Facultad de Ingeniería

Memoria del proyecto para optar al Título de
Ingeniero Civil Oceánico

Calibración y validación de OpenFOAM para la modelación de un canal de ondas en 2D

Carlos Contreras Ruiz

Octubre de 2020

Calibración y validación de OpenFOAM para la modelación de un canal de ondas en 2D

Carlos Contreras Ruiz

COMISIÓN REVISORA	**NOTA**	**FIRMA**
Patricio Winckler Profesor guía	_____	_____
Matías Quezada Revisor	_____	_____
David Poblete Revisor	_____	_____

Declaración

Este trabajo, o alguna de sus partes, no ha sido presentado anteriormente en la Universidad de Valparaíso, institución universitaria chilena o extranjera u organismo de carácter estatal, para evaluación, comercialización u otros propósitos. Salvo las referencias citadas en el texto, confirmo que el contenido intelectual de este Proyecto de Título es resultado exclusivamente de mis esfuerzos personales.

La Universidad de Valparaíso reconoce expresamente la propiedad intelectual del autor sobre esta Memoria de Titulación. Sin embargo, en caso de ser sometida a evaluación para los propósitos de obtención del Título Profesional de Ingeniero Civil Oceánico, el autor renuncia a los derechos legales sobre la misma y los cede a la Universidad de Valparaíso, la que estará facultada para utilizarla con fines exclusivamente académicos.

CARLOS CONTRERAS RUIZ
Alumno Memorista

PATRICIO WINCKLER GREZ
Profesor Guía

A mis abuelos

Agradecimientos

Este documento es el resultado de muchas horas de investigación y esfuerzo por parte del autor. Pero nada de esto habría sido posible sin mi familia, que desde niño me ha dado apoyo, motivación y cariño para que pueda salir a descubrir el mundo. Agradezco especialmente a mis abuelos Luz y José por entregarme los valores y formarme como una persona que se esfuerza por hacer siempre lo correcto. Agradezco a Dios por darme la curiosidad y motivación por aprender. Agradezco a mis tíos y primos porque me han acompañado en los altos y bajos a lo largo de la vida.

Vanessa, te agradezco por escucharme con interés tantas horas hablar de ecuaciones, modelos numéricos y herramientas que no pertenecen a tu área de estudio. Eres uno de los tesoros más preciados que me ha presentado la vida. Te amo.

Agradezco a mi mamá por mostrarme que la motivación y el esfuerzo son la herramienta fundamental para traer los sueños a la realidad. Agradezco a mi papá por enseñarme lo difícil que es derrotar a una persona que no se rinde.

A lo largo de mis años de estudio he conocido a muchos profesores que me han inspirado y formado. Álvaro Valdivia ha sido uno de ellos, una persona brillante, que me mantuvo pegado a libros, guías y cuadernos mientras aprendía cálculo y ecuaciones diferenciales, ha sido un gusto conocerlo. También he tenido el placer de conocer a Patricio Winckler, una gran persona y gran profesional. Agradezco a Patricio por haberme acompañado en el aprendizaje de OpenFOAM, sin su ayuda esta memoria no habría sido posible.

Agradezco la dedicación, guía y apoyo de los profesores Matías Quezada y David Poblete, quienes participaron en la revisión de este documento. Contribuyeron en gran parte a la mejora en su calidad.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Motivación

En la actualidad los puertos tienen un rol estratégico en el desarrollo de un país, ya que contribuyen al intercambio de productos, ingresos a las arcas fiscales y generación de empleos. Aunque su principal objetivo es actuar como punto de encuentro entre el medio marítimo y el terrestre, en estos se desarrollan múltiples servicios conformados por distintos agentes y organismos.

El transporte de tipo marítimo es el que maneja la mayor cantidad de mercancías, lo que favorece a las economías de escala (Rúa, 2006). Es por esto que la tecnología se enfoca en la construcción de embarcaciones cada vez más grandes, veloces y que además proporcionan una mayor seguridad a la carga. Siguiendo esta misma línea, se prevé que al favorecer la globalización económica mediante la estandarización de las normas comerciales para los países, se estimulará el intercambio de productos por lote, reforzando así el tipo de transporte que sea capaz de cumplir con el traslado de dichos productos de manera más eficiente (González, 2000). Para optimizar el intercambio de mercancías entre naciones, en las últimas dos décadas se ha requerido de la construcción de muelles más grandes y con mayor calado debido al aumento del tamaño de las naves. Esto hace muy costosa la construcción de rompeolas, forzando a su vez el diseño de muelles expuestos a condiciones adversas de oleaje, de esta manera se llevan a cabo construcciones que van más allá de los límites del conocimiento actual (Tirindelli et al., 2002).

Las operaciones de dragado se pueden definir como la succión, transporte y descarga de sedimento en el agua mediante dragas, generando un mayor calado para la maniobra segura de buques. Si se considera la opción de hacer dragados en el muelle, de manera que se pueda mantener un calado para grandes buques, sin exponer la estructura a condiciones desfavorables surgen otros problemas; como el impacto ambiental en la zona donde se deposita el sedimento, enterrando la flora y fauna marina, además, las operaciones de dragado pueden alterar las condiciones físicas, químicas y biológicas del ecosistema en las zonas de dragado y descarga; también se generan impactos físicos y/o químicos en la calidad del agua. Se deben considerar también los costos de las operaciones de dragado y descarga debido a las regulaciones ambientales de cada país, derivados de los tratamientos que requieren los contaminantes que pueda tener el sedimento (Landaeta, 1995). Por otra parte, si existe una fuente de aporte de sedimento cercana, las operaciones de dragado deben realizarse con regularidad a lo largo de la vida útil del muelle.

El diseño de obras marítimas se sustenta principalmente en el modelado físico y formulaciones semiempíricas, que entregan como resultado los criterios de diseño. La principal ventaja de las formulaciones semiempíricas es su bajo costo, simplicidad y corto tiempo en la obtención de los resultados, sin embargo, su rango de aplicabilidad es acotado solo a los casos convencionales, dejando de lado incluso efectos locales importantes que afectan significativamente los resultados. Por otra parte, el modelado físico es una representación más exacta de los procesos implicados en la propagación del oleaje y su interacción con la estructura. Sus principales inconvenientes están relacionados a los efectos de escala, costos, limitación en la cantidad de sensores disponibles y la alteración del flujo debido a los mismos sensores (Higuera, 2015).

Desde hace unos pocos años se han desarrollado modelos numéricos para estudiar el comportamiento del oleaje. En el caso de los modelos basados en las ecuaciones promediadas de Navier-Stokes (RANS), se alcanzan resultados bastante precisos en la propagación de oleaje no lineal en aguas someras; además, modelan la transformación del oleaje antes, durante y después de la rotura (Pedrozo & Torres, 2011). Las principales desventajas del modelado numérico están asociadas a que las simulaciones bidimensionales no representan todos los procesos que experimenta el oleaje, mientras que las simulaciones en tres dimensiones requieren altos costos computacionales. Además, es necesario calibrar y validar el modelo para asegurar que los datos obtenidos representen correctamente el comportamiento del oleaje (Higuera, 2015).

Dentro de los modelos que utilizan las RANS, destaca OpenFOAM (Jasak et al., 2004) por su licencia de código abierto; lo que permite el uso ilimitado de núcleos y rutinas según los requerimientos del investigador. Mientras que en el caso de software comerciales, el uso de rutinas y de núcleos depende del tipo de licencia adquirida (Davidson et al., 2015). Además, algunos de los numerosos estudios desarrollados en la última década con OpenFOAM, presentan la configuración de sus casos paso por paso, con el propósito de facilitar el uso de OpenFOAM a una comunidad creciente de usuarios. Ejemplo de esto es Herreras & Izarra (2013); de esta manera se intenta disminuir la poca documentación y soporte que presenta OpenFOAM a diferencia de sus contrapartes de pago.

El modelo OpenFOAM está limitado al estudio regiones relativamente pequeñas debido al significativo uso de recursos computacionales que demandan las RANS. Es por esto que en algunos casos, en etapas tempranas de un proyecto es necesario desarrollar modelos a escala, como canales de onda, para calibrar y validar el modelo.

Un incipiente grupo de investigadores, desde que OpenFOAM fue liberado para uso libre y gratuito en el año 2004, ha modelado canales de ondas para evaluar el comportamiento del oleaje; tales como Chenari (2014), analizó el comportamiento de oleaje regular interactuando con el fondo en un canal de ondas, además presentó las ventajas y limitaciones de la herramienta waves2Foam de OpenFOAM; Davidson et al. (2015) estudió el comportamiento de la energía producida por el oleaje en un canal de ondas tridimensional mediante la herramienta interFoam; Chen (2015) estudió las cargas de oleaje sobre estructuras costeras para la explotación de energía renovable, entre otros. Muchas investigaciones han sido la base para continuar desarrollando el modelo en términos de las necesidades de los usuarios, y por otra parte, estos mismos estudios han servido para validar los resultados del modelo.

Para el desarrollo de este trabajo se utilizó el modelo OpenFOAM con el solver olaFlow (Higuera, 2015), ya que cuenta con herramientas y subrutinas que permiten modelar canales de onda bidimensionales, para distintos regímenes de propagación de oleaje, y con un razonable uso de recursos computacionales.

1.2. Objetivos

1.2.1. Objetivo general

Calibrar y validar OpenFOAM para modelar un canal de ondas bidimensional.

1.2.2. Objetivos específicos

- Analizar el comportamiento de los principales parámetros y teorías que OpenFOAM-olaFlow utiliza para la modelación del fluido.
- Desarrollar una serie de scripts orientados a objetos en Python para automatizar el trabajo con los datos proporcionados por OpenFOAM-olaFlow.
- Analizar la sensibilidad de la malla para el modelo en un canal de ondas bidimensional, y luego comparar los resultados con la teoría de onda correspondiente a las condiciones de oleaje presentes en el canal de ondas.
- Generar oleaje progresivo y estacionario en un canal de ondas bidimensional, y compararlo con la teoría de ondas correspondiente.
- Modelar casos presentados en la literatura, con el objetivo de evaluar la capacidad del modelo para predecir los fenómenos presentes cerca de la zona de rompiente.
- Modelar casos y compararlos con artículos que estudien la relación entre el oleaje incidente sobre una estructura y las presiones desarrolladas sobre esta.

1.3. Alcances y limitaciones

- Costo computacional. Para la mayor parte de este trabajo se utilizó un notebook de solo 4 núcleos, cada uno con una capacidad de procesamiento de 2.30 GHz, lo que implicó en algunos casos una modelación de más de una semana para obtener resultados. Lo anterior repercutió directamente en el grado de precisión con que se corrieron los casos y la configuración de los casos elegidos.
- El solver desarrollado para modelación del oleaje en OpenFOAM considera la interacción de dos fluidos inmiscibles isotérmicos e incompresibles. El último punto implica que los resultados en ciertos casos dejan de tener sentido físico. Ejemplo de esto es cuando quedan bolsas de aire atrapadas bajo la losa de un muelle, pues al no poderse comprimir la fase aire, actúa como un sólido, dando lugar a presiones más altas en esa zona. Sin embargo, en estudios como Ferrer et al. (2016) o Higuera (2015) se ha demostrado que esto en general no es una limitante, ya que el solver puede capturar la dinámica esencial del comportamiento del oleaje en el dominio. Pero cuando el estudio se enfoca precisamente en analizar las presiones de esos puntos, se debe recurrir a un solver que incluya fluidos compresibles.
- Todos los canales modelados en este trabajo son bidimensionales debido al alto costo computacional que implica correr casos en tres dimensiones. Esto impide analizar el comportamiento tridimensional que experimenta el fluido en la realidad, sobre todo en las zonas de alto grado de turbulencia.

- Debido al costo computacional, para el desarrollo de este estudio no fue posible tomar en cuenta la porosidad de los materiales en la validación de los resultados, por lo que en ninguno de los artículos utilizados se considera la porosidad como una variable.

Capítulo 2

Fundamento teórico

2.1. Estado del arte

En la presente sección se analiza el desarrollo de los modelos numéricos destinados a estudiar la propagación y los procesos de transformación del oleaje en aguas someras, considerando los avances en modelación generados a lo largo de los últimos años. Luego, se hace un breve repaso de artículos donde se han utilizado canales de ondas para investigar el comportamiento del oleaje, esto es con el fin de presentar la importancia de estos modelos físicos para comprender las variables involucradas en las transformaciones que experimenta el oleaje.

2.1.1. Modelos numéricos para el estudio del oleaje

Un modelo numérico es un software que resuelve ecuaciones aproximadas utilizando métodos numéricos, con el objetivo de describir de forma simplificada un fenómeno que se desea estudiar. Tanto en zonas costeras como en aguas profundas se producen fenómenos complejos; como la interacción agua - aire, transporte de sedimentos, fenómenos no lineales en el oleaje cercano a la costa, interacción del oleaje con estructuras costeras, entre otros. Es por esto que es necesario analizar los procesos dominantes y descartar aquellos que no generan grandes cambios en el comportamiento del sistema (Winckler, 2018).

Para analizar el comportamiento del oleaje desde la zona de propagación hasta su posterior rotura existen distintos tipos de modelos, los que principalmente se diferencian en el tratamiento de las ecuaciones y los supuestos que hacen. Los modelos se pueden clasificar en dos principales categorías; modelos de flujo potencial y modelos basados en las ecuaciones de Navier-Stokes.

Modelos de flujo potencial

Los modelos de flujo potencial consideran que los esfuerzos viscosos y rotacionales en el flujo de la ola pueden ser despreciados. Como consecuencia de esta limitación, no tienen la capacidad de describir la rotura del oleaje.

Dentro de este tipo de modelos se pueden encontrar los que resuelven las ecuaciones no lineales de aguas someras (NLSWE); una forma simplificada de las ecuaciones de Navier-

Stokes; considerando un supuesto de presión hidrostática en la vertical, y como consecuencia de esto, asumiendo un perfil uniforme de velocidad en la horizontal. El principal uso que se le ha dado a este tipo de modelos es para simular la hidrodinámica sobre pendientes suaves, tsunamis, reflexión del oleaje en estructuras y bores; aun cuando los resultados pierden validez en aguas intermedias (Pedrozo & Torres, 2011).

Para modelar el comportamiento del oleaje en aguas intermedias se pueden utilizar modelos que resuelven las ecuaciones de Boussinesq. Estas ecuaciones consideran una variación menor del perfil de velocidades en la vertical, que permite describir correctamente la transformación del oleaje desde aguas intermedias hasta antes de la rotura (Pedrozo & Torres, 2011). Las ecuaciones de Boussinesq, al igual las NLSWE, tienen supuestos que disminuyen considerablemente el tiempo de cálculo por lo que enfrentan convenientemente el modelado de grandes extensiones de dominio (Higuera, 2015).

Modelos basados en las ecuaciones de Navier-Stokes

Estos modelos tienen la capacidad de describir los procesos involucrados en la transformación del oleaje que experimenta en zona de rompiente, además, se pueden desglosar cada una de las variables involucradas en la rotura del oleaje, como velocidades, turbulencia, gradientes de presión, entre otras (Pedrozo & Torres, 2011); por lo que representan una eficiente herramienta para estudiar el comportamiento del oleaje en rompiente interactuando con una estructura. Estos se pueden clasificar en eulerianos, donde el fluido se considera continuo; y lagrangianos, donde el fluido se considera compuesto de un conjunto de partículas.

Los modelos eulerianos utilizan las ecuaciones de Navier-Stokes promediadas por Reynolds (RANS), que tienen la capacidad de reproducir correctamente los perfiles verticales de presión y velocidad media, ya que no cuentan con supuestos iniciales que simplifiquen las ecuaciones. Las RANS se pueden promediar en el volumen (VARANS) para caracterizar el comportamiento de fluidos a través de medios porosos, herramienta que es útil para modelar el comportamiento del oleaje al interactuar con estructuras (Higuera, 2015).

Los modelos lagrangianos, también llamados Smooth Particle Hydrodynamics (SPH), tienen similar rendimiento que los modelos RANS, sin embargo aún se encuentran en etapas tempranas de desarrollo, dando como resultado restricciones de tiempo de cómputo y problemas en los contornos del dominio (Pedrozo & Torres, 2011).

2.1.2. OpenFOAM

Open Source Field Operation and Manipulation (OpenFOAM) es un software orientado a la resolución de una gran cantidad de problemas relacionados al flujo de fluidos. Fue desarrollado en el año 1989 por Henry Weller en el Imperial College, Londres. Inicialmente se llamó solo FOAM, pero luego en el año 2004 fue presentado al mundo con el nombre de OpenFOAM, bajo licencia pública general (GPL), la que garantiza el uso continuo y gratuito dentro de sus términos, dando libertad a los usuarios para modificar y redistribuir el software libremente.

Para la resolución de problemas, OpenFOAM aplica una discretización de volúmenes finitos, es decir, divide el dominio espacial en un conjunto de celdas. El tiempo se divide en

un número finito de pasos o intervalos de tiempo, generando un sistema de ecuaciones en términos de cantidades discretas para resolver el problema (Rusche, 2002).

Este modelo numérico se compone de un conjunto de librerías en C++ orientadas a objetos, lo que facilita la ampliación o modificación del código. Es así que los usuarios pueden agregarle extensiones de manera relativamente fácil, tales como condiciones de contorno o incluso aplicaciones con el conocimiento apropiado de programación y fluidodinámica.

OpenFOAM presenta dos tipos de aplicaciones; por un lado, se encuentran los *solvers*, en que cada uno resuelve un problema específico de dinámica de fluidos, y por otro lado, se encuentran los *utilities*, que son rutinas para la manipulación de los datos que van desde la generación del mallado, como es el caso de *blockMesh* hasta el postproceso con *postProcess*.

2.1.3. olaFlow

El solver olaFlow es una herramienta de OpenFOAM que tuvo sus inicios en base a la modificación del solver interFoam (Higuera et al. (2013a) y Higuera et al. (2013b)); el cual es una aplicación de OpenFOAM que en esa época no contaba con condiciones de contorno de generación ni absorción de oleaje, pero que permite estudiar fluidos bifásicos, considerando la fase agua y una fase gaseosa (aire). Esto lo convirtió en una herramienta ideal para modelar el comportamiento del oleaje cerca de la costa. Como parte de las modificaciones que se le hizo a interFoam, se implementó la absorción activa; componente que elimina la necesidad de generar playas o zonas disipativas, de forma que no hay aumento del dominio y su respectivo costo computacional. Además, se le agregaron funcionalidades para la generación de oleaje en distintos regímenes, por lo que el usuario puede elegir entre teorías de ondas como Stokes, Cnoidal o Stream-Function. Las anteriores subrutinas, entre otras funciones, dieron lugar a la creación de olaFlow.

En un principio, olaFlow no tenía la capacidad de modelar el paso del fluido a través de medios porosos, pero luego en el año 2015 esta mejora fue implementada y presentada en la tesis doctoral de Pablo Higuera bajo la licencia GPL (General Public License). Es así que este solver tiene la capacidad de modelar las interacciones entre el oleaje y las estructuras permeables e impermeables, por lo que actualmente es uno de los modelos más avanzados en su tipo (Higuera, 2015).

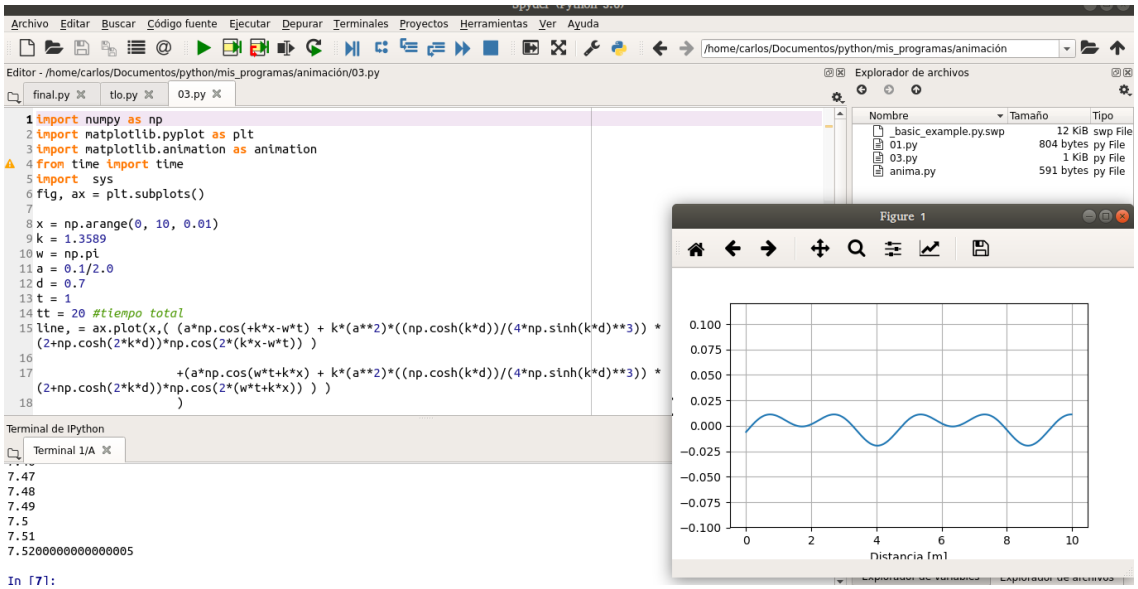
2.1.4. Python

Python es un lenguaje de programación multiparadigma que da la posibilidad de programar de forma imperativa, funcional u orientado a objetos (Challenger-Pérez et al., 2014). Fue creado en el año 1991 por el programador holandés Guido Van Rossum, y junto a la ayuda de una comunidad de desarrolladores Python se ha posicionado como uno de los lenguajes más prominentes de la actualidad. Tanto es así que en el año 2018 fue catalogado como el lenguaje del año por el índice de TIOBE (Programming Community index). En la actualidad Python es uno de los tres lenguajes más utilizados en el desarrollo de software junto con C y Java.

Con Python se pueden generar proyectos que luego pueden ser distribuidos libremente sin ser necesario publicar el código fuente, por lo tanto, Python puede utilizarse tanto para el desarrollo de software libre como privado.

Python en ciencia compite a la par con Matlab, Mathematica y otros software similares, ya que tiene una combinación de bibliotecas comparable a estos programas, y además tiene la ventaja de ser gratuito. Por otra parte, Python cuenta con entorno de desarrollo diseñado para científicos llamado "Spyder" con características parecidas a Matlab, lo que le permite al investigador migrar más fácilmente a Python (Figura 2.1). Además, la sintaxis de Python es muy similar al pseudocódigo debido a que se desarrolló en una época en la que no había grandes limitaciones de memoria o velocidad de ejecución, por lo que es fácil de aprender.

Figura 2.1: entorno de desarrollo Spyder para Python.



Fuente: elaboración propia.

En el presente trabajo la mayor parte de los scripts desarrollados son orientados a objetos, ya que los códigos generados con este paradigma tienen grandes ventajas para proyectos medianos o grandes. Una es la reutilización de clases en proyectos diferentes, lo que es posible debido a que hay rutinas que se repiten en distintos códigos, como por ejemplo, abrir un archivo, leer los datos y guardarlos en una lista. Otra ventaja de la programación orientada a objetos (POO) es la fácil lectura del código, ya que dentro de cada objeto se guardan detalles del código que solo se muestran en la clase (los planos de fabricación de cada objeto), por lo tanto es menos engorrosa la lectura del script. La POO se puede ver como ir construyendo una estructura compleja bloque a bloque; cada uno de estos corresponde a un objeto que resuelve una pequeña parte de un problema más grande, de esta manera es más fácil depurar el código.

Para el desarrollo de pequeños proyectos es aceptable utilizar el paradigma de programación funcional. Este tipo de programación se desarrolla en base a funciones, por lo que es fácil la reutilización del código y el desarrollo del script, ya que una función específica retornará lo mismo a lo largo de toda la ejecución del programa. Los scripts más pequeños desarrollados en este trabajo se crearon con este paradigma de programación. A modo de ilustración, se presenta la implementación de la Teoría Lineal de Ondas tanto en POO como en programación funcional:

Para la POO la clase TLO tiene atributos de gravedad (g), profundidad (d), periodo (T) y longitud de onda (L), además tiene el método *londa*(), por lo que todos los objetos creados a partir de la clase TLO estarán compuestos de estos elementos:

```
import math as mt # importar librería math

class TLO:

    def __init__(self,d,T):
        self.g = 9.81
        self.t = T
        self.d = d

    def londa(self, sw = 0): # sw swish
        g = self.g
        T = self.t
        d = self.d
        L = (g*(T**2))/(2*mt.pi)

        for x in range(1000):
            L =((g*T**2)/(2*mt.pi))*mt.tanh((2*mt.pi*d)/(L))


        if sw ==0:
            pass

        else:
            print('Longitud de onda', round(L,3),'[m]')

            if d/L>0.5:
                print('Aguas profundas')

            elif 1.0/20.0<d/L and  d/L<0.5:
                print('Aguas intermedias')

            elif d/L<1.0/20.0:
                print('Aguas someras')

        return L


objeto01 = TLO(0.635, 2.0).londa() # longitud de onda
objeto02 = TLO(0.8, 2.0).londa() # otra longitud de onda
```

Cuando la TLO se escribe con programación funcional queda como:

```
import math as mt # importar librería math

def londa(d,T, sw=0):
    g = 9.81
    L = (g*(T**2))/(2*mt.pi)
```

```
    for x in range(1000):
        L =((g*T**2)/(2*mt.pi))*mt.tanh((2*mt.pi*d)/(L))

    if sw ==0:
        pass

    else:
        if d/L>0.5:
            print('Aguas profundas')

        elif 1.0/20.0<d/L and  d/L<0.5:
            print('Aguas intermedias')

        elif d/L<1.0/20.0:
            print('Aguas someras')

    return L

L1 = londa(0.635, 2.0) #longitud de onda 1
L2 = londa(0.8,2.0)# otra longitud de onda
```

Se hace evidente que el primer script es más legible y ordenado, ya que las variables están encapsuladas dentro del objeto que las utiliza.

2.1.5. Canales de ondas

Los canales de ondas son estructuras que contienen un líquido en contacto con la atmósfera, donde las fuerzas que actúan sobre el líquido son la gravedad, tensión superficial, viscosidad y presión. Además, se deben considerar las fuerzas que actúan si el canal tiene sedimentos (Castellanos et al., 2017).

Un canal de ondas se compone principalmente por un borde libre, el fondo, un borde de generación de ondas y su respectiva pared opuesta, paneles laterales y opcionalmente rampas y/o estructuras para estudiar el comportamiento del fluido con respecto a dichos componentes. En la dinámica de fluidos computacional se pueden utilizar canales numéricos de ondas para modelar la propagación y transformación del oleaje al acercarse a la costa (Didier & Neves, 2012). En los últimos años se han desarrollado varios estudios mediante OpenFOAM en que se usan canales numéricos, por lo que a continuación se presentan algunas contribuciones relevantes:

Lambert (2012) calibró openFOAM para implementar un canal de ondas que generaba oleaje regular mediante la teoría Stokes II. De sus resultados obtuvo que OpenFOAM en esa época no podía modelar oleaje regular con una inclinación (H/L) superior a 0.05, donde H es la altura de ola y L es la longitud de onda.

Higuera et al. (2013a) agregó condiciones de borde al solver interFoam para la generación de oleaje, dando la posibilidad de elegir al investigador las principales teorías de ondas actuales.

Higuera et al. (2013b) implementó la absorción activa al solver interFoam. Esta condición de borde elimina, en ciertos casos, la necesidad de generar playas disipativas para evitar la

reflexión en el borde de salida del agua. La absorción activa, junto a otras implementaciones crearon el actual solver olaFlow a partir de interFoam.

Higuera et al. (2014a) y Higuera et al. (2014b) modificaron las ecuaciones de los modelos de turbulencia $k - \epsilon$ y $k - \omega\, SST$ para que el solver interFoam pueda modelar el comportamiento del fluido al interactuar con medios porosos.

Henry et al. (2013) estudió el comportamiento del oleaje que generaba slamming sobre un convertidor de sobretensiones de ondas, también llamado OWSC por sus siglas en inglés (oscillating wave surge converter). Para esto utilizó wsiFoam, una extensión de interFoam (dos fluidos incompresibles) y compressibleInterFoam (dos fluidos compresibles). Demostró que si bien el solver interFoam puede modelar adecuadamente el comportamiento del oleaje en todo el dominio, es necesario utilizar un solver compresible para modelar los efectos debido a las burbujas y bolsas de aire.

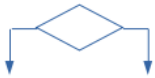
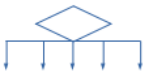
Estos, junto a otros estudios han contribuido, ya sea, en la implementación de nuevas características al solver olaFlow o para conocer las ventajas y/o limitaciones que tiene OpenFOAM para modelar el oleaje, tanto en zonas costeras como en aguas profundas.

2.2. Diagramas de flujo

Para visualizar de forma más fácil los algoritmos programados en los scripts de este trabajo se recurre al uso de diagramas de flujo, los que son acompañados con su respectiva tabla, la cual presenta las variables, constantes y funciones utilizadas en cada script. Algunos de los diagramas de flujo presentan condiciones del tipo (V/F), las cuales representan una condición verdadera o falsa respectivamente.

Los componentes de un diagrama de flujo se pueden dividir en constantes, variables, listas de valores y funciones. Las listas de valores se aplican en el código para agrupar tipos de variables, caracteres o constantes relacionadas entre sí; de esta manera se reduce enormemente la dificultad en el procesamiento de datos. Por otra parte, las funciones son los objetos en sí mismos que están compuestos por distintos atributos y métodos pueden retornar variables, listas de valores o no retornar valores. En este último caso se pueden presentar como ejemplo los objetos que generan una gráfica en base a dos listas de valores o los objetos que crean y editan archivos de texto dentro del directorio del caso de estudio.

Figura 2.2: significado de los símbolos para la construcción de diagramas de flujo.

Símbolo	Significado
	Inicio / Término
	Entrada de datos
	Proceso
	Decisión
	Decisión múltiple
	Imprimir resultados
	Flujo de datos
	Conectores

Símbolo	Significado
+	Suma
-	Resta
*	Multiplicación
/	División
^	Exponenciación
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
<>	Diferente que
=	Igual que

Fuente: adaptado de Pinales & Velázquez (2017).

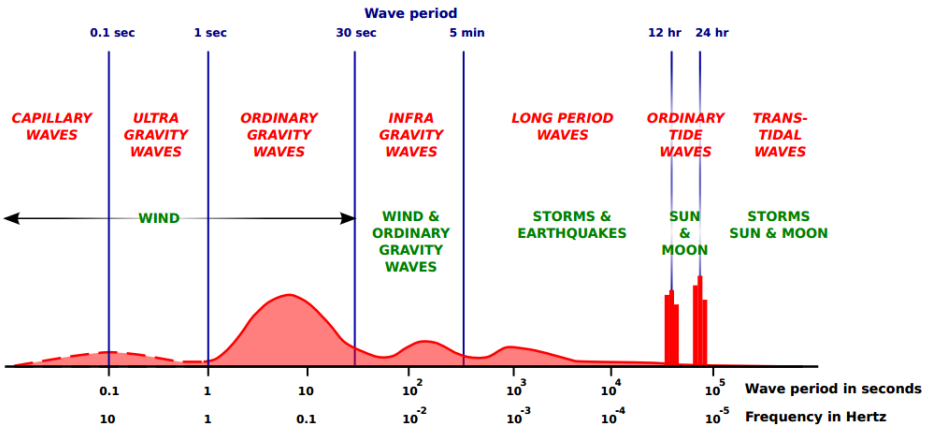
2.3. El oleaje

Las ondas son perturbaciones del fluido producto de fuerzas que actúan sobre él. En la naturaleza este fenómeno puede ser visto en masas de agua de variado tamaño, desde el océano e incluso hasta en un charco. Estas perturbaciones pueden ser generadas por fuerzas de distintas magnitudes y procedencias; tales como el viento, la atracción gravitacional de la luna y el sol sobre la tierra, la fuerza de Coriolis o impacto de meteoritos.

Una forma de clasificar los tipos de fuerzas que actúan sobre el oleaje es mediante la influencia que tienen sobre el cuerpo de agua. Las principales fuerzas generadoras de oleaje son: la atracción gravitacional de la luna, el sol y los planetas; terremotos submarinos, las tormentas y vientos. Por otra parte, las principales fuerzas restauradoras del oleaje son la fuerza de Coriolis, la fuerza de gravedad y la tensión superficial (Werlinger et al., 2004).

Luego de generarse una perturbación, la tensión superficial y la fuerza de gravedad permiten que la onda generada se propague a través del fluido (Dean, 1984). En la Figura 2.3 se presenta la clasificación de las ondas en base al periodo.

Figura 2.3: clasificación de los distintos tipos de ondas presentes en el océano con respecto al periodo.



Fuente: Ardhuin (2018).

El oleaje presenta un periodo de entre 3 y 30 s, y su mecanismo de generación es el viento. Este tipo de ondas tiene una gran relevancia debido a que casi todo el tiempo están presentes en el océano. Además, intervienen en una gran cantidad de actividades antrópicas; como la pesca artesanal, transporte marítimo, diseño y construcción de obras marítimas, extracción petróleo o gas. Es por esto que se presta especial atención a su comportamiento desde su origen, propagación, y luego hasta su posterior rotura o disipación, entre otras.

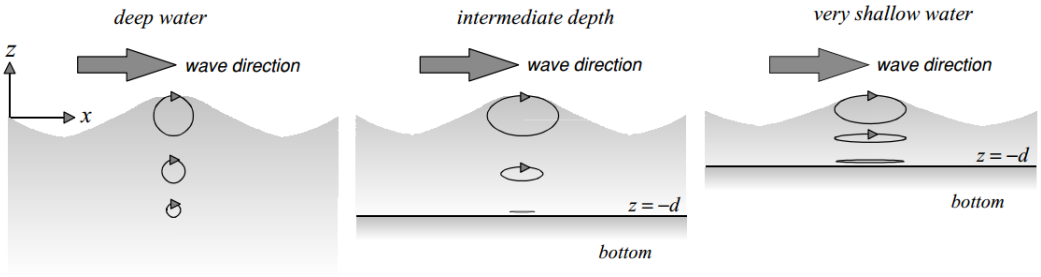
El oleaje experimenta una variación de su comportamiento a medida que empieza a interactuar con el fondo. Por esta razón se establece una clasificación en base a la profundidad relativa (d/L), donde d corresponde a la profundidad y L es la longitud de onda:

- $d/L > 1/2$ aguas profundas
- $1/20 < d/L < 1/2$ aguas intermedias
- $d/L < 1/20$ aguas someras

En aguas profundas la onda no tiene interacción con el fondo por lo que el movimiento de sus partículas sigue una trayectoria circular, tal como se muestra en la Figura 2.4

(izquierda). Luego, este desplazamiento comienza a ser elíptico a medida que la onda comienza a interactuar con el fondo.

Figura 2.4: movimiento orbital de las partículas de la onda para distintas condiciones de profundidad relativa.



Fuente: Holthuijsen (2010).

Cuando el oleaje se propaga hacia aguas someras comienza a experimentar asomeramiento, fenómeno que se traduce en un aumento de la altura de onda y reducción de su longitud. Además, se hacen evidentes otros fenómenos como la refracción; producida por la disminución de la celeridad que experimentan las ondas que se propagan en una menor profundidad, o por una corriente, dando como resultado el giro del frente de onda hacia menores profundidades (Universidad de Cantabria, 2000). Otro fenómeno que se produce en la naturaleza es la difracción, la cual se caracteriza por la transferencia lateral de energía de manera perpendicular a la dirección de propagación de la onda. Generalmente se produce cuando el tren de ondas se encuentra con un obstáculo como un rompeolas o una isla. Finalmente, se debe considerar la reflexión, que se produce cuando la onda llega a una estructura artificial como un rompeolas o una playa con gran pendiente. Durane la interacción, parte de la energía de la onda se disipa y el resto se refleja; cuanto más permeable es la estructura o más suave es la pendiente de la playa, menor será la reflexión (US Army Corps Of Engineers, 2002).

En las proximidades de la costa el oleaje comienza a aumentar su altura hasta que se vuelve inestable y rompe. La rotura tiene un comportamiento altamente inestable, aleatorio e impredecible ya que se transforma en un flujo turbulento. Este proceso tiene una gran importancia, porque es ahí donde disipa la mayor cantidad de su energía en forma de calor mediante la turbulencia (Pedrozo & Torres, 2011). Además, la turbulencia está asociada al transporte de la mayoría de sedimentos costeros (Finn & Li, 2016).

Teorías de onda

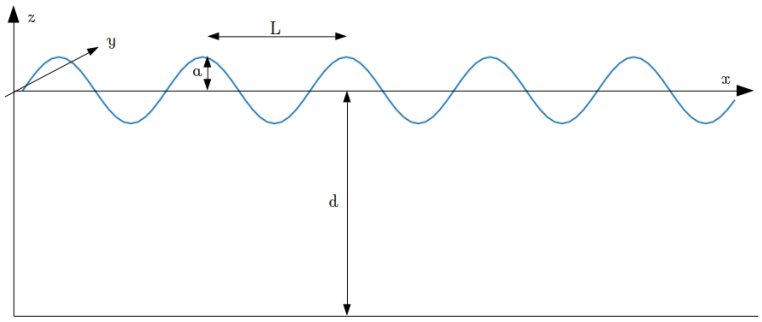
La variación del comportamiento del oleaje a medida que se propaga genera la necesidad de estudiar sus características con distintos tipos de teorías de onda, desde una teoría lineal de ondas, que representa el comportamiento del oleaje donde los efectos no lineales son despreciables, hasta la teoría Cnoidal, que representa bien el comportamiento del oleaje en aguas someras.

Teoría lineal de ondas

La TLO, teoría de Stokes de primer orden, o teoría de onda de Airy, representa el comportamiento del oleaje de oleaje de pequeña amplitud es decir, pequeña amplitud en comparación

con la longitud de onda y con la profundidad del agua (Airy, 1845). A continuación, en la Figura 2.5 se presenta una imagen descriptiva de la TLO.

Figura 2.5: representación del oleaje en aguas profundas mediante la TLO; d corresponde a la profundidad con respecto al nivel medio, a es la amplitud y L es la longitud de onda.



Fuente: elaboración propia.

Las ecuaciones de gobierno de la TLO están dadas por la ecuación de continuidad y la segunda ley de Newton, además se toman en cuenta una serie de supuestos presentados a continuación:

- El periodo de la onda es despreciable con respecto a la rotación terrestre, es decir, no se toma en cuenta el efecto Coriolis.
- El agua se considera un fluido incompresible, debido a que las fuerzas involucradas en la compresión del agua son despreciables.
- Se desprecian los efectos de la tensión superficial, es decir, las olas estudiadas por la TLO deben ser más grandes que unos pocos centímetros.
- La masa de agua debe ser continua (no se consideran burbujas de aire ni la rotura de las olas).
- Se considera la densidad del fluido constante. Por esta razón la TLO debe aplicarse con cautela cuando interactúan dos masas de agua con distinta densidad como en la desembocadura de un río (Holthuijsen, 2010).
- Las partículas de agua no interactúan con el fondo ni tampoco pueden salir a la superficie.
- La única fuerza externa es la gravedad: no se toma en cuenta la fuerza de generación, por lo tanto, no se considera el viento ni la presión.
- La profundidad a lo largo del sentido de propagación de la onda es constante.

Luego de aplicar los anteriores supuestos a la ecuación de continuidad y a la segunda ley de Newton se obtiene la ecuación 2.1. Para ver los detalles de la derivación de las ecuaciones se aconseja al lector recurrir a Universidad de Cantabria (2000). La función potencial queda como:

$$\Phi(x, y, z, t) = -\frac{ag}{\omega} \frac{\cosh(k(d+z))}{\cosh(kd)} \sin(kx - \omega t) \tag{2.1}$$

donde:

- a corresponde a la amplitud de onda.
- d es la profundidad del fluido con respecto al nivel medio.
- $k = \frac{2\pi}{L}$ es el número de onda, siendo L la longitud de onda.
- g corresponde a la gravedad.
- $\omega = \frac{2\pi}{T}$ es la frecuencia angular, con T siendo el periodo de la onda.
- t corresponde al tiempo.
- x es la distancia horizontal con dirección positiva hacia donde se propaga la onda.
- y es el eje perpendicular a la dirección de propagación de la onda.
- z es el eje vertical centrado en el promedio de las desnivelaciones de la onda.

La ecuación de dispersión, definida a partir de las condiciones de contorno de la superficie libre:

$$\omega^2 = gk \tanh(kd) \tag{2.2}$$

perminte relacionar el periodo y la longitud de onda, según:

$$L = \frac{gT^2}{2\pi} \tanh\left(\frac{2\pi d}{L}\right) \tag{2.3}$$

La ecuación 2.3 puede resolverse mediante métodos numéricos como Newton-Rapson. Por otra parte, la superficie libre se deriva a partir de la ecuación 2.1 mediante:

$$\eta(x,t) = \frac{1}{g}\left(\frac{\partial \Phi}{\partial t}\right) = a\cos(kx - \omega t) \tag{2.4}$$

Si la dirección de propagación es en sentido contrario, entonces la superficie libre se representa mediante:

$$\eta(x,t) = a\cos(kx + \omega t) \tag{2.5}$$

Cuando la onda llega a una pared, se refleja. Si la pared está perpendicular a la dirección de propagación de la onda, es lisa y no hay disipación de energía, se genera una onda estacionaria. La cual se forma cuando dos ondas progresivas con el mismo periodo y amplitud viajan en sentido opuesto (Universidad de Cantabria, 2000).

Asumiendo que el proceso es lineal, la superficie libre se puede calcular mediante la suma de las desnivelaciones de dos ondas progresivas con dirección inversa:

$$\eta(x,t) = \eta_1 + \eta_2 = \frac{H_i}{2}\cos(kx - \omega t) + \frac{H_r}{2}\cos(kx + \omega t) \tag{2.6}$$

La altura de ola incidente (H_i) y la altura de onda reflejada (H_r) para una reflexión perfecta son idénticas, entonces:

$$\eta(x,t) = \frac{H_i}{2}[\cos(kx - \omega t) + \cos(kx + \omega t)] \tag{2.7}$$

Luego, aplicando propiedades trigonométricas, se obtiene:

$$\eta(x,t) = H_i \cos(kx)\cos(wt) \tag{2.8}$$

Si se analiza el comportamiento de la onda generada por la ecuación 2.8 en el espacio y se deja el tiempo constante, las máximas alturas de ola se presentan para $t = \frac{cT}{2}$ con $c = 0, 1, 2, 3....n$, entonces la amplitud de onda en función de x está dada por:

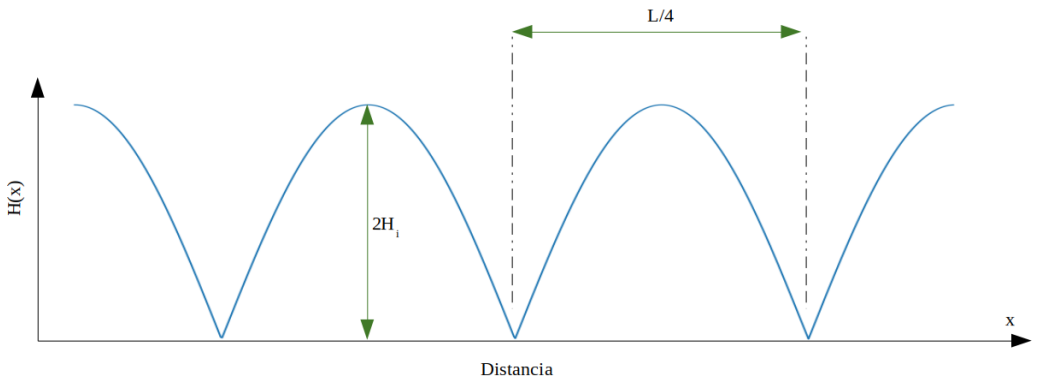
$$a(x) = H_i \cos(kx) \tag{2.9}$$

Finalmente, la altura de onda a lo largo del espacio varía mediante:

$$H(x) = |2H_i \cos(kx)| \tag{2.10}$$

Luego, la altura de onda en el espacio en una onda estacionaria tiene el comportamiento que se muestra en la Figura 2.6

Figura 2.6: curva de alturas de ola en el espacio para una onda estacionaria según la TLO. L corresponde a la longitud de onda y H_i es la altura de onda incidente.



Fuente: elaboración propia.

Teoría de Stokes de orden superior

Para el régimen de Stokes, la teoría lineal de ondas se extiende agregando términos no lineales, los que van aumentando de manera sucesiva a medida que aumenta el orden de corrección (Holthuijsen, 2010). Para el caso de Stokes de primer (Teoría Lineal de Ondas) orden se agrega el armónico $\epsilon = ak$, el cual representa el peralte de la onda:

$$\eta(x,t) = a\cos(kx - \omega t) = \epsilon\eta_1(x,t) \tag{2.11}$$

con:

$$\eta_1(x,t) = k^{-1}\cos(kx - \omega t) \tag{2.12}$$

Luego, para agregar la corrección de segundo orden se agrega un armónico de onda extra ($\epsilon^2\eta_2(x,t)$) elevado a la segunda potencia:

$$\eta(x,t) = \epsilon\eta_1(x,t) + \epsilon^2\eta_2(x,t) \tag{2.13}$$

Utilizando la solución de la TLO, la ecuación de desnivelación no lineal se presenta en 2.14. Para mayor información sobre la derivación de las ecuaciones se aconseja al lector referirse a Holthuijsen (2010).

$$\eta(x,t) = a\cos(kx-\omega t) + ka^2\frac{\cosh(kd)}{4\sinh(kd)^3}(2+\cosh(2kd))\cos[2(kx-\omega t)] \tag{2.14}$$

Repitiendo el mismo procedimiento la teoría de Stokes se puede representar para ordenes sepueriores:

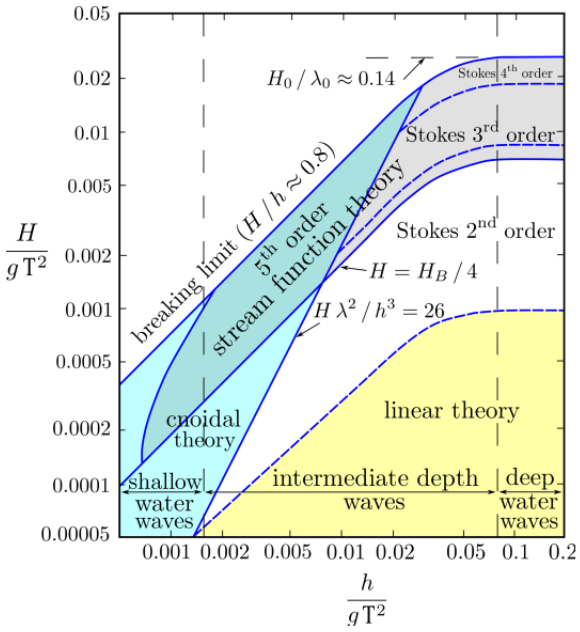
$$\eta(x,t) = \sum_{i=1}^{n}\epsilon^i\eta_i(x,t) \tag{2.15}$$

El rango de aplicabilidad para el régimen de Stokes puede obtenerse mediante el número de Ursell, el cual presenta el grado de no linealidad de la onda mediante la relación entre el peraltamiento y la profundidad relativa:

$$U_r = \frac{HL^2}{d^3} \tag{2.16}$$

donde H representa a la altura de onda, L es la longitud de onda y d es la profundidad del agua. El régimen de Stokes es adecuado cuando $1 \leq U_r < 10$. Luego, uniendo el rango de aplicabilidad de estas las teorías con las que aparecen en el Anexo A se genera el gráfico de la Figura 2.7.

Figura 2.7: rango de aplicabilidad de las teorías de onda.



Fuente: Le Méhauté (1976).

2.4. Enfoques para la resolución de las ecuaciones de Navier-Stokes

Los flujos turbulentos son irregulares, aleatorios y se caracterizan por un mayor transporte de masa, calor y cantidad de movimiento que los flujos laminares. Además, la turbulencia presenta rotacionalidad y tridimensionalidad. Si a esto se le suma que, si el flujo es completamente desarrollado, pueden generarse escalas de rotación del tamaño del dominio y por contraparte remolinos de escalas muy pequeñas, los que provocan la mayor disipación de energía. Se debe considerar que hay una transferencia de energía desde los remolinos más grandes a los de menor tamaño, luego las fuerzas viscosas actúan sobre los remolinos que alcanzan un tamaño mínimo adecuado (microescala de Kolmogorov), disipando la energía. Para mayor información se remite al lector a Kolmogorov (1941a), Kolmogorov (1941b) y Kolmogorov (1941c).

En la actualidad existen diferentes enfoques para modelar la turbulencia o bien resolverla de manera directa; la elección del enfoque depende del tamaño del dominio, la precisión y tiempos necesarios para llegar a la solución, y las capacidades computacionales, entre otros factores.

2.4.1. DNS

La simulación numérica directa o DNS (Direct Numerical Simulation) entrega soluciones de alta definición para flujos turbulentos, ya que la turbulencia se resuelve explícitamente. Mediante las DNS se obtienen resultados que no son afectados por aproximaciones y además se conoce el comportamiento del fluido en todo el dominio. Este dominio debe abarcar los remolinos más grandes, y a su vez, la malla debe ser lo suficientemente fina como para resolver la disipación presentada por los remolinos de escalas más pequeñas (Coleman &

Sandberg, 2010). Es entonces, un requisito fundamental desarrollar una malla extremadamente fina para tener resultados aceptables, lo que es incompatible para el diseño de obras marítimas con los recursos computacionales actuales (Higuera, 2015).

2.4.2. LES

El enfoque de modelado LES (Large Eddy Simulation) se utiliza para modelar el comportamiento de los remolinos más grandes presentes en la turbulencia, modelando los de pequeña escala. Como consecuencia de esto, la densidad de la malla computacional puede disminuir e incrementar los pasos de tiempo (time steps) del modelo. Sin embargo, el costo computacional aún es muy alto en comparación con el enfoque de Reynolds averaged Navier-Stokes (RANS) explicado más adelante (de Villiers, 2006).

2.4.3. RANS

Los modelos basados en las ecuaciones de Navier-Stokes promediadas por Reynolds (RANS) son utilizados para predecir los efectos de la turbulencia. Estos no resuelven pero si modelan los efectos de las fluctuaciones turbulentas de pequeña escala (Collado et al., 2007). Este enfoque requiere recursos computacionales notablemente menores que los mostrados anteriormente, sin embargo, puede reproducir los procesos que afectan al oleaje; asomeramiento, rotura, refracción, reflexión, entre otros (Higuera, 2015).

Los efectos provocados por la turbulencia en este tipo de modelos se desarrollan mediante los tensores de Reynolds, generando la necesidad de utilizar modelos empíricos para llegar a la solución. A estos se les llama modelos de cierre debido a que el uso de las RANS por si solas no constituye un sistema cerrado (Pedrozo & Torres, 2011). Si los modelos RANS se promedian en el volumen (VARANS) se puede modelar el comportamiento de un fluido interactuando con materiales porosos, dando la posibilidad de modelar el comportamiento de estructuras que podrían estar expuestas al oleaje, tales como diques, escolleras o arena.

La deducción de las ecuaciones VARANS depende de los supuestos que aplica el autor; Higuera (2015) se basa en el trabajo de Slattery (1967) y Whitaker (1967), quienes entregaron los fundamentos para el promediado en el volumen de las RANS. También tomó en cuenta el trabajo de del Jesus (2011) ya que describe las variaciones espaciales de la porosidad, sin embargo, hay discrepancias en el promediado del volumen con respecto a Slattery (1967) y Whitaker (1967), es por esto que Higuera (2015) rederivó las RANS.

2.4.4. DES

La técnica de simulación de remolino independiente DES (The Detached-Eddy Simulation) fue desarrollada por Spalart (1997). Es un híbrido entre los enfoques RANS (Reynolds Averaged Navier-Stokes) y LES. Su comportamiento se mantiene como RANS en las regiones donde la escala de turbulencia es menor, y donde es mayor se resuelve usando LES.

Capítulo 3

Materiales y métodos

3.1. Descripción general del modelo

En la presente sección se describen las ecuaciones que resuelve el modelo OpenFOAM con el solver acoplado olaFlow. Luego presentan los modelos de turbulencia empleados en los casos de estudio. Finalmente se escribe el método para definir la superficie libre y el tipo de discretización del dominio utilizado por el modelo.

3.1.1. Ecuaciones de gobierno de OpenFOAM

El modelo OpenFOAM utiliza las ecuaciones de Navier-Stokes promediadas por Reynolds (RANS) y la ecuación de conservación de la masa. Con el promediado de las ecuaciones de Navier-Stokes se pierden los detalles de las fluctuaciones turbulentas, las que luego son reemplazadas por los tensores de Reynolds. Esta modificación de las ecuaciones es necesaria debido a las limitaciones computacionales con que se cuenta en la actualidad. Las RANS promediadas en el volumen (VARANS) se pueden describir mediante (Higuera, 2015):

$$\frac{\partial \rho \langle u_i \rangle}{\partial t} + \frac{\partial}{\partial x_j}\left[\frac{1}{\phi}\rho \langle u_i \rangle \langle u_j \rangle\right] = -\phi \frac{\partial \langle p \rangle^f}{\partial x_i} + \phi \rho g_i + \frac{\partial}{\partial x_j}\left[\mu \frac{\partial \langle u_i \rangle}{\partial x_j}\right] - \frac{\partial}{\partial x_j}\left[\rho \left\langle \overline{u_i' u_j'} \right\rangle\right] + [CT] \quad (3.1)$$

donde los términos de cierre están dados por:

$$[CT] = -\alpha \frac{(1-\phi_{ST})^3}{\phi_{ST}^2} \frac{\mu}{D_{50,ST}^2} \langle u_i \rangle - \beta \left(1 + \frac{7.5}{KC_{ST}}\right) \frac{1-\phi_{ST}}{\phi_{ST}^2} \frac{\rho}{D_{50,ST}} |\langle u \rangle| \langle u_i \rangle - C \frac{\partial \langle u_i \rangle}{\partial t} - \mathbf{F}_D \quad (3.2)$$

donde:

- ρ es la densidad del fluido.
- g_i corresponde a la gravedad.
- p es la presión hidrostática que actúa sobre el fluido.
- μ es la viscosidad dinámica del fluido.
- ϕ es la porosidad total, definida como la fracción de volumen de fluido que está contenida en un volumen de control.
- ST es un subíndice indica que el término es una propiedad del material con porosidad constante tanto en el tiempo como en el espacio.
- ϕ_{ST} es la porosidad estática.
- x_i es la unidad de longitud en la componente i.
- t corresponde a la variable tiempo.
- V es el volumen total del fluido.
- $\langle\rangle$ es el operador de promediado en el volumen.
- $\langle\rangle^f$ es el volumen intrínseco promedio, operador que no considera las variaciones del volumen ocupado por el fluido. Esta información sí la considera el operador de promediado en el volumen ($\langle\rangle = \phi \langle\rangle^f$).
- u_i es la velocidad del fluido en la componente i.
- u_i' es la fluctuación turbulenta de Reynolds para u_i.
- D_{50} es el diámetro nominal del material poroso.
- α es el término de fricción lineal.
- β es el término de fricción no lineal.
- KC es el número de Keulegan-Carpenter $\left(\frac{T_0}{D_{50}}\frac{u_M}{\phi}\right)$. T_0 es el periodo de la oscilación y u_M es la máxima velocidad oscilatoria.
- C corresponde a un factor que por defecto es 0.34.
- $\mathbf{F}_D$ es el campo vectorial de la fuerza de arrastre que describe el efecto de las partículas en movimiento en el fluido.

La ecuación de conservación de la masa, promediada en el volumen, para una porosidad constante en el tiempo y fluido incompresible está dada por (Higuera, 2015):

$$\frac{\partial \delta}{\partial t} + \frac{1}{\rho}\frac{\partial \delta \langle u_i \rangle}{\partial x_i} = 0 \tag{3.3}$$

donde:

- $\langle u_i \rangle$ es la velocidad promediada en el volumen.
- δ es el volumen de agua por el volumen total de fluido (agua más aire) en el dominio, es decir, es la fase agua.

Cuando se agregan materiales porosos al dominio, la ecuación de conservación de la masa es reemplazada por:

$$\frac{\partial \phi \delta}{\partial t} + \frac{\partial \delta \left\langle u_i \right\rangle}{\partial x_i} = 0 \tag{3.4}$$

3.1.2. Modelos para la resolución de la turbulencia

El solucionador olaFlow utiliza las VARANS para modelar el oleaje, por lo tanto, es necesario elegir un modelo de cierre para establecer el comportamiento de la turbulencia.

Modelo $k - \epsilon$

El modelo $k - \epsilon$ es uno de los más utilizados en CFD debido a su robustez, bajo uso de recursos computacionales y razonable precisión. Además puede ser utilizado en un amplio rango de flujos turbulentos. Sin embargo, su rendimiento disminuye cuando se presentan casos de grandes gradientes de presión. El modelo $k-\epsilon$ promediado en el volumen se define como:

$$\frac{\partial \left\langle k \right\rangle}{\partial t} + \nabla \cdot \left(\frac{\left\langle k \right\rangle}{\phi} \left\langle \mathbf{u} \right\rangle \right) - \nabla \cdot \left[\left(\nu + \frac{1}{\phi} \frac{\left\langle \nu_t \right\rangle}{\sigma_k} \right) \nabla \left\langle k \right\rangle \right] \left\langle \epsilon \right\rangle = \frac{2}{\phi^2} \left\langle \nu_t \right\rangle \left| \frac{\nabla \left\langle \mathbf{u} \right\rangle + \left(\nabla \left\langle \mathbf{u} \right\rangle \right)^T}{2} \right|^2 + [CT]_{k_{ST}} + [CT]_{k_{DY}} \tag{3.5}$$

$$\frac{\partial \left\langle \epsilon \right\rangle}{\partial t} + \nabla \cdot \left(\frac{\left\langle \epsilon \right\rangle \left\langle \mathbf{u} \right\rangle}{\phi} \right) - \nabla \cdot \left[\left(\nu + \frac{1}{\phi} \frac{\left\langle \nu_t \right\rangle}{\sigma_k} \right) \nabla \left\langle \epsilon \right\rangle \right] + C_{\epsilon_2} \frac{\left\langle \epsilon \right\rangle^2}{\left\langle k \right\rangle} = \frac{2 C_{\epsilon_1}}{\phi^2} \left\langle \nu_t \right\rangle \frac{\left\langle \epsilon \right\rangle}{\left\langle k \right\rangle} \left| \frac{\nabla \left\langle \mathbf{u} \right\rangle + \left(\nabla \left\langle \mathbf{u} \right\rangle \right)^t}{2} \right|^2 + [CT]_{\epsilon_{ST}} + [CT]_{\epsilon_{DY}} \tag{3.6}$$

con la viscosidad cinemática turbulenta promediada en el volumen dada por:

$$\left\langle \nu_t \right\rangle = C_\mu \frac{\left\langle k \right\rangle^2}{\left\langle \epsilon \right\rangle} \tag{3.7}$$

El modelo de cierre propuesto por Nakayama & Kuwahara (1999) está definido mediante:

$$[CT]_{k_{ST}} = 39 \frac{(1-\phi)^{5/2}}{\phi} \frac{|u_i|^3}{D_{50}} \tag{3.8}$$

y

$$[CT]_{\epsilon ST} = \frac{C_{\epsilon 2}\left(39\frac{(1-\phi)^{5/2}}{\phi}\frac{|u_i|^3}{D_{50}}\right)^2}{\left(3.7\frac{1-\phi}{\sqrt{\phi}}\left|\langle u_i\rangle\right|^2\right)} \tag{3.9}$$

Los términos presentados en las ecuaciones 3.5 a la 3.9 están dados por:

- $\langle\epsilon\rangle$ corresponde al coeficiente de disipación, o tasa de disipación de la energía turbulenta promediada en el volumen.
- $C_{\epsilon_2} = 1.92$, coeficiente de cierre.
- $\langle k\rangle$ corresponde a la energía cinética turbulenta promediada en el volumen.
- $C_\mu = 0.09$, coeficiente de cierre.
- ν es la viscosidad cinemática del fluido.
- $\sigma_k = 1.0$, constante del modelo.
- $\mathbf{u}$ es el vector de velocidad del fluido.
- u_i es la velocidad del fluido en la componente i.
- ϕ es la porosidad total.
- D_{50} es el diámetro nominal del material poroso.
- $\langle\rangle$ es el operador de promediado en el volumen.
- T corresponde a la traspuesta de una matriz.

Cuando Higuera (2015) estaba desarrollando el solver olaFlow no habían modelos de cierre que representen el comportamiento del fluido al interactuar con un medio poroso dinámico, es por esto que no se consideraron los factores $[CT]_{k_{DY}}$ y $[CT]_{\epsilon_{DY}}$.

Modelo $k-\omega$ SST

Otro modelo disponible para utilizar en olaFlow es el $k-\omega$ SST (Shear-Stress Transport (SST) $k-\omega$) desarrollado por Menter (1994), es un híbrido que combina la robustez y precisión del modelo $k-\omega$ en las capas límite y separación del flujo, y el modelo $k-\epsilon$ que actúa mejor fuera de la capa límite (Higuera, 2015). El modelo promediado en el volumen se define mediante las siguientes expresiones:

$$\frac{\partial\langle k\rangle}{\partial t} + \nabla\cdot\left(\frac{1}{\phi}\langle k\rangle\langle\mathbf{u}\rangle\right) - \nabla\cdot\left[\left(\nu + \frac{\sigma_k}{\phi}\langle\nu_t\rangle\right)\nabla\langle k\rangle\right] + \frac{\beta^*}{\phi}\langle k\rangle\langle\omega\rangle =$$

$$min\left[\frac{2}{\phi^2}\langle\nu_t\rangle\left|\frac{\nabla\langle\mathbf{u}\rangle(\nabla\langle\mathbf{u}\rangle)^t}{2}\right|^2; \frac{c_1\beta^*}{\phi}\langle k\rangle\langle\omega\rangle\right] + [CT]_k \tag{3.10}$$

$$\frac{\partial \langle\omega\rangle}{\partial t} + \nabla \cdot \left(\frac{1}{\phi}\langle\omega\rangle\langle\mathbf{u}\rangle\right) - \nabla \cdot \left[\left(\nu + \frac{\sigma_\omega}{\phi}\langle\nu_t\rangle\right)\nabla\langle\omega\rangle\right]$$

$$- (1 - F_1^+)\langle CD_{k\omega}\rangle + \frac{\beta}{\phi}\langle\omega\rangle^2 = \frac{2}{\phi}\gamma\left|\frac{\nabla\langle\mathbf{u}\rangle + (\nabla\langle\mathbf{u}\rangle)^T}{2}\right|^2 \quad (3.11)$$

con:

$$\langle\nu_t\rangle = min\left[\phi\frac{\langle k\rangle}{\langle\omega\rangle}; \phi\ \frac{a_1\langle k\rangle}{F_2^+\sqrt{2}\left|\frac{\nabla\langle\mathbf{u}\rangle+(\nabla\langle\mathbf{u}\rangle)^T}{2}\right|}\right] \quad (3.12)$$

$$\langle CD_{k\omega}\rangle = max\left(2\sigma_\omega\frac{\nabla\langle k\rangle\cdot\nabla\langle\omega\rangle}{\langle\omega\rangle}; 10^{-10}\right) \quad (3.13)$$

$$F_1^+ = \tanh\left(min\left(min\left[max\left(\phi^{3/2}\frac{\sqrt{\langle k\rangle}}{\beta^* y^+\langle\omega\rangle};\ \ \phi^2\frac{500\nu}{\langle\omega\rangle(y^+)^2}\right); \phi\frac{4\sigma_\omega\langle k\rangle}{\langle CD_{k\omega}\rangle(y^+)^2}\right]; 10\right)\right)^4 \quad (3.14)$$

$$F_2^+ = \tanh\left(min\left[max\left(2\phi^{3/2}\frac{\sqrt{\langle k\rangle}}{\beta^* y^+\langle\omega\rangle}; \phi^2\frac{500\nu}{\langle\omega\rangle(y^+)^2}\right)\right]; 100\right)^2 \quad (3.15)$$

donde:

- $\langle k\rangle$ es la energía cinética turbulenta promediada en el volumen.
- t es la variable tiempo
- ϕ es la porosidad total.
- $\mathbf{u}$ es el vector de velocidad del fluido.
- $\langle\rangle$ es el operador de promediado en el volumen.
- ν es la viscosidad cinemática del fluido.
- ν_t es la viscosidad cinemática turbulenta del fluido.
- $\langle\omega\rangle$ es la frecuencia de turbulencia promediada en el volumen.
- T es la traspuesta de una matriz.
- $[CT]_k = 0$ es un término de cierre. Al momento del desarrollo de la tesis de Higuera (2015) no habían términos de cierre disponibles para el modelo $k - \omega\, SST$, por lo tanto no se consideraron en olaFlow.

La variable y^+ en las ecuaciones 3.14 y 3.15 representa la distancia desde el punto de análisis a la capa límite más cercana.

La función F_1^+ presenta valores entre 0 y 1, entonces cuando $F_1^+ = 0$ el punto se encuentra alejado de una capa límite ($y^+ >> 0$), y por lo tanto se utiliza el modelo $k - \epsilon$. En caso

contrario, cuando $F_1^+ = 1$, el punto de análisis se encuentra muy cerca de la capa límite ($y^+ \simeq 0$), entonces se utiliza el modelo $k-\omega$. Además, esta función se utiliza para calcular los valores constantes (σ_k, σ_ω, β o γ) que varían entre los modelos $k-\omega$ y $k-\epsilon$ como se observa en la Tabla 3.1, mediante $\psi_n = F_1^+\psi_1 + (1 - F_1^+)\psi_2$.

Tabla 3.1: constantes en el modelo $k-\omega$ SST.

Constante	$k-\omega\,(\psi_1)$	$k-\epsilon\,(\psi_2)$
σ_k	0.85	1
σ_ω	0.5	0.856
β	0.075	0.0828
γ	0.5532	0.4403
β^*	0.09	0.09
a_1	0.31	0.31
c_1	10.0	10.0

Fuente: (Higuera, 2015).

3.1.3. Método de volumen finito

Para resolver el sistema de ecuaciones diferenciales parciales VARANS en OpenFOAM, es necesario primero aproximarlas utilizando el método de volumen finito (o FVM por sus siglas en inglés) desarrollado por Chorin (1968) y ampliado por Kim & Moin (1985). En este método, el dominio se subdivide en una cantidad finita de volúmenes, donde las variables desconocidas se calculan en el centroide de cada uno de ellos.

La discretización de las ecuaciones se lleva a cabo sobre el dominio, transformando las ecuaciones diferenciales parciales en un sistema algebraico de ecuaciones. El método de volumen finito se divide en dos pasos; primero las ecuaciones diferenciales parciales son integradas y transformadas en ecuaciones de balance sobre un elemento, de esta manera se obtiene una serie de ecuaciones discretizadas. Luego se utilizan perfiles de interpolación para conocer el comportamiento de las variables en las celdas del dominio (Moukalled et al., 2016).

3.1.4. Método VOF

Cuando se utilizan mallas Eulerianas, es necesario realizar cálculos en cada celda del dominio donde se encuentra el fluido, con el objetivo de conocer el comportamiento de ese fluido en cada paso de tiempo. En cada celda donde se encuentra el fluido se deben promediar sus propiedades, lo que genera un comportamiento de suavizado, provocando resultados incorrectos en la superficie libre (Hirt & Nichols, 1981).

El método Volumen de Fluido o VOF por sus siglas en inglés, fue desarrollado por Hirt & Nichols (1981) para describir la superficie libre; en este método cada una de las celdas del dominio tiene un valor que define en qué fase se encuentra, pero como olaFlow solo permite la interacción entre las fases aire - agua, entonces se toma una función δ cuyo valor será 1 si la celda está llena de agua y 0 si por el contrario sólo contiene aire. El tercer posible caso es que la celda contenga agua y aire, esto ocurre en la zona de la interfaz.

Este método tiene la ventaja de poder representar complejas configuraciones de superficie libre de manera muy fácil sin que aumente el costo computacional, ya que no se requiere el movimiento de la malla. Por otra parte, va disminuyendo su efectividad cuando la tensión superficial cobra relevancia (Higuera, 2015). La ecuación que describe el movimiento de las fases se define como:

$$\frac{\partial \delta}{\partial t} + \nabla \cdot (\delta \mathbf{u}) = 0 \tag{3.16}$$

donde $\mathbf{u}$ es la velocidad del fluido. Luego, al realizarse el proceso de promediado del volumen se obtiene la ecuación de conservación de la masa (3.4). Para más detalle sobre la derivación de las ecuaciones es aconsejable que el lector recurra a Higuera (2015).

3.2. Configuración del modelo

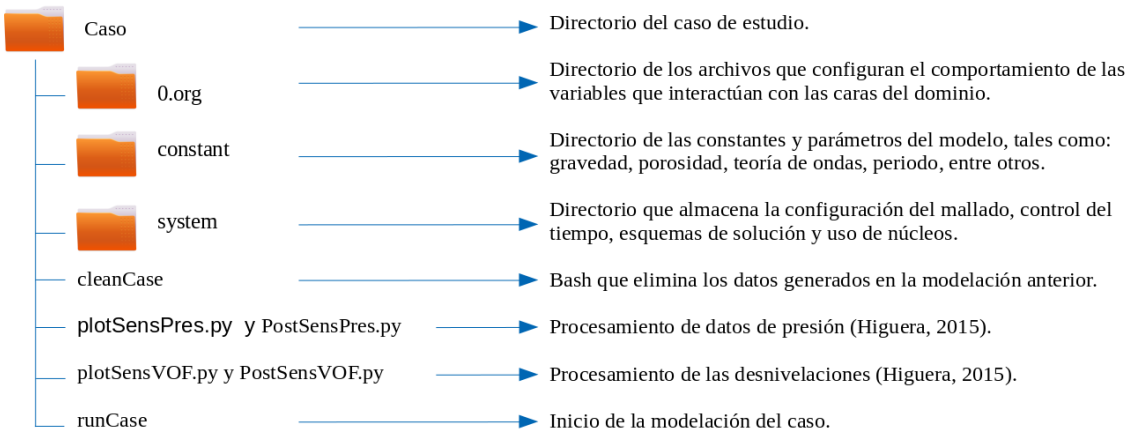
El modelo OpenFOAM, para funcionar, necesita una serie de archivos distribuidos en distintos directorios del caso de estudio. En estos archivos están definidos los parámetros y variables básicas que establecen el comportamiento del modelo. Debido a su importancia,

en esta sección se presentan en detalle los componentes necesarios para que el modelo pueda correr.

3.2.1. Directorios del modelo

OpenFOAM tiene una serie de carpetas y archivos con distintos parámetros que deben definirse correctamente para obtener resultados. En la Figura 3.1 se observa el directorio principal de un caso, que se compone de algunas subcarpetas y archivos que contribuyen al funcionamiento del modelo.

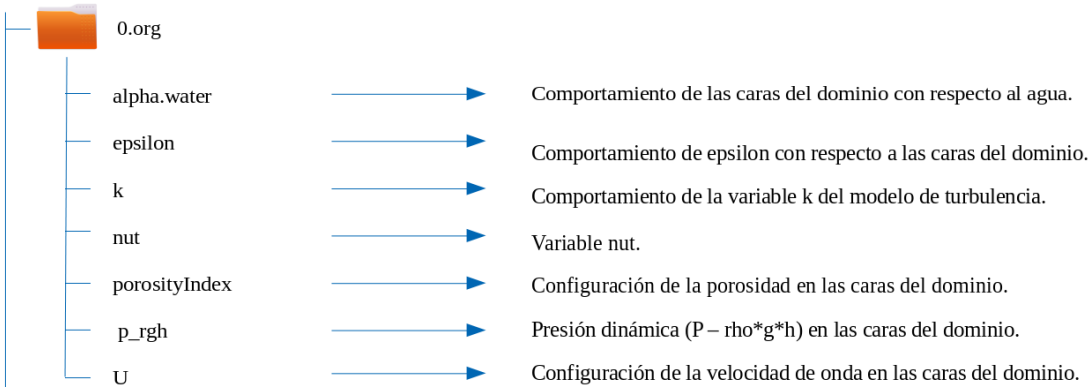
Figura 3.1: directorio principal de OpenFOAM, sus respectivos archivos y subcarpetas.



Fuente: elaboración propia.

Dentro de la carpeta *0.org* se encuentran los archivos que contienen las propiedades del fluido antes de iniciar la modelación, es decir, en el tiempo cero (Figura 3.2). Esta carpeta es copiada con el nombre *0* y luego se inicia la modelación del caso; de esta manera, el investigador puede volver a correr el caso con sus parámetros intactos las veces que sean necesarias.

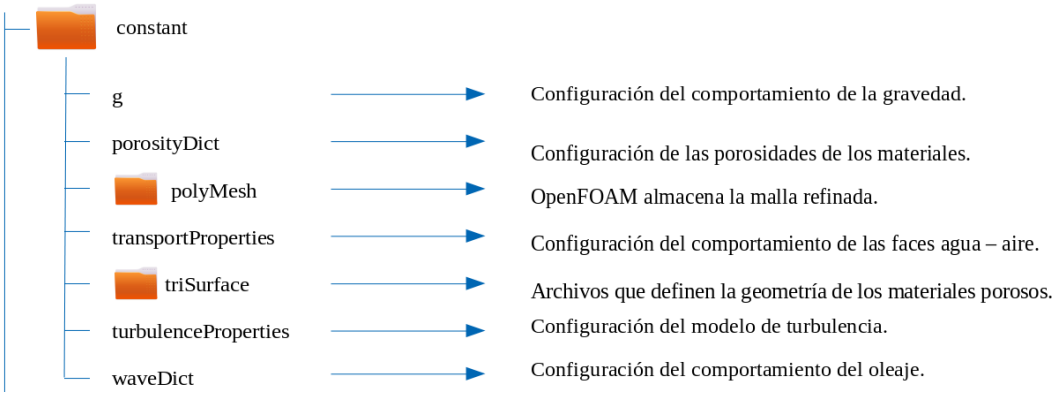
Figura 3.2: subcarpeta 0.org de un caso en OpenFOAM.



Fuente: elaboración propia.

En la carpeta *constant* se almacenan las constantes del modelo; la malla, la teoría de ondas en la cara generadora de oleaje, la gravedad, las propiedades de los materiales porosos, la definición del modelo de turbulencia y sus respectivas propiedades, entre otros parámetros. Los archivos carpetas almacenados en este directorio se pueden observar en la Figura 3.3.

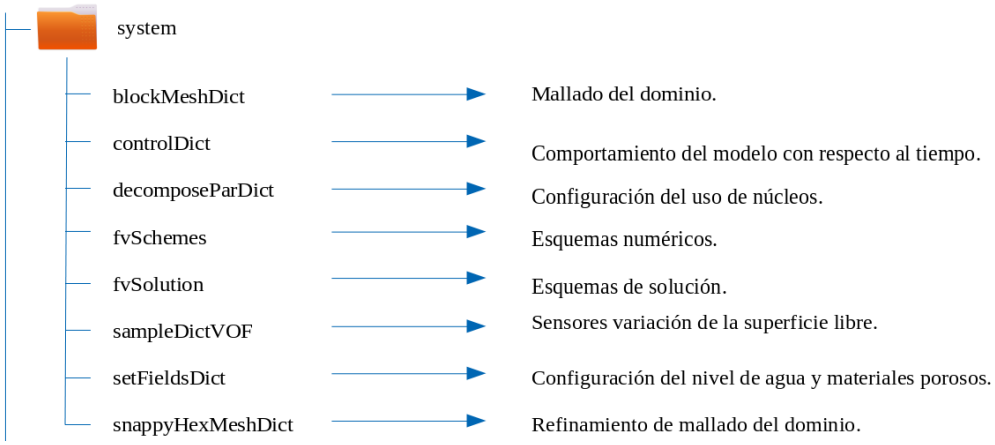
Figura 3.3: directorio que almacena las principales constantes del modelo.



Fuente: elaboración propia.

En el directorio *system* se almacenan archivos con las propiedades que definen las variables del modelo; el tiempo de modelación, profundidad del agua, el dominio computacional, densidad y forma del mallado, esquemas numéricos y de solución (Figura 3.4).

Figura 3.4: directorio system de OpenFOAM.



Fuente: elaboración propia.

3.2.2. Condiciones de borde y propiedades físicas

La definición correcta del comportamiento de los bordes que componen el dominio en OpenFOAM es básica para obtener resultados de buena calidad. Es más, un error en la definición de los contornos puede implicar que el modelo no inicie la resolución del problema. En la Tabla 3.2 se presenta la distribución de los parámetros utilizados en los contornos del dominio para todos los casos modelados, y a continuación se definen:

- waveDict: archivo que define los parámetros de oleaje en el borde de generación.
- zeroGradient: la condición límite de Newman indica que el gradiente de una propiedad como la tempreratura del fluido, presión, velocidad del fluido, entre otros, se establece en cero.
- fixedValue: esta condición indica que la propiedad toma un valor constante al interactuar con el borde.
- inletOutlet: cuando el fluido sale del dominio por el borde que tiene definida esta condición, se le aplica la condición zeroGradient. Cuando el fluido entra al dominio, el borde se comporta como fixedValue.
- uniform: condición que indica que la propiedad en todo el dominio o borde donde es aplicada tiene un valor definido.
- epsilonWallFunction: condición que proporciona disipación de la turbulencia en el borde.
- kqRWallFunction: esta condición define el comportamiento del borde en el campo k (energía cinética turbulenta) en el borde como zeroGradient.
- calculated: borde calculado a partir de otros componentes del dominio.
- NutkWallFunction: esta función pared genera una condición de viscosidad de turbulencia a partir de la energía cinética turbulenta.
- cmu: C_μ es un coeficiente de cierre del modelo $k - \epsilon$, que por defecto es 0.09.
- kappa: constante de Von Karman, con un valor por defecto de 0.41.
- E: constante del modelo. Para paredes lisas toma el valor de 9.8.
- Omega: archivo utilizado por el modelo $k - \omega\, SST$, el cual representa la tasa de disipación específica de turbulencia en los bordes donde es definida.
- OmegaWallFunction: genera una restricción en la disipación específica de la turbulencia.
- FixedFluxPressure: conidición similar a zeroGradient, pero se agrega cuando las fuerzas de cuerpo como la tensión superficial o la gravedad están presentes en las ecuaciones.
- totalPressure: corresponde a una condición de valor fijo, calculada a partir de la presión total y la velocidad local.
- gamma: proporción de calores específicos, por defecto es 1.
- waveVelocity: velocidad de entrada de la onda.

- WaveAbsorption2DVelocity: condición de contorno para la absorción activa de ondas, fuera del régimen de aguas poco profundas.
- absorptionDir: corresponde al ángulo (en grados) en el que la absorción es aplicada, considerando el eje x como cero.

Tabla 3.2: distribución de los parámetros y condiciones de borde utilizados para definir el comportamiento del dominio en los casos modelados.

Archivo	**Entrada**	**Salida**	**Fondo**
alpha.water	waveDict	zeroGradient	zeroGradient
epsilon	zeroGradient	zeroGradient	epsilonWallFunction
k	zeroGradient	zeroGradient	kqRWallFunction
nut	calculated uniform 0	calculated uniform 0	nutkWallFunction cmu 0.09 kappa 0.41 E 9.8 uniform 0
Omega	inletOutlet uniform 0.001	inletOutlet uniform 0.001	omegaWallFunction uniform 0.001
PorosityIndex	zeroGradient	zeroGradient	zeroGradient
p_rgh	fixedFluxPressure uniform 0	fixedFluxPressure uniform 0	fixedFluxPressure uniform 0
U	waveVelocity uniform 0	waveAbsorption2DVelocity absorptionDir 0.0 uniform 0	fixedValue uniform

Archivo	**Materiales**	**Atmósfera**
alpha.water	zeroGradient	inletOutlet uniform 0
epsilon	epsilonWallFunction	InletOutlet uniform 0.0001
k	kqRWallFunction	inletOutlet uniform 0.0001
nut	nutkWallFunction cmu 0.09	calculated uniform 0 kappa 0.41 E 9.8 uniform 0
Omega	omegaWallFunction uniform 0.001	inletOutlet uniform 0.001
porosityIndex	zeroGradient	zeroGradient
p_rgh	fixedFluxPressure uniform 0	totalPressure gamma 1 uniform 0
U	fixedValue uniform 0	PressureInletOutletVelocity uniform 0

Fuente: elaboración propia.

Parámetros físicos

El modelo numérico considera algunos parámetros físicos para poder realizar la simulación de los casos. Estos valores se mantuvieron constantes en todos los casos, ya que en la mayoría los estudios analizados no se presentan como supuestos. A continuación, en la Tabla 3.3 se indican los valores utilizados.

Tabla 3.3: parámetros físicos utilizados en todos los casos modelados.

Símbolo	Propiedad	Valor	Unidad de medida
ρ_w	densidad del agua	1000	kg/m^3
ν_w	viscosidad cinemática del agua	$1 \cdot 10^{-6}$	m^2/s
ρ_a	densidad del aire	1.0	kg/m^3
ν_a	viscosidad cinemática del aire	$1.48 \cdot 10^{-5}$	m^2/s
σ	tensión superficial	0.07	N/m
g	Aceleración de gravedad	9.81	m/s^2

Fuente: elaboración propia.

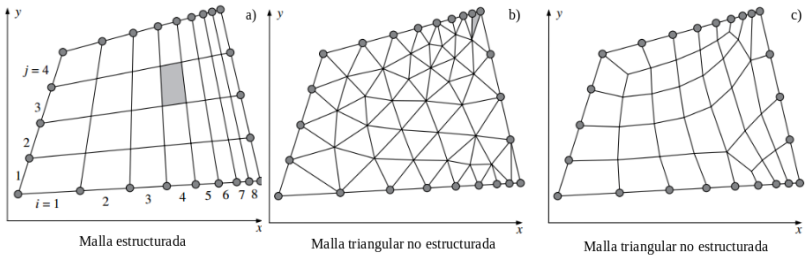
3.2.3. Configuración del mallado

En el archivo **blockMeshDict** del directorio *system* se configura el mallado del dominio. Se genera un poliedro (generalmente un hexaedro rectangular) que contiene en su interior todos los elementos que se van a modelar, y se define el comportamiento de cada cara del dominio y el ancho de las celdas.

Los resultados que proporciona el modelo se ven afectados por la configuración de la malla, por lo que es necesario introducir algunos criterios para asegurar su calidad. Es importante tener en cuenta que la malla está compuesta por una serie de celdas, donde el dominio se subdivide y se calcula la solución aproximada para cada subdominio, lo que permite conseguir una imagen completa del flujo de fluidos en todo el dominio geométrico (Gómez, 2017).

La malla puede clasificarse en estructurada, no estructurada o una combinación de ambas. Una malla estructurada está compuesta por celdas planas de cuatro lados (en 2D) o seis lados para la modelación en 3D. La principal ventaja en las mallas estructuradas es que permiten una solución con mucho menos celdas que las no estructuradas para un mismo dominio, lo que se traduce en menos cálculos. Sin embargo, es poco flexible para adaptarse a geometrías complejas (García et al., 2009). Para los propósitos de este trabajo solo se necesita trabajar con mallas estructuradas.

Figura 3.5: ejemplo de una malla estructurada versus mallas no estructuradas.



Fuente: Çengel & Cimbala (2006).

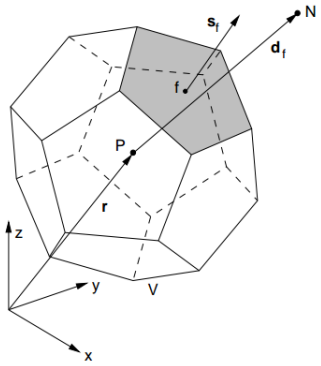
OpenFOAM es un modelo que utiliza el método de volúmenes finitos como aproximación para el proceso de discretización. En este modelo se seleccionan los métodos de aproximación de las integrales de superficie y volumen subdividiendo el dominio en una cantidad finita de volúmenes de control. En base a este método es necesario cumplir con algunos criterios de calidad del mallado.

- **Relación de aspecto (r_{asp})**: es la relación entre el lado más largo y el más corto del elemento. En OpenFOAM (r_{asp}) debe ser menor a 10 si la deformación es en dirección del flujo. Para los modelos CFD no se recomiendan r_{asp} muy grandes porque esto produce grandes cambios en algunas variables y no en otras, dando como resultado soluciones poco estables y oscilatorias (Palacios, 2015). En este trabajo se consideró la relación de aspecto como una variable que puede influir en los resultados, por lo que se probaron dos valores; 1.0 y 2.49, los que son presentados más adelante en este mismo capítulo.
- **Oblicuidad o asimetría**: Es un tipo de error de difusión numérica, que reduce la precisión de las integrales que calculan el flujo entre celdas adyacentes. La asimetría está dada por la relación de la distancia entre el centro de la celda y el centro de las caras adyacentes. Afecta particularmente a las mallas tetraédricas, mientras que

las mallas hexagonales tienen mejores resultados (Rhoads, 2014). Es por esto que en todos los casos modelados en este documento se utilizan celdas hexagonales.

- **No ortogonalidad**: Una malla es ortogonal si todas las líneas que la componen forman un ángulo recto al cruzarse. La no ortogonalidad está dada por el ángulo θ formado entre los vectores $\mathbf{s}_f$ y $\mathbf{d}_f$, donde $\mathbf{s}_f$ es un vector perpendicular al área con centroide en **f** y $\mathbf{d}_f$ es el vector formado por los puntos **P** y **N** de la cara adyacente (Figura 3.6). Mientras $\theta < 50°$ la malla no necesitará corrección. Para evitar este problema se activa la función *nNonOrthogonalCorrectors* dentro del archivo **fvSolution**.

Figura 3.6: celda polihédrica.



Fuente: Jasak (2009).

- **Smoothness**: El suavizado de la malla está dado por la relación de las dimensiones de las celdas adyacentes. Cuando se aumenta la densidad de la malla en zonas donde se necesita mayor precisión, la malla tiene celdas con distintos tamaños, sin embargo, para dos celdas contiguas la diferencia de tamaño no debe ser superior al 20 %. En ninguno de los casos modelados se hace una variación en tamaño de las celdas a lo largo del dominio debido a que la geometría generada no es compleja.

OpenFOAM cuenta con una herramienta llamada *checkMesh*, la que se utiliza para analizar la calidad de la malla antes de iniciar la modelación. En ningún caso configurado en este documento *checkMesh* presenta errores en la definición del mallado.

3.2.4. Tiempo de modelación

El archivo **controlDict** está dentro del directorio *system*, y establece la configuración del comportamiento del modelo con respecto al tiempo de simulación, tiempo de escritura, intervalo de muestreo, entre otros. Es importante tener en cuenta la condición de Courant, que indica que el paso de tiempo (δt) debe cumplir con la ecuación 3.17 para asegurar la estabilidad numérica al ejecutar olaFlow:

$$C_o = \frac{\delta t \, |\mathbf{u}|}{\delta x_j} \leq 1 \tag{3.17}$$

donde:

- C_o es el número de Courant.

- **u** es la velocidad a través de la celda que se está analizando.
- δt es un paso de tiempo.
- δx_j es el tamaño de la celda.

Si el paso de tiempo no satisface esta condición, los resultados presentados por la simulación son incorrectos (Courant et al., 1967). Se debe elegir el δt basado en el peor caso, o sea, una gran velocidad del fluido combinada con un pequeño tamaño de celda, luego:

$$\delta t \leq \frac{C_o \delta x_j}{|\mathbf{u}|} \tag{3.18}$$

Si las velocidades del fluido son más bajas, se puede configurar un paso de tiempo δt más alto para mejorar el rendimiento computacional, en caso contrario, el paso de tiempo debe ser menor y, por lo tanto, el rendimiento computacional disminuye (Greenshields, 2017).

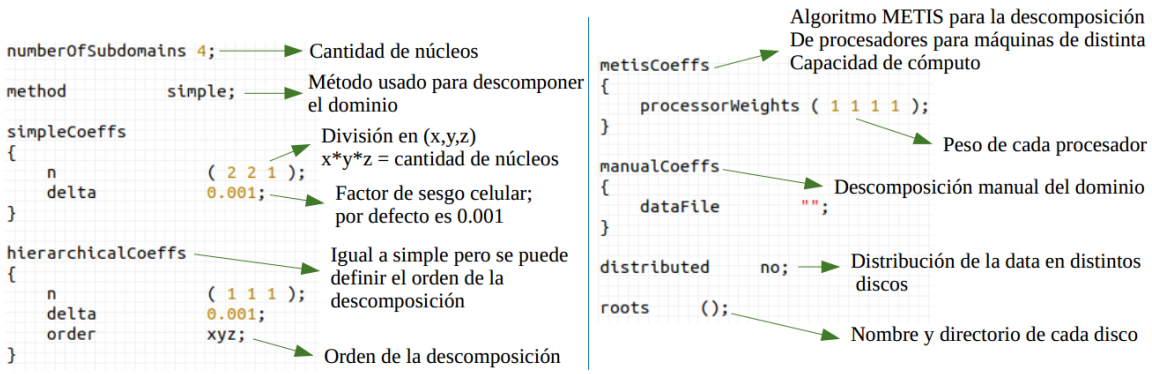
En todos los casos modelados el número de Courant se estableció en 0.15 para 12.5 celdas por altura de ola, debido a que en Larsen et al. (2018) se analizó, obteniendo resultados óptimos con este valor.

El tiempo de modelación se configuró para cumplir la sugerencia de Greenshields (2017) en todos los casos configurados; deben pasar a lo menos unas diez ondas por el dominio para alcanzar la estabilidad temporal.

3.2.5. Descomposición del dominio

Dentro del archivo **decomposeParDict** se configura el uso de múltiples núcleos en la resolución del problema (Figura 3.7). Esto condiciona al usuario a ejecutar la herramienta **decomposePar** para descomponer el problema en partes más pequeñas para cada núcleo. Luego de esto se ejecuta el solucionador olaFlow y finalmente se vuelven a juntar las partes del problema con la herramienta **reconstructPar**.

Figura 3.7: archivo para la configuración de descomposición del dominio.



Fuente: elaboración propia.

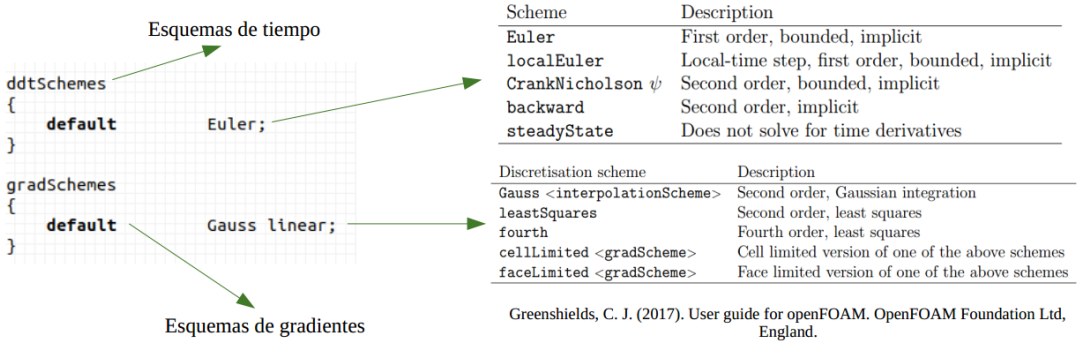
Todos los casos fueron modelados con 4 núcleos, y el dominio fue seccionado en esa misma cantidad de partes para optimizar el rendimiento.

3.2.6. Esquemas de discretización

Dentro del archivo **fvSchemes** están los esquemas numéricos, los cuales se utilizan para elegir el esquema de discretización para volúmenes finitos. Entre estos se pueden encontrar gradientes, valores de interpolaciones entre puntos, tolerancias, entre otros (ver Figura 3.8 y 3.9). Hay una extensa lista de términos disponibles en OpenFOAM; en la Tabla 3.4 se presentan los esquemas utilizados en el archivo **fvSchemes**. Dependiendo del caso a modelar, es necesario definir la configuración adecuada de esquemas de discretización:

- interpolationSchemes: Estos esquemas interpolan los valores entre el centro de las celdas al centro de las caras del dominio. Habitualmente el manual de OpenFOAM (Greenshields, 2017) recomienda *linear* por su buen rendimiento, excepto en los casos donde se usa DNS.
- ddtSchemes: Esquemas numéricos para la discretización del tiempo. Se usa el esquema *Euler* debido a que es estable para la primera derivada en el tiempo.
- snGradSchemes: Componente de gradiente normal a una cara de celda. Se elige *default corrected* por defecto.
- gradSchemes: Gradiente (∇) define todos los términos gradiente. Se mantiene *Gauss linear* por defecto.
- laplacianSchemes: Laplaciano (∇^2). La única opción de discretización es el esquema *Gauss* en conjunto con *linear*. Por defecto, el tercer esquema es *corrected*.
- divSchemes: Divergencia ($\nabla\bullet$). Los esquemas utilizados por defecto en los casos de este documento se pesentan en la Tabla 3.4.

Figura 3.8: propiedades para configurar los esquemas de tiempo y gradientes en OpenFOAM.



Scheme	Description
`Euler`	First order, bounded, implicit
`localEuler`	Local-time step, first order, bounded, implicit
`CrankNicholson` ψ	Second order, bounded, implicit
`backward`	Second order, implicit
`steadyState`	Does not solve for time derivatives

Discretisation scheme	Description
`Gauss <interpolationScheme>`	Second order, Gaussian integration
`leastSquares`	Second order, least squares
`fourth`	Fourth order, least squares
`cellLimited <gradScheme>`	Cell limited version of one of the above schemes
`faceLimited <gradScheme>`	Face limited version of one of the above schemes

Greenshields, C. J. (2017). User guide for openFOAM. OpenFOAM Foundation Ltd, England.

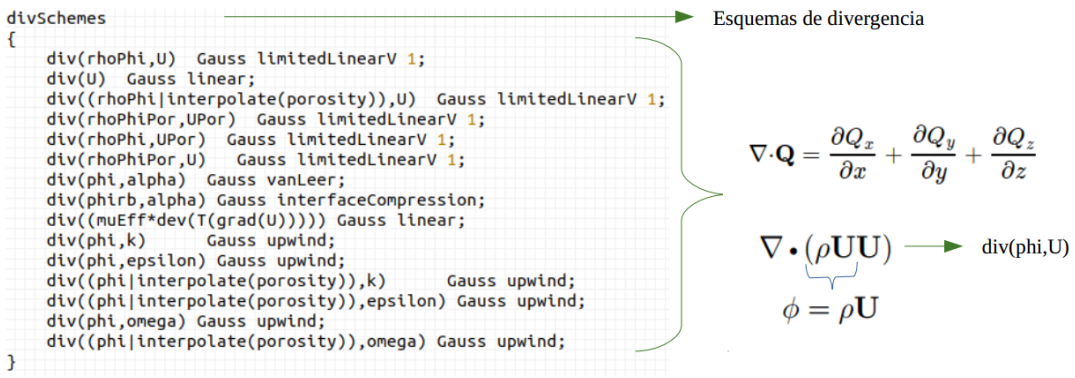
Fuente: elaboracción propia.

Tabla 3.4: esquemas de discretización utilizados en los casos modelados.

Propiedad	**Esquema utilizado**
interpolationSchemes	default linear
ddtSchemes	Euler
snGradSchemes	default corrected
gradSchemes	Gauss linear
laplacianSchemes	Gauss linear corrected
divSchemes	div(rhoPhi,U) Gauss limitedLinearV 1 div(U) Gauss linear div((rhoPhi|interpolate(porosity)),U) Gauss limitedLinearV 1 div(rhoPhiPor,UPor) Gauss limitedLinearV 1 div(rhoPhi,UPor) Gauss limitedLinearV 1 div(rhoPhiPor,U) Gauss limitedLinearV 1 div(phi,alpha) Gauss vanLeer div(phirb,alpha) Gauss interfaceCompression div((muEff*dev(T(grad(U))))) Gauss linear div(phi,k) Gauss upwind div(phi,epsilon) Gauss upwind div((phi|interpolate(porosity)),k) Gauss upwind div((phi*interpolate(rho)),k) Gauss upwind div((phi|interpolate(porosity)),epsilon) Gauss upwind div(phi,omega) Gauss upwind div((phi|interpolate(porosity)),omega) Gauss upwind div((phi*interpolate(rho)),omega) Gauss upwind div((phi*interpolate(rho)),epsilon) Gauss upwind

Fuente: elaboración propia.

Figura 3.9: esquemas de divergencia presentes en los casos modelados.



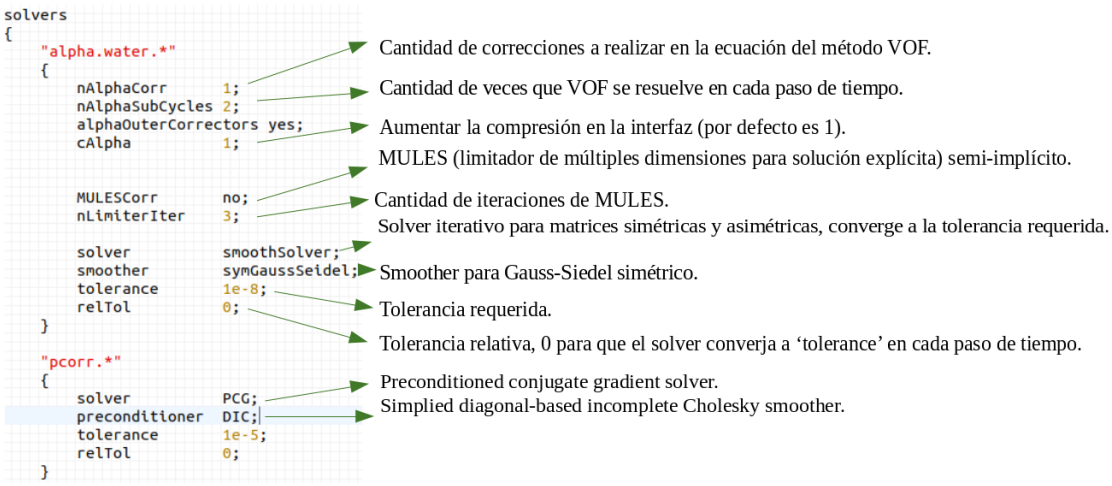
Fuente: elaboración propia.

3.2.7. Esquemas de solución

En OpenFOAM se deben definir las tolerancias y algoritmos de solución a utilizar para resolver una ecuación matricial no lineal proporcionada por los esquemas de discretización, la que se resuelve de manera iterativa en base a las propiedades configuradas en un archivo llamado **fvSolution** (Jones et al., 2016). En la figura 3.10 se describen las propiedades que deben estar presentes en el archivo **fvSolution**.

En la dinámica de fluidos computacional (CFD) rara vez se utilizan métodos directos para resolver los sistemas de ecuaciones lineales, sino más bien se utilizan métodos iterativos. Esto se debe a que no se necesita precisión en la solución en cada iteración, y además, ocupan menos recursos computacionales por cada iteración (Moukalled et al., 2016). Por ende todos los métodos utilizados en OpenFOAM son iterativos (Anexo D.2).

Figura 3.10: propiedades para la configuración de los esquemas de solución.



Fuente: elaboración propia.

3.2.8. Generación de ondas en OpenFOAM

Las instrucciones para el tipo de generación de oleaje en OpenFOAM se deben proporcionar en el archivo **waveDict**, que contiene las propiedades requeridas por el borde de generación de oleaje (Tabla 3.5). La mayoría de los valores relacionados a la absorción activa en el borde de generación se mantienen constantes, salvo algunas excepciones que modifican la onda propagada. A continuación se presentan los parámetros y criterios utilizados en los casos simulados.

- waveType: el modelo puede simular tanto oleaje regular como irregular. Esta propiedad se mantiene como oleaje regular en todos los casos modelados, debido a que "el uso de oleaje regular es la forma más eficiente de estudiar parámetros nominalmente constantes" (Kisacik et al., 2012).

- waveTheory: en el solver olaflow hay distintas teorías de ondas disponibles (Stokes I, II, V; teoría cnoidal, streamFunction y Boussinesq). Sin embargo, en todos los casos modelados se simuló la teoría de Stokes II, debido a que todos los artículos utilizados en la calibración recurrieron esta teoría.

- genAbs: esta variable controla la absorción activa de onda en el borde de generación de oleaje. Todos los casos modelados tienen activada esta opción, debido a que de lo contrario se generan problemas de inestabilidad en el modelo (Higuera, 2015).
- absDir: mediante esta variable se define la dirección de absorción. Esto es así porque el dominio en todos los casos modelados se trata de un canal numérico bidimensional, o sea, siempre el oleaje reflejado llega al borde de generación desde una sola dirección.
- nPaddles: cantidad de cortes verticales que tiene el borde de generación de oleaje. Ese parámetro influye en la absorción de oleaje en 3D, obteniendo mejores resultados en la absorción activa con valores mayores a 1. En los casos modelados se mantiene por defecto en 1.
- wavePeriod: en este parámetro se define el periodo que debe tener la onda generada en el borde de entrada.
- waveHeight: configuración de la altura de ola para la modelación.
- waveDir: en este parámetro se configura la dirección de propagación del oleaje, pero al tratarse de simulaciones bidimensionales, no se considera la variación de la dirección.
- wavePhase: fase de la onda generada, en radianes.
- tSmooth: el parámetro de suavizado de tiempo es necesario cuando se utiliza oleaje irregular y con distintas direcciones. Mediante esta propiedad se puede utilizar un factor de suavizado de la altura de ola justo en la zona de generación. Los casos analizados en este documento no presentaron esas características, por lo tanto, no se usa.

Tabla 3.5: parámetros definidos en el borde de generación de oleaje.

Propiedad	valor
waveType	regular
waveTheory	variable
genAbs	1
absDir	0.0
nPaddles	1
wavePeriod	variable
waveHeight	variable
waveDir	0.0
wavePhase	0.0
tSmooth	0.0
nPaddles	1

Fuente: elaboración propia.

3.3. Ensayos de calibración

La calibración y validación es una etapa esencial en el modelado, ya que permite conocer el grado de exactitud con que el modelo puede predecir el fenómeno que se está estudiando. Por una parte, la calibración corresponde a la introducción de los parámetros con que el modelo comienza a calcular los resultados en el dominio; tales como valores constantes en los modelos de turbulencia, densidad del fluido, porosidad de los materiales presentes en el dominio, condiciones de borde, teoría de ondas, número de Courant admisible, intervalo de tiempo para la escritura de los resultados, entre otros. Por otro lado, la validación del modelo se realiza comparando los resultados proporcionados luego de la modelación con series de datos obtenidos de manera empírica, esto es mediante modelos físicos de laboratorio o en terreno (Winckler, 2018). Si los datos obtenidos son concordantes con los datos empíricos, entonces para el caso analizado, el modelo tiene un buen comportamiento.

Los casos que se modelaron en este trabajo tuvieron como objetivo conocer qué variables influyen de manera más significativa en los resultados. En la Tabla 3.6 se pueden observar los casos modelados y las variables consideradas.

Tabla 3.6: variables consideradas para los casos modelados. Las cruces representan los casos no modelados y los vistos buenos son los casos en que se utilizó OpenFOAM.

Caso/Variable	Mallado (1)	Turbulencia (2)	Dominio (3)	Resolución temporal (4)
Oleaje progresivo (P)	✓	✓	✗	✓
Oleaje estacionario (E)	✓	✓	✓	✓
Oleaje en asomeramiento (A)	✗	✗	✗	✗
Oleaje en un muelle (M)	✓	✗	✗	✗

Fuente: elaboración propia.

Para entender el razonamiento de la elección de los casos modelados se debe considerar lo siguiente:

- La primera variable que se estudió fue la densidad del mallado, tanto para oleaje estacionario como progresivo (E-P-1), ya que probablemente es lo que más influye en los resultados. Una vez conocido qué modelo de turbulencia actuó mejor tanto para oleaje estacionario como progresivo, se mantuvo constante en la mayoría de los casos posteriores.

- La resolución temporal se investigó tanto para el caso de oleaje estacionario como para oleaje progresivo (E-P-4), ya que se pretendió demostrar que el criterio utilizado para elegir el tiempo de modelación generó la convergencia y estabilidad.

- El aumento del dominio solo se realizó para el caso de oleaje estacionario (E-3), con el objetivo de analizar la absorción activa presente en la paleta y su influencia sobre la onda reflejada.

- Al momento de modelar el oleaje en asomeramiento (A), no estaba considerado el uso del modelo de turbulencia $k-\omega\,SST$, por lo que este caso se modeló con el modelo $k-\epsilon$. Luego, al implementar el modelo $k-\omega\,SST$ en los casos de oleaje estacionario y progresivo se observó un comportamiento más cercano a la teoría. Pero por razones de costo computacional no fue posible repetir el caso de asomeramiento con el modelo $k-\omega\,SST$.

- Para el caso del muelle fue necesario hacer una variación del mallado (M-1), debido a que el criterio que dio buenos resultados para los casos precedentes no tuvo los mismos resultados en este caso, posiblemente debido al cambio en la geometría del caso con respecto a los anteriores.

3.3.1. Análisis de sensibilidad de la malla

Para configurar OpenFOAM en base a la resolución espacial que se le proporciona a los casos modelados, es necesario realizar pruebas con distintas dimensiones de celdas.

Los tres casos presentados a continuación tienen distintos criterios para definir las dimensiones de las celdas que componen la malla. El caso 01 considera el criterio propuesto por (Arjona, 2016), y para los casos 02 y 03 se considera el criterio presentado por (Larsen et al., 2018). Más adelante, estos criterios se comparan para definir el que tiene mejor comportamiento con respecto a la teoría.

Primer criterio para el dimensionamiento del mallado

En el primer caso se utilizaron las condiciones de mallado propuestos por Arjona (2016) para el modelo IH2VOF, el cual resuelve las VARANS en 2D con el modelo $k - \epsilon$.

Según Arjona (2016), las dimensiones máximas que el dominio debe tener en la vertical pueden ser entre 7 y 10 celdas por altura de ola, y en la horizontal se debe contar con entre 70 y 100 celdas por longitud de onda cuando la ola no rompe; en caso de que la ola sí rompa, en la horizontal deben haber más de 100 celdas por longitud de onda. Además, se debe tomar en cuenta que la relación entre el ancho y el alto de cada celda debe cumplir $\delta x < 2.5\,\delta y$, donde δx y δy son las de dimensiones ancho y alto de cada celda respectivamente. De esta manera se espera evitar la falsa rotura, producida por el traspaso de fluido desde celdas llenas a celdas vacías debido al método VOF.

- $\delta x < L/100$
- $\delta y < H/10$
- $\delta x < 2.5\,\delta y$

donde L corresponde a la longitud de onda, y H es la altura de ola que se le proporciona al modelo.

Segundo criterio para el dimensionamiento del mallado

En un estudio realizado por Larsen et al. (2018) se analizó el comportamiento del oleaje progresivo mediante el solver interFoam de OpenFOAM. El estudio se enfocó en analizar la precisión de los resultados obtenidos en las variaciones de la superficie libre. Para esto se simuló un canal de ondas con un largo correspondiente a una longitud de onda, y los parámetros indicados en la Tabla 3.7.

Larsen et al. (2018) consideró que el modelo tiene un mejor rendimiento con celdas de una relación de aspecto de $\delta x/\delta y = 1$, o sea, todas las celdas del dominio son cuadradas, entregando un rendimiento adecuado desde al menos unas 12.5 celdas por altura de ola, y llegando a resultados con alta precisión al utilizar 50 celdas por altura de ola, lo que en la

Tabla 3.7: parámetros de entrada para modelar los casos analizados, donde H es la altura de ola a propagar, T es el periodo de la ola y t es el tiempo modelado.

$H[m]$	$T[s]$	Teoría	Profundidad $[m]$	$t[s]$	δx [m]	δz [m]
0.125	2.0	Stream function	0.4	200.0	0.01	0.01

práctica conlleva un alto costo computacional. Es por esto que en ese estudio se opta por el uso de 12.5 celdas por altura de ola.

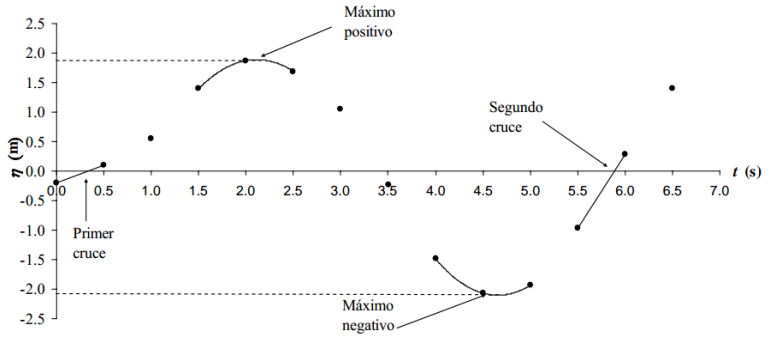
3.4. Desarrollo de herramientas para el análisis de datos

A lo largo de todos los canales simulados se pusieron sensores de desnivelación, los cuales contaron con datos de tiempo y nivel del agua. Esto se configuró mediante la herramienta **sampleDictVOF** de OpenFOAM, en la que se indica la cantidad de sensores a utilizar, ubicación de cada sensor y los métodos utilizados para obtener los datos. Dentro de estos métodos se puede elegir una interpolación de los datos con respecto a toda la cara, tomar solo el dato en el punto o una combinación de ambos.

La toma de datos mediante sensores para desarrollar los análisis pertenece al postproceso, es decir, se realiza luego de que el modelo haya terminado de simular el caso de estudio. Luego, se almacenan en un directorio del caso, discretizando el tiempo en subcarpetas que contienen los datos de desnivelación, coordenadas y paso de tiempo para cada sensor.

En este trabajo se tomaron los datos proporcionados por el modelo y se desarrolló un código en Python para la automatizar el proceso. Es así que para obtener las alturas de ola de cada sensor se implementó el método "pasos ascendentes por cero". Dicho método está basado en encontrar el máximo positivo entre dos cruces adyacentes y luego identificar el mínimo negativo entre los dos cruces siguientes, finalmente los valores absolutos se suman para encontrar la altura de ola (ver Silva (2005)).

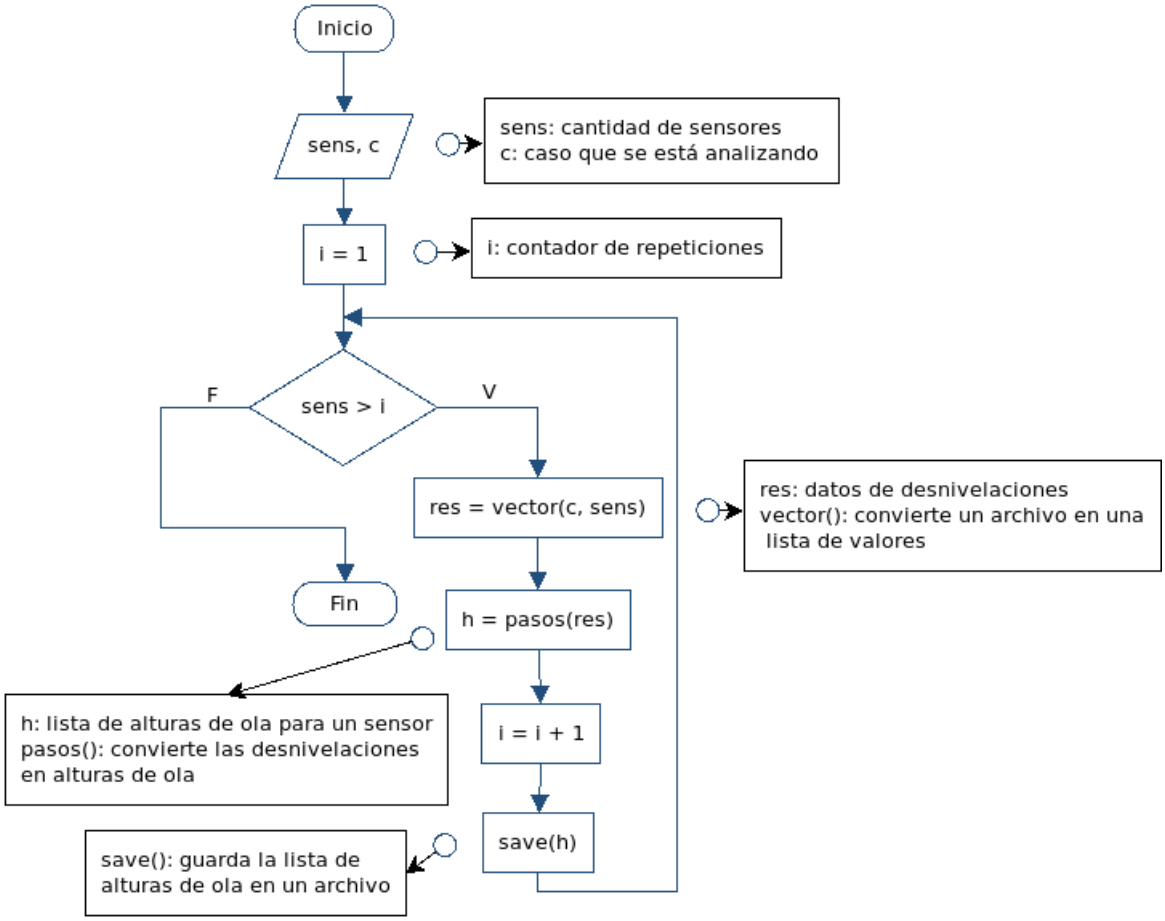
Figura 3.11: pasos ascendentes por cero.



Fuente: Silva (2005).

En el mismo script se dan las instrucciones para crear una carpeta en el caso de estudio y guardar las alturas de ola obtenidas en cada sensor (los parámetros, variables y listas son presentados en la Tabla 3.8). La función *vector* mostrada en el diagrama de la Figura 3.12 abre cada uno de los archivos de desnivelación proporcionados como resultado de la modelación de un caso, luego entrega estos datos a la función *pasos*. Esta función convierte las desnivelaciones en alturas de onda mediante la técnica pasos ascendentes por cero. Posteriormente las alturas de olas son guardadas en un archivo para su análisis.

Figura 3.12: diagrama de flujo del primer script.



Fuente: elaboración propia.

Luego, en un segundo script (Figura 3.13) se toman las alturas de ola ya calculadas en el script anterior, se calcula la altura significativa (H_s) y altura de ola media en cada uno de los sensores. La altura significativa (H_s) está definida como la media aritmética del 33 % de las alturas de ola más altas.

Es importante señalar que no se toma en cuenta la primera ola en cada sensor, esto es para no incluir datos cuando el modelo está iniciando la propagación, ya que se espera analizar datos que cuenten con estabilidad temporal y espacial. En la Tabla 3.9 se presentan los componentes del segundo script.

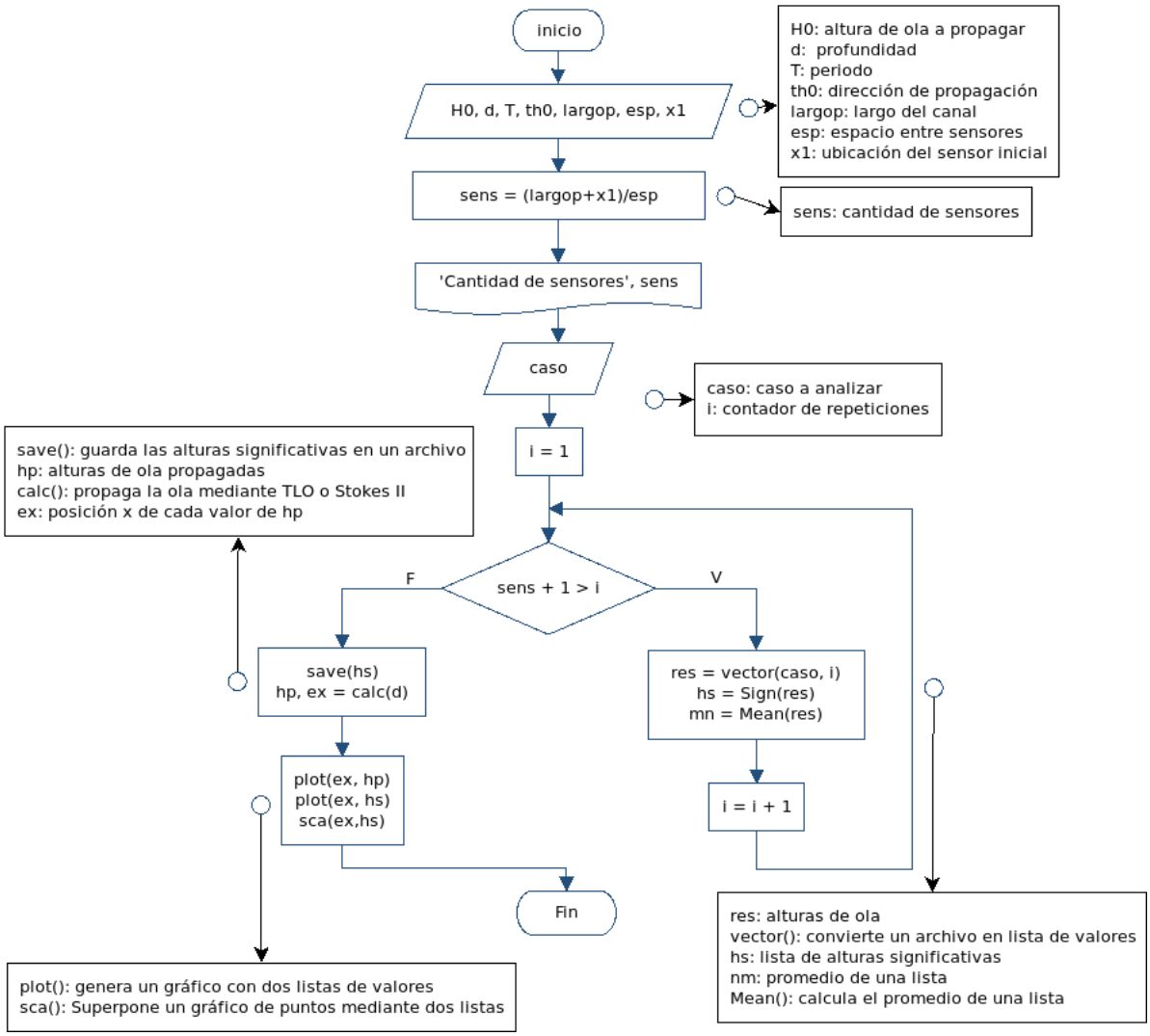
Las variaciones de la superficie libre en los casos analizados son principalmente comparadas con la teoría de Stokes II. Es por esto que a continuación se presenta el método de la obtención de las densivelaciones en el tiempo en base a esta teoría. En la Figura 3.14 se muestra el diagrama de flujo utilizado para calcular las desnivelaciones con una serie de puntos en el tiempo, y en la Tabla 3.10 se pueden observar las funciones, listas y parámetros que usa este script.

Tabla 3.8: descripción de los principales elementos presentes en el primer script.

Elemento	Tipo	Descripción	Valor
sens	constante	cantidad de sensores	entero
c	constante	caso que se está analizando	entero
i	variable	contador de repeticiones	entero
res	lista de valores	datos de desnivelaciones	real
vector	función	convierte un archivo en una lista de valores	-
pasos	función	convierte las desnivelaciones en alturas de ola	-
h	lista de valores	lista de alturas de ola para un sensor	real
save	función	guarda la lista de alturas de ola en un archivo	-

Fuente: elaboración propia.

Figura 3.13: diagrama de flujo del segundo script.



Fuente: elaboración propia.

Tabla 3.9: descripción de los principales elementos presentes en el segundo script.

Elemento	Tipo	Descripción	Valor
H_0	constante	Altura de ola a propagar	real
d	constante	profundidad	real
th0	constante	dirección de propagación	real
sens	constante	cantidad de sensores	real
largop	constante	largo del canal	real
esp	constante	espacio entre sensores	real
x1	constante	ubicación del sensor inicial	real
caso	constante	caso a analizar	entero
i	entero	contador de repeticiones	real
vector	función	convierte un archivo en lista de valores	-
res	lista de valores	alturas de ola	real
Sign	función	calcula la altura significativa (Hs)	-
hs	lista de valores	lista de alturas significativas	real
Mean	función	calcula el promedio de una lista	-
nm	constante	promedio de una lista	-
save	función	guarda las alturas significativas en un archivo	-
calc	función	propaga la ola mediante la TLO	-
hp	lista de valores	alturas de ola propagadas	real
ex	lista de valores	posición en x de cada valor de hp	real
plot	función	genera un gráfico con dos listas de valores	-
sca	función	superpone un gráfico de puntos mediante dos listas	-

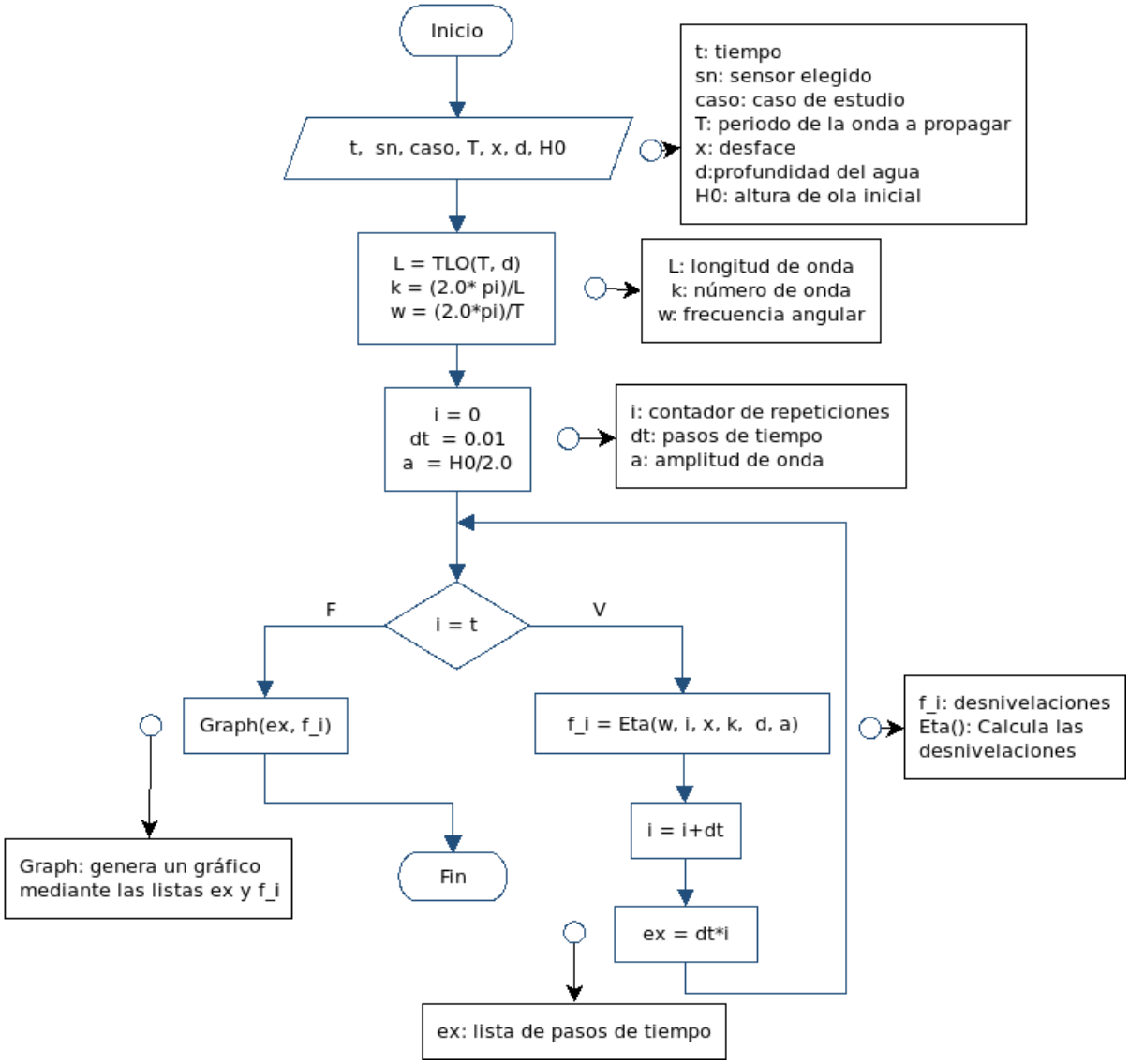
Fuente: elaboración propia.

Tabla 3.10: descripción de los elementos que componen el script para calcular la curva de Stokes II en el tiempo.

Elemento	Tipo	Descripción	Valor
t	constante	tiempo	real
sn	constante	sensor elegido	real
a	constante	amplitud de onda	real
caso	constante	caso de estudio	real
w	constante	frecuencia angular	real
k	constante	número de onda	real
x	constante	desfase	real
d	constante	profundidad del agua	real
H_0	constante	altura de ola inicial	real
L	constante	longitud de onda	real
TLO()	función	Teoría Lineal de Ondas	real
T	constante	periodo de la oda a propagar	real
i	variable	contador de repeticiones	entero
dt	constante	pasos de tiempo	real
f_i	lista de valores	desnivelaciones	real
ec	función	calcula las desnivelaciones	-
ex	lista de valores	lista de pasos de tiempo	real
Graph()	función	genera un gráfico mediante las listas ex y f_i	-

Fuente: elaboración propia.

Figura 3.14: diagrama de flujo para el cálculo de la onda de Stokes II.



Fuente: elaboración propia.

3.5. Casos de estudio

Todos los casos desarrollados en este documento consideraron alguno de los criterios para la resolución del mallado, presentados en la Tabla 3.11.

- Cuando se modeló oleaje progresivo, se compararon los tres criterios para definir cual de ellos tiene un comportamiento más cercano a la teoría.
- En el caso de oleaje estacionario nuevamente se compararon los criterios CR01, CR02 y CR03 con la teoría.
- Se realizó un análisis de sensibilidad al tiempo de modelación, tanto para el caso de oleaje progresivo como para el oleaje estacionario, considerando solo el criterio CR01 (Tabla 3.11), y dejando todos los demás parámetros constantes.
- Se analizaron los límites de generación y salida de oleaje, aumentando la longitud del canal, y utilizando el criterio CR01 para resolución del mallado.

Una vez obtenido el criterio de resolución del mallado, el tiempo mínimo de simulación necesario para lograr la estabilidad temporal, la influencia de los bordes sobre el fluido, y analizado el comportamiento de los modelos de turbulencia, se comparó el modelo con el estudio realizado por Kamath et al. (2017), donde se observó el oleaje interactuando con un cambio en la pendiente del fondo. Finalmente, se compararon los resultados del modelo con un estudio realizado por Kisacik et al. (2012), donde el oleaje interactuó con una estructura.

Tabla 3.11: resolución espacial para los casos simulados, donde dx y dz corresponden al ancho y alto de cada celda respectivamente.

Caso	Criterio	dx [cm]	dz [cm]	Celdas por altura de ola
CR01	Arjona (2016)	2.49	1.0	10
CR02	Larsen et al. (2018)	0.8	0.8	12.5
CR03	Larsen et al. (2018)	0.4	0.4	25

3.5.1. Oleaje progresivo

El criterio utilizado por Arjona (2016) (CR01) se comparó con casos en que se aumentó la densidad de la malla y se cambió la relación de aspecto de las celdas a 1:1 (configuración propuesta por Larsen et al. (2018), CR02 y CR03 de la Tabla 3.11). Para conocer la sensibilidad de los resultados a la resolución espacial solo se modificaron las dimensiones de las celdas y el modelo de turbulencia, manteniendo los demás parámetros constantes.

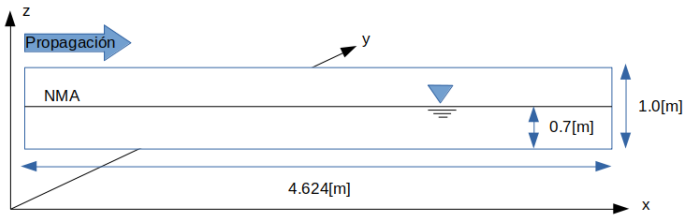
Para deducir la teoría de Stokes II, la Universidad de Cantabria (2000) utilizó la ecuación de dispersión de la TLO (ecuación 2.2 de la sección 2.3) con el fin de simplificar el procedimiento. Esto implica que la ecuación para calcular la longitud de onda tanto para la TLO como para Stokes II es la misma en ese caso. Para los propósitos de este trabajo basta solo con tener una solución aproximada de la longitud de onda para el régimen de Stokes II. Es así que se modeló un canal de 4.624 m de largo (Figura 3.15), lo que corresponde a una longitud de onda, que fue calculada con la TLO utilizando los parámetros mostrados en la Tabla 3.12.

Los parámetros presentados en la Tabla 3.12 generaron una propagación de onda en aguas intermedias, donde los términos no lineales comienzan a afectar la forma de la onda. La elección de la teoría de ondas a utilizar estuvo dada por la Figura 2.7 de la sección 2.3, por lo que Stokes II fue adecuada para describir su comportamiento. En la tabla 3.11 se presenta la resolución utilizada en las mallas del dominio.

Tabla 3.12: parámetros de entrada para modelar los casos analizados, donde H es la altura de ola a propagar, T es el periodo de la ola y t es el tiempo modelado.

$H[m]$	$T[s]$	Teoría	Profundidad $[m]$	$t[s]$
0.1	2.0	Stokes II	0.7	20.0

Figura 3.15: dimensiones del canal de ondas modelado para oleaje progresivo, donde el NMA es el nivel medio del agua.

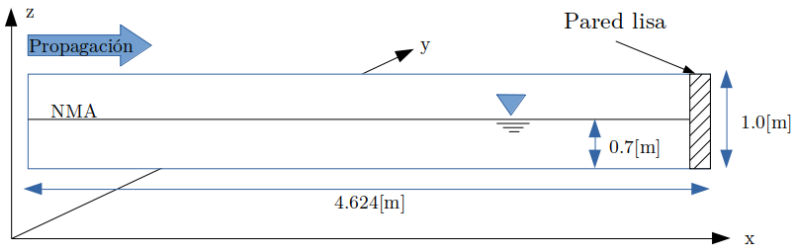


Fuente: elaboración propia.

3.5.2. Oleaje estacionario

Para analizar el comportamiento del modelo ante la reflexión, se mantuvieron los parámetros de la Tabla 3.12 constantes, y se cambiaron las condiciones de borde en la salida del canal (Figura 3.16) para que actúe como una pared lisa, de esta manera se simuló una onda estacionaria. En la primera prueba no se utilizaron modelos de turbulencia, en la segunda y tercera prueba se activaron los modelos de turbulencia $k-\epsilon$ y $k-\omega\, SST$ respectivamente.

Figura 3.16: configuración del canal de olas para analizar la reflexión mediante oleaje estacionario. NMA es el nivel medio del agua.

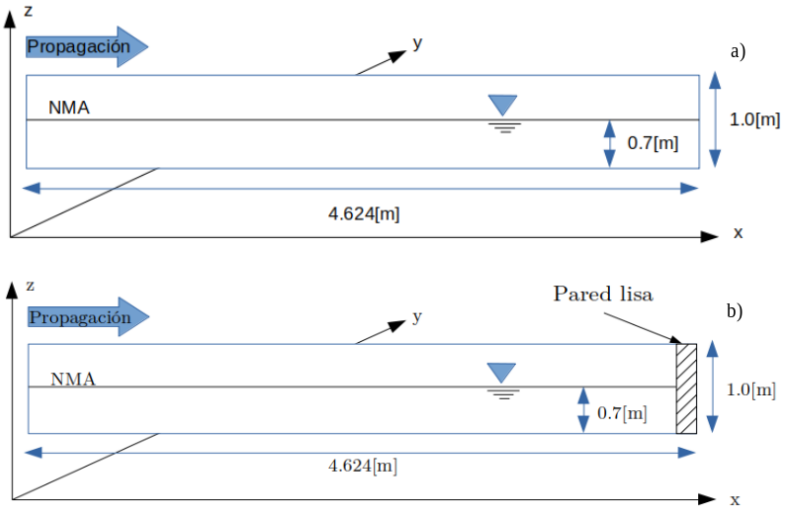


Fuente: elaboración propia.

3.5.3. Estabilidad temporal de los casos de estudio

El modelo debe dejar pasar por el dominio una cierta cantidad de ondas para poder alcanzar estabilidad temporal. Según Greenshields (2017) el fluido debe pasar a lo menos unas 10 veces. Para comprobar el rendimiento de este criterio de estabilidad, se aumentó la cantidad de tiempo simulado en el caso CR01 (Tabla 3.11) de 20 a 100 y a 200 s. La configuración utilizada en el modelo se presenta en la Figura 3.17.

Figura 3.17: configuración utilizada en el modelo numérico para analizar la convergencia temporal. a) se utiliza para oleaje progresivo y b) para oleaje estacionario.



Fuente: elaboración propia.

Se analizó sólo el caso CR01 presentado en la Tabla 3.11 debido a que no se observaron diferencias significativas con respecto a los casos CR02 y CR03 descritos en la misma

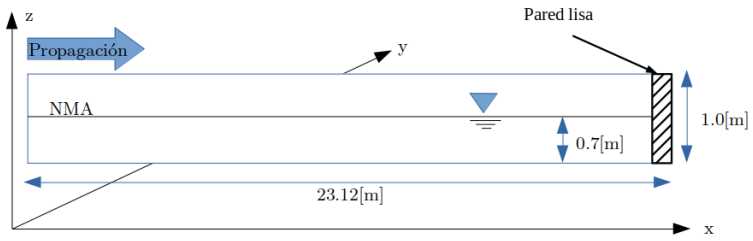
tabla. Además, el tiempo de modelación fue mucho menor que cuando se utilizó el criterio de Larsen et al. (2018) (CR02 y CR3).

Para oleaje estacionario se utilizó el modelo $k - \epsilon$, y para oleaje progresivo se utilizó el modelo $k - \omega$ SST, ya que según los resultados obtenidos en las secciones 4.1 y 4.2 respectivamente, proporcionaron resultados más cercanos a la teoría.

3.5.4. Análisis del dominio

Con el objetivo de analizar la influencia de los bordes en los resultados, se tomó el caso CR01 (Tabla 3.11) y se volvió a correr aumentando el tamaño del canal pero con todas las demás características iguales, incluyendo el modelo de turbulencia ($k - \epsilon$). El canal se aumentó a 5.0 longitudes de onda, lo que equivale a 23.12 m (Figura 3.18), y el modelo se corrió durante 100 s, lo que se traduce en el paso de unas 10 longitudes de onda por el dominio. Para calcular la altura significativa (H_s) a lo largo del canal, el espaciamiento de los sensores se mantuvo en 1.0 cm, dando como resultado un total de 2302 sensores.

Figura 3.18: aumento de las dimensiones del canal para analizar el comportamiento de OpenFOAM.



Fuente: elaboración propia.

Para realizar la comparación entre la curva obtenida por el modelo y la teoría se descartaron las primeras 10 olas en cada sensor, hasta alcanzar un régimen estadísticamente estacionario.

3.5.5. Asomeramiento del oleaje

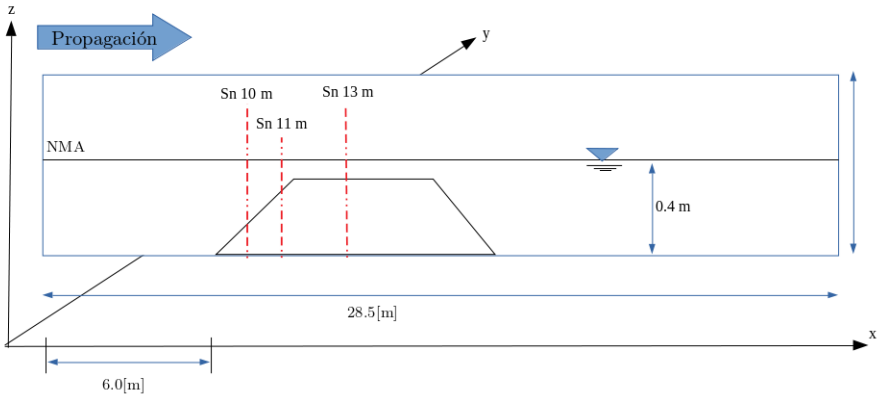
Para conocer la precisión con que OpenFOAM modela el comportamiento del oleaje en asomeramiento, se replicaron los principales casos del estudio desarrollados por Kamath et al. (2017), los que son definidos en la Tabla 3.13.

La configuración del dimensionamiento de las celdas para reproducir estos casos está dada por Arjona (2016) (caso CR01 de la Tabla 3.11), ya que los costos computacionales son notablemente menores . Los parámetros utilizados para modelar los principales casos presentados en Kamath et al. (2017) se proporcionan en la Tabla 3.13, y la disposición de los sensores en el canal modelado por OpenFOAM se presenta en la Figura 3.19.

Tabla 3.13: parámetros utilizados para modelar los casos de Kamath et al. (2017), donde H es la altura de ola regular a propagar, T es el periodo de onda y t es el tiempo simulado.

Caso	H [m]	T [s]	Teoría	t [s]
A01	0.022	2.5	Stokes II	60.0
A02	0.042	2.5	Stokes II	60.0

Figura 3.19: disposición de los principales sensores de desnivelación en el canal bidimensional modelado en OpenFOAM.

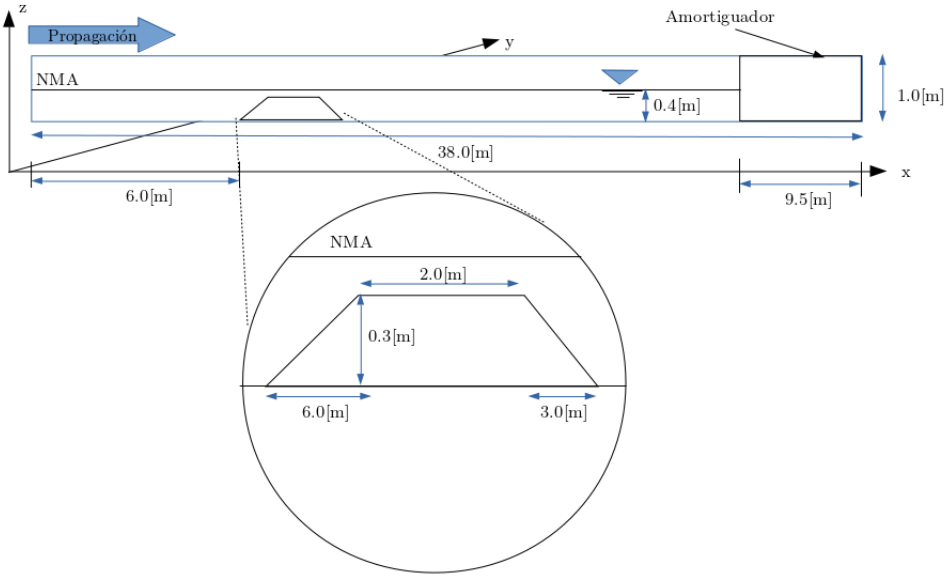


Fuente: elaboración propia.

Kamath et al. (2017) estudió el comportamiento del modelo REEF3D para la formación y rotura de las olas al propagarse a lo largo de un canal de ondas (Figura 3.20). Luego, los resultados del modelo fueron comparados con datos experimentales.

Según el análisis de refinamiento de la malla realizado en Kamath et al. (2017), fue necesario utilizar una resolución de $\delta x = \delta z = 0.5$ cm, simulación que duró 40 horas con 128 procesadores para $t = 60$ s. Kamath et al. (2017) modeló el canal de la Figura 3.20 pero sin cambio en la profundidad (sin el trapecio); de esta manera los datos de desnivelaciones se compararon con la teoría de Stokes II para calibrar el REEF3D.

Figura 3.20: configuración del canal de ondas utilizado por Kamath et al. (2017) en las simulaciones numéricas. NMA es el nivel medio del agua en el canal.



Fuente: elaboración propia.

Tabla 3.14: configuración del mallado para modelar asomeramiento en OpenFOAM.

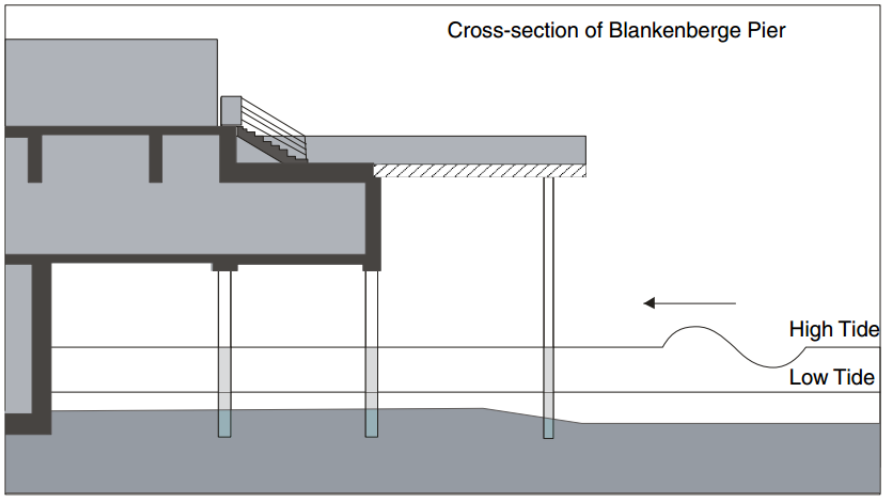
Caso	Tipo de rotura	Celdas en x	Celdas en z	δx [cm]	δz [cm]
A01	la ola no rompe	3703	227	0.54	0.22
A02	descrestamiento	1923	119	1.04	0.42

3.5.6. Comportamiento del oleaje en la interacción con estructuras

En la zona de rompiente es común que el oleaje interactúe con estructuras artificiales, tales como muelles o rompeolas. Debido a esto, en la presente sección se analiza la capacidad del modelo para representar el comportamiento del oleaje interactuando con un muelle.

Kisacik et al. (2012) analizó las cargas de oleaje que impactan sobre un muro vertical con una losa horizontal adyacente. Esto se realizó con un modelo a escala 1:20 del muelle de Blankenberge ubicado en Bélgica (Figura 3.21). Las variables en este estudio son la altura de ola inicial, el periodo, la profundidad y la geometría de la estructura. Las demás condiciones son todas constantes, además, solo se considera el uso de oleaje regular.

Figura 3.21: esquema de la sección del muelle de Blankenberge (Bélgica).



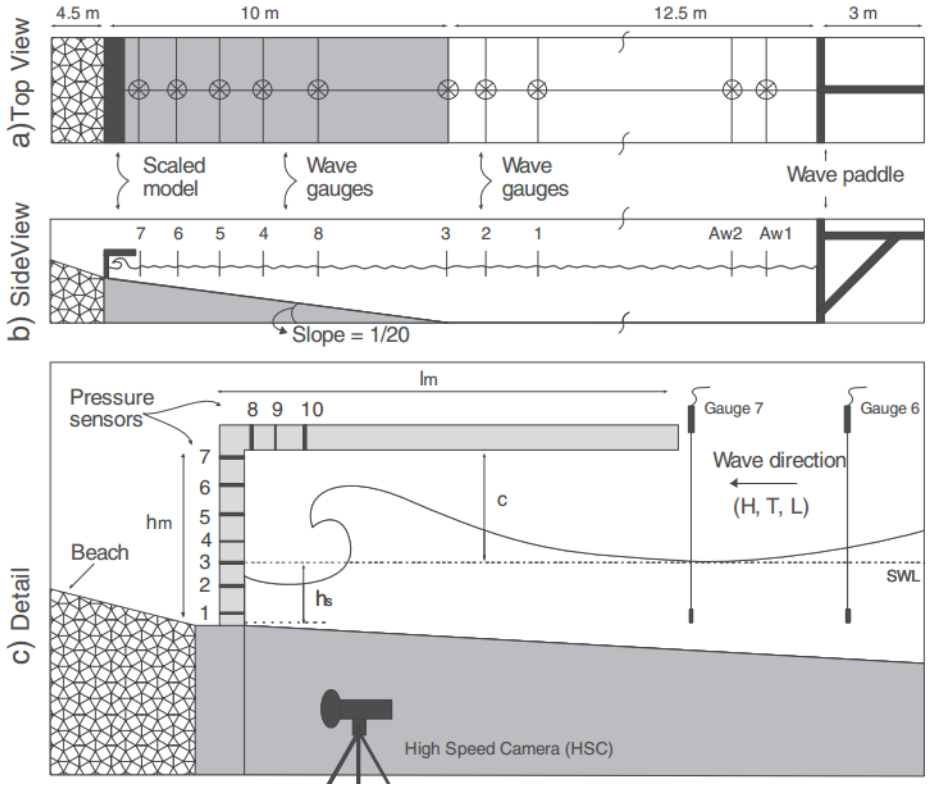
Fuente: Kisacik et al. (2012).

Las pruebas desarrolladas por (Kisacik et al., 2012) se realizaron en un canal de olas de 30 m x 1 m x 1.2 m en la Universidad de Ghent (Bélgica). El canal tiene una rampa con pendiente 1:20 que inicia a unos 12.5 m desde la zona de generación y llega hasta el muro vertical (Figura 3.22). En el muro vertical se pusieron dos corridas de 7 sensores de presión, y en el tablero del muelle se pusieron dos corridas de 3 sensores cada 1.5 cm (Figura 3.24). Aunque la precisión de los perfiles de presión dependen de la discretización, en el modelo físico no fue posible instalar sensores de presión a una distancia menor a 3 cm, por lo que fue necesario repetir un mismo caso varias veces cambiando la distribución de los sensores para poder generar un perfil de presiones de alta resolución.

En este trabajo se puso un total de 31 sensores en la parte vertical y 6 en la zona horizontal, ya que es fácil agregar una gran cantidad de sensores sin influir mayormente en el costo computacional. De esta manera se generaron curvas más suavizadas que representan el comportamiento de las presiones a lo largo de la estructura. Justo al inicio de la rampa, se colocó un sensor de desnivelaciones (sensor 3 de la Figura 3.22), el cual sirvió para conocer

la altura de ola en ese punto. Todos los casos modelados tomaron en cuenta la altura de ola medida en ese sensor (H_1) para generar los gráficos y realizar el análisis.

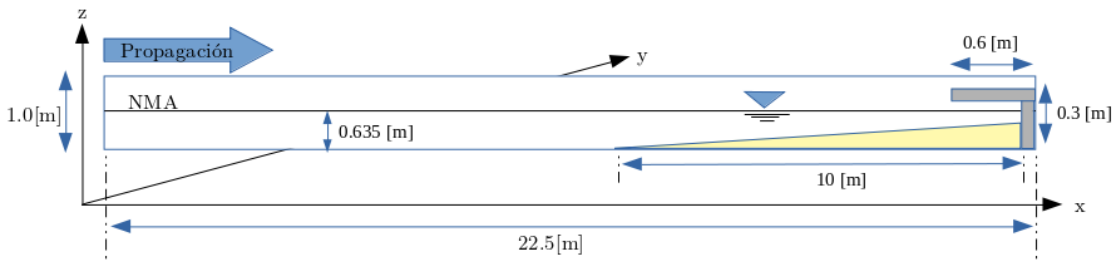
Figura 3.22: configuración experimental del caso de estudio.



Fuente: Kisacik et al. (2012).

El canal numérico desarrollado en OpenFOAM no consideró el amortiguador puesto en la pared opuesta a la propagación del canal, ni tampoco se agregó una distancia para la paleta generadora ya que se mantuvo activada propiedad de absorción activa en el borde opuesto al borde de generación, y por otra parte, la configuración utilizada en olaFlow no modeló una paleta generadora, por lo tanto ese espacio no fue necesario (Figura 3.23).

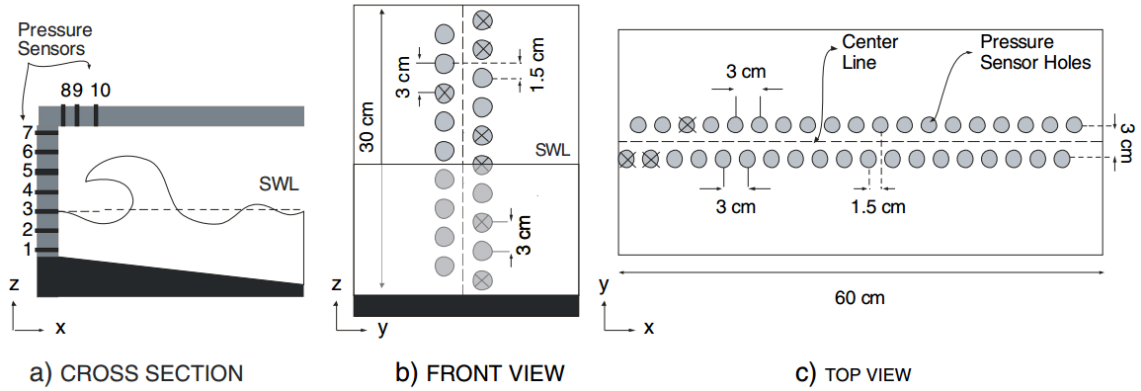
Figura 3.23: configuración utilizada en OpenFOAM para modelar la interacción del oleaje con una estructura.



Fuente: elaboración propia.

Se procesaron los datos obtenidos por OpenFOAM a partir de los parámetros presentados en la Tabla 3.15. En los casos modelados se mantiene un periodo (T) igual a 2.0 s, un h_s igual a 0.135, ya que con estas condiciones se presentaron las mayores presiones en el estudio realizado por Kisacik et al. (2012). El valor que se hizo variar fue (H_1), el cual

Figura 3.24: detalle de la distribución de los sensores de presión. SWL corresponde a la superficie libre.



Fuente: Kisacik et al. (2012).

corresponde a la altura de ola en el sensor ubicado justo al inicio de la rampa (sensor 3 en la Figura 3.24). En la Tabla 3.16 se presentan los valores utilizados H_1 en cada caso modelado.

Tabla 3.15: parámetros utilizados para para el análisis del comportamiento de las presiones; h_s corresponde a la distancia entre la superficie libre y la rampa, h_m es la longitud de la pared vertical y l_m representa la longitud horizontal de la losa.

h_s	teoría	profundidad (m)	h_m (m)	l_m (m)
0.135	Stokes II	0.635	0.3	0.6

Tabla 3.16: valores tomados por H_1 para el caso analizado.

Caso	01	02	03	04	05	06	07	08	09
H_1 (m)	0.095	0.1	0.105	0.11	0.115	0.12	0.125	0.13	0.135

Procesamiento de datos

Para realizar las pruebas fue necesario generar las alturas de ola presentadas en la Tabla 3.16, justo al inicio de la rampa. Así, para calcular las alturas de ola en ese punto se agregó un sensor de desnivelaciones, del cual se esperaba que tuviera un comportamiento cercano a la teoría de Stokes II.

En un principio se corrió el caso con una altura de ola de $H_1 = 0.095$ m (Tabla 3.16) con el criterio utilizado por Arjona (2016) (Tabla 3.11), dando como resultado la serie de tiempo presentada en la Figura 3.25, lo que no representa correctamente el fenómeno físico. Es por esto que el mismo caso se volvió a correr dos veces más, pero aumentando la densidad de la malla, tal como aparece en la Tabla 3.17. Al aumentar la densidad de la malla el costo computacional también se incrementó notoriamente, lo que se ve reflejado en el tiempo de modelación (Tabla 3.17).

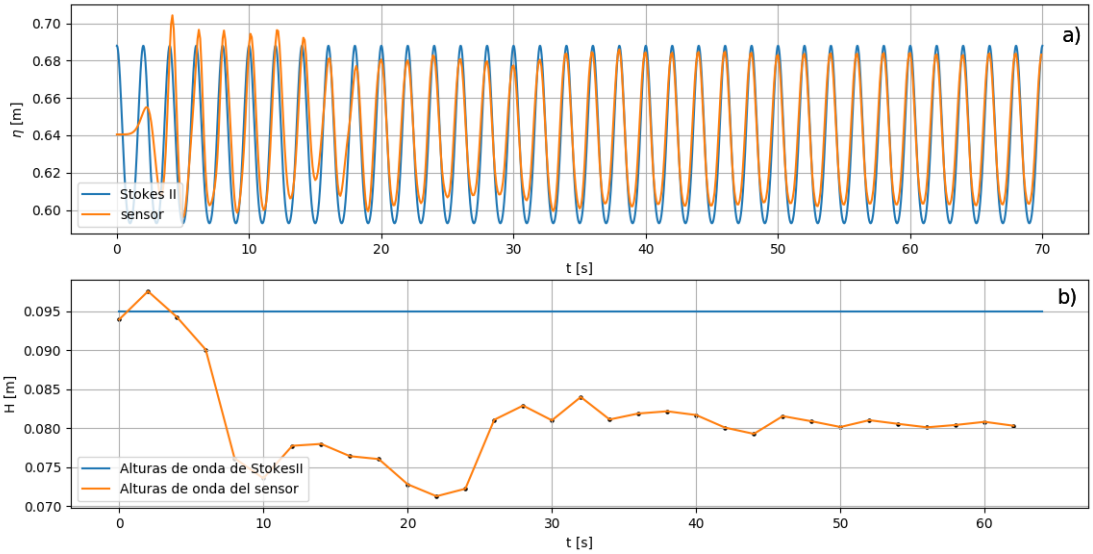
Una vez mejorada la precisión de la altura de ola en el sensor, se calcularon las presiones sobre el muelle siguiendo el diagrama de flujo de la Figura 3.28. El procedimiento para la obtención de los resultados se presenta en el Anexo E.2.

Tabla 3.17: configuración de la malla para $H_1 = 0.095$ m. Donde dx corresponde al ancho de cada celda y dz al alto, y el tiempo corresponde al tiempo real de modelación para cada caso.

Caso	Celdas en x	Celdas en z	dx (cm)	dz (cm)	Tiempo (horas)
P01	632	126	2.37	0.95	5.1
P02	902	252	1.185	0.475	21.525
P03	1805	505	0.5925	0.2375	189.52

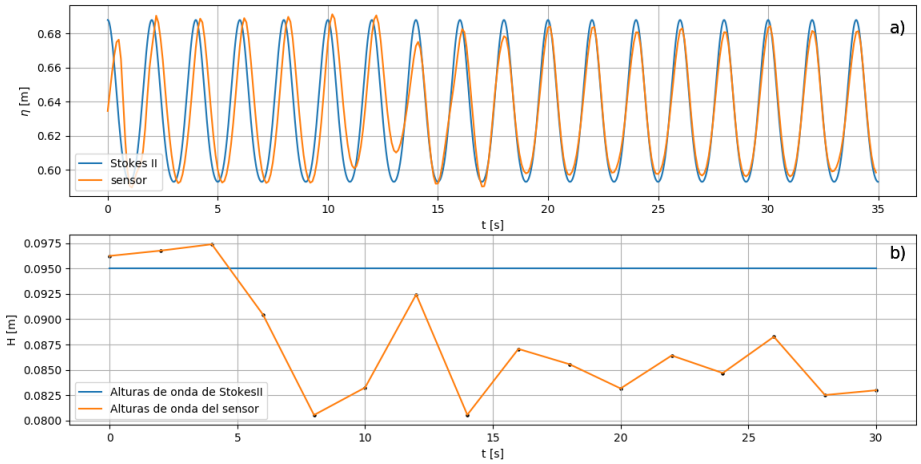
Fuente: elaboración propia.

Figura 3.25: caso P01 (Tabla 3.17), desnivelaciones en el tiempo para el sensor al inicio de la rampa (a), alturas de ola en el sensor(b).



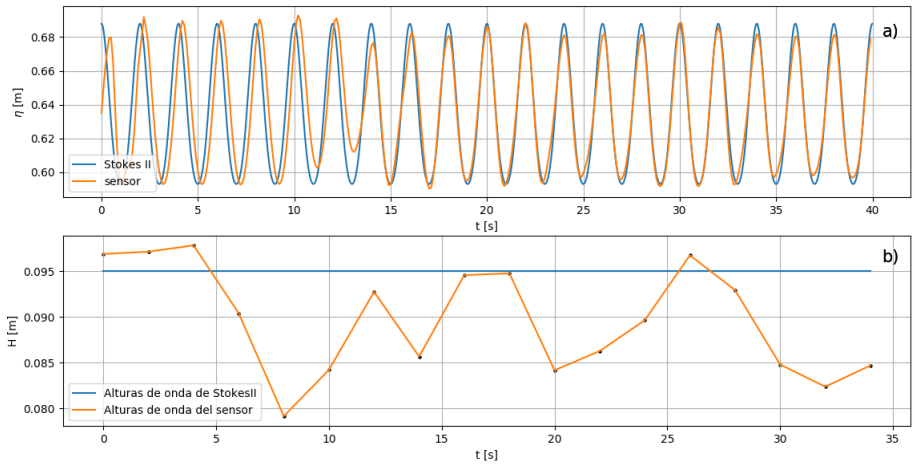
Fuente: elaboración propia.

Figura 3.26: caso P02 (Tabla 3.17), desnivelaciones en el tiempo para el sensor al inicio de la rampa (a), alturas de ola en el sensor(b).



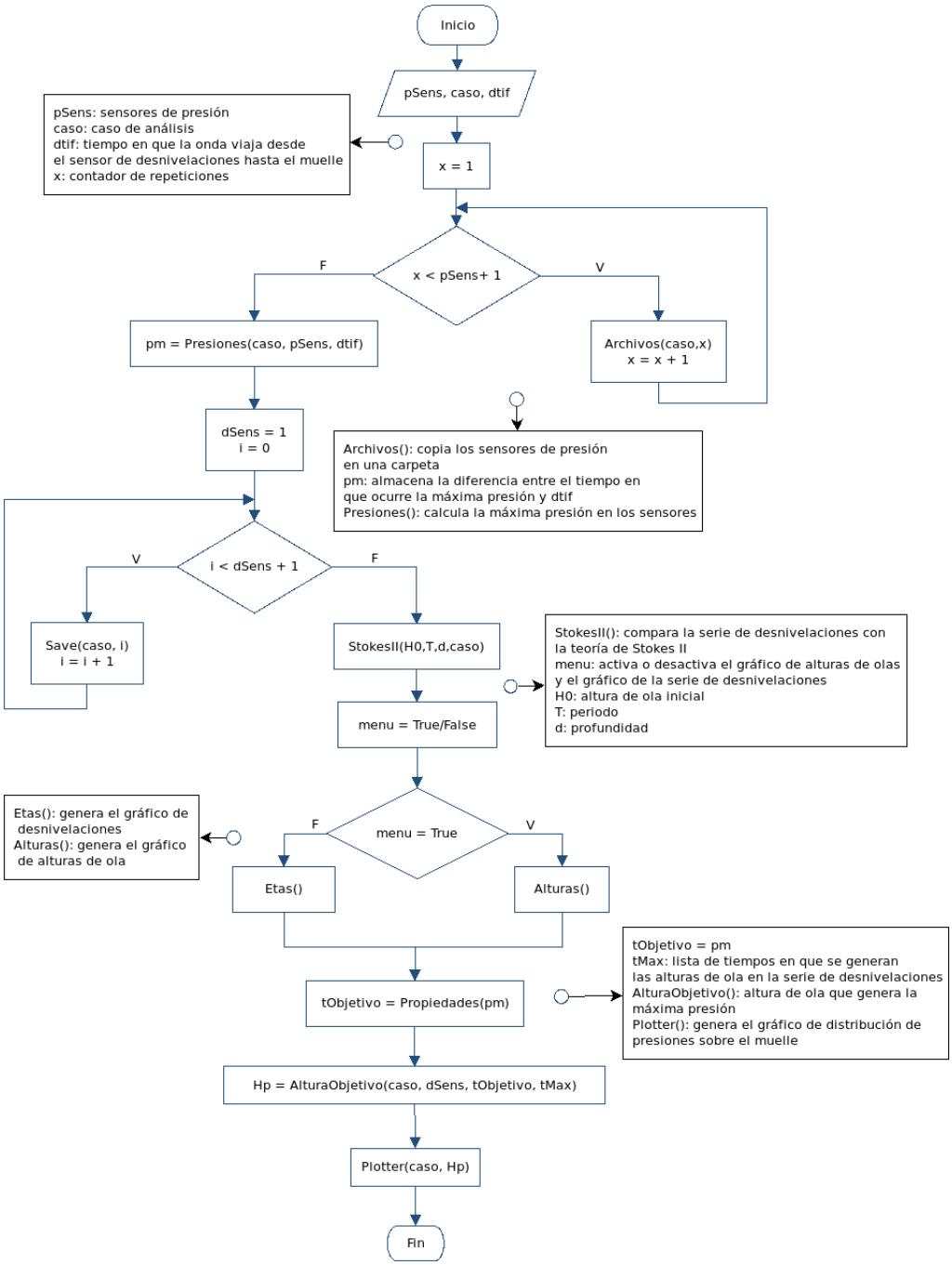
Fuente: elaboración propia.

Figura 3.27: caso P03 (Tabla 3.17), desnivelaciones en el tiempo para el sensor al inicio de la rampa (a), alturas de ola en el sensor(b).



Fuente: elaboración propia.

Figura 3.28: diagrama de flujo de los scripts que calculan las presiones sobre el muelle.



Fuente: elaboración propia.

Capítulo 4

Resultados

4.1. Onda progresiva (P-1-2)

El primer caso (CR01) se modeló con el criterio de Arjona (2016), luego los casos CR02 y CR03 se corrieron con el criterio propuesto por Larsen et al. (2018). El caso CR02 tiene 12.5 celdas por cada 10 cm (valor que corresponde a la altura de ola), y el caso CR03 tiene 25 celdas por cada 10 cm (Tabla 3.11).

Los tres casos se corrieron; sin un modelo de turbulencia (Figura 4.1 (a) y Figura 4.2 (a)), con el modelo de turbulencia $k - \epsilon$ (Figura 4.1 (b) y Figura 4.2 (b)), y con el modelo $k - \omega\, SST$ (Figura 4.1 (c) y Figura 4.2 (c)). Esto se hizo para para observar si hay cambios significativos en los resultados.

Los tres casos definidos sin modelo de turbulencia tuvieron un comportamiento diferente entre sí a lo largo del canal en la Figura 4.1 (a), pero solo los casos CR02 y CR03 presentaron un aumento en la altura significativa (H_s) a los 2 m en el eje x. Luego, a los 3.5 m el caso CR01 se alejó de la altura calculada por la teoría de Stokes II y los casos CR02 y CR03 se mantuvieron más cerca de la teoría. Cerca de $x = 4.6$ m, los casos CR02 y CR03 tuvieron valores por sobre Stokes II, y el caso CR01 tuvo una altura significativa bajo la teoría pero más cerca que los casos CR02 y CR03.

Se realizó una comparación de las desnivelaciones obtenidas en $x = 1.0$ m. Para esto se pusieron los tres casos junto a las desnivelaciones proporcionadas por la teoría de Stokes II, dando como resultado la Figura 4.2 (a). Si bien es cierto que los tres casos tuvieron un comportamiento similar a la teoría de Stokes II, desde los 7.0 s se hace evidente la diferencia entre ellos. La resolución para la malla planteada por Arjona (2016) tuvo un mejor desempeño que los casos CR02 y CR03 con respecto a la teoría.

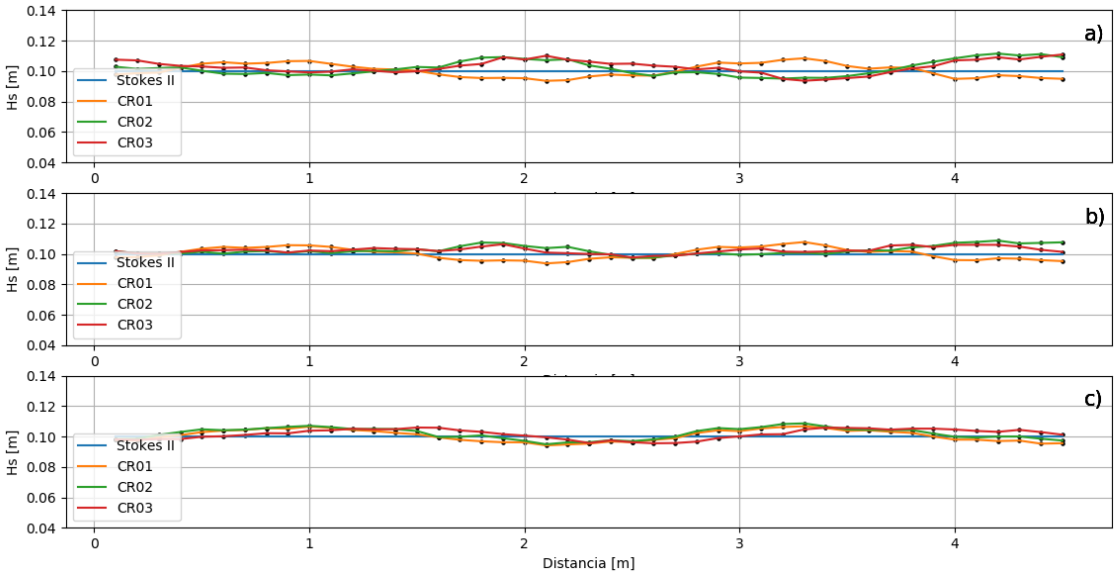
En una segunda prueba se activó el modelo $k - \epsilon$ y se volvieron a correr los tres casos con todos sus demás parámetros constantes, dando como resultado la Figura 4.1 (b) y la Figura 4.2 (b). Al comparar las imágenes a y b de la Figura 4.1 es posible notar algunas diferencias; todas las curvas de (b) tuvieron un mejor comportamiento con respecto a la teoría de Stokes II en comparación con las curvas de (a).

Al observar las imágenes (a) y (b) en la Figura 4.2 nuevamente se observa un leve mejor comportamiento en las curvas (b); aunque en ambas figuras desde los 7.5 s las ondas comenzaron a disminuir su amplitud alejándose de la teoría, a excepción del caso CR01 que se ajustó bien a la teoría durante todo el tiempo de simulación (Figura 4.2 a,b). Por

otra parte, en la Figura 4.2 (b) se observa que luego de los 10 s las curvas de los casos CR02 y CR03 comenzaron a acercarse a la teoría de Stokes II hasta los 20 s.

En una tercera prueba (imágenes (c) de las Figuras 4.1 y 4.2) se utilizó el modelo $k-\omega\, SST$, dando resultados aún más cercanos a la teoría de Stokes II que en los casos anteriores.

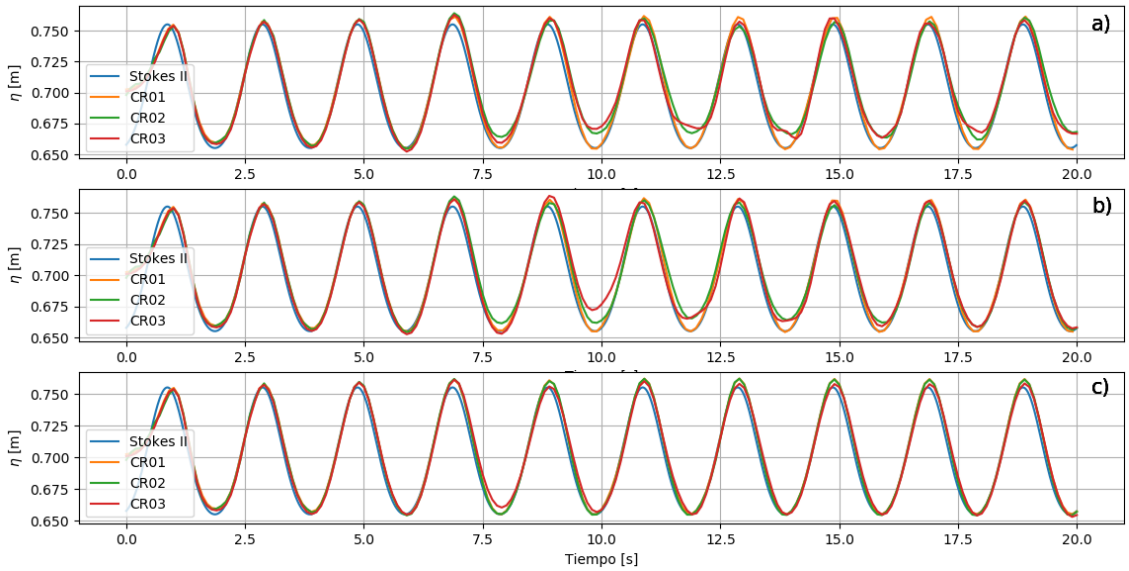
Figura 4.1: comparación de los casos estudiados con la teoría de Stokes II; los puntos negros representan sensores puestos a lo largo del canal cada 0.1 m. La simulación sin modelo de turbulencia se presenta en la gráfica (a), en la gráfica (b) se presenta el modelo $k-\epsilon$, y finalmente en la gráfica de la letra (c) se presenta el modelo $k-\omega\, SST$. El mallado del caso CR01 está definido por el criterio de Arjona (2016), mientras que el mallado de los casos CR02 y CR03 está definido por el criterio de Larsen et al. (2018), tal como se indica en la Tabla 3.11.



Fuente: elaboración propia.

En un principio, al hacer variar la densidad del mallado, no se generó una mejora significativa en los resultados. Sin embargo, se debe considerar el gasto de tiempo que lleva correr cada uno de los tres casos; para el primer criterio (CR01) el modelo se demoró hasta un cuarto menos que CR02 y CR03 para entregar resultados en las mismas condiciones. Por esta razón el criterio de Arjona (2016) es utilizado para definir la densidad del mallado de los casos en secciones posteriores. Por otra parte, existe una clara mejora en los resultados al agregar el modelo $k-\omega$ SST.

Figura 4.2: comparación de los casos modelados con la teoría de Stokes de segundo orden para oleaje progresivo a los $x = 1.0$ m (sin modelo de turbulencia (a), modelo $k - \epsilon$ (b), y (c) modelo $k - \omega\, SST$). En el caso CR01, el mallado se definió mediante el criterio de Arjona (2016), mientras que en los casos CR02 y CR03, el mallado se definió mediante el criterio de Larsen et al. (2018).



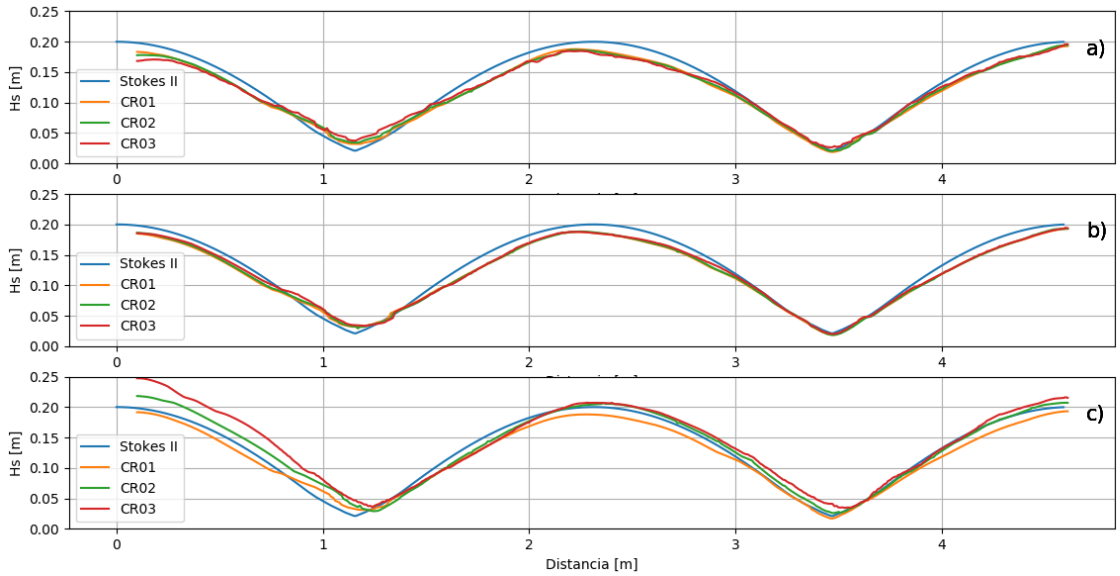
Fuente: elaboración propia.

4.2. Onda estacionaria (E-1-2)

Las pruebas (a) y (b) de la Figura 4.3 tuvieron un comportamiento muy similar. Sin embargo, es evidente que cuando el modelo de turbulencia $k-\epsilon$ se activó generó un comportamiento más suavizado en las curvas en la Figura 4.3 (b).

En la prueba (c) de la Figura 4.3 se observan resultados más sensibles a la variación de la densidad del mallado; los tres casos de la prueba (c) no tuvieron un comportamiento similar entre ellos, sobre todo en la zona de generación de oleaje. Las desnivelaciones para los modelos de turbulencia se presentan en la Figura 4.4.

Figura 4.3: altura de onda estacionaria para los tres casos analizados, comparados con la teoría de Stokes II (simulación sin modelo de turbulencia (a), modelo $k-\epsilon$ activado (b), y modelo $k-\omega\,SST$ (c)). En el caso CR01 se utilizó el criterio de Arjona (2016) para la definición del mallado, y en los casos CR02 y CR03 se utilizó el criterio de Larsen et al. (2018), tal como se observa en la Tabla 3.11.

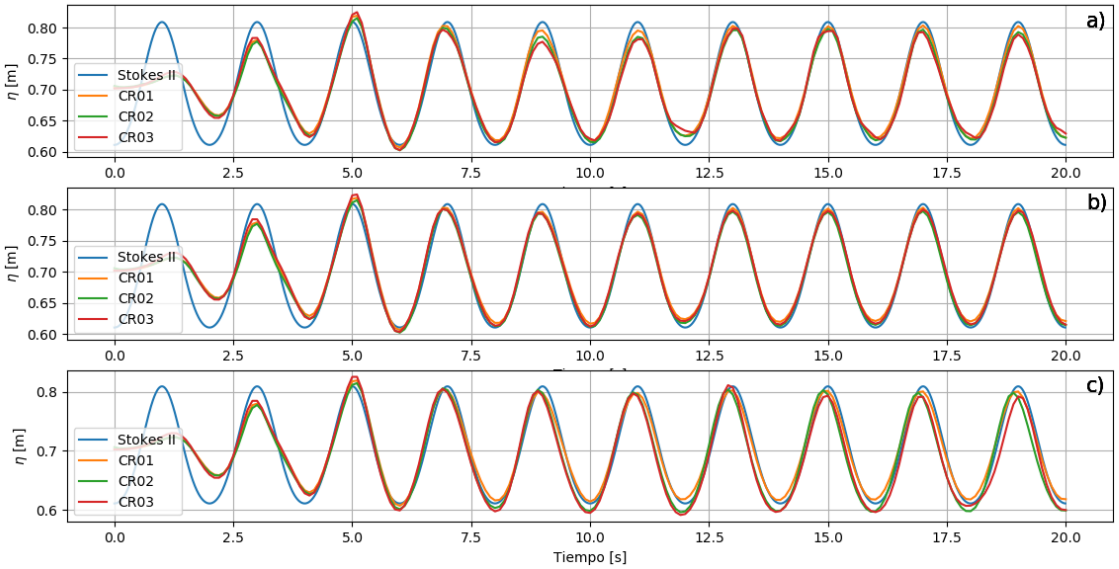


Fuente: elaboración propia.

A partir de los 5.0 s se observó un aumento de la amplitud de la onda en todas las pruebas producto de la superposición de las dos ondas progresivas con distinto sentido (Figura 4.4). Antes de este tiempo no hubo superposición debido a que la onda reflejada en la pared aún no había vuelto hasta esa posición. Algo similar ocurrió en la Figura 4.5, donde se muestra que a partir de los 5.0 s los datos comenzaron a parecerse a la teoría. No obstante, los mejores resultados fueron proporcionados para los casos en que el modelo de turbulencia $k-\epsilon$ estaba activado, ya que generó un comportamiento más suavizado y cercano a la teoría en todas las curvas. Por contraparte, a medida que se aumentaba la resolución de la malla (casos CR02 y CR03) los resultados estaban más lejos de la teoría en el modelo $k-\omega\,SST$.

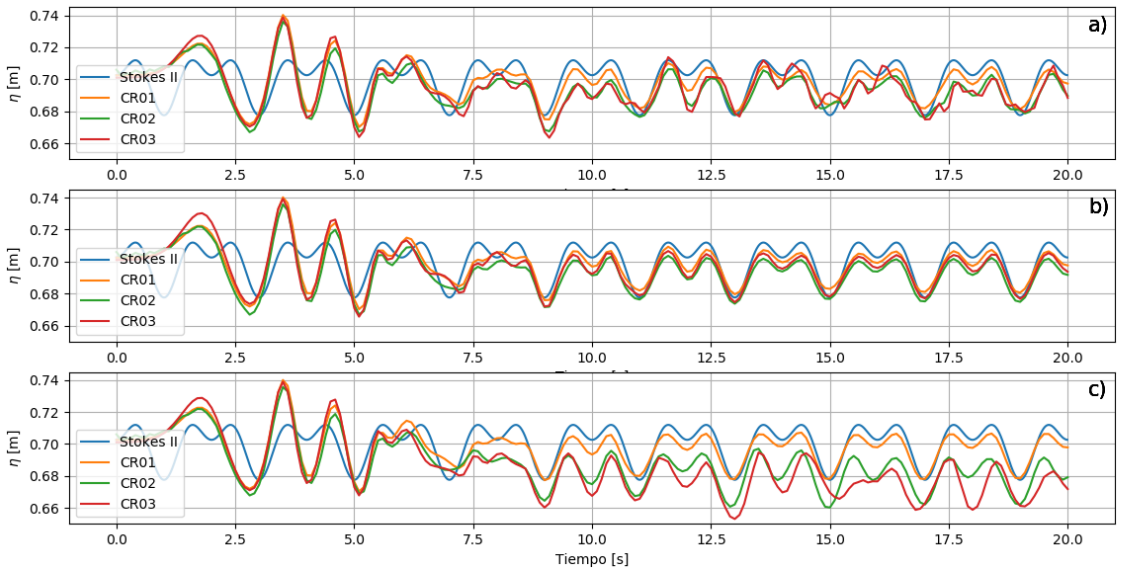
Al aumentar la resolución de la malla (casos CR02 y CR03), el rendimiento del modelo $k-\omega$ SST fue más bajo en comparación con el modelo $k-\epsilon$ en oleaje estacionario. Por otra parte, cuando se generó oleaje progresivo, el modelo $k-\omega$ SST entregó resultados más cercanos a la teoría de Stokes II que el modelo $k-\epsilon$.

Figura 4.4: desnivelaciones en el tiempo para los casos de estudio, comparadas con la teoría de Stokes II en oleaje estacionario, $x = 2.31$ m (simulación sin modelo de turbulencia (a), modelo $k - \epsilon$ (b), y modelo $k - \omega\, SST$ (c)).



Fuente: elaboración propia.

Figura 4.5: desnivelaciones en el tiempo para los casos de estudio, comparadas con la teoría de Stokes II en oleaje estacionario, $x = 3.46$ m (simulación sin modelo de turbulencia (a), modelo $k - \epsilon$ activado (b), y modelo $k - \omega\, SST$ (c)).

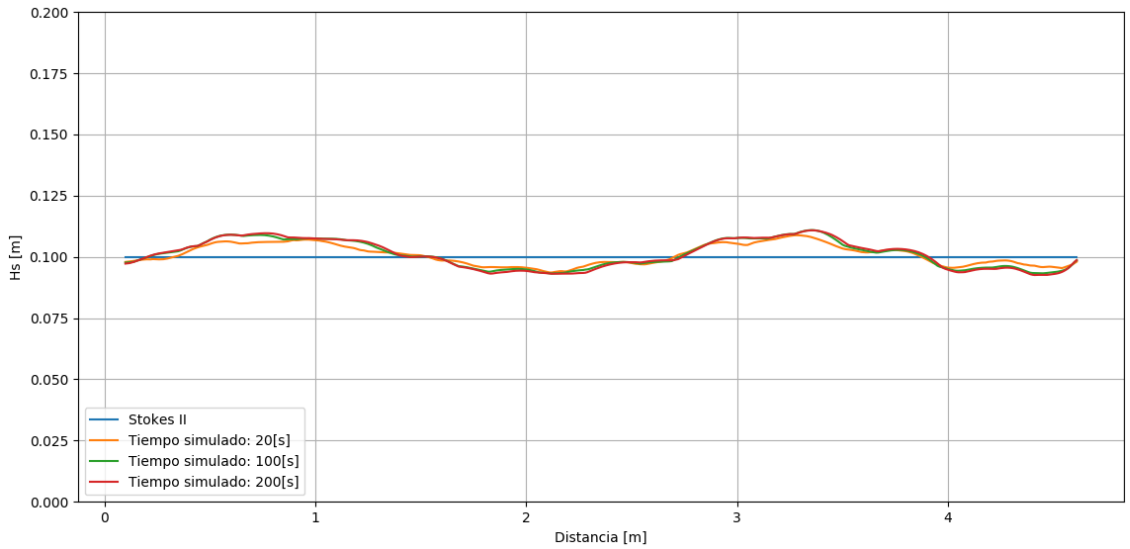


Fuente: elaboración propia.

4.3. Resolución temporal (P-4)

Según los resultados proporcionados por el modelo en oleaje progresivo, la diferencia entre 20 y 100 s para el caso CR01 fue despreciable, y fue menor entre 100 y 200 s, además, todas las curvas convergieron en un comportamiento oscilatorio en torno a los datos de Stokes II (Figura 4.6).

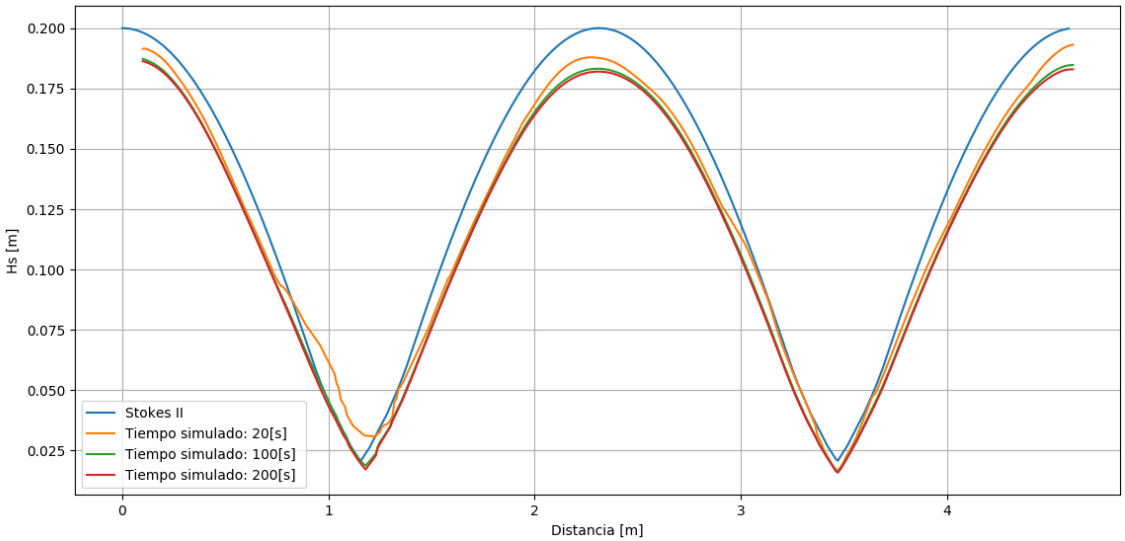
Figura 4.6: altura significativa del caso CR01 al aumentar el tiempo de simulación en 20, 100 y 200 s para oleaje progresivo.



Fuente: elaboración propia.

En la Fgura 4.7 se observa un comportamiento más suavizado en las curvas 100 y 200 s que en la de 20 s, por lo que al aumentar el tiempo de simulación mejoraron los resultados con respecto a la teoría.

Figura 4.7: Altura significativa del caso CR01 al aumentar el tiempo de simulación en 20, 100 y 200 s para oleaje estacionario.

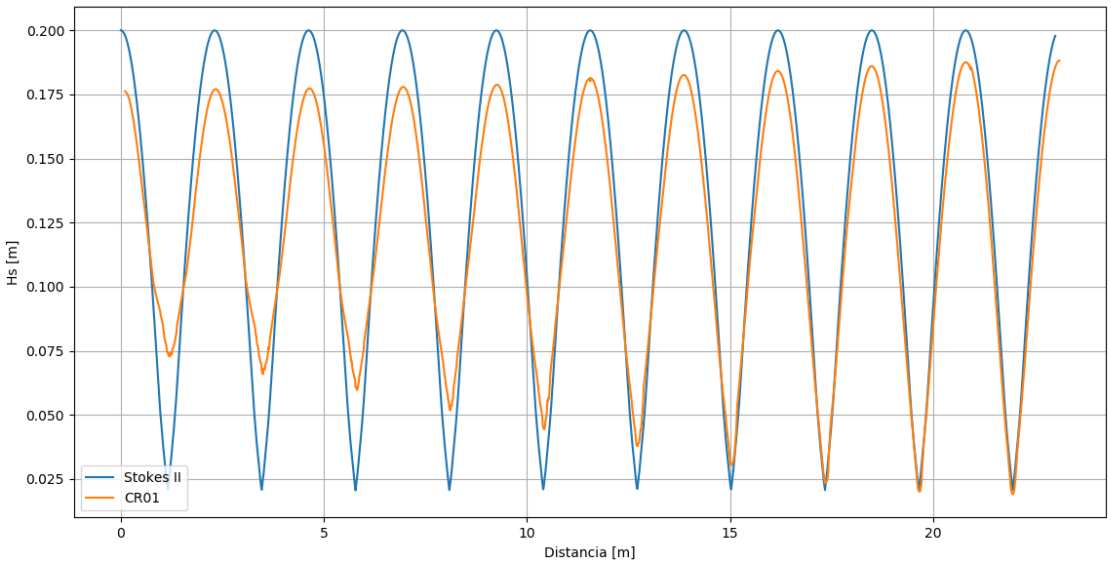


Fuente: elaboración propia.

4.4. Análisis de sensibilidad del dominio (E-3)

En la zona de generación de oleaje no es posible desactivar la absorción activa, ya que según Higuera (2015) se generan problemas de estabilidad en el modelo. Es por esto que los resultados de la Figura 4.8 muestran que a mayor distancia del borde de generación ($x = 0$), la altura significativa converge la teoría de Stokes II.

Figura 4.8: comparación entre el caso CR01 y la teoría de Stokes II para un canal de 23.12 m de largo y un oleaje estacionario.

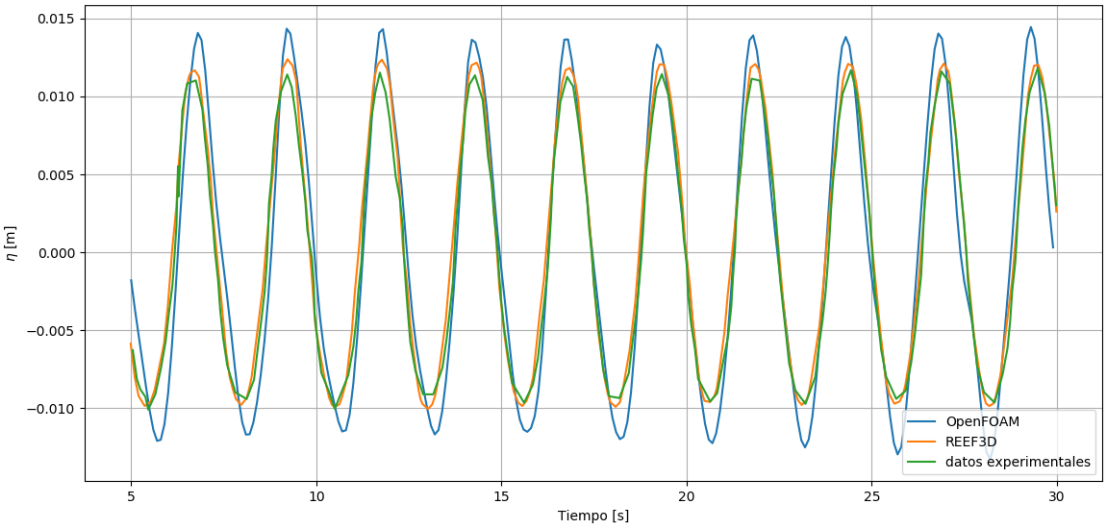


Fuente: elaboración propia.

4.5. Análisis del comportamiento de OpenFOAM en asomeramiento (A)

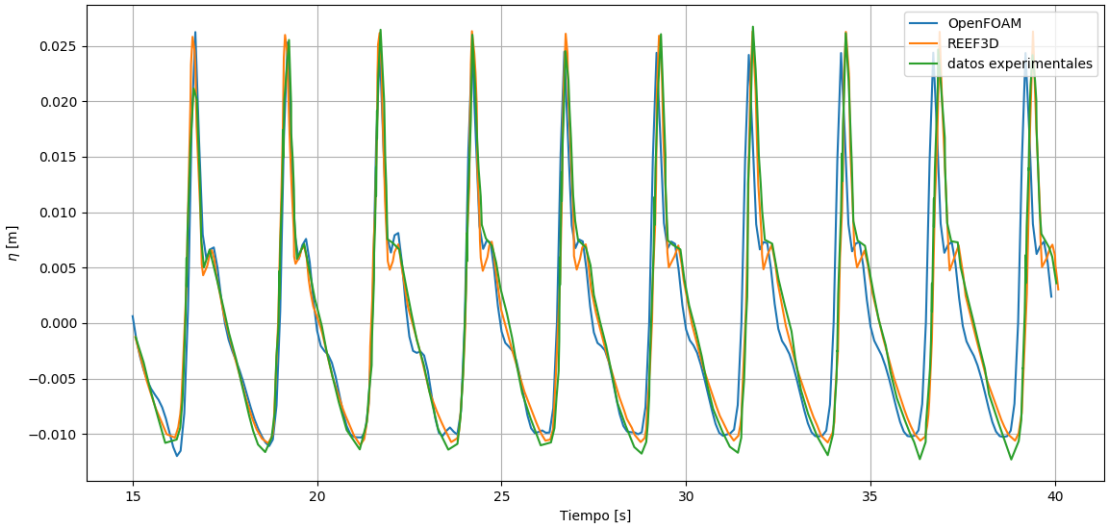
En el caso A01 de la Tabla 3.14 (sección 3.5.5) no se generó rotura debido a la configuración de los parámetros de onda presentados en la Tabla 3.13 (sección 3.5.5) y la profundidad del canal. Las variaciones de la superficie libre desarrolladas por OpenFOAM, en un principio presentaron una sobreestimación de las crestas y valles con respecto a los resultados del modelo REEF3D y los datos experimentales (Figura 4.9). Sin embargo, a medida que la onda avanzaba a lo largo del canal, se generó una mejora sustancial en los resultados de OpenFOAM con respecto a los datos experimentales (Figura 4.10 y anexo B).

Figura 4.9: desnivelaciones en el canal a los 10 m, caso A01 (Tabla 3.14).



Fuente: elaboración propia.

Figura 4.10: desnivelaciones en el canal a los 13 m, caso A01 (Tabla 3.14).

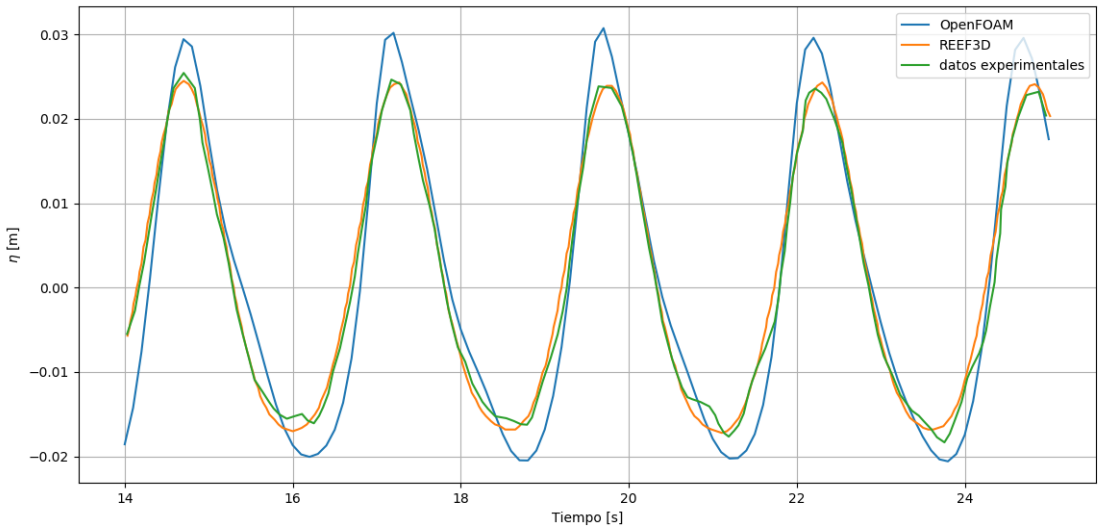


Fuente: elaboración propia.

En el caso 02 (Tabla 3.14), el comportamiento de OpenFOAM fue similar al caso 01.

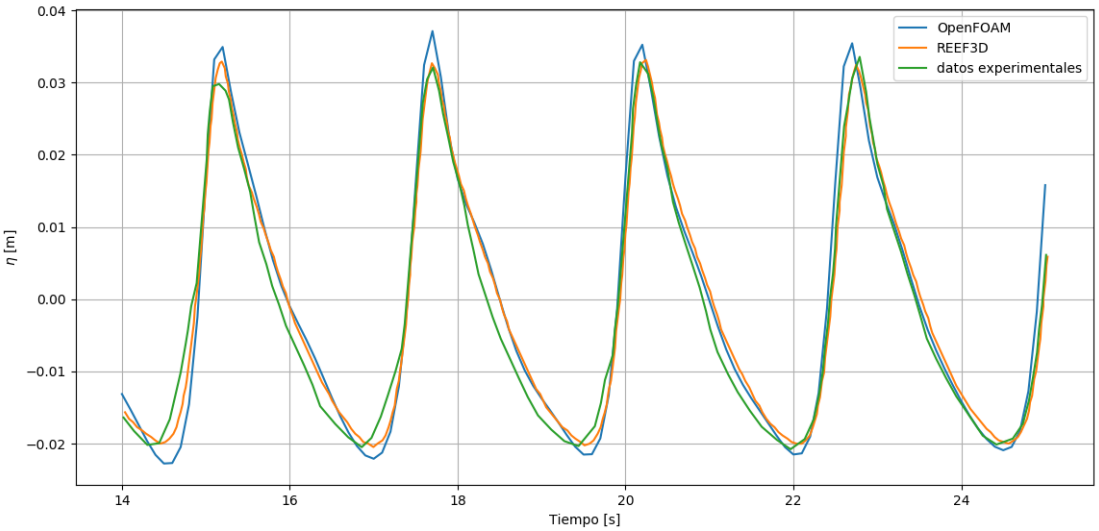
En un principio, las desnivelaciones presentadas a los 10 m (Figura 4.11) distaban de los datos teóricos y del REEF3D, pero desde los 11 a los 17 m los datos presentados por OpenFOAM describieron adecuadamente el comportamiento del oleaje con respecto a los datos experimentales (Figura 4.12 y anexo B).

Figura 4.11: desnivelaciones en el canal a los 10 m, caso A02 (Tabla 3.14).



Fuente: elaboración propia.

Figura 4.12: desnivelaciones en el canal a los 11 m, caso A02 (Tabla 3.14).



Fuente: elaboración propia.

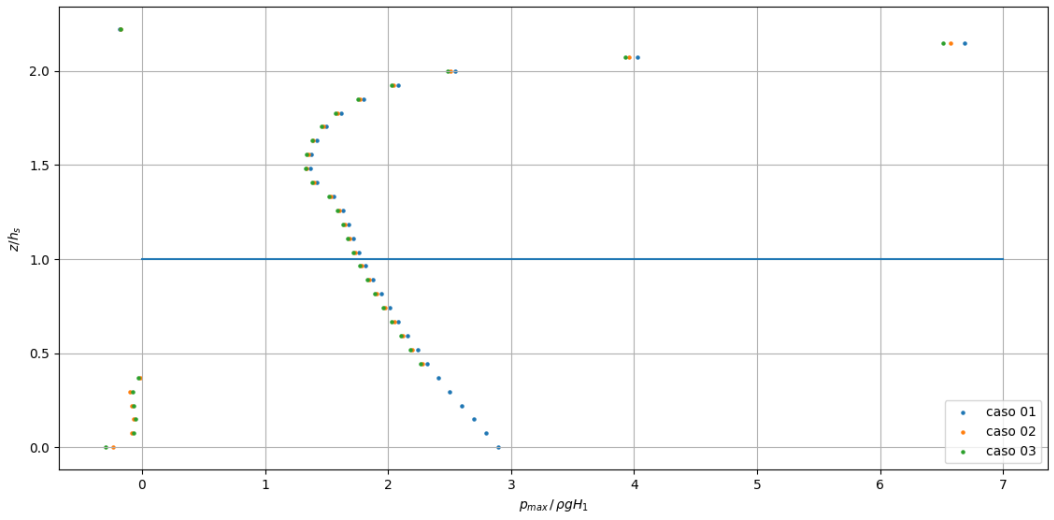
4.6. Interacción del oleaje con una estructura (M-1)

Luego de haber realizado el análisis de sensibilidad a la malla (Tabla 3.17 de la sección 3.5), y obtener los resultados presentados en las Figuras 3.25, 3.26 y 3.27, se observó un notorio aumento en la precisión de las desnivelaciones obtenidas por el sensor a medida que aumentaba la densidad de las celdas. Sin embargo, las presiones sobre el muelle para los tres casos analizados (Figura 4.13) no presentaron una variación significativa.

Se corrieron todos los casos presentados en la Tabla 3.16 con la configuración inicial de (Arjona, 2016), es decir, con la configuración del caso 01 presentada en la Tabla 3.17, dando como resultado la gráfica (b) de la Figura 4.14.

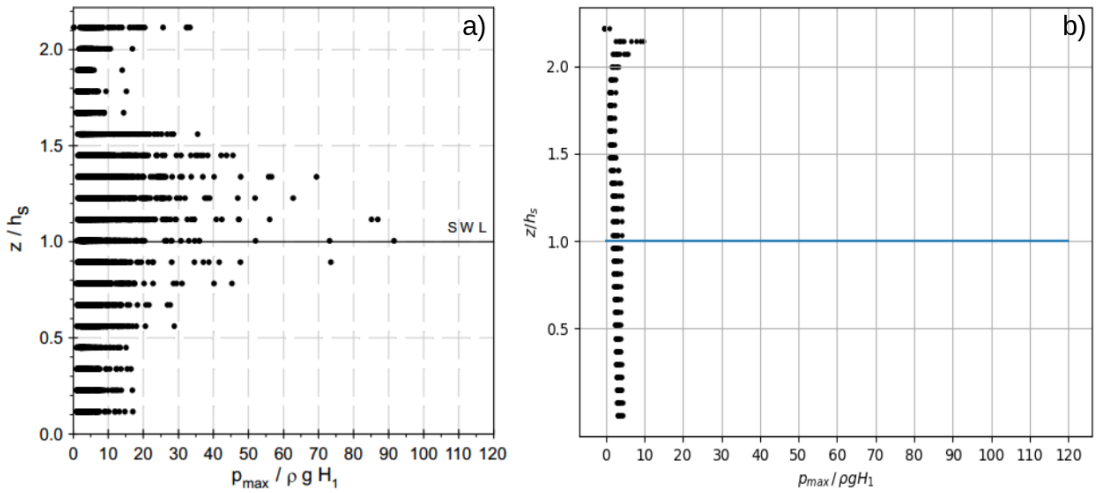
Al comparar los datos obtenidos con OpenFOAM (b) y el modelo físico (a) (Figura 4.14), se observó que openFOAM no captaba los peaks que obtuvo Kisacik et al. (2012). De hecho, el perfil de máximas presiones fue de una intensidad mucho menor que el obtenido en ese estudio. Se debe considerar que los peaks de presiones se generan en un intervalo de tiempo muy corto, lo que obligó a Kisacik et al. (2012) a utilizar una frecuencia de muestreo de 20 KHz en los sensores de presión, mientras que en OpenFOAM la frecuencia de muestreo fue solo de 10 Hz por limitaciones de equipo computacional y despacio en el disco duro. Lo que sí se pudo observar en la Figura 4.14 (b) es un aumento en la intensidad de las máximas presiones justo en la esquina entre el muro vertical y la losa del muelle, datos que también presentaron las pruebas de Kisacik et al. (2012).

Figura 4.13: comparación de las presiones en la vertical del muelle para los tres casos analizados (Tabla 3.17), con $H_0 = 0.095$ m.



Fuente: elaboración propia.

Figura 4.14: resultados obtenidos por el artículo en base a datos experimentales para la parte vertical del muelle (Kisacik et al., 2012) (a), versus resultados obtenidos por OpenFOAM con las dimensiones propuestas por Arjona (2016) para cada celda del dominio (b). SWL corresponde a la superficie libre.



Fuente: elaboración propia.

Capítulo 5

Conclusiones y futuros trabajos

5.1. Conclusiones

Principales parámetros configurados en OpenFOAM-olaFlow

A lo largo del desarrollo de este estudio se presentaron los principales parámetros y su rango de aplicabilidad para obtener resultados adecuados al utilizar OpenFOAM-olaFlow, a su vez, se mostraron las propiedades que se mantuvieron constantes por sugerencia de otro artículo científico o por las instrucciones de los desarrolladores del modelo.

- Una importante propiedad que se mantuvo constante fue el número de Courant, que según Larsen et al. (2018), el valor óptimo es 0.15 para modelar unos 20 periodos onda, considerando una resolución espacial de 12.5 celdas por altura de ola. De esta manera la altura de onda generada permanece constante durante la modelación. Para los casos en que se consideró la resolución espacial de Arjona (2016), el número de Courant fue adecuado también, ya que se consideraron 10 celdas por altura de ola en el plano vertical, y unas 25 celdas por altura de ola en el plano horizontal, o sea, con una menor resolución espacial con respecto al criterio propuesto por Larsen et al. (2018).
- La generación de oleaje con el solver olaFlow es bastante sencilla y se obtienen resultados muy similares a la teoría, sin embargo, es importante señalar que en la zona de generación, la absorción activa no permite generar correctamente oleaje estacionario. Por otra parte, la absorción activa que se agrega en la pared opuesta a la zona de generación solo funciona bien para oleaje lineal en aguas someras (Higuera, 2017). Este punto se debe tomar en cuenta ya que es una limitante que puede condicionar la generación ruido en los datos tomados, por lo que es necesario descartar los datos espurios cercanos al borde.
- Si bien Higuera (2015) dedujo las ecuaciones VARANS para modelar la interacción de un fluido con estructuras porosas, y además consideró los términos que definen la variación temporal de la porosidad; no desarrolló un componente en olaFlow que pudiera simular el transporte de sedimentos, ya que esto fue pensado para futuras lineas de trabajo. Este componente es clave para estudiar una gran cantidad de fenómenos que se producen de manera recurrente en zonas costeras; como la variación del perfil de una playa con respecto a las propiedades del tren de ondas propagado a la zona de rompiente, la influencia de marejadas en el ciclo de pérdida y recuperación

de sedimentos en una playa expuesta, o la socavación en muros verticales. Por lo que esta es una de las limitaciones al considerar el uso de OpenFOAM-olaFlow en ingeniería costera.

Procesamiento de datos en Python

El desarrollo de scripts para la automatización de procesos debe ser sencillo y con una suave curva de aprendizaje. Python cumple muy bien ambos requisitos, por lo que la manipulación y procesamiento de datos es muy fácil una vez aprendido a utilizar este lenguaje de programación. Además, Python no tiene restricciones de uso por lo que no es necesario pagar una licencia. Todas las anteriores características fomentan la reducción de costos en el desarrollo de proyectos.

El modelo OpenFOAM, luego del postproceso, proporcionó series con las variaciones de la superficie libre versus la variable tiempo. O sea que por cada sensor agregado al dominio se generó un archivo de este tipo. De ahí la importancia de recurrir a Python para procesar los datos, ya que en algunos casos se agregaron 2302 sensores. Una vez desarrollados los scripts todo el procesamiento no duró más de uno o dos minutos por caso.

Análisis de sensibilidad de la malla

Como parte de este trabajo se comparó el criterio de Arjona (2016) con los criterios propuestos por Larsen et al. (2018) para el dimensionamiento del mallado. Aunque Arjona (2016) utilizó otro modelo numérico (IH2VOF) para simular un canal numérico, este tiene algunas similitudes con respecto a la configuración OpenFOAM-olaFlow utilizada en este estudio; como el hecho de que también utiliza las VARANS o que modela canales bidimensionales. Por otra parte, los criterios propuestos por Larsen et al. (2018) fueron aplicados a OpenFOAM-interFoam, y ya que olaFlow es una modificación de las VARANS de interFoam, entonces estos requisitos de dimensionamiento de la malla pueden ser aplicables en olaFlow.

El criterio propuesto por Arjona (2016) (CR01) es más permisivo en términos de la calidad del mallado que los criterios de Larsen et al. (2018) (CR02 y CR03), pero obtiene resultados parecidos a CR02 y CR03 para oleaje progresivo, por lo que OpenFOAM se tarda bastante menos tiempo en realizar la simulación del mismo caso en comparación con CR02 y CR03.

Cuando se modeló el caso de oleaje estacionario, no se presentaron grandes diferencias entre los tres criterios para el mallado, sino más bien, la variación entre estos estuvo relacionada a la elección del modelo de turbulencia.

Estabilidad temporal

Una vez establecido que el criterio de Arjona (2016) presenta resultados más competitivos con respecto al costo computacional, se analizó el comportamiento del modelo para oleaje progresivo y estacionario considerando el tiempo de modelación como variable. El aumento del tiempo de simulación de 20 a 100 y a 200 s no generó mejoras significativas en los resultados del modelo para oleaje progresivo, por lo que se alcanzó una estabilidad temporal a los 20 s, o sea, al pasar unas diez ondas por el dominio. Por otra parte, para oleaje estacionario la curva generada con 20 s de modelación si presentó una diferencia

con respecto a las curvas de 100 y 200, por lo que para esta configuración fue insuficiente el tiempo mínimo de modelación sugerido por los desarrolladores de OpenFOAM para alcanzar estabilidad temporal.

Influencia de los límites del dominio en los resultados

Tanto para oleaje progresivo como estacionario, se observó una notoria diferencia de los tres criterios para el mallado con respecto a la teoría en los bordes de generación y absorción de oleaje, lo que era de esperarse, ya que para este modelo se debe considerar una distancia prudente en la toma de datos con respecto a los bordes para evitar datos espurios (Higuera, 2015).

Cuando se aumentó la longitud del dominio y se simuló un oleaje estacionario, los bordes influyeron en los datos cercanos a estos, pero principalmente esto se observó en la zona de generación debido a que no es posible desactivar la absorción activa en ese borde por inestabilidad del modelo. Luego, gradualmente los datos fueron acercándose a la teoría.

Rendimiento de los modelos de turbulencia

En este estudio se consideraron tres configuraciones con respecto al uso de modelos de turbulencia; para oleaje progresivo, el modelo $k-\omega\, SST$ presentó resultados más cercanos a la teoría que el modelo $k-\epsilon$ y que la simulación laminar. Sin embargo, para oleaje estacionario el modelo $k-\epsilon$ presentó mejores resultados. En ningún caso los resultados propuestos por la simulación laminar fueron más precisos que cuando se usó algún modelo de turbulencia.

Modelación de un canal de ondas bidimensional con OpenFOAM

Una vez obtenidos los principales parámetros para la modelación de un canal bidimensional; resolución de la malla, número de Courant adecuado, tiempo mínimo de simulación con respecto a la longitud del dominio, comportamiento de los modelos de turbulencia, entre otros factores, el modelo se comparó con dos artículos científicos.

Los resultados de OpenFOAM se compararon con el artículo desarrollado por Kamath et al. (2017), donde se modelaron dos casos de oleaje progresivo con distinta altura de onda a propagar mediante el modelo REEF3D. Los datos mostraron que, aunque en ese artículo científico se utilizaron 128 procesadores, con una modelación que duró 40 horas, OpenFOAM obtuvo resultados competitivos considerando una resolución de mallado mucho más baja debido a las limitaciones de hardware.

Al comparar los resultados de OpenFOAM con el estudio realizado por Kisacik et al. (2012), donde el oleaje interactúa con una estructura, OpenFOAM obtuvo un bajo rendimiento. Muy probablemente esto se debió a la frecuencia de muestreo empleada en OpenFOAM (10 Hz), ya que en Kisacik et al. (2012) se utilizó una frecuencia de 20 Khz, un valor acorde con la duración de las fuerzas producidas sobre la estructura. Esta configuración hace que OpenFOAM genere un gran aumento en el espacio del disco duro, y además, el tiempo de modelación también aumenta en gran cantidad. Es por esto que la limitación de hardware al utilizar OpenFOAM nuevamente jugó un papel crucial en los resultados.

Finalmente, considerando todas las configuraciones de oleaje estudiadas, los supuestos y limitaciones utilizadas en este trabajo, fue posible obtener resultados bastante cercanos a la teoría cuando las limitaciones de hardware no eran muy influyentes, o sea, fue posible calibrar y validar un canal numérico bidimensional en el que se propagó oleaje, siempre y cuando las limitaciones computacionales no influyeran de manera significativa en la configuración de los casos modelados. Sin embargo, cuando fue necesario aumentar la resolución del mallado, disminuir los pasos de tiempo, o disminuir el número de Courant, el costo computacional impidió que el computador de escritorio utilizado en este trabajo pudiera resolver el caso analizado.

5.2. Futuros trabajos

Al inicio de cualquier estudio del comportamiento del oleaje o la modelación de un canal de ondas con OpenFOAM, se recomienda primero hacer algunas pruebas para ver si el modelo se apega a la teoría de ondas adecuada a los parámetros del caso de estudio. A continuación, se presentan los principales pasos para obtener resultados cercanos a los datos empíricos.

- Configuración inicial. Utilizar un caso de ejemplo presentado en los solvers del modelo.
- Análisis de sensibilidad a la malla para optimizar el uso de los recursos computacionales; luego comparar los resultados con la teoría.
- Se debe configurar el intervalo de escritura para que pueda captar fenómenos de corta duración, como las fuerzas impulsivas sobre un muelle.
- Se debe configurar la porosidad adecuada de los materiales insertos en el dominio.
- Se debe hacer un análisis de sensibilidad al tiempo de modelación para alcanzar la estabilidad temporal.

Para el estudio de las presiones producto de fuerzas de oleaje sobre un muelle, es necesario considerar una frecuencia de muestreo acorde con la duración de las fuerzas producidas, es decir, se debe realizar un análisis de sensibilidad al intervalo de escritura de datos en el modelo numérico en torno a los 20 KHz. Este es el valor que se presenta Kisacik et al. (2012) para obtener las presiones peaks sobre un muelle.

Bibliografía

Airy, G. (1845). On tides and waves: Encyclopaedia metropolitana, v. 5.

Ardhuin, F. (2018). *Ocean waves in geosciences*. Laboratory of Physical and Spatial Oceanography, France.

Arjona, S. (2016). Análisis funcional de un disipador de oleaje pasivo basado en medios porosos. Master's thesis, Universidad de Cantabria, Cantabria.

Castellanos, H. E., Collazos, C. A., Farfan, J. C., & Meléndez-Pertuz, F. (2017). Diseño y construcción de un canal hidráulico de pendiente variable. *Información tecnológica*, 28(6):103–114.

Challenger-Pérez, I., Díaz-Ricardo, Y., & Becerra-García, R. A. (2014). El lenguaje de programación python. *Ciencias Holguín*, 20(2).

Chen, L. (2015). *Modelling of marine renewable energy*. PhD thesis, University of Bath.

Chenari, B. (2014). Numerical modelling of regular wave propagation using openfoam. Master's thesis.

Chorin, A. J. (1968). Numerical solution of the navier-stokes equations. *Mathematics of computation*, 22(104):745–762.

Coleman, G. N. & Sandberg, R. D. (2010). A primer on direct numerical simulation of turbulence-methods, procedures and guidelines.

Collado, L., Collado Contreras, M., Rodríguez Malaver, E., & Patiño, L. (2007). Análisis numérico del comportamiento del aire en un sistema de distribución de aire acondicionado empleando los modelos de turbulencia ke, rng ke y el modelo de las tensiones de reynolds. *Ingeniare. Revista chilena de ingeniería*, 16(2):370–382.

Courant, R., Friedrich, K., & Lewy, H. (1967). On the Partial Difference Equations of Mathematical Physics. *IBM JOURNAL*, pages 215–234.

Davidson, J., Cathelain, M., Guillemet, L., Le Huec, T., & Ringwood, J. (2015). Implementation of an openfoam numerical wave tank for wave energy experiments. In *Proceedings of the 11th European wave and tidal energy conference*. European Wave and Tidal Energy Conference 2015.

de Villiers, E. (2006). *The potential of large eddy simulation for the modelling of wall bounded flows*. PhD thesis, University of London.

Dean, R. G. (1984). Water wave mechanics for engineers and scientists. *Advanced series on ocean engineering*, 2.

del Jesus, M. (2011). *Three-dimensional interaction of water waves with maritime structures*. PhD thesis, University of Cantabria.

Didier, E. & Neves, M. (2012). A semi-infinite numerical wave flume using smoothed particle hydrodynamics.

Dyson, J. (2018). *GPU accelerated linear system solvers for OpenFOAM and their application to sprays*. PhD thesis, Brunel University London.

Engelund, F. (1953). *On the laminar and turbulent flows of ground water through homogeneous sand*. Akad. for de Tekniske Videnskaber.

Fenton, J. D. (1999). The cnoidal theory of water waves. In *Developments in Offshore Engineering*, pages 55–100. Elsevier.

Ferrer, P. M., Causon, D., Qian, L., Mingham, C., & Ma, Z. (2016). Numerical simulation of wave slamming on a flap type oscillating wave energy device. In *The Twenty-sixth International Ocean and Polar Engineering Conference, Rhodes, Greece*, pages 65–71.

Finn, J. R. & Li, M. (2016). Regimes of sediment-turbulence interaction and guidelines for simulating the multiphase bottom boundary layer. *International Journal of Multiphase Flow*, 85:278–283.

García, F., Palacio, C., & García, U. (2009). Generación de mallas no estructuradas para la implementación de modelos numéricos. *Dyna*, 76(157):17–25.

González, F. (2000). Perspectivas de los tráficos marítimos y competitividad portuaria. *Revistas ICE*.

Greenshields, C. J. (2017). *User guide for openFOAM*. OpenFOAM Foundation Ltd, England.

Gómez, S. (2017). *Mallado y simulación CFD de automóvil*. Tesis de master, Escola Técnica Superior d'Enginyeria Industrial de Barcelona, Cantabria.

Henry, A., Schmitt, P., Whittaker, T., Rafiee, A., Dias, F., et al. (2013). The characteristics of wave impacts on an oscillating wave surge converter. In *The Twenty-third International Offshore and Polar Engineering Conference*. International Society of Offshore and Polar Engineers.

Herreras, N. & Izarra, J. (2013). *Two-Phase Pipe Flow Simulations with OpenFOAM*. PhD thesis, Master's thesis. Norwegian University of Science and Technology, Norway.

Higuera, P. (2015). *Aplicación de la dinámica de fluidos computacional a la acción del oleaje sobre estructuras*. Tesis doctoral, Universidad de Cantabria, Cantabria.

Higuera, P. (2017). olaflow: CFD for waves.

Higuera, P., Lara, J. L., & Losada, I. J. (2013a). Realistic wave generation and active wave absorption for navier–stokes models: Application to openfoam®. *Coastal Engineering*, 71:102–118.

Higuera, P., Lara, J. L., & Losada, I. J. (2013b). Simulating coastal engineering processes with openfoam®. *Coastal Engineering*, 71:119–134.

Higuera, P., Lara, J. L., & Losada, I. J. (2014a). Three-dimensional interaction of waves and porous coastal structures using openfoam®. part i: formulation and validation. *Coastal Engineering*, 83:243–258.

Higuera, P., Lara, J. L., & Losada, I. J. (2014b). Three-dimensional interaction of waves and porous coastal structures using openfoam®. part ii: Application. *Coastal Engineering*, 83:259–270.

Hirt, C. W. & Nichols, B. D. (1981). Volume of fluid (vof) method for the dynamics of free boundaries. *Journal of computational physics*, 39(1):201–225.

Holthuijsen, L. H. (2010). *Waves in oceanic and coastal waters*. Cambridge university press.

Jasak, H. (2009). Dynamic mesh handling in openfoam.

Jasak, H., Weller, H. G., & Nordin, N. (2004). In-cylinder cfd simulation using a c++ object-oriented toolkit. Technical report, SAE Technical Paper.

Jones, D. A., Chapuis, M., Liefvendahl, M., Norrison, D., & Widjaja, R. (2016). Rans simulations using openfoam software. Technical report, Defence Science and Technology Group Fishermans Bend Victoria Australia.

Kamath, A., Alagan Chella, M., Bihs, H., & Arntsen, Ø. A. (2017). Energy transfer due to shoaling and decomposition of breaking and non-breaking waves over a submerged bar. *Engineering Applications of Computational Fluid Mechanics*, 11(1):450–466.

Kim, J. & Moin, P. (1985). Application of a fractional-step method to incompressible navier-stokes equations. *Journal of computational physics*, 59(2):308–323.

Kisacik, D., Troch, P., & Van Bogaert, P. (2012). Description of loading conditions due to violent wave impacts on a vertical structure with an overhanging horizontal cantilever slab. *Coastal engineering*, 60:201–226.

Kolmogorov, A. N. (1941a). Energy dissipation in locally isotropic turbulence. In *Dokl. Akad. Nauk. SSSR*, volume 32, pages 19–21.

Kolmogorov, A. N. (1941b). The local structure of turbulence in incompressible viscous fluid for very large reynolds numbers. *Cr Acad. Sci. URSS*, 30:301–305.

Kolmogorov, A. N. (1941c). On degeneration (decay) of isotropic turbulence in an incompressible viscous liquid. In *Dokl. Akad. Nauk SSSR*, volume 31, pages 538–540.

Lambert, R. J. (2012). Development of a numerical wave tank using openfoam. Master's thesis.

Landaeta, C. J. (1995). Potenciales impactos ambientales generados por el dragado y la descarga del material dragado. *Instituto Nacional de Canalizaciones. Dirección de Proyectos e Investigación, Caracas–Venezuela*.

Larsen, B. E., Fuhrman, D. R., & Roenby, J. (2018). Performance of interfoam on the simulation of progressive waves. *arXiv preprint arXiv:1804.01158*.

Meng, D., Wen, M., Wei, J., & Lin, J. (2016). Hybrid implementation and optimization of openfoam on the sw26010 many-core processor.

Menter, F. R. (1994). Two-equation eddy-viscosity turbulence models for engineering applications. *AIAA journal*, 32(8):1598–1605.

Moukalled, F., Mangani, L., Darwish, M., et al. (2016). The finite volume method in computational fluid dynamics.

Nakayama, A. & Kuwahara, F. (1999). A macroscopic turbulence model for flow in a porous medium. *Journal of fluids engineering*, 121(2):427–433.

Palacios, R. B. (2015). Estudio aerodinámico de un disco volador en el seno de un flujo turbulento.

Pedrozo, A. & Torres, A. (2011). Sobre el uso de las ecuaciones de Navier-Stokes con el promedio de Reynolds en el campo de la ingeniería de costas. *Tecnología y ciencias del agua*, 2(2):51–67.

Pinales, F. & Velázquez, C. (2017). Algoritmos resueltos con diagramas de flujo y pseudocódigo.

Polubarinova-Kochina, P. (1962). Theory of ground water movement princeton university press. *Princeton, NJ*.

Rhoads, J. (2014). Effects of grid quality on solution accuracy.

Rusche, H. (2002). *Computational fluid dynamics of dispersed two-phase flows at high phase fractions*. Tesis doctoral, Imperial College, London.

Rúa, C. (2006). Los puertos en el transporte marítimo. *UPCommons*.

Silva, R. (2005). *Análisis y descripción estadística del oleaje*. Universidad Nacional Autóma de México.

Sirevaag, O. (2015). Cfd simulation of an offshore air intake and exhaust system. Master's thesis, University of Stavanger, Norway.

Slattery, J. C. (1967). Flow of viscoelastic fluids through porous media. *AIChE Journal*, 13(6):1066–1071.

Spalart, P. R. (1997). Comments on the feasibility of les for wings, and on hybrid rans/les approach. In *Proceedings of First AFOSR International Conference on DNS/LES, 1997*.

Tirindelli, M., Cuomo, G., Allsop, W., & McConnell, K. (2002). Exposed jetties: inconsistencies and gaps in design methods for wave-induced forces. In *Coastal Engineering 2002*, pages 1684–1696. WORLD SCIENTIFIC.

Universidad de Cantabria (2000). *Documento de referencia Dinámicas*, volume 1. UC.

US Army Corps Of Engineers (2002). Coastal engineering manual. *Engineer Manual*, 1110:2–1100.

Vazquez, J. L. (2018). Las ecuaciones de la filtración de fluidos en medios porosos.

Werlinger, C., Alveal, K., & Romo, H. (2004). *Biología marina y oceanografía: conceptos y procesos*. Consejo Nacional del Libro y la Lectura.

Whitaker, S. (1967). Diffusion and dispersion in porous media. *AIChE Journal*, 13(3):420–427.

Winckler, P. (2018). *Introducción al modelado de procesos costeros*. Universidad de Valparaíso, Chile, 1st edition.

Çengel, Y. A. & Cimbala, J. M. (2006). *Mecánica de fluidos, fundamentos y aplicaciones*. The McGraw-Hill, New York, 1st edition.

Anexos

Anexos A

Otras teorías de ondas

Teoría Cnoidal

Cuando la longitud de onda es grande en comparación con la profundidad, el régimen de Stokes no describe bien el comportamiento del oleaje, por lo tanto, es necesario utilizar la teoría Cnoidal desarrollada por Korteweg y Devries en el año 1895. Esta teoría está descrita mediante funciones elípticas Jacobianas, lo que le da su nombre y además es la razón por la que su utilización tiene cierta complejidad. Por otra parte, la convergencia para las funciones elípticas es muy lenta debido al tratamiento de las expresiones presentado en textos estándares. Sin embargo, en la actualidad se han desarrollado expresiones menos engorrosas y que convergen más rápidamente (Fenton, 1999). La desnivelación para la teoría Cnoidal de primer orden está dada por:

$$\eta = H\left(\frac{1}{m^2} - 1 - \frac{E}{m^2 K}\right) + Hcn^2\left(2K\left(\frac{x}{L} - \frac{t}{T}\right)\right) \tag{A.1}$$

donde:

- H corresponde a la altura de ola.
- T es el periodo.
- L es la longitud de onda.
- x es la coordenada horizontal.
- $cn(x, m)$ es la función cnoidal elíptica de Jacobi.
- m corresponde al parámetro elíptico Jacobiano en el intervalo $[0.5, 0.\bar{9}]$.
- K y E integrales elípticas.

Para conocer el rango de aplicabilidad de la teoría Cnoidal es necesario utilizar el número de Ursell dado por la ecuación 2.16, se debe cumplir que $U_r > 26$ (Holthuijsen, 2010).

Teoría Stream-function

Cuando la teoría Cnoidal o la de Stokes no representan bien el comportamiento del oleaje, es necesario recurrir a la teoría Stream-function. Esta teoría modela el comportamiento del

oleaje cerca de la condición de rotura, pero, al igual que la teoría de Stokes, Stream-function no presenta buenos resultados en aguas muy someras, entonces, para profundidades del orden de la altura de la onda es necesario utilizar la teoría Cnoidal (Holthuijsen, 2010). La superficie libre para la teoría Stream-function se puede calcular mediante:

$$\eta = d \sum_{j=1}^{n} E_j \cos j[(kx - \omega t)] \tag{A.2}$$

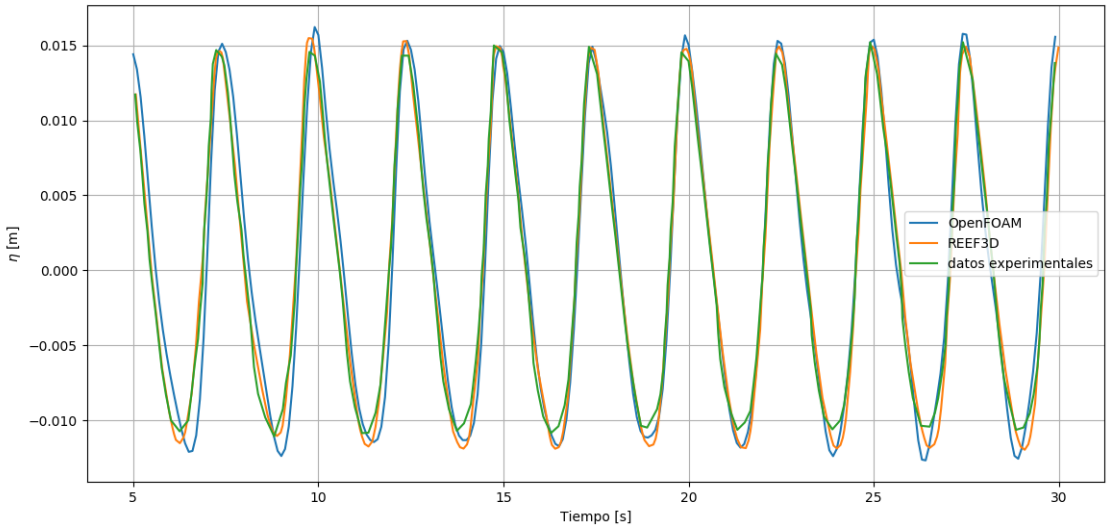
donde E_j corresponde a una serie de coeficientes empíricos, d es la profundidad, ω es la frecuencia angular y t es el tiempo. El intervalo de aplicabilidad de esta teoría está entre $H_b/4$ y H_b, con $H_b = H/d \approx 0.8$ siendo la altura de ola en rotura, d la profundidad y H la altura de ola antes de la rotura.

Al unir todas las condiciones de aplicabilidad para cada teoría es posible establecer un gráfico que incluya todas ellas, tal como se muestra en la Figura 2.7.

Anexos B

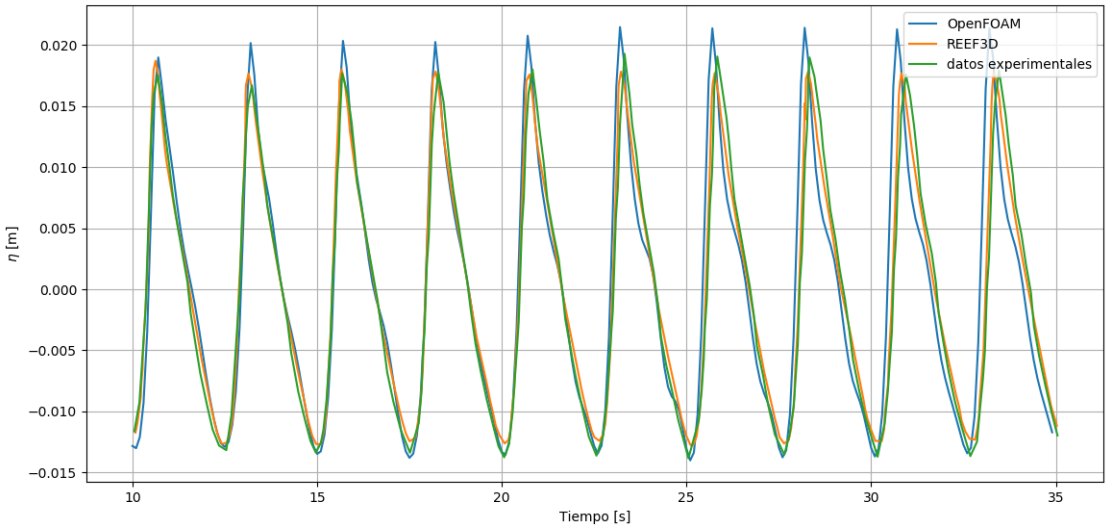
Resultados para el análisis en asomeramiento

Figura B.1: desnivelaciones en el canal a los 11 m, caso A01 (Tabla 3.14).



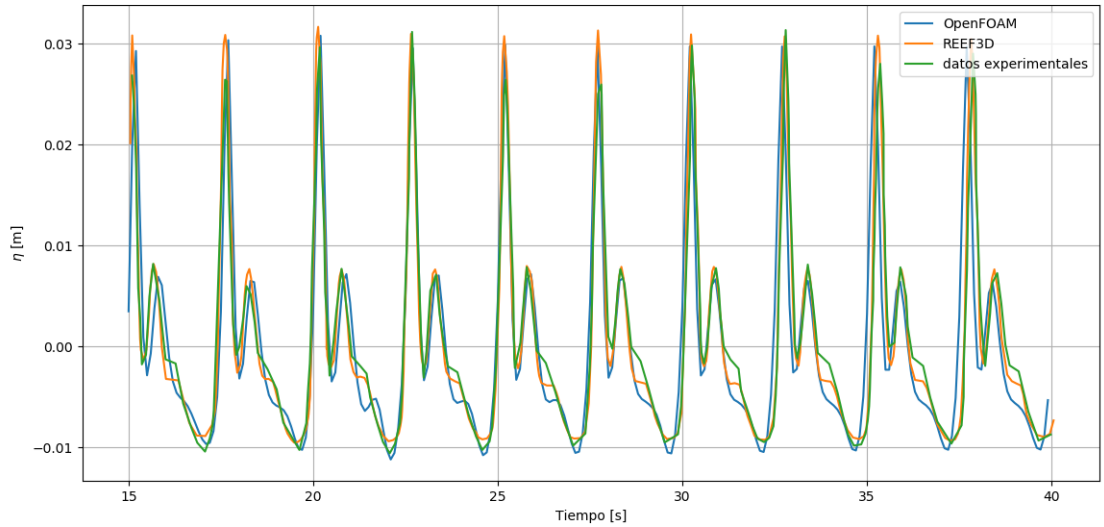
Fuente: elaboración propia.

Figura B.2: desnivelaciones en el canal a los 12 m, caso A01 (Tabla 3.14).



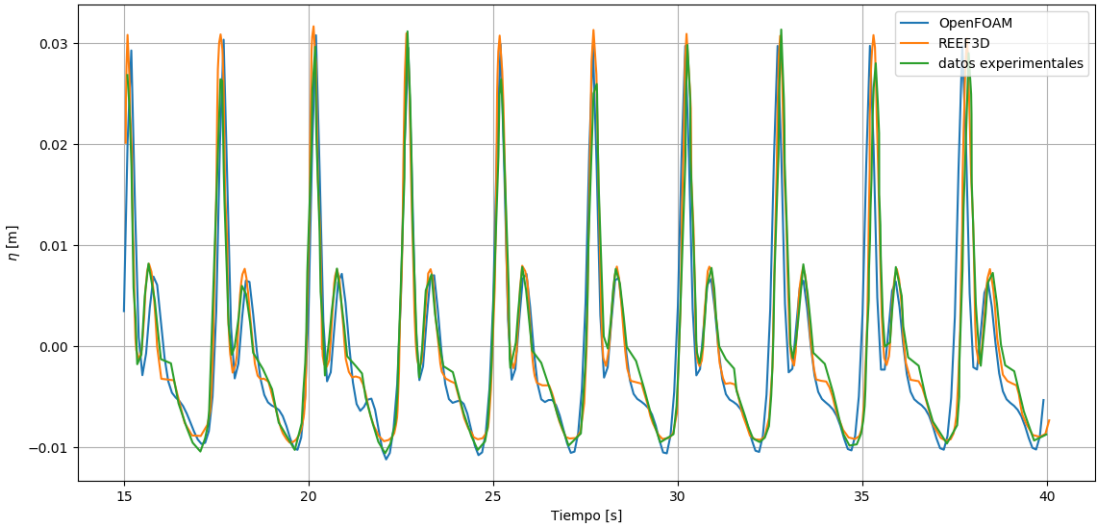
Fuente: elaboración propia.

Figura B.3: desnivelaciones en el canal a los 14 m, caso A01 (Tabla 3.14).



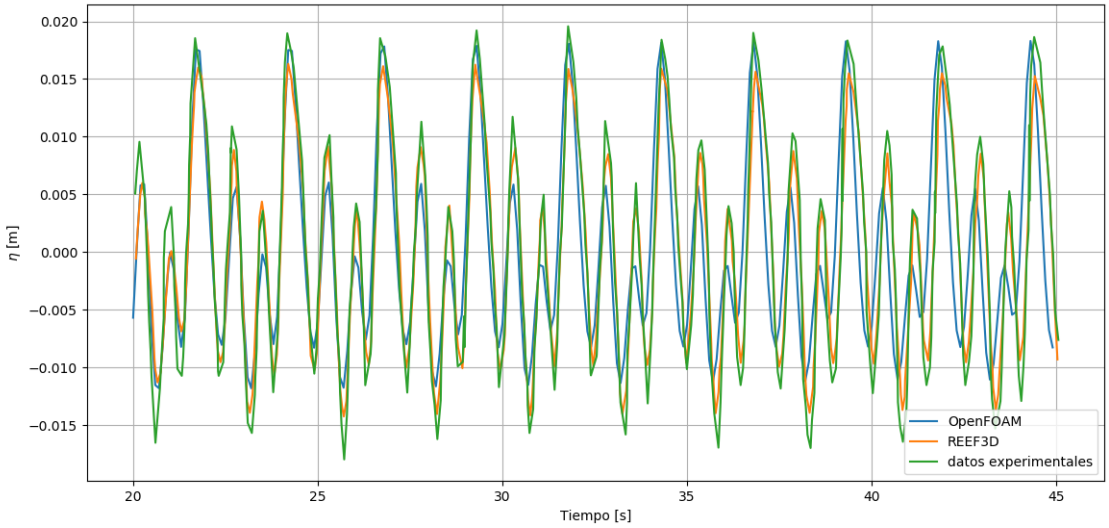
Fuente: elaboración propia.

Figura B.4: desnivelaciones en el canal a los 15 m, caso A01 (Tabla 3.14).



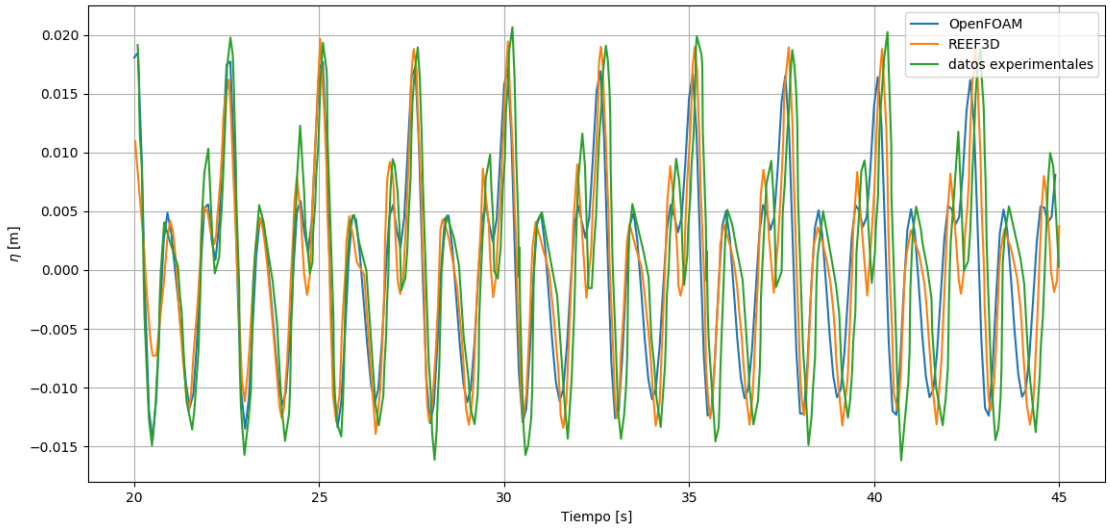
Fuente: elaboración propia.

Figura B.5: desnivelaciones en el canal a los 16 m, caso A01 (Tabla 3.14).



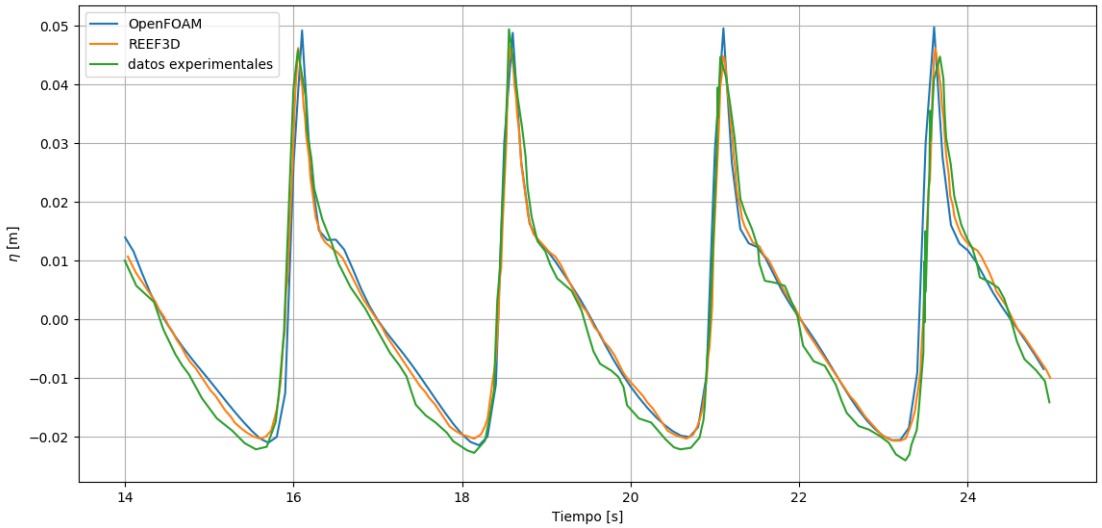
Fuente: elaboración propia.

Figura B.6: desnivelaciones en el canal a los 17 m, caso A01 (Tabla 3.14).



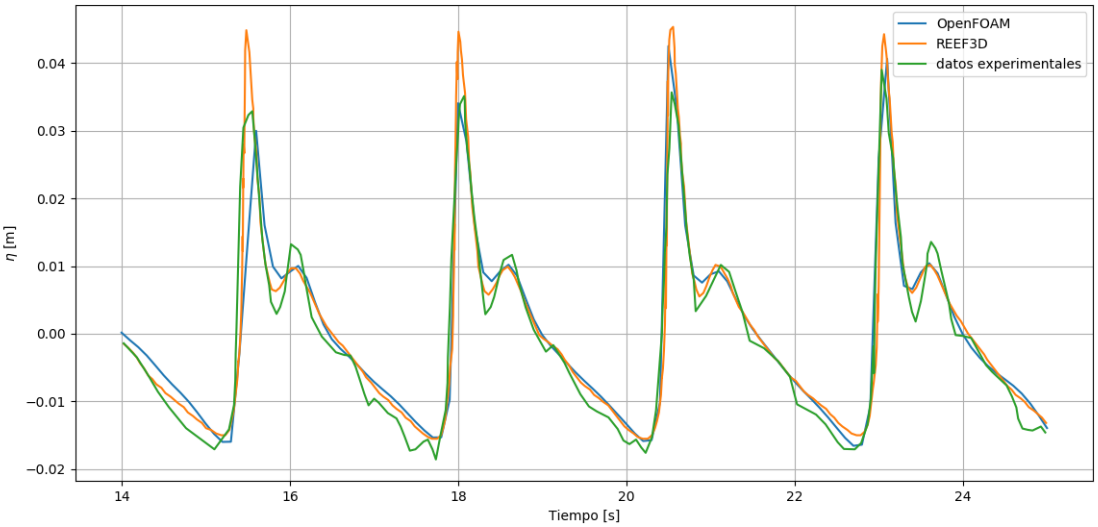
Fuente: elaboración propia.

Figura B.7: desnivelaciones en el canal a los 12 m, caso A02 (Tabla 3.14).



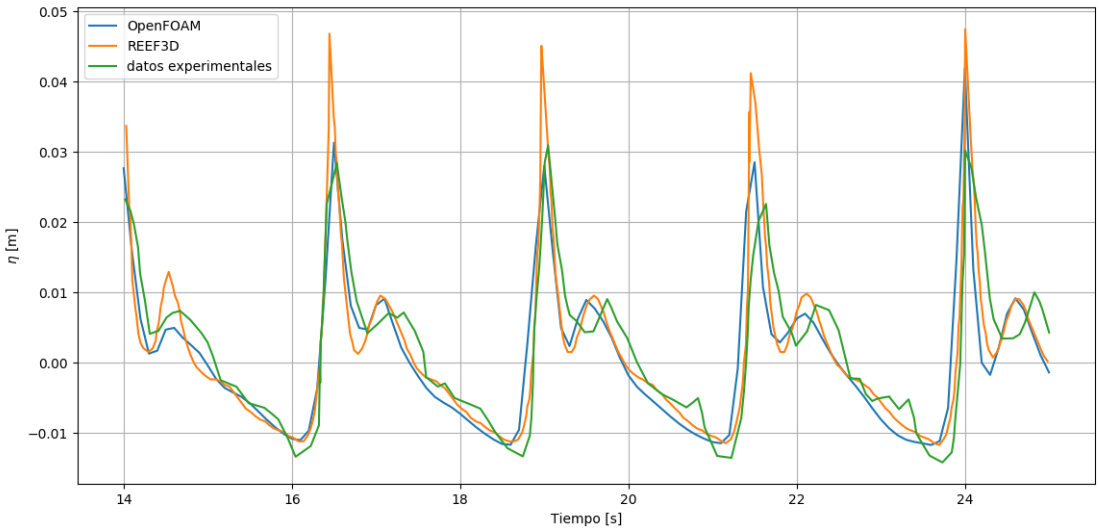
Fuente: elaboración propia.

Figura B.8: desnivelaciones en el canal a los 13 m, caso A02 (Tabla 3.14).



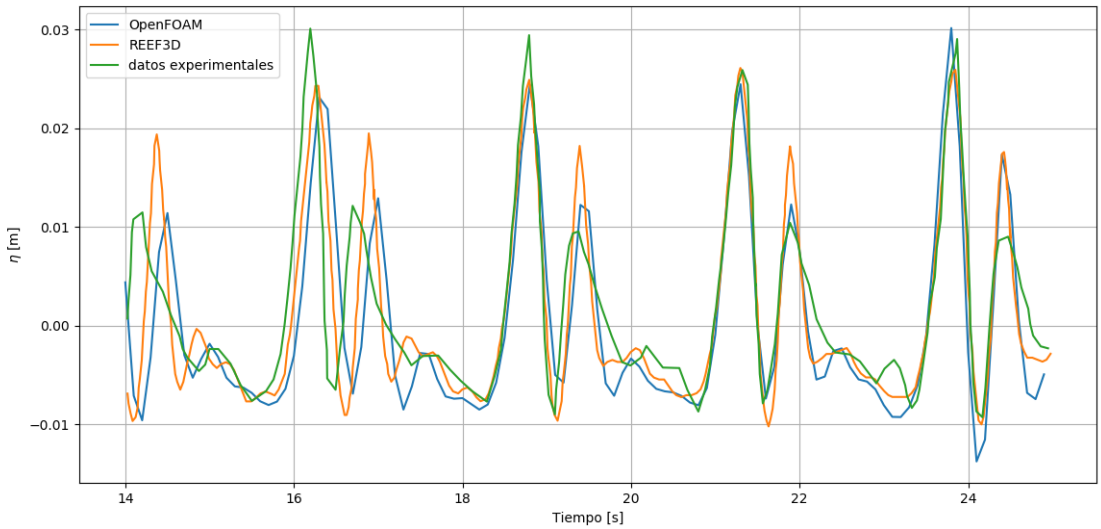
Fuente: elaboración propia.

Figura B.9: desnivelaciones en el canal a los 14 m, caso A02 (Tabla 3.14).



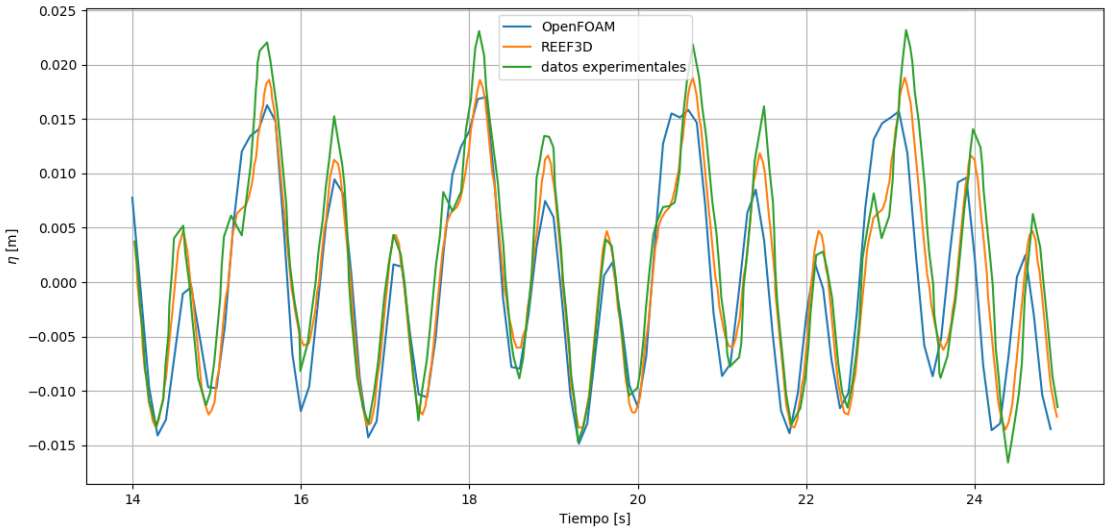
Fuente: elaboración propia.

Figura B.10: desnivelaciones en el canal a los 15 m, caso A02 (Tabla 3.14).



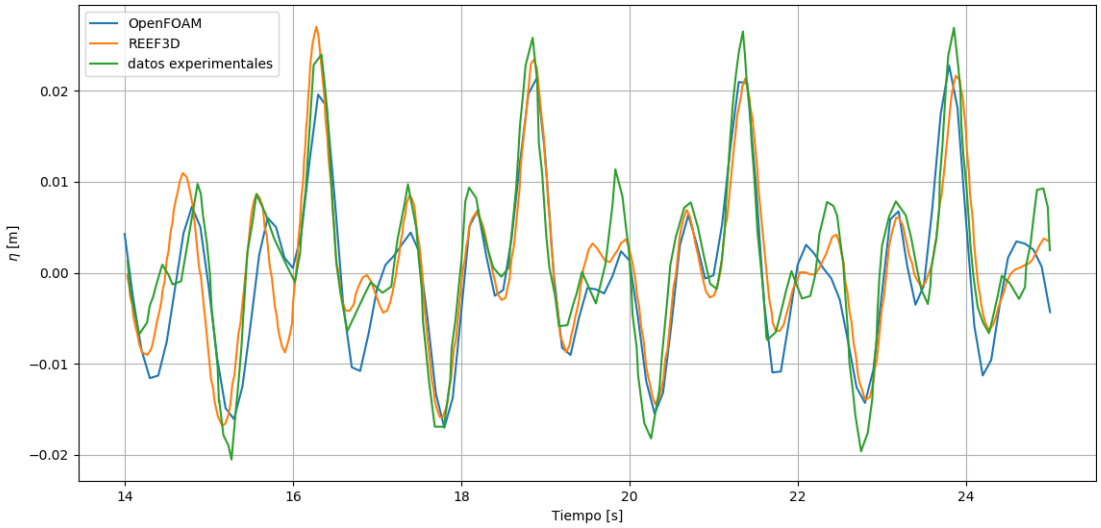
Fuente: elaboración propia.

Figura B.11: desnivelaciones en el canal a los 16 m, caso A02 (Tabla 3.14).



Fuente: elaboración propia.

Figura B.12: desnivelaciones en el canal a los 17 m, caso A02 (Tabla 3.14) .



Fuente: elaboración propia.

Anexos C

Deducción de las ecuaciones de Navier-Stokes

En el siguiente apartado se presentan los teoremas utilizados, supuestos y simplificaciones para deducir las ecuaciones de Navier-Stokes.

C.1. Teorema de transporte de Reynolds

Un sistema abierto (o volumen de control) está limitado por contornos llamados superficies de control, donde la masa puede entrar o salir del sistema. El teorema de transporte de Reynolds relaciona la variación de una propiedad extensiva (B) (es decir, una propiedad del fluido que depende de la masa total o volumen total, como la energía o cantidad de movimiento) en el tiempo entre una superficie de control y un volumen de control (Çengel & Cimbala, 2006):

$$\frac{dB_{sis}}{dt} = \frac{d}{dt}\int_{VC} \rho b \, d\nu + \int_{SC} \rho b \vec{V} \cdot \vec{n} \, dA \tag{C.1}$$

con:

- $b = B/m$, B es una propiedad extensiva, m es la masa, luego b es una propiedad intensiva (independiente del volumen total o la masa total).
- ρ corresponde a la densidad del fluido.
- $\vec{n}$ es el vector normal exterior unitario.
- dA es el área superficial diferencial.
- $\vec{V}$ es el vector velocidad.

El teorema de transporte de Reynolds puede describirse como: la variación de una propiedad extensiva B en el sistema con respecto al tiempo, es igual a la variación de la propiedad B en el volumen de control, más la razón neta de flujo de salida debido a la masa que cruza la superficie de control. Si además este volumen de control se mueve o se deforma, entonces la ecuación C.1 se transforma en:

$$\frac{dB_{sis}}{dt} = \frac{\partial}{\partial t}\int_{VC} \rho b \, d\nu + \int_{SC} \rho b \vec{V} \cdot \vec{n} \, dA \tag{C.2}$$

C.2. Teorema de la divergencia

El teorema de la divergencia (o teorema de Gauss) permite transformar una integral de volumen de la divergencia de un vector en una integral de área sobre la superficie en que se define el volumen. En las subsecciones posteriores se utiliza para deducir la forma diferencial de la ecuación de conservación de la masa, y también para deducir la forma diferencial de la conservación de cantidad de movimiento.

Si se tiene un vector $\vec{G}$, su divergencia está dada por $\vec{\nabla} \cdot \vec{G}$, y el teorema de la divergencia se describe como:

$$\int_{\nu} \vec{\nabla} \cdot \vec{G} d\nu = \oint \vec{G} \cdot \vec{n} \, dA \tag{C.3}$$

La integral debe evaluarse en toda el área; A es el área, $d\nu$ es el diferencial de volumen y $\vec{n}$ es el vector normal exterior unitario.

C.3. Ecuación de continuidad

En un volumen de control, la masa puede entrar o salir libremente por las superficies de control, por lo tanto, se debe tener en cuenta la razón de flujo de masa (cantidad de masa que fluye a través de una superficie de control por unidad de tiempo) en el sistema. El principio de conservación de la masa para un volumen de control indica que: la razón de flujo de masa que entra a ese volumen de control, menos la razón de flujo masa que sale de él, debe ser igual a la razón de cambio de la masa que está dentro de las fronteras de ese volumen de control, esta relación para un volumen diferencial de control se denomina ecuación de continuidad:

$$\dot{m}_{ent} - \dot{m}_{sal} = \frac{dm}{dt} \tag{C.4}$$

Si en la ecuación C.2, la propiedad extensiva (B) es la masa (m), entones $b = 1$ y si se tiene en cuenta que la masa en el sistema es constante ($dm_{sist}/dt = 0$), entonces se obtiene la ecuación general de conservación de la masa para un volumen de control (fijo o en movimiento):

$$\int_{VC} \frac{\partial \rho}{\partial t} d\nu + \int_{SC} \rho \vec{V} \cdot \vec{n} \, dA = 0 \tag{C.5}$$

Luego, aplicando el teorema de la divergencia (ecuación C.3) para deducir la forma diferencial de la conservación de la masa, la ecuación C.5 se transforma en:

$$\int_{VC} \left[\frac{\partial \rho}{\partial t} + \vec{\nabla} \cdot (\rho \vec{V}) \right] d\nu = 0 \tag{C.6}$$

Como los términos de la integral deben ser iguales a cero, entonces se obtiene la ecuación de continuidad, válida tanto como para flujo compresible o incompresible:

$$\frac{\partial \rho}{\partial t} + \vec{\nabla} \cdot (\rho \vec{V}) = 0 \tag{C.7}$$

Cuando el flujo es incompresible, la variación de la densidad ρ en el tiempo tiende a cero, luego ρ se puede sacar del operador de divergencia en la ecuación C.7, entonces la ecuación de continuidad para un flujo incompresible queda expresada como:

$$\rho(\vec{\nabla} \cdot \vec{V}) = 0 \tag{C.8}$$

C.4. Ecuación de conservación de momento

La segunda ley de Newton indica que la variación de la cantidad de movimiento de un cuerpo es igual a la fuerza neta que actúa sobre él, esto queda expresado como:

$$\vec{F} = m\vec{a} = m\frac{d\vec{V}}{dt} = \frac{d(m\vec{V})}{dt} \tag{C.9}$$

con $\vec{F}$ representando a la fuerza que actúa sobre un cuerpo, m es la masa, $\vec{V}$ es la velocidad, y $\vec{a}$ es la aceleración. Finalmente el momento lineal está dado por $m\vec{V}$, también se le llama cantidad de movimiento del cuerpo.

La ecuación C.9 sobre un sistema se puede expresar mediante:

$$\sum \vec{F} = \frac{d}{dt}\int_{sis} \rho \vec{V} d\nu \tag{C.10}$$

La masa de un elemento diferencial de volumen está dada por $\delta m = \rho d\nu$, luego la cantidad de movimiento es $\rho\vec{V}d\nu$. Pero para utilizar la ecuación C.10 en un volumen de control, es necesario transformarla mediante el teorema de transporte de Reynolds (ecuación C.2). Si se hace $b = \vec{V}$, entonces $B = m\vec{V}$, luego el teorema de transporte de Reynolds para el momento lineal puede expresarse como:

$$\sum \vec{F} = \frac{d(m\vec{V})_{sis}}{dt} = \frac{d}{dt}\int_{VC} \rho \vec{V} d\nu + \int_{SC} \rho \vec{V}(\vec{V}_r \cdot \vec{n}) dA \tag{C.11}$$

donde $\vec{V}_r$ es la velocidad del flujo con relación a la superficie de control ($\vec{V}_r = \vec{V} - \vec{V}_{SC}$), tomando en cuenta la ecuación C.23:

$$\sum \vec{F} = \int_{VC} \rho \vec{g}\, d\nu + \int_{SC} \sigma_{ij} \cdot \vec{n}\, dA = \frac{\partial}{\partial t}\int_{VC} \rho \vec{V} d\nu + \int_{SC} \rho \vec{V}(\vec{V}_r \cdot \vec{n})\, dA \tag{C.12}$$

Finalmente, para deducir la forma diferencial de la conservación de momento se aplica el teorema de divergencia, ya que se cumple también para tensores, entonces se toma el último término en la ecuación C.12 y se le aplica la ecuación C.3:

$$\int_{SC} (\rho\vec{V})\vec{V} \cdot \vec{n}\, dA = \int_{VC} \vec{\nabla} \cdot (\rho\vec{V}\vec{V})\, d\nu \tag{C.13}$$

En la integral de volumen de control se genera un producto vectorial entre los dos vectores de velocidad, a esta operación se le conoce como producto exterior.

Se vuelve a aplicar el teorema de divergencia a la integral de superficie de control (SC) que contiene al tensor en la ecuación C.12:

$$\int_{SC} \sigma_{ij} \cdot \vec{n}\, dA = \int_{VC} \vec{\nabla} \cdot \sigma_{ij}\, d\nu \tag{C.14}$$

De esta manera las integrales de superficie de la ecuación C.12 se convierten en integrales de volumen, luego se reordenan quedando:

$$\int_{VC} \left[\frac{\partial}{\partial t}(\rho\vec{V}) + \vec{\nabla} \cdot (\rho\vec{V}\vec{V}) - \rho\vec{g} - \vec{\nabla} \cdot \sigma_{ij} \right] d\nu = 0 \tag{C.15}$$

Para que la ecuación C.15 se cumpla para cualquier volumen de control, es necesario que el integrando sea igual a cero (Çengel & Cimbala, 2006), entonces esta simplificación se conoce como la ecuación de Cauchy, válida tanto para flujo compresible como para incompresible:

$$\frac{\partial}{\partial t}(\rho\vec{V}) + \rho(\vec{\nabla} \cdot \vec{V})\vec{V} = \rho\vec{g} + \vec{\nabla} \cdot \sigma_{ij} \tag{C.16}$$

Si se considera que la derivada material de un elemento de fluido está dada por:

$$\frac{D}{Dt} = \frac{\partial}{\partial t} + (\vec{V} \cdot \vec{\nabla}) \tag{C.17}$$

entonces la ecuación de Cauchy también puede ser representada mediante:

$$\rho\frac{D\vec{V}}{Dt} = \rho\vec{g} + \vec{\nabla} \cdot \sigma_{ij} \tag{C.18}$$

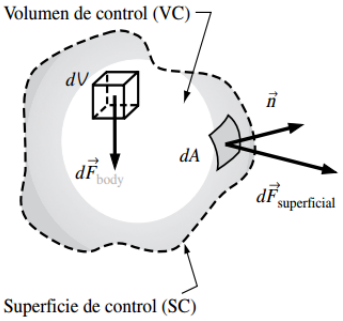
Luego, la ecuación de Cauchy puede ser descompuesta en las tres componentes (x, y, z):

$$\begin{aligned}
\rho\frac{Du}{Dt} &= \rho g_x + \frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \sigma_{yx}}{\partial y} + \frac{\partial \sigma_{zx}}{\partial z} \\
\rho\frac{Dv}{Dt} &= \rho g_y + \frac{\partial \sigma_{xy}}{\partial x} + \frac{\partial \sigma_{yy}}{\partial y} + \frac{\partial \sigma_{zy}}{\partial z} \\
\rho\frac{Dw}{Dt} &= \rho g_z + \frac{\partial \sigma_{xz}}{\partial x} + \frac{\partial \sigma yz}{\partial y} + \frac{\partial \sigma_{zz}}{\partial z}
\end{aligned} \tag{C.19}$$

C.5. Fuerzas que actúan sobre un volumen de control

Para representar todas las fuerzas que actúan sobre un volumen de control, es necesario considerar las fuerzas de cuerpo y adherirlas a las fuerzas de superficie. Estas fuerzas de cuerpo se caracterizan por actuar en todo el cuerpo del volumen analizado.

Figura C.1: volumen de control en el que actúan fuerzas de cuerpo y fuerzas de superficie.



Fuente: Çengel & Cimbala (2006).

Si se analiza un elemento diferencial de ese volumen de control, la fuerza de cuerpo está dada por su propio peso $d\vec{F} = \rho\vec{g}d\nu$, con ρ representando la densidad, el vector $\vec{g}$ es la gravedad y $d\nu$ es una porción diferencial de volumen. Entonces, la fuerza total de cuerpo que actúa sobre el volumen de control se expresa como:

$$\sum \vec{F}_{cuerpo} = \int_{VC} \rho\vec{g}\, d\nu \tag{C.20}$$

Para describir el comportamiento de las fuerzas superficiales sobre un volumen de control, es necesario presentar primero el tensor de esfuerzos. El tensor de segundo orden o tensor de esfuerzos (σ_{ij}) sirve para describir de manera adecuada los esfuerzos superficiales en un punto del flujo:

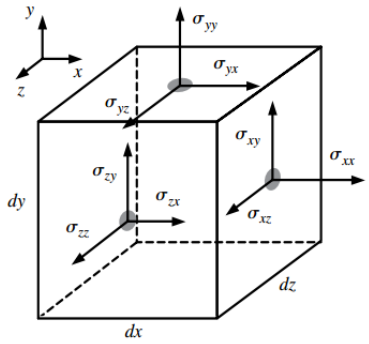
$$\sigma_{ij} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{bmatrix}$$

La matriz presentada está compuesta por esfuerzos normales, los que están en la diagonal del tensor $\sigma_{xx}, \sigma_{yy}, \sigma_{zz}$, y fuera de la diagonal están los esfuerzos cortantes. En un tensor de esfuerzos se presenta el esfuerzo (fuerza por unidad de área) en la dirección j, que actúa sobre una cara cuya normal está en la dirección i, en la figura C.2 se muestra esta componente del tensor de esfuerzos para el caso de un elemento diferencial de fluido.

El producto de un tensor de segundo orden y un vector entrega como resultado otro vector, esta operación se conoce como producto contraído o producto interior de un tensor y un vector. Al aplicar el producto contraído del tensor de esfuerzos σ_{ij} con el vector normal unitario $\vec{n}$ de una porción diferencial de área, el resultado obtenido es la fuerza superficial que actúa sobre un elemento diferencial de área:

$$d\vec{F}_{superficie} = \sigma_{ij} \cdot \vec{n}\, dA \tag{C.21}$$

Figura C.2: componente de tensor de esfuerzos en coordenadas cartesianas, sobre la cara derecha, superior y del frente, las componentes solo se muestran en caras positivas.



Fuente: Çengel & Cimbala (2006).

Luego integrando la ecuación C.21 sobre la superficie de control se obtiene la fuerza que actúa sobre toda la superficie de control:

$$\sum \vec{F}_{sup} = \int_{SC} \sigma_{ij} \cdot \vec{n}\, dA \tag{C.22}$$

Finalmente las fuerzas que actúan sobre el volumen de control se describen mediante:

$$\sum \vec{F} = \int_{VC} \rho \vec{g}\, d\nu + \int_{SC} \sigma_{ij} \cdot \vec{n}\, dA \tag{C.23}$$

C.6. Ecuaciones de Navier-Stokes

Las ecuaciones de Navier-Stokes se deducen a partir de la ecuación de continuidad y la ecuación de Cauchy. A continuación, se presenta el desarrollo de las ecuaciones basado en Çengel & Cimbala (2006).

Es necesario escribir las componentes del tensor de esfuerzos en términos del campo de velocidad y presión para generar ecuaciones constitutivas, las cuales sirven para igualar la cantidad de incógnitas a la de ecuaciones de Cauchy y de continuidad para que puedan ser matemáticamente solucionables.

La presión hidrostática P actúa sobre un fluido de forma normal a la superficie y hacia adentro, cuando el fluido está en reposo el tensor de esfuerzos se puede representar mediante:

$$\sigma_{ij} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{bmatrix} = \begin{bmatrix} -P & 0 & 0 \\ 0 & -P & 0 \\ 0 & 0 & -P \end{bmatrix} + \begin{bmatrix} \tau_{xx} & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \tau_{yy} & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \tau_{zz} \end{bmatrix} \tag{C.24}$$

El tensor de esfuerzo viscoso (τ_{ij}) se ha agregado como parte del tensor de esfuerzo σ_{ij} ya que se generan ecuaciones constitutivas que lo relacionan al campo de velocidad.

Si se considera un fluido incompresible (ρ = constante) y además newtoniano, es decir, el tensor de esfuerzos es linealmente proporcional al tensor de razón de deformación, y si

además se considera el cambio de temperatura despreciable en el fluido, se puede descartar la ecuación diferencial de conservación de la energía; esto como consecuencia genera el supuesto de que tanto la viscosidad dinámica (μ) como la viscosidad cinemática (ν) son constantes (Çengel & Cimbala, 2006). Los anteriores supuestos establecen que:

$$\tau_{ij} = 2\mu\epsilon_{ij} \tag{C.25}$$

con ϵ siendo el tensor de razón de deformación, el cual combina la tasa de deformación lineal y la tasa de deformación por esfuerzo constante (para mayor información referirse a Çengel & Cimbala (2006)):

$$\epsilon_{ij} = \begin{bmatrix} \epsilon_{xx} & \epsilon_{xy} & \epsilon_{xz} \\ \epsilon_{yx} & \epsilon_{yy} & \epsilon_{yz} \\ \epsilon_{zx} & \epsilon_{zy} & \epsilon_{zz} \end{bmatrix} = \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{1}{2}\left(\frac{\partial u}{\partial y} + \frac{\partial \nu}{\partial x}\right) & \frac{1}{2}\left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right) \\ \frac{1}{2}\left(\frac{\partial \nu}{\partial x} + \frac{\partial u}{\partial y}\right) & \frac{\partial \nu}{\partial y} & \frac{1}{2}\left(\frac{\partial \nu}{\partial z} + \frac{\partial w}{\partial y}\right) \\ \frac{1}{2}\left(\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z}\right) & \frac{1}{2}\left(\frac{\partial w}{\partial y} + \frac{\partial \nu}{\partial z}\right) & \frac{\partial w}{\partial z} \end{bmatrix} \tag{C.26}$$

Luego, el tensor de esfuerzo viscoso queda como:

$$\tau_{ij} = \begin{bmatrix} 2\mu\frac{\partial u}{\partial x} & \mu\left(\frac{\partial u}{\partial y} + \frac{\partial \nu}{\partial x}\right) & \mu\left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right) \\ \mu\left(\frac{\partial \nu}{\partial x} + \frac{\partial u}{\partial y}\right) & 2\mu\frac{\partial \nu}{\partial y} & \mu\left(\frac{\partial \nu}{\partial z} + \frac{\partial w}{\partial y}\right) \\ \mu\left(\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z}\right) & \mu\left(\frac{\partial w}{\partial y} + \frac{\partial \nu}{\partial z}\right) & 2\mu\frac{\partial w}{\partial z} \end{bmatrix} \tag{C.27}$$

Finalmente, el tensor de esfuerzo queda representado mediante:

$$\sigma_{ij} = \begin{bmatrix} -P & 0 & 0 \\ 0 & -P & 0 \\ 0 & 0 & -P \end{bmatrix} + \begin{bmatrix} 2\mu\frac{\partial u}{\partial x} & \mu\left(\frac{\partial u}{\partial y} + \frac{\partial \nu}{\partial x}\right) & \mu\left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right) \\ \mu\left(\frac{\partial \nu}{\partial x} + \frac{\partial u}{\partial y}\right) & 2\mu\frac{\partial \nu}{\partial y} & \mu\left(\frac{\partial \nu}{\partial z} + \frac{\partial w}{\partial y}\right) \\ \mu\left(\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z}\right) & \mu\left(\frac{\partial w}{\partial y} + \frac{\partial \nu}{\partial z}\right) & 2\mu\frac{\partial w}{\partial z} \end{bmatrix} \tag{C.28}$$

Ahora, tomando el tensor de esfuerzo de la ecuación C.28 y reemplazándolo en la componente en x de la ecuación C.19 queda:

$$\rho\frac{Du}{Dt} = \rho g_x + \frac{\partial}{\partial x}\left[-P + 2\mu\frac{\partial u}{\partial x}\right] + \frac{\partial}{\partial y}\left[\mu\left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\right)\right] + \frac{\partial}{\partial z}\left[\mu\left(\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z}\right)\right] \tag{C.29}$$

Reordenando los términos de la ecuación C.29:

$$\rho\frac{Du}{Dt} = \rho g_x - \frac{\partial P}{\partial x} + \mu\left[\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) + \frac{\partial^2 u}{\partial x^2} + \frac{\partial}{\partial z}\left(\frac{\partial w}{\partial x}\right) + \frac{\partial}{\partial y}\left(\frac{\partial v}{\partial x}\right)\right] \tag{C.30}$$

Si se considera que las componentes de la velocidad son funciones suaves de x, y y z, entonces:

$$\frac{\partial}{\partial z}\left(\frac{\partial w}{\partial x}\right) = \frac{\partial}{\partial x}\left(\frac{\partial w}{\partial z}\right)$$

$$\frac{\partial}{\partial y}\left(\frac{\partial v}{\partial x}\right) = \frac{\partial}{\partial x}\left(\frac{\partial v}{\partial y}\right)$$

Si además se considera que el laplaciano está definido por:

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \tag{C.31}$$

Entonces la ecuación C.30 puede reescribirse como:

$$\rho\frac{Du}{Dt} = \rho g_x - \frac{\partial P}{\partial x} + \mu\left[\nabla^2 u + \frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right)\right] \tag{C.32}$$

Al considerarse la ecuación de continuidad:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{C.33}$$

Finalmente, para la componente x de la ecuación de cantidad de movimiento puede escribirse como:

$$\rho\frac{Du}{Dt} = \rho g_x - \frac{\partial P}{\partial x} + \mu\nabla^2 u \tag{C.34}$$

Si se desarrolla un procedimiento similar para las otras componentes en y y en z y se reescriben en forma vectorial, entonces la ecuación de Navier-Stokes está dada por:

$$\rho\frac{D\vec{V}}{Dt} = \rho\vec{g} - \vec{\nabla}P + \mu\nabla^2\vec{V} \tag{C.35}$$

Anexos D

Consideraciones en la calibración del modelo

D.1. Porosidad de los materiales

Cuando el fluido interactúa con un medio poroso, avanza a través de los espacios que hay en el material, lo que representa un inconveniente al modelar su comportamiento, ya que la geometría de la estructura es muy compleja y además es necesario tomar en cuenta su resistencia al paso del fluido (Vazquez, 2018).

El solucionador olaFlow utiliza las ecuaciones RANS promediadas en el volumen, por lo que modela el movimiento del fluido a través de medios porosos sin tomar en cuenta su geometría interior, esto es posible debido a la técnica de derivación de las VARANS desarrollada por Slattery (1967) y Whitaker (1967), quienes establecieron los fundamentos matemáticos para llevar a cabo este procedimiento.

La ecuación para flujo de fluidos a través de medios porosos, utilizada por olaFlow fue presentada por Polubarinova-Kochina (1962):

$$I = A\mathbf{u} + B\left|\mathbf{u}\right|\mathbf{u} + C\frac{\partial \mathbf{u}}{\partial t} \tag{D.1}$$

donde $\mathbf{u}$ corresponde a la velocidad del fluido, I es el gradiente hidráulico, A y B son coeficientes de fricción dados por las formulaciones de Engelund (1953):

$$A = \frac{\alpha(1-\phi)^3}{\phi^2}\frac{\nu}{D_{50}^2} \tag{D.2}$$

$$B = \beta\frac{1-\phi}{\phi^3}\frac{1}{D_{50}} \tag{D.3}$$

con ϕ siendo la porosidad (fracción de volumen de fluido contenido en un volumen de control), ν es la visocidad cinemática del fluido y D_{50} es el diámetro medio nominal de los poros.

El coeficiente de fricción lineal α, el coeficiente de fricción no lineal β y C dependen de factores como el número de Reynolds, la permeabilidad y las condiciones del flujo,

sin embargo, aún no se conoce completamente su comportamiento (Arjona, 2016). Estos valores influyen en gran medida en los resultados, ya que tener una fricción excesiva puede disminuir la velocidad del fluido en el medio poroso, y al tener insuficiente fricción ocurre lo contrario (Higuera, 2015). El factor C tiene menos variación por lo que se utiliza el valor 0.34, propuesto por del Jesus (2011). Para el caso de α y β se utilizan los valores obtenidos experimentalmente por Higuera (2015):

Tabla D.1: factores de fricción para distintos regímenes de flujo.

	Laminar	Transición 01	Transición 02	Turbulento
α	300	500	5000	0.0
β	0.2	0.0	1.1	2.0

Fuente: (Higuera, 2015)

El archivo **setFieldsDict** almacena el comportamiento de los materiales que interactúan con el oleaje dentro del dominio, se indica además la profundidad del agua y el índice de porosidad que tendrá cada elemento, cada índice está relacionado con un archivo llamado **porosityDict**, este archivo está ubicado dentro del directorio **constant** de cada caso.
Los distintos materiales deben estar en formato ".stl" en la carpeta **triSurface** dentro del directorio **constant** y son usados por el archivo **setFieldsDict**, es por esto que se debe especificar la ruta de cada material.

D.2. Métodos iterativos para resolver ecuaciones

A continuación, se presenta un resumen de los principales métodos disponibles en OpenFOAM para resolver las ecuaciones matriciales proporcionadas por los esquemas de discretización.

D.2.1. PBiCG

El método PBiCG (Preconditioned Bi-conjugate Gradient Method) es un solver del subespacio de Krilov, por lo que se puede definir como un método iterativo que se utiliza para resolver grandes sistemas de ecuaciones. Además, este tipo de solvers está diseñado para evitar la factorización sobre el espacio vectorial resultante, minimizando así el residuo (Meng et al., 2016).

D.2.2. Método GAMG

El método GAMG genera una solución inicial mediante la utilización de una malla gruesa para suavizar errores de alta frecuencia, luego, en base a esta estimación se desarrolla una solución más precisa y una convergencia más eficiente. Esto se hace mediante la transferencia de la solución a mallas más finas hasta alcanzar la malla original (Dyson, 2018).

Según Jones et al. (2016) el solver PBiCG tiene un mejor comportamiento en geometrías complejas que GAMG, ya que el solver GAMG tiene problemas de estabilidad en este tipo de geometrías. Por otra parte, GAMG es más rápido en geometrías simples.

D.2.3. Método Gauss-Siedel

El método Gauss-Siedel es un suavizador que se utiliza para resolver sistemas de ecuaciones lineales que generan matrices con diagonales distintas de cero, al pertenecer al tipo smoothSolver funciona bien tanto para matrices simétricas como para asimétricas (Sirevaag, 2015).

El método Gauss-Siedel puede ser descrito mediante:

$$\phi_i^{(n)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}\phi_j^{(n)} - \sum_{j=i+1}^{N} a_{ij}\phi_j^{(n-1)} \right) \qquad i = 1, 2, 3, \ldots, N \tag{D.4}$$

en forma matricial:

$$\boldsymbol{\phi}^n = -(\mathbf{D} + \mathbf{L})^{-1}\mathbf{U}\boldsymbol{\phi}^{(n-1)} + (\mathbf{D} + \mathbf{L})^{-1}\mathbf{b} \tag{D.5}$$

D.2.4. Método DILU

Otra técnica de descomposición de matrices es DILU (Diagonal Incomplete Lower Upper), la que converge más rápidamente que los métodos Gauss-Siedel y Jacobi. Esto es posible mediante precondicionadores más avanzados, los que generan una factorización incompleta de la matriz original de coeficientes $\mathbf{A}$. Como consecuencia de esto se reduce el costo computacional y los requerimientos de memoria (Moukalled et al., 2016). Primero es necesario introducir la descomposición incompleta LU o ILU definida como:

$$\mathbf{A}\boldsymbol{\phi} = \mathbf{b} \Rightarrow \mathbf{b} - \mathbf{A}\boldsymbol{\phi} = 0 \Rightarrow (\mathbf{A} - \mathbf{R})\boldsymbol{\phi} = (\mathbf{A} - \mathbf{R})\boldsymbol{\phi} + (\mathbf{b} - \mathbf{A}\boldsymbol{\phi}) \tag{D.6}$$

donde $\mathbf{R}$ es el residuo del proceso de factorización, y luego según Moukalled et al. (2016) el proceso iterativo queda descrito mediante:

$$(\mathbf{A} - \mathbf{R})\boldsymbol{\phi}^{(n)} = (\mathbf{A} - \mathbf{R})\boldsymbol{\phi}^{(n-1)} + (\mathbf{b} - \mathbf{A}\boldsymbol{\phi}^{(n-1)}) \tag{D.7}$$

Ahora, para el caso en que se utilizan precondicionadores y se considera que $\mathbf{A} \approx \overline{\mathbf{L}}\,\overline{\mathbf{U}}$. Luego, eligiendo $\mathbf{P} = \overline{\mathbf{L}}\,\overline{\mathbf{U}}$ se introduce el método diagonal ILU o DILU, donde solo se modifican los elementos en la diagonal de la matriz. Entonces el precondicionador queda como:

$$\mathbf{P} = (\mathbf{D}' + \ \mathbf{L})\ \mathbf{D}'^{-1}(\mathbf{D}' + \mathbf{U}) \tag{D.8}$$

el nuevo término $\mathbf{D}'$ es la matriz diagonal de $\overline{\mathbf{L}}\,\overline{\mathbf{U}}$.

Anexos E

Spripts para la automatización de procesos

E.1. Scripts para el procesamiento de datos

E.1.1. Script 01

```
# -*- coding: cp1252 -*-

import numpy as np
import string
import math as mt
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import os

#----------------Variables-----------------------
prs = 5 # precisión (en decimales) para la altura de ola y los sensores
hr = []  # altura de ola relativa de rotura
def plot(y,dt,ly ='-'):
    x = []
    # graficador
    for t in range(len(y)):
        x.append(t)
    xx= []
    for   v in x:
        xx.append(v/dt)

    plt.plot(xx,y, label ='Sensor '+ str(ly))
    plt.xlabel('tiempo[s]')
    plt.ylabel('desnivel [$\eta$]')
    #plt.show()

def plot2(t,y,r):
    plt.plot(t,y, color =r)
```

```
    #plt.show()

def sca(x,y):
    # graficador
    xx= []
    for   v in x:
        xx.append(v/1.0)

    plt.plot(xx,y)
    plt.xlabel('tiempo[s]')
    plt.ylabel('desnivel [$\eta$]')
    #plt.show()

def vector(caso,sensor):
    a = open('../'+str(caso)+'/data/data_sensor'+str(sensor),'r')
    # esta función convierte un archivo en una lista
    lista = [] # donde se guarda l resultado
    # esta función convierte los datos de un archivo en un vector
    # esta función solo devuelve la profundidad, no el tiempo
    for linea in a:
        linea = linea.rstrip() # elimina el \n
        lista.append(linea)

    cadena = ' '.join(lista) # convierte la lista en cadena
                             #para separar los números por espacio
    vector = cadena.split() # se convierte en lista de nuevo,
    #los números ya están separados por espacios
    res = [] # resultado
    rt ='H/caso'+str(caso) + '/sensores' # donde se guardarán
    #los sensores de desnivelación
    if not os.path.isdir(rt): # si ese directorio no existe,
                              #entonces se crea
        os.makedirs(rt)

    dat = open(rt + '/sensor'+str(sensor), 'w')

    for x in range(len(vector)):
        if x%2 !=0: # no se toma en cuenta el tiempo
            res.append(round(float(vector[x]),prs)) # guarda los sensores
            #en la lista res.
            dat.write(str(round(float(vector[x]),prs))) # escribe los
            #sensores en la carpeta 'sensores' sin tomar en cuenta el tiempo
            dat.write('\n')

    dat.close()
    a.close()
    return res, sensor, caso

def mean(a):
```

```
    # esta función resta el promedio a toda la
    #data que recibe, de esta manera cuando se
    #plotea el gráfico se muestra en cero.
    l = len(a)
    s = 0
    for x in a:
        s = s + x
    nm = s/l # media

    r = []
    for x in a:
        r.append(x-nm)
    return r

def prom(a):
    # esta función calcula el promedio de la
    # data entregada y lo devuelve
     le  = len(a)
     su = 0
     for x in a:
         su = x + su
     pr = su/le
     return pr

def escritura(h):
    ruta ='H/caso'+str(caso) + '/alturas'
    if not os.path.isdir(ruta): # si ese directorio
                                #no existe, entonces se crea
        os.makedirs(ruta)

    data = open(ruta + '/h'+str(sensor), 'w')
    del h[0] # elimina la primera altura de ola para que no contamine la data
    for x in h:
        # escribe las altura de ola en un archivo
        data.write(str(round(x,prs)))
        data.write('\n')

    print('Cantidad de alturas de ola H:', len(h))
    data.close()
    return h

def cruce(a,esc):
    # cruces ascentes por cero para conocer la altura de ola
    #a = cp(a) # corrección parabólica
    #a = cl(a)
    a = mean(a)
    delta = a[:]
    arriba = []
    tmx = []
```

```
tmn = []
abajo = []
h=[] # altura de ola
c = 0 # contador
tmax = []
tmin = []
vmin = []
vmax = []
alt = 0
for x in range(len(a)):
    ma = 0
    mi = 0

    while a[c]>0 and c < len(a)-1:
        #print 'arriba'
        #print 'a[',c,']:',a[c]
        if a[c]*a[c+1]>0:
            arriba.append(a[c])
            tmx.append(c)

        if a[c+1]<0 and a[c-1]<0:
            # Caundo soloo hay un dato positivo
            arriba.append(a[c])
            tmx.append(c)

        c+=1
    while a[c]<0 and c < len(a)-1:
        #print 'abajo'
        #print 'a[',c,']:',a[c]

        if a[c]*a[c+1]>0:
            abajo.append(a[c])
        if a[c-1]>0 and a[c+1]>0:
            # cuando solo hay una dato bajo la media
            abajo.append(a[c])
        c+=1

    if a[c]==0.0:
        c+=1
    if arriba != [] and abajo !=[]:
        h.append(max(arriba)- min(abajo)) # altura de ola

        #print 'h:', max(arriba)-min(abajo)
        s=0 # este es un índice para encontrar el t
        vmax.append(max(arriba))

        for dt in tmx:
            #print('dt',dt)
            if a[dt] == max(arriba):
```

```
                    #print('max',dt)
                    tmax.append(dt*0.1)

                    break

            #print('otro cruce')
            tmx = []
            vmin.append(min(abajo))
            arriba = [] # se vacían las listas para trabajar con cada periodo
            abajo = []

        else:
            pass # si no hay datos de desnivelaciones
    if esc == 1:
        # si es true, tonces se escribe
        escritura(h)
    else:
        return tmax,vmax,tmin, vmin

for x in range(1,2303):
    caso = '01'
    print('Sensor ',x)
    w = vector(caso, x)[0]
    sensor = vector(caso, x)[1]
    cruce(w,1) # altura  en el sensor, el 1
    #plt.plot(range(len(w)),w) # plot para ver todos los sensores juntos
p = open('Propiedades','w')

def an(a):
    vct = vector(caso,int(a))[0] # data original
    tmx,vmx,tmin,vmin  = cruce(vct,0)
    print('Nivel medio del sensor',a,':',round(prom(vct),prs) )
    prm = mean(vct) # data centrada en el cero
    #plot2(tmx,vmx,'black')
    plot(prm,10,a)

lista = [14,45]

lt =open('lista','w') # los sensores a analizar
for x in lista:
    lt.write(str(x))
    lt.write('\n')

lt.close()

print('-'*50)
print('-'*50)

for x in lista:
```

```
    an(x)

plt.legend()
plt.show()
p.write(caso)
p.write('\n')
p.write(str(prs))
print('Caso:', caso)
p.close()
os.system('python serieV2.py')
```

E.1.2. Script 02

```
# -*- coding: cp1252 -*-


'''
calcula parámetros de la TLO para distintos puntos
además grafica los sensores
este código trabaja con las alturas de ola ya entregadas.

- Altura significativa
- Altura media
- Grafíca los resultados
- Grafíca la TLO con su respectiva propagación

'''

import math as mt
import matplotlib.pyplot as plt


print(100*'-')
print('\t \t \t \t \t'+'Script Serie')
print(100*'-')


H0  =  0.1 # Altura de ola a propagar
d   =  0.7 # Profundidad inicial
dp  =  d#1.0 #profundidad final
T   =  2.0  # periodo
th0 =  0.0 # ángulo de incidencia del oleaje
#---------------------------------------
#               Otros+
largop = 4.624 # largo plano del canal
lpend = 0.0 # Largo horizontal en zona de pendiente


#pendiente = False # el canal tiene pendiente
```

```
esp = 0.1# Espacio entre sensores
x1 = 0.1 # sensor inicial en la distancia horizontal

sens = int((largop+lpend-x1)/esp)

print('Cantidad de sensores', sens)

#----------------------------------------
#           Propagación




class Onda:

    def __init__(self):

        self.g = 9.81
        self.T = T

    def londa(self,d ):

        self.d = d # profundidad en el punto
        self.L = (self.g*(self.T**2))/(2*mt.pi)

        for x in range(1000):
            self.L =((self.g*self.T**2)/(2*mt.pi))*...
                ...mt.tanh((2*mt.pi*self.d)/(self.L))
        return self.L



class Graph:
    #graficador
    def __init__(self, x , y, label):
        self.x = x
        self.y = y
        self.label = label
        #self.xx = []
        #for  z in self.x:
        #    self.xx.append(z/1.0)(dp-d)/( lpend-largop ))*...
            ...(xx-largop)+d


    def plot(self):
        plt.plot(self.x, self.y, label = self.label)
```

```
        plt.xlabel('Distancia [m]')
        plt.ylabel('H [m]')
        plt.ylim(0.04,0.14)

    def sca(self):
        # los puntitos
        plt.scatter(self.x, self.y, s=10)
        #plt.xlabel('Distancia[m]')
        #plt.ylabel('Datos [m]')



class Propagation(Onda,Graph):

    def __init__(self):
        self.lista = [] # lista de profundidades
        self.esp = 0.1#esp # espacio entre cada punto en que
        # se calcula hp mediante la  TLO.
        self.sensm = lpend/esp # sensores en pendiente
        self.dv = [] # propagación
        self.d = d # profundidad inicial
        self.sens = sens # cantidad de snsores
        self.th0 = (mt.pi * th0)/180 # ángulo inicial de incidencia
        # del oleaje, en radianes
        self.c0 = Onda().londa(d)/T # celeridad  antes de propagar
        # el oleaje
        self.hp = [] # altura de ola propagada
        self.ex = []
        self.snp = self.sens - self.sensm # cantidad de sensores en
        # la zona plana
        self.dt = 0.1 # intervalo de escritura

        #for x in range(self.sens): # eje x, siempre debe ir.
        #    self.ex.append(x1+self.esp*x)


    def clear(self):

        self.ks = [] #Shoaling
        self.ci = [] # celeridad en el punto i
        self.cg = [] #Celeridad de grupo
        self.th = [] # lista de ángulos de propagación
        self.kr = [] # coeficiente de reflexión
        self.dv = [] # profundidad en el punto  propagarse



    def ec(self, xx):
        # ecuación para generar la pendiente
```

```
    fx = 0
    if xx <= largop: # mientras se trate de sensores e la zona plana
        fx = self.d

    else: # caundo los sensores estén en pendiente
        fx = ((dp-d)/(lpend))*(xx-largop)+d
        #self.lista.append(fx)



    return fx  #

def calc(self,xdv ):
    dv =xdv

    for x in range(len(self.dv)):

        self.ci.append(Onda().londa(dv[x])/T) # la celeridad
        # en cada punto
        k = (2*mt.pi)/Onda().londa(dv[x])
        n = 0.5*(1 + (2.0*k*dv[x])/(mt.sinh(2*k*dv[x])))
        # factor "n" de la celeridad

        self.cg.append(n*self.ci[x]) # celeridad antes de
        # la propagación.

    for x in range(len(dv)-1): # -1 porque tiene uno de más
    # correspondiente a c0

        self.th.append(mt.asin( (self.ci[x]*...
            ...mt.sin(self.th0) )/self.c0 ))
        self.kr.append( ( (mt.cos(self.th0))/...
            ...(mt.cos(self.th[x]))  )**(0.5))
        self.ks.append(mt.sqrt(self.cg[0]/self.cg[x+1])) # shoaling
        self.hp.append(H0*self.ks[x]*self.kr[x])



def param(self):
    pendiente = True

    self.clear()
    if pendiente == True:

        self.esp = esp # espacio entre sensores

        self.clear() # limpia las variables para calcularlas
        # en pendiente
```

```
            self.dv.append(d) # la primera profundidad es al
            #incicio de la prpagación.
            for x in range(int((self.sens/self.dt)*esp)):
            # se agregan los valores al eje x
                self.ex.append(x1+self.dt*x) # Revisar esto,
                # solo el esp hace que la tlo llegue hasta el final
                self.dv.append(self.ec(x*self.dt)) # se genera
                #la pendiente de acuerdo al método ecuación


            self.calc(self.dv)

        Graph(self.ex, self.hp, 'TLO').plot() # grafíca la TLO

        return self.hp


class Sign:
    def __init__(self,  a):
        # calcula la altura significativa
        self.a = a
        del self.a[0]
        self.a.sort(reverse=True) # orden de mayor a menor




        self.la = len(self.a) # longitud de a
        self.c = int(( 1.0/3.0)*self.la)
        self.s = 0.0 # suma
        self.v = [] #

    def hs(self):
        for x in range(self.c):
            self.v.append( self.a[x] )

        if self.c ==0:
            self.c = 1
            print('Error: alturas insuficientes')
        for x in self.v:
            self.s = self.s+x
        res = self.s/self.c # esultados
        #print('Altura significativa:',res)
        return res

class Mean:
    # calcula la altura media de la ola
    def __init__(self, d):
```

```
        self.d = d # datos de entrada
        self.mn = [] # nivel medio

       def nvm(self):


        le = len(self.d)
        su = 0 # suma los datos

        for x in self.d:
            su = su + x

        self.mn = su/le # promedio de las alturas de ola
        #print('Altura media de ola:', round(self.mn,4))
        return self.mn


class Clean:
    def __init__(self, v):

        self.v = v
        self.lista = []
        self.l2 = []
        self.res = []
    def cl(self):
        for linea in self.v:
            linea = linea.rstrip() #Elimina
            self.lista.append(linea)

        self.cadena = ' '.join(self.lista) # convierte la lista
        # en cadena para separar
      #los números por espacio
        self.l2 = self.cadena.split() # Se convierte en lista de
        # nuevo, los números ya están separados por espacio

        for x in self.l2:
            # pasar los datos a enteros
            self.res.append(int(x))


        return self.res


class Archivos(Sign):

    # Esta función convierte los datos de un arcvhivo en un vector,
    # eliminando el tiempo
    def __init__(self, caso, c):
```

```
        self.caso = caso
        self.c = c # la altura a analizar
        self.f = open('H/caso'+str(self.caso)+...
            ...'/alturas/h'+str(self.c),'r') # alturas
        self.lista = [] # Se guarda el resultado

        #self.p = open('Propiedades', 'r')
        #self.dato = []
        #for x in self.p:
        #    x = x.rstrip() # elimina el el \n parece...
        #    self.dato.append(x)

    def vector(self):

        for linea in self.f:
            linea = linea.rstrip() #Elimina
            self.lista.append(linea)

        self.cadena = ' '.join(self.lista) # convierte la lista
        # en cadena para separar los números por espacio
        self.v = self.cadena.split() # Se convierte en lista de nuevo,
        # los números ya están separados por espacio



        self.res = []# resultado de las alturas de ola de cada sensor

        for x in range(len(self.v)):
            self.res.append(float(self.v[x]))

        #-----------------------------------
        #En esta parte del código se eliminan la primera y última ola


        #-----------------------------------


        self.hs = Sign(self.res).hs() # llama a la altura significativa
        self.mn = Mean(self.res).nvm() # llama al objeto nivel medio


    def fin(self):
        self.f.close()
        return self.hs, self.mn, self.res



class Rev():
    # este objeto sirve para revisar la altura significativa para
```

```
    # un sensor en particular
    def __init__(self,caso,sensor):
        self.caso = caso
        self.sensor = sensor
        self.prs = Omega().prs

    def call(self):

        ob = Archivos(self.caso,self.sensor)
        ob.vector()
        a =ob.fin()[2]
        hs = Sign(a).hs() # calcular la altura significativa para
        # un solo sensor

        print('Hs para el sensor '+str(self.sensor)+...
            ...':',round(hs*100,self.prs),'[cm]' )

        return round(hs,self.prs)



class Omega():
    def __init__(self):
        self.x1 = x1
        self.hs = []
        self.mn = []
        self.lista = [] # lista de propiedades
        self.prop = [] # las propiedades, caso, precisión....
        self.f = open('Propiedades','r')

        for linea in self.f:
            linea = linea.rstrip() #Elimina
            self.lista.append(linea)

        self.cadena = ' '.join(self.lista) # convierte la lista
        # en cadena para separar los números por espacio
        self.prop = self.cadena.split() # Se convierte en lista de
        # nuevo, los números ya están separados por espacio



        self.caso = self.prop[0]
        self.prs = int(self.prop[1])


        self.f.close()
        self.arc = 0 # defino arc, solo eso
        self.ex = [] # eje x para Hs
```

```
    def alfa(self):
        # función para llamar las alturas significativas del otro objeto
        # grafíca los datos

        for x in range(1, sens+1):
            self.arc = Archivos(self.caso, x) # Le doy como
            #dato de entrada el caso y el número de sensor
            #al objeto Archivos
            # entonces self.arc es ahora un objeto archivos

            self.arc.vector() # llamo al método vector del
            #objeto arc para que se calcula hs y mn
            self.hs.append(round(self.arc.fin()[0],self.prs))
            # altura significativa
            self.mn.append(self.arc.fin()[1]) # nivel medio

        f = open('H/caso'+str(self.caso)+'/hs','w') # alturas
        s = 1 # sensor
        for x in self.hs:
            # este for gruada los datos hs en un archivo
            f.write(str(s)) #  sensor
            f.write('\t')
            f.write(str(round(x,self.prs))) # hs
            f.write('\n')
            s +=1

        f.close()


        self.sens = Propagation().sens # para conocer los
        # sensores que se usan

        for x in range(int(self.sens)):

            self.ex.append(self.x1 + esp*x)


        #llamado a la TLO
        tlo = Propagation()
        tlo.param()
        #c = 1
        #for x in self.hs:
        #    print('sensor, Hs:',c,x)
        #    c+=1

        Graph(self.ex, self.hs, 'Sensores [Hs]').plot() # dibuja
        # la linea que forman los sensores
        Graph(self.ex, self.hs, 'A').sca() # dibuja los puntos
```

```
        # de los sensores

        #Graph(self.ex, self.mn, 'Alturas medias').plot()
        plt.legend()

        lt = open('lista','r')

        lista = Clean(lt).cl() # llamo a la clase Clean para
        #que convierta el archivo en una lista de enteros
        lt.close()

        r = [] # lista de resultados
        for x in lista:


            r.append( Rev(self.caso,x).call())



        print('diferencia entre sensores:',...
            ...abs(round((r[0]-r[1])*100*10,self.prs)),'[mm]')


ob = Propagation()

o2 = Omega()

o2.alfa()

plt.grid()
plt.show()
```

E.2. Scripts para el análisis de las presiones

A continuación, se presenta la metodología utilizada para procesar los datos de presiones generadas por el oleaje sobre la estructura de la sección 3.5.6.

Conociendo la cantidad de sensores de presión ($pSens$) puestos en un canal, se le puede indicar al script cuantas series de datos debe copiar desde el caso de estudio y pegarlas en una carpeta para su posterior análisis. Además, es necesario indicarle al script el tiempo ($dtif$) que se demora la onda en llegar al muelle desde el sensor de desnivelaciones, ya que este intervalo de tiempo toma relevancia en los cálculos posteriores.

En un segundo script llamado **sPresiones.py** se encuentra una función llamada $Presiones$, en la cual se calcula la máxima presión de todos los sensores. Después se busca en qué sensor y tiempo ocurre esta máxima presión y se guarda en el archivo llamado pm.

En un tercer script llamado **desnivelaciones.py** se toma el archivo del único sensor de desnivelaciones, se copia y se guarda en una carpeta para el procesamiento de datos mediante la función *Save*().

En un script llamado **stokesII.py** se calculan las desnivelaciones de la teoría de Stokes II. Se abre el sensor de desnivelaciones y se compara con Stokes II, dando como resultado las Figuras 3.25, 3.26 y 3.27.

Un último script llamado **final.py** abre el archivo *pm* y se extrae el valor *tObjetivo*, el cual es una variable de tiempo en segundos, esta variable es la resta entre el tiempo en que ocurre la máxima presión sobre el muelle (t) y el tiempo que se demora la onda que provocó esa máxima presión en llegar desde el sensor de desnivelaciones hasta el muelle ($dtif$). Este valor ($dtif$) se obtiene a partir del video generado luego de obtener los datos de modelación, normalmente los valores varían entre 5.0 y 5.2 s. Después, sabiendo el tiempo en que la ola que generó la máxima presión pasó por el sensor (H_p) se puede buscar esa altura de ola mediante la función *AlturaObjetivo*(). Una vez encontrado H_p se llama al objeto *Plotter*(), el cual genera la gráfica presentada en la Figura **??**.

E.2.1. Script presiones.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import os

'''
-Copia la presiones sin tomar en cuenta el tiempo
-redondea las presiones

'''

#----------------Variables-----------------------

prs = 5 # precisión (en decimales) para la altura de ola y los
#sensores
print(30*'-.-')
print('\t \t \t presiones')
print(30*'-.-')

hr = []  # altura de ola relaiva de rotura

class Archivos:

    def __init__(self):
        #Toma los sensores, le quita el tiempo y los guarda en
        #el directorio caso/H/presiones
        pass

    def vector(self,caso, sensor):
```

```
        a = open('../'+str(caso)+'/data/presiones/pSensor'...
            ...+str(sensor),'r')
        # esta función convierte un archivo en una lista
        lista = [] # donde se guarda el resultado
        # esta función convierte los datos de un archivo en un vector
        # esta función solo devuelve la profundidad, no el tiempo

        for linea in a:
            linea = linea.rstrip() # elimina el \n
            lista.append(linea)

        cadena = ' '.join(lista) # convierte la lista en cadena para
        #separar los números por espacio
        vector = cadena.split() # se convierte en lista de nuevo,
        # los números ya están separados por espacios
        res = [] # resultado
        rt ='H/caso'+str(caso) + '/presiones' # donde se guardarán
        # los sensores de desnivelación

        if not os.path.isdir(rt): # si ese directorio no existe,
        # entonces se crea
            os.makedirs(rt)

        dat = open(rt + '/pSensor'+str(sensor), 'w')

        for x in range(len(vector)):
            if x%2 !=0: # no se toma en cuenta el tiempo
                res.append(round(float(vector[x]),prs)) # guarda los
                # datos del sensor en la lista res.
                dat.write(str(round(float(vector[x]),prs))) # escribe
                # los sensores en la carpeta 'sensores'
                #sin tomar en cuenta el tiempo
                dat.write('\n')

        dat.close()
        a.close()
        return res, sensor, caso

sens = 35 # cantidad de sensores

for x in range(1,sens+1):
    caso = '01'
    print('Sensor de presión ',x)
    Archivos().vector(caso, x)[1]

print('\t \t \t \t Hecho')
print(30*'--')
```

E.2.2. Script spresiones.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import matplotlib.pyplot as plt
import os

'''
--
'''

print(30*'-.-')
print('\t \t \t spresiones')
print(30*'-.-')


os.system('rm pm')

class Graph:

    #Esta clase crea objetos que grafican las curvas
    def __init__(self, x , y, label=''):
        self.x = x
        self.y = y
        self.label = label

    def plot(self, lx ='', ly=''):
        # este método grafíca curvas
        plt.plot(self.x, self.y, label = self.label)
        plt.xlabel(lx)
        plt.ylabel(ly)
        #plt.ylim(0,0.2)

    def sca(self, lx ='', ly='' ):
        # este método grafíca puntos
        plt.scatter(self.x, self.y, label = self.label, s=10)
        plt.xlabel(lx)
        plt.ylabel(ly)


class Archivos:

    # Esta clase crea objetos que convierten los datos de
    #un archivo en un vector.
    def __init__(self, caso, c):
        self.caso = caso
```

```
        self.c = c # la altura a analizar
        # sensores de presión:
        self.f = open('H/caso'+str(self.caso)+...
            .../presiones/pSensor'+str(self.c),'r')
        self.lista = [] # Se guarda el resultado
        self.vector()

    def vector(self):
        # Este método crea un vector a partir de un archivo

        for linea in self.f:
            linea = linea.rstrip() #Elimina
            self.lista.append(linea)

        self.cadena = ' '.join(self.lista) # convierte la lista
        #en cadena para
        #separar los números por espacio
        self.v = self.cadena.split() # Se convierte en lista de nuevo,
        # los números ya están separados por espacio
        self.res = []# resultado de las alturas de ola de cada sensor

        for x in range(len(self.v)):
            self.res.append(float(self.v[x]))

    def fin(self):
        self.f.close()
        return self.res

class Herr:
    #clase que genera objetos que son herramientas de análisis
    def __init__(self,a):
        self.a = a

    def mx(self):
        mx = max(self.a)
        mi = min(self.a)

        if mi>mx:
            print('Error: la presión negativa es mayor a la presión...
                ...positiva')
            return 0
        else:
            return max(self.a)

class Nma:
    # nivel medio del agua
    def __init__(self,long):
        self.long = long
```

```
        x = []
        y = []
        for i in range(self.long+1):
            x.append(i)
            y.append(1.0)

        Graph(x,y).plot()

class Propiedades:

    # Esta clase crea un objetos que leen las propiedades
    # de un archivo externo y retornan los resultados

    def __init__(self, archivo):
        self.prs = 5
        self.archivo = archivo
        self.f = open(str(self.archivo),'r')

    def datos(self):
# El método datos convierte el archivo en un vector
#de valores
        lista = []

        for linea in self.f:
            linea = linea.rstrip() # elimina el \n
            lista.append(linea)

        cadena = ' '.join(lista) # convierte la lista en
        #cadena para separar
        #los números por espacio
        vector = cadena.split() # se convierte en lista de nuevo,
        # los números
        #ya están separados por espacios
        res = [] # resultado

        for x in range(len(vector)):
            res.append(round(float(vector[x]),self.prs)) # guarda los
            # datos del
            #sensor en la lista res.

        print('Propiedades heredadas:',res)


        self.f.close()
        return res[0]


class Casos: # presiones para el diagrama
```

```
# Esta clase crea objetos que trabajan con las presiones
def __init__(self,caso,sens,dtif):
    self.sens = sens #sensores
    self.caso = caso # caso de estudio
    self.dtif = dtif #[s] intervalo de tiempo en que la onda
    # se demora en
    # llegar desde el inicio de la rampa hasta el muelle
    self.presionesMx() # este método devuelve la
    #ubicación de la máxima
    #presiOn de entre todos los sensores
    self.hPres() # se llama al método hPres()

def presionesMx(self):
    # este método genera una lista de la máxima presión
    # por cada sensor
    lpMax  = [] # lista de las máximas presiones
    for i in range(1,self.sens+1):
        r = Archivos(self.caso,i).fin() # se abre la lista
        # de datos
        #(el sensor) i
        pMax = Herr(r).mx() # presión máxima encontrada en
        # la lista sensor
        lpMax.append(pMax) # lista de las presiones máximas

    pMaxt = Herr(lpMax).mx() # presión máxima teniendo en cuenta
    # todos los
    #sensores
    #-------------------------------------------
    #   conocer qué sensor tiene esa máxima presión
    cont = 1 # contador de sensores

    for i in range(1,self.sens+1):
        r = Archivos(self.caso,i).fin() # se abre la lista de datos
        #(el sensor) i
        res = Herr(r).mx() # se crea el objeto res de la clase Herr()

        if pMaxt == res:
            print('Sensor encontrado:',cont,'valor:',res)
            ind = cont # se agrega el valor del índice para el
            #caso del sensor que tiene la presión más alta

        cont+=1
    #-------------------------------------------
#   conocer en qué tiempo ocurre esta máxima presión
    vof = Archivos(self.caso,ind).fin() # se abre la lista del
    # sensor que tiene la máxima presión
    dt = 0.1 # pasos de tiempo

    for i in range(len(vof)):
```

```
            t = (dt*i+0.1) # este 0.1 es porque el tiempo no parte
                    # en cero en el sensor

            if vof[i] == pMaxt:
                #tiempo = t # se guarda el valor en la variable tiempo
                return t, ind # tiempo en que ocurre la máxima presión,
                #  índice del sensor que tiene la máxima presión

    def hPres(self):
        # este método busca la onda que generó esa presión
        f = open('pm','a') # se abre un archivo para guardar
        # el indice de la máxima presión encontrada
        t, ind = self.presionesMx() # tiempo en que ocurre
        # la máxima presión
        print('caso',self.caso,'\t t en rampa',round(t,3))
        tObjetivo = round(t-self.dtif,5) # el tiempo objetivo
        # para encontrar
        #la onda que generó la presión máxima
        f.write('caso:\t'+ str(self.caso)+'\t')
        f.write('tiempo:\t'+str(tObjetivo)+'\n')
        f.close()
        #os.system('python etas.py')

Casos('01',31,5.2)   # caso, sensor, dtif
Casos('02',31,5.2) # estos son objetos de la clase Casos()

print('\t \t \t \t Hecho')
print(45*'--')
```

E.2.3. Script desnivelaciones.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import os

'''
-Copia los sensores de superficie libre sin tomar en
 cuenta el tiempo
-

'''

#----------------Variables-----------------------

prs = 5 # precisión (en decimales) para la altura de
# ola y los sensores
```

```
print(30*'-.-')
print('\t \t \t \t Desnivelaciones')
print(30*'-.-')

hr = []  # altura de ola relativa de rotura

class Archivos:

    def __init__(self):
        #Toma los sensores y le quita el tiempo
        pass

    def vector(self,caso, sensor):
        a = open('../'+str(caso)+'/data/desniveles/data_sensor'...
            ...+str(sensor),'r')
        # esta función convierte un archivo en una lista
        lista = [] # donde se guarda el resultado
        # esta función convierte los datos de un archivo en un vector
        # esta función solo devuelve la profundidad, no el tiempo

        for linea in a:
            linea = linea.rstrip() # elimina el \n
            lista.append(linea)

        cadena = ' '.join(lista) # convierte la lista en cadena para
        #separar los números por espacio
        vector = cadena.split() # se convierte en lista de nuevo,
        #los números ya están separados por espacios
        res = [] # resultado
        rt ='H/caso'+str(caso) + '/desniveles' # donde se guardarán
        # los  sensores de desnivelación

        if not os.path.isdir(rt): # si ese directorio no existe,
            # entonces se crea
            os.makedirs(rt)

        dat = open(rt + '/dSensor'+str(sensor), 'w')

        for x in range(len(vector)):

            if x%2 !=0: # no se toma en cuenta el tiempo
                res.append(round(float(vector[x]),prs)) # guarda
                # los datos del sensor en la lista res.
                dat.write(str(round(float(vector[x]),prs)))
                # escribe los sensores en la carpeta 'sensores'
                # sin tomar en cuenta el tiempo
                dat.write('\n')
```

```
        dat.close()
        a.close()
        return res, sensor, caso

class Propiedades:

    # lee las propiedades de un archivo externo y retorna los
    # resultados
    def __init__(self, archivo):
        self.archivo = archivo
        self.f = open(str(self.archivo),'r')

    def datos(self):
        lista = []
        for linea in self.f:
            linea = linea.rstrip() # elimina el \n
            lista.append(linea)

        cadena = ' '.join(lista) # convierte la lista en
        # cadena para separar los números por espacio
        vector = cadena.split() # se convierte en lista de nuevo,
        # los números ya están separados por espacios
        res = [] # resultado

        for x in range(len(vector)):

            if x%2 !=0: # no se toma en cuenta el tiempo
            # guarda los datos del sensor en la lista res:
                res.append(round(float(vector[x]),prs))


        print('Propiedades heredadas:',res)
        caso = int(res[0])
        self.f.close()
        return caso,res[1]

#caso,tob = Propiedades('pm').datos() #caso, tObjetivo

sens = 1 # cantidad de sensores
for x in range(1,sens+1):
    caso = '01'
    print('Caso',caso)
    print('Sensor de desnivelaciones ',x)
    sensor = Archivos().vector(caso, x)[1]

print('\t \t \t Hecho')
print(30*'--')
```

E.2.4. Script stokesII.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Feb 14 11:32:05 2019

Este script busca tener la altura de ola adecuada al
pie de la rampa con
respecto a los datos requeridos por el artículo.


@author: carlos
"""
import sys
import math as mt
import matplotlib.pyplot as plt
import os



class Graph:
    def __init__(self, x , y, label ='',xl='',yl=''):
        self.x = x
        self.y = y
        self.label = label
        self.xl = xl # título del eje
        self.yl = yl



    def plot(self):
        plt.plot(self.x, self.y, label = self.label)
        #plt.xlabel('Distancia [m]')
        #plt.ylabel('Hs [m]')
        #plt.ylim(0.0,0.2)


    def sca(self):
        plt.scatter(self.x, self.y,color= 'k', s=5)
        #plt.xlabel('Distancia[m]')
        #plt.ylabel('Datos [m]')

    def subplot(self,com,sc): #sc = scatte
        self.sc = sc
        self.com = com
```

```
        if self.com ==1:
            self.com = 211
            self.letra = 'a)'

        elif self.com ==2:
            self.com = 212
            self.letra = 'b)'

        else:
            print('Error en subplot [1,2] , comando:',self.com)
            sys.exit()
        plt.subplot(self.com)
        plt.plot(self.x,self.y, label = self.label)

        if self.sc ==1:
            self.sca()
        else:
            print('Scatter desactivado')
            pass

        plt.xlabel(self.xl)
        plt.ylabel(self.yl)
        #plt.ylim(0.04,0.14)
        #plt.text(4.6, 0.12,self.letra , fontsize=15)

class Tlo:

    def __init__(self,d,T):
        self. g = 9.81
        self.t = T
        self.d = d

    def londa(self, sw = 0): # sw swish

        g = self.g
        T = self.t
        d = self.d
        L = (g*(T**2))/(2*mt.pi)

        for x in range(1000):
            L =((g*T**2)/(2*mt.pi))*mt.tanh((2*mt.pi*d)/(L))

        hmax = 0.14*L*mt.tanh((2*mt.pi*d)/(L))

        if sw ==0:
            pass
        else:

            print('Longitud de onda', round(L,3),'[m]')
```

```
            print('Hmax:', round(hmax,3), '[m]')

            if d/L>0.5:
                print('Aguas profundas')

            elif 1.0/20.0<d/L and  d/L<0.5:

                print('Aguas intermedias')
            elif d/L<1.0/20.0:
                print('Aguas someras')

        return L


class Cruces:

    def __init__(self,a,caso,sensor,esc =1):
        self.a = a # lista de desnivelaciones
        self.esc = esc # escritura
        self.sensor = sensor
        self.caso = caso
        self.prs = 5 # precisión

    def mean(self,a):
        #esta función resta el promedio a toda la data que recibe,
        #de esta manera cuando se plotea el gráfico la data se
        #muestra centrada en cero.
        l = len(a)
        s = 0

        for x in a:
            s = s + x
        nm = s/l # media
        r = []
        for x in a:
            r.append(x-nm)

        return r

    def escritura(self,h):
        caso = self.caso
        ruta ='H/caso'+str(caso) + '/stokesII'

        if not os.path.isdir(ruta): # si ese directorio no existe,
        # entonces se crea
            os.makedirs(ruta)

        data = open(ruta + '/h'+str(self.sensor), 'w')
        del h[0] # elimina la primera altura de ola para
```

```
        # que no contamine la data

        for x in h:
            # escribe las altura de ola en un archivo
            data.write(str(round(x,self.prs)))
            data.write('\n')

        print('Cantidad de alturas de ola H:', len(h))
        data.close()

        return h

    def rounder(self,a,prs=3):
        salida = []

        for i in a:
            salida.append(round(i,prs))

        return salida

    def cruce(self):
        a = self.a # el sensor de desnivelaciones
        esc = self.esc
        # cruces ascentes por cero para conocer la altura
        # de ola
        a = self.mean(a) # los datos se centran en cero
        #delta = a[:]
        arriba = []
        tmx = [] #lista de tiempos para los datos positivos
        # de la onda
        abajo = []
        h=[] # altura de ola
        c = 0 # contador
        tmax = []
        tmin = []
        vmin = [] # valores máximos y mínimos
        vmax = []

        for x in range(len(a)):
            #ma = 0
            #mi = 0

            while a[c]>0 and c < len(a)-1:
                # mientras los datos del sensor  sean mayores a
                # cero y además no se trate del último dato
                #-----Parte de arriba de la onda----

                if a[c]*a[c+1]>0:
                    # sí el punto actual y el siguente son mayores
```

```
        # a cero
        arriba.append(a[c]) # los datos se guardan en
        #la lista
        #"arriba"
        tmx.append(c)

    if a[c+1]<0 and a[c-1]<0:
        # Cuando solo hay un dato positivo
        arriba.append(a[c])
        tmx.append(c)
    c+=1 # se van analizando los datos de la lista 'a'

while a[c]<0 and c < len(a)-1:
    # Mientras que el dato actual sea negativo y que
    #además no se trate del último dato, entonces:

    if a[c]*a[c+1]>0:
        # si tanto el dato actual 'a[c]' como el dato
        #siguiente son negativos, entonces:
        abajo.append(a[c]) # el dato actual se va
        #guardando en una lista llamada "abajo"

    if a[c-1]>0 and a[c+1]>0:
        # si tanto el dato anterior como el siguiente
        #son positivos es decir, solo hay un dato
        #negativo , entonces:
        abajo.append(a[c])

    c+=1

if a[c]==0.0:
    # si el dato actual es igual a cero, entonces no
    # se considera
    c+=1

if arriba != [] and abajo !=[]:
    # sí hay datos negativos y positivos, entonces:
    h.append(max(arriba)- min(abajo)) # altura de ola
    #print 'h:', max(arriba)-min(abajo)
    #s=0 # este es un indice para encontrar el t
    vmax.append(max(arriba)) # valor máximo

    for dt in tmx:

        if a[dt] == max(arriba):
            tmax.append(dt*0.1)

            break
```

```
                #print('otro cruce')
                tmx = []
                vmin.append(min(abajo))
                arriba = [] # se vacían las listas para trabajar con
                # cada periodo de onda
                abajo = []

            else:

                pass # si no hay datos de desnivelaciones

        del tmax[0] # se elimina el primer tiempo porque no
        # se considera la
        # primera altura de olas

        if esc == 1:
            # si es true, tonces se escribe
            self.escritura(h)
            tmax = self.rounder(tmax) # redondea una lista
            return h,tmax,vmax,tmin, vmin

        else:
            tmax = self.rounder(tmax) # redondea una lista
            return h,tmax,vmax,tmin, vmin

class Archivos:

    # Esta función convierte los datos de un arcvhivo en un
    # vector.
    def __init__(self, caso):
        self.sensor = 1 # solo hay un sensor de desnivelaciones
        self.caso = caso
        print('Alturas del caso:',caso)
        self.f = open('H/caso'+str(self.caso)+'/alturas/h'...
            ...+str(self.sensor),'r')
        ## sensores de desnivel
        self.lista = [] # Se guarda el resultado
        self.vector()

    def vector(self):

        for linea in self.f:
            linea = linea.rstrip() #Elimina
            self.lista.append(linea)

        self.cadena = ' '.join(self.lista) # convierte la lista
        # en cadena para separar los números por espacio
        self.v = self.cadena.split() # Se convierte en lista
        # de nuevo, los números ya están separados por espacio
```

```
        self.res = []# resultado de las alturas de ola de cada
        # sensor

        for x in range(len(self.v)):
            self.res.append(float(self.v[x]))

    def fin(self):
        self.f.close()

        return self.res

class sensEta:

    # Esta lcase crea objetos que convierten los datos de un
    # arcvhivo en un vector.
    def __init__(self, caso):
        self.sensor = 1 # solo hay un sensor de desnivelaciones
        self.caso = caso
        print('Alturas del caso:',caso)
        self.f = open('H/caso'+str(self.caso)+'/desniveles...
            .../dSensor'+str(self.sensor),'r')
        # sensores de desnivel
        self.lista = [] # Se guarda el resultado
        self.vector()

    def vector(self):

        for linea in self.f:
            linea = linea.rstrip() #Elimina
            self.lista.append(linea)

        self.cadena = ' '.join(self.lista) # convierte la lista
        #en cadena para separar los números por espacio
        self.v = self.cadena.split() # Se convierte en lista
        # de nuevo, los números ya están separados por espacio
        self.res = []# resultado de las alturas de ola de cada
        # sensor

        for x in range(len(self.v)):
            self.res.append(float(self.v[x]))

    def fin(self):

        self.f.close()
        return self.res

class StokesII:

    def __init__(self,H0,T,caso):
```

```
        self.caso = caso
        self.a = H0/2.0 # amplitud de onda
        self.h0 = H0
        self.T = T
        self.d = 0.635 # profundidad
        self.tiempo = 40 #segundos de modelación
        self.L = Tlo(self.d,self.T).londa()
        self.dn = []
        self.ex = [] # eje x
        self.f = []
        self.z = []
        self.k = (2.0*mt.pi)/self.L # número de onda
        self.w = (2.0*mt.pi)/self.T  # frecuencia angular

    def rounder(self,a,prs=3):
        salida = []
        for i in a:
            salida.append(round(i,prs))

        return salida

    def retSt(self):
        k = self.k
        w = self.w
        d = self.d
        self.time = []
        self.dstokes = []

        for z in range(int(self.tiempo*100)):
            self.ex.append(0.01*z)
            x = 0.0#(3*mt.pi)/2 # desfase
            df = 0.0#mt.pi/4.0
            t = z*0.01
            #f0 = (H0)*mt.cos(k*x*0.01) # cecuacion para amplitudes
            # de onda

            self.time.append(t)
            f1 = (self.a*mt.cos(w*t+k*x+df) + k*(self.a**2)*...
                ...((mt.cosh(k*d))/(4*mt.sinh(k*d)**3)) *...
                ...(2+mt.cosh(2*k*d))*mt.cos(2*(w*t+k*x+df)) )
            #d = 0.0 # se analiza en el nivel medio
            self.dn.append(f1) # d+f1
            self.dstokes.append(f1+d)

        return self.dn, self.dstokes

    def etas(self,pos):
        self.pos = pos # posición
        self.s1 = sensEta(self.caso).fin()
```

```
        del self.s1[-1]
        self.n1 = []
        self.dt1 = 0.1 # intervalo de escritura

        for t in range(len(self.s1)): # se redondea en 0 para que
        #el entero redondee correctamente
            self.n1.append(t*self.dt1)

        dstokes = self. retSt()[1]
        Graph(self.time, dstokes,'Stokes II','t [s]',...
            ...'$\eta$ [m]').subplot(pos,0)
        Graph(self.n1, self.s1,'sensor','t [s]',...
            ... '$\eta$ [m]').subplot(pos,0)
        plt.legend(loc=3)
        plt.grid()

    def alturas(self,pos): # el sensor
        self.pos = pos # posición del sensor
        sens = 1 # sensor, solo importa para el nombre del archivo
        sdn = self.retSt()[0]
        h = Cruces(sdn,self.caso,sens,1).cruce()[0]
        del h[-1]
        y = []
        tt = [] # lita de periodos

        for i in range(len(h)):
            tt.append(i*self.T)
            y.append(self.h0)

        #Graph(tt,y,'Altura requerida').plot()
        Graph(tt,h,'Alturas de onda de StokesII', 't [s]',...
            ...'H [m]').subplot(self.pos,0)
        #--------------------- Agregando las ondas-----------------------
        dns = sensEta(self.caso).fin() # se sacan las desnivelaciones
        hSens = Cruces(dns,self.caso,sens).cruce()[0]
        del hSens[0]
        tt = []

        for i in range(len(hSens)):
            tt.append(i*self.T)

        Graph(tt,hSens,'Alturas de onda del sensor',...
                ...'t [s]','H [m]').subplot(self.pos,1)

StokesII(0.095,2.0,'01').etas(1) # H0, T, D, casospresiones.py
StokesII(0.095,2.0,'01').alturas(2) # H0, T, D, caso
plt.grid()
plt.legend(loc=3)
plt.show()
```

E.2.5. Script final.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import matplotlib.pyplot as plt

import os

'''

-El objeto Archivos toma el sensor de desnivelaciones y lo entrega
convertido en una lista.
-El objeto cruces lo convierte en alturas de ola (sin considerar
 la primera altura)  luego guarda esas alturas en un archivo
llamado "alturas".

'''

print(30*'-.-')
print('\t \t \t \t sDesnivelaciones')
print(30*'-.-')

class Graph:
    #graficador
    def __init__(self, x , y, label=''):
        self.x = x
        self.y = y
        self.label = label


    def plot(self, lx ='', ly=''):
        plt.plot(self.x, self.y, label = self.label)
        plt.xlabel(lx)
        plt.ylabel(ly)
        #plt.ylim(0,0.2)

    def sca(self, lx ='', ly='' ):
        # los puntitos

        plt.scatter(self.x, self.y, label = self.label, s=5)
        #, color='k')
        plt.xlabel(lx)
        plt.ylabel(ly)
        #z = []
```

```
        #yy = []

        #for i in range(6):
        #    yy.append(i*0.5)

        #for i in range(13):
        #    z.append(i*10)
        #plt.xticks(z)
        #plt.yticks(yy)


class Cruces:

    def __init__(self,a,caso,sensor,esc =1):
        self.a = a
        self.esc = esc
        self.sensor = sensor
        self.caso = caso

        self.prs = 5 # precisión


    def mean(self,a):
        #esta función resta el promedio a toda la data que recibe,
        #de esta manera cuando se genera el gráfico la data se muestra
        #centrada en cero
        l = len(a)
        s = 0

        for x in a:
            s = s + x
        nm = s/l # media
        r = []
        for x in a:
            r.append(x-nm)
        return r

    def escritura(self,h):
        caso = self.caso
        ruta ='H/caso'+str(caso) + '/alturas'
        if not os.path.isdir(ruta): # si ese directorio no existe,
            #entonces se crea
            os.makedirs(ruta)

        data = open(ruta + '/h'+str(self.sensor), 'w')
        del h[0] # elimina la primera altura de ola para que no
        #contamine la data

        for x in h:
```

```
            # escribe las altura de ola en un archivo
            data.write(str(round(x,self.prs)))
            data.write('\n')

        print('Cantidad de alturas de ola H:', len(h))
        data.close()

        return h

    def rounder(self,a,prs=3):
        salida = []

        for i in a:
            salida.append(round(i,prs))

        return salida


    def cruce(self):
        a = self.a # el sensor de desnivelaciones
        esc = self.esc
        # cruces ascentes por cero para conocer
        #la altura de ola
        a = self.mean(a) # los datos se centran en cero
        arriba = []
        tmx = [] #lista de tiempos para los datos positivos
        #de la onda
        abajo = []
        h=[] # altura de ola
        c = 0 # contador
        tmax = []
        tmin = []
        vmin = [] # valores máximos y mínimos
        vmax = []
        #alt = 0
        for x in range(len(a)):
            #ma = 0
            #mi = 0

            while a[c]>0 and c < len(a)-1:
                # mientras los datos del sensor  sean mayores
                #a cero y además
                # no se trate del último dato
                #-----Parte de arriba de la onda----

                if a[c]*a[c+1]>0:
                    # sí el punto actual y el siguiente son mayores
                    # a cero
                    arriba.append(a[c]) # los datos se
```

```
            #guardan en la lista "arriba"
            tmx.append(c)

        if a[c+1]<0 and a[c-1]<0:
            # Cuando solo hay un dato positivo
            arriba.append(a[c])
            tmx.append(c)
        c+=1 # se van analizando los datos de la lista 'a'
    while a[c]<0 and c < len(a)-1:
        # Mientras que el dato actual sea negativo y que
        # además no se trate del último dato, entonces:

        if a[c]*a[c+1]>0:
            # si tanto el dato actual 'a[c]' como el dato
            # siguiente son negativos, entonces:
            abajo.append(a[c]) # el dato actual
            #se va guardando en una lista
            # llamada "abajo"
        if a[c-1]>0 and a[c+1]>0:
            # si tanto el dato anterior como el
            #siguiente son positivos
            # es decir, solo hay un dato negativo , entonces:
            abajo.append(a[c])
        c+=1

    if a[c]==0.0:
        # si el dato actual es igual a cero, entonces no
        #se considera
        c+=1
    if arriba != [] and abajo !=[]:
        # sí hay datos negativos y positivos, entonces:

        h.append(max(arriba)- min(abajo)) # altura de ola

        #print 'h:', max(arriba)-min(abajo)

        #s=0 # este es un indice para encontrar el t
        vmax.append(max(arriba)) # valor máximo

        for dt in tmx:

            if a[dt] == max(arriba):
                tmax.append(dt*0.1)

                break

        #print('otro cruce')
        tmx = []
        vmin.append(min(abajo))
```

```
                arriba = [] # se vacían las listas para trabajar
                #con cada periodo de onda
                abajo = []

            else:

                pass # si no hay datos de desnivelaciones

        del tmax[0] # se elimina el primer tiempo porque no
        #se considera la primera altura de olas

        if esc == 1:
            # si es true, tonces se escribe
            self.escritura(h)
            tmax = self.rounder(tmax) # redondea una lista
            return h,tmax,vmax,tmin, vmin

        else:

            tmax = self.rounder(tmax) # redondea una lista

            return h,tmax,vmax,tmin, vmin

class Archivos:

    # Esta función convierte los datos de un arcvhivo en un vector.
    def __init__(self, caso, sensor, rt = 'desniveles'):
        if rt =='desniveles':
            cod = '/d'
        else:
            cod = '/p'
        self.v = [] # lista de desniveles
        self.caso = caso
        #print('caso:',caso)
        #sensores de desnivel:
        self.sensor = sensor # El sensor a analizar
        self.f = open('H/caso'+str(self.caso)+...
            .../'+str(rt)+cod+'Sensor'+str(self.sensor),'r')
        self.lista = [] # Se guarda el resultado
        self.vector()

    def vector(self):

        for linea in self.f:
            linea = linea.rstrip() #Elimina
            self.lista.append(linea)

        self.cadena = ' '.join(self.lista) # convierte la lista
        # en cadena para separar los números por espacio
```

```
        self.v = self.cadena.split() # Se convierte en lista
        # de nuevo, los números ya están separados por espacio
        self.res = []#

        for x in range(len(self.v)):
            self.res.append(float(self.v[x]))

    def fin(self):
        self.f.close()
        return self.res

class Propiedades:
    # lee las propiedades de un archivo externo y retorna
    # los resultados
    def __init__(self, archivo):
        self.archivo = archivo
        self.f = open(str(self.archivo),'r')

    def datos(self):
        lista = []
        for linea in self.f:
            linea = linea.rstrip() # elimina el \n
            lista.append(linea)

        cadena = ' '.join(lista) # convierte la lista en
        # cadena para separar los números por espacio
        vector = cadena.split() # se convierte en lista de nuevo,
        #los números ya están separados por espacios
        casos = []
        tiempos = []

        for i in range(len(vector)):
            if i%2 !=0: # no se toman en cuenta las palabras

                for j in range(len(vector)):

                    if i==(4*j+1):
                        casos.append(vector[i])

                for j in range(len(vector)):

                    if i==(4*j+3):
                        tiempos.append(float(vector[i]))

        print('Casos a analizar:',casos)
        print('tObjetivo:',tiempos)
        self.f.close()

        return casos, tiempos
```

```
class Nma:
    # nivel medio del agua
    def __init__(self,long):
        self.long = long
        x = []
        y = []

        for i in range(self.long+1):
            x.append(i)
            y.append(1.0)

        Graph(x,y).plot()

class AlturaObjetivo: # altura de onda ob (jetivo
    def __init__(self,caso,sens,tob, tmax):
        self.tob = tob #tObjetivo
        self.tmax = tmax #Son todos los tiempos asociados a las
        #máximas alturas de ola
        # se debe considerar que este tiempo es
        self.caso = caso
        self.sensor = sens
        self.lista = [] # guarda el resultado
        self.tiempo()
        self.altob()

    def tiempo(self):
        # calcula el indice del tiempo
        for i in range(len(self.tmax)-1):
            # En cada uno de las alturas de ola se guarda un
            # tiempo (tMax), el cual indica cuando se generó esa
            #altura de ola en el sensor.
            #En este ciclo se toma la lista de todos los tiempos
            # en que e generaron alturas de ola

            if (self.tob >=self.tmax[i] ) and (self.tob<= self.tmax[i+1]):
                #Se compara la lista de tiempos de la altura de ola
                # (tmax) con el tiempo en que se creó la altura de ola
                # que generó la máxima presión sobre el muelle (tObjetivo)
                # y se guarda el índice objetivo, o sea, el índice de la
                # lista tmax al cual corresponde el tiempo en que se creó
                # la altura de ola que generó la  máxima presión sobre el
                # muelle.
                self.ihob = i # indice objetivo

        self.ihob -=1 # se disminuye en uno porque el contador
        # comenzará dede cero en la lista h1
    def altob(self): # altura objetivo
        ruta ='H/caso'+str(self.caso) + '/alturas'
```

```
        self.f = open(ruta + '/h'+str(self.sensor), 'r')

        for linea in self.f:
            linea = linea.rstrip() #Elimina el '\n'
            self.lista.append(linea)

        self.cadena = ' '.join(self.lista) # convierte la
        #lista en cadena para separar los números por espacio
        self.v = self.cadena.split() # Se convierte en lista de nuevo,
        #los números ya están separados por espacio
        self.res = []# resultado de las alturas de ola de cada sensor

        for x in range(len(self.v)):
            self.res.append(float(self.v[x]))

        self.f.close()
        ruta ='H/caso'+str(self.caso) + '/H1'

        if not os.path.isdir(ruta): # si ese directorio no existe,
            #entonces se crea
            os.makedirs(ruta)

        self.Hp=self.v[self.ihob] # se obtiene Hp (La altura de ola
        # que generó la máxima presión)
        f = open(ruta+'/h1','w') # H1 se guarda en el archivo h1
        f.write(str(self.Hp))
        f.close()

        return self.Hp

class Herr:
    #herramientas de análisis
    def __init__(self,a):
        self.a = a

    def mx(self):
        mx = max(self.a)
        mi = min(self.a)

        if abs(mi)>mx:
            return mi
        else:
            return mx


#AlturaObjetivo()
class Ploter:
    #  plotea los casos de estudio
    def __init__(self,caso,h1):
```

```
        self.H1 = h1 # altura de onda que genera la máxima presión
        self.caso = caso # caso de estudio
        self.hs = 0.135#[m] distancia desde la rampa hasta la
        #superficie libre
        self.g = 9.81 #[m/s^2] gravedad
        self.rho = 1000 #[kg/m^3] densidad del agua
        #self.dtif = 5.2 #[s] intervalo de tiempo en que la onda
        # se demora en llegar desde el inicio de la rampa hasta
        # el muelle
        self.vertical()

    def vertical(self):
        #Calculos en la vertical
        self.sens = 31 # cantidad de sensores en la vertical
        #self.presionesMx()
        #self.hPres() # llama al método que encuentra la onda
        # que generó la máxia presión
        yy = []
        xx = []

        for i in range(1,self.sens+1):
            r = Archivos(self.caso,i,'presiones').fin() # se abre la
            #lista de datos (el sensor i )
            pMax = Herr(r).mx() # presión máxima encontrada en la
            # lista sensor
            print(i,pMax)
            xx.append((pMax)/(self.rho*self.g*self.H1))  # presiones
            # máximas

        z = 0.0 # este valor toma en cuenta la profundidad del agua

        for  k in range(len(xx)):
            aux = z/self.hs
            yy.append(round(aux,3))
            z+=0.01 # [m] distancia entre cada sensor en la vertical

        Graph(xx,yy,'caso '+str(self.caso)).sca('$p_{max} \, / \,...
            ...\\rho g H_1$','$z/h_s$')

    def horiz(self,ini,fin):
        self.ini = ini
        self.fin = fin
        x = []
        c = 0.1
        y = []

        for i in range(ini,fin):

            r = Archivos(self.caso,i).fin() # Se obtiene la lista
```

```
            # sensor

            y.append((Herr(r).mx())/(self.rho*self.g*self.H1))#
            #se calcula el máximo de la lista sensor

        while c <= 0.6:
            x.append(c/self.hs)
            c+=0.1

        Graph(x,y).sca('$x/h_S$','$p_{max}/\\rho g H_1$')


class Procesador:
    # procesador de los casos; genera los bucles necesarios
    #para analizar todos los casos
    def __init__(self):
        self.sensor = 1 #solo se analiza el sensor al pie de la rampa
        self.casos, self.tObjetivo = Propiedades('pm').datos() #caso,
        # tObjetivo
        # tObjetivo = t (tiempo en que ocurre la máxima presión) - dtif
        #(tiempo en que se demora la onda, que generó esa máxima presión,
        #en recorrer desde el sensor de desnivelaciones hasta
        # el muelle).
        self.bucle()

    def bucle(self):

#caso = '0'+str(caso)
        tmax = [] # tiempos máximos

        for i in range(len(self.casos)):

            y = Archivos(self.casos[i],self.sensor).fin()
            ## se abre el sensor 1 del caso 'i'
            # y: lista de desnivelaciones
            tmax.append( Cruces(y,self.casos[i],self.sensor).cruce()[1])
            # entrega como resultado los tiempos
            #máximos
        #altura objetivo

        h1 =[] #H1
        for i in range(len(self.casos)):

            s = AlturaObjetivo(self.casos[i],self.sensor,...
                ...self.tObjetivo[i],tmax[i]).altob()
            # caso,
            #sensor=1 (desnivelaciones),  tObjetivo, ¿tMax ?
            #tmax
            h1.append(float(s))
```

```
        print('\n Altura de ola que generó la máxima presión...
            ... sobre el muelle (H_1):',h1)

        for i in range(len(self.casos)):

            Ploter(self.casos[i],h1[i])


#Nma(110)
Procesador()

#plt.gca().invert_yaxis()

plt.legend()

plt.grid()

plt.show()

print('\t \t \t \t Hecho')
print(45*'--')
```

E.3. Scripts para configurar OpenFOAM

E.3.1. TLO

```
# -*- coding: utf-8 -*-
'''
calcula parámetros de la TLO
'''
import math as mt
import os
#               Parámetros para la TLO
#-------------------------------------------
#-------------------------------------------
H0 = 0.1 #[m] Altura de ola a propagar
h = 0.635 #[m] Profundidad inicial
T = 2.0  #[s] periodo
nc  = 12.5 #[12.5,25,50] número de celdas por altura de ola
hp = h#4.0 #[m] profundidad hasta donde se propaga
th0 = 0 #[grados] ángulo de incidencia del oleaje

#-------------------------------------------
#               Parámetros para la condición de Courant y para indicar
# la cantidad de celdas
lCanal = 10.7 #[m] largo total del dominio en x
```

```
aCanal = 1.2 #[m]Alto del canal (dominio completo)
lo = 1.0 # cantidad de longitudes de onda a lo largo del canal a analizar
stream = False # si está activado muestra las opciones de Stream Function
#-------------------------------------------
f = open('propiedades', 'w')
f.write('H0'+'\t'+str(H0) +'\n' )
f.write('T'+'\t'+str(T) +'\n' )
f.write('d'+'\t'+str(h) +'\n' )
th0 = (mt.pi * th0)/180
g = 9.81 # gravedad

def londa(d, sw =0):
    L = (g*(T**2))/(2*mt.pi)

    for x in range(1000):
        L =((g*T**2)/(2*mt.pi))*mt.tanh((2*mt.pi*d)/(L))

    print('Longitud de onda', round(L,5),'[m]')

    if sw ==0:
        pass

    else:

        if d/L>0.5:
            print('Aguas profundas')

        elif 1.0/20.0<d/L and  d/L<0.5:

            print('Aguas intermedias')

        elif d/L<1.0/20.0:
            print('Aguas someras')

    return L

def celdas(l):
    dx = L/100
    dy = H0/10
    d = 0.00002# esta es la variación de los valores de las celdas

    while dy>= H0/10 or dy**2>=0.05:
        dy = dy-d

    while dx>=L/100 or dx**2>=0.05 or dx>2.5*dy:
        dx = dx-d

    if dx>=L/100:
        print('dx>=L/100')
```

```
    elif dy>=H0/10:
        print('dy>=H0/10')

    elif dx>=2.5*dy:
        print('dx>=2.5dy')

    elif dx**2>=0.05:
        print('dx^2>=0.05')

    elif dy**2>=0.05:
        print('dy^2>0.0.5')

    dx = round(dx,4)
    dy2 = (H0/nc)
    dx = round(dx*100,2) #[cm]
    dy = round(dy*100,2) #[cm]
    print('lCanal:',lCanal)
    print('\n'+'Arjona: '+'dx:',dx,'[cm],', 'dy', dy,'[cm]')
    print('Cantidad de celdas para Arjona (x , y):',...
        ...int(lCanal/(dx/100)),',',int(aCanal/(dy/100)))
    print('\n'+'Larsen: '+'dx = dy =',round(dy2*100,2),'[cm]')
    print('Cantidad de celdas para Larsen (x , y):',...
        ...int(lCanal/dy2),',',int(aCanal/dy2))

    return dx, dy

def Re(L):
    d50 = 0.003 # diámetro medio del poro
    v = 1.007*(10**(-6)) # viscosidad
    V = ((H0*g)/2)*(T/L) # velocidad
    re = round((V*d50)/v, 2)
    print('Re:', re)

    if re<=2300:
        print('Flujo laminar')

    elif 2300 < re and re <= 4000:
        print('Flujo transicional')

    elif re> 4000:
        print('Flujo turbulento')

#---------------------------------------------------------------------
# propagación
print('Profundidad inicial')
L = londa(h)
f.write('L0'+'\t'+str(L) +'\n' )
Re(L)
```

```
deltax,deltay = celdas(L)
d = h
c0 = L/T
print('Velocidad de onda C0:',round(c0,2),'[m/s]')

def co(u):
    #               Parámetros para Courant
    #-----------------------------------------
    d = lCanal #Largo total del dominio en x
    n = d/deltax #Cantidad de celdas
    dx = d/n #
    co = 0.45 # Courant
    dt = (dx*co)/u
    return round(dt,5)
#-----------------------------------------
cr = []
cr.append(co(c0))
print(50*'-')
k = (2*mt.pi)/L # número de onda
n = 0.5*(1 + (2*k*h)/(mt.sinh(2*k*h))) # factor n, de la celeridad
cg0 = n*c0
#_---------------------Calculando Ks.------------------
print(50*'-')
ms = lCanal/c0 # tiempo mínimo de simulación para flujo laminar
ms = ms*10 # el fluido debe pasar unas 10 veces por el dominio
print('El tiempo de simulación para un fluido en condición laminar...
    ... debe ser de al menos:',round(ms,3),'[s]')
print('El intervalo de escritura debe ser menor a deltaT:',min(cr),'[s]')

class Ur:

    def __init__(self):
        # calcula en qué tipo de no linealidad está la onda
        self.ur = (H0*(L**2))/(d**3)

    def u(self):

        if self.ur>26:
            print('Es aplicable ta teoría Cnoidal')

        if self.ur<10:
            print('Es aplicable la teoría de Stokes')

        if 10<self.ur<26:
            print('Cnoidal o Stokes funcionan bien')
Ur().u()
f.close()

class StreamFunction:
```

```
    def __init__(self):
        self.H0ad = H0/h # altura de ola adimensional
        self.Tad = T*mt.sqrt(g/h) # periodo adimensional

        if stream == True:
            self.pRes()

    def pRes(self): # print resultados
        print(50*'-.')
        print('\t\t\t\t\t\t Stream function')
        print(50*'-.')
        print('Altura de ola relativa [H/d]:',round(self.H0ad,4))
        print('Periodo relativo [T sqrt(g/d)]:', round(self.Tad,4))
        print('Criterio: 2')
        print('Al no haber transporte de masa, Current magnitude:',0)
        print('Componentes de Fourier(N) [10,32]: E_j< 10^-4')
        print('Número de pasos [1,20]')

StreamFunction()

#os.system("python passive.py")
#os.system('python waveTheory.py')
```

E.3.2. Creador de sensores VOF

```
# -*- coding: cp1252 -*-


'''
Genera el archivo para los sensores
'''

end = 0.6   #[m] altura hasta la que está el sensor
esp = 0.01  #[m]espacio entre sensores
x1  = 0.1   #[m]Sensor de inicio
lx  = 3.79 #[m]longitud del canal en x
ly  = 0.01  #[m]ancho del canal
sn = (lx-x1)/esp # Cantidad de sensores
print('Cantidad de sensores ', int(sn))

f = open('sampleDictVOF', 'w')

f.write('/*--------------------------------*- C++ -*----------------------*\\\n')
f.write('| =========                 |                                       |\n')
f.write('| \\\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox |\n')
f.write('|  \\\    /   O peration     | Version:  1.7.1                       |\n')
```

```
f.write('|   \\\  /    A nd           | Web:      www.OpenFOAM.com           |\n')
f.write('|    \\\/     M anipulation  |                                       |\n')
f.write('\*---------------------------------------------------------------------*/\n')

f.write('FoamFile\n')
f.write('{\n')
f.write('    version     2.0;\n')
f.write('    format      ascii;\n')
f.write('    class       dictionary;\n')
f.write('    location    "system";\n')
f.write('    object      sampleDict;\n')
f.write('}\n')
f.write('// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *//\n')

f.write('\n')
f.write('interpolationScheme cellPoint;\n')
f.write('\n')
f.write('setFormat       raw;\n')
f.write('\n')
f.write('surfaceFormat   raw;\n')
f.write('\n')
f.write('sets\n')
f.write('(\n')

def lapiz(a,dx):
    a = str(a)
    dx = str(round(dx,3))
    dy = str(ly/2.0)
    f.write('    GaugeVOF0'+a+'\n')
    f.write('    {\n')
    f.write('        type    midPointAndFace;\n')
    f.write('        axis    xyz;\n')
    f.write('        start   ( '+dx+' '+dy+' 0.0 );\n')
    f.write('        end     ( '+dx+' '+dy +' '+str(end)+' );\n')
    f.write('    }\n')

for x in range(int(sn)+1):
    lapiz(x+1,float(x1+x*esp))
    f.write('\n')

f.write(');\n')
f.write('\n')
f.write('surfaces        ();\n')
f.write('\n')
f.write('fields          ( alpha.water );\n')
f.write('\n')
f.write('// For OpenFOAM 4.0.0\n')
f.write('type sets;\n')
f.write('libs ("libsampling.so");\n')
```

```
f.write('\n')
f.write('// ************************************************************************//\n')

f.close()
```

E.3.3. Creador de blockMeshDict

```
# -*- coding: cp1252 -*-


'''
Genera el archivo para BlockmeshDict
'''

f = open('blockMeshDict', 'w')
#_-----------------------------
#-----------------------------
x = 10.7 # largo del canal
y = 0.01 # ancho
z = 1.2 # alto
ancho = 2.37  #dx[cm] ancho de cada celda en centímetros
alto = 0.95# dy[cm] alto de cada celda
#alto  = ancho#0.8  #dy alto de cada celda
#-----------------------------
factor = 2.0 #Factor de disminución de las dimensiones de cada celda
ancho = ancho/100.0 # pasarlos a centímetros
alto  = alto/100.0
ancho = ancho/factor
alto = alto/factor
dx = x/ancho  # cantidad de cortes
dy = 1
dz = z/alto
#.................................
print('Cantidad de celdas en x:',int(dx))
print('Cantidad de celdas en z:', int(dz))
#------------
x = float(x)
y = float(y)
z = float(z)
f.write('/*--------------------------------*- C++ -*----------------------*\\\n')
f.write('| =========                 |                                       |\n')
f.write('| \\\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox |\n')
f.write('|  \\\    /   O peration     | Version:  1.7.1                       |\n')
f.write('|   \\\  /    A nd           | Web:      www.OpenFOAM.com            |\n')
f.write('|    \\\/     M anipulation  |                                       |\n')
f.write('\*-------------------------------------------------------------------*/\n')
f.write('FoamFile\n')
f.write('{\n')
f.write('    version         2.0;\n')
```

```
f.write('    format          ascii;\n')
f.write('    class           dictionary;\n')
f.write('    object          blockMeshDict;\n')
f.write('}\n')
f.write('\n')
f.write('// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *//\n')
f.write('\n')
f.write('convertToMeters 1;\n')
f.write('\n')
f.write('vertices        \n')
f.write('(\n')
f.write('   (0.0     0.0     0.0)\n') #0
f.write('   ('+str(x)+'   0.0     0.0)\n') # 1
f.write('   ('+str(x)+'   '+str(y)+'     0.0)\n') #2
f.write('   (0.0     '+str(y)+'     0.0)\n') #3
f.write('   (0.0     0.0    '+str(z)+')\n') # 4
f.write('   ('+str(x)+'   0.0    '+str(z)+')\n') # 5
f.write('   ('+str(x)+'   '+str(y)+'    '+str(z)+')\n') #6
f.write('   (0.0     '+str(y)+'    '+str(z)+')\n') # 7
f.write(');\n')
f.write('\n')
f.write('\n')
f.write('blocks          \n')
f.write('(\n')
f.write('    hex (0 1 2 3 4 5 6 7)   ('+str(int(dx))+' '+str(dy)+'...
    ...'+str(int(dz))+') simpleGrading (1 1 1)\n')
f.write(');\n')
f.write('\n')
f.write('edges           \n')
f.write('(\n')
f.write(');\n')
f.write('\n')
f.write('boundary         \n')
f.write('(\n')
f.write('    inlet\n')
f.write('    {\n')
f.write('        type patch;\n')
f.write('        faces\n')
f.write('        (\n')
f.write('            (0 3 7 4)\n')
f.write('        );\n')
f.write('    }\n')
f.write('    outlet\n')
f.write('    {\n')
f.write('        type patch;\n')
f.write('        faces\n')
f.write('        (\n')
f.write('            (1 2 6 5)\n')
f.write('        );\n')
```

```
f.write('    }\n')
f.write('    wall1\n')
f.write('    {\n')
f.write('        type wall;\n')
f.write('        faces\n')
f.write('        (\n')
f.write('            (0 1 2 3)\n')
f.write('        );\n')
f.write('    }\n')
f.write('    atmosphere\n')
f.write('    {\n')
f.write('        type patch;\n')
f.write('        faces\n')
f.write('        (\n')
f.write('            (4 5 6 7)\n')
f.write('        );\n')
f.write('    }\n')
f.write(');\n')
f.write('\n')
f.write('mergePatchPairs\n')
f.write('(\n')
f.write(');\n')
f.write('\n')
f.write('// ************************************************************************//\n')
f.close()
```

E.3.4. Generador del archivo blockMeshDict para dos parches laterales

```
# -*- coding: cp1252 -*-

'''
Genera el archivo para BlockmeshDict
'''
f = open('blockMeshDict', 'w')
#_----------------------------
#-----------------------------
x = 4.624 # largo del canal
y = 0.01 # ancho
z = 1.0 # alto

#-----------------------------------------
diam = 7.62 #[cm] (diámetro de la manguera) Largo en 'z' del parche inferior
ancho = 2.49  #dx[cm] ancho de cada celda en centímetros
alto = 1.0# dy[cm] alto de cada celda
#alto  = ancho#0.8  #dy alto de cada celda

#-----------------------------
```

```
factor = 1.0 #Factor de disminución de las dimenciones de cada celda

#----------------------------------------------
ancho = ancho/100.0 # pasarlos a centímetros
alto  = alto/100.0
diam = diam/100.0

ancho = ancho/factor
alto = alto/factor

dx = x/ancho  # cantidad de cortes
dy = 1
dz = (z-diam)/alto
ddiam = diam/alto

print('z, diam',z,diam)
#.....................................
print('Cantidad de celdas en x:',int(dx))
print('Cantidad de celdas en z:', int(dz+diam))
#------------
x = float(x)
y = float(y)
z = float(z)
diam = float(diam)

f.write('/*---------------------------------*- C++ -*-------------------------*\\\n')
f.write('| =========                 |                                         |\n')
f.write('| \\\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox |\n')
f.write('|  \\\    /   O peration     | Version:  1.7.1                       |\n')
f.write('|   \\\  /    A nd           | Web:      www.OpenFOAM.com            |\n')
f.write('|    \\\/     M anipulation  |                                       |\n')
f.write('\*---------------------------------------------------------------------*/\n')
f.write('FoamFile\n')
f.write('{\n')
f.write('    version        2.0;\n')
f.write('    format         ascii;\n')
f.write('    class          dictionary;\n')
f.write('    object         blockMeshDict;\n')
f.write('}\n')
f.write('\n')
f.write('// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *  * //\n')
f.write('\n')
f.write('convertToMeters 1;\n')
f.write('\n')
f.write('vertices        \n')
f.write('(\n')
f.write('   (0.0     0.0     0.0)\n') #0
f.write('   ('+str(x)+'   0.0     0.0)\n') # 1
f.write('   ('+str(x)+'   '+str(y)+'     0.0)\n') #2
```

```
f.write('   (0.0     '+str(y)+'     0.0)\n') #3
f.write('   (0.0     0.0    '+str(diam)+')\n') # 4
f.write('   ('+str(x)+'   0.0    '+str(diam)+')\n') # 5
f.write('   ('+str(x)+'   '+str(y)+'    '+str(z)+')\n') #6
f.write('   (0.0     '+str(y)+'    '+str(diam)+')\n') # 7

f.write('   (0.0     0.0     '+str(z)+')\n') #8
f.write('   ('+str(x)+'     0.0     '+str(z)+')\n') #9
f.write('   ('+str(x)+'     '+str(y)+'     '+str(z)+')\n') #10
f.write('   (0.0     '+str(y)+'     '+str(z)+')\n') #11

f.write(');\n')
f.write('\n')
f.write('\n')
f.write('blocks          \n')
f.write('(\n')
f.write('    hex (0 1 2 3 4 5 6 7)   ('+str(int(dx))+...
    ...' '+str(dy)+' '+str(int(ddiam))+') simpleGrading (1 1 1)\n')

f.write('    hex (4 5 6 7 8 9 10 11)   ('+str(int(dx))+...
    ...' '+str(dy)+' '+str(int(dz))+') simpleGrading (1 1 1)\n')
f.write(');\n')
f.write('\n')
f.write('edges           \n')
f.write('(\n')
f.write(');\n')
f.write('\n')
f.write('boundary         \n')
f.write('(\n')
f.write('    inlet\n')
f.write('    {\n')
f.write('        type patch;\n')
f.write('        faces\n')
f.write('        (\n')
f.write('            (4 7 11 8)\n')
f.write('        );\n')
f.write('    }\n')
f.write('    outlet\n')
f.write('    {\n')
f.write('        type patch;\n')
f.write('        faces\n')
f.write('        (\n')
f.write('            (5 6 10 9)\n')
f.write('        );\n')
f.write('    }\n')
f.write('    mIn\n')
f.write('    {\n')
f.write('        type patch;\n')
f.write('        faces\n')
```

```
f.write('        (\n')
f.write('            (0 3 7 4)\n')
f.write('        );\n')
f.write('    }\n')
f.write('    mOut\n')
f.write('    {\n')
f.write('        type patch;\n')
f.write('        faces\n')
f.write('        (\n')
f.write('            (1 2 6 5)\n')
f.write('        );\n')
f.write('    }\n')
f.write('    wall1\n')
f.write('    {\n')
f.write('        type wall;\n')
f.write('        faces\n')
f.write('        (\n')
f.write('            (0 1 2 3)\n')
f.write('        );\n')
f.write('    }\n')
f.write('    atmosphere\n')
f.write('    {\n')
f.write('        type patch;\n')
f.write('        faces\n')
f.write('        (\n')
f.write('            (8 9 10 11)\n')
f.write('        );\n')
f.write('    }\n')
f.write(');\n')
f.write('\n')
f.write('mergePatchPairs\n')
f.write('(\n')
f.write(');\n')
f.write('\n')
f.write('// ************************************************************************* //\n')
f.close()
```