Universidad Zaragoza
1542

Trabajo Fin de Grado

Análisis y explotación del framework Palabos para la resolución de problemas de aerodinámica utilizando el método Lattice-Boltzmann

Analysis and exploitation of the Palabos framework for the solution of solid body aerodynamics using the Lattice-Boltzmann method

Autor/es

Javier Justes Larrosa

Director/es

Norberto Fueyo Díaz

ESCUELA DE INGENIERÍA Y ARQUITECTURA

Año 2017

Análisis y explotación del framework Palabos para la resolución de problemas de aerodinámica utilizando el método Lattice-Boltzmann

Resumen

Para el presente proyecto, se ha desarrollado un software de fluidodinámica computacional (CFD) que permite realizar simulaciones de aerodinámica externa utilizando el método Lattice-Boltzmann.

Frente a otras soluciones, en las que se requiere un gran trabajo para obtener un mallado adecuado, pre-procesando la geometría de trabajo y posteriormente depurando la malla obtenida, con esta herramienta el proceso de discretización del dominio se realiza de manera automática, realizando unos ajustes mínimos en el fichero con la geometría de trabajo.

Resulta especialmente interesante su facilidad de uso, como ya se ha indicado, y que se trate de una herramienta gratuita y Open Source, permitiendo así que cualquier persona puede acceder a su código y modificarlo para adaptarlo a sus necesidades. Para no tener que desarrollar todo el código desde cero, se ha utilizado el framework Palabos en el núcleo del programa, parte encargada de realizar la simulación, a la que se le ha dotado de una interfaz gráfica y un sistema de avisos en Python.

Para comprobar el funcionamiento del túnel, se han realizado una serie de simulaciones, todas ellas con números de Reynolds bajos, ya que, como se explicará, haberlas realizado para números de Reynolds más elevados hubiera supuesto un coste computacional muy elevado, del cual no se disponía.

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. Javier Justes Larrosa,

con nº de DNI 18055531W en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado, (Título del Trabajo) Análisis y explotación del framework Palabos para la resolución de problemas de aerodinámica utilizando el método Lattice-Boltzmann,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 18 de abril de 2017

Fdo: 

A mi familia por su constante apoyo.

Al personal de SHU DIGITAL por su valiosa ayuda.

Índice general

Índice de figuras

Índice de cuadros

CAPÍTULO 1

Introducción

La aerodinámica es el campo de la Mecánica de Fluidos que estudia el movimiento de los fluidos cuando estos circulan alrededor un cuerpo, y las acciones que producen sobre éste, como las fuerzas y los momentos.

Para el estudio de la aerodinámica externa de cuerpos, una de las herramientas más potentes de las que se dispone son los túneles de viento, con los cuales se pueden simular situaciones en las que se encontraran los objetos de estudio en situaciones reales de uso. Por desgracia, el precio de estos túneles de viento hace que no estén fácilmente disponibles; además, para su uso hace falta una formación adecuada.

1.1 Objetivo y justificación

El objetivo de este proyecto es obtener un software que permita simular un túnel de viento de manera virtual. Dicho "túnel de viento virtual", tendrá que ser capaz de calcular las fuerzas que el flujo ejerce sobre el cuerpo, y exportar los datos necesarios para poder realizar una visualización del flujo obtenido.

Todo se desarrollará bajo licencia de tipo Open Source, para que esté libre de licencias comerciales y pueda ser mejorado o adaptado a las necesidades que cualquiera necesite.

Para la simulación, se utilizará el método de Lattice-Boltzmann, por las ventajas que ofrece, por ejemplo en términos de representación de la geometría, frente a otras alternativas numéricas como elementos o volúmenes finitos.

Por último, se pretenden crear unas herramientas auxilares de análisis de los datos que las simulaciones proporcionan.

Como base para este proyecto se va a utilizar Palabos[1]. Palabos es un framework que proporciona una librería muy completa para la resolución de problemas Computational Fluid Dynamics (CFD) mediante el método Lattice-Boltzmann.

Para representar los sólidos en el túnel de viento virtual se va a utilizar el formato STL. STL es un formato abierto, y por lo tanto es fácil encontrar herramientas que interactúen con él; la mayoría del software CAD permite exportar en este formato, o se pueden encontrar las extensiones necesarias para que hacerlo.

El proyecto pretende, por tanto, crear un prototipo de herramienta tipo "túnel de viento virtual" que permita explorar de forma sencilla la aerodinámica externa de cuerpos sólidos. Dicha herramienta puede utilizarse, debidamente extendida:

- Para la investigación de flujos en entornos académicos;
- Para su explotación comercial (por ejemplo, como una herramienta en la nube);
- Para la realización de prácticas docentes.

Sin embargo, es importante aclarar desde el principio que la mayor parte de las simulaciones presentadas son para números de Reynolds mucho más bajos de los encontrados en problemas prácticos de Ingeniería Aerodinámica. Esto no es una limitación del software desarrollado, sino de las posibilidades de cálculo disponibles para el desarrollo del Trabajo Fin de Grado. Para simular flujos a mayor número de Reynolds solo haría falta más potencia de cálculo (por ejemplo, más "cores" utilizando las tecnologías de paralelización que se introducen más adelante.).

1.2 Tecnologías de software usadas en este proyecto

Este proyecto combina e integra el uso de diferentes tecnologías de vanguardia en el campo de la Ingeniería Computacional:

- **Palabos**. Es un Framework Open Source de Computational Fluid Dynamics (CFD) basado en el método Lattice-Bolztmann. Proporciona una librería de clases con las que desarrollar un programa en C++ que desarrolle la simulación que se desea realizar. Es utilizado en este Trabajo como motor que realiza las simulaciones.

[1] (*Palabos* n.d.).

- **Git**. Software de control de versiones. Se ha utilizado para gestionar los avances en el código de los programas creados. Dicho código ha sido almacenado remotamente en los repositorios de Bitbucket. Esta memoria también se ha gestionado en Git y Bitbucket.
- **Python**. Lenguaje de programación interpretado y multiparadigma. De su excelente conjunto de librerías cobran especial relevancia aquellas de uso científico, utilizadas en el presente trabajo para el análisis de datos.
- **ParaView**. Software de visualización científica Open Source. Desarrollado mediante la librería VTK, que proporciona un motor de visualización y procesado de campos escalares, vectoriales y tensoriales. Soporta la arquitectura cliente-servidor y su ejecución en entorno de cluster, con el fin de poder visualizar grandes conjuntos de datos de manera remota. Es usado para visualizar las líneas de corriente y *slices* del campo de velocidades y presiones.
- **Cluster de ordenadores en Linux**. Debido al alto coste computacional que conlleva realizar una simulación fluidodinámica, se ha utilizado un cluster de ordenadores que permite ejecutar de manera paralela una simulación, con la correspondiente reducción de tiempo.
- **MPI** o Interfaz de Paso de Mensajes. Es un estándar utilizado por programas de cálculo científico, que se ejecutan paralelamente, para intercambiar información entre procesos y sincronizar su ejecución.
- **STL**. Formato de diseño asistido por ordenador. Representa la geometría de un cuerpo tridimensional mediante un conjunto de caras triangulares que se ajustan a su superficie. Es un formato abierto y, actualmente, está soportado por la mayoría de herramientas CAD del mercado.
- **Qt**. Framework multiplataforma para el desarrollo de interfaces gráficas. En este proyecto es utilizada a través del binding PyQt del lenguaje Python para generar una interfaz gráfica al software desarrollado.
- **Telegram Messenger**. Es un servicio de mensajería gratuito a través de Internet. Originalmente estaba orientado a dispositivos móviles, pero actualmente dispone de una librería para Python que permite el uso de los denominados *bots*, cuentas controladas a través de un programa informático. Gracias a esta utilidad, en este proyecto es usado para mandar un aviso directamente al teléfono del usuario que ejecute la simulación cuando ésta finaliza.

1.3 Estructura de la memoria

La presente memoria está estructurada en cuatro capítulos. El primero de los capítulos es la presente introducción.

El segundo capítulo empieza con una introducción teórica al método de Lattice-Boltzmann, y al framework de programación Palabos. Posteriormente se explica la herramienta desarrollada y una guía de uso para ésta.

En el tercero se presentan las simulaciones que se han realizado para probar dicha herramienta. En primer lugar se ha estudiado la precisión que proporciona el método Lattice-Boltzmann, para ello se ha comparado el error obtenido en una serie de simulaciones con los parámetros usados en cada una de ellas. Posteriormente, se han realizado distintas pruebas aerodinámicas a diferentes geometrías, dos perfiles alares, una motocicleta y un edificio.

Para finalizar, un último capítulo donde se plasman las conclusiones y líneas de trabajo futuro.

CAPÍTULO 2

El Túnel de Viento Virtual

En este capítulo se introducen brevemente los túneles de viento y las técnicas de Computational Fluid Dynamics (CFD). A continuación se describe en más detalle el trabajo realizado en este proyecto: el túnel de viento virtual.

2.1 Túneles de viento y técnicas CFD en aerodinámica

Los túneles de viento son herramientas muy útiles en aerodinámica, que permiten estudiar el movimiento del aire alrededor de un cuerpo sólido en condiciones reales de operación.

Con el desarrollo de herramientas computacionales modernas, como el método de los volúmenes finitos, y junto con el avance de la informática en los últimos años, la Fluidodinámica Computacional (CFD) se ha convertido en un potente complemento, y a veces alternativa, al túnel de viento experimental.

Entre las ventajas de la simulación está que proporciona un conjunto muy detallado de resultados, difícil de obtener experimentalmente, y que es muy económica, particularmente cuando es necesario ensayar muchos cambios geométricos para optimizar un diseño.

En un estudio aerodinámico, es esencial obtener una indicación visual de cómo se mueve el flujo alrededor del cuerpo de estudio, para comprobar la localización de las zonas de remanso, las zonas turbulentas o el desprendimiento del flujo y las estelas que genera. En los túneles reales se utilizan técnicas experimentales como la inyección de trazadores (por ejemplo, humo). En las técnicas CFD, se utilizan las líneas de corriente generadas a través del software de visualización de datos.

Para la medición de las fuerzas, en un túnel real se requiere de sistemas de medida, tales como anemómetros, acoplados a la maqueta del sólido. En las técnicas CFD las fuerzas se calculan

integrando numéricamente el tensor de esfuerzos sobre la superficie del cuerpo.

2.2 El Método de Lattice-Boltzmann

El método de Lattice-Boltzmann es un método de resolución numérica de las ecuaciones de Navier-Stokes utilizado en Dinámica de Fluidos Computacional.

El método de Lattice-Boltzmann método se inspira en las ecuaciones utilizadas por la física estadística para la modelización de gases, las cuales suponen que dicho gas está formado por un conjunto de partículas que se mueven libremente, colisionando elásticamente entre ellas.

En el método de Lattice-Boltzmann se discretiza el espacio fluido en una serie de nodos, a los cuales se les asocia un número de "partículas virtuales" que conforman el fluido, y que se transfieren entre nodos vecinos en el dominio de acuerdo con la ecuación:

$$f_a(\vec{x}+\vec{e_a}\delta_t, t+\delta_t) = f_a(\vec{x},t) + \frac{1}{\tau}(f_a^{eq} - f_a) \tag{2.1}$$

donde:

- f_a es el número "partículas virtuales" que se mueven hacia el nodo vecino a con velocidad $\vec{e_a}$.
- $\vec{x}$ nodo en el cual se está calculando dicho flujo.
- $\vec{e_a}$ el vector de velocidad discreta del nodo actual con el nodo contiguo en la dirección a.
- δ_t incremento discreto de tiempo en cada paso temporal.
- τ es un parámetro de relajación, cuyo valor determinará la viscosidad del fluido.
- f_a^{eq} distribución de partículas en el equilibrio.

Dicha ecuación es una adaptación para un dominio discreto de la ecuación de Boltzmann, en la cual se ha simplificado el término de colisión. Dicho término recoge el efecto de las colisiones entre las partículas que representan el fluido. Las variables macroscópicas, tales como la velocidad del fluido o la densidad, se pueden calcular como funciones de la distribución de partículas.

Para facilitar el análisis, es frecuente adimensionalizar las ecuaciones diferenciales (macroscópicas) que rigen el problema, llegando a un nuevo sistema donde el problema quede definido por las ecuaciones adimensionales de continuidad y cantidad de movimiento, respectivamente:

$$\nabla_d \cdot \vec{u}_d = 0 \tag{2.2}$$

$$\frac{\partial \vec{u}_d}{\partial t_d} + (\vec{u}_d \cdot \nabla_d)\vec{u}_d = -\nabla_d p_d + \frac{1}{Re}\nabla_d^2 \vec{u}_d \tag{2.3}$$

La ventaja de adimensionalizar las ecuaciones es que todos los problemas geométricamente semejantes tienen la misma solución, siempre que el número de Reynolds *Re* (que representa la relación entre fuerzas de inercia y viscosas en el fluido) sea el mismo, y las correspondientes condiciones de contorno sean también semejantes.

Se pueden utilizar diferentes modelos de fluido, tales como gas ideal o Van der Waals entre otros, que dan lugar a diferentes formulaciones. La formulación usada en este proyecto utiliza el modelo de gas ideal.

El Apéndice A contiene más información relativa al método de Lattice-Boltzmann.

2.3 Modelo de turbulencia

Es bien conocido que, en flujo turbulento, una simulación no puede discretizar todas las escalas espaciales y temporales del flujo. Por esta razón, en las simulaciones de flujo turbulento utilizan modelos de turbulencia.

En el Túnel de Viento Virtual de este proyecto, se ha utilizado un modelo tipo de *Large Eddy Simulation* (LES), que estima e introduce una 'viscosidad turbulenta' (*eddy viscosity*, en inglés), para aproximar el efecto de la energía disipada por las turbulencia de pequeña escala, no resuelta en el modelo.

La viscosidad turbulenta ν_t está relacionada con el tensor de esfuerzos de las escalas resueltas τ^r_{ij} mediante la ecuación:

$$\tau^r_{ij} - \frac{1}{3}\tau_{ij}\delta_{ij} = -2\nu_t \overline{S}_{ij} \tag{2.4}$$

Para la estimación de dicha viscosidad se ha utilizado el modelo de Smagorinsky–Lilly, en el que se calcula como:

$$\nu_t = (\Delta_g C_s)^2 |S| \tag{2.5}$$

donde C_s es una constante que depende de cada problema pero que, usualmente, toma un valor entre 0.1 y 0.2. Para las simulaciones que se hacen en el presente proyecto se ha tomado un valor de 0.14.

2.4 Palabos

Para evitar tener que programar toda la funcionalidad del programa desde cero, se utilizó Palabos.

Palabos es un framework para la realización de simulaciones CFD mediante el método Lattice-Boltzmann. Éste proporciona una serie de librerías en C++ que implementan un conjunto de plantillas de clases, tipos de datos y funciones. Además, proporciona un conjunto de ejemplos para tomarlos como base a la hora de desarrollar un programa o, simplemente, para aprender de ellos.

Todo el código se encuentra bajo la licencia AGPLv3, por lo que los programas desarrollados a partir de Palabos presentan condicionantes en el tipo de licencia que pueden utilizar. En particular, es necesario liberar el código de los programas que utilicen Palabos y estén usándose para prestar un servicio.

Como los cálculos por el método Lattice-Boltzmann son altamente paralelizables, las funciones del framework se han construido sobre la Interfaz de Paso de Mensajes (MPI) y el multiprocesamiento simétrico (SMP). De esta manera, es posible ejecutarlo en una maquina multi-núcleo, o un cluster de ordenadores, para distribuir la carga de trabajo entre los distintos núcleos de cálculo.

La potencia de Palabos se basa en sus plantillas. Programadas para almacenar arrays de 2 o 3 dimensiones, según sea una simulación 2D o 3D, necesarios para modelar las nubes de puntos. Existen diferentes clases en función del tipo de dato que se desee almacenar: escalares, vectoriales, tensoriales u otros propios del método.

Con el objetivo de mejorar la estabilidad y convergencia del método, se desaconseja trabajar con bloques que contengan, directamente, la densidad y la velocidad del fluido; en su lugar se recomienda trabajar con objetos que almacenen las magnitudes:

$$\bar{\rho} = \rho - 1 \tag{2.6}$$

$$\vec{j} = \rho \cdot \vec{u} \tag{2.7}$$

La densidad oscila en torno a la unidad (el método numérico es para flujo compresible) y en caso contrario conviene realizar un proceso de adimensionalización para conseguirlo. Con la resta de la ecuación 2.6 se obtiene una densidad para el cálculo cuyo valor está alrededor de cero.

Este cambio permite almacenar el valor de la densidad con una mayor precisión, ya que el sistema interno de almacenamiento de números decimales utilizado por los ordenadores permite almacenar una mayor cantidad de decimales en el rango (0,1) que en cualquier otro.

Existen variaciones del método Lattice-Boltzmann para una gran cantidad formulaciones, conocidas como "dinámicas". Palabos incorpora algunas de ellas, tales como la dinámica *Single-relaxation-time BGK*, otras más elaboradas que permiten simulaciones para mayores números de Reynolds cómo *Regularized BGK* o *Multiple Relaxation Time*, dinámicas que permiten la existencia de varias fases como el modelo *Shan/Chen*, modelos para fluidos no newtonianos, e incluso dinámicas que incorporan la temperatura como variable para la simulación de flujos térmicos.

Para establecer las condiciones iniciales lo más usual consiste en asignar, a todo el bloque, una densidad constante y velocidad nula. El valor de presión será calculado por Palabos a través de la ecuación de estado que corresponda a la dinámica utilizada.

Para definir una superficie que está inmersa en un fluido, hay que establecer qué nodos del bloque lattice pertenecen al cuerpo y cuáles al fluido. Una de las formas de especificar esto en Palabos es mediante un fichero STL. Suministrándole un cuerpo cerrado, en formato STL, Palabos identifica qué nodos quedan dentro del volumen de dicho cuerpo, y les asocia los parámetros necesarios para que actúen como un sólido.

Para definir las fronteras exteriores del fluido hay dos opciones. La más simple consiste en establecer, directamente, las condiciones en los límites del bloque lattice, o suministrar una frontera con una superficie definida mediante un fichero STL, de manera que solo los nodos que queden en el interior del volumen del STL se considerarán pertenecientes al fluido.

Palabos ofrece también la opción de crear las llamadas *Sponge Zones*. Esto consiste en asignar una dinámica especial a los nodos del fluido próximo a las condiciones de contorno. De este manera se amortiguan ciertas perturbaciones que tienden a producirse en estas zonas, como por ejemplo ondas de presión que viajan por el dominio debido a que el tratamiento numérico del flujo es compresible.

2.5 El Túnel de Viento Virtual

Para poner a punto el Túnel de Viento Virtual del presente Trabajo Fin de Grado, se partió de uno de los ejemplos que ofrece Palabos, que incorpora algunas funcionalidades que se buscaban, que permite importar un fichero STL y exportar los datos de la simulación en VTK.

Para dotarlo de más utilidad se modificaron las siguientes funciones:

- Fueron eliminadas partes del código que no eran necesarias, como la exportación de los resultados en formatos que no resultaran útiles.
- Se cambió la manera de definir los parámetros de la simulación. En el nuevo sistema se definen la discretización espacial δ_x y temporal δ_t, parámetros básicos para realizar el cambio al sistema discreto.

  El tiempo durante el cual la velocidad de entrada se incrementa progresivamente, hasta alcanzar el valor final, ha pasado de expresarse en función a la iteración en la que se alcanzaba dicho valor a indicarse en momento en unidades de tiempo.
- Ajuste de los datos que se exportan de la simulación. Resultaba interesante exportar las tres componentes de la fuerzas a las que se ve sometido el sólido. Se decidió utilizar un fichero en formato CSV, donde se van volcando cada una de estas componentes conforme transcurre el tiempo de la simulación.
- Posibilidad de cambiar la densidad. Dada la vital importancia de que la densidad, durante la simulación, permanezca cercana a la unidad, en el código original se establecía dicho valor y no existía posibilidad de cambiarla. Se modifico el túnel para que realizara, de manera automática, un proceso de adimensionalización de la densidad y, posteriormente, reajustaba los valores obtenidos, de manera que fuera transparente al usuario.
- Nuevas dinámicas. Los modelos que inicialmente incorporaba el código presentaban problemas de estabilidad para números de Reynolds altos. Tras estudiar los modelos que proporcionaba Palabos se incorporaron los modelos *Regularized BGK*, *Multiple Relaxation Time* y *Entropic model*. Los tres proporcionan mejores resultados para números de Reynolds altos.
- Creación de una interfaz gráfica. El uso de Palabos es complejo. Los parámetros de configuración se indican en un fichero XML, que hay que modificar manualmente, y su ejecución es mediante línea de comandos a través del programa mpirun. Para facilitar su uso se

ha programado una interfaz con la librería de Qt para Python, PyQt. Desde esta interfaz se pueden establecer todos los parámetros y, posteriormente, lanzar la simulación.

- Nuevas condiciones de contorno. Existe interés en realizar simulaciones donde influya la contribución del suelo al flujo y a la aerodinámica. Se añadió la posibilidad de fijar, en una de las caras del túnel, condiciones de contorno para una pared sólida.

- Incorporada funcionalidad para rotar el objeto entre simulaciones. Con el formato original del programa, no se disponía de ninguna funcionalidad que permitiese realizar giros al cuerpo, de manera autónoma, sin recurrir a un software externo. Gracias a los cambios realizados, no solo es posible orientar el cuerpo antes de empezar la simulación, también se ha desarrollado un mecanismo que permite realizar giros consecutivos, de manera automática, entre simulaciones.

- Sistema de avisos. Como una simulación puede durar un tiempo considerable, se ha incorporado un sistema de avisos que permite al programa enviar un mensaje cuando la simulación haya finalizado. Este mensaje puede enviarse de dos maneras: mediante un correo electrónico a la cuenta que se desee, o enviando un mensaje a la aplicación de mensajería instantánea Telegram.

2.5.1 Ciclo de uso del Túnel de Viento Virtual

El ciclo de uso del Túnel de Viento Virtual es como sigue:

- Se parte de un fichero STL que tenga el modelo del cuerpo sólido. Si el modelo se encontrase en otro formato CAD, es posible cambiarlo a formato STL con prácticamente cualquier software CAD disponible en el mercado. Si la malla tuviese errores, estos son fácilmente reparables siguiendo las instrucciones del apartado 2.7.

- Una vez se dispone del sólido, hay que escoger los parámetros de la simulación. En la sección 2.6.1 se indican detalles.

- Finalmente, se lanza la simulación en el túnel de viento virtual. La guía para el uso de este programa se puede consultar en la sección 2.5.2.

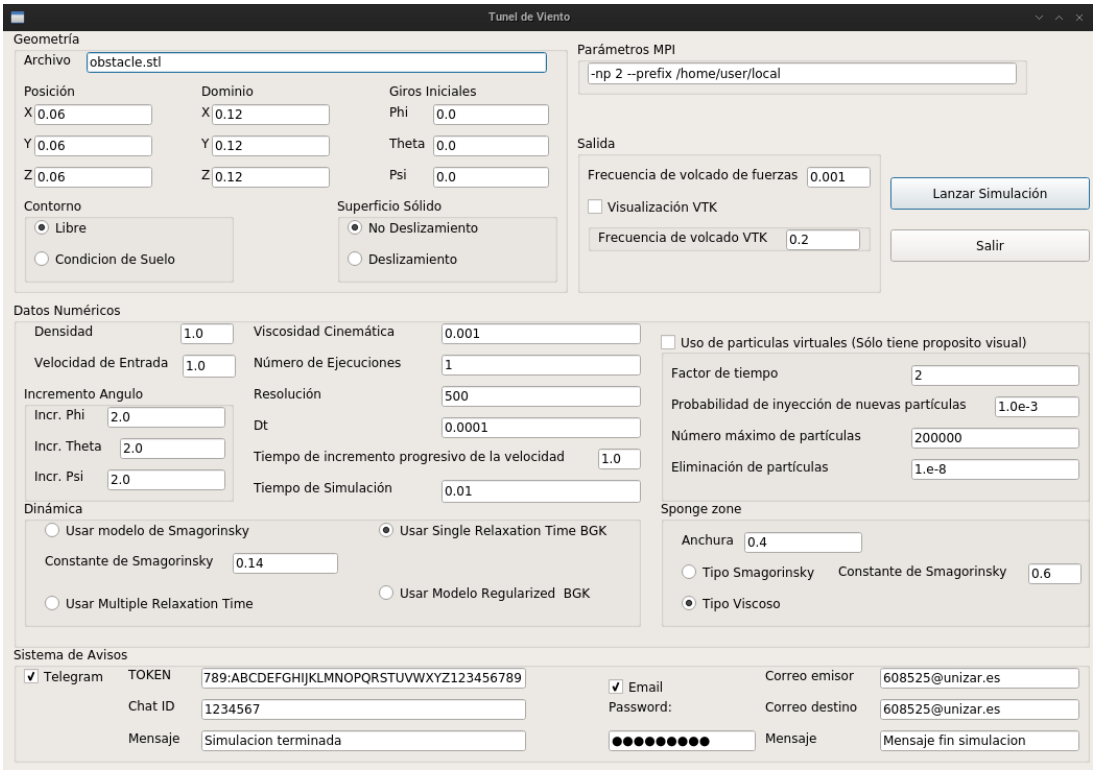


Figura 2.1: Túnel de Viento: Interfaz Gráfica.

2.5.2 Guía de uso

Para comenzar a utilizar el software hay que ejecutar la interfaz gráfica, la cual se muestra en la figura 2.1.

En ella podemos ver todas las opciones de configuración que permite el túnel.

En la opción de “Archivo” tenemos que indicar el nombre que tiene el archivo STL que contiene el sólido.

Las dimensiones del túnel se especifican en la opción de “Dominio”. La unidad de medida utilizada para referenciar la longitud del problema tiene que ser la misma en el fichero STL y la usada en los valores introducidos en la interfaz.

El fichero STL tiene un origen de coordenadas respecto al cual se definen los distintos vértices, como se podrá ver en el apartado 2.7. Para posicionar el cuerpo en el túnel se especifica el lugar que va a ocupar dicho origen de coordenadas en el túnel en la opción “Posición”.

El túnel permite establecer giros iniciales al sólido:

- “Phi”: Rotación en el eje X.

- "Theta": Rotación en el eje Y.
- "Psi": Rotación en el eje Z.

Hay dos posibles condiciones de contorno para las paredes laterales:

- "Libre" donde la componente normal de la velocidad en la frontera es nula, ecuación 2.8, en las cuatro paredes laterales del túnel.
- "Condiciones de Suelo" tres paredes del túnel tienen componente normal de la velocidad nula, ecuación 2.8 de nuevo, y en la cuarta, correspondiente al plano $Y = 0$, la velocidad tiene un valor nulo en todas sus componentes, que es la condición de no deslizamiento, ecuación 2.9.

$$\vec{u}_n = 0 \tag{2.8}$$

$$\vec{u} = \vec{0} \tag{2.9}$$

Condiciones similares son las que se pueden aplicar a la superficie del cuerpo:

- "Deslizamiento" la componente normal de la velocidad con el sólido es nula, ecuación 2.8.
- "No Deslizamiento" la velocidad en el contorno del sólido es nula, ecuación 2.9.

Los valores de densidad y viscosidad cinemática se especifican en los campos con sus respectivos nombres.

La velocidad del fluido no perturbado se especifica en el campo "Velocidad de Entrada".

Para los casos en los que se desea realizar varias simulaciones en las cuales se modifiquen los ángulos del sólido, hay que indicar el incremento del ángulo en cada simulaciñon en cada una de las opciones de "Incremento Ángulo". El eje que corresponde a cada giro ya ha sido explicado anteriormente. El número total de simulaciones que se van a realizar se indica en la opción "Número de Ejecuciones".

En la versión actual del túnel, el mallado del dominio se realiza con una resolución constante, como es el caso en la mayor parte de los métodos de Lattice Boltzmann. El tamaño de la

discretización espacial δ_x se indica en la opción "Resolución" el número de particiones que se van a realizar a lo largo del eje Y. De esta manera se obtiene la unidad básica de distancia en el sistema Lattice-Boltzmann:

$$\delta_x = \frac{l_y}{r} \tag{2.10}$$

donde

- l_y es la altura del túnel en el eje Y.
- r es la resolución, número de nodos a lo largo del eje Y.

La discretización temporal, δ_t, se indica en la opción "Dt".

Durante la simulación, la velocidad en la entrada se incrementa progresivamente siguiendo la ecuación:

$$V(t) = \begin{cases} U \sin(\frac{\pi t}{2T}) & \text{si} \quad t \leq T \\ U & \text{si} \quad t > T \end{cases} \tag{2.11}$$

donde

- $V(t)$ es la velocidad de entrada del fluido en el instante t.
- U es el valor indicado en la opción "Velocidad de Entrada".
- T es el valor indicado en la opción "Tiempo de incremento progresivo de la velocidad".

Una vez alcanzado el valor de U se mantiene dicha velocidad hasta finalizar la simulación. La duración de la simulación se indica en "Tiempo de Simulación".

Para la simulación del flujo, el método de Lattice-Boltzmann implementado dispone de cuatro dinámicas distintas.

- Para simulaciones con un número de Reynolds bajo se recomienda utilizar el modelo *Single Relaxation Time BGK*.
- Para flujos turbulentos, se utiliza el modelo de turbulencia de*Smagorinsky*. El valor de la constante de Smagorinsky se indica en el campo "Constante de Smagorinsky".

- Los modelos *Regularized BGK* y *Entropic* son modificaciones de *Single Relaxation Time BGK*, que dan mejores resultados para números de Reynolds altos.

Si se desea incorporar una *Sponge Zone* en la parte final del túnel, se debe indicar su espesor en unidades físicas en el campo "Anchura". Si no se desea utilizar, es suficiente con dejar este campo a cero. En caso de que se use, hay que especificar la dinámica utilizada en ella, "Tipo Viscoso" para que se emplee una dinámica *Single Relaxation Time BGK* o "Tipo Smagorinsky" para que sea utilizada una tipo *Smagorinsky*. En este último caso, hay que indicar el valor de la constante de Smagorinsky en el campo "Constante de Smagorinsky".

Como resultado de la simulación, el túnel virtual exporta por un lado las fuerzas que experimenta el sólido, en un archivo con formato CSV, y, por otro lado, el estado del fluido en el túnel en formato VTK, donde se almacena la velocidad y la presión en cada nodo.

Si se desea exportar la solución, se debe marcar la opción "Visualización VTK", e indicar la frecuencia de volcado en "Frecuencia de volcado VTK". Si se desea crear líneas de corriente, se debe utilizar el sistema de "Partículas Virtuales". El "Factor de tiempo" indica el factor de iteración de las partículas virtuales frente al fluido. Un valor recomendable para este factor es de dos, de manera que por cada dos iteraciones en la simulación del fluido se hace una iteración para calcular el flujo de las partículas virtuales.

Para mejorar la calidad del resultado, una técnica que se utiliza es la de inyectar nuevas partículas virtuales y la de eliminar las partículas más lentas. El factor indicado en "Probabilidad de inyección de nuevas partículas" será la probabilidad de crear una partícula virtual nueva en cada uno de los nodos de la malla en cada iteración del sistema de partículas virtuales. En la opción de "Eliminación de partículas" se indica la velocidad, tal que, toda partícula cuya velocidad al cuadrado sea inferior a este valor, será eliminada del sistema. La última opción "Número máximo de partículas" indica la cantidad de partículas virtuales máximas a escribir en el fichero VTK.

Dada la variedad de parámetros que permite MPI, se decidió dejar un campo de texto donde escribir que parámetros que el interfaz para a MPI en el comando de ejecución.

Una vez finaliza la simulación el programa llama al sistema de avisos. Si se desea recibir un aviso, por cualquiera de los dos medios disponibles, es necesario indicarlo activando la checkbox que corresponda.

Para poder recibir un mensaje en Telegram, se tiene que haber creado un bot en dicha aplicación. Una vez creado el bot, se indica el token necesario para utilizarlo en el campo "Token", e indicar el chat_id asociado a la conversación a la que enviar el mensaje en el campo "Chat id".

En caso de que no se disponga de Telegram, es posible avisar mediante un correo electrónico. Para ello solo hay que indicar una dirección de origen desde la que enviar el correo en el campo "Correo emisor", la contraseña de dicho correo en "Password", el mensaje a enviar en "Mensaje" y la dirección de correo al que enviar dicho mensaje en "Correo destino".

Por motivos de seguridad, la contraseña para iniciar sesión no es almacenada en el disco. De igual modo, la conexión con el servidor se realiza encriptada bajo el protocolo TLS.

2.6 Parámetros de una simulación

A día de hoy, no existe un método establecido con el que determinar el valor a asignar a los diferentes parámetros en una simulación por Lattice-Boltzmann. Sí existen diversas reglas que se tienen que cumplir de manera obligada por la naturaleza del método, y ciertas recomendaciones para conseguir que el método tenga una mayor estabilidad.

2.6.1 Reglas para la selección de parámetros

La notación utilizada en este punto está explicada en Apéndice A.4.

Los valores de viscosidad cinemática y velocidad del fluido se establecen mediante las ecuaciones:

$$\nu_{lb} = \nu_d \frac{\delta_t}{\delta_x^2} \tag{2.12}$$

$$\nu_{lb} = (\tau - \frac{1}{2})\frac{1}{3} \tag{2.13}$$

$$u_{lb} = u_d \frac{\delta_t}{\delta_x} \tag{2.14}$$

Como la viscosidad tiene que ser positiva, $\nu_{lb} > 0$, esto obliga a seleccionar $\tau > 0{,}5$. Se ha comprobado que cuanto más cerca se encuentre el parámetro τ de 0.5 mayores problemas de estabilidad presenta el método. Un valor orientativo es tomar $\nu_{lb} \sim 0{,}55$.

El método Lattice-Boltzmann es para flujo subsónico. Esto lleva a la condición:

$$u_{lb} < c_s \longrightarrow u_{lb} < \frac{1}{\sqrt{3}} \tag{2.15}$$

La influencia de la velocidad en el error restringe el valor que ésta puede tomar sin que el error se dispare. Un valor orientativo para la velocidad es $u_d \sim 0{,}25$

El parámetro δ_x es la distancia entre nodos en la discretización. El valor de δ_x tiene que ser seleccionado de manera que todos los detalles del cuerpo queden suficientemente resueltos.

Una relación recomendada para elegir δ_t es:

$$\delta_t \sim \delta_x^2 \tag{2.16}$$

2.6.2 Análisis del error

La precisión del método viene determinada por los parámetros τ, δ_x, δ_t y el número de Mach de la simulación.

La influencia de τ es compleja, dado que su valor incide en el número de Reynolds y por lo tanto puede cambiar la estructura del flujo. Para un valor del número de Reynolds, el valor del factor τ puede mejorar la precisión. Se ha comprobado que los mejores resultados se obtienen para valores $\tau \sim 0{,}9$ (Krüger et al. 2009).

El factor más influyente es el número de Mach de la simulación:

$$Ma = \frac{u_{lb}}{c_s} \tag{2.17}$$

Se ha comprobado que el error del método aumenta considerablemente conforme aumenta el número de Mach de la simulación.

La influencia de δ_x y δ_t en el error se puede dividir en las tres partes:

$$\varepsilon \sim \varepsilon(\delta_x) + \varepsilon(\delta_t) + \varepsilon(Ma) \tag{2.18}$$

donde:

- $\varepsilon(\delta_x)$ error relativo al parámetro δ_x. Lattice-Boltzmann es un método de orden cuadrático, por lo que $\varepsilon(\delta_x) \sim O(\delta_x^2)$.
- $\varepsilon(\delta_t)$ error relativo al parámetro $\delta_t x$. De nuevo, al ser un método de orden cuadrático tenemos $\varepsilon(\delta_t) \sim O(\delta_t^2)$.
- $\varepsilon(Ma)$ error relativo a la compresibilidad. En este caso tiene un orden $\varepsilon(Ma) \sim O(u_{lb}^2)$.

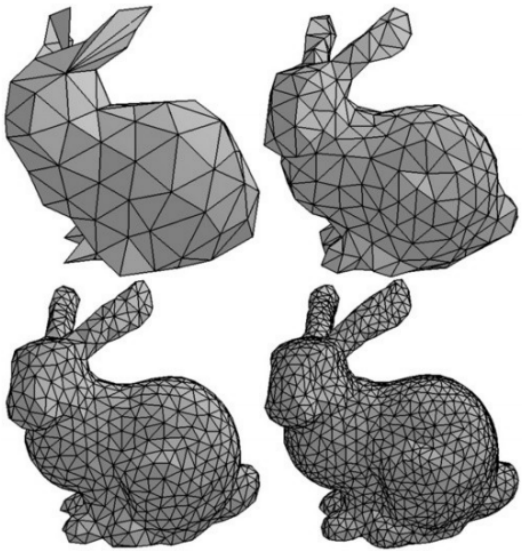


Figura 2.2: Resolución STL (Touretzky 2017).

Con estos órdenes, el orden del error total es:

$$\varepsilon \sim O(\delta_x^2) + O(\delta_t^2) + O(u_{lb}^2) \tag{2.19}$$

Una buena idea es mantener el error relativo a la compresibilidad (número de Mach) del mismo orden que el relativo al parámetro δ_x. Para conseguir esto, basta con cumplir la relación anteriormente citada 2.16.

2.7 Formato STL

STL es el formato para importar sólidos 3D al túnel virtual. STL es un formato abierto, usado en el diseño asistido por ordenador, que permite definir superficies mediante un conjunto de triángulos que se ajustan al contorno de la superficie.

Para ello se definen las caras triangulares según los vértices que las forman, especificando las coordenadas cartesianas de cada uno de estos vértices. Todo ello es almacenado en formato de texto plano.

Esta técnica obliga a que, si se quiere una mayor precisión en la descripción de la superficie, se tienen que utilizar triángulos de menor tamaño, como se ilustra en la figura 2.2.

Los sólidos representados de esta manera pueden presentar errores que impidan que se pueda

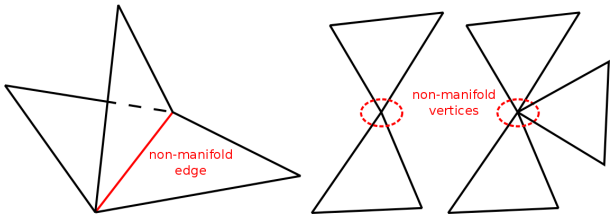


Figura 2.3: Non Manifold Edge y Non Manifold Vertices (*Non-Manifold edge and vertices* 2017).

trabajar con ellos. Algunos de los más comunes son:

- Huecos. Son errores en el proceso de transformación de una superficie a triángulo, dando lugar a huecos donde debería haber superficie mallada.
- Gaps. Errores en la unión entre los diferentes triángulos que forman la malla.
- Non Manifold Edges. Aristas que son comunes a más de un triángulo de la malla. Un ejemplo se puede ver en la figura 2.3.
- Non Manifold Vertices. Vértices que son comunes para más de una superficie al mismo tiempo. Un ejemplo se puede ver en la figura 2.3 .

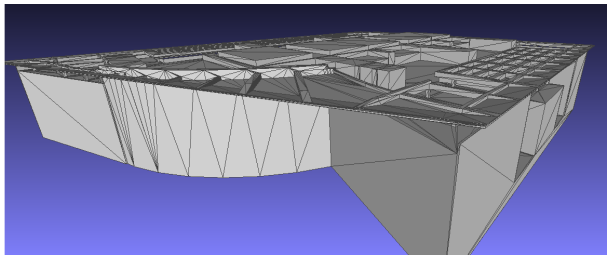
Si inicialmente se dispone del modelo que se desea simular en un formato nativo de un software CAD, conviene realizar la exportación atendiendo a los distintos mallados que el software permita y realizar el que genere una malla sin ningún defecto o con el menor número de defectos posibles.

Para el presente proyecto no se dispuso de ningún modelo previo en STL, y fue necesario generar dichos modelos y realizar un proceso de reparación de algunas de las mallas utilizadas.

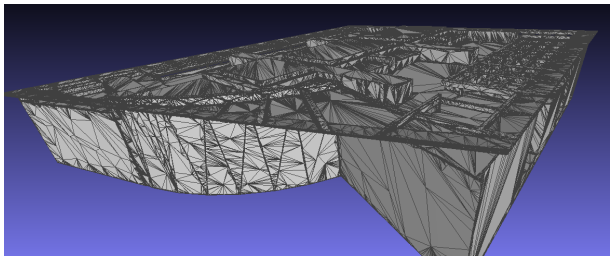
Para la reparación de las mallas dañadas se dispuso de diversas soluciones de software que realizan una reparación automática de los problemas. Las soluciones probadas fueron:

- **SpaceClaim** perteneciente al paquete de software ANSYS.
- **Netfabb** de la casa Autodesk.

A pesar de la potencia que presentan los algoritmos de reparación de las citadas soluciones, es posible que el mallado de los sólidos no permita una reparación automática. Esto suele presentarse en objetos con configuraciones complicadas o mallados con poca resolución. En estos



(a) STL con mallado inicial



(b) STL con nuevo mallado

Figura 2.4: Ejemplo de remallado más fino.

casos fue necesario realizar un remallado a otro que sí permitiese la reparación, como se puede ver en la figura 2.4.

Para la realización de nuevos mallados, se pueden utilizar las soluciones de software:

- **ZBrush** de Pixologic.
- **3DS MAX** de Autodesk.

CAPÍTULO 3

Simulaciones típicas

Para comprobar el buen funcionamiento del Túnel de Viento Virtual, se ha realizado un conjunto de simulaciones de la aerodinámica de diversos cuerpos. En todas ellas el número de Reynolds es bajo, entre uno y cien. Como queda dicho, simulaciones a más alto número de Reynolds no suponen ninguna dificultad conceptual u operativa adicional; simplemente, requieren mayor potencia de cálculo de la que está fácilmente disponible para un Trabajo Fin de Grado.

Inicialmente, se ha realizado un estudio de la variación de la precisión del método respecto a los parámetros de la simulación. Igualmente, se ha comprobado cómo mejora dicha precisión al utilizar las dinámicas *Regularized BGK* y *Multiple Relaxation Time* frente a la dinámica *Single Relaxation Time BGK*.

Posteriormente, se ha realizado el estudio de dos perfiles alares. En ellos se comprueba la variación del coeficiente de arrastre respecto al ángulo de ataque para dos números de Reynolds.

A continuación, se ha realizado el análisis aerodinámico de dos geometrías representativas de objetos completos: una motocicleta y un edificio. Se analizan las líneas de corriente y los campos de presión y velocidad alrededor de los cuerpos.

En ninguna de las magnitudes se indican unidades al presentar resultados, ya que puede utilizarse cualquier conjunto de unidades siempre y cuando sea consistente, por ejemplo el sistema internacional o el sistema cegesimal.

Por último, se presenta un breve estudio de tiempos de cálculo.

3.1 Análisis numérico de un problema canónico: Flujo alrededor de una esfera

Sobre este caso, geométricamente sencillo, se ha estudiado la influencia de los parámetros de la simulación: velocidad del fluido, viscosidad, δ_t y δ_x en el error de la simulación. Igualmente, se quiere comprobar la mejora que suponen las dinámicas *Multiple Relaxation Time* y *Regularized BGK* frente a *Single Relaxation Time BGK* en la precisión del método. Para ello, todas las simulaciones han sido realizadas con las tres dinámicas y los errores han sido comparados.

Para el análisis, es necesario acudir a un caso ampliamente estudiado con el que contrastar los resultados de las simulaciones. Se ha elegido el caso de una esfera alrededor de la cual fluye un fluido. El fichero STL de dicha esfera estaba incluido en la instalación de Palabos.

Para medir la precisión, se ha utiliza como referencia el coeficiente de arrastre de la esfera, el cual se calcula como:

$$C_d = \frac{F_d}{\frac{1}{2}\rho u^2 A} \tag{3.1}$$

donde

- C_d es el coeficiente de arrastre.
- F_d es la fuerza medida en la dirección de flujo del fluido.
- ρ es la densidad del fluido.
- u es la velocidad del fluido.
- A es el área de referencia. Para el caso de la esfera de radio r es $A = \pi r^2$.

El error se expresa mediante la ecuación:

$$Error = \frac{C_d^a - C_d^s}{C_d^a} \tag{3.2}$$

donde C_d^d es el coeficiente de arrastre obtenido en la simulación, y C_d^a es el coeficiente de arrastre de referencia, extraído de la figura 3.1.

La figura 3.1 tiene en su eje de abscisas el número de Reynolds, *Re*, el cual puede calcularse con la ecuación:

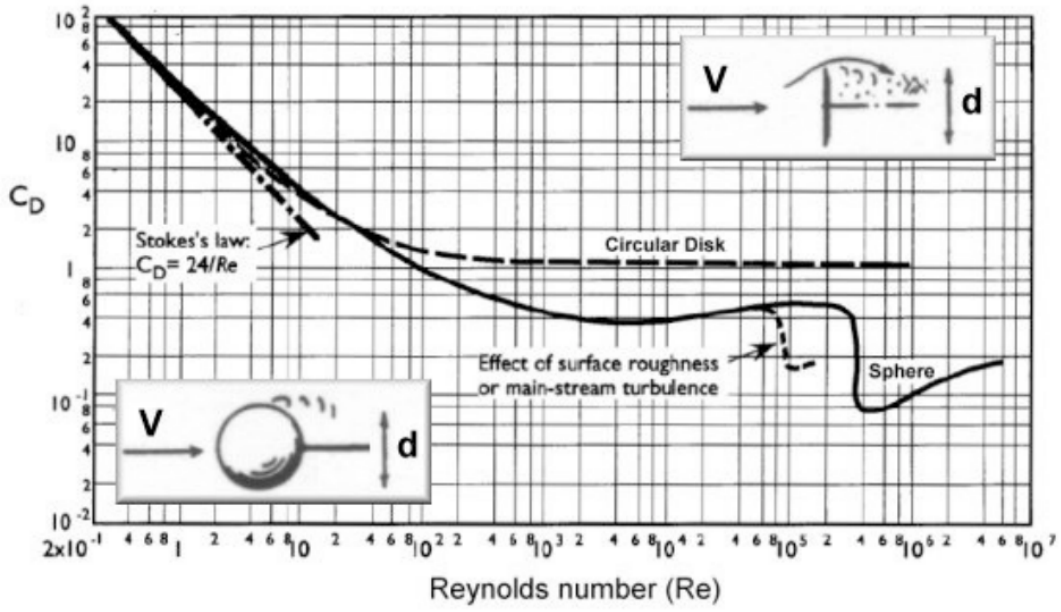


Figura 3.1: Análisis numérico: Cd frente a Re (Dosselaer 2014).

$$Re = \frac{ul}{\nu} \tag{3.3}$$

donde

- l es la longitud de referencia. Para el caso de una esfera es su diámetro.
- u es la velocidad del flujo no perturbado.
- ν es la viscosidad del fluido.

En todas las simulaciones de esta sección se ha utilizado una esfera de diámetro 0.2. Esta esfera es colocada en un punto equidistante de las paredes del túnel, y lo suficientemente alejada de estas como para que su influencia no altere el flujo. Del mismo modo, la esfera se ha posicionado cerca de la entrada del túnel, para dejar espacio tras ella, donde puedan desarrollarse los torbellinos en caso de producirse.

Se ha utilizado un valor de densidad de 1,0.

Las condiciones de contorno utilizadas han sido:

- En la superficie de la esfera, las condición denominada “No Deslizamiento” en la interfaz, que impone un valor de la velocidad nulo en la superficie de la esfera.

Parámetro	Valor
Giros iniciales	0 en cada uno de ellos
Contorno	Libre
Superficie sólido	No deslizamiento
Densidad	1,0
Número de ejecuciones	1
Tiempo de incremento progresivo de la velocidad	1
Tiempo de la simulación	7,5
Incremento de ángulo	0,0 en cada uno de ellos
Anchura de la Sponge Zone	0
Frecuencia del volcado de fuerzas	0,01
Dominio X	2,95
Dominio Y	0,9
Dominio Z	0,9

Cuadro 3.1: Análisis numérico: Inputs en la interfaz gráfica.

- En las paredes del túnel, se han usado las condiciones denominadas “Libres” en la interfaz. Éstas imponen en la velocidad un gradiente nulo en la componente tangencial a la frontera, y un valor nulo de la velocidad en la componente normal a dicha superficie.

Estos parámetros son comunes para todas las simulaciones de este apartado. En la tabla 3.1 se pueden ver todos los valores comunes, expresados con la nomenclatura utilizada en la interfaz del programa.

3.1.1 Influencia de la velocidad del fluido

En este apartado se estudia la influencia de la velocidad en el error del método. La velocidad tiene una influencia directa en el número de Mach de la simulación, y por lo tanto en el error relativo a la compresibilidad de la ecuación 2.18.

Para comprobar la influencia de la velocidad, se han realizado varias simulaciones dejando fijos todos los parámetros menos la velocidad, u_{lb}, que se ha variado entre 0,0015 y 0,1067.

	Error (%)		
Velocidad	*Single Relaxation Time BGK*	*Regularized BGK*	*Multiple Relaxation Time*
0,0016	−9,74	−9,65	−9,59
0,0022	−2,23	−2,14	−2,01
0,0044	−20,11	−19,99	−19,53
0,0089	−24,15	−23,98	−22,90
0,0178	−48,57	−48,21	−45,55
0,0356	−87,22	−86,44	−79,98
0,0711	−737,83	−734,69	−681,82
0,1067	−1650,67	−1659,46	−1503,28

Cuadro 3.2: Análisis numérico: Influencia de la velocidad en la precisión.

En todos estos casos, la viscosidad cinemática tiene un valor de 0,02109, δ_x adquiere el valor 0,01125, y δ_t se establece en 0,0001.

El error ha obtenido es mostrado en el cuadro 3.2.

En la figura 3.2 se representa el error, en tanto por uno, en forma de gráfica de puntos.

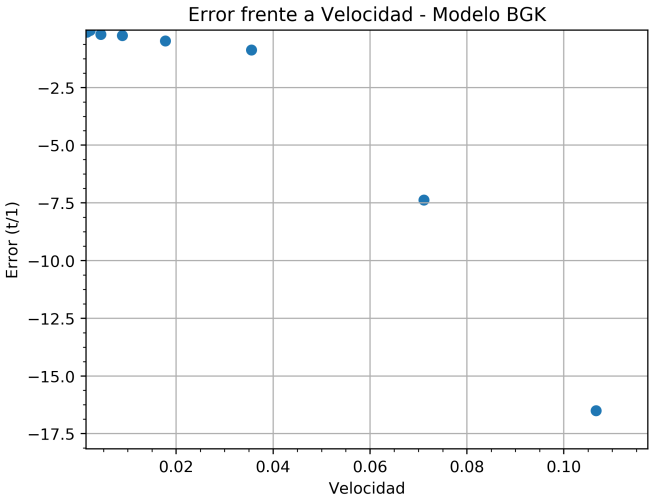
Si se comparan las tres dinámicas, se ve que, si bien existe una ligera mejora, en este caso es irrelevante.

Si se quiere mantener el error del método dentro de un orden aceptable, un valor recomendable para la velocidad es $u_{lb} = 0{,}002$. En caso de que la velocidad que se desea simular sea elevada, y no sea posible utilizar unos valores de δ_x y δ_t suficientemente bajos como para reducir el error de la compresibilidad, es recomendable realizar un proceso de adimensionalización como el explicado en el Apéndice A.4.

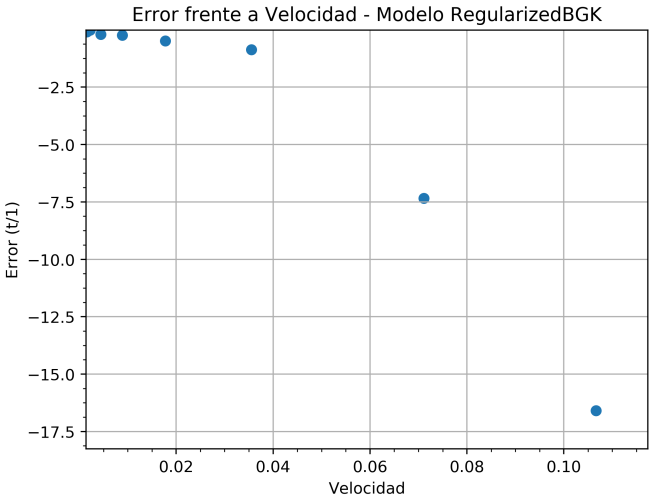
3.1.2 Influencia de la viscosidad

La viscosidad influye directamente en el parámetro de relajación, τ, y, por lo tanto, en la precisión del método.

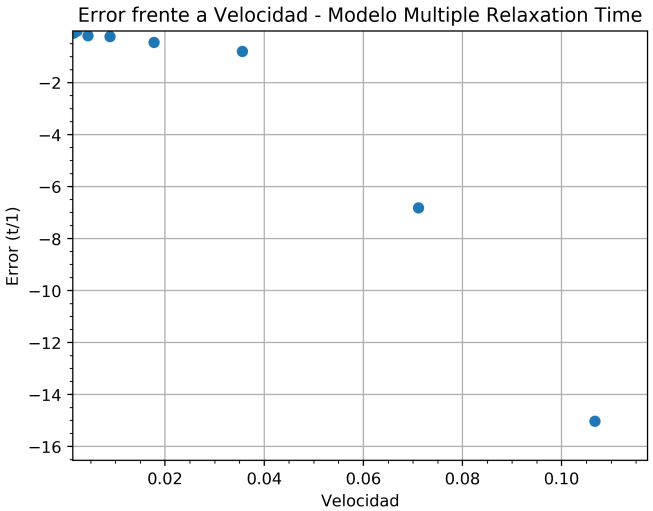
Para estudiar su influencia se procede de la misma manera que en el caso anterior, se realiza la misma simulación manteniendo fijos todos los parámetros menos la viscosidad, la cual se ha variado entre 0,005 y 0,021.



(a) Dinámica BGK



(b) Dinámica Regularized BGK



(c) Dinámica Multiple Relaxation Time

Figura 3.2: Análisis numérico: Error frente a velocidad.

	Error (%)		
Viscosidad	*Single Relaxation Time BGK*	*Regularized BGK*	*Multiple Relaxation Time*
0,005	−25,22	−25,45	−24,79
0,007	−20,77	−20,91	−20,41
0,009	−19,62	−19,70	−19,29
0,011	−18,55	−18,58	−18,23
0,013	−12,93	−12,92	−12,64
0,015	−7,57	−7,54	−7,31
0,017	−4,13	−4,08	−3,89
0,019	−2,50	−2,43	−2,27
0,021	−2,23	−2,14	−2,01

Cuadro 3.3: Análisis numérico: Error frente a viscosidad.

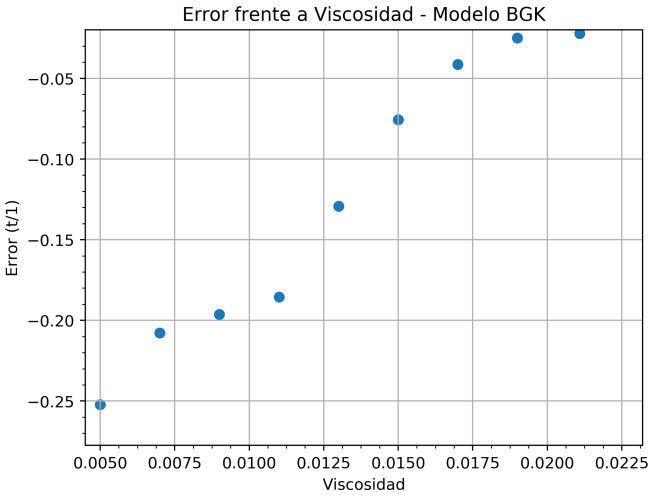
Los parámetros δ_x y δ_t toman un valor de 0,01125 y 0,0001 respectivamente. Para la velocidad de entrada, se utiliza el valor con el que menor error se obtuvo en el apartado anterior $u_d = 0{,}25$.

El error obtenido se muestra en el cuadro 3.3. Para facilitar su visualización se representa en forma gráfica en la figura 3.3.

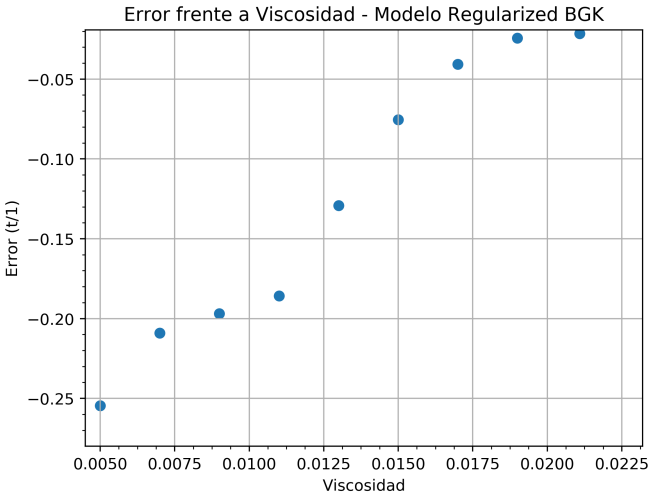
Para valores bajos de la viscosidad, y por lo tanto de τ, el error es elevado. Sin embargo, conforme τ crece el error disminuye.

Si de nuevo se comparan los dos métodos, se observa que, para valores bajos de viscosidad, el método *Regularized BGK* ofrece peor precisión que *Single Relaxation Time BGK*. Sin embargo, llegado a un punto, los papeles se invierten, y el método *Regularized BGK* ofrece mejores resultados. La dinámica *Multiple Relaxation Time BGK* proporciona una mejora en la precisión para todos los valores del intervalo. No obstante, como ocurría en el caso de la velocidad, la mejora que nos proporciona es muy poco significativa.

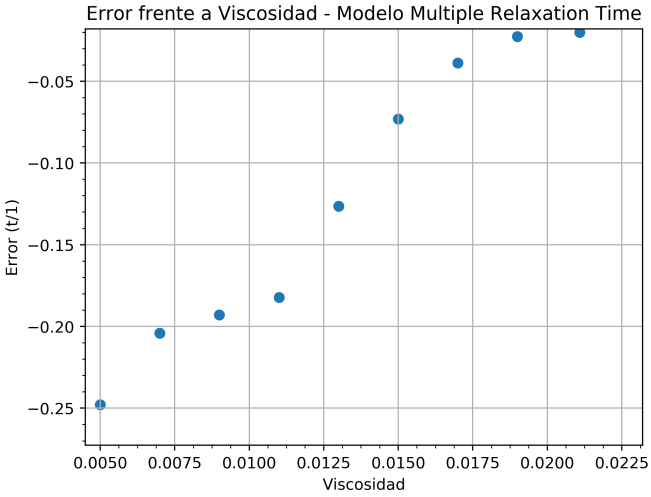
Si se quiere conseguir un error reducido, sería conveniente usar un valor $\nu_p = 0{,}021$. Como en el caso anterior, si la simulación que se desea realizar tiene un valor muy inferior a éste, sería necesario utilizar el proceso de adimensionalización explicado en el apéndice A.4.



(a) Dinámica BGK



(b) Dinámica Regularized BGK



(c) Dinámica Multiple Relaxation Time

Figura 3.3: Análisis numérico: Error frente a viscosidad.

3.1.3 Influencia de los parámetros δ_x y δ_t

Una vez se tiene un valor para la viscosidad, la velocidad del fluido y las dimensiones del problema, queda definido el número de Reynolds, y con él la solución analítica. Sin embargo, aún queda por definir el valor de los incrementos discretos δ_t y δ_x.

Al igual que en los dos casos anteriores, se realizan varias simulaciones manteniendo fijos todos los parámetros salvo el que se desea estudiar.

Se estudia en primer lugar la influencia del parámetro δ_x.

Se han utilizado los valores de velocidad $u_d = 0{,}25$, $\delta_t = 0{,}0001$ y Viscosidad Cinemática de 0,02109. Los valores utilizados para δ_x se han hecho variar entre 0.0225 y 0.0078.

El error obtenido está representado en la gráfica 3.4.

En ellas se pueden identificar dos zonas:

- Para valores reducidos del parámetro, el error permanece acotado en un margen aceptable, una variación en δ_x tiene una influencia moderada en el error.
- Para valores más elevados del parámetro, el error se hace mayor, y una variación en δ_x causa un mayor incremento del error.

A la hora de seleccionar un valor para δ_x es conveniente mantenerlo dentro de la primera zona, en caso contrario la simulación tendrá menor precisión y pueden presentarse problemas de estabilidad. Igualmente, δ_x tiene que seleccionarse atendiendo a los detalles del cuerpo, haciendo el mallado lo suficiente fino como para que todos los detalles queden reflejados en él.

También es importante tener en cuenta que, a mayor número de nodos, mayor coste computacional tiene la simulación.

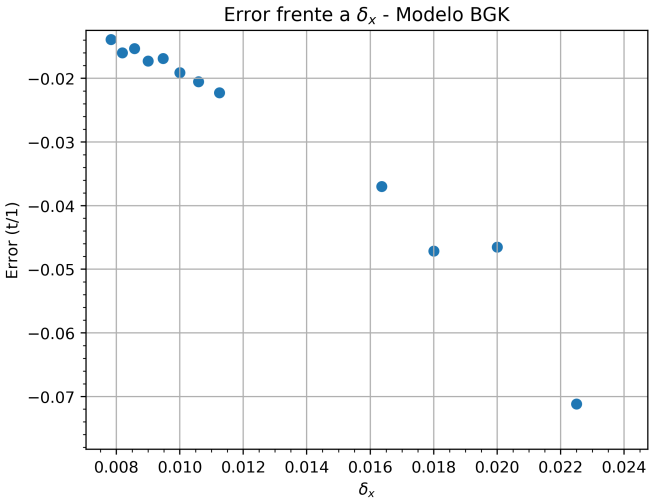
En último lugar se estudia la influencia de δ_t.

Se utilizan los valores $\delta_x = 0{,}01125$, velocidad $u_d = 0{,}25$ y viscosidad cinemática 0,02109.

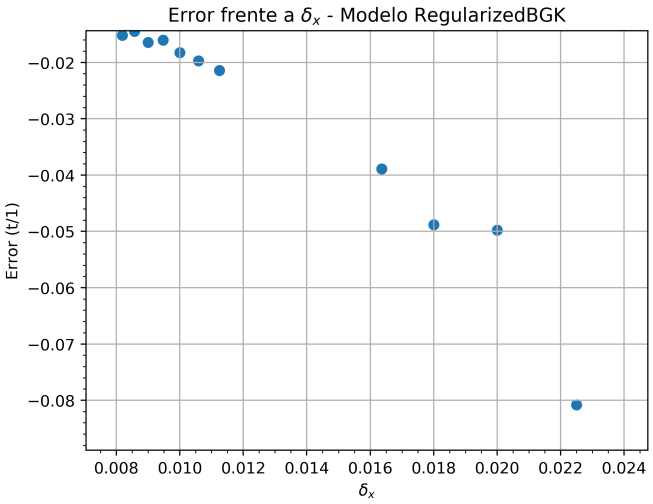
Los valores de δ_t se han hecho variar entre 0,0002 y 0,000025. Como se puede ver, se han elegido los valores llegando a bajar hasta dos órdenes de magnitud el orden recomendado por 2.16.

El error obtenido en estas simulaciones se muestra en la figura 3.5.

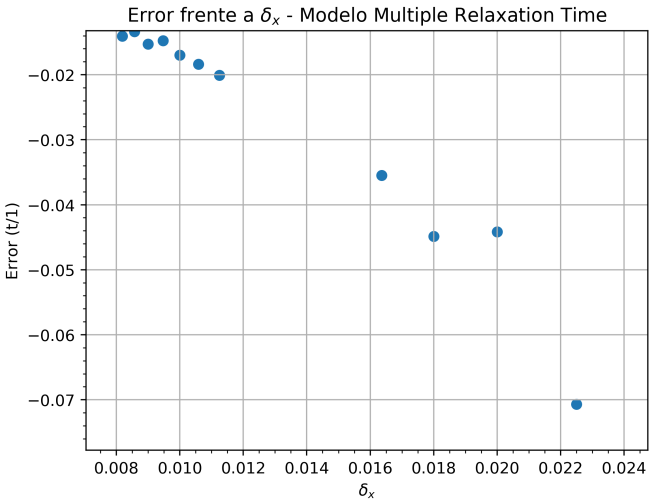
A pesar de haber variado hasta dos órdenes de magnitud el valor de δ_t, el error de la simulación solo ha variado del $-2{,}6\,\%$ al $-1{,}6\,\%$. De esta manera se muestra que resulta poco eficiente intentar mejorar la precisión modificando δ_t, sin modificar a la par δ_x mediante la relación 2.16.



(a) Dinámica BGK

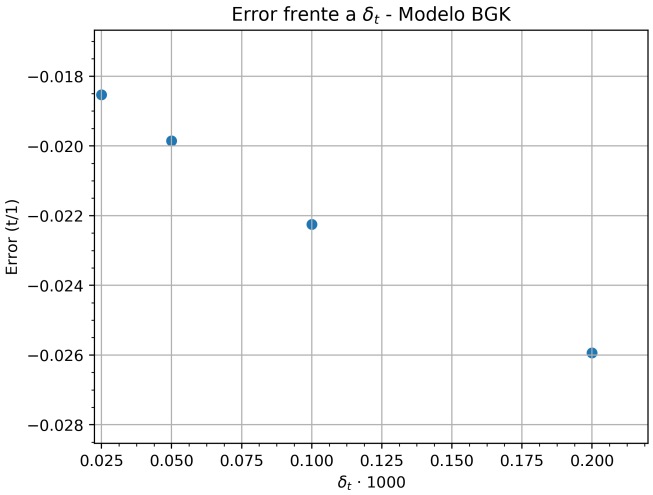


(b) Dinámica Regularized BGK

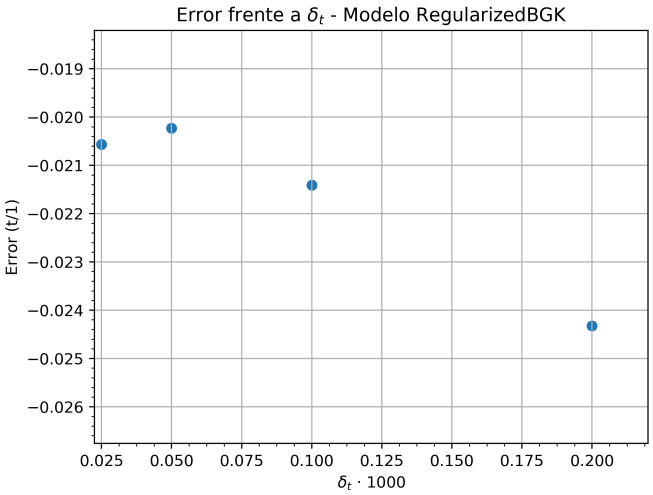


(c) Dinámica Regularized BGK

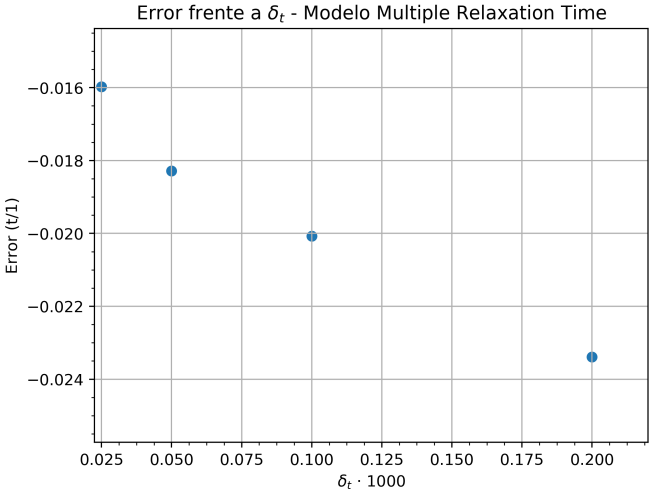
Figura 3.4: Análisis numérico: Error frente a δ_x.



(a) Dinámica BGK



(b) Dinámica Regularized BGK



(c) Dinámica Regularized BGK

Figura 3.5: Análisis numérico: Error frente a δ_t.

Eje	Dimensión Eje - Caso A	Dimensión Eje - Caso B
Eje X	0,9	1,3
Eje Y	0,3	0,4
Eje Z	0,6	0,85

Cuadro 3.4: NACA: Dimensiones Túnel.

Con todo lo expuesto hasta este punto, se ha visto cómo las dinámicas *Multiple Relaxation Time* y *Regularized BGK* han conseguido una mejora en la precisión de método. Sin embargo, esta mejora no supone una diferencia significativa respecto a la tradicional *Single Relaxation Time BGK*.

3.2 Estudio de perfiles alares

Un problema clásico de la aerodinámica es el estudio de perfiles alares. A modo de ejemplo, en esta sección se va a realizar un estudio tipo de este caso.

Se han conseguido ficheros STL para los modelos de perfiles NACA 0021 y NACA 6713[1].

El objetivo de este estudio tipo es obtener la variación de los coeficientes de arrastre y sustentación con el ángulo de ataque, para dos números de Reynolds diferentes.

Para modificar el número de Reynolds sin modificar los valores de la velocidad y viscosidad, evitando así aumentar el error, como se ha observado en el apartado anterior, se han modificado las dimensiones del perfil. Según las dimensiones del perfil se van a diferenciar dos casos, caso A y caso B. Las dimensiones correspondientes a cada caso se especificarán en cada perfil.

Las dimensiones del túnel en cada uno de estos casos se ven en el cuadro 3.4.

Los perfiles han sido situados en el punto medio del túnel en el plano Y-Z. Para el caso A se ha colocado a 0,3 unidades de distancia de la entrada del túnel, mientras que en el caso B a 0,4 unidades.

El parámetro δ_x también se ha modificado entre los casos A y B. Para el caso A se ha usado un valor de $\delta_x = 0{,}00375$ y para el caso B de $\delta_x = 0{,}005$.

El resto de parámetros de la simulación se han mantenido iguales entre los casos A y B, ver cuadro 3.5.

[1] Los STL de dichos perfiles han sido obtenidos de la referencia (*GrabCAD* 2017).

Parámetro	Valor
Giros iniciales	0 en cada uno de ellos
Condición de contorno	Libre
Condición en la superficie del sólido	No Deslizamiento
Densidad	1,0
Número de Ejecuciones	10
Tiempo de incremento progresivo de la velocidad	1
Tiempo de la simulación	7,5
Incremento de Ángulo Psi	0,0873
Incremento de Ángulo Phi	0,0
Incremento de Ángulo Theta	0,0
Anchura de la Sponge Zone	0
Frecuencia del Volcado de las fuerzas	0,01
Viscosidad Cinemática	0,02109
Velocidad	0,25
Modelo	Regularized BGK

Cuadro 3.5: NACA: Inputs en la interfaz gráfica.

Eje	Dimensión Eje - Caso A	Dimensión Eje - Caso B
Eje X	0,1	0,2
Eje Y	0,021	0,042
Eje Z	0,5	0,75

Cuadro 3.6: NACA 0021: Dimensiones Perfiles.

Con estos parámetros se realizan varias simulaciones, variando el ángulo de ataque del perfil 5 grados (0,0873 radianes) entre cada una de ellas.

Se ha usado la ecuación 3.4 para el cálculo del número de Reynolds, 3.5 para el coeficiente de arrastre y 3.6 para el coeficiente de sustentación.

$$Re = \frac{u l_c}{\nu} \tag{3.4}$$

$$C_D = \frac{F_D}{\frac{1}{2}\rho u^2 S} \tag{3.5}$$

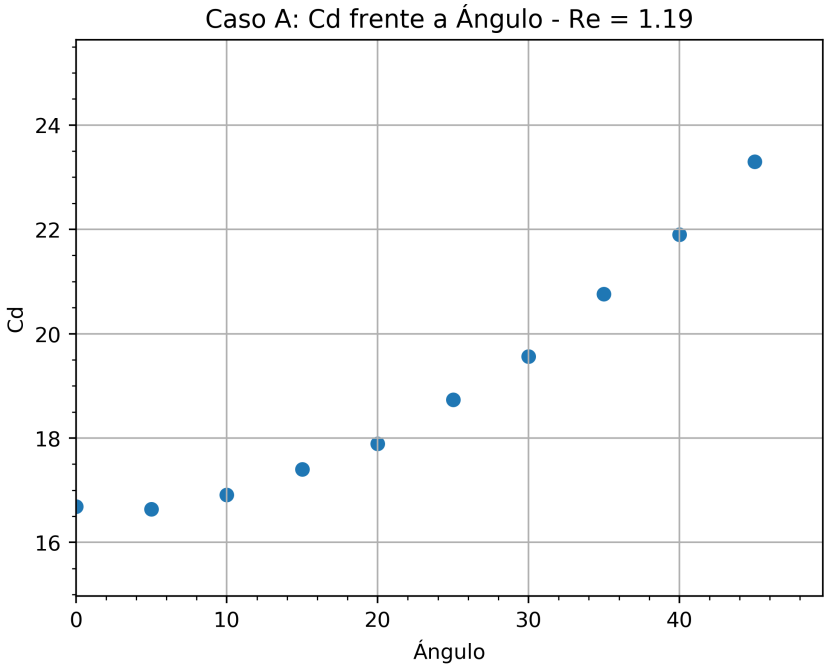
$$C_L = \frac{F_L}{\frac{1}{2}\rho u^2 S} \tag{3.6}$$

donde:

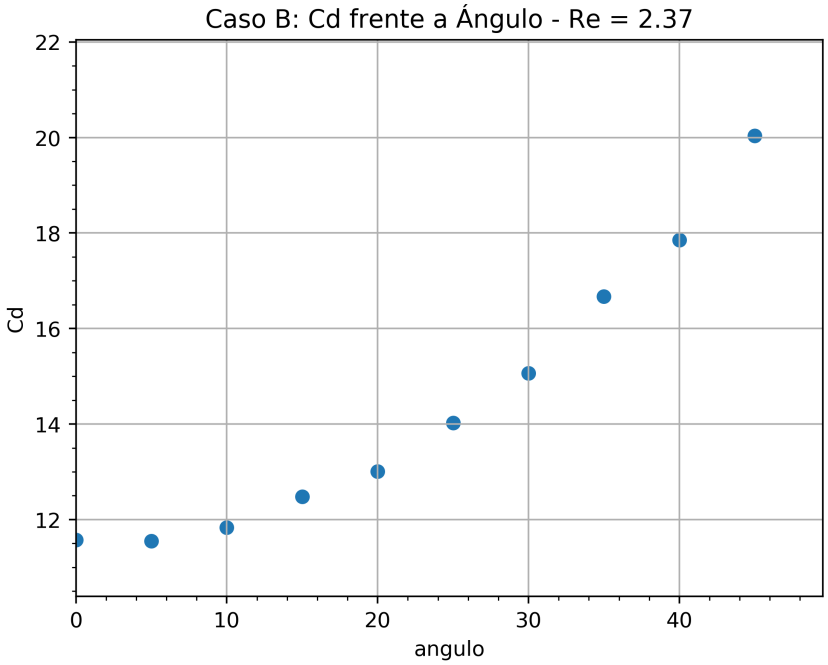
- l_c es la longitud de la cuerda.
- $S = l_c l_a$.
- l_a es la longitud del ala.
- u es la velocidad del fluido sin alterar.
- ρ es la densidad del fluido.
- F_D es la fuerza de arrastre.
- F_L es la fuerza de sustentación.

3.2.1 NACA 0021

Las dimensiones de cada caso pueden verse en el cuadro 3.6.



(a) Caso A



(b) Caso B

Figura 3.6: NACA 0021: Coeficiente de Arrastre frente a ángulo de ataque

Eje	Dimensión Eje - Caso A	Dimensión Eje - Caso B
Eje X	0,1	0,2
Eje Y	0,0176	0,0351
Eje Z	0,5	0,75

Cuadro 3.7: NACA 6713: Dimensiones Perfiles.

Los coeficiente de arrastre obtenidos están en la figura 3.6.

Los coeficiente de sustentación se representan en la figura 3.7.

El comportamiento de este perfil NACA es muy regular: tanto el coeficiente de arrastre como el de sustentación tienen un incremento gradual conforme se incrementa el ángulo de ataque. Al estar trabajando con números de Reynolds bajos, no se llega a alcanzar el ángulo de entrada en pérdida, en el cual el coeficiente de sustentación empezaría a descender.

Los elevados valores que toman el coeficiente de arrastre y el coeficiente de sustentación son debidos, de nuevo, a los bajos números de Reynolds. Para Reynolds suficientemente grandes, los coeficientes se reducirían varios órdenes de magnitud.

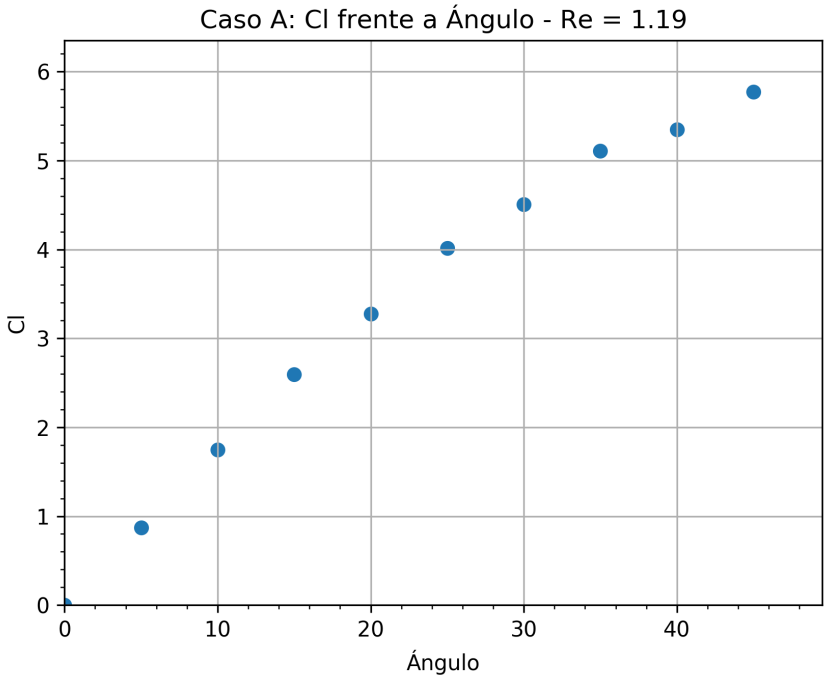
Un fenómeno que tiene lugar en el extremo de los perfiles alares es la creación de vórtices, el ingles llamados *wingtip vortex*. Estos vórtices, típicos de la aerodinámica de perfiles alares, se generan cuando el fluido a alta presión de la parte inferior del ala asciende a la parte superior por el lateral de ésta, debido al gradiente de presiones. A mayor diferencia de presiones, mayor vórtice, por lo que el fenómeno se incrementa al aumentar el ángulo de ataque, como se puede ver en la figura 3.8.

3.2.2 NACA 6713

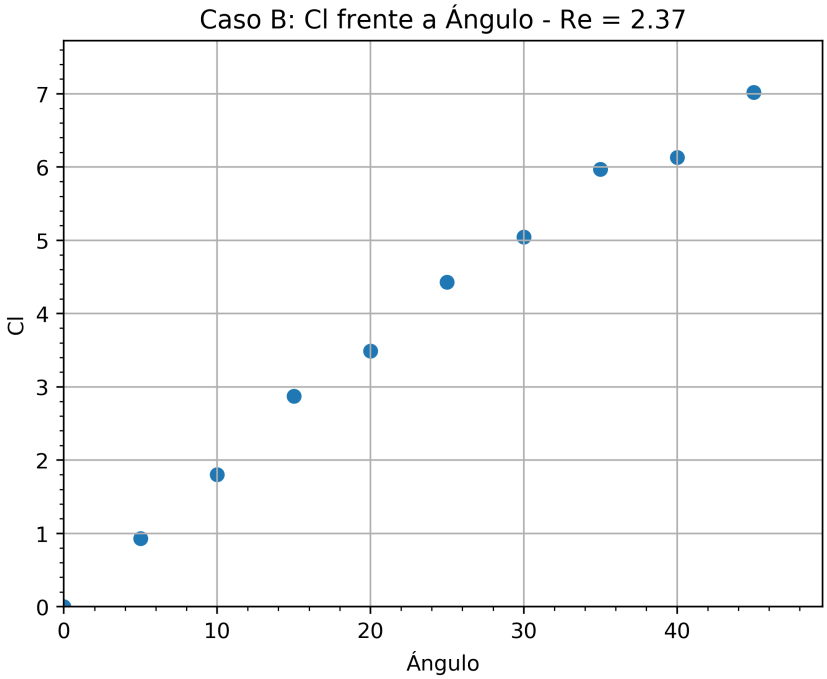
Las dimensiones de los perfiles que se han usado en ese caso de prueba están en el cuadro 3.7.

Los coeficientes de arrastre obtenidos están representados en la figura 3.9. Y los coeficiente de sustentación se representan en la figura 3.10.

Comparado con el perfil anterior, el presente perfil presenta unos coeficientes de sustentación menores para el mismo ángulo de ataque. Para ángulos bajos se aprecia el mismo comportamiento para el coeficiente de sustentación, hasta llegar a 35 grados, donde aparece una salida de la tendencia, con una bajada brusca en el valor del coeficiente. Esta anomalía puede tener origen en el cambio que sufren los nodos de la retícula al rotar el perfil, causando una perdida

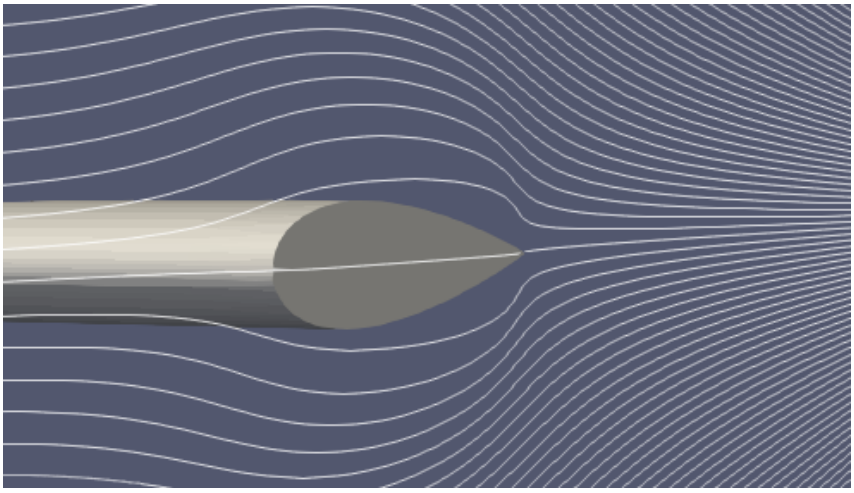


(a) Caso A

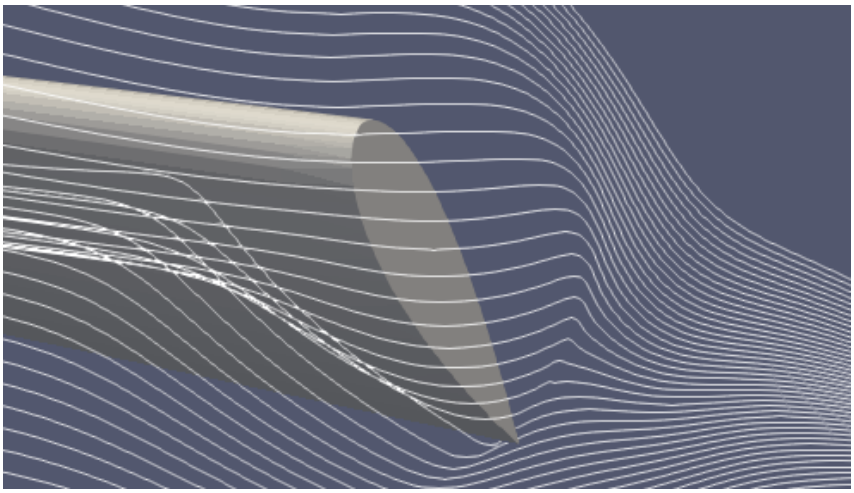


(b) Caso B

Figura 3.7: NACA 0021: Coeficiente de Sustentación frente a ángulo de ataque.

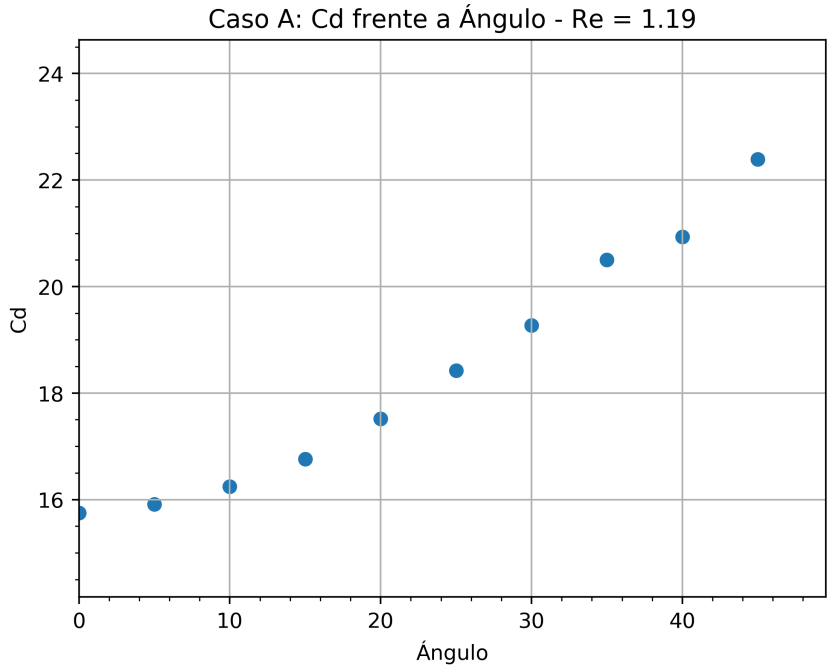


(a) 0 Grados

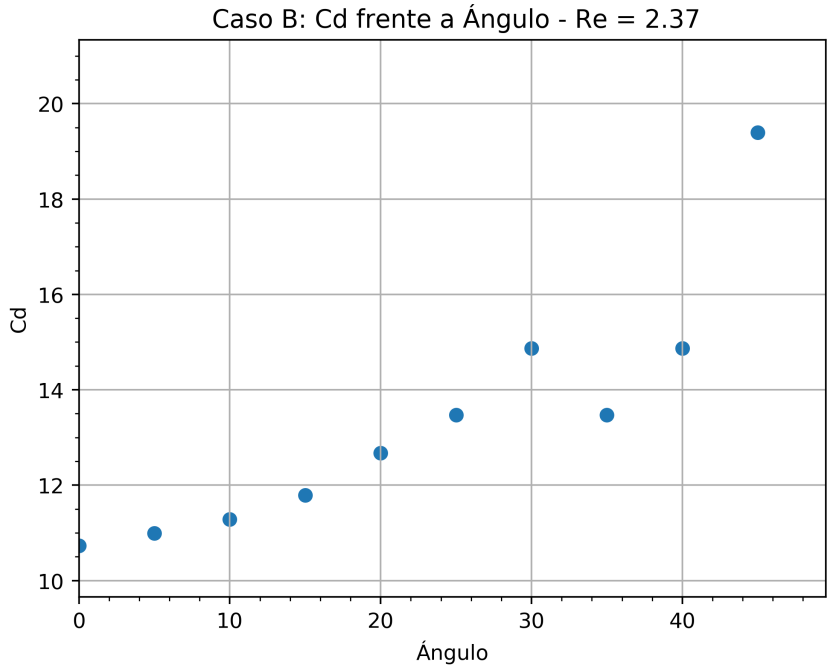


(b) 45 Grados

Figura 3.8: NACA 0021: Flujo en el extremo del perfil.

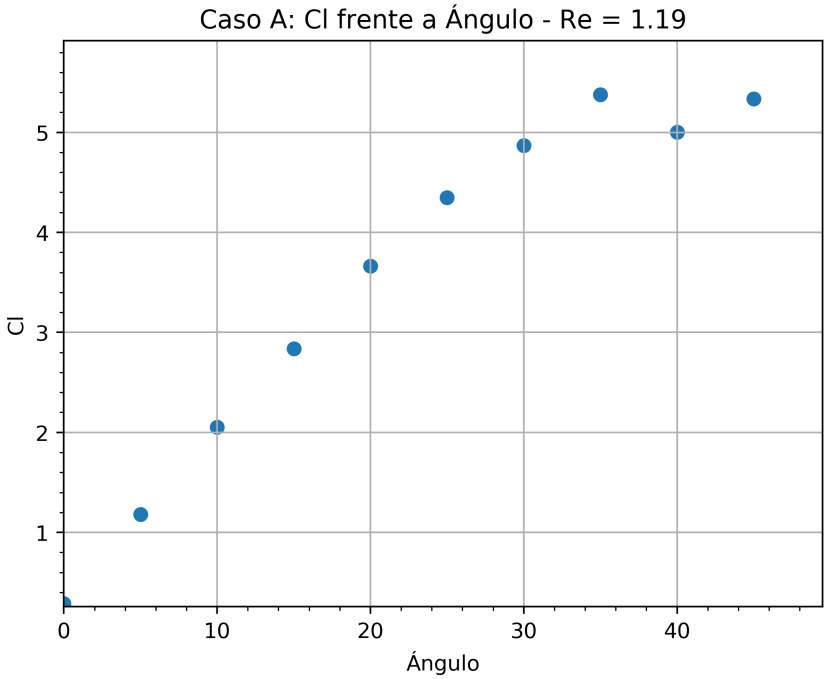


(a) Caso A

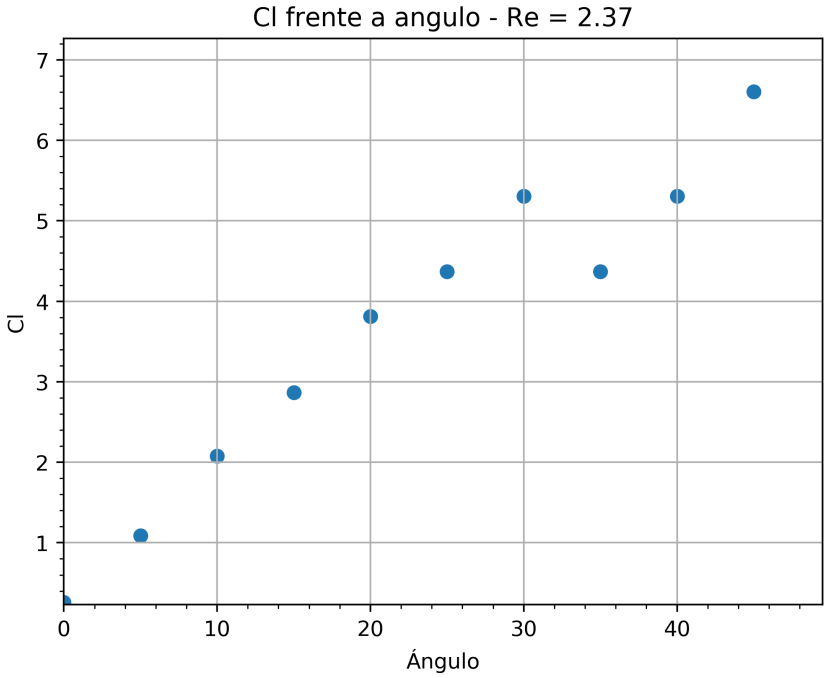


(b) Caso B

Figura 3.9: NACA 6713: Coeficiente de Arrastre frente a ángulo de ataque.



(a) Caso A



(b) Caso B

Figura 3.10: NACA 6713: Coeficiente de Sustentación frente a ángulo de ataque.

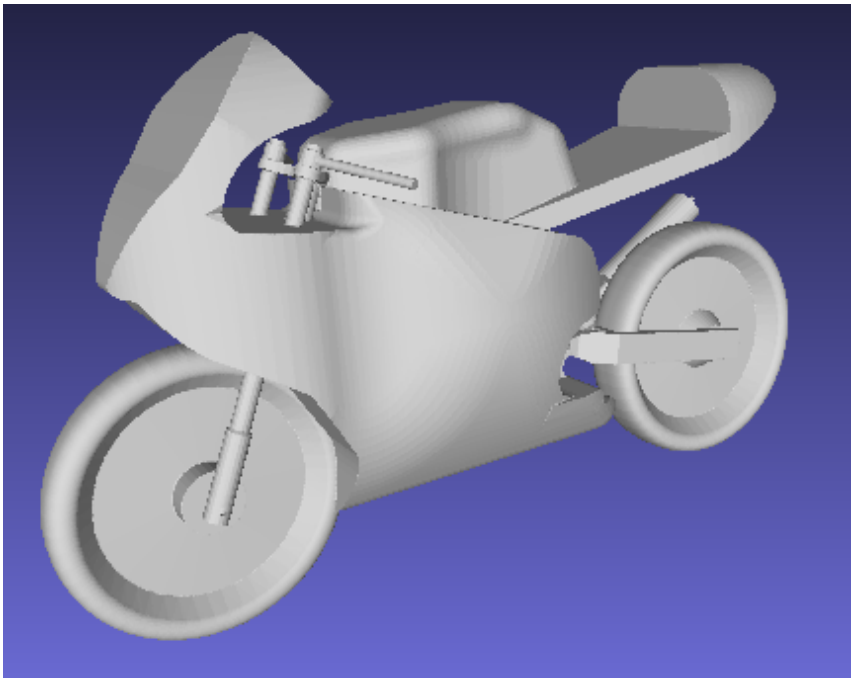


Figura 3.11: Estudio aerodinámico de motocicleta: Modelo STL de Motocicleta.

de precisión. Para obtener mayor precisión en dichos puntos sería preciso realizar un mallado más fino.

3.3 Estudio aerodinámico de una motocicleta

Con este caso se pretende mostrar la viabilidad del túnel de viento virtual creado en este trabajo para estudiar la aerodinámica externa de cuerpos reales, y con una cierta complejidad geométrica.

La figura 3.11 representa un modelo en formato STL de una motocicleta[2] sobre la cual se va a estudiar la evolución del flujo, conforme aumenta el número de Reynolds. Para el cálculo del número de Reynolds se utiliza la ecuación:

$$Re = \frac{lu}{\nu} \tag{3.7}$$

donde

- l es la altura de la moto.

[2]El fichero STL fue suministrado por el Grupo de Fluidodinámica Numérica de la Universidad de Zaragoza.

- u es la velocidad del fluido no perturbado.
- ν es la viscosidad cinemática.

Se ha usado un valor de $\delta_x = 0{,}0057$, velocidad de entrada de 0,25, $\delta_t = 0{,}0001$, densidad igual a 1.0 y anchura de la *Sponge Zone* de 0.

Se ha usado una dinámica *Regularized BGK* en las primeras simulaciones, hasta llegar a la simulación con un valor $Re = 100$, donde al observar que en el flujo empezaban a aparecer inestabiliades se cambió a una dinámica de *Smagorinsky*, con un valor para la constante de Smagorinsky de 0,14.

Al tratarse de una motocicleta, hay que tener en cuenta la contribución del suelo a la aerodinámica. Se consigue simular el comportamiento del suelo en el lateral del túnel que esta en contacto con las ruedas.

Para representar el flujo alrededor del cuerpo se van a utilizar líneas de corriente, generadas por el software de visualización ParaView. Las líneas de corriente son tangentes en cada punto al vector velocidad del fluido en el punto.

Se parte de un número de Reynolds bajo, $Re = 1$, donde todo el flujo es laminar. Se han tomado dos vistas como referencia: una vista lateral en la figura 3.12 (a) y una vista detalle del manillar en la figura 3.12 (b).

Para un número de Reynolds más alto, $Re = 10$, se ve el perfil con la variación de la presión con respecto a su valor inicial, figura 3.13 (b). En ella se identifica las zonas de altas presiones en la parte frontal de la motocicleta.

Si se aumenta el número de Reynolds, $Re = 100$, aparecen las primeras zonas de recirculación, en el manillar 3.14 (a), asiento 3.14 (b) y parte posterior 3.14 (c).

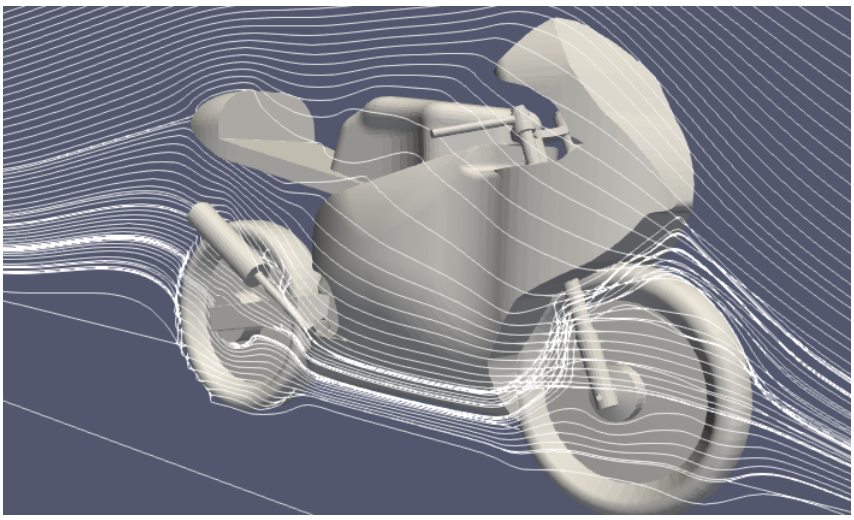
Por último, se puede ver la variación del coeficiente de arrastre y de sustentación en el cuadro 3.8. Las fórmulas usadas para el cálculo de estos coeficientes son 3.8 para el coeficiente de arrastre y 3.9 para el coeficiente de sustentación.

$$C_D = \frac{F_D}{\frac{1}{2}\rho u^2 S} \tag{3.8}$$

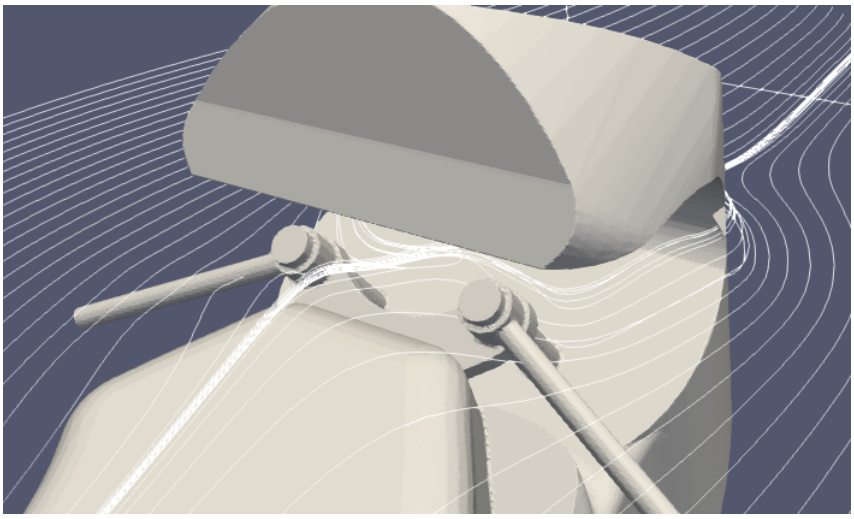
$$C_L = \frac{F_L}{\frac{1}{2}\rho u^2 S} \tag{3.9}$$

donde

- S es la superficie frontal de la motocicleta.

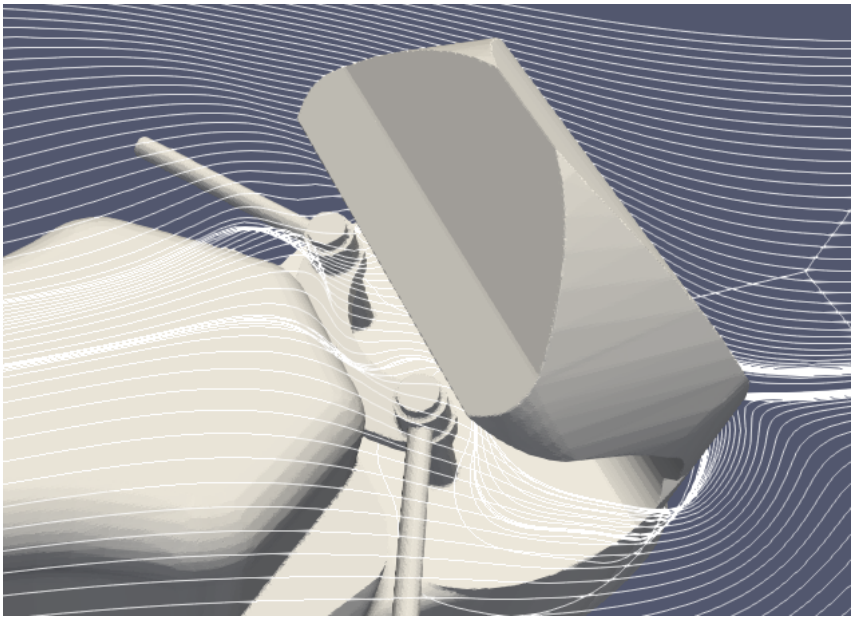


(a) Vista lateral

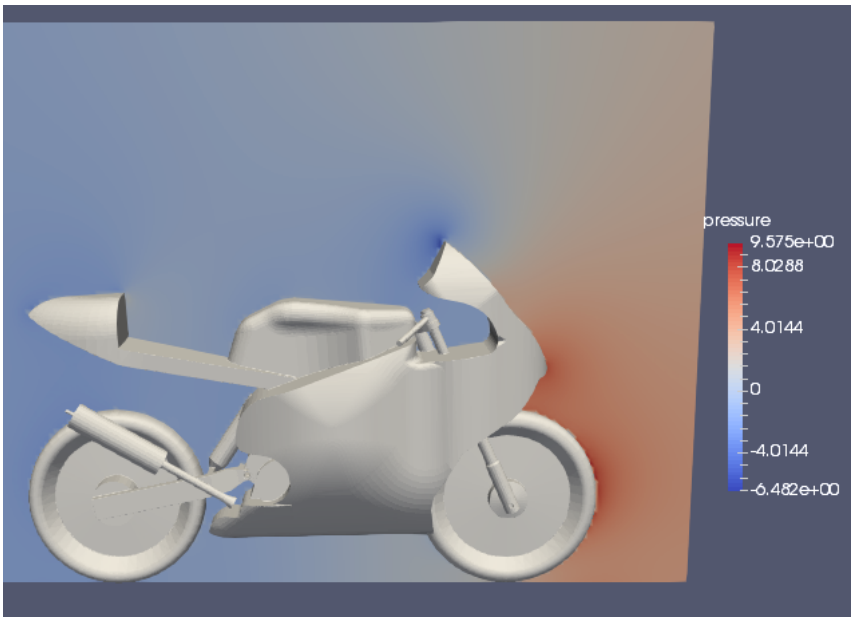


(b) Detalle del manillar

Figura 3.12: Estudio aerodinámico de una motocicleta: $Re = 1$.

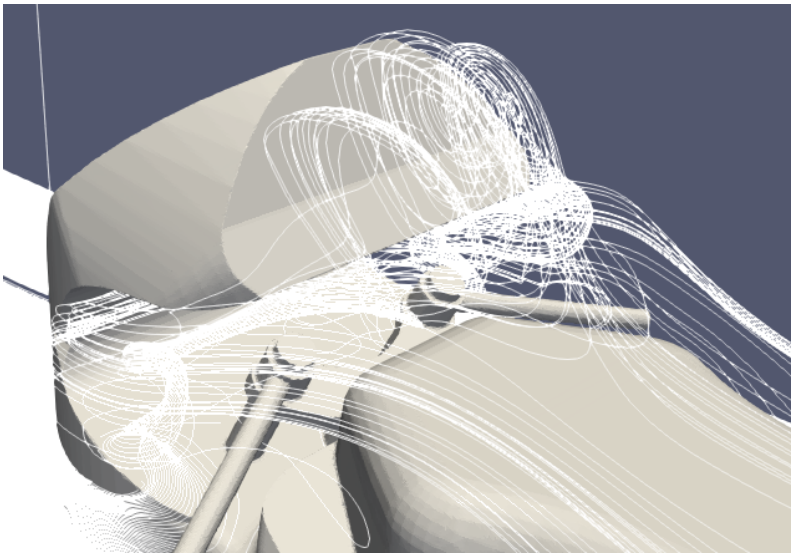


(a) Detalle del manillar

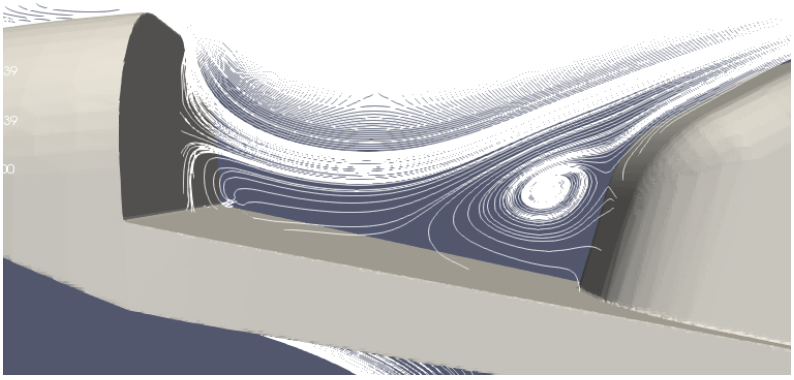


(b) Presiones en un corte transversal de la motocicleta

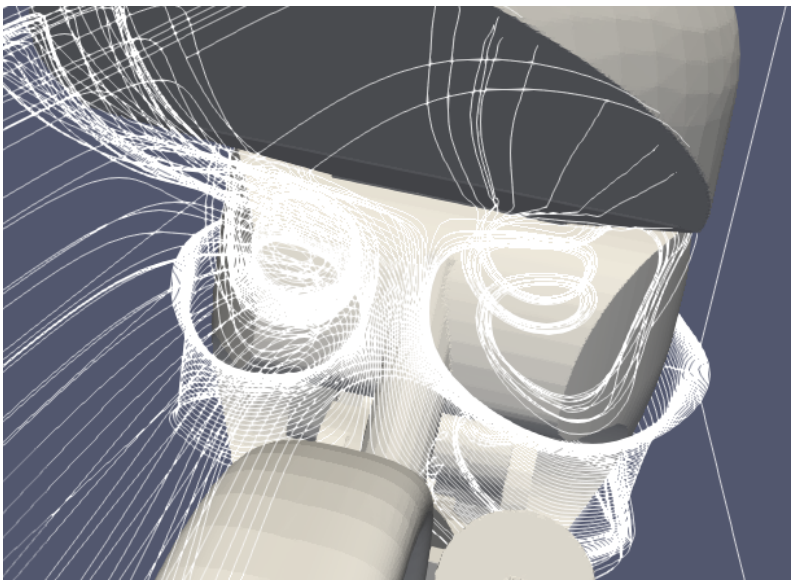
Figura 3.13: Estudio aerodinámico de motocicleta: $Re = 10$.



(a) Detalle del manillar



(b) Asiento



(c) Parte posterior

Figura 3.14: Estudio aerodinámico de motocicleta: $Re = 100$.

Número de Reynolds	Coeficiente de arrastre	Coeficiente de sustentación
1	120,19	14,45
10	16,62	6,71
100	16,64	1,57

Cuadro 3.8: Estudio aerodinámico de motocicleta: Coeficiente de arrastre y de sustentación frente a número de Reynolds.

- u es la velocidad del fluido no perturbado.
- ρ es la densidad del fluido.
- F_D es la fuerza de arrastre.
- F_L es la fuerza de sustentación.

3.4 Estudio aerodinámico de edificio

En la figura 3.15 se representa el modelo en formato STL del edificio[3] sobre el que se va a estudiar la evolución del flujo para distintas orientaciones.

Para caracterizar cada simulación se usa el número de Reynolds, definido por la ecuación:

$$Re = \frac{lu}{\nu} \tag{3.10}$$

donde

- l es la altura del edificio.
- u es la velocidad del fluido no perturbado.
- ν es la viscosidad cinemática.

Para las dos orientaciones se ha usado un valor de $Re = 30$.

[3]El fichero STL del edificio fue suministrado por Raúl Losantos Viñuales.

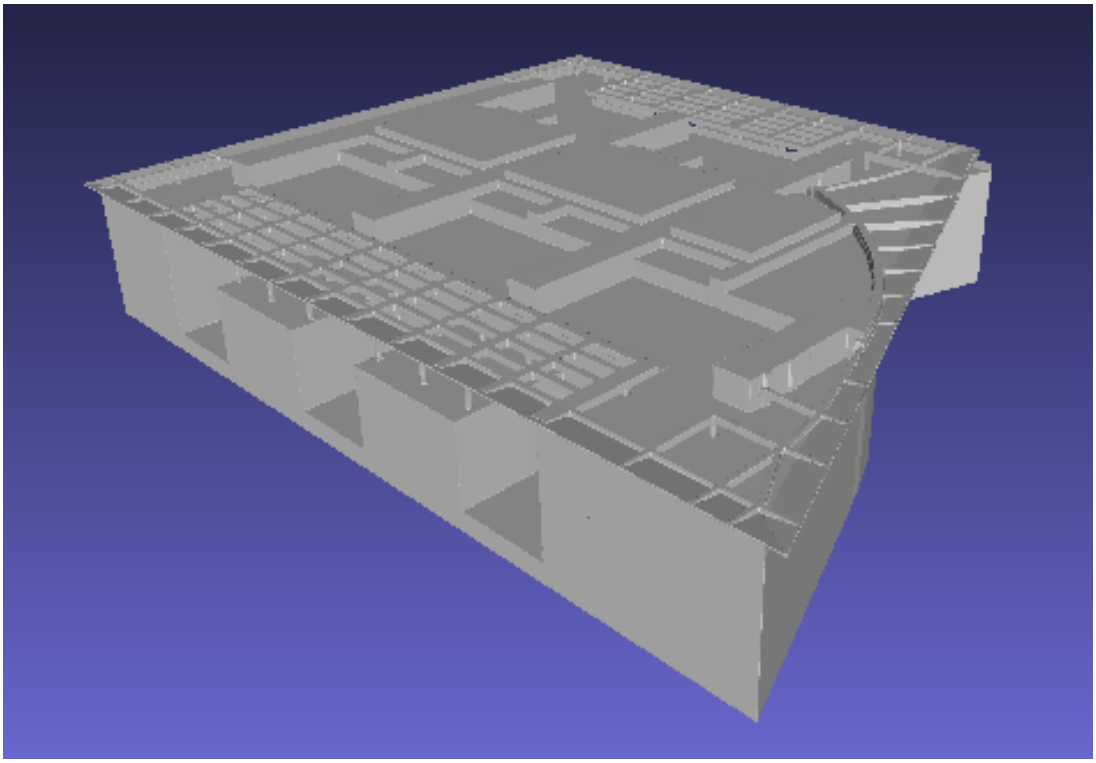


Figura 3.15: Estudio aerodinámico de edificio: Modelo STL de edificio.

Se han elegido dos orientaciones, en la primera el flujo impacta directamente sobre una pared del edificio (Caso A), mientras que, en la segunda, el flujo impacta en una de sus esquinas (Caso B).

Se ha usado un valor de $\delta_x = 0{,}041$, velocidad de entrada de 0,25, $\delta_t = 0{,}002$, densidad igual a 1,0, anchura de la *Sponge Zone* de cero y se ha usado una dinámica *Regularized BGK*.

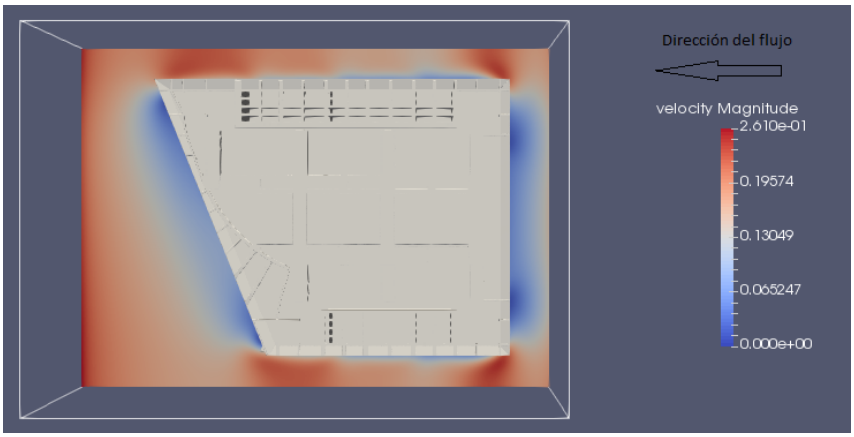
Al tratarse de un edificio, la contribución del suelo es importante en su estudio. Se simula el comportamiento del suelo en el lateral del túnel que esta en contacto con el edificio.

Se puede ver la velocidad del flujo en las zonas colindantes en la figura 3.16.

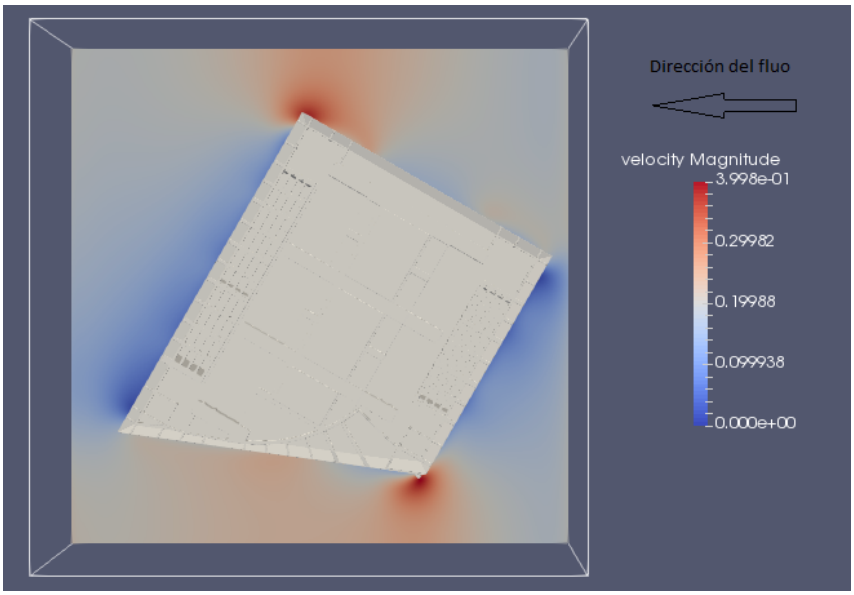
En ellas se ven las zonas dónde el flujo tiene mayor velocidad, en color rojo. Estas no serían recomendables, por ejemplo, para la instalación de terrazas o parques.

Para visualizar el movimiento del flujo, se crean las líneas de corriente del flujo.

En el caso A se aprecia la formación de remolinos, zonas de recirculación, en la parte lateral, figura 3.17 (a) y (b), y el la parte posterior, figura 3.17 (c). Mientras que en el caso B el flujo pasa sin ser perturbado.

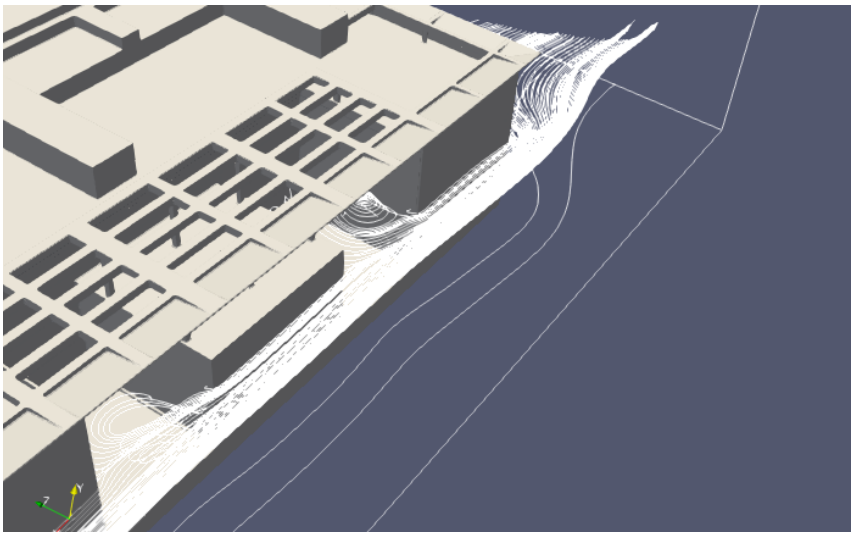


(a) Caso A

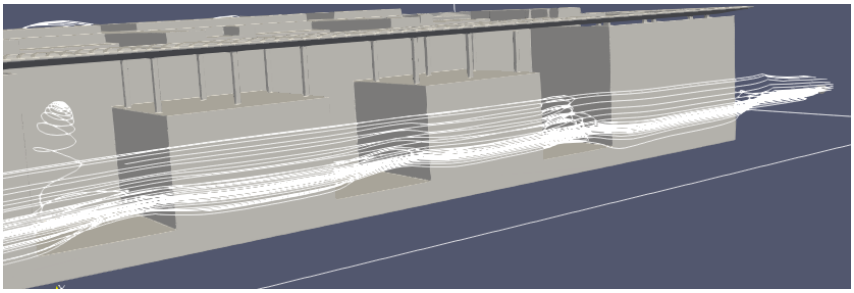


(b) Caso B

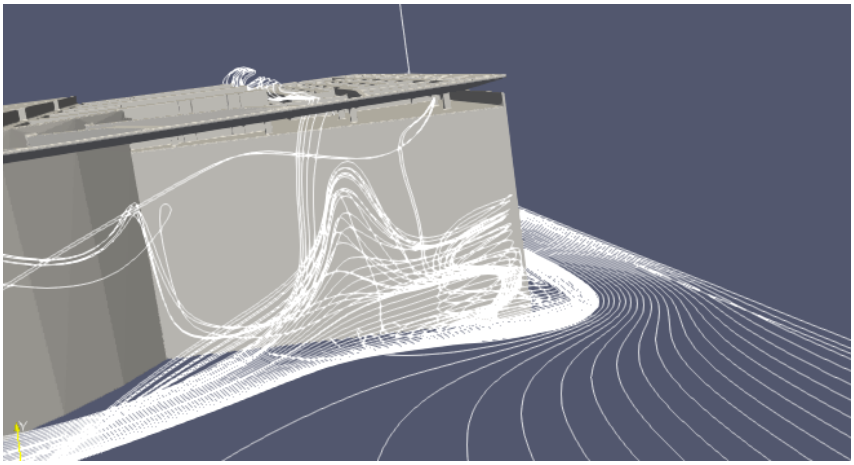
Figura 3.16: Estudio aerodinámico de edificio: Campo de velocidades.



(a) Vista 1



(b) Vista 2



(c) Vista 3

Figura 3.17: Estudio aerodinámico de edificio: Lineas de corriente orientación A.

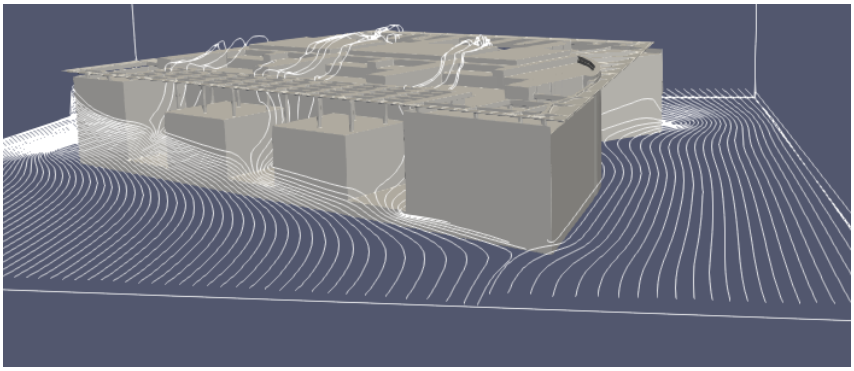


Figura 3.18: Estudio aerodinámico de edificio: Lineas de corriente orientación B.

3.5 Tiempos de cálculo

Dos factores que influyen en el tiempo que dura una simulación son:

- Número de nodos del sistema. Es función de las dimensiones del túnel y la distancia entre nodos, δ_x.
- Número de iteraciones para completar la simulación. Es función del incremento de tiempo entre iteraciones, δ_t, y duración de la simulación.

Para obtener unos valores orientativos, se ha realizado una simulación con $3 \cdot 10^6$ nodos en la que se realizan 10^4 iteraciones, la cual se ha realizado en una sola unidad de procesamiento, un núcleo de un procesador.

La simulación finaliza tras $5,92 \cdot 10^3$ segundos. Con un escalamiento lineal se puede ver cómo la simulación de la motocicleta, que constaba de 10^8 nodos y se ejecutaban $6 \cdot 10^4$ iteraciones, requeriría de un tiempo aproximado de $1,18 \cdot 10^6$ segundos, 13 días. Gracias a la paralelización del código fue posible reducir dicha cifra y realizarla en un tiempo menor. Sin embargo, si se desease realizar una simulación con mayor números de Reynolds, el escalamiento del problema implicaría una mayor cantidad de nodos en el sistema, aumentando el coste computacional hasta límites que sólo podría ser tratados mediante simulaciones masivamente paralelas en entornos adecuados, de los cuales no se ha dispuesto en el presente proyecto.

CAPÍTULO 4

Conclusiones y lineas de trabajo futuro

4.1 Conclusiones

Se ha conseguido una herramienta de cálculo aerodinámico que permite la obtención de resultados a partir de un modelo CAD, sobre el cual se realizan unas reparaciones mínimas, en caso de necesitare. De esta manera se elimina la mayor dificultad que aparece en los métodos basados en malla, la preparación de dicha malla.

En lo referente al software realizado, se ha comprobado la facilidad con la que es posible desarrollar una herramienta usando distintos lenguajes de programación, utilizando cada uno de ellos para la tarea en la que resulte más apto, por ejemplo un lenguaje compilado para la tarea que requería mayor coste computacional, realización de la simulación, y otro cómo Python para el desarrollo de la interfaz gráfica o el sistema de avisos. Al realizar esta separación entre diferentes programas especializados en una tarea determinada, se ha seguido la denominada *filosofía UNIX*. Esto proporciona ventajas con vistas al futuro, por ejemplo permite seguir utilizar el núcleo del programa como motor de cálculo y cambiar la interfaz a un diseño web.

4.2 Líneas de trabajo futuro

Como líneas sobre las que seguir trabajando para ampliar dicho proyecto se proponen:

- Nuevas simulaciones con números de Reynolds superiores, entrando así en régimen turbulento, y estudiar el funcionamiento de la herramienta en estas condiciones.
- Mejoras en la herramienta. Una mejora que en su momento se planteo pero fue desechada por falta de tiempo fue el refinamiento local de la malla en la zona de interés. Gracias a

esto sería posible reducir en gran medida el tiempo de simulación.

- Adaptación a nuevos entornos de trabajo. Con la aparición de nuevos dispositivos, cómo los *smartphone* o *tablet*, sería de interés la adaptación de la herramienta para pudiera correr en ellos.

- Nuevo sistema de uso. Una de las dificultades que se experimentó a la hora de utilizar la herramienta fue el sistema de conexión con el cluster, SSH. Dado el amplio desarrollo de otras tecnologías, como la tecnología web, sería de interés la creación de una interfaz a través de un portal web, que permitiese su uso a través de un navegador.

- Estudio de la viabilidad como modelo de negocio. Dado el coste nulo de la herramienta, su facilidad de uso y los resultados que proporciona, podría usarse como motor de calculo en una empresa de análisis aerodinámico.

Bibliografía

C. Sukop, M. y T.Thorne, D. (2006), *Lattice Boltzmann Modeling An Introduction for Geoscientists and Engineers*.

Dosselaer, I. V. (2014), Buoyant Aerobot Design and Simulation Study, PhD thesis, Delft University of Technology.

GrabCAD (2017).
URL: https://grabcad.com/

Krüger, T., Varnik, F. y Raabe, D. (2009), 'Shear stress in lattice Boltzmann simulations', *Physical Review E* **79**(4). arXiv: 0812.3242.
URL: http://arxiv.org/abs/0812.3242

Latt, J. (2008), 'Choice of units in lattice Boltzmann simulations'.
URL: http://wiki.palabos.org/_media/howtos:lbunits.pdf

Lätt, J. (2014), Hydrodynamic limit of lattice Boltzmann equations, PhD thesis, Université de Genève.

Lutz, M. (2010), *Programming Python*, 4(th) edn, O'REILLY.

McKinney, W. (2012), *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*, 1(st) edn, O'REILLY.

Moufekkir, F., Moussaoui, M., Mezrhab, A., Naji, H. y Lemonnier, D. (2012), 'Numerical prediction of heat transfer by natural convection and radiation in an enclosure filled with an isotropic scattering medium', *Journal of Quantitative Spectroscopy and Radiative Transfer* **113**, 1689–1704.

Márquez, F. M. (2004), *UNIX Programación avanzada*, 3(rd) edn, Ra-Ma.

Non-Manifold edge and vertices (2017).
URL: http://pointclouds.org/blog/_images/non_manifold.png

Palabos (n.d.).
URL: http://www.palabos.org/

Palabos LBM Wiki (2017).
URL: http://wiki.palabos.org/

Stroustrup, B. (1985), *El lenguaje de programación C++*, 1(st) edn, ADDISON WESLEY.

Touretzky, D. (2017), 'STL Files and Slicing Software'.
URL: https://www.cs.cmu.edu/afs/cs/academic/class/15294-s15/lectures/stl/stl.pdf

Viggen, E. M. (2014), The lattice Boltzmann method: Fundamentals and acoustics, PhD thesis, Norwegian University of Science and Technology.

APÉNDICE A

Introducción al método de Lattice-Boltzmann

El método de Lattice-Boltzmann es un método de resolución numérico compresible para flujo incompresible, usado en Mecánica de Fluidos Computacional (CFD). Ofrece muy buenos resultados para casos tales como interacción con superficies, flujos multifásicos, evaporación, condensación, cavitación, turbulencia y muchos otros.

El método tiene su origen en los trabajos de Ludwig Boltzmann sobre el comportamiento de los gases. La idea principal era que, estos, estaban formados por un conjunto de partículas, que interactuaban entre ellas de acuerdo a las leyes de la mecánica clásica. Como un gas estaba formado por un gran número de estas partículas, el comportamiento macroscópico de éste tenía que ser descrito de una manera estadística (Teoría cinética de los gases). A partir de este trabajo, se desarrolló el denominado *Lattice gas automaton*, un autómata celular que sirvió como base para los actuales modelos de Lattice-Boltzmann.

A.1 El Modelo de Lattice-Boltzmann

Es posible demostrar que, de la ecuación de Boltzmann, A.1, que modela el cambio de la distribución de probabilidad de un gas, se pueden derivar las ecuaciones de Navier-Stokes para el caso de que la densidad sufra pequeños cambios.

$$f_i(\vec{x}+\vec{e}_i\delta_t, t+\delta_t) = f_i(\vec{x},t) + \Omega_i \tag{A.1}$$

El término Ω_i se le denomina termino de colisión, y existen diversas ecuaciones para su cálculo. Posteriormente se verá una de las posibles implementaciones, llamada Bhatnagar–Gross–Krook

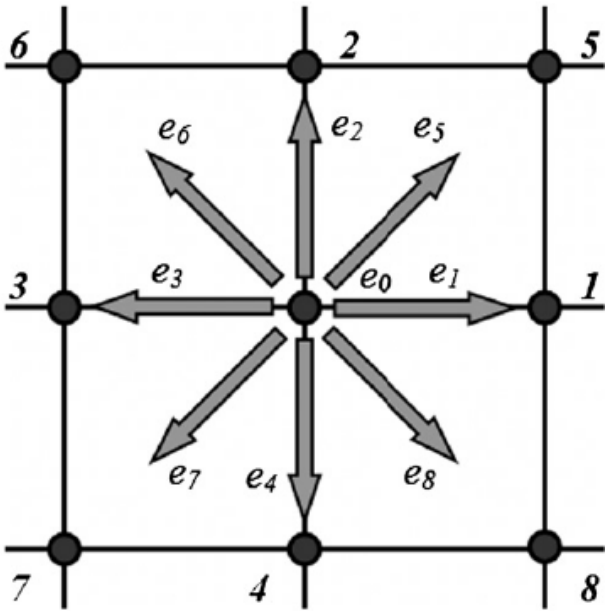


Figura A.1: Malla tipo D2Q9 con microvelocidades.

(BGK), la cual, a pesar de ser la más sencilla posible, proporciona unos buenos resultados.

En esencia, el método de Lattice-Boltzmann consiste en discretizar el dominio, el momento de las partículas y el tiempo. De esta manera el fluido queda modelizado como un conjunto de partículas virtuales. La función que marca la dinámica de dichas partículas es la discretización de la ecuación de Boltmann.

Con la discretización del dominio se crea una malla de puntos entre los cuales se pueden mover las partículas. Existen diversas mallas, que se pueden nombrar atendiendo a la nomenclatura DnQm, donde n es el número de dimensiones espaciales,2 o 3 generalmente, y m es el número de velocidades, número de direcciones en las que se pueden mover las partículas incluyendo la opción del propio nodo.

Con esta discretización del dominio se introduce un nuevo aspecto del método, la conversión a unidades lattice. La unidad básica de longitud $\delta_x = 1(lu)$ se hace coincidir con la distancia entre los nodos más cercanos.

Posteriormente hay que adimensionalizar la dimensión temporal, para ello se toma la unidad de tiempo discreto $\delta_t = 1(ts)$. Para facilitar los cálculos, se hace coincidir con el tiempo que transcurre entre cada iteración del método.

Una vez discretizado el dominio, en cada punto se establecen una serie de vectores velocidad:

$$\vec{e}_a \ , a = 0,1,2...,m-1 \tag{A.2}$$

A cada uno de los vectores se les denomina microvelocidades del nodo. Para el caso D2Q9, figura A.1[1], tienen el valor mostrado en A.3, A.4 y A.5.

$$\vec{e}_0 = 0 \ lu \ ts^{-1} \tag{A.3}$$

$$\vec{e}_1 = \vec{e}_2 = \vec{e}_3 = \vec{e}_4 = 1 \ lu \ ts^{-1} \tag{A.4}$$

$$\vec{e}_5 = \vec{e}_6 = \vec{e}_7 = \vec{e}_8 = \sqrt{2} \ lu \ ts^{-1} \tag{A.5}$$

Llegados a este punto, se tiene que incorporar la función de distribución f, con la cual se pueden calcular las variables macroscópicas de densidad, A.6, y velocidad del fluido, A.7. La ecuación de Boltzmann proporciona para cada dirección un valor, que puede interpretarse como el "flujo" de las partículas virtuales en esa dirección.

$$\rho = \sum_{a=0}^{m-1} f_a \tag{A.6}$$

$$\vec{u} = \frac{1}{\rho} \sum_{a=0}^{m-1} f_a \vec{e}_a \tag{A.7}$$

Pero para calcular el valor de f, primero se tiene que establecer un valor para el término de colisión.

A.2 El Operador de Bhatnagar–Gross–Krook

Para el operador de colisión nombrado anteriormente, Ω_i, tiene en su forma general una forma integral compleja. Sin embargo es posible encontrar una aproximación lineal, como la que nos ofrece el operador BGK, desarrollado en la ecuación:

$$\Omega_i = \frac{1}{\tau}(f_i^{eq} - f_i) \tag{A.8}$$

[1](Moufekkir et al. 2012)

Donde se ve que, el término de colisión, es proporcional a la diferencia entre el valor la función de distribución y un valor de equilibrio.

Discretizando la ecuación de Boltzmann, y aplicando el operador de Bhatnagar–Gross–Krook al termino de colisión, se obtiene la ecuación:

$$f_i(\vec{x}+\vec{e}_i\delta_t, t+\delta_t) = f_i(\vec{x},t) + \frac{1}{\tau}(f_i^{eq} - f_i) \tag{A.9}$$

Esta ecuación sintetiza dos fenómenos distintos que pueden ser separados en el término de colisión A.10 y el término de flujo:

$$\frac{1}{\tau}(f_i^{eq} - f_i) \tag{A.10}$$

$$f_i(\vec{x}+\vec{e}_i\delta_t, t+\delta_t) = f_i(\vec{x},t) \tag{A.11}$$

En el caso de que se esté tratando una condición de contorno de un sólido, estos dos términos tienen que ser separados para poder ser resueltos.

El valor de τ está ligado con la viscosidad del fluido, ya que, este término de la ecuación, es equivalente al termino viscoso de la ecuación de Navier-Stokes, al tipo de mallado, y al modelo de ecuación de estado que se tome para el fluido, ecuación de gas ideal, ecuación de Van der Waals... Para el caso más básico, un mallado D2Q9 se puede calcular como:

$$\nu_{lb} = c_s^2(\tau - \frac{1}{2}) \tag{A.12}$$

donde

- c_s^2 es la velocidad del sonido en el fluido al cuadrado. Es posible demostrar que medido en unidades lattice, tiene que tener un valor de $\frac{1}{3}$ para mantener la isotropía del fluido.
- ν_{lb} es la viscosidad cinemática del fluido en unidades lattice, posteriormente se explicará cómo realizar este cambio de unidades.

En esta situación se puede calcular el termino de equilibrio como:

$$f_a^{eq}(\vec{x}) = \omega_a \rho(\vec{x}) \left[1 + 3\frac{\vec{e}_a \cdot \vec{u}}{c_s^2} + \frac{9}{2}\frac{(\vec{e}_a \cdot \vec{u})^2}{c_s^4} - \frac{3}{2}\frac{\vec{u}\cdot\vec{u}}{c_s^2}\right] \tag{A.13}$$

donde

- $\omega_a = \begin{cases} \frac{4}{9} & \text{si a = 0} \\ \frac{1}{9} & \text{si a = 1,2,3,4} \\ \frac{1}{36} & \text{si a = 5,6,7,8} \end{cases}$
- c_s es la velocidad del sonido en el fluido
- $\vec{u}$ es la velocidad en el nodo $\vec{x}$

Dicho modelo recibe el nombre de Single-relaxation-time BGK, es la implementación más sencilla de los distintos métodos Lattice-Boltzmann. A pesar de eso es uno de los más utilizados, ya que presenta una gran versatilidad y buenos resultados.

Le velocidad del sonido en el fluido, c_s, es un parámetro muy importante en la simulación. Como se ha comentado, por motivos de isotropía del fluido, tiene un valor fijo en la simulación de $c_s = \frac{1}{\sqrt{3}}$.
Una condición necesaria que se tiene que cumplir en las simulaciones es A.14, ya que el modelo Lattice-Boltzmann no soporta flujos supersónicos.

$$|\vec{u}_{lb}| < c_s \tag{A.14}$$

A.3 Modelos más complejos

Debido a la simpleza del modelo *Single Relaxation Time BGK*, aparecen limitaciones a la hora de simular flujos con números de Reynolds elevados. El valor relativo de la baja viscosidad, que hace que el número de Reynolds sea elevado, causa problemas de inestabilidad, y hace que la resolución de las ecuaciones constitutivas produzca resultados erróneos o incluso no sea convergente. Con el fin de solucionar esto, se han desarrollado nuevos modelos que proporcionan mejores resultados para dichos casos.

El denominado *Entropic model* desarrollado por la Escuela Politécnica Federal de Zúrich, se basa en restaurar las consecuencias de la segunda ley de la termodinámica mediante la construcción de una función de entropía, que se asegura en cada iteración de que la entropía del sistema no decrece. Esta solución proporciona un modelo más estable para altos números de Reynolds, pero a cambio de aumentar en gran medida el coste computacional de la simulación.

Para mantener la mejora en la precisión y estabilidad que se ha obtenido con el Entropic mode, pero reduciendo el coste computacional, se puede utilizar el denominado *Regularized BGK*. En

él se definen una serie de nuevas funciones, como llamada función de no equilibrio, establecida por:

$$f_\alpha^{neq}(\vec{x}) = f_\alpha(\vec{x}) - f_\alpha^{eq}(\vec{x}) \tag{A.15}$$

El tensor de esfuerzos por:

$$\Pi_{\alpha\beta} = \sum_k e_\alpha e_\beta f_k \tag{A.16}$$

Para el caso del equilibrio:

$$\Pi_{\alpha\beta}^{eq} = \sum_k e_\alpha e_\beta f_k^{eq} \tag{A.17}$$

De forma análoga a como se definió la función de no equilibrio, se puede definir un tensor de no equilibrio:

$$\Pi_{\alpha\beta}^{neq} = \Pi_{\alpha\beta} - \Pi_{\alpha\beta}^{eq} \tag{A.18}$$

Por último se introduce un nuevo tensor:

$$Q_{\alpha ij} = e_{\alpha i} e_{\alpha j} - c\delta_{ij} \tag{A.19}$$

Donde c ya se ha definido con anterioridad y δ_{ij} es la Delta de Kronecker.

La clave del modelo es realizar la aproximación:

$$f_\alpha^{neq}(\vec{x}) \approx -\frac{\omega_\alpha}{2\tau c^4} Q_{\alpha ij} \Pi_{ij}^{neq} \tag{A.20}$$

Existen otros modelos más complejos, como por ejemplo el *Carreau model* para fluidos no-newtonianos. También existen modelos que permiten introducir el parámetro temperatura y modelar, así, flujos térmicos con cambios de temperatura, o incluso modelos que permiten incorporar cambios de fase o varios fluidos en varias fases. Si el flujo es turbulento, existe el llamado *Static Smagorinsky model* que permite acelerar las simulaciones.

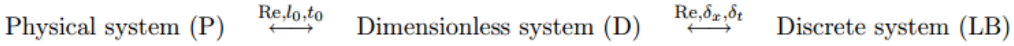


Figura A.2: Sistema Físico - Sistema Adimensional - Sistema Discreto.

A.4 Elección de unidades en simulaciones Lattice-Boltzmann

El flujo obtenido al circular un fluido sobre un cuerpo depende del número de Reynolds que se tenga. Este depende de los valores de viscosidad, velocidad del flujo y dimensiones del sólido. La estabilidad y velocidad de convergencia del método dependen del valor de estas dos variables. Para algunas simulaciones los valores que se tienen pueden no ser los adecuados, para ello una solución es crear un sistema adimensional intermedio, figura A.2[2].

Para identificar a qué sistema pertenece cada variable, se va a utilizar la siguiente nomenclatura de subíndices

- Subíndice p para los parámetros del sistema físico.
- Subíndice d para los parámetros del sistema adiomensional.
- Subíndice lb para los parámetros del discreto, o sistema Lattice-Boltzmann.

A pesar de que es posible pasar directamente del sistema físico (P) al sistema discreto (LB), la elección de este sistema adimensional es fuertemente recomendable, ya que adapta nuestro sistema a uno en el que ofrezca mejores resultados.

Algo muy importante, en términos de estabilidad y convergencia del método, es tener una densidad próxima a la unidad, por lo tanto, un cambio obligatorio será adimensionalizar ésta dividiendo por su valor ρ_0 para igualarla a uno $\rho_d = \frac{\rho_0}{\rho_0} = 1$.

A.4.1 Sistema físico a adiomensional

Para este paso se eligen dos variables características del problema

- Longitud de escala l_0
- Tiempo de escala t_0

[2](Latt 2008)

Cambiando así la escala de tiempo y las dimensiones del problema según las ecuaciones:

$$t_d = \frac{t_p}{t_0} \tag{A.21}$$

$$\vec{r_d} = \frac{\vec{r_p}}{l_0} \tag{A.22}$$

Este cambio implica que, las dimensiones del problema $\vec{r_p}$, tienen que ser cambiadas con una escala $\frac{1}{l_0}$.

De la misma manera se está obligado a hacer un cambio de unidades en el resto de las variables mediante las ecuaciones:

$$\vec{u_d} = \frac{t_0}{l_0}\vec{u_p} \tag{A.23}$$

$$\frac{\partial}{\partial t_d} = t_0 \frac{\partial}{\partial t_p} \tag{A.24}$$

$$\nabla_d = l_0 \nabla_p \tag{A.25}$$

$$p_d = \frac{1}{\rho_0}\frac{t_0^2}{l_0^2} p_p \tag{A.26}$$

Con estos parámetros, las ecuaciones de Navier-Stokes, A.27 y A.28, quedan adimensionalizadas en las ecuaciones:

$$\nabla_p \cdot \vec{u_p} = 0 \tag{A.27}$$

$$\frac{\partial \vec{u_p}}{\partial t_p} + (\vec{u_p} \cdot \nabla_p)\vec{u_p} = -\frac{1}{\rho_p}\nabla_p p_p + \nu_p \nabla_p^2 \vec{u_p} \tag{A.28}$$

$$\nabla_d \cdot \vec{u_d} = 0 \tag{A.29}$$

$$\frac{\partial \vec{u_d}}{\partial t_d} + (\vec{u_d} \cdot \nabla_d)\vec{u_d} = -\nabla_d p_d + \frac{1}{Re}\nabla_d^2 \vec{u_d} \tag{A.30}$$

Donde el parámetro *Re* sigue la ecuación A.31. Por lo que el valor de ν_d viene determinado por:

$$Re = \frac{l_0^2}{t_0 \nu_p} \tag{A.31}$$

$$\nu_d = \frac{1}{Re} \tag{A.32}$$

El factor *Re* es el número de Reynolds, el cual tiene el mismo valor medido en el sistema físico y en el sistema adimensional.

Con estos cambios se consigue que la presión y la densidad del fluido queden unificadas en una sola variable, a través de la ecuación de estado, ecuación de gas ideal. El único parámetro que queda para caracterizar el problema es la viscosidad adimensinal, ν_d.

Por lo tanto, dos flujos obtenidos por las ecuaciones de Navier-Stokes, con la ecuación de gas ideal como ecuación de estado, son idénticos si tienen la misma geometría (excepto un factor de escala) y el mismo número de Reynolds.

También se puede pasar las fuerza entre los dos sistemas mediante:

$$\vec{F}_d = \frac{t_0^2}{l_0^4 \rho_0} \vec{F}_p \tag{A.33}$$

A.4.2 Sistema adimensional a discreto

Para discretizar el sistema se tiene que elegir un parámetro espacial,δ_x, y otro temporal,δ_t. Donde δ_x es la distancia entre nodos contiguos, y δ_t el tiempo que transcurre entre dos iteraciones.

El resto de las magnitudes se obtienen mediante las ecuaciones:

$$\vec{u_{lb}} = \frac{\delta_t}{\delta_x} \vec{u_d} \tag{A.34}$$

$$\nu_{lb} = \frac{\delta_t}{\delta_x^2} \nu d \tag{A.35}$$

A.5 Modelo de turbulencia

Si se intentan resolver directamente las ecuaciones de Navier-Stokes para un flujo turbulento, utilizando un método numérico, lo que es conocido como Simulación Numérica Directa (DNS),

tiene como problema el enorme coste computacional que esto requiere.

Con el fin de simplificar este coste, se han desarrollado los denominados modelos de turbulencia.

Uno de los modelos es el denominado Modelo de Simulación de Grandes Remolinos, *Large Eddy Simulation* (LES), el cual se basa en aplicar un filtro a las ecuaciones que rigen el problema. Matemáticamente hablando, esto se consigue aplicando una operación de convolución que actúa como un filtro de paso bajo sobre la función objetivo.

Gracias a esto se consigue que se resuelva el problema para grandes escalas, ignorando la resolución de pequeñas turbulencias, cuya influencia sobre el resto del flujo sí se tiene en cuenta. Los modelos funcionales, o modelos de viscosidad de Eddy, cuantifican la energía disipada en las pequeñas escalas, gracias a la introducción de una viscosidad artificial, o viscosidad de Eddy. Esto se puede expresar con la ecuación:

$$\tau^r_{ij} - \frac{1}{3}\tau_{ij}\delta_{ij} = -2\nu_t \overline{S}_{ij} \tag{A.36}$$

donde:

- ν_t es la viscosidad de Eddy
- δ_{ij} es la delta de Kronecker
- τ_{ij} es el tensor de esfuerzos
- τ^r_{ij} es el tensor de esfuerzos residual anisótropo
- $\overline{S}_{ij} = \frac{1}{2}(\frac{\partial \vec{u}_i}{\partial x_j} + \frac{\partial \vec{u}_j}{\partial x_i})$

El tensor de esfuerzos residual anisótropo puede calcularse como:

$$\tau^r_{ij} = \tau^R_{ij} - \frac{1}{3}\tau^R_{kk}\delta_{ij} \tag{A.37}$$

Siendo τ^R_{ij} el tensor de esfuerzos residual que puede calcularse como:

$$\tau^R_{ij} = \overline{u_i u_j} - \overline{u_i}\,\overline{u_j} \tag{A.38}$$

No existe un método para calcular la viscosidad de Eddy, ya que este valor exacto depende del problema en concreto. Por lo que se han desarrollado diferentes modelos que intentan aproximar

su valor. El modelo de Smagorinsky–Lilly es el modelo más simple, y aproxima dicho valor mediante:

$$\nu_t = (\delta_g C_s)^2 \left|\overline{S}\right| \tag{A.39}$$

donde:

- δ_g El tamaño de la escala, a partir de la cual, para valores menores, se aplica el efecto del filtrado.
- C_s Constante de Smagorinsky. Su valor exacto depende de cada problema. En la práctica se le asocia un valor entre 0,1 y 0,2.
- $|S| = \sqrt{2\overline{S}_{ij} : \overline{S}_{ij}}$

APÉNDICE B

Código

En el presente apéndice se presentan todo el código utilizado en el presente proyecto.

B.1 Interfaz gráfica

Esta primera sección esta dedicada a la interfaz del programa. Una vez se ha terminado de utilizar y se presiona el botón para comenzar la simulación, este proceso termina y se pasa el control al código en C++ de la sección B.2.

```
'''
This file is part of the U-Virtual Wind Tunnel.

Copyright (C) 2017

The U-Virtual Wind Tunnel is free software: you can redistribute it and/or
modify it under the terms of the GNU Affero General Public License as
published by the Free Software Foundation, either version 3 of the
License, or (at your option) any later version.

The U-Virtual Wind Tunnel is distributed in the hope that it will be useful
    ,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU Affero General Public License for more details.

You should have received a copy of the GNU Affero General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
'''

#Se cargan los modulos necesarios
```

```
21 import sys
22 from lxml import etree
23 from io import StringIO
24 import os
25 #Se carga la GUI creada con Qt Creator
26 from interfazv5 import *
27 class ventana(QtWidgets.QDialog):     #Se crea el objeto Ventana, heredada
       del objeto QDialog
28   def __init__(self, parent=None):     #Construrctor
29     QtWidgets.QWidget.__init__(self, parent)     #Se llama al contructor de
      la superclase.
30     self.ui=Ui_Dialog()     #Atributo ui, contiene la GUI.
31     self.ui.setupUi(self)   #Se indica que el propio objeto contiene la GUI
       del programa.
32     #Leer el archivo XML
33     self.f = open("externalFlowAroundObstacle.xml")
34     self.xml = self.f.read()
35     self.f.close()
36     #Parseamos el fichero XML y creamos un objeto que tendra el nuevoXML
37     self.nuevoXML = ''   # Variable en blanco que se usara posteriormente.
38     self.tree = etree.parse(StringIO(self.xml))   # Se parsea el archivo de
       configuración, es un fichero XML. Se crea el objeto etree que contiene
      los datos del documento XML. Mediante la funcion StringIO ajustamos el
      tipo de formato que necesita el método parse como entrada, ya que el
      tipo de dato en que se encontraba el atributo xml no era el indicado.
       '''
39     '''
40     Se establecen los valores de la configuracion inicial. Para ello se
      indica cada uno de los componentes, self.ui.elemento. En caso de que sea
       un cuadro de texto con el método setText se indica el texto. Si es un
      elemento radio button, con setChecked(True) se queda marcado, con
      setChecked(False) se queda desmarcado. Para buscar el valor de la
      configuracion, se busca el nodo que almacena dicha configuración
      mediante XPath. Como se devuelve una tupla de elementos, se accede al ú
      nico elemento de dicha tupla mediante el operador [0]. Para saber la
      información que almacena dicho nodo, se utiliza el atributo text.
       '''
41     '''
42     #Geometria
43     self.ui.lineArchivo.setText(self.tree.xpath('/configuracion/geometry/
      filename')[0].text)   # Nombre del fichero STL.
44     self.ui.linePosicionX.setText(self.tree.xpath('/configuracion/geometry/
      center/x')[0].text)   # Coordenada X donde posicionar el obstaculo.
45     self.ui.linePosicionY.setText(self.tree.xpath('/configuracion/geometry/
      center/y')[0].text)   # Coordenada Y donde posicionar el obstaculo.
```

```
46      self.ui.linePosicionZ.setText(self.tree.xpath('/configuracion/geometry/
        center/z')[0].text)    # Coordenada Z donde posicionar el obstaculo.
47      self.ui.lineDominioX.setText(self.tree.xpath('/configuracion/geometry/
        domain/x')[0].text)    # Tamaño del túnel en el eje X.
48      self.ui.lineDominioY.setText(self.tree.xpath('/configuracion/geometry/
        domain/y')[0].text)    # Tamaño del túnel en el eje Y.
49      self.ui.lineDominioZ.setText(self.tree.xpath('/configuracion/geometry/
        domain/z')[0].text)    # Tamaño del túnel en el eje Z.
50      self.ui.linePhi.setText(self.tree.xpath('/configuracion/geometry/Giros/
        phi')[0].text)    # Giro inicial Phi.
51      self.ui.lineTheta.setText(self.tree.xpath('/configuracion/geometry/
        Giros/theta')[0].text)    # Giro inicial Theta.
52      self.ui.linePsi.setText(self.tree.xpath('/configuracion/geometry/Giros/
        psi')[0].text)    # Giro inicial Psi.
53      if self.tree.xpath('/configuracion/geometry/lateralFreeSlip')[0].text==
        'True':    # Condiciones de contorno.
54          self.ui.radioContornoLibre.setChecked(True)
55      else:
56          self.ui.radioContornoSuelo.setChecked(True)
57      if self.tree.xpath('/configuracion/geometry/freeSlipWall')[0].text=='
        True':    # Mediante una busqueda con XPath, se busca el nodo que indica
         el tipo de condiciones de contorno a utilizar en la superficie del só
        lido. Posteriormente, se marca el radio button adecuado.
58          self.ui.radioSolidoDeslizamiento.setChecked(True)
59      else:
60          self.ui.radioSolidoNoDeslizamiento.setChecked(True)
61      #Numericos
62      self.ui.lineDensidad.setText(self.tree.xpath('/configuracion/numerics/
        ro')[0].text)    # Densidad
63      self.ui.lineTiempoSimulacion.setText(self.tree.xpath('/configuracion/
        numerics/maxT')[0].text)    # Tiempo de simulación.
64      self.ui.lineVelocidadEntrada.setText(self.tree.xpath('/configuracion/
        numerics/inletVelocity')[0].text)    # Velocidad de entrada del fluido.
65      self.ui.lineViscosidadCinematica.setText(self.tree.xpath('/
        configuracion/numerics/nu')[0].text)    # Viscosidad cinemática del
        fluido.
66      self.ui.lineIncrementoPhi.setText(self.tree.xpath('/configuracion/
        numerics/deltaphi')[0].text)    # Incremento del angulo Phi en cada
        simulación.
67      self.ui.lineIncrementoTheta.setText(self.tree.xpath('/configuracion/
        numerics/deltatheta')[0].text)    # Incremento del angulo Theta en cada
        simulación.
```

```
68      self.ui.lineIncrementoPsi.setText(self.tree.xpath('/configuracion/
        numerics/deltapsi')[0].text)    # Incremento del angulo Psi en cada
        simulación.
69      self.ui.lineNumeroEjecuciones.setText(self.tree.xpath('/configuracion/
        numerics/iter')[0].text)    # Número de iteraciones.
70      self.ui.lineResolucion.setText(self.tree.xpath('/configuracion/numerics
        /resolution')[0].text)    # Resolución del eje Y.
71      self.ui.lineDt.setText(self.tree.xpath('/configuracion/numerics/dt')
        [0].text)    # Valor dt.
72      # Tipo de dinámica.
73      if self.tree.xpath('/configuracion/numerics/modelo')[0].text=='
        Smagorinsky':
74          self.ui.radioModeloSmagorinsky.setChecked(True)
75      elif self.tree.xpath('/configuracion/numerics/modelo')[0].text=='
        Entropic':
76          self.ui.radioModeloEntropic.setChecked(True)
77      elif self.tree.xpath('/configuracion/numerics/modelo')[0].text=='BGK':
78          self.ui.radioModeloBGK.setChecked(True)
79      elif self.tree.xpath('/configuracion/numerics/modelo')[0].text=='
        Regularized':
80          self.ui.radioModeloRegularized.setChecked(True)
81      self.ui.lineConstanteSmagorinsky.setText(self.tree.xpath('/
        configuracion/numerics/cSmago')[0].text)    # Constante de Smagorinsky.
82      if self.tree.xpath('/configuracion/numerics/useParticles')[0].text=='
        True':    # Usar particulas Virtuales.
83          self.ui.checkParticulasVirtuales.setChecked(True)
84      else:
85          self.ui.checkParticulasVirtuales.setChecked(False)
86      self.ui.lineVirtualesTiempo.setText(self.tree.xpath('/configuracion/
        numerics/particleTimeFactor')[0].text)    # Frecuencia de computo de las
        particulas virtuales.
87      self.ui.lineVirtualesInyeccion.setText(self.tree.xpath('/configuracion/
        numerics/particleProbabilityPerCell')[0].text)    # Probabilidad de
        inyección de nuevas particulas.
88      self.ui.lineVirtualesMaximo.setText(self.tree.xpath('/configuracion/
        numerics/maxNumParticlesToWrite')[0].text)    # Número máximo de
        particulas a escribir en el fichero VTK.
89      self.ui.lineVirtualesEliminacion.setText(self.tree.xpath('/
        configuracion/numerics/cutOffSpeedSqr')[0].text)    # Condición de
        eliminación de particulas.
90      self.ui.lineSpongeAnchura.setText(self.tree.xpath('/configuracion/
        numerics/outletSpongeZoneWidth')[0].text)    # Anchura de la Sponge Zone
        .
```

```
91      # Dinámica de la Sponge Zone
92      if self.tree.xpath('/configuracion/numerics/outletSpongeZoneType')[0].
        text=='Viscosity':
93          self.ui.radioSpongeViscoso.setChecked(True)
94      else:
95          self.ui.radioSpongeSmagorinsky.setChecked(True)
96      self.ui.lineSpongeConstanteSmagorinsky.setText(self.tree.xpath('/
        configuracion/numerics/targetSpongeCSmago')[0].text)    # Constante de
        Smagorinsky en la Sponge Zone.
97      self.ui.lineIncrProgresivoT.setText(self.tree.xpath('/configuracion/
        numerics/initialIterT')[0].text)    # Tiempo de aceleración del fluido.
98      self.ui.lineTiempoSimulacion.setText(self.tree.xpath('/configuracion/
        numerics/maxT')[0].text)    # Duración de la simulación.
99      #output
100     self.ui.lineFrecuenciaDatos.setText(self.tree.xpath('/configuracion/
        output/statT')[0].text)    # Frecuencia de volcado de la fuerza.
101     self.ui.lineFrecuenciaVTK.setText(self.tree.xpath('/configuracion/
        output/vtkT')[0].text)    # Frecuencia de volcado del VTK.
102     if self.tree.xpath('/configuracion/output/vtkFile')[0].text=='True':
        # Utilizar volcado VTK.
103         self.ui.checkBoxVTK.setChecked(True)
104     else:
105         self.ui.checkBoxVTK.setChecked(False)
106     #MPI
107     self.ui.lineParametrosMPI.setText(self.tree.xpath('/configuracion/MPI')
        [0].text)
108     #Avisos
109     if self.tree.xpath('/configuracion/email/enviar')[0].text=='True':    #
        Utilizar sistema de aviso por email.
110         self.ui.checkBoxEmail.setChecked(True)
111     else:
112         self.ui.checkBoxEmail.setChecked(False)
113     if self.tree.xpath('/configuracion/telegram/enviar')[0].text=='True':
        # Utilizar sistema de aviso por Telegram.
114         self.ui.checkBoxTelegram.setChecked(True)
115     else:
116         self.ui.checkBoxTelegram.setChecked(False)
117     #Email
118     self.ui.lineEmailEmisor.setText(self.tree.xpath('/configuracion/email/
        emailFuente')[0].text)    # Email desde el que enviar el correo.
119     self.ui.lineEmailDestino.setText(self.tree.xpath('/configuracion/email/
        emailDestino')[0].text)    # Email al que enviar el email.
```

```
120     self.ui.lineEmailMensaje.setText(self.tree.xpath('/configuracion/email/
        msg')[0].text)     # Mensaje a enviar.
121     #Telegram
122     self.ui.lineTelegramTOKEN.setText(self.tree.xpath('/configuracion/
        telegram/TOKEN')[0].text)    # Token del bot.
123     self.ui.lineTelegramID.setText(self.tree.xpath('/configuracion/telegram
        /chat_id')[0].text)     # Id del chat con el bot.
124     self.ui.lineTelegramMensaje.setText(self.tree.xpath('/configuracion/
        telegram/msg')[0].text)     # Mensaje a enviar.
125     ####Conectando Signals y Slots
126     #Lanzar Simulacion
127     self.ui.pushLanzar.clicked.connect(self.lanzarSimulacion) # Al pulsar
        el boton de Lanzar, la señal producida ser recogera por
128     #Salir
129     self.ui.pushSalir.clicked.connect(self.salir)
130   def salir(self):    #Slot para cuando se pulse Salir
131     app.exit(1)
132   def lanzarSimulacion(self):    #Slot para cando se pulse Lanzar.
133     '''
134     Se lee el valor de los campos de texto mediante el método text().Con el
        método isChecked() se verifica si los radio button estan marcados o no.
        Mediante XPath se busca cada nodo del archivo XML. Para buscar el valor
        de la configuracion, se busca el nodo que almacena dicha configuración
        mediante XPath. Como se devuelve una tupla de elementos, se accede al ú
        nico elemento de dicha tupla mediante el operador [0]. Para saber la
        información que almacena dicho nodo, se utiliza el atributo text.
        '''
135
136     #Geometria
137     self.tree.xpath('/configuracion/geometry/filename')[0].text = self.ui.
        lineArchivo.text()    # Nombre del fichero STL.
138     self.tree.xpath('/configuracion/geometry/center/x')[0].text = self.ui.
        linePosicionX.text()    # Coordenada X donde posicionar el obstaculo.
139     self.tree.xpath('/configuracion/geometry/center/y')[0].text = self.ui.
        linePosicionY.text()    # Coordenada Y donde posicionar el obstaculo.
140     self.tree.xpath('/configuracion/geometry/center/z')[0].text = self.ui.
        linePosicionZ.text()    # Coordenada Z donde posicionar el obstaculo.
141     self.tree.xpath('/configuracion/geometry/domain/x')[0].text = self.ui.
        lineDominioX.text()    # Tamaño del túnel en el eje X.
142     self.tree.xpath('/configuracion/geometry/domain/y')[0].text = self.ui.
        lineDominioY.text()    # Tamaño del túnel en el eje Y.
143     self.tree.xpath('/configuracion/geometry/domain/z')[0].text = self.ui.
        lineDominioZ.text()    # Tamaño del túnel en el eje Z.
```

```
144     self.tree.xpath('/configuracion/geometry/Giros/phi')[0].text = self.ui.
    linePhi.text()    # Giro inicial Phi.
145     self.tree.xpath('/configuracion/geometry/Giros/theta')[0].text = self.
    ui.lineTheta.text()    # Giro inicial Theta.
146     self.tree.xpath('/configuracion/geometry/Giros/psi')[0].text = self.ui.
    linePsi.text()    # Giro inicial Psi.
147     if self.ui.radioContornoLibre.isChecked():    # Condiciones de contorno.
148       self.tree.xpath('/configuracion/geometry/lateralFreeSlip')[0].text =
    'True'
149     else:
150       self.tree.xpath('/configuracion/geometry/lateralFreeSlip')[0].text =
    'False'
151
152     if self.ui.radioSolidoDeslizamiento.isChecked():
153       self.tree.xpath('/configuracion/geometry/freeSlipWall')[0].text = '
    True'
154     else:
155       self.tree.xpath('/configuracion/geometry/freeSlipWall')[0].text = '
    False'
156     #Numericos
157     self.tree.xpath('/configuracion/numerics/ro')[0].text = self.ui.
    lineDensidad.text()    # Densidad.
158     self.tree.xpath('/configuracion/numerics/nu')[0].text = self.ui.
    lineViscosidadCinematica.text()    # Viscosidad cinemática.
159     self.tree.xpath('/configuracion/numerics/deltaphi')[0].text = self.ui.
    lineIncrementoPhi.text()    # Incremento del angulo Phi en cada simulació
    n.
160     self.tree.xpath('/configuracion/numerics/deltatheta')[0].text = self.ui
    .lineIncrementoTheta.text()    # Incremento del angulo Theta en cada
    simulación.
161     self.tree.xpath('/configuracion/numerics/deltapsi')[0].text = self.ui.
    lineIncrementoPsi.text()    # Incremento del angulo Psi en cada simulació
    n.
162     self.tree.xpath('/configuracion/numerics/iter')[0].text = self.ui.
    lineNumeroEjecuciones.text()    # Número de iteraciones.
163     self.tree.xpath('/configuracion/numerics/resolution')[0].text = self.ui
    .lineResolucion.text()    # Resolución del eje Y.
164     self.tree.xpath('/configuracion/numerics/dt')[0].text = self.ui.lineDt.
    text()    # Valor dt.
165     # Tipo de dinámica.
166     if self.ui.radioModeloSmagorinsky.isChecked():
167       self.tree.xpath('/configuracion/numerics/modelo')[0].text = '
    Smagorinsky'
```

```
168     elif self.ui.radioModeloBGK.isChecked():
169       self.tree.xpath('/configuracion/numerics/modelo')[0].text = 'BGK'
170     elif self.ui.radioModeloRegularized.isChecked():
171       self.tree.xpath('/configuracion/numerics/modelo')[0].text = '
    Regularized'
172     elif self.ui.radioModeloEntropic.isChecked():
173       self.tree.xpath('/configuracion/numerics/modelo')[0].text = 'Entropic
    '
174     self.tree.xpath('/configuracion/numerics/cSmago')[0].text = self.ui.
    lineConstanteSmagorinsky.text()   # Constante de Smagorinsky.
175     if self.ui.checkParticulasVirtuales.isChecked():   # Usar particulas
    Virtuales.
176       self.tree.xpath('/configuracion/numerics/useParticles')[0].text = '
    True'
177     else:
178       self.tree.xpath('/configuracion/numerics/useParticles')[0].text = '
    False'
179     self.tree.xpath('/configuracion/numerics/particleTimeFactor')[0].text =
     self.ui.lineVirtualesTiempo.text()   # Frecuencia de computo de las
    particulas virtuales.
180     self.tree.xpath('/configuracion/numerics/particleProbabilityPerCell')
    [0].text = self.ui.lineVirtualesInyeccion.text()   # Probabilidad de
    inyección de nuevas particulas.
181     self.tree.xpath('/configuracion/numerics/maxNumParticlesToWrite')[0].
    text = self.ui.lineVirtualesMaximo.text()    # Número máximo de
    particulas a escribir en el fichero VTK.
182     self.tree.xpath('/configuracion/numerics/cutOffSpeedSqr')[0].text =
    self.ui.lineVirtualesEliminacion.text()    # Condición de eliminación de
     particulas.
183     self.tree.xpath('/configuracion/numerics/outletSpongeZoneWidth')[0].
    text = self.ui.lineSpongeAnchura.text()    # Anchura de la Sponge Zone.
184     # Dinámica de la Sponge Zone
185     if self.ui.radioSpongeViscoso.isChecked():
186       self.tree.xpath('/configuracion/numerics/outletSpongeZoneType')[0].
    text = 'Viscosity'
187     else:
188       self.tree.xpath('/configuracion/numerics/outletSpongeZoneType')[0].
    text = 'Smagorinsky'
189     self.tree.xpath('/configuracion/numerics/targetSpongeCSmago')[0].text =
     self.ui.lineSpongeConstanteSmagorinsky.text()   # Constante de
    Smagorinsky en la Sponge Zone.
190     self.tree.xpath('/configuracion/numerics/initialIterT')[0].text = self.
    ui.lineIncrProgresivoT.text()   # Tiempo de aceleración del fluido.
```

```
191     self.tree.xpath('/configuracion/numerics/maxT')[0].text = self.ui.
    lineTiempoSimulacion.text()    # Duración de la simulación.
192     #output
193     self.tree.xpath('/configuracion/output/statT')[0].text = self.ui.
    lineFrecuenciaDatos.text()     # Frecuencia de volcado de la fuerza.
194     self.tree.xpath('/configuracion/output/vtkT')[0].text = self.ui.
    lineFrecuenciaVTK.text()     # Frecuencia de volcado del VTK.
195     if self.ui.checkBoxVTK.isChecked():     # Utilizar volcado VTK.
196       self.tree.xpath('/configuracion/output/vtkFile')[0].text = 'True'
197     else:
198       self.tree.xpath('/configuracion/output/vtkFile')[0].text = 'False'
199     #MPI
200     self.tree.xpath('/configuracion/MPI')[0].text = self.ui.
    lineParametrosMPI.text()
201     #Avisos
202     if self.ui.checkBoxTelegram.isChecked():    # Utilizar sistema de aviso
    por Telegram.
203       self.tree.xpath('/configuracion/telegram/enviar')[0].text = 'True'
204     else:
205       self.tree.xpath('/configuracion/telegram/enviar')[0].text = 'False'
206     if self.ui.checkBoxEmail.isChecked():     # Utilizar sistema de aviso
    por email.
207       self.tree.xpath('/configuracion/email/enviar')[0].text = 'True'
208     else:
209       self.tree.xpath('/configuracion/email/enviar')[0].text = 'False'
210     #Telegram
211     self.tree.xpath('/configuracion/telegram/TOKEN')[0].text = self.ui.
    lineTelegramTOKEN.text()    # Token del bot.
212     self.tree.xpath('/configuracion/telegram/chat_id')[0].text = self.ui.
    lineTelegramID.text()     # Id del chat con el bot.
213     self.tree.xpath('/configuracion/telegram/msg')[0].text = self.ui.
    lineTelegramMensaje.text()     # Mensaje a enviar.
214     #Email
215     self.tree.xpath('/configuracion/email/emailFuente')[0].text = self.ui.
    lineEmailEmisor.text()     # Email desde el que enviar el correo.
216     self.tree.xpath('/configuracion/email/emailDestino')[0].text = self.ui.
    lineEmailDestino.text()     # Email al que enviar el email.
217     self.tree.xpath('/configuracion/email/msg')[0].text = self.ui.
    lineEmailMensaje.text()     # Mensaje a enviar.
218
219     #Escribir el nuevo fichero XML
220     self.nuevoXML = etree.tostring(self.tree, pretty_print=True).decode("
    utf-8")    # Se transforma la representación interna que utiliza para
```

```
    representar los nodos a texto plano. De utiliza la codificación Unicode.
221     self.f = open('externalFlowAroundObstacle.xml', 'w')   # Se abre el
    fichero.
222     self.f.write(self.nuevoXML)   # Se escribe en el fichero.
223     self.f.close()   # Se cierra el fichero.
224     self.close()   # Se cierra la ventana.
225 if __name__ == "__main__":    #Solo se cumple si lo estamos ejecutando
    directamente, no si lo cargamos a otro modulo, punto de partida del
    programa
226   app = QtWidgets.QApplication(sys.argv)    #Funcion necesaria en cualquier
      aplicacion Qt, se utilizara para lanzar el bucle principal del programa
      que caputara los eventos.
227   myapp = ventana()    # Se carga la interfaz grafica que se ha definido
      dentro del objeto ventana.
228   myapp.show()    # Mostrar la ventana creada
229   estado=app.exec_()    # Se lanza el bucle principal del programa,
      encargado de capturar los eventos, o en nomenclatura Qt, las señales que
      estos producen. Almacenando el resultado en la variable estado
      conseguimos que el programa no finalice directamente al acabar dicho
      bucle, salvo que se pulse al boton de salir, en cuyo caso se devolver un
      valor de 1
230   if estado!=1:    # Si no se ha pulsado salir
231     ruta = os.popen('which mpiexec').readline()[:-1]    # Para saber en que
      ruta esta el programa mpiexec y usarlo para lanzar el programa, se
      localiza mediante el comando which de la consola de linux. Lanzamos este
      comando con la función popen. Usando la función readline se lee la
      primera linea del comando. Posteriormente se elimina el ultimo caracter,
      el salto de linea /n, mediante el operador de python para cadenas
      [:-1], donde indicamos que seleccione toda la cadena menos el ultimo
      caracter.
232     parametrosMPI=['mpiexec']    #Lista que almacenara las opciones que se
      le pasaran a mpiexec
233     for i in myapp.ui.lineParametrosMPI.text().split(' '):    #Del cuadro
      de texto lineParametrosMPI de la interfaz se lee el texto mediante el mé
      todo text(), y separamos cada uno de los parametros separados mediante
      el espacio con el método split.
234       parametrosMPI.append(i)    #Se agrega cada parametro a la lista
      mediante el método append.
235     parametrosMPI.append('externalFlowAroundObstacle')    #Se añade como
      el nombre del programa que ejecuta el tunel.
236     parametrosMPI.append('externalFlowAroundObstacle.xml')    #Se añade
      como parametro tambien nombre del archio de configuración XML creado.
```

```
237     if myapp.ui.lineEmailPassword.text()!='':     # Se verifica que el texto
        en el cuadro de texto que tiene la contraeña del correo no esta en
       blanco
238       parametrosMPI.append(myapp.ui.lineEmailPassword.text())     # En caso
       de que no estuviera vacia, se añade tambien a la lista de parametros.
239     print(ruta, parametrosMPI)     # Se imprime por pantalla el comando que
       se va a ejecutar, es solo por información.
240     """
241     Se llama a la syscall execv, la cual hace que el programa actual deje
       de ejecutarse y cambie al programa que se le indique.
242     Como primer parametro se le indica el ejecutable al que se quiere pasar
       , en nuestro caso la ruta a mpiexec, que se encargará de lanzar el tunel
        compilado con MPI.
243     Como segundo parametro se le indica la lista con las opciones a pasarle
       .
244     """
245     os.execv(ruta, parametrosMPI)
246   sys.exit(estado)     # Si se pulsa al botón salir se sale del programa sin
       lanzar la ejecución.
```

B.2 Túnel de Viento Virtual

El código en C++ es el encargado de realizar la simulación, el fichero XML es un ejemplo de fichero de configuración, el cual es generado por la interfaz del programa. Un vez finaliza la simulación, el programa lanza el sistema de avisos, el cual puede verse en la sección B.3.

```
1  /*
2   * This file is part of the U-Virtual Wind Tunnel.
3   *
4   * Copyright (C) 2017
5   *
6   * The U-Virtual Wind Tunnel is free software: you can redistribute it and/
        or
7   * modify it under the terms of the GNU Affero General Public License as
8   * published by the Free Software Foundation, either version 3 of the
9   * License, or (at your option) any later version.
10  *
11  * The U-Virtual Wind Tunnel is distributed in the hope that it will be
        useful,
12  * but WITHOUT ANY WARRANTY; without even the implied warranty of
13  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
```

```
14  * GNU Affero General Public License for more details.
15  *
16  * You should have received a copy of the GNU Affero General Public License
17  * along with this program.  If not, see <http://www.gnu.org/licenses/>.
18 */
19
20 //Incluidas cabeceras de Palabos
21 #include "palabos3D.h"
22 #include "palabos3D.hh"
23
24 #include <sys/types.h> //Cabecera para fork
25 #include <sys/wait.h> // Cabecera para wait
26 #include <unistd.h> // Cabecera para execl
27
28 //Añadidos namespace que necesitamos
29 using namespace plb;
30 using namespace std;
31
32 /*
33 Plantilla creada para transformar
34 TODO: ver si se puede con la misma funcion pero en std http://stackoverflow
      .com/questions/5590381/easiest-way-to-convert-int-to-string-in-c
35 http://en.cppreference.com/w/cpp/string/basic_string/to_string
36 */
37 namespace util    // Namespace utilizado para almacenar clases y funcione
      utiles para ciertas tareas.
38 {
39     template < typename TT > std::string to_string( const TT& n )    //
      Funcion para convertir a un string una variable de tipo arbitraria TT.
40     {
41         std::ostringstream stm ;    // º1 -> Creamos un objeto de clase
      ostringstream, un objeto de clase "string buffer" muy util para
      operaciones de salida con strings.
42         stm << n ;    // º2 -> mediante el operador << le asignamos el valor
       de la variable que queremos pasar, este operador ya se encuentra
      sobrecargado para los diferetes tipos de datos posibles, de ahi su gran
      utilidad para poder ser utilizado con diversos tipos de datos.
43         return stm.str() ;    // º3 -> Mediante el método str() recuperamos
      el valor de la variable pasada como un string.
44     }
45 }
46
47
```

```
48
49 typedef double T; //T indica el tipo de dato utilizado para almacenar
       decimales. Si no hay problemas de memoria se utiliza double, si los
       hubiera, utilizar float.
50 typedef Array<T,3> Velocity; //Tipo de dato utilizado para almacenar
       velocidades. Un array de 3 dimensiones que almacena valores decimales.
51 #define DESCRIPTOR descriptors::D3Q19Descriptor     // Tipo de mallado a
       utilizar.
52 typedef DenseParticleField3D<T,DESCRIPTOR> ParticleFieldT;     //Tipo de
       campo de particulas virtuales a utilizar.
53
54 #define PADDING 8     // Número de digitos a mostrar en el nombre del
       fichero VTK.
55
56 static std::string outputDir("./tmp/"); //Directorio donde volcar los
       resultados
57
58
59 // Estructura que contiene todos los parametros del tunel y
60 // y deriva los necesarios para su utilización.
61 struct Param
62 {
63     T nu;                                  // Velocidad cinemática
64     T lx, ly, lz;                          // Tamaño del espacio fluido, en
       unidades fisicas
65     T cx, cy, cz;                          // Posicion del centro del
       obstaculo, en unidades fisicas
66     plint cxLB, cyLB, czLB;                // Posicion del centro del
       obstaculo, en unidades lattice
67     bool freeSlipWall;                     // Usar condiciones free-slip en el
        obstaculo
68     bool lateralFreeSlip;                  // Usar condicion free-slip en los
       laterales o condicion de suelo
69     T maxT, statT, vtkT;                   // Tiempo de duracion de la
       simulacion, frecuencia con el que se vuelcan los datos de simulacion en
       pantalla, frecuencia con el que se vuelcan los datos en VTK
70     plint resolution;                      // Número de nodos a lo largo de la
        dirección Y
71     T inletVelocity;                       // Velocidad de entrada en el eje X
       , unidades fisicas
72     T uLB;                                 // Velocidad de entrada en el eje X
       , unidades lattice.
```

```
73    T cSmago;                          // Parametro del modelo Smagorinsky
      LES.
74    plint nx, ny, nz;                  // Resolución de la malla.
75    T omega;                           // Parametro de relajacion
76    T dx, dt;                          // Incremento discreto del espacio
      y el tiempo.
77    plint maxIter, statIter;           // Número total de iteraciones, y
      numero de iteraciones entre volcado en pantalla.
78    plint vtkIter;                     // Número de iteraciones entre
      volcado en VTK
79    bool useParticles;                 // Usar particulas virtuales o no.
80    int particleTimeFactor;            // Factor de tiempo con el cual se
      calcula el flujo de particulas virtuales.
81    T particleProbabilityPerCell;      // Probabilidad de inyeccion de una
      nueva particula virtual en cada iteración.
82    T cutOffSpeedSqr;                  // Criterio de velocidad minima
      para eliminar particulas
83    int maxNumParticlesToWrite;        // Numero maximo de particulas a
      escribir en VTK
84
85    T outletSpongeZoneWidth;           // Espesor de la Sponge-Zone
86    plint numOutletSpongeCells;        // Número de nodos que ocupa la
      Sponge-Zone
87    int outletSpongeZoneType;          // Tipo de Sponge-Zone (Viscosity o
      Smagorinsky).
88    T targetSpongeCSmago;              // En caso de que la Sponge-Zone
      sea de tipo Smagorinsky, el valor de la contante de Smagorinsky a
      utilizar.
89    plint initialIter;                 // Numero de iteaciones hasta que
      la velocidad alcanza su valor final.
90
91    Box3D inlet, outlet, lateral1;     // Entrada, Salida y laterales del
      tunel
92    Box3D lateral2, lateral3, lateral4;
93
94    std::string geometry_fname;        // Nombre del fichero STL
95
96    // Parametros propios
97
98    plint iteracionesPrograma;       // Numero de ejecuciones del programa
      en las que se cambian los angulos entre ellas.
99    T Cs;       // Velocidad del sonido en el fluido.
100   T densidad;       // Dendisas del fluido.
```

```
101     T initialIterT;      // Numero de iteraciones durante el cual la
    velocidad en la entrada del tunel se va acelerando
102     bool vtkFile;            // Exportar los datos en VTK o no.
103     T phi;                   // Ángulo Phi
104     T theta;                   // Ángulo theta
105     T psi;                   // Ángulo Psi
106     T deltaphi;        // Incremento en phi en cada iteracion
107     T deltatheta;        // Incremento en theta en cada iteracion
108     T deltapsi;        // Incremento en psi en cada iteracion
109     std::string modelo;      // Nombre de la dinamica que se va a utilizar
110
111     Param()                  // Funcion Param llamada sin parametros.
112     { }
113
114     Param(std::string xmlFname)     //Funcion Param llamada con el nombre
    del fichero XML a procesar.
115     {
116         XMLreader document(xmlFname);     // XMLreader es un objeto que abre
     y procesa el documento XML. Posteriormente a traves del operador [] es
    posible acceder a los distintos nodos del fichero y escribirlos en las
    distintas variables gracias al metodo read.
117     // Leemos los nodos del fichero XML que nos interesa y almacenamos la
    informacion en las variables de la estructura Param.
118         document["configuracion"]["geometry"]["filename"].read(
    geometry_fname);
119         document["configuracion"]["geometry"]["center"]["x"].read(cx);
120         document["configuracion"]["geometry"]["center"]["y"].read(cy);
121         document["configuracion"]["geometry"]["center"]["z"].read(cz);
122         document["configuracion"]["geometry"]["freeSlipWall"].read(
    freeSlipWall);
123         document["configuracion"]["geometry"]["lateralFreeSlip"].read(
    lateralFreeSlip);
124         document["configuracion"]["geometry"]["domain"]["x"].read(lx);
125         document["configuracion"]["geometry"]["domain"]["y"].read(ly);
126         document["configuracion"]["geometry"]["domain"]["z"].read(lz);
127         document["configuracion"]["geometry"]["Giros"]["phi"].read(phi);
128         document["configuracion"]["geometry"]["Giros"]["theta"].read(theta)
    ;
129         document["configuracion"]["geometry"]["Giros"]["psi"].read(psi);
130         document["configuracion"]["numerics"]["iter"].read(
    iteracionesPrograma);
131         document["configuracion"]["numerics"]["nu"].read(nu);
132         document["configuracion"]["numerics"]["ro"].read(densidad);
```

```
133         document["configuracion"]["numerics"]["inletVelocity"].read(
    inletVelocity);
134         document["configuracion"]["numerics"]["deltaphi"].read(deltaphi);
135         document["configuracion"]["numerics"]["deltatheta"].read(deltatheta
    );
136         document["configuracion"]["numerics"]["deltapsi"].read(deltapsi);
137         document["configuracion"]["numerics"]["resolution"].read(resolution
    );
138         document["configuracion"]["numerics"]["dt"].read(dt);
139         document["configuracion"]["numerics"]["modelo"].read(modelo);
140         if (modelo == "Smagorinsky") {
141             document["configuracion"]["numerics"]["cSmago"].read(cSmago);
      // Solo se lee la variable del modelo de Smagorinsky si utilizamos
    dicho modelo
142         }
143         document["configuracion"]["numerics"]["useParticles"].read(
    useParticles);
144         if (useParticles) {      // Solo se leen las variables del modelo de
     particulas virtuales si utilizamos dicho modelo
145             document["configuracion"]["numerics"]["particleTimeFactor"].
    read(particleTimeFactor);
146             document["configuracion"]["numerics"]["
    particleProbabilityPerCell"].read(particleProbabilityPerCell);
147             document["configuracion"]["numerics"]["cutOffSpeedSqr"].read(
    cutOffSpeedSqr);
148             document["configuracion"]["numerics"]["maxNumParticlesToWrite"
    ].read(maxNumParticlesToWrite);
149         }
150         document["configuracion"]["numerics"]["outletSpongeZoneWidth"].read
    (outletSpongeZoneWidth);
151         std::string zoneType;    //Dinamica usada en la Sponge Zone
152         document["configuracion"]["numerics"]["outletSpongeZoneType"].read(
    zoneType);  // Almacenamos el nombre del tipo de zona utilizada en la
    Sponge Zone
153
154         if ((util::tolower(zoneType)).compare("viscosity") == 0) {
155             outletSpongeZoneType = 0;     //Si es de tipo viscoso damos a
    outletSpongeZoneType un valor de 0
156         } else if ((util::tolower(zoneType)).compare("smagorinsky") == 0) {
157             outletSpongeZoneType = 1; //Si es de tipo Smagorinsky damos a
    outletSpongeZoneType un valor de 1
158         } else {     // En otro caso mostramos un mensaje de error
```

```
159             pcout << "The sponge zone type must be either \"Viscosity\" or
    \"Smagorinsky\"." << std::endl;     // Imprimir datos por pantalla, su
    unico proposito es informar.
160             exit(-1);
161         }
162
163         document["configuracion"]["numerics"]["targetSpongeCSmago"].read(
    targetSpongeCSmago);
164         document["configuracion"]["numerics"]["initialIterT"].read(
    initialIterT);
165         document["configuracion"]["numerics"]["maxT"].read(maxT);
166         document["configuracion"]["output"]["statT"].read(statT);
167         document["configuracion"]["output"]["vtkFile"].read(vtkFile);
168      if (vtkFile){
169        document["configuracion"]["output"]["vtkT"].read(vtkT);     // Solo se
     establece la frecuencia de volcado de ficheros VTK si usamos dicho
    modelo
170      }
171
172         computeLBparameters();     // Llamamos a la funcion que termina de
    establecer los parametros que faltan por establecer su valor, y se
    calcula a partir de lso datos que hemos leido anteriormente.
173      }
174
175      void computeLBparameters()
176      {
177      dx = ly / (resolution - 0.0);     // Al ser resolucion de tipo entero,
    para que la division de un resultado flotante se le resta el valor 0.0.
    Dividiendo la longitud del tunel en el eje Y por la resolucion que le
    damos obtenemos el incremento discreto de distancia, utilizado como
    unidad de distancia Lattice.
178    Cs = dx/dt;     // Velocidad del sonido en el fluido.
179    uLB = inletVelocity/Cs;     // Adimensionalización de la velocidad del
     fluido mediante la velocidad del sonido en el fluido.
180     T nuLB = nu * dt/(dx*dx);     // Adimensionalizacion de la viscosidad
     del fluido.
181    initialIter = util::roundToInt(initialIterT/dt);     // Número de
     iteraciones a realizar hasta que qla velocidad de entrada del fluido
     alcanza el valor establecido. Lo obtenemos adimensionalizando el tiempo
     que se tarda en alcanzarlo con el incremento temporal discreto de la
     simulación.
182      omega = 1.0/(DESCRIPTOR<T>::invCs2*nuLB+0.5);     // Parámetro de
     relajación, calculado segun la formula necesaria para nuestro caso.
```

```
183
184     nx = util::roundToInt(lx/dx) + 1;     // Número de nodos Lattice a lo
    largo del eje X, redondeo a entero de longitud del tunel en el eje X
    dividido incremento discreto de distancia más uno.
185     ny = util::roundToInt(ly/dx) + 1;     // Número de nodos Lattice a lo
    largo del eje Y, redondeo a entero de longitud del tunel en el eje Y
    dividido incremento discreto de distancia más uno.
186     nz = util::roundToInt(lz/dx) + 1;     // Número de nodos Lattice a lo
    largo del eje Z, redondeo a entero de longitud del tunel en el eje Z
    dividido incremento discreto de distancia más uno.
187
188     cxLB = util::roundToInt(cx/dx);     // Posicion del centro del obstaculo
     en el eje X en unidades Lattice,  redondeo a entero de la posición del
    centro del obstaculo en el eje X dividido incremento discreto de
    distancia.
189     cyLB = util::roundToInt(cy/dx);     // Posicion del centro del obstaculo
     en el eje Y en unidades Lattice,  redondeo a entero de la posición del
    centro del obstaculo en el eje Y dividido incremento discreto de
    distancia.
190     czLB = util::roundToInt(cz/dx);     // Posicion del centro del obstaculo
     en el eje Z en unidades Lattice,  redondeo a entero de la posición del
    centro del obstaculo en el eje Z dividido incremento discreto de
    distancia.
191     maxIter    = util::roundToInt(maxT/dt);     // Número total de
    iteraciones de la simulación. Redondeando a entero el tiempo total de la
     simulación dividido entre incremento discreto de tiempo.
192     statIter   = util::roundToInt(statT/dt);     // Número de iteraciones
    entre volcado de la fuerza sobre el sólido. Redondeando a entero el
    tiempo entre volcados dividido entre incremento discreto de tiempo.
193     vtkIter    = util::roundToInt(vtkT/dt);     // Número de iteraciones
    entre volcado del fichero VTK. Redondeando a entero el tiempo entre
    volcados dividido entre incremento discreto de tiempo.
194
195     numOutletSpongeCells = util::roundToInt(outletSpongeZoneWidth/dx);
    // Número de nodos en la Sponge Zone. Redondeando a entero la longitud
    de esta entre el incremento discreto de distancia.
196     // Las caras del tunel esta delimitado entre objetos de la clase Box3D,
     que representan cada una de las caras rectangulares o cuadradas del
    tunel. Para definir cada una de las caras se sigue la nomenclatura (
    Posicion X de uno de los vertices, Posicion X del vertice opuesto,
    Posicion Y de uno de los vertices, Posicion Y del vertice opuesto,
    Posicion Z de uno de los vertices, Posicion Z del vertice opuesto);
```

```
197  inlet     = Box3D(0,       0,       0,       ny-1,    0,       nz-1);     //
      Cara de entrada
198   outlet    = Box3D(nx-1,    nx-1,    0,       ny-1,    0,       nz-1);     //
      Cara de salida
199   lateral1  = Box3D(1,       nx-2,    0,       0,       0,       nz-1);     //
      Lateral 1
200   lateral2  = Box3D(1,       nx-2,    ny-1,    ny-1,    0,       nz-1);     //
      Lateral 2
201   lateral3  = Box3D(1,       nx-2,    1,       ny-2,    0,       0);     //
      Lateral 3
202   lateral4  = Box3D(1,       nx-2,    1,       ny-2,    nz-1,    nz-1);     //
      Lateral 4
203
204   }
205
206   Box3D boundingBox() const     // Función de utilidad, devuelve todo el
      dominio fluido como un poliedro de 6 caras rectangulares. Se definde con
       la nomenclatura anterior con un vertice en (0,0,0) y la opuesta en (nú
      mero de vértices en eje X - 1, número de vértices en eje Y - 1, número
      de vértices en Z eje - 1)
207   {
208       return Box3D(0, nx-1, 0, ny-1, 0, nz-1);
209   }
210
211   T getInletVelocity(plint iIter)     // Como la velocidad del fluido en
      la entrada varia con el tiempo se establece esta funcion que nos indica
      para la iteración iIter, que velocidad corresponde en la entrada.
212   {
213       static T pi = std::acos((T) -1.0);     // Número pi
214
215       if (iIter >= initialIter) {     // Hasta la iteración initialIter la
       velocidad se va incrementando progresivamente siguiendo una funcion
      senoidal. A partir de entonces el valor permanece constante.
216           return uLB;
217       }
218
219       if (iIter < 0) {     // Si hubier una iteracion negativa, cosa
      imposible, se reasignaria a un vaor de 0.
220           iIter = 0;
221       }
222
223       return uLB * std::sin(pi * iIter / (2.0 * initialIter));     //
      Funcion que incrementa progresivamente la velocidad en la entrada
```

```
    siguiendo una forma senoidal.
224     }
225 };
226
227 Param param;     // Declaramos la variable global param de tipo Param.
228
229 // Función que establece las condiciones de contorno de las paredes
    laterales del tunel.
230 void outerDomainBoundaries(MultiBlockLattice3D<T,DESCRIPTOR> *lattice,
    // Bloque Lattice que contiene la dinamica del fluido.
231                            MultiScalarField3D<T> *rhoBar,     // Bloque
    rhobar, utilizado para procesar la densidad
232                            MultiTensorField3D<T,3> *j,     // Bloque j,
    utilizado para procesar la velocidad
233                            OnLatticeBoundaryCondition3D<T,DESCRIPTOR> *bc)
       // Bloque que contiene métodos necesarios para establecer las
    condiciones de contorno.
234 {
235     Array<T,3> uBoundary(0.0, 0.0, 0.0);     // Velocidad de las paredes
    laterales.
236   Array<T,3> velSuelo(0.0, 0.0, 0.0);     // Velocidad del fluido en
    contacto con el suelo.
237
238     if (param.lateralFreeSlip) {     // Condicion de laterales libres.
239         pcout << "Laterales sin deslizamiento." << std::endl;     //
    Imprimir información por pantalla, su unico proposito es informar.
240
241         lattice->periodicity().toggleAll(false);     // Parametro necesario
    en cada bloque, solo es necesario cambiarlo a True en caso de que se
    utilicen condiciones periodicas.
242         rhoBar->periodicity().toggleAll(false);
243         j->periodicity().toggleAll(false);
244
245         bc->setVelocityConditionOnBlockBoundaries(*lattice, param.inlet,
    boundary::dirichlet);     // En la entrada establecemos una condicion de
    contorno de tpo Dirichlet, en dicho tipo se establece un valor
    determinado para la variable.
246         setBoundaryVelocity(*lattice, param.inlet, uBoundary);     // En
    este caso establecemos un valor inicial para la velocidad de (0,0,0),
    este valor sera actualizado en cada iteración de la simulación hasta
    llegar a un valor final.
247
```

```
248     bc->setVelocityConditionOnBlockBoundaries(*lattice, param.lateral1,
    boundary::freeslip);    // Los laterales del tunel se establecen como
    freeslip, esto implica un gradiente cero de la velocidad en la dirección
     tangencial al lateral y un valor fijo de cero en el resto de las
    componentes.
249     bc->setVelocityConditionOnBlockBoundaries(*lattice, param.lateral2,
    boundary::freeslip);
250     bc->setVelocityConditionOnBlockBoundaries(*lattice, param.lateral3,
    boundary::freeslip);
251     bc->setVelocityConditionOnBlockBoundaries(*lattice, param.lateral4,
    boundary::freeslip);
252     setBoundaryVelocity(*lattice, param.lateral1, uBoundary);    //
    Establecemos un valor inicial a la velocidad de (0,0,0) en cada uno de
    los laterales.
253     setBoundaryVelocity(*lattice, param.lateral2, uBoundary);
254     setBoundaryVelocity(*lattice, param.lateral3, uBoundary);
255     setBoundaryVelocity(*lattice, param.lateral4, uBoundary);
256
257     // Las condiciones en la salida son las más complejas, ya que en el
     tratamiento interno del problema lattice boltzmann implican una mayor
    cantidad de operaciones.
258     Box3D globalDomain(lattice->getBoundingBox());    // Variable que
    almacena todo el dominio del fluido, se obtiene mediante una llamada a
    la función getBoundingBox explicada más arriba.
259     std::vector<MultiBlock3D*> bcargs;    // Objeto que almacena los
    bloques lattice, rhobar y j. Necesario para funciones que requieren esta
     forma más compacta para pasar las variables.
260     bcargs.push_back(lattice);    // Introducimos cada una de las
    variables mediante el metodo push_back.
261     bcargs.push_back(rhoBar);
262     bcargs.push_back(j);
263     T outsideDensity = 1.0;    // Densidad del fluido a la salida
264     int bcType = 1;    // Parametro de la función VirtualOutlet que
    establece el tipo de dominio a usar en la salida.
265     integrateProcessingFunctional(new VirtualOutlet<T,DESCRIPTOR>(
    outsideDensity, globalDomain, bcType),
266             param.outlet, bcargs, 2);  // Con
    integrateProcessingFunctional asociamos una función, en este caso
    VirtualOutlet, que se ejecuta cada vez que uno de los bloques indicados
    en el vector bcargs ejecute el metodo executeInternalProcessors. El ú
    ltimo parámetro indica la prioridad de ejecución de dicha función para
    el caso en el que haya más de una.
```

```
267         setBoundaryVelocity(*lattice, param.outlet, uBoundary);     //
    Condiciones de la simulación en la salida del tunel. Se empieza con una
    velocidad de (0,0,0)
268     } else {     // Condicion de existencia de suelo.
269         pcout << "Condiciones de Suelo." << std::endl;     // Imprimir datos
     por pantalla, su unico proposito es informar.
270 
271         lattice->periodicity().toggleAll(false);      // Parametro necesario
     en cada bloque, solo es necesario cambiarlo a True en caso de que se
    utilicen condiciones periodicas.
272         rhoBar->periodicity().toggleAll(false);
273         j->periodicity().toggleAll(false);
274 
275         bc->setVelocityConditionOnBlockBoundaries(*lattice, param.inlet,
    boundary::dirichlet);     // En la entrada establecemos la misma
    condicion que para el caso anterior.
276         setBoundaryVelocity(*lattice, param.inlet, uBoundary);
277     bc->setVelocityConditionOnBlockBoundaries(*lattice, param.lateral1,
    boundary::dirichlet);     // El lateral con coordenada Y = 0 actuara como
     suelo, para ello establecemos en el una condición de Dirichlet y
    fijando la velocidad del fluido a (0,0,0)
278         setBoundaryVelocity(*lattice, param.lateral1, velSuelo);
279 
280         bc->setVelocityConditionOnBlockBoundaries(*lattice, param.lateral2,
     boundary::freeslip);     // Las condiciones en el resto de los laterales
     y parte posterior del tunel son las mismas que para el caso anterior.
281         bc->setVelocityConditionOnBlockBoundaries(*lattice, param.lateral3,
     boundary::freeslip);
282         bc->setVelocityConditionOnBlockBoundaries(*lattice, param.lateral4,
     boundary::freeslip);
283         setBoundaryVelocity(*lattice, param.lateral2, uBoundary);
284         setBoundaryVelocity(*lattice, param.lateral3, uBoundary);
285         setBoundaryVelocity(*lattice, param.lateral4, uBoundary);
286 
287         Box3D globalDomain(lattice->getBoundingBox());
288         std::vector<MultiBlock3D*> bcargs;
289         bcargs.push_back(lattice);
290         bcargs.push_back(rhoBar);
291         bcargs.push_back(j);
292         T outsideDensity = 1.0;
293         int bcType = 1;
294         integrateProcessingFunctional(new VirtualOutlet<T,DESCRIPTOR>(
    outsideDensity, globalDomain, bcType),
```

```
295                     param.outlet, bcargs, 2);
296             setBoundaryVelocity(*lattice, param.outlet, uBoundary);
297         }
298 }
299
300 // Funcion que vuelca el estado del fluido en formato VTK.
301 void writeVTK(OffLatticeBoundaryCondition3D<T,DESCRIPTOR,Velocity>& bc,
        plint iT, int j)
302 {
303     VtkImageOutput3D<T> vtkOut(createFileName("volume_iter_"+util::
        to_string(j)+"_", iT, PADDING));     // Creamos el fichero y le asignamos
         una codificación que nos ayude a identificar cada fichero en caso de
        que estempos realizando una misma simulación cambiando los angulos del
        cuerpo. Un comienzo comun para todos con la cadena "volume_iter_", a la
        que se añade el número de giros que se le han realizado al sólido y por
        último la iteración sobre la cual se realiza el volcado. En caso de que
        el número de la iteración tenga menos dígitos que los definidos en
        PADDING, se rellena con 0 hasta igualar dichos digitos.
304     vtkOut.writeData<float>(*bc.computeVelocityNorm(param.boundingBox()),
           // param.boundingBox() devuelve el espacio fluido, sobre el cual se
        calcula la velocidad normal mediante el método bc.computeVelocityNorm.
        Para pasarlo de unidades Lattice a las unidades en las que se definio el
         problema se multiplica por la unidad de distancia Lattice dividido
        unidad de tiempo Lattice, dicho valor de cambio de escala es el indicado
         en el tercer parametro del método de volcado de datos del objeto que
        identifica al fichero creado. Cada campo exportado al formato VTK debe
        tener un nombre asociado, dicho nombre es el que se indica en el segundo
         parametro del método, en este caso velocityNorm. En la plantilla
        indicamos que es de tipo float para indicar que cada valor decimal tiene
         que usar 32 bits de espacio para almacenar el valor.
305                                 "velocityNorm", param.dx/param.dt );
306     vtkOut.writeData<3,float>(*bc.computeVelocity(param.boundingBox()), "
        velocity", param.dx/param.dt);     // Mismo caso que el inmediatamente
        superior, pero en este caso volcamos volcamos el valor completo de toda
        la velocidad.
307     vtkOut.writeData<float>(*bc.computePressure(param.boundingBox()),
308                                 "pressure", param.densidad*util::sqr(param.dx/
        param.dt)) );     // Volcado del valor de la presión. El factor de cambio
         de dimensiones en este caso es densidad*unidad de distancia Lattice al
        cuadrado dividido unidad de tiempo Lattice al cuadrado.
309 }
310
```

```
311 void runProgram(int jj)    // Función que realiza la simulación en si,
    recibe un parametro ya que puede ejecutarse más de una vez para el mismo
     cuerpo simplemente cambiando la rotación de este, el parametro indica
    el número de veces que se ha rotado el cuerpo.
312 {
313     /*
314      * Lectura del fichero que contiene el cuerpo sólido.
315      */
316
317     pcout << std::endl << "Reading STL data for the obstacle geometry." <<
    std::endl;
318     Array<T,3> center(param.cx, param.cy, param.cz);     // Array con la
    posicion del centro del obstaculo en unidades fisicas
319     Array<T,3> centerLB(param.cxLB, param.cyLB, param.czLB);     // Array
    con la posicion del centro del obstaculo en unidades lattice
320     TriangleSet<T> triangleSet(param.geometry_fname, DBL);     // Objeto que
     define la superficie de la geometria
321
322     // Posicionamiento del obstaculo en las coordenadas idicadas.
323     // El centro del obstaculo es calculado por el programa de manera
    automtica,
324   // como la media aritmetica de las posiciones de sus aristas del ortoedro
325   // que lo circunscribe, representado por el objeto bCuboid.
326     Cuboid<T> bCuboid = triangleSet.getBoundingCuboid();
327     Array<T,3> obstacleCenter = (T) 0.5 * (bCuboid.lowerLeftCorner +
    bCuboid.upperRightCorner);
328     triangleSet.translate(-obstacleCenter);
329
330     triangleSet.scale(1.0/param.dx); // A partir de ahora en unidades
    lattice
331     triangleSet.rotate(param.phi,param.theta,param.psi);     // Rotamos el
    obstculo los angulos indicados
332     triangleSet.translate(centerLB);     // Trasladamos el centro del cuerpo
     a las coordenadas indicadas
333     triangleSet.writeBinarySTL(outputDir+"obstacle_LB_"+util::to_string(jj)
    +".stl");     // Guardamos el nuevo obstaculo en un STL nuevo para poder
    ser visualizado posteriormente.
334
335     plint xDirection = 0;     // Direccion de referencia, el primer elemento
     de cada vector es la componente X.
336     plint borderWidth = 1;       // sobreespesor del borde para el proceso
    de "Voxelizado"
337                                  // Obligatorio: margin>=borderWidth.
```

```
338     plint margin = 1;           // Margen extra asignado a las celdas
    alrededor del obstaculo, usado en caso de paredes moviles.
339     plint blockSize = 0;        // Un valor de cero significa no usar una
    representacion dispersa de las matrices
340
341     DEFscaledMesh<T> defMesh(triangleSet, 0, xDirection, margin, Dot3D(0,
    0, 0));     // Objeto de uso temporal para definir el objeto que se
    define a continuacion
342     TriangleBoundary3D<T> boundary(defMesh);     // Objeto que identifica
    las fronteras del solido
343
344     pcout << "tau = " << 1.0/param.omega << std::endl;     // Imprimir datos
     por pantalla, su unico proposito es informar.
345     pcout << "dx = " << param.dx << std::endl;
346     pcout << "dt = " << param.dt << std::endl;
347   pcout << "uLB = " << param.uLB << std::endl;
348     pcout << "Numero de iteraciones: " << maxIter << std::endl;
349
350     /*
351      * "Voxelizar" (del ingles Voxelize) el dominio.
352      */
353
354     // Voxalizar (Voxelize) el dominio: Decidir que nodos lattice
    pertenecen al obstculo y cuales al dominio del fluido.
355     pcout << std::endl << "Voxelizando el dominio." << std::endl;     //
    Imprimir datos por pantalla, su unico proposito es informar.
356     plint extendedEnvelopeWidth = 2;    // Parametro del proceso de "
    voxelizado" que indca el sobreespesor del cuerpo.
357     const int flowType = voxelFlag::outside;     // Flag que indica que se
    trata de un flujo que circula sobre un cuerpo, la otra opcion es de un
    flujo que circula dentro de un cuerpo.
358     VoxelizedDomain3D<T> voxelizedDomain (
359             boundary, flowType, param.boundingBox(), borderWidth,
    extendedEnvelopeWidth, blockSize );     // Objeto que almacena el dominio
     de nodos lattice "voxelizados"
360     pcout << getMultiBlockInfo(voxelizedDomain.getVoxelMatrix()) << std::
    endl;     // Imprimir datos por pantalla, su unico proposito es informar.
361
362     /*
363      * Generacion de los bloques lattice, rhoBar (densidad) y j (momento)
364      */
365
```

```
366     pcout << "Generating the lattice, the rhoBar and j fields." << std::
    endl;     // Imprimir datos por pantalla, su unico proposito es informar.
367     MultiBlockLattice3D<T,DESCRIPTOR> *lattice = new MultiBlockLattice3D<T,
    DESCRIPTOR>(voxelizedDomain.getVoxelMatrix());     // Creacion del bloque
     lattice a partir del dominio "voxeliado". Este objeto es el más
    importante de todos, ya que contiene todos los datos necesarios de la
    simulación, y a traves de sus metodos se realizan las diversas
    iteraciones del metodo LB.
368     //     Asignación de la dimanica deseada al bloque lattice, se compara
    el valor almacenado en la variable modelo con las distintas dinamicas
    aceptadas por el programa y se asigna dicha dinamica con los parametros
    establecidos.
369     if (param.modelo == "Smagorinsky") {     // Dinamica BGK con un modelo
    de turbulencia de tipo "Large Eddy Simulation"
370         defineDynamics(*lattice, lattice->getBoundingBox(),     //
    defineDynamics es la funcion que asocia al bloque lattice una dinamica,
    dicha dinamica se pasa como un objeto en la tercera posicion de la funci
    ón, la primera es el bloque lattice en si y la segunda es el dominio
    fluido sobre el cual se define.
371                 new SmagorinskyBGKdynamics<T,DESCRIPTOR>(param.omega, param
    .cSmago));
372         pcout << "Using Smagorinsky BGK dynamics." << std::endl;     //
    Imprimir datos por pantalla, su unico proposito es informar.
373     } else if (param.modelo == "BGK") {     // Dinamica BGK simple
374         defineDynamics(*lattice, lattice->getBoundingBox(),
375                 new BGKdynamics<T,DESCRIPTOR>(param.omega));
376         pcout << "Using BGK dynamics." << std::endl;
377     } else if (param.modelo == "Entropic") {     // Usando un modelo
    entropico.
378         defineDynamics(*lattice, lattice->getBoundingBox(),
379                 new EntropicDynamics<T,DESCRIPTOR>(param.omega));
380         pcout << "Using Entropic dynamics." << std::endl;
381     } else if (param.modelo == "Regularized"){     // Usando un modelo BGK
    Regularizado.
382         defineDynamics(*lattice, lattice->getBoundingBox(),
383                 new RegularizedBGKdynamics<T,DESCRIPTOR>(param.omega));
384         pcout << "Using Regularized BGK dynamics." << std::endl;
385     }
386     bool velIsJ = false;     // flag utilizada en el objeto offLatticeModel
    para saber si el bloque de velocidades que se utiliza es j (densidad*
    velocidad)
387     defineDynamics(*lattice, voxelizedDomain.getVoxelMatrix(), lattice->
    getBoundingBox(),
```

```
388             new NoDynamics<T,DESCRIPTOR>(), voxelFlag::inside);     //
    Dinamica nula para los nodos en el interior del sólido.
389     lattice->toggleInternalStatistics(false);     //
    toggleInternalStatistics es un metodo utilizado para obtener
    estadisticas internas del bloque en cada iteración, al ponerlo com false
     desactivamos esta funcionalidad.
390
391     // Los bloques rhoBar y j son usados ambos en the collision y en la
    implementación de las condiciones de salida del tunel.
392     MultiScalarField3D<T> *rhoBar = generateMultiScalarField<T>((
    MultiBlock3D&) *lattice ).release();     // Generación del bloque rhobar,
     a traves del bloque lattice generamos un bloque
    generateMultiScalarField que posteriormente genera un bloque
    MultiScalarField a traves del método release(). Se utiliza este metodo,
    con el bloque lattice como bloque madre, ya que nos interesa que sean
    bloques con identico número de nodos, aunque en los nodos de rhoBar se
    almacenara el valor de la densidad en cada nodo - 1.
393     rhoBar->toggleInternalStatistics(false);     // toggleInternalStatistics
     es un metodo utilizado para obtener estadisticas internas del bloque en
     cada iteración, al ponerlo com false desactivamos esta funcionalidad.
394
395     MultiTensorField3D<T,3> *j = generateMultiTensorField<T,3>((
    MultiBlock3D&) *lattice ).release();     // Generación del bloque j, a
    traves del bloque lattice generamos un bloque generateMultiTensorField
    que posteriormente genera un bloque MultiTensorField3D a traves del mé
    todo release(). Se utiliza este metodo, con el bloque lattice como
    bloque madre, ya que nos interesa que sean bloques con identico número
    de nodos, aunque en los nodos de j se almacenara un vector con los
    valores de la velocidad en cada nodo multiplicado por la densidad en
    dicho nodo.
396     j->toggleInternalStatistics(false);     // toggleInternalStatistics es
    un metodo utilizado para obtener estadisticas internas del bloque en
    cada iteración, al ponerlo com false desactivamos esta funcionalidad.
397
398     std::vector<MultiBlock3D*> lattice_rho_bar_j_arg;     // Objeto que
    almacena los bloques lattice, rhobar y j. Necesario para funciones que
    requieren esta forma más compacta para pasar las variables.
399     lattice_rho_bar_j_arg.push_back(lattice);     // Con el método push_back
     vamos introduciendo cada uno de los bloques.
400     lattice_rho_bar_j_arg.push_back(rhoBar);
401     lattice_rho_bar_j_arg.push_back(j);
402     integrateProcessingFunctional(
403             new ExternalRhoJcollideAndStream3D<T,DESCRIPTOR>(),
```

```
404             lattice->getBoundingBox(), lattice_rho_bar_j_arg, 0);     // Con
    integrateProcessingFunctional asociamos una función, en este caso
    ExternalRhoJcollideAndStream3D, que se ejecuta cada vez que uno de los
    bloques indicados en el vector bcargs ejecute el metodo
    executeInternalProcessors. El último parámetro indica la prioridad de
    ejecución de dicha función para el caso en el que haya más de una.
405     integrateProcessingFunctional(
406             new BoxRhoBarJfunctional3D<T,DESCRIPTOR>(),
407             lattice->getBoundingBox(), lattice_rho_bar_j_arg, 3);      //
    Con integrateProcessingFunctional asociamos una función, en este caso
    BoxRhoBarJfunctional3D, que se ejecuta cada vez que uno de los bloques
    indicados en el vector bcargs ejecute el metodo
    executeInternalProcessors. El último parámetro indica la prioridad de
    ejecución de dicha función para el caso en el que haya más de una.
408       // Con BoxRhoBarJfunctional3D se calculan los bloques rhoBar and j.
    Se calculan en el nivel 3, primero se ejecutan las funciones
    implementadas en niveles inferiores,
409             // ya que las condiciones de contorno son calculadas a nivel 1
    y 2.
410
411     /*
412      * Generación de las condiciones de contorno off-lattice (las
    condiciones alrededor del obstaculo), y las condiciones outer-domain (
    condiciones alrededor de todo el domino fluido)
413      */
414
415     pcout << "Generating boundary conditions." << std::endl;     // Imprimir
     información por pantalla, su unico proposito es informar.
416
417     OffLatticeBoundaryCondition3D<T,DESCRIPTOR,Velocity> *boundaryCondition
    ;     // Bloque que almacena información de las condiciones de contorno
    del obstculo, tambien almacena información del estado del flujo en los
    nodos pegados a este.
418
419     BoundaryProfiles3D<T,Velocity> profiles;     // Objeto que almacena las
    especificaciones que tienen que cumplir los nodos que esten en contacto
    con el cuerpo para cumplir con las condiciones de contorno que se
    impongan a este.
420     OffLatticeModel3D<T,Velocity>* offLatticeModel=0;     // Bloque auxiliar
     para acabar constuyendo boundaryCondition
421     if (param.freeSlipWall) {     // Si se desea que el cuerpo tenga la
    condicion de no deslizamiento:
```

```
422         profiles.setWallProfile(new FreeSlipProfile3D<T>);      // Le
    asociamos mediante el metodo setWallProfile dicha condicion condicion
    mediante el objeto FreeSlipProfile3D.
423     }
424     else {      // En cso contrario:
425         profiles.setWallProfile(new NoSlipProfile3D<T>);      // Establecemos
     una condicion de no deslizamiento
426     }
427     offLatticeModel =
428         new GuoOffLatticeModel3D<T,DESCRIPTOR> (
429             new TriangleFlowShape3D<T,Array<T,3> >(voxelizedDomain.
    getBoundary(), profiles),
430             flowType );     /* Para darle un valor a offLatticeModel creamos
    :
431       º1 -> Creamos un objeto de tipo TriangleFlowShape3D, donde se
    especifica el dominio voxelizado, nos interesan los nodos cercanos al
    bloque interno, donde deja de ser dominio fluido.
432       º2 -> Pasamos el objeto profiles, donde hemos especificado que
    condiciones queremos que tengan dichos nodos.
433
434       Posteriormente, pasamos este objeto creado y el tipo de flujo, en
    este caso es flujo esterior a un objeto, al constructor de la funcion
    GuoOffLatticeModel3D, que nos devuelve el objeto offLatticeModel.
435       */
436     offLatticeModel->setVelIsJ(velIsJ);      // Mediante la flag velIsJ le
    indicamos al bloque offLatticeModel creado que trabaje con valores de
    velocidad, no de velocidad*densidad.
437     boundaryCondition = new OffLatticeBoundaryCondition3D<T,DESCRIPTOR,
    Velocity>(
438             offLatticeModel, voxelizedDomain, *lattice);      // A partir del
     bloque auxilizar offLatticeModel,que contiene la información relativa
    al tipo de condicion de contorno a usar en el cuerpo, el dominio
    voxelizado y el bloque lattice creamos el objeto boundaryCondition, a
    partir de ahora utilizamos este objeto para cualquier calculo requerido
    en el que este involucrado el cuerpo de estudio.
439
440     boundaryCondition->insert();      // Metodo para inicizalizar las
    condiciones del objeto.
441
442     // Condiciones de contorno del exterior del dominio fluido.
443     OnLatticeBoundaryCondition3D<T,DESCRIPTOR>* outerBoundaryCondition
444         = createLocalBoundaryCondition3D<T,DESCRIPTOR>();      // Objeto
    creado que almacena la informacion relativa a las condiciones de
```

```
contorno exteriores.
445  outerDomainBoundaries(lattice, rhoBar, j, outerBoundaryCondition);
// funcion que asocia las condiciones creadas con los distintos bloques.
 Al contrario que con las condiciones del cuerpo, donde el objeto qe las
 carcterza impone las condiciones sobre el dominio fluido, bloque
lattice, en el caso de las condiciones exteriores este objeto creado no
se utiliza como interfaz para establecer las condiciones, en su lugar
mediante la funcion setBoundaryVelocity se impondran las condiciones
directamente sobre el bloque lattice, la utilidad de este objeto es solo
 para so interno del framework.
446
447  /*
448   * Implementación de la sponge zone.
449   */
450
451  if (param.numOutletSpongeCells > 0) {     // Solo se implementa si el nú
mero de nodos que la implementen es mayor que 0.
452      T bulkValue;     // Parametro utilizado en la dinamica de la sponge
zone, coincide con omega o con la viscosidad de Smagorinsky, segun como
se defina dicha dinamica.
453      Array<plint,6> numSpongeCells;     // Vector que indica en cada
lateral del tunel el número de nodos a implementar con Sponge Zone. Más
adelante se especifica como se utiliza en cada lateral.
454
455      if (param.outletSpongeZoneType == 0) {     // Si se desea utilizar
una dinamica viscosa en la Sponge Zone
456          pcout << "Generating an outlet viscosity sponge zone." << std::
endl;     // Imprimir información por pantalla, su unico proposito es
informar.
457          bulkValue = param.omega;     // Se utliza una dinamica similar
al caso BGK, tiene una constante igual al parametro omega.
458      } else if (param.outletSpongeZoneType == 1) {     // Si se desea
utilizar una dinamica de Smagorinsky en la Sponge Zone
459          pcout << "Generating an outlet Smagorinsky sponge zone." << std
::endl;     // Imprimir información por pantalla, su unico proposito es
informar.
460          bulkValue = param.cSmago;     // Se utliza una dinamica similar
al caso Smagorinsky, tiene una constante igual a la constante de
Smagorinsky.
461      } else {     // En caso de que no se haya especificado ninguna de
las dos, es un caso de error.
462          pcout << "Error: unknown type of sponge zone." << std::endl;
 // Se muestra un mensaje de error par informar de que se ha
```

```
especificado mal.
463             exit(-1);       // Se sale del programa.
464         }
465
466         // Número de nodos lattice sponge a lo largo de todos las fronteras
    .
467         //      El indice 0 indica el número de nodos a tomar a partir de x
    = 0
468         //      El indice 1 indica el número de nodos a tomar a partir de x
    = nx-1
469         //      El indice 2 indica el número de nodos a tomar a partir de y
    = 0
470         //      ...
471      //
472      //    En este programa solo se permite utilizar una zona Sponge en la
    parte final del tunel, si se deseasen utilziar en los demas entornos
    habría que realizar una modificación en el indice del vector
    numSpongeCells correspondiente, indicando el número de nodos en los que
    implementar la Sponge Zone.
473         numSpongeCells[0] = 0;
474         numSpongeCells[1] = param.numOutletSpongeCells;     // Solo se
    permite establecer nodos en la parte trasera del tunel.
475         numSpongeCells[2] = 0;
476         numSpongeCells[3] = 0;
477         numSpongeCells[4] = 0;
478         numSpongeCells[5] = 0;
479
480      // La dinamica en la Sponge Zone se implementa mediante
    applyProcessingFunctional, funciona de manera similar a
    integrateProcessingFunctional, pero mientras que con
    integrateProcessingFunctional se ejecutaba la funcion cada vez que se
    llamaba al metodo executeInternalProcessors(), con
    applyProcessingFunctional solo se aplica la funcion en el momento de su
    asignacion
481
482         std::vector<MultiBlock3D*> args;     // Parametros a pasar a la
    funcion applyProcessingFunctional
483         args.push_back(lattice);     // Solo es necesario el bloque lattice.
484
485         if (param.outletSpongeZoneType == 0) {     // Si se quiere
    implementar una dinamica viscosa.
486         // applyProcessingFunctional recibe como parametros:
```

```
487       // º1 -> La dinamica deseada en forma de objeto, en este caso
      ViscositySpongeZone para indicar que queremos una dinamica viscosa. A su
       ves este recibe como parametros: el tamaño del dominio fluido, el valor
       de su constante dinamica y las celdas en las que implemntar la sponge
      zone, esta ultima información va codificada segun el vector explicado
      anteriormente.
488       // º2 -> Dominio fluido sobre los que aplicar la función.
489       // º3 -> Bloques a los que implementar la funcion, pasados en forma de
       vector.
490           applyProcessingFunctional(new ViscositySpongeZone<T,DESCRIPTOR
      >(
491                       param.nx, param.ny, param.nz, bulkValue,
      numSpongeCells),
492                   lattice->getBoundingBox(), args);
493       } else {     // Si se quiere implementar una dinamica de Smagorinsky
      .
494       // Mismo caso que el anteror, solo qe esta vez se aplica una
      dinamica de tipo SmagorinskySpongeZone.
495           applyProcessingFunctional(new SmagorinskySpongeZone<T,
      DESCRIPTOR>(
496                       param.nx, param.ny, param.nz, bulkValue, param.
      targetSpongeCSmago, numSpongeCells),
497                   lattice->getBoundingBox(), args);
498       }
499    }
500
501    /*
502     * Se establecen las condiciones iniciales
503     */
504
505    // Condiciones Iniciales: Presión (y densidad) constante y velocidad
      nula en todo el dominio fluido.
506    Array<T,3> uBoundary(param.getInletVelocity(0), (T)0.0, (T)0.0);     //
      Velocidad Inicial del flujo.
507   // La funcion initializeAtEquilibrium impone en el bloque lattice unas
       condiciones iniciales de equilirbio como se especifican en los
       parametros.
508   // Parametros:
509   // º1 -> El bloque lattice.
510     // º2 -> El dominio del fluido.
511   // º3 -> La densidad del fluido.
512   // º4 -> La velocidad a imponer en el dominio del fluido.
```

```
513     initializeAtEquilibrium(*lattice, lattice->getBoundingBox(), (T)1.0,
    uBoundary);     // Es MUY importante que la densidad del fluido sea 1.0 o
     cercano a este valor, ya que por motivos de estabilidad del método es
    necesario que dicho valor permanezca cercano a 1.0. Para poder fijarla
    como 1.0 el programa ha relaizado todo el proceso de adimensionalización
     respecto a la densidad transparente al usuario y poder fijar aqui una
    densidad adimensionalizada que garantiza la mayor estabilidad posible al
     método.
514     applyProcessingFunctional(
515             new BoxRhoBarJfunctional3D<T,DESCRIPTOR>(),
516             lattice->getBoundingBox(), lattice_rho_bar_j_arg); // Se
    calcula el valor de los bloques rhoBar and j con las condiciones
    iniciales.
517
518     /*
519      * Particulas Virtuales (Lineas de corriente).
520      */
521
522     // Esta parte del codigo es la encargada de preparar los bloques que
    trabajan con las paticulas virtuales.
523     // Las particulas virtuales se utilizan únicamente para el calculo de
    lineas de corriente.
524
525     // Definición del campo de particle virtuales.
526     MultiParticleField3D<ParticleFieldT>* particles = 0;     // Campo de
    particulas virtuales. Se inicializa nulo, si se ha especificado usar
    particular virtuales se creara un dominio util.
527
528     if (param.useParticles) {     // Si se ha especificado usar particular
    virtuales.
529         particles = new MultiParticleField3D<ParticleFieldT>(     // Campo
    de particulas virtuales, para vincularlo con el dominio del bloque
    lattice utilizamos su metodo getMultiBlockManagement
530             lattice->getMultiBlockManagement(),
531             defaultMultiBlockPolicy3D().getCombinedStatistics() );
532
533         std::vector<MultiBlock3D*> particleArg;     // Vector qe almacenara
    el vector de particulas para integrarle una función mediante
    integrateProcessingFunctional.
534         particleArg.push_back(particles);     // Se añade el bloque de
    particulas virtuales al vector.
535
```

```
536     std::vector<MultiBlock3D*> particleFluidArg;     // Segundo vector
    que almacena bloques para asociarles funciones. Este vector almacena
    tanto el campo de particulas virtuales como al bloque lattice.
537     particleFluidArg.push_back(particles);
538     particleFluidArg.push_back(lattice);
539
540     // Funcion que posiciona las particulas en su nueva posición en
    cada iteracion
541     integrateProcessingFunctional (
542             new AdvanceParticlesEveryWhereFunctional3D<T,DESCRIPTOR>(
    param.cutOffSpeedSqr),
543             lattice->getBoundingBox(), particleArg, 0);
544     // Función que asigna a las particulas su velocidad, de acuerdo al
    estado del bloque lattice del fluido.
545     integrateProcessingFunctional (
546             new FluidToParticleCoupling3D<T,DESCRIPTOR>((T) param.
    particleTimeFactor),
547             lattice->getBoundingBox(), particleFluidArg, 1 );
548
549     // Dominio sobre el cual se inyectaran nuevas particlas virtuales
    de forma aleatoria. Este dominio es un paralelogramo en el plano
    perpendicular a X=0, centrado en los ejes Y y Z del dominio y de
    longitud en cada uno de estos ejes la mitad del tamaño del dominio
    fluido en ellos.
550     Box3D injectionDomain(0, 0, centerLB[1]-0.25*param.ny, centerLB
    [1]+0.25*param.ny,
551             centerLB[2]-0.25*param.nz, centerLB[2]+0.25*param.nz);
552
553     // Campo de particulas sin masa, utilizado para inyectar particulas
554     Particle3D<T,DESCRIPTOR>* particleTemplate=0;
555     particleTemplate = new PointParticle3D<T,DESCRIPTOR>(0, Array<T
    ,3>(0.,0.,0.), Array<T,3>(0.,0.,0.));
556
557     // Vector con los parametros en los que inyectar particulas
    virtuales nuevas, logicamente es el campo de particulas virtuales.
558     std::vector<MultiBlock3D*> particleInjectionArg;
559     particleInjectionArg.push_back(particles);
560
561     integrateProcessingFunctional (     // Asocimos al bloque de
    particulas virtuales la funcion que inyecta nuevas particulas virtuales
    en el vector definido justo anteriormente, del bloque particleTemplate
    de particulas sin masa, con la probabilidad indicada en
    particleProbabilityPerCell, en el dominio injectionDomain,
```

```
562             new InjectRandomParticlesFunctional3D<T,DESCRIPTOR>(
    particleTemplate, param.particleProbabilityPerCell),
563             injectionDomain, particleInjectionArg, 0 );
564
565     // La salida del tunel se define como un bloque, posteriormente se
    asociara a este bloque una funcion que absorbera las particulas
    virtuales en esta zona.
566     Box3D absorbtionDomain(param.outlet);
567
568     // Asociamos al bloque de particulas virtuales, que se encuentra
    dentro del vector particleArg, una funcion que absorbera las particulas
    virtuales que se encuentren en la salida del tunel.
569     integrateProcessingFunctional (
570             new AbsorbParticlesFunctional3D<T,DESCRIPTOR>,
    absorbtionDomain, particleArg, 0 );
571
572     particles->executeInternalProcessors();     // Ejecutamos las
    funciones asociadas con integrateProcessingFunctional para inicializar
    el bloque de particulas virtuales.
573 }
574
575 /*
576  * Comienzo de la simulación.
577  */
578
579 plb_ofstream ForcesFile((outputDir + "ForcesFile_" + util::to_string(jj
    ) + ".dat").c_str());     // Creción de fichero qe contiene las fuerzas
    sobre el sólido, se trata como un fichero de texto normal sobre el cual
    se iran imprimiendo los datos siguiendo un formato CSV. El nombre del
    fichero sigue una codificación especial para identificar cada uno de
    ellos cuando se realizan varias simulaciones siguiendo una rotación del
    cuerpo de estudio. Todos comienzan por la cadena ForcesFile_ siguiendo
    el número de giros que acumula en sólido en la simulación, y por ultimo
    se añade una extenson .dat
580
581 pcout << std::endl;     // Imprimir salto de linea extra por pantalla,
    su unico proposito es organizar mejor la informción mostrada.
582 pcout << "Starting simulation." << std::endl;     // Imprimir informació
    n por pantalla, su unico proposito es informar de que empieza a simulaci
    ón.
583 for (plint i = 0; i < param.maxIter; ++i) {     // Bucle de la simulació
    n. Cada iteración de este bucle es una iteración del método Lattice-
    Boltzmann.
```

```
584         if (i <= param.initialIter) {     // Durante las primeras
    iteraciones se produce una aceleración del flujo en la entrada.
585             Array<T,3> uBoundary(param.getInletVelocity(i), 0.0, 0.0);
    // La funcion getInletVelocity se encara de dar obtener un valor que
    incrementa gradualmente hasta alcanzar el valor final. En las
    iteraciones iniciales la velocidad en el eje X va acelerandose siguendo
    esta progresión.
586             setBoundaryVelocity(*lattice, param.inlet, uBoundary);     //
    Funcion que asigna al bloque lattice (primer parametro), en su cara de
    entrada (segundo parametro), la velocidad indicada en el tercer
    parametro.
587         }
588
589         if (i %param.statIter == 0) {     // Cada número de iteraciones
    indicadas en la variable statIter volcamos los datos por pantalla y al
    fichero que almacena las fuerzas sobre el cuerpo.
590             pcout << "At iteration " << i << ", t = " << i*param.dt << std
    ::endl;      // Se imprime por pantalla el tiempo de simulación que se
    lleva realizado.
591             Array<T,3> force(boundaryCondition->getForceOnObject());     //
     Mediante el método getForceOnObject() del objeto boundaryCondition, que
     contiene la informacion referente a las condiciones en el contorno del
    sólido, se obtiene la fuerza resultante sobre el objeto.
592             T factor = (util::sqr(util::sqr(param.dx)) / util::sqr(param.
    dt))*param.densidad;     // Factor utilizado para pasar las unidades de
    la fuerza del sistema adimensonal usado en el método Lattice Botzmann,
    al sistema de dimensiones que ha introducido el usuario, mediante la
    multiplicación de los valores de la fuerza adimensional por dicho factor
    .
593             pcout << "Force on object over fluid density: F[x] = " <<
    force[0]*factor << ", F[y] = "     // Escribimos por pantalla el
    resultado de las fuerzas sobre el cuerpo. Su unico proposito es
    proporcionar información durante la simulación.
594                   << force[1]*factor << ", F[z] = " << force[2]*factor <<
    std::endl;
595             T avEnergy = boundaryCondition->computeAverageEnergy() * util
    ::sqr(param.dx) / util::sqr(param.dt);     // Creamos una variable que
    almacena el promedio de la energia cinetica del fluido que rodea al
    cuerpo entre la densidad del fluido. Su unico proposito es imprimirse
    por pantalla como información adicional.
596             ForcesFile << i*param.dt << ";" << force[0]*factor << ";" <<
    force[1]*factor << ";" << force[2]*factor << std::endl;     // Volcado de
     las fueras en el fichero ForcesFile. Se sigue un formato de CSV usando
```

```
como separador ";". Se almacenan por orden los valores del vector de
fuerza, primero el valor en el eje X, almacenado en el elemento [0] del
array force, despues el valor en la dirección del eje Y, almacenado en
el elemento [1] y por último el valor en el eje Z, almacenado en [2].
Para escalarlo con el sistema de unidades introducido por el usuario, lo
 multiplicamos por el factor de escla almcenado en la variable factor.
597             pcout << "Average kinetic energy over fluid density: E = " <<
avEnergy << std::endl;     // Imprimir datos por pantalla, su unico
proposito es informar.
598             pcout << std::endl;     // Imprimir salto de linea extra por
pantalla, su unico proposito es organizar mejor la informción mostrada.
599         }
600
601         if (param.vtkFile) {     // Si se desea volcar el estado de la
simulación en VTK.
602           if (i %param.vtkIter == 0) {     // Cada número de iteraciones
indicadas en vtkIter realizamos el volcado en formato VTK.
603         pcout << "Writing VTK at time t = " << i*param.dt << endl;     //
Imprimir datos por pantalla, su unico proposito es informar.
604         writeVTK(*boundaryCondition, i, jj);     // Llamamos a la funcion
que se encarga de volcar los datos.
605         if (param.useParticles) {     // Si se desea utilizar particulas
virtuales con proposito de visualizacion se escriben en un fichero con
extension vtk
606           writeParticleVtk<T,DESCRIPTOR> (     // Funcion que vuelca el
estado de las particulas, el primer parametro es el bloque que las
almacena, el segundo el fichero sobre el cual se vuelcan y el tercero el
 número máximo de particulas a escribir.
607             *particles, createFileName(outputDir+"particles_iter_"+util::
to_string(jj)+ "_", i, PADDING) + ".vtk",     // La codificación en el
nombre del fichero que utilizamos es la misma que la utilizada en los
ficheros vti, explicada en la función writeVTK.
608             param.maxNumParticlesToWrite );
609         }
610       }
611     }
612
613         lattice->executeInternalProcessors();     // Ejecución del metodo
executeInternalProcessors del bloque lattice, con esto ejecutamos todas
las funciones que le hemos asignado al bloque mediante la funcion
integrateProcessingFunctional.
614         lattice->incrementTime();     // Incrementamos el tiempo de la
simulacion en un valor dt.
```

```
615         if (param.useParticles && i % param.particleTimeFactor == 0) {
    // Cada número de iteraciones indicadas en la variable
    particleTimeFactor calculamos la nueva distribucion de  particulas del
    bloque particles.
616             particles->executeInternalProcessors();     // Ejecución del
    metodo executeInternalProcessors ddel bloque lattice, con esto
    ejecutamos todas las funciones que le hemos asignado al bloque mediante
    la funcion integrateProcessingFunctional.
617         }
618     }
619
620     ForcesFile.close();     // Cerramos el fichero en el que hemos volcado
    el valor de las fuerzas.
621     delete outerBoundaryCondition;     // Borramos de forma segura el bloque
     outerBoundaryCondition de la memoria.
622     delete boundaryCondition;     // Borramos de forma segura el bloque
    boundaryCondition de la memoria.
623     if (param.useParticles) {
624         delete particles;     // Borramos de forma segura el bloque
    particles de la memoria. Lo introducimos dentro de un if para verificar
    que se ha creado con anterioridad.
625     }
626     delete j;     // Borramos de forma segura el bloque j de la memoria.
627     delete rhoBar;     // Borramos de forma segura el bloque rhoBar de la
    memoria.
628     delete lattice;     // Borramos de forma segura el bloque lattice de la
    memoria.
629 }
630
631 int main(int argc, char* argv[])
632 {
633     plbInit(&argc, &argv);     // Funcion que se encarga de pasar los
    parametros a todos los procesos MPI que ejecutan el programa. Tambien
634     global::directories().setOutputDir(outputDir);     // Se establece el
    lugar donde se volcaran las salidas.
635     pid_t pid;     // pid del proceso una vez se llame a fork.
636     int i;     // Variable que almacenara la respuesta del proceso hijo.
637
638     // Se usan bloques try-catch por si ocurren errores. En caso de que
    ocurran son capturados por el bloque catch
639     // y termina el programa de manera correcta mostrando el mensaje de
    error adecuado.
640
```

```
641     // 1. Leer los parametros de la linea de comandos: el fichero XML y
    opcionalmente la contraseña del correo
642     string xmlFileName;     // Nombre del fichero XML que almacena los datos
     de configuración.
643     string password;    // Almacena la contraseña del correo, no se pasa por
     el XML por motivos de seguridad
644     try {
645         global::argv(1).read(xmlFileName);     // Primera opcion de la linea
     de comandos el nombre del fichero XML
646         if (global::argc()==3) {     // Si se han pasado 2 opciones al
    comando
647             global::argv(2).read(password);     // Leer la contraseña del
    correo
648         }
649     }
650     catch (PlbIOException& exception) {
651         pcout << "Wrong parameters; the syntax is: "
652               << (std::string)global::argv(0) << " input-file.xml [email
    password]" << std::endl;     // Se indica la manera correcta de utilizaci
    ón
653         return -1;
654     }
655
656     // 2. Leer el fichero XML
657     try {
658         param = Param(xmlFileName);     // Ejecucion de la funcion Param con
     el nombre del fichero XML como parametro.
659     }
660     catch (PlbIOException& exception) {
661         pcout << exception.what() << std::endl;
662         return -1;
663     }
664
665   // 3. Ejecucion de la simulación
666
667     // 3.1. Se establecen los giros iniciales del sólido en cada iteracion
668   for (int j = 0; j < param.iteracionesPrograma; ++j) {
669
670     if (j != 0) {
671       param.psi = param.psi + param.deltapsi;
672                         param.theta = param.theta + param.deltatheta;
673                         param.phi = param.phi + param.deltaphi;
674     }
```

```
675
676     // 3.2. Se ejecuta el programa principal
677       try {
678           runProgram(j); // Se pasa como parametro la variable del bucle
    for para diferenciar los ficheros de volcado de datos en cada iteración.
679       }
680       catch (PlbIOException& exception) {
681           pcout << exception.what() << std::endl;
682           return -1;
683       }
684   }
685
686     // 4. Llamarda al sistema de avisos
687
688         global::mpi().barrier();     // Espera a que todos los procesos
    lleguen a este punto
689
690         if (global::mpi().getRank() == 0) {      // Si el proceso tiene
    rango 0
691             if ((pid = fork()) == 0) {     // Se crea un nuevo subproceso
    con fork, en caso de que sea el hijo
692       //TODO incluir el caso del error
693                 if (global::argc()==3) {        // Si se ha llamado con 3
    parametros significa que se le ha pasado la contraseña del email
694                     execl( "/usr/bin/python", "python", "bot.py", password.
    c_str(), (char*)0 ); //El subproceso hijo se transforma en un nuevo
    proceso que ejecuta el sistema de avisos
695                 } else if (global::argc()==2) {       // Si se ha ejecutado
    con 2 parametros significa que no se desea utilizar el sistema de avisos
     por email.
696                     execl( "/usr/bin/python", "python", "bot.py", (char*)0
    );  //El subproceso hijo se transforma en un nuevo proceso que ejecuta
    el sistema de avisos
697                 }
698             }
699             else {
700                 pid = wait(&i);     // El proceso padre espera a que el hijo
     acabe su ejecución.
701             }
702         }
703         global::mpi().barrier();      // Espera a que todos los procesos
    lleguen a este punto, se espera a que el proceso encargado de llamar al
    sistema de avisos llegue para finalizarse todos al mismo tiempo.
```

```
}
<!-- Todos los valores estan en unidades fisicas, no en unidades lattice. -->
<!-- El flujo curre a lo largo del eje X. -->
<configuracion>
<geometry>
    <!-- Nombre del archivo STL que contiene la representación del obstaculo. -->
    <filename>esfera_0.2.stl</filename>
    <!-- Posición del centro del obstaculo en el dominio fluido. -->
    <center> <x>0.45</x> <y>0.455</y> <z>0.455</z> </center>
    <!-- Si es False, la superficie del obstaculo no tiene deslizamiento, en cualquier otro caso la superficie tendra deslizamiento. -->
    <freeSlipWall>False</freeSlipWall>
    <!-- Si es False, los laterales del tunel tendran unas condiciones de contorno libres en las paredes laterales y superior, y condicion de no deslizamiento en la inferior. En otro caso serán todas libres. -->
    <lateralFreeSlip>True</lateralFreeSlip>
    <!-- Tamaño del dominio fluido. -->
    <domain> <x>2.95</x> <y>0.9</y> <z>0.9</z> </domain>
  <!-- Giros iniciales a realizar sobre el sólido. -->
    <Giros>
  <phi> 0.0 </phi> <!-- Giros en eje X. -->
  <theta> 0.0 </theta> <!-- Giros en eje Y. -->
  <psi> 0.0 </psi> <!-- Giros en eje Z. -->
  </Giros>
</geometry>
<numerics> <!-- Todo en unidades fisicas, o adimensionales si se necesita un sistema intermedio, nada en unidades lattice. -->
    <!-- Densidad -->
    <ro> 1.00</ro>
    <!-- Viscosidad Cinemática. -->
    <nu>0.02109</nu>
    <!-- Velocidad de entrada en la dirección X. Velocidad real. -->
    <inletVelocity>2.0</inletVelocity>
    <!-- Número de iteraciones a ejecutar, incrementando en cada una de ellas el valor en los angulos de giro indicados posteriormente. -->
    <iter> 1 </iter>
  <!-- Giros acumulativos a realizar sobre el sólido en cada iteración. -->
    <deltaphi> 2.0 </deltaphi> <!-- Giros en eje X. -->
    <deltatheta> 2.0 </deltatheta> <!-- Giros en eje Y. -->
    <deltapsi> 2.0 </deltapsi> <!-- Giros en eje Z. -->
    <!-- Resolución, indicada en número de particiones a realizar sobre el
```

```
    dominio fluido en el eje Y. -->
36   <resolution>85</resolution>
37  <!-- Valor de la unidad básica de tiempo, incremento de tiempo discreto
    en cada iteración. -->
38   <dt> 0.0001</dt>
39   <!-- Modelo a utilizar, disponibles BGK -> modelo de
    -BhatnagarGrossKrook, Smagorinsky -> modelo de Smagorinsky, Entropic ->
     Modelo entrópico y Regularized -> Modelo BGK regularizado. -->
40   <modelo>BGK</modelo>
41   <!-- Parametro para el modelo Large Eddy Simulation de Smagorinsky. -->
42   <cSmago> 0.14 </cSmago>
43   <!-- usar particulas virtuales o no. En el caso de usarlas tener en
    cuenta que solo tenen un proposito visual, utilizadas para calcular las
    lineas de corriente. -->
44   <useParticles>False</useParticles>
45   <!-- Frecuencia de actualización del campo de particulas virtuales. Un
    valor de 2 actualiza el campo cada dos iteraciones del método. -->
46   <particleTimeFactor> 2 </particleTimeFactor>
47   <!-- Probabilidad de inyección de una particula virtual nueva en cada
    celda para cada iteración. -->
48   <particleProbabilityPerCell> 1.0e-3 </particleProbabilityPerCell>
49   <!-- Criterio para eliminar particulas de muy baja velocidad.
50        Cuando el cuadrado de la velociadad sea inferior a este valor, la
    particula sera eliminada de la simulación. -->
51   <cutOffSpeedSqr> 1.e-8 </cutOffSpeedSqr>
52   <!-- Número máximo de particulas a escribir en el fichero VTK. -->
53   <maxNumParticlesToWrite> 200000 </maxNumParticlesToWrite>
54   <!-- Espesor de la Sponge Zone en la salida. Un valor de 0.0 indica que
     no se quiere utlizar Sponge Zone. -->
55   <outletSpongeZoneWidth>0</outletSpongeZoneWidth>
56   <!-- Tipo de modelo a usar en la Sponge Zone ("Viscosity" o "
    Smagorinsky"). -->
57   <outletSpongeZoneType>Viscosity</outletSpongeZoneType>
58   <!-- Parametro para el modelo de Smagorinsky en la Sponge Zone en caso
    de que se vaya a utilizar.
59        Este parametro solo es relevante si se ha elegido una Sponge Zone
    de tipo Smagorinsky. -->
60   <targetSpongeCSmago> 0.6 </targetSpongeCSmago>
61   <!-- Tiempo hasta el cual la velocidad de entrada se va incrementando
    gradualmente hasta su valor final. -->
62   <initialIterT>1.0</initialIterT>
63   <!-- Maximum simulation time. -->
64   <maxT>7.5</maxT>
```

```
</numerics>
<output>
    <!-- Frecuencia de volcado de las fuerzas en pantalla y en el archivo de fuerzas. -->
    <statT> 0.01</statT>
    <!-- Salida en archivo VTK, True/False -->
    <vtkFile>False</vtkFile>
    <!-- Frecuencia con la cual los ficheros VTK son generados. -->
    <vtkT> 0.2 </vtkT>
</output>

<!-- Parametros a pasarle al lanzador MPI. -->
<MPI>-np 4 --prefix /home/user/local</MPI>
<!-- Sistema de aviso por Email. -->
<email>
    <enviar>False</enviar> <!-- Utilizar aviso por email. True/False. -->
    <emailFuente>608525@unizar.es</emailFuente> <!-- Direccion desde la que enviar el email. -->
    <emailDestino>608525@unizar.es</emailDestino> <!-- Direccion en la que recibir el email. -->
    <msg>Simulación terminada</msg> <!-- Texto a poner en el email. -->
</email>
<!-- Sistema de aviso por Telegram. -->
<telegram>
    <enviar>True</enviar> <!-- Utilizar aviso por telegram. True/False. -->
    <TOKEN>123456789:ABCDEFGHIJKLMNOPQRSTUVWXYZ123456789</TOKEN> <!-- TOKEN Bot para poder acceder a él. -->
    <chat_id>1234567</chat_id> <!-- ID que identifica a cual de las conversaciones que tiene activas el Bot hay que enviar el mensaje. -->
    <msg>Simulación terminada</msg> <!-- Texto a poner en el mensaje. -->
</telegram>
</configuracion>
```

B.3 Sistema de avisos

En esta sección se muestra el código encargado de lanzar los avisos por email y Telegram.

```
'''
This file is part of the U-Virtual Wind Tunnel.

Copyright (C) 2017

```

```
The U-Virtual Wind Tunnel is free software: you can redistribute it and/or
modify it under the terms of the GNU Affero General Public License as
published by the Free Software Foundation, either version 3 of the
License, or (at your option) any later version.

The U-Virtual Wind Tunnel is distributed in the hope that it will be useful
,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU Affero General Public License for more details.

You should have received a copy of the GNU Affero General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
'''

import sys
from lxml import etree
from io import StringIO
import telebot
import smtplib

f = open("externalFlowAroundObstacle.xml") # Abrir el archivo de
    configuración.
xml = f.read() # Lectura de archivo de configuración.
f.close() # Una vez leido, se cierra el fichero.

tree = etree.parse(StringIO(xml)) # Se parsea el archivo de configuración,
    es un fichero XML. Se crea el objeto etree que contiene los datos del
    documento XML. Mediante la funcion StringIO ajustamos el tipo de formato
     que necesita el método parse como entrada, ya que el tipo de dato en
    que se encontraba la variable xml no era el indicado.

email = tree.xpath('/configuracion/email/enviar')[0].text # Mediante XPath,
     un lenguaje de consulta sobre XML, se busca el nodo que indica si hay
    que enviar un email o no. El métod XPath devuelve una tupla, por si hay
    mas de una coincidencia, como sólo hay un nodo que coincida con la
    busqueda retomamos el elemento número 0. Por último se llama al atributo
     text, que contiene el texto de la etiqueta buscada.
telegram =  tree.xpath('/configuracion/telegram/enviar')[0].text #
    Realizamos la misma tarea que el caso anterior, pero esta vez para saber
     si hay que enviar un mensaje por Telegram.

```

```
35 if len(sys.argv)==2: # Se verifica que se ha pasado un parámetro. En caso
     de que lo tenga, es la contraseña del correo.
36   password=sys.argv[1] # Primer argumento, contraseña del correo.
37
38 if email=='True': # En caso de que se desee enviar una notificación por
     email
39   emailFuente = tree.xpath('/configuracion/email/emailFuente')[0].text #
      Lectura de la dirección de origen del correo.
40   emailDestino = tree.xpath('/configuracion/email/emailDestino')[0].text #
      Lectura de la dirección de destino.
41   msg = tree.xpath('/configuracion/email/msg')[0].text # Lectura del
      mensaje a enviar.
42   posicionArroba = emailFuente.find('@',0,len(emailFuente)) # Se usa el mé
     todo find para buscar la arroba. Primer parámetro letra a buscar,
     segundo parámetro posición en la que se empieza a buscar, tercer pará
     metro posición en la que se acaba de buscar. Se utiliza la funcion len
     que devuelve la longitud de la cadena, en este caso de la dirección
     email para buscar en toda ella.
43   dominioEmail = emailFuente[posicionArroba+1:] # Desde la posición de la
      arroba hasta el final de la cadena se guarda el texto, el cual
     corresponde con el dominio del correo.
44   if dominioEmail == 'unizar.es': # En caso de que sea una cuenta unizar
45     server = smtplib.SMTP('smtp.unizar.es', 587) # Nos conectamos al
     servidor SMTP de unizar desde el qe enviar el correo. Creamos el objeto
     server desde el cual se gestiona la conexión con el servidor.
46 #    server.ehlo() # Se envía un mensaje de saludo, necesario por el
     protocolo.
47     server.starttls() # Se utiliza en protocolo TLS para proteger la conexi
     ón.
48     server.ehlo() # Se envía un mensaje de saludo, necesario por el
     protocolo.
49     server.login(emailFuente, password) # Nos identificamos con el email y
     la contraseña indicadas por el usuario.
50     server.sendmail(emailFuente, emailDestino, msg) # Se envia el correo
     mediante el método sendmail. Primer parámetro el email de origen,
     segundo el email de destino  y tercero el mensaje a enviar.
51     server.quit() # Desconexión del servidor.
52   else: # en caso de que no sea unizar.es se muestra un mensaje de error.
53     print('Se necesita una cuenta unizar para que funcione') # Mensaje de
     error
54
55 if telegram=='True': # En caso de que se desee enviar una notificación por
     Telegram
```

```
56 TOKEN = tree.xpath('/configuracion/telegram/TOKEN')[0].text # Se obtiene
     el Token del bot al que conectarse.
57 chat_id = int(tree.xpath('/configuracion/telegram/chat_id')[0].text) # Se
     obtiene el id del chat al que se desea enviar el mensaje.
58 mi_bot = telebot.TeleBot(TOKEN) # Se inicia la sesion con el bot,
     pasandole como parámetro el Token. Se crea el objeto mi_bot con el que
     actuar sobre el bot.
59 mi_bot.send_message(chat_id,tree.xpath('/configuracion/telegram/msg')[0].
     text) # Mediante el método send_message del objeto mi_bot se envía un
     mensaje al char indicado. Primer parámetro id de la conversación a la
     que enviar el mensaje, segundo parámetro el mensaje a enviar. En una
     sola linea se lee el mensaje a enviar del objeto tree y se envía.
```

B.4 Programa de análisis de datos

Se incluye como ejemplo el programa utilizado para analizar el error en el coeficiente de arrastre frente al parámetro δ_t usado en la sección 3.1.3.

Para adaptarlo al resto de casos, sólo hay que cambiar la búsqueda *XPath* de la variable *resolucionString* por el parámetro del cual se esta estudiando su influencia.

```
1 #!/usr/bin/env python
2 import pandas as pd
3 import numpy as np
4 import matplotlib.pyplot as plt
5 from lxml import etree
6 from io import StringIO
7 import glob
8 import os
9 from scipy import interpolate
10 from math import *
11 '''Unico valor que no se lee del archivo XML'''
12 D=np.float64(0.2) # Diámetro de la esfera.
13 '''Nombres ficheros utiles'''
14 nombreArchivoFueras = 'ForcesFile_0.csv' # Fichero que contiene las fuerzas
15 nombreArchivoConfiguracion = 'externalFlowAroundObstacle.xml' # Fichero de
     configuración
16 nombreArchivoFuerzasReferencia = 'Referencia.csv' # Fichero con los
     coeficientes de arrastre de referencia
17 '''Variables usadas'''
18 area=np.float64(np.pi*(D/2)*(D/2)) #Área frontal de la esfera
```

```
numerosReynolds=[] # Número de Reynolds de cada simulación, se van almacenando en forma de tupla
coeficienteArrastre=[] # Coeficientes de arrastre de cada simulación, se van almacenando en forma de tupla
resolutiones=[] # Conjunto de valores dt, se van almacenando en forma de tupla
for carpeta in glob.glob('tmp*'): # Para cada carpeta que empiece por tmp-. Cada uno de estos ficheros contiene el resultado de una de las simulaciones realizadas.
  '''Ir a la carpeta, leer los ficheros necesarios y volver'''
  os.chdir(carpeta) # Ir a esa carpeta
  f = open(nombreArchivoConfiguracion) # Abrir el fichero de configuración
  xml = f.read() # Leer el fichero de configuración y se guarda su contenido en la variable xml
  f.close() # Cerrar el fichero de configuración.
  tree = etree.parse(StringIO(xml)) # Parsear el contenido del archivo de configuración con el parser XML
  velocidadString = tree.xpath('//inletVelocity')[0].text # Buscar la entrada que indica la velocidad del fluido
  velocidad = np.float64(velocidadString) # Se para la velocidad del fluido a una variable numérica
  visc_dString = tree.xpath('//nu')[0].text # Buscar la entrada que indica la viscosidad del fluido
  visc_d=np.float64(visc_dString) # Se para la viscosidad del fluido a una variable numérica
  rho_dString = tree.xpath('//ro')[0].text # Buscar la entrada que indica la densidad del fluido
  rho_d=np.float64(rho_dString) # Se para la viscosidad del densidad a una variable numérica
  resolucionString=tree.xpath('//dt')[0].text # Se busca la entrada del parámetro dt
  resolucion=np.float64(resolucionString) # Se pasa el parámetro dt a una variable numérica
  file = pd.read_csv(nombreArchivoFueras,sep=';',names=['time','fx','fy','fz']) # Procesado del fichero que tiene las fuerzas. Es un fichero CSV con cuatro columnas, instante de tiempo y fuerza en cada uno de los tres ejes, separados por el caracter ';'.
  os.chdir('..') # Salir de la carpeta
  '''Calculo de la fuerza final'''
  fxMean=(file[file.time > 5.0]['fx'].mean()) # Para calcular la fuerza resultante, se calcula la média a partir de un valor lo suficientemente alto como para que se haya estabilicado el flujo.
  '''Agregar los valores que se quieren almacenar en las listas'''
```

```
42    coeficienteArrastre.append(2*fxMean/(rho_d*(velocidad**2)*area)) # Se
      calcula el coeficiente de arrastre para la simulación dada y se almacena
       en la tupla.
43    numerosReynolds.append(velocidad*D/visc_d) # Se calcula el número de
      Reynolds para la simulación dada y se almacena en la tupla.
44    resolutiones.append(resolucion) # Se van almacenando los valores de dt
      probados en una tupla.
45  resolutiones=np.array(resolutiones) # Se cambia el conjunto de valores de
      dt probados a un array.
46  '''Crear los fucheros Cd_dt.dat y Force_vs_dt.dat'''
47  coeficienteArrastreResolucion=pd.Series(coeficienteArrastre, index=
      resolutiones)  # Se crea una serie con el indice el valor de dt y el
      coeficiente de arrastre asociado
48  coeficienteArrastreResolucion.to_csv('Cd_dt.csv',sep=';') # Se vuelca la
      serie anterior a un fichero csv, utilizando como separador en parámetro
      ';'.
49  '''Calculo del error'''
50  csvReferenciaFuerza=pd.read_csv(nombreArchivoFuerzasReferencia,sep=';',
      names=['Re','Cd']) # Se toman los valores de Cd frente a Re de
      referencia extraidos.
51  interpolador = interpolate.interp1d(csvReferenciaFuerza['Re'],
      csvReferenciaFuerza['Cd'], kind='quadratic', bounds_error=True) # Se
      crea un interpolador cuadrático que interpola entre los valores cargados
       anteriormente.
52  error = (interpolador(numerosReynolds)-coeficienteArrastre)/interpolador(
      numerosReynolds) # Se calcula el error relativo. Para ello se resta el
      valor teórico, obtenido interpolando el número de Reynolds de la
      simulación con el interpolador creado anteriormente, con el coeficiente
      de arrastre obtenido, y dividiendo por el valor teórico.
53  '''Dibujar Error / dt'''
54  plt.figure('Error frente a dt') # Nombre de la figura, solo tiene interes a
       nivel interno del programa
55  plt.scatter(resolutiones*1000,error*100) # Se indican las coordenadas de
      los puntos en el eje X y en el eje Y
56  plt.title(u'Error frente a  $\delta_t$') # Titulo de la figura
57  plt.ylim(min(error*100)*1.1,max(error*100)*0.9) # Se establecen los limites
       de los valores a mostrar en el eje Y
58  plt.xlim(min(resolutiones*1000)*0.9,max(resolutiones*1000)*1.1) # Se
      establecen los limites de los valores a mostrar en el eje X
59  plt.grid(True) # Se mostrara una malla de fondo
60  plt.minorticks_on() # Se muestran más marcas en los ejes
61  plt.xlabel(u'$\delta_t$ ·(1000)') # Nombre del eje X
62  plt.ylabel(u'Error (%)') # Nombre del eje Y
```

```
63 plt.savefig('Error_vs_dt.png', dpi=400, bbox_inches='tight') # Se guarda la
       gráfica en un fichero png
64 plt.close # Se cierra la figura
65 '''Crear fichero Error / dt'''
66 df=pd.DataFrame(data=error, index=resolutiones, columns=['error (t/1)']) #
     Se crea un dataframe que contiene el error obtenido para cada dt.
67 df.to_csv('Error_vs_dt.csv', sep=';') # Se vuelca el dataframe anterior a
     un fichero csv, utilizando como separador en parámetro ';'.
```