Technical University of Catalonia

FINITE ELEMENTS IN FLUIDS

ASSIGNMENT 1

STEADY/UNSTEADY

CONVECTION DIFFUSION REACTION TRANSPORT

M.Sc. Computational Mechanics – CIMNE

Mohammad Mohsen Zadehkamand

22 May 2017

Table of Contents

1. Introduction

This is the report for Assignment_1 of the course "Finite Element in Fluids" which deals with *Steady-Unsteady convection diffusion reaction transport problems in 2D*. In this project, using the supplied MATLAB codes for the *steady convection diffusion (elliptic)* and *unsteady convection (hyperbolic)* problems plus doing modification and changes, the structure underlying the *steady-unsteady convection diffusion reaction (parabolic)* problems are studied in 2D. **Chart1** introduces the different differential equations which are going to be solved by the comprehensive program provided in this project.

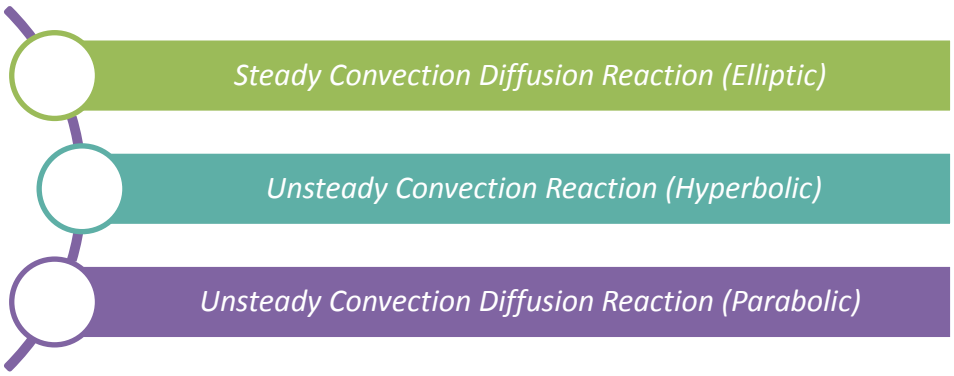


Chart1. Problem Types.

For evaluating the correctness of implementations first we use ample examples provided in the main reference book of this course "***Finite Element Methods for Flow Problems***" by *Jean Donea and Antonio Huerta* which is going to be called the ***"reference book"*** in the rest of this report, and solve each chapter's examples and provide their results in the ***Annex1*** and ***Annex2*** and compare them to the reference.

Following in **chapter2** the assignment is stated from theoretical background, beside time and space discretization and the matrix form of the discretized weak form are presented.

Chapter3 is related to the code explanation and changes which is done on the code in order to take into account different methods in 2D parabolic and elliptic problems. However the detailed code changes are discussed line by line in ***Annex3***.

As the last goal, finally we try to capture and present the results of the problems provided in the assignment for unsteady case in **chapter4,** and for the steady case in **chapter5.**

2. Formulation

Considering the transient convection diffusion reaction problem with the unknown "u", the convective term "$\boldsymbol{a}$ ", the diffusivity term "ν" , the reaction term "σ" and source term "s " we would have:

$$u_t + \boldsymbol{a}.\boldsymbol{\nabla}u - \boldsymbol{\nabla}.(\nu\boldsymbol{\nabla}u) + \sigma u = s$$

This is called the "***strong form***" equation and in this part we would derive the "***weak form***" equation of the problem and write down the system obtained after discretizing the weak form and provide comprehensive matrix and vectorial forms of the resulting systems for each proposed method in the assignment.

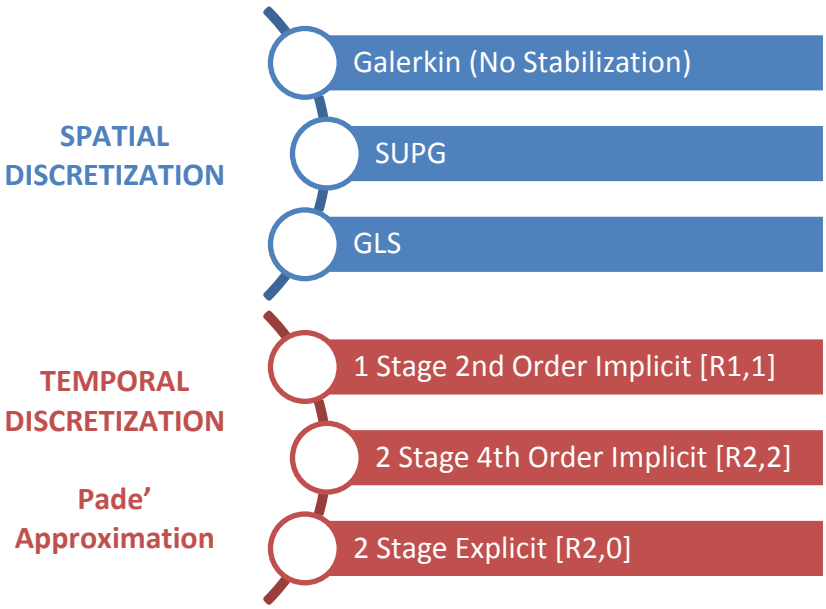


Chart2. Discretization Methods.

In order to do this we would combine spatial discretization techniques and stabilization methods for steady convection diffusion problems which are discussed in the reference book, chapter 2 and then would study the numerical schemes for time integration presented in the reference book, chapter 3. The objective is to produce time accurate finite element methods for unsteady problems describing transport including convection, diffusion and reaction effects. ***Chart2*** describes the methods which are used in this assignment to solve the problem. The detailed formulations for the so called methods are provided, in details, in the reference book and in the rest of this chapter main lines would be mentioned.

2.1. Time Discretization

2.1.1. ϑ family methods (Crank Nicolson ϑ=0.5)

As a single step method this routine is widely used based on the following formulation:

$$\frac{\Delta u}{\Delta t} - \theta\Delta u_t = u_t^n$$

Replacing the relation for time derivative from the original strong form equation we would have:

$$\frac{\Delta u}{\Delta t} - \theta[\boldsymbol{a}.\boldsymbol{\nabla} - \boldsymbol{\nabla}.(\nu\boldsymbol{\nabla}) + \sigma]\Delta u = \theta S^{n+1} + (1-\theta)S^n - [\boldsymbol{a}.\boldsymbol{\nabla} - \boldsymbol{\nabla}.(\nu\boldsymbol{\nabla}) + \sigma]u^n$$

$$for\ \theta = 0.5: \frac{\Delta u}{\Delta t} - 0.5[\boldsymbol{a}.\boldsymbol{\nabla} - \boldsymbol{\nabla}.(\nu\boldsymbol{\nabla}) + \sigma]\Delta u = 0.5[S^{n+1} + S^n] - [\boldsymbol{a}.\boldsymbol{\nabla} - \boldsymbol{\nabla}.(\nu\boldsymbol{\nabla}) + \sigma]u^n$$

2.1.2. Implicit Pade' methods

Multistage schemes can be derived directly from the Pade' approximations in the following form:

$$\frac{\Delta \boldsymbol{u}}{\Delta t} - \boldsymbol{W}\Delta \boldsymbol{u}_t = \boldsymbol{w}u_t^n$$

Where the unknown $\Delta \boldsymbol{u}$ is a vector with dimension related to number of stages in the method, namely 1 for the [R1,1] and 2 for the [R2,2]. On the other hand, $\Delta \boldsymbol{u}_t$ is the partial derivative of the $\Delta \boldsymbol{u}$ with respect to time. Here we can substitute the time derivative from the strong form equation and in this case we would have only space derivatives in the equation and it can be reinterpreted as a system of steady convection-diffusion-reaction equations.

$$u_t + \boldsymbol{a}.\boldsymbol{\nabla}u - \boldsymbol{\nabla}.(\nu\boldsymbol{\nabla}u) + \sigma u = s$$

$$u_t + \mathcal{L}(u) = s \qquad \mathcal{L} = \boldsymbol{a}.\boldsymbol{\nabla} - \boldsymbol{\nabla}.(\nu\boldsymbol{\nabla}) + \sigma$$

$$\frac{\Delta \boldsymbol{u}}{\Delta t} + \boldsymbol{W}\boldsymbol{\mathcal{L}}(\Delta \boldsymbol{u}) = \boldsymbol{w}[s^n - \mathcal{L}(u^n)] + \boldsymbol{W}\Delta \boldsymbol{s}$$

Method	*R1,1*	*R2,2*
$\Delta \boldsymbol{u}$	$u^{n+1} - u^n$	$\begin{Bmatrix} u^{n+1/2} - u^n \\ u^{n+1} - u^{n+1/2} \end{Bmatrix}$
$\Delta \boldsymbol{s}$	$s^{n+1} - s^n$	$\begin{Bmatrix} s^{n+1/2} - s^n \\ s^{n+1} - s^{n+1/2} \end{Bmatrix}$
$\boldsymbol{W}$	$\frac{1}{2}$	$\frac{1}{24}\begin{pmatrix} 7 & -1 \\ 13 & 5 \end{pmatrix}$
$\boldsymbol{w}$	1	$\frac{1}{2}\begin{Bmatrix} 1 \\ 1 \end{Bmatrix}$

As it is clear the 2nd order Pade' approximation method [R1,1] is exactly fit to the famous so called Crank-Nicolson method

2.1.3. Explicit Pade' method

For the 2nd order Pade' approximation [R2,0] a 2 stage method can be derived from the factorization below which yields to the 2nd order explicit Lax-Wendroff method:

$$u(t^{n+1}) = u(t^n) + \Delta t \frac{\partial}{\partial t}\left(u + \frac{\Delta t}{2}\frac{\partial u}{\partial t}\right) + O(\Delta t^3)$$

$$u^{n+\frac{1}{2}} = u^n + \frac{\Delta t}{2}u_t^n \qquad u^{n+1} = u^n + \Delta t\, u_t^{n+1/2}$$

And finally, the time derivative is replaced using the strong form of the unsteady convection-diffusion-reaction equation and the result would be the semi-discrete system which is the basis for the finite element spatial discretization.

2.2. Spatial Discretization

2.2.1. General Idea

The first step for the finite element spatial discretization of the model problem consists of formulating a weak form of the boundary value problem. This is achieved by multiplying the governing strong form equation by the weighting function "ω" and integrating over the computational domain:

$$(\omega, u_t) + (\omega, \boldsymbol{a}.\boldsymbol{\nabla} u) - (\omega, \boldsymbol{\nabla}.(\nu \boldsymbol{\nabla} u)) + (\omega, \sigma u) = (\omega, s)$$

Based on the divergence theorem and integration by parts we can change the 2nd order derivative term to a 1st order term plus a boundary term (weakening of the equation), and so get rid of the second order derivatives as following:

$$-\big(\omega, \boldsymbol{\nabla}.(\nu \boldsymbol{\nabla} u)\big) = -\int_{\Omega} \omega \boldsymbol{\nabla}.(\nu \boldsymbol{\nabla} u) d\Omega = \int_{\Omega} \boldsymbol{\nabla}\omega.(\nu \boldsymbol{\nabla} u) d\Omega - \int_{\Gamma} \omega.\Delta h \, d\Gamma$$

Replacing this integrated by part term the whole weak form of equation would be as:

$$(\omega, u_t) + \boldsymbol{c}(\boldsymbol{a}; \omega, u) + \boldsymbol{a}(\omega, u) + (\omega, \sigma u) = (\omega, s) + (\omega, h)_{\Gamma N}$$

And finally the formal matrix assembly procedure which yields to the system of 1st order differential equations which are going to be used in the computer code is:

$$\boldsymbol{M}\dot{\boldsymbol{u}} + (\boldsymbol{C} + \boldsymbol{K} + \sigma \boldsymbol{M})\boldsymbol{u} = \boldsymbol{f}$$

$$\boldsymbol{a}(\omega, u) = \int_{\Omega} \boldsymbol{\nabla}\omega.(\nu \boldsymbol{\nabla} u) d\Omega \qquad K^e_{ab} = \int_{\Omega e} \boldsymbol{\nabla} N_a.(\nu \boldsymbol{\nabla} N_b) \, d\Omega$$

$$\boldsymbol{c}(\boldsymbol{a}; w, u) = \int_{\Omega} \omega.(\boldsymbol{a}.\boldsymbol{\nabla} u) d\Omega \qquad C^e_{ab} = \int_{\Omega e} N_a.(\boldsymbol{a}.\boldsymbol{\nabla} N_b) \, d\Omega$$

$$(\omega, \sigma u) = \int_{\Omega} \omega \, \sigma u \, d\Omega \qquad M^e_{ab} = \int_{\Omega e} N_a.N_b \, d\Omega$$

$$(\omega, s) = \int_{\Omega} \omega \, s \, d\Omega \qquad \int_{\Omega e} N_a \, s \, d\Omega$$

$$(\omega, h)_{\Gamma N} = \int_{\Gamma N} \omega \, \mathrm{h} \, \mathrm{d}\Gamma \qquad \int_{\Gamma N} N_a \, h \, d\Gamma$$

2.2.2. Galerkin formulation for Explicit Pade' scheme

It is trivial that the Galerkin formulation of multistage explicit Pade' method is:

$$\left(\omega, u^{n+\frac{1}{2}}\right) = (\omega, u^n) + \frac{\Delta t}{2} u_t^n$$
$$= (\omega, u^n) + \frac{\Delta t}{2}[-\boldsymbol{c}(\boldsymbol{a}; \omega, u^n) - \boldsymbol{a}(\omega, u^n) - (\omega, \sigma u^n) + (\omega, s^n) + (\omega, h^n)_{\Gamma N}]$$

$$(\omega, u^{n+1}) = (\omega, u^n) + \Delta t\, u_t^{n+\frac{1}{2}}$$
$$= (\omega, u^n)$$
$$+ \Delta t\left[-\boldsymbol{c}\left(\boldsymbol{a}; \omega, u^{n+\frac{1}{2}}\right) - \boldsymbol{a}\left(\omega, u^{n+\frac{1}{2}}\right) - \left(\omega, \sigma u^{n+\frac{1}{2}}\right) + \left(\omega, s^{n+\frac{1}{2}}\right) + \left(\omega, h^{n+\frac{1}{2}}\right)_{\Gamma N}\right]$$

Finally the matrix form of time-space discretized weak form is:

$$\boldsymbol{M}\,\Delta\boldsymbol{u} = -\Delta t\,[(\boldsymbol{C} + \boldsymbol{K} + \sigma\boldsymbol{M})\boldsymbol{u} + \boldsymbol{f}\,]$$

2.2.3. Galerkin formulation for Implicit Pade' scheme

In part 2.1.2 the time discretized strong form of the problem equation was provided. This equation shall be solved in each time step. But from spatial point of view, the weighted residual form of these methods would be:

$$\left(\omega, \frac{\Delta \boldsymbol{u}}{\Delta t}\right) - (\omega, \boldsymbol{W}\Delta\boldsymbol{u}_t) = (\omega, \boldsymbol{w}u_t^n)$$

Application of integration by part in the case of linear operator with constant coefficients yields to:

$$\left(\omega, \frac{\Delta \boldsymbol{u}}{\Delta t}\right) + \boldsymbol{c}(\boldsymbol{a}; \omega, W\Delta u) + \boldsymbol{a}(\omega, W\Delta u) + (\omega, \sigma W\Delta u) =$$

$$= (\omega, ws^n + W\Delta s) + (\omega, h^n + W\Delta h)_{\Gamma N} - [\boldsymbol{c}(\boldsymbol{a}; \omega, wu^n) + \boldsymbol{a}(\omega, wu^n) + (\omega, \sigma wu^n)]$$

Finally the matrix form of time-space discretized weak form for [R1,1] is:

$$(\boldsymbol{M} + 0.5\Delta t\boldsymbol{J})(\boldsymbol{u}^{n+1} - \boldsymbol{u}^n) = \Delta t[\boldsymbol{s}^n - \boldsymbol{J}(\boldsymbol{u}^n) + 0.5(\boldsymbol{s}^{n+1} - \boldsymbol{s}^n)]$$

$$\boldsymbol{J} = \boldsymbol{C} + \boldsymbol{K} + \sigma\boldsymbol{M}$$

And the same procedure for [R2,2], yields to:

$$\begin{bmatrix} \boldsymbol{M} + W_{11}\Delta t\boldsymbol{J} & W_{12}\Delta t\boldsymbol{J} \\ W_{21}\Delta t\boldsymbol{J} & \boldsymbol{M} + W_{22}\Delta t\boldsymbol{J} \end{bmatrix} \begin{Bmatrix} \boldsymbol{u}^{n+\frac{1}{2}} - \boldsymbol{u}^n \\ \boldsymbol{u}^{n+1} - \boldsymbol{u}^{n+\frac{1}{2}} \end{Bmatrix} =$$

$$= \begin{bmatrix} -0.5\Delta t\boldsymbol{J} \\ -0.5\Delta t\boldsymbol{J} \end{bmatrix} \begin{Bmatrix} \boldsymbol{u}^n \\ \boldsymbol{u}^n \end{Bmatrix} + \begin{bmatrix} W_{11}\Delta t\boldsymbol{f} & W_{12}\Delta t\boldsymbol{f} \\ W_{21}\Delta t\boldsymbol{f} & W_{22}\Delta t\boldsymbol{f} \end{bmatrix} \begin{Bmatrix} \boldsymbol{s}^{n+1/2} - \boldsymbol{s}^n \\ \boldsymbol{s}^{n+1} - \boldsymbol{s}^{n+1/2} \end{Bmatrix} + \begin{bmatrix} -0.5\Delta t\boldsymbol{J} \\ -0.5\Delta t\boldsymbol{J} \end{bmatrix} \begin{Bmatrix} \boldsymbol{s}^n \\ \boldsymbol{s}^n \end{Bmatrix}$$

2.2.4. Stabilization methods (SUPG – GLS)

As it is discussed widely in the reference book, Galerkin formulation lacks sufficient spatial stability when convective effects are important in the presence of boundary layers. Lots of methods are provided in reference book, chapter2, in order to stabilize the convection consistently, which means to ensure that the solution of the differential equation is also a solution of the weak form and we are not actually changing the problem statement by adding artificial inconsistent diffusivity. A stabilized formulation of the convection diffusion reaction problem can be stated as:

$$(\omega, u_t + \boldsymbol{a}.\boldsymbol{\nabla}u + \sigma u) + \boldsymbol{a}(\omega, u) + \sum_e \big(\tau P(\omega), R(u)\big)_{\Omega e} = (\omega, s) + (\omega, h)_{\Gamma N}$$

$$R(u) = u_t + \boldsymbol{a}.\boldsymbol{\nabla}u - \boldsymbol{\nabla}.(\nu\boldsymbol{\nabla}u) + \sigma u - s$$

Here, the perturbation operator $P(\omega)$ characterizes the stabilization method. The stabilization term involves the residual $R(u)$ of the governing equation in strong form, thus giving in principle a consistent formulation and holding the exact solution within itself. Note that the residual includes the time derivative u_t of the unknown which will result in a rather cumbersome implementation. So in practice we usually prefer to first discretize the strong form equation in time and then do the weakening procedure in space. As for the τ parameter, we use the convection diffusion reaction extensions of the parameters described in reference book for the family of theta methods:

$$\tau = \left[\left(\frac{1}{\theta\Delta t}\right)^2 + \left(\frac{2a}{h}\right)^2 + 9\left(\frac{4\upsilon}{h^2}\right)^2 + \sigma^2\right]^{-\frac{1}{2}}$$

As it is discussed widely in the reference book, when accuracy requires us to go beyond 2nd order schemes and higher-order time integration schemes are employed, the standard stabilization of the semi-discrete equations is not trivial. In order to have a consistent stabilization of time discretized equations, a residual must be defined after time discretization:

$$\left(\boldsymbol{\omega}, \frac{\Delta\boldsymbol{u}}{\Delta t}\right) - (\boldsymbol{\omega}, \boldsymbol{W}\Delta\boldsymbol{u}_t) + \sum_e (\boldsymbol{\tau}P(\boldsymbol{\omega}), R(\Delta\boldsymbol{u}))_{\Omega e} = (\boldsymbol{\omega}, \boldsymbol{w}u_t^n)$$

$$R(\Delta\boldsymbol{u}) = \frac{\Delta\boldsymbol{u}}{\Delta t} - \boldsymbol{W}\Delta\boldsymbol{u}_t - \boldsymbol{w}u_t^n = \frac{\Delta\boldsymbol{u}}{\Delta t} + \boldsymbol{W}\mathcal{L}(\Delta\boldsymbol{u}) - \boldsymbol{w}[s^n - \mathcal{L}(u^n)] + \boldsymbol{W}\Delta\boldsymbol{s}$$

Where $\mathcal{L} = \boldsymbol{a}.\boldsymbol{\nabla} - \boldsymbol{\nabla}.(\nu\boldsymbol{\nabla}) + \sigma$ as a spatial operator acts on each component of $\Delta\boldsymbol{u}$.

The perturbation operator $P(\boldsymbol{\omega})$ can include only the convective term which yields to the ***Streamline-upwind Petrov-Galerkin [SUPG]*** stabilization method or it can include all terms in the ***Galerkin/Least-squares [GLS]*** stabilization method. The non-symmetric structure of the stabilization term in SUPG method induces some technical difficulties in the stability analysis of this method. This is avoided in the GLS stabilization technique, because it introduces a symmetric stabilization term in a consistent manner.

$$P(\boldsymbol{\omega})\ for\ SUPG = \boldsymbol{W}(\boldsymbol{a}.\boldsymbol{\nabla})\boldsymbol{\omega}$$

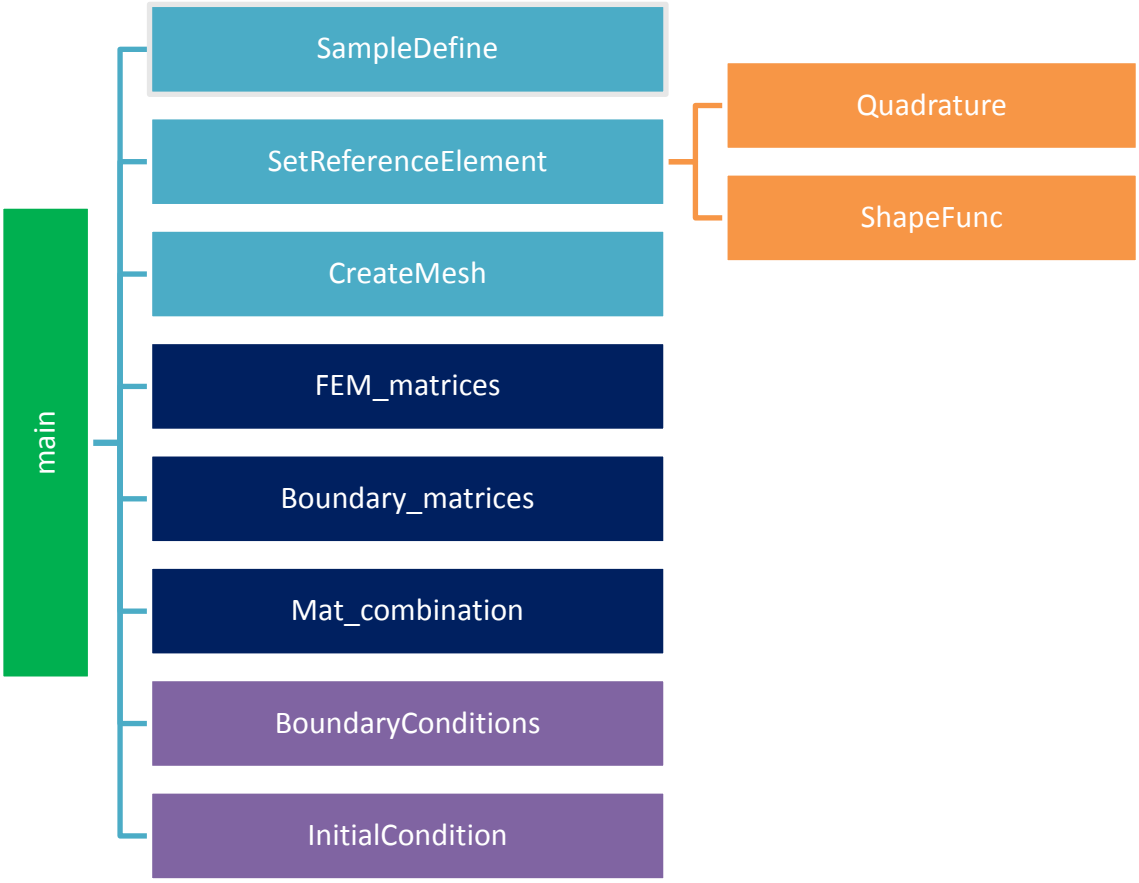
$$P2(\boldsymbol{\omega})\ for\ GLS = \frac{\boldsymbol{\omega}}{\Delta t} + \boldsymbol{W}\mathcal{L}(\boldsymbol{\omega}) = \frac{\boldsymbol{\omega}}{\Delta t} + \boldsymbol{W}[(\boldsymbol{a}.\boldsymbol{\nabla})\boldsymbol{\omega} - \boldsymbol{\nabla}.(\nu\boldsymbol{\nabla}\boldsymbol{\omega}) + \sigma\boldsymbol{\omega}]$$

In the implicit Pade' methods, the stabilization matrix can be computed as:

$$\tau = [\left(\frac{\boldsymbol{W}^{-1}}{\Delta t}\right) + \left(\frac{2a}{h} + \frac{4\nu}{h^2} + \sigma\right)\boldsymbol{I}]^{-T}\boldsymbol{W}^{-1}$$

Method	*R1,1*	*R2,2*
$\Delta\boldsymbol{u}$	$u^{n+1} - u^n$	$\begin{Bmatrix} u^{n+1/2} - u^n \\ u^{n+1} - u^{n+1/2} \end{Bmatrix}$
$\Delta\boldsymbol{s}$	$s^{n+1} - s^n$	$\begin{Bmatrix} s^{n+1/2} - s^n \\ s^{n+1} - s^{n+1/2} \end{Bmatrix}$
$\boldsymbol{W}$	$\frac{1}{2}$	$\frac{1}{24}\begin{pmatrix} 7 & -1 \\ 13 & 5 \end{pmatrix}$
$\boldsymbol{w}$	1	$\frac{1}{2}\begin{Bmatrix} 1 \\ 1 \end{Bmatrix}$

3. Code Algorithms and Modifications



❖ Subroutine (function) brief explanation:

- **main**
  This program solves 2D transient and steady convection-diffusion-reaction-source problem and all main subroutines are called from this Matlab main file. Finally after doing the solution loop for time increments and obtaining the solution at each time step, the post processing mechanisms are also defined in this file.

- **SampleDefine**
  All samples are defined here:
  - *4 steady case from the reference book, chapter2.*
  - *3 steady case for the assignment.*
  - *1 unsteady (non-diffusive) case from reference book, chapter3*
  - *1 unsteady (complete) case as the modified sample of the reference book, chapter5*
  - *1 unsteady (complete) case for the assignment*
  - *An open sample in which user can input any parameter from the scratch.*

- **SetReferenceElement**
  This subroutine gives back the Gauss points and their weights, shape functions and their derivatives (in reference coordinates) and the element local reference coordinates base in the type of element (i.e., quadrilateral or triangle) and interpolation degree (degree 1 or 2).

- **Quadrature**
  It returns Gauss points and weights on the reference element.

- **ShapeFunc**
  It returns shape functions and their derivatives (in reference coordinates). For the 2^{nd} order elements the 2^{nd} order derivatives are also calculated in this function.

- **CreateMesh**
  Returns back the element connectivity and the node coordinate matrices.

- **FEM_matrices**
  As the main core of the program, this subroutine has been under sever changes in order to calculates the global mass, stiffness, diffusion, reaction, source and also stabilization matrices based on the method of time and space discretization chosen by user.

- **Boundary_matrices**
  For the unsteady non-diffusive problems as hyperbolic cases, we need to have the outlet boundary matrices. This subroutine does this job.

- **Mat_combination**
  Based on the problem type and discretization method, this subroutine accepts the output matrices of "FEM_matrices" function as its input and adds them properly in order to have the whole right hand side matrix of the system of equations.

- **BoundaryConditions**
  Based on the sample chosen by user, this function defines appropriate boundary conditions. Note that for unsteady cases since we introduce the r.h.s. main equation in virational form of the unknown we should impose all Dirichlet B.Cs including zero and non-zero valus equal to 0 in all steps and then impose the non-zero Dirichlet B.Cs somewhere in the Initial B.Cs.

- **InitialCondition**
  Based on the sample chosen by user, this function defines appropriate Initial conditions which in this code can follow the boundary conditions shape or can be as an impulse initial condition.

4. Unsteady Case result

For all the exercises in this assignment we will consider a 2D domain of $\{0,2\} \times \{0,3\}$ with Dirichlet boundary conditions as following:

$$u = 1 \; on \; \Gamma\{0,0\} \times \{1.5,3\}$$

$$u = 0 \; on \; \Gamma\{2,2\} \times \{0,3\}$$

In this part we are going to solve the unsteady problem with the following convective velocity, diffusion parameter, reaction and source values:

UNSTEADY	a	v	σ	s
Case1	$(-10^{-3}, 0)$	10^{-3}	1	0

All three Pade' methods have the same results for each stabilization method. The results are provided in the following:

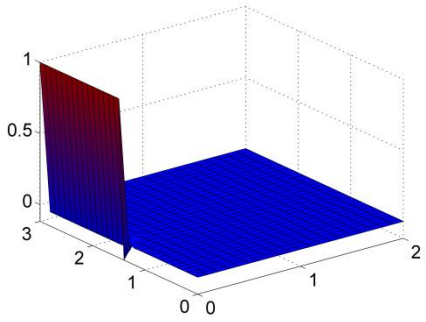
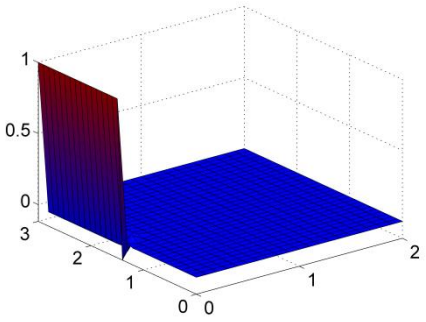
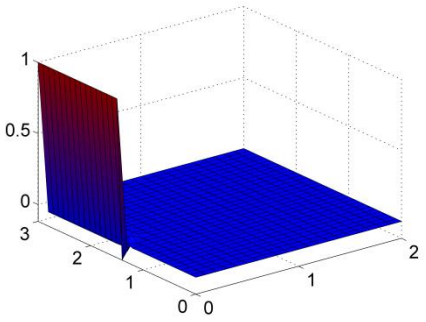
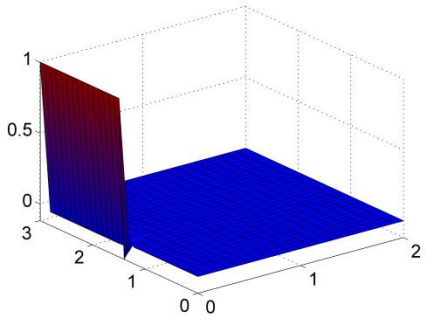
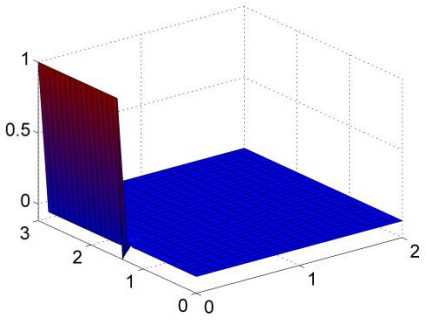
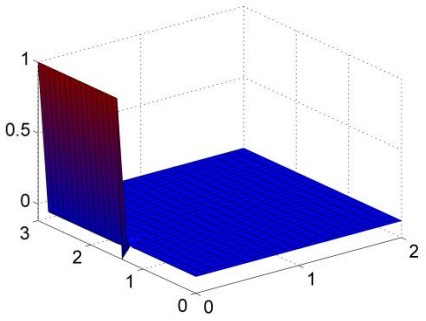
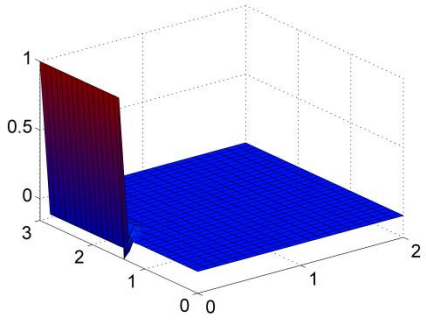
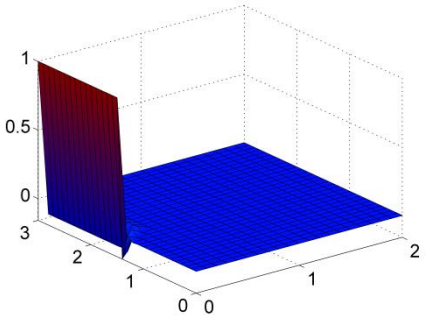
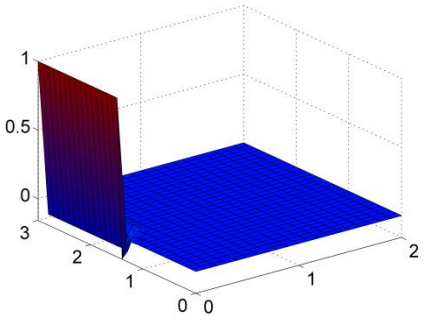
Stabilization	Min	Max
NO	-0.11868	1
SUPG	-0.11782	1
GLS	-0.15859	1

Although, the main difference between methods is the sensitivity of explicit method to the Peclet and Courant numbers while the implicit methods, are not. Based on the literature from the reference book it was supposed that for really small Courant numbers we would have stable explicit method behavior and it was also observed in the tests. Now we have chosen different parameters as the Courant number to be as small as we have a stable explicit method but when we play with dt and increase it we can observe that this method explodes.

$$Pe = \frac{ah}{2\vartheta} = \frac{0.001 * 0.1}{2 * 0.001} = 0.05$$

$$C = \frac{a\Delta t}{h} = \frac{0.001 * 0.1}{0.1} = 0.001$$

On the other hand, from computational cost point of view R1,1 is faster compared to others while R2,2 is slower.

Figue1. Case 1 – R1,1	*Figure2. Case 1 – R2,2*	*Figure3. Case 1 – R2,0*
		
No Stabilized	**No Stabilized**	**No Stabilized**
		
SUPG	**SUPG**	**SUPG**
		
GLS	**GLS**	**GLS**

5. Steady Case results

In this part we are going to solve the steady problem with the following convective velocity, diffusion parameter, reaction and source values:

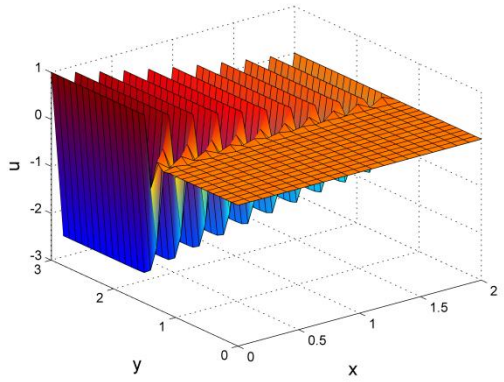
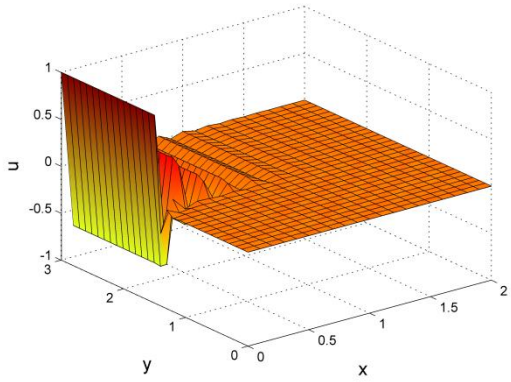
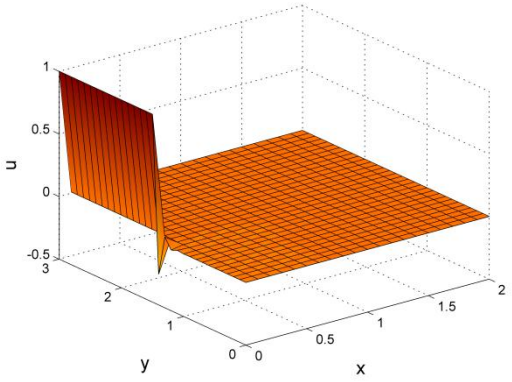
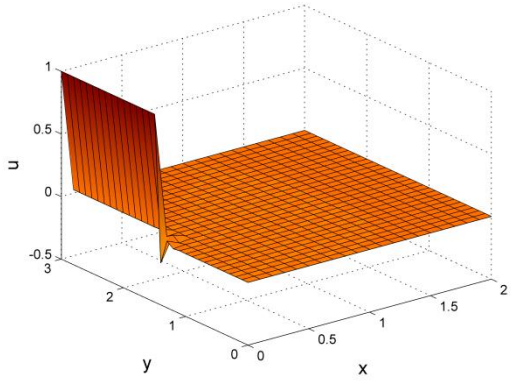
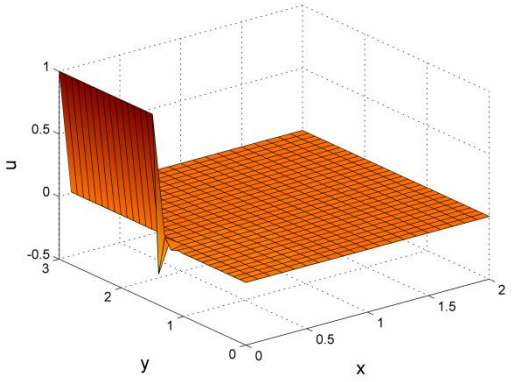
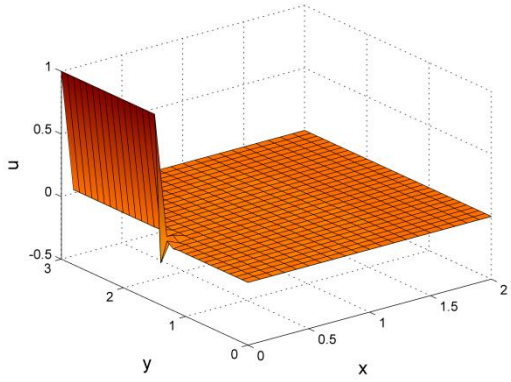
STEADY	Dominant parameter	a	v	σ	s
Case2_Main	Convection	$(-1 \quad ,0)$	10^{-3}	10^{-3}	0
Case2_New	- 10%	$(-0.1 \quad ,0)$	10^{-3}	10^{-3}	0
Case3_ Main	Reaction	$(-10^{-3},0)$	10^{-3}	1	0
Case3_New	- 10%	$(-10^{-3},0)$	10^{-3}	0.1	0
Case4_ Main	Source	$(-10^{-3},0)$	10^{-3}	0	1
Case4_New	- 10%	$(-10^{-3},0)$	10^{-3}	0	0.1

As it is introduced in the table we have 4 main steady cases in the assignment, but here as a case study for sensitivity analysis we may define an additional variant for each case with decreased dominant variable by 10 percent which are called new cases in the table.

It is interesting to note that the 2nd steady case (Case3_Main) has exactly the same parameters as the unsteady case. So we can expect that we would see similar results from both of them.

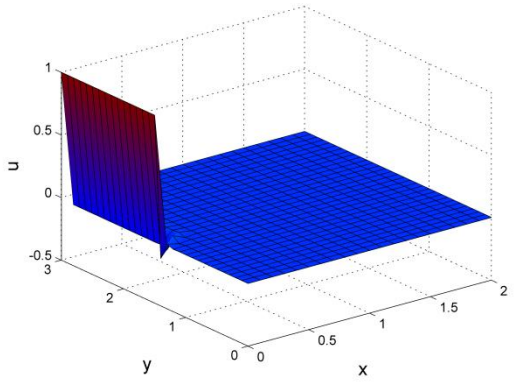
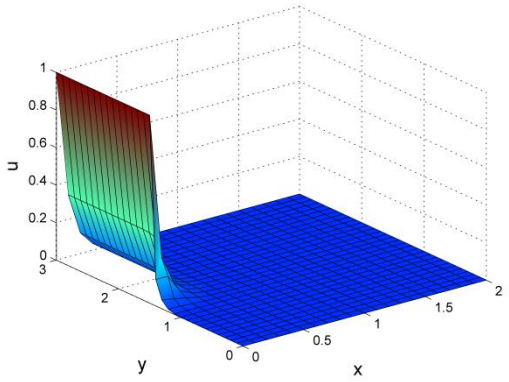
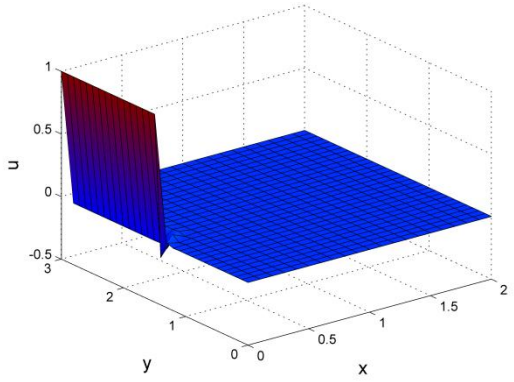
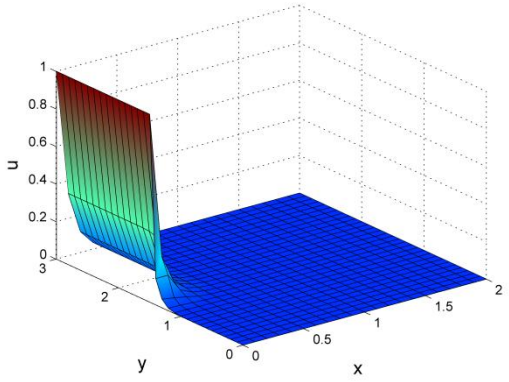
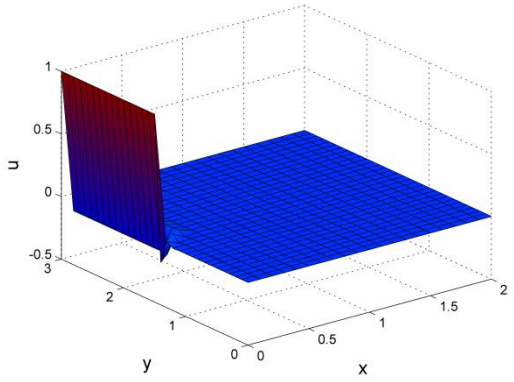
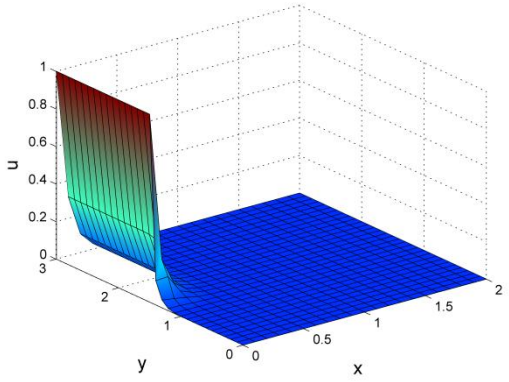
STEADY	a	v	σ	s
Case2_Main	$(-1 \quad ,0)$	10^{-3}	10^{-3}	0
Case2_New	$(-0.1 \quad ,0)$	10^{-3}	10^{-3}	0

As it is clearly visible in **Figure4** and **5**, the main difference is related to non-stabilized method in which we can observe that due to lower convection velocity in the Case2_New we have less oscillations compared to the main case. But by adding the stabilization terms, the differences become negligible

Figure4. Case 2_Main	*Figure5. Case 2_New*
	
No Stabilized	**No Stabilized**
	
SUPG	**SUPG**
	
GLS	**GLS**

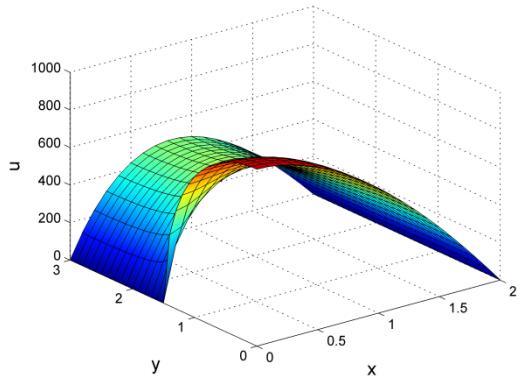
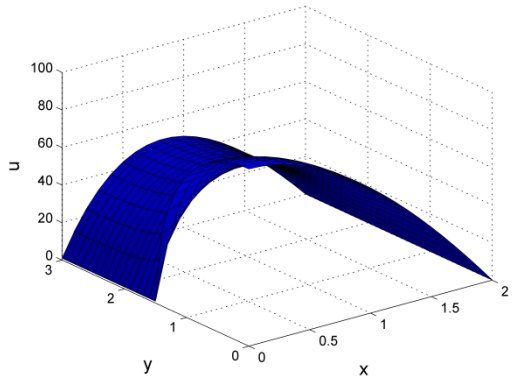
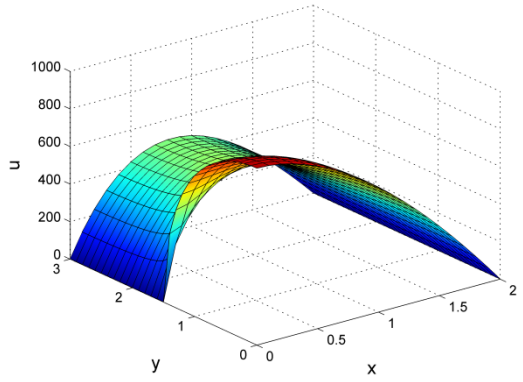
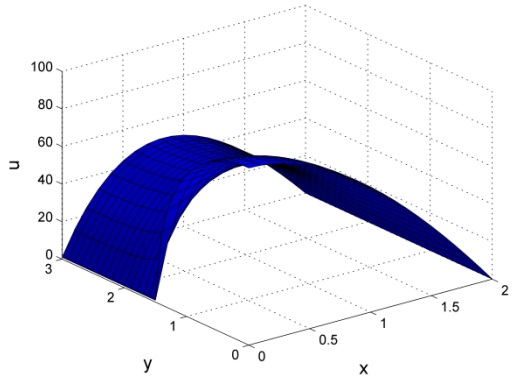
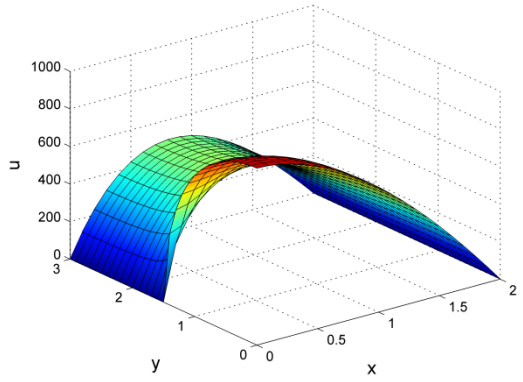
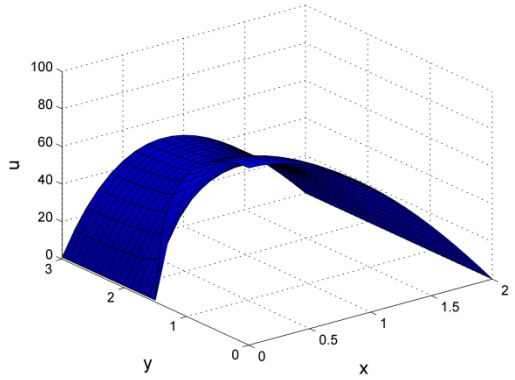
STEADY	a	v	σ	s
Case3_ Main	$(-10^{-3}, 0)$	10^{-3}	1	*0*
Case3_New	$(-10^{-3}, 0)$	10^{-3}	0.1	*0*

We can compare **Figure6** and **7** and point out that the effect of reduction in reaction term is obvious in the reduction of boundary oscillation and smoothness of results even in the non-stabilized method. As discussed before this steady case could be considered as the final step for the unsteady case (Case1) and we can see that ***Figure6*** clearly fits to ***Figure1***, ***2***, and ***3***.

Figue6. Case 3_Main	***Figue7. Case 3_New***
 **No Stabilized**	 **No Stabilized**
 **SUPG**	 **SUPG**
 **GLS**	 **GLS**

STEADY	a	v	σ	s
Case4_ Main	$(-10^{-3}, 0)$	10^{-3}	0	*1*
Case4_New	$(-10^{-3}, 0)$	10^{-3}	0	*0.1*

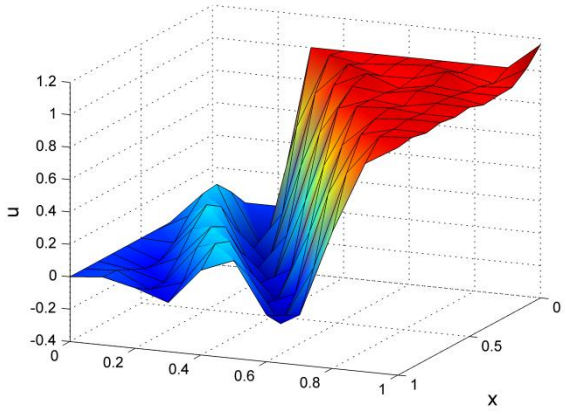
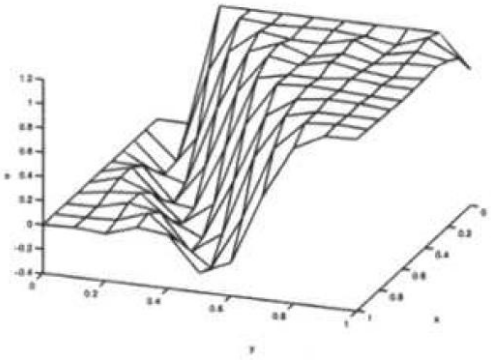
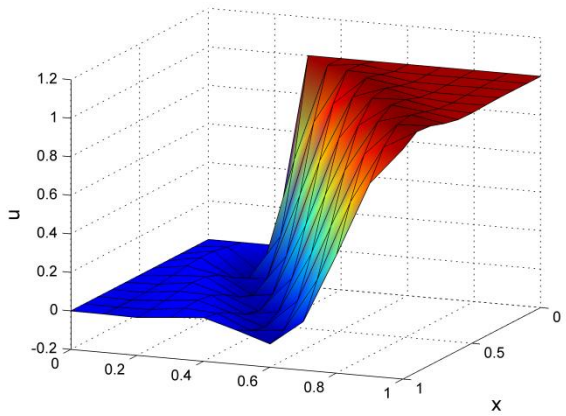
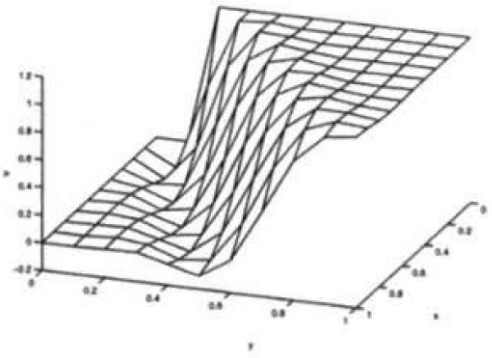
Here, we can stress out the effect of reduction in source term by looking to the lower bound of results in ***Figure9*** compared to ***Figure8*** by 10 percent, but the interesting point is that the global shape of domain after reaching the steady state is almost identical.

Figue8. Case 4_Main	***Figue9. Case 4_New***
 **No Stabilized**	 **No Stabilized**
 **SUPG**	 **SUPG**
 **GLS**	 **GLS**

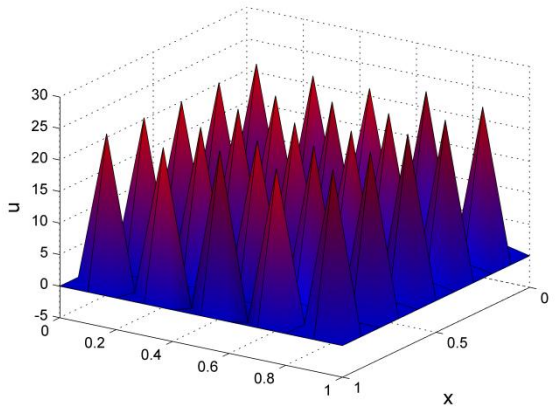
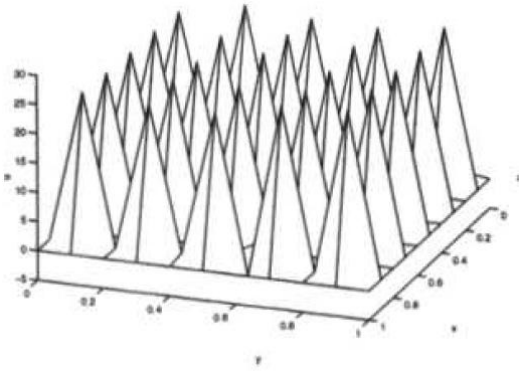
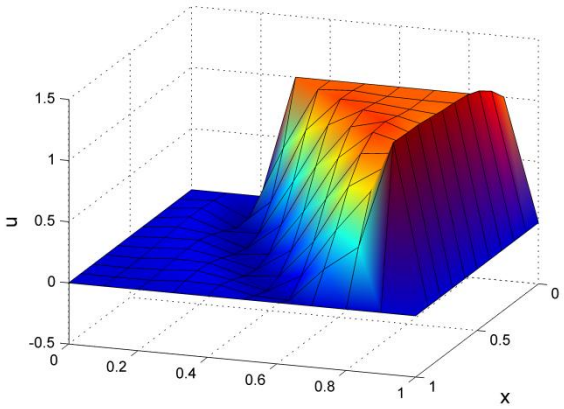
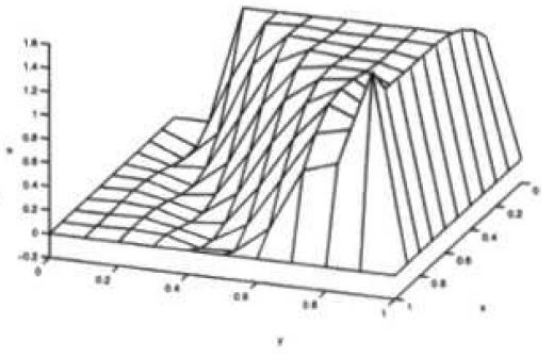
❖ *Appendix*

1. Appendix1: Code Verification [Steady case]

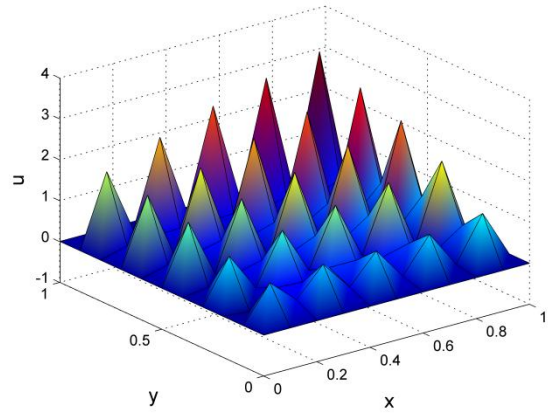
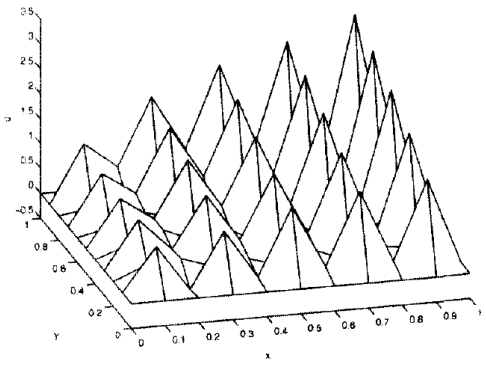
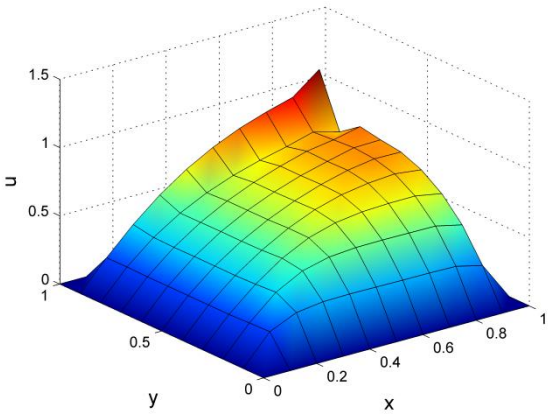
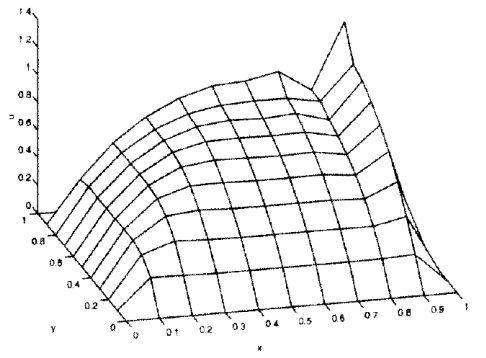
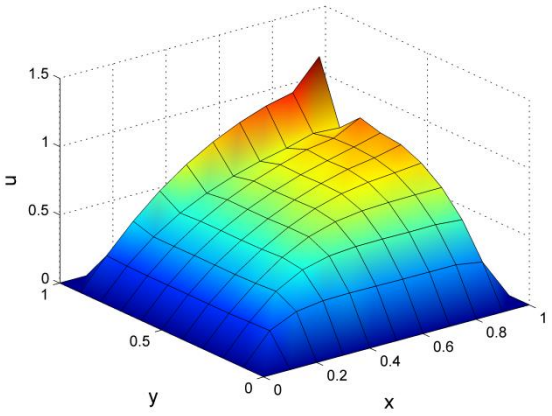
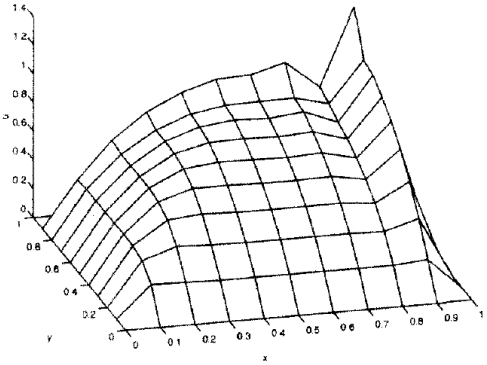
Here we compare the results for the solved sample [convection diffusion skew to the mesh with natural boundary conditions] in the reference book part 2.6.4 and we can see that the results obtained from code are exactly fit to the results provided in the book.

Figue10. Convection dominated (Code)	***Figue11. Convection dominated (reference)***
	
No Stabilized	**No Stabilized**
	
SUPG	**SUPG**

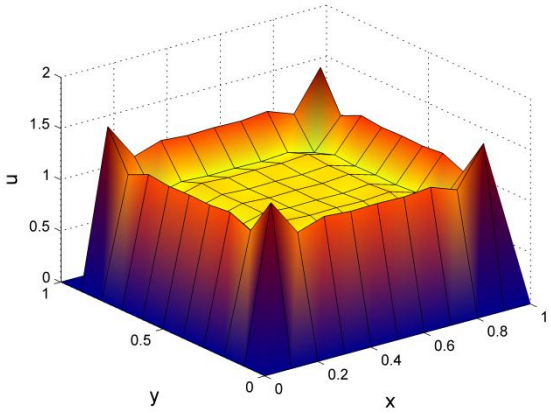
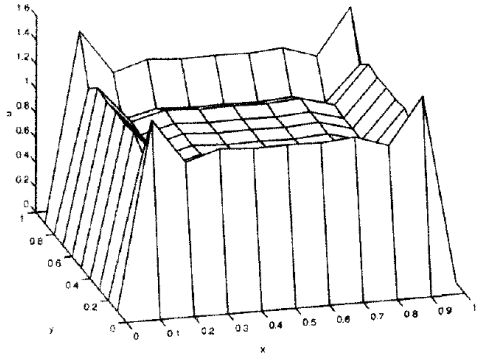
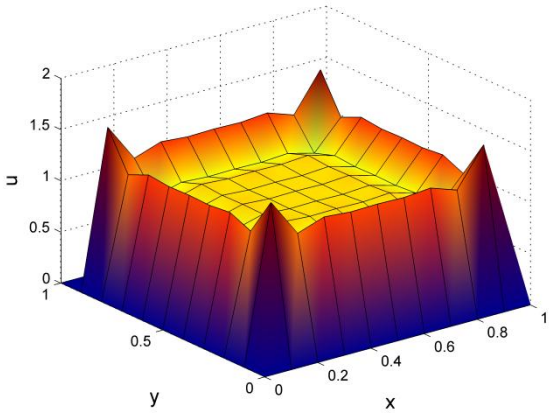
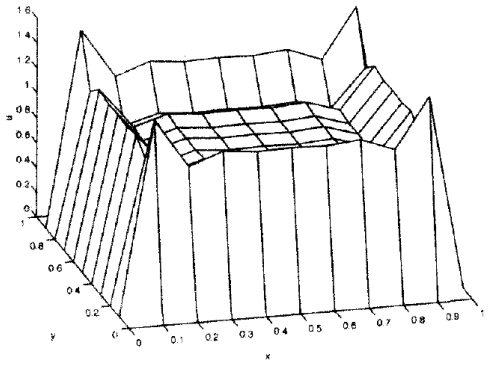
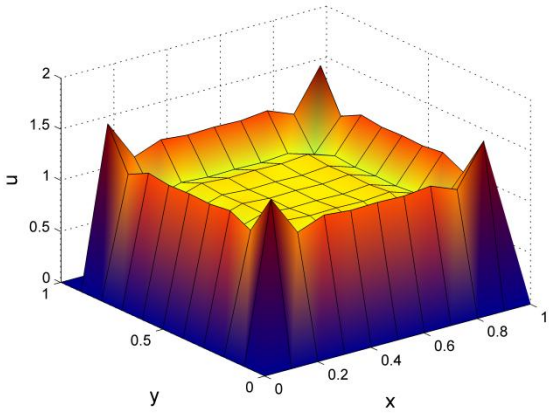
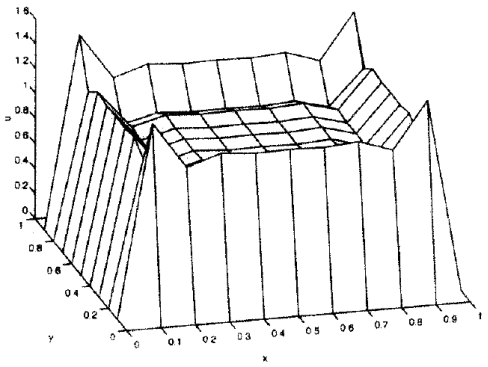
Here we compare the results for the solved sample [convection diffusion skew to the mesh with essential boundary conditions] in the reference book part 2.6.4 and we can see that the results obtained from code are exactly fit to the results provided in the book.

Figue12. Convection dominated (Code)	*Figue13. Convection dominated (reference)*
	
No Stabilized	**No Stabilized**
	
SUPG	**SUPG**

Here we compare the results for the solved sample [convection diffusion reaction skew to the mesh with essential boundary conditions] in the reference book part 2.6.5 and we can see that the results obtained from code are exactly fit to the results provided in the book.

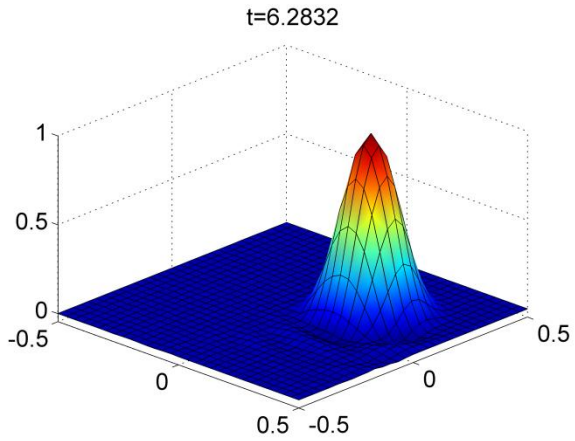
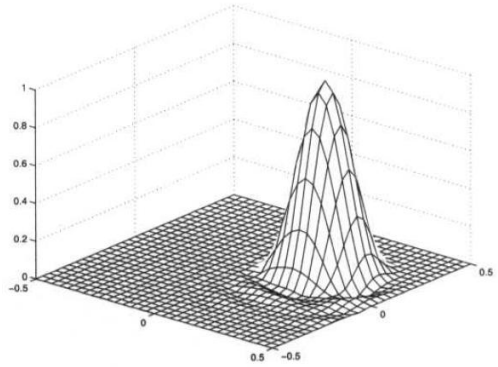
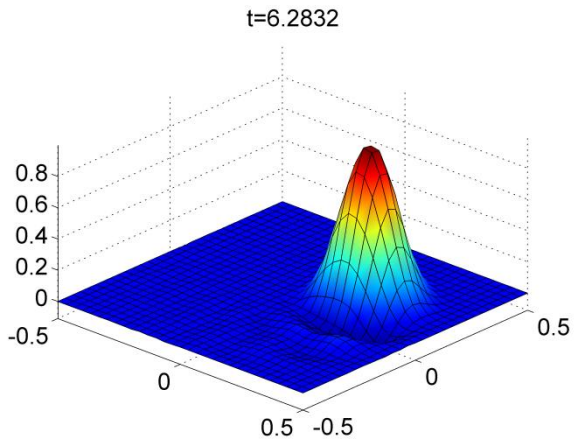
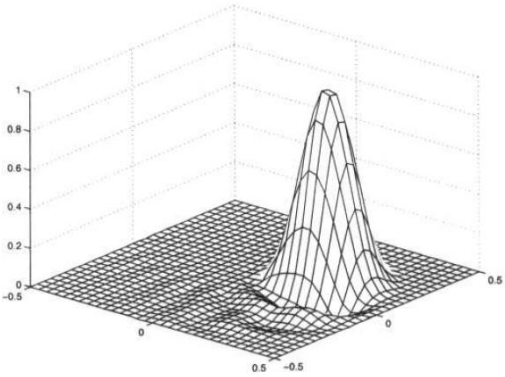
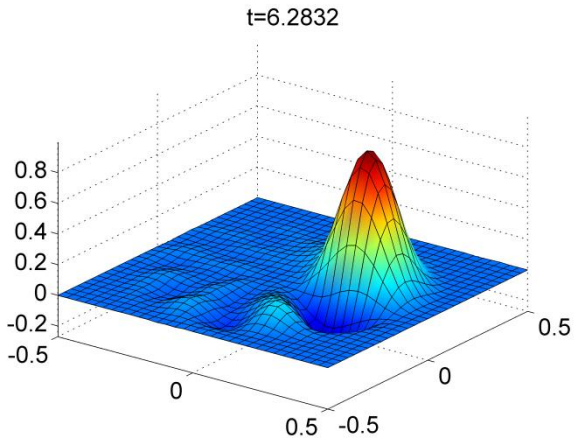
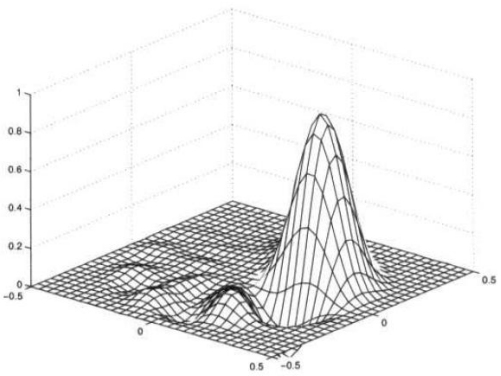
Figue14. Convection-Reaction dominated (Code)	*Figue15. Convection-Reaction dominated (reference)*
	
No Stabilized	**No Stabilized**
	
SUPG	**SUPG**
	
GLS	**GLS**

Here we compare the results for the solved sample [convection diffusion reaction skew to the mesh with essential boundary conditions] in the reference book part 2.6.5 and we can see that the results obtained from code are exactly fit to the results provided in the book.

Figue16. Reaction dominated (Code)	***Figue17. Reaction dominated (Reference)***
	
No Stabilized	**No Stabilized**
	
SUPG	**SUPG**
	
GLS	**GLS**

2. Appendix2: Code Verification [Unsteady case]

Here we compare the results for the solved sample [the rotating cone] in the reference book part 3.11.3 for unsteady pure convection equations and we can see that the results obtained from code are exactly fit to the results provided in the book.

Figue18.	***Figue19.***
	
No Stabilized – umax = 0.9969 [dt=2π/120]	**No Stabilized – umax = 0.9969 [dt=2π/120]**
	
SUPG – umax = 0.9691 [dt=2π/60]	**SUPG – umax = 0. 9691 [dt=2π/60]**
	
GLS– umax = 0.8931 [dt=2π/30]	**GLS – umax = 0. 8931 [dt=2π/30]**

3. Appendix3: Code Routines

a. Main

```
%% This program solves 2D transient convection-diffusion-reaction-source problem

clear; close all; clc
disp(' ')
disp('Problem_1: 2D steady    convection diffusion reaction equation [     a ux
- nu uxx + sig u = src]')
disp('Problem_2: 2D transient convection          reaction equation [ut + a ux
+ sig u = src]')
disp('Problem_3: 2D transient convection diffusion reaction equation [ut + a ux
- nu uxx + sig u = src]')
disp(' ')

ProblemType = input('Problem Type : ');

%% INPUT


if ProblemType==1

    disp(' ')
    disp('example_1  | Book-ch2-2.6.4 |
essential(0 & 1)+natural B.C')
    disp('example_2  | Book-ch2-2.6.4 |
essential(0 & 1)         B.C')
    disp('example_3  | Book-ch2-2.6.5 | conection-reaction dominated |
essential(all 0)         B.C')
    disp('example_4  | Book-ch2-2.6.5 | reaction           dominated |
essential(all 0)         B.C')
    disp('example_0  | New Input Data by user')
    disp('example_11 | Assignment - steady case1')
    disp('example_12 | Assignment - steady case2')
    disp('example_13 | Assignment - steady case3')
    disp(' ')
    examp = input('example : ');
elseif ProblemType==2
    disp(' ')
    disp('example_5  | Book-ch3-3.11.3 | rotating cone | No Source - No Reaction
| CN')
    disp('example_0  | New Input Data by user')
    disp(' ')
    examp = input('example : ');
elseif ProblemType==3
    disp(' ')
    disp('example_6  | modified Book-ch3-3.11.3 | rotating cone | No Source - No
Reaction | CN')
    disp('example_14 | Assignment - unsteady case')
    disp('example_0  | New Input Data by user')
    disp(' ')
    examp = input('example : ');
end

data = SampleDefine(examp);
data.ProblemType = ProblemType;
%-----------------------------------------------
elem     = 0;                                    %Element Type
p        = 1;                                    %Element Degree
referenceElement = SetReferenceElement(elem,p);
%-----------------------------------------------
```

In this part user can select 3 types of problems, namely elliptic, parabolic and hyperbolic.

Various predefined samples from the book and from the assignment are also available.

Users can skip the samples and enter any preferred values for data input.

```
[X,T] = CreateMesh(data,referenceElement);
nPt          = size (X,1);                          %Num of all Nodes
nElem        = size (T,1);                          %Num of all Elements
data.nPt     = nPt;
data.nElem   = nElem;
%-----------------------------------------------
ax           = data.ax;
ay           = data.ay;
velotype     = data.velotype;
velodeg      = data.velodeg;
if velotype==1                                      %Directional Velocity
    Conv = [ax*cos(velodeg)*ones(nPt,1), ay*si
elseif velotype==2
    Conv = [-X(:,2), X(:,1)];
end

%% Method Selection

if ProblemType==3
    disp(' ')
    disp('Temporal discretiztion methods : Pade approximation')
    disp('    [1] = [R11] Implicit - 1 stage - 2nd order : Crank-Nicolson')
    disp('    [2] = [R20] Explicit - 2 stage -           : Lax-Wendroff  ')
    disp('    [3] = [R22] Implicit - 2 stage - 4th order :               ')
    method1 = input('Method1 : ');
else
    method1=0;
end

if ProblemType~=2
    disp(' ')
    disp('Stabilization technique for Spatial discretization')
    disp('    [0] = Galerkin - No Stabilization!'           )
    disp('    [1] = SUPG'                                   )
    disp('    [2] = GLS'                                    )
    method2 = input('Method2 : ');
else
    method2=0;
end

data.method1 = method1;
data.method2 = method2;

%% Time Space Discritization

[M,K,C,R,f,S1,S3,S2,S11,S12,S21,S22,S31,S32] =
FEM_matrices(X,T,Conv,referenceElement,data);
[Mout,Cout]       = Boundary_matrices(X,T,Conv,referenceElement,data);
[A,B,method1Name] =
Mat_combination(M,K,C,R,S1,S3,S11,S12,S21,S22,S31,S32,Mout,Cout,data);

%% Boundary condition (Lagrange multipliers)

[ADir,bDir,nDir,nodesDir1,boundr] = BoundaryConditions(X,data);

if method1==3
    A2Dir = [ADir zeros(size(ADir)) ; zeros(size(ADir)) ADi
    Atot = [A A2Dir';A2Dir zeros(2*nDir,2*nDir)];
else
    Atot = [A ADir';ADir zeros(nDir,nDir)];
end
```

Various methods for solving unsteady cases are available.

2 types of stabilization methods are also prepared to run.

Main core of the program:

FEM_matrices

Mat_combination

Applying the

Boundary condition by

Lagrange multipliers

```
%% Initial condition

u0 = InitialCondition(X,data,boundr);
maxu=0; minu=0; maxuall=0; minuall=0;

%% Solution for Steady case [ProblemType = 1]

if ProblemType==1
    Btot    = [f + S2 ;bDir];
    sol     = Atot\Btot;
    Temp    = sol(1:nPt);
    maxu    = max(Temp(:));
    minu    = min(Temp(:));
end

%% Solution for Unsteady case [ProblemType = 2 or 3]

nStep   = data.nStep;
dt      = data.dt;
u       = zeros(nPt , nStep+1);
uMID    = zeros(nPt , nStep+1);
u(:,1)  = u0;

%-------------------------------                                --ProblemType=2
if ProblemType==2
    for n = 1:nStep           %loop at each time step
        disp(['Step number:  ', num2str(n),'   of   ', num2str(nStep)]);
        ftot  =[B * u(:,n) + dt * (f + S2) ; bDir];
        Du    = Atot\ftot;
        u(:,n+1) = u(:,n) + Du(1:nPt);
        maxuall = max(maxuall,max(u(:,n+1)));
        minuall = min(minuall,min(u(:,n+1)));
    end
    maxu = max(u(:,nStep+1));
    minu = min(u(:,nStep+1));
    Temp = u;
end

%---------------------------                                -----ProblemType=3
if ProblemType==3
    switch method1
%-------------------------------------------------------------------------R11
        case 1
        for n = 1:nStep           %loop at each time step
            disp(['Step number:  ', num2str(n),'   of   ', num2str(nStep)]);
            ftot        = [B * u(:,n) + dt * (f + S2) ; bDir];
            Du          = Atot\ftot;
            u(:,n+1)    = u(:,n) + Du(1:nPt);
            maxuall = max(maxuall,max(u(:,n+1)));
            minuall = min(minuall,min(u(:,n+1)));
        end
        maxu = max(u(:,nStep+1));
        minu = min(u(:,nStep+1));
        Temp = u;
%-------------------------------------------------------------------------R20
        case 2
        for n = 1:nStep           %loop at each time step
            disp(['Step number:  ', num2str(n),'   of   ', num2str(nStep)]);

            ftot        = [0.5 * B * u(:,n) + 0.5 * dt * (f + S2) ; bDir];
            DuMID         = Atot\ftot;
            uMID(:,n)     = u(:,n) + DuMID(1:nPt);
```

Solution for

Steady Case (elliptic)

Solution for

Unsteady Case (hyperbolic)

Solution for

Unsteady Case (parabolic)

```
            ftot        = [B * uMID(:,n) + dt * (f + S2) ; bDir];
            Du          = Atot\ftot;
            u(:,n+1)    = u(:,n) + Du(1:nPt);

            maxuall = max(maxuall,max(u(:,n+1)));
            minuall = min(minuall,min(u(:,n+1)));

        end
        maxu = max(u(:,nStep+1));
        minu = min(u(:,nStep+1));
        Temp = u;
%-------------------------------------------------------------------------R22
        case 3
        for n = 1:nStep           %loop at each time step
            disp(['Step number:  ', num2str(n),'   of   ', num2str(nStep)]);
            ftot        = [B * u(:,n) ; bDir ; bDir];
            Du          = Atot\ftot;
            u(:,n+1)    = u(:,n) + Du(1:nPt) + Du(nPt+1:2*nPt);
            maxuall = max(maxuall,max(u(:,n+1)));
            minuall = min(minuall,min(u(:,n+1)));
        end
        maxu = max(u(:,nStep+1));
        minu = min(u(:,nStep+1));
        Temp = u;

%-------------------------------------------------------------------------
    end
end


%% PostProcess
nx  = data.nx;
ny  = data.ny;
dom = data.dom;
Pe  = data.Pe;
Co  = data.Co;
tEnd= data.tEnd;


disp(['maxu  :  ', num2str(maxu)]);
disp(['minu  :  ', num2str(minu)]);

%-----------------------------------------------mesh
figure; PlotMesh(T,X,'b-');

%-----------------------------------------------velocity field on mesh 2D
figure(1); hold on;
quiver(X(:,1),X(:,2),Conv(:,1),Conv(:,2))
plot(dom([1,2,2,1,1]), dom([3,3,4,4,3]), 'k')

%-----------------------------------------------
xx  = reshape(X(:,1), nx+1, ny+1)';
yy  = reshape(X(:,2), nx+1, ny+1)';
%-----------------------------------------------

if ProblemType==1
    figure(9), clf
    sol = reshape(Temp, nx+1, ny+1)';
    surface(xx,yy,sol,'FaceColor','interp');
    xlabel('x','FontSize',14);
```

Post Processing

```
    ylabel('y','FontSize',14);
    zlabel('u','FontSize',14);
    grid on; view(3)

else
    %-----------------------------------------------initial condition 3D
    u0_dib = reshape(u0, nx+1, ny+1)';
    figure(2); clf;
    surface(xx,yy,u0_dib,'FaceColor','interp');    %plot figure in 2D
    set(gca,'FontSize',16)                         %edit axis values
    grid on
    view(3)                                        %make figure 3D

    %-----------------------------------------------final step
    figure(3), clf
    sol = reshape(Temp(:,nStep+1), nx+1, ny+1)';
    surface(xx,yy,sol,'FaceColor','interp');
    set(gca,'FontSize',16)
    grid on;
    axis([dom(1) dom(2) dom(3) dom(4) minuall maxuall])
    view(3)
%   title(['t=',num2str(tEnd),'    Pe=',num2str(Pe),'    C=',num2str(Co)])

    %-----------------------------------------------time intervals
    tt=0.0;
    for n = 1:10:nStep+1
        figure(4), clf
        sol = reshape(Temp(:,n), nx+1, ny+1)';
        surface(xx,yy,sol,'FaceColor','interp');
        set(gca,'FontSize',16)
        grid on;
        axis([dom(1) dom(2) dom(3) dom(4) minuall maxuall])
        view(3)
        tt=(n-1)*dt;
        title(['t=',num2str(tt)])
        pause(0.5)
    end
end
```

b. SampleDefine

```
function data = SampleDefine(examp)

if examp<= 2      %Chapter 2
    dom  = [ 0.0 , 1.0 ,  0.0 , 1.0];        %Domain
    nx = 10; ny = 10;                        %Num of Elements in X & Y dir

    velotype = 1; velodeg  = pi/6;           %directional velocity
    ax = 1.0  ; ay = 1.0;                    %velocity amplitide

    sig = 0.0;                               %reaction term
    src = 0.0;                               %source term
    nu  = 0.0001;                            %diffusion coefficent

    tEnd  = 0;                               %End Time
    nStep = 0;                               %Num of Time Steps

elseif examp== 3      %Chapter 2
    dom  = [ 0.0 , 1.0 ,  0.0 , 1.0];
    nx = 10; ny = 10;

    velotype = 1; velodeg  = pi/6;
    ax = 0.5 ; ay = 0.5;

    sig = 1.0;
    src = 1.0;
    nu  = 0.0001;

    tEnd  = 0;
    nStep = 0;

elseif examp== 4      %Chapter 2
    dom  = [ 0.0 , 1.0 ,  0.0 , 1.0];
    nx = 10; ny = 10;

    velotype = 1; velodeg  = pi/6;
    ax = 0.001 ; ay = 0.001 ;

    sig = 1.0;
    src = 1.0;
    nu  = 0.0001;

    tEnd  = 0;
    nStep = 0;

elseif examp==5  %Chapter 3
    dom  = [-0.5 , 0.5 , -0.5 , 0.5];
    nx = 30; ny = 30;

    velotype = 2; velodeg  = 0;          %rotational velocity (special)
    ax = 0 ; ay = 0 ;

    sig = 0;
    src = 0;
    nu  = 0;

    tEnd  = 2*pi();
    nStep = 30;
```

Defining different sample from the reference book of from the assignment.

User can read adequate information about each sample in the main file

```
elseif examp==6  % modified Chapter 3
    dom  = [-0.5 , 0.5 , -0.5 , 0.5];
    nx = 30; ny = 30;

    velotype = 2; velodeg  = 0;
    ax = 0 ; ay = 0 ;

    sig = 0;
    src = 0;
    nu  = 0.001;

    tEnd  = 2*pi();
    nStep = 60;

elseif examp==11  %=========================================assignment 1
    dom  = [ 0.0 , 2.0 ,  0.0 , 3.0];
    nx = 20; ny = 30;

    velotype = 1; velodeg  = 0.0;
%    ax = -1 ; ay = 0.0 ;
    ax = -0.1 ; ay = 0.0 ;        %New sample

    sig = 0.001;
    src = 0.0;
    nu  = 0.001;

    tEnd  = 10;
    nStep = 100;
elseif examp==12  %=========================================assignment 2
    dom  = [ 0.0 , 2.0 ,  0.0 , 3.0];
    nx = 20; ny = 30;

    velotype = 1; velodeg  = 0.0;
    ax = -0.001 ; ay = 0.0 ;

%    sig = 1.0;
     sig = 0.1;                      %New sample
    src = 0.0;
    nu  = 0.001;

    tEnd  = 10;
    nStep = 100;
elseif examp==13  %=========================================assignment 3
    dom  = [ 0.0 , 2.0 ,  0.0 , 3.0];
    nx = 20; ny = 30;

    velotype = 1; velodeg  = 0.0;
    ax = -0.001 ; ay = 0.0 ;

    sig = 0.0;
%     src = 1.0;
   src = 0.1;                    %New Sample
    nu  = 0.001;

    tEnd  = 5000;
    nStep = 100;

elseif examp==14  %=========================================assignment 4
    dom  = [ 0.0 , 2.0 ,  0.0 , 3.0];
    nx = 20; ny = 30;
```

```
    velotype = 1; velodeg  = 0;
    ax = 0.001 ; ay = 0.0 ;

    sig = 1.0;
    src = 0.0;
    nu  = 0.001;

    tEnd  = 10;       %Implicit
    nStep = 100;      %Implicit

%=============================================================new data input
elseif examp==0
    dom_1       = input('dom_1 : ');
    dom_2       = input('dom_2 : ');
    dom_3       = input('dom_3 : ');
    dom_4       = input('dom_4 : ');
    dom         = [dom_1 dom_2 dom_3 dom_4];
    nx          = input('nx : ');
    ny          = input('ny : ');
    velotype    = input('velotype : ');
    velodeg     = input('velodeg : ');
    ax          = input('ax : ');
    ay          = input('ay : ');
    sig         = input('sig : ');
    src         = input('src : ');
    nu          = input('nu : ');
    tEnd        = input('tEnd : ');
    nStep       = input('nStep : ');
end

dt = tEnd / nStep;                                  %Time discret. intervals
hx = (dom(2) - dom(1))/nx;                            %Space discret. intervals
hy = (dom(4) - dom(3))/ny;

h  = (hx+hy)/2;
a  = sqrt(ax^2+ay^2);

Pe = a   * h  / (2 * nu);
Co = a   * dt / h;
dd = nu  * dt / (h * h);
r  = sig * dt;
tau = ( (2*a/h)^2 + 9*(4*nu/(h^2))^2 + sig^2 )^(-1/2);

Dirval=1;
%-----------------------------------------------
data.examp      = examp;
data.dom        = dom;

data.ax         = ax;
data.ay         = ay;

data.nx         = nx;
data.ny         = ny;
data.hx         = hx;
data.hy         = hy;
data.h          = h;

data.nu         = nu;
data.sig        = sig;
data.src        = src;

data.velotype   = velotype;
```

Option for data input by user

```
data.velodeg    = velodeg;

data.tEnd       = tEnd;
data.nStep      = nStep;
data.dt         = dt;

data.Dirval     = Dirval;

data.Pe         = Pe;
data.Co         = Co;

end
```

c. SetReferenceElement

```
function Element = SetReferenceElement(elem,p)
% elem    : type of element (0: quadrilatera, 1: triangles)
% p       : interpolation degree (1 or 2)
% Element : struture with the reference element's properties

%-----------------------------------------------------Quadrilateral element
if elem == 0
%-------------------------------------linear order: 4 node - 4 gauss point
    if p == 1
        ngaus = 4;
        Xe_ref = [-1,-1; 1,-1; 1,1; -1,1];
%-----------------------------------quadratic order: 9 node - 9 gauss point
    elseif p == 2
        ngaus = 9;
        Xe_ref =  [-1,-1; 1,-1; 1,1; -1,1; 0,-1; 1,0; 0,1; -1,0; 0,0];
    else
        error('not avilable interpolation degree');
    end
%-------------------------------------------------------------Trianle element
elseif elem == 1
%-------------------------------------linear order: 3 node - 3 gauss point
    if p == 1
        ngaus = 3;
        Xe_ref = [0,0; 1,0; 0,1];
%-----------------------------------quadratic order: 6 node - 6 gauss point
    elseif p == 2
        ngaus = 6;
        Xe_ref = [0,0; 1,0; 0,1; 0.5,0; 0.5,0.5; 0,0.5];
    else
        error('not avilable interpolation degree');
    end
else
    error('unavailable element')
end

%-----------------------------------------------------Gauss points and weights
[zgp,wgp] = Quadrature(elem,ngaus);

%----------Shape functions and their derivatives (in reference coordinates)
[N,Nxi,Neta,N2xi,N2eta] = ShapeFunc(elem,p,zgp);

%-------------------------------------------------------------------------------
Element.degree          = p;
Element.elem            = elem;
Element.Xe_ref          = Xe_ref;
Element.nen             = size(Xe_ref,1);
Element.ngaus           = ngaus;
Element.GaussPoints     = zgp;
Element.GaussWeights    = wgp;
Element.N               = N;
Element.Nxi             = Nxi;
Element.Neta            = Neta;
Element.N2xi            = N2xi;
Element.N2eta           = N2eta;
```

NO CHANGE EXCEPT DEFINING 2ND ORDER DERIVATIVES FOR PUTTING INTO THE RESIDUAL TERM IN 2ND ORDER ELEMENTS

d. Quadrature

```
function [zgp,wgp] = Quadrature(elem,ngaus)
% zgp, wgp  : Gauss points and weights on the reference element
% ngaus     : number of Gauss points in the element
% elem      : type of element (0: quadrilatera, 1: triangles)

%-----------------------------------------------------quadrilateral element
if elem == 0
    n = ceil(sqrt(ngaus));          % here n = 2 or 3
    [pg_1D, wg_1D] = Quadrature_1D(n);

    xx = pg_1D;
    yy = pg_1D;
    [xx,yy] = meshgrid(xx,yy);
    xx = reshape(xx,n^2,1);     %vector (column)  of gauss point positions
    yy = reshape(yy,n^2,1);

    zgp = [xx,yy];              %matrix (n^2 * 2) of gauss point positions
    wgp = wg_1D'*wg_1D;
    wgp = reshape(wgp,1,n^2);   %vector (row)     of gauss point weights

%-----------------------------------------------------------triangle element
elseif elem == 1
    zgp  = zeros(ngaus, 2);
    wgp = zeros(1, ngaus);
%-------------------------------------------------------------------------order 1
    if ngaus==1
        zgp(1,1:2)=[1/3,1/3];
        wgp = 0.5;
%-------------------------------------------------------------------------order 2
    elseif ngaus == 3
        zgp(1,:)=[2/3,1/6];
        zgp(2,:)=[1/6,2/3];
        zgp(3,:)=[1/6,1/6];
        wgp = 0.5*[1/3 1/3 1/3];
%-------------------------------------------------------------------------order 3
    elseif ngaus == 4
        zgp(1,1) = 1/3;     zgp(1,2) = 1/3;
        zgp(2,1) = 0.6;     zgp(2,2) = 0.2;
        zgp(3,1) = 0.2;     zgp(3,2) = 0.6;
        zgp(4,1) = 0.2;     zgp(4,2) = 0.2;

        wgp(1) = -27.0/96.0;
        wgp(2) =  25.0/96.0;
        wgp(3) =  25.0/96.0;
        wgp(4) =  25.0/96.0;
%-------------------------------------------------------------------------order 4
    elseif ngaus == 6
        a=0.659027622374092;
        b=0.231933368553031;
        c=0.109039009072877;
        zgp(1,:)=[a,b];
        zgp(2,:)=[b,a];
        zgp(3,:)=[a,c];
        zgp(4,:)=[c,a];
        zgp(5,:)=[b,c];
        zgp(6,:)=[c,b];
        wgp = 0.5*[1/6 1/6 1/6 1/6 1/6 1/6];
%-------------------------------------------------------------------------order 6
    elseif ngaus == 12
        a=0.873821971016996;
```

```
        b=0.063089014491502;
        zgp(1,:)=[a,b];
        zgp(2,:)=[b,b];
        zgp(3,:)=[b,a];
        a=0.501426509658179;
        b=0.249286745170910;
        zgp(4,:)=[a,b];
        zgp(5,:)=[b,b];
        zgp(6,:)=[b,a];
        a=0.636502499121399;
        b=0.310352451033785;
        c=0.053145049844816;
        zgp( 7,:)=[a,b];
        zgp( 8,:)=[b,a];
        zgp( 9,:)=[a,c];
        zgp(10,:)=[c,a];
        zgp(11,:)=[b,c];
        zgp(12,:)=[c,b];
        wgp = 0.5*[  0.050844906370207,0.050844906370207,0.050844906370207, ...
            0.116786275726379,0.116786275726379,0.116786275726379, ...
            0.082851075618374,0.082851075618374,0.082851075618374, ...
            0.082851075618374,0.082851075618374,0.082851075618374];
    else
        error('unavailable quadrature')
    end
else
    error('unavailable quadrature')
end


function [z, w] = Quadrature_1D(ngaus)

if ngaus == 1
    z = 0;
    w = 2;
elseif ngaus == 2
    pos1 = 1/sqrt(3);
    z = [-pos1; pos1];
    w = [ 1 1 ];
elseif ngaus == 3
    pos1 = sqrt(3/5);
    z = [-pos1; 0; pos1];
    w = [ 5/9   8/9   5/9 ];
elseif ngaus == 4
    pos1 = sqrt(525+70*sqrt(30))/35;
    pos2 = sqrt(525-70*sqrt(30))/35;
    z = [-pos1; -pos2; pos2; pos1];
    w1 = sqrt(30)*(3*sqrt(30)-5)/180;
    w2 = sqrt(30)*(3*sqrt(30)+5)/180;
    w = [w1   w2   w2   w1];
elseif ngaus == 5
    r70 = sqrt(70);
    pos1 = sqrt(245+14*r70)/21;
    pos2 = sqrt(245-14*r70)/21;
    z = [-pos1; - pos2; 0; pos2; pos1];
    w1 = (7+5*r70)*3*r70/(100*(35+2*r70));
    w2 = -(-7+5*r70)*3*r70/(100*(-35+2*r70));
    w0 = 128/225;
    w = [w1,w2,w0,w2,w1];
elseif ngaus == 6
    z = [0.23861918608319690863050172168066;
0.66120938646626451366139959501991;
```

```
         0.93246951420315202781230155449406;    -
0.23861918608319690863050172168066;
        -0.66120938646626451366139959501991;    -
0.93246951420315202781230155449406];
    w = [0.46791393457269104738987034398801
0.36076157304813860756983351383812, ...
         0.17132449237917034504029614217260
0.46791393457269104738987034398891, ...
         0.36076157304813860756983351383816
0.17132449237917034504029614217271 ];
elseif ngaus == 7
    z = [0.
         0.40584515137739716690660641207692;
0.74153118559939443986386477328078
         0.94910791234275852452618968404784;    -
0.40584515137739716690660641207692
        -0.74153118559939443986386477328078;    -
0.94910791234275852452618968404784 ];
    w = [0.41795918367346938777551020408166, ...
         0.38183005050511894495036977548841
0.27970539148927666790146777142377, ...
         0.12948496616886969327061143267904
0.38183005050511894495036977548964, ...
         0.27970539148927666790146777142336
0.12948496616886969327061143267912 ];
elseif ngaus == 8
    z = [0.18343464249564980493947614236027;
0.52553240991632898581773904918921
         0.79666647741362673959155393647586;
0.96028985649753623168356086856950
        -0.18343464249564980493947614236027;    -
0.52553240991632898581773904918921
        -0.79666647741362673959155393647586;    -
0.96028985649753623168356086856950];
    w = [0.36268378337836198296651504492780
0.31370664587788728733796220198797, ...
         0.22238103445337447054435599442573
0.10122853629037625915253135431028, ...
         0.36268378337836198296651504492834
0.31370664587788728733796220198807, ...
         0.22238103445337447054435599442632
0.10122853629037625915253135431015];
elseif ngaus == 9
    z = [0.
         .32425342340380892903853801464336;
.61337143270059039730870203934149
         .83603110732663579429942978806972;
.96816023950762608983557620290365
        -.32425342340380892903853801464336;    -
.61337143270059039730870203934149
        -.83603110732663579429942978806972;    -
.96816023950762608983557620290365];
    w = [.33023935500125976316452506692903;
         .31234707704000284006863040658887;      .26061069640293546231874286941888
         .18064816069485740405847203124168;
0.81274388361574411971892158110806e-1
        .31234707704000284006863040658836;       .26061069640293546231874286941508
        .18064816069485740405847203124263;
0.81274388361574411971892158110938e-1 ]';
elseif ngaus == 10
    z = [.14887433898163121088482600112972;
.43339539412924719079926594316579
         .67940956829902440623432736511487;
.86506336668898451073209668842349
```

```
         .973906528517171720077964012084 45;     -
.148874338981631210884826001129 72
        -.433395394129247190799265943165 79;     -
.679409568299024406234327365114 87
        -.865063366688984510732096688423 49;     -
.973906528517171720077964012084 45];
    w = [.295524224714752870173892994660 1;
.269266719309996355091226921568 67
         .219086362515982043995534934227 96;
.149451349150580593145776339660 48
        0.666713443086881375935688098948 98e-1;
.295524224714752870173892994648 4
        .269266719309996355091226921573 23;
.219086362515982043995534934228 77
        .149451349150580593145776339655 78;
0.666713443086881375935688098932 98e-1]';
else
    error('unavailable quadrature')
end
```

e. ShapeFunc

```
function [N,Nxi,Neta,N2xi,N2eta] = ShapeFunc(elem,p,z)
% N, Nxi, Neta  : matrices storing the values of the shape functions
%                 on the Gauss points of the reference element
%                 Each row concerns to a Gauss point
% z             : coordinates of Gauss points in the reference element
% elem          : type of element (0: quadrilatera, 1: triangles)
% p             : interpolation degree

xi = z(:,1); eta = z(:,2);
if elem == 0                    %quadrilateral element
    if p == 1                   %linear order    : 4 node - 4 gauss point
        N    = [(1-xi).*(1-eta)/4, (1+xi).*(1-eta)/4, (1+xi).*(1+eta)/4, (1-
xi).*(1+eta)/4];
        Nxi  = [(eta-1)/4, (1-eta)/4, (1+eta)/4, -(1+eta)/4];
        Neta = [(xi-1)/4, -(1+xi)/4,   (1+xi)/4,  (1-xi)/4 ];
        N2xi = [0*xi, 0*xi, 0*xi, 0*xi];
        N2eta= [0*eta, 0*eta, 0*eta, 0*eta];
    elseif p == 2               %quadratic order : 9 node - 9 gauss point
        N    = [xi.*(xi-1).*eta.*(eta-1)/4, xi.*(xi+1).*eta.*(eta-1)/4, ...
                xi.*(xi+1).*eta.*(eta+1)/4, xi.*(xi-1).*eta.*(eta+1)/4, ...
                 (1-xi.^2).*eta.*(eta-1)/2, xi.*(xi+1).*(1-eta.^2)/2,   ...
                 (1-xi.^2).*eta.*(eta+1)/2, xi.*(xi-1).*(1-eta.^2)/2,   ...
                 (1-xi.^2).*(1-eta.^2)];
        Nxi  = [(xi-1/2).*eta.*(eta-1)/2,   (xi+1/2).*eta.*(eta-1)/2, ...
                (xi+1/2).*eta.*(eta+1)/2,   (xi-1/2).*eta.*(eta+1)/2, ...
                -xi.*eta.*(eta-1),          (xi+1/2).*(1-eta.^2),   ...
                -xi.*eta.*(eta+1),          (xi-1/2).*(1-eta.^2),   ...
                -2*xi.*(1-eta.^2)];
        Neta = [xi.*(xi-1).*(eta-1/2)/2,    xi.*(xi+1).*(eta-1/2)/2, ...
                xi.*(xi+1).*(eta+1/2)/2,    xi.*(xi-1).*(eta+1/2)/2, ...
                 (1-xi.^2).*(eta-1/2),      xi.*(xi+1).*(-eta),   ...
                 (1-xi.^2).*(eta+1/2),      xi.*(xi-1).*(-eta),   ...
                 (1-xi.^2).*(-2*eta)];
        N2xi = [eta.*(eta-1)/2,   eta.*(eta-1)/2, ...
                eta.*(eta+1)/2,   eta.*(eta+1)/2, ...
                -eta.*(eta-1),        (1-eta.^2), ...
                -eta.*(eta+1),        (1-eta.^2), ...
                -2*(1-eta.^2)];
        N2eta = [xi.*(xi-1)/2,    xi.*(xi+1)/2, ...
                 xi.*(xi+1)/2,    xi.*(xi-1)/2, ...
                    (1-xi.^2),   -xi.*(xi+1),   ...
                    (1-xi.^2),   -xi.*(xi-1),   ...
                 -2*(1-xi.^2)];
    else
        error('not available interpolation degree')
    end
elseif elem == 1                %triangle element
    if p == 1                   %linear order    : 3 node - 3 gauss point
        N    = [1-xi-eta, xi, eta];
        Nxi  = [-ones(size(xi)), ones(size(xi)), zeros(size(xi))];
        Neta = [-ones(size(xi)), zeros(size(xi)), ones(size(xi))];
    else
        error('not available interpolation degree')
    end
else
    error('not available element')
end
```

NO CHANGE EXCEPT DEFINING 2ND ORDER DERIVATIVES FOR PUTTING INTO THE RESIDUAL TERM IN 2ND ORDER ELEMENTS

f. CreateMesh

```
function [X,T] = CreateMesh(data,referenceElement)

elem = referenceElement.elem;
nen = referenceElement.nen;
p = referenceElement.degree;

dom = data.dom;
nx  = data.nx;
ny  = data.ny;

x1 = dom(1); x2 = dom(2);
y1 = dom(3); y2 = dom(4);

npx = p*nx + 1;
npy = p*ny + 1;

npt = npx*npy;
x = linspace(x1,x2,npx);
y = linspace(y1,y2,npy);
[x,y] = meshgrid(x,y);
X = [reshape(x',npt,1), reshape(y',npt,1)];

%-----------------------------------------------------quadrilateral element
if elem == 0
    T = zeros(nx*ny,nen);
%-------------------------------------------------------1st order element
    if nen == 4
        for i=1:ny
            for j=1:nx
                ielem = (i-1)*nx+j;
                inode = (i-1)*(npx)+j;
                T(ielem,:) = [inode   inode+1   inode+npx+1   inode+npx];
            end
        end
%-------------------------------------------------------2nd order element
    elseif nen == 9
        T = zeros(nx*ny,9);
        for i=1:ny
            for j=1:nx
                ielem = (i-1)*nx + j;
                inode = (i-1)*2*npx + 2*(j-1) + 1;
                nodes_aux = [inode+(0:2)  inode+npx+(0:2)  inode+2*npx+(0:2)];
                T(ielem,:) = nodes_aux([1  3  9  7  2  6  8  4  5]);
            end
        end
    else
        error('not available element')
    end

%-----------------------------------------------------------triangle element
elseif elem == 1
%-------------------------------------------------------1st order element
    if nen == 3
        T = zeros(nx*ny,3);
        for i=1:ny
            for j=1:nx
                ielem = 2*((i-1)*nx+j)-1;
                inode = (i-1)*(npx)+j;
                T(ielem,:) = [inode   inode+1   inode+(npx)];
                T(ielem+1,:) = [inode+1   inode+1+npx   inode+npx];
```

```
            end
        end
        % Modification of left lower and right upper corner elements
        % to avoid them having all their nodes on the boundary
        if npx > 2
            T(1,:) = [1  npx+2   npx+1];
            T(2,:) = [1    2     npx+2];
            aux = size(T,1);
            T(aux-1,:) = [npx*ny-1    npx*npy   npx*npy-1];
            T(aux,:)   = [npx*ny-1    npx*ny    npx*npy];
        end
%-------------------------------------------------------------2nd order element
    elseif nen == 6
        T = zeros(2*nx*ny,6);
        for i=1:ny
            for j=1:nx
                ielem=2*((i-1)*nx+j)-1;
                inode=(i-1)*2*(npx)+2*(j-1)+1;
                nodes_aux = [inode+(0:2)  inode+npx+(0:2)  inode+2*npx+(0:2)];
                T(ielem,:) = nodes_aux([1  3  7  2  5  4]);
                T(ielem+1,:) = nodes_aux([3  9  7  6  8  5]);
            end
        end
        % Modification of left lower and right upper corner elements
        % to avoid them having all their nodes on the boundary
        if npx > 3
            inode = 1;
            nodes_aux = [inode+(0:2)  inode+npx+(0:2)  inode+2*npx+(0:2)];
            T(1,:) = nodes_aux([1  9  7  5  8  4]);
            T(2,:) = nodes_aux([1  3  9  2  6  5]);

            ielem  = size(T,1)-1;
            inode = npx*(npy-2)-2;
            nodes_aux = [inode+(0:2)  inode+npx+(0:2)  inode+2*npx+(0:2)];
            T(ielem,:) = nodes_aux([1  9  7  5  8  4]);
            T(ielem+1,:) = nodes_aux([1  3  9  2  6  5]);
        end
    else
        error('not available element')
    end
else
    error('not available element')
end
```

g. FEM_matrices

```
function [M,K,C,R,f,S1,S3,S2,S11,S12,S21,S22,S31,S32] =
FEM_matrices(X,T,Conv,referenceElement,data)

nen      = referenceElement.nen;
ngaus    = referenceElement.ngaus;
wgp      = referenceElement.GaussWeights;
N        = referenceElement.N;
Nxi      = referenceElement.Nxi;
Neta     = referenceElement.Neta;
N2xi     = referenceElement.N2xi;
N2eta    = referenceElement.N2eta;
p        = referenceElement.degree;

nElem    = data.nElem;
nPt      = data.nPt;

M  = zeros(nPt);
K  = zeros(nPt);
C  = zeros(nPt);
R  = zeros(nPt);
S1 = zeros(nPt);
S3 = zeros(nPt);
f  = zeros(nPt,1);
S2 = zeros(nPt,1);

S11 = zeros(nPt);
S12 = zeros(nPt);
S21 = zeros(nPt);
S22 = zeros(nPt);
S31 = zeros(nPt);
S32 = zeros(nPt);

% Loop on elements
for ielem=1:nElem
    Te = T(ielem,:);
    Xe = X(Te,:);
    Conve = Conv(Te,:);

    % Loop on Gauss points
    [Me,Ke,Ce,Re,fe,Se1,Se3,Se2,Se11,Se12,Se21,Se22,Se31,Se32] =
EleMat(Xe,Conve,data,nen,ngaus,wgp,N,Nxi,Neta,N2xi,N2eta,p);

    % Assembly
    M(Te,Te) = M(Te,Te)  + Me;
    K(Te,Te) = K(Te,Te)  + Ke;
    C(Te,Te) = C(Te,Te)  + Ce;
    R(Te,Te) = R(Te,Te)  + Re;
    S1(Te,Te)= S1(Te,Te) + Se1;
    S3(Te,Te)= S3(Te,Te) + Se3;
    f(Te)    = f(Te)     + fe;
    S2(Te)   = S2(Te)    + Se2;

    S11(Te,Te)= S11(Te,Te) + Se11;
    S12(Te,Te)= S12(Te,Te) + Se12;
    S21(Te,Te)= S21(Te,Te) + Se21;
    S22(Te,Te)= S22(Te,Te) + Se22;
    S31(Te,Te)= S31(Te,Te) + Se31;
    S32(Te,Te)= S32(Te,Te) + Se32;
```

THIS IS THE MAIN CORE OF THE PROGRAM WHERE ALL MATRICS ARE DEFINED.

THERE WERE LOTS OF MANIPULATIONS IN THIS PART OF THE CODE.

```
end


function [Me,Ke,Ce,Re,fe,Se1,Se3,Se2,Se11,Se12,Se21,Se22,Se31,Se32] =
EleMat(Xe,Conve,data,nen,ngaus,wgp,N,Nxi,Neta,N2xi,N2eta,p)

Me  = zeros(nen);
Ke  = zeros(nen);
Ce  = zeros(nen);
Re  = zeros(nen);
Se1 = zeros(nen);
Se3 = zeros(nen);
fe  = zeros(nen,1);
Se2 = zeros(nen,1);

Se11 = zeros(nen);
Se12 = zeros(nen);
Se21 = zeros(nen);
Se22 = zeros(nen);
Se31 = zeros(nen);
Se32 = zeros(nen);

nu      = data.nu;
sig     = data.sig;
src     = data.src;
method2 = data.method2;
method1 = data.method1;
ProblemType = data.ProblemType;
hx      = data.hx;
hy      = data.hy;
h       = data.h;
dt      = data.dt;

    % Loop on Gauss points
for ig = 1:ngaus
    N_ig        = N(ig,:);
    Nxi_ig      = Nxi(ig,:);
    Neta_ig     = Neta(ig,:);
    N2xi_ig     = N2xi(ig,:);
    N2eta_ig    = N2eta(ig,:);

    dxdxi       = Nxi_ig    * Xe(:,1);
    dxdeta      = Neta_ig   * Xe(:,1);
    dydxi       = Nxi_ig    * Xe(:,2);
    dydeta      = Neta_ig   * Xe(:,2);
    d2xd2xi     = N2xi_ig   * Xe(:,1);
    d2xd2eta    = N2eta_ig  * Xe(:,1);
    d2yd2xi     = N2xi_ig   * Xe(:,2);
    d2yd2eta    = N2eta_ig  * Xe(:,2);

    switch p
        case 1
            Jacob = [Nxi_ig*(Xe(:,1))   Nxi_ig*(Xe(:,2))
                Neta_ig*(Xe(:,1))   Neta_ig*(Xe(:,2))];
            res_v = [Nxi_ig ; Neta_ig];
         case 2
             Jacob = [dxdxi^2 , dydxi*dxdxi , d2xd2xi , dxdxi*dydxi , dydxi^2 ,
d2yd2xi ; ...
                 dxdeta*dxdxi , dydeta*dxdxi , dxdxi*dxdeta , dxdeta*dydxi ,
dydeta*dydxi , dydxi*dydeta ; ...
                 0 , 0 , dxdxi , 0 , 0 , dydxi ; ...
```

NEW JACOBIAN matrix for the 2nd order elements

```
                dxdxi*dxdeta , dydxi*dxdeta , dxdeta*dxdxi , dxdxi*dydeta ,
dydxi*dydeta , dydeta*dydxi ; ...
                dxdeta^2 , dydeta*dxdeta , d2xd2eta , dxdeta*dydeta , dydeta^2
, d2yd2eta ; ...
                0 , 0 , dxdeta , 0 , 0 , dydeta ];
            res_v = [N2xi_ig ; Nxi_ig.*Neta_ig ; Nxi_ig ; Nxi_ig.*Neta_ig ;
N2eta_ig ; Neta_ig];
    end

    res = Jacob\res_v;
    dvolu = wgp(ig)*det(Jacob);

    switch p
        case 1
            Nx = res(1,:);
            Ny = res(2,:);
            N2x = 0*res(1,:);
            N2y = 0*res(2,:);
        case 2
            Nx = res(3,:);
            Ny = res(6,:);
            N2x = res(1,:);
            N2y = res(5,:);
    end

    a_vec  = N_ig * Conve;
    ax = a_vec(1);
    ay = a_vec(2);
   a  =ax;
%    a  =(ax+ay)/2;
%  a  =sqrt(ax^2+ay^2);

    Pe = a   * h  / (2 * nu);
    Co = a   * dt / (h)     ;
    dd = nu  * dt / (h^2)   ;
    r  = sig * dt           ;

    aGradN = ax * Nx + ay * Ny;
    GradN2 = N2x + N2y;

    switch ProblemType
        case 2
            Ke = Ke + aGradN'* aGradN * dvolu;
        otherwise
            Ke = Ke + ( Nx' * Nx + Ny' * Ny) * dvolu * nu ;
    end
            Me = Me + N_ig' * N_ig   * dvolu;
            Ce = Ce + N_ig' * aGradN * dvolu;
            Re = Re + N_ig' * N_ig   * dvolu * sig;
            fe = fe + N_ig'          * dvolu * src;

    switch method2
        case 0          % Galerkin
            tau = 0;
        otherwise       % SUPG - GLS
             tau = ( (2*a/h)^2 + 9*(4*nu/(h^2))^2 + sig^2 )^(-1/2);
    end

    if method1==3
        switch method2
            case 0          % Galerkin
                tauT_fin = [0 , 0 ; 0 , 0];
```

Defining all matrics:

Convection

Diffusion

Reaction

Source

Mass

+ Stabilization

```
            otherwise       % SUPG - GLS
                W_inv= [5/2 , 1/2 ; -13/2 , 7/2];
                tauT = (1/dt) * W_inv + (2*a/h+4*nu/(h*h)+sig)*[1,0;0,1];
                tauT_inv=inv(tauT);
                tauT_inv_T=tauT_inv';
                tauT_fin=tauT_inv_T*W_inv;
        end
    end

    switch method2
        case 0              %Galerkin - No Stabilization
            Pw = 0*aGradN;
        case 1              %SUPG
            Pw = aGradN;
        case 2              %GLS
            Pw = aGradN - nu * GradN2 + sig * N_ig;
    end

    switch ProblemType
%-----------------------------------------------------------------ProblemType=1
        case 1          %   Steady conv. diff.
            Se1 = Se1 + Pw'* (aGradN - nu * GradN2 + sig * N_ig) * dvolu * tau;
            Se2 = Se2 + Pw'* src * dvolu * tau;
%-----------------------------------------------------------------ProblemType=3
        case 3          % Unsteady conv. diff.
            switch method1
%-------------------------------------------------------------R11
                case 1
                    Se1 = Se1 + Pw'* (N_ig + 0.5 * dt * (aGradN - nu * GradN2 +
sig * N_ig)) * dvolu * tau;
                    Se3 = Se3 + Pw'* (             dt * (aGradN - nu * GradN2 +
sig * N_ig)) * dvolu * tau;
                    Se2 = Se2 + Pw'* src * dvolu * tau;
%-------------------------------------------------------------R20
                case 2
                    Se1 = Se1 + Pw'* N_ig                                      *
dvolu * tau;
                    Se3 = Se3 + Pw'* dt * (aGradN - nu * GradN2 + sig * N_ig) *
dvolu * tau;
                    Se2 = Se2 + Pw'* src * dvolu * tau;
%-------------------------------------------------------------R22
                case 3
                    W11=7/24 ; W12=-1/24 ; W21=13/24 ; W22=5/24;
                    Se11 = Se11 + Pw'* ((1/W11)* N_ig + dt * (aGradN - nu *
GradN2 + sig * N_ig)) * dvolu * tau;
                    Se12 = Se12 + Pw'* (               dt * (aGradN - nu *
GradN2 + sig * N_ig)) * dvolu * tau;
                    Se21 = Se21 + Pw'* (               dt * (aGradN - nu *
GradN2 + sig * N_ig)) * dvolu * tau;
                    Se22 = Se22 + Pw'* ((1/W22)* N_ig + dt * (aGradN - nu *
GradN2 + sig * N_ig)) * dvolu * tau;
                    Se31 = Se31 + Pw'* (               dt * (aGradN - nu *
GradN2 + sig * N_ig)) * dvolu * tau;
                    Se32 = Se32 + Pw'* (               dt * (aGradN - nu *
GradN2 + sig * N_ig)) * dvolu * tau;
                    Se2  = Se2  + Pw'* src * dvolu * tau;
            end

    end

end
```

h. Boundary_matrices

```
function [Mout,Cout] = Boundary_matrices(X,T,Conv,referenceElement,data)

% calls function (OutflowBOundary) INSIDE itself
% calls function (ShapeFunc_aux)   INSIDE itself

velotype = data.velotype;

T_boundary = OutflowBOundary(X,T,velotype);

elem    = referenceElement.elem;
nen     = referenceElement.nen;
p       = referenceElement.degree;

% 1D quadrature
ngaus = 3;
zgp = [-sqrt(15)/5; 0; sqrt(15)/5];
wgp = [5/9   8/9   5/9];

nPt   = size(X,1);
nElem = size(T_boundary,1);

Mout = zeros(nPt);
Cout = zeros(nPt);

for i = 1:nElem                          % Loop on elements
    ielem = T_boundary(i,1);
    Te = T(ielem,:);
    Conve = Conv(Te,:);

    P1 = X(T_boundary(i,2),:);
    P2 = X(T_boundary(i,3),:);
    h = norm(P2 - P1);
    n = T_boundary(i,4:5);
    [N,Nxi,Neta] = ShapeFunc_aux(zgp,n,elem,p);

    Mout_e = zeros(nen);
    Cout_e = zeros(nen);
    for ig = 1:ngaus                     % Loop on gauss points
        N_ig    = N(ig,:);
        Nxi_ig  = Nxi(ig,:);
        Neta_ig = Neta(ig,:);

        dvolu = wgp(ig)*h/2;
        % Derivatives of the shape functions on global coordinates
        Nx = Nxi_ig * 2/h;
        Ny = Neta_ig* 2/h;
        % velocity on the Gauss point
        a  = N_ig * Conve;
        % normal component of the velocity at the Gauss point
        an = n*a';
        % Contribution to element matrix
        Mout_e = Mout_e + an*(N_ig'*N_ig)*dvolu;
        Cout_e = Cout_e + an*(N_ig'*(a(1)*Nx+a(2)*Ny))*dvolu;
    end

    Mout(Te,Te) = Mout(Te,Te) + Mout_e;
    Cout(Te,Te) = Cout(Te,Te) + Cout_e;
end
function T_boundary = OutflowBOundary(X,T,velotype)
```

```
nElem = size(T,1);
x1 = min(X(:,1)); x2 = max(X(:,1)); xM = (x1+x2)/2;
y1 = min(X(:,2)); y2 = max(X(:,2)); yM = (y1+y2)/2;
T_boundary = zeros(nElem,5);
ind = 1;

if velotype == 2
    for i = 1:nElem
        Te = T(i,:);
        Xe = X(Te,:);
        xElem = Xe(:,1);
        yElem = Xe(:,2);
        xx1 = abs(xElem-x1); aux_x1 = find(xx1 < 1e-6);
        xx2 = abs(xElem-x2); aux_x2 = find(xx2 < 1e-6);
        yy1 = abs(yElem-y1); aux_y1 = find(yy1 < 1e-6);
        yy2 = abs(yElem-y2); aux_y2 = find(yy2 < 1e-6);
        if length(aux_x1) == 2 && all(yElem(aux_x1) >= yM)
            T_boundary(ind,:) = [i, Te(aux_x1), -1, 0];
            ind = ind+1;
        end
        if length(aux_x2) == 2 && all(yElem(aux_x2) <= yM)
            T_boundary(ind,:) = [i, Te(aux_x2), 1, 0];
            ind = ind+1;
        end
        if length(aux_y1) == 2 && all(xElem(aux_y1) <= xM)
            T_boundary(ind,:) = [i, Te(aux_y1), 0, -1];
            ind = ind+1;
        end
        if length(aux_y2) == 2 && all(xElem(aux_y2) >= xM)
            T_boundary(ind,:) = [i, Te(aux_y2), 0, 1];
            ind = ind+1;
        end
    end
    T_boundary = T_boundary(1:ind-1,:);
elseif velotype == 1
    for i = 1:nElem
        Te = T(i,:);
        Xe = X(Te,:);
        xElem = Xe(:,1);
        xx2 = abs(xElem-x2); aux_x2 = find(xx2 < 1e-6);
        if length(aux_x2) == 2
            T_boundary(ind,:) = [i, Te(aux_x2), 1, 0];
            ind = ind+1;
        end
    end
    T_boundary = T_boundary(1:ind-1,:);
end

function [N,Nxi,Neta] = ShapeFunc_aux(zgp_1D,n,elem,p)
ngaus = length(zgp_1D);
if n(1) == 1
    zgp = [ ones(ngaus,1), zgp_1D];
elseif n(1) == -1
    zgp = [-ones(ngaus,1), zgp_1D];
elseif n(2) == 1
    zgp = [zgp_1D,  ones(ngaus,1)];
else
    zgp = [zgp_1D, -ones(ngaus,1)];
end
[N,Nxi,Neta] = ShapeFunc(elem,p,zgp);
```

i. Mat_combination

```
function [A,B,method1Name] =
Mat_combination(M,K,C,R,S1,S3,S11,S12,S21,S22,S31,S32,Mout,Cout,data,tauT_fin)

method1     = data.method1;
ProblemType = data.ProblemType;

dt      = data.dt;

switch ProblemType
%-------------------------------                                  -ProblemType=1
    case 1 % Steady case
        A = K + C + R + S1;
        B = 0;
        method1Name = 'ST';
%-----------------------------------------------------------------ProblemType=2
    case 2 % Crank-Nicolson + Galerkin
        A = M - 0.5 * dt * C' + 0.5 * dt * Mout;     % A = M + 0.5*dt*C;
        B =           dt * C' -       dt * Mout;     % B =         -dt*C;
        method1Name = 'TR';
%-----------------------------------------------------------------ProblemType=3
    case 3
        switch method1
%-------------------------------------------------------------------------R11
            case 1
                A = M + 0.5 * dt * (C + K + R) + S1;
                B =          -dt * (C + K + R) - S3;
                method1Name = 'R11';
%-------------------------------------------------------------------------R20
            case 2
                A = M + S1 ;
                B = -dt * (C + K + R) - S3;
                method1Name = 'R20';
%-------------------------------------------------------------------------R22
            case 3
                W11=7/24 ; W12=-1/24 ; W21=13/24 ; W22=5/24;
                J = C + K + R;
                A = [M + W11 * dt * J + W11 * S11 ,     W12 * dt * J + W12 *
S12;...
                         W21 * dt * J + W21 * S21 , M + W22 * dt * J + W22 *
S22];
                B = [-0.5 * dt * J - 0.5 * S31 ; -0.5 * dt * J - 0.5 * S32];
                method1Name = 'R22';
            otherwise
                error('not available method1')
        end
    otherwise
        error('not available ProblemType')
end

end
```

In this function all necessary matrices for each method of discretization and each method of stabilization are going to put together and consist the right hand side and left hand side matrices of global system.

j. BoundaryConditions

```
function [ADir,bDir,nDir,nodesDir1,boundr] = BoundaryConditions(X,data)

ProblemType = data.ProblemType;
examp       = data.examp;
velotype    = data.velotype;

x1 = min(X(:,1)); x2 = max(X(:,1)); xM = (x1+x2)/2;
y1 = min(X(:,2)); y2 = max(X(:,2)); yM = (y1+y2)/2;

nodes_x1 = find( (abs(X(:,1) - x1)<1e-6) );
nodes_x2 = find( (abs(X(:,1) - x2)<1e-6) );
nodes_y1 = find( (abs(X(:,2) - y1)<1e-6) );
nodes_y2 = find( (abs(X(:,2) - y2)<1e-6) );

nodes_x1_tophalf    = find( (abs(X(:,1) - x1)<1e-6) & (X(:,2)>=yM) );
nodes_x1_bothalf    = find( (abs(X(:,1) - x1)<1e-6) & (X(:,2)<=yM) );
nodes_x2_tophalf    = find( (abs(X(:,1) - x2)<1e-6) & (X(:,2)>=yM) );
nodes_x2_bothalf    = find( (abs(X(:,1) - x2)<1e-6) & (X(:,2)<=yM) );
nodes_y1_righthalf  = find( (abs(X(:,2) - y1)<1e-6) & (X(:,1)>=xM) );
nodes_y1_lefthalf   = find( (abs(X(:,2) - y1)<1e-6) & (X(:,1)<=xM) );
nodes_y2_righthalf  = find( (abs(X(:,2) - y2)<1e-6) & (X(:,1)>=xM) );
nodes_y2_lefthalf   = find( (abs(X(:,2) - y2)<1e-6) & (X(:,1)<=xM) );

boundr.nodes_x1              =nodes_x1;
boundr.nodes_x2              =nodes_x2;
boundr.nodes_y1              =nodes_y1;
boundr.nodes_y2              =nodes_y2;
boundr.nodes_x1_tophalf      =nodes_x1_tophalf;
boundr.nodes_x1_bothalf      =nodes_x1_bothalf;
boundr.nodes_x2_tophalf      =nodes_x2_tophalf;
boundr.nodes_x2_bothalf      =nodes_x2_bothalf;
boundr.nodes_y1_righthalf    =nodes_y1_righthalf;
boundr.nodes_y1_lefthalf     =nodes_y1_lefthalf;
boundr.nodes_y2_righthalf    =nodes_y2_righthalf;
boundr.nodes_y2_lefthalf     =nodes_y2_lefthalf;

switch ProblemType
%-------------------------------------------------------------------------
    case 1
        nx = data.nx;
        ny = data.ny;
        nodes_y00 = [1:nx+1]';                          % Nodes on the boundary
y=0
        nodes_x11 = [2*(nx+1):nx+1:(ny+1)*(nx+1)]' ;  % Nodes on the boundary
x=1
        nodes_y11 = [ny*(nx+1)+nx:-1:ny*(nx+1)+1]' ;  % Nodes on the boundary
y=1
        nodes_x00 = [(ny-1)*(nx+1)+1:-(nx+1):nx+2]';  % Nodes on the boundary
x=0
        if examp== 1
            nodesDir1 = nodes_x00( X(nodes_x00,2) > 0.2 );                 %
Dirichlet B.C. u=1
            nodesDir0 = [nodes_x00( X(nodes_x00,2) <= 0.2 ); nodes_y00];   %
Dirichlet B.C. u=0
        elseif examp== 2
            nodesDir1 = nodes_x00( X(nodes_x00,2) > 0.2 );
            nodesDir0 = [nodes_x00( X(nodes_x00,2) <= 0.2 ); nodes_y00;
nodes_x11; nodes_y11];
        elseif examp== 3 || examp==4
            nodesDir1 = [];
```

Based on each sample different B.C are introduced here

```
            nodesDir0 = [nodes_x00; nodes_y00; nodes_x11; nodes_y11];
        else
            nodesDir1 = nodes_x1_tophalf;
            nodesDir0 = nodes_x2;
        end
%-------------------------------------------------------------------------
    case 2
        if velotype==2
            nodesDir0 = unique([nodes_x1_bothalf; nodes_x2_tophalf;
nodes_y1_righthalf; nodes_y2_lefthalf]);
            nodesDir1 = [];
        elseif velotype == 1
            nodesDir0 = nodes_x2;
            nodesDir1 = nodes_x1_tophalf;
        end
%-------------------------------------------------------------------------
    case 3
        if velotype==2
            nodesDir0 = unique([nodes_x1; nodes_x2; nodes_y1; nodes_y2]);
%           nodesDir0 = unique([nodes_x1_bothalf; nodes_x2_tophalf;
nodes_y1_righthalf; nodes_y2_lefthalf]);
            nodesDir1 = [];
        elseif velotype == 1
            nodesDir0 = unique([nodes_x1_tophalf; nodes_x2]);
            nodesDir1 = [];
        end
%-------------------------------------------------------------------------
end

Dirval=data.Dirval;

C = [nodesDir1, Dirval*ones(length(nodesDir1),1);
     nodesDir0,       zeros(length(nodesDir0),1)];

nDir    = size(C,1);
nPt     = size(X,1);
ADir    = zeros(nDir,nPt);
ADir(:,C(:,1))  = eye(nDir);
bDir            = C(:,2);
end
```

k. InitialCondition

```
function u0 = InitialCondition(X,data,boundr)

nPt = data.nPt;
velotype=data.velotype;

if velotype==2
    sigma = 0.2;
    xref = 1/6;

    xdim = ( X(:,1) - xref) / sigma;
    ydim = ( X(:,2) - xref) / sigma;
    ind = find( (xdim.^2 + ydim.^2)<=1 );
    u0  = zeros(nPt,1);
    u0(ind) = 0.25*(1 + cos(pi*xdim(ind))).*(1 + cos(pi*ydim(ind)));
else
    u0 = zeros(nPt,1);

%     xref1 = max(X(:,1))/4.0;
%     xref2 = max(X(:,1))/4.0*3.0;
%     yref1 = max(X(:,2))/4.0;
%     yref2 = max(X(:,2))/4.0*3.0;
%     ind = find(X(:,1)>=xref1 & X(:,1)<=xref2 & X(:,2)>=yref1 & X(:,2)<=yref2
);
%     u0(ind,1) = 1.0;

    nodes_x1_tophalf = boundr.nodes_x1_tophalf;
    u0(nodes_x1_tophalf)=1.0;

end
```

2 options are available:

Impulse initial condition

A copy of boundary condition