Facultad de Ciencias
Sección de Física
Universidad de La Laguna

FACULTAD DE CIENCIAS
SECCIÓN DE FÍSICA

Trabajo de Fin de Grado

Desarrollo de una simulación FDTD para la estimación de parámetros acústicos en entornos cerrados

Titulación: Grado en Física

Estudiante:
Elías Gabriel Ferrer Jorge

Tutor: Fernando Luis Rosa González

2 de Febrero de 2022

Facultad de Ciencias
Sección de Física
Universidad de La Laguna

Departamento de
Ingeniería Industrial
Universidad de La Laguna

IMPRESO DE AUTORIZACIÓN DEL TRABAJO DE FIN DE GRADO POR EL TUTOR

Curso 2021/2022

El Dr. D. **Fernando Luis Rosa González**, con D.N.I. 43611314-W, como tutor del estudiante D. **Elías Gabriel Ferrer Jorge** en el Trabajo de Fin de Grado titulado

Desarrollo de una simulación FDTD para la estimación de parámetros acústicos en entornos cerrados,

da su autorización, acreditada por la firma electrónica de este documento, para la presentación y defensa de dicho proyecto, a la vez que confirma que el estudiante ha cumplido con los objetivos generales y particulares que lleva consigo la realización del mismo.

La Laguna, a 2 de Febrero de 2022.

Facultad de Ciencias
Sección de Física
Universidad de La Laguna

FACULTAD DE CIENCIAS
SECCIÓN DE FÍSICA

Trabajo de Fin de Grado

Desarrollo de una simulación FDTD para la estimación de parámetros acústicos en entornos cerrados

TOMO I

Memoria

Titulación: Grado en Física

Estudiante:

Elías Gabriel Ferrer Jorge

Tutor: Fernando Luis Rosa González

2 de Febrero de 2022

Índice general

I Memoria **5**

Abstract **14**

1. Introducción **15**

1.1. Objetivos 15

1.2. Material y método 15

1.3. Organización de la memoria 16

2. Fundamentos **17**

2.1. Ecuación de Ondas Acústicas 17

2.2. Fuente del campo de presión 19

2.2.1. Monopolo 19

2.3. Respuesta Impulsiva 20

2.4. Parámetros Acústicos 21

2.4.1. Tiempo de reverberación 21

2.4.2. Nivel de fuerza sonora G 23

2.4.3. Claridad acústica C_{80} 23

2.4.4. Nivel de fuerza sonora G_{80} 23

2.5. Métodos de diferencias finitas 24

2.5.1. Mallados de Presión y Velocidad 24

2.5.2. Aplicabilidad del método FDTD 26

2.5.3. Aplicabilidad de otros métodos numéricos 29

3. Desarrollo de la API **31**

3.1. Clases y Funciones 32

3.1.1. *Room* 32

3.1.2. *Medium* 32

3.1.3. *Source* 33

3.1.4. *Microphone* 33

3.1.5. *Sim* 35

3.1.6. *Counter* 35

3.1.7. *Filter* 35

3.1.8. Otras herramientas 35

4. Simulación FDTD **37**

4.1. Discretización de la ecuación de ondas 37

4.1.1. Algoritmo de propagación acústica 37

4.1.2. Implementación de algoritmo FDTD 39

4.2. Condiciones de contorno 40

4.2.1. Condiciones de contorno reflectante 41

4.2.2. Condiciones de contorno absorbente 42

4.3. Respuesta Impulsiva 43

4.3.1. Implementación de la función impulso 43

5. Validación y resultados **45**

5.1. Caso con solución exacta 45

5.2. Simulación del caso elegido 46

6. Conclusiones y trabajo futuro **51**

6.1. Trabajo futuro 51

II Anexos **57**

A. Resumen de los capítulos **59**

A.1. Capítulo 2. Fundamentos 59
A.2. Capítulo 3. Desarrollo de la API 59
A.3. Capítulo 4. Simulación FDTD 59
A.4. Capítulo 5. Validación y resultados 60
A.5. Capítulo 6. Conclusiones y trabajo futuro 60

B. Summaries of chapters **61**

B.1. Chapter 2. Fundamentals 61
B.2. Chapter 3. API development 61
B.3. Chapter 4. FDTD Simulation 61
B.4. Chapter 5. API's validation and results 62
B.5. Chapter 6. Conclusions and future 62

C. Clases de la API **63**

C.1. *Room* 64
C.2. *Medium* 66
C.3. *Source* 69
C.4. *Microphone* 73
C.5. *Sim* 76
C.6. *Counter* 86
C.7. *Filter* 87

D. Funciones **89**

D.1. *show.py* 90
D.2. *datafiles.py* 93
D.3. *wav.py* 94
D.4. *acoustic_param.py* 95

D.5. *write_vert_fac.py* 104

E. Ejemplos de ejecución **105**

E.1. valar.py 106

Índice de figuras

2.1. Campo de presiones generado por monopolo con frecuencia $f = 200\ Hz$, en agua salada ($\rho = 1027\ Kg/m^3$, $c = 1500\ m/s$) y aire ($\rho = 1{,}225\ Kg/m^3$, $c = 342\ m/s$) 19

2.2. Tiempo de reverberación. Aproximaciones de Sabine y Eyring. ($K = 1, V = 1, S_j = 1$) 22

2.3. Mallado de presión y velocidad del método FDTD en 2D 25

2.4. Celda unidad del método FDTD en 3D 26

2.5. Algoritmo de resolución temporal de celda unidad en FDTD . . . 27

2.6. Longitud de onda frente a frecuencia sonora en agua y aire en el rango audible para el humano. 28

2.7. Longitud de onda en la escala del mallado de simulación 28

2.8. Limitación de puntos en el mallado 29

3.1. Mapa de Funciones y Clases 31

4.1. Diagrama de ejecución temporal del algoritmo FDTD 38

4.2. Técnica de doble buffer aplicada en el algoritmo FDTD. 39

4.3. Diagrama de cálculo de diferencias para los datos de presión . . . 40

4.4. Tiempo de simulación aumentando el volumen contenedor de nodos y la resolución 41

4.5. Incremento de tiempo de ejecución con la API desarrollada 41

4.6. Señal impulsiva y ruido estacionario causado por reflexiones perfectas 42

4.7. Función sinc y gaussiana como fuentes de presión diferenciable . 43

4.8. Respuesta en frecuencia del filtro pasobajo aplicado ($\Delta x = 0{,}1$ [m]) 44
4.9. Fuente impulsiva filtrada y sin filtrar 44

5.1. Representación 3D del *setup* para validación de la herramienta desarrollada. El entorno de simulación representado como un cubo con el contorno PML en negro, conteniendo en su interior una esfera (la fuente) y cubos rojos (los micrófonos) 48
5.2. Intensidad acústica obtenida por los micrófonos con la simulación de validación **valar.py** 49
5.3. Presión acústica media obtenida por los micrófonos con la simulación de validación **valar.py** y resultado analítico 49
5.4. Intensidad acústica obtenida por los micrófonos con la simulación de validación **valar.py** 50
5.5. Planos transversales de simulación de validación con **valar.py** para distintos instantes de tiempo. 50

6.1. Ruido estacionario y modos normales en el entorno de simulación para condiciones de contorno reflectantes 52
6.2. Absorción de la onda acústica por el algoritmo PML 52
6.3. Representación 3D de las piscinas de *OrcaOcean* en Loro Parque, Puerto de la Cruz, Tenerife 53

Índice de tablas

3.1. Parámetros de la clase ***Room*** 32
3.2. Parámetros de la clase ***Medium*** 33
3.3. Métodos de la clase ***Medium*** 33
3.4. Parámetros de la clase ***Source*** 34
3.5. Métodos de la clase ***Source*** 34
3.6. Parámetros de la clase ***Microphone*** 34
3.7. Métodos de la clase ***Microphone*** 34

5.1. Parámetros de entrada para la simulación de validación 47

Abstract

Numerical simulation dealing with acoustic phenomenology requieres different methods to approximate the value of characteristic and geometric properties of the environment where these phenomena occurs. Have been developed a collection of methods appropriate for different applications and use cases.

Acoustic phenomena in underwater acoustics within closed environments are addressed. The use of fluid dynamics in specific regimes greatly simplify the number of variables involved in these environments allowing its characterisation. Furthermore, these closed environments are of special interest in the study of marine fauna, because marine mammals communication is based on the use of sound. Thus, it is of interest for the study of species in captivity in ponds, pools or in the wild. So it could be helpful since working with animals becomes complicated and expensive when it is intended to maintain their welfare.

For this reason, this TFG presents an API that facilitates the development of acoustic simulation programs, based on accoustic propagation in time domain in closed spaces, allowing the study of their acoustic properties. Some programs have been developed using the API in the course of this work. The wave propagation algorithm is based on FDTD, with application for different boundary conditions, being useful for the accoustic property studies, by the use of the impulsive response.

Capítulo 1

Introducción

1.1. Objetivos

El presente trabajo tiene como objeto principal el desarrollo de una API para la creación de programas de simulación numérica, capaces de resolver el fenómeno de propagación acústica en entornos cerrados para la estimación de los parámetros acústicos en piscinas.

Se ha optado por la implementación del método FDTD para la solución numérica del problema planteado. Se elige este método por que permite obtener la respuesta impulsiva en el sistema analizado.

1.2. Material y método

La Universidad de La Laguna, ha facilitado el acceso a los recursos bibliográficos necesarios para este trabajo.

Para la realización del Trabajo de Fin de Grado, constituido por la presente memoria y las librerías que componen la API, se ha utilizado un equipo con las siguientes características técnicas:

- Sistema Operativo Windows 10
- Sistema Operativo Ubuntu 18.04
- Procesador AMD Ryzen 7 3700X
- RAM 16GB DDR4

La memoria se ha desarrollado con *TeXstudio* y compilada con LaTeX. Mientras que la API ha sido desarrollada en *Python3 (version 3.8.5)*, con la IDE *Pycharm*, la terminal virtual *Jupyter QTConsole*, además de usarse editores de texto como *Atom* y *Sublime Text*.

Para el desarrollo de los modelos 3D de las piscinas se ha usado *Blender*, mientras que las diferentes figuras ilustrativas a lo largo de la presente memoria se han desarrollado con herramientas como *InkScape* y *Canva*.

Durante el desarrollo de este Trabajo de Fin de Grado se ha realizado una revisión bibliográfica de la fenomenología acústica, de los métodos numéricos y de sus aplicaciones. Se ha realizado la selección de los algoritmos y sus modificaciones específicas atendiendo a la bibliográfia para el desarrollo de la simulación. Por último se ha continuado con la validación de la API desarrollada.

1.3. Organización de la memoria

Tras la introducción se incluye el capítulo 2 para tratar los fundamentos físicos de la acústica, relacionando distintas variables de la ecuación de ondas desde un contexto general hasta el desarrollo numérico de las mismas a partir del método FDTD. A continuación, se desarrolla una API para facilitar la creación de programas de simulación acústica en el capítulo 3 y, tras el mismo, se profundiza en el desarrollo del método FDTD aplicado, con las condiciones que cumplen los elementos de la simulación, además de las propias del contorno en el capítulo 4. Más adelante, en el capítulo 5, se comprueba la validez de las simulaciones que se pueden crear con la API desarrollada y el impacto de las modificaciones propias que se han realizado a la hora de aplicar el método PML en el contorno, para concluir con el capítulo 6, en el que se exponen las conclusiones y el trabajo futuro a partir del presente trabajo desarrollado.

Se han incluido los resúmenes en español e inglés de cada capítulo en los anexos A y B.

Capítulo 2

Fundamentos

Para el estudio de los fenómenos acústicos se parte de un dominio definido mediante la Dinámica de Fluidos y la Termodinámica con ciertas consideraciones físicas y matemáticas. Además, se tiene en consideración la aplicación en medios incompresibles para la correcta simulación numérica.

2.1. Ecuación de Ondas Acústicas

A partir de los principios básicos de conservación de la masa, balance de momento y ecuaciones de estado para fluidos en régimen adiabático [8], se desarrolla el comportamiento en la propagación de ondas de presión en medios fluidos.

Con la ecuación de Euler para fluidos no viscosos en coordenadas esféricas, ecuación 2.1, se relacionan los campos de presión y velocidad,

$$-\frac{\partial P}{\partial r} = \rho\left(\frac{Dv_r}{Dt} - \frac{v_\theta + v_\phi}{r}\right) \tag{2.1}$$

donde v_r, v_θ, v_ϕ son las componentes vectoriales del campo de velocidad, P la componente escalar del campo de presión y ρ la densidad. A partir de esta ecuación, se tienen en consideración las condiciones del entorno físico y de aplicación para simplificar la relación como se ve en la ecuación 2.2.

$$-\frac{\partial P}{\partial r} = \rho\frac{\partial v_r}{\partial t} \tag{2.2}$$

De esta forma, la solución analítica del fenómeno se basa en la resolución de las ecuaciones diferenciales acopladas de primer orden [15], para el campo escalar de presión 2.3 y el vectorial de velocidad 2.4,

$$\frac{\partial P}{\partial t} = -\rho(c_0)^2 \nabla \cdot \mathbf{v} \tag{2.3}$$

$$\frac{\partial \mathbf{v}}{\partial t} = \frac{-1}{\rho} \nabla \cdot P \tag{2.4}$$

donde $\mathbf{v}$ es el campo vectorial de velocidad y c_0 la velocidad de propagación de las ondas en el medio.

La propagación de ondas acústicas se considera un proceso adiabático en un régimen de frecuencias específico. Por ello, se aproxima el proceso de compresión del fluido como un proceso reversible isoentrópico para frecuencias menores de 10^9 Hz [13]. Esta aproximación implica que la variación de presión solo depende de su relación con la densidad del medio ρ, ecuación 2.5,

$$dp = \frac{\partial p}{\partial \rho} \mid_s d\rho \tag{2.5}$$

donde p es la presión, ρ la densidad y s la entropía. Se relacionan p y ρ con la velocidad del sonido en el medio c_0^2, ecuación 2.6, mediante la ecuación de estado 2.7 [13] {1.1 *Conservation laws and constitutive equations (1.12)*}. Lo que, en conjunto con el resto de consideraciones tomadas, define características del medio de propagación.

$$\frac{\partial p}{\partial \rho} \mid_s = c_0^2 \tag{2.6}$$

$$dp = c_0^2 d\rho + (\frac{\partial p}{\partial s})_\rho ds \tag{2.7}$$

A partir de las ecuaciones 2.3, 2.4 y 2.6 se obtiene la ecuación de ondas acústica 2.8

$$\nabla^2 P = \frac{1}{c_0^2} \frac{\partial^2 P}{\partial t^2} \tag{2.8}$$

2.2. Fuente del campo de presión

En el estudio de los fenómenos de propagación acústica se tienen en cuenta las variaciones de presión producidas por las fuentes del campo y sus características en el medio. Por este motivo, se definen las características de la fuente en el contexto de estudio.

2.2.1. Monopolo

En la modelización ideal de fuentes de presión es común el uso de fuentes isotrópicas puntuales y estáticas [10]. Por este motivo se ha usado una fuente de tipo monopolo como generadora del campo, descrita por la ecuación 2.9,

$$P(r) = A\frac{e^{ikr}}{r};\ \ r \neq 0 \tag{2.9}$$

con $P(0) = A$, donde A es la amplitud del monopolo y k el número de onda de la perturbación emitida por la fuente, siendo de esta forma, dependiente del medio por el que se propaga la onda, tal y como se puede ver en la figura 2.1.

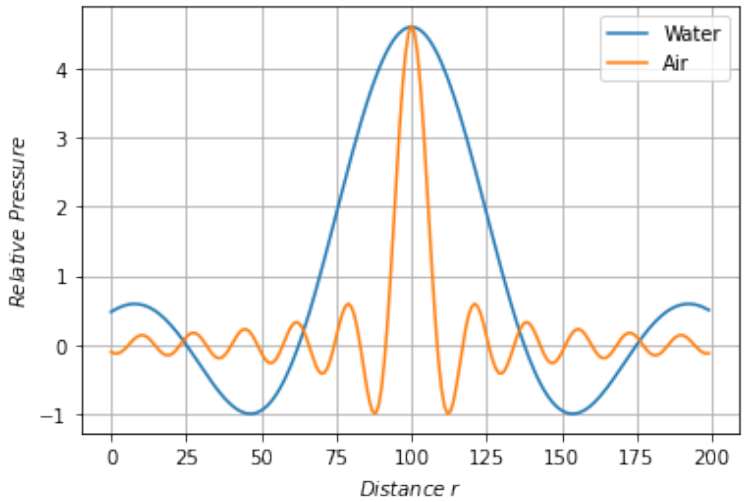

Figura 2.1: Campo de presiones generado por monopolo con frecuencia $f = 200\ Hz$, en agua salada ($\rho = 1027\ Kg/m^3$, $c = 1500\ m/s$) y aire ($\rho = 1{,}225\ Kg/m^3$, $c = 342\ m/s$)

2.3. Respuesta Impulsiva

La fenomenología asociada a las características isoentrópicas de la propagación acústica permiten tratar los medios como sistemas lineales invariantes en el tiempo o LTI [2]. Por ello, se toman las siguientes premisas:

- La fuente sonora se considera puntual.
- La fuente y el micrófono están posicionados en un lugar fijo.
- El perfil de emisión de la fuente es isotrópico.

A partir de este tratamiento del entorno de trabajo se evalúa analíticamente la propagación de la onda, tal y como se ve en la expresión 2.10,

$$s_i(t) = \int_0^\infty h_{ij}(\tau) \cdot f_j(t-\tau) d\tau \tag{2.10}$$

donde $s_i(t)$ es la información recibida por el micrófono en $\mathbf{x}_i$, $f_j(t-\tau)$ la señal emitida por la fuente sonora desde $\mathbf{x}_j$ y $h_{ij}(\tau)$ la respuesta impulsiva del medio entre cada fuente y cada micrófono..

La respuesta impulsiva del medio se puede obtener mediante un impulso, o delta de Dirac de presión [17]. Por lo que, si $f_j(t) = \delta_j(t)$, se desarrolla la solución en la ecuación 2.11,

$$s_i(t) = \int_0^\infty h_{ij}(\tau) \cdot \delta_i(t-\tau) d\tau = h_{ij}(t) \tag{2.11}$$

siendo $h_{ij}(t)$ la respuesta impulsiva del medio en unas coordenas fijas.

Por otro lado, la expresión 2.11 en el espacio transformado de frecuencias, se vuelve más fácil de tratar, como se puede ve en la expresión 2.12

$$S_i(w) = F_j(w) \cdot H_{ij}(w) \tag{2.12}$$

donde $S_i(w) = \mathfrak{F}\{s_i(t)\}$ es la densidad espectral del micrófono, $F_j(w) = \mathfrak{F}\{f_j(t)\}$ la densidad espectral de la fuente y $H_{ij}(w) = \mathfrak{F}\{h_{ij}(t)\}$ la función de respuesta en frecuencia. Por lo que se podría calcular la respuesta impulsiva entre esa fuente y ese micrófono mediante la expresión 2.13.

$$h_{ij}(t) = \mathfrak{F}^{-1}\{\frac{S_i(\omega)}{F_j(\omega)}\} \tag{2.13}$$

2.4. Parámetros Acústicos

La instrucción para la medición de parámetros acústicos en entornos cerrados está recogida en el estándar **ISO 3382**. Por lo que se tiene en especial consideración el apartado ISO 3382-1:2008, centrado en la medición del tiempo de reverberación acústica.

El tiempo de reverberación es de gran utilidad a la hora de obtener multitud de parámetros acústicos [18] y aproximar el comportamiento simulado al real [1]. Por otro lado, son de especial interés el nivel de fuerza sonora G, relacionado a las cualidades reverberantes de los recintos y el balance de energía sonora o Claridad acústica C, relacionado con el transporte de energía de la onda.

Por otro lado, cabe remarcar que la medición de la presión sonora en dB depende del medio, utilizándose las siguientes presiones de referencia para su cálculo:

- **En aire**; $P_{ref} = 20 \cdot 10^{-6}$ [Pa], según la norma **UNE-ISO 1996-2:2020**.
- **En agua**; $P_{ref} = 1 \cdot 10^{-6}$ [Pa], según el {**Apéndice A**} de la referencia [4]

2.4.1. Tiempo de reverberación

La reverberación en un recinto es un fenómeno de disipación continua de la potencia sonora de las ondas acústicas. Se usa el parámetro T_n, que indica el tiempo que tarda el sonido emitido en disminuir en n dB. Según el contexto de uso los más usados son T_{60}, T_{30} y T_{20}.

Existen diferentes aproximaciones analíticas para el cálculo del tiempo de reverberación [3]. En acústica clásica se encuentran las aproximaciones de *W.C Sabine*, ecuación 2.14 y de *Eyring*, basado en el estudio de la intensidad sonora a partir de las reflexiones que tiene en un recinto, ecuación 2.15,

$$TR_{Sabine} = \frac{KV}{\sum \alpha_j S_j} \tag{2.14}$$

donde K es una constante experimental, V el volumen del recinto, mientras que α_j y S_j corresponden a los coeficientes de absorción y las superficies de las paredes j-ésimas que encierran el volumen.

La aproximaciones experimentales que simplifican el problema a un espacio de absorción continua donde se propaga la onda, como desarrolló *W.C Sabine*, asume una respuesta directa en el flujo de energía transmitido por parte de la superficie contenedora. Esto limita la aplicación del rango de frecuencias para los

cuales se puede aplicar dicha aproximación. Por este motivo, una de las primeras modificaciones que se realizó fue la de *Eyring*, que asumía una respuesta logarítmica del tiempo de reverberación.

La aproximación de *Eyring* se basa en un desarrollo disipativo de la intensidad sonora, ecuación 2.15, en función de un conjunto de fenómenos no lineales que afectan a la reflexión de la onda. Estos, dependen de la capacidad vibracional del material que conforma la superficie y la acumulación no destructiva de vibraciones residuales en la misma.

$$I(t) = I_0 \cdot exp[(S \cdot c_0 \ln(1-\alpha)/4V) \cdot t] \tag{2.15}$$

Donde I_0 es la intensidad sonora de la fuente, α el coeficiente de absorción medio del conjunto de paredes del recinto, S la superficie total de las paredes contenedoras y V el volumen total, mientras que el tiempo de reverberación corresponde a la ecuación 2.16.

$$TR_{Eyring} = \frac{-4V}{S \cdot c_0 \cdot ln(1-\alpha)} \tag{2.16}$$

Se aprecia el comportamiento de los tiempos de reverberación para distintos coeficientes de absorción en un cubo unitario, con un medio de $c_0 = 1$, en la figura 2.2.

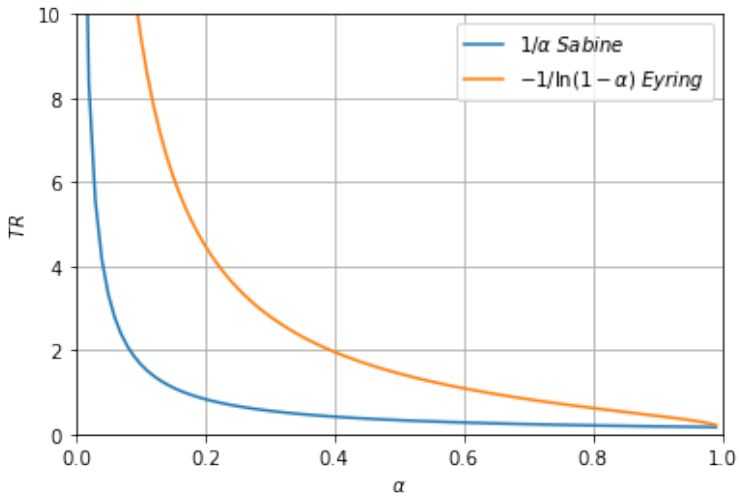

Figura 2.2: Tiempo de reverberación. Aproximaciones de Sabine y Eyring. ($K = 1, V = 1, S_j = 1$)

Actualmente, se desarrollan métodos para la estimación del tiempo de reverberación basados en modelos numéricos [6] y modelos analíticos a partir de los modos normales de las geometrías del recinto reverberante [12].

2.4.2. Nivel de fuerza sonora G

El nivel de fuerza sonora G es una magnitud indicadora del grado de amplificación natural de un recinto. Se mide mediante una fuente sonora onmidireccional y se calcula mediante la expresión 2.17,

$$G = 10\log_{10}\left(\frac{\int_0^\infty p^2(t)dt}{\int_0^\infty p_{10}^2(t)dt}\right)[dB] \tag{2.17}$$

donde $p(t)$ es la presión instantánea medida con la respuesta impulsiva del medio y $p_{10}(t)$ es la medición de la presión de la misma fuente a una distancia de 10 m por espacio libre.

2.4.3. Claridad acústica C_{80}

La claridad acústica C_{80} permite conocer la calidad acústica de un recinto o inteligibilidad de las ondas que se propagan en el mismo. Para calcular esta magnitud se toma la ecuación 2.18 y, se mide el balance entre el nivel de energía sonora o claridad acústica entre la recepción instantánea y la que se recibe 80 ms después de la misma.

$$C_{80} = 10\log_{10}\left(\frac{\int_0^{0,08} p^2(t)dt}{\int_{0,08}^\infty p^2(t)dt}\right)[dB] \tag{2.18}$$

2.4.4. Nivel de fuerza sonora G_{80}

También se puede medir el nivel de fuerza sonora, para un frente de onda en un rango de 80 ms G_{80} a partir de la emisión de un sonido, aplicando la ecuación 2.19.

$$G_{80} = 10\log_{10}\left(\frac{\int_0^{0,08} p^2(t)dt}{\int_{0,08}^\infty p_{10}^2(t)dt}\right)[dB] \tag{2.19}$$

2.5. Métodos de diferencias finitas

En general, la implementación de los algoritmos de resolución de ecuaciones diferenciales, requiere de una discretización del espacio, y en ocasiones del tiempo, en base a las características del método numérico.

La metodología FDTD permite obtener una solución numérica a ecuaciones diferenciales en el dominio temporal. En este caso se aplica a los problemas acústicos definidos con la ecuación de Helmholtz, ecuación 2.20,

$$\Delta P + \frac{w^2}{c_0^2} P = 0 \tag{2.20}$$

donde P [Pa] es la presión acústica, $w = 2\pi \cdot f$ [rad/s] es la frecuencia angular, con f [Hz], y c_0 [m/s] la velocidad de propagación de la onda por el medio.

2.5.1. Mallados de Presión y Velocidad

A la hora de aplicar la metodología FDTD, la discretización del espacio y el tiempo, se realiza con dos mallados de cálculo, los cuales contienen la información en cada instante temporal.

En primer lugar, la discretización espacial del campo escalar de presiones P, ecuación 2.21 y del campo vectorial de velocidades $\mathbf{V} = (v_x, v_y, v_z)$, ecuaciones 2.22, 2.23 y 2.24,

$$P(x,y,z,t) = P(m\Delta x, n\Delta y, p\Delta z, q\Delta t) \tag{2.21}$$

$$v_x(x,y,z,t) = v_x([m+\frac{1}{2}]\Delta x, n\Delta y, p\Delta z, [q+\frac{1}{2}]\Delta t) \tag{2.22}$$

$$v_y(x,y,z,t) = v_x(m\Delta x, [n+\frac{1}{2}]\Delta y, p\Delta z, [q+\frac{1}{2}]\Delta t) \tag{2.23}$$

$$v_z(x,y,z,t) = v_x(m\Delta x, n\Delta y, [p+\frac{1}{2}]\Delta z, [q+\frac{1}{2}]\Delta t) \tag{2.24}$$

con x,y,z las coordenadas espaciales y $\Delta x, \Delta y, \Delta z$ su resolución espacial, con lo índices m,n,p que denotan la posición en el mallado. Mientras, en la coordenada temporal t, con Δt como su resolución y el índice q para concretar instante temporal.

Ambos mallados tienen una separación espacial y temporal asociada a la dependencia mutua que relaciona ambos campos. Mientras los nodos de cálculo de presión y velocidad están a contratiempo entre sí, su posición espacial es dependiente de cada tipo de mallado, tomando desfases espaciales en cada uno de los ejes de los mismos. Se puede ver una representación de ambos mallados en la figura 2.3.

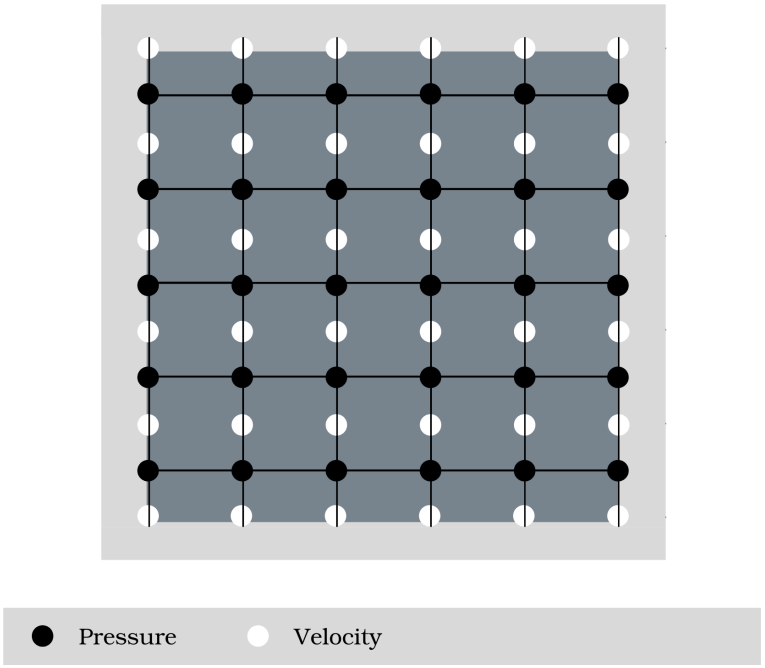

Figura 2.3: Mallado de presión y velocidad del método FDTD en 2D

La celda unidad toma en conjunto los nodos de presión y velocidad próximos e intermedios, siendo esta la mínima expresión de resolución del algoritmo FDTD a cada paso temporal, figura 2.4.

A modo de evitar la divergencia de la solución de las ecuaciones diferenciales, se mantiene el número de Courant, ecuación 2.25, como $S_c = 1/2$. Esto crea una dependencia entre la resolución temporal del método Δt, con el resto de parámetros del entorno fijados previamente.

$$Sc = \frac{c_0 \Delta t}{\Delta x} \tag{2.25}$$

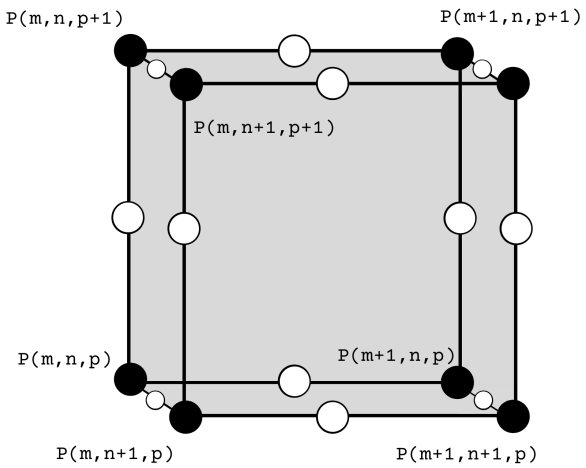

Figura 2.4: Celda unidad del método FDTD en 3D

A la hora de discretizar el tiempo se tiene en cuenta la condición CFL [5], en la que se aplica el límite de resolución temporal a partir del número de Courant, ecuación 2.25, asegurando que se mantenga por debajo de 1. Para conseguir esto, el algoritmo de ejecución toma datos del mallado de velocidad para el cálculo de la propagación de las ondas de presión a lo largo del tiempo, figura 2.5, siendo iterativo y manteniendo la convergencia a cada paso temporal.

2.5.2. Aplicabilidad del método FDTD

Existen limitaciones derivadas de la discretización de ecuaciones diferenciales y del espacio de simulación, por lo que se tratan con especial cuidado para que no ocurran fenómenos de aliasing, dispersión o divergencias en la propagación de la solución.

La propia naturaleza del método FDTD permite aproximar soluciones en entornos cerrados de forma mucho más simple que en el caso de problemas abiertos, en los que se tienen que tener en cuenta la radiación acústica al infinito, lo cual, no se deriva directamente del propio método numérico.

Por otro lado, la facilidad de aplicación del método FDTD, unido a su eficiencia y el bajo ratio de propagación de errores computacionales, han facilitado la expansión del mismo en el entorno industrial y académico. La naturaleza del

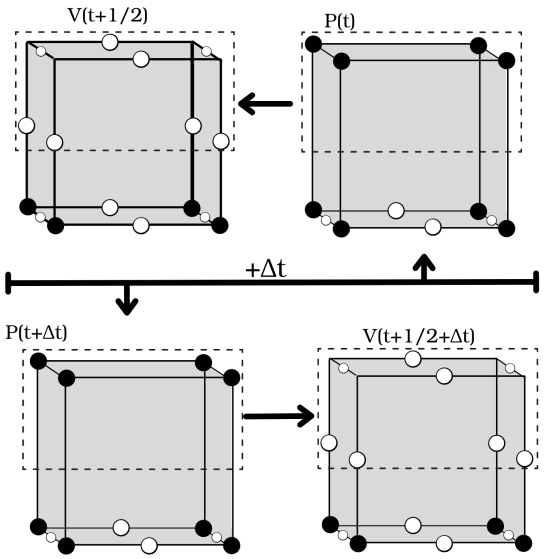

Figura 2.5: Algoritmo de resolución temporal de celda unidad en FDTD

método permite desarrollar soluciones a problemas con una mayor complejidad, como se ha hecho con la ecuación de Navier-stokes teniendo en cuenta la viscosidad del medio de propagación [9]. Cabe destacar la implementación de FDTD en GPU [14], lo que aumenta su rango de aplicaciones. Aún así, los algoritmos que resuelven la propagación acústica son altamente dependientes de los parámetros del medio, por lo que su uso es más preciso en medios de densidad homogénea [20].

En el caso de aplicación que se trata en este trabajo, se tiene en cuenta la propagación de ondas en medios absorbentes, además del tratamiento de las condiciones de contorno con algoritmos matemáticos basados en la amortiguación de las ondas acústicas. Lo cual, evita la propagación de fenómenos de dispersión con reflexión en cada punto del mallado, mientras que para la aplicación de condiciones de contorno absorbentes, se hace uso del algoritmo PML para medios absorbentes [11] con algunas modificaciones.

Limitaciones de FDTD

La dicretización del espacio de simulación hace que se tenga una determinada resolución en el mallado, lo que limita la frecuencia de la onda que se puede resolver, ecuación 2.26. Por ello, se aplicará el rango audible de frecuencias del ser humano, el cual está entre 20 Hz y 20 KHz para distintos medios, obteniendo

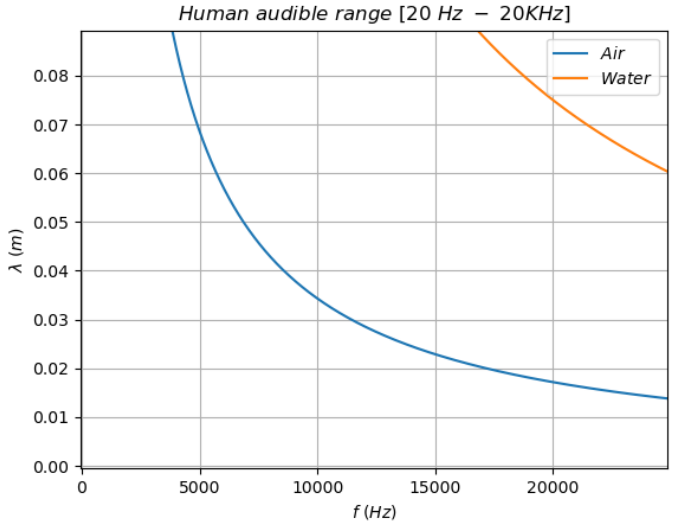

Figura 2.6: Longitud de onda frente a frecuencia sonora en agua y aire en el rango audible para el humano.

el rango de longitudes de onda a tener en cuenta para la resolución del mallado, figura 2.6.

$$\lambda = \frac{c}{f} \tag{2.26}$$

El teorema del muestreo de Nyquist-Shannon aplicado en las coordenadas espaciales [16], limita la longitud de onda mínima o frecuencia máxima capaz de resolverse en los mallados de presión y velocidad definidos.

En una propagación lineal en paralelo a un eje coordenado, 2.7, se requiere un número menor de nodos que en el caso de una propagación en general 2.8, ya que la onda no se resuelve correctamente, al no contenerse λ por un mínimo de 4 nodos.

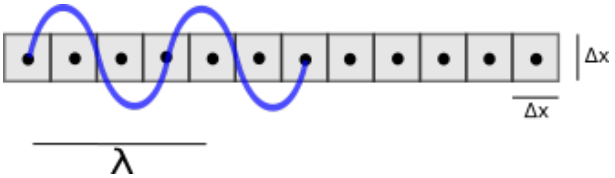

Figura 2.7: Longitud de onda en la escala del mallado de simulación

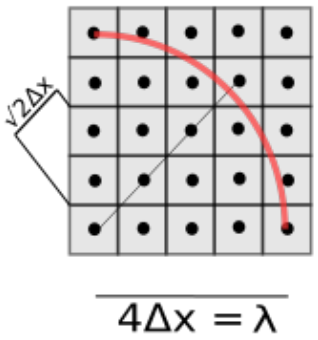

Figura 2.8: Limitación de puntos en el mallado

Por otro lado, en la aplicación general del método, se toma en consideración la necesidad de aumentar la densidad de puntos en el mallado de simulación, ya que ocurren problemas en la propagación de la solución en resoluciones menores a $\lambda/10$ cuando se introducen interfaces curvas [7] en el entorno de simulación.

2.5.3. Aplicabilidad de otros métodos numéricos

Una de las principales desventajas que presentan los distintos métodos numéricos es el tiempo de computación asociado a su aplicación. Sin embargo, existe la posibilidad de que en un futuro las mediciones reales sean sustituidas por simulaciones al completo, ya que los principales avances en este campo han ido de la mano de algoritmos cada vez más rápidos adaptados a procesamiento en GPU.

Sin embargo, cabe destacar su popularidad en la predicción de modos de vibración acústicos de estructuras y espacios cerrados, normalmente abordado con métodos BEM y FEM en el dominio de las frecuencias.

El uso de los métodos FEM se popularizó debido a su optimizada y sencilla forma de aplicación, lo que trajo consigo el desarrollo de algoritmos derivados de estos que permiten desarrollar soluciones a ecuaciones diferenciales en el dominio temporal, denominados Métodos Híbridos. Por otro lado surgen métodos basados en el trazado de rayos, para estudiar la propagación de las ondas vibracionales en sistemas complejos, como el método SEA [19].

Capítulo 3

Desarrollo de la API

La estructura de la API desarrollada está basada en la filosofía de Programación Orientada a Objetos o POO. Debido a ello, está compuesta por diferentes Clases, las cuales cumplen una función en materia de la simulación matemática, además de estructural, facilitando la implementación o mejora de funcionalidades.

La API desarrollada implementa las diferentes Clases, funciones y algoritmos de propagación acústica tal y como se representa en la figura 3.1.

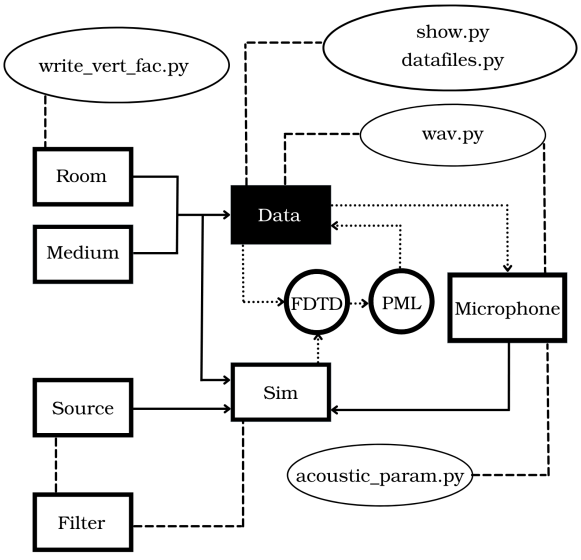

Figura 3.1: Mapa de Funciones y Clases

3.1. Clases y Funciones

Las distintas Clases y funciones desarrolladas para la ejecución de este trabajo contienen parámetros y métodos específicos. Si bien algunas Clases heredan de otras algunos parámetros, no hay una jerarquía de herencia explícita. Por lo que, para heredar parámetros de otras clases se ha optado por implementar la creación de instancias a partir de otras.

3.1.1. *Room*

Esta Clase, anexo C.1, pemite crear una instancia de Room a partir de la dimensión y resolución física deseadas para el mallado de simulación. Dicho mallado está adaptado para la ejecución del algoritmo de propagación de ondas acústicas.

La creación de una instancia Room guarda en memoria la información geométrica de los mallados de presión y velocidad necesarios para la ejecución del algoritmo FDTD, de forma predeterminada.

Se pueden ver los principales parámetros de la Clase **Room**, en la tabla 3.1.

Room

Parámetro	**Tipo**	**Definición**
dims	Lista de floats	Dimensión del entorno de simulación
dres	Float	Resolución del mallado
pini	Lista de floats	Coordenadas iniciales de creación del mallado

Tabla 3.1: Parámetros de la clase ***Room***

3.1.2. *Medium*

Esta Clase, anexo C.2, permite crear una instancia de Medium para definir los parámetros físicos del medio de simulación, ocupado en los puntos del mallado de presión y velocidad. Por lo que guarda en memoria los parámetros que definen el medio.

Los distintos parámetros que inicializan ***Medium*** se encuentran en la tabla 3.2 y sus métodos en la tabla 3.3.

Medium

Parámetro	Tipo	Definición
room	Instancia	Instancia de la Clase Room
medium	String	Nombre del medio material
pini	Lista de floats	Coordenadas iniciales de creación del mallado
medium	String	Nombre del medio material
c0	Float	Velocidad de propagación del sonido en el medio
cr0	Float	Velocidad relativa de propagación del sonido en el medio
r0h	Float	Densidad homogénea del medio

Tabla 3.2: Parámetros de la clase ***Medium***

Método	Uso
change_medium()	Permite editar los parámetros de inicialización
add_zone_density()	Permite añadir zonas con diferente densidad en el medio

Tabla 3.3: Métodos de la clase ***Medium***

3.1.3. *Source*

Esta Clase, anexo C.3, permite crear instancias de Source para implementar fuentes de presión en una región del mallado de simulación. Dichas fuentes son sinusoidales e isotrópicas de manera predeterminada. Además, la amplitud de referencia para el cálculo de la presión acústica en decibelios es de $20 \cdot 10^{-6}$ Pa, la cual está definida como amplitud de la fuente.

Los distintos parámetros que inicializan ***Source*** se encuentran en la tabla 3.4 y los métodos en la tabla 3.5.

3.1.4. *Microphone*

Esta Clase, anexo C.4, permite crear instancias de Microphone para implementar un micrófono situado en una región del mallado de simulación. Las instancias de Microphone obtienen lecturas en tiempo real del mallado de presión durante la ejecución de la simulación.

Los distintos parámetros que inicializan ***Microphone*** se encuentran en la tabla 3.6 y sus métodos en la tabla 3.7.

Source

Parámetro	Tipo	Definición
amplitude	Float	Presión sonora de emisión
sample_rate	Integer	Número de muestras por segundo
t0	Float	Tiempo en el que se inicia la emisión sonora
duration	Float	Duración de la emisión sonora
frequency	Float	Frecuencia de la onda emitida
phase	Float	Fase de la fuente
coords_0	Lista de floats	Coordenadas iniciales de la fuente
v	Lista de floats	Velocidad de desplazamiento de la fuente

Tabla 3.4: Parámetros de la clase ***Source***

Método	Uso
retone()	Remuestrea los datos para implementarlos en la simulación
create_v_in_t()	Genera un array de velocidades para cada instante de tiempo
d_dirac()	Genera una emisión en forma de delta de Dirac

Tabla 3.5: Métodos de la clase ***Source***

Microphone

Parámetro	Tipo	Definición
samples	Integer	Número de muestras
micro_label	String	Nombre de la instancia
t0	Float	Tiempo en el que se inicia la recepción sonora
adquisition_time	Float	Duración de la adquisición de datos
sensitivity_dB	Float	Sensibilidad en dB
snr	Float	Proporción de señal ruido
acoustic_overload	Float	Señal sonora límite para su lectura
coords	Lista de floats	Coordenadas para adquisición de datos

Tabla 3.6: Parámetros de la clase ***Microphone***

Método	Uso
read_data_from_file()	Carga los datos de Microphone desde un fichero binario
resample()	Remuestrea los datos para implementarse en la simulación
save_wav()	Guarda los datos del micrófono en un fichero con extensión wav

Tabla 3.7: Métodos de la clase ***Microphone***

3.1.5. *Sim*

La clase ***Sim***, anexo C.5, permite implementar las instancias de ***Room, Medium, Source, Microphone*** para ejecutar el algoritmo FDTD bajo los parámetros definidos en las mismas. El funcionamiento de esta clase se basa en la implementación y ejecución paso a paso de la propagación de ondas acústicas en entornos cerrados bajo las condiciones de contorno definidas.

Por otro lado, la instancia ***Sim***, permite guardar la información para cada paso temporal en ficheros binarios que contienen los arrays con la información, para cada punto de la simulación de presión, velocidad, densidad del medio en cada punto y cocientes para el cálculo del algoritmo FDTD y PML.

3.1.6. *Counter*

La clase ***Counter***, anexo C.6, permite llevar el contador de procesos y alternar el estado de una variable dependiendo de la paridad de las acciones realizadas.

3.1.7. *Filter*

La clase ***Filter***, anexo C.7, permite generar un filtro pasobajo, para aplicarse en los datos de la fuente impulsiva, a partir de los datos de la simulación.

3.1.8. Otras herramientas

A continuación se incluye un listado del ficheros con el resto de herramientas desarrolladas y algunas utilidades.

Show.py Anexo D.1, contiene un conjunto de funciones que permiten representar de una forma más simple los datos obtenidos en las simulaciones realizadas, asdemás de generar animaciones.

datafiles.py Anexo D.2, contiene funciones que pueden ser usadas para guardar datos en ficheros binarios. En el caso de la clase ***Sim***, se usa para liberar espacio en la memoria *RAM* durante la ejecución sin perder toda la información de la misma.

wav.py Anexo D.3, contiene funciones que permiten la conversión de la información obtenida por los micrófonos en archivos de audio en formato wav.

acoustic_param.py Anexo D.4, contiene multitud de funciones útiles para calcular parámetros acústicos como: intensidad sonora, intensidad instantánea, nivel de presión sonora, impedancia acústica, impedancia acústica específica y claridad sonora, entre otros.

write_vert_fac.py Anexo D.5, es un script que se ejecuta en la consola de Blender. Su principal función es la obtención de nodos geométricos o vértices de mallados 3D en Blender.

Capítulo 4

Simulación FDTD

La API desarrollada implementa un método basado en FDTD para entornos cerrados, bajo las consideraciones descritas en el capítulo 2.

4.1. Discretización de la ecuación de ondas

A la hora de resolver las ecuaciones de ondas acústicas computacionalmente, se desarrolla una aproximación a partir de la discretización del espacio y el tiempo.

4.1.1. Algoritmo de propagación acústica

El mallado de simulación es multidimensional y resuelve a pasos dispares y no en paralelo, lo que hace conveniente el uso de índices que faciliten el seguimiento del algoritmo. En la discretización de las ecuaciones diferenciales, se usa t como índice discreto temporal y m,n,p como índices discretos espaciales de las coordenadas (x,y,z).

Para el desarrollo del algoritmo de resolución de dependencia espacio temporal entre los mallados, se denota con pasos discretos $\Delta_{m,n,p} = 1$ y $t_{i+1} = t_i + \Delta t$. Además, se requiere de información previa del mallado contrario para cada nodo y sus alrededores. Este requerimiento hace que se use el doble de memoria computacional para los arrays de presión y velocidad.

Partiendo de las ecuaciones 2.3 y 2.4, se desarrolla un algoritmo de ejecución entrelazada para cada instante de tiempo, figura 4.1. En este algoritmo rigen la discretización del mallado de presión, siguiendo la ecuación 4.1 y el de velocidad

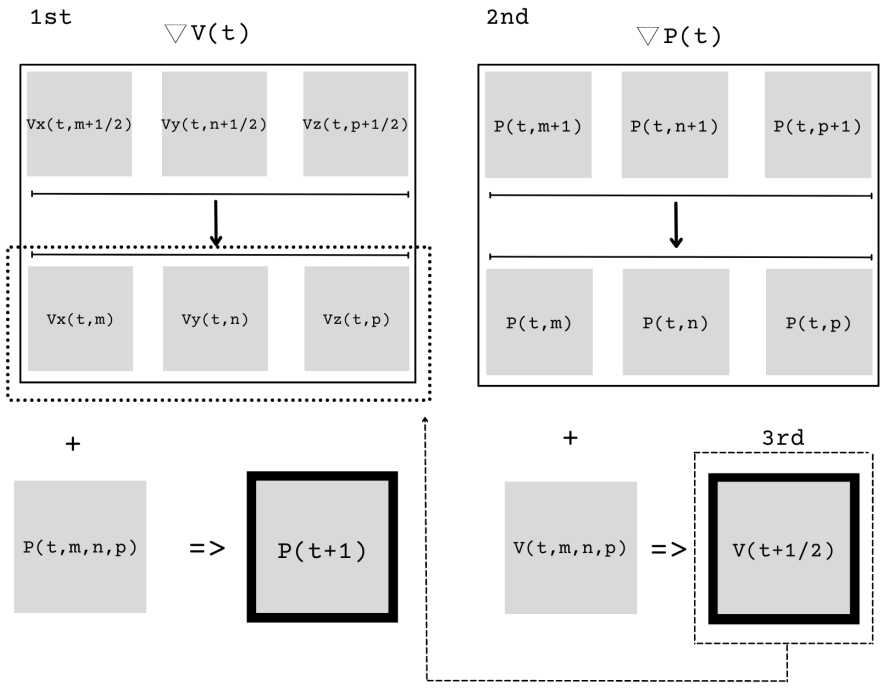

Figura 4.1: Diagrama de ejecución temporal del algoritmo FDTD

siguiendo las ecuaciones 4.2, 4.3 y 4.4. Además, se implementa el número de Courant S_c para el cálculo convergente de las diferencias finitas.

$$\begin{aligned} P_{t+1}[m,n,p] = P_t[m,n,p] - \rho_t[m,n,p](c_r \cdot c_0)^2 S_c \\ \cdot \{(V_{x,\, t+\frac{1}{2}}[m+1,n,p] - V_{x,\, t+\frac{1}{2}}[m,n,p]) \\ + (V_{y,\, t+\frac{1}{2}}[m,n+1,p] - V_{y,\, t+\frac{1}{2}}[m,n,p]) \\ + (V_{z,\, t+\frac{1}{2}}[m,n,p+1] - V_{z,\, t+\frac{1}{2}}[m,n,p])\} \end{aligned} \tag{4.1}$$

donde $c_r \leqslant 1$ es la velocidad relativa de la onda, la cual consideramos máxima de forma predeterminada.

$$V_{x,\, t+\frac{1}{2}}[m,n,p] = V_{x,\, t-\frac{1}{2}} - \frac{2S_c}{c_o} \frac{P_{t-1}[m+1,n,p] - P_t[m,n,p]}{\rho_{t-1}[m+1,n,p] + \rho_t[m,n,p]} \tag{4.2}$$

$$V_{y,\,t+\frac{1}{2}}[m,n,p] = V_{y,\,t-\frac{1}{2}} - \frac{2S_c}{c_o}\frac{P_{t-1}[m,n+1,p] - P_t[m,n,p]}{\rho_{t-1}[m,n+1,p] + \rho_t[m,n,p]} \tag{4.3}$$

$$V_{z,\,t+\frac{1}{2}}[m,n,p] = V_{z,\,t-\frac{1}{2}} - \frac{2S_c}{c_o}\frac{P_{t-1}[m,n,p+1] - P_t[m,n,p]}{\rho_{t-1}[m,n,p+1] + \rho_t[m,n,p]} \tag{4.4}$$

4.1.2. Implementación de algoritmo FDTD

La implementación eficiente del algoritmo se hace especialmente necesaria en cálculo con CPU, ya que de otra forma el tiempo de ejecución tiende a ser alto, haciendo poco práctico el método.

La ejecución de los cálculos parte de la independencia entre nodos en el sistema. Tomando en cuenta que se calculan las diferencias de los nodos de distintos mallados, se utiliza la técnica de doble buffer para la obtención de los valores propagados a los nodos vecinos para cada paso temporal, los cuales están contenidos en Data(i) y Data(i+1), siendo i un índice temporal, tal y como se representa en la figura 4.2.

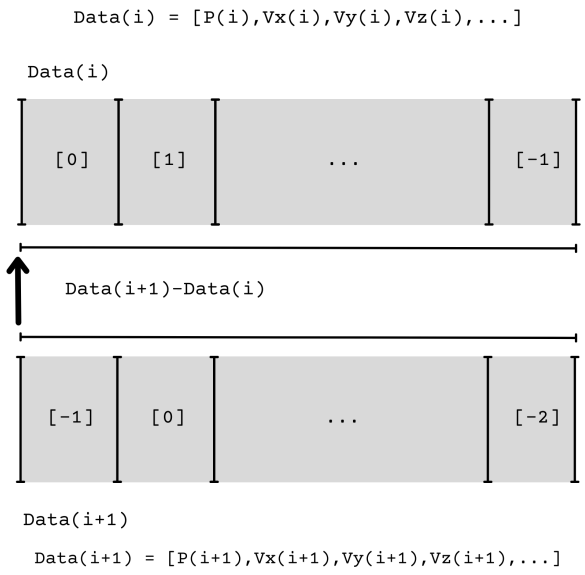

Figura 4.2: Técnica de doble buffer aplicada en el algoritmo FDTD.

La aplicación de dicha solución en el algoritmo de ejecución se usa tanto en los datos espaciales del mallado presión, figura 4.3, como en los de velocidad,

los cuales se almacenan en el mismo array Data, junto al resto de datos del entorno. De esta forma, la ejecución del cálculo de las diferencias depende de un bucle temporal, disminuyendo el tiempo de ejecución. En la ejecución del algoritmo se puede produce un crecimiento de tiempo n-potencial, si aumentan tanto la resolución del mallado como el volumen contenedor de nodos de la simulación, figura 4.4.

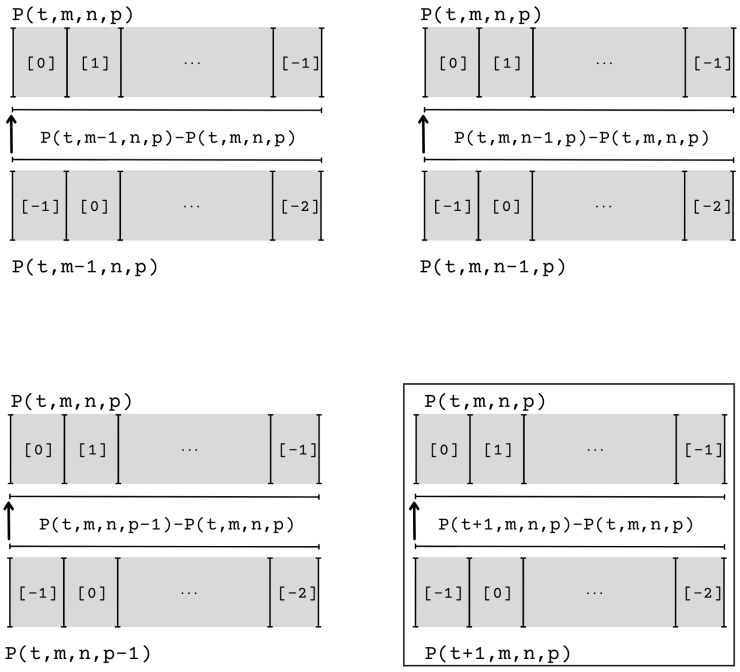

Figura 4.3: Diagrama de cálculo de diferencias para los datos de presión

Comparando el incremento de tiempo de ejecución para ambos casos, se puede apreciar que el incremento no es lineal, 4.5. Lo cual, puede ser debido a efectos derivados de la gestión y uso de la memoria del propio equipo.

4.2. Condiciones de contorno

El borde o contorno en el entorno de simulación es la región límite del mallado, compuesta por la capa de nodos que limitan el volumen de simulación. Es en esta capa donde, atendiendo al comportamiento del algoritmo de propagación FDTD, se aplican las condiciones de reflexión o absorción de las ondas.

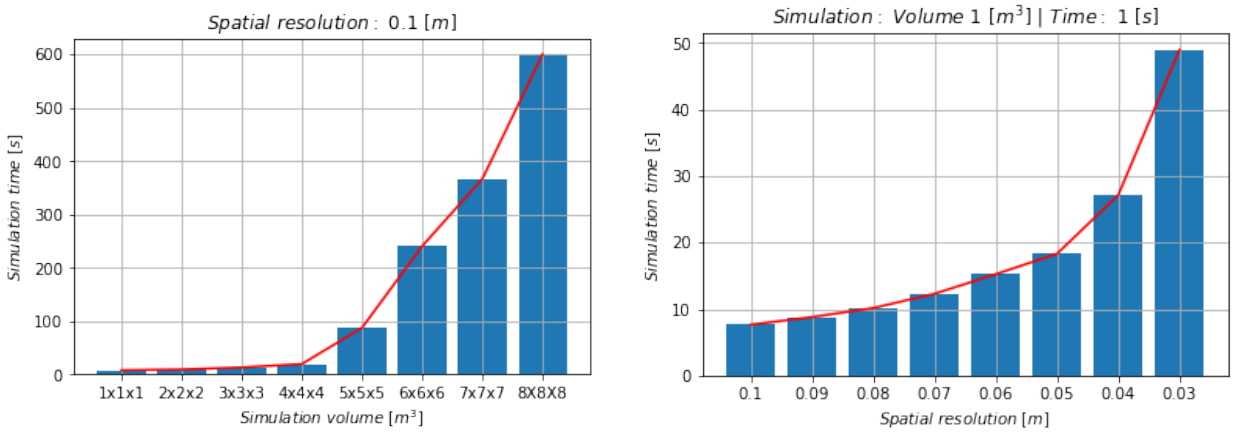

Figura 4.4: Tiempo de simulación aumentando el volumen contenedor de nodos y la resolución

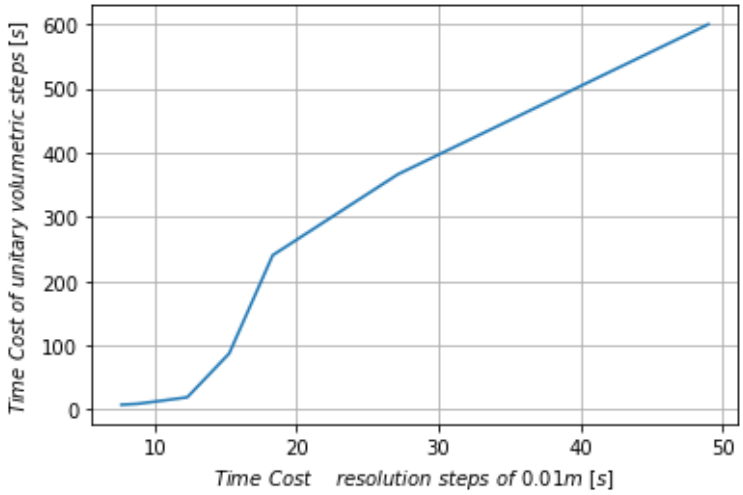

Figura 4.5: Incremento de tiempo de ejecución con la API desarrollada

4.2.1. Condiciones de contorno reflectante

El algoritmo FDTD, descrito en la sección 4.1.2, introduce las condiciones de contorno perfectamente reflectantes. Las ondas que llegan a los nodos limítrofes del volumen de simulación son reflejadas completamente, produciéndose ruido estacionario en las simulaciones. En la figura 4.6 se compara la implementación de la señal impulsiva en **REVISAR**.

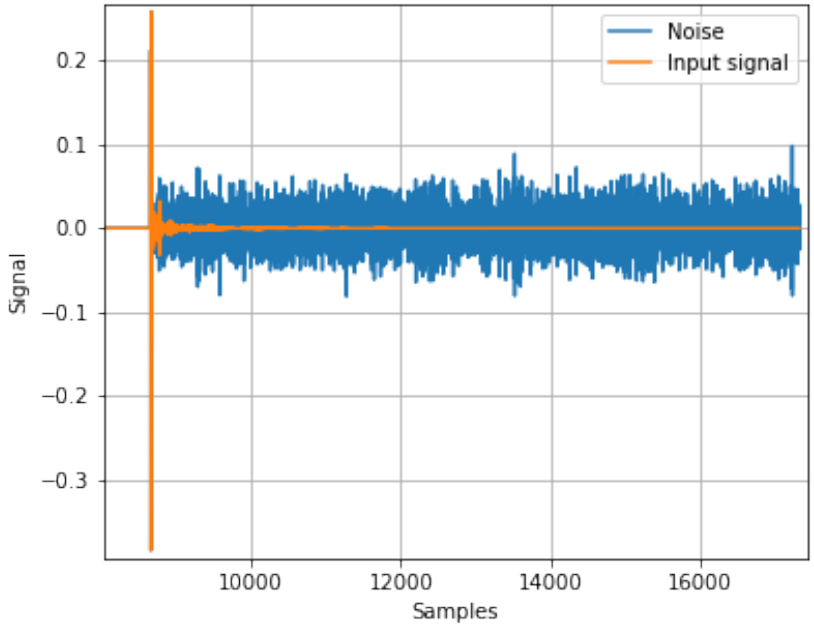

Figura 4.6: Señal impulsiva y ruido estacionario causado por reflexiones perfectas

4.2.2. Condiciones de contorno absorbente

Para la aplicación de condiciones absorbentes en el entorno de simulación, se aplica el algoritmo PML, comentado previamente en la sección 2.5.2, para absorber las ondas evitando la producción de reflexiones.

Para la implementación del método PML en los bordes exteriores de la simulación, se ha adaptado el algoritmo para hibridar el método con el FDTD desarrollado. Su implementación implica un mayor número de cálculos, por lo que se realizan algunas modificaciones del mismo para aplicar el método sin que aumente tanto el tiempo de ejecución.

La dimensión del contorno exterior reservado para la amortiguación de la onda es de un píxel, a diferencia de los 12 que se recomienda para la aplicación del mismo. Este cambio es aceptable gracias a las pequeñas reflexiones producidas en las esquinas del volumen, que son de varios órdenes de magnitud menores a las ondas absorbidas.

4.3. Respuesta Impulsiva

Cuando se aplica la respuesta impulsiva en un espacio y tiempo discretos, se toma en consideración la ecuación 2.13 de la sección 2.3. Ya que, desde el punto de vista discreto, una delta de dirac se puede representar como una función impulso muy estrecha para la obtención de la respuesta impulsiva $h_{ij}(t)$.

4.3.1. Implementación de la función impulso

La función impulso produce divergencias debido a la gran cantidad de componentes en frecuencias que la componen, por ello, es necesario eliminar las componentes de alta frecuencia de la fuente impulsiva, obteniendo convergencia en la simulación.

A partir del desarrollo en series de fourier de la función impulso, se puede aproximar a una función sinc(t) o una gaussiana, ecuación 4.5. Como resultado se modela una fuente de presión puntual con comportamiento impulsivo, figura 4.7,

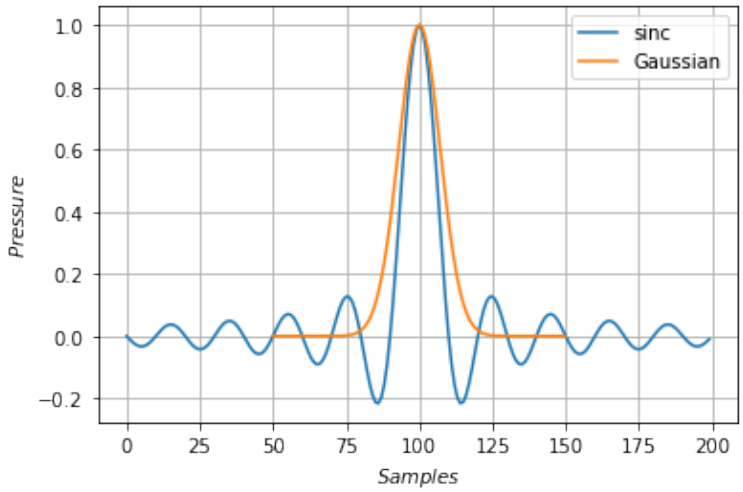

Figura 4.7: Función sinc y gaussiana como fuentes de presión diferenciable

$$P_\delta(t) = \frac{sin(\frac{t}{\varepsilon})}{\pi t} \;\; ; \;\; P_\delta(t) = \frac{e^{\frac{-t^2}{4\varepsilon}}}{2\sqrt{\pi}\varepsilon} \tag{4.5}$$

donde ε es una constante que modula el ancho.

Filtrado de la función impulso

Se filtran las componentes de alta frecuencia con la aplicación de un filtro pasobajo, diseñado para ajustar el filtrado a la máxima frecuencia posible en el entorno de simulación, figura 4.8.

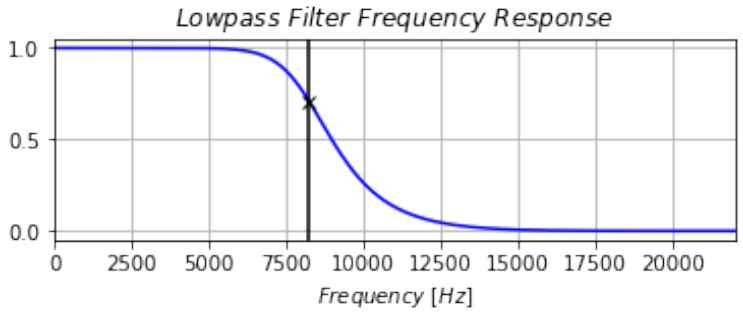

Figura 4.8: Respuesta en frecuencia del filtro pasobajo aplicado ($\Delta x = 0{,}1$ [m])

Este filtrado, dependiente de la resolución temporal y a su vez de la resolución espacial, se aplica en la fuente impulsiva para evitar propagación de aliasing, obtieniendo un contorno derivable de la función impulso, disminuyendo la divergencia en la propagación del algoritmo FDTD, figura 4.9

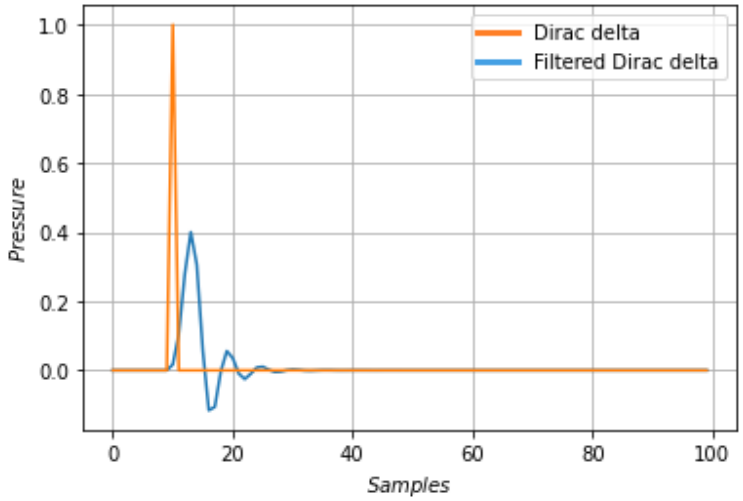

Figura 4.9: Fuente impulsiva filtrada y sin filtrar

Capítulo 5

Validación y resultados

En el presente capítulo se desarrolla un caso con solución exacta a un problema analítico para validar la simulación en base a la fenomenología acústica. Para ello se desarrolla un caso con solución exacta y se desarrolla un programa de simulación con la API descrita en el capítulo 3.

5.1. Caso con solución exacta

Para desarrollar una solución exacta se parte de la ecuación 2.8. La cual, en coordenadas esféricas $(r,\ \theta,\ \phi)$, es expresada como se ve en la ecuación 5.1,

$$\frac{1}{r^2}\frac{\partial}{\partial r}(r^2\frac{\partial P}{\partial r}) - \frac{1}{c_0^2}\frac{\partial^2 P}{\partial t^2} = 0 \tag{5.1}$$

teniendo en cuenta que se cumplen las propiedades de simetría en las coordenadas radiales θ y ϕ, como se muestra en la expresión 5.2.

$$\left.\frac{\partial P}{\partial \theta}\right|_r = 0 \ ; \ \left.\frac{\partial P}{\partial \phi}\right|_r = 0 \tag{5.2}$$

De esta forma y teniendo en cuenta que $P(r,t) = A(r)e^{iw_0 t}$, donde $A(r)$ es la amplitud en función de la distancia a la fuente y w_0 la frecuencia angular de la fuente, se desarrolla la expresión 5.3.

$$\frac{1}{r^2}\frac{\partial}{\partial r}(r^2\frac{\partial P}{\partial r}) = \frac{2}{r}\frac{\partial P}{\partial r} + \frac{\partial^2 P}{\partial r^2} \tag{5.3}$$

Teniendo en consideración de que para $r \gg 0$ el término $1/r$ se anula, la solución a la que se llega a partir de este planteamiento corresponde a un campo de presiones $P(r,t)$ como se ve en la ecuación 5.4, al que se le aplica un término de dispersión esférica en la intensidad de la onda ($I = |P(r,t)|^2$).

$$P(r,t) = \frac{A_0}{2\sqrt{\pi}r} e^{i(w_0 t - kr)} \tag{5.4}$$

Desarrollando la expresión teniendo en cuenta que para una fuente de tipo monopolo en frecuencias, $F_s(w) = \mathfrak{F}\{f(t)\}$, se tiene como solución en coordenadas esféricas, la expresión 5.5,

$$p(r,w) = -\frac{iw}{4\pi c_0^2 r} e^{ikr} F_s(w) \tag{5.5}$$

a la que se aplica una fuente continua sinusoidal $f(t) = A_0 sin(w_0 t)$, de amplitud A_0 y frecuencia w_0, permitiendo obtener el campo de intensidades en el dominio temporal, correspondiente a la ecuación 5.6,

$$I(r,t) = |P(r,t)|^2 = \frac{A_0^2}{4c_0^4 r^2} \tag{5.6}$$

y el campo de presiones, que toma la expresión 5.7,

$$P(r,t) = \frac{A_0'}{r} e^{i(w_0 t - kr)} \tag{5.7}$$

con $A_0' = \frac{A_0}{2c_0^2}$.

Este resultado corresponde a un campo generado por un monopolo, lo que permite validar el funcionamiento de la API desarrollada.

5.2. Simulación del caso elegido

El problema que se resuelve se basa en la propagación de la onda con absorción en los contornos PML, por lo que teniendo en cuenta esta premisa, se coloca una fuente centrada en el entorno de simulación y se comprueba la validez de los datos en distintos puntos del mallado.

El programa de simulación para validar la simulación se encuentra en el Anexo E.1 y ha sido escrito en base a la API desarrollada. Los parámetros de entrada

se encuentran descritos en la tabla 5.1 y en la figura 5.1 se representa visualmente el montaje de la simulación.

Room	dims = [2.2, 2.2, 2.2]
	dres = 0.1
	pini = 0.1
Source	sample_rate = 44100
	t_0 = 0.
	duration_src = 0.5
	phase_src = 0.
	coords_src = [1.1, 1.1, 1.1]
	freq_src = 200.
Sim	sim_type = 'pml'
	sim_time = 1.3
Microphone	mic0.coords = [1.1, 1.1, 1.1]
	mic1.coords = [0.1, 0.5, 0.5]
	mic2.coords = [0.2, 0.5, 0.5]
	mic3.coords = [0.3, 0.5, 0.5]
	mic4.coords = [0.4, 0.5, 0.5]
	mic5.coords = [0.5, 0.5, 0.5]

Tabla 5.1: Parámetros de entrada para la simulación de validación

Los micrófonos se encuentran a las distancias, r_i; $(i = 1, ..., 5)$ con valores:

$$r_1 = 1{,}311\ [m], \quad r_2 = 1{,}237\ [m], \quad r_3 = 1{,}166\ [m], \quad r_4 = 1{,}1\ [m], \quad r_5 = 1{,}039\ [m]$$

En esta simulación, los micrófonos han obtenido datos sobre la intensidad acústica que decae de forma exponencial en dB, figura 5.2, mientras que la presión acústica media obtenida por los mismos disminuye con relación $1/r$, figura 5.3. Se puede ver que el comportamiento de la simulación difiere cuanto más cerca está el micrófono del borde, ya que es donde se concentran los errores prevenientes de la adaptación que se hizo del método PML.

El error asociado a las medidas de los distintos micrófonos está representado en la figura 5.4, constituyendo una desviación mínima de los valores analíticos.

Los resultados obtenidos para la validación de la API desarrollada están dentro de unos márgenes aceptables de uso, teniendo en cuenta la característica de los errores incrementales con la aproximación a los límites del entorno de simulación.

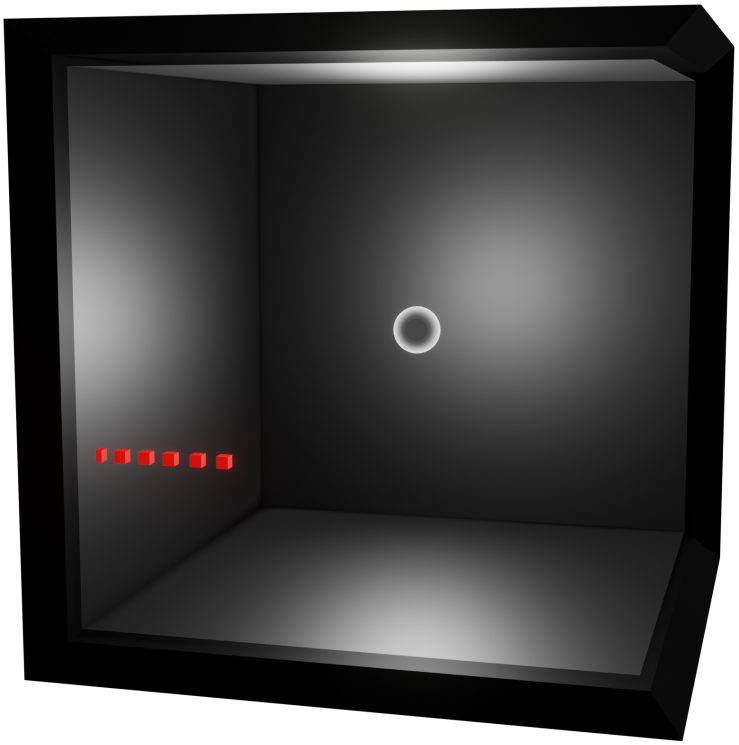

Figura 5.1: Representación 3D del *setup* para validación de la herramienta desarrollada. El entorno de simulación representado como un cubo con el contorno PML en negro, conteniendo en su interior una esfera (la fuente) y cubos rojos (los micrófonos)

En la figura 5.5, se pueden ver planos transversales de la simulación en 3D, lo que ayuda a ver los fenómenos de distorsión que ocurren en los bordes.

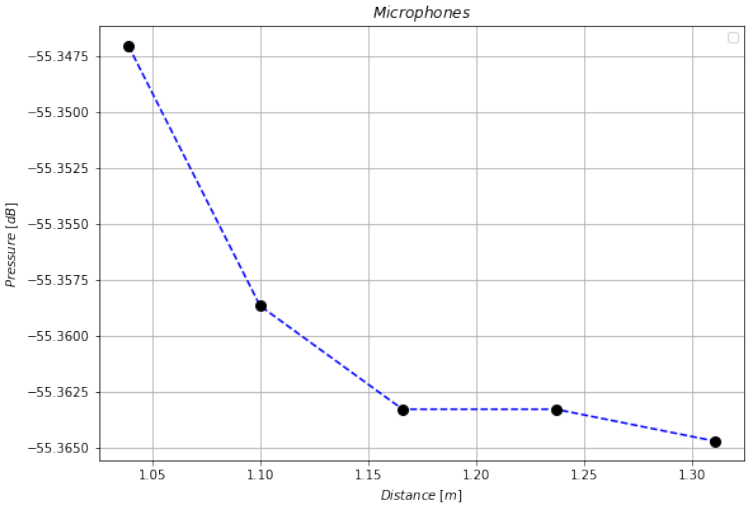

Figura 5.2: Intensidad acústica obtenida por los micrófonos con la simulación de validación **valar.py**

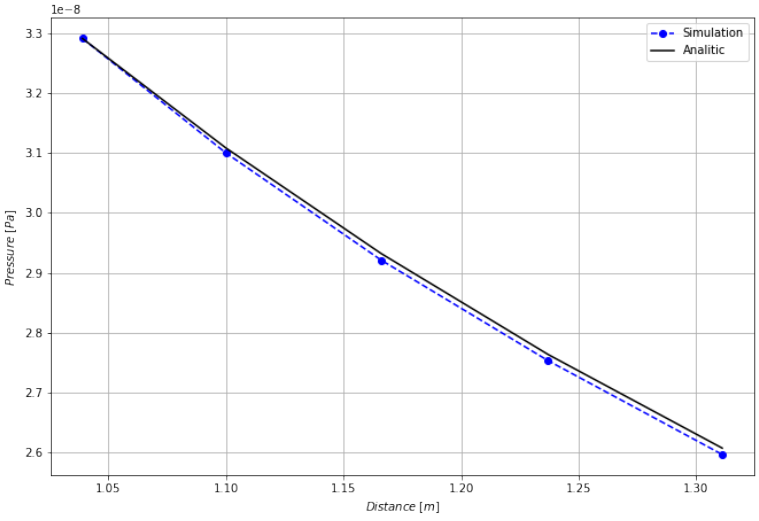

Figura 5.3: Presión acústica media obtenida por los micrófonos con la simulación de validación **valar.py** y resultado analítico

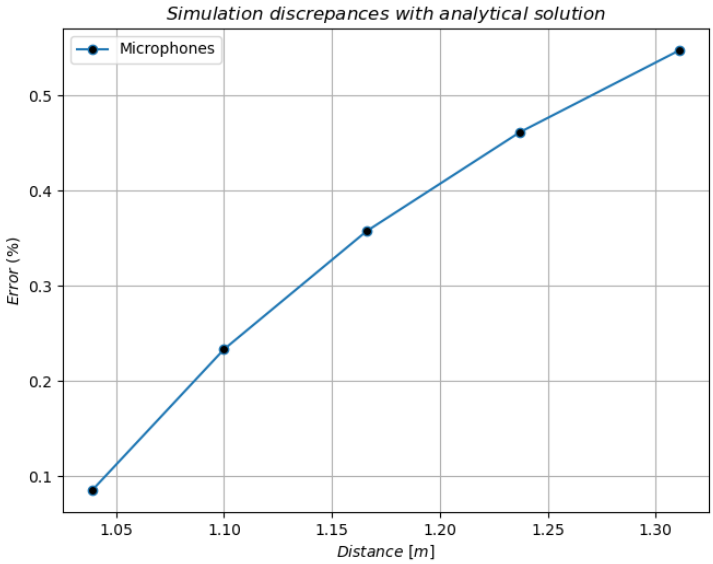

Figura 5.4: Intensidad acústica obtenida por los micrófonos con la simulación de validación **valar.py**

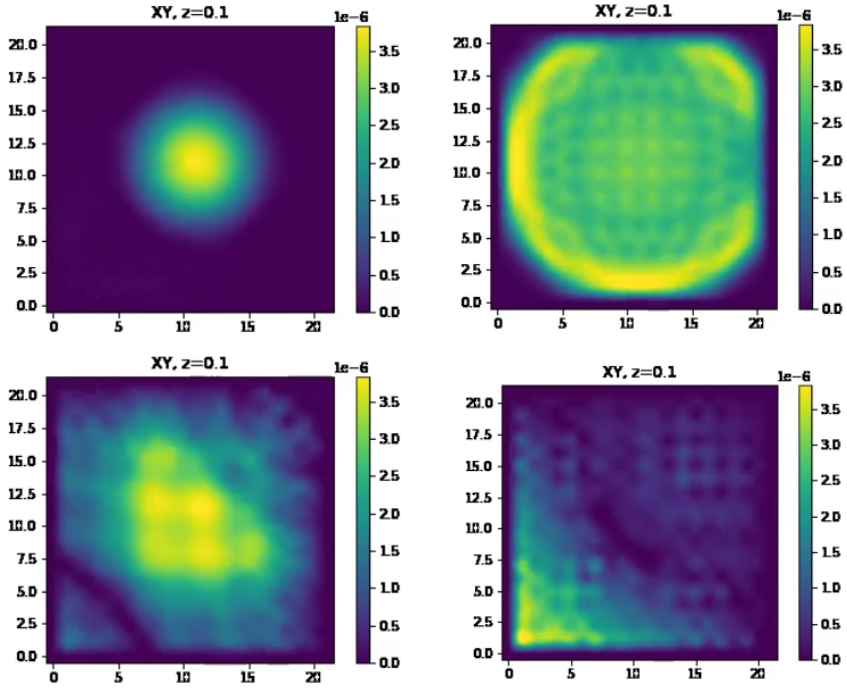

Figura 5.5: Planos transversales de simulación de validación con **valar.py** para distintos instantes de tiempo.

Capítulo 6

Conclusiones y trabajo futuro

La API desarrollada permite un buen control del entorno de simulación y los fenómenos acústicos y sus parámetros, simulando correctamente el fenómeno de propagación acústica en ordenadores de sobremesa, sin requerir de un tiempo ni potencia elevados. Además, está desarrollada para ser ejecutada por CPU, lo que la hace más lenta que con GPU, pero más accesible para sus futuras mejoras e implementaciones.

Se puede ver que el algoritmo FDTD converge en la simulación, cumpliendo con las expectativas requeridas inicialmente, mientras que la implementación del algoritmo PML se ha realizado dentro de unos márgenes de error inferiores al 1 % en los bordes más problemáticos del entorno de simulación.

Además, aplicando condiciones de contorno completamente reflectantes, figura 6.1, los resultados no divergen gracias a la condición CFL, mientras que aplicando las condiciones completamente absorbentes no se generan reflexiones notables no deseadas, figura 6.2.

6.1. Trabajo futuro

Tras el desarrollo de este trabajo se abren varias líneas de desarrollo. La más inmediata, es la mejora y ampliación de funcionalidades de la API presentada, además de la generación de documentación técnica para su completo aprovechamiento por la comunidad.

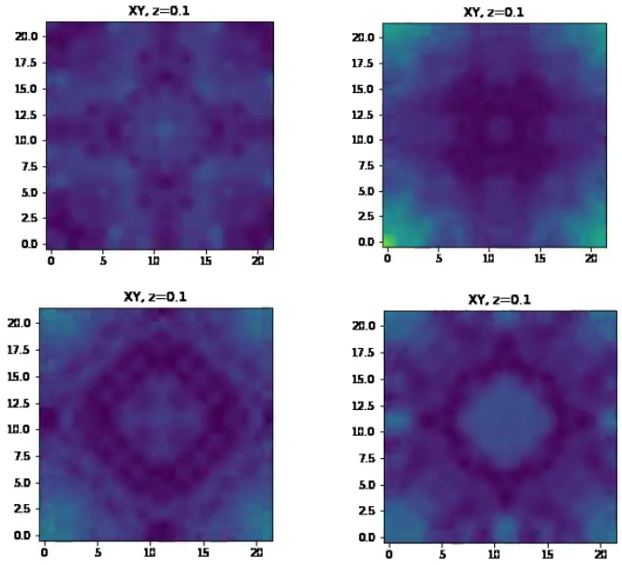

Figura 6.1: Ruido estacionario y modos normales en el entorno de simulación para condiciones de contorno reflectantes

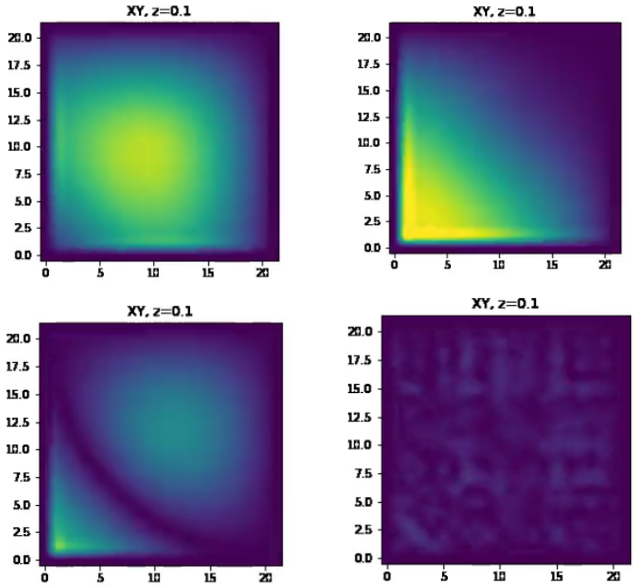

Figura 6.2: Absorción de la onda acústica por el algoritmo PML

Por otro lado, se queda pendiente la caracterización acústica de las piscinas de orcas del Loro Parque, las cuales han sido modeladas en 3D, figura 6.3 en Blender, para su uso en simulaciones acústicas generadas a partir de la API.

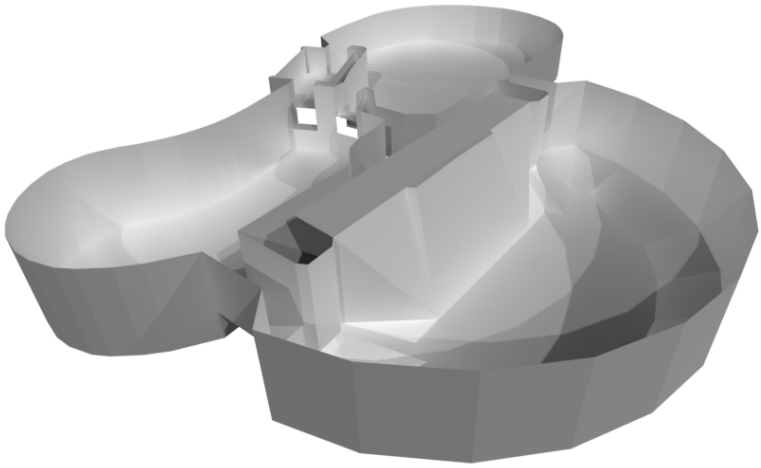

Figura 6.3: Representación 3D de las piscinas de *OrcaOcean* en Loro Parque, Puerto de la Cruz, Tenerife

Bibliografía

[1] Mike Barron. «Using the standard on objective measures for concert auditoria, ISO 3382, to give reliable results». En: *Acoustical Science and Technology* 26.2 (2005), págs. 162-169.

[2] Densil Cabrera, Jianyang Xun y Martin Guski. «Calculating reverberation time from impulse responses: A comparison of software implementations». En: *Acoustics Australia* 44.2 (2016), págs. 369-378.

[3] António Pedro O. Carvalho. «The use of the Sabine and Eyring reverberation time equations to churches». En: *The Journal of the Acoustical Society of America* 97.5 (1995), págs. 3319-3319. DOI: 10.1121/1.412850. eprint: https://doi.org/10.1121/1.412850. URL: https://doi.org/10.1121/1.412850.

[4] National Research Council y col. «Low-frequency sound and marine mammals: Current knowledge and research needs». En: (1994).

[5] Carlos A De Moura y Carlos S Kubrusly. «The courant–friedrichs–lewy (cfl) condition». En: *AMC* 10.12 (2013).

[6] James Eaton y col. «Estimation of Room Acoustic Parameters: The ACE Challenge». En: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 24.10 (2016), págs. 1681-1693. DOI: 10.1109/TASLP.2016.2577502.

[7] Hugh D Geiger y Pat F Daley. «Finite difference modelling of the full acoustic wave equation in Matlab». En: *Acoustic Finite Difference Modeling, CREWES Research Report* 15 (2003), págs. 1-9.

[8] Kajishima. *Computational Fluid Dynamics*. eng. Springer International Publishing, 2017. ISBN: 3-319-45302-5.

[9] D Kawai y col. «Acoustic FDTD analysis considering viscous and advective effects». En: *Forum Acusticum*. 2011, págs. 2583-2587.

[10] Yang-Hann Kim. *Sound propagation: an impedance based approach*. John Wiley & Sons, 2010.

[11] Qing-Huo Liu y Jianping Tao. «The perfectly matched layer for acoustic waves in absorptive media». En: *The Journal of the Acoustical Society of America* 102.4 (1997), págs. 2072-2082.

[12] Mirosław Meissner. «Acoustics of small rectangular rooms: Analytical and numerical determination of reverberation parameters». En: *Applied Acoustics* 120 (2017), págs. 111-119.

[13] Sjoerd W Rienstra y Avraham Hirschberg. «An introduction to acoustics». En: *Eindhoven University of Technology* 18 (2004), pág. 19.

[14] Lauri Savioja. «Real-time 3D finite-difference time-domain simulation of low-and mid-frequency room acoustics». En: *13th Int. Conf on Digital Audio Effects*. Vol. 1. 2010, pág. 75.

[15] John B Schneider. «Understanding the finite-difference time-domain method». En: *School of electrical engineering and computer science Washington State University* 28 (2010).

[16] Rick Scholte, Bert Roozen e Ines Lopez. «On spatial sampling and aliasing in acoustic imaging». En: *12th Intern. congress on sound and vibration, Lisbon, Portugal*. 2005.

[17] Manfred R Schroeder. «New method of measuring reverberation time». En: *The Journal of the Acoustical Society of America* 37.6 (1965), págs. 1187-1188.

[18] Magne Skålevik. «Reverberation Time–The mother of all room acoustic parameters». En: *CD Proceedings of 20th International Congress on Acoustic, ICA*. Vol. 10. 2010.

[19] Otto Von Estorff. «Numerical methods in acoustics: facts, fears, future». En: *Revista de Acústica* 38.3-4 (2007), págs. 83-101.

[20] MNH Zahari, SH Dahlan y A Madun. «A review of acoustic fdtd simulation technique and its application to underground cavity detection». En: *International Conference on Electrical and Electronic Engineering*. Citeseer. 2015.

Facultad de Ciencias
Sección de Física
Universidad de La Laguna

FACULTAD DE CIENCIAS
SECCIÓN DE FÍSICA

Trabajo de Fin de Grado

Desarrollo de una simulación FDTD para la estimación de parámetros acústicos en entornos cerrados

TOMO II

Anexos

Titulación: Grado en Física

Estudiante:

Elías Gabriel Ferrer Jorge

Tutor: Fernando Luis Rosa González

2 de Febrero de 2022

Apéndice A

Resumen de los capítulos

A.1. Capítulo 2. Fundamentos

Para definir las características del entorno se realiza una introducción a la fenomenología acústica desde la dinámica de fluidos y la termodinámica. Concluye con la ecuación de onda acústica, desarrollada a partir de un sistema de ecuaciones acopladas de los campos de presión y velocidad. A continuación, se procede con una aproximación teórica a la respuesta impulsiva y su aplicación en la obtención de diferentes parámetros acústicos, como el tiempo de reverberación y su importancia en los trabajos de caracterización acústica.

A.2. Capítulo 3. Desarrollo de la API

Se define la estructura de clases y funciones desarrolladas que componen la API orientada a la metodología FDTD, mostrando los diferentes parámetros y métodos que pertenecen a las clases.

A.3. Capítulo 4. Simulación FDTD

Se desarrolla el algoritmo de la metodología FDTD aplicada y la discretización de la ecuación de onda, procediendo a la aplicación de diferentes condiciones de contorno aplicables con la API desarrollada. Finalmente, se desarrolla cómo se ha abordado la discretización de la función impulsiva para obtener la respuesta impulsiva del entorno y las premisas que se han tomado para ello.

A.4. Capítulo 5. Validación y resultados

Se propone un problema analítico que puede ser simulado creando un programa de simulación a partir de la API diseñada, con el fin de estudiar sus características en la aplicación de los métodos utilizados y sus limitaciones.

A.5. Capítulo 6. Conclusiones y trabajo futuro

Se concluye con un análisis de los resultados adquiridos a lo largo del desarrollo de este trabajo y se proponen futuras líneas de investigación orientadas a mejorar la API y su uso para el estudio de las piscinas de orcas en Loro Parque.

Apéndice B

Summaries of chapters

B.1. Chapter 2. Fundamentals

In order to define the characteristics of the environment, an introduction to acoustic phenomenology is made from fluid dynamics and thermodynamics. It concludes with the acoustic wave equation, developed from a system of coupled equations of the pressure and velocity fields. Next, it proceeds with a theoretical approach to the impulse response and its application in obtaining different acoustic parameters, such as reverberation time and its importance in the work of acoustic characterization.

B.2. Chapter 3. API development

The structure of classes and developed functions that make up the API oriented to FDTD methodology are defined, showing the different parameters and methods that belong to the classes.

B.3. Chapter 4. FDTD Simulation

Further development of the applied FDTD methodology and the discretization of the wave equation, proceeding with the application of different applicable boundary conditions with the developed API. Finally, it is developed how the discretization of the impulsive function has been approached to obtain the impulsive response of the environment and the premises that have been taken for it.

B.4. Chapter 5. API's validation and results

An analytical problem is proposed that can be modeled by creating a simulation program from the designed API, in order to study its characteristics in the application of the methods used and its limitations.

B.5. Chapter 6. Conclusions and future

It concludes with an analysis of the results acquired throughout the development of this work and future research lines are proposed aimed at improving the API and its use for the study of orca pools in Loro Parque.

Apéndice C

Clases de la API

C.1. *Room*

```
#coding:utf-8
'''
TFG Acoustics Simulations

ROOM CLASS

@author Elias Gabriel Ferrer Jorge
'''

import numpy as np
import fdtd.medium as me

class Room:
  '''
  Room mesh generator for FDTD Method: Pressure and Velocity mesh and physical
  parameters in the room.


  '''

  def __init__(self, dims=[1,1,1], dres=0.1, pini=[0,0,0]):
    '''
    INPUTS:

    dims -> (list of floats) Dimensions of room (x,y,z) [meters]
    Example: [1.3,1.3,1.3]

    dres -> (float) Mesh resolution or space between nodes in mesh [meters]
    Example: 0.1

    pini -> (list of floats) Initial point in general coordinates (x,y,z) where mesh is created [meters]
    Example: [0.,0.,0.]
    '''

    self.dims   = np.zeros(len(dims),dtype='int')
    self.dres   = dres
    self.pini   = np.zeros(len(pini),dtype='float')

    for n in range(len(dims)):

      self.dims[n] = int(dims[n]/self.dres)
      self.pini[n] = float(pini[n])

```

```
44      self.dims_m = self.dims * self.dres #dims in meters
45      return
46
47    def __str__(self):
48      '''Information of room instance
49      '''
50      cad = '+' + '-'*80 + '+\n'
51      cad+= '|'+ ' '*37 + ' Room' + ' '*37 + ' |\n'
52      cad+= '+'+'-'*80+'+\n'
53      cad+= 'Mesh resolution: %.2f [nodes/m]\n' %(1./self.dres)
54      cad+= 'Dimension of mesh: (%.2f x %.2f x %.2f)[m^3]\n' %(
        self.dims_m[0],self.dims_m[1],self.dims_m[2])
55      cad+= 'Total number of nodes: %s\n' %(self.dims[0]*self.dims
        [1]*self.dims[2])
56      cad+= '-'*80 + '\n\n'
57      return(cad)
```

C.2. *Medium*

```
#coding:utf-8
'''
TFG Acoustics Simulations

MEDIUM CLASS

@author Elias Gabriel Ferrer Jorge
'''

import numpy as np


class Medium:
  '''
  Medium functions:

  change_medium(co,cr,co)

  add_zone_density(dims_ro, pini_ro, ro)

  add_ wall_medium(dims_med, pini_med, c0,cr,ro,) -> In process
  '''

  def __init__(self,room, medium='Water', c0=1500., cr0=1., r0h
    =1200):
    '''
    Physical parameters:

    c0 -> (float) Sound velocity in medium

    cr -> (np.ndarray) Relative velocity Sound mesh

    ro -> (np.ndarray) Medium density mesh

    Inital physical parametes in room:

      Homogeneous medium: Air
        co = 340.
        cr = 1 * np.ones(dims, dtype='float')
        ro = 1.225 * np.ones(dims, dtype='float')
        gamma = 2e-3 s/m2

      Homogeneous medium: Water
        gamma = 2e-3 s/m2
    '''

```

```
46      self.room = room
47
48      self.medium_name = medium
49      self.c0 = c0
50      self.cr = cr0*np.ones(self.room.dims, dtype='float')
51      self.ro_homogeneous = r0h
52      self.ro = self.ro_homogeneous * np.ones(self.room.dims,
    dtype='float')
53      self.pref = 1*10**-6
54
55    def change_medium(self, medium_name ,pref, c0, cr, ro):
56      '''
57      Change physical parameters of medium
58
59      INPUTS:
60        medium_name -> (str) Name of medium defined
61        c0      -> (float) Sound velocity in medium [m/s]
62        cr      -> (float) Relative velocity Sound [m/s]
63        ro      -> (float) Medium density [kg/m3]
64
65      Example:
66        change_medium(medium_name = 'water', c0 = 331 ,cr = 1, ro
    = 1.29)
67        change_medium(medium_name = 'saltwater', c0 = 1533 ,cr =
    1, ro = 1027)
68      '''
69
70      if type(medium_name) == str:
71        self.medium_name = medium_name
72      elif type(medium_name) != str:
73        raise ValueError('medium_name type must to be string type'
    )
74
75      if c0 == 0:
76        raise ValueError('c0 value cannot by 0')
77      elif c0 !=0:
78        self.c0 = c0
79
80      if type(cr) == int:
81        self.cr = cr*np.ones(self.room.dims, dtype='float')
82      else:
83        self.cr = cr
84
85      if type(ro) == int:
86        self.ro_homogeneous = ro
87        self.ro = self.ro_homogeneous * np.ones(self.room.dims,
    dtype='float')
88      else:
89        self.ro_homogeneous = ro
```

```
90
91      self.pref = pref
92
93    def add_zone_density(self, dims_ro, pini_ro, ro):
94      '''
95      Add Homogeneous density zone
96
97      INPUTS:
98
99        dims_ro -> (list of floats) Dimensions of room (x,y,z) [m]
100         Example: [0.3,0.3,0.3]
101
102       pini_ro -> (list of floats) Initial point in general
      coordinates (x,y,z) where mesh is created [m]
103         Example: [0.,0.,0.]
104
105       ro      -> ro -> (float or matrix) wall density [kg/m3]
106         Example: 150.
107     '''
108     if ro == 0:
109       raise ValueError('ro value cannot by 0')
110     else:
111       self.dims_ro = np.array(np.array(dims_ro)/self.room.dres,
      int)
112       self.pini_ro = np.array(np.array(pini_ro)/self.room.dres,
      int)
113
114       self.ro[self.pini_ro[0]:self.dims_ro[0]+self.pini_ro[0],
      self.pini_ro[1]:self.dims_ro[1]+self.pini_ro[1],self.pini_ro
      [2]:self.dims_ro[2]+self.pini_ro[2]] = ro
115   def __str__(self):
116     '''Information of medium object
117     '''
118     cad = '+' + '-'*80 + '+\n'
119     cad+= '|'+ ' '*38 + ' Medium' + ' '*38 + ' |\n'
120     cad = '+' + '-'*80 + '+\n'
121     cad+= 'Medium: %10s\n'%(self.medium_name)
122     cad+= 'Sound velocity: co = %.2f [m/s]\n' %(self.c0)
123     cad+= 'Homogeneous density: ro = %.2f [Kg/m3]\n' %(self.
      ro_homogeneous)
124     return(cad)
```

C.3. *Source*

```
#coding:utf-8
'''
TFG Acoustics Simulations

SOURCE CLASS

@author: Elias Gabriel Ferrer Jorge
'''
import numpy as np
from scipy import signal
import fdtd.room as ro
import fdtd.medium as m
import fdtd.counter as co
import fdtd.filter as fil


class Source:
  '''
  Punctual isotropic Sources of pressure
  '''
  def __init__(self, label="Main Source", amplitude = 1*10**-6,
   sample_rate=44100, t0 = 0, duration=1, frequency=980, phase
   =0, coords_0 = [0.5,0.5,0.5], v = [0.,0.,0.]):
    '''
    INPUTS:
      sample_rate -> (int)     Samples per second [samples/s]
      t0          -> (float)   Initial time of source [s]
      duration    -> (float)   Duration of source [s]
      frequency   -> (float)   Frequency of tone [Hz]
      phase       -> (float)   Phase of wave of source [rad]
      coords      -> (ndarray) Coordinates (x,y,z) of source [m]



      amplitude = 1*10e-6 [Pa] is the reference value IN WATER
      amplitude = 20*10e-6 [Pa] is the reference value IN AIR
    '''
    self.type_source   = 'Punctual isotropic Source of pressure'
    self.label         = label
    self.amplitude     = amplitude
    self.sample_rate   = sample_rate
    self.frequency     = frequency
    self.t0            = t0
    self.duration      = duration
    self.phase         = phase

```

```
45      self.coords_0       = np.array(coords_0)
46      self.coords_n       = np.array(coords_0)
47
48      self.v              = np.array(v)
49
50      self.t              = np.arange(self.t0,self.duration+self.t0
        ,1./self.sample_rate)
51
52      self.tone = self.amplitude * np.sin(2*np.pi * self.frequency
        *self.t + self.phase)
53
54      self.create_v_in_t()
55
56
57  def __call__(self, t):
58      ''' This obtain pressure of source at its own time t [s]
59      '''
60      self.coords_n = self.coords_0 + self.v * t
61
62      self.tone = self.amplitude * np.sin(2*np.pi * self.frequency
        *self.t + self.phase)
63
64      return self.tone
65
66  def retone(self,t_sim ,dt):
67  '''Resampling of source in a specific simulation
68  '''
69  self.sample_rate = 1./dt
70  self.t              = np.arange(self.t0,self.duration+self.t0
        ,1./self.sample_rate)
71
72  self.tone = self.amplitude * np.sin(2*np.pi * self.frequency*
        self.t + self.phase)
73
74  if self.t0>0:
75      self.tone = np.array(np.zeros(int(self.t0/dt)).tolist() +
        self.tone.tolist())
76  if self.t[-1]<t_sim and self.t0>0:
77      self.tone = np.array(self.tone.tolist() + np.zeros(int((
        t_sim-self.duration)/dt)).tolist())
78  if self.t[-1]<t_sim:
79      self.tone = np.array(self.tone.tolist() + np.zeros(int((
        t_sim-self.t[-1])/dt)).tolist())
80
81  def retone_delta(self,t_sim,dt):
82      '''
83      Dirac's delta
84      d(x-n)
85      '''
```

```
86      n = int(self.t0/dt)
87      x = int(t_sim/dt+1)
88      self.tone = signal.unit_impulse(x,n)
89
90    def create_v_in_t(self):
91    '''Velocity array data in source time length
92    '''
93      if len(np.shape(self.v)) > 1 and len(np.shape(self.t)) > 0:
94        if len(self.v)<len(self.t):
95          count = len(self.v)*(self.t[1]-self.t[0])
96          n = 0
97          v_list = []
98        for t in self.t:
99          if  t<count:
100           v_list.append(self.v[n])
101         else:
102           count += len(self.v)*(self.t[1]-self.t[0])
103           if n==len(self.v)-1:
104             v_list.append(self.v[n])
105         else:
106           n+=1
107           v_list.append(self.v[n])
108
109       self.v = np.array(v_list)
110
111   def d_dirac(self,dt):
112     '''
113     Dirac's delta
114     d(x-n)
115     '''
116
117     n = int(self.t0/dt)
118     x = int(self.duration/dt+1)
119     self.tone = signal.unit_impulse(x,n)
120
121
122   def filt(self,freq):
123     '''Lowpass Filtering of source
124
125       FILT AFTER retone() SOURCE!!!
126     '''
127     fl1 = fil.Filter(freq,self.sample_rate)
128     self.tone = fl1.filtsource(self.tone)
129
130   def __str__(self):
131     '''Information of object source
132     '''
133     cad = '\n'
134     cad+= '+' + '-'*80 + '+\n'
```

```
135     cad+= '|' + ' '*37  + 'Source' + ' '*37 + '|\n'
136     cad+= '+'+'-'*80+'+\n'
137     cad+= self.type_source + '\n'
138     cad+= 'Amplitude: %.e [Pa]\n' %(self.amplitude)
139     cad+= 'Sample rate: %.3f [samples/s]\n' %(self.sample_rate)
140     cad+= 'Frequency of source: %.2f [Hz]\n' %(self.frequency)
141     cad+= 'Duration: %.2f [s]\n' %(self.duration)
142     cad+= 'Phase: %.2f [rad]\n' %(self.phase)
143     cad+= 'Initial coordinates (x,y,z): %s [m]\n' %(str(self.
        coords_0))
144     cad+= 'Initial time: %.2f [s]\n'%(self.t0)
145     cad+= '-'*80 + '\n\n'
146     return(cad)
```

C.4. *Microphone*

```
#coding:utf-8
'''
TFG Acoustics Simulations

MICROPHONE CLASS

@author: Elias Gabriel Ferrer Jorge
'''

import numpy as np
import fdtd.datafiles as fl
import fdtd.wav as w
import os

def read_data_from_file(microphone_number, file_number,folder = 'micro_data'):
  '''
  '''
  return np.load(os.path.join(str(folder), 'mic_'+str(microphone_number) + '_' + str(file_number) + '.npy'))

class Microphone:
  '''
  '''
  def __init__(self, samples = 0, label= 'Main Microphone', t0 = 0., sensitivity_dB = 62, snr=1., acoustic_overload= 120 , directionality = None, frequency_response = None, coords = [0.,0.,0.]):
    '''MICROPHONE INPUTS
      micro_label: name of microphone (str)

      t0: time for turn on microphone (float) [s]

      adquisition_time: time acquiring data (float) [s]

      sensitivity_dB: value of microphone's response (float) [dB]

      snr: specifies the ratio of a reference signal to the noise level of the microphone output (float) [dB]

      acoustic_overload: maximum noise that can record microphone
      without distorsion (float) [dB]

```

```
38      directionality: pattern in which response changes when
    source
39      changes position in space (float) [dB]
40
41
42    OTHERS
43      sensitivity: value of microphone's response (float) [mV/Pa
    ]
44      ein: equivalent input noise -> {acoustic_overload - snr}
    (float)[dB]
45      dynamic_range: range of high quality aquisition's
    microphone (float)
46      thd: total harmonid distorsion is a measurement of the
    level
47      of distortion on the output signal for a given pure tone
48      input signal (float) [percent%]
49
50    '''
51
52    self.label = label
53
54    self.t0                 = t0
55    self.sensitivity_dB     = sensitivity_dB
56    self.acoustic_overload  = acoustic_overload
57    self.directionality     = directionality
58    self.frequency_response = frequency_response
59    self.coords             = coords
60    self.samples            = samples
61    self.data               = np.zeros(samples)
62
63  def resample(self):
64    '''
65    '''
66    self.data = np.zeros(self.steps)
67
68  def save_data(self, micro_label, data):
69    '''
70    '''
71    self.data = data
72    fl.record(micro_label,self.data)
73
74  def save_data_from_file(self, micro_label, filename):
75    '''
76    '''
77    for ti in range(self.t*self.fs):
78
79    self.data = fl.input(int(ti),folder = 'sim_data')
80
```

```
81      fl.record(micro_label,self.data[0,self.coords[0],self.coords
        [1],self.coords[2]])
82
83      def save_wav(self,name,t):
84      w.towav(name,t,self.data)
85      print('\n--Microphone array data exported to wav--\n')
86
```

C.5. *Sim*

```
#coding:utf-8
'''
TFG Acoustics Simulations

SIM CLASS
  FDTD and PML 3D Implementation

@author: Elias Gabriel Ferrer Jorge
'''

import numpy as np
import time
import fdtd.datafiles as fl
import fdtd.counter as co

SIM_NO_ERR = 0
SIM_ERR_FREQ = 1
SIM_ERR_RES = 2
SIM_ERR_DIMS = 4


class Sim:
  ''' Acoustic propagation solver FDTD with PML
  '''

  def __init__(self, sim_type, room, medium, sources, micros, t):
    '''Initialite Sim with room, medium, source and microphone instances

    '''
    #ERROR VALUES
    self.SIM_NO_ERR = 0
    self.SIM_ERR_FREQ = 1
    self.SIM_ERR_RES = 2
    self.SIM_ERR_DIMS = 4

    self.sim_type = sim_type


    self.room    = room
    self.medium = medium

    self.sources         = sources
    self.sources_coords = (np.array([sources.coords_n for sources in self.sources])/self.room.dres).astype(int)
```

```
44      self.source_index    = 0
45
46      self.micros            = micros
47      self.micros_coords     = (np.array([mics.coords for mics in
        self.micros])/self.room.dres).astype(int)
48      self.microphone_index = 0
49
50      self.t      = t
51      self.cadat = 0
52
53      self.sim_duration = 0
54      #Parameters. Courant Coefficient = (c0 * dt) * 1/2 / (np.
        sqrt(3) * dx)
55      self.sc = 0.5
56      self.dt = np.sqrt(3)*self.sc*self.room.dres / self.medium.c0
57      #self.sc = self.room.c0 * self.dt / self.room.dres/np.sqrt
        (3)
58
59      # if self.t > self.sc:
60      #    self.status = 1 #
61      #    self.t = self.sc
62
63
64      #PML_REGION
65      ##
66      self.dim_region = 1
67      self.w_max       = 0.
68      self.fpml        = np.zeros((7,3),dtype=float)
69
70
71      self.counter = co.Counter()
72      self.save_status = 0 # 0== Disabled save option | 1 ==
        Enabled save option
73      #self.cprv = self.room.ro * self.room.cr**2 * self.room.c0 *
         self.sc
74
75      #Data p,vx,vy,vz,ro,cprv
76      self.pvro = np.zeros((2,7,self.room.dims[0],self.room.dims
        [1],self.room.dims[2]),dtype=float)
77
78      self.pvro[:,4] = self.medium.ro
79      self.pvro[:,5] = self.pvro[:,4] * self.medium.cr**2 * self.
        medium.c0 * self.sc
80
81      #PML_REGION
82      ## w_i ; i = (x,y,z) => w_i = self.pvro[0,6]
83      ## D_i ; i = (x,y,z) => D_i = self.pvro[1,6]
84
85      # #Velocities
```

```
86      # self.vx = room.v
87      # self.vy = room.v
88      # self.vz = room.v
89      #
90      # #
91      # self.vxbf = room.v
92      # self.vybf = room.v
93      # self.vzbf = room.v
94      #
95      # self.pbf = room.p
96      #
97      # self.n = 0
98      # self.n_1 = 1
99
100     # #Simulation output Data
101     # self.data_sim_p = []
102     # self.data_sim_v = []
103
104     #Check data avoiding aliasing effects
105     self.status = SIM_NO_ERR
106
107     self.check_status()
108
109     for mic in micros:
110     mic.steps = int(self.t/self.dt)
111     mic.resample()
112
113
114   def calc(self, cadat):
115     if self.sim_type.lower()=='pml':
116       self.implement_prop_calc()
117       self.calc_pml_region()
118       self.save_sim()
119       self.counter.swap()
120
121     if self.sim_type.lower()=='nopml':
122       self.implement_prop_calc()
123       self.save_sim()
124       self.counter.swap()
125
126   def initial_cond(self, data):
127     '''Set inital conditions of pressure and velocity meshes as
      final state of
128     data file or explicit data ndarray
129
130     INPUT:
131       data: filename (str) or (ndarray) with shape (t,6,dims[0],
      dims[1],dims[2])
132
```

```
133       p    = data[-1,0]
134       vx   = data[-1,1]
135       vy   = data[-1,2]
136       vz   = data[-1,3]
137       ro   = data[-1,4]
138       cprv = data[-1,5]
139     '''
140
141     if type(data) == str:
142       data = fl.input(data)
143
144     elif type(data)[1] == 7:
145       data = data
146
147     #Pressure
148     self.pvro[0,0] = data[-1,0]
149
150     #Velocities
151     self.pvro[0,1] = data[-1,1]
152     self.pvro[0,2] = data[-1,2]
153     self.pvro[0,3] = data[-1,3]
154
155     #Density
156     self.pvro[:,4] = data[-1,4]
157
158     #Cprv
159     self.pvro[:,5] = data[-1,5]
160
161     #PML_Region
162     ##w_i
163     self.pvro[:,6] = data[-1,6]
164
165
166   def save(self, status):
167     '''
168     INPUT
169     status = 0 -> save  datafiles disabled
170     status = 1 -> save  datafiles enabled
171     '''
172     self.save_status =status
173
174   def init__micro(self,coords_micro):
175     self.coords_micro = coords_micro
176
177   def init_sources(self,sources):
178     '''
179     '''
180     for i in sources:
181       i.retone(self.t,self.dt)
```

```
182
183     self.sources = sources
184
185
186   def set_sc(self,sc):
187     '''Set value of Sc, Courant's coefficient
188     '''
189     self.sc = sc
190
191   def algorithm_prop(self):
192     '''FDTD 3D Algorithm
193     '''
194     alfa = 2*self.sc/self.medium.c0 #this parameter is constant
      in general cases
195
196     self.pvro[self.counter.n_1,1] =  self.pvro[self.counter.n,1]
      - alfa*(np.roll(self.pvro[self.counter.n,0],-1,axis=0)-self.
      pvro[self.counter.n,0])/(self.pvro[self.counter.n,4]+np.roll(
      self.pvro[self.counter.n,4],-1,axis=0))
197
198     self.pvro[self.counter.n_1,2] =  self.pvro[self.counter.n,2]
      - alfa*(np.roll(self.pvro[self.counter.n,0],-1,axis=1)-self.
      pvro[self.counter.n,0])/(self.pvro[self.counter.n,4]+np.roll(
      self.pvro[self.counter.n,4],-1,axis=1))
199
200     self.pvro[self.counter.n_1,3] =  self.pvro[self.counter.n,3]
      - alfa*(np.roll(self.pvro[self.counter.n,0],-1,axis=2)-self.
      pvro[self.counter.n,0])/(self.pvro[self.counter.n,4]+np.roll(
      self.pvro[self.counter.n,4],-1,axis=2))
201
202     self.pvro[self.counter.n_1,0] =  self.pvro[self.counter.n,0]
      - self.pvro[self.counter.n,5]*((self.pvro[self.counter.n_1
      ,1]-np.roll(self.pvro[self.counter.n_1,1],1,axis=0))+(self.
      pvro[self.counter.n_1,2]-np.roll(self.pvro[self.counter.n_1
      ,2],1,axis=1))+(self.pvro[self.counter.n_1,3]-np.roll(self.
      pvro[self.counter.n_1,3],1,axis=2)))
203
204   def calc_pml_region(self):
205     '''PML 3D Algorithm
206     '''
207     alfa = 2*self.sc/self.medium.c0 #this parameter is constant
      in general cases
208
209     #CONSTANT PARAMETERS FOR PML, WHOSE VALUES CAN BE CHANGED
210     a_i = 0.9
211     p = 2
212     gamma = 2e-3
213
214     for j in range(self.dim_region):
```

```
215         #w_i
216         self.w_max = np.sqrt(np.mean(self.pvro[0,1][-self.dim_region
            ,:,:]**2+self.pvro[0,2][:,-self.dim_region,:]**2 + self.pvro
            [0,3][:,:,-self.dim_region]**2))
217         w_i = ((self.dim_region - j - 1/2)/(self.dim_region- 1/2))**
            p * self.w_max
218         self.pvro[0,6][:self.dim_region,:,:] = w_i
219         self.pvro[0,6][:,:self.dim_region,:] = w_i
220         self.pvro[0,6][:,:,:self.dim_region] = w_i
221
222         #f1
223         self.fpml[0,0] = (a_i/self.dt - self.pvro[0,6][:self.
            dim_region,0,0]/2) / (a_i/self.dt + self.pvro[0,6][:self.
            dim_region,0,0]/2)
224         self.fpml[0,1] = (a_i/self.dt - self.pvro[0,6][0,:self.
            dim_region,0]/2) / (a_i/self.dt + self.pvro[0,6][0,:self.
            dim_region,0]/2)
225         self.fpml[0,2] = (a_i/self.dt - self.pvro[0,6][0,0,:self.
            dim_region]/2) / (a_i/self.dt + self.pvro[0,6][0,0,:self.
            dim_region]/2)
226
227         #f2
228         self.fpml[1,0] = 1/ (a_i / self.dt + self.pvro[0,6][:self.
            dim_region,0,0]/2 * self.medium.ro_homogeneous * self.room.
            dres)
229         self.fpml[1,1] = 1/ (a_i / self.dt + self.pvro[0,6][0,:self.
            dim_region,0]/2 * self.medium.ro_homogeneous * self.room.dres
            )
230         self.fpml[1,2] = 1/ (a_i / self.dt + self.pvro[0,6][0,0,:
            self.dim_region]/2 * self.medium.ro_homogeneous * self.room.
            dres)
231
232         #Di
233         D_i = a_i/self.dt + (a_i * gamma * self.medium.c0**2 + w_i)
            /2 + w_i * gamma * self.medium.c0**2 * self.dt/2
234
235         #f3
236         f3 = (a_i/self.dt - (a_i*gamma*self.medium.c0**2 + w_i)/2)/
            D_i
237         self.fpml[2,0],self.fpml[2,1], self.fpml[2,2] = f3,f3,f3
238
239         #f4
240         f4 = -(w_i* gamma * self.medium.c0**2 * self.dt) / D_i
241         self.fpml[3,0], self.fpml[3,1], self.fpml[3,2] = f4, f4, f4
242
243         #f5
244         f5 = -(self.medium.ro_homogeneous * self.medium.c0**2) /
            D_i
245         self.fpml[4,0], self.fpml[4,1], self.fpml[4,2] = f5, f5, f5
```

```
246
247     #f6
248     f6 = 1/D_i
249     self.fpml[5,0], self.fpml[5,1], self.fpml[5,2] = f6, f6, f6
250
251     #algorithm in the boundary
252
253     self.pvro[self.counter.n_1,1][:self.dim_region,:,:] = self.
fpml[0,0]*self.pvro[self.counter.n,1][:self.dim_region,:,:] +
 self.fpml[1,0]*(np.roll(self.pvro[self.counter.n,0][:self.
dim_region,:,:],-1,axis=0)-self.pvro[self.counter.n,0][:self.
dim_region,:,:])
254
255     self.pvro[self.counter.n_1,2][:,:self.dim_region,:] = self.
fpml[0,1]*self.pvro[self.counter.n,2][:,:self.dim_region,:] +
 self.fpml[1,1]*(np.roll(self.pvro[self.counter.n,0][:,:self.
dim_region,:],-1,axis=1)-self.pvro[self.counter.n,0][:,:self.
dim_region,:])
256
257     self.pvro[self.counter.n_1,3][:,:,:self.dim_region] = self.
fpml[0,2]*self.pvro[self.counter.n,3][:,:,:self.dim_region] +
 self.fpml[1,2]*(np.roll(self.pvro[self.counter.n,0][:,:,:
self.dim_region],-1,axis=2)-self.pvro[self.counter.n,0][:,:,:
self.dim_region])
258
259     self.pvro[self.counter.n_1,0][:self.dim_region,:,:] = f3*
self.pvro[self.counter.n,0][:self.dim_region,:,:] + f5*(self.
pvro[self.counter.n_1,1][:self.dim_region,:,:]-np.roll(self.
pvro[self.counter.n_1,1][:self.dim_region,:,:],1,axis=0))
260
261     self.pvro[self.counter.n_1,0][-self.dim_region:,:,:] = f3*
self.pvro[self.counter.n,0][-self.dim_region:,:,:] + f5*(self
.pvro[self.counter.n_1,1][-self.dim_region:,:,:]-np.roll(self
.pvro[self.counter.n_1,1][-self.dim_region:,:,:],1,axis=0))
262
263     self.pvro[self.counter.n_1,0][:,:self.dim_region,:] = f3*
self.pvro[self.counter.n,0][:,:self.dim_region,:] + f5*(self.
pvro[self.counter.n_1,2][:,:self.dim_region,:]-np.roll(self.
pvro[self.counter.n_1,2][:,:self.dim_region,:],1,axis=1))
264
265     self.pvro[self.counter.n_1,0][:,-self.dim_region:,:] = f3*
self.pvro[self.counter.n,0][:,-self.dim_region:,:] + f5*(self
.pvro[self.counter.n_1,2][:,-self.dim_region:,:]-np.roll(self
.pvro[self.counter.n_1,2][:,-self.dim_region:,:],1,axis=1))
266
267     self.pvro[self.counter.n_1,0][:,:,:self.dim_region] = f3*
self.pvro[self.counter.n,0][:,:,:self.dim_region] + f5*(self.
pvro[self.counter.n_1,3][:,:,:self.dim_region]-np.roll(self.
pvro[self.counter.n_1,3][:,:,:self.dim_region],1,axis=2))
```

```
268
269     self.pvro[self.counter.n_1,0][:,:,-self.dim_region:] =    f3*
        self.pvro[self.counter.n,0][:,:,-self.dim_region:] + f5*(self
        .pvro[self.counter.n_1,3][:,:,-self.dim_region:]-np.roll(self
        .pvro[self.counter.n_1,3][:,:,-self.dim_region:],1,axis=2))
270
271     self.calc_sources()
272
273   def calc_sources(self):
274     for i in range(len(self.sources)):
275       self.source_index = i
276       self.pvro[self.counter.n,0,self.sources_coords[i,0],self.
        sources_coords[i,1],self.sources_coords[i,2]] += self.sources
        [i].tone[self.counter.cont]
277
278   def calc_mics(self):
279     for j in range(len(self.micros)):
280       self.microphone_index = j
281       self.micros[j].data[self.counter.cont-1] = self.pvro[self.
        counter.n,0,self.micros_coords[j,0],self.micros_coords[j,1],
        self.micros_coords[j,2]]
282
283   def implement_prop_calc(self):
284     self.calc_sources()
285     self.calc_mics()
286     self.algorithm_prop()
287
288   def save_sim(self):
289     if self.save_status == 1:
290       fl.output('p_'+str(self.counter.cont),self.pvro[self.
        counter.n,0])
291       fl.output('vx_'+str(self.counter.cont),self.pvro[self.
        counter.n,1])
292       fl.output('vy_'+str(self.counter.cont),self.pvro[self.
        counter.n,2])
293       fl.output('vz_'+str(self.counter.cont),self.pvro[self.
        counter.n,3])
294
295   def savedatamic(self):
296     self.exportwav()
297     self.exportdatamic()
298
299   def exportwav(self):
300     print("Exporting data to wav...\n")
301     n=0
302     for i in self.micros:
303       print(i.label+" [mic"+str(n)+"]:")
304       i.save_wav("mic"+str(n),self.t)
305       n+=1
```

```
306
307   def exportdatamic(self):
308     print("Exporting data to numpy binary files")
309     for i in self.micros:
310       fl.record(i.label, i.data)
311
312   def check_status(self):
313     '''Checking aliasing problems that could appear by frequency
      of source or mesh resolution
314     '''
315     self.long_wave = self.medium.c0 / np.array([sources.
    frequency for sources in self.sources])
316     self.dres_min_antialising = self.long_wave/(2 * np.sqrt(3))
317     self.frequency_min_antialising = self.medium.c0 / (2*np.sqrt
    (3)*self.room.dres)
318
319     self.status=SIM_NO_ERR-7
320     if (np.array([sources.frequency for sources in self.sources
    ]) <= self.frequency_min_antialising).any():
321       self.status+=SIM_ERR_FREQ       # SIM_ERR_FREQ
322
323     if (self.room.dres <= self.dres_min_antialising).any():
324       self.status+=SIM_ERR_RES
325
326     if (self.room.dims[0] or self.room.dims[1] or self.room.dims
    [2])  >= 4:
327       self.status+=SIM_ERR_DIMS
328
329   def __str__(self):
330     '''Information of simulation
331     '''
332     cad=''
333
334     cad+= self.room.__str__()
335     cad+= self.medium.__str__()
336
337     for source in self.sources:
338       cad+= source.__str__()
339
340     # Simulation
341     cad+= '+' + '-'*80+'+\n'
342     cad+= '|' + ' '*30 + 'Simulation parameters' + ' '*29 + '|'
    + '\n'
343     cad+= '+' + '-'*80+'+\n'
344     cad+= 'Courant Coefficient: %.2f\n' %(self.sc)
345     cad+= 'Time simulated: %.2f [s]\n'%(self.t)
346     cad+= 'Time step: %.8f [s]\n' %(self.dt)
347     cad+= 'Total steps: %i\n' %(self.t/self.dt)
348     cad+= 'Time of simulation: %.2f [s]\n' %(self.sim_duration)
```

```
349     cad+= '-'*80 + '\n\n'
350     return(cad)
```

C.6. *Counter*

```
#coding:utf-8
'''
TFG Acoustics Simulations

COUNTER CLASS

@author Elias Gabriel Ferrer Jorge
'''

class Counter:
'''
'''
def __init__(self):
self.n = 0
self.n_1 = 1

self.cont = 0

def swap(self):
'''Swap counter values
n_1 -> n
n   -> n_1
'''
self.n+=1
self.n_1+=1

self.n = self.n%2
self.n_1 = self.n_1%2

self.cont +=1

```

C.7. *Filter*

```
#coding:utf-8
"""
TFG Acoustics Simulations

FILTER CLASS
Lowpass filter for implementing impulse response source

@author: Elias Gabriel Ferrer Jorge
"""

import numpy as np
from scipy.signal import butter, lfilter, freqz
from matplotlib import pyplot as plt

class Filter:

  def __init__(self, cutoff, fs,order=5):
    self.cutoff = cutoff
    self.fs = fs
    self.order = order
    self.b = None
    self.a = None
    self.data = None
    self.datafilt = None
    nyq = self.fs / 2
    normal_cutoff = self.cutoff / nyq
    self.b, self.a = butter(self.order, normal_cutoff, btype='
   low', analog=False)


  def filtsource(self,data):
    self.data = data
    nyq = self.fs / 2
    normal_cutoff = self.cutoff / nyq
    self.b, self.a = butter(self.order, normal_cutoff, btype='
   low', analog=False)
    self.datafilt = lfilter(self.b, self.a, self.data)
    return self.datafilt

  def plotfilt(self):
    # Plot the frequency response.
    w, h = freqz(self.b, self.a, worN=8000)
    plt.subplot(2, 1, 1)
    plt.plot(0.5*self.fs*w/np.pi, np.abs(h), 'b')
    plt.plot(self.cutoff, 0.5*np.sqrt(2), 'kx')
    plt.axvline(self.cutoff, color='k')
```

```
45      plt.xlim(0, self.fs/2.)
46      plt.title(r"$Lowpass\ Filter\ Frequency\ Response$")
47      plt.xlabel(r'$Frequency\ [Hz]$')
48      plt.grid()
49
50    def plotfiltdata(self):
51      plt.subplot(2, 1, 2)
52      plt.plot(self.data, 'b-', label='data')
53      plt.plot(self.datafilt, 'g-', linewidth=2, label='filtered
       data')
54      plt.xlabel('Time [sec]')
55      plt.grid()
56      plt.legend()
57
58      plt.subplots_adjust(hspace=0.35)
59      plt.show()
```

Apéndice D

Funciones

D.1. *show.py*

```
#coding:utf-8
'''
TFG Acoustics Simulations

Show 3D data with matplotlib in 2d

@author: Elias Gabriel Ferrer Jorge
'''
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.colors import LogNorm
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.animation as animation
import fdtd.sim as sm
import fdtd.datafiles as fl


def saveanim(data,plottitle,filename):
  print("Loading data...")
  fig, ax = plt.subplots()
  plt.title(str(plottitle))

  ims=[]
  for i in range(len(data)):
    im = ax.imshow(data[i], vmin= np.max(data)/1000, vmax=np.max
    (data), animated=True, origin="lower", interpolation="
    spline36")
    ims.append([im])

  ani = animation.ArtistAnimation(fig, ims, interval=10, blit=
    True, repeat_delay=500)
  print("Exporting to mp4...")
  ani.save(str(filename)+".mp4")

class Show:
  def __init__(self):
    '''
    '''
    self.count_control = 0
    self.data = None

    self.fig  = None
    self.axes = {}
    self.step = 1
    self.mx   = 0
```

```
44      self.my  = 0
45      self.mz  = 0
46
47      self.fps = 0
48
49    def read_data(self, data):
50      '''
51      '''
52      self.data = data
53
54    def layers(self):
55      '''
56      '''
57
58      plt.close()
59
60      self.fig = plt.figure()
61
62      self.ax2 = self.fig.add_subplot(332)
63      self.ax3 = self.fig.add_subplot(333)
64      self.ax4 = self.fig.add_subplot(336)
65      self.ax5 = self.fig.add_subplot(313)
66     #plot 2D
67      self.im2 = self.ax2.imshow(self.data[0,:,:,self.mz], vmin
       =-0.9, vmax = 0.9, origin='lower')
68    self.im3 = self.ax3.imshow(self.data[0,self.mx,:,:], vmin
       =-0.9, vmax = 0.9, origin='lower')
69      self.im4 = self.ax4.imshow(self.data[0,:,self.my,:], vmin
       =-0.9, vmax = 0.9, origin='lower')
70      self.ax2.title.set_text('XY')
71      self.ax3.title.set_text('YZ')
72      self.ax4.title.set_text('XZ')
73
74      self.count_control+=1
75
76    def sources(self):
77      '''
78      '''
79
80      self.fig = plt.figure()
81
82      for i in range(len(sm.sources.source_label)):
83        self.axes
84
85
86
87    def set_mx(self,mx = 0):
88      '''
89      '''
```

```
90      self.mx = mx
91
92    def set_my(self,my = 0):
93      '''
94      '''
95      self.my = my
96
97    def set_mz(self,mz = 0):
98      '''
99      '''
100     self.mz = mz
101
102
103   def __call__(self):
104     '''
105     '''
106     self.ax2.set_title("XY:%f"%self.step)
107     #plot 2D
108     self.im3.set_data(self.data[0,:,:,self.mz])
109     self.im4.set_data(self.data[0,self.mx,:,:])
110     self.fig.canvas.draw()
```

D.2. *datafiles.py*

```
#coding:utf-8
'''
TFG Acoustics Simulations

Writer of output file of data of simulation

@author: Elias Gabriel Ferrer Jorge
'''
import numpy as np
import sys
import os

import fdtd.sim as si


def output(filename,data, folder='sim_data'):
  '''
  '''
  print("Saving " + str(filename) + " in " + str(folder))
  np.save(os.path.join('test_fdtd/data/'+str(folder), str(
    filename)),data)

def input(filename, folder = 'sim_data'):
  '''
  '''
  return np.load(os.path.join('test_fdtd/data/'+str(folder), str
    (filename) + '.npy'))

def record(microphone_label, data, folder='micro_data'):
  '''
  '''
  np.save(os.path.join('test_fdtd/data/'+str(folder), str(
    microphone_label)),data)

def save_number(filename,data):
  f = open(str(filename+'.txt'),'wt')
  f.write(str(data))
  f.close()

def read_number(filename):
  f = open(str(filename),'r')
  number = int(f.read())
  f.close()
  return number
```

D.3. *wav.py*

```
#coding:utf-8
'''
TFG Acoustics Simulations

Export to wav file array data from microphone

@author: Elias Gabriel Ferrer Jorge
'''
import numpy as np
from scipy.io.wavfile import write

def towav(name,t,data):
  if len(data)==0:
    raise ValueError("Data length 0")

  if np.max(data)==0:
    samples_s = int(len(data))
    print("No sound recorded")

  else:
    samples_s = int(len(data)/t)
    data = data/np.max(data)

  waveform_integers = np.int16(data * 32767)
  return write("data/micro_data/data_wav/"+str(name)+".wav",
   samples_s, waveform_integers)
```

D.4. *acoustic_param.py*

```
#coding:utf-8
'''
TFG Acoustics Simulations

Acoustic parameters calculation tool
https://newt.phys.unsw.edu.au/jw/z.html
https://www.animations.physics.unsw.edu.au/jw/sound-impedance-
    intensity.htm

@author Elias Gabriel Ferrer Jorge
'''
import numpy as np

def v(vx,vy,vz):
  '''Calculate v = sqrt(vx^2 + vy^2 + vz^2)
  '''
  v = np.sqrt(vx*vx+vy*vy+vz*vz)
  return v

def U(type='spherical', area = 0, vx = 0, vy = 0, vz = 0):
  '''Consider a  wave passing through an surface with an area
  then we have a flux U through this surface.
    type of surface can be set as:
    type = 'spherical'
    type = 'cylindrical'
    for example
  '''
# TODO: definir el vector normal para realizar el
# producto escalar por el vector velocidad
  proy_normal_vel = 1
  U = area * proy_normal_vel
  U = A * (vx + vy + vz)

  print('flux through '+type +' surface')

  return U

def intensity(p, rho, c):
  '''Sound intensity
  INPUT
    p
    rho
    c

  OUTPUT
    I : Sound intensity
```

```
46
47      ****************************
48      I = p^2/(2Z) [W/m^2]
49
50      p: Sound pressure [Pa]
51      Z: Sound impedance -> Z = rho * c
52      rho = medium density [Kg/m^3]
53      c   = sound velocity [m/s]
54      '''
55
56
57    I = p**2/(rho*c)
58
59    return I
60
61  def inst_intensity(p,v):
62    '''Instantaneous Intensity
63      I_inst = p * v
64      where v = sqrt(vx^2 + vy^2 + vz^2)
65    '''
66
67    I_inst = p * v
68
69    return I_inst
70
71  def lvl_sound_pressure(P1, Pref= 20 * 10**-6):
72    '''
73    INPUT
74      data -> P1 [Pa]
75    OUTPUT
76      L_P : level of sound Pressure [dB]
77
78    ****************************
79      L_P = 20 * log10(P1/P2) [dB]
80
81      P1 : Sound pressure acquired
82      P2 : Sound pressure of reference -> P2 = 20 [micro Pa]
83    '''
84
85    L_P = 20 * np.log10(P1/Pref)
86
87    return L_P
88
89  def impedance_z_sp(p,v):
90    '''Specific impedance z
91      z_sp = p/v
92    '''
93    z_sp = p/v
94
```

```
95    return z_sp
96
97  def impedance_z(p,U):
98    '''z is the ratio of acoustic pressure p to acoustic volume
       flow U
99    z = p/U
100   '''
101   z = p/U
102
103   return z
104
105 def data_conv_impulse(impulse_response,data):
106   '''Convolution of data with impulse response function
107   '''
108
109 """
110 Reference GITHUB
111 https://github.com/python-acoustics/python-acoustics/blob/master
       /acoustics/room.py
112
113 Room
114 ====
115 The room acoustics module contains several functions to
       calculate the reverberation time in spaces.
116 """
117 import numpy as np
118
119 from scipy.io import wavfile
120 from scipy import stats
121
122 #from acoustics.utils import _is_1d
123 #from acoustics.signal import bandpass
124 #from acoustics.bands import (_check_band_type, octave_low,
       octave_high, third_low, third_high)
125
126 #SOUNDSPEED =
127
128
129 def mean_alpha(alphas, surfaces):
130   """
131   Calculate mean of absorption coefficients.
132   :param alphas: Absorption coefficients :math:`\\alpha`.
133   :param surfaces: Surfaces :math:`S`.
134   """
135   return np.average(alphas, axis=0, weights=surfaces)
136
137
138 def nrc(alphas):
139   """
```

```
140     Calculate Noise Reduction Coefficient (NRC) from four
         absorption
141     coefficient values (250, 500, 1000 and 2000 Hz).
142     :param alphas: Absorption coefficients :math:`\\alpha`.
143     """
144     alpha_axis = alphas.ndim - 1
145     return np.mean(alphas, axis=alpha_axis)
146
147
148     def t60_sabine(surfaces, alpha, volume, c):
149     """
150     Reverberation time according to Sabine.
151     :param surfaces: Surface of the room :math:`S`.
152       NumPy array that contains different surfaces.
153     :type surfaces: :class:`np.ndarray`
154     :param alpha: Absorption coefficient of the room :math:`\\
         alpha`.
155       Contains absorption coefficients of ``surfaces``.
156       It could be one value or some values in different bands (1D
         and 2D
157       array, respectively).
158     :type alpha: :class:`np.ndarray`
159     :param volume: Volume of the room :math:`V`.
160     :type volume: :class:`float`
161     :param c: Speed of sound :math:`c`.
162     :type c: :class:`float`
163     :returns: Reverberation time :math:`T_{60}`
164     Sabine's formula for the reverberation time is:
165     .. math:: T_{60} = \\frac{24 \\ln(10)}{c} \\frac{V}{S\\alpha}
166     """
167     mean_alpha_ = np.average(alpha, axis=0, weights=surfaces)
168     S = np.sum(surfaces, axis=0)
169     A = S * mean_alpha_
170     t60 = 4.0 * np.log(10.0**6.0) * volume / (c * A)
171     return t60
172
173
174     def t60_eyring(surfaces, alpha, volume, c):
175     """
176     Reverberation time according to Eyring.
177     :param surfaces: Surfaces :math:`S`.
178     :param alpha: Mean absorption coefficient :math:`\\alpha` or
         by frequency bands
179     :param volume: Volume of the room :math:`V`.
180     :param c: Speed of sound :math:`c`.
181     :returns: Reverberation time :math:`T_{60}`
182     Eyring's formula for the reverberation time is:
183     .. math:: T_{60} = \\frac{24 \\ln{10} V}{c \\left( 4 mV - S \\
         ln{\\left( 1 - \\alpha \\right)} \\right)}
```

```
184     """
185     mean_alpha_ = np.average(alpha, axis=0, weights=surfaces)
186     S = np.sum(surfaces, axis=0)
187     A = -S * np.log(1 - mean_alpha_)
188     t60 = 4.0 * np.log(10.0**6.0) * volume / (c * A)
189     return t60
190
191
192     def t60_millington(surfaces, alpha, volume, c):
193     """
194     Reverberation time according to Millington.
195     :param surfaces: Surfaces :math:`S`.
196     :param alpha: Mean absorption coefficient :math:`\\alpha` or
          by frequency bands
197     :param volume: Volume of the room :math:`V`.
198     :param c: Speed of sound :math:`c`.
199     :returns: Reverberation time :math:`T_{60}`
200     """
201     mean_alpha_ = np.average(alpha, axis=0, weights=surfaces)
202     A = -np.sum(surfaces[:, np.newaxis] * np.log(1.0 - mean_alpha_
          ), axis=0)
203     t60 = 4.0 * np.log(10.0**6.0) * volume / (c * A)
204     return t60
205
206
207     def t60_fitzroy(surfaces, alpha, volume, c):
208     """
209     Reverberation time according to Fitzroy.
210     :param surfaces: Surfaces :math:`S`.
211     :param alpha: Mean absorption coefficient :math:`\\alpha` or
          by frequency bands
212     :param volume: Volume of the room :math:`V`.
213     :param c: Speed of sound :math:`c`.
214     :returns: Reverberation time :math:`T_{60}`
215     """
216     Sx = np.sum(surfaces[0:2])
217     Sy = np.sum(surfaces[2:4])
218     Sz = np.sum(surfaces[4:6])
219     St = np.sum(surfaces)
220     alpha = _is_1d(alpha)
221     a_x = np.average(alpha[:, 0:2], weights=surfaces[0:2], axis=1)
222     a_y = np.average(alpha[:, 2:4], weights=surfaces[2:4], axis=1)
223     a_z = np.average(alpha[:, 4:6], weights=surfaces[4:6], axis=1)
224     factor = -(Sx / np.log(1.0 - a_x) + Sy / np.log(1.0 - a_y) +
          Sz / np.log(1 - a_z))
225     t60 = 4.0 * np.log(10.0**6.0) * volume * factor / (c * St
          **2.0)
226     return t60
227
```

```
228
229   def t60_arau(Sx, Sy, Sz, alpha, volume, c):
230   """
231   Reverberation time according to Arau. [#arau]_
232   :param Sx: Total surface perpendicular to x-axis (yz-plane) :
       math:`S_{x}`.
233   :param Sy: Total surface perpendicular to y-axis (xz-plane) :
       math:`S_{y}`.
234   :param Sz: Total surface perpendicular to z-axis (xy-plane) :
       math:`S_{z}`.
235   :param alpha: Absorption coefficients :math:`\\mathbf{\\alpha}
       = \\left[ \\alpha_x, \\alpha_y, \\alpha_z \\right]`
236   :param volume: Volume of the room :math:`V`.
237   :param c: Speed of sound :math:`c`.
238   :returns: Reverberation time :math:`T_{60}`
239   .. [#arau] For more details, please see
240   http://www.arauacustica.com/files/publicaciones/pdf_esp_7.pdf
241   """
242   a_x = -np.log(1 - alpha[0])
243   a_y = -np.log(1 - alpha[1])
244   a_z = -np.log(1 - alpha[2])
245   St = np.sum(np.array([Sx, Sy, Sz]))
246   A = St * a_x**(Sx / St) * a_y**(Sy / St) * a_z**(Sz / St)
247   t60 = 4.0 * np.log(10.0**6.0) * volume / (c * A)
248   return t60
249
250
251 def t60_impulse(file_name, bands, rt='t30'):  # pylint: disable=
       too-many-locals
252   """
253   Reverberation time from a WAV impulse response.
254   :param file_name: name of the WAV file containing the impulse
       response.
255   :param bands: Octave or third bands as NumPy array.
256   :param rt: Reverberation time estimator. It accepts ''t30'',
       ''t20'', ''t10'' and ''edt''.
257   :returns: Reverberation time :math:`T_{60}`
258   """
259   fs, raw_signal = wavfile.read(file_name)
260   band_type = _check_band_type(bands)
261
262   if band_type == 'octave':
263     low = octave_low(bands[0], bands[-1])
264     high = octave_high(bands[0], bands[-1])
265   elif band_type == 'third':
266     low = third_low(bands[0], bands[-1])
267     high = third_high(bands[0], bands[-1])
268
269   rt = rt.lower()
```

```
270   if rt == 't30':
271     init = -5.0
272     end = -35.0
273     factor = 2.0
274   elif rt == 't20':
275     init = -5.0
276     end = -25.0
277     factor = 3.0
278   elif rt == 't10':
279     init = -5.0
280     end = -15.0
281     factor = 6.0
282   elif rt == 'edt':
283     init = 0.0
284     end = -10.0
285     factor = 6.0
286 
287   t60 = np.zeros(bands.size)
288 
289   for band in range(bands.size):
290     # Filtering signal
291     filtered_signal = bandpass(raw_signal, low[band], high[band
      ], fs, order=8)
292     abs_signal = np.abs(filtered_signal) / np.max(np.abs(
      filtered_signal))
293 
294     # Schroeder integration
295     sch = np.cumsum(abs_signal[::-1]**2)[::-1]
296     sch_db = 10.0 * np.log10(sch / np.max(sch))
297 
298     # Linear regression
299     sch_init = sch_db[np.abs(sch_db - init).argmin()]
300     sch_end = sch_db[np.abs(sch_db - end).argmin()]
301     init_sample = np.where(sch_db == sch_init)[0][0]
302     end_sample = np.where(sch_db == sch_end)[0][0]
303     x = np.arange(init_sample, end_sample + 1) / fs
304     y = sch_db[init_sample:end_sample + 1]
305     slope, intercept = stats.linregress(x, y)[0:2]
306 
307     # Reverberation time (T30, T20, T10 or EDT)
308     db_regress_init = (init - intercept) / slope
309     db_regress_end = (end - intercept) / slope
310     t60[band] = factor * (db_regress_end - db_regress_init)
311   return t60
312 
313 
314 def clarity(time, signal, fs, bands=None):
315   """
316   Clarity :math:`C_i` determined from an impulse response.
```

```
317   :param time: Time in miliseconds (e.g.: 50, 80).
318   :param signal: Impulse response.
319   :type signal: :class:`np.ndarray`
320   :param fs: Sample frequency.
321   :param bands: Bands of calculation (optional). Only support
       standard octave and third-octave bands.
322   :type bands: :class:`np.ndarray`
323   """
324   band_type = _check_band_type(bands)
325
326   if band_type == 'octave':
327     low = octave_low(bands[0], bands[-1])
328     high = octave_high(bands[0], bands[-1])
329   elif band_type == 'third':
330     low = third_low(bands[0], bands[-1])
331     high = third_high(bands[0], bands[-1])
332
333   c = np.zeros(bands.size)
334   for band in range(bands.size):
335     filtered_signal = bandpass(signal, low[band], high[band], fs
      , order=8)
336     h2 = filtered_signal**2.0
337     t = int((time / 1000.0) * fs + 1)
338     c[band] = 10.0 * np.log10((np.sum(h2[:t]) / np.sum(h2[t:])))
339   return c
340
341
342 def c50_from_file(file_name, bands=None):
343   """
344   Clarity for 50 miliseconds :math:`C_{50}` from a file.
345   :param file_name: File name (only WAV is supported).
346   :type file_name: :class:`str`
347   :param bands: Bands of calculation (optional). Only support
       standard octave and third-octave bands.
348   :type bands: :class:`np.ndarray`
349   """
350   fs, signal = wavfile.read(file_name)
351   return clarity(50.0, signal, fs, bands)
352
353
354 def c80_from_file(file_name, bands=None):
355   """
356   Clarity for 80 miliseconds :math:`C_{80}` from a file.
357   :param file_name: File name (only WAV is supported).
358   :type file_name: :class:`str`
359   :param bands: Bands of calculation (optional). Only support
       standard octave and third-octave bands.
360   :type bands: :class:`np.ndarray`
361   """
```

```
362     fs, signal = wavfile.read(file_name)
363     return clarity(80.0, signal, fs, bands)
```

D.5. *write_vert_fac.py*

Script de ejecución en consola de Blender 2.7.x en la configuración en español.

```
#coding:utf-8
'''
TFG Acoustics Simulations

Obtain vert and normals from mesh data with Blender 2.7.x in
    Spanish Language

@author: Elias Gabriel Ferrer Jorge
'''
import bpy

#obj = bpy.data.objects["Cube"]  # Choosing object by nametag
obj = bpy.context.view_layer.objects.active # Choosing object by
     active in viewport

mesh = obj.data

f=open('Vertex.txt','w')
f.write("# Vertex Total = %d\n" % len(mesh.vertices))
i=0
for vert in mesh.vertices:
  i+=1
  f.write('v%d' % (i))
  f.write( '\n%f %f %f\n' % (vert.co.x, vert.co.y, vert.co.z) )
f.close()



f=open('Faces.txt','w')

f.write("# Face Total = %d" % len(mesh.polygons))

i=0
for face in mesh.polygons:
  i+=1
  f.write('\nface%d' % (i))
  for vert in face.vertices:
  f.write('\n%d'%(vert))
f.close()
```

Apéndice E

Ejemplos de ejecución

E.1. valar.py

Programa de ejemplo desarrollado a partir de la API para simular la validación del software.

```
# coding:utf-8
'''
TFG Acoustics Simulations

Main program launcher Example

@author: Elias Gabriel Ferrer Jorge
'''

#import fdtd.room        as ro
#import fdtd.medium      as me
#import fdtd.source      as sr
#import fdtd.sim         as si
#import fdtd.filter      as fil
#import fdtd.datafiles   as fl
#import fdtd.waw         as w
#import fdtd.counter     as count
#import fdtd.micophone   as mc


# Load modules including new changes writen in src pathfile

modules = [['fdtd.room'],\
['fdtd.medium'],
['fdtd.source'],
['fdtd.sim'],
['fdtd.filter'],
['fdtd.datafiles'],
['fdtd.wav'],
['fdtd.counter'],
['fdtd.microphone']]

for i in modules:
  if i[0] in sys.modules:
    i[0] = importlib.reload(sys.modules[i[0]])
  else:
    i[0] = importlib.import_module(i[0])

ro     = modules[0][0]
me     = modules[1][0]
sr     = modules[2][0]
si     = modules[3][0]
```

```
fil    = modules[4][0]
fl     = modules[5][0]
w      = modules[6][0]
count  = modules[7][0]
mc     = modules[8][0]


f = open('log.txt','wt')

# TEST
#-----------------------------------------
# PARAMETERS
#-----------------------------------------

# ROOM
dims = [2.2,2.2,2.2] #[m x m x m]
dres = 0.1           #[m]
pini = [0,0,0]       #[m]


# MEDIUM
# medium_name = 'An awesome medium'
# c0  = 1.  #[m/s]
# cr0 = 1.  # 0 < cr0 <= 1
# r0h = 1.  #[kg/m^3]

# SOURCE
source_label = 'Secondary Source'
sample_rate  = 44100   #Sample rate  [samples/s]
t_0          = 0.      #Initial time [s]
duration_src = 2.      #Duration of Source's tone [s]
frec_src     = 200.0    # Frequency of Source [Hertz]
phase_src    = 0.     # Phase angle of Source's tone [rad]
coords_src   = [1.1,1.1,1.1] # Coordinates of position [m]


# SIMULATION
sim_type  = "pml" #PML algorithm
# sim_type = "nopml" #Perfect reflection in boundaries

print("Simulation Type: "+ sim_type + '\n\n')

sim_time = 5. #[s]


#-----------------------------------------
# INSTANCES
#-----------------------------------------

```

```
92 # ROOM
93 rom  = ro.Room(dims,dres,pini)
94
95 # MEDIUM
96 med = me.Medium(rom)
97
98
99 # SOURCES
100 src1 = sr.Source() #Create Main Source
101 src2 = sr.Source(label = "Secondary Source", amplitude= 1e-3, t0
        = t_0, frequency=frec_src, duration=duration_src) # Create
    Secondary Source
102
103 # List of Sources
104 sources = [src2]
105
106
107 # MICROPHONES
108 mic0 = mc.Microphone() #Default Microphone
109 mic1 = mc.Microphone(coords=[0.5,0.5,0.5])
110 mic2 = mc.Microphone(coords=[0.1,0.5,0.5])
111 mic3 = mc.Microphone(coords=[0.2,0.5,0.5])
112 mic4 = mc.Microphone(coords=[0.3,0.5,0.5])
113 mic5 = mc.Microphone(coords=[0.4,0.5,0.5])
114
115 #List of Microphones
116 mics = [mic0\
117      ,mic1
118      ,mic2
119      ,mic3
120      ,mic4
121      ,mic5
122      ]
123
124
125 # SIMULATION
126 sim  = si.Sim(sim_type,rom,med,sources,mics,sim_time)
127
128
129 # FILTER
130 #Creation of Lowpassfilter
131 freq_filtering = 1/sim.dt/2.1*dres*10 #Max frecuency [Hz]
    without aliasing effects
132
133 fl1 = fil.Filter(freq_filtering,src1.sample_rate) #Filter
    instance for src1
134 fl2 = fil.Filter(freq_filtering,src2.sample_rate) #Filter
    instance for src2
135
```

```
136 #Filtering sources
137 src1.tone = fl1.filtsource(src1.tone)
138 src2.tone = fl2.filtsource(src2.tone)
139
140
141 #Implement sources in simulation environment
142 sim.init_sources(sources)
143
144
145 #Check status of simulation
146 if (sim.status==sim.SIM_NO_ERR):
147   f.write("Simulation OK\n")
148   print("Simulation OK")
149   f.write(sim.__str__())
150   f.close()
151   fl.save_number('steps',int(sim.t/sim.dt))
152   print(sim)
153
154 else:
155   f.write('Simulation impossible due to aliasing!!!!\n' + sim.
      __str__() + '\n' + 'ABORTED\n')
156   f.close()
157   raise ValueError('Simulation impossible due to aliasing \
      nSource Frequency:%s -> Max Frequency allowed: %s \n' %(sim.
      sources[sim.source_index].frequency,sim.
      frequency_min_antialising))
158
159 #Execute simulation
160 def start_sim():
161   sim.run_sim()
162   sim.exportwav()
163   sim.exportdatamic()
```