Solving the Wave Equation (IBVP) using Finite Element Method

Rohen Agarwal, Charlie Ray, Dennis McCann, Justin Chung

May 12, 2021

tocloft

Contents

*This paper was created using LaTeX.

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:

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

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:

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

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:

$$u(x,0) = f(x)\ x < 0 < L \tag{3}$$

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

$$u(a,t) = 0 \text{ t} > 0 \tag{5}$$

$$u(b,t) = 0 \text{ t} > 0 \tag{6}$$

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)

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

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 $x_1, ..., x_{n+1}$, and satisfy the boundary conditions we set for this problem. For the basis functions, we will use the "hat" functions shown in Eq. (8)

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

2.4 Derive Numerical Method

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

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

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):

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

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

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

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.

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

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

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

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:
$$(u, \phi_j)_E = c^2 \left[u'\phi_j\right]_a^b - c^2 \int_a^b u''\phi_j\, dx \tag{14}$$

2. Apply boundary conditions $\phi(a) = \phi(b) = 0$:
$$= -c^2 \int_a^b u''\phi_j\, dx \tag{15}$$

3. Substitute in u'', derived from rearranging the wave equation:
$$= c^2 \int_a^b \frac{1}{c^2}\left[\frac{\partial^2 u}{\partial t^2} + A\frac{\partial u}{\partial t} - F\right]\phi_j\, dx \tag{16}$$

4. Cancel out c^2 and expand integrals:
$$= -\int_a^b \frac{\partial^2 u}{\partial t^2}\phi_j\, dx - A\int_a^b \frac{\partial u}{\partial t}\phi_j\, dx + \int_a^b F\phi_j\, dx \tag{17}$$

5. Replace u with our approximation $\hat{u}$, which we can write in terms of the basis functions according to Eq. (9):
$$= -\int_a^b \frac{\partial^2}{\partial t^2}\left[\sum_{i=2}^{n} u_i\phi_i\right]\phi_j\, dx - A\int_a^b \frac{\partial}{\partial t}\left[\sum_{i=2}^{n} u_i\phi_i\right]\phi_j\, dx + \int_a^b F\phi_j\, dx \tag{18}$$

6. Pull summations out of integrals as they do not depend on x:
$$= -\sum_{i=2}^{n} \ddot{u}_i \int_a^b \phi_i\phi_j\, dx - A\sum_{i=2}^{n} \dot{u}_i \int_a^b \phi_i\phi_j\, dx + \int_a^b F\phi_j\, dx \tag{19}$$

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

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

Our system can also be expressed in matrix form:

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

where:

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

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

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

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

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

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.

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

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

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

3 Justification of our Numerical Methods

3.1 Theoretical Stability

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

$$
\begin{aligned}
u_{k+1} &= u_k + \Delta t \Delta u_k &(27)\\
&= (I + \Delta t\Lambda) u_k &(28)\\
&= (I + \Delta t\Lambda)(I + \Delta t\Lambda) u_{k-1} &(29)\\
&= (I + \Delta t\Lambda)^{k+1} u_0 &(30)
\end{aligned}
$$

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

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

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

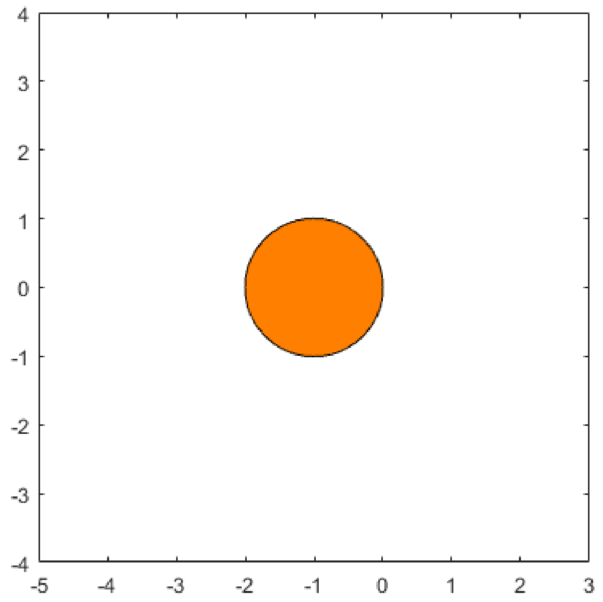

Figure 1: Stability plot

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).

$$N_{vector} = \begin{bmatrix}20\ 40\ 60\ 80\ 100\end{bmatrix} \tag{32}$$

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).

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

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.

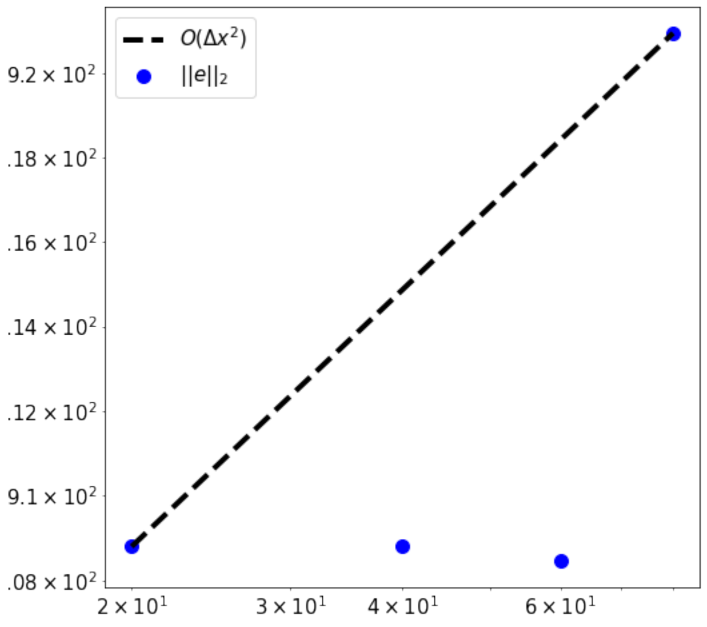

Figure 2: Error for different n-values

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.

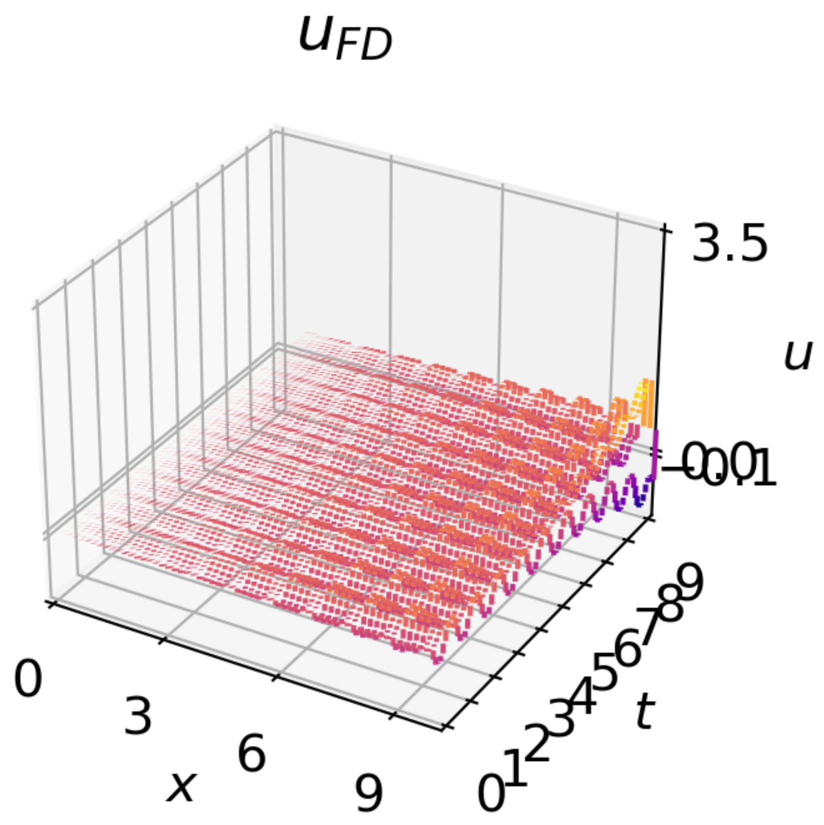

Figure 3: Visualization of Approximation u

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 dx's, then compare the points between the dx's. The same points between the dx'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.

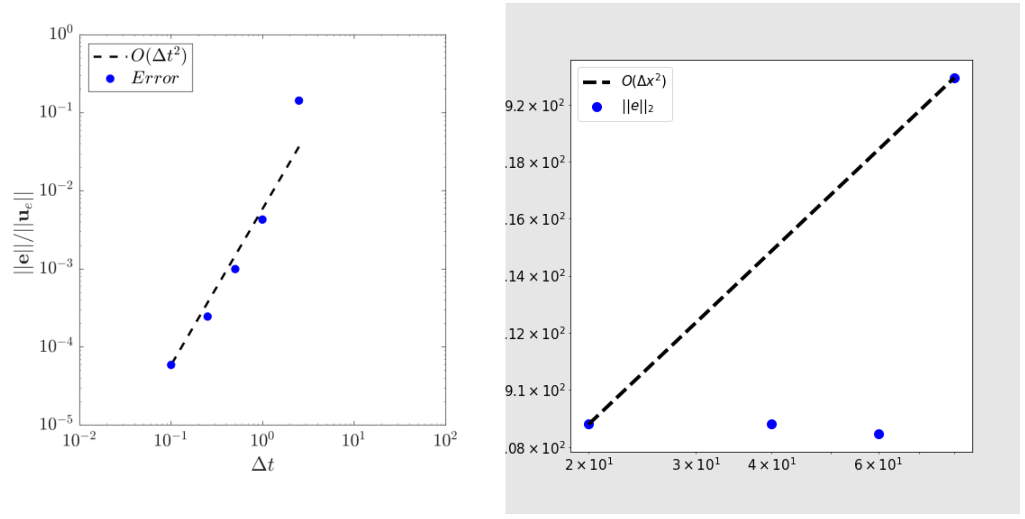

Figure 4: Ideal Error plot vs Derived Error plot

```python
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import axes as ax
from numpy.testing._private.utils import build_err_msg
import sympy
from numpy import linalg as LA
from matplotlib.collections import LineCollection
from mpl_toolkits.mplot3d import Axes3D

# pi = np.pi

# function for waterfall plot
def waterfall_plot(fig,ax,X,Y,Z):
    # Set normalization to the same values for all plots
    norm = plt.Normalize(Z.min().min(), Z.max().max())
    # Check sizes to loop always over the smallest dimension
    n,m = Z.shape
    print('X shape:', X.shape)
    print('Z shape:', Z)
    if n > m:
        X = X.T
        Y = Y.T
        Z = Z.T
        m,n = n,m
    for j in range(n):
        # reshape the X,Z into pairs
        points = np.array([X[j,:], Z[j,:]]).T.reshape(-1, 1, 2)
        print(points)
        segments = np.concatenate([points[:-1], points[1:]], axis=1)
        lc = LineCollection(segments, cmap='plasma', norm=norm)
        # Set the values used for colormapping
        lc.set_array((Z[j,1:]+Z[j,:-1])/2)
        lc.set_linewidth(1.5) # set linewidth a little larger to see properly the colormap variation
        line = ax.add_collection3d(lc, zs=(Y[j,1:]+Y[j,:-1])/2, zdir='y')


# ---------- APPROX FOR N = 20 ---------- #
# parameters
T = 10
dt = 1e-3
c = 0.4
n = 20
a = 0
b = 10
A = 60

```

```

dx = (b-a)/(n-1)
xj = np.linspace(a, b, n+1)
xjj = np.block([xj, xj])

# arrays of t- and x- values
tvect = np.arange(dt, T+0.1*dt, dt)
nt = len(tvect)
# Snapshots to save
nsnaps = 100
ind = max(1, np.floor(nt/nsnaps))
tsv = np.linspace(dt, T, nsnaps)
ntsv = len(tsv)
up = np.zeros((n - 1, ntsv))

# forcing function
def func(x, t):
    return (np.cos(2*np.pi*t)*np.sin(2*np.pi*x))
    # return 6*t*(x - 1) - 2*(t**3 + 1)

# build F
def F_build(nl, xl, dx, tl):
    F = np.zeros((nl - 1, 1))

    for jjj in range(nl - 1):
        # define x from j-1 to j
        x_lft = np.array([xl + jjj*dx, xl + (jjj+1)*dx, dx])

        # define phi_j from j-1 to j
        phij_lft = (1/dx)*(x_lft - xl - jjj*dx)

        # use trapz to compute LHS integral
        F[jjj] = np.trapz(func(x_lft, tl) * phij_lft, x_lft)

        # define x from j to j+1
        x_rgt = np.array([xl + (jjj+1)*dx, xl+(jjj+2)*dx,dx])

        # define phi_j from j to j+1
        phij_rgt = (-1/dx)*(x_rgt - xl - (jjj+2)*dx)

        # use trapz to compute RHS integral
        F[jjj] += np.trapz(func(x_rgt, tl) * phij_rgt, x_rgt)

    return F

# P, Q, and R
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))
# print('P', P.shape)
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))
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))

# get to steppin'
zk = np.zeros((2*(n-1), 1))
z = np.zeros((2*(n-1), ntsv))
# print(z.shape)
uk = np.zeros((n-1, 1))
up = np.zeros((n-1, ntsv))
uKeep1 = np.zeros((n-1, ntsv))
tk = 0
data = []
cnt = 0
for tt in range(len(tvect)):
    tkp1 = tk + dt

    # u and udot
    u = zk[:int(zk.shape[0] / 2)]
    udot = zk[int(zk.shape[0] / 2):]

    F = F_build(n, a, dx, tk)

    #build f
    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
    f2 = np.block([[np.zeros((n-1, 1))], [LA.inv(Q) @ (F - P@u - A*R@udot)]])
    f = f1 + f2

    # FEM
    zkp1 = zk + dt*f
    # print(zkp1)
    ukp1 = zkp1[:int(zkp1.shape[0] / 2)]
    # print(ukp1.shape)
    for i in range(int(zkp1.shape[0] / 2)):
        if i % 2 != 0 or i == 1:
            ukp1[i - 1] = zkp1[i]

    # print(ukp1)

    # save data for plotting
    data.append(zkp1)

    # iterate
    zk = zkp1
    uk = ukp1
    # print('zk',zk.shape)
    tk = tkp1
    # print('z', z[:, cnt].shape)
    if tkp1 > tsv[cnt]:
        z[:, [cnt]] = zk
        up[:, [cnt]] = uk
        cnt = cnt+1

uN20 = uk
uN20final = []
for i in range(uN20.shape[0]):
    uN20final.append(uN20[i, 0])



# ------ APPROX FOR N = 40 ------ #
# parameters
T = 10
dt = 1e-3
c = 0.4
n = 40
a = 0
b = 10
A = 60

dx = (b-a)/(n-1)
xj = np.linspace(a, b, n+1)
xjj = np.block([xj, xj])

# arrays of t- and x- values
tvect = np.arange(dt, T+0.1*dt, dt)
nt = len(tvect)
# Snapshots to save
nsnaps = 100
ind = max(1, np.floor(nt/nsnaps))
tsv = np.linspace(dt, T, nsnaps)
ntsv = len(tsv)
up = np.zeros((n - 1, ntsv))

# P, Q, and R
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))
# print('P', P.shape)
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))
```

```
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))

# get to steppin'
zk = np.zeros((2*(n-1), 1))
z = np.zeros((2*(n-1), ntsv))
# print(z.shape)
uk = np.zeros((n-1, 1))
up = np.zeros((n-1, ntsv))
uKeep1 = np.zeros((n-1, ntsv))
tk = 0
data = []
cnt = 0
for tt in range(len(tvect)):
    tkp1 = tk + dt

    # u and udot
    u = zk[:int(zk.shape[0] / 2)]
    udot = zk[int(zk.shape[0] / 2):]

    F = F_build(n, a, dx, tk)

    #build f
    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
    f2 = np.block([[np.zeros((n-1, 1))], [LA.inv(Q) @ (F - P@u - A*R@udot)]])
    f = f1 + f2

    # FEM
    zkp1 = zk + dt*f
    # print(zkp1)
    ukp1 = zkp1[:int(zkp1.shape[0] / 2)]
    # print(ukp1.shape)
    for i in range(int(zkp1.shape[0] / 2)):
        if i % 2 != 0 or i == 1:
            ukp1[i - 1] = zkp1[i]

    # print(ukp1)

    # save data for plotting
    data.append(zkp1)

    # iterate
    zk = zkp1
    uk = ukp1
    # print('zk',zk.shape)
    tk = tkp1
```

```python
    # print('z', z[:, cnt].shape)
    if tkp1 > tsv[cnt]:
        z[:, [cnt]] = zk
        up[:, [cnt]] = uk
        cnt = cnt+1
uN40 = uk
uN40final = []
for i in range(uN40.shape[0]):
    uN40final.append(uN40[i, 0])


# ----- APPROX FOR N = 60 ------ #
# parameters
T = 10
dt = 1e-3
c = 0.4
n = 60
a = 0
b = 10
A = 60

dx = (b-a)/(n-1)
xj = np.linspace(a, b, n+1)
xjj = np.block([xj, xj])

# arrays of t- and x- values
tvect = np.arange(dt, T+0.1*dt, dt)
nt = len(tvect)
# Snapshots to save
nsnaps = 100
ind = max(1, np.floor(nt/nsnaps))
tsv = np.linspace(dt, T, nsnaps)
ntsv = len(tsv)
up = np.zeros((n - 1, ntsv))

# P, Q, and R
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))
# print('P', P.shape)
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))
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))

# get to steppin'
zk = np.zeros((2*(n-1), 1))
z = np.zeros((2*(n-1), ntsv))
```

```python
# print(z.shape)
uk = np.zeros((n-1, 1))
up = np.zeros((n-1, ntsv))
uKeep1 = np.zeros((n-1, ntsv))
tk = 0
data = []
cnt = 0
for tt in range(len(tvect)):
    tkp1 = tk + dt

    # u and udot
    u = zk[:int(zk.shape[0] / 2)]
    udot = zk[int(zk.shape[0] / 2):]

    F = F_build(n, a, dx, tk)

    #build f
    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
    f2 = np.block([[np.zeros((n-1, 1))], [LA.inv(Q) @ (F - P@u - A*R@udot)]])
    f = f1 + f2

    # FEM
    zkp1 = zk + dt*f
    # print(zkp1)
    ukp1 = zkp1[:int(zkp1.shape[0] / 2)]
    # print(ukp1.shape)
    for i in range(int(zkp1.shape[0] / 2)):
        if i % 2 != 0 or i == 1:
            ukp1[i - 1] = zkp1[i]

    # print(ukp1)

    # save data for plotting
    data.append(zkp1)

    # iterate
    zk = zkp1
    uk = ukp1
    # print('zk',zk.shape)
    tk = tkp1
    # print('z', z[:, cnt].shape)
    if tkp1 > tsv[cnt]:
        z[:, [cnt]] = zk
        up[:, [cnt]] = uk
        cnt = cnt+1
uN60 = uk
```

```python
uN60final = []
for i in range(uN60.shape[0]):
    uN60final.append(uN60[i, 0])





# ------ APPROX FOR N = 80 ------ #
# parameters
T = 10
dt = 1e-3
c = 0.4
n = 80
a = 0
b = 10
A = 60

dx = (b-a)/(n-1)
xj = np.linspace(a, b, n+1)
xjj = np.block([xj, xj])

# arrays of t- and x- values
tvect = np.arange(dt, T+0.1*dt, dt)
nt = len(tvect)
# Snapshots to save
nsnaps = 100
ind = max(1, np.floor(nt/nsnaps))
tsv = np.linspace(dt, T, nsnaps)
ntsv = len(tsv)
up = np.zeros((n - 1, ntsv))

# P, Q, and R
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))
# print('P', P.shape)
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))
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))

# get to steppin'
zk = np.zeros((2*(n-1), 1))
z = np.zeros((2*(n-1), ntsv))
# print(z.shape)
uk = np.zeros((n-1, 1))
up = np.zeros((n-1, ntsv))
```

```
uKeep1 = np.zeros((n-1, ntsv))
tk = 0
data = []
cnt = 0
for tt in range(len(tvect)):
    tkp1 = tk + dt

    # u and udot
    u = zk[:int(zk.shape[0] / 2)]
    udot = zk[int(zk.shape[0] / 2):]

    F = F_build(n, a, dx, tk)

    #build f
    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
    f2 = np.block([[np.zeros((n-1, 1))], [LA.inv(Q) @ (F - P@u - A*R@udot)]])
    f = f1 + f2

    # FEM
    zkp1 = zk + dt*f
    # print(zkp1)
    ukp1 = zkp1[:int(zkp1.shape[0] / 2)]
    # print(ukp1.shape)
    for i in range(int(zkp1.shape[0] / 2)):
        if i % 2 != 0 or i == 1:
            ukp1[i - 1] = zkp1[i]

    # print(ukp1)

    # save data for plotting
    data.append(zkp1)

    # iterate
    zk = zkp1
    uk = ukp1
    # print('zk',zk.shape)
    tk = tkp1
    # print('z', z[:, cnt].shape)
    if tkp1 > tsv[cnt]:
        z[:, [cnt]] = zk
        up[:, [cnt]] = uk
        cnt = cnt+1
uN80 = uk
uN80final = []
for i in range(uN80.shape[0]):
    uN80final.append(uN80[i, 0])
```

```
# ——— APPROX FOR N = 100 ———— #
# parameters
T = 10
dt = 1e-3
c = 0.4
n = 100
a = 0
b = 10
A = 60

dx = (b-a)/(n-1)
xj = np.linspace(a, b, n+1)
xjj = np.block([xj, xj])

# arrays of t- and x- values
tvect = np.arange(dt, T+0.1*dt, dt)
nt = len(tvect)
# Snapshots to save
nsnaps = 100
ind = max(1, np.floor(nt/nsnaps))
tsv = np.linspace(dt, T, nsnaps)
ntsv = len(tsv)
up = np.zeros((n - 1, ntsv))

# P, Q, and R
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))
# print('P', P.shape)
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))
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))

# get to steppin'
zk = np.zeros((2*(n-1), 1))
z = np.zeros((2*(n-1), ntsv))
# print(z.shape)
uk = np.zeros((n-1, 1))
up = np.zeros((n-1, ntsv))
uKeep1 = np.zeros((n-1, ntsv))
tk = 0
data = []
```

```
cnt = 0
for tt in range(len(tvect)):
    tkp1 = tk + dt

    # u and udot
    u = zk[:int(zk.shape[0] / 2)]
    udot = zk[int(zk.shape[0] / 2):]

    F = F_build(n, a, dx, tk)

    #build f
    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
    f2 = np.block([[np.zeros((n-1, 1))], [LA.inv(Q) @ (F - P@u - A*R@udot)]])
    f = f1 + f2

    # FEM
    zkp1 = zk + dt*f
    # print(zkp1)
    ukp1 = zkp1[:int(zkp1.shape[0] / 2)]
    # print(ukp1.shape)
    for i in range(int(zkp1.shape[0] / 2)):
        if i % 2 != 0 or i == 1:
            ukp1[i - 1] = zkp1[i]

    # print(ukp1)

    # save data for plotting
    data.append(zkp1)

    # iterate
    zk = zkp1
    uk = ukp1
    # print('zk',zk.shape)
    tk = tkp1
    # print('z', z[:, cnt].shape)
    if tkp1 > tsv[cnt]:
        z[:, [cnt]] = zk
        up[:, [cnt]] = uk
        cnt = cnt+1
uN100 = uk
uN100final = []
for i in range(uN100.shape[0]):
    uN100final.append(uN100[i, 0])


# ------ PLOTTING ERROR ------ #
```

```python
comp20 = np.zeros(len(uN20final))
comp40 = np.zeros(len(uN20final))
comp60 = np.zeros(len(uN20final))
comp80 = np.zeros(len(uN20final))
comp100 = np.zeros(len(uN20final))
for j in range(len(uN20final)):
    comp20[j] = uN20[j]
    comp40[j] = uN40[2*j]
    comp60[j] = uN60[3*j]
    comp80[j] = uN80[4*j]
    comp100[j] = uN100[5*j]

err20 = LA.norm(comp20 - comp100)
err40 = LA.norm(comp40 - comp100)
err60 = LA.norm(comp60 - comp100)
err80 = LA.norm(comp80 - comp100)

err = np.array([err20, err40, err60, err80])

print(f'Error 20 = {err20}')
print(f'Error 40 = {err40}')
print(f'Error 60 = {err60}')
print(f'Error 80 = {err80}')

nvect = np.array([20, 40, 60, 80])
errx = np.array([nvect[0], nvect[-1]])
erry = np.array([err[0], err[-1]])

plt.figure(figsize=(8, 8))
plt.loglog(errx, erry, linewidth=4, linestyle='--', color='k', label ='$O(\Delta x
    ^2)$')
plt.scatter(nvect, err, s=100, color='b', label ='$||e||_2$')
plt.legend()
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.