MICSc: Un Código Paralelo de Dinámica de Fluidos Computacional Basado en PETSc

Proyecto Final de Carrera

Autor: Víctor González Cortés

Dirigido por José E. Román

Escuela Técnica Superior de Ingeniería Informática,

1 de febrero de 2011

Índice general

Agradecimientos

Doy gracias a mi família y amigos por el apoyo incondicional durante toda mi etapa de estudiante y porque sin ellos nada de esto habría sido posible.

También agradezco principalmente a José E. Román y a Guillermo Palau que me hayan dado la oportunidad y todo el apoyo y material necesario para la realización del proyecto, así como a Ana Cubero por permitirme formar parte de su trabajo y por su disposición y ayuda durante todo el trabajo.

Además, quiero dar las gracias, tanto al Vicerrectorado de Investigación de la Universidad Politécnica de Valencia y al Ministerio de Ciencia e Innovación por promover y financiar el proyecto de investigación RHELES en que se enmarca este proyecto final de carrera, como a todo el grupo GRyCAP por su apoyo, y en concreto a Andrés Tomás y Eloy Romero por su ayuda para este trabajo.

Capítulo 1

Introducción

Este trabajo se sitúa en el contexto del proyecto de investigación RHELES, que tiene como principal objeto de análisis e investigación la resolución de sistemas de ecuaciones mediante algoritmos numéricos y su aplicación en el campo de la Dinámica de Fluidos Computacional (CFD por sus siglas en inglés, *Computational Fluid Dynamics*); concretamente, se centra en la resolución iterativa de las ecuaciones de Navier-Stokes que rigen el comportamiento del fluido.

En este capítulo se realiza una breve introducción a la CFD, así como al paralelismo y las librerías numéricas, y finalmente se presentan los objetivos tanto del proyecto de investigación RHELES como del proyecto final de carrera.

1.1. Dinámica de fluidos computacional

La dinámica de fluidos computacional, consistente en la resolución y el análisis de problemas relacionados con el flujo de fluidos mediante el uso de tecnología computacional, tiene su comienzo en los años 60, cuando se reducía al ámbito académico y su visibilidad fuera de éste era prácticamente nula. Sin embargo, en las últimas décadas ha experimentado una gran evolución hasta convertirse en una de las herramientas industriales más importantes para el diseño y análisis de dispositivos de ingeniería.

Las causas principales de esta rápida evolución se deben, por un lado, al aumento de la potencia y el abaratamiento de los recursos computacionales; y, por otra parte, al desarrollo de algoritmos numéricos para resolver de manera eficiente los problemas que se plantean en el contexto de la ingeniería. De esta manera, se pueden destacar como hitos importantes en el desarrollo de la CFD, la publicación en 1972 del primer método segregado para la resolución de ecuaciones de flujo incompresible [1], o la aparición de aplicaciones comerciales como FLUENT y CFX entre otros.

Posteriormente, en la última década, la CFD está experimentando un

nuevo desarrollo, en esta ocasión, propiciado por el aumento de la capacidad computacional y el progreso en los métodos iterativos para resolver sistemas de ecuaciones lineales. Además, la mayor complejidad en el patrón de llenado de la matriz de coeficientes, así como el aumento de la exigencia debido al mayor uso industrial de la CFD, ha propiciado un replanteamiento de los algoritmos tradicionales. De esta manera, se requieren códigos eficientes a la par que robustos, capaces de resolver grandes problemas numéricos en el menor tiempo posible, y minimizando el número de parámetros libres introducidos por el usuario.

1.2. Paralelismo

El paralelismo o computación paralela se define como la ejecución concurrente de una misma tarea mediante diferentes unidades de proceso. Esta concurrencia se puede dar en distintos escenarios: dentro del propio procesador, en un único computador o incluso en Internet. En el propio procesador existe la posibilidad de incluir varias unidades de cálculo de manera que es posible ejecutar de manera simultánea diversas operaciones básicas. Cuando la computación paralela se da en Internet (o en cualquier otra red de interconexión local o global), aparece el concepto de tecnología *grid*, que consiste en la unión de computadores en diferentes puntos del planeta conectados a través de una red de forma que se comportan como un único elemento de cálculo. Por último, el caso en que el paralelismo se produce en un mismo computador es precisamente el que se da a lo largo de todo este trabajo y presenta diferentes variantes.

El paralelismo a nivel de computador consiste en la integración de diferentes procesadores en un único sistema y existen principalmente dos paradigmas. En primer lugar se da el paradigma de memoria compartida, en el que todos los procesadores tienen acceso a la misma memoria y por otro lado existe el paradigma de memoria distribuida en el que cada procesador tiene acceso exclusivo a una zona de memoria propia. En este segundo paradigma, que es el que se utiliza de manera exclusiva en este proyecto, se utiliza la interfaz de paso de mensajes (MPI por sus siglas en inglés, *Message Passing Interface* [2]). Este estándar define las funciones necesarias para la comunicación entre los distintos procesadores, ya que siendo la memoria distribuida se hace necesario algún mecanismo para la transmisión de datos entre diferentes procesadores. Las funciones principales que define MPI se pueden clasificar en tres grupos:

- Inicializar, administrar y finalizar comunicaciones:
    - `MPI_Init`: Inicia una sesión MPI. Se debe llama siempre antes de cualquier otra función de MPI.

  - `MPI_Finalize`: Termina una sesión MPI. Debe ser la última llamada a MPI del programa.
  - `MPI_Comm_size`: Obtiene el número total de procesos.
  - `MPI_Comm_rank`: Obtiene el identificador (*rank*) del proceso.

- Comunicaciones punto a punto:

  - `MPI_Send`: Envía un dato a otro proceso.
  - `MPI_Recv`: Recibe un dato de otro proceso.

  Para que se produzca la comunicación ambos procesos deben llamar a las respectivas funciones, dándose un bloqueo si alguna de las dos no se produce.

- Comunicaciones colectivas:

  - `MPI_Bcast`: Difunde un dato de un proceso a todos los demas.
  - `MPI_Scatter`: Distribuye desde un proceso p un conjunto de n datos entre los n procesadores (un dato por procesador). Equivale a realizar desde p, por cada proceso, una llamada a `MPI_Send` con el dato y destino correspondientes.
  - `MPI_Gather`: Recibe en el procesador p un conjunto de n datos desde los n procesadores (un dato por procesador). Es la operación inversa a `MPI_Scatter` y equivale a realizar en cada proceso una llamada a `MPI_Send` con el dato correspondiente y con destino el procesador p.
  - `MPI_Allgather`: Recibe en todos y cada uno de los procesadores, n datos desde los n procesadores (un dato por procesador). Es equivalente a realizar una llamada a `MPI_Gather` en cada uno de los procesadores.

Por último, también es interesante conocer las magnitudes que nos permiten medir el comportamiento paralelo de una cierta aplicación. Es por esto que definiremos el concepto de *speedup* o aceleración y eficiencia.

La aceleración se define como el factor de ganancia de un programa paralelo con respecto a la versión secuencial:

$$S_p = \frac{t_1}{t_p},$$

donde t_1 es el tiempo del programa secuencial y t_p es el tiempo del programa paralelo. Es común no disponer de la versión secuencial del algoritmo, por lo que generalmente se realiza la comparación con respecto al programa paralelo ejecutado con un solo procesador. También es importante destacar que el valor de esta magnitud viene condicionado por el número de procesos y se pueden dar las siguientes situaciones:

- **Speed-down** ($S_p < 1$): El algoritmo paralelo es más lento que el secuencial.
- **Speed-up sublineal** ($1 < S_p < p$): El algoritmo paralelo es más rápido que el secuencial, pero no aprovecha al máximo la capacidad de los procesadores.
- **Speed-up lineal** ($S_p = 1$): El algoritmo paralelo es lo más rápido posible.
- **Speed-up superlineal** ($S_p > p$): Teóricamente esta situación no es posible. Puede deberse a una situación anómala en la que el algoritmo paralelo ejecute menos instrucciones; o bien, en el caso de que la diferencia entre S_p y p sea pequeña, a factores no contemplados tales como la memoria caché.

Por último, para disponer de una medida que no dependa del número de procesadores, se define la eficiencia:

$$E_p = \frac{S_p}{p},$$

que suele expresarse en tanto por cien o directamente en tanto por 1. Es posible realizar una clasificación análoga a la de la aceleración para la eficiencia sublineal, lineal y superlineal. Ambas medidas son las que utilizaremos en este proyecto para el análisis de resultados.

1.3. Librerías numéricas

Los problemas basados en la resolución de sistemas de ecuaciones lineales obtenidos a partir de la discretización de ecuaciones de derivadas parciales hacen necesario el uso de librerías numéricas para dicha resolución. El estándar de facto para este tipo de librerías, cuya definición se inició en 1979 y que define la sintaxis y semántica de las operaciones básicas del álgebra lineal es BLAS (por sus siglas en inglés, *Basic Linear Algebra Subprograms* [3]). Según BLAS, estas operaciones se dividen en tres niveles:

- **Nivel 1:** Operaciones vectoriales del tipo $y = \alpha x + y$, así como productos escalares, normas y demás operaciones que incluyen únicamente vectores y magnitudes escalares.
- **Nivel 2:** Operaciones matriz-vector de la forma $y = \alpha Ax + \beta y$ junto con la resolución de sistemas de ecuaciones triangulares y cualquier otra operación entre matrices y vectores.
- **Nivel 3:** Operaciones matriz-matriz del tipo $C = \alpha AB + \beta C$ y, en general, cualquier operación básica que incluya la multiplicación de matrices.

Por otro lado, la librería LAPACK (*Linear Algebra PACkage* [4]), que surgió en 1992, depende de BLAS y proporciona funciones para la resolución de sistemas de ecuaciones lineales, problemas de mínimos cuadrados y de valores propios entre otros. Generalmente, las operaciones del nivel 3 de BLAS suelen tener coste cúbico, por lo que para tamaños relativamente grandes de matriz se requiere de paralelismo para realizar las operaciones. Para ello se creó ScaLAPACK (*Scalable LAPACK*), que esencialmente consiste en la implementación paralela de un subconjunto de rutinas de LAPACK mediante MPI. Sin embargo, tanto LAPACK como ScaLAPACK trabajan con matrices densas, por lo que el tamaño del problema está limitado y se hace necesario el uso de algoritmos e implementaciones para matrices dispersas. De esta manera es posible alcanzar tamaños de problema con millones de incógnitas como el que se plantea en este trabajo.

Por último, PETSc (Portable, Extensible Toolkit for Scientific Computation [5]), que nació en 1991, es una librería que actualmente se construye sobre BLAS, LAPACK y un conjunto muy amplio y configurable de librerías, y que hace uso del estándar de paso de mensajes MPI. De esta manera, proporciona estructuras de datos y funciones para la resolución paralela de aplicaciones científicas modeladas mediante ecuaciones de derivadas parciales. El desarrollo de este proyecto se basa fundamentalmente en el uso de PETSc, por lo que el Capítulo 3 se dedica íntegramente a la descripción de esta librería.

1.4. Objetivos

Desde los comienzos del proyecto de investigación RHELES (Rural and Hydraulic Engineering Large Eddy Simulation), el objetivo principal era desarrollar una aplicación científica que permitiera resolver las ecuaciones de Navier-Stokes mediante la ejecución de algoritmos numéricos y haciendo uso de computación paralela. De esta manera, se inició el proyecto con la tarea de desarrollar esta aplicación por completo. Sin embargo, al poco tiempo se tuvo constancia de la aplicación MICSc, desarrollada por A. Cubero en la Universidad de Zaragoza, y que perseguía esos mismos objetivos. De esta manera, se adoptó MICSc como punto de partida para el desarrollo y la investigación y los objetivos principales de RHELES se concretaron en:

1. Definición de los casos para testar el código LES con la obtención de datos experimentales fiables de trabajos presentados en revistas de reconocido prestigio o con la utilización de datos experimentales obtenidos en el laboratorio de Ingeniería Hidráulica y Riego Localizado de la E.T.S.I. Agrónomos.

2. Validación del programa básico para la resolución de los casos planteados en este proyecto de investigación.

3. Desarrollo de nuevas opciones en el código, como la implementación de la aplicación de "Inmersed Boudary Method", la inclusión de una ecuación de transporte de contaminantes o el estudio del movimiento de partículas en el interior del flujo relacionado con el transporte de sedimentos.

Por otra parte, con motivo de la presentación del trabajo realizado en el grupo de investigación como proyecto final de carrera, se debieron concretar también unos objetivos académicos que se adecuaran tanto al contexto de la investigación como a la presentación del proyecto final de carrera. Son los siguientes:

1. Optimización y refactorización del código MICSc adoptado. Este objetivo incluye tareas como la inclusión de trazabilidad de errores, la modificación de estructuras de datos e implementaciones para la adopción de buenas prácticas de programación.

2. Medida de prestaciones para, al menos, un caso de estudio. Se deberán realizar mediciones de tiempos para la resolución del problema con diferentes *solvers* lineales, distinto número de procesadores, etc, obteniendo medidas de aceleración y eficiencia que caractericen el comportamiento paralelo del código.

3. Implementación de nuevas funcionalidades y/o modelos que sean requeridos por los miembros del grupo y principales usuarios del código, tanto de la E.T.S.I Agrónomos como del Área de Mecánica de Fluidos de la Universidad de Zaragoza.

4. Generación de documentación. Esta tarea incluye documentación tanto del código fuente existente y desarrollado, como de los desarrollos y decisiones tomadas a lo largo del proyecto con el fin de facilitar la inclusión de nuevos miembros al grupo de investigación.

Capítulo 2

Dinámica de fluidos computacional

En este capítulo definiremos las ecuaciones que gobiernan el flujo para conocer qué parámetros influyen en su comportamiento. Analizaremos los tipos de fluidos existentes y qué consecuencias tiene sobre las ecuaciones considerar un tipo de flujo específico. Además, entraremos en más detalle en los flujos que más relevancia tienen en el contexto del proyecto. Por último, abordaremos los diferentes métodos para la resolución de sistemas de ecuaciones que se utilizan tanto en este proyecto, como en el campo de la dinámica de fluidos computacional en general.

2.1. Ecuaciones de Navier-Stokes

Las ecuaciones que gobiernan el flujo y que nos permiten simular su comportamiento se conocen como ecuaciones de Navier-Stokes y están basadas en el principio de conservación, tanto de la masa como del momento. Este principio relaciona el cambio de una magnitud en un determinado volumen de control con las fuerzas o efectos que actúan sobre el mismo.

De esta manera, aplicando el principio de conservación a la masa, y teniendo en cuenta que la masa no se crea ni se destruye, obtenemos

$$\frac{dm}{dt} = 0$$

Es posible trasformar esta ecuación en otra de derivadas parciales, dependiente de la velocidad y densidad del fluido para un sistema de coordenadas cartesiano de la forma

$$\frac{\partial \rho}{\partial t} + \frac{\partial(\rho u_i)}{\partial x_i} = 0$$

donde x_i (i= 1,2,3) (o x,y,z) son las coordenadas cartesianas, u_i (o u_x, u_y, u_z) son las componentes cartesianas del vector velocidad $\vec{v}$ y ρ es la densidad del fluido. Esta ecuación también se conoce como *ecuación de conservación.*

Por otro lado, aplicando el principio de conservación al momento, y utilizando la segunda ley de Newton, se obtiene que

$$\frac{d(m\vec{v})}{dt} = \Sigma \mathbf{f}$$

donde $\boldsymbol{f}$ son las fuerzas que actúan sobre el flujo.

De la misma manera que para la masa, es posible obtener una ecuación de derivadas parciales para un sistema de coordenadas cartesiano obteniendo

$$\frac{\partial(\rho u_i)}{\partial t} + \frac{\partial(\rho u_j u_i)}{\partial x_j} = \frac{\partial \tau_{ij}}{\partial x_j} - \frac{\partial p}{\partial x_i} + \rho g_i$$

donde g_i es la componente de la aceleración de la gravedad $\vec{g}$ en la dirección de la coordenada cartesiana x_i. El término τ_{ij} se conoce como *tensor de esfuerzos viscosos* y se obtiene de la siguiente manera:

$$\tau_{ij} = \mu\Big\{\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{2}{3}\delta_{ij}\frac{\partial u_l}{\partial x_l}\Big\}$$

siendo μ la viscosidad del fluido (ver 2.2.5) y δ_{ij} la delta de Kronecker. Esta ecuación también se conoce como *ecuación del momento*.

Así, pudiendo añadir una tercera ecuación de manera análoga para el caso en que se desee simular la temperatura (T), donde κ es el coeficiente de difusión y el calor específico c_P se ha supuesto constante, obtenemos el conjunto de ecuaciones de Navier-Stokes:

$$\frac{\partial \rho}{\partial t} + \frac{\partial(\rho u_i)}{\partial x_i} = 0 \tag{2.1}$$

$$\frac{\partial(\rho u_i)}{\partial t} + \frac{\partial(\rho u_j u_i)}{\partial x_j} = \frac{\partial \tau_{ij}}{\partial x_j} - \frac{\partial p}{\partial x_i} + \rho g_i \tag{2.2}$$

$$\frac{\partial(\rho T)}{\partial t} + \frac{\partial(\rho u_j T)}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\kappa \frac{\partial T}{\partial x_j}\right) \tag{2.3}$$

2.2. Clasificación de los fluidos

En función las diferentes características de los fluidos, éstos pueden clasificarse de diferentes maneras. Estas clasificaciones no son útiles únicamente para la simple caracterización de los fluidos, sino que en ciertos casos tienen una repercusión directa sobre las ecuaciones que gobiernan el flujo. De esta manera es posible clasificar un fluido de modo que permite la simplificación, en mayor o menor medida, de las ecuaciones y así facilitar la resolución global del problema.

En los siguientes apartados analizaremos las clasificaciones de fluidos más relevantes para nuestro contexto y, cuando sea posible, sus consecuencias en las ecuaciones de conservación en función de distintas propiedades del fluido.

2.2.1. Subsónico / Transónico / Supersónico / Hipersónico

El número de Mach (Ma) es una magnitud adimensional que relaciona el módulo de la velocidad del fluido en un determinado medio con respecto de la velocidad del sonido en ese mismo medio, de manera que aquel fluido cuya velocidad sea exactamente igual a la velocidad del sonido tendrá un número de Mach de 1. De esta manera, en función del número de Mach, los fluidos pueden caracterizarse de la siguiente manera:

- **Subsónico:** $0 < Ma < 0.75$
- **Transónico:** $0.75 < Ma < 1.2$
- **Supersónico:** $1.2 < Ma < 5$
- **Hipersónico:** $5 < Ma$

Cabe destacar que la velocidad del fluido es una de las magnitudes que debe ser resuelta en la simulación. Sin embargo, a pesar de no conocer el valor exacto de dicha velocidad, es posible determinar de manera intuitiva el régimen en que se encuentra el fluido simplemente por la naturaleza del mismo y las condiciones que se dan en su entorno.

Por último, aunque la clasificación del fluido en función de su velocidad no afecta de manera directa a las ecuaciones que lo gobiernan, sí que es posible derivar ciertas conclusiones a partir de ésta, tal y como se detalla en el siguiente apartado.

2.2.2. Compresible / Incompresible

En dinámica de fluidos computacional, se determina si un fluido es compresible o no en función de su número de Mach, definido tal y como se comenta en el apartado anterior. De esta manera, un fluido es incompresible cuando su número de Mach es pequeño ($Ma < 0.3$), mientras que en otro caso será compresible.

Cabe destacar que, a nivel general, todos los fluidos son compresibles; es decir, cambios en la presión y/o temperatura producen cambios en la densidad de dicho fluido. Además, la compresibilidad es una propiedad característica de cada fluido y no depende de su velocidad. Sin embargo, se realiza esta clasificación debido a que el flujo de un fluido compresible a bajo número de Mach es esencialmente incompresible; esto es, los cambios en la densidad son despreciables.

De esta manera, la ventaja en cuanto a la resolución del problema que aporta un fluido compresible con respecto de uno incompresible es que se puede suprimir la derivada parcial de la densidad con respecto del tiempo en las ecuaciones de conservación. Esto aporta una simplificación clara en la cálculo de la solución, pero por otro lado deja la ecuación de conservación

de la masa sin ningún término que se corresponda con la presión de manera directa o indirecta. Puesto que generalmente esta ecuación se utiliza para resolver la presión, esto puede llevarnos, tarde o temprano, a la aparición de un 0 en la diagonal del sistema de ecuaciones que debe ser resuelto de alguna manera; este problema se trata con mayor nivel de detalle en 4.1.

2.2.3. Laminar / Turbulento

Se dice que un fluido es turbulento cuando está caracterizado por recirculaciones o "remolinos" que se mueven de manera caótica o aparentemente aleatoria. Se introduce en este contexto el concepto de *Eddy*, con el que nos referimos a estos torbellinos turbulentos que se crean en un fluido, por ejemplo, tras pasar junto a un obstáculo. Debido a la complejidad del fenómeno en sí y de que el proyecto da gran importancia a los flujos turbulentos, en 2.3 se comenta más detalladamente esta situación.

Por otro lado, un fluido en el que no se den estas circunstancias, donde todas las partículas sigan una trayectoria relativamente uniforme, provocada exclusivamente por las fuerzas externas causantes del movimiento, se denomina laminar.

Para la caracterización de un flujo turbulento o laminar es de gran utilidad el número de Reynolds. Este número es una magnitud adimensional que relaciona las fuerzas inerciales con las fuerzas debidas a la viscosidad de la siguiente manera:

$$Re = \frac{\rho V L}{\mu}$$

donde ρ es la densidad, V la velocidad media del fluido, L la longitud característica del sistema (por ejemplo, en el caso del flujo en una tubería, el diámetro de dicha tubería) y μ la viscosidad dinámica del fluido.

De esta manera, conociendo el número de Reynolds de un fluido, es posible aproximar el nivel de turbulencia del mismo. Sirva como ejemplo el caso del flujo a través de una tubería, donde el flujo sería laminar cuando $\mathrm{Re} < 2300$, turbulento cuando $\mathrm{Re} > 4000$ y un estado de transición, donde ambas situaciones son posibles y dependen de otros factores, en el intervalo intermedio.

Por último, cabe destacar que, a pesar de que el fenómeno de la turbulencia no simplifica ni modifica de manera directa las ecuaciones del fluido, sí que es de capital importancia su consideración de cara a adoptar una de las distintas estrategias disponibles en los flujos turbulentos, tal y como se comenta más adelante.

2.2.4. Estacionario / No estacionario

Los fluidos también pueden ser clasificados en función de su comportamiento a lo largo del tiempo. De esta manera, un flujo cuyas propiedades no

varíen a lo largo del tiempo es estacionario, mientra que si estas propiedades cambian en el tiempo, se dice que el flujo está en régimen transitorio o que es no estacionario.

Los fluidos turbulentos son no estacionarios por definición, ya que el propio fenómeno turbulento propicia un movimiento continuo a través de los *Eddies* o torbellinos. Sin embargo, Pope afirma que un determinado campo aleatorio *U(x,t)* es estadísticamente estacionario si todas las estadísticas permanecen invariables bajo un determinado paso de tiempo; esto es, todas las propiedades permanecen constantes en el tiempo [6]. De esta manera, tomando la media de los campos de interés en un flujo turbulento con un determinado paso de tiempo puede llevarnos a la invariabilidad de los mismos y, por tanto, a que el flujo pueda alcanzar un régimen estadísticamente estacionario.

Por último, la principal ventaja que plantean un flujo estacionario con respecto de un flujo no estacionarios es la eliminación de todas las derivadas temporales de las ecuaciones de conservación. Así pues, al margen de la ventaja debida a la estabilidad del problema para un flujo estacionario, el sistema resultante constará de una dimensión menos: el tiempo.

2.2.5. Newtoniano / No newtoniano

Un fluido es newtoniano cuando la curva que relaciona el estrés o fuerza aplicada al fluido y su deformación (que se pone de manifiesto mediante un cambio en la velocidad del fluido) es una recta que pasa por el origen; es decir, están relacionados linealmente mediante una constante de proporcionalidad. Dicha constante es la viscosidad del fluido y se define de la siguiente manera:

$$\tau = \mu \frac{du}{dy}$$

donde τ es el esfuerzo cortante (o fuerza aplicada sobre el fluido de manera tangencial), μ es la constante de proporcionalidad o *viscosidad* y $\frac{du}{dy}$ es el gradiente de velocidad, perpendicular a la dirección del esfuerzo cortante.

Así pues, un fluido cuya viscosidad sea constante será newtoniano (agua, aire, vino, ...), mientras que en caso contrario será no newtoniano (miel, pegamento, ...). Esto tiene una consecuencia directa en la parte derecha de las ecuaciones de conservación del momento, puesto que parte de las fuerzas que actúan sobre el fluido se pueden escribir en función del esfuerzo cortante, que a su vez depende de una viscosidad constante. Como explican Ferziger y Perić [7], muy diferente es el caso de fluidos no newtonianos, donde la relación entre el esfuerzo cortante y la velocidad se define mediante un conjunto de ecuaciones en derivadas parciales que aumentan en gran medida la complejidad del problema. Sin embargo, la mayor parte de los fluidos reales, y más concretamente los tratados en este proyecto, se comportan como fluidos newtonianos.

2.3. Flujos turbulentos

Como se ha explicado en el apartado anterior, los flujos turbulentos se caracterizan principalmente por la aparición de recirculaciones de manera caótica o aparentemente aleatoria. Así, es necesario poder entender y predecir este comportamiento con la mayor exactitud posible con el fin de obtener buenos diseños en la ingeniería. Por otro lado, con el aumento en los requisitos de las aplicaciones industriales, se requieren cada vez mayores niveles de detalle y precisión, por lo que los métodos numéricos se han vuelto imprescindibles en el estudio de flujos turbulentos. En las siguientes subsecciones definiremos con algo más de detalle en qué consisten las principales aproximaciones disponibles para la predicción de flujos turbulentos. Cabe destacar que, al margen de las que aquí se exponen, existen más aproximaciones [8] pero que carecen de importancia para este trabajo y a la vez, exceden el alcance del mismo.

2.3.1. DNS

El primer método es la simulación numérica directa (*DNS* por sus siglas en inglés, *Direct Numerical Simulation*). Se trata de la técnica de simulación de flujos turbulentos más precisa, ya que no se promedia ni se aproxima ningún resultado, sino que se resuelven las ecuaciones de Navier-Stokes para todos los *Eddies*, independientemente de su tamaño o escala. Constituye la aproximación más simple desde el punto de vista conceptual y en cierto modo se puede considerar equivalente a la realización de un experimento en laboratorio con un flujo y dominio de las mismas características.

Sin embargo, para capturar todas las estructuras del fenómeno turbulento, incluyendo las de menor escala, es necesario disponer de una malla con una densidad suficiente. Generalmente, esto llega a ser prohibitivo para la posterior ejecución de la simulación. No obstante, existen simulaciones bien conocidas, realizadas con la técnica DNS, que suelen utilizarse para la validación de resultados obtenidos con otra técnica (u otro código) para el mismo problema.

2.3.2. LES

El método de simulación de grandes escalas (*LES*, por sus siglas en inglés *Large Eddy Simulation*) consiste en resolver o simular los *Eddies* de gran escala, mientras que los menores se modelan, pudiendo elegir entre diferentes modelos para este propósito. La idea básica tras esta técnica es que los *Eddies* de mayor escala son mucho más energéticos que los más pequeños y, por tanto, contribuyen en mayor medida al transporte de las propiedades del flujo, tal y como explica Kolmogorov en su teoría de 1941. De esta manera, sin entrar en detalles formales, el proceso consiste en dividir los campos

del fluido (principalmente velocidades y presión) en la parte resuelta para las grandes escalas y la parte que debe ser modelada. Así, se resolverán las ecuaciones de Navier-Stokes para las grandes escalas, pero se añadirá un término a estas ecuaciones que incluya la parte modelada. Es importante destacar que el filtro que determina qué se debe resolver y qué se debe modelar es, generalmente, la propia malla, por lo que los modelos se conocen como modelos SGS por sus siglas en inglés (*SubGrid Scale*).

De esta manera, el término adicional que se introduce en las ecuaciones, y que representa la diferencia entre el campo total y el campo considerando únicamente las fenómenos de mayor escala, es

$$\frac{1}{\rho}\frac{\partial \tau_{ij}}{\partial x_j}$$

donde normalmente se utiliza la siguiente igualdad para el término τ_{ij}:

$$\tau_{ij} - \frac{1}{3}\tau_{kk}\delta_{ij} = 2\mu_t \overline{S}_{ij}$$

siendo $\overline{S}_{ij}$ la velocidad de deformación

$$\overline{S}_{ij} = \frac{1}{2}\left(\frac{\partial \overline{u}_i}{\partial x_j} + \frac{\partial \overline{u}_j}{\partial x_i}\right)$$

y μ_t la viscosidad turbulenta a simular. En los siguientes apartados trataremos el modelo de Smagorinsky para esta viscosidad turbulenta, y los modelos de van Driest y (de manera muy superficial) dinámico, que se basan en este primero.

Modelo de Smagorinsky

El modelo de Smagorinsky define la viscosidad turbulenta como

$$\mu_t = C_S^2 \rho \Delta^2 |\overline{S}|$$

donde Δ es la raíz cúbica del volumen finito, $\overline{S}$ es $\sqrt{2S_{ij}S_{ij}}$ y C_S es la constante de Smagorinsky, propia del modelo y que suele tomar valores en el rango $[0.065, 1.1]$ dependiendo del tipo de flujo turbulento.

Este modelo ha demostrado ser satisfactorio en la práctica. Sin embargo, plantea el principal problema de que la constante de Smagorinsky mantiene el mismo valor en todo el dominio, por lo que la viscosidad turbulenta no decrece conforme se considera el flujo más cercano a las superficies, donde el flujo tiende a ser laminar. Con el fin de atajar este inconveniente se crearon los siguientes modelos.

Modelo de van Driest

Como se ha comentado en el apartado anterior, la simple aplicación del modelo de Smagorinsky cerca de las superficies nos lleva a una simulación que se aleja de los resultados correctos a medida que la distancia a la superficie disminuye. Para contrarrestar esto se define la amortiguación de van Driest [9], que consiste principalmente en multiplicar la constante de Smagorinsky (C_S) por un factor de escala que disminuye dicha constante en función de la distancia a las superficies de la siguiente manera:

$$C_S = C_S(y) = C_S \cdot \left(1 - e^{(-y^*/A)}\right). \tag{2.4}$$

En esta expresión, A es la constante de van Driest y suele tomar el valor 25, mientras que y^* (también conocido como número de Reynolds turbulento), se calcula mediante

$$y^* = \frac{d}{\nu}\sqrt{\frac{\tau_w}{\rho}}, \tag{2.5}$$

donde d es la distancia mínima a la pared, ν es la viscosidad cinemática del fluido y τ_w el esfuerzo cortante. Además, la viscosidad cinemática cumple que

$$\nu = \frac{\mu}{\rho}, \tag{2.6}$$

siendo μ la viscosidad cinemática (parámetro de entrada). Por último, sabiendo que es posible calcular el esfuerzo cortante mediante

$$\tau_w = \mu\frac{v_{tan}}{d}, \tag{2.7}$$

donde v_{tan} es la componente de la velocidad tangencial a la pared, es posible reescribir la ecuación 2.5 como

$$y^* = \frac{d}{\nu}\sqrt{\frac{\tau_w}{\rho}} = \frac{d}{\nu}\sqrt{\frac{\mu\, v_{tan}}{d\,\rho}} = \frac{d}{\nu}\sqrt{\frac{\nu\, v_{tan}}{d}} = \sqrt{\frac{d\, v_{tan}}{\nu}} = \sqrt{\frac{d\,\rho\, v_{tan}}{\mu}} \tag{2.8}$$

y sustituyendo en la ecuación inicial (2.4), se obtiene

$$C_S = C_S(d) = C_S\left(1 - e^{-\sqrt{\frac{d\,\rho\, v_{tan}}{\mu A^2}}}\right). \tag{2.9}$$

Modelo dinámico

Por último, el modelo dinámico se dirige al mismo fin que el modelo de van Driest y plantea una solución similar. En este caso, la constante de Smagorinsky pasa de ser constante en todo el dominio a ser un valor dependiente del espacio y el tiempo. Así, en cada instante de tiempo y cada

volumen finito, el valor podrá ser distinto. A pesar de que se considera un desarrollo necesario a corto plazo para el proyecto de investigación, para el desarrollo de este trabajo no se ha necesitado este modelo, por lo que no se detallará formalmente de qué manera es posible calcular la constante de Smagorinsky en cada punto e instante de tiempo.

2.3.3. RANS

Este método se basa en un conjunto de ecuaciones obtenidas promediando las ecuaciones de conservación en el tiempo, que se conocen como ecuaciones *Reynolds-Averaged Navier-Stokes (RANS)*. De esta manera, toda la componente no turbulenta es promediada; es decir, es considerada como parte de la turbulencia, lo cual introduce en las ecuaciones promediadas ciertos términos que deben ser modelados. Además es importante saber que, debido a la complejidad de los flujos turbulentos, es complicado que exista un modelo RANS capaz de representar todos los flujos, por lo que estos modelos y resultados deben considerarse como aproximaciones con aplicación en la ingeniería y no como leyes científicas. Por último cabe destacar que, al margen de que está fuera del objetivo de este documento, esta técnica no ha sido utilizada a lo largo de este trabajo, por lo que no entraremos a detallar los diferentes modelos.

2.4. Sistema de ecuaciones lineales

Independientemente de la estrategia seguida para afrontar los flujos turbulentos (DNS, LES, RANS), la linealización del sistema de ecuaciones (Newton 3.5, Picard 4.3.1, ...) o la discretización (4.1), finalmente hemos de afrontar la resolución de un gran sistema de ecuaciones lineales con una matriz de coeficientes de gran dispersión de la forma $Ax = b$.

Para resolver este tipo de problemas existen dos aproximaciones fundamentales: los métodos directos y los métodos iterativos. Entre los métodos directos cabe destacar la factorización LU o eliminación gaussiana, que es uno de los métodos más intuitivos y simples para resolver sistemas de ecuaciones lineales (ver [10] para más detalles). Sin embargo, este método produce llenado; es decir, a medida que el método avanza aumenta el número de elementos no nulos de la matriz, por lo que en gran parte se pierde la ventaja de la dispersión de la matriz. Es por esto que el coste de este algoritmo resulta prohibitivo para la dimensión de nuestro problema y, en general, para problemas resultantes de discretizar ecuaciones de derivadas parciales.

Por otro lado, se encuentran los métodos iterativos que se basan en dar una primera aproximación arbitraria de la solución e ir aproximándola de manera iterativa a la solución exacta hasta que se cumple un cierto criterio de convergencia. Tal vez, la implementación más simple de esta idea sea

reescribir el sistema $Ax = b$ como una iteración del punto fijo. Esta iteración se define sobre una función $f : \mathbb{R} \to \mathbb{R}$ de la siguiente manera:

$$x_{n+1} = f(x_n), n = 0, 1, 2, ...$$

Esta función genera una sucesión x_0, x_1, x_2, ... que se espera que converja a un determinado valor x. Si esta sucesión es convergente es posible demostrar que esa x es un punto fijo de la función; es decir, $x = f(x)$.

De esta manera, podemos reescribir el sistema $Ax = b$ como

$$x = (I - A)x + b.$$

y definir la *iteración de Richardson* de la siguiente manera:

$$x_{k+1} = (I - A)x_k + b.$$

En general, nos referimos a cualquier método donde $x_{k+1} = Mx_k + c$ como *método estacionario*, ya que el paso de x_k a x_{k+1} no depende de nada más que de el resultado anterior y de la matriz M o *matriz de iteración*, que permanece invariable durante toda la resolución.

Sin embargo, en este trabajo no se hace uso de los métodos iterativos estacionarios, sino de aquellos basados en los *subespacios de Krylov*. Estos métodos no se basan en la existencia de una matriz de iteración, sino en el subespacio generado por una matriz A de dimensión $n \times n$ y un vector b de tamaño n. Este subespacio se genera mediante las primeras k ($k < n$) potencias de A aplicadas sobre r_0, esto es:

$$\mathcal{K}_k(A, r_0) = \text{span}\{r_0, Ar_0, A^2r_0, \ldots, A^{k-1}r_0\}$$

Partiendo de una solución inicial arbitraria (generalmente, $x_0 = 0$), y haciendo uso de los subespacios de Krylov, los métodos que se exponen a continuación, para una determinada iteración k, tratan de minimizar alguna medida del error de la aproximación calculada sobre el subespacio $x_0 + \mathcal{K}_k$.

2.4.1. Gradiente Conjugado

El método del gradiente conjugado (en adelante CG por sus siglas en inglés, *Conjugate Gradient*), se basa en la minimización de la siguiente función cuadrática:

$$\phi(x) = \frac{1}{2}x^T Ax - x^T b.$$

Es posible demostrar que esta función alcanza un mínimo cuando $Ax = b$. De esta manera, en cada iteración, el método realiza una búsqueda unidimensional a lo largo de una determinada dirección s_k de manera que $x_{k+1} = x_k + \alpha s_k$, donde α es el parámetro que se debe determinar de forma

que la función $\phi(x_k + \alpha s_k)$ se minimice. Este mínimo se produce cuando el nuevo residuo es ortogonal a la dirección de búsqueda, es decir: $r_{k+1}^T s_k = 0$. Llegados a este punto, es posible expresar el nuevo residuo en función del parámetro α de la siguiente manera:

$$r_{k+1} = b - Ax_{k+1} = b - A(x_k + \alpha s_k) = b - Ax_k - \alpha As_k = r_k - \alpha As_k$$

y, por tanto, expresar α como:

$$\alpha = \frac{r_k^T s_k}{s_k^T (As_k)}.$$

Habiendo determinado la función a minimizar, la actualización del vector solución y el cálculo del parámetro a lo largo de la dirección de búsqueda, sólo queda determinar cómo obtener la nueva dirección de búsqueda. Desde el punto de vista conceptual, esta nueva dirección debería ser ortogonal a todas las direcciones anteriores, haciendo uso de la ortogonalización de Gram-Schmidt o similar. Sin embargo, esta ortogonalización requeriría mayor almacenamiento para todas las direcciones utilizadas, así como un coste computacional prohibitivo. Es por esto que en la práctica se opta por que la nueva dirección sea A-ortogonal (o conjugada, de ahí el nombre del método) con respecto a la dirección anterior. De esta manera, el coste computacional y de almacenamiento es mínimo y el algoritmo resultante es:

Algoritmo 1 Gradiente Conjugado

x_0 = solución inicial
$r_0 = b - Ax_0$
$s_0 = r_0$
for k = 0,1,2,... **do**
  $\alpha_k = \dfrac{r_k^T r_k}{s_k^T A s_k}$
  $x_{k+1} = x_k + \alpha_k s_k$
  $r_{k+1} = r_k - \alpha_k A s_k$
  $\beta_{k+1} = \dfrac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
  $s_{k+1} = r_{k+1} + \beta_{k+1} s_k$
end for

2.4.2. Gradiente Biconjugado

El principal problema que plantea el método CG es el hecho de que es aplicable únicamente a matrices simétricas y definidas positivas, haciéndose imposible la ortogonalización de los vectores residuo de manera eficiente en cualquier otro caso (ver [11] para más detalles). Para evitar este problema

se desarrolló el algoritmo del gradiente biconjugado (en adelante BiCG por su acrónimo en inglés *BiConjugate Gradient*).

Este método se basa en la idea de mantener dos subespacios de Krylov cuyos residuos en una determinada iteración k son ortogonales entre sí. Concretamente se mantiene el mismo subespacio utilizado en el método CG

$$\mathcal{K}_k(A, r_0) = \text{span}\{r_0, Ar_0, A^2 r_0, \ldots, A^{k-1} r_0\}$$

y además se genera otro de la forma

$$\overline{\mathcal{K}_k}(A^T, \hat{r}_0) = \text{span}\{\hat{r}_0, A^T \hat{r}_0, (A^T)^2 \hat{r}_0, \ldots, (A^T)^{k-1} \hat{r}_0\}$$

donde $\hat{r}_0$ es un vector aleatorio (generalmente $\hat{r}_0 = r_0$) y $r_k^T w = 0$ para todo $w \in \overline{\mathcal{K}_k}$. Es decir, en una determinada iteración k, cualquier vector del subespacio $\overline{\mathcal{K}_k}$ (y, en concreto, el residuo $\hat{r}_0$) es ortogonal al residuo r_k. Cabe destacar que, puesto que los residuos son ortogonales, las direcciones de búsqueda generadas serán *biconjugadas*; es decir, $\hat{s}_k^T A s_l = 0$ si $k \neq l$. El algoritmo es el siguiente:

Algoritmo 2 Gradiente Biconjugado

x_0 = solución inicial
$r_0 = b - Ax_0$
$s_0 = r_0$
$\hat{r}_0 = r_0$
$\hat{s}_0 = \hat{r}_0$
for k = 0,1,2,... **do**
  $\alpha_k = \dfrac{\hat{r}_k^T r_k}{\hat{s}_k^T A s_k}$
  $x_{k+1} = x_k + \alpha_k s_k$
  $r_{k+1} = r_k - \alpha_k A s_k$
  $\hat{r}_{k+1} = \hat{r}_k - \alpha_k A^T \hat{s}_k$
  $\beta_{k+1} = \dfrac{\hat{r}_{k+1}^T r_{k+1}}{\hat{r}_k^T r_k}$
  $s_{k+1} = r_{k+1} + \beta_{k+1} s_k$
  $\hat{s}_{k+1} = \hat{r}_{k+1} + \beta_{k+1} \hat{s}_k$
end for

Es importante destacar que el uso de BiCG no está generalizado debido principalmente al carácter errático del algoritmo en la convergencia. Es por ello que se han desarrollado diferentes optimizaciones del algoritmo, las cuales sí que se utilizan de manera generalizada. Concretamente en este trabajo se utiliza el método *BiCGStab(ℓ)*, aunque su desarrollo e implementación exceden el ámbito de este trabajo y se pueden encontrar en [12], junto con los principales algoritmos de resolución de sistemas lineales iterativos, tanto estacionarios como no estacionarios.

2.4.3. Residuo mínimo generalizado

Por último, el método del residuo mínimo generalizado (GMRES por su acrónimo en inglés, *Generalized Minimum Residue*) se basa en la idea de resolver un problema de mínimos cuadrados en cada iteración del problema de manera que se minimiza el residuo

$$\|r_k\|_2 = \|b - Ax_k\|_2,$$

siendo x_k la aproximación de la solución exacta x^* en la iteración k. Para conseguir esto, se utiliza el subespacio de Krylov generado por la matriz A y el vector b:

$$\mathcal{K}_k(A, b) = \text{span}\{b, Ab, A^2b, \ldots, A^{k-1}b\},$$

y, por tanto, la matriz de Krylov es

$$K_k = [b, Ab, A^2b, \ldots, A^{k-1}b].$$

De esta manera, es posible escribir la aproximación del vector solución como $x_k = K_k c$, donde $c \in \mathbb{C}$. Así pues, el residuo a minimizar se convierte en

$$\|r_k\|_2 = \|b - Ax_k\|_2 = \|AK_k c - b\|_2.$$

Por otro lado, cabe destacar que las columnas de la matriz de Krylov K_k tienden a ser linealmente dependientes conforme aumenta el número de columnas. Es por esto que se utiliza la iteración de Arnoldi con el fin de obtener una base ortonormal Q_k para el subespacio $\mathcal{K}_k$ de forma que el vector solución aproximado se reescribe como $x_k = Q_k y$ para un cierto $y \in \mathbb{C}$, resultando el residuo a minimizar como

$$\|b - Ax_k\|_2 = \|AQ_k y - b\|_2.$$

Teniendo en cuenta la idea principal de Arnoldi $AQ_k = Q_{k+1}\hat{H}_k$, donde $\hat{H}_k$ es una matriz Hessenberg superior, es posible transformar el residuo a minimizar en:

$$\|Q_{k+1}\hat{H}_k y - b\|_2 = \|\hat{H}_k y - Q_{n+1}^T b\|_2.$$

Es posible demostrar ([13]) que $Q_{n+1}^T b = \|b\|_2 e_1$, por lo que nuestro problema resultante consiste en minimizar la siguiente norma:

$$\|\hat{H}_k y - \|b\|_2 e_1\|_2,$$

con $x_k = Q_k y$.

Así pues, el método GMRES consiste en obtener las matrices Q_k y $\hat{H}_k$ mediante la iteración de Arnoldi y, posteriormente, resolver un problema de mínimos cuadrados (por ejemplo, mediante la factorización QR). El algoritmo que implementa este método es el siguiente:

Algoritmo 3 Residuo mínimo generalizado

$q_0 = b/\|b\|$
for k = 0,1,2,... **do**
  $v = Aq_k$
  for i = 0,...,k **do**
    $\hat{h}_{ik} = q_i^T v$
    $v = v - \hat{h}_{ik} q_i$
  end for
  $\hat{h}_{k+1,k} = \|v\|_2$
  $q_{k+1} = \frac{v}{\hat{h}_{k+1,k}}$
  Encontrar y que minimice $\|\hat{H}_k y - \|b\|_2 e_1\|_2$ (factorización QR)
  $x_k = Q_k y$
end for

2.5. Precondicionadores

Se conoce como precondicionamiento a la técnica que consiste en transformar un determinado sistema de ecuaciones en uno equivalente con unas propiedades numéricas mas adecuadas para su resolución. Generalmente, en los sistemas de ecuaciones lineales, el precondicionamiento se centra en reducir el número de condicion de la matriz de coeficientes, que como se puede ver en [10] tiene relación con la estimación del error cometido en la aproximación de la solución por métodos iterativos.

Para conseguir esto, se premultiplica la matriz de coeficientes por la inversa de otra matriz P, conocida como *precondicionador*. Así, si aplicamos el precondicionador P a un sistema $Ax = b$, éste se convierte en $P^{-1}Ax = P^{-1}b$. Cabe destacar que también es posible postmultiplicar la matriz de coeficientes por el precondicionador, conociéndose la primera opción como precondicionado por la izquierda y esta última como precondicionado por la derecha.

De esta manera, la elección de la matriz P es de gran importancia, ya que la mejora en la precisión y el tiempo de cálculo de la solución debe compensar el coste de aplicar el precondicionador. En los siguientes apartados se explican brevemente los precondicionadores más simples que han sido utilizados en este trabajo.

2.5.1. Jacobi

El precondicionador de Jacobi es el más simple que existe, ya que se forma a partir de la diagonal de la matriz de coeficientes; es decir, $P = diag(A)$. Esto es, la matriz P se forma de la siguiente manera:

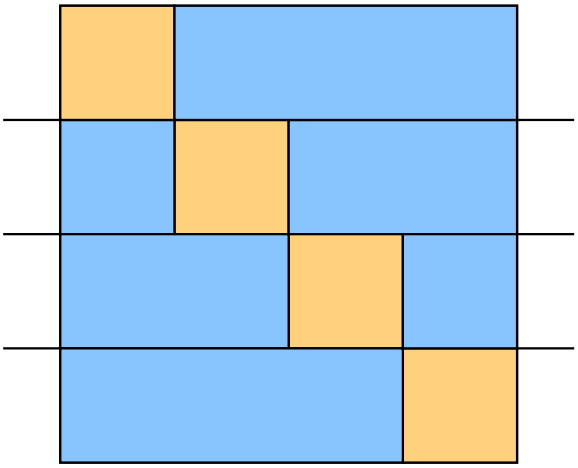

Figura 2.1: Bloques diagonales de una matriz de coeficientes para el precondicionador Block Jacobi.

$$P_{ij} = \begin{cases} A_{ii} & \text{si } i = j \\ 0 & \text{en caso contrario} \end{cases}$$

2.5.2. ILU

Este precondicionador se basa en la factorización LU de la matriz A de manera incompleta (ILU por sus siglas en inglés, *Incomplete LU*). Como se ha comentado anteriormente, el principal inconveniente de la factorización LU es que produce llenado; es decir, se pierde la dispersión de la matriz, convirtiéndose el problema en inabordable.

De esta manera, la factorización LU incompleta consiste en definir el precondicionador P como la factorización LU de A, pero manteniendo como nulos todos o parte de los elementos que lo son en A. Así, si mantenemos exactamente los mismos elementos no nulos que en A; es decir, si utilizamos el mismo patrón de dispersión que A y evitamos completamente el llenado, estaremos haciendo uso de ILU(0). Cabe destacar que también existen otras variantes que producen un llenado parcial del precondicionador.

Así, se obtiene un precondicionador tal que al obtener la matriz inversa P^{-1}, ésta será una aproximación de A^{-1} y, por tanto, la multiplicación $P^{-1}A$ será una aproximación de la matriz identidad I. Es por esto que la eficacia de este precondicionador vendrá dada por la diferencia entre el precondicionador y la inversa real de la matriz de coeficientes.

2.5.3. Block Jacobi

Por último, el precondicionador de Jacobi por bloques, o *Block Jacobi* por su nombre en inglés, es una generalización del precondicionador de

Jacobi simple. Concretamente, consiste en obtener un conjunto de bloques diagonales en lugar de un único elemento. Así, si dividimos los índices de las filas de la matriz de coeficientes en varios grupos disjuntos y con mismo número de filas (en la medida de lo posible), tal y como se muestra en la Figura 2.1, la matriz P se formará mediante

$$P_{ij} = \begin{cases} A_{ij} & \text{si } i \text{ y } j \text{ están en el mismo grupo} \\ 0 & \text{en caso contrario} \end{cases}$$

De esta manera, para aplicar el precondicionador P^{-1} se han de invertir cada uno de los bloques. Puesto que calcular la inversa es demasiado costoso, generalmente se recurre a una inversa aproximada, como puede ser la que produce la factorización ILU.

Por otro lado, en función del tamaño del bloque elegido, Block Jacobi es equivalente a Jacobi si existen n bloques de un solo elemento (siendo n el número de filas de la matriz A), o a ILU si existe un solo bloque de tamaño $n \times n$.

Cabe destacar que el tamaño del bloque elegido suele basarse en la agrupación de ecuaciones aplicadas para un mismo punto del dominio o en el número de procesadores utilizados para la resolución del sistema.

Capítulo 3

PETSc

PETSc (*Portable, Extensible Toolkit for Scientific Computation*) [5] es un conjunto de rutinas y estructuras de datos que proporcionan las herramientas necesarias para la implementación de aplicaciones científicas escalables enfocadas a la resolución numérica de ecuaciones de derivadas parciales en máquinas paralelas (y/o secuenciales) de alto rendimiento. Para ello, hace uso del estándar de paso de mensajes MPI [2].

PETSc proporciona gran flexibilidad a los usuarios debido al uso de técnicas de la programación orientada a objetos integrables en aplicaciones escritas en C, Fortran o C++. Está organizada de manera jerárquica en diferentes módulos de manera que cada una de estas librerías proporciona una interfaz abstracta y una o varias implementaciones de las estructuras de datos particulares, incluyendo *solvers* (lineales y no lineales), integradores del paso de tiempo (*time steppers*), así como estructuras de datos y rutinas de más bajo nivel para el manejo de matrices y vectores de manera que el usuario puede emplear el nivel de abstracción más apropiado para su aplicación particular, tal y como se muestra en la Figura 3.1.

Por otra parte, debido a la complejidad de PETSc y al hecho de que incorpora utilidades adicionales, tales como monitorización de aplicaciones, para muchos usuarios presenta una curva de aprendizaje mucho más abrupta que una librería de subrutinas simple. Sin embargo, la cantidad de tiempo empleada en el aprendizaje se convierte posteriormente en un aumento de la eficiencia y la productividad debido a las múltiples ventajas que proporciona PETSc.

Entre estas ventajas encontramos, además del manejo de matrices y vectores y la resolución de sistemas de ecuaciones lineales y no lineales, la flexibilidad de modificar el comportamiento de las aplicaciones en tiempo de ejecución. De esta manera, propiciado por la flexibilidad de la programación orientada a objetos de la librería, es posible especificar multitud de parámetros por línea de comandos en la ejecución que toman prioridad con respecto a las establecidas en el código. En las siguientes secciones pasaremos

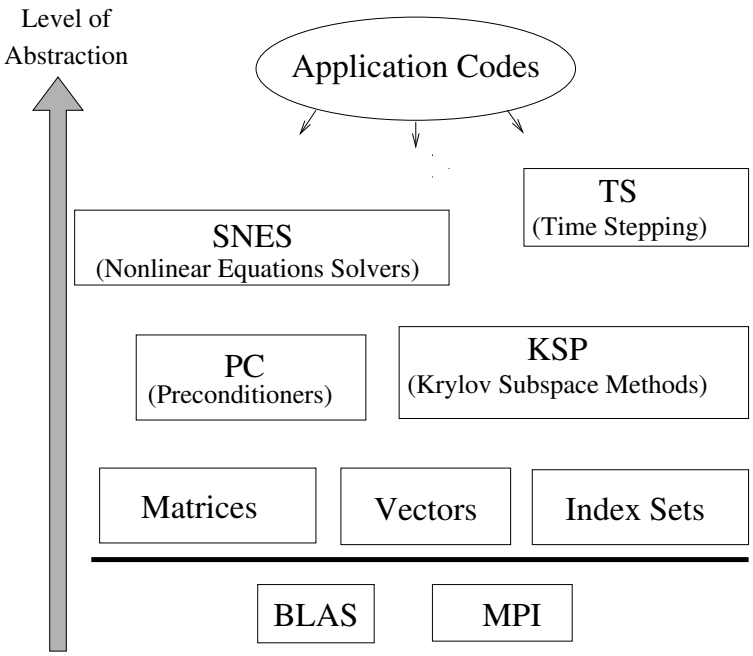

Figura 3.1: Nivel de abstracción de la librería PETSc.

a detallar las librerías más relevantes para el desarrollo de este trabajo.

3.1. Vectores

Los vectores son los elementos más básicos de PETSc y existen dos implementaciones distintas para la misma interfaz abstracta: secuenciales y paralelos (basados en MPI). Se utilizan principalmente para almacenar soluciones discretas de ecuaciones de derivadas parciales, vectores de términos independientes en sistemas de ecuaciones lineales, etc.

En cuanto a las operaciones que PETSc permite realizar, el nombre de las funciones comienza con el nombre del objeto al que referencian. De esta manera, para crear un vector se deberá usar las rutinas `VecCreateMPI` o `VecCreateSeq` en función de si el vector que se quiere crear se ha de manejar mediante MPI o de manera secuencial, o bien con el constructor genérico `VecCreate`, cuyo primer argumento será un comunicador que determinará el tipo de vector. Así, en la siguiente instrucción

```
VecCreate(PETSC_COMM_WORLD, &x);
```

se crea un vector paralelo, puesto que el comunicador `PETSC_COMM_WORLD` equivale al comunicador `MPI_COMM_WORLD`. El caso contrario sería el vector secuencial para el cual se utilizaría el comunicador `PETSC_COMM_SELF`. De

cualquier manera, se crea un objeto de tipo `Vec` que es devuelto en el parámetro `x`. En este momento es posible (y recomendable) hacer una llamada a las funciones `VecSetSizes` y `VecSetFromOptions`, que se encarga de establecer el tamaño y las demás características que se han podido establecer mediante la línea de comandos, respectivamente:

```
VecSetSizes(x, PETSC_DECIDE, 20);
VecSetFromOptions(x);
```

Aquí se puede observar como la llamada a `VecSetSizes` espera como primer parámetro el objeto de tipo vector, y como segundo y tercer parámetro los tamaños local y global del vector, respectivamente. Cabe destacar el uso de `PETSC_DECIDE`, que nos permite especificar únicamente el tamaño global del vector y delegar en PETSc la división del vector; o viceversa.

Posteriormente se pueden realizar multitud de operaciones, que tendrán también como primer parámetro el vector creado, tales como establecer todos los valores del vector a un determinado escalar (`VecSet`), calcular la norma (`VecNorm`), el sumatorio de todos sus componentes (`VecSum`), el producto escalar (`VecDot`) u operaciones más complejas junto con otros objetos (matrices o vectores):

```
VecGetSize(x, &N);
MPI_Comm_rank(PETSC_COMM_WORLD, &rank);

if (rank == 0) {
  for(i = 0, val = 0.0; i < N; i++, val += 10.0) {
    VecSetValues(x, 1, &i, &val, INSERT_VALUES);
  }
}

VecAssemblyBegin(x);
VecAssemblyEnd(x);

VecDot(x, x, &norm);
...
VecDestroy(x);
```

En el anterior ejemplo se obtiene el tamaño global del vector en `N` mediante la función `VecGetSize`, así como el rango del proceso en `rank` mediante `MPI_Comm_rank`. Cabe destacar que este es uno de los pocos casos en los que se suele llamar directamente a MPI, ya que PETSc trata de proporcionar unas funciones con las que el desarrollador debe saber cuál es el funcionamiento paralelo que se está produciendo, pero no suele verse obligado a llamar a rutinas de MPI de manera directa, salvo en casos muy particulares. Así, sabiendo el tamaño del vector y el rango, el proceso 0 (y

sólo el 0) establece todos los valores del vector siguiendo una función lineal $x(i) = 10 * i$. Finalmente, se hace el ensamblado del vector, que consiste en realizar las comunicaciones necesarias para que los valores establecidos por el procesador 0 se distribuyan de manera correcta por todos los procesos. Es importante notar que esta etapa se hace en dos fases, `VecAssemblyBegin` y `VecAssemblyEnd`. La utilidad de esto es principalmente agrupar comunicaciones del ensamblado de varios vectores de forma que se utilizara un tiempo de latencia único para todas las comunicaciones, en lugar de uno por cada vector. Esto se consigue agrupando primero todas las llamadas a `VecAssemblyBegin` seguidas de todas las llamadas a `VecAssemblyEnd`. Finalmente, y tras realizar todos los cálculos que sean necesarios, se libera la memoria ocupada por el vector haciendo uso de la función `VecDestroy`.

3.2. Matrices

PETSc proporciona distintas implementaciones para las matrices ya que no existe una única implementación adecuada para todos los tipos de problemas. De esta manera, la librería proporciona formatos de almacenamiento denso o disperso en sus versiones paralela y secuencial, así como varios formatos especializados. Es importante destacar que desde PETSc se pone mucho énfasis en el hecho de que los objetos se definen como un conjunto de operaciones que se pueden realizar con ellos, y no como una implementación concreta. Así, una matriz no es un conjunto de $n \times m$ elementos, sino un objeto que permite el cálculo de la multiplicación matriz por matriz, la obtención de la transpuesta, etc. Esto se debe al hecho de que la implementación puede (y debe) cambiar en función del problema y de características tales como el patrón de dispersión de la matriz, pero el resultado de las operaciones sobre ella debe ser siempre el mismo.

También es de especial interés la forma en que PETSc reserva memoria para las matrices dispersas en formato de fila comprimida (CSR por sus siglas en inglés, *Compressed Sparse Row*), conocida en PETSc como formato disperso AIJ. El funcionamiento en PETSc para este tipo de matrices (AIJ y ciertos formatos derivados de éste), muy comunes en el ámbito de los algoritmos numéricos, es que reserva memoria para un número por defecto de elementos por cada fila de manera que, a medida que se construye la matriz, cuando se alcanza el límite se reserva más memoria, se hace una copia del contenido actual de la fila y se libera la anterior memoria; este proceso se repite mientras sea necesario hasta la construcción completa de la matriz. Esto presenta el problema de que este manejo de memoria y las consiguientes copias de elementos suponen un gasto computacional nada despreciable que puede ser eliminado si se sabe de antemano el número (máximo) de elementos que contendrá la fila más densa (o cada una de las filas, si se tiene dicha información). Este proceso se conoce como *preasignación*.

Así, de manera análoga a la creación de vectores, se puede hacer uso de las rutinas `MatCreate`, `MatSetSizes` y `MatSetFromOptions` para crear las matrices en función de los valores especificados en la línea de comandos. Sin embargo, cabe destacar que PETSc proporciona multitud de constructores como `MatCreateMPIAIJ`, `MatCreateMPIDense` o `MatCreateSeqAIJ` con el fin de crear la matriz especificando la implementación concreta que se desea utilizar.

```
MatCreate(PETSC_COMM_WORLD, &m);
MatSetSizes(m, PETSC_DECIDE, PETSC_DECIDE, 20, 20);
MatSetFromOptions(m);
```

En este punto, es cuando se puede hacer uso de la preasignación mediante la función `MatMPIAIJSetPreallocation`.

```
MatMPIAIJSetPreallocation(m, 4, PETSC_NULL, 0, PETSC_NULL);
```

Cabe destacar que esta función divide sus argumentos de dos maneras distintas. Por un lado, se entiende la porción de la matriz correspondiente a un determinado proceso como la yuxtaposición de una submatriz de tamaño $m \times m$ (siendo m, el número de filas que maneja el proceso) conocida como porción diagonal (ya que la diagonal de la matriz y la submatriz coinciden) y otra submatriz de tamaño $m \times N$ con el resto de la porción local de la matriz, tal y como aparece en la figura 3.2.

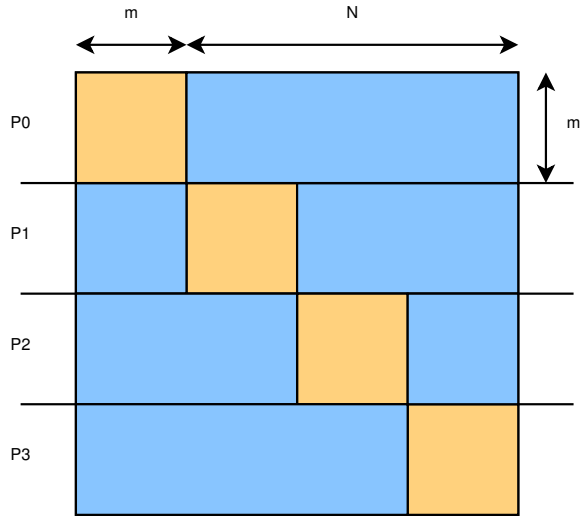

Figura 3.2: División de la matriz para la preasignación. Las regiones en naranja se corresponden con la sección diagonal de la submatriz local. Las regiones en azul se corresponden con la sección no diagonal.

Así, el segundo y tercer parámetro se corresponden con la sección diagonal, mientras que el cuarto y quinto se corresponden con el resto. Por

otro lado, se puede realizar la preasignación de manera idéntica para todas las filas (parámetros segundo y cuarto, enteros) o con valores distintos para cada una de las filas (parámetros tercero y quinto, vectores de enteros). De esta manera, en el ejemplo anterior se hace la preasignación con 4 elementos no nulos en todas las filas de la sección diagonal y ningún elemento no nulo en el resto de la matriz.

Una vez se ha creado la matriz, se pueden establecer los valores de la matriz con `MatSetValues` y posteriormente realizar el ensamblado con `MatAssemblyBegin` y `MatAssemblyEnd` de manera análoga al caso de los vectores. Además, obviamente también se proporcionan funciones para operar con la matriz, tales como el cálculo de la norma (`MatNorm`), el escalado (`MatScale`) y, por supuesto, la liberación de memoria (`MatDestroy`). Se puede encontrar la documentación de las funciones, así como multitud de ejemplos y tutoriales en [5].

3.3. Mallas estructuradas

En muchos problemas que se modelan mediante ecuaciones de derivadas parciales, es común representar el dominio mediante mallas estructuradas rectangulares. En estos casos, una vez se ha realizado la distribución de los datos, suele ser necesaria información sobre elementos que no se encuentran en local (generalmente elementos contiguos almacenados en nodos vecinos, conocidos como *ghost nodes*), antes de realizar ciertas operaciones locales. Para ello, PETSc proporciona una estructura de datos específica: `DA` (*Distributed Array*). Los *arrays* distribuidos se utilizan junto con los vectores y están diseñados para comunicar la información necesaria que no está almacenada localmente en mallas estructuradas, tal y como se muestra en la figura 3.3. Es por esto que no se deben utilizar para almacenar matrices o en dominios no estructurados.

Al igual que con el resto de objetos de PETSc, los *arrays* distribuidos se deben crear con `DACreate` y `DASetFromOptions`, aunque cabe destacar que la librería proporciona los constructores `DACreate1d`, `DACreate2d` y `DACreate3d` para crear el objeto especificando el número de dimensiones si este se conoce en tiempo de compilación y no se desea cambiar dinámicamente en tiempo de ejecución. Además, cabe destacar que el uso de los *arrays* distribuidos está fuertemente ligado al uso de vectores. Como se ha comentado anteriormente, las porciones locales del vector deben reservar espacio para los nodos vecinos que serán utilizados en el cálculo. De esta manera, la llamada a la función `DAGetLocalVector` nos permitirá obtener un vector local con la memoria necesaria para almacenar todos estos datos. De manera similar, `DAGetGlobalVector` nos servirá para obtener un vector que contenga espacio para toda la información del dominio, distribuida de manera adecuada entre todos los procesos. Por último, es importante notar

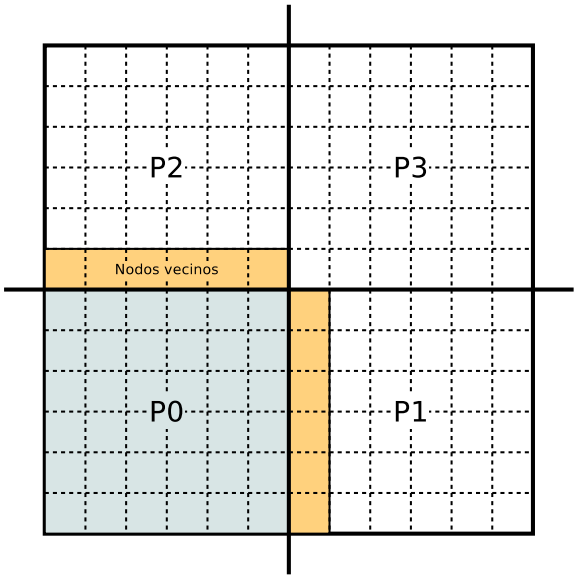

Figura 3.3: Malla estructurada dividida entre cuatro procesos. Las celdas en gris corresponden a la porción local de P0. Las celdas en naranja se corresponden con celdas de procesos distintos a P0 y que se necesitan en este último.

que cuando los datos globales deben ser distribuidos a las copias locales o viceversa, se debe hacer un manejo especial de los nodos vecinos, ya que éstos se encuentran replicados en varios nodos. Esto se realiza mediante las rutinas `DAGlobalToLocal` y `DALocalToGlobal` que, mediante un parámetro, permite especificar si en la comunicación se deben insertar los valores de los nodos vecinos, reemplazando los antiguos, o por el contrario, se debe añadir el valor al dato existente.

3.4. *Solvers* lineales

El objeto `KSP` es el núcleo de PETSc ya que proporciona un acceso uniforme a todo el conjunto de métodos de resolución de sistemas de ecuaciones lineales, paralelos y secuenciales, directos e iterativos. `KSP` está destinado a resolver sistemas no singulares de la forma $Ax = b$, donde A es la matriz de coeficientes, b el vector del lado derecho y x es el vector solución. Para ello, permite entre otros el uso de multitud de métodos de Krylov en conjunción con diversos precondicionadores, lo cual se ha convertido en el proceso más común en la resolución de sistemas de ecuaciones lineales de manera iterativa.

En cuanto al uso de los objetos `KSP`, en primer lugar ha de crearse el objeto mediante la llamada `KSPCreate`. Posteriormente, se deben especificar los

operadores del objeto KSP mediante `KSPSetOperators`, esto es, la matriz de coeficientes y la matriz a partir de la cual se debe construir el precondicionador que, generalmente, suele coincidir. Tras establecer estas matrices, se debe hacer una llamada a `KSPSetFromOptions`, que establecerá los parámetros, tales como el tipo de método iterativo, precondicionador y múltiples parámetros de éstos, todo ello en tiempo de ejecución. A partir de este momento ya es posible resolver el sistema de ecuaciones especificando el vector del lado derecho del sistema y el vector donde se almacenará la solución mediante la función `KSPSolve`. El flujo de ejecución más simple para el uso de `KSP` se`KSP` sería el siguiente:

```
...
KSPCreate(PETSC_COMM_WORLD, &ksp);
KSPSetOperators(ksp, A, A, DIFFERENT_NONZERO_PATTERN);
KSPSetFromOptions(ksp);
KSPSolve(ksp, b, x);
...
```

donde `A` es la matriz de coeficientes, `b` el vector de términos independientes y `x` el vector solución. Además, el último parámetro de la llamada a `KSPSetOperators` especifica si la nueva matriz de coeficientes tiene o no el mismo patrón de elementos no nulos que en la anterior llamada a esta misma función. En el ejemplo, puesto que se establece la matriz de coeficientes una sola vez, este parámetro es irrelevante.

También cabe destacar que, al establecer por separado la matriz de coeficientes y el vector del lado derecho, se optimiza la creación del precondicionador en caso de que se deseen resolver más de un sistema de ecuaciones con la misma matriz de coeficientes y el mismo precondicionador.

Por otro lado, las opciones que se le pueden especificar al objeto KSP mediante línea de comandos (y que se aplican al ejecutar la instrucción `KSPSetFromOptions`) son muy numerosas. En primer lugar, es posible especificar el método de Krylov (u otros) que se desea utilizar en la resolución del sistema con la opción `-ksp_type` seguida del método, pudiendo elegir entre los métodos CG (`cg`), CGS (`cgs`), BiCG (`bicg`), GMRES (`gmres`), Richardson (`richardson`), Chebychev (`chebychev`) entre otros. Además, para cada uno de estos métodos se pueden especificar opciones particulares, tales como el factor de Richardson (`KSPRichardsonSetScale` o `-ksp_richardson_scale`) o el reinicio de GMRES (`KSPGMRESSetRestart` o `-ksp_gmres_restart`). Por otra parte, también es posible especificar el precondicionador mediante la opción `-pc_type`, pudiendo elegir entre Jacobi (`jacobi`), Block Jacobi (`bjacobi`), SOR (`sor`), LU incompleta (`ilu`), Cholesky incompleta (`icc`) entre otros. Para cada uno de estos precondicionadores existen parámetros concretos que se pueden especificar tales como la omega de SOR (`PCSORSetOmega` o `-pc_sor_omega`).

3.5. *Solvers* no lineales

El módulo `SNES` de PETSc proporciona un conjunto de estructuras de datos y operaciones para la resolución de problemas no lineales. Así, esta librería se construye sobre los elementos comentados en apartados anteriores (vectores, matrices, *solvers* lineales,...) con el fin de proporcionar al usuario una manera flexible y fácil de establecer los parámetros del objeto y resolver el sistema de su problema particular.

El uso del módulo `SNES` es análogo al resto de objetos de PETSc, creándolos con `SNESCreate`, `SNESSetFromOptions`, resolviendo con `SNESSolve` y liberando memoria con `SNESDestroy`.

Sin embargo, esta librería se basa en el método de Newton para manejar la componente no lineal del sistema. Este método consiste en la aplicación de la serie de Taylor sobre el sistema no lineal $F(x) = 0$. Así, se utilizan los dos primeros términos de la serie para aproximar la función:

$$F(x) \approx F(x_0) + F'(x_0)(x - x_0) \tag{3.1}$$

e igualando a 0 se obtiene la estimación de la nueva solución:

$$x_1 = x_0 - \frac{F(x_0)}{F'(x_0)} \text{ o, en general, } x_k = x_{k-1} - \frac{F(x_{k-1})}{F'(x_{k-1})} \tag{3.2}$$

La solución final se obtiene repitiendo este proceso hasta que el cambio en la solución $x_k - x_{k-1}$ es aceptado por el criterio de convergencia. Así pues, el principal inconveniente del método es que se necesita calcular la derivada de la función, conociéndose esta derivada como el Jacobiano de la función inicial, lo cual plantea dos problemas. Por un lado, la propia evaluación del Jacobiano se suele convertir en la parte que mayor tiempo de computación requiere, mientras que por otro lado, muchos sistemas se construyen a partir de ecuaciones implícitas o que simplemente son tan complejas que la derivación se hace prácticamente imposible. De este modo, tal como se explica en [7], el método de Newton se ha utilizado muy pocas veces para resolver las ecuaciones de Navier-Stokes de manera directa, demostrándose en estos casos que, a pesar de que el método converge en muy pocas iteraciones, el tiempo empleado en el cálculo del Jacobiano lo convierte en una alternativa peor que otros métodos iterativos. Es por esto que en este trabajo se implementa la iteración de Picard para el problema no lineal y no se hace uso de la librería `SNES` de PETSc.

3.6. Integradores temporales

La librería `TS` (por sus siglas en inglés, *Time Steppers*) proporciona un entorno de trabajo para la programación de soluciones escalables, tanto en

problemas de ecuaciones de derivadas parciales dependientes del tiempo, como en problemas estacionarios con del tiempo.

De manera análoga al caso de los *solvers* no lineales, en el problema particular que se pretende resolver en este trabajo, la integración temporal es compleja y se requiere de una gran flexibilidad, por lo que el código cuenta con una implementación propia y no se hace uso de la librería `TS`.

Capítulo 4

MICSc

En un principio, el proyecto se inicia con la idea de desarrollar por completo una nueva aplicación para la resolución numérica de las ecuaciones de Navier-Stokes. Sin embargo, poco tiempo después, y tras revisar las distintas aplicaciones de la librería PETSc, se fija la atención en una tesis realizada en la Universidad de Zaragoza por A. Cubero bajo la tutela de N. Fueyo [14]. Este código se centra exactamente en el objetivo principal del proyecto de investigación RHELES y finalmente, tras contactar con los autores, se decide aprovechar el trabajo ya realizado y se adopta MICSc como punto de partida para el desarrollo del proyecto.

Así pues, MICSc es el nombre del código escrito en C, de aproximadamente 15000 líneas en ficheros fuente (incluyendo documentación y espacios), que hace uso de PETSc para la resolución acoplada e implícita de las ecuaciones de Navier-Stokes. En este capítulo se detallan las principales características de este código, muchas de las cuales se han mantenido mayormente inalteradas y son sobre las que se han aplicado correcciones y nuevas funcionalidades (comentadas en el siguiente apartado) con el fin de alcanzar los objetivos del proyecto. De esta manera, en primer lugar trataremos la forma en que se abordan los elementos más problemáticos para la implementación y las decisiones que se han tomado y, al final del capítulo, entraremos con más detalle en la implementación concreta de la solución.

4.1. Discretización

La discretización en MICSc se realiza siguiendo el modelo de volúmenes finitos. Este método se fundamenta en la división estructurada del dominio en un conjunto de celdas sobre las que se aplican individualmente las ecuaciones de gobierno del flujo. Sin embargo, existen dos aproximaciones distintas a la hora de ubicar los valores del problema sobre la malla: mallas decaladas y mallas colocalizadas (se puede ver una comparativa de ambas en [15]). La principal diferencia es que en las mallas colocalizadas el valor

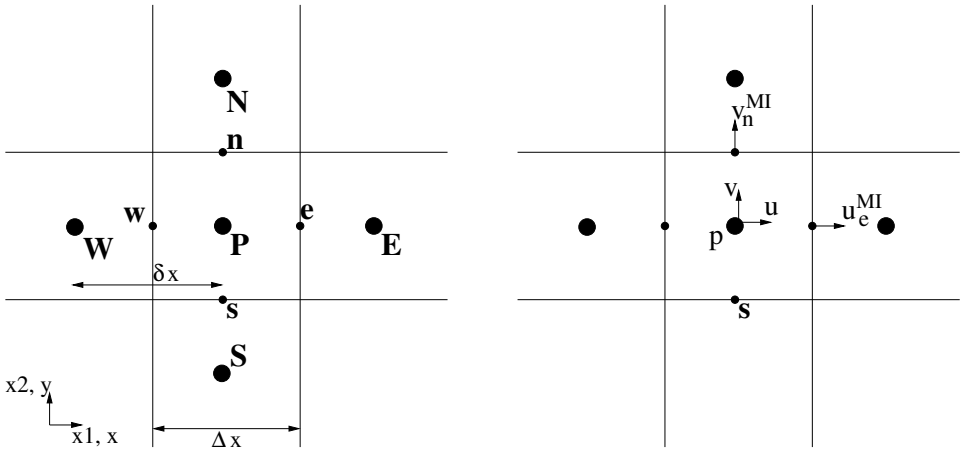

Figura 4.1: Notación de las celdas, en mayúscula los centros de las celdas y en minúscula las caras (izquierda). Distribución de las variables, subíndice indicando el vecino, superíndice MI indicando la Interpolación del Momento de una componente de la velocidad.

de todas las incógnitas se almacena en el centro de la celda, lo cual puede llevar al desacoplamiento de la presión (de la que sólo aparece el gradiente) y la velocidad y, por tanto, a resultados con poco sentido físico. Por contra, en las mallas decaladas se transportan las componentes de la velocidad a las caras de las celdas, manteniendo los valores escalares en el centro.

En el caso de MICSc, se hace uso de la Interpolación del Momento [16] (MI por sus siglas en inglés, *Moment Interpolation*), que consiste en almacenar tanto las componentes de la velocidad como los escalares en las celdas, y definiendo unas nuevas variables sobre las caras (velocidades de convección) que se obtienen mediante una interpolación concreta de las componentes de la velocidad. La Figura 4.1 ilustra la disposición de estas variables y su notación.

Por otro lado, una vez decidida la implementación de la malla y la distribución de las variables sobre ella, discretizaremos las ecuaciones. Es posible demostrar que cualquier ecuación de conservación sobre una variable ϕ es posible expresarla de manera semi-discretizada como la suma de los términos temporal, convectivo, difusivo y fuente de la siguiente manera:

$$\frac{\partial \rho\phi}{\partial t}\Big|_P V_P + \sum_{nb(P)} m_{nb}\phi_{nb} - \sum_{nb(P)} \Gamma^{\phi}_{nb} A_{nb} \frac{\partial \phi}{\partial x_{nb}}\Big|_{nb} = S^{\phi}_P V_P$$

donde V_P es el volumen de la celda P; los sumatorios sobre $nb(P)$ recorren todas las caras de la celda; m_{nb} es el flujo a través de la cara nb, definido como $m_{nb} = \rho_{nb} A_{nb} v_{nb}$ donde v_{nb} es la componente de la velocidad normal

a la cara nb y $A_{nb} = A\hat{n} \cdot \hat{e}$; Γ^{ϕ}_{nb} es un coeficiente de difusión para la variable ϕ; x_{nb} representa la dirección normal a la cara nb y S^{ϕ}_{P} engloba los términos fuente (o sumidero) de ϕ.

Llegados a este punto, la discusión sobre las diferentes maneras de discretizar los distintos términos se puede encontrar en [14] y excede los límites de este trabajo. Únicamente destacar que para el término temporal se utilizan esquemas temporales implícitos (Euler, Adams-Moulton, ...) y para el término convectivo se utilizan esquemas convectivos de alto orden (SMART, QUICK, ...).

Sin embargo, sí que es importante destacar el hecho de que, una vez discretizados estos términos, las ecuaciones de conservación del momento son realmente apropiadas para despejar las componentes de la velocidad. No obstante, la ecuación restante (conservación de la masa) no resulta adecuada para la presión, puesto que no aparece dicha variable y el sistema algebraico resultante presenta un 0 en la diagonal. Nuevamente, existen diversas aproximaciones y multitud de estudios sobre la solución de este problema que exceden el ámbito del proyecto. Para el caso concreto de MICSc, la solución pasa por derivar una ecuación de tipo Poisson a partir de la ecuación de continuidad. Para más detalles matemáticos y de implementación sobre esta solución, ver [14] [17].

4.2. Entrada/Salida

Otro aspecto importante de la aplicación es la forma en que recibe los datos y produce los resultados. Al margen de las opciones que proporciona PETSc tanto para la entrada de parámetros como para la salida de resultados, es necesario contar con otro tipo de datos y formatos de salida que se especifican en los siguientes apartados.

4.2.1. Entrada

Uno de los ficheros de entrada más importantes es el que contiene la malla que define los volúmenes finitos que se utilizarán en la resolución del problema. Aunque actualmente se sigue un formato propio para este fichero, se consideran otras alternativas como trabajo futuro (ver 7.2). En cuanto al formato propio de este fichero, actualmente se define de la siguiente manera:

```
<{0,1}>
<num_divisiones_eje_X>
<num_divisiones_eje_Y>
<num_divisiones_eje_Z>
<lista_coords_eje_X>
<lista_coords_eje_Y>
```

```
<lista_coords_eje_Z>
<lista_condiciones_de_contorno>
```

El primer valor representa si la malla es rectangular (0) o cilíndrica (1). Aunque actualmente sólo se soportan mallas rectangulares, también es posible que en un futuro se implementen mallas cilíndricas. Las siguientes tres líneas contienen un único número entero por línea, que define el número de divisiones para cada uno de los ejes. Posteriormente deben aparecer tres líneas con una lista de coordenadas para cada uno de los ejes. El tamaño de la lista se deberá corresponder con el número definido en las líneas anteriores y deberán estar formados por números reales separados por espacios. En este punto es importante destacar que se asume la coordenada $[0, 0, 0]$ como origen de la malla y no es necesario incluirla en este fichero. Por último, aparece una lista de condiciones de contorno donde cada una de éstas ocupa una línea distinta con el siguiente formato:

```
<x> <nx> <y> <ny> <z> <nz> <vecino> <tipo> <porosidad>
```

Los primeros 6 parámetros son de tipo entero y especifican la región de la malla sobre la que se define la condición de contorno ($[x, y, z] - [x + nx, y + ny, z + nz]$); el siguiente parámetro especifica sobre qué cara de las celdas se define la condición ({*north, south, east, west, high, low, none*}); el tipo define la condición de contorno ({*wall, moving_wall, inlet, outflow, outpress_newman, outpress_extrap, symmetry, fix*}); y la porosidad es un número real en el rango [-1, 1].

Además de este fichero, existe otro fichero de entrada donde se le pueden especificar las mismas opciones que en línea de comandos. Esto es útil cuando se quieren establecer unos parámetros por defecto en el fichero y posteriormente realizar distintas pruebas cambiando algún parámetro, dando prioridad a la línea de comandos sobre el fichero. El conjunto de todas las opciones disponibles se puede obtener mediante la opción `-help` en línea de comandos.

4.2.2. Salida

PETSc permite la escritura de vectores y matrices en formato binario de manera transparente y con apenas unas pocas líneas de código. Sin embargo, de cara a realizar un análisis de los resultados o a importar éstos en una aplicación de visualización, este formato se vuelve inútil. Para remediar esto, MICSc proporciona la posibilidad de compilar y ejecutar las simulaciones con soporte para SILO y HDF5.

HDF5 es un formato de almacenamiento de datos de tipo jerárquico [18]. Básicamente se compone de *grupos* y *conjuntos de datos*. Los conjuntos de datos almacenan valores en vectores multidimensionales, mientras que los grupos pueden contener otros grupos o conjuntos de datos. De esta manera,

la idea tras este formato es la misma que la de los directorios en un sistema UNIX.

Por otro lado, SILO ([19]) es una librería que proporciona un conjunto de llamadas de más alto nivel que implementa su funcionalidad mediante llamadas a HDF5. De esta manera es posible escribir vectores multidimensionales o mallas en HDF5 mediante llamadas a SILO del tipo `DBPutQuadVar` o `DBPutQuadMesh`. Así, SILO se encarga de crear los grupos y conjuntos de datos con sus atributos y metadatos asociados de manera que luego se pueden visualizar en aplicaciones libres como `VisIt` ([20]).

Por último, cabe destacar que en el grupo de investigación también existe la necesidad de poder visualizar los datos con ciertas aplicaciones que no soportan SILO, por lo que también existe un conversor que transforma los datos en formato binario o SILO a formato Tecplot ([21]).

4.3. Sistema no lineal

Uno de los principales problemas que plantean las ecuaciones de Navier-Stokes 2.1 son la componente no lineal. En este punto, se deben tomar unas decisiones que comprenden la linealización de las ecuaciones, el método a seguir o el criterio de convergencia. Así, tal y como se comenta en 3.5, hay casos en los que el método de Newton no es recomendable. Es por esto que para la implementación de la resolución del sistema de ecuaciones en MICSc se optó por seguir una estrategia distinta que se detalla en los siguientes apartados.

4.3.1. Linealización

Como se ha explicado anteriormente, las ecuaciones de Navier-Stokes plantean una complejidad debida a la no linealidad de las ecuaciones, que se plasma en términos tales como el término convectivo de la ecuación de conservación del momento:

$$\frac{\partial(\rho u_j u_i)}{\partial x_j}.$$

En estos casos, con el fin de obtener un sistema algebraico de ecuaciones lineales que se pueda resolver con cualquier método iterativo como los basados en los subespacios de Krylov (ver 2.4 para más detalles sobre los métodos de Krylov), en MICSc se considera el flujo de masa como conocido, lo cual resulta en la siguiente aproximación:

$$\rho u_j u_i \approx (\rho u_j) u_i^{\circ}$$

donde el superíndice $\circ$ denota que los valores utilizados son obtenidos de los resultados de una iteración anterior. De esta manera, a partir de los

valores anteriores de ciertas variables, es posible linealizar las ecuaciones de manera aproximada. En los siguientes apartados se explica cómo se trata esa aproximación, así como la determinación de cuándo esa aproximación es suficientemente exacta.

4.3.2. Método de Picard

Como se ha comentado en el apartado anterior, para la linealización de las ecuaciones de Navier-Stokes se consideran conocidas ciertas variables a partir de ciertos resultados anteriores. Concretamente el procedimiento, conocido como método o iteración de Picard, consiste en partir de una solución arbitraria del problema, utilizarla como valor conocido para la resolución del sistema, y corregir la solución para obtener una solución más aproximada.

Este método se basa en el concepto de la iteración del punto fijo, explicado anteriormente en 2.4. Recordemos que esta iteración se define sobre una función $F : \mathbb{R}^n \to \mathbb{R}^n$ de forma que esta función genera una sucesión x_0, x_1, x_2, ... que converge a un vector x tal que $x = F(x)$. Para el caso concreto de nuestra aproximación de las ecuaciones de Navier-Stokes, la función F se define como:

$$F(\phi) = \phi + \Delta\phi$$

y, por tanto, la sucesión generada vendrá dada por

$$\phi_{n+1} = \phi_n + \Delta\phi, \quad n = 0, 1, 2, ...$$

donde ϕ es la solución del sistema y $\Delta\phi$ es la corrección que debe aplicarse al resultado obtenido en la iteración anterior. Así, para obtener dicha corrección, se resuelve el sistema de ecuaciones en forma de corrección o delta:

$$A\Delta\phi = r$$

donde el vector incógnita $\Delta\phi$ contiene la corrección $\phi_{n+1} - \phi_n$ necesaria para la iteración de Picard y el término de la derecha es el vector residuo $r = b - A\phi$ que proviene de la discretización de las ecuaciones de Navier-Stokes.

4.3.3. Convergencia

Siguiendo la linealización y el método de Picard de los apartados anteriores, a partir de una aproximación inicial de la solución, y resolviendo el sistema de ecuaciones en forma delta sucesivamente, se obtiene una serie de aproximaciones de la solución que deben converger a la solución exacta del problema. Así, es necesario imponer unas condiciones que determinen cuándo debe parar este proceso.

Por una parte, en MICSc se imponen dos condiciones que afectan al número de iteraciones realizadas y no dependen de los resultados obtenidos: el número mínimo y máximo de iteraciones que se deben realizar. En este punto es importante aclarar que estas iteraciones propias del método de Picard que aproximan progresivamente la solución del problema se conocen como *iteraciones exteriores*, debido a que cada una de estas iteraciones comprende la resolución de un sistema de ecuaciones lineal que se resuelve mediante un objeto `KSP` de PETSc, el cual a su vez también realiza un conjunto de iteraciones, conocidas como *iteraciones interiores*.

Y por otro lado, es necesario determinar cuándo la aproximación de la solución es suficientemente buena y, por tanto, consideramos que el sistema ha convergido. Para esto, en MICSc se imponen dos condiciones que se han de dar simultáneamente. Por un lado, el vector de corrección $\Delta\phi$ ha de ser suficientemente pequeño, mientras que por el otro lado, el vector residuo R ha de ser despreciable también. En ambos casos, el criterio que se sigue es la 2-norma de cada uno de los vectores y la cota máxima se puede determinar en tiempo de ejecución mediante la línea de comandos.

4.4. Estructuración del código

En esta sección vamos a tratar las estructuras de datos que existen en MICSc, así como el número de instancias, la relación entre cada una de ellas y el flujo de ejecución que se sigue con el fin de simular el problema.

4.4.1. Definición de las estructuras de datos

En primer lugar, definiremos las estructuras de datos que, salvo las excepciones comentadas en el capítulo siguiente, se han mantenido inalteradas desde el inicio del proyecto. Básicamente, las estructuras de datos se corresponden con una agrupación de diversos datos que se corresponden de manera unívoca con elementos propios del problema a resolver. Así pues, nos encontramos con las siguientes definiciones de estructuras de datos, en orden alfabético:

St_Bcond:

```
typedef struct St_Bcond {
  PetscInt id, ineighb;
  St_Patch *Patch;
  BcondType bCondType;
  St_BcondApplication *applicationList;
  struct St_Bcond *Next;
} St_Bcond;
```

Condición de contorno, definida sobre una región local de la malla (`Patch`). Incluye un identificador (`id`), el tipo (`bCondType`), la cara de

la celda (`ineighb`) sobre la que se aplica y una lista de aplicaciones que representan los distintos efectos de esta condición de contorno sobre el sistema de ecuaciones lineales. Además, se crea una lista de condiciones de contorno que se puede recorrer mediante `Next`.

St_BcondApplication:

```
typedef struct St_BcondApplication {
  PetscInt ivar, ieq;
  St_Prop *CoeffA, *CoeffB;
  struct St_BcondApplication *next;
} St_BcondApplication;
```

Aplicación de una condición de contorno sobre el sistema de ecuaciones. Contiene el índice de la variable (`ivar`) y ecuación (`ieq`) sobre la que se aplica y las propiedades para evaluar los valores a introducir (`CoeffA`, `CoeffB`). También contiene un puntero (`next`) que permite recorrer la lista desde `St_Bcond`. La idea tras esta estructura de datos es que exista una única condición de contorno (`St_Bcond`) de un determinado tipo, aplicada sobre una determinada región, pero que pueda tener diferentes aplicaciones sobre el sistema de ecuaciones.

St_Geom:

```
typedef struct {
  PetscReal *Surface[3], *Edge[3], *Volume;
  PetscReal *wallDistance, *normalVec[3];
} St_Geom;
```

Propiedades geométricas de la malla local. Incluye la superficie del área de las caras de las celdas (`Surface`), la longitud de las aristas de dichas celdas (`Edge`) y el volumen (`Volume`). Además, con el fin de implementar el modelo de van Driest (5.2.3), se incluyen dos vectores (`wallDistance`, `normalVec`) que para cada celda contienen la distancia a la pared más cercana y un vector unitario normal a la dirección de la propia celda a la celda de pared más cercana.

St_Global:

```
typedef struct {
  MPI_Comm comm;
  PetscInt NPatch, NBcond, NPorosity, NProp;
  St_Patch *FirstPatch, *LastPatch;
  St_Bcond *FirstBcond, *LastBcond;
  St_Prop *FirstProp, *LastProp;
} St_Global;
```

Parámetros globales del problema. Contiene información del comunicador MPI y un contador del número de objetos creados para las

regiones de la malla (`NPatch`), condiciones de contorno (`NBcond`), porosidades (`NPorosity`) y propiedades (`NProp`). Por último, contiene varias listas para acceder a todas las regiones de la malla (`FirstPatch` y `LastPatch`), condiciones de contorno (`FirstBcond` y `LastBcond`) y propiedades (`FirstProp` y `LastProp`).

St_Grid:

```
typedef struct {
  PetscInt Nnode[3];
  PetscReal *Xnode[3];
  ktypeGrid ktype;
} St_Grid;
```

Definición y propiedades de la malla. Esta estructura almacena el número de nodos a lo largo de cada eje (`Nnode`) así como sus coordenadas (`Xnode`). Además, el parámetro `ktype` puede tomar los valores del tipo enumerado `ktypeGrid` y determina el tipo de malla, si es cartesiana (`RECT`) o cilíndrica (`CYLI`).

St_Input:

```
typedef struct {
  PetscInt log_level, log_ievery, out_ievery, out_idtevery;
  PetscInt startInstantMeans, startVarianceMeans;
  PetscTruth initial_field, final_field;
  PetscTruth printProps;
  PetscInt max_outer_its, min_outer_its;
  PetscReal resnorm_max, corrnorm_max;
  RestartType k_restart;
  PetscInt init_it;
  PetscInt k_wrap;
  PetscInt n_ghost;
  PetscReal gravity;
  PetscInt n_order_trans;
  PetscReal deltat, tinit, tfinal;
  PetscReal uref, mu, tref, rho, cs0, dpdx;
  PetscInt ndim, nvar, ncoupvar;
  PetscInt contEqMethod;
  PetscErrorCode (*inletFunction)(PetscInt*, PetscReal*);
  PetscTruth useLES, useKEMOD, useT;
  St_InputBcond *firstBcond, *lastBcond;
} St_Input;
```

Esta estructura de datos se encarga de almacenar todos los parámetros de entrada especificados por el usuario mediante un fichero de entrada. Es la estructura de datos que más cambios ha sufrido a lo largo de

todo el desarrollo debido a que almacena los parámetros de entrada del problema y éstos se han ido modificando, tanto por el hecho de añadir nueva funcionalidad a la aplicación como por la extracción de múltiples parámetros que en el inicio se especificaban dentro del código fuente y se extrajeron al fichero de parámetros de entrada.

St_InputBcond:

```
typedef struct St_InputBcond {
  PetscInt x, y, z, nx, ny, nz, ineighb;
  BcondType type;
  PetscReal porosity;
  struct St_InputBcond *next;
} St_InputBcond;
```

Contiene la definición de una condición de contorno tal y como se especifica en el fichero de entrada. Esta estructura únicamente tiene utilidad para almacenar los valores temporalmente antes de crear las estructuras `St_Bcond` asociadas. Sus atributos son el punto inicial (`x`, `y`, `z`) de la región sobre la que se define la condición de contorno, así como el número de celdas de dicha región en cada dirección (`nx`, `ny`, `nz`). También contiene el tipo de condición (`type`), la porosidad (`porosity`) y un puntero para crear listas (`next`).

St_Interp:

```
typedef struct {
  PetscReal **linear, **upwind;
  PetscInt **icell_upw, **icell_uupw, **icell_dow;
} St_Interp;
```

Esta estructura de datos es útil para el cálculo de los coeficientes de interpolación del esquema convectivo. Para cada dimensión y cada celda contiene datos como el índice local de las celdas *upwind* (`icell_upw`), *up-upwind* (`icell_uupw`) y *downstream* (`icell_dow`). También incluye los coeficientes para la interpolación tanto lineal (`linear`) como *upwind* (`upwind`).

St_Local:

```
typedef struct {
  PetscMPIInt rank;
  MeshRegion meshRegion, ghostMeshRegion;
  PetscInt **icell_neighb;
  PetscReal **porosity;
  St_Porosity *FirstPorosity, *LastPorosity;
} St_Local;
```

Parámetros locales del problema. Contiene el índice del procesador (`rank`) e información sobre la porción de malla que maneja, tanto considerando los nodos *ghost* (`ghostMeshRegion`), como sin considerarlos (`meshRegion`). También se almacena una lista de porosidades (`FirstPorosity, LastPorosity`) que se aplican sobre la porción local de la malla. Por último, para cada celda, y en función del número de dimensiones del problema, se almacenan los índices de sus vecinos (`icell_neighb`) y el valor numérico de su porosidad (`porosity`).

St_MeshRegion:

```
typedef struct {
  PetscInt nCells[3];
  PetscInt lowerCorner[3];
  PetscInt upperCorner[3];
  PetscInt totalCells;
} St_MeshRegion;
```

Define una región rectangular tridimensional de la malla, especificando el punto origen (`lowerCorner`), el punto final (`upperCorner`), así como el número de celdas que contiene dicha región en cada dirección (`nCells`) y en total (`totalCells`).

St_Patch:

```
typedef struct St_Patch {
  PetscInt id;
  PetscInt x, y, z, nx, ny, nz;
  DA da;
  MPI_Comm comm;
  struct St_Patch *Next;
} St_Patch;
```

Esta estructura de datos contiene un identificador (`id`), así como las dimensiones globales de una determinada región de la malla (`x`, `y`, `z`, `nx`, `ny`, `nz`). Además, contiene un objeto `DA` y el comunicador asociado (`comm`) para manejar las propiedades que se definan sobre esta región de la malla. Por último, proporciona un puntero para construir listas (`Next`).

St_Porosity:

```
typedef struct St_Porosity {
  PetscInt id, ineighb;
  St_Patch *Patch;
  PetscReal Cons;
  struct St_Porosity *Next;
} St_Porosity;
```

Esta estructura almacena parámetros útiles para relajar o limitar flujos a través de las caras (con un factor entre 0 y 1). Concretamente, contiene un identificador (`id`), el índice de la cara sobre la que se aplica el factor (`ineighb`), el conjunto de celdas sobre la que se aplica la porosidad (`Patch`) y el factor por el que se multiplica el flujo (`Cons`). Cabe destacar también el hecho de que cada vez que se crea una nueva porosidad, ésta se añade a una lista de porosidades locales que se puede recorrer haciendo uso del atributo `Next`.

St_Prop:

```
typedef struct St_Prop {
  PetscInt id;
  St_Patch *Patch;
  ktypeProp ktype;
  ktypeIp kip;
  PetscErrorCode (*FluxLimiter)(PetscInt icell,
          PetscInt ineighb, PetscReal upwdelta,
          PetscReal dowdelta, PetscReal *value);
  PetscReal linrlx;
  PetscReal Cons;
  PetscErrorCode (*Funct)(struct St_Prop *Prop);
  PetscReal *array;
  Vec instantMeans, instantSums, varianceSums, values;
  PetscInt numReferences;
  struct St_Prop *Next;
} St_Prop;
```

Propiedad (constante o variable) asociada a una región de la malla (o *patch*). Entre sus atributos encontramos un identificador (`id`), la región de la malla sobre la que se aplica la propiedad (`Patch`), y el tipo de propiedad (`ktype`), que define si la propiedad es constante (`CONS`) o depende de la geometría (`GEOM`), de las incógnitas (`VARI`) o de los términos transitorios (`TRANS`). Además, contiene el tipo de interpolación (`kip`), que define la función utilizada para el limitador de flujo (`FluxLimiter`) y un valor de relajación (`linrlx`). Por último, si el tipo de propiedad es constante, el atributo `Cons` almacenará este dato constante y, en caso contrario hará uso de la función `Funct` para calcular los datos. También contiene objetos `Vec` donde se almacenarán los valores instantáneos de la propiedad (`values`), así como valores para el cálculo de medias (`instantMeans`, `instantSums` y `varianceSums`, ver 5.2.2). Es posible acceder al almacenamiento del vector de valores instantáneos mediante el atributo `array`. Puesto que una propiedad puede estar asociada a distintos objetos de la aplicación (principalmente a la `gamma` de las variables), existe un contador con el que se

determina el número de referencias a esta propiedad(`numReferences`). Además, cada vez que se crea una propiedad, se añade a una lista de propiedades locales, iterables mediante `Next`.

St_Sys:

```
typedef struct {
  Vec Phi, B;
  Mat A;
  DA da;
  KSP solver;
  PetscInt *ltog;
  PetscInt ndof;
  PetscInt ivar;
  PetscReal resnorm, resref, corrnorm, corref;
} St_Sys;
```

Estructura para la creación y resolución del sistema de ecuaciones distribuido. Contiene el vector incógnita (`Phi`), el vector del lado derecho (`B`), la matriz de coeficientes (`A`), el *array* distribuido `DA` de PETSc (`da`), el *solver* lineal (`solver`). Además, contiene un vector de enteros para emparejar los índices de las celdas del contexto local al contexto global (`ltog`). También almacenan los grados de libertad del sistema (`ndof`), que toma el valor 1 para el sistema segregado y el número de variables acopladas en el sistema acoplado. Así, sabiendo el grado de libertad y el índice de la primera variable que resuelve el sistema (`ivar`) se puede conocer el conjunto de variables que resuelve el sistema. Por último, también contiene valores de referencia (`resref`, `corref`) para la normalización de las tolerancias relativa y absoluta (`resnorm`, `corrnorm`).

St_Trans:

```
typedef struct {
  TransientOrder norder;
  PetscReal time;
  PetscInt idt;
} St_Trans;
```

Parámetros específicos de un flujo no estacionario, como el orden del esquema temporal (`norder`), que puede ser Euler de primer orden (`EUL1`), Euler de segundo orden (`EUL2`), Adams-Moulton de segundo orden (`ADM2`) o Adams-Moulton de tercer orden (`ADM3`). Además, también contiene el instante de tiempo actual a lo largo de toda la ejecución (`time`) y el número de pasos de tiempo efectuados (`idt`).

St_Var:

```
typedef struct St_Var {
```

```
  char name[128];
  PetscInt index;
  ktypeIp kip;
  PetscErrorCode (*FluxLimiter)(PetscInt icell,
          PetscInt ineighb, PetscReal upwdelta,
          PetscReal dowdelta, PetscReal *value);
  PetscReal valmin, valmax, linrlx, fdtrlx,
          resref, corref, resnorm, corrnorm;
  St_Prop *Valini, *Gamma;
  Vec locPhi;
  PetscReal *Array;
  struct St_Var *next;
} St_Var;
```

Contiene parámetros de una variable dependiente (o incógnita). Entre los datos que incluye podemos encontrar el nombre (`name`), el índice (`index`), así como el vector de PETSc con los valores locales de la variable (`locPhi`) y un puntero a los valores almacenados en dicho objeto (`Array`). Además, también contiene el tipo de interpolación (`kip`) que, de manera análoga al caso de `St_Prop`, define el limitador de flujo (`FluxLimiter`). Junto con todo esto, incluye los valores mínimo y máximo para la variable (`valmin`, `valmax`), así como coeficientes de relajación (`linrlx`, `fdtrlx`), valores de referencia (`resref`, `corref`) para la normalización de las tolerancias relativa y absoluta (`resnorm`, `corrnorm`) y dos propiedades utilizadas para el cálculo de los valores iniciales (`Valini`) y el coeficiente de difusión (`Gamma`). Por último, cada vez que se crea una variable nueva antes del inicio de la resolución del problema, ésta se añade a una lista enlazada haciendo uso del atributo `next`.

4.4.2. Instancias y relación de las estructuras de datos

A pesar de que a partir de las definiciones de las estructuras de datos se deja intuir el número de instancias que se crearán de cada una de ellas, en este apartado especificaremos de manera concreta dichas instancias:

St_Bcond: Las condiciones de contorno se especifican en el fichero de entrada y cada una de las condiciones de contorno se convierte en una instancia de la estructura `St_Bcond`.

St_BcondApplication: Como se ha comentado anteriormente, cada condición de contorno contiene diversas aplicaciones (`applicationList`) que introducen valores en el sistema de ecuaciones para cada ecuación y variable particular. Así pues existirá una instancia distinta para cada una de estas contribuciones y este número de instancias dependerá de

la condición de contorno concreta, por lo que se debería especificar para cada caso particular.

St_Geom: Los datos que contiene esta estructura de datos son propios de cada proceso, por lo que existirá una instancia por cada uno de ellos que contendrá la información de la porción local de la malla.

St_Global: Puesto que la información almacenada en esta estructura de datos es global, existirá un instancia en cada proceso que contendrá dicha información replicada en cada uno de ellos.

St_Grid: Análogamente al caso anterior, los datos que se incluyen en esta estructura de datos son globales, por lo que habrá una instancia idéntica en cada uno de los procesos.

St_Input: Nuevamente, la información de `St_Input` es global y habrá una misma instancia replicada en todos y cada uno de los diferentes procesos.

St_InputBcond: Puesto que esta estructura únicamente se utiliza para leer los datos de entrada del fichero para posteriormente crear las estructuras `St_Bcond`, existirá una instancia idéntica en todos los procesos por cada condición de contorno especificada por el usuario en el fichero de entrada.

St_Interp: En este caso, la estructura de datos contiene varios vectores cuyo tamaño depende del número de celdas en cada nodo local. Por tanto, existirá una instancia en cada uno de los procesos, pero la información que almacene cada instancia será diferente, puesto que las celdas que se manejan en cada nodo son distintas.

St_Local: Este caso es muy similar al de `St_Interp`. Existirá una instancia en cada proceso y dicha instancia incluirá la información sobre la región local de la malla que se maneja, así como listas de propiedades, condiciones de contorno y porosidades que se aplican en esa región y demás información local que, puesta en común con la información del resto de procesos, debe reunir todos los datos del dominio completo.

St_MeshRegion: No existen instancias independientes de esta estructura de datos, sino que se encuentran siempre ligadas como atributos de las estructuras `St_Local` y `St_Patch`.

St_Patch: Esta estructura de datos contiene datos de la región de la malla (*patch*). Es por esto que, por cada región que se deba definir, existirá una instancia en cada proceso con datos locales e información global replicada en todos y cada uno de ellos. Cabe destacar también que, salvo la región que define el domino completo, todas las instancias

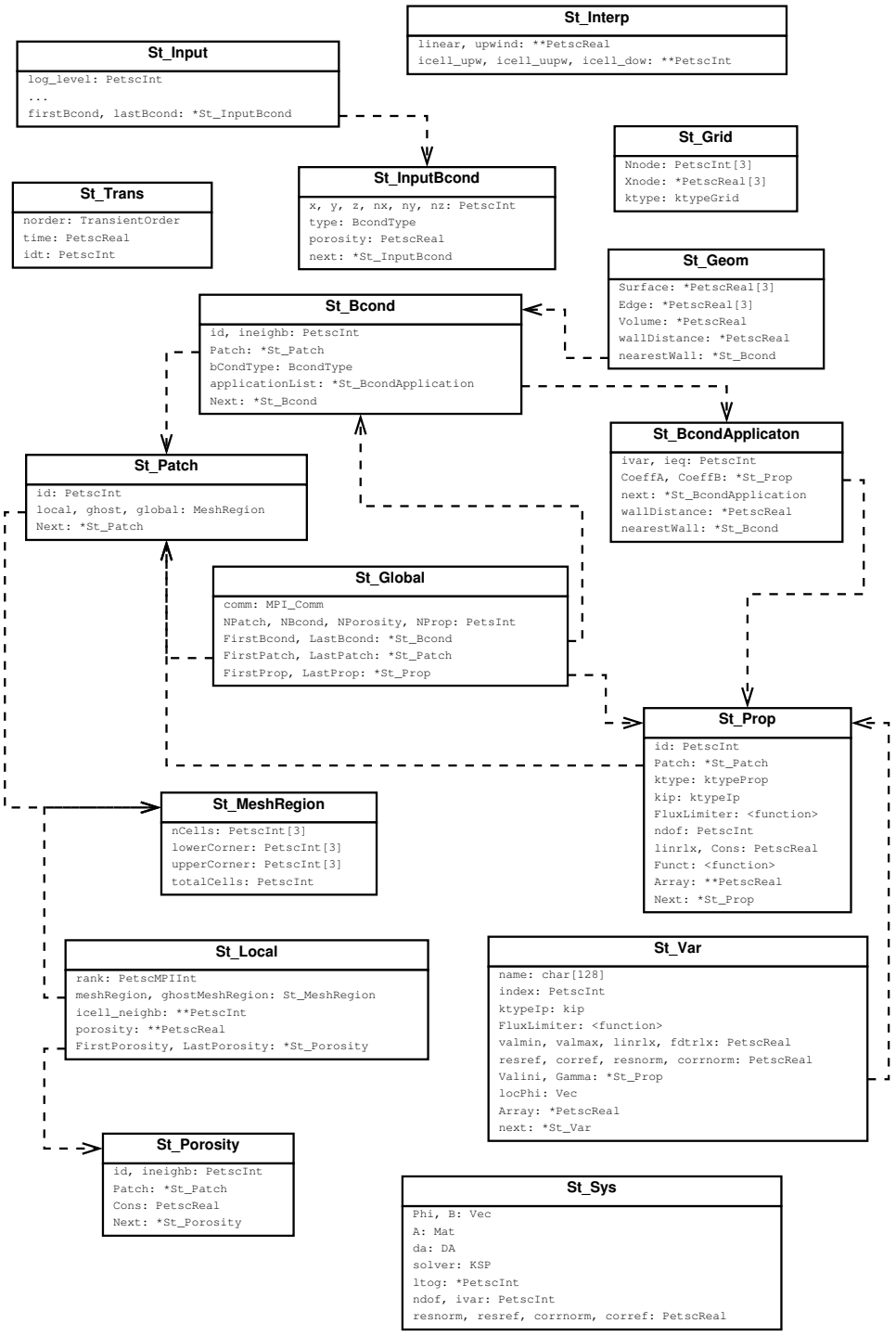

Figura 4.2: Relaciones entre las estructuras de datos en MICSc. Por simplicidad se obvian las relaciones recursivas.

de `St_Patch` están asociadas a otras instancias de estructuras de datos diferentes y, por tanto, el número total de instancias que se crearán de esta estructura dependerá del número de variables, condiciones de contorno y porosidades que se definan para cada problema particular.

St_Porosity: Las porosidades son estructuras de datos que se utilizan para modificar el flujo a través de las caras de las celdas y están siempre asociadas a condiciones de contorno. Así pues, existirá una porosidad por cada condición de contorno especificada en el fichero de entrada. Sin embargo, puesto que las condiciones de contorno y, por tanto, las porosidades se aplican sobre una región del dominio concreta, únicamente se crearán instancias en los procesos que contengan la región de aplicación (completa o parcialmente).

St_Prop: Esta estructura de datos se utiliza para evaluar una cierta propiedad en una región del dominio y almacenar los resultados obtenidos para cada celda local en un vector. Por este motivo, y de manera análoga al caso anterior, sólo se crearán instancias de `St_Prop` en los procesos que manejen celdas donde se aplique la propiedad de manera completa o parcial, almacenando el vector de manera distribuida en el segundo caso. Por otra parte, existen ciertas propiedades que se evalúan y se introducen en el sistema de ecuaciones de manera independiente, tales como los términos convectivo (`Conv`) y difusivo (`Diff`) o la densidad (`Rho`). Sin embargo, la mayor parte de las propiedades se encuentran asociadas a condiciones de contorno (`CoeffA`, `CoeffB`) o a variables (`Valini`, `Gamma`) y el número de instancias que se crearán dependerá del número de condiciones de contorno y variables que se creen y, por tanto, de cada problema particular.

St_Sys: Esta estructura de datos contiene principalmente la información del sistema de ecuaciones: matriz de coeficientes, vector del lado derecho, vector de incógnitas y el contexto del *array* distribuido (`DA`). Todos ellos son objetos de PETSc que se manejarán de manera distribuida en todos los procesos haciendo uso de la librería de manera transparente. Además, también contiene ciertos parámetros globales de problema, tales como valores numéricos globales del problema tales como valores de referencia (`resref`, `corref`) para la normalización de las tolerancias relativa y absoluta (`resnorm`, `corrnorm`) que se replicarán en todos los nodos. Por último, es destacable también que existirán dos instancias de esta estructura de datos en cada proceso: una de ellas se encargará de la resolución del sistema de ecuaciones acoplado y la otra, del segregado.

St_Trans: En este caso, los datos que incluye la estructura de datos son simplemente valores numéricos para el manejo de flujos no estacionarios

que se replicarán en todos los procesos, creando una única instancia idéntica en todos y cada uno de los procesos.

St_Var: Esta estructura de datos contiene varios parámetros (valores mínimo y máximo, relajaciones, ...) que se replicarán en todos los procesos, junto con un vector que contendrá el vector solución para las celdas que se manejen en cada uno de los procesos, así como las propiedades `Gamma` y `Valini` que se distribuirán tal y como se ha detallado anteriormente para la estructura `St_Prop`. Además, existirá una instancia con dichos datos por cada una de las variables que se defina en el problema (velocidades, presión, temperatura, ...)

Conociendo ya la definición de las estructuras y el número de instancias que se generan de cada una de ellas, pasamos a resumir y esquematizar las relaciones que existen entre cada una de estas estructuras de datos. En un principio, es importante determinar las estructuras de datos que únicamente contienen valores numéricos, objetos de PETSc y demás datos que no son estructuras de datos propias de MICSc. Éstas son: `St_Grid`, `St_Interp`, `St_Sys` y `St_Trans`.

Además, cabe destacar el hecho de que tanto `St_Global` como `St_Local` están relacionados con las propiedades, condiciones de contorno y/o porosidades, pero únicamente con un puntero al inicio de cada una de las listas de objetos.

Por otro lado, tanto `St_Bcond` como `St_Porosity` y `St_Prop` tienen una referencia a `St_Patch`, que define la región de la malla (*patch*) donde se aplican las condiciones de contorno, porosidades y propiedades respectivamente.

Por último, tanto `St_BcondApplication` como `St_Var` tienen referencias a `St_Prop`. Es importante notar que las condiciones de contorno y las variables tienen relación con las regiones de la malla y con propiedades que, a su vez, también se relacionan con estas regiones. En este caso, debido a la transitividad, se ha cuidado que la implementación de las diferentes referencias a regiones de la malla apunten exactamente a la misma instancia, por lo que no existe ninguna instancia que se replique de manera innecesaria en ningún nodo.

En la figura 4.2 se muestran tanto las estructuras de datos que no dependen de ninguna otra estructura de datos definida en MICSc como aquellas que si que tienen dependencias entre sí.

4.4.3. Flujo de ejecución

Como se ha comentado anteriormente, MICSc únicamente hace uso de los *solvers* lineales de PETSc e implementa su propia versión del método de Picard para las iteraciones exteriores y la integración temporal, por lo que esto se verá reflejado en el flujo de ejecución de la simulación.

Básicamente, este flujo consiste en una primera sección de lectura de parámetros de entrada, creación de estructuras e inicialización de valores.

Posteriormente, en la parte central se encuentra la resolución del sistema de ecuaciones. Esta sección contiene, en primer lugar, el bucle correspondiente a las iteraciones temporales, que al inicio del bucle se encarga de evaluar las propiedades dependientes del tiempo en cada iteración y, al final, actualizar el siguiente paso de tiempo en caso de que la simulación lo requiera, o haciendo falsa la condición del bucle en caso contrario. Dentro de este bucle se encuentra anidado otro bucle que se corresponde con las iteraciones del método de Picard. Aquí, en cada iteración se evalúan las propiedades que dependen de las variables y se resuelve el sistema acoplado (en caso de que haya variables acopladas) y tantos sistemas segregados como variables no acopladas existan. La resolución de este sistema consiste en la construcción de la matriz de coeficientes y el vector del lado derecho, introduciendo incrementalmente las contribuciones de las condiciones de contorno, la gravedad (en caso de que sea relevante), etc. Finalmente, dentro del bucle del método de Picard, y tras la resolución de todos los sistemas, se limitan los valores de las variables en caso de que hayan sobrepasado los límites mínimo y/o máximo.

Y, por último, la finalización de los objetos de PETSc, liberación de memoria y demás tareas de terminación del programa. Tanto en el Algoritmo 4 como en la Figura 4.3 se muestra el flujo de ejecución.

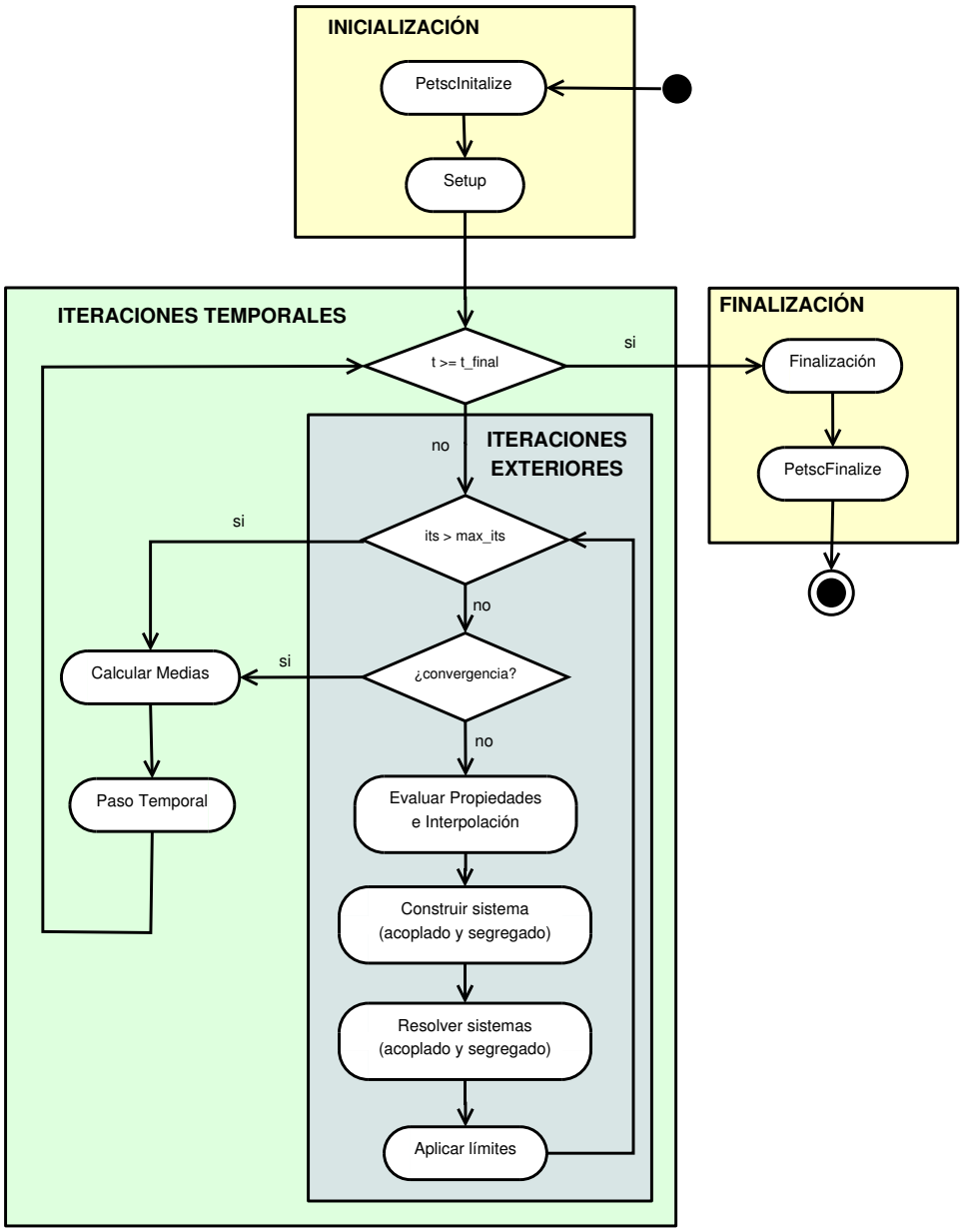

Figura 4.3: Flujo de ejecución en MICSc.

Algoritmo 4 Algoritmo de resolución de MICSc

```
Inicialización
while t < t_final do
  Evaluar propiedades dependientes del tiempo
  i = 0
  convergencia = false
  while i < its_max and not convergencia do
    Evaluar propiedades dependientes de las variables
    Evaluar interpolación de las variables
    if num_variables_acopladas > 0 then
      Reconstruir sistema de ecuaciones
      Resolver sistema de ecuaciones acoplado
    end if
    for i = (num_variables_acopladas + 1) : num_variables_total do
      Reconstruir sistema de ecuaciones
      Resolver sistema de ecuaciones segregado i
    end for
    Aplicar límites a valores de las variables
    i = i + 1
    convergencia = Evaluar convergencia
  end while
  Calcular medias
  t = t + Δt
end while
```

Capítulo 5

Desarrollos

En este capítulo se detallan los desarrollos realizados en este proyecto sobre el código adoptado. Así, el capítulo se encuentra dividido en dos secciones: la primera detalla aquellos desarrollos que no aportan nueva funcionalidad, sino que tienen como objetivo el mantenimiento y la optimización de la funcionalidad ya existente; y una segunda sección donde encontramos los desarrollos que han aportado nueva funcionalidad a MICSc.

5.1. Mantenimiento y optimización del código

Al inicio de adoptar MICSc como código base para el desarrollo del proyecto, se comenzó una labor de mantenimiento del código que tenía como objetivo tanto la eliminación de elementos obsoletos y la adopción de buenas prácticas de desarrollo, como la toma de contacto y asimilación de la arquitectura y el funcionamiento del código.

Entre estas tareas podemos encontrar algunas de ellas que, aunque costosas, no revisten ninguna gran relevancia desde el punto de vista de la implementación, tales como la eliminación de *warnings*, variables y funciones no utilizadas, etc., como la inclusión de la trazabilidad proporcionada por PETSc como herramienta de depuración durante la fase de desarrollo (más información en [5]).

En los siguientes apartados se explican de manera detallada aquellas tareas de mantenimiento que merecen una mención especial.

5.1.1. Estructuración de librería y casos

Al inicio del proyecto, MICSc era una aplicación con su propia función `main` que debía ser recompilada para cada problema distinto. De esta manera, existía un conjunto de ficheros que formaban el módulo de usuario y que se detallan a continuación:

userbcond.c : Creación de las condiciones de contorno asociadas al caso a resolver.

userdeclarat.h : Conjunto de macros necesarias para resolver el caso.

userfunct.c : Funciones definidas por el usuario, utilizadas por las propiedades especificadas en la creación de variables o condiciones de contorno.

userfunctions.h : Declaración de las funciones de usuario.

userprop.c : Creación de propiedades independientes.

uservar.c : Creación de todas las variables a resolver en el caso.

Esto se traducía en que cada vez que se deseaba resolver un caso distinto, era necesario modificar estos ficheros para incluir las nuevas características del problema. Una primera consecuencia de esto era que para poder especificar un nuevo caso, era necesario tener unas nociones básicas sobre el lenguaje de programación y el uso del módulo de usuario. Además de la incomodidad que esto representaba, junto con la necesidad de recompilar todo el código, este mecanismo era demasiado propenso a introducir errores por lo que se tomó la decisión de refactorizar MICSc para darle una estructura de librería con un núcleo compilado capaz de leer ficheros de entrada del usuario y resolver el sistema de ecuaciones asociado a los datos de entrada. Además, con la finalidad de no perder la potencia que proporcionaba la posibilidad de definir nuevas variables, condiciones de contorno, etc., se optó por que cada caso tuviera su función `main` y que dicha función pudiera acceder a un conjunto de funciones definidas de manera clara.

De esta manera, las macros definidas en `userdeclarat.h` se establecen como parámetros definidos por el usuario en el fichero de entrada (o por línea de comandos). La creación de las condiciones de contorno (`userbcond.c`) se automatiza de manera que en el fichero de entrada donde se especifica la malla, también se detalla para cada condición de contorno el tipo, las celdas sobre las que se aplica y toda la información necesaria para la completa definición de éstas. Las funciones (`userfunct.c`, `userfunctions.h`) se introducen en la librería como funciones accesibles por el usuario a la hora de crear sus propias variables y/o condiciones de contorno. Con respecto a las propiedades, se ha analizado la manera en que se introducen en el sistema de ecuaciones cuando se definen de manera independiente y, puesto que esta manera depende de la propiedad concreta y debe saberse a priori, se ha determinado que no tiene sentido permitir al usuario crear nuevas propiedades si éstas no se pueden introducir en el sistema de ecuaciones; de este modo, y puesto que para los distintos casos la única propiedad que se creaba en el fichero `userprop.c` era la densidad, se ha introducido como parte de la librería. Por último, se ha establecido en la librería una definición general

para las variables comunes a todos los problemas (velocidades, presión y temperatura) y se ha añadido la posibilidad de modificar estas variables y/o introducir nuevas variables mediante llamadas a funciones desde la función `main` de cada caso. Las funciones accesibles por el usuario son las siguientes, en orden alfabético:

`MICScFinalize()`:

Libera toda la memoria utilizada por la aplicación y llama a la rutina de finalización de los objetos de PETSc. En el código, debe ser la última llamada.

`MICScInitialize(PetscInt argc, char** argv[])`:

Se encarga de realizar la inicialización de PETSc, así como de leer los ficheros de entrada y reservar e inicializar todos los valores y estructuras de datos comunes a todos los casos. Debe ser la primera llamada del programa y los parámetros de entrada deben de ser punteros a los parámetros de entrada de la función `main`.

`MICScSetProp(St_Prop *prop, St_Patch *patch, ktypeProp propType, PetscReal cons, PetscErrorCode(funct*)(St_Prop *prop), ktypeIp interpType, PetscReal linRelax, PetscTruth calcMeans)`:

Modifica los valores por defecto de una propiedad. El primer parámetro es el puntero a la propiedad a modificar y el resto son la región de la malla sobre la que se aplica (`patch`), el tipo de propiedad (`propType`), el valor de la propiedad si es constante (`cons`), la función de evaluación si es variable (`funct`), el tipo de interpolación (`interpType`) y la relajación lineal (`linRelax`). Además, es posible especificar si se desean calcular medias (ver 5.2.2 para más detalle) para esta propiedad mediante `calcMeans`.

`MICScSolve()`:

Resuelve el sistema de ecuaciones. Es el núcleo principal de MICSc y la llamada que consume prácticamente todo el tiempo de ejecución de la simulación.

`MICScVarSetGamma(PetscInt index, ktypeProp gammaType, ktypeIp gammaInterpType, PetscReal gammaConst, PetscErrorCode (gammaFunct*) (St_Prop *prop))`:

Sustituye la propiedad del coeficiente de difusión por una nueva con los valores especificados como parámetro. El primer parámetro es el índice de la variable; es posible utilizar las macros `U`, `V`, `W`, `P`, etc para este parámetro. Esto es aplicable para todas las funciones de nombre `MICScVarSet*`.

`MICScVarSetInitValues(PetscInt index, ktypeProp initValuesType, ktypeIp initValuesInterpType, PetscReal initValuesConst, PetscErrorCode (initValuesFunct*)(St_Prop *prop))`:

Se define de manera análoga a `MICScVarSetGamma` para los valores iniciales de la variable en cada celda.

`MICScVarSetInterpolation(PetscInt index, ktypeIp interpType)`:

Modifica el tipo de interpolación de la variable.

`MICScVarSetLimits(PetscInt index, PetscReal minValue, PetscReal maxValue)`:

Modifica los valores mínimo y máximo de la variable.

`MICScVarSetReference(PetscInt index, PetscReal resRef, PetscReal corrRef)`:

Modifica los valores de referencia de la variable.

`MICScVarSetRelaxation(PetscInt index, PetscReal linRelax, PetscReal fdtRelax)`:

Modifica los valores de relajación de la variable.

`MICScSetInletFunction(PetscErrorCode (*function)(PetscInt*, PetscReal*))`:

Establece la función que se debe utilizar para calcular el flujo de entrada, en caso de que se defina tal condición de contorno.

Así, un posible flujo básico de la ejecución de un caso con MICSc queda de la siguiente manera:

```
MICScInitialize();

MICScVarSetLimits(U, 20, 20);
...

MICScSolve();

MICScFinalize();
```

5.1.2. Tratamiento dinámico de las variables

Como consecuencia de la refactorización comentada en el apartado anterior, se planteó la posibilidad de que el usuario pudiera definir nuevas variables. Sin embargo, debido a que esto no se hizo necesario para los casos

de prueba que se utilizaron en el proyecto, no se añadieron estas funciones al conjunto de funciones disponibles por el usuario y consta como trabajo futuro. No obstante, sí que es posible ejecutar un cierto caso contemplando la temperatura como variable o no (`-use_t`).

Por otra parte, existen ciertas secciones del código que necesitan conocer el número total de variables con el fin de reservar memoria en función del número de estas variables, tales como diversas condiciones de contorno, propiedades asociadas a las derivadas temporales de las variables o la misma creación del vector que contiene a las variables. En definitiva, es necesario conocer el número total de variables en el momento de la inicialización.

Una primera solución que se contempló pasaba por incluir un parámetro de entrada en el programa (`-nvar`) para que el usuario determinara en el mismo inicio de la ejecución el número final de variables que se iban a necesitar. El problema que planteaba esta solución es que daba lugar a inconsistencias si el usuario determinaba un número de variables por línea de comandos y, posteriormente, el número real era distinto. De esta manera, tal y como se ha dejado intuir en 4.4.1, la solución final consiste en crear las variables como una lista enlazada mediante un puntero a la siguiente variable (`next`). La siguiente sección de código muestra la creación de variables con esta nueva implementación:

```
St_Var pointer, newVariable;

PetscNew(St_Var, &newVariable);
if (firstVar == PETSC_NULL) {
  firstVar = newVariable;
} else {
  pointer = firstVar;
  while (pointer->next != PETSC_NULL) {
    pointer = pointer->next;
  }
  pointer->next = newVariable;
}

newVariable->next = PETSC_NULL;
newVariable->index = num_vars;
...
<establecer atributos>
...

num_vars++;
```

donde `firstVar` es una variable global de tipo `St_Var` que apunta al primer elemento de la lista.

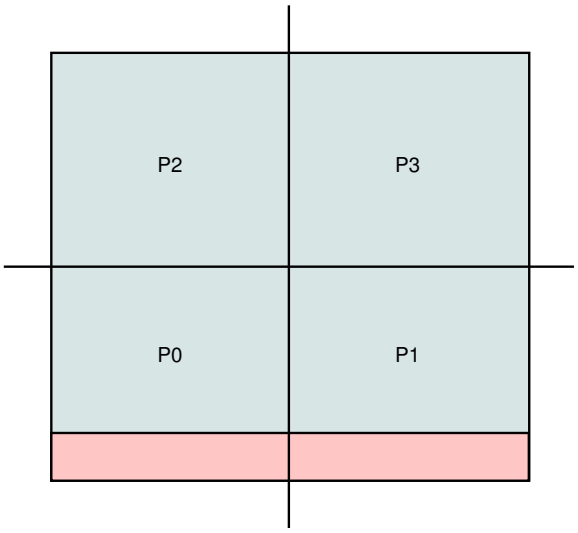

Figura 5.1: Distribución de un *patch*. En color azul el dominio completo, en color salmón el *patch*, dividido entre los procesos 0 y 1.

5.1.3. Refactorización de los *patches*

En la versión inicial de MICSc que se recogió al principio del proyecto, los *patches* se implementaban mediante tres instancias de la estructura `St_MeshRegion`: para la región global del *patch* y la subregión local con y sin nodos *ghost*. Sin embargo, se comprobó que esta implementación proporcionaba exactamente la misma funcionalidad que los objetos `DA` de PETSc. Además, también se daba el hecho de que en el inicio las propiedades se evaluaban de manera local (incluyendo nodos vecinos) y aislada en cada proceso, sin producirse comunicación entre procesos, lo cual podía ser la causa de cierto comportamiento paralelo de divergencia para instantes de tiempo avanzados en la simulación. Por último, y con el fin de simplificar la implementación de muchas secciones del código, se deseaba que la evaluación de las propiedades fuera exclusivamente local (sin nodos vecinos), y que se produjera comunicación entre procesos después de evaluar cada una de las propiedades.

Por todo esto, se decidió sustituir la implementación inicial por una implementación con un objeto `DA` de PETSc. Esta implementación planteaba numerosas dificultades. La primera de ellas es que el *array* distribuido se debía organizar de la misma manera que se organizaba el resto del dominio; es decir, dado un cierto *patch*, éste debía utilizar las mismas divisiones que las usadas en el dominio completo, pudiendo ser posible que la región se dividiera únicamente entre un subconjunto de procesos (ver Figura 5.1). De esta forma, lo primero que se debe hacer es calcular la porción local que corresponde a cada proceso:

```
*r = Local.meshRegion;
IsPatchInProc(x, nx, y, ny, z, nz, &flag);
if (flag) {
  overlap.lowerCorner[XDIR] = MAX(x, r->lowerCorner[XDIR]);
  overlap.lowerCorner[YDIR] = MAX(y, r->lowerCorner[YDIR]);
  overlap.lowerCorner[ZDIR] = MAX(z, r->lowerCorner[ZDIR]);
  overlap.upperCorner[XDIR] = MIN(x + nx,
        r->lowerCorner[XDIR] + r->nCells[XDIR]) - 1;
  overlap.upperCorner[YDIR] = MIN(y + ny,
        r->lowerCorner[YDIR] + r->nCells[YDIR]) - 1;
  overlap.upperCorner[ZDIR] = MIN(z + nz,
        r->lowerCorner[ZDIR] + r->nCells[ZDIR]) - 1;
  overlap.nCells[XDIR] = overlap.upperCorner[XDIR] -
                         overlap.lowerCorner[XDIR] + 1;
  overlap.nCells[YDIR] = overlap.upperCorner[YDIR] -
                         overlap.lowerCorner[YDIR] + 1;
  overlap.nCells[ZDIR] = overlap.upperCorner[ZDIR] -
                         overlap.lowerCorner[ZDIR] + 1;
  overlap.totalCells = overlap.nCells[XDIR] *
      overlap.nCells[YDIR] * overlap.nCells[ZDIR];
} else {
  overlap.totalCells = 0;
}
```

donde `x`, `y`, `z`, `nx`, `ny`, `nz` representan el *patch*. Así, se determina si éste se debe distribuir en el proceso local (`IsPatchInProc`), marcándose el número total de celdas como 0 en caso negativo. En caso de que el *patch* se solape con el subdominio local, se asignan los atributos del objeto `overlap`, de tipo `St_MeshRegion`, que contienen la intersección del subdominio local con el *patch*. Posteriormente, se deben distribuir estas intersecciones a todos los procesos:

```
MPI_Type_struct(4, lengths, displs, types, &meshRegionType);
MPI_Type_commit(&meshRegionType);
MPI_Allgather(&overlap, 1, meshRegionType, overlaps, 1,
              meshRegionType, PETSC_COMM_WORLD);
```

donde `lengths`, `displs` y `types` son, respectivamente, las longitudes, el desplazamiento en bytes y los tipos de los atributos de la estructura de datos `St_MeshRegion`. De esta manera, se crea un nuevo tipo de datos MPI que se corresponde con dicha estructura (`meshRegionType`) y se distribuyen los objetos calculados anteriormente a todos los procesos mediante la operación `MPI_Allgather`, obteniendo en `overlaps` (*array* de `St_MeshRegion` con tantos elementos como procesos) el conjunto de intersecciones en todos los

procesos. El siguiente paso es crear un comunicador MPI para los procesos en los que la intersección calculada no sea nula. Cabe destacar que la comunicación `MPI_Allgather` es necesaria ya que posteriormente se debe llamar a `DACreate` con los mismos argumentos en todos los procesos y, por tanto, los cálculos que siguen se deben realizar en todos los procesos implicados.

```
count = 0;
for (i = 0; i < mpi_size; i++)
  if (overlaps[i].totalCells > 0)
    ranks[count++] = i;
...
MPI_Comm_group(PETSC_COMM_WORLD, &worldGroup);
MPI_Group_incl(worldGroup, count, ranks, &patchGroup);
MPI_Comm_create(PETSC_COMM_WORLD, patchGroup, &patchComm);
```

De esta manera se crea un nuevo grupo conteniendo únicamente los procesos implicados (`ranks` contiene el rango de los procesos con intersección no nula) y un nuevo comunicador para ese grupo, que posteriormente se almacenará en la estructura `St_Patch`. Posteriormente, es necesario calcular cuántas divisiones horizontales y verticales tendrá el *patch* y cuál será el tamaño de éstas para que se correspondan con las intersecciones obtenidas:

```
divs = 0;
for (i = 0; i < mpi_size; i++)
  if (overlaps[i].totalCells > 0) {
    found = PETSC_FALSE;
    for (j = 0; j < i && !found; j++)
      if (overlaps[j].totalCells > 0 &&
        overlaps[i].lowerCorner[dir] == overlaps[j].
                                      lowerCorner[dir]) {
        found = PETSC_TRUE;
      }
    if (!found) sizes[divs++] = overlaps[i].nCells[dir];
  }
```

siendo `dir` el eje considerado en cada caso. Así, repitiéndose este proceso tres veces, para cada dirección se obtiene el número de coordenadas de inicio distintas para todas las intersecciones, junto con el número de celdas a lo largo de dicho eje para cada división distinta. Por último, sólo queda la creación del objeto DA:

```
if (overlaps[Local.rank].totalCells > 0) {
  DACreate3d(patchComm, periodicity, DA_STENCIL_BOX, nx, ny,
          nz, m, n, p, dof, stencil, lx, ly, lz, &da);
} else {
```

```
    da = PETSC_NULL;
  }
```

donde `periodicity`, `dof` y `stencil` son parámetros de la creación para la periodicidad, el número de grados de libertad y el número de nodos *ghost* respectivamente; `m`, `n` y `p` son el número de divisiones obtenido anteriormente en `divs` para cada dirección; y `lx`, `ly` y `lz` son los tamaños de las divisiones obtenidos anteriormente en `sizes` para cada dirección.

De esta manera, se obtiene un `DA` para el *patch* en todos aquellos procesos que deben manejar parte del mismo. Además, en todos aquellos procesos donde la intersección del *patch* con el subdominio local sea nula, el objeto `DA` será nulo y se podrá utilizar esta comprobación para, en cualquier otro punto del código, saber si el *patch* es local o no. Esta comprobación es de vital importancia ya que las posteriores llamadas a las funciones de tipo `DAXXX` deberán realizarse únicamente por aquellos procesos cuyo objeto `DA` no sea nulo.

5.1.4. Refactorización de las propiedades

Como se ha comentado en el apartado anterior, existía un comportamiento divergente en ciertos casos y que daba resultados notablemente diferentes para casos paralelos con respecto a los resultados secuenciales. Puesto que las propiedades se evaluaban de manera local con nodos vecinos y no existía comunicación entre ellas, se pensó que podía ser una causa de dicho comportamiento. Además, puesto que la evaluación en local sin nodos vecinos podía solucionar este problema y facilitar otras implementaciones, se optó por incluir objetos `Vec` en la implementación de la estructura `St_Prop` (en lugar de los *arrays* `double*` iniciales) y utilizarlos, junto con los `DA` explicados en el apartado anterior, para realizar la comunicación. Así, se incluyó el atributo `values` de tipo `Vec` que se crea de la siguiente manera:

```
if (patch->da != PETSC_NULL) {
  ...
  DACreateLocalVector(patch->da, &prop->values);
  VecSetFromOptions(prop->values);
  ...
}
```

siendo `patch` (`St_Patch`) la región de la malla donde se define la propiedad y `prop` (`St_Prop`) la propiedad que se está creando. De esta manera, se puede acceder al vector mediante las llamadas de tipo `VecXXX` o mediante el *array* obtenido con la llamada a `VecGetArray`. Cabe destacar que en el caso de acceder a los elementos del vector mediante *array*, es necesario tener en cuenta que para una propiedad con más de un grado de libertad, los elementos en el vector se ordenan de la manera $(a_0, b_0, c_0, \dots, a_1, b_1, c_1, \dots, a_n, b_n, c_n, \dots)$,

donde la letra indica el grado de libertad y el subíndice, el índice de la celda. De esta manera, para acceder al grado de libertad d de la celda i en una propiedad con n grados de libertad, se debe usar `prop->array[i * n + d]`. El mismo comportamiento es aplicable a los vectores utilizados para el cálculo de medias, aunque en principio el acceso a elementos particulares de estos vectores carece de sentido.

Por último, gracias a la implementación de los *patches* mediante objetos `DA` es posible realizar la evaluación de propiedades únicamente en las celdas locales de cada proceso (sin nodos *ghost*) y realizar posteriormente la comunicación de la siguiente manera:

```
DALocalToLocalBegin(prop->Patch->da, prop->values,
                    INSERT_VALUES, prop->values);
DALocalToLocalEnd(prop->Patch->da, prop->values,
                    INSERT_VALUES, prop->values);
```

que se encarga de actualizar los nodos *ghost* del vector `values` en todos los procesos con los valores correctos, descartando los valores almacenados anteriormente.

5.2. Nuevas funcionalidades

Al margen de desarrollos dedicados a la depuración, optimización y refactorización del código existente, también se han producido desarrollos que dotan a MICSc de nueva funcionalidad. En los siguientes apartados se detallan los más importantes.

5.2.1. Introducción de la viscosidad

Al comienzo del proyecto, la única posibilidad de contemplar la viscosidad propia de fluidos turbulentos (implementada mediante el modelo de Smagorinsky 2.3.2) era introducirla como parte del módulo de usuario, que definía la función de cálculo de la viscosidad turbulenta a partir de la densidad, la viscosidad propia del modelo LES y la viscosidad especificada como parámetro de entrada.

Tras la refactorización de MICSc a la estructura de librería con diferentes casos, la viscosidad se introdujo como una modificación de `Gamma` mediante la llamada `MICScVarSetGamma` donde se le especificaba como parámetro la función de evaluación de la viscosidad. Sin embargo, teniendo en cuenta que entre los principales objetivos del proyecto se encontraba la simulación de flujos turbulentos mediante LES, y que la función de evaluación de la viscosidad es conocida y no varía de un caso a otro, se decidió que la viscosidad debía formar parte del núcleo de la librería para que el usuario pudiera introducirla en su caso de la manera más cómoda posible.

Así, se ha tomado una implementación de la función de la viscosidad total proporcionada por A. Cubero y se ha introducido en la librería de forma que en el momento de la creación de las velocidades, en función de si el usuario especifica un parámetro de entrada (`-les`), es posible determinar si se debe introducir el modelo LES y, en su caso, crear las variables especificando la propiedad `Gamma` con la función de la viscosidad:

```
if (Input.useLES) {
  CreateProp(DomainPatch, VARI, Input.mu, ViscTotal,
      1, CDS, 1, PETSC_FALSE, &gamma);
}

...
CreateVar("U", CDS, -10, 10.0, linRel, fdtRel, resref,
    coref, initValues, gamma, &varU);
```

Al margen de valores de relajación y demás parámetros de estas llamadas, es importante destacar que la llamada a la función `CreateProp` crea la propiedad (`St_Prop`) `gamma`, que se evalúa mediante la función `ViscTotal`. Así, en la creación de la variable (`CreateVar`) se envía como parámetro la propiedad `gamma`, que actúa como coeficiente de difusión para la componente x de la velocidad. La implementación de `ViscTotal` no se incluye en este documento pero sigue el modelo de Smagorinsky y se puede encontrar en el código fuente y la documentación.

Llegados a este punto es importante destacar que la viscosidad afecta a todas las velocidades pero el cálculo de ésta no depende de dichas velocidades. Teniendo en cuenta que en un principio cada una de las propiedades asociadas a las variables eran exclusivas de esa variable, y con el fin de optimizar la aplicación tanto en el uso de memoria como en el cálculo, se ha modificado ligeramente la creación de variables, tal y como se ha mostrado, para que varias variables puedan compartir una misma propiedad de forma que dentro del sistema existirá una única propiedad para la evaluación de la viscosidad que se evaluará una sola vez en cada iteración exterior y se almacenará una sola vez donde sea accesible para todas las variables que la necesiten.

5.2.2. Cálculo de medias

Como se explica de manera detallada en [22], existen una gran variedad de elementos que pueden llevar a la obtención de resultados instantáneos distintos para un mismo problema. Estos factores comprenden desde elementos propios de la computación como el número de procesadores, la precisión de la máquina o los errores de redondeo, hasta factores físicos como las condiciones iniciales o la propia inestabilidad del fenómeno turbulento. Es por esto

que para el análisis de los resultados proporcionados por la simulación, es interesante calcular ciertos valores estadísticos sobre los valores que toman las distintas variables del problema. Estos valores son, tanto las medias de los valores instantáneos, como las medias de las covarianzas de una variable con respecto de todas las demás (incluyendo la propia variable), que no son más que las varianza y covarianzas muestrales de las variables. Cabe destacar que los valores sobre los que se han de calcular las medias son siempre aquellos que se han alcanzado en cada iteración de tiempo, al finalizar la convergencia de las iteraciones exteriores de Picard; es decir, no tiene sentido tener en cuenta los valores intermedios tras resolver los sistemas lineales puesto que estos valores no tienen sentido físico mientras no se alcance la convergencia de la iteración de Picard.

Así pues, para ilustrar los cálculos que se deben realizar, utilizaremos un ejemplo en dos dimensiones donde las medias a calcular son las siguientes:

$\overline{u}$: Media de los valores instantáneos de la velocidad sobre el eje X. Se calcula como

$$\frac{1}{k}\sum_{i=1}^{k} u_i \tag{5.1}$$

donde k es el número de pasos de tiempo realizados y u_i es el valor instantáneo de la componente X de la velocidad en el paso de tiempo t_k.

$\overline{v}$: Media de los valores instantáneos de la velocidad sobre el eje Y. Se calcula de manera análoga a la componente X sustituyendo u_i por v_i (5.1).

$\overline{p}$: Media de los valores instantáneos de la presión. Se calcula de manera análoga a las velocidades sustituyendo u_i por p_i (5.1).

$\overline{u'u'}$: Varianza muestral de la componente X de la velocidad. Se calcula como:

$$\frac{1}{k}\sum_{i=1}^{k}(u_i - \overline{u})^2 \tag{5.2}$$

$\overline{v'v'}$: Varianza muestral de la componente Y de la velocidad. Se calcula de manera análoga a la componente X sustituyendo u_i y $\overline{u}$ por v_i y $\overline{v}$ respectivamente (5.2).

$\overline{p'p'}$: Varianza muestral de la presión. Se calcula de manera análoga a las componentes de la velocidad sustituyendo u_i y $\overline{u}$ por p_i y $\overline{p}$ respectivamente (5.2).

$\overline{u'v'}$: Covarianza muestral de las dos componentes (X e Y) de la velocidad. Se calcula de manera muy similar a la varianza:

$$\frac{1}{k}\sum_{i=1}^{k}(u_i - \overline{u})(v_i - \overline{v}) \tag{5.3}$$

$\overline{u'p'}$: Covarianza muestral de la componente X de la velocidad con la presión. Se calcula sustituyendo v_i y $\overline{v}$ por p_i y $\overline{p}$ en la ecuación anterior (5.3).

$\overline{v'p'}$: Covarianza muestral de la componente Y de la velocidad con la presión. Se calcula sustituyendo u_i y $\overline{u}$ por p_i y $\overline{p}$ en la ecuación anterior (5.3).

Con el fin de calcular las medias, varianzas y covarianzas de las variables, y sabiendo que en cada paso de tiempo el valor de las variables (tras escribirlo a disco en caso necesario) no se mantiene en memoria, es necesario conocer en cualquier momento los sumatorios de los numeradores. Además, puesto que para el cálculo de la varianza y la covarianza, es necesario conocer la media en cada instante de tiempo, las medias se calcularán y se almacenarán en memoria en cada paso de tiempo. Sin embargo, puesto que no en todas las iteraciones temporales se va a escribir a disco y, por tanto, no va a ser necesario conocer todos los valores estadísticos, la división del sumatorio por el número de pasos de tiempo en las ecuaciones de la varianza y covarianza se optimizarán de forma que únicamente se calcularán cuando se vayan a escribir a fichero.

5.2.3. Modelo de van Driest

El modelo de van Driest modifica la constante de Smagorinsky con el fin de amortiguarla en las celdas más cercanas a la pared. Para ello, se mostró en 2.3.2 que es necesario conocer la distancia a la superficie más cercana, junto con la velocidad tangencial a dicha superficie. De esta manera, puesto que este modelo proporciona una clara mejora frente al modelo de Smagorinsky simple, y no plantea ningún problema adicional, se implementó este modelo por defecto siempre que se afronte un caso turbulento con LES (`-les`).

Para esta implementación, en primer lugar es necesario calcular y almacenar la distancia a la pared más cercana (`Geom.wallDistance`), junto con un vector para cada una de las celdas (`Geom.normalVec`). Este vector tendrá en cuenta la dirección desde la propia celda hasta la celda de pared más cercana y calculará un vector unitario normal a dicha dirección, que maximice la componente X y que minimice la componente Y de dicho vector. En la Figura 5.2 se muestra un ejemplo en dos dimensiones. De esta manera, se asume que el fluido siempre fluye de oeste a este, ya que es imposible obtener un vector normal en 3D en la dirección y sentido del fluido si no se

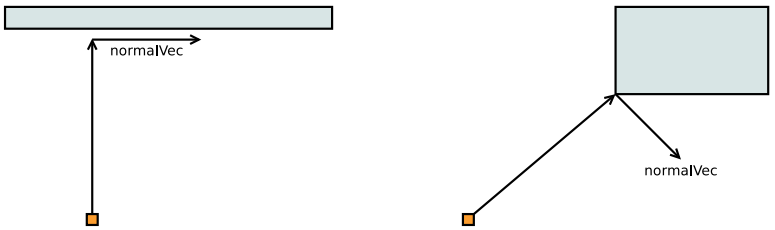

Figura 5.2: Vector normal unitario sin componente Y (izquierda) y con componente Y (derecha) para un dominio de dos dimensiones. Se muestra en gris la pared y en naranja la celda evaluada.

conoce dicha dirección. Este vector se utilizará para calcular posteriormente la velocidad tangencial y aplicar la fórmula del amortiguamiento (2.7).

```
pointer = Global.FirstBcond;
Geom.wallDistance[i] = GREAT;
while (pointer != PETSC_NULL) {
  if (pointer->bCondType == WALL) {
    distance = 0;
    if (pointer->Patch->x > I[XDIR]) {
      vec[XDIR] = Grid.Xnode[XDIR][pointer->Patch->x] -
                  Grid.Xnode[XDIR][I[XDIR]];
      distance += PetscPowScalar(vec[XDIR], 2);
    } else if (pointer->Patch->x +
               pointer->Patch->nx < I[XDIR]) {
      vec[XDIR] = Grid.Xnode[XDIR][I[XDIR]] -
                  Grid.Xnode[XDIR][pointer->Patch->x +
                    pointer->Patch->nx];
      distance += PetscPowScalar(vec[XDIR], 2);
    } else {
      vec[XDIR] = 0;
    }
    ...
    <igual para las direcciones Y, Z>
    ...
    distance = PetscSqrtScalar(distance);
    if (distance < Geom.wallDistance[i]) {
      Geom.wallDistance[i] = distance;
      if (vec[XDIR] == 0) {
        Geom.normalVec[XDIR][i] = 1;
        Geom.normalVec[YDIR][i] = 0;
        Geom.normalVec[ZDIR][i] = 0;
      } else if (vec[ZDIR] == 0) {
        a2 = vec[XDIR] * vec[XDIR];
```

```
          b2 = vec[YDIR] * vec[YDIR];
          aux = a2 + b2;
          Geom.normalVec[XDIR][i] = PetscSqrtScalar(b2 / aux);
          Geom.normalVec[YDIR][i] = PetscSqrtScalar(a2 / aux);
          Geom.normalVec[ZDIR][i] = 0;
        } else {
          a2 = vec[XDIR] * vec[XDIR];
          b2 = vec[ZDIR] * vec[ZDIR];
          aux = a2 + b2;
          Geom.normalVec[XDIR][i] = PetscSqrtScalar(b2 / aux);
          Geom.normalVec[YDIR][i] = 0;
          Geom.normalVec[ZDIR][i] = PetscSqrtScalar(a2 / aux);
        }
      }
    }
    pointer = pointer->Next;
  }
```

El cálculo del vector unitario normal se basa en las dos propiedades que debe cumplir. En primer lugar, el producto escalar del vector dirección (de la celda a la pared, $[a, b, c]$) y el vector normal ($[x, y, z]$) debe ser nulo:

$$ax + by + cz = 0 \rightarrow ax + cz = 0 \rightarrow z = -\frac{ax}{c}. \tag{5.4}$$

En este caso se asume que las componentes X y Z del vector dirección son no nulas y que se puede minimizar la componente Y hasta 0. En el caso de ser nula la componente X, el vector normal es trivial, siendo $[1, 0, 0]$. Por último, en caso de ser nula la componente Z se debería realizar un desarrollo análogo al aquí explicado, despejando y en lugar de z. De esta manera, aseguramos que el vector sea normal. Para que además sea unitario, aplicamos la ecuación del módulo:

$$\begin{aligned} x^2 + z^2 = 1 \Rightarrow x^2 + (-\frac{ax}{c})^2 = 1 \Rightarrow x^2 + \frac{a^2x^2}{c^2} = 1 \Rightarrow \\ \Rightarrow x^2(c^2 + a^2) = c^2 \Rightarrow x = \sqrt{\frac{c^2}{a^2 + c^2}}. \end{aligned} \tag{5.5}$$

Y, despejando en la primera ecuación:

$$z = -\frac{ax}{c} = -\frac{a}{c}\sqrt{\frac{c^2}{a^2 + c^2}} = -\sqrt{\frac{a^2}{a^2 + c^2}}. \tag{5.6}$$

Una vez realizado este cálculo en tiempo de inicialización, es necesario aplicar la fórmula del amortiguamiento en cada celda i, cada vez que cambie la velocidad. El código que aplica dicha fórmula es el siguiente:

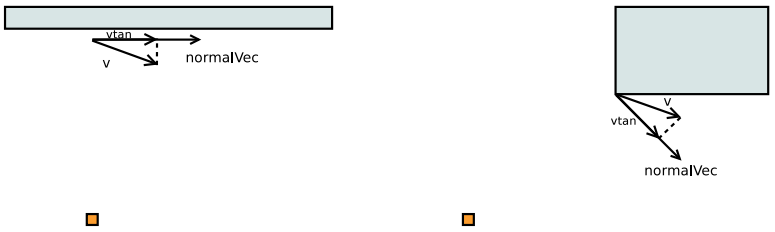

Figura 5.3: Proyección del vector velocidad v sobre un vector normal unitario sin componente Y (izquierda) y con componente Y (derecha) para un dominio de dos dimensiones. Se muestra en gris la pared y en naranja la celda evaluada.

```
vMod = PetscSqrtScalar(PetscPowScalar(VALUE(varU, i), 2)
    + PetscPowScalar(VALUE(varV, i), 2)
    + PetscPowScalar(VALUE(varW, i), 2));
alpha = vMod * PetscSqrtScalar(1 /
     (PetscPowScalar(Geom.normalVec[XDIR][i], 2)
    + PetscPowScalar(Geom.normalVec[YDIR][i], 2)
    + PetscPowScalar(Geom.normalVec[ZDIR][i], 2)));
n[XDIR] = alpha * Geom.normalVec[XDIR][i];
n[YDIR] = alpha * Geom.normalVec[YDIR][i];
n[ZDIR] = alpha * Geom.normalVec[ZDIR][i];

vtan = (n[XDIR] * VALUE(varU, i) + n[YDIR] *
  VALUE(varV, icell) + n[ZDIR] * VALUE(varW, i)) / vMod;
ratio = PetscSqrtScalar((d * VALUE_PROP(Rho, i) * vtan) /
                  (Input.mu * 625));

damping = (ratio < 12) ? 1 - PetscExpScalar(ratio) : 1;
```

En el código se puede observar que en primer lugar se calcula el módulo del vector velocidad. Este módulo se utiliza para escalar el vector normal calculado en la inicialización de manera que se obtenga un nuevo vector normal con el mismo módulo que el vector velocidad. Así, nos aseguramos de que al proyectar este último sobre el vector normal, la proyección siempre sea correcta, tal y como se muestra en la Figura 5.3. Para ello, aplicamos la siguiente ecuación:

$$\sqrt{\alpha^2x^2+\alpha^2y^2+\alpha^2z^2} = \|\vec{v}\|_2 \Rightarrow \alpha^2(x^2+y^2+z^2) = \|\vec{v}\|_2^2 \Rightarrow$$
$$\Rightarrow \alpha = \frac{\|\vec{v}\|_2^2}{x^2+y^2+z^2} \Rightarrow \alpha = \|\vec{v}\|_2\sqrt{\frac{1}{x^2+y^2+z^2}}, \tag{5.7}$$

donde x, y y z son las componentes del vector normal y $\vec{v}$ es el vector velocidad. Así, obtenemos el nuevo vector normal en `n`. Por último, aplicando

el producto escalar para obtener el módulo de la proyección, se obtiene que:

$$Proy_{\vec{n}}\vec{v} = \frac{\vec{v}^T\vec{n}}{\|\vec{v}\|_2}, \tag{5.8}$$

que es el módulo de la velocidad tangencial necesario para acabar de aplicar la fórmula del amortiguamiento de van Driest (2.9) en las siguientes líneas.

5.2.4. Documentación

Al margen de funcionalidades que tienen como finalidad la inclusión de cálculos necesarios para el análisis de resultados o nuevos modelos, se han realizado desarrollos dirigidos a la usabilidad y mejor entendimiento del código utilizado. Concretamente, se ha incluido el sistema de documentación utilizado por PETSc en MICSc.

Este sistema se basa en herramientas como *Sowing* [23] y *C2HTML* [24] (al margen de herramientas estándar como *pdflatex* o *python*). Así, incluyendo los ficheros de reglas y variables proporcionados en la instalación de PETSc, es relativamente sencillo introducir nuevas reglas en el fichero *makefile* de MICSc que hagan uso de las herramientas de PETSc para generar documentación. Las siguientes reglas implementan la funcionalidad para generar la documentación:

```
alldoc: alldoc1 alldoc2

alldoc1:
  -${OMAKE} ACTION=manualpages_buildcite tree_basic LOC=${LOC}
  -${OMAKE} ACTION=manualpages tree_basic LOC=${LOC}
  -${PETSC_DIR}/bin/maint/wwwindex.py ${MICSC_DIR} ${LOC}

alldoc2:
  -${OMAKE} ACTION=micsc_html PETSC_DIR=${PETSC_DIR}\\
            alltree LOC=${LOC}
```

Antes de nada, es importante notar que PETSc proporciona las reglas `tree_basic` y `alltree` que permiten llamar a una determinada regla (especificada en la variable `ACTION`) de manera recursiva a partir del directorio especificado en `LOC`. Además, la variable `$LOC` se especifica al ejecutar la regla de la siguiente manera sobre el directorio raíz de MICSc:

```
make alldoc LOC=$PWD
```

Por otro lado, se observa que la generación de documentación se divide en dos fases (`alldoc1` y `alldoc2`). La primera, mediante la regla `manualpages_buildcite`, que hace uso internamente de *Sowing*, genera un fichero HTML "crudo" (únicamente texto con formato) por cada uno de los

elementos (funciones, estructuras de datos, enumeraciones) a documentar que contiene la información pertinente, junto con un fichero que contiene las referencias entre ellos. Posteriormente, se vuelven a generar los ficheros, esta vez con todas las referencias cruzadas satisfechas (`manualpages`). Finalmente, mediante el *script* de Python `wwwindex.py` se generan páginas HTML adicionales para la navegación entre los distintos documentos generados.

En la segunda etapa de la documentación (`alldoc2`) se genera una página HTML para cada uno de los ficheros fuente mediante *C2HTML* (`micsc_html`, que por simplicidad no se comenta su implementación). Estos documentos tiene enlaces con toda la documentación, números de línea, y diversas simplificaciones con el fin de hacer la navegación por el código más cómoda.

También cabe destacar que para generar esta documentación es necesario disponer de una versión de PETSc de repositorio, puesto que las versiones de distribución carecen de las herramientas necesarias, tales como `wwwindex.py`. Por último, a modo de ejemplo se muestra la sección de código introducida en el código fuente que genera la documentación de la función `MICScVarSetLimits` (para más información sobre el formato de la documentación, ver [23]):

```
/*@
  MICScVarSetLimits - Sets the limits of the specified
                      variable.

  Input parameters:
+ index - Index of the variable to set.
- minValue, maxValue - The limit values of the variable.

  Level: intermediate

.keywords variable, limits
@*/
```

5.2.5. Salida de resultados

Como se ha comentado en 4.2.2, SILO permite la escritura de ficheros HDF5 de manera cómoda y transparente para aplicaciones científicas. Este es uno de los desarrollos realizados en este proyecto, ya que al inicio del mismo, la única opción era la escritura de ficheros Tecplot (ASCII) de manera secuencial.

Una característica importante de SILO es el hecho de que, en cierto modo, permite la escritura a disco en paralelo. Concretamente, SILO es una librería secuencial pero que ejecutada en paralelo, permite que cada uno de los procesos escriba sus datos en un fichero independiente y, posteriormente, uno de ellos (generalmente el proceso de rango 0) escriba un fichero nuevo

que defina las conexiones entre los distintos ficheros.

Por otra parte, la funcionalidad proporcionada por MICSc para el soporte de ficheros SILO se basa en la implementación de la escritura de distintos elementos: la malla, las variables y las propiedades, todo ello en el contexto de las mallas estructuradas rectangulares. Para el caso de la malla, el siguiente código implementa la funcionalidad deseada:

```
...
outputLocal = DBCreate(localFile, DB_CLOBBER, DB_LOCAL,
    PETSC_NULL, DB_HDF5);
DBPutQuadmesh(outputLocal, "/Mesh", PESTC_NULL, coords,
  Local.meshRegion.nCells, 3, DB_DOUBLE, DB_COLLINEAR, opt);
DBClose(outputLocal);

if (Local.rank == 0) {
  ...
  for (i = 0; i < mpiSize; i++) {
    PetscSNPrintf(meshNames[i], 1024, "output_%d.silo:%s",
      i, "/Mesh");
    meshTypes[i] = DB_QUADMESH;
  }
  outputGlobal = DBCreate("output.silo", DB_CLOBBER,
    DB_LOCAL, PETSC_NULL, DB_HDF5);
  DBPutMultimesh(outputGlobal, MESH_PATH, mpiSize,
    meshNames, meshTypes, PETSC_NULL);
  DBClose(outputGlobal);
  ...
}
...
```

Por simplicidad se obvian la reserva y liberación de memoria. `localFile` contiene el nombre del fichero local de la forma `output_<rango>.silo`; `coords` contiene las coordenadas de la porción local de la malla en cada proceso; `opt` es una lista de posibles opciones, argumentos o metadatos para la escritura; `mpiSize` es el número de procesos; `meshNames` es un vector de cadenas que contiene la ruta completa de las distintas regiones de la malla (`<nombre_fichero>:<ruta>`) y `meshTypes` contiene los tipos de malla (en todos los casos, malla estructurada rectangular).

Se puede observar cómo primero se crea el fichero local (`DBCreate`), se escribe la región local de la malla (`DBPutQuadMesh`) y, finalmente, se cierra el fichero (`DBClose`). Posteriormente, el proceso de rango 0 crea un nuevo fichero donde define una (multi-)malla a partir de las diferentes porciones anteriormente escritas (`DBPutMultimesh`).

Por otro lado se encuentra la escritura de variables, que generalmente, se realiza cada cierto número de pasos temporales:

```
output = DBOpen(localFile, DB_HDF5, DB_APPEND);
...
var = firstVar;
...
while (var != PETSC_NULL) {
  WriteMultivar(output, var->name, 1, 0, &var->Array);
  var = var->next;
}
if (instantMeans != PETSC_NULL) {
  CreateAndEnterDir(output, meansPath);
  var = firstVar;
  for (i = 0; i < Input.nvar; i++) {
    PetscSNPrintf(name, 1024, "Mean_%s_", var->name);
    VecGetArray(instantMeans[i], &array);
    WriteMultivar(output, name, 1, 0, &array);
    var = var->next;
  }
  ...
}
DBClose(output);
```

En este caso se obvian tanto la reserva y liberación de memoria, como el caso de la escritura de las medias de las varianzas, que se implementan de manera prácticamente idéntica a las medias de los valores instantáneos. `var` es una variable de tipo `St_Var`; `meansPath` es una cadena con la ruta donde se deben escribir los resultados de las medias dentro del fichero SILO; `name` es una cadena y `array` es un vector de valores reales; además, `CreateAndEnterDir` es una función que equivale a `mkdir <dir>; cd <dir>` en un sistema UNIX.

Se puede observar como tanto la escritura de variables como de medias descansa sobre la función `WriteMultivar`. Esta función recibe una referencia al fichero SILO donde se debe escribir la variable, el nombre, los grados de libertad, una lista de cadenas para los nombres de las subvariables (si el grado de libertad es mayor que uno) y la lista de los valores de las variables. Es importante notar que todos los procesos llaman a esta función, cada uno con los valores locales de la variable. La implementación de `WriteMultivar` es la siguiente:

```
WriteMultivar(DBfile *output, const char *varName,
    PetscInt numSubVars, char **subVarNames,
    PetscReal **values) {
  ...

  if (numSubVars == 1) {
```

```
    DBPutQuadvar1(output, varName, "/Mesh", values[0],
        Local.meshRegion.nCells, 3, PETSC_NULL, 0, DB_DOUBLE,
        DB_NODECENT, PETSC_NULL);
  } else {
    DBPutQuadvar(output, varName, "/Mesh", numSubVars,
        subVarNames, values, Local.meshRegion.nCells, 3,
        PETSC_NULL, 0, DB_DOUBLE, DB_NODECENT, PETSC_NULL);
  }

  if (Local.rank == 0) {
    ...
    global = DBOpen("output.silo", DB_HDF5, DB_APPEND);
    CreateAndEnterDir(global, path);
    for (i = 0; i < mpiSize; i++) {
      PetscSNPrintf(varNames[i], 1024, "output_%d.silo"
        ":%s/%s", i, path, varName);
      varTypes[i] = DB_QUADVAR;
    }

    DBPutMultivar(global, varName, mpiSize, varNames,
        varTypes, PETSC_NULL);
    DBClose(global);
    ...
  }
  ...
}
```

Como se puede observar, el funcionamiento es completamente análogo a la implementación de la escritura de la malla. Las diferencias radican en las funciones para escribir las variables (`DBPutQuadVar` y `DBPutMultivar`) y en la consideración de que una variable pueda estar formada por distintas subvariables (`numSubVars` > 1), como es el caso de la velocidad que, por simplicidad, no se ha detallado al completo.

Por último, destacar que el caso de la escritura de propiedades se basa también en llamadas a la función `WriteMultivar` y es prácticamente idéntico al caso de la escritura de variables, por lo que no se incluye su implementación en este documento.

5.2.6. *Configure / Makefile*

La posibilidad de escribir los resultados en formato SILO implica que se debe tener instalado tanto SILO con soporte para HDF5 y se debe de poder enlazar MICSc con estas librerías en tiempo de compilación con el fin de dar la funcionalidad requerida. Sin embargo, existen ciertos casos en los que esta

funcionalidad puede no ser posible o incluso contraproducente. Por ejemplo, es posible que se esté intentando ejecutar en una máquina donde no se tengan permisos de administrador o no se tenga acceso a Internet para obtener SILO y HDF5; o bien simplemente se quiere realizar una ejecución para comprobar la corrección del código o medir tiempos de cara a calcular aceleraciones y eficiencia como los presentados en 6.2.2. En estos casos resulta más adecuado poder compilar MICSc sin soporte para SILO.

Así, con el fin de poder realizar distintas configuraciones de MICSc, y adecuándose a las convenciones GNU [25], se generó un *script* de configuración `configure` y un fichero `makefile` con la regla `make` para la compilación de MICSc.

El *script* `configure` se encuentra escrito en Python y para su implementación, al igual que para la implementación de ciertas secciones del fichero `makefile`, se utilizaron como documentación y ejemplo las implementaciones de SLEPc [26], un proyecto llevado a cabo por compañeros del mismo grupo de investigación (GRyCAP [27]). Así pues, únicamente comentaremos las opciones que proporciona este *script*:

`--with-silo`: Configura la instalación buscando las librerías y cabeceras de SILO en los directorios por defecto (i.e. `/usr/local`, `/opt`, ...).

`--with-silo-dir=<silo_dir>`: Configura la instalación de MICSc buscando las librerías de SILO en `<silo_dir>/lib` y las cabeceras en `<silo_dir>/include`.

`--with-silo-flags=<silo_flags>`: Configura la instalación especificando exactamente los parámetros que se deben utilizar en el comando de compilación (por ejemplo, `-lsiloh5 -lhdf5 -L<dir> -I<dir>` ...).

Cabe destacar que en el caso de especificar el directorio de SILO, las librerías de HDF5 deberán estar en los directorios por defecto. En caso de no estar en dichos directorio, la configuración fallará. Esto tiene como solución incluir los argumentos de la compilación mediante `--with-silo-flags`. Sin embargo, aunque efectiva, esta solución es algo incómoda y admite ciertas mejoras menores que se contemplan en la hoja de ruta del proyecto pero que no merecen una mención especial en la sección de trabajo futuro (7.2).

Una vez se ha realizado la configuración, se debe definir la variable de entorno `MICSC_DIR` (si no se ha definido antes) que contenga la ruta raíz de MICSc. Además, se hará uso de la variable de entorno `PETSC_ARCH` para crear un nuevo directorio dentro de `$MICSC_DIR` que contendrá las librerías, cabeceras y ficheros de configuración específicos. Posteriormente, mediante la orden `make` se compilará la librería. Llegados a este punto, es posible compilar todos los casos proporcionados con MICSc mediante la regla `make examples`.

Por último, el fichero `makefile` también proporciona diversas reglas independientes que nos permiten, entre otras cosas, crear un conversor de SILO a Tecplot, de binario a Tecplot o una aplicación basada en PETSc para la comparación de normas de vectores en distintos ficheros binarios. La implementación de esto se basa principalmente en las prácticas que utiliza PETSc para su configuración y compilación y se puede encontrar en la documentación y el código fuente.

De esta manera, y teniendo en cuenta que la única herramienta de compilación al principio del proyecto era un fichero `makefile` para la compilación de todo el proyecto (con los problemas planteados en 5.1.1) con una sola regla `main`, estas implementaciones también constituyen uno de los avances en usabilidad más importantes de este proyecto.

Capítulo 6

Experimentos y resultados

En este capítulo se detallan los experimentos realizados con el código para su presentación en el congreso ECT2010 celebrado en Valencia. En ellos se analizan tanto los resultados de cara a validarlos frente a otros resultados conocidos, como las medidas temporales con el fin de obtener una estimación del comportamiento y eficiencia paralelos. Cabe destacar que existen otros casos de validación realizados anteriormente para el código en el contexto de la tesis doctoral de A. Cubero que pueden encontrarse en [14].

6.1. Validación LES: caso del canal periódico

Para la validación del comportamiento del código en la resolución de flujos turbulentos mediante LES se utilizó el caso del canal periódico. Este caso consiste en un flujo turbulento a través de un canal caracterizado por tener dos condiciones de contorno de tipo pared en las regiones norte y sur; es decir, a lo largo del plano XZ para los valores mínimo y máximo de Y en el dominio. Además, se caracteriza por la periodicidad tanto en el eje X como en el eje Y. En nuestro caso concreto, la malla utilizada consta de $121 \times 121 \times 81$ nodos a lo largo de las direcciones x, y, z respectivamente. Por otro lado, cabe destacar que se ha utilizado una malla con diferente tamaño en los volúmenes finitos a lo largo de la dirección normal al flujo (eje Y), siguiendo una determinada función hiperbólica. El resultado es una malla representando el dominio [0, 0, 0] - [6.45, 2, 3.24] como se muestra en la Figura 6.1. Para la validación, en primer lugar se realizó la ejecución del caso con la implementación simple del modelo de Smagorinsky para la turbulencia, como se detalla en 2.3.2. Además, con el fin de contrastar y validar los resultados obtenidos, estos se compararon con dos simulaciones distintas: una realizada por Moser [28] siguiendo la aproximación DNS y otra realizada por Piomelli mediante LES. Por último, con el fin de obtener un flujo con el número de Reynolds adecuado y poder comparar el flujo frente a estas otras simulaciones, se fijó un diferencial de presión $\frac{dp}{dx}$ cons-

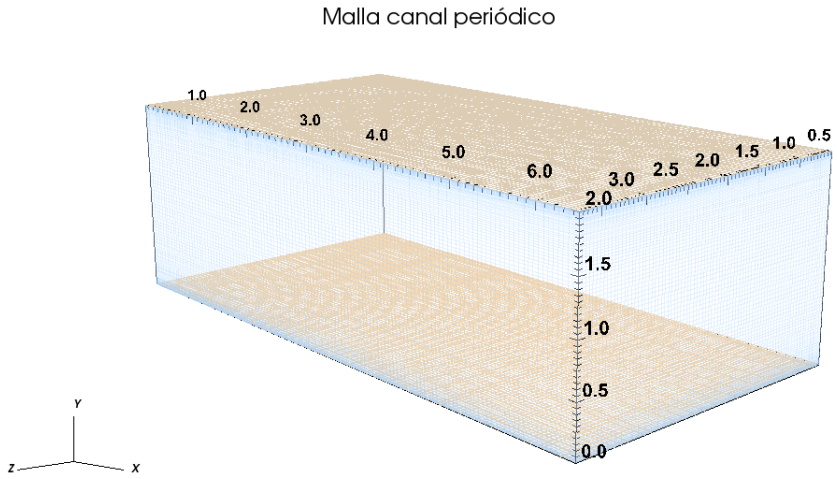

Figura 6.1: Malla 121×121×81 para el caso del canal periódico.

tante e igual a 1. Los resultados obtenidos se muestran en la Figura 6.2. En estos resultados se puede observar como los perfiles de variables promediadas salen cualitativamente bien, en el sentido de que las curvas son suaves, simétricas y con los picos localizados en las regiones donde se espera. Sin embargo, las comparaciones con los resultados de Moser y Piomelli no obtienen el resultado deseado, debido a las grandes diferencias en magnitud entre simulaciones. En este momento se pensó que las diferencias eran debidas al modelo LES utilizado, y que la implementación del modelo de van Driest 2.3.2 que amortigua el efecto de la turbulencia en las celdas cercanas a las paredes podría dar mayor precisión a los resultados. Así pues, tras realizar la implementación del modelo de van Driest, y siendo ya presentado en el congreso ECT2010 celebrado en Valencia [17], se obtuvieron los resultados que se muestran en la Figura 6.3. En estos resultados se puede observar como la tendencia de los resultados, comparados esta vez únicamente con el DNS de Moser, sigue siendo satisfactoria, pero además la magnitud de los resultados se acerca hasta alcanzar unos valores aceptables.

Sin embargo, los resultados de la simulación muestran como el flujo se desarrolla hasta alcanzar el estado estadísticamente estacionario con un número de Reynolds de 224, mientras que los resultados de Moser se aplican a un flujo con un número de Reynolds de 180. En la práctica, esto resulta en que se está validando una simulación frente a otra muy similar pero no idéntica. De esta manera, las diferencias menores en los resultados pueden corregirse (o por el contrario, acrecentarse) simulando con el mismo número

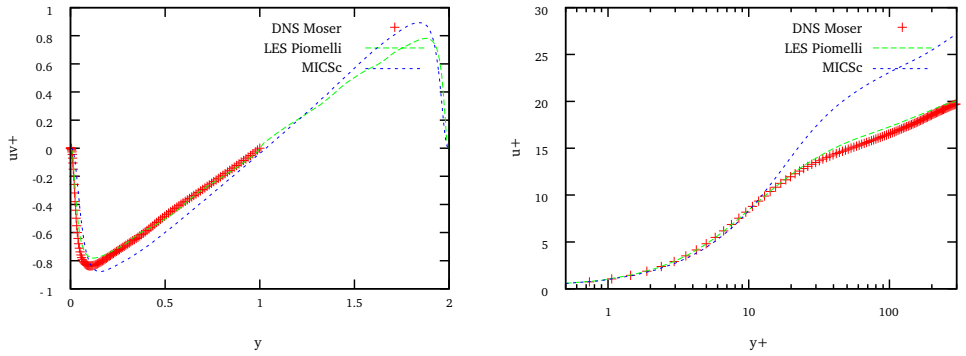

Figura 6.2: Resultados para la simulación del caso del canal periódico con el modelo de Smagorinsky.

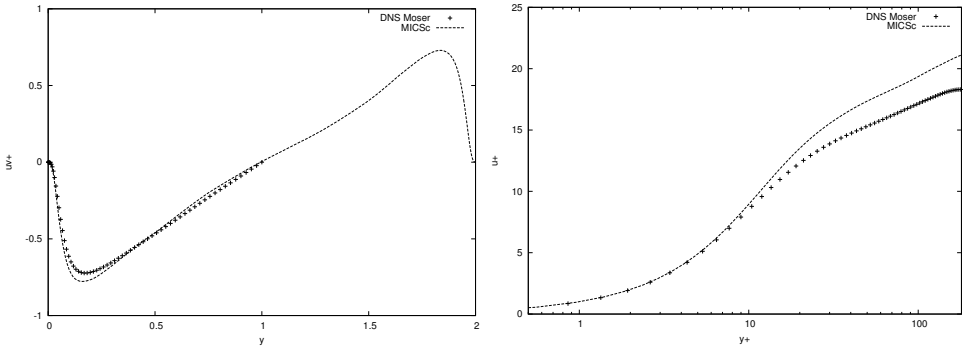

Figura 6.3: Resultados para la simulación del caso del canal periódico con el modelo de van Driest.

de Reynolds. Así, para obtener el número de Reynolds deseado, el diferencial de presión fijado debería de ajustarse en cada paso de tiempo para que de esta manera no tuviera que variar la velocidad media del fluido, y así mantener el número de Reynolds constante e igual a 180.

6.2. Rendimiento del código paralelo

Para cuantificar el rendimiento del código en el caso del canal periódico, explicado en la sección anterior, en esta sección se presentan ciertas medidas temporales, principalmente para la discusión de dicho rendimiento en términos de eficiencia paralela.

En cuanto a la configuración utilizada para las simulaciones, se realizaron 30 pasos temporales con un total de 360 resoluciones de sistemas lineales en el total de cada una de las pruebas. Las pruebas se realizaron en

KSP	PC	Setup	Solve	Total	Its
gmres	jacobi	0.019	0.692	0.711	12.84
gmres	bjacobi	0.129	0.322	0.451	3.83
bcgs	jacobi	0.019	0.570	0.589	7.25
bcgs	bjacobi	0.127	0.332	0.459	2
bcgsl	jacobi	0.020	0.812	0.832	9.33
bcgsl	bjacobi	0.129	0.556	0.685	3.67
tfqmr	jacobi	0.019	1.052	1.071	11.86
tfqmr	bjacobi	0.130	0.353	0.483	2.03

Tabla 6.1: Comparación de *solvers* lineales (KSP) y precondicionadores (PC).

CaesarAugusta, un supercomputador con 256 nodos JS20, cada uno de ellos con dos procesadores PowerPC 970FX de 64 bits funcionando a 2.2 GHz, interconectados con una red Myrinet de baja latencia.

6.2.1. Comparación de *solvers* lineales y precondicionadores

La primera prueba se centra en la comparación en cuanto a tiempo y número de iteraciones en la resolución de los sistemas lineales que componen el problema. Concretamente, se considera el tiempo medio de *setup*, de resolución y el total (suma de ambos), todo ello en valores medios por resolución de sistema lineal. Por otro lado, el número de iteraciones se muestra de la misma manera, promediado por cada resolución de sistema lineal, que como se ha comentado anteriormente suman un total de 360 resoluciones.

En cuanto a los *solvers* lineales utilizados para la comparación, estos son GMRES (*gmres*), BiCGStab (*bcgs*) y BiCGStab(ℓ) con $\ell = 2$ (*bcgsl*), mientras que los precondicionadores utilizados son Jacobi (*jacobi*) y Jacobi por bloques o Block Jacobi (*bjacobi*). En el caso de Jacobi por bloques se utilizan p bloques, siendo p el número de procesadores utilizados en la simulación y utilizando una factorización incompleta LU o ILU por cada uno de los bloques. En el caso concreto de esta prueba, el número de procesadores utilizados para todas las ejecuciones fue de 32. Se pueden encontrar más detalles sobre estos métodos y precondicionadores en [29]. Los resultados obtenidos se muestran en la Tabla 6.1. En esta tabla se puede observar en primer lugar que el uso de Block Jacobi reduce de manera notable el número de iteraciones necesarias para alcanzar la convergencia, mientras que por otro lado aumenta sensiblemente el tiempo de *setup*. Esto es debido a que, por un lado el precondicionador de Jacobi se obtiene de forma inmediata a partir de la diagonal de la matriz A del sistema $Ax = b$, mientras que el precondicionador por bloques requiere de la computación de la factorización LU para cada uno de los bloques diagonales que, aunque incompleta, obviamente requiere un mayor tiempo de computación.

Por otro lado, el tiempo por iteración con el uso de Jacobi por bloques es mayor que para Jacobi simple, pero siendo el número de iteraciones menor, en la medición del tiempo global, el resultado es siempre menor con el uso de Jacobi por bloques.

Por último, en cuanto a la comparación de *solvers* lineales, los mejores resultados los presentan tanto GMRES como BiCGStab, con prácticamente los mismos resultados. Es por esto que para la evaluación de aceleración y eficiencia del siguiente apartado se consideran las combinaciones GMRES + Block Jacobi y BiCGStab + Block Jacobi, junto con la combinación que existía por defecto cuando se adoptó el código, GMRES + Jacobi simple.

6.2.2. Evaluación de aceleración y eficiencia paralela

Como se comenta en el apartado anterior, en esta prueba se realizó la ejecución del caso del canal periódico con distintas configuraciones de *solvers* lineales y precondicionadores. Concretamente GMRES + Block Jacobi, BiCGStab + Block Jacobi y GMRES + Jacobi simple. Para cada una de estas pruebas se realizaron varias pruebas con distinto número de procesadores p, en concreto $p = \{1, 2, 4, 8, 16, 32, 64\}$. Además, para cada ejecución se muestran dos secciones: en la primera de ellas se muestra el tiempo total de la ejecución junto con la aceleración (S_p) y eficiencia (E_p), mientras que en la segunda se considera únicamente el tiempo medio de resolución de sistemas lineales (dividido en *setup* y resolución), también acompañado de las correspondientes aceleración (S_p) y eficiencia (E_p). Los resultados se muestran en la Tabla 6.2.

De esta tabla se puede extraer una conclusión global a partir de los resultados de eficiencia para los tiempos globales, que incluyen tanto la resolución de sistemas como la evaluación previa de propiedades y el ensamblado del sistema. Estos resultados, muestran una tendencia a descender conforme se aumenta el número de procesadores, tal y como se esperaba; pero por otro lado, el valor de la eficiencia se muestra por encima del 60 % para todos los casos, incluso para el caso de 64 procesadores. Es por ello que los valores de aceleración y eficiencia para el tiempo total se pueden considerar razonablemente satisfactorios.

Sin embargo, para el caso de los *solvers* lineales, los resultados son ligeramente peores. En concreto, BiCGStab muestra una eficiencia muy a la par con la eficiencia global de la ejecución, mientras que el peor resultado se muestra para la combinación de GMRES + Jacobi, estando siempre la aceleración y eficiencia siempre por debajo de los resultados del tiempo total, muy posiblemente debido al mayor número de iteraciones efectuadas.

Por último, se puede observar que el comportamiento del precondicionador de Block Jacobi es satisfactorio, a pesar de que la efectividad disminuye conforme aumenta el número de procesadores, ya que el tamaño del bloque es cada vez menor.

GMRES + Jacobi

P	TOTAL			MEDIAS KSP				
	Solve	S_p	E_p	**Setup**	**Solve**	**Total**	S_p	E_p
1	15309.17	-	-	0.37	12.39	12.77	-	-
2	9926.51	1.5	77.1 %	0.29	9.65	9.94	1.3	64.2 %
4	5391.1	2.8	71 %	0.14	4.95	5.09	2.5	62.7 %
8	2791.4	5.5	68.6 %	0.073	2.54	2.62	4.9	61 %
16	1397.5	11	68.5 %	0.038	1.37	1.41	9	56.5 %
32	742.55	20.6	64.4 %	0.019	0.69	0.71	18	56.1 %
64	386.87	39.6	61.8 %	0.0099	0.35	0.36	35.3	55.1 %

GMRES + Block Jacobi

P	TOTAL			MEDIAS KSP				
	Solve	S_p	E_p	**Setup**	**Solve**	**Total**	S_p	E_p
1	14777.78	-	-	3.79	5.81	9.59	-	-
2	8703.70	1.7	84.9 %	2.14	3.94	6.1	1.6	79 %
4	4669.2	3.2	79.1 %	1	1.93	2.93	3.3	81.8 %
8	2387.5	6.2	77.4 %	0.54	0.96	1.5	6.4	79.9 %
16	1151.7	12.8	80.2 %	0.26	0.52	0.78	12.3	76.6 %
32	631.86	23.4	73.1 %	0.13	0.32	0.45	21.3	66.4 %
64	351.56	42	65.7 %	0.064	0.17	0.23	41.2	64.4 %

BiCGStab + Block Jacobi

P	TOTAL			MEDIAS KSP				
	Solve	S_p	E_p	**Setup**	**Solve**	**Total**	S_p	E_p
1	15312.97	-	-	3.79	7.06	10.86	-	-
2	8860.89	1.7	86.4 %	2.10	4.80	6.91	1.6	78.6 %
4	4793.98	3.2	79.9 %	1.04	2.4	3.44	3.2	79 %
8	2559.25	6	74.8 %	0.53	1.32	1.85	5.9	73.5 %
16	1186.53	12.9	80.7 %	0.25	0.64	0.89	12.2	76.4 %
32	639	24	74.9 %	0.13	0.33	0.46	23.7	73.9 %
64	354.49	43.2	67.5 %	0.065	0.18	0.24	44.9	70.1 %

Tabla 6.2: Rendimiento paralelo para tres combinaciones diferentes de *solver* lineal y precondicionador.

Capítulo 7

Conclusiones y trabajo futuro

En este capítulo se presentan las principales conclusiones a las que se ha llegado tras la realización del proyecto final de carrera, así como los trabajos más relevantes que se plantean en un futuro para el proyecto RHELES.

7.1. Conclusiones

En general, se considera que los objetivos planteados al inicio del proyecto final de carrera se han cumplido mayoritariamente. La tarea de optimización y refactorización del código es satisfactoria, ya que ha mantenido la corrección de los resultados y al mismo tiempo ha facilitado la implementación de nuevas funcionalidades.

Por otra parte, la implementación de nuevos modelos ha sido algo menor que lo esperado en un principio, reduciéndose básicamente a la implementación del modelo de van Driest para la turbulencia. Sin embargo, cabe destacar que esta implementación se ha comportado correctamente, tal y como muestran los resultados del capítulo 6.

En lo que respecta a la medida de prestaciones, las realizadas sobre el caso de estudio del canal periódico se han considerado satisfactorias, aunque bien es cierto que admiten optimizaciones aun por estudiar.

Además, la generación de documentación se ha conseguido que adopte el formato de PETSc y que se genere de manera automática. De este modo, a pesar de no estar completa para todas las estructuras datos y funciones, cubren la mayor parte del código.

También es importante destacar que la estructuración del código como una librería con casos, junto con la inclusión de los elementos de configuración y compilación *configure* y *makefile*, y la implementación de la funcionalidad de entrada/salida han mejorado la portabilidad del código, dejándolo listo para su uso en producción con casos de interés.

Por último, cabe destacar que en cuanto a publicaciones, el trabajo realizado se ha publicado en el contexto del congreso ECT2010 celebrado en Valencia [17].

7.2. Trabajo futuro

Como se pudo intuir en los objetivos del proyecto RHELES, dicho proyecto es ambicioso y se pretende que tenga una cierta continuidad de cara a la aplicación del código para la simulación de situaciones interesantes desde el punto de vista de la ingeniería rural e hidráulica, tales como la dispersión de contaminantes o transporte de sedimentos. Además, conforme se ha ido avanzando el proyecto, y tal y como se ha podido intuir en ciertas secciones de este documento, se han detectado tareas necesarias para la mejora y desarrollo de MICSc. En concreto, a la finalización del proyecto final de carrera, las tareas consideradas para el futuro eran las siguientes:

Implementación del modelo LES dinámico: Al margen del modelo de Smagorinsky simple y la posterior inclusión de la amortiguación de van Driest, es interesante la implementación del modelo dinámico (ver 2.3.2) con el fin de obtener resultados con distintos modelos y realizar comparaciones entre ellos.

Implementación de modelos físicos: Este trabajo consiste en la inclusión de nuevas ecuaciones para el transporte de sedimentos de cara al análisis de los resultados y su comparación con los resultados obtenidos por los miembros del grupo de investigación mediante otros códigos de simulación.

Implementación del modelo Kϵ: El modelo Kϵ es un modelo para representar la turbulencia basado en la aproximación RANS (2.3.3). El objetivo principal de este desarrollo consiste en la comparación de resultados mediante diferentes aproximaciones (LES y RANS).

Implementación del "Immersed Boundary Method": Se trata de simular la presencia de estructuras elásticas o membranas dentro del fluido. Este método tiene una gran aplicación en el contexto de la ingeniería rural ya que permite simular situaciones tales como vegetación en el meandro de un río.

Ficheros de entrada: Como se ha comentado en 4.2.1, el formato del fichero que contiene la malla es un formato propio. Una de las mejoras que se plantean es adoptar la plataforma SALOMÉ [30]. Esta plataforma proporciona una herramienta CAD útil como mallador y un formato de fichero (MED) junto con una librería para acceder a este tipo de ficheros. Es importante destacar que para este desarrollo también

se consideró la opción de utilizar OpenFOAM, pero se descartó debido por un lado al alto acoplamiento que existía entre el mallador y el resto de componentes del paquete; y, por otro lado, al hecho de estar desarrollado en C++, lo cual no se consideró adecuado por el grupo de investigación, ya que todo MICSc está desarrollado en C.

Nuevos métodos iterativos: Se valora como una posibilidad interesante la introducción de nuevos métodos iterativos propuestos recientemente, tales como IDR(s) o GBiCGStab(s,L), y la evaluación de su funcionamiento.

Precondicionamiento: Consiste en el empleo de técnicas de precondicionamiento basadas en la física, las cuales tratan de forma diferenciada las variables de velocidad y presión, y aplican técnicas matriciales a bloques como el complemento de Schur.

Apéndice A

Manual de usuario de MICSc

Este manual contiene las instrucciones básicas para la instalación, ejecución y creación de nuevos casos de uso, así como una descripción de la estructura jerárquica de los ficheros de salida producidos en formato SILO.

Instalación y configuración

Para instalar MICSc, una vez se ha obtenido el paquete, se debe descomprimir y entrar en el directorio creado.

```
$ tar xvzf micsc.tar.gz
$ cd micsc
```

Posteriormente, para poder realizar la configuración del sistema, el requisito mínimo es disponer de una instalación de PETSc y definir las variables de entorno `PETSC_DIR` y `PETSC_ARCH` si no están ya definidas.

```
$ PETSC_DIR=<directorio_petsc>; export PETSC_DIR
$ PETSC_ARCH=<arquitectura_petsc>; export PETSC_ARCH
```

En este punto, MICSc proporciona la opción de realizar una instalación que haga uso de SILO, en cuyo caso este último debe estar también instalado en la máquina. Las opciones que admite el *script* de configuración para incluir SILO son:

`--with-silo`: Configura la instalación buscando las librerías y cabeceras de SILO en los directorios por defecto (i.e. `/usr/local`, `/opt`, ...).

`--with-silo-dir=<silo_dir>`: Configura la instalación de MICSc buscando las librerías de SILO en `<silo_dir>/lib` y las cabeceras en `<silo_dir>/include`.

`--with-silo-flags=<silo_flags>`: Configura la instalación especificando exactamente los parámetros que se deben utilizar en el comando de compilación (por ejemplo, `-lsiloh5 -lhdf5 -L<dir> -I<dir>` ...).

De esta manera, es posible realizar la configuración con soporte para SILO:

```
$ ./configure --with-silo-dir=<silo_dir>
```

o bien realizar una instalación sin soporte para SILO, con PETSc como única dependencia.

```
$ ./configure
```

Llegados a este punto, es imprescindible definir la variable de entorno `MICSC_DIR` (en caso de no haberse definido ya anteriormente) para poder proceder con la compilación.

```
$ MICSC_DIR=$PWD; export MICSC_DIR
$ make
```

Esta compilación creará un directorio `$PETSC_ARCH` dentro del directorio de MICSc (`$MICSC_DIR`) que contendrá la librería compilada así como cualquier otro fichero específico de la configuración realizada. En este punto se puede comprobar la corrección de la instalación mediante

```
$ make test
```

Es importante destacar que estos tests hacen uso MPI, por lo que el testeo en máquinas con sistemas de colas para este tipo de ejecuciones puede ser problemático. Además, cabe destacar que es posible compilar los casos básicos proporcionados con la distribución de MICSc con

```
$ make examples
```

Los binarios generados, junto con el resto de ficheros necesarios para la ejecución de estos casos se pueden encontrar en `$MICSC_DIR/cases`, en un subdirectorio distinto para cada uno de los casos.

Ejecución

MICSc proporciona diferentes casos de uso sobre los que se pueden realizar diferentes pruebas. Para ello, accedemos al directorio de un caso como, por ejemplo, el del canal periódico, lo compilamos (si no lo hemos compilado ya antes con `make examples`) y lo ejecutamos.

```
$ cd $MICSC_DIR/cases/periodchannel
$ make
$ mpirun -np <p> ./main <args>
```

Las opciones que admiten los casos generados con MICSc se pueden obtener con el argumento `-help`. Al margen de las opciones proporcionadas por PETSc para vectores, matrices, *solvers*, precondicionadores y demás, las opciones de MICSc son las siguientes (`<n>` determina un entero, `r` un real, `e` un enumerado y `f` un fichero):

- Opciones de inicialización

  `-grid_file <f>` : Fichero de malla.

  `-rod_file <f>` : Fichero con los parámetros por defecto del caso.

- Opciones de monitorización

  `-log_level <e>` : Nivel de monitorización {0, 1, 2}

  `-out_ievery <n>` : Número de iteraciones exteriores a realizar para escribir resultados. -1 para no escribir resultados intermedios.

  `-out_idtevery <n>` : Número de pasos temporales a realizar para escribir resultados. -1 para no escribir resultados intermedios.

  `-start_instant_means <n>` : Paso de tiempo en el que se debe empezar el cálculo de medias de valores instantáneos. -1 para no realizar dicho cálculo.

  `-start_variance_means <n>` : Paso de tiempo en el que se debe empezar el cálculo de medias de las varianzas. -1 para no realizar dicho cálculo.

  `-inifl` : Determina si se debe escribir el resultado inicial.

  `-finfl` : Determina si se debe escribir el resultado final.

  `-print_props` : Determina si se deben escribir resultados de propiedades. Actualmente la única propiedad que admite la escritura de resultados es la viscosidad LES.

- Opciones de resolución del sistema de ecuaciones

  `-max_outer_its <n>` : Número máximo de iteraciones exteriores.

  `-min_outer_its <n>` : Número mínimo de iteraciones exteriores.

  `-res_max <r>` : Norma máxima del residuo normalizado.

  `-res_max <r>` : Norma máxima del vector corrección normalizado.

  `-k_restart` : Determina si se debe reiniciar una ejecución anterior a partir de un fichero de resultados binario `restart.bin`.

`-init_it <n>` : Número de la primera iteración.

`-k_wrap <e>` : Periodicidad del dominio del problema {`NONPERIODIC`, `XPERIODIC`, `YPERIODIC`, `XYPERIODIC`, `XYZPERIODIC`, `XZPERIODIC`, `YZPERIODIC`, `ZPERIODIC`, `XYZGHOSTED`}

`-n_ghost <n>` : Número de nodos vecinos.

`-gravity <r>` : Fuerza de la gravedad. 0 para ignorarla.

`-n_order_trans <e>` : Esquema temporal {`steady`, `eul1`, `eul2`, `adm2`, `adm3` }

`-deltat <r>` : Paso de tiempo.

`-tinit <r>` : Instante de tiempo inicial.

`-tfinal <r>` : Instante de tiempo final.

- Opciones de variables del problema

`-uref <r>` : Valor constante para la entrada de flujo en la dirección X.

`-mu <r>` : Valor de μ (viscosidad dinámica).

`-cs0 <r>` : Constante de Smagorinsky para el modelo LES.

`-dpdx <r>` : Valor del flujo de masa $\partial p/\partial x$

`-tref <r>` : Valor de referencia para la temperatura.

`-rho <r>` : Valor de ρ (densidad).

`-ndim <n>` : Número de dimensiones.

`-ncoupvar <n>` : Número de variables acopladas.

`-cont_eq <e>` : Ecuación de continuidad {`poisson`, `simple`, `simplec`, `simpler` }

`-inlet <e>` : Función para el flujo de entrada {`u`, `f`, `rho` }

`-les` : Determina si se debe utilizar el modelo LES.

`-kemod` : Determina si se debe utilizar el modelo Kϵ.

`-var_t` : Determina si se debe considerar la temperatura como una variable a resolver.

Implementación de nuevos casos

Al margen de los casos base proporcionados, MICSc permite la creación de nuevos casos. Para ello, en primer lugar se debe crear un nuevo directorio para el caso

```
$ cd $MICSC_DIR/cases
$ mkdir <caso>
```

Este directorio generalmente suele contener (al margen de otros ficheros que puedan ser necesarios para el caso particular) cuatro ficheros principales:

- `grd.inp`: Contiene la malla y las condiciones de contorno tal y como se especifica en el siguiente apartado.
- `rod.inp`: Contiene las opciones por defecto del caso. Se proporciona por comodidad para que no sea necesario especificarlas siempre por línea de comandos. Cabe destacar que las opciones que luego se puedan especificar en el momento de la ejecución sobreescriben las de este fichero.
- `makefile`: Fichero para la compilación del caso. Se puede copiar de cualquier otro caso.
- `main.c`: Código fuente para la ejecución del caso. Aparte de las funciones propias del caso, consta de una función `main` en la que se puede hacer uso de las siguientes llamadas a la librería de MICSc:

`MICScFinalize():`

Libera toda la memoria utilizada por la aplicación y llama a la rutina de finalización de los objetos de PETSc. En el código, debe ser la última llamada.

`MICScInitialize(PetscInt argc, char** argv[]):`

Se encarga de realizar la inicialización de PETSc, así como de leer los ficheros de entrada y reservar e inicializar todos los valores y estructuras de datos comunes a todos los casos. Debe ser la primera llamada del programa y los parámetros de entrada deben de ser punteros a los parámetros de entrada de la función `main`.

`MICScSetProp(St_Prop *prop, St_Patch *patch, ktypeProp propType, PetscReal cons, PetscErrorCode(funct*)(St_Prop *prop), ktypeIp interpType, PetscReal linRelax, PetscTruth calcMeans):`

Modifica los valores por defecto de una propiedad. El primer parámetro es el puntero a la propiedad a modificar y el resto son la región de la malla sobre la que se aplica (`patch`), el tipo de propiedad (`propType`), el valor de la propiedad si es constante (`cons`), la función de evaluación si es variable (`funct`), el tipo de interpolación (`interpType`) y la relajación lineal (`linRelax`). Además, es posible especificar si se desean calcular medias para esta propiedad mediante `calcMeans`.

`MICScSolve():`

Resuelve el sistema de ecuaciones. Es el núcleo principal de MICSc y la llamada que consume prácticamente todo el tiempo de ejecución de la simulación.

`MICScVarSetGamma(PetscInt index, ktypeProp gammaType, ktypeIp gammaInterpType, PetscReal gammaConst, PetscErrorCode (gammaFunct*) (St_Prop *prop)):`

Sustituye la propiedad del coeficiente de difusión por una nueva con los valores especificados como parámetro. El primer parámetro es el índice de la variable; es posible utilizar las macros `U`, `V`, `W`, `P`, etc para este parámetro. Esto es aplicable para todas las funciones de nombre `MICScVarSet*`.

`MICScVarSetInitValues(PetscInt index, ktypeProp initValuesType, ktypeIp initValuesInterpType, PetscReal initValuesConst, PetscErrorCode (initValuesFunct*) (St_Prop *prop)):`

Se define de manera análoga a `MICScVarSetGamma` para los valores iniciales de la variable en cada celda.

`MICScVarSetInterpolation(PetscInt index, ktypeIp interpType):`

Modifica el tipo de interpolación de la variable.

`MICScVarSetLimits(PetscInt index, PetscReal minValue, PetscReal maxValue):`

Modifica los valores mínimo y máximo de la variable.

`MICScVarSetReference(PetscInt index, PetscReal resRef, PetscReal corrRef):`

Modifica los valores de referencia de la variable.

`MICScVarSetRelaxation(PetscInt index, PetscReal linRelax, PetscReal fdtRelax):`

Modifica los valores de relajación de la variable.

`MICScSetInletFunction(PetscErrorCode (*function)(PetscInt*, PetscReal*)):`

Establece la función que se debe utilizar para calcular el flujo de entrada, en caso de que se defina tal condición de contorno.

Formato de entrada de la malla

Cada caso debe disponer de un fichero que defina tanto la malla como las condiciones de contorno. Este fichero se estructura de la siguiente manera:

```
<{0,1}>
<num_divisiones_eje_X>
<num_divisiones_eje_Y>
<num_divisiones_eje_Z>
<lista_coords_eje_X>
<lista_coords_eje_Y>
<lista_coords_eje_Z>
<lista_condiciones_de_contorno>
```

El primer valor representa si la malla es rectangular (0) o cilíndrica (1). Aunque actualmente sólo se soportan mallas rectangulares, también es posible que en un futuro se implementen mallas cilíndricas. Las siguientes tres líneas contienen un único número entero por línea, que define el número de divisiones para cada uno de los ejes. Posteriormente deben aparecer tres líneas con una lista de coordenadas para cada uno de los ejes. El tamaño de la lista se deberá corresponder con el número definido en las líneas anteriores y deberán estar formados por números reales separados por espacios. En este punto es importante destacar que se asume la coordenada $[0, 0, 0]$ como origen de la malla y no es necesario incluirla en este fichero. Por último, aparece una lista de condiciones de contorno donde cada una de éstas ocupa una línea distinta con el siguiente formato:

```
<x> <nx> <y> <ny> <z> <nz> <vecino> <tipo> <porosidad>
```

Los primeros 6 parámetros son de tipo entero y especifican la región de la malla sobre la que se define la condición de contorno ($[x, y, z] - [x + nx, y + ny, z + nz]$); el siguiente parámetro especifica sobre qué cara de las celdas se define la condición ({*north, south, east, west, high, low, none*}); el tipo define la condición de contorno ({*wall, moving_wall, inlet, outflow, outpress_newman, outpress_extrap, symmetry, fix*}); y la porosidad es un número real en el rango [-1, 1].

Formato de salida SILO

Por último, en caso de haber configurado la instalación de MICSc con SILO, los ficheros producidos podrán abrirse en diversos visores, tales como VisIt. Sin embargo, es necesario conocer la estructura jerárquica que presentan los ficheros SILO generados por MICSc. Dicha estructura se puede encontrar en la Figura A.1. En esta figura, los elementos marcados en negrita son hojas (variables y malla), mientras que los elementos en cursiva son directorios.

Así, los resultados se dividen en iniciales (`InitialResults`), intermedios (`IntermediateResults`), finales (`FinalResults`) y de paso temporal (`TimeStepResults`). Cabe destacar que los resultados de paso temporal

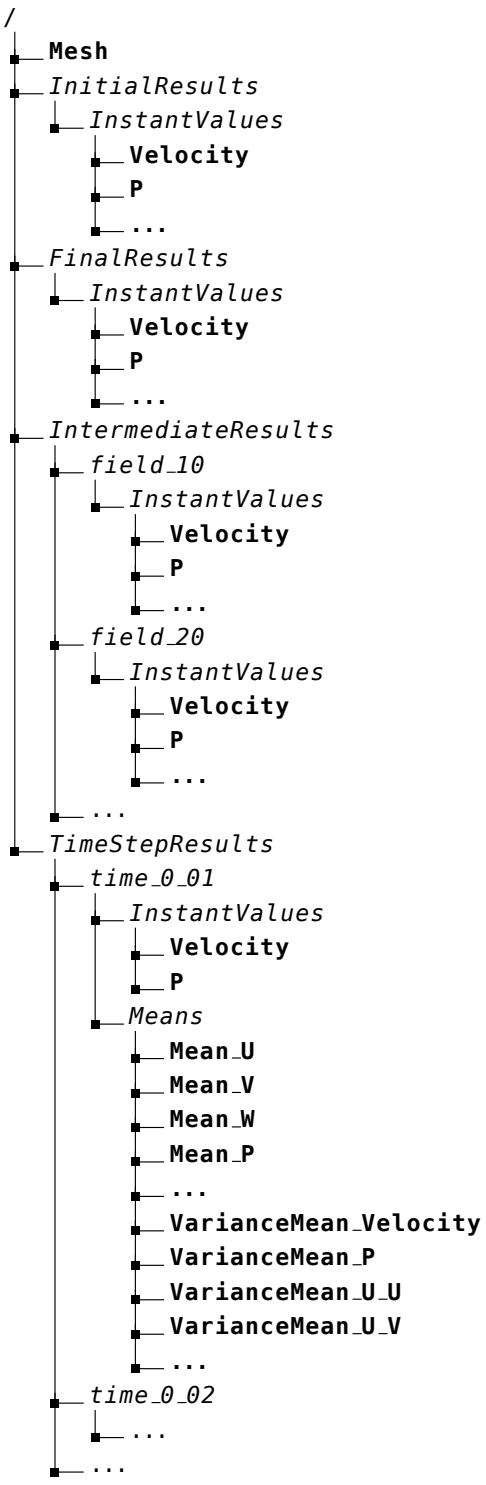

Figura A.1: Estructura jerárquica de un fichero SILO generado por MICSc.

se escriben al obtener la convergencia (o alcanzar el número máximo de iteraciones), mientras que los resultados intermedios se refieren a aquellos resultados producidos en una determinada iteración, a mitad de un paso temporal. Dentro de cada uno de estos directorios habrá subdirectorios para el paso de tiempo (`time_<tiempo>`) o la iteración (`field_<it>`) cuando corresponda. Por último, los directorios más alejados de la raíz contendrán valores instantáneos para las variables (`InstantValues`) o las medias cuando corresponda (`Means`). Es importante notar que para las medias `Mean_<var>` indica la media de los valores instantáneos de la variable `var`, `VarianceMean_<var>` indica la media de la varianza de la variable `var` y `VarianceMean_<var1>_<var2>` indica la media de la covarianza entre las variables `var1` y `var2`.

Bibliografía

[1] S.V. Patankar and D.B. Spalding. A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. *International Journal of Heat and Mass Transfer*, 15:1787–1806, 1972.

[2] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.

[3] BLAS - Basic Linear Algebra Subprograms, 2010. `http://www.netlib.org/blas`.

[4] LAPACK - Linear Algebra PACKage, 2010. `http://www.netlib.org/lapack`.

[5] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Web page, 2010. `http://www.mcs.anl.gov/petsc`.

[6] S. Pope. *Turbulent Flows*. Cambridge University Press, Cambridge, UK, 2000.

[7] J.H Ferziger and M. Perić. *Computational Methods for Fluid Dynamics*. Springer, 3rd edition, 2001.

[8] J. Bardina, J.H. Ferziger, and W.C. Reynolds. Improved subgrid scale models for large eddy simulation. *AIAA paper*, 80-1357, 1980.

[9] E.R. van Driest. On turbulent flow near a wall. *AIAA journal*, 23(11):1007–1010, 1036, 1956.

[10] M.T. Heath. *Scientific Computing. An Introductory Survey*. Mc Graw Hill, 2nd edition, 2000.

[11] V. Faber and T. Manteuffel. Necessary and sufficient conditions for the existence of a conjugate gradient method. *SIAM Journal on Numerical Analysis*, 21(2):352–362, 1984.

[12] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J.M. Donato, J Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1994.

[13] G. Fasshauser. Arnoldi iteration and GMRES, 2006. `http://www.math.iit.edu/~fass/477577_Chapter_14.pdf`.

[14] A. Cubero. *Resolución acoplada de las ecuaciones de Navier-Stokes mediante una implementación parcialmente implícita de la interpolación del momento*. PhD thesis, Área de Mecánica de Fluidos, Universidad de Zaragoza, 2007.

[15] M. Peric, R. Kessler, and G. Scheuerer. Comparison of finite-volume numerical methods with staggered and colocated grids. *Computers & Fluids*, 16(4):389–403, 1987.

[16] C.M. Rhie and W.L. Chow. Numerical study of the turbulent flow past an airfoil with trailing edge separation. *AIAA Journal*, 21(11):1525, 1983.

[17] A. Cubero, V. González, J.E. Román, N. Fueyo, and G. Palau-Salvador. MICSc: a PETSc-based parallel code for Large Eddy Simulation. In *Proceedings of the Seventh International Conference on Engineering Computational Technology*, Stirlingshire, UK, 2010. Civil-Comp Press.

[18] The HDF Group. HDF5 Web page, 2010. `http://www.hdfgroup.org/HDF5/`.

[19] The SILO Team. A mesh and field i/o library and scientific database, 2010. `https://wci.llnl.gov/codes/silo/index.html`.

[20] VisIt Visualization Tool, 2010. `https://wci.llnl.gov/codes/visit`.

[21] Tecplot, 2010. `http://www.tecplot.com`.

[22] G. Staffelbach, J.M. Senorer, L. Gicquel, and T. Poinsot. Large eddy simulation of combustion on massively parallel machines. In *High Performance Computing for Computational Science - VECPAR 2008*, Toulouse, France, 2008. Springer.

[23] B. Gropp. Sowing - Tools for Developing Portable Programs, 2010. `http://www.cs.uiuc.edu/homes/wgropp/projects/software/sowing/index.htm`.

[24] F. Schintke and M. Kammerhofer. Source to .html converters, 2010. `http://user.cs.tu-berlin.de/~schintke/x2html/index.html`.

[25] GNU Coding Standards, 2010. `http://www.gnu.org/prep/standards/standards.html#Managing-Releases`.

[26] SLEPc - Scalable Library for Eigenvalue Problem Computations, 2010. `http://www.grycap.upv.es/slepc/`.

[27] GRyCAP - Grid y Computación de Altas Prestaciones, 2010. `http://www.grycap.upv.es`.

[28] R.D. Moser, J. Kim, and N. Mansour. Direct numerical simulation of turbulent channel flow up to Reτ = 590. *Physics of Fluids*, 11(4):943–945, 2007.

[29] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA, 2nd edition, 2003.

[30] SALOMÉ Platform, 2010. `http://www.salome-platform.org/`.