

Solving the Wave Equation (IBVP) using Finite Element Method

Rohen Agarwal, Charlie Ray, Dennis McCann, Justin Chung

May 12, 2021

tocloft

Contents

1	Introduction	2
2	Model	3
2.1	Description of Problem	3
2.2	Discretization in Space	3
2.3	Pick a Subspace and Basis Functions	3
2.4	Derive Numerical Method	3
2.5	Recast Equation into Linear ODE	5
3	Justification of our Numerical Methods	7
3.1	Theoretical Stability	7
3.2	Error and Convergence	7
4	Simulation Experimentation	9
4.1	Implementation	9
5	Results and Conclusions	10

*This paper was created using L^AT_EX.

1 Introduction

The wave equation is a type of hyperbolic partial differential equation that is a well-posed initial value problem. The governing differential equation cannot be solved by simple algebra, so another method needs to be used to approximate the solution [1]. In this project, we will explore the solution to an wave equation using finite element methods. The most general form of the wave equation is:

$$\frac{\partial^2 u}{\partial t^2} + A \frac{\partial u}{\partial t} + ku = c^2 \frac{\partial^2 u}{\partial x^2} + F(x, t) \quad (1)$$

Where c is the wave speed, A is the damping coefficient, k is the external restoration factor and $F(x, t)$ is the arbitrary external forcing function. This equation in its various forms proves to be a crucial partial differential equation in engineering since it can be used to model seismic waves, deformation in elastic rods, motion of springs, or dynamics of acoustic waves. In our project we want to explore acoustic waves generated by a string. Let's assume a string of length of L is clamped at both ends. The vertical displacement of the string between $0 < x < L$ at any time $t > 0$ is given by displacement function $u(x, t)$. This satisfies the one-dimensional damped wave equation:

$$\frac{\partial^2 u}{\partial t^2} + A \frac{\partial u}{\partial t} = c^2 \frac{\partial^2 u}{\partial x^2} + F(x, t) \quad (2)$$

The damping coefficient A is non-zero, and it arises from the viscosity of the medium and density of the string [1]. Also, we are assuming k is zero. The initial and boundary conditions for this PDE are:

$$u(x, 0) = f(x) \quad x < 0 < L \quad (3)$$

$$\frac{\partial u}{\partial t}(x, 0) = g(x) \quad x < 0 < L \quad (4)$$

$$u(a, t) = 0 \quad t > 0 \quad (5)$$

$$u(b, t) = 0 \quad t > 0 \quad (6)$$

Where $a = 0$ and $b = L$ (length of string).

2 Model

2.1 Description of Problem

The wave equation represents an initial boundary value problem (IBVP) that must be approximated through both time and space. The method of lines will be employed to do this approximation.

2.2 Discretization in Space

The first step in the method of lines is to discretize the problem in space. This procedure results in a series of initial value problems (IVPs) that can be evaluated using any preferred finite element method. In other words, we will select a certain number of points at which to approximate the progression of the system over time. We will use this using a continuous point distribution as described in Eq. (7)

$$x_j = a + \frac{(b-a)(j-1)}{n} \quad j = 1, \dots, n+1 \quad (7)$$

2.3 Pick a Subspace and Basis Functions

The next step is to choose a subspace to approximate onto, and a set of basis functions to describe that subspace. We will be using the subspace \mathcal{V}_n^L which defines the collection of functions that are on the interval $[a, b]$, can be defined piecewise using the points x_1, \dots, x_{n+1} , and satisfy the boundary conditions we set for this problem. For the basis functions, we will use the "hat" functions shown in Eq. (8)

$$\phi_i = \begin{cases} \frac{1}{\Delta x}[x - a - (i-1)\Delta x] & \text{if } x \in [x_{i-1}, x_i] \\ \frac{-1}{\Delta x}[x - a - (i+1)\Delta x] & \text{if } x \in [x_i, x_{i+1}] \\ 0 & \text{else} \end{cases} \quad (8)$$

2.4 Derive Numerical Method

We can start by writing our approximation as a linear combination of the basis functions:

$$\hat{u}(x, t) = \sum_{i=2}^n u_i(t) \phi_i(x) \quad (9)$$

The numerical method itself can be determined by requiring that the least squares error between our approximation and the true solution be minimized (this approach to formulating a numerical method will generate a specific kind of method called a spectral method) [2]. The approximate solution that satisfies this relation is described as follows in Eq. (10):

$$(u - \hat{u}, \phi_j) = 0, \quad j = 2, \dots, n \quad (10)$$

We can apply this numerical method to our problem by combining equations 9 and 10:

$$\sum_{i=2}^n u_i(\phi_i, \phi_j)_E = (u, \phi_j)_E \quad j = 2, \dots, n \quad (11)$$

This is a good start, but now the equation is full of energy inner products. An appropriate EIP must be chosen before moving any further. After some trial and error, we decided to use the following one.

$$(p, q)_E = c^2 \int_a^b p'(x, t) q'(x, t) dx \quad (12)$$

Now we can use this energy inner product to expand the right hand side of Eq. (11).

$$(u, \phi_j)_E = c^2 \int_a^b u'(x, t) \phi_j'(x, t) dx \quad (13)$$

Using this energy inner product, we can expand and then simplify the right hand side of Eq. (11) according to the following steps:

1. Integrate by parts:

$$(u, \phi_j)_E = c^2 [u' \phi_j]_a^b - c^2 \int_a^b u'' \phi_j dx \quad (14)$$

2. Apply boundary conditions $\phi(a) = \phi(b) = 0$:

$$= -c^2 \int_a^b u'' \phi_j dx \quad (15)$$

3. Substitute in u'' , derived from rearranging the wave equation:

$$= c^2 \int_a^b \frac{1}{c^2} \left[\frac{\partial^2 u}{\partial t^2} + A \frac{\partial u}{\partial t} - F \right] \phi_j dx \quad (16)$$

4. Cancel out c^2 and expand integrals:

$$= - \int_a^b \frac{\partial^2 u}{\partial t^2} \phi_j dx - A \int_a^b \frac{\partial u}{\partial t} \phi_j dx + \int_a^b F \phi_j dx \quad (17)$$

5. Replace u with our approximation \hat{u} , which we can write in terms of the basis functions according to Eq. (9):

$$= - \int_a^b \frac{\partial^2}{\partial t^2} \left[\sum_{i=2}^n u_i \phi_i \right] \phi_j dx - A \int_a^b \frac{\partial}{\partial t} \left[\sum_{i=2}^n u_i \phi_i \right] \phi_j dx + \int_a^b F \phi_j dx \quad (18)$$

6. Pull summations out of integrals as they do not depend on x :

$$= - \sum_{i=2}^n \ddot{u}_i \int_a^b \phi_i \phi_j dx - A \sum_{i=2}^n \dot{u}_i \int_a^b \phi_i \phi_j dx + \int_a^b F \phi_j dx \quad (19)$$

Now we can plug Eq. (19) back into Eq. (11) and rearrange to arrive at the final expression for our system:

$$\sum_{i=2}^n \ddot{u}_i(\phi_i, \phi_j)_S = (F, \phi_j)_S - A \sum_{i=2}^n \dot{u}_i(\phi_i, \phi_j)_S - \sum_{i=2}^n u_i(\phi_i, \phi_j)_E \quad (20)$$

Our system can also be expressed in matrix form:

$$\begin{aligned} \mathbf{Q}\ddot{\mathbf{u}} &= \mathbf{F} - \mathbf{P}\mathbf{u} - A\mathbf{R}\dot{\mathbf{u}} \\ \ddot{\mathbf{u}} &= \mathbf{Q}^{-1}(\mathbf{F} - \mathbf{P}\mathbf{u} - A\mathbf{R}\dot{\mathbf{u}}) \end{aligned} \quad (21)$$

where:

$$\begin{aligned} \mathbf{Q} &= \begin{bmatrix} (\phi_2, \phi_2)_S & (\phi_3, \phi_2)_S & \dots & (\phi_{n-1}, \phi_2)_S & (\phi_n, \phi_2)_S \\ (\phi_2, \phi_3)_S & (\phi_3, \phi_3)_S & \dots & (\phi_{n-1}, \phi_3)_S & (\phi_n, \phi_3)_S \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ (\phi_2, \phi_{n-1})_S & (\phi_3, \phi_{n-1})_S & \dots & (\phi_{n-1}, \phi_{n-1})_S & (\phi_n, \phi_{n-1})_S \\ (\phi_2, \phi_n)_S & (\phi_3, \phi_n)_S & \dots & (\phi_{n-1}, \phi_n)_S & (\phi_n, \phi_n)_S \end{bmatrix} \\ \mathbf{F} &= \begin{bmatrix} (F, \phi_1)_S \\ (F, \phi_2)_S \\ \vdots \\ (F, \phi_{n-1})_S \\ (F, \phi_n)_S \end{bmatrix} \\ \mathbf{P} &= \begin{bmatrix} (\phi_2, \phi_2)_E & (\phi_3, \phi_2)_E & \dots & (\phi_{n-1}, \phi_2)_E & (\phi_n, \phi_2)_E \\ (\phi_2, \phi_3)_E & (\phi_3, \phi_3)_E & \dots & (\phi_{n-1}, \phi_3)_E & (\phi_n, \phi_3)_E \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ (\phi_2, \phi_{n-1})_E & (\phi_3, \phi_{n-1})_E & \dots & (\phi_{n-1}, \phi_{n-1})_E & (\phi_n, \phi_{n-1})_E \\ (\phi_2, \phi_n)_E & (\phi_3, \phi_n)_E & \dots & (\phi_{n-1}, \phi_n)_E & (\phi_n, \phi_n)_E \end{bmatrix} \\ \mathbf{R} &= \begin{bmatrix} \phi_2, \phi_2)_S & (\phi_3, \phi_2)_S & \dots & (\phi_{n-1}, \phi_2)_S & (\phi_n, \phi_2)_S \\ (\phi_2, \phi_3)_S & (\phi_3, \phi_3)_S & \dots & (\phi_{n-1}, \phi_3)_S & (\phi_n, \phi_3)_S \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ (\phi_2, \phi_{n-1})_S & (\phi_3, \phi_{n-1})_S & \dots & (\phi_{n-1}, \phi_{n-1})_S & (\phi_n, \phi_{n-1})_S \\ (\phi_2, \phi_n)_S & (\phi_3, \phi_n)_S & \dots & (\phi_{n-1}, \phi_n)_S & (\phi_n, \phi_n)_S \end{bmatrix} \end{aligned} \quad (22)$$

2.5 Recast Equation into Linear ODE

While the model described in Eqs. (21) and (22) is correct, it cannot be applied to any of the numerical methods we have learned in class. This is because, in order to utilize one of the methods we have studied, the model must take on the form

$$\dot{u} = f(u, t) \quad (23)$$

Eqs. (21) and (22) instead describe a model of the form

$$\ddot{u} = f(u, t) \quad (24)$$

To work around this issue, the equation can be "recast" in terms of some intermediary variable. We chose to use z . By setting $z = [u, \dot{u}]^T$, the whole system can be expressed in the form $\dot{z} = f(z, t)$, and we can use any of the methods studied in class to approximate z over time.

$$\begin{aligned} \begin{bmatrix} \dot{\mathbf{u}} \\ \ddot{\mathbf{u}} \end{bmatrix} &= \begin{bmatrix} 0 & \mathbf{I} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \dot{\mathbf{u}} \end{bmatrix} + \begin{bmatrix} 0 \\ \mathbf{Q}^{-1}(\mathbf{F} - \mathbf{P}\mathbf{u} - A\mathbf{R}\dot{\mathbf{u}}) \end{bmatrix} \\ \dot{z} &= \begin{bmatrix} 0 & \mathbf{I} \\ 0 & 0 \end{bmatrix} z + \begin{bmatrix} 0 \\ \mathbf{Q}^{-1}(\mathbf{F} - \mathbf{P}\mathbf{u} - A\mathbf{R}\dot{\mathbf{u}}) \end{bmatrix} \\ &= f(z, t) \end{aligned} \quad (25)$$

Our initial conditions must also be recast in terms z , so that they can be applied to the system.

$$z(x, 0) = \begin{bmatrix} f(x) \\ g(x) \end{bmatrix} \text{ for } x < 0 < L \quad (26)$$

3 Justification of our Numerical Methods

3.1 Theoretical Stability

We must find the absolute stability for the forward Euler method.

$$u_{k+1} = u_k + \Delta t \Delta u_k \quad (27)$$

$$= (I + \Delta t \Lambda) u_k \quad (28)$$

$$= (I + \Delta t \Lambda)(I + \Delta t \Lambda) u_{k-1} \quad (29)$$

$$= (I + \Delta t \Lambda)^{k+1} u_0 \quad (30)$$

We know Λ is a diagonal matrix. Using that information we can express the j^{th} entry in u_{k+1_j} as:

$$(u_{k+1})_j = (I + \Delta t \Lambda_j)^{k+1} (u_0)_j \quad (31)$$

Using Matlab, we were able to plot the stability region shown below.

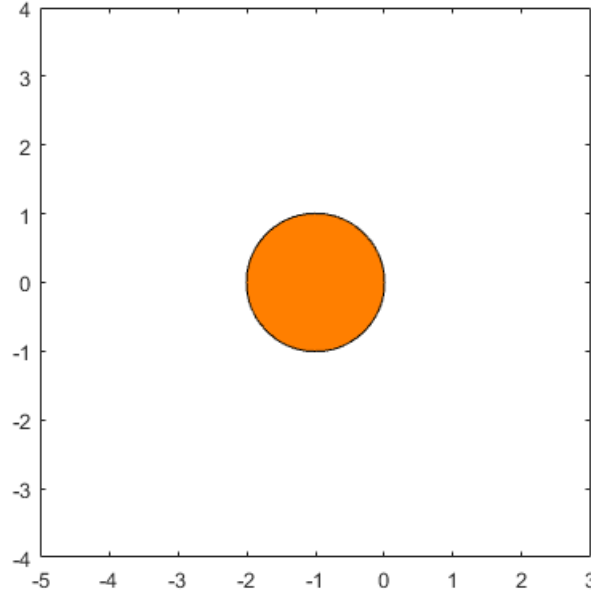


Figure 1: Stability plot

3.2 Error and Convergence

To ensure that the approximation obtained through executing the numerical method described in Section 2 is correct, the error between the approximate solution and exact solution should be taken. However, in this case for the wave equation for an acoustic wave, an exact solution cannot be provided. Because of this, a more clever way to determine the

correctness of the approximation needs to be implemented. As the number of points taken to approximate the solution increases, theoretically the approximation should converge towards the exact solution. Based off this assumption, a comparison between approximations for different numbers of points can be taken. The different numbers of points used in the calculation for error can be seen below in Eq. (32).

$$N_{vector} = [20 \ 40 \ 60 \ 80 \ 100] \quad (32)$$

The number of points used for approximation corresponds directly to the size of the variable dx used in the numerical method. This variable is the distance between points of approximation. The equation for dx can be seen below in Eq. (33).

$$dx = \frac{(b - a)}{(N - 1)} \quad (33)$$

From the equation, it can be seen that as N , the number of points, increases, dx , the space in between points of approximation will decrease.

The tell tale sign that the approximation method is sufficient is to show, through plotting the error, that the error plot is of rate $O(\Delta x^2)$. This means as the dx decreases, or N increases, the error will also decrease.

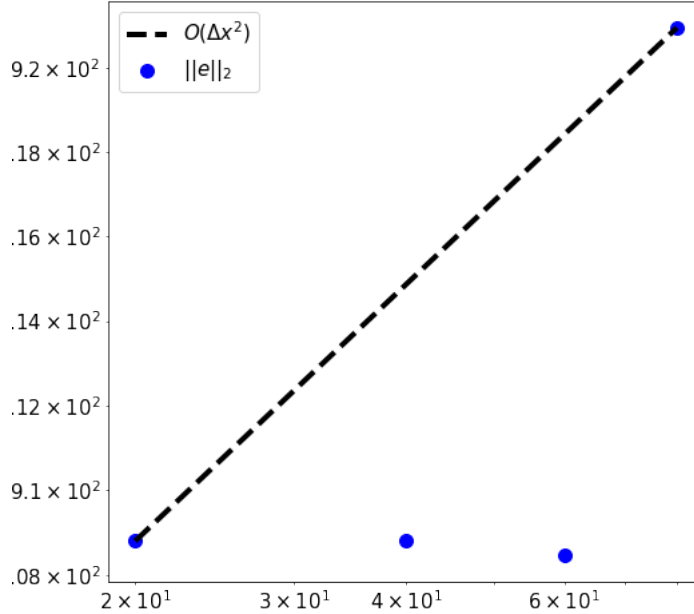


Figure 2: Error for different n-values

4 Simulation Experimentation

4.1 Implementation

To implement the numerical method, python was used to iterate through different numbers of points, and for each point the numerical method was implemented. From this implementation in python we can visualize the numerical method in a waterfall plot. This can be seen in a waterfall plot in Fig. 2.

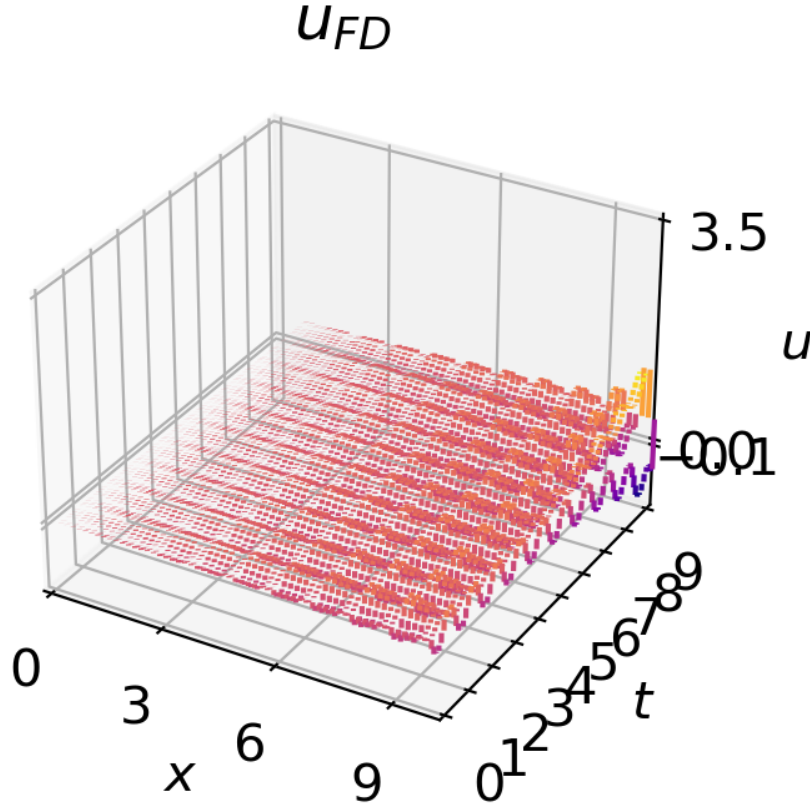


Figure 3: Visualization of Approximation u

5 Results and Conclusions

In this project we intended to explore our knowledge of the different numerical methods in a real life application. We chose the wave equation, which is a type of partial differential equation for our purpose. The wave equation has many forms depending on its application, be in seismic waves, acoustic waves, waves formed by disturbing a taut spring or and any other form. In this report the form of wave equation we have used is the simplest form for waves in a taut spring.

One problem we had on this project was approaching the numerical methods. In previous homeworks, we were given an exact solution to use as a function, but for this project we were not given an exact solution. One solution to this problem was to choose the Δx 's, then compare the points between the Δx 's. The same points between the Δx 's can be used as our 'exact' solution. Thank you Catherine Rogers for giving us that idea on Campuswire.

Another problem we had was our error plot, we first had to take into account the fact the euler-forward was no longer $u_{k+1} = u_k + \Delta t \Delta u_k$ it was now $z_{k+1} = z_k + \Delta t \Delta z_k$ the code was unfortunately difficult to debug because the code had an upward trend of at least 5 minutes to run. Although after several hours, we are confident the process of coding the error plot is correct.

One final problem we had was the derivation of the numerical methods, after getting the error plot, we have come to the conclusion that our numerical methods derivation was incorrect. One error that might have caused the inaccuracy in the error plot could have been the damping coefficient. In the future, we will look back to find our mistake to improve our derivation.

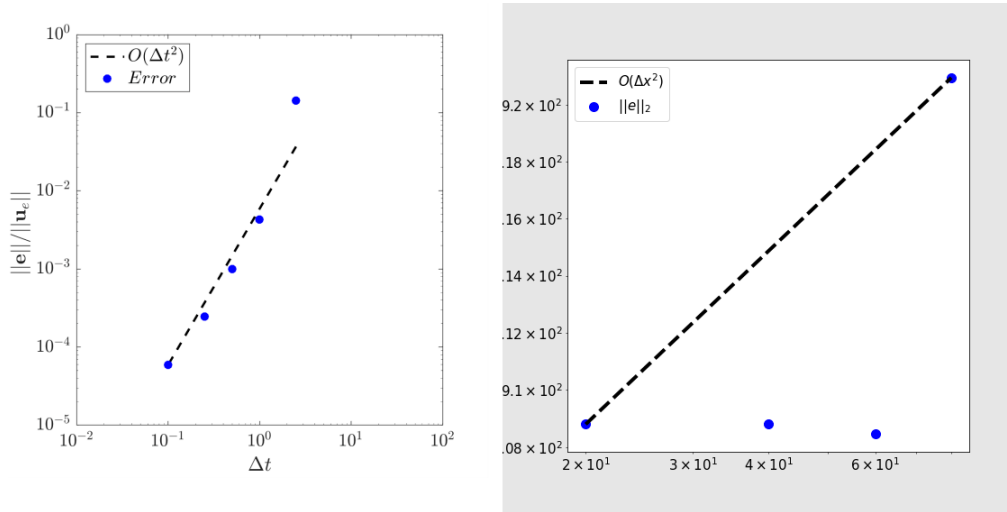


Figure 4: Ideal Error plot vs Derived Error plot

```

1 import numpy as np
2 from matplotlib import pyplot as plt
3 from matplotlib import axes as ax
4 from numpy.testing._private.utils import build_err_msg
5 import sympy
6 from numpy import linalg as LA
7 from matplotlib.collections import LineCollection
8 from mpl_toolkits.mplot3d import Axes3D
9
10 # pi = np.pi
11
12 # function for waterfall plot
13 def waterfall_plot(fig,ax,X,Y,Z):
14     # Set normalization to the same values for all plots
15     norm = plt.Normalize(Z.min().min(), Z.max().max())
16     # Check sizes to loop always over the smallest dimension
17     n,m = Z.shape
18     print('X shape:', X.shape)
19     print('Z shape:', Z)
20     if n > m:
21         X = X.T
22         Y = Y.T
23         Z = Z.T
24         m,n = n,m
25     for j in range(n):
26         # reshape the X,Z into pairs
27         points = np.array([X[j,:], Z[j,:]]).T.reshape(-1, 1, 2)
28         print(points)
29         segments = np.concatenate([points[:-1], points[1:]], axis=1)
30         lc = LineCollection(segments, cmap='plasma', norm=norm)
31         # Set the values used for colormapping
32         lc.set_array((Z[j,1:]+Z[j,:-1])/2)
33         lc.set_linewidth(1.5) # set linewidth a little larger to see properly the
34         colormap variation
35         line = ax.add_collection3d(lc, zs=(Y[j,1:]+Y[j,:-1])/2, zdir='y')
36
37 # ——— APPROX FOR N = 20 ——— #
38 # parameters
39 T = 10
40 dt = 1e-3
41 c = 0.4
42 n = 20
43 a = 0
44 b = 10
45 A = 60
46

```

```

47
48 dx = (b-a)/(n-1)
49 xj = np.linspace(a, b, n+1)
50 xjj = np.block([xj, xj])
51
52 # arrays of t- and x- values
53 tvect = np.arange(dt, T+0.1*dt, dt)
54 nt = len(tvect)
55 # Snapshots to save
56 nsnaps = 100
57 ind = max(1, np.floor(nt/nsnaps))
58 tsv = np.linspace(dt, T, nsnaps)
59 ntsv = len(tsv)
60 up = np.zeros((n - 1, ntsv))
61
62 # forcing function
63 def func(x, t):
64     return (np.cos(2*np.pi*t)*np.sin(2*np.pi*x))
65     # return 6*t*(x - 1) - 2*(t**3 + 1)
66
67 # build F
68 def F_build(nl, xl, dx, tl):
69     F = np.zeros((nl - 1, 1))
70
71     for jjj in range(nl - 1):
72         # define x from j-1 to j
73         x_lft = np.array([xl + jjj*dx, xl + (jjj+1)*dx, dx])
74
75         # define phi_j from j-1 to j
76         phij_lft = (1/dx)*(x_lft - xl - jjj*dx)
77
78         # use trapz to compute LHS integral
79         F[jjj] = np.trapz(func(x_lft, tl) * phij_lft, x_lft)
80
81         # define x from j to j+1
82         x_rgt = np.array([xl + (jjj+1)*dx, xl+(jjj+2)*dx,dx])
83
84         # define phi_j from j to j+1
85         phij_rgt = (-1/dx)*(x_rgt - xl - (jjj+2)*dx)
86
87         # use trapz to compute RHS integral
88         F[jjj] += np.trapz(func(x_rgt, tl) * phij_rgt, x_rgt)
89
90     return F
91
92 # P, Q, and R
93 P = (c**2)*(1/dx)*(-2*np.diag(np.ones(n-1), k=0) + np.diag(np.ones(n-2), k=-1) +np.

```

```

    diag(np.ones(n-2), k=1))
94 # print('P', P.shape)
95 Q = dx*((2/3)*np.diag(np.ones(n-1), k=0) + (1/6)*np.diag(np.ones(n-2), k=-1) + (1/6)
    *np.diag(np.ones(n-2), k=1))
96 R = dx*((2/3)*np.diag(np.ones(n-1), k=0) + (1/6)*np.diag(np.ones(n-2), k=-1) + (1/6)
    *np.diag(np.ones(n-2), k=1))
97
98 # get to steppin'
99 zk = np.zeros((2*(n-1), 1))
100 z = np.zeros((2*(n-1), ntsv))
101 # print(z.shape)
102 uk = np.zeros((n-1, 1))
103 up = np.zeros((n-1, ntsv))
104 uKeep1 = np.zeros((n-1, ntsv))
105 tk = 0
106 data = []
107 cnt = 0
108 for tt in range(len(tvect)):
109     tkp1 = tk + dt
110
111     # u and udot
112     u = zk[:int(zk.shape[0] / 2)]
113     udot = zk[int(zk.shape[0] / 2):]
114
115     F = F_build(n, a, dx, tk)
116
117     #build f
118     f1 = np.block([[np.zeros((n-1, n-1)), np.eye(n-1)], [np.zeros((n-1, n-1)), np.
    zeros((n-1, n-1))]]) @ zk
119     f2 = np.block([[np.zeros((n-1, 1))], [LA.inv(Q) @ (F - P@u - A*R@udot)]])
120     f = f1 + f2
121
122     # FEM
123     zkp1 = zk + dt*f
124     # print(zkp1)
125     ukp1 = zkp1[:int(zkp1.shape[0] / 2)]
126     # print(ukp1.shape)
127     for i in range(int(zkp1.shape[0] / 2)):
128         if i % 2 != 0 or i == 1:
129             ukp1[i - 1] = zkp1[i]
130
131     # print(ukp1)
132
133     # save data for plotting
134     data.append(zkp1)
135
136     # iterate

```

```

137     zk = zkp1
138     uk = ukp1
139     # print('zk',zk.shape)
140     tk = tkp1
141     # print('z', z[:, cnt].shape)
142     if tkp1 > tsv[cnt]:
143         z[:, [cnt]] = zk
144         up[:, [cnt]] = uk
145         cnt = cnt+1
146
147 uN20 = uk
148 uN20final = []
149 for i in range(uN20.shape[0]):
150     uN20final.append(uN20[i, 0])
151
152
153
154 # ----- APPROX FOR N = 40 ----- #
155 # parameters
156 T = 10
157 dt = 1e-3
158 c = 0.4
159 n = 40
160 a = 0
161 b = 10
162 A = 60
163
164 dx = (b-a)/(n-1)
165 xj = np.linspace(a, b, n+1)
166 xjj = np.block([xj, xj])
167
168 # arrays of t- and x- values
169 tvect = np.arange(dt, T+0.1*dt, dt)
170 nt = len(tvect)
171 # Snapshots to save
172 nsnaps = 100
173 ind = max(1, np.floor(nt/nsnaps))
174 tsv = np.linspace(dt, T, nsnaps)
175 ntsv = len(tsv)
176 up = np.zeros((n - 1, ntsv))
177
178 # P, Q, and R
179 P = (c**2)*(1/dx)*(-2*np.diag(np.ones(n-1), k=0) + np.diag(np.ones(n-2), k=-1) +np.
    diag(np.ones(n-2), k=1))
180 # print('P', P.shape)
181 Q = dx*((2/3)*np.diag(np.ones(n-1), k=0) + (1/6)*np.diag(np.ones(n-2), k=-1) + (1/6)
    *np.diag(np.ones(n-2), k=1))

```

```

182 R = dx*((2/3)*np.diag(np.ones(n-1), k=0) + (1/6)*np.diag(np.ones(n-2), k=-1) + (1/6)
    *np.diag(np.ones(n-2), k=1))
183
184 # get to steppin'
185 zk = np.zeros((2*(n-1), 1))
186 z = np.zeros((2*(n-1), ntsv))
187 # print(z.shape)
188 uk = np.zeros((n-1, 1))
189 up = np.zeros((n-1, ntsv))
190 uKeep1 = np.zeros((n-1, ntsv))
191 tk = 0
192 data = []
193 cnt = 0
194 for tt in range(len(tvect)):
195     tkp1 = tk + dt
196
197     # u and udot
198     u = zk[:int(zk.shape[0] / 2)]
199     udot = zk[int(zk.shape[0] / 2):]
200
201     F = F_build(n, a, dx, tk)
202
203     #build f
204     f1 = np.block([[np.zeros((n-1, n-1)), np.eye(n-1)], [np.zeros((n-1, n-1)), np.
        zeros((n-1, n-1))]]) @ zk
205     f2 = np.block([[np.zeros((n-1, 1))], [LA.inv(Q) @ (F - P@u - A*R@udot)]])
206     f = f1 + f2
207
208     # FEM
209     zkp1 = zk + dt*f
210     # print(zkp1)
211     ukp1 = zkp1[:int(zkp1.shape[0] / 2)]
212     # print(ukp1.shape)
213     for i in range(int(zkp1.shape[0] / 2)):
214         if i % 2 != 0 or i == 1:
215             ukp1[i - 1] = zkp1[i]
216
217     # print(ukp1)
218
219     # save data for plotting
220     data.append(zkp1)
221
222     # iterate
223     zk = zkp1
224     uk = ukp1
225     # print('zk',zk.shape)
226     tk = tkp1

```

```

227     # print('z', z[:, cnt].shape)
228     if tkp1 > tsv[cnt]:
229         z[:, [cnt]] = zk
230         up[:, [cnt]] = uk
231         cnt = cnt+1
232 uN40 = uk
233 uN40final = []
234 for i in range(uN40.shape[0]):
235     uN40final.append(uN40[i, 0])
236
237
238 # ----- APPROX FOR N = 60 ----- #
239 # parameters
240 T = 10
241 dt = 1e-3
242 c = 0.4
243 n = 60
244 a = 0
245 b = 10
246 A = 60
247
248 dx = (b-a)/(n-1)
249 xj = np.linspace(a, b, n+1)
250 xjj = np.block([xj, xj])
251
252 # arrays of t- and x- values
253 tvect = np.arange(dt, T+0.1*dt, dt)
254 nt = len(tvect)
255 # Snapshots to save
256 nsnaps = 100
257 ind = max(1, np.floor(nt/nsnaps))
258 tsv = np.linspace(dt, T, nsnaps)
259 ntsv = len(tsv)
260 up = np.zeros((n - 1, ntsv))
261
262 # P, Q, and R
263 P = (c**2)*(1/dx)*(-2*np.diag(np.ones(n-1), k=0) + np.diag(np.ones(n-2), k=-1) +np.
    diag(np.ones(n-2), k=1))
264 # print('P', P.shape)
265 Q = dx*((2/3)*np.diag(np.ones(n-1), k=0) + (1/6)*np.diag(np.ones(n-2), k=-1) + (1/6)
    *np.diag(np.ones(n-2), k=1))
266 R = dx*((2/3)*np.diag(np.ones(n-1), k=0) + (1/6)*np.diag(np.ones(n-2), k=-1) + (1/6)
    *np.diag(np.ones(n-2), k=1))
267
268 # get to steppin'
269 zk = np.zeros((2*(n-1), 1))
270 z = np.zeros((2*(n-1), ntsv))

```



```

271 # print(z.shape)
272 uk = np.zeros((n-1, 1))
273 up = np.zeros((n-1, ntsv))
274 uKeep1 = np.zeros((n-1, ntsv))
275 tk = 0
276 data = []
277 cnt = 0
278 for tt in range(len(tvect)):
279     tkp1 = tk + dt
280
281     # u and udot
282     u = zk[:int(zk.shape[0] / 2)]
283     udot = zk[int(zk.shape[0] / 2):]
284
285     F = F_build(n, a, dx, tk)
286
287     #build f
288     f1 = np.block([[np.zeros((n-1, n-1)), np.eye(n-1)], [np.zeros((n-1, n-1)), np.
zeros((n-1, n-1))]]) @ zk
289     f2 = np.block([[np.zeros((n-1, 1))], [LA.inv(Q) @ (F - P@u - A*R@udot)]])
290     f = f1 + f2
291
292     # FEM
293     zkp1 = zk + dt*f
294     # print(zkp1)
295     ukp1 = zkp1[:int(zkp1.shape[0] / 2)]
296     # print(ukp1.shape)
297     for i in range(int(zkp1.shape[0] / 2)):
298         if i % 2 != 0 or i == 1:
299             ukp1[i - 1] = zkp1[i]
300
301     # print(ukp1)
302
303     # save data for plotting
304     data.append(zkp1)
305
306     # iterate
307     zk = zkp1
308     uk = ukp1
309     # print('zk',zk.shape)
310     tk = tkp1
311     # print('z', z[:, cnt].shape)
312     if tkp1 > tsv[cnt]:
313         z[:, [cnt]] = zk
314         up[:, [cnt]] = uk
315         cnt = cnt+1
316 uN60 = uk

```

```

317 uN60final = []
318 for i in range(uN60.shape[0]):
319     uN60final.append(uN60[i, 0])
320
321
322
323
324
325 # ----- APPROX FOR N = 80 ----- #
326 # parameters
327 T = 10
328 dt = 1e-3
329 c = 0.4
330 n = 80
331 a = 0
332 b = 10
333 A = 60
334
335 dx = (b-a)/(n-1)
336 xj = np.linspace(a, b, n+1)
337 xjj = np.block([xj, xj])
338
339 # arrays of t- and x- values
340 tvect = np.arange(dt, T+0.1*dt, dt)
341 nt = len(tvect)
342 # Snapshots to save
343 nsnaps = 100
344 ind = max(1, np.floor(nt/nsnaps))
345 tsv = np.linspace(dt, T, nsnaps)
346 ntsv = len(tsv)
347 up = np.zeros((n - 1, ntsv))
348
349 # P, Q, and R
350 P = (c**2)*(1/dx)*(-2*np.diag(np.ones(n-1), k=0) + np.diag(np.ones(n-2), k=-1) + np.
    diag(np.ones(n-2), k=1))
351 # print('P', P.shape)
352 Q = dx*((2/3)*np.diag(np.ones(n-1), k=0) + (1/6)*np.diag(np.ones(n-2), k=-1) + (1/6)
    *np.diag(np.ones(n-2), k=1))
353 R = dx*((2/3)*np.diag(np.ones(n-1), k=0) + (1/6)*np.diag(np.ones(n-2), k=-1) + (1/6)
    *np.diag(np.ones(n-2), k=1))
354
355 # get to steppin'
356 zk = np.zeros((2*(n-1), 1))
357 z = np.zeros((2*(n-1), ntsv))
358 # print(z.shape)
359 uk = np.zeros((n-1, 1))
360 up = np.zeros((n-1, ntsv))

```

```

361 uKeep1 = np.zeros((n-1, ntsv))
362 tk = 0
363 data = []
364 cnt = 0
365 for tt in range(len(tvect)):
366     tkp1 = tk + dt
367
368     # u and udot
369     u = zk[:int(zk.shape[0] / 2)]
370     udot = zk[int(zk.shape[0] / 2):]
371
372     F = F_build(n, a, dx, tk)
373
374     #build f
375     f1 = np.block([[np.zeros((n-1, n-1)), np.eye(n-1)], [np.zeros((n-1, n-1)), np.
zeros((n-1, n-1))]]) @ zk
376     f2 = np.block([[np.zeros((n-1, 1))], [LA.inv(Q) @ (F - P@u - A*R@udot)]])
377     f = f1 + f2
378
379     # FEM
380     zkp1 = zk + dt*f
381     # print(zkp1)
382     ukp1 = zkp1[:int(zkp1.shape[0] / 2)]
383     # print(ukp1.shape)
384     for i in range(int(zkp1.shape[0] / 2)):
385         if i % 2 != 0 or i == 1:
386             ukp1[i - 1] = zkp1[i]
387
388     # print(ukp1)
389
390     # save data for plotting
391     data.append(zkp1)
392
393     # iterate
394     zk = zkp1
395     uk = ukp1
396     # print('zk', zk.shape)
397     tk = tkp1
398     # print('z', z[:, cnt].shape)
399     if tkp1 > tsv[cnt]:
400         z[:, [cnt]] = zk
401         up[:, [cnt]] = uk
402         cnt = cnt+1
403 uN80 = uk
404 uN80final = []
405 for i in range(uN80.shape[0]):
406     uN80final.append(uN80[i, 0])

```

```

407
408
409
410
411
412 # ——— APPROX FOR N = 100 ——— #
413 # parameters
414 T = 10
415 dt = 1e-3
416 c = 0.4
417 n = 100
418 a = 0
419 b = 10
420 A = 60
421
422 dx = (b-a)/(n-1)
423 xj = np.linspace(a, b, n+1)
424 xjj = np.block([xj, xj])
425
426 # arrays of t- and x- values
427 tvect = np.arange(dt, T+0.1*dt, dt)
428 nt = len(tvect)
429 # Snapshots to save
430 nsnaps = 100
431 ind = max(1, np.floor(nt/nsnaps))
432 tsv = np.linspace(dt, T, nsnaps)
433 ntsv = len(tsv)
434 up = np.zeros((n-1, ntsv))
435
436 # P, Q, and R
437 P = (c**2)*(1/dx)*(-2*np.diag(np.ones(n-1), k=0) + np.diag(np.ones(n-2), k=-1) + np.
    diag(np.ones(n-2), k=1))
438 # print('P', P.shape)
439 Q = dx*((2/3)*np.diag(np.ones(n-1), k=0) + (1/6)*np.diag(np.ones(n-2), k=-1) + (1/6)
    *np.diag(np.ones(n-2), k=1))
440 R = dx*((2/3)*np.diag(np.ones(n-1), k=0) + (1/6)*np.diag(np.ones(n-2), k=-1) + (1/6)
    *np.diag(np.ones(n-2), k=1))
441
442 # get to steppin'
443 zk = np.zeros((2*(n-1), 1))
444 z = np.zeros((2*(n-1), ntsv))
445 # print(z.shape)
446 uk = np.zeros((n-1, 1))
447 up = np.zeros((n-1, ntsv))
448 uKeep1 = np.zeros((n-1, ntsv))
449 tk = 0
450 data = []

```

```

451 cnt = 0
452 for tt in range(len(tvect)):
453     tkp1 = tk + dt
454
455     # u and udot
456     u = zk[:int(zk.shape[0] / 2)]
457     udot = zk[int(zk.shape[0] / 2):]
458
459     F = F_build(n, a, dx, tk)
460
461     #build f
462     f1 = np.block([[np.zeros((n-1, n-1)), np.eye(n-1)], [np.zeros((n-1, n-1)), np.
zeros((n-1, n-1))]]) @ zk
463     f2 = np.block([[np.zeros((n-1, 1))], [LA.inv(Q) @ (F - P@u - A*R@udot)]])
464     f = f1 + f2
465
466     # FEM
467     zkp1 = zk + dt*f
468     # print(zkp1)
469     ukp1 = zkp1[:int(zkp1.shape[0] / 2)]
470     # print(ukp1.shape)
471     for i in range(int(zkp1.shape[0] / 2)):
472         if i % 2 != 0 or i == 1:
473             ukp1[i - 1] = zkp1[i]
474
475     # print(ukp1)
476
477     # save data for plotting
478     data.append(zkp1)
479
480     # iterate
481     zk = zkp1
482     uk = ukp1
483     # print('zk',zk.shape)
484     tk = tkp1
485     # print('z', z[:, cnt].shape)
486     if tkp1 > tsv[cnt]:
487         z[:, [cnt]] = zk
488         up[:, [cnt]] = uk
489         cnt = cnt+1
490 uN100 = uk
491 uN100final = []
492 for i in range(uN100.shape[0]):
493     uN100final.append(uN100[i, 0])
494
495
496 # ——— PLOTTING ERROR ——— #

```

```

497 comp20 = np.zeros(len(uN20final))
498 comp40 = np.zeros(len(uN20final))
499 comp60 = np.zeros(len(uN20final))
500 comp80 = np.zeros(len(uN20final))
501 comp100 = np.zeros(len(uN20final))
502 for j in range(len(uN20final)):
503     comp20[j] = uN20[j]
504     comp40[j] = uN40[2*j]
505     comp60[j] = uN60[3*j]
506     comp80[j] = uN80[4*j]
507     comp100[j] = uN100[5*j]
508
509 err20 = LA.norm(comp20 - comp100)
510 err40 = LA.norm(comp40 - comp100)
511 err60 = LA.norm(comp60 - comp100)
512 err80 = LA.norm(comp80 - comp100)
513
514 err = np.array([err20, err40, err60, err80])
515
516 print(f'Error 20 = {err20}')
517 print(f'Error 40 = {err40}')
518 print(f'Error 60 = {err60}')
519 print(f'Error 80 = {err80}')
520
521 nvect = np.array([20, 40, 60, 80])
522 errx = np.array([nvect[0], nvect[-1]])
523 erry = np.array([err[0], err[-1]])
524
525 plt.figure(figsize=(8, 8))
526 plt.loglog(errx, erry, linewidth=4, linestyle='—', color='k', label = '$0(\Delta x$
    ^2)$')
527 plt.scatter(nvect, err, s=100, color='b', label = '$||e||_{-2}$')
528 plt.legend()
529 plt.savefig('error_plot')

```

References

- [1] Kisabo, A., and Ibrahim, J., “Solving Wave Equation using Finite Element Method,” <https://www.researchgate.net/publication/350192695>, 2021.
- [2] Goza, A., “AE 370 - Spring 2021 course page,” <https://sites.google.com/illinois.edu/ae-370/course-contenth.ias2p88mgxfr>, 2021.