INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

RICE UNIVERSITY

**OBJECT-ORIENTED IMPLEMENTATION OF
THE GALERKIN FINITE ELEMENT
METHOD AND ITS APPLICATION TO THE
NUMERICAL STUDY OF NATURAL
CONVECTIVE FLOWS IN ENCLOSURES**

by

Rafael Moreno

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Bala Ramaswamy

Dr. Ramaswamy, Balasubramaniam,
Chairman
Assistant Professor
Mechanical Engineering and Materials
Science

J C Akin

Dr. Akin, John E.
Professor
Mechanical Engineering and Materials
Science

Danny C. Sorensen

Dr. Sorensen, Danny C.
Professor
Computational and Applied Mathematics

Houston, Texas

June, 1996

UMI Number: 9727584

**Copyright 1997 by
Moreno, Rafael**

All rights reserved.

**UMI Microform 9727584
Copyright 1997, by UMI Company. All rights reserved.**

**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

Copyright

Rafael Moreno

1997

OBJECT-ORIENTED IMPLEMENTATION OF THE GALERKIN FINITE ELEMENT METHOD AND ITS APPLICATION TO THE NUMERICAL STUDY OF NATURAL CONVECTIVE FLOWS IN ENCLOSURES

Rafael Moreno

Abstract

Using object-oriented programming (OOP) techniques and philosophies, a collection of C++ tools for the rapid development of finite element applications has been created. The Object-Oriented Finite Element Analysis (OOFEA) toolkit provides both the geometrical and mathematical management tools necessary for this task in the form of useful class hierarchies. In particular, the OOFEA toolkit features methods for evaluating arbitrary weak forms provided by the user in order to solve particular problems of interest. A detailed description of the underlying concepts, philosophies and techniques used to develop the toolkit, as well as description of its contents and usage are included. A strong effort has been made to concentrate on its possibly beneficial usage in the computational fluid dynamics area. Hence, a number of sample CFD and heat transfer applications of increasing difficulty and interests are thoroughly discussed. Moreover, to demonstrate the toolkit capabilities of managing complex projects, a simulator for laminar and turbulent natural convective flows in enclosures has been developed and a numerical study of some of these flows has been conducted.

Using a primitive variable approach, the Galerkin FEM is used to obtain the weak form of the coupled unsteady Navier-Stokes and energy equations for incompressible, viscous, Newtonian fluids in two and three dimensions. By including a k - ϵ turbulence model in the governing equations, the analysis of both laminar and turbulent convective flows in enclosures is possible. With the help of a semi-implicit time stepping scheme, combined with a projection scheme, the resulting systems of equations are solved iteratively using the preconditioned conjugate gradient (PCG) algorithm.

Time accurate two-dimensional simulations have been performed for a differentially heated square cavity in the laminar and turbulent regimes, for air with a Prandtl number of 0.71, and values of the Rayleigh number ranging between 10^3 and 10^{10} . Consistency tests show that the simulator correctly implements the k - ϵ turbulence model, and the numerical results compare well with results reported in the literature. Furthermore, three-dimensional simulations have been performed for a differentially heated cubic cavity in the laminar regime for air with a Prandtl number of 0.71, and values of the Rayleigh number ranging from 10^3 to 10^6 . The results obtained compare well with other results in the literature, and characterize some of the three-dimensional effects that are ignored in two-dimensional simulations. In particular, it can be observed that the three-dimensional effects can change the predicted dimensionless heat transfer rate by as much as 10%.

Acknowledgments

I want to sincerely thank my thesis advisor Dr. Balasubramaniam Ramaswamy, for his guidance, advice, and encouragement during my years at Rice. I would also like to thank Dr. John E. Akin for his invaluable suggestions and hints which led to the final form of this dissertation, Dr. Danny C. Sorensen for his willingness to help in the time of need, and my wife Evelmarie for her seemingly unlimited love, patience and support, without which this work would have never been completed.

The financial support of Rice University, the Ford Foundation, and specially Dr. Richard A. Tapia, whom from day one was always there for me, will always be appreciated.

Final thanks to Mr. Sivaramakrishnan Krishnamoorthy, Mr. Alvaro A. Fernandez and Mr. Gregory S. Lind with whom I brainstormed more than once, and had many fruitful discussions which clarified the path to this, the goal of my graduate career.

Nomenclature

General symbols:

x^*, y^*, z^*	nondimensional coordinates
u^*	nondimensional velocity in the x^* direction
v^*	nondimensional velocity in the y^* direction
w^*	nondimensional velocity in the z^* direction
p^*	nondimensional pressure
t^*	nondimensional time
k^*	nondimensional turbulent kinetic energy
x, y, z	Cartesian coordinates
u	velocity in the x direction
v	velocity in the y direction
w	velocity in the z direction
p	pressure
t	time
k	turbulent kinetic energy
P_k^*	shear production rate of turbulent kinetic energy
G_k^*	buoyancy production rate of turbulent kinetic energy
$\bar{u}$	time-averaged velocity in the x direction
$\bar{v}$	time-averaged velocity in the y direction
$\bar{w}$	time-averaged velocity in the z direction

$\bar{p}$	time-averaged pressure
U	reference velocity
g	gravitational acceleration
W, H	representative domain length scales
Δx	mesh spacing in the x direction
Δy	mesh spacing in the y direction
Δz	mesh spacing in the z direction
Δt	time step
$C_{1\epsilon}, C_{2\epsilon}$	empirical constants for turbulence model
$C_{3\epsilon}, C_\mu$	empirical constants for turbulence model

Greek symbols:

θ^*	nondimensional temperature
θ_r	reference temperature
θ_h	maximum (hot) temperature
θ_c	minimum (cold) temperature
θ	temperature
$\bar{\theta}$	time-averaged temperature
α	thermal diffusivity
$\bar{\beta}$	thermal expansion coefficient
$\bar{\rho}$	average density
μ	dynamic viscosity
ν	kinematic viscosity
ν_t	eddy diffusivity
ν_t^*	nondimensional eddy diffusivity

ϵ	dissipation of turbulence
σ_θ	turbulent Prandtl number for θ
σ_k	turbulent Prandtl number for k
σ_ϵ	turbulent Prandtl number for ϵ

Subscripts:

i	i th entry of a vector quantity
j	j th entry of a vector quantity
$.t^*$	partial derivative w.r.t. t^*
$.i^*$	partial derivative w.r.t. x^* , y^* or z^*
$.j^*$	partial derivative w.r.t. x^* , y^* or z^*

Superscripts:

$-$	time-averaged value
$*$	nondimensional value

Nondimensional numbers:

Ra	Rayleigh number
Pr	Prandtl number
Nu	Nusselt number
Gr	Grashof number
A	Aspect ratio

Note: The symbols defined above are subject to alternation on occasion.

Table of Contents

Abstract	ii
Acknowledgments	iv
Nomenclature	v
List of Figures	xii
List of Tables	xviii
1 Object-oriented finite element analysis	1
1.1 Introduction	1
1.2 The object-oriented design paradigm - concepts and terminology . . .	2
1.3 Object-oriented programming (OOP) in C++	6
1.4 OOFEA in the literature	10
1.5 OOFEA in C++ - Abstractions	11
1.6 Mathematical management - Matrix algebra	13
1.6.1 The <code>matrix</code> hierarchy	14
1.6.2 The <code>sparse_matrix</code> hierarchy	21
1.7 Geometrical management	26
1.7.1 The <code>node</code> class	26
1.7.2 The <code>element</code> hierarchy	29
1.7.3 The <code>mesh</code> class	36
1.7.4 Supporting iterator classes	43
2 C++ OOFEA sample applications for CFD	47
2.1 Introduction	47

2.2	General structure of an OOFEA application	47
2.3	General structure of user-defined weak forms	48
2.4	Preconditioned conjugate gradient solver	49
2.5	Notes on the sample applications	51
2.6	Static heat transfer	51
2.7	Potential flow	57
2.8	Transient heat transfer	60
2.9	Linear scalar convection	66
2.10	Nonlinear convection	72
2.11	Nonlinear convection-diffusion	78
3	Introduction to Natural Convective Flows and Turbulence	86
3.1	Introduction	86
3.2	Natural convective flows	86
3.3	Unresolved problems of natural convection	88
3.4	Natural convection in enclosures	89
3.4.1	Laminar regime - Literature Survey	89
3.4.2	Turbulent regime - Theory	92
3.4.2.1	Reynolds Averaging	93
3.4.2.2	The Boussinesq assumption	96
3.4.2.3	Turbulence models using the Boussinesq assumption	98
3.4.2.4	The k - ϵ turbulence model	100
3.4.3	Turbulent regime - Literature Survey	102
4	Mathematical Models, Solution Algorithms and their OOFEA Implementation	105
4.1	Problem formulation and governing equations	105
4.2	Numerical procedure	110
4.2.1	Time discretization	110

4.2.1.1	Step 1 - Intermediate velocity calculations	110
4.2.1.2	Step 2 - Velocity corrections	111
4.2.1.3	Step 3 - Pressure calculations	111
4.2.1.4	Step 4 - Temperature, k and ϵ calculations	112
4.2.2	Spatial discretization	112
4.3	Solver implementation using the OOFEA classes and methods	117
5	Laminar and Turbulent Natural Convection in Square Cavities	121
5.1	Thermally driven cavity	121
5.2	Laminar simulations for Ra between 10^3 and 10^6	123
5.3	Consistency tests at $Ra = 10^6, 10^7$	133
5.4	Turbulent simulations for Ra between 10^8 and 10^{10}	135
6	Laminar Three Dimensional Natural Convection in a Cubic Cavity	149
6.1	Thermally driven cavity	149
6.2	Three dimensional results	149
6.2.1	Temperature and velocity profiles for $Ra = 10^4$	150
6.2.2	Temperature and velocity profiles for $Ra = 10^6$	151
6.3	Heat transfer characteristics	152
7	General Conclusions	156
7.1	Concluding remarks	156
7.2	Future work	157
A	Turbulent natural convection simulator	159
	Bibliography	178

List of Figures

2.1	Solution to a 2D static heat transfer problem where 4 heat sources are placed at several location in heated walls while the rest of the cavity convects the heat away.	57
2.2	Stream function pattern within a $5m \times 5m$ square enclosure with an inlet at the bottom horizontal surface and an outlet at the right vertical surface. The grid system is regular with 25×25 elements. . .	60
2.3	Temperature distribution of the solution to a 2D transient heat transfer problem where 4 heat sources are placed at several location in heated walls while the rest of the cavity convects the heat away to the environment.	65
2.4	Initial and final solution to the 1D scalar convection problem. The exact solution should be identical to the original waveform translated by the distance traveled in the elapsed time. The dispersion error is expected for the Semi-explicit leapfrog scheme. Here $\Omega = [0, 2]m$, $u = 200m/s$, $\Delta x = 2 \times 10^{-2}m$, $\Delta t = 2.5 \times 10^{-5}s$ and $c = u \frac{\Delta t}{\Delta x} = 0.25$	69
2.5	Initial distribution of a cosine hill of height $2m$ which is to travel across a rectangular domain of $33m \times 33m$ at a constant velocity $u = 0.7071\hat{i} + 0.7071\hat{j}m/s$ via a 2D scalar transport process.	72

- 2.6 Final distribution of the solution to the 2D first-order wave equation problem after 150 time steps with $\Delta t = 0.07071s$ and $\Delta x = 1m$, which yields a Courant number $c = 0.1$, using the 2nd-order Adams-Bashforth scheme. The solution has non-zero dispersion which can be observed as a wake behind the moving wave. The dispersion error is expected of this 2nd-order scheme. 73
- 2.7 Surface plot of initial distribution of a cosine hill of height $2m$ which is to travel across a rectangular domain of $33m \times 33m$ at a constant velocity $u = 0.7071\hat{i} + 0.7071\hat{j}m/s$ via a 2D scalar transport process. 73
- 2.8 Surface plot of final distribution of the solution to the 2D first-order wave equation problem after 150 time steps with $\Delta t = 0.07071s$ and $\Delta x = 1m$, which yields a Courant number $c = 0.1$, using the 2nd-order Adams-Bashforth scheme. The solution has non-zero dispersion which can be observed as a wake behind the moving wave. The dispersion error is expected of this 2nd-order scheme. 74
- 2.9 Initial and final solution to the 1D nonlinear convection (1D inviscid Burger's) problem. The exact solution should be identical to the original waveform translated by the distance traveled in the elapsed time. The solution is slightly dissipated, but there is no overshoot. This is expected for the two-step $\frac{2}{3}$ scheme being used. Here $\Omega = [0, 2]m$, $u_{max} = 1m/s$ at $t = 0$, $\Delta x = 2 \times 10^{-2}m$, $\Delta t = 5 \times 10^{-3}s$ and $\frac{\Delta t}{\Delta x} = 0.25$ 77

- 2.10 Exact and approximate solution to the 1D convection-diffusion (1D viscous Burger's) problem. The exact solution is of the form $-\frac{2 \sinh x}{\cosh x - e^{-t}}$. The solution is slightly dissipated, but this is expected for the two-step $\frac{1}{2}$ scheme being used. Here $\Omega = [-9, 9]$, $u = -\frac{2 \sinh x}{\cosh x - e^{-0.1}}$ at $t = 0.1$, $\Delta x = 2 \times 10^{-1}$ and $\Delta t = 1 \times 10^{-2}$ 81
- 2.11 Initial velocity distribution of an unit height discontinuity which is to be transported across the rectangular domain via a 2D convection-diffusion process. 84
- 2.12 Final velocity distribution of the solution to the 2D convection-diffusion problem after 150 time steps with $\Delta t = 0.125$ and a kinematic viscosity $\nu = 0.2582$ using the two-step $\frac{1}{2}$ scheme. . . 84
- 2.13 Surface plot of initial velocity of an unit height discontinuity which is to be transported across the rectangular domain via a 2D convection-diffusion process. 85
- 2.14 Surface plot of final velocity of the solution to the 2D convection-diffusion problem after 150 time steps with $\Delta t = 0.125s$ and a kinematic viscosity $\nu = 0.2582m^2/s$ using the two-step $\frac{1}{2}$ scheme. 85
- 5.1 From top left to center bottom: Streamlines, vorticity contours, pressure contours, isotherm pattern and velocity vector plot for the laminar case of $Ra = 10^3$ 125
- 5.2 From top left to center bottom: Streamlines, vorticity contours, pressure contours, isotherm pattern and velocity vector plot for the laminar case of $Ra = 10^4$ 126

5.3	From top left to center bottom: Streamlines, vorticity contours, pressure contours, isotherm pattern and velocity vector plot for the laminar case of $Ra = 10^5$	128
5.4	From top left to center bottom: Streamlines, vorticity contours, pressure contours, isotherm pattern and velocity vector plot for the laminar case of $Ra = 10^6$	129
5.5	From left to right: Velocity profiles for the dimensionless vertical and horizontal velocities of the flow at the horizontal and vertical centerlines of the cavity ($y^* = 0.5$ and $x^* = 0.5$) when $Ra = 10^3$	130
5.6	From left to right: Velocity profiles for the dimensionless vertical and horizontal velocities of the flow at the horizontal and vertical centerlines of the cavity ($y^* = 0.5$ and $x^* = 0.5$) when $Ra = 10^4$	131
5.7	From left to right: Velocity profiles for the dimensionless vertical and horizontal velocities of the flow at the horizontal and vertical centerlines of the cavity ($y^* = 0.5$ and $x^* = 0.5$) when $Ra = 10^5$	131
5.8	From left to right: Velocity profiles for the dimensionless vertical and horizontal velocities of the flow at the horizontal and vertical centerlines of the cavity ($y^* = 0.5$ and $x^* = 0.5$) when $Ra = 10^6$	132
5.9	Comparison between laminar and turbulent simulations at $Ra = 10^6$	135
5.10	Comparison between laminar and turbulent simulations at $Ra = 10^7$	136
5.11	Distribution of the dimensionless vertical velocity and the dimensionless turbulence kinetic energy in the horizontal centerline of the cavity in the region next to the left isothermal wall for the cases of $Ra = 10^6$ (top) and $Ra = 10^7$ (bottom).	140

5.12	Distribution of the dimensionless vertical velocity and the dimensionless turbulence kinetic energy in the horizontal centerline of the cavity in the region next to the left isothermal wall for the cases of $Ra = 10^9$ (top) and $Ra = 10^{10}$ (bottom).	141
5.13	Distribution of the dimensionless vertical velocity and the dimensionless turbulence kinetic energy in the horizontal centerline of the cavity in the region next to the left isothermal wall for the case of $Ra = 10^8$ when the a 33×33 mesh is used (top) and a 51×51 mesh is used (bottom).	141
5.14	Transient recirculation near the adiabatic walls of the cavity as observed during the time-accurate simulation of turbulent thermally driven cavity flows.	142
5.15	Transient counterclockwise rotating cell within the core of the flow as observed during the time-accurate simulation of turbulent thermally driven cavity flows.	143
5.16	From top left to bottom right: Streamlines, isotherms, pressure contours and vector plot for the turbulent solution of the thermally driven cavity flow at $Ra = 10^9$	144
5.17	From left to right: Velocity profiles for the dimensionless vertical and horizontal velocities of the flow at the horizontal and vertical centerlines of the cavity ($y^* = 0.5$ and $x^* = 0.5$ for the turbulent case with $Ra = 10^9$	146
5.18	From top left to bottom center: Contour plots of the nondimensional turbulence kinetic energy k^* , the dimensionless rate of dissipation of turbulence kinetic energy ϵ^* , and the nondimensional turbulence viscosity for the turbulent case with $Ra = 10^9$	147

5.19	Distribution of the Nusselt number over the vertical isothermal left wall of the cavity for the turbulent case with $Ra = 10^9$	148
6.1	Top left: temperature contours inside the cavity for $Ra = 10^4$. Contour levels: A) 0.9667, B) 0.9833, C) 1.0, D) 1.017, E) 1.033. Top right to bottom right: isovelocity contours for u -velocity, v -velocity and w -velocity components respectively for $Ra = 10^4$. Contour levels for u : A) -0.1333, B) -0.06667, C) 0.06667, D) 0.1333. Contour levels for v : A) -0.15, B) -0.075, C) 0.075, D) 0.15. Contour levels for w : A) -0.01667, B) -0.008333, C) 0.008333, D) 0.01667.	154
6.2	Top left: temperature contours inside the cavity for $Ra = 10^6$. Contour levels: A) 0.9667, B) 0.9833, C) 1.0, D) 1.017, E) 1.033. Top right to bottom right: isovelocity contours for u -velocity, v -velocity and w -velocity components respectively for $Ra = 10^6$. Contour levels for u : A) -0.12, B) -0.06, C) 0.06, D) 0.12. Contour levels for v : A) -0.2, B) -0.1, C) 0.1, D) 0.2. Contour levels for w : A) -0.024, B) -0.012, C) 0.012, D) 0.024.	155

List Of Tables

1.1	Sample listing demonstrating the concept of a class	3
1.2	Sample listing demonstrating the use of the private and public keywords within a class declaration.	7
1.3	Sample listing demonstrating the use of constructors and destructors within a class declaration.	9
1.4	Listing of the base_matrix base class.	15
1.5	Listing of the matrix derived class.	16
1.6	Listing of the temp_matrix derived class.	17
1.7	Supported overloaded operators and operations for full matrix algebra via the matrix class hierarchy. M1 and M2 each represent a matrix or temp_matrix instance, while s1 represents a scalar, and i and j represent integers.	17
1.8	Sample listing demonstrating the use of the matrix class hierarchy. .	20
1.9	Comparison of cost of linear algebra operations when only the matrix class is used and when it is used together with the temp_matrix class.	21
1.10	Listing of the base_sparse_matrix base class.	22
1.11	Listing of the sparse_matrix derived class.	24
1.12	Listing of the temp_sparse_matrix derived class.	24

1.13	Supported overloaded operators and operations for full matrix algebra via the <code>matrix</code> class hierarchy. M1 and M2 each represent a <code>sparse_matrix</code> or <code>temp_sparse_matrix</code> instance, while C1 represents a column <code>matrix</code> or <code>temp_matrix</code> , and s1 represents a scalar.	25
1.14	Additional functions supported by the <code>matrix</code> and <code>sparse_matrix</code> class hierarchies.	26
1.15	Listing of the <code>node</code> class.	27
1.16	Listing of the <code>element</code> base class.	30
1.17	Listing of the <code>domain_element</code> derived class.	33
1.18	Listing of the <code>flux_boundary_element</code> derived class.	34
1.19	Listing of the <code>convective_boundary_element</code> derived class.	35
1.20	Listing of the <code>prescribed_boundary_element</code> derived class.	35
1.21	Listing of the <code>mesh</code> class.	38
1.22	Listing of the <code>Nodes_of_Element</code> iterator class.	44
1.23	Listing of the <code>Nodes_of_Mesh</code> iterator class.	44
1.24	Listing of the <code>Domain_Elements_of_Mesh</code> iterator class.	44
1.25	Listing of the <code>Prescribed_Elements_of_Mesh</code> iterator class.	45
1.26	Sample listing of code which uses all iterator classes.	46
2.1	General structure of an user-defined weak form using the OOFEA toolkit.	49
2.2	Pseudocode for the Preconditioned Conjugate Gradient Iterative Algorithm.	49
2.3	Listing of the preconditioned conjugate gradient algorithm implementation using the OOFEA toolkit.	51
2.4	Listing of a 2D static heat transfer simulator using the OOFEA toolkit.	56

2.5	Listing of a 2D potetial flow simulator using the OOFEA toolkit. . .	59
2.6	Listing of a 2D transient heat transfer simulator using the OOFEA toolkit.	64
2.7	Listing of the Crank-Nicolson algorithm developed using the OOFEA toolkit.	65
2.8	Listing of a 1D linear convection simulator using the OOFEA toolkit.	68
2.9	Listing of a 2D linear convection simulator using the OOFEA toolkit.	70
2.10	Listing of the Semi-explicit leapfrog algorithm developed using the OOFEA toolkit.	70
2.11	Listing of the Adams-Bashforth algorithm developed using the OOFEA toolkit.	71
2.12	Listing of the explicit Euler algorithm developed using the OOFEA toolkit.	71
2.13	Listing of a 1D nonlinear convection (1D inviscid Burger's equation) simulator using the OOFEA toolkit.	76
2.14	Listing of the nonlinear two-step $\frac{2}{3}$ scheme using the OOFEA toolkit.	77
2.15	Listing of the nonlinear explicit Euler scheme using the OOFEA toolkit.	78
2.16	Listing of a 1D convection-diffusion (1D viscous Burger's equation) simulator using the OOFEA toolkit.	80
2.17	Listing of a 2D convection-diffusion (2D viscous Burger's equation) simulator using the OOFEA toolkit.	83
5.1	Comparison of the present numerical results (PRE) and the bench mark solutions by de Vahl Davis (DVD).	134

5.2	Quantitative results obtained from the simulation of turbulent flow inside a thermally driven cavity at a Rayleigh number of 10^9 . The variables in the table are the nondimensional variables even though the * notation is not being used.	145
6.1	Comparison of representative numerical results in the present study (PRS) and the solutions by Fusegi <i>et al.</i> (FUS) for the symmetry xy plane at $z = 0.5$	153
A.1	Listing of the K weak form using the OOFEA toolkit.	159
A.2	Listing of the K_m weak form using the OOFEA toolkit.	159
A.3	Listing of the K_t weak form using the OOFEA toolkit.	160
A.4	Listing of the K_k weak form using the OOFEA toolkit.	160
A.5	Listing of the K_ε weak form using the OOFEA toolkit.	161
A.6	Listing of the M weak form using the OOFEA toolkit.	161
A.7	Listing of the P weak form using the OOFEA toolkit.	161
A.8	Listing of the D_DX weak form using the OOFEA toolkit.	162
A.9	Listing of the D_DY weak form using the OOFEA toolkit.	162
A.10	Listing of the EK weak form using the OOFEA toolkit.	162
A.11	Listing of the G_k weak form using the OOFEA toolkit.	163
A.12	Listing of the G_ε weak form using the OOFEA toolkit.	163
A.13	Listing of the PKU_k weak form using the OOFEA toolkit.	164
A.14	Listing of the PKV_k weak form using the OOFEA toolkit.	165
A.15	Listing of the PKU_ε weak form using the OOFEA toolkit.	165
A.16	Listing of the PKV_ε weak form using the OOFEA toolkit.	166

A.17 Listing of user input and mesh and parameter initialization section using the OOFEA toolkit.	167
A.18 Listing of the data initializatin and time step determination sections of the simulator.	169
A.19 Listing of the momentum, pressure and energy solution steps in the simulator.	171
A.20 Listing of the k - ϵ solution steps of the simulator including the determination and update of boundary conditions.	174
A.21 Listing of the intermediate and final post-processing sections of the simulator.	177

Chapter 1

Object-oriented finite element analysis

1.1 Introduction

Languages such as FORTRAN, Pascal and C are what are called **non-structured** and **structured** programming languages. A common programming philosophy in these languages is **procedural programming**. Procedural programming is characterized by the concept that data structures and functions are separate entities. A function can operate on many different data structures, while a data structure might have many different functions acting upon it. In this type of programming, the implementation of the functions tend to depend on the intrinsic characteristics of the data structure, i.e. changes in the data structures in question tend to necessitate change in the functions that operate on them. This tendency implies a certain level of global interdependency between all components of a particular application which by itself is not a necessary weakness unless the code in question has to change to accommodate new or changing user and design requirements.

When an application needs to evolve, global interdependencies can create such a rigid framework, that change becomes costly and sometimes unmanageable. In this sense, procedural languages might violate the complexity vs. generality principle: a costly and complex code change to accommodate new features will not necessarily yield a more general and flexible code. Fortunately, procedural programming is not the only programming philosophy, although it might be the first one scientists and engineers are formally exposed to. In particular, a different programming paradigm has

emerged from its relative isolation in the past decade. **Object-oriented programming (OOP)**, by virtue of its goals, is capable of overcoming some of the limitations of procedural languages, such as managing complexity and ensuring generality. In the next few sections, the concepts and terminology of OOP will be introduced, as well as how those concepts are present in the most popular of object-oriented languages: C++.

1.2 The object-oriented design paradigm - concepts and terminology

First, it has been said that object-oriented design is an engineering discipline whose goal is to manage dependencies within a program. This goal is achieved by partitioning the application being developed into self-contained pieces of code of manageable size called **objects**. In fact, an object-oriented application can be defined as a set of objects working together to accomplish a task. Each object is then provided with an **interface** which permits exchange of information between objects. In this manner, the internal representation, state and operations of a particular object are **encapsulated** and **hidden** from any other object. In this way, object-oriented design accomplishes an inversion of dependencies when compared to procedural languages: the implementation of a particular object is irrelevant for a set of them to interact; the only requirement is that their interfaces are fully functional (i.e. they actually do reflect and affect the state and behavior of the object). In this fashion, it is clear that by carefully designing an object and its interface, internal changes in its representation, mechanics, states, data structures, etc. do not affect its interactions with other objects. Hence, global interdependencies in an application are eliminated from the onset of development.

- **Objects** - Where do these objects come from? How are they defined? First, the developer determines the requirements and concepts that describe the problem to be solved. Then, by the mental process of **abstraction**, the designer represents these characteristic concepts as software objects with particular behaviors and properties.
- **Classes** - Since a particular application usually requires the interaction between similar or even identical objects, object-oriented design allows the developer to design a general specification for an object in a form similar to that of an user-defined type. This mechanism is known as a **class**. A class declaration establishes a new type of variable. The class serves as a description from which an object can be created. Every object created from a class is called an **instance** of the class. The process of object creation from a class declaration is called **instantiation**. Classes arise from the developer's abstraction of the problem in question. For example, an engineer interested in solving linear algebra problems realizes that one the main concepts is that of a matrix. Hence, he can abstract the concept of a matrix such that he creates a matrix class from which matrix objects can be instantiated for later manipulation:

```
class matrix;
matrix small_matrix, big_matrix;
```

Table 1.1: Sample listing demonstrating the concept of a class.

Here, `small_matrix` and `big_matrix` are two distinct objects which are instances of the `matrix` class.

- **Class or object data** - The properties that characterize an object are what are called the data. The data is defined in the class declaration and instances of the class contain the data. The data can be variables of primitive as well

as user-defined types. In particular, they can be instances of other classes, i.e. objects. In particular, if a class A defines a data member which is an instance of another class B, then it is said that class A is a **client** of class B.

- **Class or object methods** - After the designer decides what is the necessary data to describe a class of objects, a set of **methods** or **member functions** is created which becomes the interface to the objects created from the class. These methods, like the data, are included in the class declaration and like the data, are part of the objects which are instances of the class. Notice that the fact that the functions are declared within the class body relates them with the objects created from the class. Normal functions are completely independent entities in procedural languages. However, in object-oriented languages they can be related to a particular class of objects (they do not have to be). The design of the interface is a crucial part of the object-oriented design paradigm. Careless interface design can expose the internal workings of the objects, violating data hiding principles, or even worse, make the interface directly independent on the implementation, violating encapsulation principles. In both cases, global interdependencies are introduced that can cost the developer dearly when the application needs to evolve. Nonetheless, careful design can overcome these problems at the cost of some extra development time.
- **Inheritance** - The process of abstraction used by the designer can involve many levels of resolution, from the very general to the very specific. Objects at one level of resolution can represent many objects at other levels of resolution. For example, a red notebook and a blue notebook are both notebooks, identical objects, unless one considers color as a differentiating characteristic. Invariably though, it is clear that both notebooks share all their characteristic behavior and

properties except for color. The object-oriented paradigm recognizes this fact, and provides a mechanism to exploit such close similarities between objects.

In addition to the concept of a class, the concept of **inheritance** is key to understand object-oriented design. By the process of inheritance, a class can be **derived** from another in such a way that all the behavior from the parent or generating class is conserved and present in the child or inheriting class without need for redefinition. Only new data and new methods need to be defined. For example, a class `notebook` is created in such a way that all the properties of a notebook are taken into account (length, width, number of pages). Methods for manipulating the notebook are also defined (current page number, pages left, go to next page, etc.). A new class `red_notebook` can be declared in a particular way which expresses that it is a child of the `notebook` class. In that way, all the data and methods of the `notebook` class are inherited by `red_notebook`. At that point, only the new data (color) and the new methods (what is your color?, set color to, etc.) have to be declared. Inheritance is an extremely powerful concept which allows the development of **class hierarchies**: collections of classes interrelated via inheritance which share data and methods among them.

- **Polymorphism** - It is conceivable that several different classes of objects have similar or identical interface methods (both notebooks and flowers have color, hence they conceivably could have the need for a “what is your color?” method.) In common procedural languages, functions or procedure names are unique. This is a reflection of the independence of data and functions. However, in object-oriented languages, data and methods can be related in an object. In particular, object-oriented languages allow the use of identical function names

as long as the ambiguities can be resolved via the context. Hence, functions having the same name but acting on different arguments are interpreted differently and are valid. Similarly, functions with the same name and same arguments, but related to different classes are also valid. This process is known as **polymorphism**. Polymorphism is extremely useful in particular when large class hierarchies are developed since with it a small number of function names can be used to manipulate many different but related objects.

The concepts presented above are the essentials of object-oriented design. Different object-oriented languages can and do implement these concepts in different ways, but the functionality is similar.

1.3 Object-oriented programming (OOP) in C++

Smalltalk, Ada, Objective-C, Eiffel and C++ are some of the currently available object-oriented programming languages. C++ stands today as the most popular of all. Originally called “C with Classes” it was developed by Bjarne Stroustrup working at Bell Laboratories at Murray Hill, New Jersey in 1980. The name C++ was coined later in 1983 [1].

As the original name suggests, C++ represents the combination of two powerful programming methods: procedural and object-oriented. In that sense, C++ makes certain compromises to be a superset of the C computer language. Some critics indicate that C++ lacks some features of “true” object-oriented languages. However, the features in question are controversial and non-essential for the design and implementation of true object-oriented applications. Moreover, C++ has been described as popular, readily available, portable, efficient, general, capable of enforcing code correctness, and capable of supporting legacy code [2]. Since most engineers and scientists are usually introduced to computing via procedural languages, C++ allows

them to build upon their previous knowledge, permitting a smoother programming paradigm shift. For these and many other reasons, C++ has been selected as the language for the implementation of an object-oriented finite element analysis system.

C++ supports all of the object-oriented essentials presented in section 1.2. However, to understand the C++ implementations that will be introduced in the next section, it is necessary to describe how some of those concepts are achieved in C++. These definitions will clarify some of the details to be presented later.

- **The private and public keywords** - These are two ways in which C++ achieves encapsulation. In a class declaration, the data and methods can either be `private` or `public`. By default, all items declared within the class body are `private`. Let us examine a typical class declaration:

```
class point
{
private:
... //Other data or methods can go here
double x_coordinate;
... //Other data or methods can go here
public:
... //Other data or methods can go here
double get_x_coordinate(void){return x_coordinate;};
void set_x_coordinate(double x){x_coordinate=x;};
... //Other data or methods can go here
};

int main(void)
{
point the_center, the_corner;
double center_x=3.0,corner_x=1.0;
the_center.set_x_coordinate(center_x);
the_corner.set_x_coordinate(corner_x);
center_x=the_center.get_x_coordinate();
corner_x=the_corner.get_x_coordinate();
return 0;
}
```

Table 1.2: Sample listing demonstrating the use of the private and public keywords within a class declaration.

The fact that `x_coordinate` is declared `private` means that it cannot be accessed by any function that is not a method of the class `point`. Making `get_x_coordinate()` and `set_x_coordinate()` public means that any part of the program can access these methods. Hence, these public methods are the interface of any `point` objects subsequently instantiated. This is the way C++ allows the user to define the object's interface and the object's data. Notice that one can access the individual members of the class using the single dot character `.`, which is called the **member access operator**.

- **Function and operator overloading** - These are two important ways in which C++ achieves polymorphism. Simply stated, two or more functions can share the same name as long as the parameters in their declarations are different. When this occurs, the functions sharing the same name are said to be **overloaded**. In a similar fashion, a set of about forty (40) C++ operators can be overloaded such that they take on an additional meaning with respect to a certain class. This is accomplished by including a redefinition of the operator within a class declaration. Notice that there is no provision for user-defined operators, i.e. no new operators can be created. In addition, neither the precedence nor the unary or binary character of an operator can be changed. Realize that these limitations are justified in order to manage the complexity of the applications to be generated.
- **Constructors and destructors** - When a variable is declared as a primitive type, the user must initialize it before use. The C and C++ language take care of destroying the variable once the procedure or function it belongs to ends. In a similar fashion, objects require initialization before use, and destruction when they are no longer needed. In C++, these ideas are accomplished by two special

functions that have no return type: the **constructor** and the **destructor**. The constructor of a class is a method of the class which has the same name as the class. The destructor, on the other hand, is a method of the class that has the name of the class preceded by the `~` character.

An object's constructor function is called when the object is instantiated, and it performs the tasks specified by the user, which usually involve initialization of the object's data. The object's destructor function is called when the object in question is no longer needed. For local objects it means that they are destroyed when the function they belong to is left. For global objects, it means that they are destroyed when the main program is left. The destructor also performs the tasks specified by the user, which in general concern deallocation of dynamic memory.

Destructors are called implicitly by the compiler when the object is no longer needed, so there is no need for explicitly invoking a destructor. Additionally, if constructors and destructors are not defined, a simple `public` default with no arguments is provided by the compiler. However, be aware that these defaults are only adequate in the simplest of cases. Example:

```
class matrix
{
private:
    long rows;
    long columns;
    double* entries;
public:
    matrix(){rows=0;columns=0;entries=new double[rows*columns];};
    ~matrix(){delete [] entries;};
};
```

Table 1.3: Sample listing demonstrating the use of constructors and destructors within a class declaration.

- **virtual functions** - This is another way in which C++ achieves polymorphism. Functions that are **virtual** appear in the context of inheritance. A class can declare a method as **virtual** by preceding the declaration with the keyword **virtual**. Each class derived from it can then implement the function differently. Then, depending on which kind of object calls the method, the correct version is used. Examples of **virtual** functions will arise later in our discussion.
- **friend functions** - There are situations in which two classes must share the same function for the sake of efficiency. In those cases, the function might want to access the private parts of both classes without using their interface. This is done by declaring the function within the class body with the keyword **friend** preceding it. Examples of **friend** functions will arise later in our discussion.

1.4 OOFEA in the literature

A limited number of papers in the literature identify object-oriented design as a technique from which finite element analysis can significantly benefit. Among these efforts, some concentrate in describing the basics [3][4][5][6], as well as the details [7][8] of object-oriented finite element analysis, including supporting tools such as linear algebra classes [9][10]. However, most if not all of them concentrate on applications in the structural mechanics area. Some of these include the static analysis of plasticity [11], interlaminar stress analysis [12], and the modeling and analysis of truss structures [13]. Furthermore, many implementations are not in C++, but on less popular languages such as Smalltalk and Objective-C. Little if no literature can be found on the use of OOFEA in C++ for CFD applications although object-oriented techniques are being used in commercial products at least for the design of graphic users interfaces (GUIs).

For these reasons, the next sections describe in detail the components of an OOFEA system in C++ to be used to simplify the development of CFD applications. Later sections will demonstrate the flexibility and generality of these tools via example applications, culminating with the development of a turbulent natural convection simulator.

1.5 OOFEA in C++ - Abstractions

At the first level of abstraction of an OOFEA toolkit, two different but related issues have to be addressed: geometrical management and mathematical management. First of all, FEA involves the geometrical description of a domain of interest. The domain is discretized into well defined partitions called finite elements. Every finite element contains a collection of control points called nodes which may or may not coincide with the element's vertices. The collection of all the finite elements is usually called a mesh or grid. In general, the domain discretization is accomplished by the process of grid generation. At the same time, a mathematical description in terms of partial differential equations (PDEs) and boundary and/or initial conditions is provided such that it is valid in the domain. A weak form of the governing equations is obtained by multiplying the PDEs by suitable weighting functions and integrating over the domain. These global integrals are then transformed into summations over the domain of local integrals for each finite element. By assuming a particular behavior of the solution to the PDEs within every finite element, the local integrals are evaluated either analytically or numerically. In either case, a system of equations in matrix form is obtained for each finite element. These local systems are then assembled into a global system of equations for the domain based on the geometrical connectivity of the finite elements within it. The system is then solved with an appropriate numerical

scheme, and an approximate solution for the governing PDEs is obtained over the domain. In practice these schemes are also expressed in matrix form.

From this analysis, an argument can be made that to manage the mathematical requirements of these finite element applications a set of classes supporting matrix algebra would be extremely useful. On the other hand, further analysis is still necessary to determine what is required to manage the geometrical aspects of the application.

The geometrical definition of the domain of interest in the FEM is usually comprised of both a coordinate table and a connectivity table. The coordinate table is simply a list of coordinates for every node in every finite element in the domain mesh. The connectivity table is a list which indicates which nodes correspond to which elements. In this fashion, every node and every element in the mesh receives a particular global ordering. In a procedural implementation, it is common to abstract the FE mesh directly into the coordinate and connectivity tables, and store them in the form of multidimensional arrays. However, in an object-oriented implementation, the driving force is to abstract the problem in such a way that the important and characterizing entities of the problem at hand are represented directly and provided with interfaces that mimic their actual behavior. Then, it is reasonable to say at this point that a possible object-oriented design solution is to create classes which represent nodes, elements and meshes. Furthermore, it is obvious from the previous description that elements are not a type of node but contain nodes, and meshes are not type of nodes or elements but contain elements. Notice that, even at this superficial level of abstraction, it is clear that an element class should be a client of a node class, and a mesh class should be a client of an element class (and thus indirectly a client of a node class). This solution immediately provides encapsulation of the main concepts necessary for geometry management in a FE application.

Independently of how an **element** object organizes and stores its component **node** objects, and how a **mesh** object organizes and stores its **element** and **node** objects (these are issues related to the data organization and should be hidden), a mechanism for transversing the particular data structure at will is necessary. Such a mechanism is called an **iterator**. Iterator classes can be developed that provide a simple and common interface with which to transverse data structures. It is therefore suggested that three different types of iterator classes should be developed: an iterator class to transverse nodes within elements, an iterator class to transverse elements within a mesh, and an iterator class to transverse nodes in a mesh directly.

At the deepest level of abstraction, there is a need to identify and decide how each of the classes that encapsulates the FEA concepts should behave, what kind of data should they contain and what kind of interface should they posses. At this level we are concerned with information hiding. The core of this level of abstraction is the actual implementation of the encapsulating classes and therefore, can be quite involved. The goal of the next several sections is to describe these implementations in detail.

1.6 Mathematical management - Matrix algebra

FEA is an engineering technique which, as previously mentioned, makes extensive use of linear algebra to describe and accomplish the solution of complex physical problems. FEA involves two (2) major types of matrices: small full matrices to describe and perform some of the mathematical operations required at the element level, and large sparse matrices which arise as the result of the assembly process from the element level to the domain level. Both types of matrices need to be manipulated efficiently due to the large number of operations they are involved with in FE applications. Furthermore, due to the large number of elements involved in a typical

FE application, efficient memory management is also necessary. This usually involves keeping data sequentially in memory, but more importantly, it involves designing a matrix class hierarchy capable of minimizing storage of temporary calculations as well as of reducing expensive copy operations.

Languages such as FORTRAN, Pascal or C do not provide facilities for direct manipulation of linear algebra entities such as vectors and matrices. Only interpreted tools such as MATLAB, MATHCAD, MATHEMATICA, and newer languages such as FORTRAN-90 provide such facilities directly. However, OOP languages such as C++ provide the engineer with the necessary framework for the development of linear algebra tools which can be accessed by the user in a manner almost identical to their written counterparts. It is via the creation of related classes or class hierarchies and the extensive use of operator overloading that C++ can be used to accomplish these mathematical management tasks.

1.6.1 The matrix hierarchy

Tables 1.4, 1.5 and 1.6 describe a C++ implementation of a `matrix` class hierarchy. Its base or parent class is called `base_matrix` and it implements the basic components and behavior of full matrices.

```
class base_matrix
{
private:
public:
    long rows;
    long columns;
    char* name;
    double* entries;
    //CONSTRUCTORS
    base_matrix(void);
    //DESTRUCTORS
    ~base_matrix(void);
    //FUNCTIONS
    void set_name(char* new_name);
    double* get_entries(void);
}
```



```

long get_columns(void);
long get_rows(void);
double absmax(void); //Entry with maximum absolute value
double sumabs(void); //Sum of absolute values of all entries
friend ostream& operator << (ostream& s, base_matrix& any_matrix);
};

```

Table 1.4: Listing of the `base_matrix` base class.

Its four (4) data members consist of the number of rows and columns, a name tag for display and/or debugging purposes, and a dynamic sequential one-dimensional array of entries. Only one default constructor and destructor are provided, and only seven (7) methods are implemented. The `base_matrix` class is only meant to handle the intrinsic properties of a full matrix. On the other hand, the derived classes `matrix` and `temp_matrix` are the core of the full matrix implementation.

The derived `matrix` class inherits all its data from the `base_matrix` class and consists of twenty-nine (29) methods: seven (7) control methods, seven (7) constructors, a destructor and fourteen (14) overloaded operators.

```

class matrix:public base_matrix
{
private:
public:
//FUNCTIONS
void show(void);
void show(long r_start, long r_end, long c_start, long c_end);
void load(char *s);
void save(char *s);
void copy(const matrix& m);
void resize(long r, long c);
void allocate(long r, long c);
//CONSTRUCTORS
matrix(void); //Alternate constructor
matrix(char *s); //Alternate constructor
matrix(long r, long c); //Alternate constructor
matrix(long r, long c, double val); //Alternate constructor
matrix(long r, long c, char *s); //Alternate constructor
matrix(long r, long c, double val, char *s); //Alternate constructor
matrix(const matrix &other); //Copy constructor
//DESTRUCTORS
~matrix(void); //Destructor
//OPERATORS

```

```

friend temp_matrix operator+(const matrix &m1, const matrix &m2); //matrix+matrix
friend temp_matrix operator+(const double &d1, const matrix &m1); //scalar+matrix
friend temp_matrix operator+(const matrix &m1, const double &d1); //matrix+scalar
friend temp_matrix operator-(const matrix &m1, const matrix &m2); //matrix-matrix
friend temp_matrix operator-(const double &d1, const matrix &m1); //scalar-matrix
friend temp_matrix operator-(const matrix &m1, const double &d1); //matrix-scalar
friend temp_matrix operator*(const matrix &m1, const matrix &m2); //matrix*matrix
friend temp_matrix operator*(const double &d1, const matrix &m1); //scalar*matrix
friend temp_matrix operator*(const matrix &m1, const double &d1); //matrix*scalar
friend temp_matrix operator->*(const matrix &m1, const matrix &m2); //matrix*matrix pointwise
matrix& operator=(const matrix &m); //matrix=matrix equality
matrix& operator=(const temp_matrix& m); //matrix=temp equality
matrix& operator=(double val); //Scalar assignment
double& operator()(long row, long column); //matrix(i,j) subscripting
};

```

Table 1.5: Listing of the matrix derived class.

Similarly, the derived `temp_matrix` class inherits all of its data from the `base_matrix` class and consists of twenty-five (25) methods: four (4) constructors, a destructor and twenty (20) overloaded operators.

```

class temp_matrix:public base_matrix
{
private:
public:
//FUNCTIONS
//CONSTRUCTORS
temp_matrix(void); //Alternate constructor
temp_matrix(long r, long c); //Alternate constructor
temp_matrix(long r, long c, char *s); //Alternate constructor
temp_matrix(const temp_matrix& other); //Copy constructor
//DESTRUCTORS
~temp_matrix(void); //Destructor
//OPERATORS
friend temp_matrix operator+(const temp_matrix& m1, const temp_matrix& m2); //temp+temp
friend temp_matrix operator+(const matrix& m1, const temp_matrix& m2); //matrix+temp
friend temp_matrix operator+(const temp_matrix& m1, const matrix& m2); //temp+matrix
friend temp_matrix operator+(const double &d1, const temp_matrix &m1); //scalar+temp
friend temp_matrix operator+(const temp_matrix &m1, const double &d1); //temp+scalar
friend temp_matrix operator-(const temp_matrix& m1, const temp_matrix& m2); //temp-temp
friend temp_matrix operator-(const matrix& m1, const temp_matrix& m2); //matrix-temp
friend temp_matrix operator-(const temp_matrix& m1, const matrix& m2); //temp-matrix
friend temp_matrix operator-(const double &d1, const temp_matrix &m1); //scalar-temp
friend temp_matrix operator-(const temp_matrix &m1, const double &d1); //temp-scalar
friend temp_matrix operator*(const temp_matrix& m1, const temp_matrix& m2); //temp*temp
friend temp_matrix operator*(const matrix& m1, const temp_matrix& m2); //matrix*temp
friend temp_matrix operator*(const temp_matrix& m1, const matrix& m2); //temp*matrix

```

```

friend temp_matrix operator*(const double &d1, const temp_matrix &m1); //scalar*temp
friend temp_matrix operator*(const temp_matrix &m1, const double &d1); //temp*scalar
friend temp_matrix operator->*(const temp_matrix &m1, const temp_matrix &m2); //temp*temp pointwise
friend temp_matrix operator->*(const matrix &m1, const temp_matrix &m2); //matrix*temp pointwise
friend temp_matrix operator->*(const temp_matrix &m1, const matrix &m2); //temp*matrix pointwise
temp_matrix& operator=(temp_matrix& m); //matrix=temp equality
double& operator()(long row, long column); //matrix(i,j) subscripting
};

```

Table 1.6: Listing of the `temp_matrix` derived class.

Table 1.7 summarizes the operations available to the user via the `matrix` class hierarchy, their mathematical interpretation and their return values is applicable.

Overloaded operator or operation	Mathematical description	Return type
$M1+M2$	matrix-matrix summation	<code>temp_matrix</code>
$s1+M1$	scalar-matrix summation	<code>temp_matrix</code>
$M1+s1$	matrix-scalar summation	<code>temp_matrix</code>
$M1-M2$	matrix-matrix subtraction	<code>temp_matrix</code>
$s1-M1$	scalar-matrix subtraction	<code>temp_matrix</code>
$M1-s1$	matrix-scalar subtraction	<code>temp_matrix</code>
$M1*M2$	matrix-matrix multiplication	<code>temp_matrix</code>
$s1*M1$	scalar-matrix multiplication	<code>temp_matrix</code>
$M1*s1$	matrix-scalar multiplication	<code>temp_matrix</code>
$M1->*M2$	matrix-matrix pointwise multiplication	<code>temp_matrix</code>
$M1=M2$	matrix-matrix assignment	<code>matrix& or temp_matrix&†</code>
$M1=s1$	matrix-scalar assignment	<code>matrix& or temp_matrix&</code>
$M1(i,j)$	matrix subscripting producing scalar	<code>double&</code>
$s1=M1(i,j)$	scalar-scalar assignment	<code>void‡</code>
$M1(i,j)=s1$	scalar-scalar assignment	<code>matrix& or temp_matrix&</code>
$absmax(M1)$	entry with maximum absolute value	<code>double</code>
$sumabs(M1)$	sum of absolute values of all entries	<code>double</code>

†:& stands for a reference which is simply an alias for a variable. It is one of the ways C++ implements passing by reference instead of passing by value.

‡:The keyword `void` indicates that no value is returned by the operation.

Table 1.7: Supported overloaded operators and operations for full matrix algebra via the `matrix` class hierarchy. `M1` and `M2` each represent a `matrix` or `temp_matrix` instance, while `s1` represents a scalar, and `i` and `j` represent integers.

It is interesting to find that the matrix transpose is not one of the member overloaded operators or one of the other methods. It was decided to not include it for two main reasons. First, out of all the overloadable operators in C++, none was satisfactorily close to the actual mathematical notation to justify its use. Secondly, creating a method to implement the transpose would force the user to use the C++ notation `object.method(parameters)`. It is felt that in order to express a mathematical

equation, say $M = P^T Ax$, it is clearer and easier to understand the corresponding C++ expression using the `matrix` hierarchy, which reads `M=transpose(P)*A*x` rather than the one which reads `M=P.transpose()*A*x`. It is obvious, however, that this might be a learned response after years of using MATLAB as the primary tool for these types of mathematical operations. In any case, the argument is still a valid one, specially if the objective is the pursue of an intuitive and clear way of implementing matrix algebra in an object-oriented language. Hence, the `transpose()` function is an auxiliary function that is defined with the `matrix` class hierarchy, but is not a method of the classes. It is simply a function which takes a reference to a `matrix` object as an argument and returns a `temp_matrix` object.

The difference between the `matrix` and `temp_matrix` classes lies in the form of their destructor method, in the way the results of the mathematical operations performed by their overloaded operators are stored, and in how their data is copied when an equality operator is used and when a compound operation is performed.

First, both types of matrices allocate memory for their entries dynamically. Upon termination of their use, their destructor is invoked. The destructor for the `matrix` class deallocates the memory used to store its entries, i.e. both the pointer `*entries` and the data pointed to by it, essentially erasing the data. On the other hand, the destructor for the `temp_matrix` class does not deallocate the memory, it simple deletes the pointer but not the data pointed to by it.

Secondly, the overloaded operators of class `matrix` operate only on `matrix` objects and objects of primitive type. Within the bodies of these overloaded operators `temp_matrix` objects are created to store the results of the corresponding operations (summation, subtraction, products, etc.) making sure the original arguments are not changed in any way during the process. These temporary objects are then returned.

The overloaded operators of class `temp_matrix` operate on `matrix`, `temp_matrix` and primitive types. Within the bodies of the overloaded operators, the corresponding operations are performed in place, i.e. the `temp_matrix` object which is received as an argument is overwritten with the result of the operation in question when the operation can be done in place and the it is returned. In particular, when both arguments are `temp_matrix` objects, one of them is overwritten with the solution and the data of the remaining argument is destroyed since it is no longer needed. This is true for all operations in table 1.7 except for matrix-matrix multiplication.

Lastly, when the equality operator is used to assign a `matrix` object to another `matrix` object, all their data is copied from one to the other. The body of the `operator=()` method copies the data members one at a time. However, when the `*entries` pointer is reached, the pointer is not simply copied since that would mean that both objects would be pointing to the same data in memory. This is not what is needed since in the act, the user does not necessarily want the data of the `matrix` object on left hand side destroyed when the object on the right hand side is destroyed. In fact, a new block of memory is allocated for the object on the left hand side of the expression and the entries are copied one by one from the left hand side. The same thing occurs when a `matrix` object is initialized using a constructor that takes an object of the same type as an argument. This particular constructor is called a **copy constructor**. When the equality operator is used to assign a `temp_matrix` object to a `matrix` object, all the data members of the `temp_matrix` are simply copied to the `matrix` object, including the `*entries` pointer. Since the `temp_matrix` destructor does not delete the actual data pointed to by `*entries`, no data is lost. The same sequence of events occurs within the body of the `temp_matrix` class copy constructor.

Why is there a need for both a `matrix` class and a `temp_matrix` class and all this complexity? The answer to this question is somewhat involved, but it has everything to do with both computational efficiency and efficient memory management. In order to see the effects of having or not having both types of objects, we need to examine in detail how a mathematical expression in C++ would be evaluated in either scenario. A simple example will suffice, since the ramifications of either choice become rapidly important.

Consider the sample listing in table 1.8:

```
#include <iostream.h>
#include <matrix.h>

int main(void)
{
    matrix A(10,10,1.0);
    matrix B(10,10,2.0);
    matrix C(10,10,3.0);
    matrix D;

    D=A+B+C;
    cout << "The contents of matrix D=A+B+C are:" << D;

    return 0;
}
```

Table 1.8: Sample listing demonstrating the use of the matrix class hierarchy.

The expression $D=A+B+C$ is actually equivalent to the more complex expression $D.operator=(operator+(operator+(A,B),C))$. The operations performed in either scenario are summarized in table 1.9. By using both the `matrix` and `temp_matrix` classes, only one block of memory is repeatedly used to store the current result of the operation. Whenever the copy constructor or the equality operator are called, only the pointer to the memory block is copied, not the contents of the block. Then, when the temporary `temp_matrix` objects are destroyed, only the interface disappears, not the memory block with the data, since their destructor is empty. It is clear that using

only the `matrix` class, involves allocating and deallocating memory unnecessarily, and copying possibly enormous memory blocks a number of times directly proportional to the number of operators involved. The use of the `matrix` class hierarchy as depicted in this study, prevents all these inefficiencies automatically and transparently for the user.

Operations history for $D=A+B+C$	
matrix class only	matrix and temp_matrix classes together
<ul style="list-style-type: none"> • <code>operator+(A,B)</code> is called. • <code>matrix temp1</code> is created to store $A+B$. • <code>internal matrix internal1</code> is created using copy constructor to receive <code>temp1</code> from the return statement in the <code>operator+(A,B)</code> method. • <code>matrix temp1</code> is destroyed. • <code>operator+(internal1,C)</code> is called. • <code>matrix temp2</code> is created to store $internal1+C$. • <code>internal matrix internal2</code> is created using copy constructor to receive <code>temp2</code> from the return statement in the <code>operator+(internal1,C)</code> method. • <code>matrix temp2</code> is destroyed. • <code>D.operator=(internal2)</code> is called. • <code>matrix internal2</code> is destroyed. • <code>matrix internal1</code> is destroyed. 	<ul style="list-style-type: none"> • <code>operator+(A,B)</code> is called. • <code>temp_matrix temp1</code> is created to store $A+B$. • <code>internal temp_matrix internal1</code> is created using copy constructor to receive <code>temp1</code> from the return statement in the <code>operator+(A,B)</code> method. • <code>temp_matrix temp1</code> is destroyed. • <code>operator+(internal1,C)</code> is called. • <code>temp_matrix internal1</code> is overwritten with $internal1+C$. • <code>internal temp_matrix internal2</code> is created using copy constructor to receive <code>internal1</code> from the return statement in the <code>operator+(internal1,C)</code> method. • <code>D.operator=(internal2)</code> is called. • <code>temp_matrix internal2</code> is destroyed. • <code>temp_matrix internal1</code> is destroyed.
Total memory cost: 3 concurrent matrices of same size excluding A,B,C and D	Total memory cost: 1 concurrent matrix which ends up being D
Total time cost: 3 full copies and 4 full deletes	Total time cost: 0 full copies and 0 full deletes

Table 1.9: Comparison of cost of linear algebra operations when only the `matrix` class is used and when it is used together with the `temp_matrix` class.

1.6.2 The `sparse_matrix` hierarchy

Tables 1.10, 1.11 and 1.12 describe a C++ implementation of a `sparse_matrix` class hierarchy. Its base or parent class is called `base_sparse_matrix` and it implements the basic components and behavior of sparse matrices. Some authors argue that a class such as `sparse_matrix` should be derived from a full `matrix` class, since otherwise it is implied that a sparse matrix is not a matrix [9]. The philosophy behind the `sparse_matrix` class hierarchy implementation presented in this study is that, in fact, all matrices are full matrices. The concept of a sparse matrix is simply mathematically convenient, and is thus an artifact. Furthermore, most operations involving sparse

matrices benefit from drastically different approaches from those utilized to perform operations between full matrices. Hence, it is believed that attempting to derive one class from the other will inevitably unnecessarily burden their implementation.

```
class base_sparse_matrix
{
private:
double* entries;
long* column_index;
long* row_index;
long length;
long num_of_diagonal_entries;
char* name;
public:
//INTERFACE FUNCTIONS
char* get_name(void);
void set_name(char* new_name);
double* get_entries(void);
void set_entries(double* new_entries);
long* get_column_index(void);
void set_column_index(long* new_column_index);
long* get_row_index(void);
void set_row_index(long* new_row_index);
long get_length(void);
void set_length(long new_length);
long get_num_of_diagonal_entries(void);
void set_num_of_diagonal_entries(long num);
//CONSTRUCTORS
base_sparse_matrix(void);
//DESTRUCTORS
~base_sparse_matrix(void);
//FUNCTIONS
};
```

Table 1.10: Listing of the `base_sparse_matrix` base class.

There exists no standard unique representation scheme for sparse matrices. On the other hand, there are popular storage formats which have been thoroughly used in the past. The implementation presented in this study attempts to take advantage of the possible structure encountered in matrices generated by FEA formulations. Unfortunately, none of the popular storage formats by itself seemed to provide all the desired features. A decision was made to derive a storage format which combined

the features of the compressed row storage, the compressed column storage and the compressed diagonal storage formats [14].

The six (6) data members of the base class accomplish this task. They consists of a dynamic sequential one-dimensional array of entries, two dynamic sequential one-dimensional arrays of column and row indexes, the total length of these arrays, the number of diagonal entries, and a name tag for display and/or debugging purposes. In addition, only fourteen (14) basic self-descriptive interface methods are implemented.

For the same reasons why the `matrix` class hierarchy consists of two derived classes `matrix` and `temp_matrix`, the `sparse_matrix` class hierarchy consists of two derived classes `sparse_matrix` and `temp_sparse_matrix`. The derived `sparse_matrix` class inherits all its data and basic interface from the `base_sparse_matrix` class. It also implements twenty (20) methods: six (6) constructors, a destructor, five (5) control methods, and eight (8) overloaded operators.

```
class sparse_matrix:public base_sparse_matrix
{
private:
public:
//CONSTRUCTORS
sparse_matrix(void); //Alternate constructor
sparse_matrix(char *s); //Alternate constructor
sparse_matrix(long the_length, double val); //Alternate constructor
sparse_matrix(long the_length, char *s); //Alternate constructor
sparse_matrix(long the_length, double val, char *s); //Alternate constructor
sparse_matrix(sparse_matrix &other); //Copy constructor
//DESTRUCTORS
~sparse_matrix(void);
//FUNCTIONS
void copy(sparse_matrix& m);
void allocate(long the_length);
void save(char* file);
void load(char* file);
void show(void);
//OPERATORS
friend temp_sparse_matrix operator+(sparse_matrix& m1, sparse_matrix& m2); //matrix+matrix
friend temp_sparse_matrix operator-(sparse_matrix& m1, sparse_matrix& m2); //matrix-matrix
friend temp_sparse_matrix operator*(const double& d1, sparse_matrix& m2); //scalar*matrix
friend temp_sparse_matrix operator*(sparse_matrix& m1, const double& d2); //matrix*scalar
```

```

friend temp_matrix operator*(sparse_matrix& m1, matrix& m2); //Matrix*column matrix
friend temp_matrix operator*(sparse_matrix& m1, temp_matrix& m2); //Matrix*temp column matrix
sparse_matrix& operator=(sparse_matrix &m); //matrix=matrix equality
sparse_matrix& operator=(temp_sparse_matrix& m); //matrix=temp equality
};

```

Table 1.11: Listing of the sparse_matrix derived class.

Similarly, the derived temp_sparse_matrix class inherits its data and basic interface from the base_sparse_matrix and implements an additional sixteen (16) methods: four (4) constructors, a destructor, a control method and ten (10) overloaded operators.

```

class temp_sparse_matrix:public base_sparse_matrix
{
private:
public:
//CONSTRUCTORS
temp_sparse_matrix(void); //Alternate constructor
temp_sparse_matrix(char *s); //Alternate constructor
temp_sparse_matrix(long the_length, char *s); //Alternate constructor
temp_sparse_matrix(temp_sparse_matrix &other); //Copy constructor
//DESTRUCTORS
~temp_sparse_matrix(void);
//FUNCTIONS
void allocate(long the_length);
//OPERATORS
friend temp_sparse_matrix operator+(temp_sparse_matrix& m1, temp_sparse_matrix& m2); //temp+temp
friend temp_sparse_matrix operator+(sparse_matrix& m1, temp_sparse_matrix& m2); //matrix+temp
friend temp_sparse_matrix operator+(temp_sparse_matrix& m1, sparse_matrix& m2); //temp+matrix
friend temp_sparse_matrix operator-(temp_sparse_matrix& m1, temp_sparse_matrix& m2); //temp-temp
friend temp_sparse_matrix operator-(sparse_matrix& m1, temp_sparse_matrix& m2); //matrix-temp
friend temp_sparse_matrix operator-(temp_sparse_matrix& m1, sparse_matrix& m2); //temp-matrix
friend temp_sparse_matrix operator*(const double &d1, temp_sparse_matrix &m1); //scalar*temp
friend temp_sparse_matrix operator*(temp_sparse_matrix &m1, const double &d1); //temp*scalar
friend temp_matrix operator*(temp_sparse_matrix& m1, matrix& m2); //temp*column matrix
friend temp_matrix operator*(temp_sparse_matrix& m1, temp_matrix& m2); //temp*temp column matrix
};

```

Table 1.12: Listing of the temp_sparse_matrix derived class.

Table 1.13 summarizes the operations available to the user via the sparse_matrix class hierarchy, their mathematical interpretation and their return values is applicable.

It is important to notice that the number of operations that are supported by the sparse_matrix class hierarchy is smaller than those supported by the matrix

Overloaded operator or operation	Mathematical description	Return type
$M1+M2$	matrix-matrix summation	<code>temp_sparse_matrix</code>
$M1-M2$	matrix-matrix subtraction	<code>temp_sparse_matrix</code>
$M1 \cdot C1$	matrix-column matrix multiplication	<code>temp_matrix</code>
$s1 \cdot M1$	scalar-matrix multiplication	<code>temp_sparse_matrix</code>
$M1 \cdot s1$	matrix-scalar multiplication	<code>temp_sparse_matrix</code>
$M1=M2$	matrix-matrix assignment	<code>sparse_matrix&</code>

Table 1.13: Supported overloaded operators and operations for full matrix algebra via the matrix class hierarchy. $M1$ and $M2$ each represent a `sparse_matrix` or `temp_sparse_matrix` instance, while $C1$ represents a column matrix or `temp_matrix`, and $s1$ represents a scalar.

class hierarchy. The reason is simple. The supported operations are the most common in FEA applications. For example, in all of the applications presented in this study, not a single one was identified which required a product of two or more sparse matrices. Hence, such an operation was not supported. However, due to the object-oriented implementation, such an addition would not be complicated and could be achieved either by adding new methods to the existing classes, or by additions to the `sparse_matrix` class hierarchy via derivation and inheritance. On the contrary, the most common operation identified and therefore, the most important overloaded operator in the class hierarchy is the product operator which takes a `sparse_matrix&` or a `temp_sparse_matrix&` and a `matrix&` or `temp_matrix&` with only one column. Such an operation is used in FEA applications to transform the global unknowns into new ones using the information gathered from all elements in the domain via the assembly process. The unknowns are stored in a column vector, i.e. a matrix with only one column, and the element information is stored in a sparse matrix obtained by assembling all the local full matrices based on the element connectivity within the domain of interest. Examples of the use of this and all other operators will be presented later.

Finally, as with the transpose operator, several additional functions have been added to the hierarchy, but are not implemented as methods of any of the classes

for the sake of clarity. As will be seen later, the operations in table 1.14 are also supported.

Overloaded operator or operation	Mathematical description	Return type
<code>tanh()</code>	calculates the hyperbolic tangent of every entry in a matrix	<code>matrix</code>
<code>condense()</code>	places the sum of all entries in each row in the diagonal of the matrix	<code>temp.matrix</code> or <code>temp.sparse.matrix</code>
<code>abs()</code>	calculates the absolute value of every entry in a matrix	<code>temp.matrix</code>
<code>reciprocal()</code>	calculates the reciprocal of every entry in a matrix	<code>temp.matrix</code>

Table 1.14: Additional functions supported by the `matrix` and `sparse.matrix` class hierarchies.

1.7 Geometrical management

After determining the different classes needed to manage the geometrical needs of FEA applications, a detailed discussion of their implementation is required. The following sections do just that. Notice however, that detailed description of applications making use of these and the mathematical management classes will be postponed until chapter 2.

1.7.1 The node class

The declaration of the `node` class is present in table 1.15. In this implementation, the properties and behavior of a physical node have been encapsulated and hidden within the `node` class declaration via abstraction in the form of eight (8) data members and twenty-three (23) function members or methods:

```
class node
{
private:
long dimension;           //Dimensions of the problem indicate how many different
                           //coordinates are needed to specify the node uniquely
double *coordinates;      //Array of nodal coordinates whose length is the variable
```

```

//"dimension"
long node_type;           //Flag indicating whether the node is an internal
                           //node, or a boundary node with Dirichlet, von
                           //Neumann or flux, or mixed boundary conditions
long node_number;         //Global node number used for latter assembly process
long num_of_parent_elements; //Number of parent elements for this node, i.e. how many
                           //elements does this node belong to?
element **parent_elements; //Array of pointers to parent elements
long variables_per_node;   //Stores the number of variables per node
double *nodal_values;      //Stores the values of all the nodal variables for this object

public:
//VIRTUAL FUNCTIONS
//FUNCTIONS
void set_dimensions(long number_of_coordinates);
long get_dimensions(void);
void set_coordinates(double *coords);
double *get_coordinates(void);
void set_node_type(long type_of_node);
long get_node_type(void);
void set_node_number(long global_node_number);
long get_node_number(void);
void set_variables_per_node(long vars_per_node);
long get_variables_per_node(void);
void set_nodal_values(double *values);
double *get_nodal_values(void);
void set_num_of_parent_elements(long num);
long get_num_of_parent_elements(void);
void increase_num_of_parent_elements(void);
void add_parent_element(element *lmnt);
element *get_parent_element(long num);
//CONSTRUCTORS
node(void);
//DESTRUCTORS
virtual ~node(void);
//OPERATORS
friend ostream& operator << (ostream& s, node& any_node);
};

```

Table 1.15: Listing of the node class.

As can be seen, the node class declaration includes short descriptions of its data members. They have been declared `private` in order to hide them from node class clients. These data members are:

- **dimension** - indicates the number of coordinates which are necessary to specify the node uniquely.

- `coordinates` - list which stores the coordinates of the node. The number of coordinates in the list is `dimension`.
- `node_number` - stores the global number of the node which is obtained by the finite element discretization of the domain.
- `num_of_parent_elements` - stores the number of elements this particular node belongs to.
- `parent_elements` - stores a list of pointers to the node's parent elements. The number of pointers in the list is `num_of_parent_elements`.

From them, it is clear that the node class simply serves to keep track of the coordinates of the finite element domain. The fact that the node objects contain pointers to their parent elements indicates that nodes and elements form a data structure similar to a **linked list**. The main advantage of such a data structure is the fact that the addition of new `node` objects only involves the reassignment of pointers to account for the location of the new object in the list.

The methods of the class have self descriptive names, so it is not necessary to go into detail on their form. However, it is important to observe that all internal properties of a `node` object are available to the user via the method interface. Although, `node` objects can be created by the user directly, the OOFEA toolkit uses them extensively only in the context of its clients. The first `node` client is the `element` class hierarchy, and its second client is the `mesh` class. In either case, the most used and thus most important method is `node::get_coordinates` (this notation establishes the class-method relationship as *class::method*), which returns a list of coordinates corresponding to the `node` object in question.

1.7.2 The element hierarchy

The `element` class is much more involved than the `node` class as it is responsible for most of the data handling in an OOFEA application. Its implementation is composed of sixteen (16) data members and thirty-eight (38) methods:

```
class element //node client
{
private:
    long element_number;    //Number of the element in the mesh
    long dimension;        //Dimension of the element (1D, 2D or 3D)
    long number_of_nodes;   //How many nodes in the element
    long element_order;     //Stores the order or degree of the element
    long number_of_unknows; //Stores the number of unknowns per element;
    long *connectivity;     //Array of length "number_of_nodes" which stores
                           //the global node number of all the nodes in the element
    double generalized_area; //Stores the generalized area of the element
    double dx;              //Stores the representative x-spacing
    double dy;              //Stores the representative y-spacing
    double det_Jacobian;    //Stores the determinant of the Jacobian of the element
    node **element_nodes;   //Dynamic array of length "number_of_nodes" which store
                           //pointers to the global nodes which make the element up
    matrix Jacobian;        //Stores the Jacobian of the element
    matrix inv_Jacobian;    //Stores the inverse of the Jacobian of the element
    matrix N_vector;        //Stores the shape functions N_sub_i
    matrix B_matrix;        //Stores the derivatives of the shape functions
    matrix* local_solution; //Stores all the unknowns for the nodes of the element
public:
    //VIRTUAL FUNCTIONS
    //PURE VIRTUAL FUNCTIONS
    virtual void calculate_N_vector(double* Gauss_points)=0;
    virtual void calculate_B_matrix(double* Gauss_points)=0;
    virtual void calculate_Jacobian(double* Gauss_points)=0;
    virtual void calculate_det_Jacobian(void)=0;
    virtual void calculate_inv_Jacobian(void)=0;
    //FUNCTIONS
    void set_dimension(long dim);
    long get_dimension(void);
    void set_number_of_nodes(long num);
    long get_number_of_nodes(void);
    void set_element_order(long order);
    long get_element_order(void);
    void set_number_of_unknows(long num);
    long get_number_of_unknows(void);
    void set_connectivity(long *connect, long num_of_nodes);
    long *get_connectivity(void);
    void set_element_nodes(node **enodes, long num_of_nodes);
    node **get_element_nodes(void);
}
```

```

void set_area(double area);
double get_area(void);
void calculate_generalized_area(void);
void calculate_dx(void);
void calculate_dy(void);
void set_Jacobian(matrix& jacobian);
matrix& get_Jacobian(void);
void set_det_Jacobian(double det);
double get_det_Jacobian(void);
void set_inv_Jacobian(matrix& inv);
matrix& get_inv_Jacobian(void);
void set_element_number(long e_number);
long get_element_number(void);
matrix& get_W_vector(void);
matrix& get_B_matrix(void);
matrix* get_local_solution(void);
matrix& get_local_solution(int num);
element *get_my_address(void);
//CONSTRUCTORS
element(void);
//DESTRUCTORS
virtual ~element(void);
//OPERATORS
friend ostream& operator << (ostream& s, element& any_element);
};

```

Table 1.16: Listing of the element base class.

The data members are accompanied by a short description in the class declaration. A better description follows below:

- **element number** - stores the global number of the element determined from the finite element discretization of the problem domain.
- **dimension** - stores the element dimension for possible use in selecting the appropriate interpolation functions for the element.
- **number_of_nodes** - stores the total number of nodes in the element which might or might not be directly dependent on the element order and the element dimension.

- **element_order** - stores the order of the possibly polynomial interpolation functions used to approximate the behavior of the problem unknowns over the finite element.
- **number_of_unknowns** - stores the number of unknown variables which are to be approximated over the element and which are problem dependent.
- **connectivity** - stores a list of node numbers corresponding to the global number of the nodes that form the element. The length of the list is **number_of_nodes**.
- **generalized_area** - stores the n-dimensional area of the element (length, area, volume, etc.) depending on the value of **dimension**. This is particularly useful when dealing with triangular and tetrahedral elements.
- **element_nodes** - stores a list of pointers to the nodes that are part of the element. The size of the list is **number_of_nodes**.
- **local_solution** - list of matrices which hold the local value of the unknowns at the nodes of the element. The variable **number_of_unknowns** is the number of matrices in the list.
- **N_vector** - stores the values of the interpolation functions evaluated at key points in the element (generally at the Gauss points).
- **B_matrix** - stores the values of the derivatives of the interpolation functions evaluated at key points in the element (generally at the Gauss points).
- **Jacobian** - stores the values of the Jacobian of the element, evaluated at key points in the element (generally at the Gauss points), which represents a geometric transformation of the element from the actual coordinate system of the

problem domain to a general regular element in a domain described by natural or unit coordinates.

- `inv_Jacobian` - stores the values of the inverse of the Jacobian, evaluated at key points in the element (generally at the Gauss points).
- `det_Jacobian` - stores the determinant of the Jacobian, evaluated at key point in the element (generally at the Gauss points),

As can be seen, all data members are declared `private` as in the case of the `node` class. The `element` interface allows the user to manipulate all its features directly and, as with the `node` class, the method names are self-descriptive, and for most of them there is no need for further description. One thing to notice is that the `element` class contains data members which are instances of both the `node` class and the `matrix` class. For that reason, the `element` class is a client of both these classes. In particular, the `element` class contains a list of pointers to its component nodes. This implies that the `node` and `element` objects form a special kind of linked list, called a **doubly linked list**: the data structure can be transversed from the element to the node and from the node to the element.

Although the data members presented are capable of describing the data that is usually associated with any finite element, the way in which these data members are calculated and determined is not unique. In particular, the information related to the way in which local unknowns and coordinates are interpolated can vary from element to element. The data members usually affected are: `N_vector`, `B_matrix`, `Jacobian`, `inv_Jacobian` and `det_Jacobian`.

Furthermore, some application might require elements to store additional information, such as properties, boundary conditions, etc. It is for that reason that the

`element` class introduces virtual functions to the picture, and becomes the base class for an `element` class hierarchy.

Every single FEA application has the potential need for new types of elements. The OOFEA toolkit satisfies this requirement by developing a base `element` class which encapsulates the common aspects of all elements, and the allowing the user via inheritance to derive all the elements desired.

As an example, four (4) derived element classes are included in the `element` class hierarchy, and their declarations are presented in tables 1.17 to 1.20. Need for these particular element classes emerged as the applications described later in this study where being developed.

It is particularly interesting to see that only the functions declared virtual in the base class have to redefined. If one recalls the definition of a virtual function, it becomes clear that each of the derived classes can and will in general implement these functions differently. For example, the `domain_element` class virtual functions will implement interpolation functions and their derivatives based on the dimension of the problem (i.e. two-dimensional domains will require two-dimensional interpolation functions).

```
class domain_element:public element
{
private:
public:
//VIRTUAL FUNCTIONS
virtual void calculate_N_vector(double* Gauss_points); //Calculates shape function vector N
virtual void calculate_B_matrix(double* Gauss_points); //Calculates shape function derivative matrix B
virtual void calculate_Jacobian(double* Gauss_points); //Calculates Jacobian
virtual void calculate_det_Jacobian(void); //Calculates determinant of Jacobian
virtual void calculate_inv_Jacobian(void); //Calculates inverse of Jacobian
//FUNCTIONS
//OPERATORS
friend ostream& operator << (ostream& s, domain_element& any_element);
};
```

Table 1.17: Listing of the `domain_element` derived class.

On the other hand, `boundary_element` classes will implement corresponding interpolation functions of smaller dimension i.e. the boundaries of a two-dimensional domain are one-dimensional). In particular, a `flux_boundary_element` class implements one-dimensional interpolation functions and at the same time introduces an additional data member and two (2) new methods to its declaration. The data member `flux` stores the value of the flux at the boundary which can then be used to apply natural boundary conditions. The new methods permit the user to access the new data member.

```
class flux_boundary_element:public element
{
private:
double flux; //Stores the flux boundary condition for a flux boundary element
public:
//VIRTUAL FUNCTIONS
virtual void calculate_N_vector(double* Gauss_points); //Calculates shape function vector N
virtual void calculate_B_matrix(double* Gauss_points); //Calculates shape function derivative matrix B
virtual void calculate_Jacobian(double* Gauss_points); //Calculates Jacobian
virtual void calculate_det_Jacobian(void); //Calculates determinant of Jacobian
virtual void calculate_inv_Jacobian(void); //Calculates inverse of Jacobian
//FUNCTIONS
void set_flux(double the_flux);
double get_flux(void);
};
```

Table 1.18: Listing of the `flux_boundary_element` derived class.

Similarly, a `convective_boundary_element` class implements one-dimensional interpolation functions and at the same time introduces two (2) additional data members and four (4) new methods to its declaration. The data member `h` stores the convection coefficient at the wall of the domain, and the data member `T_inf` stores the value far away from the boundary of the ambient property of interest being convected. The new methods simply serve as new interface functions to manipulate the new data members. Such an element can be used to describe and manage convective boundary conditions (for example, in heat transfer problems).

```

class convective_boundary_element:public element
{
private:
double h; //Convection coefficient
double T_inf; //Ambient property far away from the domain
public:
//VIRTUAL FUNCTIONS
virtual void calculate_N_vector(double* Gauss_points); //Calculates shape function vector N
virtual void calculate_B_matrix(double* Gauss_points); //Calculates shape function derivative matrix B
virtual void calculate_Jacobian(double* Gauss_points); //Calculates Jacobian
virtual void calculate_det_Jacobian(void); //Calculates determinant of Jacobian
virtual void calculate_inv_Jacobian(void); //Calculates inverse of Jacobian
//FUNCTIONS
void set_convection_coefficient(double convcoeff);
double get_convection_coefficient(void);
void set_ambient_property(double tinf);
double get_ambient_property(void);
};

```

Table 1.19: Listing of the convective_boundary_element derived class.

Finally, a prescribed_boundary_element class implements its virtual functions by leaving them empty, and introduces a new data member and two (2) new methods. This element is used to store Dirichlet type boundary conditions. The reason the virtual methods are left empty is that these types of elements are actually zero-dimensional elements and hence equivalent to nodes.

```

class prescribed_boundary_element:public element
{
private:
double Dirichlet_BC;
public:
//VIRTUAL FUNCTIONS
virtual void calculate_N_vector(double* Gauss_points); //Calculates shape function vector N
virtual void calculate_B_matrix(double* Gauss_points); //Calculates shape function derivative matrix B
virtual void calculate_Jacobian(double* Gauss_points); //Calculates Jacobian
virtual void calculate_det_Jacobian(void); //Calculates determinant of Jacobian
virtual void calculate_inv_Jacobian(void); //Calculates inverse of Jacobian
//FUNCTIONS
void set_Dirichlet_BC(double bc);
double get_Dirichlet_BC(void);
};

```

Table 1.20: Listing of the prescribed_boundary_element derived class.

What makes the element class hierarchy so important? The answer is simple. The element classes contain all the data and methods which are necessary to implement

The `mesh` class is the culmination of the OOFEA toolkit. It is a client of all the previous classes and hierarchies, and provides the user with the methods necessary to process the crucial elemental weak forms previously described and generate global systems of equations which when appropriately solved, yield the solutions to the problems being attacked. Table 1.21 is a listing of the `mesh` class declaration.

[illegible]

```

//elements that form the mesh
matrix* global_solution;           //Dynamic array of length "number_of_unknowns"
                                   //which stores pointers to the unknown global
                                   //global solutions of the problem over the mesh

public:
//VIRTUAL FUNCTIONS
//FUNCTIONS
void set_number_of_nodes(long num_of_nodes);
long get_number_of_nodes(void);
void set_number_of_domain_elements(long num_of_elements);
void set_number_of_flux_elements(long num_of_elements);
void set_number_of_convective_elements(long num_of_elements);
long get_number_of_domain_elements(void);
long get_number_of_flux_elements(void);
long get_number_of_convective_elements(void);
long get_number_of_prescribed_elements(long var_num);
node** get_mesh_nodes(void);
domain_element** get_domain_elements(void);
flux_boundary_element** get_flux_elements(void);
convective_boundary_element** get_convective_elements(void);
prescribed_boundary_element*** get_prescribed_elements(void);
void load_nodal_coordinates(char *file);
void load_domain_elements(char *file);
void load_flux_elements(char *file);
void load_convective_elements(char *file);
void load_prescribed_elements(long var_num, char *file);
void allocate_mesh_nodes(long num_of_nodes);
void allocate_domain_elements(long num_of_elements);
void allocate_flux_elements(long num_of_elements);
void allocate_convective_elements(long num_of_elements);
void allocate_prescribed_elements(long num_of_unknowns);
void display(void){cout << *this;};
void display_ith_element(long ith_element);
void display_ith_global_node(long ith_global_node);
void calculate_areas(void);
matrix& apply_Dirichlet_BCs(long var_num, matrix& A);
sparse_matrix& apply_Dirichlet_BCs(long var_num, sparse_matrix& A);
matrix* get_global_solution(void){return global_solution;};
matrix& get_global_solution(int num);
void load_global_solution(long num, char* name, char* file);
void save_global_solution(long num, char* name, char* file);
void disassemble_global_solution(long num);
temp_matrix calculate_local_matrix(domain_element& any_element,
    temp_matrix(*problem_dependent_function)(domain_element&));
temp_sparse_matrix assemble_global_matrix_from_local_ones(temp_matrix
    (*problem_dependent_function)(domain_element&));
temp_matrix calculate_local_vector(domain_element& any_element,
    temp_matrix (*problem_dependent_function)(domain_element&));
temp_matrix assemble_global_vector_from_local_ones(temp_matrix
    (*problem_dependent_function)(domain_element&));

```

```

temp_matrix calculate_local_vector(flux_boundary_element& any_element,
    temp_matrix (*problem_dependent_function)(flux_boundary_element&));
temp_matrix assemble_global_vector_from_local_ones(temp_matrix
    (*problem_dependent_function)(flux_boundary_element&));
temp_matrix calculate_local_matrix(convective_boundary_element& any_element,
    temp_matrix (*problem_dependent_function)(convective_boundary_element&));
temp_sparse_matrix assemble_global_matrix_from_local_ones(temp_matrix
    (*problem_dependent_function)(convective_boundary_element&));
temp_matrix calculate_local_vector(convective_boundary_element& any_element,
    temp_matrix (*problem_dependent_function)(convective_boundary_element&));
temp_matrix assemble_global_vector_from_local_ones(temp_matrix
    (*problem_dependent_function)(convective_boundary_element&));

//CONSTRUCTORS
mesh(void);
//DESTRUCTORS
virtual ~mesh(void);
//OPERATORS
friend ostream& operator << (ostream& s, mesh& any_mesh);
};

```

Table 1.21: Listing of the mesh class.

The `mesh` class is composed of twelve (12) data members and forty-one (41) methods. The data members are:

- `number_of_nodes` - stores the total number of nodes which compose the domain of the problem.
- `number_of_unknowns` - stores the number of different variables which are unknown within the domain of the problem and which are to be solved for.
- `number_of_domain_elements` - stores the number of elements which compose the domain of the problem.
- `number_of_flux_elements` - stores the number of elements which composes the part of the boundary of the domain in which natural boundary conditions are to be applied.

- `number_of_convective_elements` - stores the number of elements which composes the part of the boundary of the domain in which convective boundary conditions are to be applied.
- `number_of_prescribed_elements` - stores the number of elements which composes the part of the boundary of the domain in which Dirichlet boundary conditions are to be applied.
- `mesh_nodes` - list of pointers to the `node` objects which compose the domain of the problem.
- `domain_elements` - list of pointers to the `domain_element` objects which compose the domain of the problem.
- `flux_elements` - list of pointers to the `flux_boundary_element` objects which compose the domain of the problem.
- `convective_elements` - list of pointers to the `convective_boundary_element` objects which compose the domain of the problem.
- `prescribed_elements` - list of pointers to the `prescribed_boundary_element` objects which compose the domain of the problem.
- `global_solution` - list of matrices which store the values of the unknowns of the problem in column matrix form.

Although the `tt mesh` class provides a large number of interface methods, there are twelve (12) particular ones which are at the core of the OOFEA toolkit. Using these twelve methods, a finite element engine can be built. These methods are:

- *mesh::load_nodal_coordinates* - Takes a filename as an argument. The file must specify the number of nodes, the dimension of the problem, the node types and their coordinates. The method then creates `node` objects based on the user file, and arranges these newly created objects into the internal `mesh` data structure.
- *mesh::load_domain_elements* - Takes a filename as an argument. The file must specify the dimension of the elements, the number of domain elements, the number of unknowns in the domain, and for each element, its order of interpolation, its global element number and its connectivity. The method then creates `domain_element` objects based on the user file, and arranges these newly created objects into the internal `mesh` data structure.
- *mesh::load_flux_elements* - Takes a filename as an argument. The file must specify the dimension of the elements, the number of flux elements, and for each element, its order of interpolation, its global element number, its connectivity and the value of the flux. The method then creates `flux_boundary_element` objects based on the user file, and arranges these newly created objects into the internal `mesh` data structure.
- *mesh::load_convective_elements* - Takes a filename as an argument. The file must specify the dimension of the elements, the number of convective elements, and for each element, its order of interpolation, its global element number, its connectivity, and the value of the convection coefficient and the ambient value of the convected property away from the element. The method then creates `convective_boundary_element` objects based on the user file, and arranges these newly created objects into the internal `mesh` data structure.

- *mesh::load_prescribed_elements* - Takes an integer and a filename as arguments. The integer argument specifies for which of the problem unknowns the Dirichlet boundary conditions are being specified. The file must specify the dimension of the elements, the number of prescribed elements, and for each element, its order of interpolation, its global node number, and the Dirichlet boundary condition at that node. The method then creates `prescribed_boundary_element` objects based on the user file, and arranges these newly created objects into the internal `mesh` data structure.
- *mesh::load_global_solution* - Takes an integer, a string and a filename as arguments. Reads data from file and stores it in the appropriate `global_solution` matrix specified by the integer argument. The character string is used for display and/or debugging purposes.
- *mesh::disassemble_global_solution* - Takes an integer as an argument. The appropriate segments of the appropriate `global_solution` matrix specified by the integer argument are distributed to the appropriate `local_solution` matrix in each of the elements in the domain.
- *mesh::apply_Dirichlet_BCs* - Takes an integer and a `sparse_matrix` as arguments. The integer argument specifies which of the Dirichlet boundary conditions is to be applied to the sparse matrix argument.
- *mesh::calculate_local_matrix* - Takes two arguments: a reference to an object derived from the `element` class, and a function which operates on a reference to an object derived from the `element` class and returns a `temp_matrix`. Given an user-defined function as argument which represents a weak form, this method will call that function as many times as necessary to evaluate the weak form

numerically and store as a `temp_matrix` object. If new types of elements are derived for an application, a new type of mesh must be derived to include a data structure to store these new type of elements, and this function must be overloaded to operate on those new elements.

- *`mesh::assemble_global_matrix_from_local_ones`* - Takes one argument: a function which operates on a reference to an object derived from the `element` class, and returns a `temp_matrix`. Given an user-defined function which represents a weak form as argument, this method will call *`mesh::calculate_local_matrix`* for each element in the mesh. Then, all the obtained element matrices will be assembled into a sparse matrix based on the connectivity of the elements in the domain. If new types of elements are derived for an application, a new type of mesh must be derived to include a data structure to store these new type of elements, and this function must be overloaded to operate on those new elements.
- *`mesh::calculate_local_vector`* - Takes two arguments: a reference to an object derived from the `element` class, and a function which operates on a reference to an object derived from the `element` class and returns a `temp_matrix`. Given an user-defined function as argument which represents a weak form, this method will call that function as many times as necessary to evaluate the weak form numerically and store as a `temp_matrix` object. In particular, the resulting matrix should be a column matrix, i.e. a matrix with only one column. a column vector. If new types of elements are derived for an application, a new type of mesh must be derived to include a data structure to store these new type of elements, and this function must be overloaded to operate on those new elements.

- *mesh::assemble_global_vector_from_local_ones* - Takes one argument: a function which operates on a reference to an object derived from the `element` class and returns a `temp_matrix`. Given an user-defined function which represents a weak form as argument, this method will call *mesh::calculate_local_vector* for each element in the mesh. Then, all the obtained element column matrices will be assembled into a global column matrix based on the connectivity of the elements in the domain. If new types of elements are derived for an application, a new type of mesh must be derived to include a data structure to store these new type of elements, and this function must be overloaded to operate on those new elements.

Why does the `mesh` class operates on functions defined by the user instead of on some sort of `weak_form` object? The reason is that since there is no general structure for a weak form generated by the FEA, other than the fact that they involve matrix algebra operations, any particular `weak_form` object quickly becomes inadequate for a new application. Letting the user define the needed weak forms as functions which operate on `element` objects, and return `temp_matrix` objects which are the result of the required matrix operations seems to be a more general approach. The `mesh` class then uses these functions as arguments to the assembly methods previously described.

1.7.4 Supporting iterator classes

As described in section 1.5, a set of iterator classes has to be defined in order to access the objects that describe the problem domain easily and in a standardized manner. Tables 1.22 to 1.25 declare four (4) of such iterator classes.

```
class Nodes_of_Element
{
private:
```

```

int cur;
node** nodes;
public:
Nodes_of_Element(element& any_element);
~Nodes_of_Element(void){};
int more(void){return cur>=1;};
void advance(void){--cur;};
node& current(void){return *nodes[cur-1];};
};

```

Table 1.22: Listing of the Nodes_of_Element iterator class.

```

class Nodes_of_Mesh
{
private:
int cur;
node** nodes;
public:
Nodes_of_Mesh(mesh& any_mesh);
~Nodes_of_Mesh(void){};
int more(void){return cur>=1;};
void advance(void){--cur;};
node& current(void){return *nodes[cur-1];};
};

```

Table 1.23: Listing of the Nodes_of_Mesh iterator class.

```

class Domain_Elements_of_Mesh
{
private:
int cur;
domain_element** domain_elements;
public:
Domain_Elements_of_Mesh(mesh& any_mesh);
~Domain_Elements_of_Mesh(void){};
int more(void){return cur>=1;};
void advance(void){--cur;};
domain_element& current(void){return *domain_elements[cur-1];};
};

```

Table 1.24: Listing of the Domain_Elements_of_Mesh iterator class.

```

class Prescribed_Elements_of_Mesh
{
private:
int cur;
long var_number;
prescribed_boundary_element*** prescribed_elements;
public:
Prescribed_Elements_of_Mesh(mesh& any_mesh, long var_num);

```

```

~Prescribed_Elements_of_Mesh(void){};
int more(void){return cur>=1;};
void advance(void){--cur;};
prescribed_boundary_element& current(void){return *prescribed_elements[var_number-1][cur-1];};
};

```

Table 1.25: Listing of the Prescribed_Elements_of_Mesh iterator class.

These iterator classes are used in conjunction with the standard C/C++ form of the `for` loop: `for(initialization; condition; increment){loop body}`, where the *initialization* is usually an assignment statement that sets up the loop control variable, the *conditional* is a relational or Boolean expression which determines when the loop will exit (exit if FALSE or 0, continue if TRUE or 1), and the *increment* defines how the loop control variable changes each the loop is executed. As seen in table 1.26, the *initialization* involves the creation of an instance of the appropriate iterator class using its constructor method. The *condition* involves the use of the `more()` method which returns TRUE or 1 while unaccessed objects remain in the data structure and FALSE or zero only when all objects have been accessed. The *increment* involves the use of the `advance()` method which appropriately changes the contents of the iterator object to reflect the fact that the number of remaining objects in the data structure has been reduce by one, and that the object returned by the `current()` method is the next one in the data structure. The *loop body* involves using the `current()` method to access the actual object in the data structure being transversed.

```

#include <iostream.h>
#include <fem.h>

#define EXIT 0

int main(void)
{
    mesh any_mesh;

    any_mesh.load_nodal_coordinates("sample_coordinates.dat");
    any_mesh.load_domain_elements("sample_connectivity.dat");
    any_mesh.load_global_solution(1,"sample variable","sample_data.dat");
}

```

```

any_mesh.load_prescribed_elements(1,"sample_Dirichlet_BCs.dat");

for(Nodes_of_Mesh ith_node(any_mesh);ith_node.more();ith_node.advance())
{
    cout << ith_node.current(); //Displays every node in the mesh any_mesh
}

for(Domain_Elements_of_Mesh ith_element(any_mesh);ith_element.more();ith_element.advance())
{
    cout << ith_element.current(); //Displays every element in the mesh any_mesh
    for(Nodes_of_Element jth_node(ith_element.current());jth_node.more();jth_node.advance())
    {
        cout << jth_node.current(); //Displays every node in every element of the mesh any_mesh
    }
}

for(Prescribed_Elements_of_Mesh jth_element(any_mesh);jth_element.more();jth_element.advance();)
{
    cout << jth_element.current(); //Displays every prescribed boundary element in the mesh any_mesh
}

return EXIT;
}

```

Table 1.26: Sample listing of code which uses all iterator classes.

Chapter 2

C++ OOFEA sample applications for CFD

2.1 Introduction

The following sections are a collection of CFD applications developed with the help of the OOFEA toolkit described in the previous chapter. These will serve as a practical guide to the use of the toolkit as well as a validation of the generated code. Within the applications, several common CFD solution algorithms are also developed. In particular, it will become clear that the suggested approach generates reusable code which migrates from application to application and can be easily combined into useful libraries.

2.2 General structure of an OOFEA application

The general structure of an OOFEA application can be described as follows:

- Include the `fem.h` file using the `#include<fem.h>` command.
- Include the declaration of all user-defined weak forms.
- Include the declaration of any other user-defined functions.
- Start the main program.
 - Create a `mesh` object for the domain in question
 - Using the provided `mesh` methods, load the
 - * mesh coordinate information,

- * mesh connectivity information, and
- * mesh boundary information
- Create `sparse_matrix` objects to store the assembled finite element matrices obtained by invoking the appropriate `mesh` methods with the user-defined weak forms as arguments.
- Create `matrix` objects to store the assembled finite element load vectors obtained by invoking the appropriate `mesh` methods with the user-defined weak forms as arguments.
- Write and solve the finite element formulations for the problem using the operators defined for the `matrix` and `sparse_matrix` hierarchies.
- Post-process the solutions as necessary
- Store any of the generated information in permanent storage
- End the main program.

2.3 General structure of user-defined weak forms

As will be seen in the next sections, the main differentiating factor between different OOFEA applications is the user-defined weak forms. These weak forms are nothing but user defined functions which return a `temp_matrix` object and operate on a reference to an `element` object or any object derived from the `element` class. Within the function body, `element` methods are used to access all the necessary information to evaluate the weak form of choice. These weak form functions will later be used within the body of the OOFEA application as arguments to the assembly methods of the `mesh` class, which in turn evaluate the user-defined weak form numerically using an appropriate quadrature rule. Table 2.1 is an annotated example of such a function.

```
//The name of the function, here USER_DEFINED_FUNCTION is a user-defined name,
//i.e. it can be any valid string. Also, a reference to an object derived from
//the element class is also valid as an argument
```

```
temp_matrix USER_DEFINED_FUNCTION(element& any_element)
{
    //Use any methods necessary to determine the desired weak form.
    //For example, a capacitance matrix usually has the form
    matrix& M=any_element.get_M_vector();
    double det_J=any_element.get_det_Jacobian();
    return transpose(M)*M*det_J;
}
```

Table 2.1: General structure of an user-defined weak form using the OOFEA toolkit.

2.4 Preconditioned conjugate gradient solver

In this study, all solutions of systems of equations will be performed using the preconditioned conjugate gradient iterative algorithm. Using the `matrix` and `sparse_matrix` hierarchies, the implementation is straightforward given a pseudocode like the one in table 2.2.

```
Pseudocode for Preconditioned Conjugate Gradient Iterative Algorithm
Compute  $r_0 = b - Ax_0$  for some initial guess  $x_0$ 
for  $i = 1, 2, \dots$ 
    Solve  $Mz_{i-1} = r_{i-1}$  where  $M$  is the preconditioner of  $A$ 
     $\rho_{i-1} = r_{i-1}^T z_{i-1}$ 
    if  $i = 1$ 
         $p_1 = z_0$ 
    else
         $\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$ 
         $p_i = z_{i-1} + \beta_{i-1} p_{i-1}$ 
    endif
     $q_i = Ap_i$ 
     $\alpha_i = \rho_{i-1} / p_i^T q_i$ 
     $x_i = x_{i-1} + \alpha_i p_i$ 
     $r_i = r_{i-1} - \alpha_i q_i$ 
    check convergence and continue if necessary
end
```

Table 2.2: Pseudocode for the Preconditioned Conjugate Gradient Iterative Algorithm.

Table 2.3 is a listing of the OOFEA implementation of the PCG algorithm. Except maybe for the calculation of the diagonal preconditioner, which accesses some of the

data members of the `sparse_matrix` argument, the rest of the code is directly taken from the pseudocode almost verbatim.

```
matrix& PCG(sparse_matrix& A, matrix& b, matrix& x, const double& tolerance)
{
    long iterations=0;
    matrix r("r"), p("p"), q("q"), z("z");
    matrix new_rho("new_rho"), old_rho("old_rho"), alpha(1,1,"alpha"), beta(1,1,"beta");
    matrix tol(1,1,tolerance,"tolerance");
    matrix norm_residual(1,1,0,"norm of residual");

    //Preconditioner calculations
    //Inverse of Diagonal or Point or Jacobi preconditioning
    long num_of_diagonal_entries=A.get_num_of_diagonal_entries();
    sparse_matrix M(num_of_diagonal_entries,"M");
    M.set_num_of_diagonal_entries(num_of_diagonal_entries);
    double* entriesA=A.get_entries();double* entriesM=M.get_entries();
    long* row_indA=A.get_row_index();long* row_indM=M.get_row_index();
    long* col_indA=A.get_column_index();long* col_indM=M.get_column_index();
    long loop1;
    for (loop1=1;loop1<=num_of_diagonal_entries;loop1++)
    {
        entriesM[loop1-1]=1.0/entriesA[loop1-1];
        row_indM[loop1-1]=row_indA[loop1-1];
        col_indM[loop1-1]=col_indA[loop1-1];
    }

    //CG Algorithm
    r=b-A*x;
    norm_residual=transpose(r)*r;
    norm_residual(1,1)=sqrt(norm_residual(1,1));
    while (norm_residual(1,1) > tolerance)
    {
        iterations++;
        z=M*r;
        new_rho=transpose(r)*z;
        if (iterations==1)
        {
            p=z;
        }
        else
        {
            beta=new_rho(1,1)/old_rho(1,1);
            p=z+beta(1,1)*p;
        }
        old_rho=new_rho;
        q=A*p;
        alpha=transpose(p)*q;
        alpha=new_rho(1,1)/alpha(1,1);
```

```

x=x+alpha(1,1)*p;
r=r-alpha(1,1)*q;
norm_residual=transpose(r)*r;
norm_residual(1,1)=sqrt(norm_residual(1,1));
if (SOLVERS_DEBUG_FUNCTIONS) cout << "Current PCG step: " << iterations << "\n";
}
cout << "Number of PCG iterations: " << iterations << endl;
return x;
}

```

Table 2.3: Listing of the preconditioned conjugate gradient algorithm implementation using the OOFEA toolkit.

2.5 Notes on the sample applications

All the applications presented in the next sections have been solved with the use of rectangular 4-node linear isoparametric finite elements. For all the applications the governing equations, appropriate domain, boundary and/or initial conditions will be defined, and the resulting finite element equations will also be presented. In addition, the corresponding OOFEA implementation is included for comparison purposes, and sample results are included as necessary to demonstrate the toolkit functionality.

2.6 Static heat transfer

The governing equations in index notation for a static heat transfer boundary value problem are:

$$(k_{ij}T_{,j})_{,i} + f = 0 \text{ in } \Omega \quad (2.1)$$

$$T = \hat{T} \text{ on } \Gamma_1 \quad (2.2)$$

$$(k_{ij}T_{,j})n_i = q \text{ on } \Gamma_2 \quad (2.3)$$

$$(k_{ij}T_{,j})n_i = h(T - T_\infty) \text{ on } \Gamma_3 \quad (2.4)$$

where Ω is the domain, and $\Gamma = \Gamma_1 \cup \Gamma_2 \cup \Gamma_3$ is its boundary. After introducing the appropriate interpolation functions and applying the Galerkin finite element method, the resulting set of finite element equations is:

$$(\mathbf{K} + \mathbf{C})\mathbf{T} - \mathbf{LV} - \mathbf{FV} - \mathbf{CV} = 0 \quad (2.5)$$

where $\mathbf{T}$ is the unknown temperature vector, $\mathbf{K}$ is the stiffness matrix, $\mathbf{C}$ is the contribution of the convective boundary conditions to the stiffness matrix, and $\mathbf{LV}$, $\mathbf{FV}$ and $\mathbf{CV}$ are the load vector, the flux contribution to the load vector and the convective contribution to the load vector, respectively. These are obtained by evaluating the following integrals over each finite element and assembling:

$$\mathbf{K} = \iint_{st} \mathbf{B}_{st}^T (\mathbf{J}^{-1})^T (\mathbf{J}^{-1}) \mathbf{B}_{st} \det \mathbf{J} \, ds dt \quad (2.6)$$

$$\mathbf{C} = \iint_{st} \mathbf{N}^T \mathbf{h} \mathbf{J}^T \mathbf{N} \, ds dt \quad (2.7)$$

$$\mathbf{LV} = \int_s \mathbf{N}^T f \det \mathbf{J} \, ds \quad (2.8)$$

$$\mathbf{FV} = \int_s \mathbf{N}^T \mathbf{q} \mathbf{J}^T \, ds \quad (2.9)$$

$$\mathbf{CV} = \int_s T_\infty \mathbf{N}^T \mathbf{h} \mathbf{J}^T \, ds \quad (2.10)$$

where $\mathbf{q} = [q_x \, q_y]$, and $\mathbf{h} = [h_x \, h_y]$.

The code in table 2.4 implements equation 2.5, first by declaring the appropriate user-defined functions for the two (2) matrices and three (3) vectors involved, and then using the `mesh` methods to define the geometry, assemble the components of the finite element formulation, and finally solve it using an appropriate solver, in this case, the preconditioned conjugate gradient of section 2.4.

It is noticeable that within the body of the functions defining the contributions to the load vector, certain geometrical operations are taking place. These are nec-

essary to decompose the boundary integrals into the appropriate coordinate system, since in general, boundary elements will not be aligned with them. In these sample problems, the coordinate system of choice is Cartesian, and then the fluxes have to be decomposed appropriately into their x and y components. Mathematically this means:

$$\begin{aligned}
 \oint \mathbf{N}^T \mathbf{q} \, d\Gamma &= \int \mathbf{N}^T q_x \, dy + \int \mathbf{N}^T q_y \, dx \\
 &= \int \mathbf{N}^T q_x \frac{dy}{ds} ds + \int \mathbf{N}^T q_y \frac{dx}{ds} ds \\
 &= \int \mathbf{N}^T [q_x \, q_y] \begin{bmatrix} \frac{dy}{ds} & \frac{dx}{ds} \end{bmatrix}^T ds \\
 &= \int \mathbf{N}^T [q_x \, q_y] \mathbf{J}^T ds \\
 &= \int \mathbf{N}^T \mathbf{q} \mathbf{J}^T ds
 \end{aligned} \tag{2.11}$$

which is the form of the equations for **LV**, **FV** and **CV**. These transformations are left to the user since they are dependent on the choice of coordinate system. These types of transformations are mostly necessary when there is a need to apply natural boundary conditions.

```

#include <fem.h>
#include <matrix.h>
#include <stdio.h>
#include <solvers.h>

temp_matrix HTS2D_stiffness_matrix(domain_element& any_element)
{
    matrix& B=any_element.get_B_matrix();
    matrix& inv_J=any_element.get_inv_Jacobian();
    double det_J=any_element.get_det_Jacobian();
    matrix k(2,2,"k");
    k(1,1)=6;k(1,2)=0;k(2,1)=0;k(2,2)=6;
    return transpose(B)*transpose(inv_J)*k*inv_J*B*det_J;
}

```

```

temp_matrix HTS2D_load_vector(domain_element& any_element)
{
    matrix& M=any_element.get_M_vector();
    double det_J=any_element.get_det_Jacobian();
    double f=3.0;
    return transpose(M)*f*det_J;
}

temp_matrix HTS2D_naturalbcs_load(flux_boundary_element& any_element)
{
    //qn is the given value of the flux at the corresponding element
    //This should be stored in the particular flux boundary element
    //and requested using an interface function
    double qn;
    qn=any_element.get_flux();

    double **coords;coords=new (double* [2]);long i=0;
    for (Nodes_of_Element ith_node(any_element);ith_node.more();ith_node.advance())
    {
        coords[i]=ith_node.current().get_coordinates();i++;
    }

    double dx,dy;
    dx=fabs(coords[0][0]-coords[1][0]);dy=fabs(coords[0][1]-coords[1][1]);

    double alpha,theta;
    alpha=atan2(dy,dx);theta=M_PI_2-alpha;

    double qx,qy;
    qx=qn*cos(theta);qy=qn*sin(theta);
    matrix q(1,2,"q");
    q(1,1)=qy;q(1,2)=qx;

    matrix& M=any_element.get_M_vector();
    matrix& J=any_element.get_Jacobian();

    return transpose(M)*q*transpose(J);
}

temp_matrix HTS2D_convectivebcs_stiffness(convective_boundary_element& any_element)
{
    //hn is the given value of the convection coefficient
    //This should be stored in the particular convective boundary element
    //and requested using an interface function
    double hn;
    hn=any_element.get_convection_coefficient();

    double **coords;coords=new (double* [2]);long i=0;
    for (Nodes_of_Element ith_node(any_element);ith_node.more();ith_node.advance())

```



```

{
coords[i]=ith_node.current().get_coordinates();i++;
}

double dx,dy;
dx=fabs(coords[0][0]-coords[1][0]);dy=fabs(coords[0][1]-coords[1][1]);

double alpha,theta;
alpha=atan2(dy,dx);theta=M_PI_2-alpha;

double hx,hy;
hx=hn*cos(theta);hy=hn*sin(theta);
matrix h(1,2,"h");
h(1,1)=hy;h(1,2)=hx;

matrix& H=any_element.get_H_vector();
matrix& J=any_element.get_Jacobian();

return transpose(H)*h*transpose(J)*H;
}

temp_matrix HTS2D_convectivebcs_load(convective_boundary_element& any_element)
{
//hn is the given value of the convection coefficient
//This should be stored in the particular convective boundary element
//and requested using an interface function
double hn,T_inf;
hn=any_element.get_convection_coefficient();
T_inf=any_element.get_ambient_property();

double **coords;coords=new (double* [2]);long i=0;
for (Nodes_of_Element ith_node(any_element);ith_node.more();ith_node.advance())
{
coords[i]=ith_node.current().get_coordinates();i++;
}

double dx,dy;
dx=fabs(coords[0][0]-coords[1][0]);dy=fabs(coords[0][1]-coords[1][1]);

double alpha,theta;
alpha=atan2(dy,dx);theta=M_PI_2-alpha;

double hx,hy;
hx=hn*cos(theta);hy=hn*sin(theta);
matrix h(1,2,"h");
h(1,1)=hy;h(1,2)=hx;

matrix& H=any_element.get_H_vector();
matrix& J=any_element.get_Jacobian();

```

```

return transpose(I)*h*transpose(J)*T_inf;
}

int main(void)
{
mesh mesh1;

mesh1.load_nodal_coordinates("d:\\users\\rafa\\c-code\\data\\coords5.dat");
mesh1.load_domain_elements("d:\\users\\rafa\\c-code\\data\\connect5.dat");
mesh1.load_flux_elements("d:\\users\\rafa\\c-code\\data\\flux5.dat");
mesh1.load_convective_elements("d:\\users\\rafa\\c-code\\data\\convect5.dat");

sparse_matrix K, C, A;
matrix FV, CV, LV, b;

K=mesh1.assemble_global_matrix_from_local_ones(HTS2D_stiffness_matrix);
C=mesh1.assemble_global_matrix_from_local_ones(HTS2D_convectivebcs_stiffness);
LV=mesh1.assemble_global_vector_from_local_ones(HTS2D_load_vector);
FV=mesh1.assemble_global_vector_from_local_ones(HTS2D_naturalbcs_load);
CV=mesh1.assemble_global_vector_from_local_ones(HTS2D_convectivebcs_load);

A=K+C;
b=LV+FV+CV;
matrix guess(mesh1.get_number_of_nodes(),1,0,"guess");
PCG(A,b,guess,1e-5);
cout << "Solution is: " << guess;

return 0;
}

```

Table 2.4: Listing of a 2D static heat transfer simulator using the OOFEA toolkit.

The sample problem in this area involves the heat flow within a square cavity whose left wall, and left halves of the top and bottom walls are isothermal at $400K$; whose right wall, and the right halves of the top and bottom walls convect heat to an ambient medium at $300K$ with a convection heat transfer coefficient of $3W/m^2K$; which has four heat sources at several locations within the isothermal walls with a heat transfer rate of $500W$; and is filled with a material with a thermal conductivity of $6W/mK$, which also generates heat at a rate of $3W/m^3$. Figure 2.1 is a plot of the labeled isotherms of the solution to the problem.

Figure 2.1: Solution to a 2D static heat transfer problem where 4 heat sources are placed at several location in heated walls while the rest of the cavity convects the heat away.

2.7 Potential flow

The governing equations in index notation for a potential flow boundary value problem are:

$$\Phi_{,ii} = 0 \text{ in } \Omega \quad (2.12)$$

$$\Phi = \hat{\Phi} \text{ on } \Gamma_1 \quad (2.13)$$

$$(\Phi_{,j}) n_i = q \text{ on } \Gamma_2 \quad (2.14)$$

where Ω is the domain, and $\Gamma = \Gamma_1 \cup \Gamma_2$ is its boundary. After introducing the appropriate interpolation functions and applying the Galerkin finite element method, the resulting set of finite element equations is:

$$\mathbf{K}\Phi - \mathbf{FV} = 0 \quad (2.15)$$

where Φ is the unknown potential vector, $\mathbf{K}$ is the stiffness matrix, and $\mathbf{FV}$ is the flux contribution to the load vector.

The code in table 2.5 implements equation 2.15, first by declaring the appropriate user-defined functions for the stiffness matrix and load vector, and then using the mesh methods to define the geometry, assemble the components of the finite element formulation, and finally solve it using the preconditioned conjugate gradient algorithm.

```
#include <fem.h>
#include <matrix.h>
#include <stdio.h>
#include <solvers.h>

temp_matrix PF_stiffness_matrix(domain_element& any_element)
{
    matrix& B=any_element.get_B_matrix();
    matrix& inv_J=any_element.get_inv_Jacobian();
    double det_J=any_element.get_det_Jacobian();
    return transpose(B)*transpose(inv_J)*inv_J*B*det_J;
}

temp_matrix PF_naturalbcs_load(flux_boundary_element& any_element)
{
    double qn;
    qn=any_element.get_flux();

    double **coords;coords=new (double* [2]);long i=0;
    for (Nodes_of_Element ith_node(any_element);ith_node.more();ith_node.advance())
    {
        coords[i]=ith_node.current().get_coordinates();i++;
    }

    double dx,dy;
    dx=fabs(coords[0][0]-coords[1][0]);dy=fabs(coords[0][1]-coords[1][1]);

    double alpha,theta;
    alpha=atan2(dy,dx);theta=M_PI_2-alpha;

    double qx,qy;
    qx=qn*cos(theta);qy=qn*sin(theta);
    matrix q(1,2,"q");
    q(1,1)=qy;q(1,2)=qx;

    matrix& N=any_element.get_N_vector();
```

```

matrix& J=any_element.get_Jacobian();

return transpose(M)*q*transpose(J);
}

int main(void)
{
mesh mesh1;

mesh1.load_nodal_coordinates("d:\\users\\rafa\\c-code\\data\\coords6.dat");
mesh1.load_domain_elements("d:\\users\\rafa\\c-code\\data\\connect6.dat");
mesh1.load_flux_elements("d:\\users\\rafa\\c-code\\data\\flux6.dat");
mesh1.load_prescribed_elements(1,"d:\\users\\rafa\\c-code\\data\\prescr6.dat");

sparse_matrix K;
K=mesh1.assemble_global_matrix_from_local_ones(PF_stiffness_matrix);
matrix F;
F=mesh1.assemble_global_vector_from_local_ones(PF_naturalbcs_load);

mesh1.apply_Dirichlet_BCs(1,K);
mesh1.apply_Dirichlet_BCs(1,F);

matrix guess(mesh1.get_number_of_nodes(),1,0,"guess");
PCG(K,F,guess,1.0e-5);
cout << "Solution is: " << guess;

return 0;
}

```

Table 2.5: Listing of a 2D potetial flow simulator using the OOFEA toolkit.

The sample problem in this area involves calculating the streamline pattern obtained within a $5m \times 5m$ square cavity when fluid is injected through an opening at the bottom wall, and fluid exits the cavity via an opening in the left wall. The boundary conditions for the problem are all of the Dirichlet type, hence the value of the stream function is specified at every point in the mesh. The problem is discretized regularly into a mesh with 25×25 elements. The value of the stream function in the line segment which joins both the inlet and the outlet and includes both the bottom left, top left and top right corners of the cavity is zero ($0s^{-1}$). The value of the stream function in the line segment which joins the inlet and the outlet and includes

the bottom right corner of the cavity is $100s^{-1}$. Figure 2.2 is a plot of the labeled streamlines which represent the solution of the problem.

Figure 2.2: Stream function pattern within a $5m \times 5m$ square enclosure with an inlet at the bottom horizontal surface and an outlet at the right vertical surface. The grid system is regular with 25×25 elements.

2.8 Transient heat transfer

The governing equations in index notation for a transient heat transfer initial boundary value problem are:

$$\rho C_p \dot{T} - (k_{ij} T_{,j})_{,i} - f = 0 \text{ in } \Omega \quad (2.16)$$

$$T = \hat{T} \text{ on } \Gamma_1 \quad (2.17)$$

$$(k_{ij} T_{,j}) n_i = q \text{ on } \Gamma_2 \quad (2.18)$$

$$(k_{ij} T_{,j}) n_i = h (T - T_\infty) \text{ on } \Gamma_3 \quad (2.19)$$

$$T = \tilde{T} \text{ at } t = 0 \quad (2.20)$$

where Ω is the domain, and $\Gamma = \Gamma_1 \cup \Gamma_2 \cup \Gamma_3$ is its boundary. After introducing the appropriate interpolation functions and applying the Galerkin finite element method, the resulting set of finite element equations is:

$$\mathbf{M}\dot{\mathbf{T}} + (\mathbf{K} + \mathbf{C}) \mathbf{T} - \mathbf{LV} - \mathbf{FV} - \mathbf{CV} = 0 \quad (2.21)$$

where $\mathbf{T}$ is the unknown temperature vector, $\mathbf{M}$ is the mass or capacitance matrix, $\mathbf{K}$ is the stiffness matrix, $\mathbf{C}$ is the contribution of the convective boundary conditions to the stiffness matrix, and $\mathbf{LV}$, $\mathbf{FV}$ and $\mathbf{CV}$ are the load vector, the flux contribution to the load vector and the convective contribution to the load vector, respectively. The mass matrix is obtained by evaluating and assembling:

$$\mathbf{M} = \iint_{\Omega} \mathbf{N}^T \rho C_p \mathbf{N} \det \mathbf{J} \, ds dt \quad (2.22)$$

The code in table 2.6 implements equation 2.21, first by declaring the appropriate user-defined functions for the three (3) matrices and three (3) vectors involved, and then using the `mesh` methods to define the geometry, assemble the components of the finite element formulation, and finally solve it using an appropriate solver. The solution algorithm employed in this sample problem is the Crank-Nicolson scheme. Table 2.7 is a listing of the Crank-Nicolson function as used. As can be seen, it has also being implemented using the OOFEA toolkit.

```
#include <fem.h>
#include <matrix.h>
#include <stdio.h>
#include <solvers.h>

temp_matrix HTT2D_capacitance_matrix(domain_element& any_element)
{
    matrix& N=any_element.get_N_vector();
    double det_J=any_element.get_det_Jacobian();
    double rho_Cp=1.0;
    return transpose(N)*rho_Cp*N*det_J;
}
```

```

temp_matrix HTT2D_stiffness_matrix(domain_element& any_element)
{
    matrix& B=any_element.get_B_matrix();
    matrix& inv_J=any_element.get_inv_Jacobian();
    double det_J=any_element.get_det_Jacobian();
    matrix k(2,2,"k");
    k(1,1)=6;k(1,2)=0;k(2,1)=0;k(2,2)=6;
    return transpose(B)*transpose(inv_J)*k*inv_J*B*det_J;
}

temp_matrix HTT2D_load_vector(domain_element& any_element)
{
    matrix& F=any_element.get_F_vector();
    double det_J=any_element.get_det_Jacobian();
    double f=3.0;
    return transpose(F)*f*det_J;
}

temp_matrix HTT2D_naturalbcs_load(flux_boundary_element& any_element)
{
    double qn;
    qn=any_element.get_flux();

    double **coords;coords=new (double* [2]);long i=0;
    for (Nodes_of_Element ith_node(any_element);ith_node.more();ith_node.advance())
    {
        coords[i]=ith_node.current().get_coordinates();i++;
    }

    double dx,dy;
    dx=fabs(coords[0][0]-coords[1][0]);dy=fabs(coords[0][1]-coords[1][1]);

    double alpha,theta;
    alpha=atan2(dy,dx);theta=M_PI_2-alpha;

    double qx,qy;
    qx=qn*cos(theta);qy=qn*sin(theta);
    matrix q(1,2,"q");
    q(1,1)=qy;q(1,2)=qx;

    matrix& F=any_element.get_F_vector();
    matrix& J=any_element.get_Jacobian();

    return transpose(F)*q*transpose(J);
}

temp_matrix HTT2D_convectivebcs_stiffness(convective_boundary_element& any_element)
{
    double hn;
    hn=any_element.get_convection_coefficient();

```



```

double **coords; coords=new (double* [2]); long i=0;
for (Nodes_of_Element ith_node(any_element); ith_node.more(); ith_node.advance())
{
    coords[i]=ith_node.current().get_coordinates(); i++;
}

double dx, dy;
dx=fabs(coords[0][0]-coords[1][0]); dy=fabs(coords[0][1]-coords[1][1]);

double alpha, theta;
alpha=atan2(dy, dx); theta=M_PI_2-alpha;

double hx, hy;
hx=hn*cos(theta); hy=hn*sin(theta);
matrix h(1,2,"h");
h(1,1)=hy; h(1,2)=hx;

matrix& H=any_element.get_H_vector();
matrix& J=any_element.get_Jacobian();

return transpose(H)*h*transpose(J)*H;
}

temp_matrix HTT2D_convectivebcs_load(convective_boundary_element& any_element)
{
    double hn, T_inf;
    hn=any_element.get_convection_coefficient();
    T_inf=any_element.get_ambient_property();

    double **coords; coords=new (double* [2]); long i=0;
    for (Nodes_of_Element ith_node(any_element); ith_node.more(); ith_node.advance())
    {
        coords[i]=ith_node.current().get_coordinates(); i++;
    }

    double dx, dy;
    dx=fabs(coords[0][0]-coords[1][0]); dy=fabs(coords[0][1]-coords[1][1]);

    double alpha, theta;
    alpha=atan2(dy, dx); theta=M_PI_2-alpha;

    double hx, hy;
    hx=hn*cos(theta); hy=hn*sin(theta);
    matrix h(1,2,"h");
    h(1,1)=hy; h(1,2)=hx;

    matrix& H=any_element.get_H_vector();
    matrix& J=any_element.get_Jacobian();

```

```

return transpose(M)*h*transpose(J)*T_inf;
}

int main(void)
{
mesh mesh1;

mesh1.load_nodal_coordinates("d:\\users\\rafa\\c-code\\data\\coords5.dat");
mesh1.load_domain_elements("d:\\users\\rafa\\c-code\\data\\connect5.dat");
mesh1.load_flux_elements("d:\\users\\rafa\\c-code\\data\\flux5.dat");
mesh1.load_convective_elements("d:\\users\\rafa\\c-code\\data\\convect5.dat");
mesh1.load_prescribed_elements(1,"d:\\users\\rafa\\c-code\\data\\prescr5.dat");

sparse_matrix M, K, C, KC;
matrix LV, FV, CV, b;

M=mesh1.assemble_global_matrix_from_local_ones(HTT2D_capacitance_matrix);
K=mesh1.assemble_global_matrix_from_local_ones(HTT2D_stiffness_matrix);
C=mesh1.assemble_global_matrix_from_local_ones(HTT2D_convectivebcs_stiffness);
LV=mesh1.assemble_global_vector_from_local_ones(HTT2D_load_vector);
FV=mesh1.assemble_global_vector_from_local_ones(HTT2D_naturalbcs_load);
CV=mesh1.assemble_global_vector_from_local_ones(HTT2D_convectivebcs_load);

b=LV+FV+CV;
KC=K+C;

matrix guess(mesh1.get_number_of_nodes(),1,0,"guess");
cout << "Solution is: ";
cout << Crank_Nicolson(mesh1,1,M,KC,b,guess,0.5,500,1.0e-5);

return 0;
}

```

Table 2.6: Listing of a 2D transient heat transfer simulator using the OOFEA toolkit.

```

matrix& Crank_Nicolson(mesh& any_mesh, const long var_num,
                        sparse_matrix& M, sparse_matrix& K,
                        matrix& F, matrix& x, const double& dt,
                        long steps, double tolerance)
{
sparse_matrix A;
A=(1/dt)*M+0.5*K;
any_mesh.apply_Dirichlet_BCs(var_num,A);
matrix b;
long step;
for (step=1;step<=steps;step++)
{
b=((1/dt)*M-0.5*K)*x+F;
any_mesh.apply_Dirichlet_BCs(var_num,b);
PCG(A,b,x,tolerance);
}
}

```

```

    }
    return x;
}

```

Table 2.7: Listing of the Crank-Nicolson algorithm developed using the OOFEA toolkit.

The sample problem in this area is the same as that described in the static heat transfer case. The solution is time dependent, hence the density and specific heat of the material have to be provided. They are chosen such that their product ρC_p is $1J/m^3K$. The solution will reach a steady-state if one allows enough time for the simulation. The steady results are identical to those seen in figure 2.1. Instead of repeating the same plot, a plot of the temperature distribution within the cavity is presented in figure 2.3.

Figure 2.3: Temperature distribution of the solution to a 2D transient heat transfer problem where 4 heat sources are placed at several location in heated walls while the rest of the cavity convects the heat away to the environment.

2.9 Linear scalar convection

The governing equations in index notation for a linear convection initial boundary value problem are:

$$\dot{T} + u_j T_{,j} = 0 \text{ in } \Omega \quad (2.23)$$

$$T = \hat{T} \text{ on } \Gamma \quad (2.24)$$

$$T = \tilde{T} \text{ at } t = 0 \quad (2.25)$$

where Ω is the domain, and Γ is its boundary. After introducing the appropriate interpolation functions and applying the Galerkin finite element method, the resulting set of finite element equations is:

$$\mathbf{M}\dot{\mathbf{T}} + \mathbf{S}\mathbf{T} = 0 \quad (2.26)$$

where $\mathbf{T}$ is the unknown temperature vector, $\mathbf{M}$ is the mass or capacitance matrix, and $\mathbf{S}$ is the linear convection matrix. The linear convection matrix is obtained by evaluating and assembling:

$$\mathbf{S} = \iint_s \mathbf{N}^T \mathbf{u} \mathbf{J}^{-1} \mathbf{B}_{,i} \det \mathbf{J} \, ds dt \quad (2.27)$$

where $\mathbf{u}$ is the row vector of velocities u_i , i.e. $\mathbf{u}=u$ in 1D, $\mathbf{u}=[u \ v]$ in 2D.

The code in tables 2.8 and 2.9 implements equation 2.26 in one and two dimensions, respectively. After declaring the appropriate user-defined functions for the two (2) matrices involved, the `mesh` methods are used to define the geometry, assemble the components of the finite element formulation, and finally solve it using an appropriate solver. The solution algorithm employed in this sample problem is the Semi-explicit leapfrog scheme. This scheme is a two-step scheme which requires two distinct time

steps to progress. An explicit Euler scheme is used to generate the additional time step required based on the initial conditions. As can be seen in tables 2.10 and 2.12, both are implemented using the OOFEA toolkit.

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <fem.h>
#include <matrix.h>
#include <solvers.h>

temp_matrix CNVC1D_capacitance_matrix(domain_element& any_element)
{
    matrix& N=any_element.get_N_vector();
    double det_J=any_element.get_det_Jacobian();
    return transpose(N)*N*det_J;
}

temp_matrix CNVC1D_convective_matrix(domain_element& any_element)
{
    matrix& N=any_element.get_N_vector();
    matrix& B=any_element.get_B_matrix();
    matrix& inv_J=any_element.get_inv_Jacobian();
    double det_J=any_element.get_det_Jacobian();
    double velocity=200.0;
    return transpose(N)*velocity*inv_J*B*det_J;
}

int main(void)
{
    mesh mesh1;

    mesh1.load_nodal_coordinates("d:\\users\\rafa\\c-code\\data\\coords8.dat");
    mesh1.load_domain_elements("d:\\users\\rafa\\c-code\\data\\connect8.dat");
    mesh1.load_prescribed_elements(1,"d:\\users\\rafa\\c-code\\data\\prescr8.dat");

    sparse_matrix M, S;
    S=mesh1.assemble_global_matrix_from_local_ones(CNVC1D_convective_matrix);
    M=mesh1.assemble_global_matrix_from_local_ones(CNVC1D_capacitance_matrix);

    matrix guess1(mesh1.get_number_of_nodes(),1,"guess1");
    guess1.load("d:\\users\\rafa\\c-code\\data\\xinit1d.dat");
    mesh1.load_global_solution(1,"t","d:\\users\\rafa\\c-code\\data\\xinit1d.dat");

    matrix& t=mesh1.get_global_solution(1);
    Explicit_Euler(mesh1, 1, M, S, t, 0.000025, 1, 1e-12);
    Semi_Explicit_Leap_Frog(mesh1, 1, M, S, t, guess1, 0.000025, 120, 1e-12);
}
```

```

mesh1.save_global_solution(1,"t","d:\\users\\rafa\\c-code\\data\\solutnid.dat");

return 0;
}

```

Table 2.8: Listing of a 1D linear convection simulator using the OOFEA toolkit.

The sample problem in this area is nothing but the first-order one-dimensional wave equation. A triangular waveform is to travel across a domain $2m$ in length at a velocity $u = 200m/s$. The exact solution is that the waveform travels without deformation for a period of t seconds a distance of ut meters. The domain is regularly discretized into 100 elements of length $\Delta x = 2 \times 10^{-2}m$, and a time step $\Delta t = 2.5 \times 10^{-5}s$ is chosen. Given these values, the Courant number $c = u \frac{\Delta t}{\Delta x} = 0.25$ which emerges when one analyzes the stability of the time stepping schemes used to solve time-dependent problems of this sort. For the semi-explicit leapfrog scheme, this Courant number yields a solution with a very small dispersion error. Figure 2.4 plots both the initial and final waveforms after a time period $t = 250\Delta t = 0.00625s$.

```

#include <stdio.h>
#include <math.h>
#include <time.h>
#include <fem.h>
#include <matrix.h>
#include <solvers.h>

temp_matrix CVC2D_capacitance_matrix(domain_element& any_element)
{
    matrix& N=any_element.get_N_vector();
    double det_J=any_element.get_det_Jacobian();
    return transpose(N)*N*det_J;
}

temp_matrix CVC2D_convective_matrix(domain_element& any_element)
{
    matrix& N=any_element.get_N_vector();
    matrix& B=any_element.get_B_matrix();
    matrix& inv_J=any_element.get_inv_Jacobian();
    double det_J=any_element.get_det_Jacobian();
    matrix velocity(1,2,"velocity");
    velocity(1,1)=2.0/sqrt(2);velocity(1,2)=2.0/sqrt(2);
}

```

Figure 2.4: Initial and final solution to the 1D scalar convection problem. The exact solution should be identical to the original waveform translated by the distance traveled in the elapsed time. The dispersion error is expected for the Semi-explicit leapfrog scheme. Here $\Omega = [0, 2]m$, $u = 200m/s$, $\Delta x = 2 \times 10^{-2}m$, $\Delta t = 2.5 \times 10^{-5}s$ and $c = u \frac{\Delta t}{\Delta x} = 0.25$.

```

return transpose(M)*velocity*inv_J*B*det_J;
}

int main(void)
{
    mesh mesh1;

    mesh1.load_nodal_coordinates("d:\\users\\rafa\\c-code\\data\\crds7.dat");
    mesh1.load_domain_elements("d:\\users\\rafa\\c-code\\data\\cnct7.dat");
    mesh1.load_prescribed_elements(1,"d:\\users\\rafa\\c-code\\data\\diri7.dat");

    sparse_matrix M, S;
    S=mesh1.assemble_global_matrix_from_local_ones(CMVC2D_convective_matrix);
    M=mesh1.assemble_global_matrix_from_local_ones(CMVC2D_capacitance_matrix);

    matrix guess1(mesh1.get_number_of_nodes(),1,"guess");
    guess1.load("d:\\users\\rafa\\c-code\\data\\xint7.dat");
    mesh1.load_global_solution(1,"t","d:\\users\\rafa\\c-code\\data\\xint7.dat");
    matrix& t=mesh1.get_global_solution(1);

    Explicit_Euler(mesh1, 1, M, S, t, 0.00707, 1, 1e-12);
    Adams_Bashforth(mesh1, 1, M, S, t, guess1, 0.0707, 150, 1e-12);

    mesh1.save_global_solution(1,"t","d:\\users\\rafa\\c-code\\data\\solution.dat");

```

```

return 0;
}

```

Table 2.9: Listing of a 2D linear convection simulator using the OOFEA toolkit.

```

matrix& Semi_Explicit_Leap_Frog(mesh& any_mesh, const long var_num,
                                sparse_matrix& M, sparse_matrix& S,
                                matrix& xn, matrix& xnm1, double dt,
                                long steps, double tolerance)
{
    sparse_matrix A;
    A=M;
    any_mesh.apply_Dirichlet_BCs(var_num,A);
    matrix b;
    long step;
    for (step=1;step<=steps;step++)
    {
        b=(M*xnm1)-(2*dt*S*xn);
        any_mesh.apply_Dirichlet_BCs(var_num,b);
        xnm1=xn;
        PCG(A,b,xn,tolerance);
    }
    return xn;
}

```

Table 2.10: Listing of the Semi-explicit leapfrog algorithm developed using the OOFEA toolkit.

```

matrix& Adams_Bashforth(mesh& any_mesh, const long var_num,
                          sparse_matrix& M, sparse_matrix& S,
                          matrix& xn, matrix& xnm1, double dt,
                          long steps, double tolerance)
{
    sparse_matrix A;
    A=M;
    any_mesh.apply_Dirichlet_BCs(var_num,A);
    matrix b;
    long step;
    for (step=1;step<=steps;step++)
    {
        b=M*xn - S*xn*(1.5*dt) + S*xnm1*(0.5*dt);
        any_mesh.apply_Dirichlet_BCs(var_num,b);
        xnm1=xn;
        PCG(A,b,xn,tolerance);
    }
    return xn;
}

```


Table 2.11: Listing of the Adams-Bashforth algorithm developed using the OOFEA toolkit.

```

matrix& Explicit_Euler(mesh& any_mesh, const long var_num,
                      sparse_matrix& M, sparse_matrix& S,
                      matrix& x, double dt, long steps,
                      double tolerance)
{
    sparse_matrix A;
    A=condense(M);
    any_mesh.apply_Dirichlet_BCs(var_num,A);
    matrix b;
    long step;
    for (step=1;step<=steps;step++)
    {
        b=(M-dt*S)*x;
        any_mesh.apply_Dirichlet_BCs(var_num,b);
        PCG(A,b,x,tolerance);
    }
    return x;
}

```

Table 2.12: Listing of the explicit Euler algorithm developed using the OOFEA toolkit.

The two-dimensional sample problem involves the transport of a wave traveling at a constant velocity $u = 0.7071\hat{i} + 0.7071\hat{j} m/s$ over a $33m \times 33m$ rectangular domain. The initial waveform is the result of rotating a cosine hill distribution of the form

$$1 + \cos\left(\frac{2\pi x}{\lambda} - \pi\right) \quad (2.28)$$

about its symmetry axis, such that λ is spanned by 8 finite elements of length $1m$ and the waveform is centered about the point $(x, y) = (7m, 7m)$. This implies that the domain is regularly discretized with 32×32 finite elements of length $\Delta x = 1m$. The exact solution of the problem involves a translation of the waveform without distortion across the domain by a distance ut after a time period of t seconds. By selecting a time step $\Delta t = 0.07071s$, the Courant number is $c = 0.1$. Using a 2nd-order Adams-Bashforth two-step scheme for 150 time steps, an approximate solution is obtained. Figures 2.5 and 2.6 show the distribution of the waveform over the domain in its initial and final states, respectively. Figures 2.7 and 2.8 show the waveform itself in

its initial and final states, respectively. A discernible dispersion error is observed with this scheme as a wake left behind the moving wave.

Figure 2.5: Initial distribution of a cosine hill of height $2m$ which is to travel across a rectangular domain of $33m \times 33m$ at a constant velocity $u = 0.7071\hat{i} + 0.7071\hat{j}m/s$ via a 2D scalar transport process.

2.10 Nonlinear convection

The governing equations in index notation for a nonlinear convection initial boundary value problem (inviscid Burger's equation) are:

$$\dot{u} + u_j u_{,j} = 0 \text{ in } \Omega \quad (2.29)$$

$$u = \hat{u} \text{ on } \Gamma \quad (2.30)$$

$$u = \tilde{u} \text{ at } t = 0 \quad (2.31)$$

where Ω is the domain, and Γ is its boundary. After introducing the appropriate interpolation functions and applying the Galerkin finite element method, the resulting

Figure 2.6: Final distribution of the solution to the 2D first-order wave equation problem after 150 time steps with $\Delta t = 0.07071s$ and $\Delta x = 1m$, which yields a Courant number $c = 0.1$, using the 2nd-order Adams-Bashforth scheme. The solution has non-zero dispersion which can be observed as a wake behind the moving wave. The dispersion error is expected of this 2nd-order scheme.

Figure 2.7: Surface plot of initial distribution of a cosine hill of height $2m$ which is to travel across a rectangular domain of $33m \times 33m$ at a constant velocity $u = 0.7071\hat{i} + 0.7071\hat{j}m/s$ via a 2D scalar transport process.

Figure 2.8: Surface plot of final distribution of the solution to the 2D first-order wave equation problem after 150 time steps with $\Delta t = 0.07071s$ and $\Delta x = 1m$, which yields a Courant number $c = 0.1$, using the 2nd-order Adams-Bashforth scheme. The solution has non-zero dispersion which can be observed as a wake behind the moving wave. The dispersion error is expected of this 2nd-order scheme.

set of finite element equations is:

$$\mathbf{M}\dot{\mathbf{u}} + \mathbf{P}(\mathbf{u})\mathbf{u} = 0 \quad (2.32)$$

where $\mathbf{u}$ is the unknown velocity vector, $\mathbf{M}$ is the mass or capacitance matrix, and $\mathbf{P}(\mathbf{u})$ is the nonlinear convection matrix which is dependent on the unknown velocity vector. It is the same as $\mathbf{S}$ in the linear convection case, but it is necessary to emphasize its dependence on the velocity, which is now unknown.

The code in table 2.13 implements equation 2.32 in one dimension. After declaring the appropriate user-defined functions for the two (2) matrices involved, the `mesh` methods are used to define the geometry, assemble the components of the finite element formulation, and finally solve it using an appropriate solver. The solution algorithm employed in this sample problem is the two-step $\frac{2}{3}$ scheme. This scheme

is a two-step scheme which requires two distinct time steps to progress. An explicit Euler scheme is used to generate the additional time step required based on the initial conditions. In contrast with the schemes used for the linear convection case, at every time step, the nonlinear convection matrix has to be recalculated based on previously available velocity. The code in tables 2.14 and 2.15, implement both schemes.

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <fem.h>
#include <matrix.h>
#include <solvers.h>

temp_matrix MCVC1D_capacitance_matrix(domain_element& any_element)
{
    matrix& N=any_element.get_N_vector();
    double det_J=any_element.get_det_Jacobian();
    return transpose(N)*N*det_J;
}

temp_matrix MCVC1D_convective_matrix(domain_element& any_element)
{
    matrix& N=any_element.get_N_vector();
    matrix& B=any_element.get_B_matrix();
    matrix& inv_J=any_element.get_inv_Jacobian();
    matrix& local_solution=any_element.get_local_solution(1);
    double det_J=any_element.get_det_Jacobian();
    return transpose(N)*N*local_solution*inv_J*B*det_J;
}

int main(void)
{
    mesh mesh1;
    double dt;long steps;

    cout << "Enter dt: ";cin >> dt;
    cout << "Enter steps: ";cin >> steps;

    mesh1.load_nodal_coordinates("d:\\users\\rafa\\c-code\\data\\coords9.dat");
    mesh1.load_domain_elements("d:\\users\\rafa\\c-code\\data\\connect9.dat");
    mesh1.load_prescribed_elements(1,"d:\\users\\rafa\\c-code\\data\\prescr9.dat");

    matrix guess1(mesh1.get_number_of_nodes(),1,"guess1");
    guess1.load("d:\\users\\rafa\\c-code\\data\\isol1d02.dat");
    mesh1.load_global_solution(1,"u","d:\\users\\rafa\\c-code\\data\\isol1d02.dat");
```

```

sparse_matrix M, P;
M=mesh1.assemble_global_matrix_from_local_ones(MCNCVC1D_capacitance_matrix);
mesh1.disassemble_global_solution(1);
P=mesh1.assemble_global_matrix_from_local_ones(MCNCVC1D_convective_matrix);
matrix& u=mesh1.get_global_solution(1);

NL_Explicit_Euler(MCNCVC1D_convective_matrix, mesh1, 1, M, P, u, 0.000025, 1, 1e-12, 1);
NL_Two_Step_Two_Thirds(MCNCVC1D_convective_matrix, mesh1, 1, M, P, u, guess1, dt, steps, 1e-12, 1);

mesh1.save_global_solution(1,"u","d:\\users\\rafa\\c-code\\data\\fsolid02.dat");

return 0;
}

```

Table 2.13: Listing of a 1D nonlinear convection (1D inviscid Burger's equation) simulator using the OOFEA toolkit.

The sample problem in this area is that of a discontinuous wave (a shock) traveling in a one-dimensional domain of $2m$ in length. The maximum velocity of the shock wave is $u_{max} = 1m/s$. The exact solution should be that the shock moves forward within the domain without deformation a distance $u_{max}t$ after a time period of t seconds. By discretizing the domain regularly with 100 elements of length $\Delta x = 2 \times 10^{-2}m$, and after choosing a time step $\Delta t = 5 \times 10^{-3}s$, a Courant number of 0.25 is obtained. With such a Courant number, the chosen two-step $\frac{2}{3}$ solution scheme produces a slightly dissipated solution, but with no overshoot or undershoot, which is expected for this solver. Figure 2.9 is a plot of the initial and final conditions of the problem.

```

matrix& NL_Two_Step_Two_Thirds(temp_matrix (*problem_dependent_function)(domain_element&),
                                mesh& any_mesh, const long var_num,
                                sparse_matrix& M, sparse_matrix& P,
                                matrix& xn, matrix& xnm1, double dt,
                                long steps, double tolerance, long solution)
{
    sparse_matrix A;
    A=condense(M);
    any_mesh.apply_Dirichlet_BCs(var_num,A);
    matrix b;
    long step;
    for (step=1;step<=steps;step++)
    {

```

Figure 2.9: Initial and final solution to the 1D nonlinear convection (1D inviscid Burger's) problem. The exact solution should be identical to the original waveform translated by the distance traveled in the elapsed time. The solution is slightly dissipated, but there is no overshoot. This is expected for the two-step $\frac{2}{3}$ scheme being used. Here $\Omega = [0.2]m$, $u_{max} = 1m/s$ at $t = 0$, $\Delta x = 2 \times 10^{-2}m$, $\Delta t = 5 \times 10^{-3}s$ and $\frac{\Delta t}{\Delta x} = 0.25$.

```

b=(M-(2.0/3.0)*dt*P)*xn;
any_mesh.apply_Dirichlet_BCs(var_num,b);
xnm1=xn;
PCG(A,b,xn,tolerance);
b=(M-0.25*dt*P)*xnm1-(0.75*dt*P*xn);
any_mesh.apply_Dirichlet_BCs(var_num,b);
xnm1=xn;
PCG(A,b,xn,tolerance);
any_mesh.disassemble_global_solution(solution);
P=any_mesh.assemble_global_matrix_from_local_ones(problem_dependent_function);
}

return xn;
}

```

Table 2.14: Listing of the nonlinear two-step $\frac{2}{3}$ scheme using the OOFEA toolkit.

```
matrix& NL_Explicit_Euler(temp_matrix (*problem_dependent_function)(domain_element&),  
                           mesh& any_mesh, const long var_num,  
                           sparse_matrix& M, sparse_matrix& P,  
                           matrix& x, double dt, long steps,  
                           double tolerance, long solution)  
{
```

```

sparse_matrix A;
A=condense(M);
any_mesh.apply_Dirichlet_BCs(var_num,A);
matrix b;
long step;
for (step=1;step<=steps;step++)
{
  b=(M-dt*P)*x;
  any_mesh.apply_Dirichlet_BCs(var_num,b);
  PCG(A,b,x,tolerance);
  any_mesh.disassemble_global_solution(solution);
  P=any_mesh.assemble_global_matrix_from_local_ones(problem_dependent_function);
}
return x;
}

```

Table 2.15: Listing of the nonlinear explicit Euler scheme using the OOFEA toolkit.

2.11 Nonlinear convection-diffusion

The governing equations in index notation for a nonlinear convection-diffusion initial boundary value problem (viscous Burger's equation) are:

$$\dot{u} + u_j u_{,j} - \nu u_{,jj} = 0 \text{ in } \Omega \quad (2.33)$$

$$u = \hat{u} \text{ on } \Gamma \quad (2.34)$$

$$u = \tilde{u} \text{ at } t = 0 \quad (2.35)$$

where Ω is the domain, and Γ is its boundary. After introducing the appropriate interpolation functions and applying the Galerkin finite element method, the resulting set of finite element equations is:

$$\mathbf{M}\dot{\mathbf{u}} + (\mathbf{P}(\mathbf{u}) + \mathbf{K})\mathbf{u} = 0 \quad (2.36)$$

where $\mathbf{u}$ is the unknown velocity vector, $\mathbf{M}$ is the mass or capacitance matrix, $\mathbf{K}$ is the stiffness matrix, and $\mathbf{P}(\mathbf{u})$ is the nonlinear convection matrix which is dependent

on the unknown velocity vector. The stiffness matrix can be obtained by evaluating and assembling:

$$\mathbf{K} = \iint \mathbf{B}_{st}^T (\mathbf{J}^{-1})^T \nu (\mathbf{J}^{-1}) \mathbf{B}_{st} \det \mathbf{J} \, ds dt \quad (2.37)$$

In the one-dimensional case, the equation can be nondimensionalized by:

$$x^* = \frac{x}{L}, u^* = \frac{uL}{\nu}, t^* = t \frac{\nu}{L^2} \quad (2.38)$$

such that equation 2.36 can be used if one interprets u as its nondimensional counterpart and ν within $\mathbf{K}$ is unity. In the two-dimensional version, the dimensional version is used instead.

The code in tables 2.16 and implements equation 2.17 in one and two dimensions, respectively. After declaring the appropriate user-defined functions for the three (3) matrices involved, the `mesh` methods are used to define the geometry, assemble the components of the finite element formulation, and finally solve it using an appropriate solver. The solution algorithm employed is the same one used for the inviscid cases: a combination of a nonlinear explicit Euler, followed by as many two-step $\frac{2}{3}$ steps as necessary.

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <fem.h>
#include <matrix.h>
#include <solvers.h>

temp_matrix WLCD1D_stiffness_matrix(domain_element& any_element)
{
    matrix& B=any_element.get_B_matrix();
    matrix& inv_J=any_element.get_inv_Jacobian();
    double det_J=any_element.get_det_Jacobian();
    return transpose(B)*transpose(inv_J)*inv_J*B*det_J;
}

temp_matrix WLCD1D_capacitance_matrix(domain_element& any_element)
{
```

```

matrix& H=any_element.get_H_vector();
double det_J=any_element.get_det_Jacobian();
return transpose(H)*H*det_J;
}

temp_matrix WLCD1D_convective_matrix(domain_element& any_element)
{
matrix& H=any_element.get_H_vector();
matrix& B=any_element.get_B_matrix();
matrix& inv_J=any_element.get_inv_Jacobian();
matrix& local_solution=any_element.get_local_solution(1);
double det_J=any_element.get_det_Jacobian();
return transpose(H)*H*local_solution*inv_J*B*det_J;
}

int main(void)
{
mesh mesh1;
double dt;long steps;

cout << "Enter dt: ";cin >> dt;
cout << "Enter steps: ";cin >> steps;

mesh1.load_nodal_coordinates("d:\\users\\rafa\\c-code\\data\\coords10.dat");
mesh1.load_domain_elements("d:\\users\\rafa\\c-code\\data\\conect10.dat");
mesh1.load_prescribed_elements(1,"d:\\users\\rafa\\c-code\\data\\prescri10.dat");

matrix guess1(mesh1.get_number_of_nodes(),1,"guess1");
guess1.load("d:\\users\\rafa\\c-code\\data\\isol1d03.dat");
mesh1.load_global_solution(1,"u","d:\\users\\rafa\\c-code\\data\\isol1d03.dat");

sparse_matrix K, M, P;
K=mesh1.assemble_global_matrix_from_local_ones(WLCD1D_stiffness_matrix);
M=mesh1.assemble_global_matrix_from_local_ones(WLCD1D_capacitance_matrix);
mesh1.disassemble_global_solution(1);
P=mesh1.assemble_global_matrix_from_local_ones(WLCD1D_convective_matrix);
matrix& u=mesh1.get_global_solution(1);

WL_Explicit_Euler(WLCD1D_convective_matrix, mesh1, 1, M, P, u, 0.000025, 1, 1e-12, 1);
WL_Two_Step_One_Half(WLCD1D_convective_matrix, mesh1, 1, M, P, K, u, guess1, dt, steps, 1e-12, 1);
mesh1.save_global_solution(1,"u","d:\\users\\rafa\\c-code\\data\\fsol1d03.dat");

return 0;
}

```

Table 2.16: Listing of a 1D convection-diffusion (1D viscous Burger's equation) simulator using the OOFEA toolkit.

The sample problem in this area involves the convection-diffusion of a wave of the form

$$u^* = -\frac{2 \sinh x^*}{\cosh x^* - e^{-0.1}} \quad (2.39)$$

The exact solution of the problem is

$$u^* = -\frac{2 \sinh x^*}{\cosh x^* - e^{-t^*}} \quad (2.40)$$

hence the initial condition is valid. The nondimensional domain is the segment $[-9, 9]$, which is regularly discretized into 90 elements of length $\Delta x = 2 \times 10^{-1}$. Choosing a time step $\Delta t = 1 \times 10^{-2}$ and the two-step $\frac{1}{2}$ solution scheme, figure 2.10 represents the exact and approximate solutions at $t^* \approx 0.7$. As one can see, the solution is slightly dissipated, but this is expected for the chosen solver.

Figure 2.10: Exact and approximate solution to the 1D convection-diffusion (1D viscous Burger's) problem. The exact solution is of the form $-\frac{2 \sinh x}{\cosh x - e^{-t}}$. The solution is slightly dissipated, but this is expected for the two-step $\frac{1}{2}$ scheme being used. Here $\Omega = [-9, 9]$, $u = -\frac{2 \sinh x}{\cosh x - e^{-0.1}}$ at $t = 0.1$, $\Delta x = 2 \times 10^{-1}$ and $\Delta t = 1 \times 10^{-2}$.

```

#include <stdio.h>
#include <math.h>
#include <time.h>
#include <fem.h>
#include <matrix.h>
#include <solvers.h>

temp_matrix NLCD2D_stiffness_matrix(domain_element& any_element)
{
    matrix& B=any_element.get_B_matrix();
    matrix& inv_J=any_element.get_inv_Jacobian();
    double det_J=any_element.get_det_Jacobian();
    double viscosity=0.2582;
    return viscosity*transpose(B)*transpose(inv_J)*inv_J*B*det_J;
}

temp_matrix NLCD2D_capacitance_matrix(domain_element& any_element)
{
    matrix& N=any_element.get_N_vector();
    double det_J=any_element.get_det_Jacobian();
    return transpose(N)*N*det_J;
}

temp_matrix NLCD2D_convective_matrix(domain_element& any_element)
{
    matrix& N=any_element.get_N_vector();
    matrix& B=any_element.get_B_matrix();
    matrix& inv_J=any_element.get_inv_Jacobian();
    matrix& local_solution=any_element.get_local_solution(1);
    double det_J=any_element.get_det_Jacobian();
    matrix U(4,2,"U");
    U(1,1)=local_solution(1,1);U(2,1)=local_solution(2,1);
    U(3,1)=local_solution(3,1);U(4,1)=local_solution(4,1);
    U(1,2)=local_solution(1,1);U(2,2)=local_solution(2,1);
    U(3,2)=local_solution(3,1);U(4,2)=local_solution(4,1);
    return transpose(N)*N*U*inv_J*B*det_J;
}

int main(void)
{
    mesh mesh1;
    double dt;long steps;

    cout << "Enter dt: ";cin >> dt;
    cout << "Enter steps: ";cin >> steps;

    mesh1.load_nodal_coordinates("d:\\users\\rafa\\c-code\\data\\coords11.dat");
    mesh1.load_domain_elements("d:\\users\\rafa\\c-code\\data\\conect11.dat");

    matrix guess1(mesh1.get_number_of_nodes(),1,"guess1");

```

```

guess1.load("d:\\users\\rafa\\c-code\\data\\isol2d01.dat");
mesh1.load_global_solution(1,"U","d:\\users\\rafa\\c-code\\data\\isol2d01.dat");

sparse_matrix K, M, P;
K=mesh1.assemble_global_matrix_from_local_ones(NLCD2D_stiffness_matrix);
M=mesh1.assemble_global_matrix_from_local_ones(NLCD2D_capacitance_matrix);
mesh1.disassemble_global_solution(1);
P=mesh1.assemble_global_matrix_from_local_ones(NLCD2D_convective_matrix);
matrix& u=mesh1.get_global_solution(1);

NL_Explicit_Euler(NLCD2D_convective_matrix, mesh1, 0, M, P, u, 0.000025, 1, 1e-12, 1);
NL_Two_Step_One_Half(NLCD2D_convective_matrix, mesh1, 0, M, P, K, u, guess1, dt, steps, 1e-12, 1);

mesh1.save_global_solution(1,"U","d:\\users\\rafa\\c-code\\data\\fsol2d01.dat");

return 0;
}

```

Table 2.17: Listing of a 2D convection-diffusion (2D viscous Burger's equation) simulator using the OOFEA toolkit.

In the two-dimensional case, a two-dimensional shock is introduced in a $33m \times 33m$ domain. Its maximum velocity is $u_{max} = 0.7071\hat{i} + 0.7071\hat{j}m/s$. Given a non-zero kinematic viscosity $\nu = 0.2582m^2/s$, the solution should be a dissipated version of the original shock. The domain is discretized regularly with 33×33 elements of length $\Delta x = 1m$. Using a two-step $\frac{1}{2}$ scheme with a time step of $\Delta t = 0.125s$, a Courant number of 0.125 is obtained, which yields a solution which has a small but non-zero level of artificial diffusion. Figures 2.11 and 2.12 are plots which represent the velocity distribution of the initial shock and its final shape after 150 time steps, respectively. Similarly, figures 2.13 and 2.14 are surface plots of the initial and final states of the shock after the same time period, respectively.

Figure 2.11: Initial velocity distribution of an unit height discontinuity which is to be transported across the rectangular domain via a 2D convection-diffusion process.

Figure 2.12: Final velocity distribution of the solution to the 2D convection-diffusion problem after 150 time steps with $\Delta t = 0.125$ and a kinematic viscosity $\nu = 0.2582$ using the two-step $\frac{1}{2}$ scheme.

Figure 2.13: Surface plot of initial velocity of an unit height discontinuity which is to be transported across the rectangular domain via a 2D convection-diffusion process.

Figure 2.14: Surface plot of final velocity of the solution to the 2D convection-diffusion problem after 150 time steps with $\Delta t = 0.125s$ and a kinematic viscosity $\nu = 0.2582m^2/s$ using the two-step $\frac{1}{2}$ scheme.

Chapter 3

Introduction to Natural Convective Flows and Turbulence

3.1 Introduction

In chapter 2 a number of problems of increasing difficulty and importance in the computational fluid dynamics (CFD) arena were discussed, implemented and solved using the OOFEA toolkit. It should be clear by this point that CFD applications can be generated relatively quickly and with minor effort. However, truly complex CFD applications usually involves a larger number of coupled governing equations, a larger number of variables, time and space varying boundary conditions, and a number of additional complications which might not seem directly manageable via the OOFEA toolkit and object-oriented techniques. To demonstrate otherwise, the remaining chapters of this study will concentrate on a particular area of the CFD arena which introduces many of these additional complications: the simulation of laminar and turbulent natural convective flows in enclosures. A natural convection simulator will be developed using the OOFEA classes and methods, it will be used to solve representative problems in this area, and a numerical study of such flows will be presented.

3.2 Natural convective flows

A large group of fluid motions encountered in every day natural and engineering systems is buoyancy induced. The buoyancy forces inducing these motions are the result

of density differences in the fluid. Such density differences can be attributed to a number of mechanisms: chemical concentration differences, phase changes, temperature variations, and others.

Buoyancy induced flows are different from other everyday flows which are driven by other, usually man-generated forces like pumps, fans, pistons, and many other mechanical devices. The differences lie in the fact that, as opposed to the latter, where the forced motion dominates most of the flow and is well known *a priori*, in buoyant flows almost nothing is known about the final state of motion *a priori*. In fact, most buoyant flows can be generated from rest conditions. This is so because of the direct coupling between the motion and the mechanism of density change. In that sense, transient as well as steady-state behavior is of major importance.

Buoyancy induced flows can be classified into two (2) categories based on the extent of the medium in which they arise. *External* buoyant flows are those which arise in very extensive mediums where external effects such as boundaries are not present. These are commonly generated by the presence of local inhomogeneities of density due to any of the mechanisms previously mentioned. *Internal* buoyant flows are those which arise in regions where the medium is completely or partially bounded by surfaces.

However, other classifications of buoyant flows are not as straightforward, and are quite controversial. *Free* convection is a term used to sometimes refer to external flows, or both internal and external flows. Furthermore, *natural* convection is a term which is also used to describe both internal and external flows [15]. This study will use the latter nomenclature. In particular, the rest of the study will concentrate on natural convection induced by temperature differences only. The choice for thermally

buoyant flows is partially due to their common appearance in mechanical engineering applications related to energy collection, power generation, and heat rejection.

3.3 Unresolved problems of natural convection

A number of aspects of natural convection flows have been identified by leading experts in the field as topical areas worthy of in-depth study and research [16][17]. Among all these possible areas of study, three (3) are of special interest: (1) flow in enclosures, (2) turbulence modeling, and (3) numerical methods.

In this and a previous study [18], the main objective will be to analyze thermally buoyant internal flows. This coincides with the interest in thermally buoyant flows, since most of the mechanical systems in which these are involved, the medium is fully or partially enclosed. In particular, there is interest in describing the motions that arise in these types of cavities as the coupling between momentum and energy exchange becomes more pronounced. There is also interest in describing the differences between two-dimensional and three-dimensional motions in these enclosures. Furthermore, the effects of baffles and internal bodies is not well understood and very little work has been done. These questions in particular were addressed previously [18].

On the other hand, some important questions remain. In particular, it is of interest to show how to use turbulence modeling to describe the flow motions when the momentum-energy exchange is very large and therefore, the flows are beyond the laminar range. It is also of interest to show how the finite element method can be used to attack these problems, and it is even further interest to demonstrate the use of techniques capable of describing the transient behavior of such flows. These are the questions to be addressed in this study.

3.4 Natural convection in enclosures

Two-dimensional flows in vertical, horizontal and inclined rectangular cavities have been studied extensively due to the considerable simplifications which can be made in comparison to full three-dimensional counterparts. Although these flows are in general three-dimensional in general, two-dimensional approximations are often valid when the third dimension in question is not too small in comparison with the other two. In addition, three-dimensional experiments are lacking, although there is an increase in numerical simulations in three-dimensions. For these reasons, most of what is known about these types of flows is limited to two-dimensional experimental and numerical studies. In particular, one can divide these contributions into two major groups: laminar and turbulent studies.

3.4.1 Laminar regime - Literature Survey

The geometry in consideration is generally a vertical rectangular cavity of height H and width W . Such a cavity is defined as an enclosure bounded by two vertical surfaces which are held at different temperatures, while the remaining parallel horizontal surfaces are either insulated or linearly varying in temperature between those of the vertical boundaries.

For flows generated in this type of cavity, three (3) dimensionless parameters become important. These are the Rayleigh number,

$$Ra = \frac{\rho\beta g\Delta TW^3}{\mu\alpha} \quad (3.1)$$

which represents the ratio of buoyant forces to viscous forces, the Prandtl number,

$$Pr = \frac{\nu}{\alpha} \quad (3.2)$$

which represents the ratio of viscous dissipation to thermal conduction, and the aspect ratio,

$$A = \frac{H}{W} \quad (3.3)$$

where g is the gravitational acceleration; μ , ν , α , β are the dynamic viscosity, the kinematic viscosity, the thermal diffusivity, and the coefficient of thermal expansion of the fluid, respectively; ΔT is the temperature difference between the vertical walls.

Given a finite aspect ratio A , different applications for high Rayleigh number flows can be identified based on the value of the Prandtl number [19][20]: applications involving fluids of $Pr \approx 1$ can be found in food sterilization, fires in enclosures, crystal growth and meteorology; applications involving fluids of $Pr \ll 1$ can be found in semi-conductors, liquid metals, and crystal growth. On the other hand, as the A approaches zero (0) and the Pr approaches ∞ , applications in geophysics can be found.

One of the earliest experimental studies of this problem was performed by Mull & Reiher and has been discussed by Jakob [21] and Batchelor [22]. Others in the same period include Lewis [23] and Ostrach [24]. Lewis analyzed cases of $Ra < 1$ where there is no boundary layer phenomena. Ostrach investigated the asymptotic case of infinite aspect ratio. Batchelor analyzed the effects of the aspect ratio at low Rayleigh numbers using a perturbation scheme. He concluded that for large A , the main heat transfer mechanism is that of conduction for $Ra < 10^3$, while as A approaches ∞ , conduction is the only mechanism for any Ra .

Until 1960, for large Ra , the interior region of the cavity away from the boundary layers at the walls, known as the core, was assumed to be isothermal and to rotate as a solid body. This assumption delayed the understanding of the problem since it was later proven wrong numerically and experimentally. The experimental studies

by Eckert and Carlson [25] for a variety of aspect ratios ($A = 2.1 - 46.7$) in air at Ra ranging from 10^2 to 10^5 , and those by Elder [26] for large $Pr \approx 1000$ in silicone oil with A ranging between 1 and 60, and Ra up to 10^8 , showed that at low Ra , conduction was dominant and a linear temperature distribution was present between the vertical walls away from the horizontal ends. However, for large Ra they observed the formation of boundary layers at the vertical boundaries, and that the core was nearly stagnant and thermally stratified with essentially horizontal isotherms, which contradicted the assumptions of Ostrach and Batchelor. Furthermore, the numerical studies by Wilkes [27], and De Vahl Davis [28] predict the observed core behavior.

Later on, De Vahl Davis and Mallinson [29] furthered their numerical studies by analyzing cavities with aspect ratios up to 20, and Ra up to 10^6 , for $Pr \approx 1000$. They predicted the same secondary and tertiary flows observed by Elder experimentally.

Then, in 1983, De Vahl Davis [30] presented a benchmark numerical solution for the case $A = 1$, $Pr = 0.71$ and $Ra = 10^3 - 10^6$, which has since been used as a validation tool for computational fluid dynamic-heat transfer computer codes. Saitoh and Hirose [31] presented a similar benchmark with fourth-order accuracy in 1989. Three-dimensional numerical solutions of these flows have also been presented in the literature [32][33][34][35][36][18].

Although this is a small portion of the available literature, it provides a picture of the advances in this area. On the other hand, another body of literature exists which is related to the experimental and numerical study of turbulent flows in these cavities. In order to introduce these contributions, the next section is an attempt to introduce the topic of turbulence and its modeling. Afterwards, the contributions in the literature can be better characterized.

3.4.2 Turbulent regime - Theory

When the nondimensional Reynolds number,

$$Re = \frac{\rho u d}{\mu} \quad (3.4)$$

which represents the ratio of inertia forces to viscous forces, reaches a cut-off value Re_c , the flow in question undergoes a drastic change from a well defined simple laminar flow to an irregular and seemingly random turbulent flow. Although there is no appropriate complete definition of turbulence, two (2) complementary definitions describe some of its physical as well as the statistical properties. Bradshaw [37] defines turbulence as

Turbulence is a three-dimensional time-dependent motion in which vortex stretching causes velocity fluctuations to spread to all wavelengths between a minimum determined by viscous forces and a maximum determined by the boundary conditions of the flow. It is the usual state of fluid motion except at low Reynolds numbers.

Hinze [38] defines it as

Turbulent fluid motion is an irregular condition of flow in which various quantities show a random variation with and space coordinates, so that statistically distinct average value can be discerned.

Physically, these definitions indicate that the rotation and deformation of fluid elements at a particular length scale produce similar motions and deformations on fluid elements of smaller length scales in every direction, which themselves induce motions at lower scales, and so on, producing a continuous cascade of kinetic energy which would continue indefinitely if it was not for the fact that the lowest possible scale is that of molecular motion, where viscosity transforms the cascaded energy into internal energy. Similarly, the largest possible length scale of these motions is dependent on the problem domain. Mathematically, although these motions appear irregular

and random, they are the result of physical processes that impart them spatial and temporal correlations which can be discerned via statistical analysis.

Although the three-dimensional Navier-Stokes equations are believed to govern three-dimensional turbulent motions, it has been estimated that to resolve all the important features of a typical three-dimensional turbulent flow domain, a typical finite difference solution scheme would need to solve the governing equations for over one trillion grid nodes. This is beyond the general capacity of today's computers. This shows that a complete direct numerical simulation (DNS) of the Navier-Stokes equations is still beyond our reach. Nevertheless, theoretical as well as computational advances have yielded DNS results for several significant lengthscales already [39].

On the other hand, the statistical approach developed by the immortalized Reynolds [40], known as Reynolds Averaging, has been the mathematical tool of choice for the analysis and simulation of turbulent flows.

3.4.2.1 Reynolds Averaging

Via the Reynolds Averaging procedure, all fluid properties are expressed as the sum of a mean value and a fluctuating part. Given property ϕ , it can be decomposed into

$$\phi = \bar{\phi} + \phi' \quad (3.5)$$

where $\bar{\phi}$ is the mean part, and ϕ' is the fluctuating part.

The meaning of the mean and fluctuating parts, and of the Reynolds Averaging procedure is dependent on the type of averaging used. Furthermore, the type of averaging depends on the type of turbulence being analyzed. In particular, *time averaging* is appropriate for *stationary turbulence*, which occurs in flows where, on the average, the turbulence does not vary with time. *Spatial averaging* is more appropriate for *homogeneous turbulence*, which occurs in flows where, on the average, the turbulence

is uniform in all spatial directions. *Ensemble averaging* is the most general type of averaging, and can be interpreted as the average value of a measurement performed in N identical experiments. The *ergodicity hypothesis* is the common assumption that the turbulence is both stationary and homogeneous, conditions that ensure all three types of averaging are equal [41].

In a Newtonian, incompressible, viscous, thermal, constant properties flow, the fluid velocities, the pressure and the temperature are decomposed into their mean and fluctuating components. In index notation, these are

$$u_i = \bar{u}_i + u'_i \quad (3.6)$$

$$p = \bar{p} + p' \quad (3.7)$$

$$\theta = \bar{\theta} + \theta' \quad (3.8)$$

The governing conservation of mass, momentum and energy equations can be written as

$$u_{j,j} = 0 \quad (3.9)$$

$$\rho(u_{i,t} + u_j u_{i,j}) = \sigma_{ij,j} + f_i(\theta) \quad (3.10)$$

$$(\theta_{,t} + u_j \theta_{,j}) = (\alpha \theta_{,j})_{,j} \quad (3.11)$$

where $f_i(\theta)$ is a body force per unit mass term which is usually dependent on the temperature and couples the momentum transfer with the heat transfer. The term σ_{ij} is the *stress tensor* and has the form

$$\sigma_{ij} = -p\delta_{ij} + \mu(u_{i,j} + u_{j,i}) \quad (3.12)$$

where ρ is the density, μ is the dynamic viscosity, α is the thermal diffusivity, $u_{i,j}$ is the velocity gradient, and δ_{ij} is the Kronecker delta which is one if $i = j$ and zero otherwise. After the Reynolds Averaging procedure, the resulting equations can be written as [42]

$$\bar{u}_{j,j} = 0 \quad (3.13)$$

$$\rho(\bar{u}_{i,t} + \bar{u}_j \bar{u}_{i,j}) = \bar{\sigma}_{ij,j} + f_i(\theta) \quad (3.14)$$

$$(\bar{\theta}_{,t} + \bar{u}_j \bar{\theta}_{,j}) = (\alpha \bar{\theta}_{,j} - \overline{u'_i \theta'})_{,j} \quad (3.15)$$

where $\bar{\sigma}_{ij}$ is the *total mean stress tensor* which has the form

$$\bar{\sigma}_{ij} = -\bar{p}\delta_{ij} + \mu(\bar{u}_{i,j} + \bar{u}_{j,i}) - \rho \overline{u'_i u'_j} \quad (3.16)$$

The term $-\rho \overline{u'_i u'_j}$ is called the *Reynolds stress tensor*, and will be labeled τ_{ij} . It is symmetric, its diagonal components are normal stresses (pressures). The off-diagonal components are shear stresses. As one can see from the equations, the Reynolds averaging procedure has isolated the effects of fluctuations on the mean flow. The $\overline{u'_i u'_j}$ part of the Reynolds stress is non-zero if the fluctuations are correlated, and zero if the fluctuations are uncorrelated. Its divergence can be interpreted as the mean transport of fluctuating momentum by turbulent velocity fluctuations and is analogous to the convection terms in the momentum equation. If the fluctuations of the velocity were uncorrelated, there would be no turbulent momentum transfer. Experience demonstrates that momentum transfer is a key feature of turbulent motions, hence these are likely non-zero correlations in any practical turbulent flow. It is also well known that the normal Reynolds stresses contribute little to the transport of mean momentum while the shear Reynolds stresses are dominant in the transport.

In a similar fashion, the divergence of the $\overline{u'_i \theta'}$ can be interpreted as the mean transport of fluctuating energy by turbulent velocity and temperature fluctuations, and is analogous to the convection terms in the energy equation. Furthermore, the analogy between the turbulent mean momentum and energy transport is the analytical foundation for the idea that turbulence transports heat, chemical species, particles, etc. in much the same way it transports momentum.

Although the Reynolds Averaging procedure has isolated the contributions of the turbulent fluctuations to the mean flow variables, it does not produce enough equations to close the system. In fact, any attempt to generate new equations by taking moments of the Navier-Stokes equations introduces higher and higher order correlations between the fluctuating flow quantities. At no point will this process yield a balance between number of unknowns and number of equations [41]. This is due in particular to the nonlinear convection terms in the Navier-Stokes equations. The problem of having more unknowns than equations after the Reynolds Averaging process is familiarly called the *closure problem*. This is where *turbulence modeling* enters the picture. The function of turbulence modeling is to design approximations for the unknown correlations in terms of determinable flow properties such that in the end a sufficient number of equations is available to close the system.

3.4.2.2 The Boussinesq assumption

Turbulence models can be broadly divided into two categories: those which incorporate the Boussinesq assumption, and those that do not. Recalling that the total mean stress tensor $\bar{\sigma}_{ij}$ is of the form

$$\bar{\sigma}_{ij} = -\bar{p}\delta_{ij} + \mu(\bar{u}_{i,j} + \bar{u}_{j,i}) - \overline{\rho u'_i u'_j} \quad (3.17)$$

it can be rewritten as

$$\bar{\sigma}_{ij} = -\bar{p}\delta_{ij} + 2\mu S_{ij} - \overline{\rho u'_i u'_j} \quad (3.18)$$

where S_{ij} is the mean strain-rate tensor

$$S_{ij} = \frac{1}{2} (\bar{u}_{i,j} + \bar{u}_{j,i}) \quad (3.19)$$

The Boussinesq assumption involves writing the Reynolds stress tensor in a form analogous to that of the Stokes viscosity law such that

$$\tau_{ij} = 2\mu_t S_{ij} - \frac{2}{3}\rho k \delta_{ij} \quad (3.20)$$

where k is called the kinetic energy per unit mass of the turbulent fluctuations. It has the form

$$k = \frac{1}{2} \overline{u'_i u'_i} \quad (3.21)$$

The term involving k is necessary to obtain the appropriate trace of the Reynolds stress tensor ($\text{tr } \tau_{ij}$). Recall that the trace of a tensor is defined as the sum of its diagonal entries. In other words, since for an incompressible flow, $S_{ii} = 0$, then

$$\tau_{ii} = -\overline{\rho u'_i u'_i} = -2\rho k = -\frac{2}{3}\rho k \delta_{ii} = 2\mu_t S_{ii} - \frac{2}{3}\rho k \delta_{ii} = \tau_{ii} \quad (3.22)$$

In addition, the term involving k can be considered as turbulent pressure. The term μ_t is called the *turbulent eddy-viscosity*. The turbulent eddy-viscosity is not a fluid property like the dynamic viscosity, but it is a property of the flow, hence it is a scalar field of the flow. The Boussinesq assumption attempts to characterize the increased mixing and thus diffusion of momentum of turbulent flows by an appropriate increase in the fluid's effective viscosity. Hence, with the Boussinesq assumption in effect, the total mean stress tensor $\bar{\sigma}_{ij}$ can be rewritten as

$$\bar{\sigma}_{ij} = -\left(\bar{p} + \frac{2}{3}k\right) \delta_{ij} + 2(\mu + \mu_t) S_{ij} \quad (3.23)$$

In terms of the energy equation, the Boussinesq assumption involves writing the turbulent heat fluxes in a form analogous to that of the Fourier heat conduction law, such that

$$-\rho \overline{u'_i \theta'_i} = \rho \alpha_t \theta_{,i} \quad (3.24)$$

where α_t is the *turbulent eddy-diffusivity*. The turbulent eddy-diffusivity is not a property of the fluid, but a property of the flow, hence it is a scalar field. To complete the Boussinesq assumption, the turbulent eddy-diffusivity is related to the turbulent eddy-viscosity by the *turbulent Prandtl number* σ_t

$$\sigma_t = \frac{\mu_t}{\rho \alpha_t} \quad (3.25)$$

where the turbulent Prandtl number is assumed to be constant for the entire flow. For ordinary fluids ($Pr \geq 1$), a constant value of order one ($\sigma_t = 0.9 - 1.0$) is appropriate [43]. This is called the *Reynolds analogy*. Substituting 3.25 into 3.24 yields

$$-\rho \overline{u'_i \theta'_i} = \frac{\mu_t}{\sigma_t} \theta_{,i} \quad (3.26)$$

3.4.2.3 Turbulence models using the Boussinesq assumption

After the Boussinesq assumption is in effect, the process of turbulence modeling reduces to the intelligent determination of the eddy-viscosity. These models can be further classified by the number and type of closure equations that are used to determine the eddy-viscosity [44] [41] [45]:

- *Zero-equation models* - use algebraic equations for closure. The most common is the Prandtl's mixing-length hypothesis, which is simply an analogy between the molecular viscosity, based on a thermal velocity and the mean free path, and the turbulent eddy-viscosity based on an idealized picture of turbulence

where fluid particles mix and exchange momentum with a *mixing velocity* and a *mixing length*. Provided that the mixing velocity is a function of the mixing length, Prandtl proposed that the eddy-viscosity had the form

$$\mu_t = \rho l_{mix}^2 |\bar{u}_{i,j} + \bar{u}_{j,i}| \quad (3.27)$$

This formulation is thus incomplete since the calculation of the eddy-viscosity has been substituted by the empirical prescription of the mixing length. Prandtl suggested that the mixing length varied proportionately with the distance, but it can be shown that this assumption is only valid for a limited region of the turbulent boundary layer. Variations to the Prandtl's proposals have been made in the past, but still, although the model is very simple and computationally inexpensive, prescription of the mixing length is not straightforward for complex flows.

- *One-half equation models* - use ordinary differential equations for closure. They are still incomplete models since the mixing length or some other parameter must be known *a priori*, or a value is chosen and has to iterated upon.
- *One-equation models* - use one partial differential equation for closure. They are also incomplete, since distributions of the mixing length have to provided *a priori*.
- *Two-equation models* - use two partial differential equations for closure. Via dimensional arguments, the eddy-viscosity is found to depend on both a turbulent length scale and a turbulent time scale. Two-equation models propose the actual form of this dimensional relation and provide partial differential equations to solve for these scales without the need for *a priori* information. These methods have been classified depending on the parameters that are solved for via the

two governing PDEs. In any of the possible two-equation models, the result is an equation which relates the eddy-viscosity to the calculated flow properties. Two of these methods are the k - ω and the k - ϵ models. The first one computes the kinetic energy per unit mass of the turbulent fluctuations or simply the turbulence kinetic energy, and a turbulent vorticity scale, while the second method computes the turbulence kinetic energy and the rate of dissipation of turbulence kinetic energy per unit mass.

In this dissertation, the k - ϵ turbulence model has been selected among all others due to its popularity, its completeness, its previously successful use in the area of natural convection and the previously successful experience of our team with its implementation.

3.4.2.4 The k - ϵ turbulence model

The two variables which give the model its name can be defined in terms of the fluctuating properties of the flow in question as

$$k = \frac{1}{2} \overline{u'_i u'_i} \quad (3.28)$$

$$\epsilon = \nu \overline{u'_{i,k} u'_{i,k}} \quad (3.29)$$

where $\nu = \mu/\rho$ is the kinematic viscosity. These turbulence quantities are determined via the solution of advection-diffusion type transport equations. Presentation of the particular form of these equations for the solution of turbulent natural convective flows in enclosures will be postponed until chapter 4.

It is sufficient to say here that the formulation used is that proposed by Launder and Spalding [46], but simply altered to take into account the production of turbulence energy and the rate of dissipation of turbulence kinetic energy by buoyancy effects.

Even the original equation coefficients are the same except for several ones associated with the new buoyancy terms. Physically, the k equation represents the way in which the turbulence gains energy from the mean flow, cascades it down to lower and lower scales, until it is dissipated via viscous effects, which is represented by the ϵ equation.

In general, the initial conditions are dependent on the type of flow of interest. In the case of natural convective flows in enclosures, the initial conditions used have been very small uniform distributions of k and ϵ . Once any particular simulation yields a converged value for both k and ϵ , these values are used as initial conditions instead.

The k - ϵ model used in this dissertation is a high Reynolds number model and is only valid in the fully turbulent regime. Since, close to any boundary there exists a small fluid layer where viscous effects dominate over turbulent ones, these two-equation model is not valid. This small fluid layer is called the *viscous sublayer*. Although modified k - ϵ models valid for low Reynolds number regions exist, the most common approach is to use the standard k - ϵ model, and specify the boundary conditions for k and ϵ away from the wall using what are called *wall functions*.

Computationally, this means first that the boundary conditions will be applied at the grid point closest to the wall but not in it, and that the boundary conditions will have to be recalculated every time the flow characteristics change. In particular, for natural convective flows in enclosures, Dirichlet boundary conditions can be derived for both k and ϵ as long as they are placed close enough to the wall to be in the fully turbulent log layer. By assuming that the behavior of turbulence in the small layer of fluid next to the wall is such that the rate at which turbulence kinetic energy is produced is the same at which it is being dissipated, then the turbulence is in equilibrium. In that case, the k - ϵ model reduces to the mixing length hypothesis and

the universal log-law is assumed to be valid. Using these assumptions and arguments, boundary conditions for k and ϵ can be derived in the form

$$k_p = \frac{u_*^2}{C_\mu^{1/2}} \quad (3.30)$$

$$\epsilon_p = \frac{u_*^3}{\kappa y_p} \quad (3.31)$$

where y_p is the normal distance between the boundary and the grid point at which the boundary condition will be applied, $C_\mu = 0.09$, $\kappa = 0.41$ is the von Karman constant, and $u_* = \sqrt{\tau_w/\rho}$ is the friction velocity which can be approximated numerically by the gradient of the tangential velocity at the point where the Dirichlet boundary condition is to be applied [44][47][48][46].

3.4.3 Turbulent regime - Literature Survey

In this section we will concentrate on trying to present a small picture of some of the work that has been done in relation to the investigation of turbulent natural convective flows in enclosures. In particular, previous work performed in the vertical cavity with differentially heated walls described previously is of interest here.

Fraikin *et al.* [49] apparently obtained the first stable solution for fundamentally two-dimensional turbulent natural convection using the k - ϵ model. They studied a square channel with isothermally heated and cooled vertical walls and linear temperature profiles along the lower and upper horizontal boundaries. Their calculations were for air at Grashof numbers between 10^7 and 10^8 . The Grashof number Gr is defined as the ratio Ra/Pr . Farouk and Güçeri [50] also used the k - ϵ model to compute the turbulent natural convection in the horizontal concentric annulus between a heated inner and cooled outer cylinder at Rayleigh numbers up to 10^7 . Later on, Markatos and Pericleous [51] studied the thermally driven cavity problem for Rayleigh num-

bers between 10^3 and 10^{16} and a Prandtl number of 0.71. Their study attacks the steady form of the governing equations, including the steady form of the k - ϵ model. Similarly, Güçeri and Farouk [52], used a steady formulation to solve both laminar and turbulent flows in several geometries. On the other hand, Ozoe *et al.* [47], used the time dependent k - ϵ equations to obtain numerical results for differentially heated vertical rectangular cavities at several Rayleigh numbers between 10^6 and 10^{11} for Prandtl numbers between 6 and 10. They performed an extensive sensitivity study of the constants used in the k - ϵ equations. In their study, they also observed transient and steady oscillations in the core which, of course, cannot be observed using steady formulations. It seems that Staehle and Hahne [53] also observed these oscillations before. Later on, Ozoe *et al.* extended their solution scheme to three-dimensions and performed an analysis on a cubic cavity heated from the bottom and from the side. However, they did not make use of their previous sensitivity results to change the constants of the k - ϵ model.

The importance of the two-dimensional driven cavity problem cannot be ignored if one examines some of the work done in the past five years. Paolucci [39] performed a direct numerical simulation of the driven cavity problem which suggests that these kinds of problems can be resolved to some extent with today's techniques, specially if adaptive meshing techniques are used. Barakos and Mitsoulis [54] presented a study of both laminar and turbulent results in the thermally driven cavity for Ra between 10^3 and 10^{10} and attempting to use different types of wall function for several or all problem variables, but found that introducing wall functions for the velocity and temperature pushed the transition to turbulence down to the $Ra = 10^6$ region. Furthermore, just in the last year, Henkes and Hoogendoorn [55] presented a paper which represents the effort of 10 international groups which attempt to solve the

thermally driven cavity problem in two-dimensions for one particular case, $Ra = 5 \times 10^{10}$, $Pr = 0.71$ and $A = 1$. The results presented involve solutions using a multitude of in-house and commercial codes, solver implementations, grid resolutions, boundary condition variations. They also report a multitude of numerical difficulties, differences in the solutions, problems with commercial codes and necessary tricks for the successful simulation of the steady formulation of the governing equations. Finally, Pérez-Segarra *et al.* [56] have solved a set of two-dimensional thermally driven cavity problems which internal bodies and also inflow and outflow conditions using a time-dependent formulation of the governing equations. All these works made use of the k - ϵ turbulence and some of its low-Reynolds number variants to evade the use of wall functions.

Although this literature survey and introduction chapter is by no means complete, it introduces the issues, approaches, and some of the previous work done in the area. The following chapters will describe in detail the governing equations, the numerical methods, the computational implementation and the numerical studies performed using the OOFEA toolkit to develop a finite element based turbulent and laminar natural convection simulator for flows in enclosures.

Chapter 4

Mathematical Models, Solution Algorithms and their OOFEA Implementation

4.1 Problem formulation and governing equations

In the current study, the fluid in question is considered to be incompressible and viscous, and the domain is three-dimensional (3-D). To examine the fluid flow, we will solve the unsteady Reynolds averaged Navier-Stokes equations coupled with the energy equation, all in their primitive variable $(\bar{u}-\bar{v}-\bar{w}-\bar{p}-\bar{\theta})$ form. To account for turbulent motions, a $k-\epsilon$ turbulence model is also included. In this formulation $\bar{u}$, $\bar{v}$ and $\bar{w}$ represent the average fluid velocities in the x , y and z directions respectively, while $\bar{p}$ and $\bar{\theta}$ represent the average pressure and average temperature, and k and ϵ represent the kinetic energy of the turbulence and the dissipation of turbulence, in that order.

By introducing a reference velocity

$$U = \frac{\alpha}{W} \quad (4.1)$$

where W is the representative length scale of the domain, usually the width of the cavity in question, the set of non-dimensional variables

$$u^* = \frac{\bar{u}}{U} \quad v^* = \frac{\bar{v}}{U} \quad w^* = \frac{\bar{w}}{U} \quad (4.2)$$

$$\theta^* = \frac{(\bar{\theta} - \theta_r)}{(\theta_h - \theta_c)} \quad (4.3)$$

$$x^* = \frac{x}{W} \quad y^* = \frac{y}{W} \quad z^* = \frac{z}{W} \quad (4.4)$$

$$p^* = \frac{\bar{p}}{\bar{\rho}U^2} \quad (4.5)$$

$$t^* = \frac{tU}{W} \quad (4.6)$$

$$k^* = \frac{k}{U^2} \quad (4.7)$$

$$\epsilon^* = \frac{\epsilon W}{U^3} \quad (4.8)$$

is generated, as well as the set of dimensionless parameters, coefficients and predefined constants

$$Ra = \frac{\bar{\rho}\bar{\beta}g(\theta_h - \theta_c)W^3}{\mu\alpha} \quad (4.9)$$

$$Pr = \frac{\nu}{\alpha} \quad (4.10)$$

$$A = \frac{H}{W} \quad (4.11)$$

$$\nu_t^* = \frac{\nu_t}{\alpha} = C_\mu \frac{(k^*)^2}{\epsilon^*} \quad , \text{ Prandtl-Kolmogorov relation} \quad (4.12)$$

$$\sigma_t = 0.90 \sim 1.00 \quad \sigma_k = 1.00 \quad \sigma_\epsilon = 1.30 \quad (4.13)$$

$$C_\mu = 0.09 \quad C_{1\epsilon} = 1.44 \quad C_{2\epsilon} = 1.92 \quad (4.14)$$

$$C_{3\epsilon} = \begin{cases} 0 & \text{at vertical walls} \\ 1 & \text{at horizontal walls} \end{cases} = \tanh \left| \frac{u^*}{v^*} \right| \quad (4.15)$$

where, H is the height of the cavity, Ra is the Rayleigh number, P is the Prandtl number, A is the aspect ratio, and in particular, ν_t^* is the turbulent viscosity which

incorporates all Reynold stresses into the diffusion terms of all transport equations. Given these, we can write the momentum, energy and conservation of mass equations in index notation as

$$u_{i,t}^* + u_j^* u_{i,j}^* = -p_{,i}^* - \frac{2}{3} k_{,i}^* + \left[(Pr + \nu_t^*) (u_{i,j}^* + u_{j,i}^*) \right]_{,j} + \delta_{i2} Ra Pr \theta^* \quad (4.16)$$

$$\theta_{,t}^* + u_j^* \theta_{,j}^* = \left[\left(1 + \frac{\nu_t^*}{\sigma_t} \right) \theta_{,j}^* \right]_{,j} \quad (4.17)$$

$$u_{j,j}^* = 0 \quad (4.18)$$

where the u_i^* or u_j^* notation represents the nondimensional average velocities u^* , v^* and w^* , the $_{,i}$ or $_{,j}$ notation represents partial derivation with respect to the nondimensional directions x^* , y^* and z^* , the $_{,t}$ notation represents partial derivation with respect to the nondimensional time t^* , and the δ_{ij} notation is the Kronecker Delta which is defined as:

$$\delta_{ij} = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases} \quad (4.19)$$

Using the same notation, the nondimensionalized k - ϵ turbulence equations can be written as

$$k_{,t}^* + u_j^* k_{,j}^* = \left[\left(Pr + \frac{\nu_t^*}{\sigma_k} \right) k_{,j}^* \right]_{,j} + P_k^* + G_k^* - \epsilon^* \quad (4.20)$$

$$\epsilon_{,t}^* + u_j^* \epsilon_{,j}^* = \left[\left(Pr + \frac{\nu_t^*}{\sigma_\epsilon} \right) \epsilon_{,j}^* \right]_{,j} + \frac{\epsilon^*}{k^*} C_{1\epsilon} P_k^* + \frac{\epsilon^*}{k^*} C_{1\epsilon} C_{3\epsilon} G_k^* - C_{2\epsilon} \frac{(\epsilon^*)^2}{k^*} \quad (4.21)$$

$$P_k^* = \nu_t^* \left[(u_{i,j}^* + u_{j,i}^*) u_{i,j}^* \right] \quad (4.22)$$

$$G_k^* = -\frac{\nu_t^*}{\sigma_t} Ra Pr \frac{\partial \theta^*}{\partial y^*} \quad (4.23)$$

where P_k^* and G_k^* correspond to the nondimensional shear and buoyancy production rates of the turbulent kinetic energy, respectively [57].

As described in section 3.4.2.4, the boundary conditions for the turbulence model are specified at the grid point next to the wall. In their dimensionless form, the Dirichlet boundary conditions are

$$k_p^* = \frac{(u_*^*)^2}{C_\mu^{1/2}} \quad (4.24)$$

$$\epsilon_p^* = \frac{(u_*^*)^3}{\kappa y_p^*} \quad (4.25)$$

$$u_*^* = \frac{u^*}{U}, \quad y_p^* = \frac{y_p}{W} \quad (4.26)$$

where u_*^* represents the non-dimensional friction velocity and y_p^* is the non-dimensional normal distance between the innermost grid point and its closest boundary.

As for the initial conditions, careful selection is necessary. During the numerical study of chapter 5, it was observed that in the case where the fluid starts from rest conditions, the initial values of k and ϵ can be selected as uniform over the domain, as long as their initial distribution is consistent with the selected distribution of the dimensionless eddy-viscosity. The general concept is that at the rest conditions, the state of turbulent viscosity is small with respect to the actual viscosity. Initial zero values for k and ϵ are not valid, since their ratio would be undefined in the ϵ equation. As an alternative, the value of ν_t^* was selected as $Pr/100$ and distributed uniformly over the domain. Using the Prandtl-Kolmogorov relation relating the eddy-viscosity to the values of k and ϵ , the value of ϵ can be calculated at every point of the grid given the eddy-viscosity and the value of k . Care must be taken to choose k and ϵ in such a way that their ratio, present in the ϵ equation is not too large to begin with, in order to not kick the solution of the ϵ equation too hard to begin with. The issue of

initial conditions for these equations is not alluded to in the literature, since almost all studies involve the solution of the steady versions of the governing equations presented here. In the steady formulation the problem is no longer an initial boundary value problem, but a boundary value problem, so only boundary conditions are necessary. An extraordinary effort was made to find references addressing the topic of initial conditions for the k - ϵ turbulence model, but no significant contributions in this area were found. In this study, it has been observed that poor selection of the initial conditions can lead to rapidly diverging solutions of the equations. Once a solution is obtained for a case which started from rest, the value of k and ϵ which do satisfy the governing equations, are used as initial conditions for any other simulations.

The major assumptions made in this formulation of the governing equations of fluid flow are as follows. First, the Boussinesq approximation is in effect, which states that the temperature induced density variation in the natural convective flow is negligible in the governing equations except in the buoyant body force term which elicits the buoyant fluid motion. Evoking this assumption implies that the temperature dependence of the density can be written as $\Delta\rho = -\beta\rho\Delta\theta$ where ρ is the density, β is the coefficient of thermal expansion and θ is the temperature. Furthermore, the enthalpy of the fluid is assumed to be a function of temperature only, such that the energy equation reduces to an equation of temperature variation [58]. Finally, the physical properties of the flow are constant [59]. By choosing a reference temperature $\theta_r = 293K$ and a temperature difference $(\theta_h - \theta_c) = 20K$, and overheat ratio $\delta = \frac{(\theta_h - \theta_c)}{\theta_r} = 0.068$ is obtained, which is small enough to guarantee the validity of the Boussinesq approximation applied to the governing equations. At this reference temperature, and using air as the working fluid, $Pr = 0.71$. By holding the overheat

ratio constant, and increasing the cavity's characteristic length, increasing values of Ra are obtained.

The next few sections concentrate on how these equations are to be solved using the finite element method and a semi-implicit time-stepping technique, combined with a projection scheme. For the sake of clarity, the * notation indicating the variables are nondimensional will be dropped in the subsequent discussion.

4.2 Numerical procedure

4.2.1 Time discretization

The temporal variables in the governing equations are discretized using a semi-implicit time-splitting scheme. The advection terms are advanced using the explicit Adams-Bashforth methods, while the viscous terms are advanced with the help of the implicit Euler scheme. The implicit aspect of the method enables one to use large time steps due to better stability, while the explicit aspect of the method reduces the required storage due to the use of condensed matrices [60][61][62][63].

On the other hand, the fractional-step method is used to solve the governing equations. The method, as applied to equations 4.16-4.21 involves four (4) major steps.

4.2.1.1 Step 1 - Intermediate velocity calculations

In this step, the pressure terms in the momentum equations are dropped. Their solution then is solely based on the advection and diffusion terms. The resulting velocities do satisfy the boundary conditions, but are not divergence-free. These intermediate velocities will be labeled $\tilde{u}$, $\tilde{v}$ and $\tilde{w}$. The resulting formulation in index

notation is:

$$\tilde{u}_i^{n+1} - \Delta t \left[(Pr + \nu_t) (\tilde{u}_{i,j}^{n+1} + \tilde{u}_{j,i}^{n+1}) \right]_{,j} = u_i^n - \Delta t \left[\frac{3}{2} u_j^n u_{i,j}^n - \frac{1}{2} u_j^{n-1} u_{i,j}^{n-1} \right] + S \quad (4.27)$$

where

$$S = \delta_{i2} \Delta t Ra Pr \theta \quad (4.28)$$

4.2.1.2 Step 2 - Velocity corrections

The resulting intermediate velocity field is corrected by the dynamic effect of the still unknown pressure p by including the pressure gradient that was eliminated in step 1. By the Helmholtz decomposition theorem, the intermediate velocity field can be written as a sum of a divergence-free field and a zero-curl field [59]. The divergence-free field is the actual velocity one is interested in, and the irrotational field is related to the gradient of the pressure. The resulting formulation in index notation is:

$$u_i^{n+1} = \tilde{u}_i^{n+1} - \Delta t p_{,i}^{n+1} \quad (4.29)$$

4.2.1.3 Step 3 - Pressure calculations

By taking the divergence of the velocity correction equations and adding them, a Poisson equation can be obtained for the unknown pressure field p . The resulting formulation in index notation is:

$$p_{,ii}^{n+1} = \frac{1}{\Delta t} u_{i,i}^{n+1} \quad (4.30)$$

Solving equation 4.30 yields the pressure field necessary to correct the velocities in step 2. To solve the equation, appropriate boundary conditions must be specified. For example, a Neumann condition at the boundaries can be obtained which has the form:

$$p_{,n}^{n+1} = f \text{ where } f = 0 \text{ at no-slip boundaries} \quad (4.31)$$

4.2.1.4 Step 4 - Temperature, k and ϵ calculations

Given the divergence-free velocities, the energy equation, together with the k and ϵ equations can be solved. The resulting formulation in index notation is:

$$\theta^{n+1} - \Delta t \left[\left(1 + \frac{\nu_t}{\sigma_t} \right) \theta_j^{n+1} \right]_j = \theta^n - \Delta t \left[\frac{3}{2} u_j^n \theta_j^n - \frac{1}{2} u_j^{n-1} \theta_j^{n-1} \right] \quad (4.32)$$

$$k^{n+1} - \Delta t \left[\left(Pr + \frac{\nu_t}{\sigma_k} \right) k_j^{n+1} \right]_j + \Delta t \left(\frac{\epsilon}{k} \right)^n k^{n+1} = \Delta t P_k^{n+1} + \Delta t G_k^{n+1} + k^n - \Delta t \left[\frac{3}{2} u_j^n \theta_j^n - \frac{1}{2} u_j^{n-1} \theta_j^{n-1} \right] \quad (4.33)$$

$$\epsilon^{n+1} - \Delta t \left[\left(Pr + \frac{\nu_t}{\sigma_\epsilon} \right) \epsilon_j^{n+1} \right]_j + \Delta t \left(\frac{\epsilon}{k} \right)^n C_{2\epsilon} \epsilon^{n+1} = \Delta t \left(\frac{\epsilon}{k} \right)^n C_{1\epsilon} P_k^{n+1} + \Delta t \left(\frac{\epsilon}{k} \right)^n C_{1\epsilon} C_{3\epsilon} G_k^{n+1} + \epsilon^n - \Delta t \left[\frac{3}{2} u_j^n \epsilon_j^n - \frac{1}{2} u_j^{n-1} \epsilon_j^{n-1} \right] \quad (4.34)$$

4.2.2 Spatial discretization

The spatial variables are discretized by splitting the domain Ω whose boundary is Γ into non overlapping, finite elements. To apply the finite element method, an approximate solution to the governing equation is assumed. If the differential operator describing any of the governing equations is $\mathcal{L}_i$, and the corresponding solution is x , then the governing equation can be written as $\mathcal{L}_i(x) = 0$. Applying the operator to the approximate solution $\bar{x}$, yields $\mathcal{L}_i(\bar{x}) \neq 0$. The difference between both expressions is called the residual $R = \mathcal{L}_i(\bar{x}) - \mathcal{L}_i(x)$.

Using the Galerkin method of weighted residuals, the objective is to make the residual as close to zero as possible in the mean. This implies that the approximate solution approaches the exact solution in some norm. This is accomplished by first multiplying the residual by a weighting function and then integrating the product over

the domain Ω . This resulting integral form is what is called the **weak form**. As indicated in section 1.5, the finite element methods establishes a process by which these global weighted residual integrals are transformed into local weighted residual integrals over each of the finite elements spanning Ω , call them Ω_e . Within each finite element Ω_e , the variables of the governing equations are assumed to behave in a prescribed manner. In the Galerkin finite element method, the weighting functions are assumed to behave in the same manner as the problem variables. Then, the local integrals are evaluated and the local results are put together via an assembly process which generates matrix equations which approximate the governing equations, and if solved approximate the exact solution over the domain [64][65][66].

The functional behavior of the variable x within an element can be written as a linear combination of n interpolation functions N_i , appropriately scaled by n scalar values x_i of the variable at key points within the element, i.e. $x = x_i N_i$ in index notation, or $x = \mathbf{N}\mathbf{x} = (\mathbf{x}^T \mathbf{N}^T)^T$ in matrix notation, where $\mathbf{N} = [N_1, N_2, N_3, \dots, N_n]$ and $\mathbf{x} = [x, x_2, x_3, \dots, x_n]^T$.

If the approximate solutions for the variables in the time-discretized governing equations 4.27-4.34 are written as:

$$\tilde{u} = \mathbf{N}\tilde{\mathbf{u}} , \tilde{v} = \mathbf{N}\tilde{\mathbf{v}} , \tilde{w} = \mathbf{N}\tilde{\mathbf{w}} , u = \mathbf{N}\mathbf{u} , v = \mathbf{N}\mathbf{v} , w = \mathbf{N}\mathbf{w} \quad (4.35)$$

$$p = \mathbf{N}\mathbf{p} , \theta = \mathbf{N}\mathbf{r} , k = \mathbf{N}\mathbf{k} , \epsilon = \mathbf{N}\mathbf{e} , x = \mathbf{N}\mathbf{x} , y = \mathbf{N}\mathbf{y} , z = \mathbf{N}\mathbf{z} \quad (4.36)$$

then, after the application of the divergence theorem on the resulting weak forms and after assembly into global form, the following system of nonlinear matrix equations is obtained:

$$\left[\frac{1}{\Delta t} \mathbf{M} + \mathbf{K}_m \right] \tilde{\mathbf{u}}^{n+1} = \frac{1}{\Delta t} \mathbf{M}\mathbf{u}^n - \left[\frac{3}{2} \mathbf{P}^n \mathbf{u}^n - \frac{1}{2} \mathbf{P}^{n-1} \mathbf{u}^{n-1} \right] \quad (4.37)$$

$$\left[\frac{1}{\Delta t} \mathbf{M} + \mathbf{K}_m \right] \tilde{\mathbf{v}}^{n+1} = \frac{1}{\Delta t} \mathbf{M} \mathbf{v}^n - \left[\frac{3}{2} \mathbf{P}^n \mathbf{v}^n - \frac{1}{2} \mathbf{P}^{n-1} \mathbf{v}^{n-1} \right] + \mathbf{S}_1 \quad (4.38)$$

$$\mathbf{S}_1 = RaPr\mathbf{M}\mathbf{r}^n \quad (4.39)$$

$$\left[\frac{1}{\Delta t} \mathbf{M} + \mathbf{K}_m \right] \tilde{\mathbf{w}}^{n+1} = \frac{1}{\Delta t} \mathbf{M} \mathbf{w}^n - \left[\frac{3}{2} \mathbf{P}^n \mathbf{w}^n - \frac{1}{2} \mathbf{P}^{n-1} \mathbf{w}^{n-1} \right] \quad (4.40)$$

$$\mathbf{K} \mathbf{p}^{n+1} = -\frac{1}{\Delta t} \left(\mathbf{D} \cdot \mathbf{D} \mathbf{X} \tilde{\mathbf{u}}^{n+1} + \mathbf{D} \cdot \mathbf{D} \mathbf{Y} \tilde{\mathbf{v}}^{n+1} + \mathbf{D} \cdot \mathbf{D} \mathbf{Z} \tilde{\mathbf{w}}^{n+1} \right) \quad (4.41)$$

$$\mathbf{M}_C \mathbf{u}^{n+1} = \mathbf{M}_C \tilde{\mathbf{u}}^{n+1} - \Delta t \mathbf{D} \cdot \mathbf{D} \mathbf{X} \mathbf{p}^{n+1} \quad (4.42)$$

$$\mathbf{M}_C \mathbf{v}^{n+1} = \mathbf{M}_C \tilde{\mathbf{v}}^{n+1} - \Delta t \mathbf{D} \cdot \mathbf{D} \mathbf{Y} \mathbf{p}^{n+1} \quad (4.43)$$

$$\mathbf{M}_C \mathbf{w}^{n+1} = \mathbf{M}_C \tilde{\mathbf{w}}^{n+1} - \Delta t \mathbf{D} \cdot \mathbf{D} \mathbf{Z} \mathbf{p}^{n+1} \quad (4.44)$$

$$\left[\frac{1}{\Delta t} \mathbf{M} + \mathbf{K}_\theta \right] \mathbf{r}^{n+1} = \frac{1}{\Delta t} \mathbf{M} \mathbf{r}^n - \left[\frac{3}{2} \mathbf{P}^n \mathbf{r}^n - \frac{1}{2} \mathbf{P}^{n-1} \mathbf{r}^{n-1} \right] \quad (4.45)$$

$$\left[\frac{1}{\Delta t} \mathbf{M} + \mathbf{K}_k + \mathbf{E} \mathbf{K} \right] \mathbf{k}^{n+1} = \frac{1}{\Delta t} \mathbf{M} \mathbf{k}^n - \left[\frac{3}{2} \mathbf{P}^n \mathbf{k}^n - \frac{1}{2} \mathbf{P}^{n-1} \mathbf{k}^{n-1} \right] + \mathbf{S}_2 \quad (4.46)$$

$$\mathbf{S}_2 = \mathbf{P} \mathbf{K} \mathbf{U}_k^{n+1} \mathbf{u}^{n+1} + \mathbf{P} \mathbf{K} \mathbf{V}_k^{n+1} \mathbf{v}^{n+1} + \mathbf{P} \mathbf{K} \mathbf{W}_k^{n+1} \mathbf{w}^{n+1} - \frac{RaPr}{\sigma_t} \mathbf{G}_k \mathbf{r}^{n+1} \quad (4.47)$$

$$\left[\frac{1}{\Delta t} \mathbf{M} + \mathbf{K}_\epsilon + C_{2\epsilon} \mathbf{E} \mathbf{K} \right] \epsilon^{n+1} = \frac{1}{\Delta t} \mathbf{M} \epsilon^n - \left[\frac{3}{2} \mathbf{P}^n \epsilon^n - \frac{1}{2} \mathbf{P}^{n-1} \epsilon^{n-1} \right] + C_{1\epsilon} \mathbf{S}_3 \quad (4.48)$$

$$\mathbf{S}_3 = \mathbf{P} \mathbf{K} \mathbf{U}_\epsilon^{n+1} \mathbf{u}^{n+1} + \mathbf{P} \mathbf{K} \mathbf{V}_\epsilon^{n+1} \mathbf{v}^{n+1} + \mathbf{P} \mathbf{K} \mathbf{W}_\epsilon^{n+1} \mathbf{w}^{n+1} - \frac{RaPr}{\sigma_\epsilon} \mathbf{G}_\epsilon \mathbf{r}^{n+1} \quad (4.49)$$

The coefficient matrices in the above finite element equations are defined as:

$$\mathbf{M} = \iiint_{stkr} \mathbf{N}^T \mathbf{N} \det \mathbf{J} \, ds dt dr \quad (4.50)$$

$$\mathbf{K} = \iiint_{stkr} \mathbf{B}_{str}^T \left(\mathbf{J}^{-1} \right)^T \left(\mathbf{J}^{-1} \right) \mathbf{B}_{str} \det \mathbf{J} \, ds dt dr \quad (4.51)$$

$$\mathbf{K}_m = \iiint_{s|t|k} (Pr + \nu_t) \mathbf{B}_{str}^T (\mathbf{J}^{-1})^T (\mathbf{J}^{-1}) \mathbf{B}_{str} \det \mathbf{J} \, ds dt dr \quad (4.52)$$

$$\mathbf{K}_\theta = \iiint_{s|t|k} \left(1 + \frac{\nu_t}{\sigma_t}\right) \mathbf{B}_{str}^T (\mathbf{J}^{-1})^T (\mathbf{J}^{-1}) \mathbf{B}_{str} \det \mathbf{J} \, ds dt dr \quad (4.53)$$

$$\mathbf{K}_k = \iiint_{s|t|k} \left(Pr + \frac{\nu_t}{\sigma_k}\right) \mathbf{B}_{str}^T (\mathbf{J}^{-1})^T (\mathbf{J}^{-1}) \mathbf{B}_{str} \det \mathbf{J} \, ds dt dr \quad (4.54)$$

$$\mathbf{K}_\epsilon = \iiint_{s|t|k} \left(Pr + \frac{\nu_t}{\sigma_\epsilon}\right) \mathbf{B}_{str}^T (\mathbf{J}^{-1})^T (\mathbf{J}^{-1}) \mathbf{B}_{str} \det \mathbf{J} \, ds dt dr \quad (4.55)$$

$$\mathbf{P} = \iiint_{s|t|k} \mathbf{N}^T \mathbf{N} [\mathbf{u} \, \mathbf{v} \, \mathbf{w}] (\mathbf{J}^{-1}) \mathbf{B}_{str} \det \mathbf{J} \, ds dt dr \quad (4.56)$$

$$\mathbf{D_DX} = \iiint_{s|t|k} \mathbf{N}^T \, 1^{st} \text{row} (\mathbf{J}^{-1} \mathbf{B}_{str}) \det \mathbf{J} \, ds dt dr \quad (4.57)$$

$$\mathbf{D_DY} = \iiint_{s|t|k} \mathbf{N}^T \, 2^{nd} \text{row} (\mathbf{J}^{-1} \mathbf{B}_{str}) \det \mathbf{J} \, ds dt dr \quad (4.58)$$

$$\mathbf{D_DZ} = \iiint_{s|t|k} \mathbf{N}^T \, 3^{rd} \text{row} (\mathbf{J}^{-1} \mathbf{B}_{str}) \det \mathbf{J} \, ds dt dr \quad (4.59)$$

$$\mathbf{EK} = \iiint_{s|t|k} \frac{\epsilon}{k} \mathbf{N}^T \mathbf{N} \det \mathbf{J} \, ds dt dr \quad (4.60)$$

$$\mathbf{G}_k = \iiint_{s|t|k} \nu_t \mathbf{N}^T \, 2^{nd} \text{row} (\mathbf{J}^{-1} \mathbf{B}_{str}) \det \mathbf{J} \, ds dt dr \quad (4.61)$$

$$\mathbf{G}_\epsilon = \iiint_{s|t|k} \frac{\epsilon}{k} \nu_t C_{3\epsilon} \mathbf{N}^T \, 2^{nd} \text{row} (\mathbf{J}^{-1} \mathbf{B}_{str}) \det \mathbf{J} \, ds dt dr \quad (4.62)$$

$$\mathbf{PKU}_k = \iiint_{s|t|k} \nu_t \left[2 \mathbf{N}^T \, 1^{st} \text{row} (\mathbf{J}^{-1} \mathbf{B}_{str}) \mathbf{u} \, 1^{st} \text{row} (\mathbf{J}^{-1} \mathbf{B}_{str}) \right] \det \mathbf{J} \, ds dt dr \quad (4.63)$$

$$+ \iiint_{s|t|k} \nu_t \left[\mathbf{N}^T \, 2^{nd} \text{row} (\mathbf{J}^{-1} \mathbf{B}_{str}) \mathbf{u} \, 2^{nd} \text{row} (\mathbf{J}^{-1} \mathbf{B}_{str}) \right] \det \mathbf{J} \, ds dt dr$$

$$+ \iiint_{s|t|k} \nu_t \left[\mathbf{N}^T \, 3^{rd} \text{row} (\mathbf{J}^{-1} \mathbf{B}_{str}) \mathbf{u} \, 3^{rd} \text{row} (\mathbf{J}^{-1} \mathbf{B}_{str}) \right] \det \mathbf{J} \, ds dt dr$$

$$+ \iiint_{s|t|k} \nu_t \left[2 \mathbf{N}^T \, 1^{st} \text{row} (\mathbf{J}^{-1} \mathbf{B}_{str}) \mathbf{w} \, 3^{rd} \text{row} (\mathbf{J}^{-1} \mathbf{B}_{str}) \right] \det \mathbf{J} \, ds dt dr$$

$$\begin{aligned}
& + \iiint \frac{\epsilon}{k} \nu_t \left[2 \mathbf{N}^T \text{2}^{nd}\text{row} \left(\mathbf{J}^{-1} \mathbf{B}_{str} \right) \mathbf{u} \text{1}^{st}\text{row} \left(\mathbf{J}^{-1} \mathbf{B}_{str} \right) \right] \det \mathbf{J} \, ds dt dr \\
\mathbf{PKW}_\epsilon = & \iiint \frac{\epsilon}{k} \nu_t \left[\mathbf{N}^T \text{1}^{st}\text{row} \left(\mathbf{J}^{-1} \mathbf{B}_{str} \right) \mathbf{w} \text{1}^{st}\text{row} \left(\mathbf{J}^{-1} \mathbf{B}_{str} \right) \right] \det \mathbf{J} \, ds dt dr \quad (4.68) \\
& + \iiint \frac{\epsilon}{k} \nu_t \left[\mathbf{N}^T \text{2}^{nd}\text{row} \left(\mathbf{J}^{-1} \mathbf{B}_{str} \right) \mathbf{w} \text{2}^{nd}\text{row} \left(\mathbf{J}^{-1} \mathbf{B}_{str} \right) \right] \det \mathbf{J} \, ds dt dr \\
& + \iiint \frac{\epsilon}{k} \nu_t \left[2 \mathbf{N}^T \text{3}^{rd}\text{row} \left(\mathbf{J}^{-1} \mathbf{B}_{str} \right) \mathbf{w} \text{3}^{rd}\text{row} \left(\mathbf{J}^{-1} \mathbf{B}_{str} \right) \right] \det \mathbf{J} \, ds dt dr \\
& + \iiint \frac{\epsilon}{k} \nu_t \left[2 \mathbf{N}^T \text{3}^{rd}\text{row} \left(\mathbf{J}^{-1} \mathbf{B}_{str} \right) \mathbf{v} \text{2}^{nd}\text{row} \left(\mathbf{J}^{-1} \mathbf{B}_{str} \right) \right] \det \mathbf{J} \, ds dt dr
\end{aligned}$$

4.3 Solver implementation using the OOFEA classes and methods

Appendix A contains the listing of the turbulent natural convection simulator designed with the help of the OOFEA toolkit to implement the weak forms as well as to solve the finite element equations presented in the previous section. The equations and weak forms in the previous section represent the most general form of the equations: their three-dimensional form. Their two-dimensional representation simply involves eliminating the transverse velocity components (i.e. the w components) in any of the equations and weak forms, as well as rewriting the integrals as double instead of triple integrals.

The structure of the code presented in appendix A can be described in the following manner:

- User definition of the necessary weak forms represented by equations 4.50-4.68.
- User input processing - involves obtaining data from the user concerning parameters of the simulation, filenames for initial and boundary conditions, filenames

for mesh information such as nodal coordinates and element connectivities, and other user options.

- Declaration of `mesh` object
- Initialization of the `mesh` object using its methods to read the data from the user provided filenames. This creates all necessary `node` and `element` objects for the simulation.
- Simulation (time) loop:
 - intermediate velocity calculations,
 - pressure calculations,
 - velocity correction calculations,
 - temperature calculations,
 - calculation of boundary conditions for k ,
 - calculation of boundary conditions for ϵ ,
 - solution of k equation,
 - solution of ϵ equation,
 - calculation of turbulent viscosity ν_t ,
 - permanent storage of results and post-processing if necessary,
 - convergence check,
- Final post-processing and permanent storage of solutions.

At this point, it is important to stress the advantages of the OOFEA toolkit implementation of the solver in comparison to other implementations:

- *Fast prototyping of the necessary weak forms* - all weak forms involved are declared in the same manner, all utilizing the method interface of their `element` arguments.
- *Ready-to-use gather and scatter operations* - distribution of global solutions to the elements (*scatter*) and collection of local element solutions into global solutions (*gather*) operations are directly available via the `mesh` interface.
- *Ready-to-use assembly operations* - automatic evaluation and assembly of user-defined weak forms into `sparse_matrix` objects which can be used directly in mathematical operations.
- *Linear algebra support* - all resulting mathematical entities are linear algebra objects, for which all important mathematical operators have been redefined for clarity and efficiency. This provides "what-you-see-is-what-you-get" (WYSIWYG) mathematical equations.
- *Standardized access to problem data structures* - via the provided and user-derivable iterator classes, access to the problem's data structures is standardized. This is particularly important in order to accomplish tasks such as: boundary condition changes (for time-varying boundary conditions), nodal coordinate changes (for *r*-adaption codes), connectivity changes (for *h*-adaption codes).
- *Completely dynamic memory allocation* - all memory is allocated dynamically and automatically deallocated for better memory management.
- *Unlimited object capability* - with the provided system of classes and methods, multiple domains and/or problems can be worked on simultaneously (for domain-decomposition methods).

After describing all the necessary components for a simulator of turbulent natural convective flows in enclosures, the next two (2) chapters present a numerical study performed with the implemented simulator. It should be clear by this point that tools such as this can be used to reduce development time of complex CFD and other FE applications; can be used as a platform for group implementation of both old and new algorithms, and solution techniques; and can be extended and transformed via inheritance in ways that increase generality and breath as their complexity increases.

Chapter 5

Laminar and Turbulent Natural Convection in Square Cavities

5.1 Thermally driven cavity

The domain in question is a square vertical cavity whose left and right vertical walls are isothermal at two slightly different temperatures. The left wall is kept at a higher temperature than the right wall, so there is a thermal gradient between them. The remaining parallel horizontal walls are adiabatic and enclose air with a Prandtl number of 0.71 within the cavity. The aspect ratio of the cavity is $A = 1$, and by appropriately increasing the characteristic length W of the domain, a variety of Rayleigh numbers can be achieved. In all cases, the nondimensional form of the equations yields an unit height and length cavity. A no-slip boundary condition is valid at all the enclosure boundaries for both the intermediate and divergence-free velocities of the resulting formulation, the temperature is specified at the isothermal walls, and the normal gradient of the temperature is zero at the adiabatic walls, while the normal gradient of the pressure at all system walls is zero.

For both laminar and turbulent results, a strongly non-uniform mesh is used to correctly resolve the large velocity and temperature gradients that emerge as the Rayleigh number increases, in particular for the turbulent cases. If there is enough resolution in the mesh, the velocity profile within the boundary layer should be smooth. If it was not, discrepancies in the boundary layer would affect the fluid core, which in turn would affect the boundary layer, leading to possibly different numerical solutions for

the flows involved. A power-law clustering technique is used to generate the meshes for this numerical study [44]. Given an uniform distribution of grid points within a rectangular domain of width W and height H in the mutually orthogonal directions ξ and η , the corresponding distributions of x and y are given by the transformation equations

$$x = W \frac{2\alpha - \beta + (2\alpha + \beta) \left[\frac{\beta+1}{\beta-1} \right]^{\frac{\xi-\alpha}{1-\alpha}}}{(2\alpha + 1) \left\{ 1 + \left[\frac{\beta+1}{\beta-1} \right]^{\frac{\xi-\alpha}{1-\alpha}} \right\}} \quad (5.1)$$

$$y = H \frac{2\alpha - \beta + (2\alpha + \beta) \left[\frac{\beta+1}{\beta-1} \right]^{\frac{\eta-\alpha}{1-\alpha}}}{(2\alpha + 1) \left\{ 1 + \left[\frac{\beta+1}{\beta-1} \right]^{\frac{\eta-\alpha}{1-\alpha}} \right\}} \quad (5.2)$$

where β is the clustering parameter which can range between 1 and ∞ , and α controls where the clustering takes place. For $\alpha = 0$, the clustering occurs at $x = 0$ and $y = 0$, while for $\alpha = 0.5$, the clustering is distributed equally at $x = 0$ and $x = W$, and $y = 0$ and $y = H$. As β approaches 1, the clustering becomes more and more pronounced near the boundaries. Values of β for this study range between 1.05 for most laminar results, and 1.01 for most turbulent results.

Laminar results were obtained for low Rayleigh numbers using rest conditions as initial conditions, that is, all velocities are zero to begin with, as well as the temperature. Once a result was obtained for a particular Rayleigh number, these solutions were used as initial conditions for subsequent Rayleigh numbers.

Turbulent results were also obtained in a similar manner. For a particular Rayleigh number, the velocities and temperature are zero to begin with. However, as stated in chapter 4, the definition of boundary and initial conditions for k and ϵ and their non-dimensional counterparts is not straightforward. Great care was taken to define the initial conditions such as to be consistent with the values of ν_t^* , k^* and ϵ^* , to

prevent the ratio ϵ^*/k^* to be too large during the initial steps of the simulation, and to be consistent with the physical conditions of the problem. As will be seen later, consistency tests, which involve solving known laminar states using the k - ϵ turbulence model to determine if they also predict laminar solutions, have been used to corroborate the implementation of the turbulence model. For the case where $Ra = 10^6$, a known laminar state, a solution with the k - ϵ turbulence model was sought. In this case, initial rest conditions were used for the velocities and temperatures. Knowing the solution should be laminar, ν_t^* , k^* and ϵ^* should be zero. However, this would introduce undefined terms in the solution. Hence, an initial distribution of ν_t^* was chosen to be uniform over the domain with a value of $Pr/100$, which is small in comparison with the actual viscous effects. Solutions were obtained that yielded actual distributions of k^* and ϵ^* , which produced vanishing values of the turbulent eddy-viscosity, in other words, an essentially laminar solution was obtained with the k - ϵ model. The resulting distributions of k^* and ϵ^* for this case were then used as initial conditions for all other turbulent numerical experiments.

5.2 Laminar simulations for Ra between 10^3 and 10^6

The results in this section were obtained in strongly non-uniform meshes, using the previously described power-law clustering technique. Grids with 51×51 grid points used, and the clustering parameter β had the value 1.05. All solutions were obtained by solving the time dependent governing equations described in chapter 4. The resulting simulations are then time accurate. Time histories could have been generated, but the amount of data necessary to store an adequate time history for display is too large for meaningful discussions. The simulations performed converged to a steady-state solution, and the following discussion describes those steady results.

Figure 5.1 contains the simulation results for $Ra = 10^3$. At this Rayleigh number, the streamline pattern is that of a single circulation cell, whose center coincides with that of the cavity. The isotherm pattern is essentially parallel to the isothermal vertical walls, except for a slight inclination in the isotherms within the core, indicating that the dominant heat transfer process is that of conduction. Any convective effects are the cause for the departures from perfectly vertical isotherms. The circulation cell in the cavity is generated by the momentum transfer from the temperature field to the velocity field. The negative temperature gradient across the cavity gives rise to a clockwise rotating cell.

As the Rayleigh number is increased to 10^4 (see figure 5.2), the streamline pattern begins to elongate in the horizontal direction into an ellipse, while the isotherm pattern indicates a definite change from conduction to convection as the main heat transfer process. The temperature gradient is now more pronounced near the walls, particularly near the bottom of the left and the top of the right walls. However, the temperature gradient in the horizontal direction is almost zero (horizontal isotherms) close to center while the vertical temperature gradient seems to be linear in this region.

At $Ra = 10^5$ (see figure 5.3), the streamline pattern has changed from a one circulation cell to a two circulation cell pattern. The original cell elongated further in the horizontal direction until it snapped into two new cells. These cell are both rotating clockwise like the original cell at lower Rayleigh numbers. However, the two new cells are still surrounded by closed streamlines, so it is only in the core where these motions occur. Near the walls the flow is still circulating as one cell, around the whole cavity. Hence, the cells near the center are secondary motions of the fluid. These secondary circulations are the direct result of the increase in convective effects

Figure 5.1: From top left to center bottom: Streamlines, vorticity contours, pressure contours, isotherm pattern and velocity vector plot for the laminar case of $Ra = 10^3$.

Figure 5.2: From top left to center bottom: Streamlines, vorticity contours, pressure contours, isotherm pattern and velocity vector plot for the laminar case of $Ra = 10^4$.

within the cavity. As can be seen in the isotherm patterns, in the region next to the wall, the horizontal temperature gradient has increased from the previous Rayleigh number. The dominating heat transfer process is obviously convection. Furthermore, between the core and the wall, regions exist where the horizontal temperature gradient has changed sign from negative to positive. These regions coincide with the regions where the secondary circulation cells are observed. Furthermore, the region of the center of the cavity where the temperature horizontal temperature gradient is zero, and the vertical temperature gradient is linear has become more extensive. This linear stratification of the core is impeding any vertical motion, so the core is also becoming stagnant.

As the Rayleigh number is increased further to 10^6 (see figure 5.4), the streamline pattern reveals that the secondary circulation cells observed at 10^5 have moved further in the horizontal direction and have elongated in the vertical direction as they are affected by the boundary layer flow near the isothermal walls. By observing the isotherm pattern, it is clear that they are directly associated with local region where the temperature gradient changes sign. Although almost not discernible, a small positive temperature gradient has now emerged in the center of the cavity. According to the previous statement, such a change in sign of the temperature gradient is the cause of circulating motions. Thus a circulation cell should be expected also in this region. The streamline pattern reveals exactly that, a weak third clockwise rotating circulation cell can be observed in the center of the cavity. Although it might seem that the circulation should be counterclockwise to reduce the shear between the secondary vortices [51], it seems that viscous dissipation takes care of that and the temperature gradient in the center yields this motion.

Figure 5.3: From top left to center bottom: Streamlines, vorticity contours, pressure contours, isotherm pattern and velocity vector plot for the laminar case of $Ra = 10^5$.

Figure 5.4: From top left to center bottom: Streamlines, vorticity contours, pressure contours, isotherm pattern and velocity vector plot for the laminar case of $Ra = 10^6$.

Figures 5.5 to 5.8 represent the velocity profiles of both the dimensionless vertical and horizontal velocities at the centerlines of the cavity. As the Rayleigh number increases, the vertical velocity becomes negligible in the center of the cavity, so that vertical motions are inhibited and the core is essentially stagnant, as previously stated.

Figure 5.5: From left to right: Velocity profiles for the dimensionless vertical and horizontal velocities of the flow at the horizontal and vertical centerlines of the cavity ($y^* = 0.5$ and $x^* = 0.5$) when $Ra = 10^3$.

Table 5.1 is a summary of the results obtained using the laminar portion of the natural convection simulator. The results are compared with the bench mark solutions reported by de Vahl Davis [30]. Good agreement is found for all field variables and locations. The results reported by de Vahl Davis have been extrapolated using Richardson's extrapolation to obtain their desired accuracy. Furthermore, the bench mark results are given at the extrapolated locations, not at their closest grid locations, using a fourth-order polynomial approximation. Accordingly, the bench mark values are supposed to be within 1% and 6% of the actual grid values. The results of this study are reported at their grid locations. As can be seen, good agreement is obtained. The difference in the Nusselt number however can reach up to 10% at

Figure 5.6: From left to right: Velocity profiles for the dimensionless vertical and horizontal velocities of the flow at the horizontal and vertical centerlines of the cavity ($y^* = 0.5$ and $x^* = 0.5$) when $Ra = 10^4$.

Figure 5.7: From left to right: Velocity profiles for the dimensionless vertical and horizontal velocities of the flow at the horizontal and vertical centerlines of the cavity ($y^* = 0.5$ and $x^* = 0.5$) when $Ra = 10^5$.

Figure 5.8: From left to right: Velocity profiles for the dimensionless vertical and horizontal velocities of the flow at the horizontal and vertical centerlines of the cavity ($y^* = 0.5$ and $x^* = 0.5$) when $Ra = 10^6$.

the highest Rayleigh number. We expect better agreement if extrapolation is used, but as has been pointed out by Markatos and Pericleous [51], the calculation of the Nusselt number is very sensitive to the form of its calculation. Since we are using an irregular mesh instead of a regular one, the differences observed can be attributed to the different formulations used to estimate the Nusselt number at the wall. In fact, better agreement of the Nusselt number is obtained when compared to the results of Markatos and Pericleous. The results of Markatos and Pericleous are so similar to those of de Vahl Davis, that they are not presented here. However, their large Rayleigh number results will be used later on for the discussion of the turbulent simulations in this study.

It is necessary to point out that the results by de Vahl Davis and by Markatos and Pericleous, and of many others as well, are the result of solving the steady-formulations of the governing equations. Hence, the results they obtain are only steady results. Furthermore, they cannot generate time-accurate histories with these

formulations. As was mentioned in chapter 3, time evolution of these processes is of practical importance and scientific interest. Furthermore, as will be seen later, the use of steady formulations cannot predict steady oscillations in the core of a turbulent flow as time dependent schemes can.

5.3 Consistency tests at $Ra = 10^6, 10^7$

A *consistency test* is defined here as a numerical experiment in which the validity of a computer model and its correct implementation can be assessed. For the case of turbulent natural convection in a differentially heated square enclosure, it is suggested here that such a consistency test exists.

If the k - ϵ turbulence model is appropriate for these types of flows at high Rayleigh numbers, and furthermore, if it has been correctly implemented computationally, then numerical experiments should yield appropriate physically meaningful results. It is suggested here that numerical experiments can be performed with the turbulence model, for which no turbulence should be observed, and in the limit of the simulation, close to laminar results should be obtained.

In this study, laminar solutions for $Ra = 10^6$ and $Ra = 10^7$ were obtained numerically using a laminar solver. Then, solutions were sought using the k - ϵ model at this same Rayleigh numbers. Strongly non-uniform meshes with 33×33 grid points were used to resolve the boundary layers. It is here suggested that a correct implementation and adequate mesh resolution should force the turbulent simulation to yield close to laminar results at these Rayleigh numbers, which are known to be laminar. This should be reflected in a non-zero but extremely small turbulent eddy-viscosity, and mean velocity and mean temperature distributions similar to those obtained in the laminar case.

Unknown	Ra							
	10^3		10^4		10^5		10^6	
	PRE	DVD	PRE	DVD	PRE	DVD	PRE	DVD
$ \Psi_{mid} $	1.1910	1.174	5.1029	5.071	9.1651	9.111	16.4712	16.32
$ \Psi_{max} $	1.1910	1.174	5.1029	5.071	9.6755	9.612	16.9129	16.750
x	0.5000	-	0.5000	-	0.2724	0.285	0.8535	0.151
y	0.5000	-	0.5000	-	0.6197	0.601	0.4611	0.547
u_{max}	3.6473	3.649	16.2300	16.178	35.0088	34.73	64.9712	64.63
$x = 0.5, y$	0.8066	0.813	0.8312s	0.823	0.8535	0.855	0.8535	0.850
v_{max}	3.7474	3.697	19.6857	19.617	68.9947	68.59	220.5700	219.36
$x, y = 0.5$	0.1934	0.178	0.1262	0.119	0.0642	0.066	0.0339	0.0379
Nu_0	1.1723	1.117	2.2695	2.238	4.6203	4.509	9.4660	8.817
Nu_{max}	1.5945	1.505	3.7414	3.528	8.1868	7.717	18.6560	17.925
$x = 0, y$	0.0917	0.092	0.1465	0.143	0.0771	0.081	0.0428	0.0378
Nu_{min}	0.7305	0.692	0.6283	0.586	0.7946	0.729	1.0937	0.989
$x = 0, y$	1.0000	1	1.0000	1	1.0000	1	1.0000	1

Table 5.1: Comparison of the present numerical results (PRE) and the bench mark solutions by de Vahl Davis (DVD).

The $Ra = 10^6$ test was performed from scratch, that is, the fluid was at rest initially, the temperature distribution over the domain was zero, and an initial distribution of $\nu_t^* = Pr/100$ as previously described was used. Initially, the solution develops as the laminar solution does. At some point in time during the simulation, small oscillations appear near the isothermal walls. These oscillations seem to be the result of the turbulence model attempting to take the solution from a laminar state to a turbulent state. However, as time progresses these oscillations are quenched. After these oscillations disappear, the solution proceeds forwards in time in such a way that the turbulent eddy-viscosity approaches zero. After a long time, the solution reaches the state in figure 5.9. As expected, the resulting velocity and temperature fields are almost identical in both cases

The $Ra = 10^6$ results were used as the initial conditions for the case $Ra = 10^7$. Figure 5.10 is the state reached by the turbulent simulator where the turbulent viscosity is extremely small but non-zero. As can be seen, the agreement between the solutions is very good at this Rayleigh number too, as expected.

Figure 5.9: Comparison between laminar and turbulent simulations at $Ra = 10^6$.

5.4 Turbulent simulations for Ra between 10^8 and 10^{10}

Since the simulations at some laminar Rayleigh numbers were successful using the turbulence model in question, it was thought that solutions at higher Rayleigh num-

Figure 5.10: Comparison between laminar and turbulent simulations at $Ra = 10^7$.

bers would not be much harder to come by. That was simply not true. Throughout our study of turbulent of these turbulent flows, a significant number of technical difficulties in applying the turbulence model were encountered, which are independent of the solver implementation. The OOFEA toolkit served well as the development platform for the simulator: it permitted the straightforward declaration of the necessary weak forms and the solution of the governing matrix equations. Furthermore, it allowed the direct manipulation of the boundary conditions for the turbulence model, which turn out to be extremely important in order to obtain non-diverging solutions of the problem.

As it turns out, the major hurdle in turbulent simulations is the appropriate resolution of the boundary layer so as to obtain reasonable profiles for the different field variables, such as the vertical velocity, but in particular the turbulence kinetic energy and the rate of dissipation of the turbulence kinetic energy (i.e. v^* , k^* and ϵ^* in their dimensionless form). On the other hand, another major hurdle in this study has been the lack of time-dependent studies of these flows in the literature. Most studies in the literature use steady-formulations, for which a number of tricks, acceleration techniques, exist and can be used to reach a steady-state solution. However, these steady results cannot predict the richness of behavior that can be observed via time-dependent simulations. Necessarily then, they do not experience the same kind of difficulties, and seldom can guide the researcher in the right path.

To begin with, after the successful simulation of a laminar solution for $Ra = 10^7$ using the k - ϵ turbulence model, a simulation for $Ra = 10^8$ was performed using the same 33×33 grid point mesh with a clustering parameter of $\beta = 1.05$ was attempted. Markatos and Pericleous [51] present results for this and higher Rayleigh numbers

using a very similar k - ϵ model, but in its steady form. The results for $Ra = 10^7$ were used as initial conditions.

Initially, the simulation seemed to be approaching the steady-state described by Markatos and Pericleous, but it became obvious as more and more iterations were performed that the solution was not converging to a particular state, but seemed to approach the steady solution to later on return to a previous state. At first, we thought that we might be observing some kind of transition phenomena, a jump from laminar to turbulent flow since 10^8 seems to be a borderline value between fully turbulent and laminar flows. For that reason, it was decided to proceed with the calculation at a larger Ra such as 10^9 .

At $Ra = 10^9$, the boundary layer is even thinner than at previous Rayleigh numbers, and we suspected that after using a 33×33 mesh for the previous three Rayleigh numbers of interest, a finer mesh would be needed. A decision was made then to use a 51×51 mesh with a clustering parameter $\beta = 1.01$ for this new simulation. Using the results at $Ra = 10^7$ as an initial condition, an oscillating solution was again found after a large number of iterations (about 6000). At this Rayleigh number, the oscillations could not be attributed to a transition phenomena. However, under careful scrutiny, it was observed that the dominating wall bounded region of the flow seemed to have a steadier profile of variables. As it turns out, several other researchers have observed steady oscillations of the time-averaged variables of the flow. In particular, Staehle and Hahne [53] observed damped oscillations in these types of flows, and noticed that their dampening decreases as the Rayleigh number increases. Ozoe *et al.* [47], who also perform a time-dependent analysis of these enclosed flows observes exactly the same behavior we observe and suggest that it corresponds to real physical behavior. None of these oscillations can be observed if steady formulations are

used, and that is why they are not reported in most of the numerical work on natural convective flows in enclosures.

If we accept that these oscillations within the core of the flow are physical, and that these oscillations are damped within the relatively fast moving boundary layers at this Rayleigh number, why is it that similar behavior in the boundary layers was not found at $Ra = 10^8$. Careful examination of the distribution of v^* and k^* at various locations within the domain, reveals that it is not only important to resolve the peak of the velocity in the boundary layer, but also to resolve the peaks of the turbulence kinetic energy. It is true that less resolution is needed to resolve the velocity peak than the turbulence kinetic energy peak. If these peaks are not resolved, then the distributions of the velocity and turbulence kinetic energy are not physical correct and the whole cavity flow is affected as the simulation progresses. If one examines these profiles for the cases between 10^6 and 10^9 previously discussed, one can discover that the velocity peak was adequately resolved in all cases with the selected meshes, while the peak of the turbulence kinetic energy profile was adequately resolved in every case *except* in that where the Rayleigh number was 10^8 . It seems then that using a finer mesh such as the 51×51 mesh used for the $Ra = 10^9$ case would have yielded the solution we sought. Figure 5.11 represents the wall bound portion of the distribution of v^* and k^* at the horizontal centerline of the cavity ($y^* = 0.5$) for the cases of $Ra = 10^6$ and $Ra = 10^7$, while figure 5.12 are those for the cases of $Ra = 10^9$ and $Ra = 10^{10}$. There are enough grid points surrounding the peak value of both v^* and k^* to provide a turbulent solution. On the other hand, when compared to the top part of figure 5.13, it is clear that the k^* distribution has not been resolved correctly and extra peaks are observed near the wall. The bottom part of the figure results

from refining the mesh from 33×33 grid points to 55×55 grid points as previously suggested. The change in the resolution of the peak value is obvious.

Figure 5.11: Distribution of the dimensionless vertical velocity and the dimensionless turbulence kinetic energy in the horizontal centerline of the cavity in the region next to the left isothermal wall for the cases of $Ra = 10^6$ (top) and $Ra = 10^7$ (bottom).

As for the $Ra = 10^{10}$ case, preliminary results of simulations using the 10^9 results as initial conditions and the same mesh structure indicate that the mesh is adequate to resolve the velocity and turbulence kinetic energy peaks. Only time is needed to let the boundary layer flow stabilize.

Some of the transient behavior observed in these simulations is quite interesting. As the Rayleigh number is suddenly increased, as when a simulation for a particular Rayleigh number is started with results from a similar simulation at a lower Rayleigh number, the velocity at which the fluid near the isothermal walls moves increases due to the increased strength of the coupling between the momentum and energy equations. This increase in velocity forces the fluid moving up the hot left wall, and

Figure 5.12: Distribution of the dimensionless vertical velocity and the dimensionless turbulence kinetic energy in the horizontal centerline of the cavity in the region next to the left isothermal wall for the cases of $Ra = 10^9$ (top) and $Ra = 10^{10}$ (bottom).

Figure 5.13: Distribution of the dimensionless vertical velocity and the dimensionless turbulence kinetic energy in the horizontal centerline of the cavity in the region next to the left isothermal wall for the case of $Ra = 10^8$ when the a 33×33 mesh is used (top) and a 51×51 mesh is used (bottom).

the fluid moving down the cold right wall, to impact the corners of the cavity with more force. As the fluid turns due to the geometry, the flow is observed to recirculate at the top and bottom adiabatic walls. However, as time progresses, the intensity of the turbulence also increases due to the increase in the Rayleigh number and the recirculations over the adiabatic boundaries are dissipated.

Figure 5.14: Transient recirculation near the adiabatic walls of the cavity as observed during the time-accurate simulation of turbulent thermally driven cavity flows.

Another interesting effect of these sudden changes in the Rayleigh number has to do with the formation of rotation cells between other rotation cells, as in the laminar case for $Ra = 10^6$. Markatos and Pericleous [51] were surprised to observe a third clockwise rotating cell form in the middle of the cavity when they expected a counterclockwise one to reduce the shear between the other two. In their steady results for $Ra = 10^{10}$ the similar set of three clockwise rotating cells is observed. In our preliminary results at this Rayleigh number, early on in the simulation, three circulation cells can also be found (arguably they are the same cells at a previous state). However, as they expected, the cell in the core rotates *counterclockwise*.

Since the turbulence has not yet had time to compensate for the change in Rayleigh number, there is not enough dissipation yet to dissipate this motions, such that the temperature gradients in the core can generate clockwise circulations instead.

Figure 5.15: Transient counterclockwise rotating cell within the core of the flow as observed during the time-accurate simulation of turbulent thermally driven cavity flows.

Figure 5.16 represents the mean flow characteristics of the flow for the turbulent case where the Rayleigh number is 10^9 . These results were obtained after a very large number of iterations. The reason is mainly because we were expecting a converging solution, and instead obtained an oscillating one as previously described. However, at the stage presented, the results in the region of dominance (not in the core) have stabilized considerably. In particular, the core is again thermally stratified, and the thermal gradient near the vertical walls is extremely large. In fact, by looking at the Nusselt number distribution over the heated vertical left wall, the Nusselt number is about 10 times larger than that observed a $Ra = 10^6$. Furthermore, by examining figure 5.18 and table 5.2, the maximum nondimensional turbulent viscosity is about 7.58, which is about eleven (11) times larger than the Prandtl number ($Pr = 0.71$)

which indicates that the maximum turbulent diffusivity is about eleven (11) times the molecular kinematic viscosity. The observed patterns are qualitatively similar to those observed for similar problems in the literature [47][55][51].

Figure 5.16: From top left to bottom right: Streamlines, isotherms, pressure contours and vector plot for the turbulent solution of the thermally driven cavity flow at $Ra = 10^9$.

$ \Psi_{max} $	208.7369
x	0.3945
y	0.5534
u_{max}	2470.3154
x	0.0685
y	0.9874
v_{max}	6017.5696
x	0.0066
y	0.3945
$(\nu_t)_{max}$	7.5770
x	0.9557
y	0.2548
k_{max}	2.0548×10^6
x	0.0278
y	0.6055
ϵ_{max}	8.5980×10^{10}
x	0.9974
y	0.5534
Nu_0	81.3930
Nu_{max}	186.6759
$x = 0, y$	0.0044
Nu_{min}	6.0957
$x = 0, y$	1.0000

Table 5.2: Quantitative results obtained from the simulation of turbulent flow inside a thermally driven cavity at a Rayleigh number of 10^9 . The variables in the table are the nondimensional variables even though the * notation is not being used.

Figure 5.17: From left to right: Velocity profiles for the dimensionless vertical and horizontal velocities of the flow at the horizontal and vertical centerlines of the cavity ($y^* = 0.5$ and $x^* = 0.5$ for the turbulent case with $Ra = 10^9$).

Figure 5.18: From top left to bottom center: Contour plots of the nondimensional turbulence kinetic energy k^* , the dimensionless rate of dissipation of turbulence kinetic energy ϵ^* , and the nondimensional turbulence viscosity for the turbulent case with $Ra = 10^9$.

Figure 5.19: Distribution of the Nusselt number over the vertical isothermal left wall of the cavity for the turbulent case with $Ra = 10^9$.

Chapter 6

Laminar Three Dimensional Natural Convection in a Cubic Cavity

6.1 Thermally driven cavity

The domain in question is a rectangular cavity of unit height, width and depth. Its left and right vertical walls are isothermal, such that the left wall is at a higher temperature than the right wall. On the other hand, the top, bottom, front and back walls are adiabatic. In particular, no-slip boundary conditions are applied to all the fluid velocities, and the initial conditions are either rest conditions, or previous solutions to the problem (i.e. for example, the $Ra = 10^3$ solution can be used as an initial condition for the $Ra = 10^4$ case). These conditions imply that the normal gradient of the pressure at all boundaries is identically zero.

6.2 Three dimensional results

The three-dimensional numerical experiments have been compared with those by Fusegi *et al.* [35] which performed similar numerical experiments utilizing a control-volume based finite difference procedure with an iterative solver, together with the a pressure correction algorithm and some convergence acceleration techniques. Their simulations were carried out in an irregularly spaced $62 \times 62 \times 62$ mesh.

In contrast, the results in this study were obtained in a regularly spaced $41 \times 41 \times 17$ mesh using the Navier-Stokes-Energy simulator developed with the OOFEA toolkit

and using linear isoparametric 8-node quadrilateral elements, and the semi-implicit projection scheme described in chapter 4.

The resulting matrix equations for the three-dimensional laminar case can be obtained readily from equations 4.35-4.68 by setting the turbulent viscosity ν_t to zero, and by eliminating the k and ϵ equations and all coefficient matrices which depend on them.

6.2.1 Temperature and velocity profiles for $Ra = 10^4$

By observing figure 6.1 it can be seen that the isotherms are very similar to those obtained from the two-dimensional solution. On the other hand, as one approaches the insulated front and back walls, the end effects cause some variation on the pattern. The effect is a reduction in the convective effects, i.e. the conduction effects are more pronounced near the insulated walls. Since the dimensions of the box in the z -direction are comparable to those in the x and y directions these three-dimensional effects are expected.

A better characterization of these three-dimensional effects can be obtained by observing the isovelocity manifolds. As expected from the unicellular clockwise roll of the fluid at this Rayleigh number, the vertical velocities v have a larger magnitudes near the isothermal left and right walls of the cavity, while the horizontal velocities u have larger magnitudes near the top and bottom insulated walls. The end effects help close the isovelocity manifolds since there is a no slip boundary condition at all boundaries including the front and back walls. However, it is not until one examines the transverse velocity w isovelocity manifolds that the three-dimensional effects are evident.

First of all, the relative magnitude of the w velocity components is about one order of magnitude smaller than the u and v velocity components. Furthermore, these three-dimensional effects can be seen to reside in the neighborhood of the cavity's top-left corners and bottom right corners, where the fluid rising near the hot wall and the fluid falling near the cold wall impinge on the insulated top and bottom walls, respectively. Moreover, it can be seen that there are regions of activity near the cavity's central z axis, on both sides of the $z = 0.5$ plane. In fact, there is an underlying symmetry across this plane, where the velocities on one side are the negative value of the ones on the opposite side. On the other hand, it is interesting to observe no significant three-dimensional effects near the bottom-left and top-right corners of the domain. This will change as the Rayleigh number increases, as will be seen in the next section.

6.2.2 Temperature and velocity profiles for $Ra = 10^6$

By observing figure 6.2 it can be seen that at this Rayleigh number, a thermal boundary layer has formed near the isothermal walls, and that the core is has a linearly stratified temperature distribution. This is a drastic change from that observed at $Ra = 10^4$. However, the nature of such behavior is two-dimensional, since it is clearly observed in two-dimensional simulations, and since the three-dimensional variations in the temperature field are almost negligible except very close to the adiabatic front and back walls.

In terms of the fluid motion, the unicellular roll of lower Rayleigh numbers is no longer present. It has been replaced by two independent cells close to the isothermal walls which also roll clockwise. At the surface of the isothermal walls, thin momentum boundary layers have formed which match the temperature boundary layers previously described. This is evident in the v isovelocity manifolds. Although no boundary

layer is formed, the larger magnitude u isovelocity manifolds are now confined to a smaller region next to the top and bottom adiabatic walls.

The transverse velocity w isovelocity manifolds demonstrate the change in the nature of the three-dimensional motions as opposed to the lower Rayleigh number case. The effects near the central z axis are no longer equally important as the three-dimensional effects near the corners. In fact, the importance of three-dimensional motions in all corners of the cavity has now become more significant. Nevertheless, the magnitude of the transverse velocity contributions is still an order of magnitude smaller than the corresponding vertical and horizontal velocity components.

Finally, from the lack of dominating motions within the core of the cavity, it is evident that the core is nearly stagnant at this Rayleigh number, as observed in two-dimensional simulations.

6.3 Heat transfer characteristics

In order to characterize the heat transfer rate in enclosures of this type the dimensionless heat transfer rate or Nusselt number Nu at the isothermal walls can be calculated. There is interest in determining the maximum, minimum and mean values of the Nusselt number at these walls. The results of the study as well as those of Fusegi *et al.* point out that differences in the heat transfer rate between two and three-dimensional simulations are more pronounced at lower Rayleigh numbers. Since three-dimensional effects are observed to confine themselves to regions near the cavity's corners as the Rayleigh number increases, it is not surprising that the differences become less pronounced as the Rayleigh number increases. These differences however, range between 2% at most Rayleigh numbers to 10% for the minimum Nusselt

number at $Ra = 10^5$, a significantly different predicted value. This is also observed by Fusegi *et al.* in their study.

Table 6.1 summarizes and compares the results of the numerical experiments in this study and those by Fusegi *et al.* [35]. Excellent agreement is observed even at our coarser mesh resolution.

Unknown	Ra							
	10^3		10^4		10^5		10^6	
	PRS	FUS	PRS	FUS	PRS	FUS	PRS	FUS
u_{max}	0.1321	0.1314	0.1983	0.2013	0.1587	0.1468	0.1320	0.08416
y	0.1750	0.2000	0.1750	0.1833	0.1250	0.1453	0.0500	0.1443
v_{max}	0.1337	0.1320	0.2210	0.2252	0.2412	0.2471	0.2416	0.2588
x	0.8250	0.8333	0.8750	0.8833	0.9250	0.9353	0.9750	0.9669
Nu_{max}	1.4191	1.4200	3.6104	3.6520	7.8709	7.7950	16.5170	17.6700
y	0.1000	0.0833	0.1500	0.6232	0.1000	0.08256	0.0500	0.03793
Nu_{min}	0.7306	0.7639	0.5624	0.6110	0.7139	0.7867	1.0874	1.2570
y	1.0000	1.0000	1.0000	1.000	1.0000	1.0000	1.0000	1.0000
Nu_{mean}	1.0875	1.1050	2.2363	2.3020	4.5776	4.6460	8.8168	9.012

Table 6.1: Comparison of representative numerical results in the present study (PRS) and the solutions by Fusegi *et al.* (FUS) for the symmetry xy plane at $z = 0.5$.

Figure 6.1: Top left: temperature contours inside the cavity for $Ra = 10^4$. Contour levels: A) 0.9667, B) 0.9833, C) 1.0, D) 1.017, E) 1.033. Top right to bottom right: isovelocity contours for u -velocity, v -velocity and w -velocity components respectively for $Ra = 10^4$. Contour levels for u : A) -0.1333, B) -0.06667, C) 0.06667, D) 0.1333. Contour levels for v : A) -0.15, B) -0.075, C) 0.075, D) 0.15. Contour levels for w : A) -0.01667, B) -0.008333, C) 0.008333, D) 0.01667.

Figure 6.2: Top left: temperature contours inside the cavity for $Ra = 10^6$. Contour levels: A) 0.9667, B) 0.9833, C) 1.0, D) 1.017, E) 1.033. Top right to bottom right: isovelocity contours for u -velocity, v -velocity and w -velocity components respectively for $Ra = 10^6$. Contour levels for u : A) -0.12, B) -0.06, C) 0.06, D) 0.12. Contour levels for v : A) -0.2, B) -0.1, C) 0.1, D) 0.2. Contour levels for w : A) -0.024, B) -0.012, C) 0.012, D) 0.024.

Chapter 7

General Conclusions

7.1 Concluding remarks

In this dissertation, an object-oriented implementation of the finite element method in C++ has been introduced, designed and tested in the context of computational fluid dynamics applications. The resulting system of class hierarchies allows an user to computationally represent arbitrary weak forms obtained from the mathematical analysis of problems of interest. These weak forms can then be managed, evaluated, and assembled into linear algebra entities which can then be efficiently manipulated. The resulting OOFEA toolkit can then be used to rapidly develop new applications, and implement new concepts and ideas in a coherent and integrated platform.

A significant number of problems has been attacked to introduce the syntax and functionality of the toolkit, culminating in its use for the development of a laminar and turbulent flow simulator for natural convection in enclosures.

The simulation of the turbulent flows turned out to be much more complicated than expected. In particular, the selection of appropriate boundary conditions, mesh resolution and initial conditions is never straightforward. Furthermore, even when the time-dependent formulation which works so well with the laminar simulations is used, very long simulations are required to obtain adequate turbulent results. Also, the possibility of comparison with other simulations is also a very complicated issue, since a significant number of slightly different approaches exist in the literature. On the other hand, our main goal was to attack a complex and significant CFD application of

interest with the OOFEA toolkit. This task has been accomplished, and a significant contribution has been made.

The future of CFD and other computationally intensive research areas seems directly linked to object-oriented design. The further along the research community comes to actually simulate all interesting phenomena related to practical engineering problems, the more complex and flexible the resulting codes will have to become in order to justify their use and development costs. Complexity and flexibility are two concepts that do not in general coexist in harmony. However, object-oriented techniques have these types of concerns in mind, and provide a design and programming philosophy in which careful design and abstraction of principles can yield complex and flexible computational tools.

7.2 Future work

In terms of the OOFEA toolkit, there are several areas which can be extended. In particular, the toolkit currently implements the Galerkin method of weighted residuals. This simply implies that the interpolation functions used to describe the mathematical behavior of the weighting function and the residual are the same. However, any other method of weighted residuals could be implemented if the `element` class is re-defined to include both interpolation functions for the weighting function and for the residual. This would simply be an extension of the toolkit, requiring the inclusion of a small number of new methods in the `element` class. Such an extension would yield an even more general toolkit since other popular PDE solution techniques could be identically performed (e.g. finite volume method using a finite element method together with the subdomain method of weighted residuals).

The mathematical management tools can also be enhanced. In particular, after a linear algebra interface is provided, the actual implementation of the mathematical operations is transparent to the user (he/she has no direct contact with the implementation of the operations). Transparent parallelization of the operations is the a possibility. Every time the user requires a linear algebra operation, the appropriate parallel operation is performed using parallel programming tools such as PVM or MPI. Although this more simply said than done, the concept is clear: with encapsulation, the inner workings of the mathematical tools can be hidden from the user. Parallel programming is simply to complex today for the average engineer to master. Hence, transparent interfaces like those which can be generated in object-oriented languages may be the only logical way of transferring these technologies to the user community.

Finally, the representation of scientific and engineering knowledge is not exclusively mathematically rigorous (not everything is governing equations). Experience as well as empirical information are other sources of engineering knowledge which in general have been difficult to represent in the past. With the advent of "softer" computing techniques such as expert systems, artificial neural networks and fuzzy logic, these other types of knowledge can be represented in a computational environment. Object-oriented implementation of these techniques seems natural and has been used. By creating an object-oriented FEA toolkit, we are making both types of knowledge share the same common language. It is then time to attack problems using both computing arsenals. We intend to pursue these ideas in the near future, since we have already developed object-oriented neural network and fuzzy logic simulators.

Appendix A

Turbulent natural convection simulator

Tables A.1 to A.21 are listings of all sections which compose the turbulent natural convection simulator described in chapter 4 and used to generate the results of chapter 5.

```
temp_matrix NS2D_stiffness_matrix(domain_element& any_element)
{
    matrix& B=any_element.get_B_matrix();
    matrix& inv_J=any_element.get_inv_Jacobian();
    double det_J=any_element.get_det_Jacobian();
    return transpose(B)*transpose(inv_J)*inv_J*B*det_J;
}
```

Table A.1: Listing of the K weak form using the OOFEA toolkit.

```
temp_matrix NS2D_momentum_stiffness_matrix(domain_element& any_element)
{
    matrix& B=any_element.get_B_matrix();
    matrix& inv_J=any_element.get_inv_Jacobian();
    double det_J=any_element.get_det_Jacobian();
    matrix& k=any_element.get_local_solution(7);
    matrix& epsilon=any_element.get_local_solution(8);
    double c_mu=0.09;
    matrix nu_t(k.get_rows(),k.get_columns(),0,"local nu_t");
    for (int i=1;i<=k.get_rows();i++)
    {
        if (epsilon(i,1)<=1e-10) nu_t(i,1)=0;
        else nu_t(i,1)=c_mu*k(i,1)*k(i,1)/epsilon(i,1);
    }
    double nu_t_avg;
    nu_t_avg=0.25*(nu_t(1,1)+nu_t(2,1)+nu_t(3,1)+nu_t(4,1));
    double Pr=0.71;
    return transpose(B)*transpose(inv_J)*inv_J*B*det_J*(nu_t_avg+Pr);
}
```

Table A.2: Listing of the K_m weak form using the OOFEA toolkit.

```

temp_matrix NS2D_temperature_stiffness_matrix(domain_element& any_element)
{
    matrix& B=any_element.get_B_matrix();
    matrix& inv_J=any_element.get_inv_Jacobian();
    double det_J=any_element.get_det_Jacobian();
    matrix& k=any_element.get_local_solution(7);
    matrix& epsilon=any_element.get_local_solution(8);
    double c_mu=0.09;
    double sigma_t=0.9; //sigma_t is between 0.9-1.0
    matrix nu_t(k.get_rows(),k.get_columns(),0,"local nu_t");
    for (int i=1;i<=k.get_rows();i++)
    {
        if (epsilon(i,1)<=1e-10) nu_t(i,1)=0;
        else nu_t(i,1)=c_mu*k(i,1)*k(i,1)/epsilon(i,1);
    }
    double nu_t_avg;
    nu_t_avg=0.25*(nu_t(1,1)+nu_t(2,1)+nu_t(3,1)+nu_t(4,1));
    return transpose(B)*transpose(inv_J)*inv_J*B*det_J*((nu_t_avg/sigma_t)+1.0);
}

```

Table A.3: Listing of the K_t weak form using the OOFEA toolkit.

```

temp_matrix NS2D_k_stiffness_matrix(domain_element& any_element)
{
    matrix& B=any_element.get_B_matrix();
    matrix& inv_J=any_element.get_inv_Jacobian();
    double det_J=any_element.get_det_Jacobian();
    matrix& k=any_element.get_local_solution(7);
    matrix& epsilon=any_element.get_local_solution(8);
    double c_mu=0.09;
    double sigma_k=1.00;
    matrix nu_t(k.get_rows(),k.get_columns(),0,"local nu_t");
    for (int i=1;i<=k.get_rows();i++)
    {
        if (epsilon(i,1)<=1e-10) nu_t(i,1)=0;
        else nu_t(i,1)=c_mu*k(i,1)*k(i,1)/epsilon(i,1);
    }
    double nu_t_avg;
    nu_t_avg=0.25*(nu_t(1,1)+nu_t(2,1)+nu_t(3,1)+nu_t(4,1));
    double Pr=0.71;
    return transpose(B)*transpose(inv_J)*inv_J*B*det_J*((nu_t_avg/sigma_k)+Pr);
}

```

Table A.4: Listing of the K_k weak form using the OOFEA toolkit.

```

temp_matrix NS2D_epsilon_stiffness_matrix(domain_element& any_element)
{
    matrix& B=any_element.get_B_matrix();
    matrix& inv_J=any_element.get_inv_Jacobian();
    double det_J=any_element.get_det_Jacobian();

```

```

matrix& k=any_element.get_local_solution(7);
matrix& epsilon=any_element.get_local_solution(8);
double c_mu=0.09;
double sigma_e=1.30;
matrix nu_t(k.get_rows(),k.get_columns(),0,"local nu_t");
for (int i=1;i<=k.get_rows();i++)
{
    if (epsilon(i,1)<=1e-10) nu_t(i,1)=0;
    else nu_t(i,1)=c_mu*k(i,1)*k(i,1)/epsilon(i,1);
}
double nu_t_avg;
nu_t_avg=0.25*(nu_t(1,1)+nu_t(2,1)+nu_t(3,1)+nu_t(4,1));
double Pr=0.71;
return transpose(B)*transpose(inv_J)*inv_J*B*det_J*((nu_t_avg/sigma_e)+Pr);
}

```

Table A.5: Listing of the K_e weak form using the OOFEA toolkit.

```

temp_matrix NS2D_capacitance_matrix(domain_element& any_element)
{
    matrix& N=any_element.get_N_vector();
    double det_J=any_element.get_det_Jacobian();
    return transpose(N)*N*det_J;
}

```

Table A.6: Listing of the M weak form using the OOFEA toolkit.

```

temp_matrix NS2D_NL_convective_matrix(domain_element& any_element)
{
    matrix& N=any_element.get_N_vector();
    matrix& B=any_element.get_B_matrix();
    matrix& inv_J=any_element.get_inv_Jacobian();
    matrix& u=any_element.get_local_solution(1);
    matrix& v=any_element.get_local_solution(3);
    double det_J=any_element.get_det_Jacobian();
    matrix velocity(4,2,"velocity");
    velocity(1,1)=u(1,1);velocity(2,1)=u(2,1);
    velocity(3,1)=u(3,1);velocity(4,1)=u(4,1);
    velocity(1,2)=v(1,1);velocity(2,2)=v(2,1);
    velocity(3,2)=v(3,1);velocity(4,2)=v(4,1);
    return transpose(N)*N*velocity*inv_J*B*det_J;
}

```

Table A.7: Listing of the P weak form using the OOFEA toolkit.

```

temp_matrix NS2D_d_dx(domain_element& any_element)
{
    matrix& N=any_element.get_N_vector();
    matrix& B=any_element.get_B_matrix();
    matrix& inv_J=any_element.get_inv_Jacobian();
}

```

```

double det_J=any_element.get_det_Jacobian();
matrix BXY; //Stores inv(J)*B
BXY=inv_J*B;
matrix FRB(1,4,"first row of B"); //First row of B matrix
FRB(1,1)=BXY(1,1);FRB(1,2)=BXY(1,2);FRB(1,3)=BXY(1,3);FRB(1,4)=BXY(1,4);
return transpose(N)*FRB*det_J;
}

```

Table A.8: Listing of the D_DX weak form using the OOFEA toolkit.

```

temp_matrix WS2D_d_dy(domain_element& any_element)
{
matrix& N=any_element.get_N_vector();
matrix& B=any_element.get_B_matrix();
matrix& inv_J=any_element.get_inv_Jacobian();
double det_J=any_element.get_det_Jacobian();
matrix BXY; //Stores inv(J)*B
BXY=inv_J*B;
matrix SRB(1,4,"second row of B"); //Second row of B matrix;
SRB(1,1)=BXY(2,1);SRB(1,2)=BXY(2,2);SRB(1,3)=BXY(2,3);SRB(1,4)=BXY(2,4);
return transpose(N)*SRB*det_J;
}

```

Table A.9: Listing of the D_DY weak form using the OOFEA toolkit.

```

temp_matrix KE2D_EK_matrix(domain_element& any_element)
{
matrix& N=any_element.get_N_vector();
matrix& k=any_element.get_local_solution(7);
matrix& epsilon=any_element.get_local_solution(8);
double det_J=any_element.get_det_Jacobian();
double k_avrg, epsilon_avrg;
k_avrg=0.25*(k(1,1)+k(2,1)+k(3,1)+k(4,1));
epsilon_avrg=0.25*(epsilon(1,1)+epsilon(2,1)+epsilon(3,1)+epsilon(4,1));
if (k_avrg<=1e-15) {k_avrg=1;epsilon_avrg=0;}
return transpose(N)*N*(det_J*epsilon_avrg/k_avrg);
}

```

Table A.10: Listing of the EK weak form using the OOFEA toolkit.

```

temp_matrix KE2D_G_matrix(domain_element& any_element)
{
matrix& N=any_element.get_N_vector();
matrix& B=any_element.get_B_matrix();
matrix& inv_J=any_element.get_inv_Jacobian();
double det_J=any_element.get_det_Jacobian();
matrix BXY; //Stores inv(J)*B
BXY=inv_J*B;
matrix SRB(1,4,"second row of B"); //Second row of B matrix;
SRB(1,1)=BXY(2,1);SRB(1,2)=BXY(2,2);SRB(1,3)=BXY(2,3);SRB(1,4)=BXY(2,4);
}

```

```

matrix& k=any_element.get_local_solution(7);
matrix& epsilon=any_element.get_local_solution(8);
double c_mu=0.09;
matrix nu_t(k.get_rows(),k.get_columns(),0,"local nu_t");
for (int i=1;i<=k.get_rows();i++)
{
    if (epsilon(i,1)<=1e-10) nu_t(i,1)=0;
    else nu_t(i,1)=c_mu*k(i,1)*k(i,1)/epsilon(i,1);
}
double nu_t_avrg;
nu_t_avrg=0.25*(nu_t(1,1)+nu_t(2,1)+nu_t(3,1)+nu_t(4,1));
return transpose(N)*SRB*(det_J*nu_t_avrg);
}

```

Table A.11: Listing of the G_k weak form using the OOFEA toolkit.

```

temp_matrix KE2D_EKG_matrix(domain_element& any_element)
{
    matrix& N=any_element.get_N_vector();
    matrix& B=any_element.get_B_matrix();
    matrix& inv_J=any_element.get_inv_Jacobian();
    double det_J=any_element.get_det_Jacobian();
    matrix BXY; //Stores inv(J)*B
    BXY=inv_J*B;
    matrix SRB(1,4,"second row of B"); //Second row of B matrix;
    SRB(1,1)=BXY(2,1);SRB(1,2)=BXY(2,2);SRB(1,3)=BXY(2,3);SRB(1,4)=BXY(2,4);
    matrix& k=any_element.get_local_solution(7);
    matrix& epsilon=any_element.get_local_solution(8);
    double c_mu=0.09;
    matrix nu_t(k.get_rows(),k.get_columns(),0,"local nu_t");
    for (int i=1;i<=k.get_rows();i++)
    {
        if (epsilon(i,1)<=1e-10) nu_t(i,1)=0;
        else nu_t(i,1)=c_mu*k(i,1)*k(i,1)/epsilon(i,1);
    }
    double nu_t_avrg;
    nu_t_avrg=0.25*(nu_t(1,1)+nu_t(2,1)+nu_t(3,1)+nu_t(4,1));
    double k_avrg, epsilon_avrg, c_3e_avrg;
    k_avrg=0.25*(k(1,1)+k(2,1)+k(3,1)+k(4,1));
    epsilon_avrg=0.25*(epsilon(1,1)+epsilon(2,1)+epsilon(3,1)+epsilon(4,1));
    if (k_avrg<=1e-15) {k_avrg=1;epsilon_avrg=0;}
    matrix& u=any_element.get_local_solution(1);
    matrix& v=any_element.get_local_solution(3);
    matrix c_3e;c_3e=tanh(abs(u->reciprocal(v)));
    c_3e_avrg=0.25*(c_3e(1,1)+c_3e(2,1)+c_3e(3,1)+c_3e(4,1));
    return transpose(N)*SRB*(det_J*nu_t_avrg*c_3e_avrg*epsilon_avrg/k_avrg);
}

```

Table A.12: Listing of the G_c weak form using the OOFEA toolkit.

```

temp_matrix KE2D_PK_u_matrix(domain_element& any_element)
{
    matrix& N=any_element.get_N_vector();
    matrix& B=any_element.get_B_matrix();
    matrix& inv_J=any_element.get_inv_Jacobian();
    double det_J=any_element.get_det_Jacobian();
    matrix BXY; //Stores inv(J)*B
    BXY=inv_J*B;
    matrix FRB(1,4,"first row of B"); //First row of B matrix
    matrix SRB(1,4,"second row of B"); //Second row of B matrix;
    FRB(1,1)=BXY(1,1);FRB(1,2)=BXY(1,2);FRB(1,3)=BXY(1,3);FRB(1,4)=BXY(1,4);
    SRB(1,1)=BXY(2,1);SRB(1,2)=BXY(2,2);SRB(1,3)=BXY(2,3);SRB(1,4)=BXY(2,4);
    matrix& u=any_element.get_local_solution(1);
    matrix& k=any_element.get_local_solution(7);
    matrix& epsilon=any_element.get_local_solution(8);
    double c_mu=0.09;
    matrix nu_t(k.get_rows(),k.get_columns(),0,"local nu_t");
    for (int i=1;i<=k.get_rows();i++)
    {
        if (epsilon(i,1)<=1e-10) nu_t(i,1)=0;
        else nu_t(i,1)=c_mu*k(i,1)*k(i,1)/epsilon(i,1);
    }
    double nu_t_avg;
    nu_t_avg=0.25*(nu_t(1,1)+nu_t(2,1)+nu_t(3,1)+nu_t(4,1));
    matrix result;
    result=2.0*transpose(N)*FRB*u*FRB*det_J*nu_t_avg; //2(du/dx)^2 term
    return result+transpose(N)*SRB*u*SRB*det_J*nu_t_avg; //(du/dy)^2 term
}

```

Table A.13: Listing of the PK_u weak form using the OOFEA toolkit.

```

temp_matrix KE2D_PK_v_matrix(domain_element& any_element)
{
    matrix& N=any_element.get_N_vector();
    matrix& B=any_element.get_B_matrix();
    matrix& inv_J=any_element.get_inv_Jacobian();
    double det_J=any_element.get_det_Jacobian();
    matrix BXY; //Stores inv(J)*B
    BXY=inv_J*B;
    matrix FRB(1,4,"first row of B"); //First row of B matrix
    matrix SRB(1,4,"second row of B"); //Second row of B matrix;
    FRB(1,1)=BXY(1,1);FRB(1,2)=BXY(1,2);FRB(1,3)=BXY(1,3);FRB(1,4)=BXY(1,4);
    SRB(1,1)=BXY(2,1);SRB(1,2)=BXY(2,2);SRB(1,3)=BXY(2,3);SRB(1,4)=BXY(2,4);
    matrix& u=any_element.get_local_solution(1);
    matrix& v=any_element.get_local_solution(3);
    matrix& k=any_element.get_local_solution(7);
    matrix& epsilon=any_element.get_local_solution(8);
    double c_mu=0.09;
    matrix nu_t(k.get_rows(),k.get_columns(),0,"local nu_t");
    for (int i=1;i<=k.get_rows();i++)
    {

```

```

    if (epsilon(i,1)<=1e-10) nu_t(i,1)=0;
    else nu_t(i,1)=c_mu*k(i,1)*k(i,1)/epsilon(i,1);
  }
double nu_t_avrg;
nu_t_avrg=0.25*(nu_t(1,1)+nu_t(2,1)+nu_t(3,1)+nu_t(4,1));
matrix result;
result=2.0*transpose(N)*SRB*v*SRB*det_J*nu_t_avrg; //2(dv/dy)^2 term
result=result+transpose(N)*FRB*v*FRB*det_J*nu_t_avrg; //(dv/dx)^2 term
return result+2.0*transpose(N)*SRB*u*FRB*det_J*nu_t_avrg; //2(du/dy)(dv/dx) term
}

```

Table A.14: Listing of the PKV_k weak form using the OOFEA toolkit.

```

temp_matrix KE2D_EKPK_u_matrix(domain_element& any_element)
{
  matrix& N=any_element.get_N_vector();
  matrix& B=any_element.get_B_matrix();
  matrix& inv_J=any_element.get_inv_Jacobian();
  double det_J=any_element.get_det_Jacobian();
  matrix BXY; //Stores inv(J)*B
  BXY=inv_J*B;
  matrix FRB(1,4,"first row of B"); //First row of B matrix
  matrix SRB(1,4,"second row of B"); //Second row of B matrix;
  FRB(1,1)=BXY(1,1);FRB(1,2)=BXY(1,2);FRB(1,3)=BXY(1,3);FRB(1,4)=BXY(1,4);
  SRB(1,1)=BXY(2,1);SRB(1,2)=BXY(2,2);SRB(1,3)=BXY(2,3);SRB(1,4)=BXY(2,4);
  matrix& u=any_element.get_local_solution(1);
  matrix& k=any_element.get_local_solution(7);
  matrix& epsilon=any_element.get_local_solution(8);
  double c_mu=0.09;
  matrix nu_t(k.get_rows(),k.get_columns(),0,"local nu_t");
  for (int i=1;i<=k.get_rows();i++)
  {
    if (epsilon(i,1)<=1e-10) nu_t(i,1)=0;
    else nu_t(i,1)=c_mu*k(i,1)*k(i,1)/epsilon(i,1);
  }
  double nu_t_avrg;
  nu_t_avrg=0.25*(nu_t(1,1)+nu_t(2,1)+nu_t(3,1)+nu_t(4,1));
  double k_avrg, epsilon_avrg;
  k_avrg=0.25*(k(1,1)+k(2,1)+k(3,1)+k(4,1));
  epsilon_avrg=0.25*(epsilon(1,1)+epsilon(2,1)+epsilon(3,1)+epsilon(4,1));
  if (k_avrg<=1e-15) {k_avrg=1;epsilon_avrg=0;}
  matrix result;
  result=2.0*transpose(N)*FRB*u*FRB*(det_J*nu_t_avrg*epsilon_avrg/k_avrg); //2(du/dx)^2 term
  return result+transpose(N)*SRB*u*SRB*(det_J*nu_t_avrg*epsilon_avrg/k_avrg); //2(du/dy)^2 term
}

```

Table A.15: Listing of the PKU_c weak form using the OOFEA toolkit.

```

temp_matrix KE2D_EKPK_v_matrix(domain_element& any_element)
{

```

```

matrix& J=any_element.get_J_vector();
matrix& B=any_element.get_B_matrix();
matrix& inv_J=any_element.get_inv_Jacobian();
double det_J=any_element.get_det_Jacobian();
matrix BXY; //Stores inv(J)*B
BXY=inv_J*B;
matrix FRB(1,4,"first row of B"); //First row of B matrix
matrix SRB(1,4,"second row of B"); //Second row of B matrix;
FRB(1,1)=BXY(1,1);FRB(1,2)=BXY(1,2);FRB(1,3)=BXY(1,3);FRB(1,4)=BXY(1,4);
SRB(1,1)=BXY(2,1);SRB(1,2)=BXY(2,2);SRB(1,3)=BXY(2,3);SRB(1,4)=BXY(2,4);
matrix& u=any_element.get_local_solution(1);
matrix& v=any_element.get_local_solution(3);
matrix& k=any_element.get_local_solution(7);
matrix& epsilon=any_element.get_local_solution(8);
double c_mu=0.09;
matrix nu_t(k.get_rows(),k.get_columns(),0,"local nu_t");
for (int i=1;i<=k.get_rows();i++)
{
    if (epsilon(i,1)<=1e-10) nu_t(i,1)=0;
    else nu_t(i,1)=c_mu*k(i,1)*k(i,1)/epsilon(i,1);
}
double nu_t_avrg;
nu_t_avrg=0.25*(nu_t(1,1)+nu_t(2,1)+nu_t(3,1)+nu_t(4,1));
double k_avrg, epsilon_avrg;
k_avrg=0.25*(k(1,1)+k(2,1)+k(3,1)+k(4,1));
epsilon_avrg=0.25*(epsilon(1,1)+epsilon(2,1)+epsilon(3,1)+epsilon(4,1));
if (k_avrg<=1e-15) {k_avrg=1;epsilon_avrg=0;}
matrix result;
result=2.0*transpose(J)*SRB*v*SRB*(det_J*nu_t_avrg*epsilon_avrg/k_avrg); //2(dv/dy)^2 term
result=result+transpose(J)*FRB*v*FRB*(det_J*nu_t_avrg*epsilon_avrg/k_avrg); //(dv/dx)^2 term
return result+2.0*transpose(J)*SRB*u*FRB*(det_J*nu_t_avrg*epsilon_avrg/k_avrg); //2(du/dy)(dv/dx) term
}

```

Table A.16: Listing of the PKV_c weak form using the OOFEA toolkit.

```

int main(int argc, char* argv[])
{
    mesh mesh1;

    //ADDITIONAL DECLARATIONS...

    //BEGIN USER INPUT...

    cout << "BEGIN LOADING...";
    mesh1.load_nodal_coordinates(filename[0]);
    mesh1.load_domain_elements(filename[1]);
    mesh1.load_prescribed_elements(1,filename[2]);
    mesh1.load_prescribed_elements(2,filename[3]);
    mesh1.load_prescribed_elements(3,filename[4]);
    mesh1.load_prescribed_elements(4,filename[5]);
    mesh1.load_prescribed_elements(5,filename[6]);
}

```



```

mesh1.load_prescribed_elements(6,filename[7]);
mesh1.load_prescribed_elements(7,filename[38]);
mesh1.load_prescribed_elements(8,filename[39]);
cout << "END\n";

cout << "BEGIN LOADING INITIAL CONDITIONS...";
mesh1.load_global_solution(1,"U",filename[8]);
mesh1.load_global_solution(2,"Ubar",filename[9]);
mesh1.load_global_solution(3,"V",filename[10]);
mesh1.load_global_solution(4,"Vbar",filename[11]);
mesh1.load_global_solution(5,"P",filename[12]);
mesh1.load_global_solution(6,"T",filename[13]);
mesh1.load_global_solution(7,"k",filename[36]);
mesh1.load_global_solution(8,"epsilon",filename[37]);
long rows=mesh1.get_number_of_nodes();
matrix old_u(rows,1),old_ubar(rows,1),old_v(rows,1),old_vbar(rows,1);
matrix old_p(rows,1),old_t(rows,1),old_k(rows,1),old_epsilon(rows,1);
cout << "END\n";

cout << "BEGIN CALCULATING AT ELEMENT LEVEL AND ASSEMBLING...";
sparse_matrix D_DX, D_DY, M, MC, K, P, old_P;
M=mesh1.assemble_global_matrix_from_local_ones(NS2D_capacitance_matrix);
D_DX=mesh1.assemble_global_matrix_from_local_ones(NS2D_d_dx);
D_DY=mesh1.assemble_global_matrix_from_local_ones(NS2D_d_dy);
mesh1.disassemble_global_solution(1);
mesh1.disassemble_global_solution(3);
P=mesh1.assemble_global_matrix_from_local_ones(NS2D_ML_convective_matrix);
MC=condense(M); //Only need to condense M once outside time loop

sparse_matrix EK, G, EKG, PKU, PKV, EKPKU, EKPKV;
matrix nu_t; //Wondimensional eddy viscosity
matrix c_3e; //Epsilon equation coefficient dependent on fluid velocities

matrix u_diff,v_diff,t_diff;
cout << "END\n";

```

Table A.17: Listing of user input and mesh and parameter initialization section using the OOFEA toolkit.

```

cout << "BEGIN SOLVING...";
sparse_matrix A;matrix b;

//ADDITIONAL DECLARATIONS...

for (time_loop=1;time_loop<=steps;time_loop++)
{
    //Here we will have the following solution steps:
    //0. Display the current solution step number
    cout << "Time step number: " << time_loop << endl;

```

```

//0.1. Reference all the necessary matrix components
matrix& u=mesh1.get_global_solution(1);
matrix& ubar=mesh1.get_global_solution(2);
matrix& v=mesh1.get_global_solution(3);
matrix& vbar=mesh1.get_global_solution(4);
matrix& p=mesh1.get_global_solution(5);
matrix& t=mesh1.get_global_solution(6);
matrix& k=mesh1.get_global_solution(7);
matrix& epsilon=mesh1.get_global_solution(8);
/*DEBUG*/ cout << "CALCULATING EDDY VISCOSITY!\n";
if (time_loop==1 && start_option==0) //First step of scratch run only
{
    nu_t.allocate(k.get_rows(),k.get_columns());
    for (int i=1;i<=k.get_rows();i++)
    {
        if (epsilon(i,1)==0) nu_t(i,1)=0;
        else nu_t(i,1)=c_mu*k(i,1)*k(i,1)/epsilon(i,1);
    }
}
if (time_loop>1 || start_option==1) //Continuation run
{
    nu_t.allocate(k.get_rows(),k.get_columns());
    for (int i=1;i<=k.get_rows();i++)
    {
        if (epsilon(i,1)==0) nu_t(i,1)=0;
        else nu_t(i,1)=c_mu*k(i,1)*k(i,1)/epsilon(i,1);
    }
}
/*DEBUG*/ cout << "CALCULATING C_3e!\n";
c_3e=tanh(abs(u->reciprocal(v))); //Divide-by-zero error if v is zero!

//0.2. If we are continuing a previous run, we do not want to
//      perform an Explicit Euler step, but want to directly
//      go to a Semi-Implicit step. To do that, we must calculate
//      old_P using old_u and old_v.

if (time_loop==1 && start_option==1) //This is the first step of a continuation run
{
    matrix temp_u, temp_v;
    temp_u=u;u=old_u;temp_v=v;v=old_v;
    old_P=mesh1.assemble_global_matrix_from_local_ones(WS2D_FL_convective_matrix);
    u=temp_u;v=temp_v;
}

//0.3. Dynamic time step - use an ever increasing fraction of
//      dt=1/Ra*Pr until it is larger than the dt determined
//      based on the fluid velocities, mesh spacing and Courant
//      number desired.

max_u=u.absmax();max_v=v.absmax();

```

```

if (time_loop==1) dt=1.0/(Ra*Pr); //Force dt for Explicit Euler to be small enough
if (max_u<=0 || max_v<=0)
{
    dt=1.1*dt;
}
else
{
    if (flag1==0 && dt<Courant*fabs(average_h/max_v))
    {
        dt=1.1*dt;
        flag1=0;
    }
    else if (flag1==0 && dt>=Courant*fabs(average_h/max_v))
    {
        dt=Courant*fabs(average_h/max_v);
        flag1=1;
    }
    else
    {
        dt=Courant*fabs(average_h/max_v);
        flag1=1;
    }
}
}

```

Table A.18: Listing of the data initialization and time step determination sections of the simulator.

```

//1.0. Disassemble global solution for k and epsilon
//      such that elements can calculate local nu_t
mesh1.disassemble_global_solution(7);
mesh1.disassemble_global_solution(8);

//1.1. PCG solution of ubar at n+1 - intermediate velocity
cout << "\nSOLVING Ubar...";
//Assemble corresponding stiffness matrix including the eddy viscosity
K=mesh1.assemble_global_matrix_from_local_ones(NS2D_momentum_stiffness_matrix);
//Explicit Euler step performed only as the first step of a scratch run
if (time_loop==1 && start_option==0)
    A=MC;
if (time_loop==1 && start_option==0)
    b=(M - P*dt - K*dt)*u;
//Semi-Implicit step performed as the remaining steps of a scratch run
//or also as the first step of a continuation run
if (time_loop>1 || (time_loop==1 && start_option==1))
    A=M+K*dt;
if (time_loop>1 || (time_loop==1 && start_option==1))
    b=M*u - P*u*(1.5*dt) + old_P*old_u*(0.5*dt);
mesh1.apply_Dirichlet_BCs(2,A);
mesh1.apply_Dirichlet_BCs(2,b);
old_ubar=ubar;

```

```

PCG(A, b, ubar, tolerance[0]);
cout << "DONE!\n";

//2. PCG solution of vbar at n+1 - intermediate velocity
cout << "SOLVING Vbar...";
//Explicit Euler step performed only as the first step of a scratch run
if (time_loop==1 && start_option==0)
    A=MC;
if (time_loop==1 && start_option==0)
    b=(M - P*dt - K*dt)*v + M*t*(dt*Ra*Pr);
//Semi-Implicit step performed as the remaining steps of a scratch run
//or also as the first step of a continuation run
if (time_loop>1 || (time_loop==1 && start_option==1))
    A=M+K*dt;
if (time_loop>1 || (time_loop==1 && start_option==1))
    b=M*v - P*v*(1.5*dt) + old_P*old_v*(0.5*dt) + M*t*(dt*Ra*Pr);
mesh1.apply_Dirichlet_BCs(4,A);
mesh1.apply_Dirichlet_BCs(4,b);
old_vbar=vbar;
PCG(A, b, vbar, tolerance[0]);
cout << "DONE!\n";

//4. PCG solution of Poisson equation of p at n+1
cout << "SOLVING P...";
A=mesh1.assemble_global_matrix_from_local_ones(NS2D_stiffness_matrix);
b=-(1/dt)*(D_DX*ubar + D_DY*vbar);
mesh1.apply_Dirichlet_BCs(5,A);
mesh1.apply_Dirichlet_BCs(5,b);
old_p=p;
PCG(A, b, p, tolerance[1]);
cout << "DONE!\n";

//6. PCG solution of u at n+1 - velocity correction
cout << "SOLVING U...";
A=MC;
b=MC*ubar-D_DX*p*dt;
mesh1.apply_Dirichlet_BCs(1,A);
mesh1.apply_Dirichlet_BCs(1,b);
old_u=u;
PCG(A, b, u, tolerance[0]);
cout << "DONE!\n";

//7. PCG solution of v at n+1 - velocity correction
cout << "SOLVING V...";
A=MC;
b=MC*vbar-D_DY*p*dt;
mesh1.apply_Dirichlet_BCs(3,A);
mesh1.apply_Dirichlet_BCs(3,b);
old_v=v;
PCG(A, b, v, tolerance[0]);

```

```

cout << "DONE!\n";

//8. Disassemble global solution for u and v
mesh1.disassemble_global_solution(1);
mesh1.disassemble_global_solution(3);

//9. Recalculate nonlinear convective matrix and assemble
old_P=P;
P=mesh1.assemble_global_matrix_from_local_ones(NS2D_NL_convective_matrix);

//9.1. CG solution of t at n+1 - temperature calculation
cout << "SOLVING T...";
//Assemble corresponding stiffness matrix including the eddy viscosity
K=mesh1.assemble_global_matrix_from_local_ones(NS2D_temperature_stiffness_matrix);
//Explicit Euler step performed only as the first step of a scratch run
if (time_loop==1 && start_option==0)
    A=MC;
if (time_loop==1 && start_option==0)
    b=(M - P*dt - K*dt)*t;
//Semi-Implicit step performed as the remaining steps of a scratch run
//or also as the first step of a continuation run
if (time_loop>1 || (time_loop==1 && start_option==1))
    A=M+K*dt;
if (time_loop>1 || (time_loop==1 && start_option==1))
    b=M*t - P*t*(1.5*dt) + old_P*old_t*(0.5*dt);
mesh1.apply_Dirichlet_BCs(6,A);
mesh1.apply_Dirichlet_BCs(6,b);
old_t=t;
PCG(A, b, t, tolerance[2]);
cout << "DONE!\n";

```

Table A.19: Listing of the momentum, pressure and energy solution steps in the simulator.

```

//9.2. Change Dirichlet BCs for k using element and node iterators
//to access the appropriate boundary conditions based on value nu_t
//and the gradient of the tangential velocity at the wall
matrix dvdx, dudy;dvdx=D_DX*v;dudy=D_DY*u;
cout << "CHANGING k DIRICHLET BOUNDARY CONDITIONS...";
//long count=0;
for(Prescribed_Elements_of_Mesh pelements1(mesh1,7);pelements1.more();pelements1.advance())
{
    double x,y;
    long node_number;
    //Get coordinates of the one node in the element by using a node iterator
    for(Nodes_of_Element enodes(pelements1.current());enodes.more();enodes.advance())
    {
        x=(enodes.current()).get_x_coordinate();
        y=(enodes.current()).get_y_coordinate();
        node_number=(enodes.current()).get_node_number();
    }
}

```

```

//Get nu_t value for the prescribed node
double nu_t_value=nu_t(node_number,1);
//Get velocity gradient value
double dvdx_value=dvdx(node_number,1);
double dudy_value=dudy(node_number,1);
//Logic for determining which is the correct velocity gradient using wall distance
double velocity_gradient;
double wall_distance;
if (x>=0.5 && y>=0.5) wall_distance=min(1.0-x,1.0-y); //Point is close to top right
else if (x>=0.5 && y<0.5) wall_distance=min(y,1.0-x); //Point is close to bottom right
else if (x<0.5 && y<0.5) wall_distance=min(x,y); //Point is close to bottom left
else wall_distance=min(1.0-y,x); //Point is close to top left
if (x==wall_distance) {velocity_gradient=-dvdv_value;} //Left wall
else if (y==wall_distance) {velocity_gradient=-dudy_value;} //Bottom wall
else if ((1.0-x)==wall_distance) {velocity_gradient=dvdv_value;} //Right wall
else {velocity_gradient=dudy_value;} //Top wall
velocity_gradient=fabs(velocity_gradient);
//Use nu_t_value and velocity gradient to calculate k_value
double k_value;
k_value=nu_t_value*Pr*velocity_gradient/pow(c_mu,1.5);
//Check if node is at the wall, and if it is, make k_value zero
if (x==0 || x==1 || y==0 || y==1) k_value=0; //Wall node
//Check if value calculated is negative, and if it is, set it to zero
if (k_value < 0) k_value=0;
//Set the Dirichlet BC for the prescribed element to k_value
(pelements1.current()).set_Dirichlet_BC(k_value);
}
cout << "DONE!\n";

//9.3. Change Dirichlet BCs for epsilon using element and node iterators
//to access the appropriate boundary conditions based on value of k
cout << "CHANGING epsilon DIRICHLET BOUNDARY CONDITIONS...";
for(Prescribed_Elements_of_Mesh pelements2(mesh1,8);pelements2.more();pelements2.advance())
{
double x,y;
long node_number;
//Get coordinates of the one node in the element by using a node iterator
for(Nodes_of_Element enodes(pelements2.current());enodes.more();enodes.advance())
{
x=(enodes.current()).get_x_coordinate();
y=(enodes.current()).get_y_coordinate();
node_number=(enodes.current()).get_node_number();
}
//Get k value for the prescribed node
double k_value=k(node_number,1);
//Get the prescribed node's distance to the wall
double wall_distance;
if (x>=0.5 && y>=0.5) wall_distance=min(1.0-x,1.0-y); //Point is close to top right
else if (x>=0.5 && y<0.5) wall_distance=min(y,1.0-x); //Point is close to bottom right
else if (x<0.5 && y<0.5) wall_distance=min(x,y); //Point is close to bottom left

```

```

else wall_distance=min(1.0-y,x); //Point is close to top left
//Use k_value and wall_distance to calculate epsilon_value
double epsilon_value;
//Check if node is at the wall, and if it is, make k_value zero
if (x==0 || x==1 || y==0 || y==1) epsilon_value=0; //Wall node
else epsilon_value=pow(c_mu,0.75)*pow(k_value,1.5)/(0.42*wall_distance);
//Check if value calculated is negative, and if it is, set it to zero
if (epsilon_value < 0) epsilon_value=0;
//Set the Dirichlet BC for the prescribed element to epsilon_value
(pelements2.current()).set_Dirichlet_BC(epsilon_value);
}
cout << "DONE!\n";

//9.3.5. Disassemble global solution for k and epsilon
//      such that elements can calculate local nu_t
//      as well as the average epsilon and k
mesh1.disassemble_global_solution(7);
mesh1.disassemble_global_solution(8);

//9.4. Recalculate production terms of the k-e turbulence model
cout << "CALCULATING PKU!\n";
PKU=mesh1.assemble_global_matrix_from_local_ones(KE2D_PK_u_matrix);
cout << "CALCULATING PKV!\n";
PKV=mesh1.assemble_global_matrix_from_local_ones(KE2D_PK_v_matrix);
cout << "CALCULATING EKPKU!\n";
EKPKU=mesh1.assemble_global_matrix_from_local_ones(KE2D_EKPK_u_matrix);
cout << "CALCULATING EKPKV!\n";
EKPKV=mesh1.assemble_global_matrix_from_local_ones(KE2D_EKPK_v_matrix);

//9.5. Recalculate the nonlinear terms in the k-e turbulence model
cout << "CALCULATING EK!\n";
EK=mesh1.assemble_global_matrix_from_local_ones(KE2D_EK_matrix);
cout << "CALCULATING G!\n";
G=mesh1.assemble_global_matrix_from_local_ones(KE2D_G_matrix);
cout << "CALCULATING EKG!\n";
EKG=mesh1.assemble_global_matrix_from_local_ones(KE2D_EKG_matrix);

//9.6. CG solution of k at n+1 - turbulent kinetic energy calculation
cout << "SOLVING k...";
//Assemble corresponding stiffness matrix including the eddy viscosity
cout << "CALCULATING k STIFFNESS MATRIX!\n";
K=mesh1.assemble_global_matrix_from_local_ones(NS2D_k_stiffness_matrix);
//Explicit Euler step performed only as the first step of a scratch run
if (time_loop==1 && start_option==0)
    A=MC;
if (time_loop==1 && start_option==0)
    b=(M - P*dt - K*dt - EK*dt)*k - G*t*(dt*Ra*Pr/sigma_t) + PKU*u*dt + PKV*v*dt;
//Semi-Implicit step performed as the remaining steps of a scratch run
//or also as the first step of a continuation run
if (time_loop>1 || (time_loop==1 && start_option==1))

```

```

    A=M+K*dt+EK*dt;
    if (time_loop>1 || (time_loop==1 && start_option==1))
        b=M*k - P*k*(1.5*dt) + old_P*old_k*(0.5*dt) - G*t*(dt*Ra*Pr/sigma_t)
          + PKU*u*dt + PKV*v*dt;
    mesh1.apply_Dirichlet_BCs(7,A);
    mesh1.apply_Dirichlet_BCs(7,b);
    old_k=k;
    PCG(A, b, k, tolerance[2]);
    cout << "DONE!\n";

//9.7. CG solution of epsilon at n+1 - turbulent dissipation calculation
cout << "SOLVING epsilon...";
//Assemble corresponding stiffness matrix including the eddy viscosity
K=mesh1.assemble_global_matrix_from_local_ones(NS2D_epsilon_stiffness_matrix);
//Explicit Euler step performed only as the first step of a scratch run
if (time_loop==1 && start_option==0)
    A=MC;
if (time_loop==1 && start_option==0)
    b=(M - P*dt - K*dt - EK*(c_2e*dt))*epsilon - EKG*t*(c_1e*dt*Ra*Pr/sigma_t)
      + EKPKU*u*(c_1e*dt) + EKPKV*v*(c_1e*dt);
//Semi-Implicit step performed as the remaining steps of a scratch run
//or also as the first step of a continuation run
if (time_loop>1 || (time_loop==1 && start_option==1))
    A=M+K*dt+EK*(c_2e*dt);
if (time_loop>1 || (time_loop==1 && start_option==1))
    b=M*epsilon - P*epsilon*(1.5*dt) + old_P*old_epsilon*(0.5*dt)
      - EKG*t*(c_1e*dt*Ra*Pr/sigma_t) + EKPKU*u*(c_1e*dt) + EKPKV*v*(c_1e*dt);
    mesh1.apply_Dirichlet_BCs(8,A);
    mesh1.apply_Dirichlet_BCs(8,b);
    old_epsilon=epsilon;
    PCG(A, b, epsilon, tolerance[2]);
    cout << "DONE!\n";

//9.8. Zero value of k and epsilon which are negative (non-physical)
for (long i=1;i<=mesh1.get_number_of_nodes();i++)
{
    if (k(i,1)<0) k(i,1)=0.0; //Turbulent kinetic energy should always be positive
    if (epsilon(i,1)<0) epsilon(i,1)=0.0; //Dissipation should also always be positive
}

```

Table A.20: Listing of the k - ϵ solution steps of the simulator including the determination and update of boundary conditions.

```

//10. Save if appropriate
if (time_loop % save_step == 0)
{
    cout << "BEGIN SAVING...";
    mesh1.save_global_solution(1,"U",filename[14]);
    mesh1.save_global_solution(2,"Ubar",filename[15]);
    mesh1.save_global_solution(3,"V",filename[16]);
}

```



```

    mesh1.save_global_solution(4,"Vbar",filename[17]);
    mesh1.save_global_solution(5,"P",filename[18]);
    mesh1.save_global_solution(6,"T",filename[19]);
    mesh1.save_global_solution(7,"k",filename[44]);
    mesh1.save_global_solution(8,"epsilon",filename[45]);
    old_u.save(filename[23]);
    old_u_bar.save(filename[24]);
    old_v.save(filename[25]);
    old_v_bar.save(filename[26]);
    old_p.save(filename[27]);
    old_t.save(filename[28]);
    old_k.save(filename[46]);
    old_epsilon.save(filename[47]);
    cout << "END\n";
    cout << "BEGIN POST-PROCESSING...\n";
    matrix& u=mesh1.get_global_solution(1);
    matrix& v=mesh1.get_global_solution(3);
    matrix& t=mesh1.get_global_solution(6);
    cout << "SOLVING STREAM FUNCTION...";
    matrix Psi;Psi=p; //Use the same data as that for the pressure
    A=K;b=D_DY*u- D_DX*v;
    mesh1.apply_Dirichlet_BCs(5,A);
    mesh1.apply_Dirichlet_BCs(5,b);
    PCG(A, b, Psi, tolerance[0]);
    Psi.save(filename[20]);
    cout << "DONE!\n";
    cout << "SOLVING VORTICITY...";
    matrix Vorticity;Vorticity=D_DX*v-D_DY*u;
    Vorticity.save(filename[21]);
    cout << "END\n";
    cout << "SOLVING NUSSELT NUMBER...";
    matrix dt_dx;dt_dx=D_DX*t*(-1.0);
    dt_dx.save(filename[22]);
    cout << "DONE!\n";
    cout << "END\n";
}

//11. Check for convergence
u_diff=u-old_u;v_diff=v-old_v;t_diff=t-old_t;
if (u_diff.absmax()<1e-6 && v_diff.absmax()<1e-6 && t_diff.absmax()<1e-6)    time_loop+=steps+10;

//12. Loop back to step 0
}
cout << "END\n";

cout << "BEGIN SAVING...";
mesh1.save_global_solution(1,"U",filename[14]);
mesh1.save_global_solution(2,"Ubar",filename[15]);
mesh1.save_global_solution(3,"V",filename[16]);
mesh1.save_global_solution(4,"Vbar",filename[17]);

```

```

mesh1.save_global_solution(5,"P",filename[18]);
mesh1.save_global_solution(6,"T",filename[19]);
mesh1.save_global_solution(7,"k",filename[44]);
mesh1.save_global_solution(8,"epsilon",filename[45]);
old_u.save(filename[23]);
old_ubar.save(filename[24]);
old_v.save(filename[25]);
old_vbar.save(filename[26]);
old_p.save(filename[27]);
old_t.save(filename[28]);
old_k.save(filename[46]);
old_epsilon.save(filename[47]);
cout << "END\n";

cout << "BEGIN POST-PROCESSING...\n";
matrix& u=mesh1.get_global_solution(1);
matrix& v=mesh1.get_global_solution(3);
matrix& t=mesh1.get_global_solution(6);
cout << "SOLVING STREAM FUNCTION...";
matrix& Psi=mesh1.get_global_solution(5); //Use the same data as that for the pressure
mesh1.load_global_solution(5,"P",filename[12]);
A=K;
b=D_DY*u- D_DX*v;
mesh1.apply_Dirichlet_BCs(5,A);
mesh1.apply_Dirichlet_BCs(5,b);
PCG(A, b, Psi, tolerance[0]);
Psi.save(filename[20]);
cout << "DONE!\n";
cout << "SOLVING VORTICITY...";
matrix Vorticity;Vorticity=D_DX*v-D_DY*u;
Vorticity.save(filename[21]);
cout << "DONE!\n";
cout << "SOLVING NUSSELT NUMBER...";
matrix dt_dx;dt_dx=D_DX*t*(-1.0);
dt_dx.save(filename[22]);
cout << "DONE!\n";
cout << "SAVING PROGRAM STATISTICS...";
ofstream output;
output.open(filename[35]);
output.setf(ios::scientific | ios::showpos);
output.precision(20);
if (time_loop>=steps+10) time_loop-=steps+1;
output << "Number of time steps: " << time_loop-1 << endl;
output << "Elapsed time: " << difftime(end_clock,start_clock)*1e-3 << " sec\n";
output << "Physical time: " << physical_time << endl;
output << "Maximum u velocity: " << max_u << endl;
output << "Maximum v velocity: " << max_v << endl;
output << "Maximum stream function: " << Psi.absmax() << endl;
output.unsetf(ios::scientific | ios::showpos);
output.close();

```

```

//Calculate nu_t
matrix& k=mesh1.get_global_solution(7);
matrix& epsilon=mesh1.get_global_solution(8);
for (int i=1;i<=k.get_rows();i++)
{
    if (epsilon(i,1)==0) nu_t(i,1)=0;
    else nu_t(i,1)=c_mu*k(i,1)*k(i,1)/epsilon(i,1);
}
nu_t.save("d:\\users\\rafa\\c-code\\data\\nuttst3.dat");
cout << "DONE!\n";
cout << "END\n";

//END PROGRAM
return 0;
}

```

Table A.21: Listing of the intermediate and final post-processing sections of the simulator.

Bibliography

- [1] Schildt, H., *Turbo C/C++: The Complete Reference*, Osborne McGraw-Hill, 1990
- [2] Barton, J. J. and Nackman, L. R., *Scientific and Engineering C++: an Introduction with Advanced Techniques and Examples*, Addison-Wesley Publishing Company, Inc., 1994
- [3] Raphael, B. and Krishnammorthy, C. S., "Automating Finite Element Development Using Object-Oriented Techniques," *Engineering Computations*, Vol. 10, No. 3, pp. 267-278, 1993
- [4] Kong, X. A. and Chen, D. P., "An Object-Oriented Design of FEM Programs," *Computers and Structures*, Vol. 57, No. 1, pp. 157-166, 1995
- [5] Zimmermann, T., Dubois-Pèlerin, Y. and Bomme, P., "Object-Oriented Finite Element Programming: I. Governing Principles," *Computer Methods in Applied Mechanics and Engineering*, Vol. 98, No. 2, pp. 291-303, 1992
- [6] Mackie, R. I., "Object-Oriented Programming of the Finite Element Method," *International Journal for Numerical Methods in Engineering*, Vol. 35, No. 2, pp. 425-436, 1992
- [7] Forde, B. W. R., Foschi, R. O. and Stiemer, S. F., "Object-Oriented Finite Element Analysis," *Computers and Structures*, Vol. 34, No. 3, pp. 355-374, 1990

- [8] Scholz, S. P., "Elements of an Object-Oriented FEM++ Program in C++." *Computers and structures*, Vol. 47, No. 3, pp. 517-529, 1992
- [9] Lu, J., White, D. W., Chen, W. and Dunsmore, H. E., "A Matrix Class Library in C++ for Structural Engineering Computing," *Computers and Structures*, Vol. 55, No. 1, pp. 95-111, 1995
- [10] Zeglinski, G. W., Han, R. P. S. and Aitchison, P., "Object-Oriented Matrix Classes for Use in a Finite Element Code Using C++," *International Journal for Numerical Methods in Engineering*, Vol. 37, No. 22, pp. 3921-3937, 1994
- [11] Menétrey, P. and Zimmermann, T., "Object-Oriented Non-Linear Finite Element Analysis: Application to J2 Plasticity," *Computers and Structures*, Vol. 49, No. 5, pp. 767-777, 1993
- [12] Yu, G. and Adell, H., "Object-Oriented Finite Element Analysis Using EER Model," *Structural Engineering*, Vol. 119, No. 9, pp. 2763-2781, 1993
- [13] Miki, M. and Murotsu, Y., "Object-Oriented Approach to Modeling and Analysis of Truss Structures," *AIAA Journal*, Vol. 33, No. 2, pp. 348-353, 1995
- [14] Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., romine, C. and van der Vorst, H., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Society for Industrial and Applied Mathematics, Philadelphia, 1994
- [15] Gebhart, B., Jaluria, Y., Mahajan, R. L. and Sammakia, B., *Buoyancy-Induced Flows and Transport: Reference Edition*, Hemisphere Publishing Corporation, 1988

- [16] Aung, W. and Yener, Y., "Research Directions in Natural Convection," in *Natural convection: Fundamentals and Applications*, eds. Kakaç, S., Aung, W. and Viskanta, R., Hemisphere Publishing Corporation, pp. 1155-1171, 1985
- [17] Kakaç, S., Aung, W. and Viskanta, R., eds., "Preface," *Natural convection: Fundamentals and Applications*, Hemisphere Publishing Corporation, 1985
- [18] Moreno, R., "Numerical Study of Three Dimensional Incompressible Flows in Complex Geometries," Master's thesis, Rice University, Houston, Texas, 1994
- [19] Selamet, E., "Simulation of Laminar Buoyancy-Driven Flows in an Enclosure." Doctoral dissertation, University of Michigan, 1991
- [20] Evren-Selamet, E., Arpaci, V. S. and Borgnakke, C., "Simulation of Laminar Buoyancy-Driven Flows in an Enclosure," *Numerical Heat Transfer, Part A*, Vol. 22, pp. 401-420, 1992
- [21] Jakob, M., *Heat Transfer*, Wiley, New York, 1949
- [22] Batchelor, G. K., "Heat Transfer by Free Convection across a Closed Cavity between Vertical Boundaries at Different Temperatures," *Q. Appl. Math.*, Vol. 12, pp. 209-233, 1954
- [23] Lewis, J. A., "Free Convection in Commercial Insulating Materials." Doctoral dissertation, Graduate Division of Applied Mathematics, Brown University, Providence, Rhode Island, 1950
- [24] Ostrach, S., "Laminar Natural Convection Flow and Heat Transfer of Fluids with and without Heat Sources in Channels with Constant Wall Temperatures," NACA TN 2863, 1952

- [25] Eckert, E. R. G. and Carlson, W. O., "Natural Convection in an Air Layer Enclosed between Two Vertical Plates with Different Temperatures," *International Journal of Heat and Mass Transfer*, Vol. 2, pp. 106, 1961
- [26] Elder, J. W., "Laminar Free Convection in a Vertical Slot," *Journal of Fluid Mechanics*, Vol. 23, pp. 77, 1965
- [27] Wilkes, J. O., "The Finite Difference Computation of Natural Convection in an Enclosed Rectangular Cavity," Doctoral dissertation, University of Michigan, Ann Arbor, 1963
- [28] de Vahl Davis, G., "Laminar Natural Convection in an Enclosed Rectangular Cavity," *International Journal of Heat and Mass Transfer*, Vol. 11, pp. 1675, 1968
- [29] de Vahl Davis, G. and Mallinson, G. D., "A Note on Natural convection in a Vertical slot," *Journal of Fluid Mechanics*, Vol. 72, pp. 87, 1975
- [30] de Vahl Davis, G., "Natural Convection of Air in a Square Cavity: a Bench Mark Numerical Solution," *International Journal for Numerical Methods in Fluids*, Vol. 3, pp. 249-264, 1983
- [31] Saitoh, T. and Hirose, K., "High-Accuracy Bench Mark Solutions to Natural Convection in a Square Cavity," *Computational Mechanics*, Vol. 4, No. 2, pp. 417-427, 1989
- [32] Mallinson, G. D. and de Vahl Davis, G., "Three-Dimensional Natural convection in a Box: a Numerical Study," *Journal of Fluid Mechanics*, Vol. 83, pp. 1-31, 1977

- [33] Freitas, C. J., Street, R. L., Findikakis, A. N. and Koseff, J. R., "Numerical Simulation of Three-Dimensional Flow in a Cavity," *International Journal of Numerical Methods in Fluids*, Vol. 5, pp. 561-575, 1985
- [34] Pepper, D. W., "Modeling of Three-Dimensional Natural Convection with a Time-Split Finite-Element Technique," *Numerical Heat Transfer*, Vol. 11, No. 1, pp. 31-55, 1987
- [35] Fusegi, T., Hyun, J. M., Kuwahara, K. and Farouk, B., "A Numerical Study of Three-Dimensional Natural Convection in a Differentially Heated Cubical Enclosure," *International Journal of Heat and Mass Transfer*, Vol. 34, No. 6, pp. 1543-1557, 1991
- [36] Shaw, H., "Laminar Mixed Convection Heat Transfer in Three-Dimensional Horizontal Channel with a Heated Bottom," *Numerical Heat Transfer, Part A*, Vol. 23, pp. 445-461, 1993
- [37] Bradshaw, P., *An Introduction to Turbulence and its Measurement*, Pergamon Press, 1971
- [38] Hinze, J. O., *Turbulence*, McGraw Hill Book Co., 1975
- [39] Paolucci, S., "Direct Numerical Simulation of Two-Dimensional Turbulent Natural Convection in an Enclosed Cavity," *Journal of Fluid Mechanics*, Vol. 215, pp. 229-262, 1990
- [40] Reynolds, O., "On the Dynamical theory of Incompressible Viscous Fluids and the Determination of the Criterion," *Philosophical Transactions of the Royal Society of London*, Series A, Vol. 186, pp. 123, 1895

- [41] Wilcox, D. C., *Turbulence Modeling for CFD*, DCW Industries, Inc., 1993
- [42] Tennekes, H. and Lumley, J. L., *A First Course in Turbulence*, The MIT Press, 1972
- [43] Mohamad, A. A. and Viskanta, R., "Modeling of Turbulent Buoyant Flow and Heat Transfer in Liquid Metals," *International Journal of Heat and Mass Transfer*, Vol. 36, No.11, pp. 2815-2826, 1993
- [44] Hoffmann, K. A., *Computational Fluid Dynamics for Engineers*, Engineering Education System, Austin, Texas, 1989
- [45] Launder, B. E. and Spalding, D. B., *Lectures in Mathematical Models of Turbulence*, Academic Press, London and New York, 1972
- [46] Launder, B. E. and Spalding, D.B., "The Numerical Computation of Turbulent Flows," *Computer Methods in Applied Mechanics and Engineering*, Vol. 3, pp. 269-289, 1974
- [47] Ozoe, H., Mouri, A., Ohmuro, M., Churchill, S. W. and Lior, N., "Numerical Calculations of Laminar and Turbulent Natural Convection in Water in Rectangular Channels Heated and Cooled Isothermally on the Opposing Vertical Walls," *International Journal of Heat and Mass Transfer*, Vol. 28, No. 1, pp. 125-138, 1985
- [48] Ozoe, H., Mouri, A., Hiramitsu, M., Churchill, S. W. and Lior, N., "Numerical Calculations of Three-Dimensional Turbulent Natural Convection in a Cubical Enclosure Using a Two-Equation Model for Turbulence," *Journal of Heat Transfer*, Vol. 108, pp. 806-813, 1986

- [49] Fraikin, M. P., Portier, J. J. and Fraikin, C. J., "Application of a k - ϵ Turbulence Model to an Enclosed Buoyancy Driven Recirculating Flow," *19th ASME-AIChE National Heat Transfer Conference* paper, 80-HT-68, 1980
- [50] Farouk, B. and Güçeri, S. I., "Laminar and Turbulent Natural Convection in the Annulus between Horizontal Concentric Cylinders," *Journal of Heat Transfer*, Vol. 104, pp. 631-636, 1982
- [51] Markatos, N. C. and Pericleous, K. A., "Laminar and Turbulent Natural Convection in an Enclosed Cavity," *International Journal of Heat and Mass Transfer*, Vol. 27, No. 5, pp. 755-772, 1984
- [52] Güçeri, S. I. and Farouk, B., "Numerical solutions in Laminar and Turbulent Natural Convection," in *Natural convection: Fundamentals and Applications*, eds. Kakaç, S., Aung, W. and Viskanta, R., Hemisphere Publishing Corporation, 1985, pp. 615-654
- [53] Staehle B. and Hahne, E., "Overshooting and Damped Oscillations of Transient Natural Convection in Cavities," *Proceedings of the Seventh International Heat Transfer Conference*, Vol. 2, pp. 287-292, 1982
- [54] Barakos, G. and Mitsoulis, E., "Natural Convection Flow in a Square Cavity Revisited: Laminar and Turbulent Models with Wall Functions," *International Journal for Numerical Methods in Fluids*, Vol. 18, pp. 695-719, 1994
- [55] Henkes, R. A. W. M. and Hoogendoorn, C. J., "comparison Exercise for Computations of Turbulent Natural convection in Enclosures," *Numerical Heat Transfer. Part B*, Vol. 28, pp. 59-78, 1995

- [56] Pérez-Segarra, C. D., Oliva, A., Costa M. and Escanes, F., "Numerical Experiments in Turbulent Natural and Mixed Convection in Internal Flows," *International Journal of Numerical Methods in Heat and Fluid Flow*, Vol. 5, pp. 13-33, 1995
- [57] Yang, R. J. and Aung, W., "Equations and Coefficients for Turbulence Modeling," in *Natural convection: Fundamentals and Applications*, eds. Kakaç, S., Aung, W. and Viskanta, R., Hemisphere Publishing Corporation, 1985, pp. 259-300
- [58] Jaluria, Y., *Natural convection: Heat and Mass Transfer*, HMT, the Science and Applications of Heat and Mass Transfer: Reports, Reviews and Computer Programs, Pergamon Press, Vol. 5, 1980
- [59] Ramaswamy, B., "Finite Element Solution for Advection and Natural Convection Flows," *Computers & Fluids*, Vol. 16, No. 4, pp. 349-388, 1988
- [60] Ho, M. C. and Ramaswamy, B., "Use of Baffles in Cavity Flow for the Suppression of Thermal Oscillations Under Microgravity," *International Journal of Numerical Methods in Heat and Fluid Flow*, Vol. 5, pp. 141-173, 1994
- [61] Chippada, S., "Numerical Study of Thin-Film Flows and Open-Channel Flows," Doctoral dissertation, Rice University, Houston, Texas, 1995
- [62] Jue, T. C., "Numerical Study of Cavity Natural Convection Flow with Augmenting and Counteracting Effects by Projection Finite Element Method." Doctoral dissertation, Rice University, Houston, Texas, 1992

- [63] Ramaswamy, B., Jue, T. C. and Akin, J. E., "Semi-Implicit and Explicit Finite Element Schemes for Coupled Fluid/Thermal Problems," *International Journal for Numerical Methods in Engineering*, Vol. 34, pp. 675-696, 1992
- [64] Akin, J. E., *Finite Elements for Analysis and Design*, packet for MECH 517:Finite Element Methods, Rice University, Houston, Texas, 1992
- [65] Baker, A. J. and Oden, J. T., *A Primer on Finite Element Analysis*, for the Annual Shortcourse on Finite Element Analysis in Fluid Mechanics and Heat Transfer, University of Texas, Texas Institute for Computational and Applied Mathematics, 1994
- [66] Computational Mechanics Corporation, *AKCESS.1-2-3: A Tutorial for Finite Element Modeling and Rapid Implementation*, Knoxville, Tennessee, 37919-3382, U.S.A.