MODELADO Y SIMULACIÓN DEL FLUJO DE UN FLUIDO SOBRE UNA PLACA MEDIANTE VOLÚMENES FINITOS

MANUEL FERNANDO JEREZ CARRIZALES

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERÍAS FISICOMECÁNICAS
ESCUELA DE INGENIERÍA MECÁNICA
BUCARAMANGA
2012

MODELADO Y SIMULACIÓN DEL FLUJO DE UN FLUIDO SOBRE UNA PLACA MEDIANTE VOLÚMENES FINITOS

Presentado por:
MANUEL FERNANDO JEREZ CARRIZALES

Trabajo de grado como requisito para optar al título de
Ingeniero Mecánico

Director:
Ph.D. DAVID ALFREDO FUENTES

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERÍAS FISICOMECÁNICAS
ESCUELA DE INGENIERÍA MECÁNICA
BUCARAMANGA
2012

DEDICATORIA

Dedico este trabajo a mis padres, *Mariluz Carrizales* y *Manuel Jerez*,
por todo lo que me han enseñado, por los momentos que hemos vivido
y por la familia que hemos formado junto a mi hermana.

Dedico este trabajo a mi hermana, *Liliana Patricia*,
por ser una gran compañia a lo largo de mi vida.

"Existe un equilibrio a mantener,
optimización de recursos,
estabilidad de resultados,
que prefieres escoger,
un mundo desértico mal optimizado,
una naturaleza lejos del límite,
flora, fauna, tierra, agua, aire."

Manuel F.

AGRADECIMIENTOS

A la Universidad Industrial de Santander y a la Escuela de Ingeniería Mecánica por apoyar el desarrollo de este proyecto que genera un escalón más hacia el conocimiento de la CFD.

A David Fuentes por la asesoría brindada y el tiempo dedicado.

CONTENIDO

pág.

INTRODUCCIÓN **20**

1 DINÁMICA DE FLUIDOS **21**

1.1 CARACTERÍSTICAS DEL FLUJO Y DE LOS FLUIDOS 21

1.2 ECUACIONES DIFERENCIALES DE LA DINÁMICA DE FLUIDOS . . . 23

1.2.1 Principio de conservación de la masa 23

1.2.2 Ley de la conservación de la cantidad de movimiento 24

1.2.3 Ley de la conservación de la energía 24

1.3 ECUACIÓN DIFERENCIAL GENERAL EN LA DINÁMICA DE FLUIDOS 25

1.4 CLASIFICACIÓN DE LAS ECUACIONES DIFERENCIALES 25

1.5 FLUJO EN LA CAPA LÍMITE 26

1.5.1 Modelamiento matemático 28

1.5.2 Flujo fuera de la zona de la capa límite 31

1.5.3 Solución de Blasius de las ecuaciones de la capa límite. 31

1.5.4 Otros modelos matemáticos del flujo en la capa límite 33

2 MÉTODO DE LOS VOLÚMENES FINITOS (FVM) **36**

2.1 VENTAJAS Y DESVENTAJAS DEL FVM 36

2.2 PASOS GENERALES DEL FVM 37

2.3 MALLA 39

2.3.1 Malla estructurada 39

2.3.2 Malla no estructurada 41

2.3.3 Técnicas de definición del volumen de control 42

2.4 APROXIMACIÓN DE LAS VARIABLES . . . 43
2.4.1 Interpolación aguas-arriba (UDS) . . . 43
2.4.2 Interpolación lineal (CDS) . . . 44
2.4.3 Interpolación cuadrática (QUICK) . . . 46
2.5 CONDICIONES DE FRONTERA . . . 46
2.6 SOLUCIÓN DE LAS ECUACIONES . . . 47
2.6.1 Ecuaciones lineales . . . 47
2.6.2 Ecuaciones no lineales . . . 49
2.7 POST-PROCESAMIENTO . . . 50
2.8 CARACTERÍSTICAS A EVALUAR EN LA CFD . . . 50
2.9 METODOLOGÍA DE TRABAJO . . . 51

3 FLUJO DIFUSIVO EN ESTADO ESTABLE 53
3.1 IMPLEMENTACIÓN . . . 55
3.1.1 Método intuitivo . . . 55
3.1.2 Desarrollo mediante el uso de las librerías . . . 75
3.2 RESULTADOS . . . 86

4 FLUJO DIFUSIVO-CONVECTIVO 88
4.1 ESQUEMA DE APROXIMACIÓN . . . 89
4.1.1 Interpolación aguas-arriba . . . 89
4.1.2 Otros esquemas para la ecuación de difusión-convección . . . 90
4.2 IMPLEMENTACIÓN . . . 91
4.3 RESULTADOS . . . 103

5 ECUACIÓN TRANSITORIA DE LA D. Y LA C.D. 105
5.1 DIFUSIÓN EN ESTADO TRANSITORIO . . . 105
5.1.1 Implementación . . . 106

5.1.2 Resultados 117
5.2 CONVECCIÓN-DIFUSIÓN EN ESTADO TRANSITORIO 118
5.2.1 Implementación 118
5.2.2 Resultados 126

6 CFD DEL FLUJO EN LA CAPA LÍMITE 127

6.1 MÉTODO SIMPLE 128
6.1.1 Mallas colocadas, mallas desplazadas y cálculo del residuo másico . . 130
6.2 MODIFICACIONES AL MÉTODO SIMPLE 133
6.3 IMPLEMENTACIÓN SIMPLE 133
6.4 CASOS DESARROLLADOS DEL FLUJO EN LA CAPA LÍMITE POR MEDIO DE LAS ECUACIONES COMPLETAS DE NAVIER-STOKES . . . 176
6.4.1 Flujo en la capa límite para $0 \leq Re \leq 500000$ con las ecuaciones N.S. completas: NSCL 178
6.5 MÉTODO SIMPLIFICADO PARA LA SOLUCIÓN DE LAS ECUACIONES DE LA CAPA LÍMITE: NOSIMPLE 192
6.6 IMPLEMENTACIÓN 195
6.7 SIMULACIÓN DE LAS ECUACIONES SIMPLIFICADAS DE LA CAPA LÍMITE SOBRE UNA PLACA PLANA 199
6.7.1 Simulación de las ecuaciones de Blasius incluyendo el término adicional $\partial^2 u/\partial x^2$. 199
6.7.2 Comparación de la simulación con un método explicito para el flujo en la capa límite. 208
6.7.3 Simulación de las ecuaciones de Falkner-Skan para el punto de estancamiento. 210
6.8 ANÁLISIS DE RESULTADOS 214

7 CONCLUSIONES 216

8 RECOMENDACIONES Y OBSERVACIONES **217**

BIBLIOGRAFÍA **218**

ANEXOS **221**

LISTA DE TABLAS

pág.

Tabla 1 Valores de η, f, f' y f'' para la ecuación de Blasius. 32

Tabla 2 Métodos existentes en las librerías pdelib para resolver sistemas de ecuaciones lineales. 48

Tabla 3 Coeficientes de la ecuación de difusión para un volumen de control interno en 2D y una malla de tamaño de celda fijo. 56

Tabla 4 Modificación de los coeficientes debido a la condición Dirichlet en la frontera. 57

Tabla 5 Modificación de los coeficientes debido a la condición Neumann en la frontera. 57

Tabla 6 Valor del campo de temperatura para el problema propuesto al usar la clase diftemp. 74

Tabla 7 Coeficientes de la ecuación de difusión para un volumen de control interno. 75

Tabla 8 Coeficientes de la ecuación de difusión para las condiciones de frontera. 75

Tabla 9 Resumen de los coeficientes necesarios para aplicar la ecuación de difusión (Ec 3.11) a cada volumen de control. 76

Tabla 10 Valor del campo de temperatura para el problema propuesto al usar la clase diffgeneral. 85

Tabla 11 Función A(Pe) 90

Tabla 12 Distribución del campo de temperatura para el problema propuesto al usar la clase convdiff. 103

Tabla 13 Ecuaciones conservativas 127

Tabla 14 Comparación de resultados para el grosor de la capa límite hidráulica del caso NSCL. 183
Tabla 15 Comparación de resultados para el grosor de la capa límite térmica del caso NSCL. 183
Tabla 16 Comparación de resultados para $f''_{y=0}$ en el caso NSCL. 186
Tabla 17 Comparación de resultados para $C_{f,x}$ en el caso NSCL. 187
Tabla 18 Comparación de resultados para el coeficiente adimensional de transferencia de calor en el caso NSCL. 188
Tabla 19 Comparación de resultados para el grosor de la capa límite hidráulica del caso Blasius1. 201
Tabla 20 Comparación de resultados para el grosor de la capa límite térmica del caso Blasius1. 202
Tabla 21 Comparación de resultados para el coeficiente de fricción local $C_{f,x}$ del caso Blasius1. 204
Tabla 22 Comparación de resultados para v_{max} del caso Blasius1. 205
Tabla 23 Comparación de resultados para el coeficiente adimensional de transferencia de calor del caso Blasius1. 207
Tabla 24 Comparación de resultados para el espesor de la capa hidráulica del caso Falkner1. 213

LISTA DE FIGURAS

pág.

Figura 1 Clasificación de los fluidos de acuerdo al comportamiento reológico. . . 22

Figura 2 Perfil de velocidad y ancho de la capa límite. 26

Figura 3 Capa límite a través de un cilindro. 27

Figura 4 Distribución de la velocidad de acuerdo al modelo de Falkner-Skan. . . 34

Figura 5 Pasos generales para resolver un problema en el FVM. 37

Figura 6 Malla estructurada ortogonal de tamaño de celda fijo. 39

Figura 7 Malla ortogonal de tamaño de celda variable. 40

Figura 8 Malla estructurada no ortogonal. 40

Figura 9 Malla no estructurada. 41

Figura 10 Doble mallado. 41

Figura 11 Nodo en el centro de la celda ó cell-centered scheme. 42

Figura 12 Nodo en el vértice de la malla ó cell-vertex scheme. 43

Figura 13 Molécula general. 44

Figura 14 Interpolación aguas-arriba ó upwind interpolation. 45

Figura 15 Definición del problema de difusión en una placa plana. 55

Figura 16 Vectores unitarios $\hat{n}_i$ para un volumen de control rectangular. 56

Figura 17 Subíndices del elemento P. 58

Figura 18 Orden de ejecución de los bucles para el llenado de los coeficientes de la matriz. 68

Figura 19 Presentación de los resultados para la clase diftemp. 74

Figura 20 Presentación de los resultados para la clase diffgeneral. 86

Figura 21 Variación del número de volúmenes de control para diffgeneral. 86
Figura 22 Líneas isotérmicas para el problema de la difusión. 87
Figura 23 Temperatura a través de una trayectoria específica. 87
Figura 24 Problema ejemplo para la difusión-convección 91
Figura 25 Presentación de los resultados para la clase convdiff, $\Gamma = 0.01$. 104
Figura 26 Presentación de los resultados para la clase DifusionTransitoria 117
Figura 27 Presentación de los resultados para la clase ConveccionTransitoria. . . . 126
Figura 28 Método SIMPLE. 131
Figura 29 Posición del valor de la velocidad y de la presión en un VC para una malla colocada y una malla desplazada. 132
Figura 30 Método SIMPLEC. 134
Figura 31 Problema del flujo en la capa límite. 177
Figura 32 Condiciones de frontera para la ecuación de corrección de presión. . . . 177
Figura 33 Capa límite hidráulica δ_h del caso NSCL. 182
Figura 34 Capa límite térmica δ_{ter} del caso NSCL. 182
Figura 35 Perfil de velocidad u y temperatura T para $x = 4[m]$ 184
Figura 36 Velocidad v para NSCL 185
Figura 37 Valor f'' en la superficie para NSCL. 185
Figura 38 Coeficiente local de fricción para NSCL. 186
Figura 39 Término fuente para la ecuación de la cantidad de movimiento en dirección x. 187
Figura 40 Coeficiente adimensional de transferencia de calor para NSCL. 188
Figura 41 Presión a lo largo de la placa. 189
Figura 42 Norma del residuo másico 190
Figura 43 Norma del residuo másico sin tener en cuenta el volumen de control de referencia para la ecuación de corrección de presión 190

Figura 44 Variación de la velocidad u. 191
Figura 45 Máximo ΔP_{corr}. 192
Figura 46 Método simplificado para la solución de las ecuaciones de la capa límite sin realizar correción de presión, NOSIMPLE. 194
Figura 47 Espesor hidráulico del caso Blasius1. 200
Figura 48 Espesor hidráulico del caso Blasius1 en $x < 1.5[m]$. 201
Figura 49 Espesor de la capa límite térmica del caso Blasius1. 202
Figura 50 Perfil de velocidad del caso Blasius1. 203
Figura 51 Comparación del valor de $f''|_{y=0}$. 204
Figura 52 Comparación del valor del coeficiente de fricción local $C_{f,x}$. 205
Figura 53 Comparación de la velocidad v máxima. 206
Figura 54 Comparación del coeficiente adimensional de transferencia de calor. . . 206
Figura 55 Convergencia del residuo másico para Blasius1. 208
Figura 56 Convergencia de la velocidad para Blasius1. 208
Figura 57 Espesor de la capa límite térmica e hidráulica para 300x30 nodos. . . . 209
Figura 59 Porcentaje de error de la capa límite hidráulica. 209
Figura 58 Espesor de la capa límite térmica e hidráulica para 600x40 nodos. . . . 210
Figura 60 Espesor de la capa límite hidráulica del caso Falkner1. 212
Figura 61 Líneas de flujo para el caso Falkner1. 213

Figura 62 Método explícito para calcular $u_{i+1,j}$ y posteriormente $v_{i+1,j}$. 334
Figura 63 Resultados de un método explícito para hallar el grosor de la capa límite térmica e hidráulica sobre un placa plana 336
Figura 64 Dirección del flujo del fluido. 337
Figura 65 Verificación de la estabilidad del análisis. 337
Figura 66 Campos de las componentes x e y de la velocidad del fluido. 338
Figura 67 Campo de temperatura para el fluido. 338

LISTA DE ANEXOS

pág.

Anexo A. CLASES EN C++ . . . 221

Anexo B. SOLUCIÓN INTUITIVA DE LA ECUACIÓN DE DIFUSIÓN: ENCABEZADO DE LA CLASE . . . 228

Anexo C. SOLUCIÓN INTUITIVA DE LA ECUACIÓN DE DIFUSIÓN: ARCHIVO DE CÓDIGO FUENTE . . . 230

Anexo D. PROCESO DE SOLUCIÓN DE LA ECUACIÓN DE DIFUSIÓN A PARTIR DEL USO DE LAS LIBRERÍAS: ENCABEZADO DE LA CLASE . . . 235

Anexo E. PROCESO DE SOLUCIÓN DE LA ECUACIÓN DE DIFUSIÓN A PARTIR DEL USO DE LAS LIBRERÍAS: ARCHIVO DE CÓDIGO FUENTE . . . 237

Anexo F. ENCABEZADO DE LA CLASE CONVDIFF . . . 242

Anexo G. ARCHIVO DE CÓDIGO FUENTE DE LA CLASE CONVDIFF . . . 245

Anexo H. CLASE PARA LA SOLUCIÓN DE LA ECUACIÓN DE DIFUSIÓN EN ESTADO TRANSITORIO: ENCABEZADO DE LA CLASE . . . 252

Anexo I. CLASE PARA LA SOLUCIÓN DE LA ECUACIÓN DE DIFUSIÓN EN ESTADO TRANSITORIO: ARCHIVO DE CÓDIGO FUENTE . . . 255

Anexo J. CLASE PARA LA SOLUCIÓN DE LA ECUACIÓN DE CONVECCIÓN-DIFUSIÓN EN ESTADO TRANSITORIO: ENCABEZADO DE LA CLASE . . . 267

Anexo K. CLASE PARA LA SOLUCIÓN DE LA ECUACIÓN DE CONVECCIÓN-DIFUSIÓN EN ESTADO TRANSITORIO: ARCHIVO DE CÓDIGO FUENTE 270
Anexo L. MODIFICACIONES A LA CLASE CONVECCIONTRANSITORIA PARA SU USO EN LA CLASE NAVIERSTOKES 285
Anexo M. CLASE PARA EL CÁLCULO DEL FLUJO EN LA CAPA LÍMITE: ENCABEZADO DE LA CLASE 300
Anexo N. CLASE PARA EL CÁLCULO DEL FLUJO EN LA CAPA LÍMITE: ARCHIVO DE CÓDIGO FUENTE 305
Anexo O. MÉTODO EXPLÍCITO PARA LA SOLUCIÓN DE LAS ECUACIONES DE LA CAPA LÍMITE 333
Anexo P. CLASE PARA LA SOLUCIÓN EXPLICITA DE LAS ECUACIONES DE LA CAPA LÍMITE: ENCABEZADO DE LA CLASE 339
Anexo Q. CLASE PARA LA SOLUCIÓN EXPLICITA DE LAS ECUACIONES DE LA CAPA LÍMITE: ARCHIVO DE CÓDIGO FUENTE 341

RESUMEN

TÍTULO: MODELADO Y SIMULACIÓN DEL FLUJO DE UN FLUIDO SOBRE UNA PLACA MEDIANTE VOLÚMENES FINITOS[1]

AUTOR: MANUEL FERNANDO JEREZ CARRIZALES[2].

PALABRAS CLAVE: FVM, CAPA LÍMITE, SIMPLE, PLACA PLANA.

Se presentan dos métodos para resolver el problema del flujo en la capa límite sobre una placa plana. El primer método se basó en las ecuaciones completas de Navier-Stokes asumiendo un gradiente de presión desconocido, este gradiente fue calculado por el método SIMPLE. El segundo método se basó en las ecuaciones simplificadas de Blasius y se resolvió por un método de correción de la velocidad. En base a las librerías desarrolladas por Ph.D. David Fuentes para el método de los volúmenes finitos, se programó en C++ el método de solución para las ecuaciones planteadas. Se explica el código desarrollado y adicionalmente se enseña el modo de usar algunas librerías.
Las ecuaciones de la dinámica de fluidos se dividieron en dos grupos: ecuación de convección difusión y ecuación de difusión. Cada grupo de ecuaciones se calculó y resolvió de una forma diferente teniendo en cuenta el estado transitorio. Con el fin de verificar la funcionalidad de las dos clases se presenta para cada clase un ejemplo resuelto.
Una vez se explica cómo resolver los dos tipos básicos de ecuaciones, se une el trabajo realizado en una clase general llamada NavierStokes. La clase NavierStokes permite resolver mediante dos formas las ecuaciones no lineales de la mecánica de fluidos como un proceso iterativo de varias ecuaciones lineales. Los dos métodos de solución se implementaron para resolver una gran variedad de problemas de flujo laminar. Los resultados obtenidos de la simulación del flujo en la capa límite sobre una superficie plana en regimen laminar se comparan con la teoría desarrollada por Blasius.

[1]Trabajo de grado.
[2]Facultad de Ingenierías Fisicomecáncias, Escuela de Ingeniería Mecánica, Director Ph.D. David Fuentes.

ABSTRACT

TITLE: MODELLING AND SIMULATION OF FLOW OVER A FLAT PLATE BY FINITE VOLUME METHOD[1].

AUTHOR: MANUEL FERNANDO JEREZ CARRIZALES[2].

KEY WORDS: FVM, BOUNDARY LAYER, SIMPLE, FLAT PLATE.

Two methods for solving boundary layer problem over a flat plate have been implemented. The first method is based on the full Navier-Stokes equations by assuming an unknown gradient pressure, which is calculated by SIMPLE method. On the other hand, the second method is based on the simplified equation by Blasius and it is solved by using a velocity's correction method.
A finite volume method is used as approach. The FVM is programmed in C++ with Ph.D. Fuentes' libraries. The created code is explained in each instruction and it is taught how to use some libraries.
Equations in the FVM are classified into two groups: diffusion's equations and diffusion convection's equations. Each group is calculated and solved in a different class, where unsteady state is taking into account. Intermediate results are shown to check functionality in each group.
Once it is explained how to solve the two types of basics equations, created classes are modified to be part of a more general class called NavierStokes.
NavierStokes allows solving non-linear equations in an iterative process made by linear equations. Both semi-implicit method for pressure-linked equations and velocity's correction method are implemented to solve a great variety of laminar flow problems.
Results from the simulation of boundary layer over a flat plate are compared with theory made by Blasius.

[1]Work degree.
[2]Physical-Mechanical Engineering Faculty, Mechanical Engineering School, Director Ph.D. David Fuentes

INTRODUCCIÓN

El problema del flujo en la capa límite es un tema fundamental para el estudio de la transferencia de calor. Para el caso del flujo sobre una superficie plana se han descrito varias soluciones analíticas de acuerdo a una serie de parámetros. Son estas soluciones junto a la simplicidad del problema las que hacen del flujo en la capa límite un excelente candidato a ser simulado por medio del método de los volúmenes finitos.

La forma de operar del método de los volúmenes finitos puede ser vista como el balance de una propiedad a través de un volumen de control. El MVF no requiere del uso de variables de transformación, estas variables por lo general, convierten el problema de dinámica de fluidos computacional en un conjunto de ecuaciones diferenciales que posteriormente son discretizadas.

Para cumplir con los objetivos planteados se realiza la siguiente dristibución de los capítulos en el libro:

En el capítulo 1 se introduce el marco teórico y se desarrolla un modelo matemático de las ecuaciones simplificadas del flujo en la capa límite a partir de las ecuaciones completas de Navier-Stokes para flujo laminar a bajas velocidades. En el capítulo 2 se continua con el marco teórico pero en este caso se enfatiza en el método de los volúmenes finitos.

En los capítulos 3, 4 y 5 se presenta la discretización de dos tipos de ecuaciones, la ecuación de difusión y la ecuación de convección-difusión. El modelado a partir de estos dos tipos de ecuaciones permite desarrollar una gran cantidad de problemas de la dinámica de fluidos.

Finalmente, en el capítulo 6 se realiza la simulación del flujo en la capa límite a partir de dos modelos, el primer modelo tiene en cuenta las ecuaciones completas de Navier-Stokes y el segundo modelo resuelve las ecuaciones de Blasius.

El código desarrollado para cada una de las librerías creadas se explica en cada capítulo y se incluye de forma completa en los anexos del presente trabajo, incluyendo comentarios adicionales.

1. DINÁMICA DE FLUIDOS

La dinámica de fluidos es la ciencia que trata del comportamiento de los fluidos en reposo o en movimiento y de su interacción con sólidos o con otros fluidos en las fronteras.

El análisis de la dinámica de fluidos varía sustancialmente del nivel macroscópico al nivel microscópico. El estudio a nivel macroscópico está basado en la teoría del continuo. La **teoría del continuo** asume que no existen discontinuidades en la materia, por lo tanto, se pueden definir las propiedades del fluido como la densidad o la presión como funciones continuas para todo el espacio. La teoría del continuo no es válida para análisis microscópicos, región en la que debe ser considerado las interacciones entre las moléculas y los espacios vacíos existentes entre ellas. Para el desarrollo del presente trabajo de grado se asumirá la teoría del continuo ya que todos los análisis que se realizan son a nivel macroscópico.

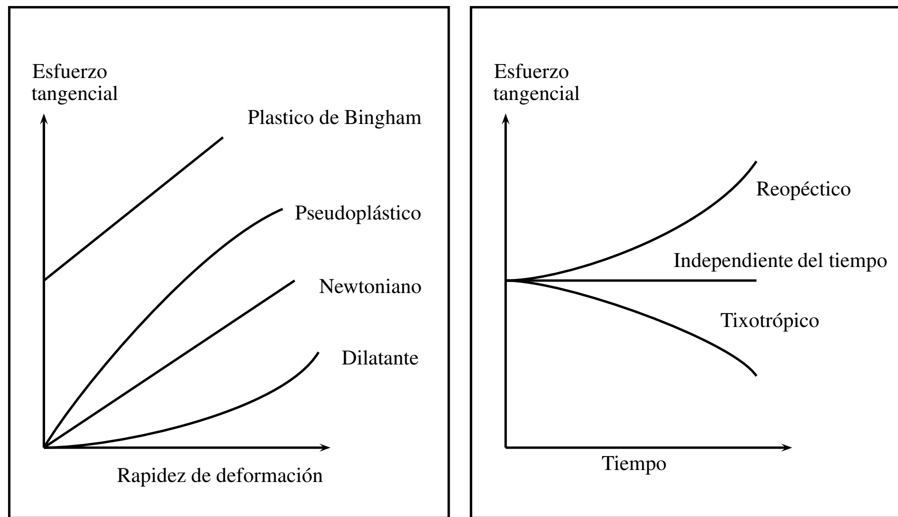
1.1 CARACTERÍSTICAS DEL FLUJO Y DE LOS FLUIDOS

Un fluido se caracteriza por no resistir esfuerzos de corte. Los fluidos pueden ser líquidos o gaseosos dependiendo del estado de la materia en el cual se encuentran. Existen varias maneras de clasificar los fluidos según el punto de vista o interés, se mencionan algunas de estas clasificaciones para ayudar a interpretar el problema que se va a desarrollar.

- Según la variación de la densidad.
  - Fluido compresible: es un fluido en el que la densidad varía convirtiendose en una función del tiempo o de la posición, para que existan variaciones en la densidad de un fluido debe existir una variación en la presión o la temperatura.
  - Fluido incompresible: es aquel en donde la densidad permanece relativamente constante. La variación de la densidad en un líquido causada por la presión es muy pequeña y de una manera similar pasa con la temperatura, por lo que en la mayoría de los casos se considera que los líquidos son incompresibles. Un gas no necesariamente es un fluido compresible, bajo ciertas condiciones, un gas se puede tratar como un fluido incompresible, especialmente en el caso en donde la variación de la presión es debida a la transformación de la energía cinética en energía de presión, si la velocidad del fluido es menor a un valor característico de dicho fluido se puede considerar un gas como fluido incompresible.
- Según el comportamiento reológico: la viscosidad puede variar a causa de varios factores, los dos factores más comunes son el esfuerzo de corte y el tiempo. En la figura

1(a) se muestra la variación del esfuerzo de corte con el aumento de la rapidez de deformación y en la figura 1(b) se muestra la variación del esfuerzo de corte con el tiempo. El presente trabajo será basado en fluidos newtonianos independientes del tiempo.

Figura 1: Clasificación de los fluidos de acuerdo al comportamiento reológico.



(a) Variación con la rapidez de deformación.

(b) Variación con el tiempo.

- Según la variación de las propiedades: en algunos casos, los cambios en las propiedades como la densidad y el Prandtl son lo suficientemente pequeños que se pueden despreciar, al despreciar estos cambios, el análisis de un problema suele ser más sencillo.
  - Fluido de propiedades constantes: las propiedades de un fluido no dependen de la posición, del tiempo u otros factores como campos magnéticos. Para que el fluido tenga propiedades constantes debe ser un elemento puro o una mezcla homogénea. Esta es una suposición válida para simplificar los problemas de ingeniería.
  - Fluido de propiedades variables: en este caso, no se pueden despreciar las variaciones de las propiedades ni aproximar a un único valor medio.
- Según las peturbaciones en el flujo
  - Flujo laminar: cuando el movimiento de un fluido se realiza en capas paralelas que no se mezclan se dice que el flujo es laminar.
  - Flujo turbulento: es un flujo que se caracteriza por la formación de remolinos y el mezclado del fluido.

  - Flujo de transición: en algunos casos el flujo laminar se puede convertir en flujo turbulento en una zona de transición que se caracteriza por la generación de remolinos a través de las capas laminares.

- Según la variación en el tiempo
  - Flujo en estado estable: todas las propiedades son constantes a través del tiempo, puede existir una variación de las propiedades con respecto a la posición.
  - Flujo transitorio: los efectos transitorios (derivadas en el tiempo) son significativos y deben tenerse en cuenta. En algunos casos se puede considerar un flujo transitorio como un flujo en estado estable, esta aproximación se puede llevar a cabo si se quiere realizar un análisis general para un proceso cíclico.

1.2 ECUACIONES DIFERENCIALES DE LA DINÁMICA DE FLUIDOS

Las ecuaciones de la dinámica de fluidos se pueden construir usando el enfoque de Euler o el enfoque de Lagrange, cada uno de estos enfoques produce una ecuación para cada principio de conservación que, aunque parece diferente, puede ser transformada de tal manera que se llega al mismo resultado presentado por el otro enfoque. En el presente trabajo se usará el enfoque de Euler que determina las propiedades como la velocidad en un punto fijo en el espacio; las ecuaciones generadas al utilizar el enfoque de Euler se llaman **ecuaciones conservativas**.

La formulación matemática que se presenta a continuación está basada en tres principios de conservación[1].

1.2.1 Principio de conservación de la masa

En realidad la conservación de la masa es un principio mas no una ley, lo anterior es debido a que la masa puede transformarse en energía, a modo de ejemplo, si se considera la cantidad de masa requerida para proporcionar una energía igual en magnitud a la energía cinética, el valor de la masa sería tan pequeño que se requerirían procesos especiales para realizar su medición [2].

En el presente trabajo, en el cual se realiza un análisis macroscópico y no se tienen en cuenta reacciones nucleares ni químicas, es válido utilizar el principio de conservación de la masa

[1] Aunque existen otros principios, como la ley de especies químicas, no serán comentados dado que están fuera del alcance del presente proyecto.

[2] CENGEL, Y.A. Y BOLES, M. Termodinámica: un enfoque práctico, 5 ed. México: McGraw Hill, 2007. p.71.

que se muestra en la (Ec 1.1).

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

1.2.2 Ley de la conservación de la cantidad de movimiento

A partir de la segunda ley de Newton, se deduce que la sumatoria de las fuerzas aplicadas a un elemento infinitesimal de fluido es igual a la variación en el tiempo de la cantidad de movimiento (Ec 1.2).

$$\frac{\partial (\rho u)}{\partial t} + \nabla \cdot (\rho u \, \vec{V}) = -(\nabla p) \cdot \widehat{i} + \rho \vec{f} \cdot \widehat{i} + (\nabla \tau) \cdot \widehat{i} \qquad \text{(Ec 1.2a)}$$

$$\frac{\partial (\rho v)}{\partial t} + \nabla \cdot (\rho v \, \vec{V}) = -(\nabla p) \cdot \widehat{j} + \rho \vec{f} \cdot \widehat{j} + (\nabla \tau) \cdot \widehat{j} \qquad \text{(Ec 1.2b)}$$

$$\frac{\partial (\rho w)}{\partial t} + \nabla \cdot (\rho w \, \vec{V}) = -(\nabla p) \cdot \widehat{k} + \rho \vec{f} \cdot \widehat{k} + (\nabla \tau) \cdot \widehat{k} \qquad \text{(Ec 1.2c)}$$

Para fluidos newtonianos, τ es

$$\boldsymbol{\tau} = \begin{bmatrix} \tau_{xx} & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \tau_{yy} & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \tau_{zz} \end{bmatrix} \qquad \begin{matrix} \tau_{xx} = \lambda\,(\nabla \cdot \vec{V}) + 2\,\mu\,\frac{\partial u}{\partial x} & \tau_{yy} = \lambda\,(\nabla \cdot \vec{V}) + 2\,\mu \frac{\partial v}{\partial y} \\ \tau_{zz} = \lambda\,(\nabla \cdot \vec{V}) + 2\,\mu\,\frac{\partial w}{\partial z} & \tau_{yz} = \tau_{zy} = \mu\,(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}) \\ \tau_{xy} = \tau_{yx} = \mu\,(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}) & \tau_{xz} = \tau_{zx} = \mu\,(\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z}) \end{matrix}$$

En donde

$$\lambda = -\frac{2}{3}\,\mu \qquad \text{Hipótesis de Stokes}$$

1.2.3 Ley de la conservación de la energía

La ley de la conservación de la energía o la primera ley de la termodinámica explica que la energía no puede crearse ni destruirse, solo se transforma. La primera ley de la termodinámica puede expresarse en función de la energía interna como (Ec 1.3) o en función de la entalpía como (Ec 1.4).

$$\frac{\partial}{\partial t}\left[\rho\left(e + \frac{V^2}{2}\right)\right] + \nabla \cdot \left[\rho\left(e + \frac{V^2}{2}\right)\right] = \nabla \cdot (k \nabla T) + \rho q - \nabla \cdot (p\vec{V}) + \Phi + \rho \vec{f} \cdot \vec{V} \qquad \text{(Ec 1.3)}$$

$$\frac{\partial (\rho (h + \frac{V^2}{2}))}{\partial t} + \nabla \cdot \left[\rho\,(h + \frac{V^2}{2})\vec{V}\right] = \nabla \cdot (k \nabla T) + \rho q + \Phi + \rho \vec{f} \cdot \vec{V} \qquad \text{(Ec 1.4)}$$

para

$$\Phi = \mu\,\left[2\,\left(\frac{\partial u}{\partial x}\right)^2 + 2\,\left(\frac{\partial v}{\partial y}\right)^2 + 2\,\left(\frac{\partial w}{\partial z}\right)^2 + \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial w}{\partial y} + \frac{\partial v}{\partial z}\right)^2 + \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right)^2\right] + \lambda\,\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right)^2 = \nabla \cdot (\boldsymbol{\tau}\,\vec{V})$$

1.3 ECUACIÓN DIFERENCIAL GENERAL EN LA DINÁMICA DE FLUIDOS

La conservación de la masa, cantidad de movimiento y energía pueden ser escritas de una manera general como se muestra en la (Ec 1.5) utilizando cuatro términos: el término transitorio, el término convectivo, el término difusivo y el término fuente.

$$\underbrace{\frac{\partial(\rho\,\phi)}{\partial t}}_{término\ transitorio} + \underbrace{\nabla\cdot(\rho\,\phi\,\vec{V})}_{término\ convectivo} = \underbrace{\nabla\cdot(\Gamma\,\nabla\phi)}_{término\ difusivo} + \underbrace{S}_{término\ fuente} \qquad \text{(Ec 1.5)}$$

Donde

Ec. de continuidad	$\phi = 1$	$\Gamma = 0$	$S = 0$
Ec. cant. mvto.	$\phi = u_i$	$\Gamma = \mu$	$S = -\frac{\partial p}{\partial x_i} + \rho B_i + S_{u,i}$
Ec. de energía	$\phi = h$	$\Gamma = k/C_p$	$S = \frac{1}{C_p}\rho q + \frac{1}{C_p}\nabla\cdot(\boldsymbol{\tau}\cdot\vec{V}) + \frac{1}{C_p}\rho\vec{f}\cdot\vec{V}$

En el presente trabajo se usará una versión simplificada de la **ecuación de la energía en términos de la temperatura** (Ec 1.6) puesto que en la ingeniería mecánica es bastante familiar el concepto del gradiente de la temperatura que está relacionado con el flujo de calor. La (Ec 1.6) fue obtenida al asumir que $C_p = \ constante$, y $h = C_p\,(T - T_{ref})$.

$$\frac{\partial(\rho\,T)}{\partial t} + \nabla\cdot(\rho\,T\,\vec{V}) = \nabla\cdot(\frac{k}{C_p}\,\nabla T) + \frac{q}{C_p} \qquad \text{(Ec 1.6a)}$$

$$\phi\ =\ T \qquad \Gamma\ =\ \frac{k}{C_p} \qquad S\ =\ \frac{q}{C_p} \qquad \text{(Ec 1.6b)}$$

En la investigación de la CFD es común usar la ecuación general para varios valores de Γ como prueba para los métodos seleccionados.

1.4 CLASIFICACIÓN DE LAS ECUACIONES DIFERENCIALES

La propagación de la información de un punto se puede realizar en distintas direcciones de acuerdo al tipo de ecuación que gobierne el problema. Para aclarar las posibles formas de propagación, se definen los siguientes tres grupos:

1. Ecuaciones hiperbólicas (Ec 1.7): en este tipo de ecuación, la información se propaga en dos direcciones, se debe definir una condición para cada frontera para resolver la ecuación. El ejemplo más común de este tipo de ecuación es el flujo supersónico.

$$\frac{\partial^2\phi}{\partial x^2} + \alpha\,\frac{\partial^2\phi}{\partial y^2} + \beta\frac{\partial\phi}{\partial x} + \gamma\frac{\partial\phi}{\partial y} = f \qquad \alpha < 0 \qquad \text{(Ec 1.7)}$$

2. Ecuaciones parabólicas (Ec 1.8): la propagación de la información se realiza aguas-abajo y por esta razón, solo se necesita una condición en la frontera aguas-arriba para resolver las ecuaciones. Los dos ejemplos más comunes de este tipo de ecuación son el flujo en la capa límite y cualquier ecuación dependiente del tiempo.

$$\frac{\partial^2\phi}{\partial x^2}+\beta\frac{\partial\phi}{\partial x}+\gamma\frac{\partial\phi}{\partial y}=f \qquad \alpha=0 \qquad \text{(Ec 1.8)}$$

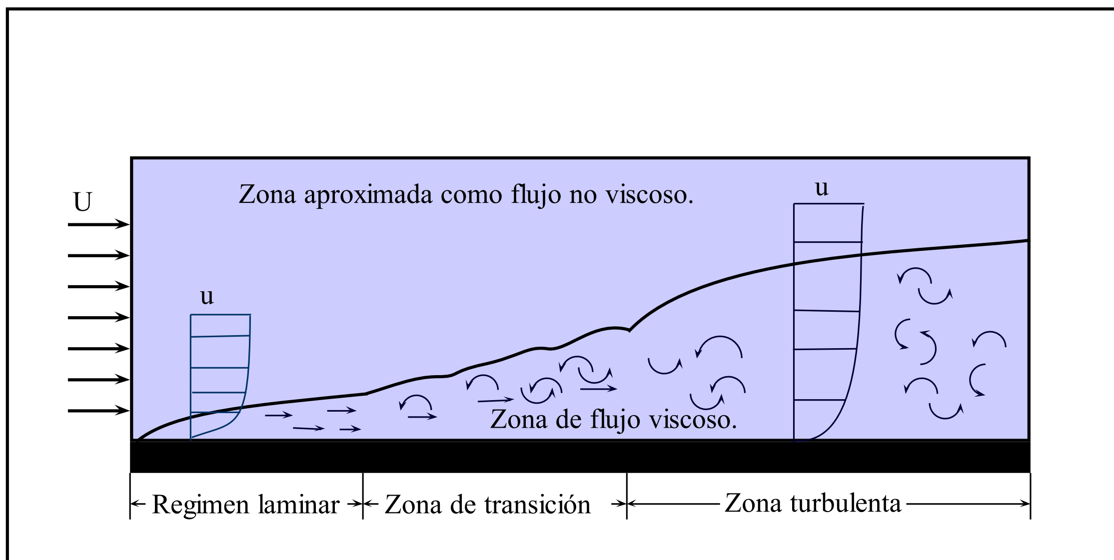
3. Ecuaciones elípticas (Ec 1.9): en este tipo de ecuación la información de un punto se propaga en todas las direcciones, es necesario definir una condición en cada frontera. La ecuación de Poisson es el ejemplo más común de este tipo.

$$\frac{\partial^2\phi}{\partial x^2}+\alpha\frac{\partial^2\phi}{\partial y^2}+\beta\frac{\partial\phi}{\partial x}+\gamma\frac{\partial\phi}{\partial y}=f \qquad \alpha>0 \qquad \text{(Ec 1.9)}$$

1.5 FLUJO EN LA CAPA LÍMITE

El problema del flujo en la capa límite consiste en encontrar el coeficiente de fricción y el coeficiente de transferencia de calor por convección (locales y promedios) así como también el grosor de la capa hidráulica, el flujo de calor o definir la zona en la cual el fluido no se ha visto afectado por la superficie, ya sea hidráulica o térmicamente. En la figura 2 se presenta un perfil típico de la velocidad y el ancho de la capa límite para el flujo sobre una placa plana.

Figura 2: Perfil de velocidad y ancho de la capa límite.

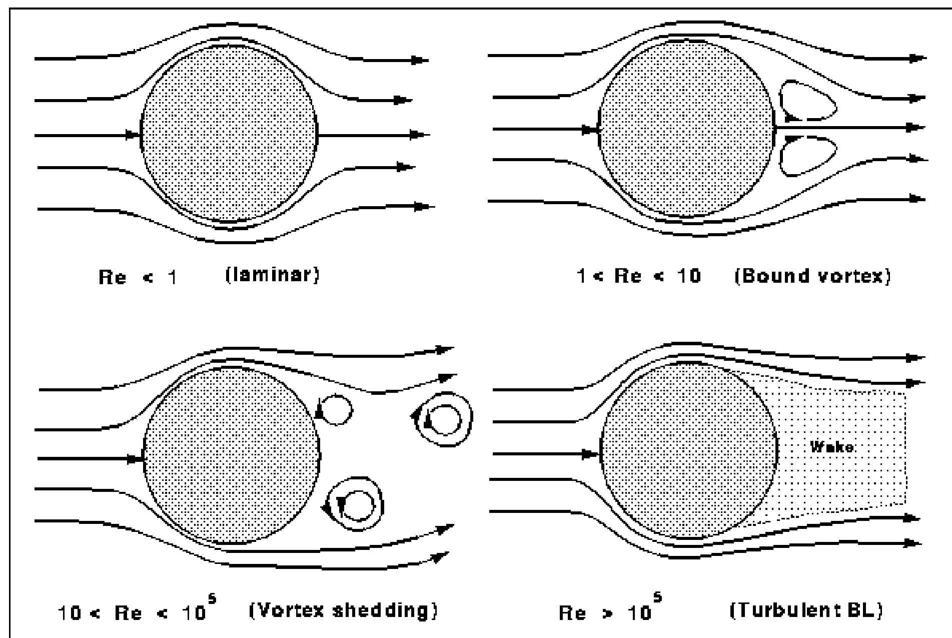


El cálculo de la capa límite no se limita al flujo sobre una placa plana, es común el análisis del flujo a través de un cilindro (figura 3) o de perfiles de ala de avión; en flujos internos,

se analizan ductos de aire e incluso se puede tener en cuenta la mezcla de dos flujos y su evolución a través de la tubería. El flujo en la capa límite tampoco está limitado al flujo sobre una superficie, más allá del borde de salida del ala de un avión se puede producir el fenómeno de la capa límite en donde la "superficie" (una línea imaginaria que no es atravesada por el flujo del fluido) es el mismo fluido.

En muchos casos, existe en la capa límite una región de desprendimiento del flujo. En esta región, el gradiente de presión produce la inversión de la velocidad del fluido, se puede encontrar este fenómeno en geometrías con cambios drásticos como cilindros o difusores que provocan un aumento en el gradiente de presión.

Figura 3: Capa límite a través de un cilindro.



Fuente:
http://www.see.ed.ac.uk/ johnc/teaching/fluidmechanics4/2003-04/fluids14/image41.gif

La teoría del flujo en la capa límite se basa en la **condición de no deslizamiento**, es decir, que el fluido que está en contacto con la superficie tiene la velocidad de esta. La condición de no deslizamiento es válida para análisis macroscópicos pero a nivel miscroscópico se debe realizar otro planteamiento.

El **grosor de la capa límite hidráulica** es la distancia a la superficie en donde el fluido tiene el 99% de la velocidad de entrada. Si se define y como la distancia de un punto a la superficie y u_∞ o V como la velocidad del fluido lejos de la superficie, el grosor de la capa límite hidráulica se puede definir como (Ec 1.10). La línea generada por δ_h divide el análisis en dos

zonas, la zona de flujos viscosos, también llamada capa límite hidráulica, es la zona más cercana a la superficie; más alla de esta zona, se encuentra la zona de flujo no viscoso en donde los efectos de la fricción son despreciables, esta zona se puede modelar matemáticamente con la ecuación de Euler.

$$\delta_h = y|_{u=0.99\,u_\infty} = y|_{u=0.99\,V} \qquad [m] \qquad \text{(Ec 1.10)}$$

El **coeficiente de fricción** (Ec 1.11) es el cociente entre el esfuerzo de corte en la superficie y la energía cinética por unidad de volumen. El coeficiente de fricción puede ser local si corresponde a un solo punto de la superficie o global si es un promedio para una región en la superficie.

$$C_f = \frac{\tau_s}{\rho\, V^2/2} = \frac{2}{\rho\, V^2}\,\mu\,\frac{\partial u}{\partial y}|_{y=0} \qquad \text{[adimensional]} \qquad \text{(Ec 1.11a)}$$

$$C_{f,\,prom} = \frac{\tau_{s,\,prom}}{\rho\, V^2/2} \qquad \text{(Ec 1.11b)}$$

En los casos en donde se debe tener en cuenta la transferencia de calor, resulta conveniente definir el **grosor de la capa límite térmica** (Ec 1.12) de forma análoga al grosor de la capa límite hidráulica.

$$\delta_t = y|_{T-T_{sup}=0.99\,(T_\infty - T_{sup})} \qquad [m] \qquad \text{(Ec 1.12)}$$

El **flujo de calor** en la superficie es hallado a partir de la conducción de calor en el fluido cerca a la superficie, por lo tanto, se usa la (Ec 1.13) para hallar el flujo de calor, este valor puede ser local o global.

$$q_s = -\,k\,\frac{\partial T}{\partial y}|_{y=0} \qquad [W/m^2] \qquad \text{(Ec 1.13)}$$

El **coeficiente de transferencia de calor por convección** (Ec 1.14) se define con base en el flujo de calor en la superficie y la diferencia de temperatura existente. El valor de h puede ser expresado indirectamente de una forma más general a partir del coeficiente adimensional de transferencia de calor o número de Nusselt (Ec 1.15).

$$h = \frac{q_s}{T_s - T_\infty} \qquad [W/(m^2\,C)] \qquad \text{(Ec 1.14)}$$

$$Nu = \frac{h\,L_c}{k} \qquad [adimensional] \qquad \text{(Ec 1.15)}$$

1.5.1 Modelamiento matemático del flujo en la capa límite sobre una placa plana

Para poder analizar el flujo en la capa límite sobre una placa plana se deben tener en cuenta las siguientes simplificaciones al modelo matemático comúnmente conocidas como **aproximaciones de la capa límite** (Ec 1.16).

$$v \ll u \qquad \text{(Ec 1.16a)}$$

$$\frac{\partial v}{\partial x} \approx 0 \quad \text{(Ec 1.16b)}$$

$$\frac{\partial v}{\partial y} \approx 0 \quad \text{(Ec 1.16c)}$$

$$\frac{\partial u}{\partial x} \ll \frac{\partial u}{\partial y} \quad \text{(Ec 1.16d)}$$

$$\frac{\partial T}{\partial x} \ll \frac{\partial T}{\partial y} \quad \text{(Ec 1.16e)}$$

La velocidad del fluido en la zona de entrada tiene únicamente componente en u, la velocidad v es generada por la ecuación de continuidad al disminuir gradualmente el valor de u, por esta razón, v es mucho menor que u (Ec 1.16a). Las aproximaciones de (Ec 1.16b) y (Ec 1.16c) se tienen en cuenta solamente al analizar la cantidad de movimiento en y, en general estos valores son muy pequeños. La variación de la velocidad u (Ec 1.16d) es mucho más significativa en el eje y que en la dirección del flujo de entrada y sucede de igual forma con la variación de la temperatura (Ec 1.16e).

En esta sección se utilizarán dos números adimensionales, el Reynolds (Re) y el Prandtl (Pr). El número de Reynolds (Ec 1.17) depende del flujo y de las propiedades del fluido, en cambio, el número de Prandtl (Ec 1.18) depende solamente de las propiedades del fluido.

$$Re = \frac{\text{Fuerzas de inercia}}{\text{Fuerzas viscosas}} = \frac{V\,L_c}{\nu} \quad \text{(Ec 1.17)}$$

$$Pr = \frac{\substack{\text{Difusividad molecular} \\ \text{de la cantidad de movimiento}}}{\text{Difusividad molecular del calor}} = \frac{\nu}{\alpha} \quad \text{(Ec 1.18)}$$

Además de las anteriores simplificaciones se tendrá en cuenta:

- Estado estable, no se considerará las variaciones en el tiempo como se muestra en la siguiente ecuación:

$$\frac{\partial T}{\partial t} = 0 \qquad \frac{\partial \vec{V}}{\partial t} = \mathbf{0}$$

- Flujo laminar, el análisis solo se realizará en el flujo laminar, no se tendrá en cuenta la zona de transición y ni la zona de turbulencia. Para el flujo sobre una placa plana se debe cumplir la siguiente condición para garantizar que el flujo sea laminar.

$$Re \leq 5x10^5$$

- Validez de las aproximaciones de la capa límite (Ec 1.16): es necesario satisfacer un valor mínimo de Re para que el modelo matemático sea coherente con la realidad.

$$Re > 1000$$

- Flujo a bajas velocidades[3], la variación de la densidad debido a cambios en la presión se considera despreciable [30]. Para definir matemáticamente esta condición se usa el número de Mach con la siguiente restricción:

$$Mach \leq 0.3$$

A continuación se presentan las ecuaciones de la capa límite obtenidas a partir de las simplificaciones de (Ec 1.1), (Ec 1.2) y (Ec 1.4):

Ecuación de continuidad: se asumen densidad constante y estado estable.

$$\cancelto{0}{\frac{\partial \rho}{\partial t}} + \nabla \cdot (\rho \vec{V}) \Rightarrow \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \qquad \text{(Ec 1.19)}$$

Ecuación de cantidad de movimiento: para el caso del flujo en la capa límite sobre una superficie plana solo es necesario analizar la ecuación de la cantidad de movimiento en dirección x[4] en donde, una vez realizadas algunas simplificaciones se obtiene (Ec 1.20).

$$\nabla \cdot (\rho u \cdot \vec{V}) = -\frac{\partial p}{\partial x} + (\nabla \tau) \cdot \hat{i}$$

$$\nabla \cdot (\rho u \cdot \vec{V}) = \frac{\partial(\rho uu)}{\partial x} + \frac{\partial(\rho uv)}{\partial y} = 2\rho u \frac{\partial u}{\partial x} + \rho u \frac{\partial v}{\partial y} + \rho v \frac{\partial u}{\partial y} = \rho u \cancelto{0}{\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)} + \rho u \frac{\partial u}{\partial x} + \rho v \frac{\partial u}{\partial y}$$

$$(\nabla \tau) \cdot \hat{i} = \frac{\partial}{\partial x}\left[\lambda(\cancelto{0}{\nabla \cdot \vec{V}})\right] + \frac{\partial}{\partial x}\left[2\mu \frac{\partial u}{\partial x}\right] + \frac{\partial}{\partial y}\left[\mu\,(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x})\right] =$$

$$= \mu \frac{\partial^2 u}{\partial x^2} + \mu \frac{\partial}{\partial x}\left[\cancelto{0}{\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}}\right] + \mu \frac{\partial^2 u}{\partial y^2} = \mu \frac{\partial^2 u}{\partial x^2} + \mu \frac{\partial^2 u}{\partial y^2} \approx \mu \frac{\partial^2 u}{\partial y^2}$$

$$\rho u \frac{\partial u}{\partial x} + \rho v \frac{\partial u}{\partial y} = -\frac{dp}{dx} + \mu \frac{\partial^2 u}{\partial y^2} \qquad \text{(Ec 1.20)}$$

En (Ec 1.20) se utiliza la notación de derivada exacta de la presión porque esta no depende del eje y ni del tiempo. Normalmente, el valor de la derivada de la presión es dado como un dato de entrada[5]. Para determinar el valor de $\frac{dp}{dx}$ es suficiente con hacer el cálculo fuera de la región de la capa límite en donde es posible usar la ecuación de Euler para flujo no viscoso.

[3] No existen ondas de choque en el flujo a bajas velocidades.

[4] La ecuación de la cantidad de movimiento en dirección y da como resultado que la presión es independiente del eje y, esto es posible utilizando las simplificaciones de (Ec 1.16) y sin tener en cuenta la fuerza gravitatoria.

[5] Si la presión o la derivada no se da como un valor de entrada, se debería definir la velocidad de salida para calcular la presión.

Ecuación de conservación de la energía: los primeros términos a eliminar son los términos transitorios, la generación de calor, la energía cinética y Φ. Para flujos a bajas velocidades se pueden despreciar la transformación de la energía cinética y el término Φ, la entalpía y la conductividad térmica son los dos efectos a considerar en el balance de la energía. Teniendo en cuenta la última aproximación de (Ec 1.16) se obtiene (Ec 1.21).

$$\cancelto{0}{\frac{\partial(\rho(h+\frac{V^2}{2}))}{\partial t}}+\nabla\cdot\left[\rho\,(h+\cancelto{0}{\frac{V^2}{2}})\vec{V}\right]=\nabla\cdot(k\nabla T)+\cancelto{0}{\rho q}+\cancelto{0}{\Phi}$$

$$\rho\left[u\frac{\partial T}{\partial x}+v\frac{\partial T}{\partial y}+\Delta T(\cancelto{0}{\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}})\right]=\frac{k}{C_p}\left[\cancelto{0}{\frac{\partial^2 T}{\partial x^2}}+\frac{\partial^2 T}{\partial y^2}\right]$$

$$\rho\left[u\frac{\partial T}{\partial x}+v\frac{\partial T}{\partial y}\right]=\frac{k}{C_p}\frac{\partial^2 T}{\partial y^2} \qquad \text{(Ec 1.21)}$$

1.5.2 Flujo fuera de la zona de la capa límite

El flujo fuera de la zona de la capa límite se analiza como un flujo no viscoso por lo que, solo se tiene en cuenta la variación de la velocidad debido al gradiente de la presión. La ecuación de la cantidad de movimiento en x para el flujo fuera de la capa límite sobre una placa plana es:

$$\rho u\frac{\partial u}{\partial x}=-\frac{dp}{dx} \qquad \text{(Ec 1.22)}$$

La (Ec 1.22) es particularmente útil para determinar el gradiente de presión para un perfil externo de velocidad dado.

1.5.3 Solución de Blasius de las ecuaciones de la capa límite.

Las ecuaciones de continuidad (Ec 1.19) , cantidad de movimiento (Ec 1.20) y conservación de la energía (Ec 1.21) del flujo en la capa límite han sido resueltas analíticamente mediante el uso de variables de semejanza.

La primera solución a la ecuación de (Ec 1.19) y (Ec 1.20) fue desarrollada por H. Blasius en 1908 para el flujo sobre una superficie plana. Blasius usó la variable de semejanza η (Ec 1.24a) junto a la función corriente ψ (Ec 1.23) y la variable dependiente $f(\eta)$ (Ec 1.24b).

$$u=\frac{\partial\psi}{\partial y} \qquad v=-\frac{\partial\psi}{\partial x} \qquad \text{(Ec 1.23)}$$

$$\eta=y\sqrt{\frac{U}{\nu\,x}} \qquad \text{(Ec 1.24a)}$$

$$f(\eta) = \frac{\psi}{U\sqrt{\frac{\nu x}{U}}} \tag{Ec 1.24b}$$

Una vez se ha realizado una serie de operaciones se transforma las dos ecuaciones parciales en una ecuación diferencial no lineal ordinaria (Ec 1.25) que puede ser resuelta por métodos numéricos. Algunos de los valores de η, f, f', f''' que satisfacen (Ec 1.25) se muestran en la tabla 1 tomada de Schlichting[6].

$$2\frac{d^3 f}{d\eta^3} + f\,\frac{d^2 f}{d\eta^2} = 0 \tag{Ec 1.25}$$

Tabla 1: Valores de η, f, f' yf'' para la ecuación de Blasius.

η	0	0.4	1.0	1.4	2.0	2.4	3.0	3.4
f	0	0.02656	0.16557	0.32298	0.65003	0.92230	1.39682	1.74696
$\frac{df}{d\eta}$	0	0.13277	0.32979	0.45627	0.62977	0.72899	0.846	0.90177
$\frac{d^2f}{d\eta^2}$	0.33206	0.33147	0.32301	0.30787	0.26675	0.22809	0.16136	0.11788
η	4.4	4.6	4.8	5.0	6.0	7.6	8.0	∞
f	2.69238	2.88826	3.08534	3.28329	4.27964	5.87924	6.27923	∞
$\frac{df}{d\eta}$	0.97587	0.98269	0.98779	0.99155	0.99898	0.99999	1.00000	1.0
$\frac{d^2f}{d\eta^2}$	0.03897	0.02948	0.02187	0.01591	0.00240	0.00004	0.00001	0

En (Ec 1.26a) se muestra el grosor de la capa límite hidráulica, en (Ec 1.26b) el coeficiente local de fricción. Adicionalmente, para una superficie isotérmica y $10 > Pr > 0.6$ se muestran el valor del grosor de la capa límite térmica[7] (Ec 1.26c), el flujo de calor (Ec 1.26d) y el coeficiente adimensional de transferencia de calor (Ec 1.26e).

$$\delta_h = \frac{5.0\,x}{\sqrt{Re_x}} \tag{Ec 1.26a}$$

$$C_{f,x} = 0.664\,Re_x^{-1/2} \tag{Ec 1.26b}$$

$$\delta_t = \frac{5.0x}{Pr^{1/3}\sqrt{Re_x}} \tag{Ec 1.26c}$$

$$q_s = 0.332\,Pr^{1/3}\,k\,\sqrt{\frac{V}{\nu\,x}}(T_s - T_\infty) \tag{Ec 1.26d}$$

$$Nu_{=}0.332\,Pr^{1/3}\,Re_x^{1/2} \tag{Ec 1.26e}$$

[6]SCHLICHTING, HERMANN. Teoría de la capa límite, 5 ed. Bilbao: Urmo, 1972. p.141.

[7]La solución de la ecuación diferencial para la capa límite térmica fue realizada por primera vez por Pohlhausen en 1921.

El coeficiente del grosor de la capa límite hidráulica es aproximado por Schlichting [26], Schetz [25], Mills [21] y White [30] a 5.0 pero se pueden encontrar otros valores en la literatura, por ejemplo, Cengel [7] utiliza el valor de 4.91.

Adicional a (Ec 1.26) se presentan otras relaciones en (Ec 1.27) que serán utilizadas posteriormente.

$$u = U\,\frac{df}{d\eta} \tag{Ec 1.27a}$$

$$v = \frac{1}{2}\frac{U}{\sqrt{Re_x}}(\eta\frac{df}{d\eta} - f) \tag{Ec 1.27b}$$

$$v_{(x,\infty)} = \frac{0.8604\,U}{\sqrt{Re_x}} \tag{Ec 1.27c}$$

$$\tau_w = 0.332\,\frac{\rho\,U^2}{\sqrt{Re_x}} \qquad \Rightarrow \qquad \frac{\mu}{\rho}\frac{\sqrt{Re_x}}{U^2}(\frac{\partial u}{\partial y})|_{y=0} \;=\; 0.332 \;=\; cte \tag{Ec 1.27d}$$

1.5.4 Otros modelos matemáticos del flujo en la capa límite

La variable de semejanza presentada por Blasius asume que la velocidad del fluido es constante más allá de la capa límite, es decir, no se produce una variación con respecto a la coordenada x, por esta razón, se presentan otras variables de semejanza y transformaciones en las coordenadas[8] para permitir un análisis general. Sin duda alguna, la geometría del problema afecta la solución de la ecuación pero si se emplean métodos numéricos para solucionar la ecuación diferencial, se pueden utilizar las variables de semejanza para una gran cantidad de geometrías modificando las condiciones de frontera.

En 1931 Falkner-Skan definieron una ecuación (Ec 1.28d) un poco más flexible que tiene en cuenta la variación de la velocidad u_∞ con respecto a la coordenada x. En este método se usa una definición similar para la variable adimensional η (Ec 1.28a) y se define una función corriente $\psi(x,y)$ adimensional (Ec 1.28b). C es una constante al igual que m con las cuales se define la velocidad del fluido (Ec 1.28c) en la región lejos de la superficie. La (Ec 1.28d) es una ecuación diferencial ordinaria no lineal que puede ser resuelta por métodos númericos teniendo en cuenta las condiciones de frontera.

$$\eta = y\,\sqrt{\frac{u_e}{\nu\,x}}\sqrt{\frac{m+1}{2}} \tag{Ec 1.28a}$$

$$\psi(x,y) \;=\; \sqrt{u_e\,\nu\,x}\,f(x,\eta) \tag{Ec 1.28b}$$

$$\begin{gathered} u_e = C\,x^m \\ -0.0904 \leq m \leq \infty \end{gathered} \tag{Ec 1.28c}$$

[8]Estos métodos se presentan de manera informativa pues la utilización de ellos está fuera del alcance del presente proyecto.

$$f''' + \frac{m+1}{2} f\, f'' + m[1 - (f')^2] = 0 \qquad \text{(Ec 1.28d)}$$

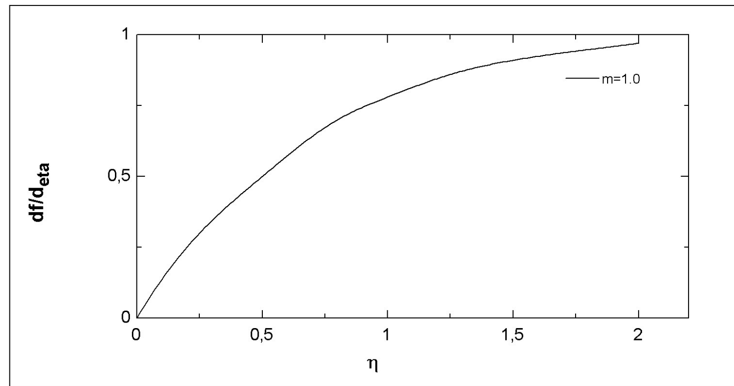
$$\beta = \frac{2\,m}{m+1} \qquad \text{(Ec 1.28e)}$$

Falkner-Skan definieron el angulo β bajo el cual se modela el flujo del fluido sobre una cuña. Algunos de los valores más comunes son:

1. $\beta = 0 \quad m = 0$ placa plana sin gradiente de presión.
2. $\beta = 1 \quad m = 1$ punto de estancamiento.
3. $\beta = -0.199 \quad m = -0.091$ inicio del desprendimiento de la capa límite.

En la figura 4 se muestra la distribución de la velocidad para m=1.

Figura 4: Distribución de la velocidad de acuerdo al modelo de Falkner-Skan.



En 1946 Crocco propuso que se tomara τ y h como variables dependientes de x y u a partir de las cuales se llega a la (Ec 1.29).

$$y(x,u) = \int_0^u \frac{\mu(h)\, du'}{\tau(x,u)} \qquad \text{(Ec 1.29)}$$

En 1956 se presentó la transformación Levy-Lees para el flujo compresible. En esta transformación se cambió el sistema de coordenadas (x,y) a $(\bar{s},\bar{\eta})$ (Ec 1.30).

$$\bar{s} \equiv \int_0^x \rho_e\, U_e\, \mu_e\, r_0^{2j}\, dx' \qquad \text{(Ec 1.30a)}$$

$$\bar{\eta} \equiv \frac{\rho_e\, U_e\, r_0^j}{\sqrt{2\,\bar{s}}} \int_0^y \frac{\rho}{\rho_e} dy' \qquad \text{(Ec 1.30b)}$$

En el año 2011 S. J. Karabelas [17] definió una nueva variable de semejanza (Ec 1.31a) que puede ser aplicada al flujo con un perfil de velocidad externo potencial o exponencial (Ec 1.31b).

$$\eta = \ln\left(\frac{u_e(x)\ y^2}{\nu\ x^{\Pi}}\right)^n \quad \text{(Ec 1.31a)}$$

$$\Pi = e^{-\frac{u_e(x) - C\ x^m}{u_e(x) - C\ e^{m\ x}}} \quad \text{(Ec 1.31b)}$$

$$\psi(x,y) = f(\eta)\ g(x,y) \quad \text{(Ec 1.31c)}$$

2. MÉTODO DE LOS VOLÚMENES FINITOS (FVM)

La mecánica de fluidos computacional (en inglés, Computational Fluids Dynamics) es una rama de la mecánica de fluidos que permite resolver problemas con geometrías complejas mediante el uso del computador. El término CFD fue utilizado en los años 70 como una combinación de fundamentos físicos, métodos numéricos y computación para simular flujos. En los años 80 se resolvieron las ecuaciones de Euler en CFD en dos y tres dimensiones. Además, en esta década se simuló el flujo viscoso de las ecuaciones de Navier-Stokes. La mecánica de fluidos computacional se ha empleado en aeronáutica, automóviles, turbomaquinaria, cámaras de combustión, metereología, oceanografía, diseño de edificios, medicina, entre otros; la CFD es una ayuda en el diseño y optimización de equipos.

Una parte necesaria para la CFD, los métodos numéricos, fueron desarrollados a partir del siglo XVIII pero su uso en la CFD sólo fue posible décadas después. Otra parte necesaria, el mallado, ha tenido grandes avances entre los que se destaca el mallado no estructurado.

Existen varios métodos en la CFD, los más comúnes son el método de volúmenes finitos, el método de diferencias finitas, FDM, y el método de elementos finitos, FEM. El **método de volúmenes finitos** (FVM) fue desarrollado en 1980 por Imperial College para resolver problemas de dinámica de fluidos. En la simulación del flujo turbulento se han desarrollado los métodos DNS (Direct Numerical Simulation) y LES (Large Eddy Simulation) y se sigue investigando para lograr una mayor aproximación a la realidad y ser aplicado en la ingeniería cotidiana. En el presente trabajo se resuelve el problema del flujo en la capa límite mediante el método de los volúmenes finitos, por esta razón, se explicará las decisiones más importantes a tomar para el método.

2.1 VENTAJAS Y DESVENTAJAS DEL FVM

Aunque los métodos FVM, FDM y FEM tienen fundamentos diferentes, en algunos casos es posible llegar a una misma solución para todos los métodos. El FVM usa las ecuaciones integrales de la dinámica de fluidos en vez de las ecuaciones diferenciales parciales del FDM. Esta característica le permite al FVM ser aplicado a cualquier tipo de malla logrando una mayor aproximación a la geometría.

En el FVM es fácil recordar que las ecuaciones utilizadas cumplen los principios de conservación ya que la discretización se realiza directamente de estos principios, los términos evaluados en las fronteras son los flujos de entrada de la propiedad evaluada y los términos evaluados en el volumen representan la generación en el volumen de control.

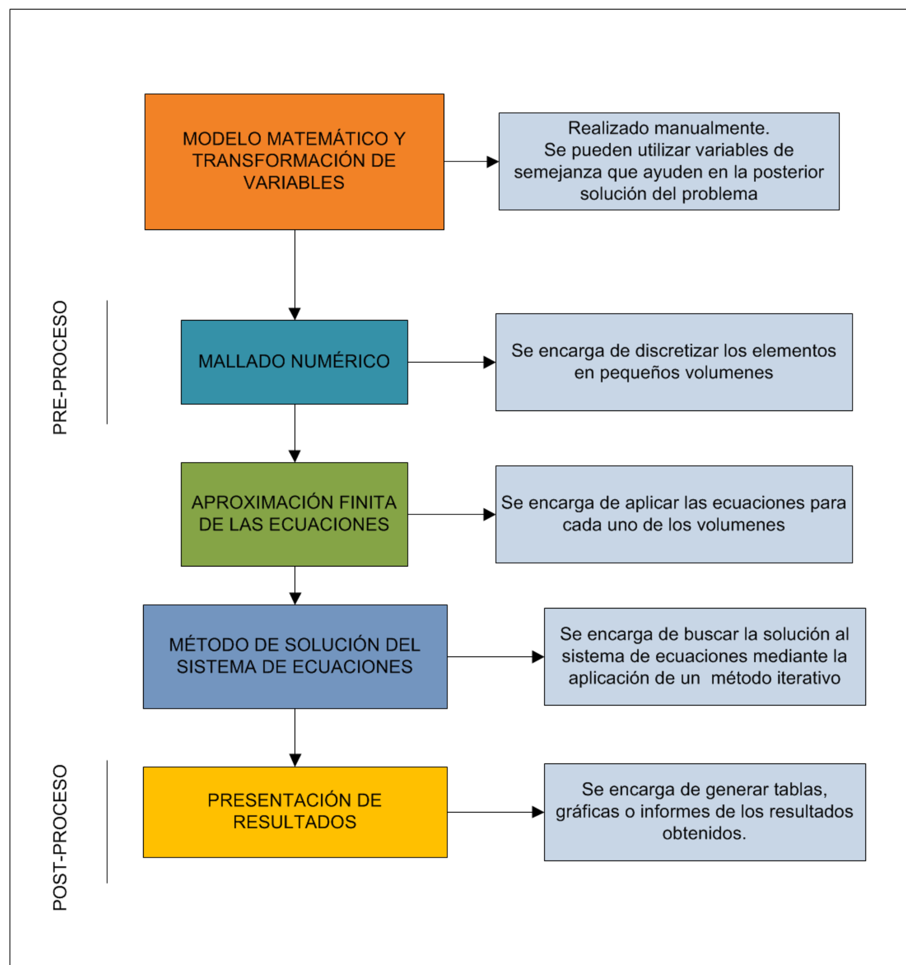
El FVM puede no producir resultados correctos cuando se trabaja con flujo turbulento, en estos casos puede resultar más conveniente usar otros métodos como DNS o LES. En el FVM comúnmente se modifica la viscosidad para incluir el efecto de la turbulencia y por lo tanto, no se pueden ver los remolinos generados. Otro problema encontrado en el FVM de primer orden es la falsa difusión que se presenta en la ecuación de difusión-convección cuando el flujo no está alineado con la malla.

2.2 PASOS GENERALES DEL FVM

En la figura 5 se muestran los pasos para resolver un problema en el FVM, se explicarán cada uno de los pasos a continuación.

Primero se debe definir el modelo matemático y los principios de conservación que se van

Figura 5: Pasos generales para resolver un problema en el FVM.



a aplicar. Las ecuaciones obtenidas de los principios de conservación se deben simplificar en algunos casos, por ejemplo, usando variables de semejanza para transformar el sistema de coordenadas o usando uno de los siguientes teoremas, como posteriormente se mostrará en el presente trabajo:

1. **Teorema de stokes (Ec 2.1):** Es una relación entre una integral de línea de un campo vectorial y una integral de superficie para las siguientes condiciones:

- F es un campo vectorial que tiene derivadas parciales continuas en una región del espacio $\mathbb{R}^3$ en donde S esta definida.
- S es una superficie orientada, diferenciable por trozos y limitada por C.
- C es una curva cerrada simple, diferenciable por trozos con orientación positiva que limita a la superficie S.

$$\int_{C+} \vec{F} \cdot d\vec{r} = \iint_S \vec{\nabla} \times \vec{F} d\vec{S} \tag{Ec 2.1}$$

2. **Teorema de green (Ec 2.2):** en el caso de 2D (2 dimensiones: x e y) es conveniente transformar una integral de área en una integral de línea; al realizar este proceso, se evalúa el flujo de una propiedad en la superficie del volumen de control evitando la dificultad de integrales dobles. La correcta aplicación del teorema implica que la integral de línea debe realizarse en sentido antihorario y se deben cumplir las siguientes condiciones:

- D es una región que pertenece a un plano y está limitada por C.
- C es una curva cerrada, positivamente orientada y diferenciable.
- $L\ y\ M$ son funciones que tienen derivadas parciales continuas en una región abierta que contiene a D.

$$\oint_C Pdx + Qdy = \iint_D \left(\frac{\partial Q}{\partial x} - \frac{\partial P}{\partial y} \right) dA \tag{Ec 2.2}$$

Por otro lado, definiendo $F = P\,\widehat{i} + Q\,\widehat{j}$ el teorema de green se puede expresar de forma vectorial como:

$$\oint_C \vec{F} \cdot \widehat{n}\, dS = \iint_D \vec{\nabla} \cdot \vec{F}\ \ dA \tag{Ec 2.3}$$

3. **Teorema de la divergencia (Ec 2.4):** Son necesarias las siguientes condiciones para definir el teorema de la divergencia:

- E es una región sólida simple.
- S es la superficie que limita E con una orientación positiva.
- F es un campo vectorial con derivadas parciales continuas en una región abierta que contiene E.

$$\iint_S \vec{F} \cdot d\vec{S} = \iiint_E \nabla \cdot \vec{F} d\mathbb{V} \tag{Ec 2.4}$$

El siguiente paso a realizar es el mallado, existen diferentes tipos de mallado que permiten adaptarse a una gran variedad de geometrías, algunos de estos se presentarán en la sección 2.3. Posteriormente, se discretizan (ver la sección 2.4) las ecuaciones obtenidas aproximando las integrales y las derivadas, este proceso produce un conjunto de coeficientes que definen las ecuaciones a resolver. Las ecuaciones se resuelven conmúnmente por métodos iterativos como se explica en la sección 2.6. Finalmente, se realiza un post-procesamiento de los resultados (ver sección 2.7) que incluye la graficación, generación de tablas, informes, etc.

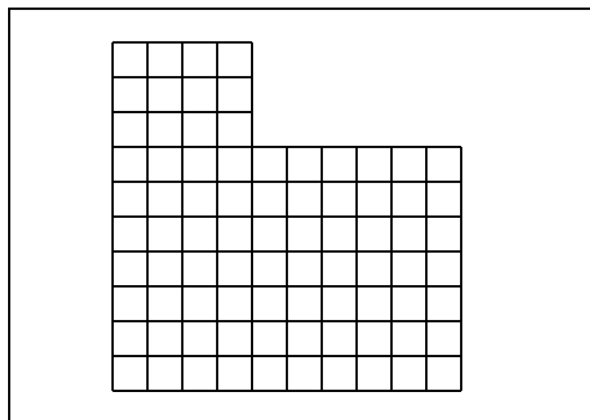
2.3 MALLA

El MVF se puede aplicar a varios tipos de malla que pueden adaptarse mejor a ciertas geometrías o simplificar el proceso de solución. En esta sección se presentan algunos tipos de mallas explicando su característica principal en dos dimensiones.

2.3.1 Malla estructurada

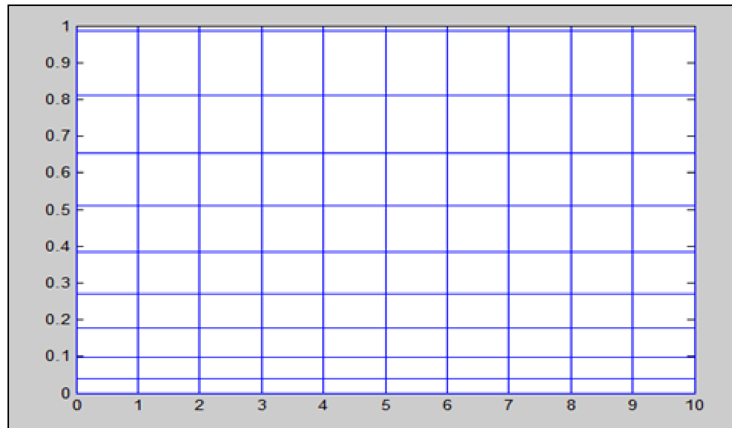
Este tipo de malla está compuesta por dos conjuntos de líneas en los cuales las líneas que pertenecen a cada conjunto no se interceptan. En la figura 6 se presenta un ejemplo de una malla estructurada.

Figura 6: Malla estructurada ortogonal de tamaño de celda fijo.



Malla ortogonal: la malla ortogonal es un tipo especial de malla estructurada en la que los volúmenes de control están formados por lados rectos y perpendiculares. Este tipo de malla puede tener volúmenes de igual tamaño, figura 6, o volúmenes de tamaño variable, figura 7.

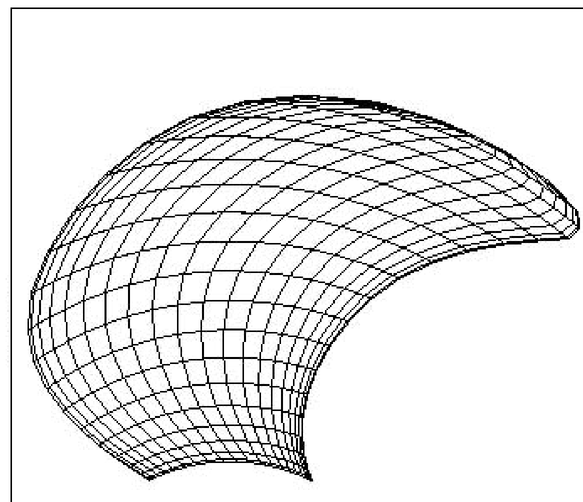
Figura 7: Malla ortogonal de tamaño de celda variable.



Aunque las mallas de tamaño de celda variable pueden generarse “aleatoriamente” es común usar un factor de proporción constante entre el tamaño de la celda actual y la celda siguiente.

Malla no ortogonal: los dos conjuntos de líneas que conforman una malla estructurada no tienen que ser necesariamente líneas rectas que formen ángulos de 90º con las líneas del otro conjunto. En la figura 8 se presenta un mallado estructurado no ortogonal para una región con lados curvos.

Figura 8: Malla estructurada no ortogonal.

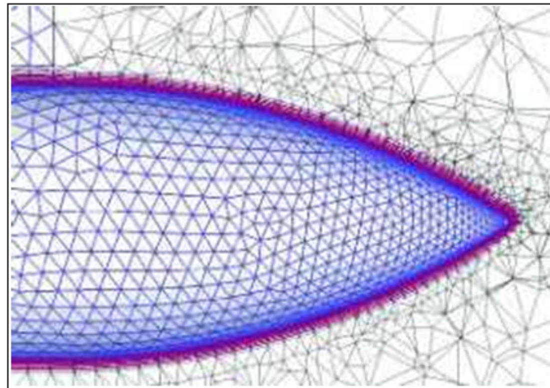


Fuente: http://www.scielo.cl/scielo.php?pid=S0718-07642006000300016&script=sci_arttext

2.3.2 Malla no estructurada

En este tipo de malla se usan volúmenes de control con la forma de polígonos o figuras en general para lograr la adaptación de la malla a cualquier geometría sin la necesidad de una transformación del sistema de coordenadas. En la figura 9 se presenta un ejemplo de este tipo de malla en la que se usan elementos triangulares y cuadriláteros.

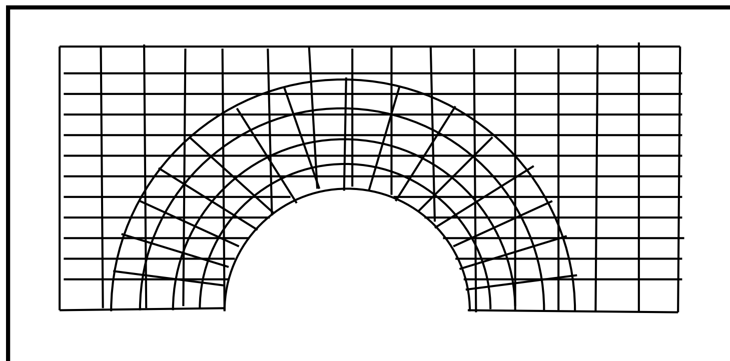
Figura 9: Malla no estructurada.



Fuente: http://machinedesign.com/article/cfd-helps-cure-separation-anxiety-0414

Doble mallado de bloques estructurados (composite grid or chimera grid): es particularmente útil cuando se tienen cuerpos en movimiento inmersos en un fluido en donde se usa una malla fija y una malla que sigue el movimiento del cuerpo o, cuando se tienen geometrías complejas. Las condiciones de frontera de la segunda malla son obtenidas por interpolación de los valores obtenidos en la primera malla. Se muestra un ejemplo de este tipo de malla en la figura 10.

Figura 10: Doble mallado.

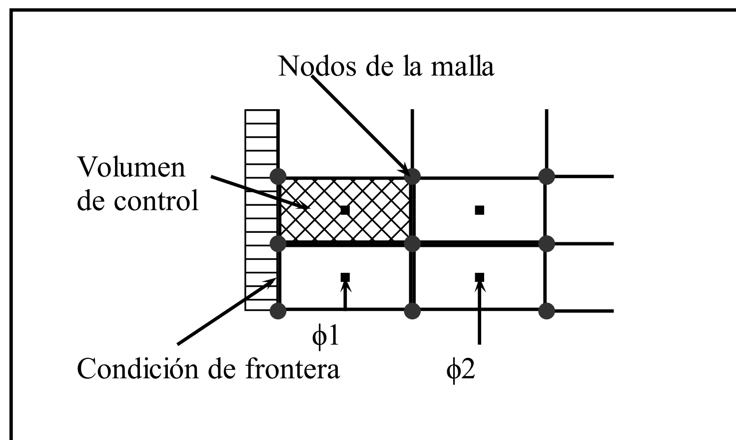


2.3.3 Técnicas de definición del volumen de control

Un volumen de control no necesariamente debe corresponder a la celda de la malla. Se presentan algunos esquemas para ubicar el volumen de control dentro de la celda:

- **Nodo en el centro del volumen de control (cell centered scheme):** en este caso, el volumen de control corresponde exactamente a una celda de la malla, la ubicación de las variables de flujo se asume en el centroide del volumen de control. Un ejemplo de este tipo se puede ver en la figura 11.

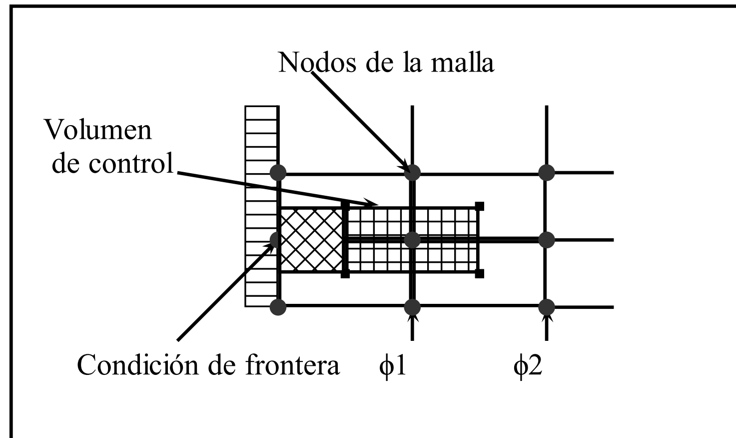
Figura 11: Nodo en el centro de la celda ó cell-centered scheme.



- **Nodo en los vértices de la malla (cell vertex scheme):** existen dos tipos, con traslape (overlaping) el cual mantiene la definición del volumen de control de la malla pero el valor de las propiedades se ubica en cada una de las esquinas del volumen de control. Por otro lado se encuentra el doble volumen de control (dual control volumes) en el cual se define un volumen de control que contiene una sección de cada una de las celdas que tienen el nodo como vértice, se muestra un ejemplo de este tipo de malla en la figura 12.

En el presente trabajo no se requiere condiciones especiales de mallado que ameriten una mayor complejidad en la definición del método por lo que se ubicará el nodo en el centro de la celda . Adicionalmente, al usar una única malla cuyas celdas coinciden con los volúmenes de control se disminuye el espacio requerido para almacenar la información de otras mallas. En el capítulo 6 se mostrará la corrección de la velocidad por la presión cuando se asume el esquema nodo en el centro de la celda para la presión y la velocidad, también llamado mallas colocadas.

Figura 12: Nodo en el vértice de la malla ó cell-vertex scheme.



2.4 APROXIMACIÓN DE LAS VARIABLES

Con el fin de mostrar la forma de aproximar el comportamiento de una variable dentro de un volumen de control, todas las ecuaciones mostradas en esta sección serán en base a una malla ortogonal de tamaño de celda fijo pero fácilmente pueden ser aplicados a otros tipos de malla. Adicionalmente, se mostrará cómo se aproxima el valor de una variable en la superficie de control.

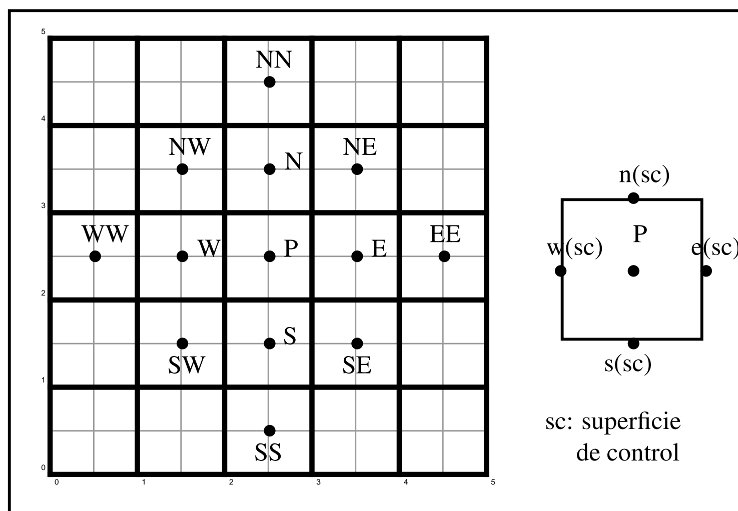
La nomenclatura usada a continuación incluye los subíndices P, E, N, W y S como suele aparecer en la literatura, esta nomenclatura implica que el análisis actual se hace a un volumen de control P, el cual limita a la derecha por el volumen E o `este` , a la izquierda por el volumen W o `west` , al norte por el volumen N y al sur por el volumen S. Si el subíndice aparece en minúscula significa que la propiedad se evalúa en la frontera[1] o en otro punto diferente al centro del volumen de control. A veces se usan dos subíndices, por ejemplo, ϕ_{NN}, este caso significa que se toma el valor de ϕ del segundo volumen de control al norte del volumen actual. En la figura 13 se muestra el uso de los subíndices para un volumen de control, los subíndices necesarios permiten definir el tamaño de la molécula computacional del esquema usado.

2.4.1 Interpolación aguas-arriba (UDS)

Este método considera que el valor de una propiedad dentro del volumen de control es constante e igual a ϕ_P. Para evaluar las propiedades en la frontera se tiene en cuenta la dirección del flujo; en la (Ec 2.5) se muestra la forma de evaluar una propiedad en la frontera para una

[1] En el presente documento se usa adicionalmente *sc* para indicar que la propiedad se evalúa en la superficie de control

Figura 13: Molécula general.



malla ortogonal, gráficamente se presenta la información en la Figura 14:

$$\phi_{e(sc)} = \begin{cases} \phi_E & ssi \quad (\boldsymbol{v}\cdot\boldsymbol{n})_{e(sc)} < 0 \\ \phi_P & ssi \quad (\boldsymbol{v}\cdot\boldsymbol{n})_{e(sc)} > 0 \end{cases} \qquad \phi_{n(sc)} = \begin{cases} \phi_N & ssi \quad (\boldsymbol{v}\cdot\boldsymbol{n})_{n(sc)} < 0 \\ \phi_P & ssi \quad (\boldsymbol{v}\cdot\boldsymbol{n})_{n(sc)} > 0 \end{cases}$$

$$\phi_{w(sc)} = \begin{cases} \phi_W & ssi \quad (\boldsymbol{v}\cdot\boldsymbol{n})_{w(sc)} < 0 \\ \phi_P & ssi \quad (\boldsymbol{v}\cdot\boldsymbol{n})_{w(sc)} > 0 \end{cases} \qquad \phi_{s(sc)} = \begin{cases} \phi_S & ssi \quad (\boldsymbol{v}\cdot\boldsymbol{n})_{s(sc)} < 0 \\ \phi_P & ssi \quad (\boldsymbol{v}\cdot\boldsymbol{n})_{s(sc)} > 0 \end{cases} \quad \text{(Ec 2.5)}$$

La definición utilizada en (Ec 2.5) puede tratarse de una forma más general como en (Ec 2.6) que permite su aplicación a cualquier tipo de malla en donde $\boldsymbol{v}$ es el vector de la velocidad del fluido y $\boldsymbol{n}$ es un vector normal a la superficie que apunta hacia afuera del volumen de control.

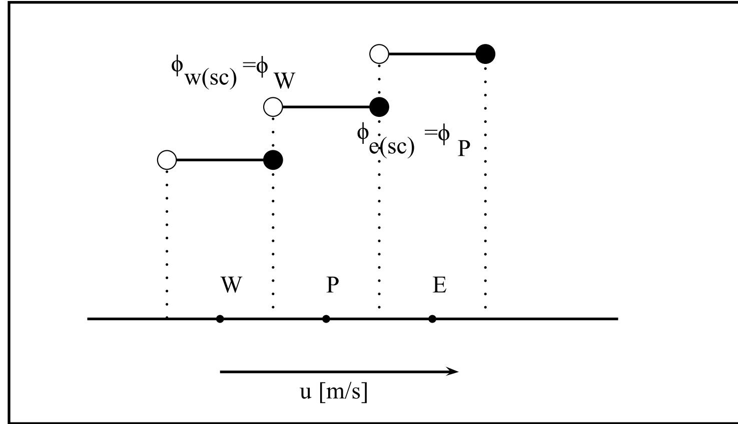
$$\phi_{sc} = \begin{cases} \phi_p & ssi \quad \vec{v}\cdot\vec{n} \geq 0 \\ \phi_{vecino} & ssi \quad \vec{v}\cdot\vec{n} < 0 \end{cases} \quad \text{(Ec 2.6)}$$

El uso de este procedimiento conlleva a un error proporcional a la longitud del lado Δx_i del volumen de control, algunas veces se usa $\boldsymbol{O}(\Delta x)$ para designar que este método de aproximación es de primer orden.

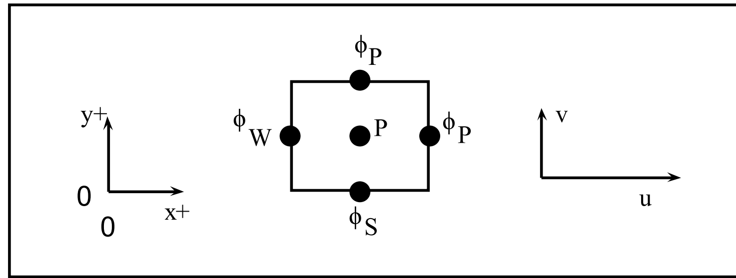
2.4.2 Interpolación lineal (CDS)

El método recibe también el nombre de CDS (siglas en inglés de *central difference scheme*) debido a su similitud con el método de diferencias centradas del FDM. Se asume una distribución lineal de una propiedad entre dos nodos adyacentes por lo que el error producido por su uso es de segundo orden, $\boldsymbol{O}(\Delta x^2)$. La desventaja de este método y otros métodos de

Figura 14: Interpolación aguas-arriba ó upwind interpolation.



(a) Representación en una dimensión.



(b) Representación en 2D junto con la dirección del perfil de velocidad.

orden superior es que no son incondicionalmente estables por lo que se limita el tamaño de los volúmenes de control o el avance en el tiempo.

En el caso de una malla ortogonal de dos dimensiones se tienen 4 volúmenes de control colindando con el VC actual por lo que la definición de una propiedad en la superficie se hace a partir de la ecuación (Ec 2.7) en donde x_I y y_I representan la posición del centro del volumen de control I en el eje x y el eje y.

$$\phi_{e(sc)} = \phi_E \frac{x_{e(sc)} - x_P}{x_E - x_P} + \phi_P \frac{x_E - x_{e(sc)}}{x_E - x_P} \qquad \phi_{n(sc)} = \phi_N \frac{y_{n(sc)} - y_P}{y_N - y_P} + \phi_P \frac{y_N - y_{n(sc)}}{y_N - y_P}$$

$$\phi_{w(sc)} = \phi_W \frac{x_{w(sc)} - x_P}{x_W - x_P} + \phi_P \frac{x_W - x_{w(sc)}}{x_W - x_P} \qquad \phi_{s(sc)} = \phi_S \frac{y_{s(sc)} - y_P}{y_S - y_P} + \phi_P \frac{y_S - y_{s(sc)}}{y_S - y_P} \qquad \text{(Ec 2.7)}$$

Si la malla posee un tamaño de celda fijo la (Ec 2.7) se puede simplificar en (Ec 2.8)

$$\phi_{e(sc)} = \frac{\phi_E + \phi_P}{2} \qquad \phi_{n(sc)} = \frac{\phi_N + \phi_P}{2}$$

$$\phi_{w(sc)} = \frac{\phi_W + \phi_P}{2} \qquad \phi_{s(sc)} = \frac{\phi_S + \phi_P}{2} \qquad \text{(Ec 2.8)}$$

2.4.3 Interpolación cuadrática (QUICK)

Por sus siglas en inglés, *quadratic upwind interpolation for convective kinematics*, el método utiliza tres puntos para definir el valor de una propiedad en la frontera. Para una malla ortogonal la definición del flujo en las cuatro fronteras se muestra en (Ec 2.9). Este método tiene un error de $\boldsymbol{O}(\Delta x^4)$.

$$
\begin{aligned}
\phi_{e(sc)} &= \phi_P + \frac{(x_{e(sc)}-x_P)(x_{e(sc)}-x_W)}{(x_E-x_P)(x_E-x_W)}(\phi_E-\phi_P) + \frac{(x_{e(sc)}-x_P)(x_E-x_{e(sc)})}{(x_P-x_W)(x_E-x_W)}(\phi_P-\phi_W)\\
\phi_{n(sc)} &= \phi_P + \frac{(y_{n(sc)}-y_P)(y_{n(sc)}-y_S)}{(y_N-y_P)(y_N-y_S)}(\phi_N-\phi_P) + \frac{(y_{n(sc)}-y_P)(y_N-y_{n(sc)})}{(y_P-y_S)(y_N-y_S)}(\phi_P-\phi_S)\\
\phi_{w(sc)} &= \phi_w + \frac{(x_{w(sc)}-x_w)(x_{w(sc)}-x_{ww})}{(x_P-x_w)(x_P-x_{ww})}(\phi_P-\phi_w) + \frac{(x_{w(sc)}-x_w)(x_P-x_{w(sc)})}{(x_w-x_{ww})(x_P-x_{ww})}(\phi_w-\phi_{ww})\\
\phi_{s(sc)} &= \phi_S + \frac{(y_{s(sc)}-y_S)(y_{s(sc)}-y_{SS})}{(y_P-y_S)(y_P-y_{SS})}(\phi_P-\phi_S) + \frac{(y_{s(sc)}-y_S)(y_P-y_{s(sc)})}{(y_S-y_{SS})(y_P-y_{SS})}(\phi_S-\phi_{SS})
\end{aligned}
\qquad \text{(Ec 2.9)}
$$

2.5 CONDICIONES DE FRONTERA

Existen tres tipos de condiciones de frontera que fueron definidas por Dirichlet, Neumann y Cauchy/Robin

- **Condición de frontera de Dirichlet:** se define el valor de una propiedad en un punto determinado, por ejemplo, el valor de la velocidad, de la temperatura, de la función corriente, etc. Se expresa como (Ec 2.10).

$$\phi = \gamma \qquad \text{(Ec 2.10)}$$

- **Condición de frontera de Neumann:** los ejemplos más comunes son el flujo de calor, derivada parciales (de la función corriente, la velocidad, ...) Su definición en forma matemática se muestra en (Ec 2.11).

$$\frac{\partial \phi}{\partial x_i} = \gamma \qquad \text{(Ec 2.11)}$$

- **Condición Cauchy/Robin:** este tipo de condición de frontera no es muy usada pero se encuentra en la literatura, es una combinación de los dos casos anteriores, se puede definir en una expresión como (Ec 2.12).

$$\alpha\phi + \beta\frac{\partial \phi}{\partial x_i} = \gamma \text{ para } \alpha \neq 0 \;\; y \;\; \beta \neq 0 \qquad \text{(Ec 2.12)}$$

2.6 SOLUCIÓN DE LAS ECUACIONES

Se deben tener en cuenta varios factores al momento de escoger un método de solución de las ecuaciones, se destacan los siguientes factores:

- Linealidad de las ecuaciones.
- Tiempo requerido para resolver el sistema, normalmente expresado en términos del número de incógnitas en el problema.
- Para los métodos iterativos se debe tener en cuenta:
  - Convergencia de la solución.
  - Número de iteraciones necesarias para resolver el sistema.
  - Otros factores como error en la solución y propagación, criterios de parada, etc.

Se muestra de manera informativa algunos métodos comúnmente usados, los cuales se encuentran implementados en las librerías de Ph.D. David Fuentes, por esta razón, no es necesario conocer de manera precisa el proceso de solución, en cambio, se debe tener en cuenta cuando es posible usar un método e incluso, cuál método es mejor para un problema específico.

2.6.1 Ecuaciones lineales

Un sistema de ecuaciones lineales está compuesto de n ecuaciones al estilo de $a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{m\,m-1}x_{m-1} + a_{mm}x_m + a_{m\,m+1}x_{m+1} + \cdots + a_{m\,n-1}x_{n-1} + a_n x_n = b_m$ en donde m es el subíndice del número de la ecuación. De manera compacta, un sistema de ecuaciones lineales se representa como (Ec 2.13) en donde $\boldsymbol{A}$ es la matriz de coeficientes, $\boldsymbol{b}$ es el vector de términos independientes y $\boldsymbol{x}$ es el vector solución.

$$\boldsymbol{A}\,\boldsymbol{x} = \boldsymbol{b} \tag{Ec 2.13}$$

Los sistemas de ecuaciones lineales pueden ser resueltos por métodos directos y métodos indirectos. Los métodos directos no son eficientes respecto al tiempo requerido para la solución aunque se pueden obtener en algunos casos, resultados más exactos. Por otro lado, los métodos indirectos no siempre convergen. En la tabla 2 se nombran los métodos disponibles en las librerías para resolver los sistemas de ecuaciones lineales y posteriormente se comentan algunos de los métodos más comúnmente usados.

Cálculo de la matriz inversa: es un método directo que consiste en encontrar la matriz inversa $\boldsymbol{A}^{-1}$ y multiplicarla por el vector de términos independientes, este proceso requiere $\boldsymbol{O}(n^3)$ operaciones.

$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b} \tag{Ec 2.14}$$

Tabla 2: Métodos existentes en las librerías pdelib para resolver sistemas de ecuaciones lineales.

Métodos directos		
Cálculo de la matriz inversa	Factorización LU	Factorización Cholesky
Métodos iterativos		
Jacobi	Relajaciones sucesivas (SOR)	Relajaciones sucesivas para sistemas simétricos (SSOR)
Método del gradiente conjugado (GradConj)	Método generalizado del mínimo residuo para sistemas no simétricos (GMres)	Gradiente conjugado cuadrado para sistemas no simétricos (GCcuad)
Minimización ortogonal (Orthomin)	Gradiente bi-conjugado estabilizado para sistemas no simétricos (GBiCEst)	Método iterativo para sistemas dispersos simétricos grandes (Symmlq)
Transpuesta-libre de residuo cuasi-mínimo para sistemas lineales no simétricos (TFQMR)		

Factorización LU: la matriz A se factoriza en dos matrices (Ec 2.15a), L y U , L es una matriz triangular inferior con 1 en los elementos de la diagonal y U es una matriz triangular superior, de esta forma, el sistema de ecuaciones se transforma a (Ec 2.15b). Se procede a hallar Y en (Ec 2.15c) y finalmente se halla x en (Ec 2.15d). Este método directo requiere un número de operaciones del orden de $\boldsymbol{O}(2n^3/3)$ operaciones y es particularmente útil cuando se desea resolver varios sistema de ecuaciones con la misma matriz de coeficientes y diferentes vectores de términos independientes ya que el proceso de factorización es independiente del vector de términos independientes.

$$\boldsymbol{A} = \boldsymbol{L}\,\boldsymbol{U} \tag{Ec 2.15a}$$

$$\boldsymbol{L}\,\boldsymbol{U}\,\boldsymbol{x} = \boldsymbol{b} \tag{Ec 2.15b}$$

$$\boldsymbol{Y} = \boldsymbol{U}\,\boldsymbol{x} \tag{Ec 2.15c}$$

$$\boldsymbol{U}\,\boldsymbol{x} = \boldsymbol{Y} \tag{Ec 2.15d}$$

Factorización Cholesky: es un caso especial de la factorización LU que solo se puede aplicar a matrices simétricas definidas positivas. La matriz A se descompone en dos matrices (Ec 2.16), L y L^T en donde, L es una matriz triangular inferior y L^T es la conjugada traspuesta. Este proceso requiere $\boldsymbol{O}(n^3/3)$ operaciones.

$$\boldsymbol{A} = \boldsymbol{L}\,\boldsymbol{L^T} \tag{Ec 2.16}$$

Método iterativo Jacobi: la matriz A se define como la suma de tres matrices (Ec 2.17a), una matriz diagonal D, una matriz estrictamente triangular superior L y una matriz estrictamente triangular inferior U. Al realizar este proceso el sistema a resolver es (Ec 2.17b)

del cual se puede definir la regla para el método iterativo de forma vectorial (Ec 2.17c) en donde k representa el contador de iteraciones, el valor de x_i^{k+1} para $i = 1,2,3,..,n$ se halla mediante (Ec 2.17d). El método Jacobi converge cuando A es una matriz estrictamente diagonal dominante.

$$\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{L} + \boldsymbol{U} \tag{Ec 2.17a}$$

$$\boldsymbol{D}x + (\boldsymbol{L} + \boldsymbol{U})x = b \tag{Ec 2.17b}$$

$$x^{(k+1)} = \boldsymbol{D}^{-1}[b - (\boldsymbol{L} + \boldsymbol{U})x^{(k)}] \tag{Ec 2.17c}$$

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij}\, x_j^{(k)} \right) \tag{Ec 2.17d}$$

Método de sobrerrelajaciones sucesivas (SOR): el sistema de ecuaciones se re-escribe como (Ec 2.18a) en donde ω es el factor de relajación, D es una matriz diagonal, L es una matriz estrictamente triangular inferior y U es una matriz estrictamente triangular superior. El valor de la variable x_i en la iteración $k+1$ se halla mediante (Ec 2.18b). Si el método converge, el número de iteraciones necesarias para obtener la solución depende del valor de ω siendo importante seleccionar el valor correcto para el parámetro. El número de operaciones necesario por cada iteración es $\boldsymbol{O}(2n^2)$.

$$(\boldsymbol{D} + \omega \boldsymbol{L})\, \boldsymbol{x} = \omega\, \boldsymbol{b} - [\omega\, \boldsymbol{U} + (\omega - 1)\, \boldsymbol{D}]\, \boldsymbol{x} \tag{Ec 2.18a}$$

$$x_i^{k+1} = (1-\omega)\, x_i^{(k)} + \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=i+1}^{n} [a_{ij} x_j^{(k)}] - \sum_{j=1}^{i-1} [a_{ij} x_j^{(k+1)}] \right) \tag{Ec 2.18b}$$

Método de sobrerrelajaciones sucesivas para sistemas simétricos (SSOR): es un caso especial del método SOR que solo es aplicable cuando la matriz A es simétrica. El método SSOR converge para cualquier valor de $\omega \in (0,2)$ si A es una matriz hermítica definida positiva.

Para matrices de gran tamaño, los métodos directos requieren una gran cantidad de tiempo por lo que se prefiere la solución del sistema de ecuaciones por métodos iterativos pero, para lograr una disminución significativa del tiempo de cálculo, se utilizan matrices dispersas, si se requiere disminuir aún más el tiempo, se usa el cálculo en paralelo con varios computadores.

2.6.2 Ecuaciones no lineales

Los métodos disponibles para resolver un sistema de ecuaciones no lineales son el **método de NewtonRaphson**, el **método de gradiente conjugado** y el **método de substitución sucesiva**.

Una forma opcional de resolver las ecuaciones no lineales es reordenar la ecuación para poder tratarla como una ecuación lineal que se resuelve por métodos iterativos, un ejemplo

de este tipo es el método SIMPLE para resolver el sistema de ecuaciones diferenciales parciales no lineales compuesto por las ecuaciones de Navier-Stokes, este método se explicará posteriormente.

2.7 POST-PROCESAMIENTO

El post-procesamiento incluye todas las operaciones a realizar después de haber obtenido la solución al sistema de ecuaciones, dentro de esta categoría se incluye la generación de tablas y gráficas. En el presente trabajo, este proceso se realizará en Paraview.

La generación de gráficas presenta un sinnúmero de opciones, se pueden generar gráficas en 2D y 3D, isocontornos, campos vectoriales, mostrar la malla para el análisis usado y graficar varias propiedades en una misma gráfica, entre otros.

2.8 CARACTERÍSTICAS A EVALUAR EN LA CFD

Existen algunas características cualitativas y cuantitativas que permiten decidir cuando el algoritmo implementado obtiene una solución satisfactoria o presenta deficiencias bajo ciertas circunstancias.

1. Consistencia: un método numérico es consistente si se puede obtener la ecuación original de la ecuación discretizada al disminuir el tamaño de los volúmenes de control y el avance en el tiempo, para lograr esto, se debe recordar la definición de las derivadas y las integrales. Como explica Tu en [29], es además necesario que se cumpla $\Delta t \ll \Delta x$ para lograr la consistencia de las ecuaciones, la consistencia está además relacionada con el orden n del error del proceso usado, $\boldsymbol{O}(\Delta x^n)$.

2. Estabilidad: es la capacidad de disminuir el error o de no magnificarlo en cada proceso de cálculo; para los casos en los que el problema a desarrollar tiene una solución acotada, ya sea por que se conoce la solución analítica o por datos reales de un experimento, el uso de métodos numéricos conduce a una solución estable si la solución encontrada es también acotada. Existen algunos métodos que son estables bajo ciertas condiciones como suele ser el caso para aproximaciones de alto orden, por otro lado, existen métodos que son incondicionalmente estables como es el caso del esquema UDS para la ecuación de convección-difusión en estado estable.

   Se han desarrollado algunos métodos para verificar la condición de estabilidad, los dos métodos más usados para verificar la estabilidad en problemas lineales son el método de Von Neumann y el método de la matriz. Cabe destacar para el tema del presente trabajo, el uso del método de la analogía de circuitos de Karplus por parte de Schetz [25] para verificar la estabilidad de un método explícito para resolver las ecuaciones de la capa límite en estado estable con propiedades constantes.

El uso de solucionadores iterativos crea una nueva característica a evaluar, la estabilidad del método de solución, aunque el problema a resolver sea incondicionalmente estable, es posible no encontrar una solución si se escoge un método incorrecto para resolver el sistema de ecuaciones.

3. Convergencia: en general para que un método numérico sea definido como convergente debe ser consistente y estable. La convergencia se define como la capacidad de acercarse cada vez más a la solución exacta del problema cuando el tamaño de los volúmenes de control y el avance en el tiempo son cada vez más pequeños.

4. Precisión: la precisión determina que tan cerca está la solución encontrada a la solución real. La precisión se ve afectada por el error del modelado, el error de discretización, el error de iteración y el error de truncamiento; algunos de estos factores son más importantes que otros y en el análisis de los resultados se debe verificar si la solución encontrada satisface los requerimientos o es necesario hacer un análisis más preciso. Existen casos en los cuales un análisis más complejo no garantiza un aumento significativo en la precisión, como ejemplo, el aumento del número de volúmenes de control puede implicar un aumento relevante en el tiempo de cálculo pero la mejora en la precisión no es significativa.

5. Conservación: la conservación en el método utilizado se garantiza si el balance de las entradas, las salidas y la generación se satisface para cada volumen de control y para la región analizada.

6. Solución acotada: esta propiedad está relacionada con la estabilidad, una solución no acotada puede ser fácilmente descartada si la solución es inestable y en general se recomienda analizar los resultados para verificar que la solución encontrada es acotada. En el caso de conocer el valor teórico mínimo y máximo de una propiedad, se deben verificar los valores encontrados; los límites a la solución pueden ser dados por las condiciones de frontera o valores fisicamente imposibles como presiones absolutas negativas o temperaturas inferiores a 0 [K].

2.9 METODOLOGÍA DE TRABAJO

Antes de desarrollar el código para resolver el problema del flujo en la capa límite mediante FVM, se mostrará el proceso a seguir para tres problemas más sencillos, los cuales son:

- Ecuación de difusión.
- Ecuación de difusión y convección.
- Ecuaciones transitorias de la difusión y la convección-difusión.

Con el primer ejemplo se introduce[2] el uso de las librerías desarrolladas por Ph.D. David Fuentes y se van dando más detalles con los demás ejemplos. La razón para resolver estos problemas más sencillos es familiarizar al lector con el método de operar de cada problema para, posteriormente integrar los dos procesos para la solución de las ecuaciones de la capa límite que involucra un problema de difusión para la correción de presión y un problema de convección difusión para las ecuaciones de la cantidad de movimiento.

[2]En el presente trabajo no se pretende realizar un manual de referencia de todas las funciones, en cambio, se presenta el código necesario para desarrollar los problemas planteados.

3. FLUJO DIFUSIVO EN ESTADO ESTABLE

Es necesario desarrollar un código que permita calcular la ecuación de difusión y la ecuación de difusión-convección para resolver las ecuaciones no lineales de la capa límite. En este capítulo se presenta el proceso para resolver la ecuación de difusión y la implementación desde dos puntos de vista diferentes.

Para obtener la solución de la ecuación de difusión se debe partir de la ecuación diferencial general de la mecánica de fluidos (Ec 1.5).

$$\underbrace{\frac{\partial(\rho\,\phi)}{\partial t}}_{término\,transitorio} + \underbrace{\nabla\cdot(\rho\,\phi\,\vec{V})}_{término\,convectivo} = \underbrace{\nabla\cdot(\Gamma\,\nabla\phi)}_{término\,difusivo} + \underbrace{S}_{término\,fuente}$$

Si se elimina el término transitorio y el término convectivo de (Ec 1.5) se obtiene la ecuación diferencial de la difusión con término fuente (Ec 3.1). Las simplificaciones realizadas fueron basadas en un análisis para estado estable y $\vec{V} = \mathbf{0}$.

$$\nabla\cdot(\Gamma\,\nabla\phi) + S = 0 \tag{Ec 3.1}$$

En este capítulo se tomará $\phi = T$ ya que se tiene una mejor percepción del valor de la temperatura y su derivada (proporcional al flujo de calor) pero el procedimiento se puede realizar para una propiedad ϕ cualquiera que cumpla un principio de conservación; es por esta razón que se tomará Γ como la conductividad térmica k y S como la generación de calor por unidad de volumen q. Se vuelve a escribir (Ec 3.1) en términos de T, k y q en (Ec 3.2):

$$\nabla\cdot(k\nabla T) + q = 0 \tag{Ec 3.2}$$

La (Ec 3.2) se integra a través de un volumen de control Ω.

$$\iiint_{\Omega} \nabla\cdot(k\nabla T)\; dV + \iiint_{\Omega} q\; dV = 0 \tag{Ec 3.3}$$

Al aplicar el teorema de la divergencia (Ec 2.4) a la primera integral de volumen de (Ec 3.3) se obtiene (Ec 3.4) en donde S es la superficie que rodea al volumen de control.

$$\iint_{S} (k\nabla T)\cdot\vec{dA} + \iiint_{\Omega} q\; dV = 0 \tag{Ec 3.4}$$

Resulta conveniente representar $\vec{dA}$ como la multiplicación de un vector unitario $\hat{n}$ que apunta hacia afuera del volumen de control y el diferencial de área dA como se muestra en (Ec 3.5).

$$\iint_{S} k(\nabla T\cdot\hat{n})\; dA + \iiint_{\Omega} q\; dV = 0 \tag{Ec 3.5}$$

La (Ec 3.5) debe ser discretizada y para ello, se debe analizar de una forma diferente cada una de las dos integrales.

La solución de la **integral de volumen del término fuente** dependerá del valor de q. Si q es una función se debe aplicar métodos numéricos para hallar el valor de la integral en el volumen de control. En este capítulo se asumirá un valor constante para q de manera que se cumple la siguiente igualdad:

$$\iiint_{\Omega} q \; dV = q \; \Omega \qquad \text{(Ec 3.6)}$$

Por otro lado, la **integral del término difusivo** se aproxima a una sumatoria de n flujos de calor para cada uno de los n lados del volumen de control. Se presenta esta sumatoria en (Ec 3.7):

$$\iint_{S} k(\nabla T \cdot \hat{n}) \; dA \approx \sum_{i=1}^{n} \left[k_i \left((\nabla T)_i \cdot \widehat{n_i}\right) A_i\right] \qquad \text{(Ec 3.7)}$$

En (Ec 3.7) el término $[k_i \left((\nabla T)_i \cdot \widehat{n_i}\right) A_i]$ representa el flujo de calor a través del área A_i. Se debe discretizar dicho término, como se muestra en (Ec 3.8), donde T_p denota la temperatura en el nodo actual, T_i representa la temperatura del nodo vecino a la superficie considerada y U_i representa la conductancia entre los dos nodos en consideración.

$$[k_i \left((\nabla T)_i \cdot \widehat{n_i}\right) A_i] = U_i \, A_i \, (T_i - T_p) = \begin{pmatrix} Flujo \\ calor \end{pmatrix}_i \qquad \text{(Ec 3.8)}$$

Si la conductividad térmica es constante, la conductancia se define como (Ec 3.9):

$$U_i = \frac{k}{\delta_i} \qquad para \;\; k = constante \qquad \text{(Ec 3.9)}$$

Si se considera que la conductividad térmica es constante en cada volumen de control y se permite que varíe de un volumen a otro, la conductancia se puede definir como (Ec 3.10) en donde δ_{ei} es la distancia perpendicular desde el centro del volumen de control e a la superficie i, δ_{pi} es la distancia perpendicular del centro del volumen p a la superficie i. R representa la resistencia térmica, el cual es el inverso multiplicativo de la conductancia térmica.

$$U_i = \frac{1}{\sum R_T} = \frac{1}{R_{ei} + R_{pi}} = \frac{1}{\frac{1}{U_{ei}} + \frac{1}{U_{pi}}} = \frac{1}{\frac{\delta_{ei}}{k_i} + \frac{\delta_{pi}}{k_p}}$$

$$U_i = \frac{1}{\frac{\delta_{ei}}{k_i} + \frac{\delta_{pi}}{k_p}} \qquad \text{(Ec 3.10)}$$

Se reemplazan (Ec 3.6), (Ec 3.7) y (Ec 3.8) en (Ec 3.5) y se obtiene la **ecuación discretizada de la difusión para un volumen de control** (Ec 3.11):

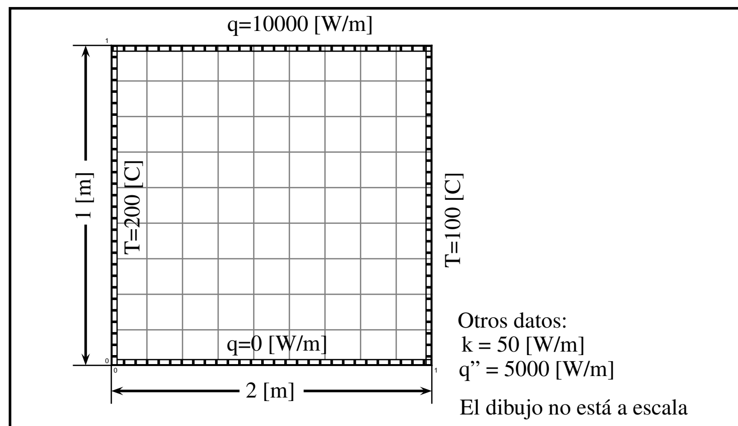
$$\sum_{i=1}^{n} [U_i \, A_i \, (T_i - T_p)] + q \; \Omega_p = 0 \qquad \text{(Ec 3.11)}$$

3.1 IMPLEMENTACIÓN

La (Ec 3.11) se aplicará de dos formas para dar fundamento al código de computador presentado. Se presenta una manera intuitiva de tratar el problema de la difusión y una forma más general que permite mayor flexibilidad y requiere la escritura de menos líneas de código.

Para mostrar el proceso de difusión se escoge un ejemplo sencillo, una placa rectangular con las condiciones de frontera mostradas en la figura 15.

Figura 15: Definición del problema de difusión en una placa plana.



El objetivo será encontrar y graficar el perfil de temperaturas a través de la placa. El análisis se realiza en dos dimensiones y se asume que la placa tiene una profundidad igual a la unidad. En esta sección se presenta la forma como normalmente se desarrollaría el problema del flujo difusivo utilizando una malla ortogonal, de una manera rígida y posteriormente, se presenta un desarrollo más general para el mismo problema. Como información adicional, se usará una malla estructurada y específicamente una malla ortogonal, para la primera parte (3.1.1) se usará un tamaño de celda fijo y, en los dos códigos que se presentan, se asume que el valor de las propiedades está ubicado en el centro de la celda de la malla.

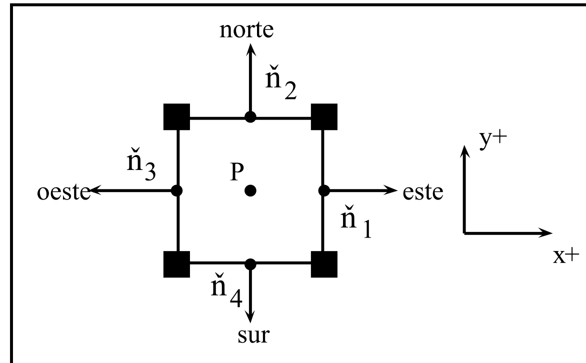
3.1.1 Método intuitivo

En este método se asume que cada volumen de control es rectangular, tiene el mismo tamaño y está alineado con los ejes coordenados como se muestra en la figura 16. Se desarrolla la sumatoria de (Ec 3.11) y se obtiene (Ec 3.12):

$$U_{norte}\, A_{norte} = \frac{k}{\Delta y}\,\Delta x \quad U_{este}\, A_{este} = \frac{k}{\Delta x}\,\Delta y \quad U_{sur}\, A_{sur} = \frac{k}{\Delta y}\,\Delta x \quad U_{oeste}\, A_{oeste} = \frac{k}{\Delta x}\,\Delta y$$

$$\frac{k\Delta x}{\Delta y}(T_N - T_P) + \frac{k\Delta x}{\Delta y}(T_S - T_P) + \frac{k\Delta y}{\Delta x}(T_E - T_P) + \frac{k\Delta y}{\Delta x}(T_W - T_P) + \Delta x \Delta y\, q = 0 \quad \text{(Ec 3.12)}$$

Figura 16: Vectores unitarios $\hat{n}_i$ para un volumen de control rectangular.



La (Ec 3.12) es una ecuación lineal de la forma $a_e\, T_e + a_n\, T_n + a_w\, T_w + a_s\, T_s + a_p\, T_P = b$ que se aplica a cada volumen de control. La información de todas las ecuaciones se almacenará en la matriz de coeficientes y el vector de términos independientes que, posteriormente, se resolverá para encontrar el valor de la temperatura en cada volumen de control. Además se puede ver de (Ec 3.12) que el coeficiente a_P es igual a $a_P = -(a_e + a_n + a_w + a_s)$.

La (Ec 3.12) define los coeficientes a aplicar para cada volumen de control interno, es decir, el volumen de control al cual no se imponen condiciones de frontera en sus caras. Los coeficientes de la ecuación (Ec 3.12) se presentan en la tabla 3.

Tabla 3: Coeficientes de la ecuación de difusión para un volumen de control interno en 2D y una malla de tamaño de celda fijo.

a_e	a_w	a_n	a_s	a_p	b
$\frac{k\Delta y}{\Delta x}$	$\frac{k\Delta y}{\Delta x}$	$\frac{k\Delta x}{\Delta y}$	$\frac{k\Delta x}{\Delta y}$	$-(a_e + a_n + a_w + a_s) = -\sum a_i$	$-\Delta x \Delta y\, q = b_{gen}$

Cuando se presenta una condición de frontera en el volumen de control se debe modificar los coeficientes de la tabla 3 de acuerdo al tipo de frontera que puede ser:

Frontera tipo Dirichlet: en este tipo de frontera se define el valor de la temperatura en la frontera. Los coeficientes a aplicar para una frontera tipo Dirichlet se muestran en la tabla 4.

Tabla 4: Modificación de los coeficientes debido a la condición Dirichlet en la frontera.

Valor dado	Se calcula	Términos a modificar
$T_{e(wall)}$	$a_{dir,e} = \frac{k\Delta y}{\Delta x/2}$ $b_{dir,e} = -a_{dir,e}\, T_{e(wall)}$	$a_P = -(a_{dir,e} + a_n + a_w + a_s)$ $a_e = \varnothing \qquad b = b_{gen} + b_{dir,e}$
$T_{n(wall)}$	$a_{dir,n} = \frac{k\Delta x}{\Delta y/2}$ $b_{dir,n} = -a_{dir,n}\, T_{n(wall)}$	$a_P = -(a_e + a_{dir,n} + a_w + a_s)$ $a_n = \varnothing \qquad b = b_{gen} + b_{dir,n}$
$T_{w(wall)}$	$a_{dir,w} = \frac{k\Delta y}{\Delta x/2}$ $b_{dir,w} = -a_{dir,w}\, T_{w(wall)}$	$a_P = -(a_e + a_n + a_{dir,w} + a_s)$ $a_w = \varnothing \qquad b = b_{gen} + b_{dir,w}$
$T_{s(wall)}$	$a_{dir,s} = \frac{k\Delta x}{\Delta y/2}$ $b_{dir,s} = -a_{dir,s}\, T_{s(wall)}$	$a_P = -(a_e + a_n + a_w + a_{dir,s})$ $a_s = \varnothing \qquad b = b_{gen} + b_{dir,s}$
El coeficiente a_i para la frontera Dirichlet i no se asigna pues no existe ningún volumen de control en esa dirección		

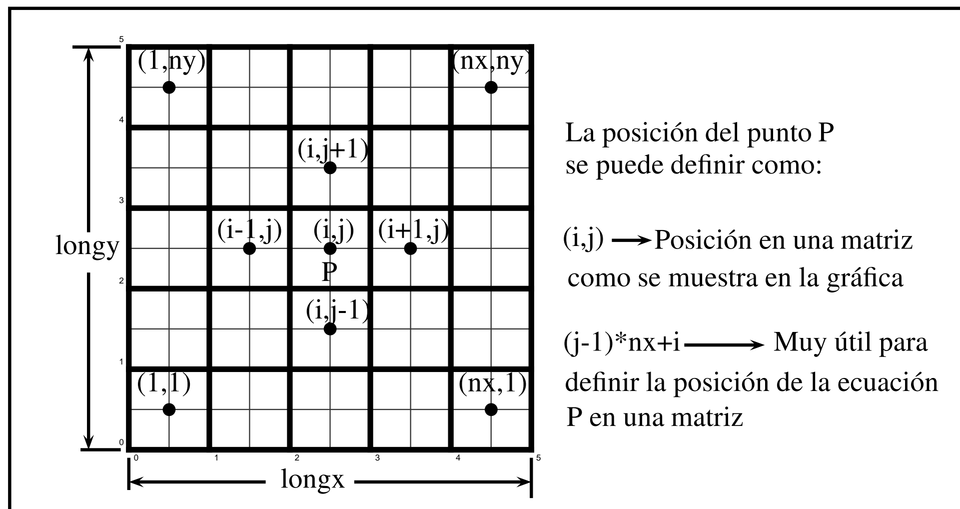
Frontera tipo Neumann: se presenta cuando se define el valor de la derivada en la superficie de control. Existe un caso especial de la frontera tipo Neumann para la temperatura y es el flujo de calor por unidad de área. El flujo de calor está relacionado con el gradiente de la temperatura mediante $q_{wall} = -k(\frac{\partial T}{\partial x_i})_{wall}$. Los coeficientes para una condición de frontera tipo Neumann se dan en la tabla 5.

Tabla 5: Modificación de los coeficientes debido a la condición Neumann en la frontera.

Valor dado	Se calcula	Términos a modificar
$\left(\frac{\partial T}{\partial x}\right)_{e(wall)}$	$b_{neu,e} = -k\,\Delta y \left(\frac{\partial T}{\partial x}\right)_{e(wall)}$	$a_P = -(a_n + a_w + a_s)$ $a_e = \varnothing \qquad b = b_{gen} + b_{neu,e}$
$\left(\frac{\partial T}{\partial y}\right)_{n(wall)}$	$b_{neu,n} = -k\,\Delta x \left(\frac{\partial T}{\partial y}\right)_{n(wall)}$	$a_P = -(a_e + a_w + a_s)$ $a_n = \varnothing \qquad b = b_{gen} + b_{neu,n}$
$\left(\frac{\partial T}{\partial x}\right)_{w(wall)}$	$b_{neu,w} = -k\,\Delta y \left(\frac{\partial T}{\partial x}\right)_{w(wall)}$	$a_P = -(a_e + a_n + a_s)$ $a_w = \varnothing \qquad b = b_{gen} + b_{neu,w}$
$\left(\frac{\partial T}{\partial y}\right)_{s(wall)}$	$b_{neu,s} = -k\,\Delta x \left(\frac{\partial T}{\partial y}\right)_{s(wall)}$	$a_P = -(a_e + a_n + a_w)$ $a_s = \varnothing \qquad b = b_{gen} + b_{neu,s}$
El coeficiente a_i para la frontera Neumann i no se asigna pues no existe ningún volumen de control en esa dirección. Se puede ver que no hay contribución a a_P de parte de la condición de frontera Neumann i		

Por otro lado y antes de entrar en detalle en la programación, se debe asignar un número a cada volumen de control, para realizar este trabajo se utiliza la definición presentada en la Figura 17.

Figura 17: Subíndices del elemento P.



De manera general, se necesitan los siguientes pasos para desarrollar el código de computador:

1. Declarar la clase `diftemp`.
2. Agregar las variables de la clase `diftemp`.
3. Agregar funciones a la clase.
4. Implementación de las funciones.

A continuación se explica cada uno de los pasos necesarios para desarrollar la clase.

Paso 1. Declarar la clase diftemp. Al crear una clase en Visual Studio se crean por defecto dos archivos, el archivo de encabezado con el nombre de la clase de extensión ".h", `diftemp.h`, y el archivo de código fuente con el nombre de la clase de extensión ".cpp", `diftemp.cpp`. En el archivo de encabezado se encuentra la siguiente información:

```
#pragma once
class diftemp
{
public:
diftemp(void);
```

```
˜diftemp(void);
};
```

1. Crear la clase: la instrucción `#pragma once` permite la optimización del tiempo de compilación al compilar una única vez el archivo. A continuación se encuentra la declaración de la clase, `class diftemp`, la cual tiene dentro de llaves la declaración de las funciones y datos de la clase. Con respecto a los miembros que componen una clase, estos se pueden declarar como `public` si se desea que otras partes del programa tengan acceso al elemento, `private` si se desea que solo el objeto tenga acceso a esos elementos o `protected`[1]. Toda clase tiene por lo menos dos métodos `public`, el constructor `diftemp(void)` y el destructor `˜diftemp(void)`. El constructor contiene la información necesaria para inicializar el objeto, en este caso, se usa solo el constructor sin argumentos y se añaden dos instrucciones que asignan el valor `false` a dos variables de tipo `bool` que posteriormente serán explicadas. Por otro lado, el destructor contiene las instrucciones a ejecutar al eliminar el objeto y no puede tener parámetros de entrada; en este caso, no es necesario ejecutar instrucciones adicionales por lo que se cambia el signo ; por dos llaves `{}`.

```
#pragma once
class diftemp
{
public:
diftemp(void){ leido= false; calculado=false;}
˜diftemp(void){}
};
```

Paso 2 Agregar las variables de la clase `diftemp`. Un objeto de la clase `diftemp` necesita de un conjunto de datos para almacenar la información del problema que posteriormente solucionará. Las variables a agregar a la clase se muestran a continuación:

1. Agregar los datos del problema: por defecto, los datos de la clase se declaran como miembros `private` de manera que solo el mismo objeto pueda modificar los valores; cuando se necesita modificar o mostrar el valor de una variable fuera del objeto, se debe crear una función de acceso que realice dichos cambios y, en algunos casos, modifique otras variables para una correcta ejecución del programa.

```
...
#include <real.h>
class diftemp
{
private:
real longx,longy ;
```

[1] Se presenta un resumen de la declaración de diferentes tipos de elementos en un clase en el Anexo A.

```
int  nx,   ny   ;
real dx,   dy   ;
real k;
public:
...
};
```

Las variables creadas son: longitud de la placa en dirección x `longx`, longitud de la placa en dirección y `longy`, número de volúmenes de control en dirección x `nx` y en dirección y `ny`, tamaño de un volumen de control[2] en dirección x `dx` y en dirección y `dy`. Por último, se crea la variable que almacena la información de las propiedades del material, la conductividad térmica `k`. Se utilizan dos tipos de variables, `int` y `real`, las variables tipo `int` son enteros con signo que utilizan 4 bytes para su almacenamiento, en cambio, el tipo de variable `real` esta definida en `real.h` por lo que es necesario agregar `#include <real.h>`, la ventaja presentada al usar el tipo `real` se encuentra en el manejo de un tipo común de datos para las librerías siendo posible escoger entre datos de tipo `float`, para un menor consumo de espacio de memoria, o datos de tipo `double`, para una mayor precisión en las operaciones.

2. Agregar objetos `Matriz` y `Vector`: este tipo de objetos presenta múltiples ventajas con respecto a los vectores o matrices tradicionales en C++; para la clase `Matriz`, se tienen las operaciones de suma y multiplicación, el determinante de la matriz, varios tipos de factorización, un fácil dimensionamiento del objeto, métodos de escritura y lectura, entre otras funciones. Para la clase `Vector` se tienen, entre otras características, las operaciones matemáticas para vectores, redimensionamiento, guardado, métodos de lectura y escritura. Para acceder a la clase `Vector` y la clase `Matriz` se debe incluir en el encabezado de archivo las librerías `Matriz.h` y `Vector.h`.

Además de la clase `Matriz` existen otros tipos de matrices, entre las que se encuentran: la clase `Matriz_basico` que permite manejar datos que no necesiten operaciones matemáticas, matrices eficientes en el uso del espacio de memoria como, matriz diagonal `Matriz_Diag`, matriz bandeada `Matriz_Band`, matriz tridiagonal `Matriz_Tri` y matriz dispersa `Matriz_Dispersa`.

Para usar las matrices y los vectores se debe agregar un argumento a la plantilla para definir el tipo de datos a utilizar, por esta razón, la declaración de un objeto tipo `Matriz` se realiza mediante la instrucción `Matriz<tipo_dato> objeto`;.

```
...
#include <Matriz.h>
#include <Vector.h>
class diftemp
{
```

[2] Todos los volumenes de control tienen el mismo tamaño y forma.

```
private:
...
Matriz<real>   matriz_a;
Vector<real>   b;
Vector<real>   x;
Matriz<real>   temp;
prm_Matriz <real> pm;
public:
...
};
```

Se debe almacenar la información para el sistema de ecuaciones lineales, por tal motivo, se crea la matriz de coeficientes `matriz_a`, el vector de términos independientes `b` y el vector solución `x`. Se crea la matriz `temp` para un método alternativo[3] de almacenamiento y acceso de la información del campo de temperatura en el problema. Adicionalmente se crea un objeto de tipo `prm_Matriz<real>` para almacenar la información del tipo de matriz a usar, si bien es cierto que en este caso no es necesario, se recomienda su creación, la importancia de su uso se explicará posteriormente.

3. Condiciones de frontera: la información del tipo de condición de frontera en cada lado se guarda en una cadena de caracteres de la clase `Cadena`, para acceder a esta clase es necesario agregar la librería correspondiente, `#include <Cadena.h>`; la clase `Cadena` define operaciones de recorte `trim`, `antes`, `despues`, `subcadena`, unión `juntar`, búsqueda de caracteres `contiene`, `primercar`, `ultimocar`, entre otras; el uso de la clase `Cadena` permite un acceso más sencillo a las operaciones con caracteres.

```
...
#include <Cadena.h>
class diftemp
{
private:
...
Cadena x0tipo,y0tipo,xfinaltipo,yfinaltipo;
public:
...
};
```

Se definen cuatro objetos tipo `Cadena`: condición de frontera para $x = 0$ o el lado sur `x0tipo`, para $y = 0$ o el lado oeste `y0tipo`, para $x = longx$ o el lado este `xfinaltipo` y para $x = longy$ o el lado norte `yfinaltipo`. En cada cadena se almacenará la palabra `dirichlet` o la palabra `neumann`.

[3] Ver la Figura 17.

4. Administrador de ecuaciones lineales: antes de hablar acerca del administrador de ecuaciones lineales se presenta la clase `Puntero`. La clase `Puntero` permite administrar inteligentemente el manejo de memoria dinámica. Para declarar un objeto de tipo `Puntero` se debe dar un argumento a la plantilla, `Puntero<argumento_plantilla> objeto;`. Por otro lado, la clase `AdmonEcLin` contiene las instrucciones necesarias para resolver, por diferentes métodos, un sistema de ecuaciones lineales ya sea directa o iterativamente. Para acceder a la clase `AdmonEcLin` se debe agregar al encabezado `#include <AdmonEcLin.h>`.

```
...
#include <AdmonEcLin.h>
class diftemp
{
private:
...
Puntero<AdmonEcLin> admlin;
public:
...
};
```

Se crea un puntero inteligente `admlin` a un objeto `AdmonEcLin` para posteriormente realizar una correcta inicialización.

5. Parámetros de control: se deben crear dos variables que permitan verificar que el programa se ejecute en el orden correcto, por ejemplo, no se puede resolver un sistema de ecuaciones que aun no ha sido definido, posteriormente se mostrará detenidamente el uso de estas dos variables.

```
...
class diftemp
{
private:
...
bool  calculado;
bool  leido;
public:
...
};
```

En este caso, los parámetros de control son variables de tipo `bool`, las cuales son `calculado` y `leido`; el valor de las variables debe ser iniciado en `false`, proceso que se realiza en el constructor mediante `leido=false;calculado=false;`.

Paso 3. Agregar funciones a la clase. Todas las funciones se declaran `public` de manera que se puede acceder a ellas fuera de la clase; sin embargo, en algunos casos es conveniente

definir los métodos como `private` para evitar errores en la ejecución del programa. Se deben agregar los siguientes métodos a la clase:

1. Lectura de la información: se agrega el método `Leer` el cual toma la información de la geometría, las particiones y el solucionador de ecuaciones para realizar la inicialización de las variables de la clase. La función `Leer` presenta una sobrecarga la cual se define para dar un ejemplo del valor de los argumentos.

```
...
class diftemp
{
private:
...
public:
...
void Leer (real ancho, real alto, real divx,real divy);
void Leer (void)  {Leer(1.0,1.0,20,20);}
};
```

Los argumentos a introducir son el ancho de la placa `ancho`, la altura de la placa `alto` , el número de divisiones en el eje x `divx` y en el eje y `divy` . La versión sobrecargada tiene como parámetros por defecto una placa cuadrada de 1x1 metro y 20 divisiones en cada dirección. El método se define como `void` pues no es necesario entregar un valor como resultado.

2. Condiciones de frontera y generación de calor: se definen 5 funciones para las condiciones de frontera y la generación de calor dentro de la placa, estás funciones de tipo `real` presentan la ventaja de poder definir el valor de acuerdo a la posición. En el caso de las condiciones de frontera el único parámetro es la coordenada que permite obtener en un punto específico el valor de la temperatura o el valor del flujo de calor en la frontera por unidad de longitud. Por otro lado, se encuentra la función `q` la cual devuelve el valor de la generación de calor por unidad de volumen dependiendo de la posición en x e y.

```
...
class diftemp
{
private:
...
public:
...
real condx0 (real y);
real condy0 (real x);
real condxf (real y);
real condyf (real x);
```

```
real q (real x, real y );
};
```

La ventaja lograda al definir 5 funciones y no 5 variables es permitir a la librería ser más general, de manera que se pueden definir varias combinaciones de fronteras y generación de calor.

3. Funciones de procesamiento: se agrega una función que calcula los coeficientes del sistema de ecuaciones lineales, dicha función se llama `calcular` y no requiere de parámetros de funcionamiento ni devolver un valor. Después de realizar el llenado del sistema de ecuaciones se debe resolver el sistema mediante la llamada a la función `resolver` que no necesita parámetros pero devuelve un valor booleano que indica si se logró obtener la solución al sistema de ecuaciones.

```
...
class diftemp
{
private:
...
public:
...
void calcular();
bool resolver();
};
```

4. Postprocesamiento: para llevar acabo esta etapa se agrega la función `exportar` que se encarga de tomar los resultados del vector solución y generar un archivo que puede ser leído en Matlab. La función `exportar` tiene un único argumento con el nombre del archivo a crear.

```
...
class diftemp
{
private:
...
public:
...
void exportar(Cadena archivo);
};
```

La forma final de la declaración de la clase se encuentra en el Anexo B.

Paso 4. Implementación de las funciones. La implementación de la clase se escribe en el archivo de código fuente `diftemp.cpp`. Al crear una clase en Visual Studio, el archivo de código fuente contiene la instrucción de inclusión del archivo de encabezado correspondiente a la clase, `diftemp.h` en este caso.

```
#include  "diftemp.h"
```

No es necesario incluir otras librerías en el archivo de código fuente debido a que la inclusión se realiza en el archivo `diftemp.h`. A continuación se especifican algunos de los métodos agregados a la clase.

1. Método `Leer`: la versión de la función `Leer` con 4 argumentos está compuesta de las siguientes etapas.

   I. Inicialización de la geometría: se toman los parámetros de la función y se almacenan en las variables correspondientes de la clase. De esta manera, se modifica el tamaño de la placa `longx` y `longy`, el número de divisiones en cada dirección `nx` y `ny` y el tamaño de un volumen de control en la dirección x `dx` e y `dy`.

```
void diftemp::Leer (real ancho, real alto, real divx,real divy)
{
longx = ancho ; longy = alto ;
nx   = divx ; ny    = divy ;
dx   =longx/nx ; dy   = longy/ny;
...
```

   II. Dimensionamiento y llenado de las matrices y los vectores: las variables a dimensionar son en orden respectivo: la matriz de coeficientes, el vector de términos independientes, el vector solución y la matriz que almacena el valor de la temperatura. Se sugiere realizar el llenado inicial de todos los vectores y las matrices pero solo es estrictamente necesario en la matriz de coeficientes `matriz_a`, el vector de términos independientes `b` y el vector solución `x` el cual se inicia con el valor de `190.0` para disminuir el número de iteraciones necesarias en el método de solución.

```
...
matriz_a.chaSize(nx*ny);    matriz_a.llenar(0.0);
b.chaSize(nx*ny);    b.llenar(0.0);
x.chaSize(nx*ny);    x.llenar(190.0);
temp.chaSize(nx,ny);    temp.llenar(0.0);
...
```

   III. Selección del tipo de frontera: se escribe una cadena de texto con los dos tipos de frontera más comunes. Para el tipo de frontera `dirichlet` se da el valor de la propiedad en la superficie; en cambio, para la frontera `neumann` se da el gradiente de la propiedad en dirección normal a la superficie, en el caso que la propiedad es la temperatura, dicho gradiente es proporcional al flujo de calor.

```
...
x0tipo = "dirichlet";  xfinaltipo = "dirichlet";
y0tipo = "neumann";     yfinaltipo = "neumann";
...
```

IV. Definir las propiedades del material: solo es necesario definir la conductividad térmica del material.

```
...
k = 50;
...
```

V. Inicializar el administrador de ecuaciones lineales: para que el proceso sea general, es conveniente leer la información del método de solución en un archivo de texto llamado `solve_datos.txt`. El archivo[4] contiene tres partes, la información de la matriz, la información del método de solución y la información de los precondicionadores.

```
Parametros de la matriz (clase prmMatriz);
>guarda  = Matriz;
>guarda_sim = FALSE
>pivoteo = FALSE
>umbral =0.0

prm_solucionEcLin parametros:
>solver_classname=Orthomin
>metodo=R-OM(7)
>cambiar=0.0
>pivoteo=NO_PIVOT
>relajacion=1.2
>startvector=USER_START
>criterio_def=TRUE
>nextern_crit=0
>maxit=1000
>estimate_eigvals=FALSE

Parametros del precondiciondor:
>precondicionador=PrecIdentidad
>left_prec=TRUE
>auto_iniciar=TRUE
>nivel_llenado=0
>prec_RILU=1
>prec_SSOR=1
>pasos_int=1
```

[4]La información que debe contener el archivo mostrado hace parte del uso de las librerías, no se hará un análisis detallado de esa información en este proyecto.

Para leer el archivo de texto es necesario crear un objeto de tipo `Is`. Seguidamente, se leen los parámetros de la matriz con el objeto de tipo `prm_Matriz`, se inicia el puntero inteligente `admlin` con un nuevo objeto de la clase `AdmonEcLin` y se leen los parámetros correspondientes al método de solución y al precondicionador.

```
...
Is is("ARCH=solve_datos.txt");
pm.Leer ( is );
admlin.vincular(new AdmonEcLin(EXTERNAL_STORAGE));
admlin().Leer(is);
...
```

VI. Asignar el valor a la bandera: finalmente se debe asignar `true` para reconocer que el programa ya ejecutó este procedimiento y por lo tanto puede seguir con el siguiente paso.

```
..
leido=true;//bandera
}
```

2. Condiciones de frontera y generación de calor: la definición de las funciones es relativamente sencilla por lo que no se hace una explicación adicional de estas funciones:

```
real diftemp::condx0(real y) {  return 150.0; }
real diftemp::condxf(real y) {  return 100.0; }
real diftemp::condy0(real x) {  return 0.0; }
real diftemp::condyf(real x) {  return 200.0; }
real diftemp::q(real x,real y) { return 5000.0; }
```

3. Método `Calcular`: para calcular los coeficientes del sistema de ecuaciones lineales se requieren los siguientes pasos.

I. Verificación del parámetro de control: antes de realizar cualquier proceso, se debe verificar que el código ha sido ejecutado de la manera correcta, es por esto que se debe verificar el valor de la bandera `leido`.

```
void diftemp::calcular()
{
if (leido==false)
{ Leer( ); std_o<<"Advertencia: posible error en la ejecucion"; }
if (calculado== true)
{std_o<<"Ya se habia calculado los coeficientes\n"<<
"POSIBLES ERRORES EN LOS RESULTADOS\n";}

...
```

II. Declaraciones: se declaran las variables a usar en `calcular()` como contadores y otros.

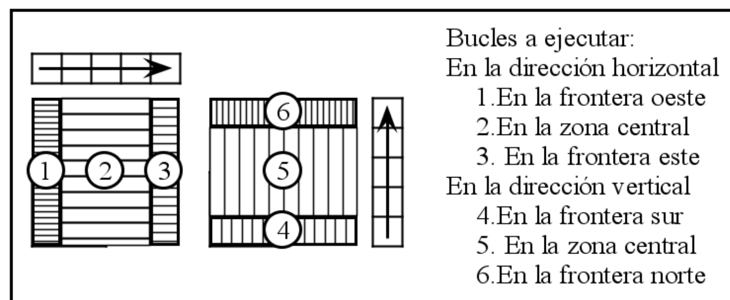
```
...
int  i, j;
real dxsy = dx/dy;
...
```

III. Cálculo del efecto de la generación de calor: existen diferentes formas de comenzar a asignar los coeficientes a la matriz y al vector, se sugiere comenzar por asignar al vector de términos independientes el valor de la generación de calor mostrado en la tabla 3. El término $(j-1)*nx+i$ se toma de la Figura 17.

```
...
for( i=1 ; i<=nx ; i++ )
for (j=1 ; j<=ny ; j++)
  b( (j-1)*nx + i ) = -dx*dy* q( (i-0.5)*dx ,(j-0.5)*dy );
...
```

Para el llenado de los coeficientes de la matriz se sigue el procedimiento mostrado en la Figura 18: se realizan 6 bucles consecutivos para asignar el valor de los coeficientes, primero se analiza el efecto este-oeste desde la frontera oeste, pasando por la zona central hasta la frontera este, seguidamente se agrega el efecto norte-sur comenzando los cálculos desde la zona inferior hasta la última fila.

Figura 18: Orden de ejecución de los bucles para el llenado de los coeficientes de la matriz.



IV. Asignación del efecto este-oeste en la frontera oeste: lo primero que se debe hacer es verificar el tipo de frontera, en este caso, si no corresponde al tipo de frontera `dirichlet` corresponderá a la frontera tipo `neumann`. Los subíndices usados en la matriz de coeficientes corresponden a `matriz_a(numero_nodo_actual,numero_nodo_vecino)`.

A. Para la condición de frontera `dirichlet`: se toman los coeficientes de la tabla 4 y se asignan los valores correspondientes a la matriz de coeficientes y al vector de términos independientes.

```
...
for(j=1;j<=ny;j++)
{
if(x0tipo.contiene("diri") || x0tipo.contiene("Diri"))
{
matriz_a( (j-1)*nx+1,  (j-1)*nx+1)  -= 3.0*k/dxsy;
matriz_a( (j-1)*nx+1,  (j-1)*nx+2)   =  1.0*k/dxsy;
b( (j-1)*nx+1 ) += - condx0( (j-0.5)*dy ) * 2.0*k/dxsy;
}
...
```

B. Para la condición de frontera `neumann`: se tienen en cuenta los valores presentados en la tabla 5. Se asigna el valor correspondiente al volumen actual, el volumen este y por último, al vector de términos independientes.

```
...
else
{
matriz_a( (j-1)*nx+1 , (j-1)*nx+1 ) -= k/dxsy;
matriz_a( (j-1)*nx+1 , (j-1)*nx+2 )  = k/dxsy;
b       ( (j-1)*nx+1 )              -= k*dy*condx0 ( (j-0.5)*dy ) ;
}
...
```

V. Asignación del efecto este-oeste en la zona central: en este caso no se tiene en cuenta las condiciones de frontera y el valor de los coeficientes se extrae de la tabla 3.

```
...
for(i=2;i<=nx-1;i++)
{
for(j=1;j<=ny;j++)
{
matriz_a((j-1)*nx+i , (j-1)*nx+i  )  -= 2.0*k/dxsy;//nodo actual
matriz_a((j-1)*nx+i,  (j-1)*nx+i+1)  = 1.0*k/dxsy;//nodo east
matriz_a((j-1)*nx+i,  (j-1)*nx+i-1)  = 1.0*k/dxsy;//nodo west
}
}
...
```

VI. Asignación del efecto este-oeste en la frontera este: los coeficientes se toman de las tablas 4 y 5.

```
...
for(j=1;j<=ny;j++)
{
if(xfinaltipo.contiene("diri") || xfinaltipo.contiene("Diri"))
{
```

```
matriz_a( j*nx, j*nx   )  -= 3.0*k/dxsy;
matriz_a( j*nx, j*nx-1 )   = 1.0*k/dxsy;
b       ( j*nx )          -= condxf( (j-0.5)*dy )*2.0*k/dxsy;
}
else
{
matriz_a( j*nx , j*nx   ) -= k/dxsy;
matriz_a( j*nx , j*nx-1 ) = k/dxsy;
b( j*nx    ) -=  k*dy*condxf  ( (j-0.5)*dy ) ;
}
}
...
```

VII. Asignación del efecto norte-sur: se procede de una forma similar al paso anterior.

```
...
for(i=1;i<=nx;i++)
{
if(y0tipo.contiene("Diri") ||y0tipo.contiene("diri"))
{
matriz_a( i,  i    )  -= 3.0*k*dxsy;
matriz_a( i,  i+nx )  = 1.0*k*dxsy;
b( i )  -= condy0( (i-0.5)*dx ) * 2.0*k*dxsy;
}
else
{
matriz_a(i,    i)  -= k*dxsy ;
matriz_a(i, i+nx)   = k*dxsy;
b  ( i )           -= k*dx*condy0 ( (i-0.5)*dx ) ;
}
}
for(i=1;i<=nx;i++)
{
for(j=2;j<=ny-1;j++)
{
matriz_a((j-1)*nx+i , (j-1)*nx+i)  -= 2.0*k*dxsy;
matriz_a((j-1)*nx+i , (j-0)*nx+i)  = 1.0*k*dxsy;
matriz_a((j-1)*nx+i , (j-2)*nx+i)  = 1.0*k*dxsy;
}
}
for(i=1;i<=nx;i++)
{
if(yfinaltipo.contiene("Diri") ||yfinaltipo.contiene("diri"))
{
```

```
    matriz_a( (ny-1)*nx+i, (ny-1)*nx+i )  -= 3.0*k*dxsy;
matriz_a( (ny-1)*nx+i, (ny-2)*nx+i )  = 1.0*k*dxsy;
b( (ny-1)*nx+i  )-= condyf( (i-0.5)*dx )*2.0*k*dxsy;
}
else
{
matriz_a( (ny-1)*nx+i , (ny-1)*nx+i ) -= k*dxsy;
matriz_a( (ny-1)*nx+i , (ny-2)*nx+i ) = k*dxsy;
b( (ny-1)*nx+i ) -=  k*dx*condyf ((i-0.5)*dx ) ;
}
}
...
```

VIII. Finalmente, se modifica el valor de la bandera de manera que la instancia de `diftemp` reconozca que ya ha sido ejecutado este proceso:

```
...
calculado=true;
}
```

4. Método `resolver`: para resolver el sistema de ecuaciones solo es necesario dos instrucciones, pero se debe realizar otros procedimientos adicionales para un mejor manejo de la información:

I. Verificación del parámetro de control: solo es posible resolver el sistema de ecuaciones una vez se haya asignado los coeficientes a la matriz y al vector de términos independientes, por lo tanto, si no se ha ejecutado el proceso `calcular()` se debe llamar a dicho proceso o generar una advertencia.

```
bool diftemp::resolver()
{
if (calculado==false)
{std_o<<"Advertencia: posible error en la ejecucion";  calcular( ); }
...
```

II. Solución de las ecuaciones: para resolver el sistema de ecuaciones se debe indicar al administrador de ecuaciones lineales `admlin()` cuál es la matriz de coeficientes, el vector solución y el vector de términos independientes. Una vez realizado esto, se llama a la función `solve()`de `admlin()` que devuelve un valor de tipo `bool` el cual es `true` si fue encontrada la solución o `false` si el sistema no pudo llegar a la solución.

```
...
admlin().adjuntar(matriz_a,x,b);
bool solucion=admlin().solve();
...
```

III. Asignación a la matriz de temperatura: una vez se ha encontrado el vector solución se transfiere la información a la matriz `temp` con los subíndices mostrados en la figura 17.

```
...
for( int i=1 ;  i<=nx ; i++)
for( int j=1 ;  j<=ny ; j++)
temp(i,j)=x( (j-1)*nx + i );
...
```

IV. Finalmente se devuelve el valor booleano que permite conocer si se ha encontrado la solución al problema o no.

```
...
return solucion;
}
```

5. Etapa de post-procesamiento, `exportar`: los resultados serán guardados en un archivo ".m" el cual puede ser leído por Matlab. De manera general, se crean dos vectores que contienen la posición del centro de los volumenes de control en cada dirección, se genera una matriz con el campo de temperaturas, se genera una gráfica y se guarda la información en el archivo.

```
void diftemp::exportar ( Cadena archivo) //public
{
int i,j;
Os exportar(archivo);
exportar << "%Difusion con termino fuente\n" ;
exportar << "clear";
exportar << "\n%Mallado: " <<  aform(
"d=2 dominio = [0,%10g]x[0,%10g] indices= [1: %5d]x[1:%5d]",
longx , longy , nx , ny )<<"\n\n";
exportar << "'Posicion x',Px=zeros(1,"<<nx<<");\n" ;
exportar << "Px=["<< 0.5*dx;
for(i=2;i<=nx;i++)   exportar << "," <<  (i-0.5)*dx;
exportar << "];\n";
exportar << "'Posicion y',Py=zeros(1,"<< ny<<");\n";
exportar <<"Py=["<<  0.5*dy;
for(j=2;j<=ny;j++)   exportar << "," <<  (j-0.5)*dy;
exportar << "];\n";
exportar << "\n'Temperatura',temp=zeros(" << ny << "," << nx <<");\n" ;
for(j=1;j<=ny;j++)
{
exportar << "temp("<<j<<",:)=[" << temp(1,j);
for(i=2;i<=nx;i++) exportar << "," << temp(i,j);
exportar << "];\n";
```

```
 }
 exportar<< "\nfigure(1); surf(Px,Py,temp);%hold on;\n"<<
 "title('Temperatura')\n"
 <<"xlabel('Eje x[m]');ylabel('Eje y [m]');\n";
 exportar->actualizar();  exportar->cerrar();
 }
```

Paso 5. Creación de un objeto de la clase `diftemp`. Es necesario incluir la librería creada mediante comillas porque se encuentra en el directorio del proyecto. Dentro del cuerpo principal del programa, se encuentra la función `main()`. Inicialmente, dentro de la función `main()` debe llamarse a la función `iniciarLIB` para una correcta inicialización del sistema. Después, se crea un objeto de la clase `diftemp`, se ejecuta la función `Leer(2.0,1.0,6,10)` en donde los valores dados son el tamaño de la placa y el número de particiones en cada dirección. Posteriormente, se llama a la función `calcular()` y luego `resolver()`. Finalmente, se llama a la función `exportar` con el argumento `''ARCH=difusion_6_10.m''` en donde `difusion_6_10.m` es el nombre del archivo a crear. En algunos casos puede ser recomendable imprimir un valor en la pantalla para conocer el estado de ejecución del programa además de introducir una función que permita mantener en pantalla la información por un tiempo determinado o hasta realizar una acción determinada, por tal motivo, se realiza una salida en pantalla mediante el `std_o` y se utiliza `system("pause")` (definido en `stdlib.h`) bloqueando la pantalla hasta que se oprima cualquier tecla. El código final se muestra a continuación:

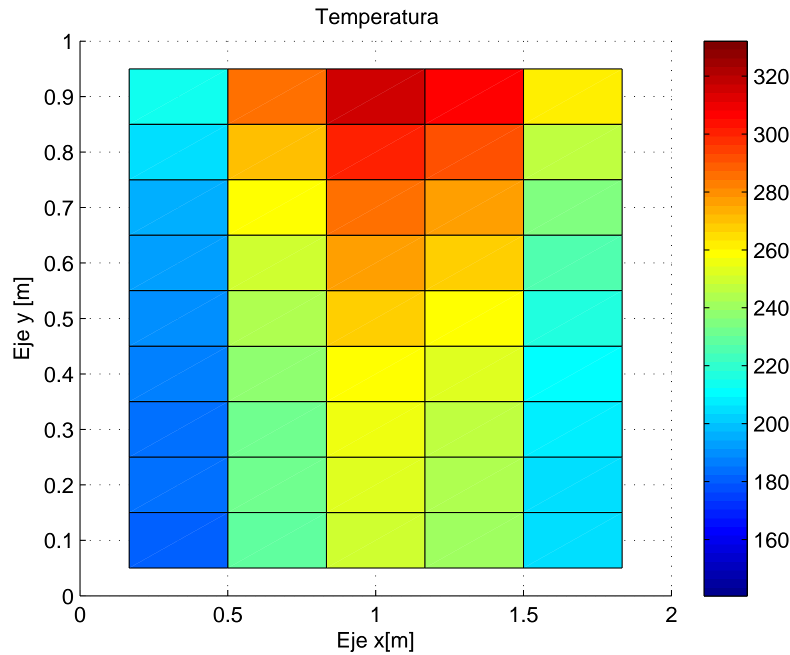
```
#include "stdafx.h"
#include <stdlib.h>
#include "diftemp.h"

int main(int argc,const char ** argv)
{
iniciarLIB(argc,argv);
diftemp difusivo;
difusivo.Leer(2.0,1.0,6,10);
difusivo.calcular();
difusivo.resolver();
difusivo.exportar("ARCH=difusion_6_10.m");
std_o<<"Hecho";
system("pause");

}
```

Al abrir y ejecutar el archivo `difusion_6_10.m` en Matlab se obtiene la figura 19 con el comando `surf` que muestra el valor de la temperatura en el centro de cada volumen de control además de mostrar el tamaño de la placa.

Figura 19: Presentación de los resultados para la clase diftemp.



Adicionalmente se presenta en la tabla 6 el campo de temperaturas obtenido; un volumen de control se ubica mediante el subíndice horizontal y el subíndice vertical.

Tabla 6: Valor del campo de temperatura para el problema propuesto al usar la clase diftemp.

Nodo	1	2	3	4	5	6
1	182,160	229,001	250,005	241,672	204,001	140,494
2	182,733	230,327	251,647	243,313	205,327	141,067
3	183,915	233,018	254,958	246,625	208,018	142,249
4	185,783	237,153	259,995	251,662	212,153	144,116
5	188,468	242,856	266,839	258,506	217,856	146,801
6	192,181	250,296	275,593	267,260	225,296	150,514
7	197,256	259,690	286,375	278,042	234,690	155,590
8	204,218	271,302	299,310	290,976	246,302	162,552
9	213,902	285,430	314,516	306,182	260,430	172,235
10	227,650	302,379	332,091	323,757	277,379	185,984

El método presentado tiene varias limitaciones, se ha implementado en dos dimensiones y se requieren cambios sustanciales para realizar la implementación en 3D. Se encuentra otra limitación en la forma de cada volumen de control, el método presentado solo puede usarse para volúmenes de control rectangulares y numerados de la forma presentada en la figura 17 restringiendo también la forma de la placa.

3.1.2 Desarrollo mediante el uso de las librerías

Como se comentó anteriormente, se presenta un método general[5] para la solución de la ecuación de difusión estacionaria. Al desarrollar la (Ec 3.11) y hallar los coeficientes para la temperatura del volumen e y el volumen p, se obtienen los datos registrados en la tabla 7 para un volumen de control interno.

Tabla 7: Coeficientes de la ecuación de difusión para un volumen de control interno.

a_e	a_P	b
$U_i\, A_i$	$-\sum_{i=1}^{n} a_e$	$-q\ \Omega_p = b_{gen}$
El coeficiente mostrado para a_e se debe aplicar a cada uno de los volúmenes que rodean al volumen de control actual		

Como ya se había mencionado, se deben realizar modificaciones para imponer las condiciones de frontera; los cambios en los coeficientes para los dos tipos de condiciones de frontera se presentan en la tabla 8.

Tabla 8: Coeficientes de la ecuación de difusión para las condiciones de frontera.

Tipo de frontera	Se calcula		Sumar a a_P	Sumar a b
Dirichlet	$a_{dir,i} = \frac{k_i}{\delta_i} A_i$	$b_{dir,i} = -a_{dir,i}\, T_{i(wall)}$	$-a_{dir,i}$	$b_{dir,i}$
Neumann	$a_{neu,i} = 0$	$b_{neu,i} = -k\, A_i \left(\frac{\partial T}{\partial n_i}\right)_{(wall)}$	0	$b_{neu,i}$

En resumen, el coeficiente correspondiente a a_P se define de forma general como (Ec 3.13) en donde bc es el número de condiciones de fronteras existentes para el volumen de control analizado.

$$a_P = -\left[\sum_{i=1}^{n-bc} a_i + \sum_{i=1}^{bc} a_{b,i}\right] \tag{Ec 3.13}$$

El coeficiente b se define de forma general como (Ec 3.14)

$$b = b_{gen} + \sum_{i=1}^{bd} b_{dir,i} + \sum_{i=1}^{bn} b_{neu,i} = b_{gen} + \sum_{i=1}^{bc} b_{b,i} \tag{Ec 3.14}$$

[5] Válido para mallas estructuradas.

Para obtener los valores de (Ec 3.13) y (Ec 3.14) se debe remitir a diferentes tablas y ecuaciones como se resume en la tabla 9.

Tabla 9: Resumen de los coeficientes necesarios para aplicar la ecuación de difusión (Ec 3.11) a cada volumen de control.

Variable a calcular	Ecuación o tabla
U_i	(Ec 3.10)
a_e, b_{gen}	Tabla 7
$a_{dir,i}$, $b_{dir,i}$ $a_{neu,i}$, $b_{neu,i}$ $a_{b,i}$ $b_{b,i}$	Tabla 8

Para desarrollar el código a partir del método presentado en esta sección se hace uso de las librerías de mallado, solución de ecuaciones, exportación de resultados a Paraview, entre otras. Para la creación de la clase se tiene en cuenta los siguientes pasos:

1. Declarar la clase `diffgeneral`.
2. Agregar las variables de la clase `diffgeneral`.
3. Agregar los métodos a la clase.
4. Implementación de las funciones.
5. Crear un objeto de la clase.

Los pasos generales son los mismos pero, la implementación de algunas secciones del código son bastantes diferentes.

Paso 1. Declarar la clase `diffgeneral`. Se crea una clase llamada `diffgeneral` que contiene inicialmente la siguiente información en el archivo de encabezado.

```
#pragma once
class diffgeneral
{
public:
diffgeneral(void);
~diffgeneral(void);
};
```

Se realizarán modificaciones al archivo de encabezado (`diffgeneral.h`) hasta definir todos los elementos necesarios. En un primer paso es necesario definir la clase como una clase derivada de `GuardarEnsight` para hacer uso de las funciones que permitirán realizar el reporte de los resultados de una forma muy sencilla.

```
...
#include <GuardarEnsight.h>
class diffgeneral:public GuardarEnsight
{
public:
diffgeneral(void);
~diffgeneral(void);
};
```

Paso 2. Agregar las variables a la clase `diffgeneral`. Las variables necesarias se dividen en dos grupos:

1. Variables correspondients a la malla, cálculo de la geometría, conectividad y campo de temperatura: se crea la variable `malla` de tipo `Puntero<MallaFV>` la cual se encarga de manejar la información de la malla. Para realizar un uso correcto de la `malla` es necesario definir otras variables como: `elems` de tipo `Vector_basico <PunteroElmDefs>` que ayudará a realizar el cálculo de la geometría de cada volumen de control en la malla, `vecino` de tipo `VecinoFVM` el cual ayuda a calcular los elementos que rodean a cada volumen de control en la malla. Finalmente se declara el campo de temperatura llamado `Temperatura` del tipo `Puntero<CampoFVM>` en el cual se asignará los valores obtenidos del vector solución; para realizar este proceso es necesario declarar `Puntero<GradoLibertadFV> gdl` como ayuda para realizar una correcta conversión de la información.

   Es necesario declarar varias librerías para acceder a las clases como se muestra en el siguiente código:

```
//Archivo diffgeneral.h
...
#include <MallaFV.h>
#include <ConstOLeemalla.h>
#include <GradoLibertadFV.h>
class diffgeneral:public GuardarEnsight
{
private:
Puntero<MallaFV> malla;
Vector_basico <PunteroElmDefs> elems;
VecinoFVM  vecino;
Puntero<CampoFVM>     Temperatura;
Puntero<GradoLibertadFV> gdl;
public:
...
};
```

2. Variables correspondientes al sistema de ecuaciones, su solución y los parámetros de control, se muestran en el siguiente fragmento de código:

```
...
#include <AdmonEcLin.h>
class diffgeneral:public GuardarEnsight
{
private:
...
int nelementos  ;
Vector<real>   k;
Matriz<real>   matriz_a;
prm_Matriz <real> pm;
Vector<real>   b;
Vector<real>   x;
Puntero<AdmonEcLin> admlin;
bool  calculado;
bool  leido;
bool  solucionado;
public:
...
};
```

Las variables creadas son: el número de volúmenes de control, `nelementos`, la conductividad térmica definida como un vector, `k`, la matriz de coeficientes, `matriz_a`, el vector de términos independientes, `b`, el vector solución, `x`, el puntero del administrador de ecuaciones lineales (es necesario incluir `AdmonEcLin.h` para acceder a la clase del administrador de ecuaciones lineales) y finalmente tres parámetros de control: `calculado`, `leido` y `solucionado`.

Paso 3. Agregar los métodos a la clase. Ya no es necesario definir las condiciones de frontera como funciones porque la información de estas se escribirá en un archivo de texto. Las funciones a agregar son:

```
...
class diffgeneral:public GuardarEnsight
{
private:
...
public:
...
int Leer (Cadena archivo);
int calcular();
bool resolver();
virtual void Reportarresultados();
real q (real x, real y);
};
```

Para `Leer`, los argumentos son dos cadenas de texto que contienen las direcciones de dos archivos, como posteriormente se explicará. Se crea la función calcular que no toma parámetros y devuelve un valor entero. Por otro lado, se encuentra la función `resolver` que devuelve un valor `bool` indicando si se encontró la solución para el sistema de ecuaciones. El método encargado de exportar los resultados para ser visualizados en Paraview es `Reportarresultados()` el cual se declara como `virtual` para permitir que sea redefinida en clases derivadas. La generación de calor se define como una función dependiente de la posición.

La declaración completa de la clase `diffgeneral` se encuentra en el anexo D.

Paso 4. Implementación de las funciones. A continuación se presenta la definición de cada una de las funciones y algunos comentarios del proceso realizado en el archivo de código fuente `diffgeneral.cpp`.

1. Constructor y destructor: no es necesario definir el constructor y el destructor en el archivo de encabezado. Para el constructor se asigna el valor de `false` a los tres parámetros de control. No se realizan procesos adicionales a la liberación de memoria al llamar al destructor de la clase `diffgeneral`.

```
#include "diffgeneral.h"

diffgeneral::diffgeneral(void)
{
calculado = leido = solucionado = false;
}
diffgeneral::~diffgeneral(void) { }
...
```

2. Función `Leer`: solo se define la función `diffgeneral::Leer(Cadena archivo, Cadena solucionador)` en donde `archivo` y `solucionador` contienen cada uno un nombre de un archivo.

   I. Definición y declaración de variables: es necesario crear una variable de tipo `static const char` que contiene dos cadenas indispensables para obtener el comando correcto para crear la malla. Se crea una `Cadena` para obtener el argumento y se crea un objeto de tipo `Is` para leer un archivo.

   ```
   ...
   static const char *mallador_tb[] = {"mallador", NULL};
   Cadena cadmalla;
   Is is(archivo);
   ...
   ```

   Un ejemplo del contenido del archivo a leer es el siguiente:

```
> mallador = PREPROCESSOR=ConjMalladorSupElem/ARCH=datos_difusion.geom
/ARCH=datos_difusion.parts
```

Se puede ver que el archivo contiene la dirección de dos archivos, el archivo `datos_difusion.geom` y el archivo `datos_difusion.parts`. El archivo `datos_difusion.geom` contiene la geometría de la placa y las condiciones de frontera.

```
> nsd = 2;
> subdominios = 1;
> no_de_superels = 1;
>nbind = 4
>bname T=100 T=150 Q=10000 Q=0

>SupEl;
>subdominio_no = 1;
>tipoelemento = ElmB4n2D;
>fronteras = [1(1)] [3 (2)] [2 (3)] [4(4)];
>nodos = [1(0 0)]+[2(2 0)]+[3(0 1)]+[4(2 1)];
>lados=;
```

En cambio, el archivo `datos_difusion.parts` contiene el número de particiones a realizar en cada superelemento y el tipo de partición.

```
>nsd = 2;
>no_de_superels = 1;

>SE;>d=2;>e=ElmB4n2D;>div=[6,10];>grad=[1,1];
```

Las estructuras que forman estos archivos hacen parte de las librerías, y por lo tanto, en este proyecto solo se usan.

II. Creación de la malla: se lee el comando en el archivo de texto, se inicializa la memoria del puntero `malla` y se construye la malla.

```
...
is->obtComando(cadmalla,mallador_tb);
malla.vincular(new MallaFV());
ConstOLeeMalla(malla(),cadmalla);
...
```

III. Cálculo de la conectividad: se debe conocer los volúmenes de control que rodean a cada volumen para llevar a cabo la asignación de coeficientes, es por este motivo que se inicia `vecino` y se llama a la función `CalcConectividad`.

```
...
vecino.iniciar(malla());
malla->CalcConectividad(vecino);
...
```

IV. Creación del campo de temperaturas: debido a que `Temperatura` y `gdl` fueron declaradas como instancias de la clase `Puntero` se realizan las siguientes dos inicializaciones.

```
...
Temperatura.vincular(new CampoFVM(malla(),"Temperatura"));
gdl.vincular(new GradoLibertadFV(malla(),1));
...
```

El valor `1` en `GradoLibertadFV(malla(),1)` implica que en cada volumen de control existe un solo grado de libertad o incógnita.

V. Dimensionalización de los vectores y las matrices: se obtiene el número de volúmenes de control, se dimensionan e inicializan los siguientes elementos.

```
...
nelementos=malla->obtNoElem();
matriz_a.chaSize(nelementos, nelementos);  matriz_a.llenar(0.0);
b.chaSize(nelementos);  b.llenar(0.0);
x.chaSize(nelementos);  x.llenar(90.0);
k.chaSize(nelementos);  k.llenar(50);
elems.chaSize(nelementos);
for (int i=1; i<=nelementos;i++)
{
elems(i).refill( malla->obtTipoElem(i), malla->obtNoDimEspacio() );
elems(i)->CalcGeomParameters( malla->obtCoorElem(i) );
}
...
```

VI. Inicialización del administrador de ecuaciones lineales: el proceso se realiza de la misma forma como se había mostrado en la página 67.

```
...
Is solucionador("ARCH=Solve_datos.txt");
pm.Leer ( solucionador );
admlin.vincular(new AdmonEcLin(EXTERNAL_STORAGE));
admlin().Leer(solucionador);
...
```

VII. Actualización del parámetro de control y valor de retorno: se actualiza el valor de `leido` indicando que el proceso ha sido realizado. Se devuelve el valor `1` para dar a conocer que la función se ejecutó exitosamente.

```
...
leido=true;
return 1;
}
```

3. Función `calcular`: para calcular los coeficientes del sistema de ecuaciones se requiere, de forma general, los siguientes pasos.

I. Verificación de los parámetros de control: si no se ha cargado los datos de entrada mediante `Leer` no se ejecuta la asignación de coeficientes, por otro lado, si ya se había ejecutado la función `calcular`, no es necesario volver a realizar el procedimiento.

```
int diffgeneral::calcular(void)
{
if (leido==false)
{std_o<<"Advertencia: no se ha leido los datos"
<<" del problema\n"; return 0;
}
if (calculado)
{std_o<<"Ya se habia calculado los coeficientes\n"; return 0;}
...
```

II. Declaración de las variables de la función: se necesitan contadores y otros tipos de elementos necesarios para la ejecución de las instrucciones de asignación.

```
...
int i , j , l ;
int veci;
int conx, conxvec;
int h, nbi=malla->obtNoIndFront ();;
Vector_xyz  <real> posicion(3);
real u1,u2,ut;
Cadena boname, bovalue;
real valor;
...
```

III. Proceso de asignación: para cada volumen de control se agrega el efecto de la generación de calor y se realiza la sumatoria de flujos de calor que entran al volumen de control, ya sea el flujo dentro de la placa o el flujo en la frontera; dependiendo del tipo de frontera se realiza el cálculo de los coeficientes que se resumen en la tabla 9.

```
...
for(i=1;i<=nelementos;i++)
 {
  posicion = elems(i)->centroid(malla->obtCoorElem(i));
  b(i) = - elems(i)->getVolume()* q( posicion(1),posicion(2) );
  conx=elems(i)->GetNumOfConex();
  for( j=1 ; j<=conx ; j++ )
  {
   veci=malla->getConx(i,j);
   if (veci!=0)//lado interior
```

```
{
 conxvec=elems(veci)->GetNumOfConex();
 for ( l=1 ; l<=conxvec ; l++ )
 {
  if(malla->getConx(veci,l)==i)   break;
 }
 u1=k(i)/elems(i)->getDelta(j);
 u2=k(veci)/elems(veci)->getDelta(l);
 ut=1.0 / ( (1.0/u1)+ (1.0/u2) );
 matriz_a( i,  i  ) -= ut* elems(i)->getArea(j);
 matriz_a( i,veci) =  ut* elems(i)->getArea(j);
}
else //lado frontera
{
 for (h=1; h<=nbi; h++)
 {
 if (malla->LadoFront(i,j,h,elems(i)()))
  {
   boname=malla->obtNombreIndFront(h);
   bovalue=boname.despues('=');
   valor=atof(bovalue.carts());
   if ( boname.contiene("T") ||
      boname.contiene("Dirichlet")||
     boname.contiene("dirichlet")
    )
   {
    ut = k(i)/elems(i)->getDelta(j);
    matriz_a(i,i)   -=  ut * elems(i)->getArea(j);
     b(i)  -=  ut * elems(i)->getArea(j)*valor;
   }
   else
   {
    if (boname.contiene("Neumann")||
        boname.contiene("neumann") )
    {
    b(i)   -=  valor * k(i) * elems(i)->getArea(j);
    }
    if (boname.contiene("Q"))
    {
    b(i)   -=  valor * elems(i)->getArea(j);
    }
   }
  }
```

```
      }
    }
  }
 };
...
```

IV. Se modifica el valor de la bandera y se entrega un valor de retorno: se modifica el valor de `calculado` para poder continuar correctamente con el siguiente proceso. También se devuelve el valor `1` para indicar que el proceso se ejecutó correctamente.

```
...
calculado=true;
return 1;
}
```

4. Función `resolver`: además de resolver el sistema de ecuaciones, se presenta la información del número de iteraciones y tiempo de solución en pantalla y, se entrega un booleano como resultado; es necesario pasar la información contenida en el vector solución `x` al campo de temperatura `Temperatura()`, este trabajo es realizado mediante una función de la clase `GradoLibertadFV` llamada `vector2campo`.

```
bool diffgeneral::resolver()
{
if(calculado==false) calcular();
admlin().adjuntar(matriz_a,x,b);
bool solucion=admlin().solve();
int itera; bool conv;
admlin().obtEstadisticas(itera, conv);
std_o<<"iteraciones="<<itera<<"\nTiempo"<<
admlin().obttiempoCPUsolve() ;
if (conv==1) std_o<<"\nEl sistema si convergio";
gdl->vector2campo(x,Temperatura());
solucionado=true;
return solucion;
}
```

5. Función `Reportarresultados`: se verifica el parámetro de control y se llama a la función `volcar` la cual se define en la clase base `GuardarEnsight`.

```
void diffgeneral::Reportarresultados()
{
 if(solucionado == false) resolver();
 volcar(Temperatura(),0);
}
```

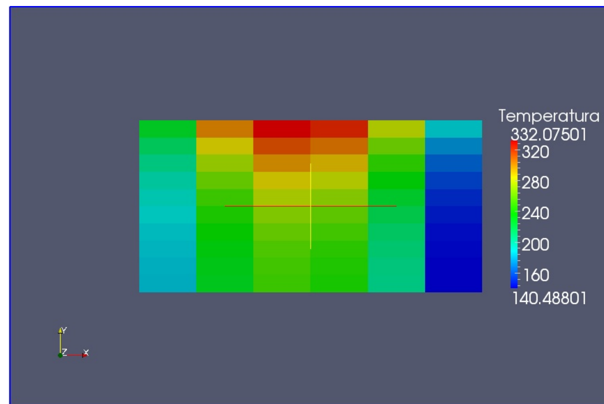
La información contenida en el archivo de código fuente `diffgeneral.cpp` se encuentra en el anexo E.

Los resultados al crear un objeto de la clase `diffgeneral` y ejecutar cada una de las instrucciones anteriormente enseñadas se muestran en la tabla 10 y en la figura 20. En la tabla 10 se muestra la temperatura para cada volumen de control, para realizar una comparación de los valores con la tabla 6 se colocan en cada fila 6 volúmenes de control; el máximo error relativo porcentual obtenido es 0.006 %. En la figura 20 se presenta la gráfica generada por Paraview al leer el archivo generado por la ejecución del programa.

Tabla 10: Valor del campo de temperatura para el problema propuesto al usar la clase diffgeneral.

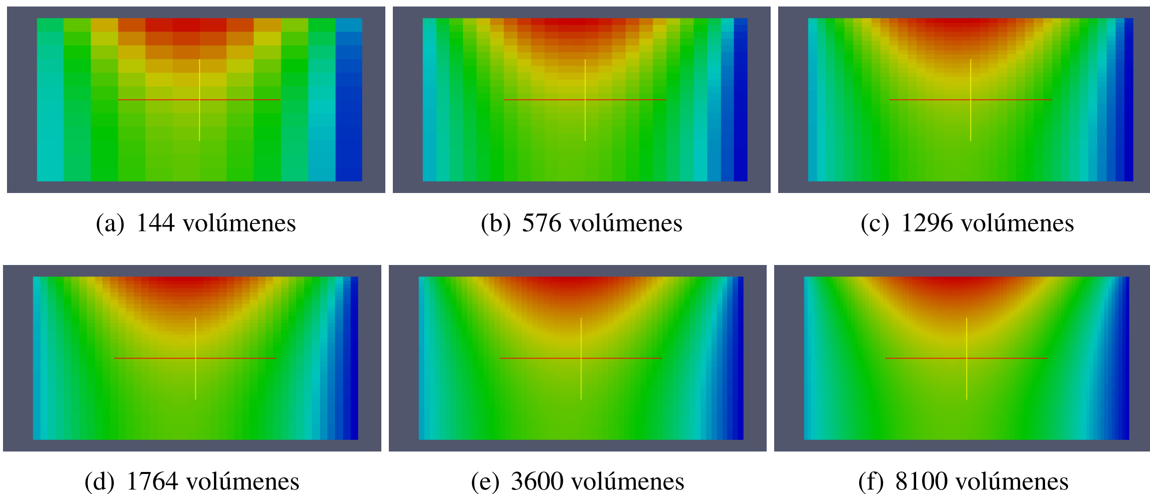
Volumen	1	2	3	4	5	6
Valor	182.156	228.99	249.991	241.658	203.99	140.488
Volumen	7	8	9	10	11	12
Valor	182.729	230.315	251.632	243.299	205.315	141.061
Volumen	13	14	15	16	17	18
Valor	183.911	233.006	254.943	246.61	208.006	142.243
Volumen	19	20	21	22	23	24
Valor	185.778	237.141	259.981	251.647	212.141	144.11
Volumen	25	26	27	28	29	30
Valor	188.463	242.844	266.825	258.492	217.844	146.795
Volumen	31	32	33	34	35	36
Valor	192.176	250.284	275.578	267.245	225.284	150.508
Volumen	37	38	39	40	41	42
Valor	197.251	259.677	286.36	278.027	234.677	155.584
Volumen	43	44	45	46	47	48
Valor	204.213	271.289	299.294	290.961	246.289	162.545
Volumen	49	50	51	52	53	54
Valor	213.897	285.417	314.5	306.167	260.417	172.229
Volumen	55	56	57	58	59	60
Valor	227.645	302.366	332.075	323.742	277.366	185.977

Figura 20: Presentación de los resultados para la clase diffgeneral.



3.2 RESULTADOS

Figura 21: Variación del número de volúmenes de control para diffgeneral.



(a) 144 volúmenes (b) 576 volúmenes (c) 1296 volúmenes

(d) 1764 volúmenes (e) 3600 volúmenes (f) 8100 volúmenes

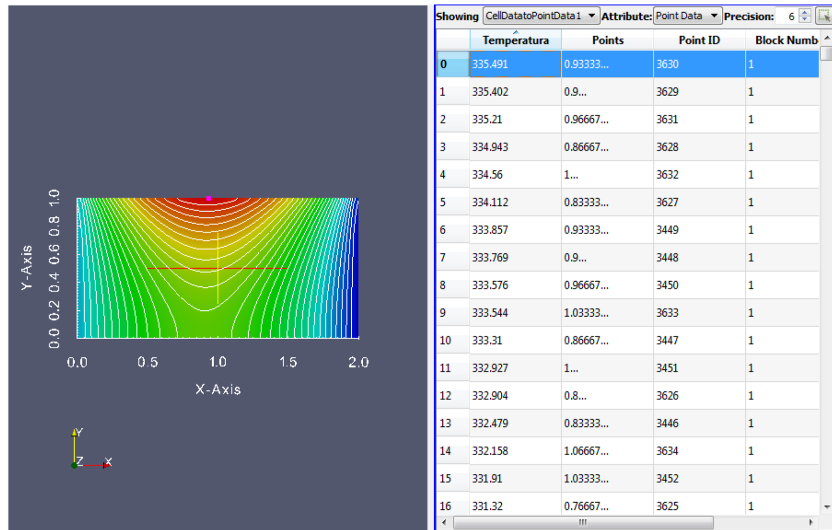
En la figura 21 se muestran los resultados del problema planteado utilizando una instancia de `diffgeneral` para varios números de volúmenes de control. Como ejemplo, para obtener la gráfica de 3600 volúmenes de control es necesario modificar las divisiones en el archivo `datos_difusion.parts`:

```
>nsd = 2;
>no_de_superels = 1;

>SE;>d=2;>e=ElmB4n2D;>div=[60,60];>grad=[1,1];
```

Paraview ofrece varios filtros que pueden ser muy útiles para analizar los datos encontrados. Para el problema resuelto, se muestran las líneas isotérmicas (Figura 22) junto con la temperatura máxima en la placa. Para lograr esto, se convirtieron los datos de celda a datos de punto con `Cell Data to Point Data` y se aplicó `Contour`, para visualizar el punto de máxima temperatura se reordenaron los datos de mayor a menor temperatura en `Spreadsheet View`.

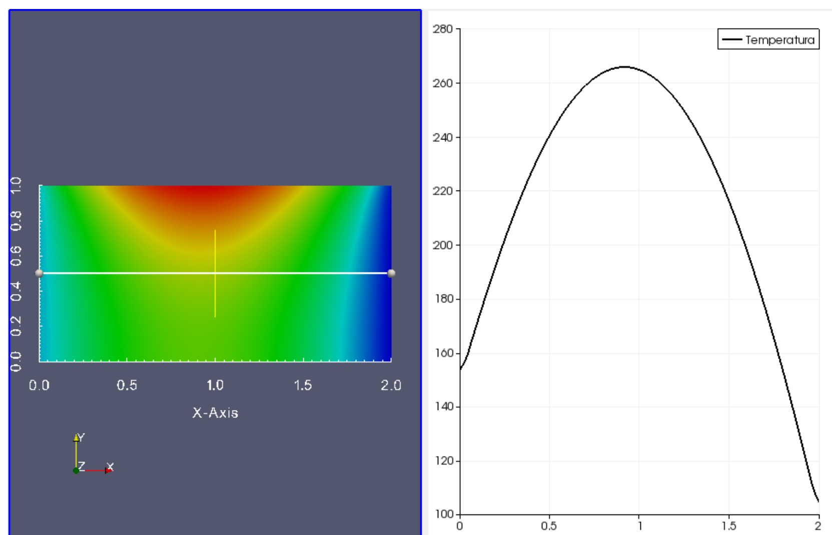
Figura 22: Líneas isotérmicas para el problema de la difusión.



	Temperatura	Points	Point ID	Block Numb
0	335.491	0.93333...	3630	1
1	335.402	0.9...	3629	1
2	335.21	0.96667...	3631	1
3	334.943	0.86667...	3628	1
4	334.56	1...	3632	1
5	334.112	0.83333...	3627	1
6	333.857	0.93333...	3449	1
7	333.769	0.9...	3448	1
8	333.576	0.96667...	3450	1
9	333.544	1.03333...	3633	1
10	333.31	0.86667...	3447	1
11	332.927	1...	3451	1
12	332.904	0.8...	3626	1
13	332.479	0.83333...	3446	1
14	332.158	1.06667...	3634	1
15	331.91	1.03333...	3452	1
16	331.32	0.76667...	3625	1

También se muestra en la figura 23 la distribución de temperatura a lo largo del eje x mediante la utilización del comando `Plot Over Line`.

Figura 23: Temperatura a través de una trayectoria específica.



4. FLUJO DIFUSIVO-CONVECTIVO

En este capítulo se tratará el problema del flujo difusivo-convectivo que se muestra en (Ec 4.1) asumiendo la condición de estado estable en la ecuación diferencial general de la mecánica de fluidos (Ec 1.5).

$$\nabla \cdot (\rho \vec{V} \phi) = \nabla \cdot (\Gamma \nabla \phi) + S_\phi \qquad \text{(Ec 4.1)}$$

En esta ecuación se tiene en cuenta el movimiento del fluido, además, se asumirá que se conoce el perfil de velocidad del fluido por lo que se debe hallar solamente el valor de la propiedad ϕ en cada volumen de control. De nuevo, por la familiaridad en el concepto de la temperatura, se tomará $\phi = T$ pero el proceso que se muestra puede ser generalizado a cualquier variable ϕ que cumpla la (Ec 4.1).

Se reescribe la (Ec 4.1) en términos de la temperatura (Ec 4.2).

$$\nabla \cdot (\rho \vec{V} T) = \nabla \cdot \left(\frac{k}{C_p} \nabla T\right) + \frac{q}{C_p} \qquad \text{(Ec 4.2)}$$

Se realiza la integración de la ecuación diferencial a través de un volumen de control Ω.

$$\iiint_\Omega \nabla \cdot (\rho T \vec{V})\ dV = \iiint_\Omega \nabla \cdot \left(\frac{k}{C_p} \nabla T\right)\ dV + \iiint_\Omega \frac{q}{C_p}\ dV$$

Se aplica el teorema de la divergencia (Ec 2.4):

$$\iint_S \rho T (\vec{V} \cdot \hat{n})\ dA = \iint_S \frac{k}{C_p} (\nabla T \cdot \hat{n})\ dA + \iiint_\Omega \frac{q}{C_p}\ dV \qquad \text{(Ec 4.3)}$$

La (Ec 4.3) debe ser discretizada para cada volumen de control. Ya se ha presentado la discretización del término difusivo (Ec 3.7) y del término fuente (Ec 3.6). Ahora se muestra la discretización del término convectivo, el cual se transforma en la sumatoria mostrada en (Ec 4.4) en donde V_{nl} es la velocidad normal al lado l.

$$\iint_S \rho T (\vec{V} \cdot \hat{n})\ dA \approx \sum_{l=1}^{n} [\rho_l\, T_l\, V_{n_l}\, S_l] \qquad \text{(Ec 4.4)}$$

Se reemplaza en la (Ec 4.3) las ecuaciones (Ec 3.6), (Ec 3.7) y (Ec 4.4) obteniendo la ecuación discretizada de la convección-difusión en estado estable (Ec 4.5).

$$\sum_{l=1}^{n} [\rho_l\, T_l\, V_{n_l}\, S_l] = \sum_{l=1}^{n} \left[\frac{k_l}{C_p}\, S_l\, \frac{T_{vecino} - T_p}{\delta_l}\right] + \frac{q}{C_p}\ \Omega_p \qquad \text{(Ec 4.5)}$$

4.1 ESQUEMA DE APROXIMACIÓN

Se definen dos variables que permiten ver más claramente el efecto convectivo y el efecto difusivo en la (Ec 4.5).

$$F = \rho * V_n * S \qquad D = \frac{\Gamma\, S}{\delta} = \frac{(k/C_p)\, S}{\delta} \tag{Ec 4.6}$$

Se debe tener en cuenta en (Ec 4.6) que V_n será positiva o negativa si el flujo está respectivamente entrando o saliendo del volumen de control, por tal motivo, F puede ser positivo o negativo. Por otro lado, D siempre será positivo. Se reemplazan las definiciones de (Ec 4.6) en (Ec 4.5) y se obtiene (Ec 4.7):

$$\sum_{l=1}^{n}[F_l\ T_l] = \sum_{l=1}^{n}[D_l\ (T_{vecino} - T_p)] + q\ \ \Omega_p \tag{Ec 4.7}$$

Dependiendo del esquema que se use, el valor de T_l cambiará. Para determinar la dependencia del nodo actual o el nodo vecino se debe definir el número adimensional de Peclet (Ec 4.8) que relaciona el efecto de la convección con el efecto de la difusión.

$$Pe = \frac{F}{D} \tag{Ec 4.8}$$

4.1.1 Interpolación aguas-arriba

Para asignar el valor de T_l se debe tener en cuenta la definición general del esquema UDS (Ec 2.6) que, se reescribe en términos de F:

$$T_l = \begin{cases} T_P & si \quad F \geq 0 \\ T_{vecino} & si \quad F < 0 \end{cases} \tag{Ec 4.9}$$

Para cumplir la (Ec 4.9) se hace uso de la función máximo como se muestra en (Ec 4.10).

$$F_l\ T_l = \ máx(\ 0,\ F_l\ )T_P - \ máx(\ 0,\ -F_l\ )T_{vecino} \tag{Ec 4.10}$$

La siguiente ecuación es el resultado de aplicar (Ec 4.10) y reordenar los términos en función de T_P y T_{vecino}:

$$\sum_{l=1}^{n}[(D_l + \ máx(\ 0,\ F_l\ ))\ T_P] - \sum_{l=1}^{n}[(D_l + \ máx(\ 0,\ -F_l\ ))\ T_{vecino}] = q\ \ \Omega_p$$

Para un procedimiento más general se incluirá la función $A(Pe, UDS) = 1$ que permite una fácil generalización a otros métodos como se muestra en la tabla 11 . La ecuación a aplicar a cada volumen de control que no tenga definidas condiciones de frontera mediante el esquema UDS es (Ec 4.11):

$$\sum_{l=1}^{n}[(D_l * A(Pe_l, UDS) + máx(0, F_l))\ T_P] - \sum_{l=1}^{n}[(D_l * A(Pe_l, UDS) + máx(0, -F_l))\ T_{vecino}] = q\ \Omega_p \tag{Ec 4.11}$$

Las correcciones a la (Ec 4.11) para las condiciones de frontera son:

1. Para la condición de frontera tipo Dirichlet (se conoce la temperatura en la frontera) se aplica (Ec 4.12) en donde el término *bd* es el número de condiciones de frontera tipo Dirichlet existentes.

$$\sum_{l=1}^{n}[(D_l*A(Pe_l,UDS)+máx(0,F_l))\,T_P]-\sum_{l=1}^{n-bd}[(D_l*A(Pe_l,UDS)+máx(0,-F_l))\,T_{vecino}]=$$
$$q\ \Omega_p+\sum_{d=1}^{bd}[(D_d*A(Pe_d,UDS)+máx(0,-F_d))\,T_{wall,d}]$$

(Ec 4.12)

2. Para la condición de frontera tipo Neumann (se conoce la derivada parcial en la superficie) se debe reemplazar el valor directamente en (Ec 4.3), finalmente se obtiene (Ec 4.13) en donde *bn* es el número de fronteras tipo Neumann existentes:

$$\left[\sum_{l=1}^{n-bn}((D_l*A(Pe_l,UDS)+máx(0,F_l))+\sum_{l=1}^{bn}F_l\right]T_P$$
$$-\sum_{l=1}^{n-bn}[(D_l*A(Pe_l,UDS)+máx(0,-F_l))T_{vecino}]=q\ \Omega_p+\sum_{h=1}^{bn}[k_h\,S_h\,(\tfrac{\partial T}{\partial n})_{wall,h}]$$

(Ec 4.13)

4.1.2 Otros esquemas para la ecuación de difusión-convección

Se hace uso de la función $A(Pe,''\text{metodo}'')$ para adaptar fácilmente el método de análisis a diferentes tipos de esquemas, el valor de $A(Pe,''\text{metodo}'')$ se muestra en la tabla 11 y se reemplaza en la (Ec 4.11), (Ec 4.12) y (Ec 4.13).

Tabla 11: Función A(Pe)

Equema	Funcion A(Pe)
Upwind (UDS)	1
Diferencia centrada (CDS)	$1-0.5\|Pe\|$
Hibrido	$max(0,1-0.5\|Pe\|)$
Power law	$max(0,(1-0.1\|Pe\|^5))$
Exponencial	$\frac{\|Pe\|}{e^{\|Pe\|}-1}$

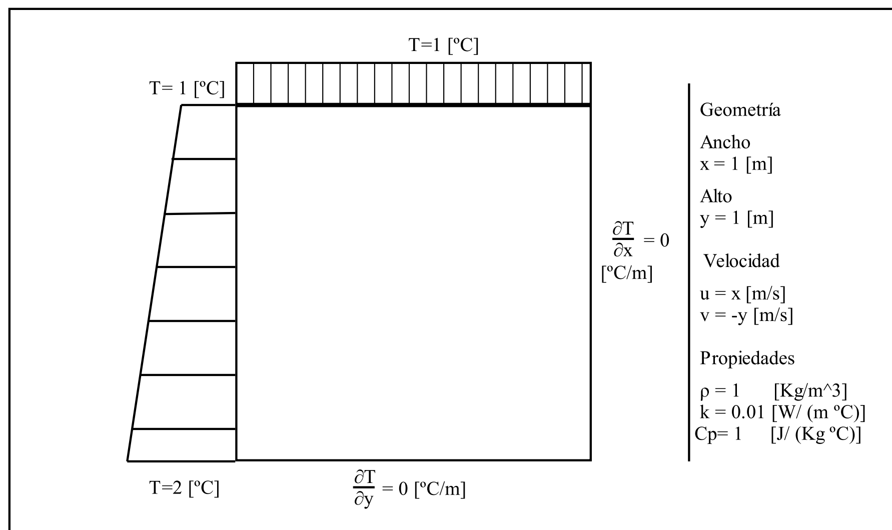
El esquema UDS de primer orden presenta la desventaja de generar una falsa difusión, es decir, el método suaviza el gradiente del campo de temperatura calculado, este proceso se evidencia más cuando el flujo no es paralelo a la malla. El método CDS de $\boldsymbol{O}(\Delta x^2)$ es inestable para $|Pe|>2$ pero para menores valores de Peclet el error de la solución es menor que el método UDS. El esquema híbrido aprovecha la ventaja de la estabilidad del método UDS para altos valores de Peclet y la precisión del método CDS para pequeños valores de Peclet. Los esquemas power law y exponencial proporcionan soluciones más precisas para el caso

unidimensional pero, para situaciones de 2D y 3D no son tan exactos además de su alto costo computacional para calcular los coeficientes.

4.2 IMPLEMENTACIÓN

La implementación del código de difusión-convección se pondrá a prueba en una modificación del problema de Ferziger[1]. El problema de la figura 24 que utiliza condiciones de frontera tipo Neumann y Dirichlet resulta conveniente para verificar la correcta codificación de la condición de frontera tipo Dirichlet debido a que el flujo en la dirección normal a la superficie (en la frontera norte) es diferente de cero.

Figura 24: Problema ejemplo para la difusión-convección



Para desarrollar el código de computador del problema propuesto, se debe definir una ecuación que tenga en cuenta los dos tipos de condiciones de frontera de forma general. La ecuación de difusión-convección se puede definir como:

$$a_p * T_p - \sum_{e=1}^{n-nb} [a_e * T_e] = b \qquad \text{(Ec 4.14)}$$

En donde

$$a_p = \sum_{e=1}^{n-nb} [a_e + F_e] + \sum_{j=1}^{nb} [a_{b,j}] \qquad \text{(Ec 4.15a)}$$

$$a_e = D_e * A(Pe_e, \text{"metodo"}) + m\acute{a}x(0, -F_e) \qquad \text{(Ec 4.15b)}$$

[1]FERZIGER, J.H y PERIC, M. Computational Methods for Fluids Dynamics, 3 ed. Berlín: Springer,2002. p. 83.

$$a_{b,j} = \begin{cases} D_j * A(Pe_j, \text{"metodo"}) + máx(F_j, 0.0) & \text{si es} \quad \text{Frontera tipo Dirichlet} \\ F_j & \text{si es} \quad \text{Frontera tipo Neumann} \end{cases}$$
(Ec 4.15c)

$$b = b_{gen} + \sum_{j=1}^{nb} [b_{b,j}] \quad \text{(Ec 4.15d)}$$

$$b_{gen} = (\frac{q}{C_p}) * \Omega_p \quad \text{(Ec 4.15e)}$$

$$b_{b,j} = \begin{cases} (D_j * A(Pe_j, \text{"metodo"}) + máx(-F_j, 0.0)) * T_{wall} & \text{si es} \quad \text{Frontera tipo Dirichlet} \\ (\frac{\partial T}{\partial n})_{wall} * S_j * k & \text{si es} \quad \text{Frontera tipo Neumann} \end{cases}$$
(Ec 4.15f)

Se crea la clase `convdiff` para solucionar el problema. La declaración de la clase `convdiff`, las variables y los métodos son muy similares a la clase `diffgeneral` por lo que no se hará énfasis en algunos elementos. De forma general, se realizan los siguientes pasos:

1. Declarar la clase `convdiff` y agregar las variables.
2. Agregar los métodos a la clase.
3. Definición de las funciones.
4. Crear un objeto de la clase.

Paso 1. Declarar la clase convdiff y agregar las variables: la clase `convdiff` es una clase derivada de `GuardarEnsight` por lo que se declara de la siguiente forma:

```
#pragma once
#include <MallaFV.h>
#include <ConstOLeemalla.h>
#include <AdmonEcLin.h>
#include <GuardarEnsight.h>
#include <GradoLibertadFV.h>

class convdiff:public GuardarEnsight
{
real longx,longy ;
int nelementos ;
Puntero<MallaFV> malla;
Vector_basico <PunteroElmDefs> elems;
VecinoFVM  vecino;
Puntero<CamposFVM>  campo_vel;
Puntero<CampoFVM>     Temperatura;
```

```
Puntero<GradoLibertadFV> gdl;
Matriz<real>    matriz_a;
Vector<real>    b;
Vector<real>    x;
Vector<real>    u;
Vector<real>    v;
Vector<real>    k;
Vector<real>    rho;
Vector<real>    C_p;
real pe_min ;
real pe_max ;
Puntero<AdmonEcLin> admlin;
prm_Matriz <real> pm;
bool  calculado;
bool  leido;
bool  solucionado;
bool  primer_peclet;
...
```

Se destaca, entre otras variables, la creación de `campo_vel` de tipo `Puntero<CamposFVM>` en la cual se almacena la información a ser exportada del campo vectorial de la velocidad. Se define la densidad del material `rho`, la velocidad `u` en dirección x y la velocidad `v` en dirección y. Además se definen `pe_min` y `pe_max` de tipo `real` para almacenar el valor del peclet mínimo y el peclet máximo, el parámetro adimensional peclet es muy importante en el análisis de la convección-difusión. Se define el parámetro de control `primer_peclet` para ayudar en el cálculo del peclet mínimo y máximo.

Paso 2. Agregar los métodos a la clase. Además de definir el constructor `convdiff(void)`, el destructor `˜convdiff(void)`, el método de lectura de datos `Leer (Cadena problema, Cadena solucionador)`, el método de cálculo de coeficientes `Calcular()`, el método para resolver las ecuaciones `Resolver()` y el reporte de resultados a Paraview `Reportarresultados()`, se definen las funciones `difusion(...)`, `conveccion(...)`, `peclet(...)`, `peclet_min()`, `peclet_max()` y `funcion_a(...)`.

```
...
public:
convdiff(void) {primer_peclet=true; calculado=false; leido=false;}
˜convdiff(void);
int Leer (Cadena problema, Cadena solucionador);
int Calcular();
bool Resolver();
real funcion_a ( real pe, Cadena modo);
real difusion(int vol1, int dir1, int vol2, int dir2, real area);
real difusion(int vol1, int dir1, real area);
```

```
real difusion(int vol1, int dir1, real area, real conveccion_f);
real conveccion( int vol1, int dir1 ,real area, Vector_xyz<real> Vel);
virtual real q (real x, real y);
real peclet(real F, real D);
real peclet_min(void);
real peclet_max(void);
virtual void Reportarresultados();
};
```

La función `conveccion(...)` permite calcular el valor de F en (Ec 4.6); el valor de D en dicha ecuación es calculado mediante una de las sobrecargas de la función `difusion(...)`. La función $A(Pe,''metodo'')$ de la tabla 11 se calcula mediante `funcion_a(...)`. Se agrega la función `peclet(...)` que además de calcular el valor del peclet, calcula el valor del peclet mínimo y el peclet máximo. Finalmente, se incluyen la funciones `peclet_min()` y `peclet_max()` para tener acceso al valor del peclet fuera del objeto.

Paso 3. Definición de las funciones. Para resolver el problema de la difusión-convección propuesto se especificó el siguiente procedimiento en cada una de las funciones:

1. Función `Leer`: esta función toma como argumentos dos cadenas de caracteres que contienen las direcciones de archivo de los parámetros del problema y los parámetros del método de solución:

```
int convdiff::Leer(Cadena problema, Cadena solucionador)
{
...
```

Se leen los parámetros de la malla y se calcula la conectividad entre los volúmenes de la misma forma como se explicó en el capítulo anterior.[2]

```
...
static const char *mallador_tb[] = {"mallador", NULL};
Cadena cadmalla;
Is is(problema);
is->obtComando(cadmalla,mallador_tb);
malla.vincular(new MallaFV());
ConstOLeeMalla(malla(),cadmalla);
vecino.iniciar(malla());
malla->CalcConectividad(vecino);
...
```

Para asignar el valor de la temperatura en la frontera oeste, que es una función de la posición, se debe calcular el tamaño de la región a trabajar para realizar un correcto escalado del valor de la temperatura.

[2]Ver sección 3.1.2

```
...
nelementos=malla->obtNoElem();
Vector_xyz <real> min,max;
malla->obtCoorMinMax(min,max);
longx=max(1)-min(1);
longy=max(2)-min(2);
...
```

Se dimensionan y asignan valores a los vectores y las matrices a usar.

```
...
matriz_a.chaSize(nelementos, nelementos); matriz_a.llenar(0.0);
b.chaSize(nelementos); b.llenar(0.0);
x.chaSize(nelementos); x.llenar(1.0);
u.chaSize (nelementos);
v.chaSize (nelementos);
k.chaSize(nelementos); k.llenar(0.01);
rho.chaSize(nelementos); rho.llenar(1.0);
C_p.chaSize(nelementos); C_p.llenar(1.0);
...
```

Además, se asigna la memoria a los punteros de `CampoFVM` y `CamposFVM`.

```
...
Temperatura.vincular(new CampoFVM(malla(),"Temperatura"));
campo_vel.vincular(new CamposFVM(malla(),"Velocidad"));
gdl.vincular(new GradoLibertadFV(malla(),1));
...
```

Se calculan los parámetros geométricos y se utiliza el bucle repetitivo para asignar la velocidad a cada volumen de control.

```
...
elems.chaSize(nelementos);
for (int i=1; i<=nelementos;i++)
{
elems(i).refill( malla->obtTipoElem(i), malla->obtNoDimEspacio() );
elems(i)->CalcGeomParameters( malla->obtCoorElem(i) );
u(i) =  elems(i)->centroid(malla->obtCoorElem(i))(1);
v(i) = -elems(i)->centroid(malla->obtCoorElem(i))(2);
}
...
```

Se inicializa el administrador de ecuaciones lineales como se había comentado en la sección 3.1.2.

```
...
Is solucion(solucionador);
pm.Leer ( solucion );
admlin.vincular(new AdmonEcLin(EXTERNAL_STORAGE));
admlin().Leer(solucion);
leido=true;
return 1;
}
```

2. Función difusión y sobrecargas: hay tres casos en donde se presenta la difusión.

I. Difusión entre dos volúmenes de control: se toman como parámetros los números que identifican a los dos volúmenes de control y la dirección en la cual se encuentra la superficie en común. Se realiza el cálculo de D asumiendo que cada volumen de control tiene una conductividad térmica constante.

```
real convdiff::difusion(int vol1,int dir1,int vol2,int dir2,real area)
{
real D1, D2, Dt, gamma;
gamma = k(vol1)/C_p(vol1);  D1= gamma / elems(vol1)->getDelta(dir1);
gamma = k(vol2)/C_p(vol2);  D2= gamma / elems(vol2)->getDelta(dir2);
Dt   = area / (1.0/D1 + 1.0/D2);
return Dt;
}
```

II. Difusión en un lado de un volumen de control que limita con una frontera tipo Dirichlet: se calcula el valor de D existente entre el nodo del volumen de control y la superficie definida con la condición de frontera tipo Dirichlet.

```
real convdiff::difusion(int vol1, int dir1, real area)
{
real Dt, gamma  ;
gamma = k(vol1)/C_p(vol1) ;
Dt = gamma / elems( vol1  )->getDelta(dir1);
Dt = area *Dt;
return Dt;
}
```

III. Difusión en un lado de un volumen de control que limita con una frontera en la cual se define la convección del fluido y la temperatura del fluido[3]: en este caso se presenta la

[3]Este caso solo se presenta cuando la variable ϕ de la ecuación general de la mecánica de fluidos se aplica a la temperatura.

difusión dentro de un volumen de control junto a la convección de un fluido en la frontera, el flujo de calor en estos dos fenómenos es el mismo por lo que se calcula la conductancia equivalente.

```
real convdiff::difusion(int vol1,int dir1,real area,real conveccion_f)
{
real D1, D2, Dt, gamma;
gamma = k(vol1)/C_p(vol1)    ;
D1 = gamma / elems( vol1  )->getDelta(dir1);
D2 = conveccion_f;
Dt = area / (1.0/D1 + 1.0/D2);
return Dt;
}
```

3. Función `q`: en el ejemplo presentado no existe generación de calor por lo que esta función devuelve un valor 0.

```
real convdiff::q(real x, real y) { return 0; }
```

4. Función `peclet`: además de calcular el valor del peclet, se actualiza el valor mínimo y el valor máximo de esta variable. A modo de ejemplo, el esquema "CDS" es inestable para números $Pe > 2$.

```
real convdiff::peclet(real F, real D)
{
real peclet=F/D;
if ( primer_peclet )
{ pe_max = pe_min = abs(peclet); primer_peclet=false; }
if ( abs(peclet) > pe_max ) pe_max=abs(peclet);
if ( abs(peclet) < pe_min ) pe_min=abs(peclet);
return peclet;
}
```

5. Función `funcion_a`: se calcula el valor de acuerdo a la tabla 11.

```
real convdiff::funcion_a ( real pe, Cadena modo)
{
if (modo.contiene("upwind") || modo.contiene("Upwind")
|| modo.contiene("UDS"))
{  return 1;  }
if (modo.contiene("CDS"))
{  return 1-0.5*fabs(pe);  }
if ( modo.contiene("hybrid") || modo.contiene("hibrido") )
{ return max(0.0,1-0.5*fabs(pe));  }
```

```
if ( modo.contiene("power") || modo.contiene("law") )
{ return max(0.0,pow(1-0.1*fabs(pe),5)); }
if ( modo.contiene("exp") || modo.contiene("Exp") )
{ return fabs(pe)/(exp(fabs(pe))-1.0); }
return 0;
}
```

6. Función `peclet_min` y `peclet_max`: debido a que las variables `pe_min` y `pe_max` no son declaradas como públicas, se debe crear una función de acceso a su valor.

```
real convdiff::peclet_min(void) {return pe_min;}
real convdiff::peclet_max(void) {return pe_max;}
```

7. Función `Calcular`: se realiza la asignación a los coeficientes de la matriz `matriz_a` y al vector de términos independientes `b`. Para realizar este proceso, se requieren los siguientes pasos.

   I. Verificación del parámetro de control: solo se deben calcular los coeficientes después de haber establecido las condiciones del problema a desarrollar. También se debe verificar que el proceso no sea ejecutado múltiples veces y de esta forma evitar posibles errores durante la ejecución del método.

```
int convdiff::Calcular(void)
{
if (leido==false)
{std_o<<"ERROR: Calcular NO SE EJECUTO EL PROCESO CORRECTAMENTE";
return 0;
}
if (calculado)
{
std_o<<"Ya se habia ejecutado la funcion Calcular: "
<<"No se realizan los calculos\n"; return 0;
}
...
```

8. Declaración de las variables del método: se definen todas las variables necesarias para realizar los cálculos.

```
...
int e, h, l;
int veci; int conx;
Cadena boname, bovalue;
int nbi = malla->obtNoIndFront ();
```

```
double valor;
real F, D, Pe, a_e, a_b, b_b,  A;
real area, y, volumen;
Vector_xyz<real> vel(2), pos;
...
```

9. Proceso de asignación: está compuesto de los siguientes pasos.

I. Para cada volumen de control, se calcula la generación de calor y se obtiene el número de lados.

```
...
for(e=1;e<=nelementos;e++)
{
 pos=elems(e)->centroid(malla->obtCoorElem(e));
 volumen=elems(e)->getVolume();
 b(e) = q (pos(1), pos(2)) * volumen;
 conx = elems(e)->GetNumOfConex();
...
```

II. Para cada lado se verifica si el elemento lateral es una condición de frontera o un volumen de control.

```
...
 for( l=1 ; l<=conx ; l++ )
 {
  veci = malla->getConx(e,l);
  area = elems(e)->getArea(l);
...
```

III. Si no es una condición de frontera, se calcula la velocidad, F, D, Pe y se asignan los coeficientes.

```
...
  if (veci!=0)   {
   for ( h=1 ; h<=conx ; h++ )
   { if(malla->getConx(veci,h)==e)   break; }
   vel(1) = ( u(e) + u(veci) )/2.0;
   vel(2) = ( v(e) + v(veci) )/2.0;
   F  = conveccion(e,l,area,vel);
   D  = difusion( e, l, veci, h, area);
   Pe = peclet( F,D );
   A  = funcion_a(Pe, "upwind");
   a_e = D * A + std::max( 0.0 , -F );
   matriz_a(e,e   ) +=   a_e + F;
   matriz_a(e,veci)  =  -a_e;
```

```
  }
...
```

IV. Si es una condición de frontera, se realiza el proceso de asignación dependiendo del tipo de frontera.

```
...
  else {
   for (h=1; h<=nbi; h++) {
    if (  malla->LadoFront(e,l,h,elems(e)())    ) {
     vel(1) =  elems(e)->centroArea(malla->obtCoorElem(e),l)(1) ;
     vel(2) = -elems(e)->centroArea(malla->obtCoorElem(e),l)(2) ;
     F      = conveccion(e,l,area,vel) ;
     boname  = malla->obtNombreIndFront(h);
     bovalue = boname.despues('=');
     valor   = atof(bovalue.carts());
     D     = difusion(e,l,area);
     if ( boname.contiene("T") || boname.contiene("dirichlet")
      || boname.contiene("Dirichlet") )
     {
      if( boname.contiene("L") ) {
       y     = elems(e)->centroArea(malla->obtCoorElem(e),l)(2);
       valor = valor * ( 2.0 - y/longy ); }
      Pe = peclet( F,D );
      A  = funcion_a(Pe, "upwind");
      a_b =  D*A + std::max(F,0.0);
      b_b = (D*A + std::max(-F,0.0))*valor;
     }
     if ( boname.contiene("Q") ) {
      a_b = F;
      b_b = valor * area/C_p(e); }
     if ( boname.contiene("Neumann")||boname.contiene("neumann") ) {
      a_b = F;
      b_b = valor * area * k(e)/C_p(e); }
     matriz_a (e,e) += a_b;
     b (e) += b_b;
}}}}}
```

V. Actualización del parámetro de control: se actualiza el valor del parámetro de control y se devuelve el valor 1 que indica que el proceso fue realizado satisfactoriamente.

```
...
calculado=true;
return 1;
}
```

10. Función `Resolver`: se verifica el parámetro de control y posteriormente se resuelve el sistema de ecuaciones como se mostró en la sección 3.1.2.

```
bool convdiff::Resolver()
{
if(calculado==false) Calcular();
admlin().adjuntar(matriz_a,x,b);
bool solucion=admlin().solve();
gdl->vector2campo(x,Temperatura());
gdl->vector2campo(u,campo_vel()(1));
gdl->vector2campo(v,campo_vel()(2));
solucionado=true;
return solucion;
}
```

11. Función `Reportarresultados`: se exporta el campo de temperaturas y el campo vectorial de la velocidad.

```
void convdiff::Reportarresultados()
{
volcar(Temperatura(),0);
volcar(campo_vel()  ,0);
}
```

El archivo de encabezado de la clase `convdiff.h` y el archivo de código fuente `condvdiff.cpp` se encuentran, respectivamente en el anexo F y G.

Paso 4. Crear un objeto de la clase. El siguiente código es un ejemplo de aplicación de esta clase al problema presentado:

```
convdiff con;
con.Leer("ARCH=datos_conveccion_difusion.txt", "ARCH=Solve_datos.txt");
con.Calcular();
con.Resolver();
con.Reportarresultados();
```

El archivo `datos_conveccion_difusion.txt` contiene el siguiente texto:

```
> mallador = PREPROCESSOR=ConjMalladorSupElem/
ARCH=datos_conv_dif.geom/ARCH=datos_conv_dif.parts
```

La información del archivo `datos_conv_dif.geom` es la siguiente:

```
> nsd = 2;
> subdominios = 1;
> no_de_superels = 1;
>nbind = 3
>bname T=1 Q=0 TL=1

>SupEl;
>subdominio_no = 1;
>tipoelemento = ElmB4n2D;
>fronteras = [2(1)] [1 (2)] [3 (3)] [2(4)];
>nodos = [1(0 0)]+[2(1 0)]+[3(0 1)]+[4(1 1)];
>lados=;
```

El archivo de las particiones `datos_conv_dif.parts` es:

```
>nsd = 2;
>no_de_superels = 1;

>SE;>d=2;>e=ElmB4n2D;>div=[6,10];>grad=[1,1];
```

El contenido del archivo `Solve_datos.txt` es:

```
Parametros de la matriz (clase prmMatriz);
>guarda  = Matriz;
>guarda_sim = FALSE
>pivoteo = FALSE
>umbral =0.0

prm_solucionEcLin parametros:
>solver_classname=Orthomin
>metodo=R-OM(7)
>cambiar=0.0
>pivoteo=NO_PIVOT
>relajacion=1.9
>startvector=USER_START
>criterio_def=TRUE
>nextern_crit=0
>maxit=3000
>estimate_eigvals=FALSE

Parametros del precondiciondor:
>precondicionador=PrecIdentidad
>left_prec=TRUE
>auto_iniciar=TRUE
>nivel_llenado=0
```

Tabla 12: Distribución del campo de temperatura para el problema propuesto al usar la clase convdiff.

Volumen	1	2	3	4	5	6
Valor	1.54995	1.37608	1.29020	1.23736	1.20121	1.17589
Volumen	7	8	9	10	11	12
Valor	1.43724	1.27334	1.19841	1.15511	1.12698	1.10806
Volumen	13	14	15	16	17	18
Valor	1.32028	1.17380	1.11406	1.08258	1.06364	1.05165
Volumen	19	20	21	22	23	24
Valor	1.22687	1.10449	1.06055	1.03971	1.02824	1.02147
Volumen	25	26	27	28	29	30
Valor	1.15607	1.06051	1.03061	1.01796	1.01164	1.00819
Volumen	31	32	33	34	35	36
Valor	1.10274	1.03331	1.01463	1.00763	1.00448	1.00290
Volumen	37	38	39	40	41	42
Valor	1.06297	1.01690	1.00640	1.00296	1.00157	1.00093
Volumen	43	44	45	46	47	48
Valor	1.03424	1.00749	1.00243	1.00100	1.00048	1.00026
Volumen	49	50	51	52	53	54
Valor	1.01485	1.00260	1.00072	1.00026	1.00012	1.00006
Volumen	55	56	57	58	59	60
Valor	1.00362	1.00051	1.00012	1.00004	1.00002	1.00001

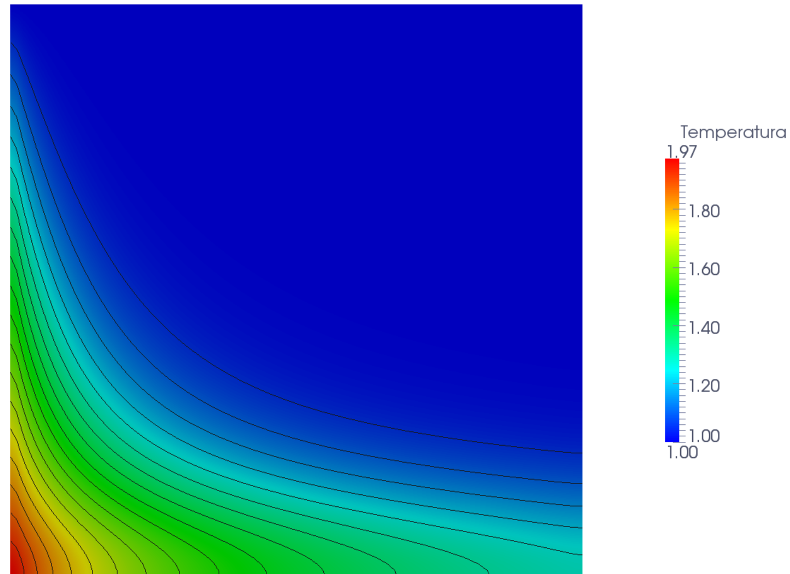
```
>prec_RILU=1
>prec_SSOR=1
>pasos_int=1
```

El vector solución para 60 volúmenes de control se encuentra en la tabla 12.

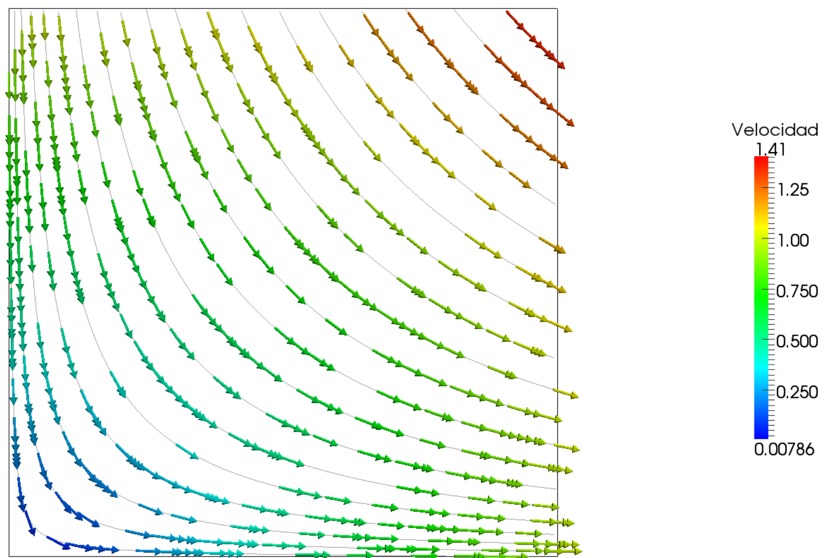
4.3 RESULTADOS

En la figura 25(a) se muestra la distribución de la temperatura (filtro `CellDatatoPointData`), las líneas isotérmicas (filtro `Contour`) y en la figura 25(b) se muestra el campo vectorial de la velocidad (filtro `StreamTracer` y `Glyph`) para 8100 volúmenes de control.

Figura 25: Presentación de los resultados para la clase convdiff, $\Gamma = 0.01$.



(a) Distribución del campo de temperatura.



(b) Campo vectorial de la velocidad.

5. ECUACIÓN TRANSITORIA DE LA DIFUSIÓN Y LA CONVECCIÓN-DIFUSIÓN

Ya se han presentado las ecuaciones de difusión y convección-difusión en estado estable, ahora se complementará la discretización de las dos ecuaciones en estado transitorio.

5.1 DIFUSIÓN EN ESTADO TRANSITORIO

En la (Ec 5.1) se muestra la ecuación a discretizar. Esta ecuación se debe integrar a través de un volumen de control, como se muestra en (Ec 5.2):

$$\frac{\partial(\rho\, T)}{\partial t} = \nabla \cdot (\frac{k}{C_p}\, \nabla T) + \frac{q}{C_p} \tag{Ec 5.1}$$

$$\iiint_{\Omega} \frac{\partial(\rho\, T)}{\partial t}\, dV = \iiint_{\Omega} [\nabla \cdot (\frac{k}{C_p}\, \nabla T)]\ dV + \iiint_{\Omega} \frac{q}{C_p}\ dV \tag{Ec 5.2}$$

En la (Ec 5.2) se tiene el término fuente, el término transitorio y el término difusivo. La discretización del término fuente y el término difusivo se mostró en el capítulo 3. Ahora se presenta la **integral del término transitorio** en (Ec 5.3) en donde el término $\rho\, \Omega_p\, T$ se evalúa en el tiempo futuro $t + \Delta t$ y en el tiempo presente t.

$$\iiint_{\Omega} \frac{\partial(\rho\, T)}{\partial t}\, dV \approx \frac{(\rho\, \Omega_p\, T_p)^{t+\Delta t} - (\rho\, \Omega_p\, T_p)^t}{\Delta t} \tag{Ec 5.3}$$

La integral del término difusivo se puede evaluar en el tiempo futuro, el tiempo actual o como una dependencia de ambos valores. Para definir esta dependencia se introduce la variable θ en (Ec 5.5) y el número discreto de Fourier (Fo) en (Ec 5.4).

$$Fo = \frac{\alpha\, \Delta t}{\delta^2} \tag{Ec 5.4}$$

$$T = \theta\, T^{t+\Delta t} + (1-\theta)\, T^t$$

$$\underbrace{0}_{\text{Depende del tiempo actual}} \leq \theta \leq \underbrace{1}_{\text{Depende del tiempo futuro}} \tag{Ec 5.5}$$

El caso $\theta = 0$ se conoce comúnmente como esquema explícito porque solo se utiliza el valor de la temperatura en el presente (la cual se conoce) para calcular la temperatura en el futuro. El caso $\theta = 1$ se conoce como esquema implícito en donde la temperatura de un volumen de control en el futuro depende de la temperatura de otros volúmenes de control en el futuro. Para el caso $\theta = 1/2$ se tiene el esquema Crank-Nicolson, en este esquema, la temperatura de

un volumen de control es un promedio entre la temperatura en el tiempo actual y el tiempo futuro. En el caso explícito, la matriz de coeficientes es una matriz diagonal por lo que resulta ineficiente su almacenamiento en una matriz y su solución por métodos iterativos, la desventaja de este método radica en la condición de estabilidad, $Fo \leq 1/2$. El caso implícito es incondicionalmente estable aunque se necesita un mayor tiempo para solucionar el sistema de ecuaciones.

Teniendo en cuenta la variable θ, se reemplazan (Ec 3.6), (Ec 3.8) y (Ec 5.3) en (Ec 5.2) y se organiza para obtener los términos a agregar a la matriz de coeficientes:

$$\begin{aligned}[\tfrac{\rho\,\Omega}{\Delta t} + \theta \sum_{i=1}^{n} (U_i\,A_i)] \quad T_p^{t+\Delta t} - \sum_{i=1}^{n} [U_i\,A_i\,\theta\,T_i^{t+\Delta t}] = \\ \tfrac{\rho\,\Omega}{\Delta t}\,(T_p^t) + \sum_{i=1}^{n} \{U_i\,A_i\,[(1-\theta)\,(T_i^t - T_p^t)]\} + \tfrac{q}{C_p}\,\Omega_p\end{aligned} \qquad \text{(Ec 5.6)}$$

5.1.1 Implementación

Antes de explicar el código de computador desarrollado para resolver la ecuación de difusión transitoria se definen algunos coeficientes basados en la (Ec 5.6) para incorporar las condiciones de frontera y tener una ecuación general a aplicar[1].

$$a_p * T_p^{t+\Delta t} - \sum_{i=1}^{n-nb} [\theta * a_i * T_i^{t+\Delta t}] = b \qquad \text{(Ec 5.7)}$$

En donde

$$a_p = a_{tr} + \theta \sum_{i=1}^{n-nb} a_i + \sum_{h=1}^{nb} a_{b,h} \qquad \text{(Ec 5.8a)}$$

$$a_{tr} = \frac{\rho\,\Omega_p}{\Delta t} \qquad \text{(Ec 5.8b)}$$

$$a_i = U_i\,A_i \qquad \text{(Ec 5.8c)}$$

$$U_i = \begin{cases} \frac{k}{C_p\,\delta_i} & \text{si} \quad \frac{k}{C_p} \text{ es constante en toda la malla o se analiza una frontera tipo Dirichlet} \\ \frac{1}{\frac{C_p\,\delta_{ei}}{k_i} + \frac{C_p\,\delta_{pi}}{k_p}} & \text{si} \quad \text{cada volumen de control tiene una } k \text{ constante} \end{cases} \qquad \text{(Ec 5.8d)}$$

$$a_{b,h} = \begin{cases} \frac{k\,A_h}{C_p\,\delta_h}\theta & \text{si es} \quad \text{Frontera tipo Dirichlet} \\ 0 & \text{si es} \quad \text{Frontera tipo Neumann} \end{cases} \qquad \text{(Ec 5.8e)}$$

$$b = b_{gen} + \sum_{h=1}^{nb} b_{b,h} + a_{tr}\,T_p^t + (1-\theta) \sum_{i=1}^{n-nb} [a_i\,(T_i^t - T_p^t)] \qquad \text{(Ec 5.8f)}$$

[1] Se puede aplicar a mallas ortogonales.

$$b_{gen} = \frac{q}{C_p}\,\Omega \quad \text{(Ec 5.8g)}$$

$$b_{b,h} = \begin{cases} \frac{k\,A_h}{C_p\,\delta_h}[-(1-\theta)T_p^t + T_{h(wall)}] & si\ es \quad Frontera\ tipo\ Dirichlet \\ \\ \frac{k}{C_p}\,A_h\,(\frac{\partial T}{\partial n})_{wall} & si\ es \quad Frontera\ tipo\ Neumann \end{cases} \quad \text{(Ec 5.8h)}$$

Para poner a prueba el código de computador que resuelve la ecuación de difusión en estado transitorio se analizará el mismo ejemplo propuesto en la sección 3.1 pero partiendo de una temperatura inicial $T_{inicial} = 100$ $[C]$ con $\rho = 500$ $[Kg/m^3]$ y $C_p = 1$ $[J/(Kg\,C)]$. Se realizan los siguientes pasos para desarrollar el problema planteado:

1. Declarar la clase `DifusionTransitoria` y agregar las variables correspondientes.
2. Agregar los métodos a la clase.
3. Definición de las funciones.
4. Crear un objeto de la clase.

A continuación se explican cada uno de los pasos realizados omitiendo los detalles que han sido aclarados en capítulos anteriores:

Paso 1. Declarar la clase DifusionTransitoria y agregar las variables correspondientes. La clase creada es una clase derivada de `GuardarEnsight` permitiendo un método fácil para exportar los resultados.

```
#pragma once
#include <AdmonEcLin.h>
#include <MallaFV.h>
#include <MallaFE.h>
#include <ConstOLeemalla.h>
#include <GuardarEnsight.h>
#include <GradoLibertadFV.h>
class DifusionTransitoria :  public GuardarEnsight
{
int nelementos;
Puntero<MallaFV> malla;
Vector_basico <PunteroElmDefs> elems;
VecinoFVM  vecino;
Puntero<CampoFVM>     phi;
Puntero<GradoLibertadFV> gdl;
Vector<real>  rho;
Vector<real>  k;
Vector<real>  cp;
Vector<real>  alph;
```

```
Matriz<real>  matriz;
prm_Matriz <real> pm;
Vector<real>b;
Vector<real>x;
Vector<real>x_futuro;
Puntero<AdmonEcLin> admlin;
real theta;
Puntero<prmTiempo> tiempo_p;
bool calculado;
bool leido;
bool solucionado;
bool primera_solucion;
bool primer_fourier;
real phi_inicial;
real Fo_min;
real Fo_max;
...
}
```

Además de las variables utilizadas en la clase `diffgeneral`, se definen:

1. Otras propiedades del material como la densidad `rho`, el calor específico `cp`, la difusividad térmica `alph`.
2. El vector solución en el futuro `x_futuro`.
3. La dependencia del análisis del valor actual o el valor futuro `theta`.
4. Se definen los parámetros de control `primera_solucion` y `primer_fourier` que ayudan en procesos específicos como se explicará posteriormente.
5. Como se comentó anteriormente, para el esquema explícito es muy importante conocer el valor del número discreto de fourier, razón por la cual se almacena el valor absoluto mínimo `Fo_min` y máximo `Fo_max`.

Paso 2. Agregar los métodos a la clase. Se requiere las siguientes funciones para resolver el problema a trabajar.

```
...
class DifusionTransitoria :  public GuardarEnsight
{
...
real Fourier(real alpha, real dt, real dx);
protected:
virtual int CalculoUnTiempo();
```

```
virtual bool SolucionUnTiempo();
virtual void Reportarresultados();
public:
DifusionTransitoria(void);
~DifusionTransitoria(void){}
virtual int Leer (Cadena archivo,Cadena solucionador,
     Cadena def_tiempo, real T_inicial=100);
virtual int CalcularYResolver(real dependencia_futuro);
virtual real S_fuente (real x, real y);
};
```

Una vez se realiza la lectura de datos mediante `Leer(...)` se debe llamar a `CalcularYResolver()`, en donde, se llama a `CalculoUnTiempo()`, `SolucionUnTiempo()` y `Reportarresultados()` definidas como `protected`, se realiza de esta forma para asegurar un avance correcto en el tiempo, el cual se define en `CalcularYResolver(...)`. Además se define `S_fuente` para tener en cuenta el término fuente de la ecuación de difusión.

La definición completa del archivo de encabezado de la clase `DifusionTransitoria.h` se encuentra en el anexo H y la definición de las funciones en el archivo de código fuente se encuentra en el anexo I .

Paso 3. Definición de las funciones. Ahora se muestra la definición de cada una de las funciones que se encuentran en el archivo `DifusionTransitoria.cpp`.

1. Constructor: se asigna el valor inicial a los parámetros de control.

```
DifusionTransitoria::DifusionTransitoria(void)
{
primera_solucion=primer_fourier=true;
calculado = leido = solucionado = false;
}
```

2. Función `Leer(...)`: se incorporan algunas instrucciones adicionales (comparado con la función `Leer` de `diffgeneral`) como el llenado de las propiedades de la placa, `k`, `rho`, `cp`, `alph`.

```
int DifusionTransitoria::Leer(Cadena archivo,
          Cadena solucionador, Cadena def_tiempo, real phi_o)
{
...
rho.chaSize(nelementos); rho.llenar(500.0);
cp.chaSize(nelementos);  cp.llenar(1.0);
alph.chaSize(nelementos);
for (i=1;i<=nelementos;i++)
```

```
  alph(i)=k(i)/(rho(i)*cp(i));
...
```

Adicionalmente, se llena el vector x (tiempo actual) y el vector x_futuro (tiempo futuro) con el valor de la temperatura inicial mediante la llamada a la función llenar de la clase Vector<real>, la temperatura inicial se da en el argumento phi_o de la función Leer.

```
...
x.chaSize(nelementos); x.llenar(phi_o);
x_futuro.chaSize(nelementos);x_futuro.llenar(phi_o);
...
```

La Cadena def_tiempo permite introducir los datos del tiempo para el análisis transitorio o el análisis en estado estable. Para definir el estado transitorio, la cadena debe tener el formato ''dt=6 t en [0,30]'' en donde $\Delta t = 6[s]$, $t_{inicial} = 0[s]$ y $t_{final} = 30[s]$. Por otro lado, para seleccionar el análisis en estado estable, se debe introducir la cadena "dt=0". La cadena def_tiempo es leída por una instancia de Puntero<prmTiempo>, la clase prmTiempo define una serie de funciones para el manejo de los parámetros del tiempo. Teniendo en cuenta esto, se inicializa la memoria del Puntero<prmTiempo> tiempo_p, se llama a la función Leer dando como argumento la cadena def_tiempo y se inicia el ciclo del tiempo con iniciarCicloTiempo().

```
...
tiempo_p.vincular(new prmTiempo);
tiempo_p->Leer(def_tiempo);
tiempo_p->iniciarCicloTiempo();
...
}
```

3. Función CalcularYResolver: en un análisis transitorio, se requiere una gran cantidad de memoria para almacenar el valor de la propiedad calculada en cada uno de los tiempos, es por esta razón que es necesario calcular los coeficientes de la matriz, resolver las ecuaciones y finalmente guardar los datos en un archivo compatible con Paraview.

La función CalcularYResolver toma como único parámetro el valor dependencia_futuro el cual define el esquema a usar en el caso transitorio, en estado estable no se tendrá en cuenta el valor introducido.

I. Verificación de los parámetros de control: se verifica si se ha leído los datos o se había llamado la función anteriormente:

```
int DifusionTransitoria::CalcularYResolver(real dependencia_futuro)
{
if (!leido || calculado) return 0;
...
```

II. Selección del tipo de análisis: el procedimiento es algo diferente para el estado estable y el estado transitorio, por esta razón se hace la distinción del proceso.

A. Análisis en estado estable: es necesario definir `theta=1` para que la llamada de `CalculoUnTiempo()` obtenga los coeficientes correspondientes a un análisis en estado estable. Posteriormente se resuelve el sistema de ecuaciones y se guardan los resultados.

```
...
 if (tiempo_p->estacionario())
 {
  theta=1;
  CalculoUnTiempo();
  SolucionUnTiempo();
  Reportarresultados();
 }
...
```

B. Análisis en estado transitorio: se asigna el valor de `theta` con el valor dado en el argumento de la función, posteriormente, se exportan los resultados de la condición inicial del problema que corresponden al tiempo inicial.

```
...
 else
 {
  theta=dependencia_futuro;
   gdl->vector2campo(x_futuro,phi());
   Reportarresultados();
...
```

Se utiliza un ciclo repetitivo que finaliza cuando se ha llegado o sobrepasado el tiempo final especificado. Dentro del bucle se realiza el incremento del tiempo, se calcula los coeficientes de la matriz, se resuelve el sistema de ecuaciones y se exportan los resultados.

```
...
  do
  {
   tiempo_p->incrementarTiempo();
   CalculoUnTiempo();
   SolucionUnTiempo();
   Reportarresultados();
  }
  while(!tiempo_p->finalizo());
  std_o<<"\nMinimo Fourier "<<Fo_min
       <<"\nMaximo Fourier "<<Fo_max<<"\n";
 }
```

Para el caso transitorio puede ser conveniente mostrar el valor máximo del número

discreto de Fourier como se evidencia en el anterior fragmento de código.

III. Actualización del parámetro de control: se actualiza el valor de `calculado` y `solucionado` y se devuelve el valor `1` para indicar que el proceso se ha realizado satisfactoriamente.

```
calculado=solucionado=true;
return 1;
}
```

4. Función `CalculoUnTiempo()`: el procedimiento de asignación de los coeficientes a la matriz y al vector de términos independientes se realiza en esta función.

I. Declaración de las variables del método: se declaran contadores, variables de tipo `real`, un vector de posición `Vector_xyz<real>` y dos cadenas de texto para el manejo de las condiciones de frontera.

```
int DifusionTransitoria::CalculoUnTiempo()
{
int i , j , l, veci, conx, conxvec;
int h, nbi=malla->obtNoIndFront ();
real aj, vol, tr, hfrontera, bc, u1, u2,  ut, valor;
Vector_xyz  <real> posicion(3);
Cadena boname, bovalue;
...
```

II. Asignación del valor inicial: para el análisis transitorio es importante guardar el valor de la propiedad calculada en el tiempo anterior, también, se asigna el valor a 0 a la matriz de coeficientes y al vector de términos independientes:

```
...
x=x_futuro;
matriz.llenar(0.0);
b.llenar(0.0);
...
```

III. Se realiza un bucle que recorre cada uno de los volúmenes de control del análisis. Se asigna al vector de términos independientes el efecto de la generación representado por b_{gen} en la (Ec 5.8).

```
...
for(i=1;i<=nelementos;i++)
{
posicion = elems(i)->centroid(malla->obtCoorElem(i));
vol = elems(i)->getVolume();
b(i) =  vol * S_fuente( posicion(1),posicion(2) );
```

Se debe verificar el valor devuelto por `tiempo_p->estacionario()` para determinar si el análisis es en estado estable o en estado transitorio, caso en el que se suma el efecto transitorio.

```
if (!tiempo_p->estacionario())
{
tr=rho(i)* vol / tiempo_p->Delta();
matriz( i, i ) += tr;
b(i)  += tr*x(i);
}
```

IV. Se realiza un bucle repetitivo para cada uno de los lados del volumen de control `i`. Se verifica cual es el volumen que se encuentra a un lado específico.

```
...
conx=elems(i)->GetNumOfConex();
for( j=1 ; j<=conx ; j++ )
{
 veci=malla->getConx(i,j);
...
```

A. Si no es una condición de frontera, se calcula la conductancia y se asigna los valores correspondientes al vector de términos independientes y a la matriz de coeficientes. Solo en el caso que el análisis sea transitorio se llama a la función `Fourier(...)` que posteriormente se explicará.

```
...
 if (veci!=0)
 {
  conxvec=elems(veci)->GetNumOfConex();
  for ( l=1 ; l<=conxvec ; l++ )
  {if(malla->getConx(veci,l)==i)   break; }
  u1=k(i)/(cp(i)*elems(i)->getDelta(j));
  u2=k(veci)/(cp(veci)*elems(veci)->getDelta(l));
  ut = 1.0 / ( (1.0/u1)+ (1.0/u2) );
  aj = ut* elems(i)->getArea(j);
  matriz( i,  i  ) += theta*aj;
  matriz( i,veci) =  -theta*aj;
  b(i)-= (1-theta)*aj*(x(i)-x(veci));
  if (!tiempo_p->estacionario())
  {
  Fourier(alph(i),tiempo_p->Delta(),
   elems(i)->getDelta(j)+elems(veci)->getDelta(l));
  }
 }
...
```

B. Si es una condición de frontera, se debe realizar la asignación de acuerdo al tipo de frontera como se presenta en la (Ec 5.8).

```
...
else{
 for (h=1; h<=nbi; h++) {
 if (malla->LadoFront(i,j,h,elems(i)())) {
   boname=malla->obtNombreIndFront(h);
   bovalue=boname.despues('=');
   valor=atof(bovalue.carts());
   if (boname.contiene("T") ||
    boname.contiene("Dirichlet")||
    boname.contiene("dirichlet")
    )
   {if ( boname.contiene("h=") ){
     bovalue=boname.despues("h=");
     hfrontera=bovalue.obtReal();
     bovalue=boname.despues("T=");
     valor=bovalue.obtReal();
     u1= k(i)/(cp(i)*elems(i)->getDelta(j));
     u2= hfrontera/cp(i);
     ut=1.0 / ( (1.0/u1)+ (1.0/u2) );}
    else{ ut = k(i)/(cp(i)*elems(i)->getDelta(j));}
    bc    = ut * elems(i)->getArea(j);
    b(i) += bc * valor ;
    b(i) -= (1-theta) * bc * x(i);
    matriz(i,i) += bc * theta ;}
   else {
    if (boname.contiene("Neumann")||
     boname.contiene("neumann")
     )
    {bc = valor * k(i) * elems(i)->getArea(j)/cp(i);
    b(i) += bc;}
    if (boname.contiene("Q")){
    bc = valor * elems(i)->getArea(j)/cp(i);
    b(i) += bc;     }
  } }}}
...
```

V. Finalmente se devuelve el valor 1 que implica que el proceso se ha realizado satisfactoriamente.

```
...
  }  }
 return 1;
```

```
 }
```

5. Función `Fourier`: se calcula el valor del número discreto de Fourier y se almacena el valor mínimo y el valor máximo.

```
real DifusionTransitoria::Fourier(real alpha, real dt, real dx)
{
 real fo;
 fo = alpha*dt/(dx*dx);
 if (primer_fourier) { Fo_min=Fo_max=fo; primer_fourier=false;}
 if (fo < Fo_min) {Fo_min=fo;}
 if (fo > Fo_max) {Fo_max=fo;}
 return fo;
}
```

6. Función `S_fuente`: se devuelve un valor de tipo `real` que corresponde a la generación de calor del problema a resolver presentado en la sección 3.1.

```
real DifusionTransitoria::S_fuente(real x, real y)
{
return 5000;
}
```

7. Función `SolucionUnTiempo`: para el análisis transitorio, es necesario solucionar el sistema de ecuaciones varias veces, pero, la llamada a `adjuntar` de la clase `AdmonEcLin` solo se debe realizar una vez; esta labor se realiza mediante la verificación del parámetro de control `primera_solucion`.

```
bool DifusionTransitoria::SolucionUnTiempo()
{
 if (primera_solucion)
 {
  admlin().adjuntar(matriz, x_futuro, b);
  primera_solucion=false;
 }
 bool solucion=admlin().solve();
 gdl->vector2campo(x_futuro,phi());
 return solucion;
}
```

8. Función `Reportarresultados`: para exportar los resultados, ya sea en estado estable o transitorio, se llama a la función `GuardarEnsight::volcar(...)` con `tiempo_p.obtPtr()` como segundo argumento, de esta forma, se indica el tiempo al cual corresponde la solución.

```
void DifusionTransitoria::Reportarresultados()
{
GuardarEnsight::volcar(phi(),tiempo_p.obtPtr());
}
```

Paso 4. Crear un objeto de la clase. Como primer paso, se declara una instancia de la clase `DifusionTransitoria`. Posteriormente, se llama a la función `Leer()`con los siguientes parámetros:

1. Nombre del archivo de definición del problema `"ARCH=datos_difusion.txt"`:este archivo fue presentado en la página 79 y contiene el nombre del archivo de geometría y el archivo de las particiones; el archivo de geometría contiene la misma información mostrada en la página 79, el archivo de las particiones fue modificado y contiene la siguiente información:

```
>nsd = 2;
>no_de_superels = 1;

>SE;>d=2;>e=ElmB4n2D;>div=[36,36];>grad=[1,1];
```

2. Nombre del archivo de definición del método de solución `"ARCH=Solve_datos.txt"`: el contenido de este archivo fue presentado en la página 67.

3. Cadena de definición de los parámetros del tiempo: para el análisis en estado estable se utiliza la cadena `''dt=0''` y para el análisis en estado transitorio se usa `"dt=6 t en [0,30]"`.

4. Temperatura inicial: un valor de tipo real que corresponde en este caso a `100.0`.

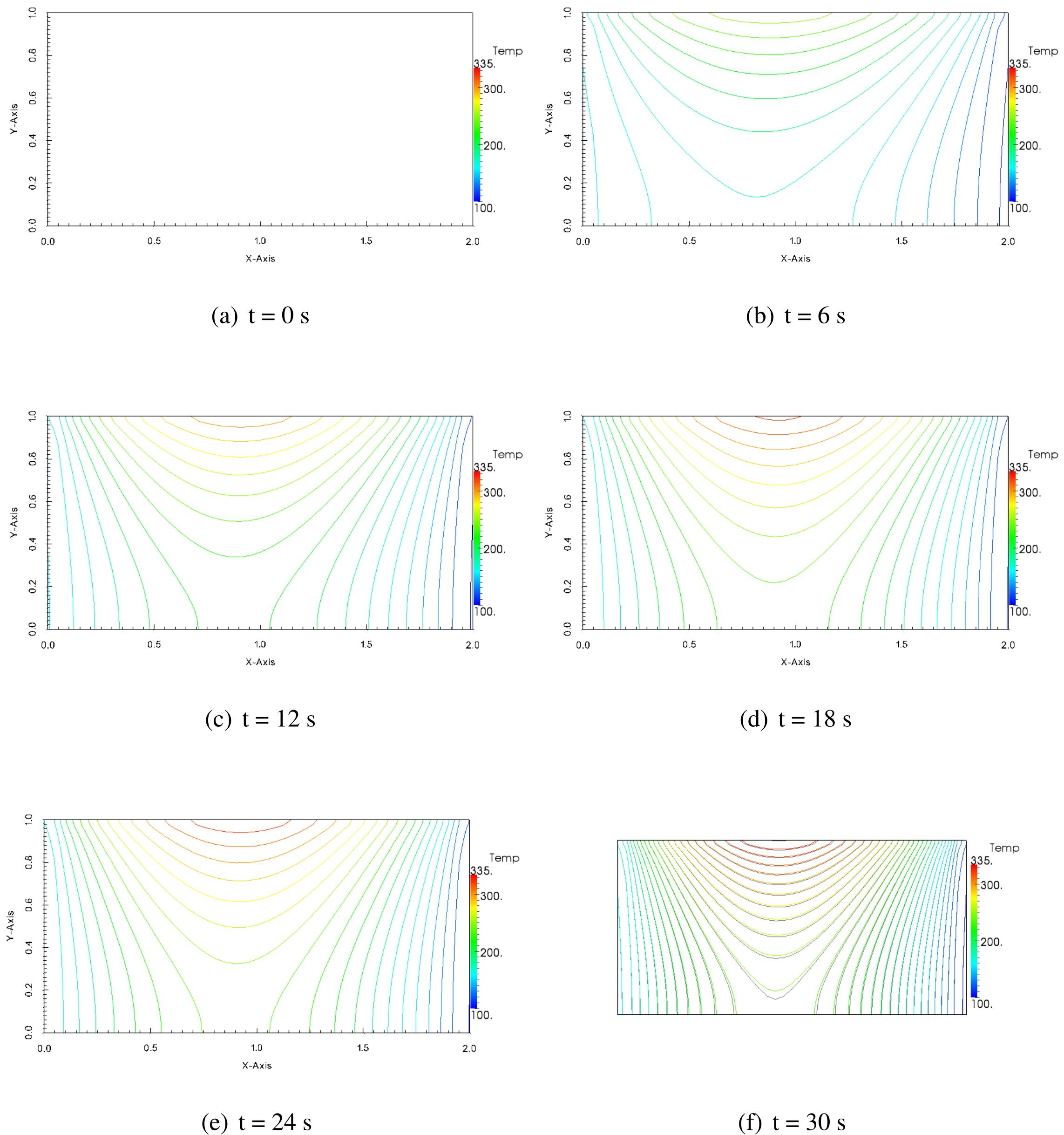
Una vez se ha leído los parámetros de entrada, se llama a la función `CalcularYResolver` en la cual se usa $\theta = 1$ para que el análisis sea incondicionalmente estable.

```
...
DifusionTransitoria temp;
temp.Leer("ARCH=datos_difusion.txt",
  "ARCH=Solve_datos.txt","dt=6 t en [0,30]",100.0);
temp.CalcularYResolver(1.0);
...
```

5.1.2 Resultados

En la figura 26 se presenta los resultados del análisis en el tiempo (líneas de colores) según los parámetros especificados anteriormente y se comparan con los resultados en estado estable mostrados con líneas grises. Se aplicó los filtros `CellDataToPointData` y `Contour`. El máximo valor obtenido del número discreto de Fourier es 1058.

Figura 26: Presentación de los resultados para la clase DifusionTransitoria



(a) t = 0 s

(b) t = 6 s

(c) t = 12 s

(d) t = 18 s

(e) t = 24 s

(f) t = 30 s

5.2 CONVECCIÓN-DIFUSIÓN EN ESTADO TRANSITORIO

A continuación se presenta la ecuación de convección-difusión en estado transitorio (Ec 5.9).

$$\frac{\partial(\rho\, T)}{\partial t} + \nabla \cdot (\rho\, \vec{V}\, T) = \nabla \cdot (\frac{k}{C_p}\; \nabla\; T) + \frac{q}{C_p} \qquad \text{(Ec 5.9)}$$

La (Ec 5.9) se integra en un volumen de control Ω y se discretiza teniendo en cuenta la aproximación del término fuente (Ec 3.6), la definición de F y D en (Ec 4.6), Pe en (Ec 4.8), la función $A(Pe)$ en la tabla 11, θ en (Ec 5.5) y la discretización del término transitorio (Ec 5.3). El resultado se muestra en la (Ec 5.10).

$$[\frac{\rho\Omega_p}{\Delta t} + \theta \sum_{l=1}^{n} (D_l A(Pe_l) + máx(0, F_l))] T_p^{t+\Delta t} - \theta \sum_{l=1}^{n} [(D_l A(Pe_l) + máx(0, -F_l)) T_l^{t+\Delta t}] =$$
$$\frac{q}{C_p}\Omega_p + \frac{\rho\;\Omega_p}{\Delta t}\; T_p^t + (1-\theta) \sum_{l=1}^{n} [(D_l * A(Pe_l)(T_l^t - T_p^t) + máx(0, F_l)\; T_l^t - máx(0, -F_l)\; T_p^t] \qquad \text{(Ec 5.10)}$$

5.2.1 Implementación

Se desarrollará el mismo problema propuesto en la sección 4.2 para poner a prueba el análisis transitorio, la temperatura inicial del problema será $T = 1[C]$.

En la (Ec 5.11) se presenta la forma general de la ecuación a aplicar teniendo en cuenta las condiciones de frontera y en la (Ec 5.12) se definen los coeficientes usados en la (Ec 5.11).

$$a_p * T_p^{t+\Delta t} - \theta \sum_{e=1}^{n-nb} [a_e * T_e^{t+\Delta t}] = b \qquad \text{(Ec 5.11)}$$

En donde

$$a_p \;= a_{tr} + \theta \sum_{e=1}^{n-nb} [a_e \;+\; F_e\;] + \theta \sum_{j=1}^{nb} [a_{b,j}] \qquad \text{(Ec 5.12a)}$$

$$a_{tr} = \frac{\rho\;\Omega}{\Delta t} \qquad \text{(Ec 5.12b)}$$

$$a_e = D_e * A(Pe_e, \text{“metodo”}) + máx(0, -F_e) \qquad \text{(Ec 5.12c)}$$

$$F_e = \rho_e\, V_n\, S \qquad \text{(Ec 5.12d)}$$

$$D_e = \begin{cases} \dfrac{k\, S_e}{C_p \delta_e} & \text{si } \dfrac{k}{C_p} \text{ es constante en toda la malla o se analiza una frontera Dirichlet} \\ \dfrac{S_e}{\frac{C_{p_e} \delta_{ee}}{k_e} + \frac{C_{p_p} \delta_{pe}}{k_p}} & \text{si cada volumen de control tiene una } k \text{ constante} \end{cases} \qquad \text{(Ec 5.12e)}$$

$$a_{b,j} = \begin{cases} D_j * A(Pe_j, \text{“metodo”}) + máx(F_j, 0) & \text{si es} \quad \text{Frontera tipo Dirichlet} \\ \\ F_j & \text{si es} \quad \text{Frontera tipo Neumann} \end{cases} \tag{Ec 5.12f}$$

$$b = b_{gen} + \sum_{j=1}^{nb} [b_{b,j} - (1-\theta) a_{b,j} \, T_p^t] + a_{tr} T_p^t + (1-\theta) \sum_{e=1}^{n-nb} [a_e * T_e^t - (a_e + F_e) \, T_p^t] \tag{Ec 5.12g}$$

$$b_{gen} = \frac{q}{C_p} * \Omega \tag{Ec 5.12h}$$

$$b_{b,j} = \begin{cases} [D_j * A(Pe_j, \text{“metodo”}) + máx(-F_j, 0)] * T_{wall} & \text{si es} \quad \text{Frontera tipo Dirichlet} \\ \\ (\frac{\partial T}{\partial n})_{wall} * S_j * \frac{k}{C_p} & \text{si es} \quad \text{Frontera tipo Neumann} \end{cases} \tag{Ec 5.12i}$$

Para desarrollar el problema, se crea la clase `ConveccionTransitoria` a partir de los siguientes pasos:

1. Declarar la clase `ConveccionTransitoria` y agregar las variables correspondientes.
2. Agregar los métodos a la clase.
3. Definición de las funciones.
4. Crear un objeto de la clase.

Paso 1. Declarar la clase ConveccionTransitoria y agregar las variables correspondientes. La declaración de la clase se hace de la misma forma que la clase `DifusionTransitoria`; además de las variables presentadas en `DifusionTransitoria` se crean las siguientes variables.

```
...
class ConveccionTransitoria :  protected GuardarEnsight
{
...
Puntero<CamposFVM>    campo_vel;
Vector<real> u;
Vector<real> v;
...
bool primer_peclet;
real phi_inicial;
real pe_min ;
real pe_max ;
...
```

Se define un objeto de tipo `Puntero<CamposFVM>` que permite definir campos vectoriales como sucede en el caso de la velocidad. Adicionalmente se definen los vectores que contienen la velocidad en cada volumen de control en dirección x y en dirección y, `u` y `v` , se

definen los parámetros de control `primer_peclet`, `phi_inicial`, `pe_min` y `pe_max`.

Paso 2. Agregar los métodos a la clase. La función `difusion(...)` y sus sobrecargas,
`conveccion(...)`, `peclet_min()`, `peclet_max()`, `funcion_a(...)` y `peclet(...)` fueron presentadas en el capítulo 4.

```
...
protected:
virtual int CalculoUnTiempo();
virtual bool SolucionUnTiempo();
virtual void Reportarresultados();
public:
ConveccionTransitoria(void);
~ConveccionTransitoria(void){}
int Leer (Cadena archivo,Cadena solucionador,
  Cadena def_tiempo, real T_inicial=1);
int CalcularYResolver(real dependencia_futuro);
virtual real S_fuente (real x, real y);
real funcion_a ( real pe, Cadena modo);
real difusion(int vol1, int dir1, int vol2, int dir2, real area);
real difusion(int vol1, int dir1, real area);
real difusion(int vol1, int dir1, real area, real conveccion_f);
real conveccion( int vol1, int dir1 ,real area, Vector_xyz<real> Vel);
real peclet_min(void);
real peclet_max(void);
real peclet(real F, real D);
};
```

Se definen las funciones `Leer(...)`, `CalculoUnTiempo()`, `SolucionUnTiempo()`, `Reportarresultados()`, `CalcularYResolver(...)` y `S_fuente(...)` que difieren en su procedimiento de las funciones homónimas definidas en la clase `DifusionTransitoria`.

La definición completa del archivo de encabezado de la clase `ConveccionTransitoria.h` se encuentra en el anexo J y la definición de las funciones en el archivo de código fuente (`ConveccionTransitoria.cpp`) se encuentra en el anexo K.

Paso 3. Definición de las funciones. Se muestran las modificaciones realizadas en las funciones que han sido previamente definidas.

1. Función `Leer()`: se define la velocidad del fluido para cada volumen de control y las propiedades de acuerdo al problema presentado en la sección 4.2. Seguidamente, se asigna la memoria al puntero inteligente `campo_vel` del campo vectorial.

```
int ConveccionTransitoria::Leer(Cadena archivo,Cadena solucionador,
```

```
   Cadena def_tiempo, real T_inicial)
{
...
 u.chaSize (nelementos); v.chaSize (nelementos);
 for (int i=1; i<=nelementos;i++)
 {
  u(i) =  elems(i)->centroid(malla->obtCoorElem(i))(1);
  v(i) = -elems(i)->centroid(malla->obtCoorElem(i))(2);
 }
 campo_vel.vincular(new CamposFVM(malla(),"Velocidad"));
 k.chaSize(nelementos); k.llenar(0.01);
 rho.chaSize(nelementos); rho.llenar(1.0);
 cp.chaSize(nelementos); cp.llenar(1.0);
 alph.chaSize(nelementos);
...
}
```

2. Función `CalcularYResolver`: para el cálculo transitorio se debe exportar como resultado la condición inicial de la temperatura y el campo vectorial de la velocidad, para ello, es necesario pasar la información contenida en los vectores hacia el campo escalar y el campo vectorial, acción realizada por `gdl->vector2campo(...)`, inmediatamente después se realiza el reporte de los resultados.

```
int ConveccionTransitoria::CalcularYResolver(real dependencia_futuro)
{
...
 if (tiempo_p->estacionario())
 {
   ...
 }
 else
 {
  theta=dependencia_futuro;
  gdl->vector2campo(x_futuro,phi());
  gdl->vector2campo(u,campo_vel()(1));
  gdl->vector2campo(v,campo_vel()(2));
  Reportarresultados();
  do
  {
   ...
  }
  while(!tiempo_p->finalizo());
  std_o<<"\nMinimo Fourier "<<Fo_min<<"\nMaximo Fourier "<<Fo_max<<"\n";
```

```
 }
 std_o<<"Peclet minimo "<< peclet_min() <<
     "Peclet maximo "<< peclet_max() << "\n";
 solucionado=true;
 return 1;
}
```

Adicionalmente se exporta el máximo y mínimo valor absoluto del número discreto de Fourier y del Peclet.

3. Función `S_fuente(...)`: para resolver el problema de la sección 4.2 se debe definir la generación de calor como 0.

```
real ConveccionTransitoria::S_fuente(real x, real y)
{
return 0.0;
}
```

4. Función `CalculoUnTiempo()`: se declaran las siguientes variables para realizar el cálculo de los coeficientes del sistema de ecuaciones.

```
int ConveccionTransitoria::CalculoUnTiempo()
{
 real tr, hfrontera;
 int e, h, l;
 int veci; int conx;
 Cadena boname, bovalue;
 int nbi = malla->obtNoIndFront ();
 double valor;
real F, D, Pe, a_e, a_bj, b_bj, A;
 real area, y, volumen,d1,d2;
 Vector_xyz<real> vel(2), pos;
...
```

Se destaca la creación de `vel` de tipo `Vector_xyz<real>` en la cual se almacenará el vector de velocidad en una superficie. Posteriormente se guarda la información del vector solución `x_futuro` en el vector `x` actualizando de esta forma el valor en el tiempo pasado.

```
...
 x=x_futuro;
 matriz.llenar(0.0);
 b.llenar(0.0);
 for(e=1;e<=nelementos;e++)
```

```
{
 ...
 if (!tiempo_p->estacionario() )
 {
 ...
 }
 for( l=1 ; l<=conx ; l++ )//por cada conexion
 {
 ...
```

Se verifica si existe o no una condición de frontera en el lado `l` del volumen de control `e` y se asignan los coeficientes de acuerdo a (Ec 5.11) y (Ec 5.12).

```
...
if (veci!=0)
{
 for ( h=1 ; h<=conx ; h++ )
 { if(malla->getConx(veci,h)==e)   break;}
 d1 = elems(e)->getDelta(l);
 d2 = elems(veci)->getDelta(h);
 vel(1) = ( u(e)*d2 + u(veci)*d1 )/(d1+d2);
 vel(2) = ( v(e)*d2 + v(veci)*d1 )/(d1+d2);
 F  = conveccion(e,l,area,vel);
 D  = difusion( e, l, veci, h, area);
 Pe = peclet( F,D );
 A  = funcion_a(Pe, "UDS");
 a_e = D * A + std::max( 0.0 , -F );
 matriz(e,e   ) +=   (a_e + F)  * theta ;
 matriz(e,veci)  = -a_e * theta;
 b(e) += (1 - theta) * (-(a_e + F)*x(e) + a_e*x(veci)) ;
 if (!tiempo_p->estacionario())
 {
  Fourier(alph(e),tiempo_p->Delta(),
  elems(e)->getDelta(l) + elems(veci)->getDelta(h));
 }
}
else
{
 for (h=1; h<=nbi; h++)
 {
 if (  malla->LadoFront(e,l,h,elems(e)())   )
 {
  vel(1) =  elems(e)->centroArea(malla->obtCoorElem(e),l)(1) ;
```

```
  vel(2) = -elems(e)->centroArea(malla->obtCoorElem(e),l)(2) ;
  F       = conveccion(e,l,area,vel) ;
  boname  = malla->obtNombreIndFront(h);
  bovalue = boname.despues('=');
  valor   = atof(bovalue.carts());
  D     = difusion(e,l,area);
  if ( boname.contiene("T") || boname.contiene("dirichlet")
      || boname.contiene("Dirichlet")
     )
  {
   if( boname.contiene("h=") )
   {
    bovalue = boname.despues("T=");
    valor   = atof(bovalue.carts());
    bovalue = boname.despues("h=");
    hfrontera   = atof(bovalue.carts());
    D = difusion(e,l,area,hfrontera);
   }
   if( boname.contiene("L") )
   {
    y     = elems(e)->centroArea(malla->obtCoorElem(e),l)(2);
    valor = 1.0 + valor * (  (1-y) );
   }
   Pe = peclet(F,D);
   A = funcion_a(Pe,''UDS'');
   a_bj =  D*A + std::max(F,0.0);
   b_bj = (D*A + std::max(-F,0.0))*valor;
  }
  if ( boname.contiene("Q") )
  {
   a_bj = F;
   b_bj = valor * area/cp(e) ;
  }
  if ( boname.contiene("Neumann")||boname.contiene("neumann") )
  {
   a_bj = F;
   b_bj = valor * area * k(e)/cp(e);
  }
  matriz (e,e)  += a_bj * theta;
  b (e) += b_bj - (1-theta)*a_bj ;
  }
 }
}
```

```
}
}
return 1;
}
```

5. Función `SolucionUnTiempo()`: se resuelve el sistema de ecuaciones y se actualiza el valor del campo escalar de la temperatura y el campo vectorial de la velocidad.

```
bool ConveccionTransitoria::SolucionUnTiempo()
{
 if (primera_solucion)
 { admlin().adjuntar(matriz, x_futuro, b);
  primera_solucion=false;  }
 bool solucion=admlin().solve();
gdl->vector2campo(x_futuro,phi());
gdl->vector2campo(u,campo_vel()(1));
gdl->vector2campo(v,campo_vel()(2));
return solucion;
}
```

6. Función `Reportarresultados()`: de nuevo se usa la función `volcar` de `GuardarEnsight` para exportar los resultados, ya sea de un campo escalar o vectorial.

```
void ConveccionTransitoria::Reportarresultados()
{
GuardarEnsight::volcar(phi(),tiempo_p.obtPtr());
GuardarEnsight::volcar(campo_vel(),tiempo_p.obtPtr());
}
```

Paso 4. Crear un objeto de la clase. La clase `ConveccionTransitoria` permite resolver la ecuación de convección-difusión ya sea en estado transitorio o en estado estable; un ejemplo de la creación de un objeto de la clase es presentado en el siguiente fragmento de código.

```
ConveccionTransitoria conv;
conv.Leer ("ARCH=datos_conveccion_difusion.txt",
      "ARCH=Solve_datos.txt","dt=[1.0 5.0] t en [0.0 4.0 9.0]",1.0);
conv.CalcularYResolver(1.0);
```

Con la cadena de texto `"dt=[1.0 5.0] t en [0.0 4.0 9.0]"` se utiliza la capacidad del objeto `prmTiempo` de definir diferentes avances en el tiempo, la cadena usada implica que se tomará $\Delta t = 1$ para $t \in [0,4]$ y $\Delta t = 5$ para $t \in (4,9]$.

Los archivos `datos_conveccion_difusion.txt`, `datos_conv_dif.geom` y `Solve_datos.txt` ya fueron presentados en el capítulo 4. Ahora se presenta la modificación realizada al archivo `datos_conv_dif.parts` en el cual se definen 3600 particiones.

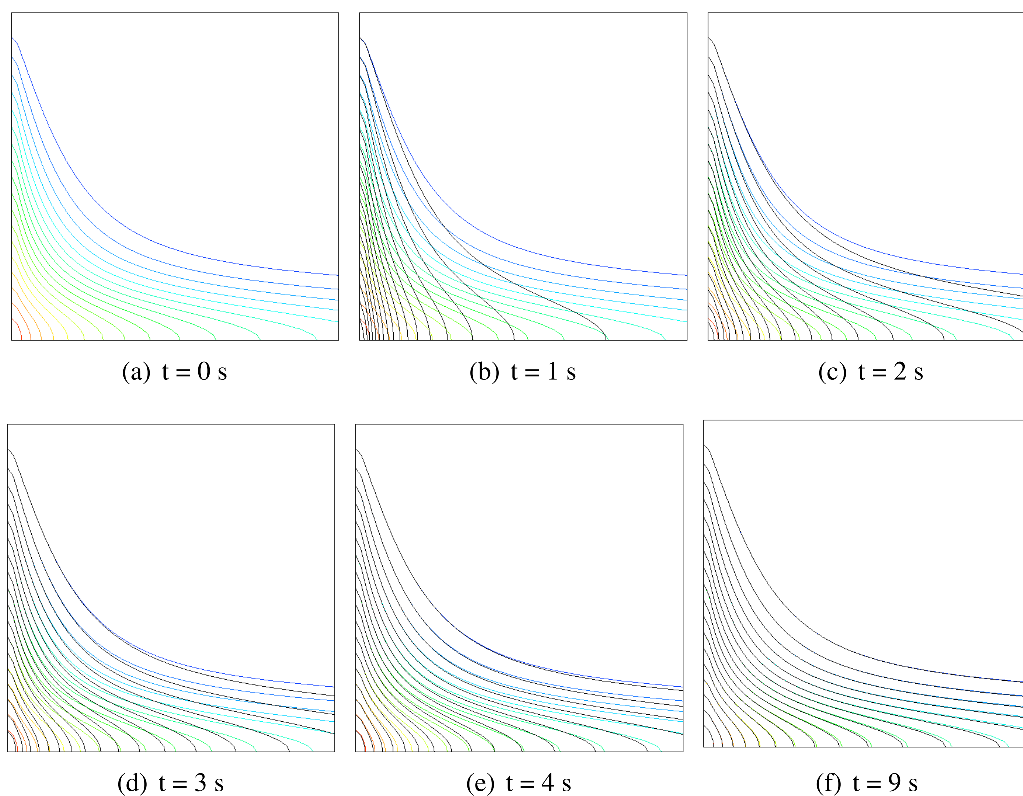
```
>nsd = 2;
>no_de_superels = 1;

>SE;>d=2;>e=ElmB4n2D;>div=[60,60];>grad=[1,1];
```

5.2.2 Resultados

En la figura 27 se presentan los resultados mostrados en Paraview al ejecutar el procedimiento previamente mostrado para una instancia de `ConveccionTransitoria`, se utilizaron los filtros `CellDataToPointData` y `Contour` para generar las gráficas. Las líneas isotérmicas en color corresponden al estado estable y las líneas isotérmicas negras corresponden al estado transitorio, se puede ver que las líneas isotérmicas en el estado transitorio tienden a acercarse a las líneas isotérmicas en el estado estable.

Figura 27: Presentación de los resultados para la clase ConveccionTransitoria.



(a) t = 0 s (b) t = 1 s (c) t = 2 s

(d) t = 3 s (e) t = 4 s (f) t = 9 s

6. CFD DEL FLUJO EN LA CAPA LÍMITE

Se presentan dos métodos para resolver las ecuaciones de la capa límite, el primer método tiene en cuenta las ecuaciones completas de Navier-Stokes y la ecuación de la energía para bajas velocidades como se muestra en la tabla 13. El segundo método resuelve las ecuaciones simplificadas de la capa límite donde la presión se asume conocida.

Tabla 13: Ecuaciones conservativas

Principio de conservación	Término transitorio		Término convectivo		Término difusivo		Término fuente
Continuidad	$\frac{\partial \rho}{\partial t}$	$+$	$\nabla \cdot (\rho \vec{V})$	$=$	0	$+$	0
Cant. mov.	$\frac{\partial (\rho\, u)}{\partial t}$	$+$	$\nabla \cdot (\rho\, u \vec{V})$	$=$	$\nabla \cdot (\mu \nabla u)$	$+$	$-\frac{\partial p}{\partial x} + b^u$
Eje x	$b^u = \frac{\partial}{\partial x}[\lambda(\nabla \vec{V})] + \frac{\partial}{\partial x}\left(\mu \frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left[\mu \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\right)\right] + \frac{\partial}{\partial z}\left[\mu \left(\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z}\right)\right]$						
Cant. mov.	$\frac{\partial (\rho\, v)}{\partial t}$	$+$	$\nabla \cdot (\rho\, v \vec{V})$	$=$	$\nabla \cdot (\mu \nabla v)$	$+$	$-\frac{\partial p}{\partial y} + b^v$
Eje y	$b^v = \frac{\partial}{\partial y}[\lambda(\nabla \vec{V})] + \frac{\partial}{\partial y}\left(\mu \frac{\partial v}{\partial y}\right) + \frac{\partial}{\partial x}\left[\mu \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)\right] + \frac{\partial}{\partial z}\left[\mu \left(\frac{\partial w}{\partial y} + \frac{\partial v}{\partial z}\right)\right]$						
Cant. mov.	$\frac{\partial (\rho\, w)}{\partial t}$	$+$	$\nabla \cdot (\rho\, w \vec{V})$	$=$	$\nabla \cdot (\mu \nabla w)$	$+$	$-\frac{\partial p}{\partial z} + b^w$
Eje z	$b^w = \frac{\partial}{\partial z}[\lambda(\nabla \vec{V})] + \frac{\partial}{\partial z}\left(\mu \frac{\partial w}{\partial z}\right) + \frac{\partial}{\partial x}\left[\mu \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right)\right] + \frac{\partial}{\partial y}\left[\mu \left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right)\right]$						
Energía	$\frac{\partial (\rho\, T)}{\partial t}$	$+$	$\nabla \cdot (\rho\, T \vec{V})$	$=$	$\nabla \cdot \left(\frac{k}{C_p} \nabla T\right)$	$+$	b^T
	$b^T = \frac{q}{C_p} + \frac{1}{C_p}\frac{\partial p}{\partial t}$						

En la tabla 13 se separaron los términos convectivo, difusivo y transitorio. Para el FVM resulta conveniente expresar el término fuente en términos del operador ∇ como se muestra en (Ec 6.1a) y (Ec 6.1b). Posteriormente se indicará la ventaja de discretizar el término fuente de la forma (Ec 6.1).

$$\frac{\partial p}{\partial x} = (\nabla p) \cdot \hat{i} \qquad \frac{\partial p}{\partial y} = (\nabla p) \cdot \hat{j} \qquad \frac{\partial p}{\partial z} = (\nabla p) \cdot \hat{k} \tag{Ec 6.1a}$$

$$b^u = (\nabla \boldsymbol{\tau}) \cdot \hat{i} \qquad b^v = (\nabla \boldsymbol{\tau}) \cdot \hat{j} \qquad b^w = (\nabla \boldsymbol{\tau}) \cdot \hat{k} \tag{Ec 6.1b}$$

$$\boldsymbol{\tau} = \begin{bmatrix} -\frac{2}{3}\,\mu\,(\nabla \cdot \vec{V}) + \mu\,\frac{\partial u}{\partial x} & \mu\left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\right) & \mu\left(\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z}\right) \\ \mu\left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\right) & -\frac{2}{3}\,\mu\,(\nabla \cdot \vec{V}) + \mu\frac{\partial v}{\partial y} & \mu\left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right) \\ \mu\left(\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z}\right) & \mu\left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right) & -\frac{2}{3}\,\mu\,(\nabla \cdot \vec{V}) + \mu\,\frac{\partial w}{\partial z} \end{bmatrix} \tag{Ec 6.1c}$$

6.1 MÉTODO SIMPLE

El método SIMPLE (*Semi-Implicit Method for Pressure-Linked Equations*) fue presentado en 1972 por Patankar y Spalding como un método iterativo para solucionar las ecuaciones de Navier-Stokes. Este método, inicialmente diseñado para resolver problemas de ingeniería, es usado actualmente en muchos software comerciales por su capacidad de encontrar la solución a una gran variedad de problemas.

El método SIMPLE tiene como primer paso resolver las ecuaciones de la cantidad de movimiento, en el segundo paso se resuelve la ecuación de continuidad en términos de una correción de presión, en el tercero se corrige el valor de la presión y la velocidad y finalmente en el cuarto paso se calculan las demás ecuaciones que se deben resolver para el problema planteado.

Los campos obtenidos por el método SIMPLE pueden tener oscilaciones y en general tendrán una baja velocidad de convergencia, por tal motivo se han creado otros métodos que modifican o agregan otros pasos al método SIMPLE para subsanar estos inconvenientes.

En el método SIMPLE se trata la ecuación vectorial de la cantidad de movimiento como una ecuación lineal (Ec 6.2). La (Ec 6.2), escrita como una ecuación de convección-difusión, agrupa casi todos los términos que no dependen directamente de la velocidad en el término fuente, el único término que no se agrupa es el gradiente de la presión.

$$\frac{\partial(\rho\vec{V})}{\partial t}+\nabla(\rho\vec{V}\cdot\vec{V})=\nabla\cdot(\mu\nabla\vec{V})-\nabla p+\boldsymbol{b}^V \Rightarrow \quad \boldsymbol{a}_p^V\vec{V}_p=\sum_{i=1}^{n-nb}\left[\boldsymbol{a}_i^V\vec{V}_i\right]-\nabla p_p\Omega_p+\vec{b}_p^V\Omega_p \tag{Ec 6.2}$$

Para escribir la (Ec 6.2) en forma vectorial se definen los coeficientes $\boldsymbol{a}_p^V$ y $\boldsymbol{a}_i^V$ como matrices diagonales (Ec 6.3) pero, se debe recalcar que la solución de cada una de las componentes de la velocidad se halla de forma independiente a las otras velocidades. En general se cumple que $a_p^u=a_p^v=a_p^w$ y $a_i^u=a_i^v=a_i^w$ ya que cada ecuación de (Ec 6.2) tiene los mismos términos de $\rho\,\vec{V}$ y μ, solo son diferentes los términos que tienen distintos tipos de condiciones de frontera, por ejemplo, un lado con una condición de frontera tipo Dirichlet para la velocidad u y condición de frontera tipo Neumann para la velocidad v.

$$\boldsymbol{a}_p^V=\begin{bmatrix} a_p^u & 0 & 0\\ 0 & a_p^v & 0\\ 0 & 0 & a_p^w\end{bmatrix}\qquad \boldsymbol{a}_i^V=\begin{bmatrix} a_i^u & 0 & 0\\ 0 & a_i^v & 0\\ 0 & 0 & a_i^w\end{bmatrix}\qquad \vec{V}=\begin{bmatrix} u\\ v\\ w\end{bmatrix} \tag{Ec 6.3}$$

La solución de (Ec 6.2) se encuentra cuando se introduce el campo de la presión correcto; como no se conoce ni el campo de la velocidad ni el campo de la presión, se introduce un contador iterativo m obteniendo la (Ec 6.4). En todo proceso iterativo, se debe actualizar el valor de la variable que se está calculando, por este motivo se corrige el valor de la velocidad y la presión como se muestra en (Ec 6.5) y (Ec 6.6) para obtener la solución. El valor de p' y

$\vec{V}'$ representan la cantidad a agregar a p^{m-1} y a $\vec{V}^{m*}$ para encontrar una mejor aproximación para la presión y la velocidad que finalmente cumplan con las ecuaciones de continuidad y cantidad de movimiento.

$$\boldsymbol{a}_p^V\ \vec{V}_p^{m*} = \sum_{i=1}^{n-nb}\left[\boldsymbol{a}_i^V\ \vec{V}_i^{m*}\right] - \nabla p_p^{m-1}\Omega_p + \vec{b}_p^V\Omega_p \qquad \text{(Ec 6.4)}$$

$$p = p^{m-1} + p' \qquad \text{(Ec 6.5)}$$

$$\vec{V} = \vec{V}^{m*} + \vec{V}' \qquad \text{(Ec 6.6)}$$

El proceso iterativo del método SIMPLE tiene como **primer paso**[1] la solución de (Ec 6.4) en la cual se realiza el cálculo de $\boldsymbol{a}_p^V, \boldsymbol{a}_i^V, \nabla p_p^{m-1}$ y $\vec{b}_p^V$ con el valor del campo de la velocidad y la presión hallados en la iteración anterior. La (Ec 6.4) es una ecuación lineal de convección-difusión cuyo proceso de solución se presentó en los capítulos 4 y 5.

Una vez se ha calculado el valor de $\vec{V}^{m*}$ se debe hallar el valor de $\vec{V}'$ y para ello, se resta (Ec 6.4) de (Ec 6.2) obteniendo (Ec 6.7).

$$\boldsymbol{a}_p^V\ (\vec{V}_p - \vec{V}_p^{m*}) = \sum_{i=1}^{n-nb}\left[\boldsymbol{a}_i^V\ (\vec{V}_i - \vec{V}_i^{m*})\right] - \nabla(p_p - p_p^{m-1})\Omega_p \qquad \text{(Ec 6.7)}$$

Recordando la definición de (Ec 6.6) y (Ec 6.5) se puede re-escribir (Ec 6.7) como (Ec 6.8).

$$\boldsymbol{a}_p^V\ \vec{V}_p' = \underbrace{\sum_{i=1}^{n-nb}\left[\boldsymbol{a}_i^V\ \vec{V}_i'\right]}_{\text{SIMPLE}\dashrightarrow 0} - \nabla\, p_p'\Omega_p \qquad \text{(Ec 6.8)}$$

El término $\vec{V}_p'$ es la correción para la velocidad y de su valor depende la rapidez de convergencia. Para el método SIMPLE se desprecia el primer término del lado derecho de (Ec 6.8), como se explicará posteriormente, el método SIMPLE presenta una baja velocidad de convergencia comparado con los métodos SIMPLER, SIMPLEC y PISO. Una vez realizado este proceso, se despeja $\vec{V}_p'$ de (Ec 6.8) y se obtiene (Ec 6.9).

$$\vec{V}_p' \approx -(\boldsymbol{a}_p^V)^{-1}\ \Omega_p\ \nabla\ p_p' \qquad \text{(Ec 6.9)}$$

El **segundo paso** del método consiste en resolver la ecuación de continuidad con el campo de correción de presión. Como la ecuación de continuidad no depende directamente del campo de presión se usan las definiciones de (Ec 6.6) y (Ec 6.9) para resolver la ecuación de continuidad, la ecuación resultante se muestra en (Ec 6.11).

$$0 = \nabla\cdot(\rho\vec{V}) = \nabla\cdot[\rho(\vec{V}^{m*} + \vec{V}')] = \nabla\cdot[\rho\vec{V}^{m*}] + \nabla\cdot[\rho\vec{V}'] \qquad \text{(Ec 6.10a)}$$

[1] Se recuerda que en todo proceso iterativo se debe dar un valor inicial a las variables de manera que se pueda comenzar el proceso de iteración.

$$\nabla\cdot[\rho\vec{V}'] = -\nabla\cdot[\rho\vec{V}^{m*}] \quad \text{(Ec 6.10b)}$$

$$\nabla\cdot\left[(\boldsymbol{a}_p^V)^{-1}\,\rho\,\Omega\,\nabla p'_p\right] = \underbrace{\nabla\cdot(\rho\,\vec{V}_p^{m*})}_{\text{Residuo másico}} \quad \text{(Ec 6.11)}$$

La (Ec 6.11) es una ecuación de Poisson que se resuelve de la misma forma que se presentó en el capítulo 3 o 5. El término del lado derecho, que usa el valor de $\vec{V}_p^{m*}$ hallado en el primer paso, es llamado **residuo másico** y se usa como criterio de convergencia para el método. Cabe resaltar que el residuo másico es positivo cuando el volumen de control está perdiendo masa, este signo se tiene en cuenta en la implementación del método.

Una vez se ha calculado la correción de presión se actualiza el valor del campo de la velocidad y el campo de presión. El **tercer paso** del método consiste en aplicar (Ec 6.12).

$$p^m = p^{m-1} + p' \quad \text{(Ec 6.12a)}$$

$$\vec{V}^m = \vec{V}^{m*} - (\boldsymbol{a}_p^V)^{-1}\,\Omega\,\nabla p'_p \quad \text{(Ec 6.12b)}$$

La corrección de presión se aplica al valor de la presión hallado en la iteración anterior, p^{m-1}, sin embargo, la correción de la velocidad se realiza al valor hallado en el primer paso, $\vec{V}^{m*}$.

La solución de las demás ecuaciones como la conservación de la energía o la ecuación de estado para los gases se resuelve en el **cuarto paso** como se indica en (Ec 6.13) y (Ec 6.14).

$$\frac{\partial(\rho\,T^m)}{\partial t} + \nabla\cdot\left[\rho\,T^m\vec{V}\right] = \nabla\cdot\left(\frac{k}{C_p}\,\nabla T^m\right) + b^T \quad \text{(Ec 6.13)}$$

$$\frac{p}{\rho} = R\,T \quad \text{(Ec 6.14)}$$

Finalmente se comprueba si el valor del residuo másico (específicamente la norma del residuo másico) es inferior al criterio de parada y si es el caso se finaliza el proceso iterativo.

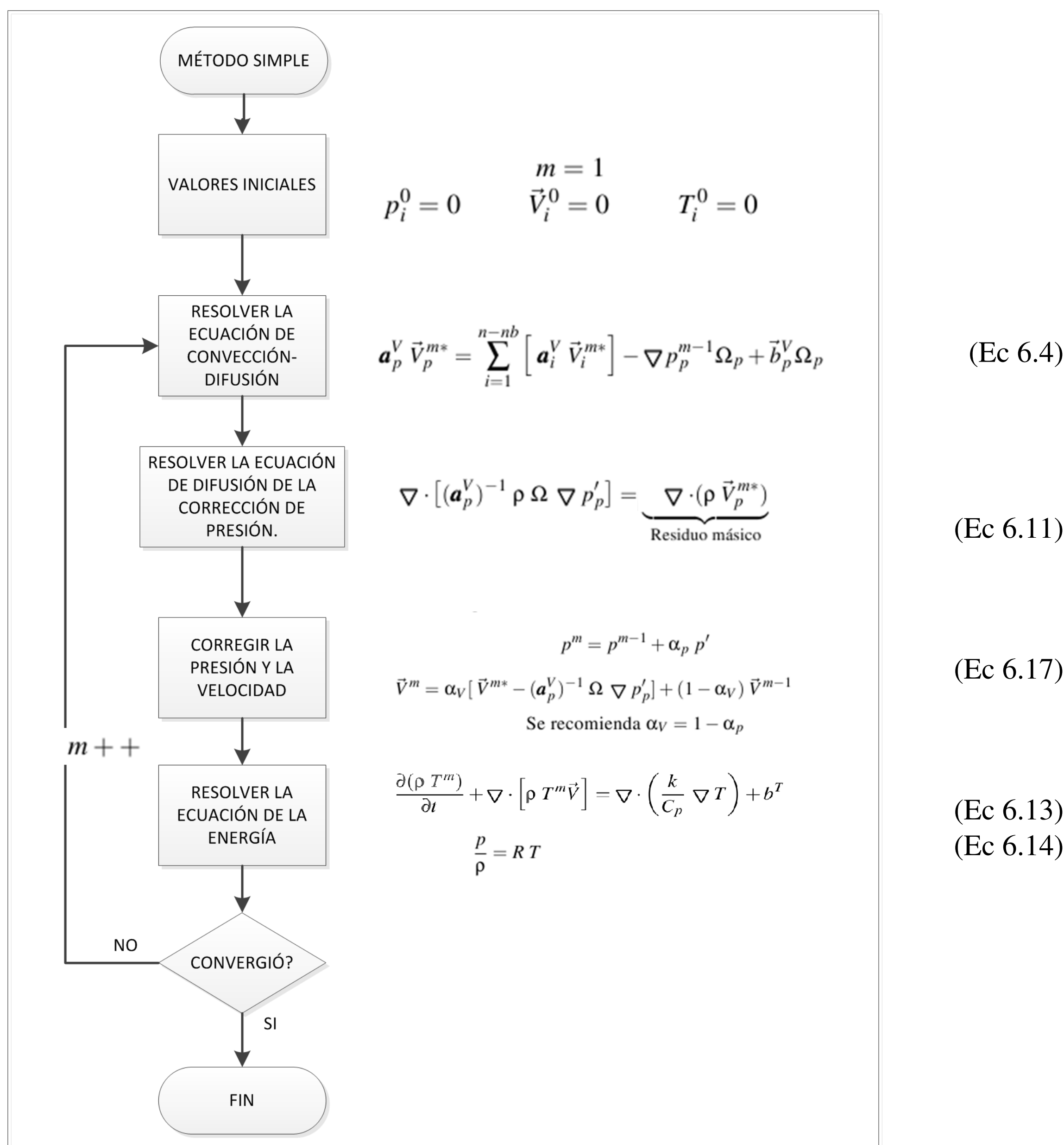
En la figura 28 se presenta un diagrama de flujo que explica los pasos a seguir para ejecutar el método SIMPLE.

6.1.1 Mallas colocadas, mallas desplazadas y cálculo del residuo másico

Para calcular el residuo másico, $\nabla\cdot(\rho\,\vec{V}_p^{m*})$, se debe conocer la velocidad del flujo en la frontera de cada volumen de control pero no siempre se tiene directamente este valor.

El caso en el que el valor de la velocidad está ubicado en el centro de cada cara y el valor de la presión se ubica en el centro del volumen de control corresponde al uso de una malla desplazada. La malla desplazada facilita el cálculo del gradiente de la velocidad, sin embargo, la malla desplazada solo se puede construir en el caso de tener una malla ortogonal.

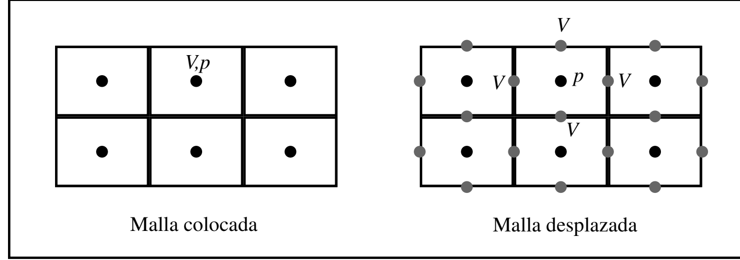
Figura 28: Método SIMPLE.



Por otro lado se encuentra la malla colocada. En la malla colocada, la velocidad se calcula a partir del mismo punto donde están ubicadas las otras variables del problema, de esta forma se pueden obtener soluciones para mallas no ortogonales. Sin embargo, el uso de una malla colocada ocasiona un efecto de zigzag en la solución de la ecuación para la correción de

presión. La solución a este problema es asumir el valor de la velocidad en la frontera a partir de Fuentes [15]. En la figura 29 se presentan los dos tipos de malla mencionados.

Figura 29: Posición del valor de la velocidad y de la presión en un VC para una malla colocada y una malla desplazada.



La correción se muestra en (Ec 6.15) para la frontera j de un volumen de control p que colinda con el volumen e.

$$\vec{V}^{m*}_{p,j} = \underbrace{\overline{(\vec{V}^{m*}_{pe})}}_{\text{(A)}} - \underbrace{\frac{\Omega_{pe}}{\vec{d}_{EP}\cdot\widehat{n}_j}}_{\text{(B)}} \underbrace{\overline{((\boldsymbol{a}^V)^{-1})}_{pe}}_{\text{(C)}} \left[\underbrace{(p_E - p_P)}_{\text{(D)}} - \underbrace{\overline{(\nabla p)}_{pe}\cdot\vec{d}_{EP}}_{\text{(E)}}\right] \quad \text{(Ec 6.15)}$$

En (Ec 6.16) se especifica cada uno de los términos usados en (Ec 6.15):

$$\overline{(\vec{V}^{m*}_{pe})} = \frac{(\vec{V}^{m*}_P + \vec{V}^{m*}_E)}{2} \quad \text{(Ec 6.16a)}$$

$$\frac{\Omega_{pe}}{\vec{d}_{EP}\cdot\widehat{n}_j} = \frac{\Omega_P}{(\vec{d}_{p,j}+\vec{d}_{e,k})\cdot\widehat{n}_j} \quad \text{ó} \quad \frac{\Omega_{pe}}{\vec{d}_{EP}\cdot\widehat{n}_j} = A_{p,j} \quad \text{(Ec 6.16b)}$$

$$\overline{((\boldsymbol{a}^V)^{-1})}_{pe} = \frac{(\boldsymbol{a}^V_P)^{-1} + (\boldsymbol{a}^V_E)^{-1}}{2} \quad \text{(Ec 6.16c)}$$

$$\overline{(\nabla p)}_{pe}\cdot\vec{d}_{EP} = \frac{(\nabla p)_P + (\nabla p)_E}{2}\cdot(\vec{d}_{p,j} + \vec{d}_{e,k}) \quad \text{(Ec 6.16e)}$$

El volumen Ω_{pe} de (Ec 6.15) se interpreta como el volumen alrededor de la fronterape. Este volumen dividido en la distancia $\vec{d}_{EP}$ resulta igual al área de la frontera del volumen p por el lado j independientemente de la forma del volumen de control.

6.2 MODIFICACIONES AL MÉTODO SIMPLE

Como se había mencionado antes, el método SIMPLE puede presentar oscilaciones y en general tiene una baja velocidad de convergencia. Una primera modificación al método consiste en definir dos coeficientes para **subrelajar** la corrección de la presión y la velocidad (Ec 6.17) en el tercer paso. Aunque no se ha encontrado un único valor para α_p y α_V que optimice el número de iteraciones requerido, si se recomienda usar la relación dada en (Ec 6.17c) .

$$p^m = p^{m-1} + \alpha_p\, p' \tag{Ec 6.17a}$$

$$\vec{V}^m = \alpha_V [\, \vec{V}^{m*} - (\boldsymbol{a}_p^V)^{-1}\, \Omega\, \nabla\, p'_p] + (1-\alpha_V)\, \vec{V}^{m-1} \tag{Ec 6.17b}$$

$$\text{Se recomienda } \alpha_V = 1 - \alpha_p \tag{Ec 6.17c}$$

Por otro lado, el método **SIMPLEC** (figura 30) aproxima de una forma distinta la corrección de la velocidad (Ec 6.18a) obteniendo[2] (Ec 6.18c). De esta forma, la ecuación de correción de presión a resolver es (Ec 6.18d).

$$\vec{V}'_p = \vec{\tilde{V}}'_p - (\boldsymbol{a}_p^V)^{-1}\, \Omega\, \nabla\, p'_p \tag{Ec 6.18a}$$

$$\vec{\tilde{V}}'_p \approx -\vec{V}'_p \frac{\sum_{i=1}^{n-nb} \boldsymbol{a}_i^V}{\boldsymbol{a}_p^V} \tag{Ec 6.18b}$$

$$\vec{V}'_p = -(\boldsymbol{a}_p^V + \sum_{i=1}^{n-nb} \boldsymbol{a}_i^V)^{-1}\, \Omega\, \nabla\, p'_p \tag{Ec 6.18c}$$

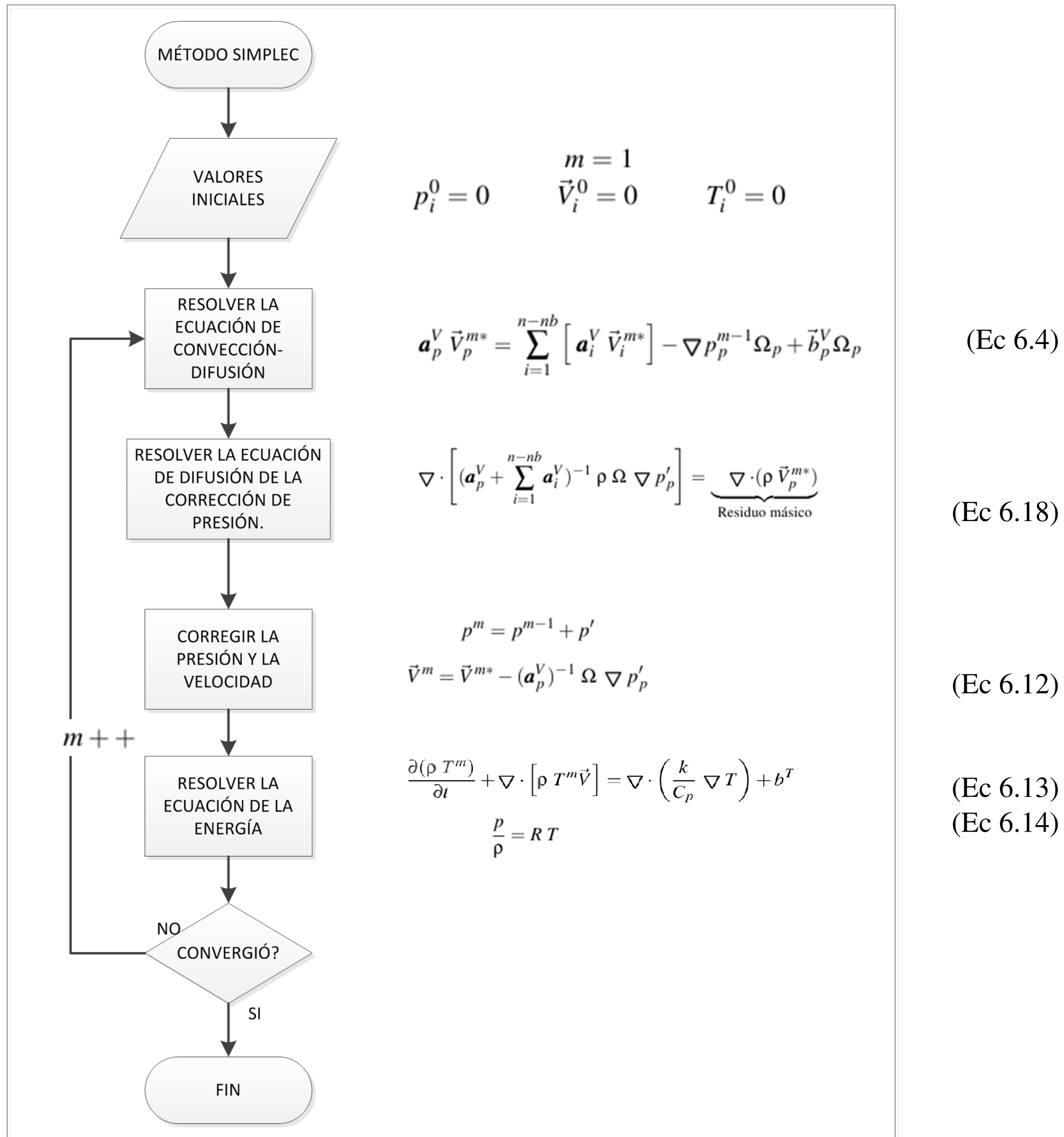
$$\nabla \cdot \left[(\boldsymbol{a}_p^V + \sum_{i=1}^{n-nb} \boldsymbol{a}_i^V)^{-1}\, \rho\, \Omega\, \nabla\, p'_p \right] = \underbrace{\nabla \cdot (\rho\, \vec{V}_p^{m*})}_{\text{Residuo másico}} \tag{Ec 6.18d}$$

6.3 IMPLEMENTACIÓN SIMPLE

Como se ha explicado, el método SIMPLE resuelve dos tipos de ecuaciones, la ecuación de convección-difusión para la cantidad de movimiento en la dirección de cada eje coordenado y por otro lado la ecuación de difusión de corrección de presión. Para implementar este método, el método SIMPLEC y un método simplificado que posteriormente se explicará, se hace uso de las clases `ConveccionTransitoria` y `DifusionTransitoria` con algunas modificaciones. Los cambios realizados a la clase `Convecciontransitoria` se explican en el anexo L.

[2] Se recuerda que los coeficientes $\boldsymbol{a}_p^V$ y $\sum \boldsymbol{a}_i^V$ se obtienen de la solución de las ecuaciones de convección-difusión de la cantidad de movimiento.

Figura 30: Método SIMPLEC.



Se crea la clase `NavierStokes` en la cual se declaran cinco objetos de la clase `ConveccionTransitoria` para las componentes de la velocidad y la temperatura, un objeto de la clase `DifusionTransitoria` para la solución de la ecuación de corrección de presión y otros objetos para ayudar en la solución del problema. Los pasos para desarrollar la clase

son:

1. Declarar la clase `NavierStokes` y agregar las variables.
2. Agregar los métodos a la clase.
3. Definición de las funciones.
4. Crear un objeto de la clase.

Paso 1. Declarar la clase `NavierStokes` y agregar las variables: se incluyen las librerías de `ConveccionTransitoria` y `DifusionTransitoria`. Posteriormente, se declaran algunos vectores de caracteres que definen las opciones disponibles para procesos específicos. En esta clase también se usa la herencia al declarar la clase `GuardarEnsight` como clase padre de `NavierStokes`.

```
//NavierStokes.h
#pragma once
#include "ConveccionTransitoria.h"
#include "DifusionTransitoria.h"
static const char *MACD[] = {"UDS", "Upwind","CDS", "upwind","hybrid",
 "hibrido", "power", "law", "exp", "Exp", NULL};
static const char *MSNS[] = {"SIMPLE","NOSIMPLE", "SIMPLEC", NULL};
const char Separador = ',';
const bool MULT_POR_RHO_ON = true;
const bool MULT_POR_RHO_OFF = false;
class NavierStokes :  protected GuardarEnsight
{
...
```

1. Se declaran tres instancias de la clase `ConveccionTransitoria` para resolver cada una de las componentes de la ecuación vectorial de la cantidad de movimiento. Para el método SIMPLE, la solución de cada una de las ecuaciones de la cantidad de movimiento se almacenará en `VmAstU`, `VmAstV` y `VmAstW` según corresponda; posteriormente, se agregarán a estas velocidades el efecto de la corrección de presión resultando el valor de `VelU`, `VelV` y `VelW` que representa el valor de la velocidad en la iteración *m*. El valor de la velocidad en la iteración anterior es necesario para el uso del parámetro de subrelajación de la presión y la velocidad, es por esto que se crea los vectores `VelU_ant`, `VelV_ant` y `VelW_ant`.

```
...
ConveccionTransitoria ConvVu, ConvVv, ConvVw;
Vector<real> VelU, VelV, VelW;
Vector<real> VelU_ant, VelV_ant, VelW_ant;
Vector<real> VmAstU, VmAstV, VmAstW;
...
```

2. Se declaran las variables en las cuales se almacenarán el gradiente de cada una de las componentes de la velocidad, ya sea en el presente (`GU_ant`,...), en el futuro (`GU_n`,...) o un valor dependiente de ambos (`GU`,...). Se utiliza la clase `Vector_basico` con la plantilla `Vector_xyz<real>` debido a que el resultado del gradiente de un escalar es un vector para cada volumen de control.

```
...
Vector_basico<Vector_xyz<real>> GU, GV, GW, GP, Gtau;
Vector_basico<Vector_xyz<real>> GU_n, GV_n, GW_n, GP_n;
Vector_basico<Vector_xyz<real>> GU_ant, GV_ant, GW_ant, GP_ant;
Vector_basico<Vector_xyz<real>> Fcalor;
Vector<real> GVel, GVel_ant,GVel_n ;
Vector_basico<Matriz<real>> Tau_n, Tau_ant, Tau;
...
```

Se declaran `GVel`, `GVel_ant` y `GVel_n` para almacenar el valor de $\nabla \cdot \vec{V}$ en el pasado, el futuro y en un tiempo intermedio. Por último, en el anterior fragmento de código, se declara el tensor simétrico `Tau` como una instancia de `Vector_basico<Matriz<real>>`.

3. Se declaran `CoefU` y otros vectores para almacenar el valor de los elementos de la diagonal principal en la matriz de coeficientes de cada una de las ecuaciones de la cantidad de movimiento. Se declaran los elementos necesarios para realizar el cálculo de la temperatura para cada volumen de control. Seguidamente, se declara una instancia de `DifusionTransitoria` y otros vectores para el cálculo de la corrección de presión y el valor final de la presión.

```
...
Vector<real> CoefU, CoefV, CoefW;
ConveccionTransitoria Temperatura;
Vector<real> Temp, Temp_ant;
DifusionTransitoria  Pprima;
Vector<real> Presionprima;
Vector<real> Presion, Presion_ant, Presion_n;
ConveccionTransitoria Corrv;
Vector<real> vcor;
...
```

La instancia `Corrv` de la clase `ConveccionTransitoria` se encarga de calcular la corrección de la velocidad por el método NOSIMPLE que posteriormente se explicará.

4. Con el fin de almacenar el volumen de las celdas se define el `vector<real> Volumen`. Se declara el vector `Fuente` el cual se usará para almacenar el valor del término fuente en cada una de las ecuaciones a resolver. Adicionalmente, se define el vector `ResiduoMasico` el cual es un término esencial para el cálculo de la correción de presión. La norma de `ResiduoMasico` será uno de los criterios de convergencia de las iteraciones exteriores.

```
...
Vector<real> Volumen;
Vector<real> Fuente;
Vector<real> ResiduoMasico;
...
```

5. Se define la malla como un `Puntero` a `MallaFV` de manera que solo es necesario generar la malla una vez y acceder a ella mediante punteros. Para calcular la geometría de la malla se crea `elems` de tipo `Vector_basico<PunteroElmDefs>` . A continuación, se declaran los campos que van a ser exportados a Paraview.

```
...
Puntero<MallaFV> malla;
Vector_basico <PunteroElmDefs> elems;
Puntero<CampoFVM>     campo_T;
Puntero<CampoFVM>     campo_u;
Puntero<CampoFVM>  campo_residuo;
Puntero<CampoFVM>  campo_pres;
Puntero<CampoFVM>  campo_pres_prima;
Puntero<CampoFVM>  campo_corr_vel;
Puntero<CamposFVM>    campo_vel;
Puntero<CamposFVM>    campo_GTau ;
Puntero<CamposFVM>    campo_GU, campo_GV, campo_GW;
Puntero<CamposFVM>  campo_GP;
Puntero<CamposFVM>  campo_Q;
Puntero<GradoLibertadFV> gdl;
...
```

6. Posteriormente se declaran las propiedades del material, la densidad `rho`, la conductividad térmica `k`, el calor específico `cp` y la viscosidad `mu`.

```
...
Vector<real>  rho;
Vector<real>  k;
Vector<real>  cp;
Vector<real>  mu;
real Rgas;
...
```

7. Se definen los parámetros del tiempo `theta`, `tiempo_p` y `t_aux`; `t_aux` se usa para no exportar los resultados a través del tiempo y solo mostrar los resultados finales en caso que el análisis sea en estado transitorio. Se declara `Nitera` para almacenar el número límite de iteraciones exteriores. La variable `MAssThreshold` de tipo `real` se usará para comparar

la norma del `ResiduoMasico` y determinar si la solución encontrada cumple la ecuación de continuidad.

```
...
real theta;
Puntero<prmTiempo> tiempo_p;
Puntero<prmTiempo> t_aux;
int Nitera;
real MAssThreshold;
Cadena MetodoSolveNS;
Cadena MetodoAprox;
...
```

Consecutivamente, las variables de tipo `Cadena` `MetodoSolveNS` y `MetodoAprox` contienen respectivamente la información del método de solución de las ecuaciones de Navier-Stokes y el método de discretización seleccionado para las ecuaciones de convección-difusión.

8. A continuación se declaran una serie de variables que almacenan otros parámetros necesarios para la ejecución de la clase.

```
...
int nvol;
int ndim;
real minP, maxP;
real Subrelajacion;
real TrefparaDensidad;
real Patm;
bool densidad_kte;
bool leido;
bool solucionado;
bool CargarDatos;
bool minPres, maxPres;
...
```

Paso 2. Agregar los métodos a la clase: la declaración de cada uno de los métodos se muestra en los siguientes fragmentos de código.

1. Debido a la cantidad de vectores a dimensionar e inicializar, se declaran las funciones `DimensionarVectoresMatrices()` y `Condiciones_iniciales()` para realizar estas dos acciones. El método `CalcPresRef()` se usa para modificar el valor de la presión de manera que la presión mínima o la presión máxima sea el valor predefinido.

```
...
void Condiciones_iniciales();
```

```
void DimensionarVectoresMatrices();
void CalcPresRef();
real DensidadfPyT (int nvol);
...
```

2. La función `CalculoUnCiclo()` realiza la llamada a los métodos correspondientes de acuerdo al esquema de solución seleccionado. La función `CuartoPasoSIMPLE(...)` es un proceso común para calcular la ecuación de la energía que, para el caso de densidad y propiedades constantes no es necesario realizar su cálculo en cada iteración exterior, razón por la que se introduce el parámetro de control `forzar`.

```
...
protected:
virtual int CalculoUnCiclo();
bool PrimerPasoSIMPLE();
bool PrimerPasoNOSIMPLE();
bool SegundoPasoSIMPLE();
bool SegundoPasoNOSIMPLE();
bool SegundoPasoSIMPLEC();
bool TercerPasoSIMPLE();
bool TercerPasoNOSIMPLE();
bool TercerPasoSIMPLEC();
bool CuartoPasoSIMPLE(bool forzar = false);
...
```

3. Para el caso que se desee continuar con las iteraciones exteriores, se puede llamar a `CargadoDeDatos()` el cual se encarga de dar una segunda alternativa de inicialización de los vectores del problema. Para facilitar el cálculo de las operaciones con $\bigtriangledown$ se definen `CalcularTauyGVs()` y `CalcularPyG()`. `CalcularTauyGVs()` realiza el cálculo de $\bigtriangledown u$, $\bigtriangledown v$, $\bigtriangledown w, \bigtriangledown \cdot \vec{V}$, llena el tensor de esfuerzos y calcula la operación $\bigtriangledown \tau$. Mediante la función `CalcularPyG()` se calcula el gradiente de la presión. La siguiente declaración es `ActualizarPyTau()` la cual se encarga de asignar los valores recién calculados a las variables correspondiente del pasado.

```
...
bool CargadoDeDatos();
bool CalcularTauyGVs();
bool CalcularPyG();
bool ActualizarPyTau ();
virtual int SolucionUnTiempo();
virtual void Reportarresultados();
bool LeerFronteras
  (Cadena TomarFrontera, Vector_basico<Cadena> &fr1);
```

```
bool LeerFronteras (Vector_basico<Cadena> &,
  Vector_basico<Cadena> &, Vector_basico<Cadena> &);
real LeerFrontEsp(int i, int j, real muestra, Cadena fr );
...
```

A continuación, se declaran `SolucionUnTiempo()` y `Reportarresultados()` . Se conserva la declaración de `SolucionUnTiempo()` pero en este caso no se realiza la solución de las ecuaciones. Por otro lado, el método `Reportarresultados()` guarda los resultados en un formato compatible con Paraview.

La declaración de `LeerFronteras` y su sobrecarga obedece a la necesidad de un procedimiento común para obtener la condición de frontera de una propiedad específica.

4. Los siguientes métodos son públicos, se declara la función `Leer()` encargada de tomar algunos de los datos iniciales del problema y, se declara `Calcular()` como el procedimiento que calcula la solución del problema a partir de los parámetros anteriormente indicados.

```
...
public:
NavierStokes(void);
~NavierStokes(void) {}
int Leer(Cadena archivo,Cadena solucionador,
  Cadena def_tiempo, real dep_futuro=1.0 ,
  Cadena solucionadorTemp = "Igual");
int Calcular ();
...
```

5. Cálculo de operaciones matemáticas: los siguientes procedimientos son usados para definir cada una de las formas de utilizar el operador ∇, algunos de los datos se toman por referencia para evitar el gasto de tiempo al crear las copias de los vectores y para entregar el resultado de las operaciones.

```
...
bool NablaEsc ( Vector<real> &Entrada,
 Vector_basico< Vector_xyz <real>> &Salida, Cadena Frontera = "Ninguna");
bool NablaDotVec ( Vector_basico< Vector_xyz <real>> &Entrada,
  Vector<real> &Salida, Cadena Frontera = "Ninguna");
bool NablaDotVecEspecial ( Vector_basico< Vector_xyz <real>> &Entrada,
   Vector<real> &Salida, bool MultRho = false);
bool TensorPorVector_xyz (Matriz<real> Me,
  Vector_xyz<real> Ve, Vector_xyz<real> &Salida);
bool NablaPorMatriz ( Vector_basico< Matriz <real>> &Entrada,
```

```
   Vector_basico<Vector_xyz<real>> &Salida);
Matriz<real> PromMatriz(Matriz<real> E1, real , Matriz<real> E2, real );
...
```

Se define la función `GuardarDatos()` más una sobrecarga para almacenar en archivos de texto los valores necesarios para continuar en otro momento la solución del problema. A continuación se declaran varios métodos para modificar el valor de los parámetros de control.

```
...
bool GuardarDatos();
bool GuardarDatos(real minvel,real maxvel);
bool SetCargarDatos(bool opcion);
void SetNoIterExt(int set);
bool SetTipoMetodoSolucion(Cadena tipo);
bool SetEsquemaConveccion(Cadena tipo);
void SetMinimaPresion(real min);
void SetMaximaPresion(real max);
void SetTrefyDensidad(real tt, bool densidad_constante);
void SetPresAtmosf(real pp);
void ResetMinMaxPresion ();
bool SetMAssThreshold(real MaxResMas);
void SubRelajaSIMPLE(real);

...
```

6. Adicionalmente a la función `Leer()` que toma los datos más esenciales para solucionar el problema, se define la función `SetParametros(...)` en la cual se modifican los procedimientos a usar para calcular y solucionar el problema.

```
...
bool SetParametros (bool Datos_iniciales_externos = false,
  int N_iteraciones = 100, Cadena Esquema_conveccion = "UDS" ,
  Cadena Esquema_solucion ="SIMPLE", bool Resultados_indep_tiempo=true);
...
};
```

Definición de las funciones: a continuación se muestra el código que se ejecuta al realizar la llamada de cada uno de los métodos anteriormente declarados.

1. El constructor: el constructor sin argumentos da valores por defecto a algunas de las variables de la clase.

```
NavierStokes::NavierStokes(void)
{
 leido = solucionado = CargarDatos = minPres = maxPres = false;
 ResEstable = true;
 Nitera = 1000; MAssThreshold = 1e-10;
 MetodoSolveNS = "SIMPLE"; MetodoAprox = "UDS";
 Rgas = 0.2870;  densidad_kte = true;
 TrefparaDensidad = 273.0; Patm = 101325;
 t_aux.vincular(new prmTiempo()); t_aux->Leer("dt=0");
}
```

2. Método `Leer(..)`: este método toma como argumentos el nombre del archivo que contiene la definición del problema, el nombre del archivo que contiene el método de solución del problema, la definición del tiempo en el cual se realizará el análisis, ya sea en estado transitorio o en estado estable, por último, el valor de θ que define la dependencia del futuro y del presente para calcular las propiedades.

```
...
int NavierStokes::Leer(Cadena archivo,Cadena solucionador,
   Cadena def_tiempo, real dep_futuro , Cadena solucionadorTemp)
{
int i;
...
```

I. La creación de la malla se realiza con la llamada a `Leer()` de `ConvVu`, no sin antes llamar a `SetPropiedadesExternas(true)`. Debido a que `Leer(...)` no toma como parámetro el nombre de la condición de frontera de la propiedad que va a calcular, se debe llamar a `CadenaFrontera(...)` con el argumento `"VelU"`. Posteriormente, se toma el puntero de la malla generada por `ConvVu` y se asigna a `malla` de la clase `NavierStokes`, también se extrae el puntero del objeto `prmTiempo`.

```
...
std_o<<"\n:::::::Malla Velocidad U:::::::::\n";
ConvVu.SetPropiedadesExternas(true);
ConvVu.Leer (archivo ,solucionador,def_tiempo,0);
ConvVu.CadenaFrontera("VelU");
ConvVu.GetMalla(malla);
ndim = malla->obtNoDimEspacio();
nvol = malla->obtNoElem();
tiempo_p = ConvVu.GetTiempo();
...
```

II. Para las demás instancias de `ConveccionTransitoria` se sigue un procedimiento diferente. Primero se ejecuta `SetPropiedadesExternas(true)` y posteriormente se

llama a `SetMallaeIniciar(...)`. `SetMallaeIniciar(...)` toma como primer parámetro el `Puntero` de la `MallaFV`.

```
...
std_o<<"\n::::::::Malla Velocidad V::::::::\n";
ConvVv.SetPropiedadesExternas(true);
ConvVv.SetMallaeIniciar(malla,solucionador,tiempo_p,"VelV");
if (ndim==3){
std_o<<"\n:::::::::Malla Velocidad W::::::::\n";
ConvVw.SetPropiedadesExternas(true);
ConvVw.SetMallaeIniciar(malla,solucionador,tiempo_p, "VelW");
}
...
```

III. Para iniciar la instancia de `DifusionTransitoria` que calcula la correción de presión se llama primero a `SetPropiedadesExternas` y posteriormente a `SetMallaeIniciar`. La ecuación de corrección de presión se resuelve siempre en estado estable, es por esto que la definición del tiempo se realiza mediante la cadena `"dt=0"`.

```
...
std_o<<"\n::::::::Malla Correcion Presion::::::::\n";
Pprima.SetPropiedadesExternas(true);
Pprima.SetMallaeIniciar(malla,solucionador,"dt=0", "P");
...
```

Se define una corrección de la velocidad que posteriormente se explicará:

```
std_o<<"\n::::::::Malla Correccion v::::::::\n";
Corrv.SetPropiedadesExternas(true);
Corrv.SetMallaeIniciar(malla,solucionador,t_aux,"Vcor");
```

IV. La instancia `Temperatura` del tipo `ConveccionTransitoria` se inicia de un modo similar al mostrado para `ConvVv` o `ConvVw`.

```
...
std_o<<"\n::::::::Malla Energia::::::::\n";
Temperatura.SetPropiedadesExternas(true);
Temperatura.SetMallaeIniciar(malla,solucionador,tiempo_p,"Temp");
...
```

V. A continuación se realiza la inicialización de cada uno de los punteros de campo de volumenes finitos.

```
...
campo_T.vincular(new CampoFVM(malla(),"Temperatura"));
campo_pres.vincular(new CampoFVM(malla(),"Presion"));
campo_pres_prima.vincular(new CampoFVM(malla(),"CorreccionPresion"));
```

```
campo_corr_vel.vincular(new CampoFVM(malla(),"CorrVelV"));
campo_vel.vincular(new CamposFVM(malla(),"Velocidad"));
campo_u.vincular (new CampoFVM(malla(),"Velocidad_U"));
campo_residuo.vincular (new CampoFVM(malla(),"Residuo_Masico"));
campo_GTau.vincular (new CamposFVM(malla(),"GradTau"));
campo_GU.vincular (new CamposFVM(malla(),"GU"));
campo_GV.vincular (new CamposFVM(malla(),"GV"));
campo_GW.vincular (new CamposFVM(malla(),"GW"));
campo_GP.vincular (new CamposFVM(malla(),"GP"));
campo_Q.vincular (new CamposFVM(malla(),"Qcond"));
gdl.vincular(new GradoLibertadFV(malla(),1));
...
```

VI. En esta oportunidad se proporciona una serie de parámetros para modificar la forma como se exportan los datos a Paraview.

```
...
Is iss("ARCH=Infomacion_grafica.txt");
GuardarEnsight::Leer(iss, ndim);
theta = dep_futuro;
...
```

El archivo ``Infomacion_grafica.txt'' al cual se hace referencia contiene la siguiente información:

```
Parametros para Guarda para visualizacion
>puntos_tiempo = [0:10,0.01]
>linea1:inicio = NONE
>linea1:parada = NONE
>linea1:resolucion = 31
>linea2:inicio = NONE
>linea2:parada = NONE
>linea2:resolucion = 31
>linea3:inicio = NONE
>linea3:parada = NONE
>linea3:resolucion = 31
>series_tiempo = d=2 n=0 <
>formato_campo =ASCII
>formato_malla = ASCII
>verbose = OFF
```

VII. Debido a la cantidad de vectores que se deben dimensionar, se proporciona una función específica para esta tarea. Una vez se han dimensionado los vectores, se procede a realizar el llenado de estos, se conserva en la función `Leer(...)` el llenado de `elems` y `Volumen` a causa de que estos vectores dependen de la malla y no son propiamente condiciones iniciales.

```
...
DimensionarVectoresMatrices();
for (i=1; i<=nvol;i++)  {
elems(i).refill( malla->obtTipoElem(i), malla->obtNoDimEspacio() );
elems(i)->CalcGeomParameters( malla->obtCoorElem(i) );
}
for (i=1; i<=nvol; i++) Volumen(i) = elems(i)->getVolume();
Condiciones_iniciales();
leido = true;
return 1;
}
```

Se finaliza la ejecución de `Leer(...)` con la modificación del parámetro de control y la devolución del valor 1.

3. Método `DimensionarVectoresMatrices()`: se define un procedimiento encargado de dimensionar cada uno de los vectores según sea el caso. Para las variables de tipo `Vector<real>` como `VelU` es suficiente con llamar a `chaSize(nvol)` en cambio, para dimensionar elementos del tipo `Vector_basico<Vector_xyz<real>>` como `GU` o `Vector_basico<Matriz<real>>` como `Tau` adicionalmente se debe llenar cada uno de los elementos del vector básico, razón por la cual es necesario utilizar un bucle repetitivo `for(i=1;i<=nvol;i++)`.

```
void NavierStokes::DimensionarVectoresMatrices()
{
 int i;
 CoefU.chaSize(nvol);  CoefV.chaSize(nvol);  CoefW.chaSize(nvol);
 VmAstU.chaSize(nvol); VmAstV.chaSize(nvol); VmAstW.chaSize(nvol);
 VelU.chaSize(nvol); VelU_ant.chaSize(nvol);
 VelV.chaSize(nvol); VelV_ant.chaSize(nvol);
 if (ndim==3) { VelW.chaSize(nvol); VelW_ant.chaSize(nvol); }
 GVel.chaSize(nvol); GVel_ant.chaSize(nvol); GVel_n.chaSize(nvol);
 Presion_ant.chaSize(nvol); Presion.chaSize(nvol);
 Presion_n.chaSize(nvol); Presionprima.chaSize(nvol);
 Fuente.chaSize(nvol); ResiduoMasico.chaSize (nvol);
 Tau.chaSize(nvol); Tau_n.chaSize(nvol); Tau_ant.chaSize(nvol);
 Gtau.chaSize(nvol); Fcalor.chaSize(nvol);
 GU.chaSize (nvol); GU_ant.chaSize(nvol);
 GU_n.chaSize (nvol); GV_n.chaSize (nvol);
 GV.chaSize (nvol); GV_ant.chaSize(nvol);
 if (ndim == 3 )
  {GW.chaSize (nvol);GW_ant.chaSize(nvol);
  GW.chaSize (nvol); }
 GP.chaSize (nvol); GP_ant.chaSize(nvol);
```

```
 GP_n.chaSize (nvol);
 for (i=1; i <= nvol; i++) {
  GU(i).chaSize(ndim); GU_ant(i).chaSize(ndim);
  GU_n(i).chaSize(ndim);GV_ant(i).chaSize(ndim);
  GV(i).chaSize(ndim); GV_n(i).chaSize(ndim);
  if (ndim == 3 )
   {GW(i).chaSize(ndim);GW_ant(i).chaSize(ndim);
   GW_n(i).chaSize(ndim);}
  GP(i).chaSize(ndim); GP_n(i).chaSize(ndim);
   GP_ant(i).chaSize(ndim);
  Tau(i).chaSize(ndim);Tau_n(i).chaSize(ndim);
  Tau_ant(i).chaSize(ndim); Gtau(i).chaSize(ndim);
  Fcalor(i).chaSize(ndim);
  }
 Temp.chaSize(nvol);  Temp_ant.chaSize(nvol);
 rho.chaSize(nvol); k.chaSize(nvol);
 cp.chaSize(nvol); mu.chaSize(nvol);
 Volumen.chaSize(nvol);
 elems.chaSize(nvol);
}
```

4. Método `Condiciones_iniciales()`: se asigna un valor a las propiedades del material y a la velocidad inicial, adicionalmente se realiza el llenado de otros vectores con el valor 0. Se agrega el condicional `if(CargarDatos)` para tener la opción de cargar las condiciones iniciales a partir de los archivos generados de una iteración anterior o realizar un procedimiento normal de inicialización de las variables.

```
void NavierStokes::Condiciones_iniciales()
{
 int i;
 if (CargarDatos) {CargadoDeDatos();}
 else {
  rho.llenar(1.172); k.llenar(0.02566);
  cp.llenar(1007);  mu.llenar(1.858e-5);
  VelU.llenar(1.0); VelU_ant = VelU;
  VelV.llenar(0.00);VelV_ant = VelV;
  VelW.llenar(0.0); VelW_ant = VelW;
  Presion.llenar(0.0);
  Presion_ant = Presion_n= Presion;

  for (i=1; i<= nvol; i++) {
   GU_ant(i).llenar(0.0);GU_n(i).llenar(0.0);
   GV_ant(i).llenar(0.0);GV_n(i).llenar(0.0);
```

```
   if (ndim == 3 )
    {GW_ant(i).llenar(0.0); GW_n(i).llenar(0.0);}
   GP_n(i).llenar(0.0);GP_ant(i).llenar(0.0);
   Tau_n(i).llenar(0.0);Tau_ant(i).llenar(0.0);
   Fcalor(i).llenar(0.0);
  }
  GVel.llenar(0.0);
  Temp.llenar(0.0);Temp_ant.llenar(0.0);
 }
 VmAstU = VelU; VmAstV = VelV;
 if (ndim == 3) VmAstW= VelW;
 Presionprima = Presion; vcor.llenar(0.0);
}
```

5. Método `Calcular()`: antes de definir el procedimiento que realiza las iteraciones exteriores se verifica el parámetro de control `leido`, se declaran e inicializan algunas variables y se llama al método `SetPasadoPartida(...)` de `ConveccionTransitoria` y `DifusionTransitoria`, es importante realizar esta llamada para proporcionar el valor de las propiedades en el tiempo anterior y un punto de partida para la solución de cada una de las ecuaciones lineales.

```
int NavierStokes::Calcular()
{
if (!leido) return 0;
bool NIterAlc = false, convergio = false;
int m=0; real norma; Cadena temm;
Vector<real> uno(nvol); uno.llenar(1.0);
Vector<real> cero(nvol); cero.llenar(0.0);
solucionado = true;
tiempo_p->iniciarCicloTiempo(); t_aux->iniciarCicloTiempo();
ConvVu.SetPasadoPartida(VelU_ant,uno);
ConvVv.SetPasadoPartida(VelV_ant,uno);
if (ndim == 3)
ConvVw.SetPasadoPartida(VelW_ant,VmAstW);
Pprima.SetPasadoPartida(Presionprima,cero);
Corrv.SetPasadoPartida(VelV_ant,uno);
Temperatura.SetPasadoPartida (Temp_ant,cero);
...
```

La solución de las ecuaciones de Navier-Stokes por el método SIMPLE se puede realizar en estado estable o en estado transitorio. A continuación se muestra el procedimiento a realizar para un análisis en estado estable. Básicamente, se realizan los cálculos de cada

uno de los pasos en la llamada a `CalculoUnCiclo()` y se calcula la norma del residuo másico la cual servirá como uno de los criterios de convergencia de las iteraciones, el segundo criterio de parada contempla el número máximo de iteraciones permitido, valor que se había almacenado en `Nitera`.

```
...
if (tiempo_p->estacionario())
{
 theta= 1.0 ;
do  {
m++;
CalculoUnCiclo();
norma = ResiduoMasico.norma();
std_o<<" RM norma "<<norma<<" ";
if (m>=Nitera)
{std_o<<"\nNo.limite iteraciones alcanzado\n";NIterAlc=true;}
if ( norma<MAssThreshold)
convergio = true;
std_o<<" Iteracion exterior No. "<<m<<" \n\n\n";
} while (!convergio && !NIterAlc);
if (ResEstable) CuartoPasoSIMPLE(true);
SolucionUnTiempo();
Reportarresultados();
}
...
```

Para el caso transitorio se tiene el tiempo máximo de simulación como criterio adicional de parada, si se excede este valor no se continuará con el análisis. Cabe agregar que la forma cómo se calcula el estado transitorio no permite obtener exactamente el resultado del análisis a través del tiempo, la razón de esto es que solo se realiza una iteración exterior por cada avance del tiempo, los resultados siempre serán aproximados para un tiempo intermedio ya que no se cumplen al mismo tiempo con las ecuaciones de la cantidad de movimiento y de continuidad. A partir de este hecho, se propone la creación del parámetro de control `ResEstable` en el cual se indica si se quiere realizar el reporte de los resultados en estado estable a pesar que el análisis haya sido realizado en estado transitorio.

```
 ...
else {
 if (!ResEstable) {
   SolucionUnTiempo();
   Reportarresultados();
  }
  m=0;
```

```
  do {
   tiempo_p->incrementarTiempo();
   m++;
   CalculoUnCiclo();
   norma = ResiduoMasico.norma();
   std_o<<" RM norma "<<norma<<" ";
   if (m>=Nitera)
    NIterAlc=true;
   if ( norma<MAssThreshold)
    {convergio = true; NIterAlc = false;}
   std_o<<" Iteracion exterior No. "<<m<<" \n\n\n";
   ActualizarPyTau();
   if (!ResEstable) {
    SolucionUnTiempo();
    Reportarresultados();
   }
   } while (!tiempo_p->finalizo() && !convergio && !NIterAlc);
  if (ResEstable) {
   CuartoPasoSIMPLE(true);
   SolucionUnTiempo();
   Reportarresultados();
  }
 }
 solucionado=true;
 return 1;
}
```

Para exportar los resultados en cada tiempo es necesario llamar a `SolucionUnTiempo()` y `Reportarresultados()`. Posteriormente se explicará la función de cada uno de estos dos métodos. Para el análisis transitorio también se debe llamar a `ActualizarPyTau()` que se encarga de pasar los valores recientemente calculados a las variables del tiempo anterior.

6. Método `CalculoUnCiclo()`: la verificación del método a seguir para resolver las ecuaciones de NavierStokes se realiza en el siguiente fragmento de código, de esta manera, resulta sencillo utilizar otros métodos para resolver las ecuaciones.

```
int NavierStokes::CalculoUnCiclo()
{
int i;
if (MetodoSolveNS=="SIMPLE")
{PrimerPasoSIMPLE();SegundoPasoSIMPLE();
TercerPasoSIMPLE();CuartoPasoSIMPLE();
}
```

```
if (MetodoSolveNS=="NOSIMPLE") {
PrimerPasoNOSIMPLE();
std_o<<"VelV Min "<<VelV.minValor()<<" Max "<<VelV.maxValor()<<" \n";
SegundoPasoNOSIMPLE();
TercerPasoNOSIMPLE();
CuartoPasoSIMPLE();
}
if (MetodoSolveNS=="SIMPLEC")
{ PrimerPasoSIMPLE();SegundoPasoSIMPLEC();
TercerPasoSIMPLE();CuartoPasoSIMPLE();
}
CalcularTauyGVs();
CalcularPyG();
return 1;
}
```

Al final de cada ciclo se debe calcular el gradiente de cada una de las componentes de la velocidad y la presión entre otros términos, para realizar este trabajo se llama a `CalcularTauyGVs()` y `CalcularPyG()`.

7. Método `CalcularTauyGVs()`: realiza el cálculo de todas las operaciones necesarias que están relacionadas con el operador $\bigtriangledown$ y la velocidad.

I. El primer paso a realizar es la declaración de variables. Se declara `VelVec` como `Vector_basico<Vector_xyz<real>>` y se inicializa con las componentes de la velocidad para posteriormente calcular $\bigtriangledown \cdot \vec{V}$.

```
bool NavierStokes::CalcularTauyGVs(  )
{
int i;
Vector_basico<Vector_xyz<real>> VelVec;
VelVec.chaSize(nvol);
for (i=1; i <= nvol; i++) {
  VelVec(i).chaSize(ndim);
  VelVec(i)(1)= VelU(i);
  VelVec(i)(2)= VelV(i);
  if (ndim == 3 )
   VelVec(i)(3)= VelW(i);
}
....
```

II. El siguiente paso es calcular $\bigtriangledown u$, $\bigtriangledown v$ y $\bigtriangledown w$ mediante la llamada a `NablaEsc(...)`, el último parámetro que se entrega es una `Cadena` la cual permite identificar la condición de frontera que se debe seleccionar. En este fragmento de código también se puede ver la instrucción `NablaDotVec(...)` que se encarga de calcular $\bigtriangledown \cdot \vec{V}$.

```
...
NablaEsc (VelU, GU_n, "VelU");
if (theta!=1.0) NablaEsc (VelU_ant, GU_ant, "VelU");
NablaEsc (VelV, GV_n, "VelV");
if (theta!=1.0) NablaEsc (VelV_ant, GV_ant, "VelV");
if (ndim == 3) NablaEsc (VelW, GW_n, "VelW");
if (theta!=1.0 && ndim == 3) NablaEsc (VelW_ant, GW_ant, "VelW");
NablaDotVec(VelVec,GVel_n, "Vel");
...
```

III. Los valores calculados en el anterior fragmento de código fueron almacenados en las variables terminadas con `"*_n''`. Para calcular los valores que realmente van a ser utilizados, se debe tener en cuenta el parámetro `theta` que define la dependencia del tiempo futuro e indirectamente la dependencia del tiempo actual.

```
...
for (i=1; i<= nvol; i++) {
GVel(i) = (GVel_n(i))*theta + (GVel_ant(i))*(1-theta);
GU(i) = (GU_n(i)) * theta + (GU_ant(i))*(1-theta);
GV(i) = (GV_n(i)) * theta + (GV_ant(i))*(1-theta);
if (ndim == 3)
GW(i) = (GW_n(i)) * theta + (GW_ant(i))*(1-theta);
}
...
```

IV. El llenado del tensor τ o `Tau` se realiza en las siguientes instrucciones. También se tiene en cuenta el cálculo en el tiempo futuro `Tau_n` y en el tiempo presente `Tau_ant` para realizar la asignación a `Tau`.

```
...
for (i=1; i <= nvol; i++) {
  Tau_n(i)(1,1) = (-2.0* mu(i) / 3.0) * GVel_n(i) +
      (1.0*mu(i)*GU_n(i)(1));
  Tau_n(i)(1,2) = Tau_n(i) (2,1) =  mu(i)*(GU_n(i)(2)+GV_n(i)(1));
  Tau_n(i)(2,2) = (-2.0* mu(i) / 3.0) * GVel_n(i) +
      (1.0*mu(i)*GV_n(i)(2));
  if (ndim==3) {
   Tau_n(i)(3,3) = (-2.0* mu(i) / 3.0) * GVel_n(i) +
      (1.0*mu(i)*GW_n(i)(3));
   Tau_n(i)(1,3) = Tau_n(i) (3,1) = mu(i)*(GU_n(i)(3)+GW_n(i)(1));
   Tau_n(i)(2,3) = Tau_n(i) (3,2) = mu(i)*(GV_n(i)(3)+GW_n(i)(2));
  }
  Tau(i)(1,1) = Tau_n(i)(1,1)  *theta + (1-theta) * Tau_ant(i)(1,1);
  Tau(i)(1,2) = Tau(i)(2,1) = Tau_n(i)(1,2)  *theta +
     (1-theta) * Tau_ant(i)(1,2);
```

```
    Tau(i)(2,2) = Tau_n(i)(2,2)  *theta + (1-theta) * Tau_ant(i)(2,2);
    if (ndim==3)  {
     Tau(i)(3,3) = Tau_n(i)(3,3)  *theta + (1-theta) * Tau_ant(i)(3,3);
     Tau(i)(1,3) = Tau(i)(3,1) = Tau_n(i)(1,3)  *theta +
        (1-theta) * Tau_ant(i)(1,3);
     Tau(i)(2,3) = Tau(i)(3,2) = Tau_n(i)(2,3)  *theta +
        (1-theta) * Tau_ant(i)(2,3);
    }
   }
  ...
```

V. Finalmente con una sencilla instrucción se calcula $\nabla\tau$ y seguidamente se finaliza la ejecución de la función al entregar como resultado el valor `true`.

```
 ...
 NablaPorMatriz(Tau,Gtau);
 return true;
 }
```

8. Método `CalcularPyG()`: se calcula el gradiente de la presión en el futuro y en el presente cuando sea necesario, adicionalmente se calculan `Presion` y `GP` que son los valores a usar a través de cada uno de los pasos.

```
bool NavierStokes::CalcularPyG()
{
int i;
if (theta !=0.0 ) NablaEsc (Presion_n, GP_n,"P");
if (theta != 1.0) NablaEsc (Presion_ant, GP_ant, "P");
for (i=1; i<=nvol ; i++)
Presion(i) = Presion_n(i)*theta + Presion_ant(i)*(1.0-theta);
for (i=1; i<=nvol ; i++)
GP(i) = GP_n(i)*theta + GP_ant(i)*(1.0-theta);
return true;
}
```

9. Método `CalcQcalor()`: se realiza el cálculo del flujo de calor por conducción que permitirá mostrar el flujo de calor hacia la placa.

```
bool NavierStokes::CalcQcalor()
{
int i;
NablaEsc(Temp,Fcalor,"Temp");
for (i=1; i<=nvol; i++)
Fcalor(i) = Fcalor(i)*k(i)*(-1.0);
```

```
return true;
}
```

10. Método `ActualizarPyTau()`: en el análisis transitorio, se debe actualizar los valores de las propiedades en el presente por los valores del futuro para continuar con el siguiente tiempo, esta es la función de `ActualizarPyTau()`.

```
bool NavierStokes::ActualizarPyTau()
{
int i;
VelU_ant = VelU; VelV_ant = VelV;
if (ndim == 3)   VelW_ant = VelW;
for (i=1; i<=nvol ; i++)
{
 GU_ant(i) = GU_n(i);   GV_ant(i) = GV_n(i);
 if (ndim == 3)     GW_ant(i) = GW_n(i);
 GVel_ant(i)=GVel_n(i); Tau_ant(i) =Tau_n(i);
}
Presion_ant = Presion_n;
for (i=1; i<=nvol ; i++) GP_ant(i) = GP_n(i);
Temp_ant = Temp ;
return true;
}
```

11. Método `PrimerPasoSIMPLE()`: en este procedimiento se resuelven las ecuaciones de la cantidad de movimiento de acuerdo a como se presentó en el primer paso del método SIMPLE. El procedimiento seguido para cada una de las componentes es muy similar.

I. En primer lugar, se declaran las variables y se dimensionan los vectores.

```
bool NavierStokes::PrimerPasoSIMPLE()
{
bool correcto; int i;
Vector<real>  dPdn(nvol),  EfVisc(nvol);
...
```

II. Antes de asignar los coeficientes a la ecuación de cantidad de movimiento en x, se debe calcular el término fuente como se muestra en la tabla 13. El término fuente está compuesto por el efecto del gradiente de la presión y el efecto del gradiente de la viscosidad. Una vez se realiza la suma algebraica, se multiplica por el volumen de cada VC y se procede a ejecutar `ConvVu.SetFuenteYFlujo(...)` la cual introduce el valor del término fuente y las componentes de la velocidad, seguidamente se asigna el valor de la propiedad ϕ en el tiempo pasado (`VelU_ant`) y se actualiza el valor de las propiedades del material. Para asignar $\Gamma = \mu$ en la clase `ConveccionTransitoria` se debe llamar a la versión sobrecargada de `SetPropiedades` de dos parámetros,

```
...
Fuente.llenar( 0.0 );
for (i=1; i<=nvol; i++) {dPdn (i) = GP (i)(1); }
Fuente -= dPdn;
for (i=1; i<=nvol; i++) {EfVisc(i) = Gtau(i)(1);}
Fuente += EfVisc;
for (i=1; i<=nvol; i++) Fuente(i) = Fuente(i) * Volumen(i);
ConvVu.SetFuenteYFlujo(Fuente,VelU,VelV, VelW);
ConvVu.SetPasado(VelU_ant);
ConvVu.SetPropiedades (rho, mu);
CoefU  = ConvVu.CalcularCoefUnTiempo
               (theta, tiempo_p, MetodoAprox,  correcto);
VmAstU = ConvVu.SolucionCoefUnTiempo (CargarDatos,correcto);
```

En este punto ya es posible calcular la matriz de coeficientes y el vector de términos independientes, la llamada a `CalcularCoefUnTiempo(...)` devuelve el valor de los elementos en la diagonal principal y modifica el valor de la variable de tipo `bool` en el último parámetro para indicar si el proceso terminó correctamente. Por último, para la ecuación de la cantidad de movimiento en x, se resuelve el sistema de ecuaciones y se obtiene el valor de u^{m*} representado por la variable `VmAstU`.

III. Se sigue un procedimiento similar para las demás componentes de la velocidad, se debe seleccionar el valor correcto del gradiente de la presión y del gradiente del tensor de esfuerzos. El procedimiento se muestra a continuación:

```
Fuente.llenar(0.0);
for (i=1; i<=nvol; i++) {dPdn (i) = GP (i)(2); }
Fuente -= dPdn;
for (i=1; i<=nvol; i++) {EfVisc(i) = Gtau(i)(2);}
Fuente += EfVisc;
for (i=1; i<=nvol; i++) Fuente(i) = Fuente(i) * Volumen(i);
ConvVv.SetFuenteYFlujo(Fuente,VelU,VelV, VelW);
ConvVv.SetPasado(VelV_ant);
ConvVv.SetPropiedades (rho, mu);
CoefV=ConvVv.CalcularCoefUnTiempo
           (theta, tiempo_p,MetodoAprox, correcto);
VmAstV = ConvVv.SolucionCoefUnTiempo (CargarDatos,  correcto );
if (ndim == 3 ) {
 Fuente.llenar(0.0);
 for (i=1; i<=nvol; i++) {dPdn (i) = GP (i)(3); }
 Fuente -= dPdn;
 for (i=1; i<=nvol; i++) {EfVisc(i) = Gtau(i)(3);}
 Fuente += EfVisc;
 for (i=1; i<=nvol; i++) Fuente(i) = Fuente(i) * Volumen(i);
 ConvVw.SetFuenteYFlujo(Fuente,VelU,VelV, VelW);
```

```
 ConvVw.SetPropiedades (rho, mu);
 ConvVw.SetPasado(VelW_ant);
 CoefW=ConvVw.CalcularCoefUnTiempo
             (theta, tiempo_p, MetodoAprox, correcto );
 VmAstW = ConvVw.SolucionCoefUnTiempo (CargarDatos,  correcto);
}
return true;
}
```

12. Método `SegundoPasoSIMPLE()`: en este método se calcula y resuelve la ecuación de correción de presión, por la forma como se obtuvo la ecuación, esta siempre se resolverá en estado estable. Si la densidad es variable a través del tiempo, el efecto de la densidad hará parte del término fuente de la ecuación de corrección de presión.

I. Se declaran las variables a utilizar en el método y se dimensionan los vectores. El `Vector<real> cero` se utiliza para resetear el punto de partida para la ecuación de correción de presión.

```
bool NavierStokes::SegundoPasoSIMPLE()
{
bool triunfo; int i;
Vector<real> cero, gammx, gammy, gammz;
Vector_basico<Vector_xyz<real>> Velvec(nvol);
cero.chaSize (nvol); cero.llenar (0.0);
gammx.chaSize (nvol);gammy.chaSize (nvol);
if (ndim == 3) gammz.chaSize (nvol) ;
...
```

II. Se calcula el valor de $\mathbf{\Gamma}$ para la ecuación de corrección de presión como se mostró en la (Ec 6.11). El valor de $\mathbf{\Gamma}$ se calcula en este caso asumiendo que `CoefU` y `CoefV` pueden ser diferentes, en general estos valores serán iguales si o solo si el tipo de condición de frontera para cada velocidad es la misma.

```
...
for ( i=1 ; i<=nvol ; i++) {
gammx (i) = rho(i)*Volumen(i)/ CoefU(i);
gammy (i) = rho(i)*Volumen(i)/ CoefV(i);
if (ndim == 3)
 gammz (i) = rho(i)*Volumen(i)/ CoefW(i);
Velvec(i).chaSize(ndim);
Velvec(i)(1) = VmAstU(i); Velvec(i)(2) = VmAstV(i);
if (ndim == 3)
 Velvec(i)(3) = VmAstW(i);
}
...
```

III. El siguiente paso es calcular el valor de $\nabla \cdot (\rho \vec{V}^{m*})$ llamado `ResiduoMasico`. El presente trabajo se basa en el uso de mallas colocadas y para calcular correctamente el valor de `ResiduoMasico` se debe realizar la correción de la velocidad según se muestra en (Ec 6.15), este procedimiento es ejecutado en `NablaDotVecEspecial(...)`. Una vez se ha calculado este valor, se realiza la integración a través del volumen de control por lo que se multiplica por `Volumen(i)`.

```
...
NablaDotVecEspecial(Velvec,ResiduoMasico, MULT_POR_RHO_ON );
for (i=1;i<=nvol;i++)ResiduoMasico(i) = ResiduoMasico(i)*Volumen(i);
Pprima.SetPropiedades(gammx, gammy,gammz);
Fuente = ResiduoMasico * (-1.0);
Pprima.SetFuente (Fuente);
Presionprima = Presionprima*(0.2);
Pprima.SetPasadoPartida(cero,Presionprima);
Pprima.CalcularCoefUnTiempo(1.0, triunfo);
Presionprima = Pprima.SolucionCoefUnTiempo(triunfo);
return true;
}
```

En los siguientes pasos se asigna el valor de $\mathbf{\Gamma}$ a `Pprima`, se determina el valor del término fuente y se introduce a `Pprima`, se resetea el valor del punto de partida de las iteraciones, se realiza el cálculo de los coeficientes y se resuelve el sistema de ecuaciones, el valor obtenido se almacena en `Presionprima`.

13. Método `TercerPasoSIMPLE()`: en este paso se calcula el valor de la presión y la velocidad final para la iteracion m. Debido a que el método SIMPLE puede presentar un comportamiento oscilatorio se utiliza un parámetro de subrelajación para evitar dicho comportamiento.

I. La primera tarea a realizar es declarar las variables a usar y realizar el dimensionamiento de los vectores. Se destaca la inicialización del parámetro de subrelajación de la velocidad `alpV` como el número decimal que junto con `alpP` suman una unidad, este valor es ampliamente recomendado por ser una forma sencilla de calcular un valor cercano al óptimo para resolver los problemas.

```
bool NavierStokes::TercerPasoSIMPLE()
{
int i;
  real alpP = Subrelajacion;
real alpV = 1.0-alpP;
Vector_xyz<real> InvVec (ndim);
Vector_basico <Vector_xyz<real>> Gradiente (nvol);
Vector<real> uprima(nvol), vprima(nvol), wprima(nvol);
Vector<real> uAst(nvol), vAst(nvol), wAst(nvol);
```

```
for (i=1;i<=nvol;i++) Gradiente(i).chaSize (ndim);
...
```

II. Para continuar el proceso se actualiza el valor de `Presion_n` con el valor de `Presionprima` teniendo en cuenta el parámetro de subrelajación `alpP`. Posteriormente, se calcula el gradiente de la presión y se almacena en `Gradiente`.

```
...
Presion_n += (Presionprima*alpP);
NablaEsc(Presionprima, Gradiente, "P");
```

III. Para cada volumen de control, se debe calcular el inverso del coeficiente de la diagonal principal de la ecuación de la cantidad de movimiento, este valor se debe multiplicar por el volumen y por el término correspondiente del gradiente de la correción de presión, de esta manera se obtiene el valor de la correción de la velocidad. La corrección de la velocidad se debe sumar con $\vec{V}^{m*}$ para obtener la velocidad corregida (`uAst`, ...). Finalmente, se calcula el valor de la velocidad para el ciclo m teniendo en cuenta el valor anterior ($\vec{V}^{m-1}$), el valor de la velocidad corregida y el valor del parámetro de subrelajación de la velocidad `alpV`. El procedimiento se repite para cada una de las componentes de la velocidad.

```
for (i=1; i<= nvol ; i++) {
InvVec(1) = 1 / CoefU(i);
uprima(i) = - Volumen(i) * InvVec(1) *Gradiente(i)(1);
uAst(i) =  VmAstU(i) + uprima(i);
VelU(i) = alpV * uAst(i) + (1.0-alpV) * VelU(i);

InvVec(2) = 1 / CoefV(i);
vprima(i) = -Volumen(i) * InvVec(2) *Gradiente(i)(2);
vAst(i) = VmAstV(i) + vprima(i);
VelV(i) = alpV * vAst(i) + (1-alpV) * VelV(i);

if (ndim == 3) {
InvVec(3) = 1 / CoefW(i);
wprima(i) = -Volumen(i) * InvVec(3)*Gradiente(i)(3);
wAst(i) = VmAstW(i) + wprima(i);
VelW(i) = alpV * wAst(i) - (1.0 - alpV) * VelW(i);
}
}
return true;
}
```

Para finalizar, se entrega el valor `true` como respuesta al segmento del programa que realizó la llamada del método.

14. Método `CuartoPasoSIMPLE`: en esta función se realiza el cálculo de la ecuación de

la energía y se actualiza el valor de la densidad en caso que sea variable. Se incorpora una serie de condicionales que permiten omitir el cálculo del balance de energía en caso que la densidad sea constante o los resultados se presenten en estado estable.

```
bool NavierStokes::CuartoPasoSIMPLE(bool forzar)
{
 bool correcto; int i; Vector<real> cero(nvol); cero.llenar(0.0);
 if ( !densidad_kte || !ResEstable || forzar) {
  Temperatura.SetFuenteYFlujo(cero,VelU,VelV, VelW);
  Temperatura.SetPasado (Temp_ant);
  Temperatura.SetPropiedades(rho,k,cp);
  if (ResEstable && forzar) {
   Temperatura.CalcularCoefUnTiempo
    (theta, t_aux, MetodoAprox, correcto);
  }
  else{
   Temperatura.CalcularCoefUnTiempo
    (theta, tiempo_p, MetodoAprox, correcto);
  }
  Temp = Temperatura.SolucionCoefUnTiempo(correcto, false);
  CalcQcalor();
 }
 if (!densidad_kte) {
  for (i=1; i<=nvol; i++)
   rho(i) = DensidadfPyT(i);
 }
 return true;
}
```

15. Método `SolucionUnTiempo()`: Aunque no es necesario resolver ningún sistema de ecuaciones en este método, se conserva la asignación de los valores hallados al respectivo campo. Antes de enviar los valores correspondientes a los campos se llama a `CalcPresRef()` para modificar el valor mínimo o el valor máximo de la presión. Para introducir los datos a los campos vectoriales es necesario declarar tres vectores adicionales de tipo `Vector<real>`.

```
int NavierStokes::SolucionUnTiempo()
{
 CalcPresRef();
 Vector<real> gtu(nvol), gtv(nvol), gtw(nvol);
 gdl->vector2campo(VelU,campo_u());
  gdl->vector2campo(Presion,campo_pres());
 gdl->vector2campo(Presionprima,campo_pres_prima());
```

```
gdl->vector2campo(vcor,campo_corr_vel());
gdl->vector2campo(Temp,campo_T());
gdl->vector2campo(ResiduoMasico,campo_residuo());
gdl->vector2campo(VelU,campo_vel()(1));
gdl->vector2campo(VelV,campo_vel()(2));
if ( malla->obtNoDimEspacio()==3)
 gdl->vector2campo(VelW,campo_vel()(3));
for (int i=1; i<= nvol; i++) {
 gtu(i)=Gtau(i)(1);
 gtv(i)=Gtau(i)(2);
 if (ndim==3)
  gtw(i)=Gtau(i)(3);
}
gdl->vector2campo(gtu,campo_GTau()(1));
gdl->vector2campo(gtv,campo_GTau()(2));
if (ndim==3)
 gdl->vector2campo(gtw,campo_GTau()(3));

for (int i=1; i<= nvol; i++) {
 gtu(i)=GU(i)(1);
 gtv(i)=GU(i)(2);
 if (ndim==3)
  gtw(i)=GU(i)(3);
}
gdl->vector2campo(gtu,campo_GU()(1));
gdl->vector2campo(gtv,campo_GU()(2));
if (ndim==3)
 gdl->vector2campo(gtw,campo_GU()(3));

for (int i=1; i<= nvol; i++) {
 gtu(i)=GV(i)(1);
 gtv(i)=GV(i)(2);
 if (ndim==3)
  gtw(i)=GV(i)(3);
}
gdl->vector2campo(gtu,campo_GV()(1));
gdl->vector2campo(gtv,campo_GV()(2));
if (ndim==3)
 gdl->vector2campo(gtw,campo_GV()(3));

for (int i=1; i<= nvol; i++) {
 gtu(i)=GP(i)(1);
 gtv(i)=GP(i)(2);
```

```
  if (ndim==3)
   gtw(i)=GP(i)(3);
 }
 gdl->vector2campo(gtu,campo_GP()(1));
 gdl->vector2campo(gtv,campo_GP()(2));
 if (ndim==3)
  gdl->vector2campo(gtw,campo_GP()(3));
 for (int i=1; i<= nvol; i++) {
  gtu(i)=Fcalor(i)(1);
  gtv(i)=Fcalor(i)(2);
  if (ndim==3)
   gtw(i)=Fcalor(i)(3);
 }
 gdl->vector2campo(gtu,campo_Q()(1));
 gdl->vector2campo(gtv,campo_Q()(2));
 if (ndim==3)
  gdl->vector2campo(gtw,campo_Q()(3));
 return true;
}
```

16. Método `Reportarresultados()`: se contemplan dos formas de exportar los resultados de acuerdo al valor de `ResEstable`. La diferencia se encuentra en el uso de 0 o de `tiempo_p` como definición del tiempo del análisis. Para el caso que el análisis sea transitorio y `ResEstable` sea `true` solo se exportará el resultado final y no se exportarán resultados para tiempos intermedios; en cambio, si `ResEstable` es `false`, se guardarán los resultados de acuerdo a como se indicó en el archivo `Infomacion_grafica.txt` en la página 144.

```
void NavierStokes::Reportarresultados()
{
 if (ResEstable)
 {
  volcar(campo_vel(),0);
  volcar(campo_pres(),0);
  volcar(campo_u(),0);
  volcar(campo_residuo(),0);
  if (MetodoSolveNS=="NOSIMPLE")
   {volcar(campo_corr_vel(),0);}
  else { volcar(campo_pres_prima(),0);}
  volcar(campo_GTau(),0);
  volcar(campo_GU(),0);
  volcar(campo_GV(),0);
  volcar(campo_GP(),0);
```

```
  volcar(campo_T(),0);
  volcar(campo_Q(),0);
 }
 else
 {
  volcar(campo_T(),tiempo_p.obtPtr());
  volcar(campo_vel(),tiempo_p.obtPtr());
  volcar(campo_pres(),tiempo_p.obtPtr());
  volcar(campo_u(),tiempo_p.obtPtr());
  if (MetodoSolveNS=="NOSIMPLE")
  {volcar(campo_corr_vel(),tiempo_p.obtPtr());}
  else { volcar(campo_pres_prima(),tiempo_p.obtPtr());}
  volcar(campo_residuo(),tiempo_p.obtPtr());
  volcar(campo_Q(),tiempo_p.obtPtr());
 }
}
```

Según el método de solución escogido, se debe guardar los resultados del campo de corrección de presión o el campo de correción de la velocidad[3].

17. Método `LeerFrontEsp(...)`: se define este procedimiento como una forma común para todos los procesos que necesiten obtener el valor de una propiedad en la frontera siendo `fr` la `Cadena` en la cual se encuentra la información del tipo y valor de la condición de frontera.

```
real NavierStokes::LeerFrontEsp(int i, int j, real muestra, Cadena fr )
{
real valor = muestra, valor2, px, py; Cadena bovalue = fr.despues("=");
 Vector_xyz<real> minc, maxc; malla->obtCoorMinMax(minc, maxc);
 if (fr.contiene("Dirichlet") ) {
  valor   = atof(bovalue.carts());
  px = elems(i)->centroArea(malla->obtCoorElem(i),j)(1);
  py = elems(i)->centroArea(malla->obtCoorElem(i),j)(2);
  if (fr.contiene("e^")) {
   bovalue = fr.despues("e^"); valor2 = atof(bovalue.carts());
   valor = valor * exp( valor2*(px-minc(1)) );
  }
  if (fr.contiene("x^")){
   bovalue = fr.despues("x^"); valor2 = atof(bovalue.carts());
   valor = valor * pow( px-minc(1) ,valor2);
  }
  if (fr.contiene("rooty")){
```

[3]Posteriormente se explicará este método.

```
   bovalue = fr.despues("p=");valor2 = atof(bovalue.carts());
   if (py<valor2){
    valor = valor * sqrt(py/valor2);
    }
  }
  if (fr.contiene("liny")){
   bovalue = fr.despues("p=");valor2 = atof(bovalue.carts());
   if (py<valor2){
   valor = valor * (py/valor2);
   }
  }
 }
 if (fr.contiene("Neumann") ){
  valor = atof(bovalue.carts());
  valor = muestra - valor*elems(i)->getDelta(j);
 }
 return valor;
}
```

18. Método `LeerFronteras(...)` y sobrecarga: se definen dos procedimientos para obtener las condiciones de frontera de la región analizada. El primer procedimiento busca la condición de frontera para una variable específica, por ejemplo `''Temp''`.

```
bool NavierStokes::LeerFronteras
   (Cadena TomarFrontera, Vector_basico<Cadena> &fr)
{
int i, j, k, nbi;
Cadena med;
nbi = malla->obtNoIndFront();
fr.chaSize (nbi);
for (i=1; i<=nbi; i++) {
fr(i) = malla->obtNombreIndFront(i);
if (fr(i).contiene(TomarFrontera)) {
j = fr(i).Buscar(0,TomarFrontera);
k = fr(i).tam();
med = fr(i).subCadena(j, k-j );
if (med.contiene(Separador)) {
j = med.Buscar(0,Separador);
med = med.subCadena(0,j);
}
fr(i) = med;
}
}
```

```
return true;
}
```

La siguiente sobrecarga a la función `LeerFronteras()` retorna el valor de las condiciones de frontera para la velocidad.

```
bool NavierStokes::LeerFronteras
 (Vector_basico<Cadena> &fr1, Vector_basico<Cadena> &fr2,
 Vector_basico<Cadena> &fr3)
{
 int i,j,k, nbi;
 nbi = malla->obtNoIndFront();
 Cadena med, ap;
 fr1.chaSize(nbi);fr2.chaSize(nbi);fr3.chaSize(nbi);
 for (i=1; i<=nbi; i++)
  fr1(i)=fr2(i)=fr3(i) = malla->obtNombreIndFront(i);
 for (i=1; i<=nbi; i++){
  if (fr1(i).contiene("VelU"))
  { j = fr1(i).Buscar(0,"VelU");
   k = fr1(i).tam();
   med = fr1(i).subCadena(j,k-j);
   if (med.contiene(Separador))
   {j = med.Buscar(0,Separador);
    med = med.subCadena(0,j);
   }
   fr1(i) = med;
  }
  if (fr2(i).contiene("VelV"))
  { j = fr2(i).Buscar(0,"VelV");
   k = fr2(i).tam();
   med = fr2(i).subCadena(j,k-j);
   if (med.contiene(Separador))
   {j = med.Buscar(0,Separador);
    med = med.subCadena(0,j);
   }
   fr2(i) = med;
  }
  if (fr3(i).contiene("VelW"))
  { j = fr3(i).Buscar(0,"VelW");
   k = fr3(i).tam();
   med = fr3(i).subCadena(j,k-j);
   if (med.contiene(Separador))
   { j = med.Buscar(0,Separador);
```

```
    med = med.subCadena(0,j);
   }
   fr3(i) = med;
  }
 }
return true;
}
```

19. Método `NablaEsc()`: esta función realiza el cálculo del gradiente de cualquier variable escalar. Para disminuir el tiempo necesario para pasar el vector `Entrada` se pasa por referencia. Adicionalmente, `Salida` es la variable en la cual se almacena el resultado obtenido por el método, razón por la que también es necesario pasar por referencia. La función toma el argumento `Cadena TomarFrontera` para determinar la forma correcta de hallar el valor de `Entrada` en la frontera, por defecto, `TomarFrontera` tiene el valor de `"Ninguna"` caso en el que se asume que el valor de `Entrada` en la frontera puede ser representado por el valor en el centro del volumen de control.

I. El primer paso es declarar las variables a usar en la ejecución de la función. Se declara `fr` como una instancia de `Vector_basico<Cadena>` para almacenar la información de la condición de frontera correcta para cada lado.

```
bool NavierStokes::NablaEsc  ( Vector<real> &Entrada,
      Vector_basico<Vector_xyz<real>> &Salida, Cadena TomarFrontera )
{ int i, j, k, veci, conx; int nbi= malla->obtNoIndFront();
real prom, area, d1, d2;
Vector_xyz<real> dir(ndim); Vector_basico<Cadena> fr;
if (TomarFrontera!="Ninguna") LeerFronteras(TomarFrontera,fr);
...
```

II. Se realiza un bucle repetitivo que recorre cada volumen de control, seguidamente se llenan todas las componentes del `Vector_xyz<real> Salida(i)` con el valor 0. Se determina el número de lados del volumen de control actual y se realiza un bucle repetitivo para cada lado del VC actual.

```
...
for (i= 1 ; i<= nvol ; i++) {
 Salida(i).llenar(0.0);
 conx = elems(i)->GetNumOfConex();
  for ( j=1; j<=conx ; j++)  {
...
```

III. Para cada lado se encuentra el VC que comparte dicho lado, si el lado `j` es una frontera (`veci=0`) se procede a calcular el valor de `prom` de acuerdo al tipo de condición de frontera y el valor de esta. Si `TomarFrontera` tiene el valor especificado por defecto, se realiza la asignación `prom=Entrada(i)`.

```
...
   veci = malla->getConx (i, j); area = elems(i)->getArea(j);
   if ( veci==0) {
    if (TomarFrontera!="Ninguna") {
     for (k=1; k<=nbi; k++)
      if (  malla->LadoFront(i,j,k,elems(i)())) break;
     prom = LeerFrontEsp (i,j, Entrada (i),fr(k));
    }
    else {prom = Entrada(i);}
   }
...
```

IV. Para el caso que el lado `j` sea compartido con otro VC, se asigna a `prom` el valor promedio de la propiedad entre los dos volúmenes de control. Finalmente, se multiplica el valor encontrado por el `area` y el vector unitario del lado, se suma el efecto de todas las caras y se divide el valor encontrado por el volumen correspondiente.

```
...
        else {for ( k=1 ; k<=conx ; k++ )
        if(malla->getConx(veci,k)==i) break;
      d1 = elems(i)->getDelta(j);
      d2 = elems(veci)->getDelta(k);
        prom = (Entrada(veci) * d1 + Entrada (i) * d2)/(d1+d2);
      }
    Salida(i) = Salida(i) + (elems(i)->GetDirSide(j)*(area*prom));
   }
  Salida(i) = Salida(i) / Volumen(i);
  if(nvol == 9)
   {Salida(i).Escribir(std_o);std_o<<"\n";}
 }
return true;
}
```

20. Método `NablaDotVec(...)`: esta función calcula la operación $\nabla \cdot \vec{F}$ siendo $\vec{F}$ cualquier propiedad vectorial, por ejemplo $\vec{V}$.

I. Se declaran las variables que van a ser utilizadas para la ejecución de la función. Se destaca la creación de `fr1`, `fr2` y `fr3` del tipo `Vector_basico<Cadena>` en las cuales se almacenarán los tipos de frontera para cada una de las componentes de $\vec{V}$.

```
bool NavierStokes::NablaDotVec ( Vector_basico< Vector_xyz <real>>
       &Entrada, Vector<real> &Salida, Cadena Frontera)
{
int i, j, k, veci, conx; int nbi= malla->obtNoIndFront();
real area, rx,ry,rz,d1,d2;  Vector_xyz<real> prom (ndim);
```

```
Vector_basico<Cadena> fr1,fr2,fr3;
if (Frontera=="Vel") LeerFronteras(fr1, fr2, fr3);
...
```

II. Para recorrer cada uno de los lados de cada volumen de control se utilizan dos bucles anidados. Las condiciones de frontera se hallan con `LeerFrontEsp(...)` si y solo si `Frontera` contiene la cadena ``Vel''.

```
...
Salida.llenar (0.0);
for (i= 1 ; i<= nvol ; i++) {
 conx = elems(i)->GetNumOfConex(); rx = ry =rz = 0;
 for ( j=1; j<=conx ; j++) {
  veci = malla->getConx (i, j); area = elems(i)->getArea(j);
  if ( veci==0) {
   if (Frontera.contiene("Vel")) {
    for (k=1; k<=nbi; k++)
     if (  malla->LadoFront(i,j,k,elems(i)())) break;
    prom(1) = LeerFrontEsp(i,j,Entrada (i)(1),fr1(k));
    prom(2) = LeerFrontEsp(i,j,Entrada (i)(2),fr2(k));
    if (ndim == 3)
    prom(3) = LeerFrontEsp(i,j,Entrada (i)(3),fr3(k));
   }
   else {prom = Entrada(i);}
  }
   else {
    for ( k=1 ; k<=conx ; k++ )
     if(malla->getConx(veci,k)==i)   break;
    d1 = elems(i)->getDelta(j);
    d2 = elems(veci)->getDelta(k);
    prom = (Entrada(veci) * d1 + Entrada (i) * d2)/(d1+d2);
   }
...
```

III. Se almacena parcialmente el valor del producto punto en `rx`, `ry`, `rz` y, finalmente se suman y se divide el valor obtenido por `Volumen(i)`.

```
...
   rx += (prom(1)*elems(i)->GetDirSide(j)(1))*area;
   ry += (prom(2)*elems(i)->GetDirSide(j)(2))*area;
   if (ndim==3)
    rz += (prom(3)*elems(i)->GetDirSide(j)(3))*area;
   }
   Salida(i) = rx + ry ;
   if (ndim==3) Salida(i) += rz;
```

```
    Salida(i) = Salida(i) / Volumen(i);
  }
 return true;
 }
```

21. Método `NablaDotVecEspecial(...)`: esta función es una versión especial para el cálculo de `NablaDotVec(...)` pero que tiene en cuenta la variación en la velocidad del fluido debido al efecto de la presión como se ve en (Ec 6.15). Esta función es usada en el segundo paso del método SIMPLE para calcular el valor del residuo másico en cada volumen de control. El último parámetro, `MultRho`, es utilizado como control para el caso en que se deba multiplicar el valor de `prom` por el valor de la densidad.

```
bool NavierStokes::NablaDotVecEspecial ( Vector_basico< Vector_xyz
  <real>> &Entrada, Vector<real> &Salida, bool MultRho )
{
 int i, j, k, veci, conx;  int nbi= malla->obtNoIndFront();
 real area, coefar, deltap, gradp, sum, act, densidadprom, d1, d2;
 Vector_xyz<real> prom (ndim), coefvel(ndim), GPprom(ndim), efec(ndim);
 Vector_basico<Cadena> fr1,fr2,fr3;  LeerFronteras(fr1, fr2, fr3);
 Salida.llenar (0.0);
 for (i= 1 ; i<= nvol ; i++) {
  conx = elems(i)->GetNumOfConex(); sum = 0.0;
  for ( j=1; j<=conx ; j++) {
   veci = malla->getConx (i, j);
   area = elems(i)->getArea(j);
   if ( veci==0) {
    for (k=1; k<=nbi; k++)
     if (  malla->LadoFront(i,j,k,elems(i)())) break;
    prom(1) = LeerFrontEsp(i,j,Entrada (i)(1),fr1(k));
    prom(2) = LeerFrontEsp(i,j,Entrada (i)(2),fr2(k));
    if (ndim == 3)
     prom(3) = LeerFrontEsp(i,j,Entrada (i)(3),fr3(k));
    densidadprom = rho(i);
    coefar = elems(i)->getArea(j);//Volumen(i)/ elems(i)->getDelta(j);
    coefvel(1) = (1.0/CoefU(i)) *
     elems(i)->GetDirSide(j)(1) ;
    coefvel(2) = (1.0/CoefV(i))*
     elems(i)->GetDirSide(j)(2) ;
    if (ndim == 3) coefvel(3) = (1.0/CoefW(i))*
     elems(i)->GetDirSide(j)(3) ;
    deltap = 0.0;
    GPprom.llenar(0.0);
    gradp = GPprom * elems(i)->GetDirSide(j);
```

```
    gradp *= elems(i)->getDelta  (j);
   }
   else {
    for ( k=1 ; k<=conx ; k++ )
    { if(malla->getConx(veci,k)==i)   break;}
...
```

A continuación se usa `densidadprom` como el valor de la densidad en el lado del volumen de control. El término Ⓐ de (Ec 6.15) está representado por `prom`, el término Ⓑ por `coefar`, el término Ⓒ por `coefvel`, el término Ⓓ por `deltap` y finalmente el término Ⓔ por `gradp`.

```
...
    d1 = elems(i)->getDelta(j);
    d2 = elems(veci)->getDelta(k);
    densidadprom = (d2*rho(i)+d1*rho(veci))/(d1+d2);
    prom = (Entrada(i)*d2+Entrada(veci)*d1)/(d1+d2);
    coefar =elems(i)->getArea(j);// Volumen(i)/
    coefvel(1) = ((d2/CoefU(i) + d1/CoefU(veci))/(d1+d2))*
     elems(i)->GetDirSide(j)(1);
    coefvel(2) = ((d2/CoefV(i) + d1/CoefV(veci))/(d1+d2))*
     elems(i)->GetDirSide(j)(2);
    if (ndim == 3)
     coefvel(3) = ((d2/CoefW(i) + d1/CoefW(veci))/(d1+d2))*
     elems(i)->GetDirSide(j)(3);
    deltap =  Presion(veci) - Presion(i) ;
    GPprom = (GP(i)*d2+ GP(veci)*d1) / (d1+d2);
    gradp = GPprom * elems(i)->GetDirSide(j);
    gradp = gradp * (elems(i)->getDelta(j) + elems(veci)->getDelta(k));
   }
   efec = coefvel * coefar * (deltap-gradp);
   prom = prom - efec;
   if (MultRho)
    prom = prom * densidadprom;
   act  = (prom*elems(i)->GetDirSide(j))*area;
   sum += act;
  }
  Salida(i) = sum;
  Salida(i) = Salida(i) / Volumen(i);
 }
return true;
}
```

22. Método `TensorPorVector_xyz()`: esta función será utilizada para multiplicar el tensor de esfuerzos por el vector normal de una superficie del volumen de control.

```
bool NavierStokes::TensorPorVector_xyz ( Matriz<real> Me,
      Vector_xyz<real> Ve, Vector_xyz<real> &Salida)
{
int i, j; Salida.llenar(0.0);
for (i = 1 ; i<= ndim; i++) for (j = 1 ; j<= ndim; j++)
Salida (i) += Me(j,i) * Ve(j);
return true;
}
```

23. Método `PromMatriz(...)`: esta función halla el valor promedio del tensor esfuerzo en una superficie común para dos volúmenes de control.

```
Matriz<real> NavierStokes::PromMatriz(Matriz<real> E1,
      real Peso1, Matriz<real> E2, real Peso2)
{
int i, j , k , l ; Matriz<real> Salida(E1);
k = E1.obtNoFilas(); l = E1.obtNoCols ();
for (i = 1 ; i<= k; i++) for (j = 1 ; j<= l; j++)
Salida(i,j) = ((E1(i,j)*Peso1)+(E2(i,j)*Peso2))
/ ( Peso1 + Peso2 );
return Salida;
}
```

24. Método `NablaPorMatriz(...)`: para el cálculo de $\nabla\tau$ se debe llamar a esta función la cual no tiene en cuenta condiciones de frontera específicas dado que esos valores no son introducidos en el archivo `''*.geom''`.

```
bool NavierStokes::NablaPorMatriz ( Vector_basico< Matriz <real>>
       &Entrada, Vector_basico<Vector_xyz<real>> &Salida)
{
 int i, j, k, veci, conx;
 real area, d1, d2; Matriz<real> prom(ndim);
 Vector_xyz<real> prod (ndim), dir(ndim);
 for (i= 1 ; i<= nvol ; i++) {
  Salida(i).llenar(0.0);  conx = elems(i)->GetNumOfConex();
  for ( j=1; j<=conx ; j++) {
   veci = malla->getConx (i, j); area = elems(i)->getArea(j);
   if ( veci==0)
   { TensorPorVector_xyz(Entrada(i),elems(i)->GetDirSide(j), prod); }
   else {
```

```
    for ( k=1 ; k<=conx ; k++ )
    { if(malla->getConx(veci,k)==i)   break;}
    d1 = elems(i)->getDelta(j);
    d2 = elems(veci)->getDelta(k);
    prom = PromMatriz(Entrada(i),d2,Entrada(veci),d1);
    TensorPorVector_xyz( prom, elems(i)->GetDirSide(j), prod);
   }
   Salida(i) = Salida(i) + (prod * area);
  }
  Salida(i) = Salida(i) / Volumen(i);
 }
 return true;
}
```

25. Método `GuardarDatos()` y sobrecarga: la versión de `GuardarDatos` con argumentos verifica si las velocidades del fluido exceden los límites esperados, si esto ocurre, se muestra un error en pantalla y no se guardan los datos. Por otro lado, si las velocidades están dentro de los límites esperados se realiza la llamada a la función `GuardarDatos()` sin argumentos.

```
bool NavierStokes:: GuardarDatos(real minvel,real maxvel)
{
 real minu, minv, minw, minimo;
 real maxu, maxv, maxw, maximo;
 minu = VelU.minValor(); minv = VelV.minValor();
 if (ndim == 3)  minw = VelW.minValor();
 else   minw = 1e6;
 minimo = minu;  if (minv<minimo) minimo = minv;
 if ( minw<minimo) minimo =minw;
 maxu = VelU.maxValor(); maxv = VelV.maxValor();
 if (ndim == 3)  maxw = VelW.maxValor();
 else maxw =-1e6;
 maximo = maxu; if (maxv>maximo) maximo = maxv;
 if (maxw>maximo) maximo = maxw;
 if (minimo>minvel && maximo<maxvel)
  {return GuardarDatos();}
 else
  {std_o<<"Error:: no se exportan los datos a *.m porque se excedio"
  <<" el limite de la velocidad establecida\n"; }
 return false;
}
```

La versión sin argumentos guarda en archivos el valor de algunos de los vectores declarados de manera que posteriormente se puede continuar el cálculo. Los datos se almacenan en

varios archivos con el nombre del caso, el nombre de la variable y el número de volúmenes de control. De esta manera, se evitan posibles errores por datos incongruentes. Un ejemplo del nombre de un archivo generado es ``CLimiteTemp900.m''.

```
bool NavierStokes:: GuardarDatos()
{
 if (!solucionado)
    {std_o<<"\nError: no se ha solucionado el problema\n";}
 char nv [10]; Cadena nArchivo; _itoa(nvol,nv,10); strcat (nv,".m");
 nArchivo = NombreCaso + "rho"; nArchivo += nv;
 rho.guardar(nArchivo.carts(),"rho");
 nArchivo = NombreCaso + "k"; nArchivo += nv;
 k.guardar(nArchivo.carts(),"k");
 nArchivo = NombreCaso + "cp"; nArchivo += nv;
 cp.guardar(nArchivo.carts(),"cp");
 nArchivo = NombreCaso + "mu"; nArchivo += nv;
 mu.guardar(nArchivo.carts(),"mu");
 nArchivo = NombreCaso + "VelU"; nArchivo += nv;
 VelU.guardar(nArchivo.carts(),"VelU");
 nArchivo = NombreCaso + "VelV"; nArchivo += nv;
 VelV.guardar(nArchivo.carts(),"VelV");
 nArchivo = NombreCaso + "VelW"; nArchivo += nv;
 if (ndim==3)
   VelW.guardar(nArchivo.carts(),"VelW");
 nArchivo = NombreCaso + "VelU_ant"; nArchivo += nv;
 VelU_ant.guardar(nArchivo.carts(),"VelU_ant");
 nArchivo = NombreCaso + "VelV_ant"; nArchivo += nv;
 VelV_ant.guardar(nArchivo.carts(),"VelV_ant");
 nArchivo = NombreCaso + "VelW_ant"; nArchivo += nv;
 if (ndim==3)
   VelW_ant.guardar(nArchivo.carts(),"VelW_ant");
 nArchivo = NombreCaso + "Presion_n"; nArchivo += nv;
 Presion_n.guardar(nArchivo.carts(),"Presion_n");
 nArchivo = NombreCaso + "Presion_ant"; nArchivo += nv;
 Presion_ant.guardar(nArchivo.carts(),"Presion_ant");
 nArchivo = NombreCaso + "Temp"; nArchivo += nv;
 Temp.guardar(nArchivo.carts(),"Temp");
 nArchivo = NombreCaso + "Temp_ant"; nArchivo += nv;
 Temp_ant.guardar(nArchivo.carts(),"Temp_ant");
 return true;
}
```

26. Método `CargadoDeDatos()`: este método no toma ningún parámetro y se encarga de leer los datos de los vectores para poder continuar con la solución del problema. Una vez

inicializados los vectores es necesario llamar a `CalcularTauyGVs()` y `CalcularPyG()` para inicializar los demás vectores que no fueron directamente guardados.

```
bool NavierStokes:: CargadoDeDatos()
{
char nv [10]; Cadena nArchivo;
_itoa(nvol,nv,10); strcat (nv,".m");
nArchivo = NombreCaso + "rho"; nArchivo += nv;
rho.cargar(nArchivo.carts(),"rho");
nArchivo = NombreCaso + "k"; nArchivo += nv;
k.cargar(nArchivo.carts(),"k");
nArchivo = NombreCaso + "cp"; nArchivo += nv;
cp.cargar(nArchivo.carts(),"cp");
nArchivo = NombreCaso + "mu"; nArchivo += nv;
mu.cargar(nArchivo.carts(),"mu");
nArchivo = NombreCaso + "VelU"; nArchivo += nv;
VelU.cargar(nArchivo.carts(),"VelU");
nArchivo = NombreCaso + "VelV"; nArchivo += nv;
VelV.cargar(nArchivo.carts(),"VelV");
nArchivo = NombreCaso + "VelW"; nArchivo += nv;
if (ndim==3)
  VelW.cargar(nArchivo.carts(),"VelW");
nArchivo = NombreCaso + "VelU_ant"; nArchivo += nv;
VelU_ant.cargar(nArchivo.carts(),"VelU_ant");
nArchivo = NombreCaso + "VelV_ant"; nArchivo += nv;
VelV_ant.cargar(nArchivo.carts(),"VelV_ant");
nArchivo = NombreCaso + "VelW_ant"; nArchivo += nv;
if (ndim==3)
  VelW_ant.cargar(nArchivo.carts(),"VelW_ant");
nArchivo = NombreCaso + "Presion_n"; nArchivo += nv;
Presion_n.cargar(nArchivo.carts(),"Presion_n");
nArchivo = NombreCaso + "Presion_ant"; nArchivo += nv;
Presion_ant.cargar(nArchivo.carts(),"Presion_ant");
nArchivo = NombreCaso + "Temp"; nArchivo += nv;
Temp.cargar(nArchivo.carts(),"Temp");
nArchivo = NombreCaso + "Temp_ant"; nArchivo += nv;
Temp_ant.cargar(nArchivo.carts(),"Temp_ant");

CalcularTauyGVs();
CalcularPyG();
CalcQcalor();
return true;
}
```

27. Método `SetCargarDatos(...)`: modifica el valor del booleano `CargarDatos`.

```
 bool NavierStokes::SetCargarDatos(bool opcion)
{ CargarDatos = opcion; return true; }
```

28. Método `SetNoIterExt(...)`: modifica el valor del número de iteraciones máximo `Nitera`.

```
 void NavierStokes::SetNoIterExt(int set){ Nitera = set;}
```

29. Método `SetTipoMetodoSolucion(...)`: compara el valor de `tipo` con las opciones establecidas en `MSNS`[4], si coincide el valor, se actualiza `MetodoSolveNS`, de lo contrario, se toma el valor por defecto definido en el constructor de la clase.

```
bool NavierStokes::SetTipoMetodoSolucion(Cadena tipo)
{
bool coincide = false; int i=0; const char *cad= tipo.carts();
while ( MSNS[i]) {
if ( strcmp(cad,(MSNS[i])) ==0 ) {coincide=true; break;}
i++; }
if (coincide)  MetodoSolveNS = tipo;
else {
std_o<<"\nMetodo no encontrado: se toma valor por defecto\n";
system("pause");
}
return coincide;
}
```

30. Método `SetEsquemaConveccion()`: verifica que el argumento introducido coincida con alguno de los métodos disponibles para resolver las ecuaciones de convección.

```
bool NavierStokes::SetEsquemaConveccion(Cadena tipo)
{
bool coincide = false; int i=0;
while ( MACD[i]){
if (tipo.contiene(MACD[i])) {coincide=true; break;}
i++;}
if (coincide) MetodoAprox=tipo;
else {
std_o<<"\nEsquema no encontrado: se toma valor por defecto\n";
system("pause");
```

[4] El puntero de cadena de caracteres se declaró en el archivo de encabezado de la clase.

```
//se toma el valor por defecto del constructor
}
return coincide;
}
```

31. Métodos para dar un valor de referencia a la presión: en el método SIMPLE el valor de la presión es hallado realizando una serie de correciones al resolver la ecuación de continuidad, sin embargo, por la definición de las condiciones de frontera de la correción de presión (tipo Neumann) el valor encontrado puede tener un valor máximo o un valor mínimo diferente al deseado. Es por esto que se definen las siguientes cuatro funciones.

```
 void NavierStokes::SetMaximaPresion(real max)
  {maxPres= true; minPres= false; maxP = max;}
void NavierStokes::SetMinimaPresion(real min)
   {maxPres= false; minPres= true; minP = min;}
void NavierStokes::CalcPresRef()
{
 int i; real r;
 if (maxPres || minPres) {
  if(maxPres){r=Presion.maxValor(); r= maxP-r;}
  if(minPres){r=Presion.minValor(); r= minP-r;}
  for (i=1; i<=nvol; i++)
   Presion(i) = Presion(i) + r;
 }
}
void NavierStokes::ResetMinMaxPresion() { minP = maxP = false;}
```

32. Método `SetParametros(...)`: en este método se asignan los valores a los parámetros más importantes que definen el modo de realizar el cálculo.

```
bool NavierStokes::SetParametros (bool Datos_iniciales_externos ,
 int N_iteraciones_ext, Cadena Esquema_conveccion,
 Cadena Esquema_solucion, bool Resultados_indep_tiempo )
{
 bool correcto = true;
 SetCargarDatos(Datos_iniciales_externos);
 SetNoIterExt(N_iteraciones_ext);
 correcto = correcto && SetEsquemaConveccion(Esquema_conveccion);
 correcto = correcto && SetTipoMetodoSolucion (Esquema_solucion);
 ResEstable = Resultados_indep_tiempo;
 return correcto;
}
```

33. Método `SetMAssThreshold(...)`: asigna el valor del criterio de convergencia `MAssThreshold`.

```
bool NavierStokes::SetMAssThreshold(real MaxResMas)
{MAssThreshold = MaxResMas; return true;}
```

34. Método `SubRelajaSIMPLE(...)`: se define el parámetro de subrelajación α_P usado en el método SIMPLE para actualizar el valor de la presión y de la velocidad.

```
void NavierStokes::SubRelajaSIMPLE(real param) {Subrelajacion = param;}
```

Crear un objeto de la clase: se da el siguiente fragmento de código que muestra como se crea un objeto de la clase:

```
...
NavierStokes NS;
NS.SubRelajaSIMPLE(0.1);
NS.SetParametros( false, 300, "hibrido", "SIMPLE",true);
NS.SetMAssThreshold(1e-8); NS.SetMinimaPresion(0.0);
NS.Leer("ARCH=NSCL1.txt", "ARCH=Solve_datos.txt",
  "dt=[0.01 0.1 1.0] t en [0.0 1 10.0 10000.0]",1.0,
  "ARCH=Solve_datostemp.txt");
NS.Calcular();
NS.GuardarDatos();
...
```

El archivo `"ARCH=NSCL1.txt"` contiene la siguiente información:

```
> mallador = PREPROCESSOR=ConjMalladorSupElem/
ARCH=NSCL1.geom/ARCH=NSCL1.parts
```

El archivo ``` ``NSCL1.geom'' ``` está compuesto por los siguientes caracteres.

```
> nsd = 2;
> subdominios = 1;
> no_de_superels = 1;
>nbind = 4
>bname VelUNeumann=0,VelVNeumann=0,PNeumann=0,TempNeumann=0
VelUDirichlet=1,VelVNeumann=0,PNeumann=0,TempDirichlet=1
VelUDirichlet=1,VelVDirichlet=0,PNeumann=0,TempDirichlet=1
VelUDirichlet=0,VelVDirichlet=0,PNeumann=0,TempDirichlet=0
>SupEl;
>subdominio_no = 1;
>tipoelemento = ElmB4n2D;
>fronteras = [1(1)] [2 (2)] [3 (3)] [4(4)];
>nodos = [1(0 0)]+[2(8.0 0)]+[3(0 0.6)]+[4(8.0 0.6)];
>lados=;
```

El contenido de ``NSCL1.parts`` es el siguiente:

```
>nsd = 2;
>no_de_superels = 1;

>SE;>d=2;>e=ElmB4n2D;>div=[60,90];>grad=[-0.8,-0.5];
```

La información del archivo ``Solve_datos.txt'' ya se ha presentado[5] por lo que se omite en este caso.

6.4 CASOS DESARROLLADOS DEL FLUJO EN LA CAPA LÍMITE POR MEDIO DE LAS ECUACIONES COMPLETAS DE NAVIER-STOKES

En esta sección se presenta la solución de las ecuaciones de Navier-Stokes aplicadas al flujo en la capa límite, no se realiza ninguna simplificación al tensor de esfuerzos y se asume que la presión puede variar tanto en el eje x como en el eje y. Las ecuaciones simplificadas de la capa límite se resuelven por otro método que se presentará posteriormente ya que, partiendo del método SIMPLE que halla una correción de presión para satisfacer la ecuación de continuidad, sería ilógico determinar la correción de presión que de antemano se conoce es cero. En esta sección se comparan los resultados obtenidos con respecto de la solución presentada por Blasius.

El problema del flujo en la capa límite con las condiciones de frontera se muestra en la figura 31. La frontera sur es una placa plana isotérmica fija, la frontera oeste es la zona de entrada del flujo con velocidad y temperatura conocida, la frontera norte se encuentra en la zona de flujo no viscoso y se considera que se conoce la velocidad en dirección x, la variación de la velocidad v y la temperatura; por último, la frontera este es la zona de salida del flujo y se asume que la variación de la velocidad y temperatura es nula en la dirección normal a la superficie.

La velocidad en la frontera norte se define de manera general como $u = U_e(x)$ y no se restringe a un único valor ya que existen varias soluciones analíticas que se basan en un perfil de velocidad expecífico para resolver las ecuaciones del flujo en la capa límite.

Además de las condiciones de frontera de la velocidad y la temperatura se definen las condiciones de frontera para la ecuación de corrección de presión, figura 32. Para la ecuación de correción de presión se considera que su gradiente es cero en las fronteras[6] lo cual suele aplicarse para diversos problemas que son solucionados por el método SIMPLE.

[5] Ver la página 66.

[6] Aunque se conociera la presión en las fronteras su valor no puede ser directamente aplicado como condición de frontera dado que se soluciona la ecuación de corrección de presión mas no una ecuación de presión.

Figura 31: Problema del flujo en la capa límite.

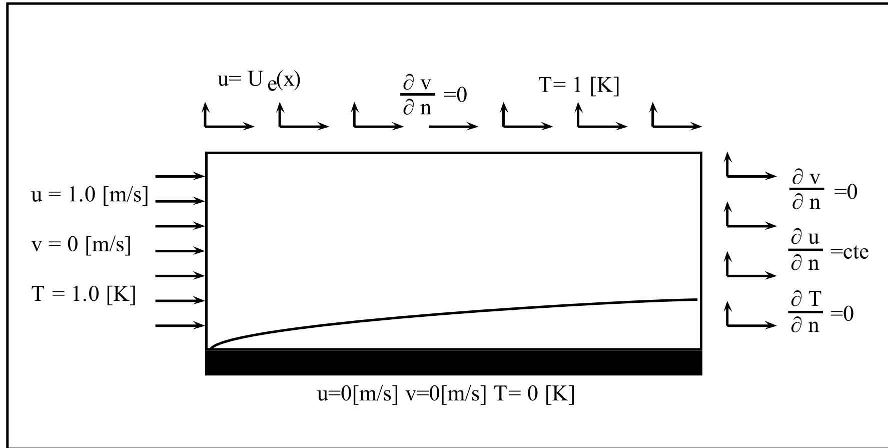
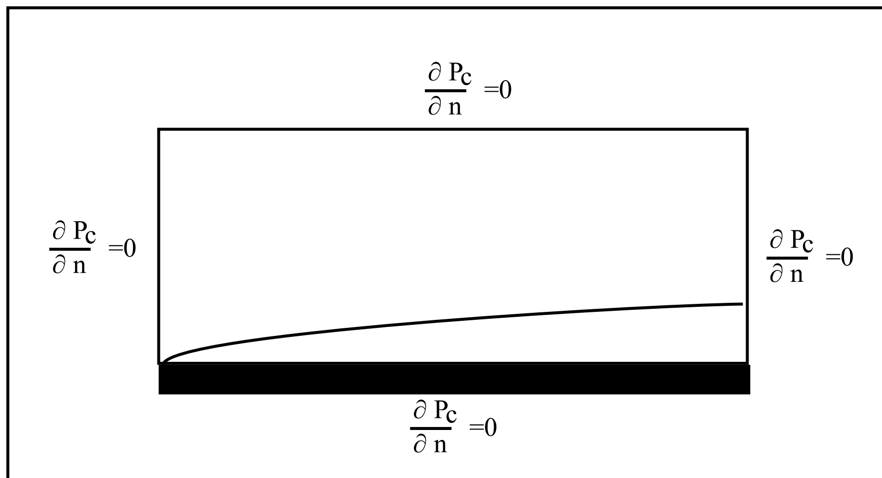


Figura 32: Condiciones de frontera para la ecuación de corrección de presión.



Se escoge el aire como el fluido a utilizar en los análisis. Se utilizan las siguientes propiedades:

$$P_{ref} = 101325[Pa] \quad \text{(Ec 6.19a)}$$

$$T_{ref} = 27[C] = 300[k] \quad \text{(Ec 6.19b)}$$

$$R_{gas} = 287.0 \left[\frac{Pa\, m^3}{KgK}\right] \quad \text{(Ec 6.19c)}$$

$$\rho = 1.172 \left[\frac{Kg}{m^3}\right] \quad \text{(Ec 6.19d)}$$

$$k = 0.02566 \left[\frac{W}{m\, K}\right] \quad \text{(Ec 6.19e)}$$

$$C_p = 1007 \left[\frac{J}{Kg\,K}\right] \quad \text{(Ec 6.19f)}$$

$$\mu = 1.858\,e-5 \left[\frac{Kg}{m\,s}\right] \quad \text{(Ec 6.19g)}$$

Se plantea una prueba para determinar la funcionalidad de la clase `NavierStokes` por el método SIMPLE junto con las clases `ConveccionTransitoria` y `DifusionTransitoria`.

6.4.1 Flujo en la capa límite para $0 \leq Re \leq 500000$ con las ecuaciones N.S. completas: NSCL

Se utilizan los siguientes parámetros para la simulación:

$$\begin{array}{lll} x = 8[m] & U = 1[m/s] & Re_{x=8[m]} = 504629 \\ \delta_{ter,t} = 0,06256[m] & \delta_{10x} = 0.6[m] & n_{vol} = variable \\ \text{Método:SIMPLE} & m_{itera} = 300 & \alpha_P = 0.1 \end{array} \quad \text{(Ec 6.20)}$$

$$\Delta t \; = 0.1\;[s]$$

El archivo de definición de la geometría contiene:

```
> nsd = 2;
> subdominios = 1;
> no_de_superels = 1;
>nbind = 4
>bname VelUNeumann=0,VelVNeumann=0,PNeumann=0,TempNeumann=0
VelUDirichlet=1,VelVNeumann=0,PNeumann=0,TempDirichlet=1
VelUDirichlet=1,VelVDirichlet=0,PNeumann=0,TempDirichlet=1
VelUDirichlet=0,VelVDirichlet=0,PNeumann=0,TempDirichlet=0
>SupEl;
>subdominio_no = 1;
>tipoelemento = ElmB4n2D;
>fronteras = [1(1)] [2 (2)] [3 (3)] [4(4)];
>nodos = [1(0 0)]+[2(8.0 0)]+[3(0 0.6)]+[4(8.0 0.6)];
>lados=;
```

Para mostrar que sucede al aumentar el número de volúmenes de control se realizan tres particiones diferentes:

1. 5400 volúmenes de control.

```
>nsd = 2;
>no_de_superels = 1;

>SE;>d=2;>e=ElmB4n2D;>div=[60,90];>grad=[-0.8,-0.5];
```

2. 27000 volúmenes de control.

```
>nsd = 2;
>no_de_superels = 1;

>SE;>d=2;>e=ElmB4n2D;>div=[300,90];>grad=[-0.8,-0.5];
```

3. 120000 volúmenes de control.

```
>nsd = 2;
>no_de_superels = 1;

>SE;>d=2;>e=ElmB4n2D;>div=[800,150];>grad=[-0.8,-0.6];
```

El método de solución para las ecuaciones de la cantidad de movimiento y conservación de la energía es el siguiente:

```
Parametros de la matriz (clase prmMatriz);
>guarda  = Matriz;
>guarda_sim = FALSE
>pivoteo = FALSE
>umbral =0.0

prm_solucionEcLin parametros:
>solver_classname=GBiCEst
>metodo=R-OM(7)
>cambiar=0.0
>pivoteo=NO_PIVOT
>relajacion=0.1
>startvector=USER_START
>criterio_def=TRUE
>nextern_crit=1
>maxit=1000
>estimate_eigvals=FALSE

Parametros del precondiciondor:
>precondicionador=PrecRILU
>left_prec=TRUE
>auto_iniciar=TRUE
>nivel_llenado=0
>prec_RILU=1
>prec_SSOR=1
>pasos_int=1
```

```
monitor de convergencia:
>monitor_tp = MCResidualRel
>residual_type = RES_ORIGINAL
>maxerr = 1.0e-9
>norm_type = l2
>monitor = ON
>graficar_ejec = OFF
>criterio = ON
>insertar = ON
>relop = CM_AND
>usar_info_vp = OFF
>rel_a_rhs = OFF
>chunk_size = 100
>base_usua = 1.0
```

Se usa el precondicionador RILU (descomposición incompleta LU) para disminuir el número de iteraciones interiores necesarias para resolver las ecuaciones convectivas.

Para resolver la ecuación de corrección de presión se usó el gradiente conjugado ya que la matriz de coeficientes de esta ecuación es simétrica.

```
Parametros de la matriz (clase prmMatriz);
>guarda  = Matriz;
>guarda_sim = FALSE
>pivoteo = FALSE
>umbral =0.0

prm_solucionEcLin parametros:
>solver_classname=GradConj
>metodo=R-OM(7)
>cambiar=0.0
>pivoteo=NO_PIVOT
>relajacion=0.1
>startvector=USER_START
>criterio_def=TRUE
>nextern_crit=1
>maxit=3000
>estimate_eigvals=FALSE

Parametros del precondiciondor:
>precondicionador=PrecRILU
>left_prec=TRUE
>auto_iniciar=TRUE
```

```
>nivel_llenado=0
>prec_RILU=1
>prec_SSOR=1
>pasos_int=1

monitor de convergencia:
>monitor_tp = MCResidualRel
>residual_type = RES_ORIGINAL
>maxerr = 1.0e-8
>norm_type = l2
>monitor = ON
>graficar_ejec = OFF
>criterio = ON
>insertar = ON
>relop = CM_AND
>usar_info_vp = OFF
>rel_a_rhs = OFF
>chunk_size = 100
>base_usua = 1.0
```

Con el precondicionador RILU la ecuación de corrección de presión puede ser resuelta en 40 iteraciones interiores comparada con más de 600 iteraciones sin el precondicionador. Como dato adicional, la solución de las tres ecuaciones lineales para cada iteración exterior toma alrededor de 2 segundos para 120000 volúmenes de control; la construcción de las matrices y el cálculo de los gradientes, entre otros, toma alrededor de 2 minutos[7].

Los resultados del espesor de la capa límite hidráulica para las simulaciones propuestas y el espesor teórico de la capa límite hidráulica se muestra en la figura 33. El valor teórico se tomó de la ecuación Ec 1.10. Para la capa límite térmica se presentan los resultados en la figura 34. La escala en dirección y en las siguientes figuras se ha exagerado para facilitar la visualización de los resultados.

[7]Datos aproximados para un computador Toshiba Satellite L505-GS5037, procesador core i3 de 2.2 GHz, 3.8GB de memoria RAM 1066MHz, Windows 7 64 bits.

Figura 33: Capa límite hidráulica δ_h del caso NSCL.

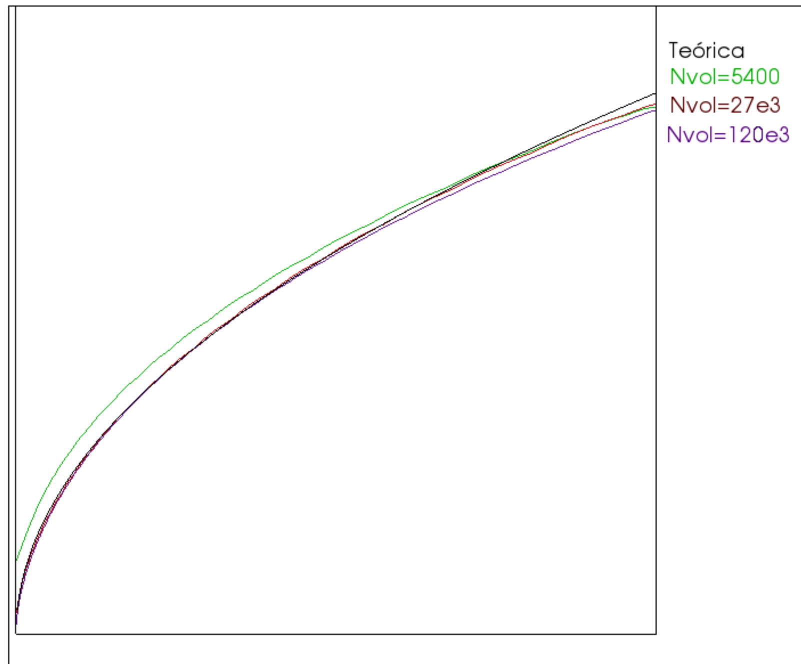
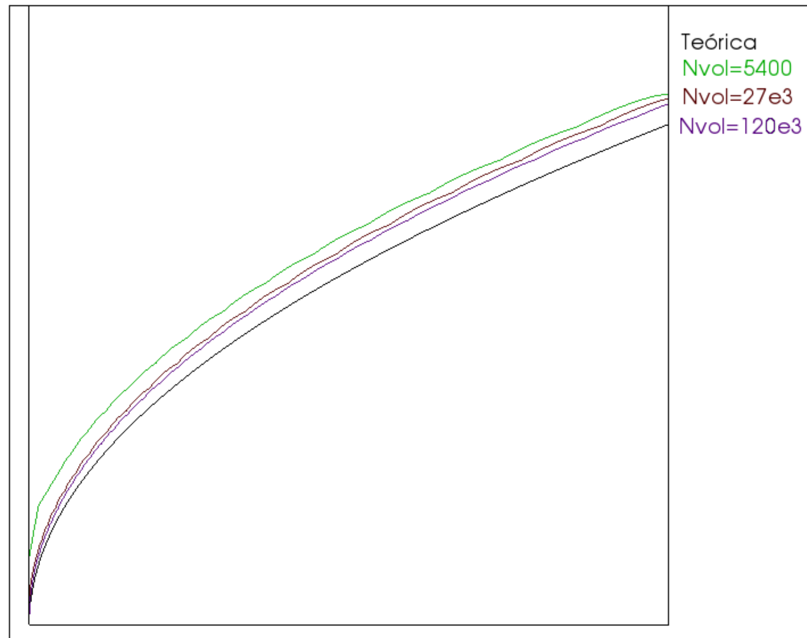


Figura 34: Capa límite térmica δ_{ter} del caso NSCL.



Para comparar los datos obtenidos, se determina el espesor de la capa hidráulica y térmica cada metro para la simulación de 120000 volúmenes de control. El primer dato se toma para $Re = 1000$ recordando que la ecuación de Blasius es válida para $Re > 1000$.

Tabla 14: Comparación de resultados para el grosor de la capa límite hidráulica del caso NSCL.

		Capa límite hidráulica δ_h			
Re $[-]$	x $[m]$	Valor Calculado $[m]$	Valor Teórico $[m]$	Error absoluto $[m]$	Error porcentual $[\%]$
1000	0,01585	0,0021	0,0026	0,0005	19
25231	0,4	0,0119	0,0126	0,0006	5
63079	1	0,0196	0,0199	0,0003	1
126157	2	0,0281	0,0282	0,0001	0
189236	3	0,0344	0,0345	0,0001	0
252314	4	0,0396	0,0398	0,0002	1
315393	5	0,0440	0,0445	0,0005	1
378471	6	0,0479	0,0488	0,0009	2
441550	7	0,0514	0,0527	0,0013	2
504629	8	0,0545	0,0563	0,0018	3

Tabla 15: Comparación de resultados para el grosor de la capa límite térmica del caso NSCL.

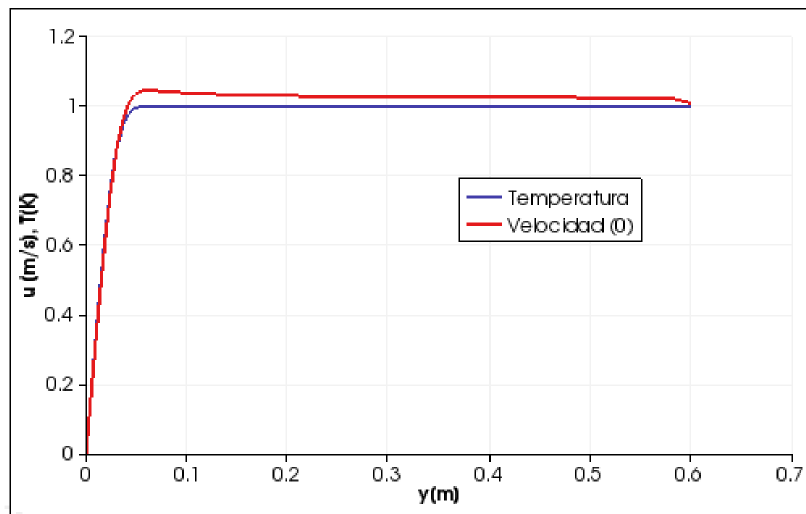
		Capa límite térmica δ_{ter}			
Re $[-]$	x $[m]$	Valor Teórico $[m]$	Valor Calculado $[m]$	Error absoluto $[m]$	Error porcentual $[\%]$
1000	0,01585	0,0029	0,0037	-0,0008	-28
63079	1	0,0221	0,0239	-0,0018	-8
126157	2	0,0313	0,0334	-0,0021	-7
189236	3	0,0383	0,0406	-0,0023	-6
252314	4	0,0442	0,0467	-0,0025	-6
315393	5	0,0495	0,0520	-0,0025	-5
378471	6	0,0542	0,0568	-0,0026	-5
441550	7	0,0585	0,0611	-0,0026	-4
504629	8	0,0626	0,0651	-0,0026	-4

Tomando como error permisible el 5%, son aceptables los valores del espesor de la capa límite hidráulica para $Re \approx 25000$. Con respecto al espesor de la capa límite térmica, se presentan

valores con un mayor margen de error alcanzando un error menor al 5% para $Re > 316000$.

Para el aire el espesor de la capa límite térmica es mayor que el espesor de la capa límite hidráulica debido al valor del Prandtl $Pr_{aire} = 0.729 < 1$, es por esto que el campo de temperatura se ve afectado por la velocidad del fluido fuera de la capa límite. En la figura 35 se muestra un sobrepaso del valor de la velocidad $u_{max} > 1$ que afecta al campo de temperatura.

Figura 35: Perfil de velocidad u y temperatura T para $x = 4[m]$

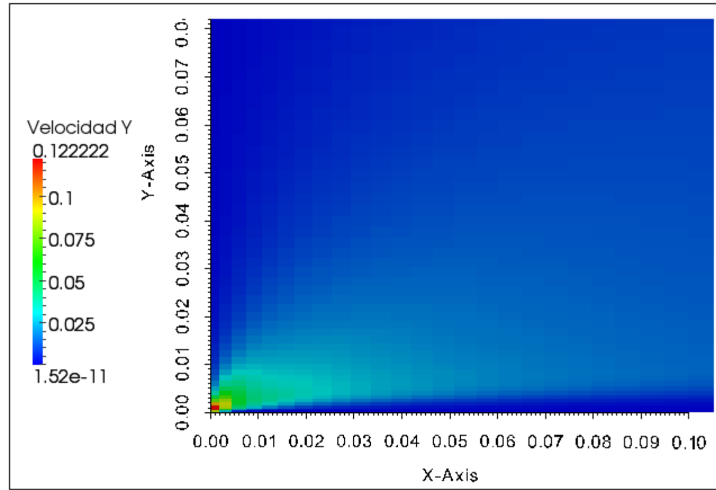


La temperatura no presenta un sobrepaso por dos motivos. En primer lugar, no existe generación de calor para el fluido y en segundo lugar, el cumplimiento del balance de masa elimina este tipo de problemas.

Para las ecuaciones de la cantidad de movimiento se agrega el gradiente del tensor de esfuerzos y el gradiente de la presión como término fuente por lo que es posible que se presente un sobrepaso de la velocidad u. En la ecuación de la cantidad de movimiento en y el término fuente es el encargado que se produzca valores diferentes de cero porque las condiciones de frontera son `Dirichlet=0` y `Neumann=0` para la velocidad v.

De acuerdo a las ecuaciones (Ec 1.27b) y (Ec 1.27c) el valor de la velocidad v fuera de la capa límite tiene una mayor variación en la dirección x que en la dirección y. En dirección $y+$ el valor de v crece hasta un valor máximo.

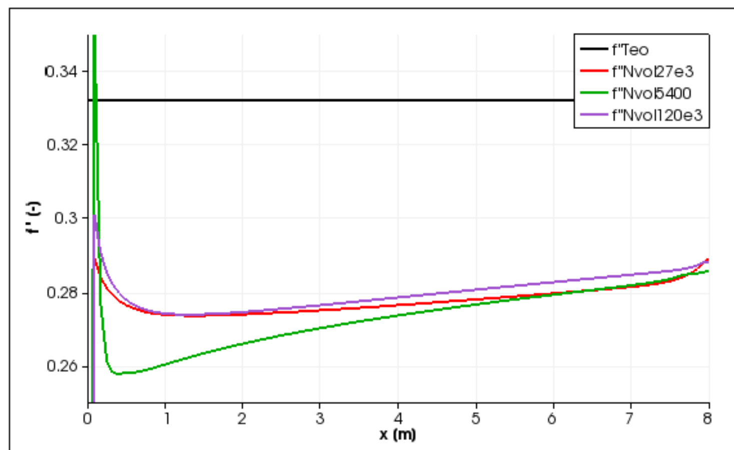
Figura 36: Velocidad v para NSCL



Los resultados encontrados no tienen este comportamiento como se muestra en la figura 36. La ecuación de la cantidad de movimiento en y se linealiza como una ecuación de convección difusión para v, la convección en x ($u >> v$) y la difusión son las encargadas de "disipar" la velocidad v. El valor inferior de la velocidad v es la responsable del sobrepaso de la velocidad u (por el balance de masa) y en general de modificar el perfil de la velocidad u.

Para continuar la comparación de los resultados se presenta ahora el valor de $f''|_{y=0} = 0.332$ mostrado en la ecuación (Ec 1.27d). El valor de f'' encontrado a lo largo de la placa para las simulaciones propuestas se muestra en la figura 37 y en la tabla 16 se presentan los valores para la simulación de 120000 volúmenes de control.

Figura 37: Valor f'' en la superficie para NSCL.



El valor de $f''|_{y=0}$ encontrado varía significativamente para los primeros $3[m]$ de la placa y a partir de $x = 1.5[m]$ es una función creciente para las tres simulaciones. En la figura 37 se observa que el valor encontrado es muy inferior al valor predecido por Blasius; la teoría de Blasius predice un valor constante, el valor encontrado depende de la posición.

Tabla 16: Comparación de resultados para $f''_{y=0}$ en el caso NSCL.

		Valor de f'' en la superficie			
Re $[-]$	x $[m]$	Valor Teórico $[-]$	Valor Calculado $[-]$	Error absoluto $[-]$	Error porcentual $[\%]$
1000	0,01585	0,332	0,060	0,272	82
63079	1	0,332	0,274	0,058	17
126157	2	0,332	0,275	0,057	17
189236	3	0,332	0,276	0,056	17
252314	4	0,332	0,279	0,053	16
315393	5	0,332	0,281	0,051	15
378471	6	0,332	0,283	0,049	15
441550	7	0,332	0,285	0,047	14
504629	8	0,332	0,288	0,044	13

El valor de $f''|_{y=0}$ está relacionado con el coeficiente local de fricción que se muestra en la figura 38 y en la tabla 17. El valor teórico de $C_{f,x}$ se muestra en (Ec 1.26).

Figura 38: Coeficiente local de fricción para NSCL.

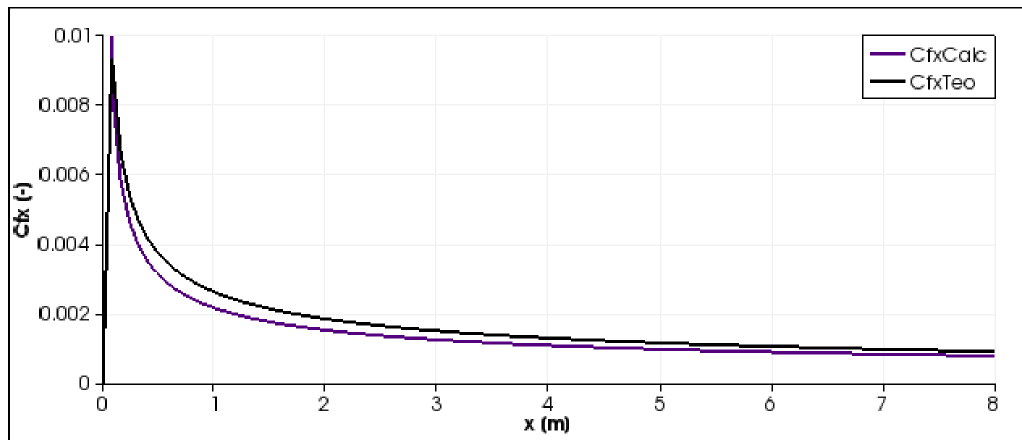
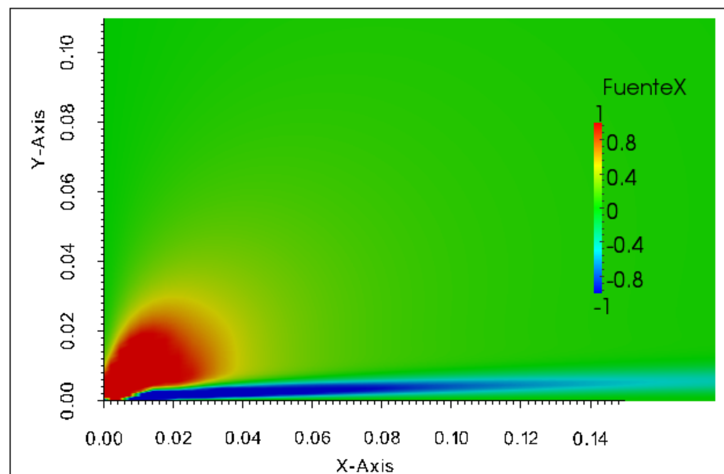


Tabla 17: Comparación de resultados para $C_{f,x}$ en el caso NSCL.

		Valor del coeficiente local de fricción			
Re $[-]$	x $[m]$	Valor Teórico $[-]$	Valor Calculado $[-]$	Error absoluto $[-]$	Error porcentual $[\%]$
1000	0,01585	2,10E-02	1,06E-02	1,04E-02	49
63079	1	2,65E-03	2,18E-03	4,60E-04	17
126157	2	1,87E-03	1,55E-03	3,23E-04	17
189236	3	1,53E-03	1,27E-03	2,55E-04	17
252314	4	1,32E-03	1,11E-03	2,13E-04	16
315393	5	1,18E-03	1,00E-03	1,83E-04	15
378471	6	1,08E-03	9,19E-04	1,60E-04	15
441550	7	9,99E-04	8,57E-04	1,42E-04	14
504629	8	9,35E-04	8,11E-04	1,24E-04	13

Como era de esperarse, si el valor encontrado de $f''|_{y=0}$ difiere del valor teórico, el valor de $C_{f,x}$ encontrado también difiere del valor teórico. El valor de $f''|_{y=0}$ y $C_{f,x}$ depende de la variación de la velocidad u en dirección y, como se mostró en la figura 35, aunque el espesor de la capa límite hidráulica encontrado sea aceptable, el perfil de la velocidad u en dirección y es diferente porque se presenta el sobresalto en el valor de la velocidad u, por esta razón, el valor de $f''|_{y=0}$ y $C_{f,x}$ no se acerca al valor teórico.

Figura 39: Término fuente para la ecuación de la cantidad de movimiento en dirección x.



En la figura 39 se muestra el término fuente para la ecuación de la cantidad de movimiento en x, este término es la resta del gradiente del tensor de esfuerzos en dirección x menos el

gradiente de la presión en la misma dirección; los valores negativos (color azul) producen una disminución en la velocidad u, estos valores no se presentan en la frontera con la placa por lo que modifican el perfil de velocidad.

El último valor a comparar es el coeficiente adimensional de transferencia de calor. El valor encontrado para las simulaciones realizadas se muestra en la figura 40 y en la tabla 18 para 120000 volúmenes de control.

Figura 40: Coeficiente adimensional de transferencia de calor para NSCL.

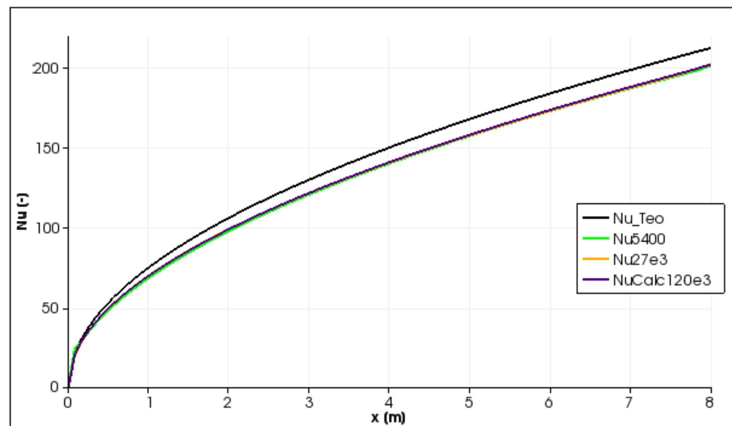


Tabla 18: Comparación de resultados para el coeficiente adimensional de transferencia de calor en el caso NSCL.

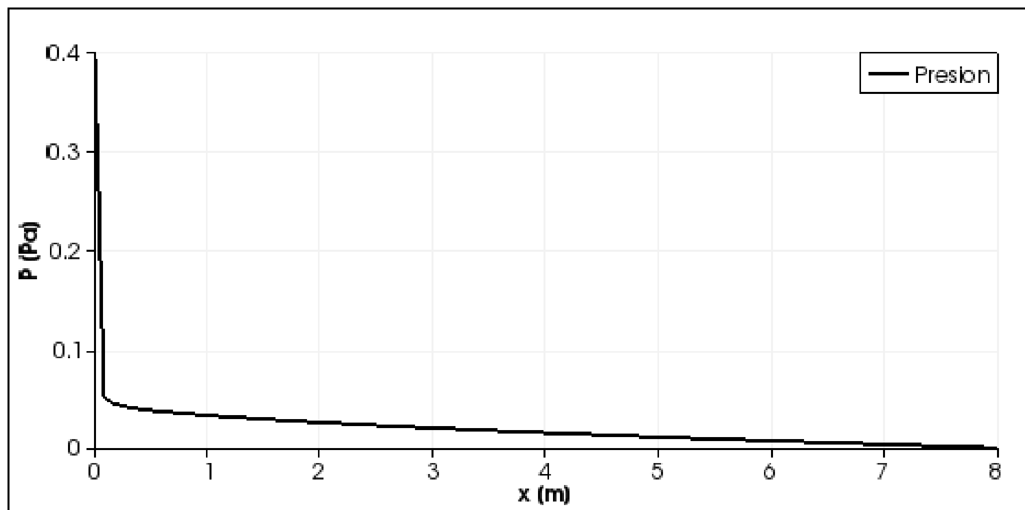
		Valor del Nusselt			
Re	x	Valor Teórico	Valor Calculado	Error absoluto	Error porcentual
$[-]$	$[m]$	$[-]$	$[-]$	$[-]$	$[\%]$
1000	0,01585	9	4	5	57
63079	1	75	70	5	7
126157	2	106	99	7	7
189236	3	130	122	8	6
252314	4	150	141	9	6
315393	5	168	158	10	6
378471	6	184	174	10	6
441550	7	199	188	11	5
504629	8	212	202	10	5

De la figura 40 se deduce que la variación del Nu con la malla es mínima en comparación con el error existente entre los valores encontrados y el valor teórico. De la tabla 18 se observa

porcentajes de error menores para el Nu comparado con el error del valor hallado de $C_{f,x}$, excepto para $Re = 1000$.

A continuación se muestra el valor de la presión sobre la placa en la figura 41 en la que, se puede ver un aumento significativo de la presión para bajos valores de Re. El comportamiento de la presión en la figura 41 confirma la existencia de un gradiente de presión para bajos valores de Re. Un estudio detallado del cálculo de la presión, coeficiente de arrastre y transferencia de calor para bajos valores de Re, específicamente para placas finitas puede ser encontrado en Dennis [11] [12].

Figura 41: Presión a lo largo de la placa.



Por último, se analiza la convergencia del método con la simulación de 5400 volúmenes de control. En la figura 42 se muestra la norma del residuo másico como se presentó en la teoría, en esta se puede ver que el valor es aún alto y no varía significativamente con las iteraciones, la principal fuente de este valor para más de 200 iteraciones exteriores es el volumen de control tomado como referencia para la ecuación de corrección de presión. Al dar un valor de referencia a este volumen de control se satisfacen las ecuaciones de los demás volúmenes de control, sin embargo, se altera la ecuación del volumen actual. Es por esta razón que se muestra en la figura 43 el residuo másico sin tener en cuenta el volumen de control de referencia para la ecuación de corrección de presión.

Figura 42: Norma del residuo másico

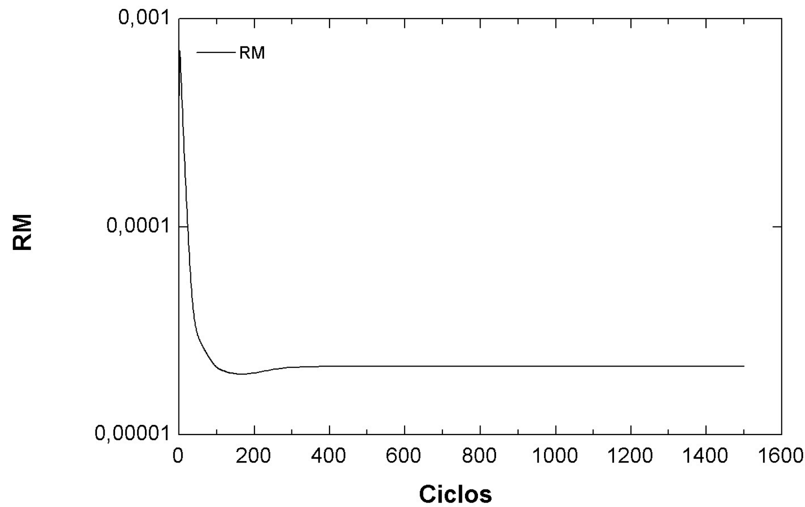
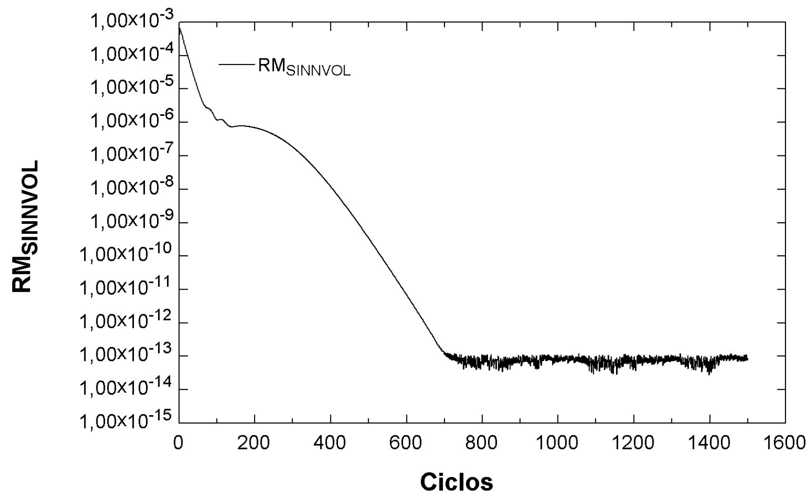


Figura 43: Norma del residuo másico sin tener en cuenta el volumen de control de referencia para la ecuación de corrección de presión



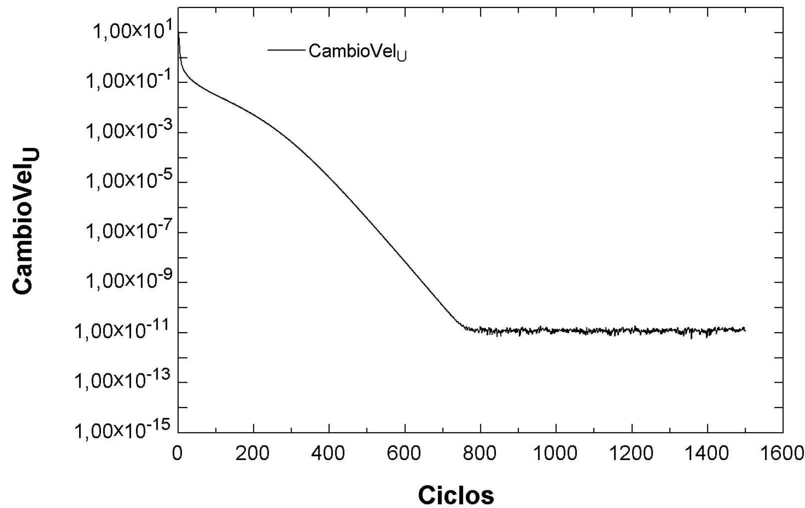
En la figura 43 se observa que el residuo másico disminuye hasta un valor cercano a $1e-13[Kg/s]$ en donde se presentan algunas oscilaciones pequeñas, aproximadamente para 800 iteraciones exteriores. Esta oscilación es el resultado de la precisión seleccionada para la solución de la ecuación de correción de presión. Aunque el residuo másico para el volumen de control tomado de referencia para la ecuación de corrección de presion es no despreciable, la perturbación en otros volúmenes de control si es despreciable. El volumen tomado como referencia es la última celda que corresponde al volumen de control de la esquina superior

derecha la cual está muy lejos de la zona en donde se desarrolla la capa límite y está en la zona de salida del flujo por lo que no es posible que se propague este error a otros volúmenes de control.

Otra forma de comparar la convergencia del método es la variación de la velocidad u en cada iteración de acuerdo a (Ec 6.21), los resultados se muestran en la figura 44. De nuevo el valor disminuye hasta llegar a las 800 iteraciones exteriores en donde presenta pequeñas oscilaciones por la precisión seleccionada para la solución de las ecuaciones lineales.

$$\Delta_u = ||Vel_u - Vel_{u,ant}|| \qquad \text{(Ec 6.21)}$$

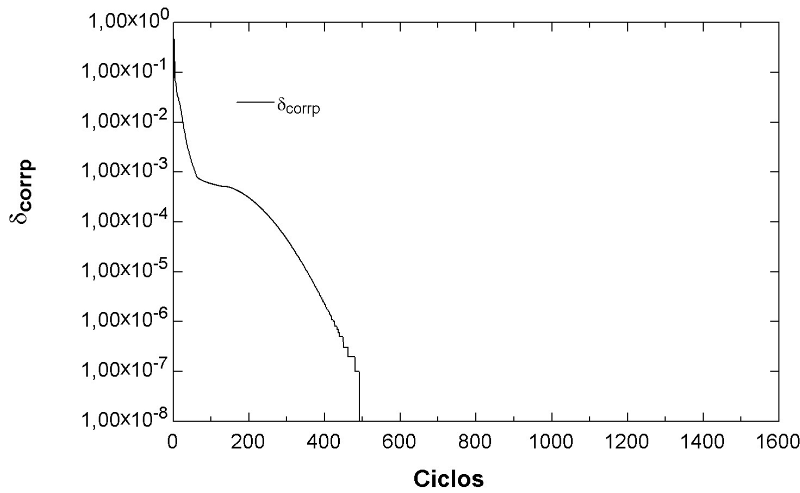
Figura 44: Variación de la velocidad u.



Finalmente se presenta $\Delta P_{corr,max}$ definido en (Ec 6.22). La corrección de presión debe ser cercana a cero cuando se ha alcanzado la convergencia, en la figura 45 se presenta este comportamiento limitado al número de decimales de los resultados exportados, debido a que el residuo másico del punto tomado como referencia no es cero, el valor mínimo de la corrección de presión no es cero pero si el cambio de la corrección de presión por lo que no introduce errores a la solución, un valor mínimo de correción de presión diferente de cero modifica la precisión de los resultados para la solución de la ecuación lineal, sin embargo, ya se ha alcanzado la convergencia del problema al llegar a este punto.

$$\Delta P_{corr,max} = P_{corr,max} - P_{corr,min} \qquad \text{(Ec 6.22)}$$

Figura 45: Máximo ΔP_{corr}.



6.5 MÉTODO SIMPLIFICADO PARA LA SOLUCIÓN DE LAS ECUACIONES DE LA CAPA LÍMITE: NO-SIMPLE

El método SIMPLE resuelve cada una de las ecuaciones de la cantidad de movimiento y la ecuación de continuidad en cada iteración exterior. Sin embargo, no es posible encontrar la solución de las ecuaciones simplificadas de la capa límite con el método SIMPLE ya que la ecuación de la continuidad se resuelve a partir de una correción de presión. Dadas las simplificaciones realizadas en la capa límite, el gradiente de presión en dirección normal a la superficie es 0. El método presentado a continuación resuelve directamente la correción de la velocidad v sin necesidad de acudir a la corrección de presión.

Se re-escriben las ecuaciones simplificadas de la capa límite para facilitar el entendimiento del método:

$$\nabla \cdot (\rho \vec{V}) = 0 \qquad \text{(Ec 6.23)}$$

$$\rho u \frac{\partial u}{\partial x} + \rho v \frac{\partial u}{\partial y} = -\frac{dp}{dx} + \mu \frac{\partial^2 u}{\partial y^2} \qquad \text{(Ec 6.24)}$$

$$\rho \left[u \frac{\partial T}{\partial x} + v \frac{\partial T}{\partial y} \right] = \frac{k}{C_p} \frac{\partial^2 T}{\partial y^2} \qquad \text{(Ec 6.25)}$$

El **primer paso** consiste en resolver la ecuación de la cantidad de movimiento en dirección x (Ec 6.26a) y actualizar el valor de v_p^{m*}.

$$a_p^u\, u_p^{m*} = \sum_{i=1}^{n-nb} [\, a_i^u\, u_i^{m*}] - \nabla p_p^{m-1} \Omega_p \cdot \widehat{i} + b_p^u \Omega_p \qquad \text{(Ec 6.26a)}$$

$$v_p^{m*} = v_p^{m-1} \qquad \text{(Ec 6.26b)}$$

La (Ec 6.26a) puede converger por sí misma para un valor de v por lo que la corrección de la velocidad u se asume cero.

$$u = u^{m*} + \cancelto{0}{u'} \qquad \text{(Ec 6.27)}$$

Por otro lado, para obtener los valores correctos en la ecuación de cantidad de movimiento en x es necesario corregir la velocidad v en dirección y. Esta correción se muestra en (Ec 6.28).

$$v = v^{m*} + v' \qquad \text{(Ec 6.28)}$$

La simplificación realizada en la correción de la velocidad es la clave para resolver directamente la ecuación de continuidad sin necesidad de definir una ecuación de correción de presión.

$$0 = \nabla \cdot (\rho \vec{V}) = \nabla \cdot [\rho(\vec{V}^{m*} + \vec{V}')] = \nabla \cdot [\rho \vec{V}^{m*}] + \nabla \cdot [\rho \vec{V}']$$

$$\nabla \cdot [\rho \vec{V}'] = \frac{\partial(\rho v')}{\partial y} = -\nabla \cdot [\rho \vec{V}^{m*}]$$

La (Ec 6.29a), que corresponde al **segundo paso**, es una ecuación de convección con una constante difusiva 0; para resolver esta ecuación se puede usar el método presentado en el capítulo 4 o 5 para un valor de $\Gamma = 0$. El esquema de aproximación para el término convectivo no puede ser CDS ya que generaría oscilaciones, se sugiere el esquema UDS.

$$\nabla \cdot [\rho v' \vec{\vartheta}] = -\nabla \cdot [\rho \vec{V}^{m*}] \qquad \text{(Ec 6.29a)}$$

$$\vec{\vartheta} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \qquad \text{(Ec 6.29b)}$$

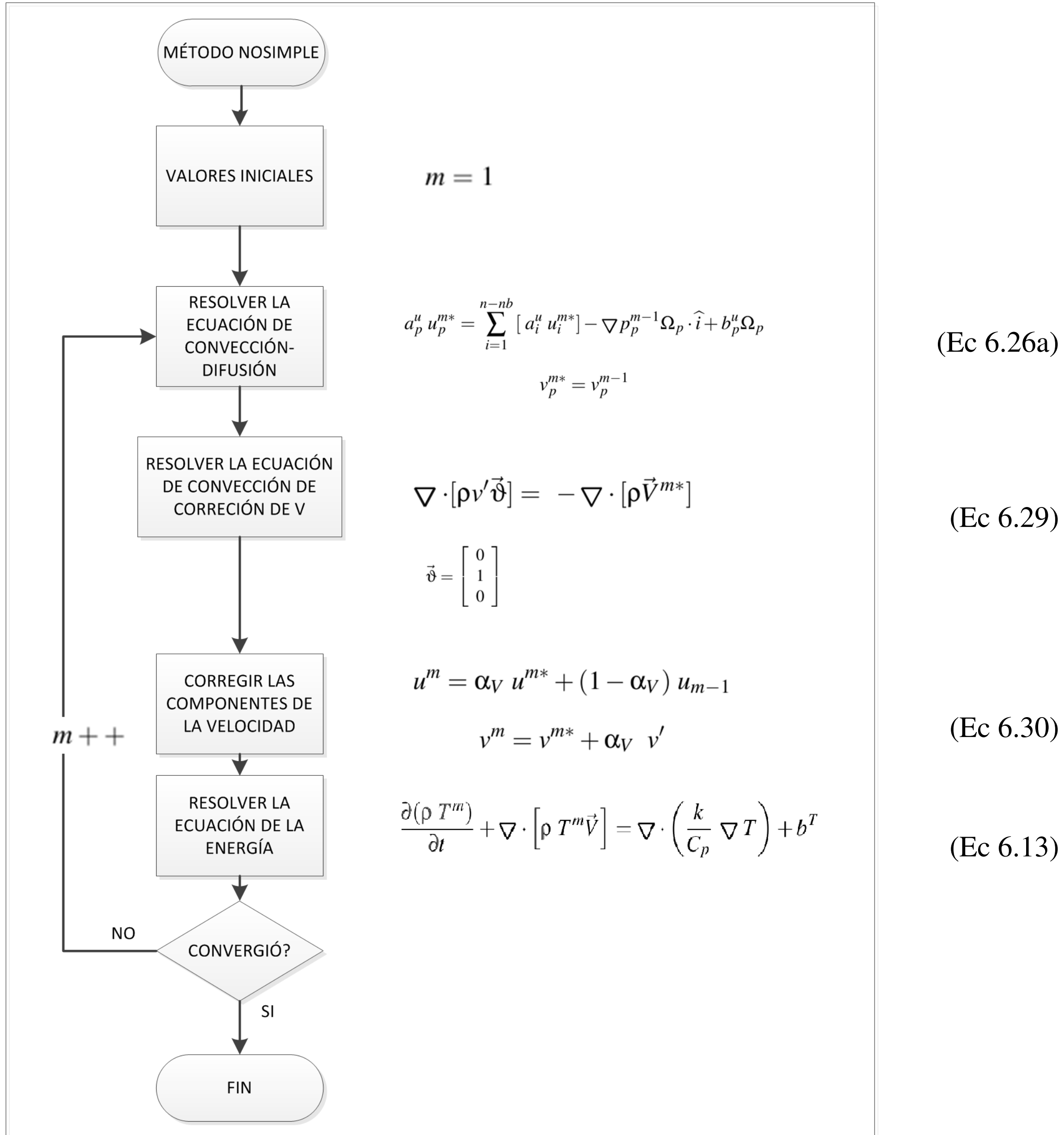
Las condiciones de frontera para la velocidad v son utilizadas para hallar el residuo másico de las celdas en la frontera por lo que se asegura que el flujo en la placa sea cero.

En el **tercer paso** se corrige el valor de la velocidad u y v como se muestra en (Ec 6.30).

$$u^m = \alpha_V\, u^{m*} + (1 - \alpha_V)\, u_{m-1} \qquad \text{(Ec 6.30a)}$$

$$v^m = v^{m*} + \alpha_V\;\, v' \qquad \text{(Ec 6.30b)}$$

Figura 46: Método simplificado para la solución de las ecuaciones de la capa límite sin realizar correción de presión, NOSIMPLE.



El **cuarto paso** consiste en resolver las demás ecuaciones, en este caso, la ecuación de la energía[8]. Para el caso que solo se presenten los resultados finales, es eficiente resolver la ecuación de la energía en estado estable una vez se ha terminado el proceso iterativo.

[8]Se asume la densidad es constante por lo que no se corrige el valor de esta en cada iteración.

El algoritmo de este método se muestra en la figura 46. Si bien es cierto en la implementación las ecuaciones de la capa límite aún se resuelven como ecuaciones elípticas, el término $\partial^2 u/\partial x^2$ es despreciable y se tiene en cuenta para no generar un único caso específico de uso de las librerías.

El uso de la correción de la velocidad en vez de la correción de presión requiere establecer correctamente las condiciones de frontera para resolver la ecuación del segundo paso. En el método SIMPLE el término fuente de las ecuaciones de la cantidad de movimiento es obtenido, en parte, por el gradiente de presión, de esta forma, el valor absoluto de la presión no importa.

En el método NOSIMPLE se pueden llegar a obtener resultados erróneos si no se da un valor de referencia para el cálculo del segundo paso. Se recomienda usar la condición tipo Neumann para todas las fronteras a excepción de la placa, la condición de frontera en la placa para la correción de velocidad debe ser de tipo Dirichlet y valer 0. Incluso si se definiera una velocidad v inicial diferente de cero, el método es capaz de reducir la velocidad a un valor cercano a cero[9] para los volúmenes de control que colindan con la placa.

6.6 IMPLEMENTACIÓN

Los pasos específicos que se agregan para utilizar el método NOSIMPLE recientemente explicado son:

```
...
class NavierStokes : protected GuardarEnsight
{
...
protected:
...
bool PrimerPasoNOSIMPLE();
bool SegundoPasoNOSIMPLE();
bool TercerPasoNOSIMPLE();
...
}
```

Para resolver el cuarto paso del método se utiliza el mismo proceso realizado por el método SIMPLE.

1. Método `PrimerPasoNOSIMPLE()`: la ecuación de la cantidad de movimiento en x obtenida por las simplificaciones de la capa límite se resuelve como una ecuación de convección-difusión en donde, el término fuente es el gradiente de la presión en dirección x[10].

[9] La "cercanía" al valor 0 está limitada por el tamaño de los volúmenes de control en la zona.

[10] En este caso el gradiente del tensor de esfuerzos no contribuye al término fuente en la ecuación de cantidad de movimiento en dirección x.

```
bool NavierStokes::PrimerPasoNOSIMPLE()
{
bool correcto; int i;
Vector<real> dPdn(nvol);
Fuente.llenar(0.0);
for (i=1; i<=nvol; i++) {dPdn (i) = GP (i)(1); }
Fuente -= dPdn;
for (i=1; i<=nvol; i++) Fuente(i) = Fuente(i) * Volumen(i);
ConvVu.SetFuenteYFlujo(Fuente,VelU,VelV, VelW);
ConvVu.SetPasado(VelU_ant);
ConvVu.SetPropiedades (rho, mu);
CoefU  = ConvVu.CalcularCoefUnTiempo
(theta, tiempo_p, MetodoAprox,  correcto);
VmAstU = ConvVu.SolucionCoefUnTiempo (correcto);
////No se resuelve la ecuacion de cant. mov. en y///////////////
//////////////////////, se realiza lo siguiente//////////
CoefV = CoefU;//aproximacion
VmAstV = VelV;//asignacion
return true;
}
```

La ecuación de la cantidad de movimiento en y no se resuelve, en cambio, se actualiza el valor de `VmAstV` que será utilizado para hallar el residuo másico.

2. Método `SegundoPasoNOSIMPLE()`: en este paso se calcula la correción de la velocidad v.

I. La primera tarea a realizar es declarar e inicializar las variables a utilizar en el método. Una vez realizado este proceso se puede calcular el valor del residuo másico de cada volumen de control.

```
bool NavierStokes::SegundoPasoNOSIMPLE()
{
bool triunfo; int i;
Vector<real> cero, uno;
Vector_basico<Vector_xyz<real>> Velvec(nvol);
cero.chaSize (nvol); uno.chaSize(nvol);
cero.llenar (0.0), uno.llenar(1.0);
for ( i=1 ; i<=nvol ; i++) {
Velvec(i).chaSize(ndim);
Velvec(i)(1) = VmAstU(i);
Velvec(i)(2) = VmAstV(i);
if (ndim == 3)
 Velvec(i)(3) = VmAstW(i);
```

```
}
NablaDotVecEspecial(Velvec,ResiduoMasico, MULT_POR_RHO_ON );
for(i=1;i<=nvol;i++) ResiduoMasico(i) = ResiduoMasico(i)*Volumen(i);
Fuente = ResiduoMasico * (-1.0);
...
```

II. Se introduce el término fuente, el campo de la velocidad y la constante difusiva según se especificó en la (Ec 6.29). La ecuación se resuelve en estado estable y por último se obtiene el vector `vcor`.

```
...
Corrv.SetFuenteYFlujo(Fuente,cero,uno, cero);
Corrv.SetPropiedades (rho, mu*1e-6);
Corrv.CalcularCoefUnTiempo (theta, t_aux, MetodoAprox,  triunfo);
vcor = Corrv.SolucionCoefUnTiempo (triunfo);
return true;
}
```

3. Método `TercerPasoNOSIMPLE()`: teniendo en cuenta el parámetro de sobrerelajación, se actualiza el valor de la velocidad u con el valor anterior u^{m-1} y el valor hallado u^{m*} en el presente ciclo. Para la velocidad v se agrega la correción de la velocidad al valor de la velocidad en la iteración anterior.

```
bool NavierStokes::TercerPasoNOSIMPLE()
{
int i;
real alpV = Subrelajacion;
for (i=1; i<= nvol ; i++) {
VelU(i) = alpV * VmAstU(i) + (1.0-alpV) * VelU(i);
VelV(i) = VelV(i) + alpV*vcor(i);
}
return true;
}
```

El siguiente fragmento de código es un ejemplo de la creación de un objeto que utilice el método simplificado para resolver el problema del flujo en la capa límite.

```
NavierStokes NS;
NS.SubRelajaSIMPLE(1.2);
NS.SetParametros( false, 100, "hibrido", "NOSIMPLE",true);
NS.SetMAssThreshold(1e-9); NS.SetMinimaPresion(0.0);
NS.Leer("ARCH=NOSIMPLELIMITE4.txt", "ARCH=Solve_datos.txt",
   "dt=[0.10 1.0] t en [0 2.1 1000.0]",1.0,
   "ARCH=Solve_datostemp.txt");
   NS.CalcularTeoriaCLimite(); NS.Calcular();
```

El archivo ``NOSIMPLELIMITE4.txt'' contiene:

```
> mallador = PREPROCESSOR=ConjMalladorSupElem/ARCH=NOSIMPLELIMITE4.geom
/ARCH=NOSIMPLELIMITE4.parts
```

El archivo NOSIMPLELIMITE4.geom contiene:

```
> nsd = 2;
> subdominios = 1;
> no_de_superels = 1;
>nbind = 4
>bname VelUNeumann=0,VelVNeumann=0,PNeumann=0,VcorNeumann=0,TempNeumann=0
VelUDirichlet=1,VelVNeumann=0,PNeumann=0,VcorNeumann=0,TempDirichlet=1
VelUDirichlet=1,VelVDirichlet=0,PNeumann=0,VcorNeumann=0,TempDirichlet=1
VelUDirichlet=0,VelVDirichlet=0,PNeumann=0,VcorDirichlet=0,TempDirichlet=0
>SupEl;
>subdominio_no = 1;
>tipoelemento = ElmB4n2D;
>fronteras = [1(1)] [2 (2)] [3 (3)] [4(4)];
>nodos = [1(0 0)]+[2(8.0 0)]+[3(0 0.12)]+[4(8.0 0.12)];
>lados=;
```

No es necesario definir las condiciones de frontera para la corrección de presión pero se sugiere para facilitar el cambio de método de solución. Por último el archivo de las particiones contiene la siguiente información:

```
>nsd = 2;
>no_de_superels = 1;

>SE;>d=2;>e=ElmB4n2D;>div=[150,60];>grad=[-0.9,-0.6];
```

El volumen de control general a simular está limitado en dirección x por el final de la zona de flujo laminar $Re = 5 * 10^5$. Para la dirección y se tiene en cuenta la recomendación de Cebeci[11] ($\eta = 8$) para simular el flujo laminar en la capa límite por el método de diferencias finitas, para el presente caso se usa la relación dada en (Ec 6.31).

$$\eta_{(Re=5E5,y=ymax)} \geq 8 \approx 1.6\,\delta_{hid}$$
$$y_{max} = 1.6\,max(\delta_{hid},\delta_{ter}) \qquad \text{(Ec 6.31)}$$

[11]CEBECI, TUNCER, et al. Computational fluid dynamics for engineers, Berlin: Springer, 2005. p. 225

6.7 SIMULACIÓN DE LAS ECUACIONES SIMPLIFICADAS DE LA CAPA LÍMITE SOBRE UNA PLACA PLANA

Para probar el método NOSIMPLE se desarrollan dos ejemplos. Por un lado se resuelven las ecuaciones sin gradiente de presión y en un segundo análisis se define un perfil de velocidad externo.

Los datos teóricos para la presente sección se toman de las ecuaciones (Ec 1.26) y (Ec 1.27) a excepción que se indique una fuente bibliográfica externa.

6.7.1 Simulación de las ecuaciones de Blasius incluyendo el término adicional $\partial^2 u/\partial x^2$.

En este caso (de ahora en adelante se llamará Blasius1) se simulará las ecuaciones de Blasius (incluyendo el término $\partial^2 u/\partial x^2$) junto con la capa térmica. De nuevo se usa al aire como el fluido de análisis con las propiedades mostradas en (Ec 6.19). Los parámetros de simulación son los siguientes:

$$\begin{array}{lll} x = 8[m] & U = 1[m/s] & Re_{x=8[m]} = 504629 \\ \delta_{ter,t} = 0,06256[m] & \delta_y = 0.1[m] & n_{vol} = variable \\ \text{Método:NOSIMPLE} & m_{itera} = 100 & \alpha_V = 1.2 \end{array}$$

$$\Delta t\ [s] = \begin{cases} 0.1 & \text{si} \quad 0 < t < 1 \\ 2.1 & \text{si} \quad 1 \leq t < 10 \\ 1 & \text{si} \quad 2.1 < t < 1000 \end{cases} \tag{Ec 6.32}$$

El parámetro de sobrerelajación de la velocidad ha sido tomado como $\alpha_V = 1.2$ con el cual se alcanza un residuo másico menor a 1e-8 para aproximadamente 75 iteraciones exteriores en casi todos los casos desarrollados. Un valor superior para el parámetro de sobrerelajación de la velocidad puede conducir a una solución no convergente.

La simulación se realiza bajo los siguientes parámetros geométricos:

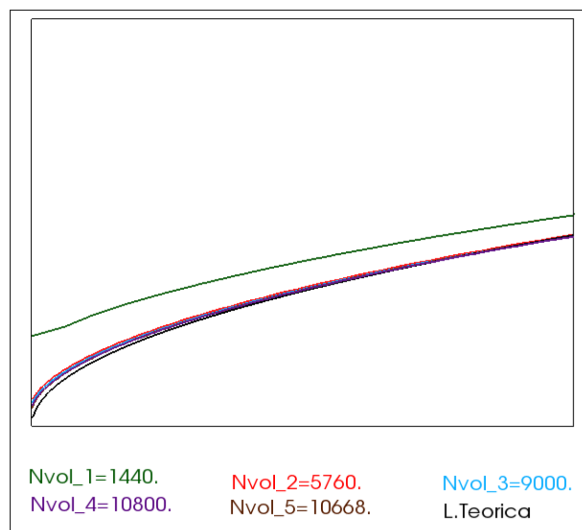
```
> nsd = 2;
> subdominios = 1;
> no_de_superels = 1;
>nbind = 4
>bname VelUNeumann=0,VelVNeumann=0,PNeumann=0,VcorNeumann=0,TempNeumann=0
VelUDirichlet=1,VelVNeumann=0,PNeumann=0,VcorNeumann=0,TempDirichlet=1
VelUDirichlet=1,VelVDirichlet=0,PNeumann=0,VcorNeumann=0,TempDirichlet=1
VelUDirichlet=0,VelVDirichlet=0,PNeumann=0,VcorDirichlet=0,TempDirichlet=0
```

```
>SupEl;
>subdominio_no = 1;
>tipoelemento = ElmB4n2D;
>fronteras = [1(1)] [2 (2)] [3 (3)] [4(4)];
>nodos = [1(0 0)]+[2(8.0 0)]+[3(0 0.1)]+[4(8.0 0.1)];
>lados=;
```

Como primera comparación se muestra la influencia del número de volúmenes de control en la solución, un fragmento de cada uno de los archivos de particiones es el siguiente:

1. `>div=[48,30];>grad=[1.0,1.0];`
2. `>div=[120,48];>grad=[-0.9,-0.6];`
3. `>div=[150,60];>grad=[-0.9,-0.6];`
4. `>div=[200,54];>grad=[-0.9,-0.6];`
5. `>div=[254,42];>grad=[-0.9,-0.6];`

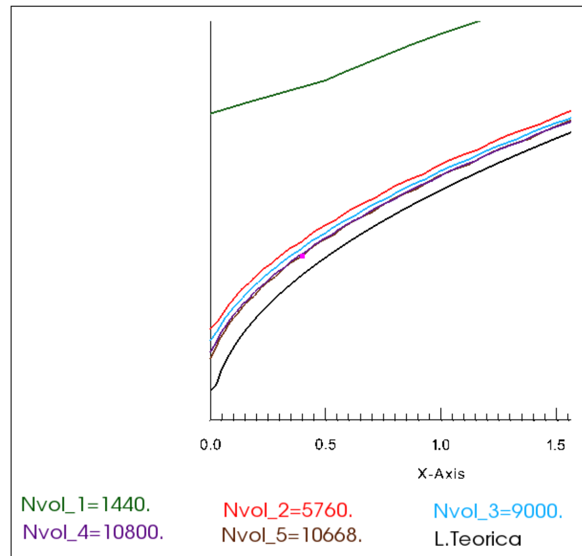
Figura 47: Espesor hidráulico del caso Blasius1.



El primer conjunto de datos a comparar es el espesor de la capa límite hidráulica. En la figura 47 se muestra el espesor de la capa hidráulica para cada uno de los casos planteados, en esta se puede ver que al aumentar el número de volúmenes de control se obtienen mejores resultados que tienden al caso teórico de Blasius. En la figura 48 se muestra en mayor detalle las diferencias para bajos números de Reynolds.

Tomando como datos experimentales los obtenidos por la simulación de 10800 volúmenes

Figura 48: Espesor hidráulico del caso Blasius1 en $x < 1.5[m]$.



de control[12] se construye la siguiente tabla en la que se obtiene el error relativo porcentual a varias distancias del borde de la placa.

Tabla 19: Comparación de resultados para el grosor de la capa límite hidráulica del caso Blasius1.

		Capa límite hidráulica			
Re $[-]$	x $[m]$	Valor Teórico $[m]$	Valor Calculado $[m]$	Error absoluto $[m]$	Error porcentual $[\%]$
9998	0,1585	0,0079	0,0102	-0,0023	-29
63079	1	0,0199	0,0212	-0,0013	-6
126157	2	0,0282	0,0289	-0,0008	-3
189236	3	0,0345	0,0351	-0,0006	-2
252314	4	0,0398	0,0403	-0,0005	-1
315393	5	0,0445	0,0447	-0,0002	-0
378471	6	0,0488	0,0490	-0,0002	-0
441550	7	0,0527	0,0528	-0,0001	-0
504629	8	0,0563	0,0562	0,0001	0

[12]Los datos de todas las tablas presentadas en esta sección se tomarán de la simulación con 10800 volúmenes de control con el método NOSIMPLE.

A partir de la tabla 19 e interpolando los valores se deduce que los resultados obtenidos son válidos para $x > 1.39[m]$ que equivale a $Re > 87500$.

Ahora se compara el espesor de la capa límite térmica y se muestra en la figura 49.

Figura 49: Espesor de la capa límite térmica del caso Blasius1.

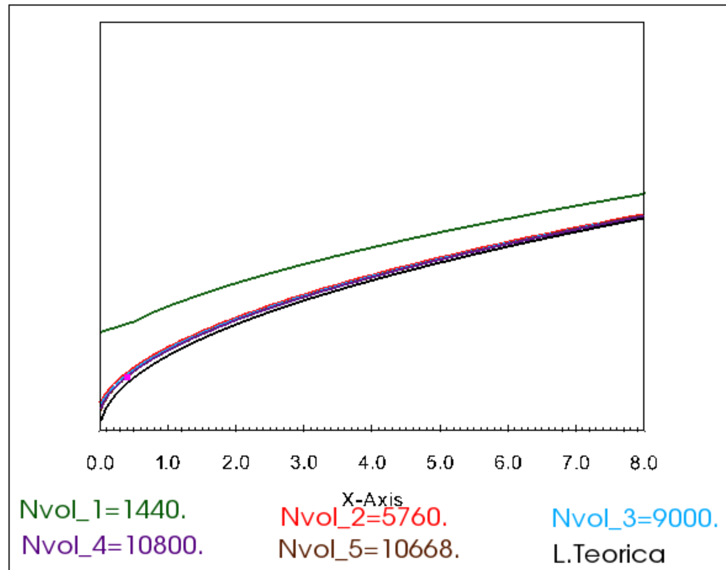


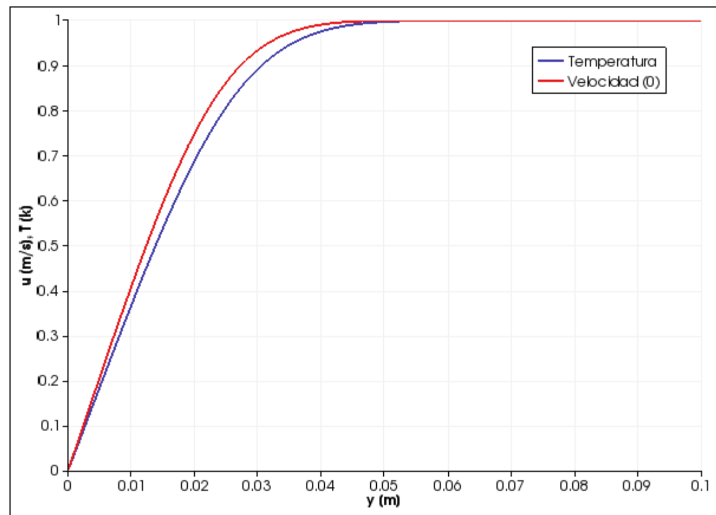
Tabla 20: Comparación de resultados para el grosor de la capa límite térmica del caso Blasius1.

		Capa límite térmica			
Re [−]	x [m]	Valor Teórico [m]	Valor Calculado [m]	Error absoluto [m]	Error porcentual [%]
9998	0,1585	0,0088	0,0112	-0,0024	-27
63079	1	0,0221	0,0237	-0,0016	-7
126157	2	0,0313	0,0326	-0,0013	-4
189236	3	0,0383	0,0393	-0,0010	-3
252314	4	0,0442	0,0452	-0,0009	-2
315393	5	0,0495	0,0504	-0,0009	-2
378471	6	0,0542	0,0551	-0,0009	-2
441550	7	0,0585	0,0594	-0,0009	-2
504629	8	0,0626	0,0632	-0,0007	-1

De acuerdo a la información en la tabla 20 los resultados obtenidos concuerdan con la teoría para $x > 1.72[m]$ o $Re > 108000$.

El perfil de la velocidad u y la temperatura para $x = 4[m]$ se muestra en la figura 50.

Figura 50: Perfil de velocidad del caso Blasius1.



Ahora se compara el valor de $f''|_{y-0}$ en la superficie el cual desde la base teórica es una constante igual a 0.332 como se muestra en la (Ec 1.27d) que se re-escribe a continuación para mayor comodidad:

$$\frac{\mu}{\rho}\frac{\sqrt{Re_x}}{U^2}\left(\frac{\partial u}{\partial y}\right)|_{y=0} \;=\; 0.332 \;=\; cte$$

Para un margen de error del 5% se aceptan los valores entre $0.3154 < f''|_{y-0} < 0.3486$ que, de acuerdo a la figura 51, se cumple para cualquier valor de $x > 0.08[m]$ o $Re > 5000$.
De forma similar se compara el coeficiente local de fricción de (Ec 1.26), los resultados se muestran el la figura 52 y en la tabla 21.

$$C_{f,x} = 0.664\, Re_x^{-1/2}$$

Figura 51: Comparación del valor de $f''|_{y=0}$.

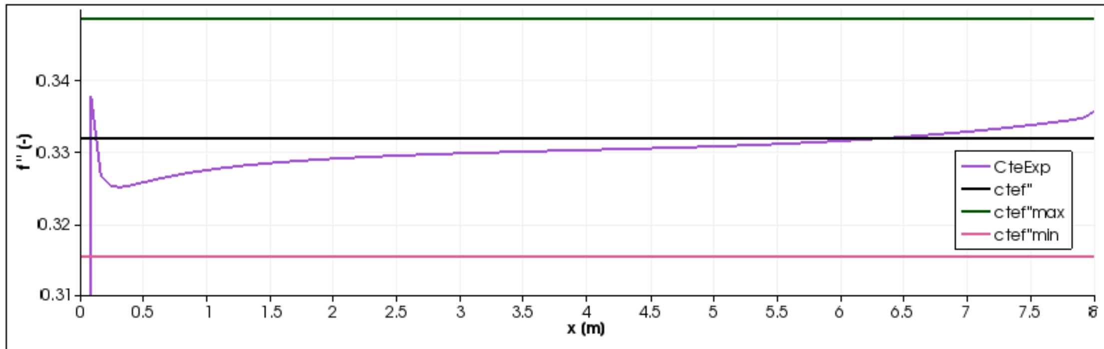


Tabla 21: Comparación de resultados para el coeficiente de fricción local $C_{f,x}$ del caso Blasius1.

		Coeficiente de fricción local $C_{f,x}$			
Re $[-]$	x $[m]$	Valor Teórico $[-]$	Valor Calculado $[-]$	Error absoluto $[-]$	Error porcentual $[\%]$
5046	0,08	0,00940	0,00957	-0,00017	-2
65602	1,04	0,00259	0,00256	0,00003	1
126157	2,00	0,00187	0,00185	0,00002	1
191759	3,04	0,00152	0,00151	0,00001	1
252314	4,00	0,00132	0,00132	0,00001	1
317916	5,04	0,00118	0,00117	0,00000	0
378471	6,00	0,00108	0,00108	0,00000	0
444073	7,04	0,00100	0,00100	-0,00000	0
504629	8,00	0,00093	0,00095	-0,00001	-1

A partir de la tabla 21 se encuentra que el coeficiente de fricción local se ajusta correctamente a los valores encontrados por Blasius.

Por último para el problema hidráulico, se compara el valor máximo teórico de la velocidad en dirección y con el valor encontrado para el y_{max}. Los resultados se muestran en la figura 53 y en la tabla 22.

Figura 52: Comparación del valor del coeficiente de fricción local $C_{f,x}$.

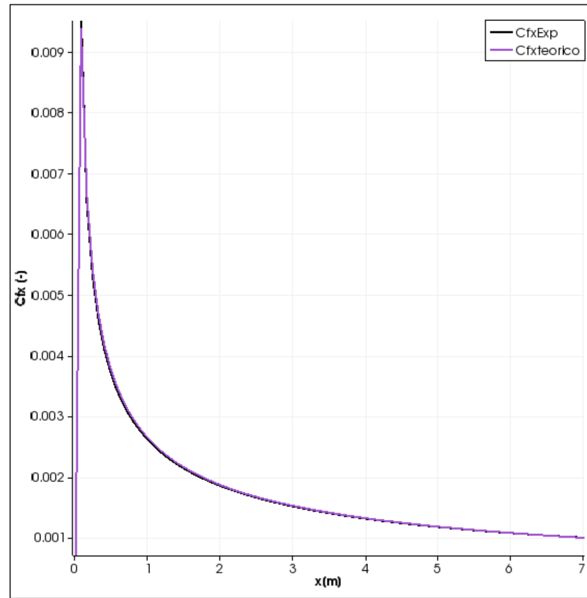
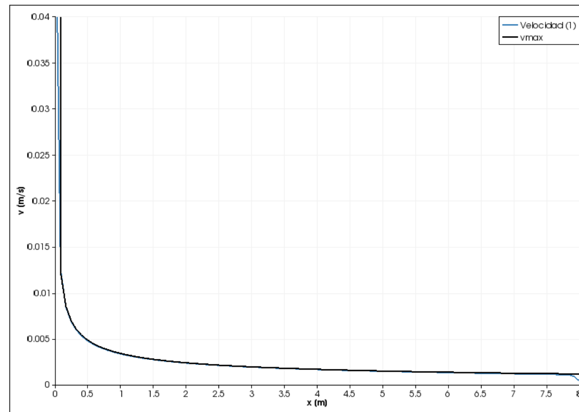


Tabla 22: Comparación de resultados para v_{max} del caso Blasius1.

		Velocidad máxima v_{max}			
Re $[-]$	x $[m]$	Valor Teórico $[m/s]$	Valor Calculado $[m/s]$	Error absoluto $[m/s]$	Error porcentual $[\%]$
5046	0,08	0,01218	0,01245	-0,00027	-2
65602	1,04	0,00336	0,00332	0,00004	1
126157	2,00	0,00242	0,00240	0,00002	1
191759	3,04	0,00196	0,00195	0,00001	1
252314	4,00	0,00171	0,00170	0,00001	1
317916	5,04	0,00153	0,00151	0,00002	1
378471	6,00	0,00140	0,00137	0,00003	2
444073	7,04	0,00129	0,00123	0,00006	5
504629	8,00	0,00121	0,00040	0,00081	67

La mayor parte de los datos de la tabla 22 se ajustan a la teoría excepto para $x = 8[m]$. En la frontera este se introduce un error al definir la variación de la velocidad u en dirección x igual a cero,`VelUNeumann=0`, este valor es correcto para la zona de flujo no viscoso pero, en la zona de la capa límite es diferente de cero. Por la ecuación de continuidad, si no existe variación de la velocidad u en x, tampoco existirá variación de la velocidad v en y por lo que se deduce es la fuente del error encontrado. El error introducido por la definición de las

Figura 53: Comparación de la velocidad v máxima.



condiciones de la frontera `este` se admite y no debería afectar en gran medida los resultados de otras zonas porque el caracter parabólico de las ecuaciones solo permite la propagación de la información en la dirección del flujo.

Retomando el problema térmico en el caso Blasius1, se compara el valor del coeficiente adimensional de transferencia de calor Nu en la tabla 23 y en la figura 54.

Figura 54: Comparación del coeficiente adimensional de transferencia de calor.

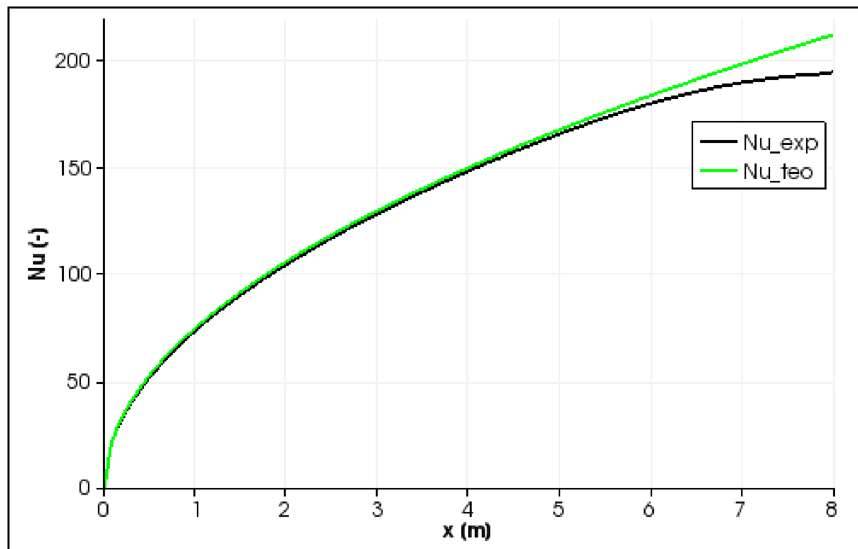


Tabla 23: Comparación de resultados para el coeficiente adimensional de transferencia de calor del caso Blasius1.

Re	x	*Nu*			
Re	x	Valor Teórico	Valor Calculado	Error absoluto	Error porcentual
$[-]$	$[m]$	$[-]$	$[-]$	$[-]$	$[\%]$
5046	0,08	21,2	21,3	-0,1	-1
65602	1,04	76,5	75,2	1,3	2
126157	2,00	106,1	104,7	1,4	1
191759	3,04	130,8	129,3	1,5	1
252314	4,00	150,1	148,5	1,6	1
317916	5,04	168,5	166,5	2,0	1
378471	6,00	183,8	180,1	3,7	2
444073	7,04	199,1	190,1	9,0	5
504629	8,00	212,3	194,9	17,4	8

De nuevo se presentan diferencias significativas para $x = 8[m]$ agregando en este caso, una fuente de error al definir la condición de frontera este para la temperatura como tipo `Neumann=0`.

Como comentario adicional, la simulación Blasius1 se realizó teniendo en cuenta el término $\nu\partial^2 u/\partial x^2$ lo cual es válido para $Re > 1000$ ó, en este caso en específico, $x > 0.0159[m]$. Otras pruebas realizadas que eliminaban este término[13] condujeron a los mismos resultados ya que los volúmenes de control que colindan con la frontera oeste tienen un ancho mayor al valor especificado.

$$\Delta x = 0.01702 > 0.0159 = x_{Re=1000} \quad [m]$$

La convergencia del método NOSIMPLE para la simulación de 10800 volúmenes de control se muestra en la figura 55. El residuo másico alcanza un valor inferior a $\boldsymbol{O}(1e-8)$ en 71 iteraciones exteriores; se presenta un sobresalto para la iteración 22 debido al cambio de tiempo, este cambio de tiempo se realizó para que el problema alcanzara un punto cercano al estado estable antes de llegar al umbral de la norma del residuo másico, de otra manera, el método habría parado a $7[s]$ de la simulación lo cual no sería suficiente para que los resultados correspondan al estado estable. La variación de la velocidad a través de las iteraciones definida en (Ec 6.33) se presenta en la figura 56. También se presenta el sobresalto para la iteración 22 debido al cambio de tiempo pero después de este punto, la variación de la velocidad disminuye hasta $\boldsymbol{O}(1e-6)$ en 85 iteraciones exteriores.

$$CambioVel_U = ||VelU - VelU_{ant}|| \qquad \text{(Ec 6.33)}$$

[13]Para eliminar el término $\nu\partial^2 u/\partial x^2$ se definió la viscosidad de cada volumen de control como un vector que luego era multiplicado por el vector unitario normal a la cara. No se recomienda seguir este proceso porque en un caso real, la viscosidad no es "seleccionada" de acuerdo a la dirección, solo se realizó con fines comparativos.

Figura 55: Convergencia del residuo másico para Blasius1.

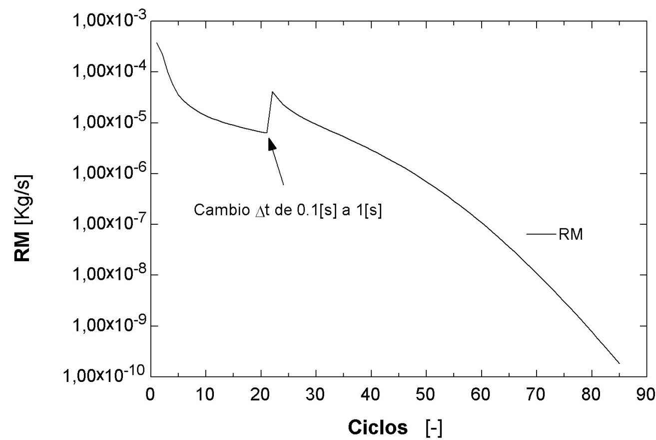
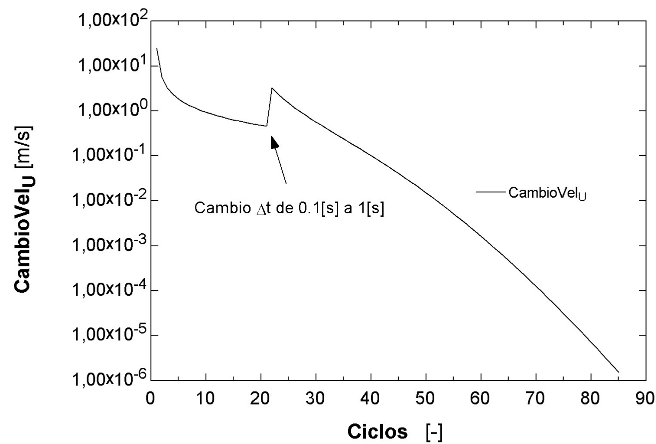


Figura 56: Convergencia de la velocidad para Blasius1.



6.7.2 Comparación de la simulación con un método explicito para el flujo en la capa límite.

Las ecuaciones del flujo en la capa límite son parabólicas lo que significa que la información solo se propaga en la dirección del flujo. Es posible resolver las ecuaciones parabólicas por un método explicito como el presentado en los anexos O, P y Q en los cuales se explica una teoría básica y se muestra el código desarrollado.

Se presentan dos ejemplos, el primer caso se realiza con una malla de 300 nodos en x y 30 nodos en y y el segundo ejemplo con una malla de 600x40. El espesor de la capa límite se presenta en la figura 57 y 58, adicionalmente se muestra el porcentaje de error para ambos casos en la figura 59.

Figura 57: Espesor de la capa límite térmica e hidráulica para 300x30 nodos.

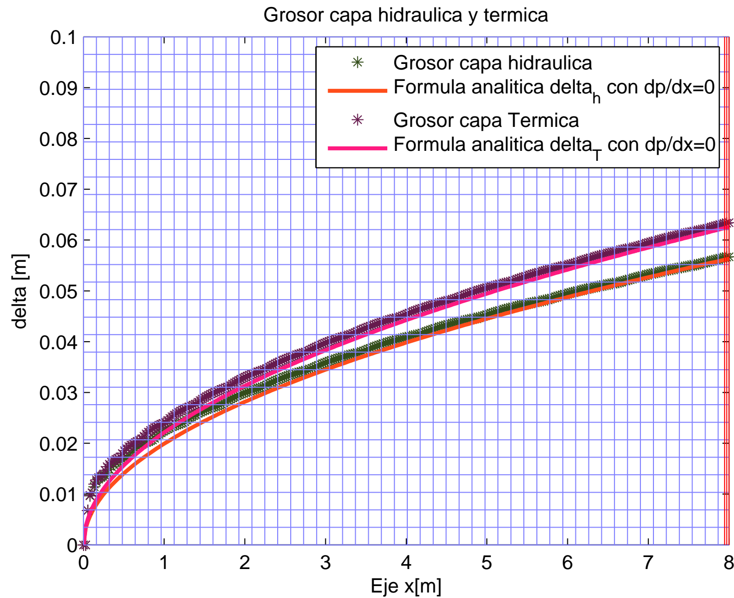
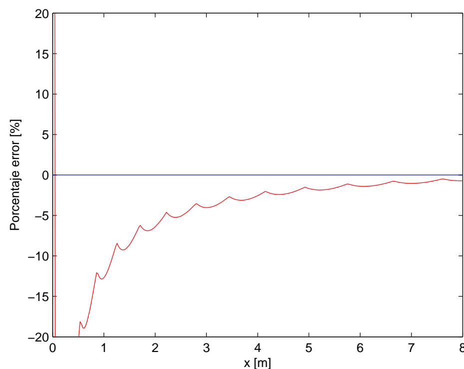
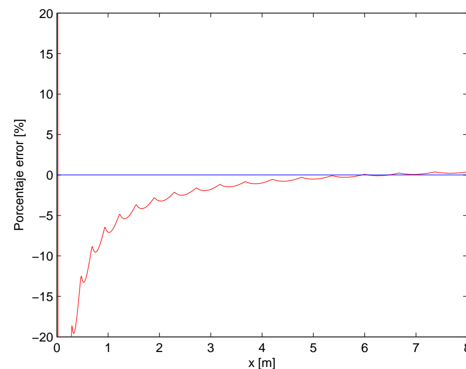


Figura 59: Porcentaje de error de la capa límite hidráulica.



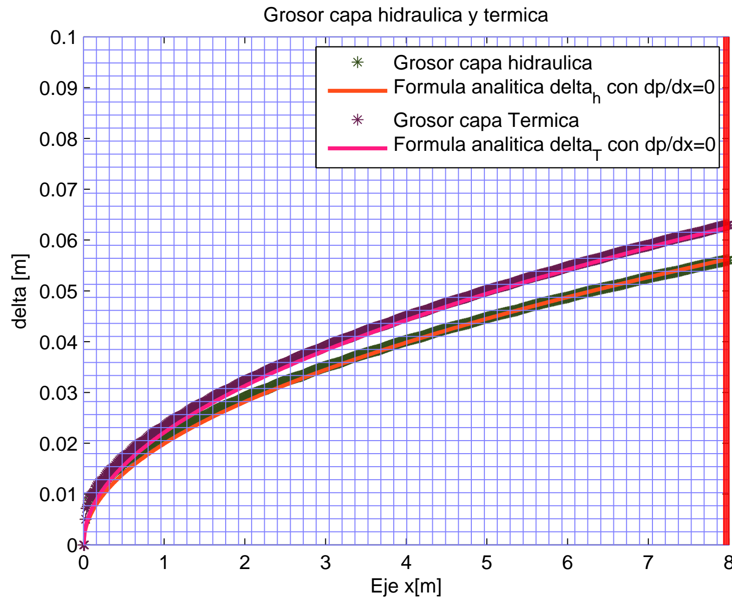
(a) 300x30 nodos



(b) 600x40 nodos

En el caso presentado en la sección 6.7.1 se utilizó 10800 volúmenes de control para lograr un porcentaje de error de 5 % en $x = 1.39[m]$, en el método explícito se necesitan 24000 nodos para lograr un nivel similar de precisión. El método explícito es fácil de implementar, rápido en encontrar la solución y no requiere grandes cantidades de memoria para el procesamiento de los datos, sin embargo, es inestable, está limitado en la geometría a la cual se puede aplicar y, solo puede ser utilizado para resolver las ecuaciones simplificadas de la capa límite.

Figura 58: Espesor de la capa límite térmica e hidráulica para 600x40 nodos.



6.7.3 Simulación de las ecuaciones de Falkner-Skan para el punto de estancamiento.

La simulación del punto de estancamiento por el modelo de Falkner-Skan se realiza con $m = 1$, para este valor es posible también comparar los resultados con el modelo de Howarth. Por facilidad se utilizará la palabra **Falkner1** para referirse a este caso.

En este caso, el gradiente de la velocidad u en dirección x a la salida no es cero ni el gradiente de presión en dirección x. El gradiente de la velocidad u se puede hallar directamente y adicionalmente se puede hallar el gradiente de presión. Para calcular su valor se utiliza la (Ec 1.22).

$$U_e[m/s] = C\,x^m \qquad \Rightarrow \qquad C = 1[1/s] \quad m = 1[-]$$

$$\frac{\partial u}{\partial x} = \frac{\partial x}{\partial x} = 1$$

$$\rho u \frac{\partial u}{\partial x} = 1\;\rho\;x = -\frac{dp}{dx}$$

$$p = -\frac{\rho\,x^2}{2}[Pa]$$

$$\left(\frac{dp}{dx}\right)|_{x=2[m]} = -2.344 \;\; [Pa/m]$$

Se simulará una región de 2 metros en dirección x y 0.015 metros en dirección y.

También es necesario calcular la variación de la velocidad v en la frontera norte, a partir de la ecuación de continuidad:

$$\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}=0 \qquad \rightarrow \qquad \frac{\partial v}{\partial y}=-\frac{\partial u}{\partial x}=-1$$

El archivo de definición de la geometría del problema contiene la siguiente información:

```
> nsd = 2;
> subdominios = 1;
> no_de_superels = 1;
>nbind = 4
>bname VelUNeumann=-1.0,VelVNeumann=0,PNeumann=2.344,
VcorNeumann=0,TempNeumann=0
VelUDirichlet=1x^1,VelVNeumann=1.0,PNeumann=0,
VcorNeumann=0,TempDirichlet=1
VelUDirichlet=0,VelVNeumann=0,PNeumann=0,
VcorNeumann=0,TempDirichlet=0.0
VelUDirichlet=0,VelVDirichlet=0,PNeumann=0,
VcorDirichlet=0,TempDirichlet=0
>SupEl;
>subdominio_no = 1;
>tipoelemento = ElmB4n2D;
>fronteras = [1(1)] [2 (2)] [3 (3)] [4(4)];
>nodos = [1(0 0)]+[2(2.0 0)]+[3(0 0.015)]+[4(2.0 0.015)];
>lados=;
```

Se realizan dos simulaciones para el caso Falkner1 mostrando la influencia del número de volúmenes de control, las particiones usadas son las siguientes:

```
>nsd = 2;
>no_de_superels = 1;

>SE;>d=2;>e=ElmB4n2D;>div=[60,30];>grad=[1.0,1.0];

>nsd = 2;
>no_de_superels = 1;

>SE;>d=2;>e=ElmB4n2D;>div=[180,48];>grad=[1.0,1.0];
```

Los resultados se presentan en la figura 60 y en la tabla 24, adicionalmente, se muestran las líneas de flujo en la figura 61.

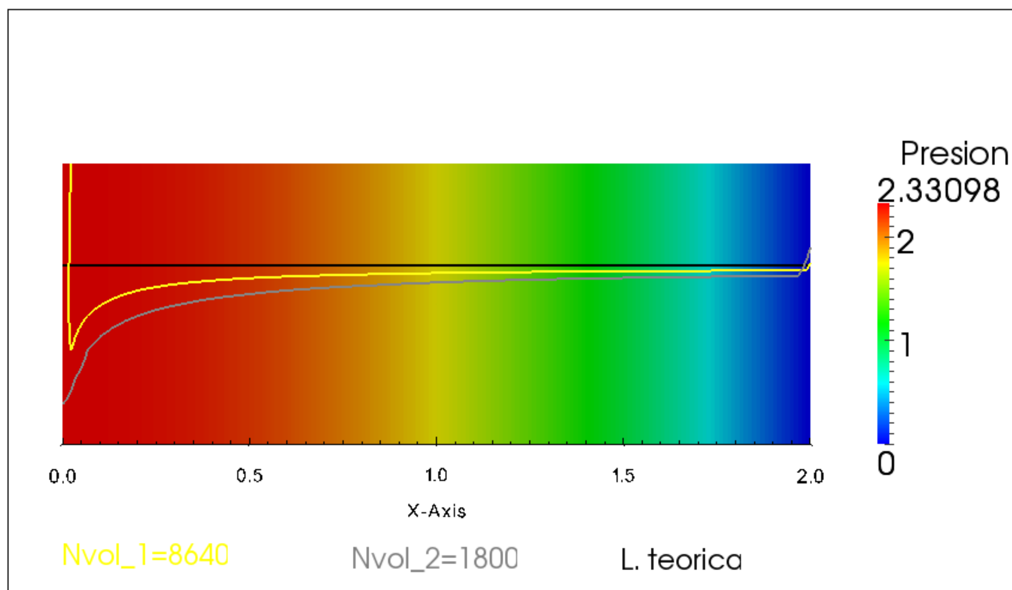
Para comparar los resultados con un modelo teórico se puede interpolar los datos de las

gráficas de la ecuación de Falkner-Skan (figura 4) ó tomar los datos del modelo de Howarth[14][15]. El modelo de Howarth asume que la velocidad del fluido fuera de la capa límite tiene un perfil igual a $u = a\,x$, en este caso, $a = 1$. Conociendo el valor de a se calcula el valor de η y se iguala a 2.4 para obtener el espesor de la capa límite hidráulica.

$$\eta_{99} = \sqrt{\frac{a}{\nu}}\,\delta_h = \sqrt{\frac{1}{1.172/1.858e-5}}\delta_h = 2.4 \quad \rightarrow \quad \delta_h = 0.00956[m] \qquad \text{(Ec 6.34)}$$

Para el caso Falkner1 el espesor hidráulico es una constante. A partir de esta constante se calculó la altura a simular que finalmente fue $0.015[m]$.

Figura 60: Espesor de la capa límite hidráulica del caso Falkner1.



[14] Es posible tomar los datos del modelo de Howarth porque el exponente usado para el modelo de Falkner-Skan es igual a 1.0.

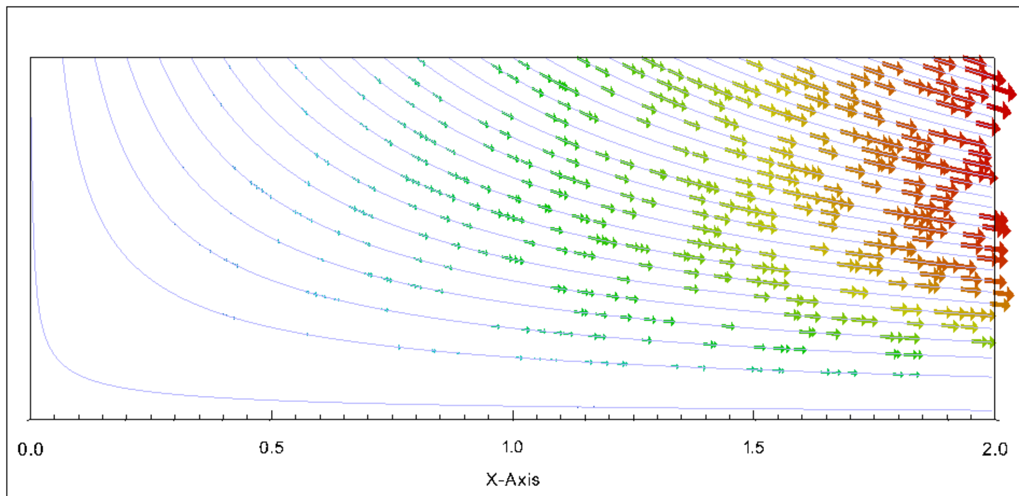
[15] SCHLICHTING, H. Teoría de la capa límite, 3 ed. New York: McGraw Hill, 1968. p.102.

Tabla 24: Comparación de resultados para el espesor de la capa hidráulica del caso Falkner1.

	δ_{hid}			
x $[m]$	Valor Teórico $[m]$	Valor Calculado $[m]$	Error absoluto $[m]$	Error porcentual $[\%]$
0,2	0,00956	0,00823	0,00133	14
0,4	0,00956	0,00876	0,00079	8
0,6	0,00956	0,00898	0,00057	6
0,8	0,00956	0,00909	0,00046	5
1	0,00956	0,00917	0,00039	4
1,2	0,00956	0,00922	0,00034	4
1,4	0,00956	0,00925	0,00030	3
1,6	0,00956	0,00928	0,00028	3
1,8	0,00956	0,00930	0,00026	3

A partir de la figura 60 se aprecia la existencia de una perturbación por las condiciones de frontera este y oeste cuya importancia disminuye al aumentar el número de volúmenes de control.

Figura 61: Líneas de flujo para el caso Falkner1.



6.8 ANÁLISIS DE RESULTADOS

Para cualquier análisis en la CFD se debe verificar que se cumplan las siguientes características[16]:

1. Consistencia.
2. Estabilidad.
3. Convergencia.
4. Precisión.
5. Conservación.
6. Solución acotada.

En los métodos SIMPLE y NOSIMPLE se asume la velocidad del fluido como un valor conocido para resolver la ecuación de cantidad de movimiento en x, este valor puede diferir significativamente del valor que se halla en esa ecuación de la velocidad para las primeras iteraciones exteriores, de esta forma, se cumple con la **consistencia** de las ecuaciones cuando se tenga el campo de la velocidad correcto.

Los dos métodos presentados son condicionalmente **estables**. Se define un parámetro de subrelajación para la presión en el método SIMPLE $\alpha_P = 0.1$ y en el método NO SIMPLE un parámetro de sobrerelajación para la velocidad $\alpha_V = 1.2$ para que el proceso sea estable. El valor óptimo de estos parámetros depende, entre otros factores, del avance en el tiempo y este último depende del tamaño de los volúmenes de control. Para el flujo sobre una superficie plana los métodos resultan inestables si se utiliza un avance de tiempo relativamente grande y se define la velocidad inicial del fluido como cero[17].

La **convergencia** del método SIMPLE es baja, para las tres simulaciones presentadas[18] la norma del residuo másico alcanzó valores menores a $\boldsymbol{O}(1e-7)$ para más de 300 iteraciones exteriores y la actualización del valor de la velocidad u es menor a $\boldsymbol{O}(1e-7)$ para más de 500 iteraciones.

Por otro lado, la velocidad de **convergencia** del método NOSIMPLE es más alta, para algunos de los casos presentados la norma del residuo másico fue del $\boldsymbol{O}(1e-8)$ en solo 75 iteraciones exteriores.

[16] Este tema se trató en el capítulo 2

[17] Este problema es el resultado del ingreso de una gran cantidad de fluido en la frontera tipo Dirichlet del oeste y una mínima salida de fluido por la velocidad inicial definida como cero.

[18] NSCL: 5400, 27000, 120000 volúmenes de control.

En lo que respecta a la **precisión** del método NOSIMPLE, se obtuvieron resultados satisfactorios comparado con la teoría de Blasius para la mayoría de los datos. Se presentan algunas diferencias en la frontera este y oeste; en la frontera de entrada del fluido se presentan errores por la resolución de la malla usada y, en la zona cercana a la frontera `este` debido a la simplificación realizada para definir las condiciones de frontera las cuales son válidas fuera de la capa límite.

Los resultados encontrados de la simulación de las ecuaciones completas de Navier Stokes y su solución por el método SIMPLE difieren ligeramente de los valores predecidos por la teoría de Blasius basada en las simplificaciones de la capa límite. Se realizaron tres simulaciones con diferentes mallados en las que la variación de los resultados era mínima con respecto al error existente con el valor teórico. La principal causa de error se atribuye a la ecuación de cantidad de movimiento en y; en esta ecuación se define la velocidad de entrada del fluido como $u = 1$ y $v = 0$ correspondientes al problema del flujo en la capa límite; la difusión y la convección de la propiedad $v = 0$ hacen que la velocidad del fluido en y sea menor a la velocidad esperada por lo que el valor de la velocidad u aumenta para cumplir el balance de masa. En las simulaciones realizadas, el valor de la presión para Reynolds bajos aumenta como es descrito por algunos autores que han investigado en el tema. Por último, las ecuaciones simplificadas de Blasius no explican cuál es la fuerza responsable de generar la velocidad v, ésta es la consecuencia de realizar el balance de masa, los resultados encontrados muestran que la fuerza generada por la presión contribuyen a la cantidad de movimiento en dirección y para aumentar la velocidad.

La propiedad de la **conservación** se cumple al satisfacer el criterio de convergencia del máximo valor permitido para el residuo másico.

Los resultados encontrados para el método NOSIMPLE son **acotados** cuando la solución converge. Cabe agregar que la temperatura calculada para el presente trabajo es en realidad un cambio en la temperatura con unidad de Kelvin, no se consideran casos criogénicos; dependiendo del monitor de convergencia usado para resolver los sistemas de ecuaciones lineales podría encontrarse soluciones erróneas si los resultados encontrados van a estar entre, por ejemplo, $1000[K]$ y $1001[K]$, además de desperdiciar algunas cifras significativas.

Como resultado de las pruebas realizadas y de la comparación con la teoría de Blasius, se encontró un mejor comportamiento para el método NOSIMPLE que corrige directamente la velocidad v comparado con el método SIMPLE que resuelve las ecuaciones completas de Navier-Stokes. El método NOSIMPLE presentó una mayor velocidad de convergencia pero está limitado su uso a la simulación de la capa límite y no tiene en cuenta los gradientes de presión inducidos por el flujo.

7. CONCLUSIONES

1. La solución de las ecuaciones de la mecánica de fluidos se puede llevar a cabo al tratar las ecuaciones como un acoplamiento de las ecuaciones de difusión con las ecuaciones de convección-difusión.

2. La simulación en la capa límite sobre una placa plana es un problema de flujo viscoso con una geometría sencilla que requiere, para números de Reynolds bajos, una resolución relativamente mayor para lograr captar los detalles del comportamiento del fluido en esta zona.

3. Las ecuaciones de la capa límite de carácter parabólico pueden ser resueltas como ecuaciones elípticas (teniendo en cuenta el término $\partial^2 u/\partial x^2$) sin incorporar un error significativo para números de Reynolds moderados.

4. El método SIMPLE es capaz de resolver las ecuaciones completas de Navier Stokes aplicadas al flujo en la capa límite cuyos resultados pueden diferir ligeramente de los resultados obtenidos a partir de las simplificaciones realizadas por Blasius. Este método presenta una tasa de convergencia más lenta comparada con el método NOSIMPLE.

5. El uso de un valor de referencia para la ecuación de corrección de presión en el método SIMPLE introduce un error en el residuo másico que no afecta significativamente los resultados.

6. La solución de las ecuaciones de Navier-Stokes aplicadas a la capa límite demuestran que el gradiente de presión en dirección y (normal a la placa) es la fuerza encargada de generar la componente vertical de la velocidad.

7. El método NOSIMPLE ha probado ser un método iterativo efectivo para resolver las ecuaciones simplificadas de la capa límite.

8. Aunque se usen esquemas de discretización incondicionalmente estables (esquema híbrido para el término convectivo y esquema implícito para el análisis transitorio) en el método SIMPLE o NOSIMPLE, el proceso es condicionalmente estable de acuerdo al valor del parámetro de subrelajación y el avance en el tiempo seleccionado.

8. RECOMENDACIONES Y OBSERVACIONES

1. El propósito del presente trabajo de grado fue simular el flujo de un fluido a bajas velocidades sobre una superficie plana en régimen laminar con propiedades constantes por medio del método de volúmenes finitos. Se podrían desarrollar otros trabajos de grado que simulen el comportamiento de un flujo turbulento para geometrías más complejas como cilindros o perfiles de ala de avión e incluso, considerar las variaciones de las propiedades por la temperatura a altas velocidades.

2. El método NOSIMPLE fue puesto a prueba con el flujo sobre una placa plana, se pueden desarrollar otras pruebas para verificar su capacidad de resolver las ecuaciones de la capa límite bajo otras condiciones, por ejemplo, simular otras geometrías, superficies porosas, perturbaciones de la superficie, etc.

3. Las librerías creadas pueden ser optimizadas en el uso de memoria y el tiempo de procesamiento. Se modificó el proceso para hacer uso de las matrices dispersas que junto con el precondicionador RILU redujo drásticamente el tiempo de solución de las ecuaciones. Aun queda por optimizar, entre otros, el preprocesamiento de las condiciones de frontera que actualmente se repite para cada volumen de control en la frontera.

4. La solución de problemas netamente convectivos ($\Gamma = 0$) se puede llevar a cabo por `ConveccionTransitoria` con el método UDS; el uso del esquema de diferencias centradas conduciría a soluciones divergentes por la evaluación del Peclet infinito.

5. Aunque gran parte del código desarrollado fue pensado para cualquier tipo de malla, incluyendo la simulación en tres dimensiones, se deben realizar algunas modificaciones que permiten obtener resultados precisos en mallados no estructurados.

6. Se pueden cometer errores en la implementación de las condiciones de frontera los cuales son más fácilmente detectados al evaluar casos en donde el valor de la condición de frontera es diferente de cero, por ejemplo, simulando el problema modificado para el flujo convectivo utilizado en el presente trabajo en vez del problema original de Ferziger.

7. Los resultados de la simulación de la difusión transitoria pueden ser verificados con las soluciones teóricas existentes para una pared plana de propiedades homogéneas e inicialmente isotérmica.

BIBLIOGRAFÍA

[1] BAKKER, A. Lecture 5- solution methods: Applied computational fluid dynamics, Fluent: 2006 [citado 13 Junio 2012], 45 p. Disponible en : http://www.bakker.org/dartmouth06/engs150/05-solv.pdf.

[2] BALTAZAR, J. M.Y EÇA, L. R. Generación de mallas estructuradas en superficie. [online], Información tecnológica [Providencia, Chile]: 2006, vol.17, n.3 [citado 12 Junio 2012], pp 107-116. Disponible en internet: http://www.scielo.cl/scielo.php?pid=S0718-07642006000300016&script=sci_arttext. ISSN 0718-0764.

[3] BELEÑO, A. A. *Seminario de investigación en mecánica de fluidos computacional*. Trabajo de grado Ingeniero Mecánico. Bucaramanga. Universidad Industrial de Santander. Escuela de Ingeniería Mecánica. Facultad de Ingenierías Fisico-Mecánicas, 2009. 361 p.

[4] BLAZEK, J. *Computational fluid dynamics: principles and applications*. Oxford: Elsevier, 2011. p. 1-128.

[5] CEBECI, T. Y COUSTEIX, J. *Modeling and computation of boundary-layer flows*. California: Horizons Publishing Inc., 2005. p. 211-241.

[6] ÇENGEL, Y.A. Y BOLES, M. *Termodinámica: un enfoque práctico*, 6 ed. México: McGraw Hill, 2009. p 70-71, 222-225.

[7] ÇENGEL, Y. A. *Transferencia de calor y masa: un enfoque práctico*, 3 ed. México: McGraw Hill, 2007. p.355-394.

[8] CFD ONLINE. History of cfd. [online], CFD online: 2008 [citado 16 Abril 2012]. Disponible en internet: http://www.cfd-online.com/Wiki/History_of_CFD.

[9] CHUNG, T. J. *Computational fluid dynamics*. Cambridge University, 2002. p. 29-42.

[10] DAILY, J. W. Y HARLEMAN, D. R. *Dinámica de los fluidos: Con aplicaciones a la ingeniería*. México: F. Trillas, 1969. p.15-16.

[11] DENNIS, S. Y DUNWOODY, J. The steady flow of a viscous fluid past a flat plate. *En: Journal of Fluids Mechanics 1966. vol. 24, no. 3*, 18 p.

[12] DENNIS, S. Y SMITH, N. Forced convection from a heated flat plate. *En: Journal of Fluids Mechanics 1966. vol. 24, no. 3*, 10 p.

[13] DÍEZ, P. F. Vii. teoría elemental de la capa límite bidimensional. [online], Red Sauce [Cantabria, España]: 2009. [citado 5 Agosto 2012], 17 p. Disponible en internet: http://libros.redsauce.net/MecanicaFluidos/PDFs/07MecFluidos.pdf.

[14] FERZIGER, J.H. Y PERIĆ, M. *Computational Methods for Fluid Dynamics*, 3 ed. Berlin: Springer, 2002. 423 p.

[15] FUENTES, D. Solución de las ecuaciones de NS en geometrías complejas mediante el método de volúmenes finitos. *Mayo, 2012*, 12 p.

[16] GONZALEZ, G. Factorización de cholesky. [online], México: 2003 [citado 7 Abril 2012]. Disponible en: http://www.matematicas.unam.mx/gfgf/pa20072/data/lecturas/sistemas/Cholesky.html.

[17] KARABELAS, S. J. A novel similarity transformation for the boundary layer equations: Solution of boundary layer flows subjected to exponential outer velocity profiles. *En: Journal of Applied Mechanics Julio, 2011. vol. 78*, 5 p.

[18] KIRICKOV, M. Sample code for solving lid-driven cavity test (Re=1000) - fortran 90. [online], CFD online: 2010 [citado 1 Junio 2012]. Disponible en internet: http://www.cfd-online.com/Wiki/Sample_code_for_solving_Lid-Driven_cavity_test_(Re%3D1000)_-_Fortran_90.

[19] LOPEZ, D. Descomposición lu. [online], Antioquia [citado 7 Abril 2012]. Disponible en: http://aprendeenlinea.udea.edu.co/lms/moodle/mod/ resource/view.php?inpopup=true &id=24488.

[20] MATEWS, J. *Métodos numéricos*. Madrid: Prentice Hall, 2007. p. 155-157.

[21] MILLS, A. *Transferencia de calor*. Madrid: McGraw Hill, 1999. p. 424-425.

[22] NPTEL. Boundary layer equations. [online], National Programme on Technology Enhanced Learning [citado 12 Junio 2012]. Disponible en internet: http://nptel.iitm.ac.in/courses/Webcourse-contents/IIT-KANPUR/FLUID-MECHANICS/lecture-28/28-2_boundary_eq.htm.

[23] PATANKAR., S. V. *Numerical Heat Transfer and Fluid Flow*. Taylor & Francis, 1980. p. 79-111.

[24] PEÑALOZA, E. R. *Fundamentos de programación C/C++*, 4 ed. México: Alfaomega, 2004. p. 216-222, p. 343-474.

[25] SCHETZ, J. A. *Boundary Layer Analysis*. Virginia: American Institute of Aeronautics and Astronautics, 2010. 520 p.

[26] SCHLICHTING, H. *Grenzschicht Theorie [Teoría de la capa límite]*, traducido por Francisco Morán Samaniego, 5 ed. Bilbao: McGraw Hill, 1968. 812 p.

[27] SOUALEM, N. Jacobi method. [online], Math Linux: 2006. [citado 15 Julio 2012]. Disponible en: http://www.math-linux.com/spip.php?article49.

[28] STEWART, J. *Cálculo de varias variables: trascendentes tempranas*, 6 ed. México: Cengage Learning, 2008. p. 1055-1103.

[29] TU, J.; HENG, G. Y. L. C. *Computational fluid dynamics : a practical approach*. Burlington : Elsevier, 2008. p. 126-223.

[30] WHITE, F. M. *Fluid mechanics*. McGraw-Hill series in mechanical engineering. WCB/McGraw-Hill, 1999. p. 3-46, 434-451.

[31] WHITE, F. M. *Viscous fluid flow*, 3 ed. New York: McGraw Hill, 2006. p. 218-334.

[32] WIKIPEDIA. Mecánica de fluidos. [online], 14 de Mayo WIKIPEDIA 2012, [citado 12 Junio 2012]. http://es.wikipedia.org/wiki/Mecánica_de_fluidos.

ANEXO A. CLASES EN C++

Una clase es una construcción que se utiliza como un modelo (o plantilla) para crear objetos de ese tipo. Un objeto se define como la unidad que en tiempo de ejecución realiza las tareas de un programa. Un objeto es una instancia de una clase.

Un ejemplo de una clase creada en Visual C++ es:

```
#pragma once
//Archivo Fluido_base.h

class Fluido_base
{

public:
Fluido_base(void);

~Fluido_base(void);

};

#include "Fluido_base.h"
//Archivo Fluido_base.cpp

Fluido_base::Fluido_base(void)
{
}
Fluido_base::~Fluido_base(void)
{
}
```

C++ define algunas características para los miembros de las clases que pueden ser implementadas para explotar las funciones del lenguaje. Se resaltan algunas caracteristicas a continuación.

SOBRECARGA DE FUNCIONES

La sobrecarga es la capacidad de tener dos o más funciones diferentes con un mismo nombre. Las funciones sobrecargadas se diferencian por los parámetros de entrada. Se presenta un ejemplo de sobrecarga de la función `Conductividad` en el siguiente código:

```
...
class Fluido_base
{
...
public:
...
real Conductividad ( real temperatura);
real Conductividad ( void );
};
```

FUNCIONES GENÉRICAS Y CLASES GENÉRICAS

Las funciones genéricas y las clases genéricas permiten definir funciones y clases para un tipo de parámetro cualquiera sin necesidad de escribir una función para cada parámetro deseado. En el siguiente ejemplo se muestra la función genérica `suma`.

```
...
template <class plantilla>
real suma (plantilla a, plantilla b) {return a+b;}
```

La anterior función puede ser usada en los siguientes tres casos:

```
cout<<suma(1.25,3.0);
cout<<suma(1,3);
cout<<suma('0','1');
//cout<<suma(1.0,3);
//cout<<suma("i","j");
```

Por la definición de la función, el compilador no creará una función para `a` de tipo `real` y `b` de tipo `int`. Por otro lado, se debe mencionar que el tipo de dato usado en la plantilla debe tener definidas las operaciones realizadas en la función, es por esto que no es posible ejecutar la función `suma("i","j")`.

CONSTRUCTOR COPIA

El constructor copia es el método encargado para realizar la copia de un objeto en otro; por defecto, existe un constructor que se encarga de asignar exactamente los mismos valores al objeto destino pero puede ser necesario definir un constructor propio, la forma más común de declarar el constructor copia es la siguiente:

```
...
class Fluido_base
{
...
public:
...
Fluido_base ( const Fluido_base &copia);//copyconstructor
bool SonIguales (Fluido_base copia);
};
```

Si se crea un objeto `Fluido_base fluido1` y se da al método `SonIguales` una instancia de `Fluido_base` como parámetro de entrada, se realiza una llamada al constructor copia en las siguientes tres instrucciones:

```
...
Fluido_base fluido2 (fluido1);
Fluido_base fluido3 = fluido1;
fluido3.SonIguales(fluido1);
...
```

Si la función `SonIguales` tomara como argumento `&Fluido_base copia`, es decir la dirección de un objeto `Fluido_base`, no se haría internamente la llamada al constructor copia.

DEFINICIÓN DE OPERADORES

En C++ es posible definir operaciones como +, -, [], <, ... en las clases como se muestra a continuación:

```
...
class Fluido_base
{

private:
...
protected:
...
...
public:
...
Fluido_base operator+ (Fluido_base segundo);//a+segundo
Fluido_base operator+ (int entero);//
Fluido_base operator+ (real doble);//
Fluido_base operator- (Fluido_base segundo);//a-segundo
Fluido_base operator= (Fluido_base segundo);//a=segundo
Fluido_base operator<< (Fluido_base segundo);//
Fluido_base operator>> (Fluido_base segundo);//
Fluido_base operator++ (void); //llamada al ++fluido1;
Fluido_base operator++ (int no_usado); //llamada al fluido1++;
};
```

Los operadores también pueden ser sobrecargados y se recomienda que la definición del operador sea similar a la definición comúnmente usada. Los operadores que no se pueden sobrecargar en C++ son `(.) (:) (.*) (?)`.

ACCESIBILIDAD DE LOS MIEMBROS DE UNA CLASE

No todos las variables y los métodos creados en una clase son accesibles por cualquier parte de un programa, en C++ existen tres tipos de accesibilidad:

1. Private: los miembros declarados como `private` solo son accesibles por el mismo objeto. Por defecto, los elementos de una clase son del tipo `private`.

2. Public: cualquier parte del programa que tenga acceso al objeto puede acceder a los miembros declarados como `public`.

3. Protected: los miembros declarados como `protected` son accesibles por el objeto al cual pertenecen y a diferencia de los miembros `private`, son accesibles por las clases heredadas.

Funciones de acceso

En algunos casos puede ser necesario obtener o modificar el valor de un dato de un objeto declarado como `protected` o `private` en el ámbito de bloque (el bloque de código en el cual se declaró el objeto que contiene el dato) sin necesidad de cambiar la declaración del dato. Para realizar este trabajo se debe crear una función de acceso al dato como se muestra en el siguiente código:

```
//Archivo Fluido_base.h
...
class Fluido_base
{
private:
...
protected:
...
bool prop_constantes;
real temp_min;
real temp_max;
real temp_muestra;

public:
...
void SetPropiedadesConstantes
(bool tipo);
bool GetPropiedadesConstantes
(void);
};
```

```
//Archivo Fluido_base.cpp
...
void Fluido_base::
SetPropiedadesConstantes (bool tipo)
{
if (tipo)
{
prop_constantes=tipo;
temp_min=temp_max=temp_muestra;
}
else
{prop_constantes=tipo;}

}

bool Fluido_base::
GetPropiedadesConstantes (void)
{
return prop_constantes;
}
```

En el código anterior se declaran 4 variables y dos métodos; prop_constantes indica si el fluido se tratará con propiedades constantes (true) o propiedades dependientes de la temperatura, false, temp_min almacena la mínima temperatura a la cual se han obtenido las propiedades y temp_max almacena la máxima temperatura, temp_muestra es una temperatura a la cual pueden aproximarse las propiedades. Para conocer si el fluido se tratará con propiedades constantes se llama a la función GetPropiedadesConstantes (void), por otro lado, se usa SetPropiedadesConstantes (bool tipo) para modificar el valor de prop_constantes. En este caso no es suficiente con modificar prop_constantes, se debe cambiar el valor de la temperatura máxima y la temperatura mínima al valor de la temperatura de muestra, por esta razón no es conveniente declarar prop_constantes como public.

HERENCIA POR EXTENSIÓN: CLASE BASE Y CLASE HEREDADA

Se pueden crear nuevas clases a partir de clases existentes utilizando la herencia, la clase heredada se llama clase derivada y la clase padre se llama clase base. La herencia por extensión consiste en agregar miembros adicionales a los existentes en la clase base. Se muestra un ejemplo de la herencia por extensión en la siguiente definición de la clase Fluido_Agua.

```
...
class Fluido_base
{
...
bool base;
protected:
...
bool prop_constantes;
public:
real Conductividad
   ( real temperatura);
};
```

```
...

class Fluido_Agua:
     private Fluido_base
{
real k;
protected:
...
public:
...
real Prandtl ( real temperatura);
};
```

Al usar la herencia por extensión se debe definir el tipo de acceso a la clase base. Para el ejemplo presentado, la función conductividad(...) que era public en Fluido_base es ahora private para Fluido_Agua, es decir, la clase Fluido_Agua puede acceder a la función conductividad(...) pero el bloque de código en el cual se creó la instancia de Fluido_Agua no puede acceder a dicha función. En general, se modifica el tipo de acceso de los elementos de la clase base si la declaración de ésta es más restrictiva. En el mismo ejemplo anterior, la clase Fluidos_Agua no puede acceder directamente a la variable base declarada como private en la clase Fluidos_base.

Funciones virtuales

Una función virtual es usada para definir en las clases derivadas nuevas funciones con el mismo nombre y los mismos argumentos. Para declarar una función como virtual se agrega la palabra `virtual` a la función en la clase base.

```
...
class Fluido_base
{
...
public:
...
virtual real C_v()
  {return 0.0;}

};
```

```
...
class Fluido_Agua:
    public Fluido_base
{
...
public:
...
real C_v ()
   {return 4120;}
real acceso() {return 1.0;}
};
```

La definición de funciones virtuales es especialmente útil cuando se usa el polimorfismo.

Polimorfismo

El polimorfismo es la posibilidad de llamar a varias funciones mediante una misma sentencia. La forma de utilizar el polimorfismo es declarar un puntero de una clase base y apuntar a una clase derivada.

```
...
Fluido_base *Pbase1 = new Fluido_base;
Fluido_base *Pbase2 = new Fluido_Agua;
cout<<"\nPuntero a clase base"<< Pbase1->C_v();
cout<<"\nPuntero a clase derivada"<< Pbase2->C_v();
...
```

En el anterior fragmento de código, se devuelve el valor 0 como resultado de la llamada a `Pbase1->C_v()` porque se está apuntando a un objeto de tipo `Fluido_base`, por otro lado, al ejecutar la instrucción `Pbase2->C_v()` se devuelve el valor 4120 porque se está apuntando a un objeto de tipo `Fluido_Agua`. Para el ejemplo presentado, no es posible llamar a `Pbase2->acceso()` porque la clase base no conoce la existencia de ese método, por esta razón es necesario declarar las funciones virtuales en la clase base.

Clase abstracta y función virtual pura

Una clase abstracta es una clase que ha sido pensada para servir de clase base a otras clases y que no permite crear objetos de esta clase. Para que una clase sea abstracta debe tener por lo menos una función virtual pura. Una función virtual pura es una función a la cual no se define un proceso y se iguala a 0 sin importar el tipo de valor devuelto. Debido al polimorfismo, un puntero a una clase abstracta permite una llamada en común para todas las clases derivadas.

Si se desea crear un objeto de una clase derivada que tiene como clase base una clase abstracta es necesario definir todos los métodos que hayan sido declarados como funciones virtuales puras en la clase base. Si no se realiza la definición de la función en la clase derivada, ésta también sera una clase abstracta.

En el siguiente fragmento de código se muestra una función virtual pura y una clase abstracta.

```
...
//Clase abstracta
class Fluido_base
{
...
public:
...
virtual real C_v()=0;
//Funcion virtual pura
};
```

```
...
class Fluido_Agua:
    public Fluido_base
{
...
public:
...
real C_v ( )
   {return 4120;}
};
```

ANEXO B. SOLUCIÓN INTUITIVA DE LA ECUACIÓN DE DIFUSIÓN: ENCABEZADO DE LA CLASE

```
#pragma once
// Encabezado de la clase diftemp: metodo intuitivo de c\'alculo de
//la ecuaci\'on de difusi\'on
// Archivo "diftemp.h"
#include <real.h>
#include <AdmonEcLin.h>
#include <Matriz.h>
#include <Vector.h>

class diftemp
{
private:
  real longx,longy ;
  //longitud x e y de la placa
  int  nx,   ny    ;
  // numero de nodos en la direccion x e y
  real dx,   dy    ;
  //ancho y alto de un volumen de control
  real k;
  //conductividad del material
  Matriz<real>           matriz_a;
  //matriz que contiene los coeficientes de las ecuaciones
  Vector<real>       b;
  //vector que contienen las terminos independientes
  Vector <real>      x;
  //vector solucion
  Matriz<real>           temp;
  //almacenar el valor de la temperatura en una matriz
  prm_Matriz <real> pm;
  //parametros de matriz, no es necesario en este caso
  //pero se recomienda,
  Cadena x0tipo,y0tipo,xfinaltipo,yfinaltipo;
  //seleccion del tipo de condicion de frontera
  Puntero<AdmonEcLin> admlin;

  //////////banderas/////////////////////
```

```
  bool  calculado;
  bool  leido;

public:
  diftemp() { leido= false; calculado=false;}
  ~diftemp() {}
  void Leer (real ancho, real alto, real divx,real divy);
  void Leer (void)  {Leer(1.0,1.0,20,20);}
  real condx0 (real y);//funcion para la condicion de frontera x0
  real condy0 (real x);//funcion para la condicion de frontera y0
  real condxf (real y);//funcion para la condicion de frontera
   yfinal
  real condyf (real x);//funcion para la condicion de frontera
   xfinal
  real q (real x, real y );//funcion para la generacion de calor
  //asignar los coeficientes de las ecuaciones
  void calcular();
  //resolver el sistema de ecuaciones
  bool resolver();
  //guardar los resultados y presentarlos en matlab
  void exportar(Cadena archivo);

};//////fin de la declaraci\'on de la clase diftemp////////
```

ANEXO C. SOLUCIÓN INTUITIVA DE LA ECUACIÓN DE DIFUSIÓN: ARCHIVO DE CÓDIGO FUENTE

```
#include  "diftemp.h"
//metodo intuitivo de c\'alculo de la ecuaci\'on de difusi\'on
//Archivo "diftemp.cpp"


void diftemp::Leer (real ancho, real alto, real divx,real divy)
{
  longx = ancho ;   longy = alto ;
  nx    = divx  ;   ny    = divy ;
  dx    =longx/nx ;   dy    = longy/ny;
  //////dimensionamiento de la matriz y el vector/////////
  matriz_a.chaSize(nx*ny);    matriz_a.llenar(0.0);
  //es importante la inicializacion en 0.0
  b.chaSize(nx*ny);           b.llenar(0.0);
  x.chaSize(nx*ny);           x.llenar(170.0);
  temp.chaSize(nx,ny);         temp.llenar(0.0);
  /////////seleccion del tipo de condicion de frontera////
  x0tipo = "Dirichlet";  xfinaltipo = "Dirichlet";
  y0tipo = "Neuman";     yfinaltipo = "Neum";
  ////////////////propiedades de material////////////////
  k = 50;
  ////////////////lectura metodo solucion//////////////
  Is is("ARCH=Solve_datos.txt");
  pm.Leer ( is );
  admlin.vincular(new AdmonEcLin(EXTERNAL_STORAGE));
  admlin().Leer(is);
  leido=true;//bandera
  calculado=false; // al realizar esta asignaci\'on se permite que
   el objeto creado pueda calcular varios problemas
}

real diftemp::condx0(real y) {  return 150.0; }
real diftemp::condxf(real y) {  return 100.0; }
real diftemp::condy0(real x) {  return 0.0;   }
real diftemp::condyf(real x) {  return 200.0; }
real diftemp::q(real x,real y) { return 5000.0; }
```

```
void diftemp::calcular()        //public
{
  if (leido==false)
  { Leer( ); std_o<<"Advertencia: posible error en la ejecucion"; }
  if (calculado== true)
  {std_o<<"Ya se habia calculado los coeficientes\n"<<
    "POSIBLES ERRORES EN LOS RESULTADOS\n";}

  int  i, j;
  real dxsy = dx/dy;
  //Aunque la matriz contiene la informacion en solo 5 diagonales
   se tratar'a como una matriz completa se asignara primero los
   coeficientes correspondientes a la direccion e y w de esta
   manera se evita crear muchos casos especificos (debido a la
   combinacion)
  //de condiciones de frontera recordar que la ecuacion  de energia
    es una sola ecuacion NO ES UNA ECUACION VECTORIAL  asignacion
   del valor al coeficiente b en lo que respecta a la generacion de
    calor
  for( i=1 ; i<=nx ; i++ )
    for (j=1 ; j<=ny ; j++)
      b( (j-1)*nx + i ) = -dx*dy* q( (i-0.5)*dx ,(j-0.5)*dy );
  //caso en el que q es funcion de la posicion

  //matriz(punto actual, punto nortesuresteoeste...)
  //condicion de frontera x0
  for(j=1;j<=ny;j++)
  {
    if(x0tipo.contiene("diri") || x0tipo.contiene("Diri"))  {
      matriz_a( (j-1)*nx+1,  (j-1)*nx+1)  -= 3.0*k/dxsy;
      //nodo actual
      matriz_a( (j-1)*nx+1,  (j-1)*nx+2)  +=  1.0*k/dxsy;
      //nodo este
      b( (j-1)*nx+1 ) += - condx0( (j-0.5)*dy ) * 2.0*k/dxsy;
      //valor nodo west como parte del coeficiente
      }
      else
      {
      matriz_a( (j-1)*nx+1 , (j-1)*nx+1 ) -= k/dxsy;
      //actual
      matriz_a( (j-1)*nx+1 , (j-1)*nx+2 ) += k/dxsy;
      //nodo este
      b ( (j-1)*nx+1 )  -= k*dy*condx0 ( (j-0.5)*dy ) ;
    }

  } //fin del for que recorre cada elemento de la frontera west
```

```
 75  //elementos intermedios
     for(i=2;i<=nx-1;i++)
     {
       for(j=1;j<=ny;j++) {
         matriz_a((j-1)*nx+i , (j-1)*nx+i  )  -= 2.0*k/dxsy;
 80      //nodo actual
         matriz_a((j-1)*nx+i,  (j-1)*nx+i+1)  += 1.0*k/dxsy;
         //nodo east
         matriz_a((j-1)*nx+i,  (j-1)*nx+i-1)  += 1.0*k/dxsy;
         //nodo west
 85
       }//fin del for que recorre cada elemento en direccion y
     } //fin del for que recorre los elementos en direccion x
      condicion de frontera x final
     for(j=1;j<=ny;j++) {
       if(xfinaltipo.contiene("diri") || xfinaltipo.contiene("Diri"))
        {
 90      matriz_a( j*nx, j*nx    )  -= 3.0*k/dxsy;
         //nodo actual
         matriz_a( j*nx, j*nx-1 )    = 1.0*k/dxsy;
         //nodo oeste
         b       ( j*nx )           -= condxf( (j-0.5)*dy )*2.0*k/dxsy;
 95      //valor nodo este como parte del coeficiente
       }
       else {
         matriz_a( j*nx , j*nx    ) -= k/dxsy;
         //actual
100      matriz_a( j*nx , j*nx-1 ) = k/dxsy;
         //oeste
         b( j*nx     ) -=  k*dy*condxf  ( (j-0.5)*dy ) ;
       }
     }//fin del for que recorre cada elemento de la frontera este
105  //condicion de frontera y0
     for(i=1;i<=nx;i++) {
       if(y0tipo.contiene("Diri") ||y0tipo.contiene("diri"))
       {
         matriz_a( i,  i    )  -= 3.0*k*dxsy;//nodo actual
110      matriz_a( i,  i+nx )   = 1.0*k*dxsy;//nodo norte
         b( i )  -= condy0( (i-0.5)*dx ) * 2.0*k*dxsy;
       }
       else {
         matriz_a(i,    i)  -= k*dxsy  ;     //actual
115      matriz_a(i, i+nx)   = k*dxsy;       //norte
         b  ( i )           -= k*dx*condy0 ( (i-0.5)*dx ) ;
       }
     } //fin del for que recorre cada elemento en la frontera sur
```

```
  //elementos intermedios
  for(i=1;i<=nx;i++)
  {
    for(j=2;j<=ny-1;j++)
    {
      matriz_a((j-1)*nx+i , (j-1)*nx+i)  -= 2.0*k*dxsy;//nodo
       actual
      matriz_a((j-1)*nx+i , (j-0)*nx+i)  = 1.0*k*dxsy;//nodo norte
      matriz_a((j-1)*nx+i , (j-2)*nx+i)  = 1.0*k*dxsy;//nodo sur
    }
  }
    //frontera yfinal
  for(i=1;i<=nx;i++)
  {
    if(yfinaltipo.contiene("Diri") ||yfinaltipo.contiene("diri"))
    {
      matriz_a( (ny-1)*nx+i, (ny-1)*nx+i )  -= 3.0*k*dxsy;
    //nodo actual
    matriz_a( (ny-1)*nx+i, (ny-2)*nx+i )  = 1.0*k*dxsy;
    //nodo sur
    b ( (ny-1)*nx+i  ) -= condyf( (i-0.5)*dx )*2.0*k*dxsy;
    }
    else
    {
      matriz_a( (ny-1)*nx+i , (ny-1)*nx+i ) -= k*dxsy;
      //actual
      matriz_a( (ny-1)*nx+i , (ny-2)*nx+i ) = k*dxsy;
      //sur
      b(   (ny-1)*nx+i    ) -=  k*dx*condyf ((i-0.5)*dx ) ;
    }
  } //fin del for que recorre cada elemento en la frontera norte

  //matriz_a.Escribir(std_o);
  //b.Escribir (std_o);
  calculado=true;
}

bool diftemp::resolver()
{
  if (calculado==false)
  { std_o<<"Advertencia: posible error en la ejecucion"; calcular(
   );  }

  admlin().adjuntar(matriz_a,x,b);
  bool solucion=admlin().solve();
  int itera=0; bool conver;
```

```
  admlin->obtEstadisticas(itera,conver);
  std_o<<"Iteraciones "<<itera<<"\n";
  for( int i=1 ;  i<=nx ; i++)
    for( int j=1 ;  j<=ny ; j++)
      temp(i,j)=x( (j-1)*nx + i );
  return solucion;
}

void diftemp::exportar ( Cadena archivo)      //public
{
  int i,j;
  Os exportar(archivo);
  exportar << "%Difusion con termino fuente\n" ;
  exportar << "clear";
  exportar << "\n%Mallado: " <<   aform(
    "d=2 dominio = [0,%10g]x[0,%10g] indices= [1: %5d]x[1:%5d]",
    longx , longy , nx , ny )<<"\n\n";
  exportar << "'Posicion x',Px=zeros(1,"<<nx<<");\n" ;
  exportar << "Px=["<< 0.5*dx;
  for(i=2;i<=nx;i++)   exportar << "," <<  (i-0.5)*dx;
  exportar << "];\n";
  exportar << "'Posicion y',Py=zeros(1,"<< ny<<");\n";
  exportar <<"Py=["<<  0.5*dy;
  for(j=2;j<=ny;j++)   exportar << "," <<  (j-0.5)*dy;
  exportar << "];\n";
  exportar << "\n'Temperatura',temp=zeros(" << ny << "," << nx <<")
   ;\n" ;
  for(j=1;j<=ny;j++)
    {
      exportar << "temp("<<j<<",:)=[" << temp(1,j);
      for(i=2;i<=nx;i++)  exportar << "," << temp(i,j);
      exportar << "];\n";
    }
  exportar<< "\nfigure(1); surf(Px,Py,temp);%hold on;\n"<<
    "title('Temperatura')\n"
    <<"xlabel('Eje x[m]');ylabel('Eje y [m]');\n";
  exportar->actualizar();  exportar->cerrar();
}
```

ANEXO D. PROCESO DE SOLUCIÓN DE LA ECUACIÓN DE DIFUSIÓN A PARTIR DEL USO DE LAS LIBRERÍAS: ENCABEZADO DE LA CLASE

```
//Soluci\'on de la ecuacion de difusi\'on en la
//forma generalizada
//archivo "diffgeneral.h"
#pragma once
#include <AdmonEcLin.h>
#include <MallaFV.h>
#include <ConstOLeemalla.h>
#include <GuardarEnsight.h>
#include <GradoLibertadFV.h>
class diffgeneral:public GuardarEnsight
{
private:
  int nelementos  ;
  // numero de volumenes de control
  Puntero<MallaFV> malla;
  Vector_basico <PunteroElmDefs> elems;
  VecinoFVM  vecino;
  Puntero<CampoFVM>     Temperatura;
  //campo de volumenes finitos
  Puntero<GradoLibertadFV> gdl;
  Vector<real>       k;
  //conductividad del material
  Matriz<real>          matriz_a;
  //matriz que contiene los coeficientes de las ecuaciones
  prm_Matriz <real> pm;
  //parametros de la matriz, actualmente no es necesario
  //pero se recomienda para posterior implementacion
  //general
  Vector<real>       b;
  //vector que contienen las "constantes"
  Vector<real>       x;
  //vector que contiene la informacion de la solucion
  //esta informacion sera transmitida al campo fv
  Puntero<AdmonEcLin> admlin;
```

```
35    //////////banderas/////////////////////
      bool  calculado;
      bool  leido;
      bool  solucionado;
    public:
40    diffgeneral(void);
      ~diffgeneral(void);
      int Leer (Cadena archivo,Cadena solucionador);
    //Encargada de  inicializar la mayoria de los elementos
      int calcular();
45  //Se encarga de calcular los coeficientes de las ecuaciones
      bool resolver();
    //Se encarga de resolver el sistema de ecuaciones
      virtual void Reportarresultados();
    //se encarga de enviar la informacion de manera que los
50  //resultados puedan verse en Paraview
       virtual real q (real x, real y);
    //Funcion de generacion de calor
    };
```

ANEXO E. PROCESO DE SOLUCIÓN DE LA ECUACIÓN DE DIFUSIÓN A PARTIR DEL USO DE LAS LIBRERÍAS: ARCHIVO DE CÓDIGO FUENTE

```
//Soluci\'on de la ecuacion de difusi\'on en la forma generalizada
//archivo "diffgeneral.cpp"
#include "diffgeneral.h"

diffgeneral::diffgeneral(void)
{
  calculado = leido = solucionado = false;
}

diffgeneral::~diffgeneral(void) { }

int diffgeneral::Leer(Cadena archivo, Cadena solucionador)
{
  static const char *mallador_tb[] = {"mallador", NULL};
  ////////definicion de la malla para el metodo de los volumenes
   finitos
  Cadena cadmalla;//Cadena para enviar el argumento
  Is is(archivo);//Se encarga de leer el archivo
  is->obtComando(cadmalla,mallador_tb);
  //Se lee el arhivo y se obtiene el comando
  malla.vincular(new MallaFV());
  ConstOLeeMalla(malla(),cadmalla);
  //Se genera la malla
  vecino.iniciar(malla());
  malla->CalcConectividad(vecino);
  //Se calcula cuales son los vecinos para c/volumen
  Temperatura.vincular(new CampoFVM(malla(),"Temperatura"));
  //inicializacion del campo de volumenes finitos
  gdl.vincular(new GradoLibertadFV(malla(),1));
  //grados de libertad para cada volumen de control
  nelementos=malla->obtNoElem();
  //obtener el numero de elementos de la malla
  matriz_a.chaSize(nelementos, nelementos);  matriz_a.llenar(0.0);
  //dimensionamiento de la matriz y el vector de constantes a*x=b
```

```
  b.chaSize(nelementos);   b.llenar(0.0);
35  //importante la inicializacion por 0
  x.chaSize(nelementos);   x.llenar(224.0);
  //Importante la inicializacion para obtener rapida convergencia
  ///////propiedades de material////////////
  k.chaSize(nelementos);   k.llenar(50);
40  ////calculo de la geometr\'ia  de cada volumen de control
  elems.chaSize(nelementos);
  //Dimensionamiento del vector de punteros
  for (int i=1; i<=nelementos;i++)
  {
45    elems(i).refill( malla->obtTipoElem(i), malla->obtNoDimEspacio
      () );
    elems(i)->CalcGeomParameters( malla->obtCoorElem(i) );
  }
  //////////solucionador de ecuaciones lineales////////////////
  Is sol(solucionador);
50  //Se abre el archivo
  pm.Leer ( sol );
  //Se lee los parametros respectivos a la matriz
  admlin.vincular(new AdmonEcLin(EXTERNAL_STORAGE));
  //Se inicializa la memoria dinamica del puntero
55  admlin().Leer(sol);
  //Se lee la informacion del metodo de solucion
  leido=true;
  //Cambio de valor del parametro de control
  return 1;
60 }
real diffgeneral::q(real x, real y) { return 5000.0; }

int diffgeneral::calcular(void)
{
65  //control de errores
  if (leido==false)
  {std_o<<"Advertencia: no se ha leido los datos"
  <<" del problema\n"; return 0;
  }
70  if (calculado)
  {std_o<<"Ya se habia calculado los coeficientes\n"; return 0;}
  int i , j , l ; //contadores
  int veci;//vecino numero veci
  int conx, conxvec;//numero de conexiones para el elemento actual
   y el vecino
75  int h, nbi=malla->obtNoIndFront ();;//Contador para el numero de
   fronteras, numero de  fronteras
```

```
Vector_xyz  <real> posicion(3); //vector para obtener la posicion
  de un volumen de control especifico
real u1,u2,ut; //u conductancia 1,2(cuando exista) y total
Cadena boname, bovalue; //nombre de la condicion de frontera (
 boundary) bovalue valor en la condicion de frontera
real valor;
//se obtiene el numero de condiciones de frontera diferentes
for(i=1;i<=nelementos;i++)
{
  posicion = elems(i)->centroid(malla->obtCoorElem(i));
  //se obtiene la posicion del centroide del volumen de control
   para determinar la generacion de calor en ese
  //punto. Si la generacion no es una constante, se recomienda
   realizar una integracion numerica
  b(i) = - elems(i)->getVolume()* q( posicion(1),posicion(2) );
  conx=elems(i)->GetNumOfConex();
  //se obtiene el numero de lados
  for( j=1 ; j<=conx ; j++ )//por cada conexion
  {
    veci=malla->getConx(i,j);
    //obtiene el numero identificador del vecino al volumen i en
     la direccion j
    if (veci!=0)//si el vecino no es una frontera
    {
      conxvec=elems(veci)->GetNumOfConex();
      //se obtiene el numero de lados del
      for ( l=1 ; l<=conxvec ; l++ )
      {
       if(malla->getConx(veci,l)==i)   break;
      }
      //se busca la direccion en la cual el vecino se conecta con
        el volumen actual,
      u1=k(i)/elems(i)->getDelta(j);
      //conductancia del elemento actual (i) en dir j
      u2=k(veci)/elems(veci)->getDelta(l);
      //conductancia del elemento vecino en direccion l
      ut=1.0 / ( (1.0/u1)+ (1.0/u2) );
      //conductancia total
      matriz_a( i,  i  ) -= ut* elems(i)->getArea(j);
      matriz_a( i,veci) =  ut* elems(i)->getArea(j);
      //asignacion de los coeficientes a la matriz
    }
    else//analisis en la frontera
    {
      for (h=1; h<=nbi; h++) //por cada frontera diferente
      {
```

```
        if (malla->LadoFront(i,j,h,elems(i)()))
          {
            //si el volumen i en direccion j tiene el tipo
            //de frontera h entonces
120         boname=malla->obtNombreIndFront(h);
            //se obtiene el nombre del tipo de frontera
            bovalue=boname.despues('=');
            //se obtiene la cadena despues del igual
            valor=atof(bovalue.carts());
125         //se obtiene el valor numerico de la frontera
            if (boname.contiene("T") ||
               boname.contiene("Dirichlet")||
               boname.contiene("dirichlet")
              ) //tipo de frontera
130         {
              ut = k(i)/elems(i)->getDelta(j);
              //se calcula la conductancia total que corresponde a
               la conductancia del elemento actual para este  tipo
               de frontera
              matriz_a(i,i)   -=  ut * elems(i)->getArea(j);
               b(i)  -=  ut * elems(i)->getArea(j)*valor;
135            //se actualiza el valor de los coeficientes de la
                matriz
            }
            else
            {
              if (boname.contiene("Neumann")||
140             boname.contiene("neumann")
                )
              {
              b(i)   -=  valor * k(i) * elems(i)->getArea(j);
              }//condicion de frontera dirichlet en la cual se da
               el valor de la derivada en direccion normal a  la
               superficie
145           if (boname.contiene("Q"))
              {
              b(i)   -=  valor * elems(i)->getArea(j);
              }//tipo especial de condicion de frontera dirichlet
               que solo es aplicada a la temperatura
            }
150       }
        }
      }//final del else
    } //final del for para cada lado
  };
155 calculado=true;
```

```
  //se actualiza el valor del parametro de control
  return 1;
}

bool diffgeneral::resolver()
{
  if(calculado==false) calcular();
  //verificacion del parametro de control
  admlin().adjuntar(matriz_a,x,b);
  //se debe decir al administrador de ecuaciones lineales cual es
   la matriz de coeficiente, el vector solucion y el vector de
   termino independientes
  bool solucion=admlin().solve();
  //se resuelve el sistema de eucaciones on una instruccion muy
   sencilla
  int itera; bool conv;
  admlin().obtEstadisticas(itera, conv);
  //para los sistemas iterativos puede ser conveniente conocer el
   numero de iteraciones necesarias para resolver el sistema de
   ecuaciones
  std_o<<"iteraciones="<<itera<<"\nTiempo"<<
    admlin().obttiempoCPUsolve() ;
  //Impresion en pantalla, para una clase general no es conveniente
    realizar la impresion en pantalla, se recomienda en este caso
   con fines educativos mostrar la informacion
  if (conv==1) std_o<<"\nEl sistema si convergio";
  gdl->vector2campo(x,Temperatura());
  //se transfiere el vector solucion al campo de volumenes finitos
  solucionado=true;
  //se actualiza el valor del parametro de control
  return solucion;
}
void diffgeneral::Reportarresultados()
{
  if (solucionado == false) resolver();//se verifica que se haya
   solucionado el sistema de ecuaciones
  //antes de exportar los resultados
  GuardarEnsight::volcar(Temperatura(),0);
  //esta instruccion es la encargada de  descargar la informacion
   del campo de temperatura  a un archivo compatible con Paraview

}
```

ANEXO F. ENCABEZADO DE LA CLASE CONVDIFF

```
#pragma once
//Archivo convdiff.h
#include <MallaFV.h>
#include <ConstOLeemalla.h>
#include <AdmonEcLin.h>
#include <GuardarEnsight.h>
#include <GradoLibertadFV.h>

//PROCEDIMIENTO: DECLARACION DE VARIABLES: se necesita una
//serie de variables y funciones para el correcto funcionamiento
//de la clase

class convdiff:public GuardarEnsight
{
private:

protected:
  //es necesario declarar la clase base GuardarEnsight como public
   pues se utilizar\'an algunas funciones para  exportar los
   resultados a matlab
  //Aunque no son estrcitamente necesarias, se declaran las
   variables que definen el ancho y alto del elemento a analizar.
   Dicha definicion puede no ser correcta para mas de un
   superelemento
  real longx,longy ;
  //Es necesario conocer el numero de volumenes de control
  int nelementos ;
  //Se define puntero de MallaFV
  Puntero<MallaFV> malla;
  //Se define un vector que contenga la informacion geometrica de
   cada volumen de control
  Vector_basico <PunteroElmDefs> elems;
  //En el MVF es necesario conocer el numero de lados que tiene
   cada volumen de control, raz\'on por la cual se debe definir una
    variable del tipo VecinoFVM que permita obtener dicha
   informacion.
  VecinoFVM  vecino;
```

```
//Para exportar los resultados es necesario guardar la
 informacion obtenida en una variable de tipo Puntero<CampoFVM> y
  definir una variable del tipo Puntero<GradoLibertadFV> que
 tendr\'a en cuenta el n\'umero de incognitas por cada volumen de
  control
Puntero<CampoFVM>     campo_u;
Puntero<CampoFVM>     campo_v;
Puntero<CamposFVM>    campo_vel;
Puntero<CampoFVM>     Temperatura;
Puntero<GradoLibertadFV> gdl;
//Para resolver sistema de ecuaciones lineales es comun almacenar
  los coeficientes del  sistema de ecuaciones en una matriz y
 almacenar el conjunto de "constantes" en un vector. Adem\'as,
 los resultados tambi\'en son almacenados en un vector.
Matriz<real>            matriz_a;  //matriz de coeficientes
Vector<real>        b;      //vector de "constantes"
Vector<real>        x;      //vector solucion
//Se crean dos variables que definen el valor de la velocidad (en
  x e y) para cada volumen de control, se asume que el valor de
 la velocidad dentro del V.C. puede  ser representado por un
 unico valor.
Vector<real>       u;//velocidad x
Vector<real>       v;//velocidad en y
//Se debe definir una serie de propiedades para el material a
 usar, en este caso, se propone la definicion de las propiedades
 para cada volumen de control Propiedades variables en funcion de
  la  temperatura requiere de m\'etodos iterativos para
 solucionar el sistema de ecuaciones por lo que no se implementa
Vector<real>       k;     //conductividad del material
Vector<real>       rho;     //densidad del material
Vector<real>       C_p; //Calor especifico del material
//k/C_p define el valor de gamma para la ec. de energia pero de
 forma general se puede tomar C_p=1 y asumir que gamma = k
 siempre y cuando se introduzca en las unidades correctas el
 valor de la generacion de calor
//La exactitud y estabilidad en el an\'alisis num\'erico de la
 ecuaci\'on de  convecci\'on y difusi\'on puede depender del n\'
 umero de peclet por lo que es  necesario crear  unas variables
 que almacenen dicha informaci\'on.
real           pe_min  ;
real           pe_max  ;
//Para resolver el sistema de ecuaciones lineales es necesario
 crear un objeto del tipo AdmonEcLin que se encarga de obtener
 toda la informaci\'on necesaria para definir el m\'etodo de
 solucion, entre otras funciones.
Puntero<AdmonEcLin> admlin;
```

```
  prm_Matriz <real> pm;
  //Es necesario establecer un orden en la soluci\'on del problema;
    dicho orden, es verificado por una serie de banderas que
   permiten saber si el proceso ya fue ejecutado
  //////////banderas////////////////////
  bool  calculado;
  bool  leido;
  bool  solucionado;
  bool  primer_peclet;

public:
  convdiff(void) {primer_peclet=true; calculado=false; leido=false;
    }
  ~convdiff(void);
  // Se leen los datos de entrada y se inicializan algunas
   variables
  int Leer (Cadena problema, Cadena solucionador);
  //Calculo de los coeficientes del sistema de ecuaciones
  int Calcular();
  //solucion del sistema de ecuaciones lineales, es necesario
   verificar que el proceso de solucion se halla llevado
   correctamente
  bool Resolver();
  //Se debe definir una funci\'on que tenga el cuenta el esquema
   que se est\'a utilizando y el numero de peclet, es por eso que
   se define la funcion_a
  real funcion_a ( real pe, Cadena modo);
  real difusion(int vol1, int dir1, int vol2, int dir2, real area);
  real difusion(int vol1, int dir1, real area);
  real difusion(int vol1, int dir1, real area, real conveccion_f);
  real conveccion( int vol1, int dir1 ,real area, Vector_xyz<real>
   Vel);
  virtual real q (real x, real y);
  //Funciones para el calculo y salida de datos del m\'aximo valor
   absoluto del numero de peclet
  real peclet(real F, real D);
  real peclet_min(void);
  real peclet_max(void);
  //La funci\'on que se encarga de guardar la informaci\'on para
   presentar  los resultados en Paraview es la siguiente
  virtual void Reportarresultados();
};
```

ANEXO G. ARCHIVO DE CÓDIGO FUENTE DE LA CLASE CONVDIFF

```
#include "convdiff.h"
//Archivo convdiff.cpp
convdiff::~convdiff(void)
{
}
int convdiff::Leer(Cadena problema, Cadena solucionador)
{
  //se crea una cadena necesaria para la funcion obtener comando
  static const char *mallador_tb[] = {"mallador", NULL};
  //Se crea una variable que almacena la instruccion a ejecutar
  Cadena cadmalla;
  //En general, la informaci\'on del problema se va a almacenar en
   una serie de archivos, es por esto, que se debe leer un archivo
   que contiene la definici\'on de la geometr\'ia y de  las
   particiones
  Is is(problema);
  is->obtComando(cadmalla,mallador_tb);
  //Se obtiene el comando que contiene el nombre de los archivos de
    la geometria y las particiones
  malla.vincular(new MallaFV());
  ConstOLeeMalla(malla(),cadmalla);
  //una vez se ha generado la malla, se calcula la conectividad
   mediante las siguientes dos instrucciones
  vecino.iniciar(malla());
  malla->CalcConectividad(vecino);
  //Es muy importante conocer el numero de volumenes de control que
    se seleccion\'o en el arhivo*.parts
  nelementos=malla->obtNoElem();
  //Como informaci\'on adicional, puede calcularse el m\'aximo tama
   \~no de la reg\'on a analizar
  Vector_xyz <real> min,max;
  malla->obtCoorMinMax(min,max);
  longx=max(1)-min(1);
  longy=max(2)-min(2);
  ///////////////////////////////////////////////////
  //////////////dimensionalizaci\'on////////////////
```

```
//dimensionamiento de la matriz y el vector de constantes a*x=b
 es importante redimensionar e inicializar estas variables, ya
 sea con el valor 0 o un valor especifico para el vector solucion
matriz_a.chaSize(nelementos, nelementos);
b.chaSize(nelementos);
x.chaSize(nelementos);
matriz_a.llenar(0.0);
b.llenar(0.0);
x.llenar(1.0);

//Adem\'as hay que inicializar el campo de temperatura
Temperatura.vincular(new CampoFVM(malla(),"Temperatura"));
campo_vel.vincular(new CamposFVM(malla(),"Velocidad"));
gdl.vincular(new GradoLibertadFV(malla(),1));
//velocidad u (direccion x ) y velocidad v (direccion y)
u.chaSize (nelementos);   v.chaSize (nelementos);
//informacion geom\'etrica de cada volumen de control
elems.chaSize(nelementos);
for (int i=1; i<=nelementos;i++) {
  elems(i).refill( malla->obtTipoElem(i), malla->obtNoDimEspacio
   () );
  elems(i)->CalcGeomParameters( malla->obtCoorElem(i) );
  u(i) =  elems(i)->centroid(malla->obtCoorElem(i))(1);
  v(i) = -elems(i)->centroid(malla->obtCoorElem(i))(2);
}
////////////propiedades de material///////////
k.chaSize(nelementos);        k.llenar(0.01);
rho.chaSize(nelementos);      rho.llenar(1.0);
C_p.chaSize(nelementos);      C_p.llenar(1.0);
//Se leen las condiciones para el solucionador de ecuaciones
Is solucion(solucionador);
pm.Leer ( solucion );
//y se analiza dicha informacion mediante el
admlin.vincular(new AdmonEcLin(EXTERNAL_STORAGE));
admlin().Leer(solucion);
//se establece la bandera como true que permite reconocer que el
//proceso de lectura ha sido completamente ejecutado
GuardarEnsight::ponModoGuardar(BINARY,BINARY);
leido=true;
return 1;
}
//se calcula la difusividad entre dos nodos vecinos
//vol1 numero correspondiente al volumen de control actual
//vol2 numero correspondiente al volumen de control vecino
//dir1 corresponde al numero del lado del volumen
//de control vol1 en la cual se realizan los calculos
```

```
real convdiff::difusion(int vol1, int dir1, int vol2, int dir2,
 real area)
{
  real D1, D2, Dt, gamma;
  gamma = k(vol1)/C_p(vol1) ;   D1= gamma / elems(vol1)->getDelta(
   dir1);
  gamma = k(vol2)/C_p(vol2) ;   D2= gamma / elems(vol2)->getDelta(
   dir2);
  Dt    = area / (1.0/D1 + 1.0/D2);
  return Dt;

}
//version de la funcion de calculo de la difusividad
//para un lado que limita con una condicion de frontera
//en donde se define el coeficiente de conveccion
//esta funcion solo es validad cuando \phi=T
real convdiff::difusion(int vol1, int dir1, real area, real
 conveccion_f)
{
  real D1, D2, Dt, gamma;
  gamma = k(vol1)/C_p(vol1)   ;
  D1 = gamma / elems( vol1  )->getDelta(dir1);
  D2 = conveccion_f;
  Dt = area / (1.0/D1 + 1.0/D2);
  return Dt;

}
//version de la funcion de calculo de la difusividad
//para un lado que limita con una condicion de frontera
real convdiff::difusion(int vol1, int dir1, real area)
{
  real Dt, gamma  ;
  gamma = k(vol1)/C_p(vol1) ;
  Dt = gamma / elems( vol1  )->getDelta(dir1);
  Dt = area *Dt;
  return Dt;
}
//Calculo del flujo convectivo
real convdiff::conveccion( int vol1, int dir1 ,real area,
 Vector_xyz<real> Vel)
{
  real F ;
  F = rho(vol1) * area * (elems(vol1)->GetDirSide(dir1) * Vel) ;
  return F;
}
//generacion de calor
```

```
real convdiff::q(real x, real y) {  return 0; }
//se introduce la funcion peclet para actualizar el valor del
//peclet minimo y maximo, el valor del peclet es un parametro
//importante en el flujo convectivo-difusivo
real convdiff::peclet(real F, real D)
{
  real peclet=F/D;
  if ( primer_peclet )
  { pe_max = pe_min = abs(peclet); primer_peclet=false; }
  if ( abs(peclet) > pe_max ) pe_max=abs(peclet);
  if ( abs(peclet) < pe_min ) pe_min=abs(peclet);
  return peclet;
}
//Calculo de la funcion que depende del tipo de analisis
//y del valor del peclet
real convdiff::funcion_a ( real pe, Cadena modo)
{
  if (modo.contiene("upwind") || modo.contiene("Upwind")
    || modo.contiene("UDS") )
  {
    return 1;
  }
  if (modo.contiene("CDS"))
  {
    return 1-fabs(pe);
  }
  if ( modo.contiene("hybrid") || modo.contiene("hibrido") )
  {
    return max(0.0,1-0.5*fabs(pe));
  }
  if ( modo.contiene("power") || modo.contiene("law") )
  {
    return max(0.0,pow(1-0.1*fabs(pe),5));
  }
    if ( modo.contiene("exp") || modo.contiene("Exp") )
  {
    return fabs(pe)/(exp(fabs(pe))-1.0);
  }

return 0;//Devolver este valor implica error
}
//Calculos de los coeficientes de la matriz de coeficientes
//y el vector de terminos independientes.
int convdiff::Calcular(void)
{
  if (leido==false)
```

```
160 {std_o<<"ERROR: Calcular NO SE EJECUTO EL PROCESO CORRECTAMENTE";
    return 0;
    }
    if (calculado)
    {
165 std_o<<"Ya se habia ejecutado la funcion Calcular: "
      <<"No se realizan los calculos\n"; return 0;
    }
    int e, h, l; //contadores, direccion
    int veci; int conx; // veci contador para el volumen vecino //
     conx numero de lados del elemento actual
170 Cadena boname, bovalue; //Nombre condiciones de frontera
    int nbi = malla->obtNoIndFront (); //numero de condiciones de
     frontera
    double valor;//valor de la condicion de frontera
    real F, D, Pe, a_e, a_b, b_b,  A;
    real area, y, volumen; //y posicion del centroide en el eje y
175 Vector_xyz<real> vel(2), pos;
    for(e=1;e<=nelementos;e++)//por cada volumen
    {
      pos=elems(e)->centroid(malla->obtCoorElem(e));
      volumen=elems(e)->getVolume();
180   b(e) = q (pos(1), pos(2)) * volumen;
      //calculo de la gen eracion de calor en funcion  de la posicion
      conx = elems(e)->GetNumOfConex();
      //numero de lados para el volumen actual
      for( l=1 ; l<=conx ; l++ )//por cada conexion
185   {
        veci = malla->getConx(e,l);
        //se obtiene el numero indicador del volumen vecino a e en la
          direccion l
        area = elems(e)->getArea(l);
        //area de la superficie del volumen e en dir l
190     if (veci!=0)//si el vecino no es una frontera
        {
          for ( h=1 ; h<=conx ; h++ )
          {
           if(malla->getConx(veci,h)==e)   break;
195       }
          //se busca el lado por el cual el vecino se conecta con el
           volumen actual
          vel(1) = ( u(e) + u(veci) )/2.0;
          vel(2) = ( v(e) + v(veci) )/2.0;
          //se toma la velocidad promedio entre los volumenes de
           control
```

```
200         F  = conveccion(e,l,area,vel); //calculo del flujo
            convectivo
            D  = difusion( e, l, veci, h, area); //calculo del flujo
            difusivo
            Pe = peclet( F,D ); //calculo del peclet, se sugiere el uso
             de la funcion
            A  = funcion_a(Pe, "UDS"); //Calculo de la funcion a que
            depende  del peclet y el m\'etodo de an\'alisis
            a_e = D * A + std::max( 0.0 , -F ); //coeficiente para el
            volumen actual
205         matriz_a(e,e   ) +=   a_e + F;
            matriz_a(e,veci)  =  -a_e;
          }
          else//caso en el cual veci es cond. frontera
          {
210         for (h=1; h<=nbi; h++)
            {
              if (  malla->LadoFront(e,l,h,elems(e)())   )
              {
                vel(1) =  elems(e)->centroArea(malla->obtCoorElem(e),l)
                 (1) ;
215             vel(2) = -elems(e)->centroArea(malla->obtCoorElem(e),l)
                 (2) ;
                F      = conveccion(e,l,area,vel) ;
                boname  = malla->obtNombreIndFront(h);
                bovalue = boname.despues('=');
                valor   = atof(bovalue.carts());
220             D      = difusion(e,l,area);
                if ( boname.contiene("T") || boname.contiene("dirichlet
                ")
                  || boname.contiene("Dirichlet")
                  )
                {
225               if( boname.contiene("L") )
                  {
                    y      = elems(e)->centroArea(malla->obtCoorElem(e),
                     l)(2);
                    valor = valor * ( 2.0 - y/longy );
                  }
230               Pe = peclet( F,D );
                  A  = funcion_a(Pe, "UDS");
                  a_b   =  D*A + std::max(F,0.0);
                  b_b   = (D*A + std::max(-F,0.0))*valor;
                }
235             if ( boname.contiene("Q") )
                {
```

```
                a_b    = F;
                b_b    = valor * area /C_p(e);
              }
240           if ( boname.contiene("Neumann")||boname.contiene("
               neumann") )
              {
                a_b    = F;
                b_b    = valor * area * k(e)/C_p(e);
              }
245           matriz_a (e,e) += a_b;//Sp;
              b (e)      += b_b;//Su;
            }
          }
        }
250
      }

    }
    calculado=true;
255 return 1;
}

bool convdiff::Resolver()
{
260 int iter; bool conv;
  if(calculado==false) Calcular();
  admlin().adjuntar(matriz_a,x,b);
  bool solucion=admlin().solve();
  admlin->obtEstadisticas(iter, conv);
265 gdl->vector2campo(x,Temperatura());
  gdl->vector2campo(u,campo_vel()(1));
  gdl->vector2campo(v,campo_vel()(2));
  solucionado=true;
  return solucion;
270 }

void convdiff::Reportarresultados()
{
  volcar(Temperatura(),0);
275 volcar(campo_vel()   ,0);
}
real convdiff::peclet_min(void) {return pe_min;}
real convdiff::peclet_max(void) {return pe_max;}
```

ANEXO H. CLASE PARA LA SOLUCIÓN DE LA ECUACIÓN DE DIFUSIÓN EN ESTADO TRANSITORIO: ENCABEZADO DE LA CLASE

```
//Archivo DifusionTransitoria.h
#pragma once
#include <AdmonEcLin.h>
#include <MallaFV.h>
#include <ConstOLeemalla.h>
#include <GuardarEnsight.h>
#include <GradoLibertadFV.h>
class DifusionTransitoria :
  public GuardarEnsight
{
  int nelementos; // numero de volumenes de control
  //////////MALLA DE VOLUMENES FINITOS/////////////
  Puntero<MallaFV> malla;
  Vector_basico <PunteroElmDefs> elems;
  VecinoFVM  vecino;
  Puntero<CampoFVM>     phi;
  Puntero<GradoLibertadFV> gdl;
  Vector<real>  S_fuente_ext;//Agregado N-S
  //////////PROPIEDADES DEL MATERIAL//////////////
  Vector<real>  rho;
  Vector<real>  k;
  Vector<real>  cp;
  Vector<real>  alph;
  Vector_basico <Vector_xyz<real>> Gamma; //Agregado N-S
  ///////////SISTEMA//DE//ECUACIONES/////////////
  //Matriz<real>  matriz;
  Puntero<Matriz_Dispersa<real>>  matriz;
  EstDispersa disp;
  prm_Matriz <real> pm; //parametros de la matriz
  Vector<real>    b;
  Vector<real>    x;
  Vector<real>    x_futuro;
  ////SOLUCION//DEL//SISTEMA//DE//ECUACIONES//////
  Puntero<AdmonEcLin> admlin;
```

```
35    /////////CALCULO//DEL//FOURIER////////////////
      real Fourier(real alpha, real dt, real dx);
      /////////DEPENDENCIA//DEL//TIEMPO////////////
      real theta;
      Puntero<prmTiempo> tiempo_p;
40    ///LECTURA//CADENA//COND//FRONTERA/////
      Cadena LecturaCadena;//Agregado N-S
      //////////PARAMETROS//DE//CONTROL////////////
      bool calculado;
      bool leido;
45    bool solucionado;
      bool PropiedadesExternas;//Agregado N-S
      bool mallaExterna;//Agregado N-S
      bool UnSoloGamma;//Agregado N-S
      bool primera_solucion;
50    bool primer_fourier;
      real phi_inicial;
      real Fo_min;
      real Fo_max;
    protected:
55    virtual int CalculoUnTiempo();
      virtual bool SolucionUnTiempo(bool vector2campo= true);
      virtual void Reportarresultados();
      void EstructurarDisp();
    public:
60    DifusionTransitoria(void);
      ~DifusionTransitoria(void){}
      virtual int Leer (Cadena archivo,Cadena solucionador,
        Cadena def_tiempo, real T_inicial=100);
      virtual int CalcularYResolver(real dependencia_futuro);
65    bool ResetCalculo ( void);
      int GetNoVol(void) {return nelementos;}
      virtual real S_fuente (real x, real y);
      ///////M\'ETODOS Agregado N-S///////////////////
      void SetPropiedadesExternas ( bool opcion );
70    bool SetMalla ( Puntero<MallaFV> &MallaEntrada  );
      void GetMalla ( Puntero<MallaFV> &Salida);
      bool SetMallaeIniciar ( Puntero<MallaFV> &MallaEntrada , Cadena ,
        Cadena , Cadena );
      bool SetFuente (Vector<real> S_f );
      virtual real S_fuente (int nvol);
75    bool SetPasadoPartida (Vector<real> phi_pasado, Vector<real>
       phi_pto_partida_iteracion) ;
      bool SetPasado (Vector<real> phi_pasado);
      bool SetPropiedades ( Vector<real> G );
```

```
  bool SetPropiedades ( Vector<real> Gx,  Vector<real> Gy, Vector<
   real> Gz);
  Vector_basico <PunteroElmDefs> GetElmDefs();
80  Puntero<prmTiempo> GetTiempo();
  void SetNombreCampo (char *ncampo);
  Vector<real> CalcularCoefUnTiempo(real   dependencia_futuro,
   Puntero<prmTiempo> t_ext , bool &triunfo);
  Vector<real> CalcularCoefUnTiempo(real dependencia_futuro, bool &
   triunfo);
  Vector<real> SolucionCoefUnTiempo(bool &triunfo,bool vector2campo
    = false);
85 };
```

ANEXO I. CLASE PARA LA SOLUCIÓN DE LA ECUACIÓN DE DIFUSIÓN EN ESTADO TRANSITORIO: ARCHIVO DE CÓDIGO FUENTE

```
//Archivo DifusionTransitoria.cpp
#include "DifusionTransitoria.h"
//Constructor de la clase: valores por defecto para los par\'
 ametros de control
DifusionTransitoria::DifusionTransitoria(void)
{
  primera_solucion=primer_fourier=true;
  calculado = leido = solucionado = false;
  PropiedadesExternas = mallaExterna = false;//Agregado N-S
  UnSoloGamma = false;//Agregado N-S
}
//Funcion para la lectura de datos de la clase
int DifusionTransitoria::Leer(Cadena archivo,
  Cadena solucionador, Cadena def_tiempo, real phi_o)
{
  int i;
  int nd; //Agregado N-S
  static const char *mallador_tb[] = {"mallador", NULL};
  ////////definicion de la malla//////////////////
  Cadena cadmalla;//Cadena para enviar el argumento
  if (!mallaExterna) {//Agregado N-S
    Is is(archivo);//Se encarga de leer el archivo
    is->obtComando(cadmalla,mallador_tb);
    //Se lee el arhivo y se obtiene el comando
    malla.vincular(new MallaFV());
    ConstOLeeMalla(malla(),cadmalla);
    vecino.iniciar(malla());
    malla->CalcConectividad(vecino);
  }//Agregado N-S
  phi.vincular(new CampoFVM(malla(),"Temperatura"));
  gdl.vincular(new GradoLibertadFV(malla(),1));
  //////////solucionador de ecuaciones lineales////////////////
  Is sol(solucionador);
  pm.Leer ( sol );
```

```
admlin.vincular(new AdmonEcLin(EXTERNAL_STORAGE));
admlin().Leer(sol);
nelementos=malla->obtNoElem();
k.chaSize(nelementos);
rho.chaSize(nelementos);
cp.chaSize(nelementos);
alph.chaSize(nelementos);
if (!PropiedadesExternas) {//Agregado N-S
  k.llenar(50.0);
  rho.llenar(500.0);
  cp.llenar(1.0);
  for (i=1;i<=nelementos;i++)
    alph(i)=k(i)/(rho(i)*cp(i));
}//Agregado N-S
else {//Agregado N-S
  Gamma.chaSize (nelementos) ;//Agregado N-S
  nd = malla->obtNoDimEspacio ();
  for (i=1;i<=nelementos;i++)//Agregado N-S
    Gamma(i).chaSize(nd);//Agregado N-S
}//Agregado N-S
x.chaSize(nelementos);
x_futuro.chaSize(nelementos);
if (!PropiedadesExternas) {//Agregado N-S
  x.llenar(phi_o);
  x_futuro.llenar(phi_o);
}//Agregado N-S
phi_inicial = phi_o ;
//matriz.chaSize(nelementos);//se dimensionan pero no
b.chaSize(nelementos);//se llenan, esto se hace en
 CalculoUnTiempo
///calculo de la geometr\'ia  de cada volumen de control
elems.chaSize(nelementos);
for (i=1; i<=nelementos;i++) {
  elems(i).refill
    ( malla->obtTipoElem(i), malla->obtNoDimEspacio() );
  elems(i)->CalcGeomParameters( malla->obtCoorElem(i) );
}
EstructurarDisp();
//Se asigna la memoria al puntero tiempo_p si no
//se realiza la llamada a SetMallaeIniciar
if (!mallaExterna)  {
tiempo_p.vincular(new prmTiempo);
tiempo_p->Leer(def_tiempo);
tiempo_p->iniciarCicloTiempo();
}
leido=true;
```

```
  return 1;
}
real DifusionTransitoria::Fourier(real alpha, real dt, real dx)
{
real fo; fo = alpha*dt/(dx*dx);
if (primer_fourier) { Fo_min=Fo_max=fo; primer_fourier=false;}
if (fo < Fo_min) {Fo_min=fo;}
if (fo > Fo_max) {Fo_max=fo;}
return fo;
}

int DifusionTransitoria::CalcularYResolver(real dependencia_futuro)
{
  if (!leido || calculado) return 0;
  //Para el caso estacionario solo se resuelve una vez
  if (tiempo_p->estacionario()) {
    theta=1; //Este valor es por defecto en el
    CalculoUnTiempo();//estado transitorio
    SolucionUnTiempo();
    Reportarresultados();
  }
  else {//Se asigna el valor del parametro de entrada
    theta=dependencia_futuro;
    gdl->vector2campo(x_futuro,phi());
    Reportarresultados();
    do {
      tiempo_p->incrementarTiempo();
      CalculoUnTiempo();
      SolucionUnTiempo();
      Reportarresultados();
    }
    while(!tiempo_p->finalizo());
    std_o<<"\nMinimo Fourier "<<Fo_min
      <<"\nMaximo Fourier "<<Fo_max<<"\n";
  }
  calculado = solucionado = true;
  return 1;
}
real DifusionTransitoria::S_fuente(real x, real y)
{ return 5000; }
int DifusionTransitoria::CalculoUnTiempo()
{
  int i , j , l ; //contadores
  int veci;//vecino numero veci
  int conx, conxvec;//numero de conexiones
```

```
int h, nbi=malla->obtNoIndFront ();//Contador para el numero de
 fronteras, numero de fronteras
real aj, vol, tr;//coeficiente, volumen, transitorio
real hfrontera;//conveccion en la frontera
real bc;//coeficiente boundary
real u1, u2,  ut; //u conductancia 1,2(cuando exista) y total
real valor;//valor en la frontera
int o;//Agregado N-S
Cadena med;//Agregado N-S
//se obtiene el numero de condiciones de frontera diferentes
Vector_xyz  <real> posicion(3); //vector para obtener la posicion
  de un volumen de control especifico
Cadena boname, bovalue;//nombre de la condicion de frontera (
 boundary) //bovalue caden valor en la condicion de frontera
if ( !PropiedadesExternas ) {//Agregado N-S
  x=x_futuro;//se actualiza el valor del pasado
}//Agregado N-S
matriz->llenar(0.0);
b.llenar(0.0);

for(i=1;i<=nelementos;i++)  {
  posicion = elems(i)->centroid(malla->obtCoorElem(i)); //se
   obtiene la posicion del centroide del VC
  vol = elems(i)->getVolume();
  if (!PropiedadesExternas) {//Agregado N-S
    b(i) =  vol * S_fuente( posicion(1),posicion(2) ); }//
     Agregado N-S
  else {//Agregado N-S
    b(i) =   S_fuente (i); }//Agregado N-S
  conx=elems(i)->GetNumOfConex(); //se obtiene el numero de lados
  if (!tiempo_p->estacionario()) {
    tr = rho(i)* vol / tiempo_p->Delta();
    matriz()( i, i ) += tr;
    b(i)       += tr*x(i);
  }//efecto transitorio
  for( j=1 ; j<=conx ; j++ ) {//por cada conexion
    veci = malla->getConx(i,j);//obtiene el numero identificador
     del vecino al volumen i en la direccion j
    if (veci!=0)//si el vecino no es una frontera
    {
      conxvec = elems(veci)->GetNumOfConex();
      //se obtiene el numero de lados del VC vecino
      for ( l=1 ; l<=conxvec ; l++ )
      { if(malla->getConx(veci,l)==i)   break; }
      //se busca la direccion en la cual el vecino se conecta con
        el volumen actual,
```

```
          if (! PropiedadesExternas ||UnSoloGamma) {//Agregado N-S
            u1=k(i)/(cp(i)*elems(i)->getDelta(j));
165         //conductancia del elemento actual (i) en dir j
            u2=k(veci)/(cp(veci)*elems(veci)->getDelta(l));
            //conductancia del elemento vecino en direccion l
            //Se puede simplificar los c\'alculos del m\'etodo SIMPLE
               si se utiliza UnSoloGamma si se tienen las mismas
             condiciones de frontera
          }//Agregado N-S
170       else {//Agregado N-S
            u1 = abs (Gamma(i) * elems(i)->GetDirSide(j))
              / elems(i)->getDelta(j);//Agregado N-S
            u2 = abs (Gamma(veci)*elems(veci)->GetDirSide(l))
              / elems(veci)->getDelta(l);//Agregado N-S
175       }//Agregado N-S //Version para Gamma diferente
          ut = 1.0 / ( (1.0/u1)+ (1.0/u2) );  //conductancia total
          aj = ut * elems(i)->getArea(j);
          matriz()( i,  i  ) += theta*aj;
          matriz()( i,veci) =  -theta*aj;
180       //asignacion de los coeficientes a la matriz
          b(i)-= (1-theta)*aj*(x(i)-x(veci)); //efecto tiempo pasado
          if (!tiempo_p->estacionario()) {
          Fourier(alph(i),tiempo_p->Delta(),
            elems(i)->getDelta(j)+elems(veci)->getDelta(l));
185       }
        }
        else {//analisis en la frontera
          for (h=1; h<=nbi; h++) //por cada frontera diferente
          {
190       if (malla->LadoFront(i,j,h,elems(i)())) {
              //si el volumen i en direccion j tiene el tipo de
               frontera h entonces
              boname = malla->obtNombreIndFront(h); //se obtiene el
               nombre del tipo de frontera
              if (PropiedadesExternas) {//Agregado N-S
                med = boname.despues(LecturaCadena);//Agregado N-S
195             if (med.contiene(',')) {//Agregado N-S
                  o =  med.Buscar(0,',');//Agregado N-S
                  boname =med.subCadena(0,o);//Agregado N-S
                }//Agregado N-S
                else {boname=med;}//Agregado N-S
200           }//Agregado N-S

              bovalue=boname.despues('=');
              //se obtiene la cadena despues del igual
              valor=atof(bovalue.carts());
```

```
            //se obtiene el valor numerico de la frontera
            if (boname.contiene("T") ||
              boname.contiene("Dirichlet")||
              boname.contiene("dirichlet") ) //tipo de frontera
            {
              if ( boname.contiene("h=") ) {
                //caso especial conveccion en la superficie
                bovalue=boname.despues("h=");
                hfrontera=bovalue.obtReal();
                bovalue=boname.despues("T=");
                valor=bovalue.obtReal();
                if (! PropiedadesExternas ||UnSoloGamma) {//
                 Agregado N-S
                  u1= k(i)/(cp(i)*elems(i)->getDelta(j));
                }//Agregado N-S
                else {//Agregado N-S
                u1 = abs (Gamma(i) * elems(i)->GetDirSide(j))
                  / elems(i)->getDelta(j);//Agregado N-S
                }//Agregado N-S
                u2= hfrontera/cp(i);
                ut=1.0 / ( (1.0/u1)+ (1.0/u2) );
              }
              else
              {
                if (! PropiedadesExternas ||UnSoloGamma) {//
                 Agregado N-S
                  ut = k(i)/(cp(i)*elems(i)->getDelta(j));
                }//Agregado N-S
                else {//Agregado N-S
                  ut = abs (Gamma(i) * elems(i)->GetDirSide(j))
                    / elems(i)->getDelta(j);//Agregado N-S
                }//Agregado N-S
              //se calcula la conductancia total que corresponde a
               la conductancia del elemento actual para este tipo
               de frontera
              }
              bc    = ut * elems(i)->getArea(j);
              b(i) += bc * valor ;
              b(i) -= (1-theta) * bc * x(i);
              //efecto pasado
              matriz()(i,i) += bc * theta ;
            }
            else {
              if (boname.contiene("Neumann")||
                boname.contiene("neumann") ) {
```

```
                    if (! PropiedadesExternas ||UnSoloGamma) {//
                     Agregado N-S
                      bc = valor * k(i) * elems(i)->getArea(j)/cp(i);
                    }//Agregado N-S
                    else {//Agregado N-S
250                 bc = valor *
                      abs (Gamma(i) * elems(i)->GetDirSide(j))
                      * elems(i)->getArea(j);//Agregado N-S//Agregado N
                       -S
                    }//Agregado N-S

255               b(i) += bc;
                  }//condicion de frontera dirichlet en la cual
                  //se da el valor de la derivada en direccion
                  //normal a  la superficie
                  if (boname.contiene("Q")) {
260               bc = valor * elems(i)->getArea(j)/cp(i);
                  b(i) += bc;
                  }//tipo especial de condicion de frontera
                  //dirichlet que solo es aplicada a la
                  //temperatura
265             }
              }
            }
          }//final condicional frontera o volumen vecino

270     } //final del for para cada lado
      };//final for para cada volumen
        return 1;
    }

275 bool DifusionTransitoria::SolucionUnTiempo(bool vector2campo)
    {
      bool solucion = false;
      int i;
      if (PropiedadesExternas)
280   {matriz()(nelementos,nelementos) +=matriz()(nelementos,nelementos
       );}

      if (primera_solucion ) {
        admlin().adjuntar(matriz(), x_futuro, b);primera_solucion=false
         ;
      }
285
      solucion = admlin().solve();
      int itera; bool conv;
```

```
  admlin().obtEstadisticas(itera, conv);
  //para los sistemas iterativos puede ser conveniente conocer el
   numero de iteraciones necesarias para resolver el sistema de
   ecuaciones
  std_o<<" Iteraciones= "<<itera<<" Tiempo= "<<
    admlin().obttiempoCPUsolve() <<"  " ;
  std_o<<"Minimo "<<x_futuro.minValor()<<" Maximo "
    <<x_futuro.maxValor()<<"\n";
  //Impresion en pantalla, para una clase general no es conveniente
    realizar la impresion en pantalla, se recomienda en este caso
   con fines educativos mostrar la informacion
  //if (conv==1) std_o<<"\nEl sistema si convergio";

  if (vector2campo)
    gdl->vector2campo(x_futuro,phi());
  return solucion;
}

void DifusionTransitoria::Reportarresultados()
{
  volcar(phi(),tiempo_p.obtPtr());
}

bool DifusionTransitoria::ResetCalculo()
{
  calculado    = false;
  solucionado   = false;
  primer_fourier  = true;
  tiempo_p->iniciarCicloTiempo();
  x.llenar(phi_inicial);
  x_futuro.llenar(phi_inicial);
  return true;
}

///METODOS AGREGADOS PARA//Agregado N-S/////////////////////
//Funcion para modificar el par\'ametro de control
void DifusionTransitoria::SetPropiedadesExternas ( bool opcion )
{ PropiedadesExternas = opcion; }
//Funcion para asignar unicamente el puntero de la malla
bool DifusionTransitoria::SetMalla ( Puntero<MallaFV> &MallaEntrada
  )
{   if (!PropiedadesExternas) return false;
  malla = MallaEntrada;
  return true;
}
//Funcion alternativa a Leer para inicializar las variables
```

```
bool DifusionTransitoria::SetMallaeIniciar ( Puntero<MallaFV> &
 MallaEntrada ,Cadena solucionador, Cadena def_tiempo,  Cadena
 nombreFrontera )
{
  if (!PropiedadesExternas) return false;
  //Se modifica el valor del parametro de control
  mallaExterna = true;
  //Se asigna el puntero de la malla
  malla  = MallaEntrada;
  //Se inicializa el puntero del tiempo
  tiempo_p.vincular(new prmTiempo);
  tiempo_p->Leer(def_tiempo);
  //Se llama a leer para inicializar otras variables
  Leer(" ",solucionador,"");
  //Se modifica el valor de la cadena que se va a leer para
   determinar las condiciones correctas de las frontera
  LecturaCadena = nombreFrontera;
  return true;
}
//Funcion para obtener el puntero de la malla
void DifusionTransitoria::GetMalla ( Puntero<MallaFV> &Salida )
{ Salida = malla; }
//Funcion para introducir el termino fuente de la ecuacion
bool DifusionTransitoria::SetFuente(Vector<real> S_f)
{ if (!PropiedadesExternas) return false;
  S_fuente_ext.chaSize(nelementos); S_fuente_ext = S_f;
  return true;
}
//Funcion para acceder al valor del termino fuente para cada VC
real DifusionTransitoria::S_fuente(int nvol){return S_fuente_ext(
 nvol);}
//Introduccion del vector solucion en el pasado y el punto de
 partida
bool DifusionTransitoria::SetPasadoPartida (Vector<real> phi_pasado
 , Vector<real> phi_pto_partida_iteracion)
{ if (!PropiedadesExternas)  { return false;  }
  x = phi_pasado ;
  x_futuro = phi_pto_partida_iteracion ;
  return true;
}
//Introduccion del vector solucion en el pasado.
bool DifusionTransitoria::SetPasado(Vector<real> phi_pasado)
{ if (!PropiedadesExternas)  { return false;  }
  x = phi_pasado ;
  return true;
}
```

```
//Modificar el valor de la conductividad, en este caso
//gamma=k y c_p=1.0
bool DifusionTransitoria::SetPropiedades ( Vector<real> G )
{ if (!PropiedadesExternas)  { return false;  }
  int i; rho.llenar(0.0); cp.llenar(1.0);
  k  = G;   UnSoloGamma = true;
  //Es importante modificar este parametro de control
  for (i=1;i<=nelementos;i++)
    alph(i)=k(i)/(rho(i)*cp(i));
  //Por defecto se recalcula alph, este valor solo es necesario Si
   se quiere verificar la estabilidad del m\'etodo
  return true;
}
//M\'etodo alternativo de asignar Gamma como un
//vector de vectores de posicion
bool DifusionTransitoria::SetPropiedades ( Vector<real> Gx, Vector<
 real> Gy, Vector<real> Gz)
{
  if (!PropiedadesExternas)  { return false;  }
  int i; rho.llenar(0.0);  cp.llenar(1.0);
  for ( i=1 ; i<=nelementos ; i++){
    Gamma(i)(1) = Gx(i); Gamma(i)(2) = Gy(i);
    if (malla->obtNoDimEspacio() == 3)
      Gamma(i)(3) = Gz(i);
//La asignacion en Z solo se realiza si la malla es tridimensional
  }
  return true;
}
//Funcion para obtener los parametros geometricos de la celda
Vector_basico<PunteroElmDefs> DifusionTransitoria::GetElmDefs()
{return elems;}
//Funcion para obtener el puntero del tiempo
Puntero<prmTiempo> DifusionTransitoria::GetTiempo()
{ return tiempo_p;}
//Funcion para modificar el nombre del campo que
//se visualiza en Paraview
void DifusionTransitoria::SetNombreCampo  (char *ncampo)
{ phi->ponNombreCampo(ncampo); }
//Calculo de los coeficientes para un tiempo especifico.
//Version PUBLIC. Aunque no es necesario devolver un valor
//para DifusionTransitoria::CalcularCoefUnTiempo
//por similitud con ConveccionTransitoria se devuelve este valor
Vector<real> DifusionTransitoria::CalcularCoefUnTiempo (real
 dependencia_futuro, Puntero<prmTiempo> t_ext, bool &triunfo)
{
  int i;  Vector<real> Salida;
```

```
  if (!PropiedadesExternas)
  {std_o<<"\nError ConveccionTransitoria::CalcularCoefUnTiempo\n\n"
   ;
  triunfo = false; return Salida; //Salida sin dimensionalizar
415  }
  //Asignacion de la dependencia del tiempo
  theta = dependencia_futuro; tiempo_p = t_ext;
  //Llamada a la funcion protected
  CalculoUnTiempo();
420  //Dimensionamiento de Salida
  Salida.chaSize(nelementos);
  for (i=1; i<=nelementos; i++) Salida(i) = matriz()(i,i);
  triunfo= true;
  return Salida;
425 }
//Version para el calculo en estado transitorio como ocurre
//con la ecuacion de correccion de la presion en SIMPLE
Vector<real> DifusionTransitoria::CalcularCoefUnTiempo (real
 dependencia_futuro, bool &triunfo)
{
430  int i; Vector<real> Salida;
  if (!PropiedadesExternas)
  {std_o<<"\nError DifusionTransitoria::CalcularCoefUnTiempo\n\n";
  triunfo = false; return Salida;
  }
435
  theta=dependencia_futuro;
  CalculoUnTiempo();
  Salida.chaSize(nelementos);
  for (i=1; i<=nelementos; i++) Salida(i) = matriz()(i,i);
440  triunfo= true;
  return Salida;
}
//Solucion de la ecuacion de difusion, el valor devuelto por la
 funcion
//es el resultado, triunfo es modificado para informar si se
 encontro
445 //la solucion
Vector<real> DifusionTransitoria::SolucionCoefUnTiempo (bool &
 triunfo, bool vector2campo )
{
  if (!PropiedadesExternas)
  {std_o<<"\nError ConveccionTransitoria::SolucionCoefUnTiempo\n\n"
   ;
450  triunfo = false; return x_futuro;
  }
```

```
  triunfo = SolucionUnTiempo(vector2campo);
  return x_futuro;
}
```

ANEXO J. CLASE PARA LA SOLUCIÓN DE LA ECUACIÓN DE CONVECCIÓN-DIFUSIÓN EN ESTADO TRANSITORIO: ENCABEZADO DE LA CLASE

```
//Archivo ConveccionTransitoria.h
//El comentario //Agregado N-S  se usa para indicar que la
 instruccion
//fue agregada para resolver las ecuaciones del flujo en la capa
 limite
#pragma once
#include <AdmonEcLin.h>
#include <MallaFV.h>
#include <ConstOLeemalla.h>
#include <GuardarEnsight.h>
#include <GradoLibertadFV.h>

class ConveccionTransitoria : protected GuardarEnsight
{
  int nelementos; // numero de volumenes de control
  //////////MALLA DE VOLUMENES FINITOS//////////////
  Puntero<MallaFV> malla;
  Vector_basico <PunteroElmDefs> elems;
  VecinoFVM  vecino;
  Puntero<CampoFVM>     phi;
  Puntero<CamposFVM>    campo_vel;
  Puntero<GradoLibertadFV> gdl;
  Vector<real>  u;
  Vector<real>  v;
  Vector<real>  w;//Agregado N-S
  Vector<real>  S_fuente_ext;//Agregado N-S
  //////////PROPIEDADES DEL MATERIAL///////////////
  Vector<real>  rho;
  Vector<real>  k;
  Vector<real>  cp;
  Vector<real>  alph;
  Vector_basico <Vector_xyz<real>> Gamma; //Agregado N-S
  ///////////SISTEMA//DE//ECUACIONES//////////////
  //Matriz<real>  matriz;
```

```
  Puntero<Matriz_Dispersa<real>>  matriz;
  EstDispersa disp;
  prm_Matriz <real> pm; //parametros de la matriz
  Vector<real>    b;
  Vector<real>    x;
  Vector<real>    x_futuro;
  ////SOLUCION//DEL//SISTEMA//DE//ECUACIONES//////
  Puntero<AdmonEcLin> admlin;
  /////////CALCULO//DEL//FOURIER////////////////
  real Fourier(real alpha, real dt, real dx);
  /////////DEPENDENCIA//DEL//TIEMPO////////////
  real theta;
  Puntero<prmTiempo> tiempo_p;
  ////TIPO//ESQUEMA//A//USAR////////////
  Cadena metodo_aprox;//Agregado N-S
  ///LECTURA//CADENA//COND//FRONTERA/////
  Cadena LecturaCadena;//Agregado N-S
  //////////PARAMETROS//DE//CONTROL////////////
  bool primer_peclet;
  bool calculado;
  bool leido;
  bool solucionado;
  bool PropiedadesExternas;//Agregado N-S
  bool mallaExterna;//Agregado N-S
  bool UnSoloGamma;//Agregado N-S
  bool primera_solucion;
  bool primer_fourier;
  real phi_inicial;
  real Fo_min ;
  real Fo_max ;
  real pe_min ;
  real pe_max ;
protected:
  virtual int  CalculoUnTiempo();
  virtual bool SolucionUnTiempo(bool vector2campo = true);
  virtual void Reportarresultados();
  void EstructurarDisp();
public:
  ConveccionTransitoria(void);
  ~ConveccionTransitoria(void){}
  int Leer(Cadena archivo,Cadena solucionador, Cadena def_tiempo,
   real T_inicial=1);
  int CalcularYResolver(real dependencia_futuro);
  bool ResetCalculo ( void);
  int GetNoVol(void) {return nelementos;}
  virtual real S_fuente (real x, real y);
```

```
 real funcion_a ( real pe, Cadena modo);
 real difusion(int vol1, int dir1, int vol2, int dir2, real area);
80 real difusion(int vol1, int dir1, real area);
 real difusion(int vol1, int dir1, real area, real conveccion_f);
 real conveccion( int vol1, int dir1 ,real area, Vector_xyz<real>
  Vel);
 real peclet_min(void);
 real peclet_max(void);
85 real peclet(real F, real D);

 ////////////////////M\'ETODOS //Agregado N-S  ///////////////
 void SetPropiedadesExternas ( bool opcion );
 bool SetMalla ( Puntero<MallaFV> &MallaEntrada  );
90 bool SetMallaeIniciar ( Puntero<MallaFV> &MallaEntrada ,  Cadena
  solucionador, Puntero<prmTiempo> &def_tiempo, Cadena );
 void GetMalla ( Puntero<MallaFV> &Salida);
 bool SetFuenteYFlujo (Vector<real> S_f, Vector<real> uu, Vector<
  real> vv, Vector<real> ww);
 bool SetPasadoPartida (Vector<real> phi_pasado, Vector<real>
  phi_pto_partida_iteracion) ;
 bool SetPasado (Vector<real> phi_pasado);
95 bool SetPropiedades ( Vector<real> set_rho , Vector<real> set_k ,
   Vector<real> set_cp);
 bool SetPropiedades ( Vector<real> set_rho , Vector<real>
  set_gamma);
 bool SetPropiedades ( Vector<real> set_rho , Vector<real> Gx,
  Vector<real> Gy, Vector<real> Gz);
 Puntero<prmTiempo> GetTiempo();
 void SetNombreCampo (char *ncampo);
100 inline real S_fuente (int nvol);
 Vector<real> CalcularCoefUnTiempo(real dependencia_futuro,
  Puntero<prmTiempo> t_ext , Cadena mAproximacion, bool &triunfo);
 Vector<real> GetCoefVecinos();
 Vector<real> SolucionCoefUnTiempo (bool &triunfo, bool
  vector2campo = false);
 bool CadenaFrontera ( Cadena cad);
105
};
```

ANEXO K. CLASE PARA LA SOLUCIÓN DE LA ECUACIÓN DE CONVECCIÓN-DIFUSIÓN EN ESTADO TRANSITORIO: ARCHIVO DE CÓDIGO FUENTE

```
//Archivo ConveccionTransitoria.cpp
//El comentario //Agregado N-S  se usa para indicar que la
 instruccion fue agregada para resolver las ecuaciones del flujo en
  la capa limite
#include "ConveccionTransitoria.h"

5 ConveccionTransitoria::ConveccionTransitoria(void)
{
  UnSoloGamma =  primera_solucion=primer_fourier=true;//Agregado N-
   S
  PropiedadesExternas=false;//Agregado N-S
  metodo_aprox = "UDS";//Agregado N-S
10  mallaExterna = false;//Agregado N-S
  tiempo_p.vincular(new prmTiempo);
}

int ConveccionTransitoria::Leer (Cadena archivo,Cadena solucionador
 , Cadena def_tiempo, real T_inicial)
15 {
  int i; int nd;//Agregado N-S
  static const char *mallador_tb[] = {"mallador", NULL};
  Cadena cadmalla;
  if (!mallaExterna) {//Agregado N-S
20    Is is(archivo);
    is->obtComando(cadmalla,mallador_tb);
    malla.vincular(new MallaFV());
    ConstOLeeMalla(malla(),cadmalla);
    vecino.iniciar(malla());
25    malla->CalcConectividad(vecino);
  }//Agregado N-S
  nelementos=malla->obtNoElem();
  //informacion geom\'etrica de cada volumen de control
  elems.chaSize(nelementos);
30  u.chaSize (nelementos);   v.chaSize (nelementos);
```

```
  w.chaSize (nelementos);w.llenar(0.0);//Agregado N-S
  for (int i=1; i<=nelementos;i++) {
    elems(i).refill( malla->obtTipoElem(i), malla->obtNoDimEspacio
     () );
    elems(i)->CalcGeomParameters( malla->obtCoorElem(i) );
    if (!PropiedadesExternas) {//Agregado N-S
      u(i) =  elems(i)->centroid(malla->obtCoorElem(i))(1);
      v(i) = -elems(i)->centroid(malla->obtCoorElem(i))(2);
    }//Agregado N-S
  }
  //Se leen las condiciones para el solucionador de ecuaciones
  Is solucion(solucionador);
  pm.Leer ( solucion );
  admlin.vincular(new AdmonEcLin(EXTERNAL_STORAGE));
  admlin().Leer(solucion);
  phi.vincular(new CampoFVM(malla(),"Temperatura"));//campo
  campo_vel.vincular(new CamposFVM(malla(),"Velocidad"));//campos
  gdl.vincular(new GradoLibertadFV(malla(),1));//grados de libertad
  if (!mallaExterna) tiempo_p->Leer(def_tiempo);//Agregado N-S
  k.chaSize(nelementos);  rho.chaSize(nelementos);//propiedades
  cp.chaSize(nelementos); alph.chaSize(nelementos);//del material
  Gamma.chaSize(nelementos);
  nd = malla->obtNoDimEspacio();
  for (i=1;i<=nelementos;i++) Gamma(i).chaSize(nd);
  if (!PropiedadesExternas) {//Agregado N-S
    k.llenar(0.01); rho.llenar(1.0); cp.llenar(1.0);
    for (i=1;i<=nelementos;i++)
       alph(i)=k(i)/(rho(i)*cp(i));
  }//Agregado N-S
  //matriz.chaSize(nelementos, nelementos);
  EstructurarDisp();
  b.chaSize(nelementos);
  x.chaSize(nelementos); x_futuro.chaSize(nelementos);//Propiedad
  if (! PropiedadesExternas) //Agregado N-S
    { x_futuro.llenar(T_inicial);}
  leido=true;
  return 1;
}
//Calculo del n\'umero discreto de Fourier
real ConveccionTransitoria::Fourier(real alpha, real dt, real dx)
{
real fo; fo = alpha*dt/(dx*dx);
if (primer_fourier) { Fo_min=Fo_max=fo; primer_fourier=false;}
if (fo < Fo_min) {Fo_min=fo;}
if (fo < Fo_max) {Fo_max=fo;}
return fo;
```

```
}
//Calculo y Solucion de la ecuaci\'on de conveccion-difusion
int ConveccionTransitoria::CalcularYResolver(real
 dependencia_futuro)
{
  if (!leido) return 0;
  tiempo_p->iniciarCicloTiempo();
  if (tiempo_p->estacionario()) {
    theta = 1.0;
    CalculoUnTiempo();
    SolucionUnTiempo();
    Reportarresultados();
  }
  else {
    theta=dependencia_futuro;
    gdl->vector2campo(x_futuro,phi());
    gdl->vector2campo(u,campo_vel()(1));
    gdl->vector2campo(v,campo_vel()(2));
    Reportarresultados();
    do  {
      tiempo_p->incrementarTiempo();
      CalculoUnTiempo();
      SolucionUnTiempo();
      Reportarresultados();
    }
    while(!tiempo_p->finalizo());
    if (Fo_min==Fo_max)
    { std_o<<"\nFourier "<<Fo_min<<"\n"; }
    else
    { std_o<<"\nMinimo Fourier "<<Fo_min<<
       "\nMaximo Fourier "<<Fo_max<<"\n";}
  }
  std_o<<"Peclet minimo "<< peclet_min() <<
    "Peclet maximo "<< peclet_max() << "\n";
  solucionado = true;
  return 1;
}

int ConveccionTransitoria::CalculoUnTiempo()
{
  if ( !PropiedadesExternas ) //Agregado N-S
    x=x_futuro;//se actualiza el valor del pasado
  matriz->llenar(0.0); b.llenar(0.0);
  real tr;//transitorio,
  real hfrontera;//conveccion en la frontera
  int m,n,o,p;//Agregado N-S
```

```
    Cadena fv1,fv2,fv3, med;//Agregado N-S
    Vector_xyz<real> minc, maxc; real px,py, valor2;//Agregado N-S
    int e, h, l; //contadores, direccion
    int veci; int conx;
125 Cadena boname, bovalue; //Nombre condiciones de frontera
    int nbi = malla->obtNoIndFront ();
    double valor;//valor de la condicion de frontera
    real F, D, Pe, a_e, a_bj, b_bj, A;
    //y posicion del centroide en el eje y
130 real area, y, volumen, d1, d2; //Agregado N-S
    Vector_xyz<real> vel(malla->obtNoDimEspacio()),
      pos(malla->obtNoDimEspacio());//Agregado N-S
    for(e=1;e<=nelementos;e++)  {//por cada volumen
      pos=elems(e)->centroid(malla->obtCoorElem(e));
135   volumen=elems(e)->getVolume();
      //calculo de la generacion de calor en funcion
      if (!PropiedadesExternas)//Agregado N-S
        {b(e) = S_fuente (pos(1), pos(2)) * volumen; }
      else{ b(e) = S_fuente (e) ; }//Agregado N-S
140   conx = elems(e)->GetNumOfConex(); //no. lados para VC actual
      //asignacion de coeficientes del efecto transitorio
      if (!tiempo_p->estacionario() ) {
        tr = rho(e) * volumen / tiempo_p->Delta() ;
        matriz()( e, e )  += tr ;
145     b(e)      += tr * x(e) ;
      }
      for( l=1 ; l<=conx ; l++ ) {//por cada conexion
        veci = malla->getConx(e,l);
        area = elems(e)->getArea(l);
150     if (veci!=0) {//si el vecino no es una frontera
          for ( h=1 ; h<=conx ; h++ )
          { if(malla->getConx(veci,h)==e)   break; }
          d1 = elems(e)->getDelta(l);
          d2 = elems(veci)->getDelta(h);
155       //se toma la velocidad promedio entre los VC
          vel(1) = ( u(e)*d2 + u(veci)*d1 )/(d1+d2);
          vel(2) = ( v(e)*d2 + v(veci)*d1 )/(d1+d2);
          if (malla->obtNoDimEspacio()==3)
            vel(3) = ( w(e)*d2 + w(veci)*d1 )/(d1+d2);//Agregado N-S
160       F  = conveccion(e,l,area,vel);
          D  = difusion( e, l, veci, h, area);
          Pe = peclet( F,D );
          A  = funcion_a(Pe, metodo_aprox); //Agregado N-S
          a_e = D * A + std::max( 0.0 , -F );
165       //Llenado matriz y vector de terminos independientes
          matriz()(e,e   ) +=   (a_e + F)  * theta ;
```

```
        matriz()(e,veci)   = -a_e * theta;
        b(e) += (1 - theta) * (-(a_e + F)*x(e) + a_e*x(veci)) ;
        if (!tiempo_p->estacionario()) {
170     Fourier(alph(e),tiempo_p->Delta(),
          elems(e)->getDelta(l) + elems(veci)->getDelta(h));
        }
      }
      else {//caso en el cual veci es cond. frontera
175     for (h=1; h<=nbi; h++) {
          if (  malla->LadoFront(e,l,h,elems(e)())) {
            if (!PropiedadesExternas) {//Agregado N-S
              vel(1) =  elems(e)->centroArea(malla->obtCoorElem(e),
               l)(1) ;
              vel(2) = -elems(e)->centroArea(malla->obtCoorElem(e),
               l)(2) ;
180         }//Agregado N-S
            else { //Agregado N-S
              vel.llenar(0.0);//Iniciacion por defecto
              //las velocidades en la frontera se calculan con las
               condiciones de frontera
              boname  = malla->obtNombreIndFront(h);
185           malla->obtCoorMinMax(minc, maxc);
              o = m=0; n=boname.tam();
              while (o<n ) { if (boname(o,1) == ',') m++; o++;}
              n=0;
              for (o=1; o<=m; o++) {
190             p=n;
                n= boname.Buscar(n,',');
                if (p!=n) {
                  med =boname.subCadena(p,n-p);
                  if (med.contiene("VelU"))  fv1 = med;
195               if (med.contiene("VelV"))  fv2 = med;
                  if (med.contiene("VelW"))  fv3 = med;
                }
                n++;
              }
200           p = boname.tam();
              med = boname.subCadena(n,p-n);
              if (med.contiene("VelU")) fv1 = med;
              if (med.contiene("VelV")) fv2 = med;
              if (med.contiene("VelW")) fv3 = med;
205           ///*********fronteras
              if (fv1.contiene("Dirichlet")){
                bovalue = fv1.despues('=');
                valor   = atof(bovalue.carts());
                vel(1) = valor;
```

```
          px = elems(e)->centroArea(malla->obtCoorElem(e),l)
           (1);
          py = elems(e)->centroArea(malla->obtCoorElem(e),l)
           (2);
          if (fv1.contiene("e^")) {
            bovalue = fv1.despues("e^"); valor2 = atof(
             bovalue.carts());
            vel(1) = valor * exp( valor2*(px-minc(1)) );
          }
          if (fv1.contiene("x^")){
            bovalue = fv1.despues("x^"); valor2 = atof(
             bovalue.carts());
            vel(1)  = valor * pow( px-minc(1) ,valor2);
          }
          if (fv1.contiene("rooty")){
            bovalue = fv1.despues("p=");valor2 = atof(bovalue
             .carts());
            if (py<valor2){
              vel(1) = valor * sqrt(py/valor2);
            }
          }
        }
        else
            if (fv1.contiene("Neumann") ) {
              bovalue = fv1.despues('=');
              valor   = atof(bovalue.carts());
              vel(1) = u(e) - valor* elems(e)->getDelta(l);
            }
        ///*********fronteras
        if (fv2.contiene("Dirichlet")) {
            bovalue = fv2.despues('=');
            valor   = atof(bovalue.carts());
            vel(2) = valor ;
            if (fv2.contiene("liny")){
              bovalue = fv1.despues("p=");valor2 = atof(
               bovalue.carts());
              if (py<valor2){
              vel(2) = valor * (py/valor2);
              }
            }
          }
          else
            if (fv2.contiene("Neumann") ) {
              bovalue = fv2.despues('=');
              valor   = atof(bovalue.carts());
              vel(2) = v(e) - valor* elems(e)->getDelta(l);
```

```
250                     }
                  ///*********fronteras
                  if (malla->obtNoDimEspacio() == 3)
                    if (elems(e)->GetDirSide(l)(3) !=0){
                      if (fv3.contiene("Dirichlet"))
255                   {bovalue = fv3.despues('=');
                      valor   = atof(bovalue.carts());
                      vel(3) = valor ;
                      }
                      else
260                     if (fv3.contiene("Neumann") )
                        {bovalue = fv3.despues('=');
                        valor   = atof(bovalue.carts());
                        vel(3) = w(e) - valor* elems(e)->getDelta(l);
                        }
265                 }
                  //Caso especial de la velocidad en la frontera para
                   la ec. de corr. de velocidad.
                  if (LecturaCadena.contiene("Vcor"))
                  {
                    vel(1) = u(e); vel(2) =v(e);
270                 if (malla->obtNoDimEspacio() == 3)
                      vel(3) = w(e);
                  }
                }
                ///*********fronteras
275             F      = conveccion(e,l,area,vel) ;
                boname  = malla->obtNombreIndFront(h); //Obtener la
                 cadena exacta del tipo y valor de condicion de
                 frontera
                if (PropiedadesExternas) {//Agregado N-S
                  med = boname.despues(LecturaCadena); //Agregado N-S
                  if (med.contiene(',')) {//Agregado N-S
280                 o =  med.Buscar(0,',');
                    boname =med.subCadena(0,o);
                  }//Agregado N-S
                  else {boname=med;}//Agregado N-S
                }//Agregado N-S
285             bovalue = boname.despues('=');
                valor   = atof(bovalue.carts());
                D      = difusion(e,l,area);
                if ( boname.contiene("T") || boname.contiene("dirichlet
                 ")
                  || boname.contiene("Dirichlet") ) {
290               if( boname.contiene("h=") ){
                    bovalue = boname.despues("T=");
```

```
             valor    = atof(bovalue.carts());
             bovalue = boname.despues("h=");
             hfrontera    = atof(bovalue.carts());
295          D = difusion(e,l,area,hfrontera);
            }
            if( boname.contiene("L") )
            {
             y      = elems(e)->centroArea(malla->obtCoorElem(e),
              l)(2);
300          valor = 1.0 + valor * (  (1-y) );
            }
            if (boname.contiene("e^")) {
             px = elems(e)->centroArea(malla->obtCoorElem(e),l)
              (1);
             bovalue = boname.despues("e^"); valor2 = atof(
              bovalue.carts());
305          valor = valor * exp( valor2*(px-minc(1)) );
           }
           if (boname.contiene("x^")){
             px = elems(e)->centroArea(malla->obtCoorElem(e),l)
              (1);
             bovalue = boname.despues("x^"); valor2 = atof(
              bovalue.carts());
310          valor = valor * pow( px-minc(1) ,valor2);
           }
           Pe     = peclet(F,D);
           A    = funcion_a(Pe,metodo_aprox);//Agregado N-S
           a_bj  =  D*A + std::max(F,0.0);
315        b_bj  = (D*A + std::max(-F,0.0))*valor;
         }
         if ( boname.contiene("Q") ) {
           a_bj  = F;
           b_bj  = valor * area/cp(e) ;
320      }
         if ( boname.contiene("Neumann")||boname.contiene("
          neumann") ) {
           a_bj  = F;
           if (!PropiedadesExternas || UnSoloGamma)
             { b_bj  = valor * area * k(e)/cp(e); }
325        else {
             b_bj = valor * area *
               abs(Gamma(e)*elems(e)->GetDirSide(l));
           }
         }
330      matriz() (e,e)  += a_bj * theta;
         b (e)      += b_bj - (1-theta)*a_bj*x(e) ;
```

```
            }
          }
        }
335     }
      }
      return 1;
    }
    bool ConveccionTransitoria::SolucionUnTiempo (bool vector2campo )
340 {
      int itera; bool conv, solucion;
      if (primera_solucion )
      { admlin().adjuntar(matriz(), x_futuro, b); primera_solucion=
       false; }
      solucion = admlin().solve();
345 
      admlin().obtEstadisticas(itera, conv);
      std_o<<" Iteraciones= "<<itera<<" Tiempo= "<<
        admlin().obttiempoCPUsolve() <<"  " ;
      std_o<<"Minimo "<<x_futuro.minValor()<<" Maximo "
350       <<x_futuro.maxValor()<<"\n";
      if (vector2campo) {//Agregado N-S
        gdl->vector2campo(x_futuro,phi());
        gdl->vector2campo(u,campo_vel()(1));
        gdl->vector2campo(v,campo_vel()(2));
355     if ( malla->obtNoDimEspacio()==3)
          gdl->vector2campo(x,campo_vel()(3));//Agregado N-S
      }//Agregado N-S
      return solucion;
    }
360 void ConveccionTransitoria::Reportarresultados() {
      volcar(phi(),tiempo_p.obtPtr());
      volcar(campo_vel(),tiempo_p.obtPtr());
    }

365 real ConveccionTransitoria::S_fuente(real x, real y) { return 0.0;
     }

    real ConveccionTransitoria::funcion_a ( real pe, Cadena modo)
    {
      if (modo.contiene("upwind") || modo.contiene("Upwind")
370     || modo.contiene("UDS"))
      { return 1.0; }
      if (modo.contiene("CDS"))
      { return 1.0-0.5*fabs(pe); }
      if ( modo.contiene("hybrid") || modo.contiene("hibrido") )
375   { return max(0.0,1.0-0.5*fabs(pe)); }
```

```
  if ( modo.contiene("power") || modo.contiene("law") )
  { return max(0.0,pow(1.0-0.1*fabs(pe),5)); }
  if ( modo.contiene("exp") || modo.contiene("Exp") )
  { return fabs(pe)/(exp(fabs(pe))-1.0); }
return 0;//Devolver este valor implica error
}

//se calcula la difusividad entre dos nodos vecinos
//vol1 numero correspondiente al volumen de control actual
//vol2 numero correspondiente al volumen de control vecino
//dir1 corresponde al numero del lado del volumen
//de control vol1 en la cual se realizan los calculos
real ConveccionTransitoria::difusion (int vol1, int dir1, int vol2,
  int dir2, real area)
{
  real D1, D2, Dt, gamma;
  if (!PropiedadesExternas || UnSoloGamma) {//Agregado N-S
    gamma = k(vol1)/cp(vol1) ; D1 = gamma / elems(vol1)->getDelta(
     dir1);
    gamma = k(vol2)/cp(vol2) ; D2 = gamma / elems(vol2)->getDelta(
     dir2);
  }//Agregado N-S
  else {//Agregado N-S
    gamma = abs (Gamma(vol1) * elems(vol1)->GetDirSide(dir1)) ;
    D1 = gamma / elems(vol1)->getDelta(dir1);
    gamma = abs (Gamma(vol2) * elems(vol2)->GetDirSide(dir2)) ;
    D2 = gamma / elems(vol2)->getDelta(dir2);
  }//Agregado N-S
  Dt    = area / (1.0/D1 + 1.0/D2);
  return Dt;
}
//version de la funcion de calculo de la difusividad
//para un lado que limita con una condicion de frontera
//en donde se define el coeficiente de conveccion
//esta funcion solo es validad cuando \phi=T
real ConveccionTransitoria::difusion (int vol1, int dir1, real area
 , real conveccion_f)
{
  real D1, D2, Dt, gamma;
  if (!PropiedadesExternas || UnSoloGamma) {//Agregado N-S
    gamma = k(vol1)/cp(vol1) ; D1 = gamma / elems(vol1)->getDelta(
     dir1);
  }
  else {//Agregado N-S
    gamma = abs (Gamma(vol1) * elems(vol1)->GetDirSide(dir1)) ;
    D1 = gamma / elems(vol1)->getDelta(dir1);
```

```
  }//Agregado N-S

  D2 = conveccion_f/cp(vol1);
420  Dt = area / (1.0/D1 + 1.0/D2);
  return Dt;
}
//version de la funcion de calculo de la difusividad
//para un lado que limita con una condicion de frontera
425 real ConveccionTransitoria::difusion(int vol1, int dir1, real area)
{
  real Dt, gamma  ;
  if (!PropiedadesExternas || UnSoloGamma) {//Agregado N-S
    gamma = k(vol1)/cp(vol1) ;
430  }
  else {//Agregado N-S
    gamma = abs (Gamma(vol1) * elems(vol1)->GetDirSide(dir1)) ;
  }
  Dt = gamma / elems( vol1  )->getDelta(dir1);
435  Dt = area *Dt;
  return Dt;
}
//Calculo del flujo convectivo
real ConveccionTransitoria::conveccion ( int vol1, int dir1 ,real
 area, Vector_xyz<real> Vel)
440 {
  real F ;
  Vector_xyz<real> side = elems(vol1)->GetDirSide(dir1);//Agregado
   N-S
  F = ( side * Vel);
  F = rho(vol1) * area * F ;//Agregado N-S
445  return F;
}
real ConveccionTransitoria::peclet_min(void) {return pe_min;}
real ConveccionTransitoria::peclet_max(void) {return pe_max;}
real ConveccionTransitoria::peclet(real F, real D)
450 {
  real peclet=F/D;
  if ( primer_peclet )
  { pe_max = pe_min = abs(peclet); primer_peclet=false; }
  if ( abs(peclet) > pe_max ) pe_max=abs(peclet);
455  if ( abs(peclet) < pe_min ) pe_min=abs(peclet);
  return peclet;
}
////////////////////METODOS //Agregado N-S //////////////////
void ConveccionTransitoria::SetPropiedadesExternas ( bool opcion )
460  {PropiedadesExternas=opcion;}
```

```
bool ConveccionTransitoria::SetMalla ( Puntero<MallaFV> &
 MallaEntrada )
{   if (!PropiedadesExternas) return false;
  malla=MallaEntrada;
  return true;
}

bool ConveccionTransitoria::SetMallaeIniciar ( Puntero<MallaFV> &
 MallaEntrada ,Cadena solucionador,  Puntero<prmTiempo> &def_tiempo
 , Cadena nombreFrontera )
{
  if (!PropiedadesExternas) return false;
  mallaExterna = true; malla   = MallaEntrada;
  tiempo_p = def_tiempo;
  Leer(" ",solucionador,"");
  LecturaCadena = nombreFrontera;
  return true;
}

void ConveccionTransitoria::GetMalla (Puntero<MallaFV> &Salida){
 Salida=malla; }
//Funcion para la introduccion del vector de termino fuente y
//el campo de la velocidad del fluido
bool ConveccionTransitoria::SetFuenteYFlujo (Vector<real> S_f,
 Vector<real> uu, Vector<real> vv, Vector<real> ww )
{ if (!PropiedadesExternas) return false;
  S_fuente_ext.chaSize(nelementos); S_fuente_ext = S_f;
  u = uu; v = vv;
  if (malla->obtNoDimEspacio() == 3)  w = ww;
  return true;
}
Puntero<prmTiempo> ConveccionTransitoria::GetTiempo()
  { return tiempo_p;}
void ConveccionTransitoria::SetNombreCampo  (char *ncampo)
    { phi->ponNombreCampo(ncampo); }
real ConveccionTransitoria::S_fuente (int nvol)
  {return S_fuente_ext(nvol);}
Vector<real> ConveccionTransitoria::CalcularCoefUnTiempo (real
 dependencia_futuro, Puntero<prmTiempo> t_ext, Cadena mAproximacion
 , bool &triunfo)
{
  int i;   Vector<real> Salida (nelementos);
  if (!PropiedadesExternas) {
    std_o<<"\nError ConveccionTransitoria::CalcularCoefUnTiempo\n\n
     ";
    triunfo = false;
```

```
    return Salida;
500 }
  theta = dependencia_futuro; tiempo_p = t_ext;
  metodo_aprox = mAproximacion;
  CalculoUnTiempo();
  for (i=1; i<=nelementos; i++) Salida(i) = matriz()(i,i);
505 triunfo= true;
  return Salida;
}
Vector<real> ConveccionTransitoria::GetCoefVecinos ()
{
510 int i, j, k, conx; Vector<real> Salida (nelementos);
  Salida.llenar(0.0);
  for (i=1; i<=nelementos; i++) {
    conx = elems(i)->GetNumOfConex();
    for ( j=1; j<=conx; j++) {
515     k = malla->getConx(i,j);
      if (k!=0) Salida(i) -= matriz()(i,k);
    }
  }
  return Salida;
520 }
//M\'etodo public para solucionar el sistema de ecuaciones
 externamente
Vector<real> ConveccionTransitoria::SolucionCoefUnTiempo ( bool &
 triunfo, bool vector2campo )
{
  if (!PropiedadesExternas) {
525   std_o<<"\nError ConveccionTransitoria::SolucionCoefUnTiempo\n\n
    ";
    triunfo = false;
    return x_futuro;
  }
  triunfo = SolucionUnTiempo(vector2campo);
530 return x_futuro;
}
//Introduccion del vector solucion en el pasado y el punto de
 partida
//para encontrar la solucion de los mismos iterativos
bool ConveccionTransitoria::SetPasadoPartida (Vector<real>
 phi_pasado, Vector<real> phi_pto_partida_iteracion)
535 {
  if (!PropiedadesExternas)  { return false;  }
  x = phi_pasado ;
  x_futuro = phi_pto_partida_iteracion ;
  return true;
```

```
540 }
    bool ConveccionTransitoria::SetPasado(Vector<real> phi_pasado)
    {
      if (!PropiedadesExternas)  { return false;  }
      x = phi_pasado ;
545   return true;
    }

    bool ConveccionTransitoria::SetPropiedades ( Vector<real> set_rho ,
      Vector<real> set_k , Vector<real> set_cp)
    {
550   int i;
      if (!PropiedadesExternas)  { return false;  }
      rho   = set_rho;   k = set_k;    cp   = set_cp;
      for (i=1;i<=nelementos;i++)
        alph(i)=k(i)/(rho(i)*cp(i));
555   return true;
    }
    bool ConveccionTransitoria::SetPropiedades ( Vector<real> set_rho ,
      Vector<real> set_gamma)
    {
      int i;
560   if (!PropiedadesExternas)  { return false;  }
      rho   = set_rho; k   = set_gamma; cp.llenar(1.0);
      for (i=1;i<=nelementos;i++)
        alph(i)=k(i)/(rho(i)*cp(i));
      return true;
565 }
    bool ConveccionTransitoria::SetPropiedades ( Vector<real> set_rho ,
      Vector<real> Gx, Vector<real> Gy, Vector<real> Gz)
    {
      int i; UnSoloGamma = false;
      if (!PropiedadesExternas)  { return false;  }
570   rho   = set_rho; cp.llenar(1.0);
      for (i=1; i<=nelementos ; i++)  {
        Gamma(i)(1)  = Gx(i);
        Gamma(i)(2)  = Gy(i);
        if (malla->obtNoDimEspacio() == 3)
575       Gamma(i)(3)  = Gz(i);
      }
      for (i=1;i<=nelementos;i++)
        alph(i)=Gx(i)/(rho(i)*cp(i));
      return true;
580 }
```

```
bool ConveccionTransitoria::CadenaFrontera ( Cadena cad)
{
  if (!PropiedadesExternas)  { return false;  }
  LecturaCadena = cad;
  return true;
}
```

ANEXO L. MODIFICACIONES A LA CLASE CONVECCIONTRANSITORIA PARA SU USO EN LA CLASE NAVIERSTOKES

Cuando se presentó el modo de resolver la ecuación de conveccion-difusión ya sea en estado estable o en estado transitorio se asumió que se conocía el campo de la velocidad del fluido, sin embargo, para resolver las ecuaciones de Navier-Stokes es ahora necesario calcular el campo de la velocidad tomando como punto de partida los valores en la iteración anterior o los valores de entrada.

Anteriormente, se utilizó un método para el cálculo del término fuente que dependía de la posición. Para el desarrollo del problema de la capa límite, es conveniente introducir el efecto de la generación en función del número que identifica a cada volumen de control. Para el caso de las ecuaciones de la cantidad de movimiento, el término de la generación representa las fuerzas viscosas y de presión.

Se hace necesario exportar el valor de algunos términos de la matriz de coeficientes[1] e introducirlos como parte del valor de la constante difusiva en la ecuación de corrreción de presión. Los términos de la matriz de coeficientes de cada una de las ecuaciones de la cantidad de movimiento son, en su mayor parte, iguales pero varían en caso de definir para algún lado más de un tipo de condición de frontera.

Se modifica la definición de la matriz de coeficientes a matriz dispersa para disminuir el tiempo requerido para resolver el sistema de ecuaciones por métodos iterativos, adicionalmente, esta modificación permite usar la descomposición incompleta LU como precondicionador reduciendo en la mayoría de los casos el número de iteraciones requerido.

Las variables que se agregan a la clase `ConveccionTransitoria` son:

```
....
class ConveccionTransitoria : protected GuardarEnsight
{
Vector<real>w;
Vector<real>S_fuente_ext;
Cadena metodo_aprox;
```

[1] Específicamente los términos de la diagonal principal de cada una de las instancias para resolver las ecuaciones de la cantidad de movimiento. Adicionalmente, el método SIMPLEC requiere de la sumatoria de los demás términos.

```
Cadena LecturaCadena;
bool PropiedadesExternas;
bool mallaExterna;
EstDispersa disp;
...
};
```

La variable que se modifica en la clase `ConveccionTransitoria` es:

```
....
class ConveccionTransitoria : protected GuardarEnsight
{
Puntero<Matriz_Dispersa<real>>matriz;
...
};
```

Se define la variable `w` para almacenar el valor de la velocidad del fluido para análisis en tres dimensiones. Se define el vector `S_fuente_ext` en el cual se almacena la información del término fuente para cada volumen de control. La `Cadena metodo_aprox` se define con el objetivo de almacenar el tipo de esquema y la `Cadena LecturaCadena` almacena el nombre de la condición de frontera que será leido para calcular la velocidad del fluido en la frontera o el valor de una propiedad ϕ en ella. Por otro lado, se definen `PropiedadesExternas` que modifica la forma como se calculan algunos valores y `mallaExterna` que modifica la forma de generar la malla.

Los métodos que se agregan son:

```
....
class ConveccionTransitoria : protected GuardarEnsight
{
...
void EstructurarDisp();
public:
...
void SetPropiedadesExternas ( bool opcion );
bool SetMalla ( Puntero<MallaFV> &MallaEntrada  );
bool SetMallaeIniciar ( Puntero<MallaFV> &MallaEntrada ,
 Cadena solucionador, Puntero<prmTiempo> &def_tiempo, Cadena );
void GetMalla ( Puntero<MallaFV> &Salida);
bool SetFuenteYFlujo
 (Vector<real> S_f, Vector<real> uu, Vector<real> vv, Vector<real> ww);
bool SetPasadoPartida
 (Vector<real> phi_pasado, Vector<real> phi_pto_partida_iteracion) ;
bool SetPasado (Vector<real> phi_pasado);
bool SetPropiedades
```

```
  ( Vector<real> set_rho , Vector<real> set_k , Vector<real> set_cp);
bool SetPropiedades ( Vector<real> set_rho , Vector<real> set_gamma );
Puntero<prmTiempo> GetTiempo();
void SetNombreCampo (char *ncampo);
inline real S_fuente (int nvol);
Vector<real> CalcularCoefUnTiempo (real dependencia_futuro,
 Puntero<prmTiempo> t_ext , Cadena mAproximacion, bool &triunfo);
Vector<real> GetCoefVecinos();
Vector<real> SolucionCoefUnTiempo
   (bool &triunfo, bool vector2campo = false);
bool CadenaFrontera ( Cadena cad);
};
```

A continuación se explican cada una de las funciones creadas:

1. Método `EstructurarDisp()`: las matrices dispersas se deben dimensionar de una forma diferente a las matrices densas, es por esto que se proporciona un método para realizar la inicialización.

   I. Primero se declaran varios contadores necesarios para el método, posteriormente se declaran dos vectores básicos para almacenar la información de cuales son los coeficientes a usar. Finalmente se declara un vector ordenado el cual se encargará de organizar la información.

   ```
   void ConveccionTransitoria::EstructurarDisp ()
   {
    int i,j,m,conx,veci;
    Vector_basico<int> irow(nelementos+1), jcol(nelementos*5);
    Vector_Ordenado<int> ord(5);
   ...
   ```

   II. El vector `irow` contiene la posición del primer elemento diferente de cero de cada fila, el vector `jcol` contiene la columna a la cual pertenece cada elemento de la matriz. El vector `irow` tiene una longitud $n+1$ siendo n el número de filas de la matriz, por otro lado, el vector `jcol` tiene una longitud de m siendo m el número de elementos diferentes de cero en la matriz. Se muestra un ejemplo de cada uno de estos vectores en la siguiente ecuación:

$$\begin{bmatrix} 1 & 1.1 & 0 & 0 \\ 4.1 & 5.2 & 5.6 & 0 \\ 0 & 4.3 & 2.3 & 4.5 \\ 0 & 0 & 1.2 & 5.4 \end{bmatrix}$$

$$irow = [1 \;\; 3 \;\; 6 \;\; 9 \;\; 11]$$

$$jcol = [1 \;\; 2 \;\; 1 \;\; 2 \;\; 3 \;\; 2 \;\; 3 \;\; 4 \;\; 3 \;\; 4]$$

III. Se realiza un primer bucle repetitivo que recorre cada volumen de control en donde se asigna al vector `irow` el valor del siguiente entero del contador de elementos `m`.

```
...
 jcol.llenar(0);
 m=0;
 for(i=1;i<=nelementos;i++) {
  irow(i) = m+1;
  conx=elems(i)->GetNumOfConex();
  ord.llenar(0);
...
```

IV. Para cada volumen de control se llena el vector `ord` con el número correspondiente a cada una de las celdas que intervienen en la ecuación del volumen *i*. Una vez se llena el vector, se ordena y se asigna a `jcol` los valores diferentes de cero.

```
...
  for ( j=1;j<=conx;j++){
   veci = malla->getConx(i,j);
   ord(j) = veci;
  }
  ord(conx+1)= i;
  ord.OrdPila();
  for ( j=1;j<=conx+1;j++){
   if (ord(j)!=0) {
    jcol(++m)=ord(j);
   }
  }
 }
...
```

V. El valor de los vectores `irow` y `jcol` se introduce en la instancia `disp` de tipo `EstDispersa` la cual contiene la información de la estructura de la matriz. Finalmente, se asigna la memoria al puntero `matriz` con la información recopilada en `disp`.

```
...
 irow(nelementos+1) = m+1;
 disp.chaSize(nelementos,m);
 for(i=1 ; i<=nelementos+1;i++) disp.irow(i) = irow(i);
 disp.chaSizeColJ(jcol,m);
 matriz.vincular(new Matriz_Dispersa<real>(disp));
}
```

2. Método `SetPropiedadesExternas(...)`: modifica el valor de `PropiedadesExternas` para permitir la introducción de valores como la velocidad, el término fuente, etc.

```
void ConveccionTransitoria::SetPropiedadesExternas ( bool opcion )
{PropiedadesExternas=opcion;}
```

3. Método `SetMalla(...)`: el procedimiento que se usará, genera una única malla la cual es llamada por las demás instancias de `ConveccionTransitoria` y `DifusionTransitoria` por lo que se hace necesario definir un método que permita introducir la malla para el problema especificado. En este método al igual que en otros, se verifica el valor del parámetro de control `PropiedadesExternas`, si este valor no ha sido establecido en `true`, no se realizará la asignación de la malla.

```
bool ConveccionTransitoria::SetMalla ( Puntero<MallaFV> &MallaEntrada )
{  if (!PropiedadesExternas) return false;
malla=MallaEntrada;  return true;
}
```

4. Método `SetMallaeIniciar()`: Se define un método alternativo a `Leer(...)` para introducir los datos del problema como la malla, la definición del tiempo y el nombre a buscar en la condición de frontera.

```
bool ConveccionTransitoria::SetMallaeIniciar
  ( Puntero<MallaFV> &MallaEntrada , Cadena solucionador,
  Puntero<prmTiempo> &def_tiempo, Cadena nombreFrontera )
{   if (!PropiedadesExternas) return false;
mallaExterna = true;  malla  = MallaEntrada;
tiempo_p = def_tiempo;  Leer(" ",solucionador,"");
LecturaCadena=nombreFrontera;
return true;
}
```

5. Método `GetMalla()`: método encargado de dar el valor del `Puntero` de la malla.

```
void ConveccionTransitoria::GetMalla ( Puntero<MallaFV> &Salida )
{ Salida = malla; }
```

6. Método `SetFuenteYFlujo(...)`: como se había mencionado, el valor del campo de la velocidad u, v y w debe ser introducido además del valor del término fuente. La asignación de la velocidad w solo se realiza si la malla usada es tridimensional.

```
bool ConveccionTransitoria::SetFuenteYFlujo (Vector<real> S_f,
  Vector<real> uu, Vector<real> vv, Vector<real> ww)
{ if (!PropiedadesExternas) return false;
S_fuente_ext.chaSize(nelementos); S_fuente_ext = S_f+;
u = uu; v = vv;
if (malla->obtNoDimEspacio() == 3)  w = ww;
return true;
}
```

7. Método `SetPasadoPartida(...)`: este método actualiza el valor de la variable a calcular y permite introducir un valor inicial para las iteraciones interiores.

```
bool ConveccionTransitoria::SetPasadoPartida
    (Vector<real> phi_pasado, Vector<real> phi_pto_partida_iteracion)
{
if (!PropiedadesExternas)  { return false; }
x = phi_pasado ;
x_futuro = phi_pto_partida_iteracion ;
return true;
}
```

8. Método `SetPasado(...)`: es necesario dar un punto de partida para resolver por métodos iterativos el sistema de ecuaciones lineales almacenado en la matriz de coeficientes y el vector de términos independientes; para lograr esto, se llama inicialmente `SetPasadoPartida(...)` pero, para un segundo ciclo, es suficiente con actualizar el valor de ϕ en el pasado[2]. Se define el método `SetPasado(...)` para actualizar el valor de ϕ en el pasado.

```
bool ConveccionTransitoria::SetPasado(Vector<real> phi_pasado)
{
if (!PropiedadesExternas)  { return false; }
x = phi_pasado ;
return true;
}
```

9. Método `SetPropiedades(...)`: se introduce el valor de ρ y de Γ correspondiente para el problema de convección-difusión; para la ecuación de la energía, $\Gamma = k/C_p$, para la ecuación de la cantidad de movimiento, $\Gamma = \mu$.

```
bool ConveccionTransitoria::SetPropiedades
    ( Vector<real> set_rho , Vector<real> set_k , Vector<real> set_cp)
{
int i;
if (!PropiedadesExternas)  { return false; }
rho  = set_rho;
k  = set_k;
cp  = set_cp;
for (i=1;i<=nelementos;i++)
alph(i)=k(i)/(rho(i)*cp(i));
return true;
}
```

[2]Se actualiza el valor de ϕ en el pasado si el análisis es transitorio.

```
bool ConveccionTransitoria::SetPropiedades
     ( Vector<real> set_rho , Vector<real> set_gamma)
{
int i;
if (!PropiedadesExternas)  { return false; }
rho  = set_rho;
k  = set_gamma;
cp.llenar(1.0);
for (i=1;i<=nelementos;i++)
alph(i)=k(i)/(rho(i)*cp(i));
return true;
}
```

10. Método `GetTiempo(...)`: se devuelve el `Puntero` al objeto `prmTiempo` de manera que solo es necesario crear un objeto que será llamado por las demás instancias de `ConveccionTransitoria` y `DifusionTransitoria`.

```
Puntero<prmTiempo> ConveccionTransitoria::GetTiempo()
 { return tiempo_p;}
```

11. Método `S_fuente(...)`: se proporciona una sobrecarga a la función `S_fuente` que toma como argumento el número que identifica al volumen de control.

```
real ConveccionTransitoria::S_fuente (int nvol)
{ return S_fuente_ext(nvol); }
```

12. Método `CalcularCoefUnTiempo(...)`: se define una función que, además de recibir varios parámetros y llamar a `CalculoUnTiempo()`, devuelve el valor de los elementos de la diagonal principal de la matriz de coeficientes. El valor del `Vector` devuelto es utilizado en conjunto con Ω y ρ para definir el valor de Γ en la ecuación de corrección de presión.

```
Vector<real> ConveccionTransitoria::CalcularCoefUnTiempo
  (real dependencia_futuro, Puntero<prmTiempo> t_ext,
     Cadena mAproximacion, bool &triunfo)
{
int i; Vector<real> Salida (nelementos);
if (!PropiedadesExternas) {
  std_o<<"\nError ConveccionTransitoria::CalcularCoefUnTiempo\n\n";
  triunfo = false;
  return Salida;
}
theta=dependencia_futuro;
tiempo_p = t_ext;
```

```
metodo_aprox = mAproximacion;
CalculoUnTiempo();
for (i=1; i<=nelementos; i++) Salida(i) = matriz()(i,i);
triunfo= true;
return Salida;
}
```

13. Método `SolucionCoefUnTiempo(...)`: se define un método que, además de resolver el sistema de ecuaciones lineales, devuelve el `Vector<real>` de la solución e indica en la variable `triunfo` si se encontró la solución.

```
Vector<real> ConveccionTransitoria::SolucionCoefUnTiempo
    (bool &triunfo,  bool vector2campo  )
{
 if (!PropiedadesExternas) {
  std_o<<"\nError ConveccionTransitoria::SolucionCoefUnTiempo\n\n";
  triunfo = false;
  return x_futuro;
 }
 triunfo = SolucionUnTiempo();
 return x_futuro;
}
```

14. Método `CadenaFrontera(...)`: el flujo de la propiedad ϕ en la frontera se determina a partir del tipo de frontera y el valor indicado en el archivo `''*.geom''`. Para seleccionar el valor correcto se debe indicar el nombre de la propiedad a leer[3] el cual se introduce en la clase mediante la función `CadenaFrontera(...)` que modifica el valor de la `Cadena LecturaCadena`.

```
bool ConveccionTransitoria::CadenaFrontera ( Cadena cad)
{
if (!PropiedadesExternas)  { return false; }
LecturaCadena = cad;
return true;
}
```

Se realizaron algunas modificaciones a funciones previamente definidas que permiten un uso más general para la clase.

1. Modificación al constructor: se da un valor inicial a los datos agregados a la clase.

[3] Por ejemplo, "VelU" para la velocidad en dirección x.

```
ConveccionTransitoria::ConveccionTransitoria(void)
{
primera_solucion=primer_fourier=true;
PropiedadesExternas=false;//Agregado N-S
metodo_aprox = "UDS";//Agregado N-S
mallaExterna = false;//Agregado N-S
tiempo_p.vincular(new prmTiempo);
}
```

2. Modificación a la función `Leer(...)`: se agregan algunos condicionales para restringir la ejecución de ciertas instrucciones.

```
int ConveccionTransitoria::Leer
 (Cadena archivo,Cadena solucionador, Cadena def_tiempo, real T_inicial)
{
...
```

I. La construcción de la malla solo se realiza si el parámetro `mallaExterna` es `false`.

```
if (!mallaExterna) {//Agregado N-S
   Is is(archivo);
   is->obtComando(cadmalla,mallador_tb);
   MallaFE malla_loc;
   ConstOLeeMalla(malla_loc,cadmalla);
   malla_loc.Escribir(Os("malla2.grid",NEWFILE));
   malla.vincular(new MallaFV());
   malla->Leer(Is("ARCH=malla2.grid"));
   vecino.iniciar(malla());
   malla->CalcConectividad(vecino);
   }//Agregado N-S
   ...
```

II. Se dimensiona e inicializa el vector que contiene la información de la velocidad w.

```
w.chaSize (nelementos);w.llenar(0.0);//Agregado N-S
```

III. La asignación de las componentes de la velocidad: no se realiza en la función `Leer(...)` si se definió a `PropiedadesExternas` como `true`.

```
for (int i=1; i<=nelementos;i++)
{
   ...
   if (!PropiedadesExternas) {//Agregado N-S
      u(i) =  elems(i)->centroid(malla->obtCoorElem(i))(1);
      v(i) = -elems(i)->centroid(malla->obtCoorElem(i))(2);
      }//Agregado N-S
```

```
}
...
```

IV. Como ya se había mencionado, la función `SetMallaeIniciar(...)` también define los parámetros del tiempo, es por esto que si la malla es externa, no se debe leer los datos del tiempo.

```
if (!mallaExterna)//Agregado N-S
     tiempo_p->Leer(def_tiempo);
...
```

V. Si las propiedades son externas, no es necesario asignar un valor a `k`, `rho`, `cp` y `x_futuro`. El dimensionamiento de la matriz de coeficientes no se realiza con la llamada a `chasize(...)`, en cambio, se llama al método `EstructurarDisp()` para realizar esta labor.

```
if (!PropiedadesExternas) {//Agregado N-S
   k.llenar(0.01);
   rho.llenar(1.0);
   cp.llenar(1.0);
   for (i=1;i<=nelementos;i++)
       alph(i)=k(i)/(rho(i)*cp(i));
}//Agregado N-S
//matriz->chaSize(nelementos, nelementos);
EstructurarDisp();//Agregado N-S
b.chaSize(nelementos);
x.chaSize(nelementos); x_futuro.chaSize(nelementos);//Propiedad
if (! PropiedadesExternas) {//Agregado N-S
   x_futuro.llenar(T_inicial);
}
...
return 1;
}
```

3. Modificación a la función `CalculoUnTiempo()`: se agregan varias instrucciones entre las que se destaca la lectura de la condición de frontera en cada lado.

I. La actualización del valor de la propiedad en el pasado no se realiza en la función `CalculoUnTiempo()` si `PropiedadesExternas` ha sido definido como `true`, en cambio, se debe llamar al método `SetPasadoPartida(...)` o `SetPasado(...)` según sea el caso.

```
int ConveccionTransitoria::CalculoUnTiempo()
{
if ( !PropiedadesExternas ) //Agregado N-S
      x=x_futuro;
```

...

II. Se definen otras variables que serán necesarias para el cálculo del tipo y valor de la condición de frontera:

```
int m,n,o,p;//Agregado N-S
Cadena fv1,fv2,fv3, med;//Agregado N-S
Vector_xyz<real> minc, maxc; real px,py, valor2;//Agregado N-S
real area, y, volumen, d1, d2; //Agregado N-S
...
```

III. El modo de asignar el efecto del término fuente difiere si las propiedades son externas o internas. Para el caso que `PropiedadesExternas` sea `true`, se llama a la sobrecarga `S_fuente(int nvol)`, en caso contrario, se introduce las coordenadas del volumen de control actual.

```
for(e=1;e<=nelementos;e++) {
pos=elems(e)->centroid(malla->obtCoorElem(e));
volumen=elems(e)->getVolume();
if (!PropiedadesExternas) {//Agregado N-S
b(e) = S_fuente (pos(1), pos(2)) * volumen;
}//Agregado N-S
else {//Agregado N-S
b(e) = S_fuente (e) ; //Agregado N-S
}//Agregado N-S
...
```

IV. Determinación de la velocidad en la frontera: se debe realizar el llenado de `vel` (de tipo `Vector_xyz<real>`) de acuerdo a las condiciones de frontera especificadas en el archivo `''*.geom''`,el procedimiento para realizar el llenado se muestra en el siguiente fragmento de código.

```
for( l=1 ; l<=conx ; l++ ) {
 ...
 if (veci!=0)
 {...}
 else
   {
    for (h=1; h<=nbi; h++)
    {
     if (  malla->LadoFront(e,l,h,elems(e)())   )
     {

      if (!PropiedadesExternas) {//Agregado N-S
       vel(1) =  elems(e)->centroArea(malla->obtCoorElem(e),l)(1) ;
       vel(2) = -elems(e)->centroArea(malla->obtCoorElem(e),l)(2) ;
```

```
}//Agregado N-S
else
{
 vel.llenar(0.0);
 boname  = malla->obtNombreIndFront(h);
 o = m=0; n=boname.tam();
 while (o<n )
  { if (boname(o,1) == ',') m++; o++;}
 n=0;
 for (o=1; o<=m; o++) {
  p=n;
  n= boname.Buscar(n,',');
  if (p!=n) {
   med =boname.subCadena(p,n-p);
  if (med.contiene("VelU"))
   fv1 = med;
  if (med.contiene("VelV"))
   fv2 = med;
  if (med.contiene("VelW"))
   fv3 = med;
  }
  n++;
 }
 p = boname.tam();
 med = boname.subCadena(n,p-n);
 if (med.contiene("VelU"))
  fv1 = med;
 if (med.contiene("VelV"))
  fv2 = med;
 if (med.contiene("VelW"))
  fv3 = med;
 ///*********fronteras
  if (fv1.contiene("Dirichlet")) {
  bovalue = fv1.despues('=');
  valor   = atof(bovalue.carts());
  vel(1) = valor;
  px = elems(e)->centroArea(malla->obtCoorElem(e),1)(1);
  py = elems(e)->centroArea(malla->obtCoorElem(e),1)(2);
  if (fv1.contiene("e^")) {
   bovalue = fv1.despues("e^"); valor2 = atof(bovalue.carts());
   vel(1) = valor * exp( valor2*(px-minc(1)) );
  }
  if (fv1.contiene("x^")){
```

```
  bovalue = fv1.despues("X^"); valor2 = atof(bovalue.carts());
  vel(1)  = valor * pow( px-minc(1) ,valor2);
 }
 if (fv1.contiene("rooty")){
  bovalue = fv1.despues("p=");valor2 = atof(bovalue.carts());
  if (py<valor2){
   vel(1) = valor * sqrt(py/valor2);
  }
 }
}
 else
  if (fv1.contiene("Neumann") ) {
   bovalue = fv1.despues('=');
   valor   = atof(bovalue.carts());
   vel(1) = u(e) - valor* elems(e)->GetDirSide(l)(1);
  }
///*********fronteras
 if (fv2.contiene("Dirichlet")) {
  bovalue = fv2.despues('=');
  valor   = atof(bovalue.carts());
  vel(2) = valor ;
 if (fv2.contiene("liny")){
   bovalue = fv1.despues("p=");valor2 = atof(bovalue.carts());
   if (py<valor2){
   vel(1) = valor * (py/valor2);
  }
 }
 }
 else
  if (fv2.contiene("Neumann") ) {
   bovalue = fv2.despues('=');
   valor   = atof(bovalue.carts());
   vel(2) = v(e) - valor* elems(e)->GetDirSide(l)(2);
  }
///*********fronteras
if (malla->obtNoDimEspacio() == 3)
  if (fv3.contiene("Dirichlet"))
  {bovalue = fv3.despues('=');
  valor   = atof(bovalue.carts());
  vel(3) = valor ;
  }
  else
   if (fv3.contiene("Neumann") )
```

```
            {bovalue = fv3.despues('=');
            valor   = atof(bovalue.carts());
            vel(3) = w(e) - valor* elems(e)->GetDirSide(l)(3);
            }
        }
        ///*********fronteras
...
```

V. Además de determinar el valor de la velocidad en la frontera, también se debe determinar el valor de la propiedad (ya sea la temperatura o una componente de la velocidad) en la frontera, el siguiente procedimiento realiza esta tarea.

```
...
F      = conveccion(e,l,area,vel) ;
boname  = malla->obtNombreIndFront(h);
if (PropiedadesExternas){//Agregado N-S
 med = boname.despues(LecturaCadena);
 if (med.contiene(',')){//Agregado N-S
  o =  med.Buscar(0,',');
  boname =med.subCadena(0,o);
 }//Agregado N-S
else//Agregado N-S
{boname=med;}//Agregado N-S
}//Agregado N-S
...
```

4. Modificación del método `SolucionUnTiempo()`: se redefine la función de manera que acepte un parámetro (`vector2campo`) el cual será verificado para realizar o no la transformación de los vectores de velocidad y la propiedad hacia los correspondientes campos.

```
bool ConveccionTransitoria::SolucionUnTiempo( bool vector2campo )
{
...
if (vector2campo) {
gdl->vector2campo(x_futuro,phi());
gdl->vector2campo(u,campo_vel()(1));
gdl->vector2campo(v,campo_vel()(2));
if ( malla->obtNoDimEspacio()==3)
  gdl->vector2campo(x,campo_vel()(3));//Agregado N-S
}
return solucion;
}
```

Para un correcto funcionamiento, se debe modificar la declaración en el encabezado de la clase por la siguiente instrucción.

```
virtual bool SolucionUnTiempo(bool vector2campo = true);
```

Los métodos creados y modificados para `DifusionTransitoria` son muy similares a los métodos recién presentados para la clase `ConveccionTransitoria`, por esta razón, no se presentan. Se puede encontrar el código fuente con las modificaciones realizadas en los anexos F, G, H, I.

ANEXO M. CLASE PARA EL CÁLCULO DEL FLUJO EN LA CAPA LÍMITE: ENCABEZADO DE LA CLASE

```
//NavierStokes.h
#pragma once
#include "ConveccionTransitoria.h"
#include "DifusionTransitoria.h"
//metodo de aproximacion para el metodo de conveccion difusion
static const char *MACD[] = {"UDS", "Upwind","CDS", "upwind","
 hybrid",
 "hibrido", "power", "law", "exp", "Exp", NULL};
//metodo de solucion de las ecuaciones de navier stokes
static const char *MSNS[] = {"SIMPLE","NOSIMPLE", "SIMPLEC", NULL};
//Calculo del tipo de problema especificado
const char Separador = ',';
const bool MULT_POR_RHO_ON = true;
const bool MULT_POR_RHO_OFF = false;
class NavierStokes :  protected GuardarEnsight
{
 //Clase para la solucion de la ec. de cant. movimiento.
 ConveccionTransitoria ConvVu, ConvVv, ConvVw;
 //Velocidad en el tiempo futuro
 Vector<real> VelU, VelV, VelW;
 //Velocidad en el tiempo presente
 Vector<real> VelU_ant, VelV_ant, VelW_ant;
 //Velocidad calculada en cada ec. de cant. de mov.
 Vector<real> VmAstU, VmAstV, VmAstW;
 //Gradientes para cada componente de la velocidad, presion, etc
 Vector_basico<Vector_xyz<real>> GU, GV, GW, GP, Gtau;
 Vector_basico<Vector_xyz<real>> GU_n, GV_n, GW_n, GP_n;
 Vector_basico<Vector_xyz<real>> GU_ant, GV_ant, GW_ant, GP_ant;
 Vector_basico<Vector_xyz<real>> Fcalor;
 Vector<real> GVel, GVel_ant,GVel_n ;
 //Tensor de esfuerzos
 Vector_basico<Matriz<real>> Tau_n, Tau_ant, Tau;
 Vector<real> CoefU, CoefV, CoefW;//En estos se almacenar\'an los
  coeficientes de la diagonal principal que son  necesarios en la
  ecuacion de correcion de presion
 //Clase para la solucion de la ec. de la energia
```

```
ConveccionTransitoria Temperatura;
//Almacenamiento de  la temperatura actual y anterior.
Vector<real> Temp, Temp_ant;
//Clase para la solucion de la ec. de correcion de presion
DifusionTransitoria  Pprima;
//Almacenamiento de la correcion de presion
Vector<real> Presionprima;
//Presion para varios tiempo
Vector<real> Presion, Presion_ant, Presion_n;
//Correcion de la velocidad.
ConveccionTransitoria Corrv;
Vector<real> vcor;
//Vector para almacenar el volumen de cada VC
Vector<real> Volumen;
Vector<real> Fuente;//La variable Fuente se usa para  almacenar la
  informaci\'on del t\'ermino fuente de  cualquiera  de las
 ecuaciones
Vector<real> ResiduoMasico;
//////////MALLA DE VOLUMENES FINITOS/////////////
Puntero<MallaFV> malla;
Vector_basico <PunteroElmDefs> elems;
Puntero<CampoFVM>     campo_T;
Puntero<CampoFVM>     campo_u;
Puntero<CampoFVM>   campo_residuo;
Puntero<CampoFVM>   campo_pres;
Puntero<CampoFVM>   campo_pres_prima;
Puntero<CampoFVM>   campo_corr_vel;
Puntero<CamposFVM>    campo_vel;
Puntero<CamposFVM>    campo_GTau ;
Puntero<CamposFVM>    campo_GU, campo_GV, campo_GW;
Puntero<CamposFVM>   campo_GP;
Puntero<CamposFVM>   campo_Q;
Puntero<GradoLibertadFV> gdl;
//////////PROPIEDADES DEL MATERIAL//////////
Vector<real> rho;
Vector<real> k;
Vector<real> cp;
Vector<real> mu;
real Rgas;//Constante del gas, para cuando se considere rho
 variable
/////////DEPENDENCIA//DEL//TIEMPO//////////
real theta;
Puntero<prmTiempo> tiempo_p;
Puntero<prmTiempo> t_aux;
//////NUMERO//ITERACIONES//MAXIMAS/////////
int Nitera;
```

```
 /////////CRITERIO//PARADA/////////////////
 real MAssThreshold;//umbral del residuo masico
 ////////METODO//SOLUCION///////////////////
80 Cadena MetodoSolveNS;
 ////TIPO//ESQUEMA//A//USAR/////////////////
 Cadena MetodoAprox;//Agregado N-S
 ////////////FACTORES DE LA MALLA/////////////
 int nvol; // numero de volumenes de control
85 int ndim; //numero de dimensiones
 /////////REFERENCIA DE LA PRESION///////////
 real minP, maxP;
 //////////PARAMETROS///////////////////////
 real Subrelajacion;//Parametro de subrelajacion
90 ///////////////////DENSIDAD/////////////////
 real TrefparaDensidad;//Temperatura de referencia
 //para calcular la densidad del gas
 real Patm;//Presion atmosferica del analisis
 //////////PARAMETROS//DE//CONTROL////////////
95 bool densidad_kte;
 bool leido;
 bool solucionado;
 bool CargarDatos;
 bool minPres, maxPres;
100 bool ResEstable;//Resultados en estado estable
 /////CALCULO//DE//LA//DENSIDAD//EN//GASES
 real DensidadfPyT (int nvol);//Densidad en funci\'on de la
 //Presion y la Temperatura
 void Condiciones_iniciales();
105 void DimensionarVectoresMatrices();
 void CalcPresRef();
protected:
 virtual int CalculoUnCiclo();
 bool PrimerPasoSIMPLE();
110 bool PrimerPasoNOSIMPLE();
 bool SegundoPasoSIMPLE();
 bool SegundoPasoNOSIMPLE();
 bool SegundoPasoSIMPLEC();
 bool TercerPasoSIMPLE();
115 bool TercerPasoNOSIMPLE();
 bool TercerPasoSIMPLEC();
 bool CuartoPasoSIMPLE(bool forzar = false);
 bool CargadoDeDatos();
 bool CalcularTauyGVs();
120 bool CalcularPyG ();
 bool CalcQcalor();
 bool ActualizarPyTau ();
```

```
 virtual int SolucionUnTiempo();
 virtual void Reportarresultados();
 bool LeerFronteras(Cadena TomarFrontera,
  Vector_basico<Cadena> &fr1);
 bool LeerFronteras(Vector_basico<Cadena> &,
  Vector_basico<Cadena> &, Vector_basico<Cadena> &);
 real LeerFrontEsp(int i, int j, real muestra, Cadena fr );
public:
 NavierStokes(void);
 ~NavierStokes(void){}
 int Leer(Cadena archivo,Cadena solucionador,
  Cadena def_tiempo, real dep_futuro=1.0 ,
  Cadena solucionadorTemp = "Igual");
 bool Calcular ();

 //////FUNCIONES DE CALCULO DE DERIVADAS////
 bool NablaEsc ( Vector<real> &Entrada, Vector_basico< Vector_xyz <
  real>> &Salida, Cadena Frontera = "Ninguna");
 bool NablaDotVec ( Vector_basico< Vector_xyz <real>> &Entrada,
  Vector<real> &Salida, Cadena Frontera = "Ninguna");
 bool NablaDotVecEspecial ( Vector_basico< Vector_xyz <real>> &
  Entrada, Vector<real> &Salida, bool MultRho = false);
 bool TensorPorVector_xyz (Matriz<real> Me,  Vector_xyz<real> Ve,
  Vector_xyz<real> &Salida);
 bool NablaPorMatriz ( Vector_basico< Matriz <real>> &Entrada,
  Vector_basico<Vector_xyz<real>> &Salida);
 Matriz<real> PromMatriz(Matriz<real> E1, real , Matriz<real> E2,
  real );


 bool GuardarDatos();
 bool GuardarDatos(real minvel,real maxvel);
 bool SetCargarDatos(bool opcion);
 void SetNoIterExt(int set);
 bool SetTipoMetodoSolucion(Cadena tipo);
 bool SetEsquemaConveccion(Cadena tipo);
 void SetMinimaPresion(real min);
 void SetMaximaPresion(real max);
 void SetTrefyDensidad(real tt, bool densidad_constante);
 void SetPresAtmosf(real pp);
 void ResetMinMaxPresion ();
 bool SetParametros (bool Datos_iniciales_externos = false,  int
  N_iteraciones = 100, Cadena Esquema_conveccion = "UDS" ,
  Cadena Esquema_solucion="SIMPLE",bool Resultados_indep_tiempo=
   true );
 bool SetMAssThreshold(real MaxResMas);
```

```
 void SubRelajaSIMPLE(real);
};//Agregado N-S
```

ANEXO N. CLASE PARA EL CÁLCULO DEL FLUJO EN LA CAPA LÍMITE: ARCHIVO DE CÓDIGO FUENTE.

```
//Archivo NavierStokes.cpp
#include "NavierStokes.h"
NavierStokes::NavierStokes(void)
{
  leido = solucionado = CargarDatos = minPres = maxPres = false;
  ResEstable = densidad_kte = true;
  Nitera = 1000;  MAssThreshold = 1e-10;
  MetodoSolveNS = "SIMPLE"; MetodoAprox = "UDS";
  Rgas = 0.2870;
  TrefparaDensidad = 273.0; Patm = 101.325;
  t_aux.vincular(new prmTiempo()); t_aux->Leer("dt=0");
}
int NavierStokes::Leer(Cadena archivo,Cadena solucionador,
  Cadena def_tiempo, real dep_futuro , Cadena solucionadorTemp)
{
  int i;
  //Se inicia cada objeto de ConveccionTransitoria para resolver
   cada una de las ecuaciones de cantidad de movimiento
  std_o<<"\n::::::::Malla Velocidad U:::::::::\n";
  ConvVu.SetPropiedadesExternas(true);  //Se definen las
   propiedades externas para permitir la introduccion del campo de
   velocidad, propiedades, etc...
  ConvVu.Leer (archivo ,solucionador,def_tiempo,0);
  ConvVu.CadenaFrontera("VelU");
  ConvVu.GetMalla(malla);//Se guarda la malla creada
  ndim = malla->obtNoDimEspacio(); //numero dimensiones
  nvol = malla->obtNoElem(); //numero volumenes
  tiempo_p = ConvVu.GetTiempo(); //se lee el tiempo
  std_o<<"\n:::::::::Malla Velocidad V::::::::\n";
  ConvVv.SetPropiedadesExternas(true);//se toma el puntero de la
   malla creada y se pasa a la malla de los dem\'as campos
  ConvVv.SetMallaeIniciar(malla,solucionador,tiempo_p,"VelV");
  if (ndim==3)
  { std_o<<"\n:::::::::Malla Velocidad W::::::::\n";
    ConvVw.SetPropiedadesExternas(true);
    ConvVw.SetMallaeIniciar(malla,solucionador,tiempo_p, "VelW");
```

```
    }
    std_o<<"\n:::::::::Malla Correcion Presion:::::::::\n";
35  Pprima.SetPropiedadesExternas(true);
    Pprima.SetMallaeIniciar(malla,solucionador,"dt=0", "P");
    std_o<<"\n:::::::::Malla Correccion v:::::::::\n";
    Corrv.SetPropiedadesExternas(true);
    Corrv.SetMallaeIniciar(malla,solucionador,t_aux,"Vcor");
40  std_o<<"\n:::::::::Malla Energia:::::::::\n";
    Temperatura.SetPropiedadesExternas(true);
    if (solucionadorTemp.contiene("Igual"))
      solucionadorTemp = solucionador;
    Temperatura.SetMallaeIniciar(malla,solucionadorTemp,tiempo_p,"
     Temp");
45  std_o<<"\n:::::::::FIN LECTURA MALLAS:::::::::\n";
    //Se calcula la conectividad con ayuda de una instancia de
     VecinoFVM Se inicia cada uno de los punteros de campo
    campo_T.vincular(new CampoFVM(malla(),"Temperatura"));
    campo_pres.vincular(new CampoFVM(malla(),"Presion"));
    campo_pres_prima.vincular(new CampoFVM(malla(),"CorreccionPresion
     "));
50  campo_corr_vel.vincular(new CampoFVM(malla(),"CorrVelV"));
    campo_vel.vincular(new CamposFVM(malla(),"Velocidad"));
    campo_u.vincular (new CampoFVM(malla(),"Velocidad_U"));
    campo_residuo.vincular (new CampoFVM(malla(),"Residuo_Masico"));
    campo_GTau.vincular (new CamposFVM(malla(),"GradTau"));
55  campo_GU.vincular (new CamposFVM(malla(),"GU"));
    campo_GV.vincular (new CamposFVM(malla(),"GV"));
    campo_GW.vincular (new CamposFVM(malla(),"GW"));
    campo_GP.vincular (new CamposFVM(malla(),"GP"));
    campo_Q.vincular (new CamposFVM(malla(),"Qcond"));
60  //Se inicia el objeto de tipo Puntero<GradoLibertadFV> con un
     solo grado de libertad por cada volumen de control
    gdl.vincular(new GradoLibertadFV(malla(),1));
    Is iss("ARCH=Informacion_grafica.txt");
    GuardarEnsight::Leer(iss, ndim);
    GuardarEnsight::ponModoGuardar(BINARY,BINARY);
65  theta = dep_futuro; //Dependencia futuro
    DimensionarVectoresMatrices();//DIMENSIONAMIENTO DE VECTORES
    for (i=1; i<=nvol;i++)
    {
      elems(i).refill( malla->obtTipoElem(i), malla->obtNoDimEspacio
       () );
70    elems(i)->CalcGeomParameters( malla->obtCoorElem(i) );
    }
    for (i=1; i<=nvol; i++) Volumen(i) = elems(i)->getVolume();
    //LECTURA DE LOS VALORES INICIALES PARA EL PROBLEMA
```

```
  Condiciones_iniciales();
  leido = true;//PARAMETRO DE CONTROL
  return 1;
}//Agregado N-S
//Funcion encargada de dimensionar todos los vectores
void NavierStokes::DimensionarVectoresMatrices()
{
  int i;
  CoefU.chaSize(nvol);  CoefV.chaSize(nvol);  CoefW.chaSize(nvol);
  VmAstU.chaSize(nvol); VmAstV.chaSize(nvol); VmAstW.chaSize(nvol);
  VelU.chaSize(nvol); VelU_ant.chaSize(nvol);
  VelV.chaSize(nvol); VelV_ant.chaSize(nvol);
  if (ndim==3) { VelW.chaSize(nvol); VelW_ant.chaSize(nvol); }
  GVel.chaSize(nvol); GVel_ant.chaSize(nvol); GVel_n.chaSize(nvol);
  Presion_ant.chaSize(nvol); Presion.chaSize(nvol);
  Presion_n.chaSize(nvol); Presionprima.chaSize(nvol);
  Fuente.chaSize(nvol); ResiduoMasico.chaSize (nvol);
  Tau.chaSize(nvol); Tau_n.chaSize(nvol); Tau_ant.chaSize(nvol);
  Gtau.chaSize(nvol); Fcalor.chaSize(nvol);
  GU.chaSize (nvol); GU_ant.chaSize(nvol); GU_n.chaSize (nvol);
  GV.chaSize (nvol); GV_ant.chaSize(nvol); GV_n.chaSize (nvol);
  if (ndim == 3 )
    {GW.chaSize (nvol);GW_ant.chaSize(nvol);GW.chaSize (nvol); }
  GP.chaSize (nvol); GP_ant.chaSize(nvol);GP_n.chaSize (nvol);
  for (i=1; i <= nvol; i++) {
    GU(i).chaSize(ndim); GU_ant(i).chaSize(ndim);
    GU_n(i).chaSize(ndim);GV_ant(i).chaSize(ndim);
    GV(i).chaSize(ndim); GV_n(i).chaSize(ndim);
    if (ndim == 3 )
      {GW(i).chaSize (ndim); GW_ant(i).chaSize (ndim);
      GW_n(i).chaSize (ndim);}
    GP(i).chaSize(ndim); GP_n(i).chaSize(ndim);
    GP_ant(i).chaSize(ndim);Tau_ant(i).chaSize(ndim);
    Tau(i).chaSize(ndim);Tau_n(i).chaSize(ndim);
    Gtau(i).chaSize(ndim); Fcalor(i).chaSize(ndim);
  }
  Temp.chaSize(nvol); Temp_ant.chaSize(nvol);
  rho.chaSize(nvol); k.chaSize(nvol);
  cp.chaSize(nvol); mu.chaSize(nvol);
  Volumen.chaSize(nvol);
  elems.chaSize(nvol);
  vcor.chaSize(nvol);
}

void NavierStokes::Condiciones_iniciales()
{
```

```
120   int i; Vector_xyz<real> pos;
      if (CargarDatos) {CargadoDeDatos();}
      else {
        rho.llenar(1.172); k.llenar(0.02566);
        cp.llenar(1007);  mu.llenar(1.858e-5);
125     VelU.llenar(1.0); VelU_ant = VelU;
        VelV.llenar(0.00);VelV_ant = VelV;
        VelW.llenar(0.0); VelW_ant = VelW;
        Presion.llenar(0.0);
        //for (i=1; i<=nvol ;i++) {
130     //  pos = elems(i)->centroid(malla->obtCoorElem(i));
        //  Presion(i) = - (rho(i)*pow(pos(1),2.0)/2.0);
        //}

        Presion_ant = Presion_n= Presion;
135
        for (i=1; i<= nvol; i++) {
          GU_ant(i).llenar(0.0);GU_n(i).llenar(0.0);
          GV_ant(i).llenar(0.0);GV_n(i).llenar(0.0);
          if (ndim == 3 )
140         {GW_ant(i).llenar(0.0); GW_n(i).llenar(0.0);}
          GP_n(i).llenar(0.0);GP_ant(i).llenar(0.0);
          Tau_n(i).llenar(0.0);Tau_ant(i).llenar(0.0);
          Fcalor(i).llenar(0.0);
        }
145     GVel.llenar(0.0);
        Temp.llenar(0.0);Temp_ant.llenar(0.0);
      }
      VmAstU = VelU; VmAstV = VelV;
      if (ndim == 3) VmAstW= VelW;
150   Presionprima = Presion; vcor.llenar(0.0);
    }
    real NavierStokes::DensidadfPyT (int nvol)
    {//P/rho=RT    P/(RT)=rho
      return (Presion_n(nvol) + Patm) /
155     ( Rgas* (TrefparaDensidad + Temp(nvol)) );
    }
    //Imposicion de la temperatura de referencia para calcular la
     densidad
    //y modificaci\'on del parametro de control de la densidad
    void NavierStokes::SetTrefyDensidad (real tt, bool
     densidad_constante)
160 {
      TrefparaDensidad = tt;
      densidad_kte = densidad_constante;
    }
```

```
void NavierStokes::SetPresAtmosf(real pp) { Patm = pp; }
bool NavierStokes::Calcular()
{
  if (!leido) return 0;
  //No. iteraciones alcanzado//convergencia
  bool NIterAlc = false, convergio = false;
  int m=0; real norma; Cadena temm;
  Vector<real> uno(nvol); uno.llenar(1.0);
  Vector<real> cero(nvol); cero.llenar(0.0);
  solucionado = true; //habilitar el guardado de datos

  tiempo_p->iniciarCicloTiempo(); t_aux->iniciarCicloTiempo();
  //inicializacion de las variables
  ConvVu.SetPasadoPartida(VelU_ant,uno);
  ConvVv.SetPasadoPartida(VelV_ant,uno);
  if (ndim == 3)  ConvVw.SetPasadoPartida(VelW_ant,VmAstW);
  Pprima.SetPasadoPartida(Presionprima,cero);
  Corrv.SetPasadoPartida(VelV_ant,uno);
  Temperatura.SetPasadoPartida (Temp_ant,cero);
  CalcularPyG();//Cuando se introduce la presi\'on y el gradiente
   es diferente de cero es recomendable realizar este c\'alculo
   inicialmente
  if (tiempo_p->estacionario()) { //caso estacionario
    theta= 1.0 ; //siempre se asigna este valor al caso
     estacionario
    do  {
      m++;//Contador iteraciones exteriores
      CalculoUnCiclo();
      norma = ResiduoMasico.norma();
      std_o<<" RM norma "<<norma<<" ";
      if (m>=Nitera)
      {std_o<<"\nNo.limite iteraciones alcanzado\n";NIterAlc=true;}
      if ( norma<MAssThreshold)
        convergio = true;
      std_o<<" Iteracion exterior No. "<<m<<" \n\n\n";
    } while (!convergio && !NIterAlc);
    //Se contemplan dos criterios de parada:
    //*el numero maximo de iteraciones
    //*Residuo masico inferior al umbral en la Ec continuidad
    //se exportan los resultados
    if (ResEstable) CuartoPasoSIMPLE(true);
    SolucionUnTiempo();
    Reportarresultados();
  }
  else { //caso transitorio
    if (!ResEstable) {
```

```
      SolucionUnTiempo();
      Reportarresultados();
    }
210 m=0;
    do {
      tiempo_p->incrementarTiempo();
      m++;//Contador iteraciones exteriores
      CalculoUnCiclo();
215   norma = ResiduoMasico.norma();
      std_o<<" RM norma "<<norma<<" ";
      if (m>=Nitera)
        NIterAlc=true;
      if ( norma<MAssThreshold)
220     {convergio = true; NIterAlc = false;}
      std_o<<" Iteracion exterior No. "<<m<<" \n\n\n";
      ActualizarPyTau();
      if (!ResEstable) {
        SolucionUnTiempo();
225     Reportarresultados();
      }
      //Prevenir que lamentar: Guardado automatico de datos
      int ds = m/20;ds = ds*20;
      if (ds==m || m==10) {
230     temm= NombreCaso; NombreCaso = NombreCaso + "temp";
        GuardarDatos(-0.5,2.5);
        NombreCaso = temm;
      }
      } while (!tiempo_p->finalizo() && !convergio && !NIterAlc);
235 //criterios de parada
    //*finalizacion del tiempo de analisis
    //*convergencia de la solucion
    //*numero de iteraciones alcanzado
    if (ResEstable) {
240   //CuartoPasoSIMPLE(true);
      SolucionUnTiempo();
      Reportarresultados();
    }
  }
245 solucionado=true;
  return convergio;
}

int NavierStokes::CalculoUnCiclo()
250 {
  int i;
  if (MetodoSolveNS=="SIMPLE")
```

```
  {PrimerPasoSIMPLE();SegundoPasoSIMPLE();
    TercerPasoSIMPLE();CuartoPasoSIMPLE();
  }
  if (MetodoSolveNS=="NOSIMPLE") {
    PrimerPasoNOSIMPLE();
    std_o<<"VelV Min "<<VelV.minValor()<<" Max "<<VelV.maxValor()<<
     " \n";
    SegundoPasoNOSIMPLE();
    TercerPasoNOSIMPLE();
    CuartoPasoSIMPLE();
  }
  if (MetodoSolveNS=="SIMPLEC")
  { PrimerPasoSIMPLE();SegundoPasoSIMPLEC();
    TercerPasoSIMPLE();CuartoPasoSIMPLE();
  }
  CalcularTauyGVs();
  CalcularPyG();
  return 1;
}
bool NavierStokes::CalcularTauyGVs( )
{
  int i;    Vector_basico<Vector_xyz<real>> VelVec;
  VelVec.chaSize(nvol);
  for (i=1; i <= nvol; i++) {
    VelVec(i).chaSize(ndim);
    VelVec(i)(1)= VelU(i);
    VelVec(i)(2)= VelV(i);
    if (ndim == 3 )
      VelVec(i)(3)= VelW(i);
  }
  NablaEsc (VelU, GU_n, "VelU");
  if (theta!=1.0) NablaEsc (VelU_ant, GU_ant, "VelU");
  NablaEsc (VelV, GV_n, "VelV");
  if (theta!=1.0) NablaEsc (VelV_ant, GV_ant, "VelV");
  if (ndim == 3) NablaEsc (VelW, GW_n, "VelW");
  if (theta!=1.0 && ndim == 3) NablaEsc (VelW_ant, GW_ant, "VelW");
  NablaDotVec(VelVec,GVel_n, "Vel");
  for (i=1; i<= nvol; i++) {
    GVel(i) = (GVel_n(i))*theta + (GVel_ant(i))*(1-theta);
    GU(i) = (GU_n(i)) * theta + (GU_ant(i))*(1-theta);
    GV(i) = (GV_n(i)) * theta + (GV_ant(i))*(1-theta);
    if (ndim == 3)
      GW(i) = (GW_n(i)) * theta + (GW_ant(i))*(1-theta);
  }
  for (i=1; i <= nvol; i++) {
      Tau_n(i)(1,1) = 1.0*(-2.0* mu(i) / 3.0) * GVel_n(i) +
```

```
                        (1.0*mu(i)*GU_n(i)(1));
        Tau_n(i)(2,1) =   mu(i)*(GU_n(i)(2)+GV_n(i)(1));
300     Tau_n(i) (1,2) =  mu(i)*(GU_n(i)(2)+GV_n(i)(1));
        Tau_n(i)(2,2) = 1.0*(-2.0* mu(i) / 3.0) * GVel_n(i) +
                        (1.0*mu(i)*GV_n(i)(2));
      if (ndim==3) {
        Tau_n(i)(3,3) = (-2.0* mu(i) / 3.0) * GVel_n(i) +
305                     (1.0*mu(i)*GW_n(i)(3));
        Tau_n(i)(1,3) = Tau_n(i) (3,1) = mu(i)*(GU_n(i)(3)+GW_n(i)(1)
         );
        Tau_n(i)(2,3) = Tau_n(i) (3,2) = mu(i)*(GV_n(i)(3)+GW_n(i)(2)
         );
      }
      Tau(i)(1,1) = Tau_n(i)(1,1)  *theta + (1-theta) * Tau_ant(i)
       (1,1);
310   Tau(i)(1,2) = Tau(i)(2,1) = Tau_n(i)(1,2)  *theta +
                    (1-theta) * Tau_ant(i)(1,2);
      Tau(i)(2,2) = Tau_n(i)(2,2)  *theta + (1-theta) * Tau_ant(i)
       (2,2);
      if (ndim==3)  {
        Tau(i)(3,3) = Tau_n(i)(3,3)  *theta + (1-theta) * Tau_ant(i)
         (3,3);
315     Tau(i)(1,3) = Tau(i)(3,1) = Tau_n(i)(1,3)  *theta +
                    (1-theta) * Tau_ant(i)(1,3);
        Tau(i)(2,3) = Tau(i)(3,2) = Tau_n(i)(2,3)  *theta +
                    (1-theta) * Tau_ant(i)(2,3);
      }
320 }
    NablaPorMatriz(Tau,Gtau); //*Calculo del gradiente del tau
    return true;
  }
  bool NavierStokes::CalcularPyG( )
325 {
    int i;
    //Modificar las siguientes cuatro l\'ineas cuando se defina un
     gradiente de presi\'on externo
    //if (theta !=0.0 ) NablaEsc (Presion_n, GP_n,"P");
    //if (theta != 1.0) NablaEsc (Presion_ant, GP_ant,"P");
330 if (theta !=0.0 ) NablaEsc (Presion_n, GP_n);
    if (theta != 1.0) NablaEsc (Presion_ant, GP_ant);
    for (i=1; i<=nvol ; i++)
      Presion(i) = Presion_n(i)*theta + Presion_ant(i)*(1.0-theta);
    for (i=1; i<=nvol ; i++)
335   GP(i) = GP_n(i)*theta + GP_ant(i)*(1.0-theta);
    return true;
  }
```

```
bool NavierStokes::CalcQcalor()
{
  int i;
  NablaEsc(Temp,Fcalor,"Temp");
  for (i=1; i<=nvol; i++) Fcalor(i) = Fcalor(i)*k(i)*(-1.0);
  return true;
}
bool NavierStokes::ActualizarPyTau()
{
  int i;
  VelU_ant = VelU; VelV_ant = VelV;
  if (ndim == 3)   VelW_ant = VelW;
  for (i=1; i<=nvol ; i++)  {
     GU_ant(i) = GU_n(i);   GV_ant(i) = GV_n(i);
     if (ndim == 3)        GW_ant(i) = GW_n(i);
     GVel_ant(i)=GVel_n(i); Tau_ant(i) =Tau_n(i);
  }
  Presion_ant = Presion_n;
  for (i=1; i<=nvol ; i++) GP_ant(i) = GP_n(i);
  Temp_ant = Temp ;
  return true;
}

bool NavierStokes::PrimerPasoSIMPLE()
{
  bool correcto;  int i;
  Vector<real> dPdn(nvol), EfVisc(nvol);
  //////////Calculo del termino fuente para vel u
   ////////////////////
  Fuente.llenar(0.0);
  for (i=1; i<=nvol; i++) {dPdn (i) = GP (i)(1); }
  Fuente -= dPdn;//efecto del gradiente de la presion
  for (i=1; i<=nvol; i++) {EfVisc(i) = Gtau(i)(1);}
  Fuente += EfVisc;//efector del grad de la viscosidad
  for (i=1; i<=nvol; i++) Fuente(i) = Fuente(i) * Volumen(i);
  ////////////////Solucion de las ecuaciones para u
   ///////////////////
  ConvVu.SetFuenteYFlujo(Fuente,VelU,VelV, VelW);
  ConvVu.SetPasado(VelU_ant);
  ConvVu.SetPropiedades (rho, mu);
  CoefU  = ConvVu.CalcularCoefUnTiempo
    (theta, tiempo_p, MetodoAprox,  correcto);
  VmAstU = ConvVu.SolucionCoefUnTiempo (correcto);
  ///////////////////Calculo del termino fuente para vel V
   ///////////////////
  Fuente.llenar(0.0);
```

```
  for (i=1; i<=nvol; i++) {dPdn (i) = GP (i)(2); }
  Fuente -= dPdn;
  for (i=1; i<=nvol; i++) {EfVisc(i) = Gtau(i)(2);}
  Fuente += EfVisc;
385  for (i=1; i<=nvol; i++) Fuente(i) = Fuente(i) * Volumen(i);
  /////////////////////Solucion de las ecuaciones para V
   ///////////////////
  ConvVv.SetFuenteYFlujo(Fuente,VelU,VelV, VelW);
  ConvVv.SetPasado(VelV_ant);
  ConvVv.SetPropiedades (rho, mu);
390  CoefV  = ConvVv.CalcularCoefUnTiempo (theta, tiempo_p,MetodoAprox
   , correcto);
  VmAstV = ConvVv.SolucionCoefUnTiempo ( correcto );
  if (ndim == 3 ) {
    Fuente.llenar(0.0);
    for (i=1; i<=nvol; i++) {dPdn (i) = GP (i)(3); }
395    Fuente -= dPdn;
    for (i=1; i<=nvol; i++) {EfVisc(i) = Gtau(i)(3);}
    Fuente += EfVisc;
    for (i=1; i<=nvol; i++) Fuente(i) = Fuente(i) * Volumen(i);
    //Solucion de las ecuaciones para w
400    ConvVw.SetFuenteYFlujo(Fuente,VelU,VelV, VelW);
    ConvVw.SetPropiedades (rho, mu);
    ConvVw.SetPasado(VelW_ant);
    CoefW=ConvVw.CalcularCoefUnTiempo (theta, tiempo_p, MetodoAprox
     , correcto );
    VmAstW = ConvVw.SolucionCoefUnTiempo (correcto);
405  }
  return true;
}

bool NavierStokes::PrimerPasoNOSIMPLE()
410 {
  bool correcto;  int i;
  Vector<real> dPdn(nvol), cero (nvol); cero.llenar(0.0);
  ///////////Calculo del termino fuente para vel u
   /////////////////////
  Fuente.llenar(0.0);
415  for (i=1; i<=nvol; i++) {dPdn (i) = GP (i)(1); }
  Fuente -= dPdn;//efecto del gradiente de la presion
  for (i=1; i<=nvol; i++) Fuente(i) = Fuente(i) * Volumen(i);
  /////////////////Solucion de las ecuaciones para u
   ///////////////////
  ConvVu.SetFuenteYFlujo(Fuente,VelU,VelV, VelW);
420  ConvVu.SetPasado(VelU_ant);
  ConvVu.SetPropiedades (rho, mu);
```

```
  //ConvVu.SetPropiedades (rho, cero, mu, cero);//Viscosidad "
   selectiva" para imitar las ecuaciones de Blasius.
  CoefU  = ConvVu.CalcularCoefUnTiempo
    (theta, tiempo_p, MetodoAprox,  correcto);
  VmAstU = ConvVu.SolucionCoefUnTiempo (correcto);
  ////No se resuelve la ecuacion de cant. mov. en y, se realiza lo
   siguiente
  CoefV = CoefU;//aproximacion
  VmAstV = VelV;//asignacion
  return true;
}
bool NavierStokes::SegundoPasoNOSIMPLE()
{
  bool triunfo; int i;
  Vector<real> cero, uno;
  Vector_basico<Vector_xyz<real>> Velvec(nvol);
  cero.chaSize (nvol); uno.chaSize(nvol);
  cero.llenar (0.0), uno.llenar(1.0);
  for ( i=1 ; i<=nvol ; i++) {
    Velvec(i).chaSize(ndim);
    Velvec(i)(1) = VmAstU(i);
    Velvec(i)(2) = VmAstV(i);
    if (ndim == 3)
     Velvec(i)(3) = VmAstW(i);
  }
  NablaDotVecEspecial(Velvec,ResiduoMasico, MULT_POR_RHO_ON );
  for(i=1;i<=nvol;i++) ResiduoMasico(i) = ResiduoMasico(i)*Volumen(
   i);
  Fuente = ResiduoMasico * (-1.0);
  //solucion por medio de la correcion de la velocidad
  Corrv.SetFuenteYFlujo(Fuente,cero,uno, cero);
  vcor = vcor* 0.2;
  Corrv.SetPasadoPartida(cero,vcor);
  Corrv.SetPropiedades (rho, cero);
  Corrv.CalcularCoefUnTiempo (theta, t_aux, "UDS",  triunfo);
  vcor = Corrv.SolucionCoefUnTiempo (triunfo);
  return true;
}

bool NavierStokes::SegundoPasoSIMPLE()
{
  bool triunfo; int i;
  Vector<real> cero, gammx, gammy, gammz;
  Vector_basico<Vector_xyz<real>> Velvec(nvol);
  cero.chaSize (nvol); cero.llenar (0.0);
```

```
  //Definicion de Gamma:Cuando se usa el mismo tipo de condicion de
    frontera, no es necesario calcular gammay y gammaz
465   gammx.chaSize (nvol);gammy.chaSize (nvol);
  if (ndim == 3) gammz.chaSize (nvol) ;
  for ( i=1 ; i<=nvol ; i++) {
    gammx (i) = rho(i)*Volumen(i)/ CoefU(i);
    gammy (i) = rho(i)*Volumen(i)/ CoefV(i);
470     if (ndim == 3)
     gammz (i) = rho(i)*Volumen(i)/ CoefW(i);
    Velvec(i).chaSize(ndim);
    Velvec(i)(1) = VmAstU(i); Velvec(i)(2) = VmAstV(i);
    if (ndim == 3)
475      Velvec(i)(3) = VmAstW(i);
  }
  NablaDotVecEspecial(Velvec,ResiduoMasico, MULT_POR_RHO_ON );
  Pprima.SetPropiedades(gammx, gammy,gammz);
  for(i=1;i<=nvol;i++)
480     ResiduoMasico(i) = ResiduoMasico(i)*Volumen(i);
  Fuente = ResiduoMasico * (-1.0);
  Pprima.SetFuente (Fuente);
  Presionprima = Presionprima*(0.2);//Iniciacion mejorada
  Pprima.SetPasadoPartida(cero,Presionprima);
485   Pprima.CalcularCoefUnTiempo(1.0, triunfo);
  Presionprima = Pprima.SolucionCoefUnTiempo(triunfo);
  return true;
}
//El m\'etodo SIMPLEC modifica el valor de gamma con respecto al
490 //valor manejado en el metodo SIMPLE
bool NavierStokes::SegundoPasoSIMPLEC()
{
  bool triunfo; int i;
  Vector<real> cero, gammx, gammy, gammz;
495   Vector<real> c2u,c2v,c2w;
  Vector_basico<Vector_xyz<real>> Velvec(nvol);
  cero.chaSize (nvol); cero.llenar (0.0);
  gammx.chaSize (nvol);gammy.chaSize (nvol);
  if (ndim == 3) gammz.chaSize (nvol) ;
500   c2u.chaSize(nvol); c2v.chaSize(nvol);
  c2u = ConvVu.GetCoefVecinos(); c2v = ConvVv.GetCoefVecinos();
  if (ndim == 3) {  c2w.chaSize(nvol);
    c2w= ConvVw.GetCoefVecinos();}
  for ( i=1 ; i<=nvol ; i++) {
505     gammx (i) = rho(i)*Volumen(i)/ (CoefU(i) + c2u(i));
    gammy (i) = rho(i)*Volumen(i)/ (CoefV(i) + c2v(i));
    if (ndim == 3)
      gammz (i) = rho(i)*Volumen(i)/ (CoefW(i) + c2w(i));
```

```
      Velvec(i).chaSize(ndim);
510   Velvec(i)(1) = VmAstU(i); Velvec(i)(2) = VmAstV(i);
      if (ndim == 3)   Velvec(i)(3) = VmAstW(i);
    }
    NablaDotVecEspecial(Velvec,ResiduoMasico, MULT_POR_RHO_ON );
    Pprima.SetPropiedades(gammx, gammy,gammz);
515 for(i=1;i<=nvol;i++) ResiduoMasico(i) = ResiduoMasico(i)*Volumen(
     i);
    Fuente = ResiduoMasico * (-1.0);
    Pprima.SetFuente (Fuente);
    Pprima.SetPasadoPartida(cero,cero);
    Pprima.CalcularCoefUnTiempo(1.0, triunfo);
520 Presionprima = Pprima.SolucionCoefUnTiempo(triunfo);
    return true;
  }
  bool NavierStokes::TercerPasoSIMPLE()
  {
525 int i;
    real alpP = Subrelajacion;
    real alpV = 1.0-alpP;
    Vector_xyz<real> InvVec (ndim);
    Vector_basico <Vector_xyz<real>> Gradiente (nvol);
530 Vector<real> uprima(nvol), vprima(nvol), wprima(nvol);
    Vector<real> uAst(nvol), vAst(nvol), wAst(nvol);
    for (i=1;i<=nvol;i++) Gradiente(i).chaSize (ndim);
    Presion_n += (Presionprima*alpP);//Correccion de Presion
    NablaEsc(Presionprima, Gradiente, "P");
535 for (i=1; i<= nvol ; i++) { //Correccion de Velocidad
      InvVec(1) = 1.0 / CoefU(i);//xxxxxxxxxxx
      uprima(i) = - Volumen(i) * InvVec(1) *Gradiente(i)(1);
      uAst(i) =  VmAstU(i) + uprima(i);
      VelU(i) = alpV * uAst(i) + (1.0-alpV) * VelU(i);
540
      InvVec(2) = 1.0 / CoefV(i);//yyyyyyyyyy
      vprima(i) = -Volumen(i) * InvVec(2) *Gradiente(i)(2);
      vAst(i) = VmAstV(i) + vprima(i);
      VelV(i) = alpV * vAst(i) + (1.0-alpV) * VelV(i);
545
      if (ndim == 3) {//zzzzzzzzzzzz
        InvVec(3) = 1.0 / CoefW(i);
        wprima(i) = -Volumen(i) * InvVec(3)*Gradiente(i)(3);
        wAst(i) = VmAstW(i) + wprima(i);
550     VelW(i) = alpV * wAst(i) + (1.0 - alpV) * VelW(i);
      }
    }
    return true;
```

```
}
bool NavierStokes::TercerPasoNOSIMPLE()
{
  int i;
  real alpV = Subrelajacion;
  //No se corrige la presion
  for (i=1; i<= nvol ; i++) { //Correccion de Velocidad
    VelU(i) = alpV * VmAstU(i) + (1.0-alpV) * VelU(i);
    VelV(i) = VelV(i) + alpV*vcor(i);
  }
  return true;
}

bool NavierStokes::TercerPasoSIMPLEC()
{
  int i;
  real alpP = 1.0; real alpV = 1.0;
  Vector_xyz<real> InvVec (ndim);
  Vector_basico <Vector_xyz<real>> Gradiente (nvol);
  Vector<real> uprima(nvol), vprima(nvol), wprima(nvol);
  Vector<real> uAst(nvol), vAst(nvol), wAst(nvol);
  for (i=1;i<=nvol;i++) Gradiente(i).chaSize (ndim);
  Presion_n += (Presionprima*alpP);//Correccion de Presion
  NablaEsc(Presionprima, Gradiente, "P");
  for (i=1; i<= nvol ; i++) { //Correccion de Velocidad
    InvVec(1) = 1.0 / CoefU(i);//xxxxxxxxxxx
    uprima(i) = - Volumen(i) * InvVec(1) *Gradiente(i)(1);
    uAst(i) =  VmAstU(i) + uprima(i);
    VelU(i) = alpV * uAst(i) + (1.0-alpV) * VelU(i);
    InvVec(2) = 1.0 / CoefV(i);//yyyyyyyyyy
    vprima(i) = -Volumen(i) * InvVec(2) *Gradiente(i)(2);
    vAst(i) = VmAstV(i) + vprima(i);
    VelV(i) = alpV * vAst(i) + (1-alpV) * VelV(i);
    if (ndim == 3) {//zzzzzzzzzzz
    InvVec(3) = 1.0 / CoefW(i);
    wprima(i) = -Volumen(i) * InvVec(3)*Gradiente(i)(3);
    wAst(i) = VmAstW(i) + wprima(i);
    VelW(i) = alpV * wAst(i) - (1.0 - alpV) * VelW(i);
    }
  }
  return true;
}
//Calculo otras ecuaciones,
bool NavierStokes::CuartoPasoSIMPLE(bool forzar)
{
  bool correcto; int i; Vector<real> cero(nvol); cero.llenar(0.0);
```

```
600   if ( !densidad_kte || !ResEstable || forzar) {
        Temperatura.SetFuenteYFlujo(cero,VelU,VelV, VelW);
        Temperatura.SetPasado (Temp_ant);
        Temperatura.SetPropiedades(rho,k,cp);
        if (ResEstable && forzar) {
605       Temperatura.CalcularCoefUnTiempo
            (theta, t_aux, MetodoAprox, correcto);
        }
        else{
          Temperatura.CalcularCoefUnTiempo
610         (theta, tiempo_p, MetodoAprox, correcto);
        }
        Temp = Temperatura.SolucionCoefUnTiempo(correcto, false);
        CalcQcalor();
      }
615   if (!densidad_kte) {
        for (i=1; i<=nvol; i++)
          rho(i) = DensidadfPyT(i);
      }
      return true;
620 }
    //Cada clase de conveccion o difusion se encarga de resolver
    //el problema, se conserva la conversion de datos de tipo
    //vector a tipo campo,
    int NavierStokes::SolucionUnTiempo()
625 {
      CalcPresRef();
      Vector<real> gtu(nvol), gtv(nvol), gtw(nvol);
      gdl->vector2campo(VelU,campo_u());
      gdl->vector2campo(Presion,campo_pres());
630   gdl->vector2campo(Presionprima,campo_pres_prima());
      gdl->vector2campo(vcor,campo_corr_vel());
      gdl->vector2campo(Temp,campo_T());
      gdl->vector2campo(ResiduoMasico,campo_residuo());
      gdl->vector2campo(VelU,campo_vel()(1));
635   gdl->vector2campo(VelV,campo_vel()(2));
      if ( malla->obtNoDimEspacio()==3)
        gdl->vector2campo(VelW,campo_vel()(3));
      for (int i=1; i<= nvol; i++) {
        gtu(i)=Gtau(i)(1);
640     gtv(i)=Gtau(i)(2);
        if (ndim==3) gtw(i)=Gtau(i)(3);
      }
      gdl->vector2campo(gtu,campo_GTau()(1));
      gdl->vector2campo(gtv,campo_GTau()(2));
645   if (ndim==3) gdl->vector2campo(gtw,campo_GTau()(3));
```

```
  for (int i=1; i<= nvol; i++) {
    gtu(i)=GU(i)(1);
    gtv(i)=GU(i)(2);
    if (ndim==3) gtw(i)=GU(i)(3);
  }
  gdl->vector2campo(gtu,campo_GU()(1));
  gdl->vector2campo(gtv,campo_GU()(2));
  if (ndim==3) gdl->vector2campo(gtw,campo_GU()(3));

  for (int i=1; i<= nvol; i++) {
    gtu(i)=GV(i)(1);
    gtv(i)=GV(i)(2);
    if (ndim==3) gtw(i)=GV(i)(3);
  }
  gdl->vector2campo(gtu,campo_GV()(1));
  gdl->vector2campo(gtv,campo_GV()(2));
  if (ndim==3)  gdl->vector2campo(gtw,campo_GV()(3));

  for (int i=1; i<= nvol; i++) {
    gtu(i)=GP(i)(1);
    gtv(i)=GP(i)(2);
    if (ndim==3) gtw(i)=GP(i)(3);
  }
  gdl->vector2campo(gtu,campo_GP()(1));
  gdl->vector2campo(gtv,campo_GP()(2));
  if (ndim==3) gdl->vector2campo(gtw,campo_GP()(3));
  for (int i=1; i<= nvol; i++) {
    gtu(i)=Fcalor(i)(1);
    gtv(i)=Fcalor(i)(2);
    if (ndim==3) gtw(i)=Fcalor(i)(3);
  }
  gdl->vector2campo(gtu,campo_Q()(1));
  gdl->vector2campo(gtv,campo_Q()(2));
  if (ndim==3) gdl->vector2campo(gtw,campo_Q()(3));
  return true;
}

void NavierStokes::Reportarresultados()
{
  if (ResEstable) {
    volcar(campo_vel(),0);
    volcar(campo_pres(),0);
    volcar(campo_u(),0);
    volcar(campo_residuo(),0);
    if (MetodoSolveNS=="NOSIMPLE")
```

```
        {volcar(campo_corr_vel(),0);}
      else { volcar(campo_pres_prima(),0);}
      volcar(campo_GTau(),0);
695   volcar(campo_GU(),0);
      volcar(campo_GV(),0);
      volcar(campo_GP(),0);
      volcar(campo_T(),0);
      volcar(campo_Q(),0);
700 }
    else {
      volcar(campo_T(),tiempo_p.obtPtr());
      volcar(campo_vel(),tiempo_p.obtPtr());
      volcar(campo_pres(),tiempo_p.obtPtr());
705   volcar(campo_u(),tiempo_p.obtPtr());
      if (MetodoSolveNS=="NOSIMPLE")
      {volcar(campo_corr_vel(),tiempo_p.obtPtr());}
      else { volcar(campo_pres_prima(),tiempo_p.obtPtr());}
      volcar(campo_residuo(),tiempo_p.obtPtr());
710   volcar(campo_Q(),tiempo_p.obtPtr());
    }
  }
  bool NavierStokes::LeerFronteras(Cadena TomarFrontera,
   Vector_basico<Cadena> &fr)
  {
715 int i, j, k, nbi;
    Cadena med;
    nbi = malla->obtNoIndFront();
    fr.chaSize (nbi);
    for (i=1; i<=nbi; i++) {//Por cada tipo de frontera
720   fr(i) = malla->obtNombreIndFront(i);
      //Verificacion para evitar errores
      if (fr(i).contiene(TomarFrontera)) {
        j = fr(i).Buscar(0,TomarFrontera);
        k = fr(i).tam();
725     med = fr(i).subCadena(j, k-j );
        if (med.contiene(Separador)) {
          j = med.Buscar(0,Separador);
          med = med.subCadena(0,j);
        }
730   fr(i) = med;
      }
    }
  return true;
  }
735 //calculo de la condicion de frontera para cada componente de la
   velocidad
```

```
bool NavierStokes::LeerFronteras(Vector_basico<Cadena> &fr1,
 Vector_basico<Cadena> &fr2, Vector_basico<Cadena> &fr3)
{
  int i,j,k, nbi;
  nbi = malla->obtNoIndFront();
  Cadena med, ap;                                              740
  fr1.chaSize(nbi);fr2.chaSize(nbi);fr3.chaSize(nbi);
  for (i=1; i<=nbi; i++)
    fr1(i)=fr2(i)=fr3(i) = malla->obtNombreIndFront(i);
  for (i=1; i<=nbi; i++){
    //Verificacion para evitar errores                         745
    if (fr1(i).contiene("VelU"))
    { j = fr1(i).Buscar(0,"VelU");
      k = fr1(i).tam();
      med = fr1(i).subCadena(j,k-j);
      if (med.contiene(Separador))                             750
      {j = med.Buscar(0,Separador);
       med = med.subCadena(0,j);
      }
      fr1(i) = med;
    }                                                          755
    if (fr2(i).contiene("VelV"))
    { j = fr2(i).Buscar(0,"VelV");
      k = fr2(i).tam();
      med = fr2(i).subCadena(j,k-j);
      if (med.contiene(Separador))                             760
      {j = med.Buscar(0,Separador);
       med = med.subCadena(0,j);
      }
      fr2(i) = med;
    }                                                          765
    if (fr3(i).contiene("VelW"))
    { j = fr3(i).Buscar(0,"VelW");
      k = fr3(i).tam();
      med = fr3(i).subCadena(j,k-j);
      if (med.contiene(Separador))                             770
      { j = med.Buscar(0,Separador);
        med = med.subCadena(0,j);
      }//se extrae exactamente la cadena que se
      fr3(i) = med;//necesita para cada condicion
    }//de frontera.                                            775
  }
return true;
}
```

```
780 real NavierStokes::LeerFrontEsp(int i, int j, real muestra, Cadena
     fr )
    {
      real valor = muestra, valor2, px, py; Cadena bovalue = fr.despues
       ("=");
      Vector_xyz<real> minc, maxc; malla->obtCoorMinMax(minc, maxc);
      if (fr.contiene("Dirichlet") ) {
785     valor   = atof(bovalue.carts());
        px = elems(i)->centroArea(malla->obtCoorElem(i),j)(1);
        py = elems(i)->centroArea(malla->obtCoorElem(i),j)(2);
        if (fr.contiene("e^")) {
          bovalue = fr.despues("e^"); valor2 = atof(bovalue.carts());
790       valor = valor * exp( valor2*(px-minc(1)) );
        }
        if (fr.contiene("x^")){
          bovalue = fr.despues("x^"); valor2 = atof(bovalue.carts());
          valor = valor * pow( px-minc(1) ,valor2);
795     }
        if (fr.contiene("rooty")){
          bovalue = fr.despues("p=");valor2 = atof(bovalue.carts());
          if (py<valor2){
            valor = valor * sqrt(py/valor2);
800         }
        }
        if (fr.contiene("liny")){
          bovalue = fr.despues("p=");valor2 = atof(bovalue.carts());
          if (py<valor2){
805       valor = valor * (py/valor2);
          }
        }
      }
      if (fr.contiene("Neumann") ){
810     valor = atof(bovalue.carts());
        valor = muestra - valor*elems(i)->getDelta(j);
      }
      return valor;
    }
815 //Gradiente de un escalar, la salida depende del numero de
     dimensiones de la malla
    bool NavierStokes::NablaEsc ( Vector<real> &Entrada, Vector_basico<
     Vector_xyz<real>> &Salida, Cadena TomarFrontera )
    { int i, j, k, veci, conx; int nbi= malla->obtNoIndFront();
      real prom, area, d1, d2;
      Vector_xyz<real> dir(ndim); Vector_basico<Cadena> fr;
820   //si se deben tomar las condiciones de frontera del archivo de
       definicio o no
```

```
  if (TomarFrontera!="Ninguna") LeerFronteras(TomarFrontera,fr);
  if (nvol == 9)  std_o<<"\n NablaEsc\n";
  for (i= 1 ; i<= nvol ; i++) {
    Salida(i).llenar(0.0);
    conx = elems(i)->GetNumOfConex();
      for ( j=1; j<=conx ; j++)   {
        veci = malla->getConx (i, j); area = elems(i)->getArea(j);
        if ( veci==0) {
          if (TomarFrontera!="Ninguna") {
            for (k=1; k<=nbi; k++)
              if (  malla->LadoFront(i,j,k,elems(i)())) break;
            prom = LeerFrontEsp (i,j, Entrada (i),fr(k));
          }
          else {prom = Entrada(i);}
        }
        else {for ( k=1 ; k<=conx ; k++ )
              if(malla->getConx(veci,k)==i) break;
            d1 = elems(i)->getDelta(j);
            d2 = elems(veci)->getDelta(k);
             prom = (Entrada(veci) * d1 + Entrada (i) * d2)/(d1+d2)
              ;
           }
        Salida(i) = Salida(i) + (elems(i)->GetDirSide(j)*(area*prom
         ));
      }
    Salida(i) = Salida(i) / Volumen(i);
    if(nvol == 9)
      {Salida(i).Escribir(std_o);std_o<<"\n";}
  }
return true;
}
//Producto punto del operador nabla por el vector entrada
bool NavierStokes::NablaDotVec ( Vector_basico< Vector_xyz <real>>
       &Entrada, Vector<real> &Salida, Cadena Frontera)
{
  int i, j, k, veci, conx; int nbi= malla->obtNoIndFront();
  real area, rx,ry,rz,d1,d2;  Vector_xyz<real> prom (ndim);
  Vector_basico<Cadena> fr1,fr2,fr3;
  //si se deben tomar las condiciones de frontera del archivo de
   definicion o no
  if (Frontera=="Vel")  LeerFronteras(fr1, fr2, fr3);
  Salida.llenar (0.0);
  for (i= 1 ; i<= nvol ; i++) {
    conx = elems(i)->GetNumOfConex(); rx = ry =rz = 0;
    for ( j=1; j<=conx ; j++) {
      veci = malla->getConx (i, j); area = elems(i)->getArea(j);
```

```
        if ( veci==0) {
865       if (Frontera.contiene("Vel")) {
            for (k=1; k<=nbi; k++)
              if (  malla->LadoFront(i,j,k,elems(i)())) break;
            prom(1) = LeerFrontEsp(i,j,Entrada (i)(1),fr1(k));
            prom(2) = LeerFrontEsp(i,j,Entrada (i)(2),fr2(k));
870         if (ndim == 3)
            prom(3) = LeerFrontEsp(i,j,Entrada (i)(3),fr3(k));
          }
          else {prom = Entrada(i);}
        }
875     else {
          for ( k=1 ; k<=conx ; k++ )
            if(malla->getConx(veci,k)==i)   break;
          d1 = elems(i)->getDelta(j);
          d2 = elems(veci)->getDelta(k);
880       prom = (Entrada(veci) * d1 + Entrada (i) * d2)/(d1+d2);
        }
        rx += (prom(1)*elems(i)->GetDirSide(j)(1))*area;
        ry += (prom(2)*elems(i)->GetDirSide(j)(2))*area;
        if (ndim==3)
885       rz += (prom(3)*elems(i)->GetDirSide(j)(3))*area;
        }
        Salida(i) = rx + ry ;
        if (ndim==3) Salida(i) += rz;
        Salida(i) = Salida(i) / Volumen(i);
890   }
return true;
}
//Funcion para el c\'alculo del gradiente de la velocidad mAsterico
  teniendo en cuenta
//la correccion de presion debida al uso de una malla colocada. No
 ser\'ia necesario
895 //llamar a esta funcion si se usara una malla escalonada.
bool NavierStokes::NablaDotVecEspecial ( Vector_basico< Vector_xyz
 <real>>
        &Entrada, Vector<real> &Salida, bool MultRho )
{
  int i, j, k, veci, conx;  int nbi= malla->obtNoIndFront();
900 real area, coefar, deltap, gradp, sum, act, densidadprom, d1, d2;
  Vector_xyz<real> prom (ndim), coefvel(ndim), GPprom(ndim), efec(
   ndim);
  //Condiciones de frontera para uvw
  Vector_basico<Cadena> fr1,fr2,fr3;  LeerFronteras(fr1, fr2, fr3);
  Salida.llenar (0.0);
905 if (nvol == 9)  std_o<<"\nNablaDotEspecial\n";
```

```
for (i= 1 ; i<= nvol ; i++) {
  conx = elems(i)->GetNumOfConex(); sum = 0.0;
  for ( j=1; j<=conx ; j++) {
    veci = malla->getConx (i, j);
    area = elems(i)->getArea(j);
    if ( veci==0) {
      for (k=1; k<=nbi; k++)
        if (  malla->LadoFront(i,j,k,elems(i)())) break;
      prom(1) = LeerFrontEsp(i,j,Entrada (i)(1),fr1(k));
      prom(2) = LeerFrontEsp(i,j,Entrada (i)(2),fr2(k));
      if (ndim == 3)
        prom(3) = LeerFrontEsp(i,j,Entrada (i)(3),fr3(k));
      densidadprom = rho(i);
      //Coeficiente del volumen y la distancia
      coefar = elems(i)->getArea(j);//Volumen(i)/ elems(i)->
       getDelta(j);
      //Coeficiente de la diagonal principal de la matriz de
       conveccion-difusion
      coefvel(1) = (1.0/CoefU(i)) *
        elems(i)->GetDirSide(j)(1) ;
      coefvel(2) = (1.0/CoefV(i))*
        elems(i)->GetDirSide(j)(2) ;
      if (ndim == 3)  coefvel(3) = (1.0/CoefW(i))*
        elems(i)->GetDirSide(j)(3) ;
      //Calculo del Delta de Presion
      deltap = 0.0;
      //Calculo del promedio del gradiente de presion
      GPprom.llenar(0.0);
      //Calculo del cambio de la presion por el grad
      gradp = GPprom * elems(i)->GetDirSide(j);
      gradp *= elems(i)->getDelta  (j);
      //Se calcula la presion en la frontera de cada volumen de
      //control a partir de los factores anteriormente
       mencionados
    }
    else {
      for ( k=1 ; k<=conx ; k++ )
      { if(malla->getConx(veci,k)==i)   break;}
      d1 = elems(i)->getDelta(j);
      d2 = elems(veci)->getDelta(k);
      //promedio de la densidad
      densidadprom = (d2*rho(i)+d1*rho(veci))/(d1+d2);
      //promedio de la velocidad-Termino
      prom = (Entrada(i)*d2+Entrada(veci)*d1)/(d1+d2);
      //Coeficiente del volumen y la distancia
      coefar =elems(i)->getArea(j);
```

```
        //Coeficiente de la diagonal principal de la matriz de
         conveccion-difusion
950     coefvel(1) = ((d2/CoefU(i) + d1/CoefU(veci))/(d1+d2))*
          elems(i)->GetDirSide(j)(1);
        coefvel(2) = ((d2/CoefV(i) + d1/CoefV(veci))/(d1+d2))*
          elems(i)->GetDirSide(j)(2);
        if (ndim == 3)
955       coefvel(3) = ((d2/CoefW(i) + d1/CoefW(veci))/(d1+d2))*
          elems(i)->GetDirSide(j)(3);
        //Calculo del Delta de Presion
        deltap =  Presion(veci) - Presion(i) ;
        //Calculo del promedio del gradiente de presion
960     GPprom = (GP(i)*d2+ GP(veci)*d1) / (d1+d2);
        //Calculo del cambio de la presion por el grad
        gradp = GPprom * elems(i)->GetDirSide(j);
        gradp = gradp * (elems(i)->getDelta(j) + elems(veci)->
         getDelta(k));
      }
965   //Se calcula la presion en la frontera de cada volumen de
        //control a partir de los factores anteriormente
         mencionados
      efec = coefvel * coefar * (deltap-gradp);
      prom = prom - efec;
      if (MultRho)
970     prom = prom * densidadprom;
      act  = (prom*elems(i)->GetDirSide(j))*area;
      sum += act;
    }
    Salida(i) = sum;
975 Salida(i) = Salida(i) / Volumen(i);
  }
return true;
}

980 bool NavierStokes::TensorPorVector_xyz ( Matriz<real> Me,
        Vector_xyz<real> Ve, Vector_xyz<real> &Salida)
{
  int i, j; Salida.llenar(0.0);
  for (i = 1 ; i<= ndim; i++) for (j = 1 ; j<= ndim; j++)
985     Salida (i) += Me(j,i) * Ve(j);
  return true;
}
//Calculo del promedio de dos matrices teniendo en cuenta un factor
  de peso para cada matriz
Matriz<real> NavierStokes::PromMatriz(Matriz<real> E1,
990     real Peso1, Matriz<real> E2, real Peso2)
```

```
{
  int i, j , k , l ; Matriz<real> Salida(E1);
  k = E1.obtNoFilas(); l = E1.obtNoCols ();
  for (i = 1 ; i<= k; i++) for (j = 1 ; j<= l; j++)
995     Salida(i,j) = ((E1(i,j)*Peso1)+(E2(i,j)*Peso2))
        / ( Peso1 + Peso2 );
  return Salida;
}
//Producto del operador Nabla por un Tensor
1000 bool NavierStokes::NablaPorMatriz ( Vector_basico< Matriz <real>>
        &Entrada, Vector_basico<Vector_xyz<real>> &Salida)
{
  int i, j, k, veci, conx;
  real area, d1, d2; Matriz<real> prom(ndim);
1005 Vector_xyz<real> prod (ndim), dir(ndim);
  for (i= 1 ; i<= nvol ; i++) {
    Salida(i).llenar(0.0);  conx = elems(i)->GetNumOfConex();
    for ( j=1; j<=conx ; j++) {
      veci = malla->getConx (i, j); area = elems(i)->getArea(j);
1010   if ( veci==0)
      { TensorPorVector_xyz(Entrada(i),elems(i)->GetDirSide(j),
       prod); }
      else {
        for ( k=1 ; k<=conx ; k++ )
        { if(malla->getConx(veci,k)==i)   break;}
1015     d1 = elems(i)->getDelta(j);
        d2 = elems(veci)->getDelta(k);
        prom = PromMatriz(Entrada(i),d2,Entrada(veci),d1);
        TensorPorVector_xyz( prom, elems(i)->GetDirSide(j), prod);
      }
1020   Salida(i) = Salida(i) + (prod * area);
    }
    Salida(i) = Salida(i) / Volumen(i);
  }
  return true;
1025 }

//Guardado de datos, Ademas de NombreCaso se
//indica el numero de VC para evitar errores en el cargado de datos
bool NavierStokes:: GuardarDatos(real minvel,real maxvel)
1030 {
  real minu, minv, minw, minimo;
  real maxu, maxv, maxw, maximo;
  minu = VelU.minValor(); minv = VelV.minValor();
  if (ndim == 3)  minw = VelW.minValor();
1035 else      minw = 1e6;
```

```
  minimo = minu;  if (minv<minimo) minimo = minv;
  if ( minw<minimo) minimo =minw;
  maxu = VelU.maxValor(); maxv = VelV.maxValor();
  if (ndim == 3)  maxw = VelW.maxValor();
1040  else maxw =-1e6;
  maximo = maxu; if (maxv>maximo) maximo = maxv;
  if (maxw>maximo) maximo = maxw;
  if (minimo>minvel && maximo<maxvel)
    {return GuardarDatos();}
1045  else
    {std_o<<"Error:: no se exportan los datos a *.m porque se
     excedio"
    <<" el limite de la velocidad establecida\n"; }
  return false;
}
1050 bool NavierStokes:: GuardarDatos()
{
  if (!solucionado)
     {std_o<<"\nError: no se ha solucionado el problema\n";}
  char nv [10]; Cadena nArchivo; _itoa(nvol,nv,10); strcat (nv,".m"
   );
1055  nArchivo = NombreCaso + "rho"; nArchivo += nv;
  rho.guardar(nArchivo.carts(),"rho");
  nArchivo = NombreCaso + "k"; nArchivo += nv;
  k.guardar(nArchivo.carts(),"k");
  nArchivo = NombreCaso + "cp"; nArchivo += nv;
1060  cp.guardar(nArchivo.carts(),"cp");
  nArchivo = NombreCaso + "mu"; nArchivo += nv;
  mu.guardar(nArchivo.carts(),"mu");
  nArchivo = NombreCaso + "VelU"; nArchivo += nv;
  VelU.guardar(nArchivo.carts(),"VelU");
1065  nArchivo = NombreCaso + "VelV"; nArchivo += nv;
  VelV.guardar(nArchivo.carts(),"VelV");
  nArchivo = NombreCaso + "VelW"; nArchivo += nv;
  if (ndim==3)
    VelW.guardar(nArchivo.carts(),"VelW");
1070  nArchivo = NombreCaso + "VelU_ant"; nArchivo += nv;
  VelU_ant.guardar(nArchivo.carts(),"VelU_ant");
  nArchivo = NombreCaso + "VelV_ant"; nArchivo += nv;
  VelV_ant.guardar(nArchivo.carts(),"VelV_ant");
  nArchivo = NombreCaso + "VelW_ant"; nArchivo += nv;
1075  if (ndim==3)
    VelW_ant.guardar(nArchivo.carts(),"VelW_ant");
  nArchivo = NombreCaso + "Presion_n"; nArchivo += nv;
  Presion_n.guardar(nArchivo.carts(),"Presion_n");
  nArchivo = NombreCaso + "Presion_ant"; nArchivo += nv;
```

```
1080   Presion_ant.guardar(nArchivo.carts(),"Presion_ant");
       nArchivo = NombreCaso + "Temp"; nArchivo += nv;
       Temp.guardar(nArchivo.carts(),"Temp");
       nArchivo = NombreCaso + "Temp_ant"; nArchivo += nv;
       Temp_ant.guardar(nArchivo.carts(),"Temp_ant");
1085   return true;
     }
     //Funcion para cargar los vectores necesarios
     //para continuar resolviendo el problema
     bool NavierStokes:: CargadoDeDatos()
1090 {
       char nv [10]; Cadena nArchivo;
       _itoa(nvol,nv,10);  strcat (nv,".m");
       nArchivo = NombreCaso + "rho"; nArchivo += nv;
       rho.cargar(nArchivo.carts(),"rho");
1095   nArchivo = NombreCaso + "k"; nArchivo += nv;
       k.cargar(nArchivo.carts(),"k");
       nArchivo = NombreCaso + "cp"; nArchivo += nv;
       cp.cargar(nArchivo.carts(),"cp");
       nArchivo = NombreCaso + "mu"; nArchivo += nv;
1100   mu.cargar(nArchivo.carts(),"mu");
       nArchivo = NombreCaso + "VelU"; nArchivo += nv;
       VelU.cargar(nArchivo.carts(),"VelU");
       nArchivo = NombreCaso + "VelV"; nArchivo += nv;
       VelV.cargar(nArchivo.carts(),"VelV");
1105   nArchivo = NombreCaso + "VelW"; nArchivo += nv;
       if (ndim==3)
         VelW.cargar(nArchivo.carts(),"VelW");
       nArchivo = NombreCaso + "VelU_ant"; nArchivo += nv;
       VelU_ant.cargar(nArchivo.carts(),"VelU_ant");
1110   nArchivo = NombreCaso + "VelV_ant"; nArchivo += nv;
       VelV_ant.cargar(nArchivo.carts(),"VelV_ant");
       nArchivo = NombreCaso + "VelW_ant"; nArchivo += nv;
       if (ndim==3)
         VelW_ant.cargar(nArchivo.carts(),"VelW_ant");
1115   nArchivo = NombreCaso + "Presion_n"; nArchivo += nv;
       Presion_n.cargar(nArchivo.carts(),"Presion_n");
       nArchivo = NombreCaso + "Presion_ant"; nArchivo += nv;
       Presion_ant.cargar(nArchivo.carts(),"Presion_ant");
       Presionprima.llenar(0.0);
1120
       nArchivo = NombreCaso + "Temp"; nArchivo += nv;
       Temp.cargar(nArchivo.carts(),"Temp");
       nArchivo = NombreCaso + "Temp_ant"; nArchivo += nv;
       Temp_ant.cargar(nArchivo.carts(),"Temp_ant");
1125
```

```
  CalcularTauyGVs();
  CalcularPyG();
  CalcQcalor();
  return true;
}
//Cuando el tiempo necesario para calcular los datos
//es alto puede ser conveniente continuar las iteraciones
//a partir del ultimo valor obtenido
bool NavierStokes::SetCargarDatos(bool opcion)
{ CargarDatos = opcion; return true; }
//Asignacion del numero de iteraciones externas maximo para
 calcular
//cada ciclo de tiempo
void NavierStokes::SetNoIterExt(int set){ Nitera = set;}
//Metodo solucion
bool NavierStokes::SetTipoMetodoSolucion(Cadena tipo)
{
  bool coincide = false; int i=0; const char *cad= tipo.carts();
  while ( MSNS[i]) {
    if ( strcmp(cad,(MSNS[i])) ==0 ) {coincide=true; break;}
    i++; }
  if (coincide)  MetodoSolveNS = tipo;
  else {
    std_o<<"\nMetodo no encontrado: se toma valor por defecto\n";
    system("pause");
    //se toma el valor por defecto del constructor
  }
  return coincide;
}
//Funcion para la asignacion del esquema de conveccion a usar para
 resolver
//las ecuaciones de conveccion difusion
bool NavierStokes::SetEsquemaConveccion(Cadena tipo)
{
  bool coincide = false; int i=0;
  while ( MACD[i]){
    if (tipo.contiene(MACD[i])) {coincide=true; break;}
    i++;}
  if (coincide) MetodoAprox=tipo;
  else {
    std_o<<"\nEsquema no encontrado: se toma valor por defecto\n";
    system("pause");
    //se toma el valor por defecto del constructor
  }
  return coincide;
}
```

```
1170 void NavierStokes::SetMaximaPresion(real max)
     {maxPres= true; minPres= false; maxP = max;}
     void NavierStokes::SetMinimaPresion(real min)
     {maxPres= false; minPres= true; minP = min;}
     //Metodo para referenciar el valor de la presion obtenido
1175 //a un valor espec\'ifico
     void NavierStokes::CalcPresRef()
     {
       int i; real r;
       if (maxPres || minPres) {
1180     if(maxPres){r=Presion.maxValor(); r= maxP-r;}
         if(minPres){r=Presion.minValor(); r= minP-r;}
         for (i=1; i<=nvol; i++)
           Presion(i) = Presion(i) + r;
       }
1185 }
     void NavierStokes::ResetMinMaxPresion() { minP = maxP = false;}
     //Funcion para la asignacion de los parametros mas importantes para
     //realizar el calculo
     bool NavierStokes::SetParametros (bool Datos_iniciales_externos ,
1190   int N_iteraciones_ext, Cadena Esquema_conveccion,
       Cadena Esquema_solucion, bool Resultados_indep_tiempo )
     {
       bool correcto = true;
       SetCargarDatos(Datos_iniciales_externos);
1195   SetNoIterExt(N_iteraciones_ext);
       correcto = correcto && SetEsquemaConveccion(Esquema_conveccion);
       correcto = correcto && SetTipoMetodoSolucion (Esquema_solucion);
       ResEstable = Resultados_indep_tiempo;
       return correcto;
1200 }
     bool NavierStokes::SetMAssThreshold(real MaxResMas)
     {MAssThreshold = MaxResMas; return true;}

     //Asignacion del parametros de subrelajacion de la presion
1205 void NavierStokes::SubRelajaSIMPLE(real param) {Subrelajacion =
      param;}
```

ANEXO O. MÉTODO EXPLÍCITO PARA LA SOLUCIÓN DE LAS ECUACIONES DE LA CAPA LÍMITE

La clase desarrollada para la solución de las ecuaciones de la capa límite se pone a prueba con los resultados analíticos de la solución planteada por Blasius. Sin embargo, debido a la existencia de un método explicito, resulta conveniente explicar el método y comparar los resultados con este método de rápida solución y mayor número de restricciones.

El método que se explica a continuación es descrito por Schetz [25] y fue originalmente desarrollado por Wu [1]. El proceso está basado en el método de diferencias finitas y es explicado para un análisis en estado estable, propiedades constantes y flujo sobre una placa plana.

Antes de comenzar a explicar el método, se reescriben las ecuaciones de continuidad (Ec O.1), cantidad de movimiento (Ec O.2) y energía (Ec O.3) del flujo en la capa límite las cuales fueron presentadas en el capítulo 1.

$$\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}=0 \qquad \text{(Ec O.1)}$$

$$\rho u\frac{\partial u}{\partial x}+\rho v\frac{\partial u}{\partial y}=-\frac{dp}{dx}+\mu\frac{\partial^2 u}{\partial y^2} \qquad \text{(Ec O.2)}$$

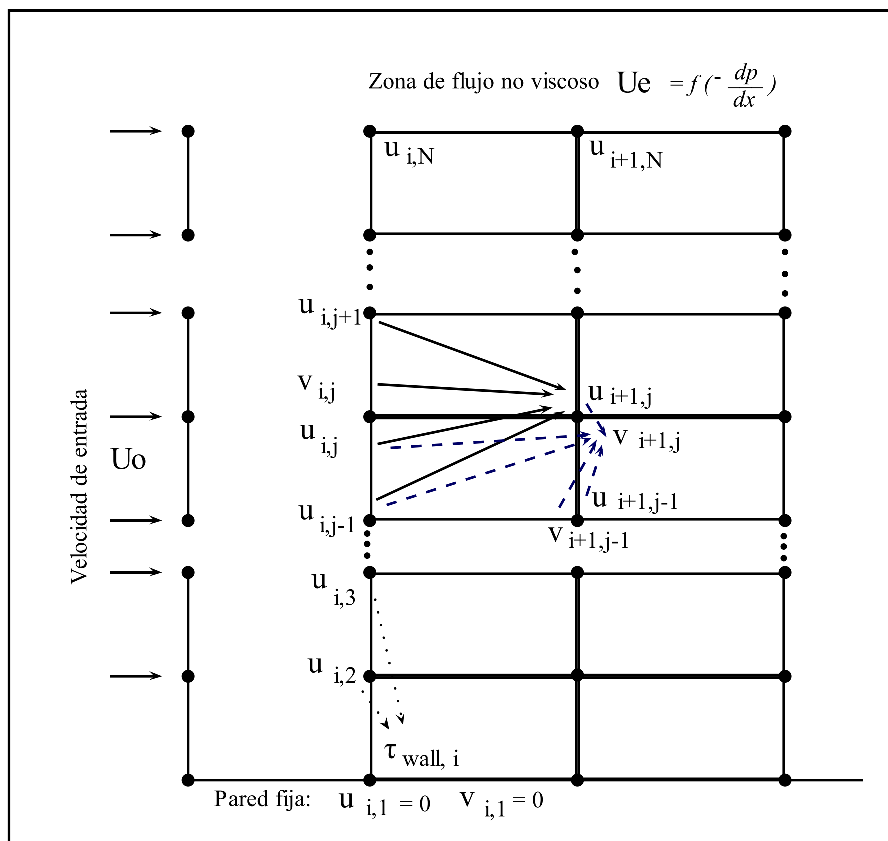
$$\rho\left[u\frac{\partial T}{\partial x}+v\frac{\partial T}{\partial y}\right]=\frac{k}{C_p}\frac{\partial^2 T}{\partial y^2} \qquad \text{(Ec O.3)}$$

En la figura 62 se muestra una malla ortogonal de MxN elementos, el flujo entra en dirección $x+$ por la frontera oeste (nodos $1,j$) con una velocidad U_o y sale por la frontera este (nodos M,j) con componentes en x e y para la velocidad. El fluido limita en la frontera sur con una superficie plana fija, es decir $u=0$ y $v=0$ para los nodos $i,1$. La frontera norte (nodos i,N) se encuentra en la zona de flujo no viscoso[2]. El método explícito plantea las ecuaciones para el nodo $i+1,j$, de $i=1$ hasta $i=M-1$ y $j=2$ hasta $j=N$, hallando primero el valor de la velocidad $u_{i+1,j}$ y posteriormente el valor de $v_{i+1,j}$.

[1]WU, J. C. On the Finite Difference Solution of Laminar Boundary Layer Problems. En: Heat Transfer and Fluid Mechanics Institute, 1961.

[2]Como se explicará posteriormente, la frontera norte debe estar fuera de la capa límite para poder calcular la velocidad en todos los puntos; el valor de la velocidad en esta frontera se halla a partir del gradiente de la presión el cual es dado como dato de entrada.

Figura 62: Método explícito para calcular $u_{i+1,j}$ y posteriormente $v_{i+1,j}$.



En primer lugar se discretiza cada uno de los términos de la ecuación de cantidad de movimiento (Ec O.1) obteniendo (Ec O.4). Se utiliza el método UDS para discretizar dos términos, $\frac{\partial u}{\partial x}$ y $\frac{dp}{dx}$, en cambio, se usa el método CDS para $\frac{\partial u}{\partial y}$ y $\frac{\partial^2 u}{\partial y^2}$.

$$\frac{\partial u}{\partial x} = \frac{u_{i+1,j} - u_{i,j}}{\Delta x} \qquad \frac{\partial u}{\partial y} = \frac{u_{i,j+1} - u_{i,j-1}}{2\,\Delta y}$$

$$\frac{dp}{dx} = \frac{p_{i+1} - p_i}{\Delta x} \qquad \frac{\partial^2 u}{\partial y^2} = \frac{u_{i,j+1} - 2\,u_{i,j} + u_{i,j-1}}{(\Delta y)^2}$$

$$\rho u_{i,j} \frac{u_{i+1,j} - u_{i,j}}{\Delta x} + \rho v_{i,j} \frac{u_{i,j+1} - u_{i,j-1}}{2\,\Delta y} = -\frac{p_{i+1} - p_i}{\Delta x} + \mu\,\frac{u_{i,j+1} - 2\,u_{i,j} + u_{i,j-1}}{(\Delta y)^2} \qquad \text{(Ec O.4)}$$

El único valor desconocido de (Ec O.4) es $u_{i+1,j}$ y para hallarlo se usa el valor de $u_{i,j}$, $u_{i,j+1}$, $u_{i,j-1}$ y $v_{i,j}$. En (Ec O.5) se despeja el valor de $u_{i+1,j}$ de forma conveniente para introducir la variable $Q_{i,j}$.

$$u_{i+1,j} = -\frac{1}{\rho\,u_{i,j}}\left[p_{i+1} - p_i\right] + \left[\frac{\mu\,\Delta x}{\rho\,(\Delta y)^2 u_{i,j}} - \frac{v_{i,j}\Delta x}{2\,\Delta y\,u_{i,j}}\right]\,u_{i,j+1} +$$

$$+\left[\frac{\mu\,\Delta x}{\rho\,(\Delta y)^2 u_{i,j}}+\frac{v_{i,j}\Delta x}{2\,\Delta y\,u_{i,j}}\right]\,u_{i,j-1}-\left[\frac{2\Delta x\mu}{(\Delta y)^2}-u_{i,j}\right]$$

$$u_{i+1,j}=-\frac{p_{i+1}-p_i}{\rho\,u_{i,j}}-(2Q_{i,j}-1)\,u_{i,j}+Q_{i,j}(u_{i,j+1}+u_{i,j-1})-\frac{v_{i,j}}{u_{i,j}}\frac{\Delta x}{\Delta y}\left[\frac{u_{i,j+1}-u_{i,j-1}}{2}\right] \quad \text{(Ec O.5a)}$$

$$Q_{i,j}=\frac{\mu\,\Delta x}{\rho\,(\Delta y)^2\,u_{i,j}} \quad \text{(Ec O.5b)}$$

Una vez se ha calculado el valor de $u_{i,j+1}$ mediante (Ec O.5) se continua hallando el valor de $v_{i,j+1}$ y para esto se resuelve la ecuación de continuidad.

En la ecuación de continuidad (Ec O.6) se discretiza el término $\frac{\partial u}{\partial x}$ como una diferencia centrada en $i+1/2, j-1/2$ y el término $\frac{\partial v}{\partial x}$ como una diferencia centrada en $i+1, j-1/2$.

$$\frac{1}{2}\left[\frac{u_{i+1,j}-u_{i,j}}{\Delta x}+\frac{u_{i+1,j-1}-u_{i,j-1}}{\Delta x}\right]+\frac{v_{i+1,j}-v_{i+1,j-1}}{\Delta y}=0 \quad \text{(Ec O.6)}$$

De (Ec O.6) se despeja el valor de $v_{i+1,j}$ como se muestra en (Ec O.7)

$$v_{i+1,j}=v_{i+1,j-1}-\frac{1}{2}\frac{\Delta y}{\Delta x}\left[(u_{i+1,j}-u_{i,j})+(u_{i+1,j-1}-u_{i,j-1})\right] \quad \text{(Ec O.7)}$$

Se define $Q_{i,j}$ para explicar una de las dos **condiciones** para lograr la **estabilidad**[3] de los resultados. La primera condición se muestra en (Ec O.8a) de dos formas, como un límite al valor de $Q_{i,j}$ o como una restricción al máximo valor de Δx. Por otro lado, en (Ec O.8b) se define un valor máximo para Δy si la velocidad $v_{i,j}$ es positiva; la velocidad $v_{i,j}$ puede ser positiva si se utiliza un gradiente de presión adverso que cause la separación de la capa límite, de lo contrario, $v_{i,j}$ será siempre negativa.

$$Q_{i,j}<\frac{1}{2} \qquad \acute{o} \qquad \Delta x<\frac{1}{2}\frac{\rho\,u_{i,j}\,(\Delta y)^2}{\mu} \quad \text{(Ec O.8a)}$$

$$\Delta y<\frac{2\mu}{\rho\,v_{i,j}} \quad si \quad v_{i,j}>0 \quad \text{(Ec O.8b)}$$

La ecuación de conservación de la energía (Ec O.3) se puede discretizar de modo similar a la ecuación de cantidad de movimiento, el resultado se muestra en (Ec O.9).

$$\rho u_{i,j}\frac{T_{i+1,j}-T_{i,j}}{\Delta x}+\rho v_{i,j}\frac{T_{i,j+1}-T_{i,j-1}}{2\,\Delta y}=\frac{k}{C_p}\;\frac{T_{i,j+1}-2\,T_{i,j}+T_{i,j-1}}{(\Delta y)^2} \quad \text{(Ec O.9)}$$

De (Ec O.9) se despeja el valor de $T_{i+1,j}$ y se obtiene (Ec O.10a)

$$T_{i+1,j}=-\left[2E-1\right]\,T_{i,j}+\left[E-\frac{1}{2}\frac{v_{i,j}}{u_{i,j}}\frac{\Delta x}{\Delta y}\right]T_{i,j+1}+\left[E+\frac{1}{2}\frac{v_{i,j}}{u_{i,j}}\frac{\Delta x}{\Delta y}\right]T_{i,j-1} \quad \text{(Ec O.10a)}$$

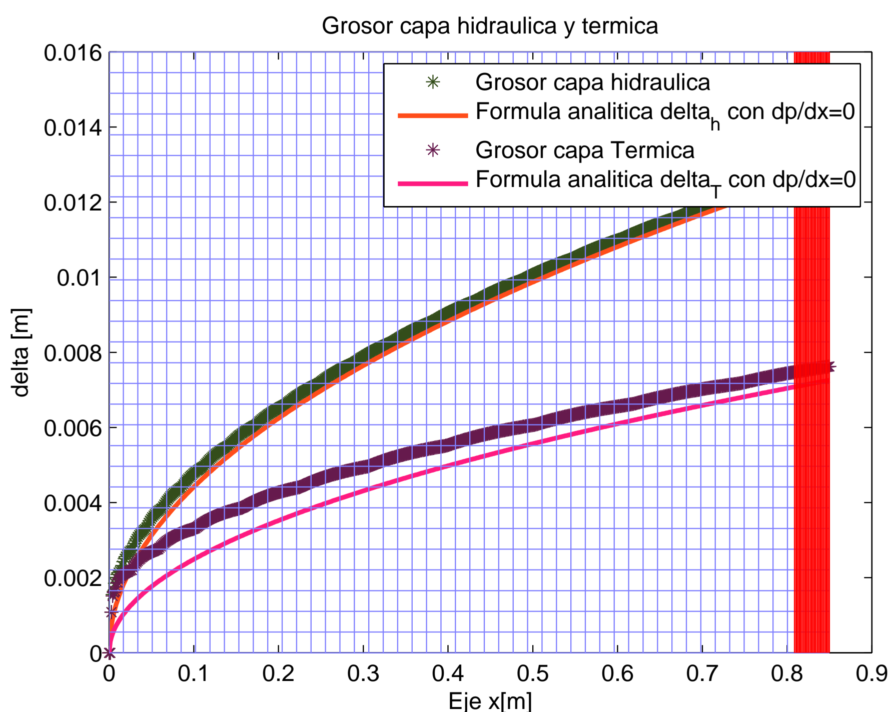
$$E=\frac{1}{\rho\,u_{i,j}}\frac{k}{C_p}\frac{\Delta x}{(\Delta y)^2} \quad \text{(Ec O.10b)}$$

[3] No se pretende explicar la forma de obtener las condiciones de estabilidad pues está fuera del alcance del presente proyecto.

IMPLEMENTACIÓN

Se crea la clase `climiteexplicito` encargada de resolver el problema y exportar los resultados a Matlab. Se da un ejemplo de las gráficas generadas por Matlab en la figura 63 en la cual se compara el valor teórico y el valor hallado del grosor de la capa límite hidráulica y térmica sobre una placa plana con propiedades constantes y sin gradiente de presión para el régimen laminar.

Figura 63: Resultados de un método explícito para hallar el grosor de la capa límite térmica e hidráulica sobre un placa plana



La clase tiene tres funciones públicas, `iniciar(...)`, `calcular()` y `expmatlab(...)`. `iniciar(...)` toma como argumentos las dimensiones de la placa `lon` y `alt`, el número de nodos en dirección x (`nx`) e y (`ny`), la viscosidad dinámica del fluido `mmu`, la densidad del fluido `rrho`, la conductividad térmica del fluido `kk`, el calor específico `ccp`, la velocidad de entrada del fluido `vv0`, la temperatura de la superficie o placa `ttsup` y finalmente, la temperatura de entrada del fluido `ttflu`.

Dentro de los resultados se muestra el campo de velocidad (figura 66) y temperatura en la figura 67, el grosor de la capa límite hidráulica y térmica en la figura 63, un informe de la revisión de las condiciones de estabilidad para el análisis en la figura 65 y la dirección del flujo del fluido en la figura 64.

Figura 64: Dirección del flujo del fluido.

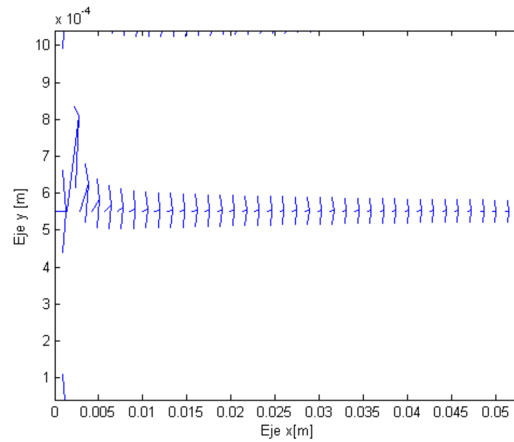
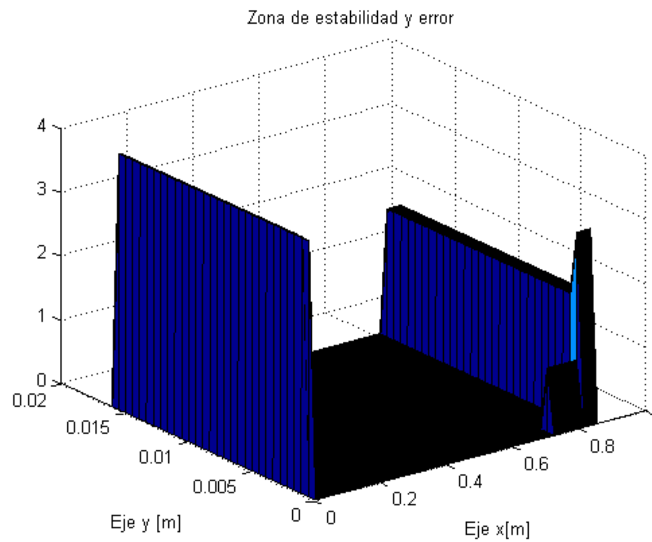


Figura 65: Verificación de la estabilidad del análisis.



Todo valor diferente de cero implica algún tipo de advertencia.

Figura 66: Campos de las componentes x e y de la velocidad del fluido.

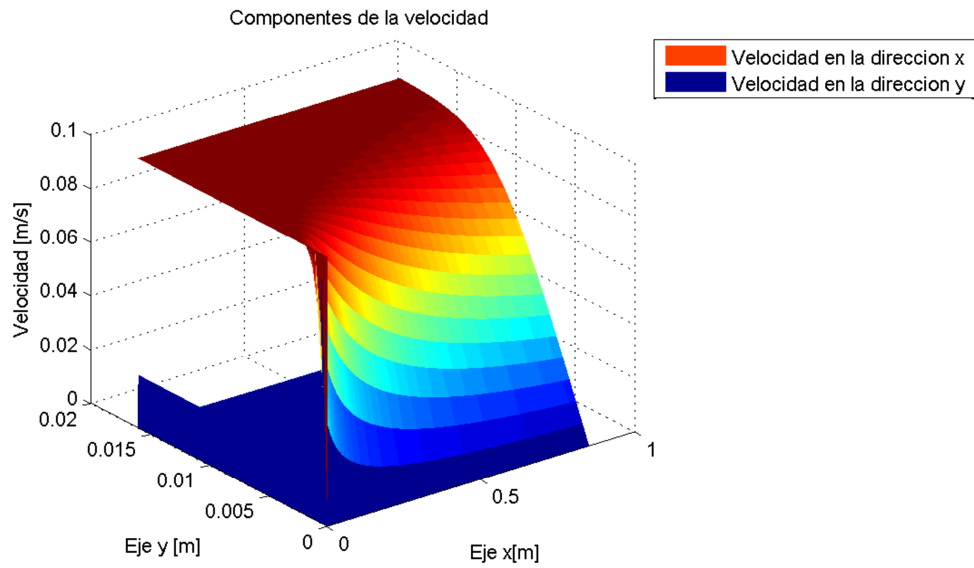
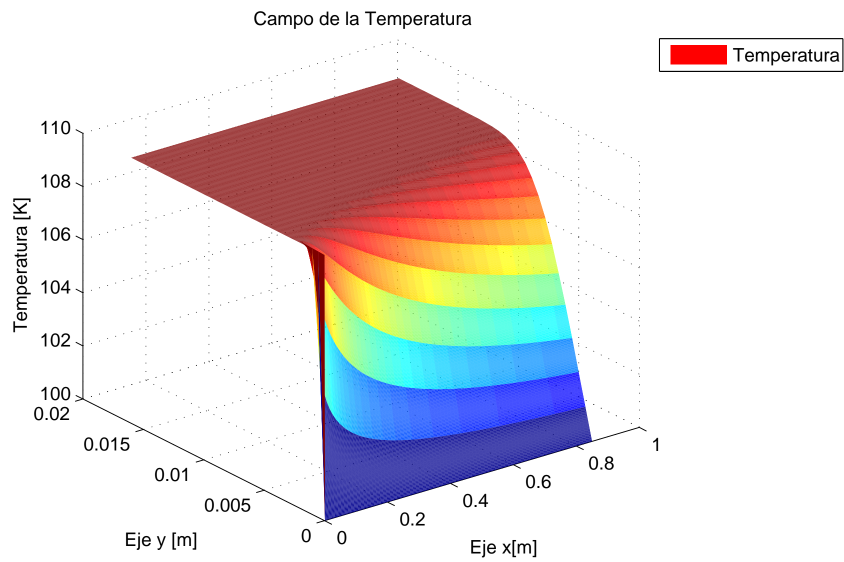


Figura 67: Campo de temperatura para el fluido.



ANEXO P. CLASE PARA LA SOLUCIÓN EXPLICITA DE LAS ECUACIONES DE LA CAPA LÍMITE: ENCABEZADO DE LA CLASE

```
//Archivo climiteexplicito.h
#pragma once

#include   <CampoDifFin.h>
#include   <prmTiempo.h>

class climiteexplicito
{

protected:
 Os  salida;  //guardar datos, mostrar datos, etc
 Boolean vinculado;   // verificar el estado de salida
 int   lx;  //numero elementos en x
 int   ly;  //numero elementos en y
 int  transicion; //se almacena el valor del indice en el cual se
  deja de estar en la zona laminar
 real   longitud; //total de la longitud en el eje x
 real   altitud ; //total de la altura en el eje y para el mallado,
   la verdadera altura del liquido es variable
 real    tempsup;    //temperatura de la superficie, por ahora se
  asume isotermica
 real    tflu; //temperatura inicial del fluido
 real   v0; //velocidad incial del fluido
 Puntero<MallaCuadricula> malla;
 Puntero<MallaCuadricula> mallax;// malla unidimensional para los
  elementos que solo dependen de x
 CampoDifFin   u  ;
 CampoDifFin   v  ;
 CampoDifFin   p  ;
 CampoDifFin   T  ;
 CampoDifFin   estabilidad  ;//verifica si se cumplio la condicion
  de estabilidad en cada punto ademas verifica si la capa limite es
   inferior a la altura de la simulacion
 CampoDifFin   DeltaHidr  ;//grosor capa hidraulica
 CampoDifFin   DeltaTerm  ;//grosor capa termica
 //**************PROPIEDADES DEL FLUIDO**************//
```

```
 real mu; //viscosidad dinamica
 real c_p; //calor especifico a presion constante
 real rho; //densidad
 real k;  //conductividad termica
public:
 climiteexplicito(void);
 ~climiteexplicito(void);
 void iniciar ( real lon = 0.85, real alt =0.016, int nx =600,
  int ny = 30, real mmu = 0.008, real rrho = 990.0, real kk = 0.6,
  real ccp = 4180, real vv0 = 0.1,
    real ttsup = 100.0, real ttflu = 110.0);
 void    calcular(void);
 void   expmatlab(Cadena Nombre);


};
```

ANEXO Q. CLASE PARA LA SOLUCIÓN EXPLICITA DE LAS ECUACIONES DE LA CAPA LÍMITE: ARCHIVO DE CÓDIGO FUENTE

```
//Archivo climiteexplicito.cpp
#include "climiteexplicito.h"
climiteexplicito::climiteexplicito(void)   {    vinculado=0; }
climiteexplicito::~climiteexplicito(void)  {}

void climiteexplicito::iniciar( real lon, real alt, int nx, int ny,
  real mmu, real rrho, real kk, real ccp, real vv0,
  real ttsup, real ttflu)
{
 Vector_xyz <int> ixy(2); Vector_xyz <real> xy(2);
 //variables para llamar a los puntos y obtener la posicion
 if(!vinculado) { salida.vincular(std_o);  vinculado=1; }

 //variables geometricas
 longitud   = lon; altitud = alt;
 lx=nx; ly=ny; mu = mmu;
 transicion = lx+10;   //se inicia de esta manera para verificar
 //facilmente si se debe advertir por flujo no laminar
 rho = rrho; k = kk; c_p =ccp;
 v0 = vv0;//velocidad inicial en [m/s]
 tempsup = ttsup; tflu = ttflu;// temperatura de la superficie y a
  la entrada del fluido

 malla.vincular(new MallaCuadricula(2));
 mallax.vincular(new MallaCuadricula(1));
 malla->     Leer(
     aform( "d=2 dominio = [0,%10g]x[0,%10g] indices= [1: %5d]x
      [1:%5d]", longitud , altitud , lx , ly )
          );//AFORM ES MAS SEGURO QUE OFORM
 u.chaSize(malla(),"Velocidad en la direccion x"); u.llenar(0.0);
 v.chaSize(malla(),"Velocidad en la direccion y"); v.llenar(0.0);
 T.chaSize(malla(),"Temperatura");           T.llenar(tflu);
 estabilidad.chaSize(malla(),"Zona de estabilidad y error ");
  estabilidad.llenar(0.0);
```

```
 mallax->    Leer(//malla UNIDIMENSIONAL
     aform( "d=1 dominio = [0,%10g] indices= [1: %5d]", longitud ,
      lx )
          );
35  DeltaHidr.chaSize(mallax(),"Grosor capa hidraulica");  DeltaHidr.
  llenar(0.0);
 DeltaTerm.chaSize(mallax(),"Grosor capa Termica");  DeltaTerm.
  llenar(0.0);
 p.        chaSize(mallax(),"Presion              ");          p.
  llenar(100.0);

 ixy(1)= 1;
40  for(int i=1;i<=ly;i++)
 {
  ixy(2)= i;
  u.valorIndice(ixy)=v0; //velocidad inicial en la direccion x,
   condicion de entrada del fluido
 }
45
}

void    climiteexplicito::calcular(void)
{
50  int i, j;
 real param; real  Rex; //parametro de estabilidad, reynolds x
 real Q,E;
/////////////////////////////////////////////////////////////
///////////////CALCULO DE LA VELOCIDAD DE ENTRADA/////////
55 /////////////////////////////////////////////////////////////
 for ( i=1; i<=ly; i++)
 {
  u.valorIndice(1,i)=v0;
 }
60 /////////////////////////////////////////////////////////////
//////CALCULO DE LAS COMPONENTE DE LA VELOCIDAD EXTERNA////
/////////////////////////////////////////////////////////////
 for ( i = 2; i <=lx ; i++)
 {
65   u.valorIndice(i, ly)=  u.valorIndice(i-1, ly)   -
    (   p.valorIndice(i)-p.valorIndice(i-1)    ) /
     ( rho*u.valorIndice(i-1, ly )  );
 }
/////////////////////////////////////////////////////////////
70 ////////CALCULO DE LA TEMPERATURA DE LA PLACA/////////////
/////////////////////////////////////////////////////////////
 for ( i=2; i<=lx; i++)
```

```
   {
    T.valorIndice(i,1)=tempsup;
75 }
///////////////////////////////////////////////////////////
////////////CALCULO DE LAS COMPONENTE DE LA VELOCIDAD//////
///////////////////////////////////////////////////////////
 for(i=1;i<=lx-1;i++)
80 {
  for(j=2;j<=ly-1;j++) //iniciar en 2 y terminar en ly-1
  {
   Q = mu * malla->Delta(1) / (rho*u.valorIndice(i , j)*pow(malla->
    Delta(2),2.0));
   u.valorIndice(i+1 , j) =
85   Q * (  u.valorIndice(i , j+1) + u.valorIndice(i , j-1)  )
    - (2*Q -1) *u.valorIndice(i , j)
   -( p.valorIndice(i+1)-p.valorIndice(i) )/( rho*u.valorIndice(i ,
    j ) )
   -(v.valorIndice(i ,j )/u.valorIndice(i ,j ))*(malla->Delta(1)/
    malla->Delta(2))*
   ( (u.valorIndice(i ,j+1)-u.valorIndice(i , j-1))/(2.0) )
90 ;
  }
  for(j=2;j<=ly;j++) //iniciar en 2 y terminar en ly, NO HASTA ly-1
  {
   v.valorIndice(i+1,j ) = v.valorIndice(i+1,j-1)
95 - (malla->Delta(2)/(2*malla->Delta(1)))*
   ( u.valorIndice( i+1, j)- u.valorIndice( i, j) +
     u.valorIndice( i+1,j-1)-u.valorIndice(i  ,j-1)
   )
   ;
100 }

 }
///////////////////////////////////////////////////////////
////////////CALCULO DE LA TEMPERATURA/////////////////////
105 ///////////////////////////////////////////////////////////
 for(i=1;i<=lx-1;i++)
 {
  for(j=2;j<=ly-1;j++) //iniciar en 2 y terminar en ly-1
  {
110  E = k * malla->Delta(1) /
    (rho* u.valorIndice(i , j)* c_p*pow(malla->Delta(2),2.0));
   param = 0.5* ( v.valorIndice(i , j) / u.valorIndice(i , j))*
    (malla->Delta(1) / malla->Delta(2));
   T.valorIndice(i+1 , j) =
115  -(2.0*E-1.0) *  T.valorIndice(i, j )
```

```
    + (E-param)* T.valorIndice(i , j+1)
    + (E+param)* T.valorIndice(i , j-1);
  }
  //std_o<<"T="<<T.valorIndice(i , 2);
120 }

//////////////////////////////////////////////////////////////
////////CALCULO DE LA ZONA DE ESTABILIDAD Y OTROS ERRORES//
//////////////////////////////////////////////////////////////
125 for(i=2;i<=lx;i++)
  {
   Rex=rho*u.valorIndice(i,ly)*malla->obtPunto(1,i)/mu;
   for(j=2;j<=ly-1;j++)
   {
130    param=0.5*rho*u.valorIndice(i ,j )*pow(malla->Delta(2),2.0)/mu;
    if(malla->Delta(1)>=param)  //  if(\Delta x >permitido)
    { estabilidad.valorIndice(i  ,j )=1;  }  //1->error de
     inestabilidad
    if(Rex>500000)     //si esta fuera del regimen de laminar
    { estabilidad.valorIndice(i  ,j )+=2; if(transicion==lx+10) {
     transicion=i;}; }
135    param= 2.0*mu/ (rho*v.valorIndice(i,j));
    if(v.valorIndice(i,j)>0 &&  malla->Delta(2) >param ) //if(\
     Delta y >permitido)
    { estabilidad.valorIndice(i  ,j )+=4; }
   }
   if(u.valorIndice(i  ,ly-1) < 0.99* u.valorIndice(i  ,ly  ))   //
    if(grosor_capa_limite>altura_simulacion)
140   {estabilidad.valorIndice(i  ,ly-1)+= 8;  estabilidad.valorIndice
    (i  ,ly  )+= 8;  }//+8-> error por grosor de la capa limite
    hidraulica
 }
//////////////////////////////////////////////////////////////
////////////////CALCULO DE LA CAPA LIMITE HIDRAULICA///////
//////////////////////////////////////////////////////////////
145 //debido al comportamientode la capa limite similar a sqrt(x) es
 mas eficiente buscar el grosor de la
//capa limite desde ly en vez de comenzar desde j=1;
 for(i=2;i<=lx;i++)
 {
  j= ly - 1 ;
150  while(u.valorIndice(i,j) > 0.99* u.valorIndice(i  ,ly  )) j--;
  //mientra la velocidad sea mayor a 99%
  if(j>1)
  {
```

```
   DeltaHidr.valorIndice(i)=( (malla->Delta(2))/(u.valorIndice(i,j
    +1)-u.valorIndice(i,j)) )
155    * (  0.99*u.valorIndice(i  ,ly  )-u.valorIndice(i,j)  )
    + malla->obtPunto(2,j)
    ;//interpolacion para hallar la capa limite hidraulica
  }

160 }
 //por otro lado, se puede buscar la posicion en la cual se
  encuentra la capa limite a partir
 //de la posicion de la capa en el punto anterior (en x), el
  problema es que si la solucion es
 //inestable, la forma de hallar la capa limite tambien
///////////////////////////////////////////////////////////
165 ////////////////CALCULO DE LA CAPA LIMITE TERMICA//////////
///////////////////////////////////////////////////////////
 for(i=2;i<=lx;i++)
 {
  j= ly - 1 ;
170  while(abs(T.valorIndice(i,j)-tempsup) > 0.99* abs (T.valorIndice(
   i  ,ly  ) - tempsup)) j--;
  //mientra la temperatura sea mayor al 99%
  if(j>1)
  {
   DeltaTerm.valorIndice(i)=( (malla->Delta(2))/(T.valorIndice(i,j
    +1)-T.valorIndice(i,j)) )
175    * abs(  0.99*(T.valorIndice(i  ,ly  )-tempsup) -
    (T.valorIndice(i,j)-tempsup)  )
    + malla->obtPunto(2,j)
    ;
  }
180
 }
}
void   climiteexplicito::expmatlab(Cadena Nombre)
{
185 int i,j; real param;
 Cadena NombreArch;
 NombreArch=Nombre;
 if(!Nombre.contiene("ARCH="))  NombreArch="ARCH=" + Nombre;
 Os exportar(NombreArch);
190 exportar<<"%Metodo numerico explicito para la simulacion de la
  capa limite\n"
  <<"%Parametros de entrada\n% Fluido:";
 exportar<<"mu="<<mu<<" ;k="<<k<<" ;rho="<<rho<<" ;cp="<<c_p<<";";
 exportar << "\n%Mallado: "
```

```
 <<  aform( "d=2 dominio = [0,%10g]x[0,%10g] indices= [1: %5d]x
  [1:%5d]",
195   longitud , altitud , lx , ly )<<"\n\n";
exportar << "'Posicion x',Px=zeros(1,"<<lx<<");\n" ;
exportar<<"Px=["<<   malla->obtPunto(1,1);
for(i=2;i<=lx;i++)   exportar << "," <<malla->obtPunto(1,i);
exportar << "];\n";
200 exportar << "P2x=zeros("<<ly<<","<<lx<<");\n" ;
exportar << "for indice=1:" << ly << "\n  P2x(indice,:)=Px;   \n
 end\n";

exportar << "'Posicion y',Py=zeros(1,"<<ly<<");\n" ;
exportar<<"Py=["<<   malla->obtPunto(2,1);
205 for(j=2;j<=ly;j++)   exportar << "," <<malla->obtPunto(2,j);
exportar << "];\n";
exportar << "P2y=zeros("<<ly<<","<<lx<<");\n" ;
exportar << "for indice=1:" << lx << "\n  P2y(:,indice)=Py';  \n
 end\n";

210 exportar << "\n'" << u.obtNombreCampo() << "',u=zeros("<<ly<<","<<
 lx<<");\n" ;
for(j=1;j<=ly;j++)
 {
  exportar<<"u("<<j<<",:)=["<<u.valorIndice(1,j);
  for(i=2;i<=lx;i++) exportar << "," << u.valorIndice(i,j);
215   exportar << "];\n";
 }

exportar << "\n'" << v.obtNombreCampo() << "',v=zeros("<<ly<<","<<
 lx<<");\n" ;
for(j=1;j<=ly;j++)
220  {
  exportar<<"v("<<j<<",:)=["<<v.valorIndice(1,j);
  for(i=2;i<=lx;i++) exportar << "," << v.valorIndice(i,j);
  exportar << "];\n";
 }
225 exportar << "\n'" << T.obtNombreCampo() << "',T=zeros("<<ly<<","<<
 lx<<");\n" ;
for(j=1;j<=ly;j++)
 {
  exportar<<"T("<<j<<",:)=["<<T.valorIndice(1,j);
  for(i=2;i<=lx;i++) exportar << "," << T.valorIndice(i,j);
230   exportar << "];\n";
 }
exportar << "\n'" << p.obtNombreCampo() << "',Presion=zeros(1,"<<
 lx<<");\n" ;
```

```
exportar << "Presion=[" <<   p.valorIndice(1);
for(i=2;i<=lx;i++)   exportar << "," << p.valorIndice(j);
exportar << "];\n";

exportar << "\n'" << estabilidad.obtNombreCampo() << "',
 estabilidad=zeros("<<ly<<","<<lx<<");\n" ;
for(j=1;j<=ly;j++)
 {
  exportar << "estabilidad("<<j<<",:)=[" << estabilidad.
   valorIndice(1,j);
  for(i=2;i<=lx;i++) exportar << "," << estabilidad.valorIndice(i,
   j);
  exportar << "];\n";
 }

exportar << "\n'" << DeltaHidr.obtNombreCampo() << "',DeltaHidr=
 zeros(1," << lx << ");\n\n" ;
exportar << "DeltaHidr=[" <<   DeltaHidr.valorIndice(1);
for(i=2;i<=lx;i++)   exportar << "," << DeltaHidr.valorIndice(i);
exportar << "];\n";
exportar << "\n'" << DeltaTerm.obtNombreCampo() << "',DeltaTerm=
 zeros(1," << lx << ");\n\n" ;
exportar << "DeltaTerm=[" <<   DeltaTerm.valorIndice(1);
for(i=2;i<=lx;i++)   exportar << "," << DeltaTerm.valorIndice(i);
exportar << "];\n";

//graficar campo velocidad
exportar<< "\nfigure(1); title('Campo de Velocidad')\n"
 <<"quiver(P2x,P2y,u,v,'b');%hold on;\naxis([0 "<<longitud<<" 0 "
  <<altitud<<"]);"
 <<"xlabel('Eje x[m]');ylabel('Eje y [m]');\n";

exportar<< "\nfigure(2); surf(Px,Py,estabilidad);%hold on;\n title
 ('"
 << estabilidad.obtNombreCampo()<<"')\n"
 <<"xlabel('Eje x[m]');ylabel('Eje y [m]');\n";
param = mu*c_p/k;
param= pow(param,1.0/3.0) ;
param = param*pow(v0*rho/mu,1.0/2.0);
exportar<< "\nfigure(3); hold off;plot(Px,DeltaHidr,'*','color
 ',[0.2 0.3 0.1]);hold on;\n"
 <<"xlabel('Eje x[m]');ylabel('\delta [m]'); title('Grosor capa
  hidraulica y termica')\n"
 <<"plot(Px,5.0*sqrt("<<mu/rho<<"/"<<v0<<")*sqrt(Px),'color',[1
  0.3 0.1],'LineWidth',2)\n"
 <<"plot(Px,DeltaTerm,'*','color',[0.4 0.1 0.3]);hold on;\n"
```

```
 <<"plot(Px,5.0*(1/"<<param<<")*Px.^(0.5),'color',[1 0.1 0.5],'
  LineWidth',2)\n"
 <<"legend('"<<DeltaHidr.obtNombreCampo()<<"','Formula analitica \
  delta_h con dp/dx=0','"
 <<DeltaTerm.obtNombreCampo()<<"','Formula analitica \delta_T con
  dp/dx=0')\n";
exportar<<"for indice=1:"<<ceil(ly/50.0)<<":"<<ly<<"; plot(Px,Py(
 indice)+0*Px,'color',[0.5 0.5 1]);end;\n";
if(transicion<lx)
{
 exportar<<"for indice=1:"<<ceil(lx/50.0)<<":"<<transicion-1<<";
  plot(Px(indice)+0*Py,Py,'color',[0.5 0.5 1]);end;\n";
    exportar<<"for indice="<<transicion<<":"<<lx<<"; plot(Px(
     indice)+0*Py,Py,'color',[1 0 0]);end;\n";
}
else
{
 exportar<<"for indice=1:"<<lx<<"; plot(Px(indice)+0*Py,Py,'color
  ',[0.5 0.5 1]);end;\n";
}
exportar<< "\nfigure(4); hold off ; surf(Px,Py,u) ;\n"
 <<"title('Componentes de la velocidad'); hold on ; surf(Px,Py,v);
  shading flat; \n"
 <<"xlabel('Eje x[m]');ylabel('Eje y [m]'); zlabel('Velocidad [m/s
  ]')\n"
 <<"legend('"<<u.obtNombreCampo()<<"','"<<v.obtNombreCampo()<<"')\
  n";
exportar<< "\nfigure(6); hold off ; surf(Px,Py,T) ;shading flat;\n
 "
 <<"title('Campo de la Temperatura'); hold on;\n"
 <<"xlabel('Eje x[m]');ylabel('Eje y [m]'); zlabel('Temperatura [K
  ]')\n"
 <<"legend('"<<T.obtNombreCampo()<<"')\n";
exportar->actualizar();  exportar->cerrar();

}
```