Escuela Politécnica Superior de Jaén
Máster Universitario en Ingeniería Industrial

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior (Jaén)

Trabajo Fin de Máster

DESARROLLO DE CÓDIGOS NUMÉRICOS EN OPENFOAM PARA LA SIMULACIÓN DE PROCESOS DE TRANSMISIÓN DE CALOR

Alumno/a: **Ruiz Martínez, Alejandro**

Tutor/a: Prof. D. Patricio Bohórquez Rodríguez de Medina
Dpto.: Ingeniería Mecánica y Minera

Julio, 2018

Universidad de Jaén
Escuela Politécnica Superior de Jaén
Departamento de Ingeniería Mecánica y Minera

Don Patricio Bohórquez Rodríguez de Medina , tutor del Proyecto Fin de Carrera titulado: **Desarrollo de códigos numéricos en OpenFOAM para la simulación de procesos de transmisión de calor**, que presenta **Alejandro Ruiz Martínez**, autoriza su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, Julio de 2018

El alumno:

Los tutores:

Alejandro Ruiz Martínez

D. Patricio Bohórquez Rodríguez de Medina

Índice

1 Introducción y motivación. 6

1.1 Objetivo del Trabajo de fin Máster 7

1.2 Software: OpenFoam 7

1.2.1 Breve descripción sobre OpenFOAM 7

1.2.2 Fundamentos 8

1.2.3 La estructura de OpenFOAM 10

1.2.4 Códigos numéricos estándar 12

1.2.5 Algunas utilidades estándar 14

2 Contexto teórico y numérico. 17

2.1 Resumen 18

2.2 Introducción a los métodos numéricos 19

2.3 Método de volúmenes finitos (Finite Volume Method - FVM) 20

2.4 PyFoam: gestión de datos en OpenFOAM. 21

2.5 Ecuaciones fundamentales y condiciones de contorno 23

2.5.1 Ecuaciones fundamentales para fluidos 23

2.5.2 Condiciones de contorno 26

3 CASO A: Transferencia de calor por conducción y convección en problema multiregión. 29

3.1 Introducción y contexto teórico 30

3.2 Descripción del *solver* existente: chtMultiRegionFoam 31

3.3 Modificaciones incluidas en el *solver* 38

3.4 Preparación del caso 38

3.5 Resultados 59

3.6 Conclusiones 62

4 CASO B: Incorporación de calentamiento de sólido por efecto Joule y radiación. 64

4.1 Introducción y contexto teórico 65

4.1.1 Campo eléctrico 65

4.1.2 Radiación 67

4.2 Descripción del *solver* existente: joulechtMultiRegionFoam 72

4.3 Modificaciones incluidas en el *solver* 72

4.4 Preparación del caso (efecto Joule) 75

4.5 Resultados 78

4.6 Adición de radiación 81
4.7 Resultados 84
4.8 Conclusiones 86
5 CASO C: Flujos incompresibles e inmiscibles multifásicos con distinta temperatura. 87
5.1 Introducción y contexto teórico 88
5.2 Descripción del *solver* existente: interFoam 90
5.3 Modificaciones incluidas en el *solver* 92
5.4 Preparación del caso 97
5.5 Resultados 99
5.6 Conclusiones 101
6 CASO D: Proceso térmico de llenado por inyección..... 103
6.1 Introducción y contexto teórico 104
6.2 Descripción del solver existente: simpleFoam 107
6.3 Modificaciones incluidas en el solver 108
6.4 Preparación del caso 109
6.5 Resultados 113
7 Conclusiones finales 115
8 Bibliografía 116

Resumen

Este Trabajo de Fin de Máster está concebido para otorgar al lector una idea básica de la metodología necesaria para la creación de nuevos códigos numéricos utilizando OpenFoam. Además, se destaca la versatilidad de este software y su capacidad de personalización a problemas actuales de diferentes ámbitos de la ingeniería.

Agradecimientos

En primer lugar, me gustaría agradecer este Trabajo de Fin de Máster a mi tutor Patricio Bohórquez Rodríguez de Medina, por confiar en mi juicio y apoyarme en todo momento. A mis padres y hermana por su respaldo incondicional desde el primer momento que decidí estudiar ingeniería. Y por último a mi pareja, Carmen, porque sin ella no hubiera llegado tan lejos.

1 Introducción y motivación.

1.1 Objetivo del Trabajo de fin Máster

El objetivo de este Trabajo de Fin de Máster es el desarrollo de códigos numéricos que incrementen el potencial del software de cálculo popularmente conocido como OpenFoam. El objetivo principal es simular numéricamente procesos de transmisión de calor en problemas acoplados de fluidos y sólidos. Además, se se busca documentar todo el proceso seguido, de manera que los trabajos realizados a lo largo de todo este proyecto puedan ser empleados por otros/as ingenieros/as con posterioridad. Ya sea para consulta o para modificación y/o adaptación de los códigos aquí empleados a sus posibles necesidades. Por lo tanto, se desarrollarán modificaciones de códigos numéricos partiendo de tutoriales que se encuentran en la versión original de dicho software relacionados con los siguientes casos de estudio:

- Transferencia de calor por conducción y convección en problema multiregión (sólido/fluido) con diferentes mallas.
- Incorporación de calentamiento de sólido por efecto Joule (resistencia eléctrica) y radiación.
- Flujos incompresibles e inmiscibles multifásicos con distinta temperatura.
- Simulación térmica en procesos de llenado por inyección.

1.2 Software: OpenFoam

1.2.1 Breve descripción sobre OpenFOAM

OpenFOAM es el software libre líder en los campos relacionados con CFD (*Computational Fluid Dynamics*), pertenece y es mantenido por la fundación OpenFOAM que distribuye el código fuente exclusivamente bajo GPL (*General Public Licence*). Está licencia otorga al usuario libertad de modificar y redistribuir el software, garantizando la libertad continua de uso dentro de unos términos.

OpenFOAM fue creada por Henry Weller en 1989 bajo el nombre de FOAM y fue lanzada al mercado como “OpenFOAM”. Sus principales autores son Henry Weller, Chris Greenshields y Mattijs Janssens desde diciembre de 2004. Desde entonces, OpenFOAM ha continuado actualizándose y lanzando nuevas versiones para el público cada año, actualmente la última versión es la 5.0.

La Fundación OpenFoam es una organización que se encarga del copyright y documentación del código numérico de OpenFOAM, cuyo propósito es la distribución de este software para beneficio de los usuarios. Es una compañía registrada con base en Inglaterra que no tiene capital social ni accionistas, aunque tiene miembros comprometidos con los valores de software libre que se encargan de gestionar y dirigir la organización. No tiene empleados y el beneficio que se obtiene se retiene en la organización, no pudiéndose distribuir a dichos miembros. OpenFOAM es desarrollado y mantenido por individuales que contribuyen con su trabajo a este proyecto con el soporte y consentimiento de compañías que los emplean. Existe una comunidad fiable donde se otorga cierta autoridad a aquellos que contribuyen con material de muy alta calidad y mantienen un compromiso duradero.

Además, OpenFOAM es una herramienta única, altamente expansible y su característica principal es su naturaleza *devkit* (*software development kit*) que para CFD consiste en más de 1 millón de líneas de código de C++. También incluye cientos de aplicaciones creadas desde *devkit* las cuales pueden ser ampliadas y personalizadas rápidamente. En la actualidad, estas aplicaciones están siendo usadas dentro del mundo del CFD en ámbitos académicos, industriales, de investigación, gubernamentales y a través de incontables campos relacionados con la ciencia e ingeniería.

1.2.2 Fundamentos

OpenFOAM es un software basado en el método volúmenes finitos y aunque está principalmente focalizado a CFD, puede resolver problemas de diferente ámbito como: mecánica de sólidos computacional, acústica computacional, electromagnetismo, etc. Puede ser usado en paralelo sin necesidad de pagar por licencias. Además, este software proporciona sus propias herramientas de mallado, las cuales pueden ser altamente exportables o importables desde otros softwares de terceros, lo que lo hace un programa muy versátil.

El lenguaje de programación usado en OpenFOAM es C++, un código de programación orientado a objetos que está diseñado para ser un entorno flexible y programable para simulaciones numéricas. Usando este lenguaje de alto nivel se asegura la directa representación de las presentes ecuaciones a resolver y se

consigue que sea muy fácil de leer, entender, modificar y/o añadir nuevas características.

Un programa orientado a objetos generalmente hace uso de los siguientes conceptos: Clases, Objetos y Funciones. Estos conceptos permiten una alta eficiencia en la organización y diseño de programas y son esenciales para casos muy complejos. El lenguaje de C++ fue desarrollado a principios de 1980 por Stroustrup con el propósito de combinar la flexibilidad y eficiencia de los sistemas programados en C y los paradigmas de los programas orientados a objetos. Simplemente, hace conveniente usar cualquier programación estándar que se adapte a la tarea en cuestión, sin forzar a todos los usuarios a cualquier estilo particular. Esto es lo que hace a C++ uno de los lenguajes de programación más potentes.

Con la programación de alto nivel, se pueden implementar nuevos programas con relativa facilidad. Los usuarios no necesitan un profundo conocimiento referido la orientación a objetos y/o programación en C ++ pero deben conocer los principios detrás de la orientación a objetos, las clases, y tener algunas nociones básicas de sintaxis de código C ++. Pero sobre todo es mucho más importante la comprensión de las ecuaciones subyacentes, modelos, métodos y algoritmos de solución. Además, es posible modificar los programas existentes o utilizarlos como punto de partida para los nuevos. Principal característica que se utilizará en este proyecto.

Por lo tanto, OpenFOAM fomenta la reutilización de códigos, bibliotecas, funciones de bajo nivel, manejo de malla, etc. La facilidad con la que podemos configurar nuevos programas o *solvers* permite a los investigadores abordar problemas mirando más allá de las capacidades de software comercial de CFD. Otra característica única comparada con otros códigos comerciales es el acceso completo a la fuente. Siempre se puede verificar lo que vamos a hacer con solo echar un vistazo al código raíz. Esto hace que OpenFOAM sea ideal para investigación y desarrollo.

Sin embargo, OpenFOAM no viene con una GUI (*Graphical User Interface*), todo está controlado a través del terminal y esto puede crear problemas, especialmente si el usuario no está familiarizado con el entorno LINUX.

Para post-procesamiento, OpenFOAM cuenta con un entorno gráfico que recibe el nombre de *ParaView*. Esta es una aplicación de análisis y visualización de datos de fuente abierta y multiplataforma con fácil interfaz gráfica. Entonces, el principal problema al comenzar a usar OpenFOAM es reunir el conocimiento necesario para operar con las funcionalidades existentes, y esto se traduce en una curva de aprendizaje con mucha pendiente para el usuario inicial.

1.2.3 La estructura de OpenFOAM

La Ilustración 1 muestra la estructura básica de un caso en OpenFOAM, el cual contiene el mínimo de archivos necesarios para poder lanzar una simulación.

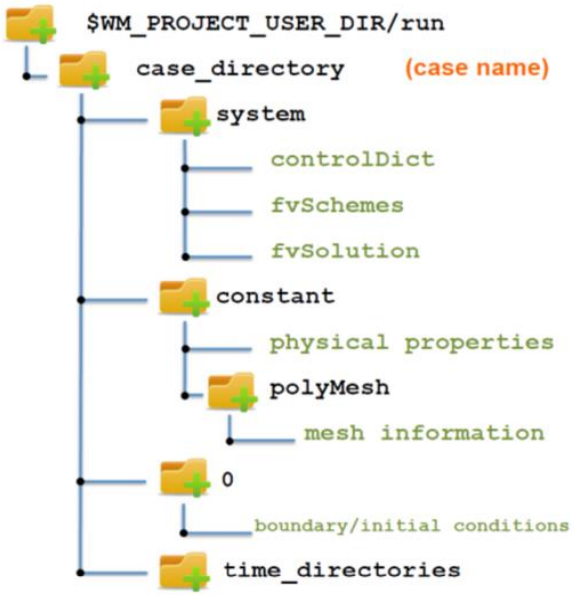

Figura 1.Estructura del directorio de un caso en OpenFOAM

La carpeta `system` es necesaria para la caracterización de los parámetros asociados al proceso de resolución del caso. En ella, podemos encontrar el control del paso de tiempo y discretización. Dentro de esta carpeta se pueden encontrar tres archivos siguientes:

- `controlDict`: Aquí se configuran los parámetros de control de la simulación, el paso de tiempo, el tiempo de comienzo y fin, etc. El siguiente código se ha sacado de uno de los tutoriales llamado *Cavity* que proporciona OpenFOAM.

Destacar que en cada uno de los archivos de este software la cabecera y el sub-diccionario *FoamFile* aparecerán siempre describiendo la versión del software que se está utilizando, la clase, carpeta origen y objeto, etc.

```
/*--------------------------------*- C++ -*----------------------------------*\
|  =========                 |                                                 |
| \\       /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\     /   O peration     | Version:  2.3.1                                 |
|   \\   /    A nd           | Web:      www.OpenFOAM.org                      |
|     \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "system";
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
application     icoFoam;
startFrom       startTime;
startTime       0;
stopAt          endTime;
endTime         0.5;
deltaT          0.005;
writeControl    timeStep;
writeInterval   20;
purgeWrite      0;
writeFormat     ascii;
writePrecision  6;
writeCompression off;
timeFormat      general;
timePrecision   6;
runTimeModifiable true;
// ************************************************************************* //
```

Nota: Para mantener claridad y ahorrar espacio, la cabecera será omitida en futuros códigos que se expondrán a lo largo de este trabajo.

- `fvSchemes`: Se establecen las características de discretización para la simulación numérica.
- `fvSolution`: este archivo está relacionado con el conjunto de ecuaciones, tolerancias y algoritmos de control para la simulación.

El directorio `constant` tiene una subcarpeta llamada:

- `polyMesh`: que contiene una descripción detallada de la malla y archivos relacionados con las propiedades físicas de las regiones computacionales. (Ejemplo: `transportProperties` o `thermophysicalproperties`).

La carpeta `0/` representa el directorio para el instante inicial (o tiempo cero) y contiene los valores iniciales y condiciones de contorno que el usuario debe especificar para definir el problema mientras que otras carpetas de tiempo contendrán los resultados escritos por OpenFOAM.

Debe recalcarse que los campos en OpenFOAM siempre deben ser inicializados, incluso cuando la solución no lo requiera, como los problemas estáticos. Es también muy común encontrar otros archivos, los cuales detallan otras características adicionales utilizadas en la simulación. Este es el caso del directorio:

- `Time directories`: contiene los archivos que se encargan de algunos campos en particular. El nombre de cada uno de ellos se basa en el tiempo de simulación en el que los resultados son escritos. El número de directorios de tiempo escritos puede ser cambiado en el valor `writeInterval` en el archivo controlDict localizado en la carpeta system. Cuanto menor sea el valor de `writeInterval`, mayor será el número de pasos de tiempo guardados. Por lo tanto, si se elige un número muy reducido se puede llegar a tener problemas de memoria. Una buena práctica de cara a la correcta simulación de casos, es lanzar el programa con una escritura de tiempos muy pequeña y a medida que veamos que la simulación se comporta correctamente ir aumentando dicho intervalo.

Además de estás carpetas iniciales, también se pueden encontrar otros archivos con extensión ".log", archivos de post-procesamiento, etc. Todo esto depende exclusivamente de las utilidades que se quieran crear a la hora de simular. Como la naturaleza de cada uno de estos archivos es muy diferente, no se detallará en profundidad ya que no está dentro del objetivo de este trabajo de fin de máster.

1.2.4 Códigos numéricos estándar

A continuación, se detalla una lista de algunos códigos numéricos que están relacionados con los temas que se tratarán en este documento. No significa que se vayan a utilizar todos, es más, solo se hará uso de 3 de ellos. Aunque a modo representativo, el lector puede tener una idea de los códigos que OpenFoam ofrece y su abanico de posibilidades.

- Códigos Básicos de CFD:
  - `potentialFoam`: resuelve flujos no transitorios, incompresibles, rotacionales e ignora el efecto de la gravedad.
- Flujo incompresible:
  - `icoFoam`: utilizado para simulaciones transitorias de flujos incompresibles y laminares de fluidos Newtonianos.
  - `PisoFoam`: para simulaciones transitorias de fluidos incompresibles.
  - `SimpleFoam`: simulaciones no transitorias con flujos incompresibles y turbulentos.
  - `pimpleFoam`: para simulaciones con un tiempo de paso considerablemente grande, flujos incompresibles usando el algoritmo PIMPLE.
- Flujo compresible:
  - `rhoPimpleFoam`: resuelve flujos compresibles transitorios tanto laminares como turbulentos.
  - `rhoSimplecFoam`: utiliza un algoritmo SIMPLEC para flujos laminares no transitorios y RANS para flujos turbulentos y compresibles.
  - `rhoSimpleFoam`: usa un algoritmo SIMPLE para flujos laminares no transitorios y RANS para flujos turbulentos y compresibles.
  - `sonicFoam`: utilizado para resolver flujos tanto laminares como turbulentos de gases con características transónicas/supersónicas.
- Transferencia de calor y convección:
  - `bouyantPimpleFoam`: resuelve flujos turbulentos y compresibles relacionados con ventilación y transferencia de calor. También está otra versión que utiliza la aproximación *Boussinesq* (`buoyantBoussinesqPimpleFoam`).
  - `bouyantSimpleFoam`: resuelve flujos no transitorios turbulentos y compresibles incluyendo radiación con ventilación y transferencia de calor.

También para esta versión existe una variante que utiliza para la aproximación *Boussinesq* (`buoyantBoussinesqPimpleFoam`).

- `chtMultiRegionFoam`: es una combinación de `heatConductionFoam` y `buoyantFoam` para una simulación entre regiones sólidas y fluidas. Hay una versión para flujo no transitorio llamada `chtMultiRegionSimpleFoam`.

- Otros

Aquí se han señalado algunos, aunque existen mucho más y sus campos de actuación pueden ser:

- Flujos con múltiples fases.
- Combustión.
- Simulación numérica directa (Direct Numerical Simulations -DNS).
- Flujos de partículas.
- Métodos numéricos moleculares.
- Electromagnetismo.
- Análisis de tensiones en sólidos.
- Finanzas.

1.2.5 Algunas utilidades estándar

Además de los códigos numéricos, OpenFOAM incluye una serie de herramientas muy útiles de cara a la simulación. Se han enumerado algunas de las más relevantes:

- Pre-procesado:
  - `changeDictionaty`: utilidad para cambiar la entrada de un diccionario.
  - `mapFields`: asigna campos de volumen de una malla a otra, leyendo e interpolando todos los campos presentes en el tiempo del directorio para

ambos casos. Casos paralelos y no paralelos son tratados sin la necesidad de reconstruirlos primero.

  - `setFields`: establece los valores de un conjunto de celdas a través de un diccionario.

- Generación de malla:
  - `blockMesh`: es un generador de malla de múltiples bloques.
  - `extrudeMesh`: extrusiona la malla desde un *patch* existente o desde un *patch* leído de un archivo.
  - `snappyHexMesh`: mallador automático de hexaedros.
- Conversión de malla:
  - `fluentMeshToFoam`: convierte una malla de Fluent a un formato compatible con OpenFOAM.
  - `foamMeshToFluent`: resulta una malla de OpenFOAM compatible con Fluent.
- Manipulación de malla:
  - `checkMesh`: comprueba la calidad de la malla.
  - `createBaffles`: convierte caras interiores en caras frontera. No duplica puntos.
  - `CreatePatch`: utilidad para crear *patches* fuera de las caras fronteras seleccionadas.
- Otras herramientas de mallado:
  - `modifyMesh`: manipula elementos de malla.
  - `refinementLevel`: intenta encontrar el nivel de refinamiento que hay en las mallas cartesianas. Se aplica antes de *snapping*.
  - `collapseEdges`: junta pequeños bordes y combina aquellos que se encuentran en una línea.
- Campos de velocidades en *post-processing:*

  - `Co`: calcula y escribe el número de Courant obtenido del campo `phi` como un `volScalarField`.
  - `Mach`: Calcula y opcionalmente escribe el número de Mach local desde el campo de velocidad *U* en cada paso de tiempo.

- *Post-processing* en turbulencia:
  - `createTurbulenceFields`: crea un conjunto completo de campos de turbulencia.
  - `R`: calcula y escribe el esfuerzo de Reynolds para el paso de tiempo actual.

- Procesamiento en paralelo:
  - `decomposePar`: automáticamente descompone una malla y sus campos en un caso de ejecución paralela en OpenFOAM.
  - `reconstructPar`: recompone campos del caso que ha sido descompuesto para la ejecución en paralelo en OpenFOAM.
  - `reconstructParMesh`: reconstruye una malla usando información geometría solo.
  - `redistributePar`: reconstruye una malla descompuesta existente de acuerdo a los ajustes pertinentes en el archivo `decomposePartDict`.

2 Contexto teórico y numérico.

2.1 Resumen

En este apartado se detallarán aquellos códigos numéricos y herramientas que son de interés para el objetivo de este trabajo y los diferentes casos a resolver. Es importante dejar claro que no se detallarán en profundidad todos los archivos involucrados en las simulaciones ya que esto supondría la extensión desmesurada del contenido. Si el lector necesita más información sobre OpenFOAM, cualquiera de los códigos numéricos u otras cuestiones referentes al software, se recomienda encarecidamente que consulte la fuente bibliográfica [1].

Por lo tanto, este documento presentará cuatro casos de gran interés ingenieril para simular. El primero de ellos o Caso A, se basa en la transferencia de calor por conducción y convección en un problema multiregión, es decir, la simulación se realizará involucrando fluido y sólido de forma que la caracterización del mallado será un parte fundamental de la simulación.

El caso B es muy parecido al Caso A pues se mantiene la geometría, pero con una peculiaridad, el sólido (cilindro) genera calor mediante efecto Joule (resistencia eléctrica). Además, de forma complementaria se simulará de nuevo dicho caso, pero añadiendo el fenómeno de radiación.

En el caso C, se estudiará fluidos incompresibles e inmiscibles. El llenado de un depósito, la caída libre de una gota de agua o incluso la rotura de una presa, son ejemplos que necesitan de un tratamiento computación especial y la creación de una interfaz entre fluido y gas.

Finalmente, el caso D, se simulará un proceso de inyección de plástico en una cavidad. Aunque OpenFoam no implementa intercambio de calor en dicho proceso, uno de los objetivos es acoplar la ecuación de la energía y obtener los resultados donde la viscosidad depende directamente de la temperatura.

En definitiva, se analizan casos que pueden ser altamente extrapolables a casos reales de ingeniería. La simulación de dichos procesos y el análisis de sus resultados pueden significar un ahorro significativo para las empresas a través de validación de

material, comportamiento térmico y otras cualidades antes comenzar con la producción.

2.2 Introducción a los métodos numéricos

Las ideas más sólidas sobre los métodos de solución numérica de ecuaciones de derivadas parciales se establecieron hace más de un siglo, pero su uso práctico solo comenzó a partir de 1950, cuando aparecieron las primeras computadoras. Aunque los primeros ordenadores eran extremadamente costosos y tenían un rendimiento relativamente bajo, la ratio coste-rendimiento ha ido aumentando considerablemente hasta nuestros días e incluso ahora no muestra signos de desaceleración. Con la incasable evolución de los ordenadores, el interés en las técnicas numéricas ha aumentado drásticamente. El campo de la ingeniería relacionado con la dinámica de fluidos computacional (CFD) ha experimentado un gran avance gracias a este desarrollo pudiendo llegar a una solución de compromiso de las ecuaciones que gobiernan la mecánica de fluidos.

Pero, ¿cómo funciona?

Para obtener una solución numérica aproximada, se debe de utilizar un método de discretización que aproxima las ecuaciones diferenciales por un sistema algebraico de ecuaciones que pueden ser resueltas mediante un ordenador. Estas aproximaciones son aplicadas a pequeños dominios en espacio y/o tiempo y la solución numérica proporciona los resultados en aquella zona discreta. La precisión de las soluciones numéricas es dependiente de la calidad de discretización utilizada. Sin embargo, no se es capaz de obtener soluciones precisas para todos los flujos, de forma que se debe determinar que se quiere producir y aprender a analizar y juzgar los resultados.

Primero, se debe considerar que los resultados numéricos siempre son aproximados. Algunas razones por las que existen diferencias entre los resultados numéricos y la realidad son las siguientes:

- Las ecuaciones diferenciales contienen aproximaciones o idealizaciones.
- Aproximaciones hechas en el proceso de discretización.

- Durante la resolución de las ecuaciones, se utilizan métodos iterativos. A menos que estos sean lanzados durante mucho tiempo, la solución exacta de las ecuaciones discretizadas no es alcanzada.

En principio, si las ecuaciones fundamentales son conocidas con precisión, se puede conseguir una solución de cualquier nivel de precisión. El hecho es que para muchos fenómenos las ecuaciones no están disponibles o la solución numérica no es factible. Esto nos obliga a introducir modelos y esta validación está basada en datos experimentales. Algunas veces ciertos modelos son usados incluso con el mejor tratamiento posible para reducir costes. También, utilizando una interpolación o aproximación más precisa, se pueden reducir errores de discretización pero normalmente esto incrementa el tiempo y coste de la solución a obtener.

2.3 Método de volúmenes finitos (Finite Volume Method - FVM)

Este trabajo se centra principalmente en el enfoque de discretización que utiliza OpenFoam. Por lo tanto, no se tratará con el Método de Diferencias Finitas (*Finite Difference Method – FDM*) y el Método de Elementos Finitos (*Finite Element Method – FEM*).

El método de volúmenes finitos usa la forma integral de las ecuaciones de conservación, de forma que la solución se subdivida en un número finito de volúmenes de control (*Control volumes - CVs*) contiguos y las ecuaciones son aplicadas a este tipo de *CVs*. Un nodo computacional reside en cada uno de los volúmenes de control donde la(s) variable(s) son calculada(s). Se utiliza la interpolación de valores para expresar valores variables en la superficie del *CV* en términos de nodos.

Como resultado, nosotros obtenemos una ecuación algebraica para cada *CV*, en la cual un número de nodos vecinos aparecen. Este método se usa con cualquier tipo de malla y es bastante aconsejable para geometrías complejas. Ya que la malla sólo especifica las fronteras del volumen de control, no es necesario que esté emparejado con un sistema de coordenadas.

Este método además es conservativo (respecto a las leyes de conservación) por definición: las integrales de superficie, representativas de los flujos de convectivos y difusivos, son las mismas para aquellos *CVs* que comparten fronteras.

Todos aquellos términos que son aproximados tienen un significado físico y es por esta razón porque FVM es fácilmente programable y muy popular entre diferentes campos de la ingeniería. La principal desventaja de este método comparado con FDM es su gran dificultad para desarrollar métodos con un orden mayor de dos para 3D. Esto está estrechamente relacionado con el hecho de que el enfoque de FV requiere tres niveles de aproximación: interpolación, diferenciación e integración. De hecho, en los códigos modernos es posible seleccionar diferentes opciones y le corresponde al usuario elegir cuál se adapta mejor para el caso investigado.

2.4 *PyFoam*: gestión de datos en OpenFOAM.

PyFoam es una librería basada en Python, que es un lenguaje de programación en el que se hace un esfuerzo considerable para generar un código legible. Es considerado un lenguaje de programación multiparadigma ya que puede soportar orientación a objetos, programación imperativa y en menor medida, programación funcional. Esta librería fue desarrollada para controlar de una manera más didáctica las simulaciones de OpenFOAM mediante código para hacer principalmente variaciones de parámetros y análisis de resultados. Sin embargo, no existe abundante material sobre dicha librería y su compatibilidad. En el caso que se quiera consultar con más profundidad estos detalles, se recomienda encarecidamente que consulte las citas bibliográficas [7] y [8].

Por lo tanto, PyFoam puede ser usado en OpenFOAM para:

- Analizar los archivos ".log" producidos por código numéricos en OpenFOAM.
- Ejecutar códigos numéricos y utilidades analizando sus resultados simultáneamente.
- Manipular parámetros de archivos y condiciones iniciales de una manera no destructiva para el código.
- Representar los residuales de las simulaciones lanzadas en OpenFOAM.

Las utilidades que están instaladas en esta librería se encuentran en `/usr/bin` y debe ser su camino estándar para la ejecución. Pueden ser llamadas también con –

`-help-option` en el terminal, lo que arroja las diferentes opciones disponibles e información. Además, dependiendo de su utilidad se pueden clasificar en:

Utilidades *Runner*: son las encargadas de empezar y controlar los archivos run (serie de comando secuenciales) en OpenFoam. La mayoría de estas herramientas son usadas para comenzar simulaciones en paralelo (el número de procesadores y host-file debe ser proporcionado por el usuario dependiendo de la implementación MPI).

- `pyFoamRunner.py`: Ayuda a ejecutar *solver* de OpenFoam. Necesita usualmente 3 argumentos (<*solver*><*directory*><*case*>), los resultados son mandados a un archivo .log que se ubica dentro del directorio del caso (`pyFoamSolver.logfile`). Además, se crea un directorio con el nombre `pyFoamSolver.analyzed` que contiene información relacionada con los residuales, tiempo de ejecución, información de continuidad y límite de variables.
- `pyFoamPlotRunner.py`: ayuda a ejecutar *solver* de OpenFoam. Necesita usualmente 3 argumentos (<*solver*><*directory*><*case*>), los resultados son mandados a un archivo .log que se ubica dentro del directorio del caso (`PyFoamSolver.logfile`). Los resultados se obtienen de forma gráfica y son los residuales del *solver* lineal, continuidad, número de Courant (opcional), tiempo de ejecución del paso de tiempo (opcional), etc.

Utilidades para *Logfiles*: Estas herramientas son usadas para analizar los resultados de los *solvers* de OpenFOAM que han sido escritos en un archivo .log:

  - `pyFoamPlotWatcher.py`: se basa en el archivo .log con los resultados del *solver* de forma que se pueda ver gráficamente la convergencia de la simulación hasta que es interrumpida.
  - `pyFoamStandardLogAnalyzer.py`: analiza los datos almacenados en un archivo .log y devuelve otro archivo con el mismo nombre más *analyzed* añadido.
- Utilidades *Networking*.
- Utilidades para manipular datos.

- Utilidades para manipular diccionarios (desde *scripts*).
- Utilidades relacionadas con *ParaView*.
- Otras.
- Herramientas GUI.
- Utilidades especiales.

En cada una de las utilidades instaladas en el paquete *PyFoam*, existen un número elevado de aplicaciones en cada uno de los tipos. Si se necesita más información sobre dichas utilidades, se recomienda encarecidamente consultar [12].

2.5 Ecuaciones fundamentales y condiciones de contorno

En esta sección, se presentan las ecuaciones diferenciales y las condiciones de contorno más relevantes para los casos de estudio. A su vez, dependiendo de la naturaleza física del problema se realizará un análisis de aquellos fenómenos dominantes en cada uno de ellos. De esta forma, las siguientes ecuaciones se centran únicamente en fluido, dejando las ecuaciones referentes al sólido para cada una de sus secciones en el caso en el que fuera necesario.

2.5.1 Ecuaciones fundamentales para fluidos

A continuación, se detallan las tres ecuaciones fundamentales dentro del ámbito de la mecánica de fluidos en forma conservativa. En el caso que se quiera más información del proceso de obtención de dichas expresiones, se recomienda que se consulte [2].

- Ecuación de la conservación de la masa:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \vec{V}) = 0 \tag{1}$$

- Ecuación de cantidad de movimiento:
  - Componente x:

$$\frac{\partial(\rho u)}{\partial t}+\frac{\partial(\rho u^2)}{\partial x}+\frac{\partial(\rho uv)}{\partial y}+\frac{\partial(\rho uw)}{\partial z} = \frac{-\partial(p)}{\partial x}+\frac{\partial}{\partial x}(\lambda\nabla\cdot\vec{V}+2\mu\frac{\partial u}{\partial x})+\frac{\partial}{\partial y}[\mu(\frac{\partial v}{\partial x}+\frac{\partial u}{\partial y})]+\frac{\partial}{\partial z}[\mu(\frac{\partial u}{\partial z}+\frac{\partial w}{\partial x})]+\rho f_x \quad (2)$$

- Componente y:

$$\frac{\partial(\rho v)}{\partial t}+\frac{\partial(\rho uv)}{\partial x}+\frac{\partial(\rho v^2)}{\partial y}+\frac{\partial(\rho vw)}{\partial z} = \frac{-\partial(p)}{\partial y}+\frac{\partial}{\partial x}[\mu(\frac{\partial v}{\partial x}+\frac{\partial u}{\partial y})]+\frac{\partial}{\partial y}(\lambda\nabla\cdot\vec{V}+2\mu\frac{\partial v}{\partial y})+\frac{\partial}{\partial z}[\mu(\frac{\partial w}{\partial y}+\frac{\partial v}{\partial z})]+\rho f_y \quad (3)$$

- Componente z:

$$\frac{\partial(\rho w)}{\partial t}+\frac{\partial(\rho uw)}{\partial x}+\frac{\partial(\rho vw)}{\partial y}+\frac{\partial(\rho w^2)}{\partial z} = \frac{-\partial(p)}{\partial z}+\frac{\partial}{\partial x}[\mu(\frac{\partial u}{\partial z}+\frac{\partial w}{\partial x})]+\frac{\partial}{\partial y}[\mu(\frac{\partial w}{\partial y}+\frac{\partial v}{\partial z})]+\frac{\partial}{\partial z}(\lambda\nabla\cdot\vec{V}+2\mu\frac{\partial w}{\partial z})+\rho f_z \quad (4)$$

- Ecuación de energía:

$$\begin{aligned}\frac{\partial}{\partial t}(\rho E)+\nabla\cdot\rho E\vec{V} &= \rho\dot{q}+\frac{\partial}{\partial x}(k\frac{\partial T}{\partial x})+\frac{\partial}{\partial y}(k\frac{\partial T}{\partial y})+\frac{\partial}{\partial z}(k\frac{\partial T}{\partial z})-\frac{\partial(up)}{\partial x}-\frac{\partial(vp)}{\partial y}-\frac{\partial(wp)}{\partial z}\\ &+\frac{\partial(u\tau_{xx})}{\partial x}+\frac{\partial(u\tau_{yx})}{\partial y}+\frac{\partial(u\tau_{zx})}{\partial z}+\frac{\partial(u\tau_{xy})}{\partial x}+\frac{\partial(u\tau_{yy})}{\partial y}+\frac{\partial(u\tau_{zy})}{\partial z}+\frac{\partial(u\tau_{xz})}{\partial x}\\ &+\frac{\partial(u\tau_{yz})}{\partial y}+\frac{\partial(u\tau_{zz})}{\partial z}+\rho\vec{f}\cdot\vec{V}\end{aligned} \quad (5)$$

donde:

- ρ es la densidad
- $\vec{V}$ es el vector velocidad; $\vec{V}=u\vec{\imath}+v\vec{\jmath}+w\vec{k}$
- p es la presión
- τ es el tensor de esfuerzos viscosos
- g es la gravedad
- E es la energía total; $E=e+V^2/2$
- T es la temperatura

- k es la conductividad térmica
- f son fuerzas de superficie

Para el caso de fluidos incompresibles, se asume que la variación de densidad es cero y se mantiene constante a lo largo del tiempo, por lo que la ecuación de la energía se puede desacoplar. De forma que se puede resolver continuidad y cantidad de movimiento primero y luego la distribución de temperaturas puede ser obtenida incluyendo la velocidad y la presión en la ecuación de la energía. Sin embargo, para los casos planteados aquí donde los fluidos se consideran compresibles, es necesario resolver la ecuación de la energía junto a la ecuación continuidad y cantidad de movimiento.

Además, la energía total específica, es la suma de la energía interna$\hat{u}$, la energía cinética $E_c = \|\vec{V}\|^2/2$ y la potencial $E_p = gz$ de forma que $E = \hat{u} + E_c + E_p$. A su vez, la ecuación de la energía puede ser escrita en término de entalpía, la cual es la suma de la energía interna y presión cinética $h = \hat{u} + p/\rho$. Generalmente la entalpía es modelada proporcional a la temperatura y a otra variable que se le conoce como calor específico a presión constate.

$$h = c_p T \tag{6}$$

En definitiva, las tres ecuaciones planteadas anteriormente contienen cinco incógnitas: velocidad, presión, densidad, energía y temperatura. Por lo tanto, necesitamos dos relaciones adicionales para poder resolver el sistema de tres ecuaciones. Estas relaciones son las ecuaciones de estado que relacionan las propiedades termodinámicas:

$$\begin{aligned} \rho &= \rho(p,T) \\ \hat{u} &= \hat{u}(p,T) \end{aligned} \tag{7}$$

Esta es conocida como la ecuación de estado y define la relación entre la densidad, la presión y la temperatura. Depende principalmente del tipo de fluido que se procesa. Por ejemplo, en el caso que el fluido sea un gas, la ley de gases ideales viene expresada como:

$$p = \rho RT \tag{8}$$

$$\hat{u} = c_v T + cte \tag{9}$$

donde $R = 8.3144598 J/(mol \cdot K)$es la constante del gas.

2.5.2 Condiciones de contorno

Las ecuaciones explicadas en el apartado anterior gobiernan el flujo de un fluido, independiente de que sea un túnel de viento o sobre un avión de pasajeros. Sin embargo, los campos fluidos son muy diferentes para cada caso. Esta diferencia la marca las condiciones de contorno. Las condiciones de contorno dictan la solución particular a obtener de las ecuaciones fundamentales. Primero, se describe la condición de contorno relacionada con el fluido viscoso. Aquí, esta premisa asume que la velocidad relativa sobre la superficie y el fluido es cero o nula. A esto se le conoce como la condición *no-slip*. Si la superficie es estacionaría con el flujo moviéndose a lo largo de ella, se tiene:

$$\vec{u} = \vec{v} = \vec{z} = 0 \tag{10}$$

Ahora si nos centramos en las condiciones relacionadas con la transferencia de calor que pueden ser aplicadas tanto a dominios fluidos como sólidos, las podemos clasificar como:

- Condición de contorno Dirichlet: también referida como condición de contorno de primer tipo. Está asociada con la condición de *no-slip* pero referida a la temperatura. Es decir, si la temperatura de la superficie de material está denotada con una temperatura T_w, entonces la temperatura del fluido inmediatamente en contacto con la superficie es también igual a T_w. En el caso de un problema, el cual la temperatura de la pared es conocida, la condición de contorno correcta para el fluido sería:

$$T = T_w \tag{11}$$

- Condición de contorno Neuman: referida a una condición de contorno de segundo tipo que describe las características en el flujo de calor. En el caso de no conocer las temperaturas de la pared y esta ser una función del tiempo,

entonces la Ley de Fourier proporciona una condición de contorno en la superficie. Es decir:

$$\dot{q_w} = -(k\frac{\partial T}{\partial n})_w \tag{12}$$

donde n denota la dirección normal de la pared.

Un caso especial para esta condición de contorno puede ser usada para modelar paredes adiabáticas, estableciendo un gradiente nulo de temperatura a lo largo de normal a la superficie. La expresión es:

$$(\frac{\partial T}{\partial n})_w = 0 \tag{13}$$

- Condición de contorno Robin: es una condición de tercer tipo, y una combinación entre las condiciones Dirichlet y Neuman. Su expresión característica es la siguiente:

$$aT + b\frac{\partial T}{\partial n} = g \tag{14}$$

En problemas de transferencia de calor, esta condición aparece típicamente relacionada con el flujo convectivo, con la siguiente forma:

$$k\frac{\partial T}{\partial n} + h(T - T_\infty) = 0 \tag{15}$$

o conocida más comúnmente como la Ley de Enfriamiento de Newton:

$$\dot{q} = h(T - T_\infty) \tag{16}$$

donde $\dot{q}$ es el flujo de calor (W/m^2), que es proporcional a la diferencia de temperaturas entre la superficie y el fluido, respectivamente. Además, h es el coeficiente térmico de convección y sus unidades en el SI son $(W/m^2 \cdot K)$.

En el caso de estar ante un análisis de flujo de calor desacoplado, estos tipos de condiciones de contorno son frecuentemente utilizadas en aproximaciones con regiones sólido y fluido. Por ejemplo, se puede asumir que la temperatura o flujo de

calor son uniformes. Además, las condiciones de contorno convectivas son usadas a menudo donde los efectos combinados del flujo están agrupados con el coeficiente térmico de convección, el cual se obtiene empíricamente. Este coeficiente no es una propiedad del material, sino que depende de muchos factores como la geometría, superficie, distribución de temperatura, condiciones de flujo, propiedades termo-físicas del fluido, etc. Para muchas aplicaciones, puede no ser práctico o posible obtener este parámetro empíricamente.

3 CASO A: Transferencia de calor por conducción y convección en problema multiregión.

3.1 Introducción y contexto teórico

El primer problema que se plantea, se basa principalmente en la transferencia de calor entre un sólido cilíndrico sumergido en un fluido. Las condiciones iniciales de temperatura del sólido son mayores que las del fluido, por lo que se experimentará una transferencia de energía del uno al otro.

A parte de la transferencia de calor, otra singularidad importante de esta simulación es la discretización de dos regiones altamente diferenciables, la región fluida y la sólida. Debido a sus características físicas dichas regiones no se pueden incluir en una sola malla, por lo que a la hora de preparar el caso se deberán de realizar dos mallas independientes pero unidas entre sí. A continuación, se muestran las ecuaciones relevantes en este estudio. La ecuación de conducción y la ecuación de convección:

- **Ecuación de conducción:** la conducción aparece a nivel molecular, donde la energía es transferida por las partículas más energéticas aquellas con menos energía. El ratio de transferencia de calor por conducción entre dos cuerpos es proporcional a la diferencia de temperatura entre ambos. Así, la variación de transferencia de calor por unidad de área o también llamado flujo de calor está basado en la ley de Fourier:

$$\vec{q} = -k\nabla T \tag{17}$$

donde:

  - $\vec{q}[W/m^2]$ es la densidad de flujo.
  - $k[W/(m \cdot K)]$ es la conductividad térmica del material.
  - $\nabla T[K/m]$ es el gradiente de temperatura.

El signo negativo indica que el calor es transferido en la dirección de menos temperatura.

- **Ecuación de convección:** el modo de transferencia de calor por convección compromete dos fenómenos. Además de la debida transferencia de energía

asociada al movimiento aleatorio molecular (difusión), también se transfiere energía a través del movimiento global o macroscópico del fluido. Es de general aceptación utilizar el término convección cuando se hace referencia a la transferencia de calor producida por movimiento aleatorio moléculas y el término advección cuando se refiere al transporte debido al movimiento volumétrico del fluido.

Dependiendo de la naturaleza del flujo, se puede distinguir entre dos tipos: convección natural y convección forzada. La convección natural ocurre debido a la expansión del fluido. Las variaciones de densidad producen que el fluido entre en movimiento, aunque comúnmente la convección natural ocurre debido al campo gravitacional, ya que eleva aquellos fluidos menos densos y hace descender aquellos con más densidad. Por otro lado, la convección forzada ocurre cuando el movimiento del fluido es causado por una fuente externa, como un ventilador o soplador, aquí los efectos producidos por la variación de densidad son poco relevantes.

La transferencia de calor por conducción viene predeterminada por la Ley de enfriamiento Newton:

$$\frac{dQ}{dt} = hA(T_s(t) - T_\infty) \tag{18}$$

donde:

- $Q[J]$ es flujo de calor por convección.
- $T_s[K]$ es la temperatura de la superficie del cuerpo sólido.
- $T_\infty[K]$ es la temperatura del fluido.
- $A[m^2]$ es el área de transferencia de calor.
- $h[W/(m^2 \cdot K)]$ es el coeficiente de transferencia de calor por convección.

3.2 Descripción del *solver* existente: chtMultiRegionFoam

En esta sección se detallará el código numérico o *solver* que se utilizará para simular el ejemplo expuesto en el apartado anterior. Ya que es un problema de transferencia de calor por conducción y convección en el que están involucrado dos

dominios (sólido y fluido), se utilizará el código numérico `chtMultiRegionFoam`. Este es un *solver* multiregión transitorio para transferencia de calor entre regiones. Es importante remarcar la diferencia entre dominio y región en este documento. Un dominio contiene múltiples regiones con las mismas propiedades físicas. Por ejemplo, en `chtMultiRegionFoam`, solo hay dos dominios (sólido y fluido), pero hay múltiples regiones (como `topAir`, `bottomAir`) para cada dominio.

La estructura de archivos en dicho código se muestra en el árbol siguiente:

```
|----- chtMultiRegionFoam.C
|----- derivedFvPatchFields
|       |----- solidWallHeatFluxTemperature
|      |       |----- solidWallHeatFluxTemperatureFvPatchScalarField.C
|       |      |----- solidWallHeatFluxTemperatureFvPatchScalarField.H
|      |----- solidWallMixedTemperatureCoupled
|             |----- solidWallMixedTemperatureCoupledFvPatchScalarField.C
|             |----- solidWallMixedTemperatureCoupledFvPatchScalarField.H
|----- fluid
|      |----- compressibleContinuityErrors.H
|      |----- compressibleCourantNo.C
|      |----- compressibleCourantNo.H
|      |----- compressibleMultiRegionCourantNo.H
|      |----- createFluidFields.H
|      |----- createFluidMeshes.H
|      |----- initContinuityErrs.H
|      |----- readFluidMultiRegionPIMPLEControls.H
|      |----- readFluidMultiRegionPISOControls.H
|      |----- setRegionFluidFields.H
|      |----- solveFluid.H
|      |----- storeOldFluidFields.H
|      |----- UEqn.H
|      |----- hEqn.H
|      |----- pEqn.H
|----- include
|      |----- setInitialMultiRegionDeltaT.H
|      |----- setMultiRegionDeltaT.H
|----- Make
|      |----- files
|      |----- options
|----- readPIMPLEControls.H
|----- regionProperties
|       |----- regionProperties.C
|      |----- regionProperties.H
|----- solid
       |----- createSolidFields.H
       |----- createSolidMeshes.H
       |----- readSolidMultiRegionPIMPLEControls.H
       |----- readSolidMultiRegionPISOControls.H
       |----- readSolidTimeControls.H
       |----- setRegionSolidFields.H
       |----- solidRegionDiffNo.C
       |----- solidRegionDiffNo.H
       |----- solidRegionDiffusionNo.H
       |----- solveSolid.H
```

donde:

- `chtMultiRegionFoam.C`: es el principal archivo fuente, este llama a los archivos necesarios y códigos.
- `derivedFvPatchFields/`: este subdirectorio contiene los archivos que establecen nuevas condiciones de contorno para el acoplamiento entre los dominios sólido y fluido.
- `fluid/`: este subdirectorio contiene archivos fuente que resuelven las ecuaciones de continuidad, momento, entalpía y presión para las regiones fluidas.
- `solid/`: este subdirectorio contiene los archivos fuente para resolver la ecuación de transferencia de calor por conducción en las regiones sólidas.
- `Include/`: es el subdirectorio encargado de contener los archivos relacionados con el set/reset del paso de tiempo.
- `Make/`: el subdirectorio que contiene archivos para la compilación.

En cambio, nos centraremos en el código fuente `chtMultiRegionFoam.C` se ha detallado de manera muy breve que realiza cada línea de código, para un mejor entendimiento del flujo de trabajo computacional.

```
#include "fvCFD.H"
#include "rhoThermo.H"
#include "turbulenceModel.H"
#include "fixedGradientFvPatchFields.H"
#include "regionProperties.H"
#include "compressibleCourantNo.H"
#include "solidRegionDiffNo.H"
#include "solidThermo.H"
#include "radiationModel.H"
#include "fvIOoptionList.H"
#include "coordinateSystem.H"
#include "fixedFluxPressureFvPatchScalarField.H"
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
int main(int argc, char *argv[])
{
    #include "setRootCase.H"
    #include "createTime.H"

    regionProperties rp(runTime);
    #include "createFluidMeshes.H"
    #include "createSolidMeshes.H"
    #include "createFluidFields.H"
```

```
    #include "createSolidFields.H"
    #include "initContinuityErrs.H"
    #include "readTimeControls.H"
    #include "readSolidTimeControls.H"
    #include "compressibleMultiRegionCourantNo.H"
    #include "solidRegionDiffusionNo.H"
    #include "setInitialMultiRegionDeltaT.H"

    while (runTime.run())
    {
        #include "readTimeControls.H"
        #include "readSolidTimeControls.H"
        #include "readPIMPLEControls.H"
        #include "compressibleMultiRegionCourantNo.H"
        #include "solidRegionDiffusionNo.H"
        #include "setMultiRegionDeltaT.H"
   runTime++
        Info<< "Time = " << runTime.timeName() << nl << endl;
        if (nOuterCorr != 1)
        {
            forAll(fluidRegions, i)
            {
                #include "setRegionFluidFields.H"
                #include "storeOldFluidFields.H"
            }
        }
        // --- PIMPLE loop
        for (int oCorr=0; oCorr<nOuterCorr; oCorr++)
        {
            bool finalIter = oCorr == nOuterCorr-1;

            forAll(fluidRegions, i)
            {
                Info<< "\nSolving for fluid region "
                    << fluidRegions[i].name() << endl;
                #include "setRegionFluidFields.H"
                #include "readFluidMultiRegionPIMPLEControls.H"
                #include "solveFluid.H"
            }
             forAll(solidRegions, i)
            {
                Info<< "\nSolving for solid region "
                    << solidRegions[i].name() << endl;
                #include "setRegionSolidFields.H"
                #include "readSolidMultiRegionPIMPLEControls.H"
                #include "solveSolid.H"
            }

        }
         runTime.write();
            Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }
    Info<< "End\n" << endl;
    return 0;
}
```

El propósito/descripción de los archivos incluidos en este código numérico se detallan a continuación:

- `fvCFD.H`: archivo estándar para el método de volúmenes finitos.
- `rhoThermo.H`: declara propiedades termodinámicas basadas en la compresibilidad.
- `turbulenceModel.H`: declara y define la clase para los modelos de turbulencia (RAS, LES y laminar).
- `fixedGradientFvPatchFields.H`: cambia un *patch* en campo y declara los tipos de campos primitivos como pueden ser escalares, tensores, vectores, etc.
- `regionProperties.H`: declara una clase para guardar la información de la región y ser utilizada en simulaciones de regiones acopladas.
- `compressibleCourantNo.H`: calcula y representa el resultado mínimo y máximo del número de Courant para las regiones fluidas.
- `solidRegionDiffNo.H`: calcula y representa el mínimo y máximo número de difusión para las regiones sólidas.

Ahora dentro de la función `main()` los siguientes archivos están incluidos antes de correr el ciclo de la simulación. Estas únicamente contienen trozos de código que son insertados en cada una de las localizaciones en el código general.

- `setRootCase.H`: sirve para comprobar la estructura de carpetas del caso.
- `createTime.H`: comprueba el tiempo de simulación de acuerdo a controlDict e inicia las variables en el tiempo.
- `createFluidMeshes.H`: crea la malla(s) para las regiones fluidas.
- `createSolidMeshes.H`: crea la malla(s) para las regiones sólidas.
- `createFluidFields.H`: crea los campos para la región fluida leyendo las propiedades físicas y térmicas de malla de: `rho`, `kappa`, `U`, `phi`, `g`, `turbulence`, `DpDtFluid`.

- `createSolidFields.H`: crea los campos para la región sólida leyendo las propiedades físicas y térmicas de malla de: `rho`, `cp`, `kappa`, `T`.
- `initContinuityErrs.H`: declara e inicializa el error acumulado de continuidad.
- `createTimeControls.H`: lee los parámetros de control usados por `setDeltaT`.
- `readSolidTimeControls.H`: lee los parámetros de control del número de difusión.
- `compressibleMultiRegionCourantNo.H`: calcula y otorga los números máximo de Courant para las regiones fluidas.
- `solidRegionDiffusionNo.H`: calcula el número de difusión para todas las regiones sólidas.
- `setInitialMultiRegionDeltaT.H`: establece el primer paso de tiempo para `chtMultiRegionFoam`.

Seguidamente en el ciclo `while (runTime.run())`, la inicialización de los archivos son ejecutadas y alguna de ellos son ejecutados de nuevo. Los propósitos son los siguientes:

- `readTimeControls.H`: leer los parámetros de control usados en `setDeltaT`.
- `readSolidTimeControls.H`: para leer los parámetros de control usados en el sólido.
- `readPIMPLEControls.H`: Para leer el `nOuterCorrectors` en `fvSolution`.
- `compressibleMultiRegionCourantNo.H`: calcula el número de Courant para regiones fluidas.
- `solidRegionDiffusionNo.H`: calcula el número de difusión para regiones fluidas.
- `setMultiRegionDeltaT.H`: resetea el paso de tiempo para mantener constante un número de Courant y difusión máximo.

Otra importante parte del código es el ciclo PIMPLE. Esto es un algoritmo basado en una combinación de los algoritmos PISO (*pressure-implicit split-operator*) y SIMPLE (*semi-implicit method for pressure-linked equations*). Para la mayoría de aplicaciones en OpenFOAM, se utiliza tanto PISO, SIMPLE o su combinación en el algoritmo PIMPLE. Estos algoritmos son procedimientos de iteración para el acoplamiento de las ecuaciones de conservación de masa y momento. PISO y PIMPLE son usado para problemas transitorios y SIMPLE para problemas estacionarios.

Dentro del algoritmo PIMPLE las regiones fluidas se calculan primero y después las sólidas son calculadas a través de los datos obtenidos del campo fluido. Ambas son calculadas al mismo paso de tiempo, aunque convergen en iteraciones separadas.

En cuanto al fluido, las ecuaciones resuelven la densidad `rho`, la velocidad `U`, la presión `p` y la presión derivativa `DpDt`. En cuanto a la parte térmica, es esencialmente lo mismo que la parte sólida, aunque la variable a resolver por la ecuación de la energía es la entalpía $h$en vez de la temperatura `T`.

$$h = c_p \cdot dT \tag{19}$$

El código `solveFluid.H` resuelve cuatro ecuaciones:

```
 if (oCorr == 0)
    {
        #include "rhoEqn.H"                      //Mass conservation equation
    }
        #include "Ueqn.H"                           //Momentum equation
        #include "Eeqn.H"                           //Thermal energy equation
    // --- PISO loop
    for (int corr=0; corr<nCorr; corr++)
    {
        #include "pEqn.H"                        //Pressure equation
    }
turb.correct();
rho = thermo.rho();
```

El código `solveSolid.H` resuelve la ecuación de la energía para las regiones fluidas de la siguiente forma:

```
    for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
    {
```

```
        tmp<fvScalarMatrix> hEqn
        (
          fvm::ddt(betav*rho, h)
          - (
                thermo.isotropic()
             ? fvm::laplacian(betav*thermo.alpha(), h, "laplacian(alpha,h)")
             : fvm::laplacian(betav*taniAlpha(), h, "laplacian(alpha,h)")
            )
          ==
            fvOptions(rho, h)
        );
        hEqn().relax();
        fvOptions.constrain(hEqn());
        hEqn().solve(mesh.solver(h.select(finalIter)));
        fvOptions.correct(h);
    }
}
thermo.correct();
Info<< "Min/max T:" << min(thermo.T()) << ' ' << max(thermo.T()) << endl;
```

En este caso el código está resolviendo la ecuación de la energía para la entalpía usando la difusividad térmica α, donde k y C_p son la conductividad y el calor específico del sólido,

$$\alpha = \frac{k}{C_p} \tag{20}$$

Para obtener más información sobre los diferentes códigos involucrados en el cálculo, se recomienda navegar por los archivos de la carpeta raíz y consultar la referencia [19].

3.3 Modificaciones incluidas en el *solver*

En este primer caso no se ha realizado ninguna modificación al *solver*. Es decir, no se ha añadido ninguna caracterización especial para un problema en concreto como se hará en los casos B, C y D. Por ello, se ha prestado más atención en la descripción detallada de los diferentes códigos o archivos que intervienen en este primer caso con la única intención de otorgar al lector de una primera base teórica del flujo de trabajo del software.

3.4 Preparación del caso

En la ilustración 2 se muestra el esquema físico del modelo, el rectángulo será el fluido que se desplazará de izquierda a derecha y el cilindro será el sólido.

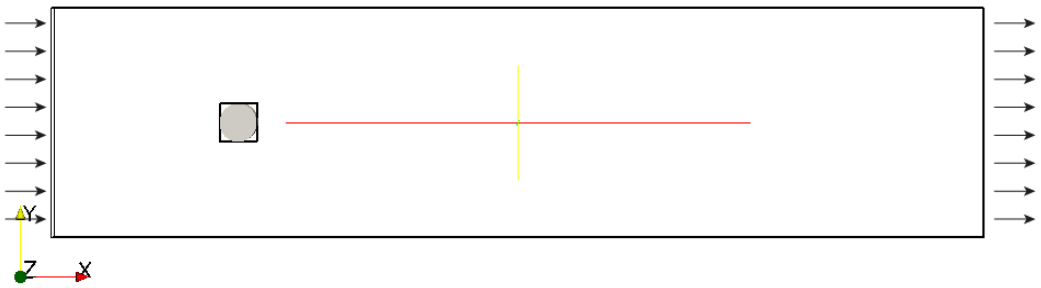

Figura 2. Esquema físico del Caso A.

Las condiciones iniciales del problema se detallan a continuación. Además, se ha incluido en aquellas en la que sea necesario su nombre característico en OpenFOAM, ya que son relevantes a la hora del desarrollo del código numérico.

- Temperatura del fluido: 300 K.
- Temperatura del sólido: 350 K.
- Difusividad térmica $alphat[kg/m \cdot s]$: 0
- Disipación de energía cinética $epsilon[m^2/s^3]$: 0.01
- Conductividad térmica $k[m^2/s^2]$: 0.1
- Presión $p[kg/m \cdot s^2]$: $1 \cdot 10^5$
- Presión manométrica $p - rgh[kg/m \cdot s^2]$: $1 \cdot 10^5$
- Densidad $rho[kg/m^3]$: 8000
- Temperatura $T[K]$: 300
- Velocidad del fluido $U[m/s]$: 0.01 (dirección X)

Una vez conocidas las ecuaciones involucradas en la simulación, es de vital importancia conocer la estructura organizativa de las carpetas y archivos para cada caso que se pretende abordar. Además, se detallará el contenido de cada una de las carpetas y aquellos archivos que se consideran relevantes sin entrar en demasiado detalle, debido a que ese no es el principal objetivo de este trabajo.

A continuación, se presenta una visión esquematiza de archivos y carpetas que componen el Caso A. Esta va desde la carpeta inicial, a través de sus subcarpetas y finalmente los archivos de texto que las componen.

```
|---- fullCylinderMultiRegionScripting2.3.1
|     |----- 0
|     |     |----- alphat
|     |     |----- epsilon
|     |     |----- k
|     |     |----- p
|     |     |----- p_rgh
|     |     |----- rho
|     |     |----- T
|     |     |----- U
|     |----- constant
|     |     |----- extAir
|     |     |     |----- g
|     |     |     |----- radiationProperties
|     |     |     |----- RASProperties
|     |     |     |----- thermophysicalProperties
|     |     |     |----- turbulenceProperties
|     |     |----- midSolid
|     |     |     |----- radiationProperties
|     |     |     |----- thermophysicalProperties
|     |     |----- polyMesh
|     |     |     |----- blockMeshDict
|     |     |----- regionProperties
|     |----- system
|     |     |----- extAir
|     |     |     |----- changeDictionatyDict
|     |     |     |----- fvSchemes
|     |     |     |----- fvSolution
|     |     |----- midSolid
|     |     |     |----- changeDictionatyDict
|     |     |     |----- fvSchemes
|     |     |     |----- fvSolution
|     |     |----- controlDict
|     |     |----- decomposeParDict
|     |     |----- extrudeMeshDict
|     |     |----- fvSchemes
|     |     |----- fvSolution
|     |     |----- meshQualityDict
|     |     |----- snappyHexMeshDict
|     |----- run
```

La carpeta `0/` es la encargada de reunir los ficheros relacionados con las propiedades físicas que intervendrán en la simulación, es decir, las condiciones iniciales. De ahí que exista un archivo de texto para uno de ellas. A continuación, se muestra, a modo de ejemplo el archivo de la velocidad U:

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       volVectorField;
    object      U;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
dimensions      [0 1 -1 0 0 0 0];
internalField   uniform (0.01 0 0);  // Velocidad de 0.01 m/s en dirección x
boundaryField
```

```
{
    ".*"
    {
        type            calculated;
    }
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

Nota: los encabezados en rojo en los próximos códigos presentados en este documento serán omitidos por motivos de simplicidad. Únicamente se mostrarán los párrafos de código que sean relevantes para el objetivo de este proyecto.

La carpeta `constant` contiene una descripción completa de los archivos relacionados con el fluido (`extAir`), el sólido (`midSolid`) y la malla de la región fluida (`polyMesh`). De esta forma, en `extAir` encontramos los diccionarios necesarios para la caracterización del sólido y el fluido: gravedad que actúa sobre él (`g`), propiedades térmicas en el diccionario `thermophysicalProperties`, los posibles modelos de turbulencia en `RASProperties` o la activación de la radiación en `radiationProperties` como se hará en el caso B.

Por lo tanto, se muestran los diccionarios que se han mencionado anteriormente para el fluido únicamente, para que el lector tenga una idea general de estos archivos y su contenido. Si necesita información detallada, para ello, consulte [13].

El primero de ellos, es `g`. Aquí se asigna el valor de la constante de gravitación tal que:

```
dimensions      [0 1 -2 0 0 0 0];
value           (0 -9.81 0);
```

En `radiationProperties` se especifican todas aquellas características relacionadas con el modelo de radiación que se pretende utilizar, en el caso de que lo hubiera. En caso contrario, el diccionario quedaría:

```
radiation       off;
radiationModel  none;
```

En `RASProperties`, se detallan las propiedades de turbulencia referente los modelos existentes.

```
RASModel laminar;
turbulence      on;
```

```
printCoeffs     on;
```

El siguiente es `thermophisicalProperties`. Aquí se consideran las propiedades termo-físicas que interviene en la simulación y referidas tanto al sólido como al fluido, en su carpeta correspondiente. El siguiente diccionario es el correspondiente al fluido:

```
thermoType
{
    type            heRhoThermo;
    mixture         pureMixture;
    transport       const;
    thermo          hConst;
    equationOfState rhoConst;
    specie          specie;
    energy          sensibleEnthalpy;
}

mixture
{
    specie
    {
        nMoles          1;
        molWeight       18;
    }
    equationOfState
    {
        rho             1000;
    }
    thermodynamics
    {
        Cp              4181;
        Hf              0;
    }
    transport
    {
        mu              959e-6;
        Pr              6.62;
```

Ahora dentro de la carpeta `polyMesh`, se encuentra el diccionario `blockMeshDict`. Es el encargado de mallar el dominio computacional que se pretende simular, estableciendo además las condiciones de contorno de las regiones computacionales. Esto se puede ver el contenido del archivo y cómo se ha creado la región que se muestra en la ilustración 2.

```
convertToMeters .1;

vertices
(
    (-5 -3 -.1)
    (20 -3 -.1)
    (20 3 -.1)
```

```
    (-5 3 -.1)
    (-5 -3 0.1)
    (20 -3 0.1)
    (20 3 0.1)
    (-5 3 0.1)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (50 12 1) simpleGrading (1 1 1)
);

edges
(
);

boundary
(
    exterior
    {
        type wall;
        faces
        (
            (3 7 6 2)
            (1 5 4 0)
        );
    }
    inlet
    {
        type patch;
        faces
        (
            (0 4 7 3)
        );
    }
    outlet
    {
        type patch;
        faces
        (
            (2 6 5 1)
        );
    }
    front
    {
        type empty;
        faces
        (
            (0 3 2 1)
        );
    }
    back
    {
        type empty;
        faces
        (
            (4 5 6 7)
        );
    }
);
```

```
mergePatchPairs
(
);
```

Finalmente, en esta carpeta raíz existe otro diccionario llamado `regionProperties` *y* necesario para diferenciar el dominio computacional sólido del fluido, como se muestra a continuación.

```
regions
(
    fluid       (extAir)
    solid       (midSolid)
);
```

Y, por último, la carpeta `system` contiene las características de la discretización y de la solución. Al ser un problema multiregión cada dominio tendrá sus características de discretización y solución. A continuación, se han representado los diccionarios `fvSchemes` y `fvSolution` para cada una de las regiones.

Nota: los diccionarios fvSchemes y fvSolution que se encuentran en la carpeta raíz, y no dentro de las carpetas correspondiente a las regiones fluida y sólida están vacías. Pero su inclusión en el caso es necesaria para el éxito de la simulación.

Fluido

```
ddtSchemes
{
    default Euler;
}

gradSchemes
{
    default         Gauss linear;
}

divSchemes
{
    default         none;

    div(phi,U)       Gauss upwind;
    div(phi,K)       Gauss linear;
    div(phi,h)       Gauss upwind;
    div(phi,k)       Gauss upwind;
    div(phi,epsilon) Gauss upwind;
    div(phi,R)       Gauss upwind;
    div(R)           Gauss linear;
   div((muEff*dev2(T(grad(U))))) Gauss
linear;
}

laplacianSchemes
{
```

Sólido

```
ddtSchemes
{
    default         Euler;
}

gradSchemes
{
    default         Gauss linear;
}

divSchemes
{
    default         none;
}

laplacianSchemes
{
    default             none;
    laplacian(alpha,h)  Gauss linear
corrected;
}

interpolationSchemes
{
    default         linear;
}
```

```
    default         Gauss linear corrected;
}

interpolationSchemes
{
    default         linear;
}

snGradSchemes
{
    default         corrected;
}

fluxRequired
{
    default         no;
    p_rgh;
}
```

```
snGradSchemes
{
    default         corrected;
}

fluxRequired
{
    default         no;
}
```

```
solvers
{
    "(rho|rhoFinal)"
    {
        solver          PCG
        preconditioner  DIC;
        tolerance       1e-7;
        relTol          0;
    }

    p_rgh
    {
        solver           GAMG;
        tolerance        1e-7;
        relTol           0.01;

        smoother         GaussSeidel;

        cacheAgglomeration true;
        nCellsInCoarsestLevel 10;
        agglomerator     faceAreaPair;
        mergeLevels      1;
    }

    p_rghFinal
    {
        $p_rgh;
        tolerance        1e-7;
        relTol           0;
    }

    "(U|h|k|epsilon|R)"
    {
        solver           PBiCG;
        preconditioner   DILU;
        tolerance        1e-7;
        relTol           0.1;
    }
```

```
solvers
{
    h
    {
        solver           PCG;
        preconditioner   DIC;
        tolerance        1e-06;
        relTol           0.1;
    }

    hFinal
    {
        $h;
        tolerance        1e-06;
        relTol           0;
    }
}

PIMPLE
{
    nNonOrthogonalCorrectors 0;
}
// * * * * * * * * * * * * * * * * * * * * //
```

```
    "(U|h|k|epsilon|R)Final"
    {
        $U;
        tolerance        1e-07;
        relTol           0;
    }
}

PIMPLE
{
    momentumPredictor   on;
    nCorrectors         2;
    nNonOrthogonalCorrectors 0;
}

relaxationFactors
{
    fields
    {
    }
    equations
    {
        "h.*"           1;
        "U.*"           1; }
}
```

El diccionario `changeDictionaryDict`, ayuda al usuario a cambiar o ajustar las condiciones de contorno de una manera directa. En este diccionario están definidas todas aquellas propiedades que intervienen en la simulación como la velocidad, temperatura, conductividad térmica, etc. En el caso que el usuario necesite modificar una de las muchas características, únicamente debe dirigirse a este diccionario (uno para región) y alterar aquella característica que se requiera, reduciendo drásticamente el tiempo de introducción de datos.

A continuación, se puede ver que contiene dicho diccionario para la región fluida, la región sólida tiene la misma estructura, pero contenido ajustado a sus propiedades.

```
dictionaryReplacement
{
// Ahora definimos las condiciones de contorno en el diccionario para no tener que
introducirlas a mano.
    U
    {
        internalField   uniform (0.01 0 0);

        boundaryField
        {
            exterior
            {
                type            fixedValue;
                value           uniform (0 0 0);
```

```
        }
        outlet
        {
            type            inletOutlet;
            inletValue      uniform ( 0 0 0 );
            value           uniform ( 0.01 0 0 );
        }
        inlet
        {
            type            zeroGradient;
            value           uniform (0.01 0 0);
        }
        "extAir_to_.*"
        {
            type            fixedValue;
            value           uniform (0 0 0);
        }
    }
}

T
{
    internalField   uniform 300;

    boundaryField
    {
        inlet
        {
            type            fixedValue;
            value           uniform 300;
        }
        outlet
        {
            type            inletOutlet;
            inletValue      uniform 300;
            value           uniform 300;
        }
        exterior
        {
            type            zeroGradient;
        }
        "extAir_to_.*"
        {
      type            compressible::turbulentTemperatureCoupledBaffleMixed;
            Tnbr            T;
            kappa           fluidThermo;
            kappaName       none;
            value           uniform 300;
        }
    }
}

epsilon
{
    internalField   uniform 0.01;

    boundaryField
    {
        inlet
        {
            type            fixedValue;
```

```
            value           uniform 0.01;
        }
        outlet
        {
            type            inletOutlet;
            inletValue      uniform 0.01;
            value           uniform 0.01;
        }
        exterior
        {
            type            compressible::epsilonWallFunction;
            value           uniform 0.01;
        }
        "extAir_to_.*"
        {
            type            compressible::epsilonWallFunction;
            value           uniform 0.01;
        }
    }
}

k
{
    internalField   uniform 0.1;

    boundaryField
    {
        inlet
        {
            type            fixedValue;
            value           uniform 0.1;
        }
        outlet
        {
            type            inletOutlet;
            inletValue      uniform 0.1;
            value           uniform 0.1;
        }
        exterior
        {
            type            compressible::kqRWallFunction;
            value           uniform 0.1;
        }
        "extAir_to_.*"
        {
            type            compressible::kqRWallFunction;
            value           uniform 0.1;
        }
    }
}

p
{
    internalField   uniform 100000;

    boundaryField
    {
        inlet
        {
            type            calculated;
            value           uniform 1e5;
```

```
            }
            outlet
            {
                type            calculated;
                value           uniform 1e5;
            }
            exterior
            {
                type            calculated;
                value           uniform 1e5;
            }
            "extAir_to_.*"
            {
                type            calculated;
                value           uniform 1e5;
            }
        }
    }

    p_rgh
    {
        internalField   uniform 100000;

        boundaryField
        {
            inlet
            {
             type            zeroGradient; //fixedFluxPressure; //fixedValue;
             value           uniform 1e5;
            }
            outlet
            {
             type            fixedValue; //zeroGradient;
             value           uniform 1e5;
            }
            exterior
            {
             type            zeroGradient; //fixedFluxPressure; //zeroGradient;
             value           uniform 1e5;
            }
            "extAir_to_.*"
            {
         type        zeroGradient; //fixedFluxPressure; //zeroGradient;
//buoyantPressure;
                value           uniform 1e5;
            }
        }
    }
}
```

Además de los diccionarios relacionados con extrusión (`extrudeMeshDict`) y calidad de malla (`meshQualityDict`), contiene una de las herramientas de mallado más importantes y usadas en OpenFOAM, `snappyHexMesh`. Esta utilidad genera mallas hexaédricas de 3 dimensiones y divide dichos hexaedros automáticamente como un método de refinamiento de malla en superficies triangulares, permitiendo

geometrías importadas con formato STL (*Stereolithography*) o OBJ (*Wavefront Object*).

La malla se conforma en todo el dominio de forma iterativa, refinando una malla inicial (la creada con `blockMesh`) y dividiendo hexaedros en otros con menos superficie. Esta herramienta otorga una notable versatilidad a la hora del refinamiento y calidad final de malla. Este diccionario puede abrumar al lector menos experimentado en OpenFOAM, por su complejidad y la ausencia de pre-visualización sobre la malla final de cada una de las características del documento. Si este es el caso, se recomienda encarecidamente que se consulte [14]. Además, las aclaraciones de cada uno de las líneas de código se han mantenido y cambiado de color para una mejor compresión.

```
// Which of the steps to run
castellatedMesh true;
snap            true;
addLayers       false;


// Geometry. Definition of all surfaces. All surfaces are of class
// searchableSurface.
// Surfaces are used
// - to specify refinement for any mesh cell intersecting it
// - to specify refinement for any mesh cell inside/outside/near
// - to 'snap' the mesh boundary to the surface

geometry
{
      midSolid
      {
        type searchableCylinder;
        point1 (0 0 -0.01);
       point2 (0 0 0.01);
        radius 0.05;
      }
      extAir
      {
        type searchableBox;
        min (-.5 -.3 -0.01);
        max ( 2.0 .3  0.01);
      }
};

// Settings for the castellatedMesh generation.

castellatedMeshControls
{

    // Refinement parameters
    // ~~~~~~~~~~~~~~~~~~~~
```

```
    // If local number of cells is >= maxLocalCells on any processor
    // switches from from refinement followed by balancing
    // (current method) to (weighted) balancing before refinement.

maxLocalCells 2000000;

    // Overall cell limit (approximately). Refinement will stop immediately
    // upon reaching this number so a refinement level might not complete.
    // Note that this is the number of cells before removing the part which
    // is not 'visible' from the keepPoint. The final number of cells might
    // actually be a lot less.

maxGlobalCells 4000000;

    // The surface refinement loop might spend lots of iterations
    // refining just a few cells. This setting will cause refinement
    // to stop if <= minimumRefine are selected for refinement. Note:
    // it will at least do one iteration (unless the number of cells
    // to refine is 0)

minRefinementCells 0;

    // Number of buffer layers between different levels.
    // 1 means normal 2:1 refinement restriction, larger means slower
    // refinement.

nCellsBetweenLevels 3;

    // Explicit feature edge refinement
    // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    // Specifies a level for any cell intersected by its edges.
    // This is a featureEdgeMesh, read from constant/triSurface for now.
    features
(
);
    // Surface based refinement
    // ~~~~~~~~~~~~~~~~~~~~~~~~

    // Specifies two levels for every surface. The first is the minimum level,
    // every cell intersecting a surface gets refined up to the minimum level.
    // The second level is the maximum level. Cells that 'see' multiple
    // intersections where the intersections make an
    // angle > resolveFeatureAngle get refined up to the maximum level.

    refinementSurfaces
    {
        extAir
        {
            // Surface-wise min and max refinement level
            level (1 2);

            faceZone extAir;
            cellZone extAir;
            cellZoneInside inside;
        }

        midSolid
        {
            // Surface-wise min and max refinement level
            level (2 3);
```

```
            faceZone midSolid;
            cellZone midSolid;
            cellZoneInside inside;
        }
    }

    // Resolve sharp angles
    resolveFeatureAngle 30;


    // Region-wise refinement
    // ~~~~~~~~~~~~~~~~~~~~~~

    // Specifies refinement level for cells in relation to a surface. One of
    // three modes
    // - distance. 'levels' specifies per distance to the surface the
    //   wanted refinement level. The distances need to be specified in
    //   descending order.
    // - inside. 'levels' is only one entry and only the level is used. All
    //   cells inside the surface get refined up to the level. The surface
    //   needs to be closed for this to be possible.
    // - outside. Same but cells outside.

    refinementRegions
    {
    }


    // Mesh selection
    // ~~~~~~~~~~~~~~

    // After refinement patches get added for all refinementSurfaces and
    // all cells intersecting the surfaces get put into these patches. The
    // section reachable from the locationInMesh is kept.
    // NOTE: This point should never be on a face, always inside a cell, even
    // after refinement.

    locationInMesh (0 0 0);


    // Whether any faceZones (as specified in the refinementSurfaces)
    // are only on the boundary of corresponding cellZones or also allow
    // free-standing zone faces. Not used if there are no faceZones.

    allowFreeStandingZoneFaces false;
}

// Settings for the snapping.

snapControls
{
    //- Number of patch smoothing iterations before finding correspondence
    //  to surface
    nSmoothPatch 3;

    //- Relative distance for points to be attracted by surface feature point
    //  or edge. True distance is this factor times local
    //  maximum edge length.
    tolerance 2.0;
```

```
    //- Number of mesh displacement relaxation iterations.
    nSolveIter 100;

    //- Maximum number of snapping relaxation iterations. Should stop
    //  before upon reaching a correct mesh.
    nRelaxIter 5;

    //- Highly experimental and wip: number of feature edge snapping
    //  iterations. Leave out altogether to disable.
    //  Of limited use in this case since faceZone faces not handled.
    nFeatureSnapIter 4;

        //- Detect (geometric only) features by sampling the surface
        //  (default=false).
        implicitFeatureSnap false;

        //- Use castellatedMeshControls::features (default = true)
        explicitFeatureSnap true;

        //- Detect points on multiple surfaces (only for explicitFeatureSnap)
        multiRegionFeatureSnap true;
}

// Settings for the layer addition.
addLayersControls
{
    relativeSizes true;

    // Per final patch (so not geometry!) the layer information
    layers
    {
    }

    // Expansion factor for layer mesh
    expansionRatio 1.3;

    // Wanted thickness of final added cell layer. If multiple layers
    // is the thickness of the layer furthest away from the wall.
    // Relative to undistorted size of cell outside layer.
    finalLayerThickness 1;

    // Minimum thickness of cell layer. If for any reason layer
    // cannot be above minThickness do not add layer.
    // Relative to undistorted size of cell outside layer.
    minThickness 0.1;

    // If points get not extruded do nGrow layers of connected faces that are
    // also not grown. This helps convergence of the layer addition process
    // close to features.
    // Note: changed(corrected) w.r.t 17x! (didn't do anything in 17x)
    nGrow 0;

    // Advanced settings

    // When not to extrude surface. 0 is flat surface, 90 is when two faces
    // are perpendicular
    featureAngle 30;

    // Maximum number of snapping relaxation iterations. Should stop
    // before upon reaching a correct mesh.
    nRelaxIter 3;
```

```
    // Number of smoothing iterations of surface normals
    nSmoothSurfaceNormals 1;

    // Number of smoothing iterations of interior mesh movement direction
    nSmoothNormals 3;

    // Smooth layer thickness over surface patches
    nSmoothThickness 2;

    // Stop layer growth on highly warped cells
    maxFaceThicknessRatio 0.5;

    // Reduce layer growth where ratio thickness to medial
    // distance is large
    maxThicknessToMedialRatio 1;

    // Angle used to pick up medial axis points
    // Note: changed(corrected) w.r.t 17x! 90 degrees corresponds to 130 in 17x.
    minMedianAxisAngle 90;

    // Create buffer region for new layer terminations
    nBufferCellsNoExtrude 0;

    // Overall max number of layer addition iterations. The mesher will exit
    // if it reaches this number of iterations; possibly with an illegal
    // mesh.
    nLayerIter 50;
}

// Generic mesh quality settings. At any undoable phase these determine
// where to undo.
meshQualityControls
{
    #include "meshQualityDict"

    // Advanced

    //- Number of error distribution iterations
    nSmoothScale 4;
    //- amount to scale back displacement at error points
    errorReduction 0.75;
}
// Advanced

// Merge tolerance. Is fraction of overall bounding box of initial mesh.
// Note: the write tolerance needs to be higher than this.
mergeTolerance 1e-6;
```

Como se dijo al inicio de este trabajo, OpenFOAM no trabaja con mallas 2D. Por lo que es necesario otorgar de un espesor significativo a la geometría para poder simular. La herramienta para ello es `extrudeMesh` (`extrudeMeshDict`). Esta utilidad otorga de espesor a los hexaedros que ha creado `blockMesh`, dándole a la simulación un carácter tridimensional. A continuación, se muestra el contenido de dicho archivo.

```
// What to extrude:
//      patch   : from patch of another case ('sourceCase')
//      mesh    : as above but with original case included
//      surface : from externally read surface

constructFrom patch;
sourceCase ".";
sourcePatches (front);

// If construct from patch: patch to use for back (can be same as sourcePatch)
exposedPatchName back;

// Flip surface normals before usage. Valid only for extrude from surface or
// patch.
flipNormals false;

//- Linear extrusion in point-normal direction
extrudeModel        linearNormal;

nLayers             1;

expansionRatio      1.0;

linearNormalCoeffs
{
    thickness       0.02;
}

// Do front and back need to be merged? Usually only makes sense for 360
// degree wedges.
mergeFaces false;   //true;

// Merge small edges. Fraction of bounding box.
mergeTol 0;
```

Por lo tanto, una vez explicado los diccionarios que se encuentran en la carpeta raíz del caso, se está en disposición de discretizar el dominio computacional y establecer los preparativos necesarios para lanzar la simulación. El paso siguiente es la preparación de malla. Para ello se aplicarán los siguientes diccionarios: `blockMesh` seguido de `extrudeMesh` y por último `snappyHexMesh`. Con esto obtenemos unas características de malla como se muestra en la ilustración 3, 4 y 5, respectivamente.

Destacar que la parte sólida y la fluida todavía no se han separado aún, todo es un único dominio computacional. Si se ejecuta ahora la herramienta `splitMeshRegions` en la consola de la siguiente forma:

```
splitMeshRegions -cellZones -overwrite
```

Esto divide el dominio en tantos subdominios como `cellZones` existan. Por lo que ahora se tendrá correctamente diferenciado la parte fluida de la sólida como muestra la ilustración 6. Una vez hecho, en la carpeta `polyMesh` se creará un fichero con el nombre `cellZones`. Aquí se pueden encontrar las dos regiones creadas:

- `midSolid`: con las celdas del dominio sólido.
- `extAir`: con las celdas del dominio fluido.

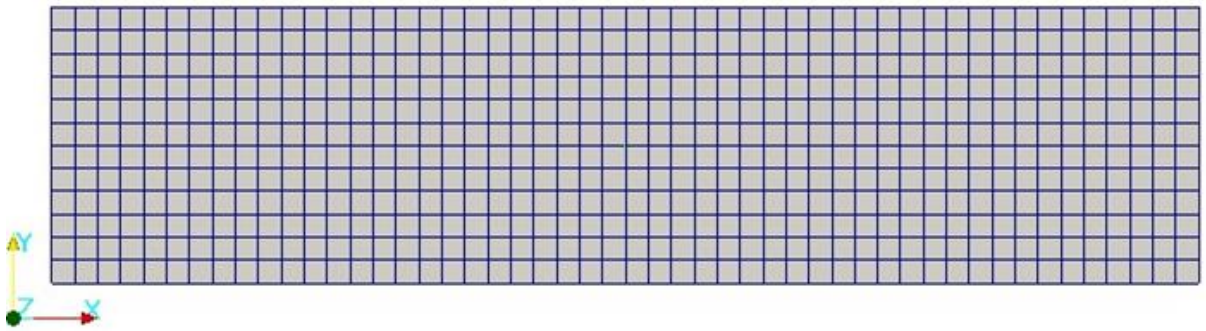

Figura 3. Malla del dominio computacional usando blockMesh.

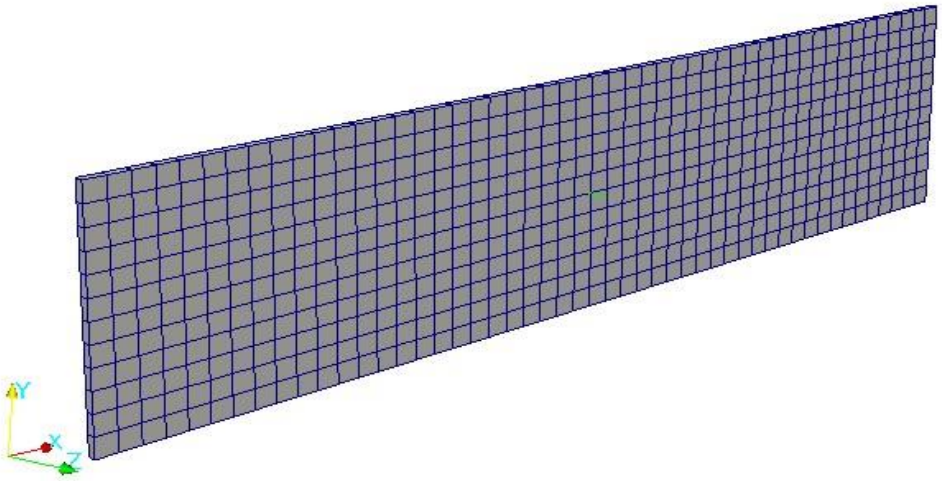

Figura 4. Malla del dominio computacional usando extrudeMesh.

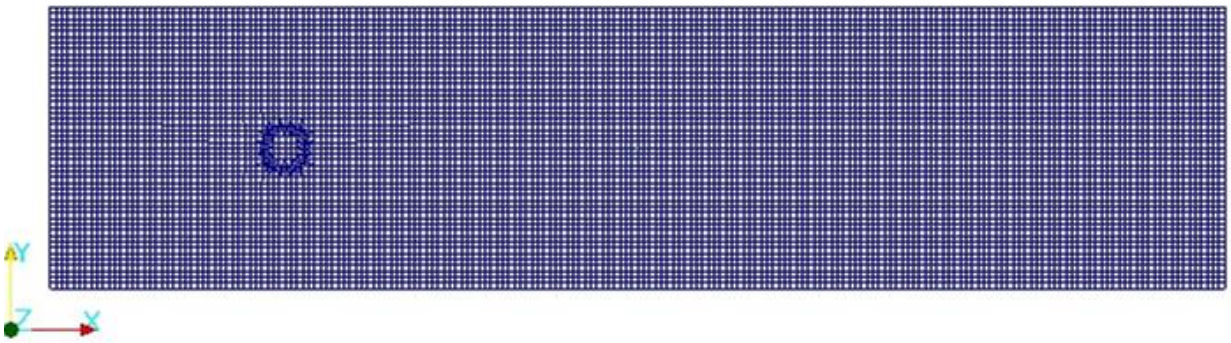

Figura 5. Malla del dominio computacional usando snappyHexMesh.

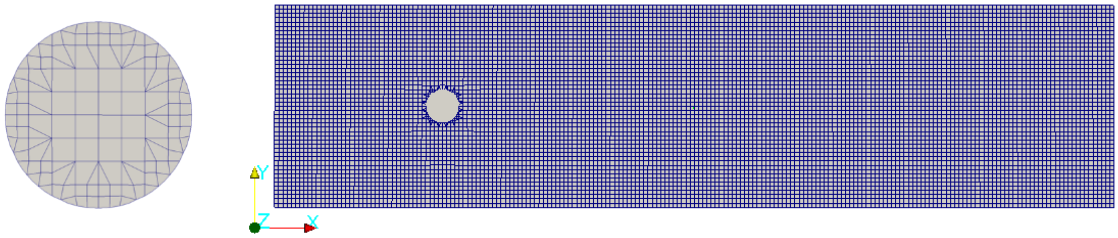

Figura 6. Malla del dominio computacional usando slipMeshRegions.

Finalmente, un objetivo adicional para los cuatros casos que se plantean en este documento, es la creación de un *script* o conjunto de instrucciones que hagan todo el proceso de mallado y lanzamiento de la simulación de manera automática.

Es decir, eliminar la introducción secuencial de comandos y/o instrucciones para lanzar la simulación desde el terminal del sistema operativo. Esto consigue reducir drásticamente el tiempo de lanzamiento de la simulación y ser una solución excelente para procesos de optimización topológica. A través de un archivo ejecutable de texto en la carpeta donde se encuentran los ficheros necesarios para la simulación, se consigue monitorizar y automatizar estas tareas haciendo que la experiencia del usuario en OpenFOAM sea más satisfactoria. A continuación, se muestra el contenido del archivo `run`:

```
rm -rf 0 constant system
rm *
cp -r ficherosNecesarios/0 .
cp -r ficherosNecesarios/constant .
cp -r ficherosNecesarios/system .

blockMesh > log.blockMesh
snappyHexMesh -overwrite > log.snappyHexMesh
extrudeMesh > log.extrudeMesh
splitMeshRegions -cellZones -overwrite > log.splitMesh

# remove fluid fields from solid regions (important for post-processing)
```

```
for i in midSolid
do
   rm -f 0*/$i/{mut,alphat,epsilon,k,U,p_rgh}
done

for i in midSolid extAir
do
   changeDictionary -region $i >& log.changeDictionary.$i
done

#-- Run on single processor
pyFoamPlotRunner.py chtMultiRegionFoam > log.chtMultiRegionFoam

paraFoam -touchAll
```

El código mostrado arriba es muy intuitivo. Incluso sin muchas nociones de programación en C++ se puede intuir el funcionamiento del mismo. Solo es necesario un gestor de texto, en este caso se ha utilizado *Gedit*. Una vez creado el código es necesario dotar al archivo de los permisos de ejecutable, esto se puede conseguir escribiendo en el terminal:

```
chmod +x run
```

En las primeras líneas lo que se pretende es eliminar posibles archivos residuales que se hayan generado por lanzamiento de simulaciones fallidas, previniendo de futuros problemas a la hora de sobrescribir archivos en las carpetas involucradas con la simulación. Las siguientes instrucciones se basan en ejecutar los diccionarios encargados de la malla: `blockMesh`, `snappyHexMesh`, `extrudeMesh` y `splitMeshRegions`. Los bucles `for` para ambas regiones están encargados de la eliminación y gestión de propiedades para el post-procesamiento. PyFoamPlotRunner.py es una aplicación escrita en Python que ayuda a representar los residuales de la simulación durante todo el proceso.

Esta herramienta es de gran utilidad porque ayuda al usuario de forma gráfica a comprobar el estado de la simulación en todo momento, ahorrando tiempo cancelando la simulación en el caso de apreciar una divergencia en los términos residuales. Por último, desde el terminal y en el directorio raíz del caso se ejecuta el archivo `run` en el terminal mediante:

```
./run
```

3.5 Resultados

Una vez lanzada con éxito la simulación, gracias a la herramienta de *Pyfoam* nombrada anteriormente, se pueden representar los residuales de la simulación. Esto ayuda al usuario menos experto a determinar gráficamente si, la solución de la simulación ha alcanzado niveles de convergencia. Como se especificó en apartados anteriores, el límite de convergencia para todas variables que intervinieron en la simulación se estableció en 1e-6 y un máximo de 1200 iteraciones. Se puede apreciar que la simulación finaliza por la condición límite de número de iteraciones, en vez de por el límite de convergencia. Esto permite llegar a las siguientes premisas; se necesita más tiempo para alcanzar convergencia debido a las condiciones iniciales o por la naturaleza transitoria de la simulación no se podrá llegar nunca a un estado final. De todas maneras, no es el objetivo de este documento el análisis exhaustivo de los criterios de convergencia, comportamiento de los residuales o niveles de precisión alcanzados. Es por ello, que no se profundizará más allá de los resultados obtenidos por la herramienta de post-procesamiento *Paraview.* Esta puede ser invocada desde el terminal sencillamente escribiendo:

```
paraview
```

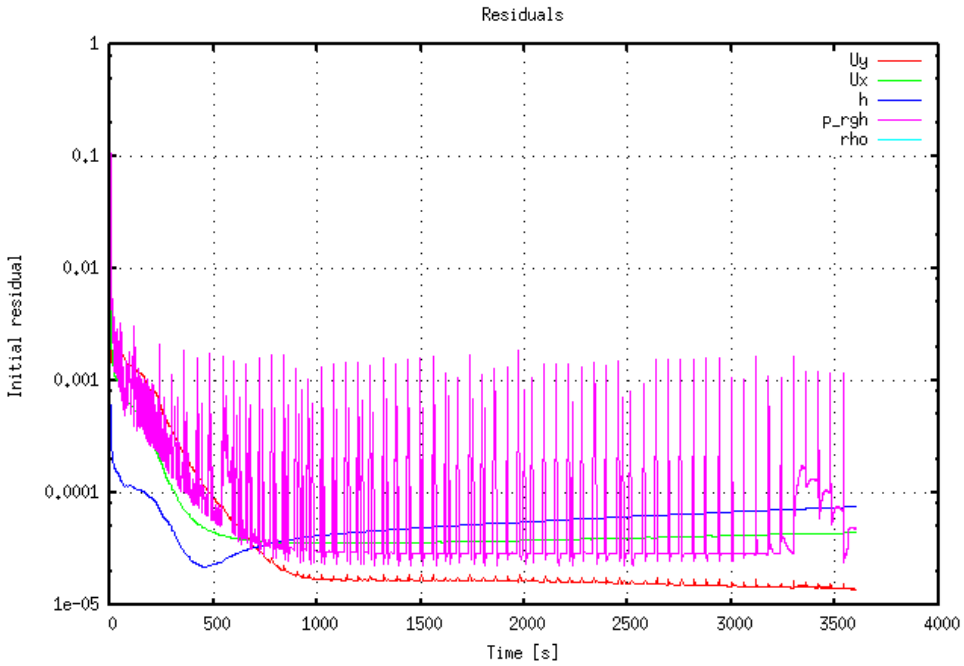

Figura 7. CASO A – Residuales Cilindro.

Así, la figura 7 presenta el comportamiento de estos residuales en dicha simulación. Las siguientes ilustraciones mostradas a continuación presentan de forma gráfica los resultados obtenidos y se puede observar al menos que, la aplicación otorga soluciones lógicas dentro de los parámetros establecidos. Se han sacado instantáneas de los resultados en el último instante de tiempo de la simulación.

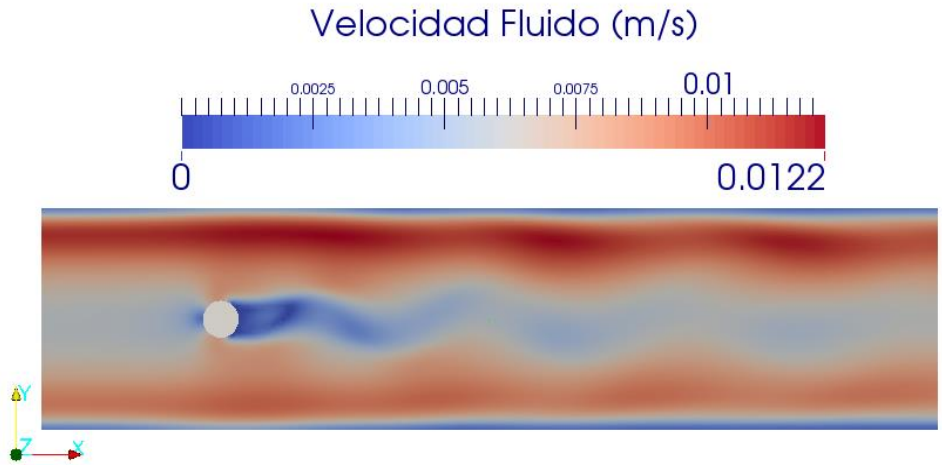

Figura 8. CASO A - Velocidad del Fluido.

Además, destacar una de las características más relevantes del diccionario utilizado `snappyHexMeshDict`. Este te permite cambiar de geometría en la región `midSolid` muy fácilmente e incluso introducir una desde un software CAD externo, siempre que sea formato .stl o .obj.

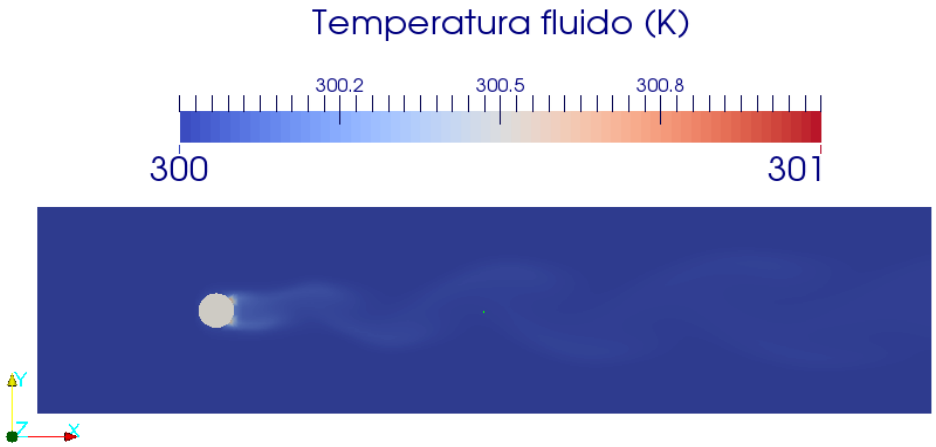

Figura 9. CASO A - Temperatura del Fluido.

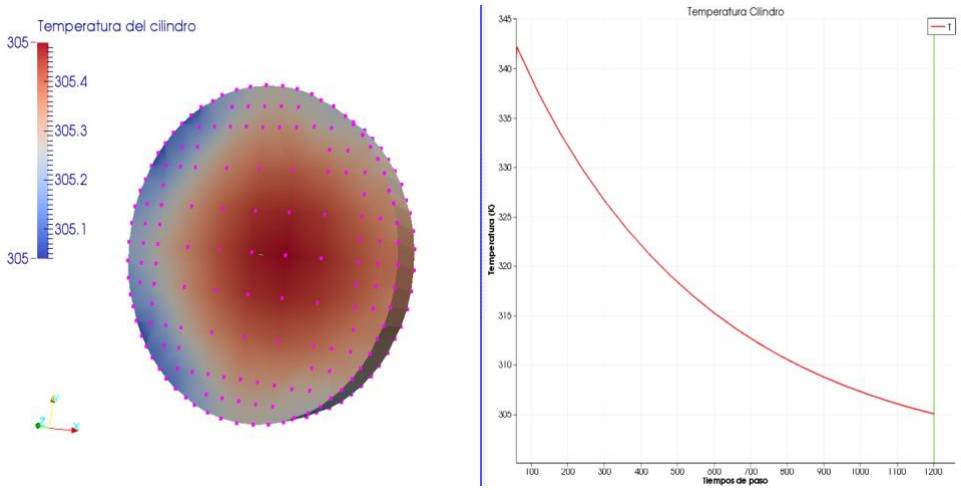

Figura 10. CASO A – Temperatura cilindro respecto del tiempo.

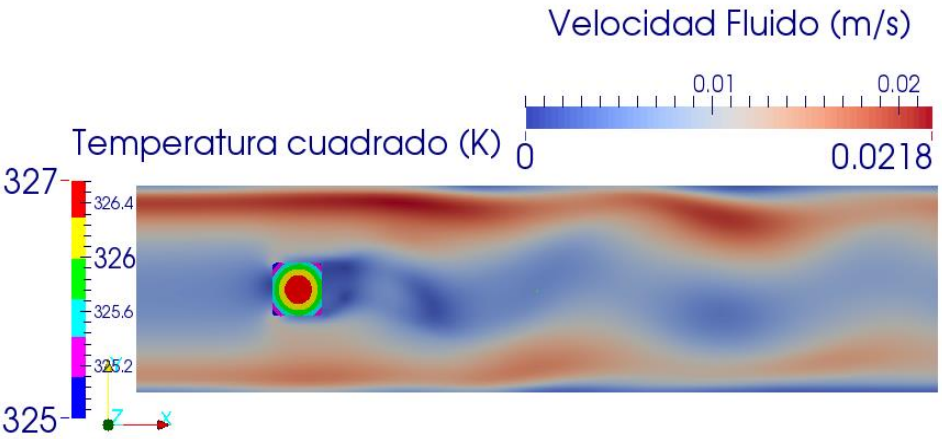

Figura 11. CASO A – Resultados del sólido cuadrado.

Esta increíble ventaja produce una reducción drástica de los tiempos de preparación de simulaciones, muy beneficioso en términos económicos para procesos ingenieriles como es la optimización de geometría. Mediante esta metodología se consigue un ahorro de tiempo del 50%, de forma que se pueden lazar múltiples simulaciones secuenciales reduciendo el tiempo total del proceso de desarrollo. A modo de ejemplo, en los siguientes ejemplos se ha cambiado la geometría de la región computacional `midSolid`; primero por un cuadrado (pudiendo hacerse desde el diccionario `snappyHexMeshDict`) y segundo por un perfil Naca.

Se puede observar claramente como al aumentar la superficie caliente en la geometría, esta transfiere más energía al fluido y, por consiguiente, aumenta su velocidad de transferencia.

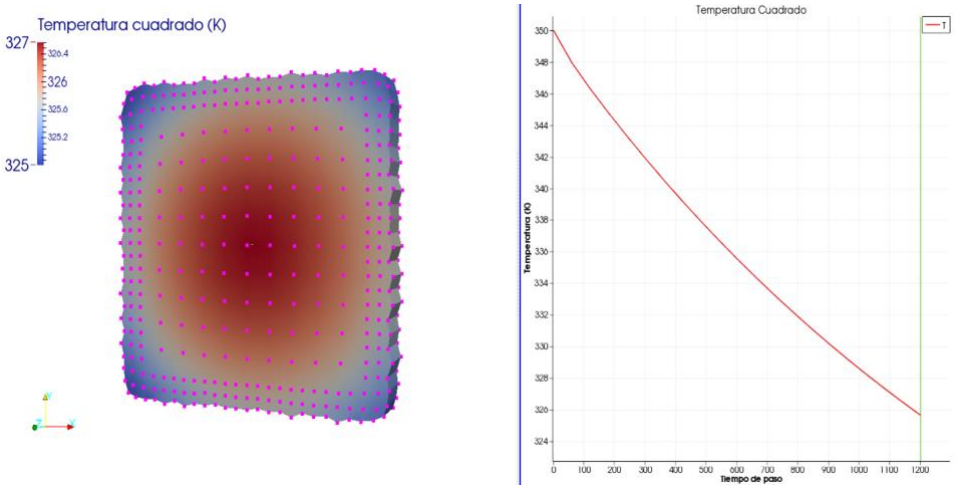

Figura 12. CASO A – Temperatura cuadrado respecto del tiempo.

Por ejemplo, esto también se puede extrapolar a un caso de optimización geométrica de un perfil NACA. Aquí se evalúan diferentes propiedades geométricas como el ángulo de ataque, longitud, grosor, etc. Realizando múltiples simulaciones y reduciendo el tiempo de configuración entre casos se llega a una validación de resultados reduciendo el tiempo considerablemente.

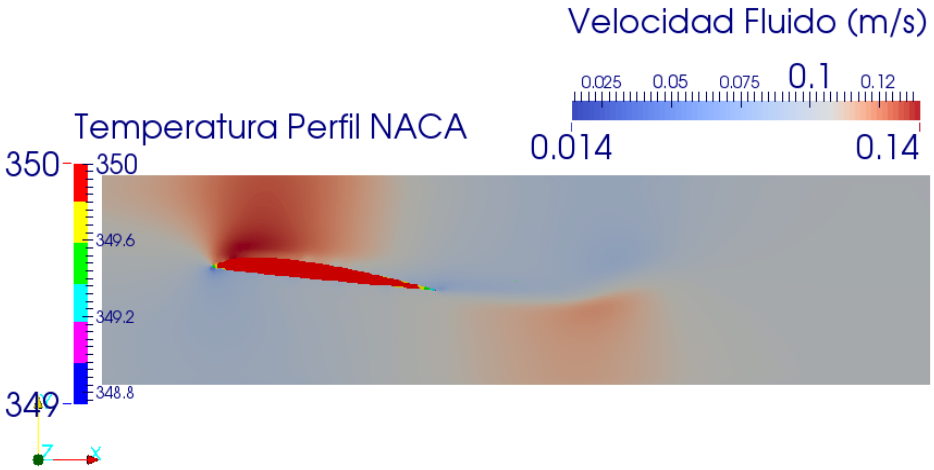

Figura 13. CASO A – Perfil NACA.

3.6 Conclusiones

El código numérico expuesto en este apartado ha servido para obtener una visión global de cómo se puede simular un problema que involucra transferencia de calor con dominios computacionales totalmente distintos. En conjunto se puede considerar

como una posible metodología para la optimización de procesos topológicos, permitiendo así evaluar diferentes geometrías para un mismo problema minimizando drásticamente el tiempo de todo el proceso.

Además, las simulaciones en el mundo industrial cada vez más están cobrando relevancia debido al impacto económico que conllevan. Mediante los resultados obtenidos se puede llegar a detectar errores o problemas mucho antes incluso que la fase de prototipado, reduciendo el proceso de validación y el coste económico significativamente. Todo esto se traduce, en líneas generales, en una mayor eficiencia en todo el proceso de producción, repercutiendo directamente en la competitividad de la empresa.

4 CASO B: Incorporación de calentamiento de sólido por efecto Joule y radiación.

4.1 Introducción y contexto teórico

En el caso anterior se ha detallado el proceso de simulación de un cilindro sólido sumergido en un fluido. Mediante condiciones de contorno iniciales, se postuló que la región solida se encontraba a diferente temperatura que la fluida. Esto producía una transferencia de calor del elemento más caliente al más frio, pudiéndose simular todo este proceso mediante la aplicación de `chtMultiRegionFoam`. Aunque a priori en este caso se va a reutilizar la geometría, mallado e incluso el *solver* del caso anterior, lo que se pretende en este apartado es añadir al sólido el efecto Joule (resistencia eléctrica) y considerar el efecto de radiación en un segundo caso adicional. Es decir, ahora se quiere que la región sólida genere calor por efecto Joule, produciéndose una transferencia de energía desde el cilindro al fluido.

Por lo tanto, se deberán editar el *solver* o código numérico utilizado con anterioridad y añadir las ecuaciones características relacionadas con la densidad de corriente eléctrica y el campo eléctrico. Para ello, se deberán de modificar una serie de parámetros y añadir líneas de código que hagan referencia a este tipo de fenómenos físicos antes de lanzar la simulación.

4.1.1 Campo eléctrico

El campo eléctrico en cualquier punto del espacio está íntimamente relacionado con la fuerza eléctrica que se ejerce sobre una carga colocada en dicho punto. Según la definición de campo eléctrico se tiene que:

$$\vec{F}(x,y) = q \cdot \vec{E}(x,y) \tag{21}$$

siendo:

- $\vec{F}(x,y)$ fuerza ejercida por el campo eléctrico.
- q es el valor escalar de la carga.
- $\vec{E}(x,y)$ es el campo eléctrico existente.

Por otra parte, el potencial eléctrico, Ve, está también relacionado con el trabajo que se debe realizar para llevar una carga desde un punto a otro. Es decir, el cambio

de potencial entre el punto de inicio hasta el final. Esto se puede expresar de la siguiente manera:

$$dVe = \frac{-dW}{q} = \frac{-1}{q}\vec{F}(x,y) \cdot d\vec{l} = -\vec{E} \cdot d\vec{l} \tag{22}$$

expresando las componentes del campo eléctrico en función del potencial obtenemos:

$$E_x = \frac{-dVe}{dx}, E_y = \frac{-dVe}{dy}, E_y = \frac{-dVe}{dy} \tag{23}$$

escrito de otra manera:

$$\vec{E} = -\nabla \cdot Ve \tag{24}$$

La propiedad interesante que debe incluirse es la densidad de corriente eléctrica, a través de la siguiente expresión:

$$\vec{J} = -\sigma(T)\vec{E} = -\sigma(T)\nabla Ve \tag{25}$$

siendo:

- $\sigma(T)$es la conductividad eléctrica y es una función de la temperatura.

Si se centran los esfuerzos en las consecuencias energéticas de la Ley de Ohm relacionadas con la disipación y el efecto Joule, se puede postular que la cantidad de energía calorífica producida por una corriente eléctrica es directamente proporcional al cuadrado de la intensidad de corriente, la resistencia que opone el conductor y al tiempo que circula sobre el mismo. Con ayuda de la Ley de Ohm:

$$Ve = R \cdot I \tag{26}$$

se obtiene la Ley de Joule:

$$E = I^2 \cdot R \cdot t \tag{27}$$

Siendo E la energía disipada por unidad de tiempo.

De esta manera, la potencia disipada es proporcional a la densidad de corriente y el campo eléctrico existente:

$$P = I^2 \cdot R = I \cdot V = \frac{1}{R} Ve^2 = J \cdot E = \sigma(T)(\nabla Ve)^2 \quad (28)$$

4.1.2 Radiación

En el caso A se trabajó con la transferencia de calor por conducción y convección, como procesos térmicos que interaccionan con un material para dicho intercambio de energía. En cambio, la transferencia de energía por radiación no requiere material alguno y es de vital importancia a grandes temperaturas. Por eso se asocia la emisión térmica de radiación con las condiciones térmicas de excitación de la materia. Es ampliamente conocido que toda materia emite radiación. Existen dos teorías de transporte de energía por radiación: aquella que considera la propagación como un conjunto de partículas llamadas fotones o cuantos. Y alternativamente, la que se utilizará en este proyecto, la teoría que considera la radiación como la propagación de ondas electromagnéticas. A continuación, se detallan brevemente las propiedades relevantes dentro de la teoría de radiación:

- Intensidad de radiación: si se considera la variación de emisión la cual pasa dedA_1a través de dA_ncomo muestra la ilustración 14. La dirección puede ser expresada en términos de ángulos cenital y acimutal, θy Φrespectivamente. Si además, suponemos que dicha superficie emite una cantidad de calor o intensidad de radiación (W/m^2) para una cierta longitud de onda:

$$dq_\lambda = I_{\lambda,e}(\lambda, \theta, \Phi) dA \cos(\theta) d\omega \quad (29)$$

por lo tanto, si se integra la expresión se obtiene el calor irradiado:

$$dq_\lambda = \int_0^{2\pi} \int_0^{\pi/2} I_{\lambda,e}(\lambda, \theta, \Phi) \cos(\theta) \sin(\theta) d\theta d\Phi \quad (30)$$

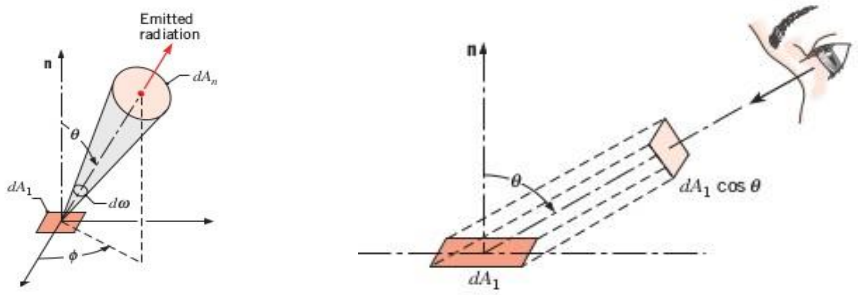

Figura 14. Emisión de radiación a través de diferenciales de área. Referencia [3].

- Emisión: La potencia de emisión espectral, $E_\lambda(W/m^2 \cdot \mu m)$ es la tasa a la cual la radiación de una determinada longitud de ondaλes emitida en todas las direcciones desde la superficie por unidad de longitud de onda y superficie. A través de:

$$E_\lambda(\lambda) = \int_0^{2\pi} \int_0^{\pi/2} E_\lambda\,(\lambda) cos(\theta) sin(\theta) d\theta d\Phi \tag{31}$$

La potencia de emisión total (W/m^2) es el ratio el cual radiación es emitida por unidad de are en todas las posibles longitudes de onda y en todas las posibles direcciones.

$$E = \int_0^{\infty} E_\lambda(\lambda) d\lambda \tag{32}$$

o

$$E = \int_0^{\infty} \int_0^{2\pi} \int_0^{\pi/2} I_{\lambda,e}\,(\lambda, \theta, \Phi) cos(\theta) sin(\theta) d\theta d\Phi d\lambda \tag{33}$$

En el caso en el que la irradiación es difusa, es decir, que no depende de la dirección, la integral mostrada anteriormente se puede simplificar y da lugar a:

$$E = \pi I_e \tag{34}$$

- Irradiación: esta radiación se origina por la emisión y reflexión sobre superficies que tendrán distribuciones espectrales y direccionales determinadas por la intensidad de radiación incidente. Es decir, la variación por la cual la radiación,

a una longitud de onda, incide sobre una superficie por unidad de área y longitud de onda. La ecuación característica es:

$$G_\lambda(\lambda) = \int_0^{2\pi} \int_0^{\pi/2} I_{\lambda,i}\,(\lambda) cos(\theta) sin(\theta) d\theta d\Phi \tag{35}$$

De igual forma, se define la irradiación total como:

$$G = \int_0^{\infty} G_\lambda\,(\lambda) d\lambda \tag{36}$$

La irradiación incidente sobre una superficie puede descomponerse en tres partes diferenciables; radiación reflejada, transmitida y absorbida. De forma que estas tres reflejen la radiación incidente total sobre un área.

$$G = G_r + G_t + G_a \tag{37}$$

Radiación reflejada: esta radiación es aquella parte de la irradiación total que se refleja. Esto se detalla a través de una propiedad de superficie llamada reflectividad espectral y direccional. Su coeficiente se define como:

$$\rho_{\lambda,\theta} = \frac{I_{\lambda,i,ref(\lambda,\theta,\Phi)}}{I_{\lambda,i(\lambda,\theta,\Phi)}} \tag{38}$$

Radiación transmitida: Es la parte de la radiación total que atraviesa la superficie. Esto se mide a través de una propiedad conocida como la transmisividad. Su expresión es la siguiente:

$$\tau_{\lambda,\theta} = \frac{I_{\lambda,i,trans(\lambda,\theta,\Phi)}}{I_{\lambda,i(\lambda,\theta,\Phi)}} \tag{39}$$

Radiación absorbida: es la parte absorbida por el medio, y cuyo coeficiente es:

$$\alpha_{\lambda,\theta} = \frac{I_{\lambda,i,ab(\lambda,\theta,\Phi)}}{I_{\lambda,i(\lambda,\theta,\Phi)}} \tag{40}$$

Radiosidad: se define como toda la energía radiante que sale de una superficie. Esta radiación incluye la parte de irradiación reflejada al igual que la emisión directa. La radiosidad espectral $J_\lambda(W/m^2 \cdot \mu m)$ representa la variación la cual la radiación con una determinada longitud de onda sale por unidad de superficie y por unidad de longitud de onda.

$$J_\lambda(\lambda) = \int_0^{2\pi} \int_0^{\pi/2} I_{\lambda,e+r}(\lambda, \theta, \Phi) cos(\theta) sin(\theta) d\theta d\Phi \quad (41)$$

Por lo tanto, la radiosidad total $J(W/m^2)$ asociada con todo es espectro es:

$$J = \int_0^\infty J_\lambda(\lambda) d\lambda \quad (42)$$

- Radiación de cuerpo negro: con el fin de evaluar las características descritas anteriormente sobre una superficie real opaca, se debe presentar el concepto o premisas de un cuerpo negro.
    - Un cuerpo negro absorbe toda la radiación incidente, con cualquier longitud de onda o dirección.
    - Para una temperatura y longitud de onda específica, ninguna superficie puede emitir más energía que un cuerpo negro.
    - Aunque la radiación emitida por un cuerpo negro es una función de la longitud de onda y la temperatura, esta es independiente de la dirección. Por lo tanto, el cuerpo negro es un emisor difuso.

La intensidad espectral de un cuerpo negro está determinada por la distribución de Plank, su expresión característica es la siguiente:

$$I_{\lambda,b}(\lambda, T) = \frac{2hc_0^2}{\lambda^5[exp(hc_0/\lambda k_B T) - 1]} \quad (43)$$

Donde $h = 6.626x10^{-34} J \cdot s$ y $k_B = 1.381x10^{-23} J/K$ son las constantes universales de Plank. También, $c_0 = 2.998x10^8 m/s$ es la velocidad de la luz en vacío y $T(K)$ es la temperatura absoluta del cuerpo negro en Kelvin.

Además, como el cuerpo negro es considerado un emisor difuso, su potencia de emisión espectral es:

$$E_{\lambda,b}(\lambda,T) = \pi I_{\lambda,b}(\lambda,T) = \frac{C_1}{\lambda^5[exp(C_2/\lambda T)-1]} \quad (44)$$

donde $C_1 = 2\pi hc_0^2 = 3.742x10^8 W \cdot \mu m^4/m^2$ y $C_2 = hc_0/k_B = 1.439x10^4 \mu m \cdot K$ son la primera y la segunda constante de radiación, respectivamente. Esta ecuación, es conocida como la distribución de Plank o Ley de Plank, está representada en la ilustración 15. Si se utiliza la distribución de Plank para la obtención de la emisión total de un cuerpo negro, se obtiene:

$$E_b = \int_0^\infty \frac{C_1}{\lambda^5[exp(C_2/\lambda T)-1]} d\lambda = \sigma T^4 \quad (45)$$

donde aquí σes la constante de Stefan-Boltzman.

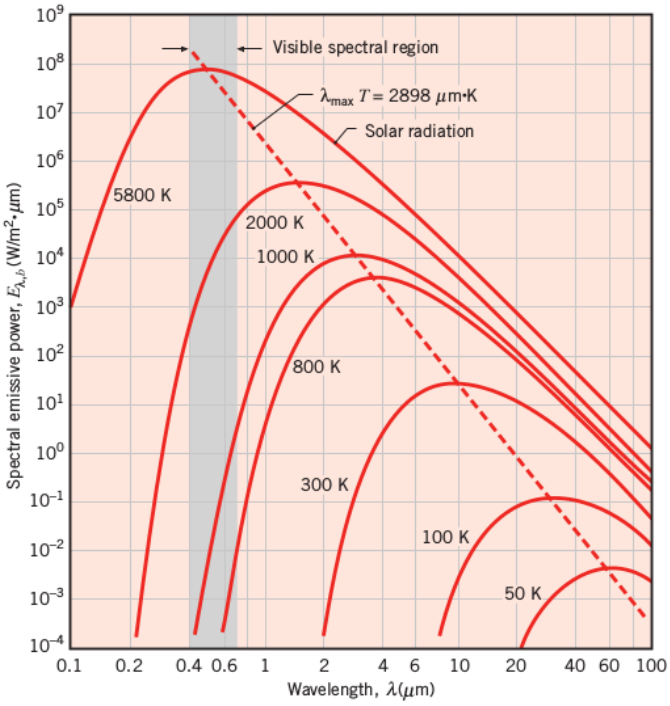

Figura 15. Potencia de emisión espectral de un cuerpo negro. Referencia [3].

4.2 Descripción del *solver* existente: joulechtMultiRegionFoam

Al inicio del documento se detalló el carácter de OpenFOAM de promover la reutilización de códigos numéricos. En este trabajo se ha seguido dicha metodología, y por tanto, se reutilizarán la mayoría de archivos y carpetas del Caso A. La única salvedad es que se añadirá el efecto térmico provocado por el efecto Joule y después el efecto de radiación.

El *solver* existente, como se ha dicho al inicio del apartado 4, será `chtMultiRegionFoam`. Esto nos obligará a modificar el código, adaptando las condiciones detalladas en los apartados 4.1.1 y 4.1.2. Por lo tanto, llamaremos al *solver* `joulechtMultiRegionFoam`.

4.3 Modificaciones incluidas en el *solver*

Primero se debe crear una copia de `chtMultiRegionFoam` en la capeta `$WM_PROJECT_USER_DIR/solvers/` de OpenFOAM, eliminar la carpeta `chtMultiRegionSimpleFoam` ya este archivo está relacionado con el estacionario y no el transitorio. Finalmente, se debe editar `Make/file:`

```
EXE = $(FOAM_USER_APPBIN)/jouleChtMultiRegionFoam
```

Se debe compilar (`./Allwmake`) y observar que no se obtiene ningún error. Por lo tanto, los objetivos que se deben perseguir a la hora de abordar este problema son los siguientes:

- Modificar la ecuación de la entalpía (fase sólida) y de la energía total (fase fluida) para acoplar el efecto Joule a partir del campo electrostático descrito en la sección 4.1.1.
- Definir las nuevas variables que aparecen en dichas ecuaciones.
- Modificar el tutorial `fullCylinderMultiRegionScripting2.3.1` presentado en el caso A para incluir las nuevas variables a resolver. Esto se detallará en el apartado 4.4.

Lo primero que se va hacer es crear un nuevo fichero con la ecuación de Laplace, se llamará `Veqn.H` y se añadirá tanto a la capeta fluid como a la `solid` del directorio de *solver*.

```
{
for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
{
solve
(
fvm::laplacian(sigma, Vel)
);
}
Info<< "Min/max V:" << min(Vel) << ' '
<< max(Vel) << endl;
}
```

Ahora se modifican la ecuación de la entalpía tanto para fluido (`EEqn.H`) como para sólido (`solveSolid.H`) para acoplar ambos problemas. Para el dominio sólido entonces:

```
fvOptions(rho, h)

//acoplamiento de los dos problemas
+ ((fvc::grad(Vel)) & (fvc::grad(Vel)))*sigma
        );

hEqn().relax();
```

Para el dominio fluido de igual manera en `EEqn.H` se añade:

```
rad.Sh(thermo) + fvOptions(rho, he)
+ ((fvc::grad(Vel)) & (fvc::grad(Vel)))*sigma
  );
EEqn.relax();
```

Añadimos ahora los nuevos campos escalares para el fluido `createFluidFields.H`:

```
PtrList<volScalarField> VFluid(fluidRegions.size());
PtrList<volScalarField> sigmaFluid(fluidRegions.size());
```

```
Info<< "Adding to VFluid\n" << endl;
        VFluid.set
        (
            i,
            new volScalarField
            (
                IOobject
                (
                 "Vel",
                  runTime.timeName(),
```

```
Info<<"Adding to sigmaFluid\n" << endl;
        sigmaFluid.set
        (
            i,
            new volScalarField
            (
                IOobject
                (
                 "sigma",
                  runTime.timeName(),
```

```
                fluidRegions[i],
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
            ),
            fluidRegions[i]
        )
    );
```

```
                fluidRegions[i],
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
            ),
            fluidRegions[i]
        )
    );
```

Los campos escalares para el sólido `createSolidFields.H`:

```
PtrList<volScalarField> VSolid(solidRegions.size());
PtrList<volScalarField> sigmaSolid(solidRegions.size());
```

```
  Info<< " Adding to VSolid\n" << endl;
      VSolid.set
      (
          i,
          new volScalarField
          (
              IOobject
              (
               "Vel",
                runTime.timeName(),
                fluidRegions[i],
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
              ),
               solidRegions[i]
          )
      );
```

```
Info<<"Adding to sigmaSolid\n" << endl;
      sigmaSolid.set
      (
          i,
          new volScalarField
          (
              IOobject
              (
               "sigma",
                runTime.timeName(),
                fluidRegions[i],
                IOobject::MUST_READ,
                IOobject::AUTO_WRITE
              ),
               solidRegions[i]
          )
      );
```

Transferimos ahora las soluciones de las diferentes regiones al dominio. Añadimos al final del documento `setRegionSolidFields.H`:

```
volScalarField& Vel = VSolid[i];
volScalarField& sigma = sigmaSolid[i];
```

De forma análoga con `setRegionFluidFields.H`:

```
volScalarField& Vel = VFluid[i];
volScalarField& sigma = sigmaFluid[i];
```

Finalmente, solo queda añadir el archivo que contiene el campo eléctrico `Veqn.H` en los códigos principales de cada dominio computacional. Como ya se añadió En `solveFluid.H`:

```
#include "UEqn.H"
#include "EEqn.H"
#include "VEqn.H"

// --- PISO loop
```

```
 for (int corr=0; corr<nCorr; corr++)
```

Y en solveSolid.H:

```
fvOptions.correct(h);
    }
       #include "VEqn.H"
    }
thermo.correct();
```

Como último paso se debe de compilar (`./Allwmake`) el *solver* para comprobar que se ha realizado todo correctamente.

4.4 Preparación del caso (efecto Joule)

Lo primero es establecer unas condiciones de contorno iguales que el caso A. Añadiendo un valor para la diferencia de potencia eléctrico aleatorio de 50 V de corriente continua por efecto Joule y una conductividad eléctrica de 70, medida en el SI en siemens (S). Al utilizar el mismo problema que el caso anterior el árbol de ficheros es el mismo mostrado en la sección 3.4. La única distinción de la carpeta `0/` es que deben ser añadidas las propiedades del potencial eléctrico `Vel` y `sigma`.

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      Vel;
}
// * * * * * * * * * * * * * * * * * * //

dimensions      [1 2 -3 0 0 -1 0];

internalField   uniform 25;

boundaryField
{
    ".*"
    {
        type            calculated;
    }
}
```

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      sigma;
}
// * * * * * * * * * * * * * * * * * * //

dimensions      [-1 -3 3 0 0 2 0];

internalField   uniform 70;

boundaryField
{
    ".*"
    {
        type            calculated;
    }
}
```

Añadimos las propiedades de las regiones tanto fluida (`extAir`) como sólida (`midSolid`) en `Vel` y `sigma`, en el diccionario `changeDictionaryDict`. Esto nos permitirá cambiar las condiciones de contorno de dichas variables de una forma rápida, en caso de obtener datos poco coherentes en la solución. En `extAir`:

Vel	sigma

```
    {
        internalField   uniform 25;

        boundaryField
        {
            inlet
            {
          type            zeroGradient;
            }
            outlet
            {
          type            zeroGradient;
            }
            exterior
            {
          type            fixedValue;
          value           uniform 0;
            }
            extAir_to_midSolid
            {
            type             fixedValue;
            value          uniform 50;
            }
            midSolid_to_extAir
            {
            type             fixedValue;
            value          uniform 50;
            }
        }
    }
```

```
    {
        internalField   uniform 70;

        boundaryField
        {
            inlet
            {
            type            fixedValue;
            value         uniform 70;
            }
            outlet
            {
            type             fixedValue;
            value         uniform 70;
            }
            exterior
            {
            type             fixedValue;
            value          uniform 70;
            }
            extAir_to_midSolid
            {
            type             fixedValue;
            value          uniform 70;
            }
            midSolid_to_extAir
            {
            type             fixedValue;
            value          uniform 70;
            }
        }}
```

en `midSolid`:

```
Vel
    {
        internalField   uniform 25;

        boundaryField
        {
            ".*"
            {
           type            empty;
           value           uniform 25;
            }
            midSolid_to_extAir
            {
            type             fixedValue;
            value          uniform 50;
            }
            extAir_to_midSolid
            {
            type             fixedValue;
            value          uniform 50;
            }
        } }
```

```
sigma
  {
        internalField   uniform 70;

        boundaryField
        {
            ".*"
            {
            type             empty;
            value            uniform 70;
            }
            extAir_to_midSolid
            {
            type             fixedValue;
            value          uniform 70;
            }
            midSolid_to_extAir
            {
            type             fixedValue;
            value          uniform 70;
            }
        } }
```

Finalmente, detallamos como se va a calcular la ecuación de Laplace en `fvSchemes`:

```
laplacianSchemes
{
...
laplacian(sigma,Vel) Gauss linear corrected;
}
```

Y para señalar el *solver*, las tolerancias y los algoritmos deben de añadir en `fvSolution` las dos variables:

```
Vel
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-14;
        relTol          0.1;
    }
```

```
VelFinal
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-14;
        relTol          0;
    }
```

Con todo esto, el caso está listo para simular. De igual forma que el caso anterior ejecutamos un *script* (`run`) para hacer todos los pasos necesarios de forma automática cambiando únicamente la aplicación. Esto se muestra a continuación:

```
rm -rf 0 constant system
rm *
cp -r ficherosNecesarios/0 .
cp -r ficherosNecesarios/constant .
cp -r ficherosNecesarios/system .

blockMesh > log.blockMesh
snappyHexMesh -overwrite > log.snappyHexMesh
extrudeMesh > log.extrudeMesh
splitMeshRegions -cellZones -overwrite > log.splitMesh

# remove fluid fields from solid regions (important for post-processing)
for i in midSolid
do
   rm -f 0*/$i/{mut,alphat,epsilon,k,U,p_rgh}
done

for i in midSolid extAir
do
   changeDictionary -region $i >& log.changeDictionary.$i
done

#-- Run on single processor
pyFoamPlotRunner.py jouleChtMultiRegionFoam > log.jouleChtMultiRegionFoam

paraFoam -touchAll
```

4.5 Resultados

Utilizando las aplicaciones proporcionadas por *Pyfoam* se puede obtener la representación gráfica del comportamiento de los residuales. Aunque hay que destacar que esta primera simulación se ha realizado teniendo en cuenta solo el efecto Joule.

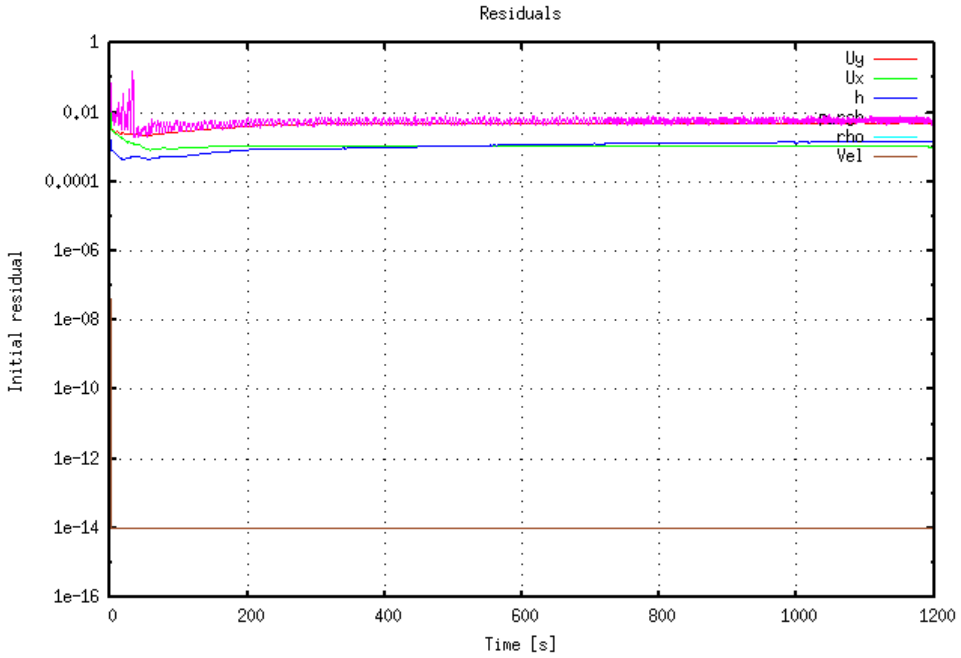

Figura 16. Caso B – Residuales cilindro efecto Joule.

De nuevo la simulación no termina porque se cumplen las condiciones de convergencia sino porque llega al límite de iteraciones. De igual manera que el caso anterior, es altamente probable que sea debido a la naturaleza transitoria de la simulación. A continuación, se presentan los resultados obtenidos prestando especial atención a las variables incluidas en este caso. Por ejemplo, en la siguiente imagen se ha representado el dominio computacional perteneciente a `extAir` o fluido, en él se puede apreciar cómo se comporta el campo eléctrico de 50V en el último instante de tiempo.

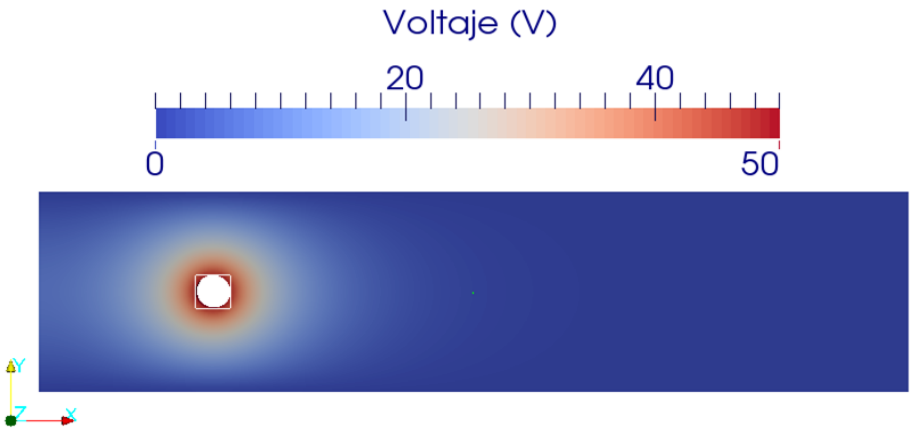

Figura 17. Caso B – Residuales cilindro efecto Joule.

Otra propiedad importante a observar sería el campo térmico que provoca el efecto Joule en dicho dominio computacional. Esto se puede observar en la siguiente imagen.

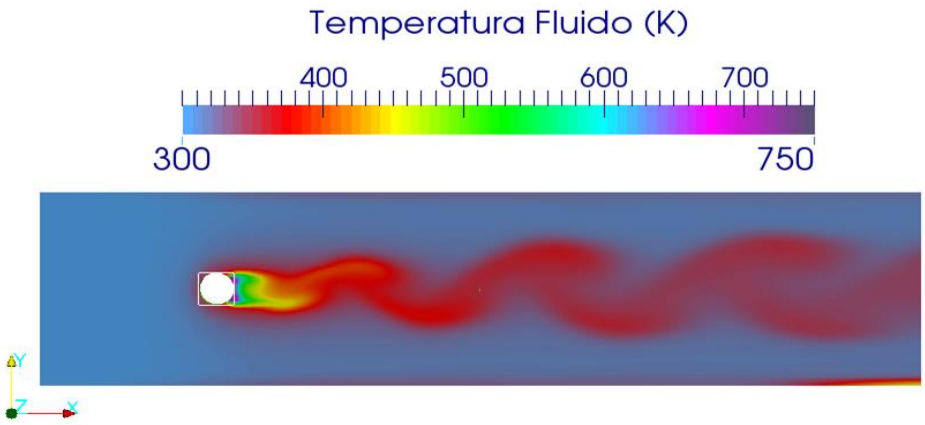

Figura 18. Caso B – Campo de temperatura fluido.

En este caso, se ha cambiado la escala de colores con el fin de apreciar de manera visual los gradientes de temperatura producidos por el cilindro en el fluido. También se puede apreciar la estela del fluido debido al carácter transitorio de la simulación. Finalmente, otra de las imágenes interesantes de esta simulación es la representación gráfica de la velocidad del campo fluido y de la temperatura alcanzada por el cilindro. Esto se muestra en la siguiente ilustración.

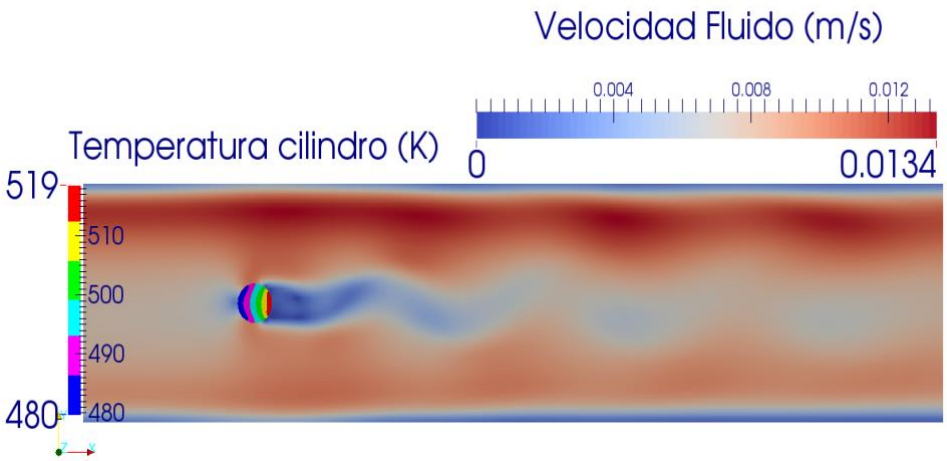

Figura 19. Caso B – Velocidad fluido & Temperatura cilindro.

Aquí se ha mantenido un régimen de colores básico (azul-rojo) para representar las variaciones de velocidad a través del dominio fluido, en contraposición con un régimen de colores más amplio y sin transición para la temperatura del cilindro. De forma que sean ambas regiones claramente diferenciables y se obtenga una visión clara de su comportamiento.

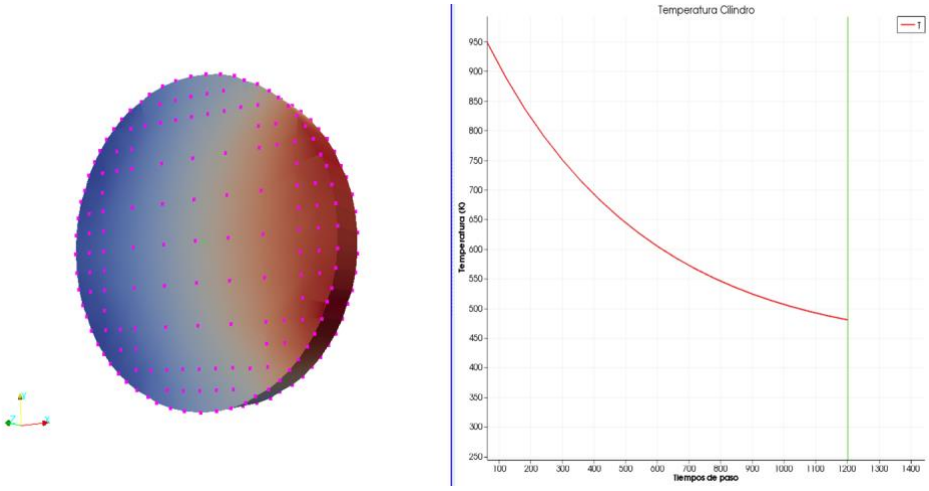

Figura 20. Caso B – Temperatura cilindro respecto del tiempo.

Finalmente se ha representado una gráfica de la evolución de la temperatura del cilindro respecto al tiempo. Se puede observar, debido a las condiciones de contorno impuestas, el cilindro tiende a enfriarse debido al fluido durante los instantes de tiempo que dura la simulación

4.6 Adición de radiación

Este apartado pretende añadir una propiedad física adicional al Caso B anterior. Esta es la radiación, que se ha explicado de manera breve en la sección 4.1.2. La preparación de este caso es relativamente sencilla ya que la aplicación principal que se ha estado utilizando `chtMultiRegionFoam` y `joulechtMultiRegionFoam` implementa este tipo de comportamiento mediante diferentes modelos computacionales. De forma que no es necesario modificar el *solver*, aunque sí que se deberán adecuar algunos diccionarios.

OpenFoam implementa una serie de modelos de radiación que pueden ser activados desde un diccionario llamado `radiationProperties`, dentro de la carpeta `constant` y para cada uno de los dominios computacionales. A continuación, se detallan brevemente las características de cada uno de estos modelos. En el caso de que se necesitara más información se recomienda consultar [16].

- `fvDOM`: es un método de volúmenes finitos el cual resuelve la ecuación de RTE (*Radiative transfer equation*) en un medio para n direcciones.
- `noRadiation`: es el modelo que no incluye la radiación en la ecuación de la energía, lo que devuelve son ceros.
- `OpaqueSolid`: No tiene nada que ver con los términos involucrados en la ecuación de la energía, pero crea el modelo `absorptionEmissionModel` y `scatterModel`.
- `P1`: funciona bien para aplicaciones relacionadas con la combustión donde la superficie óptica es relativamente grande. Asume que todas las superficies son difusas.
- `solarLoad`: este modelo incluye características del sol, como los flujos reflexivos y flujos radiactivos difusos.

Primero será necesario declarar la irradiación total `G` dentro de la carpeta `0/` como se muestra a continuación, manteniendo todas las propiedades del caso B intactas.

```
dimensions      [1 0 -3 0 0 0 0];
internalField   uniform 0;
boundaryField
{
    ".*"
    {
        type            calculated;
    }
}
```

Ahora dentro de la carpeta `constant`, para aplicar la radiación debemos editar el diccionario `radiationProperties` tanto para `extAir` como `midSolid`. Entonces `extAir` y quedaría:

```
radiation       on;

radiationModel             P1;

// Number of flow iterations per radiation iteration
solverFreq 3;

absorptionEmissionModel constantAbsorptionEmission;

constantAbsorptionEmissionCoeffs
{
    absorptivity    absorptivity [ 0 -1 0 0 0 0 0 ] 0.5;
    emissivity      emissivity   [ 0 -1 0 0 0 0 0 ] 0.5;
    E               E            [ 1 -1 -3 0 0 0 0 ] 0;
}

scatterModel   constantScatter;

constantScatterCoeffs
{sigma      sigma [0 -1 0 0 0 0 0] 0;
 C              C [0 0 0 0 0 0 0] 0;
}

sootModel       none;
```

Para `midSolid` quedaría:

```
radiation       off;

radiationModel  opaqueSolid;

absorptionEmissionModel constantAbsorptionEmission;

constantAbsorptionEmissionCoeffs
{
    absorptivity    absorptivity [ 0 -1 0 0 0 0 0 ] 0.0;  //opaque
    emissivity      emissivity [ 0 -1 0 0 0 0 0 ] 1;      //black hole
    E               E [ 1 -1 -3 0 0 0 0 ] 0;
}
constantScatterCoeffs
{
      sigma         sigma [0 -1 0 0 0 0 0] 0;
```

```
      C                 C [0 0 0 0 0 0 0] 0;
}
sootModel none;

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

Finalmente, dentro de la carpeta system los únicos diccionarios que se deberán de modificar en `extAir` serían `changeDictionaryDict` *y* `fvSolution` y para `midSolid` solo `fvSolution`. A continuación, se muestra únicamente lo que se ha añadido a cada diccionario. Para `extAir`, se han añadido las condiciones iniciales de *G* para cada una de las regiones. En `fvSolution`, se han especificado la ecuaciones, tolerancias y algoritmos usados para resolver dicha propiedad.

En changeDictionaryDict:

```
G
    {
        internalField   uniform 0;
        boundaryField
        {
   exterior
    {
        type            MarshakRadiation;
        value           uniform 0;
        T               T;
        emissivityMode  lookup;
        emissivity      uniform 1;
    }
    inlet
    {
        type            MarshakRadiation;
        value           uniform 0;
        T               T;
        emissivityMode  lookup;
        emissivity      uniform 1;
    }
    outlet
    {
        type            MarshakRadiation;
        value           uniform 0;
        T               T;
        emissivityMode  lookup;
        emissivity      uniform 1;
    }
    front
    {
        type            empty;

    }
    back
    {
        type            empty;
```

En fvSolution:

```
G

    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-5;
        relTol          0.1;
    }

GFinal
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-5;
        relTol          0.1;
     }
```

```
    }
    extAir_to_midSolid
    {
        type            MarshakRadiation;
        value           uniform 0;
        T               T;
        emissivityMode  lookup;
        emissivity      uniform 1;
    }
        }
    }
```

Para `midSolid`, únicamente se debe añadir el mismo código de `fvSolutions` en `extAir`. La malla que se ha generado para este apartado es la misma que se ha descrito en el Caso B anterior. Los pasos seguidos son idénticos y se secuencian en el mismo orden.

4.7 Resultados

Ahora en las siguientes imágenes se podrán observar los resultados añadiendo el efecto de la radiación, más concretamente activando el modelo P1.

Como en cada una de las simulaciones, los residuales se han representado, comportándose de forma análoga a los casos anteriores debido a su naturaleza transitoria. Es importante destacar que en este caso casi todas las gráficas relacionadas con la velocidad del fluido, la temperatura del fluido y del sólido y el campo eléctrico permanecen iguales.

Sin embargo, una de las propiedades más interesantes de representar es la irradiación total G, que se puede ver en la siguiente imagen para el último instante de tiempo.

Esta propiedad se ha representado en forma de contorno para obtener de forma más visual de su comportamiento a través del fluido `extAir`. Obteniendo unos valores que oscilan entre 2,513e+03 y 2,53e+03 W/m².

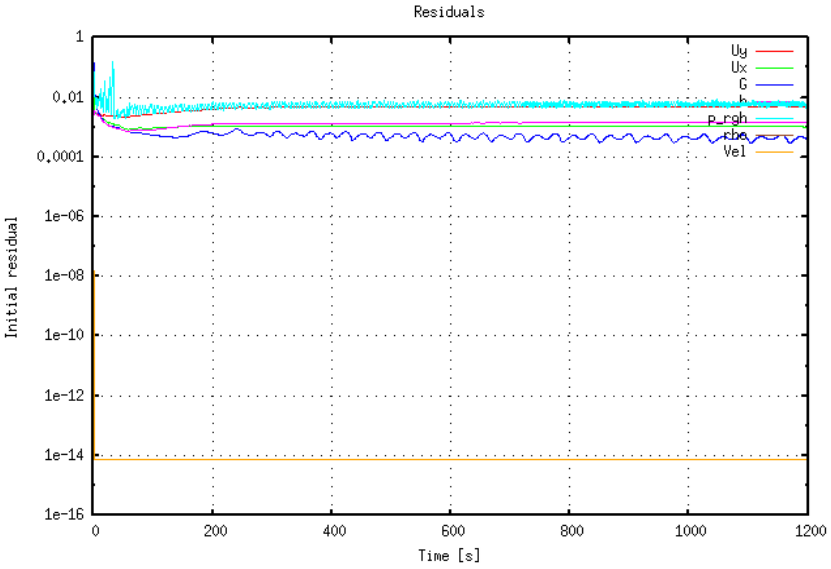

Figura 21. Caso B – Residuales cilindro efecto Joule y radiación.

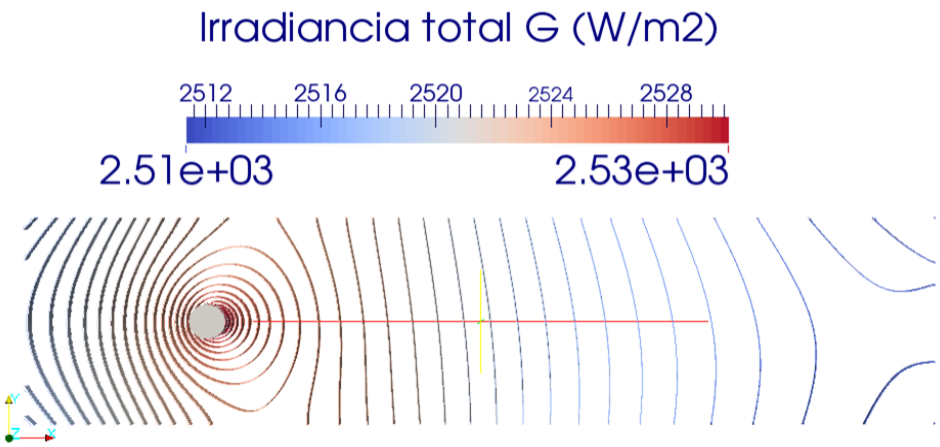

Figura 22. Caso B – Campo de irradiación total fluido.

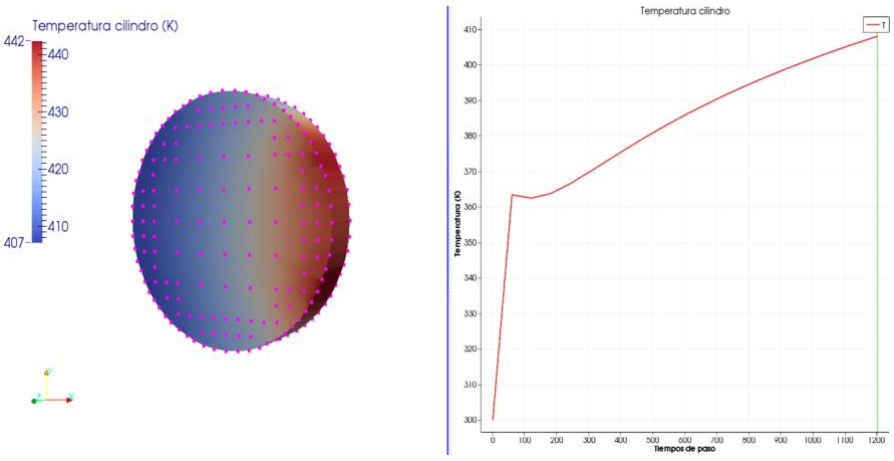

Figura 23. Caso B – Temperatura cilindro radiación respecto del tiempo.

Finalmente, se ha representado la evolución de la temperatura respecto del tiempo, obteniéndose conclusiones interesantes. Debido a la configuración de la simulación por la inclusión de la radiación la temperatura del cilindro aumenta linealmente. Esto significa que no se está refrigerando correctamente. Probablemente esto sea debido a las propiedades otorgadas al sólido (`midSolid`) en el modelo de radiación ya que no consigue desprender el calor generado y, por lo tanto, aumenta su temperatura.

4.8 Conclusiones

Sin duda, las obtenciones de los resultados en la simulación anterior arrojan interesantes resultados del comportamiento térmico tanto de la región fluida como la sólida a través de las modificaciones que se ha realizado en el *solver* y la adecuación del problema. En términos numéricos (despreciado la relevancia de las propiedades físicas del sólido y fluido), con un potencial de 50V se consigue que el fluido llegue hasta unos 750K y el sólido a 520K. Además, como se dijo al principio de este capítulo la radiación es un fenómeno que no necesita medio para su transferencia. Es uno de los procesos más importantes y su sentido físico es uno de los más interesantes de todos los modelos de transferencia de calor. Es relevante para muchos procesos en la industria de calor, refrigeración, secado y también método de transformación de energía que involucran combustión de combustibles fósiles y radiación solar.

5 CASO C:
Flujos incompresibles e inmiscibles multifásicos con distinta temperatura.

5.1 Introducción y contexto teórico

Este caso está centrado en procesos de flujos incompresibles e inmiscibles que pueden observar a día de hoy tanto en situaciones cotidianas como en ámbitos industriales. Algunos ejemplos son la caída libre de una gota de agua, rotura de una presa, etc. De esta forma, este capítulo será similar al llenado de una cavidad en el que interaccionan un fluido y el aire. Esto se puede asemejar al llenado de un depósito de combustible o a la inyección de un plástico en su molde.

A priori existen dos métodos para la resolución computacional de fluidos inmiscibles; *Interface tracking* e *Interface capturing* para n regiones fluidas inmiscibles con bajos números de Mach. Esto quiere decir que las ecuaciones a resolver son la ecuación de conservación de la masa para dos fluidos incompresibles y cantidad de movimiento para ambos fluidos:

$$\nabla \cdot \overrightarrow{V_g} = 0 \tag{46}$$

$$\frac{\partial \rho_{g,f} \overrightarrow{V_{g,f}}}{\partial t} + \nabla \cdot (\rho_{g,f} \overrightarrow{V_{g,f}} \overrightarrow{V_{g,f}}) = -\nabla p_{g,f} + \nabla \cdot (\bar{\bar{\tau}}'_{g,f} + \bar{\bar{\tau}}''_{g,f}) \tag{47}$$

donde

- t es el tiempo.
- g y f representa gas y fluido, respectivamente.
- p es la presión reducida; $p = P - \rho \vec{g} \cdot \vec{x}$
- P es la presión absoluta.
- ρ es la densidad.
- $\bar{\bar{\tau}}'$ es el tensor de esfuerzos viscosos.
- $\bar{\bar{\tau}}''$ es el tensor de esfuerzos de Reynolds.

Entre ambas fases existe una superficie libre o interfase en la que se acoplan las ecuaciones (46) y (47). Esta condición de contorno está compuesta por una condición cinemática y otra dinámica.

$$\vec{x} \in \Gamma: \overrightarrow{V_f} = \overrightarrow{V_g}, (\bar{\bar{\tau}}_f - \bar{\bar{\tau}}_g) \cdot \vec{n} = (P_f - P_g + \sigma\kappa) \cdot \vec{n} \quad (48)$$

siendo

- σ es el coeficiente de tensión superficial.
- κ es la curvatura local de la superficie; $\kappa \equiv \nabla \cdot \vec{n}$
- $\vec{n}$ es el vector unitario normal a la interfase

Por lo tanto, el método utilizado en este caso es *interface capturing*. De forma que el sistema de ecuaciones compuesto por (46), (47) para el fluido y el gas y la condición de contorno (48) se puede pueden combinar modelando la fuerza de superficie como fuerza de volumen de la forma:

$$\rho \frac{\partial \vec{V}}{\partial t} + \rho \nabla \cdot (\vec{V}\vec{V}) = -\nabla p + \nabla \cdot (\bar{\bar{\tau}}) - \sigma\kappa\delta(d)\vec{n} \quad (49)$$

$$\nabla \cdot \vec{V} = 0; \vec{x} \in \Omega \quad (50)$$

donde el vector velocidad se clasifica:

- $\overrightarrow{V_f} \rightarrow \vec{x} \in líquido$
- $\overrightarrow{V_f} = \overrightarrow{V_g} \rightarrow \vec{x} \in \Gamma$
- $\overrightarrow{V_g} \rightarrow \vec{x} \in aire$

La interfaz Γ se parametriza por la función distancia $\Phi \equiv d$ conocida como *level set*. El inconveniente que tiene es que se debe de reiniciar en cada paso de tiempo y por lo tanto no garantiza la conservación de la masa. Normalmente, esta función pierde un porcentaje de masa fluida a lo largo de la simulación. Por lo que hace imposible trabajar con propiedades discontinuas como la densidad o la viscosidad en la interfaz. Para evitar esto, se suaviza la transición entre fases mediante la función VoF (*Volume of Fluid*). Esto supone que las fases son miscibles una región muy próxima a la interfaz. Esto resulta que:

$$\rho = \rho_f \gamma + \rho_g (1 - \gamma) \tag{51}$$

$$\mu = \mu_f \gamma + \mu_g (1 - \gamma) \tag{52}$$

Por lo tanto, las ecuaciones a resolver serán:

$$\frac{\partial \rho}{\partial t} = \nabla \cdot (\rho \vec{V}) = 0 \tag{53}$$

$$\frac{\partial \rho \vec{V}}{\partial t} + \nabla \cdot (\rho \vec{V} \vec{V}) = -\nabla p + \nabla \cdot (\bar{\bar{\tau}}) - \sigma \kappa \nabla \gamma - \overrightarrow{g} \cdot \vec{x} \nabla \rho \tag{54}$$

$$\frac{\partial \gamma}{\partial t} + \nabla \cdot (\gamma \vec{V}) + \nabla \cdot [\gamma (1 - \gamma)(\overrightarrow{V_f} - \overrightarrow{V_g})] = 0 \tag{55}$$

Este método es conservativo, permite analizar flujos con cambios de forma y es fácil de implementar. En cambio, uno de los principales problemas que tiene es que normalmente se pierde la localización de la interfaz e introduce corrientes parásitas en flujos donde la tensión superficial es dominante.

5.2 Descripción del *solver* existente: interFoam

Como se ha dicho a lo largo de este proyecto, OpenFoam incentiva la reutilización de código y por tanto se utilizará como base el *solver* `InterFoam`. A continuación, se muestra el árbol de carpetas de dicho código numérico.

```
|----- interFoam.C
|----- UEqn.H
|----- setDeltaT.H
|----- pEqn.H
|----- createFields.H
|----- correctPhi.H
|----- alphaEqnSubCycle.H
|----- alphaEqn.H
|----- alphaCourantNo.H
|----- porousInterFoam
|       |----- createPorousZones.H
|       |----- UEqn.H
|       |----- porousInterFoam.C
|       |----- Make
|             |----- options
|             |----- files
|----- LTSInterFoam
|      |----- setInitialrDeltaT.H
```

```
|      |----- LTSInterFoam.C
|      |----- setrDeltaT.H
|       |----- Make
|            |----- options
|            |----- files
|----- Make
|      |----- files
|      |----- options
|----- InterMixingFoam
       |----- interMixingFoam.C
       |----- createFields.H
       |----- alphaEqnsSubCycle.H
       |----- alphaEqns.H
    |----- threephaseInterfaceproperties
    |        |----- threephaseInterfaceproperties.C
    |        |-----  threephaseInterfaceproperties.H
    |----- incompressibleThreePhaseMixture
    |        |----- incompressibleThreePhaseMixture.C
    |        |----- incompressibleThreePhaseMixture.H
    |----- immiscibleIncompressibleThreePhaseMixture
    |        |----- immiscibleIncompressibleThreePhaseMixture.C
    |        |----- immiscibleIncompressibleThreePhaseMixture.H
    |----- Make
    |        |----- options
    |        |----- files
|----- InterDyFoam
       |----- correctPhi.C
       |----- interDyMFoam.C
       |----- pEqn.H
       |----- readControls.H
    |----- Make
    |        |----- options
    |        |----- files
```

En este código numérico existen archivos comunes a los explicados en los casos anteriores. Por lo que se omitirá su descripción. El código principal es el archivo `interFoam.C`, el lector puede consultar el código en la referencia [19]. Los demás archivos tienen como objetivo:

- `setDeltaT.H`: resetea el paso de tiempo para mantener un máximo número de Courant.
- `CorrectPhi.H`: corrige funciones de flujo para asegurar continuidad.
- `alphaEqnSubCycle.H`: crea un campo de superficie escalar para el flujo másico.
- `alphaEqn.H`: resuelve la ecuación para `alpha`.
- `alphaCourantNo.H`: calcula y otorga el mínimo y máximo número de Courant.

- `createPorousZones.H`: crea las regiones porosas.
- `porousInterFoam.C`: código numérico para calcular 2 fluidos inmiscibles, isotérmicos e incompresibles usando la aproximación VOF de captura de interfaz basada en fracciones de fase.
- `setInitialrDeltaT.H`: establece el tiempo de paso inicial correspondiente al algoritmo de ajuste del paso de tiempo en `setDeltaT`.
- `setrDeltaT.H`: resetea el paso de tiempo para mantener un máximo número de Courant para LTS (*Local Time Stepping*).
- `interMixingFoam.C`: código numérico para 3 fluidos incompresibles, dos de ellos miscibles usando el método VOF para capturar la interfaz.
- `threePhaseInterfaceProperties.C`: calcula propiedades para ayudar al solver principal.
- `threePhaseInterfaceProperties.H`: calcula propiedades para ayudar al solver principal.
- `incompressibleThreePhaseMixture.C`
- `incompressibleThreePhaseMixture.H`
- `immiscibleIncompressibleThreePhaseMixture.C`
- `immiscibleIncompressibleThreePhaseMixture.H`: modelo de transporte para una mezcla inmiscible incompresible de fases.
- `interDyFoam.C`: código numérico que resuelve 2 fluidos incompresible, isotérmicos e inmiscibles usando el método VOF de captura de interfaz basada en fracciones de fase, con movimiento de malla opcional y cambios de topología de malla que incluyen adaptación de re-mallado.

5.3 Modificaciones incluidas en el *solver*

En el apartado anterior se recalcó que el *solver* `interFoam` resuelve flujos incompresibles, isotérmicos e inmiscibles. Es decir, el proceso debe ser a temperatura constante. Pero, ¿y si entre ambos fluidos existe una diferencia de temperatura? Con el *solver* actual no seríamos capaces de obtener el gradiente de temperatura entre

ambas regiones. Por ello, el objetivo de este caso se resume al acoplamiento en dicho *solver* de la ecuación de la energía y simular el llenado de una cavidad sencilla. El código pasará a llamarse `interThermalFoam`.

Para un caso monofásico, a la hora de incorporar la energía interna se requiere de dos parámetros adicionales como son la capacidad térmica c_p y la difusividad térmica$k$. En un caso bifásico, por lo tanto, el número de variables se dobla y se requiere capturar la interfaz entre fases que induce discontinuad en la densidad ρ y en la viscosidad μ como se señaló en el apartado anterior. Además, hay que destacar que en `interFoam` la difusividad térmica k no se define a través de valores iniciales, sino que se calcula a partir del calor específico c_p y el número de Prandtl Pr con la expresión siguiente

$$k = \rho \frac{C_p}{Pr} \tag{56}$$

Para llevar a cabo la configuración de este caso, se deberán de modificar librerías. Antes de comenzar las modificaciones, se recomienda que el usuario se copie la carpeta del código numérico a modificar a una carpeta propia para así evitar posibles modificaciones indeseadas. La primera carpeta es `incompressibleThreePhaseMixture`, esta está localizada en el directorio raíz de instalación de OpenFoam y se debería de copiar al directorio raíz del usuario. Por ejemplo;

`$WM_PROJECT_USER_DIR/src/transportModels/myincompressible).`

Aquí se debe editar el archivo con extensión .H y añadir el calor específico c_p y el número de Prandtl Pr para ambas fases:

```
DimensionedScalar rho1_;
DimensionedScalar rho2_;
DimensionedScalar cp1_;
DimensionedScalar cp2_;
DimensionedScalar Pr1_;
DimensionedScalar Pr2_;
```

```
const dimensionedScalar& rho2() conts
{
      return rho2_;
```

```
};
const dimensionedScalar& cp1() conts
{
      return cp1_;
};
const dimensionedScalar& cp2() conts
{
      return cp2_;
};
const dimensionedScalar& Pr1() conts
{
      return Pr1_;
};
const dimensionedScalar& Pr2() conts
{
      return Pr2_;
};
```

Nota: En este apartado, no se han detallado todas las líneas de código correspondientes a todos los archivos a modificar, debido a que supondría una extensión desmesurada del documento. Sin embargo, se ha representado con un color gris claro aquellas líneas de código previas a la modificación o adicción de código.

Seguidamente se edita el archivo con extensión .C, de forma que en el constructor se deben de incorporar las líneas de código que siguen:

```
rho2_("rho", dimDensity, nuModel2_->viscosityProperties().lookup("rho")),
cp1_("cp", dimensionSet(0 , 2 , -2 , -1 , 0 , 0 , 0 ),
nuModel_->viscosityProperties().lookup("cp")),
```

Esto se debe hacer de igual manera para `cp2_`, `Pr_1` y `Pr_2`. Ahora en la función de `read()`:

```
rho2_("rho", dimDensity, nuModel2_->viscosityProperties().lookup("rho")),
cp1_("cp", dimensionSet(0 , 2 , -2 , -1 , 0 , 0 , 0 ),
nuModel_->viscosityProperties().lookup("cp")),
```

Esta librería accede a `cp` y le asigna el valor de la variable `cp1_` con las dimensiones establecidas en `dimesionSet`.

Una vez definidos el calor específico c_p y el número de Prandtl Pr hay que calcular la difusividad térmica k en las caras de las celdas para evaluar el flujo de calor $k\nabla T$, por lo que se debe añadir una función como se hace con `muf()` y `nuf()` en el código.

```
Foam::tmp<Foam::surfaceScalarField>
Foam::myincompressibleTwoPhaseMixture::kappaf() const
{
```

```
    const surfaceScalarField alpha1f
    (
      min(max(fvc::interpolate(alpha1_), scalar(0)), scalar(1))
    );
}
return tmp<surfaceScalarField>
(
  new surfaceScalarField
     (
      "kappaf",
     (
       alpha1f*rho1_*cp1_*(1/Pr1_)
       *fvc::interpolate(nuModel1_->nu())
       + (scalar(1) - alpha1f)*rho2_*cp2_
        *(1/Pr2_)*fvc::interpolate(nuModel2_->nu())
      )
   )
 );
}
```

Tras estas modificaciones la librería queda lista. Compilando la librería se asegura el re-uso en cualquier otra aplicación de OpenFoam. Aunque antes de ello, se debe de editar el archivo `files`, localizado en la capeta `Make`, indicando el nuevo destino de los archivos una vez copiados en la ruta de actuación del usuario.

Para iniciar los campos escalares c_p y T así las variables ρc_p y $\rho\phi c_p$, ya que todas aparecen en la ecuación de la energía, primero se debe de modificar `createFields.H`. Por lo tanto, añadimos:

```
Info<< "Reading field T\n" << endl;
volScalarField T
(
  IOobject
  (
    "T",
     runTime.timeName(),
     mesh,
     IOobject::MUST_READ,
     IOobject::AUTO_WRITE
  ),
 mesh
);
```

Se deben leer los objetos `cp1` y `cp2` desde `twoPhaseProperties` de manera análoga a `rho1` y `rho2`:

```
const dimensionedScalar& rho1 = twoPhaseProperties.rho1();
const dimensionedScalar& rho2 = twoPhaseProperties.rho2();
const dimensionedScalar& cp1 = twoPhaseProperties.cp1();
const dimensionedScalar& cp2 = twoPhaseProperties.cp2();
```

Añadimos el campo ρc_p necesario para la derivada temporal en la ecuación de la energía:

```
volScalarField rhoCp
(
   IOobject
   (
     "rhoCp",
      runTime.timeName(),
      mesh,
      IOobject::READ_IF_PRESENT
    ),
      alpha1*rho1*cp1 + alpha2*rho2*cp2,
      alpha1.boundaryField().types()
  );
 rhoCp.oldTime();
```

Y de forma análoga $\rho\phi c_p$ para evaluar los flujos de calor:

```
volScalarField rhoCpPhi
(
   IOobject
   (
     "rhoCpPhi",
      runTime.timeName(),
      mesh,
      IOobject::NO_READ,
      IOobject::NO_WRITE
    ),
  fvc::interpolate(rhoCp)*phi
);
```

Ahora se deben de actualizar estos valores. Esto se realiza en `alphaEqnSubCycle.H` y `alphaEqn.H` respectivamente;

```
rhoPhi = tphiAlpha()*(rho1 - rho2) + phi*rho2;
rhoCpPhi = tphiAlpha()*(rho1*cp1 - rho2*cp2) + phi*rho2*cp2;
```

```
rho == alpha1*rho1 + alpha2*rho2;
rhoCp == alpha1*rho1*cp1 + alpha2*rho2*cp2;
```

Finalmente, escribimos la ecuación de la energía como `Teqn.H`:

```
surfaceScalarField kappaf = twoPhaseProperties.kappaf();
fvScalarMatrix TEqn
(
fvm::ddt(rhoCp,T)
+ fvm::div(rhoCpPhi,T)
- fvm::laplacian(kappaf,T)
);
TEqn.solve();
```

Como último paso se recomienda encarecidamente que se compile el código para saber si todo se ha realizado correctamente y no se obtiene ningún error de lectura.

5.4 Preparación del caso

A través de la metodología de reutilización de código extendida por OpenFoam, se va hacer uso del tutorial `damBreak` localizado en la carpeta de tutoriales de OpenFoam. La estructura de carpetas es la siguiente:

```
|---- damBreak
|     |----- 0
|     |     |----- alpha.water.
|     |     |----- p_rgh
|     |     |----- U
|     |----- constant
|     |     |----- dynamicMeshDict
|     |     |----- g
|     |     |----- transportProperties
|     |     |----- turbulenceProperties
|     |     |----- polyMesh
|     |     |     |----- blockMeshDict
|     |----- system
|     |     |----- controlDict
|     |     |----- decomposeParDict
|     |     |----- fvSchemes
|     |     |----- fvSolution
|     |     |----- setFieldDict
|     |----- run
```

Las condiciones iniciales están esquematizadas en la ilustración siguiente. En ella, se puede apreciar las diferentes regiones (pared, salida, simetría y entrada) dentro del dominio computacional. La entrada del fluido será en dirección Y a una velocidad de 1mm/s a una temperatura de 400K. Aquellas regiones con la condición de pared se otorgará una condición de contorno inicial de 300K. Esto producirá el enfriamiento del fluido por diferencia de temperatura. Además, se presenta la malla y su nivel de refinamiento.

Comenzando con la carpeta `0/`, se debe añadir la propiedad de la temperatura como un archivo independiente a los demás existentes de la siguiente forma:

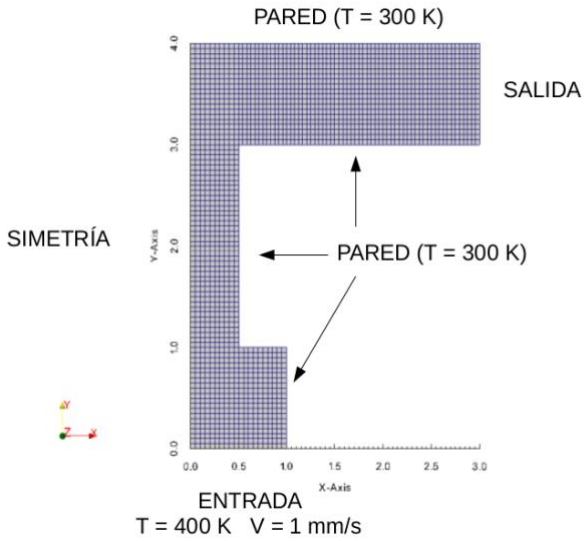

Figura 24. Caso C – Malla Caso C.

```
Dimensions      [0 0 0 1 0 0 0];
internalField uniform 0;
boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform 400;
    }
    outlet
    {
        type            zeroGradient;
    }
    Right
    {
        type            symmetryPlane;
    }
    wall
    {
        type            fixedValue;
        value           uniform 300;
    }
    frontAndBack
    {
        type            empty;
    }
}
```

En `fvSchemes`:

```
div(rhoPhi,U) Gauss linearUpwind grad(U);
div(rhoCpPhi,T) Gauss linearUpwind grad(T);
```

En `fvSolutions`:

```
T
{
```

```
solver   BICCG;
preconditioner DILU;
tolerance 1e-07;
relTol 0;
}
```

Finalmente, en `setFieldDict`:

```
defaultFieldValues
   (
     volScalarFieldValue alpha.water 0

     volScalarFieldValue T 0

    );
  regions
    (
       boxToCell
        {
           box (0 0 -1) (0.1461 0.292 1);
           fieldValues
              (
                volScalarFieldValue alpha.water 1
              );
        }
       boxToCell
        {
           box (0.2 0.3 -1) (0.3 0.4 1);
           fieldValues
        (
              volScalarFieldValue T 300
        );
        }
    );
```

Finalmente, para correr la simulación se ha utilizado el siguiente *script*:

```
pyFoamClearCase.py /home/alex/Desktop/damTempbreak

blockMesh > log.blockMesh

#-- Run on single processor

pyFoamPlotRunner.py interThermalFoam > log.interThermalFoam
```

5.5 Resultados

Una vez terminada la simulación para 6000s, en la que se han ido guardando instante de tiempo cada 100s, que no asuste la gráfica obtenida de residuales. Como se ha nombrado en varias ocasiones a lo largo de este proyecto, el carácter transitorio de la simulación se refleja de esta manera.

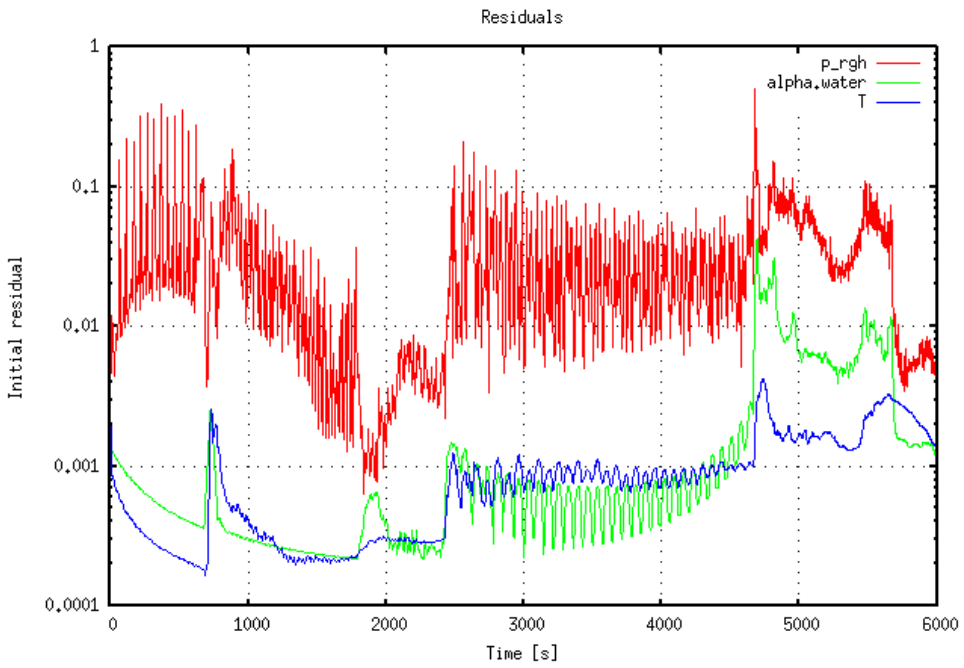

Figura 25. Caso C – Residuales inyección molde.

En este caso, las variables más representativas son la temperatura, la velocidad y la interfaz entre ambos fluidos inmiscibles. En las siguientes imágenes se puede ver el estado de la cavidad en el primer instante de tiempo:

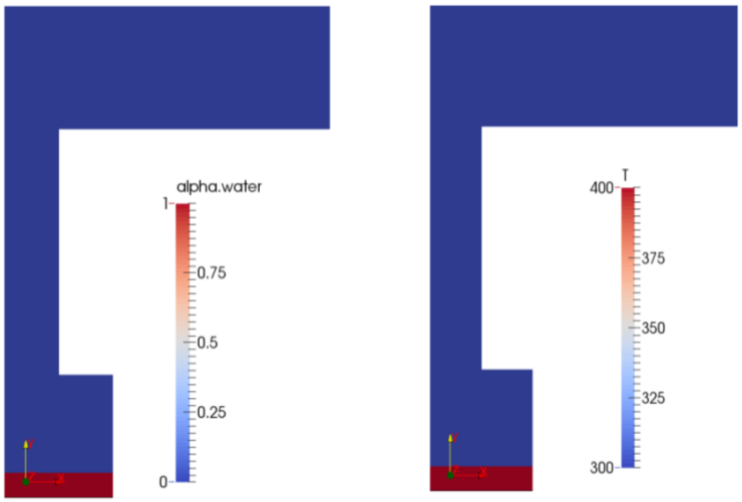

Figura 26. Caso C – Alpha.water y temperatura de fluidos inmiscibles.

En cuanto a la solución de la temperatura respecto del tiempo, en las siguientes imágenes se puede observar el comportamiento de esta propiedad hasta el último paso de tiempo (los valores de temperatura corresponden a la leyenda de la imagen anterior).

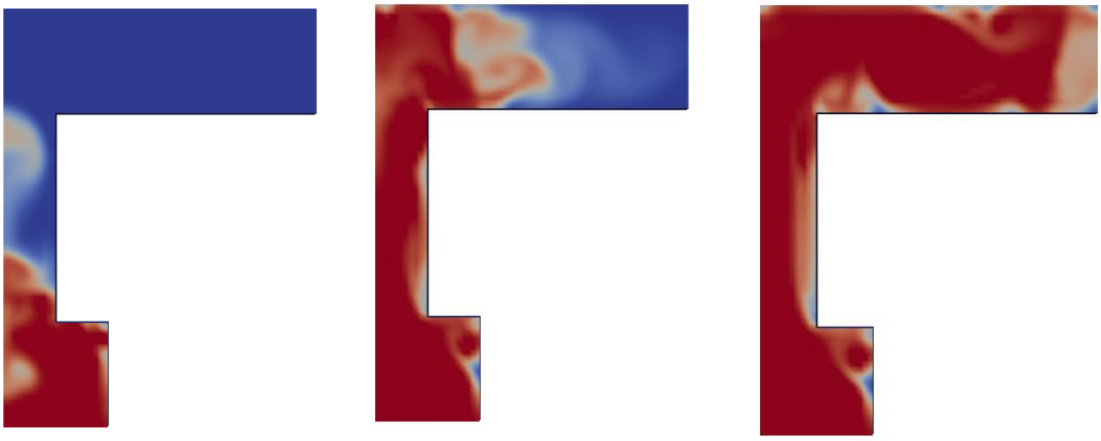

Figura 27. Caso C – Evolución de la temperatura en el molde.

Finalmente, si representamos `alpha.water` encargada de la interfaz entre los dos fluidos inmiscibles podemos observar en se forman burbujas. Si extrapolamos este fenómeno a un proceso de inyección; la producción de este tipo de cavidades o atrapamientos de aire, se consideran situaciones no deseables debido a su alta probabilidad de dejar algún defecto óptico, topológico o funcional a la pieza que se está inyectando.

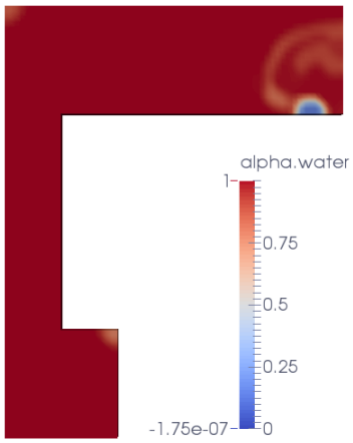

Figura 28. Caso C – Creación de una burbuja en la inyección.

5.6 Conclusiones

A través de la personalización de código en OpenFoam se ha llegado a la caracterización de un problema determinado, adaptado a las necesidades del usuario. Este proceso se puede extrapolar a numerosos ámbitos de la ingeniería donde el

desarrollo y optimización de este tipo de simulaciones producirían un impacto relevante tanto en términos de tiempo como económicos. Además, mediante la simulación se podrían detectar posibles problemas como el caso singular de la burbuja atrapada entre la pared y el fluido.

6 CASO D: Proceso térmico de llenado por inyección.

6.1 Introducción y contexto teórico

Este caso se adentra en la Reología. Es decir, se estudiará la relación entre el esfuerzo y la deformación de los materiales que son capes de fluir. Recientemente, este proceso físico se presenta en su mayoría en el proceso de inyección de moldes, considerado económicamente uno de los procesos más importantes dentro de la producción productos de plástico. En la actualidad, algunos termoplásticos se han reforzado con porcentajes de fibra, pudiendo aumentar el rendimiento mecánico y reducir su peso considerablemente. Además, este proceso uno de los más atractivos actualmente para diferentes industrias como la aeronáutica o automóvil. La variable relevante en este tipo de estudios es la viscosidad, que es la propiedad del fluido que le permite transferir impulso en virtud de la difusión. Por lo tanto, un fluido puede ser Newtoniano si su viscosidad se considera constante o no Newtoniano si esta varía a lo largo del tiempo. Denotaremos la viscosidad dinámica como la propiedad del fluido que describe la relación entre la velocidad de deformación y el esfuerzo tangencial, como muestra la siguiente expresión:

$$\tau = \mu \frac{\partial u}{\partial x} \tag{57}$$

también se puede calcular la viscosidad cinemática υ, como la división entre la viscosidad dinámica y la densidad:

$$\upsilon = \frac{\mu}{\rho} \tag{58}$$

El comportamiento reológico de un fluido no Newtoniano se divide en cuatro categorías principales:

- Comportamiento tixotrópico (depende del tiempo): muestra la relación de la viscosidad a lo largo del tiempo, está relacionado principalmente con la estructura interna del fluido y el tiempo de relajación.
- Comportamiento pseudoplástico (adelgazamiento por cizalladura): relaciona los cambios de incremento del esfuerzo cortante con la disminución de la viscosidad.

- Comportamiento dilatante (espesor de corte): relaciona los cambios de incremento del esfuerzo cortante con el aumento de la viscosidad.
- Reopéptico: la viscosidad aumenta con el esfuerzo a lo largo del tiempo.

Por lo tanto, queda bastante claro que modelar la viscosidad de un fluido no Newtoniano está ligada a aquella dependencia que tengan. Por lo tanto, un modelo universal para el comportamiento de dicho fluido no sería realista. Las siguientes imágenes muestra el comportamiento explicado anteriormente de forma gráfica:

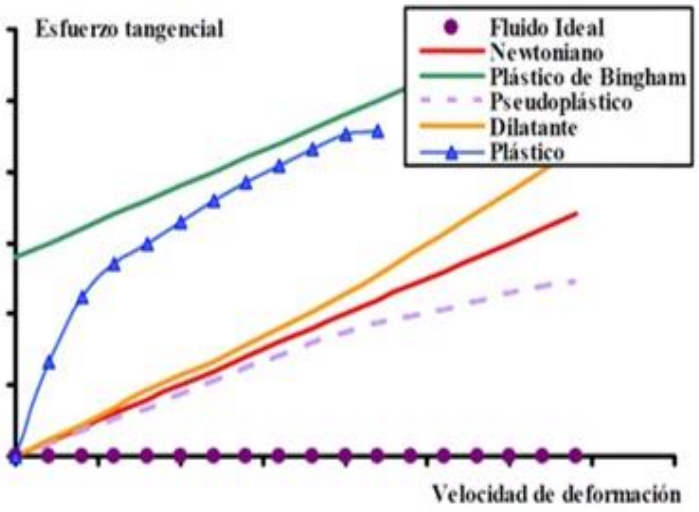

Figura 29. Caso D – Esfuerzo tangencial frente Velocidad de deformación.

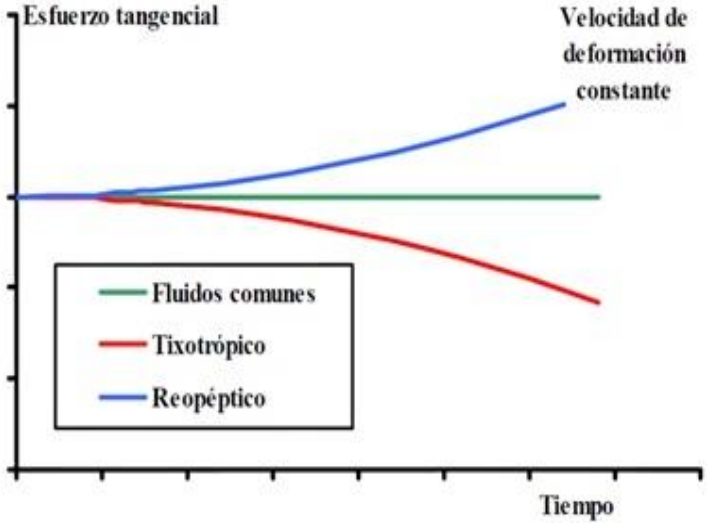

Figura 30. Caso D – Esfuerzo tangencial frente Velocidad de deformación.

Es por ello que el caso expuesto en este apartado se centrará principalmente en un fluido no Newtoniano con dependencia de temperatura, monofásico, incompresible y estacionario. Se utilizará la geometría y la malla que el caso C, por simplificación.

En cuanto a OpenFoam, se pueden implementar varios modelos de viscosidad como *BirdCarreau*, *CrossPowerLaw*, *HerschelBulkley*, *Newtonian* y finalmente el que se utilizará será *powerLaw*. Este modelo otorga una relación básica para la viscosidad y la velocidad de deformación. El valor de la viscosidad puede ser asignado por un valor mínimo y otro máximo. La relación es la siguiente:

$$\upsilon = k\dot{\gamma}^{n-1} \tag{59}$$

$$k = k_0 - m_k(T - T_0) \tag{60}$$

donde

- k es el índice de consistencia del flujo (m²/s)
- k_0 es el valor inicial del índice de consistencia del flujo (m²/s)
- m_k es el coeficiente de dependencia de temperatura.
- $\dot{\gamma}$ es la velocidad de deformación (s^{-1})
- n es el índice de comportamiento del flujo

En cuanto al índice de comportamiento del flujo tenemos:

- Si $0 < n < 1$: el fluido muestra comportamiento pseudoplástico o adelgazamiento por cizalladura.
- Si $n = 1$: el fluido muestra comportamiento Newtoniano.
- Si $1 < n$: el fluido muestra comportamiento dilatador o espesor de corte.

Por lo tanto, vamos a adaptar dicho modelo para que incluya primero la dependencia con la temperatura y luego lo incorporaremos al *solver* `simpleFoam`. Este *solver* resuelve las ecuaciones de continuidad, cantidad de movimiento y energía explicadas al inicio de este documento y las expresiones relacionas con la Reología (49) (50).

6.2 Descripción del *solver* existente: simpleFoam

El código numérico `simpleFoam` está constituido para resolver flujos estacionarios, incompresibles y la opción de varios modelos de turbulencia. Este está localizado la carpeta raíz de instalación de OpenFOAM:

```
$WM_PROJECT_DIR/applications/solvers/incompressible/simpleFoam
```

A continuación, se muestra su estructura de archivos:

```
|----- UEqn.H
|----- simpleFoam.C
|----- pEqn.H
|----- createFields.H
|----- SRFSimpleFoam
|       |----- createFields.H
|       |----- pEqn.H
|       |----- SRFSimpleFoam.C
|       |----- UrelEqn.H
|       |----- Make
|            |----- options
|            |----- files
|----- porousSimpleFoam
|      |----- createZones.H
|      |----- pEqn.H
|      |----- porousSimpleFoam.C
|      |----- Ueqn.H
|       |----- Make
|            |----- options
|            |----- files
|----- Make
|      |----- files
|      |----- options
```

Algunos de los archivos no vistos hasta el momento se detallan a continuación:

- `SRFSimpleFoam.C`: código numérico estacionario para fluidos no Newtoniano incompresible y turbulento en una única fracción rotacional.
- `UrelEqn.H`: predice el momento relativo.
- `porousSimpleFoam.C`: código numérico estacionario para fluidos incompresibles, turbulentos, explícitos o implícitos y con soporte para fracciones múltiples de referencia (*MRF*).

Centrándonos en el *solver* principal vemos que únicamente se basa en la solución de la ecuación de continuidad para un fluido incompresible y la ecuación de cantidad de movimiento:

```
#include "fvCFD.H"
#include "singlePhaseTransportModel.H"
#include "RASModel.H"
#include "simpleControl.H"
#include "fvIOoptionList.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

int main(int argc, char *argv[])
{
    #include "setRootCase.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "createFields.H"
    #include "createFvOptions.H"
    #include "initContinuityErrs.H"

    simpleControl simple(mesh);

    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

    Info<< "\nStarting time loop\n" << endl;

    while (simple.loop())
    {
        Info<< "Time = " << runTime.timeName() << nl << endl;

        // --- Pressure-velocity SIMPLE corrector
        {
            #include "UEqn.H"
            #include "pEqn.H"
        }

        turbulence->correct();

        runTime.write();

        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }

    Info<< "End\n" << endl;

    return 0;
}
```

6.3 Modificaciones incluidas en el *solver*

A priori, se deben de eliminar las carpetas de `SRFSimpleFoam` y `porousSimpleFoam` por no ser necesarias, después se deben renombrar los archivos ya copiados en el directorio raíz del usuario como se ha hecho en los casos anteriores. Se actualizan también los archivos file and `options` dentro de `Make`. Llamaremos al nuevo *solver* `tempSimpleFoam`.

Ahora se crea el fichero que contiene la ecuación de la energía, `Teqn.H`;

```
tmp<fvScalarMatrix> TEqn
( fvm::div(phi, T)
- fvm::laplacian(TempD_, T) );
Teqn().relax();
solve(TEqn());
```

Y la invocamos desde el archivo renombrado de `simpleFoam` a `tempSimpleFoam.C`:

```
// --- Pressure-velocity SIMPLE corrector
{ #include "UEqn.H"
 #include "pEqn.H"
 #include "Teqn.H" }
```

Por último, definimos la variable de la temperatura en `createFields.H`:

```
Info<< "Reading field T\n" << endl;
volScalarField T
(
      IOobject
      (
        "T",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
      ),
  mesh
);
```

Y la difusividad térmica también:

```
dimensionedScalar TempD_(laminarTransport.lookup("TempD"));
```

6.4 Preparación del caso

Lo primero que se hace es copiar con otro nombre el modelo de viscosidad `powerLaw` al directorio raíz del usuario, de esta forma evitamos posibles problemas en caso de modificaciones indeseadas. Por consiguiente, se deben actualizar las dependencias de la carpeta `Make` y sus archivos correspondientes `files` y `options`.

Seguidamente en el archivo con la extensión .H, debemos añadir:

```
  dimensionedScalar n_;
  dimensionedScalar kslope_;
  dimensionedScalar Tbase_;
```

Ahora en el archivo renombrado con la extensión .C, debemos añadir en el constructor:

```
k_(tempdeppowerLawCoeffs_.lookup("k")),
n_(tempdeppowerLawCoeffs_.lookup("n")),
kslope_(tempdeppowerLawCoeffs_.lookup("kslope")),
Tbase_(tempdeppowerLawCoeffs_.lookup("Tbase")),
```

Y en el `Member Funtions` :

```
tempdeppowerLawCoeffs_.lookup("kslope") >> kslope_;
tempdeppowerLawCoeffs_.lookup("Tbase") >> Tbase_;
```

En `Private Member Funtions`, añade en `calcNu`:

```
const volScalarField& T= U_.mesh().lookupObject<volScalarField>("T");
```

Remplaza por último el valor de `k_` y compila:

```
(k_-kslope_*(T-Tbase_))
```

Una vez terminado los pasos anteriores, se puede preparar el tutorial que se utilizará como base para la simulación. La geometría será el misma que el apartado C y la estructura organizativa de ficheros es la siguiente:

```
|---- damBreak
|    |----- 0
|    |    |----- p
|    |    |----- T
|    |    |----- U
|    |----- constant
|    |    |----- RASProperties
|    |    |----- transportProperties
|    |    |----- polyMesh
|    |    |    |----- blockMeshDict
|    |    |    |----- boundary
|    |    |    |----- faces
|    |    |    |----- neighbour
|    |    |    |----- owner
|    |    |    |----- points
|    |----- system
|    |    |----- controlDict
|    |    |----- fvSchemes
|    |    |----- fvSolution
|    |----- run
```

Las condiciones de contorno se han establecido de acuerdo a la siguiente imagen, análoga al caso C. En la región de entrada se inyectará el fluido a una temperatura de 400K y a una velocidad de 0.001 mm/s.

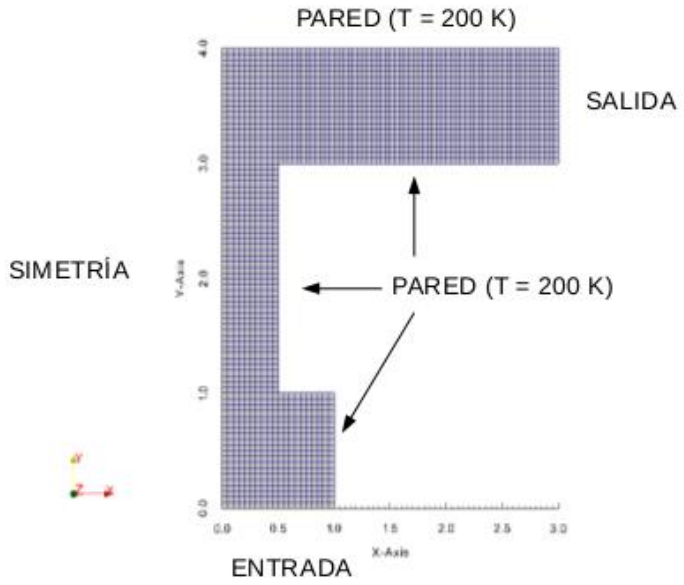

Figura 31. Caso D – Condiciones de contorno.

Por lo tanto, en `0/` tendremos la presión, la temperatura y la velocidad como:

```
FoamFile
{
  version     2.0;
  format      ascii;
  class  volScalarField;
  object      p;
}
// * * * * * * * * * * * //

dimensions [0 2 -2 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    inlet
    {
    type
zeroGradient;
    }

    outlet
    {
    type
fixedValue;
    value           uniform
0;
    }

    Right
    {
    type      symmetryPlane;
    }
```

```
FoamFile
{
    version     2.0;
    format      ascii;
    class
volVectorField;
    object      U;
}

// * * * * * * * * * * * * * * *
//

dimensions  [0 1 -1 0 0 0
0];

internalField   uniform (0 0
0);

boundaryField
{
    inlet
    {
    type fixedValue;
   value uniform (0 0.000001
0);
    }

    outlet
    {
        type
zeroGradient;
    }
    Right
    {
```

```
FoamFile
{
    version     2.0;
    format      ascii;
    class
volScalarField;;
    object      T;
}
// * * * * * * * * * * * * * *
*//

dimensions      [0 0 0 1 0 0
0];

internalField   uniform 30;

boundaryField
{
    inlet
    {
    type        fixedValue;
    value       uniform 400;
    }

    outlet
    {
   type
zeroGradient;
    }
    Right
    {
   type
symmetryPlane;
    }
```

```
    wall
    {
    type       zeroGradient;
    }
    frontAndBack
    {
    type         empty;
    }
}
```

```
    type
symmetryPlane;
    }

    wall
    {
    type         fixedValue;
    value      uniform (0 0
0);
    }
    frontAndBack
    {
    type         empty;
    }
}
```

```
    wall
    {
    type       fixedValue;
    value      uniform 200;
    }
    frontAndBack
    {
    type            empty;
    }
}
```

En `constant/` tendremos la carpeta `polyMesh` relacionada con la malla, el diccionario `RASProperties` y `transportProperties` se detallan a continuación:

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      RASProperties;
}
// * * * * * * * * * * * * * * * * * * //
RASModel        laminar;

turbulence      off;

printCoeffs     off;
```

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * //
transportModel  tempdeppowerLaw;

nu              nu [ 0 2 -1 0 0 0 0 ] 1;

tempdeppowerLawCoeffs
{
  k       k [0 2 -1 0 0 0 0] 2800;
  n       n [0 0 0 0 0 0 0] 0.64;
  kslope  kslope [0 2 -1 -1 0 0 0] 0.5;
  Tbase   Tbase [0 0 0 1 0 0 0] 300;
  nuMi    nuMin [0 2 -1 0 0 0 0] .1;
 nuMax nuMax [0 2 -1 0 0 0 0] 10000000;
   }
TempD    TempD  [0 2 -1 0 0 0 0] 1e-8;
```

En archivos en /system se pueden reutilizar de cualquier otro tutorial que utilice el *solver* `simpleFoam`.

6.5 Resultados

Una vez terminada la simulación, al igual que los demás casos, la herramienta de post-procesado de OpenFoam conocida como *Paraview* ayuda a obtener un resultado gráfico de los resultados. Además, mediante la herramienta *Pyfoam*, en concreto pyFoamPlotWatcher.py se representan los residuales de la simulación.

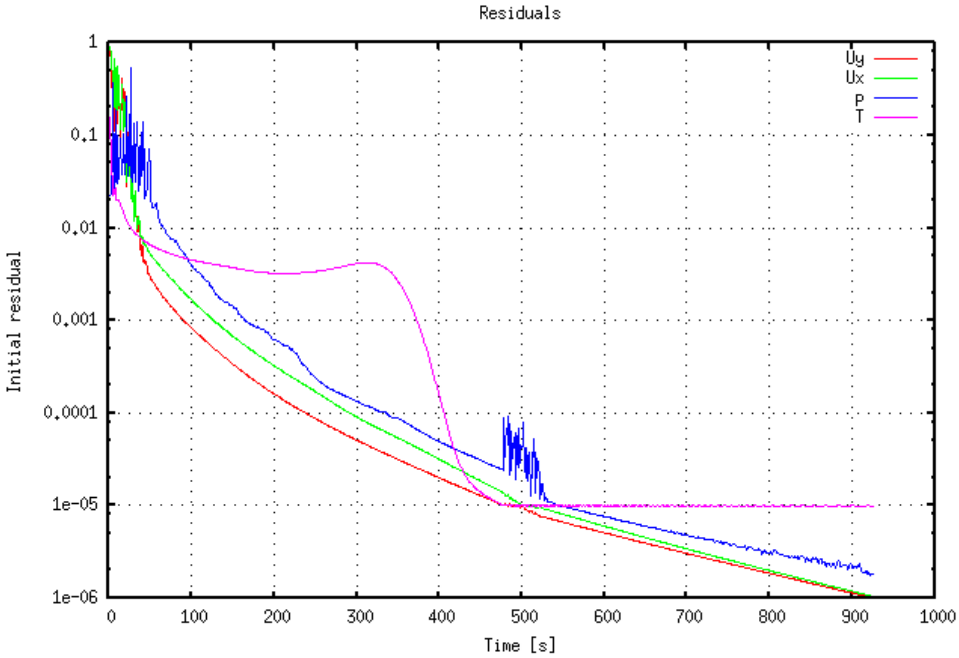

Figura 32. Caso D – Residuales modelo de inyección.

En esta simulación, los residuales han llegado al límite de convergencia en vez de agotar el número de iteraciones máximas. Las siguientes imágenes muestran los resultados de algunas propiedades físicas en el último instante de tiempo.

En ellas se puede observar el comportamiento de las cuatro propiedades características del problema simulado. La velocidad del fluido alcanza los 2,78e-06 m/s en la zona más estrecha de paso. En el mapa de temperatura se puede observar claramente la transición de la zona de la pared del molde a 200K hasta el cuerpo de la inyección el cual se encuentra a 400K.

Finalmente, se ha representado también el comportamiento estacionario de la presión y un en la figura 32.

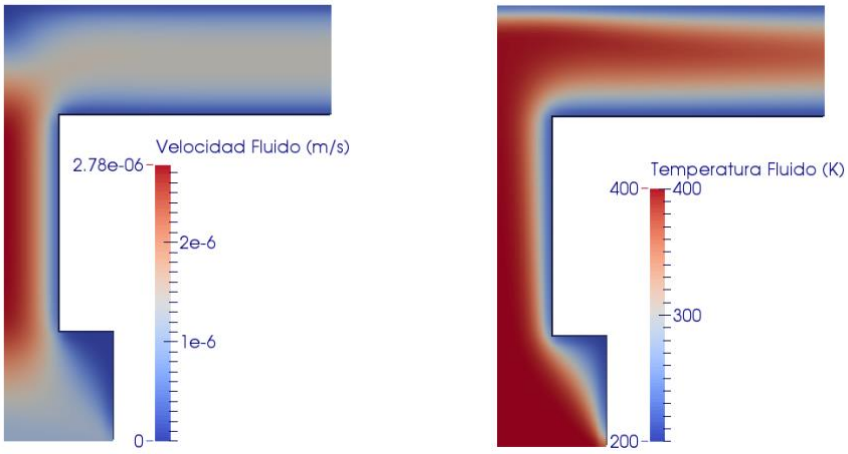

Figura 33. Caso D – Resultados de la velocidad y la temperatura.

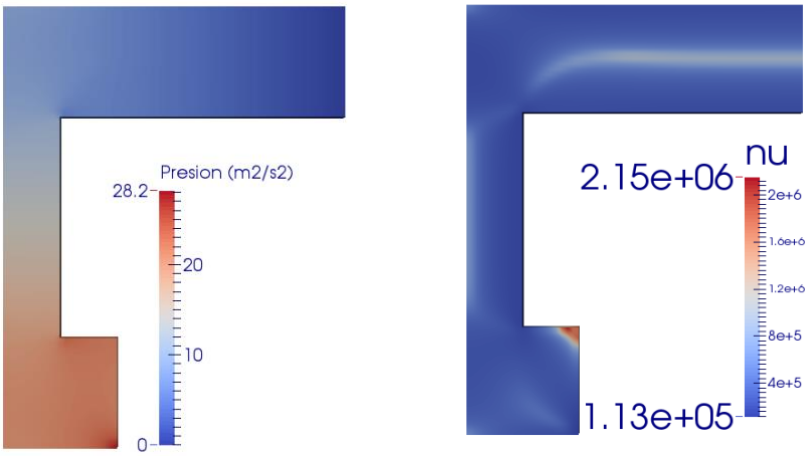

Figura 34. Caso D – Resultados de la presión y nu.

7 Conclusiones finales

Después de la obtención de los resultados, cabe destacar como primera premisa la versatilidad, personalización y potencial del software de cálculo OpenFoam. En segundo lugar, su naturaleza *open source* hace que sus limitaciones se reduzcan ya que el impacto de ahorro económico en términos de empresa es considerable debido a que no es necesaria la adquisición de licencias. Lo cual lo hace un software a tener en cuenta en numerosas ocasiones. Los diferentes casos detallados en este documento se han elegido por sus amplias intervenciones en el mundo de la ingeniería. Cada uno de ellos se puede extrapolar a numerosos procesos relacionados con el amplio abanico de la mecánica de fluidos y la transferencia de calor. Adaptando los diferentes códigos numéricos a las necesidades del usuario para optimizar procesos.

También, mediante el proceso de s*cripting* (comandos secuenciales) se obtiene un compromiso entre ahorro de tiempo y uso del software. Esto simplifica enormemente el tiempo de preparación de simulaciones relacionadas, por ejemplo, con la optimización topológica (caso A). Proceso crucial en industrias como la aeronáutica o automovilística, donde la cantidad de elementos a validar es abrumadoramente elevada.

Además, debido a los altos estándares de calidad, a veces simulaciones térmicas se llevan en paralelo con simulaciones de inyección en molde, para reducir considerablemente el tiempo de validación y poder pasar al proceso de manufactura cuanto antes.

Por lo tanto, lo que se ha querido plasmar en este proyecto a grandes rasgos ha sido el nivel de personalización que puede llegar a tener OpenFoam aplicado a situaciones nada lejanas de la realidad como pueden ser: optimización topológica, calentamiento de un fluido por efecto Joule (y radiación), llenado de una cavidad con dos fluidos inmiscibles a diferentes temperaturas, o inyección de plástico en un molde con transferencia de calor.

8 Bibliografía

[1] Christopher J. Greenshields, (2016). *OpenFOAM User Guide 4.0* . England: OpenFOAM Foundation Ltd.

[2] John D. Andersonm Jr., (1995). Computational Fluid Dynamics: The basics with applications. New York: Mc Graw Hill Education.

[3] Frank P. Incropera, David P. DeWitt, Adrienne S. Lavine & Theodore L. Bergman (2011). Fundamentals of Heat and Mass Transfer – 4th Edition. Canada: John Wiley & Sons, Inc.

[4] John D. Andersonm Jr., (1995). Computational Fluid Dynamics: The basics with applications. New York: Mc Graw Hill Education.

[5] Randall J. LeVeque, (2006). Finite Difference Methods for Differential Equations. Seattle, Washington: Siam (Society for Industrial & Applied Mathematics).

[6] Antonio Crespo Martínez, (2006). Mecánica de Fluidos. España, Madrid: Paraninfo, S.A.

[7] Bernhard F.W. Gschaider (2009). pyFoam - Happy foaming with Python. ICE Strömungsforschung, Gothenburg.

[8] Bernhard F.W. Gschaider (2010). Automatization with pyFoam. ICE Strömungsforschung , Gothenburg.

[9] Pijush K.Kundu & Ira M.Cohen (2008). Fluid Mechanics. Fourth Edition. Academic Press, Elsevier Inc. San Diego, USA.

[10] Jonh C. Tannehill, Dale A. Anderson, Richard H. Pletcher (1997). Computational Fluid Mechanics and Heat Transfer. Second Edition. Taylor&Francis Ltd. Philadelphia, USA.

[11] Frank M. White (2004). Mecánica de Fluidos. Quinta Edición.McGraw-Hill/Interamericana de España S.A.U. Madrid, España

[12] Bernhard Gschaider. Contrib/PyFoam. OpenFOAM Wiki. (2005). https://openfoamwiki.net/index.php/Contrib/PyFoam

[13] Henry Weller, Chris Greenshields, Cristel de Rouvray. OpenFOAM, the OpenFOAM Foundation. https://openfoam.org/

[14] Alessandro Bottaro, Joel Guerrero, Damiano Natali, Jan Pralits. Mesh generation using snappyHexMesh. Wolf-Dynamics. http://www.wolfdynamics.com/wiki/meshing_OF_SHM.pdf

[15] Yuzhu Pearl Li. Implementation of multiple time steps for the multi-physics solver based on chtMultiRegionFoam (2017). University of Stavanger. http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2016/YuzhuPearlLi/final_report_Jan2017.pdf

[16] OpenCFD Ltd, OpenFOAM Trademark. ESI Group. OpenFOAM v1712 API – Thermophysical models: Radiation models. https://www.openfoam.com/documentation/cppguide/html/group__grpRadiationModels.html

[17] Patricio Bohórquez & Luis Parras. Curso de Mecánica de Fluidos Computacional Aplicada. Técnicas Reunidas S.A. Escuela Politécnica Superior de Jaén, Universidad de Jaén; Escuela Politécnica Superior de Ingenieros Industriales, Universidad de Málaga. http://blogs.ujaen.es/prmedina/wp-content/uploads/2014/04/slides.pdf

[18] Rong Zhen, Roger I. Tanner & Xi-Jun Fan. (2011) Injection Molding; Integration of Theory and Modeling Methods. University of Sydney. Australia. Springer.

[19] OpenFOAM Ltd. OpenFOAM v1712, The open source CFD toolbox; Extended Code Guide. (2018). https://www.openfoam.com/documentation/cpp-guide/html/index.html