MODELADO Y SIMULACION 2D DEL FLUJO DE UN FLUIDO EN UN CANAL CON UNA EXPANSIÓN INCLINADA MEDIANTE VOLÚMENES FINITOS

KATHERINE LISETH MARTINEZ RUEDA

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERÍAS FISICOMECÁNICAS
ESCUELA DE INGENIERÍA MECÁNICA
BUCARAMANGA
2014

MODELADO Y SIMULACION 2D DEL FLUJO DE UN FLUIDO EN UN CANAL CON UNA EXPANSIÓN INCLINADA MEDIANTE VOLÚMENES FINITOS

Presentado por:

KATHERINE LISETH MARTINEZ RUEDA

Trabajo de grado como requisito para optar el título de

Ingeniero Mecánico

Director:

Ph.D. JULIAN ERNESTO JARAMILLO IBARRA

UNIVERSIDAD INDUSTRIAL DE SANTANDER

FACULTAD DE INGENIERÍAS FISICOMECÁNICAS

ESCUELA DE INGENIERÍA MECÁNICA

BUCARAMANGA

2014

DEDICATORIA

Este trabajo lo dedico a mis padres, Alfonso Martínez y Nancy Rueda, por todo el cariño y apoyo incondicional que siempre me han brindado, por su paciencia y sus enseñanzas, gracias a ellos me he convertido en la persona que soy.

A mi hermano Alfonso Daniel que toda la vida me ha acompañado, por los momentos, las peleas, los juegos y las risas.

A Andrés por estar a mi lado desde el comienzo de esta etapa, por su cariño y por animarme a salir adelante cada día.

A toda mi familia que siempre me ha acompañado, que ha compartido conmigo las alegrías y tristezas, este logro es de todos.

AGRADECIMIENTOS

A la Universidad Industrial de Santander, especialmente a la Escuela de Ingeniería Mecánica, por su formación académica.

Al profesor Julian Jaramillo por el tiempo dedicado, por su acompañamiento y asesoría en el desarrollo del proyecto.

CONTENIDO

LISTA DE TABLAS

LISTA DE FIGURAS

LISTA DE ANEXOS

RESUMEN

TÍTULO: MODELADO Y SIMULACIÓN 2D DEL FLUJO DE UN FLUIDO EN UN CANAL CON UNA EXPANSIÓN INCLINADA MEDIANTE VOLÚMENES FINÍTOS*

AUTOR: KATHERINE LISETH MARTINEZ RUEDA**

PALABRAS CLAVE: Dinámica de fluidos computacional**,** MVF**,** Método del paso fraccionado, Canal con expansión, Separación de flujo, Recirculación de flujo.

En este trabajo se presenta el planteamiento numérico para la solución de las ecuaciones de conservación de cantidad de movimiento y de masa de Navier-Stokes para un flujo incompresible; la ecuación general de transporte se discretiza mediante el método de volúmenes finitos, y se hace uso del método del paso fraccionado para la resolución del acoplamiento entre la velocidad y la presión. Se utiliza un mallado no estructurado para la discretización física del dominio, el cual se realiza mediante la plataforma Salome.

El planteamiento se implementa en un código computacional en lenguaje C++ utilizando las rutinas de PETSc (Portable, Extensible Toolkit for Scientific Computation).

Con el propósito de validar el planteamiento y el código desarrollado se resuelven dos casos de estudio bidimensionales: el problema conocido como Smith-Hutton y el problema conocido como Driven Cavity, sus resultados se comparan con los encontrados en la literatura y resultan satisfactorios.

Se desarrolla un modelo numérico para el problema del flujo laminar bidimensional en un canal con una expansión inclinada, y se resuelve para números de Reynolds de 50 y 100, una relación de expansión de 1,9423 y ángulos de inclinación de 15°, 30°,45° y 90°. El modelo se valida comparando los resultados encontrados para el ángulo de 90° con resultados de la literatura para el problema de escalón recto.

* Trabajo de grado.

** Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingeniería Mecánica. Director: PhD Julian Ernesto Jaramillo Ibarra.

ABSTRACT

TITLE: 2D MODELLING AND SIMULATION OF FLUID FLOW IN A CHANNEL WITH INCLINED BACKWARD-FACING STEP BY FINITE VOLUME METHOD [*]

AUTHOR: KATHERINE LISETH MARTINEZ RUEDA[**]

KEY WORDS: CFD, FVM**,** FSM, Backward-facing step, Flow separation, Recirculating flow.

In this paper the numerical approach for the solution of the Navier-Stokes conservation equations of momentum and mass for an incompressible flow is presented. The general transport equation is discretized by the finite volume method and using the fractional step method to solve the coupling between velocity and pressure. The physical domain is discretized by unstructured meshing, the Salome platform is used to create the mesh.

The numerical approach is implemented in a computer code in C++ language using PETSc (Portable, Extensible Toolkit for Scientific Computation) routines.

In order to validate the approach and the developed code two study cases are solved in two dimensions: the Smith -Hutton problem and the Driven Cavity problem, the results are compared with those found in the literature and they are found satisfactory.

A numerical model for the two-dimensional laminar problem of flow in a channel with an inclined backward-facing step is developed and solved for Reynolds numbers 50 and 100, an expansion ratio of 1.9423 and inclination angles of 15°, 30 °, 45 ° and 90 °. The model is validated by comparing the results for the 90° angle with results from the literature for the backward-facing step problem.

[*] Thesis.

[**] Physical-Mechanic Engineering Faculty. Mechanical Engineering School. Director: PhD Julian Ernesto Jaramillo Ibarra.

INTRODUCCIÓN

El fenómeno de separación de flujo que se presenta en el problema del canal con expansión es de gran interés en estudios numéricos y experimentales debido a que ocurre en una gran variedad de equipos de aplicaciones ingenieriles afectando la transferencia de calor y el flujo en los mismos. La geometría del escalón recto es la más simple de este problema, y es por esta razón que es la más estudiada, sin embargo, la geometría inclinada tiene mayor aplicabilidad industrial.

El problema del flujo puede ser estudiado numéricamente al describirlo en un modelo matemático utilizando las ecuaciones gobernantes de los fluidos. El método de los volúmenes finitos permite realizar una discretización de estas ecuaciones teniendo en cuenta los principios de conservación, obteniendo un sistema de ecuaciones algebraicas que se pueden resolver por diferentes métodos de solución.

La implementación de estas ecuaciones en un código computacional resulta ser una herramienta eficiente para encontrar la solución, estas técnicas numéricas han evolucionado en gran escala y continúan haciéndolo actualmente.

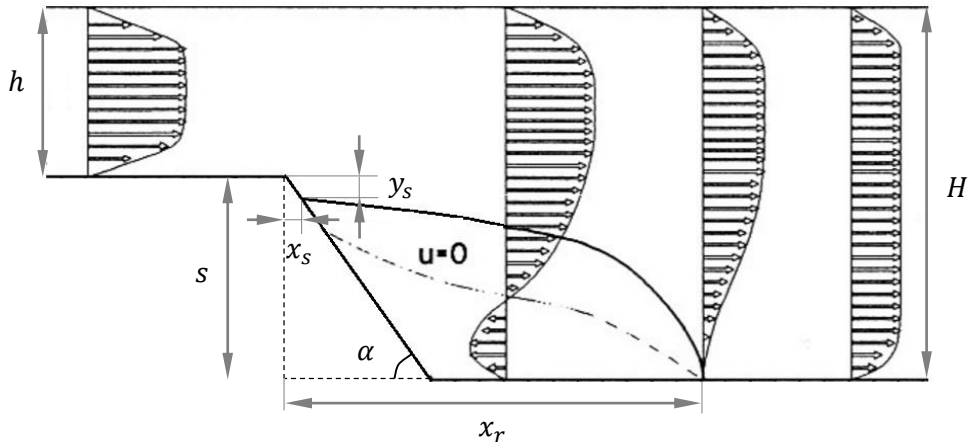
Este trabajo pretende aportar al desarrollo de estas técnicas numéricas mediante la modelación y solución del problema planteado. El documento se organiza de la siguiente forma:

Inicialmente se presentará la descripción del problema principal del trabajo, el canal con expansión inclinada, se dará una breve reseña de los estudios presentes en la literatura y se explicará el modelo desarrollado. En el capítulo 2 se introduce el método de los volúmenes finitos, y se explican los pasos a seguir para su uso. En los siguientes capítulos se desarrollan los pasos del MVF para resolver el problema en estudio, incluyendo la implementación en el código computacional de cada uno. Y, finalmente, se presentan los resultados numéricos obtenidos.

1. FLUJO EN UN CANAL CON EXPANSIÓN

El estudio bidimensional del flujo en un canal que experimenta un cambio brusco en su geometría se basa en el fenómeno de la separación del fluido; el cual consiste en un desprendimiento de la capa límite debido a la existencia de gradientes adversos de presión, que da lugar a zonas de separación y recirculaciones del flujo. Este fenómeno ocurre en una gran variedad de aplicaciones ingenieriles como: equipos de generación de potencia, difusores, flujo en los álabes de una turbina, equipos de combustión, tubos y ductos con cambio repentino del área. En la región de recirculación ocurre una mezcla de fluido con alta y baja energía, la cual ocasiona un impacto significativo en el desempeño del flujo y la transferencia de calor en estos equipos; por esta razón resulta de gran importancia conocer la ubicación de esta zona(s).

Figura 1. Flujo en un canal con escalón inclinado.



Cuando el cambio en la geometría consiste en un escalón inclinado (Inclined Backward-Facing Step. Fig. 1) el fenómeno varía en función de tres parámetros: el ángulo de inclinación del escalón α; la relación de expansión H/h; y el número de Reynolds $Re_D = \frac{\rho U_b D}{\mu}$, donde ρ y μ son la densidad y la viscosidad dinámica del fluido, respectivamente; D es el diámetro hidráulico del canal a la entrada

equivalente a $D = 2h$; y U_b es la velocidad promedio del flujo a la entrada, la cual, para el caso laminar, corresponde a dos tercios de la velocidad máxima a la entrada.

El problema consiste en encontrar el punto en el que se produce la separación del flujo del contorno, denominado punto de separación, y el punto en el que el flujo se vuelve a unir, denominado punto de re-encuentro. La distancia horizontal entre estos dos puntos, es conocida como la longitud de separación o longitud de unión.

1.1 ESTUDIOS RELACIONADOS

El interés en este tipo de problemas ha aumentado desde el estudio numérico y experimental publicado por Armaly et al. [14], donde se presenta una investigación detallada en la geometría de un canal con expansión recta ($\alpha = 90°$) para una relación de expansión $H/h = 1.9423$, y número de Reynolds entre $70 < Re_D < 8000$; se demuestra que el flujo presenta características tridimensionales para números de Reynolds cercanos y mayores a 400, y que deja de ser laminar a partir de $Re_D = 1200$. Los resultados presentados, además de determinar la zona de recirculación adyacente al escalón, muestran cómo se crean otras regiones de separación del flujo aguas abajo del escalón a ambos lados del canal (para $Re_D > 400$). Además del estudio experimental, también se presentan resultados del estudio numérico bidimensional del flujo para $Re_D < 1250$ y son comparados con los hallados experimentalmente, se observa que para $Re_D > 400$ comienzan a presentarse inconsistencias entre ambos resultados, lo cual se atribuye a la tridimensionalidad.

Muchos autores han publicado resultados experimentales y numéricos confirmando lo hallado por Armaly, y generando nuevas conclusiones principalmente en cuanto a la tridimensionalidad y la inestabilidad del problema.

G. Biswas et. al. [16] (por ejemplo) presenta resultados numéricos del flujo en canal recto para números de Reynolds entre $10^{-4} < Re_D < 800$; utiliza el método de volúmenes finitos para la discretización de las ecuaciones, el algoritmo SIMPLE

para el procedimiento de solución y emplea un esquema “Multigrid” para acelerar la convergencia de la solución. Los resultados obtenidos concuerdan con la investigación de Armaly, y confirman la bidimensional del flujo para $Re_D < 400$; también se proporcionan resultados del flujo para diferentes relaciones de expansión.

Un estudio presentado por Kim y Moin [9] utiliza una variación del método del paso fraccionado para llevar a cabo la simulación numérica del flujo laminar en el canal como problema dependiente del tiempo; sus resultados hasta $Re_D = 500$ se asemejan a los hallados experimentalmente, y a partir de ahí difieren significativamente, la causa es atribuida a la tridimensionalidad expuesta por Armaly.

En cuanto al canal con escalón inclinado, son menos las investigaciones que existen al respecto a pesar de ser una geometría con mayor aplicabilidad industrial.

Ruck y Makiola [18] proporcionan resultados experimentales del flujo en canales con expansión inclinada para ángulos de inclinación entre 10° y 90°, y flujo turbulento $Re_D > 5000$; demuestran que para un número de Reynolds constante, la longitud de separación disminuye al disminuir el ángulo de inclinación: en el rango de ángulos menores de 45° la longitud disminuye rápidamente conforme disminuye el ángulo, sin embargo en el rango de 45°-90°los cambios en la longitud de separación no son muy significantes.

Y.T. Chen et. al. [17] realiza simulaciones del flujo tridimensional en canales con expansión inclinada para $Re_D = 345$ y ángulos de 15°, 30°, 45° y 90°; las ecuaciones son resueltas numéricamente utilizando el método de volúmenes finitos, y el algoritmo SIMPLE. Sus resultados muestran también como disminuye la longitud de separación al disminuir el ángulo de inclinación.

Otros autores han realizado investigaciones del flujo en canales con expansión inclinada enfocándose en el efecto de la separación del flujo en la transferencia de calor. Por ejemplo, Gandjalikhan Nassab et. al. [19][20] estudian la generación de entropía convección forzada en el flujo laminar ($Re_D = 500$) adyacente a un canal

con expansión inclinada ($\alpha = 45°$) y su influencia en la distribución del número de Nusselt en el canal. Utilizan el método de mallado en bloque para la discretización física del dominio.

Sin embargo, a la fecha no se han publicado resultados del estudio del flujo laminar bidimensional en un canal con escalón inclinado para varios ángulos de inclinación.

1.2 MODELO MATEMÁTICO

Cuando se quiere simular el flujo de un fluido, la transferencia de calor, transferencia de masa u otro fenómeno físico, es necesario describir este fenómeno en términos matemáticos. Los principios o leyes que gobiernan la mecánica de fluidos se pueden expresar en forma de ecuaciones diferenciales, conocidas como ecuaciones gobernantes: Ecuación de conservación de masa, ecuación de conservación de energía, ecuación de conservación de cantidad de movimiento. Cada una de estas expresa el principio de conservación de la magnitud o propiedad a la que hace referencia, y puede ser escrita de forma conservativa o no conservativa. En este trabajo se utiliza la forma conservativa de las ecuaciones que corresponden al enfoque de Euler, en el cual la propiedad en estudio se determina en un punto fijo en el espacio [3].

Ecuación de conservación de masa:

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

Ecuación de conservación de energía:

$$\frac{\partial (\rho h)}{\partial t} + \nabla \cdot (\rho \mathbf{V} h) = \nabla \cdot \left(\frac{k}{C_p} \nabla h \right) + S_h \qquad \text{(Ec. 1.2)}$$

Ecuación de conservación de cantidad de movimiento.

$$\frac{\partial (\rho u)}{\partial t} + \nabla \cdot (\rho \mathbf{V} u) = \nabla \cdot (\mu \nabla u) - \nabla P \cdot \mathrm{i} + S_u \qquad \text{(Ec. 1.3)}$$

La Ec. 1.3 corresponde a la ecuación de momento para un fluido Newtoniano en la dirección x, de igual forma es posible escribir la ecuación en las otras direcciones. El término S_u contiene los términos del tensor de esfuerzos que no aparecen en el término difusivo, y las fuerzas volumétricas externas sobre el fluido.

Las ecuaciones gobernantes pueden representarse mediante una ecuación general compuesta por cuatro términos, en el orden de la ecuación: término transitorio, término convectivo, término difusivo y término fuente; esta ecuación es conocida como la ecuación general escalar de transporte.

$$\frac{\partial(\rho\phi)}{\partial t} + \nabla \cdot (\rho \mathrm{V}\phi) = \nabla \cdot (\Gamma \nabla\phi) + \mathrm{S} \qquad \text{(Ec. 1.4)}$$

Donde

Ec. de continuidad	$\phi = 1$	$\Gamma = 0$	$S = 0$
Ec. de energía	$\phi = h$	$\Gamma = k/C_p$	$S = S_h$
Ec. de momento (en x)	$\phi = u$	$\Gamma = \mu$	$S = -\nabla P \cdot \mathrm{i} + S_u$

1.2.1 Simplificaciones al modelo

El estudio bidimensional del flujo en un canal isotermo con expansión está definido por las ecuaciones de momento y la ecuación de continuidad; para llevar a cabo el análisis es necesario realizar simplificaciones de estas ecuaciones.

- Flujo incompresible: La densidad permanece constante en todo el dominio y durante todo el recorrido de tiempo. $\frac{\partial \rho}{\partial t} = 0$
- Propiedades constantes: La propiedad gamma permanece constante en todo el dominio.

- Flujo laminar y bidimensional: El análisis solo se realizará para números de Reynolds $Re_D < 400$ [1].
- Convección forzada: Al ser un flujo con convección forzada no se consideran las fuerzas volumétricas externas como la gravedad. $S_u = 0$

El estudio del flujo en el canal, es un problema de estado estable, sin embargo, en este trabajo se evalúa como problema transitorio donde el recorrido por el tiempo se detiene al llegar al estado estacionario.

Las ecuaciones de flujo a partir de las simplificaciones quedan así:

Ecuación de continuidad

$$\nabla \cdot (\rho \mathrm{V}) = 0 \Rightarrow \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \qquad \text{(Ec. 1.5)}$$

Ecuaciones de momento

$$\frac{\partial u}{\partial t} + \nabla \cdot (\mathrm{V}u) = \nabla \cdot (\upsilon \nabla u) - \frac{1}{\rho}\nabla p \cdot \mathrm{i} \qquad \text{(Ec. 1.6a)}$$

$$\frac{\partial v}{\partial t} + \nabla \cdot (\mathrm{V}v) = \nabla \cdot (\upsilon \nabla v) - \frac{1}{\rho}\nabla p \cdot \mathrm{j} \qquad \text{(Ec. 1.6b)}$$

1.2.2 Condiciones de frontera

Las condiciones de frontera son parámetros fundamentales en la modelación de un flujo, se encargan de diferenciar un problema de otro. Las condiciones de frontera comúnmente se clasifican en tres tipos según la información especificada.

- Condición de frontera Dirchlet: Se conoce el valor de la magnitud o propiedad ϕ en la frontera, denotado como ϕ_b.

$$\phi_b = \phi_{conocido}. \qquad \text{(Ec. 1.7)}$$

[1] En la literatura se ha demostrado que para números de Reynolds dentro de este rango, el flujo permanece con esas condiciones [9][13][15].

- Condición de frontera Neumann: Se conoce el valor del gradiente de la magnitud o propiedad en dirección normal a la cara frontera ($\hat{n}$).

$$(\nabla\phi)_b \cdot \hat{n} = q_{conocido} \qquad \text{(Ec. 1.8)}$$

- Condición de frontera mixta: Es una combinación de los dos casos anteriores.[2]

El problema en estudio requiere de condiciones de frontera para la velocidad y para la presión. Para la velocidad se tienen tres fronteras: Flujo a la entrada, flujo a la salida y paredes.

En las paredes se asume una condición de no deslizamiento, es decir, que el fluido que está en contacto con la superficie tiene igual velocidad a ella (Frontera Dirichlet); en este caso:

$$u = v = 0$$

En la entrada del canal la velocidad también es conocida (Frontera Dirichlet), el valor se asigna según el número de Reynolds a analizar. Es decir:

$$u = U_b = \frac{R_e \mu}{\rho D}$$

La longitud entre la entrada del canal y el escalón se denota L_u, y debe ser $L_u > 5h$ para que el flujo sea totalmente desarrollado y no afecte el flujo en el escalón[3].

A la salida del canal el flujo también debe ser totalmente desarrollado; para asegurar esto, la longitud entre el escalón y la salida el canal, denotada como L_d, debe ser $L_d > 4x_r$.[4] Una vez garantizado el flujo desarrollado, es posible imponer una condición de frontera de tipo Neumann a la salida del canal:

$$\frac{\partial u}{\partial x} = 0 \qquad \frac{\partial v}{\partial x} = 0$$

[2] No es utilizada en este trabajo.

[3] Valor presentado por G. Biswas et.al. [16] tras realizar simulaciones para diferentes longitudes L_u

[4] En el estudio numérico presentado por Armaly et.al. [14] se establece que este valor es suficiente para que la longitud de re-encuentro sea independiente de la longitud L_d

En el caso de la presión se establece una condición de frontera de tipo Neumann para todos los contornos del dominio exceptuando la salida del canal, allí se establece una condición de tipo de Dirichlet:

Entrada y Paredes: $\frac{\partial P}{\partial x} = 0, \quad \frac{\partial P}{\partial y} = 0$

Salida: $P = 0$

2. MÉTODO DE LOS VOLÚMENES FINITOS

La dinámica de fluidos computacional, CFD (Computational Fluid Dynamics) [2], es una rama de la mecánica de fluidos que consiste en el empleo de técnicas numéricas y herramientas computacionales para resolver problemas físicos relacionados con el movimiento de los fluidos y los fenómenos asociados a este; su objetivo es la creación de un programa numérico que calcule una solución aproximada de las ecuaciones de gobierno de la mecánica de fluidos aplicadas a un problema dado.
Estas técnicas numéricas han evolucionado en gran escala y se han introducido en las distintas áreas de la ingeniería, dando soluciones a necesidades reales de la industria.

Existen varios métodos numéricos para la solución de las ecuaciones gobernantes, entre los más conocidos se encuentran: el método de diferencias finitas, el método de elementos finitos y el método de volúmenes finitos.

El método de diferencias finitas representa las derivadas parciales de las ecuaciones diferenciales en una aproximación de las variables por diferencias entre los puntos vecinos, utilizando series de Taylor. El método de elementos finitos aplica a una representación funcional el concepto de las diferencias finitas, los parámetros de la representación son puntos que dividen el dominio en una serie de elementos. Estos métodos no tienen en cuenta de manera explícita el principio de conservación al derivar las ecuaciones discretas.

A diferencia de los anteriores, el método de volúmenes finitos[5] se basa en la forma integral de las ecuaciones de conservación; en primer lugar se divide el dominio en un número finito de celdas o volúmenes de control que constituyen el mallado del dominio a estudiar, y a continuación se aplican los balances de los flujos de las variables a cada uno de estos elementos, de modo que es posible observar que

[5] También llamado método de volumen de control [1]

cuando una cantidad específica de una variable conservada es transportada fuera del volumen de control, la misma cantidad es transportada al volumen adyacente. Al final, se obtiene un sistema de ecuaciones algebraicas que podrán ser resueltas por métodos directos o iterativos.

El uso de la forma integral de las ecuaciones le permite mayor flexibilidad en cualquier tipo de malla logrando una mayor aproximación a la geometría dada por el problema. Por esta razón este método constituye la técnica más comúnmente empleada en la discretización de las ecuaciones en CFD.

A continuación se da una explicación general de los pasos a seguir en el método de volúmenes finitos.

2.1 DEFINICIÓN DEL MODELO MATEMÁTICO

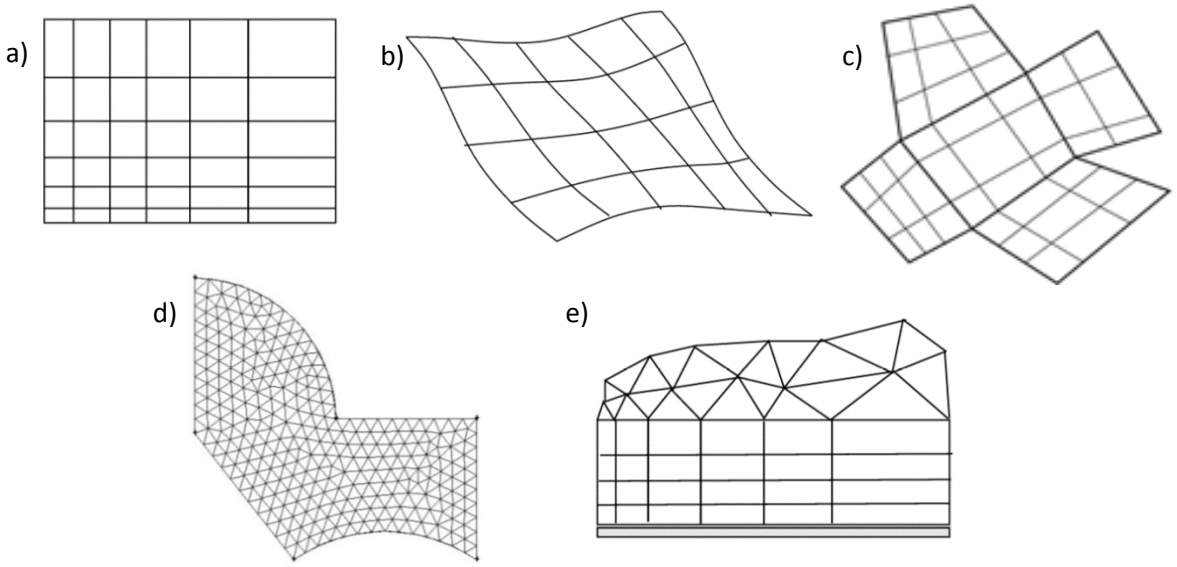
El primer paso es definir el modelo matemático y los principios de conservación que se van a aplicar. Las ecuaciones obtenidas de los principios de conservación se deben simplificar en algunos casos, por ejemplo, usando variables de semejanza para transformar el sistema de coordenadas, aproximaciones del flujo, y simplificación de las ecuaciones.

EL objetivo es que el modelo matemático desarrollado sea capaz de reproducir el fenómeno físico estudiado.

2.2 MALLADO ESPACIAL

El siguiente paso es el mallado que consiste en la división del dominio de interés en elementos o celdas; existen diferentes tipos de mallas que permiten adaptarse a una gran variedad de geometrías, algunas de las más comunes se encuentran en la Figura 2.

Figura 2. Algunos tipos de mallas. a) Malla estructurada ortogonal. b) Malla estructurada ajustada al cuerpo. c) Malla estructurada en bloque. d) Malla no estructurada. e) Malla híbrida.



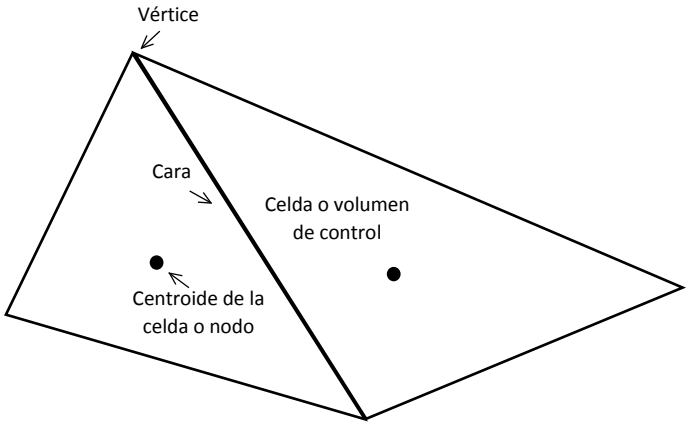
Fuente: MURTHY, J. Y. Numerical Methods in Heat, Mass, and Momentum Transfer.

Este trabajo se centra específicamente en el mallado no estructurado; en este tipo de malla se usan volúmenes de control con la forma de polígonos u otras figuras simples (usualmente triángulos y cuadriláteros), cada uno se encuentra conectado a un número arbitrario de vértices. Debido a su flexibilidad de forma se logra fácilmente la adaptación de la malla a cualquier geometría, por esta razón este tipo de malla es la más utilizada en problemas con geometrías complejas. En el caso del problema de estudio en este trabajo, esta malla permite ajustarse a la geometría inclinada del dominio.

Existen dos tipos de esquemas para definir el lugar en que se va a guardar el valor de las variables en la malla [3]: el esquema basado en los vértices (Vertex-Based Scheme), en el cual las variables son ubicadas en los vértices de las celdas; y el esquema basado en las celdas (Cell-Based Scheme), en el cual los nodos en donde se ubican las variables están ubicados en el centroide de la celda (Fig. 3).

En este trabajo se utiliza el esquema centrado en las celdas, debido a que es el más utilizado para el método de volúmenes finitos.

Figura 3. Terminología de malla.



2.3 DISCRETIZACIÓN DE LAS ECUACIONES

La clave del MVF es la integración de las ecuaciones de conservación sobre el volumen de control para producir una ecuación discretizada en el nodo de la forma:

$$a_P \phi_P = \sum_{nb} a_{nb} \phi_{nb} + b \qquad \text{(Ec. 2.1)}$$

Donde:

P es el volumen de control actual.

a_P es el coeficiente del volumen de control actual.

ϕ_P es el valor de la variable dependiente ϕ en el centroide de la celda P.[6]

$nb = 1,2,\dots,M$, con M igual al número de vecinos de la celda P.[7]

a_{nb} es el coeficiente del volumen de control vecino.

b es la suma de los términos independientes de la ecuación.

[6] Dado que se tomó en cuenta que el valor representativo de la variable en la celda corresponde al valor de la variable en el centroide de la misma.

[7] En este trabajo se utilizan mallas con celdas triangulares únicamente, luego M=3.

La ecuación se repite para cada uno de los nodos, con el objetivo de llegar a un sistema de ecuaciones algebraicas de la forma.

$$\boldsymbol{A}\phi = B \qquad \text{(Ec. 2.2)}$$

Donde A es una matriz de coeficientes, ϕ es el vector solución que se desea obtener, el cual contiene los valores calculados de la variable y B es un vector de términos independientes de las ecuaciones.

2.4 SOLUCIÓN DE LAS ECUACIONES

Las ecuaciones resultantes de la discretización puede ser lineales (i.e. los coeficientes son independientes de ϕ) o no lineales (i.e. los coeficientes son función de ϕ). Esto es un aspecto importante a la hora de escoger el camino de solución que se va a utilizar, ya que para las ecuaciones no lineales no se puede garantizar una única solución, esta puede depender de factores como los valores iniciales asumidos y el método de solución.

Las técnicas de solución son independientes del método de discretización y pueden ser clasificados en métodos directos y métodos iterativos.

En la actualidad existen programas y librerías que contienen los algoritmos para varios métodos de solución, y constantemente se modifican y actualizan conforme avanzan las investigaciones en estos temas.

En este proyecto se ha seleccionado el PETSc (Portable, Extensible Toolkit for Scientific Computation) [22]. Este es un conjunto de estructuras de datos y rutinas que incluye una serie de métodos de solución para ecuaciones lineales y no lineales que se pueden aplicar directamente al código. Debido a que se va a hacer uso de esta herramienta, no se hará énfasis en el proceso de cálculo que conlleva cada método de solución[8].

[8] En el Anexo A se presenta una breve descripción de algunos de los métodos que se utilizaron en este trabajo.

2.4.1 Ecuaciones Lineales

Los sistemas de ecuaciones lineales pueden ser resueltos por métodos directos o métodos iterativos. Los métodos directos resuelven las ecuaciones mediante procedimientos del algebra lineal, estos pueden arrojar resultados más exactos en algunos casos, sin embargo, no son eficientes respecto al tiempo requerido para la solución y la demanda de memoria computacional, especialmente en problemas complejos que requieren una gran cantidad de cálculos; por esta razón no son muy utilizados en aplicaciones de CFD y no se tendrán en cuenta como opción de método de solución para el problema en estudio.

En cuanto a los métodos iterativos, la idea básica es asumir un vector solución inicial, y a partir de este se calcula un vector solución corregido basado en alguna estrategia para reducir la diferencia entre el vector de solución inicial y el actual; luego, el procedimiento se repite hasta alcanzar la convergencia.

Al utilizar métodos iterativos se debe tener en cuenta que el camino de solución influye en el resultado encontrado, estos no convergen para todos los sistemas de ecuaciones. El número requerido de iteraciones para alcanzar la convergencia depende de:

- El predominio diagonal de los coeficientes en la matriz; si aumenta el predominio diagonal, el número de iteraciones disminuye.
- El método de iteración usado. Es posible que la solución converja para ciertos métodos y para otros no.
- El vector de solución inicial.
- El criterio de convergencia especificado.

Para los métodos de solución de ecuaciones lineales, PETSc permite la combinación de un método de subespacio Krylov y un precondicionador, esta opción es uno de los avances más modernos en la solución iterativa de ecuaciones lineales. Una explicación más detallada de este tema es presentado en el anexo A.

2.4.2 Ecuaciones no lineales

La solución de problemas no lineales de gran escala acopla muchas facetas de la ciencia computacional y demanda estrategias de solución robustas y flexibles. PETSc proporciona una gran librería con métodos de solución para este tipo de ecuaciones.

Una forma alternativa de resolver este tipo de problemas es reordenar la ecuación para poder tratarla como una ecuación lineal que se resuelve por métodos iterativos. Más adelante se verá que el sistema de ecuaciones a resolver para los problemas que se estudian en este trabajo resultan ser no lineales, esta alternativa resulta más conveniente, y será la utilizada para llevar a cabo la solución.

2.5 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 que representen los resultados obtenidos del problema. En el presente trabajo se utilizará la plataforma ParaView [23] como herramienta para realizar los gráficos y el análisis de los datos obtenidos en las simulaciones.

2.6 CARACTERÍSTICAS DE LOS MÉTODOS NUMÉRICOS

Al utilizar métodos numéricos es necesario evaluar ciertas características para verificar si el algoritmo desarrollado generará una solución satisfactoria. Las características a evaluar son las siguientes [3]:

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.

La diferencia entre la ecuación discretizada y la ecuación diferencial parcial es el error de truncamiento. El error de truncamiento debe tender a desaparecer cuando la malla se refine.

2. Precisión: Esta característica determina que tan correcta es la solución encontrada con respecto a la solución exacta; la precisión se ve afectada por los errores de discretización, el error debido al cálculo iterativo, y el error de truncamiento.
3. Estabilidad: Un esquema numérico usado para la solución de una ecuación discretizada es estable si el error permanece acotado, es decir si al pasar de una iteración a otra o de un tiempo a otro en una solución de avance, este permanece constante o decrece.

   Esta es una propiedad del método de solución. Existen métodos de análisis de estabilidad, sin embargo, su aplicación suele ser de gran dificultad en problemas no lineales; razón por la cual no se estudiaran en este trabajo.
4. Convergencia: Para que un método numérico sea definido como convergente, debe ser consistente y estable.

   Puede hablarse de convergencia en dos casos: cuando un método iterativo obtiene una solución al sistema de ecuaciones; o cuando la solución del sistema llega al punto donde permanece invariante con respecto al refinamiento de la malla.

3. MALLADO ESPACIAL DEL DOMINIO

La división física del volumen en celdas se realizó utilizando la plataforma Salome [21] que permite realizar un mallado automático de la geometría.

Salome es un software de uso libre que provee una plataforma genérica para el pre- y pos-procesamiento de simulaciones numéricas (Fig. 4), a continuación se dará una breve explicación del proceso que se lleva a cabo para crear la malla no estructurada de dos dimensiones que se requiere para los problemas que se estudian en el presente trabajo.

Figura 4. Plataforma Salome.

3.1 CONSTRUCCIÓN DE LA GEOMETRÍA

Para empezar se crea un nuevo estudio y se ingresa al módulo *Geometry*.
Se crea el croquis de la figura **Sketch_1** (Fig. 5): (menú *New Entity/Basic/2D Sketch*)
Se crea el volumen del dominio 2D **Face_1**: (menú *New Entity/Build/Face*)

En el problema del canal con expansión resulta conveniente densificar la malla en la sección en que se produce la zona de recirculación del flujo; en este caso es necesario dividir la geometría en tres secciones y crear un volumen compuesto.
Se crean las 3 secciones **Face_1**, **Face_2**, **Face_3**.
Se seleccionan las 3 caras y se crea el volumen del dominio 2D compuesto **Compound_1** (Fig. 6): (menú *New Entity/Build/Compound*)

Figura 5. Construcción de un croquis 2D.

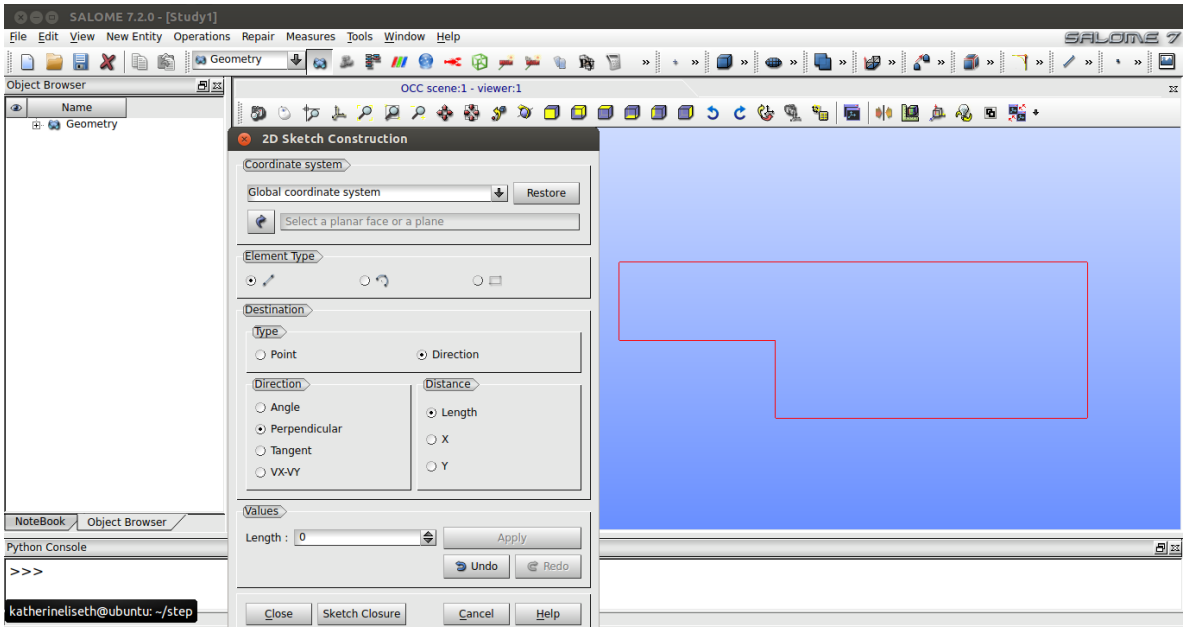
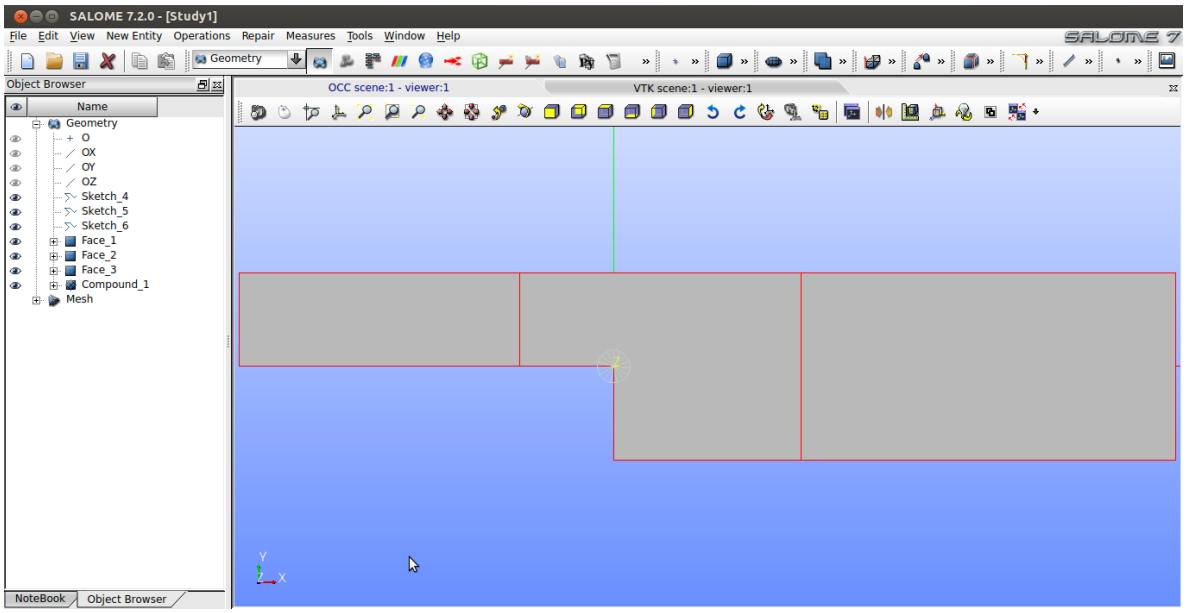


Figura 6. Geometría compuesta en 2D.



3.2 CREACIÓN DE LA MALLA

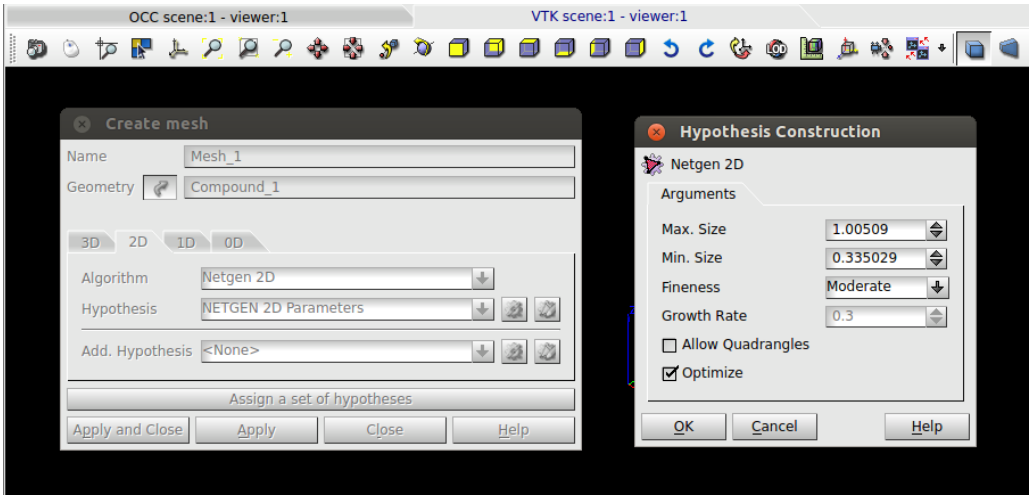
A continuación se ingresa al módulo *Mesh.*

Se selecciona el volumen compuesto y se crea la malla (menú *Mesh/Create Mesh*)

El comando Create Mesh permite seleccionar una serie de algoritmos para la creación automática de la malla, se deben seleccionar un algoritmo y su hipótesis para los elementos 2D y otro para los elementos 1D.

Para los elementos 2D (Fig 7) se escoge el algoritmo *NETGEN 2D*, y la hipótesis *NETGEN 2D Parameters*; esta permite seleccionar la longitud mínima y máxima de los segmentos que componen los triángulos de la malla; también es posible escoger entre una escala de refinamiento de muy fina a basta.

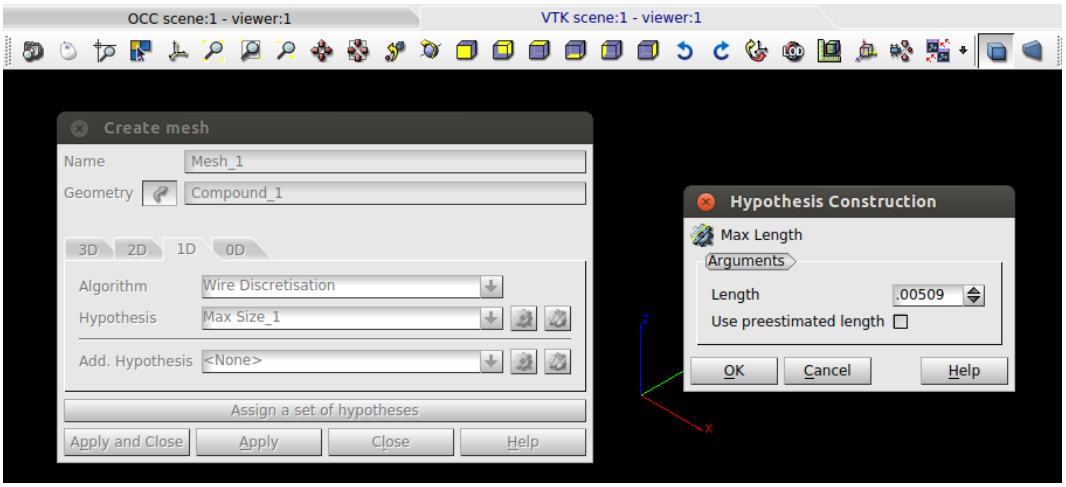
Figura 7. Parámetros 2D para crear la malla.



Para los elementos 1D (Fig. 8) se escoge el algoritmo *Wire Discretization* y la hipótesis *Max Size*; esta permite seleccionar la longitud máxima de los segmentos en los que se dividen los ejes geométricos.[9]

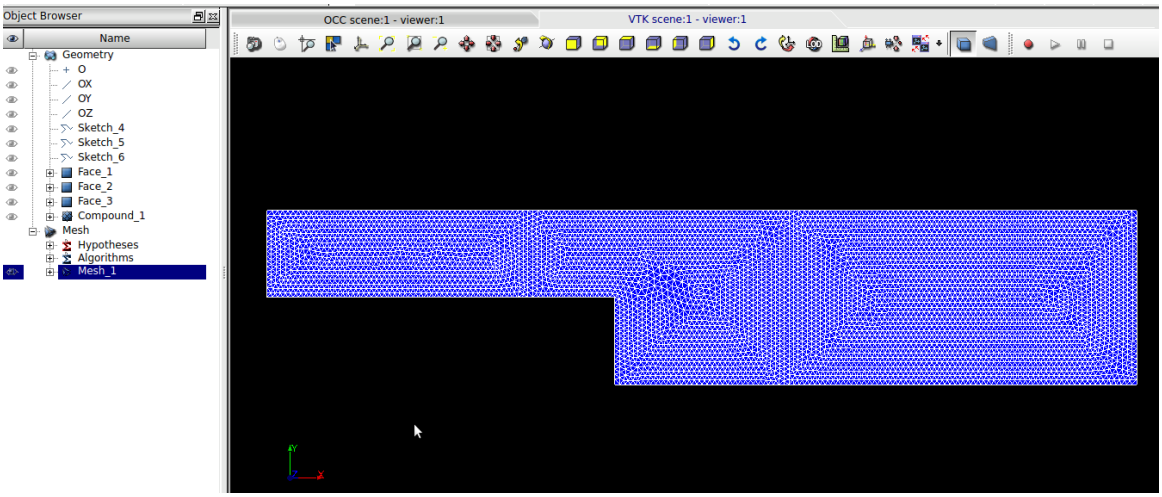
[9] La forma explicada de creación de malla es la opción utilizada para el mallado del dominio en el presente trabajo; sin embargo, es solo una de las muchas posibilidades que tiene la plataforma Salome para este propósito.

Figura 8. Parámetros 1D para crear la malla.



Una vez se hayan especificado los parámetros se selecciona *apply* y se computa la malla creada; el resultado es una malla de densidad relativamente uniforme para todo el dominio (Fig. 9).

Figura 9. Malla uniforme en geometría compuesta.



En el problema de estudio se quiere que únicamente la zona de recirculación tenga esa densidad, y las otras secciones pueden establecerse con una densidad menor, esto se logra así:

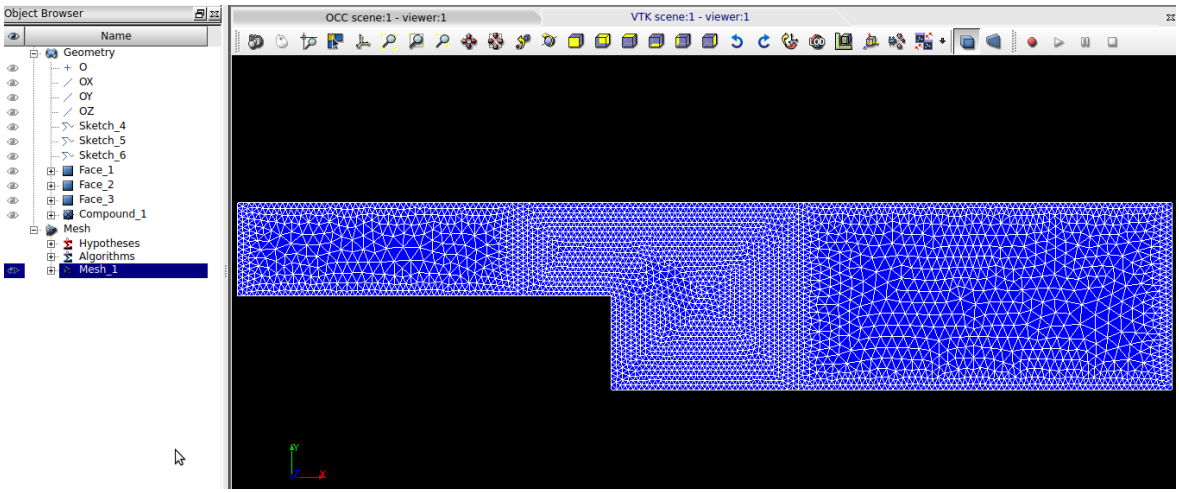
Se selecciona la malla creada y se crea una submalla de la misma (menú *Mesh/Create Submesh*):

Se debe seleccionar un elemento de la sección donde se quiere crear la submalla y se asignan los algoritmos e hipótesis de la misma forma que se hizo con la malla inicial, a excepción que en este caso la longitud máxima seleccionada en la hipótesis 2D será mayor que la anterior; la longitud mínima en la hipótesis 2D y la longitud de la hipótesis 1D debe ser la misma que la malla inicial para que los elementos coincidan en las fronteras de las secciones.

De igual forma se crea una nueva submalla para la otra sección.

Finalmente se obtiene una nueva malla con una mayor densidad en la zona que se requiere (Fig. 10)

Figura 10. Malla refinada en geometría compuesta.

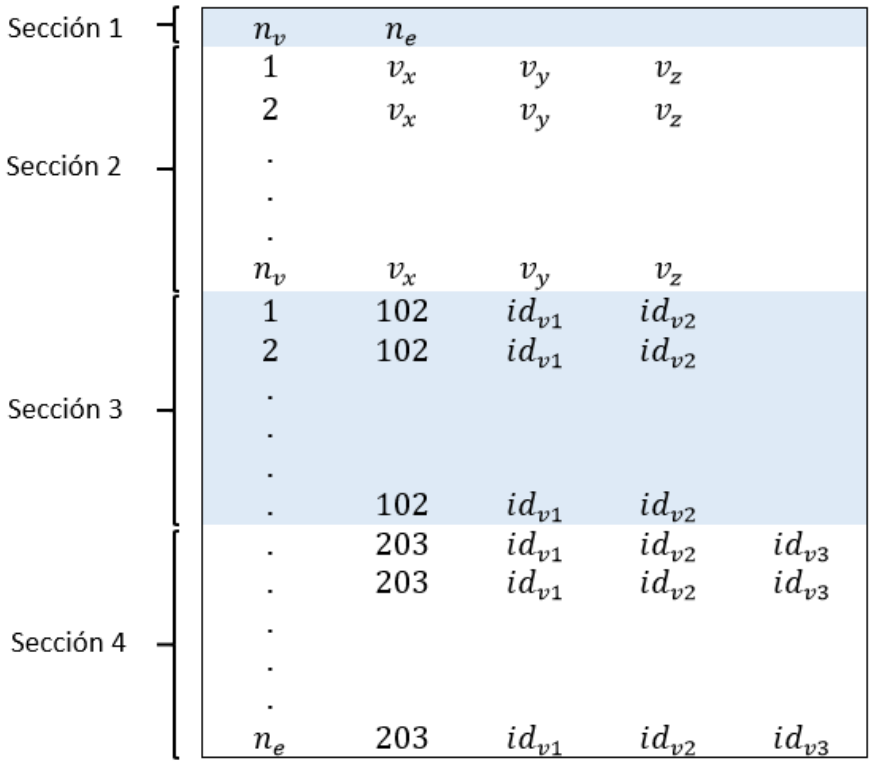


3.3 EXPORTAR LA INFORMACIÓN DE LA MALLA

Una vez obtenida la malla es necesario exportarla a un formato que pueda leer el código computacional desarrollado para la simulación. En este caso el archivo que

se exporta del programa es de carácter .dat y tiene la estructura presentada en la figura 11.

Figura 11. Estructura del archivo que se exporta de Salome.



La primera línea da la información del número de vértices y el número de elementos (1D y 2D) existentes en la malla, respectivamente.

La segunda sección indica las coordenadas de cada uno de los vértices, el número en la primera columna corresponde al id (número de identificación) de determinado vértice.

La tercera y cuarta sección establece los vértices que corresponden a cada uno de los elementos; la tercera sección corresponde a los elementos 1D (identificados por el número 102 en la segunda columna), y la cuarta sección a los elementos 2D (identificados por el número 203 en la segunda columna)

4. ESTRUCTURA DEL CÓDIGO DESARROLLADO

Antes de explicar el fundamento matemático y la discretización de las ecuaciones se explicará brevemente la estructura del código creado.

El código computacional[10] se realizó en lenguaje C++ usando las librerías de PETSc[11]. Se divide en una serie de clases que contienen funciones y variables que se enlazan para llevar a cabo la simulación de los problemas en estudio. Es importante aclarar que, por simplicidad, el desarrollo del código se hizo únicamente para celdas triangulares (2D), luego no es posible ingresar mallas con celdas de geometrías diferentes.

4.1 ESTRUCTURA DE LAS CLASES

Para crear una clase en C++, se deben crear dos archivos: un archivo de encabezado con el nombre de la clase de extensión .h y un archivo de código fuente con el nombre de la clase de extensión .cpp.

En el archivo de encabezado se declaran las variables y funciones de la clase, y tiene la siguiente estructura:

```
//Archivo de encabezado difusivo.h
class difusivo
{
     private:
          PetscErrorCode ierr;
          int i,j;
          PetscScalar value[3];
          PetscScalar *condf;
```

[10] El código computacional completo con todos los archivos de las clases se presenta en el Anexo C.

[11] En el Anexo B se presenta una explicación detallada de las rutinas de PETSc que se utilizaron en el código desarrollado.

```
        const PetscScalar *area;
    public:
        PetscErrorCode calcular_D(double gamma,geom b,Mat
        *matriz_difusivo,Vec *vec_diffront);
};
```

Las variables son declaradas de tipo `private` que significa que solo serán modificadas por el objeto de esta clase; las variables listadas en el ejemplo corresponden, en el orden mostrado a: variable de error de Petsc[12], variables de tipo entero utilizadas como contadores, variable usada para realizar los cálculos en la función y guardar los valores en vectores o matrices, variable para guardar valores extraídos de un vector creado anteriormente, variable para guardar valores extraídos de una fila de una matriz creada anteriormente[13].

Las funciones de la clase se declaran de tipo `public` y todas devuelven el valor de la variable de error de Petsc.

En el archivo de código fuente se escribe la implementación de las funciones de la clase, tiene la siguiente estructura:

```
//Archivo de código fuente difusivo.cpp
PetscErrorCode difusivo::calcular_D(double gamma,geom b,Mat
*matriz_difusivo,Vec *vec_diffront)
{
    //Implementación de la función
    ...
    return ierr;
}
```

[12] El uso de esta variable se explica en el Anexo B.
[13] Básicamente estos son los tipos de variables utilizadas en todas las clases desarrolladas.

4.2 CLASES DESARROLLADAS

Las clases creadas en este trabajo pueden clasificarse en cuatro tipos:

Tipo I: son clases que serán utilizadas únicamente al comienzo del programa en la función main, los objetos creados de estas clases serán utilizados por casi toda las funciones del programa, y los elementos que hacen parte de ellos no serán modificados en ningún momento. Las clases de este tipo son: `geom` y `frontera`.

Tipo II: son aquellas que se encargan de calcular los términos de la ecuación a resolver: términos difusivos, convectivos e independientes; estas clases no son llamadas desde la función main, son llamadas directamente por la clase que lleva a cabo la solución. Las clases de este tipo son: `difusivo`, `gradiente`, `convectivo` y `termindep`.

Tipo III: en estas clases se encuentra el algoritmo de solución del problema, cada una de estas clases corresponde a un problema distinto, por esta razón, solo una de ellas será utilizada en la simulación dependiendo del problema que se requiera solucionar. Las clases de este tipo son: `solestable`, `solexplicito`, `solimplicito` y `fractional`.

Tipo IV: esta clase es llamada únicamente al final de la clase solución y se encarga de exportar la solución calculada al formato de archivo necesario para el post.-procesamiento. La clase de este tipo es: `resultados`.

4.3 DESCRIPCIÓN GENERAL

A continuación se da una breve explicación de las clases `geom`, `frontera` y `resultados`, las demás clases serán explicadas en los siguientes capítulos en las secciones de implementación del código. También se presenta el esquema general de la función main.

4.2.1 Clase geom

La clase `geom` se encarga de obtener todos los elementos geométricos necesarios para el programa, para ello cuenta con dos funciones:

- `leerdatos(char *malla)`: Se encarga de leer y guardar la información de la malla presente en el archivo exportado de Salome; requiere de un argumento que corresponde al nombre del archivo, el cual se designa en la función main.
- `calcular()`: Utiliza los valores obtenidos del archivo de malla para realizar todos los cálculos correspondientes y obtener los elementos geométricos que se necesitan en el programa: coordenadas de nodos, identificación de celdas vecinas, volumen de las celdas, vectores de áreas, vectores de dirección.

4.2.2 Clase frontera

La clase `frontera` se encarga de establecer las condiciones y los valores de frontera que son proporcionados en la función main, en cada uno de los volúmenes de control que contienen una cara frontera.

Hay dos opciones para las condiciones de frontera: un valor de 1 corresponde a una condición de frontera de tipo Dirichlet; un valor de -1 corresponde a una condición de frontera tipo Neumann.

El programa admite tres tipos de geometrías diferentes para la asignación de la frontera, para cada una existe una función en la clase.

- `leerR(geom gm, double *cf, double *vf)`: Esta función está diseñada para un dominio rectangular con cuatro fronteras perceptibles; requiere de tres argumentos: un objeto de la clase `geom`, y dos arrays de tipo doble que corresponden a la condición de frontera y el valor de ϕ para cada una de las cuatro fronteras existentes

- `leerBFS(geom gm, double *cf, double *vf)`: Esta función está diseñada para la geometría del problema del canal con expansión (recto e inclinado); requiere de los mismos argumentos que la función anterior,
- `leerSH(geom gm)`: Esta función está diseñada específicamente para el problema conocido Smith-Hutton[14], cuyos valores en las fronteras son siempre los mismos, razón por la cual son establecidos directamente desde esta función y no en la función main; requiere de un solo argumento que corresponde a un objeto de la clase `geom`.

Dependiendo del problema de estudio, se escoge una de las tres funciones mencionadas.

4.2.3 Clase resultados

La clase `resultados` se encarga de exportar la solución obtenida a un archivo de carácter .vtk, de modo que pueda ser leído por la plataforma ParaView para realizar el post-procesamiento de la solución. Esto lo realiza mediante la siguiente función:

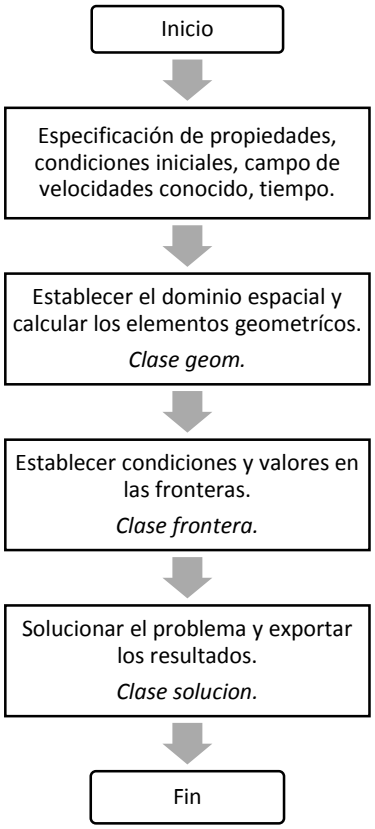
- `imprimir(geom gm,Vec vec_solucionu,Vec vec_solucionv,Vec vec_solucionp)`: Requiere de cuatro argumentos que corresponden a un objeto de la clase `geom`, dos vectores que contienen las componentes del campo de velocidad de la solución, y un vector que contiene el campo escalar de la solución.

4.2.4 Función main

La función main es el cuerpo principal del programa, su distribución se muestra en la figura 12.

[14] La descripción de este problema se encuentra más adelante en el caítulo 6.

Figura 12. Diagrama de flujo de la función main.



En el último paso, la clase solución hace referencia a una de las clases de tipo III mencionadas anteriormente, la estructura de estas clases se explicará en los siguientes capítulos conforme se haga referencia a cada problema.

5. DIFUSIÓN-CONVECCIÓN EN ESTADO ESTABLE

En este capítulo se muestra como son discretizados los términos de la ecuación de transporte para luego ser utilizados en el problema de flujo. Se obtiene el sistema de ecuaciones discretas para el problema de difusión-convección en estado estable y se describen los pasos para encontrar la solución; además, se hace una breve explicación de su implementación en el código computacional.

Inicialmente se considerará el fenómeno de difusión-convección en estado estable en un dominio bidimensional; a la ecuación general de transporte (Ec. 1.4) se aplican las simplificaciones descritas en la sección 1.2.1 y se descarta el término transitorio y el término fuente[15]; luego la ecuación resultante a discretizar consiste únicamente del término difusivo y el término convectivo.

$$\nabla \cdot (\rho \mathbf{V}\phi) = \nabla \cdot (\Gamma \nabla \phi) \qquad \text{(Ec. 5.1)}$$

El campo de velocidades $\mathbf{V}=u\,\mathbf{i} + v\,\mathbf{j}$ es conocido en todo el dominio**.**

Es posible determinar un vector $\mathbf{J} = J_x\,\mathbf{i} + J_y\,\mathbf{j}$ que represente el flujo de difusión convección, dado por:

$$\mathbf{J} = \rho \mathbf{V}\phi - \Gamma \nabla \phi \qquad \text{(Ec. 5.2)}$$

De modo que la ecuación pueda escribirse como:

$$\nabla \cdot \mathbf{J} = 0 \ ^{16} \qquad \text{(Ec. 5.3)}$$

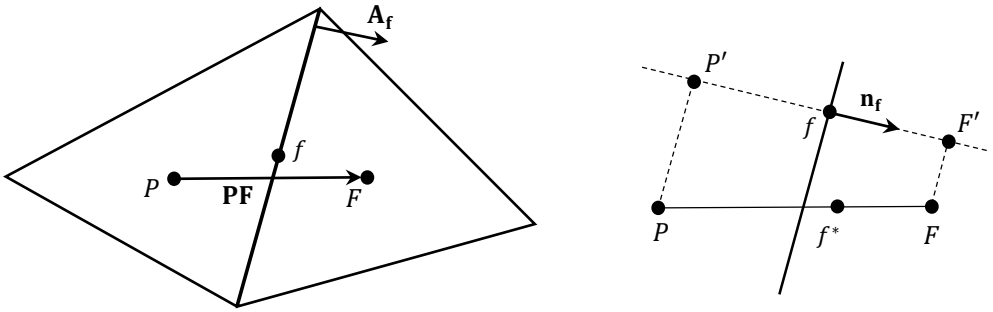
El proceso de discretización comienza integrando respecto al volumen P (Fig. 5.1).

$$\int_{d\forall_P} \nabla \cdot \mathbf{J} d\forall = 0 \qquad \text{(Ec. 5.4)}$$

[15] El término fuente se descarta debido a que su discretización no es requerida para el problema de evaluación de flujo que se estudia en este trabajo.

[16] Dado que ρ y Γ son constantes esta resulta ser una ecuación de Laplace.

Figura 13. Detalle de malla no estructurada y parámetros geométricos.



Aplicando el teorema de la divergencia, siendo $\mathbf{J}$ el campo vectorial con derivadas parciales de primer orden continuas, se obtiene:

$$\int_{\mathrm{A}} \mathbf{J} \cdot d\mathbf{A} = 0 \qquad \text{(Ec. 5.5)}$$

Esta representa la integral sobre la superficie de control $\mathbf{A}$ del volumen de control.

La primera suposición que se tiene en cuenta, es que $\mathbf{J}$ puede ser representado por el valor en el centroide de la cara del volumen de control $\mathbf{J}_f$**,** teniendo esto en cuenta, de la Ec. 5.5 se obtiene que:

$$\sum_f \mathbf{J}_f \cdot \mathbf{A}_f = 0 \qquad \text{(Ec. 5.6)}$$

La sumatoria se realiza sobre todas las caras que pertenecen al volumen de control. El vector de área está dado por: $\mathbf{A}_f = A_x\,\mathbf{i} + A_y\,\mathbf{j}$

El transporte de la variable ϕ en la cara f se puede escribir como:

$$\mathbf{J}_f \cdot \mathbf{A}_f = \rho \mathbf{V}_f \cdot \mathbf{A}_f \phi_f - \Gamma(\nabla\phi)_f \cdot \mathbf{A}_f \qquad \text{(Ec. 5.7)}$$

En el primer término, el término convectivo, es posible observar que el valor que acompaña la variable ϕ_f corresponde al flujo másico que pasa por la cara f, y se definirá como:

$$F_f = \rho \mathbf{V}_f \cdot \mathbf{A}_f \qquad \text{(Ec. 5.8)}$$

Además, este término requiere evaluar el valor de ϕ en las caras, para lo cual es necesario de un esquema matemático de aproximación, el esquema utilizado es el de interpolación aguas-arriba (Upwind), el cual establece el valor de ϕ_f como el valor de ϕ aguas arriba, es decir:

$$\phi_f = \phi_P \;\; si\; F_f \geq 0$$
$$\phi_f = \phi_F \;\; si\; F_f < 0 \qquad \text{(Ec. 5.9)}$$

Se utiliza este esquema debido a que resulta fácil de implementar y no presenta restricciones de estabilidad como el esquema de diferencias centradas [3].

Para la evaluación del segundo término, el término difusivo, se utilizó el criterio del gradiente directo [6], en este, se define el producto punto entre el gradiente de ϕ y el área como se muestra a continuación.

$$\Gamma(\nabla\phi)_f \cdot \mathbf{A}_f = \Gamma\left(\frac{\partial\phi}{\partial n}\right)_f A_f = \Gamma\frac{\mathbf{A}_f \cdot \mathbf{A}_f}{\mathbf{PF} \cdot \mathbf{A}_f}(\phi_{F'} - \phi_{P'}) \qquad \text{(Ec. 5.10)}$$

Siendo $\mathbf{PF}$ el vector que une los centroides de las celdas P y F, y $\phi_{F'}$ y $\phi_{P'}$ son las proyecciones de los valores de ϕ en los centroides de los volúmenes de control sobre el vector normal a la cara $\mathbf{n}_f$ que pasa por el centroide de la cara f (Fig. 13); estos valores pueden ser aproximados de la siguiente forma:

$$\phi_{P'} = \phi_P + \nabla\phi_P \cdot \mathbf{PP'} \qquad \text{y} \qquad \phi_{F'} = \phi_F + \nabla\phi_F \cdot \mathbf{FF'} \qquad \text{(Ec. 5.11)}$$

De modo que el flujo difusivo en la cara se puede reescribir como:

$$\Gamma(\nabla\phi)_f \cdot \mathbf{A}_f = \Gamma\frac{\mathbf{A}_f \cdot \mathbf{A}_f}{\mathbf{PF} \cdot \mathbf{A}_f}(\phi_F - \phi_P) + \Gamma\frac{\mathbf{A}_f \cdot \mathbf{A}_f}{\mathbf{PF} \cdot \mathbf{A}_f}(\nabla\phi_F \cdot \mathbf{FF'} - \nabla\phi_P \cdot \mathbf{PP'}) \qquad \text{(Ec. 5.12)}$$

El segundo término de la ecuación se denotará como el gradiente secundario de la difusión $\mathcal{S}_f$ [17].

[17] Es posible ver que en mallas ortogonales este término se hace cero, y por tanto el flujo difusivo depende únicamente del primer término de la ec. 5.12.

$$\mathcal{S}_f = \Gamma \frac{\mathbf{A}_f \cdot \mathbf{A}_f}{\mathbf{PF} \cdot \mathbf{A}_f} \left(\nabla \phi_F \cdot \mathbf{FF}' - \nabla \phi_P \cdot \mathbf{PP}' \right) \quad \text{(Ec. 5.13)}$$

Así mismo se define:

$$D_f = \Gamma \frac{\mathbf{A}_f \cdot \mathbf{A}_f}{\mathbf{PF} \cdot \mathbf{A}_f} \quad \text{(Ec. 5.14)}$$

5.1 CÁLCULO DEL GRADIENTE

El cálculo del gradiente está basado en el teorema del gradiente [3][6], que establece que para un volumen cerrado $\Delta\forall_P$, rodeado por una superficie A:

$$\int_{\Delta\forall_P} \nabla\phi \, d\forall = \int_A \phi \, d\mathbf{A} \quad \text{(Ec. 5.15)}$$

Nuevamente, asumiendo que el gradiente en el volumen de control $\Delta\forall_P$ es constante, y que el valor de ϕ en las caras está definido por su valor en el centroide de la misma, se tiene que:

$$\nabla\phi_P = \frac{1}{\forall_P} \sum_f \phi_f \mathbf{A}_f \quad \text{(Ec. 5.16)}$$

Para poder utilizar la ecuación 5.16 es necesario conocer el valor de ϕ_f, existen varios criterios de interpolación dedicados a este fin. En este trabajo se utilizó el criterio denotado como I1 propuesto en [6]; este consiste en realizar una interpolación lineal de los valores en los nodos de las celdas en un punto de la línea del vector **PF**, que se denotará como f^*(Fig. 13); de modo que:

$$\phi_f = \phi_{f^*} = \left(1 - \alpha_f\right)\phi_P + \alpha_f \phi_F \quad \text{(Ec. 5.17)}$$

El factor de interpolación α_f es la relación entre las distancias de Pf^* y PF; de modo que el cálculo de ϕ_f depende del criterio utilizado para definir la posición de f^*; en

esta formulación, la posición de f^* está definida de tal forma que la distancia de f^* a f sea mínima. En ese caso:

$$\alpha_f = \frac{\mathbf{Pf} \cdot \mathbf{PF}}{\mathbf{PF} \cdot \mathbf{PF}} \qquad \text{(Ec. 5.18)}$$

5.2 ECUACIÓN DISCRETA

La ecuación discreta de difusión-convección que se obtiene a partir de la Ec. 5.1 es:

$$\sum_{nb}\left(MAX\left(F_f, 0\right) + D_f\right)\phi_P - \sum_{nb}\left(MAX\left(-F_f, 0\right) + D_f\right)\phi_{nb} - \sum_{nb}\left(\mathcal{S}_f\right)_{nb} = 0 \quad \text{(Ec. 5.19)}$$

Esta ecuación es de la forma (Ec. 2.1):

$$a_P\phi_P = \sum_{nb} a_{nb}\phi_{nb} + b$$

Donde,

$a_{nb} = MAX\left(-F_f, 0\right) + D_f$, con $nb = 1,2,3$

$$a_P = \sum_{nb} a_{nb} + \sum_{f} F_f$$

$$b = \sum_{nb}\left(\mathcal{S}_f\right)_{nb}$$

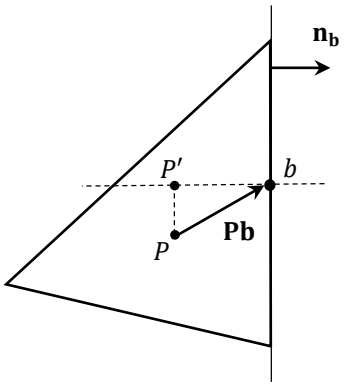
La ecuación discreta se aplica a cada uno de los volúmenes de control del dominio que no tengan frontera.

Un volumen de control frontera es aquel que tiene una cara perteneciente a la frontera del dominio (Fig. 14). Al igual que antes, los valores discretos de la variable ϕ son guardados en el centroide de la celda, y, adicionalmente se guarda un nuevo valor discreto de ϕ en el centroide de la cara frontera.

En la figura 14 es posible ver que una de las caras de la celda P pertenece a la frontera del dominio, esta cara se denotará con la letra b y tiene la característica

que permite guardar un valor conocido de la magnitud ϕ en el centroide de la misma denotado como ϕ_b.

Figura 14. Parámetros geométricos en un volumen de control frontera.



El flujo $\mathbf{J_b}$ en la cara b está dado por:

$$\mathbf{J}_b \cdot \mathbf{A}_b = \rho \mathbf{V}_b \cdot \mathbf{A}_b \phi_b - \Gamma (\nabla \phi)_b \cdot \mathbf{A}_b \qquad \text{(Ec. 5.20)}$$

En este caso, la ecuación 5.19 debe modificarse según el tipo de frontera que se tenga. En el capítulo 1 se mencionaron tres tipos de fronteras comunes, la discretización se realizará únicamente para dos de ellos:

- Frontera Dirichlet

Como se mencionó anteriormente, en este tipo de frontera se conoce el valor de ϕ_b.

Para evaluar los flujos difusivos en la frontera es necesario, primero, determinar el valor de $(\nabla \phi)_b$ con el fin de poder establecer el gradiente secundario en la frontera; el criterio que se usará es asumir que $(\nabla \phi)_b = (\nabla \phi)_P$, de modo que el gradiente secundario en la frontera es:

$$\mathcal{S}_b = \Gamma \frac{\mathbf{A}_b \cdot \mathbf{A}_b}{\mathbf{Pb} \cdot \mathbf{A}_b} (\nabla \phi_P \cdot \mathbf{FF'} - \nabla \phi_P \cdot \mathbf{PP'}) \qquad \text{(Ec. 5.21)}$$

En cuanto al término convectivo el valor de ϕ en la cara es $\phi_f = \phi_b$, luego no es necesario utilizar el esquema de aproximación.

Con estos valores se obtiene la ecuación discreta:

$$\sum_{nb}(MAX(F_f,0)+D_f)\,\phi_P + D_b\phi_P - \sum_{nb}\left(MAX(-F_f,0)+D_f\right)\phi_{nb} + F_b\phi_b - D_b\phi_b - \sum_{nb}\left(\mathcal{S}_f\right)_{nb} - \mathcal{S}_b = 0 \qquad \text{(Ec. 5.22)}$$

Y los términos para la ecuación 2.1 son:

$$a_{nb} = MAX(-F_f,0) + D_f$$

$$a_b = -F_b + D_b$$

$$a_P = \sum_{nb} a_{nb} + \sum_{f} F_f + a_b + F_b$$

$$b = \sum_{nb}\left(\mathcal{S}_f\right)_{nb} + a_b\phi_b + \mathcal{S}_b$$

- Frontera Neumann

En esta condición de frontera el valor conocido es $q_b = (\nabla\phi)_b \cdot \mathbf{n}_b$.

En cuanto al término convectivo, este tipo de frontera es usualmente usada cuando se tiene un flujo de salida o una pared, es decir: $\mathbf{V}_b \cdot \mathbf{A}_b \geq 0$; por esta razón es posible utilizar el esquema Upwind para establecer el valore de ϕ_b en el término convectivo:

$$\phi_b = \phi_P \qquad \text{(Ec. 5.23)}$$

Con estos valores se obtiene la ecuación discreta:

$$\sum_{nb}(MAX(F_f,0)+D_f)\,\phi_P - \sum_{nb}\left(MAX(-F_f,0)+D_f\right)\phi_{nb} - \sum_{nb}\left(\mathcal{S}_f\right)_{nb} - \Gamma q_b A_b + F_b\phi_P = 0 \qquad \text{(Ec. 5.24)}$$

Y los términos para la ecuación 2.1 son:

$$a_{nb} = MAX(-F_f,0) + D_f$$

$$a_P = \sum_{nb} a_{nb} + \sum_f F_f + F_b$$

$$b = \sum_{nb} \left(\mathcal{S}_f\right)_{nb} + \Gamma q_b A_b$$

Al aplicar la ecuación discreta en cada uno de los volúmenes de control se obtiene un sistema de ecuaciones lineales mencionado en la sección 2.3 (Ec. 2.2).

$$\boldsymbol{A}\phi = B$$

Es posible observar que el término del segundo gradiente, presente en el vector de términos independientes, impide evaluar el sistema directamente debido a que involucra los valores de ϕ en las celdas[18]; por esta razón, es necesario asumir un valor inicial del vector solución para calcular este término y realizar un proceso iterativo para obtener la solución del sistema de ecuaciones.

5.4 IMPLEMENTACIÓN EN EL CÓDIGO COMPUTACIONAL

El problema de difusión-convección en estado estable se implementa en el código computacional mediante la clase `solestable`, la cual contiene una única función encargada de llevar a cabo la solución del problema.

Como se mencionó anteriormente, dentro de esta función se llaman las clases que se encargan de calcular los términos de la ecuación; a continuación se explicarán estas clases, y la estructura de la solución dentro de la clase `solestable`.

5.4.1 Clase difusivo

La clase `difusivo` se encarga de obtener los términos difusivos de la matriz de coeficientes (D_f) en las caras de cada volumen de control y en las fronteras, estos

[18] El término del segundo gradiente es el responsable de que la ecuación discreta corresponda a una ecuación no lineal, sin embargo, se utiliza la alternativa planteada en la sección 2.4.2, se reordena la ecuación, y se resuelve iterativamente utilizando un valor inicial asumido de la variable ϕ.

valores se guardan en una matriz de términos difusivos, y un vector de términos difusivos en las fronteras. Esto se realiza mediante la siguiente función:

- `calcular_D(double gamma, geom gm, Mat *matriz_difusivo, Vec *vec_diffront)`: Requiere de cuatro argumentos: la propiedad Γ, un objeto de la clase `geom`, un puntero a la matriz donde se va a guardar los términos difusivos de los volúmenes de control, y un puntero al vector donde se guardarán los términos difusivos de las fronteras.

5.4.2 Clase Gradiente

La clase `gradiente` se encarga de calcular el gradiente de determinada variable en cada uno de los volúmenes de control y guardar estos valores en una matriz de gradientes; para este fin primero es necesario realizar la interpolación de la variable en las caras.

Dentro de la clase se declaran dos funciones:

- `calcular_vcara(geom gm, frontera f, vec vec_phi, Mat *matriz_phif0)`: Esta función permite obtener el valor de phi en las caras para cada volumen de control. Requiere de cuatro argumentos: un objeto de la clase `geom`, un objeto de la clase `frontera` con la información de frontera de la variable ϕ, un vector con los valores centrados de la variable en cada uno de los volúmenes de control, un puntero a la matriz donde se guardaran los valores de ϕ en las caras.
- `calcular_grad(geom gm, frontera f, Vec vec_phi, Mat *matriz_gradphivc)`: Esta función permite obtener el valor del gradiente de ϕ en cada una de las celdas. Requiere de cuatro argumentos: un objeto de la clase `geom`, un objeto de la clase `frontera` con la información de frontera de la variable ϕ, un vector con los valores de la variable en cada uno de los volúmenes de control, un puntero a la matriz donde se guardaran los valores del gradiente de phi.

5.4.3 Clase Convectivo

La clase `convectivo` se encarga de obtener los términos convectivos de la matriz de coeficientes, es decir los flujos másicos (F_f) en las caras de cada volumen de control y en las fronteras, los valores se guardan en una matriz de términos convectivos, y un vector de términos convectivos en las fronteras. En este problema el campo de velocidades es conocido, luego la función a utilizar es:

- `calcular_F(double densidad, geom gm, double *velocidad, Mat *matriz_convectivo, Vec *vec_convecfront)`: Requiere de cinco argumentos: el valor de la densidad, un objeto de la clase `geom`, un array con los valores de u y v, un puntero a la matriz donde se va a guardar los términos convectivos de los volúmenes de control, y un puntero al vector donde se guardarán los términos convectivos de las fronteras.

5.4.4 Clase termindep

La clase `termindep` se encarga de obtener los valores del vector de términos independientes B de la ecuación discretizada; los términos independientes varían según el problema, la función declarada para el caso de difusión-convección en estado estable es la siguiente:

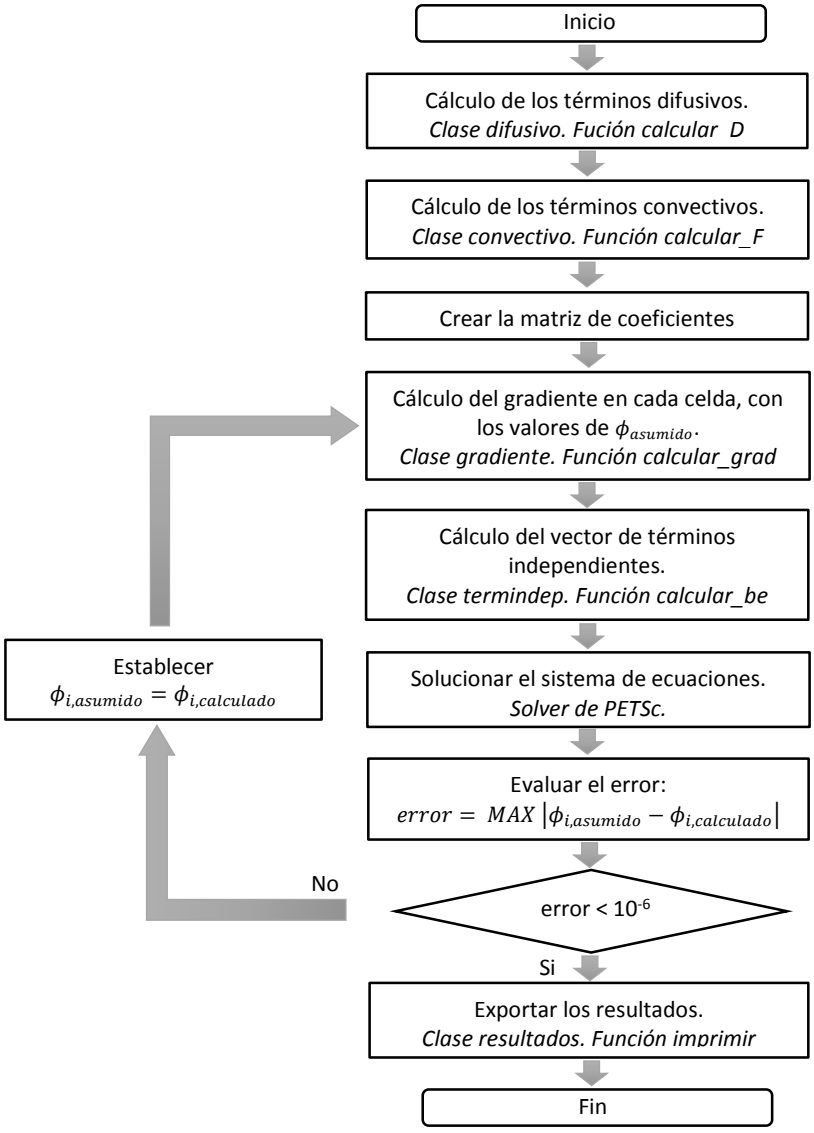
- `calcular_be(geom b, frontera f, Mat matriz_difusivo, Vec vec_diffront, Vec vec_convecfront, Mat matriz_gradphivc, Vec *vec_b)`: Requiere de siete argumentos: un objeto de la clase `geom` y uno de la clase `frontera`, la matriz de términos difusivos, el vector de términos difusivos en las fronteras, el vector de términos convectivos en las fronteras, la matriz de gradientes de la variable, y un puntero al vector donde se guardarán los valores de los términos independientes.

5.4.5 Clase solestable

La clase `solestable` es la encargada de llevar a cabo la solución del problema de difusión-convección en estado estable, esto lo hace mediante la función `calcular_Se(double densidad, double gamma, geom b, frontera f, double *velocidad)`.

En la figura 15 se presenta el diagrama de flujo que representa el procedimiento que se lleva a cabo dentro de la función.

Figura 15. Diagrama de flujo de la función calcular_Se.



6. DIFUSIÓN-CONVECCIÓN EN ESTADO TRANSITORIO

En este capítulo se presentará la discretización del término transitorio de la ecuación de transporte y la integración temporal de los términos anteriores (convectivo, difusivo). Se describen los pasos para llevar a cabo la solución de la ecuación de difusión-convección en estado transitorio y su implementación en el código computacional usando dos aproximaciones: el esquema implícito y el esquema explícito [3].

Al finalizar se presenta el caso de estudio del problema Smith-Hutton [12] para validar la correcta implementación de los términos de la ecuación difusión-convección para un escalar cualquiera.

A la ecuación 5.1 se agrega el término transitorio, como resultado la ecuación a discretizar es:

$$\rho \frac{\partial \phi}{\partial t} + \nabla \cdot (\rho \mathbf{V} \phi) = \nabla \cdot (\Gamma \nabla \phi) \qquad \text{(Ec. 6.1)}$$

Nuevamente, el campo de velocidades $\mathbf{V} = u\,\mathbf{i} + v\,\mathbf{j}$ es conocido en todo el dominio**.**

En el problema transitorio se desea obtener la solución de ϕ en cada instante de tiempo, para este propósito se proporcionan condiciones iniciales de la variable $\phi(x, y, 0)$, y se deben tomar pasos discretos de tiempo Δt.

Al igual que en la ecuación de estado estable, se determina el vector $\mathbf{J} = \rho \mathbf{V} \phi - \Gamma \boldsymbol{\nabla} \phi$. En este caso la ecuación se integra alrededor del volumen de control P y del paso de tiempo Δt.

$$\int_{\Delta t} \int_{d\forall_P} \rho \frac{\partial \phi}{\partial t} d\forall dt + \int_{\Delta t} \int_{d\forall_P} \boldsymbol{\nabla} \cdot \mathbf{J} d\forall dt = 0 \qquad \text{(Ec. 6.2)}$$

Evaluando la integral temporal del primer término y aplicando el teorema de la divergencia en el segundo término se obtiene:

$$\int_{d\forall_P} \rho(\phi^1 - \phi^0) d\forall + \int_{\Delta t} \int_{\mathrm{A}} \mathbf{J} \cdot d\mathbf{A} dt = 0 \qquad \text{(Ec. 6.3)}$$

Los súper-índices 1 y 0 en el primer término de la Ec. 6.3 denotan los valores de tiempo $t + \Delta t$ y t, respectivamente. Al igual que antes se considera que el valor de ϕ en el volumen de control puede ser tomado como el valor en el centroide del mismo, luego es posible escribir:

$$\int_{d\forall_P} \rho\phi d\forall = \rho\phi_P \Delta\forall_P \qquad \text{(Ec. 6.4)}$$

De las ecuaciones 6.3 y 6.4 es posible deducir el término transitorio como:

$$\Delta\forall_P \rho(\phi_P^1 - \phi_P^0) \qquad \text{(Ec. 6.5)}$$

En cuanto al segundo término de la Ec. 6.3, el valor de $\mathbf{J}$ en la cara se puede representar por su valor en el centroide de la misma, de modo que se puede escribir:

$$\int_{\Delta t} \sum_f \mathbf{J}_f \cdot \mathbf{A}_f dt \qquad \text{(Ec. 5.6)}$$

Para evaluar este término es necesario asumir que el flujo $\mathbf{J}$ puede ser interpolado entre los tiempos $t + \Delta t$ y t usando un factor de interpolación f cuyo valor está entre cero y uno.

$$\int_{\Delta t} \mathbf{J} \cdot \mathbf{A} dt = (f\mathbf{J^1} \cdot \mathbf{A} + (1 - f)\mathbf{J^0} \cdot \mathbf{A})\Delta t \qquad \text{(Ec. 6.7)}$$

Realizando la discretización presentada en el capítulo anterior se obtiene la siguiente ecuación discreta para la variable ϕ. Por simplicidad se removió el superíndice 1, de modo que el tiempo $t + \Delta t$ está representado por el valor sin súper-índice.

$$a_P\phi_P = \sum_{nb} a_{nb}(f\phi_{nb} + (1 - f)\phi_{nb}^0) + \left(a_p^0 - (1 - f)\sum_{nb} a_{nb}\right)\phi_P^0 + b \qquad \text{(Ec. 6.8)}$$

Donde,

$$a_{nb} = MAX\left(-F_f, 0\right) + D_f$$

$$a_P^0 = \frac{\rho \Delta \forall}{\Delta t}$$

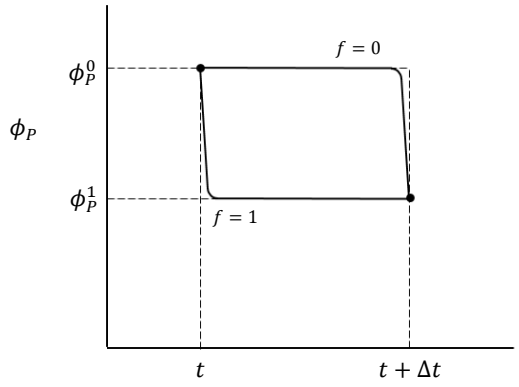
$$a_P = a_P^0 + f\left(\sum_{nb} a_{nb} + \sum_f F_f\right)$$

$$b = f\sum_{nb}\left(\mathcal{S}_f\right)_{nb} + (1-f)\sum_{nb}\left(\mathcal{S}_f\right)_{nb}^0$$

Esta ecuación se aplica a cada uno de los volúmenes de control del dominio que no pertenece a la frontera. En el caso de volúmenes de control frontera, los términos de la ecuación se modifican de la misma forma que en el capítulo anterior.

Los valores de f pueden ser interpretados en términos de la variación de ϕ_P en función de t (Fig. 16), en este trabajo se tienen en cuenta dos casos particulares de la ecuación mencionada para dos valores del término f que serán explicados a continuación.

Figura 16. Variación de la variable ϕ con el tiempo para dos esquemas de discretización temporal.



6.1 ESQUEMA EXPLÍCITO

El caso en que se establece $f = 0$ se conoce como esquema explícito, en este esquema el flujo es evaluado usando valores exclusivamente del paso de tiempo anterior; en la figura 16 se observa que el valor de ϕ_p^0 prevalece durante todo el intervalo de tiempo y cambia cuando llega a $t + \Delta t$.

6.1.1 Ecuación discreta

$$a_P \phi_P = \sum_{nb} a_{nb} \phi_{nb}^0 + \left(a_p^0 - \sum_{nb} a_{nb} \right) \phi_P^0 + b \qquad \text{(Ec. 6.9)}$$

Donde,

$a_{nb} = MAX(-F_f, 0) + D_f$

$a_P^0 = \dfrac{\rho \Delta \forall}{\Delta t}$

$a_P = a_P^0$

$b = \sum_{nb} (\mathcal{S}_f)_{nb}^0$

De la ecuación discreta se puede observar que, conociendo las condiciones en el tiempo t, es posible evaluar los términos de la Ec. 6.9 y encontrar el valor de ϕ_P en el tiempo $t + \Delta t$; no es necesario resolver un sistema de ecuaciones.

Cuando $\Delta t \to \infty$ se obtiene la ecuación de difusión-convección en estado estable, lo cual sucede cuando se alcanza el estado estacionario con el paso del tiempo, luego es seguro que la solución al alcanzar este estado será la misma que si se hubiese obtenido solucionando el problema como estable en primer lugar.

Este esquema resulta sencillo y conveniente, sin embargo, tiene problemas de estabilidad cuando $a_P^0 < \sum a_{nb}$, por esta razón es necesario aplicar restricciones al tiempo usando la condición CFL (Courant-Friedrich-Levy) (Ec. 5.10).

$$\Delta t \left(\frac{\Gamma/\rho}{A_f^2} \right)_{max} \leq C_{visc} \qquad \text{(Ec. 6.10a)}$$

$$\Delta t \left(\frac{|V_i|}{A_f} \right)_{max} \leq C_{conv} \qquad \text{(Ec. 6.10b)}$$

En este caso se tomará $C_{visc} = 0.2$ y $C_{conv} = 0.35$ [11]. Esta condición limita el paso de tiempo máximo posible, de modo que se necesitan tomar pasos cada vez más pequeños al refinar la malla.

En el caso de los problemas que se estudian en este trabajo, se requiere que el programa termine cuando la solución llegue a estado estacionario; se asumirá que la solución llega a este punto cuando la diferencia entre los valores de ϕ^t y $\phi^{t+\Delta t}$ sea menor que 10^{-6}.

6.1.2 Implementación en el código computacional

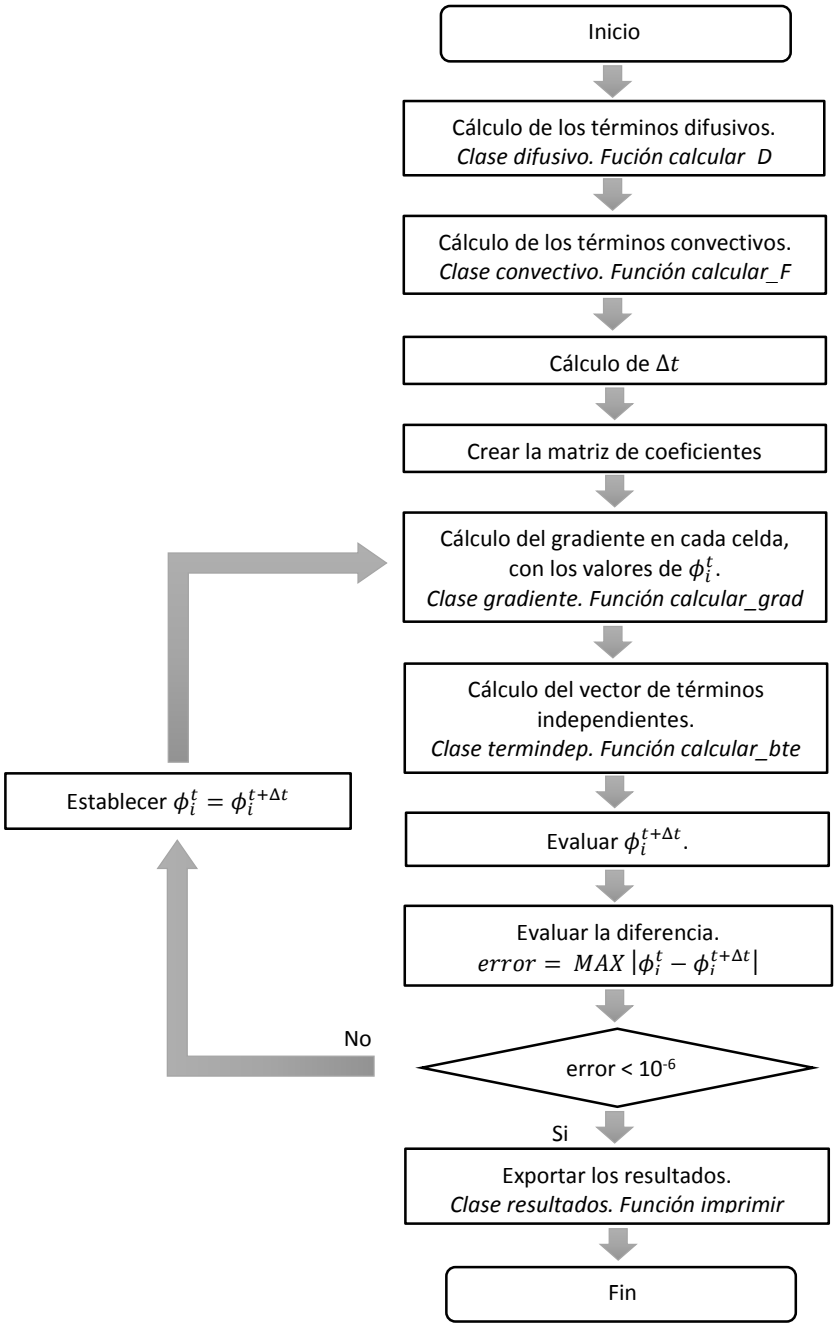
El problema de difusión-convección en estado transitorio se implementa en el código computacional mediante la clase `solexplicito`, la cual contiene una única función encargada de llevar a cabo la solución del problema utilizando el esquema explícito de discretización temporal.

Como se mencionó anteriormente, dentro de esta función se llaman las clases que se encargan de calcular los términos de la ecuación de igual forma a como se explicó en el problema de estado estable, con la diferencia de que en este caso, los términos independientes son diferentes, luego la función para calcular el vector de términos independientes es:

- `calcular_bte(geom b,frontera f, Mat matriz_difusivo, Vec vec_diffront, Vec vec_convecfront, Vec vec_soluant, Mat matriz_gradphivc, Vec *vec_b)`: Los argumentos son los mismos de la función utilizada en el problema de estado estable, más un vector `vec_soluant` con los valores de ϕ en el paso de tiempo anterior.

La función que contiene el procedimiento de solución de este problema es `calcular_Ste(double densidad,double gamma,geom b,frontera f)`. En la figura 17 se presenta el diagrama de flujo que representa el procedimiento que se lleva a cabo dentro de la función.

Figura 17. Diagrama de flujo de la función calcular_Ste.



6.2 ESQUEMA IMPLICITO

El caso en que se establece $f = 1$ se conoce como esquema implícito. En este esquema el valor de ϕ_P cambia de ϕ_p^0 a ϕ_p^1 en el tiempo t y permanece con este valor durante todo el intervalo de tiempo hasta $t + \Delta t$ (Fig. 16).

6.2.1 Ecuación discreta

$$a_P \phi_P = \sum_{nb} a_{nb} \phi_{nb} + b \qquad \text{(Ec. 6.11)}$$

Donde,

$$a_{nb} = MAX\left(-F_f, 0\right) + D_f$$

$$a_P^0 = \frac{\rho \Delta \forall}{\Delta t}$$

$$a_P = a_P^0 + \left(\sum_{nb} a_{nb} + \sum_{f} F_f \right)$$

$$b = \sum_{nb} \left(\mathcal{S}_f \right)_{nb} + a_P^0 \phi_P^0$$

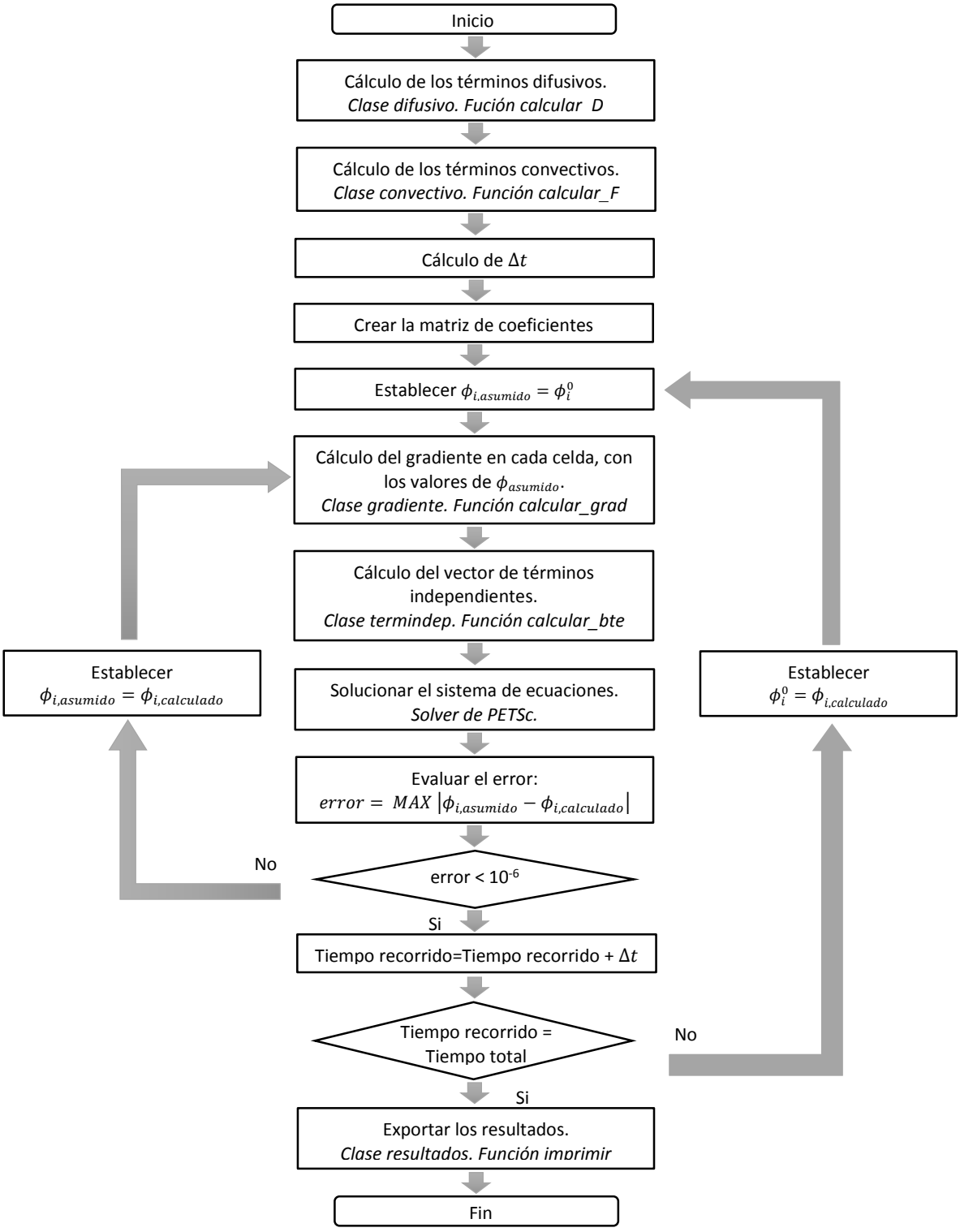
Al igual que en el esquema anterior cuando $\Delta t \to \infty$ se obtiene la ecuación de difusión-convección en estado estable.

En este caso si es necesario la solución del sistema de ecuaciones (Ec. 2.2), el procedimiento de solución es similar al presentado para la solución de la ecuación de difusión-convección en estado estable, con la diferencia de que en este caso se adiciona un bucle que corresponde al recorrido por el tiempo.

Como se puede observar en la ecuación 5.11, no hay posibilidad de que se de $a_P < \sum a_{nb}$, por esta razón este esquema es incondicionalmente estable y no implica ninguna restricción de tiempo, sin embargo, necesita de un mayor tiempo de solución debido al sistema de ecuaciones.

6.2.2 Implementación en el código computacional

Figura 18. Diagrama de flujo de la función calcular_Sti.



En el caso de la discretización temporal implícita, la implementación en el código se realiza mediante la clase `solimplicito`, esta contiene una única función encargada de llevar a cabo la solución del problema.

Al igual que en las anteriores, dentro de esta función se llaman las clases que se encargan de calcular los términos de la ecuación de igual forma a como se explicó en el problema de estado estable, con la diferencia de que en este caso, los términos independientes son diferentes, la función para calcular el vector de términos independientes es:

- `calcular_bti(geom gm,frontera f, Mat matriz_difusivo, Vec vec_diffront, Vec vec_soluant, Mat matriz_gradphivc, Vec vec_ap0, Vec *vec_b)`: Los argumentos son los mismos la función del problema anterior, más un vector con los valores del coeficiente a_P^0 en cada celda.

La función que contiene el procedimiento de solución de este problema es `calcular_Sti(geom b,frontera f,double *velocidad,double densidad,double gamma,double tiempo)`.

En la figura 18 se presenta el diagrama de flujo que representa el procedimiento que se lleva a cabo dentro de la función.

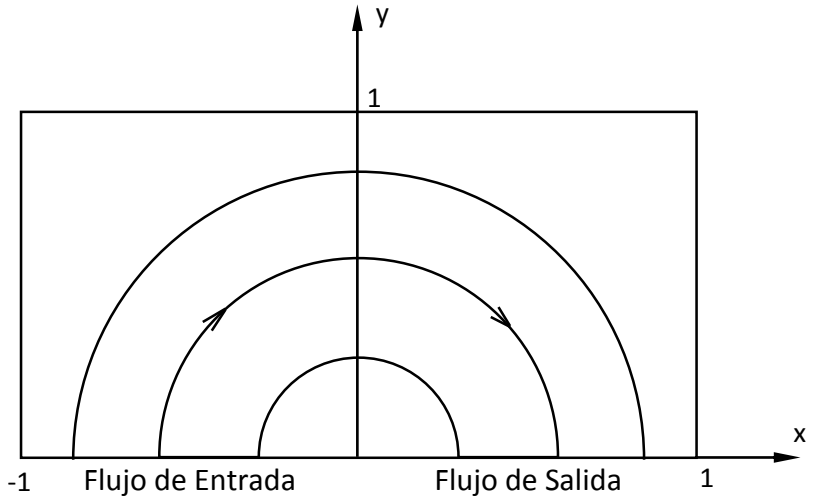
6.3 CASO DE ESTUDIO: PROBLEMA SMITH-HUTTON

Con el fin de validar la discretización de los términos de la ecuación de difusión-convección y el esquema explícito para la discretización temporal, los cuales serán utilizados más adelante en la solución del problema de estudio de este trabajo; se presentan los resultados numéricos obtenidos de la simulación del problema Smith-Hutton.

6.3.1 Descripción del problema

El problema propuesto por Smith y Hutton [12] es uno de los más utilizados en estudios numéricos para evaluar esquemas de difusión-convección. El objetivo es hallar la solución de estado estable de la ecuación de difusión-convección en una región rectangular (Fig. 19).

Figura 19. Esquema del problema Smith-Hutton.



El campo de velocidad está dado por la función de corriente $\psi = (1 - x^2)(1 - y^2)$, con esta, es posible determinar las componentes de la velocidad en dirección x y y.

$$u(x.y) = -\frac{\partial \psi}{\partial y} = 2y(1 - x^2) \qquad \text{(Ec. 6.12a)}$$

$$v(x.y) = \frac{\partial \psi}{\partial x} = -2x(1 - y^2) \qquad \text{(Ec. 6.12b)}$$

Todas las condiciones de frontera del dominio son de tipo Dirichlet, a excepción de la frontera de salida de flujo, allí se tiene una condición de tipo Neumann. Los valores son los siguientes:

$$\phi = 1 + \tanh(\alpha(2x + 1))\,; \qquad y = 0; -1 < x < 0 \ \text{(Flujo de entrada)}$$

$$\frac{\partial \phi}{\partial y} = 0; \qquad y = 0; 0 < x < 1 \ \text{(Flujo de salida)}$$

$$\phi = 1 - \tanh(\alpha)\,; \qquad \text{Demás fronteras}$$

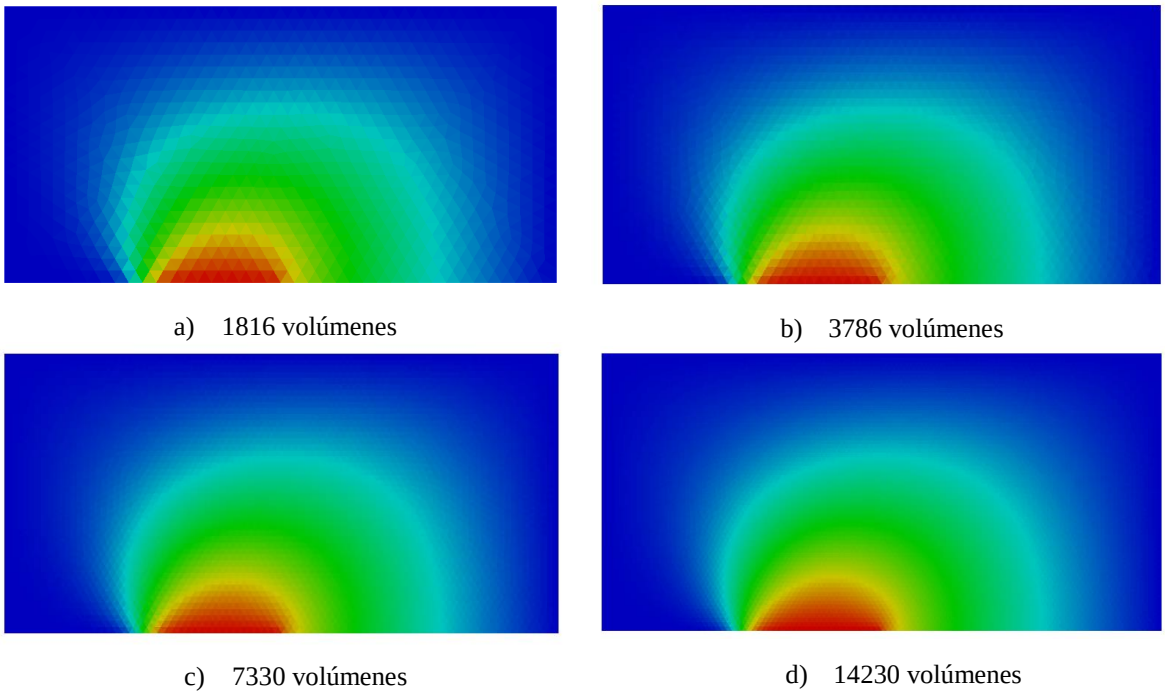
Donde α es un parámetro ajustable que controla el perfil de la variable ϕ en la frontera de entrada. En este caso se toma $\alpha = 10$.

La solución encontrada para la variable ϕ depende de la relación ρ/Γ, su valor indica que fenómeno influye más el difusivo o el convectivo.

El problema se resuelve para dos casos de relación ρ/Γ, los resultados de la simulación se comparan con los resultados encontrados en la literatura [12].

5.3.2 Resultados numéricos

Figura 20. Distribución de ϕ para $\rho/\Gamma = 10$ en diferentes densidades de malla.



a) 1816 volúmenes

b) 3786 volúmenes

c) 7330 volúmenes

d) 14230 volúmenes

Figura 21. Valores de ϕ en la frontera de salida del dominio para $\rho/\Gamma = 10$. Estudio de independencia de malla y comparación con valores de referencia.

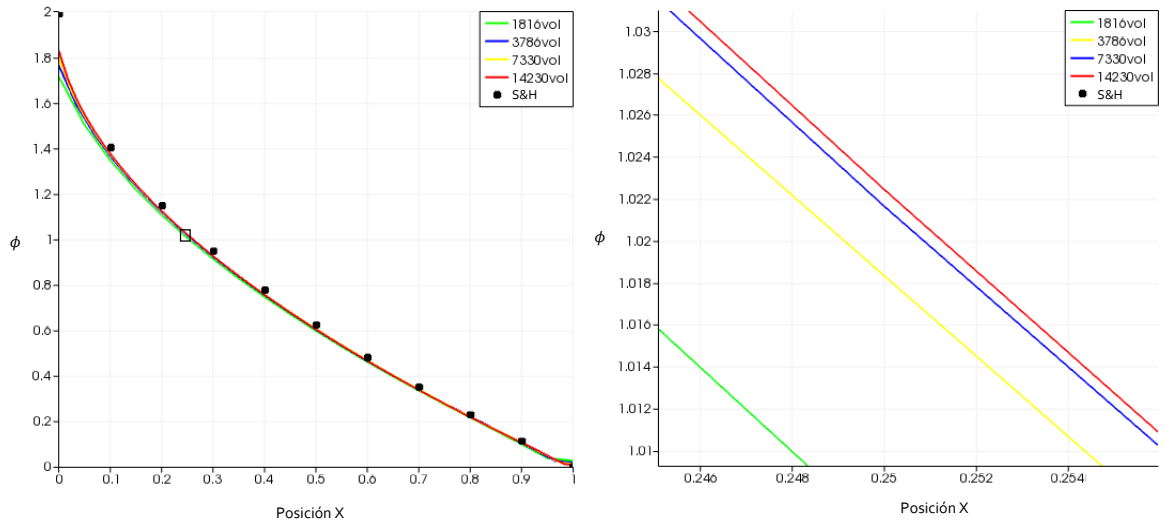
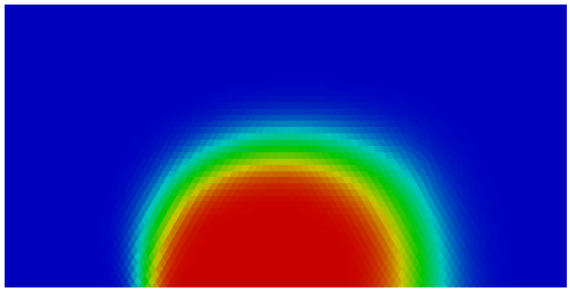
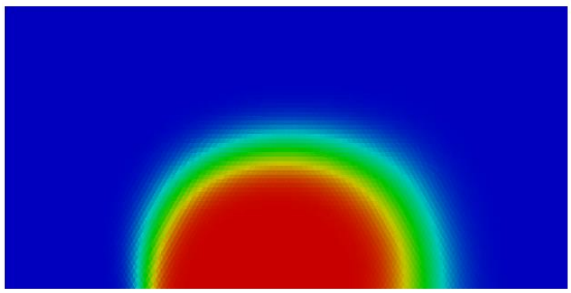


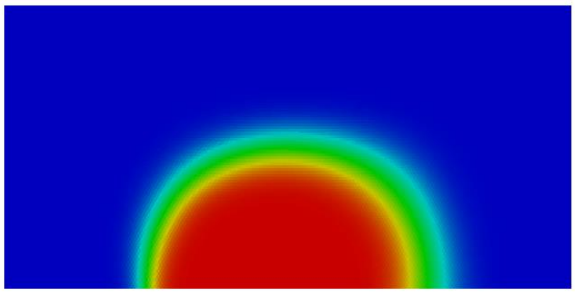
Figura 22. Distribución de ϕ para $\rho/\Gamma = 10^3$ en diferentes densidades de malla.



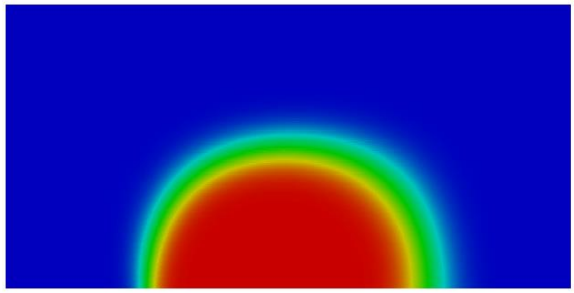
a) 7330 volúmenes



b) 10326 volúmenes



c) 19498 volúmenes



d) 31854 volúmenes

Figura 23. Valores de ϕ en la frontera de salida del dominio para $\rho/\Gamma = 10^3$. Estudio de independencia de malla y comparación con valores de referencia.

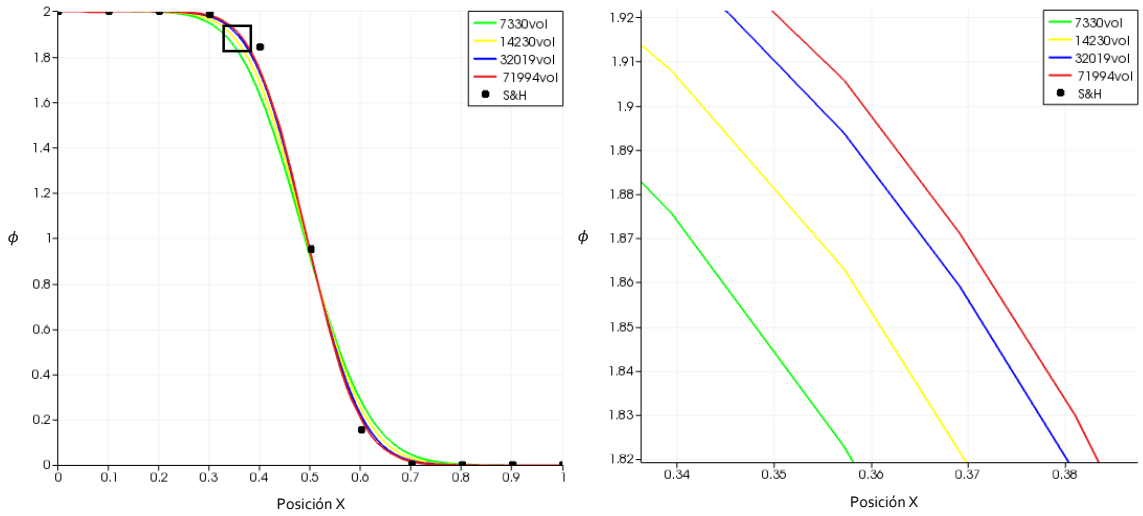
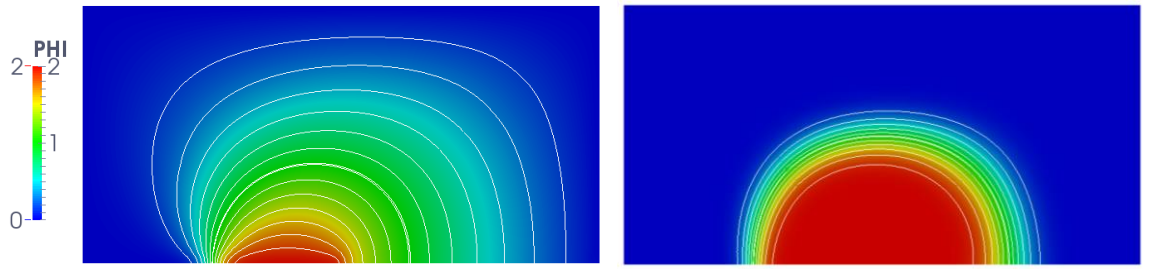


Figura 24. Distribución y líneas de valor constante de ϕ.



a) $\rho/\Gamma = 10$ b) $\rho/\Gamma = 10^3$

6.3.3 Discusión

El problema de Smith-Hutton es un caso de flujo en estado estacionario, la solución numérica presentada se halló al llegar a este punto con la condición descrita anteriormente. En la figura 24 se presentan la distribución y las líneas de valor constante de ϕ para los dos casos estudiados, estos resultados son de gran similitud con los encontrados en la literatura [12].

Con el propósito de asegurar la independencia de la malla de la solución numérica obtenida, se evalúan los dos casos del problema en diferentes densidades de mallas; en las figuras 20 y 22 se presenta la solución de la variable ϕ de ambos casos para cuatro densidades de malla diferentes.

En las figuras 21 y 23 se muestra la gráfica de los valores de ϕ en la frontera de salida del dominio, para los resultados obtenidos y los valores de referencia. En ambas, la figura de la derecha permite observar el comportamiento asintótico de la solución al refinar la malla, lo cual demuestra la consistencia del método numérico.

Se observa que al incrementar la relación ρ/Γ, la malla debe refinarse más para alcanzar una solución cercana a la solución de referencia, esto se debe a la influencia de la convección en el problema.

En el caso de $\rho/\Gamma = 10$, el fenómeno está dominado por la difusión, la convección no influye casi en la transporte de la variable ϕ; caso contrario cuando $\rho/\Gamma = 10^3$, el fenómeno es dominado por la convección y la difusión tiene poco efecto, en este caso, la solución calculada se ve mayormente influenciada por el esquema numérico utilizado para aproximar la variable ϕ en los términos convectivos, el esquema utilizado fue el Upwind. La figura 23 permite observar la estabilidad de este esquema, pues la solución permanece acotada dentro de los valores que establece las condiciones de frontera[19], sin embargo, este es un esquema de primer orden e introduce una falsa difusión en la solución, por lo cual su precisión no es muy alta.

[19] Otros esquemas de discretización de los términos convectivos ocasionan que en la solución de este problema se presenten puntos donde el valor de ϕ sobrepasa los valores de las condiciones de frontera.

7. EVALUACIÓN DEL CAMPO DE FLUJO

Hasta el momento se consideró el fenómeno de difusión-convección de un campo escalar en la presencia de un campo de velocidades conocido. En este capítulo se considera el fenómeno del flujo de un fluido donde el campo de velocidades no se conoce y se quiere hallar, para esto se deben solucionar las ecuaciones de conservación de cantidad de movimiento presentadas en el capítulo 1 (Ec. 1.6a) y Ec. 1.6b).

$$\frac{\partial u}{\partial t} + \nabla \cdot (\mathrm{V}u) = \nabla \cdot (v \nabla u) - \frac{1}{\rho} \nabla p \cdot \mathrm{i}$$

$$\frac{\partial v}{\partial t} + \nabla \cdot (\mathrm{V}v) = \nabla \cdot (v \nabla v) - \frac{1}{\rho} \nabla p \cdot \mathrm{j}$$

Estas ecuaciones tienen la misma forma de la ecuación general de convección-difusión, la cual ya se ha explicado cómo se ha de discretizar. El inconveniente es que no se conoce el campo de presión y no se tiene una ecuación explícita para hallarlo, es por esta razón que se debe recurrir a la ecuación de continuidad (Ec. 1.5) como ecuación extra.

$$\nabla \cdot (\rho \mathrm{V}) = 0$$

En general, existen varios métodos para la solución del problema de campo de flujo en un fluido incompresible; entre los más conocidos se encuentran los métodos de tipo predicción-corrección como son el FSM (Fractional Step Method) [8][9], el algoritmo SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) y sus variaciones: SIMPLER, SIMPLEC [3].

La idea principal del algoritmo SIMPLE es crear una ecuación de corrección de la presión, a partir de la ecuación de continuidad. El procedimiento general es:

1. Asumir un campo de presión p^*.
2. Solucionar las ecuaciones de momento usando p^*. Se obtiene la solución para los campos de velocidad u^* y v^*.

3. Solucionar la ecuación obtenida para la presión con los valores de u^* y v^*.
4. Corregir los campos de presión y velocidades, de modo que satisfagan la ecuación de continuidad.

En el método del paso fraccionado, se calcula una solución aproximada de las ecuaciones de momento sin tener en cuenta los gradientes de presión, esta solución se denota como velocidad predictora y no satisface la condición de incompresibilidad; el campo de presión se calcula posteriormente mediante la ecuación de Poisson y el campo de velocidades predictoras, siendo este el único sistema de ecuaciones que se debe resolver; finalmente se calcula la velocidad real a partir de la velocidad predictora y el gradiente de la presión hallada.

Los métodos FSM se han convertido en una técnica muy popular para la resolución de las ecuaciones incompresibles de Navier-Stokes. Esto se debe a que tienen un mejor rendimiento que los algoritmos de tipo SIMPLE; y a su simplicidad para llevarlo al código computacional.
Por esta razón, el método utilizado en este trabajo para solucionar el acoplamiento entre la velocidad y la presión es el método de paso fraccionado.

7.1 PROBLEMA CHECKERBOARDING: Tablero de ajedrez

La discretización de la ecuación de continuidad es de la forma:

$$\int_{\mathrm{A}} (\rho \mathbf{V}) \cdot d\mathbf{A} = \sum_{f} (\rho \mathbf{V})_f \cdot \mathbf{A}_f = 0 \qquad \text{(Ec. 7.1)}$$

Al no conocer el valor de la velocidad en la cara f es necesario de un esquema de interpolación; usualmente se utiliza el esquema de diferencias centradas (Ec. 7.2).

$$u_f = \frac{u_P + u_F}{2} \quad y \quad v_f = \frac{v_P + v_F}{2} \qquad \text{(Ec. 7.2)}$$

Al evaluar estos términos en la ecuación discreta (Ec. 7.1) obtenemos que la ecuación de continuidad para la celda P no contiene el valor de la velocidad de la celda P, depende únicamente de los valores de velocidad de sus vecinos:

$$\sum_f (\rho \mathbf{V})_f \cdot \mathbf{A}_f = \sum_f \rho \mathbf{V}_F \cdot \mathbf{A}_f = 0 \qquad \text{(Ec. 7.3)}$$

Este hecho permite un efecto de zig-zag en el campo de velocidad (Problema de Checkerboard, Fig. 25), como el campo de presión se obtiene a partir de la velocidad, es posible que se creen campos de presión con el mismo efecto.

Figura 25. Campo de velocidad con efecto de tablero de ajedrez.

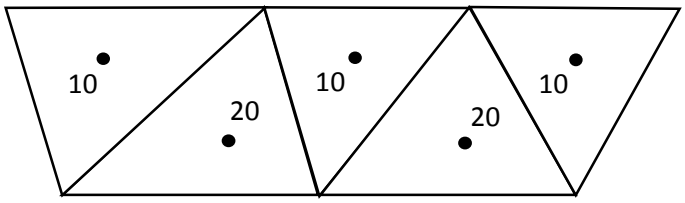
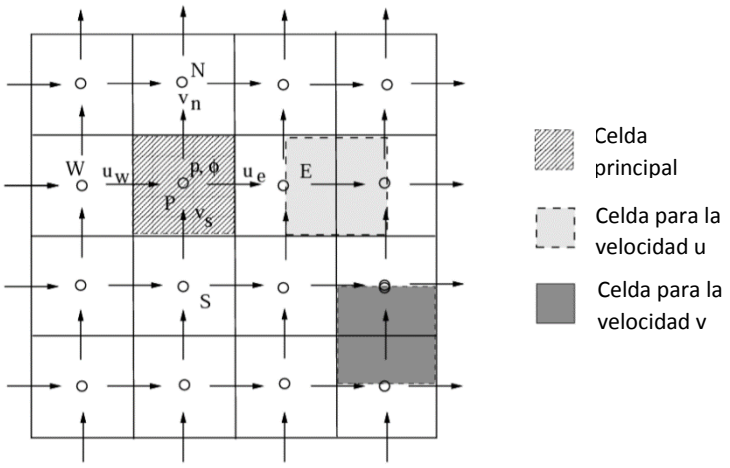


Figura 26. Malla desplazada en malla estructurada.



Fuente: PATANKAR, S. V. Numerical Heat Transfer and Fluid Flow.

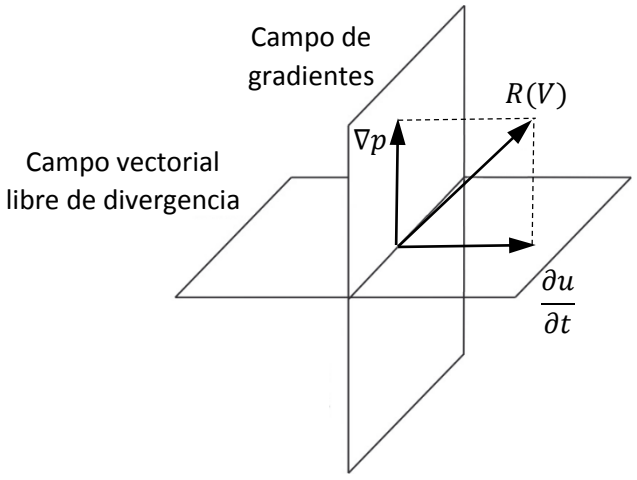
La solución común al problema mencionado es utilizar mallas desplazadas (Fig. 26) para ubicar los valores de la velocidad directamente en las caras; sin embargo, en mallas no estructuradas no es posible determinar una malla desplazada de forma evidente, luego necesariamente se deben utilizar mallas colocadas.

En el caso de mallas colocadas, el algoritmo FSM reduce el problema del checkerboarding al determinar las siguientes acciones: en la estimación de la velocidad no se tienen en cuenta los efectos de la presión, al calcular la presión centrada se utilizan las velocidades interpoladas a las caras, y al calcular la velocidad corregida se utiliza la presión interpolada en las caras.
Además se evita el esquema de diferencias centradas para la interpolación de la velocidad y de la presión, se utilizan esquemas más elaborados [6][10].

7.2 MÉTODO DEL PASO FRACCIONADO

Los métodos de paso fraccionado también son conocidos como métodos de proyección. Se denota un campo vectorial de velocidades $R(V)$ que se descompone en dos campos ortogonales (Fig. 27): uno de gradientes de presión y un campo vectorial libre de divergencia; la presión se convierte en un operador que proyecta este campo de velocidades en el campo vectorial libre de divergencia.

Figura 27. Descomposición única del campo vectorial $R(V)$.



7.2.1 Planteamiento del método

El método FSM se basa en la aplicación del teorema de Helmholtz-Hodge [11] a la ecuación vectorial de momento, de la siguiente forma:

$$\Pi\left(\frac{\partial \mathbf{V}}{\partial t}+\frac{1}{\rho}\nabla p\right)=\Pi\left(-(\nabla\cdot\mathbf{V})\mathbf{V}+\nabla\cdot(\upsilon\nabla\mathbf{V})\right) \quad \text{(Ec. 7.4)}$$

Donde $\Pi(\cdot)$ es un operador de proyección, que proyecta un campo vectorial en un campo libre de divergencia.

El campo de velocidad es incompresible, luego el término transitorio permanece invariante al aplicar el operador (Ec. 7.5); por otro lado, el gradiente de la presión pertenece al plano ortogonal al campo libre de divergencia, luego la proyección es nula (Ec. 7.6).

$$\Pi\left(\frac{\partial \mathbf{V}}{\partial t}\right)=\frac{\partial \mathbf{V}}{\partial t} \quad \text{(Ec. 7.5)}$$

$$\Pi\left(\frac{1}{\rho}\nabla p\right)=0 \quad \text{(Ec. 7.6)}$$

Por lo tanto, las ecuaciones de Navier-Stokes pueden ser divididas en dos partes: Un vector libre de divergencia (Ec. 7.7) y el gradiente de un campo escalar (Ec. 7.8).

$$\frac{\partial \mathbf{V}}{\partial t}=\Pi\left(-(\nabla\cdot\mathbf{V})\mathbf{V}+\nabla\cdot(\upsilon\nabla\mathbf{V})\right) \quad \text{(Ec. 7.7)}$$

$$\frac{1}{\rho}\nabla p=\left(-(\nabla\cdot\mathbf{V})\mathbf{V}+\nabla\cdot(\upsilon\nabla\mathbf{V})\right)-\Pi\left(-(\nabla\cdot\mathbf{V})\mathbf{V}+\nabla\cdot(\upsilon\nabla\mathbf{V})\right) \quad \text{(Ec. 7.8)}$$

Finalmente, al aplicar el operador divergencia a la Ec. 7.8 se obtiene la ecuación de Poisson para la presión:

$$\frac{1}{\rho}\nabla p=\nabla\cdot\left(-(\nabla\cdot\mathbf{V})\mathbf{V}+\nabla\cdot(\upsilon\nabla\mathbf{V})\right) \quad \text{(Ec. 7.9)}$$

El término entre paréntesis será representado por el vector $R(\mathrm{V})$:

$$R(\mathrm{V})=-(\nabla\cdot\mathbf{V})\mathbf{V}+\nabla\cdot(\upsilon\nabla\mathbf{V}) \quad \text{(Ec. 7.10)}$$

Es posible observar que el papel del gradiente de la presión, es proyectar el vector $R(\mathrm{V})$ en un campo libre de divergencia (Fig. 27).

7.2.2 Discretización temporal

La forma final del método del paso fraccionado depende del método de discretización temporal utilizado. La ecuación de momento en dirección x (Ec. 1.6a) puede reescribirse como:

$$\frac{\partial u}{\partial t} = R(u) - \frac{1}{\rho}\nabla p \cdot \mathbf{i} \qquad \text{(Ec. 7.11)}$$

Con $R(u) = -\nabla \cdot (\mathbf{V}u) + \nabla \cdot (\upsilon \nabla u)$.

Para el término transitorio se usa el esquema de diferencias centradas:

$$\left.\frac{\partial u}{\partial t}\right|^{n+1/2} = \frac{u^{n+1} - u^n}{\Delta t} \qquad \text{(Ec. 7.12)}$$

Para los términos convectivos y difusivos agrupados en $R(u)$ se utiliza el esquema de discretización explícito de segundo-orden Adams-Bashforth [11].

$$R(u^{n+1/2}) = \frac{3}{2}R(u^n) - \frac{1}{2}R(u^{n-1}) \qquad \text{(Ec. 7.13)}$$

Para el término del gradiente de la presión se utiliza un esquema de primer orden hacia atrás de Euler. Reemplazando todo en la Ec. 7.11 se obtiene:

$$\frac{u^{n+1} - u^n}{\Delta t} = \frac{3}{2}R(u^n) - \frac{1}{2}R(u^{n-1}) - \frac{1}{\rho}\nabla p^{n+1} \cdot \mathbf{i} \qquad \text{(Ec. 7.14)}$$

Se determina una velocidad predictora en dirección x:

$$u^p = u^n + \Delta t\left(\frac{3}{2}R(u^n) - \frac{1}{2}R(u^{n-1})\right) \qquad \text{(Ec. 7.15)}$$

Los valores de u^p y v^p[20] componen el vector de velocidad predictora $\mathbf{V}^p = u^p\mathbf{i} + v^p\mathbf{j}$, y se utiliza el teorema Helmholtz-Hodge para descomponer este vector en:

[20] Las ecuaciones descritas para la velocidad en dirección x se aplican de igual forma para la velocidad en dirección y (v).

$$\mathbf{V}^{p} = \mathbf{V}^{n+1} + \frac{\Delta t}{\rho}\nabla p^{n+1} \qquad \text{(Ec. 7.16)}$$

Es necesario aclarar que la ecuación de continuidad únicamente se cumple en el instante de tiempo $n+1$:

$$\nabla \cdot \mathbf{V}^{n+1} = 0 \qquad \text{(Ec. 7.17)}$$

Al aplicar la divergencia a la Ec. 7.16, se obtiene:

$$\nabla \cdot \mathbf{V}^{p} = \nabla \cdot \mathbf{V}^{n+1} + \nabla \cdot \left(\frac{\Delta t}{\rho}\nabla p^{n+1}\right) \qquad \text{(Ec. 7.18)}$$

Teniendo en cuenta la Ec. 7.17, es posible obtener la siguiente ecuación de Poisson para el campo escalar p^{n+1}.

$$\Delta\, p^{n+1} = \frac{\rho}{\Delta t}\nabla \cdot \mathrm{V}^{p} \qquad \text{(Ec. 7.19)}$$

Una vez se calcule el campo de presión es posible obtener la solución de la velocidad en el tiempo $n+1$, mediante la siguiente corrección:

$$u^{n+1} = u^{n} - \frac{\Delta t}{\rho}\nabla p^{n+1} \cdot \mathbf{i} \qquad \text{(Ec. 7.20a)}$$

$$v^{n+1} = v^{n} - \frac{\Delta t}{\rho}\nabla p^{n+1} \cdot \mathbf{j} \qquad \text{(Ec. 7.20b)}$$

En conclusión, el algoritmo que se debe realizar en cada paso de tiempo es:

1. Evaluar $R(u^{n}), R(u^{n-1}), R(v^{n}), R(v^{n-1})$
2. Evaluar $u^{p}\ y\ v^{p}$ de la Ec. 7.15
3. Resolver la ecuación de Poisson (Ec. 7.19)
4. Calcular el nuevo campo de velocidad (Ec. 7.20).

Como se mencionó anteriormente, los esquemas temporales explícitos introducen severas restricciones en cuanto al valor del paso del tiempo Δt; es por esto que debe ser limitado utilizando la condición de estabilidad explicada en la sección 6.1.

En el caso de los problemas que se estudian en este trabajo, se requiere que el programa termine cuando la solución llegue al estado estacionario; se asumirá que la solución llega a este punto cuando la diferencia entre los valores de $u^n \; y \; u^{n+1}$ y $v^n \; y \; v^{n+1}$ sea menor que 10^{-6}.

7.2.3 Ecuación discreta

1. Discretización del término R

El término R consiste en la unión del término convectivo y el término difusivo, la discretización de ambos términos fue expuesta en el capítulo 4; de la Ec. 4.19, reemplazando la variable ϕ por u se obtiene:

$$R(u) = \sum_{nb} \Big(\big(D_f (u_{nb} - u_P) + \mathcal{S}_f - MAX\big(F_f, 0\big) u_P + MAX\big(-F_f, 0\big) u_{nb} \big) \Big) \quad \text{(Ec. 7.21)}$$

2. Discretización de la ecuación de Poisson para la presión

El operador laplaciano Δ se discretizó en el capítulo 4 como operador del término difusivo, los término D_f y $\mathcal{S}_f$ son los mismos para la ecuación de velocidad como para la ecuación de presión. El operador divergencia también se discretizó anteriormente como parte del término convectivo (Flujo másico F_f, Ec 4.8). Partiendo de esto, se obtiene la ecuación discreta:

$$\sum_{nb} D_f (p_P - p_{nb}) = \sum_{nb} \Big(\big(\mathcal{S}_f \big)_{nb} - \Big(u_f^p A_{x_f} + v_f^p A_{y_f} \Big)_{nb} \Big) \quad \text{(Ec. 7.22)}$$

Luego se tiene una ecuación de la forma:

$$a_P \phi_P = \sum_{nb} a_{nb} \phi_{nb} + b$$

Donde,

$a_{nb} = D_f$

$$a_P = \sum_{nb} a_{nb}$$

$$b = \sum_{nb} \left(\left(\mathcal{S}_f \right)_{nb} - \left(u_f^p A_{x_f} + v_f^p A_{y_f} \right) \right)$$

Al aplicar la ecuación discreta en cada uno de los volúmenes de control se obtiene un sistema de ecuaciones lineales mencionado en la sección 2.3 (Ec. 2.2).

$$\boldsymbol{A}\phi = B$$

Para la solución de este sistema de ecuaciones[21], se utiliza el método de solución de mínimos residuos generalizados (GMRES) [23], este método presenta buenas características de convergencia y permite obtener una solución para cualquier condición de la matriz de coeficientes (en cuanto a simetría y positividad).

Además, se utiliza el método de precondicionamiento Multigrid algebraico [3], con el fin de acelerar la convergencia de la solución; esta técnica de aceleración resulta eficiente cuando se utilizan mallas no uniformes, y ha sido ampliamente usada en el problema del flujo en un canal con expansión [9][16].

7.2.4 Implementación en el código computacional

El problema de evaluación del campo de flujo se implementa en el código computacional mediante la clase `fractional`, la cual contiene una única función encargada de llevar a cabo la solución del problema.

Como se mencionó anteriormente, dentro de esta función se llaman las clases que se encargan de calcular los términos de la ecuación de igual forma a como se explicó en los problemas anteriores, con la diferencia de que en este caso, los términos convectivos y los términos independientes son diferentes; la función para calcular los términos convectivos que se requieren para evaluar $R(u)$ y $R(v)$ es:

[21] En el Anexo A se hace una breve descripción del método utilizado.

- `calcular_Ff(double densidad, geom b, Vec vec_u, Vec vec_v, frontera fu, frontera fv, Mat *matriz_convectivo, Vec *vec_convecfront)`: Requiere de ocho argumentos: el valor de la densidad, un objeto de la clase `geom`, dos vectores con los valores de la velocidad u y v para cada una de las celdas del dominio, dos objetos de la clase frontera con la información de frontera de u y v, un puntero a la matriz donde se va a guardar los términos convectivos de los volúmenes de control, y un puntero al vector donde se guardarán los términos convectivos de las fronteras.

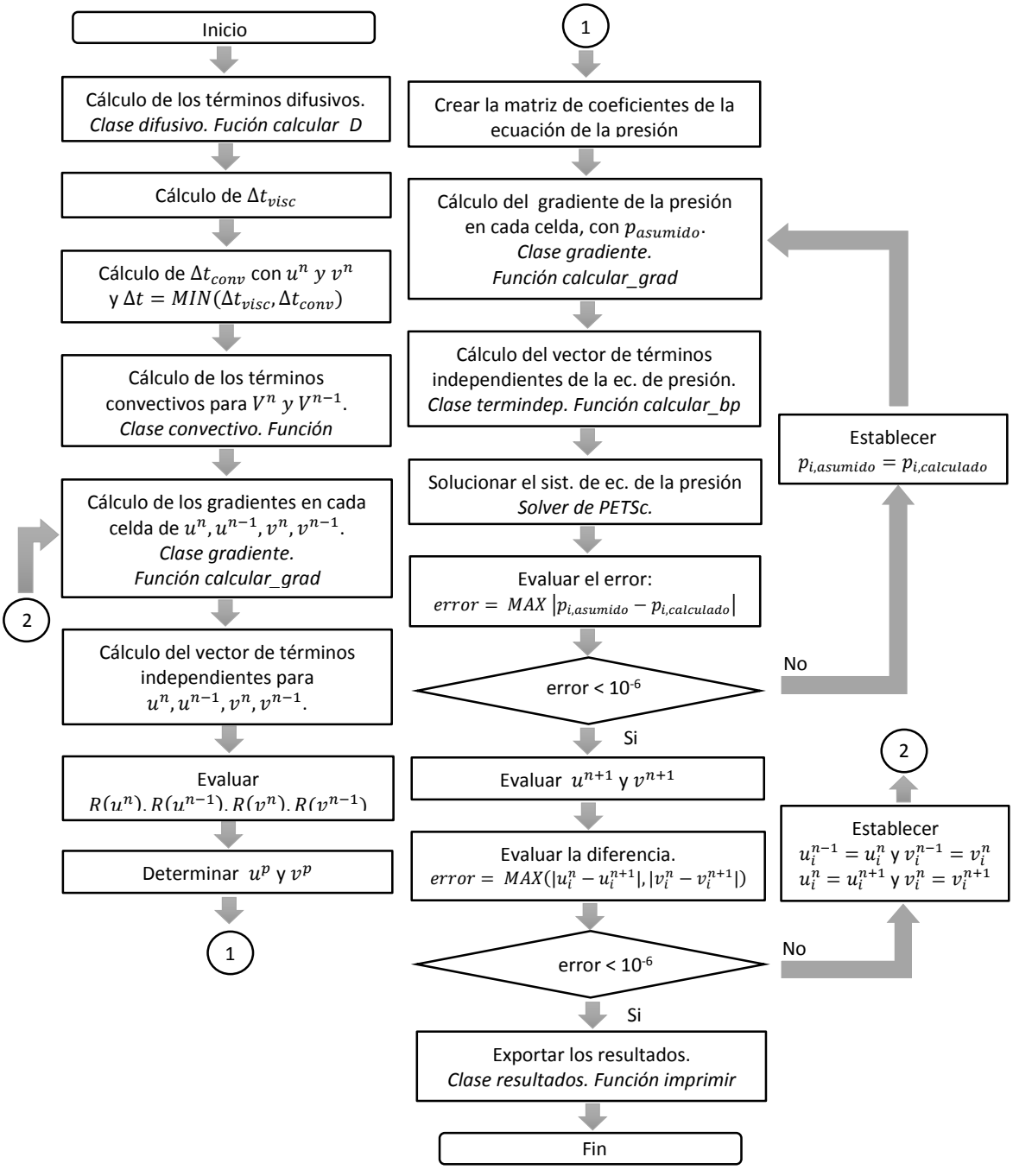
La función para calcular los términos independientes de la ecuación de presión es:

- `calcular_bp(geom b, frontera f, Mat matriz_difusivo, Vec vec_diffront, Mat matriz_gradp, Mat matriz_upf, Mat matriz_vpf, double densidad, double deltat, Vec *vec_b)`: Los argumentos son los mismos que la función del problema anterior de esta clase, más dos matrices con los valores de las velocidades predictoras interpoladas a las caras, el valor de la densidad, y el valor del Δt de la ecuación.

La función que contiene el procedimiento de solución de este problema es `calcular_SF(geom b,frontera fu,frontera fv, frontera fp, double viscosidad, double densidad)`.

En la figura 28 se presenta el diagrama de flujo que representa el procedimiento que se lleva a cabo dentro de la función.

Figura 28. Diagrama de flujo de la función calcular_Sf



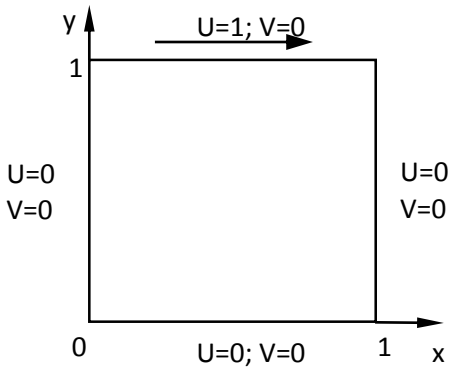
7.3 CASO DE ESTUDIO: DRIVEN CAVITY

Con el fin de validar el esquema numérico desarrollado para la solución de las ecuaciones de gobierno en un campo de flujo, se presentan los resultados numéricos obtenidos de la simulación del problema Driven Cavity.

7.3.1 Descripción del problema

El problema conocido como Driven Cavity [13] es ampliamente utilizado para probar y evaluar técnicas numéricas de solución de las ecuaciones de Navier Stokes. Consiste en un flujo laminar e incompresible que se encuentra en una cavidad cuadrada, cuya pared superior se mueve con una velocidad uniforme en su mismo plano, generando un fenómeno de convección forzada, las demás paredes de la cavidad permanecen sin movimiento (Fig. 29).

Figura 29. Esquema del problema Driven Cavity.



Este es un problema bidimensional en estado estacionario, la razón por la cual es de gran interés en la literatura, es debido a que presenta una geometría y condiciones de frontera simples, y, a su vez, introduce cierta dificultad en su solución, debido a los altos gradientes que se presentan en las zonas cercanas a las paredes.

Las condiciones de frontera para la velocidad son de tipo Dirichlet y para la presión son de tipo Newman:

$$u = 1; v = 0; \frac{\partial p}{\partial y} = 0; \qquad y = 1; 0 < x < 1$$

$$u = 0; v = 0; \frac{\partial p}{\partial y} = 0; \qquad y = 0; 0 < x < 1$$

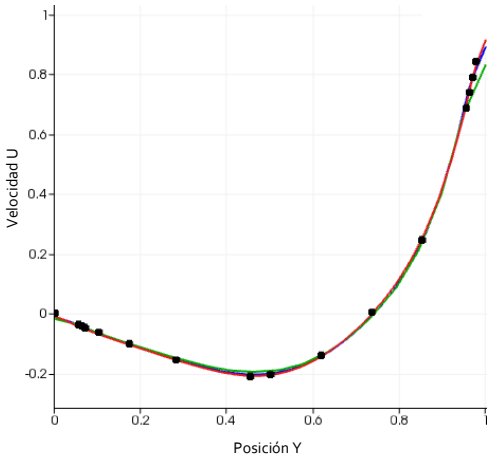
$$u = 0; v = 0; \frac{\partial p}{\partial x} = 0; \qquad x = 0; 0 < y < 1$$

$$u = 0; v = 0; \frac{\partial p}{\partial x} = 0; \qquad x = 1; 0 < y < 1$$

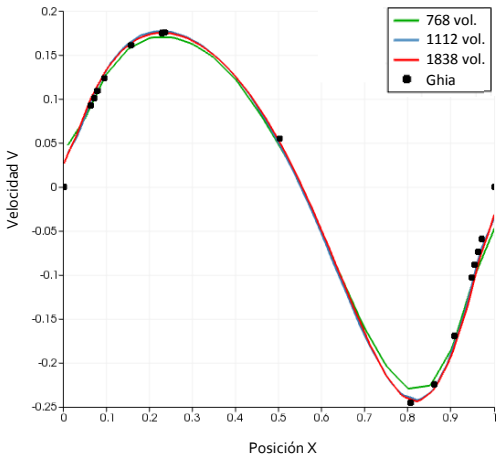
El problema se resuelve para dos números de Reynolds, y su solución se compara con valores de referencia tomados de Ghia et. al. [13].

7.3.2 Resultados numéricos

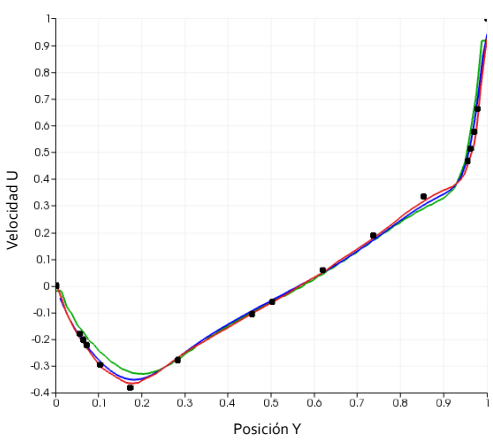
Figura 30. Velocidades en los ejes centrales de la cavidad. Estudio de independencia de malla y comparación con valores de referencia de Ghia et. al.



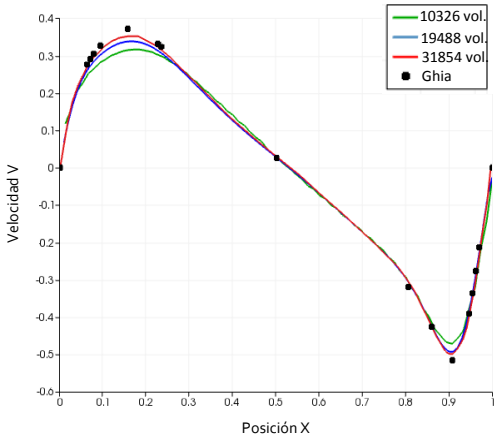
a) $Re_D = 100; x = 0{,}5$



b) $Re_D = 100; y = 0{,}5$



c) $Re_D = 1000; x = 0{,}5$



d) $Re_D = 1000; y = 0{,}5$

Tabla 1. Errores relativos asociados a las mallas empleadas.

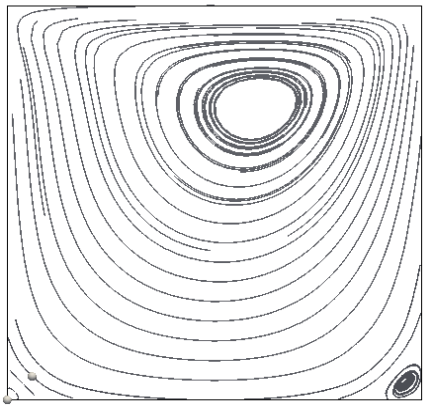
$MALLA$ $(no.\,volúmenes)$	$ERROR$ (%)		
	$U_{mín}$	$V_{mín}$	$V_{máx}$
419	4,628	4,591	2,743
1112	1,689	1,419	1,092
1838	1,167	0,830	0,735

a) $Re_D = 100$

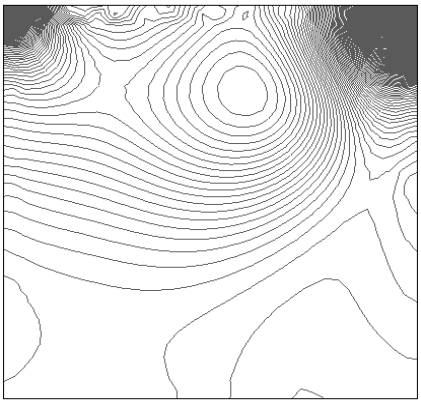
$MALLA$ $(no.\,volúmenes)$	$ERROR$ (%)		
	$U_{mín}$	$V_{mín}$	$V_{máx}$
10326	14,129	14,655	8,649
19498	7,349	7,624	2,957
31854	4,777	4,947	2,180

b) $Re_D = 1000$

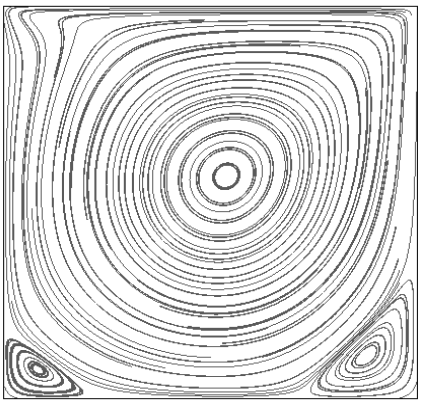
Figura 31. Líneas de corriente y líneas de presión constante en la cavidad.



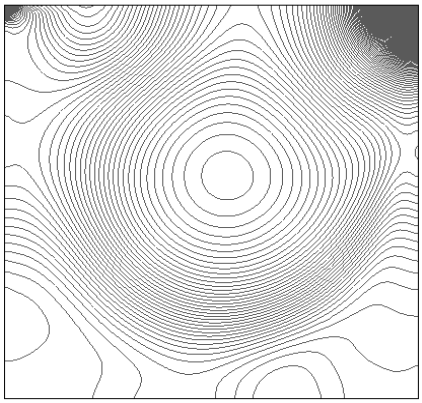
a) Líneas de corriente. $Re_D = 100$



b) Líneas de presión. $Re_D = 100$



c) Líneas de corriente. $Re_D = 1000$



d) Líneas de presión. $Re_D = 1000$

7.3.3 Discusión

Al igual que el Smith-Hutton, este es un problema de flujo en estado estacionario, el mismo criterio fue utilizado para determinar cuándo la solución temporal alcanza la este estado.

Con el propósito de asegurar la independencia de la malla de la solución numérica obtenida, se evalúan los dos casos del problema en tres densidades de malla cada uno; en la figura 30 se presenta la solución de ambos casos del problema para las mallas estudiadas y la solución de referencia tomada de [13].

La tabla 1 presenta los errores relativos asociados a las mallas empleadas para tres valores representativos de la solución: valor mínimo de la componente u de la velocidad en el eje vertical que pasa por el centro de la cavidad, valor mínimo y máximo de la componente v de la velocidad en el eje horizontal que pasa por el centro de la cavidad.

Se observa que al incrementar el número de Reynolds, la malla debe refinarse más para alcanzar una solución cercana a la solución de referencia, esto se debe a la influencia de la convección en el problema. Al aumentar el número de Reynolds las fuerzas convectivas ejercen un mayor dominio sobre las fuerzas viscosas, la velocidad será mayor, y por lo tanto los gradientes en las zonas cercanas a las

paredes serán más altos también debido a la condición de no deslizamiento; en la figura 30 (b y d) se puede observar que las pendientes de la velocidad v cercanas a las paredes son más elevadas para $Re = 1000$. Esta es la razón por la cual se debe refinar más la malla al aumentar el número de Reynolds, ya que se necesita una malla más fina para percibir los gradientes conforme se hacen más altos.

En la figura 31 se presentan las líneas de corriente y el mapa de presiones para los dos casos estudiados, estos resultan ser muy similares a los encontrados en la literatura [7] [13].

7.4 CASO DE ESTUDIO: FLUJO EN UN CANAL CON ESCALÓN RECTO

Como se mencionó anteriormente, no se han publicado estudios del flujo en un canal con escalón inclinado que presenten resultados para flujo laminar; sin embargo, para el caso del escalón recto, hay una gran cantidad de publicaciones al respecto. Por esta razón, además de validar el esquema numérico mediante el problema Driven Cavity, también se hizo el estudio para el problema de flujo en un canal con escalón recto, el cual es más cercano al problema en que se centra el presente trabajo.

7.4.1 Descripción del problema

La descripción del problema del flujo en un canal con escalón se explicó en el primer capítulo; el objetivo principal, es encontrar la posición del punto de reencuentro del flujo; el punto de separación no se tiene en cuenta, debido a que cuando el escalón es recto la separación se da en el inicio del mismo[22].

[22] El estudio de Biswas et. al. [16] confirma que para canales con escalón recto y $Re_D > 10$ el punto de separación se encuentra en la esquina donde comienza la expansión; la región de recirculación cubre completamente la cara del escalón.

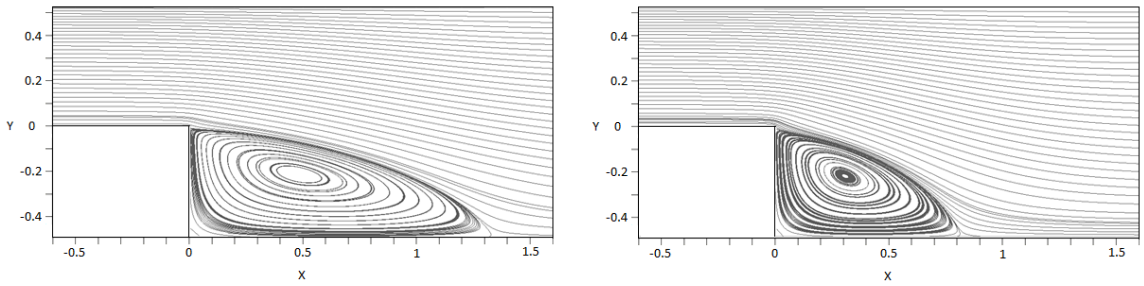
Al igual que los problemas anteriores, este es un problema de flujo en régimen permanente, el mismo criterio anterior fue utilizado para determinar cuándo la solución temporal alcanza la estabilidad.

El estudio se realizó para dos números de Reynolds que cumplen con la condición de flujo laminar, y pertenecen al rango en el que el flujo se considera bidimensional ($Re_D < 400$). La relación de expansión se tomó como $H/h = 1.9423$, este valor fue considerado en los estudios experimentales de Armaly et. al. [14] y ha sido utilizado en varios estudios numéricos.

La posición encontrada del punto de reencuentro se compara con valores de referencia tomados de Armaly et al. [14].

7.4.2 Resultados numéricos

Figura 32. Líneas de corriente en la región cercana al escalón.



a) $Re_D = 100$ b) $Re_D = 50$

Tabla 2. Posición del punto de reencuentro. Estudio de independencia de malla y errores relativos asociados.

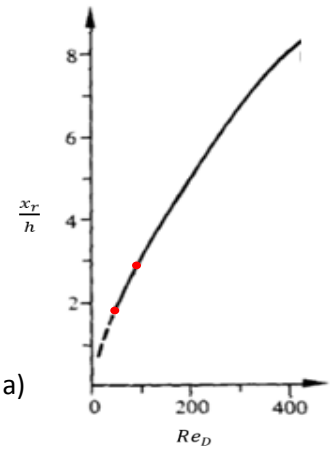
$\boldsymbol{MALLA}$ $\boldsymbol{A_{mín}/h}$	$\boldsymbol{x_r/h}$	$\boldsymbol{Error}$ (%)
$\mathbf{0,05102}$	2,59327	12,131
$\mathbf{0,04082}$	2,75510	6,648
$\mathbf{0,03265}$	2,78816	5,528

a) $Re_D = 100$

$MALLA$ $A_{mín}/h$	x_r/h	$Error$ (%)
0,05102	1,58410	11,129
0,04082	1,67551	6,007
0,03265	1,68163	5,657

b) $Re_D = 50$

Figura 33. (a) Gráfica de referencia de la variación de la posición del punto de reencuentro con respecto al número de Reynolds. Tomada de Armaly et. al. (b) Valores para los números de Reynolds estudiados obtenidos mediante el programa G3data.

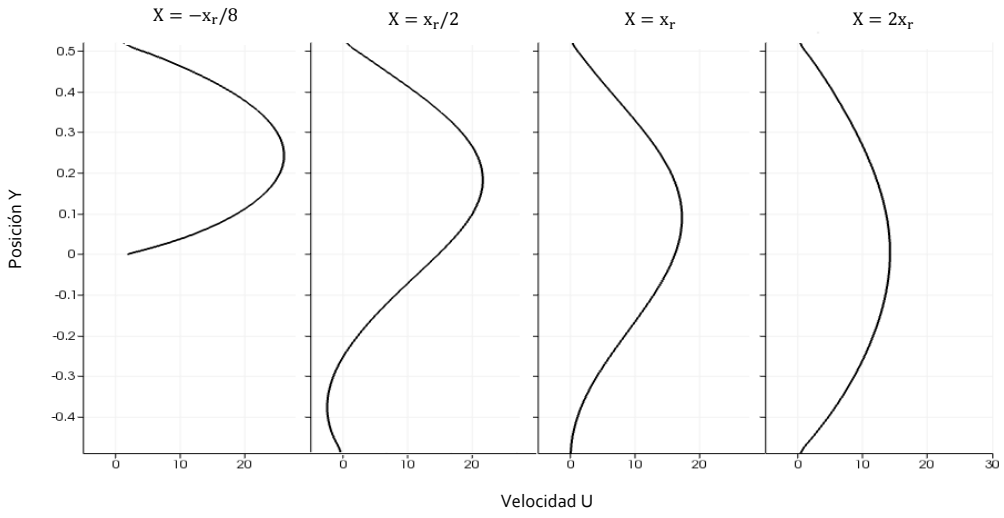


a)

Re_D	x_r/h
50	$2{,}95130 \pm 0{,}00487$
100	$1{,}78247 \pm 0{,}00487$

b)

Figura 34. Perfiles de velocidad en varias posiciones del canal para $Re_D = 100$.



7.4.3 Discusión

En la figura 32 se observan las líneas de corriente de flujo para los números de Reynolds de 100 y 50; como se mencionó anteriormente, la zona de recirculación

comienza en la esquina donde comienza el escalón y termina en el punto denominado como punto de reencuentro. Como es de esperar, la longitud de separación del flujo es mayor para el número de Reynolds mayor; esto se debe a que los gradientes son más altos.

La figura 32 no muestra el dominio completo simulado, únicamente la región cercana al escalón; la longitud del canal se tomó acorde con las condiciones de frontera explicadas en el capítulo 1, con el fin de lograr flujo completamente desarrollado en el canal aguas arriba y aguas abajo del escalón. En la figura 34 se muestran los perfiles longitudinales de velocidad en diferentes posiciones del canal, es posible ver que en la posición justo antes del escalón ($X = -x_r/8$) el flujo es completamente desarrollado (Perfil parabólico de velocidad), lo mismo sucede aguas abajo de la zona de recirculación ($X = 2x_r$).
Este comportamiento permite validar las condiciones de frontera de entrada y salida de flujo utilizadas en el esquema numérico.

La figura 34 también permite observar el comportamiento de la velocidad en la zona de separación ($X = x_r/2$), la componente de la velocidad en dirección x, u, se hace negativa, lo cual ocasiona la recirculación del flujo. En la posición del punto de reencuentro ($X = x_r$) ya no se presentan velocidades negativas, la componente x de la velocidad cercana a la pared del canal tiende a 0.

El estudio de Armaly et. al. [14] presenta una gráfica con los valores obtenidos de la posición de reencuentro en función del número de Reynolds (Fig. 33a); con el fin de obtener los valores de referencia requeridos para los números de Reynolds estudiados en este trabajo, se utilizó el programa analizador de gráficos “G3data”, este permite obtener de la gráfica los valores con un error de $\pm 0{,}00487$ en cada uno (Fig. 33b).

Con el propósito de asegurar la independencia de la malla de la solución numérica obtenida, se evalúan los dos casos del problema en tres densidades de malla cada uno; en la tabla 2 se presenta la posición del punto de reencuentro para cada una

de las mallas estudiadas y los errores relativos asociados al compararlos con los valores tomados como referencia (Fig. 33b).

El valor presentado $A_{mín}/h$ (Tabla 2) representa la densidad de malla utilizada en la región de recirculación de flujo (Longitud mínima de las caras de los triángulos de la malla), en el resto del dominio se utilizó una densidad de malla más baja.

Los errores obtenidos para las mallas más finas son aceptables, estas densidades de mallas serán las utilizadas para realizar el estudio del flujo en el canal con expansión inclinado.

8. RESULTADOS DEL ESTUDIO DEL FLUJO EN UN CANAL CON EXPANSIÓN INCLINADA

La descripción del problema del flujo en un canal con expansión inclinada se presentó en el capítulo 1, en este capítulo se presentarán los resultados obtenidos del estudio numérico.

La solución de este problema se realizó de la misma forma que el problema del flujo en el canal con expansión recta: utilizando el método FSM explicado en el capítulo anterior y un mallado no estructurado. La densidad de malla se tomó igual a la densidad más fina utilizada en ese problema: $A_{mín}/h = 0.03265$ en la zona de recirculación.

El objetivo es encontrar la posición del punto de separación y del punto de reencuentro del flujo, la longitud entre estos dos puntos es denominada longitud de separación.

El estudio se realizó para dos números de Reynolds que cumplen con la condición de flujo laminar, y pertenecen al rango en el que el flujo se considera bidimensional ($Re_D < 400$); la relación de expansión se tomó como $H/h = 1.9423$, y se estudiaron ángulos de inclinación de 15°, 30°, 45° y 90°. Adicionalmente se presentan resultados para diferentes relaciones de expansión, ángulo de inclinación de 45°y $Re_D = 50$.

8.1 Resultados numéricos

Tabla 3. Ubicación de los puntos de separación y reencuentro, y longitud de separación, para los ángulos estudiados.

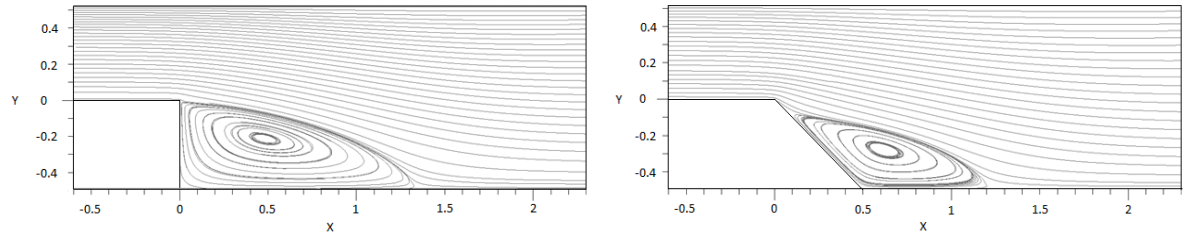
α	x_s	x_r	L_s
90°	0	1,36620	1,36620
45°	0,07250	1,21775	1,14525
30°	0,36270	1,14400	0,78130
15°	–	–	–

a) $Re_D = 100$

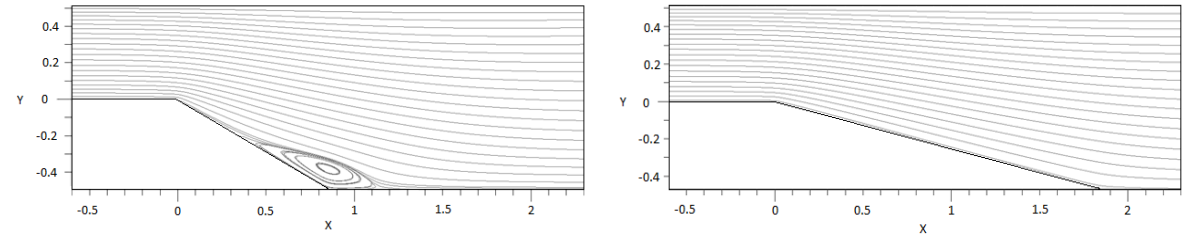
α	x_s	x_r	L_s
90°	0	0,82400	0,82400
45°	0,17260	0,72160	0,54900
30°	–	–	–
15°	–	–	–

b) $Re_D = 50$

Figura 35. Líneas de corriente en la región cercana al escalón para varios ángulos de inclinación y $Re_D = 100$.

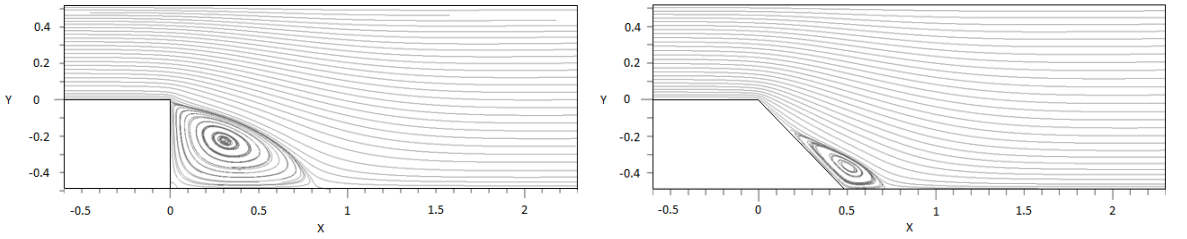


a) $\alpha = 90^{\circ}$ b) $\alpha = 45^{\circ}$

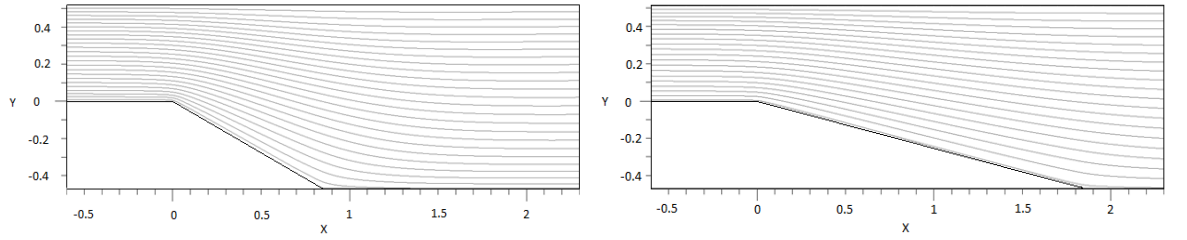


c) $\alpha = 30^{\circ}$ d) $\alpha = 15^{\circ}$

Figura 36. Líneas de corriente en la región cercana al escalón para varios ángulos de inclinación y $Re_D = 50$.



a) $\alpha = 90^{\circ}$ b) $\alpha = 45^{\circ}$

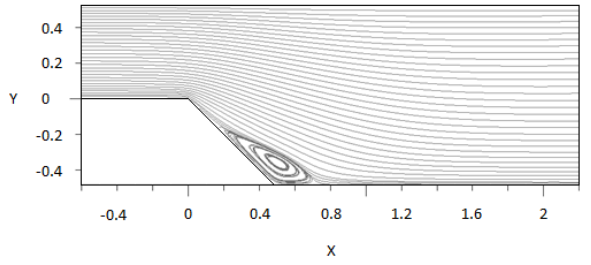


c) $\alpha = 30^{\circ}$ d) $\alpha = 15^{\circ}$

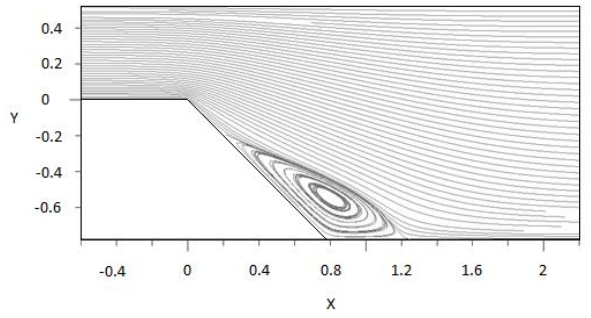
Tabla 4. Ubicación de los puntos de separación y reencuentro, y longitud de separación, para $\alpha = 45^{\circ}$ y tres relaciones de expansión.

H/h	x_s	x_r	L_s
1,9423	0,17260	0,72160	0,54900
2,5000	0,20200	1,21000	1,00800
3,0000	0,22640	1,62630	1,60366

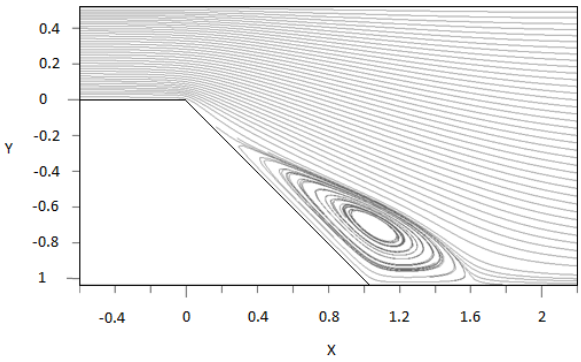
Figura 37. Líneas de corriente en la región cercana al escalón para diferentes relaciones de expansión, $\alpha = 45^{\circ}$ y $Re_D = 50$



a) $H/h = 1{,}9423$



b) $H/h = 2{,}5$



c) $H/h = 3{,}0$

8.2 Discusión

En las figuras 35 y 36 se observan las líneas de corriente del flujo para el canal con relación de expansión $H/h = 1{,}9423$ y los diferentes ángulos estudiados para $Re_D = 50$ y $Re_D = 100$, se observa como la zona de recirculación se hace más pequeña conforme disminuye el ángulo de inclinación del escalón, para algunos casos, incluso, llega a ser inexistente. La tabla 3 presenta la posición de los puntos de

separación y reencuentro y la longitud de separación en cada caso; los resultados muestran como la longitud de separación disminuye al disminuir el ángulo de inclinación; la separación del flujo se presenta cada vez más apartada del inicio del escalón conforme disminuye el ángulo de inclinación; para los casos de $Re_D = 100$ y ángulo de 15°, y $Re_D = 50$, y ángulos de 15° y 30°, no se presenta el fenómeno de separación de flujo. Este comportamiento coincide con el presentado para flujo laminar tridimensional en [17] y para flujo turbulento en [18].

Al igual que en el caso del escalón recto, la longitud de separación también disminuye al disminuir el número de Reynolds.

En la figura 37 se presentan las líneas de corriente del flujo par $Re_D = 50$, escalón con ángulo de inclinación de 45° y varias relaciones de expansión; la tabla 4 muestra las posiciones de los puntos de separación y reencuentro, y la longitud de separación para estos tres casos; la longitud de separación aumenta conforme aumenta la relación de expansión, la posición del punto de separación se aleja de la esquina donde empieza el escalón a medida que aumenta la relación, sin embargo, no es un aumento significativo con respecto al aumento en la longitud de separación. El aumento de la zona de recirculación conforme aumenta la relación de expansión coincide con los resultados encontrados en [16] para flujo laminar en un canal con escalón recto.

9. CONCLUSIONES

- Se desarrolló una herramienta computacional para la solución bidimensional de las ecuaciones incompresibles de Navier-Stokes para bajos números de Reynolds. Se utilizó el método de volúmenes finitos, y esquemas de primer orden y segundo orden para la discretización espacial y temporal, respectivamente.

- La comparación de los resultados obtenidos del problema Smith-Hutton, y el problema Driven Cavity, con las soluciones de referencia presentes en la literatura permitieron validar la herramienta desarrollada.

- El mallado no estructurado se adapta con efectividad a cualquier geometría; tanto en geometrías sencillas como dominios rectangulares (donde se suele usar malla estructurada), como en problemas con geometrías de mayor complejidad, se obtienen resultados satisfactorios. Sin embargo, aumenta el tiempo de cálculo de la simulación debido a los elementos geométricos y el término del segundo gradiente de difusión que introduce.

- El esquema de interpolación aguas arriba (Upwind) es de fácil implementación en el código computacional y garantiza resultados físicamente posibles; sin embargo es de bajo orden de precisión e introduce una falsa difusión, lo cual conlleva a errores significativos en la solución encontrada, principalmente cuando el problema de estudio se caracteriza por tener una gran influencia del término convectivo.

- Se aseguró la estabilidad de las soluciones utilizando el criterio de estabilidad CFL con valores del número de Courant convectivo y viscoso de 0,2 y 0,35, respectivamente.

- El método del paso fraccionado resulta ser de fácil implementación en el código computacional y permite reducir el problema de checkerboarding en mallas colocadas, lo cual lo hace una técnica conveniente al usar malla no estructurada.

- Se desarrolló un modelo matemático para la descripción del flujo laminar e incompresible que pasa por un canal que presenta una expansión inclinada en su geometría. El modelo fue validado comparando los resultados de la solución para el caso del problema con escalón recto con los resultados encontrados en la literatura.

- El problema del flujo en un canal con expansión inclinada tiene una geometría sencilla, sin embargo, tiene características del flujo complejas, como gradientes de presión adversos, recirculaciones de flujo, separación reencuentro y redesarrollo de la capa límite.

- La solución numérica del problema del flujo en un canal con expansión permite obtener resultados bastante aproximados de su comportamiento real estudiado experimentalmente, para el rango de números de Reynolds en que el flujo permanece bidimensional.

- El uso de un esquema de discretización temporal explícito para la solución del problema del flujo en un canal con expansión implica que se requieran grandes tiempos de cálculo debido a la restricción de tiempo que introduce, ya que, en este problema, se requieren mallas bastante finas.

10. RECOMENDACIONES

- En este trabajo se solucionó el problema del flujo laminar e incompresible en un canal con expansión; es posible continuar el desarrollo de este problema al incluir la solución en régimen turbulento utilizando modelos tipo LES (Large Eddy Simulation).

- Se podría mejorar la precisión del método numérico desarrollado utilizando un esquema de mayor orden para la discretización espacial de los términos convectivos de la ecuación de transporte.

- El código creado puede ser optimizado en cuanto al uso de memoria y el tiempo de procesamiento. También resultaría favorable aprovechar el uso de la programación en paralelo que permite PETSc.

- El código desarrollado solo admite mallas con elementos triangulares, sin embargo, es posible y no resulta complicado implementar los elementos necesarios para poder ingresar mallas con otras formas geométricas. La implementación de elementos cuadráticos permitiría el uso de mallas estructuradas (bidimensionales).

BIBLIOGRAFÍA

1. ARMALY, B. F. et al. Experimental and theoretical investigation of backward-facing step flow. En: Journal of Fluids Mechanics. 1983, vol.127.

2. CHORIN, A. J. Numerical Solution of the Navier-Stokes Equations. En: Journal of Computational Physics. 1968, vol. 22, pag. 745-62.

3. CIMEC, Centro Internacional de Métodos Computacionales en Ingeniería. Métodos iterativos para la solución de problemas lineales y no-lineales. (http://www.cimec.org.ar/) Consultado: Diciembre, 2013.

4. CTTC. Introduction to the fractional step method. Universidad Politécnica de Catalunya.

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

6. GHIA, U. et. al. High-resolutions for icompressible flow using the Navier-Stokes equations and a multigrid method. En: Journal of Computational Physics. 1982.

7. MURTHY, J. Y. Numerical Methods in Heat, Mass, and Momentum Transfer. J.Y. School of Mechanical Engineering,Purdue University,1998.

8. PATANKAR, S. V. Numerical Heat Transfer and Fluid Flow. New York: Hemispher Publishing Corporation, 1980.

9. PEREZ-SEGARRA C.D. et. al. Analysis of different numerical schemes for the resolution of convection-diffusion equations using finite-volume methods on three-dimensional unstructured. En: Numerical Heat Transfer. 2006, parte B, vol. 49.

10. SMITH. R. M. and HUTTON A. G. The numerical treatment of advection: a performance comparison of current methods. Numerical Heat Transfer, 5:439–461, 1982.

REFERENCIAS

[1] PATANKAR, S. V. Numerical Heat Transfer and Fluid Flow. New York: Hemispher Publishing Corporation, 1980.

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

[3] MURTHY, J. Y. Numerical Methods in Heat, Mass, and Momentum Transfer. J.Y. School of Mechanical Engineering,Purdue University,1998.

[4] BELEÑO MIER, A. A. et al. Seminario de investigación en dinámica de fluidos computacional. Trabajo de grado ingeniero mecánico. Bucaramanga: Universidad Industrial de Santander. Facultad de ingenierías físico mecánicas. 2009.

[5] JEREZ CARRIZALES, Manuel Fernando. Modelado y simulación del flujo de un fluido sobre una placa mediante volúmenes finitos. Trabajo de grado ingeniero mecánico. Bucaramanga: Universidad Industrial de Santander. Facultad de ingenierías físico mecánicas. 2012.

[6] PEREZ-SEGARRA C.D. et. al. Analysis of different numerical schemes for the resolution of convection-diffusion equations using finite-volume methods on three-dimensional unstructured. En: Numerical Heat Transfer. 2006, parte B, vol. 49, pag. 333-350.

[7] CASCAVITA MELLADO, K. L. et. al. Solución numérica de las ecuaciones de Navier-Stokes incompresibles por el método de los volúmenes finitos. En: Revista Ion. 2013, vol. 26, no. 2.

[8] CHORIN, A. J. Numerical Solution of the Navier-Stokes Equations. En: Journal of Computational Physics. 1968, vol. 22, pag. 745-62.

[9] KIM, J. and MOIN, P. Application of a fractional-step method to incompressible Navier-Stokes equations. En: Journal of Computational Physics. 1985, vol. 59, pag. 308–323.

[10] BOIVIN, S. et. al. A finite volume method to solve the Navier–Stokes equations for incompressible flows on unstructured meshes. En: International Journal of Thermal Sciences. 2000, vol. 39, pag. 806–25.

[11] CTTC. Introduction to the fractional step method. Universidad Politécnica de Catalunya.

[12] SMITH. R. M. and HUTTON A. G. The numerical treatment of advection: a performance comparison of current methods. Numerical Heat Transfer, 5:439–461, 1982.

[13] GHIA, U. et. al. High-resolutions for icompressible flow using the Navier-Stokes equations and a multigrid method. En: Journal of Computational Physics. 1982, vol. 48, pag. 387-411.

[14] ARMALY, B. F. et al. Experimental and theoretical investigation of backward-facing step flow. En: Journal of Fluids Mechanics. 1983, vol.127.

[15] NOH, F. et al. Solución numérica para el flujo laminar en un canal con expansión brusca. En: Revista Sociedad mexicana de ingeniería mecánica. Septiembre, 2004, vol. 1 no. 5.

[16] BISWAS, G. et al. Backward-facing step flows for various expansion ratios at low and moderate reynolds numbers. En: Journal of fluids engineering. Mayo, 2004, vol. 123.

[17] CHEN, Y. T. et al. Three-dimensional convection flow adjacent to inclined backward-facing step. En: International journal of heat and mass transfer. Agosto, 2006.

[18] RUCK B. and MAKIOLA B. Flow separation over the inclined step. Institut für Hidromechanik, Universität Karlsruhe. 1993.

[19] GANDJALIKHAN NASSAB S. A. et. al. Study of laminar forced convection of radiating gas over an inclined backward facing step under bleeding condition using the blocked-off method. En: Journal of Heat Transfer. Julio de 2011, vol. 133.

[20] GANDJALIKHAN NASSAB S. A. et. al. Numerical investigation of entropy generation in laminar forced convection flow over inclined backward and forward acing steps in a duct under bleeding condition. Mechanical Engineering Department, School of Engineering, Shahid Bahonar University, Kerman, Iran.

[21] SALOME, The open source integration platform for numerical simulation. [2014] (http://www.salome-platform.org/) Consultado: Mayo, 2014.

[22] PETSc, Portable, Extensible Toolkit for Scientific Computation. [2014] (http://www.mcs.anl.gov/petsc/) Consultado: Mayo, 2014.

[23] ParaView. [2013] (http://www.paraview.org/) Consultado: Septiembre, 2013.

[24] CIMEC, Centro Internacional de Métodos Computacionales en Ingeniería. Métodos iterativos para la solución de problemas lineales y no-lineales. (http://www.cimec.org.ar/) Consultado: Diciembre, 2013.

ANEXO A. MÉTODOS DE SOLUCIÓN DE SISTEMAS DE ECUACIONES LINEALES

A continuación se dará una breve explicación de los métodos de solución de ecuaciones lineales que combinan un método de espacio Krylov y un precondicionador; se describirá brevemente el método de solución utilizado en este trabajo.

Los métodos Krylov son actualmente los algoritmos más exitosos en la solución de ecuaciones lineales, funcionan mediante la formación de una serie de vectores lineales independientes a partir del producto de la secuencia de potencias de la matriz por el vector de producto inicial. Las aproximaciones a la solución se forman minimizando el residuo en el subespacio formado.

$$K_r(A, b) = span\{b, Ab, A^2b, \dots, A^{r-1}b\}$$

La ecuación anterior corresponde a un subespacio Krylov de orden r, generado por una matriz cuadrada A de orden n, y en vector b de tamaño n.
Algunos de los métodos Krylov más utilizados son: método del gradiente conjugado (CG), método del residuo mínimo generalizado (GMRES), y método del gradiente biconjugado (BICG).

La convergencia de los métodos basados en los subespacios de Krylov mejora con el uso de las técnicas de precondicionamiento. Éstas consisten generalmente en cambiar el sistema original $Ax = b$ por otro de idéntica solución, de forma que el número de condicionamiento de la matriz del nuevo sistema sea menor que el de A, o que tenga una mejor distribución de sus valores.
Generalmente, se considera como matriz de precondicionamiento a M^{-1}, de modo que: $M^{-1}Ax = M^{-1}b$.

Los precondicionadores deben de cumplir dos requisitos: fácil implementación, evitando un costo computacional excesivo del producto de M^{-1} por un vector; y mejorar la convergencia del método.

El campo de investigación de los precondicionadores es muy extenso. Algunos de los más usados y cuyas prestaciones son bien conocidas son: Factorizaciones ILU, diagonal o de Jacobi, el método SOR, y los métodos Multigrid.

El método de solución del sistema de ecuaciones utilzado en este trabajo, consiste en la combinación del método del residuo mínimo generalizado (GMRES) y un precondicionador Multigrid algebraico. A continuación se hace una breve descripción de en que consisten.

MÉTODO DEL RESIDUO MÍNIMO GENERALIZADO (GMRES)

Fue propuesto en 1986 por Y. Saad y M. Schulz como un método iterativo por subespacios de Krylov para sistemas no-simétricos y no necesariamente definidos positivos, no requiere el cálculo de productos de A^T con un vector, lo cual es una gran ventaja en muchos casos.

El desarrollo del algoritmo consiste en encontrar un vector $x = x_0 + V_y$, imponiendo la condición de mínimo para el residuo de la ecuación: $J(y) = \|b - Ax\|$.

Este algoritmo requiere de un elevado costo computacional y de almacenamiento en memoria, para solucionar esto se usa una técnica de reinicio y truncamiento que consiste en que se realizan n iteraciones y se comienza de nuevo a partir de la última iteración, de modo que no se exceda la memoria de la máquina.

MÉTODO DE PRECONDICIONAMIENTO MULTIGRID

La principal idea del método Multigrid es acelerar la convergencia del método iterativo mediante correcciones hechas en cada iteración, el principio es similar a calcular una solución en malla basta e interpolarla a una malla más fina.

El método multigrid algebraico construye los niveles de interpolación basándose en la matriz del sistema, se asocian las ecuaciones de un grupo de volúmenes de control y a partir de ellas se obtiene una ecuación que corresponde a una malla más basta. Así, los coeficientes de la matriz del nivel basto se obtienen sumando los coeficientes de la matriz de nivel más fino.

ANEXO B. USO DE PETSC

PETSc (Portable, Extensible Toolkit for Scientific Computation) es una serie de estructuras de datos y rutinas para la implementación de aplicaciones de código computacional en paralelo (o serie si se requiere); utiliza el estándar MPI para la comunicación de paso de mensajes.

La programación en paralelo permite mejorar la velocidad en la solución de grandes problemas al permitir que muchas instrucciones se realicen simultáneamente. El comunicador MPI es una forma de indicar una colección de procesos que se involucraran en el cálculo; PETSc da la opción de especificar si el programa o una rutina se realiza en paralelo o secuencial, o si se prefiere no se especifica y Petsc es quién decide según sea más conveniente.

Es posible referirse al manual de PETSc para conocer cómo funcionan todas las rutinas y comandos de forma detallada, así como la explicación de la descarga e instalación del mismo. A continuación se dará explicación de las rutinas utilizadas en el código computacional creado en este trabajo, y se mostrarán ejemplos de su uso para un mayor entendimiento del mismo.

1. PROGRAMANDO CON PETSC

Para poder utilizar todas las rutinas de PETSc, es necesario incluir las librerías mediante la declaración:

```
#inlcude "petsc.h"
```

Todos los programas deben empezar con el comando:

```
PetscInitialize(int *argc,char ***argv,char *file,char *help);
```

Que inicializa PETSc y MPI. Los argumentos `argc` y `argv` son los argumentos de línea de comandos dados en todos los programas de C y C++ (Función main); el

argumento `file` indica un nombre alternativo para el archivo de opciones de Petsc, y `help` corresponde a un comando de ayuda para el programa.

Al finalizar el programa se debe llamar el comando:

```
PetscFinalize();
```

2. DECLARACIÓN DE VARIABLES

Aparte de los tipos de variables usadas en C++, Petsc introduce otros tipos de variables, algunos de ellos son:

PetscInt: Tipo de variable que representa un entero, se usa principalmente para representar dimensiones e indexación.
PetscScalar: Tipo de variable que representa una número real o complejo de precisión *double*, un número real de precisión *float*, un *long double*, o un entero.
PetscReal: Tipo de variable que representa una versión real de PetscScalar.

Por otro lado, todas las rutinas de Petsc devuelven un entero que indica si un error ha ocurrido durante la ejecución del programa, esta variable se declara como:

```
PetscErrorCode ierr;
```

Para poder rastrear cualquier error es necesario que cada rutina se escriba de la siguiente forma:

```
ierr = rutina de petsc; CHKERRQ(ierr);
```

De esta forma se detectan los errores y se muestran en la terminal al ejecutar el programa.

3. USO DE VECTORES

Los vectores se declaran como objetos así: `Vec v`. Petsc provee dos tipos de vector: secuencial y paralelo (basado en MPI); al crear el vector es posible especificar a

qué tipo corresponde, o pueden utilizarse las siguientes rutinas que generan automáticamente el tipo de vector apropiado según el caso:

```
VecCreate(MPI Comm comm,Vec *v);
VecSetSizes(Vec v, int m, int M);
VecSetFromOptions(Vec v);
```

La variable `comm` declarada como `MPI Comm` es el comunicador que corresponde a un grupo de procesos sobre el que se realiza la comunicación. M es el número de componentes totales del vector, y m indica el número de componentes que se guardarán en el proceso local, es posible dejar que PETSc sea quien determine este valor mediante el argumento `PETSC_DECIDE`.

Para el llenado del vector, es posible asignar un valor singular a todos los componentes del vector con el comando:

```
VecSet(Vec v,PetscScalar valor);
```

Para asignar valores individuales a los componentes se utiliza el comando:

```
VecSetValues(Vec v,int n,int *indices,PetscScalar
*values,INSERT VALUES);
```

El argumento n establece el número de componentes a los que se asignará un valor, el array de tipo entero contiene los índices de los componentes, y el array de tipo PetscScalar contiene los valores que se van a insertar.

Una vez se hayan ingresado todos los valores del vector, es necesario llamar los siguientes comandos para ensamblar el vector:

```
VecAssemblyBegin(Vec v);
VecAssemblyEnd(Vec v);
```

Para acceder a los elementos del vector se utiliza el siguiente comando, el cual permite obtener un puntero con los valores del vector.

```
VecGetArray(Vec v,PetscScalar **array);
```

Cuando ya no se necesiten los valores, es necesario devolverlos mediante el comando:

```
VecRestoreArray(Vec v, PetscScalar **array);
```

Finalmente cuando un vector ya no se necesite debe ser destruido con el comando:

```
VecDestroy(Vec *v);
```

EJEMPLO

A continuación se presenta un ejemplo tomado del código computacional desarrollado en este trabajo para la aplicación del uso de vectores.

El ejemplo muestra cómo se crea y se llena un vector (`Vec vec_ap0`) para guardar los valores de un coeficiente en cada celda (el número de celdas se define anteriormente en el código como `int n_vc`). Para calcular el valor del coeficiente, se necesita el valor del volumen en cada celda, el cual se tiene guardado en un vector (`Vec vec_volumen`) que se creó y llenó anteriormente en el código. La variable para obtener los valores de volumen se declara como `PetscScalar *volumenvc`.

```
...
VecCreate(PETSC_COMM_WORLD,&vec_ap0);
VecSetSizes(vec_ap0,PETSC_DECIDE,n_vc);
VecSetFromOptions(vec_ap0);

VecGetArray(vec_volumen,&volumenvc);
for (i=0;i<n_vc;i++)
{
     ap_0=densidad*volumenvc[i]/timestep;
     VecSetValues(vec_ap0,1,&i,&ap_0,INSERT_VALUES);
}
VecRestoreArray(vec_volumen,&volumenvc);
```

```
ierr = VecAssemblyBegin(vec_ap0);
ierr = VecAssemblyEnd(vec_ap0);
...
```

4. USO DE MATRICES

Las matrices se declaran como objetos así: `Mat A`. Petsc provee una variedad de tipos de matrices: matrices densas y dispersas, ambas en versiones secuenciales o paralelas), también tiene otros formatos especializados.

La rutina más sencilla para crear un matriz es la siguiente:

```
MatCreate(MPI Comm comm,Mat *A);
MatSetSizes(Mat A,int m,int n,int M,int N);
```

Esta rutina genera una matriz secuencial si se corre un proceso, o una matriz paralela si se corren dos o más procesos. Los argumentos M y N designan el tamaño global de la matriz; m y n designan el tamaño local, estos últimos pueden establecerse como `PETSC_DECIDE`, de modo que sea PETSc quien determine este valor.

Para ingresar valores a una matriz se utiliza el comando:

```
MatSetValues(Mat A,int m,const int idxm[],int n,const int
idxn[],const PetscScalar values[],INSERT VALUES);
```

Esta rutina inserta un bloque de valores de dimensión m*n a la matriz, Los índices `idxm` y `idxn`, indican los números de fila y columna donde se ingresaran los valores, respectivamente. El argumento `values[]` es un array bidimensional que contiene los valores a ingresar a la matriz.

Luego de que se han ingresado los elementos a la matriz se deben ensamblar antes de poder ser usados, esto se hace mediante las rutinas:

```
MatAssemblyBegin(Mat A,MAT FINAL ASSEMBLY);
MatAssemblyEnd(Mat A,MAT FINAL ASSEMBLY);
```

Una vez la matriz es ensamblada, se compacta y puede ser utilizada en operaciones; las posiciones que no se hallan llenado con ningún valor se pierden al comprimirse la matriz, si estas posiciones serán usadas posteriormente, se deben insertar ceros en ellas antes de utilizar las rutinas mencionadas.

Para acceder a los elementos de la matriz se utiliza el comando:

```
MatGetRow(Mat A,int row, int *ncols,const PetscInt (*cols)[],const PetscScalar (*vals)[]);
```

El argumento `ncols` devuelve el número de elementos con valor diferente de cero en la fila `row`, mientras `cols` y `vals` devuelven los índices de las columnas y los valores de la fila. Si uno de estos elementos no se requiere es posible utilizar `PETSC_NULL` como argumento en lugar del otro. Los valores extraídos de la matriz con este comando no pueden ser modificados.

Una vez se haya finalizado el uso de una fila es necesario liberar el espacio de memoria mediante el siguiente comando:

```
MatRestoreRow(Mat A,int row,int *ncols,int **cols,PetscScalar **vals);
```

Finalmente cuando una matriz ya no se necesite debe ser destruida con el comando:

```
MatDestroy(Mat *A);
```

EJEMPLO

A continuación se presenta un ejemplo tomado del código computacional desarrollado en este trabajo para la aplicación del uso de matrices.

El ejemplo muestra cómo se crea y se llena una matriz (`Mat mat_coord`) para guardar los valores de las coordenadas del centroide de cada celda (el número de

celdas se define anteriormente en el código como `int vc`). Para calcular el valor del centroide se necesitan los índices de los vértices que pertenecen a la celda, y sus coordenadas; los índices de los vértices se obtuvieron anteriormente y se encuentran en la matriz `Mat matriz_vc,` sus valores se obtienen mediante la variable `const PetscScalar *vertvc`; las coordenadas de los vértices también se obtuvieron anteriormente y se encuentran en la matriz `Mat matriz_n`, sus valores se obtienen mediante la variable `const PetscScalar *coordvert`.

```
...
MatCreate(PETSC_COMM_WORLD,&matriz_coord);
MatSetSizes(matriz_coord,PETSC_DECIDE,PETSC_DECIDE,vc,2);
MatSetFromOptions(matriz_coord);
MatSetUp(matriz_coord);
...
for (j=0;j<vc;j++)
{
   MatGetRow(matriz_vc,j,PETSC_NULL,PETSC_NULL,&vertvc);
   for (i=0;i<3;i++)
   {
       k=vertvc[i];
       MatGetRow(matriz_n,k,PETSC_NULL,PETSC_NULL,&coordvert);
       vertx[i]=coordvert[0];
       verty[i]=coordvert[1];
       MatRestoreRow(matriz_n,k,PETSC_NULL,PETSC_NULL,&coordvert);
   }
   value[0]=(vertx[0]+vertx[1]+vertx[2])/3;
   value[1]=(verty[0]+verty[1]+verty[2])/3;
   MatSetValues(matriz_coord,1,&j,2,col,value,INSERT_VALUES);
   ...
   MatRestoreRow(matriz_vc,j,PETSC_NULL,PETSC_NULL,&vertvc);
}
MatAssemblyBegin(matriz_coord,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(matriz_coord,MAT_FINAL_ASSEMBLY);
```

...

5. SOLUCIÓN DE ECUACIONES LINEALES

El objeto `KSP` permite el acceso a todos los paquetes de solucionadores de sistemas lineales que provee PETSc; el sistema de ecuaciones a resolver debe ser de la forma:

$$Ax = b$$

Donde A denota una matriz que representa un operador lineal, b es el vector del lado derecho de la ecuación y x es el vector solución. KSP utiliza las mismas secuencias para solucionadores directos e iterativos.

Para solucionar un sistema de ecuaciones lineales con KSP primero se debe crear la interface con el comando:

```
KSPCreate(MPI Comm comm,KSP *ksp);
```

Además se deben especificar las matrices asociadas con el sistema de ecuaciones:

```
KSPSetOperators(KSP ksp,Mat Amat,Mat Pmat,MatStructure flag);
```

Los argumentos `Amat` y `Pmat` son el mismo valor y representan la matriz que define el sistema lineal. El argumento `flag` puede ser usado para eliminar trabajo innecesario cuando se solucionan repetidamente sistemas lineales del mismo tamaño con el mismo precondicionador, cuando se resuelve solo un sistema de ecuaciones se puede ignorar este argumento; en el caso de no conocer la estructura de la matriz antes de solucionar el sistema, se debe designar el argumento `flag` como `DIFFERENT_NONZERO_PATTERN`.

Para la solución de sistemas de ecuaciones lineales Petsc permite la combinación de un método de sub-espacio Krylov y un precondicionador.

En la siguiente tabla se muestran los métodos de subespacio Krylov que ofrece PETSc con el nombre de su respectivo comando de llamado.

Método	*KSPType*
Richardson	KSPRICHARDSON
Chebyshev	KSPCHEBYSHEV
Gradiente Conjugado	KSPCG
Gradiente Biconjugado	KSPBICG
Mínimos Residuos Generalizado	KSPGMRES
Mínimos Residuos Generalizado Flexible	KSPFGMRES
Mínimos Residuos Generalizado Reducido	KSPDGMRES
Residuos Conjugados Generalizado	KSPGCR
Gradiente Biconjugado STAB	KSPBCGS
Gradiente Conjugado Cuadrado	KSPCGS
Transpuesta Libre de Residuo Cuasi-Mínimo (1)	KSPTFQMR
Transpuesta Libre de Residuo Cuasi-Mínimo (2)	KSPTCQMR
Residuo Conjugado	KSPCR
Mínimos Cuadrados	KSPLSQR
Opción sin método KSP	KSPPREONLY

Para definir el método a utilizar se utiliza el comando:

```
KSPSetType(KSP ksp,KSPType method);
```

El argumento `method` puede ser cualquiera de los listados en la tabla anterior. Es posible definir opciones específicas según el método que se escoja, sin embargo, ese tema no es de relevancia aquí, luego no será abordado.

En la siguiente tabla se muestran las opciones de precondicionadores disponibles en PETSc; en esta se encuentran divididos, los primeros corresponden a aquellos que pueden ser usados sin subespacio Krylov (KSPPREONLY), para los otros es necesario definir un método de subspacio Krylov.

Método	*PCType*
Jacobi	PCJACOBI
Jacobi en Bloque	PCBJACOBI
SOR (y SSOR)	PCSOR
SOR con el truco Eisenstat	PCEISENSTAT
Cholesky Incompleta	PCICC
LU Incompleta	PCILU
Schwarz Aditivo	PCASM
Multigrid algebraico	PCGAMG
Método de Solución Lineal	PCKSP
Combinación de Precondicionadores	PCCOMPOSITE
LU	PCLU
Cholesky	PCCHOLESKY
Sin Precondicionador	PCNONE
Opción para que el usuario defina el precondicionador	PCSHELL

Para definir el precondicionador a usar se utilizan las siguientes rutinas:

```
KSPGetPC(KSP ksp,PC *pc);
PCSetType(PC pc,PCType type);
```

El argumento `type` puede ser cualquiera de los listados en la tabla anterior.

Una vez definido el método y el precondicionador se deben utilizar las siguientes rutinas para realizar cualquier configuración necesaria del solucionador:

```
KSPSetFromOptions(KSP ksp);
KSPSetUp(KSP ksp);
```

Con todo lo anterior ya es posible solucionar el sistema de ecuaciones mediante el comando:

```
KSPSolve(KSP ksp,Vec b,Vec x);
```

Donde el argumento `b` corresponde al vector del lado derecho de la ecuación, y `x` es el vector solución.

Finalmente cuando el espacio KSP ya no se necesite debe ser destruido con el comando:

```
KSPDestroy(KSP *ksp);
```

ANEXO C. CÓDIGO COMPUTACIONAL DESARROLLADO

1. ARCHIVO DE LA FUNCIÓN MAIN

```
1  //ARCHIVO DE LA FUNCIÓN MAIN: programa.cpp
2
3  #include <petsc.h>
4  #include "geom.h"
5  #include "frontera.h"
6  #include "solucion.h"
7  #include "soltransitoria.h"
8  #include "solimplicito.h"
9  #include "fractional.h"
10
11 using namespace std;
12
13 #undef __FUNCT__
14 #define __FUNCT__ "main"
15
16 int main(int argc,char **args)
17 {
18 /* ---------------------------- DATOS PROPORCIONADOS POR EL PROBLEMA ----------------------------*/
19
20     //Propiedades
21     double densidad=1;
22     double gamma=10;
23
24     //Tiempo
25     double tiempo=2;
26
27     //Geometría: 1-->dominio rectangular   2-->canal con expanción   3-->Smith-Hutton
28     int geometría=1;
29
30     //Archivo de la malla
31     char malla[100]="mallabfs_44000.dat";
32
33     //Problema a resolver:  1-->Ec. de D.C. en estado estable   2-->Ec. de D.C. en estado
transitorio(explícito)   3--->Ec. de D.C. en estado transitorio (ímplicito)   4-->Ec. de flujo
5-->Smith-Hutton
34     int problema=4;
35
36     //Velocidad conocida constante en todo el dominio. Ec. de D.C.
37     double velocidad[0]=0;
38     double velocidad[1]=0;
39
40     //Phi inicial constante en todo el dominio
41     double tempinicial=100;
42
43     //Variables
44     double condicion[4];
45     double valor[4];
46
47     PetscInitialize(&argc,&args,(char*)0,help);
48
49     /*---------------------------- GEOMETRÍA DEL DOMINIO ----------------------------*/
50     geom gm;
51     gm.leerdatos(malla);
52     gm.calcular();
53
```

```
54      /*--------------------------- CONDICIONES DE FRONTERA ---------------------------*/
55
56      //Condición Dirichlet=1, Condición Neumann=-1
57
58      //Frontera para phi en Ec. de D.C.
59      condicion[0]=1;
60      valor[0]=1;
61      condicion[1]=1;
62      valor[1]=0;
63      condicion[2]=1;
64      valor[2]=0;
65      condicion[3]=1;
66      valor[3]=0;
67
68      if(problema==1)
69      {
70          if(geometria==1)
71          {
72              frontera ft;
73              ft.leerR(a,condicion,valor);
74          }
75      }
76
77      //Frontera para la velocidad u en problema de flujo
78      condicion[0]=1;
79      valor[0]=1;
80      condicion[1]=1;
81      valor[1]=0;
82      condicion[2]=1;
83      valor[2]=0;
84      condicion[3]=1;
85      valor[3]=0;
86
87      if(problema==2)
88      {
89          if(geometria==1)
90          {
91              frontera fu;
92              fu.leerR(a,condicion,valor);
93          }
94          if(geometria==2)
95          {
96              frontera fu;
97              fu.leerBFS(a,condicion,valor);
98          }
99      }
100
101     //Frontera para la velocidad v en problema de flujo
102     valor[0]=0;
103     valor[1]=0;
104     valor[2]=0;
105     valor[3]=0;
106
107     if(problema==2)
108     {
109         if(geometria==1)
110         {
111             frontera fu;
112             fu.leerR(a,condicion,valor);
113         }
114         if(geometria==2)
116             frontera fu;
117             fu.leerBFS(a,condicion,valor);
118         }
119     }
120
```

```
121     //Frontera para la presion
122     condicion[0]=-1;
123     valor[0]=0;
124     condicion[1]=-1;
125     valor[1]=0;
126     condicion[2]=-1;
127     valor[2]=0;
128     condicion[3]=-1;
129     valor[3]=0;
130
131     if(problema==2)
132     {
133         if(geometria==1)
134         {
135             frontera fu;
136             fu.leerR(a,condicion,valor);
137         }
138         if(geometria==2)
139         {
140             frontera fu;
141             fu.leerBFS(a,condicion,valor);
142         }
143     }
144
145     //Frontera para el Smith-Hutton
146     if(geometria==3)
147     {
148         frontera f;
149         f.leerSH(a);
150     }
151
152
153     /*---------------------------------- Solución ----------------------------------*/
154
155     //Solución de la Ec. de D.C. en estado estable.
156     if(problema==1)
157     {
158         solucion se;
159         se.calcular_Se(gamma,densidad,gm,ft,velocidad);
160     }
161
162     //Solución a la Ec. de D.C. en estado transitorio. Solución explícita.
163     if(problema==2)
164     {
165         soltransitoria ste;
166         ste.calcular_Ste(gm,ft,densidad,gamma,velocidad,tiempo,tempinicial);
167     }
168
169     //Solución a la Ec. de D.C. en estado transitorio. Solución ímplicita.
170     if(problema==3)
171     {
172         solimplicito sti;
173         sti.calcular_Sti(gm,ft,densidad,gamma,velocidad,tiempo,tempinicial);
174     }
175
176     //Solución del problema de flujo. Fractional Step.
177     if (problema==4)
178     {
179         fractional sfs;
180         sfs.calcular_SF(gm,fu,fv,fp,densidad,gamma,tiempo);
181     }
182
183     //Solución del problema Smith-Hutton
184     if(problema==5)
185     {
186         solucion s;
```

```
187         s.calcular_S(a,f,densidad,gamma);
188     }
189
190     PetscFinalize();
191 }
```

2. ARCHIVO DE ENCABEZADO DE LA CLASE GEOM

```
1  //ARCHIVO DE ENCABEZADO DE LA CLASE GEOM: geom.h
2
3  #ifndef _GEOM_H_
4  #define _GEOM_H_
5
6  class geom
7  {
8      private:
9
10         PetscErrorCode ierr;                //Variable error de petsc
11         int i,j,k,jj[1];                    //Contadores
12         int tem,tempor,tempora,temporal;    //Variables de uso temporal
13
14         //Variables de cálculo
15         PetscInt colareas[2];
16         PetscScalar value[4];
17         PetscInt nodvec[4];
18         double nosirve;
19         int elementos;
20         int lineas;
21
22         //número de vértices
23         int vertices;
24         //número de volúmenes de control
25         int vc;
26         //Matriz de coordenadas de cada vértice
27         Mat matriz_n;
28         const PetscScalar *vertvc;
29         const PetscScalar *vertactual;
30         const PetscScalar *vertcompar;
31         const PetscScalar *vertvecino;
32        //matriz de vértices que pertenecen a cada volumen de control
33         Mat matriz_vc;
34         const PetscScalar *vcactual;
35         const PetscScalar *vcvecino;
36         //matriz de coordenadas de los centroides de cada volumen de control
37         Mat matriz_coord;
38         const PetscScalar *coordvert;
39         PetscScalar vertx[3],verty[3];
40         //Vector de volúmenes
41         Vec vec_volumen;
42         PetscScalar volumen;
43         PetscScalar ladoa,ladob,ladoc;
44         PetscScalar per;
45         //matriz de vecinos
46         Mat matriz_vecinos;
47         const PetscScalar *vecinos;
48         const PetscScalar *vvecinos;
49         //matriz de areas
50         Mat matriz_areas;
51         PetscScalar area[2];
52         PetscScalar n_x[2],n_y[2];
53         PetscScalar pendiente,bcorte;
54         //matriz del vector de dirección PF
```

```
55          Mat matriz_vecpf;
56          PetscScalar pf[2];
57          //matriz del vector de dirección Pf
58          Mat matriz_vecpcent;
59          PetscScalar pcent[2];
60          //matriz de coordenadas de los centroides de cada cara
61          Mat matriz_centrocara;
62          PetscScalar centrocara[2];
63          //matriz de alfaf
64          Mat matriz_alfaf;
65          const PetscScalar *alfaf;
66          PetscScalar alfa[1];
67          //matriz de vectores p-p_prima
68          Mat matriz_vecppprima;
69          const PetscScalar *ppprima;
70          const PetscScalar *ffprima;
71          PetscScalar pp_prima[2];
72          PetscScalar m_a,m_p,b_a,b_p,p_prima[2];
73
74      public:
75
76          //Funciones
77          PetscErrorCode leerdatos(char *malla);
78          PetscErrorCode calcular(void);
79
80  };
81
82  #endif // _GEOM_H_
```

3. ARCHIVO DE CÓDIGO FUENTE DE LA CLASE GEOM

```
1   //ARCHIVO DE CÓDIGO FUENTE DE LA CLASE GEOM: geom.cpp
2
3   #include <petsc.h>
4   #include <fstream>
5   #include <cmath>
6
7   /************* FUNCION: LEER LOS DATOS DE LA MALLA DEL ARCHIVO DE TEXTO *************/
8
9   PetscErrorCode geom::leerdatos(char *malla)
10  {
11      /* ------------- LEER Y GUARDAR LOS VALORES DEL No. DE VERTICES --------------- */
12      /* -------------- No. DE ELEMENTOS TOTALES, No. DE ELEMENTOS 1D --------------- */
13
14      std::ifstream f(malla, std::ifstream::in);
15
16      f >> vertices >> elementos >> lineas;
17      vc=elementos-lineas;
18
19      /* ----------------------  INICIALIZACIÓN DE VARIABLES ----------------------- */
20
21      PetscInt col[vc];
22      for (i=0;i<vc;i++)
23      {
24          col[i]=i;
25      }
26
27      //Matriz de coordenadas de vértices
28      ierr = MatCreate(PETSC_COMM_WORLD,&matriz_n);CHKERRQ(ierr);
29      ierr = MatSetSizes(matriz_n,PETSC_DECIDE,PETSC_DECIDE,vertices+1,2);CHKERRQ(ierr);
30      ierr = MatSetFromOptions(matriz_n);CHKERRQ(ierr);
31      ierr = MatSetUp(matriz_n);CHKERRQ(ierr);
32
```

```
33      //Matriz de vértices que pertenecen a los vc
34      ierr = MatCreate(PETSC_COMM_WORLD,&matriz_vc);CHKERRQ(ierr);
35      ierr = MatSetSizes(matriz_vc,PETSC_DECIDE,PETSC_DECIDE,vc,3);CHKERRQ(ierr);
36      ierr = MatSetFromOptions(matriz_vc);CHKERRQ(ierr);
37      ierr = MatSetUp(matriz_vc);CHKERRQ(ierr);
38
39      /* --------------- LEER Y GUARDAR LAS COORDENADAS DE LOS VERTICES ---------------*/
40
41      for (i=1; i<vertices+1; i++)
42      {
43          f >> nosirve >> value[0] >> value[1] >> nosirve;
44          ierr = MatSetValues(matriz_n,1,&i,2,col,value,INSERT_VALUES);CHKERRQ(ierr);
45      }
46
47      ierr = MatAssemblyBegin(matriz_n,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
48      ierr = MatAssemblyEnd(matriz_n,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
49
50      /* ------------------------ DESCARTAR LOS ELEMENTOS 1D --------------------------*/
51
52      for (i=0; i<lineas; i++)
53      {
54          f >> nosirve >> nosirve >> nosirve >> nosirve;
55      }
56
57      /* ------------ LEER Y GUARDAR LOS VERTICES QUE PERTENECEN A CADA VC ------------*/
58
59      for (i=0; i<vc; i++)
60      {
61          f >> nosirve >> nosirve;
62          f >> value[0] >> value[1] >> value[2];
63
64          ierr = MatSetValues(matriz_vc,1,&i,3,col,value,INSERT_VALUES);CHKERRQ(ierr);
65      }
66
67      ierr = MatAssemblyBegin(matriz_vc,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
68      ierr = MatAssemblyEnd(matriz_vc,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
69
70      return ierr;
71  }
72
73  /******************** FUNCION CALCULAR LOS TERMINOS GEOMETRICOS ********************/
74
75  PetscErrorCode geom::calcular(void)
76  {
77
78      /* ----------------------  INICIALIZACIÓN DE VARIABLES  ------------------------ */
79
80      PetscInt col[3];
81      for (i=0;i<3;i++)
82      {
83          col[i]=i;
84      }
85
86      tempor=0;temporal=0;
87
88      //Matriz de coordenadas de cada vc
89      ierr = MatCreate(PETSC_COMM_WORLD,&matriz_coord);CHKERRQ(ierr);
90      ierr = MatSetSizes(matriz_coord,PETSC_DECIDE,PETSC_DECIDE,vc,2);CHKERRQ(ierr);
91      ierr = MatSetFromOptions(matriz_coord);CHKERRQ(ierr);
92      ierr = MatSetUp(matriz_coord);CHKERRQ(ierr);
93
94      //Vector de volúmenes
95      ierr = VecCreate(PETSC_COMM_WORLD,&vec_volumen);CHKERRQ(ierr);
96      ierr = VecSetSizes(vec_volumen,PETSC_DECIDE,vc);CHKERRQ(ierr);
97      ierr = VecSetFromOptions(vec_volumen);CHKERRQ(ierr);
98
```

```
99      //Matriz de vecinos
100     ierr = MatCreate(PETSC_COMM_WORLD,&matriz_vecinos);CHKERRQ(ierr);
101     ierr = MatSetSizes(matriz_vecinos,PETSC_DECIDE,PETSC_DECIDE,vc,3);CHKERRQ(ierr);
102     ierr = MatSetFromOptions(matriz_vecinos);CHKERRQ(ierr);
103     ierr = MatSetUp(matriz_vecinos);CHKERRQ(ierr);
104
105     //Matriz de vectores de area en las caras
106     ierr = MatCreate(PETSC_COMM_WORLD,&matriz_areas);CHKERRQ(ierr);
107     ierr = MatSetSizes(matriz_areas,PETSC_DECIDE,PETSC_DECIDE,vc,6);CHKERRQ(ierr);
108     ierr = MatSetFromOptions(matriz_areas);CHKERRQ(ierr);
109     ierr = MatSetUp(matriz_areas);CHKERRQ(ierr);
110
111     //Matriz de vectores de direccion PF
112     ierr = MatCreate(PETSC_COMM_WORLD,&matriz_vecpf);CHKERRQ(ierr);
113     ierr = MatSetSizes(matriz_vecpf,PETSC_DECIDE,PETSC_DECIDE,vc,6);CHKERRQ(ierr);
114     ierr = MatSetFromOptions(matriz_vecpf);CHKERRQ(ierr);
115     ierr = MatSetUp(matriz_vecpf);CHKERRQ(ierr);
116
117     //Matriz de vectores de direccion Pf
118     ierr = MatCreate(PETSC_COMM_WORLD,&matriz_vecpcent);CHKERRQ(ierr);
119     ierr = MatSetSizes(matriz_vecpcent,PETSC_DECIDE,PETSC_DECIDE,vc,6);CHKERRQ(ierr);
120     ierr = MatSetFromOptions(matriz_vecpcent);CHKERRQ(ierr);
121     ierr = MatSetUp(matriz_vecpcent);CHKERRQ(ierr);
122
123     //Matriz de centroides en las caras de cada vc
124     ierr = MatCreate(PETSC_COMM_WORLD,&matriz_centrocara);CHKERRQ(ierr);
125     ierr = MatSetSizes(matriz_centrocara,PETSC_DECIDE,PETSC_DECIDE,vc,6);CHKERRQ(ierr);
126     ierr = MatSetFromOptions(matriz_centrocara);CHKERRQ(ierr);
127     ierr = MatSetUp(matriz_centrocara);CHKERRQ(ierr);
128
129     //Matriz de alfaf
130     ierr = MatCreate(PETSC_COMM_WORLD,&matriz_alfaf);CHKERRQ(ierr);
131     ierr = MatSetSizes(matriz_alfaf,PETSC_DECIDE,PETSC_DECIDE,vc,3);CHKERRQ(ierr);
132     ierr = MatSetFromOptions(matriz_alfaf);CHKERRQ(ierr);
133     ierr = MatSetUp(matriz_alfaf);CHKERRQ(ierr);
134
135     //Matriz de vectores p-p_prima
136     ierr = MatCreate(PETSC_COMM_WORLD,&matriz_vecppprima);CHKERRQ(ierr);
137     ierr = MatSetSizes(matriz_vecppprima,PETSC_DECIDE,PETSC_DECIDE,vc,6);CHKERRQ(ierr);
138     ierr = MatSetFromOptions(matriz_vecppprima);CHKERRQ(ierr);
139     ierr = MatSetUp(matriz_vecppprima);CHKERRQ(ierr);
140
141     /* ---- CÁLCULO DE LA MATRIZ DE COORDENADAS DE CADA VC Y VECTOR DE VOLUMENES ---- */
142
143     for (j=0;j<vc;j++)
144     {
145         ierr = MatGetRow(matriz_vc,j,PETSC_NULL,PETSC_NULL,&vertvc);CHKERRQ(ierr);
146
147         for (i=0;i<3;i++)
148         {
149             k=vertvc[i];
150             ierr = MatGetRow(matriz_n,k,PETSC_NULL,PETSC_NULL,&coordvert);CHKERRQ(ierr);
151             vertx[i]=coordvert[0];
152             verty[i]=coordvert[1];
153             ierr = MatRestoreRow(matriz_n,k,PETSC_NULL,PETSC_NULL,&coordvert);CHKERRQ(ierr);
154         }
155         value[0]=(vertx[0]+vertx[1]+vertx[2])/3;
158         ierr = MatSetValues(matriz_coord,1,&j,2,col,value,INSERT_VALUES);CHKERRQ(ierr);
159
160         ladoa=sqrt(pow((vertx[0]-vertx[1]),2)+pow((verty[0]-verty[1]),2));
161         ladob=sqrt(pow((vertx[0]-vertx[2]),2)+pow((verty[0]-verty[2]),2));
162         ladoc=sqrt(pow((vertx[1]-vertx[2]),2)+pow((verty[1]-verty[2]),2));
163         per=(ladoa+ladob+ladoc)/2;
164         volumen=sqrt(per*(per-ladoa)*(per-ladob)*(per-ladoc));
165
166         ierr = VecSetValues(vec_volumen,1,&j,&volumen,INSERT_VALUES);CHKERRQ(ierr);
167
```

```
158         ierr = MatSetValues(matriz_coord,1,&j,2,col,value,INSERT_VALUES);CHKERRQ(ierr);
159
160         ladoa=sqrt(pow((vertx[0]-vertx[1]),2)+pow((verty[0]-verty[1]),2));
161         ladob=sqrt(pow((vertx[0]-vertx[2]),2)+pow((verty[0]-verty[2]),2));
162         ladoc=sqrt(pow((vertx[1]-vertx[2]),2)+pow((verty[1]-verty[2]),2));
163         per=(ladoa+ladob+ladoc)/2;
164         volumen=sqrt(per*(per-ladoa)*(per-ladob)*(per-ladoc));
165
166         ierr = VecSetValues(vec_volumen,1,&j,&volumen,INSERT_VALUES);CHKERRQ(ierr);
167
168         ierr = MatRestoreRow(matriz_vc,j,PETSC_NULL,PETSC_NULL,&vertvc);CHKERRQ(ierr);
169     }
170
171     ierr = MatAssemblyBegin(matriz_coord,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
172     ierr = MatAssemblyEnd(matriz_coord,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
173
174     ierr = VecAssemblyBegin(vec_volumen);CHKERRQ(ierr);
175     ierr = VecAssemblyEnd(vec_volumen);CHKERRQ(ierr);
176
177
178     /* --------------------- CÁLCULO DE LA MATRIZ DE VECINOS ---------------------- */
179
180     for (j=0;j<vc;j++)
181     {
182         ierr = MatGetRow(matriz_vc,j,PETSC_NULL,PETSC_NULL,&vertactual);CHKERRQ(ierr);
183         for (i=0;i<vc;i++)
184         {
185             if (i!=j)
186             {
187                 ierr = MatGetRow(matriz_vc,i,PETSC_NULL,PETSC_NULL,&vertcompar);CHKERRQ(ierr);
188                 for (k=0;k<3;k++)
189                 {
190                     if (vertactual[k]==vertcompar[0] || vertactual[k]==vertcompar[1] || vertactual[k]==
vertcompar[2])
191                     {
192                         temporal=temporal+1;
193                     }
194                 }
195
196                 if (temporal==2) //Si los vc comparten dos vértices son vecinos
197                 {
198                     value[tempor]=i;
199                     tempor=tempor+1;
200                 }
201
202                 temporal=0;
203                 ierr = MatRestoreRow(matriz_vc,i,PETSC_NULL,PETSC_NULL,&vertcompar);CHKERRQ(ierr);
204             }
205         }
206
207         //Si tiene 3 vecinos
208         if (tempor==3)
209         {
210             ierr = MatSetValues(matriz_vecinos,1,&j,3,col,value,INSERT_VALUES);CHKERRQ(ierr);
211         }
212         //Si tiene 2 vecinos (Frontera)
213         if (tempor==2)
214         {
215             value[2]=vc*10; //Valor para identificar que un vc es frontera
216             ierr = MatSetValues(matriz_vecinos,1,&j,3,col,value,INSERT_VALUES);CHKERRQ(ierr);
217         }
218         tempor=0;
219         ierr = MatRestoreRow(matriz_vc,j,PETSC_NULL,PETSC_NULL,&vertactual);CHKERRQ(ierr);
220     }
```

```
221
222     ierr = MatAssemblyBegin(matriz_vecinos,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
223     ierr = MatAssemblyEnd(matriz_vecinos,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
224
225
226     /* ---------------- CACULO DE AREAS,CENTROIDES Y VECTORES DE DIRECCIÓN ---------------- */
227
228     tem=0;temporal=0;tempora=0;tempor=0;
229     colareas[0]=0;  colareas[1]=1;
230
231     for (i=0;i<vc;i++)
232     {
233         ierr = MatGetRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
234         ierr = MatGetRow(matriz_coord,i,PETSC_NULL,PETSC_NULL,&vcactual);CHKERRQ(ierr);
235         ierr = MatGetRow(matriz_vc,i,PETSC_NULL,PETSC_NULL,&vertactual);CHKERRQ(ierr);
236
237         for (j=0;j<3;j++)
238         {
239             tem=vecinos[j];
240             jj[0]=j;
241
242             //Se obtienen las coordenadas de los vértices en común con el vécino
243             if (tem>vc) //En caso de que la cara sea frontera
244             {
245                 if (nodvec[0]==nodvec[2])
246                 {
247                     temporal=nodvec[1];
248                     tempora=nodvec[3];
249                 }
250                 if (nodvec[0]==nodvec[3])
251                 {
252                     temporal=nodvec[1];
253                     tempora=nodvec[2];
254                 }
255                 if (nodvec[1]==nodvec[2])
256                 {
257                     temporal=nodvec[0];
258                     tempora=nodvec[3];
259                 }
260                 if (nodvec[1]==nodvec[3])
261                 {
262                     temporal=nodvec[0];
263                     tempora=nodvec[2];
264                 }
265
266                 ierr = MatGetRow(matriz_n,temporal,PETSC_NULL,PETSC_NULL,&coordvert);CHKERRQ(ierr);
267                 n_x[0]=coordvert[0];
268                 n_y[0]=coordvert[1];
269                 ierr = MatRestoreRow(matriz_n,temporal,PETSC_NULL,PETSC_NULL,&coordvert);CHKERRQ(ierr);
270                 ierr = MatGetRow(matriz_nn,tempora,PETSC_NULL,PETSC_NULL,&coordvert);CHKERRQ(ierr);
271                 n_x[1]=coordvert[0];
272                 n_y[1]=coordvert[1];
273                 ierr = MatRestoreRow(matriz_n,tempora,PETSC_NULL,PETSC_NULL,&coordvert);CHKERRQ(ierr);
274             }
275
276             else //En caso de que la cara no sea frontera
277             {
278                 ierr = MatGetRow(matriz_coord,tem,PETSC_NULL,PETSC_NULL,&vcvecino);CHKERRQ(ierr);
279                 ierr = MatGetRow(matriz_vc,tem,PETSC_NULL,PETSC_NULL,&vertvecino);CHKERRQ(ierr);
280
281                 for (k=0;k<3;k++)
282                 {
```

```
283                     if (vertvecino[k]==vertactual[0] || vertvecino[k]==vertactual[1] || vertvecino[k]==
vertactual[2])
284                     {
285                         nodvec[tempora]=vertvecino[k];
286                         tempora=tempora+1;
287                         temporal=vertvecino[k];
288                         ierr = MatGetRow(matriz_n,temporal,PETSC_NULL,PETSC_NULL,&coordvert);CHKERRQ(ierr);
289                         n_x[tempor]=coordvert[0];
290                         n_y[tempor]=coordvert[1];
291                         ierr = MatRestoreRow(matriz_n,temporal,PETSC_NULL,PETSC_NULL,&coordvert);CHKERRQ(
ierr);
292                         tempor=tempor+1;
293                     }
294                 }
295                 tempor=0;
296                 ierr = MatRestoreRow(matriz_vc,tem,PETSC_NULL,PETSC_NULL,&vertvecino);CHKERRQ(ierr);
297             }
298
299             //La dirección del vector área debe ser normal a la cara del vc actual
300             if (n_x[0]==n_x[1])
301             {
302                 area[1]=0;
303
304                 if(vcactual[0]<n_x[0])
305                 {
306                     if(n_y[0]<n_y[1])
307                     {
308                         area[0]=n_y[1]-n_y[0];
309                     }
310                     else
311                     {
312                         area[0]=n_y[0]-n_y[1];
313                     }
314                 }
315                 if(vcactual[0]>n_x[0])
316                 {
317                     if(n_y[0]<n_y[1])
318                     {
319                         area[0]=n_y[0]-n_y[1];
320                     }
321                     else
322                     {
323                         area[0]=n_y[1]-n_y[0];
324                     }
325                 }
326             }
327
329             else
330             {
331                 pendiente=(n_y[1]-n_y[0])/(n_x[1]-n_x[0]);
332                 bcorte=n_y[0]-pendiente*n_x[0];
333
334                 if (vcactual[1]<(pendiente*vcactual[0]+bcorte))
335                 {
336                     if (n_x[0]<n_x[1])
337                     {
338                         area[0]=n_y[0]-n_y[1];
339                         area[1]=n_x[1]-n_x[0];
340                     }
341                     if (n_x[0]>n_x[1])
342                     {
343                         area[0]=n_y[1]-n_y[0];
344                         area[1]=n_x[0]-n_x[1];
345                     }
346                 }
```

```
347                 if (vcactual[1]>(pendiente*vcactual[0]+bcorte))
348                 {
349                     if (n_x[0]<n_x[1])
350                     {
351                         area[0]=n_y[1]-n_y[0];
352                         area[1]=n_x[0]-n_x[1];
353                     }
354                     if (n_x[0]>n_x[1])
355                     {
356                         area[0]=n_y[0]-n_y[1];
357                         area[1]=n_x[1]-n_x[0];
358                     }
359                 }
360             }
361
362             centrocara[0]=(n_x[0]+n_x[1])/2;
363             centrocara[1]=(n_y[0]+n_y[1])/2;
364
365             //Vector PF
366             if(tem>vc) //Si la cara es frontera F está en el centroide de la cara
367             {
368                 pf[0]=centrocara[0]-vcactual[0];
369                 pf[1]=centrocara[1]-vcactual[1];
370             }
371             else //Si la cara no es frontera F es el nodo del vc vecino
372             {
373                 pf[0]=vcvecino[0]-vcactual[0];
374                 pf[1]=vcvecino[1]-vcactual[1];
375                 ierr = MatRestoreRow(matriz_coord,tem,PETSC_NULL,PETSC_NULL,&vcvecino);CHKERRQ(ierr);
376             }
377
378             //Vector Pf
379             pcent[0]=centrocara[0]-vcactual[0];
380             pcent[1]=centrocara[1]-vcactual[1];
381
382             //Valor de alfa_f
383             alfa[0]=(pcent[0]*pf[0]+pcent[1]*pf[1])/(pf[0]*pf[0]+pf[1]*pf[1]);
384
385             //Vector PP'
386             if (area[0]==0)
387             {
388                 p_prima[0]=centrocara[0];
389                 p_prima[1]=vcactual[1];
390             }
391             if (area[1]==0)
392             {
394                 p_prima[0]=vcactual[0];
395             }
396             if (area[0]!=0 && area[1]!=0)
397             {
398                 m_p=-area[0]/area[1];
399                 b_p=vcactual[1]-m_p*vcactual[0];
400                 m_a=area[1]/area[0];
401                 b_a=centrocara[1]-m_a*centrocara[0];
402                 p_prima[0]=(b_a-b_p)/(m_p-m_a);
403                 p_prima[1]=m_p*p_prima[0]+b_p;
404             }
405
406             pp_prima[0]=p_prima[0]-vcactual[0];
407             pp_prima[1]=p_prima[1]-vcactual[1];
408
409             //Ingresar todos los valores
410             ierr = MatSetValues(matriz_areas,1,&i,2,colareas,area,INSERT_VALUES);CHKERRQ(ierr);
411             ierr = MatSetValues(matriz_vecpf,1,&i,2,colareas,pf,INSERT_VALUES);CHKERRQ(ierr);
412             ierr = MatSetValues(matriz_vecpcent,1,&i,2,colareas,pcent,INSERT_VALUES);CHKERRQ(ierr);
413             ierr = MatSetValues(matriz centrocara,1,&i,2,colareas,centrocara,INSERT VALUES);CHKERRQ(ierr);
```

```
413             ierr = MatSetValues(matriz_centrocara,1,&i,2,colareas,centrocara,INSERT_VALUES);CHKERRQ(ierr);
414             ierr = MatSetValues(matriz_alfaf,1,&i,1,jj,alfa,INSERT_VALUES);CHKERRQ(ierr);
415             ierr = MatSetValues(matriz_vecppprima,1,&i,2,colareas,pp_prima,INSERT_VALUES);CHKERRQ(ierr);
416
417             colareas[0]=colareas[0]+2;
418             colareas[1]=colareas[1]+2;
419         }
420         tempora=0;
421         colareas[0]=0;
422         colareas[1]=1;
423
424         ierr = MatRestoreRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
425         ierr = MatRestoreRow(matriz_coord,i,PETSC_NULL,PETSC_NULL,&vcactual);CHKERRQ(ierr);
426         ierr = MatRestoreRow(matriz_vc,i,PETSC_NULL,PETSC_NULL,&vertactual);CHKERRQ(ierr);
427     }
428
429     //Ensamblar las matrices
430     ierr = MatAssemblyBegin(matriz_areas,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
431     ierr = MatAssemblyEnd(matriz_areas,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
432
433     ierr = MatAssemblyBegin(matriz_vecpf,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
434     ierr = MatAssemblyEnd(matriz_vecpf,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
435
436     ierr = MatAssemblyBegin(matriz_vecpcent,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
437     ierr = MatAssemblyEnd(matriz_vecpcent,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
438
439     ierr = MatAssemblyBegin(matriz_centrocara,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
440     ierr = MatAssemblyEnd(matriz_centrocara,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
441
442     ierr = MatAssemblyBegin(matriz_alfaf,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
443     ierr = MatAssemblyEnd(matriz_alfaf,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
444
445     ierr = MatAssemblyBegin(matriz_vecppprima,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
446     ierr = MatAssemblyEnd(matriz_vecppprima,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
447
448
449     return ierr;
450 }
451
```

4. ARCHIVO DE ENCABEZADO DE LA CLASE FRONTERA

```
1  //ARCHIVO DE ENCABEZADO DE LA CLASE FRONTERA: frontera.h
2
3  #ifndef _FRONTERA_H_
4  #define _FRONTERA_H_
5
6  class frontera
7  {
8      private:
9
10         PetscErrorCode ierr;    //Variable error de petsc
11         int i,temp;             //Contadores
12
13         //Número de volúmenes de control
14         int n_vc;
15
16         //Elementos geométricos
17         Mat matriz_vecinos;
18         const PetscScalar *vecinos;
19         Mat matriz_areas;
20         const PetscScalar *areas;
```

```
        Mat matriz_coord;
        const PetscScalar *coord;

        //Vector con los valores de phi en las fronteras
        Vec vec_front;
        PetscScalar valuefront;
        //Vector con las condiciones de frontera
        Vec vec_condf;
        PetscScalar condf;

    public:

        //Funciones
        PetscErrorCode leerR(geom gm,double *cf,double *vf);
        PetscErrorCode leerBFS(geom gm,double *cf,double *vf);
        PetscErrorCode leerSH(geom gm);
};

#endif // _FRONTERA_H_
```

5. ARCHIVO DE CÓDIGO FUENTE DE LA CLASE FRONTERA

```
//ARCHIVO DE CÓDIGO FUENTE DE LA CLASE FRONTERA: frontera.cpp

#include <petsc.h>
#include <cmath>
#include "geom.h"

/**************** FUNCION ESTABLECER LA FRONTERA: RECTANGULAR, 4 FRONTERAS ****************/

PetscErrorCode frontera::leerR(geom gm,double *cf,double *vf)
{
    //Tomar los elementos geométricos de la clase geom
    n_vc=gm.vc;
    matriz_vecinos=gm.matriz_vecinos;
    matriz_areas=gm.matriz_areas;
    matriz_coord=gm.matriz_coord;

    /* ------------------------ INICIALIZACIÓN DE VARIABLES ---------------------------- */

    //Condiciones de frontera
    ierr = VecCreate(PETSC_COMM_WORLD,&vec_condf);CHKERRQ(ierr);
    ierr = VecSetSizes(vec_condf,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
    ierr = VecSetFromOptions(vec_condf);CHKERRQ(ierr);
    ierr = VecSet(vec_condf,0);CHKERRQ(ierr);

    //Valores en la frontera
    ierr = VecCreate(PETSC_COMM_WORLD,&vec_front);CHKERRQ(ierr);
    ierr = VecSetSizes(vec_front,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
    ierr = VecSetFromOptions(vec_front);CHKERRQ(ierr);
    ierr = VecSet(vec_front,0);CHKERRQ(ierr);

    /* ------------------------- ESTABLECER LAS FRONTERAS ------------------------------ */

    for (i=0;i<n_vc;i++)
    {
        ierr = MatGetRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
        ierr = MatGetRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
        ierr = MatGetRow(matriz_coord,i,PETSC_NULL,PETSC_NULL,&coord);CHKERRQ(ierr);

        if (vecinos[2]>n_vc) //Si el volumen de control tiene una cara frontera
        {
            //Si pertenece a la frontera superior
```

```
41              //Si pertenece a la frontera superior
42              if (areas[4]<0.00001 && areas[4]>-0.00001 && areas[5]>0 )
43              {   condf=cf[0];
44                  valuefront=vf[0];
45              }
46              //Si pertenece a la frontera inferior
47              if (areas[4]<0.00001 && areas[4]>-0.00001 && areas[5]<0 )
48              {
49                  condf=cf[2];
50                  valuefront=vf[2];
51              }
52              //Si pertenece a la frontera derecha
53              if(areas[5]<0.00001 && areas[5]>-0.00001 && areas[4]>0)
54              {
55                  condf=cf[1];
56                  valuefront=vf[1];
57              }
58              //Si pertenece a la frontera izquierda
59              if(areas[5]<0.00001 && areas[5]>-0.00001 && areas[4]<0)
60              {
61                  condf=cf[3];
62                  valuefront=vf[3];
63              }
64
65              ierr = VecSetValues(vec_condf,1,&i,&condf,INSERT_VALUES);CHKERRQ(ierr);
66              ierr = VecSetValues(vec_front,1,&i,&valuefront,INSERT_VALUES);CHKERRQ(ierr);
67          }
68
69          ierr = MatRestoreRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
70          ierr = MatRestoreRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
71          ierr = MatRestoreRow(matriz_coord,i,PETSC_NULL,PETSC_NULL,&coord);CHKERRQ(ierr);
72      }
73
74      ierr = VecAssemblyBegin(vec_condf);CHKERRQ(ierr);
75      ierr = VecAssemblyEnd(vec_condf);CHKERRQ(ierr);
76
77      ierr = VecAssemblyBegin(vec_front);CHKERRQ(ierr);
78      ierr = VecAssemblyEnd(vec_front);CHKERRQ(ierr);
79
80      return ierr;
81  }
82
83  /************** FUNCION ESTABLECER LA FRONTERA: PROBLEMA BACKWARD-FACING STEP **************/
84
85  PetscErrorCode frontera::leerBFS(geom gm,double *cf,double *vf)
86  {
87      //Tomar los elementos necesarios de la clase geom
88      n_vc=gm.vc;
89      matriz_vecinos=gm.matriz_vecinos;
90      matriz_areas=gm.matriz_areas;
91      matriz_coord=gm.matriz_coord;
92
93      /*-------------------------- INICIALIZACIÓN DE LAS VARIABLES --------------------------*/
94
95      ierr = VecCreate(PETSC_COMM_WORLD,&vec_condf);CHKERRQ(ierr);
96      ierr = VecSetSizes(vec_condf,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
97      ierr = VecSetFromOptions(vec_condf);CHKERRQ(ierr);
98      ierr = VecSet(vec_condf,0);CHKERRQ(ierr);
99
100     ierr = VecCreate(PETSC_COMM_WORLD,&vec_front);CHKERRQ(ierr);
101     ierr = VecSetSizes(vec_front,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
102     ierr = VecSetFromOptions(vec_front);CHKERRQ(ierr);
103     ierr = VecSet(vec_front,0);CHKERRQ(ierr);
104
105     /*------------------------------ ESTABLECER LA FRONTERA ------------------------------*/
106
107     for (i=0;i<n_vc;i++)
```

```
108     {
109         ierr = MatGetRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
110         ierr = MatGetRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
111         ierr = MatGetRow(matriz_coord,i,PETSC_NULL,PETSC_NULL,&coord);CHKERRQ(ierr);
112 
113         temp=0;
114 
115         if (vecinos[2]>n_vc) //Si el volumen de control tiene una cara frontera
116         {
117             //Si pertenece a la frontera superior o inferior
118             if (areas[4]<0.00001 && areas[4]>-0.00001)
119             {   condf=cf[0];
120                 valuefront=vf[0];
121                 temp=1;
122             }
123 
124             //Si pertenece a la frontera derecha
125             if(areas[5]<0.00001 && areas[5]>-0.00001 && areas[4]>0)
126             {
127                 condf=cf[1];
128                 valuefront=vf[1];
129                 temp=1;
130             }
131 
132             //Si pertenece a la frontera izquierda
133             if(areas[5]<0.00001 && areas[5]>-0.00001 && areas[4]<0 && coord[1]>0)
134             {
135                 condf=cf[3];
136                 valuefront=vf[3];
137                 temp=1;
138             }
139 
140             //Si pertenece a la frontera step
141             if(areas[5]<0.00001 && areas[5]>-0.00001 && areas[4]<0 && coord[1]<0)
142             {
143                 condf=cf[2];
144                 valuefront=vf[2];
145                 temp=1;
146             }
147 
148             //Si pertenece a la frontera step inclinado
149             if(temp!=1)
150             {
151                 condf=cf[2];
152                 valuefront=vf[2];
153             }
154             ierr = VecSetValues(vec_condf,1,&i,&condf,INSERT_VALUES);CHKERRQ(ierr);
155             ierr = VecSetValues(vec_front,1,&i,&valuefront,INSERT_VALUES);CHKERRQ(ierr);
156         }
157 
158         ierr = MatRestoreRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
159         ierr = MatRestoreRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
160         ierr = MatRestoreRow(matriz_coord,i,PETSC_NULL,PETSC_NULL,&coord);CHKERRQ(ierr);
161     }
162 
163     ierr = VecAssemblyBegin(vec_condf);CHKERRQ(ierr);
164     ierr = VecAssemblyEnd(vec_condf);CHKERRQ(ierr);
165 
166     ierr = VecAssemblyBegin(vec_front);CHKERRQ(ierr);
167     ierr = VecAssemblyEnd(vec_front);CHKERRQ(ierr);
168 
169     return ierr;
170 }
171 
172 /****************** FUNCION ESTABLECER LA FRONTERA: PROBLEMA SMITH-HUTTON ******************/
173 
174 PetscErrorCode frontera::leerSH(geom gm)
```

```
{
    //Tomar los elementos necesarios de la clase geom
    n_vc=gm.vc;
    matriz_vecinos=gm.matriz_vecinos;
    matriz_areas=gm.matriz_areas;
    matriz_coord=gm.matriz_coord;

    /*-------------------------- INICIALIZACIÓN DE LAS VARIABLES --------------------------*/

    ierr = VecCreate(PETSC_COMM_WORLD,&vec_condf);CHKERRQ(ierr);
    ierr = VecSetSizes(vec_condf,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
    ierr = VecSetFromOptions(vec_condf);CHKERRQ(ierr);
    ierr = VecSet(vec_condf,0);CHKERRQ(ierr);

    ierr = VecCreate(PETSC_COMM_WORLD,&vec_front);CHKERRQ(ierr);
    ierr = VecSetSizes(vec_front,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
    ierr = VecSetFromOptions(vec_front);CHKERRQ(ierr);
    ierr = VecSet(vec_front,0);CHKERRQ(ierr);

    /*------------------------------ ESTABLECER LA FRONTERA ------------------------------*/

    for (i=0;i<n_vc;i++)
    {
        ierr = MatGetRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
        ierr = MatGetRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
        ierr = MatGetRow(matriz_coord,i,PETSC_NULL,PETSC_NULL,&coord);CHKERRQ(ierr);

        if (vecinos[2]>n_vc) //Si el volumen de control tiene una cara frontera
        {
            //Si pertenece a la frontera inferior positiva.
            if (areas[4]<0.00001 && areas[4]>-0.00001 && areas[5]<0 && coord[0]>0)
            {   condf=-1;
                valuefront=0;
            }

            //Si pertenece a la frontera inferior negativa.
            if (areas[4]<0.00001 && areas[4]>-0.00001 && areas[5]<0 && coord[0]<0)
            {
                condf=1;
                valuefront= 1+tanh(10*(2*coord[0]+1));
            }

            //Si pertenece a la frontera superior
            if (areas[4]<0.00001 && areas[4]>-0.00001 && areas[5]>0)
            {
                condf=1;
                valuefront= 1-(tanh(10));
            }

            //Si pertenece a la drontera derecha
            if(areas[5]<0.00001 && areas[5]>-0.00001 && areas[4]>0)
            {
                condf=1;
                valuefront=1-(tanh(10));
            }

            //Si pertence a la frontera izquierda
            if(areas[5]<0.00001 && areas[5]>-0.00001 && areas[4]<0)
            {
                condf=1;
                valuefront=1-(tanh(10));
            }

            ierr = VecSetValues(vec_condf,1,&i,&condf,INSERT_VALUES);CHKERRQ(ierr);
            ierr = VecSetValues(vec_front,1,&i,&valuefront,INSERT_VALUES);CHKERRQ(ierr);
```

```
240         }
241
242         ierr = MatRestoreRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
243         ierr = MatRestoreRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
244         ierr = MatRestoreRow(matriz_coord,i,PETSC_NULL,PETSC_NULL,&coord);CHKERRQ(ierr);
245     }
246
247
248     ierr = VecAssemblyBegin(vec_condf);CHKERRQ(ierr);
249     ierr = VecAssemblyEnd(vec_condf);CHKERRQ(ierr);
250
251     ierr = VecAssemblyBegin(vec_front);CHKERRQ(ierr);
252     ierr = VecAssemblyEnd(vec_front);CHKERRQ(ierr);
253
254     return ierr;
255 }
```

6. ARCHIVO DE ENCABEZADO DE LA CLASE DIFUSIVO

```
1  //ARCHIVO DE ENCABEZADO DE LA CLASE DIFUSIVO: difusivo.h
2
3  #ifndef _DIFUSIVO_H_
4  #define _DIFUSIVO_H_
5
6  class difusivo
7  {
8      private:
9
10         PetscErrorCode ierr;        //Variable de error de Petsc
11         int i,j;                    //Contadores
12         int tem,tempor;             //Variables de uso temporal
13
14         //Elementos geométricos
15         int n_vc;
16         Mat matriz_areas;
17         const PetscScalar *area;
18         Mat matriz_vecpf;
19         const PetscScalar *pf;
20         Mat matriz_vecinos;
21         const PetscScalar *vecinos;
22
23         //Variable para establecer las columnas de la matriz de terminos difusivos
24         PetscInt coldif[3];
25         //término difusivos a ingresar a la matriz
26         PetscScalar value[3];
27         //término difusivo a ingresar en el vector
28         PetscScalar diffront;
29
30     public:
31
32         //Funcines
33         PetscErrorCode calcular_D(double gamma,geom gm,Mat *matriz_difusivo,Vec *vec_diffront);
34
35 };
36
37 #endif // _DIFUSIVO_H_
```

7. ARCHIVO DE CÓDIGO FUENTE DE LA CLASE DIFUSIVO

```
1  //ARCHIVO DE CODIGO FUENTE DE LA CLASE DIFUSIVO: difusivo.cpp
2
3  #include <petsc.h>
4  #include <cmath>
5  #include "geom.h"
6  #include "frontera.h"
7
8  /******************** FUNCIÓN PARA EL CÁLCULO DE LOS TÉRMINOS DIFUSIVOS ********************/
9
10 PetscErrorCode difusivo::calcular_D(double gamma,geom b,Mat *matriz_difusivo,Vec *vec_diffront)
11 {
12     //Tomar los elementos geométricos de la clase geom
13     n_vc=b.vc;
14     matriz_areas=b.matriz_areas;
15     matriz_vecpf=b.matriz_vecpf;
16     matriz_vecinos=b.matriz_vecinos;
17
18     /* --------------------- CALCULAR Y GUARDAR LOS TÉRMINOS DIFUSIVOS ----------------------*/
19
20     tem=0;tempor=1;
21     for (i=0;i<n_vc;i++)
22     {
23         ierr = MatGetRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&area);CHKERRQ(ierr);
24         ierr = MatGetRow(matriz_vecpf,i,PETSC_NULL,PETSC_NULL,&pf);CHKERRQ(ierr);
25         ierr = MatGetRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
26
27         //Cálculo del término difusivo
28         for (j=0;j<3;j++)
29         {
30             value[j]=gamma*(area[tem]*area[tem]+area[tempor]*area[tempor])/(area[tem]*pf[tem]+area[tempor]*
pf[tempor]);
31             tem=tem+2;
32             tempor=tempor+2;
33         }
34         tem=0;tempor=1;
35
36         //Establecer las columnas de la matriz
37         coldif[0]=vecinos[0];
38         coldif[1]=vecinos[1];
39         coldif[2]=vecinos[2];
40
41         //Insertar los valores en la matriz y el vector(en caso de que sea frontera)
42         if (vecinos[2]>n_vc) // Si el vc tiene frontera
43         {
44             diffront=value[2];
45             ierr = VecSetValues(*vec_diffront,1,&i,&diffront,INSERT_VALUES);CHKERRQ(ierr);
46             ierr = MatSetValues(*matriz_difusivo,1,&i,2,coldif,value,INSERT_VALUES);CHKERRQ(ierr);
47         }
48
49         else //Si el vc no tiene frontera
51             ierr = MatSetValues(*matriz_difusivo,1,&i,3,coldif,value,INSERT_VALUES);CHKERRQ(ierr);
52         }
53         ierr = MatRestoreRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&area);CHKERRQ(ierr);
54         ierr = MatRestoreRow(matriz_vecpf,i,PETSC_NULL,PETSC_NULL,&pf);CHKERRQ(ierr);
55         ierr = MatRestoreRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
56     }
57
58     ierr = MatAssemblyBegin(*matriz_difusivo,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
59     ierr = MatAssemblyEnd(*matriz_difusivo,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
60
61     ierr = VecAssemblyBegin(*vec_diffront);CHKERRQ(ierr);
62     ierr = VecAssemblyEnd(*vec_diffront);CHKERRQ(ierr);
63
64     return ierr;
```

8. ARCHIVO DE ENCABEZADO DE LA CLASE GRADIENTE

```
1  //ARCHIVO DE ENCABEZADO DE LA CLASE GRADIENTE: gradiente.h
2
3  #ifndef _GRADIENTE_H_
4  #define _GRADIENTE_H_
5
6  class gradiente
7  {
8      private:
9
10         PetscErrorCode ierr;            //Variable de error de Petsc
11         int i,j,k,ii;                   //Contadores
12         int tem,tempor,tempora;         //Variables temporales
13
14         //Elementos geométricos
15         int n_vc;
16         Vec vec_volumen;
17         PetscScalar *volumenvc;
18         Mat matriz_areas;
19         const PetscScalar *areas;
20         Mat matriz_vecinos;
21         const PetscScalar *vecinos;
22         const PetscScalar *vvecinos;
23         Mat matriz_alfaf;
24         const PetscScalar *alfaf;
25
26         //Elementos de frontera
27         Vec vec_condf;
28         PetscScalar *condf;
29         Vec vec_front;
30         PetscScalar *valueb;
31
32         //Variable para tomar los valores de phi en los centroides de las celdas
33         PetscScalar *phivc0;
34
35         //Variables para el cálculo de phi en las caras
36         PetscScalar phif0[3];
37         const PetscScalar *phi_f0;
38
39         //Variable para calcular el gradiente
40         PetscScalar gradphivc[2];
41
42     public:
43
44         //Matriz de valores de phi en las caras de cada volumen de control
45         Mat matriz_phif;
46
47         //Funciones
48         PetscErrorCode calcular_vcara(geom gm,frontera f,Vec vec_soluant,Mat *matriz_phif0);
49         PetscErrorCode calcular_grad(geom gm,frontera f,Vec vec_soluant,Mat *matriz_gradphivc);
50
51 };
52
53 #endif // _GRADIENTE_H_
```

martes, 20 d

9. ARCHIVO DE CÓDIGO FUENTE DE LA CLASE GRADIENTE

```
1  //ARCHIVO DE CÓDIGO FUENTE DE LA CLASE GRADIENTE: gradiente.cpp
2
3  #include <petsc.h>
4  #include <cmath>
5  #include "geom.h"
6  #include "frontera.h"
7
8  /******************* FUNCIÓN CALCULAR LOS VALORES DE PHI EN LAS CARAS ********************/
9
10 PetscErrorCode gradiente::calcular_vcara(geom gm,frontera f,Vec vec_phi,Mat *matriz_phif0)
11 {
12     //Tomar los elementos geométricos de la clase geom
13     n_vc=gm.vc;
14     vec_volumen=gm.vec_volumen;
15     matriz_areas=gm.matriz_areas;
16     matriz_vecinos=gm.matriz_vecinos;
17     matriz_alfaf=gm.matriz_alfaf;
18
19     //Tomar los elementos de la clase frontera
20     vec_condf=f.vec_condf;
21     vec_front=f.vec_front;
22
23     /* --------------- CALCULAR Y GUARDAR LOS VALORES DE PHI EN LAS CARAS -----------------*/
24
25     PetscInt col[n_vc];
26     for (i=0;i<n_vc;i++)
27     {
28         col[i]=i;
29     }
30
31     ierr = VecGetArray(vec_volumen,&volumenvc);CHKERRQ(ierr);
32     ierr = VecGetArray(vec_phi,&phivc0);CHKERRQ(ierr);
33     ierr = VecGetArray(vec_condf,&condf);CHKERRQ(ierr);
34     ierr = VecGetArray(vec_front,&valueb);CHKERRQ(ierr);
35
36     for (i=0;i<n_vc;i++)
37     {
38         ierr = MatGetRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
39         ierr = MatGetRow(matriz_alfaf,i,PETSC_NULL,PETSC_NULL,&alfaf);CHKERRQ(ierr);
40
41         //Identificar celdas vecinas
42         ierr = MatGetRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
43         tem=vecinos[0];
44         tempor=vecinos[1];
45         tempora=vecinos[2];
46         ierr = MatRestoreRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
47
48         //Calcular la variable en las caras
49         phif0[0]=(1-alfaf[0])*phivc0[i]+alfaf[0]*phivc0[tem];
50         phif0[1]=(1-alfaf[1])*phivc0[i]+alfaf[1]*phivc0[tempor];
51
52         if (tempora>n_vc)   //Si es un vc frontera
53         {
54             if (condf[i]==1)    //Condición Dirichlet
55             {phif0[2]=valueb[i];}
56             if(condf[i]==-1)    //Condición Newman
57             {phif0[2]=phivc0[i];}
58             if(condf[i]==2)     //En el caso de la velocidad
59             {
60                 if (valueb[i]==1)
61                 {phif0[2]=((-phif0[0]*areas[0])-(phif0[1]*areas[2]))/areas[4];}
62                 if (valueb[i]==2)
63                 {phif0[2]=0;}
```

```
64              }
65          }
66          else    //Si no es un vc frontera
67          {
68              phif0[2]=(1-alfaf[2])*phivc0[i]+alfaf[2]*phivc0[tempora];
69          }
70
71          ierr = MatSetValues(*matriz_phif0,1,&i,3,col,phif0,INSERT_VALUES);CHKERRQ(ierr);
72
73          ierr = MatRestoreRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
74          ierr = MatRestoreRow(matriz_alfaf,i,PETSC_NULL,PETSC_NULL,&alfaf);CHKERRQ(ierr);
75      }
76
77      ierr = VecRestoreArray(vec_volumen,&volumenvc);CHKERRQ(ierr);
78      ierr = VecRestoreArray(vec_phi,&phivc0);CHKERRQ(ierr);
79      ierr = VecRestoreArray(vec_condf,&condf);CHKERRQ(ierr);
80      ierr = VecRestoreArray(vec_front,&valueb);CHKERRQ(ierr);
81
82      ierr = MatAssemblyBegin(*matriz_phif0,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
83      ierr = MatAssemblyEnd(*matriz_phif0,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
84
85      return ierr;
86  }
87
88
89  /************************ FUNCIÓN CALCULAR EL GRADIENTE DE PHI ****************************/
90
91  PetscErrorCode gradiente::calcular_grad(geom gm,frontera f,Vec vec_phi,Mat *matriz_gradphivc)
92  {
93      //Tomar los elementos geométricos de la clase geom
94      n_vc=gm.vc;
95      vec_volumen=gm.vec_volumen;
96      matriz_areas=gm.matriz_areas;
97      matriz_vecinos=gm.matriz_vecinos;
98      matriz_alfaf=gm.matriz_alfaf;
99
100     //Tomar los elementos de la clase frontera
101     vec_condf=f.vec_condf;
102     vec_front=f.vec_front;
103
104     /* --------------------------- INICIALIZACIÓN DE VARIABLES --------------------------- */
105
106     PetscInt col[n_vc];
107     for (i=0;i<n_vc;i++)
108     {
109         col[i]=i;
110     }
111
112     //Matriz de phi en las caras del vc
113     ierr = MatCreate(PETSC_COMM_WORLD,&matriz_phif);CHKERRQ(ierr);
114     ierr = MatSetSizes(matriz_phif,PETSC_DECIDE,PETSC_DECIDE,n_vc,3);CHKERRQ(ierr);
115     ierr = MatSetFromOptions(matriz_phif);CHKERRQ(ierr);
116     ierr = MatSetUp(matriz_phif);CHKERRQ(ierr);
117
118     /* -------------------- CALCULAR LOS VALORES DE PHI EN LAS CARAS ----------------------*/
119
120     ierr = VecGetArray(vec_volumen,&volumenvc);CHKERRQ(ierr);
121     ierr = VecGetArray(vec_soluant,&phivc0);CHKERRQ(ierr);
122     ierr = VecGetArray(vec_condf,&condf);CHKERRQ(ierr);
123     ierr = VecGetArray(vec_front,&valueb);CHKERRQ(ierr);
124
125     for (i=0;i<n_vc;i++)
126     {
127         ierr = MatGetRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
128         ierr = MatGetRow(matriz_alfaf,i,PETSC_NULL,PETSC_NULL,&alfaf);CHKERRQ(ierr);
129
```

```
130         //Identificar celdas vecinas
131         ierr = MatGetRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
132         tem=vecinos[0];
133         tempor=vecinos[1];
134         tempora=vecinos[2];
135         ierr = MatRestoreRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
136
137         //Calcular la variable en las caras
138         phif0[0]=(1-alfaf[0])*phivc0[i]+alfaf[0]*phivc0[tem];
139         phif0[1]=(1-alfaf[1])*phivc0[i]+alfaf[1]*phivc0[tempor];
140
141         if (tempora>n_vc)   //Si es un vc frontera
142         {
143             if (condf[i]==1)     //Condición Dirichlet
144             {phif0[2]=valueb[i];}
145             if(condf[i]==-1)     //Condición Newman
146             {phif0[2]=phivc0[i];}
147             if(condf[i]==2)      //Condición outflow para la velocidad
148             {
149                 if (valueb[i]==1)
150                 {phif0[2]=((-phif0[0]*areas[0])-(phif0[1]*areas[2]))/areas[4];}
151                 if (valueb[i]==2)
152                 {phif0[2]=0;}
153             }
154         }
155         else    //Si no es un vc frontera
156         {
157             phif0[2]=(1-alfaf[2])*phivc0[i]+alfaf[2]*phivc0[tempora];
158         }
159
160         ierr = MatSetValues(matriz_phif,1,&i,3,col,phif0,INSERT_VALUES);CHKERRQ(ierr);
161
162         ierr = MatRestoreRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
163         ierr = MatRestoreRow(matriz_alfaf,i,PETSC_NULL,PETSC_NULL,&alfaf);CHKERRQ(ierr);
164     }
165
166     ierr = MatAssemblyBegin(matriz_phif,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
167     ierr = MatAssemblyEnd(matriz_phif,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
168
169     /* -------------- CALCULAR Y GUARDAR LOS VALORES DEL GRADIENTE DE PHI -----------------*/
170
171     for(i=0;i<n_vc;i++)
172     {
173         ierr = MatGetRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
174         ierr = MatGetRow(matriz_phif,i,PETSC_NULL,PETSC_NULL,&phi_f0);CHKERRQ(ierr);
175
176         gradphivc[0]=((phi_f0[0]*areas[0])+(phi_f0[1]*areas[2])+(phi_f0[2]*areas[4]))/volumenvc[i];
177         gradphivc[1]=((phi_f0[0]*areas[1])+(phi_f0[1]*areas[3])+(phi_f0[2]*areas[5]))/volumenvc[i];
178
179         ierr = MatSetValues(*matriz_gradphivc,1,&i,2,col,gradphivc,INSERT_VALUES);CHKERRQ(ierr);
180
181         ierr = MatRestoreRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
182         ierr = MatRestoreRow(matriz_phif,i,PETSC_NULL,PETSC_NULL,&phi_f0);CHKERRQ(ierr);
183
184     }
185
186     ierr = VecRestoreArray(vec_volumen,&volumenvc);CHKERRQ(ierr);
187     ierr = VecRestoreArray(vec_phi,&phivc0);CHKERRQ(ierr);
188     ierr = VecRestoreArray(vec_condf,&condf);CHKERRQ(ierr);
189     ierr = VecRestoreArray(vec_front,&valueb);CHKERRQ(ierr);
190
191     ierr = MatAssemblyBegin(*matriz_gradphivc,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
192     ierr = MatAssemblyEnd(*matriz_gradphivc,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
193
194     //Se destruyen las matrices creadas que ya no se van a usar
```

```
195     ierr = MatDestroy(&matriz_phif);CHKERRQ(ierr);
196
197     return ierr;
198 }
```

10. ARCHIVO DE ENCABEZADO DE LA CLASE CONVECTIVO

```
1  //ARCHIVO DE ENCABEZADO DE LA CLASE CONVECTIVO: convectivo.h
2
3  #ifndef _CONVECTIVO_H_
4  #define _CONVECTIVO_H_
5
6  class convectivo
7  {
8      private:
9
10         PetscErrorCode ierr;                //Variable de error de Petsc
11         int i,j,k,ii;                       //Contadores
12         int tempora,temporal,tem,tempor;    //Variables de uso temporal
13
14         //Variables para establecer las columnas
15         PetscInt col[2];
16         PetscInt colconv[4];
17
18         //Elementos geométricos
19         int n_vc;
20         Mat matriz_areas;
21         const PetscScalar *areas;
22         Mat matriz_vecinos;
23         const PetscScalar *vecinos;
24         const PetscScalar *vvecinos;
25         Mat matriz_coord;
26         const PetscScalar *coordvc;
27         Mat matriz_centrocara;
28         const PetscScalar *centrocara;
29         Mat matriz_alfaf;
30         const PetscScalar *alfaf;
31         Mat matriz_vecpcent;
32         const PetscScalar *pcentp;
33         const PetscScalar *pcentf;
34
35         //Elementos de frontera
36         Vec vec_condf;
37         PetscScalar *condfu;
38         Vec vec_frontu;
39         PetscScalar *ufront;
40         Vec vec_frontv;
41         PetscScalar *vfront;
42         PetscScalar *u,*v;
43
44         //Variables para obtener los valores de las matrices y vectores dados como argumentos
45         velocidadf[2];
46         PetscScalar value[4];
47         PetscScalar convecfront;
48
49         //Objetos de la clase gradiente
50         gradiente gru;
51         gradiente grv;
52
53         //Variables para guardar y usar los valores tomados de los objetos de clase gradiente
54         Mat matriz_uf;
55         const PetscScalar *uf;
```

```
56          Mat matriz_vf;
57          const PetscScalar *vf;
58
59      public:
60
61          //Funciones
62          PetscErrorCode calcular_F(double densidad,geom gm,double *velocidad,Mat *matriz_convectivo,Vec *
vec_convecfront);
63          PetscErrorCode calcular_Ff(double densidad,geom gm,Vec vec_u,Vec vec_v,frontera fu,frontera fv,Mat *
matriz_convectivo,Vec *vec_convecfront);
64          PetscErrorCode calcular_Fsh(double densidad,geom gm,Mat *matriz_convectivo,Vec *vec_convecfront);
65
66
67  };
68
69  #endif // _CONVECTIVO_H_
```

11. ARCHIVO DE CÓDIGO FUENTE DE LA CLASE CONVECTIVO

```
1   //ARCHIVO DE CÓDIGO FUENTE DE LA CLASE CONVECTIVO: convectivo.cpp
2
3   #include <petsc.h>
4   #include <cmath>
5   #include "geom.h"
6   #include "frontera.h"
7   #include "gradiente.h"
8
9   /****************** TÉRMINOS CONVECTIVOS EN LA EC. DE DIFUSIÓN-CONVECCIÓN ******************/
10
11  PetscErrorCode convectivo::calcular_F(double densidad,geom gm,double *velocidad,Mat *matriz_convectivo,Vec
*vec_convecfront)
12  {
13      //Tomar los elementos geométricos de la clase geom
14      n_vc=gm.vc;
15      matriz_areas=gm.matriz_areas;
16      matriz_vecinos=gm.matriz_vecinos;
17
18      /* ------------------- CALCULAR Y GUARDAR LOS TÉRMINOS CONVECTIVOS ----------------------*/
19      col[0]=0;
20      col[1]=1;
21
22      for (i=0;i<n_vc;i++)
23      {
24          ierr = MatGetRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
25          ierr = MatGetRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
26
27          tempora=0;
28          temporal=1;
29
30          //Establecer las columnas de la matriz
31          colconv[0]=vecinos[0];
32          colconv[1]=vecinos[1];
33          colconv[2]=vecinos[2];
34
35          //Cálculo del término convectivo
36          for (j=0;j<3;j++)
37          {
38              value[j]=densidad*(velocidad[0]*areas[tempora]+velocidad[1]*areas[temporal]);
39              tempora=tempora+2;
40              temporal=temporal+2;
41          }
42
```

```
43          //Insertar los valores en la matriz y el vector(en caso de que sea frontera)
44          if (vecinos[2]>n_vc) //Si el vc tiene frontera
45          {
46              convecfront=value[2];
47              ierr = VecSetValues(vec_convecfront,1,&i,&convecfront,INSERT_VALUES);CHKERRQ(ierr);
48              ierr = MatSetValues(matriz_convectivo,1,&i,2,colconv,value,INSERT_VALUES);CHKERRQ(ierr);
49
50          }
51          else //Si el vc no tiene frontera
52          {
53              ierr = MatSetValues(matriz_convectivo,1,&i,3,colconv,value,INSERT_VALUES);CHKERRQ(ierr);
54          }
55
56          ierr = MatRestoreRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
57          ierr = MatRestoreRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
58      }
59
60      ierr = MatAssemblyBegin(matriz_convectivo,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
61      ierr = MatAssemblyEnd(matriz_convectivo,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
62
63      ierr = VecAssemblyBegin(vec_convecfront);CHKERRQ(ierr);
64      ierr = VecAssemblyEnd(vec_convecfront);CHKERRQ(ierr);
65
66      return ierr;
67
68  }
69
70  /********************** TÉRMINOS CONVECTIVOS EN EL PROBLEMA DE FLUJO **********************/
71
72   PetscErrorCode convectivo::calcular_F(double densidad,geom gm,Vec vec_u,Vec vec_v,frontera fu, frontera fv
Mat *matriz_convectivo,Vec *vec_convecfront)
73   {
74      //Tomar los elementos geométricos de la clase geom
75      n_vc=gm.vc;
76      matriz_areas=gm.matriz_areas;
77      matriz_vecinos=gm.matriz_vecinos;
78      matriz_alfaf=gm.matriz_alfaf;
79      matriz_vecpcent=gm.matriz_vecpcent;
80
81      //Tomar los elementos de la clase frontera
82      vec_condf=fu.vec_condf;
83      vec_frontu=fu.vec_front;
84      vec_frontv=fv.vec_front;
85
86      /* --------------------------- INICIALIZACIÓN DE VARIABLES --------------------------- */
87
88      col[0]=0;
89      col[1]=1;
90
91      //Matriz de u en las caras
92      ierr = MatCreate(PETSC_COMM_WORLD,&matriz_uf);CHKERRQ(ierr);
93      ierr = MatSetSizes(matriz_uf,PETSC_DECIDE,PETSC_DECIDE,n_vc,3);CHKERRQ(ierr);
94      ierr = MatSetFromOptions(matriz_uf);CHKERRQ(ierr);
95      ierr = MatSetUp(matriz_uf);CHKERRQ(ierr);
96
97      //Matriz de v en las caras
98      ierr = MatCreate(PETSC_COMM_WORLD,&matriz_vf);CHKERRQ(ierr);
99      ierr = MatSetSizes(matriz_vf,PETSC_DECIDE,PETSC_DECIDE,n_vc,3);CHKERRQ(ierr);
100     ierr = MatSetFromOptions(matriz_vf);CHKERRQ(ierr);
101     ierr = MatSetUp(matriz_vf);CHKERRQ(ierr);
102
103     /* --------------- CREAR LOS OBJETOS DE LA CLASE GRADIENTE PARA OBTENER ---------------*/
104     /* --------------------------- LAS VELOCIDADES EN LAS CARAS --------------------------- */
105
106     //Calculo de la velocidad u en las caras
107     gru.calcular_vcara(b,fu,vec_u,&matriz_uf);
```

```
108
109     //Calculo de la velocidad v en las caras
110     grv.calcular_vcara(b,fv,vec_v,&matriz_vf);
111
112     /* ------------------- CALCULAR Y GUARDAR LOS TÉRMINOS CONVECTIVOS ---------------------*/
113
114     for (i=0;i<n_vc;i++)
115     {
116         ierr = MatGetRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
117         ierr = MatGetRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
118         ierr = MatGetRow(matriz_uf,i,PETSC_NULL,PETSC_NULL,&uf);CHKERRQ(ierr);
119         ierr = MatGetRow(matriz_vf,i,PETSC_NULL,PETSC_NULL,&vf);CHKERRQ(ierr);
120
121         tempora=0;
122         temporal=1;
123
124         //Establecer las columnas de la matriz
125         colconv[0]=vecinos[0];
126         colconv[1]=vecinos[1];
127         colconv[2]=vecinos[2];
128
129         //Cálculo del término convectivo
130         for (j=0;j<3;j++)
131         {
132             value[j]=densidad*(uf[j]*areas[tempora]+vf[j]*areas[temporal]);
133             tempora=tempora+2;
134             temporal=temporal+2;
135         }
136
137         //Insertar los valores en la matriz y el vector(en caso de que sea frontera)
138         if (vecinos[2]>n_vc) //Si el vc tiene frontera
139         {
140             convecfront=value[2];
141             ierr = VecSetValues(*vec_convecfront,1,&i,&convecfront,INSERT_VALUES);CHKERRQ(ierr);
142             ierr = MatSetValues(*matriz_convectivo,1,&i,2,colconv,value,INSERT_VALUES);CHKERRQ(ierr);
143
144         }
145         else //Si el vc no tiene frontera
146         {
147             ierr = MatSetValues(*matriz_convectivo,1,&i,3,colconv,value,INSERT_VALUES);CHKERRQ(ierr);
148         }
149
150         ierr = MatRestoreRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
151         ierr = MatRestoreRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
152         ierr = MatGetRow(matriz_uf,i,PETSC_NULL,PETSC_NULL,&uf);CHKERRQ(ierr);
153         ierr = MatGetRow(matriz_vf,i,PETSC_NULL,PETSC_NULL,&vf);CHKERRQ(ierr);
154     }
155
156     ierr = MatAssemblyBegin(*matriz_convectivo,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
157     ierr = MatAssemblyEnd(*matriz_convectivo,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
158
159     ierr = VecAssemblyBegin(*vec_convecfront);CHKERRQ(ierr);
160     ierr = VecAssemblyEnd(*vec_convecfront);CHKERRQ(ierr);
161
162     //Se destruyen las matrices creadas que ya no se van a usar
163     ierr = MatDestroy(&matriz_uf);CHKERRQ(ierr);
164     ierr = MatDestroy(&matriz_vf);CHKERRQ(ierr);
165
166     return ierr;
167
168 }
169
170 /************************** TÉRMINOS CONVECTIVOS EN EL SMITH-HUTTON ***********************/
171
172 PetscErrorCode convectivo::calcular_Fsh(double densidad,geom gm,Mat *matriz_convectivo,Vec *vec_convecfron
```

```
)
{
    //Tomar los elementos geométricos de la clase geom
    n_vc=gm.vc;
    matriz_areas=gm.matriz_areas;
    matriz_vecinos=gm.matriz_vecinos;
    matriz_coord=gm.matriz_coord;
    matriz_centrocara=gm.matriz_centrocara;

    /* ------------------- CALCULAR Y GUARDAR LOS TÉRMINOS CONVECTIVOS ---------------------*/

    col[0]=0;
    col[1]=1;

    for (i=0;i<n_vc;i++)
    {
        ierr = MatGetRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
        ierr = MatGetRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
        ierr = MatGetRow(matriz_centrocara,i,PETSC_NULL,PETSC_NULL,&centrocara);CHKERRQ(ierr);

        tempora=0;
        temporal=1;

        //Establecer las columnas de la matriz
        colconv[3]=i;
        colconv[0]=vecinos[0];
        colconv[1]=vecinos[1];

        for (j=0;j<3;j++)
        {
            //Cálculo de la velocidad en las caras
            velocidadf[0]=2*centrocara[temporal]*(1-centrocara[tempora]*centrocara[tempora]);
            velocidadf[1]=-2*centrocara[tempora]*(1-centrocara[temporal]*centrocara[temporal]);

            //Cálculo del término convectivo
            value[j]=-densidad*(velocidad[0]*areas[tempora]+velocidad[1]*areas[temporal]);
            tempora=tempora+2;
            temporal=temporal+2;
        }

        //Insertar los valores en la matriz y el vector(en caso de que sea frontera)
        if (vecinos[2]>n_vc) //Si el vc tiene frontera
        {
                colconv[2]=i;
                convecfront=-value[2];
                value[2]=-(value[0]+value[1]-convecfront);
                ierr = MatSetValues(matriz_convectivo,1,&i,3,colconv,value,INSERT_VALUES);CHKERRQ(ierr);
                ierr = VecSetValues(vec_convecfront,1,&i,&convecfront,INSERT_VALUES);CHKERRQ(ierr);
        }
        else //Si el vc no tiene frontera
        {
            colconv[2]=vecinos[2];
            value[3]=-(value[0]+value[1]+value[2]);
            ierr = MatSetValues(matriz_convectivo,1,&i,4,colconv,value,INSERT_VALUES);CHKERRQ(ierr);
        }

        ierr = MatRestoreRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
        ierr = MatRestoreRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
        ierr = MatRestoreRow(matriz_centrocara,i,PETSC_NULL,PETSC_NULL,&centrocara);CHKERRQ(ierr);
    }

    ierr = MatAssemblyBegin(matriz_convectivo,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
    ierr = MatAssemblyEnd(matriz_convectivo,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);

```

```
236      ierr = VecAssemblyBegin(vec_convecfront);CHKERRQ(ierr);
237      ierr = VecAssemblyEnd(vec_convecfront);CHKERRQ(ierr);
238
239      return ierr;
240
241  }
```

12. ARCHIVO DE ENCABEZADO DE LA CLASE TERMINDEP

```
1   //ARCHIVO DE ENCABEZADO DE LA CLASE TERMINDEP
2   #ifndef _TERMINDEP_H_
3   #define _TERMINDEP_H_
4
5   class termindep
6   {
7       private:
8
9           PetscErrorCode ierr;                        //Variable de error de Petsc
10          int i,j,k,ii;                               //Contadores
11          int tem,tempor,tempora,ttemp,ttempo;        //Variables de uso temporal
12
13          //Elementos geométricos
14          int n_vc;
15          Vec vec_volumen;
16              PetscScalar *volumenvc;
17          Mat matriz_areas;
18              const PetscScalar *areas;
19          Mat matriz_vecinos;
20              const PetscScalar *vecinos;
21              const PetscScalar *vvecinos;
22          Mat matriz_alfaf;
23              const PetscScalar *alfaf;
24          Mat matriz_deltaxie;
25              const PetscScalar *deltaxies;
26           Mat matriz_vecppprima;
27               const PetscScalar *ppprima;
28               const PetscScalar *ffprima;
29           Mat matriz_coord;
30               const PetscScalar *coord;
31
32           //Elementos de frontera
33           Vec vec_condf;
34           Vec vec_condfp;
35               PetscScalar *condf;
36           Vec vec_front;
37           Vec vec_frontp;
38               PetscScalar *valueb;
39
40           //Variables para usar los valores de las matrices y vectores dados como argumentos
41           const PetscScalar *Df;
42           PetscScalar *difb;
43           PetscScalar *convb;
44           PetscScalar *phivc0;
45           const PetscScalar *upf;
46           const PetscScalar *vpf;
47
48           //Variables para los cálculos
49           PetscScalar areab,secondgrad,b,sumsecondgrad,sumtermp;
50           PetscScalar gradphivc[2],gradphicara[2];
51           const PetscScalar *gradphiactual;
```

```
52          const PetscScalar *gradphivecino;
53
54      public:
55
56          //Funciones
57          PetscErrorCode calcular_be(geom gm,frontera f,Mat matriz_difusivo,Vec vec_diffront,Vec
vec_convecfront,Mat matriz_gradphivc,Vec *vec_b);
58          PetscErrorCode calcular_bte(geom gm,frontera f,Mat matriz_difusivo,Vec vec_diffront,Vec
vec_convecfront,Vec vec_soluant,Mat matriz_gradphivc,Vec *vec_b);
59          PetscErrorCode calcular_bti(geom gm,frontera f,Mat matriz_difusivo,Vec vec_diffront,Vec vec_soluant,
Mat matriz_gradphivc,Vec vec_ap0,Vec *vec_b);
60          PetscErrorCode calcular_bp(geom gm,frontera fp,Mat matriz_difusivo,Vec vec_diffront,Mat matriz_gradp
,Mat matriz_upf,Mat matriz_vpf,double densidad,double deltat,Vec *vec_b);
61  };
62
```

13. ARCHIVO DE CÓDIGO FUENTE DE LA CLASE TERMINDEP

```
1  //ARCHIVO DE CÓDIGO FUENTE DE LA CLASE TERMIENDEP: termindep.cpp
2
3  #include <petsc.h>
4  #include <cmath>
5  #include "geom.h"
6  #include "frontera.h"
7
8  /********** CÁLCULO DE B PARA EL PROBLEMA DE DIFUSIÓN-CONVECCIÓN EN ESTADO ESTABLE *********/
9
10 PetscErrorCode termindep::calcular_be(geom gm,frontera f,Mat matriz_difusivo,Vec vec_diffront,Vec
vec_convecfront,Mat matriz_gradphivc,Vec *vec_b)
11 {
12     //Tomar los elementos geométricos de la clase geom
13     n_vc=gm.vc;
14     matriz_vecinos=gm.matriz_vecinos;
15     matriz_areas=gm.matriz_areas;
16     matriz_vecppprima=gm.matriz_vecppprima;
17
18     //Tomar los elementos de la clase frontera
19     vec_condf=f.vec_condf;
20     vec_front=f.vec_front;
21
22     /* ------------- CALCULAR Y GUARDAR EL VECTOR DE TERMINOS INDEPENDIENTES --------------*/
23
24     ierr = VecGetArray(vec_condf,&condf);CHKERRQ(ierr);
25     ierr = VecGetArray(vec_front,&valueb);CHKERRQ(ierr);
26     ierr = VecGetArray(vec_diffront,&difb);CHKERRQ(ierr);
27     ierr = VecGetArray(vec_convecfront,&convb);CHKERRQ(ierr);
28
29     for (i=0;i<n_vc;i++)
30     {
31
32         ierr = MatGetRow(matriz_gradphivc,i,PETSC_NULL,PETSC_NULL,&gradphiactual);CHKERRQ(ierr);
33         ierr = MatGetRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
34         ierr = MatGetRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
35         ierr = MatGetRow(matriz_vecppprima,i,PETSC_NULL,PETSC_NULL,&ppprima);CHKERRQ(ierr);
36         ierr = MatGetRow(matriz_difusivo,i,PETSC_NULL,PETSC_NULL,&Df);CHKERRQ(ierr);
37
38         tem=0;tempor=1;b=0;secondgrad=0;sumsecondgrad=0;
39
40         for (j=0;j<3;j++)
41         {
42             k=vecinos[j];
43
```

```
            if (k>n_vc)
            {
                if (condf[i]==1)    //Condición Dirichlet
                {
                    //Gradiente secundario en la frontera
                    secondgrad=-difb[i]*(gradphiactual[0]*ppprima[tem]+gradphiactual[1]*ppprima[tempor]);
                    //Termino independiente en la frontera
                    b=b+secondgrad+difb[i]*valueb[i]-convb[i]*valueb[i];
                }
                if (condf[i]==-1)   //Condición Neumann
                {
                    //Magnitud del área de la frontera
                    areab=sqrt(pow(areas[tem],2)+pow(areas[tempor],2));
                    //Termino independiente en la frontera
                    b=b+valueb[i]*areab;
                }
            }
            else
            {
                ierr = MatGetRow(matriz_vecinos,k,PETSC_NULL,PETSC_NULL,&vvecinos);CHKERRQ(ierr);
                for (ii=0;ii<3;ii++)
                {
                    {
                        ttemp=2*ii;
                        ttempo=2*ii+1;
                    }
                }
                ierr = MatRestoreRow(matriz_vecinos,k,PETSC_NULL,PETSC_NULL,&vvecinos);CHKERRQ(ierr);

                ierr = MatGetRow(matriz_gradphivc,k,PETSC_NULL,PETSC_NULL,&gradphivecino);CHKERRQ(ierr);
                ierr = MatGetRow(matriz_vecppprima,k,PETSC_NULL,PETSC_NULL,&ffprima);CHKERRQ(ierr);

                //Gradiente secundario
                secondgrad=Df[j]*((gradphivecino[0]*ffprima[ttemp]+gradphivecino[1]*ffprima[ttempo])-(
gradphiactual[0]*ppprima[tem]+gradphiactual[1]*ppprima[tempor]));
                //Sumatoria del gradiente secundario
                sumsecondgrad=sumsecondgrad+secondgrad;

                tem=tem+2;
                tempor=tempor+2;
            }
                ierr = MatRestoreRow(matriz_gradphivc,k,PETSC_NULL,PETSC_NULL,&gradphivecino);CHKERRQ(ierr
);
                ierr = MatRestoreRow(matriz_vecppprima,k,PETSC_NULL,PETSC_NULL,&ffprima);CHKERRQ(ierr);


        }
        //Término independiente
        b=b+sumsecondgrad;

        ierr = VecSetValues(*vec_b,1,&i,&b,INSERT_VALUES);CHKERRQ(ierr);

        ierr = MatRestoreRow(matriz_gradphivc,i,PETSC_NULL,PETSC_NULL,&gradphiactual);CHKERRQ(ierr);
        ierr = MatRestoreRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
        ierr = MatRestoreRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
        ierr = MatRestoreRow(matriz_vecppprima,i,PETSC_NULL,PETSC_NULL,&ppprima);CHKERRQ(ierr);
        ierr = MatRestoreRow(matriz_difusivo,i,PETSC_NULL,PETSC_NULL,&Df);CHKERRQ(ierr);
    }

    ierr = VecRestoreArray(vec_diffront,&difb);CHKERRQ(ierr);
    ierr = VecRestoreArray(vec_condf,&condf);CHKERRQ(ierr);
    ierr = VecRestoreArray(vec_front,&valueb);CHKERRQ(ierr);
    ierr = VecRestoreArray(vec_convecfront,&convb);CHKERRQ(ierr);

    ierr = VecAssemblyBegin(*vec_b);CHKERRQ(ierr);
    ierr = VecAssemblyEnd(*vec_b);CHKERRQ(ierr);

```

```
110     return ierr;
111 }
112
113 /** CÁLCULO DE B PARA EL PROBLEMA DE DIFUSIÓN-CONVECCIÓN EN ESTADO TRANSITORIO (EXPLÍCITO) **/
114
115 PetscErrorCode termindep::calcular_bct(geom gm,frontera f,Mat matriz difusivo,Vec vec_diffront,Vec
vec_convecfront,Vec vec_soluant,Mat matriz_gradphivc, Vec *vec_b)
116 {
117     //Tomar los elementos geométricos de la clase geom
118     n_vc=gm.vc;
119     matriz_vecinos=gm.matriz_vecinos;
120     matriz_areas=gm.matriz_areas;
121     matriz_vecpf=gm.matriz_vecpf;
122     matriz_vecppprima=gm.matriz_vecppprima;
123     matriz_coord=gm.matriz_coord;
124
125     //Tomar los elementos de la clase frontera
126     vec_condf=f.vec_condf;
127     vec_front=f.vec_front;
128
129     /* ------------- CALCULAR Y GUARDAR EL VECTOR DE TERMINOS INDEPENDIENTES --------------*/
130
131     ierr = VecGetArray(vec_soluant,&phivc0);CHKERRQ(ierr);
132     ierr = VecGetArray(vec_diffront,&difb);CHKERRQ(ierr);
133     ierr = VecGetArray(vec_condf,&condf);CHKERRQ(ierr);
134     ierr = VecGetArray(vec_front,&valueb);CHKERRQ(ierr);
135     ierr = VecGetArray(vec_convecfront,&convb);CHKERRQ(ierr);
136
137     for (i=0;i<n_vc;i++)
138     {
139
140         ierr = MatGetRow(matriz_gradphivc,i,PETSC_NULL,PETSC_NULL,&gradphiactual);CHKERRQ(ierr);
141         ierr = MatGetRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
142         ierr = MatGetRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
143         ierr = MatGetRow(matriz_vecpf,i,PETSC_NULL,PETSC_NULL,&deltaxies);CHKERRQ(ierr);
144         ierr = MatGetRow(matriz_vecppprima,i,PETSC_NULL,PETSC_NULL,&ppprima);CHKERRQ(ierr);
145         ierr = MatGetRow(matriz_coord,i,PETSC_NULL,PETSC_NULL,&coord);CHKERRQ(ierr);
146
147         tem=0;tempor=1;b=0;secondgrad=0;sumsecondgrad=0;
148
149         for (j=0;j<3;j++)
150         {
151             k=vecinos[j];
152
153             if (k>n_vc)
154             {
155                 if (condf[i]==1)    //Condición Dirichlet
156                 {
157                     //Gradiente secundario en la frontera
158                     secondgrad=-difb[i]*(gradphiactual[0]*ppprima[tem]+gradphiactual[1]*ppprima[tempor]);
159                     //Término independiente en la frontera
160                     b=b+secondgrad+difb[i]*(valueb[i]-phivc0[i])-convb[i]*(valueb[i]-phivc0[i]);
161                 }
162                 if (condf[i]==-1)   //Condición Neumann
163                 {
164                     //Magnitud del área en la frontera
165                     areab=sqrt(pow(areas[tem],2)+pow(areas[tempor],2));
166                     //Término independiente en la frontera
167                     b=b+valueb[i]*areab;
168                 }
169             }
170
171             else
172             {
173                 ierr = MatGetRow(matriz_vecinos,k,PETSC_NULL,PETSC_NULL,&vvecinos);CHKERRQ(ierr);
```

```
171             else
172             {
173                 ierr = MatGetRow(matriz_vecinos,k,PETSC_NULL,PETSC_NULL,&vvecinos);CHKERRQ(ierr);
174                 for (ii=0;ii<3;ii++)
175                 {
176                     if (i==vvecinos[ii])
177                     {
178                         ttemp=2*ii;
179                         ttempo=2*ii+1;
180                     }
181                 }
182                 ierr = MatRestoreRow(matriz_vecinos,k,PETSC_NULL,PETSC_NULL,&vvecinos);CHKERRQ(ierr);
183 
184                 ierr = MatGetRow(matriz_gradphivc,k,PETSC_NULL,PETSC_NULL,&gradphivecino);CHKERRQ(ierr);
185                 ierr = MatGetRow(matriz_vecppprima,k,PETSC_NULL,PETSC_NULL,&ffprima);CHKERRQ(ierr);
186 
187                 //Gradiente secundario
188                 secondgrad=Df[j]*((gradphivecino[0]*ffprima[ttemp]+gradphivecino[1]*ffprima[ttempo])-(
gradphiactual[0]*ppprima[tem]+gradphiactual[1]*ppprima[tempor]));
189                 //Sumatoria del gradiente secundario
190                 sumsecondgrad=sumsecondgrad+secondgrad;
191 
192                 tem=tem+2;
193                 tempor=tempor+2;
194 
195                 ierr = MatRestoreRow(matriz_gradphivc,k,PETSC_NULL,PETSC_NULL,&gradphivecino);CHKERRQ(ierr
);
196                 ierr = MatRestoreRow(matriz_vecppprima,k,PETSC_NULL,PETSC_NULL,&ffprima);CHKERRQ(ierr);
197             }
198 
199         }
200         //Término independiente
201         b=b-sumsecondgrad;
202 
203         ierr = VecSetValues(vec_b,1,&i,&b,INSERT_VALUES);CHKERRQ(ierr);
204 
205         ierr = MatRestoreRow(matriz_gradphivc,i,PETSC_NULL,PETSC_NULL,&gradphiactual);CHKERRQ(ierr);
206         ierr = MatRestoreRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
207         ierr = MatRestoreRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
208         ierr = MatRestoreRow(matriz_vecpf,i,PETSC_NULL,PETSC_NULL,&deltaxies);CHKERRQ(ierr);
209         ierr = MatRestoreRow(matriz_vecppprima,i,PETSC_NULL,PETSC_NULL,&ppprima);CHKERRQ(ierr);
210         ierr = MatRestoreRow(matriz_coord,i,PETSC_NULL,PETSC_NULL,&coord);CHKERRQ(ierr);
211     }
212 
213 
214     ierr = VecRestoreArray(vec_gamma,&gamma);CHKERRQ(ierr);
215     ierr = VecRestoreArray(vec_soluant,&phivc0);CHKERRQ(ierr);
216     ierr = VecRestoreArray(vec_diffront,&difb);CHKERRQ(ierr);
217     ierr = VecRestoreArray(vec_condf,&condf);CHKERRQ(ierr);
218     ierr = VecRestoreArray(vec_front,&valueb);CHKERRQ(ierr);
219     ierr = VecRestoreArray(vec_convecfront,&convb);CHKERRQ(ierr);
220 
221     ierr = VecAssemblyBegin(*vec_b);CHKERRQ(ierr);
222     ierr = VecAssemblyEnd(*vec_b);CHKERRQ(ierr);
223 
224 
225     return ierr;
226 }
227 
228 /** CÁLCULO DE B PARA EL PROBLEMA DE DIFUSIÓN-CONVECCIÓN EN ESTADO TRANSITORIO (IMPLÍCITO) **/
229 
230 PetscErrorCode termindep::calcular_bci(geom gm,frontera f,Mat matriz_difusivo,Vec vec_diffront, Vec
vec_soluant,Mat matriz_gradphivc,Vec vec_ap0,Vec *vec_b)
231 {
232     //Tomar los elementos de la clase geom
233     n vc=b.vc;
```

```
233     n_vc=b.vc;
234     matriz_vecinos=gm.matriz_vecinos;
235     matriz_areas=gm.matriz_areas;
236     matriz_vecpf=gm.matriz_vecpf;
237     matriz_vecppprima=gm.matriz_vecppprima;
238     matriz_coord=gm.matriz_coord;
239
240     //Tomar los elementos de la clase frontera
241     vec_condf=f.vec_condf;
242     vec_front=f.vec_front;
243
244     /* ------------- CÁLCULAR Y GUARDAR EL VECTOR DE TERMINOS INDEPENDIENTES --------------*/
245
246     ierr = VecGetArray(vec_diffront,&difb);CHKERRQ(ierr);
247     ierr = VecGetArray(vec_gamma,&gamma);CHKERRQ(ierr);
248     ierr = VecGetArray(vec_condf,&condf);CHKERRQ(ierr);
249     ierr = VecGetArray(vec_front,&valueb);CHKERRQ(ierr);
250
251     for (i=0;i<n_vc;i++)
252     {
253
254         ierr = MatGetRow(matriz_gradphivc,i,PETSC_NULL,PETSC_NULL,&gradphiactual);CHKERRQ(ierr);
255         ierr = MatGetRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
256         ierr = MatGetRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
257         ierr = MatGetRow(matriz_vecpf,i,PETSC_NULL,PETSC_NULL,&deltaxies);CHKERRQ(ierr);
258         ierr = MatGetRow(matriz_vecppprima,i,PETSC_NULL,PETSC_NULL,&ppprima);CHKERRQ(ierr);
259         ierr = MatGetRow(matriz_coord,i,PETSC_NULL,PETSC_NULL,&coord);CHKERRQ(ierr);
260
261         tem=0;tempor=1;b=0;secondgrad=0;sumsecondgrad=0;
262
263         for (j=0;j<3;j++)
264         {
265             k=vecinos[j];
266
267             if (k>n_vc)
268             {
269                 if (condf[i]==1) //Frontera Dirichlet
270                 {
271                     //Gradiente secundario en la frontera
272                     secondgrad=-difb[i]*(gradphiactual[0]*ppprima[tem]+gradphiactual[1]*ppprima[tempor]);
273                     //Término independiente en la frontera
274                     b=b+secondgrad+difb[i]*valueb[i]-convb[i]*valueb[i];
275                 }
276                 if (condf[i]==-1) //Frontera Neumann
277                 {
278                     //Magnitud del área en la frontera
279                     areab=sqrt(pow(areas[tem],2)+pow(areas[tempor],2));
280                     //Término independiente en la frontera
281                     b=b+valueb[i]*areab;
282                 }
283             }
284             else
285             {
286                 ierr = MatGetRow(matriz_vecinos,k,PETSC_NULL,PETSC_NULL,&vvecinos);CHKERRQ(ierr);
287                 for (ii=0;ii<3;ii++)
288                 {
289                     if (i==vvecinos[ii])
290                     {
291                         ttemp=2*ii;
292                         ttempo=2*ii+1;
293                     }
294                 }
295                 ierr = MatRestoreRow(matriz_vecinos,k,PETSC_NULL,PETSC_NULL,&vvecinos);CHKERRQ(ierr);
296
297                 ierr = MatGetRow(matriz_gradphivc,k,PETSC_NULL,PETSC_NULL,&gradphivecino);CHKERRQ(ierr);
298                 ierr = MatGetRow(matriz_vecppprima,k,PETSC_NULL,PETSC_NULL,&ffprima);CHKERRQ(ierr);
```

```
299
300                     //Gradiente secundario
301                     secondgrad=Df[j]*((gradphivecino[0]*ffprima[ttemp]+gradphivecino[1]*ffprima[ttempo])-(
gradphiactual[0]*ppprima[tem]+gradphiactual[1]*ppprima[tempor]));
302                     //Sumatoria del gradiente secundario
303                     sumsecondgrad=sumsecondgrad+secondgrad;
304
305                     ierr = MatRestoreRow(matriz_gradphivc,k,PETSC_NULL,PETSC_NULL,&gradphivecino);CHKERRQ(ierr
);
306                     ierr = MatRestoreRow(matriz_vecppprima,k,PETSC_NULL,PETSC_NULL,&ffprima);CHKERRQ(ierr);
307
308                     tem=tem+2;
309                     tempor=tempor+2;
310                 }
311
312             }
313             //Termino independiente
314             b=b-sumsecondgrad+coef_ap0[i]*phip0[i];
315
316             ierr = VecSetValues(vec_b,1,&i,&b,INSERT_VALUES);CHKERRQ(ierr);
317
318             ierr = MatRestoreRow(matriz_gradphivc,i,PETSC_NULL,PETSC_NULL,&gradphiactual);CHKERRQ(ierr);
319             ierr = MatRestoreRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
320             ierr = MatRestoreRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
321             ierr = MatRestoreRow(matriz_vecpf,i,PETSC_NULL,PETSC_NULL,&deltaxies);CHKERRQ(ierr);
322             ierr = MatRestoreRow(matriz_vecppprima,i,PETSC_NULL,PETSC_NULL,&ppprima);CHKERRQ(ierr);
323         }
324
325
326         ierr = VecRestoreArray(vec_gamma,&gamma);CHKERRQ(ierr);
327         ierr = VecRestoreArray(vec_diffront,&difb);CHKERRQ(ierr);
328         ierr = VecRestoreArray(vec_condf,&condf);CHKERRQ(ierr);
329         ierr = VecRestoreArray(vec_front,&valueb);CHKERRQ(ierr);
330
331         ierr = VecAssemblyBegin(vec_b);CHKERRQ(ierr);
332         ierr = VecAssemblyEnd(vec_b);CHKERRQ(ierr);
333
334         return ierr;
335     }
336
337     /************************ CÁLCULO DE B PARA LA ECUACIÓN DE MOMENTO ************************/
338
339     PetscErrorCode termindep::calcular_bp(geom gm,frontera fp,Mat matriz_difusivo,Vec vec_diffront,Mat
matriz_gradp,Mat matriz_upf,Mat matriz_vpf,double densidad,double deltat,Vec *vec_b)
340     {
341         //Tomar los elementos de la clase geom
342         n_vc=gm.vc;
343         matriz_vecinos=gm.matriz_vecinos;
344         matriz_areas=gm.matriz_areas;
345         matriz_vecppprima=gm.matriz_vecppprima;
346
347         //Tomar los elementos de la clase frontera
348         vec_condf=fp.vec_condf;
349         vec_front=fp.vec_front;
350
351         ierr = VecGetArray(vec_diffront,&difb);CHKERRQ(ierr);
352         ierr = VecGetArray(vec_condf,&condf);CHKERRQ(ierr);
353         ierr = VecGetArray(vec_front,&valueb);CHKERRQ(ierr);
354
355         /* ------------- CÁLCULAR Y GUARDAR EL VECTOR DE TERMINOS INDEPENDIENTES --------------*/
356
357         for (i=0;i<n_vc;i++)
358         {
359             ierr = MatGetRow(matriz_gradp,i,PETSC_NULL,PETSC_NULL,&gradphiactual);CHKERRQ(ierr);
360             ierr = MatGetRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
```

```
361         ierr = MatGetRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
362         ierr = MatGetRow(matriz_vecppprima,i,PETSC_NULL,PETSC_NULL,&ppprima);CHKERRQ(ierr);
363         ierr = MatGetRow(matriz_difusivo,i,PETSC_NULL,PETSC_NULL,&Df);CHKERRQ(ierr);
364         ierr = MatGetRow(matriz_upf,i,PETSC_NULL,PETSC_NULL,&upf);CHKERRQ(ierr);
365         ierr = MatGetRow(matriz_vpf,i,PETSC_NULL,PETSC_NULL,&vpf);CHKERRQ(ierr);
366
367         tem=0;tempor=1;b=0;secondgrad=0;sumsecondgrad=0;sumtermp=0.0;
368
369         for (j=0;j<3;j++)
370         {
371             k=vecinos[j];
372
373             if (k>n_vc)
374             {
375                 if (condf[i]==1)    //Condición Dirichlet
376                 {
377                     //Gradiente secundario en la frontera
378                     secondgrad=-difb[i]*(gradphiactual[0]*ppprima[tem]+gradphiactual[1]*ppprima[tempor]);
379                     //Termino independiente en la frontera
380                     b=b+secondgrad-difb[i]*valueb[i];
381                 }
382                 if (condf[i]==-1)   //Condición Newman
383                 {
384                     //Magnitud del área de la frontera
385                     areab=sqrt(pow(areas[tem],2)+pow(areas[tempor],2));
386                     //Termino independiente en la frontera
387                     b=b-valueb[i]*areab;
388             }
389             else
390             {
391                 ierr = MatGetRow(matriz_vecinos,k,PETSC_NULL,PETSC_NULL,&vvecinos);CHKERRQ(ierr);
392                 for (ii=0;ii<3;ii++)
393                 {
394                     if (i==vvecinos[ii])
395                     {
396                         ttemp=2*ii;
397                         ttempo=2*ii+1;
398                     }
399                 }
400                 ierr = MatRestoreRow(matriz_vecinos,k,PETSC_NULL,PETSC_NULL,&vvecinos);CHKERRQ(ierr);
401
402                 ierr = MatGetRow(matriz_gradp,k,PETSC_NULL,PETSC_NULL,&gradphivecino);CHKERRQ(ierr);
403                 ierr = MatGetRow(matriz_vecppprima,k,PETSC_NULL,PETSC_NULL,&ffprima);CHKERRQ(ierr);
404
405                 //Gradiente secundario
406                 secondgrad=Df[j]*((gradphivecino[0]*ffprima[ttemp]+gradphivecino[1]*ffprima[ttempo])-(
gradphiactual[0]*ppprima[tem]+gradphiactual[1]*ppprima[tempor]));
407                 //Sumatoria del gradiente secundario
408                 sumsecondgrad=sumsecondgrad+secondgrad;
409
410                 ierr = MatRestoreRow(matriz_gradp,k,PETSC_NULL,PETSC_NULL,&gradphivecino);CHKERRQ(ierr);
411                 ierr = MatRestoreRow(matriz_vecppprima,k,PETSC_NULL,PETSC_NULL,&ffprima);CHKERRQ(ierr);
412             }
413             //Sumatoria de la divergencia la velocidad predictora
414             sumtermp=sumtermp+upf[j]*areas[tem]+vpf[j]*areas[tempor];
415
416             tem=tem+2;
417             tempor=tempor+2;
418         }
419         //Término independiente
420         b=b-sumsecondgrad+sumtermp*densidad/deltat;
421
422         ierr = VecSetValues(*vec_b,1,&i,&b,INSERT_VALUES);CHKERRQ(ierr);
423
424         ierr = MatRestoreRow(matriz_gradp,i,PETSC_NULL,PETSC_NULL,&gradphiactual);CHKERRQ(ierr);
```

```
425         ierr = MatRestoreRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
426         ierr = MatRestoreRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
427         ierr = MatRestoreRow(matriz_vecppprima,i,PETSC_NULL,PETSC_NULL,&ppprima);CHKERRQ(ierr);
428         ierr = MatRestoreRow(matriz_difusivo,i,PETSC_NULL,PETSC_NULL,&Df);CHKERRQ(ierr);
429         ierr = MatRestoreRow(matriz_upf,i,PETSC_NULL,PETSC_NULL,&upf);CHKERRQ(ierr);
430         ierr = MatRestoreRow(matriz_vpf,i,PETSC_NULL,PETSC_NULL,&vpf);CHKERRQ(ierr);
431     }
432
433     ierr = VecGetArray(vec_diffront,&difb);CHKERRQ(ierr);
434     ierr = VecGetArray(vec_condf,&condf);CHKERRQ(ierr);
435     ierr = VecGetArray(vec_front,&valueb);CHKERRQ(ierr);
436
437     ierr = VecAssemblyBegin(*vec_b);CHKERRQ(ierr);
438     ierr = VecAssemblyEnd(*vec_b);CHKERRQ(ierr);
439
440     return ierr;
441 }
```

14. ARCHIVO DE ENCABEZADO DE LA CLASE RESULTADOS

```
1  //ARCHIVO DE ENCABEZADO DE LA CLASE RESULTADOS: resultados.h
2
3  #ifndef _RESULTADOS_H_
4  #define _RESULTADOS_H_
5
6  class resultados
7  {
8      private:
9          PetscErrorCode ierr;     //Variable de error de Petsc
10         int i;                   //Contadores
11
12         //Elementos geométricos
13         double vertices,n_vc;
14         Mat matriz_n;
15         const PetscScalar *vert;
16         Mat matriz_vc;
17         const PetscScalar *vc;
18
19         //Variables para tomar los valores de las soluciones
20         PetscScalar *sol;
21         PetscScalar *solu;
22         PetscScalar *solv;
23         PetscScalar *solp;
24
25         public:
26
27         //Funciones
28         PetscErrorCode imprimir(geom gm,Vec vec_solucionu,Vec vec_solucionv,Vec vec_solucionp);
29 };
30
31 #endif // _RESULTADOS_H_
```

15. ARCHIVO DE CÓDIGO FUENTE DE LA CLASE RESULTADOS

```
1  //ARCHIVO DE CÓDIGO FUENTE DE LA CLASE RESULTADOS: resultados.cpp
2
3  #include <petsc.h>
4  #include <fstream>
5  #include <cmath>
6  #include "geom.h"
```

```
7
8  /************ FUNCIÓN IMPRIMIR LA SOLUCIÓN DEL PROBLEMA EN UN ARCHIVO DE TEXTO *************/
9
10 PetscErrorCode resultados::imprimir(geom gm,Vec vec_solucionu,Vec vec_solucionv,Vec vec_solucionp)
11 {
12     //Tomar los elementos geométricos de la clase geom
13     vertices=gm.vertices;
14     n_vc=gm.vc;
15     matriz_n=gm.matriz_n;
16     matriz_vc=gm.matriz_vc;
17
18     char nombre[100]="resultado.vtk";
19
20     //Estructura del archivo .vtk
21     std::ofstream f(nombre);
22
23     f << "# vtk DataFile Version 3.0" << std::endl;
24     f << "vtk output" << std::endl;
25     f << "ASCII" << std::endl;
26     f << "DATASET UNSTRUCTURED_GRID" << std::endl;
27     f << "POINTS " << vertices << " float" << std::endl;
28
29     for (i=1; i<vertices+1; i++)
30     {
31         ierr = MatGetRow(matriz_n,i,PETSC_NULL,PETSC_NULL,&vert);CHKERRQ(ierr);
32         f << vert[0] <<" "<< vert[1] <<" "<< 0 << std::endl;
33         ierr = MatRestoreRow(matriz_n,i,PETSC_NULL,PETSC_NULL,&vert);CHKERRQ(ierr);
34     }
35
36     f << std::endl << "CELLS " << n_vc << " " << n_vc*4 << std::endl;
37
38     for (i=0; i<n_vc; i++)
39     {
40         ierr = MatGetRow(matriz_vc,i,PETSC_NULL,PETSC_NULL,&vc);CHKERRQ(ierr);
41         f << "3 " << vc[0]-1 <<" "<< vc[1]-1 <<" "<< vc[2]-1 << std::endl;
42         ierr = MatRestoreRow(matriz_vc,i,PETSC_NULL,PETSC_NULL,&vc);CHKERRQ(ierr);
43     }
44
45     f << std::endl << "CELL_TYPES " << n_vc << std::endl;
46
47     for (i=0; i<n_vc; i++)
48     {
49         f << "5" << std::endl;
50     }
51
52     //Solución vectorial
53     f << std::endl << "CELL_DATA " << n_vc << std::endl;
54     f << "VECTORS cell_velocity double" << std::endl;
55
56     ierr = VecGetArray(vec_solucionu,&solu);CHKERRQ(ierr);
57     ierr = VecGetArray(vec_solucionv,&solv);CHKERRQ(ierr);
58     for (i=0; i<n_vc; i++)
59     {
60         f << solu[i] << " " << solv[i] << " 0" << std::endl;
61     }
62     ierr = VecRestoreArray(vec_solucionu,&solu);CHKERRQ(ierr);
63     ierr = VecRestoreArray(vec_solucionv,&solv);CHKERRQ(ierr);
64
65     //Solución escalar
66     f << std::endl << "SCALARS cell_scalars double 1" << std::endl;
67     f << "LOOKUP_TABLE default" << std::endl;
68
69     ierr = VecGetArray(vec_solucionp,&solp);CHKERRQ(ierr);
70     for (i=0; i<n_vc; i++)
71     {
72         f << solp[i] << std::endl;
```

```
    }
    ierr = VecRestoreArray(vec_solucionp,&solp);CHKERRQ(ierr);

    f.close();

    return ierr;

```

16. ARCHIVO DE ENCABEZADO DE LA CLASE SOLESTABLE

```
//ARCHIVO DE ENCABEZADO DE LA CLASE SOLUCION ESTABLE: solestable.h

#ifndef _SOLESTABLE_H_
#define _SOLESTABLE_H_

class solestable
{
    private:

        PetscErrorCode ierr;                        //Variable de error de Petsc
        int i,j,k,ii;                               //Contadores
        int tem,tempor,tempora,ttemp,ttempo;        //Variables de uso temporal
        double temconst

        //Variables para establecer las columnas
        PetscInt *cols;
        PetscInt colcoef[4];poral;

        //Elementos geométricos
        int n_vc;
        Vec vec_volumen;
        PetscScalar *volumenvc;
        Mat matriz_areas;
        const PetscScalar *areas;
        Mat matriz alfaf;
        const PetscScalar *alfaf;
        Mat matriz_deltaxie;
        const PetscScalar *deltaxies;
        Mat matriz_vecppprima;
        const PetscScalar *ppprima;
        const PetscScalar *ffprima;
        Mat matriz_vecinos;
        const PetscScalar *vecinos;
        const PetscScalar *vvecinos;
        Mat matriz_coord;
        const PetscScalar *coord;

        //Elementos frontera
        Vec vec_front;
            PetscScalar *valueb;
        Vec vec_condf;
            PetscScalar *condf;

        //Objeto de la clase difusivo
        difusivo df;
        //Variables para calcular y usar la matriz de términos difusivos
        Mat matriz_difusivo;
        const PetscScalar *Df;
        //Variables para calcular y usar el vector de términos difusivos en las fronteras
        Vec vec_diffront;
        PetscScalar *difb;
```

```
52
53          //Objeto de la clase convectivo
54          convectivo cv;
55          //Variables para calcular y usar la matriz de términos convectivos
56          Mat matriz_convectivo;
57          const PetscScalar *Ff;
58          //Variables para calcular y usar el vector de términos convectivos en las fronteras
59          Vec vec_convecfront;
60          PetscScalar *convb;
61
62          //Objeto de la clase gradiente
63          gradiente grad;
64          //Variables para calcular la matriz de gradientes
65          Mat matriz_gradphivc;
66
67          //Objeto de la clase termindep
68          termindep ib;
69          //Variables para calcular el vector de términos independientes
70          Vec vec_b;
71
72          //Objeto de la clase resultados
73          resultados rta;
74
75          //Vector solucion actual
76          Vec vec_solucion;
77          PetscScalar *phivc0;
78          //Vector solucion de la iteracion anterior
79          Vec vec_soluant;
80          PetscScalar *phivc;
81          //Matriz de coeficientes
82          Mat matriz_coef;
83          const PetscScalar *coeficiente;
84          //Entorno KSP
85          KSP ksp;
86          PC pc;
87
88          //Variables de cálculo
89          PetscScalar value[4];
90          PetscScalar phi0;
91          int error,niter;
92
93      public:
94
95          //Funciones
96          PetscErrorCode calcular_Se(double densidad,double gamma,geom b,frontera f)
97
98  };
99
100 #endif // _SOLUCION_H_
```

17. ARCHIVO DE CÓDIGO FUENTE DE LA CLASE SOLESTABLE

```
1  //ARCHIVO DE CÓDIGO FUENTE DE LA CLASE SOLUCIÓN ESTABLE: solestable.cpp
2
3  #include <petsc.h>
4  #include <cmath>
5  #include "geom.h"
6  #include "resultados.h"
7  #include "frontera.h"
8  #include "gradiente.h"
9  #include "difusivo.h"
```

```
10 #include "convectivo.h"
11 #include "termindep.h"
12
13 PetscErrorCode solestable::calcular_Se(double densidad,double gamma,geom gm,fontera f)
14 {
15     //Tomar los elementos de la clase geom
16     n_vc=gm.vc;
17     vec_volumen=gm.vec_volumen;
18     matriz_areas=gm.matriz_areas;
19     matriz_alfaf=gm.matriz_alfaf;
20     matriz_vecpf=gm.matriz_vecpf;
21     matriz_vecppprima=gm.matriz_vecppprima;
22     matriz_vecinos=gm.matriz_vecinos;
23     matriz_coord=gm.matriz_coord;
24
25     //Tomar los elementos de la clase fontera
26     vec_font=f.vec_font;
27     vec_condf=f.vec_condf;
28
29     /* -----------------------  INICIALIZACIÓN DE VARIABLES ----------------------- */
30
31     PetscInt col[n_vc];
32     for (i=0;i<n_vc;i++)
33     {
34         col[i]=i;
35     }
36
37     //Matriz de terminos difusivos
38     ierr = MatCreate(PETSC_COMM_WORLD,&matriz_difusivo);CHKERRQ(ierr);
39     ierr = MatSetSizes(matriz_difusivo,PETSC_DECIDE,PETSC_DECIDE,n_vc,n_vc);CHKERRQ(ierr);
40     ierr = MatSetFromOptions(matriz_difusivo);CHKERRQ(ierr);
41     ierr = MatSetUp(matriz_difusivo);CHKERRQ(ierr);
42
43     //Vector de terminos difusivos en las fronteras
44     ierr = VecCreate(PETSC_COMM_WORLD,&vec_diffront);CHKERRQ(ierr);
45     ierr = VecSetSizes(vec_diffront,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
46     ierr = VecSetFromOptions(vec_diffront);CHKERRQ(ierr);
47
48     //Matriz de terminos convectivos
49     ierr = MatCreate(PETSC_COMM_WORLD,&matriz_convectivo);CHKERRQ(ierr);
50     ierr = MatSetSizes(matriz_convectivo,PETSC_DECIDE,PETSC_DECIDE,n_vc,n_vc);CHKERRQ(ierr);
51     ierr = MatSetFromOptions(matriz_convectivo);CHKERRQ(ierr);
52     ierr = MatSetUp(matriz_convectivo);CHKERRQ(ierr);
53
54     //Vector de terminos convectivos en las frontera
55     ierr = VecCreate(PETSC_COMM_WORLD,&vec_convecfront);CHKERRQ(ierr);
56     ierr = VecSetSizes(vec_convecfront,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
57     ierr = VecSetFromOptions(vec_convecfront);CHKERRQ(ierr);
58
59     //Matriz de coeficientes
60     ierr = MatCreate(PETSC_COMM_WORLD,&matriz_coef);CHKERRQ(ierr);
61     ierr = MatSetSizes(matriz_coef,PETSC_DECIDE,PETSC_DECIDE,n_vc,n_vc);CHKERRQ(ierr);
62     ierr = MatSetfomOptions(matriz_coef);CHKERRQ(ierr);
63     ierr = MatSetUp(matriz_coef);CHKERRQ(ierr);
64
65     //Matriz del gradiente de phi
66     ierr = MatCreate(PETSC_COMM_WORLD,&matriz_gradphivc);CHKERRQ(ierr);
67     ierr = MatSetSizes(matriz_gradphivc,PETSC_DECIDE,PETSC_DECIDE,n_vc,2);CHKERRQ(ierr);
68     ierr = MatSetFromOptions(matriz_gradphivc);CHKERRQ(ierr);
69     ierr = MatSetUp(matriz_gradphivc);CHKERRQ(ierr);
70
71     //Vector de terminos independientes
72     ierr = VecCreate(PETSC_COMM_WORLD,&vec_b);CHKERRQ(ierr);
73     ierr = VecSetSizes(vec_b,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
74     ierr = VecSetFromOptions(vec_b);CHKERRQ(ierr);
75
76     //Vector solucion
```

```
77      ierr = VecCreate(PETSC_COMM_WORLD,&vec_solucion);CHKERRQ(ierr);
78      ierr = VecSetSizes(vec_solucion,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
79      ierr = VecSetfomOptions(vec_solucion);CHKERRQ(ierr);
80
81      //Vector solucion de la iteracion anterior
82      ierr = VecCreate(PETSC_COMM_WORLD,&vec_soluant);CHKERRQ(ierr);
83      ierr = VecSetSizes(vec_soluant,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
84      ierr = VecSetfomOptions(vec_soluant);CHKERRQ(ierr);
85      ierr = VecSet(vec_soluant,100);CHKERRQ(ierr);
86      ierr = VecAssemblyBegin(vec_soluant);CHKERRQ(ierr);
87      ierr = VecAssemblyEnd(vec_soluant);CHKERRQ(ierr);
88
89      /*------------------------- CÁLCULO DE TÉRMINOS DIFUSIVOS -------------------------*/
90
91      df.calcular_D(gamma,b,Mat *matriz_difusivo,Vec *vec_diffont);
92
93      /*------------------------ CÁLCULO DE TÉRMINOS CONVECTIVOS ------------------------*/
94
95      cv.calcular_F(densidad,b,Mat *matriz_convectivo,Vec *vec_convecfront);
96
97      /*------------------------ CREAR LA MATRIZ DE COEFICIENTES ------------------------*/
98
99      ierr = VecGetArray(vec_diffront,&difb);CHKERRQ(ierr);
100     ierr = VecGetArray(vec_convecfront,&convb);CHKERRQ(ierr);
101
102     for (i=0;i<n_vc;i++)
103     {
104         ierr = MatGetRow(matriz_difusivo,i,PETSC_NULL,&cols,&Df);CHKERRQ(ierr);
105         ierr = MatGetRow(matriz_convectivo,i,PETSC_NULL,PETSC_NULL,&Ff);CHKERRQ(ierr);
106         ierr = MatGetRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
107
108         colcoef[0]=i;colcoef[1]=cols[0];colcoef[2]=cols[1];colcoef[3]=cols[2];
109
110         if(vecinos[2]>n_vc)
111         {
112             value[1]=Df[0]-MAX(Ff[0],0);
113             value[2]=Df[1]-MAX(Ff[1],0);
114             if(condf[i]==1)
115             {value[0]=-(value[1]+value[2]+difb[i]-convb[i]);}
116             if(condf[i]==-1)
117             {value[0]=-(value[1]+value[2]-convb[i]);}
118
119             ierr = MatSetValues(matriz_coef,1,&i,3,colcoef,value,INSERT_VALUES);CHKERRQ(ierr);
120         }
121         else
122         {
123             value[1]=Df[0]-MAX(Ff[0],0);
124             value[2]=Df[1]-MAX(Ff[1],0);
125             value[3]=Df[2]-MAX(Ff[2],0);
126             value[0]=-(value[1]+value[2]+value[3]);
128         }
129
130         ierr = MatRestoreRow(matriz_difusivo,i,PETSC_NULL,&cols,&Df);CHKERRQ(ierr);
131         ierr = MatRestoreRow(matriz_convectivo,i,PETSC_NULL,PETSC_NULL,&Ff);CHKERRQ(ierr);
132         ierr = MatRestoreRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
133     }
134     ierr = VecRestoreArray(vec_diffont,&difb);CHKERRQ(ierr);
135     ierr = VecRestoreArray(vec_convecfront,&convb);CHKERRQ(ierr);
136
137     ierr = MatAssemblyBegin(matriz_coef,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
138     ierr = MatAssemblyEnd(matriz_coef,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
139
140
141     /*--------------------- ITERACIONES PARA CALCULAR LA SOLUCIÓN ---------------------*/
142
143     //Entorno del solver
```

```
144     KSPCreate(PETSC_COMM_WORLD,&ksp);
145     KSPSetOperators(ksp,matriz_coef,matriz_coef,DIFFERENT_NONZERO_PATTERN);
146     KSPSetType(ksp,KSPGMRES);
147     KSPGetPC(ksp,&pc);
148     PCSetType(pc,PCGAMG);
149     KSPSetfomOptions(ksp);
150     KSPSetUp(ksp);
151     niter=0;
152     error=1;
153 
154     while (error>1*10e-6 && niter<30)
155     {
156         //Calcular el vector de términos independientes con los valores de phi de la iteración anterior
157         grad.calcular_grad(b,f,vec_soluant,&matriz_gradphivc);
158         ib.calcular_bc(b,f,df,cv,vec_soluant,matriz_gradphivc,&vec_b);
159 
160         //Calcular el vector solución
161         KSPSolve(ksp,vec_b,vec_solucion);
162 
163         ierr = VecGetArray(vec_solucion,&phivc);CHKERRQ(ierr);
164         ierr = VecGetArray(vec_soluant,&phivc0);CHKERRQ(ierr);
165 
166         //Calcular el error entre el vector calculado y el vector de la iteración anterior
167         error=0;
168         for (i=0;i<n_vc;i++)
169         {
170             temporal=sqrt(pow(phivc[i]-phivc0[i],2));
171             error=MAX(error,temporal);
172         }
173         //Asignar los valores del vector calculado al vector de la solución anterior
174         for (i=0;i<n_vc;i++)
175         {
176             phi0=phivc[i];
177             ierr = VecSetValues(vec_soluant,1,&i,&phi0,INSERT_VALUES);CHKERRQ(ierr);
178         }
179 
180         ierr = VecAssemblyBegin(vec_soluant);CHKERRQ(ierr);
181         ierr = VecAssemblyEnd(vec_soluant);CHKERRQ(ierr);
182         niter++;
183     }
184 
185     //Exportar los resultados
186     rta.imprimir(gm,vec_solucion);
187 
188     return ierr;
189 }
190 
191 
192 
```

18. ARCHIVO DE ENCABEZADO DE LA CLASE SOLEXPLÍCITO

```
1  //ARCHIVO DE ENCABEZADO DE LA CLASE SOLUCION TRANSITORIA (EXPLÍCITO): solexplicito.h
2  #ifndef _SOLEXPLICITO_H_
3  #define _SOLEXPLICITO_H_
4 
5  class solexplicito
6  {
7      private:
8 
9          PetscErrorCode ierr;               //Variable de error de Petsc
10         int i,j,k,ii;                      //Contadores
```

```
10          int i,j,k,ii;                       //Contadores
11          int tem,temp,tempor,tempora,ttemp,ttempo;
12          double temconst, temporal;          //Variables temporales
13
14          //Variables para establecer las columnas
15          const PetscInt *cols;
16          PetscInt colcoef[4];
17
18          //Elementos geométricos
19          int n_vc;
20          Vec vec_volumen;
21              PetscScalar *volumenvc;
22          Mat matriz_areas;
23              const PetscScalar *areas;
24          Mat matriz_alfaf;
25              const PetscScalar *alfaf;
26          Mat matriz_deltaxie;
27              const PetscScalar *deltaxies;
28          Mat matriz_vecppprima;
29              const PetscScalar *ppprima;
30              const PetscScalar *ffprima;
31          Mat matriz_vecinos;
32              const PetscScalar *vecinos;
33              const PetscScalar *vvecinos;
34          Mat matriz_coord;
35              const PetscScalar *coord;
36
37          //Elementos de frontera
38          Vec vec_front;
39              PetscScalar *valueb;
40          Vec vec_condf;
41              PetscScalar *condf;
42
43          //Objeto de la clase difusivo
44          difusivo df;
45          //Variables para calcular y usar la matriz de términos difusivos
46          Mat matriz_difusivo;
47              const PetscScalar *Df;
48          //Variables para calcular y usar el vector de términos difusivos en las fronteras
49          Vec vec_diffront;
50          PetscScalar *difb;
51          //Objeto de la clase convectivo
52          convectivo cv;
53          //Variables para calcular y usar la matriz de términos convectivos
54          Mat matriz_convectivo;
55              const PetscScalar *Ff;
56          //Variables para calcular y usar el vector de términos convectivos en las fronteras
57          Vec vec_convecfront;
58          PetscScalar *convb;
59          //Objeto de la clase gradiente
60          gradiente grad;
62          Mat matriz_gradphivc;
63          //Objeto de la clase termindep
64          termindep ib;
65          //Variables para calcular el vector de terminos independientes
66          Vec vec_b;
67          //Variable para imprimir los resultados en archivo .vtk
68          resultados rta;
69
70          //Vector de coeficientes ap0
71          Vec vec_ap0;
72          PetscScalar ap_0;
73          PetscScalar *ap0;
74          PetscScalar *independ;
75
76          //Matriz de coeficientes
```

```
77          Mat matriz_coef;
78          const PetscScalar *coef;
79          PetscScalar phip;
80          PetscScalar *termb;
81
82          //Vector de phi en el tiempo anterior
83          Vec vec_phi0;
84          PetscScalar *phip0;
85          PetscScalar *phivcnu;
86          PetscScalar *phivcnv;
87
88          //Vector solucion actual
89          Vec vec_solucion;
90              PetscScalar *phivc0;
91
92          //Variables de cálculo
93          PetscScalar value[4];
94          PetscScalar phi0,phi;
95          double error,niter;
96          double timestep;
97
98      public:
99
100         //Funciones
101         PetscErrorCode calcular_Sti(geom gm,frontera fr,double densidad,double conductividad,double tiempo,
double *velocidad);
102
103 };
104
105 #endif // _SOLTRANSITORIA_H_
```

19. ARCHIVO DE CÓDIGO FUENTE DE LA CLASE SOLEXPLICITO

```
1  //ARCHIVO DE CÓDIGO FUENTE DE LA CLASE SOLUCIÓN TRANSITORIA (EXPLICITO): solexplicito.cpp
2
3  #include <petsc.h>
4  #include <cmath>
5  #include "geom.h"
6  #include "resultados.h"
7  #include "frontera.h"
8  #include "gradiente.h"
9  #include "difusivo.h"
10 #include "convectivo.h"
11 #include "termindep.h"
12
13 PetscErrorCode solucion::calcular_Ste(double densidad,double gamma,geom gm,frontera f)
14 {
15     //Tomar los elementos de la clase geom
16     n_vc=gm.vc;
17     vec_volumen=gm.vec_volumen;
18     matriz_areas=gm.matriz_areas;
19     matriz_alfaf=gm.matriz_alfaf;
20     matriz_vecppprima=gm.matriz_vecppprima;
21     matriz_vecinos=gm.matriz_vecinos;
22     matriz_coord=gm.matriz_coord;
23     //Tomar los elementos de la clase fontera
24     vec_front=f.vec_front;
25     vec_condf=f.vec_condf;
26
27     /* ----------------------  INICIALIZACIÓN DE VARIABLES ----------------------- */
28
29     PetscInt col[n_vc];
30     PetscScalar val[n vc];
```

```
30      PetscScalar val[n_vc];
31      for (i=0;i<n_vc;i++)
32      {
33          col[i]=i;
34          val[i]=0;
35      }
36
37      //Matriz de terminos difusivos
38      ierr = MatCreate(PETSC_COMM_WORLD,&matriz_difusivo);CHKERRQ(ierr);
39      ierr = MatSetSizes(matriz_difusivo,PETSC_DECIDE,PETSC_DECIDE,n_vc,n_vc);CHKERRQ(ierr);
40      ierr = MatSetFromOptions(matriz_difusivo);CHKERRQ(ierr);
41      ierr = MatSetUp(matriz_difusivo);CHKERRQ(ierr);
42
43      //Vector de terminos difusivos en las fronteras
44      ierr = VecCreate(PETSC_COMM_WORLD,&vec_diffront);CHKERRQ(ierr);
45      ierr = VecSetSizes(vec_diffront,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
46      ierr = VecSetFromOptions(vec_diffront);CHKERRQ(ierr);
47
48      //Matriz de terminos convectivos
49      ierr = MatCreate(PETSC_COMM_WORLD,&matriz_convectivo);CHKERRQ(ierr);
50      ierr = MatSetSizes(matriz_convectivo,PETSC_DECIDE,PETSC_DECIDE,n_vc,n_vc);CHKERRQ(ierr);
51      ierr = MatSetFromOptions(matriz_convectivo);CHKERRQ(ierr);
52      ierr = MatSetUp(matriz_convectivo);CHKERRQ(ierr);
53
54      //Vector de terminos convectivos en las frontera
55      ierr = VecCreate(PETSC_COMM_WORLD,&vec_convecfront);CHKERRQ(ierr);
56      ierr = VecSetSizes(vec_convecfront,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
57      ierr = VecSetFromOptions(vec_convecfront);CHKERRQ(ierr);
58
59      //Matriz de coeficientes
60      ierr = MatCreate(PETSC_COMM_WORLD,&matriz_coef);CHKERRQ(ierr);
61      ierr = MatSetSizes(matriz_coef,PETSC_DECIDE,PETSC_DECIDE,n_vc,n_vc);CHKERRQ(ierr);
62      ierr = MatSetFromOptions(matriz_coef);CHKERRQ(ierr);
63      ierr = MatSetUp(matriz_coef);CHKERRQ(ierr);
64
65      //Matriz del gradiente de phi
66      ierr = MatCreate(PETSC_COMM_WORLD,&matriz_gradphivc);CHKERRQ(ierr);
67      ierr = MatSetSizes(matriz_gradphivc,PETSC_DECIDE,PETSC_DECIDE,n_vc,2);CHKERRQ(ierr);
68      ierr = MatSetFromOptions(matriz_gradphivc);CHKERRQ(ierr);
69      ierr = MatSetUp(matriz_gradphivc);CHKERRQ(ierr);
70
71      //Vector de terminos independientes
72      ierr = VecCreate(PETSC_COMM_WORLD,&vec_b);CHKERRQ(ierr);
73      ierr = VecSetSizes(vec_b,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
74      ierr = VecSetFromOptions(vec_b);CHKERRQ(ierr);
75
76      //Vector solucion
77      ierr = VecCreate(PETSC_COMM_WORLD,&vec_solucion);CHKERRQ(ierr);
78      ierr = VecSetSizes(vec_solucion,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
79      ierr = VecSetFromOptions(vec_solucion);CHKERRQ(ierr);
80
81      //Vector solucion de la iteracion anterior
82      ierr = VecCreate(PETSC_COMM_WORLD,&vec_soluant);CHKERRQ(ierr);
83      ierr = VecSetSizes(vec_soluant,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
84      ierr = VecSetFromOptions(vec_soluant);CHKERRQ(ierr);
85      ierr = VecSet(vec_soluant,0);CHKERRQ(ierr);
86      ierr = VecAssemblyBegin(vec_soluant);CHKERRQ(ierr);
87      ierr = VecAssemblyEnd(vec_soluant);CHKERRQ(ierr);
88
89      //Vector solucion del tiempo anterior
90      ierr = VecCreate(PETSC_COMM_WORLD,&vec_phi0);CHKERRQ(ierr);
91      ierr = VecSetSizes(vec_phi0,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
```

```
92      ierr = VecSetFromOptions(vec_phi0);CHKERRQ(ierr);
93      ierr = VecSet(vec_phi0,0);CHKERRQ(ierr);
94      ierr = VecAssemblyBegin(vec_phi0);CHKERRQ(ierr);
95      ierr = VecAssemblyEnd(vec_phi0);CHKERRQ(ierr);
96
97      //Vector de velocidades u
98      ierr = VecCreate(PETSC_COMM_WORLD,&vec_u);CHKERRQ(ierr);
99      ierr = VecSetSizes(vec_u,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
100     ierr = VecSetFromOptions(vec_u);CHKERRQ(ierr);
101
102     //Vector de velocidades v
103     ierr = VecCreate(PETSC_COMM_WORLD,&vec_v);CHKERRQ(ierr);
104     ierr = VecSetSizes(vec_v,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
105     ierr = VecSetFromOptions(vec_v);CHKERRQ(ierr);
106
107     //Vector de coeficientes ap0
108     ierr = VecCreate(PETSC_COMM_WORLD,&vec_ap0);CHKERRQ(ierr);
109     ierr = VecSetSizes(vec_ap0,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
110     ierr = VecSetFromOptions(vec_ap0);CHKERRQ(ierr);
111
112     //Cálculo de la velocidad para el problema Smith-Hutton
113     for (i=0;i<n_vc;i++)
114     {
115         ierr = MatGetRow(matriz_coord,i,PETSC_NULL,PETSC_NULL,&coord);CHKERRQ(ierr);
116
117         u=2*coord[1]*(1-coord[0]*coord[0]);
118
119         ierr = MatRestoreRow(matriz_coord,i,PETSC_NULL,PETSC_NULL,&coord);CHKERRQ(ierr);
120         ierr = VecSetValues(vec_u,1,&i,&u,INSERT_VALUES);CHKERRQ(ierr);
121         ierr = VecSetValues(vec_v,1,&i,&v,INSERT_VALUES);CHKERRQ(ierr);
122     }
123     ierr = VecAssemblyBegin(vec_u);CHKERRQ(ierr);
124     ierr = VecAssemblyEnd(vec_u);CHKERRQ(ierr);
125     ierr = VecAssemblyBegin(vec_v);CHKERRQ(ierr);
126     ierr = VecAssemblyEnd(vec_v);CHKERRQ(ierr);
127
128     /*------------------------- CÁLCULO DE TÉRMINOS DIFUSIVOS -------------------------*/
129     std::cout<<"Empieza difusivo"<<std::endl;
130     df.calcular_D(gamma,b,&matriz_difusivo,&vec_diffront);
131
132     /*------------------------ CÁLCULO DE TÉRMINOS CONVECTIVOS ------------------------*/
133     std::cout<<"Empieza convectivo"<<std::endl;
134     cv.calcular_F(densidad,b,&matriz_convectivo,&vec_convecfront);
135
136     /*------------------------------ CÁLCULO DEL DELTA T ------------------------------*/
137     double timestepvisc;
138     double timestepconv;
139     double timestep;
140
141     double areamenor=100000;
142     for (i=0;i<n_vc;i++)
143     {
144         ierr = MatGetRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
145         areamenor=MIN(areamenor,sqrt(areas[0]*areas[0]+areas[1]*areas[1]));
146         ierr = MatRestoreRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
147     }
148     timestepvisc=0.2*densidad*areamenor*areamenor/4/gamma;
149     std::cout<<"timestepvisc="<<timestepvisc<<std::endl;
150
151
152     ierr = VecGetArray(vec_u,&phivcnu);CHKERRQ(ierr);
153     ierr = VecGetArray(vec_v,&phivcnv);CHKERRQ(ierr);
154     double deltainv;
155     double phivcni;
156     double deltati=100000;
157
```

```
158     for(i=0;i<n_vc;i++)
159     {
160         temp=0; tempor=1; areamenor=10000;
161         ierr = MatGetRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
162         for(j=0;j<3;j++)
163         {
164             areamenor=sqrt(areas[temp]*areas[temp]+areas[tempor]*areas[tempor]);
165             temp=temp+2;tempor=tempor+2;
166         }
167         ierr = MatRestoreRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
168         phivcni=sqrt((phivcnu[i]*phivcnu[i])+(phivcnv[i]*phivcnv[i]));
169         if(phivcni!=0)
170         {deltati=MIN(deltati,areamenor/phivcni);}
171         else
172         {deltati=deltati;}
173
174     }
175     timestepconv=0.35*deltati;
176
177     ierr = VecRestoreArray(vec_u,&phivcnu);CHKERRQ(ierr);
178     ierr = VecRestoreArray(vec_v,&phivcnv);CHKERRQ(ierr);
179
180     /*---------------------------- CÁLCULO DEL COEFICIENTE aP0 ----------------------------*/
181     timestep=MIN(timestepvisc,timestepconv);
182
183     ierr = VecGetArray(vec_volumen,&volumenvc);CHKERRQ(ierr);
184     for (i=0;i<n_vc;i++)
185     {
186         ap_0=densidad*volumenvc[i]/timestep;
187         ierr = VecSetValues(vec_ap0,1,&i,&ap_0,INSERT_VALUES);CHKERRQ(ierr);
188     }
189     ierr = VecRestoreArray(vec_volumen,&volumenvc);CHKERRQ(ierr);
190
191     ierr = VecAssemblyBegin(vec_ap0);CHKERRQ(ierr);
192     ierr = VecAssemblyEnd(vec_ap0);CHKERRQ(ierr);
193
194     /*------------------------ CREAR LA MATRIZ DE COEFICIENTES ------------------------*/
195
196     ierr = VecGetArray(vec_diffront,&difb);CHKERRQ(ierr);
197     ierr = VecGetArray(vec_convecfront,&convb);CHKERRQ(ierr);
198     ierr = VecGetArray(vec_condf,&condf);CHKERRQ(ierr);
199     ierr = VecGetArray(vec_ap0,&ap0);CHKERRQ(ierr);
200
201     for (i=0;i<n_vc;i++)
202     {
203
204         ierr = MatGetRow(matriz_difusivo,i,PETSC_NULL,&cols,&Df);CHKERRQ(ierr);
205         ierr = MatGetRow(matriz_convectivo,i,PETSC_NULL,PETSC_NULL,&Ff);CHKERRQ(ierr);
206         ierr = MatGetRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
207
208         colcoef[0]=i;colcoef[1]=cols[0];colcoef[2]=cols[1];colcoef[3]=cols[2];
209
210         if(vecinos[2]>n_vc)
211         {
212             value[1]=-Df[0]-MAX(-Ff[0],0);
213             value[2]=-Df[1]-MAX(-Ff[1],0);
214             if(condf[i]==1)
215             {value[0]=-(value[1]+value[2])+Ff[0]+Ff[1]+difb[i]-ap0[i];}
216             if(condf[i]==-1)
217             {value[0]=-(value[1]+value[2])+Ff[0]+Ff[1]-ap0[i];}
218             if(condf[i]==-2)
219             {value[0]=-(value[1]+value[2])+Ff[0]+Ff[1]-ap0[i];}
220             ierr = MatSetValues(matriz_coef,1,&i,3,colcoef,value,INSERT_VALUES);CHKERRQ(ierr);
221         }
222         else
223         {
```

```
            value[1]=-Df[0]-MAX(-Ff[0],0);
            value[2]=-Df[1]-MAX(-Ff[1],0);
            value[3]=-Df[2]-MAX(-Ff[2],0);
            value[0]=-(value[1]+value[2]+value[3])+Ff[0]+Ff[1]+Ff[2]-ap0[i];
            ierr = MatSetValues(matriz_coef,1,&i,4,colcoef,value,INSERT_VALUES);CHKERRQ(ierr);
        }

        ierr = MatRestoreRow(matriz_difusivo,i,PETSC_NULL,&cols,&Df);CHKERRQ(ierr);
        ierr = MatRestoreRow(matriz_convectivo,i,PETSC_NULL,PETSC_NULL,&Ff);CHKERRQ(ierr);
        ierr = MatRestoreRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
    }
    ierr = VecRestoreArray(vec_diffront,&difb);CHKERRQ(ierr);
    ierr = VecRestoreArray(vec_convecfront,&convb);CHKERRQ(ierr);
    ierr = VecRestoreArray(vec_condf,&condf);CHKERRQ(ierr);
    ierr = VecRestoreArray(vec_ap0,&ap0);CHKERRQ(ierr);

    ierr = MatAssemblyBegin(matriz_coef,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
    ierr = MatAssemblyEnd(matriz_coef,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);

    /*--------------------- ITERACIONES PARA CALCULAR LA SOLUCIÓN ---------------------*/

    double tiempos=10000;
    int titer=0;
    error=1;

    while(error>1e-6 )
    {
        niter=0;
        error=1;

        grad.calcular_grad(b,f,vec_phi0,&matriz_gradphivc);
        ib.calcular_be(b,f,matriz_difusivo,vec_diffront,vec_convecfront,matriz_gradphivc,vec_phi0,timestep
,&vec_b);

        ierr = VecGetArray(vec_phi0,&phivc0);CHKERRQ(ierr);
        ierr = VecGetArray(vec_b,&termb);CHKERRQ(ierr);
        ierr = VecGetArray(vec_ap0,&ap0);CHKERRQ(ierr);

        double sumatoria=0;
        int numero;
        for (i=0;i<n_vc;i++)
        {
            ierr = MatGetRow(matriz_coef,i,PETSC_NULL,&cols,&coef);CHKERRQ(ierr);
            ierr = MatGetRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
            sumatoria=0;
            k=0;
            if (vecinos[2]>n_vc)
            {
                for (j=0;j<3;j++)
                {
                    if(cols[j]==i)
                    {numero=j;}
                    else
                    {
                        tem=vecinos[k];
                        sumatoria=sumatoria+coef[j]*phivc0[tem];
                        k++;
                    }
                }
                phip=(-sumatoria-coef[numero]*phivc0[i]+termb[i])/ap0[i];
            }

            else
            {
                for (j=0;j<4;j++)
                {
```

```
                    if(cols[j]==i)
                    {numero=j;}
                    else
                    {
                        tem=vecinos[k];
                        sumatoria=sumatoria+coef[j]*phivc0[tem];
                        k++;
                    }
                }

                phip=(-sumatoria-coef[numero]*phivc0[i]+termb[i])/ap0[i];
            }

            ierr = VecSetValues(vec_solucion,1,&i,&phip,INSERT_VALUES);CHKERRQ(ierr);
            sumatoria=0;
        }

        ierr = VecAssemblyBegin(vec_solucion);CHKERRQ(ierr);
        ierr = VecAssemblyEnd(vec_solucion);CHKERRQ(ierr);

        error=0;
        ierr = VecGetArray(vec_phi0,&phivc0);CHKERRQ(ierr);
        ierr = VecGetArray(vec_solucion,&phivc);CHKERRQ(ierr);
            for (i=0;i<n_vc;i++)
            {
                temporal=sqrt(pow(phivc[i]-phivc0[i],2));
                error=MAX(error,temporal);
                ierr = VecSetValues(vec_phi0,1,&i,&phi0,INSERT_VALUES);CHKERRQ(ierr);
            }
        ierr = VecRestoreArray(vec_phi0,&phivc0);CHKERRQ(ierr);
        ierr = VecRestoreArray(vec_solucion,&phivc);CHKERRQ(ierr);

        ierr = VecGetArray(vec_solucion,&phivc);CHKERRQ(ierr);
        for (i=0;i<n_vc;i++)
        {
            phi0=phivc[i];
            ierr = VecSetValues(vec_phi0,1,&i,&phi0,INSERT_VALUES);CHKERRQ(ierr);
        }
        ierr = VecAssemblyBegin(vec_phi0);CHKERRQ(ierr);
        ierr = VecAssemblyEnd(vec_phi0);CHKERRQ(ierr);

        titer++;
    }

    //Exportar los resultados
    rta.imprimir(gm,vec_u,vec_v,vec_solucion);

    return ierr;
}
```

20. ARCHIVO DE ENCABEZADO DE LA CLASE FRACTIONAL

```
//ARCHIVO DE ENCABEZADO DE LA CLASE SOLUCION DEL PROBLEMA DE FLUJO: fractional.h

#ifndef _FRACTIONAL_H_
#define _FRACTIONAL_H_

class fractional
{
    private:

        PetscErrorCode ierr;        //Variable de error de petsc
```

```
11
12          //Variables de uso temporal
13          int i,j,k,ii;
14          int tem,temp,tempor,tempora,ttemp,ttempo;
15          double temporal;
16          double err,nosirve;
17
18          //Elementos geométricos
19          int n_vc;
20          Vec vec_volumen;
21          PetscScalar *volumenvc;
22          Mat matriz_areas;
23          const PetscScalar *areas;
24          Mat matriz_vecinos;
25          const PetscScalar *vecinos;
26          const PetscScalar *vvecinos;
27
28          //Elementos de frontera
29          Vec vec_frontu;
30          Vec vec_frontv;
31          Vec vec_frontp;
32          PetscScalar *valueb;
33          Vec vec_condfu;
34          Vec vec_condfv;
35          Vec vec_condfp;
36          PetscScalar *condf;
37          const PetscScalar *Df;
38          PetscScalar *difb;
39          const PetscScalar *Ff;
40          PetscScalar *convecb;
41
42          //deltat
43          double deltat;
44          double deltatvisc;
45          double deltatconv;
46          //Matriz de coeficientes
47          Mat matriz_coef;
48          const PetscScalar *coeficiente;
49          //Variable para hallar los terminos difusivos para la velocidad
50          difusivo dv;
51          Mat matriz_difusivo;
52          Vec vec_diffront;
53          //Variable para hallar los terminos convectivos en cada tiempo
54          convectivo cv;
55          Mat matriz_convectivon;
56          Mat matriz_convectivon1;
57          Vec vec_convecfrontn1;
58          Vec vec_convecfrontn;
59          //Vector Ru
60          PetscScalar Ru;
61          Vec vec_Run;
62          PetscScalar *Ru_n;
63          Vec vec_Run1;
64          PetscScalar *Ru_n1;
65          //Vector Rv
66          PetscScalar Rv;
67          Vec vec_Rvn;
68          PetscScalar *Rv_n;
69          Vec vec_Rvn1;
70          PetscScalar *Rv_n1;
71          //Vector de la velocidad u predictora
72          Vec vec_up;
73          PetscScalar up;
74          PetscScalar vp;
75          //Vector de la velocidad v predictora
76          Vec vec_vp;
77          PetscScalar *u_p;
```

```
78          PetscScalar *v_p;
79          //Variables para hallar el gradiente de u y el vector de terminos independientes de u
80          gradiente gru;
81          termindep ibu;
82          Mat matriz_gradphivcun;
83          Mat matriz_gradphivcun1;
84          Vec vec_bun;
85          Vec vec_bun1;
86          //Variables para hallar el gradiente de v y el vector de terminos independientes de v
87          gradiente grv;
88          termindep ibv;
89          Mat matriz_gradphivcvn;
90          Mat matriz_gradphivcvn1;
91          Vec vec_bvn;
92          Vec vec_bvn1;
93          //Variable para hallar la velocidad predictora en las caras
94          gradiente grup;
95          gradiente grvp;
96          Mat matriz_upf;
97          Mat matriz_vpf;
98          //Variable para hallar los terminos difusivos para la presion
99          difusivo dp;
100         Mat matriz_difusivop;
101         Vec vec_diffrontp;
102         //Variables para hallar el gradiente de p y el vector de terminos independientes de p
103         gradiente grp;
104         termindep ibp;
105         Mat matriz_gradp;
106         const PetscScalar *gradp;
107         Vec vec_bp;
108
109         //Variables de cálculo
110         const PetscInt *cols;
111         PetscInt colcoef[4];
112         PetscScalar value[4];
113         PetscScalar *phivc0,*phivc,*phivcnu,*phivcnv,*phivcn1;
114         PetscScalar phi0,phi;
115         PetscScalar *termb;
116         double trecorrido;
117         PetscScalar u;
118         PetscScalar v;
119
120         //KSP
121         KSP ksp;
122         PetscScalar error;
123         PC pc;
124         //Vector solucion de la velocidad u en el tiempo n+1
125         Vec vec_solucionu;
126         //Vector solucion de la velocidad v en el tiempo n+1
127         Vec vec_solucionv;
128         //Vector solucion de u en el tiempo n-1
129        Vec vec_soln1;
130         //Vector solucion de v en el tiempo n-1
131        Vec vec_solvn1;
132         //Vector solucion de u en el tiempo n
133         Vec vec_solun;
134         //Vector solucion de v en el tiempo n
135         Vec vec_solvn;
136         //Vector solucion de la presion
137         Vec vec_solucionp;
138         //Vector solucion de la iteracion anterior para la presion
139         Vec vec_soluantp;
140
141         //Objeto para exportar los resultados
142         resultados rta;
143
```

```
144     public:
145
146         //Funciones
147         PetscErrorCode calcular_SF(geom gm,frontera fu,frontera fv, frontera fp, double viscosidad, double
densidad, double tiempo);
148
149 };
150
151 #endif // _FRACTIONAL_H_
```

21. ARCHIVO DE CÓDIGO FUENTE DE LA CLASE FRACTIONAL

```
1  //ARCHIVO DE CÓDIGO FUENTE DE LA CLASE SOLUCIÓN DEL PROBLEMA DE FLUJO: fractional.cpp
2
3  #include <petsc.h>
4  #include <cmath>
5  #include "geom.h"
6  #include "resultados.h"
7  #include "frontera.h"
8  #include "gradiente.h"
9  #include "difusivo.h"
10 #include "convectivo.h"
11 #include "termindep.h"
12
13 PetscErrorCode fractional::calcular_SF(geom gm,frontera fu,frontera fv, frontera fp, double viscosidad,
double densidad, double tiempo)
14 {
15     n_vc=gm.vc;
16     vec_volumen=gm.vec_volumen;
17     matriz_areas=gm.matriz_areas;
18     matriz_vecinos=gm.matriz_vecinos;
19     vec_frontu=fu.vec_front;
20     vec_condfu=fu.vec_condf;
21     vec_frontv=fv.vec_front;
22     vec_condfv=fv.vec_condf;
23     vec_frontp=fp.vec_front;
24     vec_condfp=fp.vec_condf;
25
26     PetscInt col[n_vc];
27     for (i=0;i<n_vc;i++)
28     {
29         col[i]=i;
30     }
31
32     //Vector solucion de la velocidad u en el tiempo n+1
33     ierr = VecCreate(PETSC_COMM_WORLD,&vec_solucionu);CHKERRQ(ierr);
34     ierr = VecSetSizes(vec_solucionu,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
35     ierr = VecSetFromOptions(vec_solucionu);CHKERRQ(ierr);
36
37     //Vector solucion de la velocidad v en el tiempo n+1
38     ierr = VecCreate(PETSC_COMM_WORLD,&vec_solucionv);CHKERRQ(ierr);
39     ierr = VecSetSizes(vec_solucionv,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
40     ierr = VecSetFromOptions(vec_solucionv);CHKERRQ(ierr);
41
42     //Vector solucion de u en el tiempo anterior n-1
43     ierr = VecCreate(PETSC_COMM_WORLD,&vec_solun1);CHKERRQ(ierr);
44     ierr = VecSetSizes(vec_solun1,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
45     ierr = VecSetFromOptions(vec_solun1);CHKERRQ(ierr);
46     ierr = VecSet(vec_solun1,0);CHKERRQ(ierr);
47     ierr = VecAssemblyBegin(vec_solun1);CHKERRQ(ierr);
48     ierr = VecAssemblyEnd(vec_solun1);CHKERRQ(ierr);
49
50     //Vector solucion de v en el tiempo anterior n-1
```

```
51      ierr = VecCreate(PETSC_COMM_WORLD,&vec_solvn1);CHKERRQ(ierr);
52      ierr = VecSetSizes(vec_solvn1,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
53      ierr = VecSetFromOptions(vec_solvn1);CHKERRQ(ierr);
54      ierr = VecSet(vec_solvn1,0);CHKERRQ(ierr);
55      ierr = VecAssemblyBegin(vec_solvn1);CHKERRQ(ierr);
56      ierr = VecAssemblyEnd(vec_solvn1);CHKERRQ(ierr);
57
58      //Vector solucion de u en el tiempo actual n
59      ierr = VecCreate(PETSC_COMM_WORLD,&vec_solun);CHKERRQ(ierr);
60      ierr = VecSetSizes(vec_solun,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
61      ierr = VecSetFromOptions(vec_solun);CHKERRQ(ierr);
62      ierr = VecSet(vec_solun,0);CHKERRQ(ierr);
63      ierr = VecAssemblyBegin(vec_solun);CHKERRQ(ierr);
64      ierr = VecAssemblyEnd(vec_solun);CHKERRQ(ierr);
65
66      //Vector solucion de v en el tiempo actual n
67      ierr = VecCreate(PETSC_COMM_WORLD,&vec_solvn);CHKERRQ(ierr);
68      ierr = VecSetSizes(vec_solvn,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
69      ierr = VecSetFromOptions(vec_solvn);CHKERRQ(ierr);
70      ierr = VecSet(vec_solvn,0);CHKERRQ(ierr);
71      ierr = VecAssemblyBegin(vec_solvn);CHKERRQ(ierr);
72      ierr = VecAssemblyEnd(vec_solvn);CHKERRQ(ierr);
73
74      //Vector R de u en el tiempo anterior n-1
75      ierr = VecCreate(PETSC_COMM_WORLD,&vec_Run1);CHKERRQ(ierr);
76      ierr = VecSetSizes(vec_Run1,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
77      ierr = VecSetFromOptions(vec_Run1);CHKERRQ(ierr);
78
79      //Vector R de v en el tiempo anterior n-1
80      ierr = VecCreate(PETSC_COMM_WORLD,&vec_Rvn1);CHKERRQ(ierr);
81      ierr = VecSetSizes(vec_Rvn1,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
82      ierr = VecSetFromOptions(vec_Rvn1);CHKERRQ(ierr);
83
84      //Vector R de u en el tiempo actual n
85      ierr = VecCreate(PETSC_COMM_WORLD,&vec_Run);CHKERRQ(ierr);
86      ierr = VecSetSizes(vec_Run,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
87      ierr = VecSetFromOptions(vec_Run);CHKERRQ(ierr);
88
89      //Vector R de v en el tiempo actual n
90      ierr = VecCreate(PETSC_COMM_WORLD,&vec_Rvn);CHKERRQ(ierr);
91      ierr = VecSetSizes(vec_Rvn,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
92      ierr = VecSetFromOptions(vec_Rvn);CHKERRQ(ierr);
93
94      //Vector de la u predictora
95      ierr = VecCreate(PETSC_COMM_WORLD,&vec_up);CHKERRQ(ierr);
96      ierr = VecSetSizes(vec_up,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
97      ierr = VecSetFromOptions(vec_up);CHKERRQ(ierr);
98
99      //Vector de la v predictora
100     ierr = VecCreate(PETSC_COMM_WORLD,&vec_vp);CHKERRQ(ierr);
101     ierr = VecSetSizes(vec_vp,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
102     ierr = VecSetFromOptions(vec_vp);CHKERRQ(ierr);
103
104     //Vector solucion de la presion
105     ierr = VecCreate(PETSC_COMM_WORLD,&vec_solucionp);CHKERRQ(ierr);
106     ierr = VecSetSizes(vec_solucionp,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
107     ierr = VecSetFromOptions(vec_solucionp);CHKERRQ(ierr);
108
109     //Vector solucion de la iteracion anterior para la presion
110     ierr = VecCreate(PETSC_COMM_WORLD,&vec_soluantp);CHKERRQ(ierr);
111     ierr = VecSetSizes(vec_soluantp,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
112     ierr = VecSetFromOptions(vec_soluantp);CHKERRQ(ierr);
113     ierr = VecSet(vec_soluantp,0.0);CHKERRQ(ierr);
114     ierr = VecAssemblyBegin(vec_soluantp);CHKERRQ(ierr);
115     ierr = VecAssemblyEnd(vec_soluantp);CHKERRQ(ierr);
116
```

```
117     //Matriz del coeficientes para el calculo de la presion
118     ierr = MatCreate(PETSC_COMM_WORLD,&matriz_coef);CHKERRQ(ierr);
119     ierr = MatSetSizes(matriz_coef,PETSC_DECIDE,PETSC_DECIDE,n_vc,n_vc);CHKERRQ(ierr);
120     ierr = MatSetFromOptions(matriz_coef);CHKERRQ(ierr);
121     ierr = MatSetUp(matriz_coef);CHKERRQ(ierr);
122
123     //Matriz de terminos difusivos de la velocidad
124     ierr = MatCreate(PETSC_COMM_WORLD,&matriz_difusivo);CHKERRQ(ierr);
125     ierr = MatSetSizes(matriz_difusivo,PETSC_DECIDE,PETSC_DECIDE,n_vc,n_vc);CHKERRQ(ierr);
126     ierr = MatSetFromOptions(matriz_difusivo);CHKERRQ(ierr);
127     ierr = MatSetUp(matriz_difusivo);CHKERRQ(ierr);
128
129     //Vector de terminos difusivos de la velocidad en las fronteras
130     ierr = VecCreate(PETSC_COMM_WORLD,&vec_diffront);CHKERRQ(ierr);
131     ierr = VecSetSizes(vec_diffront,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
132     ierr = VecSetFromOptions(vec_diffront);CHKERRQ(ierr);
133
134     //Matriz de terminos difusivos de la presion
135     ierr = MatCreate(PETSC_COMM_WORLD,&matriz_difusivop);CHKERRQ(ierr);
136     ierr = MatSetSizes(matriz_difusivop,PETSC_DECIDE,PETSC_DECIDE,n_vc,n_vc);CHKERRQ(ierr);
137     ierr = MatSetFromOptions(matriz_difusivop);CHKERRQ(ierr);
138     ierr = MatSetUp(matriz_difusivop);CHKERRQ(ierr);
139
140     //Vector de terminos difusivos de la presion en las fronteras
141     ierr = VecCreate(PETSC_COMM_WORLD,&vec_diffrontp);CHKERRQ(ierr);
142     ierr = VecSetSizes(vec_diffrontp,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
143     ierr = VecSetFromOptions(vec_diffrontp);CHKERRQ(ierr);
144
145     //Matriz de terminos convectivos de la velocidad
146     ierr = MatCreate(PETSC_COMM_WORLD,&matriz_convectivon);CHKERRQ(ierr);
147     ierr = MatSetSizes(matriz_convectivon,PETSC_DECIDE,PETSC_DECIDE,n_vc,n_vc);CHKERRQ(ierr);
148     ierr = MatSetFromOptions(matriz_convectivon);CHKERRQ(ierr);
149     ierr = MatSetUp(matriz_convectivon);CHKERRQ(ierr);
150
151     //Vector de terminos convectivos de la velocidad
152     ierr = VecCreate(PETSC_COMM_WORLD,&vec_convecfrontn);CHKERRQ(ierr);
153     ierr = VecSetSizes(vec_convecfrontn,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
154     ierr = VecSetFromOptions(vec_convecfrontn);CHKERRQ(ierr);
155
156     //Matriz del gradiente de la velocidad u
157     ierr = MatCreate(PETSC_COMM_WORLD,&matriz_gradphivcun);CHKERRQ(ierr);
158     ierr = MatSetSizes(matriz_gradphivcun,PETSC_DECIDE,PETSC_DECIDE,n_vc,2);CHKERRQ(ierr);
159     ierr = MatSetFromOptions(matriz_gradphivcun);CHKERRQ(ierr);
160     ierr = MatSetUp(matriz_gradphivcun);CHKERRQ(ierr);
161
162     //Matriz del gradiente de la velocidad v
163     ierr = MatCreate(PETSC_COMM_WORLD,&matriz_gradphivcvn);CHKERRQ(ierr);
164     ierr = MatSetSizes(matriz_gradphivcvn,PETSC_DECIDE,PETSC_DECIDE,n_vc,2);CHKERRQ(ierr);
165     ierr = MatSetFromOptions(matriz_gradphivcvn);CHKERRQ(ierr);
166     ierr = MatSetUp(matriz_gradphivcvn);CHKERRQ(ierr);
167
168     //Matriz del gradiente de la presion
169     ierr = MatCreate(PETSC_COMM_WORLD,&matriz_gradp);CHKERRQ(ierr);
170     ierr = MatSetSizes(matriz_gradp,PETSC_DECIDE,PETSC_DECIDE,n_vc,2);CHKERRQ(ierr);
171     ierr = MatSetFromOptions(matriz_gradp);CHKERRQ(ierr);
172     ierr = MatSetUp(matriz_gradp);CHKERRQ(ierr);
173
174     //Matriz de up en las caras
175     ierr = MatCreate(PETSC_COMM_WORLD,&matriz_upf);CHKERRQ(ierr);
176     ierr = MatSetSizes(matriz_upf,PETSC_DECIDE,PETSC_DECIDE,n_vc,3);CHKERRQ(ierr);
177     ierr = MatSetFromOptions(matriz_upf);CHKERRQ(ierr);
178     ierr = MatSetUp(matriz_upf);CHKERRQ(ierr);
179
180     //Matriz de vp en las caras
181     ierr = MatCreate(PETSC_COMM_WORLD,&matriz_vpf);CHKERRQ(ierr);
```

```
183     ierr = MatSetFromOptions(matriz_vpf);CHKERRQ(ierr);
184     ierr = MatSetUp(matriz_vpf);CHKERRQ(ierr);
185
186     //Vector de terminos independientes de la velocidad un
187     ierr = VecCreate(PETSC_COMM_WORLD,&vec_bun);CHKERRQ(ierr);
188     ierr = VecSetSizes(vec_bun,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
189     ierr = VecSetFromOptions(vec_bun);CHKERRQ(ierr);
190
191     //Vector de terminos independientes de la velocidad vn
192     ierr = VecCreate(PETSC_COMM_WORLD,&vec_bvn);CHKERRQ(ierr);
193     ierr = VecSetSizes(vec_bvn,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
194     ierr = VecSetFromOptions(vec_bvn);CHKERRQ(ierr);
195
196     //Vector de terminos independientes de la presion
197     ierr = VecCreate(PETSC_COMM_WORLD,&vec_bp);CHKERRQ(ierr);
198     ierr = VecSetSizes(vec_bp,PETSC_DECIDE,n_vc);CHKERRQ(ierr);
199     ierr = VecSetFromOptions(vec_bp);CHKERRQ(ierr);
200
201     //Calcular los terminos difusivos para la velocidad
202     double viscinematica=viscosidad/densidad;
203     dv.calcular_D(viscinematica,gm,&matriz_difusivo,&vec_diffront);
204
205     //calculo del delta t viscoso Cvisc=0.2
206     double areamenor=100000;
207     for (i=0;i<n_vc;i++)
208     {
209         temp=0; tempor=1;
210         ierr = MatGetRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
211         for(j=0;j<3;j++)
212         {
213             areamenor=MIN(areamenor,sqrt(areas[temp]*areas[temp]+areas[tempor]*areas[tempor]));
214             temp=temp+2;tempor=tempor+2;
215         }
216         ierr = MatRestoreRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
217     }
218     deltatvisc=0.2*areamenor*areamenor/viscinematica;
219     std::cout<<"deltat:"<<deltatvisc;
220     int titer=0;
221     int contador=0;
222     trecorrido=0;
223
224     //Comienza a correr el tiempo
225     while (titer<5000) //trecorrido<tiempo
226     {
227         //Calculo del delta t convectivo Cconv=0.35
228
229         ierr = VecGetArray(vec_solun,&phivcnu);CHKERRQ(ierr);
230         ierr = VecGetArray(vec_solvn,&phivcnv);CHKERRQ(ierr);
231         double deltainv;
232         double phivcni;
233         double deltati=100000;
234         for(i=0;i<n_vc;i++)
235         {
236             temp=0; tempor=1; areamenor=10000;
237             ierr = MatGetRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
238             for(j=0;j<3;j++)
239             {
240                 areamenor=sqrt(areas[temp]*areas[temp]+areas[tempor]*areas[tempor]);
241                 temp=temp+2;tempor=tempor+2;
242             }
243             ierr = MatRestoreRow(matriz_areas,i,PETSC_NULL,PETSC_NULL,&areas);CHKERRQ(ierr);
244             phivcni=sqrt((phivcnu[i]*phivcnu[i])+(phivcnv[i]*phivcnv[i]));
245             if(phivcni!=0)
246             {deltati=MIN(deltati,areamenor/phivcni);}
247             else
```

```
248             {deltati=deltati;}
249
250         }
251         deltatconv=0.35*deltati;
252         ierr = VecRestoreArray(vec_solun,&phivcnu);CHKERRQ(ierr);
253         ierr = VecRestoreArray(vec_solvn,&phivcnv);CHKERRQ(ierr);
254
255         //Calculo del delta t
256         deltat=0.3*MIN(deltatvisc,deltatconv);
257
258         if(deltat<1e-9) exit(0);
259
260         double tiempos;
261         tiempos=trecorrido+deltat;
262         if(tiempos>tiempo)
263         {
264             deltat=tiempo-trecorrido;
265             trecorrido=tiempo;
266         }
267         else
268         {
269             trecorrido=tiempos;
270         }
271
272         //Calcular los terminos convectivos para el tiempo actual n
273         tem=1;
274         cv.calcular_F(tem,gm,vec_solun,vec_solvn,fu,fv,&matriz_convectivon,&vec_convecfrontn);
275
276         //Calcular los terminos convectivos para el tiempo n-1
277         tem=1;
278         cv.calcular_F(tem,gm,vec_solun1,vec_solvn1,fu,fv);
279         matriz_convectivon1=cv.matriz_convectivo;
280         vec_convecfrontn1=cv.vec_convecfront;
281
282         //Calcular el gradiente de u y el vector de terminos independientes de u en n-1
283         gru.calcular_grad(gm,fu,vec_solun1);
284         matriz_gradphivcun1=gru.matriz_gradphivc;
285         ibu.calcular_be(gm,fu,dv,cv,matriz_gradphivcun1);
286         vec_bun1=ibu.vec_b;
287
288         //Calcular el gradiente de u y el vector de terminos independientes de u en n
289         gru.calcular_grad(gm,fu,vec_solun,&matriz_gradphivcun);
290         ibu.calcular_be(gm,fu,matriz_difusivo,vec_diffront,vec_convecfrontn,matriz_gradphivcun,&vec_bun);
291
292         //Calcular el gradiente de v y el vector de terminos independientes de v en n
293         grv.calcular_grad(gm,fv,vec_solvn,&matriz_gradphivcvn);
294         ibv.calcular_be(gm,fv,matriz_difusivo,vec_diffront,vec_convecfrontn,matriz_gradphivcvn,&vec_bvn);
295
296         //Calcular el gradiente de v y el vector de terminos independientes de v en n-1
297         grv.calcular_grad(gm,fv,vec_solvn1);
298         matriz_gradphivcvn1=grv.matriz_gradphivc;
299         ibv.calcular_be(gm,fv,dv,cv,matriz_gradphivcvn1);
300         vec_bvn1=ibv.vec_b;
301
302         //Calculo del Ru en n-1
303         ierr = VecGetArray(vec_solun1,&phivcn1);CHKERRQ(ierr);
304         ierr = VecGetArray(vec_bun1,&termb);CHKERRQ(ierr);
305
306         ierr = VecGetArray(vec_condfu,&condf);CHKERRQ(ierr);
307         ierr = VecGetArray(vec_diffront,&difb);CHKERRQ(ierr);
308         ierr = VecGetArray(vec_convecfrontn1,&convecb);CHKERRQ(ierr);
309         for (i=0;i<n_vc;i++)
310         {
311             ierr = MatGetRow(matriz_difusivo,i,PETSC_NULL,PETSC_NULL,&Df);CHKERRQ(ierr);
312             ierr = MatGetRow(matriz_convectivon1,i,PETSC_NULL,PETSC_NULL,&Ff);CHKERRQ(ierr);
313             ierr = MatGetRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
314
```

```
315             Ru=0;
316             if(vecinos[2]>n_vc)
317             {
318                 for (j=0;j<2;j++)
319                 {
320                     k=vecinos[j];
321                     Ru=Ru+Df[j]*(phivcn1[k]-phivcn1[i])-MAX(Ff[j],0)*phivcn1[i]+MAX(-Ff[j],0)*phivcn1[k];
322                 }
323 
324                 Ru=Ru+termb[i];
325                 if(condf[i]==1)
326                 {
327                     Ru=Ru-difb[i]*phivcn1[i];
328                 }
329                 if(condf[i]==2)
330                 {
331                     Ru=Ru-convecb[i]*phivcn1[i];
332                 }
333             }
334 
335             else
336             {
337                 for (j=0;j<3;j++)
338                 {
339                     k=vecinos[j];
340                     Ru=Ru+Df[j]*(phivcn1[k]-phivcn1[i])-MAX(Ff[j],0)*phivcn1[i]+MAX(-Ff[j],0)*phivcn1[k];
341                 }
342                 Ru=Ru+termb[i];
343             }
344 
345             ierr = VecSetValues(vec_Run1,1,&i,&Ru,INSERT_VALUES);CHKERRQ(ierr);
346 
347             ierr = MatRestoreRow(matriz_difusivo,i,PETSC_NULL,PETSC_NULL,&Df);CHKERRQ(ierr);
348             ierr = MatRestoreRow(matriz_convectivon1,i,PETSC_NULL,PETSC_NULL,&Ff);CHKERRQ(ierr);
349             ierr = MatRestoreRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
350         }
351 
352         ierr = VecAssemblyBegin(vec_Run1);CHKERRQ(ierr);
353         ierr = VecAssemblyEnd(vec_Run1);CHKERRQ(ierr);
354 
355         ierr = VecRestoreArray(vec_solun1,&phivcn1);CHKERRQ(ierr);
356         ierr = VecRestoreArray(vec_bun1,&termb);CHKERRQ(ierr);
357         ierr = VecRestoreArray(vec_convecfrontn1,&convecb);CHKERRQ(ierr);
358 
359         //Calculo del Ru en n
360 
361         ierr = VecGetArray(vec_solun,&phivcnu);CHKERRQ(ierr);
362         ierr = VecGetArray(vec_bun,&termb);CHKERRQ(ierr);
363         ierr = VecGetArray(vec_convecfrontn,&convecb);CHKERRQ(ierr);
364 
365         for (i=0;i<n_vc;i++)
366         {
367             ierr = MatGetRow(matriz_difusivo,i,PETSC_NULL,PETSC_NULL,&Df);CHKERRQ(ierr);
368             ierr = MatGetRow(matriz_convectivon,i,PETSC_NULL,PETSC_NULL,&Ff);CHKERRQ(ierr);
369             ierr = MatGetRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
370 
371             Ru=0;
372 
373             if(vecinos[2]>n_vc)
374             {
375                 for (j=0;j<2;j++)
376                 {
377                     k=vecinos[j];
378                     Ru=Ru+Df[j]*(phivcnu[k]-phivcnu[i])-MAX(Ff[j],0)*phivcnu[i]+MAX(-Ff[j],0)*phivcnu[k];
379                 }
380 
```

```
                Ru=Ru+termb[i];
                if(condf[i]==1)
                {
                    Ru=Ru-difb[i]*phivcnu[i];
                }
                if(condf[i]==2)
                {
                    Ru=Ru-convecb[i]*phivcnu[i];
                }
            }

            else
            {
                for (j=0;j<3;j++)
                {
                    k=vecinos[j];
                    Ru=Ru+Df[j]*(phivcnu[k]-phivcnu[i])-MAX(Ff[j],0)*phivcnu[i]+MAX(-Ff[j],0)*phivcnu[k];
                }
                Ru=Ru+termb[i];
            }

            ierr = VecSetValues(vec_Run,1,&i,&Ru,INSERT_VALUES);CHKERRQ(ierr);

            ierr = MatRestoreRow(matriz_difusivo,i,PETSC_NULL,PETSC_NULL,&Df);CHKERRQ(ierr);
            ierr = MatRestoreRow(matriz_convectivon,i,PETSC_NULL,PETSC_NULL,&Ff);CHKERRQ(ierr);
            ierr = MatRestoreRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
        }

        ierr = VecRestoreArray(vec_condfu,&condf);CHKERRQ(ierr);
        ierr = VecRestoreArray(vec_diffront,&difb);CHKERRQ(ierr);
        ierr = VecRestoreArray(vec_convecfrontn,&convecb);CHKERRQ(ierr);

        ierr = VecAssemblyBegin(vec_Run);CHKERRQ(ierr);
        ierr = VecAssemblyEnd(vec_Run);CHKERRQ(ierr);

        ierr = VecRestoreArray(vec_solun,&phivcnu);CHKERRQ(ierr);
        ierr = VecRestoreArray(vec_bun,&termb);CHKERRQ(ierr);

        //Calculo de la u predictora
        ierr = VecGetArray(vec solun,&phivcnu);CHKERRQ(ierr);
        ierr = VecGetArray(vec_Run,&Ru_n);CHKERRQ(ierr);
        ierr = VecGetArray(vec_Run1,&Ru_n1);CHKERRQ(ierr);
        ierr = VecGetArray(vec_volumen,&volumenvc);CHKERRQ(ierr);

        for (i=0;i<n_vc;i++)
        {
            up=phivcnu[i]+deltat/densidad/volumenvc[i]*(3.0/2.0*Ru_n[i]-1.0/2.0*Ru_n1[i]);
            ierr = VecSetValues(vec_up,1,&i,&up,INSERT_VALUES);CHKERRQ(ierr);
        }

        ierr = VecRestoreArray(vec_solun,&phivcnu);CHKERRQ(ierr);
        ierr = VecRestoreArray(vec_Run,&Ru_n);CHKERRQ(ierr);
        ierr = VecRestoreArray(vec_Run1,&Ru_n1);CHKERRQ(ierr);
        ierr = VecRestoreArray(vec_volumen,&volumenvc);CHKERRQ(ierr);

        ierr = VecAssemblyBegin(vec_up);CHKERRQ(ierr);
        ierr = VecAssemblyEnd(vec_up);CHKERRQ(ierr);

        //Calculo del Rv en n-1
        ierr = VecGetArray(vec_solvn1,&phivcn1);CHKERRQ(ierr);
        ierr = VecGetArray(vec_bvn1,&termb);CHKERRQ(ierr);

        ierr = VecGetArray(vec_condfu,&condf);CHKERRQ(ierr);
        ierr = VecGetArray(vec_diffront,&difb);CHKERRQ(ierr);
        ierr = VecGetArray(vec convecfrontn1,&convecb);CHKERRQ(ierr);
```

```
446
447         for (i=0;i<n_vc;i++)
448         {
449             ierr = MatGetRow(matriz_difusivo,i,PETSC_NULL,PETSC_NULL,&Df);CHKERRQ(ierr);
450             ierr = MatGetRow(matriz_convectivon1,i,PETSC_NULL,PETSC_NULL,&Ff);CHKERRQ(ierr);
451             ierr = MatGetRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
452
453             Rv=0;
454
455             if(vecinos[2]>n_vc)
456             {
457                 for (j=0;j<2;j++)
458                 {
459                     k=vecinos[j];
460                     Rv=Rv+Df[j]*(phivcn1[k]-phivcn1[i])-MAX(Ff[j],0)*phivcn1[i]+MAX(-Ff[j],0)*phivcn1[k];
461                 }
462                 Rv=Rv+termb[i];
463                 if(condf[i]==1)
464                 {
465                     Rv=Rv-difb[i]*phivcn1[i];
466                 }
467                 if(condf[i]==2)
468                 {
469                     Rv=Rv-convecb[i]*phivcn1[i];
470                 }
471             }
472
473             else
474             {
475                 for (j=0;j<3;j++)
476                 {
477                     k=vecinos[j];
478                     Rv=Rv+Df[j]*(phivcn1[k]-phivcn1[i])-MAX(Ff[j],0)*phivcn1[i]+MAX(-Ff[j],0)*phivcn1[k];
479                 }
480                 Rv=Rv+termb[i];
481             }
482
483             ierr = VecSetValues(vec_Rvn1,1,&i,&Rv,INSERT_VALUES);CHKERRQ(ierr);
484
485             ierr = MatRestoreRow(matriz_difusivo,i,PETSC_NULL,PETSC_NULL,&Df);CHKERRQ(ierr);
486             ierr = MatRestoreRow(matriz convectivon1,i,PETSC NULL,PETSC NULL,&Ff);CHKERRQ(ierr);
487             ierr = MatRestoreRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
488         }
489
490         ierr = VecAssemblyBegin(vec_Rvn1);CHKERRQ(ierr);
491         ierr = VecAssemblyEnd(vec_Rvn1);CHKERRQ(ierr);
492
493         ierr = VecRestoreArray(vec_solvn1,&phivcn1);CHKERRQ(ierr);
494         ierr = VecRestoreArray(vec_bvn1,&termb);CHKERRQ(ierr);
495         ierr = VecRestoreArray(vec_convecfrontn1,&convecb);CHKERRQ(ierr);
496
497         //Calculo del Rv en n
498
499         ierr = VecGetArray(vec_solvn,&phivcnv);CHKERRQ(ierr);
500         ierr = VecGetArray(vec_bvn,&termb);CHKERRQ(ierr);
501         ierr = VecGetArray(vec_convecfrontn,&convecb);CHKERRQ(ierr);
502         for (i=0;i<n_vc;i++)
503         {
504             ierr = MatGetRow(matriz_difusivo,i,PETSC_NULL,PETSC_NULL,&Df);CHKERRQ(ierr);
505             ierr = MatGetRow(matriz_convectivon,i,PETSC_NULL,PETSC_NULL,&Ff);CHKERRQ(ierr);
506             ierr = MatGetRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
507
508             Rv=0;
509
510             if(vecinos[2]>n_vc)
511             {
```

```
512                 for (j=0;j<2;j++)
513                 {
514                     k=vecinos[j];
515                     Rv=Rv+Df[j]*(phivcnv[k]-phivcnv[i])-MAX(Ff[j],0)*phivcnv[i]+MAX(-Ff[j],0)*phivcnv[k];
516                 }
517                 Rv=Rv+termb[i];
518                 if(condf[i]==1)
519                 {
520                     Rv=Rv-difb[i]*phivcnv[i];
521                 }
522                 if(condf[i]==2)
523                 {
524                     Rv=Rv-convecb[i]*phivcnv[i];
525                 }
526             }
527
528             else
529             {
530                 for (j=0;j<3;j++)
531                 {
532                     k=vecinos[j];
533                     Rv=Rv+Df[j]*(phivcnv[k]-phivcnv[i])-MAX(Ff[j],0)*phivcnv[i]+MAX(-Ff[j],0)*phivcnv[k];
534                 }
535                 Rv=Rv+termb[i];
536             }
537
538             ierr = VecSetValues(vec_Rvn,1,&i,&Rv,INSERT_VALUES);CHKERRQ(ierr);
539
540             ierr = MatRestoreRow(matriz_difusivo,i,PETSC_NULL,PETSC_NULL,&Df);CHKERRQ(ierr);
541             ierr = MatRestoreRow(matriz_convectivon,i,PETSC_NULL,PETSC_NULL,&Ff);CHKERRQ(ierr);
542             ierr = MatRestoreRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
543         }
544
545         ierr = VecRestoreArray(vec_condfu,&condf);CHKERRQ(ierr);
546         ierr = VecRestoreArray(vec_diffront,&difb);CHKERRQ(ierr);
547         ierr = VecGetArray(vec_convecfrontn,&convecb);CHKERRQ(ierr);
548
549         ierr = VecAssemblyBegin(vec_Rvn);CHKERRQ(ierr);
550         ierr = VecAssemblyEnd(vec_Rvn);CHKERRQ(ierr);
551
552         ierr = VecRestoreArray(vec_solvn,&phivcnv);CHKERRQ(ierr);
553         ierr = VecRestoreArray(vec_bvn,&termb);CHKERRQ(ierr);
554
555         //Calculo de la v predictora
556         ierr = VecGetArray(vec_solvn,&phivcnv);CHKERRQ(ierr);
557         ierr = VecGetArray(vec_Rvn,&Rv_n);CHKERRQ(ierr);
558         ierr = VecGetArray(vec_Rvn1,&Rv_n1);CHKERRQ(ierr);
559         ierr = VecGetArray(vec_volumen,&volumenvc);CHKERRQ(ierr);
560
561         for (i=0;i<n_vc;i++)
562         {
563             vp=phivcnv[i]+deltat/densidad/volumenvc[i]*(3.0/2.0*Rv_n[i]-1.0/2.0*Rv_n1[i]);
564             ierr = VecSetValues(vec_vp,1,&i,&vp,INSERT_VALUES);CHKERRQ(ierr);
565         }
566
567         ierr = VecRestoreArray(vec_solvn,&phivcnv);CHKERRQ(ierr);
568         ierr = VecRestoreArray(vec_Rvn,&Rv_n);CHKERRQ(ierr);
569         ierr = VecRestoreArray(vec_Rvn1,&Rv_n1);CHKERRQ(ierr);
570         ierr = VecRestoreArray(vec_volumen,&volumenvc);CHKERRQ(ierr);
571
572         ierr = VecAssemblyBegin(vec_vp);CHKERRQ(ierr);
573         ierr = VecAssemblyEnd(vec_vp);CHKERRQ(ierr);
574
575         //Calculo de la presion
576
577         //Calcular los terminos difusivos para la presion
```

```
578          temporal=1; //No va gamma=temporal=1
579          dp.calcular_D(temporal,gm,&matriz_difusivop,&vec_diffrontp);
580
581          ierr = VecGetArray(vec_diffrontp,&difb);CHKERRQ(ierr);
582          ierr = VecGetArray(vec_condfp,&condf);CHKERRQ(ierr);
583
584          for (i=0;i<n_vc;i++)
585          {
586              ierr = MatGetRow(matriz_difusivop,i,PETSC_NULL,&cols,&Df);CHKERRQ(ierr);
587              ierr = MatGetRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
588
589              colcoef[0]=i;colcoef[1]=cols[0];colcoef[2]=cols[1];
590              if(vecinos[2]>n_vc)
591              {
592                  value[1]=Df[0];
593                  value[2]=Df[1];
594                  if(condf[i]==1)
595                  {value[0]=-(value[1]+value[2]+difb[i]);}
596                  if(condf[i]==-1)
597                  {value[0]=-(value[1]+value[2]);}
598
599                  ierr = MatSetValues(matriz_coef,1,&i,3,colcoef,value,INSERT_VALUES);CHKERRQ(ierr);
600              }
601              else
602              {
603                  colcoef[3]=cols[2];
604                  value[1]=Df[0];
605                  value[2]=Df[1];
606                  value[3]=Df[2];
607                  value[0]=-(value[1]+value[2]+value[3]);
608                  ierr = MatSetValues(matriz_coef,1,&i,4,colcoef,value,INSERT_VALUES);CHKERRQ(ierr);
609              }
610
611              ierr = MatRestoreRow(matriz_difusivo,i,PETSC_NULL,&cols,&Df);CHKERRQ(ierr);
612              ierr = MatRestoreRow(matriz_vecinos,i,PETSC_NULL,PETSC_NULL,&vecinos);CHKERRQ(ierr);
613          }
614          ierr = VecRestoreArray(vec_diffrontp,&difb);CHKERRQ(ierr);
615          ierr = VecRestoreArray(vec_condfp,&condf);CHKERRQ(ierr);
616
617          //Ensamblamos la matriz de coeficientes
618          ierr = MatAssemblyBegin(matriz_coef,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
619          ierr = MatAssemblyEnd(matriz_coef,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
620
621          //Parametros para el solver
622          KSPCreate(PETSC_COMM_WORLD,&ksp);
623          KSPSetOperators(ksp,matriz_coef,matriz_coef,DIFFERENT_NONZERO_PATTERN);
624          KSPSetType(ksp,KSPGMRES);
625          KSPGetPC(ksp,&pc);
626          PCSetType(pc,PCGAMG);
627          KSPSetFromOptions(ksp);
628          KSPSetUp(ksp);
629
630          grup.calcular_vcara(gm,fu,vec_up,&matriz_upf);
631          grvp.calcular_vcara(gm,fv,vec_vp,&matriz_vpf);
632
633          int niter=0;
634          error=1;
635
636          //Comienzan las iteraciones para el calculo de la presion
637
638          while (error>1*10e-6 && niter<30)
639          {
640              grp.calcular_grad(gm,fp,vec_soluantp,&matriz_gradp);
641              ibp.calcular_bp(gm,fp,matriz_difusivop,vec_diffrontp,matriz_gradp,matriz_upf,matriz_vpf,
densidad,deltat,&vec_bp);
642
```

```
643             KSPSolve(ksp,vec_bp,vec_solucionp);
644
645             ierr = VecGetArray(vec_solucionp,&phivc);CHKERRQ(ierr);
646             ierr = VecGetArray(vec_soluantp,&phivc0);CHKERRQ(ierr);
647
648             error=0;
649             for (i=0;i<n_vc;i++)
650             {
651                 temporal=sqrt(pow(phivc[i]-phivc0[i],2));
652                 error=MAX(error,temporal);
653             }
654
655             for (i=0;i<n_vc;i++)
656             {
657                 phi0=phivc[i];
658                 ierr = VecSetValues(vec_soluantp,1,&i,&phi0,INSERT_VALUES);CHKERRQ(ierr);
659             }
660
661             ierr = VecAssemblyBegin(vec_soluantp);CHKERRQ(ierr);
662             ierr = VecAssemblyEnd(vec_soluantp);CHKERRQ(ierr);
663
664             niter++;
665         }
666
667         for (i=0;i<n_vc;i++)
668         {
669             phi0=0;
670             ierr = VecSetValues(vec_soluantp,1,&i,&phi0,INSERT_VALUES);CHKERRQ(ierr);
671         }
672
673         ierr = VecAssemblyBegin(vec_soluantp);CHKERRQ(ierr);
674         ierr = VecAssemblyEnd(vec_soluantp);CHKERRQ(ierr);
675
676         grp.calcular_grad(gm,fp,vec_solucionp,&matriz_gradp);
677
678         ierr = VecGetArray(vec_up,&u_p);CHKERRQ(ierr);
679         ierr = VecGetArray(vec_vp,&v_p);CHKERRQ(ierr);
680
681         for (i=0;i<n_vc;i++)
682         {
683             ierr = MatGetRow(matriz_gradp,i,PETSC_NULL,PETSC_NULL,&gradp);CHKERRQ(ierr);
684             u=u_p[i]-deltat/densidad*gradp[0];
685             v=v_p[i]-deltat/densidad*gradp[1];
686             ierr = VecSetValues(vec_solucionu,1,&i,&u,INSERT_VALUES);CHKERRQ(ierr);
687             ierr = VecSetValues(vec_solucionv,1,&i,&v,INSERT_VALUES);CHKERRQ(ierr);
688             ierr = MatRestoreRow(matriz_gradp,i,PETSC_NULL,PETSC_NULL,&gradp);CHKERRQ(ierr);
689         }
690
691         ierr = VecRestoreArray(vec_up,&u_p);CHKERRQ(ierr);
692         ierr = VecRestoreArray(vec_vp,&v_p);CHKERRQ(ierr);
693
694         ierr = VecAssemblyBegin(vec_solucionu);CHKERRQ(ierr);
695         ierr = VecAssemblyEnd(vec_solucionu);CHKERRQ(ierr);
696
697         ierr = VecAssemblyBegin(vec_solucionv);CHKERRQ(ierr);
698         ierr = VecAssemblyEnd(vec_solucionv);CHKERRQ(ierr);
699
700         titer++;
701         contador++;
702         if(contador==500)
703         {
704             rta.imprimir(gm,vec_solucionu,vec_solucionv,vec_solucionp,titer);
705             contador=0;
706         }
707
708         ierr = KSPDestroy(&ksp);CHKERRQ(ierr);
```

```
709
710         ierr = VecGetArray(vec_solucionu,&phivc);CHKERRQ(ierr);
711         ierr = VecGetArray(vec_solun,&phivc0);CHKERRQ(ierr);
712
713         for (i=0;i<n_vc;i++)
714         {
715             phi=phivc[i];
716             phi0=phivc0[i];
717             ierr = VecSetValues(vec_solun,1,&i,&phi,INSERT_VALUES);CHKERRQ(ierr);
718         }
719
720         ierr = VecAssemblyBegin(vec_solun);CHKERRQ(ierr);
721         ierr = VecAssemblyEnd(vec_solun);CHKERRQ(ierr);
722
723         ierr = VecAssemblyBegin(vec_solun1);CHKERRQ(ierr);
724         ierr = VecAssemblyEnd(vec_solun1);CHKERRQ(ierr);
725
726         ierr = VecGetArray(vec_solucionv,&phivc);CHKERRQ(ierr);
727         ierr = VecGetArray(vec_solvn,&phivc0);CHKERRQ(ierr);
728
729         for (i=0;i<n_vc;i++)
730         {
731             phi=phivc[i];
732             phi0=phivc0[i];
733             ierr = VecSetValues(vec_solvn,1,&i,&phi,INSERT_VALUES);CHKERRQ(ierr);
734             ierr = VecSetValues(vec_solvn1,1,&i,&phi0,INSERT_VALUES);CHKERRQ(ierr);
735         }
736
737         ierr = VecAssemblyBegin(vec_solvn);CHKERRQ(ierr);
738         ierr = VecAssemblyEnd(vec_solvn);CHKERRQ(ierr);
739
740         ierr = VecAssemblyBegin(vec_solvn1);CHKERRQ(ierr);
741         ierr = VecAssemblyEnd(vec_solvn1);CHKERRQ(ierr);
742
743         err=0;
744         ierr = VecGetArray(vec_solun,&phivcnu);CHKERRQ(ierr);
745         ierr = VecGetArray(vec_solun1,&phivcn1);CHKERRQ(ierr);
746         ierr = VecGetArray(vec_solvn,&phivc);CHKERRQ(ierr);
747         ierr = VecGetArray(vec_solvn1,&phivc0);CHKERRQ(ierr);
748
749         for (i=0;i<n_vc;i++)
750         {
751             temporal=MAX(sqrt(pow(phivc[i]-phivc0[i],2)),sqrt(pow(phivcnu[i]-phivcn1[i],2)));
752             err=MAX(error,temporal);
753         }
754
755         ierr = VecRestoreArray(vec_solun,&phivcnu);CHKERRQ(ierr);
756         ierr = VecRestoreArray(vec_solun1,&phivcn1);CHKERRQ(ierr);
757         ierr = VecRestoreArray(vec_solvn,&phivc);CHKERRQ(ierr);
758         ierr = VecRestoreArray(vec_solvn1,&phivc0);CHKERRQ(ierr);
759
760     }
761
762     rta.imprimir(gm,vec_solucionu,vec_solucionv,vec_solucionp,titer);
763
764     ierr = VecDestroy(&vec_solucionu);CHKERRQ(ierr);  ierr = VecDestroy(&vec_solucionv);CHKERRQ(ierr);
765     ierr = VecDestroy(&vec_solun);CHKERRQ(ierr);  ierr = VecDestroy(&vec_solvn);CHKERRQ(ierr);
766     ierr = VecDestroy(&vec_Run);CHKERRQ(ierr);  ierr = VecDestroy(&vec_Rvn);CHKERRQ(ierr);
767     ierr = VecDestroy(&vec_up);CHKERRQ(ierr);  ierr = VecDestroy(&vec_vp);CHKERRQ(ierr);
768     ierr = VecDestroy(&vec_solucionp);CHKERRQ(ierr);  ierr = VecDestroy(&vec_soluantp);CHKERRQ(ierr);
769     ierr = MatDestroy(&matriz_coef);CHKERRQ(ierr);
770
771     return ierr;
772 }
```