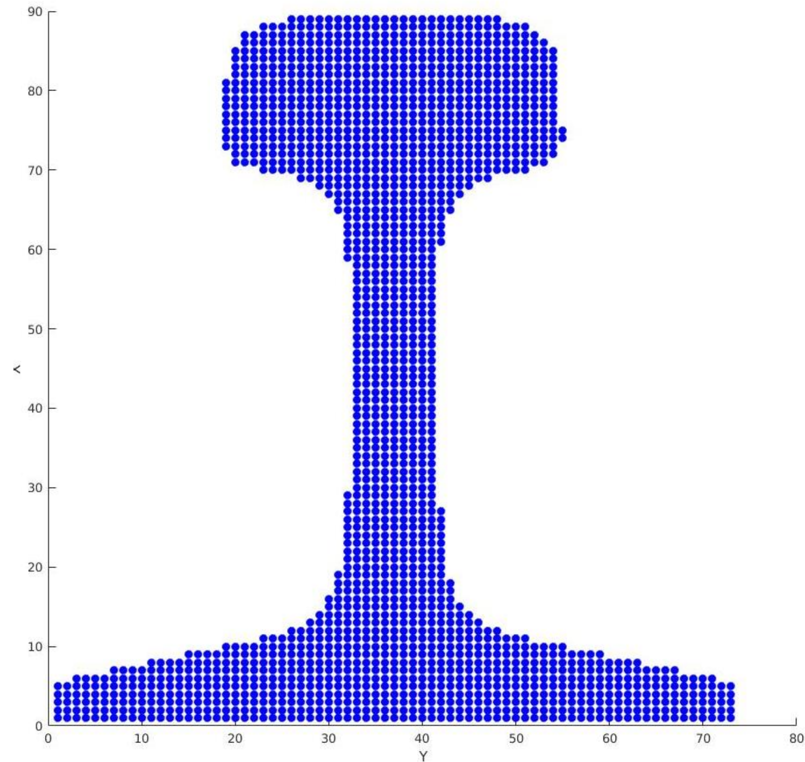


*Figure 10: Calibration rail meshing. This coarse mesh is an example of the cubic grid output of the processed STL rail.*

Multiple 3-D test cases were solved for the EFIT implementation. In all cases, the output was the magnitude of the velocity vector for each recorded element – again, only the even element positions were used in order to reduce data file size. EFIT Case 1 was a simple rectangular aluminum block with a small void inside it and a round 1 MHz transducer emitting a 5-period sine pulse normal to the top face (in other words, the drive region was defined as a circle at the top of the simulation space with no time delays). In the very simple Case 1, correct wave propagation behavior was observed, including reflections from the void and tip diffraction about the void (Figure 11).

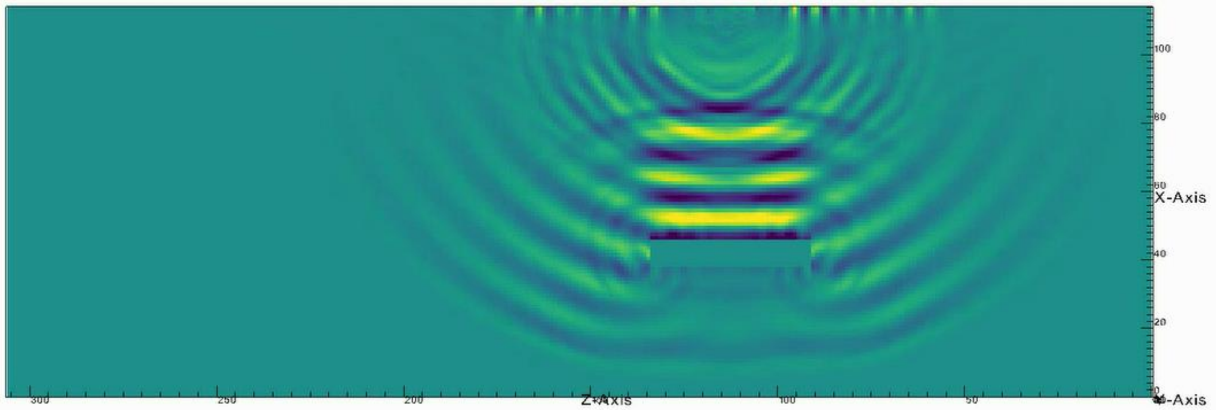


*Figure 11: EFIT Case 1 – simple block with void. A 2-D slice through the center of the 3-D block allows clear visualization of the wave's interaction with the void. The tip diffraction around the edges of the void are an expected physical response to this ultrasonic pulse.*

EFIT Case 2 was a smaller rectangular aluminum block (50x50x100mm) with no inclusions. The transducer was a 1.5 MHz source with a 21.4mm radius located at the "top" of the block emitting a 1-period sine pulse (Figure 12). Case 2 includes three separate simulations with the only difference being the angle of propagation produced by the time delay beam steering: 0º, 37.5º, and 70º. This comparison validates the methodology used to produce angled beams in the test material. In both angle beam simulations, a leading longitudinal wave is expected to travel at 6,235 m/s in the desired direction, while a slower shear wave is expected to travel at 3,139 m/s at a shallower angle. In all three simulations, surface (Rayleigh) waves are expected to travel at 2,906 m/s along the plane in which the transducer is located [22]. Using a rough, image-based interpolation technique, the actual propagation speed of the longitudinal wave in the 70º simulation is 5,963 m/s. The calculated shear wave speed is 2,981 m/s and the calculated surface wave speed is 2,793 m/s. All of these are within 4-5% of expected values, which is acceptable given the approximate method used to measure them.

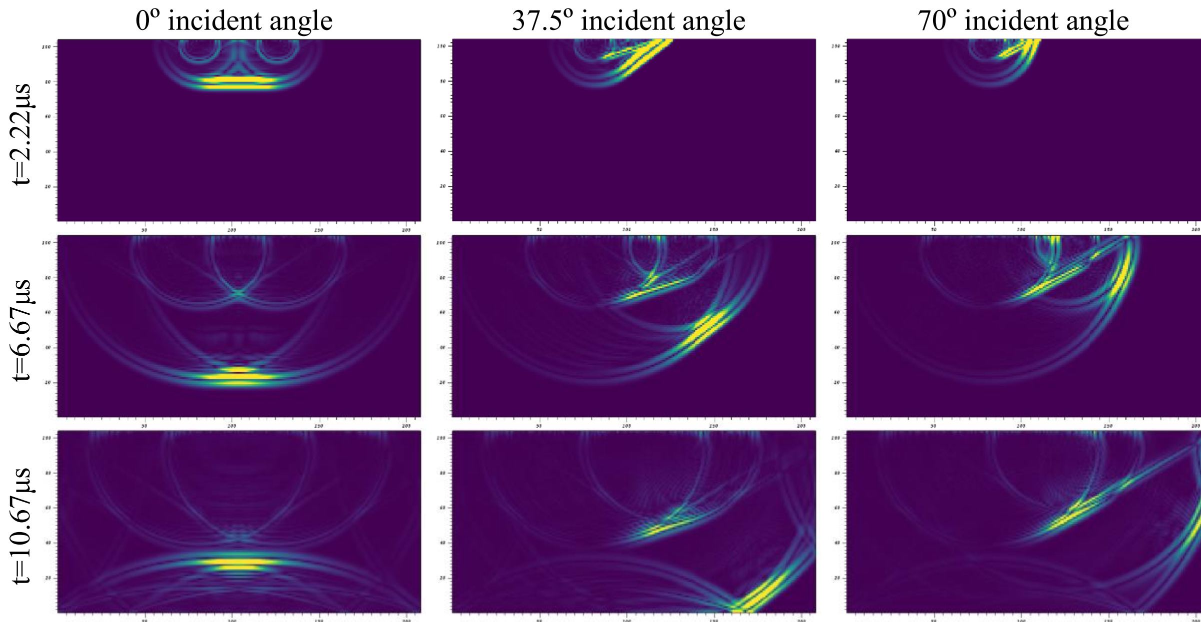


*Figure 12: EFIT Case 2 – simple block comparison (no void). The same 1.5 MHz transducer was applied to the same solid rectangular block three times at different angles: 0º from normal (left), 37.5º from normal (center), and 70º from normal (right). The 2-D slice is taken at the center of the 3-D block.*

Now that the basic simulation functions are validated, complex simulations in arbitrary geometry can be performed – for EFIT Cases 3-6, that geometry is a 12-inch-long section of steel railroad track. Processing this geometry in MATLAB with a spatial discretization of 0.212mm/element takes several hours since it can only be done by a single processor core. Fortunately, the processed geometry file can be used for multiple test cases if the transducer definition text file is altered manually. The executable phase of the simulation takes about 177 minutes on 8 processor nodes, or 111 minutes on 12 processor nodes. During executable initialization, RAM usage climbs to ~120 GB, then levels out at ~85 GB for the majority of the simulation. The output binaries, again only using data from even element positions, are 885.4 MB each – Cases 3-6 each produced 300 output files, taking up 266.5 GB of storage per simulation.

EFIT Case 3 is a 0º, ¾-inch, 2.25 MHz 1-period sine pulse at the top center of the rail (Figure 14). The ultrasonic wave reflects from the surface of a flat-bottom hole drilled in the

bottom of the rail, traveling back to the same location where the pulse was emitted. In applied UT testing, the pulse will continue up into the wheel probe, where it will excite the transducer and register a premature reflection in the UT software. In this simulation, the length of time for the wave to reach the hole was used to calculate the simulated propagation speed, which matched exactly the longitudinal speed of sound used in the original simulation setup. This is a good indicator that the simulation matches the physics.

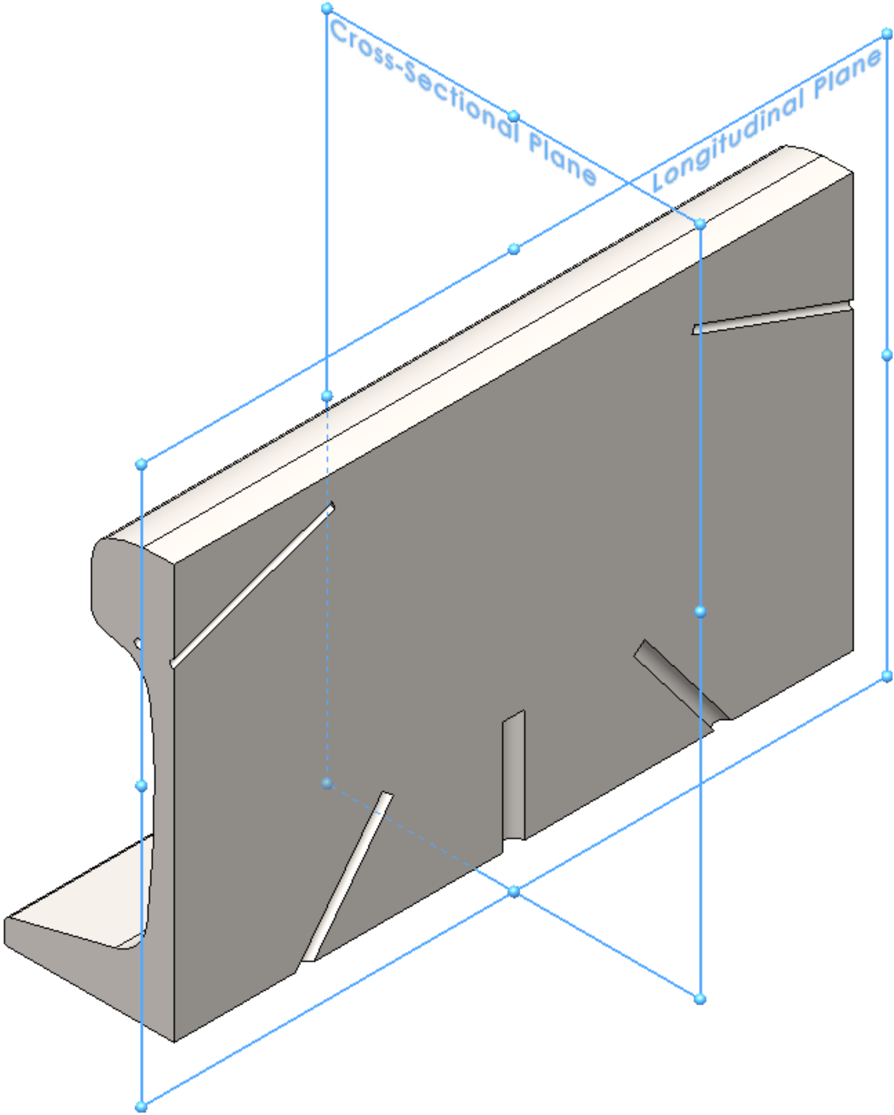


*Figure 13: Cutting planes for 2-D slices of 3-D rail space. The cross-sectional view cuts across the rail, while the longitudinal view slices down the length of the rail. Model shown with longitudinal slice active.*

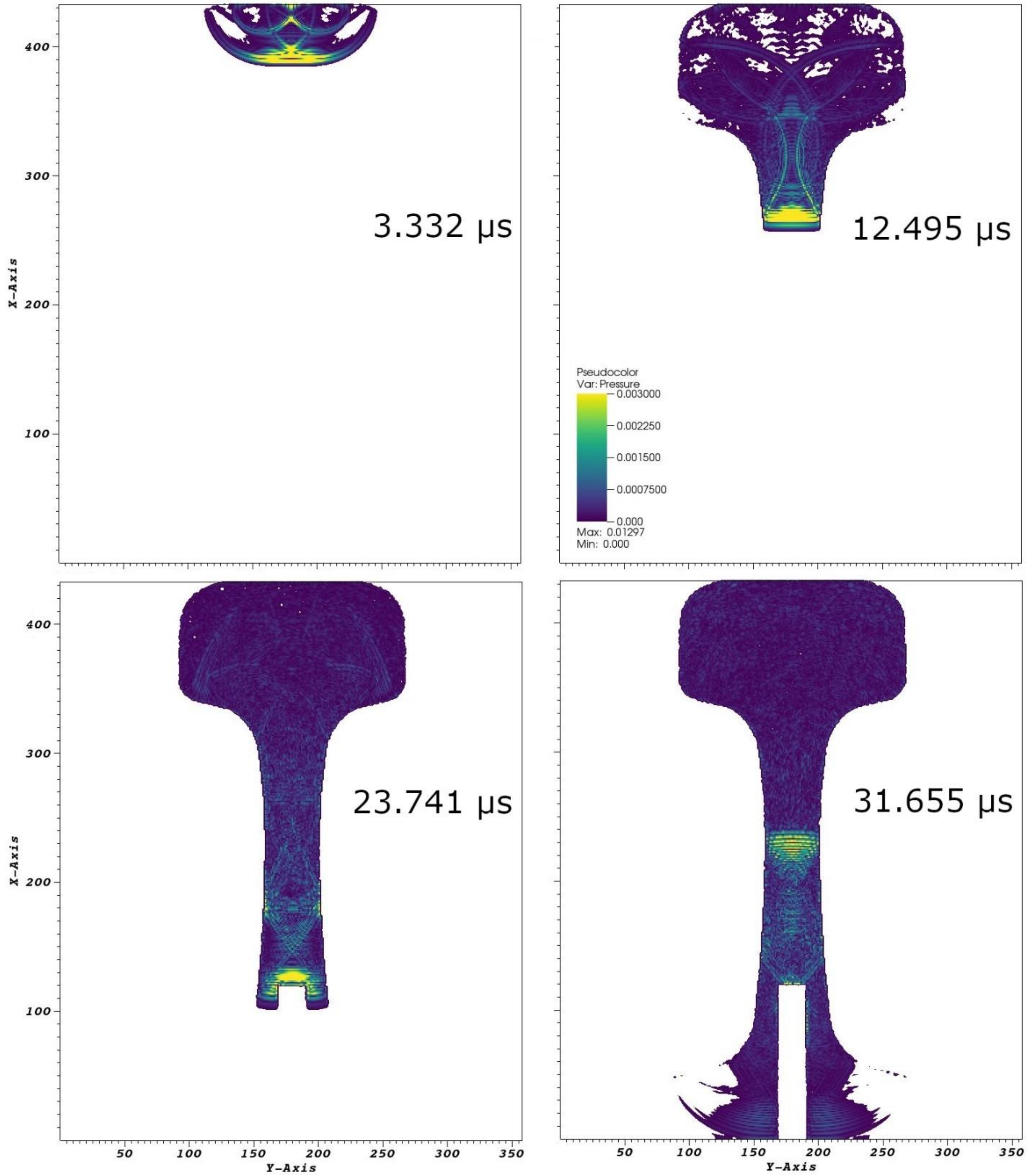


*Figure 14: EFIT Case 3 – beam in rail, 0º, 2-D slices. The ultrasonic wave enters through the head, travels straight down through the web, reflects from the flat-bottom hole drilled into the bottom of the rail, and passes back up to the pulsing transducer.*

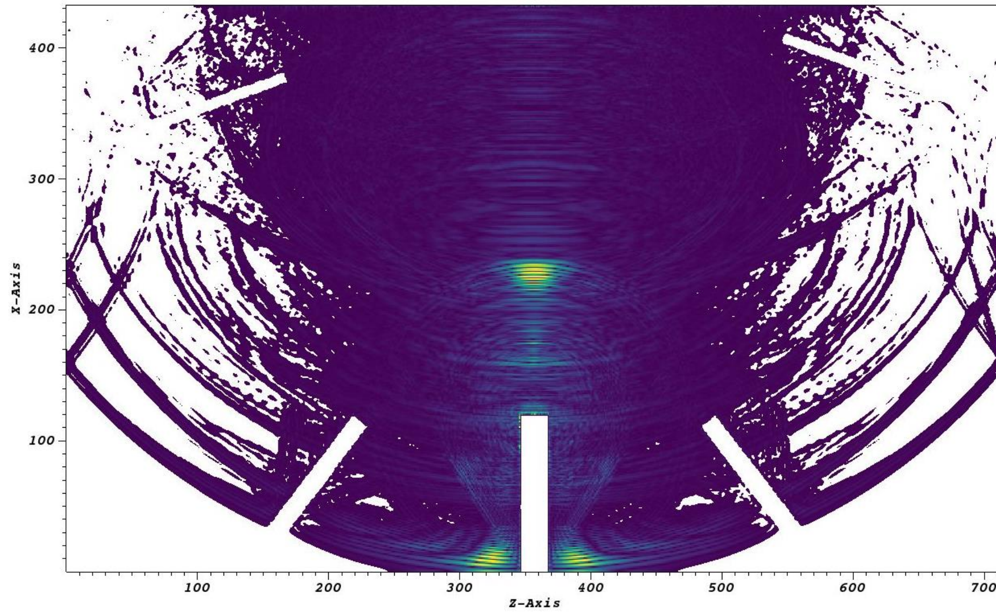


*Figure 15: EFIT Case 3 – beam in rail, 0º, 2-D slice (longitudinal). In this view, the rail is sliced in half lengthwise at t=31.655 µs. The pulse can be seen reflecting from the flat-bottom hole. Other drilled holes in the rail are seen outlined by the outer edges of the wave.*

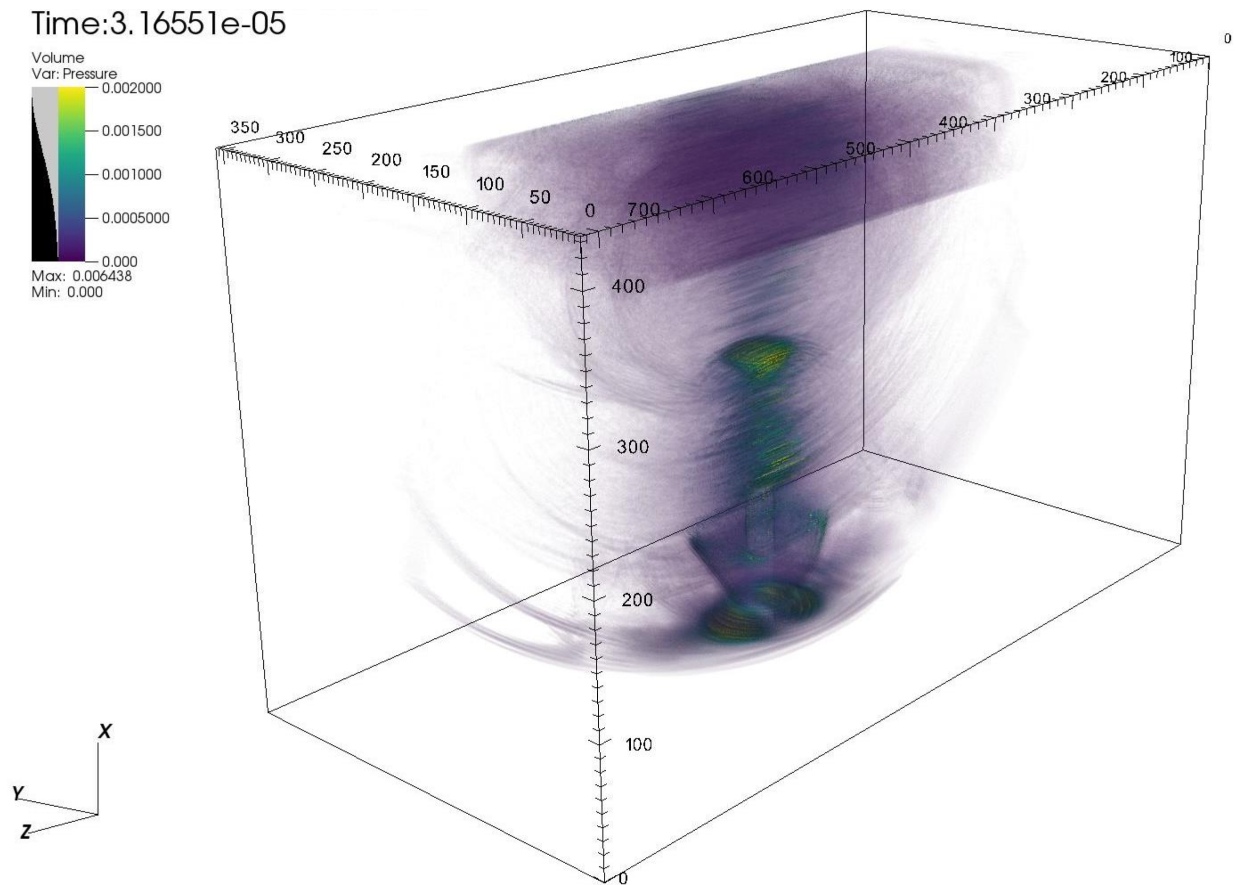


*Figure 16: EFIT Case 3 – beam in rail, 0º, 3-D view. The head of the rail can be seen slowly filling with scattered reflections from the main wave's path.*

EFIT Cases 4-6 are various angle beams which must be interpreted with an important caveat. To understand that caveat, we must first understand mode conversion and refraction. When an incident longitudinal wave passes between media with different sound speeds (more generally, with different characteristic impedances) at an angle, several things happen. First, a longitudinal wave propagates into the second medium at a refracted angle determined by the ratio of the sound speeds. Depending on the incident angle a shear wave may also be created and propagate in the second medium at a different angle. Since the shear wave speed is always less than the longitudinal wave speed, this angle is always less than the longitudinal wave (Figure 17).

This incident angle at which the refracted longitudinal wave becomes greater than 90˚ and only the shear wave propagates in the medium is known as the first critical angle, and the incident angle at which the shear wave no longer propagates is defined as the second critical angle. For incident angles beyond this, surface waves may be created [1]. In applied UT, most angle beams are positioned past the first critical angle so that only the shear wave propagates in the material under test to simplify signal interpretation. Unfortunately, the time delay beam steering method cannot create an input wave beyond the first critical angle. This means that the results for EFIT Cases 4-6 are not truly representative of the application; in real testing, only a shear wave would be present, and not a shear wave and a longitudinal wave. However, the results are no less physical because of this limitation – if a shallower incident angle was used in the UT wheel probe, it would create the waves shown in the following results.

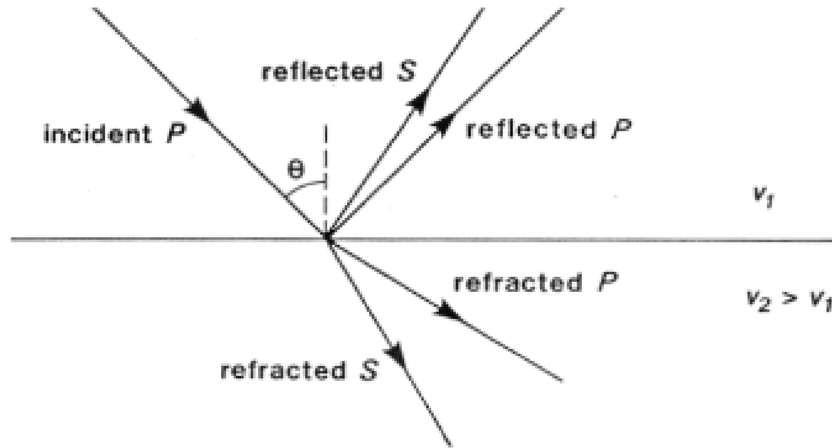


*Figure 17: Snell's Law for acoustics in two solids. The angles of the incident longitudinal (P) wave and the reflected P wave are the same. The angle of the reflected shear (S) wave is determined by a ratio of the shear and longitudinal acoustic velocities of Medium 1 (upper material). The angles of the refracted P and S waves are determined by the ratios of acoustic velocities of Medium 1 and Medium 2. From* [23].

The transducer for each of EFIT Cases 4-6 is a ¾-inch, 2.25 MHz 1-period sine pulse at the top of the rail with an offset to align with the appropriate drilled hole. EFIT Case 4 is angled at 56.4º, Case 5 at 37.5º, and Case 6 at 70º from normal. In each of these simulations, a longitudinal wave can be seen reflecting from the aligned flat-bottom hole, while a shear wave propagates slower and at a smaller angle (Figure 19-Figure 23).

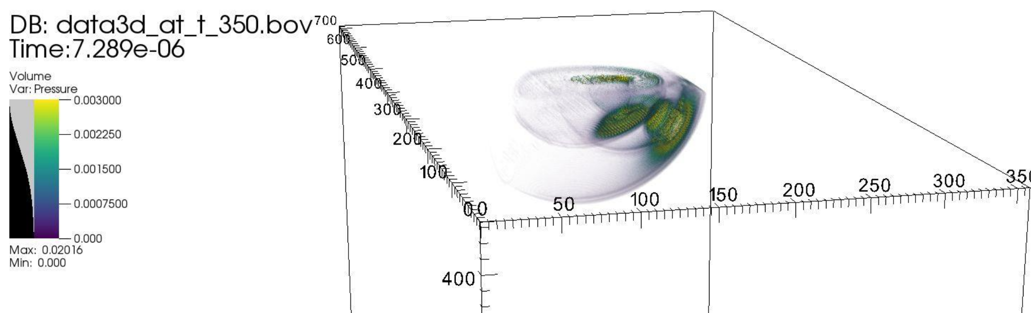


*Figure 18: EFIT Case 4 – beam in rail, 56.4º, 3-D view. Though the center of the longitudinal wave is reflecting from the flat-bottom hole, the outer region of the wavefront continues propagating past the hole.*

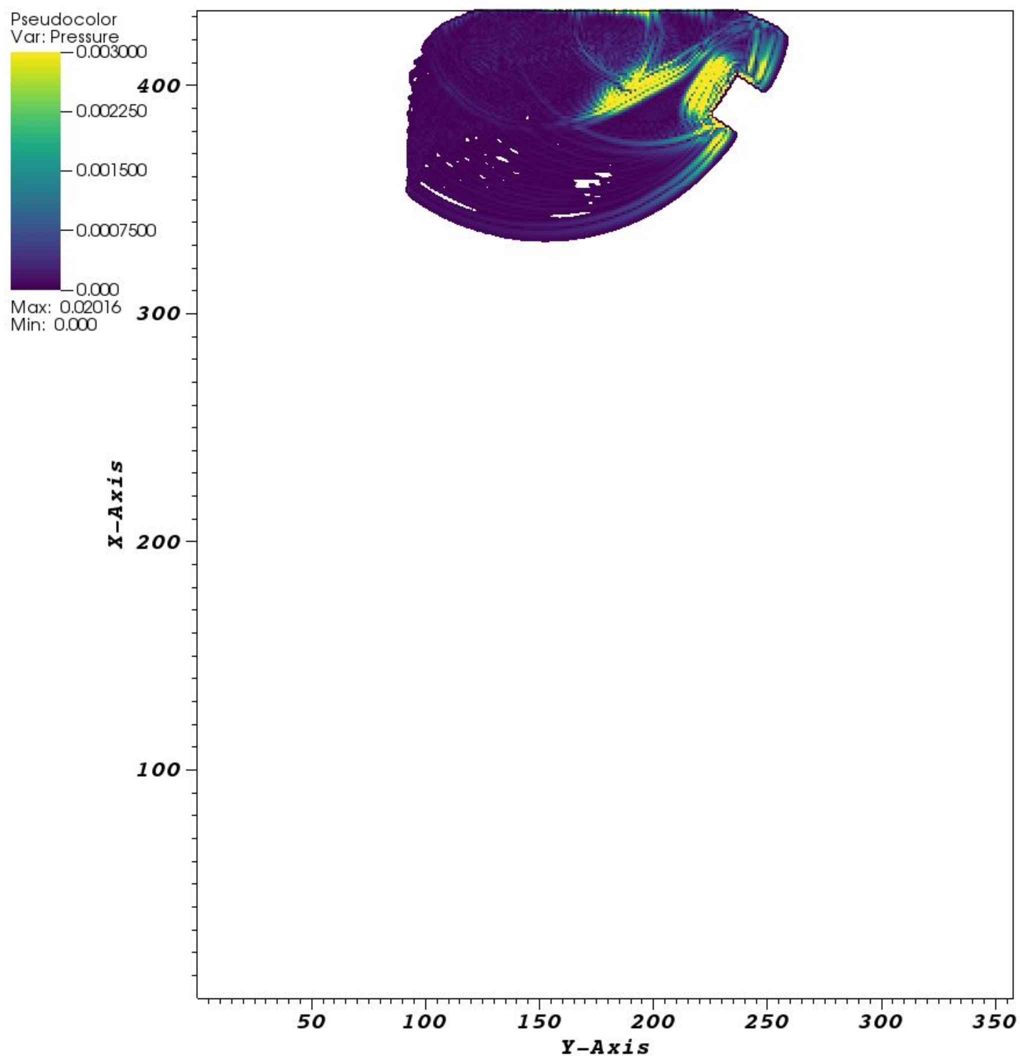


*Figure 19: EFIT Case 4 – beam in rail, 56.4º, 2-D slice. The longitudinal wave can be seen reflecting directly from an angled flat-bottom hole drilled in the rail head. This transducer is used in practice to inspect the sides of the rail head for near-vertical defects.*

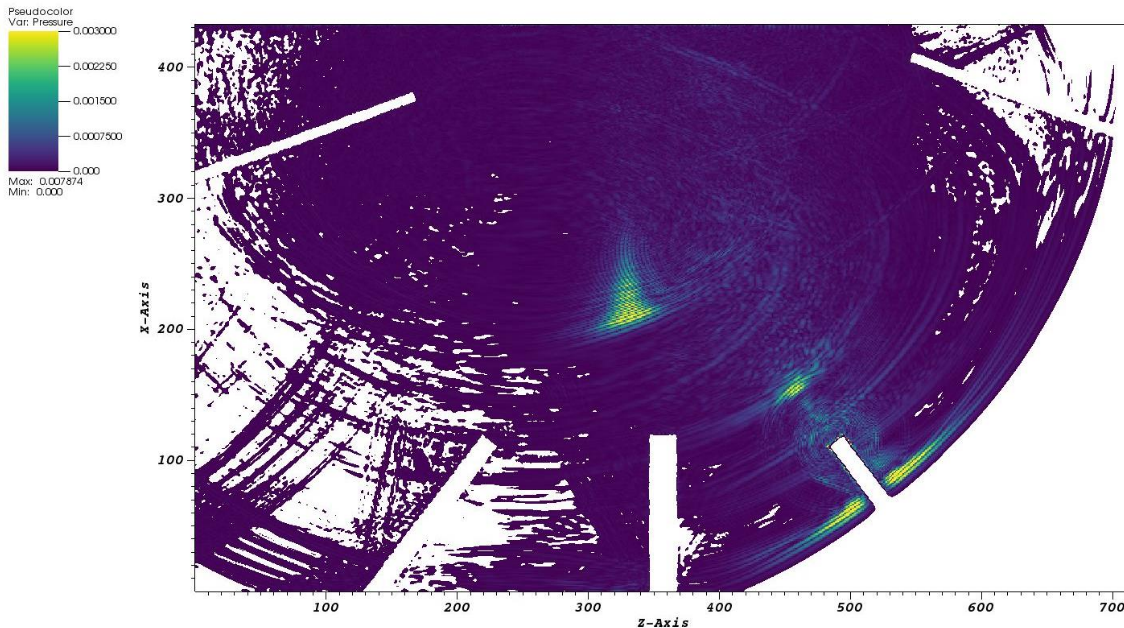


*Figure 20: EFIT Case 5 – beam in rail, 37.5º, 2-D slice (longitudinal). The longitudinal wave can be seen reflecting from a long, narrow flat-bottom hole drilled at an angle into the bottom of the rail. By happenstance, the shear wave is propagating directly toward the center-bottom hole.*

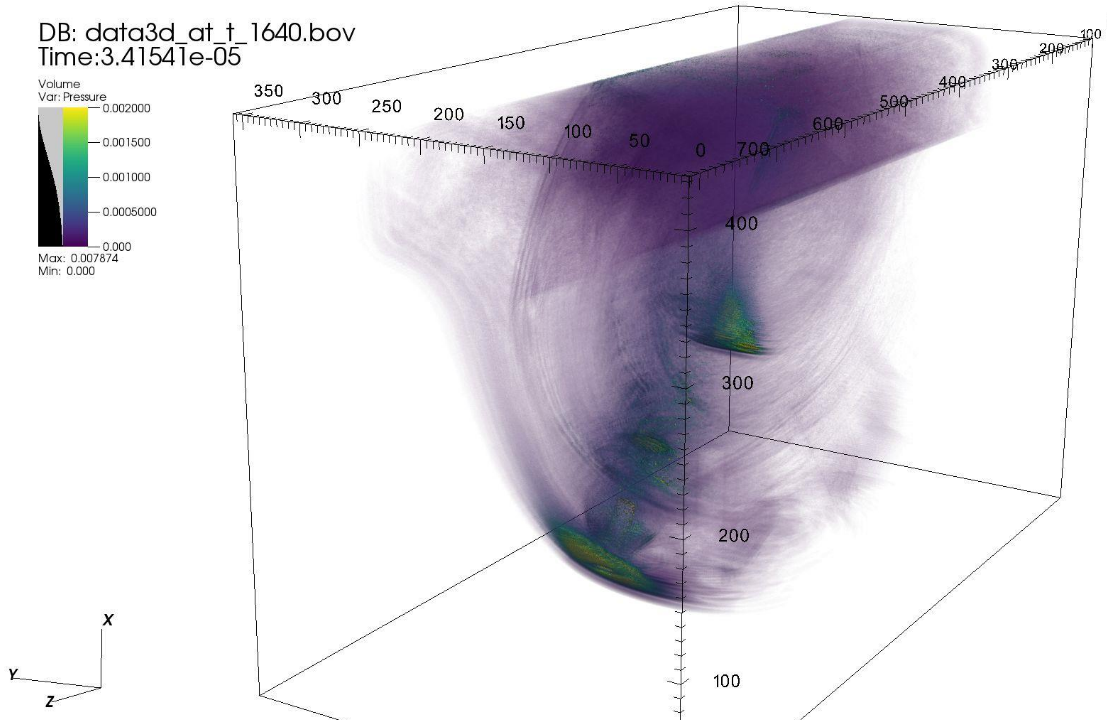


*Figure 21: EFIT Case 5 – beam in rail, 37.5º, 3-D view. Though the center of the longitudinal wave is reflecting from the flat-bottom hole, the outer region of the wavefront continues propagating past the hole. The low-energy far outer regions of the wavefront can be seen filling the rail with small reverberations.*

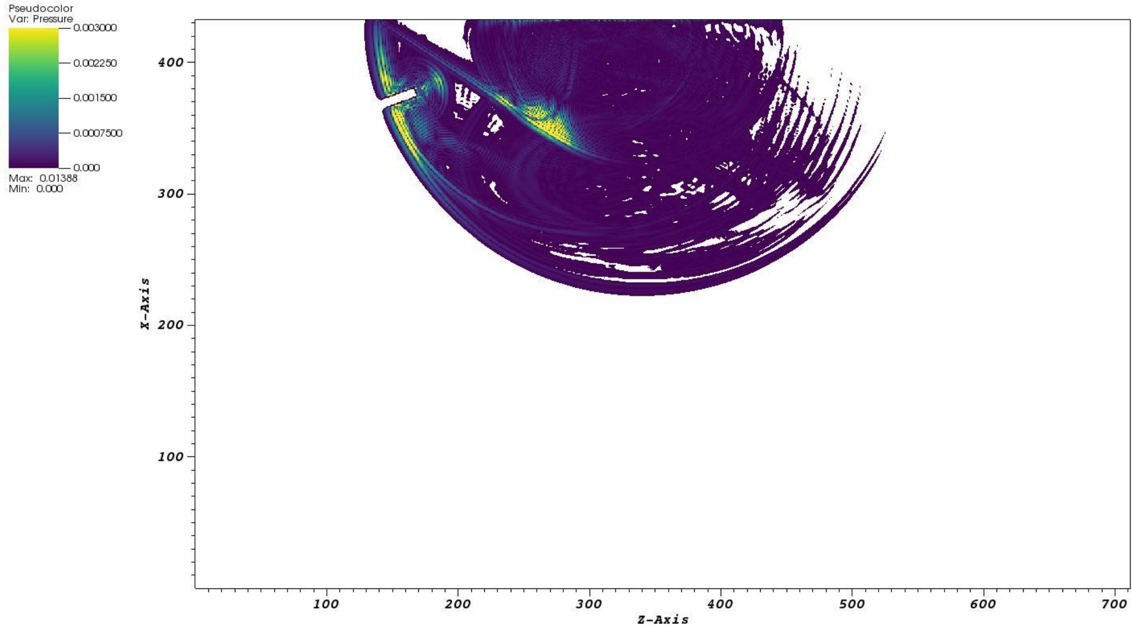


*Figure 22: EFIT Case 6 – beam in rail, 70º, 2-D slice (longitudinal). The longitudinal wave can be seen reflecting from a long, narrow flat-bottom hole drilled at an angle into the end of the rail.*

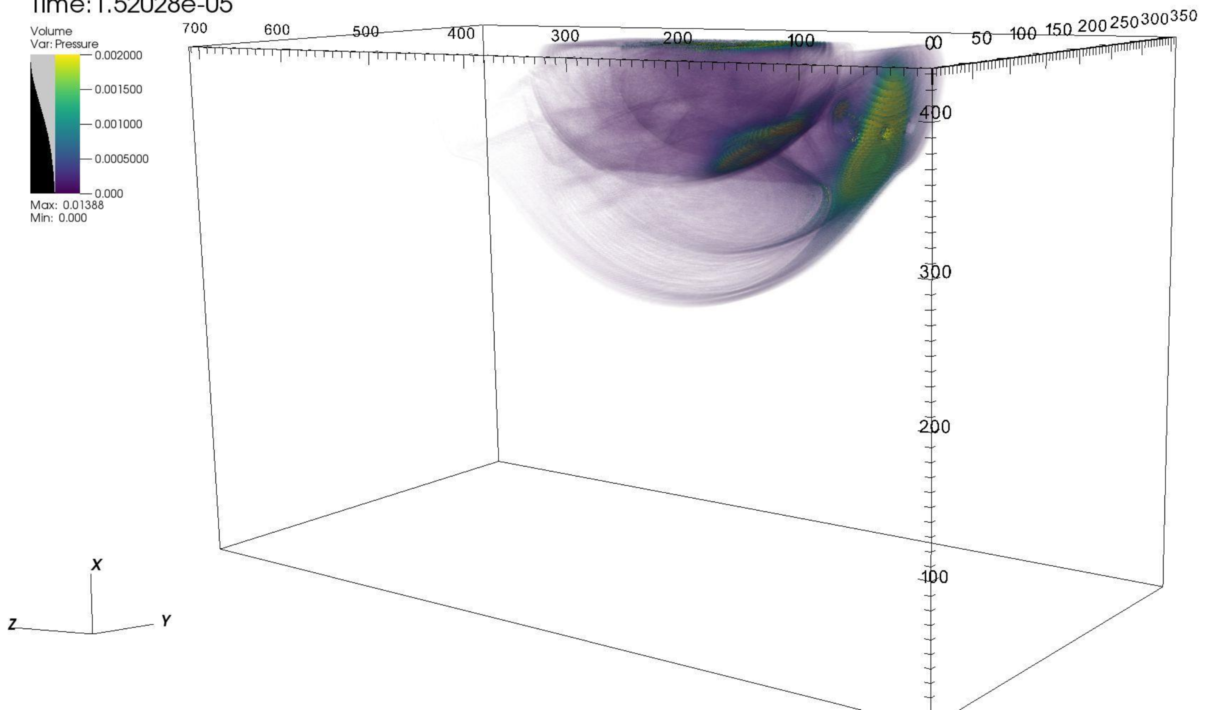


*Figure 23: EFIT Case 6 – beam in rail, 70º, 3-D view. Though the center of the longitudinal wave is reflecting from the flat-bottom hole, the outer region of the wavefront continues propagating past the hole.*

# CHAPTER IV: Conclusions and Future Work

The primary goals of this thesis were achieved: development of a tool for simulating, visualizing, and analyzing ultrasonic wave propagation in arbitrary geometry. However, there are still many opportunities for improvements and further exploration. Two features were implemented in this tool with only partial success, and thus were not included in the results chapter: multi-node transducer definition and transducer curvature mapping. Multi-node transducer definition passes information about transducer definition, drive function, time delays for beam steering, etc. from the processor node which contains the transducer center point to any other nodes that the transducer spans on the spatial domain. The drive function was able to execute correctly, but at times late in the simulation, linear time-growth instability artifacts begin to appear near the nodal boundaries of the transducer space. Very similar instabilities occur when applying transducer curvature mapping, which drops each transducer element “down” from the top of the simulation space until it reaches the top of the material geometry, thus projecting the transducer onto the surface. Both features would be beneficial, but their exclusion can be overcome for many test cases, including those in this thesis. Multi-node transducer definition can be avoided by simply examining the size and location of the transducer and planning the number of processor nodes used accordingly. It limits the speed of the simulation but does not affect the results – and as noted in EFIT Case 3, the longest simulation run time was only 3 hours. Transducer curvature mapping can be avoided by trimming the top of the simulation geometry before exporting it from the modeling software, such that the top surface of the geometry ends up being coplanar with the top face of the simulation space (which is where the transducer is defined, by default). This approach requires a small amount of additional setup work and is a minor approximation but does not significantly impact results for the geometry in EFIT Case 3.

Other features were not attempted in this implementation that could add considerable value. For instance, a multiphysics AFIT/EFIT simulation with transducers defined within a fluid region would be a more accurate representation of what is physically happening in a UT wheel probe. This would allow multiple angle beams to be focused on one area of the fluid-solid boundary, enabling simulation of the interactions of several ultrasonic beams fired at the same time. One of the largest limitations that would be eliminated is the lack of true refraction and mode conversion in the current implementation. Additionally, this method would be ideal for defining transducers at arbitrary locations and angles, eliminating the need to use time delay beam steering.

Beyond adding features, an important step would be performing experimental validation of the simulation results. For the tool to be truly useful as a development aid, the results must be tested against real-world experiments to build confidence in the implementation. This could be done first for very simple setups like EFIT Case 2 to prove the basic functionality, then expanded to more complex tests like EFIT Cases 3-6. The input parameter would be recorded at the center of the transducer in the simulation and scaled using the drive function to match the experimental data.

Though the simulation tool has limited applicability to R&D as it stands, there are many ways it can be used. The visualizations created in VisIt give great clarity to a process that is invisible to human senses – it may add significant value to educational materials on UT and the science of acoustics. In fact, the video results are so intuitive that it is possible to use them as communication tools to explain ultrasonic phenomena to non-technical parties who do not have formal education in the field. Most importantly, though, the code can be used as a basis for future research and exploration in the field of numerical simulation of acoustics.

# APPENDIX

MATLAB Code: Input File Generation

```
% EFIT_cart_3D_inputfiles.m
% Generates configuration files for EFIT simulations

%% Material Parameters
% Steel, 1020
% den = 7710; % kg/m^3
% cL = 5890; % m/s
% cT = 3240; % m/s
% mu = den*cT^2; % Pa
% lambda = den*cL^2-2*mu; % Pa

% Aluminum
den = 2698; % kg/m^3
cL = 6235; % m/s 6374
cT = 3139; % m/s 2906
mu = den*cT^2; % Pa
lambda = den*cL^2-2*mu; % Pa

% Brass
% den = 8400; % kg/m^3
% cL = 4400; % m/s
% cT = 2200; % m/s
% mu = den*cT^2; % Pa
% lambda = den*cL^2-2*mu; % Pa

cmax = cL;
cmin = cT;
clear cL cT

%% Simulation Parameters
fmax = 1.5*10^6; % max frequency (Hz)
wavelength=cmin/fmax;
ds=.7*wavelength/6.1 % step size (m) -> here we assign number of
points per wavelength (>6)

% Size of simulation space (in mm):
z_mm = 100; % 304 for rail
y_mm = 50; % 152 for rail
x_mm = 50; % 152 for rail

% Convert size from m to steps: -> must be even!
maxz = ceil(z_mm/(1000*ds))
maxy = ceil(y_mm/(1000*ds))
maxx = ceil(x_mm/(1000*ds))

maxt = 501; % max simulation time in steps (+1)
outputevery = 5; % output 3D volume how often
dt = 1/(cmax*sqrt(3/(ds^2))) % time step (s)
```

```
%% Choose a flaw type (everything should be in number of steps)

% Arbitrary fullspace scatterer:
% nS=1;                   % numref // numbers scatterers
% rftype=[5];             % typ    // Reflector type: 5 - arbitrary
(fullspace)
% nsx1  =[0];             % p1     // not used
% nsx2  =[0];             % p2     // not used
% nsx3s =[0];             % start3 // not used
% nsx3e =[0];             % end3   // not used
% rrad  =[maxz*maxy*maxx]; % rad    // not used
% rden  =[-1];             % dd     // not used
% rmu   =[-1];             % mu     // not used
% rlambda= [-1];           % lambda // not used
% % inputs: filename, ds, adjustment of origin (xyz in m), space size
(xyz in m)
% [arbscatt, ~] = import_stl_scatterer('Test Block Curved Top x-50 y-
50 z-100.STL', ds, [0 0 0], [x_mm/1000 y_mm/1000 z_mm/1000]);
% if(size(arbscatt) ~= [maxy maxx maxz])
%     disp('Error in Arbitrary scatterer definition. Terminating
script.')
%     return
% end

% Rectangular:
nS     = 0;                       % numref // number of scatterers
rftype = [3];                     % typ    // Reflector type: 3 - Right
Rectangular Prism
nsx1   = [round(25/(1000*ds))]-1;  % p1     // z-start (z-end will be
to side of space)
nsx2   = [round(75/(1000*ds))]-1;  % p2     // z-end
nsx3s  = [round(10/(1000*ds))]-1;  % start3 // y-start
nsx3e  = [round(60/(1000*ds))]-1;  % end3   // y-end
rrad   = [round(20/(1000*ds))]-1;  % rad    // x-start
rden   = [round(24/(1000*ds))]-1;  % maxx-1; % dd     // x-end
rmu    = [0];                     % mu     // null
rlambda= [0];                     % lambda // null
if((nsx1>nsx2) || (nsx3s>nsx3e) || (rrad>rden))
    disp('Error in Rectangular scatterer definition. Terminating
script.')
    return
end

%% Transducer info (everything in number of steps)
ntrans = 1;
transducer_z = [round(50/(1000*ds))]; % transducer z position
transducer_y = [round(25/(1000*ds))]; % transducer y position
transducer_x = [maxx-1]; % transducer x position; (maxx - 1) is
positioned on top
transducer_rad = [round(10.7/(1000*ds))]; % transducer radius
```

```
transducer_theta = deg2rad(70); % Angle of desired beam path from
normal, rad
transducer_phi = deg2rad(0); % Angle of desired beam path from +z
direction, rad

dffreq = fmax;  % Pulse Frequency (Hz)
cycles = 1; % Pulse cycles
dfpulselen = cycles*(1/dffreq);  % number of cycles times single pulse
length (seconds)

df(1:maxt) = 0;
dfl = ceil(dfpulselen/dt);

amplitude = 10^6;
% df(1:dfl) = amplitude*sin((0:(dfl-1))*dt*dffreq*2*pi);  % sine wave
duty = 50;
df(2:(dfl+1)) = (0.5*amplitude)+0.5*amplitude*square((0:(dfl-
1))*dt*dffreq*2*pi,duty);  % square wave, needs to start at 0

drivelen = length(df);

%% Write files
%======================================================================
=====================
% Write Inputfiles for simulation - % DO NOT CHANGE THE ORDER OF THIS
PART
%======================================================================
=====================
[fname,pname] = uiputfile('in.file', 'Save Configuration');
fp=fopen('in.file','w');

fprintf(fp, ' %8.0f ' , maxz);      % simparams[0] - num1 (will be +2)
fprintf(fp, ' %8.0f ' , maxy);      % simparams[1] - num2
fprintf(fp, ' %8.0f ' , maxx);      % simparams[2] - num3
fprintf(fp, ' %2.20f ', ds);        % simparams[3] - ds
fprintf(fp, ' %2.20f ', dt);        % simparams[4] - dt

fprintf(fp, ' %15.6f ', den);       % simparams[5] - den
fprintf(fp, ' %15.6f ', lambda);    % simparams[6] - lm
fprintf(fp, ' %15.6f ' , mu);       % simparams[7] - mu
fprintf(fp, ' %8.0f ' , maxt);          % maxt
fprintf(fp, ' %8.0f ' , outputevery);   % outputevery

fprintf(fp, ' %8.0f ' , nS);            % numref
for i = 1:nS                                              %
addReflector
    fprintf(fp, ' %8.0f ' ,  rftype(i));    % rpars[0]        typ
    fprintf(fp, ' %8.0f ' , nsx1(i));      % rpars[1]        p1
    fprintf(fp, ' %8.0f ' , nsx2(i));      % rpars[2]        p2
    fprintf(fp, ' %8.0f ' ,  nsx3s(i));     % rpars[3]        start3
    fprintf(fp, ' %8.0f ' ,  nsx3e(i));     % rpars[4]        end3
    fprintf(fp, ' %8.0f ' , rrad(i));      % rpars[5]        rad
```

```
    fprintf(fp, ' %15.6f ' , rden(i));       % rpars[6]        dd
    fprintf(fp, ' %15.6f ' , rmu(i));        % rpars[7]        mu
    fprintf(fp, ' %15.6f ' , rlambda(i));    % rpars[8]        lambda
end

%fprintf(fp, ' %s ', [ pname ]);        % working directory
fclose(fp);


[fname,pname] = uiputfile('trans.file', 'Save Configuration');
fp=fopen('trans.file','w');
fprintf(fp, ' %8.0f ' , ntrans);                     % numtrans
for i=1:ntrans
    fprintf(fp, ' %8.0f ' , transducer_z(i) );      % tparams[0] //
tposz; // transducer z location
    fprintf(fp, ' %8.0f ' , transducer_y(i) );      % tparams[1] //
tposy; // transducer y location
    fprintf(fp, ' %8.0f ', transducer_x );               % tparams[2]
// tposx; // transducer x location  --> because always on top
    fprintf(fp, ' %8.0f ', transducer_rad(i));     % tparams[3] //
trad;  // transducer radius
    fprintf(fp, ' %8.0f ', drivelen);           % tparams[4] //
drivelen; // length of drive function

    fprintf(fp, ' %15.6f ' , transducer_theta); % tparams[6] -
transducer_theta
    fprintf(fp, ' %15.6f ' , transducer_phi);   % tparams[7] -
transducer_phi

    fprintf(fp, ' %15.6f ', df(1:maxt));            % drive[i] where i
= tparams[4] MUST BE ALTERED TO INCLUDE MULTIPLE Drive functions
end

fclose(fp);

% [fname,pname] = uiputfile('arbscatt.file', 'Save Configuration');
% fp=fopen('arbscatt.file','w');
% for iz=1:maxz
%     for iy=1:maxy
%         for ix=1:maxx
%             fprintf(fp, ' %d', arbscatt(iy,ix,iz));
%         end
%     end
% end
%
% fclose(fp);
```

MATLAB Code: Arbitrary Geometry Importing and Processing

```
% import_stl_scatterer.m
% Imports STL file and creates array of scattering boundary for use in
FIT sims
% stlread based on 'cad2matdemo'
%
% Dependencies: INPOLYHEDRON, drawMesh (part of the geom3d package)

function [arbscatt, isinside] = import_stl_scatterer(filename, ds,
objorigin, spacesize)

[faces, vertices, ~] = stlread(filename); % import STL file - origin
must be correct and scale in mm

transverts = vertices; % vertices translated to new origin (and y-
corrected)
for idx = 1:3
   transverts(:,idx) = transverts(:,idx) + objorigin(idx)*1000; % move
object origin (given in m)
   if max(transverts(:,idx)) > spacesize(idx)*1000 ||
min(transverts(:,idx)) < (0-ds/2) % ensure all values are within the
simspace
       disp('Scatterer placement not possible - outside of space');
       return;
   end
end

ssverts = vertices; % convert from mm scale to simspace scale (using
ds)
for idx = 1:3
   ssverts(:,idx) = transverts(:,idx)/(ds*1000);
end

% Find points inside this object - to save memory, focus only on space
where scatterer is located
scatstart = floor(min(abs(ssverts)));
scatend = ceil(max(ssverts));
scatspace = scatend-scatstart+1; % for easier indexing and sizing
isinside = inpolyhedron(faces, ssverts, scatstart(1):1:scatend(1),
scatstart(2):1:scatend(2), scatstart(3):1:scatend(3));

% % View results:
% [xgrid,ygrid,zgrid] = meshgrid(scatstart(1):1:scatend(1),
scatstart(2):1:scatend(2), scatstart(3):1:scatend(3));
% figure; hold on;
% plot3(ygrid(isinside), xgrid(isinside),
zgrid(isinside),'bo','MarkerFaceColor','b');
% % plot3(ygrid(~isinside), xgrid(~isinside), zgrid(~isinside),'ro'),
axis image; hold off;
% xlabel('Y'); ylabel('X'); zlabel('Z');
```

```
% clear xgrid ygrid zgrid;

% Now create logical array for entire space as input to simulation
fullspace = [ceil(spacesize(2)/ds), ceil(spacesize(1)/ds),
ceil(spacesize(3)/ds)]; % size of full space (steps)
arbscatt = false(fullspace); % this could get big quickly - use
logical to reduce size

% Cuts off last face in each dimension (vertices --> nodes)
if (size(isinside,2)==(scatend(1)-scatstart(1)+1) &&
(size(isinside,2)-1)==fullspace(2)) % nothing got switched around,
sizes are correct

arbscatt(scatstart(2)+1:scatend(2),scatstart(1)+1:scatend(1),scatstart
(3)+1:scatend(3)) = isinside(1:scatspace(2)-1,1:scatspace(1)-
1,1:scatspace(3)-1);
else
    disp('Something wrong with logical array');
end

% % View results for whole space - MAY BE BIG!!
% [xgrid,ygrid,zgrid] =
meshgrid(1:1:fullspace(2),1:1:fullspace(1),1:1:fullspace(3));
% figure; hold on;
% plot3(ygrid(arbscatt), xgrid(arbscatt),
zgrid(arbscatt),'bo','MarkerFaceColor','b');
% plot3(ygrid(~arbscatt), xgrid(~arbscatt), zgrid(~arbscatt),'ro');
hold off;
% xlabel('Y'); ylabel('X'); zlabel('Z');

function [fout, vout, cout] = stlread(filename)
% Reads ASCII stl file and returns a vertex list and face list for
Matlab patch command

fid=fopen(filename, 'r'); %Open the file, assumes STL ASCII format.
if fid == -1
    error('File could not be opened, check name or path.')
end

% STL files of form:
%
%solid BLOCK
%  color 1.000 1.000 1.000
%  facet
%      normal 0.000000e+00 0.000000e+00 -1.000000e+00
%      normal 0.000000e+00 0.000000e+00 -1.000000e+00
%      normal 0.000000e+00 0.000000e+00 -1.000000e+00
%    outer loop
%      vertex 5.000000e-01 -5.000000e-01 -5.000000e-01
%      vertex -5.000000e-01 -5.000000e-01 -5.000000e-01
%      vertex -5.000000e-01 5.000000e-01 -5.000000e-01
%    endloop
```

```
% endfacet
%
% The first line is object name, then comes multiple facet and vertex
lines.
% A color specifier is next, followed by those faces of that color,
until
% next color line.
%
CAD_object_name = sscanf(fgetl(fid), '%*s %s');  %CAD object name, if
needed.
%                                                %Some STLs have it,
some don't.
vnum=0;       %Vertex number counter.
report_num=0; %Report the status as we go.
VColor = 0;
%
while feof(fid) == 0                    % test for end of file, if not
then do stuff
    tline = fgetl(fid);                 % reads a line of data from
file.
    fword = sscanf(tline, '%s ');       % make the line a character
string
% Check for color
    if strncmpi(fword, 'c',1) == 1     % Checking if a "C"olor line,
as "C" is 1st char.
       VColor = sscanf(tline, '%*s %f %f %f'); % & if a C, get the RGB
color data of the face.
    end                                % Keep this color, until the
next color is used.
    if strncmpi(fword, 'v',1) == 1     % Checking if a "V"ertex line,
as "V" is 1st char.
       vnum = vnum + 1;                % If a V we count the # of V's
       report_num = report_num + 1;    % Report a counter, so long
files show status
       if report_num > 249
           fprintf('Reading vertex num: %d.\n',vnum);
           report_num = 0;
       end
       v(:,vnum) = sscanf(tline, '%*s %f %f %f'); % & if a V, get the
XYZ data of it.
       c(:,vnum) = VColor;              % A color for each vertex,
which will color the faces.
    end                                 % we "*s" skip the name
"color" and get the data.
end
%   Build face list; The vertices are in order, so just number them.
%
fnum = vnum/3;      %Number of faces, vnum is number of vertices.  STL
is triangles.
flist = 1:vnum;     %Face list of vertices, all in order.
F = reshape(flist, 3,fnum); %Make a "3 by fnum" matrix of face list
data.
```

```
%
%   Return the faces and vertexs.
%
fout = F';  %Orients the array for direct use in patch.
vout = v';  % "
cout = c';
%
fclose(fid);
```

C++ Code: ‘efit.cpp’

```
/* efit.cpp
*
* Sean M. Raley (UNH)
* based on code by Eric A. Dieckman (WM)
* Last edited: 27 March 2019 SMR
*/

//#include "pch.h" // for Microsoft VS only

#include <mpi.h>
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <time.h>
#include <math.h>
#include "space.h"

using namespace std;

void master();
void slave();
int* DistributeSimulationParameters();
void DistributeTransducers(int *xposs);
void dump3Dbin(int t);

int mpirank, numworkers;
int maxt, outputevery, maxz, numtransducers;
int* EvenVolDims = new int[3];

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

	if (mpirank == 0)
		master();
	else
		slave();

	MPI_Finalize();
	return 0;
}
```

```
//
======================================================================
====
// Master node! -- Distributes simulation space and receives data for
output
//
======================================================================
====
void master() {
	time_t start, end;
	time(&start);

	int *zstartpos = new int[numworkers];

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

	zstartpos = DistributeSimulationParameters(); // Initialize each
node
	DistributeTransducers(zstartpos);

	// writes text file for binary->bov bash command
	string fname = "bin2bovCmd.ascii";
	ofstream outFile(fname.c_str(), ios::out);
	outFile<<"bash makebovs ";
	outFile<<maxt<<" ";
	outFile<<outputevery<<" ";
	for(int e=2;e>=0;e--)
		outFile<<EvenVolDims[e]<<" ";
	outFile.close();

	for (int t = 0; t < maxt; t++) {
		if (t%outputevery == 0 && outputevery != 1) {
			//dump3Dascii(t);
			//cout << "Saved pressure data as ASCII at time: " <<
t << "\n";
			dump3Dbin(t);
			cout << "Saved pressure data as binary at time: " << t
<< "\n";
			//dump3Dvtk(t);
			//cout << "Saved pressure data as vtk at time: " << t
<< "\n";
		}
	}

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

//
======================================================================
==
```

```
// Slave node! -- Does the grunt work
//
======================================================================
==
void slave() {
      // --- Receive sim parameters from master and initialize ---
      MPI_Status  status;
      MPI_Status status1;
      MPI_Status status2;
      MPI_Status status3;
      MPI_Status status4;
      MPI_Status status5;
      MPI_Status status6;
      MPI_Request request1;
      MPI_Request request2;
      MPI_Request request3;
      MPI_Request request4;
      MPI_Request request5;
      MPI_Request request6;

//    int tosend0 = 0; // to use when sending a 0 in MPI is desired

      double simparams[11];
      MPI_Recv(&simparams, 11, MPI_DOUBLE, 0, 201, MPI_COMM_WORLD,
&status);

      space simspace(simparams);
      maxt         = static_cast<int>(simparams[9]); // total number of
time steps
      outputevery   = static_cast<int>(simparams[10]); // output every
this many time steps
      double cL = sqrt((simparams[6]+2*simparams[7])/simparams[5]); //
default speed of sound, calculated

      if (mpirank == 1){ // node is on left
            simspace.type = 1;
            cout<<"slave is type "<<simspace.type<<endl;
      }
      else if (mpirank == numworkers){ // node is on right
            simspace.type  = 3;
            cout<<"slave is type "<<simspace.type<<endl;
      }
      else{
            simspace.type  = 2; // node is in middle
            cout<<"slave is type "<<simspace.type<<endl;
      }

      // -- Receive reflector parameters ---
      int nr;
      double *rpars = new double[9];
      MPI_Recv(&nr, 1, MPI_INT, 0, 203, MPI_COMM_WORLD, &status);
```

```
      for (int i = 0; i < nr; i++) {
            MPI_Recv(&rpars[0], 9, MPI_DOUBLE, 0, 204, MPI_COMM_WORLD,
&status);
            simspace.addReflector(rpars[0], rpars[1], rpars[2],
static_cast<int>(rpars[3]), static_cast<int>(rpars[4]), rpars[5],
rpars[6], rpars[7],rpars[8]);
      }
      delete[] rpars;
      cout << "num_z in slave(mpirank="<<mpirank<<") is: " <<
simspace.num_z << endl;

      // --- Receive transducer parameters ---
      double tparams[8];
      bool done = false;

      while (done == false){
            MPI_Recv(&tparams, 8, MPI_DOUBLE, 0, 211, MPI_COMM_WORLD,
&status);
            int drvlen=static_cast<int>(tparams[4]);
            int min_tau_local, min_tau_global;

            if (tparams[0] == -1) done = true;
            else{
                  transducer
t(tparams[0],tparams[1],tparams[2],tparams[3],static_cast<int>(tparams
[5]),maxt,tparams[6],tparams[7],simspace.num_y,simspace.num_z,simspace
.zbeg,simspace.dtods);

                  if (drvlen > 0){
                        double *drive = new double[drvlen];
                        MPI_Recv(&drive[0], drvlen, MPI_DOUBLE, 0, 212,
MPI_COMM_WORLD, &status);
                        min_tau_local = t.setDelays(cL);
                        MPI_Send(&min_tau_local, 1, MPI_INT, 0, 213,
MPI_COMM_WORLD);
                        MPI_Recv(&min_tau_global, 1, MPI_INT, 0, 214,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

                        t.setDriveFunction(drvlen,drive,min_tau_global);
                  }
                  simspace.addTransducer(t);
            }
      }

      // --- Run simulation ---
      for (int t = 0; t < maxt; t++) {
            if (mpirank == 1 && t%10==0)
                  cout << " timestep: " << t << "      " <<
simspace.num_z << ", " << simspace.num_y << ", " << simspace.num_x <<
endl;
            simspace.drivetime = t;
          simspace.UpdateTransducers(t);
```

```
        if (t%outputevery == 0) { // sends output to master node
                int len = simspace.GetEvenVolLen(); // get even num_z
length for each node
                double* x = simspace.GetEvenVol(len);

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

                delete[] x;
        }

        // --- Update V's ---
        simspace.UpdateVs(1,1);                        // Update left
boundary
        simspace.UpdateVs(simspace.num_z-2,simspace.num_z-2);
    // Update right boundary

        if (mpirank>1){                                     // send
left
         MPI_Isend(&simspace.vy[simspace.Lyx], simspace.Lyx,
MPI_DOUBLE, (mpirank-1), 301, MPI_COMM_WORLD, &request1);
         MPI_Isend(&simspace.vx[simspace.Lyx], simspace.Lyx,
MPI_DOUBLE, (mpirank-1), 302, MPI_COMM_WORLD, &request2);
        }

        if (mpirank<numworkers){                       // send  right
         MPI_Isend(&simspace.vz[(simspace.num_z-2)*simspace.Lyx],
simspace.Lyx, MPI_DOUBLE, (mpirank+1), 303, MPI_COMM_WORLD,
&request3);
        }

        simspace.UpdateVs(2,simspace.num_z-3);  // update inner nodes

        if (mpirank<numworkers){                       // receive  from
right
                MPI_Recv(&simspace.vy[(simspace.num_z-
1)*simspace.Lyx], simspace.Lyx, MPI_DOUBLE, (mpirank+1), 301,
MPI_COMM_WORLD, &status1);
                MPI_Recv(&simspace.vx[(simspace.num_z-
1)*simspace.Lyx], simspace.Lyx, MPI_DOUBLE, (mpirank+1), 302,
MPI_COMM_WORLD, &status2);
        }

        if (mpirank>1){                                     // receive
from left
         MPI_Recv(&simspace.vz[0], simspace.Lyx, MPI_DOUBLE,
(mpirank-1), 303, MPI_COMM_WORLD, &status3);
        }
```

```
          if (mpirank>1){                                      // Wait
for sends to complete
           MPI_Wait(&request1,&status1);
           MPI_Wait(&request2,&status2);
          }

          if (mpirank<numworkers){
            MPI_Wait(&request3, &status3);
          }

          // --- Update T's ---
          simspace.UpdateTs(1,1);                          // Update left
boundary
          simspace.UpdateTs(simspace.num_z-2,simspace.num_z-2);    //
Update right boundary

          if (mpirank>1){                                  // send Trz,
Tzp left
           MPI_Isend(&simspace.T11[simspace.Lyx], simspace.Lyx,
MPI_DOUBLE, (mpirank-1), 311, MPI_COMM_WORLD, &request4);
          }

          if (mpirank<numworkers){                         // send Tzz,
Trp right
           MPI_Isend(&simspace.T12[(simspace.num_z-2)*simspace.Lyx],
simspace.Lyx, MPI_DOUBLE, (mpirank+1), 313, MPI_COMM_WORLD,
&request5);
           MPI_Isend(&simspace.T13[(simspace.num_z-2)*simspace.Lyx],
simspace.Lyx, MPI_DOUBLE, (mpirank+1), 314, MPI_COMM_WORLD,
&request6);
          }

          simspace.UpdateTs(2,simspace.num_z-3);               // update
inner nodes

          if (mpirank<numworkers){                         // reveive
Trz, Tzp from right
           MPI_Recv(&simspace.T11[(simspace.num_z-1)*simspace.Lyx],
simspace.Lyx, MPI_DOUBLE, (mpirank+1), 311, MPI_COMM_WORLD, &status4);
          }

          if (mpirank>1){                                  // receive
Tzz, Trp from left
           MPI_Recv(&simspace.T12[0], simspace.Lyx, MPI_DOUBLE,
(mpirank-1), 313, MPI_COMM_WORLD, &status5);
           MPI_Recv(&simspace.T13[0], simspace.Lyx, MPI_DOUBLE,
(mpirank-1), 314, MPI_COMM_WORLD, &status6);
          }


          if (mpirank>1){                     // Wait for sends to
complete
```

```
            MPI_Wait(&request4, &status4);
           }

           if (mpirank<numworkers){
            MPI_Wait(&request5, &status5);
            MPI_Wait(&request6, &status6);
           }
      }
      return;
}

//
================================================================================
==
// Reads in parameter file (in.file), distributes to all workers,
// and divides up the simulation space
//
================================================================================
==
int* DistributeSimulationParameters() {
      char inputFilename[] = "in.file";
      ifstream inFile;
      inFile.open("in.file", ios::in);

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

      double *simparams = new double[11];
      inFile >> simparams[0];     //maxz
      inFile >> simparams[1];     //maxy
      inFile >> simparams[2];     //maxx
      inFile >> simparams[3];     //ds
      inFile >> simparams[4];     //dt
      inFile >> simparams[5];     //default den
      inFile >> simparams[6];     //lm - default Lame constant - lambda
      inFile >> simparams[7];     //mu - default Lame constant - mu

      inFile >> simparams[9];     //maxt
      inFile >> simparams[10];    //outevery

      maxt = static_cast<int>(simparams[9]);
      outputevery = static_cast<int>(simparams[10]);
      maxz = static_cast<int>(simparams[0]);

      // Send initial data to each node
      int div, divaccum = 0;
      int* xpos = new int[numworkers];
      EvenVolDims[0]=0;
      EvenVolDims[1]=static_cast<int>(simparams[1]/2);
      EvenVolDims[2]=static_cast<int>(simparams[2]/2);
```

```
	for (int n = 1; n <= numworkers; n++) {
		div = (maxz/numworkers);
		if ((n-1) <= (maxz%numworkers))
			div++;   // divide space along z direction
		simparams[0] = static_cast<double>(div);
		cout << "Divided space (div) is " << div << "\n";
		simparams[8] = static_cast<double>(divaccum); // tells the
worker where its starting z location is
		cout << "Worker's starting locations (divaccum) is " <<
divaccum << "\n";
		MPI_Send(&simparams[0], 11, MPI_DOUBLE, n, 201,
MPI_COMM_WORLD);
	       xpos[n-1] = static_cast<int>(simparams[8]);
		divaccum = divaccum + div;

		if(div%2==0)
			EvenVolDims[0]+=(div/2); // len if num_z is even
		else
			EvenVolDims[0]+=((div-1)/2); // len if num_z is even
	}

	cout << "Total simulation timesteps (maxt) = " << maxt << endl;
	delete[] simparams;


	// --- Read in reflectors and distribute to all workers ---
	int numref;
	inFile >> numref;
	double *rpars = new double[9];
	cout << "  Number of reflectors:  " << numref << endl;
	for (int n = 1; n <= numworkers; n++) {
		MPI_Send(&numref, 1, MPI_INT, n, 203, MPI_COMM_WORLD);
	}

	for (int i = 0; i < numref; i++) {
		inFile >> rpars[0];  // reflector type
		inFile >> rpars[1];  // reflector position in x1
		inFile >> rpars[2];  // reflector position in x2
		inFile >> rpars[3];  // reflector position in x3 - (start
for cylinder)
		inFile >> rpars[4];  // reflector position in x3 - (end for
cylinder)
		inFile >> rpars[5];  // refector radius
		inFile >> rpars[6];  // refector density
		inFile >> rpars[7];  // refector mu
		inFile >> rpars[8];  // refector lambda

		for (int n = 1; n <= numworkers; n++) {
			MPI_Send(&rpars[0], 9, MPI_DOUBLE, n, 204,
MPI_COMM_WORLD);
		}
```

```
	}
	delete[] rpars;

	inFile.close();
	return xpos;
}

//
===========================================================================
=============
// Reads in transducer file (trans.file) and distributes to the
correct workers
//
===========================================================================
=============
void DistributeTransducers(int *zposs){
	double tparams[8];
	int numtrans, worker;
	int min_tau=0;
	int min_tau_recv=0;

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

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

	inFile >> numtrans;
	cout << "  number of transducers: " << numtrans << endl;
	numtransducers = numtrans;

	for (int tr = 0; tr<numtrans; tr++){
		inFile >> tparams[0];  // transducer z location
		inFile >> tparams[1];  // transducer y location
		inFile >> tparams[2];  // transducer x location
		inFile >> tparams[3];  // transducer radius
		inFile >> tparams[4];  // len of drive function
		int drvlen = static_cast<int>(tparams[4]);

		tparams[5] = static_cast<double>(tr);

		inFile >> tparams[6];  // transducer theta
		inFile >> tparams[7];  // transducer phi


		double *drive{nullptr};
		if (tparams[4]>0){
			drive = new double[drvlen];
			for (int i = 0; i<drvlen; i++){
```

```
                        inFile >> drive[i];
                    }
            }

            // --- Figure out which workers get the transducer ---
            worker = 0;
            for (int tosend = 1; tosend<numworkers; tosend++)
              if (tparams[0] >= zposs[tosend-1] && tparams[0] <
zposs[tosend]) worker = tosend;
              if (tparams[0] >= zposs[numworkers-1] && tparams[0] <
maxz) worker = numworkers;
              else if (worker == 0) cout << "error: transducer postion
not found: zpos - " <<  tparams[0] << ", " << zposs[numworkers-1] <<
", " << maxz << endl;

            // --- Send transducer info to worker ---
            if (worker > 1){
                  if ((tparams[0] - tparams[3]) <= zposs[worker-1]){
                        MPI_Send(&tparams[0], 8, MPI_DOUBLE, worker-1,
211, MPI_COMM_WORLD);
                        if (tparams[4]>0){
                              MPI_Send(&drive[0], tparams[4], MPI_DOUBLE,
worker-1, 212, MPI_COMM_WORLD);
                              MPI_Recv(&min_tau_recv, 1, MPI_INT, worker-
1, 213, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                              if(min_tau_recv<min_tau) min_tau =
min_tau_recv;
                              MPI_Send(&min_tau, 1, MPI_INT, worker-1,
214, MPI_COMM_WORLD);
                        }
                  }
                  if ((tparams[0] - tparams[3]) <= zposs[worker-2]){
                        MPI_Send(&tparams[0], 8, MPI_DOUBLE, worker-2,
211, MPI_COMM_WORLD);
                        if (tparams[4]>0){
                              MPI_Send(&drive[0], tparams[4], MPI_DOUBLE,
worker-2, 212, MPI_COMM_WORLD);
                              MPI_Recv(&min_tau_recv, 1, MPI_INT, worker-
2, 213, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                              if(min_tau_recv<min_tau) min_tau =
min_tau_recv;
                              MPI_Send(&min_tau, 1, MPI_INT, worker-2,
214, MPI_COMM_WORLD);
                        }
                  }
            }

            MPI_Send(&tparams[0], 8, MPI_DOUBLE, worker, 211,
MPI_COMM_WORLD);
            if (tparams[4]>0){
                  MPI_Send(&drive[0], tparams[4], MPI_DOUBLE, worker,
212, MPI_COMM_WORLD);
```

```
                MPI_Recv(&min_tau_recv, 1, MPI_INT, worker, 213,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                if(min_tau_recv<min_tau) min_tau = min_tau_recv;
                MPI_Send(&min_tau, 1, MPI_INT, worker, 214,
MPI_COMM_WORLD);
            }
            if (worker < numworkers){
                if ((tparams[0] + tparams[3]) >= zposs[worker]){
                    MPI_Send(&tparams[0], 8, MPI_DOUBLE, worker+1,
211, MPI_COMM_WORLD);
                    if (tparams[4]>0){
                        MPI_Send(&drive[0], tparams[4], MPI_DOUBLE,
worker+1, 212, MPI_COMM_WORLD);
                        MPI_Recv(&min_tau_recv, 1, MPI_INT,
worker+1, 213, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        if(min_tau_recv<min_tau) min_tau =
min_tau_recv;
                        MPI_Send(&min_tau, 1, MPI_INT, worker+1,
214, MPI_COMM_WORLD);
                    }
                }
                if ((tparams[0] + tparams[3]) >= zposs[worker+1]){
                    MPI_Send(&tparams[0], 8, MPI_DOUBLE, worker+2,
211, MPI_COMM_WORLD);
                    if (tparams[4]>0){
                        MPI_Send(&drive[0], tparams[4], MPI_DOUBLE,
worker+2, 212, MPI_COMM_WORLD);
                        MPI_Recv(&min_tau_recv, 1, MPI_INT,
worker+2, 213, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        if(min_tau_recv<min_tau) min_tau =
min_tau_recv;
                        MPI_Send(&min_tau, 1, MPI_INT, worker+2,
214, MPI_COMM_WORLD);
                    }
                }
            }
            delete[] drive;
        }

        // --- Let all workers know we are done distributing transducers
---
        tparams[0] = -1;tparams[1] = -1;tparams[2] = -1;tparams[3] = -
1;tparams[4] = -1;tparams[5] = -1;
        for (int n = 1; n <= numworkers; n++){
            MPI_Send(&tparams[0], 8, MPI_DOUBLE, n, 211,
MPI_COMM_WORLD);
        }

        inFile.close();
        return;
}
```

```
//
======================================================================
==
// Dump data to file!
//
======================================================================
==
void dump3Dbin(int t) // pressure data as binary
{
      MPI_Status  status;
      double *data3d{nullptr};
      int len;

      stringstream strm;
      strm << t;
      string fname = "data3d_at_t_" + strm.str() + ".bin";
      ofstream outFile(fname.c_str(), ios::binary);

      for (int n = 1; n <= numworkers; n++) {
            MPI_Recv(&len, 1, MPI_INT, n, 1101, MPI_COMM_WORLD,
&status);
            data3d = new double[len];
            MPI_Recv(&data3d[0], len, MPI_DOUBLE, n, 1102,
MPI_COMM_WORLD, &status);

            for (int i = 0; i < len; i++) {
                  outFile.write((char *)(&data3d[i]),
sizeof(data3d[i]));
            }

            delete[] data3d;
      }

      outFile.close();
      return;
}
```

C++ Code: ‘space.h’

```
/* space.h
 * Sets up the transducers and reflectors for the Cartesian EFIT
simulation
 * Redesigned integration of 'setup_cart_space.h' and 'array3D.h'
(Dieckman)
 *
 *  Created on: March 27, 2019
 *      Author: Sean M. Raley (UNH)
 */

#ifndef SPACE_H_
#define SPACE_H_

#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <algorithm>
#include "transducer.h"

using namespace std;

class space{

public:
	space(double *params);
	~space();

	int num_z, num_y, num_x; // number of grid points in each
direction
	int num_zB, num_yB, num_xB; // number of Boundary grid points in
each direction

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

	double ds; // spatial step size (meters)
	double dt; // time step size (seconds)

	double den;	// default density (kg/m^3)
	double lm; // default Lame constant - lambda
	double mu; // default Lame constant - mu

	int zbeg;  // z start position (where divvacuum ends)
	int type;  // type -> 1 = left, 2 = middle, 3 = right (for MPI)

	double *vz;	// velocities in z-dir
	double *vy;	// velocities in y-dir
	double *vx;	// velocities in x-dir
	double *T11;// zz normal stress
	double *T22;// yy normal stress
```

```
	double *T33;// xx normal stress
	double *T12;// zy normal stress
	double *T23;// yx normal stress
	double *T13;// zy normal stress

	double *d; // density
	double *lmd;// Lame parameter - lambda
	double *muu;// Lame parameter - mu

	int *B;		// Boundary array

	int drivetime;

	transducer *trans{nullptr};
	int numtrans;

	int Lyx;   // array length of entire x-y plane
	int Lzyx;  // array length of entire x-y-z volume
	int LyxB;  // array length of entire x-y plane (Boundary array)
	int LzyxB; // array length of entire x-y-z volume (Boundary
array)

	double dtods;	// time step over spatial step

private:
	double lmdtods;
	double l2mdtods;
	double mdtods;

	int iz, iy, ix; // counters for spatial loops
	int pp1,pm1;

	double PIo2;

	//==========================================================
	// low-level initialization function
	template <class SomeType>
	SomeType *init(SomeType def){
		SomeType *temparray = new SomeType[Lzyx];
		clear(temparray,def);
		return temparray;
	}

	// special version for B array
	int *initB(int def){
		int *temparray = new int[LzyxB];
		clearB(temparray, def);
		return temparray;
	}

public:
```

```
	//
===========================================================================
=============
	// Update velocities and stresses
	//
===========================================================================
=============
	void UpdateVs(int zs, int zend){
		int tau_time;
		int tau_count = 0;

		for (iz = zs; iz <= zend; iz++){
			setindx(iz,0,0);
			for (iy = 0; iy < num_y; iy++){
				setindxB(iz+1,iy+1,1);
				for (ix = 0; ix < num_x; ix++){

		// --- vy ---
					if (vB(B)==0){
						sv(vy, v(vy) +
2*dtods/(v(d)+vyp(d))*((v(T12)-vzm(T12))+(vyp(T22)-v(T22))+(v(T23)-
vxm(T23))));
					}
					else if (vB(B)==2 || vzmB(B)==2 ||
vxmB(B)==2) {} // because vy requires elements in direction vxm() and
vzm()
					else if (vypB(B)==2){
						sv(vy, v(vy) + 2*dtods/(v(d)+v(d))*(-
2*(v(T22))));
					}
					else if (vymB(B)==2){
						sv(vy, v(vy) +
2*dtods/(v(d)+v(d))*(2*(vyp(T22))));
					}
					else{
						sv(vy, v(vy) +
2*dtods/(v(d)+vyp(d))*((v(T12)-vzm(T12))+(vyp(T22)-v(T22))+(v(T23)-
vxm(T23))));
					}


		// --- vz ---
					if (vB(B)==0){
						sv(vz, v(vz) +
2*dtods/(v(d)+vzp(d))*((vzp(T11)-v(T11))+(v(T12)-vym(T12))+(v(T13)-
vxm(T13))));  // middle
					}
					else if (vB(B)==2 || vymB(B)==2 ||
vxmB(B)==2) {}  //b/c vz needs elements in the vym() and vxm()
directions
					else if (vzpB(B)==2){
```

```
                              sv(vz, v(vz) + 2*dtods/(v(d)+v(d))*(-
2*(v(T11)))); //right end (only called if at last node)
                        }
                        else if (vzmB(B)==2){
                              sv(vz, v(vz) +
2*dtods/(v(d)+v(d))*(2*(vzp(T11)))); //left end (only called if at
first node)
                        }
                        else{
                              sv(vz, v(vz) +
2*dtods/(v(d)+vzp(d))*((vzp(T11)-v(T11))+(v(T12)-vym(T12))+(v(T13)-
vxm(T13))));
                        }


            // --- vx ---
                        if (vB(B)==0){
                              sv(vx, v(vx) +
2*dtods/(v(d)+vxp(d))*((v(T13)-vzm(T13))+(v(T23)-vym(T23))+(vxp(T33)-
v(T33))) );
                        }
                        else if (vB(B)==2 || vymB(B)==2 ||
vzmB(B)==2) {}
                        else if ( vB(B)>=9000){
                              sv(vx, v(vx) + 2*dtods/(v(d)+v(d))*(-
2*(v(T33)))+ 2*dtods/(den+den)*(-trans[vB(B)-
9000].drivef(drivetime)));
                        }
                        else if ( vB(B)>=1000){
                              giveindx();
                              tau_count =
currentloc[0]*num_y+currentloc[1];
                                    if(trans[vB(B)-
1000].tau[tau_count]<drivetime){
                                          tau_time = drivetime -
trans[vB(B)-1000].tau[tau_count];
                                    }
                                    else{
                                          tau_time=0;
                                    }
                              sv(vx, v(vx) + 2*dtods/(v(d)+v(d))*(-
2*(v(T33)))+ 2*dtods/(den+den)*(trans[vB(B)-1000].drivef(tau_time)));
                        }
                        else if (vxpB(B)==2){
                              sv(vx,  v(vx) + 2*dtods/(v(d)+v(d))*(-
2*(v(T33)))); //top
                        }
                        else if (vxmB(B)==2){
                              sv(vx, v(vx) +
2*dtods/(v(d)+v(d))*(2*(vxp(T33)))); //bottom
                        }
                        else{
```

```
                                    sv(vx, v(vx) +
2*dtods/(v(d)+vxp(d))*((v(T13)-vzm(T13))+(v(T23)-vym(T23))+(vxp(T33)-
v(T33))) );
                                }

                                incindx();
                                incindxB();
                        }
                }
            }
      }

      void UpdateTs(int zs, int zend){
            for (iz = zs; iz <= zend; iz++){
                  setindx(iz,0,0);
                  for (iy = 0; iy < num_y; iy++){
                        setindxB(iz+1,iy+1,1);
                        for (ix = 0; ix < num_x; ix++){

              // --- Tii ---
                                if (vB(B)==2 || vymB(B)==2 || vxmB(B)==2 ||
vzmB(B)==2) {} // 1 is iz, 2 is iy, 3 is ix
                                else{
                                      sv(T11,
v(T11)+dtods*(v(lmd)+(2*v(muu)))*(v(vz)-
vzm(vz))+dtods*(v(lmd))*((v(vy)-vym(vy))+(v(vx)-vxm(vx))));
                                      sv(T22,
v(T22)+dtods*(v(lmd)+(2*v(muu)))*(v(vy)-
vym(vy))+dtods*(v(lmd))*((v(vz)-vzm(vz))+(v(vx)-vxm(vx))));
                                      sv(T33,
v(T33)+dtods*(v(lmd)+(2*v(muu)))*(v(vx)-
vxm(vx))+dtods*(v(lmd))*((v(vz)-vzm(vz))+(v(vy)-vym(vy))));
                                }

              // --- T13 ---
                                if (vB(B)==2 || vzpB(B)==2 || vzmB(B)==2 ||
vxpB(B)==2 || vxmB(B)==2 || vymB(B)==2 || vzxpB(B)==2) {}
                                else{
                                      sv(T13,
v(T13)+dtods*(4/(((1/v(muu))+(1/vzp(muu))+(1/vxp(muu))+(1/vzxp(muu))))
)*((vxp(vz)-v(vz))+(vzp(vx)-v(vx))));
                                }

              // --- T23 ---
                                if (vB(B)==2 || vypB(B)==2 || vymB(B)==2 ||
vxpB(B)==2 || vxmB(B)==2 || vzmB(B)==2 || vyxpB(B)==2) {}
                                else{
                                      sv(T23,
v(T23)+dtods*(4/((1/(v(muu))+(1/vyp(muu))+(1/vxp(muu))+(1/vyxp(muu))))
)*((vxp(vy)-v(vy))+(vyp(vx)-v(vx))));
                                }
```

```
            // --- T12 ---
                              if (vB(B)==2 || vypB(B)==2 || vymB(B)==2 ||
vzpB(B)==2 || vzmB(B)==2 || vxmB(B)==2 || vzypB(B)==2) {}
                              else{
                                   sv(T12,
v(T12)+dtods*((4/((1/v(muu))+(1/vzp(muu))+(1/vyp(muu))+(1/vzyp(muu))))
)*((vyp(vz)-v(vz))+(vzp(vy)-v(vy))));
                              }

                              incindx();
                              incindxB();
                         }
                    }
               }
          }

          //
======================================================================
=============
          // Add/update transducers
          //
======================================================================
=============
          void UpdateTransducers(int t){
                int tr;
               for (int i1 = 1; i1<num_z-1; i1++){
                for (int i2 = 0; i2<num_y-1; i2++){
                         if (valB(B,i1+1,i2+1,num_x) >= 1000){
                              tr = valB(B,i1+1,i2+1,num_x)-1000;
//transducer ID
                              trans[tr].record[t] =  trans[tr].record[t]
+ val(vx,i1,i2,num_x-1); //record holds recorded info
                         }
                }
               }
          }


          void addTransducer(transducer t){
                numtrans = numtrans+1;
               transducer *temp = new transducer[numtrans]; // of class
transducer, length numtrans

               for (int i = 0; i<numtrans-1; i++){ // loops from 0 to actual
number of transducers
                temp[i] = trans[i];
               }

               temp[numtrans-1] = t;
               trans = temp;
               int nelems = 0;
                for (int i1 = 1; i1<num_z-1; i1++){
```

```
		for (int i2 = 0; i2<num_y; i2++){
			if (((i1+zbeg-trans[numtrans-1].posi1)*(i1+zbeg-
trans[numtrans-1].posi1)+(i2-trans[numtrans-1].posi2)*(i2-
trans[numtrans-1].posi2)) <= (trans[numtrans-1].radius*trans[numtrans-
1].radius)){
				setB(B,i1+1,i2+1,t.posi3+1,1000+numtrans-1);
//executes if (z-zo)^2+(y-yo)^2<=rad^2, ie, if within a circle of
transducer radius then set B >1000
				nelems++;
			}
		}
		}

		if(nelems != trans[numtrans-1].tau_values){
			cout<<"WARNING! For worker starting at
z="<<zbeg<<"Number of transducer elements defined by addTransducer()
does NOT equal number of transducer elements in delay array tau.
Transducer errors likely."<<endl;
		}

		temp[numtrans-1].numelems = nelems;

	}

	//
==========================================================================
=============
	// Add/update relectors - different types defined below
	//
==========================================================================
=============
	void addReflector(double typ, double p1, double p2, int start3,
int end3, double rad, double dd, double mu, double lambda){

	   // --- 3D RECTANGULAR VOID ---
	   else if (typ == 3){
		for (int i1 = 0; i1 < num_z; i1++){
			for (int i2 = 0; i2 < num_y; i2++){
				for (int i3 = 0; i3 < num_x; i3++){
					if ((i1+zbeg-1 >= p1) && (i1+zbeg-1 <= p2)
&& (i2 >= start3) && (i2 <= end3) && (i3 >= rad) && (i3 <= dd)){
						setB(B,i1+1,i2+1,i3+1,2); // Stress-
free void
					}
				}
			}
		}
		}

		// --- arb 3d scatterer from STL file ---
		else if (typ == 5)
		{
```

```
                char inputFilename[] = "arbscatt.file";
                ifstream inFile;
                inFile.open("arbscatt.file", ios::in); //arbscatt file
follows (x,y,z) orientation

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

                int tss = rad; // total size of space
                int *scatterspace = new int[tss];
                for (int i = 0; i<tss; i++){
                inFile >> scatterspace[i]; // read in entire space
including scatterer
           }

                int tempB;
             for (iz = 0; iz < num_z; iz++){
                for (iy = 0; iy < num_y; iy++){
                     for (ix = 0; ix< num_x; ix++){
                          tempB = valB(B,iz+1,iy+1,ix+1);
                          if ((scatterspace[(iz+zbeg)*num_y*num_x +
iy*num_x + ix] != 1) && (tempB != 1) && (tempB != 2)){
                               if ((lambda == -1) && (mu == -1) &&
(dd == -1)){
                                    setB(B,iz+1,iy+1,ix+1,2);
                               }
                               else{
                                    set(lmd,iz,iy,ix,lambda);
                                    set(muu,iz,iy,ix,mu);
                                    set(d,iz,iy,ix,dd);
                               }
                          }
                          //i++;
                     }
                }
             }
                inFile.close();
                delete[] scatterspace;
           }
      }


      int round(double a){
           return int(a+0.5);
      }

      //=========================================================
      // Array Manipulation
```

```
private:
      int ci; // current index
      int ciB; // current Boundary index (used for Boundary array only)
      int currentloc[3]; // for storing current zyx location

public:
      // Returns value at i_z, i_y, i_x
      template <class SomeType>
      SomeType val(SomeType *a, int i_z, int i_y, int i_x){
            return a[(i_z*Lyx)+(i_y*num_x)+i_x];
      }

      // Sets value at i_z, i_y, i_x
      template <class SomeType>
      void set(SomeType *a, int i_z, int i_y, int i_x, SomeType val){
            a[(i_z*Lyx)+(i_y*num_x)+i_x] = val;
      }

      // Returns Boundary Array value at i_z, i_y, i_x
      template <class SomeType>
      SomeType valB(SomeType *a, int i_z, int i_y, int i_x){
            return a[(i_z*LyxB)+(i_y*num_xB)+i_x];
      }

      // Sets Boundary Array value at i_z, i_y, i_x
      template <class SomeType>
      void setB(SomeType *a, int i_z, int i_y, int i_x, SomeType val){
            a[(i_z*LyxB)+(i_y*num_xB)+i_x] = val;
      }
      //==========================================================
      // Quick Access Methods

      // sets index ci
      void setindx(int iz, int iy, int ix){
            ci = (iz*Lyx)+(iy*num_x)+ix;
      }
      // increments index counter by 1
      void incindx(){
            ci = ci+1;
      }
      void sv(double *a, double x){a[ci]=x;                  }     //
sets value at ci

      template <class SomeType>
      SomeType v(SomeType *a)          {return a[ci];              }
      // equiv of a[iz][iy][ix]
      template <class SomeType>
      SomeType vzp(SomeType *a)   {return a[ci+Lyx];          }     //
equiv of a[iz+1][iy][ix]
      template <class SomeType>
      SomeType vzm(SomeType *a)   {return a[ci-Lyx];          }     //
equiv of a[iz-1][iy][ix]
```

```
      template <class SomeType>
      SomeType vyp(SomeType *a)  {return a[ci+num_x];  }     // equiv
of a[iz][iy+1][ix]
      template <class SomeType>
      SomeType vyp2(SomeType *a) {return a[ci+2*num_x];      }     //
equiv of a[iz][iy+2][ix]
      template <class SomeType>
      SomeType vym(SomeType *a)  {return a[ci-num_x];  }     // equiv
of a[iz][iy-1][ix]
      template <class SomeType>
      SomeType vxp(SomeType *a)  {return a[ci+1];            }     //
equiv of a[iz][iy][ix+1]
      template <class SomeType>
      SomeType vxm(SomeType *a)  {return a[ci-1];      }     // equiv
of a[iz][iy][ix-1]
      template <class SomeType>
      SomeType vzxp(SomeType *a) {return a[ci+1+Lyx];  }     // equiv
of a[iz+1][iy][ix+1]
      template <class SomeType>
      SomeType vyxp(SomeType *a) {return a[ci+1+num_x];      }     //
equiv of a[iz][iy+1][ix+1]
      template <class SomeType>
      SomeType vzyp(SomeType *a) {return a[ci+Lyx+num_x];}  // equiv
of a[iz+1][iy+1][ix]

      // sets array for z,y,x at current index ci
      void giveindx(){
            currentloc[0] = ((ci / num_x) / num_y) % num_z; // z
position
            currentloc[1] = (ci / num_x) % num_y; // y position
            currentloc[2] = ci % num_x; // x position
      }

      //=========================================================
      // Quick Access Methods (Boundary array)

      // sets index ciB
      void setindxB(int iz, int iy, int ix){
            ciB= (iz*LyxB)+(iy*num_xB)+ix;
      }

      // increments Boundary index counter by 1
      void incindxB(){
            ciB = ciB+1;
      }
      void svB(double *a, double x){a[ciB]=x;                      }
      // sets value at ciB

      template <class SomeType>
      SomeType vB(SomeType *a)   {return a[ciB];             }     //
equiv of a[iz][iy][ix]
      template <class SomeType>
```

```
	SomeType vzpB(SomeType *a) {return a[ciB+LyxB];        }     //
equiv of a[iz+1][iy][ix]
	template <class SomeType>
	SomeType vzmB(SomeType *a) {return a[ciB-LyxB];        }     //
equiv of a[iz-1][iy][ix]
	template <class SomeType>
	SomeType vypB(SomeType *a) {return a[ciB+num_xB];      }     //
equiv of a[iz][iy+1][ix]
	template <class SomeType>
	SomeType vyp2B(SomeType *a){return a[ciB+2*num_xB];    }     //
equiv of a[iz][iy+2][ix]
	template <class SomeType>
	SomeType vymB(SomeType *a) {return a[ciB-num_xB];      }     //
equiv of a[iz][iy-1][ix]
	template <class SomeType>
	SomeType vxpB(SomeType *a) {return a[ciB+1];           }     //
equiv of a[iz][iy][ix+1]
	template <class SomeType>
	SomeType vxmB(SomeType *a) {return a[ciB-1];           }     //
equiv of a[iz][iy][ix-1]
	template <class SomeType>
	SomeType vzxpB(SomeType *a){return a[ciB+1+LyxB];      }     //
equiv of a[iz+1][iy][ix+1]
	template <class SomeType>
	SomeType vyxpB(SomeType *a){return a[ciB+1+num_xB];    }     //
equiv of a[iz][iy+1][ix+1]
	template <class SomeType>
	SomeType vzypB(SomeType *a){return a[ciB+LyxB+num_xB];}     //
equiv of a[iz+1][iy+1][ix]

	//==========================================================
	// sets all values = def
	template <class SomeType>
	void clear(SomeType *a,SomeType def){
		for(int i=0; i<Lzyx; i++){
			a[i]=def;
		}
	}

	// special version for Boundary array
	void clearB(int *a, int def){
		for(int i=0; i<LzyxB; i++){
			a[i]=def;
		}
	}

	// sets all outer boundary surfaces for Boundary array
	void setBoundaries(){
	    for (int i1 = 0; i1< num_zB; i1++){
		for (int i3 = 0; i3< num_xB; i3++){
			setB(B,i1,0,i3,2);  //set x2 boundary
			setB(B,i1,1,i3,1);
```

```
                setB(B,i1,num_yB-1,i3,2);
                setB(B,i1,num_yB-2,i3,2);
                setB(B,i1,num_yB-3,i3,1);
            }
        }

        for (int i1 = 0; i1< num_zB; i1++){
          for (int i2 = 0; i2< num_yB; i2++){
                setB(B,i1,i2,0,2);
                setB(B,i1,i2,1,1);
                setB(B,i1,i2,num_xB-1,2);//set x3 boundary
                setB(B,i1,i2,num_xB-2,2);

                if ((i2>0) && (i2<num_zB-2)){
                      setB(B,i1,i2,num_xB-3,1);
                }
            }
        }

        for (int i2 = 0; i2< num_yB; i2++){
          for (int i3 = 0; i3< num_xB; i3++){
                if (type == 1){
                      setB(B,0,i2,i3,2);  // if at actual end of space
in z set boundary
                      if ( (i2 > 0) && (i2<(num_yB-2)) && (i3 > 0) &&
(i3<(num_xB-2))){
                            setB(B,1,i2,i3,1);
                      }
                }

                if (type == 3){
                      setB(B,num_zB-1,i2,i3,2);
                      setB(B,num_zB-2,i2,i3,2);
                      if ( (i2 > 0) && (i2<(num_yB-2)) && (i3 >0) &&
(i3<(num_xB-2))){
                            setB(B,num_zB-3,i2,i3,1);
                      }
                }
            }
        }
    }

    //=========================================================
    // returns 2D slice through 3D array at fixed index y
    double* slice_fixy(double *a, int y_slice){
          double *slice = new double[(num_z-2)*num_x];
          int count = 0;
          for(iz=1; iz<num_z-1; iz++) // does not return ends
                for(ix=0; ix<num_x; ix++){
                      slice[count] = val(a,iz,iy,ix);
                      count++;
                }
```

```
            return slice;
      }

      //============================================================
      // returns 3D volume returning only the even indexes
      double* GetEvenVol(double *a, int len){
            double *EvenArray = new double[len];
            int count = 0;
            for(iz=1+(num_z%2); iz<num_z-1; iz+=2) // does not return
ends
                  for(iy=0; iy<num_y-1; iy+=2)
                        for(ix=0; ix<num_x-1; ix+=2){
                              EvenArray[count] = val(a, iz, iy, ix);
                              count++;
                        }
            return EvenArray;
      }

      // overload to retrieve magnitude of full velocity vector
      double* GetEvenVol(int len){
            double *EvenArray = new double[len];
            int count = 0;
            double xtemp, ytemp, ztemp;
            for(iz=1+(num_z%2); iz<num_z-1; iz+=2) // does not return
ends
                  for(iy=0; iy<num_y-1; iy+=2)
                        for(ix=0; ix<num_x-1; ix+=2){
                              xtemp = val(vx, iz, iy, ix);
                              ytemp = val(vy, iz, iy, ix);
                              ztemp = val(vz, iz, iy, ix);
                              EvenArray[count] =
sqrt(xtemp*xtemp+ytemp*ytemp+ztemp*ztemp);
                              count++;
                        }
            return EvenArray;
      }

      int GetEvenVolLen(){
            int len;
            if(num_z%2==0)
                  len = (num_z-1)/2*(num_y/2)*(num_x/2); // len if num_z
is even
            else
                  len = (num_z-2)/2*(num_y/2)*(num_x/2); // len if num_z
is odd
            return len;
      }
};

space::space(double *params){
      num_z = params[0]+2;  // number of nodes in z-direction
      num_y = params[1];          // number of nodes in y-direction
```

```
	num_x = params[2];		// number of nodes in x-direction
	ds = params[3];		// spatial step size (m)
	dt = params[4];		// time step size (s)

	den = params[5];		// density
	lm = params[6];		// Lame constant - lambda
	mu = params[7];		// Lame constant - mu
	zbeg = params[8];		// simspace z-starting position for
each node

	Lyx = num_y*num_x;
	Lzyx = num_z*num_y*num_x;

	vz = init(0.0);
	vy = init(0.0);
	vx = init(0.0);
	T11 = init(0.0);
	T22 = init(0.0);
	T33 = init(0.0);
	T12 = init(0.0);
	T23 = init(0.0);
	T13 = init(0.0);

	d = init(den);
	lmd = init(lm);
	muu = init(mu);

	num_zB = num_z+2;	// number of Boundary nodes in z-direction
	num_yB = num_y+2;	// number of Boundary nodes in y-direction
	num_xB = num_x+2;	// number of Boundary nodes in x-direction
	LyxB = num_yB*num_xB;
	LzyxB = num_zB*num_yB*num_xB;

	B = initB(0);
	setBoundaries();

	dtods = dt/ds;
    lmdtods  = (lm*dt)/ds;
    l2mdtods = ((lm+2*mu)*dt)/ds;
    mdtods   = (mu*dt)/ds;

    PIo2 = 3.14159265358979/2;

    numtrans=0;
	drivetime = 0;
    abc = 80;
}

space::~space() {
	delete[] vz; // velocities in z-dir
	delete[] vy; // velocities in y-dir
	delete[] vx; // velocities in x-dir
```

```
	delete[] T11;
	delete[] T22;
	delete[] T33;
	delete[] T12;
	delete[] T23;
	delete[] T13;

	delete[] d;  // density
	delete[] muu;
	delete[] lmd;

	delete[] B;  // Boundary array

}

#endif /* SPACE_H_ */
```

C++ Code: 'transducer.h'

```
 /* 'transducer.h'
Custom transducer class for the Cartesian EFIT simulation
Cleaned up and heavily modified version of 'transducer.h' (Bertoncini,
Campbell-Leckey, Miller, etc)

Eric A. Dieckman (WM)
Last edited: 10 Apr 2019 SMR
*/

using namespace std;
class transducer{
public:
	double *drive{nullptr};		// array that holds drive function
	int dflen=0;				// length of drivefunc

	double posi1=0;			// transducer center (z-direction)
	double posi2=0;			// transducer center (y-direction)
	double posi3=0;			// transducer center (x-direction)
	double radius=0;		// transducer radius - meters
	bool driven=false;		// driven: true=active (pitch or
pitch/catch), false=passive (catch)
	int transID=0;
	int numelems=0;			// number of elements in simulation
space
	double *record{nullptr};// array that holds recorded value

	double theta=0;		// beam angle from normal / x-direction -
radians
	double phi=0;			// beam angle from (+)z-direction -
radians
	int num_y = 0;			// y elements in this worker
	int num_z = 0;			// z elements in this worker
	int *tau{nullptr};		// yz array of element time delays for
drive function
	int zbeg = 0;			// starting location for this worker
in z-direction
	int tau_values=0;		// number of elements which have
values in the tau array (equal to number of transducer elements on
this node)
	double dtods=0;

	transducer() { // blank constructor
	}

	transducer(double x1, double x2, double x3, double rad, int tID,
int maxt, double angle1, double angle2, int dim_y, int dim_z, int
z_start, double timebyspace){
		posi1  = x1;
		posi2  = x2;
```

```
			posi3  = x3;
			radius = rad;
			transID = tID;
			driven = false;
			record = new double[maxt];
			for (int i = 0; i< maxt; i++) record[i] = 0;

			theta = angle1;
			phi = angle2;
			num_y = dim_y;
			num_z = dim_z;
			zbeg = z_start;
			dtods=timebyspace;
			tau = new int[num_z*num_y];
		}

		~transducer() { // blank deconstructor
		}

//
======================================================================
=============
// Initialize (define array and dimensions - call before using!)
//
======================================================================
=============
		void setDriveFunction(int len, double df[], int
min_tau_absolute){
			for(int tau_count=0; tau_count<num_z*num_y; tau_count++){
				tau[tau_count]=tau[tau_count]-min_tau_absolute;
			}
			drive = new double[len];
			drive = df;
			dflen = len;
			driven = true;
			return;
		}

		double drivef(int t){
			if (t<dflen){
				return drive[t];
			}
			else{
				return 0;
			}
		}

		double setDelays(int speed){
			int cL = speed;		// default speed of sound, calculated
in slave()
			int min_tau_local=0;	// minimum value of tau in this node
			int temp_tau;
```

```
            for(int k=0; k<num_z; k++){
                  for(int j=0; j<num_y; j++){
                        if(((k+zbeg-posi1)*(k+zbeg-posi1)+(j-posi2)*(j-
posi2)) <= (radius*radius)){
                              temp_tau=static_cast<int>( floor(((k+zbeg-
posi1)*cos(phi) + (j-posi2)*sin(phi)) * sin(theta)/cL/dtods) );
                              tau[k*num_y+j]=temp_tau;
                              if(temp_tau<min_tau_local){
                                    min_tau_local=temp_tau;
                              }
                              tau_values++;
                        }
                        else{
                              tau[k*num_y+j]=0; // if not within the
circle of the transducer
                        }
                  }
            }
            return min_tau_local;
      }
};
```

## SELECTED REFERENCES

[1] C. J. Hellier, *Handbook of Nondestructive Evaluation*, 2nd ed. McGraw-Hill Professional Publishing, 2012.

[2] S. r. o. STARMANS electronics, "Railway Rail Testing," 2019. [Online]. Available: http://www.starmans.net/applications/railway-rail-testing/. [Accessed: 31-May-2019].

[3] *Nordco Rail Flaw Defects Identification Handbook*. Nordco Rail Services.

[4] P. M. Morse, K. U. Ingard, and F. B. Stumpf, "Theoretical Acoustics," *Am. J. Phys.*, vol. 38, no. 5, pp. 666–667, 1970.

[5] C. J. Partridge, "Sound Wave Scattering from a Rigid Sphere," Maribyrnong, Vic., 1993.

[6] P. A. Martin, "Acoustic scattering by a sphere in the time domain," *Wave Motion*, vol. 67, pp. 68–80, 2016.

[7] P. Fellinger, R. Marklein, K. J. Langenberg, and S. Klaholz, "Numerical modeling of elastic wave propagation and scattering with EFIT - elastodynamic finite integration technique," *Wave Motion*, vol. 21, no. 1, pp. 47–66, 1995.

[8] P. Thoma and T. Weiland, "A subgridding method in combination with the finite integration technique," *1995 25th Eur. Microw. Conf.*, vol. 2, no. 1, pp. 770–774, 1995.

[9] K. S. Yee, "Numerical Solution of Initial Boundary Value Problems Involving Maxwell's Equations in Isotropic Media," *IEEE Transactions on Antennas and Propagation*. 1966.

[10] W. Yu, *Advanced FDTD Methods: Parallelization, Acceleration, and Engineering Applications*. Artech House, 2011.

[11] T. Weiland, M. Timm, and I. Munteanu, "A practical guide to 3-D simulation," *IEEE Microw. Mag.*, vol. 9, no. 6, pp. 62–75, Nov. 2008.

[12] R. Marklein, "The Finite Integration Technique as a General Tool to Compute Acoustic,

Electromagnetic, Elastodynamic, and Coupled Wave Fields," *Rev. Radio Sci. 1999-2002 URSI*, pp. 201–244, 2002.

[13] J.-F. Lee, R. Lee, and A. Cangellaris, "Time-Domain Finite-Element Methods," *IEEE Trans. Antennas Propag.*, vol. 45, no. 3, pp. 430–442, 1997.

[14] L. E. Kinsler, A. R. Frey, A. B. Coppens, and J. V. Sanders, *Fundamentals of acoustics 4th edition*. 2000.

[15] K. E. Rudd, "Parallel three-dimensional acoustic and elastic wave simulation methods with applications in nondestructive evaluation," 2003.

[16] H. Childs *et al.*, "VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data," in *High Performance Visualization--Enabling Extreme-Scale Scientific Insight*, 2012, pp. 357–372.

[17] E. Gabriel *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," *Recent Adv. Parallel Virtual Mach. Messag. Passing Interface*, pp. 97–104, 2004.

[18] J.-P. Berenger, "A Perfectly Matched Layer for the Absorption of Electromagnetic Waves," *J. Comput. Phys.*, vol. 114, no. 2, pp. 185–200, 1994.

[19] University of Texas at El Paso, "Finite-Difference Time-Domain Poster." [Online]. Available: http://emlab.utep.edu/academics.htm. [Accessed: 01-Jun-2019].

[20] Sanpaz, "Components of the Cauchy stress tensor in Cartesian coordinates," *Wikipedia*, 2009. [Online]. Available: https://commons.wikimedia.org/wiki/File:Components_stress_tensor_cartesian.svg. [Accessed: 01-Jun-2019].

[21] S. Garrett, *Understanding Acoustics: An Experimentalist's View of Acoustics and*

*Vibration (Graduate Texts in Physics)*, 1st ed. Springer, 2017.

[22] A. Briggs and O. Kolosov, *Acoustic Microscopy: Second Edition*. 2010.

[23] F. Jones, "Ray Paths in Layered Media: Reflections and refractions at a plane interface," *UBC Earth and Ocean Sciences*, 2018. [Online]. Available: https://www.eoas.ubc.ca/courses/eosc350/content/methods/meth_6/raypaths.html.