

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

Place: Chennai

L Vijayaraghavan

Research Co-Guide

Professor

Dept. of Mechanical Engineering

IIT-Madras, 600 036

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

ACKNOWLEDGEMENTS	i
ABSTRACT	ii
LIST OF TABLES	v
LIST OF FIGURES	vi
1 Introduction	1
1.1 Background	1
1.1.1 The stress-strain curve	1
1.1.2 Non-Destructive Testing	2
1.1.3 Wave Propagation and Mixing	2
1.1.4 Statistical Inversion and Learning	3
1.2 Outline of the Report	3
1.3 Applications of present work	3
2 Literature Review	4
2.1 Introduction	4
3 Description Of Simulations	5
3.1 2D FDTD Simulation in Cartesian Coordinates	5
3.2 Governing Equations	5
4 Sensitivity Analysis	7
5 Inverse Model	8
6 Summary and Future Work	9
A Appendix	10

A.1 Solver Code	10
A.2 Problem Formulation code Code	18

LIST OF TABLES

LIST OF FIGURES

1.1	The Stress Strain Curve for a Ductile Material	2
-----	--	---

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

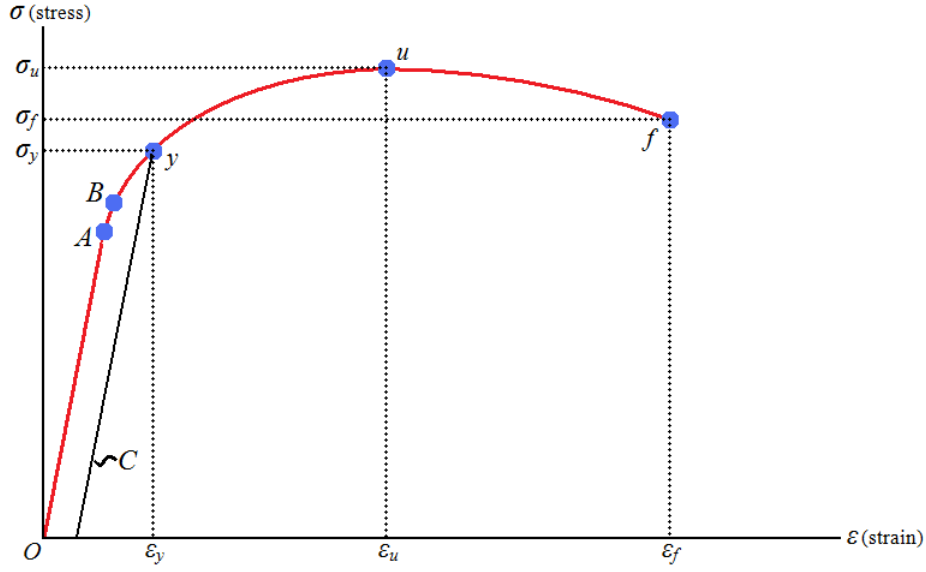


Figure 1.1: The Stress Strain Curve for a Ductile Material

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.

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

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

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

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

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

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

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

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

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

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

$$\beta_T = \frac{\lambda + 2\mu}{\mu} + \frac{m}{\mu} \quad (3.8)$$

CHAPTER 4

Sensitivity Analysis

CHAPTER 5

Inverse Model

CHAPTER 6

Summary and Future Work

APPENDIX A

Appendix

A.1 Solver Code

```
1 import numpy as np
2 import scipy as sp
3 import defaults as df
4 from math import sin, pi, cos
5 from matplotlib.pyplot import imshow, plot, show, draw, pause, clim,
    ↪ figure
6 import sys
7 from constants import *
8 #from matplotlib import figure
9 class Solver:
10
11     Simulation = None
12     Location = None
13     Width = None
14     #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.
15     def putMovie(self, pauseTime):
16         data = np.reshape(self.Simulation.Grid[:, :, 0, 1], (self.
            ↪ Simulation.ElementSpan[0], self.Simulation.ElementSpan
            ↪ [1]))          #          #
17         figure("Wave_Movie_Transverse")
18         imshow(data)
19         clim([-1e-8, 1e-8])
```

```

20     draw ()
21     pause (pauseTime)
22
23     data = np.reshape ( self . Simulation . Grid [:, :, 1, 1], ( self .
        ↪ Simulation . ElementSpan [0], self . Simulation . ElementSpan
        ↪ [1]))          #          #
24     figure ("Wave_Movie_Longitudinal")
25     imshow ( data )
26     clim ([-1e-8, 1e-8])
27     draw ()
28     pause (pauseTime)
29
30
31
32     def putSource (self , i , frequency , index , waveType = 0):
33         #Multiply with gaussian to remove edge effects .
34
35         #Adding default Source
36
37         #waveType 0 – Transverse . 1 – Longitudinal
38
39         X_S = round ( self . Location [0])
40         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))
41
42         #Raised Cosine Pulse .
43
44         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
45
46         #Sine Pulse . Trying to recreate the paper

```

```

47         #self.Simulation.Grid[Y_S, index, waveType, 2] = sin(2*pi*
           ↪ frequency*i*self.Simulation.Dt)*1e-8
48
49         #print self.Simulation.Grid[X_S, round(self.Simulation.
           ↪ ElementSpan[1]/2) - round(self.Width[0]/2) + 1, 0, 2]
50
51         #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.
52     def setSource(self, Location = None, Width = None, Theta = None)
           ↪ :
53
54         if Location is None:
55             self.Location = [df.LOCATION*self.Simulation.Dimensions
                               ↪ [0]/self.Simulation.Dx]
56         else:
57             self.Location.append(Location*self.Simulation.Dimensions
                                   ↪ [0]/self.Simulation.Dx)
58
59         if Width is None:
60             self.Width = [(df.WIDTH/self.Simulation.Dx)*D for D in
                            ↪ self.Simulation.Dimensions]
61         else:
62             self.Width.append((Width/self.Simulation.Dx)*D for D in
                                ↪ self.Simulation.Dimensions)
63
64         if Theta is None:
65             self.Location = [df.THETA]
66         else:
67             self.Location.append(Theta)
68
69
70

```

```

71     def Solve(self):
72         #First Equation We'll be solving will be the standard wave
73             ↪ equation.
74         self.setSource()
75         #Setting the Source first. Now, let's solve the DE like a
76             ↪ boss
77
78         _X = slice(0,self.Simulation.ElementSpan[0]-2)
79         X = slice(1,self.Simulation.ElementSpan[0]-1)
80         X_ = slice(2,self.Simulation.ElementSpan[0])
81
82         _Y = slice(0,self.Simulation.ElementSpan[1]-2)
83         Y = slice(1,self.Simulation.ElementSpan[1]-1)
84         Y_ = slice(2,self.Simulation.ElementSpan[1])
85
86         #_X indicates previous X coordinate and X_ indicts the one
87             ↪ after
88         r_var = round(self.Simulation.Time/self.Simulation.Dt)
89         #print "Total Iterations are " , r_var
90         c_t2 = pow(self.Simulation.MaterialProperties.WaveVelocityT
91             ↪ ,2)
92         c_l2 = pow(self.Simulation.MaterialProperties.WaveVelocityL
93             ↪ ,2)
94
95         # sdata = sp.zeros(( r_var ,1))
96         for i in range(1,int(r_var)):
97             dv_y = (self.Simulation.Grid[X,Y_,1,1] - self.Simulation
98                 ↪ .Grid[X,_Y,1,1])/(2*self.Simulation.Dx)
99             d2v_y = (self.Simulation.Grid[X,Y_,1,1] - 2*self.
100                 ↪ Simulation.Grid[X,Y,1,1] + self.Simulation.Grid[X,
101                 ↪ _Y,1,1])/pow(self.Simulation.Dx,2)
102             du_y = (self.Simulation.Grid[X,Y_,0,1] - self.Simulation
103                 ↪ .Grid[X,_Y,0,1])/(2*self.Simulation.Dx)

```

```

95         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)
96
97         #Solving for Displacements in the X directio
98
99         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))
100        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
           ↪ )
101
102
103
104        self.Simulation.SourceSignal[i,0] = sum(self.Simulation.
           ↪ Grid[:, -2,0,2])/self.Simulation.Grid.shape[1]
105
106
107        self.Simulation.SData[i,0] = sum(self.Simulation.Grid
           ↪[:, 1,1,2])/self.Simulation.Grid.shape[1]
108
109        #self.Simulation.SData[i,0] = sum(self.Simulation.Grid
           ↪ [:, 1,0,2])/self.Simulation.Grid.shape[1]
110
111        #print self.Simulation.Grid[15,15,0,2]
112
113        #Boundary COnditions. Making the ends soft reflections.

```

```

114         ↪ Let's see how that works out.
115     '''
116     if self.Simulation.Mixing is not True:
117         self.Simulation.Grid[-1,:,0,2] = self.Simulation.
118             ↪ Grid[-2,:,0,2]
119     else:
120     #     self.Simulation.Grid[:,0,1,2] = self.Simulation.
121         ↪ Grid[:,1,1,2]
122     #     self.Simulation.Grid[:,0,0,2] = self.Simulation.
123         ↪ Grid[:,1,0,2]
124     self.Simulation.Grid[:, -1,0,2] = self.Simulation.
125         ↪ Grid[:, -2,0,2]
126     #     self.Simulation.Grid[:, -2,1,2] = self.Simulation.
127         ↪ Grid[:, -1,1,2]
128
129     #Updates go Here
130     '''
131
132     if(i <= round(self.Simulation.Pulses*(1.0/(self.
133         ↪ Simulation.WaveProperties.Frequency)))/self.
134         ↪ Simulation.Dt)):
135         self.putSource(i, self.Simulation.WaveProperties.
136             ↪ Frequency, 0, TRANSVERSE)
137     else:
138         self.Simulation.Grid[:,1,0,2] = self.Simulation.Grid
139             ↪[:,0,0,2]
140         self.Simulation.Grid[:, -2,0,2] = self.Simulation.
141             ↪ Grid[:, -1,0,2]
142         #self.Simulation.Grid[:,0,1,2] = self.Simulation.
143             ↪ Grid[:,1,1,2]
144
145

```

```

135
136         if self.Simulation.Mixing == True:
137
138             if (i <= round(self.Simulation.Pulses*(1.0/(0.997*4*
139                 ↪ self.Simulation.WaveProperties.Frequency)) / self
140                 ↪ .Simulation.Dt)):
141                 self.putSource(i, 0.997*4*self.Simulation.
142                     ↪ WaveProperties.Frequency, -1, LONGITUDINAL)
143             else:
144                 self.Simulation.Grid[:, -1, 1, 2] = self.Simulation
145                     ↪ .Grid[:, -2, 1, 2]
146                 self.Simulation.Grid[:, 0, 1, 2] = self.Simulation.
147                     ↪ Grid[:, 1, 1, 2]
148                 #self.Simulation.Grid[:, -1, 0, 2] = self.
149                     ↪ Simulation.Grid[:, -2, 0, 2]
150
151                 self.Simulation.Grid[:, :, 1, 0] = self.Simulation.Grid
152                     ↪[:, :, 1, 1]
153                 self.Simulation.Grid[:, :, 1, 1] = self.Simulation.Grid
154                     ↪[:, :, 1, 2]
155
156                 self.Simulation.Grid[:, :, 0, 0] = self.Simulation.Grid
157                     ↪[:, :, 0, 1]
158                 self.Simulation.Grid[:, :, 0, 1] = self.Simulation.Grid
159                     ↪[:, :, 0, 2]
160
161                 print i
162             if i%round(0.05*r_var) == 0:
163                 #print i
164                 if self.Simulation.ViewMovie == True:
165                     self.putMovie(0.01)
166                 sys.stdout.write('=='*int(round(i/round(0.1*r_var))))
167                     ↪ )
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```



```

157         #p.plot.show()
158     #print self.Simulation.MaterialProperties.BetaL, self.
        ↳ Simulation.MaterialProperties.BetaT, self.Simulation.
        ↳ MaterialProperties.WaveVelocityL, self.Simulation.Dt
159
160     '''
161     figure("Source Signal")
162     plot(self.Simulation.SourceSignal)
163
164     pause(0.01)
165     figure("Non Linear Signal")
166     plot(self.Simulation.SData)
167     show()
168     '''
169
170     # np.save("TotalSignal",self.Simulation.SourceSignal)
171     # np.save("LinSignal",sdata)
172     def __init__(self, Simulation = None):
173         if Simulation is None:
174             raise ValueError("Simulation_Cannot_be_None._Please_
        ↳ Initialize_a_New_Simulation_to_proceed")
175         else:
176             self.Simulation = Simulation
177             self.Solve()
178
179     if __name__ == "__main__":
180         raise Exception("Cannot_run_file_as_a_standalone_file._Please_
        ↳ run_through_proper_initialized_channels")

```

A.2 Problem Formulation code Code

```
1 from data import waveProperties , materialProperties
2 import numpy as np
3 import scipy as sp
4 import matplotlib as mp
5 import defaults as df
6 import sys
7 from solver import Solver as sl
8 import scipy.io as sio
9 from matplotlib.pyplot import plot , figure
10
11
12 #
    ↪ #####
    ↪
13 #Rules of code: Class elements always begin with a capital letter.
    ↪ Defaults are always allcaps. Arguments to functions to mimic
    ↪ class members.
14 #
    ↪ #####
    ↪
15
16 class simulation:
17
18     def save(self , filename):
19         sio.savemat(filename , {"SData": self.SData , "SourceSignal":
            ↪ self.SourceSignal })
20
21     def setMixing(self , val):
22         self.Mixing = val
23
24     def setStep(self , Dx):
```

```

25         #Courant Condition check
26         return (Dx/self.MaterialProperties.WaveVelocityL)/2
27
28     def setMesh(self):
29
30         if self.Mesh == 0:
31             return (float)(self.WaveProperties.WaveLength/8.0)
32         elif self.Mesh == 1:
33             return (float)(self.WaveProperties.WaveLength/12.0)
34         elif self.Mesh == 2:
35             return (float)(self.WaveProperties.WaveLength/64.0)
36         elif self.Mesh == 3:
37             return (float)(self.WaveProperties.WaveLength/128.0)
38
39         #Time is of type float; Dimensions is a list of floats.
40
41
42     def setParam(self, paramName, value):
43
44         if paramName == 'l':
45             self.MaterialProperties.l = value
46             #self.MaterialProperties.BetaT = (self.
47                 ↪ MaterialProperties.Lambda + 2*self.
48                 ↪ MaterialProperties.Mu)/self.MaterialProperties.Mu +
49                 ↪ self.MaterialProperties.m/self.MaterialProperties.
50                 ↪ Mu
51
52             self.MaterialProperties.refreshParams()
53         if paramName == 'm':
54             self.MaterialProperties.m = value
55             self.MaterialProperties.refreshParams()
56
57         if paramName == 'BetaT':
58             self.MaterialProperties.BetaT = value

```

```

54
55     def getParam(self , paramName):
56
57         if paramName == 'l':
58             return self.MaterialProperties.l
59         if paramName == 'm':
60             return self.MaterialProperties.m
61         if paramName == 'BetaT':
62             return self.MaterialProperties.BetaT
63
64         return 0
65
66
67     def __init__(self , MaterialProperties = None, WaveProperties =
        ↪ None, Reflections = None, Dimensions = None, WaveGuide =
        ↪ None, Mesh = None, Pulses = None):
68
69         if MaterialProperties is None:
70             self.MaterialProperties = materialProperties()
71         else:
72             self.MaterialProperties = MaterialProperties
73
74         if WaveProperties is None:
75             self.WaveProperties = waveProperties()
76         else:
77             self.WaveProperties = WaveProperties
78
79         if Reflections is None:
80             self.Reflections = df.REFLECTIONS
81         else:
82             self.Reflections = Reflections
83
84         if Dimensions is None:

```

```

85         self.Dimensions = df.DIMENSIONS
86     else:
87         self.Dimensions = Dimensions
88
89     if WaveGuide is None:
90         self.WaveGuide = df.WAVEGUIDE
91     else:
92         self.WaveGuide = WaveGuide
93
94     if Mesh is None:
95         self.Mesh = df.MESH
96     else:
97         self.Mesh = Mesh
98
99     if Pulses is None:
100         self.Pulses = df.PULSES
101     else:
102         self.Pulses = Pulses
103
104     self.Time = 2*self.Reflections*self.Dimensions[1]/self.
        ↳ MaterialProperties.WaveVelocityL
105
106     #1D, 2D or 3D
107     self.DimensionCount = len(self.Dimensions)
108     ##         self.WaveProperties.WaveVelocity = self.MaterialProperties
        ↳ .WaveVelocity
109     self.WaveProperties.WaveLength = (float)(self.
        ↳ MaterialProperties.WaveVelocityL/self.WaveProperties.
        ↳ Frequency)
110     self.Mixing = False
111     self.Dx = self.setMesh()
112     self.Dt = self.setStep(self.Dx)
113

```

```

114         #print self.Dx
115         ##List of elementsb
116         self.ElementSpan = [round(X/self.Dx) for X in self.
            ↪ Dimensions]
117
118         #Append Dimensions
119         self.ElementSpan.append(3)
120         #Append Times
121         self.ElementSpan.append(3)
122
123         self.Grid = sp.zeros(tuple(self.ElementSpan), float)
124         self.NLGrid = sp.zeros(tuple(self.ElementSpan), float)
125         self.SourceSignal = sp.zeros((round(self.Time/self.Dt),1))
126         self.SData = sp.zeros((round(self.Time/self.Dt),1))
127         self.ViewMovie = False
128         self.viewPlot = True
129
130 def __init__():
131     args = sys.argv
132     args = [arg.replace('—',',') for arg in args]
133     names = []
134     sim = simulation()
135     print sim.Dt
136     if 'mixing' in args:
137         sim.setMixing(True)
138     if 'movie' in args:
139         sim.ViewMovie = True
140     solution = sl(sim)
141
142     if 'noplot' in args:
143         pass
144     else:
145         figure(5)

```

```

146         plot(sim.SData)
147
148     if 'save' in args:
149         try:
150             ind = args.index('savenames')
151             names.append(args[ind+1])
152             names.append(args[ind+2])
153         except:
154             print "Using Default File names to save data"
155             names.append("TotalSignal")
156             names.append("NLinSignal")
157             sio.savemat(names[0],{names[0]:sim.SourceSignal})
158             sio.savemat(names[1],{names[1]:sim.SData})
159
160
161
162 if __name__ == "__main__":
163     __init__()

```

```

1  import defaults as df
2  from math import sqrt
3
4
5  ## These classes are created to create a default set of elements. I
   → will implement a file reader to get element 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
6
7  class waveProperties:
8      def __init__(self, Frequency = None):
9          if Frequency is None:
10             self.Frequency = df.FREQUENCY
11          else:
12             self.Frequency = Frequency
13
14             self.WaveLength = None
15
16
17  class materialProperties:
18      def __init__(self, Mu = None, K = None, Rho = None, A = None, B
   → = None, C = None, l = None, m = None, Lambda = None):
19
20          ##Initialize All defaults if none.
21
22          if Mu is None:
23             self.Mu = df.MU
24          else:
25             self.Mu = Mu
26
27          if K is None:
28             self.K = df.K

```



```

29         else :
30             self.K = K
31
32         if Rho is None:
33             self.Rho = df.RHO
34         else :
35             self.Rho = Rho
36
37         if A is None:
38             self.A = df.A
39         else :
40             self.A = A
41
42         if B is None:
43             self.B = df.B
44         else :
45             self.B = B
46
47         if C is None:
48             self.C = df.C
49         else :
50             self.C = C
51
52         if l is None:
53             self.l = df.l
54         else :
55             self.l = l
56
57         if m is None:
58             self.m = df.m
59         else :
60             self.m = m
61

```

```

62         if Lambda is None:
63             self.Lambda = df.Lambda
64         else:
65             self.Lambda = Lambda
66
67         self.WaveVelocityL = sqrt((self.Lambda + (2*self.Mu))/self.
        ↪ Rho)
68         self.WaveVelocityT = sqrt(self.Mu/self.Rho)
69         self.BetaL = 3 + 2*(self.l + 2*self.m)/(self.Lambda + 2*self
        ↪ .Mu)
70         self.BetaT = (self.Lambda + 2*self.Mu)/self.Mu + self.m/self
        ↪ .Mu
71
72     def refreshParams(self):
73
74
75         self.WaveVelocityL = sqrt((self.Lambda + (2*self.Mu))/self.
        ↪ Rho)
76         self.WaveVelocityT = sqrt(self.Mu/self.Rho)
77         self.BetaL = 3 + 2*(self.l + 2*self.m)/(self.Lambda + 2*self
        ↪ .Mu)
78         self.BetaT = (self.Lambda + 2*self.Mu)/self.Mu + self.m/self
        ↪ .Mu
79
80
81 class waveGuide:
82
83     def __init__(self, Boundary = None):
84         if Boundary is None:
85             self.Boundary = df.BOUNDARY
86         else:
87             self.Boundary = Boundary
88

```

89 *## Boundary Legend*
90 *## 0 – All reflecting*
91 *## 1 – Sides Reflecting Ends PML*
92 *## 2 – Sides PML Ends Reflecting*
93 *## 3 – Everything PML*

1 LONGITUDINAL = 1

2 TRANSVERSE = 0

```

1 from formulation import simulation
2 from solver import Solver as sl
3
4 #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
5
6 __LIMIT = 10
7 __STEP = 1
8 for percent in range(-int(round(__LIMIT)), int(round(__LIMIT))+1,
   → __STEP):
9     sim = simulation()
10    oldl = sim.getParam('BetaT')
11    print percent/100.0
12    newl = oldl*(1 + (percent/100.0))
13    print oldl, newl
14    sim.setParam('BetaT', newl)
15    sim.setMixing(True)
16    sl(sim)
17    sim.save("%d.mat"%percent)
18
19 '''
20 for percent in range(-int(round(__LIMIT)), int(round(__LIMIT))+1,
   → __STEP):
21     sim = simulation()
22     oldl = sim.getParam('m')
23     print percent/100.0
24     newl = oldl*(1 + (percent/100.0))
25     print oldl, newl
26     sim.setParam('m', newl)
27     sim.setMixing(True)
28     sl(sim)
29     sim.save("Simulation_Save_m-%d_percent.mat"%percent)

```



```

1  import numpy as np
2  import scipy.io as sp
3  from matplotlib import pyplot as plt
4  import os
5
6  __DIR = "../data/sensitivity/tentoten"
7  __TOTALLENGTH = 2048
8  __STARTINDEX = 4900
9  __ENDINDEX = 5600
10 __PADDING = __TOTALLENGTH - (__ENDINDEX - __STARTINDEX)
11 __FILE = "amplitude_BetaT1010.txt"
12 files = [os.path.join(__DIR, f) for f in os.listdir(__DIR)]
13
14 fi = open(__FILE, 'w+')
15
16
17 def fft(signal):
18     fftsignal = np.zeros(__TOTALLENGTH)
19     #fftsignal[0:(__TOTALLENGTH - __PADDING)] = signal[__STARTINDEX:
20         ↪ __ENDINDEX]
21     fftsignal_2 = signal[__STARTINDEX:__ENDINDEX]
22     ftp = abs(np.fft.fft(fftsignal_2))
23     plot = plt.plot(ftp)
24     return plot
25
26 def ampcalc(data):
27     return abs(min(data) - max(data))
28
29 for f in files:
30     print f.split('/')
31     datafile = sp.loadmat(f)
32     fftplot = fft(datafile['SourceSignal'])
33     amplitude = ampcalc(datafile['SourceSignal'])

```

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



```

1 from sklearn.gaussian_process import GaussianProcess as GMM
2 #from sklearn.svm import SVR as GMM
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import scipy as sp
6 FILE = 'data/sheet.csv'
7 dataset = np.vstack((set(map(tuple,np.genfromtxt(FILE, delimiter=',')
    ↪ ))))
8
9 def addNoise(snr):
10     signal = dataset[:, -1]
11     #print signal
12     signalstd = np.std(signal)
13     noisestd = signalstd/np.sqrt(snr)
14     noise = np.random.normal(0, noisestd, len(signal))
15     datasetnoisy = dataset
16     datasetnoisy[:, -1] = datasetnoisy[:, -1] + noise
17     return datasetnoisy
18
19 def ensemble(value, noise):
20
21     #mixture = GMM(C = 100)
22     mixture = GMM()
23     newdataset = addNoise(noise)
24     temp = np.copy(newdataset[:, -1])
25     #newdataset[:, -1] = newdataset[:, 2]
26     #newdataset[:, 2] = temp
27     #print newdataset[:, -1], newdataset[:, 3]
28     for ensemble in range(0, value):
29         np.random.shuffle(newdataset)
30         train = np.copy(newdataset[0:-10,:])
31         test = np.copy(newdataset[-10:-1,:])
32         test_pred = np.copy(test)

```

```

33     mixture.fit(newdataset[0:-10,0:-2],newdataset[0:-10,-1])
34     preds = mixture.predict(newdataset[-10:-1,0:-2])
35     test_pred[:, -1] = preds
36     errorabs = abs(dataset[-10:-1,-1]-preds)/(dataset
        ↪ [-10:-1,-1])
37     meanerrorabs = np.mean(errorabs)
38     stderrorabs = np.std(errorabs)
39     #print errorabs
40     print meanerrorabs , stderrorabs
41     #plt.plot(abs(dataset[-10:-1,-1]-preds))
42     #plt.ylim(-5e-12,5e-12)
43     #plt.scatter(dataset[-10:-1,0],dataset[-10:-1,-1])
44     #plt.plot(preds)
45     #plt.show()
46     np.savetxt('data/forward_train_%d_snr_%d.csv'%(ensemble,
        ↪ noise),train,delimiter=',')
47     np.savetxt('data/forward_test_%d_snr_%d.csv'%(ensemble,noise
        ↪ ),test,delimiter=',')
48     np.savetxt('data/forward_test_predict_%d_snr_%d.csv'%(
        ↪ ensemble,noise),test_pred,delimiter=',')
49     #sp.io.savemat('data/train_%d_snr_%d.mat'%(ensemble,noise),
        ↪ train)
50     #sp.io.savemat('data/test_%d_snr_%d.mat'%(ensemble,noise),
        ↪ test)
51     #sp.io.savemat('data/test_predict_%d_snr_%d.mat'%(ensemble,
        ↪ noise),test_pred)
52 for noise in range(2,20,2):
53     ensemble(1, noise)

```

```

1 FREQUENCY = 2.5e6
2 A = -3.1*(10^11)
3 B = 0
4 C = 0
5 BOUNDARY = 0 #Purely Reflecting
6 DIMENSIONS = [.010 , 0.030] #metres
7 MESH = 2 #0, 1, 2, 3 Coarse, Medium, fine and extrafine mesh l/8, l
    ↪ /12, l/64, l/128
8 MU = 2.68e10
9 Lambda = 5.43e10
10 K = 76e9
11 RHO = 2719
12 TIME = 1.5 #seconds
13 WAVEGUIDE = 1
14 LOCATION = 0.5
15 THETA = 0
16 WIDTH = 0.25
17 PULSES = 10
18 REFLECTIONS = 2
19 l = -38.75e10
20 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)