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 of low-frequency shear Alfven waves and its interaction with the trapped charge particles in the Earths inner magnetosphere. This section is a brief introduction containing the background, outline and applications of the presented work.

CHAPTER 2

Future Work

CHAPTER 3

Appendix

3.1 Solver Code

```
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")
```

3.2 Problem Formulation code Code

```python
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
```

```
    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__()
```

```python
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

```python
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 , , ,

```python
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
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,epsilon = 1e-20)
    mixture = GMM()
    newdataset = addNoise(noise)
    for ensemble in range(0,value):
        np.random.shuffle(newdataset)
        mixture.fit(newdataset[0:-10,0:-2],newdataset[0:-10,-1])
        preds = mixture.predict(newdataset[-10:-1,0:-2])
        errorabs =  abs(dataset[-10:-1,-1]-preds)
        meanerrorabs = np.mean(errorabs)
        stderrorabs = np.std(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()
ensemble(5, 10)
```

```
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