UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

Acoplamiento del método de teoría de elemento de pala con volúmenes finitos en OpenFOAM

Trabajo Fin de Máster

Máster Universitario en Ingeniería Aeronáutica

AUTOR/A: Torres Pons, Alfredo Francisco

Tutor/a: García-Cuevas González, Luis Miguel

CURSO ACADÉMICO: 2022/2023

UNIVERSITAT POLITÈCNICA DE VALÈNCIA
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DEL DISEÑO

TRABAJO FINAL DE MASTER

Acoplamiento del método de teoría de elemento de pala con volúmenes finitos en OpenFOAM

REALIZADO POR:
Alfredo Torres Pons - altorpon@etsid.upv.es

TUTOR:
Luis Miguel García-Cuevas González - luiga12@upv.es

COTUTOR:
Nicolás Medina Tomás - nimeto@upv.es

MÁSTER EN
Ingeniería Aeronáutica

Valencia, julio de 2023

Abstract

Computational fluid dynamics (or CFD) programs make it possible to obtain detailed solutions to complex fluid dynamics problems. As a general rule, the more complex the problem, the higher the computational cost required. Fortunately, there are multiple methods to reduce the associated computational cost without incurring prohibitive penalties in the quality of the results. One such method is the resolution of rotors and propellers using the blade element theory (BET) method, coupled with finite volume resolution of the rest of the fluid field. Thus, it is possible to solve problems with a strong interaction between the propulsive system and the aerodynamic performances of an aircraft at a contained cost.

OpenFOAM, the most widely used open source CFD library, currently has a BET method implemented, although highly limited. In this work, the author implements the BET method in OpenFOAM providing it with many useful extra functionalities: interpolation in Reynolds number, Mach number and angle of attack, extrapolation in angle of attack, multiple meshing methods of the BET disk or resolution, among others. In addition, a modular implementation is developed, laying a foundation for future development and improvement of this method. The implementation is verified against the Star-CCM+ BET method obtaining matches in the predicted values of loads to the fourth decimal place.

The BET implementation is performed within a larger designed tool, named as *PropellerSource*. It abstracts the most important elements of the actuating disk-based methods such as meshing, control, source term calculation, interpolation or finite volume cell selection. This allows to facilitate the addition of new models or to improve the existing ones.

Resumen

Los programas de dinámica de fluidos computacional (o CFD, por sus siglas en inglés) permiten obtener soluciones detalladas en problemas complejos de dinámica de fluidos. Como regla general, cuanto más complejo es el problema, mayor es el coste computacional requerido. Afortunadamente, existen múltiples métodos para reducir el coste computacional asociado, sin incurrir en prohibitivas penalizaciones en la calidad de los resultados. Uno de estos métodos es la resolución de rotores y hélices mediante el método de la teoría de elemento de pala (BET, por sus siglas en inglés), acoplada con la resolución mediante volúmenes finitos del resto del campo fluido. Así, es posible resolver problemas con una fuerte interacción entre el sistema propulsivo y las actuaciones aerodinámicas de una aeronave con un coste contenido.

OpenFOAM, la librería de código abierto de CFD más utilizada, tiene en la actualidad un método BET implementado, aunque altamente limitado. En este trabajo, el autor implementa el método BET en OpenFOAM dotándole de multitud de funcionalidades extra de gran utilidad: interpolación en número de Reynolds, número de Mach y ángulo de ataque, extrapolación en ángulo de ataque, múltiples métodos de mallado del disco del BET o resolución, entre otros. Además, se desarrolla una implementación modular, sentando una base para el desarrollo y mejora futura de este método. La implementación es verificada frente el método BET de Star-CCM+ obteniendo concordancias en la predicción de las cargas hasta el cuarto decimal.

La implementación del BET se realiza dentro de una herramienta más grande diseñada, nombrada como *PropellerSource*. En ella se abstraen los elementos más importantes de los métodos basados en disco actuador como el mallado, el control, el cálculo de los términos fuente, la interpolación o la selección de celdas de volúmenes finitos. Esto permite facilitar la adición de modelos nuevos o mejorar los existentes.

Resum

Els programes de dinàmica de fluids computacional (o CFD, per les seues sigles en anglés) permeten obtindre solucions detallades en problemes complexos de dinàmica de fluids. Com a regla general, com més complex és el problema, major és el cost computacional requerit. Afortunadament, existeixen múltiples mètodes per a reduir el cost computacional associat, sense incórrer en prohibitives penalitzacions en la qualitat dels resultats. Un d'aquests mètodes és la resolució de rotors i hèlices mitjançant el mètode de la teoria d'element de pala (BET, per les seues sigles en anglés), acoblada amb la resolució mitjançant volums finits de la resta del camp fluid. Així, és possible resoldre problemes amb una forta interacció entre el sistema propulsiu i les actuacions aerodinàmiques d'una aeronau amb un cost contingut.

OpenFOAM, la llibreria de codi obert de CFD més utilitzada, té en l'actualitat un mètode BET implementat, encara que altament limitat. En aquest treball, l'autor implementa el mètode BET en OpenFOAM dotant-li de multitud de funcionalitats extra de gran utilitat: interpolació en número de Reynolds, número de Mach i angle d'atac, extrapolació en angle d'atac, múltiples mètodes d'emmallat del disc del BET o resolució, entre altres. A més, es desenvolupa una implementació modular, establint una base per al desenvolupament i millora futura d'aquest mètode. La implementació és verificada front el mètode BET de Star-CCM+ obtenint concordances en la predicció de les càrregues fins a l'habitació decimal.

La implementació del BET es realitza dins d'una eina més gran dissenyada, nomenada com *PropellerSource*. En ella s'abstrauen els elements més importants dels mètodes basats en disc actuador com l'emmallat, el control, el càlcul dels termes font, la interpolació o la selecció de cel · les de volums finits. Això permet facilitar l'addició de models nous o millorar els existents.

Índice general

Índice de figuras

Índice de tablas

Nomenclatura

α	Ángulo de ataque
$\boldsymbol{J}$	Matriz Jacobiana
$\boldsymbol{V}$	Velocidad del flujo libre
$\boldsymbol{x}^k$	Componente de la dimensión k del vector $\boldsymbol{x}$
$\boldsymbol{e_\psi}$	Eje de referencia de los ángulos ψ del rotor
$\boldsymbol{e_r}$	Eje en dirección radial de la pala
$\boldsymbol{e_y}$	Eje y del sistema de referencia del rotor
$\boldsymbol{e_z}$	Eje de referencia de la dirección del rotor
$\boldsymbol{I}$	Índices de una lista N-d
$\boldsymbol{U}$	Vector velocidad relativa del rotor y el fluido
η	Rendimiento propulsivo
Λ	Flecha del rotor
ω	Velocidad angular del rotor
ϕ	Ángulo de incidencia del flujo con el plano del rotor
ρ	Densidad volumétrica
θ	Torsión geométrica
θ_0	Torsión geométrica de la pala
$\theta_1 c$	Ángulo de paso cíclico del coseno
$\theta_1 s$	Ángulo de paso cíclico del seno
θ_c	Ángulo de paso colectivo
a	Factor de inducción axial
a'	Factor de inducción tangencial
A_θ	Coeficiente de la fuerza tangencial por unidad de longitud de Goldstein
a_j	Coeficientes de interpolación

A_P	Área de un polígono
A_x	Coeficiente de la fuerza normal por unidad de longitud de Goldstein
c	Cuerda del perfil aerodinámico
c_γ	Velocidad del sonido
C_D	Coeficiente de arrastre 3D
c_D	Coeficiente de arrastre 2D
c_j	Valores de interpolación
C_L	Coeficiente de sustentación 3D
c_L	Coeficiente de sustentación 2D
C_P	Coeficiente de potencia
C_Q	Coeficiente de par
C_T	Coeficiente de empuje
$c_{L,2D}$	Coeficiente de sustentación 2D sin efectos de rotación
$c_{L,pot}$	Coeficiente de sustentación 2D según la teoría potencial
$c_{L,rot}$	Coeficiente de sustentación 2D con efectos de la rotación
dF_D	Fuerza de resistencia por unidad de longitud
dF_L	Fuerza de sustentación por unidad de longitud
dR	Diferencial de radio
f	Factor de corrección para alto α en el método de Snel
F_θ	Primitiva de la distribución de la fuerza tangencial por unidad de longitud de Goldstein
f_θ	Distribución de la fuerza tangencial por unidad de longitud de Goldstein
F_D	Fuerza de resistencia
F_L	Fuerza de sustentación
F_x	Primitiva de la distribución de la fuerza normal por unidad de longitud de Goldstein
f_x	Distribución de la fuerza normal por unidad de longitud de Goldstein
i	Índice de posición en una lista 1D
J	Paso del rotor
M_θ	Integral definida del momento por unidad de longitud de Goldstein

Ma	Número de Mach
N_k	Tamaño de la dimensión k
P	Potencia propulsiva del rotor
p^+	Presión antes de cruzar el disco actuador
p^-	Presión después de cruzar el disco actuador
p_0	Presión del flujo sin perturbar
$P_i(x)$	Polinomio de interpolación cúbico
P_j	Producto del tamaño de las j primeras dimensiones
p_n	Fuerza por unidad de longitud perpendicular al plano del rotor
p_t	Fuerza por unidad de longitud contenida en el plano del rotor
p_w	Presión en la estela del rotor
Q	Par del rotor
q	Momento por unidad de longitud
R	Radio externo del rotor
r	Coordenada radial
r^*	Radio adimensional
r'	Radio adimensionalizado con el radio máximo
r'_h	Relación entre radio máximo y radio interno
Re	Número de Reynolds
T	Empuje del rotor
U_0	Velocidad del flujo sin perturbar
U_n	Velocidad relativa en dirección normal al rotor
U_t	Velocidad relativa en dirección tangencial al rotor
U_w	Velocidad en la estela del rotor
U_{rel}	Velocidad relativa efectiva percibida por el rotor
w	Velocidad inducida por el rotor
w_i	Pesos de la interpolación *inverse distance weighting*
1D	1 dimensión
2D	2 dimensiones

3D 3 dimensiones

CFD *Computer Fluid Dynamics*

FV *Finite Volumes*

N-D N dimensiones

ODS Objetivos de Desarrollo Sostenible

SI Sistema Internacional

Capítulo 1

Introducción

En los últimos años ha crecido la demanda del cálculo de rotores, ligado al crecimiento de la industria de generación de energía eólica y de la industria aeronáutica. Pero habitualmente, dado que se requiería simular un único rotor, el cálculo mediante técnicas habituales resultaba costoso pero efectivo: URANS y LES. Con el auge de las aeronaves pequeñas no tripuladas, ha aumentado la popularidad por el uso de mayor cantidad de rotores debido a los beneficios que esto conlleva. El gran aumento en el número de rotores simulados, junto a la gran cantidad de simulaciones que se requiere para el estudio y optimización de estos, ha generado una demanda en alternativas menos demandantes computacionalmente.

El uso combinado del Blade Element Theory (BET) y la Computational Fluid Dynamics (CFD) ha sido una técnica ampliamente empleada en la simulación y diseño de aerodinámica de aspas de rotor en diversas aplicaciones, especialmente en la industria de la energía eólica y la aviación.

El *Blade Element Method* es un enfoque analítico que descompone la pala de rotor en una serie de elementos (o segmentos) que interactúan con el flujo de manera independiente. Utiliza teorías aerodinámicas para calcular las fuerzas y momentos en cada elemento, teniendo en cuenta el perfil aerodinámico, el ángulo de ataque, la velocidad del viento y otras variables. Es particularmente adecuado para el análisis de turbinas eólicas y hélices de propulsores.

Por otro lado, la Dinámica de Fluidos Computacional (CFD por sus siglas en inglés) es una técnica numérica que resuelve las ecuaciones fundamentales de la mecánica de fluidos para simular el flujo de fluidos alrededor de geometrías complejas. Se basa en la discretización de un dominio en elementos pequeños (celdas) y la solución de las ecuaciones en cada uno de ellos. El CFD es muy versátil y puede manejar flujos turbulentos, tridimensionales y no lineales, lo que lo hace adecuado para analizar el flujo alrededor de las palas de rotor, que tienen geometrías complejas y pueden estar sujetas a flujos turbulentos y no uniformes.

La combinación del BET y CFD se ha vuelto cada vez más popular debido a sus ventajas complementarias. El BET proporciona una estimación rápida y eficiente de las fuerzas y momentos en las palas, mientras que la CFD ofrece una mayor precisión y puede considerar fenómenos más detallados, como la interacción de la estela con el rotor y con los elementos aguas abajo o el suelo. Esta combinación permite mejorar la precisión de las predicciones aerodinámicas y, a su vez, optimizar el diseño y rendimiento de las palas de rotor en diferentes aplicaciones.

En la industria de la energía eólica, el uso combinado del BET y CFD ha permitido mejorar la eficiencia de las turbinas eólicas, reduciendo los niveles de ruido y las cargas mecánicas en las palas, aunque es más habitual el uso de métodos de teoría potencial. En el ámbito de la aviación, esta combinación ha sido empleada para optimizar el rendimiento de los rotores de helicópteros y mejorar la eficiencia aerodinámica de las aspas de aviones. Además, permite el cálculo de interacciones más complejas, como es el caso de la propulsión distribuida y la ingestión de capa límite sin incurrir en costes prohibitivos.

1.1. Motivación y Objetivos

Las herramientas presentes en los códigos comerciales más utilizados en la industria, Ansys y Star-CCM+, presentan un código maduro y funcional que permite el acople entre la teoría de elemento de pala y CFD. No obstante, no presentan la capacidad de realizar la interpolación en número de Mach y de Reynolds de forma simultánea. La alternativa gratuita y de código abierto más extendida es OpenFOAM. El modelo de elemento de pala implementado en OpenFOAM resulta funcional, pero con las siguientes limitaciones:

- No existe la interpolación de polares en número de Reynolds o Mach.
- Solo presenta un método de mallado para el rotor.
- No presenta opciones de configuración para la interpolación.
- No presenta el cálculo transitorio de teoría de elemento de pala.
- Elementos de código repetidos entre modelo de elemento de pala y disco actuador.
- El código está acoplado entre sí y resulta complicado desarrollar sobre él.

Teniendo en cuenta estas limitaciones y la necesidad creciente de la industria, se propone la realización de este trabajo final de máster. Como el resto de proyectos del mismo tipo, uno de los objetivos principales de este trabajo es demostrar haber alcanzado los conocimientos y competencias requeridos para la obtención del título, así como aplicar a un caso real esas habilidades. A parte, este trabajo pretende responder a una necesidad del mercado, y para ello se desarrolla con los siguientes objetivos:

- Sentar las bases de la teoría de elemento de pala en OpenFOAM para facilitar el desarrollo y mejora a partir de este.
- Diseñar una arquitectura modular que englobe todos los tipos de modelos de estilo disco actuador bajo un marco teórico común.
- Facilitar mediante un diseño modular adecuado la incorporación de modelos más avanzados y fomentar el desarrollo de nuevos métodos.
- Alcanzar el nivel de madurez de los métodos del *software* comercial dentro del *software* libre.

Para cumplir con estos objetivos se desarrolla una herramienta bajo el nombre *PropellerSource* implementada dentro del código de OpenFOAM. Cabe recalcar que la obtención de resultados, validación del método de teoría de elemento de pala o la comparación con otros modelos no es objetivo del trabajo. Se realizan ciertas verificaciones que demuestran la correcta implementación del código. El objetivo del trabajo es, por tanto, el DESARROLLO del código.

Como objetivos secundarios, se pretende que este documento sirva de guía para desarrolladores que busquen aprender algunos conceptos sobre el funcionamiento del código de OpenFOAM o de la herramienta *PropellerSource*. Este documento también se desarrolla con el objetivo de servir de guía para los usuarios que quieran utilizar *PropellerSource* en sus proyectos.

1.2. Desarrollo Sostenible

En el presente proyecto, se busca impulsar la innovación y sostenibilidad en el campo de la propulsión mediante el uso de la teoría de elemento de pala. El objetivo principal es desarrollar soluciones tecnológicas que permitan mejorar el diseño para poder mejorar la eficiencia y el rendimiento de sistemas de propulsión. Este objetivo también contribuye significativamente a los Objetivos de Desarrollo Sostenible (ODS) establecidos por las Naciones Unidas.

Este proyecto se alinea estrechamente con los ODS, que representan un marco global para abordar los desafíos sociales, económicos y ambientales que enfrentamos como sociedad. Al enfocarse en la teoría de elemento de pala como solución económica para el cálculo de propulsión distribuida, buscamos impactar positivamente en los siguientes ODS clave:

- ODS 9 - Industria, Innovación e Infraestructura: al impulsar la investigación y el desarrollo en el campo de la propulsión y la teoría de elemento de pala, buscamos fomentar la innovación tecnológica y la construcción de aeronaves más eficientes y respetuosas con el medio ambiente.
- ODS 12 - Producción y Consumo responsables: al reducir el coste computacional no solo se reducen los tiempos de cálculo, también se reduce el coste energético de estas. Esto permite un consumo más responsable de los recursos disponibles. Además, el uso de estas herramientas permite la optimización y reducción del consumo de las aeronaves o el aumento de la producción energética de aerogeneradores.
- ODS 13 - Acción por el Clima: nuestra dedicación a reducir el coste computacional de cálculo ligado a un menor consumo eléctrico y la reducción del gasto energético en estos sistemas de propulsión tiene un impacto directo en la mitigación del cambio climático, al promover soluciones más limpias y sostenibles en el transporte y la industria.

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Proceede
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**				X
ODS 4. **Educación de calidad.**				X
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**				X
ODS 9. **Industria, innovación e infraestructuras.**	X			
ODS 10. **Reducción de las desigualdades**.			X	
ODS 11. **Ciudades y comunidades sostenibles**.			X	
ODS 12. **Producción y consumo responsables.**		X		
ODS 13. **Acción por el clima.**	X			
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

Tabla 1.1: Descripción de la alineación del TFM con los ODS con un grado de relación más alto.

La relación con el resto de ODS implicados se muestra en la Tabla 1.1. A través de una combinación de investigación, desarrollo tecnológico y una firme dedicación a los principios de sostenibilidad, nuestro proyecto aspira a dejar una huella positiva en el mundo. Con cada avance en la reducción de los costes asociados al cálculo y la mejora en los resultados, nos acercamos un paso más hacia un futuro más ecológico y próspero, donde la innovación y la sostenibilidad se entrelacen para el beneficio de las generaciones presentes y futuras.

Capítulo 2

Base Teórica

La mecánica de fluidos computacional (o CFD por sus siglas en inglés) es una disciplina que abarca una gran cantidad de áreas del conocimiento, como son la mecánica de fluidos, las matemáticas, la computación o la informática. Para hacer del CFD una herramienta aplicable no solo es requisito un amplio conocimiento del problema sobre el que se va a aplicar, también es indispensable conocer los métodos numéricos, modelos, tipos de mallado o post procesado de datos entre otros.

En el presente capítulo se van a desarrollar el contexto teórico en el que se ha desarrollado el proyecto, para comprender la base teórica del trabajo y los fundamentos utilizados para la implementación de las distintas herramientas utilizadas. En primer lugar se introducirá el método utilizado para modelar las fuerzas en un rotor. Posteriormente, se definirán las herramientas que permiten la implementación del método, como son los métodos numéricos, la extrapolación de polares, los algoritmos y las técnicas de mallado de disco.

2.1. Método de teoría de elemento de pala

2.1.1. Introducción

El método de teoría de elemento de pala (o BET por sus siglas en inglés) fue propuesto por W. Froude y S. Drzeweicki para calcular las cargas en un rotor. El método requiere 3 elementos para el cálculo: conocer la geometría de las palas, las polares de los perfiles que las forman y la velocidad en el rotor. Es necesario utilizar algún método para obtener la velocidad en el rotor, ya que las cargas aerodinámicas modifican el campo fluido, por lo que la velocidad en el rotor no es la misma que la del flujo sin perturbar o «libre». Los dos métodos que se explican más adelante para obtener la velocidad en el rotor son: el acople con la teoría del momento y el acople con el método de volúmenes finitos; siendo este último el que se implementa y desarrolla en el presente trabajo [3].

Las hipótesis principales asumidas por la BET son:

1. Las cargas de la pala en una posición radial se pueden obtener de forma independiente al resto de posiciones.

2. Las cargas de la pala se pueden obtener a partir de las fuerzas 2D de los perfiles aerodinámicos.

3. La velocidad del flujo en el rotor es conocida y se asume que la teoría de la linea sustentadora es válida.

2.1.2. Cálculo de un elemento

Un elemento de pala consiste en una porción de la pala caracterizado por una longitud radial dr , una posición radial r y la geometría para esa posición radial. En el presente trabajo los elementos geométricos utilizados para definir cada elemento son: la cuerda (c), la torsión (θ), la flecha (Λ) y el perfil alar (representado por sus polares). La geometría se expresa como una función arbitraria del radio.

Para el cálculo del elemento de pala primero es necesario definir los sistemas de referencia utilizados. El rotor se encuentra en una posición y orientación arbitraria en el espacio, definido por un punto (x, y, z) y un vector normal al plano del rotor $\boldsymbol{e_z}$. El sistema de referencia del rotor, esta formado por los siguientes 3 ejes:

- Eje (x), referencia de ψ: $\boldsymbol{e_\psi}$
- Eje (z), el mismo que el rotor: $\boldsymbol{e_z}$
- Eje (y): $\boldsymbol{e_y}$ de forma que $\{\boldsymbol{e_\psi},\boldsymbol{e_z},\boldsymbol{e_y}\}$ forme un triedro a derechas.

La velocidad angular del rotor, ω se define positiva según una rotación positiva del eje $\boldsymbol{e_z}$ según se observa en la Figura 2.1.

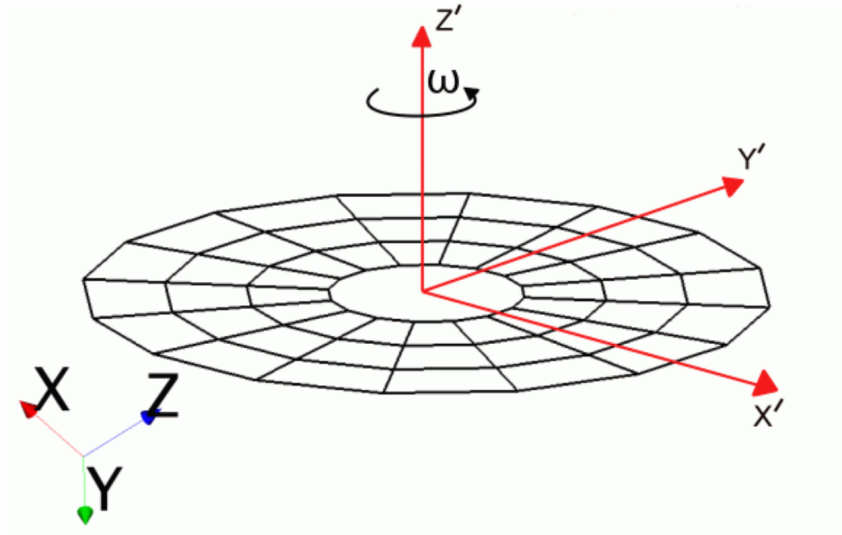

Figura 2.1: Sistema de referencia del rotor [25]

El análisis se realiza en el sistema de referencia no inercial del elemento de pala, donde los 3 ejes que definen el sistema de referencia son:

- Eje (x) en dirección radial: $\boldsymbol{e_r}$, positivo en dirección del centro a la punta de pala.
- Eje (z) en dirección del plano del rotor: $\boldsymbol{e_z}$, positivo en la misma dirección que el rotor.

- Eje (y) en dirección azimutal: $\boldsymbol{e}_{\psi}$, definido por el producto vectorial de los dos anteriores.

En la Figura 2.2 se observan los aspectos geométricos de la BET en el plano del rotor.

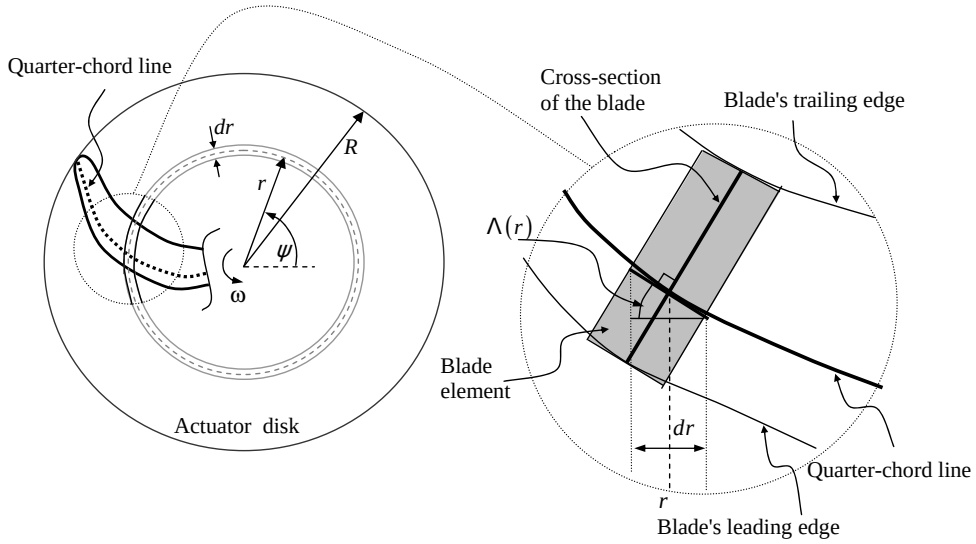

Figura 2.2: Rotor y discretización de la pala [23]

Quedando definida la geometría y el sistema de referencia, se define el ángulo de ataque del elemento como:

$$\alpha = \phi - \theta \tag{2.1}$$

$$\phi = \arctan \frac{U_n}{U_t} \tag{2.2}$$

Donde ϕ es el ángulo entre la velocidad y el plano del rotor y θ el ángulo que forma el elemento de pala con el plano del rotor. La velocidad incidente (U) se compone de la velocidad del flujo libre (V), la inducida por el rotor (w) y la de la pala, que para el caso de solo existir movimiento de rotación puro tiene la expresión de $\omega \cdot r$. En la Figura 2.3 se observa la descomposición de velocidades para el caso genérico en el que existe flecha variable. La velocidad de referencia utilizada para el cálculo de la sustentación se determina como:

$$U_{rel} = |\boldsymbol{U}| \cos \Lambda \tag{2.3}$$

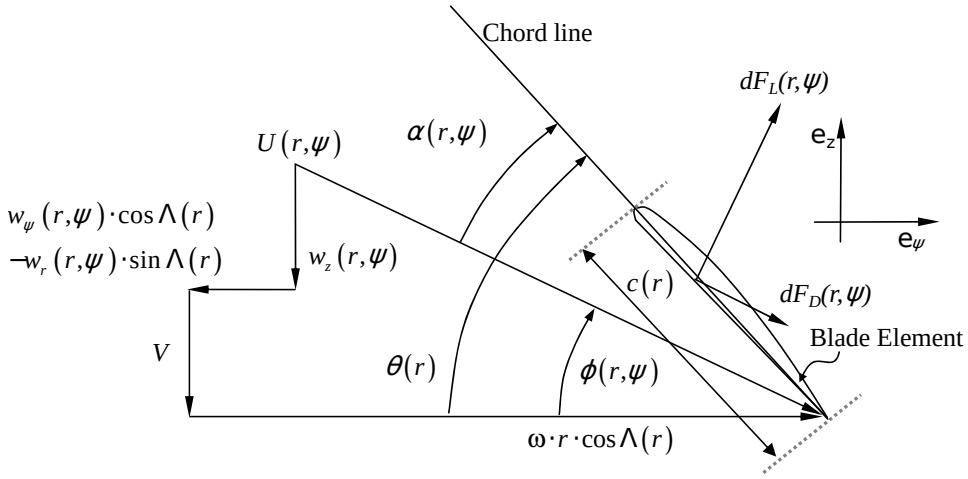

Figura 2.3: Descomposición de la velocidad en un elemento de pala [23]

El siguiente paso es definir los diferenciales de fuerzas aerodinámicas en ejes viento, siendo ρ la densidad del fluido que trasiega el elemento y considerando que el coeficiente de sustentación c_L y resistencia c_D son conocidos:

$$dF_L = \frac{1}{2}\rho\, c\, C_L\, U_{rel}^2\, \mathrm{d}r \tag{2.4}$$

$$dF_D = \frac{1}{2}\rho\, c\, C_D\, U_{rel}^2\, \mathrm{d}r \tag{2.5}$$

Expresando las fuerzas por unidad de longitud:

$$L = \frac{1}{2}\rho\, c\, C_L\, U_{rel}^2 \tag{2.6}$$

$$D = \frac{1}{2}\rho\, c\, C_D\, U_{rel}^2 \tag{2.7}$$

Proyectando las fuerzas sobre el sistema de referencia del elemento de pala:

$$p_n = L\, \cos\phi + D\, \sin\phi \tag{2.8}$$

$$p_t = L\, \sin\phi - D\, \cos\phi \tag{2.9}$$

Para obtener el empuje del rotor se integra la Ecuación 2.8 en toda la pala, para el caso del par se multiplica la Ecuación 2.9 por el radio de la siguiente forma:

$$q = p_t \cdot r \tag{2.10}$$

Y de la misma forma que el empuje, se integra en todo el dominio. Por lo tanto la fuerza, momento y potencia en una pala:

$$T = \int_0^R p_n\, \mathrm{d}r \tag{2.11}$$

$$Q = \int_0^R p_t \cdot r\, \mathrm{d}r \tag{2.12}$$

$$P = \int_0^R p_t \cdot \omega \cdot r \, \mathrm{d}r \tag{2.13}$$

Además se definen el número de Reynolds y Mach del elemento de pala como:

$$Re = \frac{\rho \, U_{rel} \, c}{\mu} \tag{2.14}$$

$$Ma = \frac{U_{rel}}{c_\gamma} \tag{2.15}$$

Donde μ es la viscosidad cinemática del fluido y c la velocidad del sonido en el fluido. Todos los cálculos del procedimiento anterior son directos, el problema surge con la forma de obtener la velocidad inducida por el rotor (w) ya que la solución requiere de información adicional. A continuación se comentarán dos métodos para obtener la velocidad inducida por el rotor en el campo fluido: la teoría del momento y el método de volúmenes finitos.

2.1.3. Acople con teoría del momento

La fuerza del rotor está determinada por la velocidad local que percibe la pala a través del ángulo de ataque y del módulo de esta. Las fuerzas aerodinámicas originadas en el rotor modifican el propio campo fluido modificando la velocidad percibida por el rotor.

Una de las soluciones es propuesta por la teoría del momento. Esta se basa en aplicar las leyes de conservación de la dinámica de fluidos. Para simplificar y poder cerrar las ecuaciones, se realizan una serie de hipótesis:

- El flujo es estacionario, incompresible y axisimétrico
- El fluido es homogéneo y no-viscoso
- Las fuerzas en el rotor son axisimétricas y concentradas en un disco actuador 2D

Para que se cumplan las hipótesis, el disco tiene que estar formado por un número infinito de palas. El fluido ejerce una fuerza resultante sobre el disco (T), un momento (Q) y una potencia (P), opuestas en signo a las que el disco ejerce sobre el fluido. Se produce, por tanto, una discontinuidad en el campo de la presión a través del disco actuador.

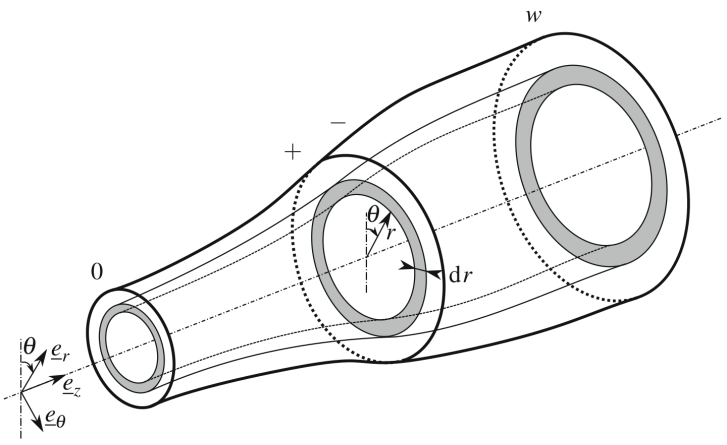

Figura 2.4: Notación utilizada para el tubo de corriente del rotor. Los tubos anulares están representados de forma que las zonas grises son secciones transversales en diferentes planos [3]

Con estos métodos se persigue llegar a una expresión para la velocidad inducida en el fluido en función de la carga del rotor. Esta velocidad se expresa habitualmente con el factor de inducción axial (a) y tangencial (a'), definidos como:

$$a = 1 - \frac{U_n}{U_0} \tag{2.16}$$

$$a' = 1 + \frac{U_t}{\omega\, r} \tag{2.17}$$

Las componentes de la velocidad inducida están incluidos en la velocidad percibida por la pala $\boldsymbol{U}$.

Según las hipótesis simplificadoras, aparecen distintos métodos de la teoría del momento, la más básica es llamada teoría del momento 1D simplificada y es atribuida al trabajo realizado por Rankine y R.E. Froude [3]. Este método añade distintas hipótesis a las enumeradas anteriormente:

- El salto de presión y la velocidad axial en el disco es uniforme en todo el área
- No existe rotación de la estela
- La presión estática aguas arriba y aguas abajo del rotor tiende a la presión de la corriente sin perturbar

El volumen de control utilizado está delimitado por las líneas de corriente que pasan por el borde del disco actuador, apareciendo una contracción/expansión de estas al pasar a través del disco como se observa en la Figura 2.5

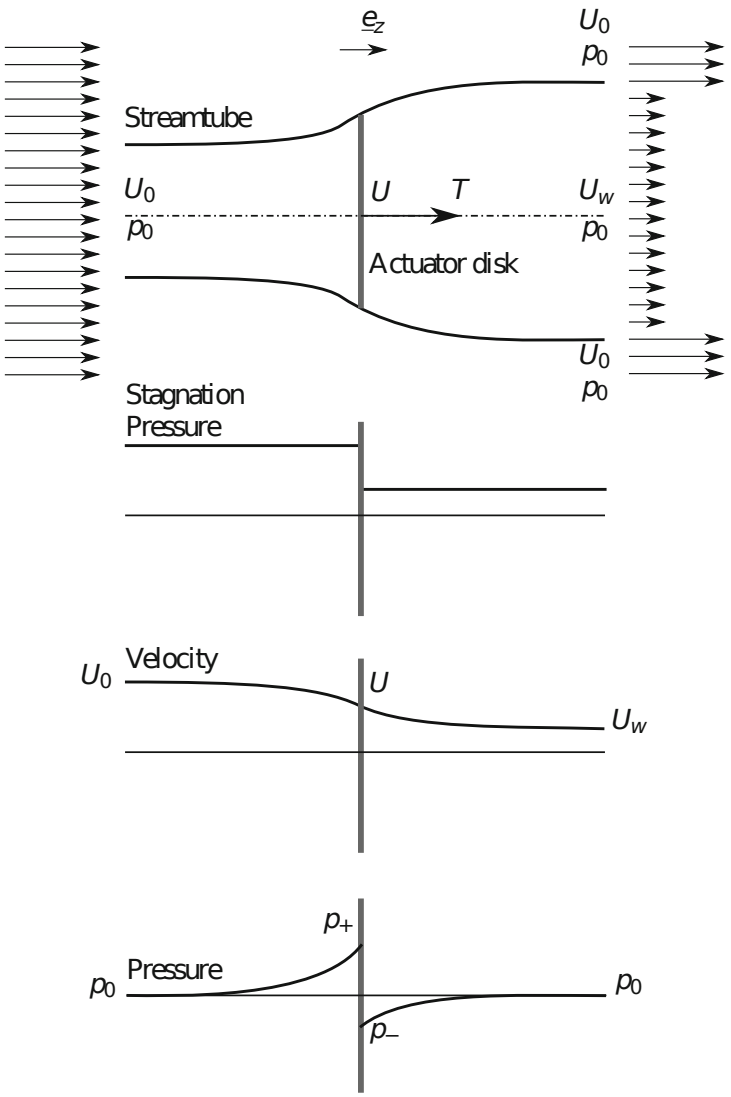

Figura 2.5: Esquema de la teoría del momento 1D simplificada y la distribución de la presión de parada, velocidad y presión estática [3]

A continuación, se enumera la formulación del método, sin entrar en detalle en el desarrollo completo, explicado en su totalidad en el libro de Branlard [3].

Partiendo de la ecuación de Bernoulli y aplicando las hipótesis simplificadoras se llega a la expresión del salto de presión en el disco actuador:

$$\Delta p = p_+ - p_- = \frac{1}{2}\rho\,(U_0^2 - U_w^2) \tag{2.18}$$

Siendo U_w la velocidad en la estela. Para obtener la expresión del empuje, se aplican las ecuaciones de conservación de la masa y del momento de Navier-Stokes al volumen de control, resultando en:

$$T = \rho A U (U_0 - U_w) \tag{2.19}$$

La potencia extraída o aplicada por el rotor es:

$$P = T\,U = \dot{m}\,U(U_0 - U_w) \tag{2.20}$$

Despejando U de las ecuaciones anteriores obtenemos la relación entre las velocidades:

$$U = \frac{U_0 - U_w}{2} \tag{2.21}$$

Y finalmente la velocidad inducida en el rotor:

$$w = \frac{U_w - U_0}{2} \tag{2.22}$$

Y el factor de inducción:

$$a = \frac{U_0 - U}{U_0} \tag{2.23}$$

Conociendo la velocidad inducida, se cierran las ecuaciones de BET, conociéndose en su conjunto como BEMT (Blade Element Momentum theory). En la Figura 2.6 se observa como se realiza el acoplamiento entre las dos teorías:

1. Se parte de un factor de inducción inicial
2. Se obtienen el ángulo de ataque y la velocidad relativa Ecuación 2.1
3. Se obtienen las cargas del rotor con la Ecuación 2.4 y la Ecuación 2.5
4. Se recalcula el factor de inducción
5. Se vuelve al paso 2

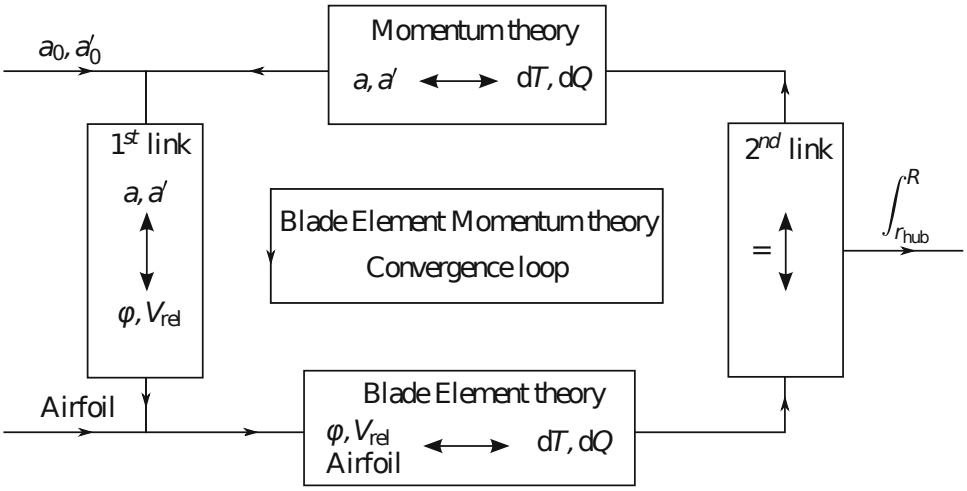

Figura 2.6: Acoplamiento de la teoría de elemento de pala con teoría del momento

2.1.4. Acople con volúmenes finitos

Un método alternativo a la teoría del momento es el uso de CFD para obtener el campo de velocidades que aparece en el rotor, en cual se centra el presente trabajo. En concreto se desarrollará el método aplicado a volúmenes finitos. En este caso no hay restricciones en las hipótesis del método, mas allá que las ya asumidas en las propias ecuaciones resueltas mediante volúmenes finitos. Las hipótesis del BET se mantienen.

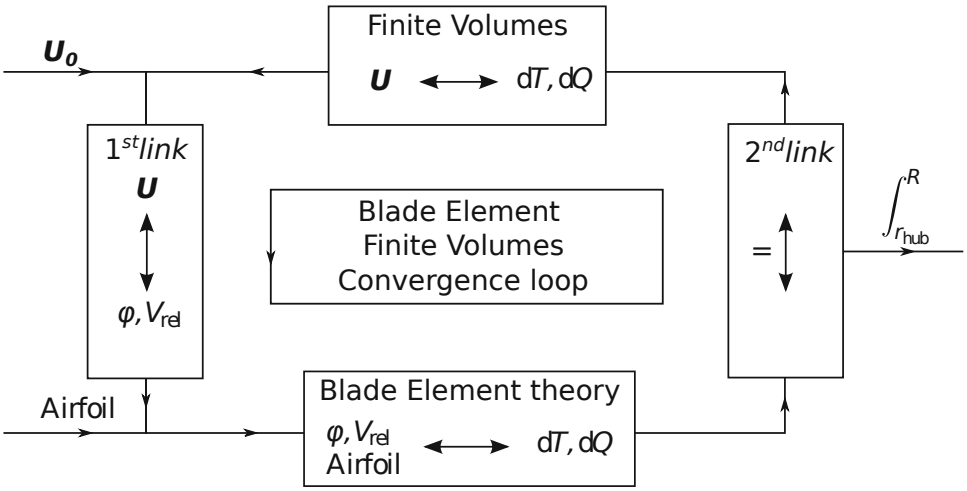

Figura 2.7: Acoplamiento de la teoría de elemento de pala con volúmenes finitos

El método de volúmenes finitos aplicado a CFD resuelve las ecuaciones de Navier-Stokes y modelos adicionales, en un dominio finito, discretizado en celdas. Por tanto, la precisión del BET estará ligada a la discretización de volúmenes finitos.

Para acoplar ambos métodos se utilizan dos mallas independientes y una interpolación entre ellas. Por un lado, se tiene la malla de volúmenes finitos (malla FV), de la cual se selecciona una región que contendrá las celdas que forman parte del rotor. Por otro lado, se tiene la malla del método de teoría de elemento de pala (malla BET), que define los elementos de pala, en los que se calculan las cargas aerodinámicas. Para realizar el primer enlace, es necesario interpolar la solución CFD (campo de velocidades y/o densidad) de la malla FV a la malla del rotor. Para el caso del segundo enlace, se interpolan las cargas del rotor de la malla del rotor a la malla FV, que aparecen en las ecuaciones de Navier-Stokes como un término fuente volumétrico.

En la Figura 2.7 se observa como se realiza el acoplamiento:

1. Se parte de un campo de velocidades inicial en la solución CFD
2. Se interpola el campo de velocidades y se obtienen el ángulo de ataque y la velocidad relativa
3. Se obtienen las cargas del rotor con la Ecuación 2.4 y la Ecuación 2.5
4. Se interpolan las cargas a la malla FV y se obtienen los términos fuente
5. Se realiza una iteración de las ecuaciones de Navier-Stokes
6. Se vuelve al paso 2

2.1.5. Efectos de rotación 3D

En algunos casos, los efectos de la rotación no son despreciables y producen resultados diferentes que los predichos por los coeficientes 2D. En el caso de las turbinas de viento, los cálculos 2D predicen valores inferiores de potencia que los obtenidos experimentalmente

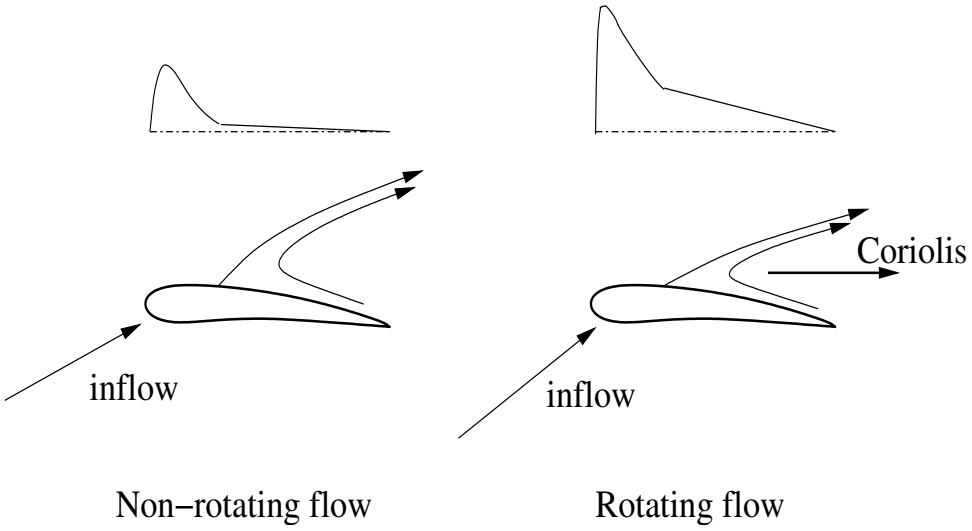

Figura 2.8: Efectos 3D de la rotación en la distribución de presión y el flujo [13]

[13]. Por esto surge la necesidad de incorporar modelos que incluyan los efectos de la rotación en el cálculo de la BET.

Un sistema de referencia sobre la pala no es inercial, por lo que aparecen fuerzas inerciales que actúan sobre el fluido: la fuerza centrífuga y de Coriolis. El efecto de la fuerza centrífuga, que actúa desde el centro hacia la punta de la pala, es un gradiente de presión, que provoca cierta velocidad radial. Al aparecer esta velocidad radial, actúa la fuerza de Coriolis, que acelera la corriente hacia el borde de fuga. Esto dificulta el desprendimiento de la capa límite y por tanto la entrada en perdida del perfil. Este fenómeno puede entenderse como un mecanismo que estabiliza la capa límite sobre el perfil, incluso a elevados ángulos de ataque. Además, modifica el ángulo de incidencia de la corriente por el efecto de succión hacia el borde de fuga [13]. Estos dos efectos combinados aumentan el pico de succión y la sustentación, quedando ilustrado en la Figura 2.8.

Método de Snel et al.

Snel, Houwink y Bosscher [26] desarrollaron un método para tener en cuenta el aumento de la sustentación por efecto de la rotación, el modelo se aplica al coeficiente de sustentación mediante la siguiente expresión:

$$c_{L,rot} = c_{L,2D} + 3{,}1\left(\frac{c}{r}\right)^2 (c_{L,pot} - c_{L,2D}) \cdot f(\alpha, r/R) \tag{2.24}$$

Donde $c_{L,rot}$ es el coeficiente de sustentación después de aplicar la corrección, V_{eff} el módulo de la velocidad percibida por el perfil y $c_{L,pot}$ es el coeficiente de sustentación del perfil según la teoría potencial. Posteriormente, se realizaron correcciones [12] al modelo de Snel, para tener en cuenta la relación entre la velocidad de rotación y la velocidad efectiva del flujo (Snel + pumping):

$$c_{L,rot} = c_{L,2D} + 3{,}1\left(\frac{\omega\, r}{V_{eff}}\right)^2 \left(\frac{c}{r}\right)^2 (c_{L,pot} - c_{L,2D}) \cdot f(\alpha, r/R) \tag{2.25}$$

Estos modelos son válidos para $\alpha \in [0, 30°]$ y radios inferiores al 80 % del radio máximo $r/R < 0{,}8$. Para ángulos de ataque superiores a 30° se reduce el factor de corrección

de forma lineal hasta 50° que se vuelve nulo [12]. Teniendo esto en cuenta, se define la función:

$$f(\alpha, r/R) = \begin{cases} 1, & if \quad \alpha \leq 30 \text{ and } r/R < 0{,}8 \\ 1 - \frac{\alpha - 30°}{20}, & if \quad 30° < \alpha < 50° \text{ and } r/R < 0{,}8 \\ 0, & otherwise \end{cases} \tag{2.26}$$

Esta función representa un factor sobre el término correctivo del c_L, aplicando las condiciones mencionadas en el párrafo anterior. En la Figura 2.9, se puede observar la comparación de los dos métodos aplicado a la polar de c_L de un perfil aerodinámico. Se puede observar que para el tramo lineal coinciden las tres curvas, puesto que aquí el desprendimiento es nulo y el efecto de la rotación despreciable. A partir de ahí, las correcciones divergen a partir del tramo en el que empieza a aparecer una zona de desprendimiento sobre el perfil. Cerca de un valor para el ángulo de ataque de 30°, las correcciones disminuyen por el factor f, colapsándose todos los valores a los 50°.

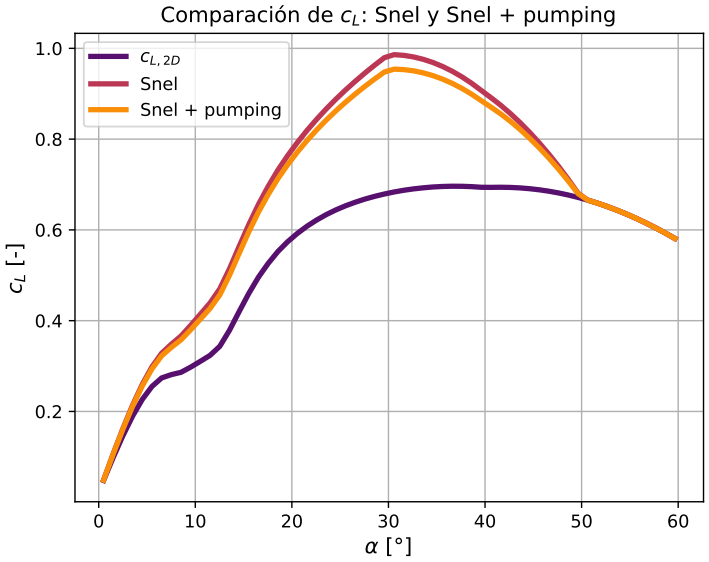

Figura 2.9: Comparación de los métodos de Snel y Snel + pumping para $r/R = 0{,}5$ y $J = 0{,}5$

En la Figura 2.10 y Figura 2.11, se comparan las correcciones por el método de Snel y Snel + pumping, para diferentes valores de radio adimensional, manteniendo las otras variables constantes. Se puede observar que para relaciones de radio menores, es decir, cerca de la raíz, los efectos son mucho mas pronunciados.

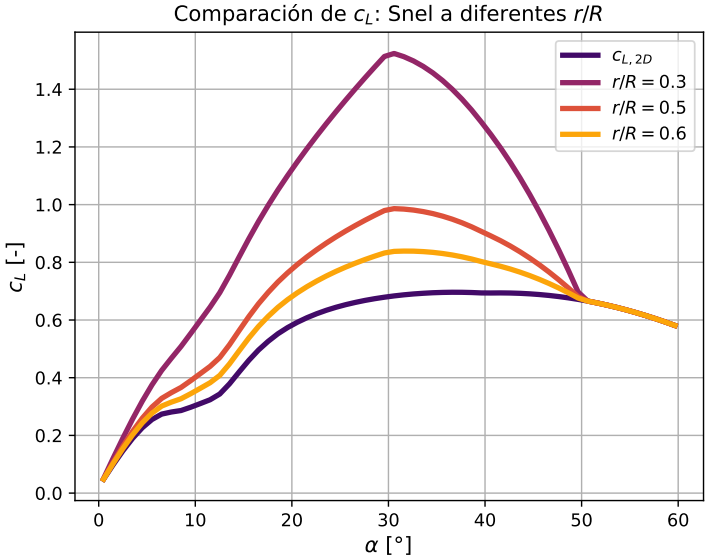

Figura 2.10: Comparación del método Snel para distintos valores de r/R

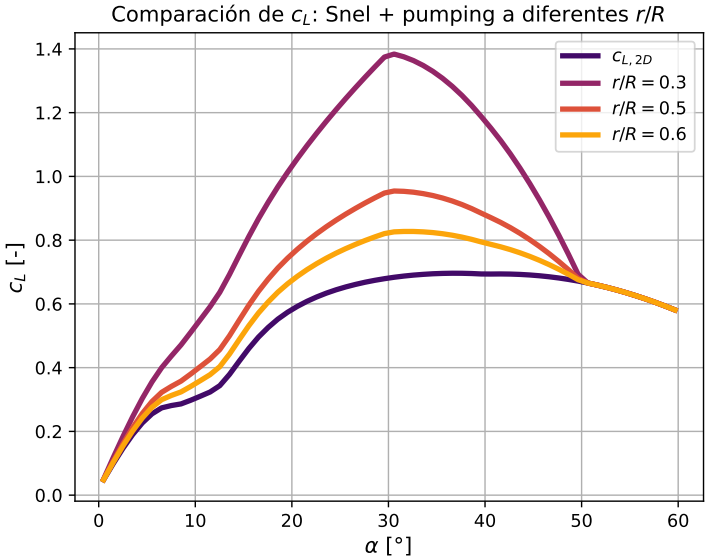

Figura 2.11: Comparación del método Snel pumping para distintos valores de r/R

En la Figura 2.12, se muestra para distintos valores de la velocidad incidente la corrección Snel+pumping, manteniendo el resto de parámetros fijos, lo que es equivalente a modificar el paso J. Se puede observar, que una mayor velocidad incidente reduce ligeramente los valores de c_L. Dado la diferencia entre las comparaciones de la variación del radio y velocidad, se puede concluir, que es mucho más sensible a la variación en el radio.

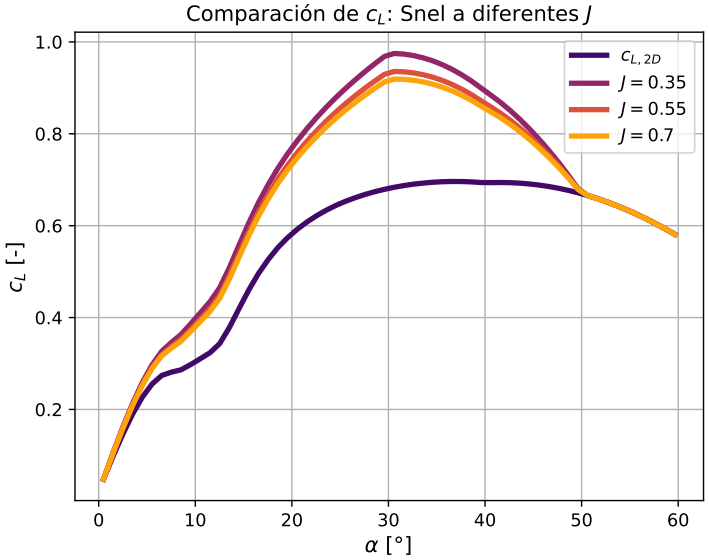

Figura 2.12: Comparación del método Snel + pumping para distintos valores de J

2.1.6. Distribución de Goldstein

En algunos casos, se quiere acoplar el método de volúmenes finitos con un rotor, pero no se dispone de los datos de las polares. Si el interés reside en estudiar los efectos del rotor sobre el campo fluido y son conocidas las curvas de coeficiente de empuje y coeficiente de par, se puede utilizar la distribución óptima de Goldstrein [25] para obtener las fuerzas sobre la pala:

$$f_x(r^*) = A_x\, r^* \sqrt{1 - r^*} \tag{2.27}$$

$$f_\theta(r^*) = A_\theta \frac{r^* \sqrt{1 - r^*}}{r^*(1 - r'_h) + r'_h} \tag{2.28}$$

$$r^* = \frac{r' - r'_h}{1 - r'_h} \tag{2.29}$$

$$r'_h = \frac{R_H}{R_P} \tag{2.30}$$

$$r' = \frac{r}{R_P} \tag{2.31}$$

Siendo R_H el radio interno de la pala, R_P el radio total de la pala, f_x la fuerza en dirección normal al plano del rotor y f_θ la fuerza en dirección azimutal (con signo positivo en dirección contraria al movimiento de la pala).
Las definiciones se obtienen del manual de StarCCM+, realizandoles algunas modificaciones. En el manual aparece una expresión para obtener los coeficientes A_x y A_θ, utilizando las distribuciones de fuerzas y el Δr de cada elemento de pala para integrar numéricamente. Para mejorar la precisión de los resultados, se decide obtener las expresiones de las primitivas para obtener los valores de forma analítica. Por tanto:

$$F_x(r^*) = \int f_x(r^*)\, \mathrm{d}r^* = A_x \int r^* \sqrt{1 - r^*}\, \mathrm{d}r^* = \frac{-2}{15}\,(3x + 2)(1 - x)^{\frac{3}{2}} \tag{2.32}$$

$$F_\theta(r^*) = \int f_\theta(r^*)\,\mathrm{d}r^* = A_\theta \int \frac{r^*\sqrt{1-r^*}}{r^*(1-r'_h)+r'_h}\,\mathrm{d}r^*$$
$$= -\frac{2\sqrt{1-r^*+r'_h(2+r^*)}}{3(r'_h-1)^2} - \frac{r'_h \log\frac{1-\sqrt{(1-r'_h)(1-r^*)}}{1+\sqrt{(1-r'_h)(1-r^*)}}}{(1-r'_h)^{\frac{5}{2}}} \tag{2.33}$$

Para obtener el coeficiente A_x se utiliza el valor del empuje obtenido a partir de las tablas proporcionadas:

$$A_x = \frac{T}{F_x(1)-F_x(0)} = \frac{15\,T}{4} \tag{2.34}$$

Para el caso del coeficiente A_θ, se utiliza el valor de par obtenido a partir de las tablas. Por tanto, se requiere integrar el momento producido por la fuerza distribuida f_θ:

$$M_\theta = \int_{r^*=0}^{r^*=1} r^*\, f_\theta(r^*)\,\mathrm{d}r^* = \int_{r^*=0}^{r^*=1} \frac{r^{*2}\sqrt{1-r^*}}{r^*(1-r'_h)+r'_h}\,\mathrm{d}r^*$$
$$= \frac{2\,(8r'^2_h+9r'_h-2)}{15(r'_h-1)^3} - \frac{r'^2_h \log\left(\frac{1-\sqrt{1-r'_h}}{1+\sqrt{1-r'_h}}\right)}{(1-r'_h)^{7/2}} \tag{2.35}$$

$$A_\theta = \frac{Q}{M_\theta} \tag{2.36}$$

Para obtener la fuerza de cada elemento anular entre r^*_i y r^*_{i+1} se utilizan las expresiones de las primitivas:

$$F_{x,i} = F_x(r^*_{i+1}) - F_x(r^*_i) \tag{2.37}$$

$$F_{\theta,i} = F_\theta(r^*_{i+1}) - F_\theta(r^*_i) \tag{2.38}$$

2.2. Extrapolación de polares

Para la aplicación del método de teoría de elemento de pala es necesario disponer de polares completas de los perfiles aerodinámicos utilizados, ya que no hay ninguna restricción al ángulo de incidencia del flujo en la simulación. Aunque es necesario, en la mayoría de casos de interés el ángulo de ataque se situará en ángulos de ataque relativamente pequeños.

La mayoría de polares de perfiles aerodinámicos se obtienen para pequeños ángulos de ataque, siendo el valor máximo, cercano al ángulo de entrada en perdida. Existen métodos para obtener las polares completas como el desarrollado por Manfred Imiela et al. [8] o Manfred Imiela et al.[9], aunque son métodos elaborados y costosos. Existen métodos con un coste más reducido, pero requieren una gran cantidad de parámetros geométricos, propuesto por Khiem Van Truong [28]. Por tanto, surge la necesidad de un método para la extrapolación de polares a partir de los datos de un rango reducido de ángulos de ataque que no añada una excesiva complejidad.

Un método sencillo y popular fue propuesto por Viterna et al.[31] en 1982 basado en el ajuste de una ecuación para el coeficiente de sustentación y otra para el coeficiente de arrastre. Esta extrapolación es común utilizarse hasta los -90^o y 90^o, a partir de esos ángulos es posible utilizar la teoría de placa plana para estimar los coeficientes de sustentación y arrastre [20].
En la implementación, se utilizan los datos disponibles de la polar para el rango de ángulos de ataque, la extrapolación de Viterna hasta los -90^o y 90^o y extrapolación de placa plana para el resto. Las ecuaciones de la extrapolación de Viterna son:

$$c_L = A_1 \sin(2\alpha) + A_2 \frac{\cos^2 \alpha}{\sin \alpha} \tag{2.39}$$

$$c_D = B_1 \sin^2 \alpha + B_2 \cos \alpha \tag{2.40}$$

Siendo los coeficientes A_1, A_2, B_1 y B_2:

$$A_1 = \frac{c_{D,max}}{2} \tag{2.41}$$

$$A_2 = \frac{\sin(\alpha_{stall})}{\cos^2(\alpha_{stall})} \left(c_{L,stall} - c_{D,max} \sin(\alpha_{stall}) \cos(\alpha_{stall})\right) \tag{2.42}$$

$$B_1 = c_{D,max} \tag{2.43}$$

$$B_2 = \frac{c_{D,stall} - c_{D,max} \sin(\alpha_{stall})}{\cos(\alpha_{stall})} \tag{2.44}$$

Las ecuaciones para placa plana son:

$$c_L = c_{L,max} \sin \alpha \cos \alpha \tag{2.45}$$

$$c_D = c_{D,max} \sin^2 \alpha \tag{2.46}$$

Para el caso de una placa plana infinita $c_{L,max}$ y $c_{D,max}$ toman el valor de 2. Para el caso de una pala se utiliza la siguiente expresión [31]:

$$c_{L,max} = c_{D,max} = 1{,}11 + 0{,}018\, AR \tag{2.47}$$

Siendo AR el alargamiento de la pala(*aspect ratio* en ingles).

En la implementación, no utiliza α_{stall} como el ángulo de ataque de entrada en pérdida, si no el último ángulo de ataque para el cual se tienen datos, tanto en positivo como en negativo. De la misma forma para $c_{D,stall}$ y $c_{L,stall}$. Esto se realiza con el objetivo de utilizar todos los datos proporcionados de las polares, ya que se supone *a priori* que son más exactos que la extrapolación.

Para extrapolar el c_L con el método Viterna a ángulos negativos, se calculan los coeficientes A_1 y A_2 con los valores absolutos de $c_{L,stall}$ y α_{stall} y se aplica una transformación antisimétrica al c_L extrapolado mediante la Ecuación 2.39:

$$c_L(-\alpha) = -c_L(\alpha) \tag{2.48}$$

El método utilizado para extrapolar el c_L y el c_D se puede expresar mediante una función a trozos, donde los superíndices $^+$ y $^-$ representan los coeficientes para valores positivos y negativos respectivamente:

$$c_L(\alpha) = \begin{cases} -c_{L,max}\sin\alpha\cos\alpha, & if \quad -180° \leq \alpha < 90° \\ -A_1^-\sin(2\alpha) - A_2^-\left(\frac{\cos^2\alpha}{\sin\alpha}\right), & if \quad -90° \leq \alpha < \alpha_{stall}^- \\ c_{L,airfoil}(\alpha), & if \quad \alpha_{stall}^- \leq \alpha \leq \alpha_{stall}^+ \\ A_1^+\sin(2\alpha) + A_2^+\left(\frac{\cos^2\alpha}{\sin\alpha}\right), & if \quad \alpha_{stall}^+ < \alpha \leq 90° \\ c_{L,max}\sin\alpha\cos\alpha, & if \quad 90° < \alpha < 180° \end{cases} \tag{2.49}$$

$$c_D(\alpha) = \begin{cases} c_{D,max}\ \sin^2\alpha, & if \quad -180° \leq \alpha < 90° \\ B_1^-\sin^2\alpha + B_2^-\cos\alpha, & if \quad -90° \leq \alpha < \alpha_{stall}^- \\ c_{D,airfoil}(\alpha), & if \quad \alpha_{stall}^- \leq \alpha \leq \alpha_{stall}^+ \\ B_1^+\sin^2\alpha + B_2^+\cos\alpha, & if \quad \alpha_{stall}^+ < \alpha \leq 90° \\ c_{D,max}\ \sin^2\alpha, & if \quad 90° < \alpha < 180° \end{cases} \tag{2.50}$$

En la Figura 2.14, se observan la extrapolación completa de c_L y c_D para el caso de un perfil tipo placa plana. Se observan en distintos colores los distintos métodos utilizados en cada intervalo. Aunque no se trate de una aproximación extremadamente precisa, permite calcular rotores mediante BET en cualquier situación, sin requerir de información adicional lo que mejora la versatilidad del método.

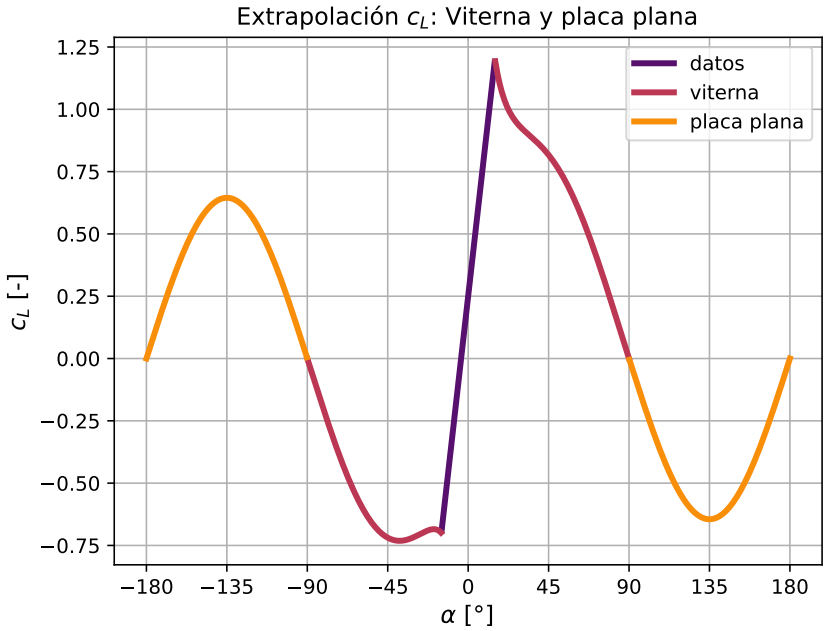

Figura 2.13: Extrapolación del c_L

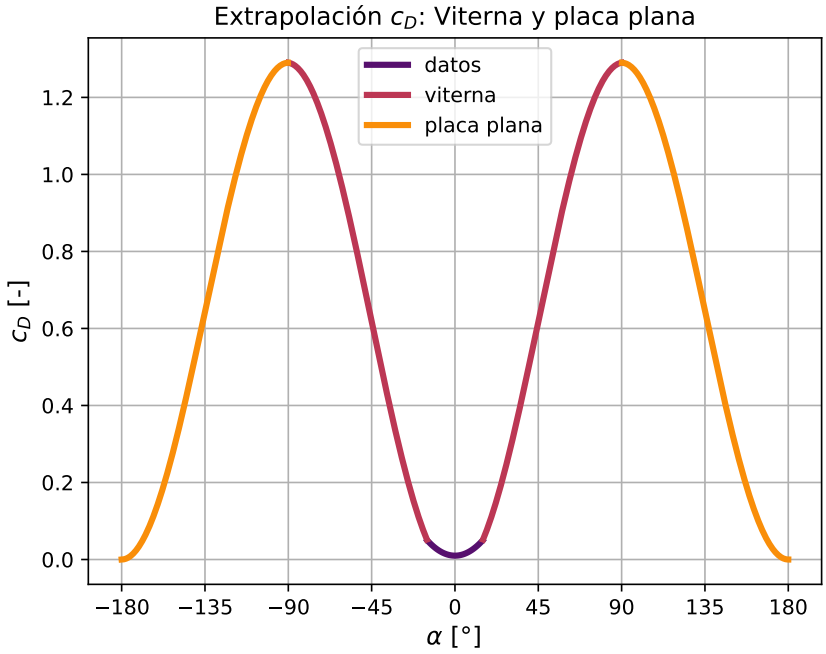

Figura 2.14: Extrapolación del c_D

Figura 2.15: Extrapolación de Viterna y placa plana junto los datos conocidos

2.3. Métodos numéricos

Los métodos numéricos son procedimientos mediante los cuales se obtienen la solución de ciertos problemas utilizando cálculos aritmético-lógicos. La solución obtenida suele

ser aproximada, permitiendo un ajuste del error aceptable mediante la tolerancia. Se presentan a continuación los métodos numéricos utilizados e implementados, necesarios para los distintas partes de la implementación del método de elemento de pala y su acople con volúmenes finitos.

Ya que muchos de los datos requeridos por BEM son introducidos en forma de tablas, es necesario disponer de distintos métodos de interpolación adecuados a los datos disponibles. También surge la necesidad, en el problema de trimado del rotor, de resolver sistemas de ecuaciones que no presentan solución analítica, ya que se obtiene su resultado mediante volúmenes finitos, por lo que para su resolución se opta por métodos numéricos, como es el caso de Newton-Raphson multidimensional. Para el cálculo de áreas de polígonos irregulares, utilizado para la malla BET, se utiliza el método de Gauss.

2.3.1. Interpolación

Una gran parte de los datos requeridos por el método de elemento de pala suelen proveerse en forma de tablas por parte del usuario, principalmente la geometría y las polares. La geometría de las palas viene determinada para valores discretos de radios, por lo que es necesario interpolar para obtener geometrías intermedias. Las polares de los perfiles, también vienen determinadas en la mayoría de casos por tablas, para valores discretos de ángulo de ataque, número de Reynolds y número de Mach. Además, los perfiles aerodinámicos pueden variar en función del radios, por lo que será necesario realizar interpolaciones entre perfiles. Por ello, se busca desarrollar métodos genéricos que sirvan para un número variable de dimensiones y puedan expresarse como un sumatorio de coeficientes multiplicados por los valores conocidos.

En el campo de las matemáticas, la interpolación consiste en la obtención de nuevos puntos a partir de un conjunto discreto de valores conocidos dentro de una región. Si se busca obtener puntos fuera de esta región delimitada por los puntos conocidos, el proceso es conocido como extrapolación. En general se conoce una lista de N puntos en $\mathcal{R}^{ND}\left[\boldsymbol{x}_0, \boldsymbol{x}_i, \ldots, \boldsymbol{x}_{N-2}, \boldsymbol{x}_{N-1}\right]$ y sus valores asociados $[a_0, a_1, \ldots, a_{N-2}, a_{N-1}]$

En función de la organización de los datos podemos distinguir:

- **Estructurados**: los puntos conocidos se sitúan en una red estructurada N-d, no necesariamente equidistantes.
- **No estructurados**: lo opuesto a lo anterior, no hay ningún requisito en la distribución de los puntos.

Para los datos estructurados, se podrán utilizar métodos de interpolación estructurada o no estructurada. Para el caso de los no estructurados, solamente se podrán utilizar métodos no estructurados. En general, es recomendado utilizar métodos estructurados, ya que al organizarse los datos en una red estructurada la conectividad entre puntos es trivial y los algoritmos de cálculos presentan menor coste computacional.

Estructurada

Cuando se disponen de datos organizados en una red estructurada N-dimensional, se puede asignar un índice a cada dimensión espacial y de forma que los datos forman un

tensor de N dimensiones. Otra opción más conveniente, cuando se trata de implementar en un lenguaje de programación, es organizar los datos como un vector 1D y acceder a cada elemento mediante un vector de N dimensiones, que representa la posición en el tensor N-D. Se puede definir una función y su inversa que relacionen de forma unívoca el vector 1D de índices del tensor N-D y un índice en el vector 1D de datos. Se define:

$$i = \boldsymbol{I}_0 + \sum_{j=1}^{j=ND-1} \left[\left(\prod_{k=0}^{k=j-1} N_k \right) \boldsymbol{I}_j \right] \tag{2.51}$$

Podemos definir:

$$P_j = \prod_{k=0}^{k=j-1} N_k \tag{2.52}$$

$$P_0 = 1 \tag{2.53}$$

de esta forma:

$$i(\boldsymbol{I}) = \boldsymbol{I}_0 + \sum_{j=1}^{j=ND-1} (P_j \, \boldsymbol{I}_j) \tag{2.54}$$

como la función que relaciona un vector 1D de índices $\boldsymbol{I}_0$, que representa los índices de un tensor N-D, con un índice i de un vector 1D. El indice $I_j \in [0, 1, \ldots, N_j - 2, N_j - 1]$, siendo N_j el número de puntos en la dimension j. La función inversa $\boldsymbol{I}(i)$, que permite el paso contrario, se define como:

$$I_j(i) = \left\lfloor \frac{i - \sum_{k=j+1}^{k=ND-1} (P_k \, \boldsymbol{I}_k)}{P_j} \right\rfloor \tag{2.55}$$

Siendo $\lfloor x \rfloor$ la función suelo(x), que devuelve la parte entera de un número real. Se puede observar que la Ecuación 2.55, requiere conocer todos los índices por encima del que se busca. Por ello se obtiene en orden inverso. Para obtener el último índice, la ecuación se simplifica:

$$I_{ND-1} = \left\lfloor \frac{i}{P_{ND-1}} \right\rfloor \tag{2.56}$$

Existe una extensa variedad de métodos de interpolación estructurada: linear, cúbica, *spline*, superficie de Bézier o Lanczos *resampling* entre otras. La más adecuada para su implementación es la linear, ya que presenta una serie de propiedades:

- Presenta bajo coste computacional.
- No requiere almacenar coeficientes adicionales.
- Permite extenderse a N-dimensiones.
- Los coeficientes de interpolación solo dependen de la localización de los puntos.

La última propiedad permite expresar la interpolación de la siguiente forma:

$$a(\boldsymbol{x}) = \sum_j a_j \cdot c_j(\boldsymbol{x}) \tag{2.57}$$

Donde a es el valor interpolado, obtenido como una suma de los valores conocidos a_j y unos coeficientes c_j, que solo dependen de la posición de los valores conocidos, pero no de

los propios valores. Esto permite generalizar la interpolación, de forma que los valores conocidos a_j, no requieren ser escalares, pudiendo ser vectores, matrices o cualquier tipo de objetos. Pero la utilidad se acentúa a nivel computacional, ya que permite interpolar clases, como polares, secciones o cualquier tipo de objeto.

Interpolación Linear

La interpolación linear se base en ajustar una recta a dos puntos consecutivos para obtener el valor intermedio mediante la ecuación de la recta. Para el caso 1D se puede expresar de la siguiente forma:

$$a(x) = a_j \cdot (1 - c(x)) + a_{j+1} \cdot c(x) \tag{2.58}$$

$$c(x) = \frac{x - x_j}{x_{j+1} - x_j} \quad \text{si} \quad \{x_j <= x < x_{j+1}\} \tag{2.59}$$

El índice j se selecciona de forma que x queda contenida en el intervalo $[x_j, x_{j+1})$.

Para extender a N-dimensiones el procedimiento es directo. Se realiza la interpolación en cada dimensión, y se continua en la siguiente con los valores obtenidos. Para el caso 2D, se dispone de 2 puntos en dirección x y 2 puntos en dirección y, formado un rectángulo en $\mathcal{R}^2$:$[\boldsymbol{x}_{i,j}, \boldsymbol{x}_{i+1,j}, \boldsymbol{x}_{i,j+1}, \boldsymbol{x}_{i+1,j+1}]$. En primer lugar se define el coeficiente $c_{i,j}$ para 2 dimensiones, donde el superíndice k indica la dimensión:

$$c^k_{i,j}(\boldsymbol{x}) = \frac{\boldsymbol{x}^k - \boldsymbol{x}^k_{i,j}}{\boldsymbol{x}^k_{i+1,j+1} - \boldsymbol{x}^k_{i,j}} \tag{2.60}$$

A continuación se realiza la interpolación en dirección x

$$a_j(\boldsymbol{x}) = a_{i,j} \cdot (1 - c^x_{i,j}) + a_{i+1,j} \cdot c^x_{i,j} \tag{2.61}$$

$$a_{j+1}(\boldsymbol{x}) = a_{i,j+1} \cdot (1 - c^x_{i,j}) + a_{i+1,j+1} \cdot c^x_{i,j} \tag{2.62}$$

Finalmente se interpola en dirección y con los valores anteriores:

$$a(\boldsymbol{x}) = a_j(\boldsymbol{x}) \cdot (1 - c^y_{i,j}) + a_{j+1}(x) \cdot c^y_{i,j} \tag{2.63}$$

Sustituyendo la Ecuación 2.61 y la Ecuación 2.62 en la Ecuación 2.63:

$$a(\boldsymbol{x}) = [a_{i,j} \cdot (1 - c^x_{i,j}) + a_{i+1,j} \cdot c^x_{i,j}] \cdot (1 - c^y_{i,j}) + [a_{i,j+1} \cdot (1 - c^x_{i,j}) + a_{i+1,j+1} \cdot c^x_{i,j}] \cdot c^y_{i,j} \tag{2.64}$$

Aplicando la propiedad distributiva:

$$\begin{aligned} a(\boldsymbol{x}) = a_{i,j} \cdot (1 - c^x_{i,j}) \cdot (1 - c^y_{i,j}) + a_{i+1,j} \cdot c^x_{i,j} \cdot (1 - c^y_{i,j}) \\ + a_{i,j+1} \cdot (1 - c^x_{i,j}) \cdot c^y_{i,j} + a_{i+1,j+1} \cdot c^x_{i,j} \cdot c^y_{i,j} \end{aligned} \tag{2.65}$$

Se puede observar que la función de interpolación puede expresarse como un sumatorio de los valores por unos coeficientes según la Ecuación 2.57. Reorganizando los valores en una lista unidimensional puede extenderse la expresión a N dimensiones. En primer lugar

se generaliza la definición del coeficiente c, donde los índices i y j, ahora representan los índices en la lista 1D según la Ecuación 2.54:

$$c_{i,j}^k(\boldsymbol{x}) = \frac{\boldsymbol{x}^k - \boldsymbol{x}_i^k}{\boldsymbol{x}_j^k - \boldsymbol{x}_i^k}; \quad x_j^k > x_i^k \tag{2.66}$$

Para expresar los coeficientes, definimos por simplicidad:

$${}^0c_{i,j}^k = 1 - c_{i,j}^k \tag{2.67}$$

$${}^1c_{i,j}^k = c_{i,j}^k \tag{2.68}$$

A continuación se obtienen los índices en cada dimensión de los puntos que delimitan la región en la que está contenido el punto de interpolación $\boldsymbol{x}$ (segmento en 1D, rectángulo en 2D, paralepípedo en 3D ...) y se almacenan en dos listas de longitud ND:

$$\boldsymbol{i} = [i^0, i^1, \ldots, i^{ND-2}, i^{ND-1}] \tag{2.69}$$

$$\boldsymbol{j} = [j^0, j^1, \ldots, j^{ND-2}, j^{ND-1}] \tag{2.70}$$

También se almacenan los valores de los puntos que forman los extremos, formados por la permutación de ambas listas, de longitud 2^{ND}:

$$a_j = [a_0, a_1, \ldots, a_{2^{ND}-2}, a_{2^{ND}-1}] \tag{2.71}$$

Por tanto para obtener los coeficientes generalizados, se multiplican los correspondientes a cada dimensión:

$$c_j = \prod_{n=0}^{n=ND-1} c_{\boldsymbol{i}_n,\boldsymbol{j}_n}^{\boldsymbol{I}^n(j)} \tag{2.72}$$

Siendo $\boldsymbol{I}^n(j)$ la componente de la dimensión n del vector N-D para la posición m según la Ecuación 2.55 Finalmente la interpolación se obtiene multiplicado los coeficientes de cada dimensión.

$$a(x) = \sum_{j=0}^{2^{ND}-1} (a_j \cdot c_j) \tag{2.73}$$

Interpolación *spline* cúbica

El método de interpolación mediante *spline* cúbica se basa en ajustar polinomios de tercer grado a los puntos conocidos. Este método solo se ha desarrollado para datos 1D. Para un conjunto de N puntos se utilizan $N-1$ polinomios cúbicos. Para el punto $i(x_i, y_i)$, se define el polinomio que pasa por ese punto como:

$$P_i(x) = y_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3, \quad si \quad x_i \leq x < x_{i+1} \tag{2.74}$$

Que tiene que satisfacer las siguientes propiedades de continuidad $\mathcal{C}'$,$\mathcal{C}^\infty$ y $\mathcal{C}^\in$:

$$P_{i-1}(x_i) = P_i(x_i) = y_i \quad para \quad i = 1, 2, \ldots, N-1 \tag{2.75}$$

$$P'_{i-1}(x_i) = P'_i(x_i) \quad para \quad i = 1, 2, \ldots, N-2 \tag{2.76}$$

$$P''_{i-1}(x_i) = P''_i(x_i) \quad para \quad i = 1, 2, \ldots, N-2 \tag{2.77}$$

Esto asegura que la curva pasa por todos los puntos, no presenta esquinas y la curvatura es suave. Se tienen 3 incógnitas (b_i, c_i, d_i) por cada polinomio, siendo un total de $3N-3$. La condición 1 se aplica a los $N-1$ puntos, y las otras dos a los $N-2$ puntos fronteras dando un total de $3N-5$ ecuaciones. Es necesario, por tanto, definir 2 ecuaciones adicionales para cerrar el problema [22].

Se consideran 3 tipos diferentes de *spline* cubica en función de las ecuaciones de cierre:

- Natural *spline*: $P''_0(x_0) = P''_{N-1}(x_{N-1}) = 0$
- Clamped *spline*: $P'_0(x_0) = y'_0$ y $P'_{N-1}(x_{N-1}) = y'_{N-1}$
- Not-a-knot *spline*: $P'''_0(x_0) = P'''_1(x_0)$ y $P'''_{N-2}(x_{N-1}) = P'''_{N-1}(x_{N-1})$

No estructurada

Cuando los datos utilizados no presentan una organización, solo se pueden utilizar métodos de interpolación no estructurada, que suelen presentar mayor coste computacional. En el presente trabajo nos centramos en dos métodos de bajo orden: *closest neightbor* y *inverse distance weighting*.

Closest Neightbor

Este método es el más sencillo de implementar, pero presenta mayor dificultad de optimizar. Este consiste en encontrar el punto conocido más cercano al punto de interpolación, de forma que el punto conocido x_i ha de satisfacer:

$$\|\boldsymbol{x} - \boldsymbol{x_i}\| <= \|\boldsymbol{x} - \boldsymbol{x_j}\| \tag{2.78}$$

Para cualquier x_j, siendo el valor interpolado igual al valor de ese punto:

$$y = y_i \tag{2.79}$$

La implementación más sencilla de este método consiste en obtener la distancia del punto de interpolación a cada uno de los datos conocidos, escogiendo el que la distancia sea mínima. Otra implementación esta basada en realizar una descomposición binaria del dominio, reduciendo así el número de normas calculadas. La implementación de este método, aunque más eficaz, no se ha realizado, puesto que se buscaba implementar métodos N-dimensionales, por lo que la complejidad de implementarlo de esta forma queda fuera del ámbito de este trabajo.

Inverse Distance Weighting

Este método también es sencillo de implementar y caro en términos computacionales. Para realizarlo se obtienen en primer lugar los M puntos más cercanos al punto de interpolación [24]. La distancia entre dos puntos en N-d se define por:

$$\mathrm{d}(\boldsymbol{a}, \boldsymbol{b}) = \|\boldsymbol{a} - \boldsymbol{b}\|^{\frac{1}{2}} \tag{2.80}$$

Quedando la función de interpolación definida como:

$$y(\boldsymbol{x}) = \frac{\sum_{i=0}^{M-1} w_i(\boldsymbol{x})\, y_i}{\sum_{i=0}^{N-1} w_i} \tag{2.81}$$

Siendo w_i los pesos de los M puntos más cercanos, obtenidos como:

$$w_i(\boldsymbol{x}) = \frac{1}{d(\boldsymbol{x}, \boldsymbol{x_i})^p} \tag{2.82}$$

Para valores de $p <=$ N-d el valor de la interpolación puede divergir. De la misma forma que el método *closest neightbor*, este también puede optimizarse mediante partición del dominio.

2.3.2. Newton-Raphson multidimensional

El método numérico de Newton-Raphson es un algoritmo basado en gradiente utilizado para encontrar los ceros de una función real mediante aproximaciones sucesivas en serie de Taylor de la función [21]. Por tanto, la función ha de ser diferenciable.
Dada una función diferenciable de una variable $f : \mathcal{R}^1 \to \mathcal{R}^1$, queremos encontrar el valor x_1 que sea solución de la ecuación:

$$f(x_1) = 0 \tag{2.83}$$

Se desarrolla en serie de Taylor alrededor de x:

$$f(x + \delta x) \approx f(x) + f'(x)\, \delta x = 0 \tag{2.84}$$

Despejando δx:

$$\delta x = -\frac{f(x)}{f'(x)} \tag{2.85}$$

Esto se puede extender a N dimensiones, permitiendo resolver sistemas de N ecuaciones con N incógnitas. Sea $f : \mathcal{R}^n \to \mathcal{R}^n$ una función cuyas derivadas parciales de primer orden existen en todo $\mathcal{R}^n$:

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0} \tag{2.86}$$

Desarrollando en serie de Taylor alrededor de $\boldsymbol{x}$:

$$\boldsymbol{f}(\boldsymbol{x} + \boldsymbol{\delta x}) \approx \boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{J}(\boldsymbol{x})\, \boldsymbol{\delta x} = \boldsymbol{0} \tag{2.87}$$

$$\boldsymbol{\delta x} = -\boldsymbol{J}^{-1}(\boldsymbol{x})\, \boldsymbol{f}(\boldsymbol{x}) \tag{2.88}$$

Para resolverlo numéricamente se actualiza el nuevo valor de $\boldsymbol{x}$ mediante el incremento obtenido de la Ecuación 2.88, multiplicado por un factor de relajación $\alpha \in (0\,,\, 1]$ que permite volver más estable el calculo a costa de aumentar el número de iteraciones. Expresándolo mediante una ecuación iterativa:

$$\boldsymbol{x_{k+1}} = \boldsymbol{x_k} + \alpha\, \boldsymbol{\delta x_k} = \boldsymbol{x_k} + \alpha\, \boldsymbol{J}^{-1}(\boldsymbol{x_k})\, \boldsymbol{f}(\boldsymbol{x_k}) \tag{2.89}$$

En la práctica, no se resuelve la inversa de la matriz Jacobiana, ya que el coste es elevado, lo mas habitual es resolver el siguiente sistema mediante eliminación de gauss o descomposición LU:

$$\boldsymbol{J}(\boldsymbol{x})\, \boldsymbol{\delta x} = -\boldsymbol{f}(\boldsymbol{x}) \tag{2.90}$$

Algoritmo

El algoritmo utilizado para la implementación de este método es el siguiente:

1. Se define una tolerancia y un número de iteraciones máximo
2. Se parte de una estimación inicial de la solución $\boldsymbol{x_0}$.
3. Si el error es inferior a la tolerancia o las iteraciones superiores a las máximas, se finaliza.
4. Se calcula el Jacobiano de la función en la estimación actual mediante algún método numérico.
5. Se obtiene la nueva estimación de la solución mediante la Ecuación 2.89
6. Se regresa al Elemento 3

Para la aplicación utilizada, dado que las funciones a resolver son lo suficientemente suaves, para el cálculo numérico del Jacobiano se ha utilizado el método de diferencias finitas. Para ello se escoge un pequeño incremento de las variables $\boldsymbol{\Delta x}$, y se calcula el Jacobiano mediante la siguiente expresión:

$$\boldsymbol{\Delta x_j} = (0 \,..\, \Delta x_j \,..\, 0) \tag{2.91}$$

$$J_{ij}(\boldsymbol{x}) = \frac{\partial f_i(\boldsymbol{x})}{\partial x_j} \approx \frac{f_i(\boldsymbol{x} + \boldsymbol{\Delta x_j}) - f_i(\boldsymbol{x} - \boldsymbol{\Delta x_j})}{2\Delta x_j} \tag{2.92}$$

Siendo $\boldsymbol{\Delta x_j}$ un vector de ceros, excepto el valor Δx_j en la componente j, J_{ij} las componentes escalares de la matriz Jacobiana, f_i las componentes escalares de $\boldsymbol{f}$ y Δx_j las componentes escalares de $\boldsymbol{\Delta x}$

2.3.3. Área de un polígono

En mallas formadas por polígonos en el cálculo de la BET, es necesario calcular el área de estos para la integración de las cargas aerodinámicas. Los polígonos son, en general, irregulares y de un número arbitrario de vértices, por lo que es necesario utilizar un método que permita calcular el área de cualquier polígono. Se presenta como solución el método del determinante de gauss o *shoelace method* [1]. Dado un conjunto de N puntos en el plano ordenados en sentido contrario a las agujas del reloj $(x_0, x_1), \ldots, (x_{N-1}, y_{N-1})$:

$$A_P = \frac{1}{2}\left(\begin{vmatrix} x_0 & x_1 \\ y_0 & y_1 \end{vmatrix} + \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} + \ldots + \begin{vmatrix} x_{N-1} & x_0 \\ y_{N-1} & y_0 \end{vmatrix}\right) \tag{2.93}$$

Si los puntos están ordenados en sentido contrario, el valor del área será negativo. En el caso de que los puntos no estén ordenados, el valor del área calculado será incorrecto. Para tener la certeza de que la ordenación es correcta se implementa un método de ordenación definido en la Subsección 2.4.1.

2.4. Algoritmos

Los algoritmos son secuencias de instrucciones ejecutadas en un orden concreto para realizar una actividad relativamente compleja. La diferencia con los métodos numéricos es que los últimos resuelven problemas matemáticos de forma numérica. Los algoritmos no están limitados a ese ámbito, y pueden abarcar temáticas tan distintas como: los pasos para realizar un café o la secuencia de lanzamiento de un cohete.

En la presente sección, se describen los algoritmos más relevantes desarrollados para la implementación de la BET. La mayoría de los que han sido desarrollados, han sido con el objetivo de la creación de la malla y el acoplamiento FVM-BET, como son la ordenación de vértices, determinar si un punto esta en el interior de un polígono convexo, la triangulación de Delaunay en conjunto con el diagrama de Voronoi, y la intersección de mallas.

2.4.1. Ordenación de vértices

En algunos métodos numéricos y algoritmos es necesario presentar los vértices de un polígono ordenados en sentido contrario a las agujas del reloj (o en el mismo sentido), para ello se ha implementado un método básico que permite ordenar los vértices de un polígono **convexo** dado.

Se parte de una lista desordenada de N puntos (x_0, y_0), ... (x_{N-1}, y_{N-1}) , y el objetivo es encontrar una lista de índices $\boldsymbol{i}$, que aplicando la transformación:

$$x_j = x_{i(j)} \tag{2.94}$$

$$y_j = y_{i(j)} \tag{2.95}$$

Es decir, que el punto en la posición j de la lista ya ordenada es el punto en la posición $i(j)$, en la Figura 2.16 se observa un ejemplo de ordenación de una lista de escalares, mediante la creación de una lista intermedia de índices.

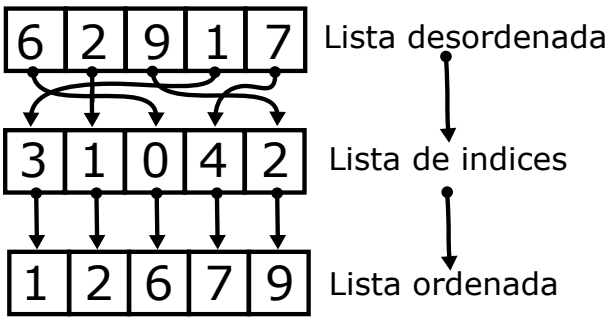

Figura 2.16: Lista de índices para ordenación de lista

El algoritmo para ordenar los vértices es el siguiente:

1. Se busca el centro del polígono $\boldsymbol{C} = \frac{1}{N}\sum_{i=0}^{i=N-1} \boldsymbol{P_i}$
2. Se calcula el ángulo φ de cada punto con: $\varphi_i = \text{atan2}(\boldsymbol{C_x} - \boldsymbol{P_{i,x}}, \boldsymbol{C_y} - \boldsymbol{P_{i,y}})$ con $\varphi_i \in (-\pi, \pi]$

3. Se crea una lista de tuplas con los ángulos de la forma (φ_i, i) y se ordena en sentido creciente de φ_i
4. La lista formada por el segundo elemento de las tuplas es la lista de índices de ordenación
5. Se reordenan los puntos según la Ecuación 2.94 y la Ecuación 2.95

En la Figura 2.17, se observa como resultaría un hexágono convexo irregular tras la ordenación de los vértices, y como se sitúa el sistema de referencia centrado en el polígono para el cálculo del ángulo.

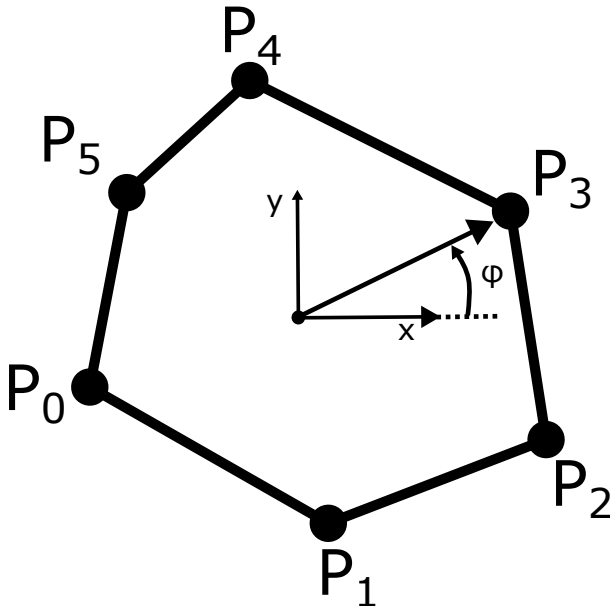

Figura 2.17: Sistema de referencia y ordenación de vértices

2.4.2. Polígono convexo

Para algunos de los algoritmos geométricos utilizados, es necesario que los polígonos cumplan con la condición de convexidad. Se define como polígono convexo, aquel polígono que todos los ángulos internos son inferiores a 180°. Para comprobar la convexidad de un polígono, se utiliza el hecho de que el $\sin(\varphi)$ es positivo para $\varphi \in (0°, 180°)$ y negativo para $\varphi \in (180°, 360°)$. Por lo que para computar si un polígono es convexo, basta con comprobar que el seno de cada ángulo interno es positivo. Para computarlo de forma más eficiente se utiliza el producto vectorial, donde el módulo (con signo) se puede obtener con la expresión:

$$|\boldsymbol{a} \times \boldsymbol{b}| = |\boldsymbol{a}| \cdot |\boldsymbol{b}| \cdot \sin(\varphi) \tag{2.96}$$

Siendo $\boldsymbol{a}$ y $\boldsymbol{b}$ dos vectores en $\mathcal{R}^3$ arbitrarios, y φ el ángulo que forman. Como el polígono esta contenido en el plano z , el producto vectorial de dos vectores formados por los lados del polígono solo tendrá componente z. Por tanto el módulo (con signo) se reduce a:

$$|\boldsymbol{a} \times \boldsymbol{b}| = \boldsymbol{a_x b_y} - \boldsymbol{a_y b_x} \tag{2.97}$$

Los pasos a seguir para su implementación, partiendo de un polígono con sus vértices ordenados $\boldsymbol{P_0}, \boldsymbol{P_1}, \ldots, \boldsymbol{P_{N-1}}$ (sentido horario o antihorario), son:

1. Se parte del primer punto $\boldsymbol{P_0}$ y se forman los vectores $\overrightarrow{\boldsymbol{P_0P_1}}$ y $\overrightarrow{\boldsymbol{P_1P_2}}$.
2. Se calcula el modulo (con signo) del producto escalar de $\overrightarrow{\boldsymbol{P_0P_1}}$ y $\overrightarrow{\boldsymbol{P_1P_2}}$ según la Ecuación 2.97 y se almacena el signo.
3. Se repite el paso anterior para el siguiente punto del polígono.
4. Si alguno de los signos es diferente del primero calculado, el polígono no es convexo. En caso contrario, sí es convexo.

2.4.3. Punto interior a polígono convexo

Durante la generación de la malla BET, es necesario comprobar si el centro del elemento de volúmenes finitos, proyectado sobre el plano del rotor, pertenece al interior del polígono definido por BET. Salvo fallos en la generación, los polígonos de la malla BET son convexos, por lo que puede utilizarse como hipótesis de partida al escoger el algoritmo. Para un polígono arbitrario, existen métodos como *raytracing* [7], basados en la paridad del número de intersecciones de una semi recta, originada en el punto de interés, con el polígono. El método escogido, explicado en el libro *Introduction to Algorithms* [4], es similar al usado para comprobar que un polígono es convexo. Se parte de una lista ordenada de vértices en el plano $\boldsymbol{P_0}, \boldsymbol{P_1}, \ldots, \boldsymbol{P_{N-1}}$ (sentido horario o antihorario) y el punto de interés $\boldsymbol{Q}$. Si $\boldsymbol{Q}$ es interior al polígono convexo, estará situado al mismo lado de todos los vectores formados por pares de puntos vecinos, recorridos en sentido horario o antihorario. Los pasos del algoritmo son los siguientes:

1. Se parte del primer punto $\boldsymbol{P_0}$ y se forman los vectores $\overrightarrow{\boldsymbol{P_0P_1}}$ y $\overrightarrow{\boldsymbol{P_1Q}}$.
2. Se calcula el modulo (con signo) del producto escalar de $\overrightarrow{\boldsymbol{P_0P_1}}$ y $\overrightarrow{\boldsymbol{P_1Q}}$ según la Ecuación 2.97 y se almacena el signo.
3. Se repite el paso anterior para el siguiente punto del polígono.
4. Si alguno de los signos es diferente del primero calculado, el punto $\boldsymbol{Q}$ no pertenece al polígono. En caso contrario, sí.

2.4.4. Triangulación de Delaunay

Partiendo de una lista de N puntos en el plano: $\boldsymbol{P_0}, \boldsymbol{P_1}, \ldots, \boldsymbol{P_{N-1}}$, el problema de la triangulación consiste en unir todos los puntos, formando triángulos que no se cortan entre sí. La triangulación de Delaunay, además, asegura que los triángulos formados, son lo más equiláteros posibles, lo cual es favorable, dado que el uso que se le va a dar es la generación de la malla BET. Los pasos del algoritmo descritos en [2] son los siguientes:

1. Se parte de una triangulación inicial como en la Figura 2.18. Si es el primer paso, se crea un (super)triángulo que englobe todos los puntos.
2. Se añade un punto adicional $\boldsymbol{P_i}$ a la triangulación, y se seleccionan todos los triángulos que su circuncírculo encierre al punto $\boldsymbol{P_i}$, como se observa en la Figura 2.19.

3. Se eliminan todos los triángulos seleccionados y se forman nuevos triángulos uniendo el punto $\boldsymbol{P_i}$ y los puntos del polígono que lo encierra como en la Figura 2.20.

4. Se repiten los pasos anteriores hasta que se han añadido todos los puntos $\boldsymbol{P_i}$.

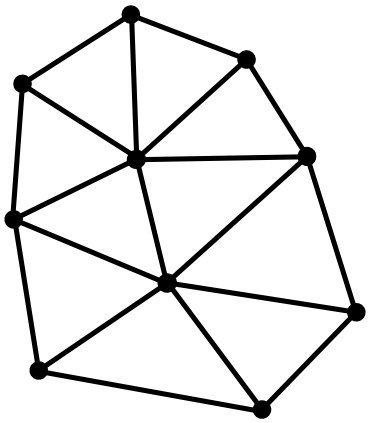

Figura 2.18: Triangulación base

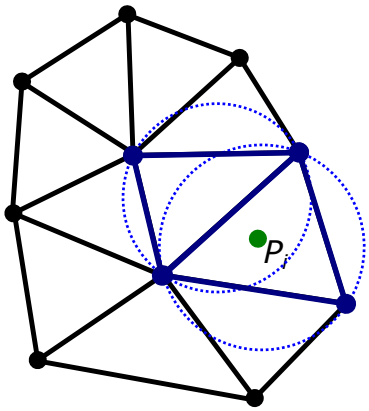

Figura 2.19: Punto a añadir y circuncírculos que lo engloban

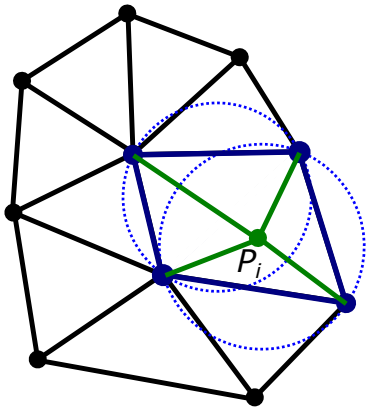

Figura 2.20: Nueva triangulación tras añadir el punto

La implementación del algoritmo en el código se ha realizado partiendo de una implementación en C [2], reescribiéndola en C++ haciendo uso de la librería estándar y evitando

llamadas expuestas a `malloc` y `free`, sustituyendo los arrays de C por `std::vector`. También se ha añadido la opción de mantener el (super)triángulo y de que este englobe también una forma arbitraria, de utilidad para realizar el diagrama de Voronoi a partir de este.

2.4.5. Diagrama de Voronoi

El diagrama de Voronoi de un conjunto de puntos se define como la división del espacio en regiones, donde cada región está más cerca del punto que contiene que de cualquier otro punto del conjunto inicial. Estos diagramas son clave en geometría computacional, y tienen gran cantidad de aplicaciones: en la repartición de áreas en un mapa, en química, robótica o arquitectura. Ademas surge espontáneamente en el mundo natural, como en: los cultivos de bacterias, la superficie del sol, las grietas de sequía en la tierra o las manchas de una jirafa. Se ha observado una posible utilidad del uso de estos diagramas para la generación de mallas BET a partir de un conjunto de puntos de la malla FV. Hay diversos métodos para generar el diagrama de Voronoi: por semiplanos, incremental, algoritmo de Fortune o el escogido en el trabajo, como complementario a la triangulación de Delaunay.

Para construirlo a partir de la triangulación de Delaunay, se siguen los siguientes pasos:

1. Se parte de una lista de vértices y otra lista de triángulos, donde cada triángulo está formado por 3 índices de la lista de vértices, proporcionado por la triangulación de Delaunay. El diagrama de Voronoi se construye como una lista de vértices y una lista de polígonos, donde cada polígono es una lista de índices que referencia la lista de vértices.

2. Se recorren todos los triángulos de la triangulación de Delaunay y se crea una lista con los orto-centros de cada uno, que serán los vértices de los polígonos del diagrama de Voronoi.

3. Los índices que forman los polígonos del diagrama de Voronoi, serán los índices de los triángulos que compartan el mismo vértice, ya que los índices de los triángulos coinciden con los de sus orto-centros.

Aunque el diagrama de Voronoi puede no estar acotado, para el uso que se le da, siempre va a estar acotado, y todos los elementos van a ser polígonos. Para conseguir esto, se mantiene el supertriangulo de la triangulación de Delaunay, siendo el límite del diagrama los lados del triángulo. En la Figura 2.21 se muestra la complementariedad del diagrama de Voronoi y la triangulación de Delaunay. Los puntos iniciales de los cuales se busca obtener el diagrama de Voronoi, están representados en verde. Estos conforman los vértices de la triangulación de Delaunay (líneas verdes discontinuas). Finalmente, los orto-centros de los triángulos (puntos azules) determinan los vértices de los polígonos que conforman el diagrama de Voronoi, representado en azul.

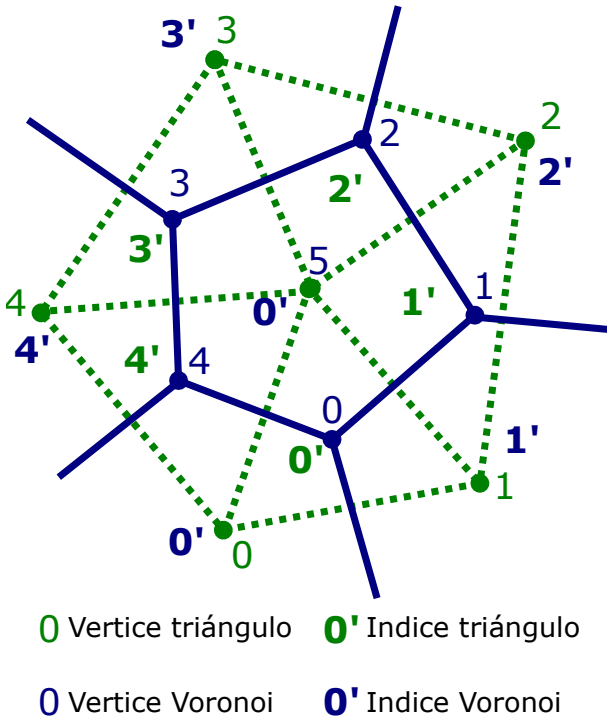

Figura 2.21: Diagrama de Voronoi(azul) a partir de la triangulación de Delaunay (verde).

2.4.6. Intersección malla BET - disco

Para el cálculo de la BET promediada en el tiempo es necesario que la malla utilizada tenga una envolvente circular, o lo más parecido posible. Dependiendo del tipo de mallado utilizado, esto se cumplirá de forma automática. Para los casos en los que no es así, se propone un método para crear una intersección entre la malla y una circunferencia para así mejorar la precisión en este tipo de mallas. Los pasos del método propuesto son los siguientes:

1. Se recorren las celdas de la malla BET.
2. Para cada celda, se comprueba si hay algún vértice dentro del círculo. Si no hay ninguno, se elimina la celda.
3. En caso de que algún vértice, pero no todos, sea interno al círculo se busca el primer vértice i externo al círculo.
4. A partir del índice i, se recorren los puntos en sentido horario y anti horario, hasta encontrar los índices j y k de forma que los lados que unen los vértices j - $(j+1)$ y k - $(k+1)$ cortan el círculo en un punto.
5. Se añaden los puntos de intersección y se eliminan los externos al círculo.

En la Figura 2.22 se muestra el resultado de la intersección de una celda. En rojo se muestran los lados que quedan externos al círculo, y por tanto eliminados; en azul se muestra la circunferencia que se utiliza para intersectar; en verde el nuevo lado y vértices creados; y finalmente, en negro, la parte del polígono que no ha sido modificado.

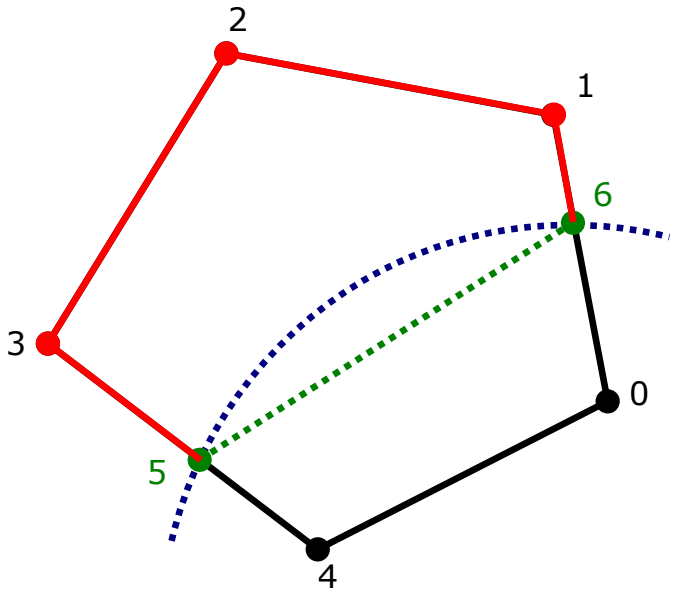

Figura 2.22: Intersección entre una celda BET (negro) y una circunferencia (azul). En verde se muestra el nuevo lado creado y en rojo los eliminados.

2.4.7. Refinamiento borde de malla BET

En los casos en los que la malla no es muy fina, puede que con la intersección se subestime el área total del disco, para ello se propone a continuación un método para refinar los bordes que presentan intersección con el disco. El método propuesto se utiliza en conjunto con el método de la intersección, tras encontrar los puntos que intersecan con el círculo. Los pasos son los siguientes:

1. Se obtiene el punto medio de los dos puntos de intersección.
2. Se traza un radio que pasa por ese punto y interseca el círculo.
3. Se encuentra el nuevo punto de intersección y se une a los puntos vecinos.
4. Se repiten los pasos 1,2 y 3 con el nuevo punto y los dos puntos vecinos. Se repite hasta conseguir el numero de refinamiento requerido.

Se observa que para cada par de puntos vecinos, aparece un nuevo punto intermedio, que duplica el número de puntos vecinos. Por cada iteración del algoritmo, el número de puntos nuevos añadidos en cada iteración crece de forma exponencial , según la expresión:

$$N_p = 2^N \tag{2.98}$$

Donde N_p es el número de puntos nuevos y N el número de iteraciones, siendo $N = 0$ la primera iteración. En la Figura 2.23 se observa la primera iteración del algoritmo, añadiendo un punto central de refinamiento. En la Figura 2.24 se observa la segunda iteración, añadiendo 2 puntos nuevos, donde se puede apreciar que con pocas iteraciones de refinamiento se puede conseguir una aproximación mucho mejor a la geometría original.

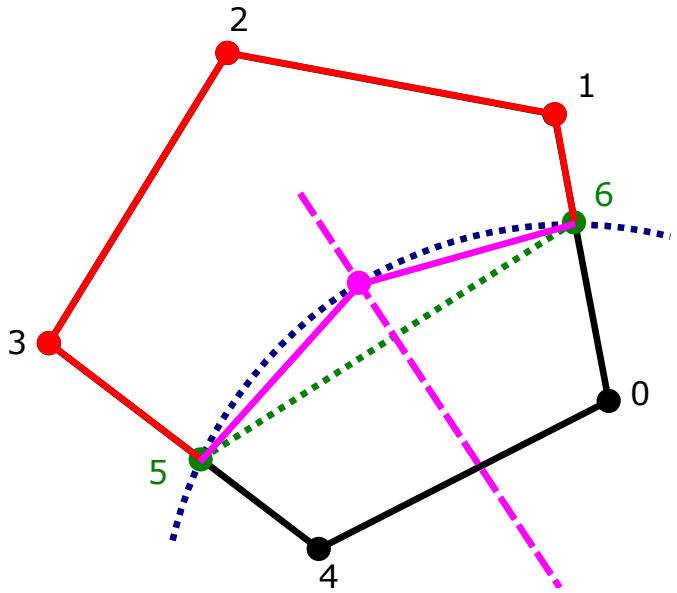

Figura 2.23: Primera iteración del algoritmo de refinado. En rosa los puntos y vértices nuevos

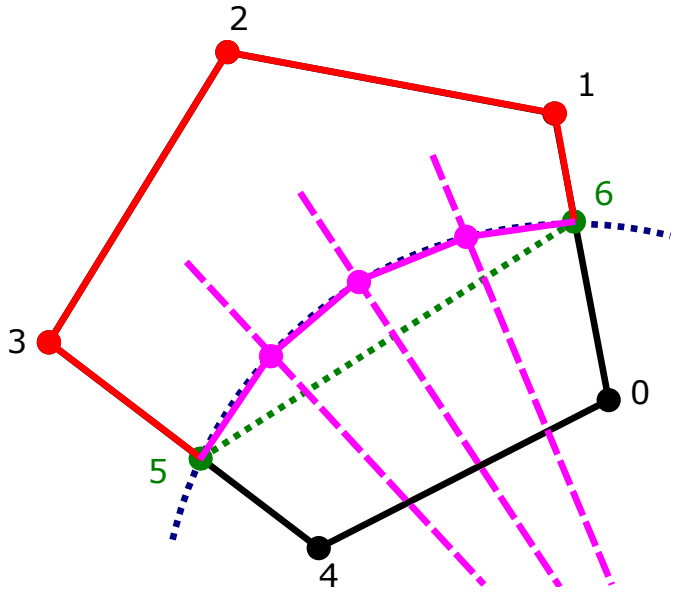

Figura 2.24: Segunda iteración del algoritmo de refinado. En rosa los nuevos lados y vértices añadidos

2.4.8. Intersección malla FV - disco

Uno de los mallados propuestos y implementados se basa en la intersección de una malla de volúmenes finitos con un disco 2D. Para ello es necesario desarrollar un algoritmo optimizado que permita obtener esta intersección sin un coste prohibitivo. La propuesta implementada en este trabajo es la siguiente:

1. Se parte de un subconjunto de la malla, cuyas celdas son potencialmente intersecadas por el disco del rotor.
2. Por cada celda de FV se crea una de BET.
3. Se recorren todas las aristas de la celda FV y se obtiene la intersección entre la arista y el plano del rotor.
4. Las celdas BET estarán formadas por los puntos de intersección.
5. Las celdas con menos de 3 vértices son inválidas y se eliminan.

2.5. Técnicas de mallado

Para la implementación del método BET se han desarrollado varias técnicas de mallado para realizar el acople entre volúmenes finitos y BET. El principal problema de este acople es que la malla de volúmenes finitos es tridimensional y formada por elementos de forma arbitraria. Para el caso del BET, se trata de una malla bidimensional. A continuación se exponen los distintos tipos de mallado implementados para resolver esta problemática, diferenciando entre dos grupos de mallas, las promediadas en el tiempo (*time-averaged*) y las transitorias (*time-accurate*).

2.5.1. Mallado de disco (time-averaged)

El mallado para promediado en el tiempo se asume que tiene forma de disco. Esto es posible que no sea estrictamente así dado que las palas pueden tener cierto movimiento fuera del plano, como es el caso del batimiento, aunque estos efectos se desprecian en la geometría del mallado. El método *time-averaged* asume que el tiempo característico de los fenómenos que ocurren en el disco es mucho más pequeño que el paso temporal de la simulación. Esto es válido cuando se trata de simulaciones *RANS* estacionarias, ya que es adecuado por hacer un tratamiento promediado de las palas, o para situaciones en las que:

$$\tau_r << \tau_{cfd} \tag{2.99}$$

Donde τ_r es el tiempo característico del rotor y τ_{cfd} el paso temporal de la simulación CFD. El tiempo característico del rotor es del orden de magnitud del tiempo que tarda una pala en dar una vuelta completa, siendo ω la velocidad angular del rotor en radianes/s. El tiempo que tarda en completar una vuelta es:

$$\tau_c \sim \frac{2\pi}{\omega} << \tau_{cfd} \tag{2.100}$$

Reordenando y tomando ordenes de magnitud:

$$\omega \cdot \tau_{cfd} >> 1 \tag{2.101}$$

Por tanto, esta aproximación será valida cuando el paso temporal de la simulación sea elevado o la velocidad angular alta, de forma que el producto de ambas sea un numero mucho mayor a 1.

Para realizar el acople, se han diseñado dos tipos de mallas:

- El mallado de BET se realiza a partir de la malla de volúmenes finitos. Pertenecen a este tipo los métodos de: proyección, intersección y Voronoi.
- El mallado de BET se realiza independientemente al de volúmenes finitos y luego se realiza el acople. El mallado polar pertenecen a este tipo.

Realizar el mallado del BET a partir de la malla de volúmenes finitos, tiene la ventaja de que los elementos tienen una correspondencia uno a uno, es decir, cada elemento de BET está asignado recíprocamente con uno de FV. Esto permite maximizar la resolución del método, pero tiene problemas con la integración de la ecuaciones. En cambio, realizar el mallado de forma independiente y luego asignar las celdas de volúmenes finitos a la celdas de BET tiene la ventaja de que la integración de las ecuaciones resulta trivial, pero se pierde resolución en el mallado. Este fenómeno quedará mejor descrito en la explicación de los distintos métodos.

Los métodos *time-average* aplican un factor de promediado a las fuerzas calculadas mediante BET, esto es debido a que las palas reales no ocupan la totalidad del disco, si no que pasa cierto instante de tiempo en cada celda. En función del tipo de mallado hay diferentes métodos para su cálculo, que serán indicados en las respectivas secciones.

Proyección

El método de la proyección, no consiste en sí en un método de mallado, ya que no genera celdas como tal. Este es el método que viene ya implementado en OpenFOAM [19], y consiste en asignar a las celdas del BET dos parámetros, el centro del elemento y el área. El centro se obtiene como la proyección sobre el plano del disco de cada uno de los elementos que forman parte del disco del rotor. El área se obtiene como la proyección de las caras orientadas en la dirección del plano (con una tolerancia del 20 %). Para este método se define el factor de promediado de la siguiente forma:

$$a_f = \frac{N_b \cdot A_c}{\pi \ r} \tag{2.102}$$

Donde N_b es el número de palas del rotor, A_c el área de la celda y r el radio donde se sitúa el centro del elemento. El inconveniente de este método de promediado es que no es conservativo, por lo que la convergencia del método depende de utilizar una resolución elevada. Además para $r \to 0$ el valor del factor tiende a infinito. Esto en la práctica no suele ocurrir ya que se define un radio interno que no forma parte del rotor.

Intersección

En el método de la intersección, como su propio nombre indica, se interseca la malla de FV con un disco 2D, formando una celda de BET a partir de la intersección de cada celda FV con el disco del rotor. El centro de la celda puede ser la proyección del centroide de la celda de FV o el centro de la celda de BET, obtenido como la media de los vértices. El área de cada celda se obtiene a partir del área de la celda intersecada utilizando la fórmula del *shoelace* (Ecuación 2.93). El factor de promediado se obtiene también mediante la Ecuación 2.102. En la Figura 2.25 se observa el resultado de aplicar esta malla sobre una malla FV hexaédrica, mostrándose en azul el centro de las celdas. Si la intersección se realiza de forma perpendicular todos los elementos son rectangulares. En la Figura 2.26 se observa el resultado cuando el plano del rotor no está alineado con la malla. Aparecen elementos triangulares y hexagonales.

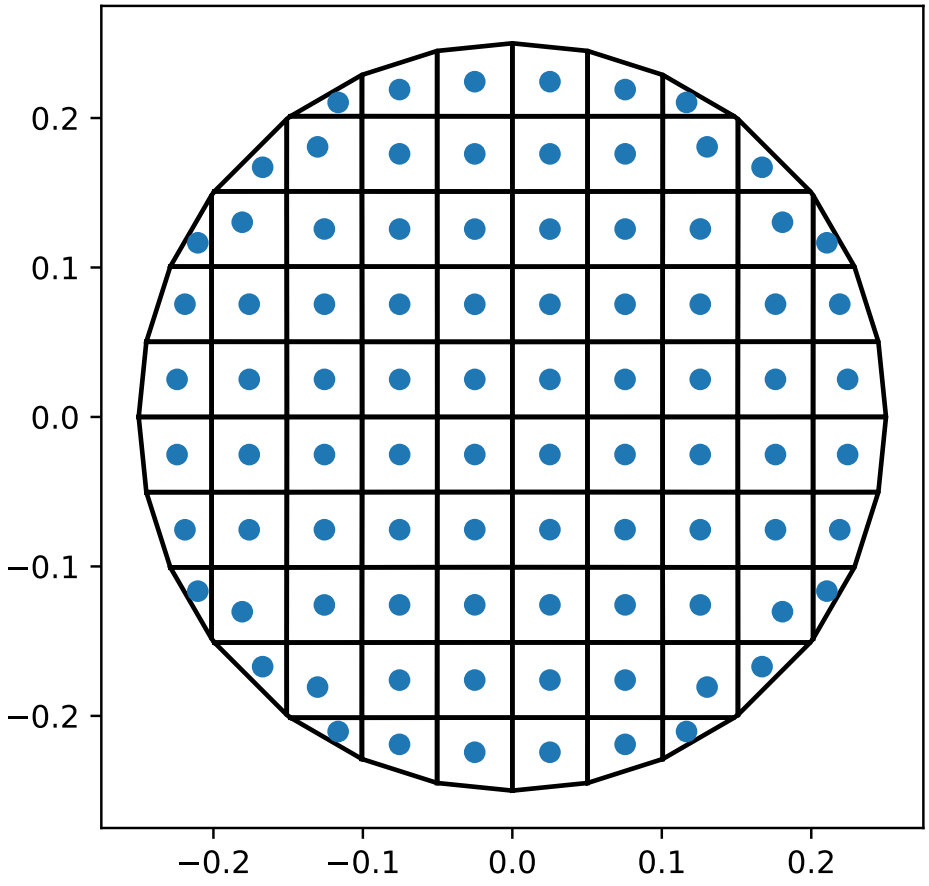

Figura 2.25: Ejemplo de mallado de intersección

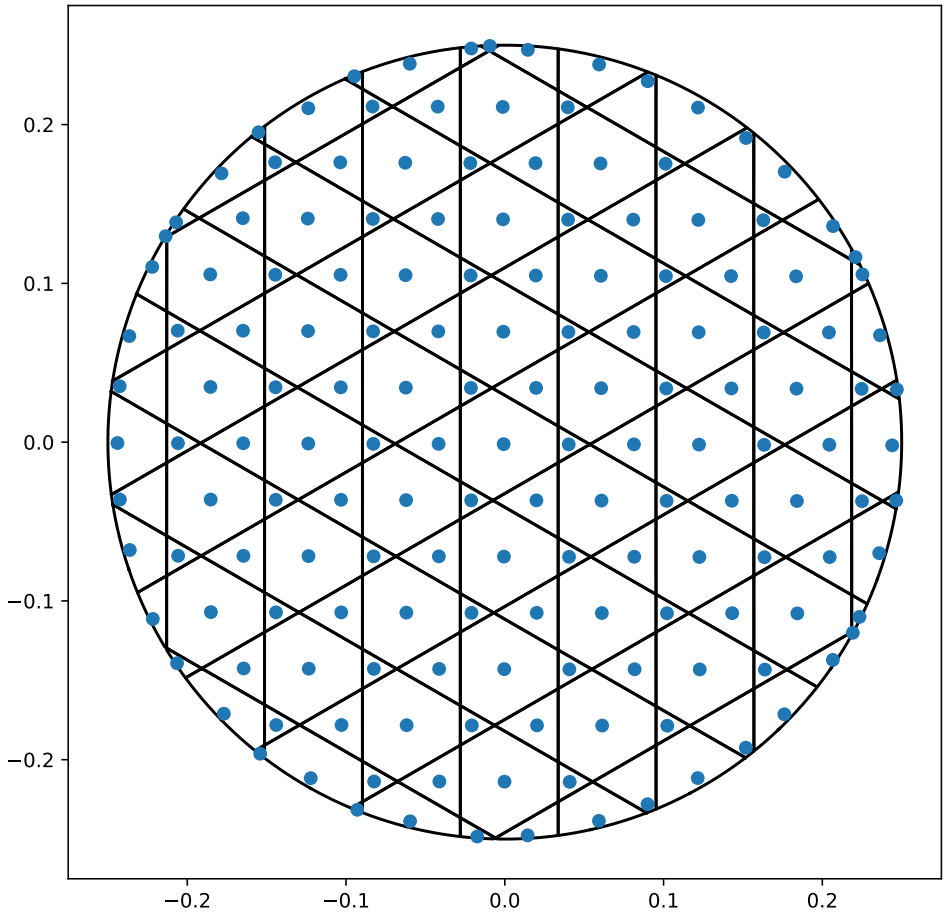

Figura 2.26: Ejemplo de mallado de intersección con disco no paralelo a la malla

Voronoi

Este método de mallado se realiza proyectando los centros de las celdas de volúmenes finitos asignadas al rotor sobre el plano del disco. Con los centros proyectados se realiza el diagrama de Voronoi con el disco como frontera. Cada celda del diagrama de Voronoi es asignada con la celda de volúmenes finitos a la cual pertenece el centroide que la conforma. Las celdas del diagrama son polígonos y su área se calcula mediante la Ecuación 2.93. El factor de promediado se calcula mediante la Ecuación 2.102. En la Figura 2.27 se observa el mallado del mismo rotor pero utilizan el diagrama de Voronoi, se observan pequeñas diferencias, sobretodo cerca de los bordes, en los que algunas celdas ocupan mayores regiones.

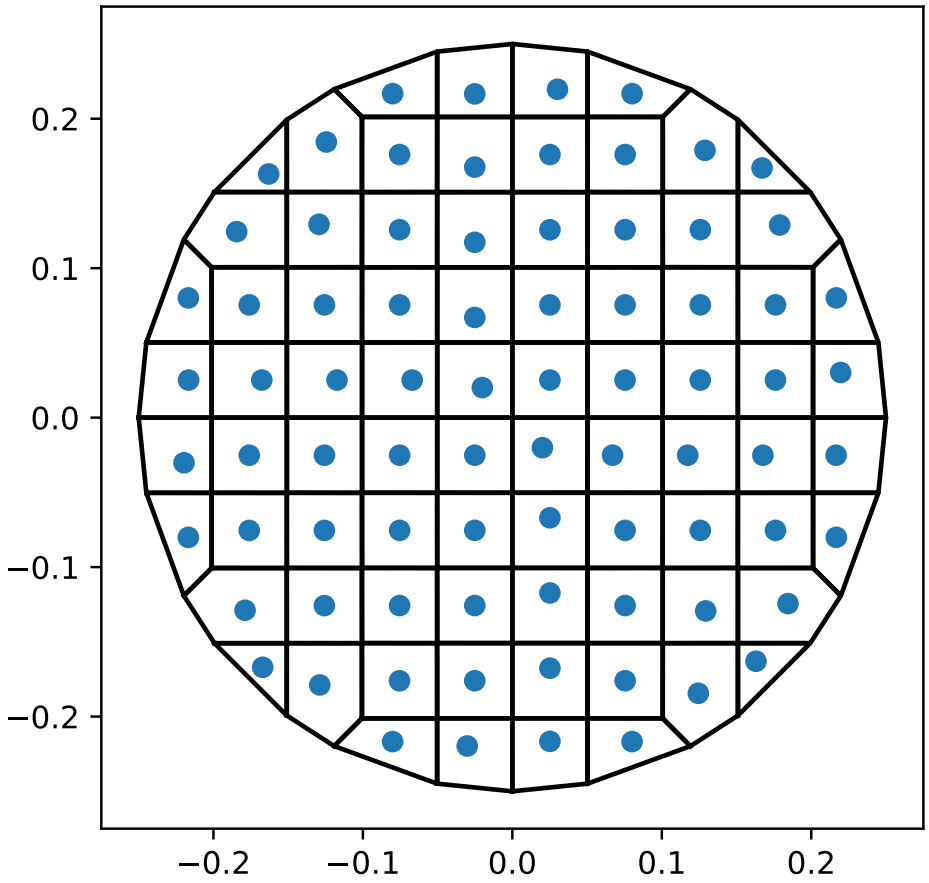

Figura 2.27: Ejemplo de mallado de Voronoi.

Polar

Para el caso de la malla polar, se crea una discretización independiente a la de volúmenes finitos. Esta se base en el uso de coordenadas polares para definir las celdas. Se parte de un disco de radio R y radio interno R_i. Se crean N_r divisiones radiales entre R_i y R y N_θ divisiones azimutales, entre 0 y 2π. La intersección entre las divisiones da lugar a los elementos de la malla polar. Si expresamos la discretización de las dos coordenadas en dos listas ordenadas: $\{R_0, R_1, \ \dots \ , R_{Nr-1}\}$ y $\{\theta_0, \theta_1, \ \dots \ , \theta_{N\theta-1,\theta_0}\}$, siendo $R_0 = R_i$ y $R_{Nr-1} = R$. Los elementos están determinados por todos los pares de coordenadas adyacentes entre si. Por tanto, el número de elementos formados es $N = N_r \cdot N_\theta$. Para asignar los elementos de volúmenes finitos a cada celda de la malla polar se expresan los centroides de los elementos en coordenadas polares, y se sitúan en las celdas que lo contengan. Este método requiere que todas las celdas de la malla polar contengan al menos una de volúmenes finitos. Si una celda de la malla polar contiene más de una de volúmenes finitos, el termino fuente correspondiente se reparte de forma proporcional al volumen de la celda de FV.

Para este tipo de malla, el factor de promediado se define de la siguiente forma:

$$a_f = \frac{N_b \cdot 2\pi}{\Delta\theta} \tag{2.103}$$

Donde $\Delta\theta$ es el incremento de la coordenada angular θ del elemento. Esta forma es

conservativa, y asegura una buena convergencia del método. En la Figura 2.28 se observa un ejemplo de esta malla para 5 elementos en dirección radial y 5 en azimutal.

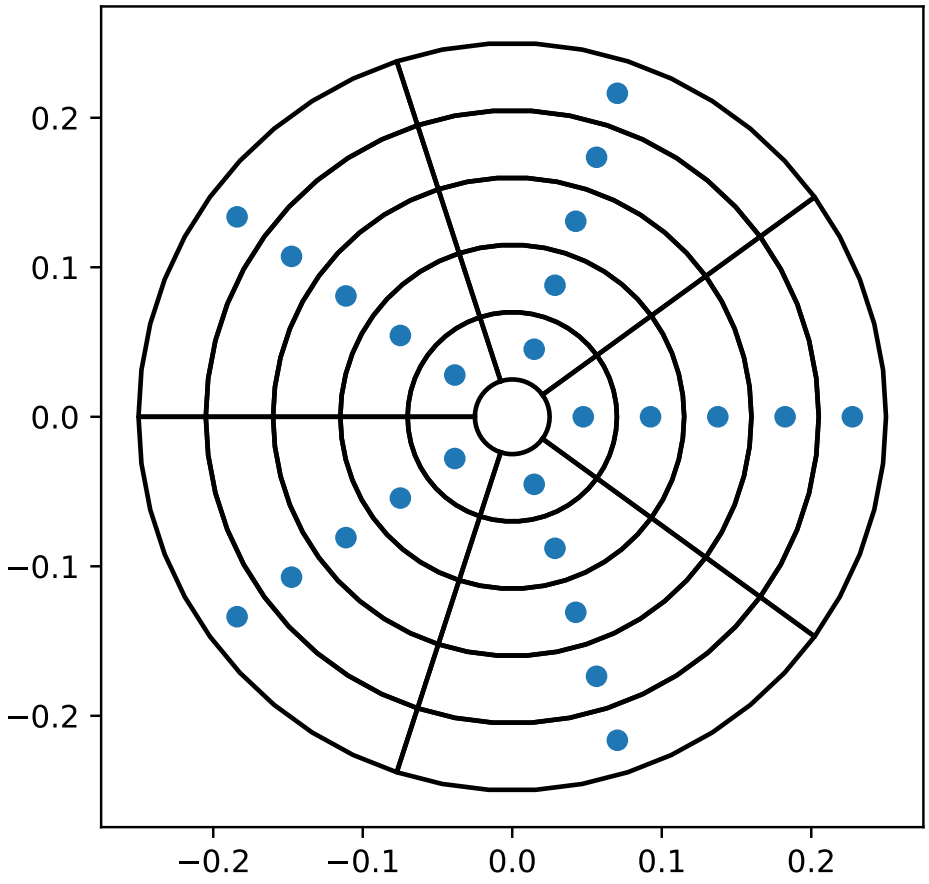

Figura 2.28: Ejemplo de mallado polar.

2.5.2. Mallado de pala (time-accurate)

El mallado de pala es utilizado cuando se requiere modelar las palas del rotor individualmente. El método de mallado se basa en determinar la geometría de las celdas de la pala y después asignar las celdas de volúmenes finitos de forma similar a la malla polar. En primer lugar se definen el número de elementos, en este caso solo se discretiza la dirección radial. Después se obtiene la geometría de las palas a partir de la cuerda y se forman los elementos de la malla a partir de cuadriláteros que se definen de la siguiente forma. Siendo la lista de radios discretizados $\{R_0, R_1, \ \dots \ , R_{N-1}\}$ y $c(r)$ una función arbitraria que indica la cuerda de la pala en función del radio, los cuatro vértices del elemento j queda definido de la siguiente forma:

- $\{R_j, -c(R_j)/2\}$
- $\{R_{j+1}, -c(R_{j+1})/2\}$
- $\{R_{j+1}, c(R_{j+1})/2\}$
- $\{R_j, c(R_j/2)\}$

Esto define la pala base, para general el resto de palas, se aplica una matriz de rotación a las coordenadas de los vértices para obtener la posición inicial de cada pala. Para obtener la malla en otro instante de tiempo, se obtiene el ángulo de cada pala para ese instante y se aplica la matriz de rotación correspondiente. Después se vuelven a asignar las celdas de volúmenes finitos a la malla de la pala.

Este método requiere que todas las celdas de la malla de la pala contengan al menos una de volúmenes finitos. Si una celda de la malla polar contiene más de una de volúmenes finitos, el termino fuente correspondiente se reparte de forma proporcional al volumen de la celda de FV. En este caso no es necesario aplicar ningún factor de promediado temporal, por que el valor de este es:

$$a_f = 1 \tag{2.104}$$

En la Figura 2.29 se observa el resultado del mallado de pala con 16 elementos en dirección radial para 2 palas y en la Figura 2.30 se observa como sería el resultado con la misma configuración pero para 3 palas.

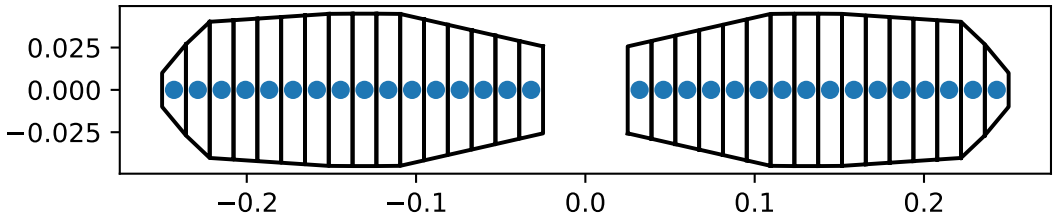

Figura 2.29: Ejemplo de mallado de pala con 2 palas

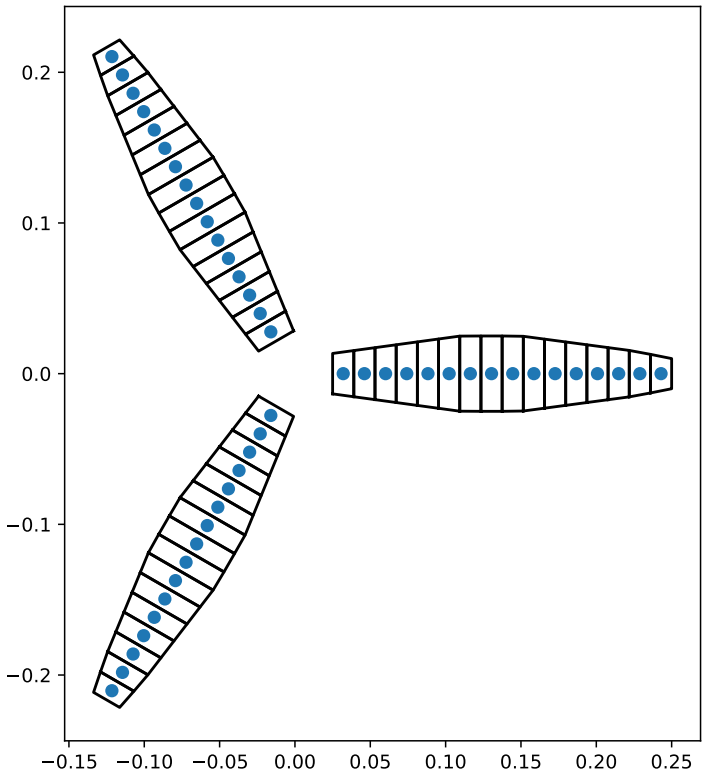

Figura 2.30: Ejemplo de mallado de pala con 3 palas

Capítulo 3

Desarrollo en OpenFOAM

OpenFOAM (Open-source Field Operation And Manipulation) es un proyecto de código abierto para el desarrollo de *solvers* numéricos y la creación de herramientas de pre/post proceso para problemas de medio continuo, enfocado en mecánica de fluidos computacional (CFD).

Origen y filosofía

El proyecto OpenFOAM se lanza por primera vez en 2004 por la empresa OpenCFD Ltd. Desde entonces, el código ha evolucionado hasta convertirse en el software líder de código abierto en la actualidad. Su filosofía se basa en una comunidad cooperativa y abierta que permita el desarrollo y mantenimiento de un código clave en el ámbito industrial y académico [14].

La gestión y el desarrollo de OpenFOAM ha sido cargo de OpenCFD Ltd. hasta 2011, cuando se le transfieren los derechos intelectuales a la fundación estadounidense *OpenFOAM Foundation, inc.* Desde entonces, OpenCFD Ltd. ha seguido manteniendo y desarrollado el código, siendo OpenFOAM Foundation Inc. el encargado de publicarlo. En 2016, OpenCFD vuelve a publicar sus propias versiones, bifurcándose la producción de OpenFOAM. Aunque ambas versiones del código son altamente similares y se realimentan entre sí, existen algunas diferencias entre ellas. A modo de clarificación, el presente trabajo ha sido desarrollado sobre la última versión de OpenFOAM de OpenCFD Ltd., siendo la web oficial: `https://www.openfoam.com` y el logotipo visible en Figura 3.1. La característica principal que ha motivado esta decisión ha sido una mejor integración de los términos fuente en los *solvers* disponibles.

Figura 3.1: Logotipo oficial de OpenFOAM.com

Además, OpenFOAM no solo ofrece un código maduro, depurado y profundamente validado, sino que además, sienta una base para la investigación y el desarrollo de métodos numéricos y modelos más avanzados. Un ejemplo de ello es el caso del trabajo presente,

que ha sido posible desarrollar gracias a la filosofía de OpenFOAM. En el presente capítulo se profundizará en la implementación en OpenFOAM. Se comentaŕan las características generales del código y como está organizado. Se mostrará la configuración y generación del caso test y las herramientas utilizadas y como se relacionan entre ellas. Finalmente se detallarán los aspectos en mayor profundidad del código, como son las clases principales utilizadas.

3.1. Nivel usuario

En esta sección, se comentará el uso general de OpenFoam a nivel de usuario, con el objetivo de proporcionar una visión general de como se realiza el flujo de trabajo. Partiendo de esta visión de usuario, resulta más asequible comprender como se organiza el código para poder desarrollar sobre este. Cabe remarcar que el objetivo no es el uso de este capítulo como guía completa. Se pretende dar una visión superficial para dar contexto al trabajo realizado y a la implementación del código.

3.1.1. Esquema general

OpenFOAM es una librería de código abierto desarrollada en C++. OpenFOAM, en conjunto con software de terceras partes, implementa distintos programas utilizables por parte de los usuarios y no requieren de conocimientos de programación para su uso. Estos programas son los que utilizan los usuarios y estos son proporcionados junto a la librería de OpenFOAM. Se dividen en 3 categorías [15]:

- Preproceso: utilidades y herramientas de mallado.
- *Solver*: resolución de las ecuaciones, con *solvers* por defecto o desarrollados por el usuario.
- Postproceso: visualización y cálculos mediante *ParaView* o otras aplicaciones.

Estas categorías se pueden observar en la Figura 3.2.

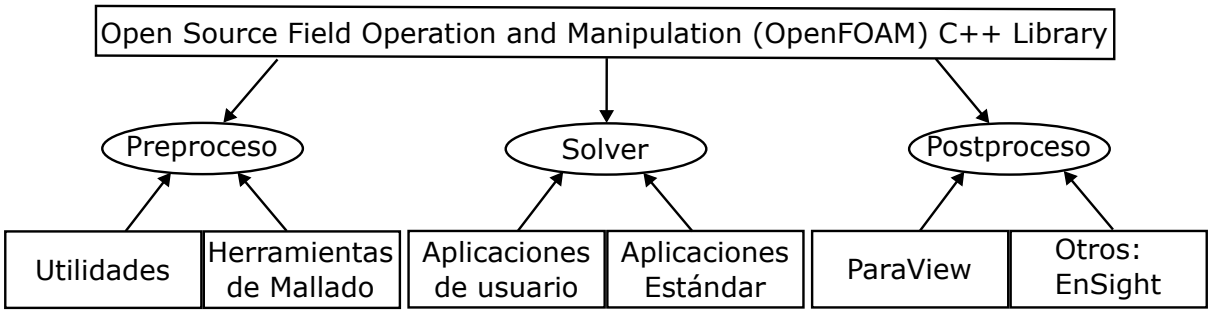

Figura 3.2: Estructura general de OpenFOAM

3.1.2. Diccionarios

La forma de configurar y interactuar con las aplicaciones de OpenFOAM, tanto para introducir datos como para extraer resultados, es mediante el uso de unos archivos de

texto conocidos como diccionarios (o *dictionaries* en inglés). Los diccionarios presentan una forma clara, amigable y concisa de introducir toda la información. Los diccionarios presentan una sintaxis similar al código fuente de C++:

- Los archivos tienen una forma libre, sin significado asignado a ninguna columna ni fila.
- Las lineas no tienen ningún significado particular excepto por // que provoca que OpenFOAM ignore todo el texto a partir del símbolo hasta final de linea.
- Un comentario de varias lineas se puede hacer encerrando el texto entre /* y */.

Para definir una **entrada** en el diccionario se utiliza la siguiente sintaxis:

```
<keyword> <dataEntry>;
```

Para el caso en el que se recibe **varias entradas**:

```
<keyword> <dataEntry1> ... <dataEntryN>;
```

Un **diccionario** puede ser en sí mismo una entrada, pudiendo anidarse indefinidamente diccionarios dentro de otros. Los diccionarios se definen entre corchetes y se escriben de la siguiente forma:

```
    <dictionaryName>
    {
        ... entradas ...
    }
```

Todos los archivos de lectura escritura, incluidos los diccionarios, son encabezados por un diccionario llamado FoamFile, que contiene un conjunto de llaves mostrado en la Tabla 3.1.

Keyword	Descripción	Entrada
version	I/O versión del formato	2.0
format	Formato de los datos	ascii / binary
location	Dirección del fichero	(opcional)
class	Nombre de la clase de OpenFOAM construida	ex: volScalarField
object	Nombre del archivo	ex: controlDict

Tabla 3.1: Palabras clave de la cabecera de los diccionarios

En el Listing 3.1 se observa un ejemplo de archivo de entrada en OpenFOAM, utilizado para definir los métodos de resolución de las ecuaciones. Al inicio del archivo, aparece un comentario, que mediante caracteres ascii recrea el logotipo de OpenFOAM. Este comentario inicial no es necesario, pero es buena práctica, ya que indica de forma clara y visualmente agradable que el archivo esta relacionado con OpenFOAM. Después aparece la cabecera, indicando los atributos necesarios. Finalmente encontramos diversos diccionarios dentro del archivo: solvers, SIMPLE y relaxationFactors. Este archivo permite configurar el solver de las ecuaciones.

Listing 3.1: Ejemplo de diccionario: fvSolution

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD |
|  \\    /   O peration     | Version:  v2206                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      fvSolution;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

solvers
{
    p
    {
        solver          GAMG;
        smoother        GaussSeidel;
        tolerance       1e-6;
        relTol          0.1;
    }

    "(U|k|omega|epsilon)"
    {
        solver          smoothSolver;
        smoother        symGaussSeidel;
        tolerance       1e-6;
        relTol          0.1;
    }
}

SIMPLE
{
    nNonOrthogonalCorrectors 0;
    consistent      yes;

    residualControl
    {
        U                   1e-1;
        p                   1e-1;
        "(k|epsilon|omega)" 1e-1;
    }
}
```

```
48
49 relaxationFactors
50 {
51     equations
52     {
53         U                       0.7;
54         "(p|k|omega|epsilon)" 0.4;
55     }
56 }
57
58 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

3.1.3. Estructura de un caso

Al contrario que en algunos programas de CFD comerciales, OpenFOAM no tiene un solo archivo definiendo el caso. En su defecto, presenta una estructura de carpetas y archivos que permiten de forma desacoplada configurar los diferentes aspectos de un problema, dándole versatilidad para reutilizar partes de forma muy sencilla. La estructura básica es la siguiente (**negrita** indica directorio):

- <**case**>: carpeta base del caso
  - **system**: ruta donde están los diccionarios que configuran el caso.
    - controlDict: define los parámetros generales de la simulación (solver, paso temporal, tiempo de simulación o librerías adicionales entre otros)
    - fvSchemes: configura los métodos de discretización.
    - fvSolution: métodos de resolución, residuales, convergencia...
  - **constant**: donde se almacenan los recursos estáticos.
    - xProperties: define diversas propiedades del fluido, turbulencia ...
    - **polyMesh**: directorio que contiene los archivos que definen la malla.
      - boundary: define las condiciones de contorno.
      - faces: define las caras de los elementos.
      - neighbour: define la conectividad de la malla.
      - owner: define la conectividad de la malla junto a neighbour.
      - points: define las coordenadas de los puntos de la malla.
  - **time directories**: carpetas que contienen los campos resultados del cálculo. El directorio 0 define las condiciones iniciales del cálculo.

3.1.4. Aplicaciones

Aunque OpenFOAM es una librería C++ utilizada para crear aplicaciones, vienen incluidas una gran cantidad de ellas. Estas se dividen en dos grandes bloques:

- Solvers: aplicaciones que resuelven problemas concretos de medio continuo (habitualmente CFD).

- Utilities: aplicaciones relacionadas con el pre/postprocesado encargadas de tareas como mallado, manipulación de datos, cálculos algebraicos o paralelizado.

Las aplicaciones están diseñadas para ser ejecutadas desde una terminal de comandos o desde un script. En la Figura 3.3 se puede observar la terminal de comandos de Ubuntu, donde se ha ejecutado el solver simpleFoam. Para indicar el caso que se quiere ejecutar, basta con abrir la terminal en la carpeta del caso. En la parte superior de la Figura 3.3 se observa que el caso ejecutado se llama *steadyFan*. Las aplicaciones presentan una gran variedad de opciones de linea de comandos indicadas en el manual de usuario de OpenFOAM [17].

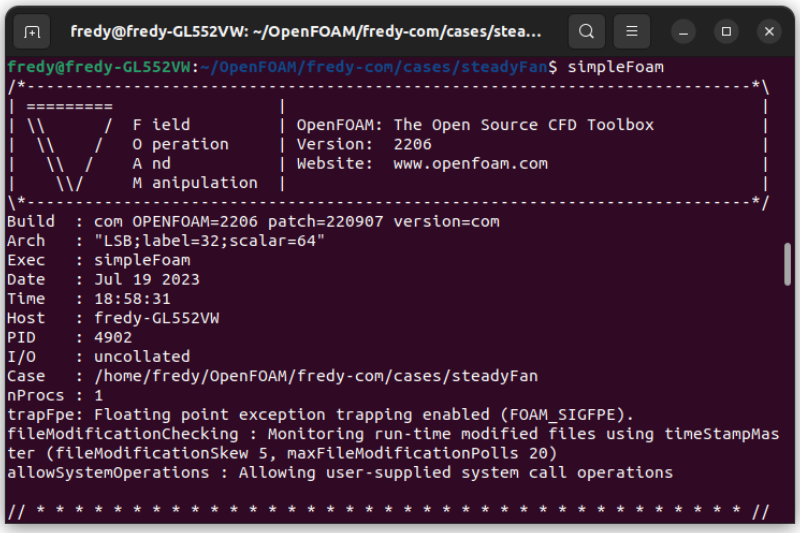

Figura 3.3: Ejecución del solver *simpleFoam* desde la terminal de Ubuntu

Solvers

OpenFOAM no cuenta con un solver genérico que sirva para todas las situaciones. No obstante, presenta una gran variedad de aplicaciones específicas para cada tipo de problema. Como ha sido comentado con anterioridad, OpenFOAM permite resolver gran variedad de problemas de sólido continuo, y reflejando esta característica en los *solver* que provee. A continuación se muestra las categorías principales de *solvers* y algunos ejemplos de ellos:

- **CFD Básico**
  - potentialFoam: flujo potencial.
  - scalarTransportFoam: transporte pasivo de una magnitud escalar.
- **Flujo Incompresible**
  - simpleFoam: estacionario, incompresible y turbulento.
  - pisoFoam: transitorio, incompresible y turbulento. Resuelto con algoritmo PISO.

- icoFoam: transitorio, incompresible para fluidos Newtonianos.
- adjointOptimisationFoam: bucle de automatización para optimización de geometría en flujo incompresible.

- **Flujo Compresible**
  - rhoCentralFoam: *density-based* para flujo compresible y con esquema central-upwind de Kurganov y Tadmor.
  - rhoSimpleFoam: estacionario, compresible y turbulento.
  - sonicFoam: transitorio, transónico/ supersónico, compresible y turbulento.
- **Flujo Multifásico**
  - cavitatingFoam: transitorio basado en equilibrio de fase liquido-vapor.
  - compressibleInterFoam: bifásico, compresible para fluidos inmiscibles.
- **Direct Numerical Simulation (DNS)**
  - dnsFoam: cajas de turbulencia isotrópica.
- **Combustión**
  - fireFoam: transitorio, para llamas de difusión turbulenta.
  - reactingFoam: transitorio, para reacciones químicas.
- **Transferencia de Calor y Convección**
  - bouyantPimpleFoam: transitorio, convección y ventilación.
- **Flujos de partículas**
  - coalChemistryFoam: transitorio, compresible, flujo turbulento, combustión y formación de hollín.
- **Dinámica Molecular**
  - mdFoam: dinámica molecular para dinámica de fluidos.
- **Monte Carlo**
  - dsmcFoam: simulación directa de Monte Carlo, para transitorio y multifásico.
- **Electromagnético**
  - mhdFoam: incompresible, laminar, flujo conductor bajo la presencia de un campo magnético.
- **Tensión en sólidos**
  - solidDisplacementFoam: transitorio, segregado y basado en volúmenes finitos para solidos lineales y pequeñas deformaciones.
- **Finanza**
  - financialFoam: resuelve la ecuación de Black-Scholes.

Utilities

Como se ha comentado anteriormente, las *utilities* de OpenFOAM son programas de pre/postproceso que dotan de herramientas básicas y adicionales para el proceso CFD. Existen una gran cantidad de *utilities*, puediendose consultar en detalle en la documentación de OpenFOAM [18]. Todos los programas se pueden ejecutar desde la terminal o mediante un *script*. Se comentan a continuación los tipos de *utilities* disponibles y algunos ejemplos de ellos, que se han considerado de interés general o han sido utilizados en el presente trabajo, mostrandose en cursiva el nombre del ejecutable:

- Preproceso
- Mallado
  - Generación: algoritmos de generación de malla.
    - *snappyHexMesh*: mallado de tetraedros con refinamiento iterativo.
    - *blockMesh*: mallado paramétrico con *grading* y bordes curvos.
  - Conversión: permite convertir desde/hasta otros formatos (Ansys, StarCCM+ ...).
  - Manipulación: modificar ciertos aspectos de la malla, a nivel global, vértices, caras o elementos.
  - Otros: como *collapseEdges*, *refineHexMesh* o *refineWallLayer*.
- Áreas finitas
- Postproceso
  - General
    - *postProcess*: permite ejecutar una serie de *functionObjects* sobre los datos generados.
  - Conversión de datos: para transformar los datos de un formato a otros, como de OpenFOAM a Fluent o VTK (*foamToVTK*).
  - Lagrangiano: aquí se engloban las utilidades para trayectorias y seguimiento de partículas.
  - *Lumped-Mass*: para representar o extraer valores de este tipo de simulaciones.
  - Acústica: utilidades para calcular y analizar espectros de frecuencias.
- Mallado de superficies
- Paralelizado
  - *decomposePar*: permite dividir un caso en cierto número de partes para su ejecución en paralelo. Se configura mediante el archivo `system/decomposeParDict`.
  - *reconstructPar*: permite reconstruir los datos generados en paralelo por cada uno de los procesos en uno solo para su postprocesado.
- Termofísica: para cálculo de tempertura adiabática de llama, equilibrio químico, concentración de monóxido de carbono ...

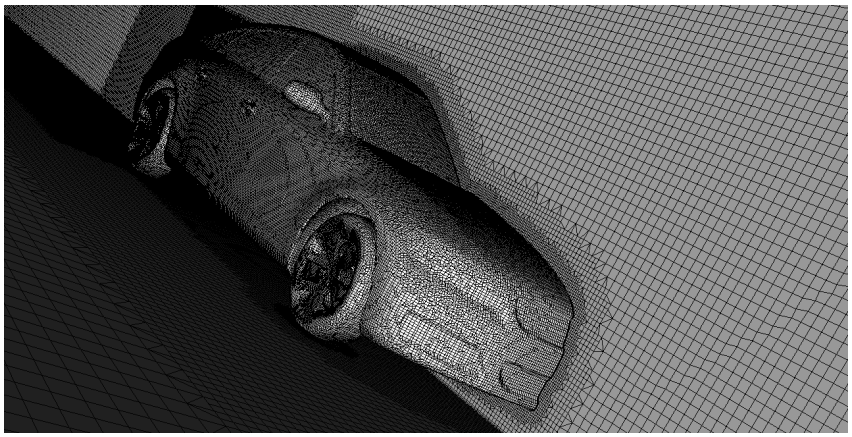

Figura 3.4: Ejemplo de *snappyHexMesh* aplicado a un automóvil [6].

- Miscelánea
  - *foamDictionary*: permite acceso y manipulación sobre los diccionarios. Permite añadir campos, modificar existentes, obtener valores. De gran utilidad para realizar estudios paramétricos.
  - *foamListTimes*: devuelve una lista de los tiempos para los cuales existe solución guardada.

3.1.5. Geometría y malla

En esta sección se comentará el procedimiento utilizado para crear la geometría y el mallado en OpenFOAM. Se han utilizado programas adicionales que han facilitado la tarea. El *mesher* utilizado ha sido el *snappyHexMesh*, una utilidad que permite generar mallas hexaédricas alrededor de objetos 3D refinando iterativamente cerca de los bordes, como se puede observar en el ejemplo de la Figura 3.4.

3.1.6. Definición de la geometría

Una de las formas de definir la geometría del problema en OpenFOAM es mediante ficheros en formato obj. Este se trata de un formato ascii en el que se detallan los vértices, normales y cara de una geometría 3D. OpenFOAM requiere del uso de un formato especial llamado .eMesh (*edgeMesh*) para el uso de *snappyHexMesh*. Para ello existe la utilidad llamada *surfaceFeautreExtractDict* que permite generar este formato a partir de los .obj. Para ello se configura el diccionario de la utilidad, llamado surfaceFeatureExtractDict de la siguiente forma:

```
1 fluid_domain.obj
2 {
3     #includeEtc
4     "caseDicts/surface/surfaceFeatureExtractDict.cfg"
5 }
```

En este caso, se aplica al archivo fluid_domain.obj. Una vez definido el diccionario se ejecuta mediante la consola de comandos *surfaceFeatureExtract*.

Mallado

A continuación se realiza un mallado *blockMesh* que servirá de base para el *snappyHexMesh*. Es necesario que todas las geometrías utilizadas intersecten al menos una celda del *blockMesh* para que el algoritmo las detecte.

Utilizando el comando:

```
surfaceCheck <nombre-archivo.obj> -blockMesh
```

podemos obtener el diccionario base para el *blockMesh*, llamado *blockMeshDict*. Aplicando este comando a un archivo de ejemplo obtenemos el siguiente resultado:

```
// blockMeshDict info

vertices
(
    (-0.25 0 -0.25)
    (0.25 0 -0.25)
    (0.25 0.01 -0.25)
    (-0.25 0.01 -0.25)
    (-0.25 0 0.25)
    (0.25 0 0.25)
    (0.25 0.01 0.25)
    (-0.25 0.01 0.25)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (10 10 10) simpleGrading (1 1 1)
);

edges
();
patches
();

// end blockMeshDict info
```

Una vez tenemos el diccionario definido se ejecuta el comando:

```
blockMesh
```

El siguiente paso es configurar el snappyHexMeshDict correctamente para la generación de esta malla [16]. Hay una serie de atributos importantes que es necesario mencionar:

- *geometry*: define la geometría de la malla que va a realizarse, así como los contornos.
- *castellatedMeshControls*: define como se realiza la división de las celdas.
- *addLayersControl*: se indica el mallado de capa límite.

FreeCad y OpenFOAM

El uso de *snappyHexMesh* puede resultar complejo para nuevos usuarios, por lo que en el presente trabajo se propone el uso de una alternativa que permite generar la geometría y configurar los diccionarios desde una interfaz gráfica. FreeCad es un programa de código abierto para diseño asistido por ordenador. Presenta una gran cantidad de herramientas y *addons*, que le permite ser un programa completo y competitivo en el mercado FreeCad y OpenFOAM. Uno de los módulos existentes, CfdOF permite integrar OpenFOAM en el entorno gráfico de FreeCad y usar este para definir la geometría, el mallado e incluso el *solver* del cálculo. Se recomienda el uso para generar un caso base con los archivos necesarios y luego modificar estos para que cumplan con los detalles más finos, ya que las capacidades del *addon* son hasta cierto punto limitadas.

Para la instalación del *addon* se realizan los siguientes pasos:

1. Se abre el programa FreeCad y se abre el *Addon manager* Figura 3.5.
2. Se instala el *addon* Plot Figura 3.6.
3. Se instala CfdOF Figura 3.7.
4. Se configura la ruta de las dependencias, y se instala *cfdMesh* y HISA desde la propia interfaz si no se dispone Figura 3.8. Para realizar este paso es necesario tener instalado OpenFOAM y ParaView.
5. Se cambia el entorno de trabajo al de CfdOF Figura 3.9. Esto activará una serie de iconos nuevos en el menú superior, indicado que la instalación ha sido correcta.

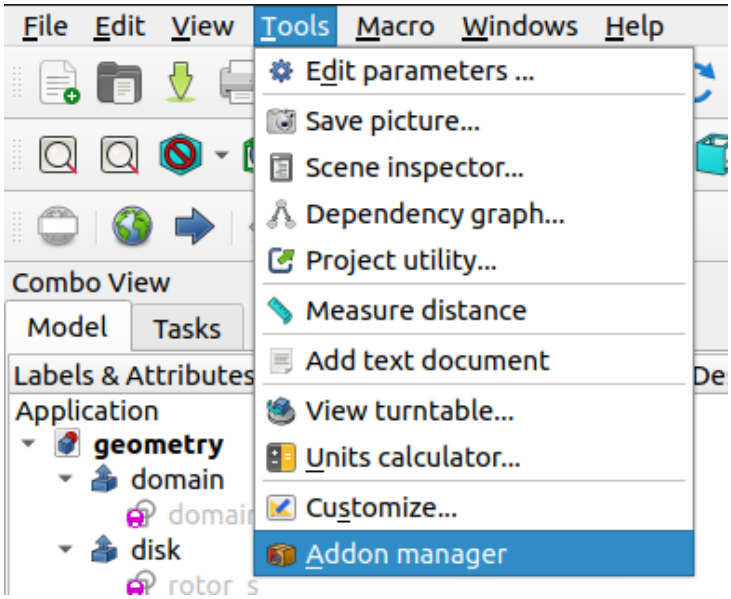

Figura 3.5: Paso 1: abrir *Addon manager*.

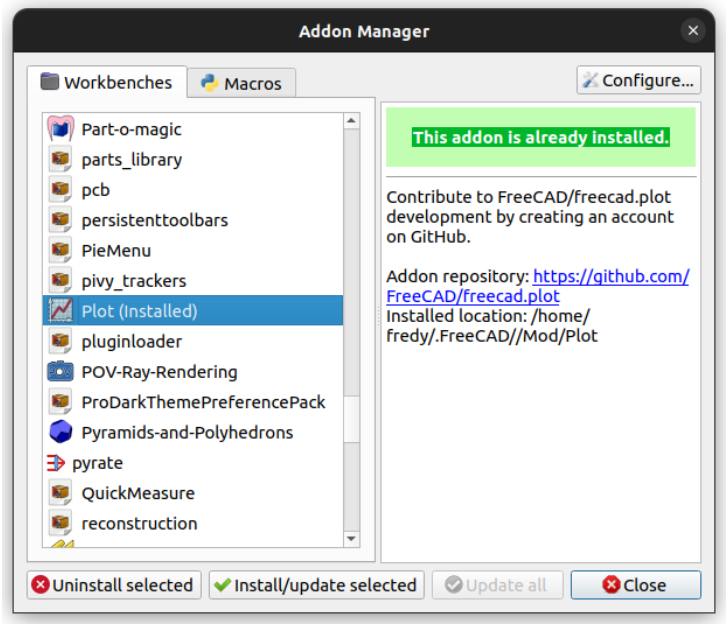

Figura 3.6: Paso 2: instalar *Plot*.

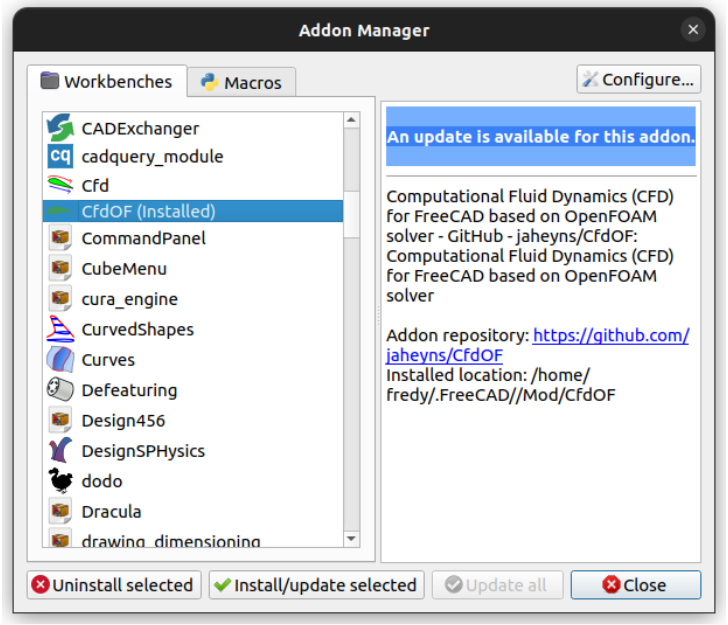

Figura 3.7: Paso 3: instalar CfdOF

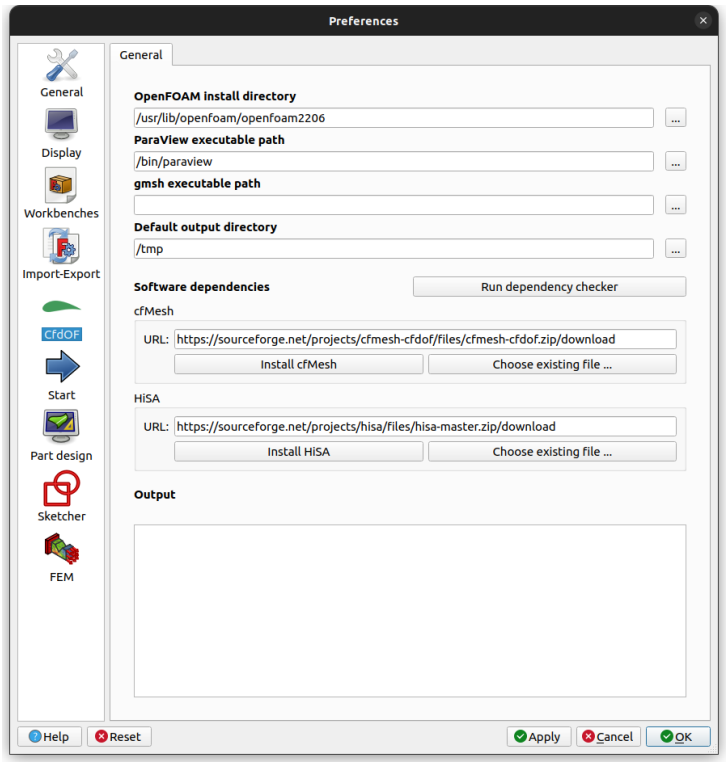

Figura 3.8: Paso 4: configurar las preferencias de CfdOF.

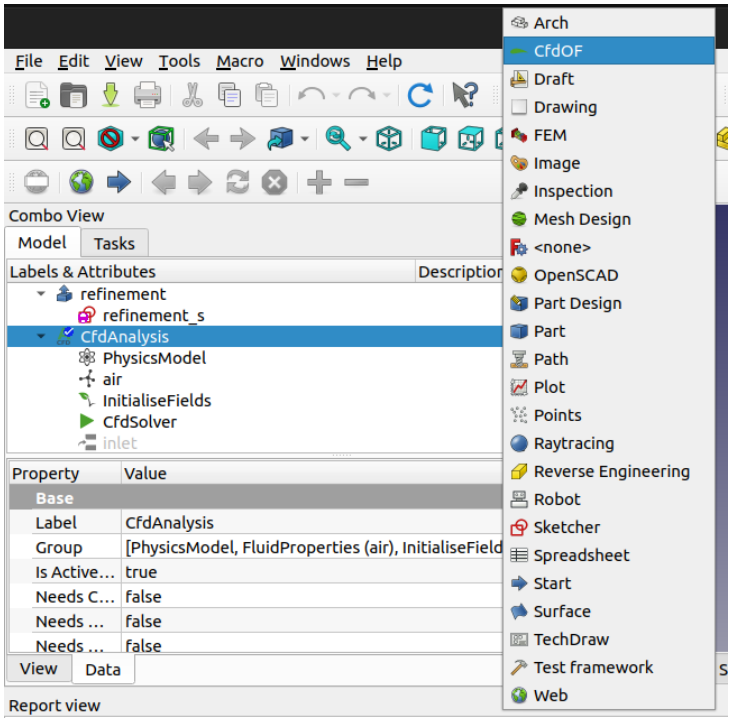

Figura 3.9: Paso 5: cambiar de entorno a CfdOF.

A continuación, se explicará brevemente como se realiza la generación y visualización del mallado:

1. Se crea la geometría del dominio, los cuerpos y las zonas de refinado. En la Figura 3.10

se observa el dominio fluido (cilindro exterior) y los objetos de refinado (cilindros interiores).

2. Se crea un nuevo caso de CFD haciendo *click* en el icono con una A, mostrado en la Figura 3.11.

3. Se crea la malla haciendo *click* en el icono mostrado en la Figura 3.12.

4. Se configuran los parámetros de la malla mostrados en la Figura 3.13.

5. Se crea una nueva zona de refinado haciendo *click* en el icono mostrado en la Figura 3.14.

6. Se indican los parámetros de refinado según la Figura 3.15.

7. Se ejecuta el mallado haciendo *click* al botón *Write mesh case*. Esto creará los archivos del caso para realizar el mallado, y tras esto pulsar en *Run mesher*. Estos botones se sitúan en la ventana principal de la malla, mostrada en la Figura 3.13. Cuando termine el proceso, la malla ya estará creada.

8. Para visualizar la malla hay dos opciones:

   - Mediante FreeCad, en ese caso se hace *click* en *Load surface mesh*. El resultado se muestra en Figura 3.16.
   - La recomendada, mediante ParaView, se pulsa el botón *Paraview*. Este método es más versátil y permite visualizar mejor la malla, mediante cortes o otras herramientas, mostrado en Figura 3.17.

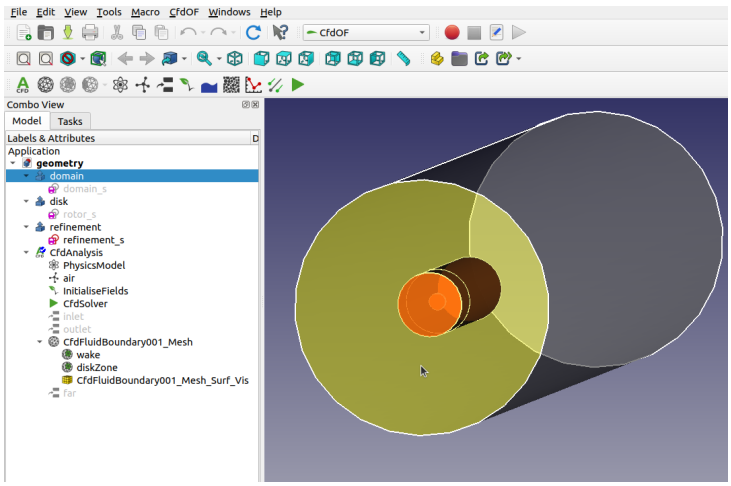

Figura 3.10: Paso 1: crear la geometría de los elementos.

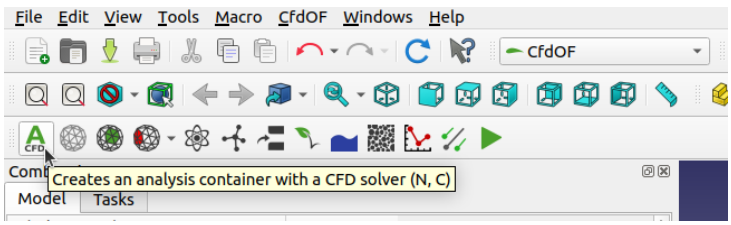

Figura 3.11: Paso 2: crear un nuevo caso de CFD.

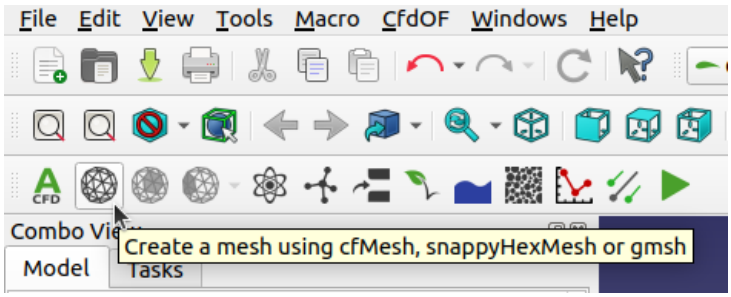

Figura 3.12: Paso 3: crear un nuevo objeto de mallado.

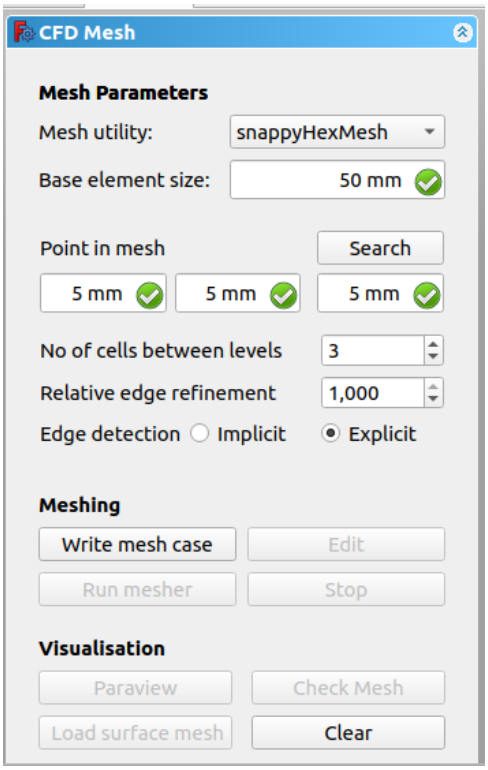

Figura 3.13: Paso 4: configurar los parámetros del mallado y la geometría utilizada.

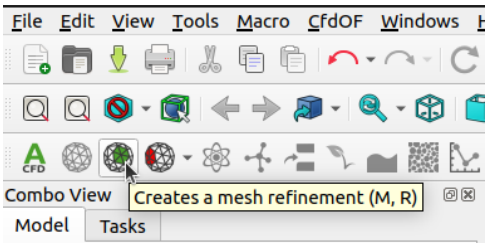

Figura 3.14: Paso 5: crear zona de refinado.

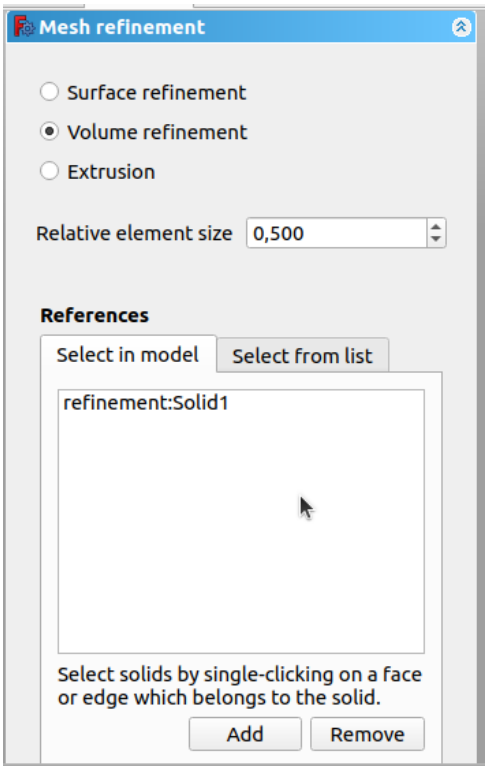

Figura 3.15: Paso 6: indicar los parámetros del refinado.

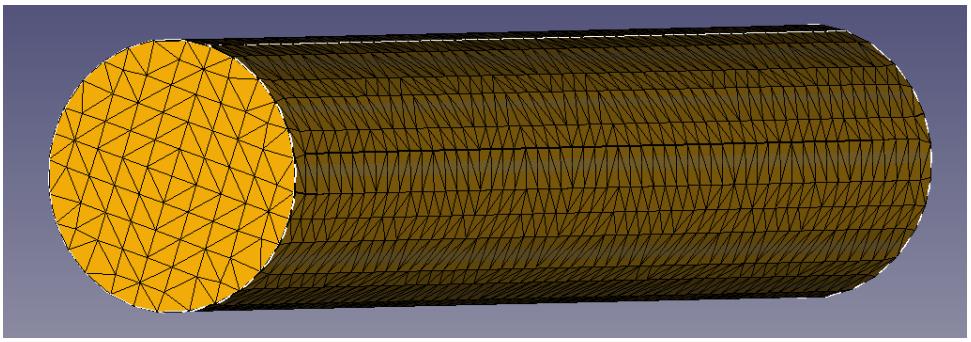

Figura 3.16: Paso 8: visualización de la malla en FreeMesh.

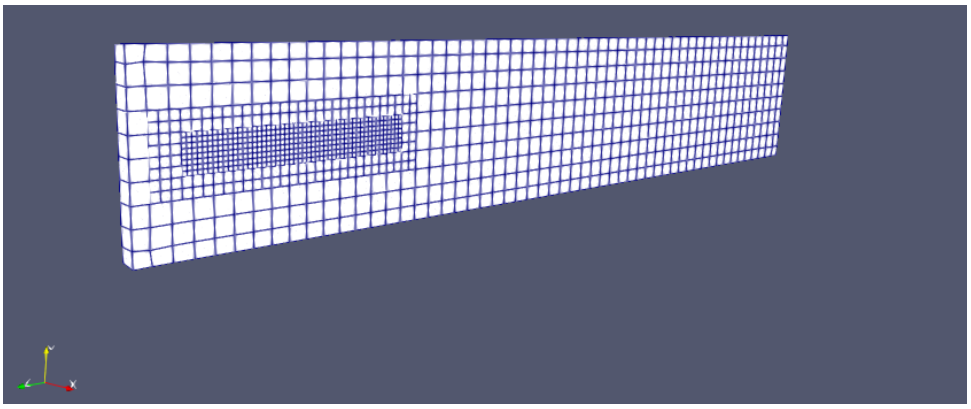

Figura 3.17: Paso 8: visualización de la malla en ParaView.

3.2. Proceso de instalación

En esta sección se detallan los pasos seguidos para la instalación y uso de OpenFOAM, siendo válido tanto para nivel de usuario como desarrollador aunque enfocado a obtener las herramientas de desarrollador.

3.2.1. Sistema Operativo

En cuanto al sistema operativo, hay distintas opciones válidas. OpenFOAM está desarrollado en un entorno Linux, pensado para su uso principalmente en máquinas de cálculo o *clusters*. Para su uso, no es necesario utilizar sistemas basados en Linux, es posible utilizar los binarios de OpenFOAM mediante una máquina virtual llamada Docker [10] en Windows o Mac. Para el desarrollo, también es posible utilizando una máquina virtual de Linux o WSL *(Windows Subsystem for Linux)*. La opción que se recomienda es el uso de algún sistema operativo basado en Linux, en este caso se ha utilizado Ubuntu. Ubuntu es un sistema operativo libre, relativamente ligero y ampliamente utilizado. Se ha considerado la mejor opción para el trabajo ya que permite utilizar al máximo los recursos disponibles sin intermediarios, obteniendo mejor rendimiento del sistema. Además, se evita problemas de compatibilidad que pueden ocasionar las otras alternativas. La instalación de Ubuntu se ha realizado en un portátil personal realizando una partición del disco duro, pudiendo usarse de forma paralela a Windows. La instalación de Ubuntu no presenta ninguna dificultad y hay gran cantidad de guías en internet [29].

3.2.2. Instalación de OpenFOAM

Para el uso como desarrollador es necesario descargar el código fuente de OpenFOAM. Esto se puede hacer desde el repositorio oficial[19], donde viene detallado el proceso completo de la instalación y información adicional. Los pasos para la compilación a partir del código fuente se detallan a continuación.

Configuración del entorno

En primer lugar, es necesario configurar el entorno de OpenFOAM. Para ello hay que indicarle al sistema de la terminal el archivo donde se encuentran las variables de entorno de OpenFOAM. En este archivo se definen las rutas de las diferentes carpetas de OpenFOAM, las opciones de compilación, la arquitectura (32 o 64bit) la precision utilizada (float o double). Hay dos formas para indicar el uso de las variables definidas: directamente en la consola cada vez que se utilice o configurando un archivo de Linux. Por comodidad se recomienda la segunda opción. Para ello se añade al final del archivo situado en la raíz del sistema llamado ~/.bashrc la siguiente línea de texto:

```
1    source <installation path>/OpenFOAM-vXXXX/etc/bashrc
```

Donde <installation path> es la ruta donde se encuentra el código fuente de OpenFOAM y vXXX es la versión utilizada. Si no se desea que la configuración sea persistente, basta con ejecutar el comando anterior cada vez que se abra la terminal.

Modo de compilación

Dado que el código fuente se va a utilizar como desarrollador es conveniente realizar la compilación en modo *debug* y en modo *release*. El modo ***debug*** genera información adicional que permite observar paso a paso las lineas de código que se ejecutan y el estado del programa para su depuración (*debugging*). El modo ***release*** realiza una compilación lo más optimizada posible para maximizar la velocidad y tamaño del código, apto para generar la aplicación final que se utilizará para resolver los casos.

Para escoger si OpenFOAM es compilado en modo *debug* o *release* se modifica la variable de entorno WM_COMPILE_OPTION del archivo:

```
1    <installation path>/OpenFOAM-vXXXX/etc/bashrc
```

Para modo *debug*:

```
1    export WM_COMPILE_OPTION=Debug
```

Y para modo *release*:

```
1    export WM_COMPILE_OPTION=Opt
```

Compilación

Para inicial el proceso de compilación, se abre la terminal y se usa el comando foam para cambiar la ruta utilizada a la ruta del código fuente de OpenFOAM. Realizado esto, se ejecuta el siguiente comando que iniciará el proceso de compilación:

```
1    ./Allwmake -s -l -j
```

Donde la opción −s se utiliza para reducir el texto generado, −l guarda la salida de texto a un archivo de texto (*log file*) y −j utiliza todos los núcleos disponibles para la compilación. Puede indicarse después de −j el número de núcleos que se desea utilizar, por ejemplo:

```
1    ./Allwmake -s -l -j 4
```

Utilizará 4 núcleos para la compilación.

3.2.3. *Debugging* con *Visual Studio Code*

Durante el desarrollo de cualquier proyecto de programación es necesario configurar correctamente un entorno eficiente de trabajo para maximizar la comodidad del desarrollo y optimizar el tiempo. Aunque no es necesario el uso de ningún programa adicional y es posible desarrollar y depurar el código mediante herramientas de consola de comandos, es altamente recomendable su uso.

En el presente trabajo se propone el uso de *Visual Studio Code*. Se trata de un entorno de desarrollo altamente configurable, con gran cantidad de extensiones. Para su uso en conjunto con OpenFOAM el primer paso es descargar la aplicación de la web oficial [5]. Una vez instalado es necesario descargar la extensión C/C++. Esto puede realizarse desde Visual Studio Code, en la pestaña de extensiones del menú de la izquierda. Después se utiliza la terminal por comodidad y se realizan los siguientes pasos:

- Se crea una nueva carpeta y se cambia la ruta a ella:

```
mkdir -p <folder>
cd <folder>
```

- Se crea una copia de los test de OpenFOAM y se cambia la ruta:

```
cp -r $WM_PROJECT_DIR/applications/test $FOAM_RUN
cd test/vector
```

- Esto es solo para entender el proceso, para usar en otro proyecto se puede copiar y pegar el contenido de la carpeta vscode.

- Se abre Visual Studio Code en el directorio actual:

```
        code .
```

- Se presiona sobre la pestaña *Terminal* en *Visual Studio Code*, seleccionado *configure build task* y se escoge *g++ build active file*. Se edita el archivo *task.json*, eliminado las opciones *args*:

```
"label": "wmake-build",
"command": "wmake"
```

- Se presiona sobre la pestaña *Debug* en *Visual Studio Code* y se crea el archivo *launch.json* para *g++ build and debug active file*. Se editan los campos siguientes:

```
"name": "OF-Debug",
"program":
    "${env:FOAM_USER_APPBIN}/${fileBasenameNoExtension}",
"preLaunchTask": "wmake-build",
```

Con esta configuración ya está listo el entorno de *Visual Studio Code* para depurar OpenFOAM. Para asistir en el desarrollo del código con funciones de autocompletar y detección de errores se puede configurar *IntelliSense*. Para ello:

- Se abre la paleta de comandos mediante *ctrl+shif+p* y se selecciona *c/c++ : Edit Configuration U*, creándose un archivo llamado *c_cpp_properties.json*.

- Para finalizar se configura el modo de funcionamiento de *IntelliSense*: File > Preferences > Settings y se modifica la opción de *C_Cpp: IntelliSense Engine* a *Tag Parser*.

3.2.4. Compilación y wmake

El sistema de compilación de OpenFOAM está basado en *make*, pero utiliza sus propios *scripts*, que resultan más sencillos y versátiles de utilizar. Este sistema se llama *wmake* y a continuación se realizará una explicación de su uso.

Dentro del directorio del código que se desea compilar, se crea una carpeta llamada *Make*. En su interior se definen dos archivos llamados: *files* y *options*. En el archivo *files* se

definen los archivos que contienen el código fuente que va a ser compilado. En el archivo *options*, se definen las librerías adicionales y directorios de inclusión utilizados.

Hay dos formas principales de compilar en OpenFOAM: como ejecutable y como librería. Los ejecutables son aplicaciones que se pueden utilizar directamente por línea de comandos, como son los *solvers* y las *utilities*. Las librerías, en cambio, contienen código compilado y han de ser unidas a un ejecutable para poder ser utilizadas. Las librerías se pueden incluir al compilar el código (archivo *options*) o de forma dinámica, indicándolo en el campo *libs* del diccionario *controlDict* de la siguiente forma:

```
libs
(
    "libdynamicSmagorinskyModel.so"
);
```

Archivo *file*

Como ya ha sido comentado, en el archivo *files* se definen los archivos que van a ser compilados y el binario generado (ya sea ejecutable o librería). Los archivos de compilación se definen indicando la ruta relativa de cada uno dentro de la carpeta base en la que se encuentra la carpeta *Make* de la siguiente forma:

```
    newApp.C
    class/someClass.C
    class/otherClass.C
```

Para crear un ejecutable, se añade al final del archivo la siguiente instrucción:

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

Donde *$ (FOAM_USER_APPBIN)* es la ruta por defecto de las aplicaciones compiladas por los usuarios, aunque cualquier ruta sería válida. El nombre de la aplicación generada será, por tanto, *appName*. Para crear una librería, se usa el siguiente comando:

```
    LIB = $(FOAM_USER_LIBBIN)/appName
```

Siendo la variable de entorno la ruta por defecto de las librerías creadas por los usuarios.

Archivo *options*

En el archivo *options*, se especifica los directorios de inclusión y la librerías utilizas para la generación del archivo binario. Los directorios de inclusión se especifican de las siguiente forma:

```
EXE_INC = \
    -I<path1>\
    -I<path2>
```

Donde <path1> y <path2> son carpetas donde se encuentran los archivos de inclusión. Para definir las librerías utilizadas, se utilizan diferentes comandos en función de si se va generar un ejecutable o una librería. Para un ejecutable se realiza de la siguiente forma:

```
EXE_LIBS = \
    -l<libName1> \
    -l<libName2>
```

Donde <libName1> y <libName2> son los nombres de las librerías. Y para crear una librería se utiliza:

```
LIB_LIBS = \
    -l<libName1> \
    -l<libName2>
```

3.3. Nivel desarrollador

En la presente sección, se entrará en detalle de los elementos y herramientas de programación principales de OpenFOAM. Estas han sido clave para la implementación del método de BET. La documentación oficial de OpenFOAM y otras no oficiales, han sido clave para entender el funcionamiento general y el uso de algunos tipos y clases del código. Aun así, gran parte del código no se encuentra explicada. El código de los diferentes módulos de OpenFOAM se encuentra bajo el directorio *OpenFOAM-vXXXX/src*. El módulo principal, donde se encuentra la base del código se encuentra en *OpenFOAM-vXXXX/src/OpenFOAM*. Como norma general las clases que empiezan en letra mayúscula, son *templates* para tipos genéricos, y las que empiezan con letra minúscula son definiciones para un tipo concreto a partir de *templates* o clases sin *template*.

3.3.1. Tipos primitivos

Los tipos primitivos conforman la base del código, sobre los que se construye el resto. El primer paso es la definición de los tipos primitivos básicos a partir de las palabras clave de C++ y la precisión con la que se quiere calcular. La definición de los tipos primitivos está localizada en el directorio *OpenFOAM-vXXXX/src/OpenFOAM/primitives* y cuenta con una gran variedad. A continuación, se comentarán los tipos utilizados en el proyecto.

label

El tipo que define las variables utilizadas como índices es llamado en OpenFOAM como `label`. Se define como entero de 32 o 64 bits mediante la variable de entorno `WM_LABEL_SIZE = 32 | 64`. El separador vertical se utiliza para distinguir las diversas opciones que pueden utilizarse. Con 32 bits se pueden definir listas de hasta 2^{32} elementos. Con 64 bits la cifra aumenta de forma considerable hasta los 2^{64} elementos. En sistemas de 64 bits es natural definir esta variable como 64bits.

scalar

El tipo que define los números reales es llamado `scalar`. De la misma forma que `label` puede definirse de 32 o 64 bits mediante la variable de entorno:

```
WM_PRECISON_OPTION = DP | SP | SPDP
```

DP para precisión doble (64bit), `SP` para precisión simple (32bit) y `SPDP` para precisión doble internamente y precisión simple de forma externa.

string

La clase `string` se utiliza para contener cadenas de texto. En los archivos I/O viene contenida entre comillas, ya que puede contener cualquier caracter: çadena de texto". Se deriva de la clase de la librería estándar `std::string`. Contiene los métodos de acceso y modificación habituales.

word

El tipo `word` es derivado de `string` y define una cadena de texto que representa una palabra. No puede contener espacios ni caracteres especiales y se define sin comillas en los archivos I/O.

fileName

Esta clase es utilizada para representar la ruta de un archivo y es derivada de la clase `string`. Tiene métodos especiales de acceso y modificación de la ruta del archivo. Tiene funcionalidad para soportar diferentes tipos de acceso, rutas relativas o absolutas, concatenación de rutas, extensión de archivo, etc. A continuación se comentan algunos métodos de utilidad:

- *word name()*: devuelve el nombre del archivo con la extensión, sin la ruta completa.
- *word lessExt()*: devuelve el nombre del archivo sin la extensión.
- *word ext()*: devuelve la extensión del archivo.
- *filename& replace_ext(const word& ending)*: permite reemplazar la extensión de un archivo.
- *fileName path()*: devuelve la ruta del archivo sin el nombre del archivo.

Vector

Esta clase define el vector del campo de las matemáticas de 3 componentes. La clase está definida como plantilla, por lo que admite cualquier tipo de variable, lo más habitual es el uso de `scalar`. Para el acceso y modificación se usan los métodos `type x()`, `type y()`, `type z()`. La palabra `type` hace referencia al tipo genérico de la *template*.

3.3.2. Tensor

La clase `Tensor` representa matrices de $3x3$, con plantilla de tipo genérico, de la misma forma que *Vector*. El acceso y modificación se puede realizar mediante métodos que combinan el nombre de las componentes `xx()`, `xy()`. También se puede acceder a filas, columnas o diagonal y realizar ciertas operaciones matemáticas, como obtener la matriz inversa o transpuesta.

Tuple2

Esta clase es un *wrapper* para dos variables de tipo genérico. Es de utilidad para realizar algoritmos o devolver varios valores de una función de tipos distintos. Permite el acceso a cada variable mediante los métodos: `type1 first()` y `type2 second()`.

Enum

La utilidad de esta clase es muy practica. Permite asociar cada valor de un *enum* a una palabra (*word*). Esto es de gran utilidad para la entrada y salida, ya que permite mostrar por terminal los valores de un *enum* mediante un nombre descriptivo y/o leer de un archivo el nombre asociado.

3.3.3. Contenedores

En programación, los contenedores son estructuras abstractas de datos. En ellas los datos se encuentran ordenados con una estructura concreta y presentan una serie de normas de acceso de lectura y escritura. A continuación, se detallan los contenedores utilizados para el proyecto.

HashTable

En este tipo de contenedor, los datos se almacenan mediante pares de clave-valor y presentan un tiempo de acceso constante. La clase en OpenFOAM esta implementada con una plantilla que acepta dos argumentos, que son el tipo variable de la clave (llamado Key) y el valor (llamado T). La clave por defecto es de tipo `word`. Los métodos principales son los de acceso y adición:

- `T& at(const Key& key)`: devuelve el valor asociado a la clave key. Si no existe se produce un `FatalError`.
- `bool found(const Key& key)`: devuelve `true` si el valor key se encuentra en la tabla y `false` si no.
- `bool insert(const Key& key, T obj)`: inserta una nueva entrada **sin sobrescribir** si ya existe. Este método se encuentra sobrecargado para soportar distintos tipos de inserciones: por copia (`const T& obj`) o mediante *move semantics* (`T&& obj`).
- `bool set(const Key& key, T obj)`: inserta una nueva entrada **con sobrescritura** si ya existe. Este método se encuentra sobrecargado para soportar distintos tipos de inserciones: por copia (`const T& obj`) o mediante *move semantics* (`T&& obj`).

FixedList

Se trata de una lista con una longitud fija. Presenta dos parámetros de `template`: el tipo de dato y el número de elementos. Los datos se almacenan internamente como un *array*. Para acceso de lectura y escritura se utiliza el operador `[ ]`. Para inicializar los elementos de la lista, se puede utilizar el constructor, pasándole un valor que se copiará

para todos los elementos, otra `FixedList` o una `std::initializer_list`. Esta última es utilizada para construir la lista de forma explícita con los elementos separados por comas dentro de llaves {a,b,c}.

List

En este caso, se trata de una lista de longitud variable. Permite modificar en tiempo de ejecución la longitud de la lista. No está optimizada para ser actualizada con mucha frecuencia. Si es este el caso, se recomienda el uso de `DynamicList`. Los métodos de acceso son equivalentes a `FixedList`. Los métodos principales son los siguientes:

- `void append(T val)`: inserta val al final de la lista.
- `void resize(const label size)`: cambia el tamaño de la lista a size.
- `void resize(const label size,const T& val)`: modifica el tamaño de la lista y asigna el valor val a los nuevos elementos creados, en caso de que se creen nuevos.

PtrList

Se trata de una lista como el caso anterior, pero adaptada para manejar internamente punteros. Se encarga de reservar y liberar la memoria de forma automática. Los métodos son equivalentes.

3.3.4. Input/Output

La entrada y salida de datos es crucial en todo tipo de programas, realizarlo de forma sencilla y eficiente es clave para un programa de simulaciones. Una estructura clara y sencilla facilita enormemente el intercambio de información.

I/O Streams

Los *streams* representan flujos de información entre dos puntos. Se tratan de estructuras abstractas encargadas de diversas cuestiones. Los *streams* proporcionan una interfaz común para distintos tipos de transmisión. La sintaxis para enviar datos es:

```
1    Stream<<data;
```

Y para recibirlos:

```
1    Stream>>data;
```

Los operadores << y >> pueden interpretarse como la dirección en la que se transmite la información, de la variable al *stream* o del *stream* a la variable. Para mostrar información por la terminal de comandos se utilizan distintos *streams*: `Info`, `Warning` y `PStream`. El primero es para mostrar datos informativos, el segundo para mostrar mensajes de advertencia y el ultimo es equivalente al primero pero para multiproceso.

Para comunicarse con ficheros se utiliza dos tipos de *streams*: IFstream(*Input File stream*) para leer datos de un fichero y OFstream(*Output File stream*) para guardarlos.

dictionary

Los ficheros de diccionario, representan la principal herramienta para comunicarse con un programa de OpenFOAM. Esta clase es la encargada de leer y escribir sobre dichos ficheros. Los métodos principales de lectura son los siguientes:

- `T getOrDefault(const word& keyword, T value)`: devuelve el valor de la entrada keyword. Si no existe devuelve `value`.
- `T get(const word& keyword)`: devuelve el valor de la entrada. FatalError si no existe.
- `bool readIfPresent(const word& keyword, T& val)`: devuelve `true` si existe la entrada `keyword` y almacena el valor en `val`.
- `dictionary& subDict(const word& keyword)`: devuelve el diccionario con nombre `keyword`. `FatalError` si no existe.
- `dictionary subOrEmptyDict(const word& keyword)`: devuelve el diccionario con nombre `keyword` o un diccionario vacío si no existe.

Los métodos principales de modificación son:

- `entry* add(const keyType& k, const word& v)`: inserta una nueva entrada en el diccionario o modifica una existente. Devuelve un puntero a la entrada añadida.
- `bool remove(const word& keyword)`: elimina la entrada `keyword`.
- `entry* add(const keyType& k, const dictionary& d)`: inserta el diccionario `d`.

IOobject y regIOobject

`IOobject` define los atributos que permiten el manejo automático de entrada y salida mediante un registro de objetos llamado `objectRegistry`. Esta clase representa un enlace con un archivo y las reglas con las que se va comunicar, el tipo de acceso y de operaciones disponibles.

Un `IOobject` se construye con un nombre, un nombre de clase, una ruta, una referencia a un `objectRegistry` (que puede ser, por ejemplo, una instancia de `time` o `fvMesh`) y parámetros que definen el modo de funcionamiento. Una forma habitual de crear un `IOobject` es la siguiente:

```
1 IOobject
2 (
3     name,             //Ex: "FuerzasBET"
4     path,             //Ex: "mesh.time().timeName()
5     objectRegistry,   //Ex: mesh or time
6     readOption,       //Ex: IOobject::NO_READ
7     writeOption       //Ex: IOobject::AUTO_WRITE
8 );
```

Extendiendo `IOobject`, existe la clase `regIOobject` que presenta la función virtual pura:

```
1     virtual bool writeData(Ostream&) = 0;
```

Esta función es llamada por el registro de objetos cada vez que se requiera guardar los datos. Esto es de gran utilidad sobretodo cuando la solución converge. Es posible guardar manualmente los datos mediante el método `write()` que presentan los objetos como `VolField`. Pero cuando la solución converge, no se puede saber de antemano, y para llamar al método `write()` después de que la solución converja habría que hacerlo desde partes del código poco accesibles y de forma poco elegante. La solución pasa por el autoguardado. Para ello se extiende la clase que se requiere guardar a partir de la clase `regIOobject` y se implementa la función `writeData(Ostream&)`, guardando los datos necesarios a través del objeto `Ostream`. El autoguardado se puede habilitar para cualquier clase que dependa de un IOobject, basta con pasarle el parámetro `IOobject::AUTO_WRITE` al constructor. Como aclaración, las clases de OpenFOAM que requieren del objeto `IOobject` en su constructor, presentan la opción de autoguardado, pero es en alguna clase derivada donde se implementa como se realiza el autoguardado, no es la clase `IOobject`. Por tanto, es requerimiento extender de la clase `regIOobject` para poder implementar esta funcionalidad.

3.3.5. Error Managment

En cualquier programa es necesario que exista una forma clara de tratar los errores que aparecen en la ejecución del código. En programas diseñados para su ejecución a largo plazo, es necesario un amplio sistema de gestión y solución de errores de tiempo de ejecución, de forma que un error no sea fatal. OpenFOAM está pensado para ejecutarse en un entorno controlado, donde no hay fenómenos extraños sobre la ejecución. La mayoría de errores que aparecen suelen formar parte de un input incorrecto del usuario que resulta en un estado incorrecto del programa. Algunos ejemplos de ello son: nombre de alguna clase mal escrito, defectos en la malla, fallos en los métodos numéricos por mala configuración, etc. Para esta serie de fallos catastróficos de los que el programa no puede recuperarse, OpenFOAM provee de una serie de macros que muestran descripciones detalladas del motivo de fallo usado conjuntamente con la función `exit(FatalError)` para cerrar el programa de forma segura. Las dos macros utilizadas principalmente son: FatalErrorInFunction y FatalErrorInLookup.

FatalErrorInFunction

Esta macro se utiliza para mostrar un error en el código de una función. Además, muestra el nombre de la función, el archivo y la línea donde ha sido producido el error, dándole versatilidad para su depuración. La macro admite un solo parámetro, un mensaje que será mostrado al usuario, explicando el motivo del error.

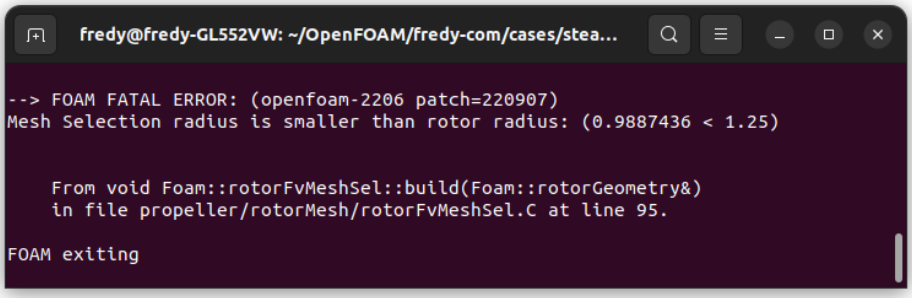

Figura 3.18: Mensaje de error mostrado con la macro FatalErrorInFunction.

FatalErrorInLookup

Esta macro se utiliza cuando hay un error en una tabla de búsqueda al no existir el elemento requerido. Un ejemplo de ello es cuando se intenta acceder mediante una RunTimeTable a una clase que no ha sido añadida. En la Figura 3.19 se muestra el mensaje de error al intentar obtener un objeto de tipo *wrongGrid*. Además del mensaje de error, y la función y línea donde se ha producido, esta macro también informa de las entradas válidas disponibles, facilitando la corrección por parte del usuario.

Figura 3.19: Mensaje de error mostrado con la macro FatalErrorInLookup.

3.3.6. Memory Managment

En C++ existen dos tipos de memoria: estática y dinámica. La memoria estática es conocida en tiempo de compilación y la reserva y liberación es automática. La duración de una variable está restringida a su ámbito. Por el contrario, la memoria dinámica se crea en tiempo de ejecución y es responsabilidad del desarrollador del código de su gestión: creación y liberación. C++ provee de dos palabras clave para ello: `new` para la reserva de memoria y `delete` para la liberación. Liberar la memoria no utilizada es necesario ya que si se reserva nueva memoria de forma constante, llega un punto en el que se agotan los recursos disponibles. Esto es conocido como *memory leaks* y es un tema importante en C++.

Para realizar un manejo eficiente y seguro de la memoria la tendencia ha sido hacia el uso de punteros inteligente. Estos son un tipo especial de contenedores que se comportan como punteros básico en su uso, pero presentan una gestión autónoma de la memoria, encapsulado la reserva y la liberación de esta, evitando los *memory leaks*.

Según el tipo de gestión de la memoria existen dos tipos principales de punteros inteligentes: autoPtr y tmp.

autoPtr

Los punteros inteligentes de tipo `autoPtr` están diseñados para punteros que poseen un objeto. Son los propietarios de este. El objeto creado mediante `autoPtr` solo puede ser referenciado por una instancia de `autoPtr` a la vez. No puede pertenecer a varios, pero es posible transferir la pertenencia entre ellos. Cuando la variable `autoPtr` sale fuera de su ámbito, se encarga de liberar los recursos del puntero que contiene. Permite, hasta cierto punto, un comportamiento de la memoria dinámica como si se tratase de memoria estática. La clase `autoPtr` es similar a `std::unique_ptr` de la librería estándar de C++.

tmp

Los punteros inteligentes de tipo `tmp` se utilizan, como indica su nombre, para objetos temporales. Diferentes instancias de `tmp` pueden apuntar a la misma región de memoria, manteniendo un contador interno de las referencias que existen en cada momento. Cuando el contador de referencias llega a 0, el punto `tmp` libera la memoria de los recursos que contiene. La clase `tmp` es similar a `std::shared_ptr` de la librería estándar de C++.

3.3.7. Macros

Las macros son instrucciones de preprocesado del código que permite implementar meta-código. Es decir, código que genera código. Las macros definen instrucciones que se sustituyen por código antes de la compilación, y permiten automatizar gran cantidad de tareas. Junto a las *templates*, son las dos herramientas de meta-programación que proporciona C++. Un ejemplo básico de macro es el siguiente:

```
1 #define MAX(a,b) a>b?a:b
```

Cada vez que en el código aparezca la instrucción `MAX`, se sustituirá por el operador ternario antes de la compilación. Este se trata de un ejemplo sencillo, pero es posible definir funciones más complejas o incluso clases, variables o cualquier tipo de código.

TypeName

La macro *TypeName*, se utiliza para dotar de un nombre a la clase dentro del propio código. Esto es utilizado para identificar las clases y reconocerlas por parte del usuario durante su uso. Para su funcionamiento se incluye en la zona pública de la clase la macro de la siguiente forma:

```
1 class className
2 {
```

```
3       ...
4   public:
5       TypeName("className");
6       ...
7   };
```

Y en el archivo .C se utiliza otra macro:

```
1   defineTypeNameAndDebug(className,0);
```

Como la variable que define el tipo debe ser estática, tiene que ser inicializada fuera de la clase. Con la macro *TypeName*, se define el nombre que se va a utilizar para la clase y mediante *defineTypeNameAndDebug* se inicializa la variable estática (compartida entre todas las instancias).

RunTimeTables

En programación orientada a objetos, es habitual la práctica de definir una clase base que sirva de interfaz y una serie de clases derivadas que implementen la funcionalidad concreta. La clase base, puede ser utilizada sin saber de forma concreta cual es el tipo derivado instanciado. Un ejemplo es la clase `airfoilModel` desarrollada. Esta define dos métodos virtuales puras:

```
1       virtual scalar cl(scalar aoa, scalar Re, scalar Ma) = 0;
2       virtual scalar cd(scalar aoa, scalar Re, scalar Ma) = 0;
```

Estas dos funciones han de ser implementadas por las clases derivadas. Al utilizar la clase, no es necesario saber cual es la implementación concreta de esas dos funciones, solo su comportamiento: reciben el ángulo de ataque, número de Reynolds y de Mach y devuelven el c_L y c_D del perfil.

La forma que tiene OpenFOAM para facilitar la selección de la implementación (clase derivada) es mediante *RunTimeTables*. Esto permite de forma sencilla y segura ante errores seleccionar en tiempo de ejecución la clase que se quiere instanciar en función de una cadena de texto (`word`), recibida generalmente como input por parte del usuario. Permite seleccionar entre una lista de clases derivadas de una clase base, cual es la que requiere ser instanciada para su uso. Para definir una *RunTimeTable*, son necesario varios elementos:

- Definir el nombre de las clases que vayan a intervenir mediante el método descrito en la Sección 3.3.7.
- Utilizar las macros *declareRunTimeSelectionTable* y *defineRunTimeSelectionTable* clase base.
- En las clases derivadas utilizar la macro *addToRunTimeSelectionTable*.

En la clase base, hay dos macros: una para la declaración de la *RunTimeTable* y otra para su implementación, ya que al ser *static* es necesario inicializarse fuera de la clase. Para la declaración se necesitan 5 argumentos: el tipo de puntero utilizado (habitualmente *autoPtr*), la clase base, el nombre de la tabla, los argumentos que recibe el constructor y la lista de los argumentos. Un ejemplo para la clase `airfoilModel` es el siguiente:

```
declareRunTimeSelectionTable
(
    autoPtr,                        //Tipo de puntero
    airfoilModel,                   //Clase base
    dictionary,                     //Nombre de la tabla
    (                               //Parametros del constructor
        word name,
        const dictionary& dict
    ),
    (name, dict)                    //Lista de los parametros
);
```

Y para la inicialización de la tabla se utiliza la siguiente macro fuera de la declaración de la clase:

```
defineRunTimeSelectionTable(airfoilModel,dictionary);
```

Siendo los parámetros requeridos la clase base y el nombre de la tabla.

Para añadir una nueva clase a la tabla, la clase tiene que declarar un constructor que acepte los mismos argumentos que los definidos en la *RunTimeTable* mediante el cuarto parámetro (parámetros del constructor). La macro para la adición debe utilizarse fuera de la clase y es la siguiente:

```
addToRunTimeSelectionTable
(
    airfoilModel,
    polarAirfoil,
    dictionary
);
```

Siendo los parámetros la clase base, la clase deriva y el nombre de la *RunTimeTable*.

El nombre que se le ha dado a la tabla (*dictionary*) es habitual atribuirlo a las tablas que son seleccionadas mediante entrada del usuario mediante diccionarios, aunque el nombre es arbitrario y puede escogerse con el criterio que se desee. Pueden definirse más de una tabla utilizando nombres distintos.

La selección del constructor concreto de la instancia se realiza utilizando la función:

```
<nombreTabla>ConstructorTable(typeName);
```

Donde <nombreTabla> es el nombre que se le ha dado a la tabla en la definición de la macro, que como se ha comentado es habitual que sea *dictionary*. A continuación se muestra el uso en un caso concreto donde se obtiene el constructor de la tabla y se utiliza para crear una nueva instancia:

```
auto* constructorPtr =
    dictionaryConstructorTable(typeName);

autoPtr<claseBase>
    instancia(contructorPtr(param1,...,paramN));
```

Este código se encuentra habitualmente en un método estático de la clase base con una definición similar a la siguiente:

```
autoPtr<claseBase> New
(
    const dictionary& dict,
    param1,
    ...,
    paramN
);
```

RunTimeTables y *templates*

Para el uso de las *RunTimeTable* con clases con `template`, existen una serie de macros equivalentes que permiten implementar esta funcionalidad. La declaración de la tabla se hace de forma idéntica al caso sin `template`, ya que se hace dentro de la propia clase. Para la inicialización de la *RunTimeTable* es necesario ciertas modificaciones, ya que se realiza fuera de la propia clase. En este caso, es necesario instanciar una tabla por cada tipo que vaya a ser utilizado como `template`. Además es necesario también instanciar el *TypeName* de cada tipo de `template` de la misma forma. Puede que haya otro método para la implementación, pero no ha sido lograda, ya que no está documentado su uso y se ha realizado una lectura extensiva del código para lograr implementarlo. Se define la instancia de la plantilla concreta para el tipo `vector` de la siguiente forma:

```
defineTemplatedRunTimeSelectionTable
(
    baseClass,  //clase base
    dictionary, //nombre de la tabla
    vector      //tipo para la template
);
```

Donde baseClass es la clase base, dictionary es el nombre de la tabla y vector el tipo concreto de la plantilla. Para la instanciación del TypeName con `template` se realiza con el siguiente código:

```
defineTemplateTypeNameAndDebugWithName
(
    baseClass<vector>, //clase base con template concreto
    "baseClass",       //nombre de la clase
    0
);
```

Donde `baseClass<vector>` es la clase base con tipo de `template vector` y `"baseClass"` es el nombre de la clase entre comillas (*string*).

Para añadir las clases derivada a la *RunTimeTable* correspondiente se realiza de forma similar al caso sin `template`, pero indicando el tipo concreto de `template` instanciado y utilizando la siguiente macro:

```
addTemplatedToRunTimeSelectionTable
(
    baseClass,       //clase base
    derivedClass,    //clase derivada
    vector,          //tipo del template
```

```
6       dictionary      //nombre de la tabla
7   );
```

Donde *baseClass* es la clase base, *derivedClass* la clase derivada, vector es el tipo concreto para el que se instancia la *RunTimeTable* y *dictionary* es el nombre de la tabla. Está permitido compartir tanto el *TypeName* como el nombre de la tabla entre diferentes argumentos de tipos de `template`. Es necesario definir el *TypeName* de cada tipo de `template` para poder incluirlo en la *RunTimeTable*.

Operators

En OpenFOAM hay una serie de clases llamadas *Operators*. Estas representan operadores matemáticos aplicados a un conjunto de datos. La utilidad reside en que utilizando estos operadores junto con funciones que admiten *template* se pueden implementar algoritmos genéricos que sirvan para cualquier tipo de operador que cumpla con los requerimientos. Algunos ejemplos de uso son para iterar sobre listas, aplicar operadores en programas en paralelo, etc. Existen distintas macros en función del tipo de operador y el número de parámetros, las macros definidas son las siguientes:

- `EqOp(opName,op)`: Admite dos parámetros `x` e `y` y modifica el valor de `x` según la operación definida. Devuelve `void`.
- `Op(opName,op)`: Admite dos parámetros `x` e `y` y devuelve el resultado de la operación. No modifica el valor de los parámetros.
- `BoolOp(opName,op)`: acepta dos parámetros `x` e `y` y devuelve un valor de tipo `bool`. No modifica el valor de los parámetros.
- `Bool1Op`: acepta un parámetro x y un valor de comparación durante la construcción de una instancia. Devuelve tipo `bool` y no modifica el parámetro.
- `WeightedOp(opName,op)`: acepta dos parámetros `weight` e `y`. Modifica el parámetro de entrada según una operación de ponderado. Devuelve tipo `void`.

Siendo *opName* el nombre asignado a la operación y *op* la definición de esta. Las macros definen estructuras que implementan el operador ( ), comportándose como funciones. La implementación del operador es la propia definida en el parámetro *op*. Al definirse los operadores como plantillas para uso género hay que indicar el tipo para el cual es utilizado mediante <>.

Coordinate systems

Las clases derivadas de `coordinateSystem` definen las transformaciones necesarias para cambiar posiciones y vectores entre distintos sistemas de referencia. Para un vector la transformación es una matriz de rotación. Para un vector que representa una posición, la trasformación consta de dos translaciones y una rotación. Existen dos clases derivadas de `coordinateSystem`: `cartesian` y `cylindrical`. La diferencia radica en que en coordenadas cartesianas las transformaciones son entre coordenadas globales cartesianas y

locales cartesianas y en cilíndricas las transformaciones son entre coordenadas globales cartesianas y locales cilíndricas.

El constructor de estas clases admite distintas formas, aunque la más habitual es especificando la posición del sistema de coordenadas y dos vectores, habitualmente los que indican las direcciones de los ejes x y z.

Los métodos que realizan las transformaciones tienen la misma definición en ambas clases y son los siguientes:

- **De coordenadas globales a locales:**
  - `vector localPosition(const point& global)`: transforma un vector que representa una posición (translación y rotación) de coordenadas globales a coordenadas locales. Devuelve el vector en coordenadas locales(cartesianas o cilíndricas).
  - `vector localVector(const vector& global)` transforma un vector (rotación) de coordenadas globales a coordenadas locales. Devuelve el vector en coordenadas locales (cartesianas o cilíndricas).
- **De coordenadas locales a globales:**
  - `vector globalPosition(const point& local)`: transforma un vector que representa una posición (translación y rotación) de coordenadas locales a coordenadas globales. Devuelve el vector en coordenadas globales (cartesianas).
  - `vector globalPosition(const vector& global)` transforma un vector (rotación) de coordenadas locales a coordenadas globales. Devuelve el vector en coordenadas globales (cartesianas).

Matrix

La librería de OpenFOAM incluye una gran cantidad de clases para el manejo de matrices, optimizadas a según el tipo de matriz. Algunos tipos de matrices implementadas son: *DiagonalMatrix*, *EigenMatrix*, *LduMatrix*, *LLTMatrix*, *QRMatrix* o *SquareMatrix*. Dependiendo del tipo de matriz y de problema que se vaya a resolver será necesario utilizar una clase o otra. En el desarrollo del trabajo solo se ha requerido el uso de *SquareMatrix* para resolver sistemas de ecuaciones lineales.

SquareMatrix

Se trata de la implementación de matrices densas cuadradas. Tiene implementada una función que permite resolver sistemas del tipo:

$$A \cdot x = b \tag{3.1}$$

Donde A es una matriz cuadrada de dimensiones NxN, x es un vector de incógnitas Nx1 y b es un vector de términos independientes Nx1. La función que permite resolver el sistema es la siguiente:

```
1     void solve(const SquareMatrix& A, Field& b);
```

La función devuelve el valor de x en el parámetro b. Para asignar los elementos de la matriz se utiliza el operador (i,j) donde i es el índice de la fila y j el de la columna.

Dimension Set

En algunos tipos de contenedores de OpenFOAM, sobretodo los enfocados a resolver ecuaciones de algún tipo o manipulaciones algebraicas, trabajan con variables con unidades definidas para mantener la consistencia de unidades y avisar de posibles errores. Las unidades se definen mediante una lista de exponentes, donde cada posición hace referencia a una magnitud fundamental del SI. En la Tabla 3.2 se muestra el índice de los exponentes de cada magnitud, el nombre que se le da dentro de OpenFOAM y las unidades utilizadas, según el SI.

Índice	Propiedad	dimensionType	Descripción SI
0	Masa	MASS	Kilogramo
1	Longitud	LENGTH	Metro
2	Tiempo	TIME	Segundo
3	Temperatura	TEMPERATURE	Kelvin
4	Cantidad de materia	MOLES	Mol
5	Corriente electrica	CURRENT	Amperio
6	Intensidad Lumínica	LUMINOUS_INTENSITY	Candela

Tabla 3.2: Magnitudes fundamentales utilizadas y nombre de referencia dentro de OpenFOAM.

Además es posible combinarlos utilizando los distintos operadores, multiplicación, división y exponenciales. También vienen definidos algunas magnitudes que suelen utilizarse de manera habitual como variables globales constantes. Algunas magnitudes habituales y sus nombres asociados se muestran en la Tabla 3.3, aunque existen un gran numero de estas además de las mostradas.

Dimensión	Variable
Adimensional	dimless
Masa	dimMass
Longitud	dimLength
Temperatura	dimTemperature
Area	dimArea
Volumen	dimVol/dimVolume
Density	dimDensity

Tabla 3.3: Ejemplo de algunas magnitudes instanciadas como variables `const`

Time

El control principal de un programa de OpenFOAM se realiza mediante la clase *Time*. Esta clase está encargada del control del tiempo de cómputo, de los instantes temporales, del control de la escritura y lectura, el formato y es el objeto del `objectRegistry` de

mayor nivel en la jerarquía. El archivo encargado de su configuración es el *controlDict*. Presenta una amplia funcionalidad en distintos ámbitos. A continuación se muestra una pequeña selección de algunas funciones utilizadas:

- `fileName path()`: ruta del caso.
- `fileName timePath()`: ruta de la carpeta para el tiempo actual de simulación.
- `bool writeTime()`: devuelve `true` si se requiere guardar los datos.
- `scalar deltaTValue()`: devuelve el valor del incremento de tiempo entre la iteración actual y la anterior.

3.3.8. Volúmenes finitos

En esta sección del código se comentarán los elementos principales donde se almacenan los datos del *solver* de volúmenes finitos y algunas herramientas de uso. En primer lugar se comentarán el acceso a los datos de la malla, como se almacenan los campos y herramientas de selección y interpolación.

fvMesh

La clase `fvMesh` representa la malla de volúmenes finitos, es decir la discretización del dominio y del contorno. Contiene todos los datos que permiten definir la malla y herramientas de I/O. A continuación, se comentan los métodos más importantes utilizados. Los datos de la malla se obtienen mediante referencias a listas de valores donde el índice de la lista es el índice de la celda/cara/arista/vértice que corresponda según el tipo de dato. Los métodos para acceso de lectura a los datos geométricos de la malla son los siguientes (indicado entre paréntesis el tipo de índice de acceso: cell, face, edge o vertex):

- `const volVectorField& C()` : devuelve una lista con los centroides de las celdas (cell based).
- `const volScalarField& V()` : devuelve una lista con el volumen de las celdas (cell based).
- `const surfaceVectorfield& Cf()`: devuelve una lista con los centroides de las caras (face based).
- `const surfaceVectorField& Sf()`: devuelve una lista con los vectores normales de las caras donde la magnitud del vector es el área de la cara (face based).
- `const surfaceScalarField& magSf()`: devuelve una lista con el área de las caras (face based).
- `const vectorField& points()`: devuelve una lista de los vértices de la malla (vertex based).
- `const edgeList& edges()`: devuelve una lista de las aristas (edge based).

Otro tipo de funciones de acceso de lectura son las que indican la conectividad de la malla, y son las siguientes:

- `const labelListList& cellCells()` : devuelve una lista de listas de índices de las celdas vecinas (cell based).
- `const labelListList& edgeCells()` : devuelve una lista de listas de índices de las celdas a las que pertenece la arista (edge based).
- `const labelListList& pointCells()` : devuelve una lista de listas de índices de las celdas que pertenecen al vértice (vertex based).
- `const cellList& cells()` : devuelve una lista de celdas (cell based).
- `const labelListList& edgeFaces()` : devuelve una lista de listas de índices de las caras que comparten la arista (edge based).
- `const labelListList& pointFaces()` : devuelve una lista de listas de índices de las caras que comparten el vértice (vertex based).
- `const labelListList& cellEdges()`: devuelve una lista de las listas de índices de las aristas que forman las celdas (cell based).
- `const labelListList& faceEdges()`: devuelve una lista de las listas de índices de las aristas que forman las caras (face based).
- `const labelListList& pointEdges()` : devuelve una lista de listas de índices de las aristas que comparten el vértice (vertex based).
- `const labelListList& pointPoints()` : devuelve una lista de listas de los vértices a los que está conectado el vértice (vertex based).
- `const labelListList& cellPoints()` : devuelve una lista de listas de los índices de los vértices que forman parte de la celda (cell based)

Además de las funciones de acceso, existen otro tipo de funciones de consulta como la siguiente:

```
1 label findCell(const point& location)
```

Que devuelve el índice de la celda que contiene el punto *location*, y -1 si está fuera del dominio.

cellPointWeights y interpolationCellPoint

Para realizar interpolación dentro del dominio, OpenFOAM dispone de dos clases que trabajan en conjunto: `interpolationCellPoint` y `cellPointWeights`. En primer lugar se crea una instancia de `cellPointWeights`, la cual contiene los pesos de la interpolación y es el paso más costoso computacionalmente. Para instanciarla, es necesario una referencia a la malla FV (`fvMesh`), las coordenadas globales del punto y la celda que contiene el punto. Para obtener el ultimo parámetro es de utilidad el método mencionado en la sección anterior: `findCell(const point& location)`.

Para realizar la interpolación del campo fluido, se utiliza la clase con el nombre de `interpolationCellPoint`, que admite el tipo de campo como argumento de `template` y el campo a interpolar como argumento del constructor. Para obtener el valor interpolado en la celda se utiliza el método de las clase `interpolationCellPoint` definido de la siguiente forma:

```
1 fieldType interpolate(cellPointWeights* weights)
```

GeometricField

Para relacionar un conjunto de datos a una geometría, OpenFOAM utiliza la clase *GeometricField*, que abstrae el tipo de datos utilizado y la geometría sobre la que se aplica. Permite mantener una fuerte relación entre las lista de datos y la geometría. Admite tres argumentos de plantilla: el tipo de dato, una plantilla al tipo de contorno y el tipo de malla. En función del tipo de campo asignado y el tipo de geometría se crean diferentes *typedef* para facilitar el uso. El *GeometricField* utilizado para valores escalares de celda en una malla de volúmenes finitos es:

```
1 typedef GeometricField<scalar,fvPatchField,volMesh>
2     volScalarField;
```

El acceso a los campos puede realizarse mediante los métodos *boundaryField()* y *internalField()* para acceder a los valores internos y del contorno.

Finite Volumes options

En esta sección, se comentarán las dos principales clases que forman la base para la implementación de métodos de termino fuente: *fv::option* y *fv::options*.

fv::option

Esta es la clase base de la cual se crea una clase derivada para añadir los términos fuente a las ecuaciones. La clase `fv::option` presenta una variedad de métodos virtuales (no virtuales puros) que pueden ser sobrescritos por las clases derivadas. Estos métodos son llamados durante la resolución de las ecuaciones para la adición de los términos fuente requeridos. El método es llamado `addSup` y tiene diferente definición para cada tipo de sistema y de problema. Existen versiones para ecuaciones de escalares, vectores y tensores; y para cada una de ellas versiones para flujo incompresible, compresible y multifásico. Para el caso del campo de velocidades de N-S las definiciones son:

```
1 virtual void addSup
2 (
3     fvMatrix<vector>& eqn,       //Eq discretizadas velocidad
4     const label fieldi           //Indice del campo
5 );
6 virtual void addSup
7 (
8     const volScalarField& rho,   //Campo de densidades
9     fvMatrix<vector>& eqn,       //Eq discretizadas velocidad
```

```
10      const label fieldi          //Indice del campo
11  );
```

Donde se muestran los dos métodos que han sido sobrecargados en el presente trabajo: la ecuación de la velocidad para flujo incompresible y compresible.

Para acceder al `volVectorField` del campo de velocidades, se utiliza el método:

```
1       const volVectorField& fvMatrix<vector>::psi();
```

Que devuelve una referencia constante al campo de velocidades. Para otro tipo de ecuaciones, devolvería el campo correspondiente: escalar, vectorial o tensorial.

Para añadir un termino fuente explicito a las ecuaciones, se utiliza el operador `+=` o el operador `-=`. Los términos fuente han de estar almacenados en una clase del tipo `GeometricField`.

fv::options

Esta clase es la encargada del manejo automático y la instanciación de las clases a partir del archivo fvOptions, localizado dentro de la carpeta *system* del caso a resolver. La clase *fv::options* busca todos los diccionarios dentro del archivo *fvOptions* y crea una instancia de la clase derivada de `fv::option` que se requiera, según el nombre indicado con el atributo `type` dentro del diccionario de cada `fv::option`.

Para que sirva la automatización, la clase derivada tiene que añadirse a la *RunTimeTable* de `fv::option` según el procedimiento indicado en la Sección 3.3.7. A continuación se muestra un ejemplo de como realizarlo:

```
1  namespace Foam
2  {
3      namespace fv
4      {
5          addToRunTimeTable(option,fvSource,dictionary);
6      }
7  }
```

En este caso el nombre *dictionary* es obligatorio, puesto que es el nombre de la tabla utilidad por `fv::options`. El nombre de la clase añadida en cuestión ha sido nombrada como*fvSource* en el ejemplo.

3.3.9. Paralelizado

A continuación, se comentarán las herramientas de código disponibles para programación de programas ejecutados en paralelo. OpenFOAM trabaja con *wrappers* sobre el código de la libreria OpenMPI.

Pstream

Pstream o *Paralel Stream* es una clase encargada de la salida de texto por terminal en paralelo. Ademas cuenta con ciertos métodos que dan información sobre la paralelización:

- `label nProcs()`: devuelve el número de procesos de la simulación.
- `label myProcNo()`: devuelve el número de proceso del hilo desde el que es llamado.

reduce

Uno de los métodos de comunicación entre procesos es el uso de la función *reduce*. Esta función admite un tipo de dato y una operación (del tipo EqOp definido en la Sección 3.3.7). Combina el valor de la variable de todos los procesos mediante la operación definida. Por ejemplo, para el tipo entero y la operación suma, *reduce* sumará el valor de la variable de cada proceso, y en cada proceso se obtendrá el valor de la suma de todos ellos. El código sería como el siguiente:

```
1 label a = Pstream::myProcNo(); //la variable a toma el
2                                //valor del indice
3                                //de cada proceso
4 reduce(a,sumOp<label>());      //suma el valor de cada proceso
5
6 Info<<a<<endl;                 //Para 4 procesos a toma el valor
7                                //a = 0 + 1 + 2 + 3 = 6
```

3.3.10. Útil

En esta sección se engloban distintas utilidades varias que han sido de utilidad para la realización del proyecto.

ijkAddressing

Esta clase define la transformación de índices unidimensionales a tridimensionales y la inversa según la Ecuación 2.54 y la Ecuación 2.55 para $ND = 3$. Sirve también para transformaciones en dos dimensiones, indicando que el tamaño de la tercera dimensión es 1. El tamaño de cada dimensión se indica durante la construcción de la clase. El método para obtener el índice unidimensional a partir de los tres índices es:

```
1     label index(const labelVector& ijk)
```

Y para obtener los tres índices a partir del índice unidimensional:

```
1     labelVector index(const label idx)
```

Constantes

OpenFOAM define en diversos archivos, agrupados en *constants.H*, las distintas constantes relevantes para los cálculos. Las constantes son clasificadas en varios tipos:

- Matemáticas: definidas en *mathematicalConstants.H*, como por ejemplo el número π o el número e.

- Fundamentales: definidas en *fundamentalConstants.H*, como por ejemplo la velocidad de la luz en el vacío o la constante de Planck.
- Universales: definidas en *universalConstants.H*, en esta sección solo se encuentra la constante reducida de Planck.
- Electromagnéticas: definidas en *electromagneticConstants.H*, como por ejemplo la impedancia del vacío o la permeabilidad magnética del vacío.
- Atómicas: definidas en *atomicConstants.H*, como por ejemplo la constante de estructura fina o el radio de Bohr.
- Físico-Químicas: definidas en *physicoChemicalConstants.H*, como por ejemplo la constante de los gases ideales o la constante de Faraday.
- Termodinámicas: definidas en *thermodynamicConstants.H*, como por ejemplo la presión estándar o la temperatura estándar.

Conversión de unidades

En OpenFOAM se incluye una serie de funciones que realizan la conversión de unidades más habituales, como por ejemplo de radianes a grados, de atmósferas a Pascales o de revoluciones por minuto (RPM) a radianes por segundo:

- `scalar degToRad(const scalar deg)`
- `scalar radtoDeg(const scalar deg)`
- `scalar rpmToRads(const scalar deg)`
- `scalar radsToRpm(const scalar deg)`
- `scalar atmToPa(const scalar deg)`
- `scalar paToAtm(const scalar deg)`

Capítulo 4

Implementación y código

El capitulo actual pretende responder a la cuestión de como se ha realizado la implementación del código, su integración en OpenFOAM y el diseño de una arquitectura modular, reutilizable y ampliable. Se entrará en detalle de la funcionalidad de las clases desarrolladas y su interacción entre ellas. Además, se justificará el enfoque utilizado para el paralelizado junto con sus ventajas e inconvenientes y el modo de implementación.

4.1. Arquitectura y abstracción

Uno de los objetivos del trabajo era realizar una generalización del acople con volúmenes finitos para la implementación de modelo de teoría de elemento de pala mediante la adición de los términos fuente como disco actuador. No obstante, la abstracción se ha llevado más allá, permitiendo incorporar otras teorías para el cálculo de las fuerzas sobre el fluido. Por ello, en la presente sección se hace referencia a ***propeller source*** como abstracción de un método genérico para calcular las fuerzas de un rotor, incluyéndose como uno de ellos la teoría de elemento de pala.

Las ventajas de un enfoque basado en la abstracción radican en la posibilidad de separar el código en bloques independientes que se comunican entre sí, lo cual hace el código más fácil de entender y de desarrollar. Aunque en un principio el proceso de abstracción resulta complejo y largo, es una inversión a largo plazo. Sin una buena arquitectura del código, el desarrollo se complica a medida que aumenta el tamaño del código.

La arquitectura del código consiste en definir la estructura general de los elementos y como se relacionan entre sí. La arquitectura se divide en diferentes niveles o capas que a la vez interactúan entre sí. Los dos niveles que se tratan son el acople de volúmenes finitos con el *propeller source* y el cálculo del *propeller source* de manera que disponga de una amplia versatilidad. En la Figura 4.1 se observa el esquema general del código para el acople y el cálculo de los términos fuente. En las siguientes secciones se explicara con detalle todos los elementos y su interacción entre ellos.

4.1.1. Acople Volúmenes Finitos - *Propeller Source*

El nivel superior de la arquitectura implementada es la separación entre volúmenes finitos y *propeller source*. Para ello es necesario definir cuales son los puntos comunes y la

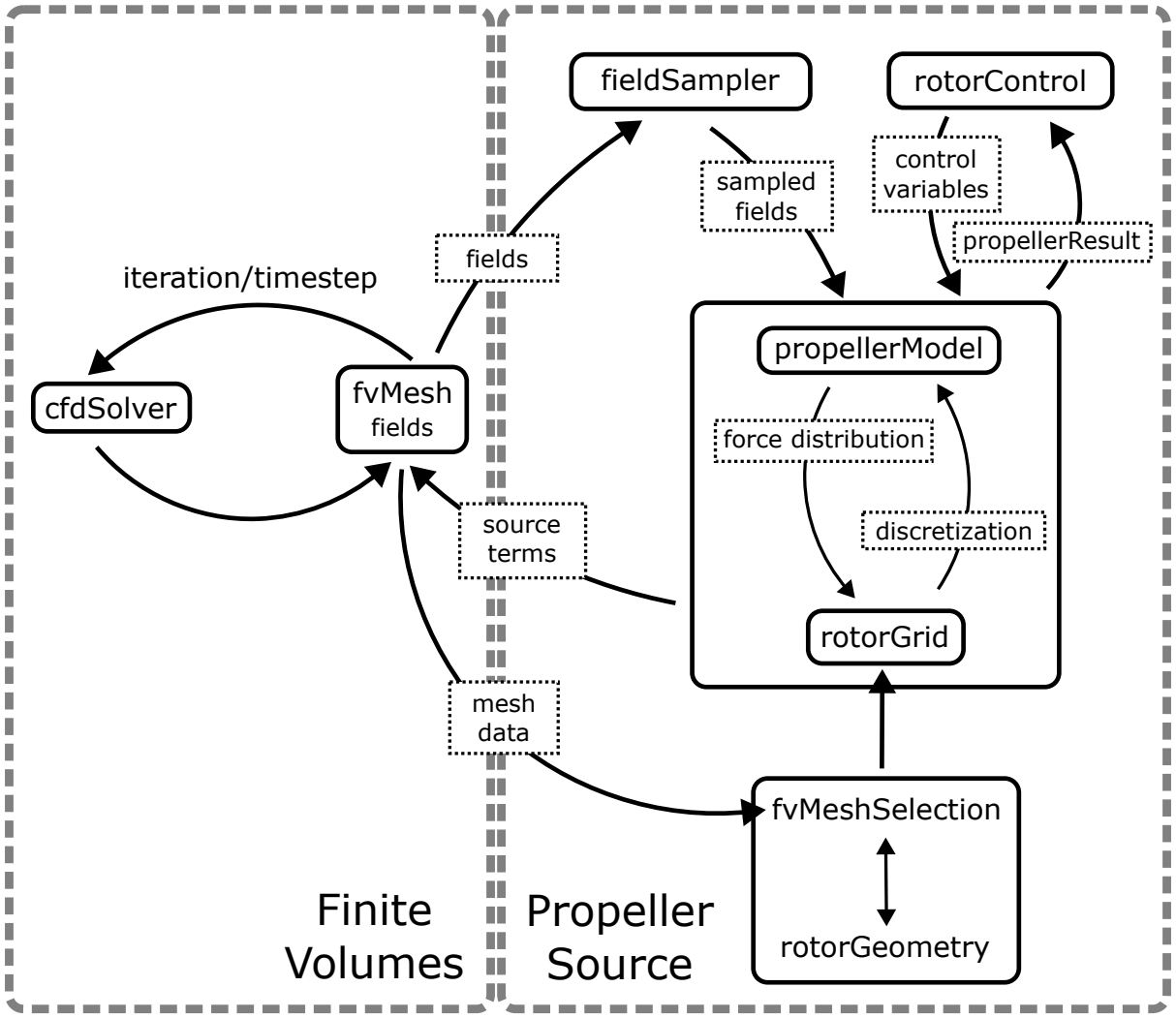

Figura 4.1: Esquema de la arquitectura del código

información que necesitan compartir entre ellos. Para ello se realizan unas hipótesis de partida que han de cumplirse para poder utilizar la arquitectura descrita:

- La solución de volúmenes fínitos puede obtenerse de forma desacoplada a la de *propeller source*. Es decir, los términos fuente son **explícitos**.
- La solución de *propeller source* solo depende de la de volúmenes finitos a través de los campos de velocidad y densidad y de la malla de volúmenes finitos.

Las hipótesis son relativamente holgadas y podrían ser modificadas de forma sencilla si nuevas implementaciones lo requieren. Estas hipótesis permiten restringir la comunicación entre el bloque de volúmenes finitos y el de *propeller source* a tres enlaces concretos:

- Enlace 1: obtención de la geometría de la malla para la construcción de la geometría del rotor, la selección de elementos finitos y la malla del rotor (`rotorGrid`).
- Enlace 2: adición de los términos fuente a las ecuaciones de momento.
- Enlace 3: Obtención de los campos de velocidad y densidad en el rotor.

rotorFvMeshSel

Para el acople de volúmenes finitos y el *propeller source* (o acople FV-PS para abreviar), se realiza una selección de las celdas que pertenecen al rotor para reducir el número de elementos del dominio a un subconjunto de ellos. La `rotorFvMeshSel` tiene un acople directo con la geometría, pudiendo especificarse la geometría y seleccionar los elementos de la malla o indicar cuales son los elementos que pertenecen al rotor y inferir la geometría a partir de estos. Establece el primer enlace con volúmenes finitos.

rotorGrid

El `rotorGrid` es la discretización del *propeller source*, define los elementos sobre los que se realizan los cálculos. Además establece la conexión con volúmenes finitos, siendo el encargado de añadir los términos fuente. Para ello cada celda del `rotorGrid` va asociada a una o más celdas de volúmenes finitos, con unos pesos que determinan como se distribuye y un factor de escalado. La forma en la que se obtienen estos valores depende del tipo de *grid*. Por lo tanto, establece el segundo enlace.

fieldSampler

El `fieldSampler` es el encargado de proveer los valores de velocidad y densidad del campo fluido a cada uno de las celdas del `rotorGrid`. Establece el tercer enlace FV-PS.

4.1.2. Arquitectura del *propeller Source*

En la arquitectura de *propeller Source* distinguimos dos vías principales: la inicialización y el cálculo de los términos fuente. Ambas vías son bastante lineales. Para el caso de la inicialización se realiza en el siguiente orden:

1. Se crea la instancia de `propellerModel`
2. Se lee la entrada de la geometría mediante el diccionario.
3. Se lee la entrada de la `rotorFvMeshSel` mediante el diccionario.
4. Se Crea la selección de la malla a partir de la geometría y/o se modifica la geometría a partir de la selección de la malla.
5. Se comprueba que la geometría obtenida es válida.
6. Se construyen los sistemas de coordenadas.
7. Se crea el `rotorGrid` con la geometría final.
8. Se crean los `fieldSamplers`.

Las instancias concretas creadas dependen de la configuración del usuario mediante los diccionarios correspondientes y es explicado en la Sección 5.2. La inicialización es realizada en el método void read(const dictionary& dict) de la clase `propellerSource` que deriva de `fv::option` para su integración automática dentro de `fv::options`. El método anterior es llamado en el constructor de la clase.

El cálculo de los términos fuente se implementa en la función `addSup`, que es llamada por la clase `fv::options` de forma automática cuando se requiera la adición de los términos fuente, es decir, antes de resolver la iteración de las ecuaciones de N-S. El proceso para calcular los términos fuente es el siguiente:

1. Inicializar a 0 el campo de términos fuente a añadir.
2. Actualizar el instante temporal del `propellerModel`.
3. *Samplear* los campos de velocidades y densidad.
4. Corregir los parámetros de control.
5. Calcular las fuerzas aerodinámicas en las celdas del `rotorGrid`.
6. Añadir los términos fuente a las ecuaciones.

rotorControl

Esta clase abstracta se encarga de proporcionar las variables de control necesarias a la clase `propellerModel`. En la sección Subsección 4.2.6 se comentará en detallé su implementación.

propellerResult

La clase `propellerResult` contiene los resultados integrales del rotor, es decir: el vector fuerza, el vector momento, la potencia, el paso y los coeficientes de empuje, par y potencia. Su implementación es analizada en detalle en Sección 4.2.5.

4.2. Clases

En la presente sección se analizará desde un punto de vista técnico el código desarrollado. Por tanto es necesario hasta cierto punto un conocimiento de programación y en concreto de C++. Se mencionarán conceptos relacionados con programación orientada a objetos como: clases, herencia, métodos virtuales, clases abstractas, polimorfismo. También elementos de programación genérica como es el caso de las plantillas. Todo esto puede consultarse en el libro `The C++ Programming Language` de Bjarne Stroustrup [27], fundador del lenguaje de programación C++.

El código completo se presenta como un documento a parte dada su extensión. Este se organiza en carpetas, donde el nombre de la carpeta coincide con el de la clase base abstracta si existe, o por su defecto la clase que contiene. Podría haberse realizado una separación distinta, de forma conceptual, pero dada la extensión no tan elevada del código (comparándolo con grandes librerías, como OpenFOAM), tan solo complicaría su lectura. La única excepción es la carpeta `util`, que contiene diversas utilidades comunes para el resto del código.

4.2.1. airfoilModel

La clase abstracta `airfoilModel`, abstrae las características de un perfil aerodinámico 2D. Aunque en el presente trabajo esta clase abstracta es utilizada para la implementación del BET, es una clase genérica desacoplada de este método y podría utilizarse para cualquier tipo de código que requiera el uso de perfiles aerodinámicos.

Esta clase presenta dos métodos virtuales puros que han de ser implementados por las clases derivadas:

```
1 virtual scalar cl(scalar aoa, scalar reynolds, scalar mach) = 0;
2 virtual scalar cd(scalar aoa, scalar reynolds, scalar mach) = 0;
```

Los dos métodos presentan la dependencia de los coeficientes aerodinámicos en función del ángulo de ataque, el número de Reynolds y el de Mach. No es necesario que esa dependencia exista en las implementación, pero se construye así para que sea compatible con cualquier modelo.

El constructor permite asignar un nombre al `airfoil` para ser seleccionado de una lista por el resto de clases. Admite también un diccionario que utiliza como input para una serie de atributos comunes a todos los perfiles aerodinámicos que permite exportarlos de forma automática a formato `csv`. La exportación permite el uso de la interpolación, indicando los ángulos de ataque, Reynolds y Mach para los que se interpolan y exportan los valores de c_L y c_D. La clase `airfoilModel` no está pensada para contener datos geométricos del perfil utilizado, tan solo sus polares adimensionales.

En esta clase abstracta, se define el método estático:

```
1 autoPtr<airofilModel> New(word name, const dictionary& dict);
```

El cual hace uso de las `RunTimeTables` para permitir al usuario seleccionar el tipo de `airfoilModel` que necesite. Su uso es explicado en Sección 3.3.7. Para crear una clase derivada de `airfoilModel` y que sea utilizable por el usuario debe cumplir con los siguientes requisitos:

1. Derivar de la clase `airfoilModel`.
2. Implementar los métodos cl y cd.
3. Incluirse en la `RunTimeTable` de `airfoilModel` con nombre `dictionary`.

A continuación, se comentan las clases derivadas que ya han sido implementadas para la realización del trabajo.

cylinderAirfoil

Esta es la implementación más básica de `airfoilModel` y tiene la utilidad de que es muy sencilla de utilizar por parte del usuario para el uso de perfiles cilíndricos. Presenta un solo parámetro de entrada que es el c_{D0} del cilindro, por lo que $c_D = c_{D0}$ y el $c_L = 0$. En el BET este tipo de perfiles suele utilizarse en la zona de la raíz para modelar la parte sin pala.

simpleAirfoil

Esta implementación es también bastante básica, aunque presenta cierta utilidad. Utiliza las ecuaciones de placa plana (Ecuación 2.45 y Ecuación 2.46) para el cálculo del c_L y el c_D.

interpolatedAirfoil

Esta clase implementa una utilidad para crear nuevos perfiles como interpolación de dos existentes. Para ello se indica el nombre de los perfiles que se quiere combinar y los pesos por los que se multiplica cada uno. El resultado es un perfil interpolado que puede exportarse para generar polares interpoladas. La clase deriva tanto de `airfoilModel` como de `Interpolated<scalar,const airfoildModel*>`, para extender la utilidad de la interpolación.

polarAirfoil

De todas las anteriores, esta es la implementación más importante y versátil. Permite definir un perfil aerodinámico como un conjunto de polares para cada Reynolds y cada Mach. Para el cálculo del c_L y c_D se realiza primero un interpolación genérica de las polares en función del número de Mach y Reynolds, utilizando la clase `Interpolated<scalar, const polar*>`. Esta clase, permite expresar la polar interpolada como una lista de punteros a polares y una lista de coeficientes. Para obtener los valores concretos, la clase `Interpolated` permite utilizar funciones `lambda` para la interpolación, por lo que el c_L y c_D se reconstruye multiplicando los coeficientes anteriores por los valores de c_L y c_D de cada polar de la lista de interpolación.

La clase `polarAirfoil` se encarga de leer los datos de las polares de tablas tipo `csv`. Presenta gran versatilidad en la forma de introducir los datos, pudiendo extraerse de un `csv` o de diversos `csv`. Estos datos se guardan en instancias de la clase polar, que contienen los datos c_L y c_D para un Reynolds y Mach concreto. Los datos son

almacenados en estructuras de datos que permiten la interpolación (linear o *spline* cúbica) y la extrapolación: constante o Viterna y placa plana, cuyas ecuaciones se describen en la Sección 2.2.

4.2.2. airfoilModelList

Esta clase concreta se encarga de crear una lista con todos las instancias de la clase `airfoilModel` indicadas en el diccionario. La clase extiende de `PtrList<airfoilModel*>`, gestionando internamente el manejo de la memoria. Presenta un método, para obtener un puntero constante al perfil a través del nombre de este, buscando en la lista interna.

El uso fundamental que se le da a esta clase es como contenedor de las instancias de los `airfoilModel` creados, para luego proporcionar acceso a estos a las clases que lo requieran, como `bladeModel`, comentada a continuación.

4.2.3. bladeModel

La clase `bladeModel` es una clase concreta, ya que no se ha necesitado abstraer. Esta clase define la pala del BET mediante la geometría (cuerda, torsión geometrica y flecha) y los perfiles aerodinámicos. Internamente utiliza una tabla de interpolación para cada variable en función del radio. La pala se puede definir de dos formas distintas:

- A partir de secciones, donde se indica: la cuerda, torsión geométrica, flecha y perfil aerodinámico utilizado para cada sección radial.
- A partir de tablas independientes para cada unos de los parámetros comentados. Esto le da más versatilidad ya que pueden definirse para posiciones radiales diferentes.

La lista de `airfoils` se especifica mediante una lista de los nombres de estos, que se buscan en la clase `airfoilModelList` para obtener un puntero a la instancia del `airfoilModel` requerido. La clase realiza una comprobación de que todos los parámetros introducidos son válidos y avisa al usuario de cuales son erróneos. Comprueba que la listas de radios introducidos son monótonamente crecientes y que los valores de los ángulos para la torsión se sitúen entre $[-\pi/2, \pi/2]$. Además permite escoger entre interpolación lineal o *spline* cúbica.

Esta clase incluye dos métodos que nos permiten obtener la interpolación de la geometría para un radio concreto o de la geometría y de los `airfoils`. El siguiente método permite obtener la geometría de la pala:

```
1 bool geometryAtRadius
2 (
3     scalar adimRadius,
4     scalar& chord,
5     scalar& twist,
6     scalar& sweep
7 ) const;
```

Donde `adimRadius` es el radio adimensional, entre 0,0 y 1,0 definido como r/R_{max}. La geometría se obtiene pasando las variables donde se guardan como referencia, siendo estas

modificadas. Devuelve `true` si `adimRadius` está entre 0,0 y 1,0. Para el caso de querer obtener también la interpolación de los `airfoil` se utiliza el método siguiente:

```
bool sectionAtRadius
(
    scalar adimRadius,
    scalar& chord,
    scalar& twist,
    scalar& sweep,
    interpolatedAirfoil& airfoil
) const;
```

Esta función devuelve el `airfoil` interpolado a través de la referencia como parámetro.

4.2.4. InterpolationTable

La clase `InterpolationTable` es una clase abstracta que representa una interpolación genérica. Se ha realizado con `template`, admitiendo 3 argumentos:

- `typeIn`: tipo de dato de entrada en la tabla de interpolación, es decir, los puntos donde se interpola. Habitualmente el tipo es `scalar`.
- `typeOu`: tipo de dato de salida en la tabla de interpolación, es decir, el tipo de datos interpolado. Puede ser cualquier tipo genérico.
- `dim`: dimensión de la interpolación. Se trata de un número natural que indica el número de variables de entrada necesarias para realizar la interpolación.

Como ejemplo, el tipo utilizado para la interpolación de polares es:

```
    InterpolationTable<scalar,const polar*,2>;
```

El tipo de entrada es escalar, ya que las polares se interpolan en número de Reynolds y número de Mach, por tanto la dimensión de la interpolación es 2 y el tipo de variable de salida es un puntero constante a una polar.

Los métodos virtuales puros que contiene son mostrados a continuación. En primer lugar:

```
virtual Interpolated<typeIn,typeOu> interpolate
(
    const FixedList<typeIn,dim>& input
) const = 0;
```

Esta función devuelve la interpolación mediante un objeto de la clase `Interpolated`, que abstrae el concepto de interpolación y es comentado en detalle en la Sección 4.2.4. Es el método que ha de implementarse según el tipo de interpolación realizada. El método recibe como parámetro un lista de *inputs* de tamaño fijo e igual al tamaño de la dimensión. Esto ha sido realizado de esta forma para que los errores de este tipo sean detectados en tiempo de compilación y sea de tipado seguro, evitando errores en *run time*.

```
virtual bool setRawData
(
    const List<List<typeIn>>& inputs,
    const List<typeOu>& outputs
) = 0;
```

Este método permite definir los puntos de interpolación conocidos y sus variables. Ha de implementarse según el tipo de interpolación. Devuelve `true` si los datos son validos para la instancia de la interpolación utilizada.

```
virtual label size() = 0;
```

Este método define como se calcula el número de elementos que contiene la interpolación, ya que no es igual en cada tipo de interpolación. Derivando de esta clase, se encuentran dos tipos abstractos: `RegularInterpolation` y `IrregularInterpolation`. Esto se hace para crear la distinción a nivel de código, y crear la restricción en el tipo de datos mediante la herencia. Esto es útil para casos en los que se requiera explícitamente un tipo u otro.

Interpolated

Esta clase abstrae el concepto de interpolación y expresa un objeto interpolado como una lista de objetos y otra de coeficientes. Se trata de una clase concreta que admite dos argumentos de `template` en su definición:

1. `typeIn`: es el tipo de los coeficientes de interpolación.
2. `typeOu`: es el tipo de los objetos interpolados.

Presenta un método `value` que admite una función como parámetro que calcula el valor interpolado. Por defecto se calcula como un sumatorio de los objetos por los coeficientes. Esto es valido cuando los objetos y los coeficientes son numéricos, o se define la operación multiplicación para ellos. En su defecto se define la función que indica como se reconstruye la interpolación. La definición del método `value` es la siguiente:

```
template<class typeRet = typeOu>
typeRet value(std::function<typeRet(typeIn,typeOu)> aggregate=
[](typeIn a, typeOu b){return a*b;}) const
{
    typeRet acc = Zero;
    for(label i = 0; i<coeff.size();i++)
    {
        acc += aggregate(coeff[i],nodes[i]);
    }
    return acc;
}
```

Se observa que el método `value` es en sí una plantilla, con el tipo de valor de retorno, pero que por defecto es del tipo `typeOu` si no se indica nada. El parámetro de entrada `aggregate` es un puntero a una función que admite dos parametros de entrada: `typeIn` y `typeOu`. Esta función es ejecutada para cada par de objeto-coeficiente, y el valor devuelto es sumado

a la variable `acc`, que acumula la suma. Indicando como se realiza esta reconstrucción mediante la función `aggregate` o definiendo el operador multiplicación, se puede trabajar con tipos de datos más complejos.

Se usa de ejemplo el caso de la interpolación de polares, ya que es bastante representativo de la funcionalidad de esta clase. El método `interpolate` de la `InterpolationTable` utilizada para las polares devuelve un objeto del tipo: `Interpolated<scalar,const polar*>`. Para reconstruir el c_L y el c_D interpolado se define la función `aggregate` mediante una función anónima o `lambda` de la siguiente forma para el caso del c_L:

```
1 [=](scalar val,polar* p){return val*p->cl(alfaRad);}
```

Y de esta para el caso del c_D:

```
1 [=](scalar val,polar* p){return val*p->cd(alfaRad);}
```

La función se ejecutará para cada polar de la interpolación, devolviendo el valor del c_L o c_D multiplicado por cada coeficiente, sumándose entre ellos en el método `value` y devolviendo el valor interpolado. La utilidad de esta clase es permitir una generalización de la interpolación a cualquier tipo de datos, como estructuras o clases más complejas, permitiendo la reutilización del código.

RegularInterpolation

Esta clase abstracta derivada de InterpolationTable representa los métodos de interpolación estructurados, comentados en la Sección 2.3.1. Derivando dos clases de esta clase abstracta, se implementa el método de interpolación linear (Sección 2.3.1) y la *spline* cúbica (Sección 2.3.1). Este último método se implementa tan solo para tipos escalares en una dimensión.

Para la construcción de una instancia de `LinearInterpolation`, que define el método de interpolación linear multidimensional, hay varias opciones:

- Mediante una lista de listas de los valores en cada dimensión que definen los puntos conocidos como una permutación de estos y una lista de los valores conocidos. Además, es posible indicar el orden en el que se iteran las dimensiones, mediante una lista de índices. Por defecto la iteración es de izquierda a derecha.
- Mediante una lista de listas de todos los puntos conocidos y los valores en estos puntos.

Estas dos formas se entienden mejor con ejemplos. Para el caso 2D el siguiente input generaría la Tabla 4.1 para los dos tipos de ordenaciones posibles en 2D:

```
1
2 LinearInterpolation<scalar,scalar,2> linInterp
3 (
4     {{1,2},{1,2}},
5     {x1,x2,x3,x4},
6     {0,1}
7 );
8 LinearInterpolation<scalar,scalar,2> linInterp
```

```
(
    {{1,2},{1,2}},
    {x1,x2,x3,x4},
    {1,2}
);
```

Se observa que los puntos en los que se tienen los datos son las permutaciones de dos elementos entre $1, 2$ y $1, 2$

orden (0,1)		orden (1,0)	
input	output	input	output
(1,1)	x1	(1,1)	x1
(2,1)	x2	(1,2)	x2
(1,2)	x3	(2,1)	x3
(2,2)	x4	(2,2)	x4

Tabla 4.1: Definición del orden y de las entradas salidas para la interpolación estructurada.

El segundo método se ejecutaría de la siguiente forma para el mismo ejemplo:

```

LinearInterpolation<scalar,scalar,2> linInterp
(
    {{1,1},{1,2},{2,1},{2,2}},
    {x1,x2,x3,x4}
);
```

Indicando para cada punto el valor que le corresponde. Aunque más inmediato de utilizar, es necesario tener en cuenta que los puntos indicados realmente forman un conjunto estructurado, en caso contrario será un error. Para solventar esto se implementa la función `setRawData`, que recibe los mismos parámetros, pero en caso de no ser estructurados, no falla y devuelve el valor `false`.

IrregularInterpolation

Esta clase abstracta derivada de `InterpolationTable` y representa los métodos de interpolación de tipo no estructurado comentado en la Sección 2.3.1. Además define un método estático para calcular la distancia entre puntos N-d. De esta clase derivan los método no estructurados:*closest neightbout*(Sección 2.3.1) y *inverse distance weight*(Sección 2.3.1).

4.2.5. propellerModel

La clase abstracta `propellerModel` define la interfaz y datos básicos que contiene el modelo de rotor utilizado para el cálculo de los términos fuente. Los elementos indispensables que contiene son:

- `const rotorFvMeshSel*`: puntero constante a la selección de celdas que forman parte del rotor, las susceptibles a ser añadidas términos fuente.
- `autoPtr<rotorGrid>`: una `rotorGrid` que establezca la discretización y la conexión con volúmenes finitos.
- `nBlades`: número de palas utilizadas. Si no contiene palas, por defecto, 1. Esto es debido a que las mallas promediadas en el tiempo realizan un escalado de los valores calculados en función del número de palas que se le indique.

Además, define una serie de métodos virtuales puros que hay que implementar en las clases derivadas:

- `void build(const rotorGeometry& rotorGeometry)`: define el proceso para la construcción del `rotorGrid`.
- `bool nextTimeStep(scalar dt)`: define como se realizan los pasos temporales. Devuelve `true` si se ha modificado el `rotorGrid`.
- `propellerResult calculate(const volVectorField& U, const scalarField *rho, volVectorField& force)`: calcula y añade los términos fuente correspondientes al `volVectorField` force. Genera el output de los campos requeridos. Devuelve los resultados integrados del cálculo.
- `propellerResult calculate(const volVectorField& U, const scalarField* rho)`: calcula los resultados integrales del cálculo pero no añade los términos fuente ni genera *output*. Sirve para el cálculo del control.

Estos métodos virtuales son llamados desde el *framework* implementado en la clase `propellerSource` comentado anteriormente y tan solo hay que implementarlos.

Además de la implementación de estos métodos virtuales, es necesario definir una serie de elementos para el uso de `rotorControl`:

- `enum class controlVar`: se define esta `enum class` con el nombre `controlVar`, incluyendo los nombres de las variables de control que requiere el modelo implementado. Los valores de los campos han de ir secuencialmente desde 0.
- `enum class outputVar`: se define esta `enum class` con el nombre `outputVar`, incluyendo los nombres de las variables de salida que calcula el modelo. Los valores de los campos han de ir secuencialmente desde 0.
- `static const label N_control`: se define con el nombre `N_control`, la variable que indica el número de variables de control disponibles, es decir, el número de elementos en la `enum class controlVar`.
- `label N_output`: se define con el nombre `N_output`, la variable que indica el número de variables de salida disponibles, es decir, el número de elementos en la `enum class outputVar`.
- `static Enum<controlVar> controlVarNames_`: esta variable no es estrictamente necesaria pero sí recomendada. Se asigna una palabra de texto a cada `enum class` de `controlVar`.

- `static Enum<outputVar> outputVarNames_`: de la misma forma, esta variable no es estrictamente necesaria pero sí recomendada. Se asigna una palabra de texto a cada `enum class` de `outputVar`.

propellerResult

La forma de representar los resultados del cálculo del rotor es mediante la clase `propellerResult`. Además, contiene las definiciones de los coeficientes adimensionales, lo que estos son calculados automáticamente al indicar el diámetro del rotor, densidad, velocidad del flujo, velocidad angular, empuje y par. Esta clase contiene los resultados siguientes, mostrando entre paréntesis el nombre de las variables que los contienen:

- Fuerza (`force`).
- Momento (`torque`).
- Potencia (`power`).
- Paso de la hélice (`J`).
- Eficiencia propulsiva η (`eta`).
- Coeficiente de empuje C_T (`CT`).
- Coeficiente de par C_Q (`CQ`).
- Coeficiente de potencia C_P(`CP`).

Los coeficientes se definen según las definiciones de la UIUC (University of Illinois Urbana-Champaign):

$$J = \frac{V}{n\ D} \tag{4.1}$$

$$C_T = \frac{T}{\rho\ n^2\ D^4} \tag{4.2}$$

$$C_P = \frac{Q}{\rho\ n^2\ D^5} \tag{4.3}$$

$$C_Q = \frac{P}{\rho\ n^3\ D^5} \tag{4.4}$$

$$\eta = \frac{T\ V}{P} \tag{4.5}$$

Donde V es la velocidad de referencia del flujo, T es el empuje del rotor,rho es la densidad del fluido, n es la velocidad angular en revoluciones por segundo, D es el diámetro del rotor, Q es el par del rotor y P es la potencia propulsiva.

Además, se crea una clase derivada de `regIOobject` y `propellerResult` llamada `regIOpropellerResult`. Esta clase tiene la función de realizar el autoguardado de los datos del rotor según se configure. Al implementar el autoguardado, se pueden obtener los valores de forma automática e incluso los que se obtienen tras la convergencia. A continuación, se muestra el formato en el que se guardan los datos de `propellerResult`, como es habitual, se guardan en un archivo de diccionario:

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD|
|  \\    /   O peration     | Version:  2206                                  |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    arch        "LSB;label=32;scalar=64";
    class       regIOobject;
    location    "0.001";
    object      propeller1:result;

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

force = (-0.027063244 0.044782108 4.3658565);
thrust = 4.3658565;
torque = (0.0066243808 -0.01163311 0.3691316);
power = 154.62148;
j = 0.6;
eta = 0.56471538;
ct = 0.012830272;
cq = 0.002169591;
cp = 0.013631935;


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

bladeElementModel

La clase `bladeElementModel` implementa el método de la teoría de elemento de pala. Esta clase contiene diferentes elementos que conjuntamente permiten implementar el método de una forma modular:

- Una instancia de `bladeModel`, que contiene la definición de la pala utilizada, su geometría y los perfiles.
- Una lista de tensores que definen el tensor local de cada elemento de pala del `rotorGrid`.
- Un `tip factor` que indica el radio adimensional a partir del cual la sustentación es nula.
- Las variables de referencia: densidad, velocidad, viscosidad y velocidad del sonido.
- Un ángulo de giro del rotor para el caso transitorio.

- Una instancia del `rotorControl` utilizado. Se define internamente ya que depende del tipo de `propellerModel` utilizado.
- Una instancia de la clase de efectos3D de polares.
- Una instancia de `airfoilModelList` con todos los `airfoil` disponibles.
- Una instancia de `outputFields` para campo escalar y otra para campo vectorial.

Estos elementos se utilizan para el cálculo de los términos fuente y la fuerza en cada rotor. Los pasos realizados para el cálculo son los siguientes, implementado la función `propellerResult calculate(const volVectorField& U, const scalarField *rho, volVectorField& force)`:

- Se itera sobre todas las celdas del `rotorGrid`.
- Para cada celda se calcula la fuerza resultante de la siguiente forma:
  - Se obtiene la interpolación de la pala para la posición radial de la celda.
  - Se obtiene la velocidad en coordenadas locales con el `bladeTensor` de la celda.
  - Se obtiene la velocidad de la pala a través de la clase `rotorControl`.
  - Se suman vectorialmente las velocidades para obtener la velocidad relativa del flujo.
  - Se obtiene la velocidad de referencia y el ángulo de incidencia según Ecuación 2.3 y Ecuación 2.2
  - Se obtiene el número de Reynolds (Ecuación 2.14) y Mach locales (Ecuación 2.15).
  - Se obtienen el c_L y c_D y se aplican las correcciones 3D y el `tipFactor`.
  - Se obtiene la sustentación (Ecuación 2.4) y resistencia (Ecuación 2.5) por unidad de longitud y se expresa en ejes cuerpo (Ecuación 2.8 y Ecuación 2.9).
  - Se expresa la fuerza anterior en el sistema de referencia del rotor.
  - Se almacenan los campos requeridos.
- Se integra el valor de la fuerza distribuida en la celda y se aplica el factor de escalado correspondiente.
- Se obtiene la contribución a la fuerza total y al momento de elemento y se suma al total acumulado.
- Se actualizan los valores del `propellerResult`.

La clase `effects3D::polarCorrection` implementa las ecuaciones de corrección de polares mencionadas en la Subsección 2.1.5. Estas se aplican sobre el c_L obtenido con la interpolación en el cálculo de cada elemento.

forceModel

La clase `forceModel` implementa el método *body force*. Este método utiliza unas tablas de coeficiente de empuje C_T y coeficiente de par C_Q en función del paso J para obtener las fuerzas del rotor. La ecuaciones se definen en la Subsección 2.1.6. Esta clase se deriva de la clase abstracta `propellerModel` al igual que `bladeElementModel`, aunque su implementación es mucho más sencilla. La clase contiene los siguientes elementos:

1. Una lista de tensores que definen el tensor local de cada elemento de pala del `rotorGrid`.
2. La variable de referencia: densidad.
3. Un ángulo de giro del rotor para el caso transitorio.
4. Una instancia del `rotorControl` utilizado. Se define internamente ya que depende del tipo de `propellerModel` utilizado.
5. Una tabla de interpolación para el C_T y otra para el C_Q.
6. Una instancia de `outputFields` para campo escalar y otra para campo vectorial.

Para el cálculo de los términos fuente, el procedimiento es idéntico que para el caso del `propellerModel`, con la diferencia de que la fuerza se obtiene mediante las tablas de empuje y par y con la distribución de Goldstein. Las ecuaciones y el método se describe en detalle en la Subsección 2.1.6.

4.2.6. rotorControl

Esta clase abstracta encapsula la lógica de control del rotor. Es decir, es la encargada de proveer todas las variables de control que necesita el rotor para el cálculo de las fuerzas. La clase tiene un método virtual puro:

```
1 virtual void correctControl
2 (
3     const vectorField& U,
4     const scalarField* rhoField
5 ) = 0;
```

Que ha de ser implementado por las clases derivadas, y es el método encargado de corregir los valores de las variables de control según sea necesario. Este método es llamado por la clase `propellerSource` en cada iteración de FV. Además contiene un método estática como *helper* para leer la velocidad angular en varios formatos a partir de un diccionario.

Derivando esta clase, se tiene la clase con `template rotorControlTemplate`. Presenta un argumento de plantilla del tipo de `propellerModel` que va a controlar. Mediante el tipo de la plantilla, se obtienen las `enum` de las variables de control y de salida definidas en la implementación de `propellerModel` utilizada. Además define el tipo de contenedor utilizado para cada tipo de variable, haciendo uso de la clase EnumField, comentada en la Sección 4.2.12, de la siguiente forma:

```
1 typedef typename model::outputVar outputVar;
2 typedef typename model::controlVar controlVar;
3
4 typedef util::EnumField<outputVar,model::N_output>
5     outputVarType;
6 typedef util::EnumField<controlVar,model::N_control>
7     controlVarType;
```

De esta forma se proporciona el acceso a la información sobre los tipos de variables de control y de salida. La clase `rotorControlTemplate` es la que ha de ser utilizada como clase base para derivar de ella los tipos de control para cada modelo de *propeller*.

Según la forma en la que se obtienen el valor de las variables de control, se distinguen dos tipos de control: `fixedControl` y `targetValue`.

4.2.7. fixedControl

En este tipo de control, las variables de control son fijadas por el usuario a través del diccionario `rotorControl`. Se aplica directamente los valores indicados, y la función `correctControl` no realiza ninguna instrucción en su implementación.

4.2.8. targetValue

Este tipo de control permite fijar el valor de las variables de salida del modelo del rotor, ajustando el valor de las variables de control. La implementación permite elegir los valores de las variables de salida que se desee modificando los valores de las variables de control indicadas. Es decir, provee de la flexibilidad de elegir cualquier combinación, siempre que sea el mismo número de variables de control como de salida. Si las variables de control escogidas no tienen capacidad de actuar sobre las salidas, como es de esperar, no será posible de realizar. El uso de la clase `EnumField` facilita en gran medida la implementación de este método de control genérico. El método para obtener las variables de control está basado en Newton-Raphson multidimensional, explicado en detalle en laEcuación 2.89.

4.2.9. rotorGrid

La clase abstracta `rotorGrid` representa la discretización de los elementos a los que se aplica el `propellerModel` para el cálculo de las fuerzas del rotor y define el enlace con volúmenes finitos para la adición de los términos fuente. Son las implementaciones de esta clase las que utilizan las distintas técnicas de mallado mencionadas en Sección 2.5. Es una clase abstracta que define una serie de atributos comunes a todas las implementaciones:

- `const rotorGeometry&`: una referencia a la geometría del rotor. Contiene los sistemas de referencia.
- `const rotorFvMeshSel&`: una referencia a la selección de celdas del rotor.
- `label nBlades`: número de palas para el cálculo de los factore de promediado.
- `sampleMode`: indica el tipo de muestreo utilizado: `center`, `closestCell` o `cellMean`.

- `word exportName_`: define el nombre asignado para exportar la malla. Si no se le proporciona ningún nombre, no es exportada.
- `PtrList<gridCell>`: una lista de celdas del rotor, que definen la geometría.

Los tres primeros son obtenidos a partir del constructor, el modo de muestreo y el `exportName` se leen a partir del diccionario. Además, se definen una serie de funciones útiles para: actualizar los centros de las celdas, obtener los centros a partir de las celdas FV más próximas, guardar los datos de las áreas, obtener una referencia a las celdas, etc. No se define ninguna función virtual pura, ya que la creación de la malla debe ser realizada por el constructor.

La clase que define las celdas del `rotorMesh` es llamada `gridCell`. Esta clase debe ser extendida para implementar los distintos tipos de celda para cada tipo de malla. No obstante, define algunos atributos comunes a todas las celdas:

- `vector center_`: define en coordenadas polares locales (r-θ-0) las coordenadas del centro de la celda, donde van a ser evaluadas las fuerzas.
- `scalar factor_`: define un factor de escalado aplicado a la fuerza obtenida. Es utilizado para el promediado temporal y para incluir el efecto del número de palas.
- `List<label> cellis_` y `List<scalar> weights`: estas dos listas definen las celdas de volúmenes finitos asignadas a esta celda del rotor y el peso asignada a cada una. El peso indica como se distribuye el término fuente aplicado a cada celda a partir de la fuerza calculada para esa celda.
- `scalar area_`: el área de la celda.
- `interpolatingCell`: el índice de la celda principal, si existe.

Cualquier clase derivada de `gridCell` será valida siempre que configure todos estos atributos correctamente. La clase base contiene una serie de métodos de utilidad para añadir las celdas y los pesos, y calcular el ponderado de forma automática, comprobar que todas las celdas son válidas y contienen al menos una celda de volúmenes finitos (no trivial en paralelo). Además, se definen una serie de métodos para la adición de los términos fuente o campos escalares arbitrarios. Para la aplicación de los **términos fuente**, hay que tener en cuenta que la **fuerza aplicada** se tiene que dividir entre el **volumen de la celda**, ya que en la resolución de las ecuaciones se realiza una **integral de volumen** de esta.

4.2.10. rotorFvMeshSel

Esta clase concreta contiene las celdas seleccionadas de volúmenes finitos que forman parte del rotor. La selección se puede hacer de dos formas:

- Indicando la geometría y seleccionando las celdas a partir de esta.
- Indicando las celdas y infiriendo la geometría a partir de ella.

Selección de celdas

Las celdas se seleccionan utilizando una herramienta de OpenFOAM llamada `cuttingPlane`, que obtiene las celdas intersectadas por un plano. Después, se seleccionan las que están situadas entre el radio interno R_i y el externo R. Para inferir la geometría, es necesario obtener el centro, el radio y el vector normal. El centro se obtiene como la media de los centros de las celdas. El radio se obtiene como el valor del centroide más alejado al centro definido anteriormente. Para obtener la normal, es necesario indicar la dirección de esta, aunque sea de forma aproximada, ya que hay ambigüedad. Para el cálculo de la normal se obtienen los vectores entre todos los pares de celdas y se calcula el producto vectorial entre ellas y se suman los resultados, cambiando el signo si no apunta en la misma dirección que la dirección aproximada que se indica. La normal se obtiene tras normalizar este vector.

4.2.11. diskSampler

La clase base `diskSampler` se encarga de interpolar a la malla del rotor los datos del campo fluido de la malla de volúmenes finitos. Para ello se debe implementar la función virtual siguiente:

```
1 virtual const Field<fType>& sampleField
2 (
3     const GeometricField<fType, fvPatchField, volMesh>& U
4 ) = 0;
```

Que recibe como parámetro el campo fluido como una referencia constante y devuelve un campo con la solución interpolada. Los índices en el campo devuelto corresponden a la posición de las celdas del rotor en la lista de la clase `rotorGrid`. A continuación, se exponen las implementaciones realizadas.

fixedValue

Esta implementación es la más básica de todas, y devuelve un valor constante indicado en el diccionario.

domainSampler

Esta implementación obtiene los valores del dominio fluido. Para ello tiene en cuenta el tipo de muestreo del `rotorMesh`: `center`, `closestCell` y `cellMean`. Además permite definir un `offset` y un factor de escala de los puntos de muestreo. Si no presenta offset y el método de muestreo es `closestCell`, la interpolación es directa, y se toma el valor de la celda más próxima de volúmenes finitos como el valor de la celda del rotor. En el resto de casos, se utiliza una utilidad de OpenFOAM llamada `cellInterpolatingWeights` que permite obtener la interpolación en puntos discretos del dominio.

azimuthalMean

Esta clase deriva de `domainSampler`, y tiene la misma funcionalidad excepto por el hecho de que realiza un promediado azimutal de los valores obtenidos. Por este motivo, esta implementación solo se puede utilizar junto a `polarGrid`.

fileSampler

Esta clase deriva de `domainSampler`, y tiene la misma funcionalidad, excepto de que los datos son muestreados de un fichero de datos. El fichero puede ser un diccionario de OpenFOAM para el dominio de estudio, o un fichero tipo `csv` donde se indican valores discretos de los campos y la posición de estos valores. Cuando se introducen mediante un `csv`, se crea una interpolación y se guardan los datos como diccionario, ya que es un formato mucho más rápido de interpretar por parte de openFOAM. En el `csv` los campos vectoriales se indican en las columnas con títulos: `VelocityX`, `VelocityY` y `VelocityZ`. La posición se indica mediante: `X`, `Y` y `Z`. Si el campo leído es escalar se indica con el título: `scalar`.

4.2.12. util

En esta sección se comentarán las diferentes utilidades desarrolladas con el fin de reutilizar la mayor cantidad de código posible. Se han agrupado en ficheros según el tipo de funciones o en su propio fichero si trata de una clase. El tipo de utilidades engloba la lectura/escritura de archivos `csv`, algoritmos geométricos, automatización de *inputs*, utilidades de paralelizado y *solver* de funciones.

csvTable

La clase `csvTable` se ha desarrollado para dar versatilidad a la lectura y escritura de archivos `csv`. Se ha desarrollado como plantilla, indicando el tipo de datos que forman las columnas y el tipo de datos que forman las cabeceras. En la mayoría de casos tratados los tipos son: `scalar` y `word` respectivamente. La clase permite configurar si el archivo tiene o no cabecera o si tiene que ignorar un número de lineas al inicio del documento. Tras realizar la lectura del fichero indicado mediante su `fileName`, se puede acceder al documento a través de filas o columnas, ya sea mediante el índice que ocupa la fila/columna o mediante el título de cabecera en el caso de las columnas. Para el caso de la escritura, los datos pueden ser añadidos por filas o columnas. Dado que la clase está pensada como interfaz de lectura/escritura no está enfocada en ser modificada y los datos añadidos no pueden ser eliminados o modificados.

geometry

En el fichero geometry.H se definen las distintas funciones (como miembros estáticos de la clase `geometry`, ya que define algunos `typedef`) y se implementan todos los algoritmos geométricos que se encuentra detallados en la Sección 2.4.

property

Esta clase añade funcionalidad a la lectura de parámetros de configuración desde un archivo. Permite comprobar si una variable ha sido o no leída, para evitar utilizar un valor por convenio para indicar que no ha sido aún definida. También para comprobar que el valor recibido es válido. Para ello se define la clase como `template` y se crea una variable interna que guarda el valor, además de una variable de tipo `bool` que indica si la variable ha sido o no inicializada. La forma de comprobar que el valor es válido se realiza incluyendo una lista de `std::functions` que se prueban sobre el valor almacenado y devuelven `true` si cumplen la condición.

Existen tres métodos para realizar estas comprobaciones:

- `bool isValid()`: devuelve `true` si la variable cumple todas las condiciones.
- `bool isSet()`: devuelve `true` si la variable ha sido inicializada.
- `bool isRead()`: devuelve `true` si la variable cumple todas las condiciones y ha sido inicializada. Es decir, que está lista para utilizarse.

Además, la clase incluye una sobrecarga del operador `static_cast` para que las instancias de esta clase se comporten como instancias del tipo indicado en la `template`.

EnumFields

La clase `EnumFields` almacena, da acceso y manipula una lista de valores escalares asociados a una `enum`. Cuenta con dos argumentos de `template`: el tipo de `enum` y el número de elementos. Permite leer de un diccionario los elementos del `enum` presentes, indicando los nombres mediante la clase `Enum` de OpenFOAM. También permite modificar varios valores a la vez mediante la función `set` utilizando una lista de valores y una lista de valores de la `enum` que indican que elementos se quiere modificar. También existe la función que realiza el proceso inverso, `get`, que obtiene en una lista los elementos indicados. También se encuentra sobrecargado el operador [] para admitir valores del `enum` como parámetros, realizando internamente un `cast` a `label` mediante `static_cast`. Esta clase se utiliza internamente en `rotorControl` para la automatización de selección y uso de las variables de control y las de salida.

GatherList

En este archivo se definen dos funciones: `GatherList` y `GatherListList`. Estas están orientadas a la paralelización y proporcionan la utilidad de unir en una sola lista, listas de elementos de cada proceso, indicando al proceso que pertenece cada elemento mediante un *array* que indica el índice desde el que empiezan los datos de cada núcleo. Si tenemos en el núcleo 0 la lista $\{1, 2, 3\}$ y en el núcleo 1 la lista $\{4, 5, 6, 7\}$ al utilizar la función `GatherList` obtendremos una sola lista con $\{1, 2, 3, 4, 5, 6, 7\}$ y una lista de índices $\{0, 3\}$ donde el 0 indica que los datos del núcleo 0 empiezan en la posición 0 y el 3 indica que los datos del núcleo 1 inician en la posición 3. Este método hace uso de la función `reduce` comentada en la Sección 3.3.9.

La función `GatherListList`, realiza la misma funcionalidad que `GatherList` pero para una lista de listas. Es decir, cada elemento de la lista, es en sí, una lista. Ambas funciones se realizan con `template` indicando el tipo de variable de la lista, o el tipo de variable de la lista de listas.

readHelper

En este archivo se han definido dos funciones para la lectura y instanciación de tablas de interpolación a partir de diccionarios. La primera función es definida como:

```
autoPtr<RegularInterpolation<scalar,scalar,1>>
NewInterpolationFromDict
(
    const dictionary& dict,
    word x_name,
    word y_name,
    bool convertToRad = false,
    bool enableCSV=false,
    const csvTable<scalar,word>* csv = nullptr
);
```

Esta función está diseñada para introducir datos 1D mediante diccionarios, pudiendo escoger entre valor constante, mediante una lista de valores o como campos desde un archivo `csv`. Se recibe un puntero a una tabla `csv` por si no requiere importarse una tabla nueva. Si se requiere, se hace uso de la ruta indicada mediante el atributo `csv` en el diccionario que recibe la función. La otra función implementada es la siguiente:

```
template<class typeIn, class typeOu, label dim>
autoPtr<InterpolationTable<typeIn,typeOu,dim>>
NewInterpolationFromRaw
(
    const dictionary& dict,
    const List<List<typeIn>>& xList,
    const List<typeOu>& yList
);
```

Esta función instancia una nueva tabla de interpolación genérica, por eso se define como `template`. Recibe un diccionario y los datos para construir la tabla. En el diccionario se indica el tipo de interpolación que se quiere utilizar: linear, *inverse distance weighting* o *closest neighbour*. Si el método escogido no puede utilizarse por el tipo de datos proporcionados, se considera un `FatalError` y se cierra el programa. Si no se escoge ningún método, este se hace automáticamente, siendo el orden de prueba el indicado en la lista anterior.

functionSolver

En esta clase se implementa el método de Newton-Raphson multidimensional, explicado en detalle en la Subsección 2.3.2. La función desarrollada permite obtener la solución de un sistema de N ecuaciones con N incógnitas. Los parámetros que admite son los siguientes:

- Número de dimensiones.
- Función que resolver, que tiene que recibir un `scalarField` como parámetro y devolver un `scalarField` con el valor de $\boldsymbol{f}(\boldsymbol{x})$.
- Un valor inicial $\boldsymbol{x_0}$.
- Un valor para d$\boldsymbol{x}$ para el cálculo de la derivada mediante diferencias finitas.
- Un valor para el factor de relajación α.
- El número máximo de iteraciones.
- La tolerancia admisible del error.
- Y si se requiere mostrar por consola información del resultado de la función.

Se utiliza la clase `scalarField` para la implementación de Newton-Raphson ya que tiene sobrecargados los operadores matemáticos y permite una implementación sencilla junto la clase `SquareMatrix`.

4.3. Paralelizado

En OpenFOAM el paralelizado se realiza ejecutando el mismo *solver* en N núcleos o procesos independientes. Los procesos se comunican y sincronizan entre si mediante OpenMPI, una librería que implementa el sistema MPI (Message Passing Interface). Los programas se diseñan de forma que al ejecutarse en paralelo, se minimicen las comunicaciones entre los diferentes procesos, ya que son muy costosas.

Para compatibilizar el módulo desarrollado con la ejecución en paralelo hay que tener en cuenta una serie de ideas:

- Cada proceso es independiente al resto y solo tiene acceso directo a la información propia.
- La malla del problema es dividida en regiones, y cada región es independiente del resto, es decir no se tiene acceso directamente desde otro núcleo a los datos de la malla global.
- La solución entre mallas se comunica a través de unas condiciones de contorno especiales, que envían y reciben los datos necesarios de las celdas vecinas.
- Las operaciones de comunicación han de mantenerse al mínimo.
- Los núcleos deben de estar balanceados, es decir, que tarden el mismo tiempo en realizar una iteración para que el paralelizado sea lo más óptimo posible.

4.3.1. Planteamiento

Teniendo en cuenta las ideas generales del paralelizado, se plantea un método para habilitar el uso de *propellerSource* en cálculos en paralelo. Los tres aspectos a tener en cuenta en el paralelizado son los puntos de comunicación entre volúmenes finitos y *propellerSource*, es decir las clases `fvMeshSelection`, `rotorGrid` y `diskSampler`. El planteamiento realizado, propone no paralelizar el cálculo del rotor, es decir, que todos los núcleos tengan acceso a los datos para realizar el cálculo. Este método no es el más eficiente, dado que podrían reducirse el número de comunicaciones realizadas. No obstante, en este método solo es necesario comunicar el campo de velocidades y o densidades del rotor a todos los núcleos, es decir solo un mensaje. Además para dominios con una gran número de elementos, los costes del cálculo del rotor son órdenes de magnitud inferiores que las iteraciones CFD, por lo que este método sería válido. Sería posible una implementación más optimizada con un esfuerzo de desarrollo mucho más elevado que el realizado para este caso.

fvMeshSelection

Con la estrategia utilizada para adaptar al cálculo paralelizado, esta clase tiene que comunicarse con la de los otros núcleos para obtener información del resto de celdas, se comunican los datos de los índices de las celdas. De esta forma puede saberse si todas las celdas del rotor pertenecen al mismo núcleo, y no es necesario comunicar datos adicionales.

rotorGrid

Para habilitar el uso de esta clase en paralelo hay que mirar en detalle como se implementa los distintos tipos de mallas. Para las mallas de rotor estáticas, los costes de comunicación solo se realizan una vez, por lo que la penalización no es relevante. Para las mallas de rotor transitorias, como es el caso de `bladeGrid`, se recomienda que todas las celdas que pertenezcan al rotor, permanezcan en el mismo núcleo, ya que esta optimizado para no realizar comunicaciones adicionales cuando se da el caso. Para el resto de situaciones no es muy relevante.

En general la estrategia seguida para adaptar la malla al cálculo en paralelo es la siguiente:

- Se comunican los datos de la malla necesarios entre procesos mediante operaciones de reduce, utilizando los métodos `GatherList` y `GatherListList` desarrollados.
- Se genera la malla completa en todos los núcleos.
- Se seleccionan las celdas FV asignadas a cada `rotorCell`, de forma que solo se asignan las correspondientes al propio núcleo. Es decir, la clase `rotorCell` del núcleo 0 no tiene asignada ninguna celda FV de un núcleo distinto al 0.
- La celda FV de referencia de cada `rotorCell` es -1 si esta no corresponde al núcleo.
- Los pesos utilizados para la adición de los términos fuente se calculan y se sincronizan entre núcleos. Por ejemplo si una `rotorCell` tiene una celda del núcleo 1 y otra del

núcleo 2, ambas definidas con un peso de 10. Tras la sincronización cada una tendrá asignada un pero de 0,5, ya que la suma de los pesos tiene que ser la unidad.

diskSampler

El método utilizado para el `diskSampler` es similar al resto. Se parte de una lista de valores inicializada a 0. Cada núcleo asigna los valores que puede muestrear a partir del trozo de malla de la que dispone. Mediante una operación de reduce, se suman las listas de todos los núcleos, resultando en una lista que contiene los valores muestreados de todo el disco. Esto es posible ya que para una posición de muestreo dada, esta solo existirá en uno de los núcleos, por lo que en el resto de núcleos tomará el valor 0.

Capítulo 5

Guía de uso

En el presente capítulo se comentará como descargar y compilar el código fuente desarrollado para su uso junto a los *solver* de OpenFOAM. El nombre del módulo desarrollado es *PropellerSource* y se referirá a él de esta manera en el capítulo actual. También se explicarán todos los parámetros de configuración y uso mediante los diccionarios. El código completo y su documentación, puede encontrarse como documento adjunto al presente trabajo, con el título: Acoplamiento del método de teoría de elemento de pala con volúmenes finitos en OpenFOAM - Documentación y Código.

5.1. Instalación

En el presente capítulo se indican los pasos para la compilación a partir del código fuente desarrollado. Los únicos prerrequisitos de instalación son tener compilado al menos la versión v2212 del código de OpenFOAM y el uso de un sistema operativo Linux. El código se encuentra disponible de forma abierta bajo la licencia **GNU GENERAL PUBLIC LICENSE**. Los pasos a seguir son los siguientes:

1. Dirigirse a la web `https://github.com/FredyTP/PropellerSourceOpenFOAM` y descargar el código fuente de la rama *main*, haciendo *click* en el botón *Code* y después en *Download Zip*, tal como se muestra en la Figura 5.1.

2. Descomprimir el fichero *Zip* donde se desee y abrir la carpeta con el terminal.

3. Ejecutar el comando `wmake -j` para compilar el código según se muestra en Figura 5.2. El binario del módulo es generado en la ruta definida por la variable de entorno de OpenFOAM: `FOAM_USER_LIBBIN`.

Siguiendo estos pasos ya está lista la libreria de *PropellerSource* para su uso junto los *solvers* de OpenFOAM para los cuales están disponible `fv::options`.

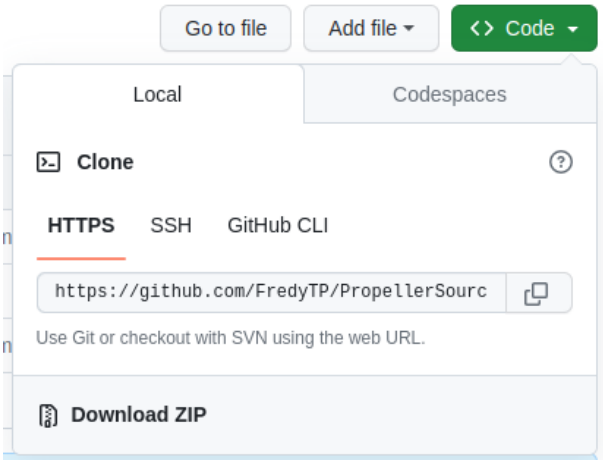

Figura 5.1: Método de descarga del código fuente.

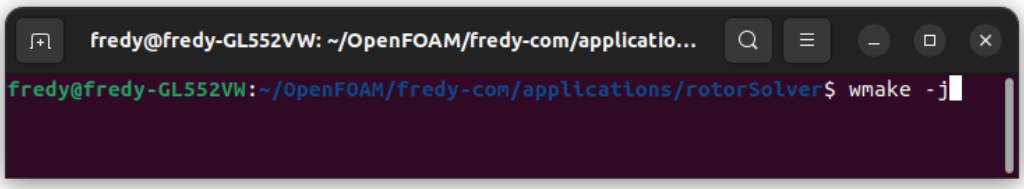

Figura 5.2: Comando para la compilación del módulo usando todos los núcleos disponibles.

5.2. Configuración del caso

En la siguiente sección se comentará como configurar un caso para hacer uso de *PropellerSource* y de todos los parámetros de configuración de los que dispone.

5.2.1. Link to solver

El primer paso antes de configurar el diccionario del *PropellerSource*, es indicar al *solver* que se va a utilizar el módulo *PropellerSource*. Para ello, es necesario tener el código compilado siguiendo los pasos de la Sección 5.1. Para vincularlo con el *solver* se añade la siguiente línea al diccionario *controlDict*:

```
libs (propellerSource.so);
```

En el caso de que `libs` ya estuviera definida, añadirlo de la siguiente forma:

```
libs (otraLib.so propellerSource.so);
```

Es necesario tener en cuenta que *PropellerSource* esta desarrollado sobre `fv::options` por lo que solo los *solvers* que utilicen esta funcionalidad, serán compatibles con el módulo. En la Tabla 5.1, se muestran todos los *solvers* compatibles con `fv::options`.

Solver	transient	compressible	turbulence	heat-transfer	buoyancy	combustion	multiphase	particles	dynamic mesh	multi-region	fvOptions
boundaryFoam											
buoyantPimpleFoam	✔	✔	✔	✔	✔						✔
buoyantSimpleFoam		✔	✔	✔	✔						✔
chemFoam	✔			✔		✔					
chtMultiRegionFoam	✔	✔	✔	✔	✔					✔	✔
coldEngineFoam	✔	✔	✔	✔		✔			✔		✔
engineFoam	✔	✔	✔	✔		✔			✔		✔
fireFoam	✔	✔	✔	✔	✔	✔				✔	✔
icoFoam	✔										
interFoam	✔		✔				✔		✔		✔
laplacianFoam	✔										✔
pimpleFoam	✔		✔						✔		✔
pisoFoam	✔		✔								✔
potentialFoam											
reactingFoam	✔	✔	✔	✔		✔					✔
reactingParcelFoam	✔	✔	✔	✔	✔	✔		✔			✔
rhoCentralFoam	✔	✔	✔	✔							
rhoPimpleFoam	✔	✔	✔	✔					✔		✔
rhoSimpleFoam		✔	✔	✔							✔
scalarTransportFoam	✔										
simpleFoam			✔								✔
sprayFoam	✔	✔	✔	✔	✔	✔		✔			✔
XiFoam	✔	✔	✔	✔		✔					✔

Tabla 5.1: Solvers compatibles con `fv::options` [15].

5.2.2. Ideas generales

Toda la configuración de `PropellerSource` es realizadas en el diccionario `fvOptions`, en este se detallan todas las instancias de `fv::option` utilizadas. Esto incluye términos fuente, limitadores y correctores. La cabecera del archivo es la siguiente:

```
/*---------------------------------*- C++ -*----------------------*\
| =========                       |                                |
| \\      /  F ield               | OpenFOAM: The Open Source CFx  |
|  \\    /   O peration           | Version:  v2206                |
|   \\  /    A nd                 | Website:  www.openfoam.com     |
|    \\/     M anipulation        |                                |
\*---------------------------------------------------------------*/
FoamFil

    version     2.0
    format      ascii
    class       dictionary
    object      controlDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *//
```

Cada instancia de *fv::option* utilizada se define mediante un diccionario en el archivo *fvOptions* de la siguiente forma:

```
nombreAsignado
{
    type            propeller;
    fields          (U);
    active          true;
}
```

Donde `type` indica el nombre de la clase utilizada, que para el caso de *PropellerSource* es `propeller`. El atributo `fields` indica los campos fluidos a los que se le aplica el término fuente, para el caso de `propeller` es únicamente `U`, es decir el campo de velocidaded. Finalemente, `active` indica si el *fvOption* esta activo o no, en caso de no estarlo no son añadidos los términos fuente, pero si que es creada la instancia. Los diccionarios requeridos por el modelo son los siguientes:

- `geometry`: define la geometría del rotor.
- `rotorMesh`: indica la forma de obtener las celdas que forman parte del motor.
- `velocitySampler`: indica la forma en la que se obtiene la velocidad del flujo en elrotor.
- `densitySampler`: indica la forma en la que se obtiene la densidad del flujo en el rotor. **Requerido únicamente en flujo compresible**.
- *propellerModel*: configura el modelo de rotor utilizado. Se detallará en su sección los requerimientos y configuración en función del modelo utilizado.

A continuación, se muestra la estructura básica del término fuente `propeller`, con los atributos y diccionarios base.

```
propeller1
{
    type            propeller;
    fields          (U);
    active          false;

    geometry
    {
    }
    rotorMesh
    {
    }
    velocitySampler
    {
    }
    densitySampler
    {
    }
    propellerModel
    {
    }
}
```

En las siguientes secciones se detallará el contenido de cada uno de estos diccionarios *base*. Para los atributos de configuración opcionales se indicará entre paréntesis el valor por defecto como un comentario: `(default 0.5)`. Cuando un atributo solo sea necesario en función del valor de otro atributo, se indicará entre paréntesis la condición mediante el comentario: `when (atribute = value)`. Para el caso de que los atributos solo puedan tomar ciertos valores concretos, se indicarán mediante separadores verticales |.

5.2.3. geometry y rotorMesh

Estos dos diccionarios se presentan de forma conjunta, ya que la definición de ambos está íntimamente relacionada, pudiéndose omitir algunos valores en la geometría en función de como se defina `rotorMesh`. El diccionario `geometry` define la geometría del rotor y los atributos que contiene son los siguiente:

```
geometry
{
    innerRadius     0.025;          //[m]
    radius          0.25;           //[m]
    center          (0.0 0.0 0.0);  //[m]
    direction       (0 0 1);
    psiRef          (1 0 0);
}
```

Siendo `innerRadius` el radio interno del rotor en metros, `radius` el radio externo en metros, `center` el centro del rotor en coordenadas globales, es decir, la posición que ocupa dentro del dominio fluido. La dirección del rotor se define mediante el atributo `direction` indicando el vector normal al plano del rotor. Para completar el sistema de referencia, hace falta un segundo eje, siendo este el eje de referencia para los ángulos en el plano del rotor, indicado por `psiRef`.

El diccionario `rotorMesh` define como se seleccionan las celdas que formarán parte de la geometría del rotor:

```
1  rotorMesh
2  {
3      selectionMode geometry | cellZone | cellSet;
4
5      //when (selectionMode = cellZone):
6      cellZone          cellZoneName;
7
8      //when (selectionMode = cellSet):
9      cellSet           cellSetName;
10
11     //when (selectionMode = geometry):
12     closestCenter     true;
13
14     correctGeometry   false;
15 }
```

La función de cada uno de los atributos es la siguiente:

- `selectionMode`: indica la forma en la que se seleccionan las celdas. El significado de cada valor es el siguiente:
  - `geometry`: la selección de celdas se obtiene a partir de la geometría, es necesario indicar todos los valores.
  - `cellZone` y `cellSet`: la selección de celdas se indica y es la geometría la que se infiere a partir de estas. Se pueden definir los valores de la geometría, utilizándose las celdas indicadas en `cellZone` y `cellSet` para obtener las celdas del rotor. La `direction`, `psiRef` y `innerRadius` siempre son requeridos.
- `cellZone`: indica el nombre de la *cellZone* utilizada.
- `cellSet`: indica el nombre del *cellSet* utilizado.
- `closestCenter`: cuando es `true` habilita (para selectionMode = cellZone) que se mueva el centro indicado por la geometría en la dirección del rotor, para que el plano del rotor corte el centro de una celda. Suele mejorar la calidad de la selección obtenida.
- *correctGeometry*: cuando el valor es `true`, modifica los valores de la geometría según los obtenidos a partir de la selección.

5.2.4. velocitySampler y densitySampler

Los dicionarios `velocitySampler` y `densitySampler` son instancias de la clase base `diskSampler`. Existen 4 tipos diferentes de `diskSampler`: `fixedValue`, `domainSampler`, `fileSampler` y `azimutalMean`. Se muestran a continuación como se configura cada uno de ellos.

El tipo `fixedValue` solo presenta dos atributos, si el valor indicado es normal a la superficie, por lo que si se trata de un vector, solo hay que indicar su módulo.

```
velocitySampler
{
    type            fixedValue;
    normal          true | false; // (default = false)
    value           20.0 | (0.0 20.0 0.0);
}
```

La clase `domainSampler` permite indicar si se requiere de un *offset* respecto al plano del disco mediante el atributo `offset` y si se quiere aumental el área de la región de muestreo mediante `scale`.

```
velocitySampler
{
    type            domainSampler;
    offset          0.1; // (default = 0.0)
    scale           1.1; // (default = 1.0)
}
```

Los parámetros `fileSampler` son los mismos que el anterior, salvo que se puede indicar el archivo desde el cual se leen los datos. Si los datos son de un csv, se indica el nombre mediante `csv`. Si provienen de un archivo de datos de OpenFOAM se utiliza `file`. Cuando los datos son introducidos a través de un csv, se guardan automáticamente en formato OpenFOAM en el directorio `0`.

```
velocitySampler
{
    type            fileSampler;
    file            foamFile;
    csv             csvFile.csv;
    offset          0.0;
    scale           1.0;
}
```

Para `azimutalMean` se utiliza los mismos parámetros que para `domainSampler`. La diferencia es que este tipo solo se puede utilizar para la malla `polarGrid`, ya que realiza un promediado azimutal.

```
velocitySampler
{
    type            azimutalMean;
    offset          0.0;
    scale           1.0;
}
```

5.2.5. propellerModel

El diccionario de configuración de `propellerModel` se comentará en función del método utilizado: `bladeElementModel` o `forceModel`. Pero antes, se explicará la configuración de dos elementos comunes a ambos: `rotorGrid` y `control`. A continución se muestra la jerarquia que siguen estos diccionarios.

```
propellerModel
{
    rotorGrid
    {

    }
    control
    {
    }
}
```

rotorGrid

En este diccionario se configura la generación de la malla del rotor conocida como `rotorGrid`. Se comentan a continuación como se realiza la configuración en función de los tres tipos de mallado disponible: `bladeGrid`, `polarGrid` o `meshGrid`(*Voronoi*, *intersection* y *proyection*). Pero antes de ello se expondrán los atributos comunes a los tres tipos de mallado:

- `sampleMode`: forma en la que se toman los datos del dominio y en que se define el centro de la celda.
  - `center`: se toma el centro geométrico de la celda.
  - `closestCell`: se toma el centroide proyectado de la celda más cercana al centro geométrico.
  - `cellMean`: se realiza un promedidado de las celdas FV que contiene.
- `includeIfVertex`: si es `true` incluye en el rotor las celdas que estén dentro del radio al menos un vértice. Esto puede mejorar los resultados en casos con baja resolución. Si es `false` solo se incluye una celda si contiene el centro.
- `exportName`: nombre asignado al *script* de Python que permite visualizar el mallado.

Para el mallado polar, se define el diccionario de la siguiente forma:

```
rotorGrid
{
    type     polarGrid;
    nRadial 16;
    nAzimutal 16;
    sampleMode center | closestCell | center | cellMean;
    includeIfVertex true;
}
```

Donde `nRadial` es el número de celdas en dirección radial, y `nAzimutal` el número en dirección azimutal. Para el caso del mallado `meshGrid` se define de la siguiente forma:

```
rotorGrid
{
    type     meshGrid;
    sampleMode center | closestCell | center | cellMean;
    discreteMethod intersection | voronoi | proyection;
    includeIfVertex true;
    borderRefinement 10;
}
```

En este caso aparece el atributo `borderRefinement` que indica el numero de iteraciones del algoritmo de refinado que se tiene que aplicar. Este algoritmo es detallado en la Subsección 2.4.7. Para el caso del mallado de tipo `bladeGrid`, que es utilizado para el cálculo transitorio se pueden utilizar los atributos siguientes:

```
rotorGrid
{
    type    bladeGrid;
    nRadial 16;
    chord 0.06; (default none)
    sampleMode center | closestCell | center | cellMean;
    includeIfVertex true;
}
```

En este caso el atributo *chord* es opcional cuando se utiliza el método `bladeElementMethod`. En ese caso utiliza los datos de distribución de cuerda de la pala para obtener la geometría de la malla. En su defecto, si se utiliza otro método que no dispone de datos de la pala o la malla es demasiado gruesa puede especificarse un valor de cuerda constante. Esto no variara el valor de la cuerda utilizada para el cálculo de las fuerzas, tan solo afecta a la geometría de la malla. Esta malla es transitoria y es actualizada en cada paso temporal según la velocidad de rotación.

control

La configuración de la clase control se reliza en función del tipo de control requerido:

- `fixedControl`: los parámetros de control son especificados por el usuario.
- `targetValue`: los parámetros de control se obtienen a partir de los valores objetivos utilizados.

Para el caso de `fixedControl`, simplemente hay que añadir las variables de control al diccionario. Se muestran a continuación las variables de control disponibles para `bladeElementModel`:

```
control
{
    type fixedControl;
    rpm 7000;
```

```
5
6       collectivePitch 0.0;
7       ciclicPitchCos 0.0;
8       ciclicPitchSin 0.0;
9
10      flapping    0.0;
11      cosFlapping 0;
12      sinFlapping 0;
13
14      cosAzimuth 0.0;
15      sinAzimuth 0.0;
16  }
```

Para el caso de `forceModel` la única variable de control es `J`.

La configuración de `targetValue` presenta un mayor número de parámetros, mostrados a continuación, los disponibles para el caso de `bladeElementModel`:

```
1   control
2   {
3       type targetValue;
4       isRadian false;
5
6       //-Parametros de Newton-Raphson
7       dx (0.1 0.1);
8       relax 0.9;
9       calcFrequency 1;
10      nIter 100;
11      tol   1e-8;
12
13      //- Valores objetivo
14      forceZ 15; //Target value
15      power 250; //Target value
16
17      controlVariables (angularVelocity collectivePitch);
18  }
```

En primer lugar se encuentra el atributo `isRadian` que indica si las variables que representan ángulos son introducidas en grados o radianes. Después se indican los parámetros de configuración de Newton-Raphson, válidos para todos los `propellerModel` y su explicación es la siguiente:

- `dx`: el incremento de variable utilizado para el cálculo de las derivadas parciales mediante diferencias finitas. Se especifica en orden para cada variable de control.
- `relax`: factor de relajación en las iteraciones del método (véase Subsección 2.3.2).
- `calcFrecuency`: cada cuantas iteraciones de volúmenes finitos es necesario recalcular el valor de la variables de control.
- `nIter`: número máximo de iteraciones del método.

- `tol`: tolerancia máxima del error obtenido.

Seguidamente se indican el valor de los `targetValues`. Para `bladeElementModel` están disponibles los siguientes:

- `forceX`
- `forceY`
- `forceZ`: empuje del rotor.
- `torqueX`
- `torqueY`
- `torqueZ`: par del rotor.
- `power`

Y las siguientes variables de control indicadas en una lista mediante el atributo llamado `controlVariables`:

- `angularVelocity`
- `collectivePitch`
- `ciclicPitchCos`.
- `ciclicPitchSin`

Para el caso de `forceModel` solo se disponen como `targetValues`: `forceZ`, `torqueZ` y `power`. Y como variables de control solamente: `angularVelocity`.

Es posible escoger cualquier combinación de variables de control y variables objetivo, pero ha de cumplirse que el número de ambas sea el mismo.

5.2.6. propellerModel: bladeElementModel

Para la configuración del diccionario de `propellerModel` del tipo `bladeElementModel` se requieren los siguientes atributos:

```
1  propellerModel
2  {
3      type            bladeElementModel;
4      nBlades         3;
5      tipFactor       0.96;                       //(default 1)
6      polarCorrection none | snel | snelPumping;  //(default none)
7      //- valores de referencia
8      speedRef        10;
9      rhoRef          1.225;
10     viscosity       1e-5;
11     soundSpeed      345;
```

```
12
13      outputFields    (aoa cd cl);                    //(default ())
14      rotorGrid
15      {
16          //...
17          //Comentado anteriormente
18          //...
19      }
20      control
21      {
22          //...
23          //Comentado anteriormente
24          //...
25      }
26      bladeModel
27      {
28      }
29      airfoils
30      {
31      }
32  }
```

Siendo `tipFactor` el radio adimensional a partir del cual la sustentación es nula y `polarCorrection` el método de corrección de polares explicado en la Subsección 2.1.5. Los diccionarios `bladeModel` y `airfoils` son explicados a continuación.

bladeModel

Mediante este diccionario se define la geometría y perfiles que componen la pala, para ello se indica la distribución de `chord`, `twist`, `sweep` y `airfoil`. Hay dos métodos para realizarlo: mediante secciones (lista de radios común) o mediante distribuciones individuales. El uso de secciones se realiza de la siguiente forma:

```
1  bladeModel
2  {
3      isRadian     false;
4      from         sections;
5
6      sections
7      (
8          //(airfoilName ( radialPos[m] chord[m] torsion  sweep ))
9          (airfoil1 (0.0 0.0 50.0 0.0))
10         (airfoil2 (0.5 0.06 20.0 0.0))
11         (airfoil3 (1.0 0.02 10.0 0.0))
12     )
13 }
```

Donde el atributo `isRadian` indica si los ángulos son introducidos en radianes. El atributo `from` especifica el método de *input*: `section` o *properties*(distribuciones individuales). La

forma de introducir las secciones se observa en el ejemplo. Para introducirlo mediante `properties` se realiza de la siguiente forma:

```
bladeModel
{
    isRadian        false;
    from            properties;
    csv             "csvFile.csv";

    chord
    {
        from        csv;
        radius      r;
        chord       c;
        cubicSpline false; //(default false)
    }
    twist
    {
        from    csv;
        csv     "otherCsv.csv"
        radius  r;
        twist   beta;
    }
    sweep
    {
        from constant;
        sweep   0;
    }
    airfoil
    {
        from list;
        radius (0.02 0.44 0.45 0.75 1);
        airfoil   (cylinder cylinder SDA0_45 SDA0_75 SDA0_100);
    }

}
```

Para introducir cada distribución se utiliza el diccionario correspondiente a cada propiedad: `chord`, `twist`, `sweep` y `airfoil`. Todos pueden ser introducidos mediante csv, lista o valor constante, excepto `airfoil` que no puede ser realizado mediante csv.

5.2.7. airfoils

En este diccionario se definen los `airfoil` utilizados por `bladeModel` , siendo referenciados mediante el nombre que se le asigna. Para definir este diccionario se realiza de la siguiente forma:

```
airfoils
{
```

```
    airfoilName1
    {
        type cylinderAirfoil;
    }
    airfoilName2
    {
        type simpleAirfoil;
    }
    airfoilName3
    {
        type polarAirfoil;
    }
    airfoilName4
    {
        type interpolatedAirfoil;
    }
}
```

En función del `type` de `airfoil` definido, presenta unas opciones de configuración diferentes. Aunque tienen una serie de parámetros en común para la exportación que se definen a continuación:

```
airfoilName1
{
    type    cylinderAirfoil | simpleAirfoil |
            polarAirfoil | interpolatedAirfoil;
    export true; //(default false)
    aoaBegin  -pi;
    aoaEnd     pi;
    nAoa       360;

    Reynolds (0 1000 10000); //(default ())
    Mach     (0 0.2 0.3);    //(default ())
    join false;              //(default false)
}
```

Utilizando estos atributos se activa la exportación automática del perfil. Se selecciona los ángulos de ataque, Reynolds y Mach para los que se requiere la exportación. Con el atributo `join` se indica si se quiere generar una tabla para cada combinación de Reynolds y Mach o se generan todos en una única.

Para la configuración de `cylinderAirfoil` solo es necesario indicar el c_{D0}:

```
cylinder1
{
    type cylinderAirfoil;
    cd0 0.6;
}
```

Para la configuración de `simpleAirfoil` se indican los siguientes valores:

```
simple1
```

```
{
    type simpleAirfoil;
    cl0   0.05;
    cl_max 1.2;
    cd0   0.1;
    cd_max 1.5;
}
```

Que definen los valores mínimos y máximos de los coeficientes de sustentación y resistencia. Para el caso de `polarAirfoil` hay que indicar el csv donde se incluyen los datos de las polares y una serie de atributos que se muestran a continuación:

```
polar1
{
    type         polarAirfoil;
    interpolation closestNeighbor;
    extrapolation polar|viterna;
    isRadian true;
    cubicSpline true;
    csv "system/airfoils/SDA/SDA_1045.csv";
    //colum name for angleOfAttack CL CD RE
    col (AoA Cl Cd Re Ma);
    logReynolds true; //(default false)
}
```

El atributo `interpolation` define el método de interpolación utilizado, `extrapolation` indica el método de extrapolación de polares: `polar` no realiza ninguna extrapolación (valor constante) y `viterna` aplica el método desarrollado en la Sección 2.2. El atributo `isRadian` indica si los datos de ángulo de ataque son proporcionados en radianes, `csv` la ruta del fichero csv con los datos de las polares y `col` el nombre que reciben las columnas de ángulo de ataque, coeficiente de sustentación, coeficiente de arrastre, número de Reynolds y número de Mach dentro del `csv`. Han de indicarse en el orden mencionado. Finalmente, el atributo `logReynolds` indica si internamente se utiliza el número de Reynolds en escala logarítmica para mejorar el resultado de la interpolación.

5.2.8. propellerModel: forceModel

Para el uso de este modelo, es necesario indicar los siguientes atributos en el diccionario de forma similar a `bladeElementModel`:

```
propellerModel
{
    type            forceModel;

    //- valores de referencia
    rhoRef          1.225;

    outputFields    (fmForce); //(default ())
    rotorGrid
    {
```

```
        //...
        //Comentado anteriormente
        //...
    }
    control
    {
        //...
        //Comentado anteriormente
        //...
    }
    curves
    {
    }
}
```

El único diccionario que aparece por primera vez es `curves`, comentado a continuación.

curves

El diccionario `curves` indica las curvas de coeficiente de empuje y de par utilizadas para el cálculo del rotor. Se pueden indicar mediante una lista o mediante un csv de las siguientes formas:

```
curves
{
    from list;
    J   (0 0.5 1);
    CT  (0 0.01 0.02);
    CQ  (0 0.0034 0.0064);
}
```

```
curves
{
    from csv;
    csv csvFile.csv;
    J   j;
    CT  ct;
    CQ  cq;
}
```

Capítulo 6

Resultados

En el presente capítulo, se comentarán los resultados principales obtenidos: la verificación de la implementado del código comparándolo con simulaciones realizadas mediante Star-CCM+ y validación de los resultados obtenidos usando medidas experimentales realizadas previamente al propio trabajo en el departamento CMT (Centro de Motores Térmicos) de la UPV.

6.1. Verificación

La verificación consiste en comprobar que la implementación del modelo es correcta y que se están resolviendo correctamente las ecuaciones planteadas. Para ello se ha utilizado como modelo comparativo simulaciones realizadas con el *software* comercial Star-CCM+ que incorpora el modelo de teoria de elemento de pala con el mallado polar.

Para realizar la verificación se ha creado una malla en OpenFOAM y se ha exporta a Star-CCM+. Se ha intentado mantener todas las condiciones de la simulación de forma idéntica: malla, condiciones de contorno, métodos de discretización, ecuaciones resultas, geometría y parámetros del rotor. Los datos de la geometría de la pala y polares han sido obtenidos en [30].

Las simulaciones se han realizado para dos tipos de malla en OpenFOAM: polar y Voronoi. En el caso de Star-CCM+ solo se implementa la malla polar, utilizándose solo esta en su defecto. En la Figura 6.1 se muestra la comparación del coeficiente de empuje. Se observa mayores discrepancias entre los resultados para valores de J inferiores, convergiendo las curvas para valores de J más elevados. En el caso del coeficiente de par, los resultados son aún más diferentes entre si para valores pequeños de J y convergiendo para valores más elevados como se puede observar en Figura 6.2.

En la Figura 6.3 y Figura 6.4 se comparan los errores relativos de los dos métodos de OpenFOAM respecto los valores de Star-CCM+. La tendencia de las curvas es la esperada. Para el caso del error del coeficiente de empuje, el valor del error es inferior que el del coeficiente de par y ambos se reducen al aumentar el valor de J. Para el último valor de J se observa un pico en las curvas. Este fenómeno es debido a que el valor de los coeficientes tiende a 0 y el error se amplifica.

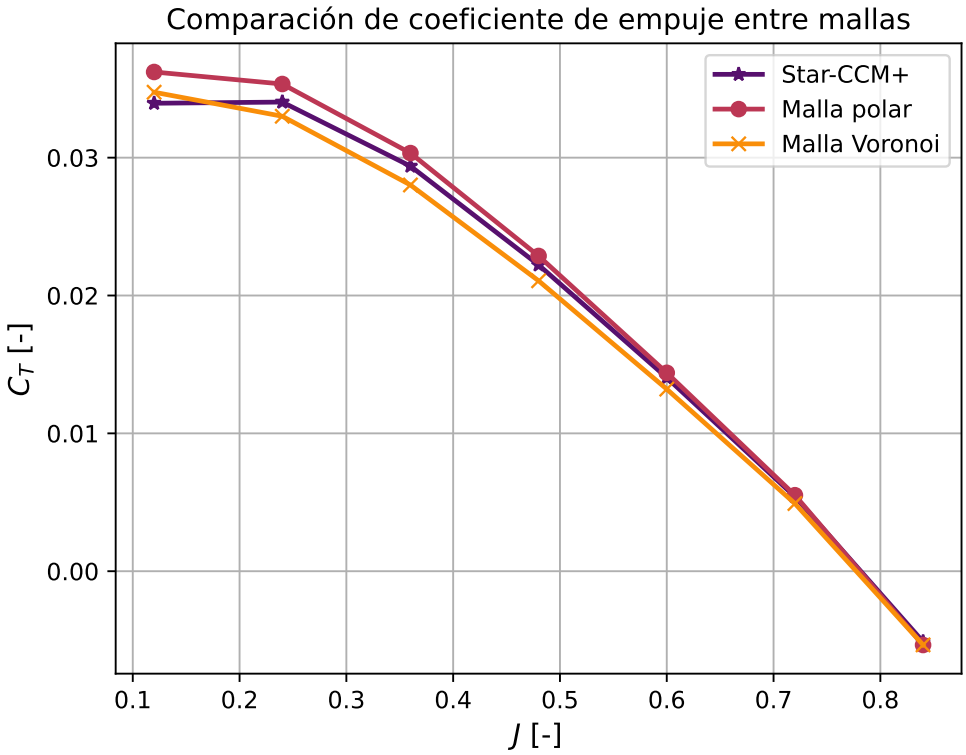

Figura 6.1: Comparación del coeficiente de empuje entre Star-CCM+ y diferentes mallados de BET.

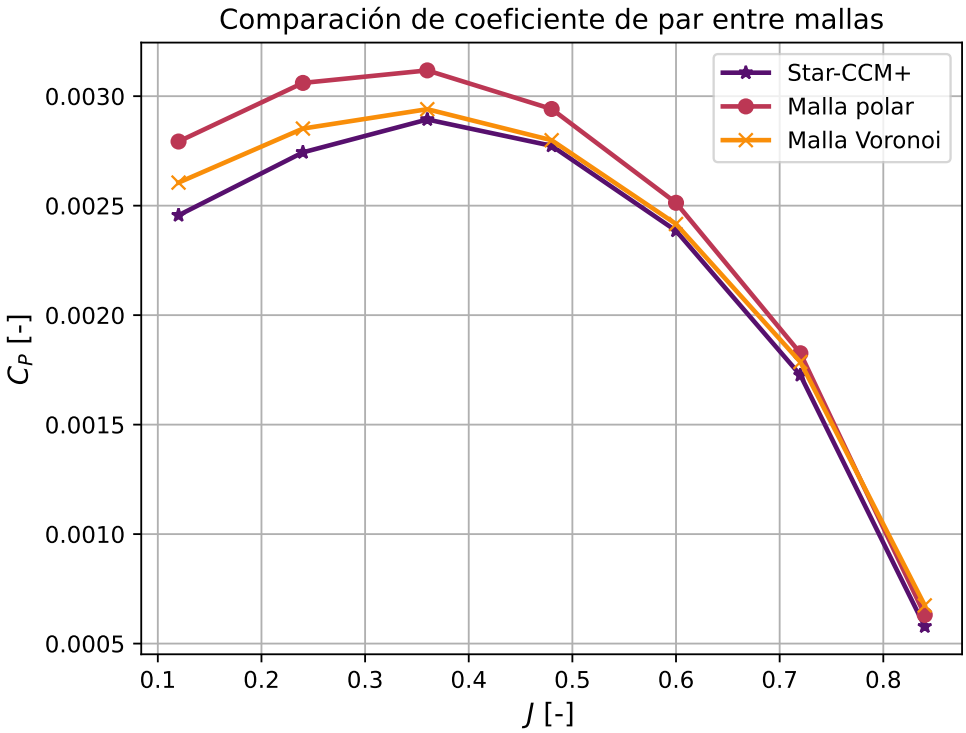

Figura 6.2: Comparación del coeficiente de par entre Star-CCM+ y diferentes mallados de BET.

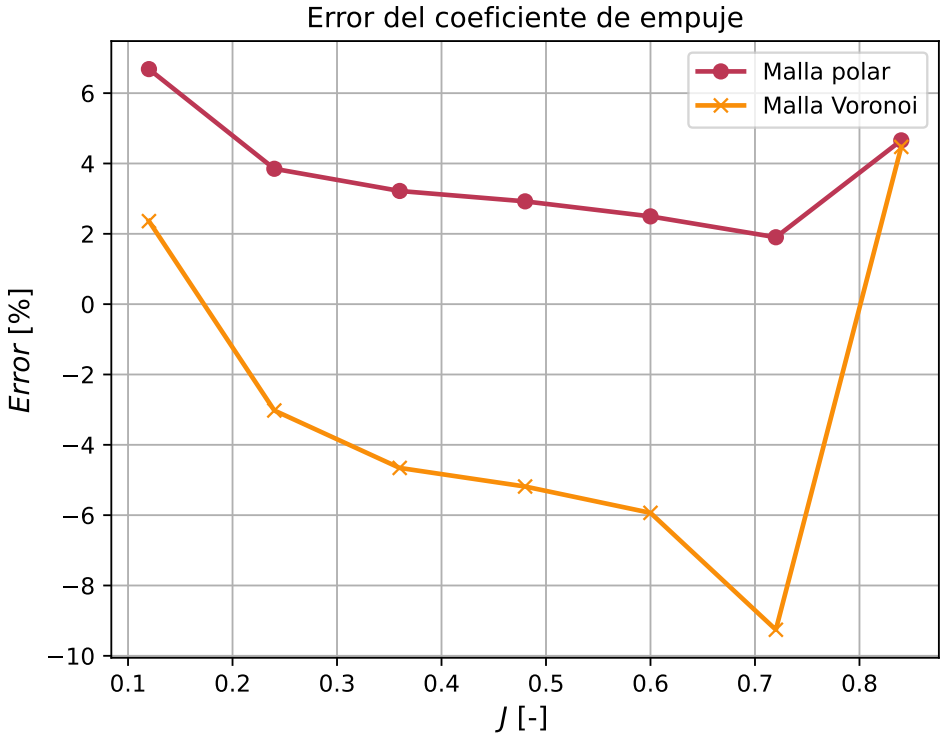

Figura 6.3: Comparación del error del coeficiente de empuje entre Star-CCM+ y diferentes mallados de BET.

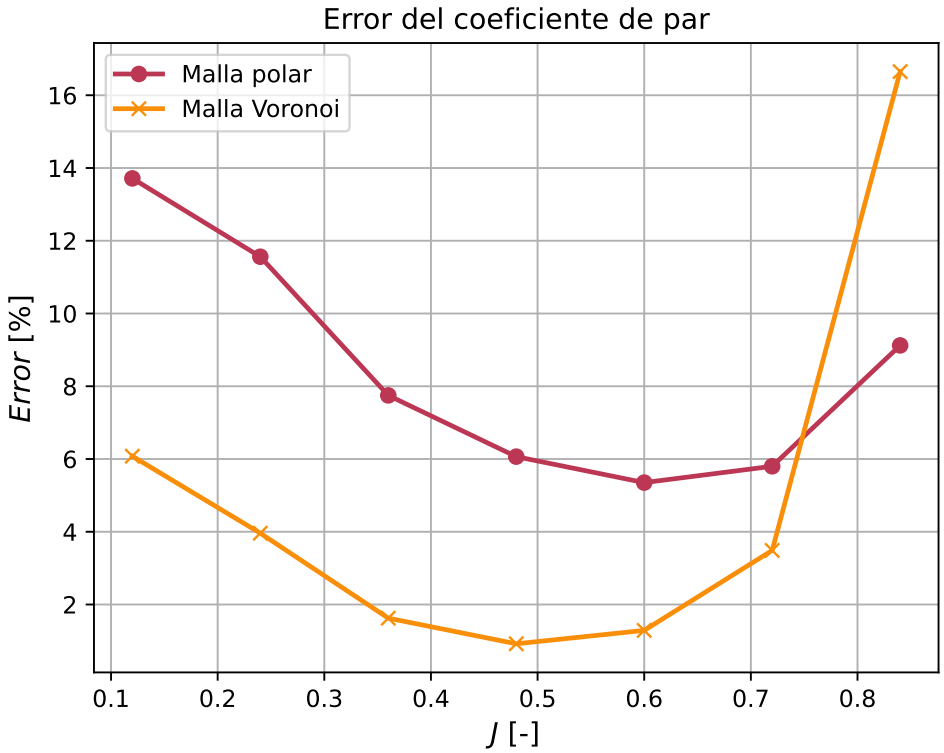

Figura 6.4: Comparación del error del coeficiente de par entre Star-CCM+ y diferentes mallados de BET.

No obstante, las diferencias entre los métodos son muy elevadas pese a que los casos están configurados de forma idéntica. Observando el campo de velocidades de la solución, se observan discrepancias muy pequeñas de los valores obtenidos en OpenFOAM respecto a los de Star-CCM+, esto no debería ser el caso para la malla polar.

Para salir de dudas se decide exportar el campo fluido de la solución de Star-CCM+ e importarlo a OpenFOAM para el cálculo de los coeficientes del rotor. Esto es posible gracias a la implementación de la clase `fileSampler` que obtiene los valores de la velocidad a partir del archivo exportado desde Star-CCM+. Eliminadas todas la variables que pudieran afectar a la solución, se observa en la Figura 6.5 y en la Figura 6.6 los resultados obtenidos mediante la malla polar en OpenFOAM y Star-CCM+ coinciden hasta el cuarto decimal, no siendo tan exactos para el caso de la malla Voronoi. Esto indica que la configuración con los mismos parámetros y con los mismos datos de partida producen los mismos resultados, permitiendo verificar la implementación del método y concluir que pequeñas diferencias en la resolución de las ecuaciones de N-S provocan grandes cambios en los resultados obtenidos. Se observa alta sensibilidad del método al campo de velocidades percibido.

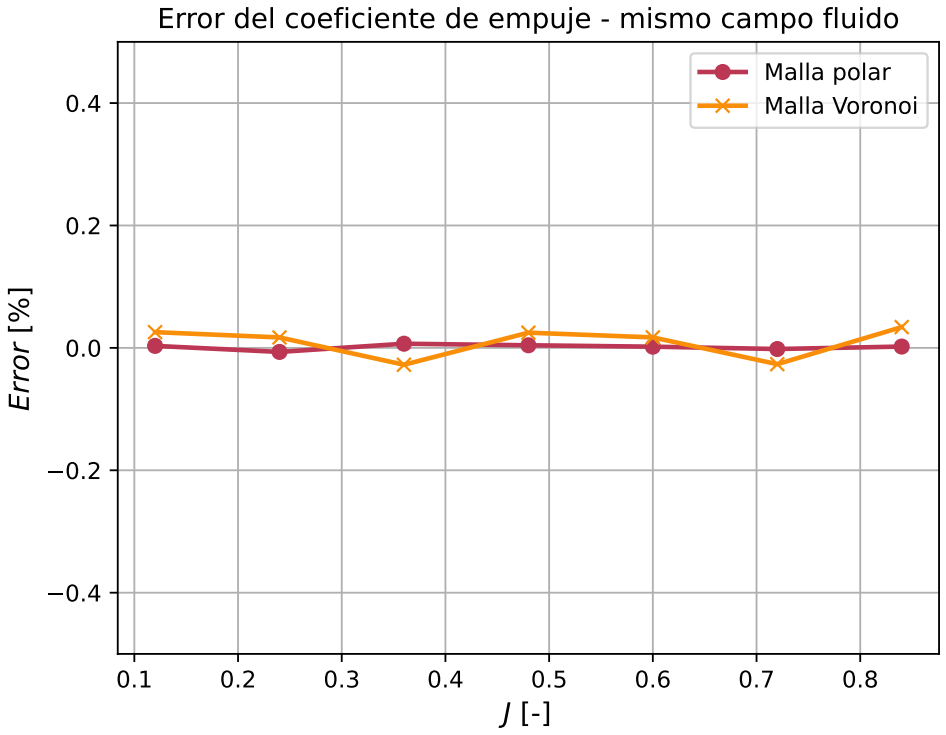

Figura 6.5: Comparación del error del coeficiente de empuje entre Star-CCM+ y diferentes mallados de BET para el mismo campo fluido.

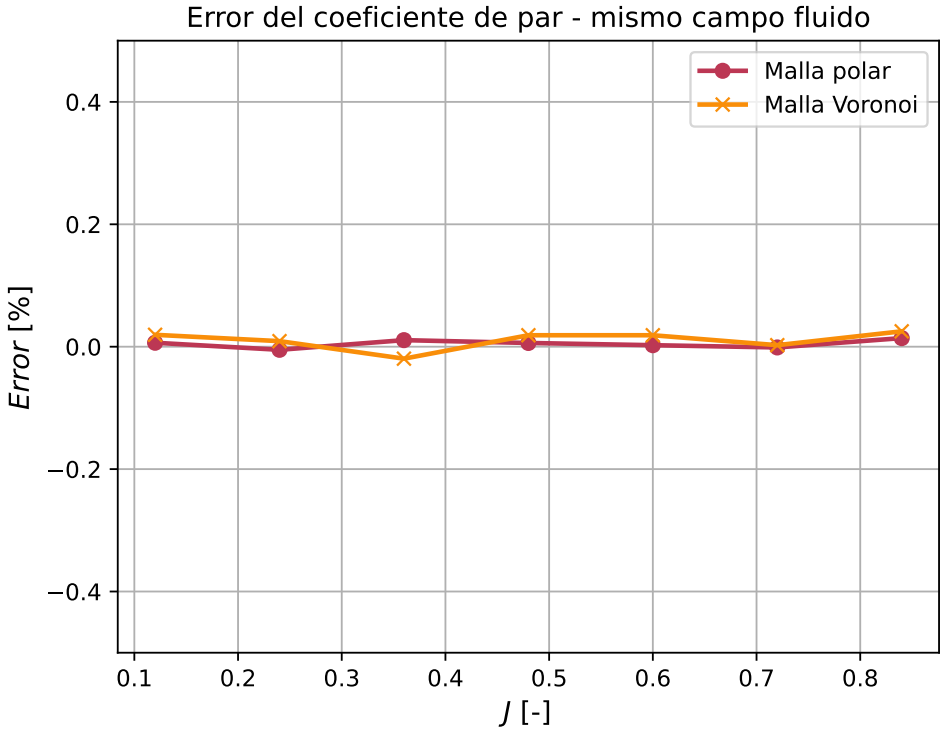

Figura 6.6: Comparación del error del coeficiente de par entre Star-CCM+ y diferentes mallados de BET para el mismo campo fluido.

6.2. Validación

La validación consiste en comprobar que los modelos utilizados representan correctamente la realidad hasta cierto grado. Esta puede realizarse de distintas formas, aunque las más fiables pasan a través de técnicas experimentales.

Se disponen datos experimentales de la hélice SDA proporcionados por el CMT obtenidos en un túnel de viento. Además se cuenta también con los datos de las polares a distintas secciones y de la geometría, obtenidos mediante escaneo 3D. A continuación, se comparan los valores obtenidos mediante los ensayos experimentales y con la aplicación del método BET + CFD.

En la Figura 6.7 se observa la comparación de los datos de empuje obtenidos experimentalmente y numéricamente. Se observa que la discrepancia en las curvas es pequeña, manteniéndose el error acotado alrededor del 10 %. También se puede observar que las tendencias de ambas curvas es similar, aumentando el error cometido para bajos valores de J. Esto podría deberse que a altas revoluciones aparecen fenómenos que el modelo del BET no está teniendo en cuenta.

En la Figura 6.8 se comparan los valores de potencia consumida en el ensayo experimental y potencia propulsiva de la hélice obtenida numéricamente. Como se ha indicado, las variables comparadas no son las mismas, ya que en la potencia consumida experimentalmente se incluyen las pérdidas mecánicas y eléctricas. Si se asumen valores típicos de eficiencia mecánica y eléctrica alrededor del 95 %, se obtienen valores similares de potencia propulsiva, con la misma tendencia del error que para el caso del coeficiente de empuje

como se puede observa en la Figura 6.9. Teniendo en cuenta esto, la potencia propulsiva deberá ser inferior en todo momento a la medida experimentalmente. Los datos obtenidos concuerdan adecuadamente con esto.

Teniendo en cuenta estos resultados, se puede concluir que el método desarrollado en el presente trabajo ha quedado validado tanto en empuje como en potencia usando medidas experimentales en túnel de viento de la misma hélice simulada.

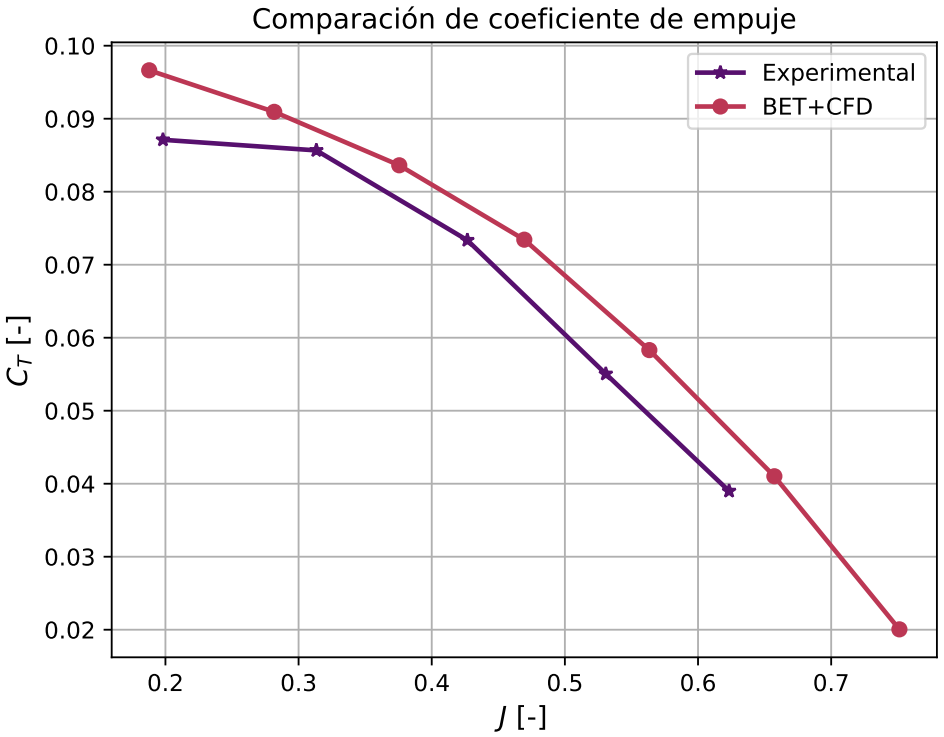

Figura 6.7: Comparación del coeficiente de empuje del rotor experimental y mediante BET+CFD.

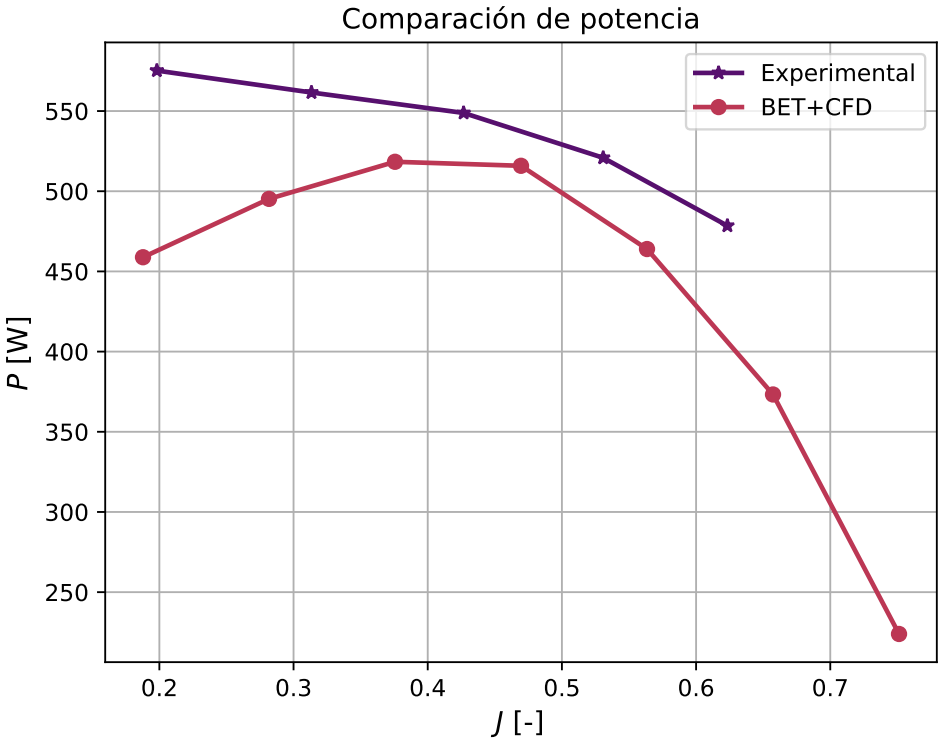

Figura 6.8: Comparación de la potencia del rotor experimental y mediante BET+CFD.

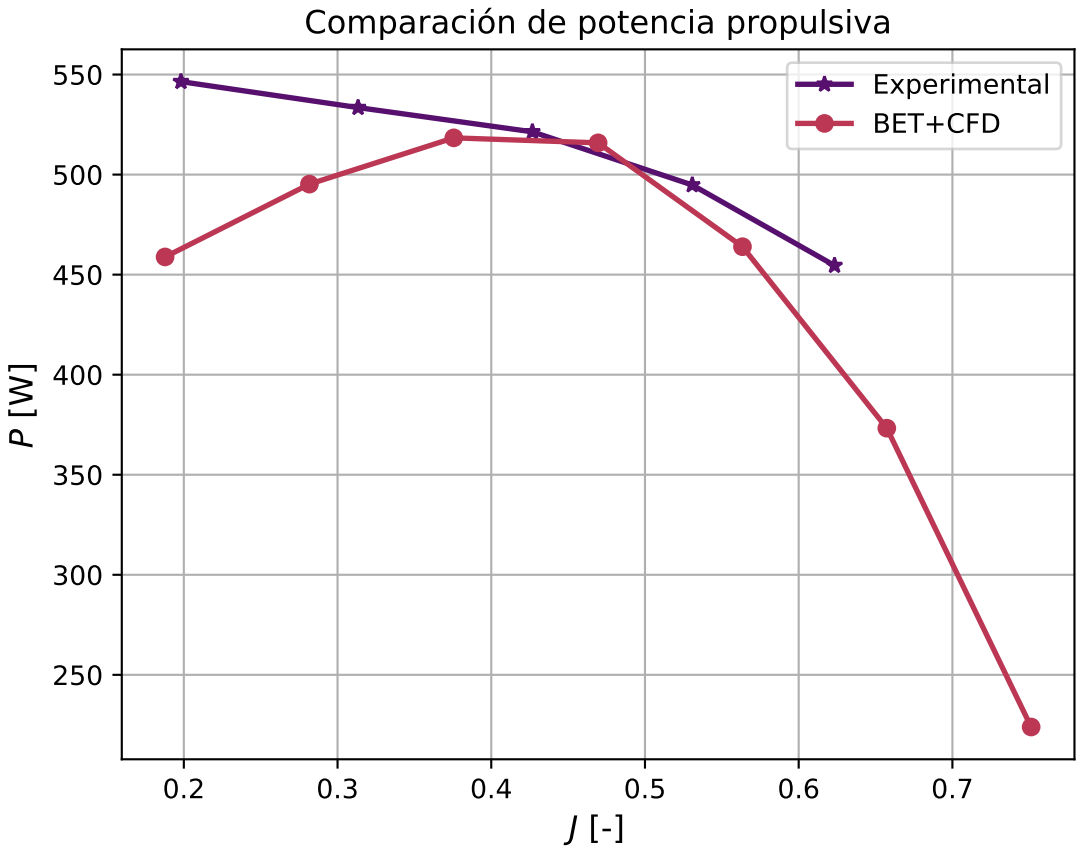

Figura 6.9: Comparación de la potencia propulsiva estimada del rotor experimental y mediante BET+CFD.

Capítulo 7

Presupuesto

En este capítulo, se proporcionará un análisis de los costes del proyecto, divididos en tres categorías: costes humanos, costes de equipo y costes de *software*.

7.1. Costes humanos

La evaluación precisa de los costes humanos puede resultar compleja. Partiremos del sueldo medio en España para un graduado en ingeniería aeroespacial, el cual es de 40 100 euros brutos al año [11]. A través de esta información, calcularemos el costo por hora al dividir dicho sueldo entre las horas totales de trabajo anual aproximadas. La tabla 7.1 resume los costes correspondientes al personal involucrado en la realización del proyecto.

Personal	Horas	Sueldo (€/h)	Subtotal (€)
Asesoría	20	50	1000
Alumno	600	15	9000
		Total del Proyecto (€)	10000

Tabla 7.1: Costes de personal.

7.2. Costes de equipo

La ejecución del proyecto se llevó a cabo en un ordenador portátil de uso personal, específicamente el modelo ASUS GL552vw, que estaba acompañado por un monitor de la marca Dell, y un ratón y teclado de la marca Logitech. Los gastos asociados a estos equipos se registrarán proporcionalmente al tiempo de uso que tuvieron en el proyecto en relación a su vida útil estimada.

Equipo	Precio de compra (€)	Uso (meses)	Vida útil (años)	Subtotal (€)
Portátil	1250	6	4	156.25
Monitor	350	6	10	17.5
Teclado	80	6	11	3.63
Ratón	50	6	2	12.5
		Total del Proyecto (€)		189.88

Tabla 7.2: Costes de equipo.

7.3. Costes de software

En el presente apartado, se abordarán las licencias de software utilizadas en el desarrollo de este proyecto. Uno de los pilares fundamentales de nuestra estrategia ha sido la adopción de software libre, el cual nos ha permitido aprovechar una amplia variedad de herramientas y soluciones de código abierto. La elección de este enfoque se ha fundamentado en los principios de accesibilidad, transparencia, colaboración y libertad, que son características inherentes al software libre.

El software libre ha demostrado ser una opción versátil y altamente eficiente, proporcionándonos la flexibilidad necesaria para adaptar y personalizar las soluciones a las necesidades específicas de este proyecto. Asimismo, ha permitido un ahorro significativo en costos de licencias, permitiéndonos redirigir recursos hacia otras áreas clave del trabajo. En la Tabla 7.4 se muestra una lista de las licencias de software utilizadas y sus costes asociados.

Tabla 7.3: Presupuesto y Licencias de Software

Software	**Licencia**	**Subtotal(€)**
OpenFOAM	GPL (GNU General Public License)	0
FreeCAD	LGPL (GNU Lesser General Public License)	0
Visual Studio Code	MIT License	0
Ubuntu	GPL (GNU General Public License)	0
ParaView	BSD (Berkeley Software Distribution)	0
Star-CCM+	Comercial	480
Overleaf	Comercial	54
	Total del Proyecto (€)	**534**

Tabla 7.4: Costes de *software*

Los costes de Star-CCM+ y Overleaf se cuentan por el uso dado. Para el caso de Star-CCM+ se ha utilizado durante 20 horas, con un coste de licencia por hora de 24€. La facturación de Overleaf se realiza mensualmente, con un precio de 9€ al mes durante 6 meses.

7.4. Costes Totales

En la Tabla 7.5 se muestran los costes totales del proyecto. El total asciende hasta los diez mil setecientos veintitrés euros con ochenta y ocho céntimos. Se observa que la mayor parte de los gastos provienen del coste de las horas de trabajo seguido por el *Software* utilizado.

Categoría	Coste (€)
Costes humanos	10000
Costes de equipo	189,88
Costes de *software*	534
TOTAL	**10 723,88**

Tabla 7.5: Costes de totales

Capítulo 8

Conclusiones

En esta sección de conclusiones recopilaremos los hallazgos y resultados más relevantes obtenidos a lo largo del trabajo. Se presentarán las principales observaciones y lecciones aprendidas durante el proceso, destacando los logros alcanzados y los desafíos identificados. Asimismo, se proporcionarán recomendaciones y posibles áreas de mejora para futuros proyectos o investigaciones. A través de esta síntesis, buscamos brindar una visión global y significativa del impacto y el valor del trabajo realizado.

8.1. Conclusiones del trabajo

El desarrollo del presente trabajo ha permitido aplicar los conocimientos obtenidos durante el grado de Ingeniería Aeroespacial y el máster en Ingeniería Aeronáutica a un problema real, respondiendo a la necesidad de desarrollar métodos más eficientes para el cálculo y interacción de rotores en situaciones de flujo complejas, como es el caso de la propulsión distribuida y la ingestión de capa límite.

Además de aplicar los conocimientos y capacidades desarrollados, la realización del presente trabajo ha permitido obtener un profundo conocimiento en el lenguaje de programación C++, utilizando elementos y patrones avanzados de programación. También se ha profundizado en el conocimiento de uno de las librerías más utilizadas en la industria, OpenFOAM. El aprendizaje se ha realizado tanto a nivel de usuario, al preparar y ejecutar los caso, como a nivel de desarrollador, comprendiendo y analizando el funcionamiento del código de OpenFOAM desde sus elementos más básicos hasta sistemas completos como los *solvers*.

El mayor desafío enfrentado durante este proyecto ha sido el aprendizaje y uso de las herramientas de OpenFOAM, dado que este software cuenta con una extensa cantidad de archivos y código, lo que implica una curva de aprendizaje significativa. Esta complejidad ha requerido un esfuerzo adicional para el dominio de la herramienta.

Se ha logrado cumplir con los objetivos del trabajo, realizando una exitosa implementación del método de elemento de pala integrado en una arquitectura más amplia, conformando la herramienta ***PropellerSource***, disponible bajo la licencia **MIT** en **GitHub** (`https://github.com/FredyTP/PropellerSourceOpenFOAM`). Esta permite utilizarse como entorno de desarrollo y proporcionar las herramientas para la implementación de otros métodos o la mejora de los implementados. Como ejemplo de ello, se ha implementado aparte del BET el método *body force*. Se ha logrado también una implementación compa-

tible con la ejecución en paralelo del código integrando una gran variedad de métodos y opciones que han dificultado la tarea.

La verificación del código se ha realizado comparando con un caso idéntico en Star-CCM+, mostrado una gran sensibilidad del método ante factores como la malla, la discretización y el método de resolución de CFD.

Durante la implementación del método, se han desarrollado una serie de herramientas que pueden ser utilizadas en otros ámbitos, utilizando siempre un enfoque modular y desacoplado. Algunos ejemplos de ello son: los métodos de interpolación multidimensional y genéricos, los modelos de perfiles aerodinámicos, los algoritmos geométricos (como la triangulación de Delaunay o el diagrama de Voronoi), la utilidad *GatherList* o el manejo de archivos csv.

8.2. Posibles mejoras

En todo trabajo bien ejecutado siempre existe cierto margen de mejora. A continuación, se comentarán aspectos en los que se podría mejorar el código realizado y algunas formas de implementarlos:

- La clase `airfoilModelList` se instancia para cada uno de los *propellers* definidos, usando tiempo de ejecución y memoria adicional. Una solución pasa por definir esta instancia de forma estática, de modo que es compartida entre todas las instancias de *propellerModel*, evitando tener que instanciar los *airfoils* más de una vez.
- La clase `csvTable` tiene poca versatilidad para modificar los datos y su uso puede resultar confuso. Se propone añadir métodos de modificación de valores, filas o columnas.
- El mallado del rotor de tipo polar presenta poca versatilidad. El mallado generado es equiespaciado y estructurado. Se propone añadir más versatilidad al espaciado en dirección radial, y en modificar el número de elementos en dirección azimutal, aumentando el número de estos elementos para valores radiales mayores.
- La clase `rotorControl` y derivadas deberían implementar el algoritmo genérico para `fixedValue` y `targetControl` de forma abstracta sin depender del tipo de *propellerModel* utilizado.
- El paralelizado de volúmenes finitos podría extenderse al calculo del rotor, reduciendo la carga por núcleo y minimizando la cantidad de datos comunicados. Se prevé que el impacto de esto sea pequeño.

8.3. Trabajos futuros

Dado que uno de los objetivos del trabajo es sentar las bases para el desarrollo de modelos de rotor en OpenFOAM, se proponen a continuación una lista de posibles desarrollos futuros:

- Incluir modelos no-estacionarios para las fuerzas aerodinámicas del método BET.

- Incluir métodos para el cálculo de la velocidad inducida en el propio código: BEMT, *Wake Vortex*, método de los paneles, etc.
- Implementar cálculos de la dinámica del rotor como sólido rígido.
- Incluir los efectos aeroelásticos utilizando métodos de elementos finitos.
- Mejorar la definición de la geometría de la pala en el mallado de pala para incluir deformaciones fuerza del plano o geometrías más complejas.

Bibliografía

[1] P Bourke. *Calculating the Area and Centroid of a Polygon*. http://paulbourke.net/geometry/polygonmesh. Accedido 25-04-2023. 1988.

[2] Paul Bourke. "An Algorithm for Interpolating Irregularly-Spaced Data with Applications in Terrain Modelling". En: Accedido 20-03-2023. 1989.

[3] Emmanuel Branlard. *Wind Turbine Aerodynamics and Vorticity-Based Methods*. Vol. 7. Ene. de 2017. ISBN: 978-3-319-55163-0. DOI: 10.1007/978-3-319-55164-7.

[4] Thomas T. Cormen, Charles E. Leiserson y Ronald L. Rivest. *Introduction to Algorithms*. Cambridge, MA, USA: MIT Press, 1990. ISBN: 0262031418.

[5] Microsoft Corporation. *Visual Studio Code*. https://code.visualstudio.com/. Accedido 01-02-2023.

[6] dicehub. *Hex-dominant mesher (snappyHexMesh)*. https://docs.dicehub.com/latest/template/hex_dominant_mesher/. Accedido 5-07-2023.

[7] P.S. Heckbert. *Graphics Gems IV*. Graphics gems series. AP Professional, 1994. ISBN: 9780123361554. URL: https://books.google.es/books?id=CCqzMm%5C_-WucC.

[8] Manfred Imiela y col. "Numerical airfoil catalogue including 360° airfoil polars and aeroacoustic footprints". En: *Wind Energy Science Discussions* (nov. de 2017), págs. 1-34. DOI: 10.5194/wes-2017-51.

[9] Manfred Imiela y col. "Numerical airfoil catalogue including 360° airfoil polars and aeroacoustic footprints". En: *Wind Energy Science Discussions* (nov. de 2017), págs. 1-34. DOI: 10.5194/wes-2017-51.

[10] Docker Inc. *Docker*. https://www.docker.com. Accedido 01-05-2023.

[11] Jobted. *Sueldo del Ingeniero Aeronáutico en España*. https://www.jobted.es/salario/ingeniero-aeronautico. Accedido 27-07-2023.

[12] C. Lindenburg. *Investigation into Rotor Blade Aerodynamics*. Jun. de 2003.

[13] C. Lindenburg. "Modeling of rotational augmentation based on engineering considerations and measurements". En: (ene. de 2004).

[14] OpenCFD Ltd. *OpenFOAM*. https://www.openfoam.com. Accedido 20-01-2023.

[15] OpenCFD Ltd. *OpenFOAM Documentation*. https://www.openfoam.com/documentation. Accedido 22-02-2023.

[16] OpenCFD Ltd. *OpenFOAM snappyHexMesh*. https://www.openfoam.com/documentation/user-guide/4-mesh-generation-and-conversion/4.4-mesh-generation-with-the-snappyhexmesh-utility. Accedido 05-01-2023.

[17] OpenCFD Ltd. *OpenFOAM User Guide*. https://www.openfoam.com/documentation/user-guide. Accedido 22-02-2023.

[18] OpenCFD Ltd. *OpenFOAM Utilities*. https://www.openfoam.com/documentation/user-guide/a-reference/a.2-standard-utilities. Accedido 22-02-2023.

[19] OpenFOAM Ltd. *OpenCFD Source Code*. https://develop.openfoam.com/Development/openfoam/-/blob/master/doc/Build.md. Accedido 01-02-2023.

[20] Faisal Mahmuddin. "The Effect of Flat Plate Theory Assumption in Post-Stall Lift and Drag Coefficients Extrapolation with Viterna Method". En: 2016.

[21] MIT. "Multidimensional-Newton". En: *MIT* (2017). URL: https://web.mit.edu/18.06/www/Spring17/Multidimensional-Newton.pdf.

[22] Gerwyn Ng. *Natural Cubic Splines Implementation with Python*. 2019.

[23] A. Rosen y Ohad Gur. "A novel approach to actuator disk modeling". En: *National Aerospace Laboratory NLR - 32nd European Rotorcraft Forum, ERF 2006* 3 (ene. de 2007), págs. 1207-1241.

[24] Donald Shepard. "A two-dimensional interpolation function for irregularly-spaced data". En: *ACM-NATIONAL-CONFERENCE*. 1968. DOI: 10.1145/800186.810616.

[25] Siemens Digital Industries Software. *Simcenter STAR-CCM+ User Guide v. 2021.1*. Ver. 2021.1. 2021.

[26] Houwink R. Snel H. y Bosschers J. "Sectional prediction of lift coefficients on rotating wind turbine blades in stall". En: (dic. de 1994).

[27] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2013. ISBN: 978-0-321-56384-2.

[28] Khiem Truong. "An analytical model for airfoil aerodynamic characteristics over the entire 360° angle of attack range". En: *Journal of Renewable and Sustainable Energy* 12 (mayo de 2020), pág. 033303. DOI: 10.1063/1.5126055.

[29] Rahul Vadrabade. *Debugging OpenFOAM with Visual Studio Code*. https://www.ionos.com/digitalguide/server/configuration/dual-boot-windows-10-and-ubuntu/. Accedido 01-02-2023.

[30] Pau Varela Martinez. "On the Analysis and Design of Series Hybrid Distributed Electric Propulsion with Boundary Layer Ingestion of Remotely Piloted Aircraft". Tesis doct. Universitat Politecnica de Valencia, 2023. DOI: 10.4995/Thesis/10251/192805.

[31] Larry Viterna y David Janetzke. "Theoretical and experimental power from large horizontal-axis wind turbines". En: *NASA Technical Memorandum* (oct. de 1982).