Universidad de Valladolid

ESCUELA DE INGENIERÍAS INDUSTRIALES

GRADO EN INGENIERÍA MECÁNICA

---

# Desarrollo de un software libre para procesar y analizar ensayos modales experimentales.

---

**Autor:**
**D. Jesús Sáenz Niño**

**Tutores:**
**D. Antolín Lorenzana Ibán**
**D. Álvaro Magdaleno González**

# Resumen

Se ha escrito una librería de python para el tratamiento de señales provenientes de un análisis modal experimental, creando una aplicación web y varios programas de ejemplo para extender la funcionalidad de la misma.
La librería se enfoca a un alto nivel, con programación orientada a objetos, indicada para usuarios con bajo conocimiento de programación. Permite eliminar tendencias lineales de la señal, incluye un filtro tipo butterworth IIR y un algoritmo para modificar la frecuencia de muestreo, gracias a la librería scipy. Las funciones de respuesta en frecuencia se calculan con algoritmos de la misma librería, mientras que las funciones de detección de modos, peak picking y circle fit, se calculan con los métodos tradicionales. Los ejemplos incluidos y la aplicación web proporcionan la interfaz gráfica y algoritmos para seleccionar los parámetros de la función de respuesta en frecuencia.

**Palabras clave**

Análisis modal experimental
circle fit
peak picking
python

# Abstract

A python library is made for signal treatment on experimental modal analysis, aimed towards impact hammer tests. A web application and several examples are programmed to extend the functionality of the library.
The library is developed for a very high level use, with an object oriented philosophy, delivered for users with low programming background. It has moderate capabilities for detrending the signal with a linear function, it includes a butterworth IIR filter and a resampling algorithm from scipy scientific library. The frequency response functions are also calculated with scipy library. The mode detection algorithms, peak picking and circle fit, are calculated with traditional methods. Included examples and the web app deliver a user interface and algorithms for selecting the frequency response function parameters.

**Keywords**

Experimental modal analysis
circle fit
peak picking
python

# Índice general

# Índice de figuras

# Índice de cuadros

# Capítulo 1

# Introducción y objetivos.

## 1.1. Introducción

Las vibraciones es una gran rama de estudio de la ingeniería con creciente interés desde los años 80. Durante su desarrollo se está incorporando de forma obligatoria en la mayor parte de los campos del diseño mecánico. En el caso de las estructuras, la necesidad del estudio vibratorio surgió por el interés en hacer estructuras con menor cantidad de material resistente y el uso de elementos más esbeltos, de forma que las cargas estáticas eran soportadas, pero las cargas dinámicas causaban mayores desplazamientos en las estructuras, que eran detectados por los usuarios de las estructuras. Precisamente han surgido en el ámbito de las estructuras dos ejemplos famosos por su errores de diseño frente a las vibraciones: el puente de Tacoma Narrows y el puente del milenio de Londres (figura 1.1).

Para el desarrollo de este campo el papel de la informática ha sido fundamental. En primer lugar, mediante métodos de elementos finitos permite realizar un análisis vibratorio sin tener que hacer una maqueta de la estructura. En segundo lugar, ha desarrollado las herramientas de medida y ha permitido que el propio usuario programe algoritmos para analizar los resultados. Si bien es verdad que existen herramientas comerciales para realizar el tratamiento de datos, éstas no permiten modificar libremente los algoritmos para el tratamiento de datos.

En este contexto, los lenguajes de programación enfocados al análisis de datos

Figura 1.1: Puentes de Tacoma y Londres

(a) Tacoma Narrows bridge[1]

(b) Millenium Bridge[2]

científicos incluyen librerías específicas para el tratamiento de ondas. Éstas herramientas proporcionan al usuario una sólida base para poder hacer un tratamiento básico de datos. En el ámbito de la investigación éstas herramientas tienen un gran interés, pues permiten realizar, compartir y modificar programas de forma sencilla.

Para el desarrollo de este trabajo se ha elegido usar Python. Ésta elección responde a varias ventajas y necesidades. En primer lugar, es un lenguaje intuitivo y sencillo de aprender. En segundo lugar, contiene una gran cantidad de librerías y módulos para el análisis de datos, desde funciones y algoritmos matemáticos hasta librerías para la visualización de datos. Todos estos recursos usados son de código libre, por lo que se puede acceder y modificar el código en caso de que se desee aumentar o personalizar la funcionalidad. En tercer lugar, es un lenguaje muy usado para el tratamiento y análisis de datos con una comunidad muy activa en internet. Finalmente, es un lenguaje multipropósito, se puede integrar cómodamente en aplicaciones web y de escritorio.

## 1.2. Planteamiento del problema y objetivos

En la introducción se han indicado las limitaciones de los programas comerciales para el análisis de vibraciones: su poca versatilidad de cara a la investigación y al uso de nuevos algoritmos. La motivación para hacer este trabajo es obtener una herramienta de software libre que no tenga limitaciones para funcionar en ningún equipo, de forma que se pueda ejecutar en cualquier máquina sin necesidad de licencias u restricciones para modificar los módulos o librerías que use.

Por otra parte, se centrará toda la atención en los test de impacto de martillo, ya que es el uno de los principales métodos de medida en el análisis modal, y no se tendrá en cuenta otros métodos experimentales de análisis modal, aunque se considerarán algunas opciones para que se pueda aplicar a un ámbito más general.

La finalidad de éste trabajo es, por tanto, crear un programa que permita analizar los datos de una prueba de martillo y obtener la frecuencia y el factor de amortiguamiento de los modos de vibración, todo ello realizado en python con el uso de todas las librerías y paquetes de código libre que sean necesarios.

A partir de la definición del problema y la finalidad del trabajo de fin de grado, podemos definir los objetivos que se perseguirán durante el desarrollo del mismo:

- Realizar un programa capaz de analizar las señales temporales, calculando los estadísticos principales y realizando un preprocesamiento de las mismas.
- Definir funciones robustas para calcular las funciones de respuesta en frecuencia y calcular las frecuencias propias y factores de amortiguamiento.
- Obtener una estructura sólida para poder aumentar la funcionalidad del programa.
- Obtener una precisión en el tratamiento de datos equiparable a programas comerciales.

Durante el transcurso del trabajo se intentará obtener una solución que responda a los objetivos anteriores que sea fácil de utilizar para el usuario final, sacrificando la

libertad de opciones para cada función. Esta idea se aleja del camino para desarrollar una librería abierta que se pueda adaptar a cualquier persona, como numpy y scipy, pero abre una vía para el uso de este lenguaje de programación en la enseñanza y la investigación.

Por último, el desarrollo del programa ha requerido de conocimientos avanzados en el ámbito de la informática: funcionamiento básico de servidores, bases de datos, programación orientada a objetos... En la memoria estos temas se abordan desde un punto de vista básico, ya que no son propios del grado de ingeniería mecánica. De la misma forma, el código del programa se ha reescrito de una forma más entendible, renombrando las variables y modificando el código para que los algoritmos sean entendidos. En el código original, las funciones incluyen numerosos objetos y funciones de soporte para encontrar errores que dificultan la legibilidad del mismo.

# Capítulo 2

# Fundamentos teóricos.

## 2.1. Introducción

El objeto de este capítulo es explicar y comprender los conocimientos y el trabajo previo de investigación que se ha realizado sobre la materia. El campo de las vibraciones mecánicas ha necesitado un complejo desarrollo matemático para su estudio.

En este capítulo se ignorarán los modelos vibratorios matemáticos sobre los diferentes tipos de sólidos y mecanismos: tanto los modelos discretos de masa-muelle y amortiguador como los medios continuos de viga y placa. Si bien sientan las bases del análisis vibratorio y están estrechamente relacionados con los métodos matemáticos usados en éste trabajo, el enfoque del programa será sobre el análisis de los datos medidos y no de las propiedades del sólido.

## 2.2. Test de martillo de impacto

El test de martillo es un método de análisis modal experimental versátil y portable. Es por lo tanto usado para piezas pequeñas como grandes estructuras. El desarrollo del ensayo es el siguiente:

1. Colocación de los acelerómetros y la cédula de carga en la estructura: se posicionan los acelerómetros en puntos donde se sospeche que puede haber frecuencias propias. Deben estar repartidos en toda la estructura. La cédula de carga se posicionará en el punto en el que se vaya a golpear la estructura.

2. Comprobación de que las condiciones son ideales para la toma de datos. Se debe asegurar que no hay elementos vibrando, o sonidos, durante la toma de datos. Dependiendo del tamaño de la estructura influirán en mayor o menor medida, pero repercutirán de forma muy negativa en las señales y en el proceso de análisis de datos.

3. Inicio de la toma de datos: puede iniciarse de forma automática al golpear la estructura, pero de ser posible, se debe iniciar la toma de datos antes del impacto.

4. Impacto y toma de datos: se golpea a la estructura, de forma controlada, en la cédula de carga. Las vibración de la estructura se disipará al cabo de un minuto, hasta entonces la sala debe permanecer en silencio.

5. Cese de la toma de datos y comprobación de la coherencia de la señal: tras la primera muestra, se comprueba que los datos son correctos y que no hay una gran cantidad de ruido.

6. Repetición del proceso: la toma de muestras se realiza por lo menos 3 veces por cada punto de impacto. Dependiendo de la intenciones del estudio, se pueden tomar medidas de diferentes puntos de impacto.

### 2.2.1. Parámetros y observaciones

## 2.3. Estadísticos

Los estadísticos son parámetros obtenidos a partir del conjunto de valores de cada acelerómetro o célula de carga. Su objetivo es estimar ciertas características de la vibración. Son valores que se pueden obtener rápidamente y caracterizan parámetros relacinados con la intensidad de la vibración producida. Tienen un gran interés fuera del análisis modal, en experimentos con cargas dinámicas reales.

### 2.3.1. RMS

Se denomina R.M.S. (*Root Mean Square*) a la raíz cuadrada de la media de los cuadrados de cada elemento de un vector.

$$R.M.S. = \sqrt{\frac{1}{i}\sum_{i=0} x_i^2} \tag{2.1}$$

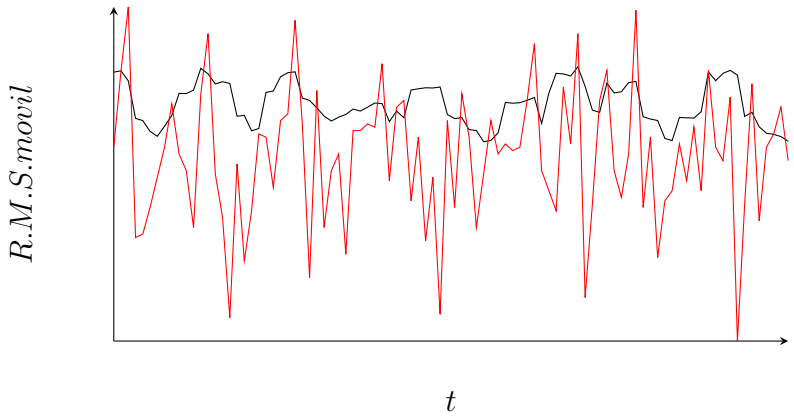
Si se aplica a un vector el resultado será un escalar, con las mismas unidades que presenta el vector original. En el caso específico de una onda, se entiende el R.M.S. como la raíz cuadrada de la media integral del cuadrado de la función onda en función del tiempo.

$$f_{RMS} = \sqrt{\frac{1}{T_2 - T_1}\int_{T_1}^{T_2} f(t)^2\, dt} \tag{2.2}$$

La utilidad de este estadísitico es, en gran parte, como comparativa entre la aceleración de dos señales. Además, para aumentar el interés del RMS, se usa el RMS móvil.

El RMS móvil (figura 2.1) aplica la ecuación (2.2) a sucesivas ventanas de tiempo, cuya longitud se denomina anchura de la ventana. El resultado es una función que representa la intensidad de vibración en cada instante de tiempo a partir del impacto inicial. El programa deja a elección del usuario el tiempo que comprende la ventana. Otro parámetro a considerar cuando se introduce el concepto de ventanas es el solapamiento entre ellas. Para este caso tendremos fijo un avance de un término a la vez. Para una ventana de $n$ términos, deberemos tener un solapamiento de $n-1$ términos. En el caso del RMS móvil, se suele fijar el tamaño de la ventana en 1 segundo, según las normas [3].

Figura 2.1: Gráfica del RMS móvil frente a la señal original en rojo.




### 2.3.2. MTVV

El MTVV (*Maximum Transient Vibration Value* [3]) se define como el mayor valor del RMS durante el período de medida. Para ello, partimos del RMS móvil con una ventana de tiempo determinada, y se halla el máximo valor de la función $RMS(t)$.

Se puede usar cualquier ventana en el RMS móvil para hallar el MTVV, sin embargo es recomendable usar una ventana de un segundo. El solape o el avance pueden ser elegidos por el usuario, simplemente modifican la resolución del resultado.

### 2.3.3. VDV

El estadístico VDV (*Vibration Dose Value*[3]) identifica desde un punto de vista humano la dosis de vibración que se recibe en un determinado intervalo de tiempo, por lo que es más adecuado para las normas y códigos técnicos que otros estadísticos. Matemáticamente se define como la raíz cuarta del sumatorio del cuadrado de valores de aceleración.

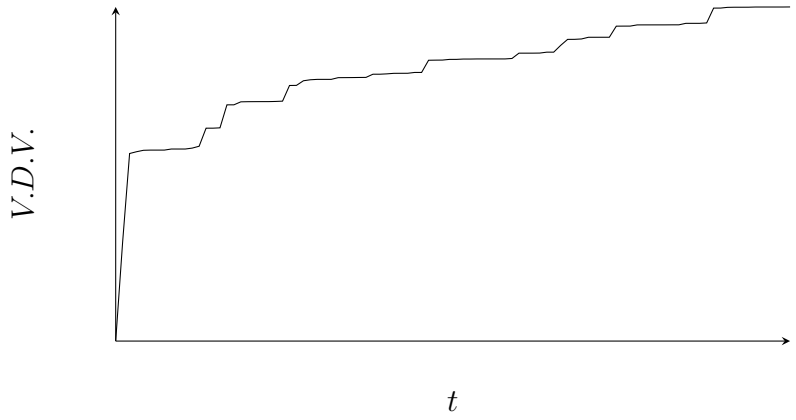
Este estadístico normalmente se toma en señales de gran duración. En todo caso, la idea de que se aproxima a la sensación de un humano viene dada porque está elevado a la cuarta potencia, lo que hace la función más sensible a señales de gran amplitud.

Es interesante graficar este estadístico como el acumulado de su valor(figura 2.2). Para ello, en el eje de abscisas se representa el tiempo y en el eje de ordenadas se representa el acumulado del VDV desde el inicio de la señal hasta el tiempo $t$.

## 2.4. Tratamiento de señal

El tratamiento de señal se refiere a todas las transformaciones, aplicadas sobre los datos temporales obtenidos directamente sobre la medición, para facilitar el trabajo del algoritmo que genera las funciones de respuesta en frecuencia. Las transformaciones realizadas están indicadas para mejorar el resultado de las señales de respuesta en frecuencia, o para mejorar el resultado de los estadísticos.

Figura 2.2: Gráfica del estadístico V.D.V. para una señal breve.




### 2.4.1. Filtrado

Filtrar es la operación de eliminar frecuencias indeseadas en la señal (figura 2.3). En los test de impacto de martillo para estructuras se suele tener ruido, generalmente a altas frecuencias. Para obtener unos valores correctos en los estadísticos se necesita eliminar estas altas frecuencias, por lo que se acude a un filtro digital *butterworth*, aplicado con *IIR*.

*Butterworth* se refiere a un tipo específico de filtro digital, determina la función de transferencia que se aplicará después, y se puede indicar el orden de los polinomios y la frecuencia de corte.

Se aplican con un algoritmo de IIR (*infinite impulse response*) en vez de usar un FIR (*finite impulse response*). Mientras que el IIR se compone de un elemento con un canal de retroalimentación, el sistema FIR se compone de un número finito de algoritmos que se aplican simultáneamente a cada muestra[4]. Esto permite un mayor control de la señal frente a mayores requerimientos computacionales. Para este trabajo no se requiere un gran control de la señal, sino una herramienta para eliminar frecuencias altas.

El filtro se aplica a la aceleración y a la fuerza conjuntamente (ya que los estadísticos se calculan de ambas magnitudes). La transformación apenas afecta a la función de respuesta en frecuencia, ya que al ser el cociente de la potencia de entrada entre la potencia de salida, se amplifica la entrada y vuelve a su nivel original.

### 2.4.2. Remuestreo

El remuestreo consiste en aumentar o disminuir el número de muestras sin modificar ninguna otra característica de la señal, desde el punto de vista vibratorio (figura 2.4). Para ello no debe modificar ninguno de los parámetros en el dominio de la frecuencia. La fase y magnitud en cada frecuencia debe ser igual, para las frecuencias a las que sea aplicable la transformación.

Para ello se recurre al algoritmo de *fast fourier transform*, que calcula transformadas de fourier en conjuntos discretos [4]. Con la transformada de fourier se puede reconstruir la función en base a senos y cosenos que genera la señal original, para

Figura 2.3: Gráfica de una señal filtrada frente a la señal original en rojo.

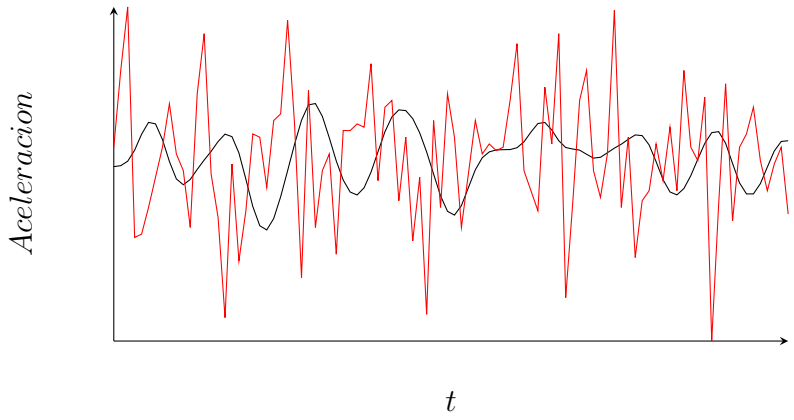
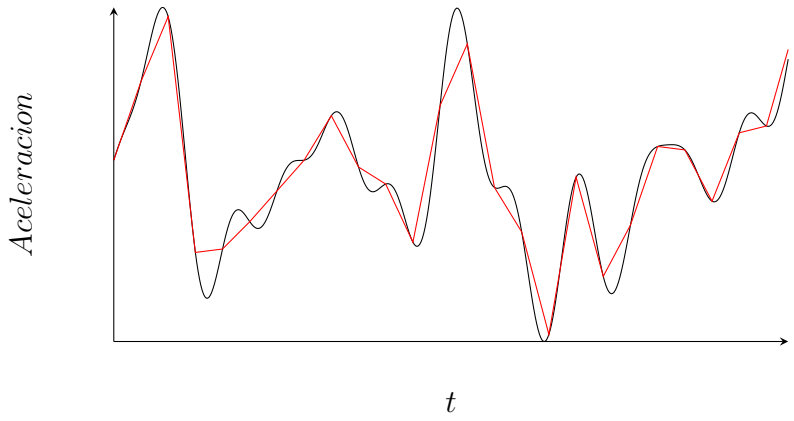



Figura 2.4: Gráfica de una señal remuestreada frente a la señal original en rojo.




posteriormente calcular la transformada inversa modificando el número de muestras.

### 2.4.3. Deducción de tendencia

En el ensayo vibratorio la estructura parte del reposo y retorna al estado inicial de reposo, siendo su posición final igual a la posición inicial. Bajo estas condiciones, la integral de la función aceleración entre el inicio y final del ensayo es nula. Sin embargo, por las imprecisiones en el equipo de medida, y la naturaleza discreta de la toma de medidas, los datos pueden no cumplir esa regla. A causa de este problema, los datos pueden no estar ajustados en torno al 0, de forma que haya un desfase por una constante $k$ o que exista una tendencia de los datos con pendiente $b$. En caso de evaluar los datos sin deducción de tendencia al algoritmo que calcula las tranformadas de fourier discretas, surgiría un pico para las frecuencias bajas: para una frecuencia 0, la transformada de fourier de una función es igual a la integral de la misma.

Para resolver este problema, simplemente se ha de sustraer a la función aceleración la recta de tendencia que presenta, de forma $k + bt$, donde $t$ representa el

tiempo.

## 2.5. Funciones de respuesta en frecuencia

Las funciones de respuesta en frecuencia representan la relación entre la entrada, frecuentemente en unidades de fuerza; con la salida del sistema, generalmente en aceleración, velocidad o desplazamiento. Éstas funciones permiten obtener las frecuencias y factores de resonancia de un sistema de una forma visual. También permiten el uso de algunos algoritmos de detección de modos; aunque hay otros algoritmos basados directamente en señales temporales, ambos se usan para aplicaciones diferentes.

### 2.5.1. Cálculo de las funciones de respuesta en frecuencia

El cálculo de funciones de respuesta en frecuencia a partir de una señal discreta es muy complejo, por lo que en el código se han usado algoritmos completos que calculan las funciones de respuesta en frecuencia en dos pasos. El proceso para calcular la función (figura 2.5) comienza con la señal temporal, que se divide en varias secciones o segmentos. Posteriormente se multiplican por una ventana [1], para después calcular la transformada de fourier discreta (DFT) con el algoritmo *fast fourier transform*. A countinuación se promedian, y se realiza una segunda transformación para obtener la densidad espectral de potencia ($P_{xx}$) y la densidad espectral de potencia cruzada ($P_{xy}$). Finalmente, se dividen ambas para obtener la función de respuesta en frecuencia.

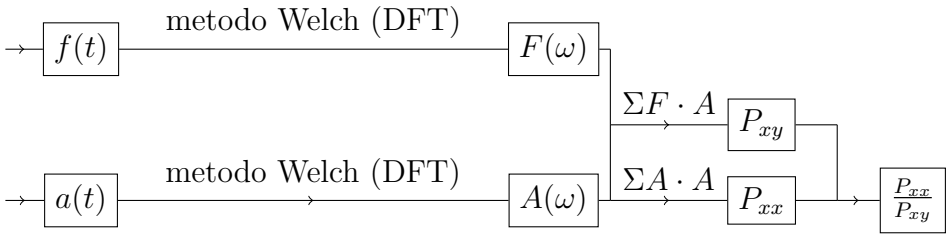
Generalmente se calculan otras dos funciones, $\frac{P_{yy}}{P_{xy}}$ y $\frac{|P_{xy}|^2}{P_{yy} \cdot P_{xx}}$, sin embargo, no se ha trabajado con dichas funciones en el trabajo.

Por otra parte, se está trabajando con impactos: en una misma señal se tendrá un número determinado de impactos, en vez de una función continua, o extendida en el tiempo. Por ese motivo se da una opción adicional: en caso de indicarse, cada impacto se tomará como una ventana completa, de términos indicados, y el algoritmo promediará las transformadas de fourier de cada impacto. Al tratar cada señal de forma idependiente los resultados obtenidos son ligeramente más aptos para su estudio.

Finalmente se quiere realizar una puntualización sobre el método Welch [5] y sus parámetros. la principal ventaja de este método reside en usar ventanas por cada intervalo: de esta forma podemos tener mejores resultado si la señal original tiene diferentes fases. Sin embargo, estos procesos tienen un problema, ya que el número -o resolución- de frecuencias depende del número de muestras: si no tenemos suficientes muestras la señal empeorará. Las funciones que se han usado tienen la capacidad de añadir muestras, de valor 0, al vector original. De esta forma, se está calculando la transformada de fourier con un número mayor de muestras iniciales, resultando en un mayor número de muestras de frecuencia. Por lo tanto, los parámetros para las funciones de respuesta en frecuencia serán:

[1] La ventana es una función que retorna un valor entre 0 y 1, se pretende amortiguar los extremos de la señal, mientras que los valores centrales corresponden a la función original [4]

Figura 2.5: Diagrama del proceso de cálculo de funciones de respuesta en frecuencia usado.




- nperseg (*n per segment*): indica el número de muestras que se toman de la muestra original.

- nfft (*Non-Uniform Fast Fourier Transform*): indica el número de muestras que se introducen en la transformada de fourier, si es mayor que nperseg se añaden muestras con valor 0.

- noverlap (*n overlap*): número de muestras que se solapan entre una ventana y la siguiente, si es 0 no se solapa ningún punto.

### 2.5.2. FRF para sistemas de un grado de libertad

#### Introducción

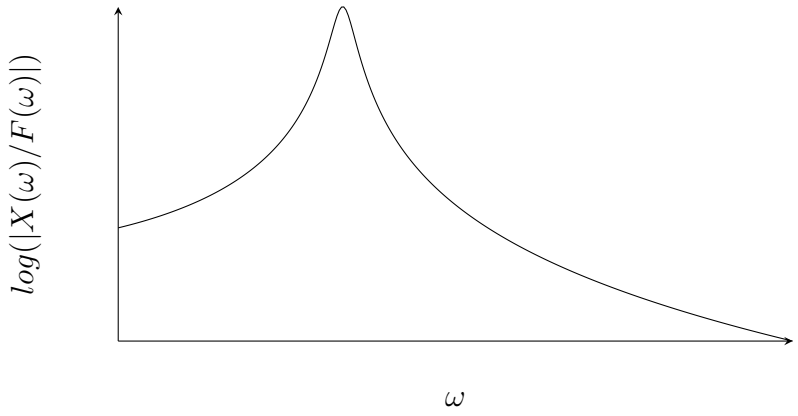
Es de gran interés comprender el funcionamiento de una funcion de respuesta en frecuencia para el caso específico de un grado de libertad. Todos los métodos usados para el cálculo de las frecuencias naturales y el amortiguamiento se basan precisamente en este sistema básico, ya que un sistema de n grados de libertad se pueden aproximar, con cierta precisión, a n sistemas de un grado de libertad.

#### Funciones de respuesta en frecuencia en aceleración, velocidad y tiempo

Un sistema de un grado de libertad se suele referenciar con una masa que oscila por la acción de una fuerza externa, un muelle de constante $k$ y un amortiguador viscoso de constante $c$ . Para el trabajo se ha usado amortiguamiento viscoso, representado por la variable $c$. Para el resto de fórmulas en el trabajo, especialmente en la identificación de modos, todos los amortiguamientos son viscosos.

El concepto de función de respuesta en frecuencia se refiere al cociente entre la magnitud de salida y la magnitud de entrada. La entrada o excitación al sistema es, en todos los casos de este trabajo, una fuerza. La magnitud de salida puede ser bien aceleración, velocidad o desplazamiento. Si bien el trabajo se ha enfocado exclusivamente al estudio de un sistema medido con acelerómetros, y por lo tanto se ha medido la aceleración, es conveniente cambiar entre aceleración, velocidad y desplazamiento para diferentes métodos de identificación modal.

Figura 2.6: Modulo de una función de respuesta en frecuencia de receptancia




### Receptancia

La receptancia se define como el conciente entre el desplazamiento y la fuerza, se representa en la ecuación (2.3). En la figura 2.6 se pueden observar sus características geométricas: La cola derecha forma una asíntota horizontal, mientras que la izquierda es una asíntota con pendiente negativa. A partir de estas asíntotas, en un sistema de un grado de libertad, se pueden calcular las constantes $k$, $m$ y $c$ de la ecuación, sin embargo no se puede aplicar en sistemas reales [6].

$$\frac{X(\omega)}{F(\omega)} = \frac{1}{k - \omega^2 m + j\omega c} \tag{2.3}$$

### Movilidad

La movilidad se define como el conciente entre la velocidad y la fuerza, se representa en la ecuación (2.4). En la figura 2.7 se pueden observar sus características geométricas: La parte derecha forma una asíntota con pendiente positiva, mientras que la izquierda es una asíntota con pendiente negativa, pero mitad que la observada en la receptancia[6].

$$\frac{\dot{X}(\omega)}{F(\omega)} = \frac{j\omega}{k - \omega^2 m + j\omega c} \tag{2.4}$$

### Inertancia

La inertancia se define como el conciente entre la velocidad y la fuerza, se representa en la ecuación (2.5). En la figura 2.8 se pueden observar sus características geométricas: La parte derecha forma una asíntota con pendiente positiva, mientras que la izquierda es una asíntota horizontal[6].

$$\frac{\ddot{X}(\omega)}{F(\omega)} = \frac{-\omega^2}{k - \omega^2 m + j\omega c} \tag{2.5}$$

Figura 2.7: Modulo de una función de respuesta en frecuencia de movilidad

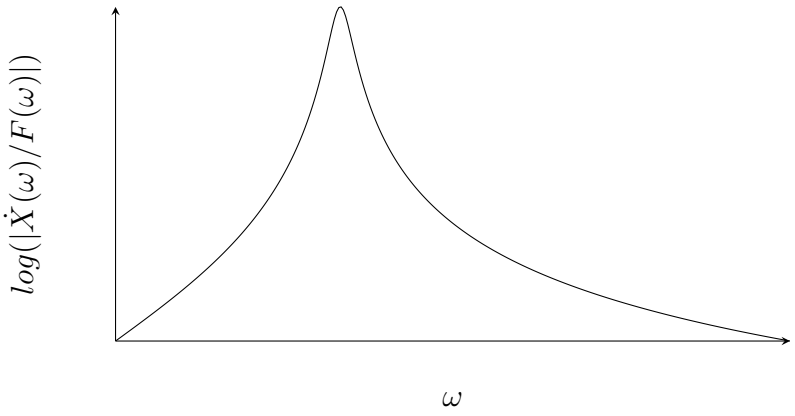
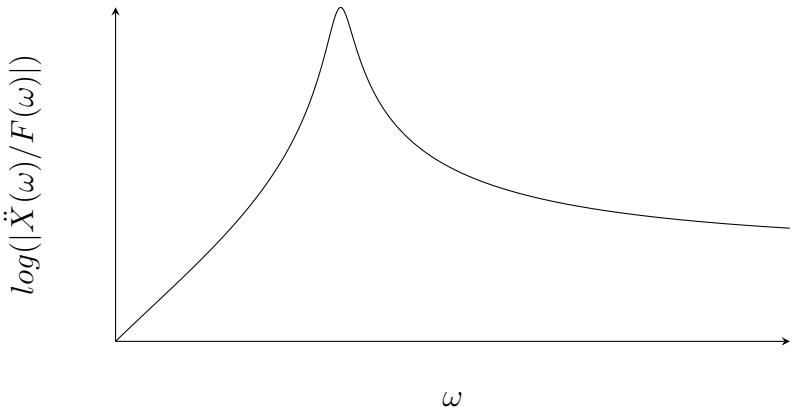



Figura 2.8: Modulo de una función de respuesta en frecuencia de inertancia



## 2.6. Detección de modos

Se han implementado dos métodos para calcular los factores de amortiguamiento y las frecuencias naturales, ambos basados en las suposiciones de un grado de libertad.

Para aplicarlos a sistemas de n grados de libertad se tienen que cumplir las siguientes condiciones:

1. Cada modo está suficientemente separado y diferenciado del resto.

2. El sistema estudiado es lineal.

En caso de que se cumplan ambas condiciones, se podrán ejecutar los métodos de identificación de modos, teniendo en cuenta que son aproximaciones y pueden dar resultados erróneos dependiendo de las condiciones de la muestra obtenida.

### 2.6.1. Peak picking

Probablemente considerado como el método de detección de modos más sencillo, se basa en detectar las frecuencias naturales directamente en los picos, y el amortiguamiento se calcula a partir del ancho del pico del módulo de la función de respuesta en frecuencia, preferiblemente en la receptancia: $\omega_r = |X(\omega)/F(\omega)|_{max}$[7].

El coeficiente de amortiguamiento viscoso, $\zeta$, se calcula a partir de dos frecuencias, $\omega_a$ y $\omega_b$, que se obtienen a partir del corte de la función de respuesta con la línea horizontal de valor $\frac{|X(\omega)/F(\omega)|_{max}}{\sqrt{2}}$ (figura 2.9). De esta forma, obtenemos $\zeta$ a partir de la expresión siguiente:

$$\zeta_r = \frac{\omega_b^2 - \omega_a^2}{4\omega_r^2} \tag{2.6}$$

El problema de el método subyace en que se está tratando con una señal de elementos discretos, siempre se deben interpolar los valores. Por otra parte, sólamente tiene en cuenta tres puntos de la función, por lo que no tiene ningún método de corregir el posible ruido de la señal.

### 2.6.2. Circle fit

En el diagrama de Nyquist, formado por el plano de los valores real e imaginario de la función de respuesta en frecuencia, se dibuja un círculo en la frecuencia propia del modo. A partir de las propiedades de dicho círculo se pueden calcular los factores de amortiguamiento del modo [8].

Para ello se toma el diagrama de Nyquist y se ajusta un círculo a los puntos obtenidos. Es posible que el círculo esté desplazado, esto se debe en parte a la contribución del resto de modos, pero no compromete el valor de la estimación. Aunque la aproximación a un círculo puede ser adecuada para los tres diagramas, sólo en el caso de la movilidad es una suposición exacta, como se observa en las figuras 2.10,2.11 y 2.12.

Como se puede comprobar, los puntos del círculo se espacian más cuanto más se acercan al valor máximo, esta propiedad nos permite interpolar la frecuencia propia

Figura 2.9: Corte de la línea de potencia mitad con la frf

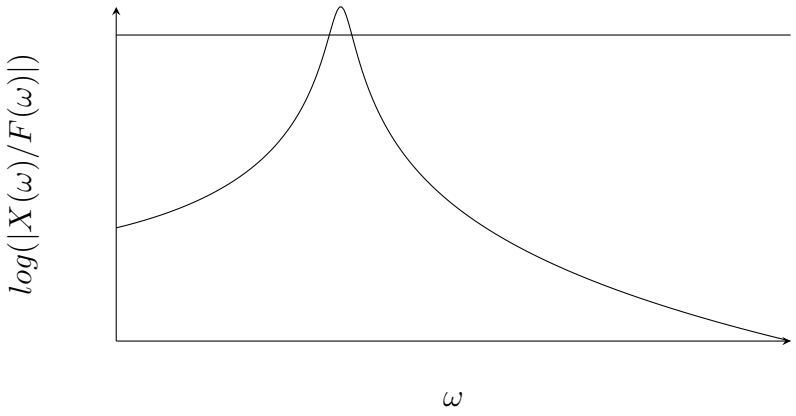
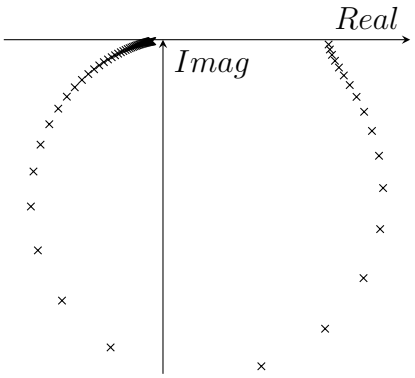



Figura 2.10: Diagrama Nyquist de receptancia para un grado de libertad




entre dos muestras. De forma matemática se calcula como el máximo de la derivada del ángulo del punto respecto del centro: punto en el que la segunda derivada es 0.

El factor de amortiguamiento se calcula a partir de los mismos ángulos, y sus respectivas frecuencias. La función del factor de amortiguamiento se puede calcular de dos formas, como función del ángulo entre cada punto del círculo y la frecuencia propia (obteniéndose un vector de $n$ elementos), o como función del ángulo de cada punto respecto a otro punto cualesquiera (obteniéndose una matriz de $(n-1)\times(n-1)$ elementos). Finalmente se calcula la media aritmética de todos los valores de la recta o el plano hallado.

Figura 2.11: Diagrama Nyquist de movilidad para un grado de libertad

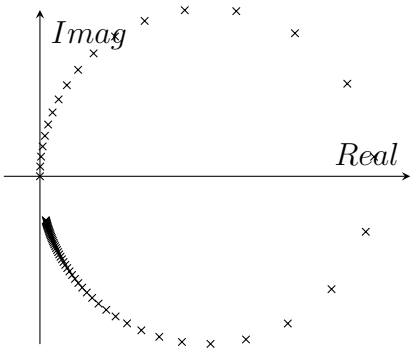
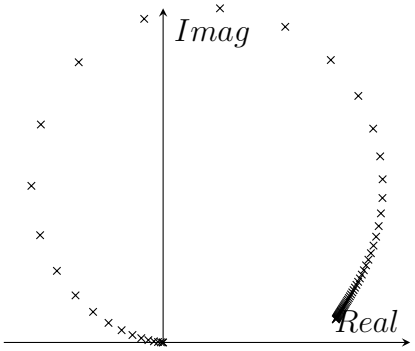



Figura 2.12: Diagrama Nyquist de inertancia para un grado de libertad



# Capítulo 3

# Desarrollo del código.

## 3.1. Introducción

El principal problema que se encuentra al intentar organizar una aplicación, más que adquirir el conocimiento sobre el lenguaje de programación, es la organización del propio programa. A medida que se insertan nuevas funciones dentro de un programa, estas deben ser accesibles y escribirse de forma clara y concisa, para que cualquier persona con mayor o menor conocimiento del lenguaje pueda entender el mismo.

El proceso de escritura del código de toda la aplicación ha sido iterativo. A medida que se han conocido las funcionalidades del programa, se han ido incorporando dentro código, reformulando gran parte del mismo con los nuevos algoritmos. Si bien no es un método eficiente para programar, la aplicación resultante es mucho más robusta, y el uso de la misma es mucho más fácil y claro, como se mostrarán en los siguientes capítulos.

## 3.2. Librerías usadas

Una librería se define como un conjunto de funciones que permiten obtener un resultado a partir de unas entradas indicadas en el propio programa. Tanto las entradas y salidas de las funciones están en el propio programa, y en ningún momento la librería tiene contacto con el usuario final. Son usadas para realizar tareas habituales y/o que requieren un alto conocimiento del programa.

- numpy v1.19.0: es la librería matemática por excelencia en python. Sirve como base para otras soluciones más específicas como scipy, pandas, jupyter...[9] Mejora la forma de guardar vectores, e integra una gran diversidad de funciones para trabajar con ellos. Para mejorar su rendimiento computacional, está escrita en otros lenguajes compilados, como C, por lo que su rendimiento es mejor que las propias funciones de python.

- scipy v1.6.0: incluye más de 3.000 funciones y módulos de álgebra, integración, optimización, procesamiento de imágenes, estadística...[10] En nuestro caso usaremos el módulo signal de la librería, que realiza procesamiento de todo tipo de señales discretas.

- math v 3.9.0: ofrece funciones matemáticas básicas con números reales. Aunque en muchos casos es superada por numpy, math tiene ciertas ventajas al realizar operaciones matemáticas con vectores.

- cmath v 3.9.0: es el equivalente a math, incluyendo el tratamiento de números complejos. Al igual que math, tiene ciertas funciones que son más adecuadas que las propias de scipy, y está mejor integrada dentro de python que su alternativa numpy.

- pandas v 1.1.4: se considera una de las mejores librerías para el análisis de datos desde el punto de vista estadístico, ya que introduce una forma alternativa para almacenar los datos. Sin embargo, en este caso se usa para la lectura de los ficheros excel.

- plotly v 4.13.0: librería para la creación de gráficas interactivas en html. En otros campos, como la customización, tiene grandes defectos frente a matplotlib.

- os: librería que determina la interacción entre python y el sistema operativo, permite trabajar con las rutas de los archivos, comprobar si un archivo existe... Forma parte del lenguaje de programación, no es necesaria su instalación.

- h5py: librería que facilita la implementación de la base de datos hdf5 en python.

## 3.3. Programación enfocada a objetos

La programación orientada a objetos es una forma de organizar el código para que sea más organizado, estructurado y que siga un orden lógico. De esta forma, se definen unos objetos que sirven de contenedor para todas las funciones y variables que comparten entre ellas una serie de características. Las ventajas que aportan al código, desde el punto de vista del usuario:

- El código es claro y conciso: tanto funciones como variables están organizadas con una estructura muy clara.

- Un objeto puede almacenar tantas variables como se quiera, no es necesario adjudicar una variable por cada vector.

Sin embargo, también tiene bastantes inconvenientes a la hora de manejar los propios datos. Una clase es un formato muy complejo en el que los datos no se encuentran en una estructura simple para el propio programa. A la hora de manejar las clases, los problemas que se han tenido son:

- Dificultad para guardar los datos en una base de datos: en una clase, los datos se estructuran de forma ramificada, mientras que muchas bases de datos presentan una estructura de tabla, donde todos los objetos tienen la misma jerarquía. Aunque existen bases de datos con jerarquía, no son tan frecuentes ni están tan optimizadas.

- No siguen la misma lógica que el resto de variables de python, ya que no es una variable al uso. Ésto hace que haya que informarse sobre sus propiedades si se quieren modificar.

Basándonos en la filosofía que ha seguido el trabajo, el uso de clases facilita la escritura de programas simples para usar el programa, mientras que la escritura de nuevas funciones, si bien requiere un cierto conocimiento de su funcionamiento, éste conocimiento no dista del uso básico de funciones.

## 3.4. Estructura de los objetos

La estructura de los objetos se ha enfocado desde un punto de vista simple para el usuario final, y compleja para programar nuevas funciones y su integración en bases de datos. Contiene muchas redundancias en los datos, y por ende un consumo alto de memoria por parte del programa. Sin embargo, los datos a analizar son del orden de megabytes, muy por debajo de la memoria de los ordenadores actuales. Ésta organización es ventajosa para funciones de gran duración, en las que es ventajoso extraer los datos de la memoria RAM que recalcularlos. Para aplicaciones web o de visualización de datos, sin embargo, aumenta el tiempo de ejecución.

En la estructura se diferencian tres clases, una de señales temporales (**Tiempo**), la segunda de señales en dominio de la frecuencia (**Frecuencia**) y la tercera para almacenar los resultados de los análisis modales (**Modal**).

Finalmente, se ha generado otra clase, denominada **Atributo**. La mayoría de los tipos de señales tienen que guardar consigo la unidad en la que se han tomado, el nombre del acelerómetro con el que se ha tomado la variable y el propio valor de la variable. Estos datos se tienen que consultar frecuentemente en conjunto, es conveniente almacenarlos en el mismo lugar.

La lista de todas las variables del programa es la siguiente, ordenadas según su clase:

- Tiempo
    - tiempo: recoge un vector de una dimensión con el tiempo discurrido entre la primera muestra y la muestra n.
    - aceleracion: recoge un vector de 2 dimensiones, donde el eje 0 indica el acelerómetro del que se están tomando muestras, y el eje 1 indica la señal generada por cada acelerómetro.
    - fuerza: almacena un vector de una dimensión con la señal de la fuerza ejercida por el martillo.
    - rms: indica el RMS de toda la señal. Es un *array* de una dimensión y un elemento por cada acelerómetro.
    - rms_movil: indica el rms durante un intervalo de tiempo, denominado ventana. de esta forma, se obtiene un valor equivalente a la potencia promedio en cada elemento del vector. Como se debe usar una ventana, la longitud total de este vector es menor que la de la señal original.
    - tiempo_rms_movil: vector de una dimensión que recoge el tiempo del RMS móvil. No recoge sus valores de unidad y nombre.

- mtvv: vector de una dimensión que recoge el máximo calculado en el RMS móvil, para cada acelerómetro.
- vdv: matriz en la que el eje 0 indica el acelerómetro a estudiar, y el eje 1 consiste en la señal del estadístico vibration dose value. su longitud es de un elemento menor que la señal original.
- samplerate: frecuencia de muestreo, se calcula a partir del vector tiempo. Es un escalar, y no recoge la estructura de la clase atributo, es usado como elemento de cálculo para las funciones internas principalmente.
- axis_0, axis_1: son números enteros, que toman el valor del número de acelerómetros y del número de valores en la señal. son útiles para iterar entre los vectores, y aumentan la legibilidad del programa.

- Frecuencia
  - complejo: tanto python como la librería numpy dan soporte para números complejos, son guardados en la forma $a+b\cdot i$. Esta variable recoge, dentro de la clase atributos, una matriz donde cada columna corresponde a la señal de un acelerómetro, en concordancia con los datos de aceleración. Los números complejos son siempre el resultado de una FRF.
  - magnitud: la magnitud es el módulo del complejo, en este caso se denomina magnitud por tradición en el campo de las vibraciones. Al igual que el objeto çomplejo", se almacena en una matriz, con el mismo tamaño.
  - fase: fase del número complejo, en radianes. Se almacena de la misma forma que la magnitud.
  - fase_corr: representa la fase corregida para su representación en gráficas. Al obtener la fase de un número complejo se obtiene un valor que oscila entre el mínimo y el máximo valor, generando líneas verticales. Presenta el mismo formato que la fase ordinaria.
  - real: se representa, de la misma forma, el valor real de la variable complejo. Si bien python acepta de forma nativa números complejos, otras librerías usadas no son capaces de guardar y representar números complejos, siendo necesario su almacenamiento en variables separadas.
  - imag: representa la parte imaginaria como un número decimal, complementando al objeto anterior.
  - frecrate: representa la resolución en frecuencia, es usado para el funcionamiento de otras funciones, principalmente.
  - axis_0, axis_1: son números enteros, que toman el valor del número de acelerómetros y del número de valores en la señal.

- **Modal**: los elementos de la clase modal no incluyen el elemento **Atributo**, dentro del alcance de este trabajo resulta superflua dicha estructura de datos. En la clase **Modal** los valores se guardan directamente en sus variables.
  - frf: dentro de la clase **Modal** se guarda el objeto de la clase **Frecuencia** para usarlos como datos para los cálculos.

- rango_frec: sirve de variable de entrada, en la que se representan los valores numéricos de el rango de frecuencias que se pretende estudiar. Tiene una dimensión de $m \times n \times x2$, donde $m$ representa el número de funciones de respuesta en frecuencia, $n$ el número de modos a calcular, y 2 referencia al valor mínimo y máximo del rango de frecuencias donde se encuentra el vector.
- pos_rango_frec: matriz con la misma geometría que la anterior. Recoge la posición de la frecuencia de la variable rango_frec en la señal del la función de respuesta en frecuencia. Sirve como variable interna para los métodos de identificación modal.
- frec_resonancia: matriz de tamaño $m \times n \times 2$, donde $m$ representa el número de funciones de respuesta en frecuencia y $n$ el número de modos calculados.
- amortiguamiento: de forma idéntica a la variable anterior, representa el amortiguamiento de cada modo.
- axis_0, axis_1: son números enteros, que toman el valor del número de funciones de respuesta en frecuencia y el número de modos a detectar.
- frec_resonancia_media: vector con tantos vectores como modos se hayan introducido. Representa la media aritmética de la frecuencia natural de los modos calculados.
- amortiguamiento_medio: vector con tantos vectores como modos se hayan introducido. Representa la media aritmética de los amortiguamientos de los modos calculados.

### 3.4.1. Acceso a los datos de clases

La estructura en forma de clases permite el acceso a los datos de una forma ordenada y clara, ya que todos los datos se sitúan en una estructura ramificada, con una nomenclatura fija para las variables.

En el siguiente ejemplo se carga la función datos en primer lugar, y posteriormente se accede a los datos de la misma:

```
import amodal as am
# Se define el objeto 'senal' a partir de la libreria
senal = am.Tiempo()
# Se asignan unos datos al objeto senal
senal.lectura('datos.xlsx')
# Se imprime en pantalla el vector tiempo, la unidad ('s'),
# y la matriz aceleracion
print(senal.tiempo )
print(senal.tiempo.u)
print(senal.aceleracion )
```

Gracias a esta herramienta, podemos enviar varios valores a una función de una forma cómoda e intuitiva:

```
import amodal as am
```

```
# Se define el objeto 'senal' a partir de la libreria
senal = am.Tiempo()
# Se asignan unos datos al objeto senal
senal.lectura('datos.xlsx')
# Se define una funcion que imprime en pantalla el vector tiempo,
# la unidad ('s'), y la matriz aceleracion
def imprimir(objeto):
    print(objeto.tiempo )
    print(objeto.tiempo.u)
    print(objeto.aceleracion )

# Se imprimen en pantalla los datos requeridos
imprimir(senal)
```

## 3.5. Funciones

Una función es un conjunto de código que se ejecuta cuando es llamada por otra función o durante la ejecución del programa. En este trabajo, conforman la parte principal del código, ejecutan todos los procesos que transforman la señal.

Al igual que las variables, también pueden agruparse en una clase, de forma que pueden tomar las variables de dicha clase como entrada, sin necesidad de nombrarlas. Por esta capacidad, la mayor parte de las funciones se encuentran dentro de la clase donde corresponda su función. Se pueden organizar de la siguiente forma, como funciones principales, de soporte y de lectura y estructura.

En los siguientes apartados nos centraremos en cómo funcionan las funciones principales del programa, aquellas que transforman los datos de una u otra forma. Sin embargo, una gran cantidad de las funciones del programa sirven de soporte para importar y exportar datos, inicializar las clases, calcular parámetros como la frecuencia de muestreo, o transformar un tipo de variable en otro. En la documentación, apéndice A, se incluye una breve descripción de cada función programada.

### 3.5.1. Cálculo de estadísticos

Los estadísticos son valores sencillos, cuyo valor está descrito por una función matemática sencilla. Si bien estas funciones incluyen llamadas a otras funciones de soporte, los elementos esenciales de las mismas se recogen en las siguientes líneas de python.

En primer lugar, el RMS es uno de los estadísticos más conocidos, definido como la raíz cuadrada del promedio de los cuadrados de los elementos de la muestra. La implementación en python es muy sencilla:

```
rms = sqrt(mean(valor ** 2))
```

A partir del RMS podemos desarrollar el RMS móvil, como una función igual al RMS que se aplica por intervalos a la señal. Para ello iteramos dentro de un bucle la función RMS, donde cada incremento, denotado por la variable i, modifica el vector que se introduce a la función.

```
for i in range(tamano):
    valor = senal[avance * i, ventana + avance * i]
    rms_movil = sqrt(mean(valor ** 2))
```

Gracias al RMS móvil podemos definir el tercer estadístico, mtvv o *maximum transient vibration value*, como el valor máximo del RMS móvil. La librería numpy nos proporciona una función para calcular el máximo, por lo que el código, de forma simplificada, se reduce a:

```
mtvv = np.amax(rms_movil)
```

Finalmente, la obtención del VDV o *vibration dose value* requiere más atención. El VDV es un estadístico que se acumula a lo largo del tiempo, el elemento i se obtiene como la función de VDV aplicada al intervalo [0,i]. El algoritmo más rápido para realizar esa operación consiste, en primer lugar, en definir una variable que guarde el sumatorio de los términos a la cuarta. Finalmente, para obtener el valor real de cada elemento, se realiza la raíz cuarta del mismo.

```
acumulado = 0
for i in range(tamano-1):
    acumulado = acumulado + aceleracion[i]**4
    vdv[i] = acumulado ** 0.25
```

### 3.5.2. Tratamiento de datos

El bloque de tratamiento de datos se compone de tres funciones, **detrend**, **resample** y **filtro**. Se ha optado por usar las respectivas funciones de la librería **scipy.signal**, por lo que las funciones de la librería generada son muy simples.

El detrend consiste en eliminar la tendencia lineal de la señal. Su programación en python sería muy sencilla, pero se ha optado por usar la librería scipy ya que generalmente incorpora métodos más optimizados. El código de la función es el siguiente:

```
aceleracion = signal.detrend(aceleracion)
```

Para modificar la frecuencia de muestreo de la señal, o *resample*, se han usado dos funciones. En primer lugar, para los datos aceleración y fuerza se usa la función **resample**, que realiza las transformadas de fourier a la señal original, y genera la transformada inversa para obtener el nuevo número de muestras. Sin embargo, para el tiempo se genera un vector nuevo, con el samplerate corregido, a partir de la función **linspace**. Dicha función cuenta de un valor inicial a un valor final con un número determinado de pasos.

```
aceleracion = signal.resample(aceleracion, num_muestras, axis=-1)
fuerza = signal.resample(fuerza, num_muestras, axis=-1)

tiempo  = np.linspace(0,fin, num_muestras)
```

Finalmente se ha programado una función de filtro. La función consiste en dos partes, en primer lugar se genera una función de transferencia de un filtro tipo *butterworth*, y en segundo lugar se aplica la función de transferencia a la señal con

un filtro tipo IIR, o *Infinite Impulse response*. Se ha escogido este tipo de filtro porque tiene menor carga computacional, y en este caso no es crítico que el filtro tenga una forma concreta.

```
b, a = signal.butter(orden_filtro, frecuencia_filtro, 'low', fs=samplerate)
# a y b son dos polinomios, el denominador y numerador de la
# funcion de transferencia, respectivamente
aceleracion = signal.lfilter(b, a, aceleracion)
fuerza = signal.lfilter(b, a, fuerza)
```

### 3.5.3. Generación de las funciones de transferencia

La generación de las funciones de transferencia es un punto crítico en el análisis modal, puede modificar en gran medida la detección de los modos de vibración del sistema, que es el objetivo final del trabajo de fin de grado. Para ello intervienen principalmente dos funciones de la librería **scipy.signal**: **csd** y **welch**. **welch** calcula la densidad espectral de potencia de la señal de fuerza,$P_{xx}$, mientras que **csd** calcula la densidad espectral cruzada de la señal de aceleración cruzada, $P_{xy}$. Ambas dependen de los factores **nfft**, **nperseg** y **noverlap**. Finalmente, la función de respuesta en frecuencia es el cociente de ambas: $P_{xy}/P_{xy}$, a partir de la cual se crea un objeto de la clase **Frecuencia**.

Sin embargo, antes de calcular la función de respuesta en frecuencia se debe adecuar la señal y los parámetros. En primer lugar, se debe procurar que las frecuencias obtenidas en la frf no sean números decimales periódicos, sino finitos. Para ello, se usa la función **frecrate_entero()**, definida dentro de la función principal. **Frecrate_entero** itera entre los valores próximos al nfft introducido, y comprueba si alguno de ellos generaría frecuencias decimales no periódicas. En caso de no ser así, retorna el valor original de los tres parámetros.

En segundo lugar se sitúa la función **impacto_a_nperseg()**. La función detecta los picos en la fuerza, gracias al límite definido en los valores por defecto, y recorta las señales para que el valor de nperseg coincida con la longitud de cada impacto. Gracias a esto, cada segmento que se introduce es exactamente un impacto, mejorando los resultados. Por último, se eliminan las tendencias de cada segmento, lo que elimina las frecuencias anormalmente bajas.

La simplificación del código de la función terminada es la siguiente:

```
# Tratamiento previo de datos
nperseg, noverlap, nfft = frecrate_entero(nperseg, noverlap, nfft,
                           samplerate)
fuerza,aceleracion = impacto_a_nperseg(fuerza, aceleracion, nperseg)

# Calculo de Pxx y Pxy
frec_1, Pxy = signal.csd(aceleracion, fuerza, fs=samplerate,
                         window=ventana_frf,
                         nperseg=nperseg, noverlap=noverlap, nfft=nfft,
                         detrend=False)

frec_2, Pxx = signal.welch(fuerza, fs=samplerate, window=ventana_frf,
```

```
                                nperseg=nperseg_frf, noverlap=noverlap,
                                nfft= nfft, detrend=False)

# Obtencion de la funcion de respuesta en frecuencia
frfrecuencia = Pxy / Pxx

# Creacion de un objeto de la clase Frecuencia
return Frecuencia(complejo=frfrecuencia, leyenda=leyenda,
unidad=unidad, frecuencia=frec_1)
```

### 3.5.4. Integración y cambio de variables complejas

La clase **Frecuencia** se compone en su mayoría de funciones para el cambio entre tipos de variables complejas, ya que cada función y gráfica requiere de un tipo específico de datos. Estas funciones son bastante simples y no requieren atención desde el punto de vista del trabajo. Sin embargo, hay dos funciones que sí que son importantes para la funcionalidad del programa: **corregir_fase()** e **integrar()**.

Tras calcular la función de respuesta en frecuencia, y representar su resultado en un bode, es frecuente que los valores de la fase oscilen entre 0 y $2\pi$ radianes. A la hora de hacer la gráfica, dichas variaciones entre el valor mínimo y máximo de la fase originan líneas horizontales que dificultan la legibilidad de los diagramas. Para solucionar el problema se ha programado una función que detecta cambios de fase de más de $2\pi$ radianes y suma o resta una vuelta completa. De esta forma, la línea de fase es una línea continua en todo el espectro.

```
for j in range(tamano - 1):
    # Definimos la variable k, para finalizar el bucle tras 30 iteraciones
    k = 1
    while True:
        # Evitamos valores por debajo de 1 Hz, genera problemas
        if self.frecuencia.v[j] <= 1.0:
            break
        valor = self.fase_corr.v[i, j + 1] - self.fase_corr.v[i, j]
        if abs(valor) < np.pi + 0.001:
            break
        if valor < 0:
            self.fase_corr.v[i, j + 1] += (np.pi * 2)
        else:
            self.fase_corr.v[i, j + 1] -= (np.pi * -2)
        if k > 30:
            break
        k += 1
```

La segunda función integra el espectro para convertir la acelerancia en movilidad o receptancia. Al estar trabajando con señales sinusoidales, integrar en el tiempo consiste en sumar $\frac{\pi}{2}$ a la fase y dividir la magnitud por la frecuencia:

```
fase  = fase  + np.pi / 2
fase_corr = fase_corr + np.pi / 2
```

```
# Se crea una nueva variable identica para operar
frecuencia_mod = frecuencia
# Se reeplaza el primer valor, que es 0, ya que se va a dividir
# la magnitud entre este vector
frecuencia_mod[0] = 1
magnitud = magnitud / frecuencia_mod
# Se recupera el valor inicial, 0
frecuencia [0] = 0
```

### 3.5.5. *Peak Picking*

Los algoritmos de detección de modos son sencillos en su funcionamiento, pero su implementación ha sido la más compleja. En primer lugar se ha programado el *peak picking*. El proceso completo se desarrolla en tres pasos: detectar la frecuencia a la que se da el pico, calcular las dos frecuencias que están exactamente 3dB por debajo del pico, y finalmente se calcula el coeficiente de amortiguamiento a partir de las tres frecuencias obtenidas.

1. Detección del pico: a partir de un intervalo introducido por el usuario, se calcula la posición de la señal donde se da el máximo, con la funcion de numpy **argmax**. Es conveniente trabajar directamente con las posiciones de los vectores, en vez de con los valores de los vectores. Al final del proceso se convertirá la posición del vector en una frecuencia para calcular el valor de la frecuencia natural.

2. En segundo lugar la señal se divide en dos: la señal previa al pico, y la señal posterior al pico, a partir de las cuales se calculará su corte con la línea horizontal a -3dB del pico. Este cálculo se ha separado en una función diferente: **peak_picking_cross**. En **peak_picking_cross** se introduce el valor de la señal, ya cortada, y el valor de la señal a -3dB.

3. El siguiente que ejecuta la función es restar 3dB a la señal, de forma que tiene que calcular ahora su corte con 0, un proceso más rápido desde el punto de vista del ordenador. Con la nueva señal, calcula en qué posiciones corta el vector con 0 (figura 3.1), interpolando para obtener una mayor precisión (figura 3.2). Finalmente, se calcula la media de todas las posiciones que ha detectado el algoritmo.

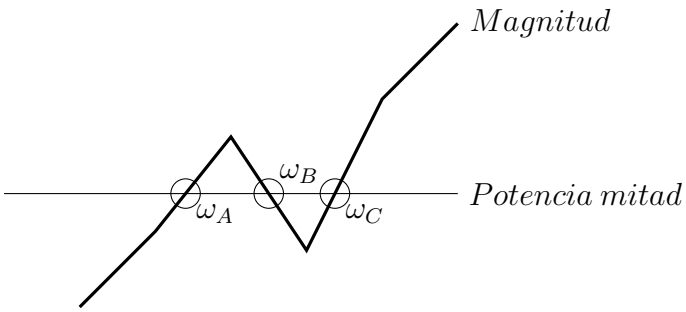
4. Finalmente, se traducen el valor de la posición en el vector a una frecuencia con la variable **frecrate**. El factor de amortiguamiento se obtiene con la ecuación $\zeta = \frac{w_2 - w_1}{2 \cdot w_n}$.

El código se ha finalizado con varias condiciones if capaces de detectar si hay un error en alguno de los elementos, y en caso de haberlo se devolverá el valor 0, tanto para la frecuencia como el amortiguamiento.

### 3.5.6. *Circle Fit*

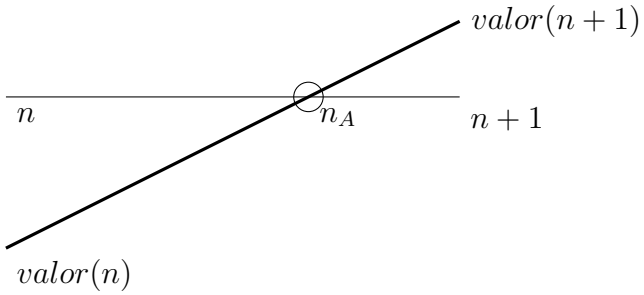
El proceso de *circle fit* es más complejo que el anterior, incorpora más algoritmos y funciones. Se obviará la estructura de funciones para centrarse en los pasos

Figura 3.1: Detalle del corte de la variable magnitud con la potencia mitad




$$\omega = \frac{\omega_A + \omega_B + \omega_C}{3}$$

Figura 3.2: Detalle de la interpolación




$$n_A = n + \frac{valor(n)}{valor(n) - valor(n+1)}$$

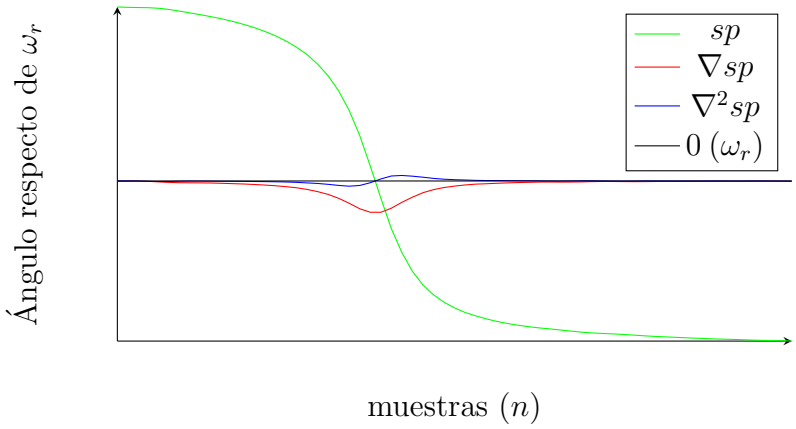
específicos que realiza el programa de forma secuencial: en este caso el único objetivo de las funciones es ordenar el código en bloques, en vez de usarse para facilitar la repetibilidad del código. En este caso se ha seguido el método de *circle fit* más común y sencillo: en primer lugar se ajusta un círculo a la nube de puntos del diagrama de Nyquist. La frecuencia natural corresponde a la máximo ángulo que formen dos puntos consecutivos, y el factor de amortiguamiento es función de los ángulos entre los puntos y la frecuencia natural.

1. En primer lugar se aproxima el centro del círculo. Se parte de una estimación: el centro corresponde al valor medio de las coordenadas x e y de todos los puntos. Es una aproximación errónea, pero más efectiva que otras aproximaciones que se han probado.

2. El siguiente paso consiste en definir una serie de funciones que calculen la distancia del centro a cada uno de los puntos: se define la función a minimizar. Sus entradas son las coordenadas x e y del centro, y su salida es un vector con la distancia que se aleja cada punto del círculo.

3. Se introduce la función descrita previamente junto con la estimación del centro en la función de scipy **optimize.leastsq()**. El valor retornado corresponde a

Figura 3.3: Gráfica de la evolución de sp y sus gradientes.




las coordenadas x e y del círculo.

4. Se calcula el ángulo entre un punto y el eje x. Para ello calculamos la arcotangente de la función $\frac{y-y_c}{x-x_c}$, y se guardan los resultados en la variable **sp**. Para calcular la tangente se usa la función de numpy arctan2, que retorna el ángulo en la circunferencia completa, teniendo en cuenta los signos del numerador y denominador. La variable **sp**, sin embargo, es un valor entre 0 y $2\pi$, no recoge valores superiores a $2\pi$, por lo que la función final puede presentar un salto en caso de que los datos comprendan más de $2\pi$ radianes. La función programada detecta si la gráfica pasa del tercer al cuarto cuadrante o viceversa, sumando o restando $2\pi$ según sea necesario.

5. El siguiente paso necesario es calcular el máximo de la derivada de la variable **sp**. para ello, con la función gradient de numpy se calcula la primera y la segunda derivada de sp (figura 3.3). Si bien podemos obtener el máximo directamente a partir de la primera derivada, es más interesante calcularlo como el corte de la segunda derivada con 0: este segundo método nos permite interpolar los datos, y obtener más precisión. El corte de la segunda derivada con 0 se calcula con la función descrita en el figura 3.2.

6. El punto de corte anteriormente descrito es la frecuencia natural de dicho modo. El siguiente paso corresponde a centrar los valores de la variable sp. Para ello sumamos a la variable **sp** el valor $k$, donde $k$ es el valor de la función sp en la frecuencia natural calculada.

7. Finalmente, la variable **sp** consiste en los ángulos de cada punto del diagrama de Nyquist con la frecuencia natural, lo que nos permite aplicar la ecuación $\zeta_i = \frac{w_r^2 - w_a^2}{2 \cdot w_r \cdot w_a} \frac{1}{tan(\frac{\theta_a}{2})}$ para calcular el factor de amortiguamiento de cada punto. El valor final del factor de amortiguamiento será la media.

## 3.6. Escritura de programas en python

Generalmente se considera que la gramática de python es una de las más fáciles para aprender, ya que evita el uso de carácteres especiales y permite mucha flexibilidad en la reasignación de variables, entre otras. En esta sección nos enfocaremos en tres ejemplos que muestran parte de las propiedades del software escrito.

### 3.6.1. Ejemplo 1

El primer ejemplo permite ver la función de respuesta en frecuencia, y muestra las gráficas del proceso de *circle fit*. Permite comprobar los datos, y ver dónde hay errores y fallos en los datos.

```
import numpy as np
import amodal as am
import graficas as gr

'''Definicion de variables'''
RUTA = 'PruebaViga200213-1.xlsx'

# Se carga la senal de tiempo
senal = am.Tiempo()
senal.lectura(RUTA)
# Obtener la frf, se integra para obtener movilidad
frf = senal.get_frf()
frf.integrar()
# se escribe un fichero con el diagrama bode
gr.bode_html(frf,'bode.html')
# Definimos el rango de frecuencias a analizar, y la clase modos
rangos_modos = np.array([[56,58]])
modos = am.Modal(frf,rango_unico=rangos_modos)
# Se calcula el circle fit, exportando las graficas de debug
modos.circle_fit(debug_sp=True, debug_psi=True ,debug_nyquist=True )
```

El programa es bastante breve, se puede usar como un primer análisis de los datos para comprobar rápidamente si se han tomado de forma adecuada.

### 3.6.2. Ejemplo 2

El segundo programa sirve para comprobar los modos con diferentes parámetros de la función de respuesta en frecuencia. Es muy importante conocer si la función de respuesta en frecuencia es correcta o no, a partir de este algoritmo se puede hallar los mejores valores para una cadena de medidas determinada.

```
import numpy as np
import amodal as am
import graficas as gr
import csv
```

```
'''PARAMETROS'''
RUTA = 'VigaMaderaA-3-01.xlsx'
# factor por el que se multiplica nperseg
NFFT = np.array([1,2,3,4])
NPERSEG = np.array([2048, 4096, 8192, 16384])
RANGO = np.array([[1,2],[3,7],[11,14],[19,24],[29,35]])

senal = am.Tiempo()
senal.lectura(RUTA)

# Bucle para calcular todos los parametros
for nfft in NFFT:
    for nperseg in NPERSEG:
        # Se calcula la frf
        am.preset.nperseg_frf = nperseg
        am.preset.nfft_frf = nfft*nperseg
        frf = senal.get_frf()
        frf.integrar()
        frf.integrar()
        # Se calculan los modos de resonancia
        modos = am.Modal(frf,rango_unico=RANGO)
        modos.peak_picking()
        modos.promedio()
        # Se guardan los datos en tablas csv, como texto.
        with open('frec-{}-{}'.format(nperseg,nfft) ,'w') as file:
            writer = csv.writer(file,delimiter='\t')
            for acelerometro in modos.frec_resonancia:
                acelerometro = np.round(acelerometro,decimals=5)
                writer.writerow(acelerometro)
            writer.writerow(np.round(modos.frec_resonancia_media,decimals=5))
        with open('amor-{}-{}'.format(nperseg,nfft) ,'w') as file:
            writer = csv.writer(file,delimiter='\t')
            for acelerometro in modos.amortiguamiento:
                acelerometro = np.round(acelerometro,decimals=5)
                writer.writerow(acelerometro)
            writer.writerow(np.round(modos.amortiguamiento_medio,decimals=5))
```

### 3.6.3. Ejemplo 3

Finalmente, se ha realizado un código, de mayor longitud y complejidad, para detectar automáticamente los modos de vibración de una pieza a través de las funciones de respuesta en frecuencia. Tiene determinados errores: en caso de que varios acelerómetros estén en valles, no se puede detectar correctamente. Sin embargo, funciona para un primer análisis, o como parte de un programa más complejo.

Su funcionamiento es similar al rms móvil u otros algoritmos que iteran segmentos de muestras. Este algoritmo, a partir de la función de respuesta en frecuencia, calcula los modos de vibración para cada segmento de la misma. Si en ese intervalo

hay efectivamente un modo, los amortiguamientos no darán errores, pero si no hay ningún modo, la mayoría de los amortiguamientos retornarán un error.

```
import numpy as np
import amodal as am
import graficas as gr

'''PARAMETROS'''
RUTA = 'VigaMaderaA-3-01.xlsx'
VENTANA = 3 # Ventana en Hz
AVANCE =  0.25  # Avance en tanto por uno

# Calculamos la frf y creamos la clase modos
senal = am.Tiempo()
senal.lectura(RUTA)
frf = senal.get_frf()
frf.integrar()
rango = np.array([[1,2]])
modos = am.Modal(frf,rango_unico= rango)

# Definimos la ventana, avance, y el numero final de elementos
ventana = int(frf.frecrate * VENTANA)
avance = int(ventana * AVANCE)
tamano = int((frf.axis_1 + avance-ventana-1)//avance)

# Se definen dos vectores vacios para guardar los datos
# Frecs guarda aquellos datos que no tienen errores en ninguna senal
frecs = np.zeros(0)
# Frecs1 guarda aquellos datos que tienen un error en alguna senal
frecs1 = np.zeros(0)

for i in range(tamano):
    # Calculamos la ventana y los modos en cada iterecion
    frec = np.array([[i*avance/modos.frf.frecrate,
                      (i*avance+ventana)/modos.frf.frecrate]])
    modos.adaptar_rango(frec)
    modos.circle_fit()
    # Se detectan cuantos valores '0' hay en los modos
    error_amortiguamiento = np.count_nonzero(
                            modos.amortiguamiento == 0)
    if error_amortiguamiento==0:
        modos.promedio()
        frecs = np.append(frecs, modos.frec_resonancia_media[0])
    if error_amortiguamiento<=2:
        modos.promedio()
        frecs1 = np.append(frecs1, modos.frec_resonancia_media[0])

print('Las frecuencias naturales sin errores en
```

```
       los amortiguamientos son:\n', frecs)
print('Las frecuencias naturales con menos de
       dos errores en el amortiguamiento son:\n', frecs1)

# Se combinan las frecuencias que estan muy cerca,
# se halla la media de los valores.
resonancias = np.zeros([0])
cache = np.zeros([0])
for i in range(len(frecs1)):
    if i == 0:
        resonancias = np.append(resonancias,frecs1[i])
        continue

    if abs(frecs1[i]- frecs1[i-1])  < 1:

        cache = np.append(cache,frecs1[i])
        resonancias[-1] = np.mean(cache)

    else:
        resonancias = np.append(resonancias,frecs1[i])
        cache = np.zeros([0])

print('Las frecuencias naturales son:',resonancias)
```

## 3.7. Aplicación web

Gracias a la facilidad para aprender a programar en Python, han surgido numerosos *frameworks* o plataformas para desarrollar aplicaciones que requerirían de otra forma el aprendizaje de otros lenguajes. A costa de una pérdida de funcionalidad, los *frameworks* aportan un soporte relativamente fácil y rápido de aprender. Gracias a Flask, un *framework* para el desarrollo de aplicaciones web, se ha desarrollado una aplicación web completamente funcional para el cálculo y la visualización de los datos tomados.

Al igual que una librería, un *framework* es un conjunto de funciones que realizan una serie de acciones. Sin embargo, el *framework* interactúa directamente con el usuario, y a partir de dicha interección manda órdenes al programa para que las ejecute, mientras que una librería es un conjunto de funciones que son llamadas por el programa principal.

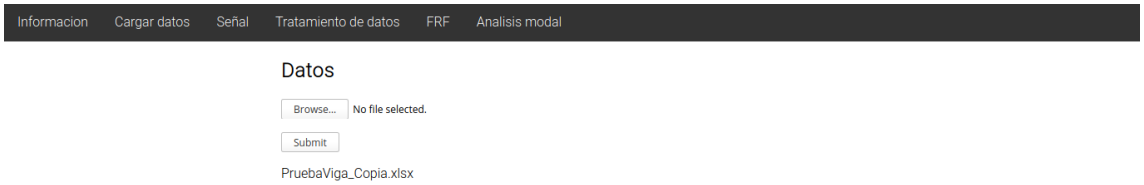
### 3.7.1. Estructura la aplicación

La aplicación se compone de 5 secciones, que corresponden a un grupo de acciones común. Cada sección es una página nueva, por lo que se pueden abrir diferentes pestañas en el navegador, y compartirán los datos.

En primer lugar, en la dirección principal, se encuentra una introducción a la propia aplicación, sobre sus usos y una hipervínculo al trabajo de fin de grado, como se observa en la figura 3.4.

Figura 3.4: Introducción a la aplicación


Figura 3.5: Pestaña para cargar datos en la aplicación




En segundo lugar se ha creado una página para cargar los datos (figura 3.9), en la ruta /cargar_datos". Desde aquí se pueden cargar ficheros tipo excel, con el formato de datos que se ha seguido durante todo el proyecto. Una vez subidos los datos, la aplicación lee los datos, calcula los estadísticos y las variables relevantes de la clase **Tiempo**, y procede a crear una base de datos tipo hdf5 dentro de una carpeta del servidor.

En tercer lugar, se ha generado una página (figura 3.6,figura 3.7) para mostrar la señal temporal y los estadísticos relevantes, /senal". Se pueden visualizar cada una de las señales indepentientemente, tanto de aceleración, fuerza, el RMS móvil, y el VDV. Finalmente, en la parte inferior de la página, se muestra una tabla con el RMS, MTVV y VDV de la señal.

En cuarto lugar se ha implementado una página (**??**,**??**) para tratar la señal original para poder filtrar, cambiar la frecuencia de muestreo y eliminar tendencias, en la ruta /tratamiento". La página introduce unos determinados valores por

Figura 3.6: Cuadros para seleccionar la variable y gráfica de la variable seleccionada

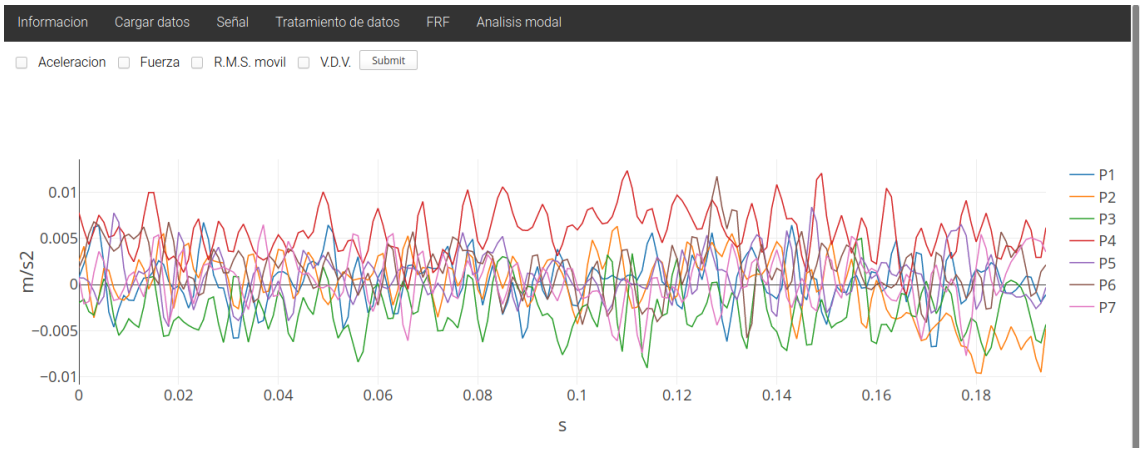



Figura 3.7: Estadísticos

| | P1 | P2 | P3 | P4 | P5 | P6 | P7 |
|---|---|---|---|---|---|---|---|
| RMS | 0.00265 | 0.00336 | 0.00384 | 0.00626 | 0.00262 | 0.00327 | 0.00269 |
| MTVV | 0.00482 | 0.00814 | 0.00627 | 0.01022 | 0.0057 | 0.00864 | 0.00506 |
| VDV | 0.01319 | 0.01649 | 0.01727 | 0.02633 | 0.01334 | 0.01644 | 0.01328 |

defecto. Por otra parte, en este apartado se ha introducido la interfaz para introducir los valores necesarios y calcular las funciones de respuesta en frecuencia. La quinta página (figura 3.10, 3.11) muestra los datos de las funciones de respuesta en frecuencia, tanto el diagrama bode como el diagrama Nyquist. La última página, /modal", (figura 3.12,3.13) calcula las frecuencias naturales de la estructura, así como los factores de amortiguamiento, a partir de una estimación de los valores. Por último, muestra los factores de amortiguamiento y frecuencia natural de cada señal, y de todas las señales promediadas.

### 3.7.2. Descripción del código de la aplicación

El código de la aplicación está desarrollado en python como lenguaje de programación, pero incluye asimismo elementos *html* y *css* para poder visualizar la página. Por otra parte, requiere una estructura de carpetas específica, que está prefijada por el propio Flask. A continuación se describe la estructura de las carpetas.

- static: la carpeta incluye el estilo o diseño de la página web, así como todos los documentos subidos a la web y las bases de datos generadas por la aplicación.
  - db_frecuencia: carpeta donde se guardan las bases de datos de señales en frecuencia.
  - db_tiempo: carpeta donde se almanenan las bases de datos de señales temporales.
  - upload_xlsx: carpeta donde se guardan los archivos subidos a la web y se almacenan de forma indefininda.
  - style.css: archivo de css para modificar el diseño de la pagina web.

Figura 3.8: Sección de tratamiento de señal


Figura 3.9: Modificación de los parámetros para obtener la frf

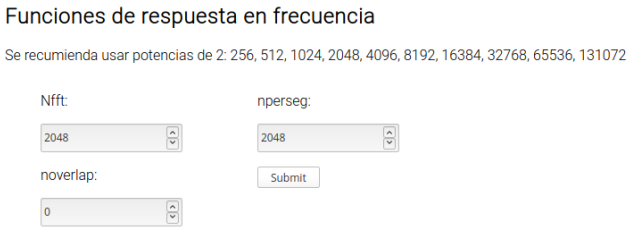



Figura 3.10: Diagrama bode de movilidad

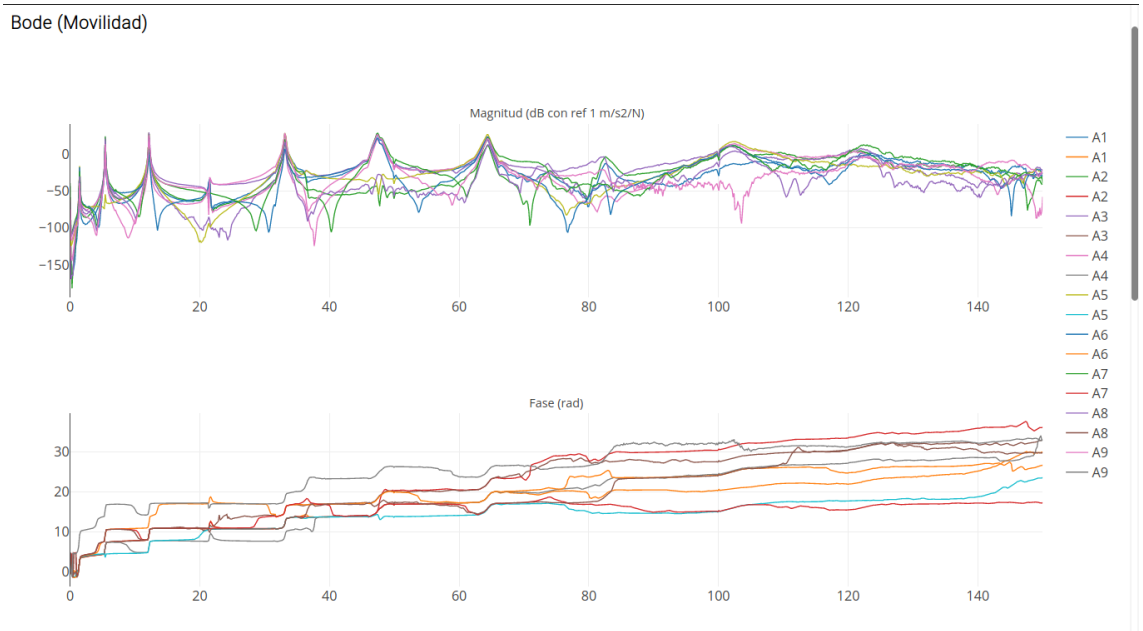


Figura 3.11: Diagrama Nyquist de movilidad

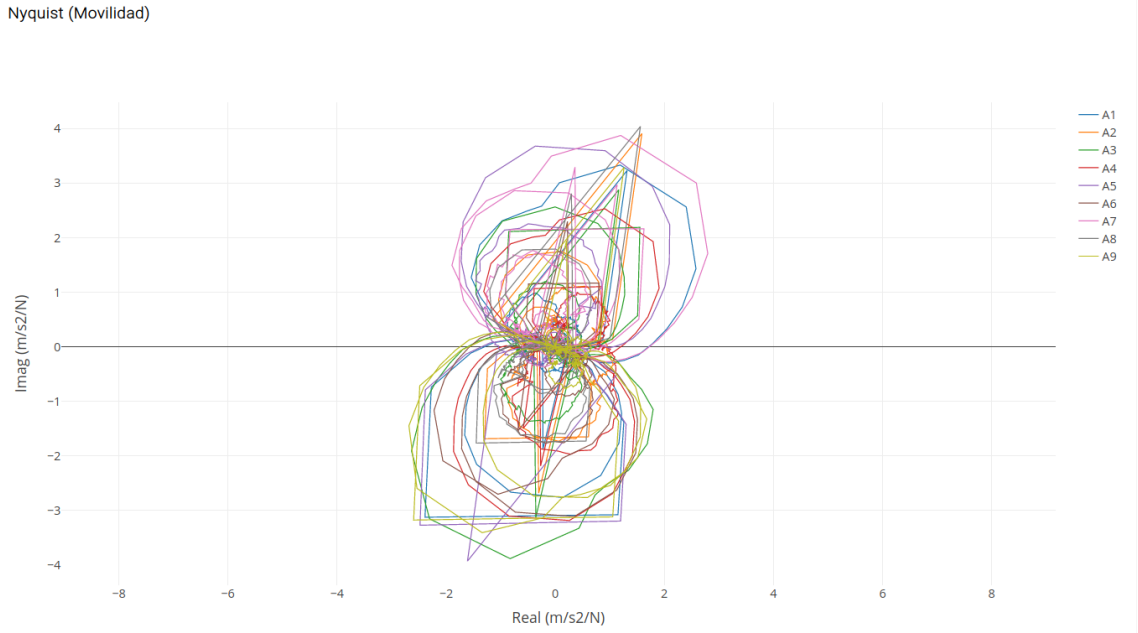
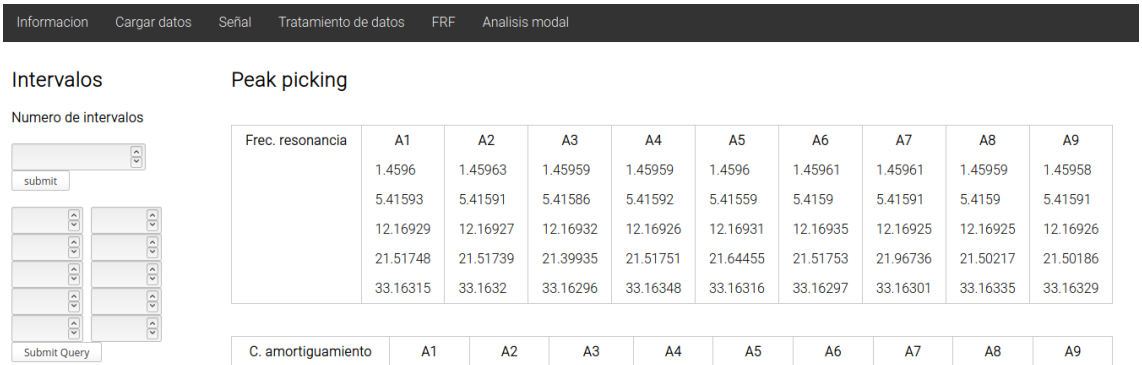



Figura 3.12: Introducción a la aplicación




| Frec. resonancia | A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | A9 |
|---|---|---|---|---|---|---|---|---|---|
| | 1.4596 | 1.45963 | 1.45959 | 1.45959 | 1.4596 | 1.45961 | 1.45961 | 1.45959 | 1.45958 |
| | 5.41593 | 5.41591 | 5.41586 | 5.41592 | 5.41559 | 5.4159 | 5.41591 | 5.4159 | 5.41591 |
| | 12.16929 | 12.16927 | 12.16932 | 12.16926 | 12.16931 | 12.16935 | 12.16925 | 12.16925 | 12.16926 |
| | 21.51748 | 21.51739 | 21.39935 | 21.51751 | 21.64455 | 21.51753 | 21.96736 | 21.50217 | 21.50186 |
| | 33.16315 | 33.1632 | 33.16296 | 33.16348 | 33.16316 | 33.16297 | 33.16301 | 33.16335 | 33.16329 |

| C. amortiguamiento | A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | A9 |
|---|---|---|---|---|---|---|---|---|---|
| | 0.00753 | 0.00756 | 0.00754 | 0.00754 | 0.00754 | 0.00754 | 0.00754 | 0.00753 | 0.00752 |
| | 0.00348 | 0.00348 | 0.00349 | 0.00348 | 0.00347 | 0.00348 | 0.00348 | 0.00348 | 0.00348 |
| | 0.00304 | 0.00305 | 0.00305 | 0.00304 | 0.00304 | 0.00304 | 0.00305 | 0.00305 | 0.00305 |
| | 0.00416 | 0.00398 | -0.00066 | 0.00388 | 0.00489 | 0.00411 | 0.00896 | 0.00297 | 0.00296 |
| | 0.00384 | 0.00385 | 0.00386 | 0.00384 | 0.00385 | 0.00386 | 0.00385 | 0.00385 | 0.00384 |

Figura 3.13: Introducción a la aplicación

| Promedio | | | | | |
|---|---|---|---|---|---|
| Frec. Resonancia | 1.4596 | 5.41587 | 12.16928 | 21.56502 | 33.16317 |
| Coef. amortiguamiento | 0.00754 | 0.00348 | 0.00305 | 0.00391 | 0.00385 |

- templates: documento donde se incluyen las plantillas de html para cada página.
  - base.html: documento html base del que se nutren el resto de elementos. Define la barra de navegación, entre otras cosas.
  - cargar_datos.html
  - frf.html
  - info.html
  - modal.html
  - senal.html
  - tratamiento.html
- FlaskApp.py: aplicación principal, el código que se ejecuta en el servidor. A partir de este código se llaman a las librerías necesarias.
- amodal.py: librería de la que se nutre FlaskApp.py para calcular todos los valores necesarios.
- graficas.py: librería para la representación de gráficas dentro de la aplicación.
- preset.py: archivo con los valores por defecto necesarios para el funcionamiento de amodal.py.

### 3.7.3. Almacenamiento de datos

Todos los datos del servidor se almacenan en una base de datos, ya que por el funcionamiento de un servidor no se pueden guardar datos de forma global para todos los usuarios, ocuparía demasiada memoria en el caso de que se abriesen varias sesiones. Para solucionar este problema se ha elegido trabajar con hdf5, un tipo de base de datos jerárquica, en la que se pueden guardar vectores con una determinada clave. Para poder implementarla, se ha elegido la librería h5py, que permite crear, guardar y sobreescribir este tipo de bases de datos. Una ventaja de esta librería es que tiene una gran implementación con numpy, la librería que se ha estado usando para los vectores. Sin embargo, el principal inconveniente de la librería es su pobre documentación, que si bien puede ser suficiente para una persona experimentada, no expone ningún ejemplo ni casos de uso que sean funcionales.

En último lugar, la estrategia que se ha llevado a cabo para las bases de datos es guardar tantos datos como contenga la clase que se estudie, ayudando al servidor a no recalcular un gran número de datos. Esta estrategia no es siempre la correcta, ya que en este caso para algunas transformaciones es mejor obtener unos datos a partir de los que se pueda operar, que guardar cada variable de forma independiente. A partir de un rediseño se podría optimizar la aplicación para reducir notablemente los tiempos de carga. En todo caso, el objetivo del trabajo no se ha enfocado en la realización precisa de una aplicación online, sino que se ha preferido hacer una herramienta enfocada a la escritura de programas simples.

# Capítulo 4

# Aplicaciones y usos.

## 4.1. Introducción

En el siguiente apartado se comprueban los resultados obtenidos por el software realizado. En un análisis modal es importante caracterizar correctamente la pieza con su peso, masa, disposición de acelerómetros, para posteriormente comprobarlo con otro análisis de elementos finitos. En este trabajo se obviarán la mayor parte de esos elementos para centrase en las señales obtenidas de los mismos, comentando sus particularidades y comprobando si el programa puede arrojar datos erróneos dependiendo del tipo de señal, contemplando la robustez del mismo.

## 4.2. Cadena de medida usada

En este tipo de ensayos es primordial conocer y entender el equipo adecuado, ya que hay diferentes elementos para cada situación. En este resumen, se repasa el equipo usado en los ensayos:

- Acelerómetros (figura 4.1): recogen datos de aceleración, y se fijan con adhesivo o una base magnética a la superficie de la estructura.
- Célula de carga (figura 4.2): dispositivo que recoge la fuerza ejercida sobre el mismo en función del tiempo. Existen diferentes dispositivos para este tipo de ensayos, muchas veces se integra dentro del propio martillo de impactos. En este caso es un componente separado. Una desventaja de este elemento frente a otros elementos comerciales es su gran recorrido antes del impacto. A partir de la gráfica fuerza contra tiempo, se puede observar una falda en torno al propio impacto, por lo que al trocear la señal según los impactos, no se puede tener claro el primer punto de impacto.
- Martillo de impactos: es un martillo sin ningún dispositivo electrónico indicado, de un tamaño acorde a las piezas de gran tamaño que se han usado.
- Tarjeta de adquisición de datos (figura 4.3): dispositivo que recoge las señales directamente de los acelerómetros, que se compone de filtros anti *aliasing* y los circuitos ADC o *Analog to Digital Converter*. El modelo específico de la tarjeta es la SIRIUS HD-SGT, y cuenta con las siguientes salidas y entradas:

Figura 4.1: Acelerómetro [11]


- Entrada de alimentación.
- 16 entradas tipo D-SUB, usadas para conectar los acelerómetros.
- Salida USB para transmitir los datos al ordenador.

- Programa de toma de datos: se ha usado DEWESoftX2 como el programa para recoger las muestras. Aunque el trabajo se haya enfocado a funciones de transferencia simples, de estructuras unidimensionales, el programa da soporte para configuraciones más avanzadas.

## 4.3. Viga de madera

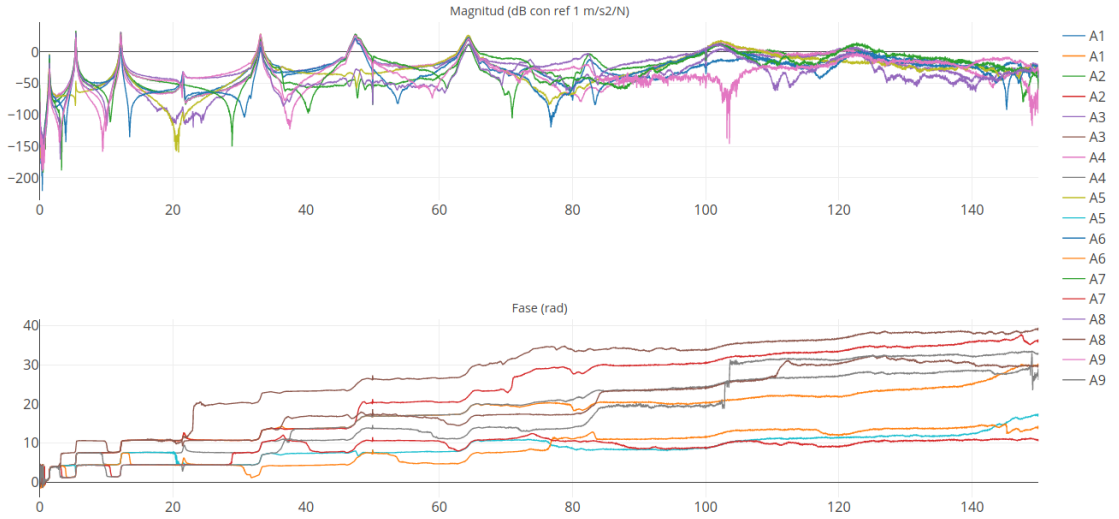
Los datos han sido tomados con una viga de madera, y los acelerómetros dispuestos de forma equiespaciada, lo que facilita que las señales entre todos los elementos sean más parecidas, ya que los acelerómetros no estarán en los valles generados en la vibración. Gracias a esto hay menos modos no detectados, y los resultados son más homogéneos. La función de respuesta en frecuencia de los resultados se observa en la figura 4.4

Por otra parte, se pueden ver los resultados favorables de añadir *zero padding*, es decir, añadir a la señal n elementos con el valor cero. Gracias a esto podemos obtener más resolución de la transformada de fourier (la resolución depende de los elementos del vector inicial). Sin embargo, no es una interpolación, sino que los datos de amortiguamiento de *peak picking* y *circle fit* convergen hacia un valor, más cercano en principio del valor real. Curiosamente, el método de *peak picking* obtiene unos valores más correctos del amortiguamiento en la mayoría de los casos, aunque con una mejor estimación de los modos, este valor puede mejorar bastante. Por otra parte, el método *circle fit* permite mayor precisión en las frecuencias naturales, ya que interpola entre varios valores. No se debe olvidar que es una interpolación lineal, y no llega a ser un valor preciso obtenido por un ajuste de curvas.

Figura 4.2: Célula [11]


Figura 4.3: Tarjeta de adquisición de datos[11]

Figura 4.4: Gráfica bode de movilidad, donde el eje x representa la frecuencia en Hz




Cuadro 4.1: Resultados de frecuencia para la viga de madera con nfft = 16384 y nperseg = 16384, usando *circle fit*

| | Modo 1 | Modo 2 | Modo 3 | Modo 4 | Modo 5 |
|---|---|---|---|---|---|
| Acelerometro 1 | 1.4572 | 5.41494 | 12.16824 | 21.50377 | 33.14582 |
| Acelerometro 2 | 1.45717 | 5.41493 | 12.16823 | 21.50444 | 33.14345 |
| Acelerometro 3 | 1.45717 | 5.41488 | 12.16826 | 23.03503 | 33.14764 |
| Acelerometro 4 | 1.45718 | 5.41492 | 12.16824 | 21.50419 | 33.11489 |
| Acelerometro 5 | 1.45719 | 0.0 | 12.16824 | 0.0 | 33.14392 |
| Acelerometro 6 | 1.45719 | 5.41493 | 12.16822 | 21.50382 | 33.14473 |
| Acelerometro 7 | 1.45718 | 5.41493 | 12.16825 | 0.0 | 33.14676 |
| Acelerometro 8 | 1.45719 | 5.41492 | 12.16821 | 21.50479 | 33.13769 |
| Acelerometro 9 | 1.4572 | 5.41494 | 12.16825 | 21.50475 | 33.16375 |
| Promedio | 1.45719 | 5.41492 | 12.16824 | 21.72297 | 33.14318 |

Cuadro 4.2: Resultados de frecuencia para la viga de madera con nfft = 32768 y nperseg = 16384, usando *circle fit*

| | Modo 1 | Modo 2 | Modo 3 | Modo 4 | Modo 5 |
|---|---|---|---|---|---|
| Acelerometro 1 | 1.4596 | 5.41591 | 12.16929 | 21.50422 | 33.16315 |
| Acelerometro 2 | 1.45963 | 5.41589 | 12.16927 | 21.50395 | 33.1632 |
| Acelerometro 3 | 1.45959 | 5.41584 | 12.16932 | 23.01758 | 33.16296 |
| Acelerometro 4 | 1.45959 | 5.41589 | 12.16926 | 21.51745 | 33.16348 |
| Acelerometro 5 | 1.4596 | 0.0 | 12.16931 | 0.0 | 33.16316 |
| Acelerometro 6 | 1.45961 | 5.41589 | 12.16935 | 21.50439 | 33.16297 |
| Acelerometro 7 | 1.45961 | 5.41589 | 12.16925 | 0.0 | 33.16301 |
| Acelerometro 8 | 1.45959 | 5.41589 | 12.16925 | 21.50237 | 33.16335 |
| Acelerometro 9 | 1.45958 | 5.41589 | 12.16926 | 21.48714 | 33.16329 |
| Promedio | 1.4596 | 5.41589 | 12.16928 | 21.71959 | 33.16317 |

Cuadro 4.3: Resultados de frecuencia para la viga de madera con nfft = 16384 y nperseg = 16384, usando *peak picking*

| | Modo 1 | Modo 2 | Modo 3 | Modo 4 | Modo 5 |
|---|---|---|---|---|---|
| Acelerometro 1 | 1.46484 | 5.41992 | 12.17651 | 21.49658 | 33.14209 |
| Acelerometro 2 | 1.46484 | 5.41992 | 12.17651 | 21.49658 | 33.14209 |
| Acelerometro 3 | 1.46484 | 5.41992 | 12.17651 | 21.44165 | 33.1604 |
| Acelerometro 4 | 1.46484 | 5.41992 | 12.17651 | 21.49658 | 33.14209 |
| Acelerometro 5 | 1.46484 | 5.41992 | 12.17651 | 0.0 | 33.14209 |
| Acelerometro 6 | 1.46484 | 5.41992 | 12.17651 | 21.49658 | 33.1604 |
| Acelerometro 7 | 1.46484 | 5.41992 | 12.17651 | 0.0 | 33.1604 |
| Acelerometro 8 | 1.46484 | 5.41992 | 12.17651 | 21.5332 | 33.14209 |
| Acelerometro 9 | 1.46484 | 5.41992 | 12.17651 | 21.5332 | 33.14209 |
| Promedio | 1.46484 | 5.41992 | 12.17651 | 21.4992 | 33.14819 |

Cuadro 4.4: Resultados de frecuencia para la viga de madera con nfft = 32768 y nperseg = 16384, usando *peak picking*

| | Modo 1 | Modo 2 | Modo 3 | Modo 4 | Modo 5 |
|---|---|---|---|---|---|
| Acelerometro 1 | 1.46484 | 5.41992 | 12.16736 | 21.49658 | 33.14209 |
| Acelerometro 2 | 1.46484 | 5.41992 | 12.16736 | 21.49658 | 33.14209 |
| Acelerometro 3 | 1.46484 | 5.41992 | 12.16736 | 21.45081 | 33.16956 |
| Acelerometro 4 | 1.46484 | 5.41992 | 12.16736 | 21.49658 | 33.14209 |
| Acelerometro 5 | 1.46484 | 5.41992 | 12.16736 | 0.0 | 33.14209 |
| Acelerometro 6 | 1.46484 | 5.41992 | 12.16736 | 21.49658 | 33.16956 |
| Acelerometro 7 | 1.46484 | 5.41992 | 12.16736 | 0.0 | 33.16956 |
| Acelerometro 8 | 1.46484 | 5.41992 | 12.16736 | 21.5332 | 33.14209 |
| Acelerometro 9 | 1.46484 | 5.41992 | 12.16736 | 21.5332 | 33.14209 |
| Promedio | 1.46484 | 5.41992 | 12.16736 | 21.50051 | 33.15125 |

Cuadro 4.5: Resultados de amortiguamiento para la viga de madera con nfft = 16384 y nperseg = 16384, usando *circle fit*

| | Modo 1 | Modo 2 | Modo 3 | Modo 4 | Modo 5 |
|---|---|---|---|---|---|
| Acelerometro 1 | 0.00737 | 0.00354 | 0.00311 | 0.00298 | 0.00419 |
| Acelerometro 2 | 0.00742 | 0.00354 | 0.00311 | 0.00302 | 0.00418 |
| Acelerometro 3 | 0.00739 | 0.00355 | 0.00311 | 0.00076 | 0.00422 |
| Acelerometro 4 | 0.00737 | 0.00355 | 0.00311 | 0.00302 | 0.00425 |
| Acelerometro 5 | 0.00738 | 0.0 | 0.00311 | 0.0 | 0.00419 |
| Acelerometro 6 | 0.00737 | 0.00354 | 0.00311 | 0.00301 | 0.00419 |
| Acelerometro 7 | 0.00738 | 0.00354 | 0.00311 | 0.0 | 0.0042 |
| Acelerometro 8 | 0.00737 | 0.00354 | 0.00312 | 0.00308 | 0.00383 |
| Acelerometro 9 | 0.00736 | 0.00354 | 0.00312 | 0.00305 | 0.0037 |
| Promedio | 0.00738 | 0.00354 | 0.00311 | 0.0027 | 0.00411 |

Cuadro 4.6: Resultados de amortiguamiento para la viga de madera con nfft = 32768 y nperseg = 16384, usando *circle fit*

| | Modo 1 | Modo 2 | Modo 3 | Modo 4 | Modo 5 |
|---|---|---|---|---|---|
| Acelerometro 1 | 0.00753 | 0.00347 | 0.00304 | 0.00293 | 0.00384 |
| Acelerometro 2 | 0.00756 | 0.00348 | 0.00305 | 0.00292 | 0.00385 |
| Acelerometro 3 | 0.00754 | 0.00349 | 0.00305 | 0.00242 | 0.00386 |
| Acelerometro 4 | 0.00754 | 0.00348 | 0.00304 | 0.00393 | 0.00384 |
| Acelerometro 5 | 0.00754 | 0.0 | 0.00304 | 0.0 | 0.00385 |
| Acelerometro 6 | 0.00754 | 0.00347 | 0.00304 | 0.00296 | 0.00386 |
| Acelerometro 7 | 0.00754 | 0.00347 | 0.00305 | 0.0 | 0.00385 |
| Acelerometro 8 | 0.00753 | 0.00347 | 0.00305 | 0.00303 | 0.00385 |
| Acelerometro 9 | 0.00752 | 0.00347 | 0.00305 | 0.00371 | 0.00384 |
| Promedio | 0.00754 | 0.00348 | 0.00305 | 0.00313 | 0.00385 |

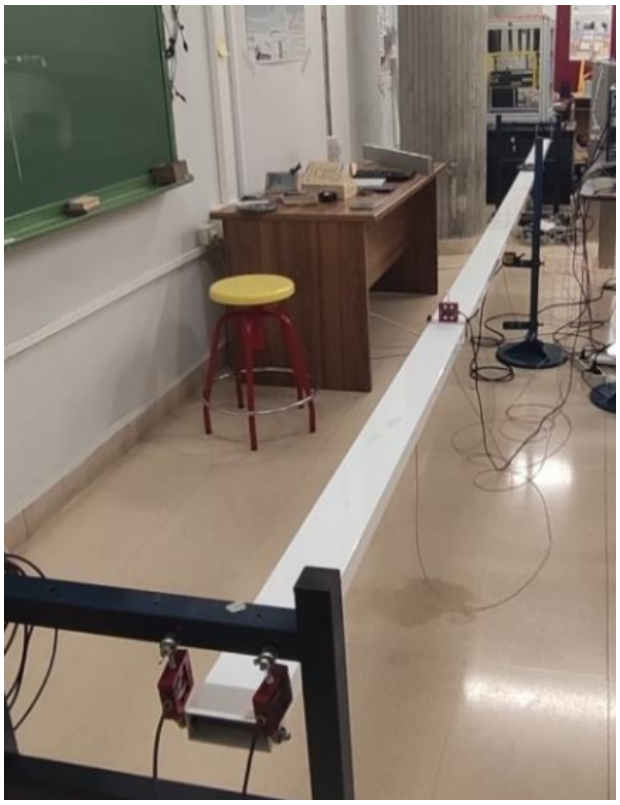
Cuadro 4.7: Resultados de amortiguamiento para la viga de madera con nfft = 16384 y nperseg = 16384, usando *peak picking*

| | Modo 1 | Modo 2 | Modo 3 | Modo 4 | Modo 5 |
|---|---|---|---|---|---|
| Acelerometro 1 | 0.00965 | 0.00386 | 0.00329 | 0.00349 | 0.00371 |
| Acelerometro 2 | 0.00965 | 0.00386 | 0.00329 | 0.00349 | 0.00373 |
| Acelerometro 3 | 0.00968 | 0.00387 | 0.00329 | 0.0345 | 0.00374 |
| Acelerometro 4 | 0.0097 | 0.00387 | 0.00327 | 0.00348 | 0.00379 |
| Acelerometro 5 | 0.00971 | 0.00441 | 0.00329 | 0.0 | 0.00372 |
| Acelerometro 6 | 0.00974 | 0.00384 | 0.00334 | 0.00359 | 0.00376 |
| Acelerometro 7 | 0.00976 | 0.00385 | 0.00325 | 0.0 | 0.00371 |
| Acelerometro 8 | 0.00974 | 0.00385 | 0.00326 | 0.00906 | 0.00385 |
| Acelerometro 9 | 0.0098 | 0.00385 | 0.00326 | 0.00733 | 0.0038 |
| Promedio | 0.00971 | 0.00392 | 0.00328 | 0.00928 | 0.00376 |

Cuadro 4.8: Resultados de amortiguamiento para la viga de madera con nfft = 32768 y nperseg = 16384, usando *peak picking*

| | Modo 1 | Modo 2 | Modo 3 | Modo 4 | Modo 5 |
|---|---|---|---|---|---|
| Acelerometro 1 | 0.00884 | 0.00376 | 0.00302 | 0.00342 | 0.00366 |
| Acelerometro 2 | 0.00885 | 0.00376 | 0.00302 | 0.00342 | 0.00364 |
| Acelerometro 3 | 0.00886 | 0.00377 | 0.00302 | 0.02642 | 0.00377 |
| Acelerometro 4 | 0.00887 | 0.00377 | 0.00303 | 0.00341 | 0.00361 |
| Acelerometro 5 | 0.00888 | 0.00439 | 0.00302 | 0.0 | 0.00365 |
| Acelerometro 6 | 0.00892 | 0.00375 | 0.00302 | 0.00344 | 0.00373 |
| Acelerometro 7 | 0.00894 | 0.00376 | 0.00304 | 0.0 | 0.00375 |
| Acelerometro 8 | 0.00893 | 0.00376 | 0.00304 | 0.00848 | 0.00379 |
| Acelerometro 9 | 0.00895 | 0.00376 | 0.00304 | 0.00683 | 0.00366 |
| Promedio | 0.00889 | 0.00383 | 0.00303 | 0.00792 | 0.0037 |

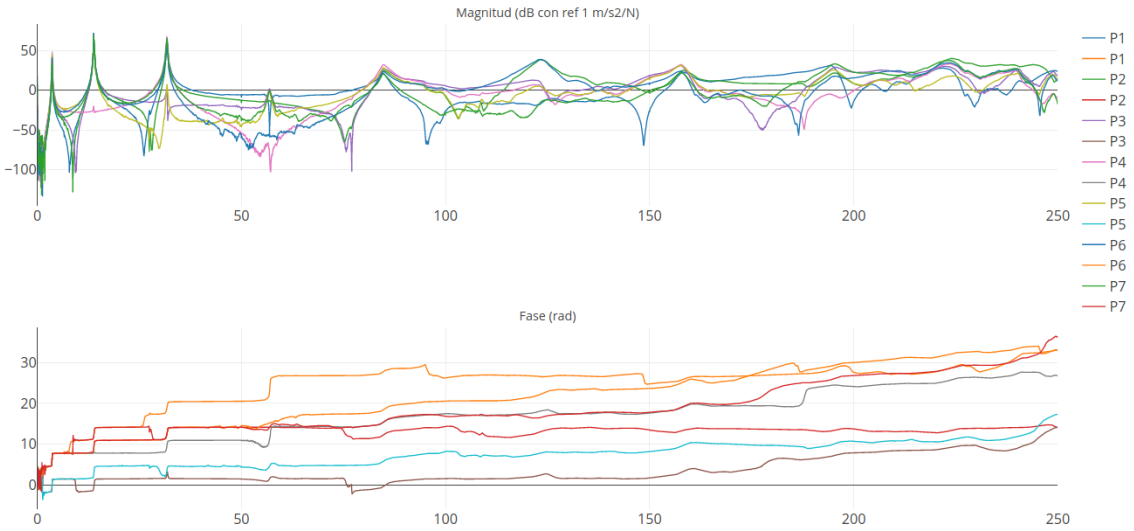
Figura 4.5: Viga de aluminio usada en el ensayo. [11]



## 4.4. Viga de aluminio

Los datos de la viga de aluminio (figura 4.5) se tomaron con 7 acelerómetros, dispuestos en los puntos de mayor amplitud de determinados modos. Gracias a esto, las señales de los picos se captan con mayor precisión, siempre que el acelerómetro sea capaz de detectar esas medidas. Sin embargo, los puntos de mayor amplitud de un modo muchas veces significan que es el punto de menor amplitud de otros modos. Por estas peculiaridades, en la señal se observa que ciertos modos no forman un pico, mientras que otros sí. En los resultados del *peak picking* se observa que, para el modo 4, no se detecta el pico ni el amortiguamiento en las señales de 4 acelerómetros. Sin embargo, el *circle fit* es capaz de detectar la mayoría de los modos. Al tener señales más débiles no es suficiente para tener una estimación correcta del modo. En ambos casos también se ha obtenido unas funciones de respuesta en frecuencia relativamente claras, sin ruido ni perturbaciones. La causa del ruido en estos casos suele ser por el proceso de toma de datos. Sin embargo también influye el punto por el que se ha cortado la señal durante el tratamiento de datos. Esto produce generalmente unas aberraciones constantes en el espectro.

Figura 4.6: Gráfica bode de movilidad, donde el eje x representa la frecuencia en Hz




Cuadro 4.9: Resultados de frecuencia para la viga de aluminio con nfft = 16384 y nperseg = 16384, usando *circle fit*

| | Modo 1 | Modo 2 | Modo 3 | Modo 4 | Modo 5 |
|---|---|---|---|---|---|
| Acelerometro 1 | 3.61927 | 13.82091 | 31.75245 | 56.87647 | 84.62462 |
| Acelerometro 2 | 3.61938 | 13.82087 | 31.75256 | 56.90798 | 84.80684 |
| Acelerometro 3 | 3.61925 | 13.82089 | 31.75297 | 56.8779 | 84.79907 |
| Acelerometro 4 | 3.61921 | 13.81614 | 31.75237 | 56.84182 | 84.7968 |
| Acelerometro 5 | 3.61921 | 13.82084 | 31.74978 | 56.87988 | 84.75082 |
| Acelerometro 6 | 3.61921 | 13.82086 | 31.75244 | 0.0 | 84.6456 |
| Acelerometro 7 | 3.61918 | 13.82086 | 31.75238 | 56.93349 | 84.83014 |
| Promedio | 3.61924 | 13.8202 | 31.75213 | 56.88626 | 84.75056 |

Cuadro 4.10: Resultados de frecuencia para la viga de aluminio con nfft = 32768 y nperseg = 16384, usando *circle fit*

| | Modo 1 | Modo 2 | Modo 3 | Modo 4 | Modo 5 |
|---|---|---|---|---|---|
| Acelerometro 1 | 3.61968 | 13.82339 | 31.75719 | 56.87989 | 84.86844 |
| Acelerometro 2 | 3.61979 | 13.8234 | 31.75743 | 56.88192 | 84.52744 |
| Acelerometro 3 | 3.61961 | 13.82338 | 31.76221 | 56.88006 | 84.86566 |
| Acelerometro 4 | 3.61952 | 13.81925 | 31.7568 | 57.11626 | 84.86384 |
| Acelerometro 5 | 3.61958 | 13.82335 | 31.75444 | 56.87985 | 84.52718 |
| Acelerometro 6 | 3.61959 | 13.82336 | 31.75729 | 0.0 | 84.52833 |
| Acelerometro 7 | 3.61957 | 13.82335 | 31.75686 | 56.88006 | 84.52673 |
| Promedio | 3.61962 | 13.82278 | 31.75746 | 56.91967 | 84.67252 |

Cuadro 4.11: Resultados de frecuencia para la viga de aluminio con nfft = 16384 y nperseg = 16384, usando *peak picking*

| | Modo 1 | Modo 2 | Modo 3 | Modo 4 | Modo 5 |
|---|---|---|---|---|---|
| Acelerometro 1 | 3.63159 | 13.82446 | 31.7688 | 0.0 | 84.65576 |
| Acelerometro 2 | 3.63159 | 13.82446 | 31.7688 | 0.0 | 84.86938 |
| Acelerometro 3 | 3.63159 | 13.82446 | 31.70776 | 56.85425 | 84.86938 |
| Acelerometro 4 | 3.63159 | 13.79395 | 31.73828 | 0.0 | 84.86938 |
| Acelerometro 5 | 3.63159 | 13.82446 | 31.7688 | 56.85425 | 84.86938 |
| Acelerometro 6 | 3.63159 | 13.82446 | 31.73828 | 0.0 | 84.86938 |
| Acelerometro 7 | 3.63159 | 13.82446 | 31.73828 | 56.88477 | 84.86938 |
| Promedio | 3.63159 | 13.8201 | 31.747 | 56.86442 | 84.83887 |

Cuadro 4.12: Resultados de frecuencia para la viga de aluminio con nfft = 32768 y nperseg = 16384, usando *peak picking*

| | Modo 1 | Modo 2 | Modo 3 | Modo 4 | Modo 5 |
|---|---|---|---|---|---|
| Acelerometro 1 | 3.61633 | 13.82446 | 31.75354 | 0.0 | 84.65576 |
| Acelerometro 2 | 3.61633 | 13.82446 | 31.75354 | 0.0 | 84.86938 |
| Acelerometro 3 | 3.61633 | 13.82446 | 31.6925 | 56.85425 | 84.86938 |
| Acelerometro 4 | 3.61633 | 13.8092 | 31.75354 | 0.0 | 84.86938 |
| Acelerometro 5 | 3.61633 | 13.82446 | 31.7688 | 56.85425 | 84.79309 |
| Acelerometro 6 | 3.61633 | 13.82446 | 31.75354 | 0.0 | 84.86938 |
| Acelerometro 7 | 3.61633 | 13.82446 | 31.75354 | 56.90002 | 84.86938 |
| Promedio | 3.61633 | 13.82228 | 31.747 | 56.86951 | 84.82797 |

Cuadro 4.13: Resultados de amortiguamiento para la viga de aluminio con nfft = 16384 y nperseg = 16384, usando *circle fit*

| | Modo 1 | Modo 2 | Modo 3 | Modo 4 | Modo 5 |
|---|---|---|---|---|---|
| Acelerometro 1 | 0.00928 | 0.00342 | 0.0034 | 0.006 | 0.01395 |
| Acelerometro 2 | 0.00928 | 0.00342 | 0.00341 | 0.00509 | 0.01329 |
| Acelerometro 3 | 0.00929 | 0.00342 | 0.00334 | 0.00601 | 0.0129 |
| Acelerometro 4 | 0.0093 | 0.00391 | 0.00341 | 0.01086 | 0.01286 |
| Acelerometro 5 | 0.0093 | 0.00342 | 0.00355 | 0.00608 | 0.01435 |
| Acelerometro 6 | 0.0093 | 0.00342 | 0.0034 | 0.0 | 0.01379 |
| Acelerometro 7 | 0.00931 | 0.00342 | 0.00341 | 0.00638 | 0.01348 |
| Promedio | 0.00929 | 0.00349 | 0.00342 | 0.00674 | 0.01352 |

Cuadro 4.14: Resultados de amortiguamiento para la viga de aluminio con nfft = 32768 y nperseg = 16384, usando *circle fit*

| | Modo 1 | Modo 2 | Modo 3 | Modo 4 | Modo 5 |
|---|---|---|---|---|---|
| Acelerometro 1 | 0.01 | 0.00296 | 0.0034 | 0.00598 | 0.0125 |
| Acelerometro 2 | 0.01 | 0.00296 | 0.00341 | 0.0011 | 0.01381 |
| Acelerometro 3 | 0.01001 | 0.00296 | 0.00332 | 0.00611 | 0.01254 |
| Acelerometro 4 | 0.01002 | 0.00318 | 0.00341 | 0.0034 | 0.01265 |
| Acelerometro 5 | 0.01003 | 0.00296 | 0.00354 | 0.00679 | 0.01454 |
| Acelerometro 6 | 0.01002 | 0.00296 | 0.0034 | 0.0 | 0.01439 |
| Acelerometro 7 | 0.01003 | 0.00296 | 0.00341 | 0.00634 | 0.01333 |
| Promedio | 0.01001 | 0.00299 | 0.00341 | 0.00495 | 0.01339 |

Cuadro 4.15: Resultados de amortiguamiento para la viga de aluminio con nfft = 16384 y nperseg = 16384, usando *peak picking*

| | Modo 1 | Modo 2 | Modo 3 | Modo 4 | Modo 5 |
|---|---|---|---|---|---|
| Acelerometro 1 | 0.01148 | 0.00237 | 0.00355 | 0.0 | 0.01405 |
| Acelerometro 2 | 0.01146 | 0.00237 | 0.00353 | 0.0 | 0.01678 |
| Acelerometro 3 | 0.01149 | 0.00237 | 0.00578 | 0.00692 | 0.01734 |
| Acelerometro 4 | 0.01146 | 0.03793 | 0.00359 | 0.0 | 0.01483 |
| Acelerometro 5 | 0.01143 | 0.00238 | 0.00361 | 0.0063 | 0.01522 |
| Acelerometro 6 | 0.01143 | 0.00238 | 0.00358 | 0.0 | 0.01591 |
| Acelerometro 7 | 0.01146 | 0.00238 | 0.00359 | 0.00689 | 0.01376 |
| Promedio | 0.01146 | 0.00745 | 0.00389 | 0.0067 | 0.01541 |

Cuadro 4.16: Resultados de amortiguamiento para la viga de aluminio con nfft = 32768 y nperseg = 16384, usando *peak picking*

| | Modo 1 | Modo 2 | Modo 3 | Modo 4 | Modo 5 |
|---|---|---|---|---|---|
| Acelerometro 1 | 0.01162 | 0.00234 | 0.00351 | 0.0 | 0.01405 |
| Acelerometro 2 | 0.01163 | 0.00234 | 0.0035 | 0.0 | 0.01683 |
| Acelerometro 3 | 0.01163 | 0.00234 | 0.0057 | 0.00692 | 0.01734 |
| Acelerometro 4 | 0.01162 | 0.05434 | 0.00355 | 0.0 | 0.01485 |
| Acelerometro 5 | 0.01161 | 0.00235 | 0.00359 | 0.00634 | 0.01533 |
| Acelerometro 6 | 0.0116 | 0.00235 | 0.00354 | 0.0 | 0.016 |
| Acelerometro 7 | 0.01163 | 0.00235 | 0.00354 | 0.00667 | 0.01376 |
| Promedio | 0.01162 | 0.00977 | 0.00385 | 0.00664 | 0.01545 |

# Capítulo 5

# Conclusiones y líneas futuras.

## 5.1. Conclusiones

El código final se compone de más de 1.400 líneas de código de python, sin incluir el *html* y *css* necesario para la implementación web. Tras la primera estimación realizada a comienzos del trabajo, se ha superado el tiempo de trabajo esperado, principalmente por la falta de conocimiento del lenguaje y las herramientas con las que se ha estado trabajando. La mayoría de las funciones se han reescrito varias veces, a medida que se implementaban las diferentes funcionalidades de python. Tras el tiempo invertido, las expectativas se han superado en determinados aspectos, mientras que otros objetivos han quedado relegados a segundo plano o como líneas futuras.

El primer objetivo, la realización de un programa capaz de leer y transformar las señales temporales, se ha cumplido satisfactoriamente. Se han incluido las funciones necesarias para un preprocesamiento básico, cálculo de estadísticos y una solución adecuada para realizar graficas de los mismos. Además, se ha superado el objetivo inicial incluyendo una función para adaptar las señales, recortándolas, a un ensayo con varios impactos. No caben puntos de mejora en este aspecto, más que un periodo de prueba para comprobar que efectivamente se adapta a todo tipo de señales. La implementación en la web ha sido igualmente satisfactoria, cumple con la función, si bien en este caso hay un tiempo de espera excesivo al cargar los datos. Este error se debe a una mala elección al interaccionar con las bases de datos. Sin embargo, los aspectos técnicos escapan ampliamente del ámbito del trabajo.

El segundo objetivo, referente a incluir funciones robustas para el cálculo de las funciones de respuesta en frecuencia y los modos de una forma robusta, se ha cumplido, aunque con ciertas simplificaciones que no serían aptas para el público general, sino que se adaptan a las circunstancias del trabajo. No se ha calculado una matriz de funciones de respuesta en frecuencia. La funcionalidad es parcialmente robusta: los errores que se han encontrado se han mitigado con funciones para detectar esos mismos fallos, sin embargo, para otro tipo de señales con otras características, el programa puede arrojar resultados no esperados. El proceso de comprobar que funcionan correctamente en diferentes situaciones ha sido reducido por la falta de señales para comprobar la veracidad de las funciones. En todo caso las funciones para detectar errores se ejecutan de forma satisfactoria para los casos en los que se han podido comprobar. Se debe tener en cuenta que los métodos de detección de modos

implementados son estimaciones que pueden fallar ante entradas no esperadas.

El tercer objetivo es obtener una estructura sólida para poder escalar el programa a una aplicación más grande. En este campo se ha obtenido una estructura sólida de funciones, tanto temporales como de frecuencia, además de una clase adicionar para los datos relacionados con el análisis modal. Gracias a esta estructura se pueden implementar cómodamente funciones para aumentar la funcionalidad de las funciones de respuesta en frecuencia, incluir nuevos métodos de detección de modos... Sin embargo, no se ha definido una estructura para otros elementos que podrían ser interesantes. El ámbito de las vibraciones comprende una gran cantidad de algoritmos y métodos diferentes, que requieren variables diferentes. Si bien se ha procurado que las variables que se crearan siguiesen una estructura clara, no se han implementado estructuras adicionales para posibles funciones que no se han programado en el trabajo. Ya que es posible que el trabajo no sea continuado, ha sido una decisión adecuada no invertir el tiempo en esta funcionalidad. Por otra parte, las estructuras de datos corresponden a otras disciplinas con un conocimiento mayor sobre las herramientas disponibles.

El cuarto objetivo es obtener una precisión similar a programas actuales. Para los mismos métodos usados, se arrojan datos adecuados como estimación. Sin embargo, distan en gran medida de la realidad, y de otros métidos considerados más precisos. Por otra parte, los programas que usan estos mismos métidos permiten una mayor configuración de parámetros que se han mantenido fijos en este trabajo, por lo que las estimaciones de un programa profesional se podrán ajustar de una forma más precisa y gráfica. Sin embargo, para los medios disponibles, se ha logrado un resultado comparable a otros programas. En todo caso, las líneas futuras del programa se enfocarán a la inclusión de métodos más avanzados, en vez de mejorar la funcionalidad de los existentes.

Como consideraciones generales, el trabajo propuesto a inicios de febrero se ha cumplido, incluso superando las expectativas. Los objetivos que no se han complatado plenamente ha sido, principalmente, por el conocimiento limitado del campo y por que la mejor opción ha sido no ajustarse a ellos, sino intentar explorar vías nuevas. Todas las funcionalidades de la aplicación propuestas funcionan correctamente en los ensayos que se han realizado, por lo que la finalidad del trabajo también se ha cumplido. Además se ha incluido una implementación en una página web, de forma que se puede ejecutar desde cualquier dispositivo, incluso móvil.

Finalmente, gran parte de las horas de trabajo han ido dedicadas a aumentar las competencias técnicas de la carrera, muchas generales en campos más propios de una ingeniería electrónica o informática, pero que están conectados a la ingeniería mecánica.

## 5.2. Líneas futuras

A pesar de haber cumplido los objetivos del trabajo, e incluso haber hecho una aplicación con más características del plan inicial, la lista de líneas futuras ha ido aumentando a medida que se ha ido desarrollando el trabajo. Las líneas futuras podemos categorizarlas en diferentes apartados, uno enfocado a la mejora de la infraestructura informática, y el segundo dirigido hacia la funcionalidad del programa.

### 5.2.1. Mejora de la funcionalidad del programa

Con los estándares actuales de los programas de detección de modos y análisis de funciones de respuesta en frecuencia, la aplicación desarrollada tiene una gran cantidad de líneas de mejora. Si bien en el tratamiento de señal y obtención de estadísticos es una herramienta sólida y funcional, sería necesaria la integración de otros métodos de detección de modos.

#### Añadir nuevas funcionalidades para el tratamiento de datos

El apartado de tratamiento de datos cumple con los objetivos, sin embargo, se debe modificar para acercarse a un software más profesional y abierto a personalizaciones. Ahora mismo las opciones de las funciones son bastante limitadas: no permite modificar el tipo de filtro, ventanas, y no se permite cortar la señal.

#### Mejora de los algoritmos de detección actuales

La precisión de los algoritmos es aceptable dentro de los márgenes esperados, sin embargo sí que se deben añadir opciones para poder modificar los parámetros como se desee. Estos parámetros son principalmente el número de muestras que se escogen para cada tipo de transformación. Por otra parte, la precisión de los métodos se puede mejorar con el ajuste de curvas, aunque en este caso el esfuerzo no mejoraría notablemente los resultados.

#### Algoritmos de detección multimodo

Es imprescindible para un programa de análisis modal el uso de algoritmos multimodo. Debido al tiempo se ha decidido no implementarlos, aunque sí que se ha realizado un proceso de investigación sobre ellos. El primer método que se ha planteado implementar es el *curve fitting*, o ajuste de curvas [8][7]. El problema de un ajuste de curvas ya se ha planteado y resuelto en el circle fit, y no requiere mayor complejidad, ya que scipy cuenta con algoritmos avanzados para el ajuste de curvas.

El segundo método sobre el que se ha investigado es el análisis por redes neuronales de la señal original [6]. Se ha descartado por su gran complejidad, ya que seleccionar una red neuronal adecuada requiere un gran conocimiento sobre el tema. Sin embargo, una red neuronal puede emular de forma artificial todas las operaciones realizadas. Pese a no ser un método novedoso, sí que puede permitir el estudio de sistemas más complejos que no se pueden analizar por métodos multimodo.

### 5.2.2. Mejora de la infraestructura informática

Durante el desarrollo de la aplicación se han implementado todas las opciones que ofrece python para hacer que el código sea funcional y legible. Sin embargo, la aplicación tiene grandes fallos en comparación con otras librerías usadas, como numpy.

### Detección de errores

Un programa bien estructurado debe incorporar una serie de funciones para identificar un error ante una entrada inesperada, o incluso si el error es del propio programa. Por ello una posible línea futura es la prueba del programa ante entradas de datos inesperados, de forma que se pueda hacer un diagnóstico rápido.

### Aumentar compatibilidad en los formatos de entrada y salida

Actualmente, el programa puede leer y exportar en ficheros excel, siempre que esté en un formato determinado de filas y columnas. Por otra parte, también es capaz de leer y escribir datos en una base de datos hdf5. Sin embargo, se deberían explorar nuevas opciones de compatibilidad con otros programas de uso frecuente, como matlab, y también con otros programas diferntes de toma de datos.

### Modificar las funciones de visualización de datos

En python existen multitud de plataformas que permiten crear gráficas de una forma sencilla. Sin embargo, la opción escogida, plotly, tiene capacidades reducidas a la hora de realizar gráficas tipo bode, en las cuales deben aparecer dos cuadros paralelos, con ejes logarítmicos. Entre las principales alternativas podemos encontrar a matplotlib y a bokeh, que permite la integración en webs de una forma cómoda.

### Mejora de la aplicación web

Una aplicación web generamente requiere una estructura diferente al de un programa de escritorio, y en este caso la forma elegida para acceder a los datos no está optimizada. Sin embargo, estas mejoras del rendimiento web requerirían cambiar en gran forma la estructura del programa y de la propia web, por lo que requiere una gran inversión de tiempo.

## 5.3. Consideraciones adicionales

El desarrollo del trabajo se ha realizado totalmente en el ordenador, con breves visitas al laboratorio, y en el caso de los enayos, se han usado datos tomados por los tutores u otros alumnos para sus correspondientes trabajos. De esta forma, el impacto medioambiental sólo corresponde a las vigas de madera y aluminio, que se reparte entre trabajos de diferentes alumnos. No se considera, sin embargo, un impacto medioambiental positivo, si bien el fin final del estudio de las vibraciones es usar menos material de forma más eficiente, el trabajo no ha tenido ningún impacto directo en un medio real.

El análisis económico es más complejo. La labor del trabajo, en función de las líneas de código escrito, se podría suponer inferior a las propias horas que conforman un trabajo de fin de grado comparado a alumnos de ingeniería informática. Sin embargo, la mayor parte del trabajo ha consistido en la formación en las herramientas informáticas; como el *framework* Flask, bases de datos hdf5, las librerías científicas y el propio python; como la formación en aspectos básicos del tratamiento de señales

y cálculo de las funciones de transferencia. Además, el trabajo ha consistido en la realización de una librería de python, donde muchos de estos proyectos son software libre financiados por organizaciones y por donaciones de los propios usuarios: no siguen un modelo de negocio simple para poder estimar el beneficio generado. Recurriendo a un análisis de las horas trabajadas se realizan las siguientes estimaciones: se han dedicado 200 horas a formación en el lenguaje y los paquetes usados, además de un entendimiento más profundo de la informática; 50 horas para formación en los algoritmos de señales usados; la escritura del programa junto con la solución de errores han supuesto 300 horas, y finalmente se han dedicado 50 horas a la escritura de la memoria. El sueldo de un trabajador, partiendo de unos conocimientos muy básicos en la materia supondría un coste de 10,5 €a la hora: 6.300€por todo el desarrollo. Los beneficios obtenidos en la formación son muy importantes en el ámbito de la ingeniería; aunque sea difícil estimar el alcance real del conocimiento adquirido, podría suponer más de 500€que serán rentabilizados en la vida profesional.

El impacto del trabajo en la sociedad es indeterminado y depende del desarrollo de las líneas futuras. Es una herramienta útil para su uso diario en los ámbitos de la investigación y la enseñanza, se ha diseñado para ser una librería de alto nivel que requiera poco conocimiento de python para poder funcionar con algoritmos básicos. Finalmente, si se realizan determinadas modificaciones para aumentar las opciones a costa de disminuir la facilidad de uso, de forma que sea una librería de bajo nivel, se alcanzaría una librería de python de análisis modal pionera en su campo, ya que no hay ninguna alternativa en desarrollo que ofrezca varios sistemas de detección de modos.

# Apéndice A

# Manual de usuario

La librería amodal es una recopilación de herramientas y algoritmos para facilicitar el tratamiento de señales vibratorias, así como obtener las funciones de respuesta en frecuencia y las frecuencias naturales con sus respectivos factores de amortiguamiento. La librería se compone de dos archivos, amodal.py, como archivo principal dedicado al propio tratamiento de datos, y graficas.py, como librería para la visualización de datos.

## A.1. Default

Clase reservada a guardar y retornar valores por defecto. Dichos valores por defecto se guardan como un archivo de pyhton, preset.py, de forma que el programa pueda importarlos como una librería, y de esta forma guardar los valores por defecto. El modelo de un archivo por defecto se representa en el siguiente código. Se podrán incluir comentarios y variables o funciones adicionales, pero en caso de modificar los nombres de las variables del archivo, el programa dará errores.

```
#
#-----------------------------------------------
# Plantilla para guardar los valores por defecto
#-----------------------------------------------
#
# Esta plantilla se debe editar siguiendo las sintaxis de
# Python, cualquier grafia que no sea propia de Python
# resultara en un error de programa. De la misma forma,
# si se cambia el nombre de cualquiera de las variables
# resultara en un error de la funcion que dependa de dicha
# variable.
#

ventana_rms = 1
solape_rms = 0.5

incremento_vdv = 1

orden_filtro = 4
```

```
frecuencia_filtro = 200

samplerate_resample = 512

ventana_frf = 'hann'
nperseg_frf = 1024
noverlap_frf = 512
nfft_frf = 2048
```

### A.1.1. Default.get(_archivo = 'preset.py')

Función que comprueba si el archivo de valores por defecto es un archivo válido de python, y en caso de ser así lo guarda en la memoria de la clase, en la variable Default.archivo . En caso de que no sea válido, retorna el valor ImportError. Es importante denotar que es imprescindible usar esta función junto con Default.set() para poder observar los cambios realizados en los valores por defecto.

### A.1.2. Default.set()

Carga los valores por defecto del archivo seleccionado en Default.set() como una variable de la clase Default, por lo que los datos por defecto son accesibles de la siguiente forma:

```
import amodal as am
# con la siguiente funcion se guarda el archivo presets.py en memoria:
am.get('presets.py')
# comprobamos el nombre del archivo, la siguiente funcion retornara el valor 'prese
print(am.Default.archivo)
# importamos los valores por defecto:
am.Default.set()
# podemos acceder al valor de la variable con Default.nombredevariable:
print(am.Default.ventana_rms)
```

### A.1.3. Default.restore(_archivo = 'preset.py')

Crea un archivo válido de valores por defecto con el nombre indicado en _archivo, ya que es posible que se elimine el archivo de valores por defecto. Incluye un valor común para cada variable.

### A.1.4. Default.printsettings(_archivo = 'plantilla_preset.py')

Similar a la función anterior, crea un archivo válido de valores por defecto, con el nombreplantilla_preset.py. Sin embargo, en este caso los valores corresponden a aquellos que se han indicado al programa.

```
import amodal as am

am.get('presets.py')
```

```
am.Default.set()
# Se cambia el valor de una variable
am.Default.ventana_rms

# se guarda la variable en el archivo 'presets.py'
am.Default.printsettings('presets.py')
```

### A.1.5. Uso de Default en el programa

Al importar el fichero amodal.py python ejecuta automáticamente todo el archivo. Para asegurarse que el programa siempre tiene unos valores por defecto, se han introducido las siguientes llamadas a la clase Default:

```
if Default.get() == ImportError:
Default.restore()
Default.get()
Default.set()
```

Gracias a este código, si no hay ningún archivo con el nombre 'preset.py' en la carpeta en la que se encuentra el programa, python creará uno con ese mismo nombre, de forma que el programa siempre tendrá unos valores por defecto.

## A.2. Tiempo

### A.2.1. Tiempo.__init__()

```
def __init__(self, tiempo=None, unidad_tiempo=None, leyenda_tiempo=None,
aceleracion=None, unidad_aceleracion=None, leyenda_aceleracion=None,
fuerza=None, unidad_fuerza=None, leyenda_fuerza=None, rms=None,
rms_movil=None,  tiempo_rms_movil=None, mtvv=None, vdv=None,
tiempo_vdv=None, samplerate=None)
```

Declara las variables de la clase, ya sea con un valor (en caso de que sea indicado al llamar a la función) o con el nombre None. En caso de que sólamente se declaren ciertas variables, la función declarará el resto con su valor correcto, siempre que sea posible. Esta función se ejecuta cada vez que se atribuye esta clase a un objecto, de la siguiente forma:

```
import amodal as am
datos = am.Tiempo()
# Al haber declarado la funcion, el resultado del siguiente comando sera None
print(datos.aceleracion.v)
```

### A.2.2. Tiempo.comprobar_samplerate()

Suele ejecutarse por otras funciones del programa. Su función es declarar la variable Tiempo.samplerate, que recoge la frecuencia de muestreo de los datos.

### A.2.3. Tiempo.comprobar_datos()

A partir de los datos recogidos, esta función comprueba que tienen la misma longitud. Sin embargo, como se comenta en el capítulo 5, sería necesario aumentar su funcionalidad para diagnosticar correctamente los problemas. La función no retorna ningún valor.

### A.2.4. Tiempo.lectura()

```
lectura(self, archivo, numero_hoja=1)
```

La función lectura se dedica exclusivamente a cargar datos de una hoja de cálculo excel (.xslx) en la función de tiempo. Para ello se debe indicar la ruta completa del archivo, de la forma 'ruta/hacia/el/archivo/archivo.xslx' o bien 'archivo.xslx' en caso de que esté en la misma carpeta que el programa. Por defecto lee la hoja 1, valor que se puede modificar. Una vez la función lee los datos, calcula determinados parámetros y variables internas.

```
import amodal as am

# En primer lugar se debe asignar la clase
datos = am.Tiempo()
# Una vez se ha iniciado la clase, se puede leer el archivo
datos.lectura('archivo.xlsx')
```

### A.2.5. Tiempo.escritura()

```
escritura(self, ruta, nombre_archivo)
```

La función escritura escribe los datos en una hoja de excel, en el mismo formato que lee la función de lectura. en este caso no toma la variable hoja, se guarda en la primera hoja.

```
import amodal as am

datos = am.Tiempo()
datos.lectura('archivo.xlsx')

datos.escritura('/','nuevoarchivo.xslx')
```

### A.2.6. Tiempo.get_h5py()

```
get_h5py(self,filepath)
```

Produce como resultado una base de datos tipo hdf5, en la ruta que se indique en filepath. La base de datos seguirá la estructura del cuadro A.1. En el apéndice A.2.7 se indica un ejemplo de código funcional.

### A.2.7. Tiempo.from_h5py()

```
from_h5py(self, filepath)
```

Cuadro A.1: Estructura de la base de datos

| Referencia del dataset | Variable que contiene |
|---|---|
| tiempo.tiempo.v | self.tiempo.v |
| tiempo.tiempo.u | self.tiempo.u |
| tiempo.tiempo.l | self.tiempo.l |
| tiempo.aceleracion.v | self.aceleracion.v |
| tiempo.aceleracion.u | self.aceleracion.u |
| tiempo.aceleracion.l | self.aceleracion.l |
| tiempo.fuerza.v | self.fuerza.v |
| tiempo.fuerza.u | self.fuerza.u |
| tiempo.fuerza.l | self.fuerza.l |
| tiempo.rms.v | self.rms.v |
| tiempo.rms_movil.v | self.rms_movil.v |
| tiempo.rms_movil_tiempo.v | self.tiempo_rms_movil.v |
| tiempo.mtvv | self.mtvv.v |
| tiempo.vdv.v | self.vdv.v |
| tiempo.vdv_tiempo.v | self.vdv.v |

Recupera los datos guardados en la base de datos tipo hdf5 guardada en la ruta filepath. La base de datos tiene que seguir la estructura del cuadro A.1. Un ejemplo de código utilizando esta función sería el siguiente:

```
import amodal as am

datos = am.Tiempo()
datos.lectura('archivo.xlsx')

# Guardamos los datos en la base de datos
datos.get_h5py('db_senal')

# Se recuperan los datos
datos.from_h5py('db_senal')
```

### A.2.8. Tiempo.get_rms()

```
get_rms(self)
```

Esta funcion guarda en la variable Tiempo.rms.v un vector unidimensional, con una longitud igual al número de señales, con el RMS de cada señal.

```
import amodal as am

datos = am.Tiempo()
datos.lectura('archivo.xlsx')

datos.get_rms()
```

```
print(datos.rms.v)
```

### A.2.9. Tiempo.get_rms_movil()

```
get_rms_movil(self)
```

Obtiene como resultado dos matrices, Tiempo.rms_movil.v y Tiempo.tiempo_rms. La matriz Tiempo.rms_movil.v representa el valor del rms móvil por cada elemento, mientras que el vector tiempo_rms, unidimensional, que reeplaza al vector Tiempo.tiempo.v para visualizar las gráficas. Le afectan los valores de Default.ventana_rms y Default.solape_rms.

```
import amodal as am

datos = am.Tiempo()
datos.lectura('archivo.xlsx')

# Se cambian los valores por defecto
am.Default.ventana_rms = 1
am.Default.solape_rms = 0.5

datos.get_rms_movil()

print(datos.rms_movil.v)
```

### A.2.10. Tiempo.get_mtvv()

```
get_mtvv(self)
```

Calcula el estadístico mtvv, guardándolo en la variable Tiempo.mtvv.v como un vector unidimensional de tantos elementos como señales existan. Depende del RMS calculado previamente, o en caso de no haber sido calculado previamente, se calcula con los valores por defecto existentes.

```
import amodal as am

datos = am.Tiempo()
datos.lectura('archivo.xlsx')

# Se cambian los valores por defecto
am.Default.ventana_rms = 1
am.Default.solape_rms = 0.5

datos.get_mtvv()

print(datos.mtvv.v)
```

### A.2.11. Tiempo.get_vdv()

```
get_vdv(self)
```

Obtiene como resultado dos matrices, Tiempo.vdv.v y Tiempo.tiempo_vdv. La matriz Tiempo.vdv.v representa el valor del vdv por cada elemento, mientras que el vector tiempo_vdv, unidimensional, que reeplaza al vector Tiempo.tiempo.v para visualizar las gráficas. Le afectan los valores de Default.incremento_vdv.

```
import amodal as am

datos = am.Tiempo()
datos.lectura('archivo.xlsx')

# Se cambian los valores por defecto
am.Default.incremento_vdv = 2

datos.get_vdv()

print(datos.vdv.v)
```

### A.2.12. Tiempo.calcular()

```
calcular(self)
```

Ejecuta las funciones get_rms, get_rms_movil, get_mtvv y get_vdv, explicadas en los apéndices A.2.8,A.2.9 ,A.2.10 y A.2.11 respectivamente. Calcula todos los estadísticos en un sólo comando. Por último, sigue siendo afectado por todos los datos por defecto al igual que las funciones anteriores.

### A.2.13. Tiempo.get_frf()

```
get_frf(self, impact = True)
```

Calcula la función de respuesta en frecuencia de los datos que se contengan en la clase, a partir de los valores por defecto de Default.nfft_frf, Default.nperseg_frf, y Default.noverlap_frf. Por otra parte, el valor nfft determina las frecuencias calculadas en la frf, por lo que se ha diseñado un sistema para que estas frecuencias sean siempre números decimales no periódicos, ya que favorece el cálculo de las frecuencias naturales. La función get_frf crea un objeto en la clase Frecuencia, que se devuelve al usuario:

```
import amodal as am

datos = am.Tiempo()
datos.lectura('archivo.xlsx')

# Se cambian los valores por defecto
am.Defaut.nperseg_frf = 1024
am.Defaut.noverlap_frf = 512
am.Defaut.nfft_frf = 2048
am.preset.threshold_frf = 50

frf = datos.get_frf()
```

```
print(frf.complejo.v)
```

El argumento impact recorta la señal basado en los picos de fuerza y en el número de muestras descrito en nperseg, adaptándola para un test de impacto. En caso de ser False, no trata la señal como una sucesión de impactos de martillo, por lo que también se pueden usar señales provenientes de otros ensayos como shakers.

### A.2.14. Tiempo.resample()

```
resample(self)
```

Transformación que afecta a los valores de Tiempo.tiempo.v, Tiempo.aceleracion.v y Tiempo.fuerza.v, modificando el número de muestras de los mismos mediante fft y transformadas de fourier inversas. Le afecta el valor por defecto Default.samplerate_resample.

```
import amodal as am

datos = am.Tiempo()
datos.lectura('archivo.xlsx')

# Se cambian los valores por defecto
am.Defaut.samplerate_resample = 512

datos.resample()

print(datos.aceleracion.v)
```

### A.2.15. Tiempo.detrend()

```
detrend(self)
```

Función que elimina las tendencias lineales de las señales en Tiempo.aceleracion.v, ya que generan frecuencias cercanas a 0 en las funciones de respuesta en frecuencia. No toma valores por defecto, un ejemplo de su uso es el siguiente:

```
import amodal as am

datos = am.Tiempo()
datos.lectura('archivo.xlsx')

datos.detrend()

print(datos.aceleracion.v)
```

### A.2.16. Tiempo.filtro()

```
filtro(self)
```

Filtro tipo IIR paso bajo. Elimina el ruido de la señal Tiempo.aceleracion.v y Tiempo.fuerza.v. Le afectan los valores por defecto Defaut.orden_filtro y Defaut.frecuencia_filtro, que indican el orden y la frecuencia de corte del filtro.

```
import amodal as am
```

```
datos = am.Tiempo()
datos.lectura('archivo.xlsx')

# Se cambian los valores por defecto
am.Defaut.orden_filtro = 4
am.Defaut.frecuencia_filtro = 200

datos.filtro()

print(datos.aceleracion.v)
```

## A.3. Frecuencia

### A.3.1. Frecuencia.__init__()

```
__init__(self, complejo=None, magnitud=None, fase=None, fase_corr= None,
real= None, imag= None,leyenda=None, unidad=None, frecrate=None,
frecuencia=None, leyenda_frec=None, unidad_frec=None)
```

Declara las variables de la clase, ya sea con un valor (en caso de que sea indicado al llamar a la función) o con el nombre None. En caso de que sólamente se declaren ciertas variables, la función declarará el resto con su valor correcto, siempre que sea posible. Esta función se ejecuta cada vez que se atribuye esta clase a un objecto, de la siguiente forma:

```
import amodal as am
datos = am.Tiempo()
```

En caso de querer introducir los valores al inicio de la función, se deben indicar por lo menos los siguientes valores:

```
datos = am.Frecuencia(complejo=complejo, leyenda=leyenda ,
unidad= unidad, frecuencia= frecuencia)
```

### A.3.2. Frecuencia.get_h5py()

```
get_h5py(self, path)
```

Se genera una base de datos tipo hdf5, donde las variables se guardan en la base de datos según el cuadro A.2.Su uso es equivalente a la función explicada en el apéndice A.2.6.

### A.3.3. Frecuencia.from_h5py()

```
from_h5py(self, path)
```

Se carga una base de datos tipo hdf5, donde las variables se recuperan de la base de datos según el cuadro A.2.Las únicas variables que se calculan son Frecuencia.axis_0 y Frecuencia.axis_1. Su uso es equivalente a la función explicada en el apéndice A.2.7.

### A.3.4. Frecuencia.get_polares()

Cuadro A.2: Estructura de la base de datos de Frecuencia

| Referencia del dataset | Variable que contiene |
|---|---|
| magnitud | self.magnitud.v |
| fase | self.fase.v |
| unidad | self.real.u |
| leyenda | self.real.l |
| fase_corr | self.fase_corr.v |
| real | self.real.v |
| imag | self.imag.v |
| frecrate | self.frecrate |

```
get_polares(self)
```

Prinicipalmente se usa de forma interna. Cambia un valor de la matriz de números complejos Frecuencia.complejo.v con dtype = np.cfloat a dos matrices, Frecuencia.magnitud.v y Frecuencia.fase.v, con el módulo y la fase de los números complejos de la matriz original.

### A.3.5. Frecuencia.get_cartesianas()

```
get_cartesianas(self)
```

Construye una matriz de números complejos, Frecuencia.complejo.v, a partir de su magnitud y fase, Frecuencia.magnitud.v y Frecuencia.fase.v. Principalmente se usa dentro de la clase.

### A.3.6. Frecuencia.get_real_imag()

```
get_real_imag(self)
```

Transforman la matriz de números complejos Frecuencia.complejo.v en dos matrices de números reales, tipo float. La parte real de la matriz compleja se guarda en Frecuencia.real.v, y la parte imaginaria en Frecuencia.imag.v. Esta representación es principalmente indicada para los diagramas de Nyquist.

### A.3.7. Frecuencia.corregir_fase()

```
corregir_fase(self)
```

Un problema común de los diagramas tipo bode es que al representar la fase como un valor entre 0 y $2\pi$ puede haber valores oscilando entre el mínimo y el máximo valor, ensuciando el diagrama con líneas verticales. Para solucionarlo se ha escrito esta función, que transforma el valor de Frecuencia.fase.v en el vector Frecuencia.fase_corr.v. El nuevo vector es una línea continua, que representa de forma correcta la variable.

### A.3.8. Frecuencia.get_frecrate()

```
get_frecrate(self)
```

Suele ejecutarse por otras funciones del programa. Su función es declarar la variable Frecuencia.frecrate, que recoge la distancia, en $\frac{1}{Hz}$, entre las frecuencias que evalúan las fft.

### A.3.9. Frecuencia.copia()

```
copia(self)
```

En python no se puede hacer una réplica exacta de un objeto, esta funcion sirve para generar una réplica exacta del objeto Frecuencia. Es usado de la siguiente forma:

```
import amodal as am
datosoriginales = am.Frecuencia()
datoscopia = datosoriginales.copia()
```

### A.3.10. Frecuencia.integrar()

```
integrar(self)
```

Se integra la señal, modificando la fase y la magnitud de la misma. Ésta función nos permite transformar la señal de intertancia en movilidad o receptancia. Se debe tener en cuenta que la función convierte los datos originales, por lo que es recomendable aplicar esta función sobre una copia de la original.

## A.4. Modal

### A.4.1. Modal.__init__()

```
__init__(self, data, rango_frec)
```

Declara las variables de la clase, donde es necesaria la declaración de la función de respuesta en frecuencia sobre la que calculará los modos, y una matriz tridimensional donde se indican las estimaciones de los rangos de frecuencia. La función de respuesta en frecuencia debe estar contenida en un objeto de la clase Frecuencia, ya que se apoya en esa misma estructura de datos.

```
import amodal as am
modos = am.Modal(frf,rango_de_frecuencia)
```

### A.4.2. Modal.frec_to_place()

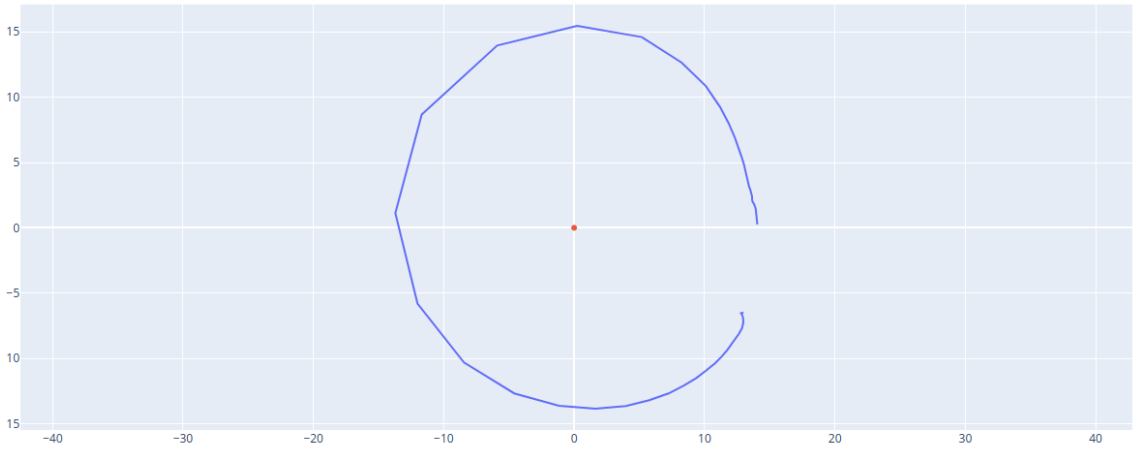
```
frec_to_place(self)
```

La función convierte el listado de frecuencias indicado en Modal.__init__() a un listado de las posiciones de dichas frecuencias en la señal de frecuencias indicada en la instancia Frecuencia.frecuencia.v . El resultado se guarda en la variable self.pos_rango_frec .

### A.4.3. Modal.peak_picking()

```
peak_picking(self)
```

Figura A.1: Diagrama Nyquist centrado de un modo




Efectúa el método peak picking en los datos introducidos en el objeto, con los rangos de frecuencias indicados. Los valores son guardados en los vectores Modal.frec_resonancia y Modal.amortiguamiento , donde se retorna una frecuencia propia y un factor de amortiguamiento por cada modo y señal de entrada. En caso de error se retorna el valor 0 para ambos casos.

### A.4.4. Modal.circle_fit()

```
circle_fit(self,debug_sp = False, debug_nyquist= False, debug_psi = False)
```

Efectúa el método circle fit en los datos introducidos en el objeto, con los rangos de frecuencias indicados. Los valores son guardados en los vectores Modal.frec_resonancia y Modal.amortiguamiento , donde se retorna una frecuencia propia y un factor de amortiguamiento por cada modo y señal de entrada. En caso de error se retorna el valor 0 para ambos casos. Esta función cuenta con la funcionalidad de retornar las gráficas de la variable sp (el ángulo de cada punto de la frf, figura A.2), el diagrama Nyquist del rango de frecuencias que se ha especificado (figura A.3), y una representación gráfica del factor de amortiguamiento obtenido en cada punto (figura A.3).

### A.4.5. Modal.promedio()

```
promedio(self)
```

Calcula la media de los amortiguamientos y frecuencias naturales para cada modo, discrimina los valores iguales a 0, ya que corresponden a un error. La media de la frecuencia propia y el factor de amortiguamiento se guardan en las variables Modal.frec_resonancia_media y Modal.amortiguamiento_medio, respectivamente.

## A.5. graficas

El módulo graficas se ha realizado como una librería diferente, pero que funciona con los mismos principios y organización de clases descrita en la librería amodal. Se puede importar en el programa de la siguiente forma:

```
import graficas
```

Figura A.2: Diagrama del ángulo entre puntos consecutivos y sus derivadas

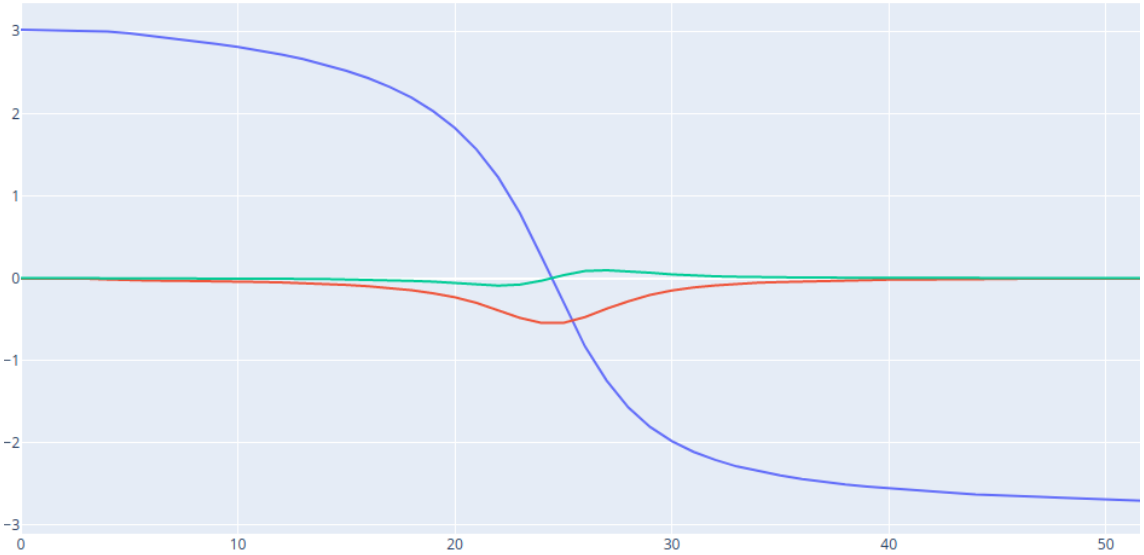



Figura A.3: Diagrama del coeficiente de amortiguamiento entre un punto n y la frecuencia propia

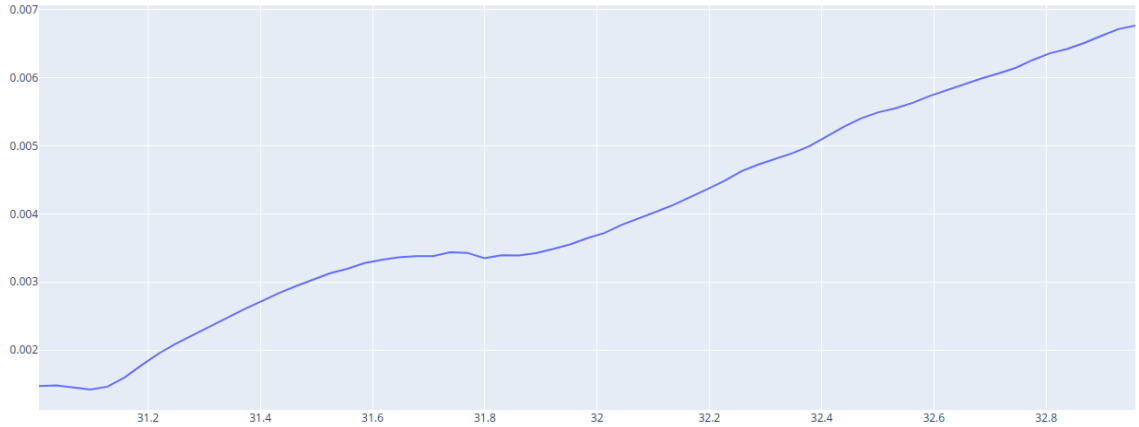
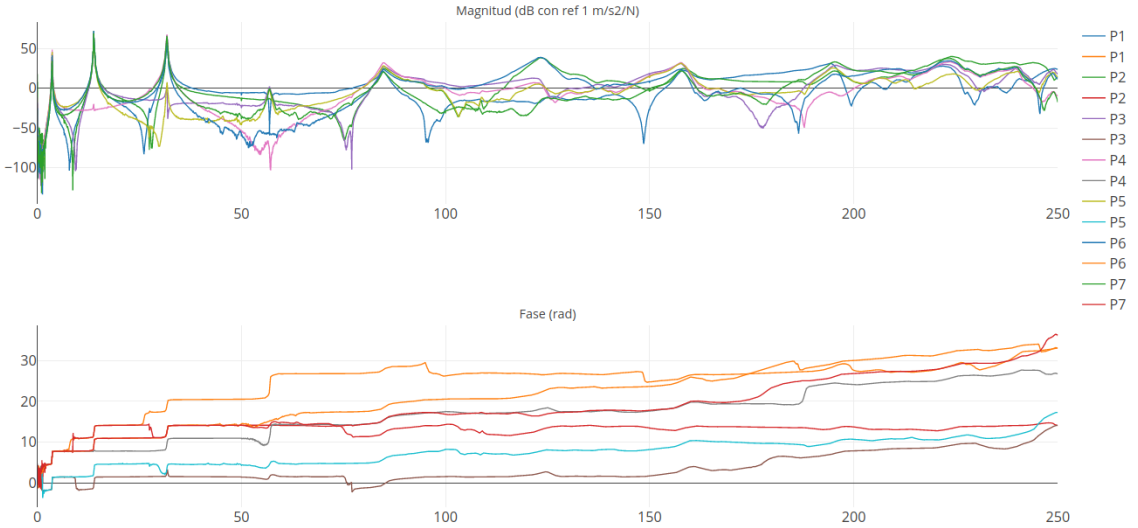


Figura A.4: Diagrama bode




### A.5.1. graficas.grafica()

```
grafica(tiempo, *args)
```

Exporta una gráfica con el tiempo, donde el primer argumento es la variable tiempo de la clase Tiempo, que se representará en el eje de abscisas. Se pueden incluir tantas variables como se deseen en el argumento *args: vdv, rms_movil, aceleración... Sin embargo, si se seleccionan varias variables no se representarán las unidades de las mismas. El valor que devuelve la función es un elemento de plotly, que se puede exportar a diversos formatos con la librería adecuada.

### A.5.2. graficas.grafica_html()

```
grafica_html(tiempo, *args, arch = None)
```

Tiene el mismo funcionamiento que la función anterior, pero exporta la gráfica directamente a un archivo html, que se puede abrir desde un navegador. Si arch = None, el nombre será grafica.html, en caso contrario será el nombre del fichero creado.

### A.5.3. graficas.bode()

```
bode(data)
```

La variable data es un objeto tipo Frecuencia. Retorna una figura de plotly, con el diagrama bode, similar a la indicada en la figura A.4.

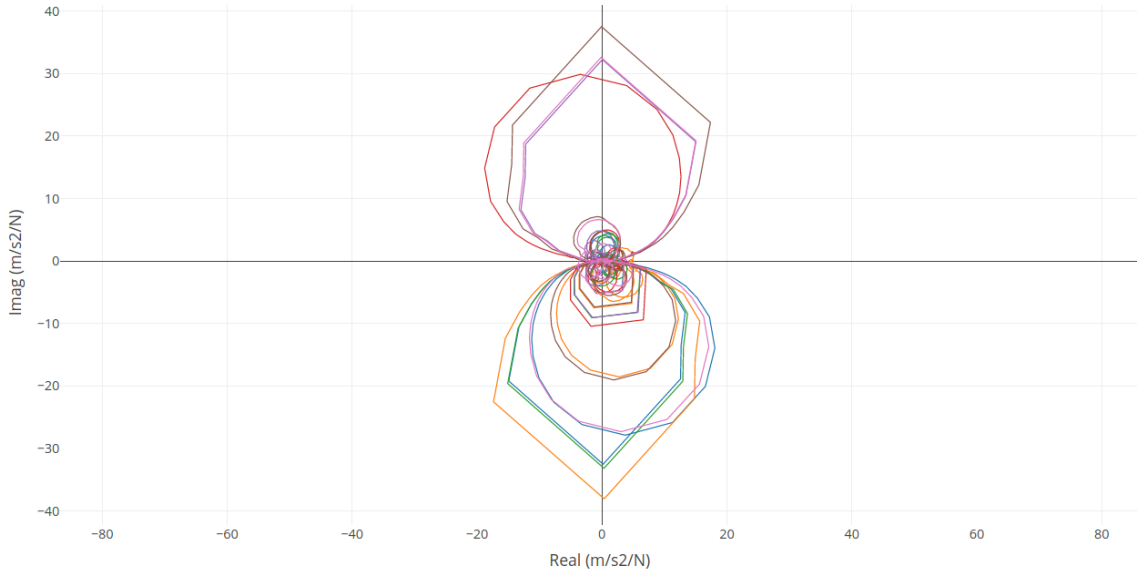
### A.5.4. graficas.bode_html()

```
bode_html(data, arch = None)
```

Funcionamiento idéntico a la función anterior. Si arch = None, el nombre será grafica.html, en caso contrario será el nombre del fichero creado.

### A.5.5. graficas.nyquist()

Figura A.5: Diagrama Nyquist




```
nyquist(data)
```

La variable data es un objeto tipo Frecuencia. Retorna una figura de plotly, con el diagrama Nyquist, similar a la indicada en la **??**.

### A.5.6. graficas.nyquist_html()

```
nyquist_html(data, arch = None)
```

Funcionamiento idéntico a la función anterior. Si arch = None, el nombre será grafica.html, en caso contrario será el nombre del fichero creado.

# Apéndice B

# Guía de instalación

## B.1. Instalación de python

Todo el trabajo se ha desarrollado en python 3, el código adicional que se ha expuesto en css y html es ejecutado en el navegador web, disponible en la mayoría de dispositivos. Para comenzar la instalación de python se debe descargar un ejecutable para instalar el intérprete de python, junto con las librerías clave. Se puede descargar en todos los sistemas desde la página web, ,https://www.python.org/downloads/. Al elegir la versión, se puede instalar cualquier versión de python 3 igual o superior a 3.6. Para versiones inferiores no se asegura un funcionamiento correcto. En linux, para las versiones de debian se puede usar el siguiente comando:

```
sudo apt install python3.6
```

Otra alternativa es instalar anaconda (https://docs.anaconda.com/anaconda/install/), un paquete que incluye diversas utilidades para el uso de python en el ámbito científico. Tiene instalados muchos paquetes que se usarán posteriormente.

## B.2. Instalación de pip3

El segundo paso para la instalación es instalar los paquetes y las librerías de las que se beneficia el programa. Para ello se ha realizado un documento de texto, requierements.txt, que determina los paquetes que se han usado en la aplicación. En python los paquetes se pueden instalar a partir de pip o pip3, un gestor de paquetes para indicapython. En primer lugar, se debe comprobar si está instalado pip3, para ello se abre una ventana de comandos y se ejecuta:

```
pip3 --version
```

Si el comando es detectado por el sistema, se puede continuar al siguiente paso. Por otra parte, en caso de haber instalado python por medio de anaconda, este mismo tiene otro gestor de paquetes propio y no es necesario instalar pip3.

Para instalar pip3 se refiere a la propia documentación del programa: https://pip.pypa.io/en/stable/installing/.

## B.3. Instalación de paquetes

En el fichero requirements.txt se indican las librerías a instalar. Pip3 incluye una opción para instalar los paquetes de forma directa. Para ello se abre una ventana de comandos en la carpeta donde esté el archivo requirements.txt, y en caso de tener instalado pip3, con la versión original de python, se ejecuta el siguiente comando:

```
pip3 install -r requirements.txt
```

En el caso de instalar anaconda, la instalación se realizará con el siguiente comando:

```
conda install --file requirements.txt
```

Finalmente, si los métodos anteriores no funcionan, se puede realizar manualmente la instalación de cada paquete:

```
pip3 install <nombre del paquete>
```

ó

```
conda install <nombre del paquete>
```

En caso de tener un conocimiento más amplio de python, se recomienda el uso de entornos virtuales para que la instalación no afecte al resto del sistema.

## B.4. Ejecutar el programa

El programa consta de, por una parte los ejemplos, y por otra parte la aplicación web realizada con Flask. Para el primer caso se debe abrir el ejemplo que se desee ejecutar: en caso de haber instalado un entorno de desarrollo se abre y se ejecuta desde el mismo. En caso de no tener instalado un entorno de desarrollo, se ejecuta el siguiente comando desde la carpeta correspondiente:

```
python3 nombre_del_ejemplo.py
```

Para ejecutar la aplicación web, nos dirigimos a la carpeta principal, donde se encuentra el fichero FlaskApp.py, y se abre con el mismo comando:

```
python3 FlaskApp.py
```

El framework creará entonces un servidor local, dando la ruta del mismo:

```
 * Serving Flask app "FlaskApp" (lazy loading)
* Environment: production
WARNING: This is a development server. Do not use it in a
production deployment.
Use a production WSGI server instead.
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 262-033-365
```

Finalmente podremos acceder a http://127.0.0.1:5000/ desde cualquier servidor local.

# Bibliografía

[1] B. Elliot, "Tacoma narrows bridge collapsed," 1940. [Online]. Available: https://commons.wikimedia.org/wiki/File:Tacoma_narrows_bridge_collapsed.jpg

[2] M. Peel, "Millennium bridge 2009-1," 2009. [Online]. Available: https://commons.wikimedia.org/wiki/File:Millennium_Bridge_2009-1.jpg

[3] ISO, "Human response to vibration — measuring instrumentation — part 1: General purpose vibration meters," International Organization for Standardization, Geneva, Switzerland, ISO 8041-1:2017, 2017.

[4] A. Oppenheim, R. Schafer, and R. Schafer, *Discrete-time Signal Processing*, ser. Prentice-Hall signal processing series. Pearson, 2010. [Online]. Available: https://books.google.es/books?id=mYsoAQAAMAAJ

[5] P. D. Welch, "The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms," *IEEE Trans. Audio & Electroacoust*, vol. 15, pp. 70–73, Jan. 1967. [Online]. Available: https://ui.adsabs.harvard.edu/abs/1967ITAE...15...70W

[6] Z. Fu and J. He, *Modal Analysis*. Elsevier Science, 2001. [Online]. Available: https://books.google.es/books?id=ElwhqUtJUj8C

[7] D. Ewins, *Modal Testing: Theory and Practice*, ser. Engineering dynamics series. Research Studies Press, 1984. [Online]. Available: https://books.google.es/books?id=hBtEAQAAIAAJ

[8] N. Maia and J. Silva, *Theoretical and Experimental Modal Analysis*, ser. Engineering dynamics series. Research Studies Press, 1997. [Online]. Available: https://books.google.es/books?id=KWxwQgAACAAJ

[9] C. R. Harris and et al., "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[10] P. Virtanen and et al., "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020. [Online]. Available: https://rdcu.be/b08Wh

[11] M. Martín Jiménez, "Identificación dinámica, calibrado computacional y simulación de tránsitos aplicados a un modelo a escala de un puente," 2020. [Online]. Available: http://uvadoc.uva.es/handle/10324/41408