A Project Report

submitted by

D AKSHAY RANGASAI

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY

under the guidance of
Dr. Jayalal Sarma M.N.

DEPARTMENT OF MECHANICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.

May 2015

THESIS CERTIFICATE

This is to certify that the thesis entitled , submitted by **D Akshay Rangasai**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bona fide record of the research work carried out by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Krishnan Balasubramanian
Research Guide
Professor
Dept. of Mechanical Engineering
IIT-Madras, 600 036

L Vijayaraghavan
Research Co-Guide
Professor
Dept. of Mechanical Engineering
IIT-Madras, 600 036

Place: Chennai

Date:

ACKNOWLEDGEMENTS

I want to thank myself for this completely pointless endeavour and my parents for paining me constantly about this. This is in the end, quite depressing.

ABSTRACT

Most materials have two regimes of operation when it comes to the relationship between stress and strain of the material, namely linear and non-linear. while phenomenon and material characterization in the linear regime of operation is pretty well understood, the non-linear regime is not as well understood. This is an intriguing part of the problem as materials that undergo plastic deformation and fatigue loading operate under this non-linear regime, and characterization of these properties help in various manufacturing processes.

This study aims to statistically model and extract relevant parameters to measure non-linearity and its effects on a material by the use of ultrasonic waves, which provide a high strain rate, but very low strain, which is ideal to test the material without changing any of its properties at the current state. We first characterize parameters through harmonics generation and then proceed to non-linear wave mixing, a technique which gives us spatial specificity in our measurements.

The forward model was first built by creating a Finite Difference Time Domain (FDTD) solution to a set of differential equations that represent two dimensional non-linear wave propagation in an isotropic solid medium in a euclidean coordinate system. Wave mixing was simulated using a transverse and a longitudinal wave mixing in collinear path, with a phased array simulated as the transducer. Sensitivity analysis was performed for this solution and this formed the basis of our inverse model that helped predict material parameters.

The inverse model for the forward model was first built using linear regression and the results were compared with a statistical learning technique. We used Gaussian Process modelling to model the predictive model, which we further used to build the inverse model. To evaluate the model, noise was added to the measurements at various Signal to Noise Ratio (SNR) and the error percentage was measured. This model proved to be sufficient for the inverse model. From this, we could effectively estimate model parameters from wave mixing measurements.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

The subject of the present work deals with the propagation, non-linear mixing effects of high-frequency shear and longitudinal waves which help us characterize material properties. This section is a brief introduction containing the background, outline and applications of the presented work.

1.1 Background

The stress-strain curve for most ductile materials starts in a linear relationship and moves into a non-linear relationship with a few invariants that define the said relationship. Determining these constants is of great use to characterize the material and predict its behaviour under various conditions and also optimize processes with respect to these constants. While the physical relationship between the linear constants and their estimation has been studied to a great extent, the non-linear constants are more difficult to estimate. Our work aims to estimate these non-linear constants through statistical estimation techniques.

1.1.1 The stress-strain curve

The significant points from fig 1.1.1 are as follows:

1. A - Proportional Limit
2. B - Elastic Limit
3. y - Yield Point
4. u - Ultimate Tensile Strength
5. f - Fracture point

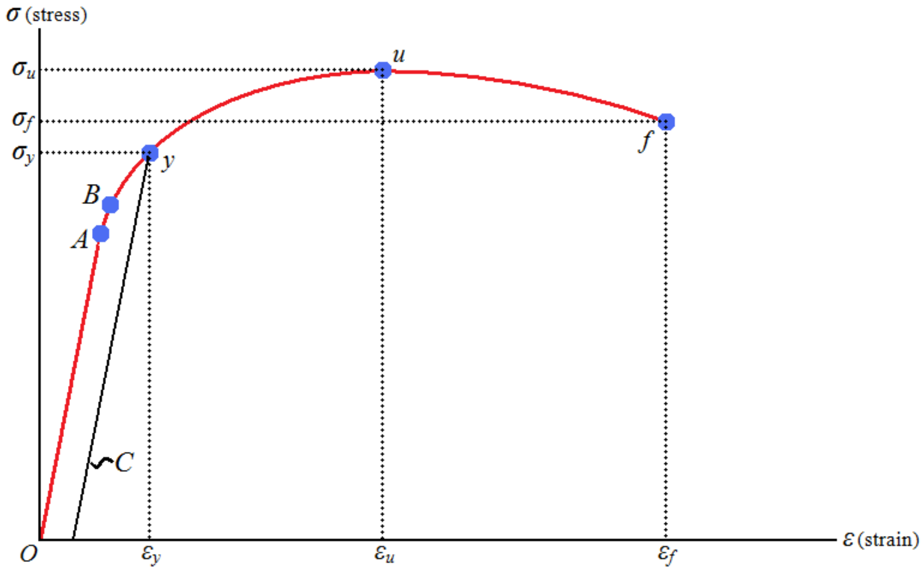

Figure 1.1: The Stress Strain Curve for a Ductile Material

1.1.2 Non-Destructive Testing

There are multiple methods of figuring out the elastic constants of a material, most significant of them being destructive methods, where the material is stressed to its Ultimate Tensile Strength and then, a curve is fit to get the second order constants of the material. This results in the material specimen being destroyed and deemed useless. This method is good for laboratory conditions, but not in a real-world scenario.

To account for this, we employ a method of Non-Destructive testing using ultrasonics. Ultrasonic waves have the characteristic of extremely high strain rates, but the magnitude of strain itself is minimal, and the changes to material properties are thus negligible. These high strain rates result in interesting phenomenon occurring, which in turn helps us estimate parameters of the material, without any physical damage to the specimen.

1.1.3 Wave Propagation and Mixing

One of the main concepts that are used in this study are that of wave propagation in solid media and mixing of waves in linear and non-linear zones. To solve the differential equations involved, we use FDTD simulations. We employed multiple solvers to evaluate the equations and multiple approaches to solve the problem. This will be discussed in detail in upcoming chapters.

1.1.4 Statistical Inversion and Learning

The forward model is inverted by using purely statistical techniques. We employ various techniques from Support Vector Machines (SVM), Gaussian Mixture Models (GMMs), and Gaussian Processes. The mathematics and results will be discussed in detail in the coming chapters.

1.2 Outline of the Report

This report is organized into 6 chapters.

1. The current chapter gives a basic introduction to the project and explains very briefly what we hope to achieve and techniques we've employed with a little bit of background information.
2. Chapter 2 deals with Literature Review and what we worked on and the subsequent results of the same with reasons as to what method we finally adopted and why with a discussion about the same.
3. Chapter 3 describes the construction of the FDTD model for the forward problem and the collinear wave mixing approach that is taken by us and describes the problem and solution in detail.
4. Chapter 4 explores the sensitivity analysis of our constructed forward model with respect to various parameters of interest and and exploratory analysis of the inverse model.
5. Chapter 5 validates the inverse model and also describes the pitfalls of the model. We make the model more real world friendly and check its performance.
6. Chapter 6 summarizes our work and has a section on how this project can be pursued along with suggestions for experimental validation.

1.3 Applications of present work

The present work has wide range of uses from aircraft industry to the shipping industry. A manufacturing specific application of this current technique will be in the estimation of material parameters in forming process and cold working processes where materials undergo plastic deformation.

This technique will help us understand material deformation better and give us a physical insight into what the constants mean and at the same time help improve existing processes and diagnose issues in current processes.

CHAPTER 2

Literature Review

2.1 Introduction

Many papers were reviewed

CHAPTER 3

Description Of Simulations

This chapter presents the description of the simulations used for the current research. The objective of the simulation is to estimate the amplitude and frequency of the resultant generated wave post mixing. This is primarily due to the non-linear behaviour of the material in the mixing zone. The following are the steps followed to achieve this objective.

3.1 2D FDTD Simulation in Cartesian Coordinates

Finite Difference Time Domain (FDTD) method is widely used to solve wave propagation problems. In the present work the FDTD algorithm is implemented in 2D Orthogonal Cartesian Coordinates. [3]

3.2 Governing Equations

The primary governing equations for the 2D FDTD simulator are the non-linear wave propagation equations in the material. Let $u(y,t)$, $v(y,t)$ describe the motion of wave propagation for a longitudinal and transverse wave in a solid. The equations can be derived as such from the standard equation of motion in a solid material.

$$\frac{\partial \sigma_{xy}}{\partial y} = \rho \frac{\partial^2 u}{\partial t^2} \tag{3.1}$$

$$\sigma_{xy} = \sigma yx = \frac{\partial u}{\partial y}(\mu + m\frac{\partial v}{\partial y}) \tag{3.2}$$

$$\frac{\partial \sigma_{yy}}{\partial y} = \rho \frac{\partial^2 v}{\partial t^2} \tag{3.3}$$

In terms of the Lame constants λ and μ, and the third order elastic constants l, m, and n (the Murnaghan coefficients), the stress components can be related to the displacement gradients through + [4]

$$\sigma_{yy} = (\lambda + 2\mu)\frac{\partial v}{\partial y} + (l + 2m)\frac{\partial v}{\partial y}^2 + \frac{m}{2}\frac{\partial u}{\partial y}^2 \tag{3.4}$$

Substituting the previous two relations to the standard wave equation, we get

$$\frac{\partial^2 u}{\partial t^2} - c_t^2 \frac{\partial^2 u}{\partial y^2} = \beta_t c_t^2 \frac{\partial}{\partial y}(\frac{\partial u}{\partial y}\frac{\partial v}{\partial y}) \tag{3.5}$$

$$\frac{\partial^2 v}{\partial t^2} - c_l^2 \frac{\partial^2 v}{\partial y^2} = \beta_l c_l^2 \frac{\partial v}{\partial y}\frac{\partial^2 v}{\partial y^2} + \beta_t c_t^2 \frac{\partial u}{\partial y}\frac{\partial^2 u}{\partial y^2} \tag{3.6}$$

Where, $c_L = \sqrt{(\lambda + 2\mu)\rho}$, $c_T = \sqrt{\mu\rho}$

$$\beta_L = 3 + \frac{2(l+m)}{\lambda + 2\mu} \tag{3.7}$$

$$\beta_T = \frac{\lambda + 2\mu}{\mu} + \frac{m}{\mu} \tag{3.8}$$

3.2.1 Discretization of the wave-equation

This wave equation, can be discretized as a FDTD grid, which using the staggered method of FDTD solver along with the central differencing technique results in a equation similar to this:

$$\begin{aligned}
\frac{u_k^{n+1} - 2 * u_k^n + u_k^{n-1}}{\Delta t^2} = \\
& c_T^2 \frac{u_{k+1}^n - 2u_k^n + u_{k-1}^n}{\Delta y^2} \\
& + \beta_T c_T^2 (\frac{u_{k+1}^n - 2u_k^n + u_{k-1}^n}{\Delta y^2} \frac{v^n k + 1 - v_{k-1}^n}{2\Delta y} \\
& + \frac{v_{k+1}^n - 2v_k^n + v_{k-1}^n}{\Delta y^2} \frac{u^n k + 1 - u_{k-1}^n}{2\Delta y})
\end{aligned} \tag{3.9}$$

$$\begin{aligned}
u_k^{n+1} = \\
& 2 * u_k^n - u_k^{n-1} + \Delta t^2 c_T^2 (\frac{u_{k+1}^n - 2u_k^n + u_{k-1}^n}{\Delta y^2} \\
& + \beta_T (\frac{u^n k + 1 - u_{k-1}^n}{2\Delta y} \frac{u_{k+1}^n - 2u_k^n + u_{k-1}^n}{\Delta y^2} + \frac{v_{k+1}^n - 2v_k^n + v_{k-1}^n}{\Delta y^2} \frac{u^n k + 1 - u_{k-1}^n}{2\Delta y}))
\end{aligned} \tag{3.10}$$

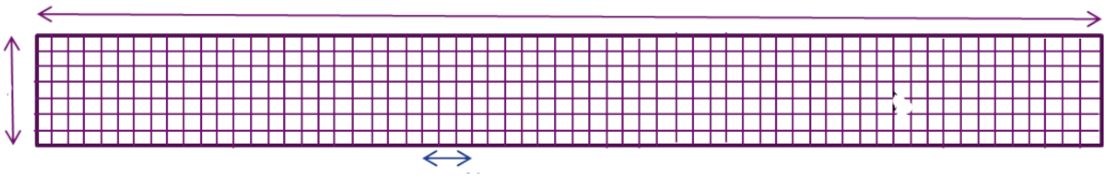

Figure 3.1: Schematic of the solid material as an FDTD grid

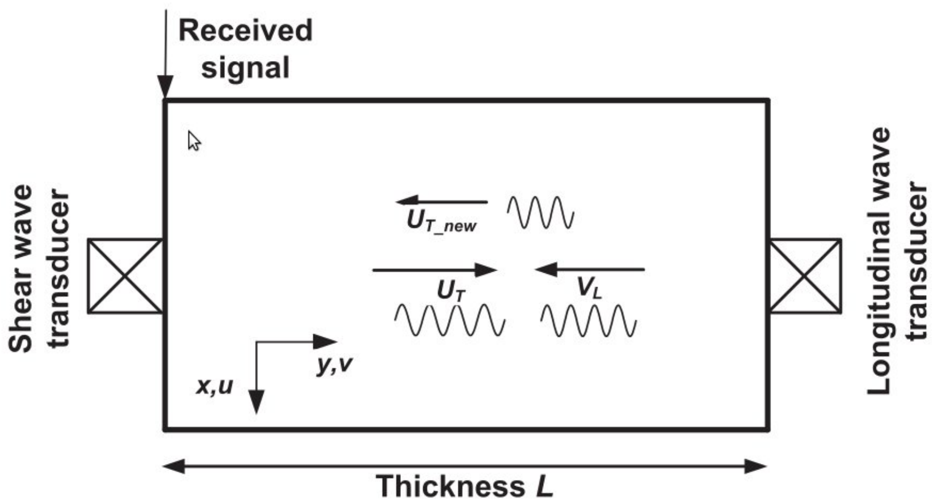

Figure 3.2: Schematic of the setup the FDTD simulation is mimicking

3.2.2 Numerical Considerations

Stability Criteria

A finite difference scheme is stable if the errors made at one time step of the calculation do not cause the errors to increase as the computations are continued. A neutrally stable scheme is one in which errors remain constant as the computations are carried forward. If the errors decay and eventually damp out, the numerical scheme is said to be stable. If, on the contrary, the errors grow with time the numerical scheme is said to be unstable. The stability of numerical schemes can be investigated by performing von Neumann stability analysis. For time-dependent problems, stability guarantees that the numerical method produces a bounded solution whenever the solution of the exact differential equation is bounded. Stability, in general, can be difficult to investigate, especially when the equation under consideration is nonlinear.

In the FDTD for sound waves, the main stability criteria is the **Courant** condition. This constraint ensures that the errors in the numerical simulation get damped out to give a fairly accurate estimate of the actual solution.

$$\Delta t \leq \frac{1}{C} \frac{1}{\sqrt{\frac{1}{(\Delta x)^2} + \frac{1}{(\Delta y)^2}}} \tag{3.11}$$

$$\Delta x = \Delta y, \Delta t \leq \frac{1}{C} \frac{1}{\sqrt{2}} \tag{3.12}$$

CHAPTER 4

Sensitivity Analysis

CHAPTER 5

Inverse Model

CHAPTER 6

Summary and Future Work

APPENDIX A

Appendix

A.1 Solver Code

```python
import numpy as np
import scipy as sp
import defaults as df
from math import sin, pi, cos
from matplotlib.pyplot import imshow, plot, show, draw, pause, clim, figure
import sys
from constants import *
#from matplotlib import figure
class Solver:

    Simulation = None
    Location = None
    Width = None
    #Create a Movie Variable to calculate number of movies and plots, to bring them up when necessary. add arguments to put it in grid, instead of what's happening here. This is hardcoded waste.
    def putMovie(self, pauseTime):
        data = np.reshape(self.Simulation.Grid[:,:,0,1], (self.Simulation.ElementSpan[0],self.Simulation.ElementSpan[1]))          #          #
        figure("Wave Movie Transverse")
        imshow(data)
        clim([-1e-8,1e-8])
```

```
        draw()
        pause(pauseTime)

        data = np.reshape(self.Simulation.Grid[:,:,1,1], (self.Simulation.ElementSpan[0],self.Simulation.ElementSpan[1]))    #    #
        figure("Wave Movie Longitudinal")
        imshow(data)
        clim([-1e-8,1e-8])
        draw()
        pause(pauseTime)



    def putSource(self, i, frequency, index, waveType = 0):
        #Multiply with gaussian to remove edge effects.

        #Adding default Source

        #waveType 0 - Transverse. 1 - Longitudinal

        X_S = round(self.Location[0])
        Y_S = slice(round(self.Simulation.ElementSpan[0]/2) - round(self.Width[1]/2), round(self.Simulation.ElementSpan[0]/2) + round(self.Width[1]/2))

        #Raised Cosine Pulse.

        self.Simulation.Grid[Y_S,index,waveType,2] = (1-cos(2*pi*frequency*i*self.Simulation.Dt/self.Simulation.Pulses))*cos(2*pi*frequency*i*self.Simulation.Dt)*1e-8

        #Sine Pulse. Trying to recreate the paper
```

```
        #self.Simulation.Grid[Y_S,index,waveType,2] = sin(2*pi*frequency*i*self.Simulation.Dt)*1e-8

        #print self.Simulation.Grid[X_S, round(self.Simulation.ElementSpan[1]/2) - round(self.Width[0]/2) + 1,0,2]

    #Line Sources only, currently. Multiple Sources must be accounted for, Must think of a matrix solution. So much fight for something that might not even work. Pain.
    def setSource(self, Location = None, Width = None, Theta = None):

        if Location is None:
            self.Location = [df.LOCATION*self.Simulation.Dimensions[0]/self.Simulation.Dx]
        else:
            self.Location.append(Location*self.Simulation.Dimensions[0]/self.Simulation.Dx)

        if Width is None:
            self.Width = [(df.WIDTH/self.Simulation.Dx)*D for D in self.Simulation.Dimensions]
        else:
            self.Width.append((Width/self.Simulation.Dx)*D for D in self.Simulation.Dimensions)

        if Theta is None:
            self.Location = [df.THETA]
        else:
            self.Location.append(Theta)



```

```python
    def Solve(self):
        #First Equation We'll be solving will be the standard wave equation.
        self.setSource()
        #Setting the Source first. Now, let's solve the DE like a boss

        _X = slice(0,self.Simulation.ElementSpan[0]-2)
        X = slice(1,self.Simulation.ElementSpan[0]-1)
        X_ = slice(2,self.Simulation.ElementSpan[0])

        _Y = slice(0,self.Simulation.ElementSpan[1]-2)
        Y = slice(1,self.Simulation.ElementSpan[1]-1)
        Y_ = slice(2,self.Simulation.ElementSpan[1])

        #_X indicates previous X coordinate and X_ indicts the one after
        r_var = round(self.Simulation.Time/self.Simulation.Dt)
        #print "Total Iterations are " , r_var
        c_t2 = pow(self.Simulation.MaterialProperties.WaveVelocityT,2)
        c_l2 = pow(self.Simulation.MaterialProperties.WaveVelocityL,2)

       # sdata = sp.zeros((r_var,1))
        for i in range(1,int(r_var)):
            dv_y = (self.Simulation.Grid[X,Y_,1,1] - self.Simulation.Grid[X,_Y,1,1])/(2*self.Simulation.Dx)
            d2v_y = (self.Simulation.Grid[X,Y_,1,1] - 2*self.Simulation.Grid[X,Y,1,1] + self.Simulation.Grid[X,_Y,1,1])/pow(self.Simulation.Dx,2)
            du_y = (self.Simulation.Grid[X,Y_,0,1] - self.Simulation.Grid[X,_Y,0,1])/(2*self.Simulation.Dx)
```

```
            d2u_y = (self.Simulation.Grid[X,Y_,0,1] - 2*self.
                ↪ Simulation.Grid[X,Y,0,1] + self.Simulation.Grid[X,
                ↪ _Y,0,1])/pow(self.Simulation.Dx,2)

            #Solving for Displacements in the X directio

            self.Simulation.Grid[X,Y,0,2] = 2*self.Simulation.Grid[X
                ↪ ,Y,0,1] - self.Simulation.Grid[X,Y,0,0] + pow(self.
                ↪ Simulation.Dt,2)*(c_t2*d2u_y + self.Simulation.
                ↪ MaterialProperties.BetaT*c_t2*(dv_y*d2u_y + du_y*
                ↪ d2v_y))
            self.Simulation.Grid[X,Y,1,2] = 2*self.Simulation.Grid[X
                ↪ ,Y,1,1] - self.Simulation.Grid[X,Y,1,0] + pow(self.
                ↪ Simulation.Dt,2)*(c_l2*d2v_y + self.Simulation.
                ↪ MaterialProperties.BetaL*c_l2*dv_y*d2v_y + self.
                ↪ Simulation.MaterialProperties.BetaT*c_t2*du_y*d2u_y
                ↪ )



            self.Simulation.SourceSignal[i,0] = sum(self.Simulation.
                ↪ Grid[:,-2,0,2])/self.Simulation.Grid.shape[1]


            self.Simulation.SData[i,0] = sum(self.Simulation.Grid
                ↪ [:,1,1,2])/self.Simulation.Grid.shape[1]

            #self.Simulation.SData[i,0] =  sum(self.Simulation.Grid
                ↪ [:,1,0,2])/self.Simulation.Grid.shape[1]

            #print self.Simulation.Grid[15,15,0,2]

```

```
            #Boundary COnditions. Making the ends soft reflections. Let's see how that works out.

            '''
            if self.Simulation.Mixing is not True:
                self.Simulation.Grid[-1,:,0,2] = self.Simulation.Grid[-2,:,0,2]
            else:
            #    self.Simulation.Grid[:,0,1,2] = self.Simulation.Grid[:,1,1,2]
            #    self.Simulation.Grid[:,0,0,2] = self.Simulation.Grid[:,1,0,2]
                self.Simulation.Grid[:,-1,0,2] = self.Simulation.Grid[:,-2,0,2]
            #    self.Simulation.Grid[:,-2,1,2] = self.Simulation.Grid[:,-1,1,2]

             #Updates go Here
            '''


            if(i <= round(self.Simulation.Pulses*(1.0/(self.Simulation.WaveProperties.Frequency))/self.Simulation.Dt)):
                self.putSource(i, self.Simulation.WaveProperties.Frequency, 0, TRANSVERSE)
            else:
                self.Simulation.Grid[:,1,0,2] = self.Simulation.Grid[:,0,0,2]
                self.Simulation.Grid[:,-2,0,2] = self.Simulation.Grid[:,-1,0,2]
               #self.Simulation.Grid[:,0,1,2] = self.Simulation.Grid[:,1,1,2]
```

```


        if self.Simulation.Mixing == True:

            if(i <= round(self.Simulation.Pulses*(1.0/(0.997*4*self.Simulation.WaveProperties.Frequency))/self.Simulation.Dt)):
                self.putSource(i,0.997*4*self.Simulation.WaveProperties.Frequency,-1,LONGITUDINAL)
            else:
                self.Simulation.Grid[:,-1,1,2] = self.Simulation.Grid[:,-2,1,2]
                self.Simulation.Grid[:,0,1,2] = self.Simulation.Grid[:,1,1,2]
                #self.Simulation.Grid[:,-1,0,2] = self.Simulation.Grid[:,-2,0,2]

        self.Simulation.Grid[:,:,1,0] = self.Simulation.Grid[:,:,1,1]
        self.Simulation.Grid[:,:,1,1] =  self.Simulation.Grid[:,:,1,2]

        self.Simulation.Grid[:,:,0,0] = self.Simulation.Grid[:,:,0,1]
        self.Simulation.Grid[:,:,0,1] =  self.Simulation.Grid[:,:,0,2]
#         print i
        if i%round(0.05*r_var) == 0:
            #print i
            if self.Simulation.ViewMovie == True:
                self.putMovie(0.01)
            sys.stdout.write('=='*int(round(i/round(0.1*r_var))))
```

```

                #p.plot.show()
        #print self.Simulation.MaterialProperties.BetaL, self.Simulation.MaterialProperties.BetaT, self.Simulation.MaterialProperties.WaveVelocityL, self.Simulation.Dt

    '''
        figure("Source Signal")
        plot(self.Simulation.SourceSignal)

        pause(0.01)
        figure("Non Linear Signal")
        plot(self.Simulation.SData)
        show()
    '''

#           np.save("TotalSignal",self.Simulation.SourceSignal)
#           np.save("LinSignal",sdata)
    def __init__(self, Simulation = None):
        if Simulation is None:
            raise ValueError("Simulation Cannot be None. Please Initialize a New Simulation to proceed")
        else:
            self.Simulation = Simulation
            self.Solve()

if __name__ == "__main__":
    raise Exception("Cannot run file as a standalone file. Please run through proper initialized channels")
```

A.2 Problem Formulation code Code

```
from data import waveProperties, materialProperties
import numpy as np
import scipy as sp
import matplotlib as mp
import defaults as df
import sys
from solver import Solver as sl
import scipy.io as sio
from matplotlib.pyplot import plot,figure


# ######################################################################
#Rules of code: Class elements always begin with a capital letter. Defaults are always allcaps. Arguments to functions to mimic class members.
# #######################################################################

class simulation:

    def save(self, filename):
        sio.savemat(filename,{"SData":self.SData, "SourceSignal":self.SourceSignal})

    def setMixing(self, val):
        self.Mixing = val

    def setStep(self, Dx):
```

```
        #Courant Condition check
        return (Dx/self.MaterialProperties.WaveVelocityL)/2

    def setMesh(self):

        if self.Mesh == 0:
            return (float)(self.WaveProperties.WaveLength/8.0)
        elif self.Mesh == 1:
            return (float)(self.WaveProperties.WaveLength/12.0)
        elif self.Mesh == 2:
            return (float)(self.WaveProperties.WaveLength/64.0)
        elif self.Mesh == 3:
            return (float)(self.WaveProperties.WaveLength/128.0)

    #Time is of type float; Dimensions is a list of floats.


    def setParam(self, paramName, value):

        if paramName == 'l':
            self.MaterialProperties.l = value
            #self.MaterialProperties.BetaT = (self.MaterialProperties.Lambda + 2*self.MaterialProperties.Mu)/self.MaterialProperties.Mu + self.MaterialProperties.m/self.MaterialProperties.Mu
            self.MaterialProperties.refreshParams()
        if paramName == 'm':
            self.MaterialProperties.m = value
            self.MaterialProperties.refreshParams()

        if paramName == 'BetaT':
            self.MaterialProperties.BetaT = value
```

```python

    def getParam(self, paramName):

        if paramName == 'l':
            return self.MaterialProperties.l
        if paramName == 'm':
            return self.MaterialProperties.m
        if paramName == 'BetaT':
            return self.MaterialProperties.BetaT

        return 0


    def __init__(self, MaterialProperties = None, WaveProperties = None, Reflections = None, Dimensions = None, WaveGuide = None, Mesh = None, Pulses = None):

        if MaterialProperties is None:
            self.MaterialProperties = materialProperties()
        else:
            self.MaterialProperties = MaterialProperties

        if WaveProperties is None:
            self.WaveProperties = waveProperties()
        else:
            self.WaveProperties = WaveProperties

        if Reflections is None:
            self.Reflections = df.REFLECTIONS
        else:
            self.Reflections = Reflections

        if Dimensions is None:
```

```
            self.Dimensions = df.DIMENSIONS
        else:
            self.Dimensions = Dimensions

        if WaveGuide is None:
            self.WaveGuide = df.WAVEGUIDE
        else:
            self.WaveGuide = WaveGuide

        if Mesh is None:
            self.Mesh = df.MESH
        else:
            self.Mesh = Mesh

        if Pulses is None:
            self.Pulses = df.PULSES
        else:
            self.Pulses = Pulses

        self.Time = 2*self.Reflections*self.Dimensions[1]/self.MaterialProperties.WaveVelocityL

        #1D, 2D or 3D
        self.DimensionCount = len(self.Dimensions)
##        self.WaveProperties.WaveVelocity = self.MaterialProperties.WaveVelocity
        self.WaveProperties.WaveLength = (float) (self.MaterialProperties.WaveVelocityL/self.WaveProperties.Frequency)
        self.Mixing = False
        self.Dx = self.setMesh()
        self.Dt = self.setStep(self.Dx)

```

```
            #print self.Dx
            ##List of elementsb
            self.ElementSpan = [round(X/self.Dx) for X in self.
                ↪ Dimensions]

            #Append Dimensions
            self.ElementSpan.append(3)
            #Append Times
            self.ElementSpan.append(3)

            self.Grid = sp.zeros(tuple(self.ElementSpan), float)
            self.NLGrid = sp.zeros(tuple(self.ElementSpan), float)
            self.SourceSignal = sp.zeros((round(self.Time/self.Dt),1))
            self.SData = sp.zeros((round(self.Time/self.Dt),1))
            self.ViewMovie = False
            self.viewPlot = True

def __init__():
    args = sys.argv
    args = [arg.replace('--','') for arg in args]
    names = []
    sim = simulation()
    print sim.Dt
    if 'mixing' in args:
        sim.setMixing(True)
    if 'movie' in args:
        sim.ViewMovie = True
    solution = sl(sim)

    if 'noplot' in args:
        pass
    else:
        figure(5)
```

```
            plot(sim.SData)

        if 'save' in args:
            try:
                ind = args.index('savenames')
                names.append(args[ind+1])
                names.append(args[ind+2])
            except:
                print "Using Default File names to save data"
                names.append("TotalSignal")
                names.append("NLinSignal")
            sio.savemat(names[0],{names[0]:sim.SourceSignal})
            sio.savemat(names[1],{names[1]:sim.SData})



if __name__ == "__main__":
    __init__()
```

```
import defaults as df
from math import sqrt


## These classes are created to create a default set of elements. I will implement a file reader to get elelment data later. createing a new object of this type ensures that we get a nice default simulation. Let's hope this works. Solver is yet to be implemented. Sigh

class waveProperties:
    def __init__(self, Frequency = None):
        if Frequency is None:
            self.Frequency = df.FREQUENCY
        else:
            self.Frequency = Frequency

        self.WaveLength = None


class materialProperties:
    def __init__(self, Mu = None, K = None, Rho = None,  A = None, B = None, C = None,l = None, m = None, Lambda = None):

        ##Initialize All defaults if none.

        if Mu is None:
            self.Mu = df.MU
        else:
            self.Mu = Mu

        if K is None:
            self.K = df.K
```

```
        else:
            self.K = K

        if Rho is None:
            self.Rho = df.RHO
        else:
            self.Rho = Rho

        if A is None:
            self.A = df.A
        else:
            self.A = A

        if B is None:
            self.B = df.B
        else:
            self.B = B

        if C is None:
            self.C = df.C
        else:
            self.C = C

        if l is None:
            self.l = df.l
        else:
            self.l = l

        if m is None:
            self.m = df.m
        else:
            self.m = m

```

```
        if Lambda is None:
            self.Lambda = df.Lambda
        else:
            self.Lambda = Lambda

        self.WaveVelocityL = sqrt((self.Lambda + (2*self.Mu))/self.Rho)
        self.WaveVelocityT = sqrt(self.Mu/self.Rho)
        self.BetaL = 3 + 2*(self.l + 2*self.m)/(self.Lambda + 2*self.Mu)
        self.BetaT = (self.Lambda + 2*self.Mu)/self.Mu + self.m/self.Mu

    def refreshParams(self):


        self.WaveVelocityL = sqrt((self.Lambda + (2*self.Mu))/self.Rho)
        self.WaveVelocityT = sqrt(self.Mu/self.Rho)
        self.BetaL = 3 + 2*(self.l + 2*self.m)/(self.Lambda + 2*self.Mu)
        self.BetaT = (self.Lambda + 2*self.Mu)/self.Mu + self.m/self.Mu


class waveGuide:

    def __init__(self, Boundary = None):
        if Boundary is None:
            self.Boundary = df.BOUNDARY
        else:
            self.Boundary = Boundary

```

```
## Boundary Legend
## 0 - All reflecting
## 1 - Sides Reflecting Ends PML
## 2 - Sides PML Ends Reflecting
## 3 - Everything PML
```

1 LONGITUDINAL = 1

2 TRANSVERSE = 0

```
from formulation import simulation
from solver import Solver as sl

#Limit of L and M in terms of percentages. How do we combine this?
    ↪ We need to run experiments, check correlations and all. Let's
    ↪ see if it has any effect:w

__LIMIT = 10
__STEP = 1
for percent in range(-int(round(__LIMIT)), int(round(__LIMIT))+1,
    ↪ __STEP):
    sim = simulation()
    oldl = sim.getParam('BetaT')
    print percent/100.0
    newl = oldl*(1 + (percent/100.0))
    print oldl, newl
    sim.setParam('BetaT', newl)
    sim.setMixing(True)
    sl(sim)
    sim.save("%d.mat"%percent)

'''
for percent in range(-int(round(__LIMIT)), int(round(__LIMIT))+1,
    ↪ __STEP):
    sim = simulation()
    oldl = sim.getParam('m')
    print percent/100.0
    newl = oldl*(1 + (percent/100.0))
    print oldl, newl
    sim.setParam('m', newl)
    sim.setMixing(True)
    sl(sim)
    sim.save("Simulation_Save_m_%d_percent.mat"%percent)
```

30 ' ' '

```
import numpy as np
import scipy.io as sp
from matplotlib import pyplot as plt
import os

__DIR = "./data/sensitivity/tentoten"
__TOTALLENGTH = 2048
__STARTINDEX = 4900
__ENDINDEX = 5600
__PADDING = __TOTALLENGTH - (__ENDINDEX - __STARTINDEX)
__FILE = "amplitude_BetaT1010.txt"
files = [os.path.join(__DIR,f) for f in os.listdir(__DIR)]

fi = open(__FILE,'w+')


def fft(signal):
    fftsignal = np.zeros(__TOTALLENGTH)
    #fftsignal[0:(__TOTALLENGTH - __PADDING)] = signal[__STARTINDEX:__ENDINDEX]
    fftsignal_2 = signal[__STARTINDEX:__ENDINDEX]
    ftp = abs(np.fft.fft(fftsignal_2))
    plot = plt.plot(ftp)
    return plot

def ampcalc(data):
    return abs(min(data) - max(data))

for f in files:
    print f.split('/')
    datafile = sp.loadmat(f)
    fftplot = fft(datafile['SourceSignal'])
    amplitude = ampcalc(datafile['SourceSignal'])
```

```
        plt.savefig("%s.png"%f.split('/')[4])
        plt.close()
        fi.write('%s - %.25f\n'%(f.split('/')[4], amplitude))
```

```
from sklearn.gaussian_process import GaussianProcess as GMM
#from sklearn.svm import SVR as GMM
import numpy as np
import matplotlib.pyplot as plt
import scipy as sp
FILE = 'data/sheet.csv'
dataset = np.vstack(set(map(tuple,np.genfromtxt(FILE, delimiter=',')
    ↪ )))

def addNoise(snr):
    signal = dataset[:,-1]
    #print signal
    signalstd = np.std(signal)
    noisestd = signalstd/np.sqrt(snr)
    noise = np.random.normal(0,noisestd,len(signal))
    datasetnoisy = dataset
    datasetnoisy[:,-1] = datasetnoisy[:,-1] + noise
    return datasetnoisy

def ensemble(value, noise):

    #mixture = GMM(C = 100)
    mixture = GMM()
    newdataset = addNoise(noise)
    temp = np.copy(newdataset[:,-1])
    #newdataset[:,-1] = newdataset[:,2]
    #newdataset[:,2] = temp
    #print newdataset[:,-1], newdataset[:,3]
    for ensemble in range(0,value):
        np.random.shuffle(newdataset)
        train = np.copy(newdataset[0:-10,:])
        test = np.copy(newdataset[-10:-1,:])
        test_pred = np.copy(test)
```

```
        mixture.fit(newdataset[0:-10,0:-2],newdataset[0:-10,-1])
        preds = mixture.predict(newdataset[-10:-1,0:-2])
        test_pred[:,-1] = preds
        errorabs =  abs(dataset[-10:-1,-1]-preds)/(dataset[-10:-1,-1])
        meanerrorabs = np.mean(errorabs)
        stderrorabs = np.std(errorabs)
        #print errorabs
        print meanerrorabs, stderrorabs
        #plt.plot(abs(dataset[-10:-1,-1]-preds))
        #plt.ylim(-5e-12,5e-12)
        #plt.scatter(dataset[-10:-1,0],dataset[-10:-1,-1])
        #plt.plot(preds)
        #plt.show()
        np.savetxt('data/forward_train_%d_snr_%d.csv'%(ensemble, noise),train ,delimiter=',')
        np.savetxt('data/forward_test_%d_snr_%d.csv'%(ensemble,noise),test ,delimiter=',')
        np.savetxt('data/forward_test_predict_%d_snr_%d.csv'%(ensemble,noise),test_pred ,delimiter=',')
        #sp.io.savemat('data/train_%d_snr_%d.mat'%(ensemble,noise),train)
        #sp.io.savemat('data/test_%d_snr_%d.mat'%(ensemble,noise),test)
        #sp.io.savemat('data/test_predict_%d_snr_%d.mat'%(ensemble,noise),test_pred)
for noise in range(2,20,2):
    ensemble(1, noise)
```

```
FREQUENCY = 2.5e6
A = -3.1*(10^11)
B = 0
C = 0
BOUNDARY = 0 #Purely Reflecting
DIMENSIONS = [.010 , 0.030] #metres
MESH = 2 #0, 1, 2, 3 Coarse, Medium ,fine and extrafine mesh l/8, l/12, l/64, l/128
MU = 2.68e10
Lambda = 5.43e10
K = 76e9
RHO = 2719
TIME = 1.5 #seconds
WAVEGUIDE = 1
LOCATION = 0.5
THETA = 0
WIDTH = 0.25
PULSES = 10
REFLECTIONS = 2
l = -38.75e10
m = -35.8e10
```

REFERENCES

[1] L.K. Zaremboi and V. A. KrasilNkov, *Non-Linear Phenomena in the propagation of Elastic Waves in Solids*, Soviet Physics Uspekhi, 13(6), May-June 1971 : 778-800.

[2] Lyle H. Taylor, Fred R. Rollins, Jr. Ultrasonic Study of Three-Phonon Interactions. I. Theory, Physical Review, 136 (3A), 1964 : A 591 - 596.

[3] Kane Yee *Numerical solution of initial boundary value problems involving Maxwells equations in isotropic media*, IEEE Transactions on Antennas and Propagation 14 (3), 1966: 302307.

[4] S. Kuchler, T. Meurer, L. J. Jacobs, and J. Qu, J. Acoust. Soc. *Am. 125(3)*, 12931301 (2009)