# SIMULACIÓN NUMÉRICA DEL CAMPO DE FLUJO DE UNA CAVIDAD RECTANGULAR BAJO CONDICIONES DE CONVECCIÓN NATURAL MEDIANTE VOLÚMENES FINITOS

**HENDER ELÍAS MORALES MUEGUES**

**UNIVERSIDAD INDUSTRIAL DE SANTANDER**
**FACULTAD DE INGENIERÍAS FÍSICOMECÁNICAS**
**ESCUELA DE INGENIERÍA MECÁNICA**
**BUCARAMANGA**
**2015**

# SIMULACIÓN NUMÉRICA DEL CAMPO DE FLUJO DE UNA CAVIDAD RECTANGULAR BAJO CONDICIONES DE CONVECCIÓN NATURAL MEDIANTE VOLÚMENES FINITOS

**HENDER ELÍAS MORALES MUEGUES**

**Trabajo de grado para optar el título de**
**Ingeniero Mecánico**

**Director:**
**DAVID ALFREDO FUENTES DÍAZ**
**Ingeniero Mecánico, Ph.D**

**UNIVERSIDAD INDUSTRIAL DE SANTANDER**
**FACULTAD DE INGENIERÍAS FÍSICOMECÁNICAS**
**ESCUELA DE INGENIERÍA MECÁNICA**
**BUCARAMANGA**
**2015**

**CONTENIDO.**

pág**.**

INTRODUCCIÓN. .....15

1. JUSTIFICACIÓN PARA SOLUCIONAR EL PROBLEMA. .....16

2. OBJETIVOS DEL TRABAJO DE GRADO. .....17

2.1. OBJETIVO GENERAL. .....17

2.2. OBJETIVOS ESPECÍFICOS. .....17

3. CONVECCIÓN NATURAL. .....18

3.1. CONVECCIÓN NATURAL EN CAVIDADES RECTANGULARES. .....18

3.1.1 Estudios previos. .....19

3.1.2 Modelamiento matemático. .....21

3.1.3 Modelo matemático de una cavidad rectangular bajo condiciones de convección natural. .....24

3.2. FORMA ADIMENSIONAL DE LAS ECUACIONES DE CONSERVACIÓN. .....26

3.3. CLASIFICACIÓN MATEMÁTICA DE LOS FLUJOS. .....28

3.4. CONDICIONES INICIALES Y DE FRONTERA. .....29

4. METODOS DE VOLUMENES FINITOS. .....31

4.1. CONCEPTOS GENERALES DEL MÉTODO DE LOS VOLÚMENES FINITOS. .....31

4.2. PASOS GENERALES DEL MVF. .....32

4.2.1 Modelamiento matemático. .....33

4.2.2 Mallado espacial .....33

4.2.3 Discretización de las ecuaciones .....34

4.2.4 Solución del sistema de ecuaciones .....34

4.2.5 Post-proceso. .....36

4.3. PROPIEDADES DE LOS METODOS DE SOLUCIÓN NUMERICA. .....36

4.3.1 Precisión. .....36

4.3.2 Consistencia. .....36

4.3.4 Estabilidad. .....37

4.4.5 Convergencia .....37

5. DIFUSIÓN DE CALOR .....38

5.1. DISCRETIZACIÓN. .....38

5.1.1 Condiciones de frontera. .....41

5.2. IMPLEMENTACIÓN DEL CODIGO COMPUTACIONAL. .....43

5.3. CASO DE ESTUDIO: DIFUSION DE CALOR ESTACIONARIA EN PLACA.

.....45

5.3.1 Solución numérica .....46

5.3.2 Validación para validar la solución numérica .....47

5.3.3 Análisis de resultados. .....50

6. DIFUSIÓN-CONVECCIÓN DE CALOR. .....51

6.1. DISCRETIZACIÓN. ..... 51

6.1.1 Condiciones de frontera ..... 54

6.2. IMPLEMENTACIÓN DEL CODIGO COMPUTACIONAL. ..... 55

6.3. CASO DE ESTUDIO: SMITH HUTTON TEST. ..... 56

6.3.1 Solución numérica ..... 57

6.3.2 Validación ..... 58

6.3.3 Análisis de resultados. ..... 62

7. SOLUCIÓN DEL CAMPO DE FLUJO. ..... 63

7.1. MALLADO COLOCADO. ..... 64

7.2. DISCRETIZACIÓN. ..... 65

7.2.1. Problema chekerboarding ..... 66

7.3. MÉTODO SIMPLE. ..... 69

7.3.1 Secuencia algoritmo simple. ..... 69

7.4. IMPLEMENTACIÓN DEL CÓDIGO COMPUTACIONAL. ..... 70

7.5. CASO DE ESTUDIO: LID DRIVEN CAVITY. ..... 70

7.5.1 Solución numérica ..... 71

7.5.2 Validación ..... 72

7.5.3 Análisis de resultados ..... 74

8. CAVIDAD RECTANGULAR BAJO CONDICIONES DE CONVECCION NATURAL. ..... 75

8.1. IMPLEMENTACIÓN DEL CÓDIGO COMPUTACIONAL. ..... 75

8.2. CASO DE ESTUDIO: BOUYANCY DRIVEN CAVITY.....77

8.2.1 Solución numérica.....79

8.2.2 Validación. .....81

8.2.3 Análisis de resultados .....88

9. CONCLUSIONES. .....90

10. RECOMENDACIONES. .....92

BIBLIOGRAFÍA. .....93

ANEXOS. .....96

## LISTA DE TABLAS.

pág

Tabla 1. Coeficientes. .....23
Tabla 2. Propiedades del acero. .....46
Tabla 3. Matriz de temperatura analítica. .....48
Tabla 4. Matriz distribución de temperaturas numérico 12x12. .....49
Tabla 5. Matriz distribución de temperaturas numérico 120x120. .....49
Tabla 6. Matriz de porcentaje de error numérico 12x12 vs analítico. .....50
Tabla 7. Matriz de porcentaje de error numérico 120x120 vs analítico. .....50
Tabla 8.Funciones A(|Pe|) para distintos esquemas. .....53
Tabla 9. Valores de la variable ϕ en la frontera de salida ρ/Γ=10. .....59
Tabla 10. Valores de la variable ϕ en la frontera de salida ρ/Γ=1000. .....60
Tabla 11. Valores de la variable ϕ en la frontera de salida ρ/Γ=1' 000.000. .....61
Tabla 12. Errores relativos Re=100. .....72
Tabla 13. Errores relativos Re=400. .....73
Tabla 14. Errores relativos Re=1000. .....73
Tabla 15. Propiedades del aire a 300 K. .....77
Tabla 16. Ángulos críticos para distintos factores de forma H/L. .....78
Tabla 17. Nusselt promedio para distintas densidades de malla, Ra=1e6. .....83
Tabla 18. Nusselt promedio. .....84
Tabla 19. Errores distintos factores de forma .....86
Tabla 20. Errores H/L=1 .....88

## LISTA DE FIGURAS.

pág

Figura 1. Cavidad rectangular.....19
Figura 2. Modelo matemático.....24
Figura 3. Pasos generales MVF.....32
Figura 4. Tipos de mallas.....33
Figura 5. Plan de trabajo escalonado. .....37
Figura 6. Volumen de control bidimensional. .....38
Figura 7. Condición de frontera tipo Dirichlet. .....42
Figura 8. Condición de frontera tipo Newmann. .....43
Figura 9. Funciones principales de la clase CDifusion.....44
Figura 10. Placa de acero. .....45
Figura 11. Distribución de temperaturas malla 12x12. .....46
Figura 12. Distribución de temperaturas malla 120x120. .....47
Figura 13. Distribución de nodos. .....47
Figura 14. Solución analítica. .....48
Figura 15. Clases principales de la clase CD. .....55
Figura 16. Problema de Smith-Hutton. .....57
Figura 17.Independencia de la malla, esquema Upwind y ρ/Γ=10. .....58
Figura 18. Variación de la variable ϕ en frontera de salida para ρ/Γ=10. .....59
Figura 19. Variación de la variable ϕ en frontera de salida para ρ/Γ=1000. .....60
Figura 20. Variación de la variable ϕ en frontera de salida para ρ/Γ=1' 000.000. .....61
Figura 21. Concentración de la variable ϕ.....62
Figura 22. Mallado colocado. .....65
Figura 23. Campo de presiones tipo chekerboard. .....67
Figura 24. Caso estándar lid driven cavity. .....70
Figura 25. Líneas de corriente simulación. .....71
Figura 26. Solución numérica Re=100. .....72
Figura 27. Solución numérica Re=400. .....73

Figura 28. Solución numérica Re=1000. ..... 73
Figura 29. Funciones principales de la clase BDC. ..... 76
Figura 30. Cavidad bajo convección natural. ..... 79
Figura 31. Campos de temperatura para distintos valores de Rayleigh, a) $Ra = 103$, b) $Ra = 104$ , c) $Ra = 105$ y d) $Ra = 106$ ..... 80
Figura 32. Patrones de convección natural simulado por MVF, $103 < Ra < 106$, a) líneas de corriente, b) iso-velocidad U, c) iso-velocidad V, d) líneas isotérmicas. ..... 81
Figura 33. Campos de temperatura. a) 30x30 vc b) 60x60 vc c) 120x120 vc. ..82
Figura 34. Variación de Nusselt local para distintas densidades de mallas, Ra=1e6. ..... 82
Figura 35. Campo de Nusselt local, a) Ra=1e3, b) Ra=1e4, c) Ra=1e5 y d) Ra=1e6. ..... 83
Figura 36. Nusselt local a lo largo de la frontera caliente para distintos valores de Rayleigh, MVF 120x120 volúmenes. ..... 84
Figura 37. Campo de temperaturas para distintos factores de forma H/L, Ra=1e4, Pr=0.71, a) H/L=2.5 b) H/L=5 c) H/L=10. ..... 85
Figura 38. Nusselt local en la pared caliente, Ra=1e4, Pr=0.71. ..... 85
Figura 39. Nusselt promedio numérico vs empírico, Ra=1e4, Pr=0.71. ..... 86
Figura 40. Campos de Nusselt para distintos ángulos, H/L=1, Ra=1e5, Pr=0.71, a) 30°, b) 40°, c) 45°, d) 60° y e) 90°. ..... 87
Figura 41. Nusselt local en la pared caliente para distintos ángulos, H/L=1, Ra=1e5, Pr=0.71 ..... 87
Figura 42. Comparación Nusselt promedio correlación vs simulación. ..... 88

## LISTA DE ANEXOS.

pág.

ANEXO A. PATRONES DE CONVECCION NATURAL……..…97
ANEXO B. CODIGO COMPUTACIONAL DESARROLLADO……..101

## RESUMEN

**TÍTULO:** "SIMULACIÓN NUMÉRICA DEL CAMPO DE FLUJO DE UNA CAVIDAD RECTANGULAR BAJO CONDICIONES DE CONVECCIÓN NATURAL MEDIANTE VOLÚMENES FINITOS" *

**AUTOR:** HENDER ELÍAS MORALES MUEGUES **

**PALABRAS CLAVE:** Volúmenes finitos, Convección natural en cavidades rectangulares, CFD, C++.

**DESCRIPCIÓN:**

El siguiente documento presenta una guía metodológica para la implementación de un programa que permite resolver el campo de flujo para una cavidad rectangular bajo condiciones de convección, mediante el método de los volúmenes finitos, para un flujo laminar y geometrías sencillas que pueda ser resultas mediante un mallado estructurado cartesiano, el fluido simulado fue aire con un numero de Prantld de 0.71 y valores de numero de Rayleigh de $10^3 < Ra < 10^6$.

El programa fue implementado en el lenguaje C++, creándose 3 librerías auxiliares que permiten resolver las ecuaciones de difusión, convección-difusión y Navier-Stokes de forma general. Estas librerías fueron validadas con soluciones analíticas, numéricas y experimentales para determinar su correcto funcionamiento.

Las simulaciones del campo de flujo de la cavidad rectangular fueron desarrolladas con un mallado colocado, utilizando un esquema de interpolación para las velocidades de Rhie-Chow y además se usó un esquema potencial para discretización del término convectivo. Para validar la solución numérica se compararó el Nusselt promedio para cada caso, respecto a la solución numérica de De Vhals Davis y las correlaciones experimentales de Catton, además se realizó un estudio de casos para distintos factores de forma de la cavidad y para distintos ángulos de inclinación los resultados obtenidos se compararon con correlaciones empíricas.

El programa desarrollado viene implementado con distintos esquemas de interpolación del término convectivo (diferencias centradas, upwind, hibrido, potencial y exponencial) todo esto con el fin de obtener una compresión más profunda acerca de la influencia de cada esquema de interpolación en la solución.


* Trabajo de grado

** Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingeniería Mecánica. Director: David Alfredo Fuentes Díaz

## ABSTRACT

**TITLE:** *"NUMERICAL SIMULATION OF NATURAL CONVECTION IN RECTANGULAR ENCLOUSURES BY FINITE VOLUME METHOD"*[*]

**AUTHOR:** HENDER ELÍAS MORALES MUEGUES[**]

**KEYWORDS:** Finite volumen method, Natural convection in rectangular enclousures, CFD, C++.

**DESCRIPTION:**

This document presents the metological procedure to implement a program that resolves the flow field of a rectangular enclousures under condition of natural convetion by finite volume method for a cartesian structured mesh. The simulated fluid was air with a Prandlt number of 0.71 and were simulated diferent case to values of Rayleigh number $10^3 < Ra < 10^6$

The program was implemented in the C ++ language, also were implemented 3 auxiliary libraries that solves the diffusion equations, convection-diffusion equations and Navier-Stokes equations. All auxiliary libraries were validated with para analytical, numerical and experimental solutions to determine a correct operation.

The simulations of the flow field in the rectangular enclosures were developed with Rhie-Chow interpolation scheme for velocities, and also was used to discretize the convective term the potential scheme. To validate the numerical solution were compared the average Nusselt para each case, and comparing each result versus Vhals Davis is numerical solution and analytic correlations of Catton. I. Also were made a study of different aspects ratios and different angle in the cavities, this cases were compared with analytical correlations.

The developed program were Implemented with different interpolation convective term scheme (Differences centered, upwind, hybrid, potential and exponential), this implementations were made to obtain a mayor and deeper knowledge.


[*] Graduation thesis

[**] Faculty of Physical-Mechanical Engineering. School of Mechanical Engineering Director: Omar Armando Gélvez Arocha

# INTRODUCCIÓN.

La transferencia de calor por convección natural en recintos cerrados, ha sido extensamente estudiada tanto teórica como experimentalmente, uno de los casos más estudiados es de la cavidad rectangular, esto es debido a la cantidad de aplicaciones industriales en que se encuentra este fenómeno involucrado.

La mecánica de fluidos computacional es una herramienta que cada vez está siendo más empleada en la industria, debido a la mejora en la competitividad que esta le ofrece, un ejemplo es la industria aeronáutica que tiene más de 50 años empleado esta herramienta con muy buenos resultados. Estos hechos no han pasado desapercibido por la Escuela de Ingeniería Mecánica de la Universidad Industrial de Santander, que está incursionando en el desarrollo y aplicación de herramientas numéricas para el estudio de la mecánica de fluidos y la transferencia de calor.

Por estos motivos se presenta un estudio numérico simulando el campo de flujo de una cavidad rectangular bajo condiciones de convección natural, este estudio se divide en 6 capítulos. En el capítulo 1 se presenta un estudio del estado del arte y modelamiento matemático de las cavidades rectangulares bajo convección natural, en el capítulo 2 se explican las generalidades del método de los volúmenes finitos, los capítulos 3, 4 y 5 se presenta la discretización por volúmenes finitos e implementación de la solución numérica de las ecuaciones de difusión, convección difusión y Navier-Stokes. Finalmente, en el capítulo 6 se presenta el estudio numérico de las cavidades rectangulares, simulando el caso de referencia llamado Bouyancy driven cavity e implementación de una variación para distintos factores de forma.

# 1. JUSTIFICACIÓN PARA SOLUCIONAR EL PROBLEMA.

Gracias al gran crecimiento en la capacidad de procesamiento que ha tenido el computador, además de su masificación en los últimos 40 años, que ha pasado de ser una herramienta usada por los grandes centros científicos, a ser un elemento común en la mayoría de hogares en el mundo. El crecimiento de la capacidad de procesamiento y la masificación del computador, también han causado un gran crecimiento asociado a las ciencias computacionales y en este caso, de las ciencias computacionales aplicadas a la mecánica de fluidos, conocido como dinámica de fluidos computacionales o CFD en inglés (*computer fluid dynamics*). La dinámica de fluidos computacionales es una rama de la mecánica de fluidos, que mediante los métodos numéricos y algoritmos, permite solucionar problemas que involucren movimiento de fluido.

Una estadística del sitio http://scholar.google.com.co/ qué permite buscar la cantidad de artículos académicos publicados en el mundo, muestra que en el 2013 se publicaron aproximadamente 33 900 artículos sobre CFD en el mundo, esto demuestra al ritmo que se está investigando y la aplicación de la técnica a los diversos problemas de la mecánica de fluidos y la transferencia de calor.

Con el desarrollo de este proyecto la Escuela de Ingeniería Mecánica dispondrá del conocimiento para solucionar problemas análogos, contribuyendo al desarrollo tecnológico de la región y el país. Adicionalmente se continúa con la línea de desarrollo que tiene la escuela en este tema.

# 2. OBJETIVOS DEL TRABAJO DE GRADO.

## 2.1. OBJETIVO GENERAL.

Apoyar el avance tecnológico de la región en cumplimiento de la misión de la Escuela de Ingeniería Mecánica de la Universidad Industrial de Santander en la generación y adecuación de conocimiento mediante la simulación numérica del campo de flujos de un recinto cerrado bajo condiciones de convección natural.

## 2.2. OBJETIVOS ESPECÍFICOS.

- Estudiar el estado del arte acerca de las técnicas numéricas aplicadas en la mecánica de fluidos computacionales (CFD) en el campo de la convección natural en cavidades.

- Establecer un modelo matemático mediante la aplicación de ecuaciones de la conservación de materia, movimiento y energía para una cavidad rectangular bajo condiciones de convección natural.

- Simular y solucionar numéricamente el modelo matemático propuesto, para una cavidad de 1X1 con dos paredes adiabáticas y dos paredes isotérmicas para el caso de flujo laminar.

- Validar el modelo mediante la comparación de correlaciones analíticas.

- Hacer el estudio de casos para distintos factores de forma de la cavidad.

# 3. CONVECCIÓN NATURAL.

La convección natural es la trasferencia de calor que se genera por la inmersión de un cuerpo en un fluido que tiene una temperatura diferente a la del cuerpo, esa diferencia de temperatura causa un flujo de energía térmica entre el cuerpo y las partículas de fluido adyacentes al cuerpo, dando como resultado que la densidad de estas partículas de fluido aumente o disminuya, debido a la relación que existente entre la densidad y la temperatura. La diferencia de densidad entre el fluido adyacente y el fluido más lejano al cuerpo genera fuerzas de flotación debido a la acción del campo gravitacional determinado por el principio de Arquímedes[1].

Otro tipo de transferencia de calor por convección, es la convección forzada, en donde el flujo se crea por medio de un ventilador o una bomba. Aunque la transferencia de calor por convección natural en comparación a la convección forzada posea unos coeficientes de transferencia mucho menores, eso no quiere decir no sea utilizada en el diseño térmico de máquinas, por el contrario, en equipos de enfriamiento y refrigeración, máquinas eléctricas y líneas de transmisión de energía eléctrica, son diseñados para usar este modo de transferencia ya que la convección natural no consume energía pasa su uso. El flujo de calor por convección natural también se puede analizar en las paredes de un edificio, en el cuerpo humano y en el ciclo climático de nuestra atmósfera.

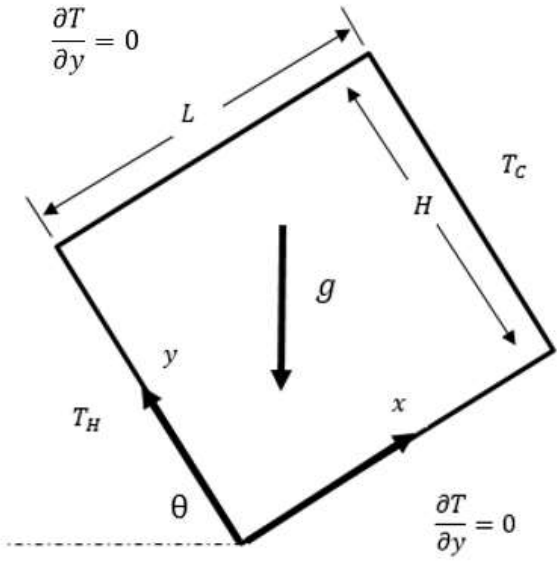
## 3.1. CONVECCIÓN NATURAL EN CAVIDADES RECTANGULARES.

El estudio de la convección natural en cavidades rectangulares ha sido abordado a través del tiempo tanto analíticamente (numéricamente) y experimentalmente.

[1] Principio de Arquimedes afirma que un cuerpo sumergido en un fluido recibe una fuerza de empuje hacia arriba igual al peso del volumen que este cuerpo posee.

Unos de los casos más estudiados es el de cavidad rectangular de dimensiones $LxH$ como se muestra en la figura 1, con dos paredes opuestas con diferentes temperaturas $T_H > T_c$ , mientras que las otras dos se mantienen adiabáticas $\frac{\partial T}{\partial y} = 0$. El ángulo inclinación Θ entre las zonas con temperatura y el eje horizontal, puede variar para distintos desde 0º (cavidad horizontal con calentamiento inferior), $90$º (cavidad vertical con calentamiento vertical) y $180$º (cavidad horizontal con calentamiento superior).

Figura 1. Cavidad rectangular




**3.1.1 Estudios previos** Las cavidades cuadradas han sido extensamente estudiadas, desde el punto de vista analítico y experimental. Catton I (1978) estudió la convección natural en cavidades desde el punto de vista experimental, especialmente las cavidades rectangulares, debido a que en los años 70's se dio un auge en el estudio de los colectores solares, en su estudio se enfocó en cavidades rectangulares horizontales, verticales e inclinadas. En los posteriores años Ayyaswasmy P. y Catton I. (1973) y Arnold y otros (1976), crearon correlaciones para determinar el número de Nusselt promedio para todos los ángulos Θ y para distintos factores de forma $\boldsymbol{H/L}$ en cavidades rectangulares.

Ostrach S. (1972), aplicó métodos numéricos para solucionar la convección natural en distintas cavidades, además, demostró las algunas deficiencias de estos métodos puramente numéricos.

Shiralkar G. y Tien C. (1981), analizaron numéricamente la convección natural para una cavidad rectangular horizontal ($\Theta = 0$), el análisis flujo de calor, además del campo de temperatura y velocidades, se calcularon para valores de Prantl de $0.01 < Pr < 100$ y Rayleigh $10^3 < Ra < 4 * 10^6$.

Raos M. (2001), estudió numéricamente la convección natural laminar en cavidades cuadradas (H/L=1) con distintos ángulos Θ ($60, 90, 160$), para ello trabajó con número de Prantl $0{,}73$ y Rayleigh $1.28 * 10^6$.

Lizardi A. (2011), realizó un estudio numérico de las cavidades rectangulares verticales ($\Theta = 90$) y factores de forma $H/L$ de 0.5, 1, 2 para establecer el campo de temperaturas y velocidades, para ello trabajó con un número de Prantl de $4.83$ y Rayleigh de $5.84 * 10^7$.

De Valhs Davis, D. (1983), realizó un estudio numérico para una cavidad cuadrada vertical ($\Theta = 90$). El fluido utilizado en las simulaciones fue aire Pr=0.71 y valores de Rayleigh $10^3 < Ra < 10^6$.

**3.1.2 Modelamiento matemático** El fin de modelar matemáticamente un problema físico es encontrar las variables y la relación entre las variables existentes que determinen el comportamiento del sistema físico, empleando algún tipo de formulismo matemático. Para crear el modelo matemático de la convección natural de una cavidad rectangular, se debe tener en cuenta que este fenómeno está regido por las ecuaciones que gobierna la dinámica de fluidos y la transferencia de calor, que son las leyes de conservación de la masa (Ec.1.1), el momento (Ec.1.2) y la energía (Ec.1.3).

Las ecuaciones de conservación pueden ser obtenidas tomando una cantidad de materia MC o masa de control, y analizando sus propiedades extensivas tales como la masa, el momento y la energía, este enfoque es muy útil en la dinámica de sólidos, pero en el caso de la dinámica de fluidos es mejor asignar un volumen de control VC y expresar los términos de forma intensiva ( por unidad de masa), ya que es más fácil analizar una cantidad de volumen en el espacio que seguir el trayecto de una masa de fluido. La ecuación de conservación puede ser escrita de forma conservativa o no conservativa. En este trabajo se utiliza la forma conservativa, eso quiere decir que el volumen de control es fijo en el espacio (enfoque de Euler), dando como resultado las siguientes ecuaciones:

- Conservación de la masa

$$\frac{\partial(\rho)}{\partial t} + \nabla \cdot (\rho \vec{v}) = 0 \qquad \text{(Ec.1.1)}$$

- Conservación de momento

$$\frac{\partial(\rho \vec{v})}{\partial t} + \nabla \cdot (\rho \vec{v} \vec{v}) = \nabla \cdot (\mu \nabla \vec{v}) - \nabla p + \rho \vec{g} \qquad \text{(Ec.1.2)}$$

- Conservación de la energía

$$\frac{\partial(\rho h)}{\partial t} + \nabla \cdot (\rho \vec{v} h) = \nabla \cdot (k \nabla h) + S_h \qquad \text{(Ec.1.3)}$$

Las ecuaciones de conservación pueden ser resumidas en una sola ecuación denominada de ecuación general de transporte (Ec.1.4), donde el término $\phi$ es la propiedad intensiva conservada ( para la conservación de masa $\phi = 1$ , para la conservación de momento $\phi = v$ , y la conservación de la energía $\phi = h$ ), Esta ecuación posee cuatro términos particulares

$$\underbrace{\frac{\partial(\rho\phi)}{\partial t}}_{Temporal} + \underbrace{\nabla\cdot(\rho\vec{v}\phi)}_{Convectivo} = \underbrace{\nabla\cdot(\Gamma\nabla\phi)}_{Difusivo} + \underbrace{S}_{Fuente} \qquad \text{(Ec.1.4)}$$

- Término temporal: muestra la variación del término $\phi$ en el tiempo.
- Término convectivo: representa el flujo de la variable $\phi$ que tiene con los otros dominios adyacentes al volumen de control.
- Término difusivo: representa el transporte a nivel molecular, Ley de Fourier para la difusión de calor, o la ley de Newton para la difusión de la cantidad de movimiento por efectos viscosos.
- Término fuente: tiene en cuenta la generación o la destrucción de la variable $\phi$.

Para transformar esta ecuación, en la ecuación de conservación de masa, momento y energía, solo se debe reemplazar los coeficientes de la tabla 1 en la Ec.1.4.

Tabla 1. Coeficientes.

| **Variable/ Coeficiente** | **Masa** | **Momento** | **Energía** |
|---|---|---|---|
| $\boldsymbol{\phi}$ | 1 | $(u, v, w)$ | $h$ |
| $\boldsymbol{\Gamma}$ | 0 | $(\mu, \mu, \mu)$ | $k$ |
| $\boldsymbol{S}$ | 0 | $(\frac{-\partial p}{dx} + S_x\ , \frac{-\partial p}{dy} + S_y\ , \frac{-\partial p}{dz} + S_z\ )$ | $S_h$ |

Para este trabajo se va a tomar la ecuación de conservación de la energía en función de la temperatura, ya que unos de los objetivos es determinar el campo de temperaturas, para ello, se asume que $\Delta h = C_p(T - T_{ref})$, asumiendo también que $C_p = constante$.

Conservación de la energía: $\phi = T \qquad \Gamma = \frac{k}{C_p} \quad S = S_h$

Reemplazando las variables en la ecuación general (Ec.1.1) se obtiene la ecuación de la conservación de la energía en términos de la temperatura (Ec.1.5).

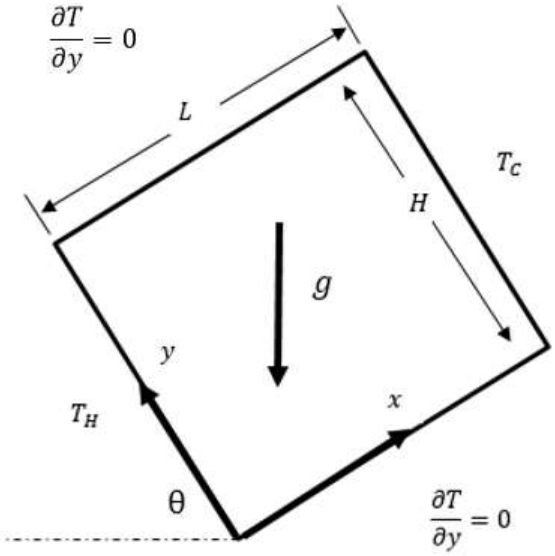
$$\frac{\partial(\rho T)}{\partial t} + \nabla \cdot (\rho \vec{v} T) = \nabla \cdot \left(\frac{k}{C_p} \nabla T\right) + S_h \qquad \text{(Ec.1.5)}$$

Aplicando el teorema de Gauss de la divergencia se puede obtener la ecuación general de transporte de forma integral (Ec.1.6)

$$\int_t^{t+\Delta t} \int_V \frac{\partial \rho \phi}{\partial t} dV dt + \int_t^{t+\Delta t} \int_S (\rho \vec{v} \phi) \cdot d\overrightarrow{A}\, dt = \int_t^{t+\Delta t} \int_S (\Gamma \nabla \phi) \cdot d\overrightarrow{A}\, dt + \int_t^{t+\Delta t} \int_V S_h dV dt \quad \text{(Ec.1.6)}$$

### 3.1.3 Modelo matemático de una cavidad rectangular bajo condiciones de convección natural.

Figura 2. Modelo matemático




Para crear el modelo matemático de una cavidad rectangular bajo condición de convección natural (Figura 2). Se deben realizar las siguientes suposiciones

- Flujo laminar: número de Rayleigh $Ra < 10^9$.
- Flujo bidimensional, además como las temperaturas son bajas se descarta la transferencia por radiación.
- El término viscoso de la ecuación de energía se desprecia debido a las bajas velocidades que maneja el flujo laminar.
- Condición de no deslizamiento $(u = v = 0)$ en las paredes
- Las propiedades como la viscosidad, el calor específico y la conductividad térmica son constantes. La densidad también será constante excepto en el término de flotación (aproximación de Boussinesq), la densidad en este término se considera una función lineal de la temperatura( Incropera y DeWitt, 1999):

$$\rho = \rho_0[1 - \beta(T - T_c)] \qquad \text{(Ec.1.6)}$$

- Donde $\rho_0$ es la densidad a la temperatura de referencia $T_c$, y $\beta$ es el coeficiente de expansión volumétrico.

Con las anteriores suposiciones las ecuaciones de conservación de masa, momento y energía son las siguientes:

- Conservación de la masa

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u}{\partial x} + \frac{\partial \rho v}{\partial y} = 0 \qquad \text{(Ec.1.7)}$$

- Conservación de momento

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = \frac{-}{\rho_0}\frac{\partial p}{\partial x} + \frac{\mu}{\rho_0}\left(\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2}\right) + \beta g(T - T_c)\cos\theta \qquad \text{(Ec.1.8)}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = \frac{-}{\rho_0}\frac{\partial p}{\partial y} + \frac{\mu}{\rho_0}\left(\frac{d^2v}{dx^2} + \frac{d^2v}{dy^2}\right) + \beta g(T - T_c)\text{sen}\theta \qquad \text{(Ec.1.7)}$$

- Conservación de la energía

$$\frac{\partial T}{\partial t} + u\frac{\partial T}{\partial x} + v\frac{\partial T}{\partial y} = \frac{k}{C_p}\left(\frac{d^2T}{dx^2} + \frac{d^2T}{dy^2}\right) \qquad \text{(Ec.1.9)}$$

Las condiciones de frontera de la figura 2 son:

$$0 < x < L \qquad y = 0 \qquad u = v = 0 \qquad \frac{\partial T}{dy} = 0$$

$$0 < x < L \qquad y = H \qquad u = v = 0 \qquad \frac{\partial T}{dy} = 0$$

$$0 < y < H \qquad x = 0 \qquad u = v = 0 \qquad T = T_H$$

$$0 < y < H \qquad x = L \qquad u = v = 0 \qquad T = T_C$$

Además las condiciones iniciales$(t = 0)$ son:

$$u = v = 0\,; \quad T = T_i$$

## 3.2. FORMA ADIMENSIONAL DE LAS ECUACIONES DE CONSERVACIÓN.

El análisis dimensional es una herramienta muy útil en el estudio experimental, esta herramienta permite simplificar un problema que depende de muchas variables independientes a un sistema de variables adimensionales más reducido, estos números adimensionales son relaciones entre las mismas variables independientes. La gran ventaja de utilizar este enfoque es que permite utilizar los resultados de un experimento en condiciones limitadas en otros experimentos donde usen otras dimensiones geométricas, cinemáticas y dinámicas. En la mecánica de fluidos computacional también se utiliza esta herramienta, en donde las variables independientes de las ecuaciones derivadas del modelo matemático (Ec.1.7-1.9) se transforman en su forma adimensional de siguiente modo:

Las variables adimensionales son las siguientes

$$t^* = \frac{t}{t_0} \qquad x_i^* = \frac{x}{L_0} \qquad u_i^* = \frac{u}{V_0} \qquad p_i^* = \frac{p}{\rho V_0^2} \qquad T^* = \frac{T - T_c}{T_1 - Tc}$$

En donde:

$t_0$=tiempo de referencia

$L_0$=longitud de referencia

$V_0$=velocidad de referencia

$\rho V_0^2$=presion de referencia

$T_1 - Tc$=temperatura de referencia

En esta tesis de grado se van a trabajar con cuatros números adimensionales, que se derivan de transformar las ecuaciones en derivadas parciales del modelo matemático (Ec.1.7-1.9) en su forma adimensional que son:

- Número de Nusselt.

Este número adimensional es un cociente entre la transferencia de calor por convección y la transferencia de calor por conducción.

$$Nu = \frac{hl}{k} = \frac{Transferencia\ de\ calor\ por\ convección}{Transferencia\ de\ calor\ por\ conducción}$$

- Número de Prantl.

El número de Prandtl proporciona el cociente entre la difusión de la cantidad de movimiento y la difusión térmica

$$Pr = \frac{C_p \mu}{k} = \frac{Velocidad\ de\ difusion\ de\ la\ cantidad\ de\ movimiento}{Velocidad\ de\ difusion\ de\ calor}$$

- Número de Reynolds.

El número de Reynolds caracteriza el régimen del flujo de un fluido

$$Re = \frac{\rho v_0 L_0}{\mu} = \frac{Fuerza\ de\ inercia}{Fuerzas\ de\ inercia\ viscosas}$$

- Número de Rayleigh.

El número de Rayleigh es un balance entre las fuerzas que promueven la convección, en el caso de este trabajo, es la diferencia de densidad y las fuerzas que se oponen que son la fricción viscosa y la difusión térmica, este número es el producto de otro dos números adimensionales que son el número de Grashof y el número de Prandtl

$$Ra = GrPr = \frac{g\beta(T_1 - T_c)L_0^3}{\mu\alpha}$$

$$Gr = \frac{g\beta(T_1 - T_c)L_0^3}{\mu^2} = \frac{Fuerza\ de\ flotacion}{Fuerzas\ de\ viscosidad}$$

### 3.3. CLASIFICACIÓN MATEMÁTICA DE LOS FLUJOS.

Las ecuaciones de conservación como se muestra en la ecuación (Ec.1.4) son ecuaciones diferenciales parciales cuasi lineales que poseen términos con derivadas tanto de primer, como de segundo orden. En el caso de los flujos convectivos aparecen términos con derivadas de primer orden que expresan propiedades de transporte, mientras que en los flujos difusivos aparecen términos con derivadas de segundo orden debido a la Ley de Fick[2].

Los flujos se clasifican principalmente por el comportamiento que poseen, es decir, como se propagan la información. Desde el punto de vista físico la propagación de la información es de dos tipos. Los problemas de equilibrio (elípticos) y los problemas transitorios (parabólicos e hiperbólicos) (Fernández Oro 2012). La solución de los problemas elípticos depende únicamente de las condiciones de frontera, mientras que los problemas transitorios dependen tanto de las condiciones de contornos como de las condiciones iniciales.

- Comportamiento elíptico: el ejemplo más conocido de un problema elíptico es la ecuación de Laplace (Ec.1.10), la característica más importante de esta ecuación es que cualquier perturbación en un punto del dominio afecta a los demás puntos, eso quiere decir que el valor de la propiedad de cualquier punto excepto lo que están en frontera, son la interpolación de los valores de los puntos vecinos.

$$\frac{\partial^2\theta}{\partial x^2} + \frac{\partial^2\theta}{\partial y^2} = 0 \qquad \text{(Ec.1.10)}$$

Para resolver este tipo de problemas es necesario conocer las condiciones de frontera. Esta ecuación es característica de la transferencia de calor por conducción en estado estacionario.

- Comportamiento parabólico: la ecuación parabólica en su forma más básica (Ec.1.11) es una ecuación de difusión de forma transitoria, la particularidad de esta ecuación es que cualquier perturbación al interior del dominio solo tiene influencia en momentos posteriores a la aparición de la perturbación, este tipo de ecuaciones además de asignarle las condiciones de frontera, también se le asigna la condición inicial (t=0).

$$\frac{\partial \phi}{\partial t} = \frac{\partial^2 \phi}{\partial x^2} \qquad \text{(Ec.1.11)}$$

Las ecuaciones parabólicas tienden a estado estacionario cuando t=∞, eso quiere decir que a medida que transcurre el tiempo, el término temporal va perdiendo influencia, y se convierte en un comportamiento elíptico.

- Comportamiento hiperbólico: la ecuación hiperbólica en su forma más básica es una ecuación de onda (Ec.1.12), este comportamiento es característico de movimientos vibratorios y ondulatorios, además difiere del parabólico y el elíptico que tiene una velocidad de propagación finita $\omega$. Esta ecuación también está ligada al flujo compresible en régimen transónico y supersónico, debido a la velocidad del sonido puede dar lugar a la aparición de ondas de choques que son una clara manifestación de un comportamiento hiperbólico.

$$\frac{\partial^2 \phi}{\partial t^2} = \omega^2 \frac{\partial^2 \phi}{\partial x^2} \qquad \text{(Ec.1.12)}$$

### 3.4. CONDICIONES INICIALES Y DE FRONTERA.

Para hallar la solución de las ecuaciones diferenciales de conservación, se requiere conocer simplemente las condiciones de frontera en el caso de un

problema estacionario, por el contrario, si el problema es transitorio se necesita además de las condiciones de frontera, la condición inicial cuando (t=0). Las condiciones de frontera se clasifican de la siguiente forma:

- Condición Dirichlet: se caracteriza por que conoce la magnitud de la propiedad ($\phi$) en la frontera:

$$\phi = constante$$

- Condición de Von Neumman: se caracteriza por que se conoce el valor del gradiente ($\nabla\phi$) de la propiedad en la frontera :

$$\nabla\phi = constante$$

- Condición de Robin: es la combinación lineal de las dos condiciones anteriores.

# 4. METODOS DE VOLUMENES FINITOS.

## 4.1. CONCEPTOS GENERALES DEL MÉTODO DE LOS VOLÚMENES FINITOS.

El objetivo principal del método de los volúmenes finitos (MVF) es desarrollar una técnica numérica que permita resolver la ecuación general de transporte (Ec.1.4); la solución obtenida mediante este método, solo permite conocer el valor $\phi$ únicamente en los puntos discretos definidos por la malla del dominio, difiriendo de la solución analítica que halla el valor de $\phi$ para todo el dominio. El primer paso de la implementación del MVF es discretizar el dominio espacial mediante una malla, para poder asignar a cada punto una coordenada y el valor $\phi$ correspondiente. El segundo paso que propone el MVF es asociar los volúmenes de control a las celdas de las mallas (cell-centred) o a los nodos (cell-vertex), para posteriormente aplicar a cada volumen de control las leyes integrales de conservación.

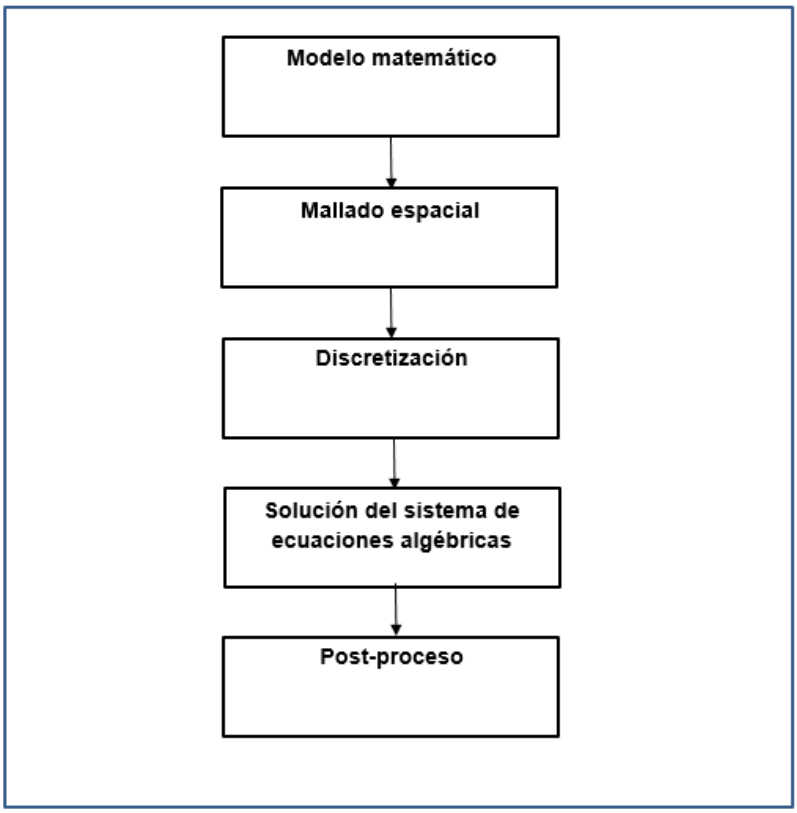
Unas de las grandes ventajas del MVF es que permite su implementación en mallas estructuradas como no estructuradas. Además que los valores discretos de $\phi$ están promediados en la celda, esta es la gran diferencia del MVF con respecto a MDF (métodos de la diferencias finitas) o MEF (método de los elementos finitos), en donde la variable $\phi$ es el valor local en el nodo de la malla, por lo anterior las discretizaciones conservativas son la base del MVF, y esto encaja perfectamente con el hecho de tener que resolver la ecuación de transporte conservativa.

En las ecuaciones discretizadas es muy importante mantener también la conservación de la variable $\phi$ que en esta tesis son la masa, la cantidad de movimiento y la energía.

## 4.2. PASOS GENERALES DEL MVF.

El mapa conceptual de la figura 3, se explican los pasos generales para la implementación de la solución de cualquier problema de transporte mediante volúmenes finitos. Se tiene que tener en cuenta que el MVF es un método de discretización, que a su vez los métodos de discretización son parte de cualquier solución por métodos numéricos. Por ende el esquema de la figura 3, más que enfocarse en la metodología del MVF, lo que muestra es, los pasos generales de la implementación de la solución numérica de cualquier fenómeno utilizando el método de los volúmenes finitos.

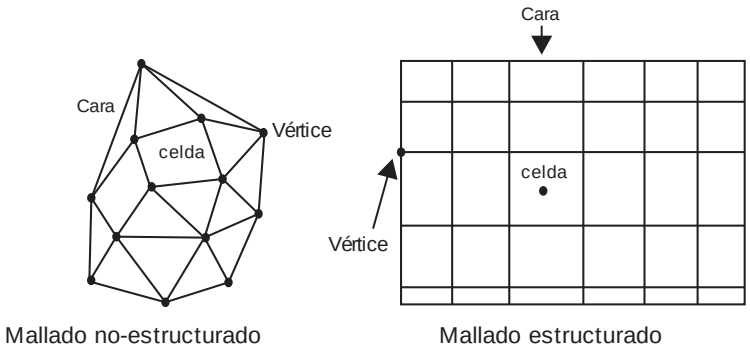
Figura 3. Pasos generales MVF.



**4.2.1 Modelamiento matemático** el modelo matemático como se explicó en la sección 1.1.2, es el punto de partida para la implementación de la solución por cualquier método numérico, debido que el modelo matemático proporciona un sistema de ecuaciones diferenciales o integrales, que posteriormente se va transformar mediante un método de discretización, en el caso de esta tesis mediante el método de los volúmenes finitos**.**

**4.2.2 Mallado espacial** en cualquier problema de MVF la generación del mallado tiene como un fin el discretizar el dominio físico en un número finito de celdas, para poder aplicar las leyes de conservación a cada una de estas. Las mallas clasifican en dos grandes grupos: las mallas estructuradas y mallas no estructuradas (figura 4), la diferencia entre ambas es que el mallado estructurado se genera mediante una red de familias coordenadas, mientras que la malla no estructurada no tienen ningún tipo de organización preferente. Debido a que en esta tesis el dominio tiene una forma rectangular, se trabajará con mallado estructurado cartesiano.

Figura 4. Tipos de mallas



**4.2.3 Discretización de las ecuaciones** unos de los objetivos del MVF es aplicar la ecuación general de transporte (Ec.1.4) a cada celda de la malla, para ello se tiene que discretizar la ecuación general de transporte de forma integral (Ec.1.6) dando como producto final una sistema de ecuaciones algebraicas que relacione la información de los centroides de cada una de las celdas de la forma:

$$a_p \phi_p = \sum_{nv} a_{nv} \phi_{nv} + b$$

Donde:

$P$ = volumen de control que se está analizando.

$a_p$ = coeficiente del volumen de control que se está analizando.

$\phi_p$ = valor de la variable dependiente en el volumen de control.

$nv$ = número de vecinos que tiene nuestro volumen de control, para esta tesis es 4.

$a_{nv}$ = coeficientes de los volúmenes de control vecinos

$\phi_{nv}$ = valor de la variable dependiente de los volúmenes de control vecinos.

$b$ = suma de los valores independiente de la ecuación.

**4.2.4 Solución del sistema de ecuaciones** una vez que se discretiza la ecuación general en todo el dominio se obtiene como producto final un sistema de ecuaciones algebraicas, que puede ser resuelto por distintos métodos, la selección del método más idóneo depende de los siguientes factores:

- La linealidad de las ecuaciones:

Cuando se desea resolver un sistema de ecuaciones se puede solucionar de dos formas: con un método directo o un método iterativo. En un método directo no se necesita iteraciones para llegar a la solución, en vez de eso, el método va resolviendo de nodo a nodo hasta solucionar todo el dominio, este método es muy aceptable para implementar en problemas lineales, debido a que resuelve el sistema de ecuaciones de una vez, pero se tiene que ser muy cuidadoso al escoger el método directo por qué no todos los métodos directos convergen en

un mismo problema. Pero cuando requiere resolver un problema no lineal no es muy económico, debido a que se necesita resolver repetidamente el sistema de ecuaciones mientras se actualizan los coeficientes. En este caso es recomendable utilizar un método de solución iterativo (Patankar 1980), en donde el sistema de ecuaciones se compacta de la siguiente forma.

$$\boldsymbol{Au} = \boldsymbol{b} \qquad \text{(Ec.2.1)}$$

Donde:

$A$ =Matriz de coeficientes

$u$ = Vector solución

$b$ =Vector de términos independientes.

En todos los métodos iterativos lo que se busca es encontrar los valores del vector solución, esto se hace proponiendo un vector solución inicial y mediante el método de interacción se halla el vector solución real, este proceso es mucho más rápido y exige menos capacidad computacional que los métodos directos. En esta tesis se van a usar las librerías del PhD. David Fuentes, estas librerías poseen distintos métodos de solución tanto directos como iterativos, por esto no se va a profundizar en los métodos de solución en este apartado.

- El tiempo de solución y la capacidad de almacenamiento del computador.

El tiempo de solución es una variable muy importante a la hora de escoger un método de solución, en forma general los métodos directos son más lentos y consumen mucha más capacidad computacional que los métodos iterativos (Patankar 1980).

**4.2.5 Post-proceso** el último paso para la implementación de la solución numérica es el postproceso, en este paso se analizan y se validan los resultados, en el caso de que los resultados no concuerden con la realidad del fenómeno se revisa y se propone mejoras al modelo. Para esta tesis de grado la herramienta utilizada para el postproceso es Paraview.

## 4.3. PROPIEDADES DE LOS METODOS DE SOLUCIÓN NUMERICA.

Para que una solución numérica tenga validez debe poseer cuatros propiedades que son, la precisión, consistencia, estabilidad y convergencia.

**4.3.1 Precisión** las soluciones numéricas en la mecánica de fluidos siempre son soluciones aproximadas, esto es debido a los errores que se generan en etapa de diseño del algoritmo, en su programación o en el mismo modelo matemático. Por ello cualquier solución numérica siempre conlleva tres tipos de errores sistemáticos que son los errores de modelo, errores de discretización y errores de iteración**.**

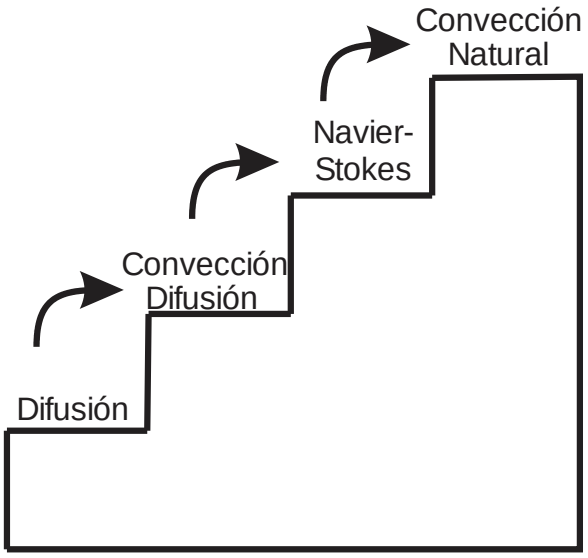
**4.3.2 Consistencia** un método numérico es consistente cuando al reducir la distancia entre nodos, $\Delta \boldsymbol{x} \rightarrow \mathbf{0}$ o cuando se reduce el paso de tiempo $\Delta \boldsymbol{t} \rightarrow \mathbf{0}$, la ecuación discreta tiende a ser igual que la ecuación exacta. La diferencia entre la ecuación discreta y la ecuación exacta se denomina error de truncamiento, debido que al transformar la ecuación exacta a una ecuación discreta por medio de la series de Taylor solo se escogen uno o dos miembros de toda la serie.

**4.3.4 Estabilidad** un método numérico es estable, cuando en el proceso solución no se generan errores que permita salirse a la solución de los valores acotados, en el caso de los métodos iterativos, una solución estable es cuando la solución no diverge.

**4.4.5 Convergencia** una solución converge si la solución de la ecuación discreta, tiende a los valores de la solución exacta cuando se reduce la distancia entre los nodos $\Delta \boldsymbol{x} \rightarrow \mathbf{0}$.

Debido a que las ecuaciones 1.7-1.9, son un sistema de ecuaciones parciales no lineales, en esta tesis la solución numérica del campo de flujo se realizará mediante un método iterativo de solución. Existen muchos métodos iterativos uno de los más conocidos es el método SIMPLE, este método será explicando con detenimiento en el capítulo 5, por ahora lo importante es tener conocimiento que para desarrollar el método SIMPLE se necesita como requisito indispensable tener un conocimiento acerca de cómo se resuelven las ecuaciones de difusión, convección-difusión y Navier-Stokes, por esto se plantea un plan de trabajo mostrado en la figura 5.

Figura 5. Plan de trabajo escalonado.



## 5. DIFUSIÓN DE CALOR.

### 5.1. DISCRETIZACIÓN.

Para discretizar la ecuación de difusión de calor en estado no estacionario, se procede a eliminar el termino convectivo de la ecuación de conservación de la energía (Ec.1.5)

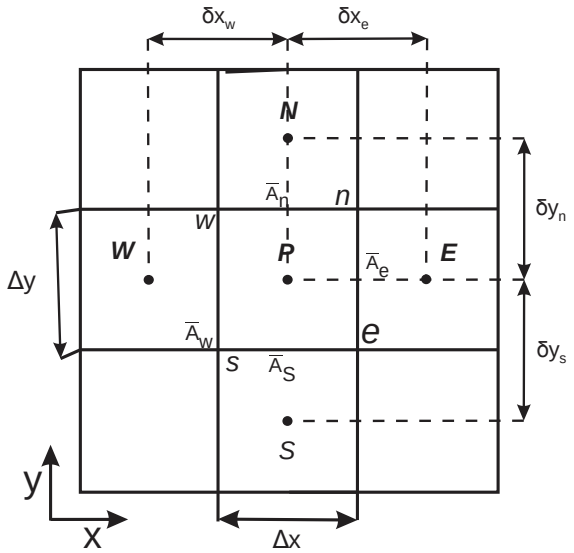
$$\frac{\partial(\rho C_p T)}{\partial t} + \cancel{\nabla \cdot (\rho \vec{v} C_p T)}^{0} = \nabla \cdot (K \nabla T) + S_h \qquad \text{(Ec.3.1a)}$$

De forma integral

$$\int_t^{t+\Delta t} \int_V \frac{\partial \rho C_p T}{\partial t} dV dt = \int_{\Delta t}^{t+\Delta t} \int_S (K \nabla T) \cdot d\overrightarrow{A}\, dt + \int_{\Delta t}^{t+\Delta t} \int_V S_h dV dt \qquad \text{(E.c.3.1b)}$$

Como primer paso antes de integrar la ecuación 3.1b, se debe seleccionar el tipo malla para definir los límites de integración en cada volumen de control. El mallado será estructurado ortogonal y los límites en cada celda están definidos como lo muestra la figura 6.

Figura 6. Volumen de control bidimensional.



Donde P es la volumen de control a integrar, mientras que N, S, W y E son los volúmenes de control vecinos y finalmente n, s, w y e son las fronteras o caras del volumen de control P.

Se procede a integrar cada término por separado empezando por el término temporal. El volumen total de P es $\Omega_p = \Delta x \Delta y$, el tiempo futuro $t + \Delta t$ de la temperatura en P es igual $T_p^1$ y el tiempo presente $t$ de la temperatura en P es igual a $T_p^0$ como se observa en la ecuación 3.2.

$$\int_t^{t+\Delta t} \int_{\Omega_p} \frac{\partial \rho C_p T}{\partial t} dV dt = \frac{\rho C_p \Delta x \Delta y (T_p^1 - T_p^0)}{\Delta t} \qquad \text{(Ec.3.2)}$$

Ahora, para integrar el término difusivo, se resuelve en primera instancia la integral de la trasferencia de calor, dando como resultado que la transferencia de calor en el volumen de control P es igual a la suma de los flujos de calor de las caras e, w, s y n.

$$\int_{\Delta t}^{t+\Delta t} \int_S (k \nabla T) \cdot d\overrightarrow{A} dt = \int_{\Delta t}^{t+\Delta t} \sum_{v=e,w,n,s} (J_v A_v) dt \qquad \text{(Ec.3.3)}$$

Para resolver la integral temporal se supone que los flujos difusivos varían de forma lineal entre el tiempo presente y el tiempo futuro según un factor de ponderación $\theta$ .

$$\int_{\Delta t}^{t+\Delta t} \sum_{v=e,w,n,s} (J_v A_v) dt = \sum_{v=e,w,ns} [\theta J_v^1 + (1-\theta) J_v^0] A_v \Delta t \qquad \text{(Ec.3.4)}$$

Donde por ejemplo para conocer el valor de los flujos de calor presentes y futuros de la cara e se calculan de la siguiente forma:

$$J_e^0 = U_e A (T_e^0 - T_p^0) \qquad\qquad J_e^1 = U_e A (T_e^1 - T_p^1) \qquad \text{(Ec.3.5-3.6)}$$

$$[\theta J_e^1 + (1-\theta)J_e^0]A_e\Delta t = \left[\theta U_e A_e\left(T_e^1 - T_p^1\right) + (1-\theta)U_e A_e(T_e^0 - T_p^0)\right]\Delta t \quad \text{(Ec.3.7)}$$

Siendo la variable $U_e$ el coeficiente global de trasferencia de calor.

$$U_e \begin{cases} \frac{k}{\delta x_{eE}+\delta x_{Pe}} \; si \; k_e = k_p = k \\ \\ \frac{1}{\frac{\delta x_{eE}}{k_e}+\frac{\delta x_{Pe}}{k_p}} \; si \; k_e \neq k_p \end{cases} \quad \text{(Ec.3.8)}$$

El término difusivo bidimensional discretizado para todas las caras del volumen de control P, asumiendo que los flujos de calor se dirigen desde los volúmenes de control vecinos al volumen de control P es:

$$[\theta J_v^1 + (1-\theta)J_v^0]A_c\Delta t = \sum_{v=e,w,ns}\left[\theta U_v A_v\left(T_v^1 - T_p^1\right) + (1-\theta)U_v A_v(T_v^0 - T_p^0)\right]\Delta t \quad \text{(Ec.3.9)}$$

Para finalizar la discretización de la ecuación de calor 2D no estacionaria, se procede con la discretización del término fuente, asumiendo para la integral temporal el mismo perfil lineal que implementó en el término difusivo.

$$\int_{\Delta t}^{t+\Delta t}\int_V S_h dV dt = [\theta S_h^1 + (1-\theta)S_h^0]V_p\Delta t \quad \text{(Ec.3.10)}$$

Reemplazando cada término en la ecuación 3.1b y dividiéndola entre $\Delta t$ , queda la ecuación discretizada de la siguiente forma:

$$\frac{\rho C_p \Delta x \Delta y\left(T_p^1 - T_p^0\right)}{\Delta t} = \sum_{v=e,w,ns}\left[\theta U_v A_v\left(T_v^1 - T_p^1\right) + (1-\theta)U_v A_v\left(T_v^0 - T_p^0\right)\right] + [\theta S_h^1 + (1-\theta)S_h^0]V_p \quad \text{(Ec.3.11)}$$

Para después agrupar la ecuación algebraicamente como:

$$a_p T_p^1 = \sum_{v=e,w,n,s} a_v T_v + b \qquad \text{(Ec.3.12)}$$

Quedando resumida de la siguiente manera para los nodos internos.

$$\underbrace{a_n T_n^1 + a_s T_s^1 + a_w T_w^1 + a_e T_e^1 - a_p T_p^1}_{\textit{Términos dependientes.}} = \underbrace{-b}_{\textit{Términos independientes.}} \qquad \text{(Ec.3.13)}$$

Donde:

$a_n = \theta U_n A_n \qquad a_s = \theta U_s A_s \qquad a_w = \theta U_w A_w \qquad a_e = \theta U_e A_e \quad a_0 = \frac{\rho C_p \Delta x \Delta y}{\Delta t}$

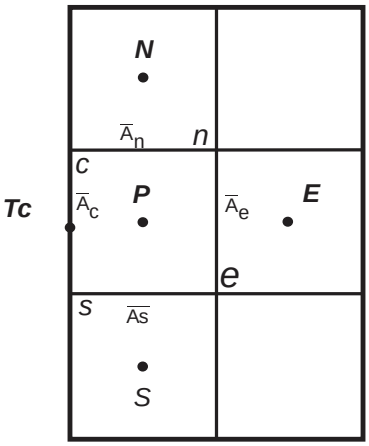
$a_p = a_n + a_s + a_w + a_e + a_0$

$$b = \sum_{v=e,w,ns} \left[(1-\theta) U_v A_v \left(T_v^0 - T_p^0\right)\right] + [\theta S_h^1 + (1-\theta) S_h^0] V_p + a_0 T_p^0$$

Si el término de ponderación es $\theta = 0$, la solución de esta ecuación diferencial parabólica se resuelve mediante un método de solución explícito, por el contrario si $\theta = 1$ la solución de la ecuación se alcanza mediante un método implícito y finalmente $\theta = 0.5$ la solución se alcanza mediante el método Crank-Nicholson.

**5.1.1 Condiciones de frontera** como se mencionó el apartado 1.4 del capítulo 1, existen tres tipos de condiciones de frontera: Dirichlet, Newmann y Mixta. En esta tesis se trabajará con la condiciones de frontera Dirichlet y Newmann.

Figura 7. Condición de frontera tipo Dirichlet.




La condición de frontera Dirichlet como se observa en la figura 6 es cuando se conoce el valor de la temperatura Tc alguno de los nodos vecinos, introduciendo la condición de frontera en la ecuación 3.12 y desarrollando se obtiene:

$$a_n T_n^1 + a_s T_s^1 + a_e T_e^1 - a_p T_p^1 = -b \qquad \text{(Ec.3.14)}$$

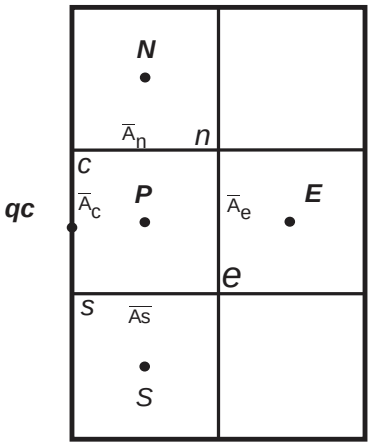
Donde:

$$a_c = \theta U_c A_c \qquad a_n = \theta U_n A_n \qquad a_s = \theta U_s A_s \qquad a_e = \theta U_e A_e \qquad a_0 = \frac{\rho C_p \Delta x \Delta y}{\Delta t}$$

$$a_p = a_n + a_s + a_e + a_0 + a_c$$

$$b = \sum_{v=e,n,s} \left[(1-\theta) U_v A_v \left(T_v^0 - T_p^0\right)\right] + \left[\theta S_h^1 + (1-\theta) S_h^0\right] V_p + a_0 T_p^0 + U_c A_c T_c$$

Figura 8. Condición de frontera tipo Newmann.




En la condición de frontera Newman se conoce directamente el flujo calor en la cara $q_c$ por lo tanto $J_c A_c = q_c \Delta y$ , introduciendo la expresión en la ecuación 3.12 y tras desarrollar conduce a:

$$a_n T_n^1 + a_s T_s^1 + a_e T_e^1 - a_p T_p^1 = -b \qquad \text{(Ec.3.15)}$$

Donde

$a_n = \theta U_n A_n$ $\quad a_s = \theta U_s A_s$ $\quad a_e = \theta U_e A_e$ $\quad a_0 = \frac{\rho C_p \Delta x \Delta y}{\Delta t}$

$a_p = a_n + a_s + a_e + a_0$

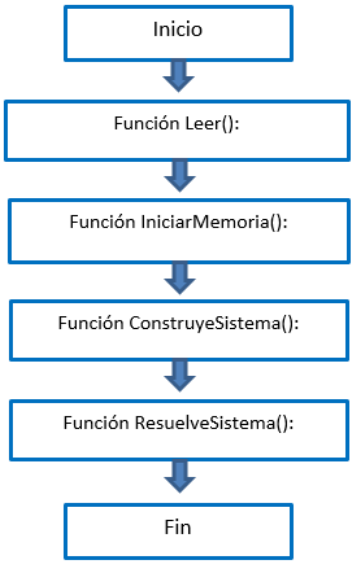
$b = \sum_{v=e,n,s} \left[(1-\theta) U_v A_v \left(T_v^0 - T_p^0\right)\right] + [\theta S_h^1 + (1-\theta) S_h^0] V_p + a_0 T_p^0 + q_c \Delta y$

## 5.2. IMPLEMENTACIÓN DEL CODIGO COMPUTACIONAL.

Para la implementación del código computacional que permite solucionar la ecuación general de la difusión de calor estacionaria y no-estacionaria, se creó

la clase CDifusion. La clase CDifusion es derivada de las clases Puntero_Base que permite manipulación de memoria dinámica y GuardarEnsight que permite volcar los datos a Paraview para su visualización. Las funciones principales de la clase CDifusion están esquematizadas de forma procedimental en la figura 9.

Figura 9. Funciones principales de la clase CDifusion.




- Función Leer():

Permite obtener los datos geométricos del dominio, propiedades físicas, condiciones de frontera, esquema temporal, tipo y tamaño de mallado del dominio.

- Función IniciaMemoria():

Asigna los espacios de memoria para las variables involucradas en el problema, además define el tamaño de la matriz de coeficientes, vector de coeficientes y vector solución.

- Función ConstruyeSistema()

Calcula el flujo de calor difusivo en las caras de cada volumen de control y construye el sistema de ecuaciones algebraicas

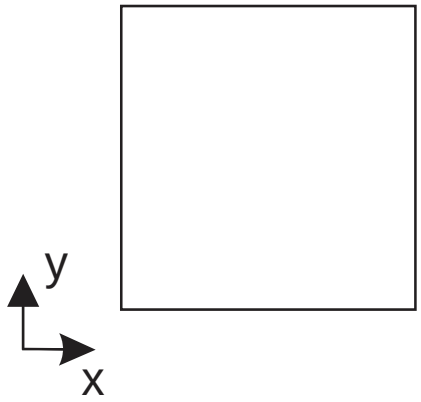
- Función ResuelveProblema()

Soluciona el sistema de ecuaciones algebraicas generando el vector solución, calcula el gradiente de la variable en cada volumen de control y permite volcar los datos a Paraview para su visualización.

## 5.3. CASO DE ESTUDIO: DIFUSION DE CALOR ESTACIONARIA EN PLACA.

Una placa de acero estructural de 10x10 cm con las siguientes condiciones de frontera.

Figura 10. Placa de acero.




Las condiciones de frontera de la figura 2 son:

$$0 < x < 10 \qquad y = 0 \qquad T = 30\,°C$$

$$0 < x < 10 \qquad y = 10 \qquad T = 250\,°C$$

$$0 < y < 10 \qquad x = 0 \qquad T = 250\,°C$$

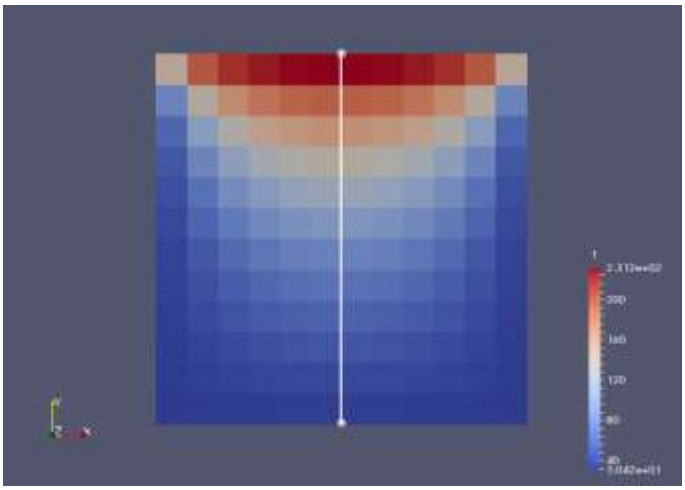
$$0 < y < 10 \qquad x = 10 \qquad T = 250\,°C$$

Tabla 2. Propiedades del acero.

| Propiedades | Valor |
|---|---|
| Densidad (rho) | 7850 kg/m^3 |
| Conductividad (k) | 60.5 W/(m°C) |
| Calor específico (C) | 434 (J/kg°C) |

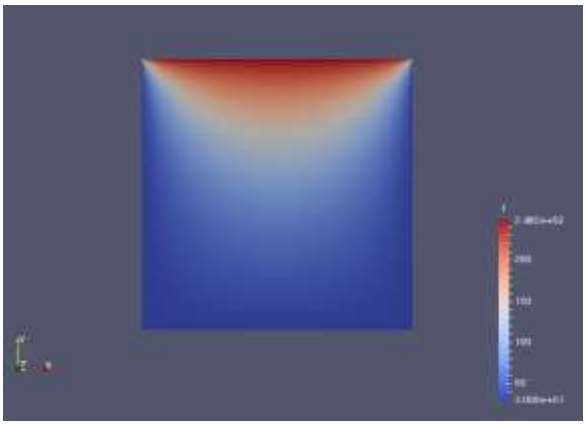
**5.3.1 Solución numérica** para la solución numérica de la transferencia de calor en la placa, como primera aproximación se generó una malla de 12x12 volúmenes de control, dando como resultado la siguiente distribución de temperaturas**.**

Figura 11. Distribución de temperaturas malla 12x12.



Después se procede a refinar el mallado, para una malla 120x120 se obtuvo la siguiente distribución de temperaturas.

Figura 12. Distribución de temperaturas malla 120x120.



**5.3.2 Validación** para validar la solución numérica se compararon los resultados con la solución analítica de la ecuación de difusión estacionaría en dos dimensiones. En donde la distribución de temperatura está determinada por la siguiente ecuación.

$$\theta(x,y) = \sum_{n=1}^{\infty} C_n \sin\left(\frac{n\pi x}{L}\right) \sinh\left(\frac{n\pi y}{L}\right) \qquad \text{(Ec. 3.16)}$$

$$C_{n=} \frac{2\left[(-1)^{n+1}+1\right]}{n\pi \sinh\left(\frac{n\pi W}{L}\right)}$$

En donde para las mismas condiciones de frontera y un mallado de 12x12 nodos como se muestra en la figura 13, se obtuvo como resultado la distribución de temperatura en la placa mostrada en la figura 14.

Figura 13. Distribución de nodos.

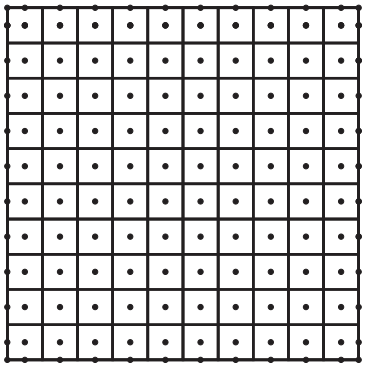
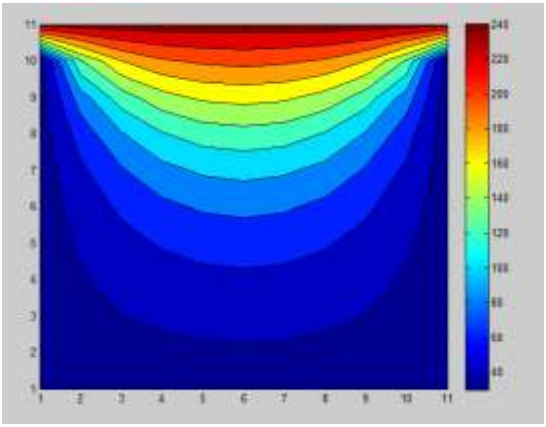


Figura 14. Solución analítica.



Para contrastar los resultados de la solución numérica respecto a la analítica, se compara los valores de temperatura en los puntos donde se encuentran los nodos internos (figura 13). Obteniendo en la solución analítica la matriz de temperaturas mostrada en al tabla 3, de la misma forma en la solución numérica para 12x12 volumen (tabla 4) y 120x120 volúmenes (tabla 5).

Tabla 3. Matriz de temperatura analítica.

| X/Y | 0.05 | 0.15 | 0.25 | 0.35 | 0.45 | 0.55 | 0.65 | 0.75 | 0.85 | 0.95 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0.05 | 30.60181 | 31.74409 | 32.71033 | 33.4075 | 33.77197 | 33.77197 | 33.4075 | 32.71033 | 31.74409 | 30.60181 |
| 0.15 | 31.86823 | 35.41207 | 38.40511 | 40.56055 | 41.68568 | 41.68568 | 40.56055 | 38.40511 | 35.41207 | 31.86823 |
| 0.25 | 33.33206 | 39.64346 | 44.95377 | 48.76041 | 50.74017 | 50.74017 | 48.76041 | 44.95377 | 39.64346 | 33.33206 |
| 0.35 | 35.15901 | 44.90199 | 53.03729 | 58.8157 | 61.79928 | 61.79928 | 58.8157 | 53.03729 | 44.90199 | 35.15901 |
| 0.45 | 37.58297 | 51.82001 | 63.53229 | 71.70707 | 75.87144 | 75.87144 | 71.70707 | 63.53229 | 51.82001 | 37.58297 |
| 0.55 | 40.98702 | 61.37232 | 77.65962 | 88.65964 | 94.12856 | 94.12856 | 88.65964 | 77.65962 | 61.37232 | 40.98702 |
| 0.65 | 46.11943 | 75.27804 | 97.23592 | 111.1843 | 117.834 | 117.834 | 111.1843 | 97.23592 | 75.27804 | 46.11943 |
| 0.75 | 54.81212 | 97.03561 | 125.0462 | 140.9664 | 148.0679 | 148.0679 | 140.9664 | 125.0462 | 97.03561 | 54.81212 |
| 0.85 | 73.29434 | 134.5879 | 164.9158 | 179.2594 | 185.122 | 185.122 | 179.2594 | 164.9158 | 134.5879 | 73.29434 |
| 0.95 | 139.7602 | 202.9504 | 219.1222 | 225.4111 | 227.6074 | 227.6074 | 225.4111 | 219.1222 | 202.9504 | 139.7602 |

Tabla 4. Matriz distribución de temperaturas numérico 12x12.

| X/Y | 0.05 | 0.15 | 0.25 | 0.35 | 0.45 | 0.55 | 0.65 | 0.75 | 0.85 | 0.95 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0.05 | 30.42 | 31.23 | 31.96 | 32.96 | 33.17 | 33.17 | 32.96 | 31.96 | 31.23 | 30.42 |
| 0.15 | 31.3 | 33.79 | 36.01 | 39.07 | 39.72 | 39.72 | 39.07 | 36.01 | 33.79 | 31.3 |
| 0.25 | 32.26 | 36.62 | 40.5 | 45.8 | 46.92 | 46.92 | 45.8 | 40.5 | 36.62 | 32.26 |
| 0.35 | 34.81 | 44.01 | 52.1 | 62.95 | 65.21 | 65.21 | 62.95 | 52.1 | 44.01 | 34.81 |
| 0.45 | 36.61 | 49.21 | 60.13 | 74.5 | 77.43 | 77.43 | 74.5 | 60.13 | 49.21 | 36.61 |
| 0.55 | 39.04 | 56.09 | 70.54 | 88.92 | 92.57 | 92.57 | 88.92 | 70.54 | 56.09 | 39.04 |
| 0.65 | 42.49 | 65.56 | 84.36 | 107 | 111.4 | 111.4 | 107 | 84.36 | 65.56 | 42.49 |
| 0.75 | 47.84 | 79.3 | 103.2 | 129.7 | 134.5 | 134.5 | 129.7 | 103.2 | 79.3 | 47.84 |
| 0.85 | 78.64 | 136.2 | 166.8 | 191.4 | 195 | 195 | 191.4 | 166.8 | 136.2 | 78.64 |
| 0.95 | 139.6 | 198.8 | 218.4 | 229.7 | 231.2 | 231.2 | 229.7 | 218.4 | 198.8 | 139.6 |

Tabla 5. Matriz distribución de temperaturas numérico 120x120.

| X/Y | 0.05 | 0.15 | 0.25 | 0.35 | 0.45 | 0.55 | 0.65 | 0.75 | 0.85 | 0.95 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0.05 | 30.51 | 31.56 | 32.45 | 33.1 | 33.45 | 33.45 | 33.1 | 32.45 | 31.56 | 30.51 |
| 0.15 | 31.66 | 35.12 | 38.05 | 40.18 | 41.32 | 41.32 | 40.18 | 38.05 | 35.12 | 31.66 |
| 0.25 | 33 | 39.21 | 44.47 | 48.27 | 50.29 | 50.29 | 48.27 | 44.47 | 39.21 | 33 |
| 0.35 | 34.65 | 44.28 | 52.37 | 58.16 | 61.22 | 61.22 | 58.16 | 52.37 | 44.28 | 34.65 |
| 0.45 | 36.85 | 50.95 | 62.62 | 70.84 | 75.12 | 75.12 | 70.84 | 62.62 | 50.95 | 36.85 |
| 0.55 | 39.93 | 60.14 | 76.41 | 87.51 | 93.15 | 93.15 | 87.51 | 76.41 | 60.14 | 39.93 |
| 0.65 | 44.56 | 73.51 | 95.54 | 109.7 | 116.6 | 116.6 | 109.7 | 95.54 | 73.51 | 44.56 |
| 0.75 | 52.36 | 94.44 | 122.8 | 139.1 | 146.5 | 146.5 | 139.1 | 122.8 | 94.44 | 52.36 |
| 0.85 | 68.84 | 130.7 | 162.1 | 177.1 | 183.3 | 183.3 | 177.1 | 162.1 | 130.7 | 68.84 |
| 0.95 | 127.8 | 198.2 | 216.2 | 223.1 | 225.8 | 225.8 | 223.1 | 216.2 | 198.2 | 127.8 |

Además se calcula el porcentaje de error (Ec. 3.17), el error medio (Ec. 3.18) y la desviación media (Ec. 3.19) de la solución numérica en los mismos puntos discretos, respecto a los mismos puntos en donde se calculó la solución analítica (Tabla 6-7).

$$\% \, Error = \left| \frac{Analítico - Numérico}{Analítico} \right| * 100 \qquad \text{(Ec. 3.17)}$$

$$\bar{X} = (\sum_{n=1}^{n} \% Error_n)/n \qquad \text{(Ec. 3.18)}$$

$$\delta^2 = (\sum_{n=1}^{n} (\% Error - \bar{X})^2)/n - 1 \qquad \text{(Ec. 3.19)}$$

Tabla 6. Matriz de porcentaje de error numérico 12x12 vs analítico.

| X/Y | 0.05 | 0.15 | 0.25 | 0.35 | 0.45 | 0.55 | 0.65 | 0.75 | 0.85 | 0.95 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0.05 | 0.594131 | 1.619468 | 2.29385 | 1.339506 | 1.782463 | 1.782463 | 1.339506 | 2.29385 | 1.619468 | 0.594131 |
| 0.15 | 1.783052 | 4.580564 | 6.236446 | 3.674878 | 4.715475 | 4.715475 | 3.674878 | 6.236446 | 4.580564 | 1.783052 |
| 0.25 | 3.216314 | 7.626619 | 9.907439 | 6.071333 | 7.528885 | 7.528885 | 6.071333 | 9.907439 | 7.626619 | 3.216314 |
| 0.35 | 0.992667 | 1.986528 | 1.767234 | 7.02925 | 5.519036 | 5.519036 | 7.02925 | 1.767234 | 1.986528 | 0.992667 |
| 0.45 | 2.588867 | 5.036679 | 5.355212 | 3.894916 | 2.054217 | 2.054217 | 3.894916 | 5.355212 | 5.036679 | 2.588867 |
| 0.55 | 4.750327 | 8.607005 | 9.167719 | 0.293661 | 1.655782 | 1.655782 | 0.293661 | 9.167719 | 8.607005 | 4.750327 |
| 0.65 | 7.869625 | 12.90953 | 13.24194 | 3.763393 | 5.460234 | 5.460234 | 3.763393 | 13.24194 | 12.90953 | 7.869625 |
| 0.75 | 12.72003 | 18.27743 | 17.47052 | 7.992248 | 9.163313 | 9.163313 | 7.992248 | 17.47052 | 18.27743 | 12.72003 |
| 0.85 | 7.293411 | 1.197798 | 1.14252 | 6.77262 | 5.33595 | 5.33595 | 6.77262 | 1.14252 | 1.197798 | 7.293411 |
| 0.95 | 0.11461 | 2.045034 | 0.329604 | 1.902719 | 1.57843 | 1.57843 | 1.902719 | 0.329604 | 2.045034 | 0.11461 |

$\bar{X} = 4.65\%$ $\delta = 2.07\%$

Tabla 7. Matriz de porcentaje de error numérico 120x120 vs analítico.

| X/Y | 0.05 | 0.15 | 0.25 | 0.35 | 0.45 | 0.55 | 0.65 | 0.75 | 0.85 | 0.95 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0.05 | 0.300031 | 0.579904 | 0.795852 | 0.920439 | 0.953373 | 0.953373 | 0.920439 | 0.795852 | 0.579904 | 0.300031 |
| 0.15 | 0.6534 | 0.824783 | 0.924654 | 0.938229 | 0.877226 | 0.877226 | 0.938229 | 0.924654 | 0.824783 | 0.6534 |
| 0.25 | 0.996229 | 1.093384 | 1.076143 | 1.005747 | 0.887204 | 0.887204 | 1.005747 | 1.076143 | 1.093384 | 0.996229 |
| 0.35 | 1.447742 | 1.385218 | 1.258158 | 1.114835 | 0.937351 | 0.937351 | 1.114835 | 1.258158 | 1.385218 | 1.447742 |
| 0.45 | 1.95028 | 1.678902 | 1.435945 | 1.209183 | 0.990407 | 0.990407 | 1.209183 | 1.435945 | 1.678902 | 1.95028 |
| 0.55 | 2.578908 | 2.007939 | 1.609093 | 1.296691 | 1.039604 | 1.039604 | 1.296691 | 1.609093 | 2.007939 | 2.578908 |
| 0.65 | 3.381278 | 2.348685 | 1.744127 | 1.334993 | 1.047247 | 1.047247 | 1.334993 | 1.744127 | 2.348685 | 3.381278 |
| 0.75 | 4.473682 | 2.674908 | 1.796322 | 1.323991 | 1.058924 | 1.058924 | 1.323991 | 1.796322 | 2.674908 | 4.473682 |
| 0.85 | 6.077335 | 2.888751 | 1.707419 | 1.204645 | 0.984207 | 0.984207 | 1.204645 | 1.707419 | 2.888751 | 6.077335 |
| 0.95 | 8.557645 | 2.340673 | 1.33361 | 1.025265 | 0.794076 | 0.794076 | 1.025265 | 1.33361 | 2.340673 | 8.557645 |

$\bar{X} = 1.21\%$ $\delta = 1.19\%$

### 5.3.3 Análisis de resultados.

- Los resultados mostrados en la figura demuestran que el código computacional generado funciona satisfactoriamente, debido a la consistencia obtenida al refinar el mallado, esto es demostrado cuando al aumentar el número de volúmenes de control en el mallado de 144 a 14,400 el error medio de la solución numérica respecto a la analítica pasa de 4.65% a 1.21%.

# 6. DIFUSIÓN-CONVECCIÓN DE CALOR.

## 6.1. DISCRETIZACIÓN.

En la discretización de la ecuación de convección-difusión como en el anterior capítulo, se desarrollará la discretización de cada término en ecuación de energía (Ec.1.5) introduciendo el término convectivo, además se discretizará la ecuación de continuidad.

Ecuación de energía

$$\frac{\partial(\rho C_p T)}{\partial t} + \nabla \cdot \left(\rho \vec{v} C_p T\right) = \nabla \cdot (K \nabla T) + S_h \qquad \text{(Ec.4.1)}$$

Ecuación de continuidad

$$\frac{\partial(\rho)}{\partial t} + \nabla \cdot (\rho \vec{v}) = 0 \qquad \text{(Ec.4.2)}$$

Para la ecuación de energía se transforma la ecuación 4.1 a su forma integral, y agrupando el término difusivo-convectivo bajo la misma integral.

$$\int_t^{t+\Delta t} \int_V \frac{\partial \rho \phi}{\partial t} dV dt + \int_t^{t+\Delta t} \int_S (\rho \vec{v} \phi - \Gamma \nabla \phi) \cdot d\overrightarrow{A}\, dt = \int_t^{t+\Delta t} \int_V S_h dV dt \qquad \text{(Ec.4.3)}$$

Los términos temporal y fuente serán discretizados de la misma forma que se desarrolló en el capítulo anterior. El término difusivo-convectivo sería el único término faltante de la ecuación de energía, se procede a discretizar este término de la siguiente forma:

$$\int_t^{t+\Delta t} \int_S (\rho \vec{v} \phi - \Gamma \nabla \phi) \cdot d\overrightarrow{A}\, dt = \int_{\Delta t}^{t+\Delta t} \sum_{v=e,w,n,s} (J_v A_v) dt \qquad \text{(Ec.4.4a)}$$

$$\int_{\Delta t}^{t+\Delta t} \sum_{v=e,w,n,s} (J_v A_v) dt = \sum_{v=e,w,ns} [\theta J_v^1 + (1-\theta) J_v^0] A_v \Delta t \qquad \text{(Ec.4.4b)}$$

Reemplazando el término temporal, difusivo-convectivo y fuente en la ecuación 4.1 se obtiene:

$$\frac{\rho C_p \Delta x \Delta y (T_p^1 - T_p^0)}{\Delta t} = \sum_{v=e,w,ns} [\theta J_v^1 + (1-\theta) J_v^0] A_v \Delta t + [\theta S_h^1 + (1-\theta) S_h^0] V_p \quad \text{(Ec.4.5)}$$

Antes de resolver la ecuación de continuidad se introducirán dos conceptos que son: la intensidad de convección $F = \rho u$ , e intensidad de difusión $D = \frac{k}{\delta x C_p}$, el cociente de ambas expresiones se define como el número de Péclet que relaciona la intensidad de convección y la intensidad de difusión en el transporte del escalar, este término también es igual al numeró de Reynolds por el número de Prandtl.

$$P_e = \frac{F}{D} = \frac{\rho u \delta x C_p}{k} = Re_l \cdot Pr \quad \text{(Ec.4.6)}$$

Introducidos estos términos se discretiza la ecuación de continuidad

$$\int_t^{t+\Delta t} \int_V \frac{\partial \rho}{\partial t} dV dt + \int_t^{t+\Delta t} \int_S (\rho \vec{v}) \cdot d\overrightarrow{A} dt = \frac{C_p \Delta x \Delta y (\rho^1 - \rho^0)}{\Delta t} + [\theta F_e^1 + (1-\theta) F_e^0]$$

$$-[\theta F_w^1 + (1-\theta) F_w^0] + [\theta F_n^1 + (1-\theta) F_n^0] - [\theta F_s^1 + (1-\theta) F_s^0] \quad \text{(Ec.4.7)}$$

Multiplicando la ecuación 4.7 por $T_p$ y restándola a la ecuación 4.5 se obtiene:

$$\frac{\Delta x \Delta y \rho^0 (T_p^1 - T_p^0)}{\Delta t} + [\theta (J_e^1 - F_e^1 T_p^1) + (1-\theta)(J_e^0 - F_e^0 T_p^0)]$$

$$-\left[\theta (J_w^1 - F_w^1 T_p^1) + (1-\theta)(J_w^0 - F_w^0 T_p^0)\right] + [\theta (J_n^1 - F_n^1 T_p^1) + (1-\theta)(J_n^0 - F_n^0 T_p^0)]$$

$$-[\theta (J_s^1 - F_s^1 T_p^1) + (1-\theta)(J_s^0 - F_s^0 T_p^0)] + [\theta S_h^1 + (1-\theta) S_h^0] V_p \quad \text{(Ec.4.8)}$$

Donde

$$J_e^1 - F_e^1 T_p^1 = a_e (T_P - T_E) = (D_e A(|Pe|) + max[-F_e, 0])(T_P^1 - T_E^1)$$

$$J_e^0 - F_e^0 T_p^0 = a_e(T_P^0 - T_E^0) = (D_e A(|Pe|) + max[-F_e, 0])(T_P^0 - T_E^0)$$

El anterior procedimiento se realiza de forma análoga para los demás términos de la ecuación 4.8, dando como resultado la ecuación discretizada de la difusión-convección para los nodos internos y para distintos esquemas de solución, donde $A(|Pe|)$ es una función para los distintos esquemas de solución como se observa en la Tabla 8.

$$a_p T_p^1 = \sum_{v=e,w,n,s} a_v T_v + b \qquad \text{(Ec.4.9)}$$

$a_e = (D_e A(|Pe|) + max[-F_e, 0])$ $\quad$ $a_w = (D_w A(|Pe|) + max[F_w, 0])$

$a_n = D_n A(|Pe|) + max[-F_n, 0])$ $\quad$ $a_s = (D_s A(|Pe|) + max[F_s, 0])$

$a_p = a_n + a_s + a_w + a_e + a_0$ $\quad$ $a_0 = \frac{\rho^0 \Delta x \Delta y}{\Delta t}$

$b = (1-\theta)\sum_{v=e,w,ns}[D_v A(|Pe|)(T_v^0 - T_P^0) + max(F_n, 0)T_v^0 + max(-F_n, 0)T_P^0)] + [\theta S_h^1 + (1-\theta)S_h^0]V_p + a_0 T_p^0$

Tabla 8.Funciones A(|Pe|) para distintos esquemas.

| **Esquema** | **Función para $A(\|Pe\|)$** |
|---|---|
| Diferencia centrada (CDS) | $1 - 0.5\|P_e\|$ |
| Upwind (UDS) | 1 |
| Hibrido | $max[0, 1 - 0.5\|P_e\|]$ |
| Potencia | $max[0, (1 - 0.1\|P_e\|)^5]$ |
| Exponencial | $\frac{\|P_e\|}{e^{\|P_e\|} - 1}$ |

**6.1.1 Condiciones de frontera** para una frontera tipo Dirichlet donde $\boldsymbol{T_c}$ sea un valor de frontera conocido, por ejemplo, si $\boldsymbol{T_w = T_c}$ :

$$a_p T_p^1 = \sum_{v=e,n,s} a_v T_v + b \qquad \text{(Ec.4.10)}$$

$a_e = (D_e A(|Pe|) + max[-F_e, 0])$ $\qquad$ $a_s = (D_s A(|Pe|) + max[F_s, 0])$

$a_n = D_n A(|Pe|) + max[-F_n, 0])$ $\qquad$ $a_c = (D_c A(|Pe|) + max[F_c, 0])$

$a_p = a_n + a_s + a_e + a_0 + a_c$ $\qquad$ $a_0 = \frac{\rho^0 \Delta x \Delta y}{\Delta t}$

$b = (1-\theta)\sum_{v=e,c,ns}[D_v A(|Pe|)(T_v^0 - T_P^0) + max(F_n, 0)T_v^0 + max(-F_n, 0)T_P^0)] +$
$a_0 T_p^0 + [\theta S_h^1 + (1-\theta)S_h^0]V_p + (D_c A(|Pe|) + max[-F_c, 0])T_c^1$

Finalmente para una frontera tipo Newman donde se conoce el valor del flujo de calor en la cara $q_c = \frac{\partial T}{dn}$ , por ejemplo el flujo en la cara w donde $Q_c = k\Delta y \frac{\partial T}{dx}$

$$a_p T_p^1 = \sum_{v=e,n,s} a_v T_v + b \qquad \text{(Ec.4.11)}$$

$a_e = (D_e A(|Pe|) + max[-F_e, 0])$ $\qquad$ $a_s = (D_s A(|Pe|) + max[F_s, 0])$

$a_n = D_n A(|Pe|) + max[-F_n, 0])$ $\qquad$ $a_c = F_c$

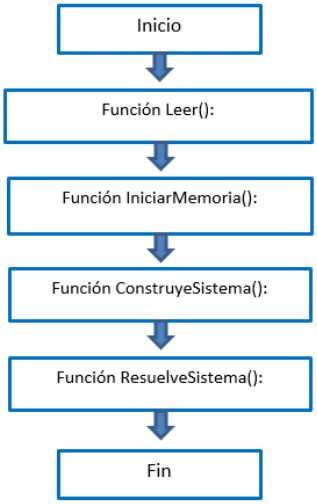
$a_p = a_n + a_s + a_e + a_0 + a_c$ $\qquad$ $a_0 = \frac{\rho^0 \Delta x \Delta y}{\Delta t}$

$b = (1-\theta)\sum_{v=e,c,ns}[D_v A(|Pe|)(T_v^0 - T_P^0) + max(F_n, 0)T_v^0 + max(-F_n, 0)T_P^0)] +$
$a_0 T_p^0 + [\theta S_h^1 + (1-\theta)S_h^0]V_p + k\Delta y \frac{\partial T}{dx}$

## 6.2. IMPLEMENTACIÓN DEL CODIGO COMPUTACIONAL.

Para la implementación del código computacional que permite solucionar de forma general la ecuación de convección-difusión de calor estacionaria y no-estacionaria, se creó la clase CD. La clase CD es derivada de las clases Puntero_Base que permite manipulación de memoria dinámica y GuardarEnsight que permite volcar los datos a Paraview para su visualización. Las funciones principales de la clase CD están esquematizadas de forma procedimental en la figura 15.

Figura 15. Clases principales de la clase CD.




- Función Leer():

Permite obtener los datos geométricos del dominio, propiedades físicas, condiciones de frontera e inicial, esquema temporal, tipo y tamaño de mallado del dominio, además el campo de velocidades para cada dimensión en el problema y el tipo de esquema de interpolación del término convectivo.

- Función IniciaMemoria():

Asigna los espacios de memoria para las variables involucradas en el problema, además define el tamaño de la matriz de coeficientes, vector de coeficientes y vector solución.

- Función ConstruyeSistema()

Calcula el flujo de calor difusivo y convectivo en las caras de cada volumen de control para generar el sistema de ecuaciones algebraicas.

- Función ResuelveProblema()

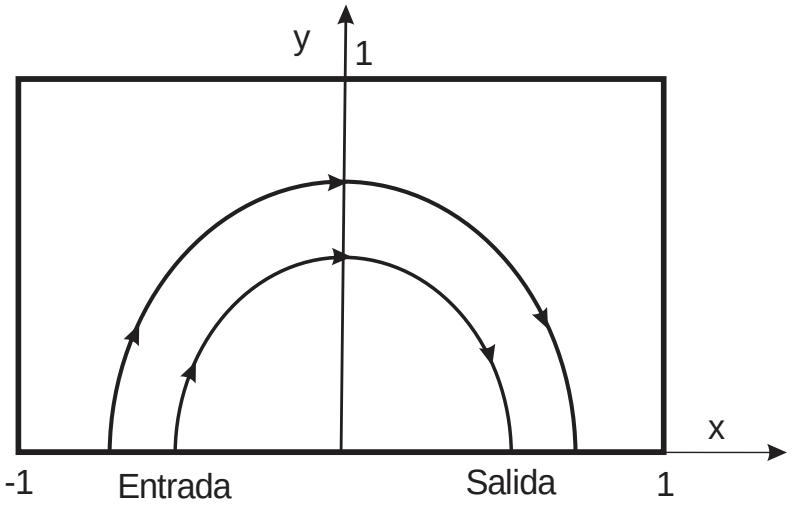
Soluciona el sistema de ecuaciones algebraicas generando el vector solución, calcula el gradiente en cada volumen de control y permite volcar los datos a Paraview para su visualización.

### 6.3. CASO DE ESTUDIO: SMITH HUTTON TEST.

El problema de Smith y Hutton [8], es un problema que sirve como test para validar el código computacional generado, comparando los resultados obtenidos de la clase CD con los valores de referencia del problema de Smith y Hutton.

El test de Smith y Hutton es un problema de convección-difusión en 2D estacionario, en donde se puede decir que el dominio está divido en dos partes, la parte izquierda está el flujo entrante y en la derecha el flujo saliente como se observa en la figura 16.

Figura 16. Problema de Smith-Hutton.




Campo de velocidades

$$u(x,y) = 2y(1-x^2)$$
$$v(x,y) = -2x(1-y^2)$$

Condiciones de frontera

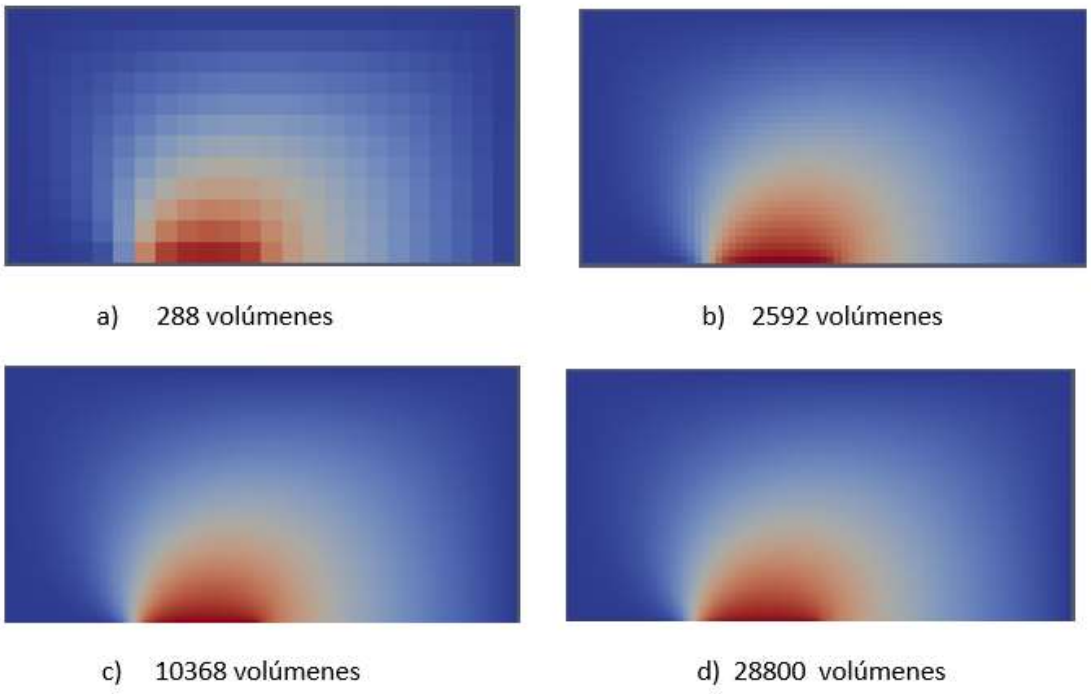
$$\phi = 1 + tanh\big(\alpha(2x+1)\big) \quad : \quad y = 0 \quad -1 \leq x \leq 0$$
$$\frac{\partial \phi}{\partial y} = 0 : \quad y = 0 \quad 0 \leq x \leq 1$$

$$\phi = 1 - tanh(\alpha) : \left\{ \begin{array}{c} x = -1 \quad 0 \leq y \leq 1 \\ x = 1 \quad 0 \leq y \leq 1 \\ y = 1 \quad -1 \leq x \leq 1 \end{array} \right\}$$

**6.3.1 Solución numérica** para validar los resultados se realizaron tres simulaciones con distintos valores $\boldsymbol{\rho/\Gamma = 10, 1E3, 1E6}$ debido a que en estos valores se realizaron los estudios [8] que serán de referencia más adelante.

Antes de realizar las tres simulaciones, se hizo un análisis de la independencia de la malla (figura 17), esto se hace con el fin de escoger una densidad de malla límite, en donde los resultados sean independientes a cualquier aumento de densidad superior a esta.

Figura 17.Independencia de la malla, esquema Upwind y ρ/Γ=10.



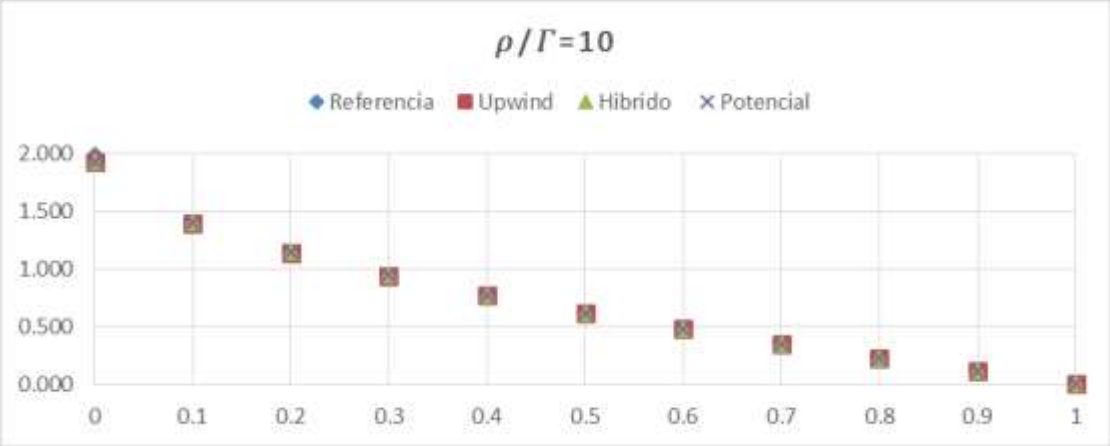

**6.3.2 Validación** en la validación los resultados se comparan con los valores de referencia de la variable $\boldsymbol{\phi}$ en la frontera de salida ($\boldsymbol{y = 0 \quad 0 \leq x \leq 1}$). Todas las simulaciones se realizaron con un mallado de 240x120 volúmenes y con distintos esquemas de interpolación espacial (Upwind, Hibrido y Potencial).

- Caso 1, $\rho/\Gamma = 10$.

Tabla 9. Valores de la variable ϕ en la frontera de salida ρ/Γ=10.

| POSICIÓN X | REFERENCIA | UPWIND | HIBRIDO | POTENCIAL |
|---|---|---|---|---|
| **0** | 1.989 | 1.9307 | 1.931 | 1.931 |
| **0.1** | 1.402 | 1.4021 | 1.4047 | 1.4047 |
| **0.2** | 1.146 | 1.1467 | 1.1498 | 1.1498 |
| **0.3** | 0.946 | 0.94752 | 0.95061 | 0.95058 |
| **0.4** | 0.775 | 0.77666 | 0.77933 | 0.77929 |
| **0.5** | 0.621 | 0.62365 | 0.62563 | 0.6256 |
| **0.6** | 0.48 | 0.48352 | 0.4847 | 0.48468 |
| **0.7** | 0.349 | 0.35355 | 0.35399 | 0.35398 |
| **0.8** | 0.227 | 0.23185 | 0.23178 | 0.23178 |
| **0.9** | 0.111 | 0.11644 | 0.11624 | 0.11624 |
| **1** | 0 | 0.004631 | 0.004619 | 0.004619 |

Figura 18. Variación de la variable ϕ en frontera de salida para ρ/Γ=10.

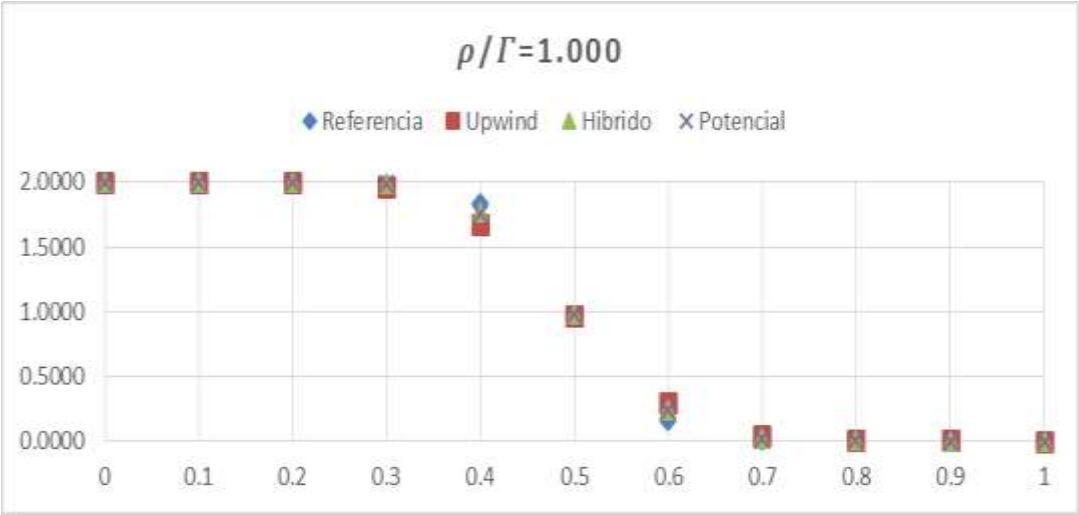


- Caso 2, $\rho/\Gamma = 1.000$.

Tabla 10. Valores de la variable ϕ en la frontera de salida ρ/Γ=1000.

| POSICIÓN X | REFERENCIA | UPWIND | HIBRIDO | POTENCIAL |
|---|---|---|---|---|
| **0** | 2.0000 | 2 | 2 | 2 |
| **0.1** | 1.9990 | 2 | 2 | 2 |
| **0.2** | 1.9997 | 1.998 | 2 | 2 |
| **0.3** | 1.9850 | 1.96 | 1.985 | 1.981 |
| **0.4** | 1.8410 | 1.677 | 1.765 | 1.751 |
| **0.5** | 0.951 | 0.964 | 0.981 | 0.9791 |
| **0.6** | 0.154 | 0.2912 | 0.232 | 0.2413 |
| **0.7** | 0.001 | 0.04019 | 0.01999 | 0.02241 |
| **0.8** | 0 | 0.002323 | 0.000649 | 0.000793 |
| **0.9** | 0 | 5.16E-05 | 9.51E-06 | 1.24E-05 |
| **1** | 0 | 1.43E-07 | 2.99E-08 | 3.78E-08 |

Figura 19. Variación de la variable ϕ en frontera de salida para ρ/Γ=1000.



- Caso 3, $\rho/\Gamma = 1'000.000$ .

Tabla 11. Valores de la variable ϕ en la frontera de salida ρ/Γ=1' 000.000.

| POSICIÓN X | REFERENCIA | UPWIND | HIBRIDO | POTENCIAL |
|---|---|---|---|---|
| **0** | 2.0000 | 2 | 2 | 2 |
| **0.1** | 2.0000 | 2 | 2 | 2 |
| **0.2** | 2.0000 | 1.9999 | 1.9999 | 1.9999 |
| **0.3** | 1.9990 | 1.9899 | 1.9899 | 1.9899 |
| **0.4** | 1.9640 | 1.7912 | 1.7913 | 1.7913 |
| **0.5** | 1.0000 | 0.98452 | 0.98454 | 0.98454 |
| **0.6** | 0.036 | 0.21396 | 0.21386 | 0.21386 |
| **0.7** | 0.001 | 0.015703 | 0.015683 | 0.015683 |
| **0.8** | 0 | 0.000415 | 0.000414 | 0.000414 |
| **0.9** | 0 | 4.65E-06 | 4.64E-06 | 4.64E-06 |
| **1** | 0 | 1.41E-08 | 1.41E-08 | 1.41E-08 |

Figura 20. Variación de la variable ϕ en frontera de salida para ρ/Γ=1' 000.000.

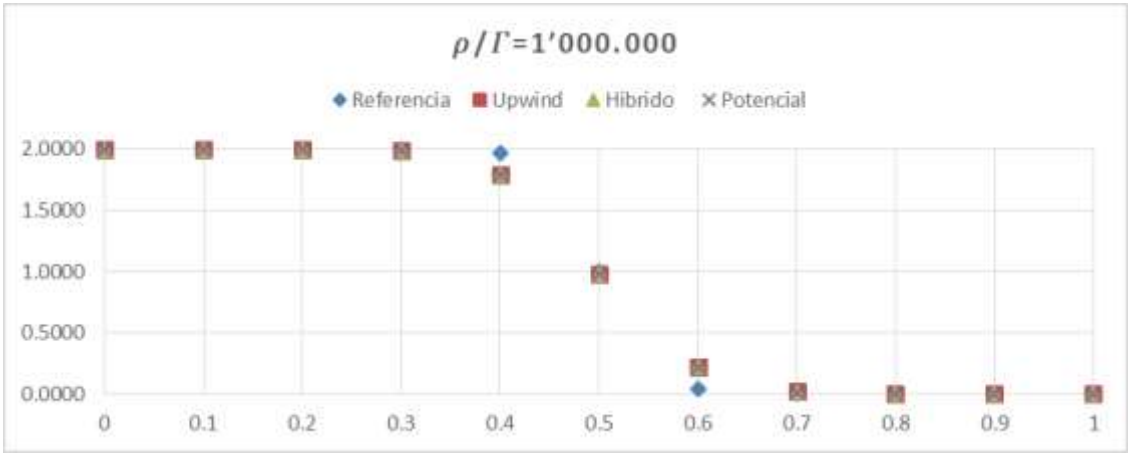
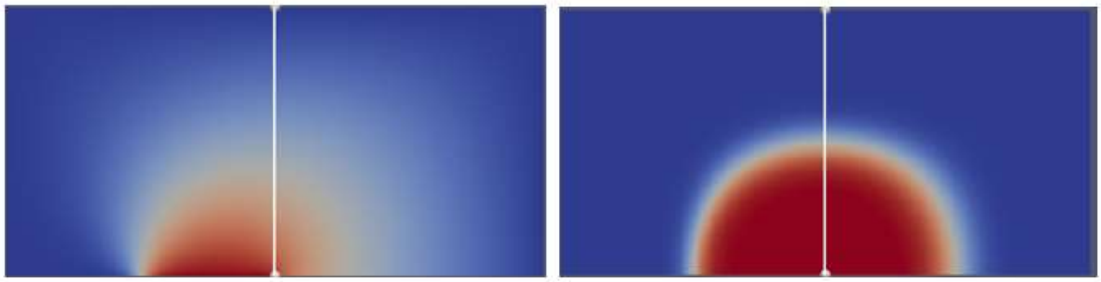


Figura 21. Concentración de la variable ϕ.



a) $\rho/\Gamma = 10$ b) $\rho/\Gamma = 1'.000.000$

### 6.3.3 Análisis de resultados.

- El problema de Smith-Hutton demostró que el código computacional simula satisfactoriamente los problemas de convección-difusión. Comparando los casos se puede concluir que la simulación numérica es más precisa cuando los números de Péclet locales son muy bajos o muy altos (figura 18 y figura 20). Mientras que cuando el número de Péclet local tiende a $\pm 2$, tanto el esquema Upwind como el Hibrido no son tan precisos como el esquema Potencial debido a que son esquemas de primer orden (Patankar 1980), por este motivo en el caso 2 (figura 19) los valores de $\phi$ en las soluciones varían mucho más que en los otros casos.

- En la figura 21 se puede concluir que cuando el número de Péclet es bajo (figura 21a). En el transporte de la variable $\phi$ el mecanismo predominate es la conducción, esto se puede ver comparando las concentraciones de la variable $\phi$ en la frontera de entrada y salida. Caso contrario cuando el número de Péclet es alto (figura 21b), en donde el mecanismo predominante es la convección, comparando gráficamente la concentración de la variable $\phi$ en la frontera de entrada y salida se puede ver que es muy similar.

## 7. SOLUCIÓN DEL CAMPO DE FLUJO.

La convección de una variable escalar $\phi$ depende de la magnitud y dirección de campo local de velocidades, en el anterior capítulo se asumió el campo de velocidades como conocido, pero generalmente el campo de velocidades es en realidad desconocido y su solución es una parte del problema total. En este capítulo se desarrollarán técnicas para la solución del campo flujo.

Reemplazando la variable escalar $\phi$ de la ecuación general de transporte (Ec.1.4) definida en el capítulo 1, por las velocidades $u$ y $v$, además asumiendo que el flujo es laminar en estado estable y que la cavidad es vertical $\theta = 90°$ , se obtienen las siguientes ecuaciones de conservación del momento.

Conservación de momento en x:

$$\frac{\partial \rho uu}{\partial x} + \frac{\partial \rho uv}{\partial y} = -\frac{\partial p}{\partial x} + \mu\left(\frac{d^2u}{dx^2} + \frac{d^2u}{dy^2}\right) + \rho g cos\theta \qquad \text{(Ec.5.1)}$$

Conservación de momento en y:

$$\frac{\partial \rho vu}{\partial x} + \frac{\partial \rho vv}{\partial y} = -\frac{\partial p}{\partial y} + \mu\left(\frac{d^2v}{dx^2} + \frac{d^2v}{dy^2}\right) + \rho g sen\theta \qquad \text{(Ec.5.2)}$$

Las velocidades $u$ y $v$ que aparecen en las ecuaciones de conservación de momento además deben satisfacer la ecuación de continuidad.

$$\frac{\partial \rho u}{\partial x} + \frac{\partial \rho v}{\partial y} = 0 \qquad \text{(Ec.5.3)}$$

Los principales obstáculos para hallar la solución del campo de flujo que satisfaga las ecuaciones 5.1, 5.2 y 5.3 son las siguientes:

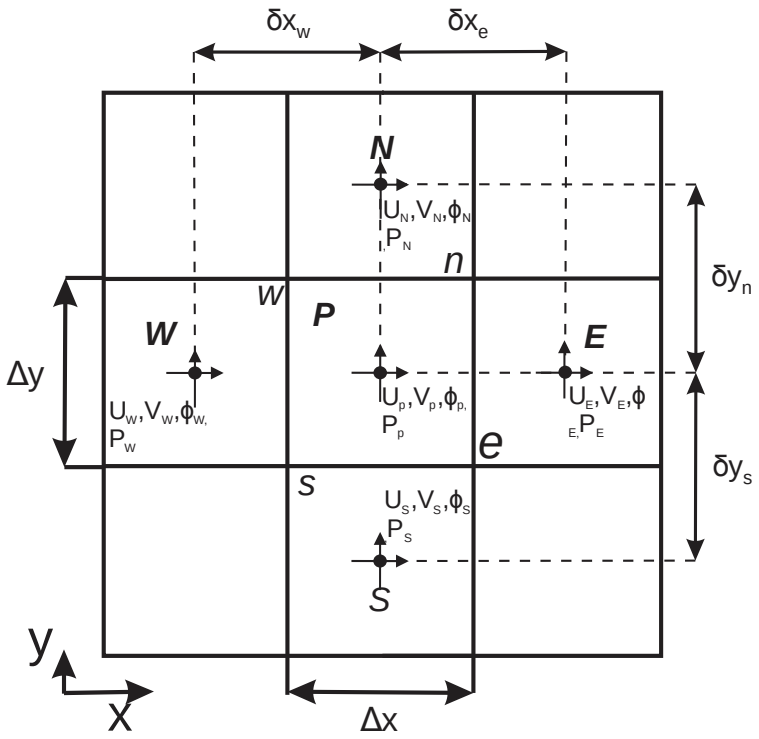
- La no-linealidad del término convectivo, por ejemplo el primer término de la ecuación 5.1 es una derivada respecto a x de $\rho u^2$.
- En las ecuaciones de momento, el término fuente (gradiente de presión) está en función de la presión. La dificultad radica en que la presión es una variable independiente y no existe una ecuación explicita que prediga su comportamiento.

Para poder sortear estos obstáculos y poder hallar la solución del campo de flujo (presión, temperatura y velocidades), se crearon métodos iterativos predictor-corrector de acoplamiento presión-velocidad como el método SIMPLE y sus evoluciones SIMPLEC y SIMPLER.

## 7.1. MALLADO COLOCADO.

Antes de discretizar las ecuaciones de momento y la de continuidad, se debe definir el tipo de mallado a usar. En esta tesis se usará un mallado colocado, el mallado colocado se caracteriza por que magnitud de las variables (velocidades, presión y temperatura) se almacena en el centroide del volumen de control como se observa en la figura 22.

Figura 22. Mallado colocado.




La gran ventaja de definir un mallado colocado es que al guardar todos los valores en los centroides de los volúmenes de control permite utilizar tanto mallados estructurados como no estructurados sin problema ninguno, además que con una sola malla se realizan todos los cálculos. Caso contrario del mallado desplazado en donde se tienen que definir dos mallados uno para guardar las presiones y la variable escalar a calcular $\phi$ , y otro para guardar las velocidades.

## 7.2. DISCRETIZACIÓN.

En este capítulo los términos convectivo-difusivo, temporal y fuente exceptuando el gradiente de presión se discretizará de la misma forma que en capítulo 4, por ello no vale la pena volver a realizar ese desarrollo. Para discretizar el gradiente de presión, como primera aproximación se asume que la presión en las caras es una interpolación lineal de los valores de presión en los nodos adyacentes, para la ecuación de momento en X:

$$\frac{\partial p}{\partial x} = \frac{p_e - p_w}{\Delta x} = \frac{(\frac{P_E + P_p}{2}) - (\frac{P_p + P_w}{2})}{\Delta x}$$

$$\frac{\partial p}{\partial x} = \frac{P_E - P_w}{2\Delta x}$$

De forma análoga para la ecuación de momento en Y

$$\frac{\partial p}{\partial y} = \frac{P_N - P_s}{2\Delta y}$$

- **Conservación de momento.**

Una vez discretizada el término del gradiente de presión, se introduce el término en la ecuación discretizada, para la conservación de momento en x:

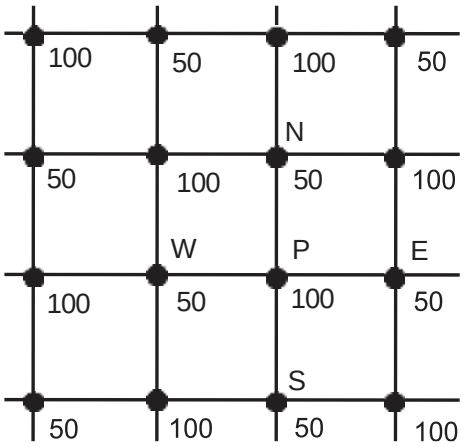
$$a_p u_p - \sum_{v=E,W} u_v T_v = A(P_E - P_w) + b \qquad \text{(Ec.5.4)}$$

Y de forma análoga para la conservación de momento en y

$$a_p v_p - \sum_{v=N,S} v_v T_v = A(P_N - P_S) + b \qquad \text{(Ec.5.5)}$$

**7.2.1. Problema chekerboarding** el fenómeno conocido chekerboarding, emerge cuando se está en presencia de un mallado colocado y además se asume que la presión en las caras es una interpolación lineal de los valores de presión en los nodos adyacentes. El fenómeno consiste en que si se calcula la velocidad en el nodo P en un campo de presiones de forma chekerboard (tablero de ajedrez) figura 23, calculando el gradiente de presión en ambas ecuaciones de momento da como resultado que el gradiente es igual a cero, dando una noción falsa de que el valor del campo de presiones es constante, cuando en realidad es un campo oscilatorio.

Figura 23. Campo de presiones tipo chekerboard.




Para evadir este problema existen dos soluciones que son:

- **Mallado desplazado.**

En el mallado desplazado los valores de la presión y de la variable extensiva se calculan en los centroides, mientras que las velocidades de calculan en las caras. Esta solución fue creada por Harlow y Welch [10], la desventaja de este mallado es que falla en mallas no-estructuradas, además que añade una mayor complejidad geométrica al problema, por estos motivos en esta tesis se decide no usar mallado desplazado.

- **Interpolación de Rhie-Chow.**

La interpolación de Rhie-Chow [10] es una forma de solucionar el problema de chekerboarding en el mallado colocado. Esta solución consiste en añadir un término de tercer orden del gradiente de presión (Ec.5.6), la adición de este término no compromete la precisión en la solución, en el caso contrario proporciona un amortiguamiento a la falsa oscilación de la presión debido al mallado colocado (Versteeg y Malalasekera 2007).

En la interpolación de Rhie-Chow la velocidad se interpola en las caras, por ejemplo, la frontera e del volumen de control p (figura 21) la velocidad queda determinada de la siguiente forma:

$$u_e = \frac{u_E+u_P}{2} + \frac{1}{2}\left(d_p + d_E\right)(p_P - p_E) - \frac{1}{4}d_p(p_w - p_E) - \frac{1}{4}d_E(p_P - p_{EE}) \quad \text{(Ec. 5.6)}$$

Donde:

$$d_n = \frac{A_n}{a_{PE}}$$

Si se asume que todos los valores de $d_n$ de los volúmenes son iguales y constante la velocidad en la cara e queda de siguiente manera:

$$u_e = \frac{u_E+u_P}{2} + \frac{d}{4}(p_{EE} - 3p_E + 3p_P - p_W) \quad \text{(Ec.5.7)}$$

Siendo en términos de la presión.

$$\Delta x^3 \left(\frac{\partial^3 p}{dx^3}\right)_e = (p_{EE} - 3p_E + 3p_P - p_W) \quad \text{(Ec.5.9)}$$

Reemplazando la ecuación 5.9 en la ecuación 5.6.

$$u_e = \frac{u_E+u_P}{2} + \frac{d}{4}\left(\frac{\partial^3 p}{dx^3}\right)_e \Delta x^3 \quad \text{(Ec.5.10)}$$

Para las otras caras se aplica el mismo procedimiento. Debido a la implementación en mallado colocalo se escoge la interpolación de Rhie-Chow como método de discretización de las ecuaciones de momento en esta tesis.

## 7.3. MÉTODO SIMPLE.

El método SIMPLE (Semi-Implicit Method for Pressure Linked Equation) es un algoritmo iterativo para solucionar el campo de flujo creado por Brian Spalding y Suhas Patankar [11], antes de conocer el algoritmo se introducen dos conceptos que son la presión corregida y la velocidad corregida

$$p^{k+1} = p^k + \alpha p' \qquad v^{k+1} = v^k + \alpha v' \qquad \text{(Ec.5.10-5-11)}$$

$p^{k+1}$= presión corregida.

$p^k$= presión supuesta.

$\alpha$= factor de subrelajación.

$p'$= corrección de presión.

$v^{k+1}$= velocidad corregida.

$v^k$= velocidad supuesta.

$\alpha$= factor de subrelajación.

$v$= corrección de velocida

### 7.3.1 Secuencia algoritmo simple.

1. Suponer el campo de presión $p^k$.
2. Resolver las ecuaciones de momento, para obtener las velocidades supuestas $u^k, v^k$y $w^k$.
3. Resolver la ecuación de corrección de presión $p'$.
4. Calcular la presión corregida $p^{k+1}$ sumando $p^k$ y $\alpha p'$.
5. Calcular $u^{k+1}, v^{k+1}$y $w^{k+1}$ de la ecuación de velocidad corregida.
6. Resolver la ecuación discretizada de la cantidad escalar $\phi$ a calcular (temperatura, turbulencia, concentración, etc.).
7. Reemplazar la presión corregida $p^k$ como la nueva presión supuesta $p^{k+1}$, volver al punto 2 y repetir el procedimiento hasta que se cumpla el criterio de convergencia supuesto.

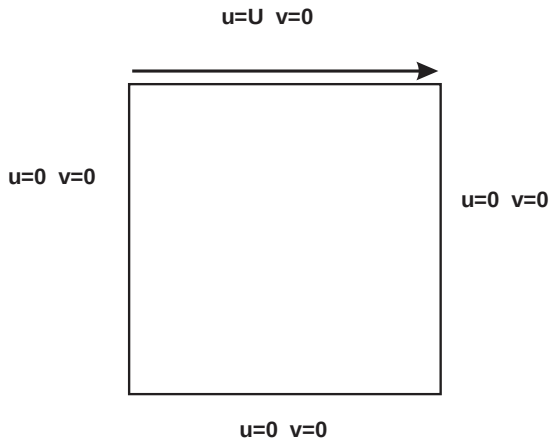
## 7.4. IMPLEMENTACIÓN DEL CÓDIGO COMPUTACIONAL.

Para la implementación del código computacional que permite solucionar los problemas de flujo de forma general se creó la clase NS. La clase NS es derivada de las clases Puntero_Base que permite manipulación de memoria dinámica y GuardarEnsight que permite volcar los datos a Paraview para su visualización. La clase NS además es una clase amiga de las clases CD y CDifusion, en este problema la clase CD permite la solución de las ecuaciones de momento y la clase CDifusion permite la solución de la ecuación de continuidad o corrección de presión.

## 7.5. CASO DE ESTUDIO: LID DRIVEN CAVITY.

Lid driven cavity es un problema que sirve como punto de referencia para validar programas que solucionen el campo de flujo. El caso estándar es de una cavidad cuadrada, en donde la frontera superior solo la velocidad tangente a la superficie tiene un valor mayor de cero (u=U) mientras la velocidad perpendicular es estática (v=0) como las otras tres fronteras (u=0, v=0) como se muestra en la figura 24.

Figura 24. Caso estándar lid driven cavity.

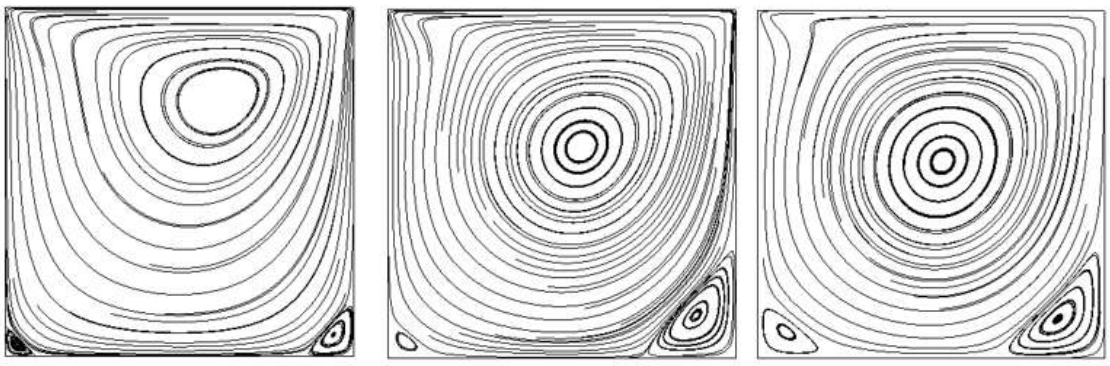


Este problema se ha estudiado tanto para flujos laminares como turbulento. Ghia et al. [9] Simularon para distintos número de Reynolds, obteniendo la velocidad central en X y Y y la vorticidad para cada caso. En el problema resuelto por Ghia et al. La velocidad superior (U=1), la densidad ($\rho = 1$) y el valor de los lados de la cavidad cuadrada ($L = 1$) son valores constantes, mientras que la viscosidad $v$ varía según el numero de Reynolds $Re$.

$$Re = \frac{\rho UL}{v} \qquad \text{(Ec.5.12)}$$

**7.5.1 Solución numérica** la solución numérica del problema de Lid driven cavity se realizó para números de Reynolds = 100, 400 y 1000 debido a que estos valores pertenecen al régimen laminar y fueron simulados en el estudio de Ghia et al. Todas las simulaciones fueron realizadas con un mallado de 60x60 y 120x120 volúmenes para comprobar la independencia de la malla y además porque el estudio de Ghia et al se realizó con un mallado de 129x129 celdas. Las ecuaciones de convección difusión fueron resultas con el esquema de interpolación Potencial y factores de relajación para la corrección de presión $\boldsymbol{\alpha_p = 0.4}$ y correcion de la velocidades $\boldsymbol{u}$ y $\boldsymbol{v}$ de $\boldsymbol{\alpha_v = 0.7}$ . En la figura 25 se puede observar las líneas de corrientes que se dieron como resultado de la simulación de 120x120 volúmenes.

Figura 25. Líneas de corriente simulación.



a) Re=100 b) Re=400 c) Re=1000

**7.5.2 Validación** Para realizar la validación de las simulaciones se compararon los valores de las velocidades $\boldsymbol{u}$ y $\boldsymbol{v}$ centrales con los datos proporcionados del estudio de Ghia el al. Además se calcularon los errores relativos.

Debido a que las funciones del valor de $u$ y $v$ centrales es discreta, Paraview tiene un filtro (cell point to data) que interpola los valores entre los puntos discretos y genera una gráfica continua como se observa en las figuras 26-28, esto se hace como una estrategia de Post-procesamiento para poder comparar los valores de la simulación contra los valores de referencia. Los errores relativos de la Tabla 12-14, fueron calculados con las funciones discretas originales.

Figura 26. Solución numérica Re=100.

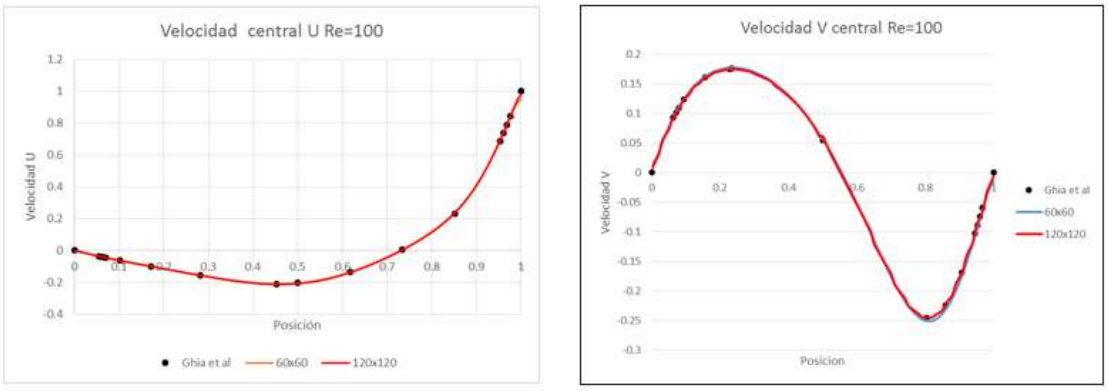



Tabla 12. Errores relativos Re=100.

| Re=100 | Error % | | | |
|---|---|---|---|---|
| | Umax | Umin | Vmax | Vmin |
| 60x60 | 4.833 | 1.725936 | 1.038398 | 2.258183 |
| 120x120 | 2.392 | 1.147463 | 0.439322 | 0.163046 |

Figura 27. Solución numérica  Re=400.

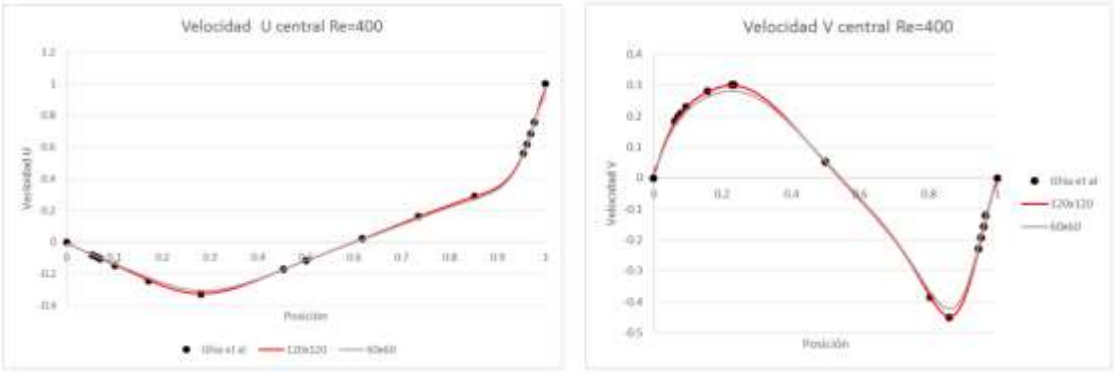



Tabla 13.  Errores relativos Re=400.

| Re=400 | Error % | | | |
|---|---|---|---|---|
| | Umax | Umin | Vmax | Vmin |
| 60x60 | 7.786 | 6.921102 | 6.943019 | 6.060943 |
| 120x120 | 3.705 | 1.038929 | 1.072741 | 0.613429 |

Figura 28. Solución numérica Re=1000.

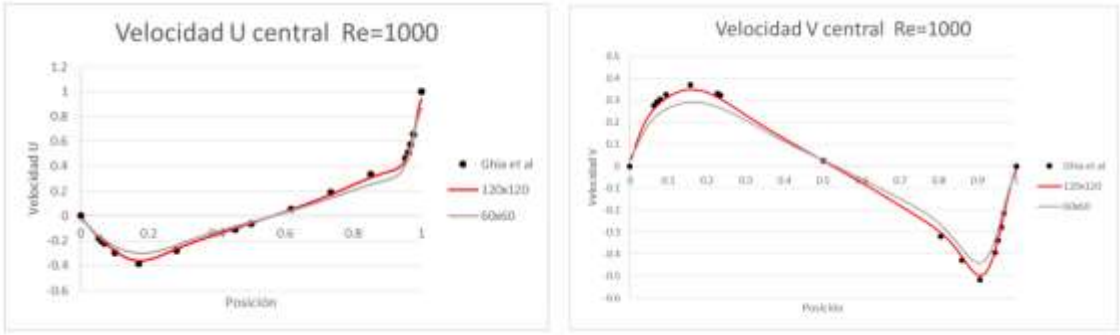



Tabla 14. Errores relativos Re=1000.

| Re=1000 | Error % | | | |
|---|---|---|---|---|
| | Umax | Umin | Vmax | Vmin |
| 60x60 | 12.863 | 21.74515 | 21.64443 | 14.87488 |
| 120x120 | 5.705 | 5.944266 | 5.863324 | 3.612027 |

### 7.5.3 Análisis de resultados

- La simulación del problema de Lid driven cavity demuestra que el código computacional resuelve satisfactoriamente los problemas de campo de flujo laminar. Como se observan en las tablas 12-14, a medida que se aumenta el número de Reynolds para una densidad de malla constante, también se aumenta el error relativo respecto a la solución proporcionada por Ghia et al, para solucionar este problema se recomienda aumentar la densidad de malla si se quiere disminuir el error relativo para números de Reynolds altos.

- Una vez validado el programa se procede a graficar las líneas de corriente para cada problema (figura 25), como se puede observar a medida que se aumenta el número de Reynolds la viscosidad disminuye, eso se evidencia en la figura 25a en donde para un Re=100 la alta viscosidad permite que la energía proporcionada en el frontera superior sea disipada en gran parte antes de llegar al fondo de la cavidad, por ello el tamaño de los vórtices es menor comparado con el Re=400 (figura25b) y Re=1000 (figura 25c) donde una menor viscosidad permite que una cantidad mayor de energía llegue al fondo de cavidad generando unos vórtices de mayor tamaño.

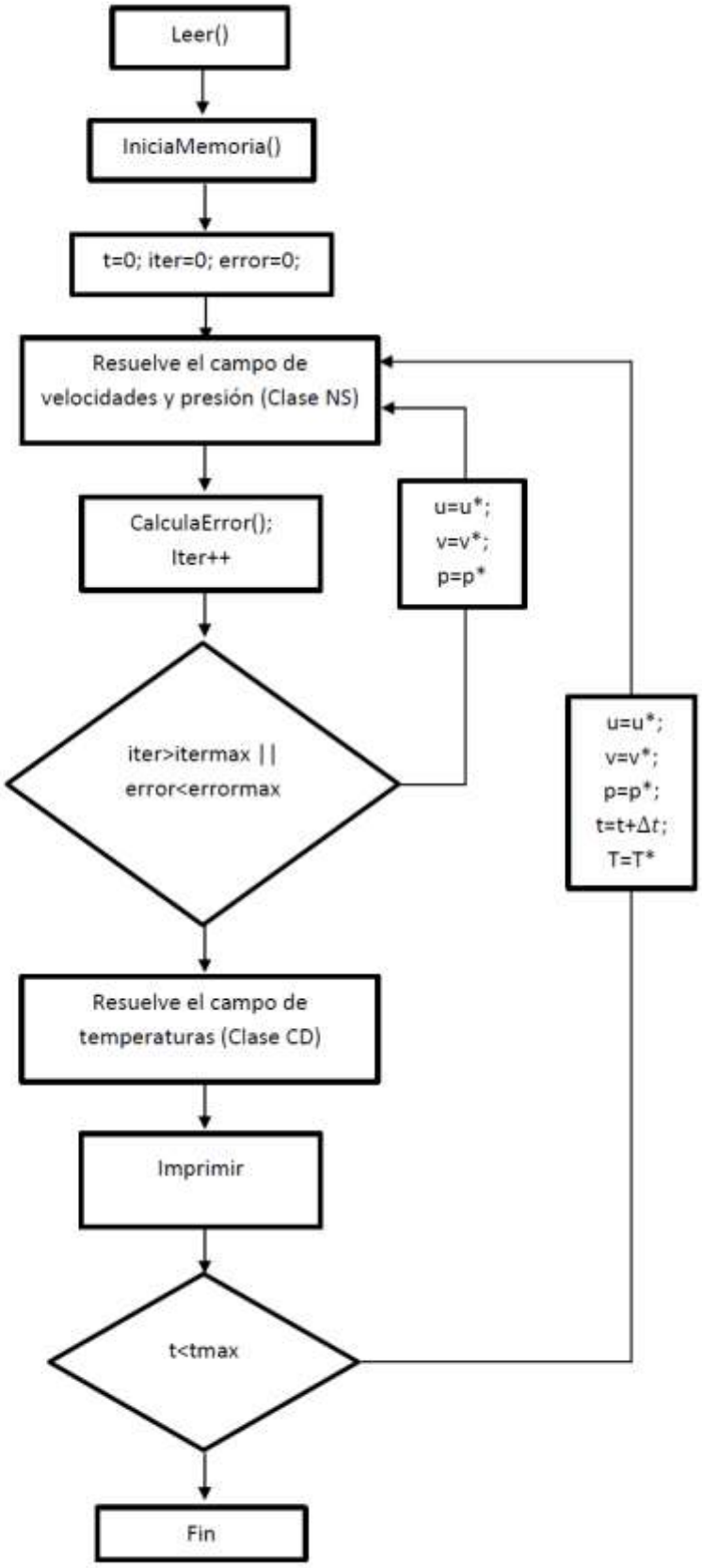
# 8. CAVIDAD RECTANGULAR BAJO CONDICIONES DE CONVECCION NATURAL.

Como se explicó en el Capítulo 1, la solución del campo de flujo de una cavidad rectangular bajo condiciones de convección natural, implica el acople de una ecuación de energía a las ecuaciones de campo de flujo solucionadas en el Capítulo 5. Además del acople de la ecuación de energía, en las ecuaciones de momento aparece un término de generación debido al movimiento influenciado por la diferencia de densidades en el fluido.

## 8.1. IMPLEMENTACIÓN DEL CÓDIGO COMPUTACIONAL.

Para la implementación del código computacional, que permite solucionar los problemas del campo de flujo en una cavidad rectangular bajo convección natural de forma general, se creó la clase BDC. La clase BDC es derivada de las clases Puntero_Base que permite manipulación de memoria dinámica y GuardarEnsight que permite volcar los datos a Paraview para su visualización. La clase BDC además es una clase amiga de las clases NS y CD, en este problema la clase NS permite la solución de las ecuaciones de momento y continuidad, la clase CD permite la solución de la ecuación de energía. En la figura 29 se explica las funciones principales de la clase BDC.

Figura 29. Funciones principales de la clase BDC.



## 8.2. CASO DE ESTUDIO: BOUYANCY DRIVEN CAVITY.

El problema de bouyancy driven cavity es un caso de referencia que sirve para validar programas que simulen convección natural, este problema consiste en una cavidad cuadrada $L = H = 1\,m$ con una inclinación de la cavidad de $\theta = 90°$, en donde se encuentra aire confinado con un $Pr = 0.71$ , todas las propiedades se calculan a $300\,K$, la diferencia de temperaturas entre las paredes debe ser de $\Delta T = 1\,K$ y la gravedad $g$ se ajusta según el número de Rayleigh deseado.

$$Ra = \frac{C_p \rho g \beta (T_1 - T_c) L^3}{\mu k}$$

Tabla 15. Propiedades del aire a 300 K.

| **Propiedades** | **Valor** |
|---|---|
| Densidad de referencia $(\rho_0)$ | 1.1614$[\frac{kg}{m^3}]$ |
| Calor especifico $(C_p)$ | 1007$[\frac{J}{kg \cdot K}]$ |
| Viscosidad $(\mu)$ | 1.85E-05$[\frac{N \cdot s}{m^2}]$ |
| Conductividad $(k)$ | 2.64E-02$[\frac{W}{m \cdot K}]$ |

Los resultados obtenidos (Nusselt promedio), se validarán con correlaciones empíricas creadas por Catton I. [3] (Ec.6.1) y la solución numérica de G. De Vahl Davis [12] obtenida mediante funciones de corriente-vorticidad y diferencias finitas, para una cavidad cuadrada bajo condiciones convección natural, con un numero de Prantdl de 0.71 y Rayleigh de 1.000 y 1’000.000.

$$\overline{Nu_L} = 0.18\left(\frac{Pr}{0.2+Pr}Ra_L\right)^{0.29} \quad \text{(Ec.6.1)}$$

$$\begin{pmatrix} 1 < \frac{H}{L} < 2 \\ 10^{-3} < Pr < 10^5 \\ 10^3 < \frac{Ra_L Pr}{0.2+Pr} \end{pmatrix}$$

Además se harán dos validaciones más, una para distintos factores de forma $\frac{H}{L} = 2.5\,, 5\; y\; 10$ para la ecuación empírica creada por Catton I (Ec.6.2), Para Ra=$10^4$ y Pr=0.71. La otra valiadacion variando el ángulo $\theta = 30°, 45°, 60°, 75°\; y\; 90°$ para la correlacion creada por Ayyaswasmy P. y Catton I. [3] (Ec.6.3), para H/L=1, Ra=$10^5$ y Pr=0.71.

$$\overline{Nu_L} = 0.22\left(\frac{Pr}{0.2+Pr}Ra_L\right)^{0.28}\left(\frac{H}{L}\right)^{-\frac{1}{4}} \quad \text{(Ec.6.2)}$$

$$\begin{pmatrix} 2 < \frac{H}{L} < 10 \\ Pr < 10^5 \\ 10^3 < Ra < 10^{10} \end{pmatrix}$$

$$\overline{Nu_L} = \overline{Nu_L}(\theta = 90)(sen\theta)^{\frac{1}{4}} \quad \text{(Ec.6.3)}$$

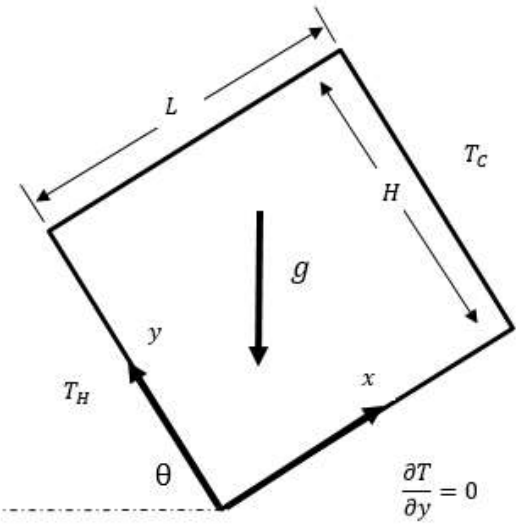
$$(\theta^* < \theta < 90)$$

Donde:

Tabla 16. Ángulos críticos para distintos factores de forma H/L.

| H/L | 1 | 3 | 6 | 12 | >12 |
|---|---|---|---|---|---|
| $\theta^*$ | 25° | 53° | 60° | 67° | 70° |

Figura 30. Cavidad bajo convección natural.




**8.2.1 Solución numérica** para validar los resultados se hicieron 4 simulaciones para números de Rayleigh $\boldsymbol{Ra = 10^3, 10^4, 10^5\ y\ 10^6}$. En todas las simulaciones se empleó una malla de 120x120 volúmenes, además las ecuaciones de convección difusión fueron resueltas con el esquema de interpolación Potencial y factores de relajación para la corrección de presión $\boldsymbol{\alpha_p = 0.4}$ y correcion de la velocidades $\boldsymbol{u}$ y $\boldsymbol{v}$ de $\boldsymbol{\alpha_v = 0.7}$.

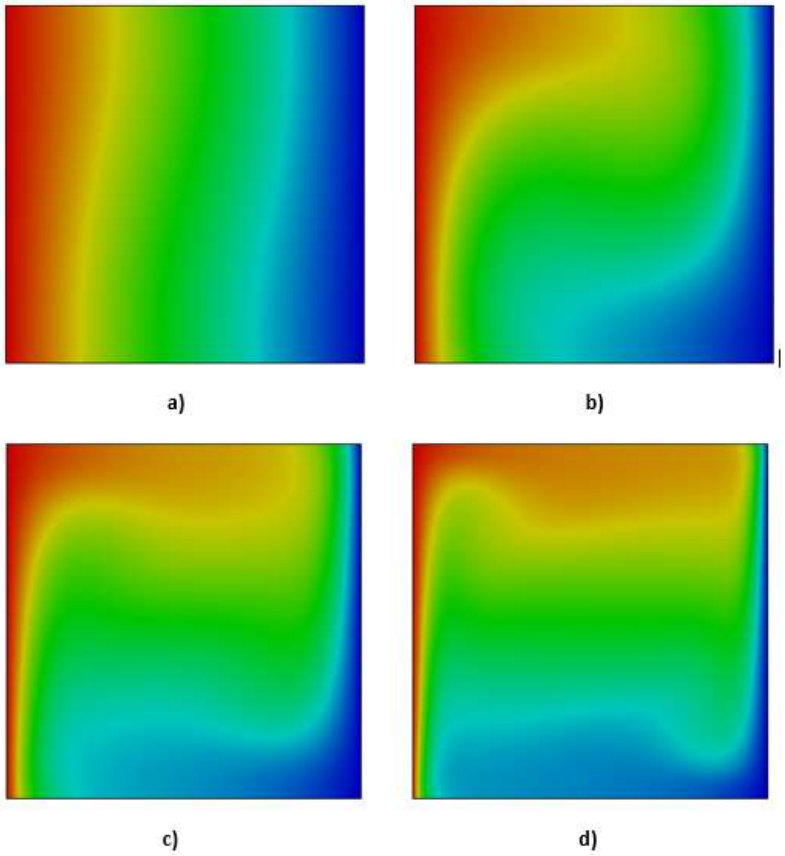
Para calcular el número de Nuselt local se utilizó la ecuación 6.4 y el número de Nusselt promedio la ecuación 6.5.

$$Nu_L = \frac{hL}{k} = -\int_0^1 Q(x,y)dy \qquad \text{(Ec.6.4)}$$

$$\overline{Nu} = \frac{1}{H}\int_0^H Nu_L dy \qquad \text{(Ec.6.5)}$$

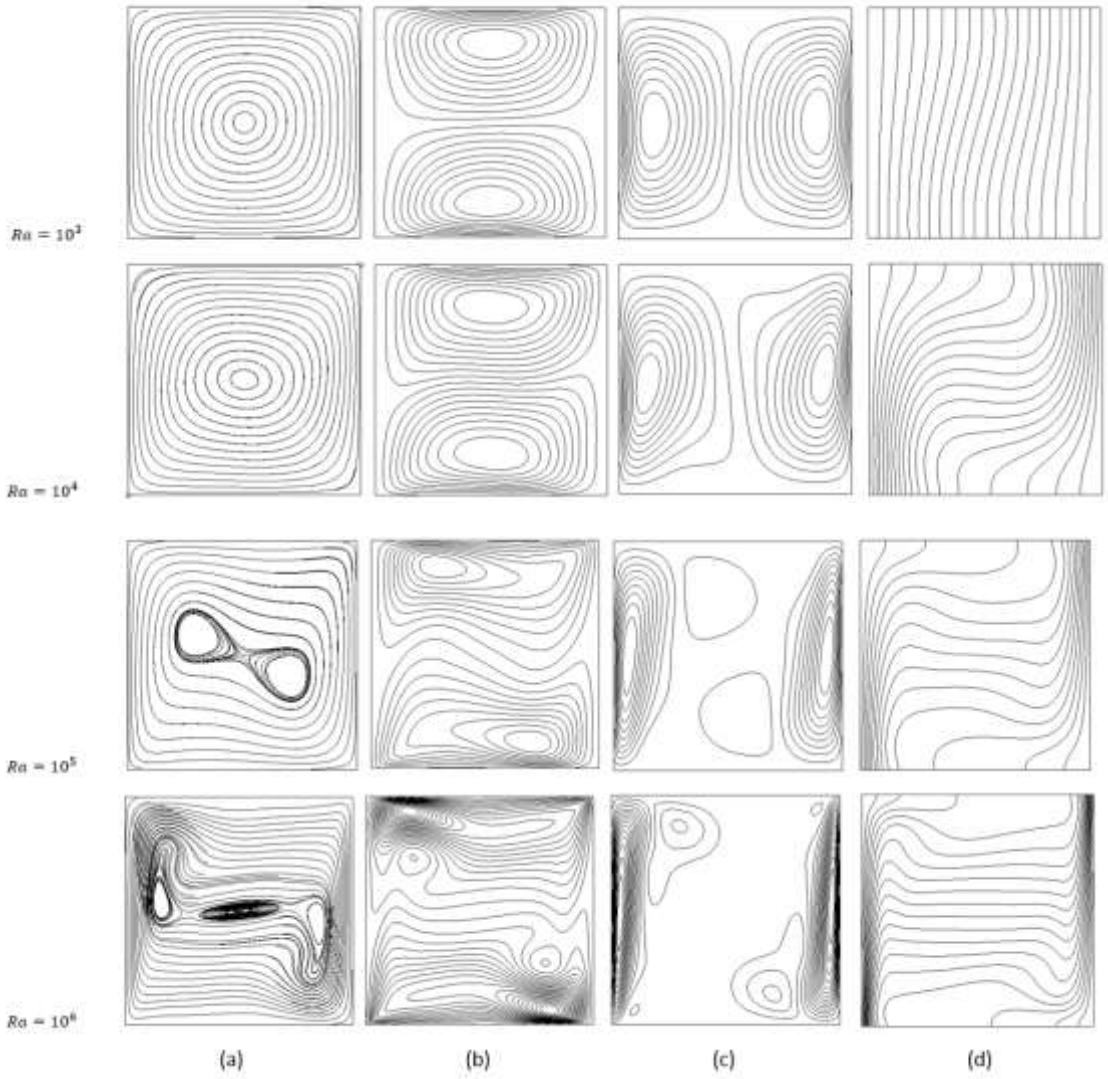
Debido a que la cavidad es adiabática en las paredes horizontales, el valor del Nusselt promedio a lo largo de cualquier línea vertical en cavidad es el mismo. En la figura 31 se puede apreciar los campos de temperaturas de cada cavidad a distinto valor de Rayleigh.

Figura 31. Campos de temperatura para distintos valores de Rayleigh, a) $Ra = 10^3$, b) $Ra = 10^4$ , c) $Ra = 10^5$ y d) $Ra = 10^6$ .




En la figura 32 se puede apreciar los patrones de convección natural para cada caso. Estos patrones sirven para analizar cómo se comporta el campo de flujo y temperaturas para cada valor de Rayleigh. Los campos que se graficaron fueron las líneas de corrientes, isovelocidades en U, isovelocidades en V y las líneas de isotérmicas.

Figura 32. Patrones de convección natural simulado por MVF, $10^3 < Ra < 10^6$, a) líneas de corriente, b) iso-velocidad U, c) iso-velocidad V, d) líneas isotérmicas.




**8.2.2 Validación** para validar la solución se calculó el campo de Nusselt (figura 35) y la variación del Nusselt local en la frontera caliente (figura 36) y se calculó el Nusselt promedio de cada variación. El valor de Nusselt promedio se compara con los valores proporcionados del estudio numérico de G. De Vahl Davis y el estudio empírico de Caton I. Tabla 18.

Antes de realizar los pasos anteriomente propuestos, se hizo un análisis de la independencia de la malla para un Ra=1e6, donde se comparó la variación del Nussel local y promedio para mallados de 30x30, 60x60 y 120x120 volúmenes de control. En la figura 33 se puede observar los campos de temperaturas para cada mallado, en la figura 34 la variación del Nusselt local, finalmente en la tabla 17 se comparó los errores de cada malla respecto a la solución de G. De Vahl Davis para el mismo número de Rayleigh.

Figura 33. Campos de temperatura. a) 30x30 vc b) 60x60 vc c) 120x120 vc.

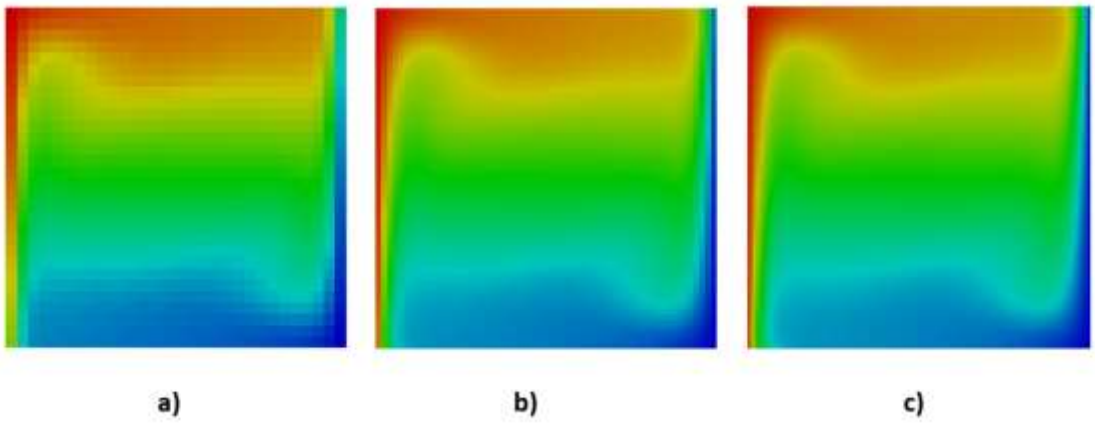



Figura 34. Variación de Nusselt local para distintas densidades de mallas, Ra=1e6.

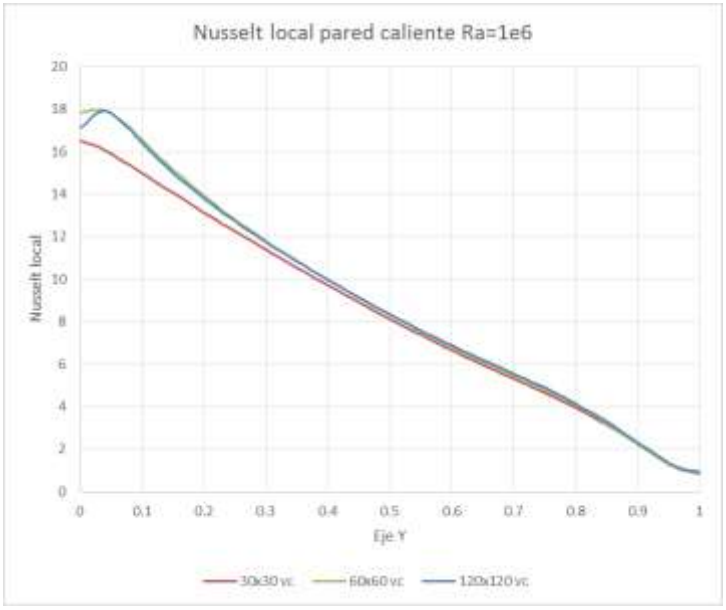


Tabla 17. Nusselt promedio para distintas densidades de malla, Ra=1e6.

| | Nusselt Promedio | | | | Error % | | |
|---|---|---|---|---|---|---|---|
| Rayleigh | 30x30 | 60x60 | 120x120 | G. De Vahl Davis (numérico) | 30x30 | 60x60 | 120x120 |
| $10^6$ | 8.25 | 8.68 | 8.69 | 8.80 | 6.22 | 1.37 | 1.28 |

Figura 35. Campo de Nusselt local, a) Ra=1e3, b) Ra=1e4, c) Ra=1e5 y d) Ra=1e6.

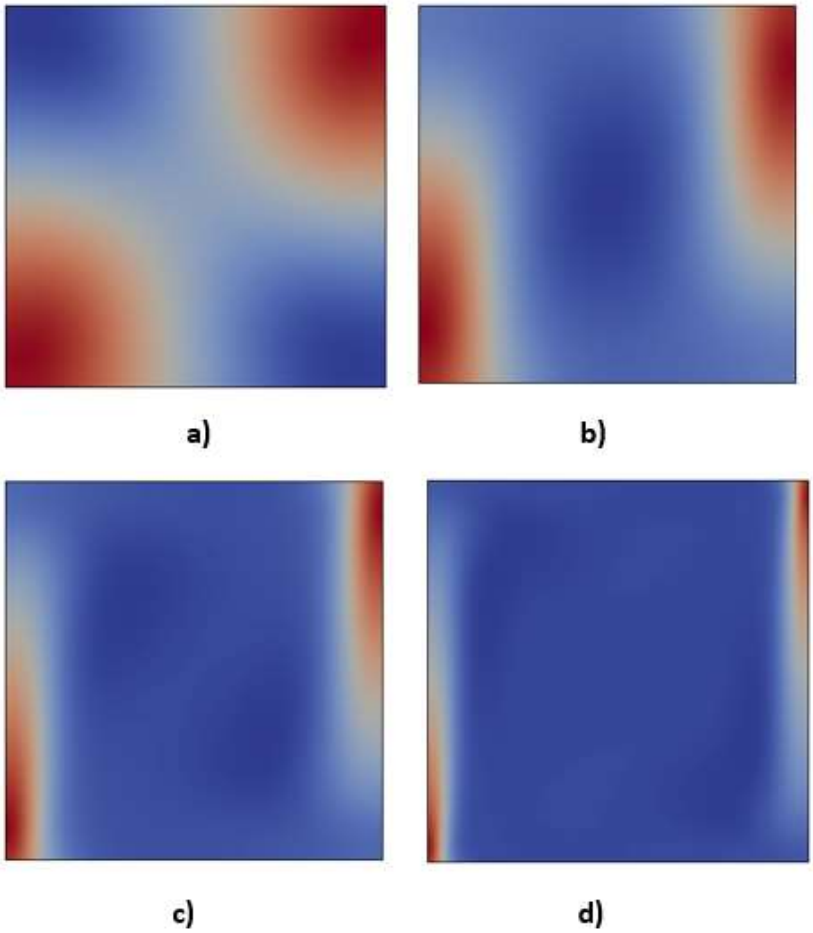


Figura 36. Nusselt local a lo largo de la frontera caliente para distintos valores de Rayleigh, MVF 120x120 volúmenes.

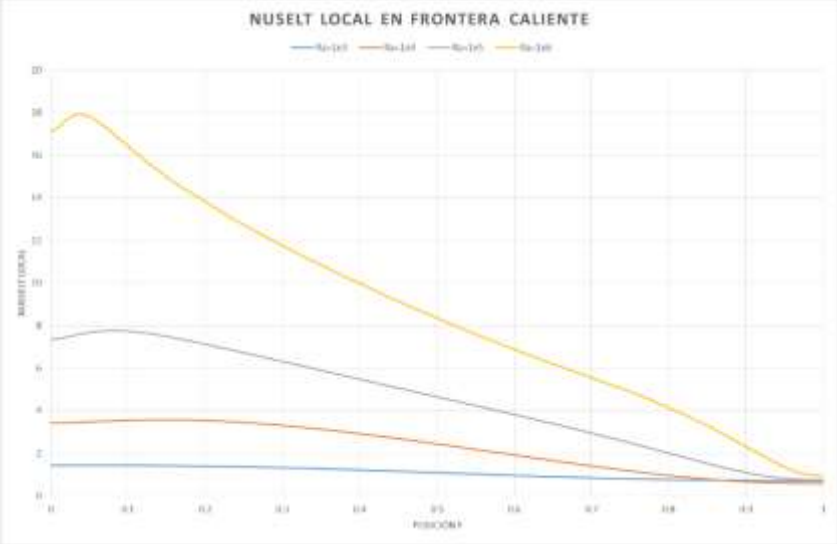



Tabla 18. Nusselt promedio.

| | Nusselt Promedio | | | Error | |
|---|---|---|---|---|---|
| Rayleigh | Presente estudio | G. De Vahl Davis (numérico) | Caton I. (experimental) | Errores relativo numerico (%) | Error relativo al experimiental (%) |
| $10^3$ | 1.07 | 1.12 | 1.24 | 4.28 | 13.68 |
| $10^4$ | 2.24 | 2.24 | 2.42 | 0.06 | 7.40 |
| $10^5$ | 4.46 | 4.52 | 4.72 | 1.41 | 5.60 |
| $10^6$ | 8.69 | 8.80 | 9.21 | 1.28 | 5.63 |

- **Distintos factores de forma.**

También se realizaron simulaciones para distintos factores de forma H/L para un valor de Raiyleigh constante (Ra=1e4), los factores de forma escogidos fueron H/L=2.5, 5 y 10. Para calcular el Nusselt promedio de cada cavidad, se extrajo el Nusselt local en la pared caliente (figura 38), el valor de Nusselt promedio se comparó con en valor de la correlación empírica (Ec.6.2) como se muestra en la figura 39 y se calcularon los errores para cada factor de forma (tabla 19).

Figura 37. Campo de temperaturas para distintos factores de forma H/L, Ra=1e4, Pr=0.71, a) H/L=2.5 b) H/L=5 c) H/L=10.

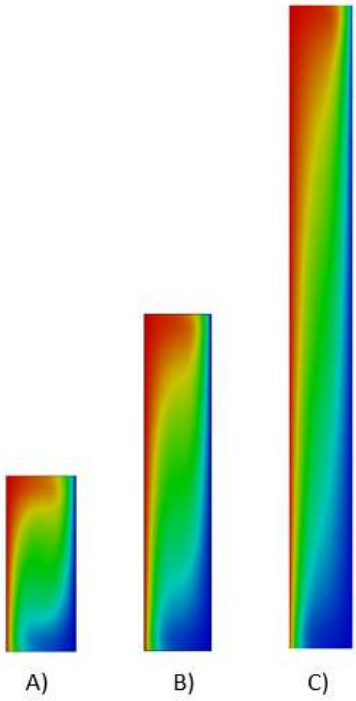



Figura 38. Nusselt local en la pared caliente, Ra=1e4, Pr=0.71.

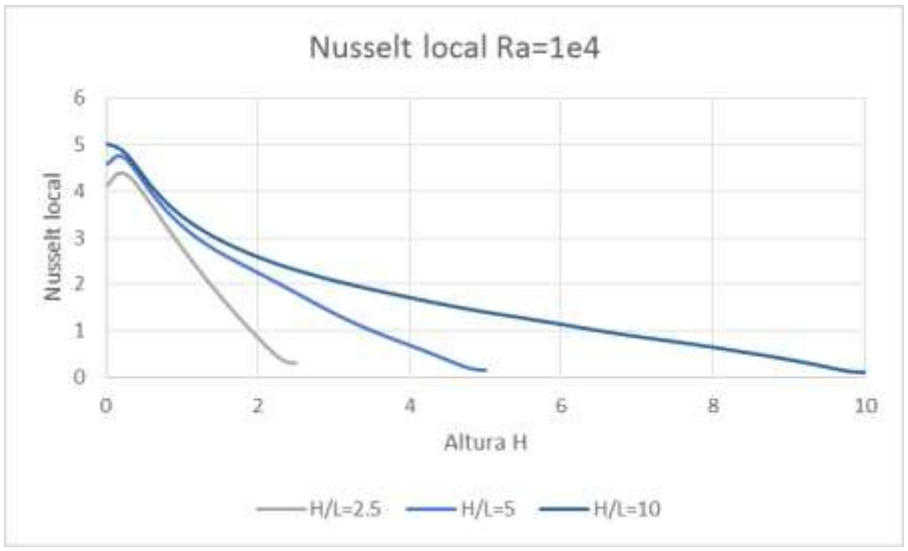


Figura 39. Nusselt promedio numérico vs empírico, Ra=1e4, Pr=0.71.

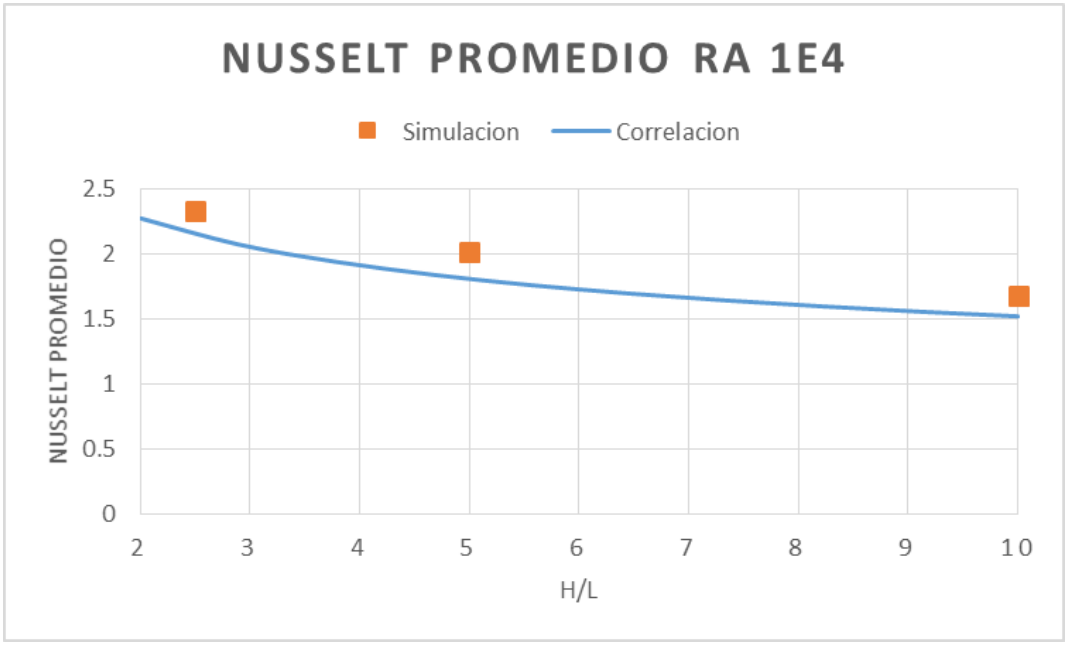



Tabla 19. Errores distintos factores de forma

| | Nusselt promedio | | |
|---|---|---|---|
| H/L | Simulacion | Experimental | Error % |
| 2.5 | 2.33 | 2.15 | 8.32 |
| 5 | 2.01 | 1.81 | 11.25 |
| 10 | 1.68 | 1.52 | 10.69 |

- **Distintos ángulos $\theta$.**

Finalmente se hizo una validación para distintos ángulos $\theta$, empleado la misma metodología se extrajo el Nusselt local en cada pared caliente (figura 41), se calculó el Nusselt promedio para cada simulación (figura 42) y finalmente se comparó cuantitativamente las solucion númerica vs experimental (tabla 20).

Figura 40. Campos de Nusselt para distintos ángulos, H/L=1, Ra=1e5, Pr=0.71, a) 30°, b) 40°, c) 45°, d) 60° y e) 90°.

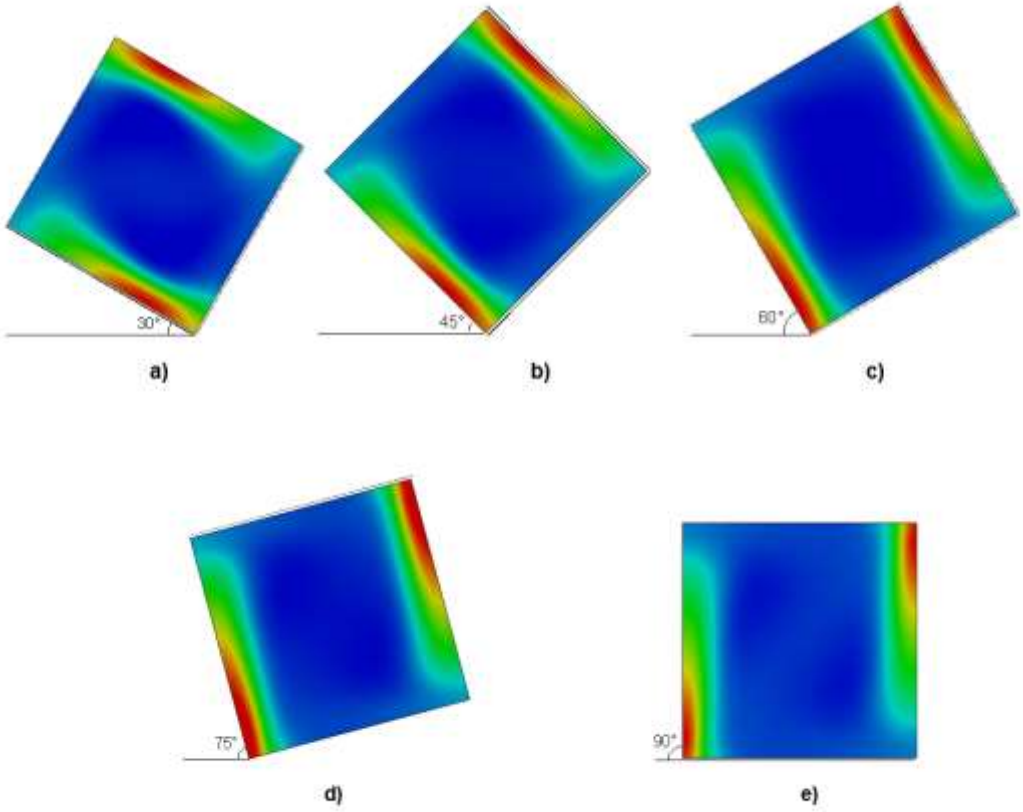



Figura 41. Nusselt local en la pared caliente para distintos ángulos, H/L=1, Ra=1e5, Pr=0.71.

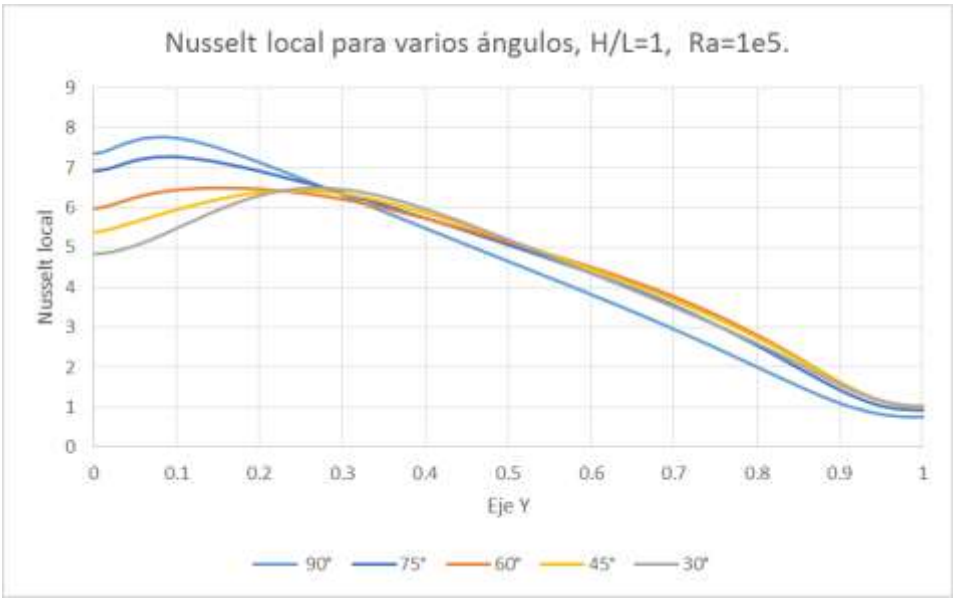


Figura 42. Comparación Nusselt promedio correlación vs simulación.

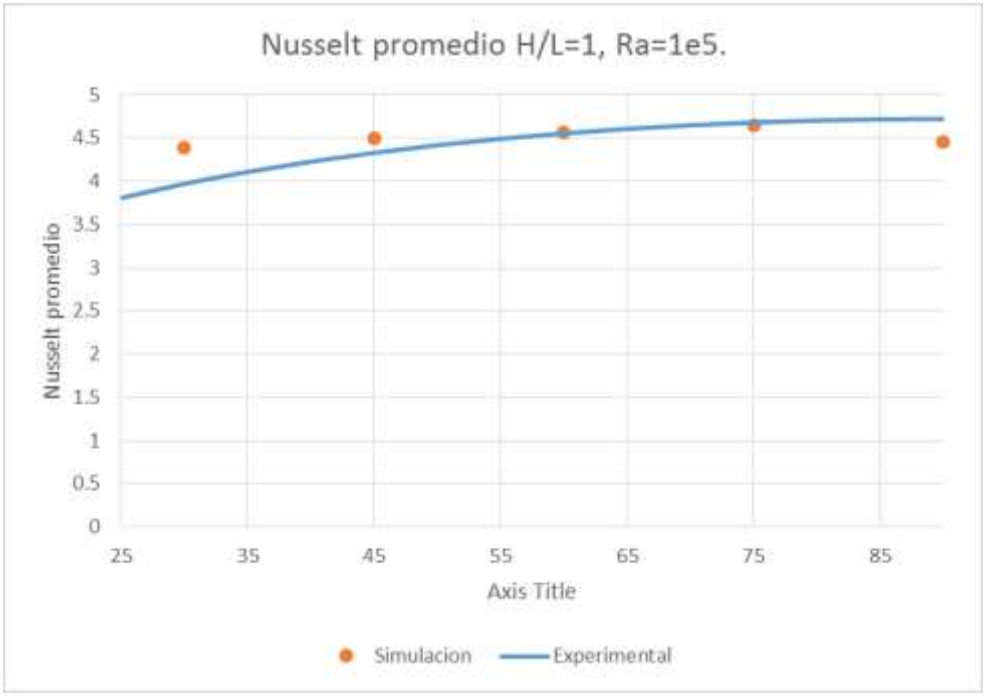



Tabla 20. Errores H/L=1

| | Nusselt promedio | | |
|---|---|---|---|
| **Ángulos** | **Simulacion** | **Experimental** | **Error %** |
| 30 | 4.39 | 3.97 | 10.47 |
| 45 | 4.49 | 4.33 | 3.74 |
| 60 | 4.56 | 4.55 | 0.17 |
| 75 | 4.65 | 4.68 | 0.67 |
| 90 | 4.46 | 4.72 | 5.60 |

### 8.2.3 Análisis de resultados

- En la figura 32 se muestran los patrones de convección natural, al comparar estos patrones con los de referencia (Anexo A), se puede observar la gran similitud que existen entre ambos. El análisis de patrones es una herramienta que permite validar gráficamente si los campos de flujo están bien calculados, ya que sin importar si el problema fue resuelto por diferencias finitas,

elementos finitos o volúmenes finitos todos tienden a la misma solución cuando el número de elementos tiende a infinito.

- En la figura 35 los campos de Nusselt local muestran cómo a medida que el número de Rayleigh aumenta, la influencia de la transferencia por conducción se va disminuyendo, esto se puede concluir observando como en las esquinas inferior izquierda y superior derecha donde se encuentran los valores más altos de Nusselt, para Ra=1e3 el Nusselt en las esquinas analizadas se transporta hacia todas las direcciones, a medida que se aumenta el número de Rayleigh se observa que el Nusselt se transporta cada vez en mayor parte siguiendo la dirección de flujo, hacia arriba en la pared caliente y hacia abajo en la pared fría.

- En la figura 36 y tabla 18 se hace un análisis cuantitativo de la solución numérica alcanzada. Comparando los valores de Nusselt promedio del presente estudio con el estudio numérico de referencia, se puede evidenciar que los resultados son muy similares debido a que se usan las mismas suposiciones, sin embargo al comparar ambas soluciones numéricas vs la correlación experimental (Ec.6.1.) se puede ver una mayor diferencia esto se debe a que las soluciones numéricas son solo una aproximación.

- En la figura 38 y 39 se realizaron simulaciones para distintos factores de forma H/L. Se pudo comprobar que cuando 2<H/L<10 el número de Nusselt es también dependiente a la relación H/L, observándose que a medida que la relación H/L se aumenta el valor de Nusselt promedio en la cavidad decrece para un numero de Rayleigh y Prandtl constante.

## 9. CONCLUSIONES.

- Se desarrolló un modelo matemático mediante la aplicación de ecuaciones de la conservación de materia, movimiento y energía para una cavidad rectangular bajo condiciones de convección natural. A partir de esta formulación matemática, se implementaron unas librerías que permiten resolver problemas de conducción, convección difusión y las ecuaciones de Navier-stokes para geometrías sencillas que puedan ser resueltas con un mallado estructurado. Las librerías fueron validadas con soluciones analíticas, numéricas y experimentales para poder determinar un correcto funcionamiento del programa. En la librería que soluciona el problema de convección-difusión, se programaron distintos esquemas de interpolación de término convectivo, todo esto con el fin de obtener una compresión más profunda acerca de la influencia de cada esquema de interpolación en la solución. Para la soluciones de las ecuaciones de Navier-Stokes y convección natural, el criterio para determinar los factores de relajación de Peric [10], $a_p = 1.1 - a_v$, ha ofrecido excelentes resultados, entre todas relaciones posibles de factores de relajación, $a_p = 0.4$ y $a_v = 0.7$ es el que ofreció una convergencia más rápida.

- La validación de los resultados arrojó una concordancia aceptable para el Nusselt promedio tanto respecto a la correlacion empírica (tabla 18) como los resultados numéricos reportador en la literatura. Para valores de Rayleigh de 1e3 el modelo propuesto obtuvo un error de 13.68% respecto al experimental, mientras que para valores de Rayleigh entre 1e4 y 1e6, el modelo propuesto obtuvo errores por debajo del 8%. Estos errores pueden estar relacionados con la aproximación Boussinesq en donde todas las propiedades son constantes excepto la densidad, algo que en la realidad no se cumple, debido a que a presión constante todas las propiedades son funciones de la temperatura. Desde el punto de vista númerico, la aproximación de Boussinesq es una solución muy poderosa por su facíl implementación.

- En el estudio de casos, el modelo propuesto reportó un comportamiento acorde con lo esperado tanto para distintos factores de forma como para disintintos ángulos, en donde para ambos casos se obtuvieron errores menores al 12%. Sin embargo, para ángulos en el rango de $45° < \theta < 90°$, el modelo propuesto obtuvo mejor comportamiendo.

- Se desarrolló un programa computacional que permite simular el flujo en una cavidad bajo condiciones de convección natural en régimen laminar. El software implementado puede tener muchos usos potenciales en distintos campos de la transferencia de calor como lo son: la simulación del proceso de termofusión en tanques o la simulación de aislantes gaseosos o líquidos en cavidades rectangulares, entre otros.

## 10. RECOMENDACIONES.

- El código desarrollando en esta tesis sirve para resolver problemas de convección natural en cavidad rectangular en régimen laminar para una geometría sencilla, se recomienda para posteriores estudios, implementar la solución para régimen turbulento y geometrías complejas.

- En la interpolación de los términos convectivos, se utilizaron esquemas de primer orden (Diferencias centradas, Upwind, Hibrido, Potencial y Exponencial), para posteriores estudios se podría investigar acerca de esquemas de orden superior, para una futura implementación.

- Se podría implementar otros métodos iterativos como lo son SIMPLEC, SIMPLER y PISO, y hacer un estudio para comparar la convergencia de cada método.

## BIBLIOGRAFÍA.

ARNOLD, J. et. al. Experimental investigation of natural convection in inclined rectangular regions of differing aspect ratios. En: Journal in Heat Transfer.1976, Vol. 98, pag. 67-71.

AYYASWAMY, P. S. et. al. The boundary-layer regime for natural convection in a differentially heated tilted rectangular cavity. En: Journal in Heat Transfer. 1973, Vol. 95, pag. 543.

BELEÑO, A. et. al. Seminario de investigación en mecánica de fluidos computacional. Trabajo de grado para optar el título de ingeniero mecánico. Bucaramanga. Universidad Industrial de Santander, Facultad de ingenierías Físico-Mecánicas, Escuela de Ingeniería Mecánica. 2009. 361 p.

CATTON, I. Natural convection in enclosures. En: Proceedings of the sixth international heat transfer conference. 1978, Vol. 6, pag. 13-31

CHUNG, T. J. Computational fluid dynamics. Cambridge: Cambridge University, 2002. p. 8-42.

DE VAHL DAVIS, D. Natural convection of air in a square cavity: a bench mark solution. En: International Journal for Numerical Methods in Fluids. 1983, Vol. 3, pag. 249 264.

FERNANDEZ ORO, Jesús Manuel. Técnicas numéricas en ingeniería de fluidos. Barcelona: Reverté, 2012.

FERZIGER, J.H. Y PERIC, M. Computational Methods for Fluid Dynamics, 3 ed.Berlin: Springer, 2002.

GHIA, U. et al .High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. En: Journal of Computational Physics. 1982, Vol. 48, pag. 387-411

JEREZ CARRIZALES, Manuel Fernando. Modelado y simulación del flujo de un fluido sobre una placa mediante volúmenes finitos. Trabajo de grado para optar el título de ingeniero mecánico. Bucaramanga. Universidad Industrial de Santander, Facultad de ingenierías Físico-Mecánicas, Escuela de Ingeniería Mecánica. 2012.

LIZARDI, A. et. al. Análisis numérico de la convección natural en recintos cerrados con distinto factor de forma. En: Nexo Revista Científica. 2011, Vol. 24, pag. 3-10.

MALALASEKERA. W. y VERSTEEG. H. An Introduction to Computational Fluid Dynamics: The Finite Volume Method, 2 edición. Harlow: Pearson Education Limited, 2007.503 p.

MARTINEZ RUEDA, Katherine Liseth. Modelado y simulación 2d del flujo de un fluido en un canal con una expansión inclinada mediante volúmenes finitos.

Trabajo de grado para optar el título de ingeniero mecánico. Bucaramanga. Universidad Industrial de Santander, Facultad de ingenierías Físico-Mecánicas, Escuela de Ingeniería Mecánica. 2014.

NPTEL. Computational Fluid Dynamics. [online], National Programme on Technology Enhanced Learning [citado 30 Noviembre 2014]. Disponible en internet: http://nptel.ac.in/courses/112105045/

OSTRACH, S. Natural convection in enclosures. En: Advances in Heat Transfer. Nueva York: Academic Press. 1972, Vol. 8, pag. 161-227.

PATANKAR, S. V. Numerical Heat Transfer and Fluid Flow. Taylor & Francis, 1980.

RAOS, M. Numerical investigation of laminar natural convection in inclined square enclosures. En: Physics, Chemistry and Technology. 2001, Vol. 2, pag. 149 – 157.

SHIRALKAR, G. S. and TIEN, C. A numerical study of laminar natural convection in shallows cavities. En: Journal of Heat Transfer. 1981, Vol. 103, pag. 226-231.

SMITH, R.M. y HUTTON, A.G. "The numerical treatment of advection: A performance comparison of current methods". En: Numerical Heat Transfer. 1982, Vol. 5, pag. 439-461.

**ANEXOS.**

## ANEXO A. PATRONES DE CONVECCION NATURAL.

Figura 1. Líneas de corrientes

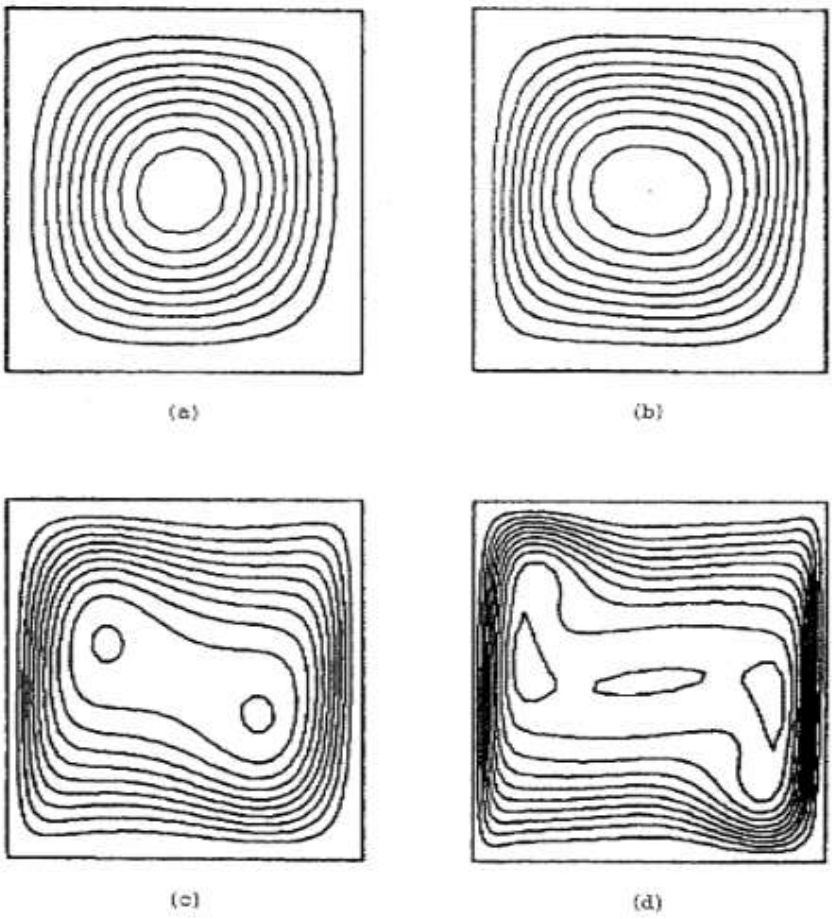



Figure 3. Contour maps of stream function $\psi$:
(a) $Ra = 10^3$; contours at $-1{\cdot}174(0{\cdot}1174)0$;
(b) $Ra = 10^4$; contours at $-5{\cdot}071(0{\cdot}5071)0$;
(c) $Ra = 10^5$; contours at $-9{\cdot}507$, $-8{\cdot}646(0{\cdot}9607)0$;
(d) $Ra = 10^6$; contours at $-16{\cdot}27$, $-15{\cdot}07(1{\cdot}675)0$

Fuente: DE VAHL DAVIS, D. Natural convection of air in a square cavity: a bench mark solution. En: International Journal for Numerical Methods in Fluids. 1983.

Figura 2. Contornos de isovelocidades U.

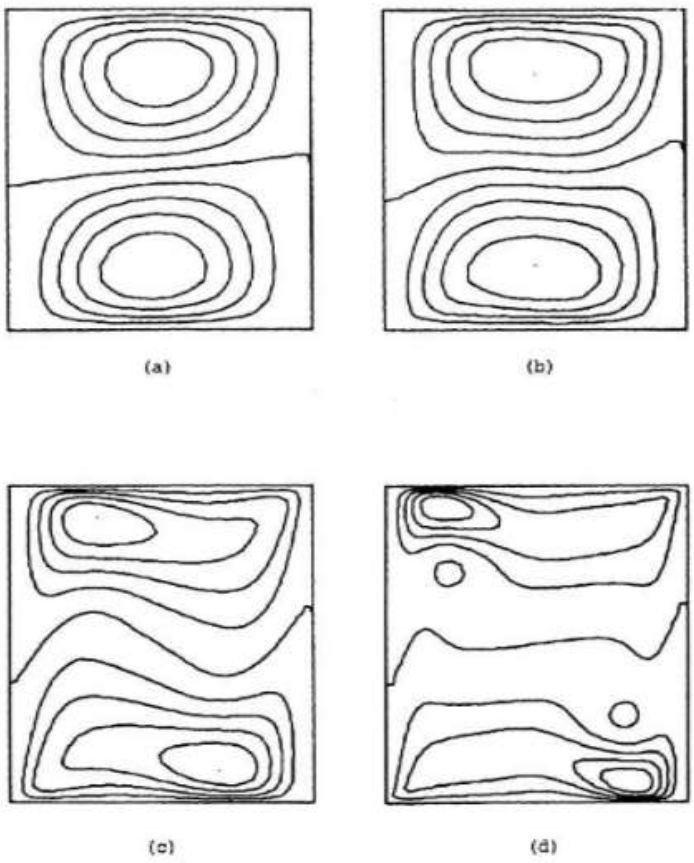



Figure 5. Contour maps of horizontal velocity $u$:
(a) $Ra = 10^3$; contours at −3·637(0·7274)3·637;
(b) $Ra = 10^4$; contours at −16·00(3·200)16·00;
(c) $Ra = 10^5$; contours at −43·59(8·719)43·59;
(d) $Ra = 10^6$; contours at −125·5(25·10)125·5

Fuente: DE VAHL DAVIS, D. Natural convection of air in a square cavity: a bench mark solution. En: International Journal for Numerical Methods in Fluids. 1983.

Figura 3. Contornos de isovelocidades V.

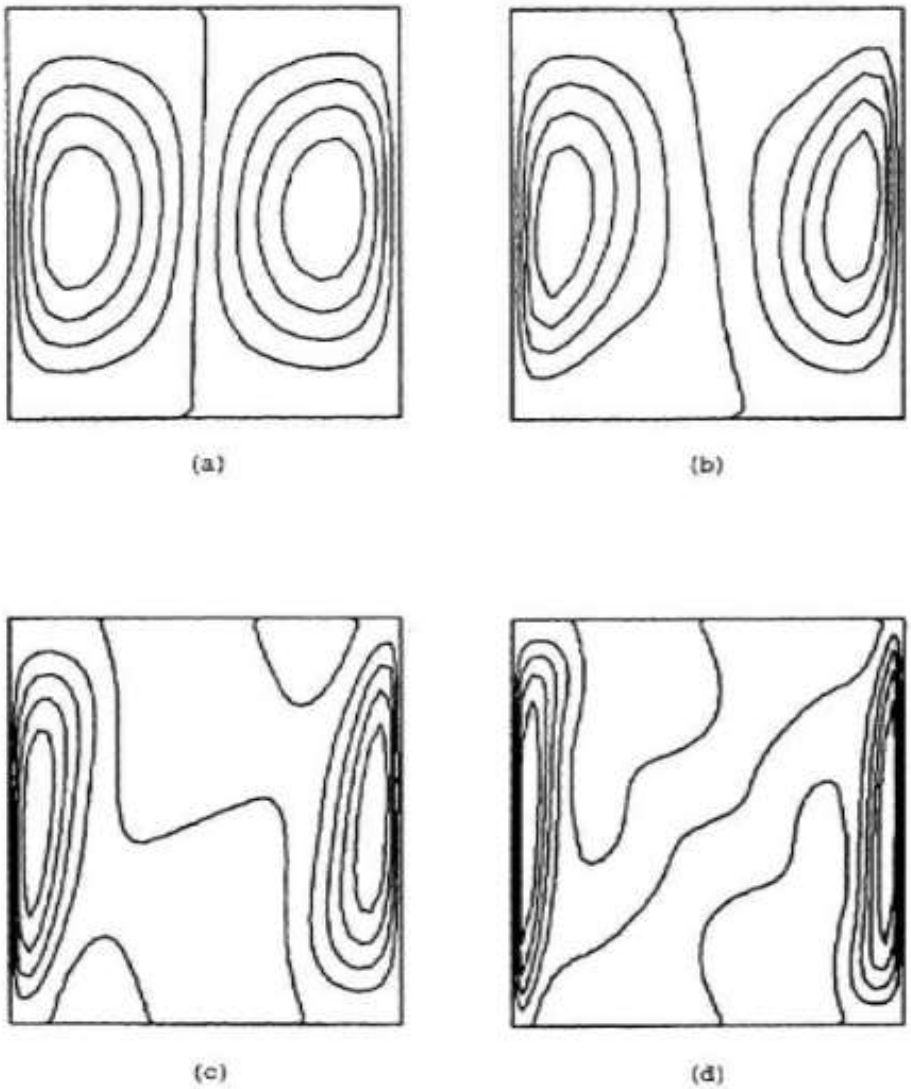



Figure 6. Contour maps of vertical velocity $w$:
(a) $Ra = 10^3$; contours at −3·663(0·7327)3·663;
(b) $Ra = 10^4$; contours at −19·39(3·877)19·39;
(c) $Ra = 10^5$; contours at −67·96(13·59)67·96;
(d) $Ra = 10^6$; contours at −207·6(41·52)207·6

Fuente: DE VAHL DAVIS, D. Natural convection of air in a square cavity: a bench mark solution. En: International Journal for Numerical Methods in Fluids. 1983.

Figura 4. Líneas isotérmicas.

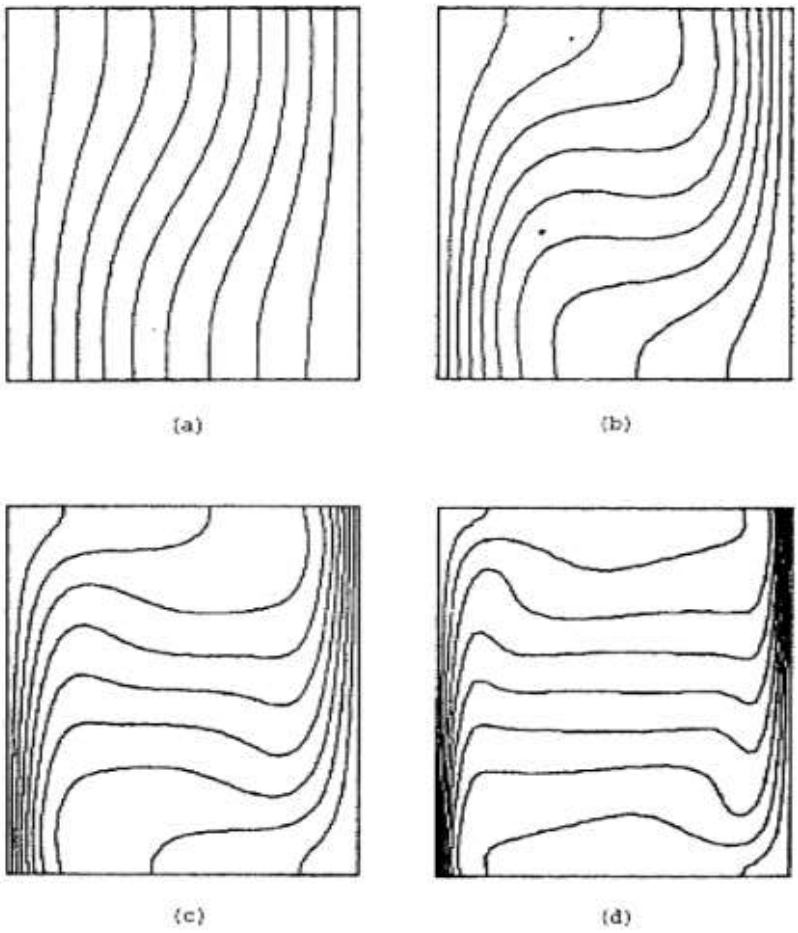



Figure 4. Contour maps of temperature $T$:
(a) $Ra = 10^3$, (b) $Ra = 10^4$,
(c) $Ra = 10^5$, (d) $Ra = 10^6$.
Contours at 0(0·1)1 in each case

Fuente: DE VAHL DAVIS, D. Natural convection of air in a square cavity: a bench mark solution. En: International Journal for Numerical Methods in Fluids. 1983.

## ANEXO B. CODIGO COMPUTACIONAL DESARROLLADO.

### 1. ARCHIVO DE CABECERA DE LA CLASE CDIFUSION.

```
1. #pragma once
2. #include <Puntero_Base.h>
3. #include <GuardarEnsight.h>
4. #include <Mallador.h>
5. #include <GradoLibertadFV.h>
6. #include <AdmonEcLin.h>
7. #include "BoCond.h"
8. //#include <ManejadorMPI.h>
9.
10. class NS;
11. class BDC;
12. /**@brief Clase que maneja los problemas de difusion, construye el
    sistema de ecuaciones para la discretización
13. de problemas de difusión basado en FVM */
14.
15. class CDifusion : public Puntero_Base, public GuardarEnsight
16. {
17. friend NS;
18. friend BDC;
19.
20. public:
21. CDifusion();
22. virtual ~CDifusion();
23. CDifusion(const char **nombre, int i=0);
24. CDifusion(const Cadena nombre, int i=0);
25. void Leer( Is is, int i=0);
26. /**Datos requeridos por el problema de difusion */
27. /** nombre del archivo que lleva las condiciones de frontera*/
28.
29. Cadena archbocond;
```

```
30. /** nombre del archivo que lleva los parametros del solver*/
31. Cadena archsolver;
32. /**Carpeta donde se descargan los archivos  */
33. Cadena carpetatrabajo;
34. /**Guarda el nombre del campo que se va a solucionar*/
35. Cadena nombrecampo;
36. /**Guarda el nombre del gradiente del campo que se va a solucionar*/
37. Cadena nombregrad;
38. /**Cadena que recibe los datos para el mallado*/
39. Cadena cadmallador;
40. /**Parametros del solucionador de ecuaciones*/
41. prm_Matriz<real> pm;
42. /**Generador de malla*/
43. Puntero<MallaFV> malla;
44. /**Administrador de ecuaciones*/
45. Puntero<AdmonEcLin> admlin;
46.
47.
48. /**Propiedades del material*/
49. double kdifusiva;
50. double densidad;
51. double cp;
52. double dt;
53. double vinicial;
54.
55. /**Esquemas de solucion*/
56. double theta;  // 1 explicito, 0.5 Crak Nicholsol y  0 implicito
57.
58.
59. /**Datos internos de la clase */
60. protected:
61. /**Numero de nodos de la topologia*/
```

```
62. int nnd;
63. /**Número de elementos de la topologia*/
64. int nne;
65. /**Número de dimensiones  en el espacio*/
66. int nsd;
67. /**Número de indicadores de frontera*/
68. int nbi;
69. /**Número de subdominios*/
70. int nsub;
71. /**Verifica si inicia NS*/
72. int iniNS;
73. /**Estrucutura de la matriz*/
74. Matriz_Dispersa<double> MatCoef;
75. /**Estructura del vectr de términos indpedendientes*/
76. Vector<double> VecCoef;
77. /**Estructura del vectr de términos solucion*/
78. Vector<double> VecSol;
79. /**Estructura del vectr de términos solucion anterior*/
80. Vector<double> SolAnt;
81. /**Campo a calcular*/
82. Puntero<CampoFVM> u;
83. /**Gradiente del campo a calcular(vectorial)*/
84. Puntero<CamposFVM> Grad;
85. /**Contiene las definiciones de las condiciones de frontera*/
86. Vector_basico<Puntero<BoCond> > bocos;
87. /**Numero de grados de libertad*/
88. Puntero<GradoLibertadFV> gdl;
89.
90. VecinoFVM vecino;
91.
92. Vector_basico<PunteroElmDefs> *elems;
93. /**Para simulacion temporal */
```

```
94. Puntero<prmTiempo> tiempo;
95. /**Puntero que guarda los flujos masicos en las caras*/
96. Puntero<Vector<double>> f_caras;
97. /**Puntero que guarda el ap en las caras*/
98. Puntero<Vector<double>> Ap;
99. /**Puntero que guarda la densidad  en los elementos*/
100.        Puntero<Vector<double>> Dens;
101.        /**Puntero que guarda la densidad  en los elementos*/
102.        Puntero<Vector<double>> DensAnt;
103.
104.  protected:
105.        int IniciaMemoria(int i=0);
106.        int ConstruyeSistema();
107.        int Solucion();
108.
109.        void CalculaFlujoCara(int cara, double *val);
110.
111.        void CalculaFlujoFrontera(int cara, double *val);
112.        void CalculaFlujoFrontera(int cara, double *val,int &tipo);
113.        void LeerBoCond();
114.        void ReportarResultados();
115.        double Generacion(int elem);
116.        void CalcularGradiente();
117.        void LlenarPatronDispersion();
118.        int EncPosicionDispersa(int e, int conex);
119.        void CalculaUnaIteracion();
120.        double CalcDensidad(int e);
121.        double CalcDensidadAnt(int e);
122.        double CalcDifusion(int e);
123.
124.
125.  public:
```

```
126.        void ResuelveProblema();
127.
128.  };
```

**2. ARCHIVO DE IMPLEMENTACION DE LA CLASE CDIFUSION**

```
1.  #include "CDifusion.h"
2.  #include <UtilidadesSisOp.h>
3.  #include <ConstOLeeMalla.h>
4.
5.  static const char *archivos_tb[] = { "archivos", "archbocond", "archsolver",
    "carpeta" };
6.  static const char *nombres_tb[] = { "nombrecampo", "nombregrad",
    "campo", "gradiente", "mallador" };
7.  static const char *propiedades_tb[] = { "densidad", "cp", "rho", "k",
    "mallador", "theta", "tinicial", "vinicial", };
8.  static const char *fronteras_tb[] = { "BoCo", "Cond_Frontera",
    "Boundary_Cond" };
9.  static const char *tiempos_tb[] = { "time_parameters", "tiempo",
    "param_tiempo", NULL };
10.
11. CDifusion::CDifusion()
12. {
13. }
14.
15. CDifusion::~CDifusion()
16. {
17. if (iniNS)
18.    delete elems;
19. }
20.
```

```
21. CDifusion::CDifusion(const char **nombre, int i)
22. {
23. Cadena arch;
24. arch = aform("ARCH=%s", nombre);
25. Is is(arch.carts());
26. Leer(is, i);
27. }
28.
29. CDifusion::CDifusion(const Cadena nombre, int i)
30. {
31. Cadena arch;
32. arch = aform("ARCH=%s", nombre.carts());
33. Is is(arch.carts());
34. Leer(is, i);
35. }
36.
37. void CDifusion::Leer(Is is, int i)
38. {
39. Cadena rutatrabajo = Path();
40. Cadena cadtiempo;
41. is->obtComando(archbocond, archivos_tb);
42. is->obtComando(archsolver, archivos_tb);
43. is->obtComando(carpetatrabajo, archivos_tb);
44. is->obtComando(nombrecampo, nombres_tb);
45. is->obtComando(nombregrad, nombres_tb);
46. is->obtComando(cadmallador, nombres_tb);
47. if (i == 0)
48. {
49.    malla.vincular(new MallaFV());
50.    ConstOLeeMalla(malla(), cadmallador);
51.
52. }
```

```
53.
54. pm.Leer(is);
55. admlin.vincular(new AdmonEcLin(EXTERNAL_STORAGE));
56. admlin().Leer(archsolver.carts());
57.
58. is->obtComando(cadtiempo, tiempos_tb);
59. if (i == 0){
60.     tiempo.vincular(new prmTiempo());
61.     tiempo->Leer(cadtiempo);
62.     nbi = malla->obtNoIndFront();
63. }
64. else
65.     nbi = i;
66. is->obtComando(densidad, propiedades_tb);
67. is->obtComando(kdifusiva, propiedades_tb);
68. is->obtComando(cp, propiedades_tb);
69. is->obtComando(theta, propiedades_tb);
70. is->obtComando(vinicial, propiedades_tb);
71.
72. LeerBoCond();
73. rutatrabajo = rutatrabajo + '\\' + carpetatrabajo;
74.
75. if (i == 0){
76.     MkDir(rutatrabajo);
77.     ChDir(rutatrabajo);
78.     GuardarEnsight::Leer(is, malla().obtNoDimEspacio());
79. }
80. }
81.
82. void CDifusion::LeerBoCond()
83. {
84. bocos.chaSize(nbi);
```

```
85. Is ifs(archbocond.carts());
86. Cadena cad;
87.
88. for (int i = 1; i <= nbi; i++){
89.     ifs->obtComando(cad, fronteras_tb);
90.     bocos(i).vincular(crearBoCond(cad, nsd));
91.     bocos(i)->Leer(ifs);
92. }
93. }
94.
95. int CDifusion::IniciaMemoria(int i)
96. {
97. nnd = malla->obtNoNodos(); //funciones
98. nne = malla->obtNoElem();
99. nsd = malla->obtNoDimEspacio();
100.         nbi = malla->obtNoIndFront();
101.         nsub = malla->obtNoSubDominios();
102.
103.         if (!nnd || !nne || !nsd || !nbi || !nsub) return 0;
104.
105.         u.vincular(new CampoFVM(malla(), nombrecampo.carts()));
106.         Grad.vincular(new CamposFVM(malla(), nombregrad.carts()));
107.         gdl.vincular(new GradoLibertadFV(malla(), 1));
108.
109.         MatCoef.chaSize(nne, nne);
110.         VecCoef.chaSize(nne);
111.         VecSol.chaSize(nne);
112.         SolAnt.chaSize(nne);
113.         SolAnt.llenar(vinicial);
114.
115.
116.         admlin().adjuntar(MatCoef, VecSol, VecCoef);
```

```
117.        iniNS = 1;
118.        if (i == 0) {
119.            iniNS = 0;
120.            vecino.iniciar(malla());
121.            malla->CalcConectividad(vecino);
122.
123.            elems = new Vector_basico<PunteroElmDefs>(nne);
124.            int e;
125.
126.            for (e = 1; e <= nne; e++)
127.            {
128.                elems->operator()(e).refill(malla->obtTipoElem(e), nsd);
129.                elems->operator()(e)->CalcGeomParameters(malla-
    >obtCoorElem(e));
130.
131.            }
132.
133.            malla->CrearVectorCaras(*elems, 1);
134.        }
135.        int n = malla->obtNoElem();
136.        pm.ncolumns = n;
137.        pm.nrows = nne;
138.        int N = nne;
139.        int M = malla->obtMaxNoNodosEnElem();
140.        pm.patron.vincular(new EstDispersa(nne, nne, (N + 1)*(M + 1)));
141.        LlenarPatronDispersion();
142.        MatCoef.chaSize(pm);
143.
144.
145.        return 1;
146.    }
147.
```

```
148.  void CDifusion::LlenarPatronDispersion()
149.  {
150.        int n, conex, numconex, j, e, fila = 0; //contador fila elemento
151.        int item = 0; //contador para los item en el patron
152.        // primer paso: se contruye la estructura de la matriz dispersa
153.        //iterar para cada elemento
154.        Vector_Ordenado<int> vord;
155.        for (e = 1; e <= nne; e++){
156.              fila++;
157.              pm.patron->irow(fila) = item + 1; //aqui empiela la fila del
    elemento
158.              // para cada elemnto conectado con el actual
159.
160.              numconex = elems->operator()(e)->GetNumOfConex();
161.              vord.chaSize(numconex + 1);
162.              vord.llenar(0);
163.              for (j = 1; j <= numconex; j++){
164.                    //obtener el elemnto conectado por el lado j
165.                    conex = malla->getConx(e, j);
166.                    if (conex) // no es lado de frontera
167.                          vord(j) = conex;
168.
169.              }
170.
171.              vord(j) = e;
172.              vord.OrdPila();
173.              n = vord.tam();
174.              for (j = 1; j <= n; j++){
175.                    if (vord(j) != 0){
176.                          item++;
177.                          pm.patron->jcol(item) = vord(j);
178.                    }
```

```
179.                }
180.        }
181.        pm.patron->irow(fila + 1) = item + 1;
182.
183. }
184.
185. void CDifusion::CalculaFlujoCara(int cara, double *val)
186. {
187.        int e = malla->caras(cara).elemento;
188.        int nb = malla->caras(cara).vecino;
189.        double area = malla->caras(cara).area;
190.        int ladovec = malla->caras(cara).ladovec;
191.        double dif_e = CalcDifusion(e);
192.        double dif_v = CalcDifusion(nb);
193.
194.        double delta_e = malla->caras(cara).dpc;// calcula el delta desde
    el centroide del elemento p
195.        double delta_v = malla->caras(cara).dec;// calcula el delta desde
    el centroide del elemento vecino
196.        double r1 = delta_e / (dif_e*area);
197.        double r2 = delta_v / (dif_v*area);
198.        val[0] = 1.0 / (r1 + r2);
199.        val[1] = 0.0;
200.        val[2] = 0.0;
201.
202. }
203.
204. void CDifusion::CalculaFlujoFrontera(int cara, double *val)
205. {
206.        int elem = malla->caras(cara).elemento;
207.        int lado = malla->caras(cara).lado;
208.        int bc = malla->caras(cara).bc;
```

```
209.        Vector_xyz<double> coord(nsd);
210.        if (bc){
211.              coord = malla->caras(cara).coc;
212.              double dif_e = CalcDifusion(elem);
213.              double delta_e = malla->caras(cara).dpc;
214.              double area = malla->caras(cara).area;
215.              double r1 = dif_e*area / delta_e;
216.              bocos(bc)->diff = r1;
217.              bocos(bc)->conv = 0.0;
218.              bocos(bc)->coord = coord;
219.              bocos(bc)->area = area;
220.              bocos(bc)->gamma = dif_e;
221.              bocos(bc)->Calcular(val[0], val[1]);// en val[0] carga el SU y
    en val[1] carga el SP
222.        }
223.  }
224.
225.  void CDifusion::CalculaFlujoFrontera(int cara, double *val, int &tipo)
226.  {
227.
228.        int bc = malla->caras(cara).bc;
229.        Vector_xyz<double> coord(nsd);
230.        if (bc){
231.              coord = malla->caras(cara).coc;
232.              bocos(bc)->coord = coord;
233.              bocos(bc)->Evaluar();
234.              tipo = bocos(bc)->ind;
235.              val[0] = bocos(bc)->valor;
236.              val[1] = bocos(bc)->valor2;
237.        }
238.  }
239.
```

```
240.  int CDifusion::EncPosicionDispersa(int e, int conex)
241.  {
242.        int inicio, fin, ll;
243.        inicio = pm.patron->irow(e);
244.        fin = pm.patron->irow(e + 1);
245.        ll = inicio - 1;
246.        do {
247.              ll++;
248.        } while (ll < fin && pm.patron->jcol(ll) != conex);
249.        if (ll >= fin) {
250.
251.              fatalerrorFP("CDifusion::EncPosicionDispersa", "Indice mal
    impuesto");
252.              // se debe general un mensaje de error falta
253.              return -1;
254.        }
255.        return ll;
256.  }
257.
258.
259.
260.  int CDifusion::ConstruyeSistema()
261.  {
262.        //MSG_PROC(std_o, "Difusion::Inicio Construye Sistema")
263.        MatCoef.llenar(0.0);
264.        VecCoef.llenar(0.0);
265.        VecSol.llenar(0.0);
266.
267.        int numconex, cara, conex;
268.        double area;
269.        double valores[3];
270.        double vol;
```

```
271.        double gen;
272.        double ap;
273.        double apo;
274.
275.        int e, j, item, fila;
276.
277.        for (e = 1; e <= nne; e++){
278.            numconex = elems->operator()(e)->GetNumOfConex();
279.            fila = EncPosicionDispersa(e, e);
280.            for (j = 1; j <= numconex; j++){
281.                cara = elems->operator()(e)->caras(j);
282.                area = elems->operator()(e)->getArea(j);
283.                conex = malla->getConx(e, j);
284.                if (conex){  //Nodo interno
285.                    CalculaFlujoCara(cara, valores);
286.                    item = EncPosicionDispersa(e, conex);
287.                    MatCoef(item) = valores[0] * (1.0 - theta);
288.                    MatCoef(fila) -= valores[0] * (1.0 - theta);
289.                    VecCoef(e) -= valores[0] * (theta)*(SolAnt(conex) -
    SolAnt(e));
290.                }
291.                else {
292.                    CalculaFlujoFrontera(cara, valores);
293.                    MatCoef(fila) -= valores[1]; //Sp
294.                    VecCoef(e) -= valores[0]; //Su
295.                }
296.            }
297.            vol = elems->operator()(e)->getVolume();
298.            gen = Generacion(e);
299.            ap = densidad*vol*cp / dt;
300.            MatCoef(fila) -= ap;
301.            apo = densidad*vol*cp / dt*SolAnt(e);
```

```
302.                VecCoef(e) -= gen*vol + apo;
303.        }
304.        //MSG_PROC(std_o, "Difusion::Fin Construye Sistema")
305.        if (nne < 25){
306.                MatCoef.Escribir(std_o, "Matriz Coeficientes", 3);
307.                VecCoef.Escribir(std_o, "Vector Coeficientes", 3);
308.        }
309.        return 1;
310. }
311.
312. double CDifusion::Generacion(int elem)
313. {
314.        if (!f_caras.ok())
315.                return 0.0;
316.        int cara;
317.        double res = 0.0;
318.        int numconex = elems->operator()(elem)->GetNumOfConex();
319.        for (int j = 1; j <= numconex; j++){
320.                cara = elems->operator()(elem)->caras(j);
321.                res += f_caras()(cara);
322.        }
323.        double vol = elems->operator()(elem)->getVolume();
324.        double masa = 0.0;// (CalcDensidad(elem) -
    CalcDensidadAnt(elem))*vol / dt;
325.        return (-res+masa) / vol;
326.
327. }
328.
329. int CDifusion::Solucion()
330. {
331.        //MSG_PROC(std_o, "Difusion::Inicio Solucion")
332.        Os logfile(NombreArchLog, APPEND);
```

```
333.        double start = omp_get_wtime();
334.        bool sol = admlin().solve();
335.        double end = omp_get_wtime();
336.        logfile << "\n Difusion :: solucion sistema ecuaciones( " <<
    nombrecampo.carts() << ")=" << end - start << "\n";
337.        if (sol) {
338.            //MSG_PROC(std_o, "\n<====================
    SOLUCION ==================>")
339.            int niteraciones;
340.            Boolean c;
341.            if (admlin->obtEstadisticas(niteraciones, c)){
342.                //MSG_PROC(std_o, "\n\n ***Solver %s convergio en
    %5d iteraciones *****\n\n", c ? "si" : "no", niteraciones);
343.                //MSG_PROC(logfile, "\n\n ***Solver %s convergio en
    %5d iteraciones *****\n\n", c ? "si" : "no", niteraciones);
344.            }
345.            if (nne < 25){
346.                VecSol.Escribir(std_o, nombrecampo.carts(), 3);
347.            }
348.            else{
349.                //MSG_PROC(logfile, "!!!! SIN SOLUCION !!!!!!!,
    %s",nombrecampo.carts);
350.            }
351.            //MSG_PROC(std_o, "Difusion::Fin Solucion")
352.        }
353.        else
354.            VecSol.llenar(0);
355.        return int(sol);
356.  }
357.
358.  void CDifusion::ReportarResultados()
359.  {
```

```
360.        habadventenciasglb = false;
361.        gdl->vector2campo(VecSol, u());
362.        volcar(u(), tiempo.obtPtr());
363.        volcar(Grad(), tiempo.obtPtr());
364.        /* volcar(u(), tiempo.obtPtr());
365.        volcar(Grad(), tiempo.obtPtr()); */
366.        SolAnt = VecSol;
367.
368.  }
369.
370.  void CDifusion::ResuelveProblema()
371.  {
372.        IniciaMemoria();
373.        tiempo->iniciarCicloTiempo();
374.        std_o << "\nInicio de simulacion\n";
375.        while (!tiempo->finalizo()){
376.              tiempo->incrementarTiempo();
377.              dt = tiempo->Delta();
378.              if (dt <= 0.0)  dt = 1.0;
379.              if (tiempo->estacionario()){
380.                    theta = 0.0;
381.                    dt = 1.0;
382.                    cp = 0.0;
383.              }
384.              else
385.                    std_o << "\n tiempo " << tiempo->obtTiempo() << "\n";
386.              if (!ConstruyeSistema())
387.                    return;
388.              std_o << "Calculando la solucion\n ";
389.
390.              if (Solucion())
391.                    CalcularGradiente();
```

```
392.            ReportarResultados();
393.
394.        }
395.  }
396.
397.  void CDifusion::CalcularGradiente()
398.  {
399.
400.        int e, j, numconex, conex;
401.        int cara, tipo;
402.        Vector_xyz<double> area(nsd), grad(nsd);
403.        double vol;
404.        double valor;
405.        double fm;
406.        double valores[2];
407.
408.        for (e = 1; e <= nne; e++){
409.            numconex = elems->operator()(e)->GetNumOfConex();
410.            vol = elems->operator()(e)->getVolume();
411.            grad.llenar(0.0);
412.
413.            for (j = 1; j <= numconex; j++){
414.                cara = elems->operator()(e)->caras(j);
415.                area = malla->caras(cara).Area;
416.                conex = malla->getConx(e, j);
417.
418.                if (conex){ // cara que tiene vecino
419.                    fm = malla->caras(cara).fm;
420.                    valor = fm*VecSol(e) + (1.0 - fm)*VecSol(conex);
421.                    grad = grad + area*valor / vol;
422.
423.                }
```

```
424.                    else {  // cara frontera
425.                        CalculaFlujoFrontera(cara, valores, tipo);
426.                        if (tipo == 1){
427.                            grad = grad + area*valores[0] / vol;
428.                        }
429.                        else if (tipo == 2){
430.                            grad = grad + area*(VecSol(e) - valores[0] *
    malla->caras(cara).dpc) / vol;
431.                        }
432.                    }
433.
434.                } //end for j
435.                for (j = 1; j <= nsd; j++)
436.                    Grad()(j).values()(e) = grad(j);
437.
438.        }
439.        if (nne < 25){
440.            std_o << "\n" << nombregrad.carts() << "\n";
441.            if (nsd >= 1)
442.                Grad()(1).values().Escribir(std_o, "grad x", 3);
443.            if (nsd >= 2)
444.                Grad()(2).values().Escribir(std_o, "grad y", 3);
445.            if (nsd >= 3)
446.                Grad()(3).values().Escribir(std_o, "grad z", 3);
447.        }
448. }
449.
450. void CDifusion::CalculaUnaIteracion()
451. {
452.        if (!ConstruyeSistema())
453.            return;
454.        if (Solucion())
```

```
455.            CalcularGradiente();
456. }
457.
458. double CDifusion::CalcDensidad(int e)
459. {
460.     if (!Dens.ok())
461.         return densidad;
462.     double res = Dens()(e);
463.     return res;
464. }
465.
466. double CDifusion::CalcDensidadAnt(int e)
467. {
468.     if (!DensAnt.ok())
469.         return densidad;
470.     double res = DensAnt()(e);
471.     return res;
472. }
473.
474.
475.
476. double CDifusion::CalcDifusion(int e)
477. {
478.     if (!Ap.ok())
479.         return kdifusiva;
480.     double res;
481.     double vol = elems->operator()(e)->getVolume();
482.     double dens = CalcDensidad(e);
483.     double ap = Ap()(e);
484.     res = dens*vol / ap;
485.
486.     return res;}
```

### 3. CÓDIGO SOLUCIÓN ANALÍTICA (MATLAB)

```
W=10;
L=10;
x=1;
y=9;
u=0;
n=1;
j=ones(30,1);
while n<30
U=((2*((-1)^(n+1)+1))*sin(n*pi*x/L)*sinh(n*pi*y/L))/(n*pi*sinh(n*pi*W/L));
    X=u+U;
    j(n)=X;
    n=n+1;
    u=X;
end

%Solucion Analitica
W=10;
```

```
L=10;
Ts=250;
Ti=30;
u=0;
T=30*ones(11,11);
y=11;
for x=1:11
    T(y,x)=250;
end

for y=1:9
    for x=1:9
        for n=1:25
            U=((2*((-1)^(n+1)+1))*sin(n*pi*x/L)*sinh(n*pi*y/L))/(n*pi*sinh(n*pi*W/L));
            P=U+u;
            u=P;
        end
        u=0;
        T(y+1,x+1)=P*(Ts-Ti)+Ti;
    end
end
contourf(T,'DisplayName','T');figure(gcf)
colorbar
```

**4. ARCHIVO DE CABECERA DE LA CLASE CD**

```
#pragma once

#include <Puntero_Base.h>
```

```
4.  #include <GuardarEnsight.h>
5.  #include <Mallador.h>
6.  #include <GradoLibertadFV.h>
7.  #include <AdmonEcLin.h>
8.  #include "BoCond.h"
9.  #include <math.h>
10.#include "TEvaluar.h"
11.#include <Tensor.h>
12.
13.//#include <ManejadorMPI.h>
14.
15.
16./**@brief Clase que maneja los problemas de difusion, construye el
    sistema de ecuaciones para la discretización
17.de problemas de difusión basado en FVM */
18.class NS;
19.class BDC;
20.
21.class CD : public Puntero_Base, public GuardarEnsight
22.{
23.friend NS;
24.friend BDC;
25.public:
26.CD();
27.virtual ~CD();
28.CD(int _dir, const char **nombre, int i = 0);
29.CD(int _dir, const Cadena nombre, int i = 0);
30.
31.void Leer(Is is, int i=0);
32.
33./**direcion de la coordenada que representa la velocidad*/
34.int dir;
```

```
35. /**Datos requeridos por el problema de difusion */
36. 
37. /** nombre del archivo que lleva las condiciones de frontera*/
38.     Cadena archbocond;
39. /** nombre del archivo que lleva los parametros del solver*/
40. Cadena archsolver;
41. /**Carpeta donde se descargan los archivos  */
42. Cadena carpetatrabajo;
43. /**Guarda el nombre del campo que se va a solucionar*/
44. Cadena nombrecampo;
45. /**Guarda el nombre del gradiente del campo que se va a solucionar*/
46. Cadena nombregrad;
47. /**Cadena que recibe los datos para el mallado*/
48. Cadena cadmallador;
49. /**Parametros del solucionador de ecuaciones*/
50. prm_Matriz<real> pm;
51. /**Generador de malla*/
52. Puntero<MallaFV> malla;
53. /**Administrador de ecuaciones*/
54. Puntero<AdmonEcLin> admlin;
55. 
56. 
57. /**Propiedades del material*/
58. double kdifusiva;
59. double densidad;
60. double cp;
61. double dt;
62. double vinicial;
63. 
64. /**Esquemas de solucion*/
65. double theta;  // 1 explicito, 0.5 Crak Nicholsol y  0 implicito
66. 
```

```
67.
68. /**Datos internos de la clase */
69. protected:
70. /**Numero de nodos de la topologia*/
71. int nnd;
72. /**Número de elementos de la topologia*/
73. int nne;
74. /**Número de dimensiones  en el espacio*/
75. int nsd;
76. /**Número de indicadores de frontera*/
77. int nbi;
78. /**Número de subdominios*/
79. int nsub;
80. /**Verifica si inicia por NS*/
81. int iniNS;
82. /**Estrucutura de la matriz*/
83. Matriz_Dispersa<double> MatCoef;
84. /**Estructura del vectr de términos indpedendientes*/
85. Vector<double> VecCoef;
86. /**Estructura del vectr de términos solucion*/
87. Vector<double> VecSol;
88. /**Estructura del vector de términos solucion anterior*/
89. Vector<double> SolAnt;
90. /**Estructura del vector de la solución anterior*/
91. Vector<double> vnm1;
92. /**Valor de relajacion*/
93. double alpha;
94. /**Puntero que guarda el ap en las caras*/
95. Puntero<Vector<double>> Ap;
96. /**Campo a calcular*/
97. Puntero<CampoFVM> u;
98. /**Gradiente del campo a calcular(vectorial)*/
```

```
99. Puntero<CamposFVM> Grad;
100.        /**Gradiente del campo a Presion(vectorial)*/
101.        Puntero<CamposFVM> Grad_p;
102.        /**Para almacenar los gradientes de las tres velociades en cada
    punto de la malla*/
103.        Puntero<Arreglo_Gen<real> > gradvels;
104.        /**Contiene las definiciones de las condiciones de frontera*/
105.        Vector_basico<Puntero<BoCond> > bocos;
106.        /**Numero de grados de libertad*/
107.        Puntero<GradoLibertadFV> gdl;
108.
109.        VecinoFVM vecino;
110.
111.        Vector_basico<PunteroElmDefs> *elems;
112.        /**Para simulacion temporal */
113.        Puntero<prmTiempo> tiempo;
114.
115.        /**Vector que guarda las velocidades en las caras*/
116.        Vector_basico<Vector_xyz<double> >  *v_caras;
117.        /**Vector que guardara los flujos masicos en las caras*/
118.        Puntero<Vector<double>> f_caras;
119.        /**Para guardar el esquema de interpolacion*/
120.        int esquema; // 1 UPWIND 2 CENTRADO 3 HIBRIDO
    4EXPONENCIAL 5 POTENCIAL
121.        /**Guarda las cadenas para la velocidad*/
122.        Vector_basico<Cadena>  vels;
123.
124.        /**Puntero que guarda la densidad  en los elementos*/
125.        Puntero<Vector<double>> Dens;
126.        /**Puntero que guarda la densidad  en los elementos*/
127.        Puntero<Vector<double>> DensAnt;
128.        /**Puntero que guarda la viscosidad  en los elementos*/
```

```
129.        Puntero<Vector<double>> Visc;
130.        /**Valor de campo gravitatorio para cada eje cordenado*/
131.        double campo;
132.        /**Activa el término fuente de flotacion*/
133.        int flot;
134.        /**Angulo de la cavidad respecto a la horizontal*/
135.        double  angulo;
136.
137.
138.  protected:
139.        int IniciaMemoria(int i=0);
140.        int ConstruyeSistema();
141.        int Solucion();
142.
143.        void CalculaFlujoCara(int cara, double *val);
144.        double CalculaFlujoDifusivo(int cara);
145.        double CalculaFlujoConvectivo(int cara);
146.        double CalculaFronteraDifusivo(int cara);
147.        double CalculaFronteraConvectivo(int cara);
148.        /// Se utiliza para llenar el sistema de ecuaciones aporta SU y SP
149.        void CalculaFlujoFrontera(int cara, double *val);
150.        // Se utiliza para calcular el valor de la variable en la frontera(
    calculo del gradiente)
151.        void CalculaFlujoFrontera(int cara, double *val,int &tipo);
152.        void LeerBoCond();
153.        void ReportarResultados();
154.        double Generacion(int elem);
155.        void CalcularGradiente();
156.        void LlenarPatronDispersion();
157.        int EncPosicionDispersa(int e, int conex);
158.        double CoefAnb(double Pe);
159.        void LlenarVel();
```

```
160.        double CalcDensidad(int e);
161.        double CalcDensidadAnt(int e);
162.        double CalcDifusion(int e);
163.        void CalculaUnaIteracion();
164.        Tensor < real > CalcTensor(int cara);
165.        Vector_xyz < real > CalcTFuente(int cara);
166.
167.
168.
169.  public:
170.        void ResuelveProblema();
171.        virtual void AdjuntarGradPresion (const CamposFVM &A);
172.        virtual void AdjuntarGradVelocidad(const Arreglo_Gen<real> &A);
173.  };
174.
```

**5. ARCHIVO DE IMPLEMENTACION DE LA CLASE CD**

```
1.  #include "CD.h"
2.  #include <UtilidadesSisOp.h>
3.  #include <ConstOLeeMalla.h>
4.  #include <math.h>
5.  #include "TEvaluar.h"
6.
7.  using namespace std;
8.
9.  static const char *archivos_tb[] = { "archivos", "archbocond",
    "archivosolver", "carpeta","archsolver" };
10. static const char *nombres_tb[] = { "nombrecampo", "nombregrad",
    "campo", "gradiente", "nombregradiente", "mallador" };
11. static const char *propiedades_tb[] = { "densidad", "cp", "rho", "k", "theta",
    "kdifusiva", "tinicial", "vinicial",
```

```
12.                                                    "campo", "gravedad", "esquema",
    "velocidad","flotacion","angulo" };
13. static const char *frontera_tb[] = {"BoCo", "Cond_Frontera",
    "Boundary_Cond" };
14. static const char *tiempos_tb[] = {"time_parameters", "param_tiempo",
    "tiempo", NULL };
15.
16.
17. static double Evaluar(const Cadena &cvalor, double x, double y, double
    z)
18. {
19. double valor;
20. TEvaluar evaluar;
21. char cadena[255];
22. strcpy(cadena, cvalor.carts());
23. evaluar.Evaluar(cadena, x, y, z, valor);
24. return valor;
25. }
26.
27. CD::CD()
28. {
29. alpha = 1.0;
30. }
31.
32.
33. CD::~CD()
34. {
35. if(iniNS=0){
36.     delete v_caras;
37.     delete elems;
38. }
39. }
```

```
40.
41.
42. CD::CD(int _dir, const char **nombre, int i)
43. {
44. Cadena arch;
45. arch = aform("ARCH=%s", nombre);
46. Is is(arch.carts());
47. dir = _dir;
48. Leer(is, i);
49. alpha = 1.0;
50. }
51.
52.
53. CD::CD(int _dir, const Cadena nombre, int i)
54. {
55. Cadena arch;
56. arch = aform("ARCH=%s", nombre.carts());
57. Is is(arch.carts());
58. dir = _dir;
59. Leer(is, i);
60. alpha = 1.0;
61.
62.
63. }
64.
65.
66.
67. void CD::Leer(Is is, int i)
68. {
69. Cadena original = Path();
70. Cadena ruttatrabajo;
71. Cadena cadtiempo;
```

```
72. is->obtComando(archbocond, archivos_tb);
73. is->obtComando(archsolver, archivos_tb);
74. is->obtComando(carpetatrabajo, archivos_tb);
75. is->obtComando(nombrecampo, nombres_tb);
76. is->obtComando(nombregrad, nombres_tb);
77. is->obtComando(cadmallador, nombres_tb);
78. if(i==0){
79.     malla.vincular(new MallaFV());
80.     ConstOLeeMalla(malla(), cadmallador);        //Crea el CASE
81.     nbi = malla->obtNoIndFront();
82. }
83. else
84.     nbi=i;
85.
86.
87. pm.Leer(is);
88. admlin.vincular(new AdmonEcLin(EXTERNAL_STORAGE));
89. admlin().Leer(archsolver.carts());
90.
91.
92. is->obtComando(cadtiempo, tiempos_tb);
93. if(i==0){
94.     tiempo.vincular(new prmTiempo());
95.     tiempo->Leer(cadtiempo);
96. }
97.
98. is->obtComando(densidad, propiedades_tb);
99. is->obtComando(kdifusiva, propiedades_tb);
100.        is->obtComando(cp, propiedades_tb);
101.        is->obtComando(theta, propiedades_tb);
102.        is->obtComando(vinicial, propiedades_tb);
103.
```

```
104.        LeerBoCond();
105.        is->obtComando(esquema, propiedades_tb);
106.        is->obtComando(campo, propiedades_tb);
107.
108.
109.        ruttatrabajo = original + '\\' + carpetatrabajo;
110.
111.        if(i==0){
112.              nsd = malla().obtNoDimEspacio();
113.              vels.chaSize(nsd);
114.              for(int ii=1; ii <= nsd; ii++)
115.                    is->obtComando(vels(ii), propiedades_tb);
116.              MkDir(ruttatrabajo);
117.              ChDir(ruttatrabajo);
118.              GuardarEnsight::Leer(is, nsd);
119.        }
120.
121.        //HASTA AQUÍ SE HIZO LA LECTURA DE DATOS DEL
    PROBLEMA
122.        //*Esto se hace con el fin de organizar los archivos de
    resultados*//
123.  }
124.
125.  void CD::LlenarVel()
126.  {
127.        int cara;
128.        Vector_xyz<real> coc(nsd);
129.        Vector_xyz<real> area(nsd);
130.        int caras = v_caras->tam();
131.        //int elem;
132.        real x, y, z ;
133.        for (cara = 1; cara <= caras; cara++){
```

```
134.            v_caras->operator()(cara).chaSize(nsd);
135.            v_caras->operator()(cara).llenar(0.0);
136.            area = malla->caras(cara).Area;
137.            coc = malla->caras(cara).coc;
138.            x = 0.0;
139.            y = 0.0;
140.            z = 0.0;
141.            if (nsd >= 1)   x = coc(1);
142.            if (nsd >= 2)   y = coc(2);
143.            if (nsd >= 3)   z = coc(3);
144.
145.            if (nsd >= 1)   v_caras->operator()(cara)(1) = Evaluar(vels(1),
    x, y, z);
146.            if (nsd >= 2)   v_caras->operator()(cara)(2) = Evaluar(vels(2),
    x, y, z);
147.            if (nsd >= 3)   v_caras->operator()(cara)(3) = Evaluar(vels(3),
    x, y, z);
148.            f_caras->operator()(cara) = v_caras-
    >operator()(cara)*area*densidad;
149.        }
150.  }
151.
152.  void CD::LeerBoCond()
153.  {
154.        bocos.chaSize(nbi);
155.        Is ifs(archbocond.carts());              //Is = input stream
156.        Cadena cad;
157.        for (int i=1; i <= nbi; i++){
158.            ifs->obtComando(cad,frontera_tb);
159.            bocos(i).vincular(crearBoCond(cad,nsd));
160.            bocos(i)->Leer(ifs);
161.        }
```

```
162.
163.  }
164.
165.  int CD::IniciaMemoria(int i)
166.  {
167.          nnd = malla->obtNoNodos();
168.          nne = malla->obtNoElem();
169.          nsd = malla->obtNoDimEspacio();
170.          nbi = malla->obtNoIndFront();
171.          nsub = malla->obtNoSubDominios();
172.
173.          //Ahora se va a iniciar memoria de esta estructura
174.
175.          if(!nne || !nnd || !nsd || !nbi || !nsub) return 0;  //Se hace esto
     porque no estaría definido el problema, no se podría continuar
176.
177.          u.vincular(new CampoFVM(malla(),nombrecampo.carts()));
        //New hace el vinculo dinámico para poder crear la variable u se debe
     definir que vaya con la malla y con el nombre
178.          Grad.vincular(new CamposFVM(malla(),nombregrad.carts()));
        //Si u es estática se debería ar cada uno de sus datos pero eso es
     posible en el .h
179.          gdl.vincular(new GradoLibertadFV(malla(), 1));
        //aquí aparecio un error<<<<<------
180.          Ap.vincular(new Vector<double>);
181.
182.          //Lo anterior se hace para poder volcarlo
183.
184.          MatCoef.chaSize(nne, nne);                    //Se dimensiona la
     matriz y vectores
185.          VecCoef.chaSize(nne);
186.          VecSol.chaSize(nne);
```

```
187.        SolAnt.chaSize(nne);
188.        vnm1.chaSize(nne);
189.        vnm1.llenar(0.0);
190.        SolAnt.llenar(vinicial);
191.        VecSol.llenar(vinicial);
192.        Ap().chaSize(nne);
193.
194.        admlin().adjuntar(MatCoef,VecSol,VecCoef);
                    //admlin administrador de ecuaciones lineales, al sistema
    que resuelve el problema hace la operación de solución
195.        iniNS = 1;
196.        if(i=0){
197.            iniNS=0;
198.            vecino.iniciar(malla());
199.            malla->CalcConectividad(vecino);
200.
201.            elems = new Vector_basico<PunteroElmDefs>(nne);
202.            int e;
203.            for(e=1; e<=nne; e++){
204.                elems->operator()(e).refill(malla->obtTipoElem(e),nsd);
        //Se asigna un espacio de memoria para el elemento e con cierta
    geometría y dimensiones
205.                elems->operator()(e)->CalcGeomParameters(malla-
    >obtCoorElem(e));        //con base en las coordenadas la función
    calgeomparameters calcula todas las vaiables geometricas
206.            }
207.
208.            malla->CrearVectorCaras(*elems, 1);                //A la cara n
    dice cupal elemento tiene como vecino y las distancias entre el lado de
    su cara y el centroide del elemento
209.            v_caras=new Vector_basico<Vector_xyz<double> >(malla-
    >caras.tam());
```

```
210.            f_caras=new Vector<double>(malla->caras.tam());
211.            LlenarVel();
212.        }
213.        int n = malla->obtNoElem();
214.        //para todas las matrices
215.        pm.ncolumns = n;
216.        pm.nrows = nne;
217.        int N = nne;
218.        int M = malla->obtMaxNoNodosEnElem();
219.        pm.patron.vincular(new EstDispersa(nne, nne, (N + 1)*(M + 1)));
220.        LlenarPatronDispersion();
221.        MatCoef.chaSize(pm);
222.        return 1;
223.  }
224.
225.  void CD::LlenarPatronDispersion()
226.  {
227.        int n, conex,numconex, j, e, fila = 0; //contador para la fila
    (elemento)
228.        int item = 0; //contador para los item en el patrón
229.        //primer paso: se construye la estructura de la matriz dispersa
230.        //iterar cada elemento
231.        Vector_Ordenado<int> vord;
232.        for(e = 1; e <= nne; e++){
233.            fila++;
234.            pm.patron->irow(fila) = item + 1; //aquí empiza la fila del
    elemento
235.            //para cada elemento conectado con el actual
236.
237.            numconex = elems->operator()(e)->GetNumOfConex();
238.            vord.chaSize(numconex + 1);
239.            vord.llenar(0);
```

```
240.            for(j = 1; j <= numconex; j++){
241.                //obtener el elemento conectado por el lado j
242.                conex = malla->getConx(e, j);
243.                if(conex) //no es lado de frontera
244.                    vord(j) = conex;
245.            }
246.            vord(j) = e;
247.            vord.OrdPila();
248.            n = vord.tam();
249.            for(j = 1; j <= n; j++){
250.                if(vord(j) !=0){
251.                    item++;
252.                    pm.patron->jcol(item) = vord(j);
253.                }
254.            }
255.        }
256.
257.        pm.patron->irow(fila + 1) = item + 1;
258. }
259.
260. double CD::CalculaFlujoDifusivo(int cara)
261. {
262.        double res;
263.        double area=malla->caras(cara).area;          //De esa cara deme
    el area
264.        int ladovec = malla->caras(cara).ladovec; //Devuelve la cara del
    vecino
265.        double dif_e=kdifusiva;
266.        double dif_v=kdifusiva;
267.        double delta_e = malla->caras(cara).dpc;
268.        double delta_v = malla->caras(cara).dec;          //distancia
    normal del centroide del vecino al centro del
```

```
269.        double r1 = delta_e/(dif_e*area);
270.        double r2 = delta_v/(dif_v*area);
271.        res = 1.0/(r1 + r2);                    //U = 1/sum(resistencias
    térmicas)
272.        return res;
273.  }
274.
275.  double CD::CalculaFlujoConvectivo(int cara)
276.  {
277.        double res;
278.        res = f_caras->operator()(cara); //tiene () en lugar de <>
279.        return res;
280.  }
281.
282.  double CD::CalculaFronteraDifusivo(int cara)
283.  {
284.
285.        double area = malla->caras(cara).area;
286.        double dif_e = kdifusiva;
287.        double delta_e = malla->caras(cara).dpc;// calcula el delta desde
    el centroide del elemento p
288.        double r1 = (dif_e*area) / delta_e;
289.        return r1;
290.  }
291.
292.  double CD::CalculaFronteraConvectivo(int cara)
293.  {
294.        double res;
295.        res = f_caras->operator()(cara);
296.        return res;
297.
298.  }
```

```
299.
300.  double CD::CoefAnb(double Pe)
301.  {
302.        double res;
303.        switch(esquema){
304.        case 2: res = 1.0; break;          //Esquema Upwind
305.        case 1: res = 1.0 - 0.5*fabs(Pe); break;                    //Esquema
    centrado
306.        case 3: res = MMax(0.0, 1.0 - 0.5*fabs(Pe)); break;
      //Hibrido
307.        case 5: res = fabs(Pe)/(exp(fabs(Pe)) - 1.0); break;
      //Exponencial
308.        case 4: res = MMax(0.0, pow(1.0 - 0.1*fabs(Pe), 5.0)); break;
    //Potencial
309.        }
310.        return res;
311.  }
312.
313.  void CD::CalculaFlujoCara(int cara, double *val)
314.  {
315.        double dif = CalculaFlujoDifusivo(cara);
316.        double conv = CalculaFlujoConvectivo(cara);
317.        double Pe = conv/dif;
318.        //**double res = CoefAnb(Pe);  //Le pasa el Pecler y él devuelve el
    valor de res correspondiente*/
319.        double res = dif*CoefAnb(fabs(Pe))+MMax(0.0,-conv);
320.        val[0] = res;
321.        val[1] = conv;
322.        val[2] = dif;
323.  }
324.
325.  void CD::CalculaFlujoFrontera(int cara, double *val)
```

```
326.  {
327.        int elem = malla->caras(cara).elemento;          //Este el lado del
    elemento que esta en contacto con la cara
328.        int lado = malla->caras(cara).lado;
329.        int bc = malla->caras(cara).bc;                  //El ya sabe
    con cuál condición de frontera esta ligado
330.        double phi_p = SolAnt(elem);
331.        Vector_xyz<double> coord(nsd);
332.        if(bc){
333.              coord = malla->caras(cara).coc;            //coc cuál
    es el centroide de la cara
334.              double dif_e = kdifusiva;
335.              double delta_e = malla->caras(cara).dpc;
336.              double area = malla->caras(cara).area;
337.              /**double r1 = dif_e*area/delta_e;*/
338.              real Rcond = CalculaFronteraDifusivo(cara);
339.              real Rconv = CalculaFronteraConvectivo(cara);
340.              bocos(bc)->calculo = _COTHER;
341.              bocos(bc)->esquema = esquema;
342.              bocos(bc)->diff = Rcond;        //Parte difusiva valor r1
343.              bocos(bc)->conv = Rconv;        //Parte convectiva
344.              bocos(bc)->coord = coord;
345.              bocos(bc)->area = area;
346.              bocos(bc)->gamma = dif_e;
347.              bocos(bc)->phi_p = phi_p;
348.              bocos(bc)->Calcular(val[0], val[1]); //val[0]:S_u, val[1]:S_p
349.
350.        }
351.  }
352.
353.  void CD::CalculaFlujoFrontera(int cara, double *val, int &tipo)
354.  {
```

```
355.        int bc = malla->caras(cara).bc;                    //El ya sabe
    con cuál condición de frontera esta ligado
356.        Vector_xyz<double> coord(nsd);
357.        if(bc){
358.            coord = malla->caras(cara).coc;                //coc cuál
    es el centroide de la cara
359.            bocos(bc)->coord = coord;
360.            bocos(bc)->Evaluar();
361.            tipo = bocos(bc)->ind;
362.            val[0] = bocos(bc)->valor;        //Aquí llega temperatura o
    flujo
363.            val[1] = bocos(bc)->valor2;       //Aquí llega el coeficiente de
    tc por convección, si es la cf mixta
364.        }
365.  }
366.
367.  int CD::EncPosicionDispersa(int e, int conex)
368.  {
369.        int inicio, fin, ll;   //Dada la posición i->e conex->j a qué indice
    corresponde ne el vector
370.        inicio = pm.patron->irow(e);
371.        fin = pm.patron->irow(e + 1);
372.        ll = inicio - 1;
373.        do{
374.            ll++;
375.        } while (ll<fin && pm.patron->jcol(ll) != conex);
376.        if(ll >= fin){
377.            fatalerrorFP("Difusion::EncPosicionDispersa", "Indice mal
    impuesto");
378.            //se debe generar un mensaje de error fatal
379.            return -1;
380.        }
```

```
381.        return ll;
382.  }
383.
384.  int CD::ConstruyeSistema()
385.  {
386.        //Mensaje con salida estandar std_o
387.        //MSG_PROC(std_o, "Difusion: Inicio Construye Sistema")
388.        MatCoef.llenar(0.0);
389.        VecCoef.llenar(0.0);
390.        VecSol.llenar(0.0);
391.
392.        int numconex,cara, conex;
393.        double valores[3];
394.        double area;
395.        double gen,vol,ap,apo;
396.        int e,j, fila, item;          //Esta contador hace el recorrido por el
    numero de elementos
397.        for(e=1; e <= nne; e++){
398.              numconex = elems->operator()(e)->GetNumOfConex();
399.              fila = EncPosicionDispersa(e, e);
400.              for(j=1; j<=numconex; j++){
401.                    cara = elems->operator()(e)->caras(j);
402.                    area = elems->operator()(e)->getArea(j);
403.                    conex=malla->getConx(e, j);                  //Con quién esta
    conectado el elemento e en el lado j
404.                    if(conex){            //Nodo interno
405.                          CalculaFlujoCara(cara, valores);
406.                          item = EncPosicionDispersa(e, conex);
407.                          MatCoef(item) = valores[0]*(1.0-theta);
      //Representa la resistencia térmica del elemento con su vecion
```

```
408.                        MatCoef(fila) -= (valores[0]+valores[1])*(1.0-theta);
                            //El coeficiente del elemento (e,e), -= quiere decir
    que se van sumando, el centro es debido a la dirección del flujo de calor
409.                        VecCoef(e) -= (valores[0] +
    valores[1])*theta*(SolAnt(conex)-SolAnt(e));
410.                }
411.                else{               //Condición de frontera
412.                    CalculaFlujoFrontera(cara, valores);
413.                    MatCoef(fila) -= valores[1]; //val[1] = S_p
414.                    VecCoef(e) -= valores[0];   //val[0] = S_u
415.                }
416.            }
417.            vol=elems->operator()(e)->getVolume();
418.            gen=Generacion(e);
419.            ap=densidad*vol*cp/dt;
420.            MatCoef(fila) -= ap;
421.            Ap()(e) = -MatCoef(fila);  //En lugar de traerce de la matriz se
    trae de un vector
422.            MatCoef(fila) = MatCoef(fila) / alpha; //Cuando el problema es
    normal alpha vale 1
423.            apo = densidad*vol*cp*SolAnt(e) / dt;
424.            VecCoef(e) -= gen*vol + apo+((1.0-
    alpha)*Ap()(e)/alpha)*vnm1(e);
425.        }
426.        //MSG_PROC(std_o, "Difusion: Fin Construye Sistema")
427.        if(nne < 25){
428.            MatCoef.Escribir(std_o, "Matriz de Coeficientes", 3);
429.            VecCoef.Escribir(std_o, "Vector de Coeficientes", 3);
430.        }
431.        return 1;
432.  }
433.
```

```
434.  double CD::Generacion(int elem)
435.  {
436.        if(!Grad_p.ok() || !gradvels.ok())
437.              return 0.0;
438.        double vol=elems->operator()(elem)->getVolume();
439.        double vg = -(densidad-CalcDensidad(elem))*campo*vol;   /**Valor
      del component fuente por la densidad y gravedad*/
440.
441.        int j, numconex, cara;
442.        double fuente = 0.0;
443.
444.        numconex = elems->operator()(elem)->GetNumOfConex();
445.
446.        Vector_xyz<real> fcara(nsd);
447.        for(j = 1; j <= numconex; j++){
448.              cara = elems->operator()(elem)->caras(j);
449.              /*fuerzas de campo y de esfuerzo solamente*/
450.              fcara = CalcTFuente(cara);
451.              fuente = fuente + fcara(dir); /**Newtons*/
452.        }
453.
454.        double divpre = Grad_p()(dir).values()(elem)*vol;
455.        double terfuente = (vg + fuente*0 - divpre)/vol;
456.        return terfuente;
457.  }
458.
459.  Tensor<real> CD::CalcTensor(int cara)
460.  {
461.        Tensor<real> tau(nsd);
462.        double visco;
463.        //1. interpolo todos los gradientes necesarios al centro del
      elemento
```

```
464.        Tensor<real> gr(nsd);
465.        int conex = malla->caras(cara).vecino;
466.        int e = malla->caras(cara).elemento;
467.        double fm = malla->caras(cara).fm;
468.        int i, j;
469.        if(conex){
470.            visco = CalcDifusion(e)*(1.0 - fm) + CalcDifusion(conex)*fm;
471.            for(i=1; i<=nsd; i++){
472.                for(j=1; j <=nsd; j++){
473.                    gr(i,j) = gradvels()(i,j,e)*(1.0 - fm) +
    gradvels()(i,j,conex)*(fm);
474.                }
475.            }
476.        }
477.        else {
478.            visco = CalcDifusion(e);
479.            for(i=1; i<=nsd; i++){
480.                for(j=1; j<=nsd; j++){
481.                    gr(i,j) = gradvels()(i, j, e);
482.                }
483.            }
484.        }
485.        double beta = 0.0;
486.        //2. Calculo en tensor de esfuerzos
487.        if (nsd<=1) tau(1,1) = visco*(1.0 - beta)*gr(1,1);
488.        else if (nsd <= 2){
489.            tau(1, 1) = visco*(1.0*gr(1, 1) - 2.0*gr(2, 2))/3.0;
490.            tau(2, 2) = visco*(1.0*gr(2, 2) - 2.0*gr(1, 1))/3.0;
491.            tau(2, 1) = tau(1, 2) = visco*(gr(1, 2) + gr(2, 1));
492.        }
493.        else if (nsd <= 3){
494.            tau(1, 1) = visco*(1.0*gr(1, 1) - 2.0*gr(2, 2) - 2.0*gr(3, 3))/3.0;
```

```
495.            tau(2, 2) = visco*(1.0*gr(2, 2) - 2.0*gr(1, 1) - 2.0*gr(3, 3))/3.0;
496.            tau(3, 3) = visco*(1.0*gr(3, 3) - 2.0*gr(1, 1) - 2.0*gr(2, 2))/3.0;
497.            tau(2, 1) = tau(1, 2) = visco*(gr(1, 2) + gr(2, 1));
498.            tau(3, 1) = tau(1, 3) = visco*(gr(1, 3) + gr(3, 1));
499.            tau(3, 2) = tau(2, 3) = visco*(gr(3, 2) + gr(2, 3));
500.        }
501.        return tau;
502.  }
503.
504.  Vector_xyz<real> CD::CalcTFuente(int cara)
505.  {
506.        Tensor<real> tt;
507.        //Ftau=fuerza que ejerce el esfuerzo
508.        Vector_xyz<real> Area(nsd), Ftau(nsd);
509.
510.        Area = malla->caras(cara).Area;
511.        tt = CalcTensor(cara);
512.        Ftau = tt*Area;
513.        return Ftau;
514.  }
515.
516.  int CD::Solucion()
517.  {
518.        //MSG_PROC(std_o, "Difusion: Inicio Solución")
519.        Os logfile(NombreArchLog, APPEND);        //Vuelve lenta la
    ejecución
520.        double start = omp_get_wtime();
521.        bool sol = admlin().solve();
522.        double end = omp_get_wtime();
523.        logfile << "\nDifusion :: solucion sistema ecuaciones (" <<
    nombrecampo.carts() <<")=" << end - start << "\n";
524.        if(sol){
```

```
525.            //MSG_PROC(std_o, "\n<==================
    SOLUCION ==================>")
526.            int niteraciones;
527.            Boolean c;
528.            if (admlin->obtEstadisticas(niteraciones, c)){
529.            //MSG_PROC(std_o, "\n\n *** Solver %s convergio en %5d
    iteraciones ***\n\n", c ? "si" : "no", niteraciones)
530.            //MSG_PROC(logfile, "\n\n *** Solver %s convergio en %5d
    iteraciones ***\n\n", c ? "si" : "no", niteraciones)
531.            }
532.            if(nne < 25){
533.            VecSol.Escribir(std_o, nombrecampo.carts(), 3);
534.            }
535.        }
536.
537.        else {
538.            //MSG_PROC(std_o, " ¡¡¡ SIN SOLUCION !!!, %s",
    nombrecampo.carts())
539.
540.        }
541.
542.        //MSG_PROC(std_o, "Difusion: Fin Solución")
543.
544.        return (int)sol;
545.  }
546.
547.  void CD::ReportarResultados()
548.  {
549.        habadventenciasglb = false;
550.        gdl->vector2campo(VecSol, u());
551.        volcar(u(), tiempo.obtPtr());
552.        volcar(Grad(), tiempo.obtPtr());
```

```
553.        /*volcar(u(), 0);
554.        volcar(u(), tiempo.obtPtr());
555.        volcar(Grad(), tiempo.obtPtr());*/
556.        SolAnt=VecSol;
557.
558.  }
559.
560.  void CD::ResuelveProblema()
561.  {
562.        IniciaMemoria();
563.        tiempo->iniciarCicloTiempo();
564.        std_o <<"\nInicio simulacion";
565.        while (!tiempo->finalizo()) { //&&solucion_ex){
566.              tiempo->incrementarTiempo();
567.              dt=tiempo->Delta();
568.              if(dt<=0.0) dt=1.0;
569.              if(tiempo->estacionario()) {
570.                    theta = 0.0;          //Así la solución explícita
571.                    dt = 1.0;
572.                    cp = 0.0;
573.              }
574.              else
575.                    std_o <<"\n tiempo" << tiempo->obtTiempo() << "\n";
576.              if(!ConstruyeSistema())
577.                    return;
578.
579.              std_o << "Calculando la solucion\n";
580.              if(Solucion()){
581.                    CalcularGradiente();
582.                    ReportarResultados();
583.              }
584.        }
```

```
585.  }
586.
587.  void CD::CalcularGradiente()
588.  {
589.        double valores[2];
590.        int e, j, numconex, conex;
591.        int cara, tipo;
592.        Vector_xyz<double> area(nsd), grad(nsd);
593.        double valor, vol;
594.        double fm;
595.
596.        for(e=1; e<=nne; e++){
597.              numconex = elems->operator()(e)->GetNumOfConex();
598.              vol = elems->operator()(e)->getVolume();
599.              grad.llenar(0.0);
600.              for(j=1; j <= numconex; j++){
601.                    cara = elems->operator()(e)->caras(j);
602.                    area = malla->caras(cara).Area; //Con mayúscula da el
    vector
603.                    conex = malla->getConx(e, j);
604.                    if (conex){  //cara que tiene vecino
605.                          fm = malla->caras(cara).fm;
606.                          valor = fm*VecSol(e) + (1.0 - fm)*VecSol(conex);
607.                          grad = grad + area*valor/vol;              //el orden
    de multiplicación es debido a que area es un vector
608.                    }
609.                    else{  //Carra en la frontera
610.                          CalculaFlujoFrontera(cara, valores, tipo);
611.                          if(tipo == 1){//Condición de frontera Dirichlet
612.                                grad = grad + area*valores[0]/vol;
613.                          }
614.                          else if(tipo == 2){
```

```
615.                                grad = grad + area*(VecSol(e) -
    valores[0]*malla->caras(cara).dpc)/vol;          //Por medio de la serie de
    Taylor con el valor de Fi(p) y la tendencia de fi en la frontera se calcula
    el fi(v) del vecino
616.                            }
617.                        }
618.                } //end for j
619.                for(j=1; j<=nsd; j++){
620.                    Grad()(j).values()(e) = grad(j);  //Gradiente del elmento j
    en le elemento e es igual al grad del elemento j que anteriormente se
    calculo
621.                }
622.            }
623.            if (nne<25){
624.                std_o << "\n" << nombregrad.carts() << "\n";
625.                if (nsd>=1)
626.                Grad()(1).values().Escribir(std_o, "grad x",3);
627.                if (nsd >= 2)
628.                Grad()(2).values().Escribir(std_o, "grad y", 3);
629.                if (nsd >= 3)
630.                Grad()(3).values().Escribir(std_o, "grad z", 3);
631.
632.            }
633.  }
634.
635.  void CD::CalculaUnaIteracion()
636.  {
637.            if(!ConstruyeSistema())
638.                        return;
639.            if(Solucion())
640.                CalcularGradiente();
641.  }
```

```
642.
643. double CD::CalcDensidad(int e)
644. {
645.         if(!Dens.ok())
646.                 return densidad;
647.         double res = Dens()(e);
648.         return res;
649. }
650.
651. double CD::CalcDensidadAnt(int e)
652. {
653.         if (!DensAnt.ok())
654.                 return densidad;
655.         double res = DensAnt()(e);
656.         return res;
657. }
658.
659.
660. double CD::CalcDifusion(int e)
661. {
662.         if(!Visc.ok())
663.                 return kdifusiva;
664.         double res = Visc()(e);
665.         return res;
666. }
667.
668. void CD::AdjuntarGradPresion(const CamposFVM &A)
669. {
670.         Grad_p.vincular(A);
671. }
672.
673. void CD::AdjuntarGradVelocidad(const Arreglo_Gen<double> &A)
```

```
674.  {
675.        gradvels.vincular(A);
676.  }
677.
```

## 6. ARCHIVO DE CABECERA DE LA CLASE NS

```
1.  #pragma once
2.  #include <IsOs.h>
3.  #include <Puntero_Base.h>
4.  #include <GuardarEnsight.h>
5.  #include <Mallador.h>
6.  #include <GradoLibertadFV.h>
7.  #include <AdmonEcLin.h>
8.  #include "BoCond.h"
9.  #include <math.h>
10. #include "TEvaluar.h"
11. #include <UtilidadesSisOp.h>
12. #include <ConstOLeeMalla.h>
13. #include "CD.h"
14. #include "CDifusion.h"
15.
16. class BDC;
17.
18. class NS : public Puntero_Base, public GuardarEnsight
19. {
20. friend BDC;
21. public:
22. NS();
23. virtual ~NS();
24. NS(const char *nombre);
25. NS(const Cadena nombre);
```

```
26.
27. /**Datos de la clase*/
28. public:
29. double dt;
30. double theta;
31. int iter;
32. double errormax;
33. double error;
34. double pinicial;
35.
36. /**Carpeta donde se descargan los archivos  */
37. Cadena carpetatrabajo;
38. /**Cadena que recibe los datos para el mallado*/
39. Cadena cadmallador;
40. /** nombre del archivo que lleva las condiciones de frontera*/
41. Cadena archbocond;
42.
43. protected:
44. Puntero<VecinoFVM> vecino;
45. /**Para simulacion temporal */
46. Puntero<prmTiempo> tiempo;
47. /**Generador de malla*/
48. Puntero<MallaFV> malla;
49. /**Número de dimensiones  en el espacio*/
50. int nsd;
51. /**Número de indicadores de frontera*/
52. int nbi;
53. /**Número de volumenes*/
54. int nne;
55. /**Factor de relajación*/
56. double alphap;
57. /**Factor de relajación para las velocidades*/
```

```
58. Vector_xyz<double> alphav;
59. /**Maneja los diferentes  problemas de conveccion difusion para en
    calculo de velocidades*/
60. Vector_basico < Puntero<CD> > vels ;
61. /**Maneja el problema de la correccion de presion*/
62. Puntero<CDifusion> corrp;
63. /** Vector */
64. Vector_basico<PunteroElmDefs> *elems;
65. /**Vector que guarda las velocidades en las caras*/
66. Vector_basico<Vector_xyz<double> >  *v_caras;
67. /**Vector que guardara los flujos masicos en las caras*/
68. Puntero<Vector<double>> f_caras;
69. /**Campo a calcular*/
70. Puntero<CampoFVM> p;
71. /**Gradiente del campo a calcular(vectorial)*/
72. Puntero<CamposFVM> Grad;
73. /**Numero de grados de libertad*/
74. Puntero<GradoLibertadFV> gdl;
75. /**Estructura del vectr de términos solucion*/
76. Puntero<Vector<double>> VecP;
77. /**Puntero que guarda la densidad  en los elementos*/
78. Puntero<Vector<double>> Dens;
79. /**Puntero que guarda la densidad  en los elementos*/
80. Puntero<Vector<double>> DensAnt;
81. /**Para almacenar los gradientes de las tres velociades en cada punto
    de la malla*/
82. Puntero<Arreglo_Gen<real> > gradvels;
83. /**Almacenará la correcion velocidad(up)  y la velocidad corregida(Vel),
    Velociad nueva (Velnew) y veloicidad anterior(Vnm1)*/
84. Arreglo_Gen<double> up , Vel, Velnew, Vnm1;
85. /**Campo de velocidades*/
86. Puntero<CamposFVM> Vels; // Campo=escalar Campos=vectorial
```

```
87. /**Contiene las definiciones de las condiciones de frontera*/
88. Vector_basico<Puntero<BoCond> > bocos;
89. // Se utiliza para calcular el valor de la variable en la frontera( calculo del
    gradiente)
90. void CalculaFlujoFrontera(int cara, double *val, int &tipo);
91.
92. public:
93. void Leer(Is is);
94. void ResuelveProblema();
95. void ReportarResultados(int iteri);
96. protected:
97. void IniciarMemoria();
98. void CalculaUnaIteracion();
99. void CalculaIteraciones();
100.        void CalcularVelCaras();
101.        void CalcularGradiente();
102.        void CorrPprima();
103.        void CalcVPrima();
104.        void CorrVelocidad();
105.        void AsignarSigIteracion();
106.        void LeerBoCond();
107.        void ActualizaTiempo();
108.        void VelAyB(int p, int ladop, int dir, double *vals);
109.
110.  };
```

## 7. ARCHIVO DE IMPLEMENTACION DE LA CLASE NS

```
8.  #include "NS.h"
9.
10.
```

```
11. static const char *relajacion_tb[] = { "theta", "alphap", "alphav", "alphaw",
    "alphau", "relajacion", "iter_int", "error_max", "pinicial", NULL };
12. static const char *archivos_tb[] = { "archivos", "archbocond",
    "archivosolver", "carpeta" };
13. static const char *nombres_tb[] = { "nombrecampo", "nombregrad",
    "campo", "gradiente", "nombregradiente", "mallador", "velx", "vely",
    "velz", "corrp", NULL };
14. static const char *propiedades_tb[] = { "densidad", "cp", "rho", "k", "theta",
    "kdifusiva", "tinicial", "vinicial",
15. "campo", "gravedad", "esquema", "velocidad" };
16. static const char *frontera_tb[] = { "BoCo", "Cond_Frontera",
    "Boundary_Cond" };
17. static const char *tiempos_tb[] = { "time_parameters", "param_tiempo",
    "tiempo", NULL };
18.
19.
20.
21. NS::NS()
22. {
23. }
24.
25. NS::~NS()
26. {
27. }
28.
29. NS::NS(const char *nombre)
30. {
31. Cadena arch;
32. arch = aform("ARCH=%s", nombre);
33. Is is(arch.carts());
34. Leer(is);
35. }
```

```
36.
37. NS::NS(const Cadena nombre)
38. {
39. Cadena arch;
40. arch = aform("ARCH=%s", nombre.carts());
41. Is is(arch.carts());
42. Leer(is);
43. }
44.
45. void NS::Leer(Is is)
46. {
47. Cadena cadtiempo;
48. Cadena original = Path();
49. Cadena rutatrabajo;
50. is->obtComando(carpetatrabajo, archivos_tb);
51.
52. is->obtComando(cadmallador, nombres_tb);
53. malla.vincular(new MallaFV());
54. ConstOLeeMalla(malla(), cadmallador);
55. nbi = malla->obtNoIndFront();
56. tiempo.vincular(new prmTiempo());
57. is->obtComando(cadtiempo, tiempos_tb);
58. tiempo->Leer(cadtiempo);
59.
60. nsd = malla->obtNoDimEspacio();
61. vels.chaSize(nsd);
62. for (int i = 1; i <= nsd; i++){
63.     is->obtComando(cadtiempo, nombres_tb);
64.     vels(i).vincular(new CD(i, cadtiempo.carts(), nbi));
65.     vels(i)->malla.vincular(malla());
66.     vels(i)->tiempo.vincular(tiempo());
67.
```

```
68. }
69.
70. is->obtComando(cadtiempo, nombres_tb);
71. corrp.vincular(new CDifusion(cadtiempo.carts(), nbi));
72. corrp->malla.vincular(malla());
73. corrp->tiempo.vincular(tiempo());
74. is->obtComando(archbocond, archivos_tb);
75. is->obtComando(theta, relajacion_tb);
76. is->obtComando(iter, relajacion_tb);
77. is->obtComando(errormax, relajacion_tb);
78. is->obtComando(pinicial, relajacion_tb);
79. LeerBoCond();
80.
81. rutatrabajo = original + '\\' + carpetatrabajo;
82. MkDir(rutatrabajo);
83. ChDir(rutatrabajo);
84.
85. GuardarEnsight::Leer(is, nsd);
86.
87. is->obtComando(alphap, relajacion_tb);
88. alphav.chaSize(3);
89. for (int i = 1; i <= nsd; i++){
90.     is->obtComando(alphav(i), relajacion_tb);
91. }
92. forzarVisual(OFF);
93. }
94.
95. void NS::LeerBoCond()
96. {
97. Cadena cad;
98. bocos.chaSize(nbi);
99. Is ifs(archbocond.carts());                    //Is = input stream
```

```
100.
101.        for (int i = 1; i <= nbi; i++){
102.              ifs->obtComando(cad, frontera_tb);
103.              bocos(i).vincular(crearBoCond(cad, nsd));
104.              bocos(i)->Leer(ifs);
105.        }
106.
107.  }
108.
109.  void NS::IniciarMemoria()
110.  {
111.        nne = malla().obtNoElem();
112.        p.vincular(new CampoFVM(malla(), "p"));
113.        Grad.vincular(new CamposFVM(malla(), "grad_p"));
114.        gdl.vincular(new GradoLibertadFV(malla(), 1));
115.        VecP.vincular(new Vector<double>(nne));
116.        VecP().llenar(pinicial);
117.        vecino.vincular(new VecinoFVM());
118.        vecino().iniciar(malla());
119.        malla->CalcConectividad(vecino());
120.        elems = new Vector_basico<PunteroElmDefs>(nne);
121.        int e;
122.        for (e = 1; e <= nne; e++) {
123.              elems->operator()(e).refill(malla->obtTipoElem(e), nsd);
124.              elems->operator()(e)->CalcGeomParameters(malla-
    >obtCoorElem(e));
125.        }
126.        malla->CrearVectorCaras(*elems, 1);
127.
128.        gradvels.vincular(new Arreglo_Gen<real>());
129.        gradvels().chaSize(nsd, nsd, nne);
130.        gradvels().llenar(0.0);
```

```
131.
132.        int caras = malla->caras.tam();
133.        v_caras = new Vector_basico<Vector_xyz<double>>(caras);
134.        for (e = 1; e <= caras; e++){
135.              v_caras->operator()(e).chaSize(nsd);
136.              v_caras->operator()(e).llenar(0.0);
137.        }
138.        f_caras.vincular(new Vector<double>(malla->caras.tam()));
139.        Dens.vincular(new Vector<double>(nne));
140.
141.        for (int i = 1; i <= nsd; i++){
142.              Grad()(i).llenar(0.0);
143.              vels(i)->elems = elems;
144.              vels(i)->IniciaMemoria(i);
145.              vels(i)->v_caras = v_caras;
146.              vels(i)->f_caras.vincular(f_caras());
147.              vels(i)->Dens.vincular(Dens());
148.              if (DensAnt.ok())
149.                    vels(i)->DensAnt.vincular(DensAnt());
150.              vels(i)->AdjuntarGradVelocidad(gradvels());
151.              vels(i)->alpha = alphav(i);
152.              vels(i)->AdjuntarGradPresion(Grad());
153.
154.
155.        }
156.        corrp->elems = elems;
157.        corrp->IniciaMemoria(nsd);
158.        corrp->f_caras.vincular(f_caras());
159.        corrp->Ap.vincular(new Vector<double>(nne));
160.        corrp->Dens.vincular(Dens());
161.        if (DensAnt.ok())
162.              corrp->DensAnt.vincular(DensAnt());
```

```
163.        up.chaSize(nsd, nne);
164.        Vel.chaSize(nsd, nne);
165.        Velnew.chaSize(nsd, nne);
166.        Vnm1.chaSize(nsd, nne);
167.        up.llenar(0.0);
168.        Vel.llenar(0.0);
169.        Velnew.llenar(0.0);
170.        Vnm1.llenar(0.0);
171.        f_caras->llenar(0.0);
172.        Dens->llenar(vels(1)->densidad);
173.        Vels.vincular(new CamposFVM(malla(), "Vels"));
174.
175.
176.
177.
178.  }
179.
180.  void NS::CorrPprima()
181.  {
182.        int e;
183.        double alpha;
184.        for (e = 1; e <= nne; e++){
185.              corrp->VecSol(e) = corrp->VecSol(e)*alphap;
186.        }
187.
188.
189.        if (nne < 25){
190.              corrp->VecSol.Escribir(std_o, corrp->nombrecampo.carts(),
    3);
191.
192.        }
193.
```

```
194.        corrp->CalcularGradiente();
195.
196.  }
197.
198.  void NS::CalcVPrima()
199.  {
200.        int i, e;
201.        double vol, app, grad;
202.        for (e = 1; e <= nne; e++){
203.              vol = elems->operator()(e)->getVolume();
204.              for (i = 1; i <= nsd; i++) {
205.                    if (i == 1) app = vels(i)->Ap()(e) / alphav(i);
206.                    if (i == 2) app = vels(i)->Ap()(e) / alphav(i);
207.                    if (i == 3) app = vels(i)->Ap()(e) / alphav(i);
208.                    grad = corrp->Grad().operator()(i).values()(e);
209.                    up(i, e) = -grad*(vol / app);
210.                    Vel(i, e) = vels(i)->VecSol(e) + up(i, e);
211.              }
212.        }
213.        if (nne < 20)
214.        {
215.              up.Escribir(std_o, "up", 3);
216.              Vel.Escribir(std_o, "V_corr", 3);
217.        }
218.  }
219.
220.  void NS::CorrVelocidad()
221.  {
222.        int i, e;
223.        for (e = 1; e <= nne; e++){
224.              for (i = 1; i <= nsd; i++){
225.                    vels(i)->vnm1(e) = Vnm1(i, e);
```

```
226.                    Velnew(i, e) = alphav(i)*Vel(i, e) + (1.0 -
    alphav(i))*Vnm1(i, e);
227.            }
228.            p().values()(e) = VecP()(e) = VecP()(e)+corrp->VecSol(e);
229.        }
230.        if (nne < 25){
231.            Velnew.Escribir(std_o, "Vnew", 3);
232.            VecP->Escribir(std_o, "Presion", 3);
233.        }
234.
235.
236.  }
237.
238.  void NS::CalculaUnaIteracion() // primera iteracion
239.  {
240.        int i, e, nne;
241.        double ap;
242.        nne = malla().obtNoElem();
243.        for (i = 1; i <= nsd; i++){
244.            vels(i)->dt = dt;
245.            vels(i)->theta = theta;
246.            vels(i)->CalculaUnaIteracion();// Calculamos velocidades
247.        }
248.        for (e = 1; e <= nne; e++){
249.            ap = 0.0;
250.            for (i = 1; i <= nsd; i++){
251.                ap += vels(i)->Ap()(e) / alphav(i);
252.            }
253.            ap /= nsd;
254.            corrp->Ap()(e) = ap;
255.        }
256.
```

```
257.        CalcularVelCaras();
258.        corrp->dt = dt;
259.        corrp->theta = theta;
260.        corrp->CalculaUnaIteracion();
261.        CorrPprima();
262.        CalcVPrima();
263.        CorrVelocidad();
264.        CalcularGradiente();
265.        AsignarSigIteracion();
266.
267.
268.  }
269.
270.  void NS::CalculaIteraciones()
271.  {
272.        error = 1.0;
273.        int iterint = 0;
274.        int iterglb = 0;
275.        do{
276.              iterint++;
277.              iterglb++;
278.              CalculaUnaIteracion();
279.              error = corrp->VecCoef.norma();
280.              std_o << "\nIteracion" << iterint << "----Residuo =" << error <<
    "\n";
281.        } while ((iterint <= iter) && fabs(error) > errormax);// iteracion
    interna
282.
283.  }
284.
285.  void NS::AsignarSigIteracion(){
286.        double valor;
```

```
287.        int i, e, k;
288.        for (i = 1; i <= nsd; i++){
289.            for (e = 1; e <= nne; e++){
290.                vels(i)->VecSol(e) = Velnew(i, e);
291.                vels(i)->vnm1(e) = Vnm1(i, e) = Velnew(i, e);
292.                Vels()(i).values()(e) = Velnew(i, e);
293.
294.
295.            }
296.
297.        }
298.        for (i = 1; i <= nsd; i++)
299.            vels(i)->CalcularGradiente();
300.        for (i = 1; i <= nsd; i++){
301.            for (e = 1; e <= nne; e++){
302.                for (k = 1; k <= nsd; k++){
303.                    valor = vels(i)->Grad->operator()(k).values()(e);
304.                    gradvels()(i, k, e) = valor;
305.                }
306.
307.            }
308.
309.        }
310.        CalcularVelCaras();
311.  }
312.
313.  void NS::CalcularVelCaras()
314.  {
315.        int nc = malla->caras.tam();
316.        int cara, dir, conex, e, lado;
317.        double dens, flujo, fm, ap_p, ap_v, ap_i, pp, pe, area;
318.        double vola, volp, dpe;
```

```
319.        double vole;
320.        double valores[2];
321.
322.        Vector_xyz<real> vp(nsd), ve(nsd), v(nsd), vi(nsd), dPE(nsd);
323.        int i;
324.        Vector_xyz<real> Area(nsd), grp(nsd), gre(nsd), gri(nsd),
    grim(nsd);
325.        for (cara = 1; cara <= nc; cara++){
326.            ap_p = 0;
327.            ap_v = 0;
328.            e = malla->caras(cara).elemento;
329.            conex = malla->caras(cara).vecino;
330.            area = malla->caras(cara).area;
331.            Area = malla->caras(cara).Area;
332.            if (conex){ // cara interna
333.                fm = malla->caras(cara).fm;
334.                dPE = malla->caras(cara).dPE;
335.                pe = p().values()(conex);
336.                pp = p().values()(e);
337.                vola = elems->operator()(conex)->getVolume();
338.                volp = elems->operator()(e)->getVolume();
339.                dpe = malla->caras(cara).dpe;
340.                vole = area*dpe;
341.                for (i = 1; i <= nsd; i++) {
342.                    ap_p = vels(i)->Ap()(e) / alphav(i);
343.                    ap_v = vels(i)->Ap()(conex) / alphav(i);
344.                    vp(i) = vels(i)->VecSol(e);
345.                    ve(i) = vels(i)->VecSol(conex);
346.                    grp(i) = Grad()(i).values()(e);
347.                    gre(i) = Grad()(i).values()(conex);
348.                    ap_i = (1.0 - fm) / ap_p + fm / ap_v;
349.                    vi(i) = vp(i)*(1.0 - fm) + ve(i)*fm;
```

```
350.                    gri(i) = grp(i)*(1.0 - fm) + gre(i)*fm;
351.                    lado = malla->caras(cara).lado;
352.                    VelAyB(e, lado, i, valores);
353.                    vi(i) = (vi(i) + valores[0])*0.5;
354.                    gri(i) = (gri(i) + valores[1])*0.5;// calculo de gradiente
    esquema peric
355.
356.                    // Esquema Peric
357.                    if (dPE(i) != 0.0)
358.                        v(i) = vi(i) - vole*ap_i*((pe - pp) / dPE(i) - gri(i));
359.                    else
360.                        v(i) = vi(i) - vole*ap_i*(-gri(i));
361.                    //Esquema Malalasekera
362.                    //grim(i) = ((volp / ap_p*grp(i)*(1.0 - fm) + vola /
    ap_v*gre(i)*fm) + valores[1])*0.5;// dividir por 0.5 y establecer valores 1
363.                    /*if (dPE(i) != 0.0)
364.                        v(i) = vi(i) + ((1.0 - fm)*volp / ap_p + fm*vola /
    ap_v)*(pp - pe) / dPE(i) -grim(i);
365.                        else
366.                        v(i) = vi(i) - grim(i);*/
367.                    v_caras->operator()(cara)(i) = v(i);
368.                }
369.                //dens = vels(1)->CalcDensidad(e)*(1.0 - fm) + vels(1)-
    >CalcDensidad(conex)*fm;
370.                dens = vels(1)->densidad*(1.0 - fm) + vels(1)-
    >densidad*fm;
371.
372.            }
373.            else {
374.                int condfrontera = malla->caras(cara).bc;
375.                Vector_xyz<real> coc(nsd);
376.                if (condfrontera){
```

```
377.                        coc = malla->caras(cara).coc;
378.                        for (i = 1; i <= nsd; i++){
379.                              vp(i) = vels(i)->VecSol(e);
380.                              vels(i)->bocos(condfrontera)->calculo = _CVEL;
381.                              vels(i)->bocos(condfrontera)->coord = coc;
382.                              v(i) = vels(i)->bocos(condfrontera)->Evaluar();
383.                              int tipo = vels(i)->bocos(condfrontera)->ind;
384.                              if (tipo == 2){
385.                                    v(i) = vp(i) - v(i)*malla->caras(cara).dpe;
386.                              }
387.                              v_caras->operator()(cara)(i) = v(i);
388.                        }
389.                  }
390.                  //dens = vels(1)->CalcDensidad(e);
391.                  dens = vels(1)->densidad;
392.            }
393.            f_caras->operator()(cara) = v*Area*dens;
394.      }
395.      if (nne < 25){
396.            v_caras->Escribir(std_o, "V caras", 3);
397.            f_caras->Escribir(std_o, "f caras", 3);
398.      }
399. }
400.
401. void NS::CalcularGradiente()
402. {
403.
404.      int e, j, numconex, conex;
405.      int cara, tipo;
406.      Vector_xyz<double> area(nsd), grad(nsd);
407.      double vol;
408.      double valor;
```

```
409.        double fm;
410.        double valores[2];
411.        int nne = malla().obtNoElem();
412.
413.
414.        for (e = 1; e <= nne; e++){
415.              numconex = elems->operator()(e)->GetNumOfConex();
416.              grad.llenar(0.0);
417.              vol = elems->operator()(e)->getVolume();
418.
419.              for (j = 1; j <= numconex; j++){
420.                    cara = elems->operator()(e)->caras(j);
421.                    area = malla->caras(cara).Area;
422.                    conex = malla->getConx(e, j);
423.
424.
425.                    if (conex){ // cara que tiene vecino
426.                          fm = malla->caras(cara).fm;
427.                          valor = fm*VecP()(e)+(1.0 - fm)*VecP()(conex);
428.                          grad = grad + area*(valor / vol);
429.
430.
431.                    }
432.                    else {  // cara frontera
433.                          CalculaFlujoFrontera(cara, valores, tipo);
434.                          if (tipo == 1){
435.                                grad = grad + area*(valores[0] / vol);
436.                          }
437.                          else if (tipo == 2){
438.                                grad = grad + area*(VecP()(e)-valores[0] *
    malla->caras(cara).dpc) / vol;
439.                          }
```

```
440.
441.
442.                     }
443.
444.             } //end for j
445.             for (j = 1; j <= nsd; j++)
446.                 Grad()(j).values()(e) = grad(j);
447.
448.         }
449.         if (nne < 25){
450.             std_o << "\nGrad Presion \n";
451.             if (nsd >= 1)
452.                 Grad()(1).values().Escribir(std_o, "grad x", 3);
453.             if (nsd >= 2)
454.                 Grad()(2).values().Escribir(std_o, "grad y", 3);
455.             if (nsd >= 3)
456.                 Grad()(3).values().Escribir(std_o, "grad z", 3);
457.
458.         }
459. }
460.
461. void NS::CalculaFlujoFrontera(int cara, double *val, int &tipo)
462. {
463.         int bc = malla->caras(cara).bc;                    //El ya sabe
    con cuál condición de frontera esta ligado
464.         Vector_xyz<double> coord(nsd);
465.         if (bc){
466.             coord = malla->caras(cara).coc;                //coc cuál
    es el centroide de la cara
467.             bocos(bc)->coord = coord;
468.             bocos(bc)->Evaluar();
469.             tipo = bocos(bc)->ind;
```

```
470.            val[0] = bocos(bc)->valor;        //Aquí llega temperatura o
    flujo
471.            val[1] = bocos(bc)->valor2;       //Aquí llega el coeficiente de
    tc por convección, si es la cf mixta
472.        }
473.  }
474.
475.  void NS::ResuelveProblema()
476.  {
477.        IniciarMemoria();
478.        tiempo->iniciarCicloTiempo();
479.        std_o << "\nInicio simulacion";
480.        int iterint = 0;
481.        int iterglb = 0;
482.        double error, errorant;
483.        error = 1.0;
484.        ReportarResultados(0);
485.        while (!tiempo->finalizo() && fabs(error) > errormax) {
    //&&solucion_ex){
486.            errorant = error;
487.            iterint = 0;
488.            tiempo->incrementarTiempo();
489.            dt = tiempo->Delta();
490.            if (dt <= 0.0) dt = 1.0;
491.            if (tiempo->estacionario()) {
492.                theta = 0.0;          //Así  es la solución explícita
493.                dt = 1.0;
494.            }
495.            else
496.                std_o << "\n tiempo " << tiempo->obtTiempo() << "\n";
497.            do{
498.                iterint++;
```

```
499.                    iterglb++;
500.                    CalculaUnaIteracion();
501.                    error = corrp->VecCoef.norma();
502.                    //std_o << "\nIteracion" << iterglb << "----Residuo =" <<
    error << "\n";
503.            } while ((iterint <= iter || error > errorant) && fabs(error) >
    errormax);// iteracion interna
504.            std_o << "\nIteracion" << iterglb << "----Residuo =" << error
    << "\n";
505.            ReportarResultados(iterglb);
506.            ActualizaTiempo();
507.        }// El tiempo de solucion
508.
509.  }
510.
511.
512.
513.  void NS::ReportarResultados(int iteri){
514.        int i;
515.
516.        habadventenciasglb = false;
517.        forzarVisual(OFF);
518.        volcar(Vels(), tiempo.obtPtr());
519.        if (iteri == 0) guardacampos = FALSE;
520.        if (nsd >= 1)volcar(vels(1)->Grad(), tiempo.obtPtr());
521.        if (iteri == 0) guardacampos = FALSE;
522.        if (nsd >= 2)volcar(vels(2)->Grad(), tiempo.obtPtr());
523.        if (iteri == 0) guardacampos = FALSE;
524.        if (nsd >= 3)volcar(vels(3)->Grad(), tiempo.obtPtr());
525.        if (iteri == 0) guardacampos = FALSE;
526.        volcar(corrp->u(), tiempo.obtPtr());
527.        if (iteri == 0) guardacampos = FALSE;
```

```
528.        volcar(corrp->Grad(), tiempo.obtPtr());
529.        if (p.ok()) {
530.              if (iteri == 0)guardacampos = FALSE;
531.              volcar(p(), tiempo.obtPtr());
532.              if (iteri == 0)guardacampos = FALSE;
533.              volcar(Grad(), tiempo.obtPtr());
534.
535.        }
536.
537.  }
538.
539.  void NS::ActualizaTiempo(){
540.
541.        int i;
542.        for (i = 1; i <= nsd; i++){
543.              vels(i)->SolAnt = vels(i)->VecSol;
544.        }
545.        corrp->SolAnt = corrp->VecSol;
546.
547.
548.  }
549.
550.  void NS::VelAyB(int p, int ladop, int dir, double *vals){
551.
552.        int ll, j, nnodos, e;
553.        real result, suma, resgrad, sumagrad, vole, ap_e;
554.
555.        Vector_basico<int> nodos;
556.        Vector_basico<int> listaelm;
557.
558.        // Calculo de las coordenadas de a
559.        elems->operator()(p)->obtNodoEnLado(ladop, nodos);
```

```
560.        result = 0.0;
561.        resgrad = 0.0;
562.        nnodos = nodos.tam();
563.        for (j = 1; j <= nnodos; j++){
564.            suma = 0.0;
565.            sumagrad = 0.0;
566.            ll = malla->loc2glob(p, nodos(j));
567.            vecino->nodo(ll, listaelm);
568.            int ne = listaelm.tam();
569.            int k;
570.            for (k = 1; k <= ne; k++){
571.                e = listaelm(k);
572.                vole = elems->operator()(e)->getVolume();
573.                ap_e = vels(dir)->Ap()(e) / alphav(dir);
574.                suma += vels(dir)->VecSol(e);
575.                sumagrad += Grad()(dir).values()(e);//*vole / ap_e;
576.            }
577.            suma /= ne;
578.            sumagrad /= ne;
579.            result += suma;
580.            resgrad += sumagrad;
581.
582.        }
583.
584.        result /= nnodos;
585.        resgrad /= nnodos;
586.        vals[0] = result;
587.        vals[1] = resgrad;
588.  }
```

## 8. ARCHIVO DE CABECERA DE LA CLASE BDC

```
9.  #pragma once
10.#include <IsOs.h>
11.#include <Puntero_Base.h>
12.#include <GuardarEnsight.h>
13.#include <Mallador.h>
14.#include <GradoLibertadFV.h>
15.#include <AdmonEcLin.h>
16.#include "BoCond.h"
17.#include <math.h>
18.#include "TEvaluar.h"
19.#include <UtilidadesSisOp.h>
20.#include <ConstOLeeMalla.h>
21.#include "CD.h"
22.#include "CDifusion.h"
23.#include "NS.h"
24.
25.class NS;
26.class CD;
27.class BDC : public Puntero_Base, public GuardarEnsight
28.{
29.friend NS;
30.friend CD;
31.
```

```
32. public:
33. BDC();
34. virtual ~BDC();
35. BDC(const char *nombre);
36. BDC(const Cadena nombre);
37.
38. /**Datos de la clase*/
39. public:
40. double dt;
41. double theta;
42. int iter;
43. double errormax;
44. double error;
45.
46. Puntero<NS> ns;
47. Puntero<CD> ener;
48.
49.
50. /**Carpeta donde se descargan los archivos  */
51. Cadena carpetatrabajo;
52. Cadena NS1;
53. Cadena cde;
54.
55. protected:
56.
57.
58. /**Para simulacion temporal */
59. Puntero<prmTiempo> tiempo;
60. /**Generador de malla*/
61. Puntero<MallaFV> malla;
62. /**Vector que guardara los flujos masicos en las caras*/
63. Puntero<Vector<double>> f_caras;
```

```
64.
65.
66.
67. public:
68. void Leer(Is is);
69. void ResuelveProblema();
70. void ReportarResultados(int iteri);
71.
72. protected:
73. void IniciarMemoria1();
74. void CalculaUnaIteracion();
75. void CalculaDensidad();
76. void AsignarSigIteracion();
77. void ActualizaTiempo();
78. void ActualizaDensidad();
79. };
80.
```

**9. ARCHIVO DE IMPLEMENTACION DE LA CLASE BDC**

```
10. #include "BDC.h"
11.
12.
13. static const char *relajacion_tb[] = { "theta", "alphap", "alphav", "alphaw",
    "alphau", "relajacion", "iter_int", "error_max", "pinicial", NULL };
14. static const char *archivos_tb[] = { "archivos", "archbocond",
    "archivosolver", "carpeta" };
15. static const char *nombres_tb[] = { "nombrecampo", "nombregrad",
    "campo", "gradiente", "nombregradiente", "mallador", "velx", "vely",
16. "velz", "corrp", "temp", "NS", "CD", NULL };
17. static const char *propiedades_tb[] = { "densidad", "cp", "rho", "k", "theta",
    "kdifusiva", "tinicial", "vinicial",
```

```
18. "campo", "gravedad", "esquema", "velocidad" };
19. static const char *frontera_tb[] = { "BoCo", "Cond_Frontera",
    "Boundary_Cond" };
20. static const char *tiempos_tb[] = { "time_parameters", "param_tiempo",
    "tiempo", NULL };
21.
22.
23.
24. BDC::BDC()
25. {
26. }
27.
28.
29. BDC::~BDC()
30. {
31. }
32.
33. BDC::BDC(const char *nombre)
34. {
35. Cadena arch;
36. arch = aform("ARCH=%s", nombre);
37. Is is(arch.carts());
38. Leer(is);
39. }
40.
41. BDC::BDC(const Cadena nombre)
42. {
43. Cadena arch;
44. arch = aform("ARCH=%s", nombre.carts());
45. Is is(arch.carts());
46. Leer(is);
47. }
```

```
48.
49. void BDC::Leer(Is is)
50. {
51. habadventenciasglb = false;
52. int nbi;
53.
54. Cadena cadtiempo;
55. Cadena original = Path();
56. Cadena rutatrabajo;
57.
58. is->obtComando(carpetatrabajo, archivos_tb);
59. is->obtComando(NS1, nombres_tb);
60. is->obtComando(cde, nombres_tb);
61.
62. ns.vincular(new NS(NS1.carts()));
63. ChDir(original);
64. nbi = ns->nbi;
65. ener.vincular(new CD(0, cde.carts(), nbi));
66.
67. malla.vincular(ns->malla());
68. tiempo.vincular(ns->tiempo());
69. ener->malla.vincular(malla());
70. ener->tiempo.vincular(tiempo());
71.
72.
73.
74. rutatrabajo = original + '\\' + carpetatrabajo;
75. MkDir(rutatrabajo);
76. ChDir(rutatrabajo);
77. GuardarEnsight::Leer(is, malla->obtNoDimEspacio());
78. }
79.
```

```
80. void BDC::IniciarMemoria1()
81. {
82. Vector_xyz<double> coc;
83. ns->DensAnt.vincular(new Vector <double>(ns->malla->obtNoElem()));
84. ns->DensAnt().llenar(ns->vels(1)->densidad);
85. ns->IniciarMemoria();
86. ener->elems = ns->elems;
87. ener->IniciaMemoria(1);
88. ener->f_caras.vincular(ns->f_caras());
89. ener->alpha = 1;
90. ener->Dens.vincular(ns->Dens());
91. ener->DensAnt.vincular(ns->DensAnt());
92. /*for (int e = 1; e <= ns->nne; e++) {
93.    coc = ns->elems->operator()(e)->centroid();
94.    ns->VecP()(e) = ns->VecP()(e)-9.81*1.16*coc(2);
95. }
96. ns->CalcularGradiente();*/
97. }
98.
99. void BDC::CalculaUnaIteracion()
100.  {
101.        ns->dt = dt;
102.        ns->CalculaIteraciones();
103.        ener->dt = ns->dt;
104.        ener->theta = ns->theta;
105.        ener->CalculaUnaIteracion();
106.        CalculaDdensidad();
107.  }
108.
109.  void BDC::CalculaDensidad()
110.  {
111.        int e;
```

```
112.        double dens;
113.        double T;
114.        int nne = malla->obtNoElem();
115.        for (e = 1; e <= nne; e++){
116.
117.            T = ener->SolAnt(e);
118.            //dens = 100 / (0.287*(273 + T));
119.            dens = 1.1614 * (1-(T - 27) / (27+273));
120.            ns->Dens()(e) = dens;
121.
122.        }
123.        std_o << "Dens" << ns->Dens()(ns->nne / 2) << '\n';
124.        std_o << "Temp" << ener->VecSol(ns->nne / 2) << '\n';
125.
126.  }
127.
128.
129.
130.  void BDC::ResuelveProblema(){
131.
132.        IniciarMemoria1();
133.        ns->tiempo->iniciarCicloTiempo();
134.        std_o << "\nInicio simulacion";
135.        int iterint = 0;
136.        int iterglb = 0;
137.        double error, errorant;
138.        error = 1.0;
139.        ReportarResultados(0);
140.        while (!ns->tiempo->finalizo() && fabs(error) > ns->errormax) {
    //&&solucion_ex){
141.            errorant = error;
142.            //errorant = 0.00001;
```

```
143.            iterint = 0;
144.            ns->tiempo->incrementarTiempo();
145.            dt = ns->tiempo->Delta();
146.
147.            if (dt <= 0.0) dt = 1.0;
148.            if (ns->tiempo->estacionario()) {
149.                  theta = 0.0;          //Así  es la solución explícita
150.                  dt = 1.0;
151.            }
152.            else
153.                  std_o << "\n tiempo" << tiempo->obtTiempo() << "\n";
154.
155.            CalculaUnaIteracion();
156.            iterglb++;
157.            ReportarResultados(iterglb);
158.            ActualizaTiempo();
159.
160.      }// El tiempo de solucion
161.
162. }
163.
164. void BDC::ReportarResultados(int iteri){
165.
166.      habadventenciasglb = false;
167.      forzarVisual(OFF);
168.      ener->gdl->vector2campo(ener->VecSol, ener->u());
169.      if (iteri == 0) guardacampos = FALSE;
170.      volcar(ener->u(), tiempo.obtPtr());
171.      if (iteri == 0) guardacampos = FALSE;
172.      volcar(ns->Vels(), tiempo.obtPtr());
173.      if (iteri == 0) guardacampos = FALSE;
174.      if (ns->nsd >= 1)volcar(ns->vels(1)->Grad(), tiempo.obtPtr());
```

```
175.        if (iteri == 0) guardacampos = FALSE;
176.        if (ns->nsd >= 2)volcar(ns->vels(2)->Grad(), tiempo.obtPtr());
177.        if (iteri == 0) guardacampos = FALSE;
178.        if (ns->nsd >= 3)volcar(ns->vels(3)->Grad(), tiempo.obtPtr());
179.        if (iteri == 0) guardacampos = FALSE;
180.        volcar(ns->corrp->u(), tiempo.obtPtr());
181.        if (iteri == 0) guardacampos = FALSE;
182.        volcar(ns->corrp->Grad(), tiempo.obtPtr());
183.        if (ns->p.ok()) {
184.            if (iteri == 0)guardacampos = FALSE;
185.            volcar(ns->p(), tiempo.obtPtr());
186.            if (iteri == 0)guardacampos = FALSE;
187.            volcar(ns->Grad(), tiempo.obtPtr());
188.
189.        }
190.        if (iteri == 0) guardacampos = FALSE;
191.        volcar(ener->Grad(), tiempo.obtPtr());
192.
193.
194.
195.  }
196.
197.
198.  void BDC::ActualizaTiempo()
199.  {
200.        ns->ActualizaTiempo();
201.        ener->SolAnt = ener->VecSol;
202.        ns->DensAnt() = ns->Dens();
203.  }
204.
```