Dynamic Semi-Structured Meshes for Fast Numerical Simulation of Multi-phase Modelling in the Energy Industry

Report 1: Progress within the First 3 Months of the Project

Amin Nadimy

Supervisors: Prof. C. Pain, Dr. P. Salinas, Dr. A.I.B. Obeysekara
Industrial Supervisors: Dr. A. Nicolle

December 2020
Imperial College London
Department of Earth Science and Engineering

Abstract

This PhD project will develop and implement a semi-structured mesh approach within the Computational Fluid Dynamics (CFD) framework Fluidity/IC-FERST to significantly improve its speed, which requires the model development, including the integration of a custom linear solver based on the PETSc framework. Fluidity/IC-FERST incorporates state-of-the-art technology for multiphase Navier-Stokes flow simulation, including unstructured Dynamic Mesh Optimisation (DMO), high order element methods. Applications of Fluidity/IC-FERST are numerous and include simulation of jets, droplets, boiling reactors, flow in pipes, mixing tanks, etc.
The performance of the developed model will validate against multiphase flow models, such as two-phase jet flow. The use of semi-structured meshes combined with the present Dynamic Mesh Optimisation methodology already represent within Multiphase- Fluidity will enable to run simulations much faster and with a much higher degree of precision than without this technology.
This report summarises the initial three months of this PhD project starting from October 2020. This report demonstrates some numerical methods (e.g. Finite Difference, Finite Volume and Finite Element methods) applied to the square wave equation in 1D and 2D. Each method is described with mathematical formulations followed by code snippets; benchmarked and validated results of the simulations are compared to each other. Finite element method is the main focus of the report that consists of Continuous-Galerkin method (CG-FEM), Discontinuous-Galerkin method (DG-FEM) and Discontinuous Petrov-Galerkin method (DPG-FEM).

Contents

1 Progress

This PhD project started from October 2020 by studying fundamental theories of numerical methods, introduction to Terminal (BASH programming), Introduction to Linux operating system, writing in LaTex Format and developing programming skills in Python.

In the first week, the square wave equation in 1D was investigated using the Finite Difference Method (FDM). The corresponding mathematical model was developed, and the model was implemented in Python. The sensitivity analysis on the Courant–Friedrichs–Lewy number (CFL) was done, see Section 3. Starting from the second week of October, the same 1D square wave equation was studied using FVM. As it was expected the results of this method was similar to FDM. From the third week, the same 1D square wave was investigated using CG-FEM, DG-FEM and DPG-FEM. Then, the problem in 2D was studied using FDM.

A Gantt chart highlighting the 3-month progress achieved from October to December 2020 is illustrated in Figure 1. Most of the time, it has been shown that it was dedicated to the Finite Element Method study (shown in green). The grey items relate to the computational aspect of the learning progress e.g. Python or Linux programming.

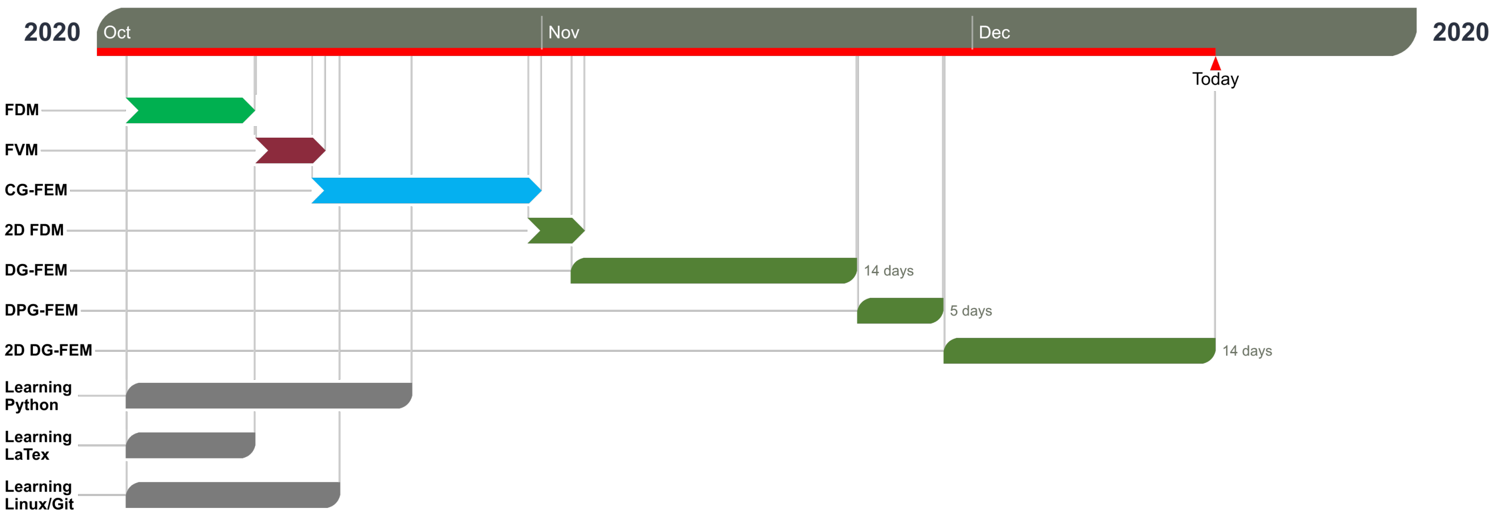

Figure 1: Tasks completed in the first three months of PhD project.

2 Work plan

At the beginning of the year 2021, Discontinuous Petrov-Galerkin method in 2D and semi-structured meshes will be studied. Then, slug flows will be simulated by the current version of Fluidity/IC-FERST. Then, the applications and development of Fluidity/IC-FERST will be investigated by studying pure advection problems. The second-year will be dedicated to incompressible flow problems and the third year will be dedicated to large scale parallel simulations and generating results. Besides, the developed method in this project will be compared to the current method in Fluidity/IC-FERST throughout these three years.

3 Finite Difference Method In 1D

In numerical analysis, the finite Difference Method (FDM) is a class of numerical techniques for solving differential equations by approximating derivatives with finite differences. Both space and time intervals are discretised into a finite number of steps. Only the value of the solutions at these discrete points are approximated by solving algebraic system of equations containing equal numbers of unknowns and equations. Each value of unknowns depends on neighbouring points, which also depend on the type of FDM used to construct the system of algebraic equations [1].

Finite difference methods convert ordinary differential equations (ODE) or partial differential equations (PDE), which may be nonlinear, into a system of linear equations that can be solved by matrix algebra techniques

3.1 Mathematical model

First, The linear unsteady-state wave equation in 1D is introduced here as the governing equation:

$$\frac{\partial U}{\partial t} + c\frac{\partial U}{\partial x} = 0 \tag{1}$$

where C is the wave speed, U is the unknown time varying value, $x_i : i = 0, 1, ..., n$ and $t_n : n = 0, 1, ..., n$. Unknown U depends on space and time so the equation ca be expressed:

$$\frac{\partial U(x_i, t^n)}{\partial t} + c\frac{\partial U(x_i, t^n)}{\partial x} = 0 \tag{2}$$

Boundary conditions for this problem are defines as $U(0, t) = 0$ and $U(L, t) = 0$. The initial condition is set to be:

$$\begin{cases} \frac{nx-1}{4} < i < \frac{nx-1}{2}, U(i,0) = 1 \\ U(i,0) = 0, else \end{cases}$$

Expanding the equation by Taylor's Theorem and using forward and backward Euler's method for space and time, respectively, the discretised form of Equation 2 becomes:

$$U_i^{n+1} = U_i^n - c\frac{\Delta t}{\Delta x}(U_i^n - U_{i-1}^n) \tag{3}$$

3.2 Results

The results of implementing Equation 3 in Python are shown in this section. the effects of Courant number (CFL) are illustrated by keeping all the parameters the same and varying CFL between 0.7 and 1. The shape of the wave does change in time if the ratio among wave speed, space and time discretised lengths is not chosen carefully. By forcing the ratio to one it can be seen that the shape conserved during the wave propagation.
Figure 2 shows the results for simulating Equation 3 in three time-steps, 1st, 50th and 99th. It can be seen that as the wave propagates, its shape changes due to CFL lower than 1.

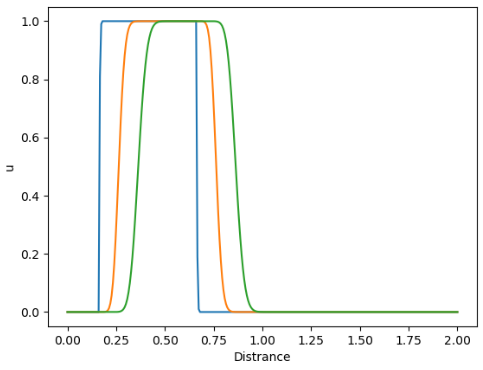

Figure 2: Number of time steps=100, number of points in the domain=500, Courant Number=0.7.

Figure 3 shows the results for simulating Equation 3 in three time-steps, 1st, 50th and 99th. This time CFL number equals 1, so it can be seen that as the wave propagates, its shape conserves.

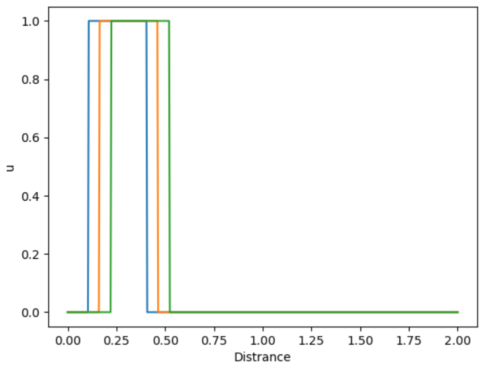

Figure 3: Number of time steps=100, number of points in the domain=500, Courant Number=1.

4 Finite Difference Method In 2D

4.1 Mathematical model

The procedure to investigate FDM in 2D is exactly the same as 1D, but the wave movement is considered in its x and y components.

$$\frac{\partial U}{\partial t} + c_x \frac{\partial U}{\partial x} + c_y \frac{\partial U}{\partial y} = 0 \tag{4}$$

Boundary Conditions:

$$\begin{cases} \text{U(0, 0, t)} = 0 \\ \text{U(L,t)} = 0 \end{cases}$$

Initial Conditions:

$$\begin{cases} \text{U(i,0)} = 1,\ \frac{nx-1}{4} < i < \frac{nx-1}{2} \\ \text{U(i,0)} = 0, \text{ else} \end{cases}$$

The discretisation of the wave equation in 2D is done by using forward and backward Euler's methods for space and time, respectively.

$$\frac{U_{i,j}^{t+1} - U_{i,j}^{t}}{\Delta t} + c_x\left(\frac{U_{i,j}^{t} - U_{i-1,j}^{t}}{\Delta x}\right) + c_y\left(\frac{U_{i,j}^{t} - U_{i,j-1}^{t}}{\Delta y}\right) = 0 \tag{5}$$

Where i and j denote x and y directions, respectively.
After variable separation, the discretised form of Equiation 4 becomes:

$$U_{i,j}^{t+1} = U_{i,j}^{t} - c_x \frac{\Delta t}{\Delta x}(U_{i,j}^{t} - U_{i-1,j}^{t}) - c_y \frac{\Delta t}{\Delta y}(U_{i,j}^{t} - U_{i,j-1}^{t}) \tag{6}$$

4.2 Results

In this section, the result of simulating Equation 6 is presented. It is shown in Figure fig FDM 2D that lower CFL number than 1, leads to a change in the wave shape over time. The beginning, the middle and the end time steps are chosen to be presented.

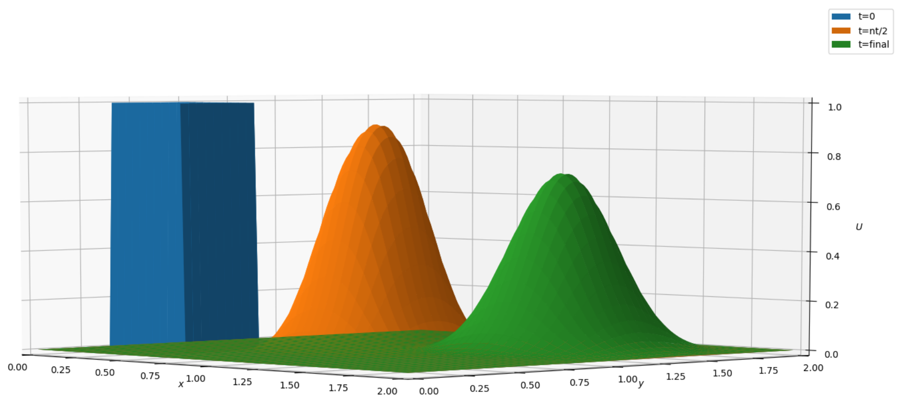

Figure 4: Number of time steps=5000, number of number of x-steps=100, number of y-steps=100 Courant Number=0.01, x-velocity=0.1, y-velocity=0.1.

5 Finite Volume Method In 1D

Finite Volume Method (FVM) is another powerful method for evaluating partial differential equations in the form of algebraic equations. Finite Volume refers to small volumes around each node of the grid. In this method, the divergence theorem is used to convert Volume integrals in 3D or surface integrals in 2D or 1D are to surface and line integrals, respectively. Then, these terms are evaluated at the surfaces, in 3D, or faces, in 2D, as fluxes for each finite volume. The FV method is conservative, leaving flux from an adjacent volume is identical to that of entering the neighbouring volume. This method is widely used in unstructured meshes due to the easy formulation of the method [2].

The FV method evaluates exact expressions for the average value of the solution and constructs approximation of the solution for each cell [2].

5.1 Mathematical model

Starting with the linear unsteady-state wave equation in 1D as the governing equation:

$$\frac{\partial U}{\partial t} + c\frac{\partial U}{\partial x} = 0 \tag{7}$$

Boundary Conditions are defined as below:

$$\begin{cases} U(0,t) = 0 \\ U(L,t) = 0 \end{cases}$$

Initial Conditions are defined as below:

$$\begin{cases} \frac{nx-1}{4} < i < \frac{nx-1}{2}, U(i,0) = 1 \\ else, U(i,t) = 0 \end{cases}$$

In this equation space and time domains need to be discretised, so we integrate the governing equation over the domains:

$$\int_{(i-\frac{1}{2})}^{(i+\frac{1}{2})} \int_{t}^{(t+\Delta t)} \frac{\partial U}{\partial t} dtdx + c\int_{t}^{(t+\Delta t)} \int_{(i-\frac{1}{2})}^{(i+\frac{1}{2})} \frac{\partial U}{\partial t} dxdt = 0 \tag{8}$$

$$\int_{(i-\frac{1}{2})}^{(i+\frac{1}{2})} [U^{(t+\Delta t)} - U^{t}]dx + c\int_{t}^{(t+\Delta t)} [U_{(i+\frac{1}{2})} - U_{(i-\frac{1}{2})}]dt = 0 \tag{9}$$

Using the explicit method in time and separating the unknown variable, the formula becomes:

$$U_i^{(t+\Delta t)} - U_i^t = -\frac{c\Delta t}{\Delta x}(U^t_{(i+\frac{1}{2})} - U^t_{(i-\frac{1}{2})}) \tag{10}$$

$$U_i^{(t+\Delta t)} = U_i^t - \frac{c\Delta t}{\Delta x}(U^t_{(i+\frac{1}{2})} - U^t_{(i-\frac{1}{2})}) \tag{11}$$

Here the Upwind method is used and the finial discretised formula becomes:

$$U_i^{(t+\Delta t)} = U_i^t - \frac{c\Delta t}{\Delta x}(U_{(i+1)}^t - U_{(i-1)}^t) \tag{12}$$

5.2 Results

The results of FVM in 1D is identical to FDM as their discretised formula are the same. Figure 5 shows the results for simulating Equation 12 in three time-steps, 1st, 50th and 99th. It can be seen that as the wave propagates, its shape changes due to CFL lower than 1.

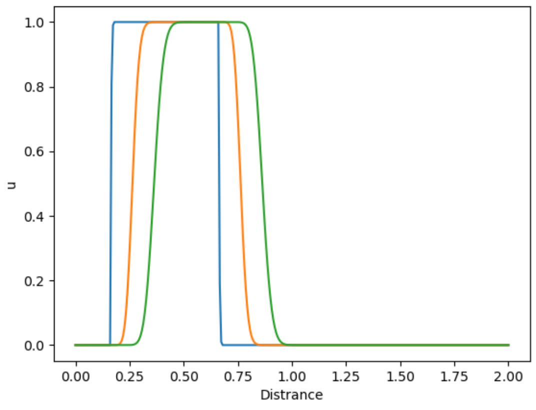

Figure 5: Number of time steps=100, number of nx=500, CFL=0.7, velocity=0.1.

Figure 6 shows the results for simulating Equation 12 in three time-steps, 1st, 50th and 99th. This time CFL number equals 1, which shows as the wave propagates, the shape conserves.

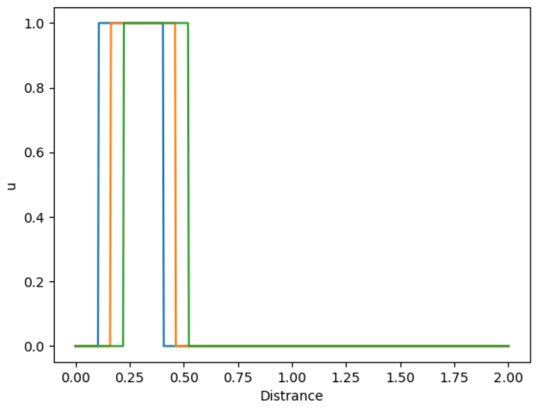

Figure 6: Number of time steps=100, number of nx=500, CFL=1, velocity=0.1.

6 Continuous-Galerkin Finite Element Method In 1D

The finite element method (FEM) is the most widely used method for solving problems of engineering and mathematical models, especially in fluid dynamics. The FEM is another numerical method for solving partial differential equations. This method subdivides the domain into smaller sections called elements. Algebraic system of equations is the results of FEM discretisation which has a finite number of points. This method approximates the solutions over the domain at the elemental scale. Then, the global system representing the whole problem is constructed. This method minimises the associated error function using the variational methods introduced by variations of the solution approximations. In the continuous-Galerkin method, the solutions are continuous throughout the domain and also at the boundaries [3].

6.1 Mathematical model

$$\frac{\partial U}{\partial t} + c\frac{\partial U}{\partial x} = 0 \tag{13}$$

Boundary Conditions:

U(0, t) = 0

U(L,t) = 0

Now both sides are multiply by a weighting function w and integrate over each element. Each element starting point is a and endpoint is b:

$$\int_a^b [\frac{\delta U}{\delta t} w]dx + c\int_a^b [\frac{\delta U}{\delta x} w] = 0 \tag{14}$$

The weak form at the element level becomes:

$$\int_a^b [\frac{\delta U}{\delta t} w - c.U\frac{\delta w}{\delta x}]dx = -cUw]_a^b \tag{15}$$

Then, Forward Euler's method is used to replace $\frac{\delta U}{\delta t}$:

$$\int_a^b [\frac{U^{t+1} - U^t}{\Delta t} w]dx - \int_a^b [c.U\frac{\delta w}{\delta x}]dx = -cUw]_a^b \tag{16}$$

$$\int_a^b [\frac{U^{t+1}}{\Delta t} w]dx - \int_a^b [\frac{U^t}{\Delta t} w]dx - \int_a^b [c.U\frac{\delta w}{\delta x}]dx = -cUw]_a^b \tag{17}$$

Based on the Galerkin's method the basis function equals to the shape function:

$$w = \phi_i \tag{18}$$

$$U(t,x) = \sum_{j=1}^{2} U_j^t \phi_j(x) \tag{19}$$

where i is the i^{th} node in the element, which here i is 2 because of linear interpolation and j is the j^{th} function of U. Also let's take the length of each element as h.

$$\int_0^h [\sum_{j=1}^{2} [U_j^{t+1}\phi_j(x)]\frac{\phi_i}{\Delta t}]dx - \int_0^h [\sum_{j=1}^{2} [U_j^{t}\phi_j(x)]\frac{\phi_i}{\Delta t}]dx - \int_0^h [c\sum_{j=1}^{2} [U_j^{t}\phi_j(x)]\frac{d\phi_i}{dx}]dx = -cUw]_a^b \tag{20}$$

$$\frac{\sum_{j=1}^{2}[U_j^{t+1}]}{\Delta t} \underbrace{\int_0^h [\phi_i(x)\sum_{j=1}^{2}\phi_j(x)]dx}_{A} = \frac{\sum_{j=1}^{2}[U_j^{t}]}{\Delta t} \underbrace{\int_0^h [\phi_i(x)\sum_{j=1}^{2}[\phi_j(x)]dx}_{A}$$
$$+ c\sum_{j=1}^{2}[U_j^t] \underbrace{\int_0^h [\sum_{j=1}^{2}[\phi_j(x)]\frac{d\phi_i}{dx}]dx}_{B} - \underbrace{cUw]_a^b}_{D} \tag{21}$$

If we consider Upwind method with $c > 0$, a and b are element boundaries, D becomes:

$$D = cU_{k(b)}\phi_{i(b)} - (cU_{(k-1)(b)}\phi_{i(a)}) \tag{22}$$

where k is the number of element.
At the left boundary of the element k, $\phi_{1(b)} = 0$ and $\phi_{1(a)} = 1$:

$$cU_{k(b)}\phi_{1(b)} - cU_{k-1(b)}\phi_{1(a)} = -cU_{k-1(b)} \tag{23}$$

At the right boundary of element k, $\phi_{2(a)} = 0$ and $\phi_{2(b)} = 1$:

$$cU_{k(b)}\phi_{2(b)} - cU_{k-1(b)}\phi_{2(a)} = cU_{k(b)} \tag{24}$$

Therefore, the D matrix for one element becomes:

$$D = c\begin{bmatrix} -U_{k-1}(x_R) \\ U_k(x_R) \end{bmatrix} \tag{25}$$

$$\text{Defining} \begin{cases} \bar{U}_i^{t+1} = \sum_{j=1}^{2}[U_j^{t+1}] \\ \bar{U}_i^{t} = \sum_{j=1}^{2}[U_j^{t}] \\ D = 0, \text{ boundary conditions} \end{cases} :$$

$$\frac{A}{\Delta t}\bar{U}_i^{t+1} = \frac{A}{\Delta t}\bar{U}_i^{t} + cB\bar{U}_i^{t} \tag{26}$$

6.1.1 Explicit method

$$\bar{U}_i^{t+1} = \bar{U}_i^t + c\Delta t A^{-1} B \bar{U}_i^t \tag{27}$$

From Lagrangian interpolation $\begin{cases} \phi_1 = 1 - \frac{\bar{x}}{h} \\ \phi_2 = \frac{\bar{x}}{h} \\ \frac{d\phi_1}{dx}\phi_1 = \frac{-1}{h} \\ \frac{d\phi_2}{dx}\phi_1 = \frac{1}{h} \end{cases}$:

Considering only one element and $A = \int_0^h \phi_i \phi_j d\bar{x}$ and $B = \int_0^h \phi_j \frac{d\phi_i}{d\bar{x}} d\bar{x}$:
For $i = 1$ and $j = 1$:

$$A = \frac{h}{3} \tag{28}$$

$$B = \frac{-1}{2} \tag{29}$$

For $i = 1$ and $j = 2$:

$$A = \frac{h}{6} \tag{30}$$

$$B = \frac{-1}{2} \tag{31}$$

For $i = 2$ and $j = 1$:

$$A = \frac{h}{6} \tag{32}$$

$$B = \frac{1}{2} \tag{33}$$

For $i = 2$ and $j = 2$:

$$A = \frac{h}{3} \tag{34}$$

$$B = \frac{1}{2} \tag{35}$$

$$B = \begin{bmatrix} \frac{-1}{2} & \frac{-1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix}, A = \begin{bmatrix} \frac{h}{3} & \frac{h}{6} \\ \frac{h}{6} & \frac{h}{3} \end{bmatrix} \longrightarrow A^{-1} = \begin{bmatrix} \frac{4}{h} & \frac{-2}{h} \\ \frac{-2}{h} & \frac{4}{h} \end{bmatrix} \xrightarrow{A^{-1} \times B} = \frac{1}{h} \begin{bmatrix} -3 & -3 \\ 3 & 3 \end{bmatrix}$$

So Equation 27 becomes:

$$\begin{bmatrix} \bar{U}_1^{t+1} \\ \bar{U}_2^{t+1} \end{bmatrix} = \begin{bmatrix} \bar{U}_1^t \\ \bar{U}_2^t \end{bmatrix} + \frac{c\Delta t}{h} \times \begin{bmatrix} -3 & -3 \\ 3 & 3 \end{bmatrix} \begin{bmatrix} \bar{U}_1^t \\ \bar{U}_2^t \end{bmatrix} \tag{36}$$

For point 1 and point 2 in an element:

$$\bar{U}_1^{t+1} = \bar{U}_1^t - \frac{c\Delta t}{h}(\bar{U}_1^t + \bar{U}_2^t) \tag{37}$$

$$\bar{U}_2^{t+1} = \bar{U}_2^t + \frac{c\Delta t}{h}(\bar{U}_1^t + \bar{U}_2^t) \tag{38}$$

6.1.2 Implicit method

Continue from Equation 26:

$$A\bar{U}_i^{t+1} = \underbrace{(A + c\Delta t B)}_{E}\bar{U}_i^t \tag{39}$$

Here for CG matrix D is zero.

$$\begin{bmatrix} \frac{h}{3} & \frac{h}{6} \\ \frac{h}{6} & \frac{h}{3} \end{bmatrix} \begin{bmatrix} U_1^{t+1} \\ U_2^{t+1} \end{bmatrix} = \begin{bmatrix} \frac{h}{3} - \frac{c\Delta t}{2} & \frac{h}{6} - \frac{c\Delta t}{2} \\ \frac{h}{6} + \frac{c\Delta t}{2} & \frac{h}{3} + \frac{c\Delta t}{2} \end{bmatrix} \begin{bmatrix} U_1^t \\ U_2^t \end{bmatrix} \tag{40}$$

Global matrix of RHS:

$$LHS = \begin{bmatrix} \frac{h}{3} & \frac{h}{6} & 0 & 0 & 0 & 0 & ... & 0 \\ \frac{h}{6} & \frac{h}{3} + \frac{h}{3} & \frac{h}{6} & 0 & 0 & 0 & ... & 0 \\ 0 & \frac{h}{6} & \frac{h}{3} + \frac{h}{3} & \frac{h}{6} & 0 & 0 & ... & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & ... & \vdots \\ 0 & 0 & 0 & 0 & ... & 0 & \frac{h}{6} & \frac{h}{3} \end{bmatrix} \begin{bmatrix} U_1^{t+1} \\ U_2^{t+1} \\ U_3^{t+1} \\ \vdots \\ U_N^{t+1} \end{bmatrix} \tag{41}$$

$$B = \begin{bmatrix} \frac{-1}{2} & \frac{-1}{2} & 0 & 0 & 0 & 0 & ... & 0 \\ \frac{1}{2} & \frac{1}{2} + \frac{-1}{2} & \frac{-1}{2} & 0 & 0 & 0 & ... & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} + \frac{-1}{2} & \frac{-1}{2} & 0 & 0 & ... & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & ... & \vdots \\ 0 & 0 & 0 & 0 & ... & 0 & \frac{1}{2} & \frac{1}{2} \end{bmatrix} \tag{42}$$

6.2 Results and CFL sensitivity analysis

In this section, the simulation results and sensitivity analysis on CFL and the number of elements in CG-FEM are represented. Figure 7 shows the comparison between the simulation results of FDM and CG-FEM with the same parameter values. Considering the stated parameters for Figure 7, it is shown that generally, FDM generates smoother results compared to standard CG-FEM, which gives strong oscillations.

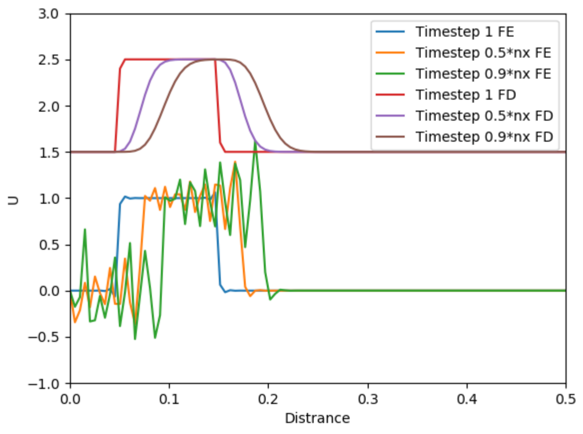

Figure 7: Number of time steps=100, number of elements=100, Courant Number=0.1.

Changing CFL number from 0.1 to 0.01 and increasing time steps from 100 to 1000 resulted in less oscillation in CG-FEM while the outcome of FDM stayed almost the same. See Figure 8.

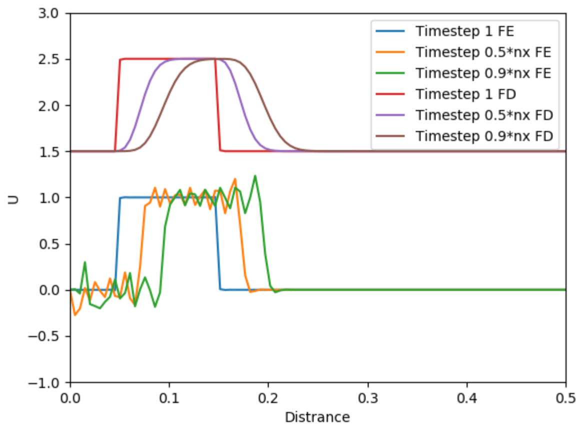

Figure 8: Number of time steps=1000, number of elements=100, Courant Number=0.01.

One way to generate a better result with less oscillation is to the increasing number of elements in the domain. Figure 9 is the results of increasing the number of elements from 100 to 1000, which simulates the wave very well apart around the jumps.

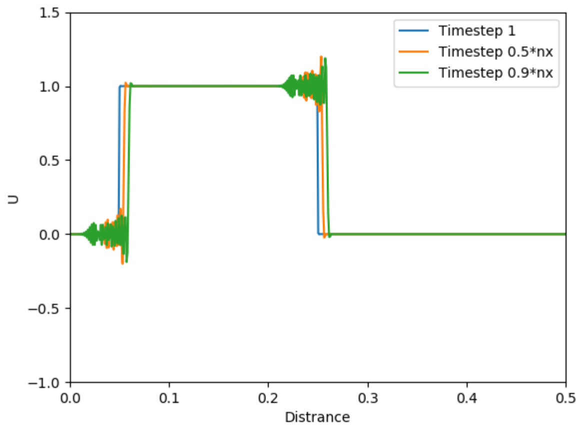

Figure 9: Number of time steps=1000, number of elements=500, Courant Number=0.01

7 Discontinuous Galerkin FEM in 1D

7.1 Mathematical model

$$\frac{\partial U}{\partial t} + \nabla.(cU) = 0 \tag{43}$$

Boundary Conditions:
$U(0,t) = U(l,t) = 0$, $0 \leq \text{x} \leq \text{l}$ for $t \geq 0$
Initial conditions:
$U(x, t = 0) =$ Known (square wave)
Estimation of $U(t,x) \approx \bar{U}(t,x)$:

$$\bar{U}(t,x) = \sum_{j=1}^{2} U_j^t \phi_j(x) \tag{44}$$

Both sides are multiplied by a weighting function w and are integrated over the domain Ω.

$$\int_\Omega [\frac{\partial \bar{U}}{\partial t} w] d\Omega + \int_\Omega [\nabla.(c\bar{U})w] d\Omega = 0 \tag{45}$$

$$\int_\Omega [\frac{\partial \bar{U}}{\partial t} w - c\bar{U}\nabla.w] d\Omega + \int_\Gamma \hat{n}.c\bar{U}w d\Gamma = 0 \tag{46}$$

For element e the equation becomes:

$$\int_{\Omega_e} \frac{\partial \bar{U}}{\partial t} w d\Omega_e - \int_{\Omega_e} c\bar{U}\nabla.w d\Omega_e + \int_{\Gamma_e} \hat{n}.c\bar{U}w d\Gamma_e = 0 \tag{47}$$

Taking $w = \phi_i$, where i is the i^{th} node in the element,which here i is 2 because of the linear interpolation function and Equation (47) becomes:

$$\int_{\Omega_e} \frac{\partial \bar{U}}{\partial t} \phi_i d\Omega_e - \int_{\Omega_e} [c\bar{U}\frac{\partial \phi_i}{\partial x}] dx = -\oint_{\Gamma_e} \hat{n}.c\bar{U}\phi_i d\Gamma_e \tag{48}$$

Here, j is the j^{th} function of U. Now considering the length of each element as h.

$$\int_{x_0}^{x_1} \frac{\partial \sum_{j=1}^{2}[U_j^t \phi_j(x)]}{\partial t} \sum_{i=1}^{2} \phi_i dx - \int_{x_0}^{x_1} [\sum_{j=1}^{2} [cU_j^t \phi_j(x)] \frac{\partial [\sum_{i=1}^{2} \phi_i]}{\partial x}] dx = \\ -(\oint_{\Gamma_e} \hat{n_x}.c\bar{U}_{in}\phi_i d\Gamma_e - \oint_{\Gamma_e} \hat{n_x}.c\bar{U}_{out}\phi_i d\Gamma_e) \tag{49}$$

$$\frac{\partial \sum_{j=1}^{2}[U_j^t]}{\partial t} \underbrace{\int_{x_0}^{x_1} [\sum_{i=1}^{2} \phi_i(x) \sum_{j=1}^{2} \phi_j(x)] dx}_{M} = \sum_{j=1}^{2}[U_j^t] \underbrace{\int_{x_0}^{x_1} [c \sum_{j=1}^{2} \phi_j(x)] \frac{\partial[\sum_{i=1}^{2} \phi_i]}{\partial x}] dx}_{K} - \underbrace{\left[c(x_1)\bar{U}^t(x_1^-)\phi_j(x_1)\phi_i(x_1) - c(x_0)\bar{U}^t(x_0^-)\phi_j(x_0)\phi_i(x_0)\right]}_{F} \tag{50}$$

$$M \frac{\partial \sum_{j=1}^{2}[U_j^t]}{\partial t} = K \sum_{j=1}^{2}[U_j^t] - F \tag{51}$$

Now, Forward Euler's method is applied to $\frac{\partial U}{\partial t}$:

$$\frac{\partial U_j^t}{\partial x} = \frac{U_j^{t+1} - U_j^t}{\Delta t} \tag{52}$$

$$M \frac{\sum_{j=1}^{2}[U_j^{t+1}] - \sum_{j=1}^{2}[U_j^t]}{\Delta t} = K \sum_{j=1}^{2}[U_j^t] - F \tag{53}$$

$$M \frac{\sum_{j=1}^{2}[U_j^{t+1}]}{\Delta t} = M \frac{\sum_{j=1}^{2}[U_j^t]}{\Delta t} + K \sum_{j=1}^{2}[U_j^t] - F \tag{54}$$

$$M \sum_{j=1}^{2}[U_j^{t+1}] = (M + \Delta t K) \sum_{j=1}^{2} [U_j^t] - \Delta t F \tag{55}$$

If the Upwind flux method with $c > 0$ is used, x_0 and x_1 are the element boundaries and F becomes:

$$F = c(x_1)\bar{U}^t(x_1^-)\phi_j(x_1)\phi_i(x_1) - c(x_0)\bar{U}^t(x_0^-)\phi_j(x_0)\phi_i(x_0) \tag{56}$$

At x_0, $\phi_1 = 1$ and $\phi_2 = 0$ and at x_1, $\phi_1 = 0$ and $\phi_2 = 1$:

$$\phi_{1(x_1)}^2 - \phi_{1(x_0)} = -\phi_{1(x_0)}^2 = -1 \tag{57}$$

$$\phi_{2(x_1)}^2 - \phi_{2(x_0)}^2 = 1 \tag{58}$$

Therefore, the D matrix for element k can express as:

$$F = c \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} U_{(x_0)}^- \\ U_{(x_1)}^- \end{bmatrix} \tag{59}$$

From Lagrangian interpolation rules: $\begin{cases} \phi_1 = 1 - \frac{\bar{x}}{h} \\ \phi_2 = \frac{\bar{x}}{h} \\ \frac{d\phi_1}{dx} = \frac{-1}{h} \\ \frac{d\phi_2}{dx} = \frac{1}{h} \end{cases}$

Considering only one element and $M = \int_0^h \phi_i \phi_j d\bar{x}$ and if c is constant then $K = c\int_0^h \phi_j \frac{d\phi_i}{d\bar{x}} d\bar{x}$:
For $i = 1$ and $j = 1$:

$$M = \frac{h}{3} \tag{60}$$

$$K = \frac{-1}{2} \tag{61}$$

For $i = 1$ and $j = 2$:

$$M = \frac{h}{6} \tag{62}$$

$$K = \frac{-1}{2} \tag{63}$$

For $i = 2$ and $j = 1$:

$$M = \frac{h}{6} \tag{64}$$

$$K = \frac{1}{2} \tag{65}$$

For $i = 2$ and $j = 2$:

$$M = \frac{h}{3} \tag{66}$$

$$K = \frac{1}{2} \tag{67}$$

$K = \begin{bmatrix} \frac{-1}{2} & \frac{-1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix}$, $M = \begin{bmatrix} \frac{h}{3} & \frac{h}{6} \\ \frac{h}{6} & \frac{h}{3} \end{bmatrix}$

$$\begin{bmatrix} \frac{h}{3} & \frac{h}{6} \\ \frac{h}{6} & \frac{h}{3} \end{bmatrix} \begin{bmatrix} U_1^{t+1} \\ U_2^{t+1} \end{bmatrix} = \begin{bmatrix} \frac{h}{3} & \frac{h}{6} \\ \frac{h}{6} & \frac{h}{3} \end{bmatrix} \begin{bmatrix} U_1^t \\ U_2^t \end{bmatrix} + \begin{bmatrix} \frac{-c\Delta t}{2} & \frac{-c\Delta t}{2} \\ \frac{c\Delta t}{2} & \frac{c\Delta t}{2} \end{bmatrix} \begin{bmatrix} U_1^t \\ U_2^t \end{bmatrix} - \begin{bmatrix} -c\Delta t & 0 \\ 0 & c\Delta t \end{bmatrix} \begin{bmatrix} U_{(x_0)}^- \\ U_{(x_1)}^- \end{bmatrix} \tag{68}$$

$$\begin{bmatrix} \frac{h}{3} & \frac{h}{6} \\ \frac{h}{6} & \frac{h}{3} \end{bmatrix} \begin{bmatrix} U_1^{t+1} \\ U_2^{t+1} \end{bmatrix} = \left(\begin{bmatrix} \frac{h}{3} & \frac{h}{6} \\ \frac{h}{6} & \frac{h}{3} \end{bmatrix} + \begin{bmatrix} \frac{-c\Delta t}{2} & \frac{-c\Delta t}{2} \\ \frac{c\Delta t}{2} & \frac{c\Delta t}{2} \end{bmatrix} - \begin{bmatrix} 0 & 0 \\ 0 & c\Delta t \end{bmatrix} \right) \begin{bmatrix} U_1^t \\ U_2^t \end{bmatrix} \tag{69}$$

7.2 Consistent mass matrix

For two elements:

$$\underbrace{\begin{bmatrix} \frac{h}{3} & \frac{h}{6} & 0 & 0 \\ \frac{h}{6} & \frac{h}{3} & 0 & 0 \\ 0 & 0 & \frac{h}{3} & \frac{h}{6} \\ 0 & 0 & \frac{h}{6} & \frac{h}{3} \end{bmatrix}}_{M} \begin{bmatrix} U_1^{t+1} \\ U_2^{t+1} \\ U_3^{t+1} \\ U_4^{t+1} \end{bmatrix} =$$

$$\left(\underbrace{\begin{bmatrix} \frac{h}{3} & \frac{h}{6} & 0 & 0 \\ \frac{h}{6} & \frac{h}{3} & 0 & 0 \\ 0 & 0 & \frac{h}{3} & \frac{h}{6} \\ 0 & 0 & \frac{h}{6} & \frac{h}{3} \end{bmatrix}}_{M} + \underbrace{\begin{bmatrix} \frac{-c\Delta t}{2} & \frac{-c\Delta t}{2} & 0 & 0 \\ \frac{c\Delta t}{2} & \frac{c\Delta t}{2} & 0 & 0 \\ 0 & 0 & \frac{-c\Delta t}{2} & \frac{-c\Delta t}{2} \\ 0 & 0 & \frac{c\Delta t}{2} & \frac{c\Delta t}{2} \end{bmatrix}}_{K} + \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -c\Delta t & 0 & 0 \\ 0 & c\Delta t & 0 & 0 \\ 0 & 0 & 0 & -c\Delta t \end{bmatrix}}_{F} \right) \begin{bmatrix} U_1^t \\ U_2^t \\ U_3^t \\ U_4^t \end{bmatrix} \tag{70}$$

$$MU_i^{t+1} = (M + K + F)U_i^t \tag{71}$$

$$U_i^{t+1} = M^{-1}(M + K + F)U_i^t \tag{72}$$

7.3 Lumped matrix

Diagonalising mas matrix using row sum method gives:

$$\underbrace{\begin{bmatrix} \frac{h}{2} & 0 & 0 & 0 \\ 0 & \frac{h}{2} & 0 & 0 \\ 0 & 0 & \frac{h}{2} & 0 \\ 0 & 0 & 0 & \frac{h}{2} \end{bmatrix}}_{M} \begin{bmatrix} U_1^{t+1} \\ U_2^{t+1} \\ U_3^{t+1} \\ U_4^{t+1} \end{bmatrix} =$$

$$\left(\underbrace{\begin{bmatrix} \frac{h}{2} & 0 & 0 & 0 \\ 0 & \frac{h}{2} & 0 & 0 \\ 0 & 0 & \frac{h}{2} & 0 \\ 0 & 0 & 0 & \frac{h}{2} \end{bmatrix}}_{M} + \underbrace{\begin{bmatrix} \frac{-c\Delta t}{2} & \frac{-c\Delta t}{2} & 0 & 0 \\ \frac{c\Delta t}{2} & \frac{c\Delta t}{2} & 0 & 0 \\ 0 & 0 & \frac{-c\Delta t}{2} & \frac{-c\Delta t}{2} \\ 0 & 0 & \frac{c\Delta t}{2} & \frac{c\Delta t}{2} \end{bmatrix}}_{K} + \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & -c\Delta t & 0 & 0 \\ 0 & c\Delta t & 0 & 0 \\ 0 & 0 & 0 & -c\Delta t \end{bmatrix}}_{F} \right) \begin{bmatrix} U_1^t \\ U_2^t \\ U_3^t \\ U_4^t \end{bmatrix} \tag{73}$$

Explicit equations for local U_i^{t+1} become:

$$U_1^{t+1} = U_1^t - \frac{c\Delta t}{h}(U_1^t + U_2^t) \tag{74}$$

$$U_2^{t+1} = U_2^t + \frac{c\Delta t}{h}(U_1^t + U_2^t) - c\Delta t U_2^t \tag{75}$$

$$U_3^{t+1} = U_3^t - \frac{c\Delta t}{h}(U_3^t + U_4^t) + c\Delta t U_2^t \tag{76}$$

$$U_4^{t+1} = U_4^t + \frac{c\Delta t}{h}(U_3^t + U_4^t) - c\Delta t U_4^t \tag{77}$$

7.4 Results

Standard Discontinuous Galerkin method was investigated and the results of using Implicit or Explicit schemes as well as using the consistent or lumped mass matrix are expressed in this section.

7.4.1 Explicit method with lumped mass matrix

The results of DG-FEM with the lumped mass matrix are presented in Figure 10 and Figure 11. Generally, DG-FEM could simulate the wave equation smoother than standard CG-FEM, but still fails to simulate the wave around jumps.

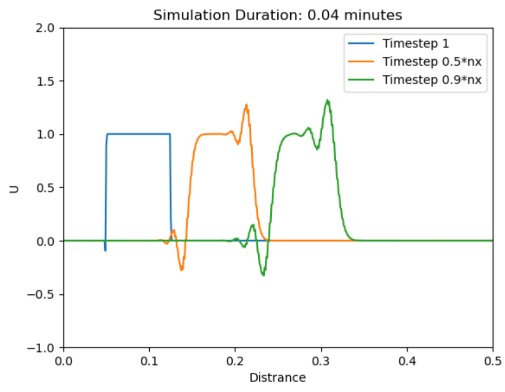

Figure 10: Number of time steps=2000, number of nx=500, CFL=0.05, velocity=0.1.

Increasing the number of elements from 500 to 1000 could reduce the domain of oscillations, but still fails to be stable in the whole domain. See Figure 8.

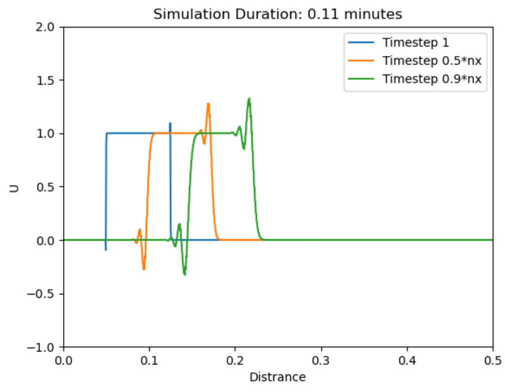

Figure 11: Number of time steps=2000, nx=1000, CFL=0.05, velocity=0.1.

7.4.2 Explicit method with consistent mass matrix

Having the same number of elements, time steps and CFL number as Section 7.4.1, but using the consistent mass matrix, the results show instability abound both sides of jumps, see Figure 12.

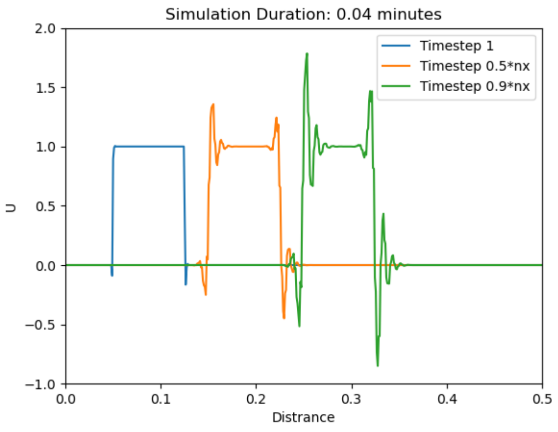

Figure 12: Number of time steps=2000, number of nx=500, CFL=0.05, velocity=0.1.

Similar to Section 7.4.1, increasing the number of elements from 500 to 1000 decreased the domain of instability, see Figure 13. In both cases, the standard DG-FEM failed to generate stable results for the governing equation simulations.

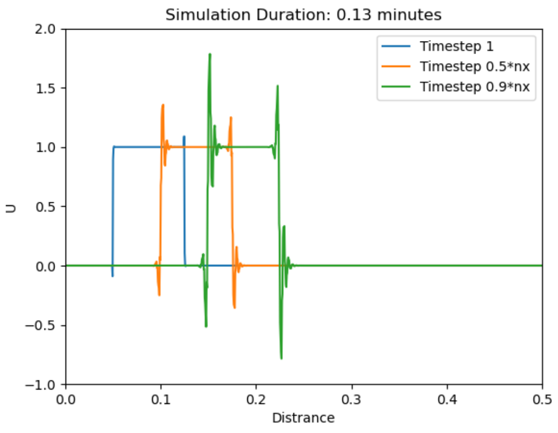

Figure 13: Number of time steps=2000, nx=1000, CFL=0.05, velocity=0.1.

7.4.3 Implicit method with consistent mass matrix

The results for DG-FEM using the implicit scheme and consistent mass matrix are illustrated in Figure 14. As it was expected, the outcome is identical with Explicit scheme when the parameter values are the same, the only difference is in the duration of the simulation. In Implicit method it took five times longer than Explicit method. The simulation time for each method is shown in the corresponding figures.

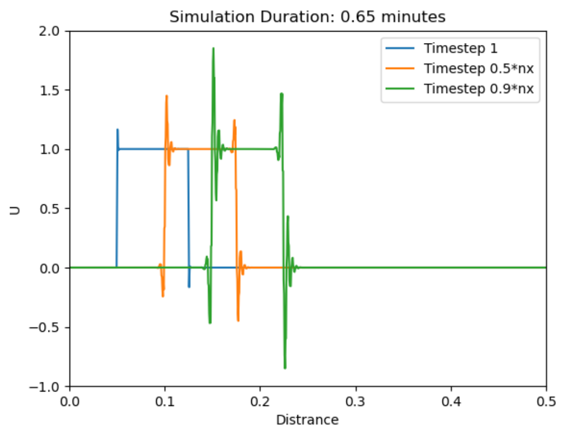

Figure 14: Number of time steps=2000, number of nx=1000, CFL=0.05, velocity=0.1.

8 Petrov-Galerkin FEM in 1D

8.1 Mathematical Model

In this section, the Discontinuous Petrov-Galerkin (DPG) method is used to stabilise the result of the standard DG-FEM.

$$\int_{\Omega}(w\Re - w\mu\frac{\partial P\Re}{\partial x})d\Omega = 0 \tag{78}$$

Where $\Re$ is the residual of the function, μ is the wave travelling direction, P is the stabilisation parameter and w is the weighting function. Implementing expressions for w in (78) and integrating by part, Equation 78 becomes:

$$\int_{\Omega}(\phi_i(x)\Re)d\Omega + \mu\int_{\Omega}(\frac{\partial\phi_i(x)}{\partial x}p\Re)d\Omega - \oint_{\Gamma} n_x.\phi_i(x)P\Re d\Gamma = 0 \tag{79}$$

Within Element method

In this method, the surface term in (79) for the stabilisation scheme is set to zero, which discards any between element coupling. Therefore, Equation 79 becomes:

$$\int_{\Omega}(\phi_i(x)\Re)d\Omega + \mu\int_{\Omega}(\frac{\partial\phi_i(x)}{\partial x}p\Re)d\Omega = 0 \tag{80}$$

Substituting trial function (44) in (80) results in :

$$\underbrace{\int_{\Omega}(\phi_i(x)\Re)d_{\Omega}}_{1st} + \underbrace{\mu\int_{\Omega}\frac{\partial\phi_i}{\partial x}P(\frac{\partial U_j^t\phi_j}{\partial t} + c\frac{\partial U_j^t\phi_j}{\partial x})d\Omega}_{2nd} = 0 \tag{81}$$

Part 1 of Equation 81 was calculated in Section 7; therefore, only the second part of the equation is considered in this section, which its results will be added to the standard DG-FEM.
If $P = 1$ then:

$$\int_{\Omega}(\phi_i(x)\Re) + \underbrace{\left[\frac{\partial U_j^t}{\partial t}\int_{\Omega}\frac{\partial\phi_i}{\partial x}\phi_j dx + cU_j^t\int_{\Omega}\frac{\partial\phi_i}{\partial x}\frac{\phi_j}{\partial x}dx\right]}_{Stabiliser} = 0 \tag{82}$$

Substituting the $\Re$ expression into the 1st term of Equation 82:

$$\frac{\partial U_j^t}{\partial t}\int_{x_0}^{x_1}\phi_i(x)\phi_j(x)dx - U_j^t\int_{x_0}^{x_1}c\phi_j(x)\frac{d\phi_i}{dx}dx + (\oint_{\Gamma_e}\hat{n_x}.c\bar{U}_{in}\phi_i d\Gamma_e - \oint_{\Gamma_e}\hat{n_x}.c\bar{U}_{out}\phi_i d\Gamma_e) + \left[\frac{\partial U_j^t}{\partial t}\int_{\Omega}\frac{\partial\phi_i}{\partial x}\phi_j dx + cU_j^t\int\frac{\partial\phi_i}{\partial x}\frac{\phi_j}{\partial x}dx\right] = 0 \tag{83}$$

$$\frac{\partial U_j^t}{\partial t}\underbrace{\left[\int_\Omega \phi_i\phi_j dx + \int_\Omega \frac{\partial \phi_i}{\partial x}\phi_j dx]\right]}_{M} = U_j^t \underbrace{\left[c\int_\Omega \frac{\partial \phi_i}{\partial x}\phi_j dx - c\int_\Omega \frac{\partial \phi_i}{\partial x}\frac{\partial \phi_j}{\partial x} dx\right]}_{K} - \underbrace{(\oint_{\Gamma_e} \hat{n_x}.c\bar{U}_{in}\phi_i d\Gamma_e - \oint_{\Gamma_e} \hat{n_x}.c\bar{U}_{out}\phi_i d\Gamma_e)}_{F} \quad (84)$$

Substituting (44) into the above equation and collecting relevant terms:

$$MU_j^{t+1} = MU_j^t + KU_j^t - F \quad (85)$$

$$MU_j^{t+1} = (M + \Delta t K)U_j^t - \Delta t F \quad (86)$$

$$M = \begin{bmatrix} \frac{h}{3} & \frac{h}{6} \\ \frac{h}{6} & \frac{h}{3} \end{bmatrix} + \begin{bmatrix} \frac{-1}{2} & \frac{-1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix} \quad (87)$$

$$K = \begin{bmatrix} \frac{-c}{2} & \frac{-c}{2} \\ \frac{c}{2} & \frac{c}{2} \end{bmatrix} - \begin{bmatrix} \frac{c}{h} & \frac{-c}{h} \\ \frac{-c}{h} & \frac{c}{h} \end{bmatrix} \quad (88)$$

$$F = \begin{bmatrix} 0 & 0 \\ 0 & cU_2^t \end{bmatrix} \quad (89)$$

9 DG FEM in 2D

9.1 Mathematical Model

$$\frac{\partial U}{\partial t} + \nabla.cU = 0 \tag{90}$$

Boundary Conditions:
$U(0,y,t) = 1$ and $U(a,y,t) = 0$, where $0 \leq x \leq a$ for $t \geq 0$
$U(x,0,t) = 1$ and $U(x,b,t) = 0$, where $0 \leq y \leq b$ for $t \geq 0$
Initial Conditions:
$U(x,y,t=0) =$Known square wave.
Estimation of $U(t,x,y) \approx \bar{U}(t,x,y)$:

$$\bar{U}(t,x,y) = \sum_{j=1}^{4} U_j^t \phi_j(x,y) \tag{91}$$

Both sides are multiplied by a weighting function w and are integrated over the domain Ω.

$$\int_\Omega [\frac{\partial \bar{U}}{\partial t} w] d\Omega + \int_\Omega [\nabla.c\bar{U}w] d\Omega = 0 \tag{92}$$

Integration by part:

$$\int_\Omega [\frac{\partial \bar{U}}{\partial t} w] d\Omega + \oint_\Gamma \hat{n}.c\bar{U}w d\Gamma - \int_\Omega [c\bar{U}.\nabla w] d\Omega = 0 \tag{93}$$

The above equation at the elemental level becomes:

$$\int_{\Omega_e} [\frac{\partial \bar{U}}{\partial t} w] d\Omega_e + \oint_{\Gamma_e} \hat{n}.c\bar{U}w d\Gamma_e - \int_{\Omega_e} [c\bar{U}.\nabla w] d\Omega_e = 0 \tag{94}$$

Estimating the weighting function as:

$$w = \sum_{i=1}^{4} \phi_i(x,y) \tag{95}$$

$$\sum_{i=1}^{4} \int_{\Omega_e} [\frac{\partial \bar{U}}{\partial t} \phi_i] d\Omega_e = \sum_{i=1}^{4} \int_{\Omega_e} [c\bar{U}.\nabla \phi_i] d\Omega_e - \sum_{i=1}^{4} \oint_{\Gamma_e} [\hat{n}_x.c\bar{U}\phi_i + \hat{n}_y.c\bar{U}\phi_i] d\Gamma_e \tag{96}$$

Implementing (91) into (96):

$$\sum_{i=1}^{4}\sum_{j=1}^{4} \frac{\partial U_j^t}{\partial t} \int_{\Omega_e} [\phi_j \phi_i] d\Omega_e = \sum_{i=1}^{4}\sum_{j=1}^{4} cU_j^t \int_{\Omega_e} [\phi_j.\nabla \phi_i] d\Omega_e - \sum_{i=1}^{4}\sum_{j=1}^{4} cU_j^t \oint_{\Gamma_e} [\hat{n}_x.\phi_j\phi_i + \hat{n}_y.\phi_j\phi_i] d\Gamma_e \tag{97}$$

Now, Forward Euler's method is applied to $\frac{\partial U}{\partial t}$:

$$\frac{\partial U_j^t}{\partial x} = \frac{U^{t+1} - U^t}{\Delta t} \tag{98}$$

$$\sum_{i=1}^{4}\sum_{j=1}^{4} \frac{U_j^{t+1} - U_j^t}{\Delta t} \int_{\Omega_e} [\phi_j \phi_i] d\Omega_e = \sum_{i=1}^{4}\sum_{j=1}^{4} cU_j^t \int_{\Omega_e} [\phi_j.\nabla\phi_i] d\Omega_e - \sum_{i=1}^{4}\sum_{j=1}^{4} cU_j^t \oint_{\Gamma_e} [\hat{n}_x.\phi_j\phi_i + \hat{n}_y.\phi_j\phi_i] d\Gamma_e \tag{99}$$

$$\begin{aligned}\sum_{i=1}^{4}\sum_{j=1}^{4} \frac{U_j^{t+1}}{\Delta t} \int_{\Omega_e} [\phi_j \phi_i] d\Omega_e = \sum_{i=1}^{4}\sum_{j=1}^{4} \frac{U_j^t}{\Delta t} \int_{\Omega_e} [\phi_j \phi_i] d\Omega_e + \\ \sum_{i=1}^{4}\sum_{j=1}^{4} cU_j^t \int_{\Omega_e} [\phi_j.\nabla\phi_i] d\Omega_e - \sum_{i=1}^{4}\sum_{j=1}^{4} cU_j^t \oint_{\Gamma_e} [\hat{n}_x.\phi_j\phi_i + \hat{n}_y.\phi_j\phi_i] d\Gamma_e\end{aligned} \tag{100}$$

$$\begin{aligned}\sum_{i=1}^{4}\sum_{j=1}^{4} U_j^{t+1} \underbrace{\int_{\Omega_e} [\phi_j \phi_i] d\Omega_e}_{M} = \sum_{i=1}^{4}\sum_{j=1}^{4} U_j^t \int_{\Omega_e} [\phi_j \phi_i] d\Omega_e + \\ \sum_{i=1}^{4}\sum_{j=1}^{4} U_j^t \underbrace{c\Delta t \int_{\Omega_e} [\phi_j.\nabla\phi_i] d\Omega_e}_{K} - \sum_{i=1}^{4}\sum_{j=1}^{4} U_j^t \underbrace{c\Delta t \oint_{\Gamma_e} [\hat{n}_x.\phi_j\phi_i + \hat{n}_y.\phi_j\phi_i] d\Gamma_e}_{F}\end{aligned} \tag{101}$$

Representing above equation in the matrix form:

$$[M_{i,j}]\left\{U_j^{t+1}\right\} = [M_{i,j}]\left\{U_j^t\right\} + [K_{i,j}]\left\{U_j^t\right\} - [F_{i,j}]\left\{U_j^t\right\} \tag{102}$$

$$[M_{i,j}]\left\{U_j^{t+1}\right\} = [[M_{i,j}] + [K_{i,j}] - [F_{i,j}]]\left\{U_j^t\right\} \tag{103}$$

$$[M_{i,j}] = \sum_{i=1}^{4}\sum_{j=1}^{4} \int_{\Omega_e} [\phi_j \phi_i] d\Omega_e \tag{104}$$

$$[K_{i,j}] = \sum_{i=1}^{4}\sum_{j=1}^{4} c\Delta t \int_{\Omega_e} [\phi_j.\nabla\phi_i] d\Omega_e \tag{105}$$

$$[F_{i,j}] = \sum_{i=1}^{4}\sum_{j=1}^{4} \oint_{\Gamma_e} [\hat{n}_x c_x \Delta t_x.\phi_j\phi_i + \hat{n}_y c_y \Delta t_y.\phi_j\phi_i] d\Gamma_e \tag{106}$$

9.1.1 Interpolation functions ϕ_i at the local level:

For rectangular element, there are four nodes with the local coordinate system:

$$\begin{aligned}\text{node } 1 &= (x_1, y_1) = (0, 0)\\ \text{node } 2 &= (x_2, y_2) = (a, 0)\\ \text{node } 3 &= (x_3, y_3) = (a, b)\\ \text{node } 4 &= (x_4, y_4) = (0, b)\end{aligned}$$

Where a and b are the lengths of the element in x and y directions, respectively.
The flux direction for one element in 2D with the local coordinates is shown in Figure (15).

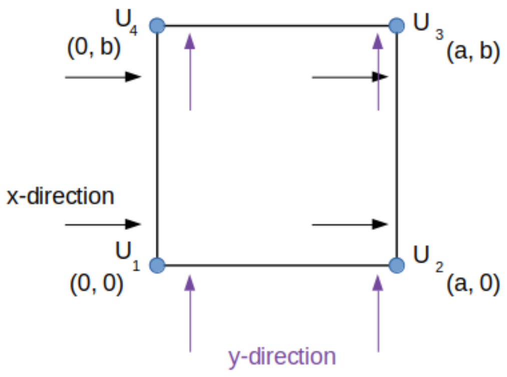

Figure 15: Fig:Flux direction for 1 element in 2D.

$$U^e(\bar{x}, \bar{y}) = C_1 + C_2\bar{x} + C_3\bar{y} + C_4\bar{x}\bar{y} = \sum_{j=1}^{4} U_j\phi_j(\bar{x}\bar{y}) \tag{107}$$

Where $\bar{x}$ and $\bar{y}$ are local coordinates.
at node 1: $U^e(0,0) = U_1 = C_1 \rightarrow C_1 = U_1$
at node 2: $U^e(a,0) = U_2 = C_1 + C_2 a \rightarrow C_2 = \frac{U_2 - U_1}{a}$
at node 3: $U^e(0,b) = U_3 = C_1 + C_3 b \rightarrow C_3 = \frac{U_4 - U_1}{b}$
at node 4: $U^e(a,b) = U_3 = C_1 + C_2 a + C_3 b + C_4 ab \rightarrow C_4 = \frac{(U_3 - U_4(+(U_1 - U_3)}{ab}$
Applying C_1, C_2, C_3 and C_4 into (107):

$$U^e(\bar{x}, \bar{y}) = U_1 \underbrace{(1 - \frac{\bar{x}}{a})(1 - \frac{\bar{y}}{b})}_{\phi_1} + U_2 \underbrace{\frac{\bar{x}}{a}(1 - \frac{\bar{y}}{b})}_{\phi_2} + U_3 \underbrace{\frac{\bar{x}\bar{y}}{ab}}_{\phi_3} + U_4 \underbrace{\frac{\bar{y}}{b}(1 - \frac{\bar{x}}{a})}_{\phi_4} \tag{108}$$

$$\phi_i^e(\bar{x}, \bar{y}) = (-1)^{i+1}[1 - \frac{\bar{x} + x_i}{a}][1 - \frac{\bar{y} + y_i}{b}] \tag{109}$$

$$\begin{cases} \frac{\partial\phi_1}{\partial\bar{x}} = \frac{\bar{y}-b}{ab} \\ \frac{\partial\phi_1}{\partial\bar{y}} = \frac{\bar{x}-a}{ab} \end{cases}, \begin{cases} \frac{\partial\phi_2}{\partial\bar{x}} = -\frac{\bar{y}-b}{ab} \\ \frac{\partial\phi_2}{\partial\bar{y}} = \frac{-\bar{x}}{ab} \end{cases}, \begin{cases} \frac{\partial\phi_3}{\partial\bar{x}} = \frac{\bar{y}}{ab} \\ \frac{\partial\phi_3}{\partial\bar{y}} = \frac{\bar{x}}{ab} \end{cases} \text{and} \begin{cases} \frac{\partial\phi_4}{\partial\bar{x}} = \frac{-\bar{y}}{ab} \\ \frac{\partial\phi_4}{\partial\bar{y}} = -\frac{\bar{x}-a}{ab} \end{cases}.$$

9.1.2 Mass matrix components:

Coordinate transformation:

If x_1 and y_1 are coordinates of node 1 of the element in the global system: $\begin{cases} x = \bar{x} + x_1^e \\ y = \bar{y} + y_1^e \\ dx = d\bar{x} \\ dy = d\bar{y} \end{cases}$

$$M_{i,j} = \int_0^a \int_0^b \phi_i \phi_j d\bar{x} d\bar{y} \tag{110}$$

Calculating for all i and j the matrix M in consistent form becomes:

$$M = \frac{ab}{36} \begin{bmatrix} 4 & 2 & 1 & 2 \\ 2 & 4 & 2 & 1 \\ 1 & 2 & 4 & 2 \\ 2 & 1 & 2 & 4 \end{bmatrix} \tag{111}$$

By using row sum method, matrix M becomes:

$$M = \begin{bmatrix} \frac{ab}{4} & 0 & 0 & 0 \\ 0 & \frac{ab}{4} & 0 & 0 \\ 0 & 0 & \frac{ab}{4} & 0 \\ 0 & 0 & 0 & \frac{ab}{4} \end{bmatrix} \tag{112}$$

9.1.3 Stiffness matrix components:

$$[K_{i,j}] = \sum_{i=1}^{4} \sum_{j=1}^{4} \int_0^a \int_0^b [c_x \Delta t \phi_j \frac{\partial \phi_i}{\partial \bar{x}} + c_y \Delta t \phi_j \frac{\partial \phi_i}{\partial \bar{y}}] d\bar{x} d\bar{y} \tag{113}$$

Calculating for all i and j:

$$K = \begin{bmatrix} \Delta t(-c_x \frac{b}{6} - c_y \frac{a}{6}) & \Delta t(-c_x \frac{b}{6} - c_y \frac{a}{12}) & \Delta t(-c_x \frac{b}{12} - c_y \frac{a}{12}) & \Delta t(-c_x \frac{b}{12} - c_y \frac{a}{6}) \\ \Delta t(c_x \frac{b}{6} - c_y \frac{a}{12}) & \Delta t(c_x \frac{b}{6} - c_y \frac{a}{6}) & \Delta t(c_x \frac{b}{12} - c_y \frac{a}{6}) & \Delta t(c_x \frac{b}{12} - c_y \frac{a}{12}) \\ \Delta t(c_x \frac{b}{12} + c_y \frac{a}{12}) & \Delta t(c_x \frac{b}{12} + c_y \frac{a}{6}) & \Delta t(c_x \frac{b}{6} + c_y \frac{a}{6}) & \Delta t(c_x \frac{b}{6} + c_y \frac{a}{12}) \\ \Delta t(-c_x \frac{b}{12} + c_y \frac{a}{6}) & \Delta t(-c_x \frac{b}{12} + c_y \frac{a}{12}) & \Delta t(-c_x \frac{b}{6} + c_y \frac{a}{12}) & \Delta t(-c_x \frac{b}{6} + c_y \frac{a}{6}) \end{bmatrix} \tag{114}$$

9.1.4 Flux matrix components:

$$
\begin{gathered}
[F_{i,j}] = \\
\Delta t \left[c(\bar{x}_2)\bar{U}^t(\bar{x}_2^-)\phi_j(\bar{x}_2)\phi_i(\bar{x}_2) - c(\bar{x}_1)\bar{U}^t(\bar{x}_1^-)\phi_j(\bar{x}_1)\phi_i(\bar{x}_1)\right] + \\
\Delta t \left[c(\bar{x}_3)\bar{U}^t(\bar{x}_3^-)\phi_j(\bar{x}_3)\phi_i(\bar{x}_3) - c(\bar{x}_4)\bar{U}^t(\bar{x}_4^-)\phi_j(\bar{x}_4)\phi_i(\bar{x}_4)\right] + \\
\Delta t \left[c(\bar{y}_4)\bar{U}^t(\bar{y}_4^-)\phi_j(\bar{y}_4)\phi_i(\bar{y}_4) - c(\bar{y}_1)\bar{U}^t(\bar{y}_1^-)\phi_j(\bar{y}_1)\phi_i(\bar{y}_1)\right] + \\
\Delta t \left[c(\bar{y}_3)\bar{U}^t(\bar{y}_3^-)\phi_j(\bar{y}_3)\phi_i(\bar{y}_3) - c(\bar{y}_2)\bar{U}^t(\bar{y}_2^-)\phi_j(\bar{y}_2)\phi_i(\bar{y}_2)\right]
\end{gathered} \tag{115}
$$

$$
F = \begin{bmatrix} -c_x\Delta t| - c_y\Delta t & 0|0 & 0|0 & 0|0 \\ 0|0 & c_x\Delta t| - c_y\Delta t & 0|0 & 0|0 \\ 0|0 & 0|0 & c_x\Delta t|c_y\Delta t & 0|0 \\ 0|0 & 0|0 & 0|0 & -c_x\Delta t|c_y\Delta t \end{bmatrix} \tag{116}
$$

We know that $U_i = U_i^x + U_i^y$ so:

$$
F = \begin{bmatrix} \Delta t(-c_x - c_y) & 0 & 0 & 0 \\ 0 & \Delta t(c_x - c_y) & 0 & 0 \\ 0 & 0 & \Delta t(c_x + c_y) & 0 \\ 0 & 0 & 0 & \Delta t(-c_x + c_y) \end{bmatrix} \tag{117}
$$

References

[1] DM Causon and CG Mingham. *Introductory finite difference methods for PDEs.* Bookboon, 2010.

[2] Randall J LeVeque et al. *Finite volume methods for hyperbolic problems*, volume 31. Cambridge university press, 2002.

[3] Olgierd C Zienkiewicz and Robert L Taylor. *The finite element method, ; volume 1: basic formulation and linear problems.* 1994.

A Python codes

A.1 Finite Difference Method in 1D, 2D and Finite Volume Method in 1D

In this section the Python codes for FDM studied in Section 3 and Section 4. The Python code for FVM in 1D is the same as FDM in 1D.

A.1.1 Finite Difference Method in 1D

This is the Python code for Finite Difference Method in 1D introduced in Section 3. It also refers to FVM studied in Section 5.

```
# This is the Python code relates to Finite Difference Method and also FVM in 1D.
import numpy as np
import matplotlib.pyplot as plt

C= 0.01                             # Courant number
nt = 1000                           # total number of time-steps
nx = 100                            # total number of displacement
L = 2                               # length of the domain in 1D
dx = L/(nx-1)                       # displacement length each time-step
c = .5                              # velocity of the wave
dt = C*dx/c                         # length of each time-step

U = np.zeros(nx)                    # initialising U matrix
U[25:100]=1                         # defining square wave
selected_elements=np.ones((3,nx)) # dummy vbl used for saving solutions in particular
    time-step

for n in range(nt):                 # marching time
    Un=U.copy()                     # dummy vbl to save current values of U
    for i in range(nx):             # marching in space
        U[i] = Un[i]-c*(dt/dx)*(Un[i]-Un[i-1])
    if n==1:                        # saving the solution at the beginning of the
    simulation
        selected_elements[0,:] = U.copy()
    if n==15:                       # saving the solution at the middle of the simulation
        selected_elements[1,:] = U.copy()
    if n==30:                       # saving the solution at the end of the simulation
        selected_elements[2,:] = U.copy()

# ------------------------------ Plotting the results ------------------------------
plt.figure(1)
plt.plot(np.linspace(0,L,nx), selected_elements[0,:])
plt.plot(np.linspace(0,L,nx), selected_elements[1,:])
plt.plot(np.linspace(0,L,nx), selected_elements[2,:])
plt.xlabel('Distrance')
plt.ylabel('U')
```

A.1.2 Finite Difference Method in 2D

This is the Python code for Finite Difference Method in 2D introduced in Section 4.

```
# This is the Python code for Finite Difference Method in 2D
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

#------------------------------------ 2-D FDM------------------------------------
C= .01                          # CLF number
nt = 5000                       # Number of time steps
nx = 100                        # Number of steps in x-direction
ny = 200                        # Number of steps in y-direction
L = 2                           # Lengths of the domain in x and y directions
dx = L/(nx-1)                   # displacement length at each time-step in x-
    direction
dy = L/(ny-1)                   # displacement length at each time-step in y-
    direction
d_displacement = (dx**2+dy**2)**0.5 # overall displacement of the wave in both
    directions
x=np.linspace(0, 2, nx)
y=np.linspace(0,2,ny)
c_x = 0.1                       # Wave velocity in x-dir
c_y = 0.1                       # Wave velocity in y-di
c = (c_x**2+c_y**2)**0.5           # overall velocity of the wave
dt = C*d_displacement/c         # length of time step.
U = np.zeros((ny,nx))           # Wave matrix
Un = np.zeros((ny,nx))          # Dummy variable to save current components of U
U1=U2=U3 = U                    # Dummy matrices to plot 3 time steps

U[int(ny*.1):int(ny*.3), int(nx*0.1):int(nx*0.3)]=1     # Defining wave components
X, Y =np.meshgrid(x,y)          # Creating a mesh grid

#------------------------------------ Initial Conds ------------------------------
U[:, 0]= U[:, -1] = 0
U[0,:] = U[-1,:] = 0

#--------------------------------------------------------------------------------
for n in range(nt+1):             # marching in time steps
    Un=U.copy()                   # dummy vbl to save current solutions
    U[1:,1:] = Un[1:,1:] - c_y*(dt/dy)*(Un[1:,1:] - Un[:-1,1:]) - c_x*(dt/dx)*(Un
    [1:,1:] - Un[1:, :-1])

    if n==0:                     # saving the first time step to plot
        U1 = Un
    elif n==int(nt/2):           # saving the half time step to plot
        U2 = Un
    elif n==int(nt*0.99):        # saving the final time step to plot
        U3 = Un
```

```
#------------------------------------ Plotting the results ------------------------------------
plt.figure()
ax = plt.gca(projection='3d')
ax.plot_surface(X, Y, U1 , label='t=0')
ax.plot_surface(X, Y, U2, label='t=nt/2')
ax.plot_surface(X, Y, U3, label='t=final')
ax.set_ylabel('$y$')
ax.set_xlabel('$x$')
ax.set_zlabel('$U$')
plt.legend()
plt.show()

fig = plt.figure()
ax = fig.gca(projection='3d')
surf = ax.plot_surface(X, Y, U[:], cmap=plt.cm.viridis)
#ax.plot(x, y, U, label='Square Wave')
plt.show()
```

A.2 Continuous-Galerkin FEM

In this section the Python codes for CG-FEM are provided, which were used to simulate the governing Equation in Section 6 and represent the Explicit scheme with the consistent mass matrices.

```
#-----------------------------------------1D-DG FEM Implicit--------------------------------------------------
# This is the Python code for the Discontinuous-Galerkin FEM. This code uses the
    Implicit scheme with the consistent  mass matrix for simulation.
# It refers to the mathematical formulations in the note in Section 8.1.
import numpy as np
import matplotlib.pyplot as plt
import sympy as sy
import time

nx = 1000                   # total number of nodes(degree of freedom)
nt = 1000                   # total number of time steps
N_i = 2                     # number of interpolation functions
L =  0.5                    # Total length
C = .05                     # Courant number
c = .1                      # Wave velocity
dx = L/(nx-1)               # Distance stepping size
dt = C*dx/c                 # Time stepping size
x  = np.arange(0, nx)*dx    # or x=np.linspace(0,2,nx)
U = np.zeros(nx)            # U is a square wave between 0 <U< 1
U_plot = np.ones((3,nx))    # A matrix to save 3 time steps used for plotting the
    results

#-------------------------------------------------------------------------------------------------------------
# Boundary Conditions
U[0] = U[nx-1] = 0          # Dirichlet BC

#-------------------------------------------------------------------------------------------------------------
# Initial conditions
U[int(L*nx*0.2):int(L*nx*0.5)]=1

#--------------------------------------- Interpolation functions ---------------------------------------------
x_bar=sy.Symbol('x_bar')                       # defining local x symbol for creating
    interpolation functions and their integrations
phi_1= 1-x_bar/dx                              # interpolation function 1
phi_2 = x_bar/dx                               # interpolation function 2

A = np.array(([phi_1], [phi_2]))               # Creating matrix A(1,2) for the basis
    of constructing Mass and stiffness matrices
A=np.transpose(A)
B = np.array(([phi_1], [phi_2]))               # Creating matrix B(2,1) for the basis
    of constructing Mass and stiffness matrices
A_by_B=A*B                                     # putting both interpolation functions
    into a matrix (2 by 2)

#-----------------------------------------Mass Matrix 'M' in Equation 55 ------------------------------------
t1 = time.time()
```

```
sub_M=np.zeros((N_i,N_i))                    # starting for timing the M_diag_inv calculation
for i in range(N_i):
    for j in range(N_i):
        sub_M[i,j]= sy.integrate(A_by_B[i,j], (x_bar,0,dx))
#sub_M = np.array([[dx/3,dx/6],[dx/6,dx/3]]) # local mass matrix
M=np.zeros((nx,nx))                          # generating global mass matrix
i=0
while i<nx:
    M[i:i+2, i:i+2]= sub_M[0:2,0:2]
    i+=2
t2 = time.time()                             # end point of M_diag_inv generation
print(str(t2-t1))

#-------------------------------Stifness Matrix 'K' in Equation 55 ---------------------
sub_K=np.array([[-c*dt/2,-c*dt/2],[c*dt/2,c*dt/2]]) # local stiffness matrix
K = np.zeros((nx,nx))                               # generating global stiffness matrix
i=0
while i<nx:
    K[i:i+2, i:i+2]= sub_K[0:2,0:2]
    i+=2

#------------------------------------Flux in Equation 55------------------------------------
F = np.zeros((nx,nx+1))
i=0
j=1
while i<=nx-1:
    F[i,i]= c*dt
    F[j,j+1] = -c*dt
    i+=2
    j+=2
F=F[:,1:]                        # excluding left boundary to get nx by nx matrix

#------------------------------------RHS Constant in Equation 55----------------------------
RHS_cst = (M + K + F)

##-Matrix method ----------------------------------------------------------------------------
##Mrching forward in time
Un=np.zeros(nx)                  # dummy vbl to save current values of U (U^t)
t3 = time.time()
for n in range(nt):              # Marching in time
    Un = U.copy()
    RHS = RHS_cst.dot(Un)        # saving U^t to be used in the next time step calculation
    U=np.linalg.solve(M,RHS)

    if n==1:
        U_plot[0,:] = U.copy()   # saving U(t=1)
    if n==int(nt/2):
        U_plot[1,:] = U.copy()      # saving U(t=nt/2)
    if n==int(nt*0.99):
        U_plot[2,:] = U.copy()      # saving U(t= almost the end to time steps)
t4 = time.time()
#------------------------------plot initiation --------------------------------------
plt.figure(1)
plt.axis([0,L, -1,2])
plt.plot(x, U_plot[0,:], label=Time step 1')
plt.plot(x, U_plot[1,:], label=Time step 0.5*nx')
plt.plot(x, U_plot[2,:], label=Time step 0.9*nx')
plt.xlabel('Distance')
plt.ylabel('U')
plt.legend()
plt.title(f'Simulation Duration: {round((t4-t3)/60, 2)} minutes')
```

A.3 Discontinuous-Galerkin FEM

In this section, three sets of Python codes are provided which were used to generate simulate the governing Equation 43, but represent the Implicit or the Explicit scheme with the lumped or consistent mass matrices.

A.3.1 Implicit method with consistent mass matrix

This is the Python code relate to Section 7.1 and Section 7.2. It uses the Implicit scheme and consistent mass matrix for the simulation.

```
#--------------------------------1D-DG FEM Implicit--------------------------------
# This is the Python code for the Discontinuous-Galerkin FEM. This code uses the
    Implicit scheme with the consistent  mass matrix for simulation.
# It refers to the mathematical formulations in the note in Section 8.1.
import numpy as np
import matplotlib.pyplot as plt
import sympy as sy
import time

nx = 1000                   # total number of nodes(degree of freedom)
nt = 1000                   # total number of time steps
N_i = 2                     # number of interpolation functions
L =  0.5                    # Total length
C = .05                     # Courant number
c = .1                      # Wave velocity
dx = L/(nx-1)               # Distance stepping size
dt = C*dx/c                 # Time stepping size
x  = np.arange(0, nx)*dx    # or x=np.linspace(0,2,nx)
U = np.zeros(nx)            # U is a square wave between 0 <U< 1
U_plot = np.ones((3,nx))    # A matrix to save 3 time steps used for plotting the
    results

#----------------------------------------------------------------------------------
# Boundary Conditions
U[0] = U[nx-1] = 0          # Dirichlet BC

#----------------------------------------------------------------------------------
# Initial conditions
U[int(L*nx*0.2):int(L*nx*0.5)]=1

#------------------------------ Interpolation functions ---------------------------
x_bar=sy.Symbol('x_bar')                    # defining local x symbol for creating
    interpolation functions and their integrations
phi_1= 1-x_bar/dx                           # interpolation function 1
phi_2 = x_bar/dx                            # interpolation function 2

A = np.array(([phi_1], [phi_2]))            # Creating matrix A(1,2) for the basis
    of constructing Mass and stiffness matrices
A=np.transpose(A)
```

```
B = np.array(([phi_1], [phi_2]))               # Creating matrix B(2,1) for the basis
    of constructing Mass and stiffness matrices
A_by_B=A*B                                     # putting both interpolation functions
    into a matrix (2 by 2)

#--------------------------------------Mass Matrix 'M' in Equation 55 ----------------------
t1 = time.time()
sub_M=np.zeros((N_i,N_i))                      # starting for timing the M_diag_inv
    calculation
for i in range(N_i):
    for j in range(N_i):
        sub_M[i,j]= sy.integrate(A_by_B[i,j], (x_bar,0,dx))
#sub_M = np.array([[dx/3,dx/6],[dx/6,dx/3]]) # local mass matrix
M=np.zeros((nx,nx))                            # generating global mass matrix
i=0
while i<nx:
    M[i:i+2, i:i+2]= sub_M[0:2,0:2]
    i+=2
t2 = time.time()                              # end point of M_diag_inv generation
print(str(t2-t1))

#---------------------------------Stifness Matrix 'K' in Equation 55 ----------------------
sub_K=np.array([[-c*dt/2,-c*dt/2],[c*dt/2,c*dt/2]]) # local stiffness matrix
K = np.zeros((nx,nx))                                  # generating global stiffness
    matrix
i=0
while i<nx:
    K[i:i+2, i:i+2]= sub_K[0:2,0:2]
    i+=2

#--------------------------------------Flux in Equation 55----------------------------------
F = np.zeros((nx,nx+1))
i=0
j=1
while i<=nx-1:
    F[i,i]= c*dt
    F[j,j+1] = -c*dt
    i+=2
    j+=2
F=F[:,1:]                             # excluding left boundary to get nx by nx matrix

#--------------------------------------RHS Constant in Equation 55--------------------------
RHS_cst = (M + K + F)

##-Matrix method
    ------------------------------------------------------------------------------------
##Mrching forward in time
Un=np.zeros(nx)                        # dummy vbl to save current values of U (U^t)
t3 = time.time()
for n in range(nt):                    # Marching in time
```

```
        Un = U.copy()
        RHS = RHS_cst.dot(Un)             # saving U^t to be used in the next time step calculation
        U=np.linalg.solve(M,RHS)

        if n==1:
            U_plot[0,:] = U.copy()        # saving U(t=1)
        if n==int(nt/2):
            U_plot[1,:] = U.copy()        # saving U(t=nt/2)
        if n==int(nt*0.99):
            U_plot[2,:] = U.copy()        # saving U(t= almost the end to time steps)
t4 = time.time()
#---------------------------------plot initiation ------------------------------------------
plt.figure(1)
plt.axis([0,L, -1,2])
plt.plot(x, U_plot[0,:], label=Time step 1')
plt.plot(x, U_plot[1,:], label=Time step 0.5*nx')
plt.plot(x, U_plot[2,:], label=Time step 0.9*nx')
plt.xlabel('Distance')
plt.ylabel('U')
plt.legend()
plt.title(f'Simulation Duration: {round((t4-t3)/60, 2)} minutes')
```

A.3.2 Explicit method with the consistent mass matrix

This is the Python code relate to Section 7.1 and Section 7.2. It uses the Explicit scheme and consistent mass matrix for the simulation.

```
#---------------------------------------1D-DG FEM---------------------------------------
# This is the Python code for the standard Discontinuous-Galerkin FEM. This code uses
    the Explicit scheme with the consistent mass matrix for simulation.
# This Python code relates to the mathematical formulations in Section 8.2 in this
    report.
import numpy as np
import matplotlib.pyplot as plt
import time

nx = 1000                   # total number of nodes(degree of freedom)
nt = 2000                   # total number of time steps
L =  0.5                    # Total length
C = 0.05                    # Courant number
c = 0.1                     # Wave velocity
dx = L/(nx-1)               # Distance stepping size
dt = C*dx/c                 # Time stepping size
x  = np.arange(0, nx)*dx    # or x=np.linspace(0,2,nx)
U = np.zeros(nx)            # U is a square wave between 0 <U< 1
U_plot = np.ones((3,nx))    # A matrix to save 3 time steps used for plotting the
    results

#---------------------------------------------------------------------------------------
# Boundary Conditions
U[0] = U[nx-1] = 0          # Dirichlet BC

#---------------------------------------------------------------------------------------
# Initial conditions
U[int(L*nx*0.2):int(L*nx*0.5)]=1    # defining square wave shape

#---------------------------Mass Matrix 'M' in Equation 72 ------------------------------
t1 = time.time()                      # starting for timing the M_diag_inv calculation
sub_M = np.array([[dx/3,dx/6],[dx/6,dx/3]]) # local mass matrix
M=np.zeros((nx-2,nx-2))                         # generating global mass matrix
i=0
while i<nx-2:
    M[i:i+2, i:i+2]= sub_M[0:2,0:2]
    i+=2
M_inv=np.linalg.inv(M)
t2 = time.time()                              # end point of M_diag_inv generation
print(str(t2-t1))
#---------------------------Stifness Matrix 'K' in Equation 72---------------------------
sub_K=np.array([[-c*dt/2,-c*dt/2],[c*dt/2,c*dt/2]]) # local stiffness matrix
K = np.zeros((nx-2,nx-2))                                  # generating global stiffness
    matrix
i=0
while i<nx-2:
    K[i:i+2, i:i+2]= sub_K[0:2,0:2]
    i+=2

# #------------------------------------------------Flux matrix in Equation 72 ----------------------
F = np.zeros((nx-2,nx-1))
i=0
j=1
while i<=nx-3:
    F[i,i]= c*dt
    F[j,j+1] = -c*dt
    i+=2
    j+=2
F=F[:,1:]                              # excluding left boundary to get nx by nx matrix

#------------------------------------------------RHS in equation 72 ----------------------
RHS_cst = M_inv.dot((M + K + F))

#---------------------------------------------Explicit using consistence mass matrix----------------
Un=np.zeros(nx)                        # dummy vbl to save current values of U (U^t)
for n in range(nt):                    # Marching in time
    Un = U.copy()                      # saving U^t to be used in the next time step
    calculation
    U[1] = RHS_cst[0,0]*Un[1] + RHS_cst[0,1]*Un[2]
    U[2] = RHS_cst[1,0]*Un[1] + RHS_cst[1,1]*Un[2]
    i=3
    j=1
    while i<nx-2:
        U[i] = RHS_cst[i-1,j]*Un[i-1] + RHS_cst[i-1,j+1]*Un[i] + RHS_cst[i-1,j+2]*Un[
    i+1]
        i+=1
        U[i] = RHS_cst[i-1,j]*Un[i-2] + RHS_cst[i-1,j+1]*Un[i-1] + RHS_cst[i-1,j+2]*
    Un[i]
        i+=1
        j+=2
    if n==1:
        U_plot[0,:] = U.copy()         # saving U(t=1)
    if n==int(nt/2):
        U_plot[1,:] = U.copy()         # saving U(t=nt/2)
    if n==int(nt*0.99):
        U_plot[2,:] = U.copy()         # saving U(t= almost the end to time steps)

#-----------------------------------------------plot initiation ----------------------
plt.figure(1)
plt.axis([0,L, -1,2])
plt.plot(x, U_plot[0,:], label=Time step 1')
plt.plot(x, U_plot[1,:], label=Time step 0.5*nx')
plt.plot(x, U_plot[2,:], label=Time step 0.9*nx')
plt.xlabel('Distance')
plt.ylabel('U')
plt.legend()
```

A.3.3 Explicit method with the lumped mass matrix

The Python code relates to Section 7.1 and Section 7.3. It uses the Explicit scheme and consistent mass matrix for the simulation.

```
#----------------------------------------------------------1D-DG FEM----------------------------------------------------------
# This is the Python code for the standard Discontinuous-Galerkin FEM. This code uses
    the Explicit scheme with the lumped mass matrix for simulation.
# It refers to the mathematical formulations in the note in Section 8.3.

import numpy as np
import matplotlib.pyplot as plt
import time

nx = 1000                      # total number of nodes(degree of freedom)
nt = 2000                      # total number of time steps
L =  0.5                       # Total length
C = .05                        # Courant number
c = .1                         # Wave velocity
dx = L/(nx-1)                  # Distance stepping size
dt = C*dx/c                    # Time stepping size
x  = np.arange(0, nx)*dx       # or x=np.linspace(0,2,nx)
U = np.zeros(nx)               # U is a square wave between 0 <U< 1
U_plot = np.ones((3,nx))       # A matrix to save 3 time steps used for plotting the
    results

#-----------------------------------------------------------------------------------------------------------------------------
# Boundary Conditions
U[0] = U[nx-1] = 0                      # Dirichlet BC

#-----------------------------------------------------------------------------------------------------------------------------
# Initial conditions
U[int(L*nx*0.2):int(L*nx*0.5)]=1        # defining square wave shape

#----------------------------------------------------Explicit method----------------------------------------------------------
Un=np.zeros(nx)                         # dummy vbl to save current values of U (U^t)
for n in range(nt):                     # Marching in time
    Un = U.copy()                       # saving U^t to be used in the next time step
    calculation
    U[1] = Un[1] - c* dt/dx *(Un[1]+Un[2])
    i=2
    while i<nx-1:
        U[i] = Un[i] + (-1)**i*c*dt/dx*(Un[i] + Un[i-1]) + (-1)**(i+1)*c*Un[i]
        i +=1
        if i==nx-1:
            StopIteration
        else:
            U[i] = Un[i] + (-1)**i*c*dt/dx*(Un[i] + Un[i+1]) + (-1)**(i+1)*c*Un[i-1]
        i +=1

    if n==1:
        U_plot[0,:] = U.copy()      # saving U(t=1)
    if n==int(nt/2):
        U_plot[1,:] = U.copy()      # saving U(t=nt/2)
    if n==int(nt*0.99):
        U_plot[2,:] = U.copy()      # saving U(t= almost the end to time steps)

#------------------------------plot initiation ----------------------------------------
plt.figure(1)
plt.axis([0,L, -1,2])
plt.plot(x, U_plot[0,:], label=Time step 1')
plt.plot(x, U_plot[1,:], label=Time step 0.5*nx')
plt.plot(x, U_plot[2,:], label=Time step 0.9*nx')
plt.xlabel(Distancee')
plt.ylabel('U')
plt.legend()
```